CSE 506: Operating System
Semester Project Report

A brief tutorial to port an Android ROM for one device to another

Compiled by Aaswad Anil Satpute
(SBU ID: 109955001)
Under the guidance of Prof. Donald Porter

At Computer Science Department,
Stony Brook University,
NY

Table of Contents

Introduction

This is a tutorial for how to port a ROM (an Android OS) specifically made for one device to other device. In this tutorial we would port Android 5.0 named as Android Lollipop made for Google Nexus 5 to Google Nexus 7 (2012 model). There are few challenges involved here. One, the device I'm porting the ROM to is quite old. Two, The Device I'm porting from is a cellphone and the device I'm porting to is a tablet. During the whole process I failed many times, but due to the interest I had in this project I completed it single handedly. Even though I have tried to keep this tutorial as illustrative as possible, I couldn't capture the images of the device during some steps of the process because, unfortunately the device's screenshot feature doesn't work when the device is not switched on completely and the OS is running properly.

Disclaimer

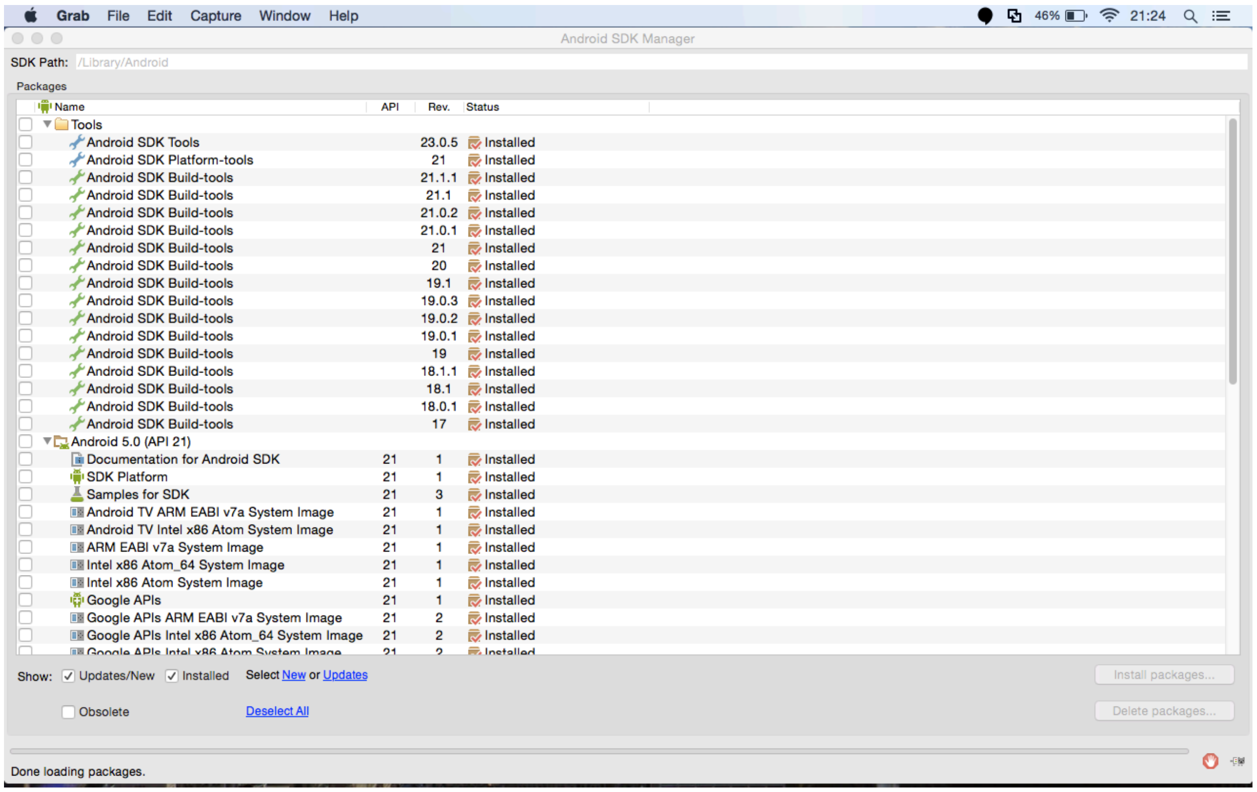
Perform this very carefully, if you want to try. This may brick your device and might void the warranty. If you wish to continue, continue at your own risk. I'm not responsible for any losses, if any, during the process.

All the trademarks and patents are owned by the respective owner. I don't claim to be owner of any of those.

Setting up the environment

We need to download all the drivers and setup the devices so that our process starts working out. For this we need to download Android SDK Manager from this link, http://developer.android.com/sdk/index.html.

For Mac we don't need the Google USB Drivers. Since I am working on my own Android app I have few extra SDKs installed as seen in the screenshot. Once the appropriate SDK for the platform is downloaded, we need to download the drivers, SDK Platform Tools and the SDKs that you need. This is illustrated in the image below.



SDK Path: /Library/Android

Packages

Name	API	Rev.	Status
Tools			
Android SDK Tools		23.0.5	Installed
Android SDK Platform-tools		21	Installed
Android SDK Build-tools		21.1.1	Installed
Android SDK Build-tools		21.1	Installed
Android SDK Build-tools		21.0.2	Installed
Android SDK Build-tools		21.0.1	Installed
Android SDK Build-tools		21	Installed
Android SDK Build-tools		20	Installed
Android SDK Build-tools		19.1	Installed
Android SDK Build-tools		19.0.3	Installed
Android SDK Build-tools		19.0.2	Installed
Android SDK Build-tools		19.0.1	Installed
Android SDK Build-tools		19	Installed
Android SDK Build-tools		18.1.1	Installed
Android SDK Build-tools		18.1	Installed
Android SDK Build-tools		18.0.1	Installed
Android SDK Build-tools		17	Installed
Android 5.0 (API 21)			
Documentation for Android SDK	21	1	Installed
SDK Platform	21	1	Installed
Samples for SDK	21	3	Installed
Android TV ARM EABI v7a System Image	21	1	Installed
Android TV Intel x86 Atom System Image	21	1	Installed
ARM EABI v7a System Image	21	1	Installed
Intel x86 Atom_64 System Image	21	1	Installed
Intel x86 Atom System Image	21	1	Installed
Google APIs	21	1	Installed
Google APIs ARM EABI v7a System Image	21	2	Installed
Google APIs Intel x86 Atom_64 System Image	21	2	Installed
Google APIs Intel x86 Atom System Image	21	2	Installed

Show: [✓] Updates/New [✓] Installed Select New or Updates Install packages...

[ ] Obsolete Deselect All Delete packages...

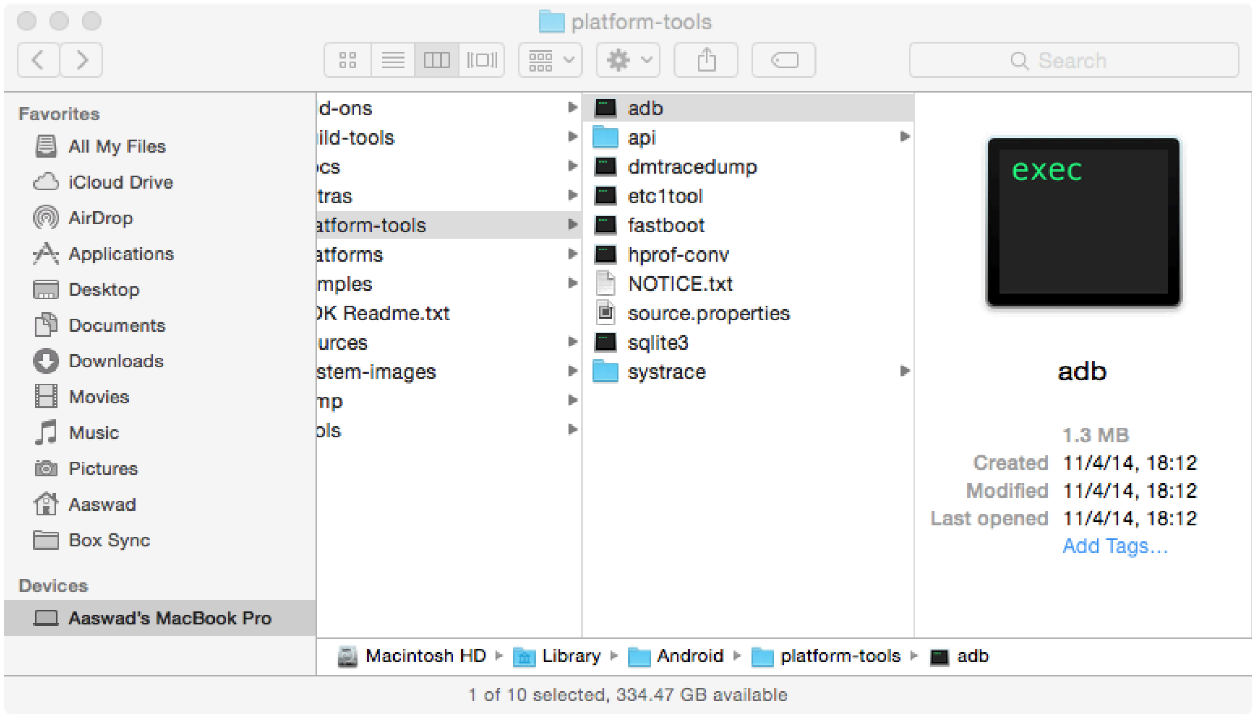
Done loading packages.

Android Debug Bridge

The adb is a tool that is located in <sdk>/platform-tools/. ADB is required for a computer to communicate to the android device. For nexus devices most of the drivers are usually packed into it. Android Debug Bridge (adb) is a versatile command line tool that lets you communicate with an emulator instance or connected Android-powered device. The following image illustrates the where to find adb.

