Boot Sequences

From Linux Rockchip

This page is currently under the review. please don't edit this page without discussion.

This page condenses information found on the net, namely IRC, the G+ talk (http://talk.radxa.com/) , the Radxa Rock wiki (http://radxa.com/Rock) . Valuable input also came from Galland's blog (http://hwswbits.blogspot.com/) , androtab.info (http://androtab.info/radxa_rock/) .

If it is lifting the mist and myths for you about booting (RockChip) SoCs then it's main purpose has already been achieved.

Disclaimer: Errors have most likely been introduced by me (--Corb (talk) 00:52, 5 December 2014 (EET)). Authoritative sources are *naobsd*, *Astralix* and others on freenode's *#linux-rockchip* IRC channel, and of course the source. This document has yet to undergo proof-reading.

Contents

Bootloaders aka Loaders

There are different types of bootloaders. A short and general description of the tasks a bootloader (http://radxa.com/Rock/bootloader%7C) has to fullfill.

RockChip Bootloader RKxxxxLoader

A proprietary (read: binary-only, closed source) bootloader released by RockChip every now and then.

- arctablet's RockChip bootloader collection (http://www.arctablet.com/blog/forum/firmware-development/rockchip-bootloader-collection-rk29xxloader-rk30xxloader-rk3168loader-rk3188loader/)
- Radxa RockChip bootloader site (http://dl.radxa.com/rock/images/loader/)

Caveat:

The bootloader initializes DRAM and flash memory. Memory is provided by electronic parts soldered on the board and different manufacturers choose different electronic parts for their products and even may vary these parts within one product line. Software that wishes to initialize the memory has to support these electronic parts. So, trying a different bootloader version, it may happen that the bootloader does not support those exact electronic parts present on the board. In this case, the bootloader already flashed will fail to boot as it cannot initialize the memory. To remedy this situation, it will come in handy to have an unmodified copy of the firmware image of your device so to extract a known working bootloader and retry.

Binary Bootloader and Linux K ernel Module Dep endencies

For the proprietary bootloader to boot successfully, the kernel the bootloader version has to be matched by the proprietary kernel NAND driver modules. See the following to get the idea what exactly has to be payed attention for:

- Bootloaders are SoC specific, so there's one binary bootloader for every SoC: i.e.
  - version 1 bootloaders *RK3066Loader(L)_v1.xx.img*, *RK3188Loader(L)_v1.xx.img* and
  - version 2 bootloaders *RK3066Loader(L)_v2.xx.img*, *RK3188Loader(L)_v2.xx.img*.
- Version 1 bootloaders have a Linux kernel module *rk<SoC>nand_ko.ko.<kVer>* corresponding to the SoC and Linux kernel version. I.e. with respect to Linux kernel versions 3.0.8+ and 3.0.36+ there are these kernel modules:
  - *rk30xxnand_ko.ko.3.0.8+*, *rk30xxnand_ko.ko.3.0.36+* and
  - *rk31xxnand_ko.ko.3.0.8+*, *rk31xxnand_ko.ko.3.0.36+*.
- Version 2 bootloaders have a Linux kernel module *rknand_ko.ko.<kVer>* corresponding to the Linux kernel version. I.e. with respect to Linux kernel versions 3.0.8+ and 3.0.36+ there are these kernel modules: *rknand_ko.ko.3.0.8+* and *rknand_ko.ko.3.0.36+*. These kernel modules are SoC generic.

Note:

```
One source for getting these files might be an  update.img  file from which to extract
the bootloader and Linux kernel modules.
```

Linux Kernel NAND Module Loading Sequence

The Linux kernel modified by RockChip will first try to load a SoC and Linux kernel version generic *rknand_ko.ko* module. If this fails it tries next the generic module name with the kernel version appended: *rknand_ko.ko.<kVer>*. The reasoning behind this is that with version 2 bootloaders and this particular modprobe sequence it is possible to boot a system with a kernel of version 3.0.8+ on a 3066 SoC and the same system with a kernel of version 3.0.36+ on a 3188 SoC without changing the system itself. The modified Linux kernel will handle the loading independently of the underlying SoC or the actual kernel version.

Binary Bootloader and the Memory La yout

Switching proprietary major bootloader versions (i.e. v1.16 <-> v2.10) implies completely wiping and reformatting NAND (as the NAND layout changes between major versions).

TODO: add IDB information here.

