X86 ROM Cooking 101

Ron Munitz
Founder & CEO - The PSCG
Founder & CTO - Nubo Software

ron@android-x86.org
https://github.com/ronubo/

The slides will be available online at:
thepscg.com/talks/2014
Agenda

● Introduction
  ○ You, Me, Android

● Introduction to Embedded Systems
  ○ Embedded Systems
  ○ Android Partition Layout

● Android X86 projects
  ○ Virtual Machine discussion
  ○ The Init sequence
  ○ Multi Booting

● The Android Build System
  ○ Building an AOSP ROM from scratch
About://Ron Munitz

- **Distributed Fault Tolerant Avionic Systems**
  - Linux, VxWorks, very esoteric libraries, 0’s and 1’s

- **Highly distributed video routers**
  - Linux

- **Real Time, Embedded, Server bringups**
  - Linux, Android, VxWorks, Windows, devices, BSPs, DSPs,...

- **Distributed Android**
  - Rdroid? Cloudroid? Too busy working to get over the legal naming, so no name is officially claimed for my open source.

- **What currently keeps me busy:**
  - Running the PSCG, a Embedded/Android consulting and Training
  - Managing R&D at *Nubo Software* and advising on Remote Display Protocols
  - Promoting open source with *The New Circle* expert network
  - Teaching, Researching and Project Advising at *Afeka’s college of Engineering*
  - Amazing present, endless opportunities. (Wish flying took less time)
Android History (2002-2007)

- **2002** - SideKick by Danger Inc. - The first “Internet Phone”.
  - Technical session at Stanford by Andy Rubin, CEO of Danger Inc.
  - Google’s Brin & Page attend, and soon become Sidekick users.
  - Sidekick fails to achieve commercial success

- **2003** - Andy Rubin forms “Android”, targeted at operating mobile phones and cameras

- **2005** - Google Acquires “Android”.

- **2007** - The Open Handset Alliance is formed
  - November 5th - The OHA Announces Android, an open source mobile phone platform based on the Linux kernel
  - November 12th - Google Announces the Android SDK, along with a $10 million Android developer challenge
Android History (2008-2009)

● **2008** - T-mobile and Google announce the first Android phone - the G1
  ○ AKA. The *HTC “Dream”*
  ○ Approved by the FCC on August 18th 2008
  ○ First available - October 22nd

● **2009** - Motorola Droid is announced, running Android 2.0
  ○ Considered by many as the opening note for the smartphone wars.
  ○ Added at least two exclusive features:
    ■ Hardware keyboard
    ■ Replaceable battery
2010 was an exciting year for Android:

- Google Announces its first flagship device - the Nexus One
  - Which is one of the best phones I have ever had.
- Samsung sets a giant’s foot on the battlefield
  - Galaxy S and its variants hit the market
- HTC’s EVO4G hits the market
  - Was widely considered as the best iPhone alternative at that time
- **Android’s market share first passes the iPhone’s market share**
- Google announces the *Gingerbread* (2.3) Android version, debuting on the Nexus S.
  - Introducing the most popular Android version until the end of 2013
  - Introducing NFC and SIP
Android History (2011 - 2013)

- **2011** - Honeycomb (3.0), google’s first aim at the tablet market is out
  - Android’s market share first passes the Blackberry’s market share
  - November 2011 - **Ice Scream Sandwich** is out, debutting on the *Galaxy Nexus*

- **2012** - JellyBean is released
  - Introducing significant enhancement in user experience (Project butter)
  - Introducing multi user for tablets
  - Samsung confidently ranks as the top Android phone maker on earth

- **2013** - More devices, more market share,
  - Android 4.3 is out: Enhanced WiFi-Display, Open GL ES 3.0,...
  - Android 4.4 (**KitKat**) is out: First time a commercial brand hits Android, better memory utilization, enhanced security, in-platform PDF rendering, enhanced printer support and more...
Android History (2014 - The Future)

- Foreseeable future:
  - More devices
  - More power
  - More features
  - More apps
  - More developers
  - More competition
  - More embedded Android engineers needed.

