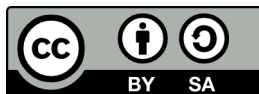**Hands-On Exercises for**

Embedded Android

v. 2015.11

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.

<u>Day 1</u>

<u>Class Preparation</u>
Notes:
- The list of initial prerequisites for Android development changes over time
- The latest version is at: http://developer.android.com/sdk/installing.html

1. Install Android Studio:
https://developer.android.com/tools/studio/index.html
This URL should give you access to all the instructions needed to get this installed and working.

2. Install the required packages to build Android:
See the "Installing required packages" instructions for the relevant Ubuntu version here:
http://source.android.com/source/initializing.html

3. Fetch the AOSP:
We'll be using 5.0/Lollipop. Note that the new lines that start with ">" are printed by the shell. Adding a ">" to the input you type will break the following commands. Assuming you have a directory called "android" in your home directory:
```
$ cd ~/android
$ mkdir aosp-5.0.2_r1
$ cd aosp-5.0.2_r1
$ repo init -u https://android.googlesource.com/platform/manifest \
> -b android-5.0.2_r1
$ repo sync
```
Note that fetching the sources is a fairly lengthy process.

4. Fetch the binaries for the Nexus 7/flo for the release we'll be using:
https://developers.google.com/android/nexus/drivers#flolrx22g
You will get a ".tgz" file for each of the packages. Download those files into the root of the AOSP and extract them:
```
$ tar xvzf asus-flo-lrx22g-c4c20146.tgz
$ tar xvzf broadcom-flo-lrx22g-10154061.tgz
$ tar xvzf qcom-flo-lrx22g-dfa10270.tgz
$ ./extract-asus-flo.sh
$ ./extract-broadcom-flo.sh
$ ./extract-qcom-flo.sh
```
You will need to read through the license and type "I ACCEPT" at the end for the binaries to extract.

5. Get the kernel and associated toolchain:

a. Create directory for working on kernel -- assuming we're at the top-level of the AOSP:

```
$ mkdir kernel; cd kernel
```

b. Get gcc for the kernel -- the built-in toolchain shipped with 5.0.2_r1 CANNOT be used to build the flo kernel; go figure:

```
$ mkdir gcc; cd gcc
$ git clone
https://android.googlesource.com/platform/prebuilts/gcc/linux-
x86/arm/arm-eabi-4.6
```

c. Get the kernel itself:

```
$ cd ..
$ git clone https://android.googlesource.com/kernel/msm.git
$ cd msm
$ git checkout android-msm-flo-3.4-lollipop-release
```

d. We'll reconfigure and build the kernel during later exercises.

6. Build the AOSP:

```
$ cd ~/android/aosp-5.0.2_r1
$ . build/envsetup.sh
$ lunch aosp_flo-eng
$ make -j8
```


7. Ensure that the build finishes successfully.

8. Flash the previous build onto your device by:
   a) Putting your device in fastboot mode by booting it while holding the "volume down" button.
   b) Using the "fastboot" command from the AOSP's root directory:

```
$ source build/envsetup.sh
$ lunch aosp_flo-eng
$ fastboot flashall
```


9. Go to Settings->"About tablet" and check that the build number signature has a date matching your build date.

Kernel Basics

1. Check kernel version on the device ("cat /proc/version")

2. Reconfigure the kernel to support modules and module unloading. You don't need to enable forced module unloading. Here's a reminder on how to get the kernel configuration that uses the default "flo" configuration as its starting point:

```
$ cd kernel/msm/
$ make ARCH=arm flo_defconfig
$ make ARCH=arm menuconfig
```

3. Rebuild, reflash, reboot. To rebuild the kernel, you need to do the following from the "kernel/msm/" directory -- note the "..." which you'll have to replace with the path to your AOSP:

```
$ export TARGET=/home/.../kernel/gcc/arm-eabi-4.6/bin/arm-eabi
$ make ARCH=arm CROSS_COMPILE=${TARGET}-
```