Details on the RockChip Boot Sequence

Remember, this description applies to the proprietary RockChip (RK) bootloaders only. Contrarily, this description does not apply to coreboot or u-boot. Furthermore, this description does not encompass the *MASKROM mode* of the proprietary RockChip (RK) bootloaders, except when explicitely stated.

The RockChip bootloader generally uses a deterministic (read: "fixed") boot sequence. However, this "fixed" sequence may change between one second stage bootloader version and another.

The bootstrapping code (first stage loader, actually, this _is_ the *MASKROM* loader) lives in the SoC ROM. On power-on, it searches for the (second stage) bootloader (the one this description applies to) on any bootable media in a fixed order.

Bootable media may be one or more of:

- eMMC,
- MMC (specific models only - see SoCs),
- SPI NOR flash and
- NAND flash
- OTG USB (firmware download only)

The order which is used to query the media for the next stage bootloader is yet to be determined.

In case the bootstrapping code finds a bootable medium, it will try to validate the bootloader code to be executed. This means the bootloader code has to conform to a particular binary header format and possibly there is a checksum to be matched. In the other case, i.e. there is no such medium or the bootloader code does not validate, then the next medium is examined.

MASKROM mode: If no bootable medium is found or none contains a valid bootloader, the bootstrapping code initializes the USB in device mode (OTG). In this mode, it is possible to download firmware (second stage bootloader) to the SoC via USB OTG. The actual transfer uses a special RockChip *MASKROM* bootloader protocol.

HINT: How to enter *MASKROM mode*.

In NAND flash memory, the second stage bootloader resides inside a reserved memory range. This memory range cannot be read from or written to by the block device interface (*/dev/mtd[0-9]*) the Linux kernel provides to access NAND flash storage.

In eMMC or MMC storage, the second stage bootloader is located at a specific logical sector. This storage type relies on the user to properly partition the storage's address space so that enough space is reserved for the bootloader.

The second stage bootloader (proprietary RockChip bootloader) tries to read the *parameters* at the first logical sector in NAND flash or at sector 0x2000 (8192) when booting from SD Card or eMMC/MMC. In the latter case the address space before sector 0x2000 is reserved.

The *parameters* include the partition layout used by the Linux kernel on the memory or storage device.

Once the *parameters* are known the bootloader tries to load an OS kernel on one of the partitions defined in those *parameters*, i.e. the partitions *boot*, *recovery*, *kernel* etc.

The bootloader tries to load the kernel from the *recovery* partition if the "recovery" button is pressed (i.e. available on the RK3188 device). It is legal for a plain ramdisk-only image to be present in the recovery partition. If so, the OS kernel itself is loaded from the *kernel* partition.

The standard boot process involves reading the OS from the *boot* partition. Again, if the partition contains a ramdisk-only image (or the code is somehow invalid), the kernel is loaded from the *kernel* partition. If the code in the *kernel* partition is equally invalid then the bootloader tries to finally load the kernel from the *recovery* partition. But please keep in mind that -- as initally stated -- this sequence may possibly be subject to change.

The *kernel* partition contains the kernel and initramfs, the image file wrapped by *rkcrc -k*, a non-empty dummy image file (one that contains NULL and is mabe 1KiB -- TODO: to be verified -- in size) may then be written to the *boot* partition, again wrapped by *rkcrc -k*.

If the partition layout has a *boot* partition but no *kernel* partition, the image file may be wrapped by *mkbootimg* (c.f. #Prepare the Files).

In summary, the *boot* partition shall generally contain an image file with kernel and ramdisk, wrapped by *mkbootimg* or an image file just containing a ramdisk (!), wrapped by *rkcrc -k*.

An image file containing a kernel that has an initramfs attached is also a valid configuration (just regard this kernel with initramfs attached as an ordinary kernel).

It is important to differentiate between a kernel attached **initramfs** and a standalone **ramdisk**. For example, an image file containing a kernel with or without an initramfs and wrapped by *rkcrc -k* must not be written to the *boot* partition (invalid configuration).

In summary, valid partition layout and image file content combinations are:

- a *kernel* partition containing a kernel and a *boot* partition containing a (standalone) ramdisk, both wrapped by *rkcrc -k* and

- a *boot* partition containing a kernel with (standalone) ramdisk, wrapped by *mkbootimg*.

In the above two combinations a *recovery* partition may be used instead of the *boot* partition. No other partition layout and image file content combinations are invalid. The bootloader will only be able to successfully load an OS if correct preconditions are met.

