



Android boot and its optimization

Taichun Yuan

01/19/2015



Confidential and Proprietary

Freescale, the Freescale logo, Altivec, C-5, CodeTEST, CodeWarrior, ColdFire, ColdFire+, C-Ware, the Energy Efficient Solutions logo, Kinetis, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, Qorivva, SafeAssure, the SafeAssure logo, StarCore, Symphony and VortIQ are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Airfast, BeeKit, BeeStack, CoreNet, Flexis, Layerscape, MagniV, MXC, Platform in a Package, QorIQ Converge, QUICC Engine, Ready Play, SMARTMOS, Tower, TurboLink, UMEMS, Vybrid and Xtrinsic are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners. © 2014 Freescale Semiconductor, Inc.



Preface

- This slide will provide a “go through” for Android’s boot process flow and a simple introduction to some directions of speeding up the boot.
- Major purpose of this slide is to provide some tips/hints on how to do Android boot optimization. It’s NOT a completed solution.
- The Android boot flow will get uboot/Linux involved, so, we will cover the boot optimization on uboot and Linux as well.
- All discussion will be based on the following configurations:
 - Android 4.4.3 release for i.MX6 which contains a 3.10.53 Linux kernel.
 - Sabre-SDP board for i.MX6Q, boot with eMMC.

Agenda

- Overview of the system boot flow.
- Ideas for uboot/kernel optimization.
- Android boot flow.
- Android boot optimization
 - Popular directions and solutions.
 - Measurement and analyzing tool.
- A try on IMX6

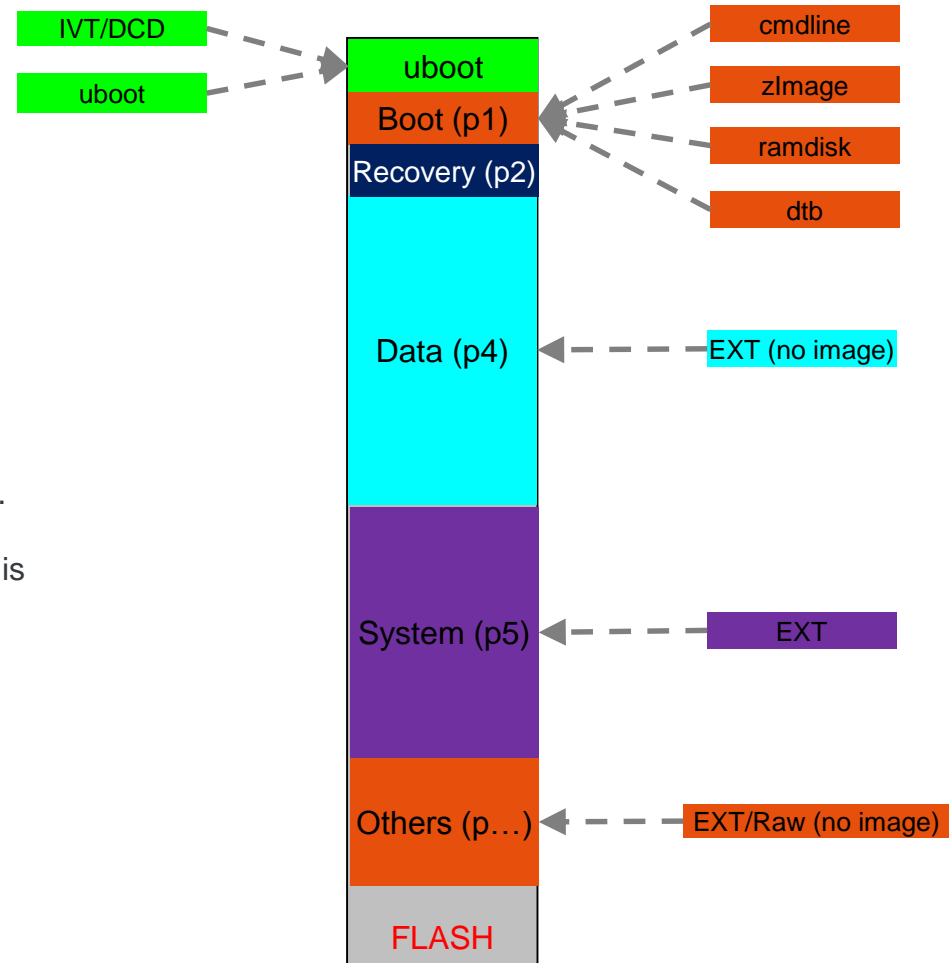


Overview of the system boot flow

Image structure (no HAB)

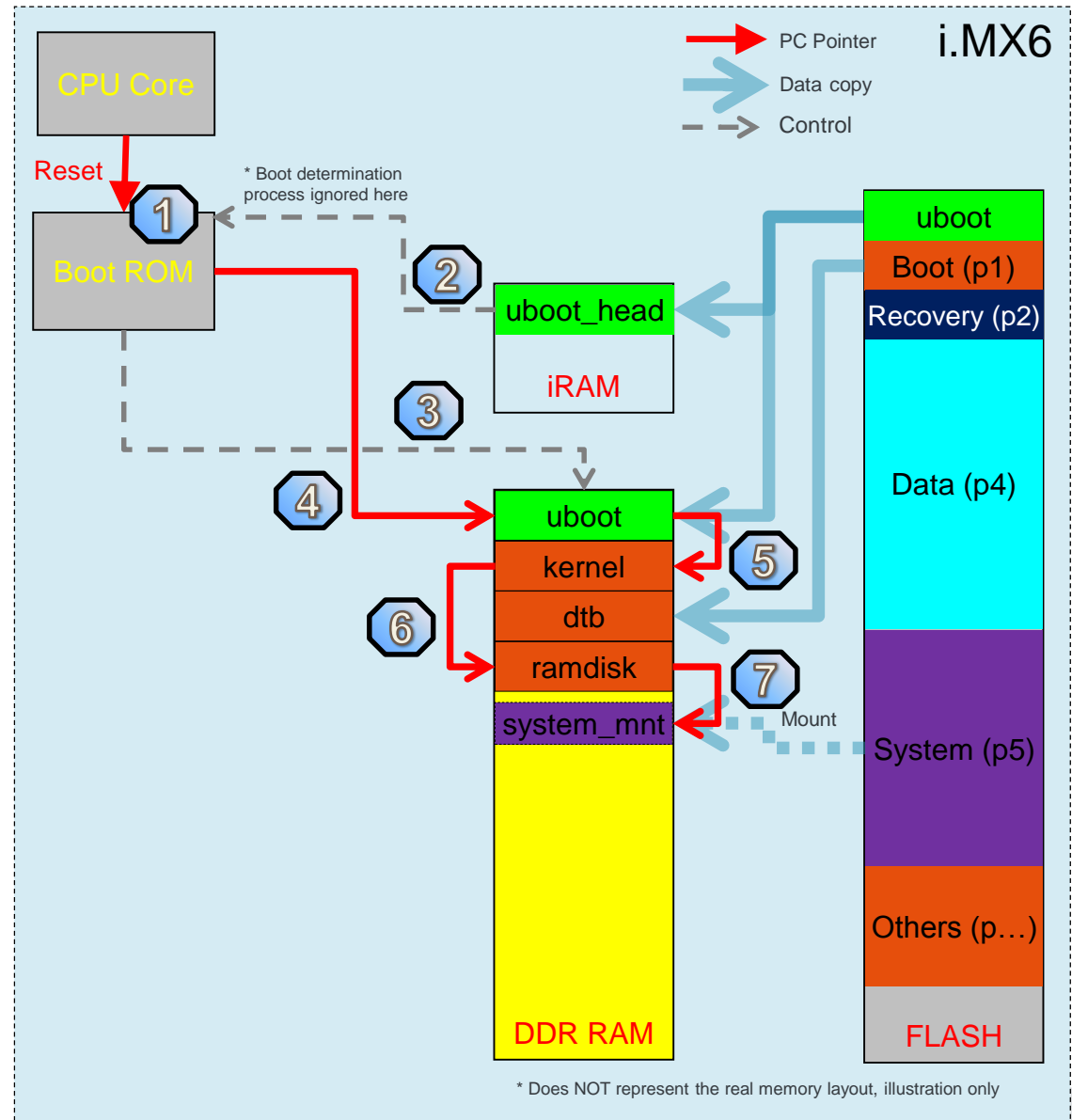
Before we start with overall boot flow, we need to be clear about the image structure:

1. **uboot.imx**: contains IVT/DCD data for DDR initialization and uboot itself.
2. **boot.img** (defined in bootimg.h):
 - a) cmdline: preset kernel command line.
 - b) zImage: Linux kernel.
 - c) ramdisk: RAM based file system image. Used as rootfs for Android.
 - d) dtb: The device tree blob. exists in boot.img as the 2-Stage image.
3. **recovery.img**: has a same structure as boot.img. Normally, only the ramdisk is different from the boot.img.
4. **Data partition**: holding user data, we don't have image built for this. Only format it when downloading.
5. **system.img**: holding system binaries and resources.
6. Others:
 - a) **misc**: for BCB (Bootloader Control Block) which is used as communication between main system and bootloader when doing recovery boot. We're not using this mechanism, SNVS_LPGPR is used instead.
 - b) **cache**: for recovery. Holding command from main system and all materials for doing upgrade.