- Will Android be crowned as the new Embedded Linux?
# Platform Versions - Present day distributions

<table>
<thead>
<tr>
<th>Version</th>
<th>Codename</th>
<th>API</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>Froyo</td>
<td>8</td>
<td>1.3%</td>
</tr>
<tr>
<td>2.3.3 - 2.3.7</td>
<td>Gingerbread</td>
<td>10</td>
<td>20.0%</td>
</tr>
<tr>
<td>3.2</td>
<td>Honeycomb</td>
<td>13</td>
<td>0.1%</td>
</tr>
<tr>
<td>4.0.3 - 4.0.4</td>
<td>Ice Cream Sandwich</td>
<td>15</td>
<td>16.1%</td>
</tr>
<tr>
<td>4.1.x</td>
<td>Jelly Bean</td>
<td>16</td>
<td>35.5%</td>
</tr>
<tr>
<td>4.2.x</td>
<td></td>
<td>17</td>
<td>16.3%</td>
</tr>
<tr>
<td>4.3</td>
<td></td>
<td>18</td>
<td>8.9%</td>
</tr>
<tr>
<td>4.4</td>
<td>KitKat</td>
<td>19</td>
<td>1.8%</td>
</tr>
</tbody>
</table>

Data collected during a 7-day period ending on February 4, 2014. Any versions with less than 0.1% distribution are not shown.
Historical Exhibit: Android Platform Overview
Introduction to ROM Cooking
Agenda

● What is a "ROM"?
● Embedded Systems Primer
● Examples of Android ROMs
● ROMs in the Android developer world
● Building your first ROM out of the AOSP
● Android and X86
"ROM" - Definition

From Wiktionary, the free Dictionary:

“ROM”:

- (electronics, computing) read-only memory
- (video games) A software image of read-only memory (as of a game cartridge) used in emulation
- (medicine) Range of Motion
- (finance) Return on Margin
- (estimating and purchasing) Rough order of magnitude. An informal cost or price estimate provided for planning and budgeting purposes only, typically expected to be only 75% accurate
"ROM" - Definition (cont)

From Wikipedia, the free Encyclopedia:

ROM, Rom, or rom is an abbreviation and name that may refer to:

In computers and mathematics (that's us!):

- **Read-only memory**, a type of storage media that is used in computers and other electronic devices
- **ROM image**, a computer file which contains a copy of the data from a read-only memory chip
- **ROM (MUD)**, a popular MUD codebase
- Random oracle model, a mathematical abstraction used in cryptographic proofs
- **ROM cartridge**, a portable form of read-only memory
- **RoM**, Request of Maintainer (see Software maintainer)
- **Rough order of magnitude estimate**
As CyanogenMod educates us in their overview of Modding:

“
You can flash a **ROM** onto the **ROM**, which isn't really **ROM**

Embedded Build Systems

- In the introduction module we saw a recipe for building Android using the AOSP Build System.
- The build procedure was done on a designated machine, which has all the required tools to turn the
  - That machine is referred as The Host
- The host is used to build the system for a designated device, may it be a handset, an emulator, a streamer etc.
  - That device is referred to as The Target
In *Embedded Software Development*, the common case is that \( \text{host} \neq \text{target} \).

They *may* have the same attributes:
- architecture (i.e. x86, arm, mips...),
- library versions (libc, libstdc++, ...)
- *toolchains* (gcc, ar, ...)

But they *do not have to*, and will usually have *little to nothing in common*.

Hence, the build system uses a *cross Toolchain*, to *cross compile* build artifacts for the *target* on the *host*. 
Native and Cross Compiling

**Native Compiling**

$ cat hello.c
#include <stdio.h>
int main()
{
    printf("Ciao Mondo\n");
    return 0;
}

$ which gcc
/usr/bin/gcc

$ gcc --static -o hello_native hello.c
$ ./hello_native
Ciao Mondo

$ file hello_native
hello_native: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, for GNU/Linux 2.6.24, BuildID [sha1]
  =0x60117523776dbf4ff7d4378cce2f184d5

**Cross Compiling**

$ CC=~/tc/bin/arm-linux-androideabi-gcc
cross-compile

$ gcc --static -o hello_cross hello.c
$ ./hello_cross
bash: ./hello_cross: cannot execute binary file

$ file hello_cross
hello_cross: ELF 32-bit LSB executable, ARM, version 1 (SYSV), statically linked, not stripped
Canadian Cross

This simplified (and very inaccurate) image depicts a technique for building Cross Compilers, known as the **Canadian Cross**

Embedded Development Overview

**HOST**
- Linux kernel (3.10.7.3)
- GNU toolchain
- GNU make (3.81)
- Python (2.7.3)
- Shell (bash 4.2.25)
- Oracle JDK (1.6.0.34)
- Git (1.7.9.5)
- repo (1.12.2)
- Cross toolchains

**TARGET**
- BSP
- Kernel
- Drivers (?)
- Userspace (?)
- Shell (?)
- Graphics (?)
- ...

**Virtual Environments**
- Android-X86 over QEMU
- Windows over VirtualBox
- Linux over VmWare
- Android Emulator (X86)
- Android Emulator (ARM)
- LFS over UML
Connecting the host with the target - The Android way

HOST
- Linux kernel (3.10.7.3)
- GNU toolchain
- GNU make (3.81)
- Python (2.7.3)
- Shell (bash 4.2.25)
- Oracle JDK (1.6.0.34)
- Git (1.7.9.5)
- repo (1.12.2)
- Cross toolchains