When one starts an adb client, the client first checks whether there is any adb server process already running. If there is no server running, then it starts the server process. When the server starts, according to Google's page for android debug bridge, it binds to local TCP port 5037 and listens for commands sent from adb clients—all adb clients use port 5037 to communicate with the adb server. The server then sets up connections to all running emulator/device instances. It locates emulator/device instances by scanning odd-numbered ports in the range 5555 to 5585, the range used by emulators/devices. Where the server finds an adb daemon, it sets up a connection to that port. Note that each emulator/device instance acquires a pair of sequential ports — an even-numbered port for console connections and an odd-numbered port for adb connections.

At the same location one can find other useful programs or tools like "fastboot" which we would be using in further processes to flash the freshly created OS or ROM.



Enabling adb Debugging

In order to use adb with a device connected over USB, you must enable USB debugging in the device system settings, under Developer options. On Nexus 7, the Developer options screen is hidden by default. To make it visible, go to Settings > About phone and tap Build number seven times. Return to the previous screen to find Developer options at the bottom.

When you connect the Nexus 7 to your computer, the system shows a dialog asking whether to accept an RSA key that allows debugging through this computer. You need to accept it so that the adb from computer can start talking to the device.

Root access

Android is a very subtle variant of Unix. When someone starts up the device it doesn't have the root privilege. In other words all the applications and everything that the user works on and works with has only user privileges. Thus one can not get into kernel space. Rooting is the process of allowing users of smartphones, Tablet and other devices running the Android mobile operating system to attain privileged control (known as "root access") within Android's sub-system. Rooting gives the ability (or permission) to alter or replace system applications and settings, run specialized apps that require administrator-level permissions, or perform other operations that are otherwise inaccessible to a normal Android user. On Android, rooting can also facilitate the complete removal and replacement of the device's operating system, usually with a more recent release of its current operating system. As Android derives from the Linux kernel, rooting an Android device gives similar access administrative permissions as on Linux.

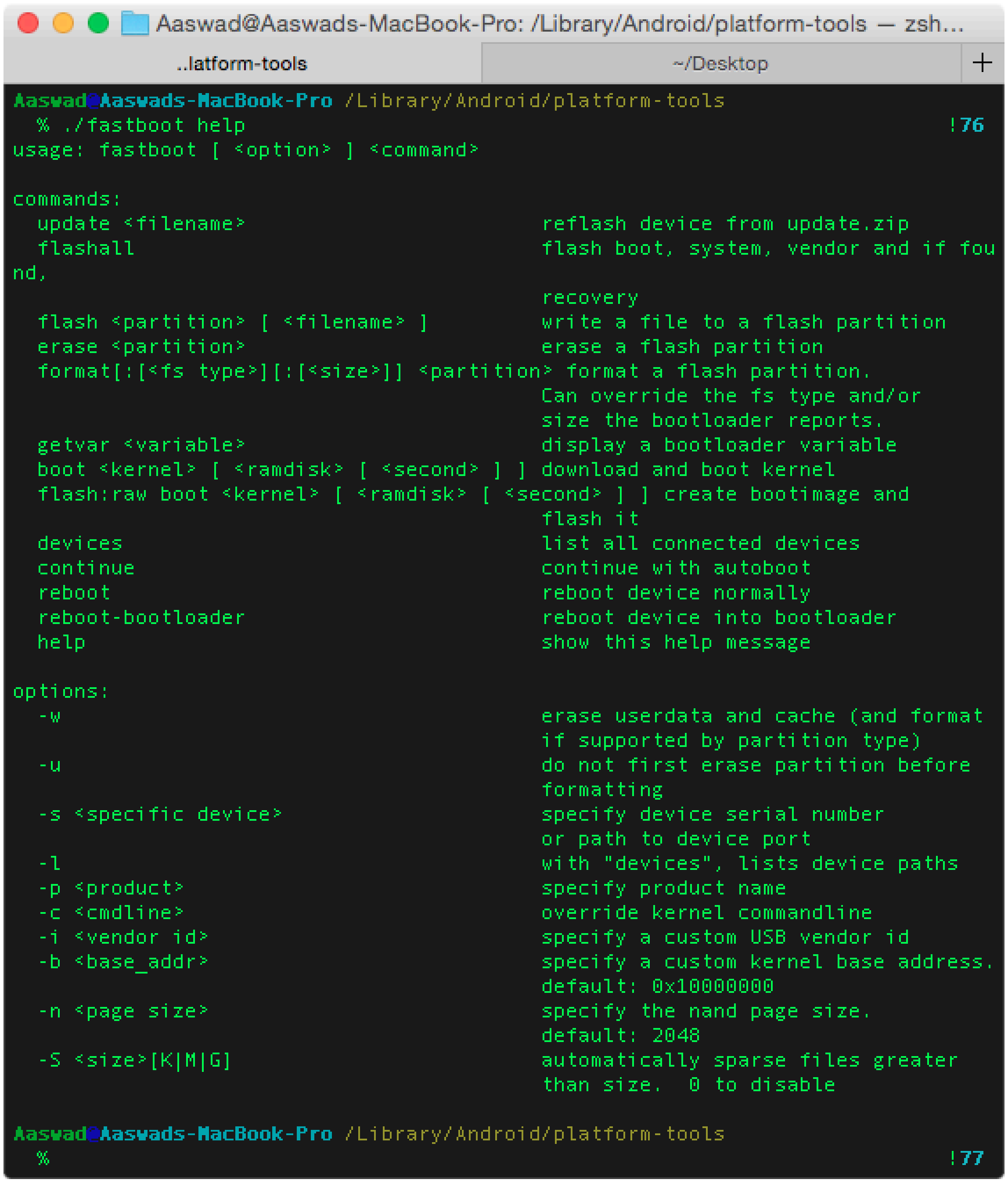
Because I need to edit the OS including the kernel, I need to get the root privileges of the device. For rooting we need to understand fastboot. This is what we would see in next part.

Fastboot

Fastboot is a acute variant of instant-on. It is the ability to boot nearly instantly, thus allowing to go online or to use a specific application without waiting for a PC's traditional operating system to launch. Instant-on technology is today mostly used on laptops, netbooks, and nettops because the user can boot up one program, instead of waiting for the PC's entire operating system to boot. Fastboot is a diagnostic protocol included with the SDK package used primarily to modify the flash file system via a USB connection from host computer. It requires that the device be started in a boot loader or Second Program Loader mode in which only the most basic hardware initialization is performed. After enabling the protocol on the device itself, it will accept a specific set of commands sent to it via USB using a command line.

We then boot into bootloader with the command "./adb reboot bootloader" in terminal from the same location where the adb and fastboot is located. It can also be loaded by switching off the device and then holding down the Volume Down button and the Power button.

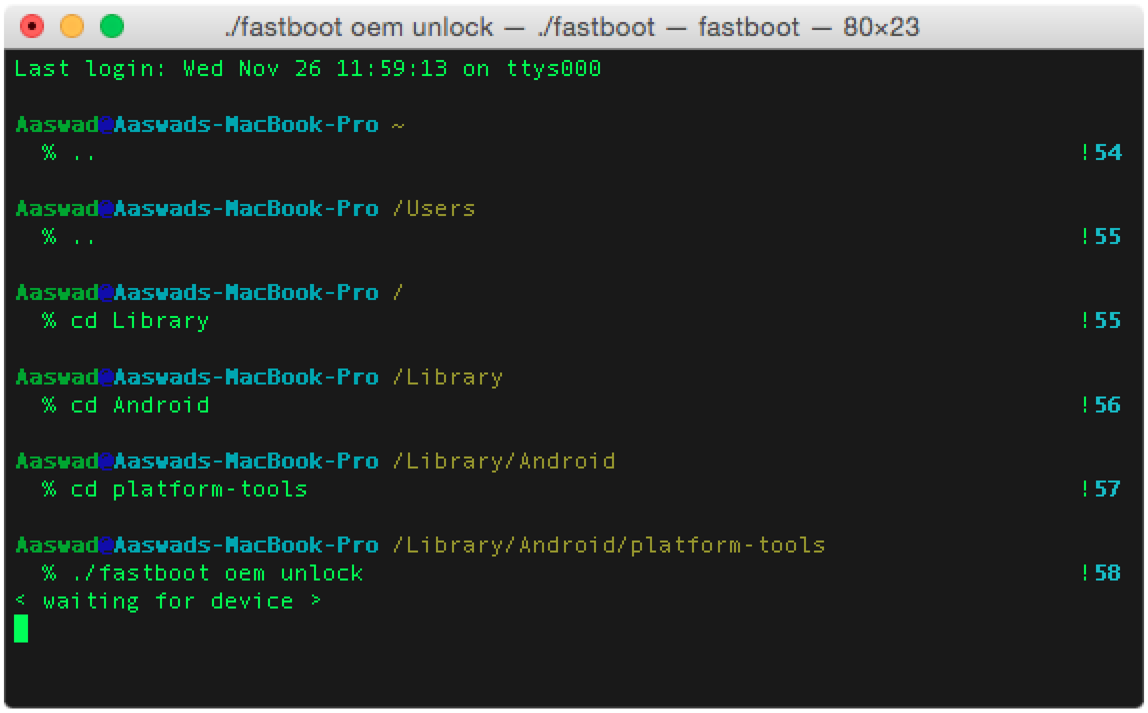
Fastboot for android is a very versatile and important tool. It can be used for flashing any ROM or OS or to clear data from some specific location. The following image shows what all is possible using fastboot.


