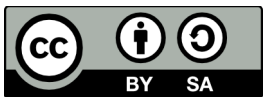
Hands-On Exercises for

Embedded Android

v. 2020.12

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:

Class Preparation Reference / Reminder

This section contains material that should have been used prior to class to prepare ahead of time. It's provided here as a reference/reminder.

Class Preparation for emulator

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

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 11.0.0 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-11.0.0_r3
$ cd android-11.0.0_r3
$ repo init -u https://android.googlesource.com/platform/manifest \
> -b android-11.0.0_r3
$ repo sync
```

Note that fetching the sources is a fairly lengthy process.

4. Build the AOSP:

```
$ cd ~/android/android-11.0.0_r3
$ . 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. It shouldn't take more than a minute or two to start.

AOSP Fix Ups

This section contains fixes that are required for future exercises. The significance, impact and necessity of these will be explained later. The reason we do these changes ahead of taking the time to explain them is that they incur a significant amount of build time. Hence, by the time we need them their addition would slow us down significantly.

In short, you are expected to conduct these changes “as is” for the moment. The reasons for these changes will be discussed later in class.

1. There is a file you need to download and put at a very specific location in your AOSP. The file to be retrieved is http://www.opersys.com/downloads/IOpersysService.aidl and it needs to be copied to this specific location in your AOSP: [aosp]/frameworks/base/core/java/android/os/

2. 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*" },
    { 00755, AID_ROOT,      AID_SHELL,     0, "odm/bin/*" },
...
```

WARNING: There are 2 data structures in that file that look fairly similar. You are modifying the one declared as follows:

```
static const struct fs_path_config android_files[] = {
```

3. Rebuild your AOSP from the toplevel using “make -j8” or equivalent. We will discuss the need for these changes later in class. Again, we're doing them in advance to avoid having to wait for the builds to finish during class.

System Server

In this section you will learn how to:

- Monitor system services
- Add your own system service to the core set of system services
- Check system services as part of system startup
- Communicate with a system service using “service call”
- Implement and retrieve the status of a running system service using “dumpsys”

1. There are several tools that can be used to monitor and interact with system services. We suggest you give some a try to see the sort of information you can obtain at runtime about the system services running on your system:

- Listing the main system services:
  # **service list**
- Listing vendor system services:
  # **vndservice list**
- Finding out what process is running the main system server process:
  # **ps -A | grep system_server**
- Dumping the internal state of all system services:
  # **dumpsys**
- Dumping the internal state of a single system service (Activity Manager in this case):
  # **dumpsys activity**
- Checking the logs for the system server process’ PID (replace [] value):
  # **logcat --pid=*[PID RETURBED BY PS COMMAND ABOVE]***
- Checking the logs for output from a given system service – WindowManager for ex:
  # **logcat | grep -i windowmanager**
- Communicate with the activity manager:
  # **am**
- Communicate with the package manager:
  # **pm**
- Communicate with the window manager:
  # **wm**

2. If you want to push the investigation further, you can use the strace command to check the various commands that communicate with system services. strace relies on the ptrace() kernel system call to monitor another command and print out what it does. For example, if you use strace on the dumpsys command, you’ll notice that dumpsys uses ioctl() system calls

to communicate with the binder driver (/dev/binder). We suggest giving strace a short spin if you've never used it before.

```
generic_x86_64:/ # strace -o /data/local/tmp/strace.out dumpsys
generic_x86_64:/ # cat /data/local/tmp/strace.out | grep ioctl
```

3. In this exercise, we'll have you add your own system service that prints out a message to the logs when one of its functions is called. There are several steps to carry out to do so:

a) Download and extract the following tarball, it contains samples we will use to get you started: http://opersys.com/downloads/systemserver-samples-201019.tar.bz2

Note regarding the suffixes to the files in tarballs we provide you – not all tarballs have all such files, but some other tarballs below may use these suffixes:

- Any file without one of the following suffixes:
  You can use that file as-is
- file-ALREADY_COPIED:
  This is a file that you were presumably asked to copy already. It's being provided here to you to reinforce its relevance in the present context, but we highly suggest you don't copy it again.
- file-DONOTCOPY:
  This is a file that you have to inspect to understand why we're asking you not to copy it as-is. Maybe it contains just a sample, maybe it's just an example. Context and instructor explanations will guide you here.
- file-EXISTS:
  This file exists in your tree, but the one we're providing you has some differences. You can use the "meld" tool to compare our version with the one in your AOSP to understand what we're adding and why we're adding it.

Make sure you "touch" any file you copy to update its timestamp, otherwise it's likely to get ignored at the next "make".

b) You need to place the OpersysService.java file from the tarball at exactly the following location in your AOSP – double-check to make sure this is the proper path as the wrong path will cause your build some serious issues:

[aosp]/frameworks/base/services/core/java/com/android/server/

OpersysService.java contains a very basic custom system service that, at this point, doesn't really do much. You can check the code to see what it does. Primarily it prints a message to logcat when it starts and responds to calls to its test() call.

WARNING: When you first build this system service, the build will fail and the compiler will complain about a problem with OpersysService.java. The resolution of this error is

part of the exercise. You need to try to understand what the issue is and try to fix the code correspondingly. The goal here is not to code anything substantial but rather to "keep the compiler happy". So look at the error and try to add just enough code to get the compiler to compile the class. You shouldn't need more than a handful of lines of code.

c) You need to modify the main system service registration routine to start the custom OpersysService system service at startup. To do so, locate the SystemServer.java file by using "godir" and open it for editing. The SystemServer.java-DONOTCOPY file contains the sample you need to add to your own SystemServer.java file in order for the OpersysService to be started and registered at startup. One easy place to add this system service is right after the registration try/catch statement for the Status Bar Manager system service. Look for a ServiceManager.addService() call with Context.STATUS_BAR_SERVICE as its first parameter.

d) You will also need to modify the selinux policies in order to allow your system service to be registered at startup. If you fail to do so then even if all your code is included at build time, the system service will fail to register at startup. To allow registration, you need to modify 2 files under the "system/sepolicy" directory:
   - "private/service_contexts"
   - "prebuilts/api/30.0/private/service_contexts"

   Both files have to be identical and must be modified to add this entry:
   ```
   opersys                                    u:object_r:serial_service:s0
   ```
   We suggest you modify only one of the files and copy it over to the second. If there is any difference between the 2 files, including just whitespace differences, then the build will fail. You need to modify 2 files because once a version of Android is released by Google then the core security policies are supposed to be fixed in stone. Yet, here we are messing with those rules. So we need to act as if we had amended the prebuilt definitions that Google itself would have published.

   Also, note that we are tagging our system service as belonging to the serial_service domain. This is a hack to simplify SELinux rule definitions. If we wanted to do this properly, we'd need to define a new domain called "opersys_service" and make modifications to several other SELinux files. As this isn't an SELinux exercise, we're using an existing definition that already works.

4. Build the AOSP and use "logcat" to check that your service is started upon system startup. Refer to the instructions from previous exercises on how to build your AOSP or using logcat to monitor system services. Make sure you start the emulator with the "-no-cache" option,

otherwise your changes may not be taken into account.

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

a) Calling the test() call from the IOpersysService.aidl, thereby causing the test() call in OpersysService.java to be called – test() is the 3rd call in the AIDL (hence the "3") and it takes an int as a parameter (hence the "i32" type for int and "412341" for the value being passed):

