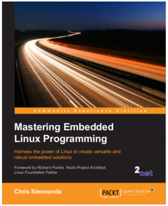
Booting Android

Bootloaders, fastboot and boot images

2net

License

About Chris Simmonds



- Consultant and trainer
- Author of *Mastering Embedded Linux Programming*
- Working with embedded Linux since 1999
- Android since 2009
- Speaker at many conferences and workshops

"Looking after the Inner Penguin" blog at http://2net.co.uk/

https://uk.linkedin.com/in/chrisdsimmonds/

https://google.com/+chrissimmonds

Overview

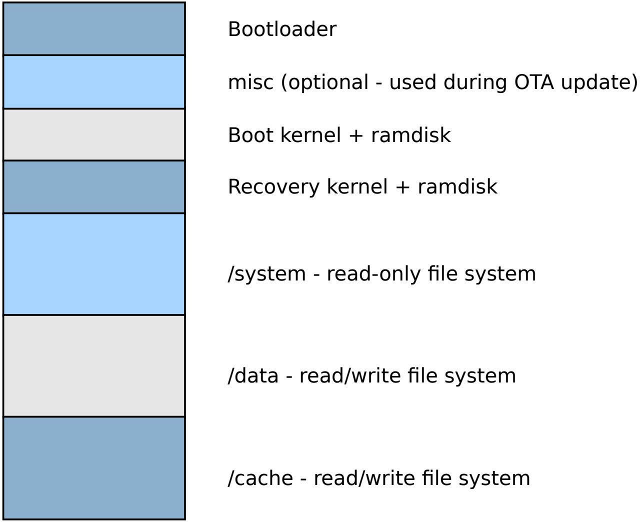
- Android system images: boot, recovery, system, userdata and cache
- Android "boot blobs"
- Bootloaders for Android
- Fastboot
- Flash memory and flash filesystems

Image files

- A typical build for an Android device produces five image files in `out/target/product/<product>`

Image	Description
boot.img	Kernel + ramdisk used for normal boot
recovery.img	Kernel + ramdisk used to boot into recovery mode
system.img	File system image for /system
userdata.img	File system image for /data
cache.img	File system image for /cache

Typical flash memory layout



The bootloader

- All systems need a bootloader
- Responsible for:
  - Early hardware initialisation
  - Load and boot kernel and initial ram filesystem
  - System maintenance, including loading and flashing new kernel and system images
- Example: U-Boot
  - Open source
  - Used in many dev boards (BeagleBone, Raspberry Pi) and in many shipping products
  - `http://www.denx.de/wiki/U-Boot/WebHome`

Booting Android

- It is possible to boot Android using a normal bootloader such as U-Boot
- However, most devices include Android-specific features:
  - Support normal and recovery boot modes
  - Ability to load kernel + ramdisk blobs (boot.img and recovery.img)
  - The fastboot protocol
- Example: LK (Little Kernel)
  - `git://codeaurora.org/kernel/lk.git`
  - Supports many Qualcomm-based devices as well as rudimentary support for BeagleBoard and PC-x86

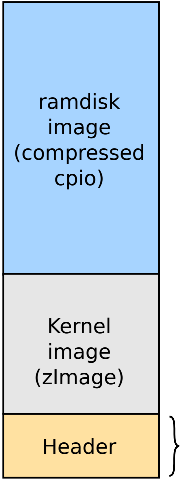
The Android bootloader

- Pre JB 4.2, AOSP had source for a simple bootloader in `bootable/bootloader/legacy`
  - Used in early handsets (Android Dev Phone, HTC Dream)
  - Not updated since the Eclair release
  - Some of this code may have found its way into proprietary bootloaders

Android boot and recovery images

- The files `boot.img` and `recovery.img` are created by the tool `mkbootimg` (the code is in `system/core/mkbootimg`)
- They contain a compressed kernel, the kernel command line and, optionally, a ramdisk in the normal Linux compressed cpio format
- Most Android bootloaders can read and load these images into memory
- The format is defined in `bootimg.h`