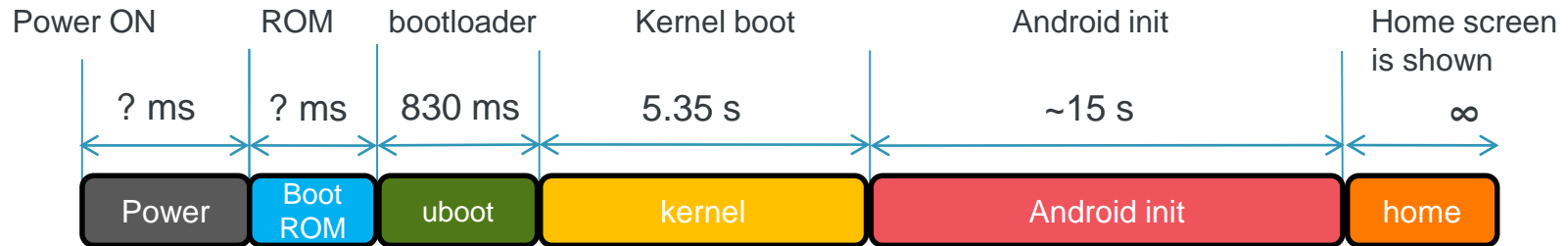
Boot flow

1. Core reset. PC Jump to Boot ROM for some basic initialization and boot decision.
2. Boot ROM read IVT/DCD header from flash.
3. Boot ROM parse IVT/DCD and init the DDR.
4. Uboot will be loaded into DDR. PC then jumps to it.
5. Uboot loads boot.img from flash, parse it and place kernel/dtb/ramdisk to specific location. Then jumps to kernel's bootstrap loader.
6. Kernel boots. After initialization, it will jumps to the "init" in the ramdisk.
7. "init" process will then done a lot of setup, including mounting system/user/... partitions. It will then start some core services for Android.



Boot flow – time breakdown

- Current status on pre-built GA image.



* The time between “Power ON” and “uboot” may have connection with specific platform design, such as power ramp time and boot device type.

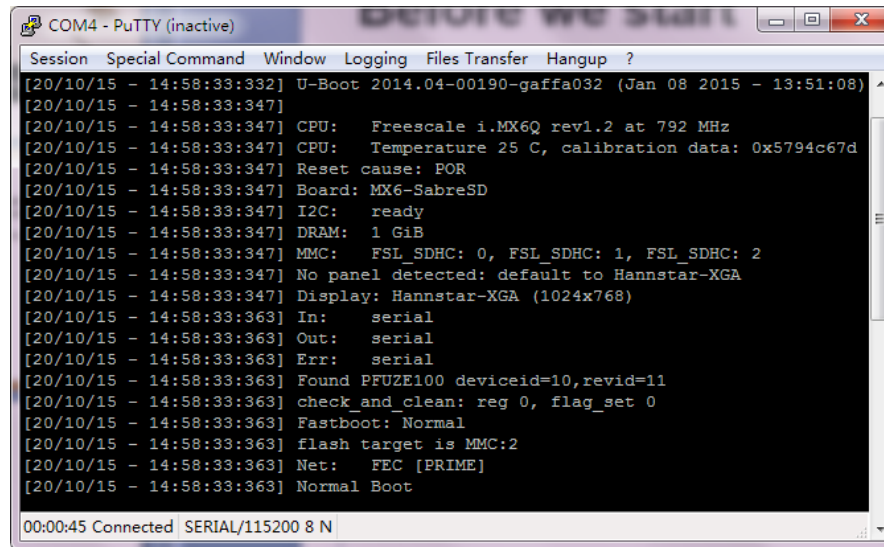
* The “Android init” here covers not only the “init” process, but also all initialization between kernel and home screen.



Ideas for uboot/kernel optimization

Before we start

- We need some way to measure the boot time:
 - Host side timestamp. Recommended: “ExtraPutty”.



The screenshot shows a PuTTY terminal window titled "COM4 - PuTTY (inactive)". The terminal displays the U-Boot boot sequence for a Freescale i.MX6Q. The logs include timestamps in brackets, such as [20/10/15 - 14:58:33:332]. The boot process shows the U-Boot version (2014.04-00190-gaffa032), CPU information (Freescale i.MX6Q rev1.2 at 792 MHz), temperature (25 C), reset cause (POR), board name (MX6-SabreSD), I2C status (ready), DRAM (1 GiB), MMC configuration (FSL_SDHC: 0, 1, 2), display information (Hannstar-XGA 1024x768), and serial port settings (In: serial, Out: serial, Err: serial). It also shows the detection of a PFUZE100 device, fastboot status (Normal), and flash target (MMC:2). The boot process concludes with "Normal Boot". The status bar at the bottom indicates "00:00:45 Connected SERIAL/115200 8 N".

```
COM4 - PuTTY (inactive)
Session Special Command Window Logging Files Transfer Hangup ?
[20/10/15 - 14:58:33:332] U-Boot 2014.04-00190-gaffa032 (Jan 08 2015 - 13:51:08)
[20/10/15 - 14:58:33:347]
[20/10/15 - 14:58:33:347] CPU: Freescale i.MX6Q rev1.2 at 792 MHz
[20/10/15 - 14:58:33:347] CPU: Temperature 25 C, calibration data: 0x5794c67d
[20/10/15 - 14:58:33:347] Reset cause: POR
[20/10/15 - 14:58:33:347] Board: MX6-SabreSD
[20/10/15 - 14:58:33:347] I2C: ready
[20/10/15 - 14:58:33:347] DRAM: 1 GiB
[20/10/15 - 14:58:33:347] MMC: FSL_SDHC: 0, FSL_SDHC: 1, FSL_SDHC: 2
[20/10/15 - 14:58:33:347] No panel detected: default to Hannstar-XGA
[20/10/15 - 14:58:33:347] Display: Hannstar-XGA (1024x768)
[20/10/15 - 14:58:33:363] In: serial
[20/10/15 - 14:58:33:363] Out: serial
[20/10/15 - 14:58:33:363] Err: serial
[20/10/15 - 14:58:33:363] Found PFUZE100 deviceid=10, revid=11
[20/10/15 - 14:58:33:363] check_and_clean: reg 0, flag_set 0
[20/10/15 - 14:58:33:363] Fastboot: Normal
[20/10/15 - 14:58:33:363] flash target is MMC:2
[20/10/15 - 14:58:33:363] Net: FEC [PRIME]
[20/10/15 - 14:58:33:363] Normal Boot
00:00:45 Connected SERIAL/115200 8 N
```

- Kernel printk timestamp: “CONFIG_PRINTK_TIME=y”.
- Stop watch, LED, ...

Ideas for boot optimization

- Size / built-in modules
- build-in modules' init speed
- Kernel: un-compressed kernel
- Kernel: built-in vs. module
- Other ideas

Size / built-in modules

- What to do with size?
 - Reduce loading time
 - Reduce init time by removing un-necessary modules.
- How?
 - Define target application: this is to make sure the image contains minimal set of software needed for the board/application.
 - Breakdown: we need some tools...
- Device Tree?
 - The concept of DT is making the kernel bigger than ever since it needs to be a “super set” which contains all possible code and let DTB “describe” / “choose” which device to support.
 - In this slide, we will NOT touch DTB because it’s defined as hardware description and the “hardware” can NOT be slimmed.