TARGET
- fastboot
- adb
- BSP
- Kernel
- Drivers (?)
- Userspace (?)
- Shell (?)
- Graphics (?)
- ...
Embedded Build System - Overview

HOST

Operating system
Host toolchains
Cross toolchains
Source control

Build (I)
Images (Build Artifacts) for target (I)

Build (II)
Images (Build Artifacts) for target (II)

Android-X86 over QEMU
Android Emulator (X86)

Windows over VirtualBox
Android Emulator (ARM)

Linux over VmWare
LFS over UML

Windows
over
VirtualBox

Linux
over
VmWare

LFS
over
UML
Embedded Build System - The Android way

HOST
Linux kernel (3.10.7.3)
GNU toolchain
GNU make (3.81)
Python (2.7.3)-
Shell (bash 4.2.25)
Oracle JDK (1.6.0.34)
Git (1.7.9.5)
repo (1.12.2)
Cross toolchains

lunch (I)
made
out/target/product/(I)/*.img
(system.img, boot.img …)

lunch (II)
made
out/target/product/(II)/*.img
(system.img, boot.img …)
Flashing build artifacts - The Android way

**HOST**

- Android build System

**TARGET**

- BSP
- Kernel
- Drivers (?)
- Userspace (?)
- Shell (?)
- Graphics (?)
- ...

*\.img:
  - system, data, recovery, boot (kernel, ramdisk)

Build flow:

- Android-X86 over QEMU
- Windows over VirtualBox
- Linux over VmWare
- Android Emulator (X86)
- Android Emulator (ARM)
- LFS over UML
ROM flashing

- ROM Flashing (simplified) - is the action of transferring the build system output (a.k.a “Build Artifacts”) onto the target memory (i.e. flash, EEPROM, Hard drive, RAM, etc).
- Once the ROM is flashed, and assuming it is functional, it will be loaded and run when your target is power cycled / reset.
- It is the responsibility of the **Bootloader** to have the target’s CPU fetch, load and execute the ROM contents.
Embedded Development Example - The Android way - Flashing Maguro

- Assuming I would like to build the AOSP for my maguro device.
  - The Host is My laptop, running Ubuntu 12.04 as its Operating System.
  - The Target is an Samsung Galaxy Nexus GSM.
    - Before flashing - it is running a stock ROM
    - After flashing - it will be running what I built using the AOSP!
  - The “Flashing” occurs when we:
    - Reboot to fastboot mode
    - Flash the boot.img, system.img etc.
      of the build output directory (out/target/product/maguro)
In the previous module we built an aosp_x86-eng build variant of the AOSP.

- **The Host** is My laptop, running Ubuntu 12.04 as its Operating System.
- **The Target** is an Android Emulator - running what I built using the AOSP!
- The “Flashing” pseudo-occurs when we run the emulator, and it loads the system.img, userdata-qemu.img, kernel and cache.img of the build output directory (out/target/product/generic-x86)
And Back to the Android World!

ELC/ABS
April 2014
Android ROM components

Traditional terminology – whatever lies on the read-only partitions of the device's internal flash memory:

- **Recovery Mode:**
  - Recovery Image (kernel + initrd)

- **Operational Mode:**
  - Boot Image (kernel + initrd)
  - System Image

- The magical link between the two:
  - Misc

What is *not* a part of the ROM?

- User data: /data, /cache, /mnt/sdcard/…
Since Android is Linux at its core, we can examine its storage layout via common Linux tools:

```
shell@android:/ $ df
```