In order for the AOSP to take your newly built kernel into account, you'll need to do this -- the following assuming that you've run the appropriate ". build/envsetup.sh" and "lunch" from the toplevel:

```
$ croot
$ cd device/asus/flo-kernel/
$ cp kernel kernel-orig
$ croot
$ cp kernel/msm/arch/arm/boot/zImage device/asus/flo-kernel/kernel
$ make -j8
```

4. Check the new kernel's version ("cat /proc/version")

AOSP Fix ups

1. Copy the following file -- the bit.ly link is just a URL shortnening service:

      http://www.opersys.com/downloads/IOpersysService.aidl

      http://bit.ly/11CWgUM

to this location in your AOSP:

      [aosp]/frameworks/base/core/java/android/os/

2. Add the highlighted line to your [aosp]/frameworks/base/Android.mk :

```
core/java/android/os/IPowerManager.aidl \
core/java/android/os/IOpersysService.aidl \
core/java/android/os/IRemoteCallback.aidl \
```

3. Modify your [aosp]/system/core/include/private/android_filesystem_config.h to add the highlighted lines:

```
{ 00750, AID_ROOT,      AID_SHELL,      0, "sbin/*" },
{ 00755, AID_ROOT,      AID_ROOT,       0, "bin/*" },
{ 00755, AID_ROOT,      AID_ROOT,       0, "system/glibc/lib/*" },
{ 00755, AID_ROOT,      AID_ROOT,       0, "system/glibc/bin/*" },
{ 00750, AID_ROOT,      AID_SHELL,      0, "init*" },
```

4. Rebuild your AOSP and reflash your device.

Day 2

Linux root filesystem

1. Get and install the Opersys Embedded Linux Workspace:
>   For ARM:
>   http://www.opersys.com/downloads/bbone-workspace.tar.bz2
>
>   For x86-64:
>   http://bit.ly/1MPEya9
>   or
>   https://drive.google.com/file/d/0B0dp197y8Ga7VVZjSmEzak9aSFE

2. Customize the workspace script in the extracted directory to match your own paths.

3. Create yourself a root filesystem that includes the glibc shared libraries as in slides 139 **and** 145, save for the "strip" command. Note that you'll need to replace BBONE_WS_DIR with the location where you installed the Workspace in exercise #1 of this section.

4. Run the the environment script:
>   For ARM:
>   `$ . devbb`
>
>   For x86:
>   `$ . devx86`

5. Make sure you have libncurses5-dev installed:
>   `$ sudo apt-get install libncurses5-dev`

6. For BusyBox:
   - Extract the tarball that's in sysapps/
   - Configure BusyBox per the slides
   - Build BusyBox per the slides
   - Install BusyBox per the slides

Native Android user-space

Preamble:
   - For every exercise, it's implied that you should build the AOSP and reflash your device 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/flo/ directory containing a directory and a corresponding image for:
  - The boot image (boot.img)
  - 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 system image. You will need to:

- Create [aosp]/rootfs-glibc
- Copy your glibc rootfs (from bbone-ws) under [aosp]/rootfs-glibc:
  ```
  $ cp -a ${PRJROOT}/rootfs [aosp]/rootfs-glibc
  ```
- Download this file as [aosp]/rootfs-glibc/Android.mk

  http://www.opersys.com/downloads/glibcroot-Android-sys.mk

  http://bit.ly/T4H9V9
- Add "rootfs-glibc" to [aosp] build/core/legacy_prebuilts.mk
- Add the marked-up lines to [aosp]/system/core/rootdir/init.rc:
  ```
  # Backward compatibility
      symlink /system/etc /etc
      symlink /sys/kernel/debug /d

      symlink /system/glibc/bin /bin
      symlink /system/glibc/lib /lib
      symlink /system/glibc/lib /lib64
  ```

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
          class main
          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. Using the client and server programs from http://www.linuxhowtos.org/C_C++/socket.htm, compile the server against Bionic and the clieng against glibc.

