Embedded Android+Automotive

Workbook

Version v 12.0.4.1

March 2022

Table of Contents

1 Preparation

At this point, you should have:

- Electronic (PDF) copies of the slides, workbook and solutions
- A laptop or PC with a recent HTML 5 capable browser, such as Chrome/Chromium or Edge
- A link to a cloud instance

The cloud instance is running Ubuntu 20.04 and has a web interface. Check that you can log on using the link, user name, and password given to you by the instructor

1.1 Guacamole

The web interface is provided by Apache Guacamole https://guacamole.apache.org/. There is a crib sheet showing how to access the Guacamole navigation menu so that you can copy files to/from the server in `$HOME/android/guacamole-file-transfer.pdf`

1.2 Text editors

The following editors are already installed:

Command-line editors

- **vim** - a great editor, so long as you are already familiar with vim
- **nano** - simpler and easier to use than vim. Does everything you will need during this training course

Graphical editors: **gedit**. Launch it via the file browser, or from the command line:
`gedit [path to file to edit]`

You are welcome to install any other editors you want. One to consider is Visual Code. You can install it like this:

```
$ sudo snap install code --classic
```

And run it like this:

```
$ code &
```


2 The Android Open Source Project

Objective

The cloud instance contains a copy of AOSP 12.0 in `$HOME/aosp`

In this lab you will start building Android for the Emulator target. Later on, you will get a chance to try it out

2.1 Build AOSP

Connect to the server and get the local manifest for marvin:

```
$ cd $HOME/aosp
$ git clone https://github.com/csimmonds/marvin-local-manifest .repo/local_manifests \
-b android12
$ repo sync -c
```

The repo sync will take a few minutes. Afterwards, check that device configuration files for marvin have been cloned to `device/sirius`

```
$ ls device/sirius/marvin
AndroidProducts.mk  BoardConfig.mk  device.mk  init.cutf_cvm.rc  marvin.mk
```

Note: you will find that there is another device in `device/sirius/marvincar`. This is marvin configured for Android Automotive OS.

Next, select a device configuration

```
$ cd $HOME/aosp
$ source build/envsetup.sh
$ lunch
```

Note that the first time you run lunch it may take short while, maybe a minute, to complete

Select **marvin-userdebug** from the lunch menu

Finally, begin the build using the command "m" (which is a shortcut for "make")

```
$ m
```

The build will take about three hours

Please indicate to the instructor when you you have started building so that he can continue the lecture while the build is taking place

2.2 repo and lunch

While the system build is still running you can look at the source code in a different terminal

Open a new terminal window and change to the `aosp` directory. Take a look at the first few lines of `.repo/manifest.xml`, and also the file it includes, `.repo/manifests/default.xml`. This confirms the version of AOSP that was fetched

Run the command `repo info` to get a report of each individual project

Set up the AOSP environment in this terminal:

```
$ source build/envsetup.sh
$ lunch
```

- Use the command **godir** to find the file `SurfaceFlinger.cpp` in the framework (ignore the one in `device/generic/vulkan-cereal/fake-android-guest`)
- Use **cgrep** to find all C/C++ files in SurfaceFlinger that contain a main() function
- Use **mgrep** to find build files that include the string "protobuf"
- Type **croot** to go to the Android root and search again

2.3 The build ID

Use the command `printconfig` to find the build ID for the AOSP code base you have.

You will find the build ID in `build/core/build_id.mk`

3 Device configuration

Objective

The AOSP build for the target should have completed by now - if not we will skip this lab and return to it later. But, if everything is done, now is a good chance to take a look at the results of the build

3.1 Image files

Open a terminal window and run lunch

Useful shortcut: there is a script in the android directory that will run lunch for you and select the last build target. This will save some typing over the next few days:

```
$ cd $HOME/aosp
$ cp ../android/relunch .
```

Then use it like so:

```
$ cd $HOME/aosp
$ . relunch
0: marvin-userdebug
[...]
```

The files generated for the target are in `out/target/product/[PRODUCT_DEVICE]` where `PRODUCT_DEVICE` is from your device makefile. In your case you have `device/sirius/marvin/marvin.mk` which contains

```
PRODUCT_DEVICE := marvin
```

Take a look in directory `out/target/product/marvin`

Note that shell variable `$OUT` contains the same path

3.2 List packages

Which files are installed?

The files installed in the system and vendor partitions are listed in

`$OUT/installed-files.txt`

and

`$OUT/installed-files-vendor.txt`

Which packages are installed?

The complete list of packages is contained in build variables `PRODUCT_PACKAGES`, `PRODUCT_PACKAGES_DEBUG` and `PRODUCT_PACKAGES_ENG`

Use `get_build_var` to display each of these variables:

```
$ get_build_var PRODUCT_PACKAGES
$ get_build_var PRODUCT_PACKAGES_DEBUG
$ get_build_var PRODUCT_PACKAGES_ENG
```

The output is not very easy to read, but there is a script that prints them out one per line in `$HOME/android/list-product-packages.sh`

Try it out. You can get a count of packages using this command:

```
$ $HOME/android/list-product-packages.sh | wc -l
```

You can switch to the directory containing a package using

```
$ gomod [module name]
```

Check that the package **adbd** is installed and then find the directory which contains the source code

3.3 View the build log

Take a look at the build log

```
$ croot
$ gzip -cd out/verbose.log.gz | less -R
```

3.4 (Optional) generate a product makefile dependency graph

Generate the graph:

```
$ cd $HOME/aosp
$ make product-graph
```

If you see this error...

```
Expected "out/target/product/marvin/.installable_files" to be readable
```

... then re-build (type m, it will take only a few seconds) and then try again to build the graph

Then convert it to pdf and view using evince:

```
$ dot -Tpdf -Nshape=box -o out/products.pdf out/products.dot
$ evince out/products.pdf
```

Hint: scroll down to the bottom and then work upwards

3.5 Run Cuttlefish

Launch the Cuttlefish Virtual Device (CVD) with VNC remote desktop:

```
$ cd $HOME/aosp
$ launch_cvd -start_vnc_server
```


You will know that the device is fully booted when you see:

```
init: starting service 'adbd'...
Unable to connect to vsock server: Connection reset by peer
transport message failed, response body:
VIRTUAL_DEVICE_BOOT_STARTED
VIRTUAL_DEVICE_NETWORK_MOBILE_CONNECTED
VIRTUAL_DEVICE_BOOT_COMPLETED
Virtual device booted successfully
```

Although, you can connect to is with ADB as soon as you see `init: starting service 'adbd'...`

In another terminal, source 'relunch' and then check that the CVD is running:

```
$ cd $HOME/aosp
$ . relunch
$ cvd_status
cvd_status I 01-14 15:58:42 116026 116026 cvd_status.cc:109] run_cvd is active.
```

Check that there is an ADB device available

```
$ adb devices
List of devices attached
0.0.0.0:6520    device
```

Now launch ADB:

```
$ cd $HOME/aosp
$ adb shell
marvin:/ $
```

Read the build description:

```
marvin:/ $ getprop ro.build.description
marvin-userdebug 12 SQ1A.220105.002 eng.chris.20220208.155855 test-keys
```

Exit the ADB shell by typing "exit" or CTRL-D

Next, check that you can use a VNC client to view the screen:

```
$ cd $HOME/aosp
$ java -jar $HOME/tightvnc-jviewer.jar -ShowControls=No -ScalingFactor=75 \
-showConnectionDialog=No localhost 6444
```

The window may appear blank to start with. Just click anywhere in the window to get the VNC server to refresh the screen

Open the apps draw (swipe up from the bottom of the home screen)

Click on the Settings app and scroll down to About phone

Note that the device name is "Paranoid android"

Close the window containing the VNC viewer

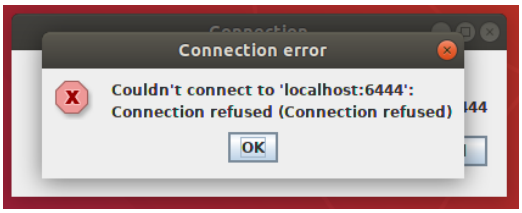
Stop the CVD:

```
$ stop_cvd
```

To make life easier there is a script that launches the CVD and the vnc viewer all in one go:


```
$ $HOME/android/run-cvd-and-vnc
```

It starts the VNC viewer after 8 seconds, which is usually enough time. If not, you will see this pop-up:



Just wait a few moments and then press OK

3.6 (Optional) change the screen size

Find the current screen size and density:

```
$ adb shell
marvin:/ $ wm size
marvin:/ $ wm density
```