Aaswad@Aaswads-MacBook-Pro: /Library/Android/platform-tools — zsh...
..latform-tools
~/Desktop

```
Aaswad@Aaswads-MacBook-Pro /Library/Android/platform-tools
  % ./fastboot help                                                       !76
usage: fastboot [ <option> ] <command>

commands:
  update <filename>                        reflash device from update.zip
  flashall                                 flash boot, system, vendor and if fou
nd,
                                           recovery
  flash <partition> [ <filename> ]         write a file to a flash partition
  erase <partition>                        erase a flash partition
  format[:[<fs type>][:[<size>]] <partition> format a flash partition.
                                           Can override the fs type and/or
                                           size the bootloader reports.
  getvar <variable>                        display a bootloader variable
  boot <kernel> [ <ramdisk> [ <second> ] ] download and boot kernel
  flash:raw boot <kernel> [ <ramdisk> [ <second> ] ] create bootimage and
                                           flash it
  devices                                  list all connected devices
  continue                                 continue with autoboot
  reboot                                   reboot device normally
  reboot-bootloader                        reboot device into bootloader
  help                                     show this help message

options:
  -w                                       erase userdata and cache (and format
                                           if supported by partition type)
  -u                                       do not first erase partition before
                                           formatting
  -s <specific device>                     specify device serial number
                                           or path to device port
  -l                                       with "devices", lists device paths
  -p <product>                             specify product name
  -c <cmdline>                             override kernel commandline
  -i <vendor id>                           specify a custom USB vendor id
  -b <base_addr>                           specify a custom kernel base address.
                                           default: 0x10000000
  -n <page size>                           specify the nand page size.
                                           default: 2048
  -S <size>[K|M|G]                         automatically sparse files greater
                                           than size.  0 to disable

Aaswad@Aaswads-MacBook-Pro /Library/Android/platform-tools
  %                                                                       !77
```

Unlocking bootloader

For us to get the root access we need to unlock the bootloader. The bootloader is usually locked by default to idiotproof the device, so that a layman would not end up rooting one's device and mess up with the kernel. Now from the same location where adb is located we will run fastboot. When we run fastboot we would need to set it to unlock the bootloader. For this we need to execute the command "./fastboot oem unlock". Now it would wait for the device to get the command. It would now look like the image below.



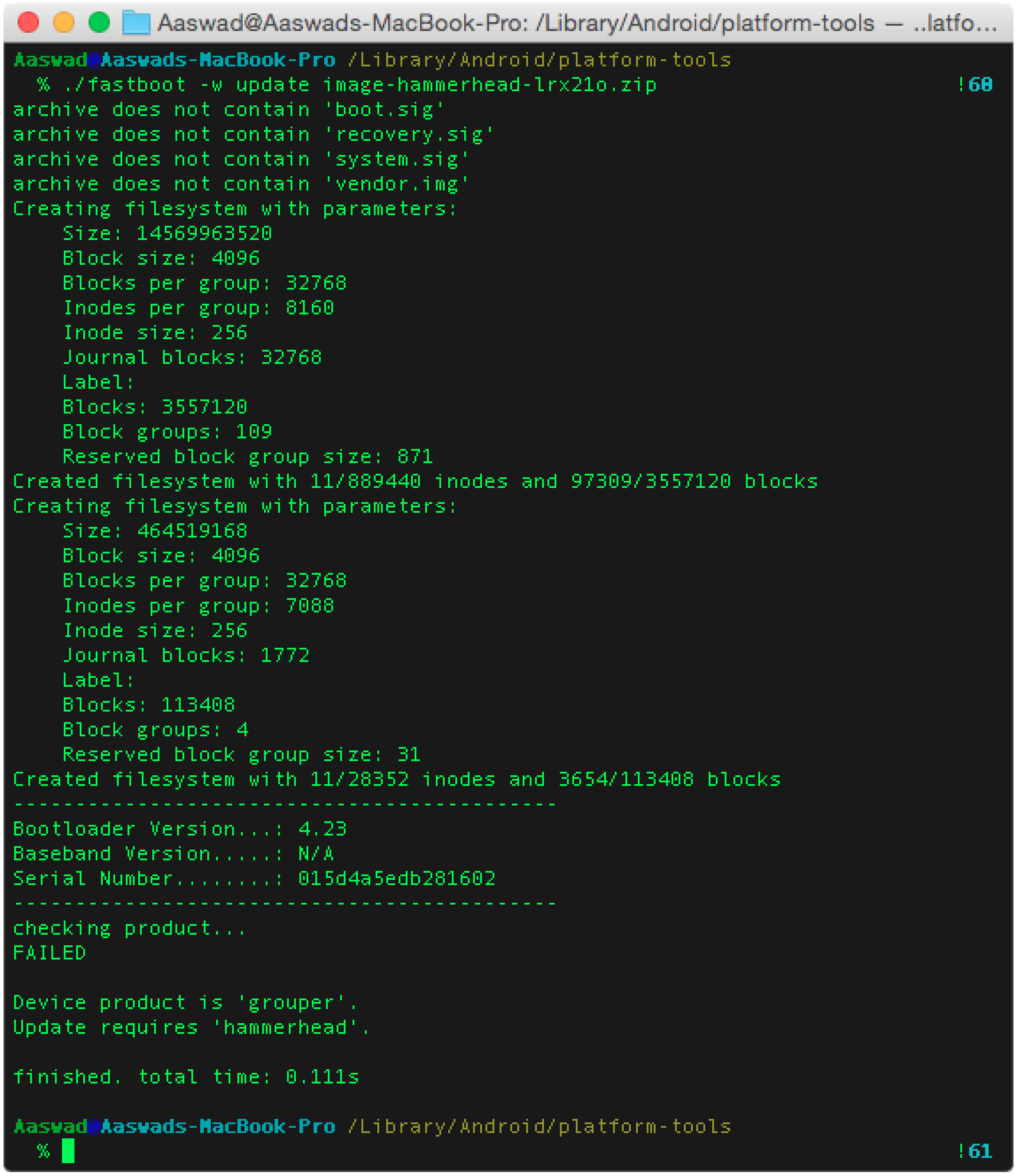
```
./fastboot oem unlock — ./fastboot — fastboot — 80×23
Last login: Wed Nov 26 11:59:13 on ttys000

Aaswad@Aaswads-MacBook-Pro ~
  % ..                                                                  !54

Aaswad@Aaswads-MacBook-Pro /Users
  % ..                                                                  !55

Aaswad@Aaswads-MacBook-Pro /
  % cd Library                                                          !55

Aaswad@Aaswads-MacBook-Pro /Library
  % cd Android                                                          !56

Aaswad@Aaswads-MacBook-Pro /Library/Android
  % cd platform-tools                                                   !57

Aaswad@Aaswads-MacBook-Pro /Library/Android/platform-tools
  % ./fastboot oem unlock                                               !58
< waiting for device >
```

Within few seconds it will ask a confirmation once, then click on "yes", and the bootloader is unlocked.

From now on one can see an unlocked icon whenever switching on the device. This means the bootlocker is successfully unlocked. Now we can go ahead with our process.

Experimenting with the OS asis

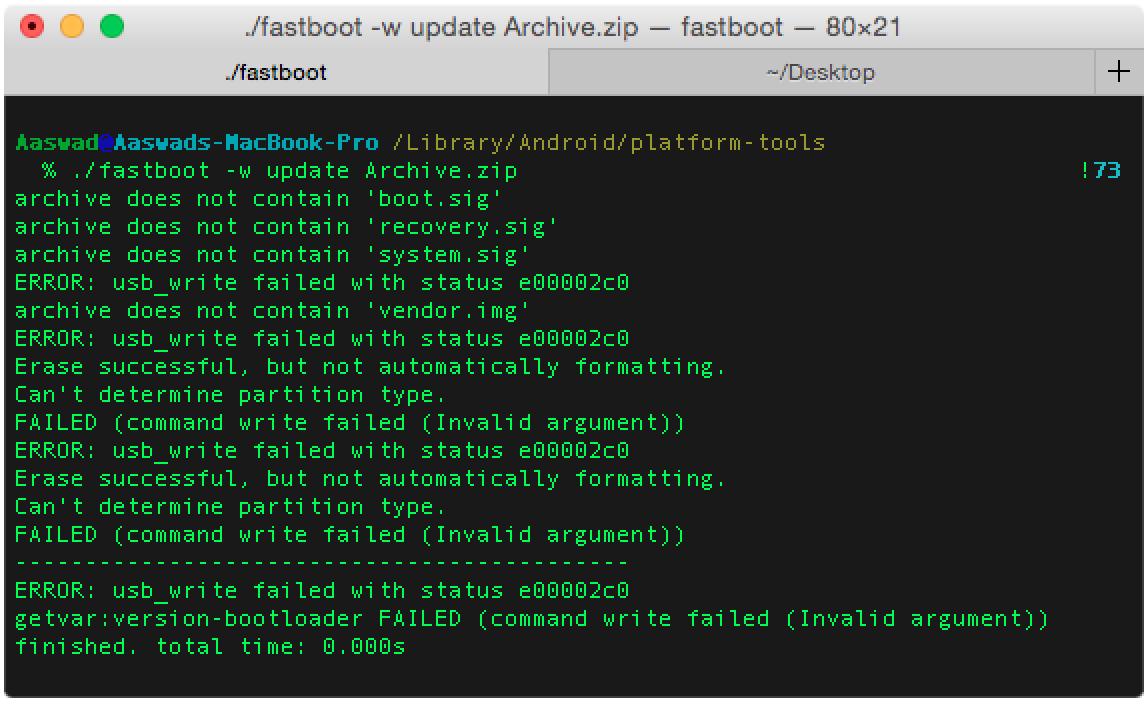
Now I tried going the easy way, only to find failure. I extracted the Android Lollipop OS developed for Nexus 5. I then tried to install this precompiled OS on the Nexus 7 Tablet. For this I went to fastboot. Then I installed the OS on the new device with the update command of fastboot program. For this I used the command, "./fastboot -w update image-hammerhead-lrx21o.zip". This command means run the fastboot tool,and when it gets the device run the update command on the device using the fastboot tool and update the OS ROM with the given image file, which in this case is "image-hammerhead-lrx21o.zip". Because it is not the same device, the device drivers must not work, and thats what happened as seen in the image below.