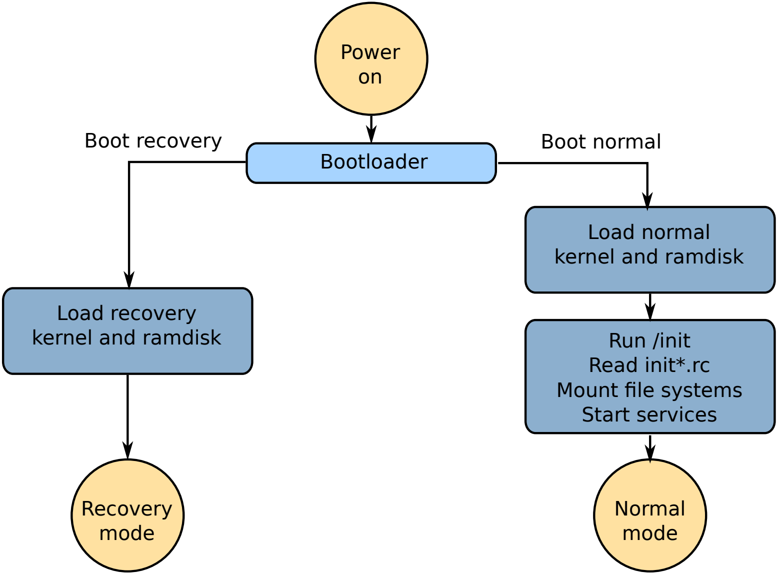
Boot and recovery image format



From system/core/mkbootimg/bootimg.h

```
struct boot_img_hdr {
    unsigned char magic[8];    // "ANDROID!"
    unsigned kernel_size;
    unsigned kernel_addr;
    unsigned ramdisk_size;
    unsigned ramdisk_addr;
    unsigned second_size;       // 2nd image: not used
    unsigned second_addr;
    unsigned tags_addr;
    unsigned page_size;         // typically 2048
    unsigned unused[2];
    unsigned char name[16];     // product name
    unsigned char cmdline[512]; // kernel cmdline
    unsigned id[8];             // timestamp/checksum/etc
    unsigned char extra_cmdline[1024];
};
```

Boot sequence



Reverse-engineering a boot image

- Sometimes it is useful to extract the files from a boot or recovery image
- There are numerous tools to do so, for example boot-extract
  https://github.com/csimmonds/boot-extract

```
$ boot-extract recovery.img
Boot header
  flash page size 2048
  kernel size 0x432358
  kernel load addr 0x10008000
  ramdisk size 0x173740
  ramdisk load addr 0x11000000
  name
  cmdline
zImage extracted
ramdisk offset 4403200 (0x433000)
ramdisk.cpio.gz extracted
$ ls
ramdisk.cpio.gz  recovery.img  zImage
```


Extracting files from a ramdisk

- The ramdisk is just a compressed cpio archive
- Extract the files like so:

```
$ zcat ramdisk.cpio.gz | cpio -i
5665 blocks
$ ls
charger         fstab.manta             property_contexts
...
```

Creating a new ramdisk

- Do the following

```
$ cd some-directory
$ find . | cpio -H newc --owner root:root -ov > ~/ramdisk.cpio
$ cd ~
$ gzip ramdisk.cpio
```

- The end result will be `ramdisk.cpio.gz`

Creating a new boot image

- You can create a boot or recovery image using the mkbootimg command
- For example:

```
$  mkbootimg --kernel zImage --ramdisk ramdisk.cpio.gz \
 --base 0x10000000 --pagesize 2048 -o recovery-new.img
```

- `--base` is used by mkbootimg to calculate the kernel and ramdisk load addresses as follows:
  - `kernel_addr = base + 0x00008000`
  - `ramdisk_addr = base + 0x01000000`

Fastboot

- Fastboot is a USB protocol and a command language for various maintenance and development tasks
- Fastboot protocol v0.4 is defined in:
    - `bootable/bootloader/legacy/fastboot_protocol.txt` (up to JB 4.1)
    - `system/core/fastboot/fastboot_protocol.txt` (JB 4.3 and later)

NOTE: fastboot is not about the speed of booting; it is about making the development process simpler (and faster)

Booting into the bootloader

- On a typical Android device you can boot into the bootloader by:
  - powering on while pressing various buttons (Google for details)
  - from a running device, typing:

```
$ adb reboot-bootloader
```

- Once the device has booted into the bootloader you can use the *fastboot* command on the development machine to communicate with it

fastboot commands (1/3)

Basic commands

Command	Description
devices	List devices attached that will accept fast-boot commands
getvar	Get a variable
continue	Continue boot process as normal
reboot	Reboot device
reboot-bootloader	Reboot back into bootloader

fastboot commands (2/3)

Flashing commands

Command	Description
erase `<partition>`	Erase `<partition>`
flash `<partition>`	Erase and program `<partition>` with `<partition>.img` of *current product*
flash `<partition> <filename>`	Erase and program `<partition>` with `<filename>`
flashall	Erase and program `boot.img`, `recovery.img` and `system.img` of *current product* and then reboot