Stop Cuttlefish and then launch it again with difference sizes. For example, for a tablet you might try:

```
$ $HOME/android/run-cvd-and-vnc -x_res=1280 -y_res=800 -dpi=240
```

Note that the screen layout for Android is determined at run-time using the dimensions of the display

3.7 (Optional) Create a custom CVD

You can create a new CVD fragment as part of the device configuration for marvin. Create `device/sirius/marvin/config_marvin.json` and enter the screen size and also the memory size, like this:

```
{
        "x_res" : 1280,
        "y_res" : 800,
        "dpi" : 240,
        "memory_mb" : 4096
}
```

Create `device/sirius/marvin/Android.bp` and create a module named `cvd_config_marvin.json` which will install the file above:

```
prebuilt_etc_host {
    name: "cvd_config_marvin.json",
    src: "config_marvin.json",
    sub_dir: "cvd_config",
}
```

Add this to the end of `device/sirius/marvin/device.mk`:


```
SOONG_CONFIG_cvd_launch_configs += cvd_config_marvin.json
```

Build marvin with these changes

```
$ m
```

Now you can select the marvin CVD when you launch Cuttlefish:

```
$ $HOME/android/run-cvd-and-vnc -config=marvin
```

As an extra bonus, you can make this the default configuration by creating `device/sirius/marvin/android-info.txt` with this text in it:

```
config=marvin
```

Then, add this to `device/sirius/marvin/device.mk`

```
TARGET_BOARD_INFO_FILE := $(LOCAL_PATH)/android-info.txt
```

Build marvin again

```
$ m
```

Now when you launch Cuttlefish, marvin is the default:

```
$ $HOME/android/run-cvd-and-vnc
```

3.8 (Optional) Use WebRTC viewer

The VNC remote viewer is simple and easy to start and stop, but it does not allow any interaction with the target except mouse and keyboard. The web interface does have such things

Stop any CVDs that may be running

Launch cuttlefish with the WebRTC interface:

```
$ cd $HOME/aosp
$ launch_cvd -start_webrtc
```

Launch a browser (must be Chrome or Chromium)

Enter URL https://localhost:8443

Initially you see a page that says "Your connection is not private". Click the "Advanced" button, then click on "Proceed to localhost (unsafe)"

You should see a page with "Available devices" and "cvd-1". Click on the "Connect" button

You should see the device screen in the browser

Try clicking on the various control buttons to find out what they do

4 The kernel

Objective

At the moment, the target is using a prebuilt kernel. In this lab you will build the kernel from source.

The lab takes 60 to 90 minutes to complete

4.1 Kernel version and modules

Start Emulator

```
$ $HOME/android/run-cvd-and-vnc
```

In a different terminal, run lunch and select marvin-userdebug Log on to marvin by running an ADB shell:

```
$ adb shell
```

In the ADB shell, make a note of the version of the current prebuilt kernel:

```
$ uname -a
```

Also, make a note of the modules that are loaded:

```
$ lsmod
```

4.2 Get the kernel

There is a copy of the Android common kernel 5.10 in `$HOME/android-kernel`

It was obtained like this (note: this has already been done, so you don't need to run these commands):

```
$ cd $HOME
$ mkdir android-kernel
$ cd android-kernel
$ repo init -u https://android.googlesource.com/kernel/manifest -b common-android12-5.10
$ repo sync -c
```

4.3 Build the kernel

The kernel build environment is separate from the AOSP build, so it is important to create a separate terminal window for the parts of this lab that relate to the kernel

In a **new terminal window**, go to directory `$HOME/android-kernel`, and build a Generic Kernel Image for x86_64 (takes about 10 minutes - time for a tea break):

```
$ cd $HOME/android-kernel
$ BUILD_CONFIG=common/build.config.gki.x86_64 \
LTO=none build/build.sh
```


Check that `out/android12-5.10/dist` contains the kernel (bzImage) and one generic module (virtio_mem.ko)

Next, build the kernel modules for the generic virtual device (takes about 10 minutes. Sorry. More tea?):

```
$ BUILD_CONFIG=common-modules/virtual-device/build.config.virtual_device.x86_64 \
LTO=none build/build.sh
```

Check that now `out/android12-5.10/dist` contains many modules (actually 61)

Put a copy of the kernel binary into the marvin device directory:

```
$ cp $HOME/android-kernel/out/android12-5.10/dist/bzImage $HOME/aosp/device/sirius/marvin
```

And also copy the kernel modules, like this:

```
$ mkdir $HOME/aosp/device/sirius/marvin/ko
$ cp $HOME/android-kernel/out/android12-5.10/dist/*.ko $HOME/aosp/device/sirius/marvin/ko
```

4.4 Integrate the kernel into your device

Now you need to make some changes to the configuration of marvin. Switch to the other terminal that is set up for AOSP builds (i.e. one where you have run lunch)

At the moment, the emulator is using a prebuilt kernel. Use this command to find out where it comes from:

```
$ cd $HOME/aosp
$ ../android/list-product-copy-files.sh | grep kernel
```

Take a look at the slide *Adding the kernel to boot.img* and add a `PRODUCT_COPY_FILES` rule at the **beginning** of `device/sirius/marvin/device.mk` that will copy **bzImage** to **kernel**

Check that it works:

```
$ ../android/list-product-copy-files.sh | grep kernel
```

You should see two lines with a copy target of kernel. Your addition must be the first one because that will override the second one.

In a similar way, the kernel modules are listed in a make variable. Use this command to show them:

```
$ get_build_var BOARD_VENDOR_RAMDISK_KERNEL_MODULES
```

The output is one long line. It helps to translate spaces to newlines:

```
$ get_build_var BOARD_VENDOR_RAMDISK_KERNEL_MODULES | tr " " "\n"
```

So, next you need to replace those modules with the ones from the kernel build by adding this line to the **end** of `BoardConfig.mk`

```
BOARD_VENDOR_RAMDISK_KERNEL_MODULES := $(wildcard device/sirius/marvin/ko/*.ko)
```


Use `get_build_var` again to check that `BOARD_VENDOR_RAMDISK_KERNEL_MODULES` only lists modules from marvin

At this point it can be useful to see the changes you have make to marvin using repo diff

```
$ croot
$ repo diff device/sirius/marvin
```

When everything looks good, build marvin (it will take only a few minutes this time):

```
$ m
```

Check that your kernel has been installed in $OUT:

```
$ ls -l $OUT/kernel
```

The size of that file should be the same as the bzImage file

Start the emulator:

```
$ $HOME/android/run-cvd-and-vnc
```

Run ADB:

```
$ adb shell
```

In the ADB shell, check that the kernel version is different from before:

```
$ uname -a
```

Side note about the date and time printed by uname:

The output looks something like this:

```
Linux localhost 5.10.66-android12-9-00018-g87a74496ed4a #1 SMP PREEMPT Wed Jan 12 17:15:55 U
TC 2022 i686
```

But you may notice that the date and time do not match the time that you built the kernel. The reason is "reproducible builds", which are described here: https://www.kernel.org/doc/html/v5.14-rc2/kbuild/reproducible-builds.html. The idea is to create the same binary each time the kernel is built, and amongst other things that means that the data/time stamp must not change. So, you can set shell variable `KBUILD_BUILD_TIMESTAMP` to the timestamp you want to put into the kernel binary

Looking at the code in `android-kernel/build/_setup_env.sh`, you can see

```
export SOURCE_DATE_EPOCH=$(git -C ${ROOT_DIR}/${KERNEL_DIR} log -1 --pretty=%ct)
export KBUILD_BUILD_TIMESTAMP="$(date -d @${SOURCE_DATE_EPOCH})"
```

The first command expands to be

```
$ git -C ~/android-kernel/common log -1 --pretty=%ct
```

... which is the date/time stamp of the last commit to the kernel tree

One final thing: `%ct` prints the commit date, whereas when you do a `git log` you see the author date. They can be different. Try these commands that show the "author date" and the "commit time"


```
$ git -C ~/android-kernel/common log -1 --pretty=%aD
Mon, 19 Jul 2021 17:12:55 +0900
$ git -C ~/android-kernel/common log -1 --pretty=%cD
Tue, 20 Jul 2021 11:40:01 +0900
```

Read the man page for git log for full details

5 Bootloaders

Objective To look at the Cuttlefish bootloader, U-Boot

5.1 The bootloader

Launch the CVD so that it stops in the bootloader and also enable the serial console:

```
$ launch_cvd -console=true -pause-in-bootloader=true
```