<table>
<thead>
<tr>
<th>Filesystem</th>
<th>Size</th>
<th>Used</th>
<th>Free</th>
<th>Blksize</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev</td>
<td>487M</td>
<td>32K</td>
<td>487M</td>
<td>4096</td>
</tr>
<tr>
<td>/mnt/secure</td>
<td>487M</td>
<td>0K</td>
<td>487M</td>
<td>4096</td>
</tr>
<tr>
<td>/mnt/asec</td>
<td>487M</td>
<td>0K</td>
<td>487M</td>
<td>4096</td>
</tr>
<tr>
<td>/mnt/obb</td>
<td>487M</td>
<td>0K</td>
<td>487M</td>
<td>4096</td>
</tr>
<tr>
<td>/system</td>
<td>639M</td>
<td>464M</td>
<td>174M</td>
<td>4096</td>
</tr>
<tr>
<td>/cache</td>
<td>436M</td>
<td>7M</td>
<td>428M</td>
<td>4096</td>
</tr>
<tr>
<td>/data</td>
<td>5G</td>
<td>2G</td>
<td>3G</td>
<td>4096</td>
</tr>
<tr>
<td>/mnt/shell/emulated</td>
<td>5G</td>
<td>2G</td>
<td>3G</td>
<td>4096</td>
</tr>
</tbody>
</table>
Android ROM Storage layout: "Standard Linux"

```
shell@android:/ $ mount
rootfs / rootfs ro,relatime 0 0
tmpfs /dev tmpfs rw,nosuid,relatime,mode=755 0 0
devpts /dev/pts devpts rw,relatime,mode=600 0 0
proc /proc proc rw,relatime 0 0
sysfs /sys sysfs rw,relatime 0 0
debugfs /sys/kernel/debug debugfs rw,relatime 0 0
```
Android ROM Storage layout: "Standard Android"

```sh
none /acct cgroup rw,relatime,cpuacct 0 0
tmpfs /mnt/secure tmpfs rw,relatime,mode=700 0 0
tmpfs /mnt/asec tmpfs rw,relatime,mode=755,gid=1000 0 0
tmpfs /mnt/obb tmpfs rw,relatime,mode=755,gid=1000 0 0
none /dev/cpuctl cgroup rw,relatime,cpu 0 0
/dev/block/platform/sdhci-tegra.3/by-name/APP /system ext4 ro,relatime,user_xattr,acl,bARRIER=1,
data=ordered 0 0
/dev/block/platform/sdhci-tegra.3/by-name/CAC /cache ext4 rw,nosuid,nodev,noatime,
errors=panic,user_xattr,acl,bARRIER=1,nomblk_io_submit,data=ordered,discard 0 0
/dev/block/platform/sdhci-tegra.3/by-name/UDA /data ext4 rw,nosuid,nodev,noatime,errors=panic,
user_xattr,acl,bARRIER=1,nomblk_io_submit,data=ordered,discard 0 0
/dev/fuse /mnt/shell/emulated fuse rw, nosuid, nodev, relatime, user_id=1023, group_id=1023,
default_permissions,allow_other 0 0
```
### Android ROM Storage Layout

```
shell@android:/ $ cat /proc/partitions

<table>
<thead>
<tr>
<th>major</th>
<th>minor</th>
<th>#blocks</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>179</td>
<td>0</td>
<td>7467008</td>
<td>mmcblk0</td>
</tr>
<tr>
<td>179</td>
<td>1</td>
<td>12288</td>
<td>mmcblk0p1</td>
</tr>
<tr>
<td>179</td>
<td>2</td>
<td>8192</td>
<td>mmcblk0p2</td>
</tr>
<tr>
<td>179</td>
<td>3</td>
<td>665600</td>
<td>mmcblk0p3</td>
</tr>
<tr>
<td>179</td>
<td>4</td>
<td>453632</td>
<td>mmcblk0p4</td>
</tr>
<tr>
<td>179</td>
<td>5</td>
<td>512</td>
<td>mmcblk0p5</td>
</tr>
<tr>
<td>179</td>
<td>6</td>
<td>10240</td>
<td>mmcblk0p6</td>
</tr>
<tr>
<td>179</td>
<td>7</td>
<td>5120</td>
<td>mmcblk0p7</td>
</tr>
<tr>
<td>179</td>
<td>8</td>
<td>512</td>
<td>mmcblk0p8</td>
</tr>
<tr>
<td>179</td>
<td>9</td>
<td>6302720</td>
<td>mmcblk0p9</td>
</tr>
</tbody>
</table>
```
So, where is my stuff?!

shell@android:/ $ ls -l /dev/block/platform/sdhci-tegra.3/by-name/

lrwxrwxrwx root root 2013-02-06 03:54 APP -> /dev/block/mmcblk0p3
lrwxrwxrwx root root 2013-02-06 03:54 CAC -> /dev/block/mmcblk0p4
lrwxrwxrwx root root 2013-02-06 03:54 LNX -> /dev/block/mmcblk0p2
lrwxrwxrwx root root 2013-02-06 03:54 MDA -> /dev/block/mmcblk0p8
lrwxrwxrwx root root 2013-02-06 03:54 MSC -> /dev/block/mmcblk0p5
lrwxrwxrwx root root 2013-02-06 03:54 PER -> /dev/block/mmcblk0p7
lrwxrwxrwx root root 2013-02-06 03:54 SOS -> /dev/block/mmcblk0p1
lrwxrwxrwx root root 2013-02-06 03:54 UDA -> /dev/block/mmcblk0p9
lrwxrwxrwx root root 2013-02-06 03:54 USP -> /dev/block/mmcblk0p6

Legend: APP is system, SOS is recovery, UDA is for data...
Why should we care about it?