Where
<partition> is one of `boot, recovery, system, userdata, cache`
current product is `$ANDROID_PRODUCT_OUT`

Note: the location and size of partitions is hard-coded in the bootloader

fastboot commands (3/3)

Special commands

Command	Description
oem	Device-specific operations
boot <kernel> <ramdisk>	Load and boot kernel and ramdisk

Example:

```
$ fastboot -c "kernel command line" boot zImage ramdisk.cpio.gz
```


fastboot variables

The **getvar** command should return values for at least these variables:

Variable	Meaning
version	Version of the protocol: 0.4 is the one documented
version-bootloader	Version string of the Bootloader
version-baseband	Version string of the Baseband Software
product	Name of the product
serialno	Product serial number
secure	If "yes" the bootloader requires signed images

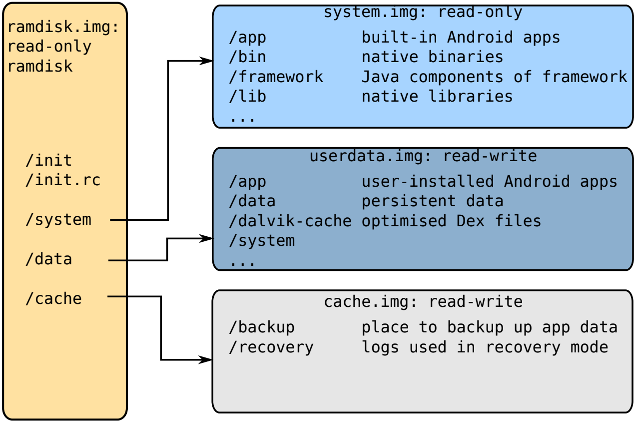
Unlocking the bootloader

- Most devices ship with the bootloader locked
  - `fastboot getvar secure` returns *true*
- Unlocking - where it is allowed - is device specific
- For example, on recent Nexus devices you use a fastboot oem command

```
$ fastboot oem unlock
```

- Answer *yes* to the on-screen prompt
- For security reasons, this wipes the data and cache partitions

What goes where?



Flash memory devices

- In almost all cases data is stored in flash memory
- There are two main types
  - **Raw NAND flash**, where the chips are accessed directly by Linux
  - **Managed flash**, which contain an on-chip controller
- Managed flash is the most common
- Examples:
  - MMC, SD and MicroSD cards: removeable storage
  - eMMC (embedded MMC): same electrical interface as MMC, but packaged as a chip
  - UFS (Universal Flash Storage): similar to eMMC, but faster and with a SCSI command set

Raw NAND flash

- NAND flash chips are accessed via the Linux MTD (Memory Technology Device) drivers
- Partitions are named /dev/block/mtdblockN where N is the partition number
- /proc/mtd lists the partitions and sizes

```
# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 05660000 00020000 "system"
mtd1: 04000000 00020000 "userdata"
mtd2: 04000000 00020000 "cache"
```


File systems for raw NAND flash

- Flash translation layer implemented in the filesystem
- NAND flash devices require special filesystem support, such as:
- jffs2 (Journalling Flash File System 2)
  - Note: incompatible with the Android run-time (no writeable mmaped files)!
- yaffs2 (Yet Another Flash File System 2)
- ubifs (Unsorted Block Image File System)
- Most Android devices with NAND flash use yaffs2

SD and eMMC

- Flash translation layer implemented in the chip
- The controller chip splits flash memory into 512-byte sectors just like hard drives
- Accessed via the Linux mmcblock driver
- Partition device nodes have names of the form *mmcblk[chip number]p[partition number]*
- For example:

```
/dev/block/mmcblk0p3 /system
/dev/block/mmcblk0p8 /data
/dev/block/mmcblk0p4 /cache
```


File systems for eMMC

- eMMC devices "look" like hard drives
- Use the same filesystem types
- The preferred type in most Android devices is *ext4*
- Alternative: F2FS (Flash Friendly File System)
  - Develpoed by Samsung, and deployed on some of their devices
  - Faster file writes than ext4

SD cards and other removable media

- This includes MMC, SD, microSD and USB flash drives
- For compatibility with other operating systems they come pre-formatted with FAT32
- Use the Linux vfat driver

Delving deeper

- This is an excerpt from my **Android Porting** class
- If you would like to find out more secrets of Android, visit `http://www.2net.co.uk/training.html` and book a course