Aaswad@Aaswads-MacBook-Pro: /Library/Android/platform-tools — ..latfo...

```
Aaswad@Aaswads-MacBook-Pro /Library/Android/platform-tools
  % ./fastboot -w update image-hammerhead-lrx21o.zip                      !60
archive does not contain 'boot.sig'
archive does not contain 'recovery.sig'
archive does not contain 'system.sig'
archive does not contain 'vendor.img'
Creating filesystem with parameters:
    Size: 14569963520
    Block size: 4096
    Blocks per group: 32768
    Inodes per group: 8160
    Inode size: 256
    Journal blocks: 32768
    Label:
    Blocks: 3557120
    Block groups: 109
    Reserved block group size: 871
Created filesystem with 11/889440 inodes and 97309/3557120 blocks
Creating filesystem with parameters:
    Size: 464519168
    Block size: 4096
    Blocks per group: 32768
    Inodes per group: 7088
    Inode size: 256
    Journal blocks: 1772
    Label:
    Blocks: 113408
    Block groups: 4
    Reserved block group size: 31
Created filesystem with 11/28352 inodes and 3654/113408 blocks
--------------------------------------------
Bootloader Version...: 4.23
Baseband Version.....: N/A
Serial Number........: 015d4a5edb281602
--------------------------------------------
checking product...
FAILED

Device product is 'grouper'.
Update requires 'hammerhead'.

finished. total time: 0.111s

Aaswad@Aaswads-MacBook-Pro /Library/Android/platform-tools
  %                                                                       !61
```

As observed from the image above, device product details conflict with the device name. So I unzipped the image. and found a file named “android-info.txt”. I edited the text file and edited the required product name to be “grouper” rather than “hammerhead”. I then again zipped the file and tried to flash the new version. Then it gave similar errors to all the fields that described the device for which the OS is made. There are even fields to describe which all bandwidth would the device support. This is not only a software mention, but it should also have the hardware to support it. for example if one writes here that the device supports LTE but the device is only 3G, it won’t install.

After editing all those fields and trying for a day, It finally failed (as expected), with multiple errors. The errors. This can be seen in the image below.



```
Aaswad@Aaswads-MacBook-Pro /Library/Android/platform-tools
  % ./fastboot -w update Archive.zip                                          !73
archive does not contain 'boot.sig'
archive does not contain 'recovery.sig'
archive does not contain 'system.sig'
ERROR: usb_write failed with status e00002c0
archive does not contain 'vendor.img'
ERROR: usb_write failed with status e00002c0
Erase successful, but not automatically formatting.
Can't determine partition type.
FAILED (command write failed (Invalid argument))
ERROR: usb_write failed with status e00002c0
Erase successful, but not automatically formatting.
Can't determine partition type.
FAILED (command write failed (Invalid argument))
--------------------------------------------
ERROR: usb_write failed with status e00002c0
getvar:version-bootloader FAILED (command write failed (Invalid argument))
finished. total time: 0.000s
```

This must have happened because the ROM image doesn't have the vendor image because it lacks a SIM slot. So there is no question of the vendor in the device but since the ROM has been written for Nexus 5, also known as hammerhead, which has a SIM slot, the ROM fails to install.

Now one might ask why not pack all the binaries for all the devices into one image, the way Windows or Mac or Ubuntu does? To this I would say, my speculation is since it is a mobile device there are much more chips that can perform same jobs, e.g. Qualcomm Snapdragon series for CPUs, Adreno GPU series, and top of that it can have various architecture, like 64 bit 32 bit or it may be intel based or ARM based. So if Google had to pack in all the drivers and install dynamically the update would be much larger. For instance updates now are about 200 megabytes, and if it would contain all the binaries for all devices it would reach up to 3 to 4 gigabytes per device at least, if not more. This would have two challenges, one being there are very few devices with such a huge RAM, to temporarily store things till being installed. And two being it would use up lots of data transfer. And Google has a huge market in developing countries where data transfer is much more costlier. So this is not in the best favor of Google and Android. Even now, Google has been criticized for fragmentation, which is defined as many Android devices are not updated to the latest flavor of Android. So if Google would have done this it would suffer from fragmentation even more.

Getting the official code

Most of the android code is written in a pattern. Or rather organized in a pattern. Leaving away very few exceptions one will find the code in the following directories, /device/[vendor]/[codename], /vendor/[vendor]/[codename], and /kernel/[vendor]/[codename].

We can still go through one more simpler way. So now I downloaded the source code of Android 5.0 Lollipop. Google claims it is 8.5 gigabytes worth of code! It took me a week to understand what code is for which part. After finally figuring out the small parts of the code. I tried to recompile it for Nexus 7. It took a really long time to compile. Because Google suggested 16 gigabyte of RAM and I have only 8 gigabyte. My present laptop's configuration is 8 gigabyte RAM, 512 Gigabyte HDD, Intel i7 - 2.9 gigahertz, Quad core, and 3 megabyte level 3 cache. Then when it compiled, I decided to try putting it on my device. It took long time to get put it on device. I have installed custom ROMs on my devices and it takes at most 15 mins to get it running. But this time it was talking more than 35 minutes to get it to run! So I decided to retry. And next time it took about an hour to boot up. I was happy to see that it was working the easy way. But to my wonder, it didn't even last for 20 minutes.

I then recompiled the code, thinking that there must have been something wrong with the first compile and thus it didn't run. But again same thing happened.

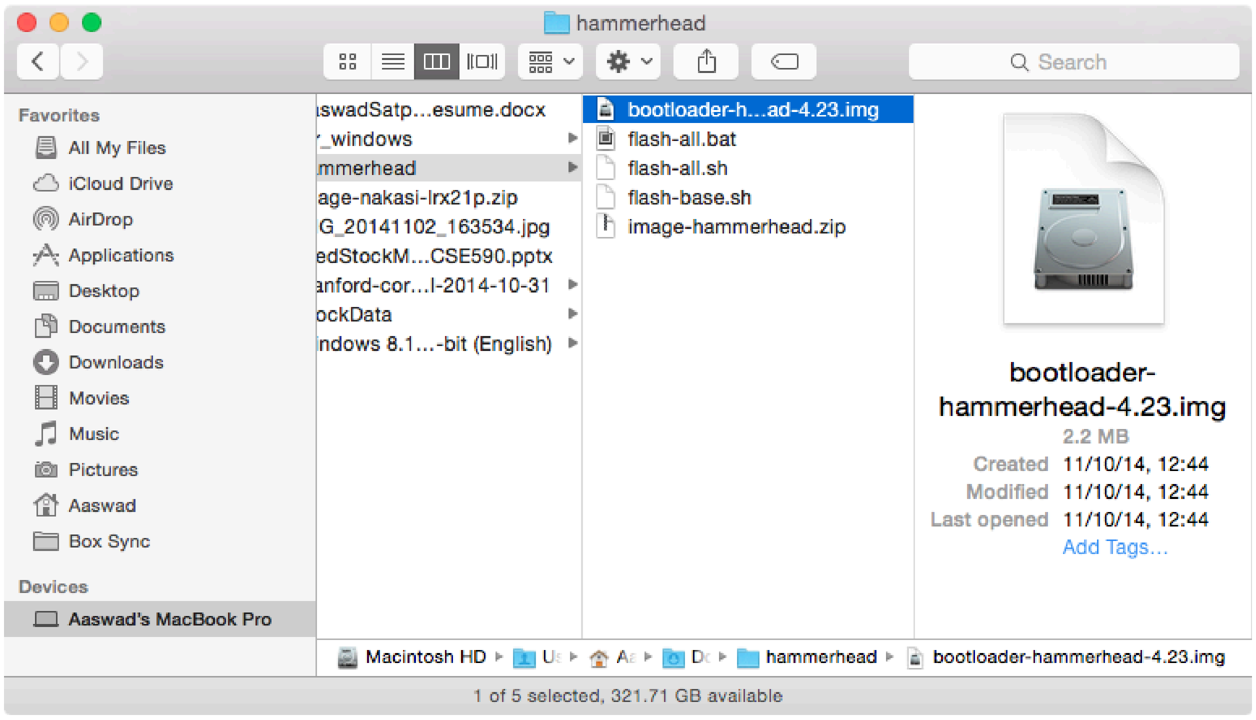
Then I decided to give it a final try, but this time on the machine with the configurations suggested by Google. For this I used my desktop which is back in India (a server grade machine). The configuration being 32 Gigabyte RAM, 512 gigabyte SSD, and 1 terabyte HDD, Intel Xeon - 3.0 gigahertz, overclocked to 3.6 gigahertz, Octa core, and 20 megabyte of level 3 cache. So this could easily handle the compilation. I had to transfer all the files and use-up my desktop using Teamviewer, which is free for non-commercial use. Even after doing this the same thing was happening.

I at this moment talked to few guys who worked at Samsung Mobiles. So I finally decided to go ahead with the other ways I could think of and that were suggested by Samsung guys. And hoped for success using that way, because the project deadline was fast approaching.