Size / built-in modules – breakdown – size

Small tips for breakdown (applicable for uboot & kernel):

```
find . -maxdepth 2 -name "built-in.o" | xargs size | sort -n -r -k 4
```

Sample result for kernel:

1488823	60422	104588	1653833	193c49	./drivers/built-in.o
1272154	56526	38628	1367308	14dd0c	./net/built-in.o
606250	26376	352068	984694	f0676	./kernel/built-in.o
859023	6571	4100	869694	d453e	./fs/built-in.o
279570	17382	25532	322484	4ebb4	./mm/built-in.o
256388	16764	2364	275516	4343c	./sound/built-in.o
115552	4786	16	120354	1d622	./crypto/built-in.o
85067	25223	2300	112590	1b7ce	./lib/built-in.o
100287	2082	1220	103589	194a5	./block/built-in.o
35347	804	304	36455	8e67	./security/built-in.o
19178	14829	152	34159	856f	./init/built-in.o
28280	760	8	29048	7178	./ipc/built-in.o
2312	0	0	2312	908	./firmware/built-in.o
516	0	0	516	204	./usr/built-in.o
text	data	bss	dec	hex	filename

- “text” is the code size
- “data” & “bss” are the data size (init & un-init)
- “dec” is the sum of the previous columns in decimal while “hex” is the HEX value.

Notes:

1. Without “-maxdepth 2”, a full list for all targets under current folder will be listed. Or, you can use this command step by step down into specific folder.
2. With “-name ‘*.o’”, similar list can be generated for separate object file. Useful for specific module. Example:

```
find drivers/usb/gadget/ -maxdepth 2 -name "*.o" | xargs size | sort -n -r -k 4
```

Size / built-in modules – breakdown – size cont'd

- After some text processing and importing to XLS, size list for original uboot and kernel:

uboot					kernel				
text	data	bss	dec	filename	text	data	bss	dec	filename
132565	8948	108051	249564	./common/built-in.o	4107005	318121	129768	4554894	./drivers/built-in.o
16690	624	196908	214222	./fs/built-in.o	3835172	0	0	3835172	./firmware/built-in.o
30212	51060	1423	82695	./drivers/built-in.o	2087224	20732	11540	2119496	./fs/built-in.o
18882	46	12143	31071	./net/built-in.o	1772363	75086	64244	1911693	./net/built-in.o
30098	0	30	30128	./lib/built-in.o	640947	26512	353108	1020567	./kernel/built-in.o
4733	0	0	4733	./disk/built-in.o	425995	42924	2948	471867	./sound/built-in.o
					283642	18150	25532	327324	./mm/built-in.o
					263907	11194	20	275121	./crypto/built-in.o
					175051	10592	11496	197139	./security/built-in.o
					93555	25255	2300	121110	./lib/built-in.o
					100275	2082	1220	103577	./block/built-in.o
					19434	14837	152	34423	./init/built-in.o
					29304	760	8	30072	./ipc/built-in.o
					516	0	0	516	./usr/built-in.o

This kind of table can then be used as a guide for slimming the binary. Using “menuconfig” to do the slimming is recommended.

Size / built-in modules – breakdown – nm

- “nm” can be used to analyze the symbol size in the kernel image.

```
nm --size-sort -r vmlinux
```

Sample result for kernel:

```
00040000 b __log_buf
00010000 b sync_dump_buf
00010000 b entries
00004af7 r kernel_config_data
000041d0 T hidinput_connect
00004138 t imx6sx_clocks_init
00003fec t imx6q_clocks_init
00003d70 b ipu_array
00003920 T __blockdev_direct_IO
00002d7c t ext4_fill_super
00002728 t imx6sl_clocks_init
000023e8 T binder_thread_write
00002080 b nf_frags
00002080 b ip6_frags
00002080 b ip4_frags
00002000 b page_address_maps
00002000 b page_address_htable
00002000 D init_thread_union
00002000 d crc32table_le
00002000 d crc32table_be
00002000 d crc32ctable_le
```

- “t”/“T” = “text”, code section
- “d”/“D” = “data”, initialized data
- “b”/“B” = “bss”, un-initialized data
- “r”, read-only data section
- ...

There’s a script in kernel source tree to diff two kernel images using “nm” command:

```
<kernel-src>/scripts/bloat-o-meter vmlinux.default vmlinux.altconfig
```

Size / built-in modules – Findings

Some most significant hot-spots listed here (not a full list):

- Uboot:
 - Some drivers and modules can be removed, such as: usb, un-used cmd_* commands, fastboot (android), network related (MII_*), un-used env_* device for environment storage (use default), display support (if no splash screen required).
 - Build-in file systems can all be removed: FAT/EXT since Android boot doesn't require FS support from uboot.
- Kernel:
 - EPD firmware can be removed/moved if not used.
 - Video in/out related drivers can be slimmed a lot.
 - Some un-used USB gadget drivers can be removed.
 - Many un-used FS can be removed: UBIFS, NFS, JFFS2, UDF, ...
 - Other un-used components/sub-components, such as MTD, ata, input, hid, ...

build-in modules' init speed

- Most of the time spent during boot is consumed by “initialize”.
- For both of uboot and kernel, major work for init is called “initcall”:

- Uboot:

common/board_f.c (“init_sequence_f” array)

common/board_r.c (“init_sequence_r” array)

We can only add timestamp print manually to breakdown the time for each init step.

- Kernel:

All init routine defined with Linux init API, such as “module_init()”, “late_initcall()”, etc.

A simple way to debug the time, add the following line into kernel cmdline and check the “dmesg” output:

```
initcall_debug
```


build-in modules' init speed, cont'd

- The message print itself will impact the boot time, but it can provide some trend information.

Sample output:

```
initcall init_static_idmap+0x0/0xe8 returned 0 after 0 usecs
initcall init_workqueues+0x0/0x36c returned 0 after 0 usecs
initcall init_mmap_min_addr+0x0/0x28 returned 0 after 0 usecs
...
initcall init_mtd+0x0/0xfc returned 0 after 532 usecs
initcall init_mtdblock+0x0/0xc returned 0 after 4 usecs
initcall init_ladder+0x0/0xc returned 0 after 2757 usecs
...
```

After process/sort:

Start time	Function name	Return	usec
6.165906	imx_serial_init+0x0/0x48	0	4038545
1.999378	mxcfb_init+0x0/0xc	0	1470073
10.309506	imx_wm8962_driver_init+0x0/0xc	0	1212897
8.471315	gpu_init+0x0/0x12c	0	341488
8.103319	sdhci_esdhc_imx_driver_init+0x0/0xc	0	177161
8.962669	wm8962_i2c_driver_init+0x0/0x10	0	169909
7.499705	ov5640_init+0x0/0x40	0	124343
7.698344	mma8x5x_driver_init+0x0/0x10	0	121189
7.081894	isl29023_init+0x0/0x10	0	114222
7.361302	ov5642_init+0x0/0x40	0	104195

- Now it will be straight forward to check who has consumed most of time.

build-in modules' init speed – Findings

Some most significant hot-spots listed here (not a full list, and, after the slimming):

- Uboot:
 - After slimming the size, the MMC's init routine consumes most of the time when detecting/initializing the MMC card.
- Kernel:
 - “imx_serial_init” is at the top of list. Can be easily eased by using lower loglevel (e.g. “loglevel=3”) in cmdline.
 - “mxcfb_init” take the 2nd position. The root caused has been found to be the CMA. When the FB driver tries to alloc the first frame buffer, the CMA will try to do the migration between buddy and CMA pool. The prepare stage will cost about 1s! 3.14.x kernel doesn't have this problem.

Kernel: un-compressed kernel

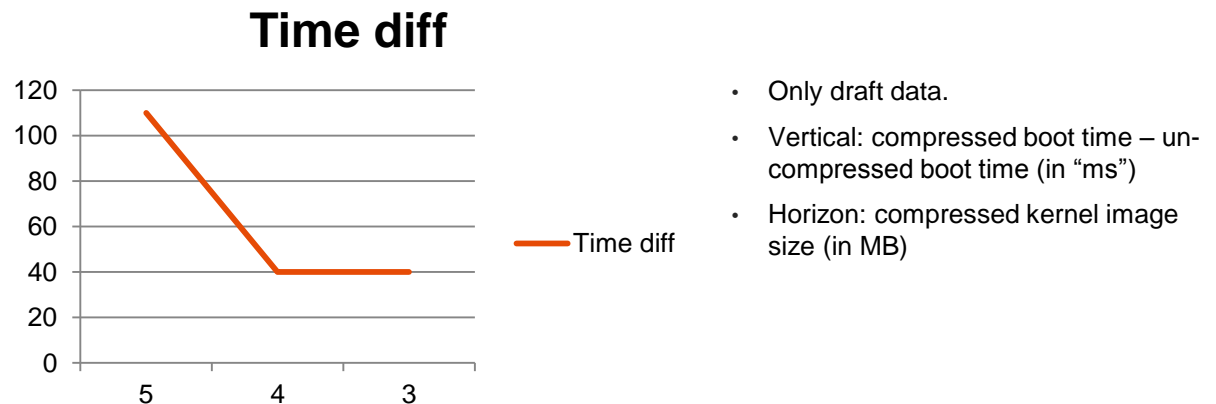
- We can use compressed (default) or uncompressed kernel to boot.
- How? Patch the “build/core/Makefile”.

```
diff --git a/core/Makefile b/core/Makefile
index 3478744..f683c23 100644
--- a/core/Makefile
+++ b/core/Makefile
@@ -811,6 +811,7 @@ TARGET_PREBUILT_KERNEL := $(PRODUCT_OUT)/kernel
KERNEL_CONFIGURE := kernel_imx/.config
TARGET_KERNEL_CONFIGURE := $(PRODUCT_OUT)/.config
KERNEL_ZIMAGE := kernel_imx/arch/arm/boot/zImage
+KERNEL_IMAGE := kernel_imx/arch/arm/boot/Image
KERNEL_OUT := $(TARGET_OUT_INTERMEDIATES)/KERNEL_OBJ

@@ -826,7 +827,7 @@ $(TARGET_KERNEL_CONFIGURE): kernel_imx/arch/arm/configs/$(TARGET_KERNEL_DEFCONF)
$(TARGET_PREBUILT_KERNEL): $(TARGET_KERNEL_CONFIGURE)
    $(MAKE) -C kernel_imx -j$(HOST_PROCESSOR) uImage $(KERNEL_ENV)
    $(MAKE) -C kernel_imx dtbs $(KERNEL_ENV)
-   install -D $(KERNEL_ZIMAGE) $(PRODUCT_OUT)/kernel
+   install -D $(KERNEL_IMAGE) $(PRODUCT_OUT)/kernel
    for dtsplat in $(TARGET_BOARD_DTS_CONFIG); do \
```