Sample steps of a valid procedure:

1. define a *kernel* and a *boot* partition
2. create kernel (**may** have an initramfs attached), wrap that by *rkcrc -k*, resulting file: *kernel.img*
3. create dummy file (for criteria see above), wrap also by *rkcrc -k*, resulting file: *boot.img*
4. flash *kernel.img* to *kernel* partition
5. flash *boot.img* (dummy file) to *boot* partition

When using kernel with with initramfs attached there is no need of a (standalone) ramdisk be present in the *boot* partition. The bootloader however requires a valid signature (that is what the *rkcrc -k* and *mkbootimg* actually do, create a signature and checksum, wrapping the input) be found in the *boot* partition, thus the -- wrapped/signed -- dummy file. A valid (signed) *kernel.img* could equally be flashed to the *boot* partition instead of creating and flashing a dummy file. The important thing is the **valid signature**.

The bootloaders requires valid signatures in the partitions even if the "code" inside is garbage. The loader does only care about the signature, not the code itself. The payload (code) "only" gets executed, if garbage is tried to be executed then the boot fails obviously, but the bootloader has been happy so far. Instead of flashing wrapped garbage it would naturally be more sensible to use a wrapped Linux or *BSD kernel.

When the kernel needs a (standalone) ramdisk, just write the wrapped ramdisk to the *boot* or *recovery* partition and the bootloader will find and load both of them. Even if the kernel has a initramfs attached an thus does not need an external (standalone) ramdisk, the bootloader will load the content of the *boot* or *recovery* partition but the kernel will simply not use that content.

The signature in the image files is added by the wrappers *rkcrc* or *mkbootimg*. The first few bytes indicate to the bootloader a valid or invalid partition content. The actual signature is really just the string

- **KRNL** when created by *rkcrc* (i.e. for a standalone kernel or a standalone ramdisk) or
- **ANDROID!** in case of *mkbootimg* (i.e. for a kernel and a ramdisk together).

It is easily verified which tool creates which signature. Just run them both on two different files and look at the first few bytes (i.e. using *head -n1*).

The *boot* partition shall contain either a standalone ramdisk (signature: **KRNL**) or a kernel with external ramdisk (signature: **ANDROID!**).

The bootloader will first try to boot from the *boot* or *recovery* partition:

- If the signature read by the bootloader is **ANDROID!** then the bootloader "knows" there is a kernel present in this *boot* or *recovery* partition and loads the kernel and the ramdisk.

- If the ignature read by the bootloader is **KRNL** then the bootloader "knows" there is a **ramdisk** in this *boot* or *recovery* partition and it must then try to load the kernel from the *kernel* partition. Note, that in case there is no ramdisk but just a dummy file the bootloader won't notice nor bother the bad payload, it only checks for the signature! This explains why we validly could, lazy devs who we are, flash a *kernel.img* wrapped by *rkcrc -k* to the *boot* partition and everything will work fine (as long as there is a valid kernel in the *kernel* partition..).

Disclaimer: Large parts of this section are rephrased from this IRC log.

MASKROM Mode

If the device boots from a NAND bootloader and it is desired to modify the bootloader or the device is just "bricked", then it is possible to deliberately enter *MASKROM mode*. When shorting pins 8 and 9 on the NAND chip, the device is prevented from reading from NAND flash memory, fails to find a valid medium and bootloader and thus enters *MASKROM mode*.

It is also necessary to enter *MASKROM mode* when messages like

- "Prepare IDB Fail" or
- "IDB download failed"

appear while flashing NAND. See also Radxa Rock unbricking (http://radxa.com/Rock/unbrick) .

U-Boot

Feature-rich and versatile OSS bootloader, supporting a multitude of boot methods.

- u-boot documentation (http://www.denx.de/wiki/DULG/Manual)
- Radxa/RockChip u-boot sources (https://github.com/radxa/u-boot-rockchip.git) , Radxa u-boot (http://radxa.com/Rock/U-Boot) , Rockchip u-boot sources (https://github.com/neo-technologies/rockchip_u-boot) , androtab u-boot (http://androtab.info/rockchip/u-boot/)

Boot Methods

There are a variety of boot methods available. Depending on the bootloader used (c.f. above bootloader details on boot sequences and documentation links) sophisticated fallback boot chain may be established.

- NAND
- SD Card
- eMMC
- network: bootp, rarp, tftp, dhcp, NFS
- USB

One particular fallback boot chain commonly used during development is to setup NFS boot, and if that fails try to boot from SD Card and USB-stick, and if none of them succeeded so far, finally try to boot from NAND.

