Application Report
SPRAC30A–January 2016–Revised February 2018

Android Boot Optimization on DRA7xx Devices

ABSTRACT

Boot time optimizations are critical for achieving the perception of immediate application availability in the Automotive Infotainment system. They impact the end user's experience by improving the launch time of applications and infotainment use-cases. This application report captures the details on how to optimize Android software stack, along with a few optional early infotainment system features, and is meant to be used as a reference guide. The end product owner (OEM/ODM/Customer) can review the guidance for improving boot time for their specific Android powered system.

Contents

Trademarks

1 Introduction

Boot time optimization is an area of interest for all automotive customers in systems where a cold boot is required more often. In these cases, a faster boot time is critical. This document provides a reference using TI’s Processor SDK Android to improve the overall system boot time to the Android Home screen. It also provides details on optimizing specific Infotainment use cases like Early Splash-screen, Early Video, Early RVC (Rear View Camera), Early Audio, and Early launch of Android applications.

The strategy shown here, along with instructions included to modify the bootloader, kernel and Android sources, are all related to reducing the total boot time for Android Marshmallow running on the Jacinto 6 family of devices. The goal is to have the bootloader (stage-1) ready in 200msec, the kernel in 2 sec, and to get to the Android home screen in 8 sec from the start of a cold boot.

1.1 Overall Strategy

- Run system at max graded efficiency (QSPI-64Mhz, eMMC-192Mhz, CPU-1.5Ghz).

NOTE: eMMC HS200 mode enablement is possible only on J6 ES2.0 devices.

- U-boot/SPL code built to do single-stage booting with direct jump from the SPL to the kernel
- Linux kernel built with a custom config fragment (based on git.ti.com/android-sdk/kernel-omap, branch: 6AM.1.3-*-rvc-video-earlyboot) [where ‘*’ is the specific family device in each example]
- Disable the additional debug options and features in the kernel for Android that are not needed in the final product.
- Disable serial console output – ‘adb-dmesg’ should still be able to show the same log
- Modularize specific kernel drivers in order to boot the uncompressed kernel Image more quickly (reducing kernel image memory footprint)
- Load kernel modules in parallel to the Android userspace initialization (or, load modules later if possible).
- Initialize Audio as early as Android stage 1 initialization
- Initialize Display and video from bootloader and continue loading of early video and audio usecases with auxiliary M4 cores in parallel to android boot thru Arm® Cortex® cores.

Using the specific techniques above, a boot time (from power-on to Android Homescreen UI) of 8.5 seconds was measured. Further Android userspace optimizations will yield potential boot time savings of 30%-40% on top of what was already achieved.

NOTE: For custom product configurations only.

1.2 Boot Optimization Demo

Demo Video showing boot optimization of key infotainment usecases exhibited on TI DRA7xx devices:

http://www.ti.com/general/docs/video/watch.tsp?entryid=5522456447001

This Optimized Demo shows how quickly you can boot and run key android infotainment applications and usecases. Similar percentage improvements can be seen on custom products once the overall product feature goals are realized and optimize specific usecases, remove or defer non-applicable scenarios to much later in the boot process.

Using the specific techniques above, the optimized boot times for various system usecases were measured as shown in Table 1.

Table 1. Boot Time Improvement Measured Times

KPI	6AM.1.3-J6	6AM.1.3-J6-Optimized	Improved %	6AM.1.3-J6 Entry	6AM.1.3-J6 Entry- Optimized	Improved %
Uboot Prompt	1.6	0.2	87	0.8	0.2	75
Kernel Start	4.0	0.7	82.5	3.2	0.7	79
Early Video	N/A	1.5	N/A	N/A	1.5	N/A
RVC	1.8	1.6	11	N/A	1.6	N/A
Audio Chime	24.7	2	92	31.5	2	93
FS Start	8.8	2	77.2	10	2	80
Kernel Prompt	10.5	3	71.7	12.2	3.2	74
Android Splash Screen	11.2	3	73.2	12.4	4.3	65
HMI	21.2	8.5	60	30.8	17	45
Early Navi	21.8	8.5	61	31.5	18	41