In another terminal, verify that the console device exists

```
$ ls -l $HOME/cuttlefish_runtime/console
lrwxrwxrwx 1 chris chris 10 Dec 16 15:02 /home/chris/cuttlefish_runtime/console -> /dev/pts/4
```

You need a terminal emulator program to attach to the console. The examples here use **screen**:

```
$ screen $HOME/cuttlefish_runtime/console
```

When you press return you should get a U-Boot prompt: "=>"

Using screen:

The hotkey sequence is Ctrl-A followed by a character

[press the Ctrl and a keys together, then release both. Then press the action character]

```
Ctrl-A ?   Help
Ctrl-A \   Terminate screen
```

The scroll history buffer (scrollback mode) in screen is a bit strange. You have to :

- type Ctrl-A Esc
- Press the "Up" and "Down" arrow keys or the "PgUp" and "PgDn" keys to scroll through previous output
- Press "Esc" to exit scrollback mode

Get some basic information about U-Boot

```
=> version
U-Boot 2021.01-07989-g74c21be757 (May 10 2021 - 19:57:34 +0000)

Android (6443078 based on r383902) clang version 11.0.1 (https://android.googlesource.com/to
olchain/llvm-project b397f81060ce6d701042b782172ed13bee898b79)
GNU ld (binutils-2.27-bd24d23f) 2.27.0.20170315
=> printenv
bootargs=console=hvc0 panic=-1 earlycon=uart8250,io,0x3f8 pci=noacpi reboot=k audit=1
bootcmd=boot_android virtio 0#misc
bootdelay=-1
fdtaddr=0x40000000
fdtcontroladdr=77fa9150
stderr=serial
stdin=serial
stdout=serial
```


Disks and partitions

```
=> virtio info
Device 0: 1af4 VirtIO Block Device
            Type: Hard Disk
            Capacity: 13585.3 MB = 13.2 GB (27822720 x 512)
Device 1: 1af4 VirtIO Block Device
            Type: Hard Disk
            Capacity: 1.0 MB = 0.0 GB (2176 x 512)
Device 2: 1af4 VirtIO Block Device
            Type: Hard Disk
            Capacity: 2048.0 MB = 2.0 GB (4194304 x 512)
```

Partition table

```
=> virtio part 0

Partition Map for VirtIO device 0  --   Partition Type: EFI

Part    Start LBA       End LBA         Name
        Attributes
        Type GUID
        Partition GUID
  1     0x00000028      0x00000827      "misc"
        attrs:  0x0000000000000000
        type:   0fc63daf-8483-4772-8e79-3d69d8477de4
        type:   linux
        guid:   9eb7b2f6-102a-0546-9674-354187283447
[...]
```

You should find these partitions

```
Device 0
  1  misc
  2  boot_a
  3  boot_b
  4  vendor_boot_a
  5  vendor_boot_b
  6  vbmeta_a
  7  vbmeta_b
  8  vbmeta_system_a
  9  vbmeta_system_b
  10 super
  11 userdata
  12 metadata

Device 1
  1  uboot_env
  2  frp
  3  bootconfig

Device 2
  1  00000000-01
```

Boot Android

```
boot
```

Note that Linux boots and you will see the kernel messages

You will get a prompt

```
console:/ $
```


6 Boot storage layout

Objective To look at the disk format of the Cuttlefish target

Start an ADB shell

List partitions

```
# ls /dev/block/by-name/
boot_a      frp       super      vbmeta_a         vbmeta_system_b  vendor_boot_a
boot_b      metadata  uboot_env  vbmeta_b         vda              vendor_boot_b
bootconfig  misc      userdata   vbmeta_system_a  vdb
```

List logical partitions in the super partition

```
# lpdump
Slot 0:
Metadata version: 10.2
Metadata size: 1480 bytes
Metadata max size: 65536 bytes
Metadata slot count: 3
Header flags: virtual_ab_device
Partition table:
------------------------
  Name: product_a
  Group: google_system_dynamic_partitions_a
  Attributes: readonly
  Extents:
    0 .. 425583 linear super 2048
[...]
```

Logical partitions

```
product_a
product_b
system_a
system_b
system_ext_a
system_ext_b
odm_a
odm_b
vendor_a
vendor_b
vendor_dlkm_a
vendor_dlkm_b
odm_dlkm_a
odm_dlkm_b
```

Active slot

```
# getprop ro.boot.slot_suffix
```

While looking at the screen via VNC, boot into recovery mode

```
# reboot recovery
```


7 ADB and logcat

Objective

To configure ADB to work with the target board and then experiment with viewing logs using logcat.

This lab will take about 30 minutes

7.1 ADB

ADB is one of the tools that is created when you build AOSP. You will find it in `$HOME/out/host/linux-x86/bin/adb`. Make sure that you have run "lunch" in your current shell in order to pick up this version.

Check that the target is connected:

```
$ adb devices
```

Run a shell on the target

```
$ adb shell
```

You will get a shell prompt, `$`. To get a root prompt, type su

Try a few Linux commands:

```
$ su
# pwd
# ls -l
# df
# cat /proc/cpuinfo
# cat /proc/meminfo
# cat /proc/version
```

Type `exit` or Ctrl-D to terminate the shell.

Try restarting adb as root so that you automatically get a root shell next time:

```
$ adb root
restarting adbd as root

$ adb shell
# id
uid=0(root) gid=0(root) groups=0(root),1004(input),1007(log),1011(adb),10
[...]
```

7.2 Using ADB to install an app

There is a useful monitor app on the server: `$HOME/android/com.eolwral.osmonitor_90.apk`

> Background information: this was downloaded from https://f-droid.org/en/packages/com.eolwral.osmonitor/

Use adb to install it

When you pull up the app draw you will see that **OS Monitor** has been added. Launch it and take a look at the information it provides

Note: this is a user app, installed into `/data/app`. It will be lost when you next over-write user-data.img

7.3 Logcat

Run **adb logcat -g -b all** to find out the names and sizes of the log buffers.

Use **adb logcat** to view the default logs. Touch the screen and observe any messages. For colour, try **adb logcat -v color**

Repeat, looking at all log buffers

Show the main and system logs filtered for error messages. What other class of messages is displayed as well?

Show the log filtered for SurfaceFlinger

Combine the two: show error messages from SurfaceFlinger

7.4 Background logging

It is good practice to keep a logcat session running while debugging. You can use this script:

```
$ $HOME/android/logcat-colour.sh
```


8 Android start-up

Objective

Look at the start-up scripts and, if there is time, install a new boot animation

This lab takes about 45 minutes

8.1 Start-up scripts

It would be nice to see the complete list of run command scripts that init parses. Normally you don't see them because kernel log messages are "rate limited" if too many are produced in a short time. You can disable kernel message rate limiting by adding `printk.devkmsg=on` to the kernel command line.

However, in the case of Cuttlefish, this is already done.

Remember that init writes messages to the kernel log buffer, which you can see using dmesg. Note that dmesg requires root access, so type this:

```
$ su
# dmesg | grep "init: Parsing"
```

How many files are parsed?

Among the files parsed should be **/init.${ro.hardware}.rc** Use `getprop` to find the value of `ro.hardware`

Explain where this value comes from

8.2 Boot scripts

The source for the hardware init script for marvin is stored in `device/sirius/marvin/init.cutf_cvm.rc`

Add a line to the **on boot** section that will write a text string to `/data/vendor/message.txt`

Build the target

Boot the target and check that the message has been written

8.3 Native services

Find the status of the native services (daemons):

```
# getprop | grep init.svc
```

8.4 Stop and start

Stop the core native services by typing

```
# stop
```


Note that zygote and surfaceflinger have stopped, as you can see by running this command again:

```
# getprop | grep init.svc
```

Now start Android:

```
# start
```

Note that the boot animation runs and the Android run-time starts up

8.5 (Optional) Boot animation

Search the Internet for boot animation files (there are lots!). Download one and put it into the device directory. The file can be installed on the target by adding to device.mk a `PRODUCT_COPY_FILES` rule like this::

```
PRODUCT_COPY_FILES += \
    $(LOCAL_PATH)/bootanimation.zip:product/media/bootanimation.zip
```

Note: if you don't have a connection to the Internet you can use `$HOME/android/bootanimation-demo.zip`

Question: where is the file for the default boot animation? You will need to look at the code for the bootanimation application to find the answer. Type `godir bootanimation` to find the code.