RK3066

Sarg's gist (https://gist.github.com/sarg/5028505%7C) : *(This description may need work.)*

1. at power-on, bootrom[0x0] is read
2. bootrom copies itself to SRAM and proceeds to find DRAM init handler in IDBROM
3. bootrom loads and executes DRAM init handler in memory
4. bootrom searches for *flashboot* and loads it in memory
    1. if there is no *flashboot* bootrom searches for USBplug and loads it
5. *flashboot*
    1. initializes NAND
    2. searches a *parameter* file
    3. loads the kernel in DRAM
    4. executes the kernel

If your hard earned research is not explicitly listed here, sorry, it's not a personal slight. May it live forever in the common brain.

SD Card Boot

This SoC is understood not beeing able to boot from sdcard. TODO: To be verified.

RK3188

SD Card Boot

Note:

Basically, it should be feasable to write a NAND bootloader to a SD Card and boot from SD Card.

If there is onboard eMMC available it will take priority over SD Card when booting a precompiled kernel preventing a boot from SD Card. Similarly, bad eMMC can prevent the device from booting from SD Card any more. To avoid/solve these issues, deactivate the *dwmmc* or the slot the eMMC is connected to.

SD Card Storage Layout for Boot

To be able to boot from SD Card there has to be a particular storage layout (structure) on the SD Card containing header fields and partitions. The partition layout is defined inside the *parameter* file as part of the Linux kernel commandline parameters:

```
CMDLINE:console=ttyFIQ0 androidboot.console=ttyFIQ0 init=/init initrd=0x62000000,0x00800000 \
mtdparts=rk29xxnand:0x00002000@0x00002000(misc),0x00008000@0x00004000(kernel),\
0x00008000@0x0000c000(boot),0x00010000@0x00014000(recovery),0x00002000@0x00024000(backup),\
0x00040000@0x00026000(cache),0x001fe000@0x00066000(userdata),0x00002000@0x00264000(metadata),\
0x00002000@0x00266000(kpanic),0x00180000@0x00268000(system),0x00200000@0x003e8000(user),\
-@0x005e8000(linuxroot)
```

Or just this (SD Card booting u-boot):

```
CMDLINE:console=ttyFIQ0 androidboot.console=ttyFIQ0 init=/init initrd=0x62000000,0x00800000 \
mtdparts=rk29xxnand:0x00002000@0x00002000(uboot)
```

As opposed to the above configurations let's now consider the following exemplary layout (sources?):

TODO: this part needs to be thoroughly verified.

ADD: I think loader code is placed at raw 4th sector in NAND, but it doesn't matter. you may put u-boot as a loader in NAND area or you may put u-boot as a kernel which is (chain-)loaded from loader in NAND you may put (any arm code here) as a kernel which is loaded from loader in NAND

NAND Sector (size 512B)	SD Card Sector (size 512B)	SD Card HEX Address (dec)	Description
logical sector 0	16	0x2000 (8192)	''Parameter'' file. SD Card sectors 0 through 15 are reserved.
64	64	0x8000 (32768)	magic header (1). Contains details about the two code-areas (68, 92).
65	65	0x8200 (33280)	magic header (2). Only used by proprietary RockChip bootloader?
68	68	0x8800 (34816)	Start of DRAM init code: *FlashData*, RC4-scrambled. I.e. parts from file *3188_LPDDR2_300MHz_DDR3_300MHz_20130830.bin*
92	92	0xb800 (47104)	Start of bootloader code: *FlashBoot*, RC4-scrambled. This is also where other bootloaders, i.e. u-boot, are located.
160	160	0x14000 (81920)	Starting with sector 160 (address 81920, or whatever has been defined in the *parameter* file) there can be user defined partitions.

From sector 160 on, the partitions follow as defined and in the same binary format known from NAND.

By default, the space up to sector 159 (roughly 36MiB) is reserved for the bootloader, kernel and ramdisk. The size of this reserved space can be modified by changing the mtdparts information in the *parameter* file in the kernel *CMDLINE*.

At the end of sector 0x2000, an RSA key is located (maybe for secureboot/DRM) and the u-boot environment is stored (if the bootloader is u-boot).

eMMC Boot

eMMC Storage Layout for Boot

The storage layout for eMMC boot is the same as for SD Card boot. First 0x2000 raw sectors (header), then user defined partitions starting from sector 0x2000 which is logical sector 0x00.

Creating the Structure

The header and code areas are scrambled by RC4.