2 Build Instructions

2.1 Basic Process - Basic Steps (high-level overview)

1. Download and build U-boot for Jacinto6.
2. Download, patch, and build kernel with alternate config fragment for quickboot.
3. Download and patch device/ti/jacinto6evm project (ramdisk.img) with init.rc changes to load modularized kernel drivers in the proper order, and with other changes to improve boot time.
4. Build the updated AFS for J6 (from 6AM.1.3 release notes Wiki, or other recent repo sync).
5. Copy uncompressed kernel Image over to AFS for boot.img build as part of AFS images build.
6. Re-build Wifi modules and SGX PVR Services kernel module (because kernel configuration has changed).
7. Re-build AFS images after all kernel modules are added.
   a. Flash MLO, u-boot.img, kernel (as part of boot.img) and new system.img.
   b. Switch to single-stage boot mode and re-boot.

2.2 Building With the 6AM.1.x Release Notes as a Guide

Most of the overall build process for U-boot, kernel, and AFS remains the same as in the Release Notes Wiki page (http://omapedia.org/wiki/6AM.1.3_Release_Notes).

However, the specific components mentioned on this page must be built using the steps shown here:

1. Build components as described in Section 2.3.
2. Build the rest of the components as described in the Release Notes (http://www.omapedia.org/wiki/6AM.1.3_Release_Notes)
3. Prepare files for flashing as described in the Release Notes.
4. Flash files as described in the Release Notes.

2.3 Preparing and Build Instructions

2.3.1 Prepare

Make sure the build environment is properly configured, see the *Build PC Setup* and *Android Filesystem Source* sections in the Release Notes wiki (http://omapedia.org/wiki/6AM.1.3_Release_Notes).

2.3.2 Building U-Boot

Build U-boot per the instructions in the release notes:

- http://omappedia.org/wiki/6AM.1.3_Release_Notes#U-Boot_Sources
- http://omappedia.org/wiki/6AM.1.3_Release_Notes#Building_U-boot_sources

Switch to an optimized uboot branch that includes fixes for boot improvements and early rvc and video usecases.

- For J6: http://git.ti.com/cgit/cgit.cgi/android-sdk/u-boot.git/log/?h=6AM.1.3-rvc-video-earlyboot
- For J6Entry: http://git.ti.com/cgit/cgit.cgi/android-sdk/u-boot.git/log/?h=6AM.1.3-j6e-rvc-video-earlyboot
- defconfig: dra7xx_evm_defconfig

1. Reset to commit ID and build bootloader images.

   http://git.ti.com/cgit/cgit.cgi/android-sdk/u-boot.git/commit/?h=6AM.1.3-rvc-video-earlyboot&id=0fe8b0ee48673a45901c65d2d574243072c5a842
2. Reset to commit ID and build final step bootloader images.

   http://git.ti.com/cgit/cgit.cgi/android-sdk/u-boot.git/commit/?h=6AM.1.3-rvc-video-earlyboot&id=72a64f93d27989dbecaf207ff67b61a0762b62d5

   For J6Entry: http://git.ti.com/cgit/cgit.cgi/android-sdk/u-boot.git/log/?h=6AM.1.3-j6e-rvc-video-earlyboot

2.3.3 Building Kernel

Clone the 6AM.1.3 kernel per instructions in the release notes.

- http://git.ti.com/cgit/cgit.cgi/android-sdk/kernel-omap.git/log/?h=6AM.1.3-rvc-video-earlyboot

Switch to optimized kernel branch that includes fixes for boot improvements and early rvc and video usecases.