Getting hands dirty with the images

I downloaded the Android 5.0 Lollipop image for Nexus 5, the cell phone. It was a tar.gz file. It is an archive which has idiot-proofed itself. So the compiled image for the bootloader is given separately. We wouldn't be using the image because it is given as a safety measure so that if at all we screw up the bootloader we can still continue with this one. If at all one manages to screw up the bootloader, "./fastboot flash bootloader bootloader-hammerhead-4.23.img" is the command to reinstall the factory bootloader. The state at which one has screwed up the bootloader and/or the recovery, is called bricking of the device. At this state one can not get into his or her device and has to either put in all the images as made by the manufacturer.

The items in the archive are illustrated better with the following screenshot of the unarchived form of the downloaded archive from Google.



iswadSatp…esume.docx
_windows
mmerhead
age-nakasi-lrx21p.zip
G_20141102_163534.jpg
edStockM…CSE590.pptx
anford-cor…l-2014-10-31
ockData
ndows 8.1…-bit (English)

bootloader-h…ad-4.23.img
flash-all.bat
flash-all.sh
flash-base.sh
image-hammerhead.zip

bootloader-hammerhead-4.23.img
2.2 MB
Created 11/10/14, 12:44
Modified 11/10/14, 12:44
Last opened 11/10/14, 12:44
Add Tags…

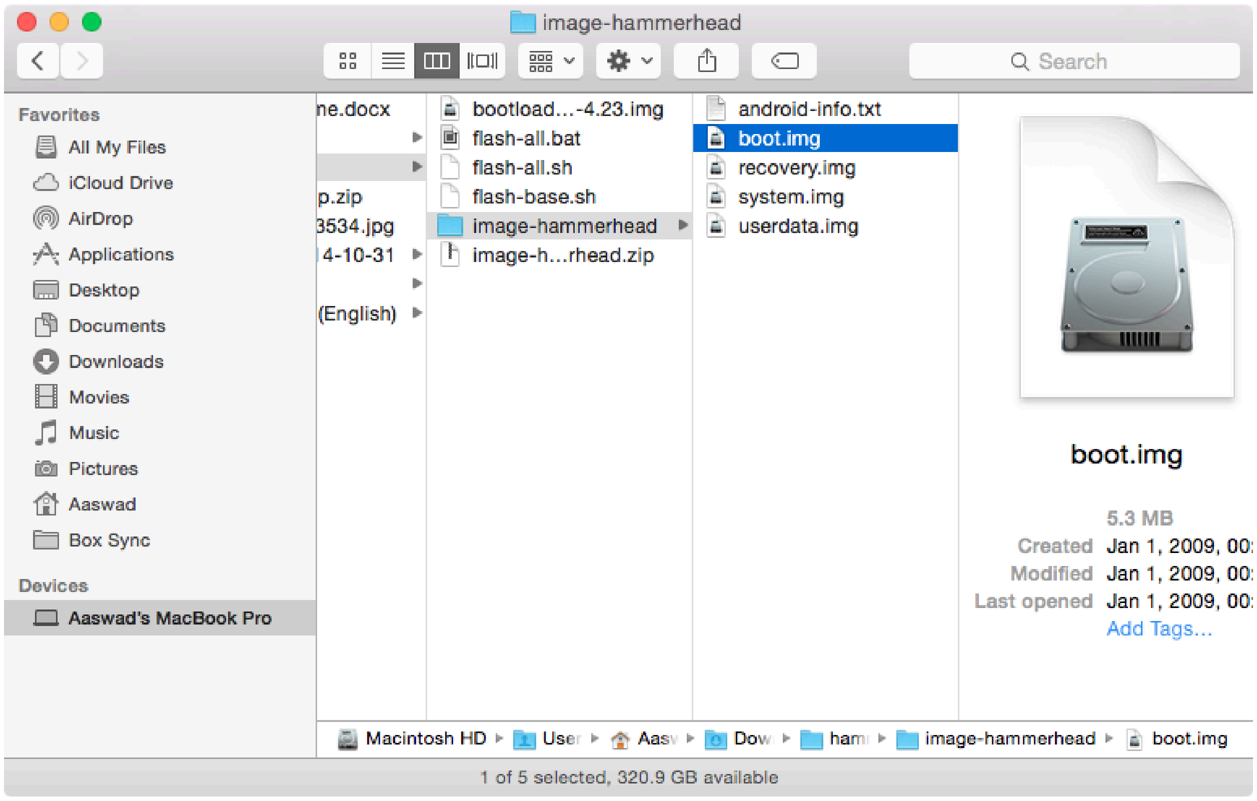
Macintosh HD ▸ Us ▸ Aa ▸ Do ▸ hammerhead ▸ bootloader-hammerhead-4.23.img

1 of 5 selected, 321.71 GB available

Google also gives us the direct shell script to install the image. But the shell script basically only runs the two commands we ran earlier.

The archive contains the final and most important part i.e. the zipped compiled images of Android 5.0 Lollipop. Now let us unarchive the zip containing the images to explore what's in it. After Unarchiving it we find images for boot, recovery, system, and user data. Boot image contains the booting details. Recovery image contains the recovery files. This is particularly important for OTA (Over The Air) updates. Then comes the system's image it has the complete Operating System in it. We will come back to this in some time. And the last we have user data images, this has all the temporary variable values, and the initial setup files. And apart from that it is completely empty with zero files. so it saves the space for user-data, which means the any user application's data. Which can be application's temporary data for example the state of game, or the high scores, or last used parameters.

The image below shows what all we have in the archived file we discussed earlier.



Brief listing of the different Images

The Android phone mainly has few partitions which is meant to hold the different file systems. They looks as follows:

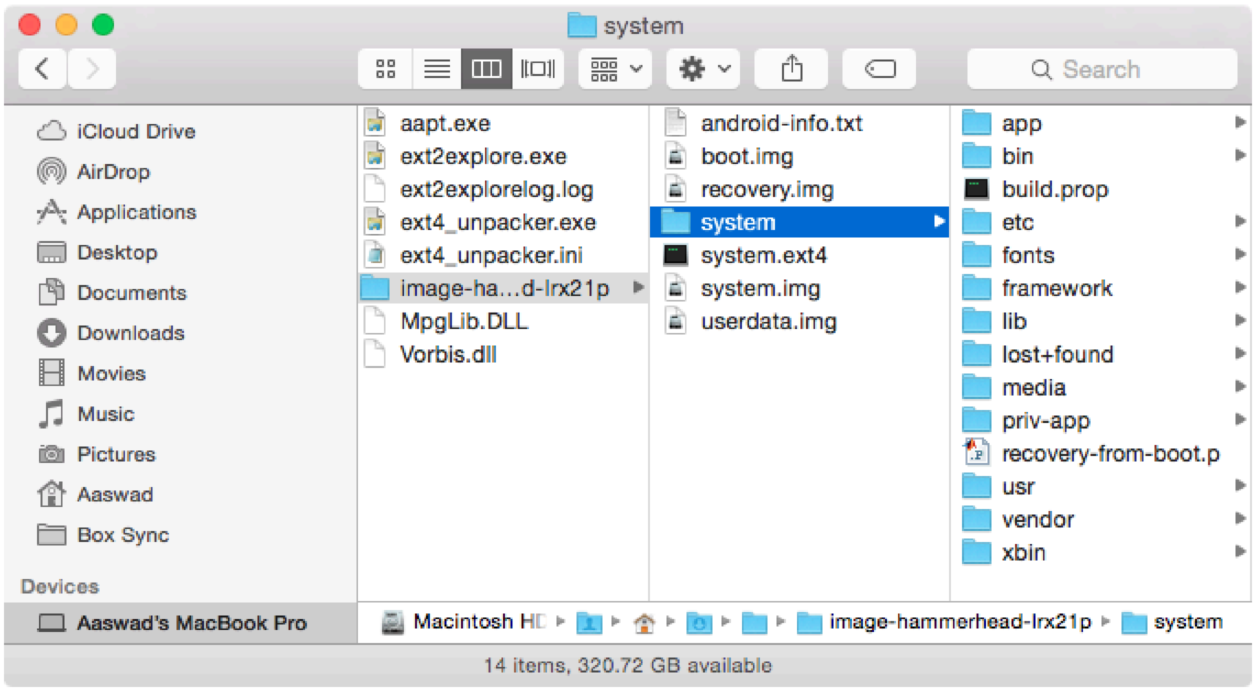
```
dev:    size      erasesize   name
mtd0:   00040000  00020000    "misc"
mtd1:   00500000  00020000    "recovery"
mtd2:   00280000  00020000    "boot"
mtd3:   04380000  00020000    "system"
mtd4:   04380000  00020000    "cache"
mtd5:   04ac0000  00020000    "userdata"
```

This usually varies from device to device. But more or less follows very similar architecture.