8.6 (Optional) Properties

Type `getprop ro.build.version.release`

Type `getprop` to list out all properties

Create a persistent property with `setprop persist.myprop test`. You can check that it is stored in `/data/property/persistent_properties` by using xxd to dump the file:

```
# xxd /data/property/persistent_properties
```


9 Android packages and modules

Objective

Write a simple "hello world" program and install it on the target

This lab takes about 45 minutes

9.1 Listing modules

You can get a list of all module names, defined in an Android.bp or Android.mk file anywhere in the aosp directory, using this command:

```
$ allmod
```

Look at the file and count how many there are.

Hint: `wc -l` is a quick way to count lines in a file

To find the code for a module, you can use `gomod`. Use it now to find module "logcat"

`allmod` reads `$OUT/module-info.json` to get information about modules. This file is created during the first build, but it is not updated during incremental builds. You use `refreshmod` to do that

9.2 Writing a Hello World module

It is normal to put modules and packages that are not part of AOSP in directory `vendor/[company name]/`. We will use the path `vendor/example/`

NOTE: don't confuse this directory with the vendor partition: they are not related at all. This directory is just a place to store your code. You decide which partition it will be installed into when you write the build file

So, in the aosp directory, create directory `vendor/example/helloworld`. In that directory, write your version of the traditional "hello world" program. In plain ANSI C it looks like this:

```
#include <stdio.h>

int main (void)
{
        printf("Hello, world!\n");
        return 0;
}
```

Using the examples on the slides, create an Android.bp file in the same directory that will build a binary module and install it in the **vendor** partition

Next, you need to add your module to device marvin. Edit `aosp/device/sirius/marvin/device.mk` and add `PRODUCT_PACKAGES += [name of your module]`

Check that your module is included by running

```
$ $HOME/android/list-product-packages.sh
```


If it is not listed, most likely there is a typing error in your change to `device.mk`

Build marvin (m)

Check that your module has been built and installed into `$OUT/vendor/bin`

Run the emulator and test that you can run your helloworld program from a **root** ADB shell

9.3 (Optional) Modify and sync

Building new images and booting the target is one way to get code onto the target, but it is quite slow, especially when you have to flash them into the memory or a real target. In this exercise you will use a shortcut where you sync changes to a module to the target without generating new images or even rebooting the target

Edit the string printed out by your helloworld

Build it, either by

- changing to the directory containing helloworld and typing `mm`
- using `mmm [directory name]` from any directory, for example:

```
$ croot
$ mmm vendor/example/helloworld
```

Now, with the target running, ask Android to remount the read-only partitions as writable filesystems:

```
$ adb root
$ adb remount
Using overlayfs for /system
Using overlayfs for /vendor
[...]
Now reboot your device for settings to take effect
remount succeeded
```

Ignore the message requesting that you reboot, it is not necessary when using Cuttlefish

Now you can use ADB to sync any modules you have compiled:

```
$ adb sync
/data/: 0 files pushed, 0 skipped.
/odm/: 0 files pushed, 11 skipped.
/product/: 0 files pushed, 329 skipped.
/system/: 240 files pushed, 1891 skipped. 0.0 MB/s (7931 bytes in 0.207s)
/system_ext/: 0 files pushed, 65 skipped.
/vendor/: 0 files pushed, 665 skipped.
```

Don't worry that the numbers reported by adb sync are larger that you would expect: everything is fine. Run the program in an ADB shell and verify that the modified version has been installed

Make another change. This time you only need to build and sync

```
$ cd vendor/example/helloworld
$ [edit helloworld.c]
$ mm
$ adb sync
```


This is much faster than doing a full build and reflash BUT remember: the images in `$OUT` still have the old versions

Use `gomod [your module]` to find your module. It will not be able to find it because the module database is out of date.

Update the database and check that your module is among them (refreshmod takes about two minutes):

```
$ refreshmod
$ allmod | grep [the name of your module]
```

Remember: `mm clean` *does not clean the module: it runs a global clean instead*. The correct command for a module named "helloworld" would be `m clean-helloworld`

9.4 (Optional) Android.mk

Create an Android.mk that will build the same program, but giving the module a different name. Check that it builds

Note: make sure there are no spaces at the end of the lines in Android.mk. If there are, you may get obscure error messages like this:

```
FAILED: ninja: '/helloworld.c', needed by 'out/target/product/marvin/obj/EXECUTABLES/
helloworld_intermediates/helloworld.o', missing and no known rule to make it
```

Test this program on the target, either using mm/adb sync or by adding it to `PRODUCT_PACKAGES` in device.mk

10 SELinux

Objective In this lab you will add a web server daemon to marvin and add the SELinux policy required to launch it as a native service

The lab will take about 60 minutes to complete

10.1 Adding sepolicy for a web server

Copy the web server project from `$HOME/android/httpd` to `vendor/example/httpd`. Add these lines to your device.mk file

```
PRODUCT_PACKAGES += \
    busybox
```

Type 'm' to do a full build

Launch Cuttlefish with SELinux in permissive mode:

```
$ $HOME/android/run-cvd-and-vnc -guest_enforce_security=false
```

`-guest_enforce_security=false` is so that httpd can run without the allow rules, which you will add in the last exercise of this lab

Start the Emulator and see if the web server is running (`ps -A | grep busybox`). Unfortunately, it is not. If you look at the log messages from init, you will see why:

```
# dmesg | grep httpd
init: Parsing file /vendor/etc/init/httpd.rc...
init: Could not start service 'httpd' as part of class 'main': File /vendor/bin/busybox(labe
led "u:object_r:vendor_file:s0") has incorrect label or no domain transition from u:r:init:s
0 to another SELinux domain defined. Have you configured your service correctly? https://sou
rce.android.com/security/selinux/device-policy#label_new_services_and_address_denials
```

So, you need to create sepolicy for `/vendor/bin/busybox`

Begin by making a place to put the sepolicy files:

```
$ mkdir device/sirius/marvin/sepolicy
```

Next, edit `device/sirius/marvin/BoardConfig.mk` and add

```
BOARD_SEPOLICY_DIRS += device/sirius/marvin/sepolicy
```

Now, following the examples on the slides, add the necessary policy file, httpd.te, to the sepolicy directory

Then add the `httpd_exec` context to the `file_contexts` in the same directory

Check that the SE policy has been updated using `ls -Z` to find the context of `/vendor/bin/busybox`

Check that busybox is running and that the SELinux type is httpd by typing `ps -AZ`

Use the web viewer shell (one of the icons at the bottom of the home screen) to load web page http://127.0.0.1/

10.2 audit2allow

Take a look at the avc messages created while httpd is starting and also while loading the web page

Pipe them to audit2allow. You should see that it prints out a section beginning

```
#============= httpd ==============
```

Copy the rules in this section to your httpd.te file and rebuild

11 Device configuration, part 2

Objective

Add an overlay that disables the lock screen

This lab will take about 30 minutes

11.1 Overlay

You may have noticed that if you run Cuttlefish twice without a rebuild in between the launcher starts with a lock screen and you have to swipe up from the bottom of the screen. In this exercise you disable it

You can see the current setting for the lockscreen by running this command in an ADB sehll:

```
# settings list secure | grep lockscreen
lockscreen.disabled=0
```

Use command `get_build_var DEVICE_PACKAGE_OVERLAYS` to obtain a list of the current overlay directories. You should find that there are two of them

Add an overlay directory in `device/sirius/marvin`

Edit device.mk and add it to `DEVICE_PACKAGE_OVERLAYS`

Create overlay path `frameworks/base/packages/SettingsProvider/res/values`

Create file `defaults.xml`

Add

```
<resources>
    <!-- Disable the lockscreen -->
    <bool name="def_lockscreen_disabled">true</bool>
</resources>
```

Run `get_build_var DEVICE_PACKAGE_OVERLAYS` again and verify that there are now three of them, with one being the new overlay you have just added

Build marvin and boot. Check that the lockscreen is not shown, even after the second boot

In an ADB shell, you should see this:

```
# settings list secure | grep lockscreen
lockscreen.disabled=1
```


12 The Android Framework

Objective

To create a new system service and test it from the command line using 'service call'

The service and manager will be installed in the `system_ext` partition

This lab takes about 60 minutes

12.1 Looking at services

Run an adb shell and get a list of services:

```
# service list
```

Find the **statusbar** service and note the binder interface (beginning with a capital 'I'). Find the corresponding AIDL source file (use godir). Note the first two interfaces.

