Reverse Engineering Android's Aboot

Jonathan Levin, http://NewAndroidBook.com

Android's boot loader is a fairly uncharted area of the landscape. What little is known is largely due to partial open source, and for some devices - notably Amazon's and Samsung's - even that isn't available. Most device modders generally leave it be, and (given an unlocked bootloader) start off with the boot.img (kernel + ramdisk) and follow on to various modifications in /system.

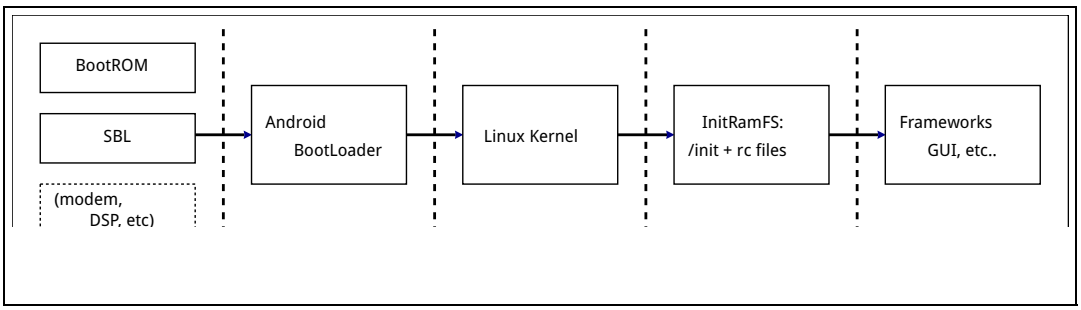
The Confectioner's Cookbook devotes an entire chapter to the boot process, wherein I touch (among other things) on the boot loader format and structure. In an effort to keep things simple, however, I stop shy of reverse engineering and disassembly - largely because the first part of the book is aimed at power users, and less at developers or hackers. There's obviously great benefit in the more advanced techniques, however, so the discussion is deferred to the companion article on the book's web site. This is said article.

For those of you who haven't read the book, I recap the key points from the Boot chapter here. I then continue to discuss how to reverse engineer the binary, which comes in especially handy when handling the proprietary loaders. What follows is a discussion of ARM-architecture specifics, and then additional observations I've found thus far.

Recap: Dramatis Personae of the Android Boot Process

Android's boot process starts with the firmware*, which loads from a ROM. The exact details of the firmware boot vary between devices and specific architectures (e.g. Qualcomm vs. NVidia). Nonetheless, they can be generalized in the following figure:

Figure 1: The generalized Android Boot Process (from ACC, Chap. 3)



The BootROM loads several other components, each from a dedicated partition. Chief amongst those is a secondary boot loader (SBL), which is responsible for overcoming the tight constraints of ROM - limited space and an inability to upgrade. When you "flash the bootloader", you flash those partitions (as discussed in the book). For now, however, our focus is Android's own boot loader, which often resides in a partition called "aboot".

aboot is a native ARM binary, which is wrapped by a very thin header, which is usually 40 bytes in length. This header provides metadata which is used by the SBL in the process of validating and loading aboot. Note, "validating and loading" - in this order - because the SBL will very likely reject an aboot image which is not properly signed by the vendor. This works to extend a chain of trust, which starts at the ROM, all the way to aboot, which further extends it to the boot.img (unless the boot loader is "unlocked"). In other words, the

ROM has a built-in key (which, by its nature, cannot be modified), used to validate the precursors of the SBL. Those, in turn, validate the SBL, which validates aboot, thus securing the boot sequence and preventing any modification by malware (or unauthorized rooting attempts). aboot is the first component which may opt to break the chain - and this is what is meant by "boot loader unlocking": The unlocking simply disables the signature check on the next part of the boot sequence, which is loading the kernel and RAM disk from the boot partition (flashed from boot.img).