Opening the System image

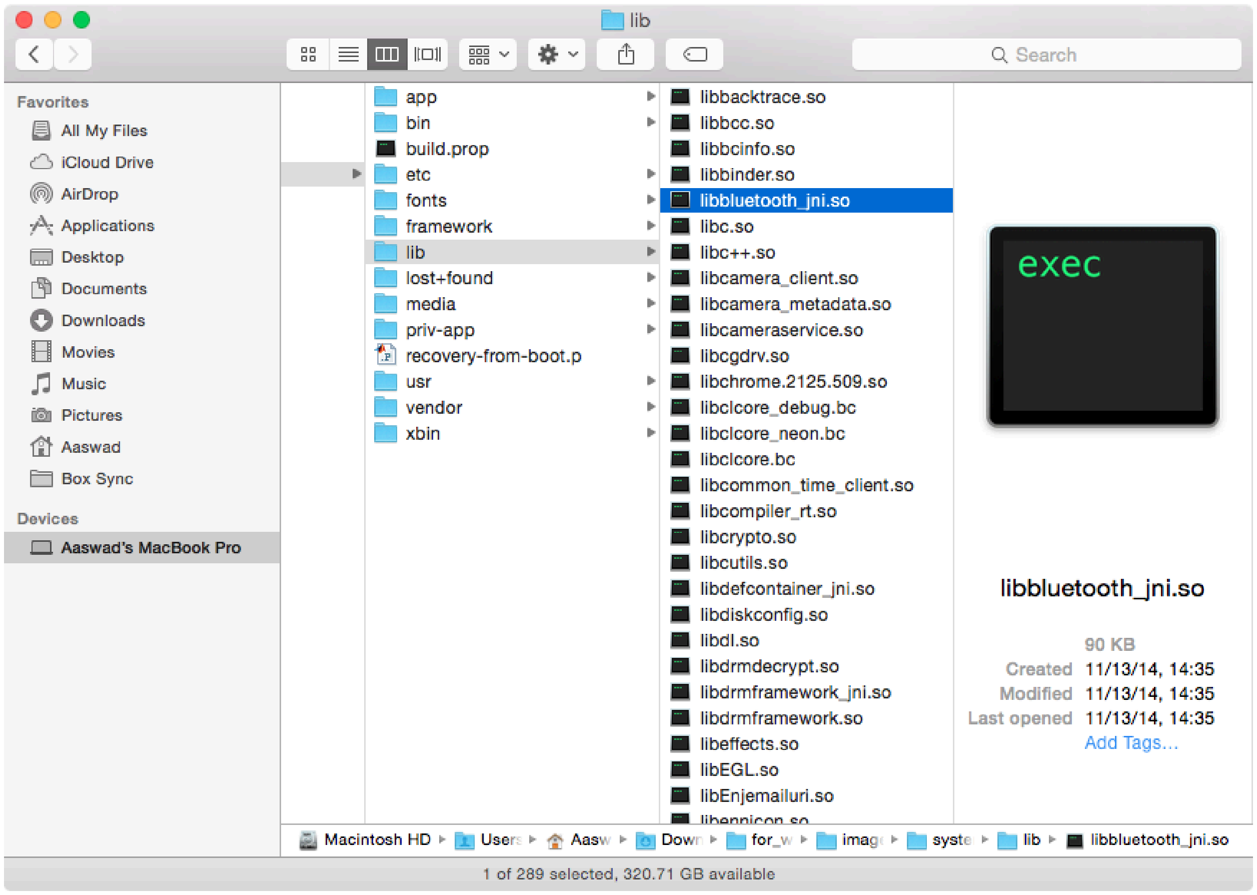
Now we open the signed image archive file, “system.img”. This contains the complete Android Operating System. This image is a signed archive of the system partition. To unarchive this I used ext4_unpacker, and ext2explore on windows. These are simple apps on windows to load the extended memory part to the program. We need to extract it from there after this. You will find the following files and folders once extracted.

During this phase, I had primarily thought that I would rewrite all the drivers from scratch. For this, I referred the video archive of the developer conference conducted by Marakana TV. This gave me many views about how a code has to be written so that it would directly run on some hardware. This was very interesting and so I started reading more about it. I then visited the official site of Android where this is properly described. The website, https://source.android.com/devices/index.html has detailed discription about the hardware on which the Android is about to run.



Here one can find all the OS files right from drivers and library files to apps and media files. This really made me very excited. I put in few hours browsing over the contents of this image file.

This fetched me many interesting facts about android like where can I edit my device name, or the resolution, or the boot loading animation. In this system image, I had thought editing the drivers would fetch me the required output. Let's see what happens! Most of the drivers are located at <system>/lib/. The following screenshot demonstrates the location of the drivers. It also illustrates the variety of driver binaries put into the OS's system image.



```
app                      libbacktrace.so
bin                      libbcc.so
build.prop               libbcinfo.so
etc                      libbinder.so
fonts                    libbluetooth_jni.so
framework                libc.so
lib                      libc++.so
lost+found               libcamera_client.so
media                    libcamera_metadata.so
priv-app                 libcameraservice.so
recovery-from-boot.p     libcgdrv.so
usr                      libchrome.2125.509.so
vendor                   libclcore_debug.bc
xbin                     libclcore_neon.bc
                         libclcore.bc
                         libcommon_time_client.so
                         libcompiler_rt.so
                         libcrypto.so
                         libcutils.so
                         libdefcontainer_jni.so
                         libdiskconfig.so
                         libdl.so
                         libdrmdecrypt.so
                         libdrmframework_jni.so
                         libdrmframework.so
                         libeffects.so
                         libEGL.so
                         libEnjemailuri.so
                         libennicon.so
```

```
libbluetooth_jni.so

              90 KB
Created       11/13/14, 14:35
Modified      11/13/14, 14:35
Last opened   11/13/14, 14:35
```

```
Macintosh HD ▸ Users ▸ Aasw ▸ Down ▸ for_w ▸ imag ▸ syste ▸ lib ▸ libbluetooth_jni.so
1 of 289 selected, 320.71 GB available
```

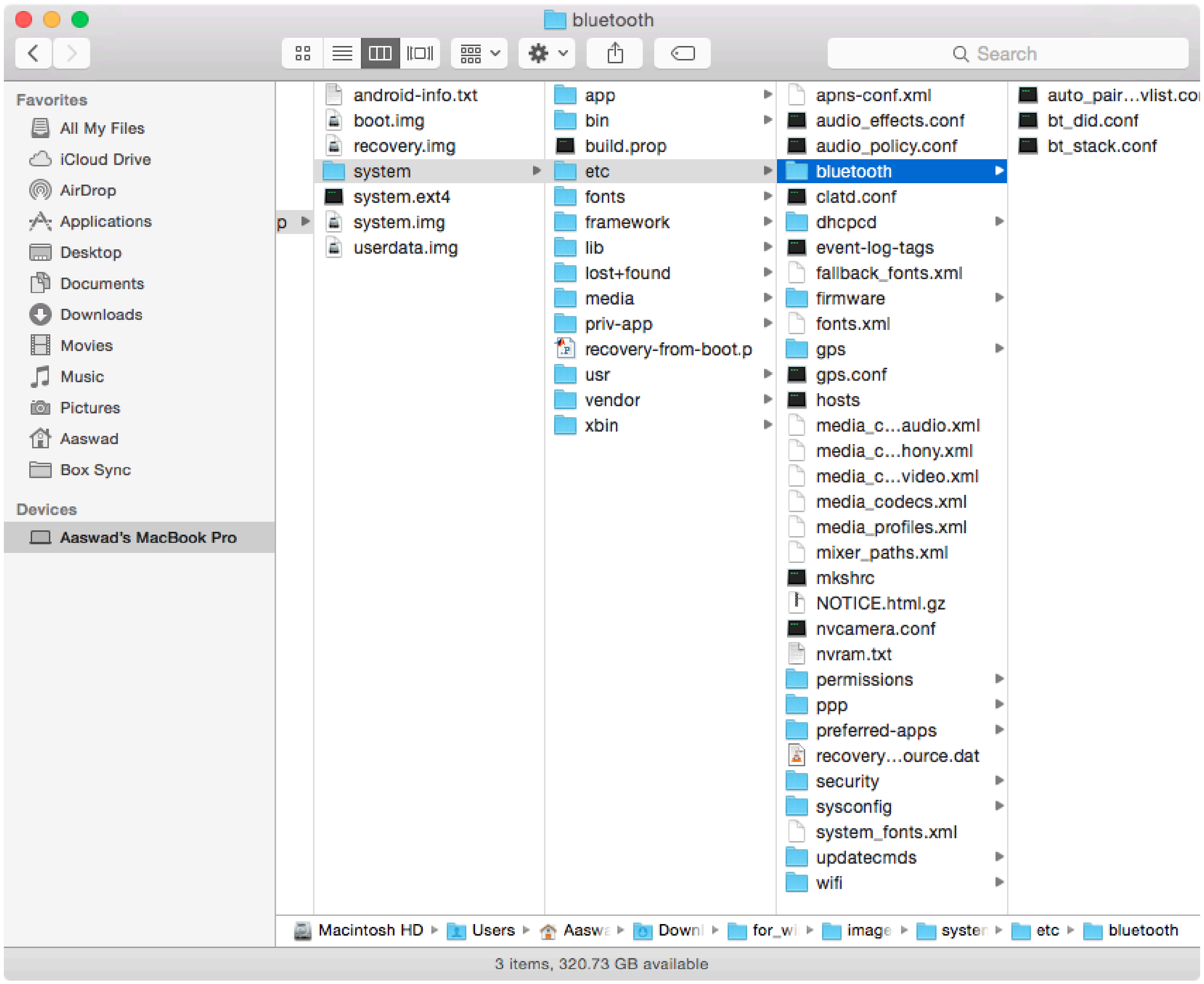
Now, I replaced all these drivers with the drivers from the similar Android 4.4.4 KitKat image made for Nexus 7.

Once this was done I recreated the system image by archiving the folder and followed the exact reverse of the unarchiving process. Then the system.img was kept at the same location from where we unarchived. Then this folder was then again archived as the final image.zip. Now again I tried to install this zip into the Nexus 7. This time it failed again.

Then I had to figure out what was going wrong. Finally after about 18 hours, I understood what is that I was doing wrong. I edited the drivers, true, but the OS needs to know how to interact with these drivers. So then I also had to update the driver configuration files. This is one of the most important step discussed as follows.

Driver configuration files

The location <system>/etc/ is where one would find almost all the driver configuration files in config file format. Few of the complex drivers like bluetooth, and wifi are packed in the folder. This folder contains few config files related to the category. For example bluetooth has stack config file as well as automatic pairing config files. This you can see from the image below.