```
# service call opersys 3 i32 412341
Result: Parcel(00000000    '....')
```

b) Checking logcat should yield something like this:

```
# logcat | grep -i opersys
...
10-18 20:53:12.548   474   815 I OpersysService: test 412341
```

6. As you saw earlier, one way to investigate the behavior of system services at runtime is to use the "dumpsys" command. In order for that command to be able to retrieve the state of system service, that system service must have a method with the following signature/prototype in its implementation:

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

Add such a method to the OpersysService system service you added earlier to allow it to be checked by "dumpsys" at runtime. One of the goals of the present exercise is to have you look at existing system service implementations for examples of how to do things. As such, we suggest you go to the same directory where you added OpersysService.java (i.e. [aosp]/frameworks/base/services/core/java/com/android/server/) and look at other classes to see how the other system services implement the "dump()" method. A few recommendations while you're doing that:

- The simplest thing to do in your dump() implementation is to just print something out for the purposes of the dump command output. A "Hello World!" will be sufficient.
- You'll notice that most system services do some security checks for permissions when they receive this call. You don't have to do that for the present exercise.
- You will add at least 2 "import" statements at the top of your OpersysService.java file in order for your class to compile once you add your dump() method. Again, look at existing implementations for guidance.

Linux Root Filesystem

In this section you will learn how to:

- Set up a workspace to create a custom embedded Linux root filesystem
- Create a skeleton of a root filesystem
- Add a glibc library to a root filesystem
- Configure, build and install BusyBox in a glibc-based embedded Linux root filesystem

NOTE: Refer to the instructor as to whether or not this exercise is conducted during class.

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. An easy way to set the path automatically is to set it as follows:

```
export PRJROOT=${PWD}
```

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 199 **and** 205, **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:
     - “Busybox Settings”->”Cross Compiler prefix” (notice the trailing dash):
       - **`${TARGET}-`**
     - “Busybox Settings”->“BusyBox installation prefix”:
       - **`${PRJROOT}/rootfs`**
   - Build BusyBox per the slides:
     - `$` **`make -j8`**
   - Install BusyBox per the slides
     - `$` **`make install`**

Native Android User-Space

In this section you will learn how to:

- Modify the AOSP build system to take into account independently-generated content
- Work with some of the AOSP's filesystem packaging/generation mechanisms
- Merge a glibc-based root filesystem into AOSP-generated filesystems
- Incorporate 3rd party sources into the AOSP and have them build against Bionic
- Make programs built against different C libraries communicate through the Linux kernel

In addition to the present exercises, several exercises in subsequent sections will also require you to modify and integrate with native Android user-space components.

1. Modify the AOSP build system to copy the content of a glibc rootfs to its default system image. You will need to:

- Create [aosp]/rootfs-glibc
  ```
  $ cd ~/android/android-11.0.0_r3/
  $ mkdir rootfs-glibc
  ```
- Create a stub file under the rootfs-glibc directory.
  ```
  $ touch rootfs-glibc/stub
  ```
  This stub file will act as the build system dependency. Every time you "touch" that file, it'll force a rebuild of this subdirectory. This is useful because we are going to put arbitrary content underneath this location which the build system will not be aware of otherwise.
- Put a prebuilt glibc-based root filesystem under the rootfs-glibc/rootfs directory in your AOSP. In the context of this class there are 2 ways to get such a root filesystem – refer to instructor directives to select which of the two to use:
  - By copying the one generated in the previous "Linux Root Filesystem" section exercises:
    ```
    $ cp -a ${PRJROOT}/rootfs rootfs-glibc/
    ```
  - By downloading a prebuilt tarball and extracting it under rootfs-glibc:
    ```
    $ cd rootfs-glibc/
    $ wget http://www.opersys.com/downloads/rootfs-200727.tar.bz2
    $ tar xvjf rootfs-200727.tar.bz2
    $ croot
    ```

  The content of [aosp]rootfs-glibc/rootfs is what will be merged into the images generated by the AOSP.
- You will need a specific build file to instruct the build system to copy the content of [aosp]rootfs-glibc/rootfs into the images it generates. Such a file is provided to you at

http://www.opersys.com/downloads/glibcroot-Android-sys.mk and must be put into rootfs-glibc and renamed to Android.mk:

```
$ cd rootfs-glibc
$ wget http://www.opersys.com/downloads/glibcroot-Android-sys.mk
$ mv glibcroot-Android-sys.mk Android.mk
$ croot
```

The semantics and working of this file will be discussed in class.

- In order for the build system to choose to build this new "build module" that we just added, the name of this module must be added to the list of modules the build system is instructed to build. If you look in the Android.mk we just added, you'll see that the name of the module is "rootfs-glibc". Hence, we need to find a proper entry to extend the PRODUCT_PACKAGES variable (which lists the packages to be built for the target). In this specific case, one of the easiest ways to do so for the type of artifacts we are adding is to modify the PRODUCT_PACKAGES in [aosp]/build/target/product/base_system.mk to add the following entry:

```
PRODUCT_PACKAGES += rootfs-glibc
```

- If you look at the Android.mk we added you'll see that it copies the content of the glibc-based rootfs to the /system/glibc directory on the final filesystem layout on the target. That entry is not by default taken into account by the build system of Android and provisioned with the proper Unix execution permissions. To fix this you need to modify the [aosp]/system/core/libcutils/fs_config.cpp file. We did this, however, earlier in the "AOSP Fix-Ups" exercise section. Hence, it's not needed here.
- Much like many other aspects of Android, you need to add SELinux rules in order for the newly-merged content to operate properly. In this specific case, here is a snippet that you need to add at the tail end of your [aosp]/device/generic/goldfish/sepolicy/common/file_contexts file:

```
# For rootfs-glibc
/lib64(/.*)?          u:object_r:rootfs:s0
```

This file controls the device-specific SELinux access rights. This rule states that "/lib64" entries belong to the "rootfs" domain and will be accessible subject to the rules set for any process that has access to that domain.

- Once you made all the changes above, you need to rebuild your AOSP and restart the emulator. Don't forget the "-no-cache" option when starting the emulator.
- To test your additions, shell into the restarted emulator and run one of the commands from the glibc-linked BusyBox:

```
$ adb shell
generic_x86_64:/ # /system/glibc/bin/busybox
BusyBox v1.26.2 (2020-07-27 21:12:36 EDT) multi-call binary.
BusyBox is copyrighted by many authors between 1998-2015.
...
```

```
generic_x86_64:/ # /system/glibc/bin/ls
...
generic_x86_64:/ # /system/glibc/bin/ps
...
```

2. For this exercise, you will be testing a configuration where you have a client process built against glibc communicate over TCP/IP with a server process linked against Bionic. Sample code for a client and a server are available from http://www.linuxhowtos.org/C_C++/socket.htm.
To have the C server build against Bionic:

- Add a new subdirectory: [aosp]/frameworks/base/cmds/server
- Add the server.c file to the just-created subdirectory
- In recent versions of Android (including 11), you need to add "#include <strings.h>" to the C file for it to build. Note that there already is a "string.h" (singular) that's present. That's not sufficient nor does it need to be removed or replaced. You really need to add "strings.h" (plural).
- Make sure "server" is added to a "PRODUCT_PACKAGES" list in [aosp]/build/target/product/base_system.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)
```

If you are using the prebuilt rootfs tarball downloaded from opersys.com then the glibc-linked client is already compiled for you. If you need to manually build the client against glibc then:

- 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
```

As before, you need to rebuild your AOSP and restart your emulator to test these changes.

To start the server (from the device's command line):

```
# server 4444 &
```

To connect to the server from the client:

```
# /system/glibc/bin/client localhost 4444
```

The client will prompt you for a message to send to the server and the server will ack reception.

Kernel Basics

In this section you will learn how to:

- Check what kernel version you are using
- Fetch, build and use a set of kernel sources “the Android way”
- Replace a default AOSP-provided kernel with a custom one

1. Check kernel version on the device (Shell into the device and type “cat /proc/version”). Your output should look something like this:

```
generic_x86_64:/ # cat /proc/version
Linux version 5.4.47-01061-g22e35a1de440 (android-build@abfarm-east4-070)
(Android (6443078 based on r383902) clang version 11.0.1
(https://android.googlesource.com/toolchain/llvm-project
b397f81060ce6d701042b782172ed13bee898b79), LLD 11.0.1
(/buildbot/tmp/tmp6_m7QH b397f81060ce6d701042b782172ed13bee898b79)) #1 SMP
PREEMPT Fri Jun 19 01:58:49 UTC 2020
```

This information is useful to know what version of the kernel you are relying on.

2. Download the kernel and associated toolchain:

a. Create directory for working on kernel – assuming this is outside your the top-level of the AOSP:

```
$ cd ~/android/
$ mkdir aosp11-kernel
$ cd aosp11-kernel
```

b. Get the kernel itself and its tools:

```
$ repo init -u https://android.googlesource.com/kernel/manifest \
> -b common-android11-5.4
$ repo sync
```

The final command will take a bit of time to download everything that you need. Once it’s done, you’ll then have a set of kernel sources and the tools to build them. More information about using this technique of fetching the kernel and the following instructions on building the kernel can be found here: https://source.android.com/setup/build/building-kernels

3. Make sure you have libncurses5-dev and libssl-dev installed:

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

These packages are required for building the kernel. If you omit them then you will typically get errors at kernel build time.

4. By default, the kernel build script attempts to generate an LZ4-compressed kernel image at build time. If you are using Ubuntu 18.04 or earlier, the tool for generating this image is either

absent or won't work. Hence, we need to reconfigure the kernel to use BZ2 compression instead. To do so, open the "common/arch/x86/configs/gki_defconfig" file from the toplevel of the kernel repository and replace the LZ4 configuration. Here is what a "git diff" looks like on that file once it has been modified (changes in bold):