You can call a service using the command `service call`. For example:

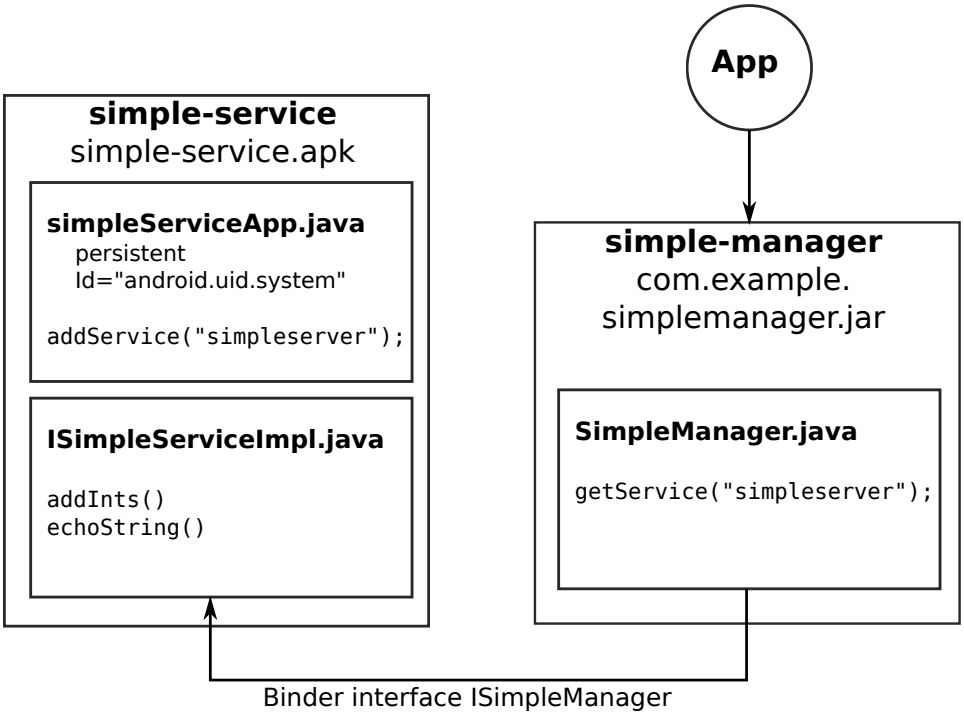
```
# service call statusbar 1
```

You may have to wait a few seconds before anything changes on the screen.

Try also `service call statusbar 2`

12.2 Create a binder interface

Creating a service involves several steps. First, we need to define the Binder interface, using AIDL, then create a manager shim for it, and finally implement the service itself. In this case, the service will be encapsulated in a system application that has UID system, which is necessary to register a service, and is persistent so that it will be started at boot-time. This diagram shows the components:



The interface to **simpleservice**, which was described in the slides, is defined in `$HOME/android/simple-manager`. Create directory `vendor/example/simple-manager` and copy these files into it.

Build the library with `mm`

Check that the extension library is installed in:
`$OUT/system_ext/framework/com.example.simplemanager.jar`

Check that the permissions file is installed in:
`$OUT/system_ext/etc/permissions/com.example.simplemanager.xml`.

12.3 Implement the service

Copy the code in `$HOME/android/simple-service` into `vendor/example/simple-service`

Build it with `mm`

Check that the simple-service app is installed in:
`$OUT/system_ext/app/simple-service/simple-service.apk`

Finally, add both the manager and service to your device configuration: look at the Android.bp files for simple-manager and simple-service. Add these packages to to your `device.mk` file

Build Android

12.4 Testing

We need the device to be running with selinux in permissive mode in order to start simpleservice. We will talk about selinux in a later chapter. For now, start the emulator like this

```
$ $HOME/android/run-cvd-and-vnc -guest_enforce_security=false
```


In an ADB shell, check the SELinux mode:

```
# getenforce
Permissive
```

Now list the services and check that verify that `simpleservice` is registered:

```
# service list
```

Use `service call` to call the functions with appropriate parameters

Check logcat for messages from simpleservice

```
# logcat -d | grep -i ISimpleServiceImpl
```

12.5 (Optional) Extend the service

Add a third function that returns the process id and thread id of the service stub

You can use these two functions from `android.os.Process`:

```
android.os.Process.myPid()
android.os.Process.myTid()
```

For simplicity, return the result as a string

You will need to declare the function in: `simple-manager/src/com/example/simplemanager/ISimpleManager.aidl`

Implement the function in the service: `simple-service/src/com/example/simpleservice/ISimpleServiceImpl.java`

Call the function from the manager: `simple-manager/src/com/example/simplemanager/SimpleManager.java`

Build and test

To verify that the result is correct,

```
# ps -A | grep simpleservice
```

Make a note of the PID, then look for the threads belonging to that PID

```
# ps -AT | grep [PID of simpleserver]
```

12.6 (Optional) Policy for simpleservice

With selinux in enforcing mode, simpleservice cannot run - as you can check for yourself

To make it work, you need extra sepolicy as described in the slides

Copy these two directories:

```
$ cd $HOME/aosp
$ cp -a ../android/solutions/selinux/simpleservice/sepolicy-private/ device/sirius/marvin
$ cp -a ../android/solutions/selinux/simpleservice/sepolicy device/sirius/marvin
```

Edit `device/sirius/marvin/BoardConfig.mk` and add


```
BOARD_PLAT_PRIVATE_SEPOLICY_DIR += device/sirius/marvin/sepolicy-private
```

Now build and test. You should find that simpleservice runs even when in enforcing mode

Notice that simpleservice app is running in the new domain:

```
# ps -AZ | grep simpleservice
u:r:simpleservice_app:s0    system  1366  265 [...]  com.example.simpleservice
```


13 Android applications and activities

Objective

Create an application that will call the simple-manager platform library

This builds on lab 12. If you did not complete lab 12, please use the worked solution from section 12 of the Solutions book

The lab takes about 60 minutes

13.1 Applications started at boot time

Find persistent applications:

```
$ adb shell dumpsys package packages > packages.txt
```

Then read the file and search for 'PERSISTENT'

13.2 Build the sample application

There is a skeleton app in `$HOME/android/simple-manager-app`. Copy it to `vendor/example`.

Use the trick mentioned in the slides to compile it with mm and sync it to the device. Reboot the Android UI

```
# stop;start
```

Now test that you can run it

What goes where?

Open an adb shell and look in these directories for components of your application:

- `/system_ext/app/[name of apk]` - the package file
- `/data/user/0/[name of package]` - data files (note that it contains a file named myfile.txt)

13.3 Platform libraries - simple manager

Note that com.example.simplemanager is a platform library:

```
# pm list libraries | grep simple
```

Now, add in the code so that the app can call simple-manager ...

Add to `Android.bp`:

```
libs: ["com.example.simplemanager"],
uses_libs: ["com.example.simplemanager"],
```

Add this to `AndroidManifest.xml`, as a child of the application tag:


```
<uses-library android:name="com.example.simplemanager" android:required="true"/>
```

Add to `src/com/example/simplemanagerapp/SimpleManagerActivity.java`

```
import com.example.simplemanager.SimpleManager;
[...]
public class SimpleManagerActivity extends Activity
{
    SimpleManager mSimpleManager;
[...]
    public void onCreate(Bundle savedInstanceState)
    {
        mSimpleManager = new SimpleManager();
[...]
```

Then, call `mSimpleManager.addInts`, `mSimpleManager.echoString` when the appropriate buttons are pressed

Build and test on the target

Check that you can call simple manager from the user interface of the app

If you added the third function function to simple manager, add code to call it from the app

14 Permissions and users

Objective

To add a permission to the simple-service and check for it when the service is called, then to request the permission to the test application

This builds on lab 13. If you did not complete lab 13, please use the worked solution from section 13 of the Solutions book

This lab takes 60 to 90 minutes

14.1 Permissions

Declare a permission in the AndroidManifest.xml of simple-service. Give it the name "com.example.simpleservice.SIMPLE" and set the protectionLevel to "dangerous"

Build the app and install on the target.

Check that the permission shows when you list the permissions using:

```
# pm list permissions -f
```

14.2 Check permissions

Add code to simple service to enforce the permission in addInts() and echoString(). Log the results.

Call simple-service from the command line and use logcat to see the result.

```
# service call simpleservice 1 i32 3 i32 6
# service call simpleservice 2 s16 "Hello world"
```

Repeat, but logged in as user "shell"

```
# su shell
$
```

(You can return to the root shell by typing exit, or Ctrl-d.)

