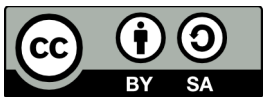
Hands-On Exercises for

Embedded Android

v. 2019.05

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 created by: Karim Yaghmour

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

Delivered and/or customized by:

Day 1

Class Preparation for HiKey board

WARNING: The list of prerequisites for Android development changes over time

1. Install Android Studio:
https://developer.android.com/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:
https://source.android.com/setup/initializing

3. Fetch the AOSP:
We'll be using 8.1/Oreo. 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-8.1.0_r18
$ cd aosp-8.1.0_r18
$ repo init -u https://android.googlesource.com/platform/manifest \
> -b android-8.1.0_r18
$ repo sync
```

Note that fetching the sources is a fairly lengthy process.

4. Patch 8.1.0_r18 to fix it for HiKey board. These instructions need to be done in the root of the AOSP just downloaded above:

```
$ wget http://opersys.com/downloads/hikey-8.1.0_r18-180319.patch
$ patch -p1 < hikey-8.1.0_r18-180319.patch
```

5. Fetch the binaries for the HiKey board (HDMI support). These instructions are also done in the AOSP root:

```
$ wget \
> https://dl.google.com/dl/android/aosp/linaro-hikey-20170523-4b9ebaff.tgz
$ tar xzf linaro-hikey-20170523-4b9ebaff.tgz
$ ./extract-linaro-hikey.sh
```

You will need to read through the license and type "I ACCEPT" at the end for the binaries to

extract.

6. Make sure you have “mtools” installed:
```
$ sudo apt-get install mtools
```

7. Build the AOSP:
```
$ cd ~/android/aosp-8.1.0_r18
$ . build/envsetup.sh
$ lunch hikey-eng
$ make -j8
```

8. Ensure that the build finishes successfully.

9. Flash your board with your AOSP:
   (a) Plug your board into the power jack and your computer through the USB cable. The board already has a system image on it that it'll boot with.
   (b) Make sure your workstation recognizes the device and gives you access to it. Have a look at the instructions here for more information: https://developer.android.com/studio/run/device.html#setting-up
   (c) Put the board in “fastboot” mode:
```
$ cd ~/android/aosp-8.1.0_r34
$ . build/envsetup.sh
$ lunch hikey-eng
$ adb reboot bootloader
```
   (d) The board should reboot and you should be able to see it using this command – example output provided:
```
$ fastboot devices
76B8809900099B63  fastboot
```
   (e) Reflash with your own images:
```
$ fastboot flashall
target reported max download size of 134217728 bytes
------------------------------------------
Bootloader Version...: 0.4
Baseband Version.....: 0.4
Serial Number........: 76B8809900099B63
------------------------------------------
checking product...
OKAY [  0.001s]
checking version-bootloader...
OKAY [  0.001s]
sending 'boot' (24380 KB)...
OKAY [  0.818s]
writing 'boot'...
...
```

```
OKAY [  3.097s]
writing 'system' 7/7...
OKAY [  3.819s]
rebooting...

finished. total time: 66.887s
```

(f) Check that your device booted properly into Android mode:

```
$ adb devices
List of devices attached
76B8809900099B63  device
```

(g) Log into your device:

```
$ adb shell
hikey:/ #
```

(h) Check that the build description matches your build date/time:

```
hikey:/ # cat system/build.prop | grep -i description
# Do not try to parse description, fingerprint, or thumbprint
ro.build.description=hikey-eng 8.1.0 OPM5.171019.017 eng.karim.20180308.124948 test-keys
```

Class Preparation for emulator

WARNING: The list of prerequisites for Android development changes over time

1. Install Android Studio:
https://developer.android.com/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:
https://source.android.com/setup/initializing

3. Fetch the AOSP:
We'll be using 9.1/Pie 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 android-9.0.0_r39
$ cd android-9.0.0_r39
$ repo init -u https://android.googlesource.com/platform/manifest \
> -b android-9.0.0_r39
$ repo sync
```

Note that fetching the sources is a fairly lengthy process.

4. Build the AOSP:

```
$ cd ~/android/android-9.0.0_r39
$ . build/envsetup.sh
$ lunch aosp_x86_64-eng
$ make -j8
```

5. Give the AOSP a spin:

From the same directory where you build the AOSP, run the emulator:

```
$ emulator &
```

At this point, an emulator should start with your custom-built Android running inside.