- For J6: http://review.omapzoom.org/#/q/status:open+project:kernel/omap+branch:p-ti-linux-3.14.y-android+topic:boottime
- For J6Entry:http://git.ti.com/cgit/cgit.cgi/android-sdk/kernel-omap.git/log/?h=6AM.1.3-j6e-rvc-video-earlyboot

Build kernel and modules with the alternate "quickboot" defconfig:

```
cd ${YOUR_PATH}/kernel/android-4.4
export CROSS_COMPILE=${MYDROID}/prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-4.9/bin/arm-
linux-androideabi-
make ARCH=arm mrproper
make android_earlyboot_defconfig
make LD=$(MYDROID)/prebuilt/gcc/linux-x86/arm/arm-linux-androideabi-4.9/bin/arm-linux-
androidkernel-ld uImage LOADADDR=0x80008000 dtbs modules
```

2.3.4 Building AFS

Follow instructions from the 6AM.1.3 Release Notes Wiki for downloading the AFS sources: http://omapedia.org/wiki/6AM.1.3_Release_Notes#Android_Filesystem_Sources.

NOTE: You can use any other Marshmallow AFS sources (it is suggested to at least include all the latest updates from 6AM.1.3 first...), as long as it works with the kernel sources you have built using the above information.

- For J6: http://review.omapzoom.org/#/q/topic:6am13_j6_benchmarking
- For J6Entry: http://review.omapzoom.org/#/q/topic:6am13_j6e_benchmarking

Do the Initial Build of AFS (if not previously done):

```
cd $MYDROID
. build/envsetup.sh
lunch full_jacinto6evm-userdebug
make -j4 clean
make -j4
```

2.3.5 Preparing Android Modules and Building Final Images

1. Copy the kernel modules that are built in kernel build process into system partition at system/lib/modules.
2. Rebuild the Graphics Kernel driver and copy the modules at system/lib/modules
   - Instructions: http://omappedia.org/wiki/6AM.1.3_Application_Notes#Re-Building_SGX_kernel_module
3. Rebuild AFS to generate updated images:

```
cd $MYDROID
lunch full_jacinto6evm-userdebug
rm out/target/product/jacinto6evm/*.img
rm out/target/product/jacinto6evm/obj/PACKAGING/systemimage_intermediates/system.img
make -j4
```

3 Deploying Instructions

3.1 Prepare for Flashing

This section describes how to obtain the files for flashing.

3.1.1 Copying MLO and U-boot

The MLO and u-boot.img have already been copied to your emmc_files directory: (http://omappedia.org/wiki/6AM.1.3_Release_Notes#Preparing_eMMC_binaries.2Fimages).

3.1.2 Copying boot.img, ramdisk.img, recovery.img, system.img

Copy the newly re-built image files and tools to your emmc_files dir:

```
cd $YOUR_PATH
cp -v ${MYDROID}/out/target/product/jacinto6evm/*img emmc_files
cp -v ${MYDROID}/device/ti/jacinto6evm/fastboot.sh emmc_files
cp -v ${MYDROID}/out/host/linux-x86/bin/{simg2img,make_ext4fs,mkbootimg,fastboot,adb} emmc_files
cp -v ${YOUR_PATH}/kernel/android-3.14/arch/arm/boot/Image emmc_files/kernel
cp -v ${YOUR_PATH}/kernel/android-3.14/arch/arm/boot/dts/dra7-evm-lcd10.dtb emmc_files
cp -v ${YOUR_PATH}/kernel/android-4.4/arch/arm/boot/Image emmc_files/kernel
cp -v ${YOUR_PATH}/kernel/android-4.4/arch/arm/boot/dts/dra7*.dtb emmc_files
```

3.2 Flashing

Program the QSPI flash and eMMC flash parts using the instructions from the 6AM.1.3 Release Notes Wiki: http://omapedia.org/wiki/6AM.1.3_Release_Notes#Flashing_eMMC_images.

NOTE: Use step-1 bootloader images (this has console enabled). Use step-2 bootloader images as final step that disables console availability

3.3 J6 EVM Single-stage Bootloader Booting

After the first-time flashing and booting of images (in instructions) is discussed in Section 3.2, switch over to the Single-stage booting method and re-boot: (http://omapedia.org/wiki/6AM.1.3_Application_Notes#Single_Stage_bootloader).