To have a the C server build against Bionic:
- Add it as [aosp]/frameworks/base/cmds/server
- Make sure "server" is added to [aosp]/build/target/product/core.mk
- Make sure you have an appropriate Android.mk within [aosp]/frameworks/base/cmds/server to get your app to build. Here's a sample:

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

LOCAL_SRC_FILES:= \
        server.c

LOCAL_SHARED_LIBRARIES := \
        libcutils

LOCAL_MODULE:= server

include $(BUILD_EXECUTABLE)
```

To have the C client build against glibc:
- Place the C client into the [aosp]/rootfs-glibc/rootfs/bin/
- Use the toolchain that's part of the glibc-based rootfs we had earlier:
  - For ARM:
```
/.../bbone-ws/tools/arm-unknown-linux-gnueabi/bin/arm-unknown-linux-gnueabi-gcc   -o   client
client.c
```
  - For x86
```
/.../custom-x86/tools/x86_64-unknown-linux-gnu/bin/x86_64-unknown-linux-gnu-gcc   -o   client
client.c
```

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.
- You can use the templates from the following tarball to copy-n-paste: http://opersys.com/downloads/systemserver-samples-141203.tar.bz2.
- 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.
- You'll need to modify the selinux policies in order to allow your system service to be registered. To do so, modify the "external/sepolicy/service_contexts" file to add this entry:
```
opersys                              u:object_r:system_server_service:s0
```

4. Build the AOSP and use "logcat" to check that your service is started upon system startup.

5. Use "`service call`" to invoke your system service from the command line and check the logcat to see that it's been properly called.

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. We will use the following driver for the exercises:

> http://www.opersys.com/downloads/circular-driver-111207.tar.bz2

This drivers implements a circular buffer over the read/write file-ops. Refer to LDD3 slide-set for both code examples and a Makefile. The driver doesn't implement blocking/waking semantics. When there's no content available, it returns 0 bytes. Have a look at the driver at the following address as another driver exxample: http://tldp.org/LDP/lkmpg/2.6/html/x569.html

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

For the Nexus 7:

- Put the driver's directory in [aosp]/device/asus/flo/
- This driver uses misc_register() instead of register_chrdev() as it will register your char dev <u>and</u> fire off a hotplug event. See: http://www.linuxjournal.com/article/2920
- Building the module:
  ```
  $ export TARGET=/home/.../kernel/gcc/arm-eabi-4.6/bin/arm-eabi
  $ make ARCH=arm CROSS_COMPILE=${TARGET}-
  ```
- To have the circular-driver.ko included in the final image, edit the device/asus/flo/device-common.mk to add the following:
  PRODUCT_COPY_FILES += \
  > device/asus/flo/circular-driver/circular-char.ko:system/modules/circular-char.ko
- To have the module loaded at startup, modified an init.rc file to get it loaded. The board-specific file in the case of the Nexus 7 is device/asus/flo/init.flo.rc:
  ```
  on post-fs
      insmod /system/modules/circular-char.ko
  ```
- To have ueventd create the proper entry in /dev at runtime, you'll need to modify a ueventd.rc to have a proper entry for the module. You can modify the device/asus/flo/ueventd.flo.rc file to add:
  ```
  /dev/circchar          0666  system      system
  ```
- To allow init to load the module at startup, you'll need to modify the selinux policies to allow the loading of the module. To do so, add "sys_module" to the list of system calls allowed by init in the "external/sepolicy/init.te" file:
  ```
  allow init self:capability { sys_rawio mknod sys_module };
  ```
- To allow the file to be accessible by the system server, you'll need to modify the selinux policies to have a rule for the /dev entry. To do so, add "/dev/circchar" to the list of files allowed by the system server in the "external/sepolicy/file_contexts" file:
  ```
  /dev/circchar           u:object_r:tty_device:s0
  ```

For the MinnowBoardMax:

- Put the driver's directory in [aosp]/device/intel/baytrail/minnow64_p/