Verify that you get an exception when calling the service

14.3 Requesting permissions

Use the simple-manager-app application to call simple-service. The app should fail the permission test

Next, add a `uses-permission` tag to the AndroidManifest.xml for the SIMPLE permission. Then, add the code to request the permission before calling `addInts` or `echoString`.

Build and install on the target

Check that it can call simple-service successfully.

Use the command "dumpsys package packages" and check that the app has the SIMPLE permission.

14.4 (Optional) User IDs

Start your hello world application, then open an adb shell and type `ps -A` to show a process listing.

Find your application and note that the user name is `u0_axx`

Locate the entry for your application in `/data/system/packages.xml`

Find the data directory: check that only this application has permission to create files in that directory.

14.5 (Optional) Group IDs

Request permission "android.permission.INTERNET" in the app. Build and install.

Check that when you run the application it belongs to group 3003

15 Hardware Abstraction Layers

Objective

To implement the light HAL for the Emulator

This lab will take 60 to 90 minutes to complete

Note: this lab does not work on Android 12

15.1 Listing HALs

Use the command `lshal` to list the HALs currently running. Note the two sections for "binderized" and "passthrough" HALs.

15.2 Implementing the Lights HAL

There is no implementation of the lights HAL for the target

Use hidl-gen to create the source (note: you MUST croot first):

```
$ croot
$ mkdir -p vendor/example/light
$ hidl-gen -L c++-impl -o vendor/example/light \
    -r vendor.example:vendor/example android.hardware.light@2.0
```

Note that files `Light.cpp` and `Light.h` have been created in `vendor/example/light`

Create an Android.bp:

```
$ hidl-gen -L androidbp-impl -o vendor/example/light \
    -r vendor.example:vendor/example android.hardware.light@2.0
```

Note that file `Android.bp` has been created in `vendor/example/light`

Next, you need to edit a few files to turn this into a working Lights HAL.

- Android.bp: change it to create a binary rather than a library
- Light.cpp: implement the setLight and getSupportedTypes methods
- service.cpp: create a new file that will register the Light service
- light-service.rc: create an init file to start the service as a native daemon
- selinux policy to allow init to launch the service
- vendor manifest: declare that ILight is implemented by the vendor

Lets do this one file at a time

Android.bp

The comments in the file tell you what to do in broad terms. Specifically, you should

- Change `cc_library_shared` to `cc_binary`

- Change the name from `android.hardware.light@2.0-impl` to `android.hardware.light@2.0-service.example`
- Change `proprietary: true,` to `vendor: true,`

service.cpp

There is a working service.cpp in `$HOME/android/light-hal/service.cpp` It is essentially the same as the code in the slide "Implementing the server"

In Android.bp

- add service.cpp to the `srcs` field
- add liblog to `shared_libs`

Check that you can build it with `mm`

Light.cpp

Add log messages so that you can see that the HAL is being called. Add

```
#define LOG_TAG "LightHAL"
#include <utils/Log.h>
```

Then write log messages using `ALOGI` in `setLight` and `getSupportedTypes`

You also want the framework to think that the backlight is implemented, so add this code to `getSupportedTypes`

```
    std::vector<V2_0::Type> types;
    types.push_back(V2_0::Type::BACKLIGHT);
    _hidl_cb(types);
    return Void();
```

Check that it still builds

light-service.rc

Copy light-service.rc from `$HOME/android/light-hal`

In Android.bp, add

```
    init_rc: ["light-service.rc"],
```

Vendor manifest

The light HAL needs an entry in the VINTF device manifest

Create a HAL manifest fragment in file `light-service.xml` with this content

```
<manifest version="1.0" type="device">
    <hal format="hidl">
        <name>android.hardware.light</name>
        <transport>hwbinder</transport>
        <version>2.0</version>
        <interface>
            <name>ILight</name>
            <instance>default</instance>
        </interface>
    </hal>
</manifest>
```


Add a reference to the fragment to your `Android.bp` like this:

```
vintf_fragments: ["light-service.xml"],
```

sepolicy

There is already a **te** file for the light HAL daemon in `system/sepolicy/vendor/hal_light_default.te`. Therefore, all you need to do is label the executable in `file_contexts` by adding this line:

```
/vendor/bin/hw/android.hardware.light@2.0-service.example u:object_r:hal_light_default_exec:s0
```

You can do this in two ways

1. The easy way

Change `file_contexts` in your device directory, as you did in the SELinux lab

2. The neat way

Keep the sepolicy local to the lights HAL by creating an sepolicy directory in in the light HAL and creating an `file_contexts` in there. Then edit BoardConfig.mk for your device and add the path to this new sepolicy directory to `BOARD_SEPOLICY_DIRS`

device.mk

Add the lights package to your device

```
PRODUCT_PACKAGES += android.hardware.light@2.0-service.example
```

Build and test

Run a top-level build (m) and write the new images to the target

Boot and run lshal and check that the lights HAL is registered

Look for messages from LightHAL in logcat

To test it, run the **Settings app**, select **Display** and then **Brightness level**. Check that you see log messages from your lights HAL as you move the slider

15.3 (Optional) Test using a simple test harness

Another way of testing is to write a test harness, such as the one in `$HOME/android/lighthaltest`

Copy that to vendor/example and add `lighthaltest` to device.mk. Build and test

Run lighthaltest and look at the logcat messages to check that your light HAL is behaving correctly

16 AIDL for HAL

Objective

To implement a lights HAL using a stable AIDL interface

This lab will take about 60 minutes to complete

16.1 AIDL Lights HAL

Take a copy of the default interface and put it in vendor/example:

```
$ croot
$ cd vendor/example
$ cp -a ../../hardware/interfaces/light/aidl/default light-hal-aidl
$ cd light-hal-aidl
```

Edit Android.bp and remove the `package` module.

Edit the `cc_binary` module to look like this:

```
cc_binary {
    name: "android.hardware.lights-service.marvin",
    relative_install_path: "hw",
    init_rc: ["lights-marvin.rc"],
    vintf_fragments: ["lights-marvin.xml"],
    vendor: true,
    shared_libs: [
        "libbase",
        "libbinder_ndk",
        "android.hardware.light-V1-ndk_platform",
    ],
    srcs: [
        "Lights.cpp",
        "main.cpp",
    ],

    overrides: ["android.hardware.lights-service.example"],
}
```

Note that there is an override which causes this module to be used in place of the default one

Rename lights-default.rc to lights-marvin.rc

Edit lights-marvin.rc to use the name of the executable:

```
service vendor.light-default /vendor/bin/hw/android.hardware.lights-service.marvin
    class hal
    user nobody
    group nobody
    shutdown critical
```

Rename lights-default.xml to lights-marvin.xml

Build the module and check that it has been installed

```
$ mm
$ ls -l $OUT/vendor/bin/hw/android.hardware.lights-service.marvin
```


Add sepolicy to `device/sirius/marvin/sepolicy/file_contexts`

```
/vendor/bin/hw/android.hardware.lights-service.marvin u:object_r:hal_light_default_exec:s0
```

Edit `device/sirius/marvin/device.mk` and add this new module

Rebuild marvin (m)

Launch the cvd and check

- it is listed when you run `ps -A`
- it is listed when you run `service list`
- you see log messages from the HAL in logcat

16.2 (Optional) Build and run the VTS for lights

Build the lights test unit

```
$ croot
$ cd hardware/interfaces/light/aidl/vts/functional
$ mm
```

Sync it to the target

```
$ adb root
$ adb remount
$ adb sync
```

Run the test in an ADB shell

```
# /data/nativetest/VtsHalLightTargetTest/VtsHalLightTargetTest
[==========] Running 6 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 6 tests from Lights/LightsAidl
[ RUN      ] Lights/LightsAidl.TestSupported/0_android_hardware_light_ILights_default
[       OK ] Lights/LightsAidl.TestSupported/0_android_hardware_light_ILights_default (0 ms)
[ RUN      ] Lights/LightsAidl.TestSupportedLightTypes/0_android_hardware_light_ILights_defa
[       OK ] Lights/LightsAidl.TestSupportedLightTypes/0_android_hardware_light_ILights_defa
[ RUN      ] Lights/LightsAidl.TestUniqueIds/0_android_hardware_light_ILights_default
[       OK ] Lights/LightsAidl.TestUniqueIds/0_android_hardware_light_ILights_default (0 ms)
[ RUN      ] Lights/LightsAidl.TestUniqueOrdinalsForType/0_android_hardware_light_ILights_de
[       OK ] Lights/LightsAidl.TestUniqueOrdinalsForType/0_android_hardware_light_ILights_de
[ RUN      ] Lights/LightsAidl.TestLowPersistence/0_android_hardware_light_ILights_default
[       OK ] Lights/LightsAidl.TestLowPersistence/0_android_hardware_light_ILights_default (
[ RUN      ] Lights/LightsAidl.TestInvalidLightIdUnsupported/0_android_hardware_light_ILight
[       OK ] Lights/LightsAidl.TestInvalidLightIdUnsupported/0_android_hardware_light_ILight
[----------] 6 tests from Lights/LightsAidl (3 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 1 test suite ran. (4 ms total)
[  PASSED  ] 6 tests.
```


17 Calling native code: JNI

Objective

To modify simple-service to call native methods that implement addInts and echoString.