Kernel: un-compressed kernel, cont'd

- Which way to go depends on “we load faster” or “we de-compress faster”.
- Some testing done on different kernel size (in different stage of the slimming):



- Un-compressed kernel boots a little bit faster on mx6q SDP.
- The smaller the kernel becomes, the smaller the boot time differs (limited sample, might not be accurate).

Kernel: built-in vs. module

- Linux kernel supports dynamically loading kernel modules.
 - Advantages:
 - Further reduce kernel size.
 - Delay loading some modules which are not necessary for booting, such as USB, EXT FS, etc. This will remove them from initcall sequence and further reduce boot time.
 - TBD: Initcall runs in a single-thread context (do_initcalls), we will have chance to do it in a multi-thread context (on different cores) if loading it as module. ??
 - Disadvantages:
 - Need to resolve dependency manually.
 - Increase boot time in Android stage.
- Verify? (TBD)

Other ideas

Some other ideas that have NOT been tried but possible to be helpful:

- printk: printk are everywhere in kernel, and they're all built in binary. Possible to totally remove them.
- LTO: Link Time Optimization. New feature in GCC 4.7. Possible performance and size gain. Need additional patch for kernel to support it.
- syscall and cmdline parameter elimination: some of the syscall and many of the kernel parameters are not used.



Android boot flow

Before we start

- Let's take a look at the PS output after boot:

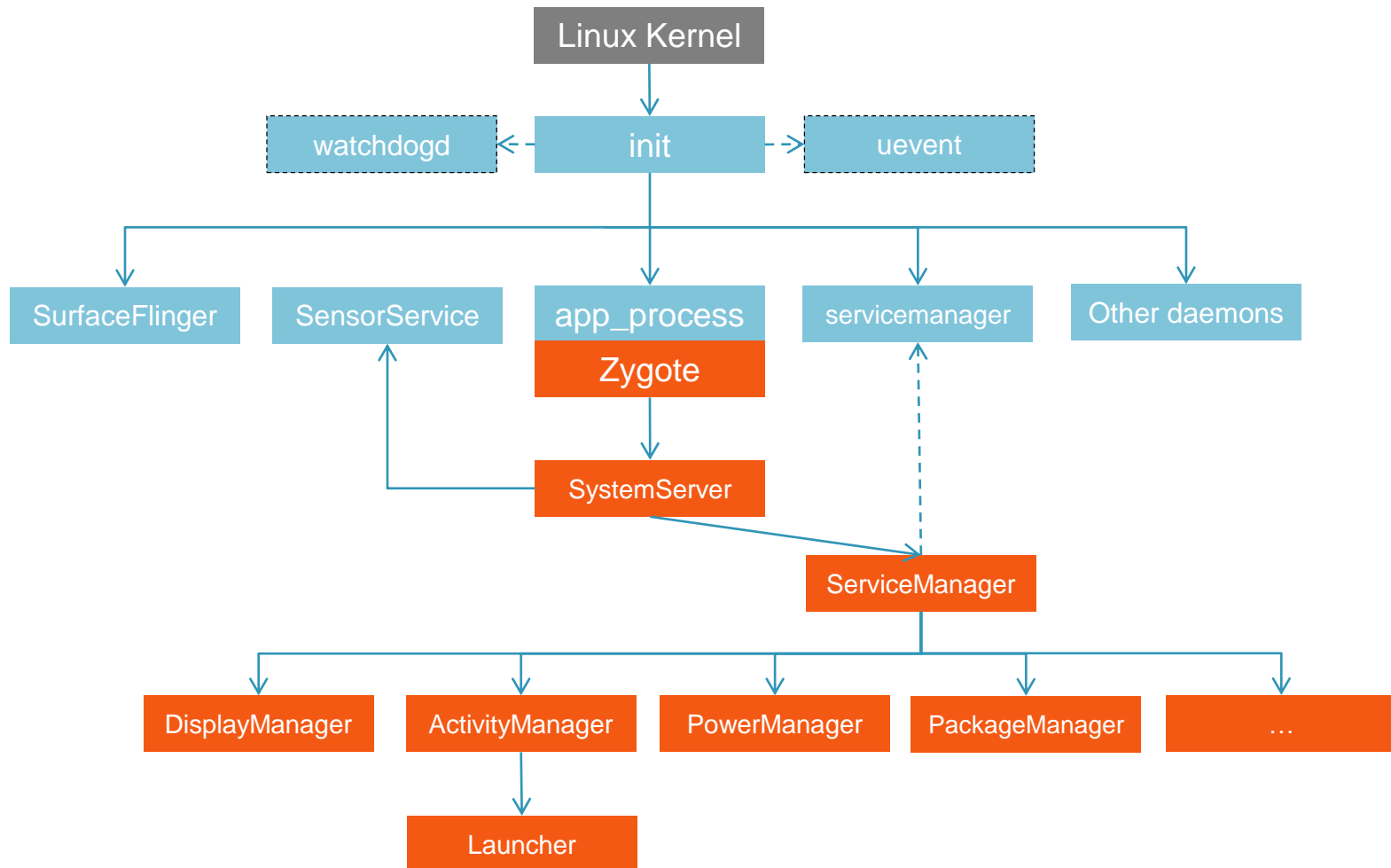
- Android's "init" take the place for normal Linux init. Process ID=1
- All processes who has a PPID = 1 means it's started by init.
- Zygote is the parent for all JAVA process. Process ID=2341

How the first Android App been brought up:
[kernel] -> init -> Zygote -> [JAVA APP]