Kernel Basics

1. Check kernel version on the device (Shell into the device and type “cat /proc/version”)

2. Get the kernel and associated toolchain (if needed) – if you haven't already done so:

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

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

   b. Get the kernel itself – for Hikey:

```
$ git clone https://android.googlesource.com/kernel/hikey-linaro
$ cd hikey-linaro
$ git checkout -b android-hikey-linaro-4.9 \
> origin/android-hikey-linaro-4.9
$ git checkout 7c9f462e1a438c37a616063c381a59840793b1a0
```

   b. Get the kernel itself – for emulator:

```
$ git clone https://android.googlesource.com/kernel/goldfish
$ cd goldfish
$ git checkout android-goldfish-4.4-dev
```

3. Make sure you have libncurses5-dev installed:

```
$ sudo apt-get install libncurses5-dev
```

4. For the Hikey board:

Reconfigure the kernel to support: CONFIG_TI_ST and CONFIG_ST_HCI. Here's a reminder on how to get the kernel configuration that uses the default “hikey” configuration as its starting point:

```
$ cd ~/android/aosp-8.1.0_r18/kernel/hikey-linaro
$ make ARCH=arm64 hikey_defconfig
$ make ARCH=arm64 menuconfig
```

4. For the emulator:
Reconfigure the kernel to make sure support for modules and module unloading is enabled. You don't need to enable forced module unloading. Here's a reminder on how to get the kernel configuration that uses the default configuration from a running emulator as its starting point:

```
$ cd kernel/goldfish
$ adb pull /proc/config.gz
$ gunzip config.gz
$ mv config .config
$ make ARCH=x86 oldconfig
$ make ARCH=x86 menuconfig
```

5.For the Hikey board:
Rebuild your kernel. To rebuild the kernel, you need to do the following from the “kernel/hikey-linaro/” directory – after having made sure you had run the “. build/envsetup.sh” and “lunch hikey-eng” commands at the top of the AOSP directory:

```
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-android- -j8
```

5. For the emulator:
Rebuild, replace kernel, reboot. To rebuild the kernel, you need to do the following from the “kernel/goldfish/” directory:

```
$ make ARCH=x86 CROSS_COMPILE=x86_64-linux-android- bzImage -j8
```

6. For the Hikey board:
Replace the default kernel and device tree blob (DTB) in the AOSP. In order for the AOSP to take your newly built kernel into account, you'll need to do this -- again assuming that you've run the appropriate “. build/envsetup.sh” and “lunch” from the toplevel:

```
$ croot
$ cd device/linaro/hikey-kernel/
$ cp hi6220-hikey.dtb-4.9 hi6220-hikey.dtb-4.9-orig
$ cp Image-dtb-4.9 Image-dtb-4.9-orig
$ croot
$ cd kernel/hikey-linaro
$ cp arch/arm64/boot/dts/hisilicon/hi6220-hikey.dtb \
> ../../device/linaro/hikey-kernel/hi6220-hikey.dtb-4.9
$ cp arch/arm64/boot/Image-dtb \
> ../../device/linaro/hikey-kernel/Image-dtb-4.9
```

6. For the emulator:
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 prebuilts/qemu-kernel/x86_64/4.9/
$ cp kernel-qemu2 kernel-qemu2-orig
$ croot
$ cp kernel/goldfish/arch/x86/boot/bzImage \
> prebuilts/qemu-kernel/x86_64/4.9/kernel-qemu2
$ make -j8
```

7. For the Hikey board:

Rebuild your AOSP to take the new kernel into account:

```
$ croot
$ make -j8
```

7. For the emulator:

Restart the emulator and check the new kernel's version (“cat /proc/version”)

8. For the Hikey board:

Reflash your board with the new kernel:

```
$ adb reboot bootloader
$ fastboot flash boot
$ fastboot reboot
```

9. For the Hikey board:

Log back into your device and check the new kernel's version (“cat /proc/version”). It should match the time/date that you just built it on.

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

OR

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.bp :

```
"core/java/android/os/IPowerManager.aidl",
"core/java/android/os/IOpersysService.aidl",
"core/java/android/os/IRecoverySystem.aidl",
```

3. Modify your [aosp]/system/core/libcutils/fs_config.cpp to add the highlighted lines:

```
    { 00755, AID_ROOT,      AID_ROOT,      0, "bin/*" },
    { 00640, AID_ROOT,      AID_SHELL,     0, "fstab.*" },
    { 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*" },
    { 00750, AID_ROOT,      AID_SHELL,     0, "sbin/*" },
```

4. Rebuild your AOSP and reflash your device if needed. We will discuss the need for these changes later in class. We're doing them now (end of 1st day) to avoid having to wait for the builds to finish during class.

Day 2

Linux root filesystem

1. Get and install the Opersys Embedded Linux Workspace:

   For ARM:
   http://bit.ly/28Rz54O -- the last letter here is capital “O”, not zero.
   or
   https://drive.google.com/file/d/0B0dp197y8Ga7ejI4ZWFIVHh0LVE

   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. The script should be called “devbb” or “devx86”, or a similar name.

3. Run the the environment script:

   For ARM:
   ```
   $ . devbb
   ```

   For x86:
   ```
   $ . devx86
   ```

4. Create yourself a root filesystem that includes the glibc shared libraries as in slides 149 **and** 155, **except** for the “strip” command.

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

6. For BusyBox:
   - For ARM:
     - Download BusyBox 1.20.2 from busybox.net/downloads/ into sysapps
   - For x86:
     - Download BusyBox 1.26.2 from busybox.net/downloads/ into sysapps

- Extract BusyBox using the “tar” command
- 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 or restart 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/*[product_name]*/ 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.img)
- If you modify the linux root filesystem created earlier, you will need to erase the “rootfs-glibc” stub file from [aosp]/out/target/product/*[product_name]*/system/glibc/ to force the AOSP's build system to regenerate the system image.
- Do **NOT** get rid of [aosp]/out/target/product/*[product_name]*/ 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 file 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
- Create a stub file under that directory – every time you “touch” that file, it'll force a rebuild of this subdirectory:
  ```
  $ touch [aosp]/rootfs-glibc/stub
  ```
- 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

  OR

  http://bit.ly/T4H9V9

- Add "rootfs-glibc" to [aosp]/device/linaro/hikey/device-common.mk for the Hikey board or [aosp]/build/target/product/emulator.mk for the emulator:
```
PRODUCT_PACKAGES += rootfs-glibc
```

2. Using the client and server programs from http://www.linuxhowtos.org/C_C++/socket.htm, compile the server against Bionic and the client against glibc.
To have a the C server build against Bionic:

- Add it as [aosp]/frameworks/base/cmds/server
- You may need to add “#include <strings.h>” to the C file for it to build
- 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:
```
${PRJROOT}/tools/arm-unknown-linux-gnueabi/bin/arm-unknown-linux-gnueabi-gcc -o client client.c
```
  - For x86
```
${PRJROOT}/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-180219.tar.bz2.

- Make sure you “touch” any file you copy to update its timestamp, otherwise it's likely to get ignored at the next “make”.
- 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 “system/sepolicy/private/service_contexts” file to add this entry:

```
opersys                                   u:object_r:serial_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:

```
$ service call opersys 3 i32 412341
```

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 Hikey / 8.1:

- Put the driver's directory in [aosp]/device/linaro/hikey/
- This driver uses misc_register() instead of register_chrdev() as it will register your char dev and fire off a hotplug event. See: http://www.linuxjournal.com/article/2920
- Building the module:
```
$ . build/envsetup.sh
$ lunch hikey-eng
$ cd device/linaro/hikey/circular-driver/
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-androidkernel-
```
- To have the circular-driver.ko included in the final image, edit the device/linaro/hikey/device-common.mk to add the following:
```
PRODUCT_COPY_FILES += \
        device/linaro/hikey/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 Hikey is device/linaro/hikey/init.common.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/linaro/hikey/ueventd.common.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 the following rules before the “neverallow” rules in the “system/sepolicy/public/init.te” file:
```
allow init self:capability sys_module;
```

```
allow init system_file:system module_load;
```

- 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 “system/sepolicy/private/file_contexts” file:
```
/dev/circchar           u:object_r:tty_device:s0
```

For the Nexus 7 / 7.x:

- 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 and 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 the following rules before the “neverallow” rules in the “system/sepolicy/init.te” file:
```
allow init self:capability sys_module;
allow init system_file:system module_load;
```
- 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 “system/sepolicy/file_contexts” file:
```
/dev/circchar           u:object_r:tty_device:s0
```