- This driver uses misc_register() instead of register_chrdev() as it will register your char dev <u>and</u> fire off a hotplug event. See: http://www.linuxjournal.com/article/2920
- Building the module:
  ```
  $  export  TARGET=/home/.../[aosp]/prebuilts/gcc/linux-x86/x86/x86_64-linux-
  android-4.8/bin/x86_64-linux-android
  $ make ARCH=arm CROSS_COMPILE=${TARGET}-
  ```
- To have the circular-driver.ko included in the final image, edit the device/intel/baytrail/minnow64_p/device.mk to add the following:
  PRODUCT_COPY_FILES += \
  ```
          device/intel/baytrail/minnow_64p/circular-driver/circular-char.ko:system/modules/circular-char.ko
  ```
- To have the module loaded at startup, modified an init.rc file to get it loaded. The board-specific   file   in   the   case   of   the   MinnowBoardMax   is device/intel/baytrail/minnow64_p/init.rc:
  ```
  on post-fs
      insmod /system/modules/circular-char.ko
  ```
- To have ueventd create the proper entry in /dev at runtime, you'll need to modify a ueventd.rc to have a proper entry for the module.   You can modify the device/intel/baytrail/minnow64_p/ueventd.rc file to add:
  ```
  /dev/circchar          0666  system      system
  ```
- To allow init to load the module at startup, you'll need to modify the selinux policies to allow the loading of the module. To do so, add "sys_module" to the list of system calls allowed by init in the "external/sepolicy/init.te" file:
  ```
  allow init self:capability { sys_rawio mknod sys_module };
  ```
- To allow the file to be accessible by the system server, you'll need to modify the selinux policies to have a rule for the /dev entry. To do so, add "/dev/circchar" to the list of files allowed by the system server in the "external/sepolicy/file_contexts" file:
  ```
  /dev/circchar            u:object_r:tty_device:s0
  ```

For the emulator:
- Put the driver's directory in [aosp]/device/generic/goldfish/
- This driver uses misc_register() instead of register_chrdev() as it will register your char dev <u>and</u> fire off a hotplug event. See: http://www.linuxjournal.com/article/2920
- Building the module:
  ```
  $ cd device/generic/goldfish/circular-driver
  $ export TARGET=/home/.../kernel/gcc/arm-eabi-4.6/bin/arm-eabi
  $ make ARCH=arm CROSS_COMPILE=${TARGET}-
  ```
- To have the circular-driver.ko included in the final image, edit the build/target/product/generic_no_telephony.mk to add the following:
  PRODUCT_COPY_FILES += \
  ```
      ...
        device/generic/goldfish/circular-driver/circular-char.ko:system/modules/circular-char.ko
  ```
- To have the module loaded at startup, modified an init.rc file to get it loaded. The

---

board-specific file in the case of the emulator is device/generic/goldfish/init.goldfish.rc:
```
on post-fs
    insmod /system/modules/circular-char.ko
```
- To have ueventd create the proper entry in /dev at runtime, you'll need to modify a ueventd.rc to have a proper entry for the module. You can modify the device/generic/goldfish/ueventd.goldfish.rc file to add:
```
/dev/circchar        0666  system     system
```
- To allow init to load the module at startup, you'll need to modify the selinux policies to allow the loading of the module. To do so, add "sys_module" to the list of system calls allowed by init in the "external/sepolicy/init.te" file:
```
allow init self:capability { sys_rawio mknod sys_module };
```
- To allow the file to be accessible by the system server, you'll need to modify the selinux policies to have a rule for the /dev entry. To do so, add "/dev/circchar" to the list of files allowed by the system server in the "external/sepolicy/file_contexts" file:
```
/dev/circchar            u:object_r:tty_device:s0
```

2. If you haven't already modified the build system and the startup configuration files to load the driver automatically as above, load your driver into your custom-built kernel on your emulator or device using 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.

3. 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. We will use the code from the following tarballs for this hands-on session:
    http://opersys.com/downloads/libhw-opersys-flo-151027.tar.bz2

Otherwise:
    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 libopersyshw.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.

2. 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.

3. 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 -- DO NOT DO

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

2. Configure Android Studio to use your newly-built SDK

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

Stack Extension Exercises

1. 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
Refer to the Linux Device Drivers book linked to earlier for a reference on how to implement ioctl() calls. Note that you'll need to implement unlocked_ioctl() instead of ioctl() in the file operations struct.

2. 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

3. 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.

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

Platform Internals Exercises
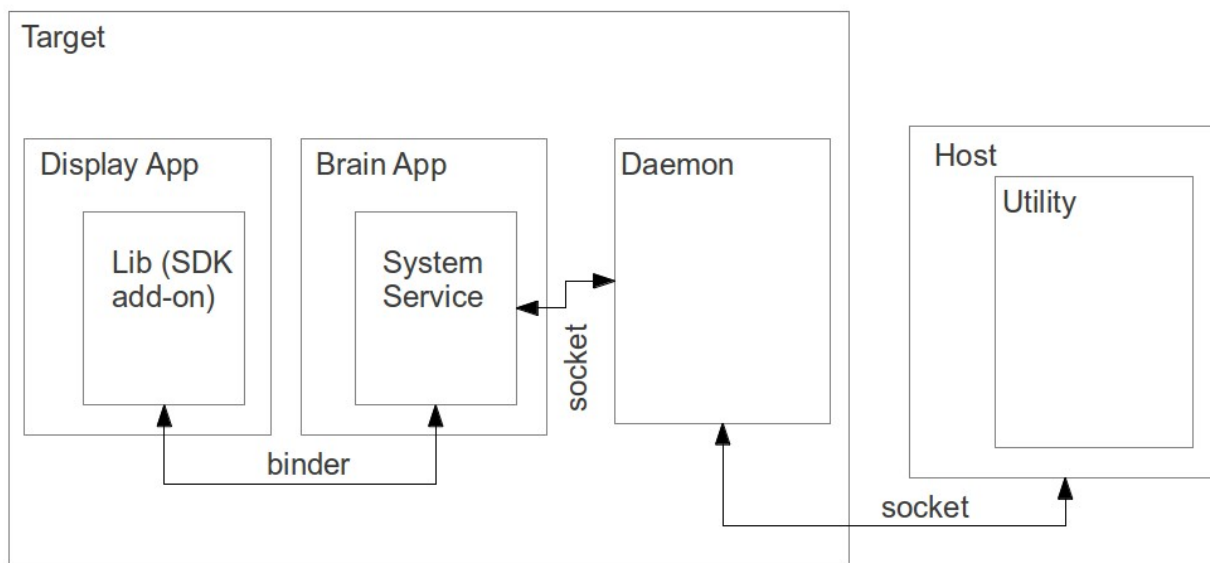
1. Have the Brain App be a standalone app in packages/apps (in the case of the emulator) or device/ti/beaglebone/ (in the case of the Beaglebone.) To make sure the app is part of PRODUCT_PACKAGES, you'll need to modify build/target/product/generic_no_telephony.mk

(in the case of the emulator) and device/ti/beaglebone/beaglebone.mk (in the case of the emulator). Follow the Phone app example on how to make your app a standalone app that has a system service. You can use your existing system service as a basis for your new system service in the Brain App.

2. The system service in the brain app should maintain an internal buffer which is fed from the buffer your circular driver has; more on this below. The API of the system service in the brain app should be:

> String read(), which returns what's in its buffer
> int write(), which writes a string to the system service's internal buffer
> 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 system service
> int getWriteStat(), which returns the number of write() calls made to the system service
> void setBufferToChar(char), which sets the buffer's content to a char value
> void enableIntentOnContent(), makes it so that system service sends an intent when there's new data
> void disableIntentOnContent(), disables the intent on data
> void registerCallbackInterface(), to allow an app to register a callback remote binder interface
> void unregisterCallbackInterface(), to deregister the remote callback interface

3. Create a thread in your system service that periodically reads the driver's buffer to fill the buffer maintained by the system service. Instead of using the HAL to read from the driver, put the code that was in opersyshw_qemu.c (in the case of the emulator) or opersyshw_beaglebone.c (in the case of the BeagleBone) inside a C library that is called through JNI by your system service. You'll have to get that library loaded by your app. Have a look at the Hello JNI sample in the NDK.

4. Extend the Status Bar app to have a new overlay window such as the one displayed by LoadAverageService.java; copy the latter and start from there. Have your new addition connect to the system service in the Brain App and register a callback interface to allow the system service to call the service back with new data added to the buffer. When new data comes in, have your Status Bar extension display the new data as an overlay as the example in the previous page shows. Have a look at the registerStatusBar() call part of the IStatusBarService.aidl in frameworks/base/core/java/com/android/internal/statusbar/ for an example of how a system service can allow a caller to register a callback interface.

5. Create a library that exposes the system service's API through a Java library that is exposed to regular applications using an SDK add-on. Import the SDK add-on to your SDK and create a sample application with Android Studio that can read() and write() to your system service through the add-on.