Not all boot loaders can be unlocked, however, as that is left to the vendor's discretion. For those which do, it is usually a straightforward matter - the device is placed into bootloader mode (adb reboot bootloader) and then a fastboot oem unlock is all it takes. Amazon's Fire-devices (FireTV, FirePhone, and the Kindle Fire) do not allow this, as well as some versions of Samsung's boot loader. Samsung appears to be more particular, in allowing some international versions of their devices to be unlockable, and other not. As discussed in Chapter 21, boot loader locking is as essential part of Android security, but by itself is insufficient to prevent rooting in most cases.

Thus, the SBL loads aboot, and inspects its header to find the "directions" for loading. You can obtain aboot from a factory image (by using imgtool to break apart the bootloader.img) or directly from the device's aboot partition (assuming a rooted device, of course). You can find the aboot partition number by consulting /proc/emmc (where available), or (in JB and later) /dev/block/platform/*platformname*/by-name/aboot. Using busybox cp or dd, copy the partition to a file (say, "aboot"). Picking up on the experiment in the book, we have:

Output 1: aboot from a Nexus 5 KK (kot49h) factory image

```
morpheus@Forge (~/...-kot49h/)% od -A d -t x4 aboot |head -5
              Magic         Version          NULL         ImgBase
0000000    00000005    00000003    00000000    0f900000
            ImgSize       CodeSize   ImgBase+CodeSize     SigSize
0000016    0003fe2c    0003e52c    0f93e52c    00000100
      ImgBase+CodeSize+SigSize   Certs
0000032    0f93e62c    00001800    ea000006    ea00351c
0000048    ea003522    ea003528    ea00352e    ea003534
0000064    ea003534    ea00354b    ee110f10    e3c00a0b
```

You can toggle between the devices shown here, but note the format of aboot is still very much the same. The Magic (0x0000005) misleads the file(1) command into thinking this is an Hitachi SH COFF object. The SBL, however, is not fooled, and validates the signature (usually 256 bytes, i.e. a 2048-bit PKI, immediately after HeaderSize + CodeSize) with the certificate chain that is loaded from the offset (HeaderSize+ImgSize + CodeSize + SigSize).If the signature is valid, the SBL proceeds to load the code immediately following the header (CodeSize bytes) by mapping into ImgBase.

The first byte of the aboot binary image is identifiable by the "eaXXXXXX" value. The book alludes to this being an ARM B(ranch) instruction, but stops short of explaining its significance - which is explained next.

ARM Processor Boot, in a Nutshell

ARM processors use an Exception Vector during their lifecycle. This vector is a set of 6 or seven addresses, set by the operating system, and instructing the processor what to do when certain traps are encountered. These traps are architecture defined, and so the vector indices are always the same across all oeprating systems**. This is shown in Table 1:

Table 1: The ARM Exception Vector

Offset	Exception/Trap	Occurs when
0x00	Reset	Processor is reset
0x04	Undef	An undefined instruction is encountered. Usually this is from an erroneous branch or corruption of code, but can also be used for emulating instruction sets on processors which do not support them

Offset	Exception/Trap	Occurs when
0x08	Swi	A "Software Interrupt" is generated, by the SWI/SVC command. This is most often used to perform system calls: Code in user mode invokes the instruction, and the processor shifts to supervisor mode, to a predefined system call handler
0x0c	PrefAbt	Instruction Prefetch abort
0x10	DataAbt	Data Abort
0x14	AddrExc	An Address Exception (invalid address) is encountered
0x18	IRQ	An Interrupt Request is singaled: The CPU stops everything and transfers control to the interrupt handler.
0x24	FIQ	A Fast Interrupt Request is singaled: The CPU stops everything and transfers control to the interrupt handler. Other interrupts are blocked during the time.

Boot loaders and operating systems alike, then, have a simple mode of operation: When a component wants to transfer control to another component, it loads it into memory, in the process overwriting the exception vector with that of the next stage. A processor reset instruction is issued, which causes the processor to jump to the next stage's entry point.