For a couple of reasons

- Backup
- Recovery
- Software updates
- Error checking
- Board design
- Curiosity
- ...
Android Open Source Project

- “Semi-Open source”
- Maintained by Google
- Contributions accepted using “gerrit”
- Mostly Apache licensed
- Provides templates for building an Android system, including bootloaders etc.
- Vendors derive their products for their hardware layout (BSP, binaries, etc.)
- Provides the complete source code (but usually missing proprietary binaries) for a bunch of supported devices (e.g. Galaxy Nexus, Nexus 4/7/10, Android Emulator)
AOSP ROM building

In a single line:
- just do whatever they say in [http://source.android.com](http://source.android.com)

In a bit more:
- Set up a 64bit Linux development machine. Officially Supported:
  - Ubuntu 10.04 LTS (Lucid) for versions < JB 4.2.1
  - Ubuntu 12.04 LTS (Precise Pangolin) for versions >= JB 4.2.1
- `mkdir / cd / repo init / repo sync`
- `.build/envsetup.sh`
- `lunch <Your Config>`
- `make` # This will take a while... Make some coffee || Get` a good nap.
- `flash/boot/run/pray/debug/show off at xda-developers et al.`
When flashing to devices – make sure the bootloader is unlocked. For “Google phones”:

- `adb reboot-bootloader`
- `fastboot oem unlock`
- Confirm on device

Then you can flash all images using “fastboot -w flashall”, or particular images using “fastboot flash -w <partition> <image>”

Some tips on flashing custom builds:

- Having trouble using “fastboot flash” due to mismatched broadband versions?
- Try modifying `device/<vendor>/<product>/board-info.txt`
- Before building, make sure you have the “binary-blobs”, under the `vendor/` subtree (note the difference from `device/`)
  - Hint: `proprietary-blobs.txt`
Building kernels

- Get a kernel to start from – or make one
  - 3.4+ kernel are pretty much “Android-Ready”
- Checkout/config/make
  - Don't get too freaky – avoid breaking “Userspace” (a.k.a “Android”)
- Replace prebuilt kernel with your generated bzImage
- Rebuild Android
- Pray/play/laugh/cry/show off on XDA-dev/Q&A on android-kernel / android-porting / android-*
$ git clone https://android.googlesource.com/kernel/<target>.git

Some kernel targets hosted by the AOSP:

- Common - common kernel tree. Based on Linux 3.4+
- msm – Qualcomm msm (HTC Nexus One, LG Nexus 4)
- Omap – TI's OMAP (Samsung Galaxy Nexus)
- Tegra – Nvidia's Tegra (Motorola Xoom, Asus Nexus 7)
- Exynos - Samsung Exynos (Samsung Nexus 10)
- Goldfish - Android emulator
Before we get our hands “dirty”, there is something I want you to know.

That something is how things were done for most of the Android project lifetime.

More precisely up until Android 4.2.

Feel free to stick to your chairs and “enjoy” some historic moments in the *Museo di Android Internals*
The Goldfish Kernel Version has traditionally been 2.6.29.
  ○ Even 4 years after the kernel.org release.
  ○ Until Android 4.2, where it was upgraded by default to 3.4

A nice thing about Android – system and kernel are reasonably decoupled

“It's just an emulator” - and most of its consumers are only interested in testing applications, so “don't fix it if it ain't broken”

And trying to fix something that is not broken in the Goldfish case is extremely time consuming.
  ○ Ask the kernel maintainers who added extremely broken code to the staging area at late 2013 (too bad I stopped following LKML…).
This is a serious topic.
  ○ So serious I won’t get into it. Seriously.

So to make a (very) long story short:
  ○ It can be argued that Android kernels were not well accepted. To say the least.
  ○ This caused an unpleasant fragmentation.
  ○ Yet Android prevailed ⇒ Staging Area.

You can basically build Android from the vanilla kernel.org. You can do it without a single patch actually for a virtual machine!

Goldfish is a different (harder) topic.
  ○ Talk to me if you need a .../3.11+/3.12+/3.13+ goldfish porting.

TIP: ${ANDROID_BUILD_TOP}/external/qemu/distrib/build-kernel.sh
This is a serious topic.
○ So serious I won’t get into it. Seriously.

Fortunately I don’t have to
○ In order to get you running on your favorite **VESA** configuration
○ *Graphic acceleration* is not only serious, but also a painful point, which we will not discuss.

Grasping the concept is a bit easier on Virtual Machines for a starter, so let’s have a quick look at such.