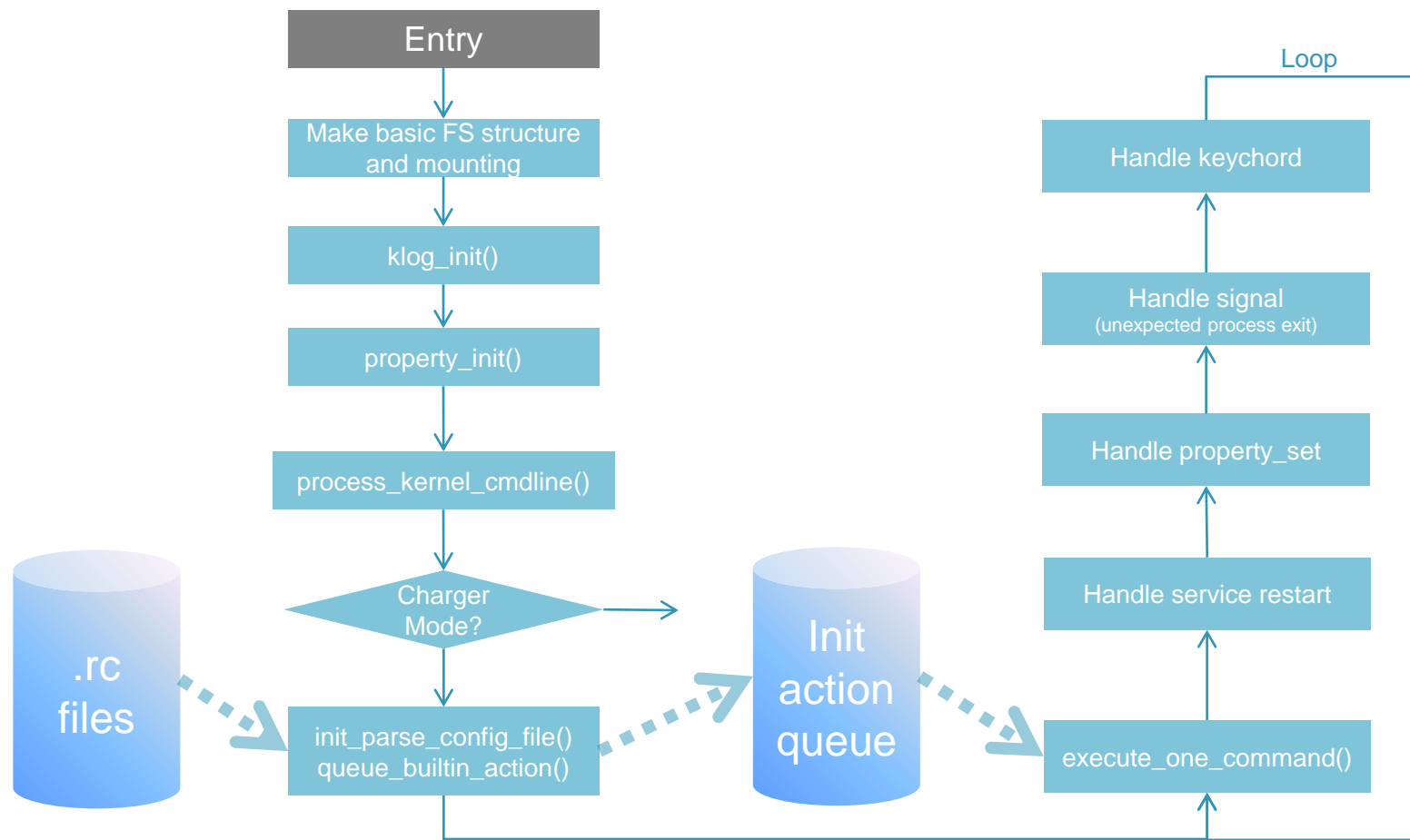
USER	PID	PPID	NAME
root	1	0	/init
root	1441	1	/sbin/ueventd
root	2330	1	/sbin/watchdogd
install	2340	1	/system/bin/installld
root	2341	1	zygote
system	2342	1	/system/bin/surfaceflinger
system	2343	1	/system/bin/servicemanager
root	2344	1	/sbin/healthd
root	2345	1	/system/bin/vold
system	2394	2341	system_server
u0_a7	2431	2341	com.android.systemui
media_rw	2453	1	/system/bin/sdcard
keystore	2454	1	/system/bin/keystore
media	2455	1	/system/bin/mediaserver
root	2456	1	/system/bin/sh
root	2457	1	/system/bin/netd
u0_a15	2838	2341	com.android.inputmethod.latin
system	2856	1	/system/bin/magd
root	2857	1	/system/bin/insvcd
u0_a1	2864	2341	android.process.media
u0_a4	2896	2341	com.android.onetimeinitializer
system	2915	2341	com.android.settings
u0_a3	2934	2341	com.android.launcher

A whole picture

Android starts from “init”...



init



Init – something to be mentioned

- “/proc/cmdline” will be parsed to convert any “androidboot.xxx=yyy” string in the kernel command line into “ro.boot.xxx=yyy” property.
- “ueventd” and “watchdogd” actually live in the same binary as “init”, running in different process (different “main()”).
- “ueventd” will scan major subfolder of sysfs and emulate a “device add” uevent. It will handle this event itself and create device node under “/dev/” accordingly. This action is called: “coldboot”. The “ueventd.*.rc” under the “/” is used to control the access/owner of the device node.
- All commands in the “.rc” files are pre-defined in init. It’s NOT shell command.
- Basic sequence for init commands in .rc:
“early-init” -> wait_for_coldboot_done() -> “init” -> “early-fs” -> “fs” -> “post-fs” -> “post-fs-data” -> “early-boot” -> “boot”

Zygote

- Zygote is (expected) the parent for all java process.
- It's started through calling "app_process" which is an executable and entry for java class. "app_process" behaves like the "java" command on PC.

In init.rc:

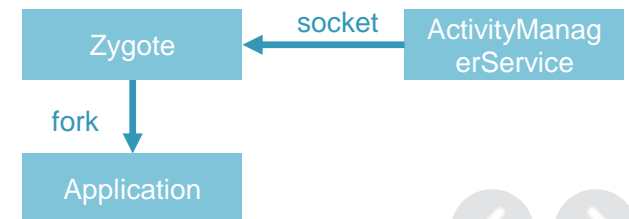
```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
    class core
    ...
```

In main() of "" frameworks/base/cmds/app_process/app_main.cpp :

```
AppRuntime runtime;

if(zygote) {
    runtime.start("com.android.internal.os.ZygoteInit", startSystemServer ? "start-system-server" : "");
} else if (className) {
    runtime.mClassName = className;
    runtime.start("com.android.internal.os.RuntimeInit", application ? "application" : "tool");
} else { }
```

- Major tasks for Zygote:
 - Do class and resource preload for java application.
 - Start SystemServer.
 - Act as the agent for starting java process.



SystemServer

- Native service will be started first and then java services.
- DexOpt will be done when first boot.
- Except the daemons started in init, all other system services are started in SystemServer. It's a very long list in:

frameworks/base/services/java/com/android/server/SystemServer.java

- Package Manager will scan all packages for meta data and permission related info.
- All services are firstly “added” then be set to “ready” one-by-one.
- Launcher started by ActivityManager when it's “ready” function been called:

// Add all services and made several basic service ready before this

```
ActivityManagerService.self().systemReady(new Runnable() {  
    public void run() {  
        try {  
            if (mountServiceF != null) mountServiceF.systemReady();  
        } catch (Throwable e) {  
            reportWtf("making Mount Service ready", e);  
        }  
    }  
})
```

// Make all other services ready.

```
I/SystemServer( 2361): Activity Manager  
I/SystemServer( 2361): Display Manager  
I/SystemServer( 2361): Telephony Registry  
I/SystemServer( 2361): Scheduling Policy  
I/SystemServer( 2361): Package Manager  
I/SystemServer( 2361): Entropy Mixer  
I/SystemServer( 2361): User Service  
I/SystemServer( 2361): Account Manager  
I/SystemServer( 2361): Content Manager  
I/SystemServer( 2361): System Content Providers  
I/SystemServer( 2361): Lights Service  
I/SystemServer( 2361): Battery Service  
I/SystemServer( 2361): Alarm Manager  
I/SystemServer( 2361): Init Watchdog  
I/SystemServer( 2361): Input Manager  
I/SystemServer( 2361): Window Manager  
I/SystemServer( 2361): Input Method Service  
I/SystemServer( 2361): Accessibility Manager  
I/SystemServer( 2361): Mount Service  
I/SystemServer( 2361): Wallpaper Service  
I/SystemServer( 2361): Status Bar  
I/SystemServer( 2361): Making services ready  
I/SystemServer( 2361): Bluetooth Manager Service  
I/SystemServer( 2361): LockSettingsService  
I/SystemServer( 2361): Device Policy  
I/SystemServer( 2361): Clipboard Service  
I/SystemServer( 2361): NetworkManagement Service  
I/SystemServer( 2361): Text Service Manager Service  
I/SystemServer( 2361): NetworkStats Service  
I/SystemServer( 2361): NetworkPolicy Service  
I/SystemServer( 2361): Wi-Fi P2pService  
I/SystemServer( 2361): Wi-Fi Service  
I/SystemServer( 2361): Connectivity Service  
I/SystemServer( 2361): Network Service Discovery Service  
I/SystemServer( 2361): Notification Manager  
I/SystemServer( 2361): Device Storage Monitor  
I/SystemServer( 2361): Location Manager  
I/SystemServer( 2361): Country Detector  
I/SystemServer( 2361): Search Service  
I/SystemServer( 2361): DropBox Service  
I/SystemServer( 2361): Audio Service  
I/SystemServer( 2361): Wired Accessory Manager  
I/SystemServer( 2361): USB Service  
I/SystemServer( 2361): Serial Service  
I/SystemServer( 2361): Twilight Service  
I/SystemServer( 2361): UI Mode Manager Service  
I/SystemServer( 2361): AppWidget Service  
I/SystemServer( 2361): DiskStats Service  
I/SystemServer( 2361): SamplingProfiler Service  
I/SystemServer( 2361): NetworkTimeUpdateService  
I/SystemServer( 2361): CommonTimeManagementService  
I/SystemServer( 2361): CertBlacklist  
I/SystemServer( 2361): IdleMaintenanceService  
I/SystemServer( 2361): Media Router Service
```



Put it together

- Start of the Android boot is “init”, and, the end?
 - Android will broadcast an intent when it thinks it has finished with boot: “Intent.ACTION_BOOT_COMPLETED”. Two properties will also be set to “1”: “sys.boot_completed” & “dev.bootcomplete”.
 - When? When anyone of the activity firstly calls into “idle”. This is done in:
frameworks/base/services/java/com/android/server/am/ActivityManagerService.java
 - The home screen is actually shown before the “boot complete” broadcast.
- Launcher will be the first user-aware application been started by ActivityManager. What is launcher?