17.1 Write the Java code

In simple-service, change ISimpleServiceImpl.java:

- Declare native methods `addIntsNative` and `echoStringNative`
- In your implementation of `addInts`, call `addIntsNative` and likewise with `echoString`
- Add a static method that loads the JNI library. Call it "simple-jni"

Check that you can compile it using mm

17.2 Write the C code

Create a directory in `vendor/example` for the C library

Create a C source file and implement functions for `addIntsNative` and `echoStringNative` using the JNI naming convention.

Implement the functions.

Create an Android.bp file to compile the shared library. The name of the module should be `libsimple-jni`

Compile the library using mm.

17.3 Test

Sync the changes to the target. Monitor the logcat and test simple-service using command line or using the app.

18 The Android graphics stack

Objective

Look at SurfaceFlinger and the configuration of OpenGL and EGL.

18.1 (Optional)Window manager

Use the wm command on the target to find the display size

18.2 (Optional) SurfaceFlinger

Using the command `dumpsys SurfaceFlinger`, find the following

- The name of the GLES driver, and the OpenGL version
- The refresh-rate
- Is h/w composer being used?
- How much memory is being used by graphic buffers?

Note the contents of directory `/vendor/lib/egl` and `/system/lib/egl`

19 Android Automotive

Objectives

To build an Android Automotive demo

This lab will take about 5 minutes to set up the build, and about 15 minutes afterwards to try out

19.1 Configuring Android Automotive

There are configuration files for device marvincar in `$HOME/aosp/device/sirius/marvincar` which were installed at the same time as marvin when you ran repo sync on the first day

Start the build

```
$ cd $HOME/aosp
$ source build/envsetup.sh
$ lunch
```

Select product **marvincar-userdebug**

Set the build running

Build it in the usual way:

```
$ m
```

The build will take about one hour and consume about 48 GiB of storage

When complete, check that the product has been generated in `$OUT`

19.2 Running Android Automotive

Start cuttlefish

```
$ $HOME/android/run-cvd-and-vnc
```

Now you will see the demo car launcher running as the home screen

Sometimes you get a pop-up with the text:

"This screen is for showing initial user notice and is not for product. Plz change config_userNoticeUiService in CarService before shipping"

The behaviour of this dialog is controlled by the Car Service configuration file, `packages/services/Car/service/res/values/config.xml` which we will look at in another chapter. For now, just click on "Do not show again"

On the next screen, "Reference Setup Wizard for Cars", click "Finish Setup"

Run an ADB shell


```
$ su
# pm list features
```

Note that this device has feature `android.hardware.type.automotive`. Locate the corresponding file in `/vendor/etc/permissions`

19.3 Users

In an ADB shell, run "ps -A" and note that there are processes with user ID U0 - the headless system user - and u10, which is the current user, "Driver"

Create a new user profile by clicking on the user icon, top right, then clicking on "Add profile". This creates a new user named "New profile" and makes it the current user

Run "ps -A" and note that there are processes with user ID "u11"

Next, change the user name by clicking on the settings icon, top left, then on "More", top right. Click on "Profiles and accounts". Click on "Rename" and enter your name

While you are there, you may want to set the unit for temperature to Celsius: click on "System" and then "Units" and then "Temperature"

You can get a list of users from CarUserService:

```
# dumpsys car_service --services CarUserService
```

Set the driving state to "driving":

```
# cmd car_service emulate-driving-state drive
```

Note that most of the app icons in the app draw are greyed out and clicking on one produces a message "[app] can't be used while driving"

19.4 Displays

Marvincar has two external displays. You can see the configuration here:

```
# getprop hwservicemanager.external.displays
```

How many displays are there altogether? The only real way to get all the information you need about displays is to use dumpsys display, but it does print out a lot of information. Try it out:

```
$ dumpsys display
```

For now, the useful parts are these: scroll back and check that you can see them:

```
[...]
LogicalDisplayMapper:
[...]
  Display 0:
    mDisplayId=0
    mBaseDisplayInfo=DisplayInfo{"Built-in Screen", [...]

  Display 2:
```



```
    mDisplayId=2
    mBaseDisplayInfo=DisplayInfo{"HDMI Screen", [...]

  Display 3:
    mDisplayId=3
    mBaseDisplayInfo=DisplayInfo{"HDMI Screen", [...]

  Display 4:
    mDisplayId=4
    mBaseDisplayInfo=DisplayInfo{"Cluster-App-VD", [...]
```

So, there are 4 displays with IDs 0, 2, 3, 4

The Cuttlefish remote desktop only shows one display, but you can still get a screen dump of all displays:

```
$ screencap -p -d 0 /data/local/tmp/disp0.png
$ screencap -p -d 2 /data/local/tmp/disp2.png
$ screencap -p -d 3 /data/local/tmp/disp3.png
$ screencap -p -d 4 /data/local/tmp/disp4.png
```

Now, exit adb, copy the screen captures and display them locally using eog (Eye of Gnome):

```
$ adb pull /data/local/tmp/
$ eog tmp
```

You can see the mapping between displays and occupant zones:

```
 $ dumpsys car_service --services CarOccupantZoneService
```

19.5 Kitchen Sink

In the app drawer, there is an app named "Kitchen Sink" which contains many test applets, some of which may be useful, some not

For example if you want to simulate notifications arriving at the car, click on NOTIFICATION, and select as source

Try some of the others ... you never know what you will find

20 The vehicle HAL

Objectives

To look at the vehicle HAL and vehicle properties, and to extend it by adding some properties of your own.

This lab will take about 60 minutes to complete

20.1 The vehicle HAL

Check that the HAL is running using the `lshal` command and look out for `android.hardware.automotive.vehicle`.

Note that the vehicle HAL is binderized

20.2 Vendor properties

We want to add a feature to marvincar and control it through the VHAL. The feature is the ability to make tea. There are three new vendor properties:

- `MAKE_TEA`: ID 0x2001; Android -> Car
  - initiates the tea making process
- `TEA_STATUS`: ID 0x2002; Car -> Android
  - reports one of `NOT_READY`, `BOILNG_WATER`, `BREWING`, `READY`
- `KETTLE_WATER_TEMPERATURE`: ID 0x2003; Car -> Android
  - reports the temperature of the water in degrees Celsius

There are two ways we could define new vendor properties:

- create a header file for C++ and a class for Java
- edit types.hal for the vehicle HAL

In this exercise, we will use the first technique. There is an optional exercise later on where you can try out the second approach

Begin by taking a copy of this skeleton code:

```
$ croot
$ cp -a $HOME/android/vhal-vendor vendor/example
```

The Android.bp file contains three modules:

- `marvin-vhal-headers`
  Contains all the header files in this directory, which will be needed by C++ code that references these properties
- `marvin-vhal-java`
  Same, but for Java

- `marvin-vhal`
  Contains code to add these properties to the property store

Add code to all three modules to define the 3 properties and initialize them. For each one you will need to consider the area (I suggest GLOBAL), the change mode, access and initial value

Add these modules to `PRODUCT_PACKAGES` for marvincar

In `hardware/interfaces/automotive/vehicle/2.0/default/`, add code to `Android.bp`, module `android.hardware.automotive.vehicle@2.0-service` that adds `marvin-vhal` to the list of `static_libs`. Also in the same module, add this line:

```
header_libs: ["marvin-vhal-headers"],
```

Then, edit `VehicleService.cpp` to include the headers for marvin-vhal:

```
    #include <marvin-vhal.h>
    #include <marvin-vhal-config.h>
```

... and call marvinVhalInit() like this:

```
    auto emulator = std::make_unique<impl::VehicleEmulator>(hal.get());
    marvinVhalInit(store.get());      <--  new code
    auto service = std::make_unique<VehicleHalManager>(hal.get());
```