6. Create a native daemon that listens to socket connections from the system service and also receives remote connections from a host utility. The commands received from the system

---

service should be (assuming a text based interface like the one between installd and the Package Manager):

"get_host_message", to retrieve any messages buffered in the daemon from the host utility

"new_data_in_buffer", for sending new messages added to the system service' buffer

The system service should connect to the daemon through a Unix domain socket like the ones in /dev/socket/ while the host utility will connect to the daemon through a classic IP port; pick a port number of your choice. You'll want to extend the thread created earlier in the system service to periodically poll the daemon for host messages.

7. Create a utility for using on the host to:

a. read new messages available from the daemon on the target

b. push new messages to the daemon on the target

At the end of this, you should be able to push a new message from the utility on the host and have it display to the screen through the extension to the status bar.

Extra Exercises

1. Mark Zygote as "disabled" in init.rc and start it by hand after boot using the "start" command.

2. Modify the bootloader parameters to have the system boot remotely from the network.

   a. Create a "uEnv.txt" file on the "boot" partition of the micro-SD card to provide appropriate values for the "net_boot" command to operate properly. You'll likely need to set "serverip" to your host's IP address. You'll likely also want to set "rootpath" to the location of your rootfs. To load environment variables from "uEnv.txt", type:

   U-Boot# mmc rescan

   U-Boot# run loadbootenv

   U-Boot# run importbootenv

   b. Create a U-Boot image of the kernel built earlier

   c. Configure your host for serving TFTP requests

   d. Use TFTP to download image to target

   e. Boot with image

   f. Configure your host and your target so that the target's configuration is obtained at boot time via DHCP and the rootfs is mounted on NFS.

3. Extend the "svc" command in frameworks/base/svc to make it capable of talking to the Opersys system service, thereby allowing you to communicate with that system service straight from the command-line. "svc" already implements commands for a couple of system services (PowerManager, ConnectivityManager, TelephonyManager, and WifiManager).

4. Create app w/ EditText and Button that sends text entered in the text box all the way down to the driver.

5. Create an app that acts as the home screen. You'll to have an activity with the following set of filters:

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.HOME"/>
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Have a look at development/samples/Home for an example basic home app. You're not expected to have a full app allowing the starting of other apps. Just make sure you can get a "HelloWorld" to kick in right after boot up.

6. Implement a C-built system service and a corresponding C command-line utility that use Binder to communicate. Have a look at:

frameworks/base/libs/surfaceflinger_client/ISurfaceComposer.cpp

frameworks/base/services/surfaceflinger/SurfaceFlinger.cpp

frameworks/base/core/jni/android_view_Surface.cpp

Android on the BeagleBone

MicroSD Programming:
1. Extract BeagleBone.tar.gz
2. Plug the MicroSD card into the MicroSD-USB connector and into your host
3. Program the MicroSD card (you need to use the "dmesg" command to check the exact name under which the device appears - **WARNING: if you get this wrong you could destroy your host's Ubuntu installation**)

        $ sudo ./mkmmc-android.sh /dev/sd???

BeagleBone Recognition
1. Modify or add an /etc/udev/rules.d/51-android.rules with this content:

        SUBSYSTEM=="usb", SYSFS{idVendor}=="18d1", MODE="0666"

Some more recent versions of Ubuntu (12.04) may require the use of this instead:

        SUBSYSTEM=="usb", ATTR{idVendor}=="18d1", MODE="0666"

2. Add a /etc/udev/rules.d/73-beaglebone.rules (also here: http://www.opersys.com/downloads/73-beaglebone.rules):

        ACTION=="add", SUBSYSTEM=="usb", ENV{DEVTYPE}=="usb_interface", \
            ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010", \
            DRIVER=="", RUN+="/sbin/modprobe -b ftdi_sio"

        ACTION=="add", SUBSYSTEM=="drivers", \
            ENV{DEVPATH}=="/bus/usb-serial/drivers/ftdi_sio", \
            ATTR{new_id}="0403 6010"

        ACTION=="add", KERNEL=="ttyUSB*", \
             ATTRS{interface}=="BeagleBone/XDS100V2", \
            ATTRS{bInterfaceNumber}=="00", \
             SYMLINK+="beaglebone-jtag"

        ACTION=="add", KERNEL=="ttyUSB*", \
             ATTRS{interface}=="BeagleBone/XDS100V2", \
            ATTRS{bInterfaceNumber}=="01", \
             SYMLINK+="beaglebone-serial"

3. Force udev to reread its rules:

        $ sudo udevadm control --reload-rules

4. Get minicom

        $ sudo apt-get install minicom

5. Configure minicom

        $ sudo minicom -s

"Serial port setup"
            /dev/beaglebone-serial
            115200 8N1 no hardware flow control
    ESC
        "Save setup as dfl"
        "Exit from minicom"

BeagleBone Bootup
1. Plug Bone through barrel connector to your host
2. Plug Bone mini USB to you host
3. Start minicom
        $ minicom
        You should now see the kernel start up and Android's boot sequence

Android Visualization
1. Connect to target using ADB (if it's not in your path already, go to the AOSP's root directory
and type ". build/envsetup.sh" and then "lunch"):
        $ adb shell
2. Start the VNC server
        # androidvncserver &
3. Tunnel VNC traffic over ADB
        $ adb forward tcp:5901 tcp:5901
4. Connect to target over VNC
        $ vinagre &
        Connect->
                Protocol: VNC
                Host: localhost:5901
                Use JPEG encoding
        You should now be able to see Android UI
        If you encounter too many issues with the VNC over USB
        See explanation on how to set up VNC over Ethernet:
        http://www.opersys.com/blog/beaglebone-android-start

Android @Work:
1. Explore the Android UI through VNC
2. Note that all events have to have a "left-click" appended for them to propagate to the target
3. Check the version of Android you're running
        apps->Settings->About Phone->Android version
Use "logcat" to observe the target's Android
        $ adb logcat
4. Create a "Hello World" application in the VM and run it on the target

5. Have fun

Building AOSP for BeagleBone
1. Install U-Boot mkimage
        $ sudo apt-get install uboot-mkimage
2. Make sure you're using the right "jar"
        $ sudo update-alternatives --config jar
        Select the jar tool from Sun's SDK
3. Extract the TI BeagleBone Sources
        $ tar xvzf TI-Android-JB-4.1.2_AM335x_4.0.1.tar.gz
4. Retrieve source
        $ cd TI-Android-JB-4.1.2_AM335x_4.0.1
        $ ./.repo/repo/repo sync --local-only
5. Build source
        $ make TARGET_PRODUCT=beaglebone OMAPES=4.x -j4

Run your own build
1. Assuming your operating directory is "~/Desktop/android"
2. Preserve old copy of default Bone images
        $ cp -a BeagleBone BeagleBone-orig
3. Create new rootfs images
        $ cd TI-Android-JB-4.1.2_AM335x_4.0.1/out/target/product/beagleboard
        $ mkdir android_rootfs
        $ cp -a root/* android_rootfs/
        $ cp -r system android_rootfs/
        $ sudo ../../../../build/tools/mktarball.sh ../../../host/linux-x86/bin/fs_get_stats
        android_rootfs . rootfs rootfs_beaglebone.tar.bz2
4. Copy new rootfs to BeagleBone directory
        $ sudo cp rootfs_beaglebone.tar.bz2 ~/Desktop/android/BeagleBone/Filesystem
5. Plug the MicroSD card into the MicroSD-USB connector and into your host
6. Program the MicroSD card
        $ sudo ./mkmmc-android.sh /dev/sdb
7. Put the SD card back into the Bone, reboot it and connect through VNC as we did earlier
8. Check the version of Android you're running, it should be time-stamped with the date of
your build
        apps->Settings->About Phone->Android version
        or
        $ adb shell
        # cat /system/build.prop
        and check the ro.build.fingerprint and ro.build.description to see if it contains your
        build's time-stamp