In “AndroidManifest.xml” of the APP:

category android:name="android.intent.category.HOME

So our target for faster Android boot is clear:

Launch the HOME application as early as possible!



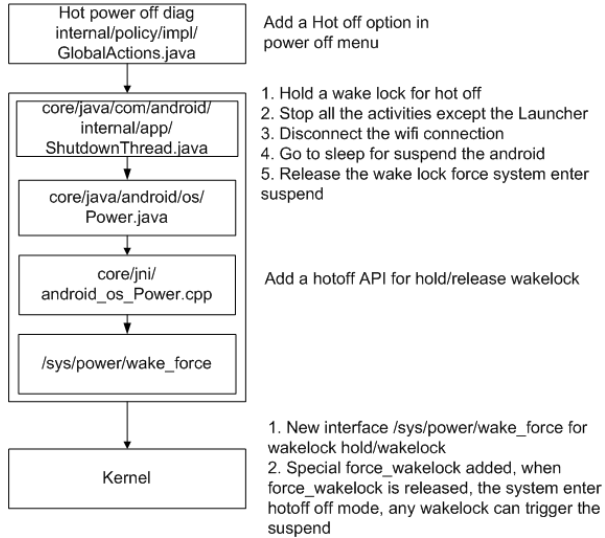
Android boot optimization

Popular directions for speed up booting

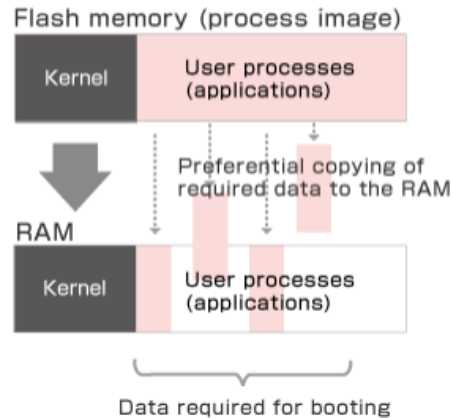
- Use suspend instead of real power off when user “power off” the unit. So-called “hot boot” or “standby”.
 - Advantage: simple to implement and very fast “boot”.
 - Disadvantage: power consumption, abnormal state recovering.
- Use hibernation to flush all/part of the DDR/cache/registers/IRAM into non-volatile memory such as flash.
 - Advantage: Normally only kernel level code touched. Fast enough.
 - Disadvantage: Difficult to handle xPU (GPU/VPU/...) and peripherals. Subject to falling into some corner cases in a complex hardware environment. Time for power on/off will increase when memory size increase or after long time using.
- Make “checkpoint” for the module that consume most boot time.
 - Advantage: Minor changes in kernel code. Easy to port to a new platform.
 - Disadvantage: May run into dependency issue and other unexpected application compatibility issue. The time improvement is not so significant as well.
- Customize the OS/Application to make it boot faster:
 1. Optimize boot related code to make it run faster.
 2. Adjust the boot sequence to make “user aware” related service start as early as possible.
 3. Custom the system components and remove un-necessary ones.
 - Advantage: Boot speed comparable with hibernation. A real boot.
 - Disadvantage: Require huge customization on OS/App level, may have impact to application compatibility.

Some solutions...

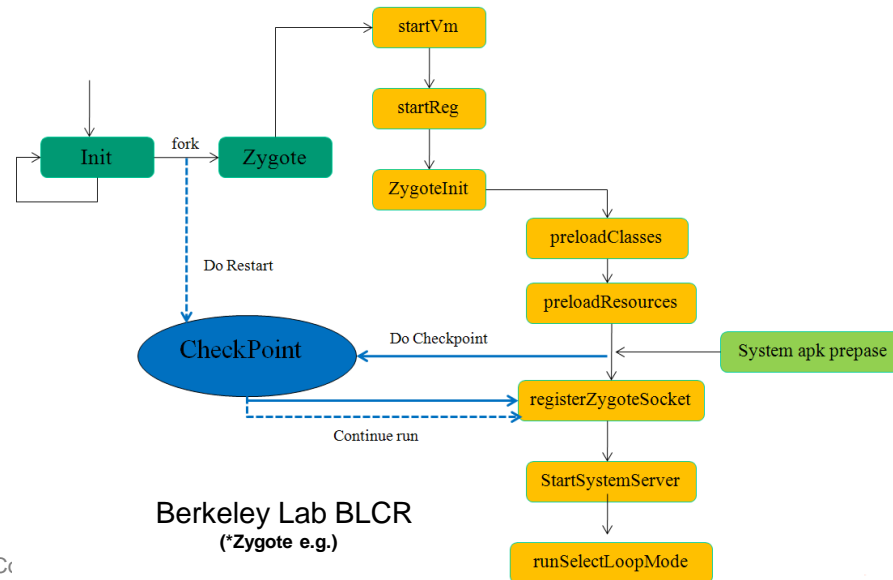
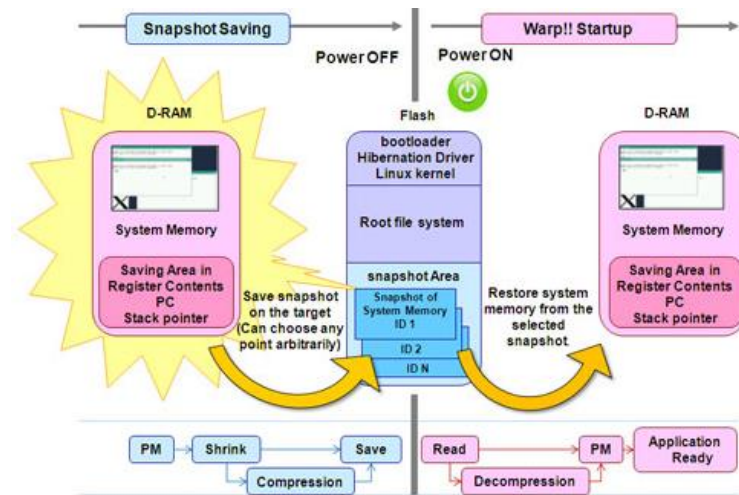
HTC Hot Boot



Ubiquitous Quickboot



Lineo Warp



Berkeley Lab BLCR
(*Zygote e.g.)

Measurement and analyzing tool

- “logcat -v time”

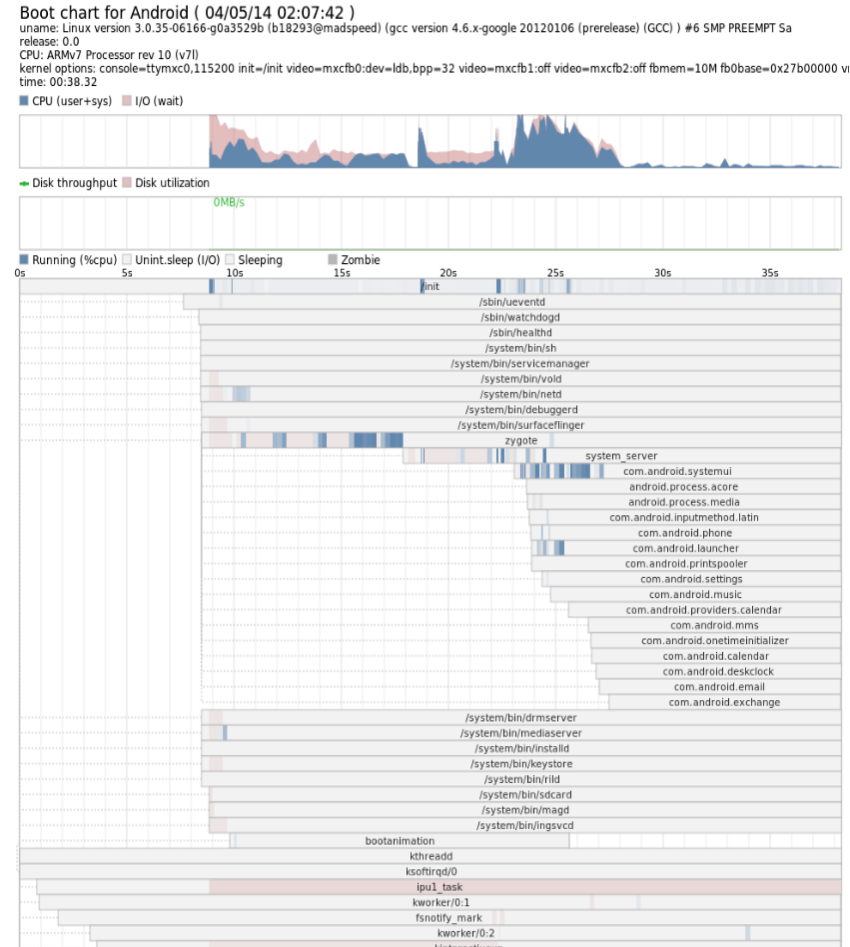
Print the timestamp. There's also a patch for logcat which can pull the kernel log into logcat, with converted timestamp.