Build marvincar

20.3 Testing

Boot the target

You can get a list of the all properties via `car_service`:

```
# dumpsys car_service get-carpropertyconfig
```

But, there is a problem: you probably don't know the full property ID (a 32-bit number). You only know the ID (a 16-bit number, which is the bottom 16 bits of the full ID)

One technique is to look for the ID followed by a comma, for example:

```
# dumpsys car_service get-carpropertyconfig | grep "2001,"
Property:0x21402001, Property name:0x21402001, access:0x3, changeMode:0x1, config:[], fs min:
0.000000, fs max:0.000000
```

Now you know that the property ID for `MAKE_TEA` is 0x21402001

Do the same for the other two properties

You can read a property in two ways:

```
# dumpsys car_service get-property-value 0x21402001
```

or

```
# lshal debug android.hardware.automotive.vehicle@2.0::IVehicle/default --get 0x21402001
```

You can set an integer value using the VHAL debug function


```
# lshal debug android.hardware.automotive.vehicle@2.0::IVehicle/default --set 0x21402001 i 1 a 0
```

You can't set a floating point value this way

20.4 Testing using SocketComm

There is a python module in `$HOME/android/marvin-vhal-test.py`

Copy it and run the test program:

```
$ croot
$ cp $HOME/android/marvin-vhal-test.py packages/services/Car/tools/emulator
$ cd packages/services/Car/tools/emulator
$ ./marvin-vhal-test.py
```

Set `MAKE_TEA` to 1 so that the tea making process runs:

```
# lshal debug android.hardware.automotive.vehicle@2.0::IVehicle/default --set 0x21402001 i 1 a 0
```

20.5 (Optional) vehiclehaltest

Copy the Vehicle HAL test program from `$HOME/android/vehiclehaltest` into `vendor/example`

Add vehiclehaltest to `PRODUCT_PACKAGES` in `device/sirius/marvincar/device.mk`. Build the images and run the target

Start an ADB shell, su to get a root prompt and then run vehiclehaltest

```
$ su
# vehiclehaltest
```

`vehiclehaltest` reads properties: `INFO_MAKE, GEAR_SELECTION, and ENV_OUTSIDE_TEMPERATURE`.

The values are returned by the default vehicle HAL server. The code is in `hardware/interfaces/automotive/vehicle/2.0/default/VehicleService.cpp` and the default values are in `hardware/interfaces/automotive/vehicle/2.0/default/impl/vhal_v2_0/DefaultConfig.h`

Running `vehiclehaltest -l` lists the property IDs that are emulated by the server. Running `vehiclehaltest -ll` lists more information.

20.6 (Optional) vendor properties in types.hal

In this exercise you will *modify the vehicle HAL* to add two new properties. This requires you to change a stable HAL interface and compute a new checksum, which will in turn trigger a complete rebuild of all the HALs, taking about 30 minutes. Only do this exercise if you have the time

Begin by editing `hardware/interfaces/automotive/vehicle/2.0/types.hal`. Add two properties to `enum VehicleProperty`

The first will be a static integer value. Set

- ID 0x1001

- VehiclePropertyGroup:VENDOR
- VehiclePropertyType:INT32
- VehicleArea:GLOBAL

The second will be a changing floating point value with

- ID 0x1002
- VehiclePropertyGroup:VENDOR
- VehiclePropertyType:FLOAT
- VehicleArea:GLOBAL

Now you need to set the default configuration for the new properties. Edit `hardware/interfaces/automotive/vehicle/2.0/default/impl/vhal_v2_0/DefaultConfig.h` and add these lines to the end of the array `kVehicleProperties[]`

```
    {.config =
         {
             .prop = toInt(VehicleProperty::VENDOR_STATIC),
             .access = VehiclePropertyAccess::READ,
             .changeMode = VehiclePropertyChangeMode::STATIC,
         },
     .initialValue = {.int32Values = {42}}},
    {.config =
         {
             .prop = toInt(VehicleProperty::VENDOR_VAR),
             .access = VehiclePropertyAccess::READ,
             .changeMode = VehiclePropertyChangeMode::ON_CHANGE,
             .minSampleRate = 1.0f,
             .maxSampleRate = 1000.0f,
         },
     .initialValue = {.floatValues = {42.0f}}},
```

Run a build by typing `m`. You will get an error similar to this:

```
[...]
ERROR: android.hardware.automotive.vehicle@2.0::types has hash d4b6d63051d037df4d9ce
0de5dd350ee662eb5aa9dd729559e44ca7c409295f6 which does not match hash on record. Thi
s interface has been frozen. Do not change it!
ERROR: Could not parse android.hardware.automotive.vehicle@2.0::IVehicle. Aborting.
10:45:21 ninja failed with: exit status 1
```

This is because `types.hal` has changed and so has a different hash value. The hashes are stored in `hardware/interfaces/current.txt`.

Next, update `current.txt` by appending the new hash value like this:

```
$ croot
$ hidl-gen -L hash -r \
android.hardware:hardware/interfaces android.hardware.automotive.vehicle@2.0::types >> \
hardware/interfaces/current.txt
```

Run the build again and you will find that the error has gone.

Boot the target and run `vehiclehaltest -l`. Note that the new properties with IDs 0x1001 and 0x1002 are listed.

20.7 (Optional) Read the properties

Extend `vendor/example/vehiclehaltest` to read the two vendor properties.

20.8 (Optional) Subscribe to a vehicle event

There is code in `$HOME/android/vhal-subscribe` which subscribes to a vehicle property. Copy it to `vendor/example`, compile it, and test it on the target.

20.9 (Optional) Generating fake changes to a property

The default vehicle property HAL service has a special test interface that modifies a property according to some rules.

The details are documented in
`hardware/interfaces/automotive/vehicle/2.0/default/impl/vhal_v2_0/DefaultConfig.h`

There is an example of how to use this in `$HOME/android/vhal-fakedata`. It sets `ENV_OUTSIDE_TEMPERATURE` to change every second starting at 10C and incrementing to 30C an then reset back to 10C

You can compile the program and run it on the target

21 The Car Service

Objectives

Look at the Car Service and test a Car application that reads the vendor properties that you created in the Vehicle HAL lab.

This lab will take about 30 minutes to complete

21.1 The car system service

Find `car_service` using command `service list`

You can get a lot of information about the car service via dumpsys:

```
$ dumpsys car_service --help
```

For example, you can list the services contained within `car_service`:

```
$ dumpsys car_service --list
```

You can get information about services with `--services [list of services]`, for example:

```
$ dumpsys car_service --services CarInputService
```

You can interact with some services: get a list of possibilities:

```
$ cmd car_service -h
```

Note: `dumpsys car_service -h` also works, but cmd is faster to type than dumpsys

21.2 A car application

Copy the demo car application from `$HOME/android/examplecar` to `vendor/example`.

Note that the application requires permission `CAR_VENDOR_EXTENSION`. Since that permission has level `signature|privileged`, the app is built with `LOCAL_PRIVILEGED_MODULE := true`

`AndroidManifest.xml`

```
<uses-permission android:name="android.car.permission.CAR_VENDOR_EXTENSION"/>
```

Build it and sync it to the target.

Touch the "All apps" icon and launch the app from there.

21.3 Reading vendor properties from an application

Extend the example Car application to read the VENDOR properties you defined in the Vehicle HAL lab

22 Appendix - setting up an AOSP build environment

22.1 Setting up the build environment

Make sure that you have a system capable of building AOSP in a reasonable amount of time, as described here...

https://source.android.com/source/building.html

...and here

https://source.android.com/source/initializing.html

22.2 Download AOSP

Install repo

```
$ curl https://storage.googleapis.com/git-repo-downloads/repo > $HOME/bin/repo
$ chmod a+x $HOME/bin/repo
```

Next, get the manifest for the version of AOSP you want. For this training course, we use android-12.0.0_r26

Note that the total amount of data to be downloaded is over 130 GiB. If you don't care too much about the git history you can reduce the download by about 60 GiB by doing doing a shallow clone by adding `--depth=1` to the repo init command

```
$ mkdir aosp
$ cd aosp
$ repo init -u https://android.googlesource.com/platform/manifest -b android-12.0.0_r26
```

Then, synchronise the components listed in the manifest. This will will take quite a long time, depending on your Internet bandwidth and whether you are doing a shallow clone or not:

```
$ repo sync -c
```