4 Tooling/Profiling

4.1 Kernel Time Logging

PRINTK_TIME

CONFIG_PRINTK_TIME, enabled in the kernel, will provide the kernel boot time information from the serial/dmesg level. This feature enables a time routine inside standard printk calls.

This information is helpful in profling and instrumentation of the kernel to identify the kernel functions which take longer time and further optimize them to improve boot time.

PRINTK_TIME helps in improving boot time from Kernel init to Android boot animation.

4.2 Serial Port Output Logging

GRABSERIAL

If boot time profling is needed from power-on-reset, grabserial is an alternate for that. Grabserial is a small host program based on python that reads the serial port and saves the logs in the host machine, along with the time stamp using the standard clock.

Grabserial is especially helpful in profiling time taken for the uncompression process, bootloader and also with a particular activity.

CONFIG_DEBUG_LL, CONFIG_EARLY_PRINTK needs to be enabled on the target TI platform.

4.3 *Android Boot Logging*

BOOTCHART

Bootchart is a built-in utility in the Android system. It is a graphically designed tool that takes the system traces from Android and represents them in a GUI on the host machine.

WARNING

Bootchart-enabled Android can increase the boot time by 7-10%.

Bootchart uses /procfs data periodically (uptime, stat, diskstats, ps).

4.3.1 Usage

1. Install host PC package “sudo apt-get install bootchart”.
2. Re-build Android Marshmallow with bootchart enabled:

```
$ cd mydroid
$ . ./build/envsetup.sh
$ lunch full_jacinto6evm-userdebug
$ export INIT_BOOTCHART=true
$ make clean
$ make
$ touch system/core/init/init.c
$ m INIT_BOOTCHART=true
```

3. Rebuild Android Kernel without SElinux:

```
$ Disable CONFIG_SECURITY_SELINUX
$ Remove from CONFIG_CMDLINE “androidboot.selinux=permissive”
```

4. Repackage and Flash Android updates system image with new “init”.
5. Trigger Bootchart using adb:

```
$ sudo adb root; sleep 5 ; sudo adb remount
$ sudo adb shell 'echo 60 > /data/bootchart-start'
$ OR can also set the timeout from CONFIG_CMDLINE "androidboot.bootchart=60"
```

6. Capture Bootchart logs and provide to the host utility:

```
$ logs will be at /data/bootchart
$ ~/mydroid/system/core/init/grab-bootchart.sh
$ java -jar /usr/share/bootchart/bootchart.jar ./bootchart.tgz
```

5 Optimization Steps

5.1 *Boot Media Selection*

The DRA7xx SOC supports booting from various boot media. However, from a fast boot perspective, only QSPI NOR and eMMC are relevant. One of the decisions involved in optimizing the boot time is the location of the bootloader, kernel and device tree. Based on the initialization time and the binary sizes, boot from QSPI was chosen. eMMC has a higher initialization time, which makes it unsuitable for small image sizes

5.2 *Single Stage Boot Mode*

When optimizing for boot time, the system integrator has already decided on the boot media and the locations where various binaries are stored. In this case, the flexibility offered by U-Boot is not needed and can boot from MLO to kernel directly in a single stage. This provides a saving of at least a second in boot process.

5.3 U-Boot Optimizations

The optimizations in MLO are limited to env support in MLO/SPL as it unnecessarily increases the time spent in the kernel.

U-Boot does not have any boot time optimizations, therefore, the single stage boot mode will be used. However, customizations are included for easy flashing of QSPI and eMMC using fastboot. Partitions to flash the required files are defiend for usecases such as early splash(logo partition), early video (data partition), remotecore partitions (for loading remotecores early).

6 Adding Early Infotainment (IVI) Features to the Android System

6.1 Early Audio

Early audio is implemented through an Android utility that combines the PCM sample rendering and the ALSA mixer control setting in a single executable. The early audio executable, the related configuration file and WAV file are stored in the ramdisk image to facilitate starting playback in early Android boot stages: http://review.omapzoom.org/#/q/topic:early_audio.