```
diff --git a/arch/x86/configs/gki_defconfig
b/arch/x86/configs/gki_defconfig
index 5f6b1e4dd7f3..5ec13a27f3f1 100644
--- a/arch/x86/configs/gki_defconfig
+++ b/arch/x86/configs/gki_defconfig
@@ -1,5 +1,5 @@
 CONFIG_UAPI_HEADER_TEST=y
-CONFIG_KERNEL_LZ4=y
+CONFIG_KERNEL_BZIP2=y
 # CONFIG_USELIB is not set
 CONFIG_AUDIT=y
 CONFIG_NO_HZ=y
```

"CONFIG_" variables are the means by which a kernel is configured.

5. Build the kernel and then the kernel modules required to boot the emulator. To do so, issue these commands at the toplevel of the kernel repository you downloaded earlier:

```
$ cd ~/android/aosp11-kernel/
$ SKIP_MRPROPER=1 BUILD_CONFIG=common/build.config.gki.x86_64 \
> build/build.sh
$ SKIP_MRPROPER=1 \
> BUILD_CONFIG=common-modules/virtual-device/build.config.goldfish.x86_64 \
> build/build.sh
```

The "build.sh" script is the primary build script for kernels and it takes parameters in the form of environment variables set prior to its execution. You can look at the script for additional options. Beware that the use of the options is not always clear and you may need to inspect the actual code to understand what's going on.

Once the build is over, you can check the "out/android11-5.4/dist/" directory for the output. It should contain a "bzImage" file, which is the kernel image, as well as many "*.ko" files, which are the kernel modules required for this kernel.

6. Replace the kernel and modules provided by default in the AOSP for the emulator with the kernel and modules you just built, and rebuild the AOSP:

```
$ cd ~/android/android-11.0.0_r3
$ cd prebuilts/qemu-kernel/x86_64/
$ mv 5.4 5.4-orig
$ mkdir -p 5.4/ko
$ cd ~/android
$ cp aosp11-kernel/out/android11-5.4/dist/bzImage \
> android-11.0.0_r3/prebuilts/qemu-kernel/x86_64/5.4/kernel-qemu2
$ cp aosp11-kernel/out/android11-5.4/dist/*.ko \
> android-11.0.0_r3/prebuilts/qemu-kernel/x86_64/5.4/ko
```

```
$ cd android-11.0.0_r3
$ . build/envsetup.sh
$ lunch aosp_x86_64-eng
$ rm out/target/product/generic_x86_64/kernel-ranchu
$ make -j8
```

The default emulator kernel and modules are provided under “prebuilts/qemu-kernel”. The commands above replace those files with the ones built earlier. For real devices, the AOSP generally stores kernel modules under a “device/VENDOR/DEVICE-kernel” directory. BSPs you get from silicon vendors might operate in a completely different way.

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

```
$ emulator -no-cache &
$ adb shell cat /proc/version
Linux version 5.4.61-android11-1-00003-g8d3d29f0729a-dirty (build-user@build-
host) (Android (6443078 based on r383902) clang version 11.0.1
(https://android.googlesource.com/toolchain/llvm-project
b397f81060ce6d701042b782172ed13bee898b79), LLD 11.0.1
(/buildbot/tmp/tmp6_m7QH b397f81060ce6d701042b782172ed13bee898b79)) #1 SMP
PREEMPT 2020-10-17 11:13:17
```

The “-no-cache” option of the emulator is very important because it’ll avoid using a cached previous emulator run. Notice that the version information printed out is different from the one you last checked it.

Linux Device Driver

In this section you will learn how to:

- Integrate a vendor driver to the Google Kernel repository
- Compile a vendor driver using Google's kernel build tools
- Modify the AOSP to take into account a new device in the native filesystem
- Integrate a driver with native Android user-space components for proper operation
- Check if a driver has been properly-loaded at boot time

NOTES:

- The exercises in this section assume that you have a kernel that was downloaded and built according to the "Kernel Basics" exercise section.
- The tarball we'll use for our exercise is here: http://www.opersys.com/downloads/vendor-circular-driver-201020.tar.bz2
- The driver we use in these exercises implements a circular buffer over the read/write file-operations. When read, if there is not data available, it returns 0 bytes.
- The driver is, however, simplistic. It doesn't implement blocking/waking semantics nor does it use proper locking primitives. It's ok for an exercise, but wouldn't be something you would ship to anyone.
- Also, the driver relies on misc_register() instead of register_chrdev() and will therefore register your char dev and fire off a hotplug event.
- For more information on device driver writing in Linux, have a look at the Linux Device Drivers book (dated as it may be): http://lwn.net/Kernel/LDD3/
- For reference, there's another driver sample here: http://tldp.org/LDP/lkmpg/2.6/html/x569.html

1. Download and extract the sample driver within the toplevel directory of the previously-downloaded kernel repository:

```
$ cd ~/android/aosp11-kernel/
$ wget http://www.opersys.com/downloads/vendor-circular-driver-201020.tar.bz2
$ tar xvjf vendor-circular-driver-201020.tar.bz2
```

Have a look at the content of the "vendor/" directory to see what's inside for building the driver. Note that Google has no known documentation at the time of this writing that explains how to build "vendor" modules alongside its GKI kernels. Hence, while the recipe here works, it's not necessarily the final way by which Google will want or instruct vendors to build their modules against GKI kernels.

2. Build the driver using Google's build script and copy it to the proper location in

```
$ SKIP_MRPROPER=1 BUILD_CONFIG=common/build.config.gki.x86_64 \
> EXT_MODULES="vendor/circular-driver" build/build.sh
$ cp out/android11-5.4/dist/circular-char.ko \
> ~/android/android-11.0.0_r3/prebuilts/qemu-kernel/x86_64/5.4/ko
```

The build process will generate a "circular-char.ko" under out/android11-5.4/dist/. This is the driver module. We need to copy it to the proper location under "prebuilts" in the AOSP in order for it to be: a) taken into account by the AOSP build system, and b) automatically loaded at start from vendor-image.

3. 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
```

Without this line, the driver will be owned by the "root" user and will be inaccessible to a system service running as the "system" user.