- bootchart

“bootchart” is originally design for linux. Can also be used to measure and analyze the Android. A sample picture shown below.

A simple guide attached below.

- Other performance related tools on Linux/Android such as: strace, top, perf, dumphsys, procrank, systrace...





A try on IMX6

A try on IMX6 – before start

- What is the current status on “Android KK4.4.3 GA” for SabreSD (with eMMC)?

About 21 s

- Preset condition and considerations:
 - Target to auto infotainment.
 - Will NOT try suspend or hibernation or checkpoint.
 - Custom Android, but, keep all core/key components to ensure MAX application compatibility.
- “Tooth paste effect”.

uboot/kernel

All ideas for optimizing uboot/kernel in previous sections applied:

- Uboot:
 - Size: 380 KB -> 120 KB
 - Reduced print and use built-in ENV
 - Remove boot logo.
- Kernel:
 - Size: 6.5 MB -> 3.5 MB (compressed version)
 - Use un-compressed kernel
 - Reduced “printk” to remove the serial initcall latency
 - CMA disabled due to the alloc latency
 - Build some drivers as module and do “insmod” in “.rc”
- Result: uboot+kernel boot time: 6.18s -> 1.48s.

SELinux

- SELinux was introduced to Android since 4.3, prior to Android 4.3, application sandboxes were defined by the creation of a unique Linux UID for each application at time of installation. Security-Enhanced Linux (SELinux) is used to further enforce the security with a new mechanism: MAC (Mandatory Access Control).
- SELinux operates on the ethos of default denial. Anything that is not explicitly allowed is denied. SELinux can operate in one of two global modes: permissive mode, in which permission denials are logged but not enforced, and enforcing mode, in which denials are both logged and enforced.
- SELinux lives in Linux kernel and initialized in “init”. It will cost about 3s! Can be disabled if the traditional DAC (Discretionary Access Control) is enough.

```
[ 5.024244] Freeing init memory: 252K
[ 5.030408] init: START ...
[ 5.033308] init: loading selinux policy
...
[ 5.643312] type=1404 audit(2.120:3): enforcing=1 old_enforcing=0 auid=4294967295 ses=4294967295
[ 8.636193] init: property init
```

```
// Add the following segment into the kernel CMDLINE:
androidboot.selinux=disabled
```

Materials from: <https://source.android.com/devices/tech/security/selinux/index.html>

Delay loading daemons

- As can be found from “init” working flow, all daemons starting are queued and then executed in a loop. This is causing a “flood” of new daemons in a very short time.
- The time between “starting Zygote” to “Zygote up” is around 1500 ~ 2000ms.

```
00:03:17.954      V/kernel ( 2619): <5>[ 6.250445] init: starting 'installd'
00:03:17.958      V/kernel ( 2619): <5>[ 6.254792] init: starting 'servicemanager'
00:03:17.963      V/kernel ( 2619): <5>[ 6.259706] init: starting 'surfaceflinger'
00:03:17.968      V/kernel ( 2619): <5>[ 6.264555] init: starting 'healthd'
00:03:17.972      V/kernel ( 2619): <5>[ 6.268770] init: starting 'zygote'
00:03:17.983      V/kernel ( 2619): <5>[ 6.279124] init: starting 'vold'
00:03:17.987      V/kernel ( 2619): <5>[ 6.283156] init: starting 'media'
00:03:18.081      V/kernel ( 2619): <5>[ 6.377111] init: starting 'XXX'...
... // Many other daemons...
00:03:18.180      I/SurfaceFlinger( 2343): SurfaceFlinger is starting
... // SF
00:03:18.200      I/Netd ( 2353): Netd 1.0 starting
... // SF
00:03:19.182      V/kernel ( 2619): <3>[ 7.477984] ERROR: v4l2 capture: slave not found!
00:03:19.450      V/NatController( 2353): runCmd(/system/bin/iptables -F natctrl_FORWARD
res=0
... // NETD
00:03:19.500      D/AndroidRuntime( 2345):
00:03:19.500      D/AndroidRuntime( 2345): >>>>>> AndroidRuntime START
com.android.internal.os.ZygoteInit <<<<<<
... // NETD
00:03:21.250      I/SystemServer( 2694): Entered the Android system server!

// This log is captured after disabling some daemons. Actually delay between the read lines will be
longer.
```

Delay loading daemons – How?

- Daemons (also called “service”) are classified into “class”. Most daemons are started by calling “class_start”.
- Our target is: make sure Zygote start faster and all other daemons that has no impact to boot will be started slower/later.
- Two possible ways to go:
 - Give Zygote a higher priority (current “0”).
 - Delay loading other daemons. We’re going this way.

```
// Original init.rc
on boot
    // start mtpd
    class_start core
    class_start main

service mtpd /system/bin/mtpd
class main
...
```

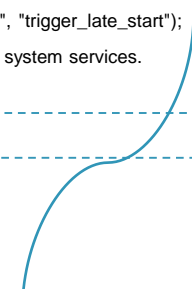
```
// frameworks/base/services/java/com/android/server/SystemServer.java
public void initAndLoop() {
    // After some GUI related critical system services have been started.
    SystemProperties.set("sys.delayload", "trigger_late_start");
    // Start other non-critical background system services.
}
```

```
// New init.rc
on post-fs
    class_start core

on property:sys.delayload=trigger_late_start
    class_start late_start

service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server
class core
...

service mtpd /system/bin/mtpd
class late_start
...
```



Class/Resource pre-loading in Zygote

- From logcat, the preloading for classes and resources costs about 7s ! And further more, it's blocking call in Zygote, before forking the SystemServer.
- Can be totally skipped, with possible penalty on launching application later. From test so far, affordable.

```
00:03:02.330 I/Zygote ( 2346): Preloading classes...  
...  
00:03:07.620 I/Zygote ( 2346): ...preloaded 2777 classes in 5291ms.  
00:03:08.020 I/Zygote ( 2346): Preloading resources...  
...  
00:03:09.230 I/Zygote ( 2346): ...preloaded 274 resources in 1206ms.  
...  
00:03:09.260 I/Zygote ( 2346): ...preloaded 31 resources in 29ms.
```

```
// frameworks/base/core/java/com/android/internal/os/ZygoteInit.java  
public static void main(String argv[]) {  
    try {  
        registerZygoteSocket();  
        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,  
            SystemClock.uptimeMillis());  
        preload();  
        EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,  
            SystemClock.uptimeMillis());  
    } }  
}
```