Armed with this information, we are now ready to proceed to a meaningful disassembly of aboot

Disassembling Aboot

As discussed in the book, the standard aboot is derived from the "LittleKernel" project, hosted by CodeAurora. This project (often referred to as "LK"), is partially open sourced, in that some of its components can be found there (A simple way to get the sources is via git clone git://codeaurora.org/kernel/lk.git). Nonetheless, quite a few of its interesting components (such as the implementation of oem fastboot commands) are not included in the source. Additionally, some vendors (notably Samsung) use their own implementation. The following steps will prepare aboot for disassembly:

1. Strip the image of its header: This can be done by using dd if=/dev/aboot of=aboot.sans.header bs=40 skip=1.
2. Trim the image to CodeSize bytes: This can be done by using split -b *CodeSize* aboot.sans.header. You will get two parts: xaa and xab. The xab contains the singature and certificates, so its size should be exactly that of (Certs + SigSize).

Following these steps, and loading the result into a disassembler (e.g. IDA or Hopper), you'll still need to rebase the image to the *ImgBase* value specified in the header (which you've stripped). You'll see something similar to this:

```
0F900000        ea000006        B       0x0F900020      ; B  _reset ---------+
0F900004        ea00351c        B       0x0F90D74C       ; B  _undef         |
0F900008        ea003522        B       0x0F90D498      ; B  _swi           |
0F90000C        ea003528        B       0x0F90D4B4      ; B  _prefabt       |
0F900010        ea00352E        B      0x0F90D4D0      ; B  _dataabt        |
0F900014        ea003534        B       0x0F90D4EC      ; B  _addrexc       |
0F900018        ea003534        B      0x0F90D4F0      ; B  _irq           |
0F90001C        ea00354B        B    0x0F90D550      ; B  _fiq           |
_reset:                                                                    |
0F900020        ee110f10        MRC     p15, 0, R0,c1,c0, 0 ;<--------------+
0F900024        e3c00a0b        BIC     R0, R0, #0xB000
..
```

The code above can be seen in LK's ARM support, vertaim. If you follow the branches to the various exception handlers you should likely see the code from LK's ARM exception support, as well. All Bootloaders I've checked (including Samsung's and Amazon's) seem to derive from LK (in fact printing out the "welcome to

lk\n\n" from its kmain), though Amazon's take steps to disable fastboot, as does Samsung, which then uses their own custom App, called Odin. This makes sense, since LK is developed by Qualcomm for use with SnapDragon (msm), which was a common denominator for my test devices (I haven't had a chance to check the Exynos-based devices).

Aboot flow

Aboot's flow spans multiple files. It is, however, a one-way flow, since functions are not expected to return. This useful tidbit lends itself to finding stop points throughout aboot, which are "B ." (i.e. goto here) instructions in the code, which effectively halt the processor for those cases that something in the flow goes wrong. This actually made it easier to piece the flow from the disassembly first ***. Calls to dprintf help, though surprisingly the Nexus 5, of all devices, strips them. The flow of aboot's LK is shown in Table 2. For convenience, the functions are mapped to the files which contain them.

Table 2: LittleKernel Boot flow

LK file	Function	Description
arch/arm /crt0.S	reset	The reset handler initializes the processor using various low level instructions (MRC, which reads from the coprocessor, and its counterpart, MCR, which writes back values after bit modifications).
	stack_setup	Sets up stack for the various processor modes (IRQ, FIQ, ABT, UND, SYS and SVC). This is done by entering each state (using MSR CPSR_c), and changing the value of the stack pointer (R14). The branch from here to the kmain is a BLX, which converts the assembly from ARM to Thumb2 (and from Assembly to C).
kernel/main.c	kmain	Main initialization function. Calls a sequence of other functions (next) in order.
lib/heap /heap.c	heap_init	Initialize LK's heap memory (free list)
kernel/thread.c	thread_init	Usually empty - handled by thread_init_early
kernel/dpc.c	dpc_init	Spins the "dpc" thread. This is one of two threads spun off by aboot (the other being bootstrap2)
kernel/timer.c	timer_init	Initialize a timer list, and request a periodic tick from the architecture
kernel/main.c	bootstrap2	A thread spun off from kmain, by a call to thread_create (), then thread_resume. The thread name ("bootstrap2") appears in all LK variants, which makes it easy to find all the functions involved. The bootstrap2 address can be discerned from its loading into R1 (as the second parameter to thread_create)