6.2 Early Video

The early video usecase is achieved by performing playback from IPU2 prior to Android boot. The IPU2 firmware is started from bootloader, and then the usecase plays back the video without communication with Android. Apart from adding the playback usecase in IPU2 code, changes were done in bootloader and kernel.

Reference fixes: Kernel and Bootloader patches are available in the referred git.ti.com/android-sdk projects and branches. The rest of the early video release is available on my SecureSW. For more information, contact your TI representative.

6.3 Early RVC

Early rear view camera (RVC) is an add-on feature on top of Android SDK and VISION SDK. Both are designed and implemented by utilizing the display sharing between A15 and M4. The early RVC is required to display the capture RVC within 2 seconds and provided robustness, which is achieved as it is running on the Cortex-M4 IPU and DSP. While the early video is required to be displayed within 1-2 seconds after the board is boot up.

On both, the functionality is started at the u-boot level where the display is configured and firmwares are configured on different partitions to allow the early video and the early RVC, which runs on the Cortex-M4 IPU and DSP to start immediately after boot up. The u-boot is also configured for late-attach functionality so that when Android HLOS boots up, it will be able to attach to the firmwares that have already been loaded at u-boot level. At the kernel side, display sharing is the major part of the feature to handle the DSS interrupt for the specific pipes that run on Android as some other DSS pipes are controlled or used by VISION SDK on Cortex-M4. The early video and early RVC use case are designed and implemented using the VISION SDK framework on the Cortex-M4.

For more information about early Video and early RVC, contact your TI local FAE to provide access as both are available through mySecureSoftware.

7 Recommended Userspace Optimizations

Individual custom Android systems have specific abstraction layers, packages, and services that need additional optimizations to achieve a quick boot experience. The recommended optimizations and guidelines discussed in the following sections will help navigate will provide information to do the right thing to satisfy the product needs.

7.1 Class Preloading

Information:

The Android zygote (app launching deamon) preloads classes and starts the package manager after initialization of all the packages on an Android system. The Zygote preloads approximately 3800 classes and instantiates them into heap memory (frameworks/base/preloaded-classes). It is possible to load an Android system without preloading these classes.

Recommendation:

Use proper instrumentation to evaluate all of the Android system use cases and remove pre-loading of non-applicable classes from the Android initialization step.

NOTE: Google started optimizing their core projects from Kitkat+. Custom removal of pre-loaded classes in Android Lollipop+ are not potentially big boot time saving processes anymore.

WARNING

Bad application load times can happen without using the proper instrumentation to identify the target classes to remove. Instrumentation of ALL system use cases must be done before removing any pre-loaded classes.

7.2 Package Manager Scanner

Information:

The Android Package Manager scans, validates certificates, and checks the permissions and dependencies (every time on each package) during boot time.

Recommendation:

Certificate caching (no-revalidation) and optimizing this package manager service to move the non-critical package initialization to a later part of the Android init sequence (using late init triggers), will help in saving boot time.

This can save potentially 30-40 ms, per package, if optimized correctly. (Approximately a total of 2 seconds on the base Marshmallow system.)

WARNING

These optimizations can cause page faults that are hard to debug (due to the usage of mmap'd files).

7.3 Deferred Init/Late triggers/ioprio

Information:

Use available Android triggers, properties (for example, property:sys.boot_completed), and prioritization of the kernel module loading with ‘ioprio’, in order to defer non-core services and drivers for better boot times.

Recommendations:

Use Android triggers (early-init, init, early-fs,fs, post-fs, early-boot, boot) efficiently in moving the non-critical services (or even removing non-custom system services), to a later part of the boot sequence.

This property can be used to defer services to the Android boot completion (property:sys.boot_completed=1).