4. To allow the device driver 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 following file:
"device/generic/goldfish/sepolicy/common/file_contexts":

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

As before, this file contains the SELinux access rules for files in the filesystem. The rule we just added states that "/dev/circchar" belongs to the "tty_device" domain and will be accessible subject to the rules set for any process that has access to that domain.

5. Recompile your AOSP and restart your emulator (again, don't forget "-no-cache"). You should then shell into your device and see the module loaded and the device under /dev:

```
$ adb shell
generic_x86_64:/ # lsmod
Module                  Size  Used by
virtio_wifi            24576  0
virtio_pmem            16384  0
virtio_pci             28672  0
...
circular_char          16384  0
generic_x86_64:/ # dmesg | grep circ
[    0.663562] init: Loading module /lib/modules/circular-char.ko with args
[    0.664258] circular_char: loading out-of-tree module taints kernel.
[    0.704016] init: Loaded kernel module /lib/modules/circular-char.ko
generic_x86_64:/ # ls -al /dev/circchar
```

```
crw-rw-rw- 1 system system 10,  59 2020-10-19 08:10 /dev/circchar
```

Notice how the entry under /dev/ belongs to the “system” user and group.

6. You can now use “cat” and “echo” commands to read and write to your device:

```
generic_x86_64:/ # echo foobar > /dev/circchar
generic_x86_64:/ # cat /dev/circchar
foobar
generic_x86_64:/ # cat /dev/circchar
generic_x86_64:/ #
```

Notice how the 2nd “cat” command returns nothing. This is important because it means that the bytes have been “consumed”. The driver is also fairly verbose in the kernel logs. In a typical driver this would be wrong to do, but the exercise is meant to be academic:

```
generic_x86_64:/ # dmesg
[  272.584741] device_open called
[  272.586177] device_write called
[  272.587692] Writing f
[  272.588957] Writing o
[  272.590306] Writing o
[  272.591285] Writing b
[  272.592183] Writing a
[  272.592995] Writing r
[  272.593807] Writing \x0a
[  272.595215] device_release called
[  276.354275] device_open called
[  276.354555] device_read called
[  276.354794] Reading o
[  276.354989] Reading o
[  276.355158] Reading b
[  276.355328] Reading a
[  276.355497] Reading r
[  276.355666] Reading \x0a
[  276.355929] Reading \x00
[  276.356125] device_read called
[  276.356707] device_release called
[  282.449302] device_open called
[  282.449536] device_read called
[  282.449756] device_release called
```

Hardware Abstraction Layer / HIDL

In this section you will learn how to:

- Connect a system service to a device driver through the HAL
- Add native code to a system service
- Introduce a new HAL using HIDL files into the HAL
- Add a legacy header HAL definition to the HAL
- Connect a HAL module to a device driver
- Add the relevant extensions/changes to the AOSP for all of the above

For this exercise, we will use a tarball that is structured with the same directory hierarchy as your AOSP. Despite the hierarchy similarity, do NOT extract this archive into or onto your AOSP. Instead, extract it separately and proceed on to copy the content bit by bit, as needed, into your AOSP. As before, the following file suffixes apply:

- Any file without one of the following suffixes:
  You can use that file as-is
- file-ALREADY_COPIED:
  This is a file that you were presumably asked to copy already. It's being provided here to you to reinforce its relevance in the present context, but we highly suggest you don't copy it again.
- file-DONOTCOPY:
  This is a file that you have to inspect to understand why we're asking you not to copy it as-is. Maybe it contains just a sample, maybe it's just an example. Context and instructor explanations will guide you here.
- file-EXISTS:
  This file exists in your tree, but the one we're providing you has some differences. You can use the "meld" tool to compare our version with the one in your AOSP to understand what we're adding and why we're adding it.
- file-EXAMPLE:
  This is a sample that you may want to either use as-is or take portions of for your own use.

Note: it's important to "touch" any files you copy to update timestamps, otherwise they'll likely get ignored at the next "make".

Tarball for this section of exercises:

http://opersys.com/downloads/libhw-opersys-goldfish-11.0-201103.tar.bz2

1. Download and extract the exercise tarball:

```
$ cd ~/android
$ wget http://opersys.com/downloads/libhw-opersys-goldfish-11.0-201103.tar.bz2
$ tar xvjf libhw-opersys-goldfish-11.0-201103.tar.bz2
```

2. Add the System Server portions of the tarball to your AOSP. These are the relevant bits (note the suffixes):

```
frameworks/
└── base
    ├── core
    │   └── java
    │       └── android
    │           └── os
    │               └── IOpersysService.aidl-ALREADY-COPIED
    └── services
        ├── core
        │   ├── java
        │   │   └── com
        │   │       └── android
        │   │           └── server
        │   │               └── OpersysService.java-EXAMPLE
        │   └── jni
        │       ├── Android.bp-EXISTS
        │       ├── com_android_server_OpersysService.cpp
        │       └── onload.cpp-EXISTS
        └── java
            └── com
                └── android
                    └── server
                        └── SystemServer.java-EXISTS
```

Regarding the various parts of this tree:

- You've already added the IOpersysService.aidl file and it's provided here for completeness. Avoid touching this file as it will trigger a rebuild of the framework.
- You already have an OpersysService.java but it doesn't contain the necessary code to interact with its JNI counter-side. The version here does provide the JNI hooks and actual implementations of read() and write(), but it doesn't have a dump() function such as the one you implemented earlier. If you want to preserve your dump() function then merge it with this file and use the result as your new OpersysService.java. Otherwise, you can use this file as-is.
- Everything under "jni" is more or less new. com_android_server_OpersysService.cpp is the JNI side of the system service. You can take this file as-is without modification.

It works “out of the box”. The corresponding Android.bp and onload.cpp already exist in your tree and you need to take the relevant parts of the versions we’re giving you and copy only the parts relevant to the present system service. You can use “meld” to compare your version to ours.