For the MinnowBoardMax / 7.x:

- 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 and 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-4.9/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 the following rules before the "neverallow" rules in the "system/sepolicy/init.te" file:
```
allow init self:capability sys_module;
allow init system_file:system module_load;
```
- 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 "system/sepolicy/file_contexts" file:
```
/dev/circchar           u:object_r:tty_device:s0
```

For the emulator / 9.0:

- 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 and fire off a hotplug event. See: http://www.linuxjournal.com/article/2920
- Building the module:
```
$ cd device/generic/goldfish/circular-driver
$ make ARCH=x86 CROSS_COMPILE=x86_64-linux-android-
```
- To have the circular-driver.ko included in the final image, edit the build/target/product/emulator.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.ranchu.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.ranchu.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 the following rules before the "neverallow" rules in the "system/sepolicy/public/init.te" and the "sepolicy/prebuilts/api/28.0/public/init.te" file:

```
allow vendor_init self:capability sys_module;
allow vendor_init system_file:system module_load;
```

- 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 "system/sepolicy/private/file_contexts" and "sepolicy/prebuilts/api/28.0/public/file_contexts" file:

```
/dev/circchar          u:object_r:tty_device:s0
```

2. (Skip this is you've followed board-specific instructions in the previous exercise.) 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 tarball for this hands-on session:

   For the Hikey board:

   http://opersys.com/downloads/libhw-opersys-hikey-8.1-180404.tar.bz2

   For the emulator:

   http://opersys.com/downloads/libhw-opersys-goldfish-9.0-190530.tar.bz2

We will go through the files in this archive in class and the specific instructions to follow. Note though that it's important to “touch” any file you copy to update its timestamp, otherwise it's likely to get ignored at the next “make”.

2. Implement a dump() function 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 -- 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.

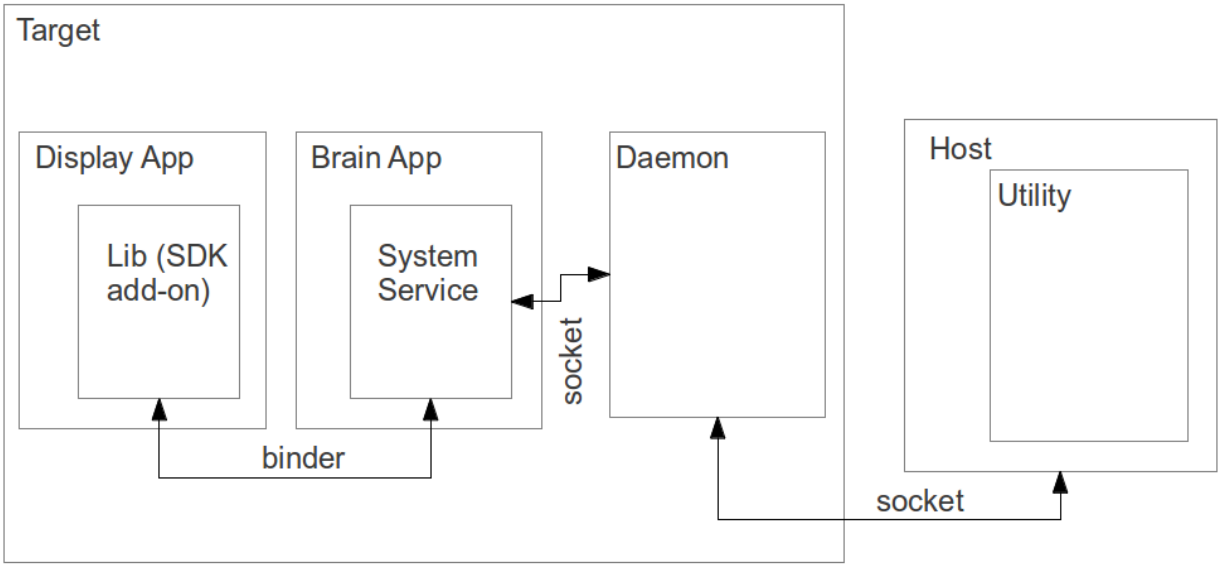
2. Modify the IOpersysService AIDL and the stack underneath it all the way to the driver to provide access to the following methods:

    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. Use the “service call” command to call on “enableIntentOnContent()” and catch the Intent with a BroadcastReceiver inside an app.

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