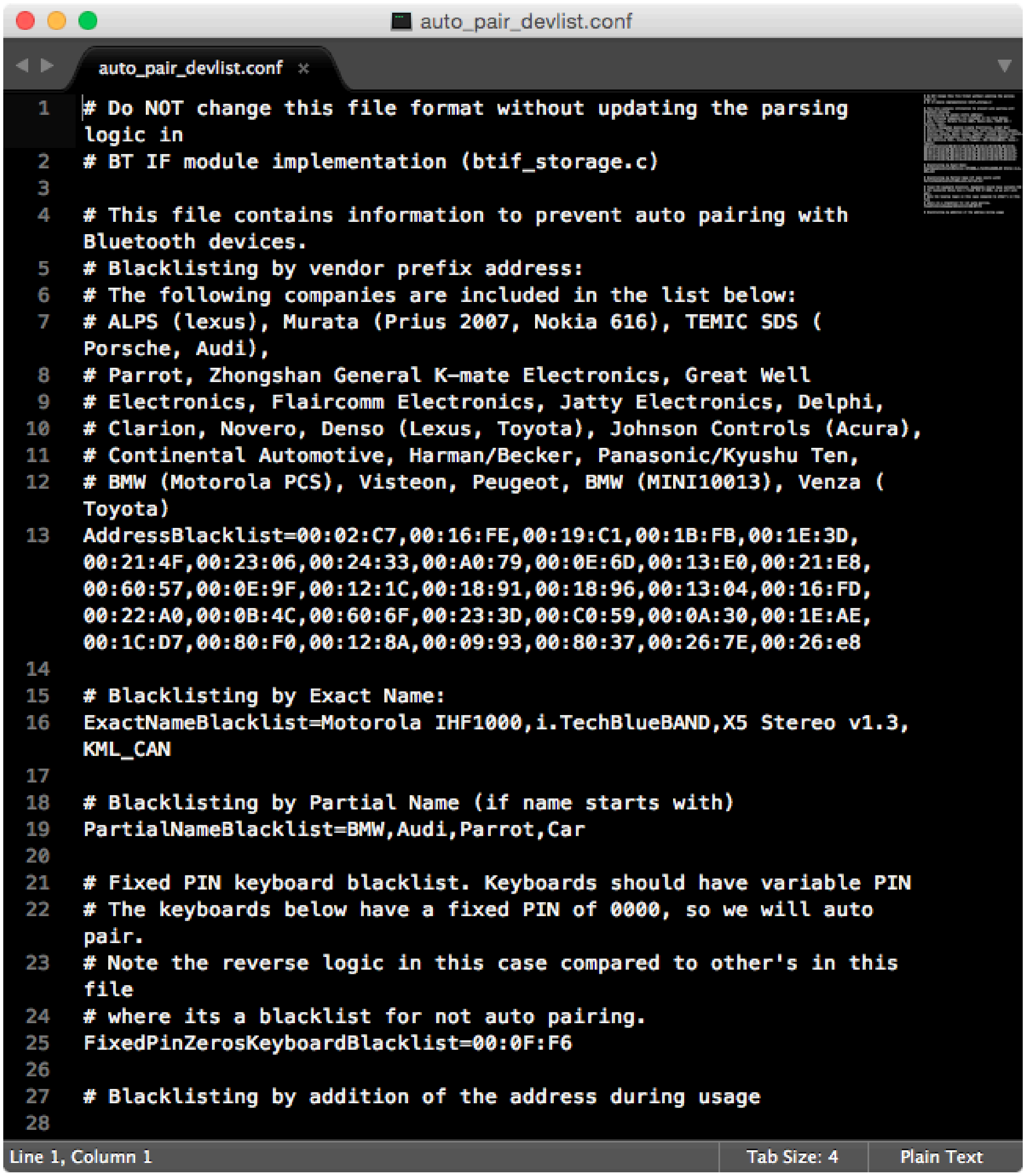


```
android-info.txt   app                   apns-conf.xml          auto_pair…vlist.co
boot.img           bin                   audio_effects.conf     bt_did.conf
recovery.img       build.prop            audio_policy.conf      bt_stack.conf
system             etc                   bluetooth
system.ext4        fonts                 clatd.conf
system.img         framework             dhcpcd
userdata.img       lib                   event-log-tags
                   lost+found            fallback_fonts.xml
                   media                 firmware
                   priv-app              fonts.xml
                   recovery-from-boot.p  gps
                   usr                   gps.conf
                   vendor                hosts
                   xbin                  media_c…audio.xml
                                         media_c…hony.xml
                                         media_c…video.xml
                                         media_codecs.xml
                                         media_profiles.xml
                                         mixer_paths.xml
                                         mkshrc
                                         NOTICE.html.gz
                                         nvcamera.conf
                                         nvram.txt
                                         permissions
                                         ppp
                                         preferred-apps
                                         recovery…ource.dat
                                         security
                                         sysconfig
                                         system_fonts.xml
                                         updatecmds
                                         wifi
```

Macintosh HD ▸ Users ▸ Aasw ▸ Downl ▸ for_wi ▸ image ▸ syster ▸ etc ▸ bluetooth

3 items, 320.73 GB available

Now I had to go through all the driver config files. And check one by one if we need to edit anything. I got all the properties of the new device from the official site as well as wikipedia. The config files were also pretty easy to read and understand. For example for bluetooth config which says which devices to connect and which not to connect, as seen in the screenshot below.



```
# Do NOT change this file format without updating the parsing logic in
# BT IF module implementation (btif_storage.c)

# This file contains information to prevent auto pairing with Bluetooth devices.
# Blacklisting by vendor prefix address:
# The following companies are included in the list below:
# ALPS (lexus), Murata (Prius 2007, Nokia 616), TEMIC SDS (Porsche, Audi),
# Parrot, Zhongshan General K-mate Electronics, Great Well
# Electronics, Flaircomm Electronics, Jatty Electronics, Delphi,
# Clarion, Novero, Denso (Lexus, Toyota), Johnson Controls (Acura),
# Continental Automotive, Harman/Becker, Panasonic/Kyushu Ten,
# BMW (Motorola PCS), Visteon, Peugeot, BMW (MINI10013), Venza (Toyota)
AddressBlacklist=00:02:C7,00:16:FE,00:19:C1,00:1B:FB,00:1E:3D,00:21:4F,00:23:06,00:24:33,00:A0:79,00:0E:6D,00:13:E0,00:21:E8,00:60:57,00:0E:9F,00:12:1C,00:18:91,00:18:96,00:13:04,00:16:FD,00:22:A0,00:0B:4C,00:60:6F,00:23:3D,00:C0:59,00:0A:30,00:1E:AE,00:1C:D7,00:80:F0,00:12:8A,00:09:93,00:80:37,00:26:7E,00:26:e8

# Blacklisting by Exact Name:
ExactNameBlacklist=Motorola IHF1000,i.TechBlueBAND,X5 Stereo v1.3,KML_CAN

# Blacklisting by Partial Name (if name starts with)
PartialNameBlacklist=BMW,Audi,Parrot,Car

# Fixed PIN keyboard blacklist. Keyboards should have variable PIN
# The keyboards below have a fixed PIN of 0000, so we will auto pair.
# Note the reverse logic in this case compared to other's in this file
# where its a blacklist for not auto pairing.
FixedPinZerosKeyboardBlacklist=00:0F:F6

# Blacklisting by addition of the address during usage

```

Now I changed the config files. And recreated the system image as discussed earlier. This time it finally worked.

But there were few bugs still in the OS. It did not scale properly, and had many weird artifacts. One of those was the touch input used to take any input data without tablet even being touched! This artifact once actually landed me into trouble. During a CSE 590 lecture, the tablet suddenly switched on some song. I first thought that it must have happened by mistake but it continued to happen like this for 2 more times.