* Graphic Acceleration is always a mess with virtual machines, so no surprise in here.
Guidelines to follow:

- Select your architecture (32/64bits, X86/arm/…, etc.)
- Enable staging area (CONFIG_STAGING=y)
- Search for ANDROID - and enable all configs
  - Some are unnecessary, but it’s a good start
- Enable VIRTIO drivers
  - CONFIG_VIRTIO, CONFIG_VIRTIO_BLK, CONFIG_VIRTIO_PCI, CONFIG_VIRTIO_NET,…
- Enable FB configurations
  - CONFIG_FB, CONFIG_FB_VESA, …
- Use the right command line when running qemu
- And don’t forget qemu=1 on the cmdline!
Mount points on standard Goldfish 2.6.29 kernel:

```
# mount
rootfs / rootfs ro 0 0
tmpfs /dev tmpfs rw,nosuid,mode=755 0 0
devpts /dev/pts devpts rw,mode=600 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
tmpfs /mnt/asec tmpfs rw,mode=755,gid=1000 0 0
tmpfs /mnt/obb tmpfs rw,mode=755,gid=1000 0 0
/dev/block/mtdblock0 /system yaffs2 ro 0 0
/dev/block/mtdblock1 /data yaffs2 rw,nosuid,nodev 0 0
/dev/block/mtdblock2 /cache yaffs2 rw,nosuid,nodev 0 0
# cat /proc/mtd
dev: size erasuresize name
mtd0: 0b460000 00020000 "system"
mtd1: 04000000 00020000 "userdata"
mtd2: 04000000 00020000 "cache"
```

## Note
Yaffs2 is obsolete. On ICS and JB devices /system is mounted as ext4
Android emulator storage (Goldfish kernel, Kit-Kat)
Android emulator storage (Custom vanilla kernel)
This is an image based on one of the earliest JB versions. Not much, if any has changed since the very early days of “Cupcake” and “Donut.”
First and foremost: Build for X86 and use **KVM**!
  ○ Check capability with “kvm-ok”
  ○ Feature must be enabled in your computer's bios
  ○ cat /proc/cpuinfo and search for vmx/avm(intel VT/AMD-V)

- Use hardware keyboard
- Much more comfortable then “touching” the soft keyboard
- Although there are uses for that
- Enable keyboard in external/qemu/android/avd/hardware-properties.ini – and rebuild external/qemu
- Windows users: Use **HAXM** (Intel's HW Acceleration Manager)
There are more emulation configurations which are supposed to be supported by AOSP, but tend to be broken:

- Building for non Linux devices from Linux
  - lunch sdk-eng && make win_sdk
- Building for VirtualBox and other virtual machines:
  - lunch vbox_x86-eng
  - make android_disk_vdi
  - Translate VDI image to your VM hard-drive format (e.g. qcow...)

Motivation for using such configurations:
Development teams working with different Operating Systems, but willing to use the same emulated platform
Motivation - fast linux bringup procedure
  ○ First, bring-up the target OS on a virtual machine
  ○ Verify basic functionality
  ○ Then adjust for a designated hardware

How to do it?
  ○ Short answer - use emulator images with some adjustments, mount ext4, set sdcard etc...
  ○ Pragmatic answer: In the next session
When to use the emulator

The short answer would be – whenever you can.

- Great for application development
  - when used with KVM/HAXM
- Has no dependency on a particular hardware
- Very easy to build
- Integrates well with the AOSP tools
- Relatively well documented

Overall – it is a good ROM.

😊 Most used ROM for a reason.
Running the Android Emulator on Windows
Various forks to the Android Open Source Project:

- **AOSP** - (4.4+ OPENMASTER/KVT49L upstream)
  - The root of all (good?)
- **Android-X86** (KOT49H upstream, JLS36I last stable release)
- **Android-IA** (JDQ39 upstream)
- Many other forks
  - CyanogenMod
  - Buildroid/AndroVM
  - And many others... Not all are known or Open-Sourced
CyanogenMod (special guest star)

A custom, open source distribution spawned off the AOSP

- Provides optimizations and support for over 40 different devices, along with binaries
- Builds routine similar to AOSP (note: “brunch”)
- http://wiki.cyanogenmod.com/wiki/Main_Page
Android, X86, Google, Intel and Android-X86
Android and X86

X86 ROMs (by chronological order):

- **Android-X86** (Debut date: 2009)
  - [http://android-x86.org](http://android-x86.org)
- **Emulator-x86** (Debut date: 2011)
  - [http://source.android.com](http://source.android.com)
- **Android-IA** (Debut date: 2012)
  - [https://01.org/android-ia](https://01.org/android-ia)
The common reference, having the most recent version of the Android platform (Userspace) versions.