After the creation of the bootstrap2 thread, the main thread re-enables interrupts, and becomes idle. The very last call in kmain handles the idle thread, as you can see by the following disassembly. The addresses are those of the S5 (as you can tell by the 0F8xxxxx). The N5 code is identical, right up to the addresses, which is why the code below has labels:

```
_become_idle_thread:
0F8164CC            STMFD   SP!, {R3,LR}
0F8164D0            LDR     R0, "idle"
0F8164D4            BL      _thread_set_name
0F8164D8            LDR     R3, current_thread
0F8164DC            MOV     R2, #0
0F8164E0            LDR     R3, [R3]
0F8164E4            STR     R2, [R3,#0x14]
0F8164E8            LDR     R2, idle_thread
0F8164EC            STR     R3, [R2]
__idle_loop:
0F8164F0            BL      _arch_idle ; Calls ARM WFI (Wait-For-Interrupt)
0F8164F4            BL      _arch_idle
```

```
0F8164F8        B       __idle_loop
```

The flow thus continues in bootstrap2..

LK file	Function	Description
platform/..	platform_init	Platform specific functions. Left with a dprintf on most loaders
target/..	target_init	Target (device) specific functions. Implementation varies.
app/app.c	apps_init	Initialize boot loader "applications". These, too, vary with device.

The apps_init function is the most important function, since it is through "apps" that the bootloader provides functionality. The function iterates over an array of app descriptors, calling each one's "init" function, and then spinning a thread for each. Each app thus runs in its own thread, and vendors are free to add or remove apps as they see fit. Commonly defined apps are:

LK file	App
app/aboot/aboot.c	aboot itself: calls on boot_linux to boot Linux from MMC or Flash
app/fastboot/fastboot.c	Fastboot protocol support: spawns two threads (fastboot, fastboot_menu). The latter is disabled by a simple check in Kindle, and both are replaced in Samsung by Odin.
app/recovery/recovery.c	Recovery mode

This should get you started on your own explorations of Android's Boot Loader. The platform support stuff isn't that interesting (USB being somewhat of an exception, due to its usage as a potential attack vector). Apps, in particular, are obviously important. Of special interest is Odin, to which I hope to devote another article at some point. As usual, feedback and/or requests are appreciated - info@ (this domain).

Q&A

Q: How did you uncover the header format?

A: I did it by reversing - all boot loaders on ARM have a similar structure imposed by the architecture (e.g. the Exception vector), so it was easy to work backwards from the eXXXXXXX instructions ("e" is ARM speak for "always" and so branch instructions stand out). Values which repeat (e.g. the code size, which then appears along with ImgBase) make it easy to figure out "hey, this value is that value + that value". When compared to the file size, the values further add up. Certs stick out like a sore thumb because they are a known format (PKCS #7? #12? whatever), and their hard coded strings can be easily seen.

Of course, it turns out that this is all also in a C file in some obscure LK directory... :-P

Q: What's this about rebasing the image?

A: Needed only if you plan to load in a disassembler, e.g. IDA. When you do, you will see the branch addresses are all relative to PC, but there are quite a few hard coded addresses (for LDR..) which are all absolute, so you need to know what the ImgBase is. Even if you don't, you will see those addresses begin with 0xF9 or similar, which gives you the image base. What's nice is that, if you get it wrong, the addresses won't make sense (i.e. will be outside the image). If you do, everything literally falls into place.

Footnotes

* - At least, on ARM devices it does. On the few Intel devices I've had the pleasure of working with, BIOS was used. Intel is likely trying to push UEFI for its devices, though I haven't seen any personally. If you have an Intel device and want to send me a dump of it's boot partition, I'd appreciate it.
** - iOS uses the same technique throughout its boot stage, which involves iBSS, iBoot, and XNU. This is covered in the other book, in Chapters 6 and (with disassembly) 8.

*** - (which I did, foolishly, before I knew about the source being accessible - talk about hours of my life I won't be getting back).