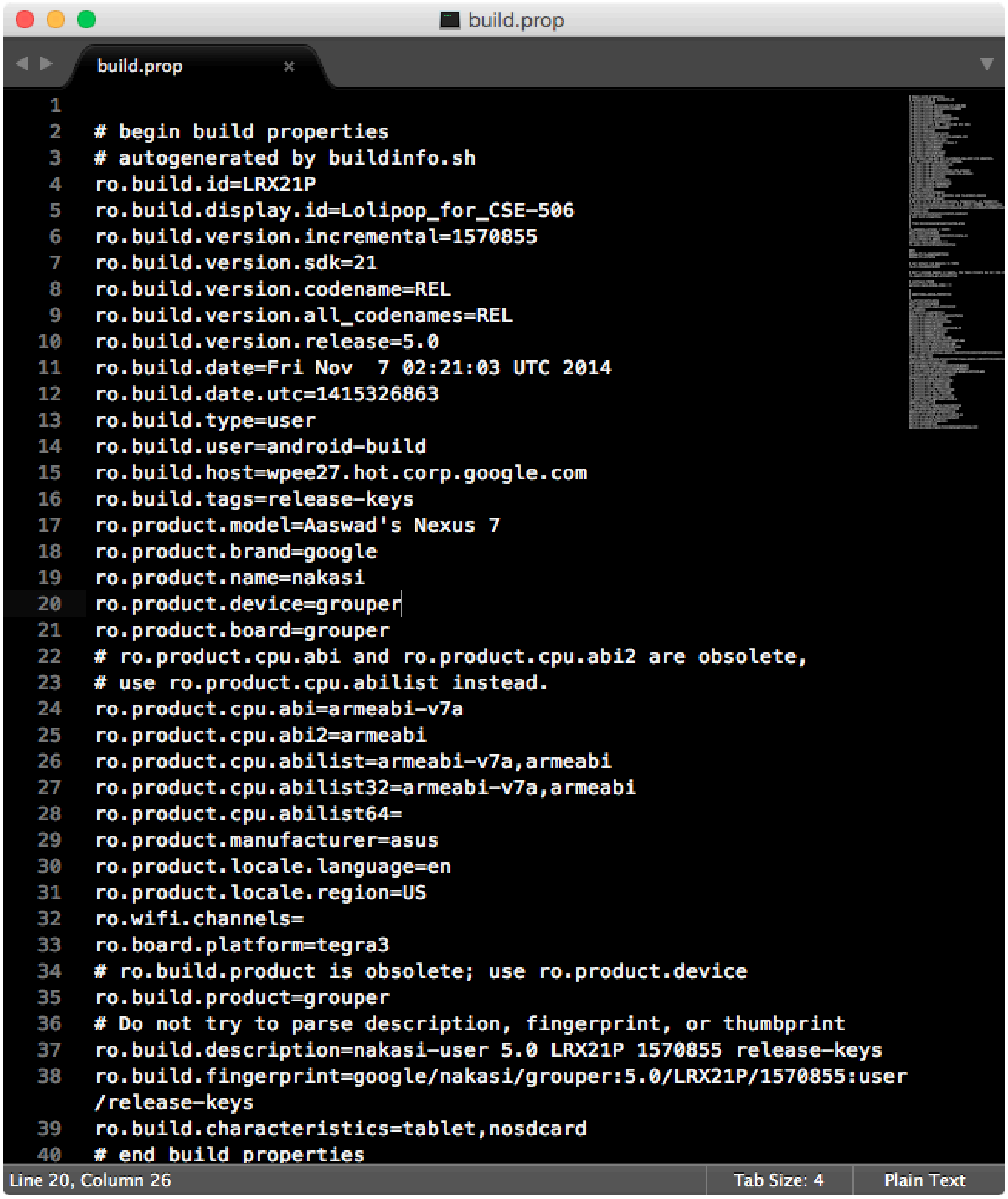
I then decided to read about it on XDA forum at http://forum.xda-developers.com/. This was because it actually computed for 1920x1080 screen, the one for Nexus 5, and so even a small change in charge nearby the device made it feel that there is some capacitive body touching it. And as it has capacitive screen, it was easily confused it as a touch. This is partially my speculation and partly suggested by the forum.

There was only one hope to get this running properly, edit the build prop file.

BuildProp File

The build prop file is what holds the majority of the runtime flags that are used when android boots, think of it as being similar to config.sys on dos or environment variables in various OS's. It only holds variables, it can’t run any scripts on it's own. Whatever is placed inside depends on android itself to pay attention to the flag and actually use it.

One can see the build prop file in the screenshot below.

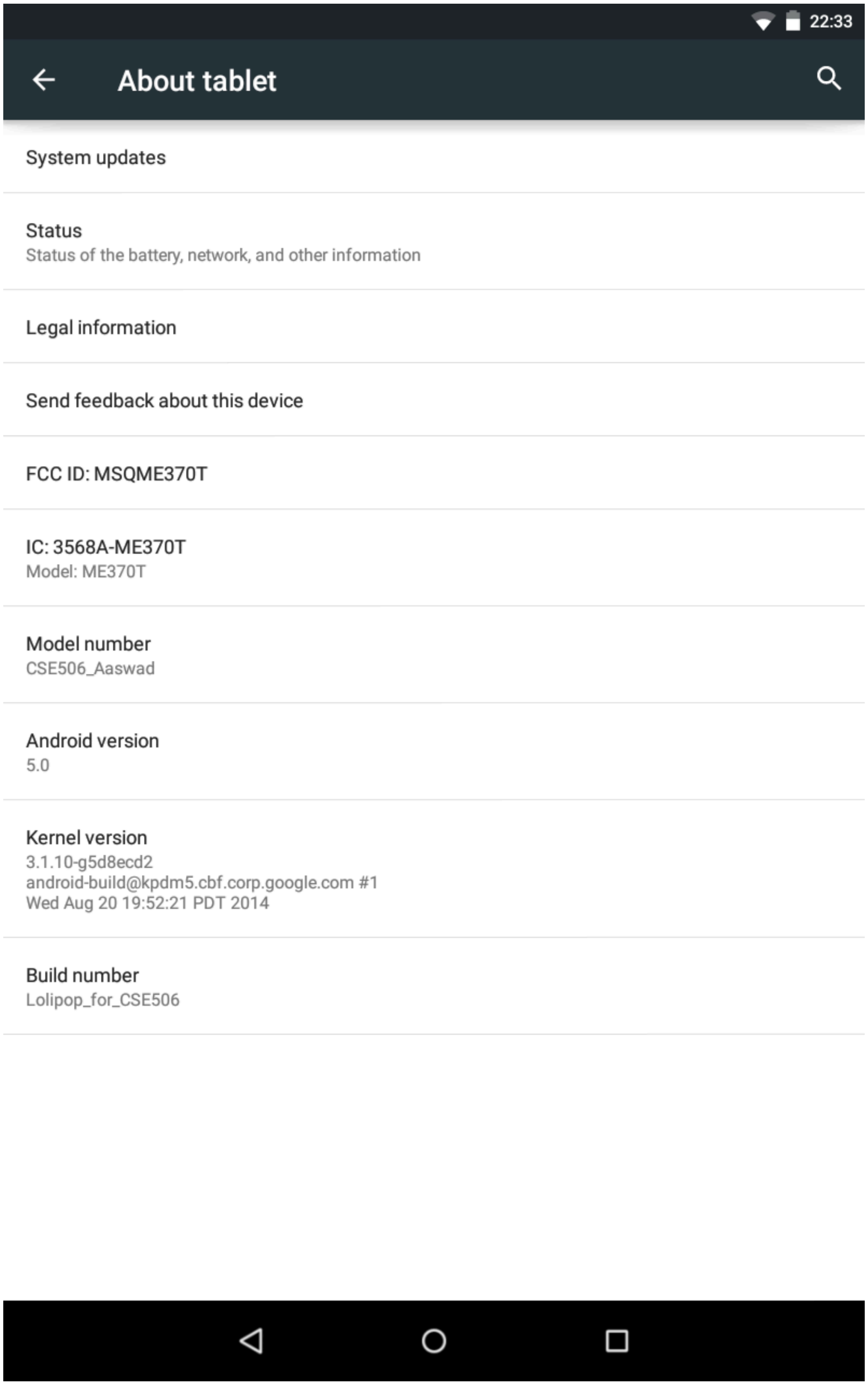


```

# begin build properties
# autogenerated by buildinfo.sh
ro.build.id=LRX21P
ro.build.display.id=Lolipop_for_CSE-506
ro.build.version.incremental=1570855
ro.build.version.sdk=21
ro.build.version.codename=REL
ro.build.version.all_codenames=REL
ro.build.version.release=5.0
ro.build.date=Fri Nov  7 02:21:03 UTC 2014
ro.build.date.utc=1415326863
ro.build.type=user
ro.build.user=android-build
ro.build.host=wpee27.hot.corp.google.com
ro.build.tags=release-keys
ro.product.model=Aaswad's Nexus 7
ro.product.brand=google
ro.product.name=nakasi
ro.product.device=grouper
ro.product.board=grouper
# ro.product.cpu.abi and ro.product.cpu.abi2 are obsolete,
# use ro.product.cpu.abilist instead.
ro.product.cpu.abi=armeabi-v7a
ro.product.cpu.abi2=armeabi
ro.product.cpu.abilist=armeabi-v7a,armeabi
ro.product.cpu.abilist32=armeabi-v7a,armeabi
ro.product.cpu.abilist64=
ro.product.manufacturer=asus
ro.product.locale.language=en
ro.product.locale.region=US
ro.wifi.channels=
ro.board.platform=tegra3
# ro.build.product is obsolete; use ro.product.device
ro.build.product=grouper
# Do not try to parse description, fingerprint, or thumbprint
ro.build.description=nakasi-user 5.0 LRX21P 1570855 release-keys
ro.build.fingerprint=google/nakasi/grouper:5.0/LRX21P/1570855:user
/release-keys
ro.build.characteristics=tablet,nosdcard
# end build properties
```

Final Outcome

Once all of this was done. I successfully ported Android to my tablet. Now I can easily enjoy the latest Android version even before Google actually releases OTA (Over The Air) update for my tablet.

Now my tablet shows the version to be Android 5.0 Lollipop. This can be seen from the screenshot below.



22:33

About tablet

System updates

Status
Status of the battery, network, and other information

Legal information

Send feedback about this device

FCC ID: MSQME370T

IC: 3568A-ME370T
Model: ME370T

Model number
CSE506_Aaswad

Android version
5.0

Kernel version
3.1.10-g5d8ecd2
android-build@kpdm5.cbf.corp.google.com #1
Wed Aug 20 19:52:21 PDT 2014

Build number
Lolipop_for_CSE506

Conclusion

This was certainly a project that interested me a lot. Even though it required a lot of efforts from my side, I enjoyed it a lot. I faced many problems as discussed earlier, and due to interest I completed the project with satisfying results. The unarchived system image is uploaded on http://aaswad.7pute.com/CSE506SysZip and is publically avaliable by anyone to whom this project intrests.

Bibliography

I have referred to following materials to go through this project:

1. Android Developers, http://developer.android.com/
2. Android Source, https://source.android.com/
3. Android Drivers, https://source.android.com/devices/index.html
4. Android Internals - Building a Custom ROM by Marakana TV, https://www.youtube.com/watch?v=1_H4AlQaNa0 and https://www.youtube.com/watch?v=rFqELLB1Kk8
5. Wikipedia, https://en.wikipedia.org/wiki
6. XDA Forum, http://forum.xda-developers.com/