- You already likely amended SystemServer.java in the previous system service exercise. It’s provided here also for completeness.

3. Add the hardware/HAL portions of the tarball to your AOSP. These are the relevant bits (note the suffixes, or lack thereof in this case):

```
hardware
├── interfaces
│   └── opersys
│       ├── 2.0
│       │   ├── Android.bp
│       │   ├── default
│       │   │   ├── Android.bp
│       │   │   ├── android.hardware.opersys@2.0.xml
│       │   │   ├── Opersys.cpp
│       │   │   └── Opersys.h
│       │   ├── IOpersys.hal
│       │   └── types.hal
│       └── Android.bp
└── libhardware
    └── include
        └── hardware
            └── opersyshw.h
```

Regarding the various parts of this tree:

- You can copy all this content as-is in the proper location of your tree without any changes. It works “out of the box”
- All Android.bp files here were originally auto-generated either using the “hidl-gen” tool or by running the “hardware/interfaces/update-makefiles.sh” script. Some of them had to be tweaked after having been generated. For the moment there is nothing for you to change here.
- The .hal files are the HIDL definitions of the “Opersys” hardware type.
- The files under “hardware/interfaces/opersys/2.0/default/” are the default HIDL implementation for the Opersys hardware type that connects the HIDL definition the legacy header-based HAL definition.
- opersyshw.h is the legacy HAL definition (pre-Treble) for the Opersys hardware type.
- android.hardware.opersys@2.0.xml contains the VINTF fragment required for this Opersys default HIDL implementation to operate properly (i.e. as a passthrough implementation).

4. Fix the API definitions to allow the new hardware type to be permitted as a same-process/passthrough implementation. These are the relevant bits (note the suffixes):

```
build
└── target
    └── product
        └── gsi
            └── 30.txt-EXISTS
```

Regarding the various parts of this tree:

- 30.txt contains the Google-published reference for what APIs/definitions are available in the AOSP release that matches API level 30 (i.e. Android 11). Same-process/passthrough implementations are only available for HALs that Google explicitly grants. If we don't modify this file, therefore, we won't be able to use the HAL parts that we added since we are using the passthrough implementation for our new hardware type. Note that in general this isn't something that you'd want to be doing in a shipping AOSP. One of the purposes in the present exercises is to understand some of the core mechanisms in the AOSP.

5. Fix the SELinux policies to allow the new additions. These are the relevant bits (note the suffixes):

```
system/
└── sepolicy
    └── private
        ├── hwservice_contexts-EXISTS
        └── service_contexts-EXISTS
```

Regarding the various parts of this tree:

- You already amended service_contexts earlier to permit the registration of your system service. It's provided here for completeness.
- You need to modify the hwservice_contexts file to allow the new hardware type to have HIDL implementations registered for it. Much like earlier for the service_contexts, we are reusing an existing domain definition to tag our hardware type. This isn't how it should be done, but it simplifies the exercise with regards to SELinux rule modifications.

6. Add the device-specific portions of the tarball to your AOSP. These are the relevant bits (note the suffixes):

```
device
└── generic
    └── goldfish
        ├── opersyshw
        │   ├── Android.mk
```

```
│   └── opersyshw.c
├── sepolicy
│   └── common
│       └── file_contexts-EXISTS
├── ueventd.ranchu.rc-EXISTS
└── x86_64-vendor.mk-EXISTS
```

Regarding the various parts of this tree:

- The "device/generic/goldfish/opersyshw/" directory contains the actual HAL module implementation. This is the portion that sits under the HAL and should implement communication with the device driver you added earlier. **IMPORTANT**: Note that the opersyshw.c file we give you does NOT actually communicate with your driver. This is part of the exercise. You need to add the necessary code to that file to communicate with "/dev/circchar". We suggest you google a bit for "Unix file I/O" and look at some examples. You shouldn't need to add more than about 5 or 6 lines of code in the entire file. Don't overdo it and make sure you use unbuffered file I/O operations.
- You should have already modified the ueventd.ranchu.rc file earlier. It's provided here with modifications for completeness.
- The x86_64-vendor.mk file contains additions for ensuring the HAL module above and necessary HAL libraries are built.
- The file_contexts file contains SELinux rules that make the HAL module and HAL libraries accessible as required by the relevant security contexts. Note that you already modified this file earlier and the version we are giving you here might contain things that you have already added – which doesn't mean there aren't new things that you don't already have.

7. Once you have made your additions, rebuild your AOSP and restart your emulator (don't forget the "-no-cache" parameter).

8. To test your additions, you should be able to write and read into the driver through the system service's AIDL interface and correspondingly read and write from the driver. For example:

```
# service call opersys 2 s16 foobar
Result: Parcel(00000000 00000006   '........')
# cat /dev/circchar
foobar
# echo test-string > /dev/circchar
# service call opersys 1 i32 40
Result: Parcel(
  0x00000000: 00000000 0000000c 00650074 00740073 '........t.e.s.t.'
  0x00000010: 0073002d 00720074 006e0069 000a0067 '-.s.t.r.i.n.g...'
  0x00000020: 00000000                            '....            ')
```

9. Implement or extend the 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.

Stack Extension Exercises

In this section you will learn how to:

- Extend a vertical stack from the app level to the driver to add new functionality
- Add new definitions to an AIDL file
- Implement new AIDL definitions in a system service
- Connect Java-based system service calls to C++ using JNI
- Implement JNI functions and their entries in JNI method tables
- Generate required build artifacts for new HAL definitions
- Generate default implementations for HAL definitions
- Modify default HAL implementations to implement calls to legacy HALs
- Extend legacy HALs
- Implement new calls in a legacy HAL module
- Connect a HAL module to a driver using ioctl() calls
- Extend a character device driver to add ioctl() calls

We suggest you use the previous file tree descriptions as your guide to this exercise. The layers are all the same, and there are modifications required at every layer.