Provides the QEMU based **Android Emulator**:

+ Works on any hosted OS
+ Supports multiple architectures
  - But slow on non X86 ones
- Performs terribly if virtualized
- Has no installer for X86 devices
- Very old kernel

+/- An **emulator**. For better and for worse.
+ Developed by the open source community
+ Developer/Linux user friendly
+ Multi-Boot friendly
+ Generally supports many Intel and AMD devices
+/- But of course requires specific work on specific HW
+ VM friendly
+ Mature, Recognized and stable
- Delays in new releases (You can help!)
  - Latest version (4.4) is still very buggy, but it's been out only for a week
  +/- Latest stable version (4.3) still needs some work for some devices
+ The ICS 4.0.4 release is amazing - including running ARM apps
Android-IA

+ Installer to device
+ Relatively new versions of android and kernel
+ Works great on Ivy-bridge devices
+ Integrated Ethernet Configuration Management
- Development for devices based on Intel solutions only
- Very unfriendly to other OS's
- Not developer friendly – unless they make it such
- Community work can be better. But it seems to be getting better
- Intel phones are not based on it (at the moment)
+ Made impressive progress in early 2013
- But suspended development at Android 4.2.2
+ Project resumption at Kit-Kat?, a bit late, but so far looks good (April 2014)
Android is Linux

- Android is Linux
  - Therefore the required minimum to run it would be:
    - A Kernel
    - A filesystem
    - A ramdisk/initrd... Whatever makes you happy with your kernel's init/main.c's run_init_process() calls.
      See http://lxr.linux.no/linux+v3.6.9/init/main.c
  - This means that we can achieve full functionality with
    - A kernel (+ramdisk)
    - A rootfs where Android system/ will be mounted (ROM)
    - Some place to read/write data
Android-IA is, of course, Linux as well. However, it was designed to conform to Android OEM's partition layout, and has no less than 9 partitions:

- boot - flashed boot.img (kernel+ramdisk.img)
- recovery - Recovery image
- misc - shared storage between boot and recovery
- system - flashed system.img - contents of the System partition
- cache - cache partition
- data - data partition
- install - Installation definition
- bootloader - A vfat partition containing android syslinux bootloader (<4.2.2)
  - A GPT partition containing gummiboot (Only option in 4.2.2)
- fastboot - fastboot protocol (flashed droidboot.img)

**Note:** On android-ia-4.2.2-r1, the bootable live.img works with a single partition, enforcing EFI. It still has its issues - but it is getting there.
Android-X86 is Linux

- One partition with two directories
  - First directory – grub (bootloader)
  - Second directory – files of android (SRC)
    - kernel
    - initrd.img
    - ramdisk.img
  - system
  - data

- This simple structure makes it very easy to work and debug

Note: Also comes with a live CD/installer. Very convenient.
Android-IA boot process

- Start bootloader
- The bootloader starts the combined kernel + ramdisk image (boot.img flashed to /boot)
- At the end of kernel initialization Android's
  - /init runs from ramdisk
- File systems are mounted the Android way – using 
  *fstab.common* that is processed (*mount_all_ all* command) from in *init.<target>.rc*
Android-X86 boot process

- Start bootloader (GRUB)
- bootloader starts kernel + initrd (minimal linux) + kernel command line
- At the end of kernel initialization
  - run the /init script from initrd.img
  - load some modules, etc.
  - At the end change root to the Android file system
- Run the /init binary from ramdisk.img
  - Which parses init.rc, and starts talking “Android-ish”
Which one is better?

It depends what you need:
- Developer options?
- Debugging the init process?
- Support for Hardware?
- Support for OTA?
- Licensing?
- Participating in project direction?
- Upstream features?
- ...

There is no Black and White.
An hybrid approach

- Use Android-X86 installer system
- And put your desired android files (*matching* kernel/ramdisk/system) in the same partition.
- Use the Android-X86 **chroot** mechanism
  - Critics: Does redundant stuff
  - But that's just a hack anyway – devise specific solutions for specific problems
- This way, we can multiboot various projects:
  - Android-IA
  - AOSP
  - Any other OS...

**Note:** You can also use chroot mechanism on any Linux Distribution, from userspace! But this is *significantly* harder...
Multi-boot recipe with legacy GRUB (simplified)

Repartition existing Linux partition (Don't do that...)
Install Android-X86
Add entries to GRUB
Reboot to Android-X86 debug mode
Copy Android-IA files from a pendrive or over SCP
  For the former: cp /mnt/USB/A-IA/ /mnt && sync
  /mnt is the root of Android-X86 installed partition
  (e.g. (hd0,1)/...)
Update GRUB entries and update GRUB
Voila :-)
Less simplified procedure: Debug GRUB... :-(
** Note: Replace Android-IA with AOSP to boot AOSP built files (system.img / kernel / ramdisk.img) on your target device.
Multi-boot recipe using GRUB2

- Repartition existing Linux partition (Don't do that...)
- Create a mount point for your multi-booting android
  - Can make a partition per distribution, it doesn't really matter.
  - For this example let's assume all Android distributions will co-exist on the same partition, and that it is mounted to /media/Android-x86
- Build your images
  - AOSP: Discussed before
  - Android-x86:
    - . build/envsetup.sh && lunch android_x86-<variant> \\ 
      && make iso_img
  - Android-IA:
    - . build/envsetup.sh && lunch core_mesa-<variant> \\ 
      && make allimages
    - . build/envsetup.sh && lunch bigcore-<variant> && make allimages