Make all possible kernel drivers modular and use the ‘ioprio’ value up to 8 (for example, ‘ioprio be 7’ in init.’board’.rc) to defer modules to as late as possible from the Android init process.

7.4 Filesystem Format Analysis

Information:

Study the behavior of Android OS on difference filesystem configurations like EXT4, F2FS, NILFS, BTRFS, SquashFS.

In Conclusion:

- F2FS/EXT4FS works better for smaller file accesses and not recommended to show improvement in infotainment scenario
- NILFS shows better formance on SQLLite operations and can be useful for continuous log recording and power backed systems. Does not show boot improvement.
- SquashFS gives a better I/O read but can be used only on ready-only partitions like system.img. Much of squashFS improvements are seen on Android Oreo hence it is not good to enable squash for boot or size optimization on Marshmallow.
- BTRFS works well in a system with Large filesystem and Larger RAM size. But, CPU utilization is larger and risk on power failure filesystem recovery.
- Hybrid of BTRFS for large RW, F2FS/EXT for small RW partitions can improve overall performance.
- Any of these non-default filesystem formats are analysed for aging. This is something important for infotainment scenario to be evaluated before making decision to swith FS format on a specific partition.

7.5 Memory Allocation Analysis

Information:

Memory allocation (malloc) should be fast for faster runtime of application. Various malloc types (nedmalloc, jemalloc, temalloc) implementations are analyzed to review what would be the better recommendation to speed up application launch.

In Conclusion:

- Nedmalloc lib has less space complexity,and wins in speed when the number of threads are less. Nedmalloc can be a better choice in low end embedded systems where low weight (number of threads) applications are expected to be run most of the time.
- Tcmalloc is faster than nedmalloc and natice android malloc. Does not show much improvement with standard android memory benchmarking applications though. Tcmalloc can be a good choice for high weight (more threads) applications are run most of the time
- Musl-malloc has limitations overall with android single trheaded applications.
- Tlsf-malloc has better size optimizations compared to tcmalloc or android standard malloc and works better on single threaded and frequent application execution.
- Dlmalloc was default for android bionic but got recently replaced to jemalloc. Shows large static size benefit over tcmalloc and jemalloc and very efficient at frequenct malloc, free for multithread applications.

- Jemalloc-svelte configuration is recently added to android and shows improvement over jemalloc. This configuration though shows less runtime performance over standard memory benchmark execution.

tcmalloc, jemalloc are better choices in multi-threaded application and run time performance.

8 Future Work

This section describes the optimizations that is being targeted for Android boot time savings (for Android-Marshmallow with the 4.4 Kernel):

- All of the previous optimizations from Section 7 will be re-evaluated on the Android Oreo with 4.4 Kernel.
- Evaluate faster image compression ZlibVs LZ4 for boot time improvements and various software stack readiness.
- Review Google's Android Oreo Boot time optimization recommendations and implement platform specific fixes accordingly on TI Product SDK: https://source.android.com/devices/tech/perf/boot-times.
- Android Oreo with treble consolidation including driver modularization needs to be evaluated with early mount requirements for boot time improvements and measurements: https://source.android.com/devices/architecture/kernel/modular-kernels.
- Usage of **Squashfs** helps in reducing on-disk size and improves boot time. This can potentially save 8% of the entire Android boot time.
- **LLVM Clang Vs GCC** adds compiler optimizations that can potentially improve boot time if the correct optimization flags (-Oz Vs –O3 Vs –Os) are instrumented.
- **System Image Ram Size Reduction** can potentially save 60-80MB from the system.img footprint from a cached second boot or subsequent boot. This can potentially save boot time as well.
- **Readahead**: Instrumenting 'readahead' with programs like minicore/logsync/treadahead can potentially save 5%-10% boot time by optimizing the Linux page caches, while adjusting the io-sched priority.
- **'Android Hibernation'**: Evaluate Android Hibernation, Suspend-to-disk readiness for all TI Processor SDK usecases. This can potentially save ~40% Boot time

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from Original (January 2016) to A Revision **Page**