Header:

```
openssl rc4 -K 7C4E0304550509072D2C7B38170D1711 < header > header.rc4
```

Code:

```
split -b 512 --filter='openssl rc4 -K 7C4E0304550509072D2C7B38170D1711' < code > code.rc4
```

The script *gen.sh* in *u-boot-rk3188-sdcard.zip (http://files.androtab.info/rockchip/u-boot/u-boot-rk3188-nand.zip)* contains:

```
#!/bin/sh

dd if=orig/sd_header_64 conv=sync | split -b 512 --filter='openssl rc4 -K \
7c4e0304550509072d2c7b38170d1711' > sd_header_64.enc

dd if=orig/3188_LPDDR2_300MHz_DDR3_300MHz_20130830.bin conv=sync | split -b 512 \
--filter='openssl rc4 -K 7c4e0304550509072d2c7b38170d1711' > FlashData.bin

dd if=orig/u-boot.bin conv=sync | split -b 512 --filter='openssl rc4 -K \
7c4e0304550509072d2c7b38170d1711' > FlashBoot.bin

rkcrc -p orig/parameter parameter.img
```

See also naobsd's excerpt (http://talk.radxa.com/topic/278/custom-sd-card-boot-image/4%7C) from sarg's gist (https://gist.github.com/sarg/5028505%7C) .

FlashData and FlashBoot contained in *RK3188Loader(L)_Vx.xx.bin* can be extracted with sarg's tools (https://gist.github.com/sarg/5028505%7C) .

Prepare the Files

As stated above, the binary format for code that can be written to SC Card partitions and the binary format for NAND code is the same.

To wrap the code into the right binary format (it adds some header bytes and CRCsums) use:

- for *parameter* files:

  ```
  rkcrc -p <parameters.txt>
  ```

  See also the *rkcrc*-call in *gen.sh*.
- for kernel, boot and recovery images:

  ```
  rkcrc -k <{kernel|boot|recovery}.img>
  ```

  alternatively there is a command for boot and recovery images (for the kernel image the above *rkcrc*-command has to be used):

  ```
  mkbootimage <{boot|recovery}.img>
  ```

Write the Code to SD Card

Once prepared, the files can now be flashed to the SD Card using the *dd* command. Unless you defined your own layout the sectors given to the command are the same as in the table above. For user defined partitions write the files to the corresponding sectors/addresses.

The parameters go to SD Card sector 16 at address 0x2000 (would be NAND logical sector 0x0):

```
dd if=parameters.img of=/dev/XXX conv=sync,fsync seek=$((0x2000+0x0))
```

The SD Card header (RC4 scrambled) goes to sector 64:

```
dd if=sd_header_64.rc4 of=/dev/XXX conv=sync,fsync seek=64
```

Presumably some proprietary code (cleartext or RC4 scrambled, too?) goes to sector 65 (but might be omitted, to be verified):

```
dd if=proprietary_sd_header_65 of=/dev/XXX conv=sync,fsync seek=65
```

DDR initialization routines (RC4 encrypted) go to sector 68:

```
dd if=FlashData.bin.rc4 of=/dev/XXX conv=sync,fsync seek=68
```

The bootloader (RC4 scrambled) lives in sector 92:

```
dd if=FlashBoot.bin.rc4 of=/dev/XXX conv=sync,fsync seek=92
```

Finally, the kernel and/or RFS/initrd/initramfs go to addresses far beyond, at sectors 160ff.:

```
dd if=kernel.img of=/dev/XXX conv=sync,fsync seek=$((0x2000+0x4000))
dd if=boot.img of=/dev/XXX conv=sync,fsync seek=$((0x2000+0xc000))
```

Note:

The size of the bootloader section (sector 92 and up) is sufficient for the proprietary Rockchip bootloaders. But it is too small for u-boot or barebox to fit in.

```
naobsd: in my sdboot img, size of loader are is good for RK proprietary loaders, but
too small for u-boot/barebox
```

See also:

- How to write images to SD Card using Linux (http://androtab.info/radxa_rock/sdboot/)
- Minimalistic Linux system able to boot another OS (http://androtab.info/miniroot/)
- How to write images to SD Card using Linux/Windos (http://wiki.radxa.com/Rock/SD_images)
- Some prebuilt SD Card images etc. (http://wiki.radxa.com/Rock/prebuilt_images)

- SD Card images with u-boot (http://androtab.info/rockchip/u-boot/)

RK3288

TBD.

Retrieved from "http://linux-rockchip.info/mw/index.php?title=Boot_Sequences&oldid=1104"
Category: Hardware