**<variant>** is either one of the following: user, userdebug, eng
Multi-boot recipe using GRUB2 (cont.)

- Create directories for your projects (e.g. jb-x86, A-IA, AOSP) under your mount point (e.g. /media/Android-x86)
- From Android-X86's out/product/target: Copy `initrd.img` to all projects.
  - Can of course only copy ramdisk to one location.
- From all projects – copy `kernel`, `ramdisk.img`, `system/` and `data/` to the corresponding directory under your mount point.
- Add entries to GRUB and update grub.
  - # e.g. sudo vi /etc/grub.d/40_custom && update-grub
**Multi-boot recipe with GRUB2 - A numerical example**

```
$ df

<table>
<thead>
<tr>
<th>Filesystem</th>
<th>1K-blocks</th>
<th>Used</th>
<th>Available</th>
<th>Use%</th>
<th>Mounted on</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/sda5</td>
<td>451656948</td>
<td>394848292</td>
<td>34199920</td>
<td>93%</td>
<td>/</td>
</tr>
<tr>
<td>udev</td>
<td>1954628</td>
<td>4</td>
<td>1954624</td>
<td>1%</td>
<td>/dev</td>
</tr>
<tr>
<td>tmpfs</td>
<td>785388</td>
<td>1072</td>
<td>784316</td>
<td>1%</td>
<td>/run</td>
</tr>
<tr>
<td>none</td>
<td>5120</td>
<td>0</td>
<td>5120</td>
<td>0%</td>
<td>/run/lock</td>
</tr>
<tr>
<td>none</td>
<td>1963460</td>
<td>2628</td>
<td>1960832</td>
<td>1%</td>
<td>/run/shm</td>
</tr>
<tr>
<td>/dev/sdal</td>
<td>15481360</td>
<td>5165416</td>
<td>9529464</td>
<td>36%</td>
<td>/media/Android-x86</td>
</tr>
</tbody>
</table>
```
A numerical example (cont.)- /etc/grub.d/40_custom

```plaintext
### JB-X86
menuentry 'jb-x86' --class ubuntu --class gnu-linux --class gnu --class os {
  recordfail
  insmod gzio
  insmod part_msdos
  insmod ext2
  set root='(hd0,msdos1)'
  echo 'Loading Android-X86'
  linux /jb-x86/kernel quiet androidboot.hardware=android_x86 video=-16 SRC=/jb-x86
  initrd /jb-x86/initrd.img
}
```
A numerical example (cont.) - /etc/grub.d/40_custom

### android-IA

menuentry 'Android-IA' --class ubuntu --class gnu-linux --class gnu --class os {
  recordfail
  insmod gzio
  insmod part_msdos
  insmod ext2
  set root='(hd0,msdos1)'
  echo 'Loading Android-IA'
  linux /A-IA/kernel console=ttyS0 pci=noearly console=tty0 loglevel=8 androidboot.hardware=ivb SRC=/A-IA
  initrd /A-IA/initrd.img
}
In this session:
- We have listed various ways to build ROMs for
  - AOSP devices
  - AOSP emulator(-X86)
  - Android-X86
  - Android-IA
- We have also discussed multi booting several configurations using the Android-X86 build system

In the next module, we will:
- Explore the Android build system
- See how to create and modify those projects for easy customizable X86 developer friendly targets!
Building Our First Customized Device

https://github.com/ronubo/
Outline

● The build/ folder
● The device/ folder
● Adding a new device
● QEMU challenges
  ○ kernel
  ○ network
  ○ graphics
  ○ sdcard
● No slides! (pay attention!)
Challenges

- Android build system sometimes varies between versions
- Different Android build systems may have their nuances
- Android runtime varies between versions
- Binary blobs may, or may not be available
- Building takes time. Being “smart” may take more time due to Dexopt.
- OS/QEMU optimal combination varies.
- Initial bringup may be challenging
References

- The AOSP is hosted at [http://source.android.com](http://source.android.com)
- The Android-x86.org project is hosted at [http://Android-X86.org](http://Android-X86.org)
- The Android-IA project is hosted at [https://01.org/android-ia](https://01.org/android-ia)
- Device trees shown in the next session are available at [https://github.com/ronubo/AnDevCon](https://github.com/ronubo/AnDevCon)
- *Introduction to Embedded Android course* - Ron Munitz. Taught at Afeka College of Engineering, Tel-Aviv, Israel
- You are welcome to contact me in the social networks (@ronubo)
Thank You!

Questions/Consulting/Training requests: ron@thepscg.com