1. Modify the IOpersysService AIDL and the system service deriving from it to add the following class:

    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

Stub the implementations in OpersysService.java to “keep the compiler happy” and restart the AOSP build. We do this in the beginning because this change will cause a framework rebuild which will itself trigger the rebuild of quite a few components. Since this takes a bit of time, it’s best to start by making this disruptive change first and then proceed in parallel to the rest of the changes while the build continues.

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

A few notes regarding this exercise:

- ioctl() calls are generally frowned upon for new drivers, /sys (sysfs) entries being preferable. We are using them here for simplification.
- You can use the Linux Device Drivers book linked to earlier for a reference on how to implement ioctl() calls.
- You'll need to implement unlocked_ioctl() instead of ioctl() in the file operations struct. Here is the signature of the function pointer you need to add an implementation for in your driver (taken from include/linux/fs.h):
  ```
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  ```
  You should define yourself a device_ioctl that has a similar signature:
  ```
  long device_ioctl (struct file * filep, unsigned int cmd, unsigned long arg) {}
  ```
- You then need to amend the file_operations struct in circular-char.c to connect the unlocked_ioctl function pointer to your implementation:
  ```
  ...
  .unlocked_ioctl = device_ioctl(),
  ...
  ```
- The device_ioctl() function must do a switch-case on the “cmd” parameter. You can define your command set as something like this – this isn’t the proper way to do that, but it works for the present exercise:
  ```
  #define CMD_BASE                0x134814323
  #define CMD_ZERO_CONTENT        CMD_BASE + 0
  #define CMD_IS_THERE_CONTENT    CMD_BASE + 1
  ...
  ```
- Once you have modified the driver, refer the previous device driver instructions on how to build the driver and copy it to your AOSP for inclusion at the next rebuild of the AOSP.

3. Once the driver is modified, go to the HAL module in device/generic/goldfish/opersyshw/ and modify opersyshw.c to add calls to invoke the new calls in the driver. You should have something like this added to your file:

```
#define CMD_BASE                0x134814323
#define CMD_ZERO_CONTENT        CMD_BASE + 0
#define CMD_IS_THERE_CONTENT    CMD_BASE + 1
...
void opersyshw__zero_content()
{
```

```
        ioctl(fd, CMD_ZERO_CONTENT, 0);
}

int opersyshw__is_there_content()
{
        return ioctl(fd, CMD_IS_THERE_CONTENT, 0);
}
```

You will also need to modify the open_opersyshw() function to register those new functions into the legacy HAL definitions that will need to also amend further in the exercises:

```
static int open_opersyshw(const struct hw_module_t* module, char const* ...
        struct hw_device_t** device)
{
...
dev->zero_content = opersyshw__zero_content;
dev->is_there_content = opersyshw__is_there_content;
...
```

4. Extend the legacy HAL struct in opersyshw.h under hardware/libhardware/include/hardware to add the new calls.

5. Rework the HIDL definitions to add the new calls. This is a bit of a tricky step. You need to:

1. Keep a backup of the hardware/interfaces/opersys/ directory we gave you
2. Go into hardware/interfaces/opersys/ and delete everything except the .hal files
3. Amend the .hal files to add your new calls
4. Use the hidl-gen script as described here (https://source.android.com/devices/architecture/hidl-cpp#creating-the-hal-server) to regenerate a default HIDL implementation:

```
$ croot
$ PACKAGE=android.hardware.opersys@2.0
$ LOC=hardware/interfaces/opersys/2.0/default/
$ hidl-gen -o $LOC -Lc++-impl -randroid.hardware:hardware/interfaces \
    -randroid.hidl:system/libhidl/transport $PACKAGE
$ hidl-gen -o $LOC -Landroidbp-impl -randroid.hardware:hardware/interfaces \
    -randroid.hidl:system/libhidl/transport $PACKAGE
```

5. Use the update-makefiles.sh script to update the toplevel opersys hal definition makefiles:

```
$ cd hardware/interfaces/
$ ./update-makefiles.sh
```

6. Compare **all** the files generated to those of the backup you did at the beginning of the exercise and merge back what we had given you.
7. Add the implementation for the functions under hardware/interfaces/opersys/2.0/default/Opersys.cpp that we hadn't given you an implementation.

6. Rework the com_android_server_OpersysService.cpp file to add:
    1. JNI calls to receive from the Java side
    2. Calls to the newly extended HIDL interface

7. Go back to OpersysService.java and extend it to now call on the new JNI calls added to com_android_server_OpersysService.cpp.

8. You should now be able to rebuild your AOSP and test your extended calls by using the “service” command line tool with the extend AIDL.

Extra:

These exercise will have you extend your system service to make it act like a real system service that reacts to hardware events by sending an Intent that might be receive by or activate an application.

9. 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.

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

11. Build your AOSP and test your additions.

AOSP sources and symbolic debugging

In this section you will learn how to:

- Import the AOSP as a project into Android Studio
- Use Studio to explore the AOSP
- Debug the AOSP's java code using Studio
- Use gdb/gdbserver to debug C/C++ code
- Step from a Java system service all the way down to a HAL module with Studio and gdb

1. Generate and run the idegen script:

```
$ cd ~/android/android-11.0.0_r3
$ . build/envsetup.sh
$ lunch 31
$ make idegen && development/tools/idegen/idegen.sh
```

This should result in your having a "android.ipr" file at the top level of your AOSP. Note that the build may complain about permission issues with some "root" directories. You can ignore this.

2. Add the Android's sources as a Java project by opening the "android.ipr" file as existing project in Android Studio. A few gotchas:

- You must make sure you have the SDK support for API 30
- Watch out for memory limit messages. You may need to increase the size of the heap attributed to Androi Studio. Click on the warning message to be taken to the proper menu.
- Watch out for filesystem watch limit. You may need to increase the watch limits by following the steps outlined in the link in the error message.
- Ignore messaging regarding git indexing.
- Accept converting the android.ipr to a new format if prompted.
- Do NOT accept migrating the project to Gradle if prompted.

3. Explore Android Studio's Java sources browsing capabilities (i.e. try out the shortcuts suggested a developer.android.com).

4. With the system_server process selected in the Android Device Monitor, set a breakpoint in frameworks/base/services/core/java/com/android/server/statusbar/StatusBarManagerService.

java:expandNotificationsPanel(). Use “service call statusbar 1” to expand the status bar. Eclipse should now break into the debug view at the breakpoint you selected. Now, you can step in the Status Bar Manager's code for expanding the notifications panel.

5. Do the same as the previous exercise but with OpersysService.java. Use “service call opersys 2 s16 foobar” to send the “foobar” string to the Opersys system service. User “service call opersys 1 i32 45” to read 45 bytes from the Opersys system service.

6. Use the “gdbclient.py” on the host to step from the JNI side of the system service all the way down the driver call. In other words, add a breakpoint to init_native(), read_native() and write_native() and follow the calls all the way to the corresponding open(), read() and write() occurring in the opersyshw HAL. For Android 11, there is a problem with the default gdbclient provided in the AOSP and you will need to replace it for debugging to work:

```
$ cd prebuilts/misc/gdbserver/android-x86_64/
$ mv gdbserver64 gdbserver64-orig
$ wget https://opersys.com/downloads/gdbserver64-10.0.0_r39
$ mv  gdbserver64-10.0.0_r39  gdbserver64
$ chmod 755 gdbserver64
$ croot
$ cd out/target/product/generic_x86_64/
$ rm -rf system*
$ croot
$ make -j8
```

Once you restart your emulator, you should be able to use “gdbclient.py”:

```
$ gdbclient.py -n system_server
```

7. Full system service debugging:

- With the system_server process attached to using Android Studio, set breakpoints in the Opersys system service added earlier, namely in read() and write()
- Use gdb to set breakpoints in the system service's JNI code: init_native(), read_native() and write_native()
- Have the system service invoked using “service call opersys ...”
- Step through the code starting from the system service's java code all the way down to the HAL module's writing to the device driver.

Tracing with ftrace

In this section you will learn how to work with ftrace and Android's integration of it to get detaild information about the system's operation.

1. Use ftrace to monitor the following:
   - Scheduling change events
   - CPU frequency scaling events
   - All events occurring while the "system_server" process is scheduled

2. Modify the circular character driver to add a static tracepoint to monitor each read() and write() operation to it. Use trace_printk() to achieve that, remember that you'll have to use MODULE_LICENSE("GPL") in order to have access to this symbol. Rebuild your driver and reload it on the device. Start ftrace tracing and make sure you can see the output in the ftrace output at runtime when you do a "cat" or "echo" as above.

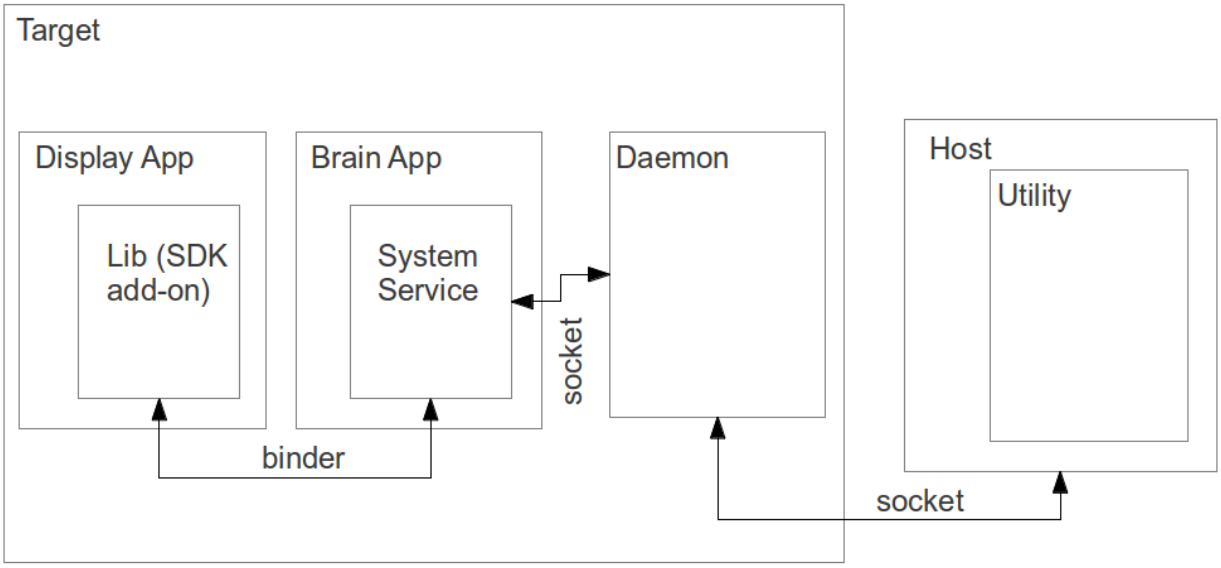
3. Instrument the native parts of the Opersys system service to log to ftrace's "trace_marker" file. Namely use the ATRACE_* macros to instrument (use "godir" to find the files):
   - opersyshw.c
   - Opersys.cpp
   - com_android_server_OpersysService.cpp

4. Update your target and rerun it with ftrace enabled and check that the instrumentation added in the previous exercise is reflected in the traces generated. Make sure you review the slides on how to enable tracing for the AOSP parts.

5. Instrument the Java side of the system service, namely OpersysService.java, to log to ftrace's trace_marker file.

6. Update your system and check that you can use ftrace to follow all the tracepoints from the java layer, to the native layer, and into the driver.

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