Sensor initialization

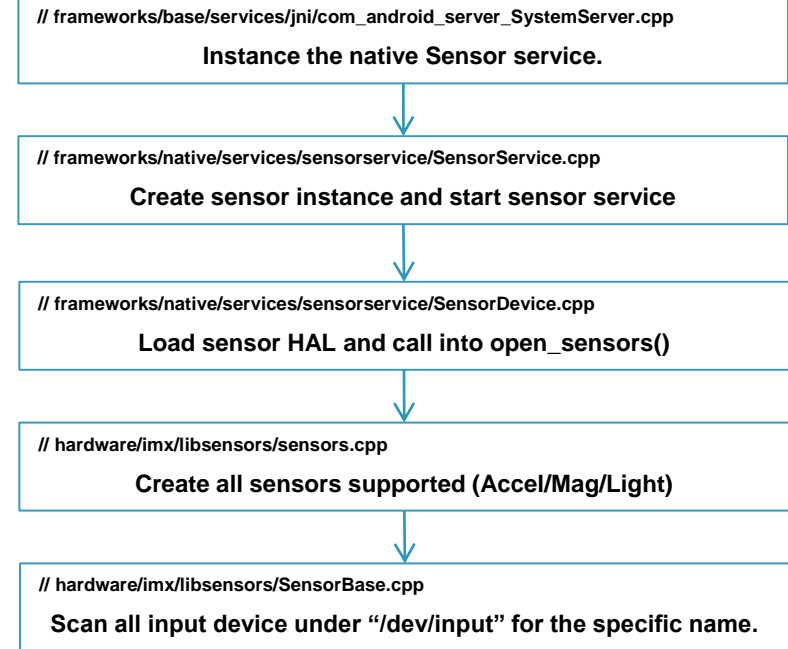
- Sensor services is started before any JAVA services.
- It will finally call into sensor HAL for current platform.
- It will scan all input device under “/dev/input” for the sensor name. A directory walk will be done, each device will be called with IOCTL to get the input device name. This will cost about 800 ms in total.
- A new mechanism implemented:
parse “/proc/bus/input/devices” for sensor device and open corresponding device directly. No IOCTL and no dummy try. Reduced to about 30 ms.

```
// frameworks/base/services/java/com/android/server/SystemServer.java
public static void main(String[] args) {
    Slog.i(TAG, "Entered the Android system server!");

    // Initialize native services. Sensor service starts here...
    nativeInit();

    ServerThread thr = new ServerThread();
    thr.initAndLoop(); // Call into main loop for other JAVA services.
}
```

```
00:03:21.150 I/dalvikvm( 2345): System server process 2694 has been created
00:03:21.160 I/Zygote ( 2345): Accepting command socket connections
00:03:21.250 I/SystemServer( 2694): Entered the Android system server!
00:03:21.250 D/SensorService( 2694): nuSensorService starting...
00:03:22.060 D/ ( 2694): AccelSensor enable 0 , handle 0 ,mEnabled 0
00:03:22.080 D/ ( 2694): AccelSensor enable 0 , handle 1 ,mEnabled 0
00:03:22.080 D/ ( 2694): MagSensor mEnabled 0, OrientationSensor mEnabled 0
00:03:22.090 I/SystemServer( 2694): Waiting for install to be ready.
```



Delay loading service

- From the workflow of SystemServer and the log, the SystemServer will firstly “add/create” about 60+ services before calling into “system ready” interface of ActivityManager. The launcher will then be started.
- The time between “start ActivityManager” and “start HOME application” is about 6800 ms!

```
02:53:07.380 I/SystemServer( 2713): Entered the Android system server!
02:53:07.980 I/SystemServer( 2713): Waiting for install2 to be ready.
02:53:07.980 I/SystemServer( 2713): Power Manager
02:53:07.980 I/SystemServer( 2713): Activity Manager
...
02:53:07.990 I/ActivityManager( 2713): Memory class: 64
...
02:53:08.080 I/SystemServer( 2713): Display Manager
02:53:08.080 I/SystemServer( 2713): Telephony Registry
02:53:08.090 I/SystemServer( 2713): Scheduling Policy
02:53:12.220 I/SystemServer( 2713): Entropy Mixer
02:53:12.240 I/SystemServer( 2713): User Service
...
// 55 more services ...
...
02:53:14.230 I/ActivityManager( 2713): System now ready
02:53:14.780 I/ActivityManager( 2713): START u0 {act=android.intent.action.MAIN
cat=[android.intent.category.HOME] flg=0x10000000
cmp=com.android.launcher/com.android.launcher2.Launcher} from pid 0
```

Delay loading service – How?

- Delay of service starting is a bit straight forward, but, tricky since the services/components have dependency/coupling between each other. No clear “cut” between “core service” and “non-core service”.
- Some “protection” might need to be added to SystemUI/DisplayManager and Launcher since they may use some services that are not really needed for the time they start.

```
// frameworks/base/services/java/com/android/server/SystemServer.java
```

```
public void initAndLoop() {  
    // Core service creation here...  
    // Making core services ready here...  
    ActivityManagerService.self().systemReady ();  
    // non-Core service creation here...  
    // Making non-core services ready here...  
}
```

```
00:28:02.170    I/SystemServer( 2362): Entered the Android system server!  
00:28:02.200    I/SystemServer( 2362): Waiting for installd to be ready.  
00:28:02.200    I/SystemServer( 2362): Power Manager  
...  
00:28:02.200    I/SystemServer( 2362): Activity Manager  
00:28:02.270    I/ActivityManager( 2362): Memory class: 64  
...  
00:28:02.560    I/SystemServer( 2362): Display Manager  
00:28:02.560    I/SystemServer( 2362): Telephony Registry  
...  
// 16 more services here  
...  
00:28:03.840    I/SystemServer( 2362): Wallpaper Service  
00:28:03.850    I/SystemServer( 2362): Status Bar  
...  
00:28:03.850    E/SystemServer( 2362): BOOTOPT: Delay loading some services ...  
00:28:03.960    I/ActivityManager( 2362): System now ready  
00:28:03.990    I/ActivityManager( 2362): START u0 {act=android.intent.action.MAIN  
cat=[android.intent.category.HOME] flg=0x10000000 cmp=com.example.android.home/.Home}  
from pid 0  
...  
// 5 more services here  
...  
00:28:06.396    V/kernel ( 2890): <4>[ 11.610716] BOOTOPT: Android boot complete  
...  
00:28:09.010    I/SystemServer( 2362): Text Service Manager Service  
00:28:09.010    I/SystemServer( 2362): NetworkStats Service  
...  
// 26 more services here, continue booting after GUI shown...  
...  
00:28:14.470    I/SystemServer( 2362): Media Router Service  
00:28:15.160    I/SystemServer( 2362): Enabled StrictMode for system server main thread.
```

Android DexOpt

- What is DEX? What is DexOpt?

- DEX:

- Each “.jar”/“.apk” (actually both ZIP file) has a “classes.dex” inside. It holds all JAVA classes (bytecode) for this package.

- DexOpt:

- DexOpt is an executable included in Android image. From an discussion on “stackoverflow”:

- “dexopt does some optimizations on the dex file. It does things like replacing a virtual invoke instruction with an optimized version that includes the vtable index (to the boot class) of the method being called, so that it doesn't have to perform a method lookup during execution. The result of dexopt is an odex (optimized dex) file. This is very similar to the original dex file, except that it uses some optimized opcodes, like the optimized invoke virtual instruction”*

- DexOpt can be done

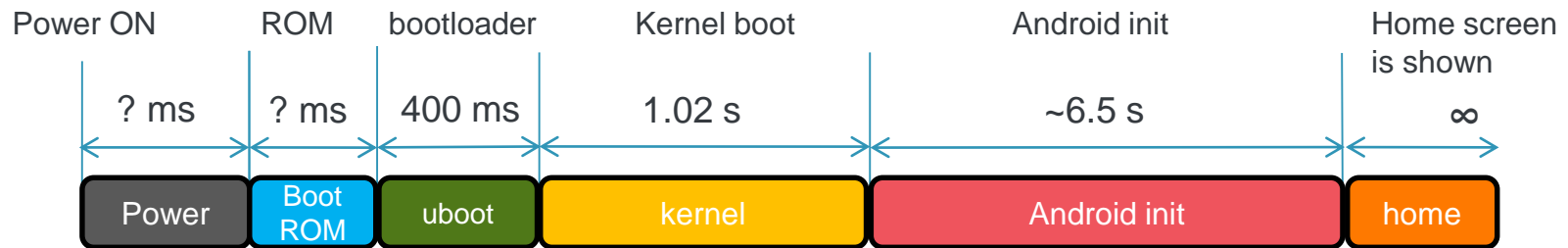
- During building/preparing the “system.img”. A “.odex” file and corresponding “.apk” (stripped one) will be included in “system.img” after this.
 - Android will do it when first boot if it's not done. If we care about the first boot time, we can avoid this by using #1.

Other successful/un-successful try...

- There's a “safe mode” detection in SystemServer which happen before calling any “system ready”. It has a timeout of “INPUT_DEVICES_READY_FOR_SAFE_MODE_DETECTION_TIMEOUT_MILLIS” waiting for input device ready. Must be handled if delay loading is enabled.
(Minor help)
- Use smaller system/data partition to save mount time. A un-gracefully power shut will increase the mount time as well. *(Minor help)*
- Some non-core services can be disabled if not used on auto, such as:
VibratorService/ConsumerIrService/UpdateLockService/DockObserver/BackupManagerService/DreamService/AssetAtlasService/PrintManagerService
(Minor help)
- Remove some un-necessary packages from system.img. *(No direct help, but can help to reduce system partition size).*

A try on IMX6 – what has been achieved

- Current status with optimized “Android KK4.4.3 GA” on SabreSD (with eMMC)?



- Limitations:
 - May impact CTS/GTS after this.



www.Freescale.com