Optimizing and Porting the U-Boot bootloader on ARM Cortex

KAVITHA (M.Tech) Embedded Systems, AITS, RAJAMPET, INDIA

ABDUL RAHIM, Professor, ECE, AITS, RAJAMPET, INDIA

kavithareddy.bkreddy@gmail.com

Abstract: ***Some applications have specific requirements for a system's boot time. Often the system does not need to be immediately ready for all its tasks, but it should be ready for certain mission-critical tasks (e.g. accepting commands over Ethernet or displaying a user interface). In this project, we are developing few methodologies and low-hanging fruit for improving boot time on ARM cortex System. Fast boot is essential for consumer devices in automotive, medical and entertainment markets. This paper describes "system level" optimization of embedded software to achieve faster boot times. We select an embedded device running open source Linux platform as the experimental setup for research. First, we describe an efficient bootloader design and explain how to optimally configure Android's Linux based kernel for embedded systems. Next, we detail Linux user space design changes to reach the home screen quickly and allow users to execute crucial applications first. We also discuss effects on memory consumption, application and feature availability caused by optimization changes in each part of the software stack. Finally, we show that our optimized application and kernel stack boots faster than the existing common approach..***

Keywords*: u boot bot loader, Linux, kernel, optimization techniques, ARM cortex.*

I. INTRODUCTION

Linux is the most widely used operating system in the world of embedded system products. Now a days, we can't imagine an embedded system product without Linux OS. Embedded Linux OS has many befits, by using Embedded Linux OS, we can accomplish all the requirements of a real time product, and of course it is an Open source OS, which allows everyone to access the source code and modify it, and use it for learning purpose. Hence the reason, it is very important to think about the boot time taken by the product to get stabilized with first screen or readiness to accept user activities. Boot time is the time taken by the particular system to show the very first screen when the product is powered ON, by pushing the power button. This varies from device to device, For example: to display first image on the screen for the particular devices containing a display e.g. iPad, or any hand held embedded device, media player, cell phone etc

For some applications boot time may be very stringent, and other products may have specific requirements for a system to boot. In some products, the system need not to be immediately ready for all its tasks but it should be ready for certain tasks.

II.NEED FOR BOOTLOADER

A boot loader is a first program that will take care various booting process requirements or checks before actually loading the runtime environment into the memory after completion of the Power ON Self-Tests (POST).

For example, to boot ARM Linux, we need a boot loader, which is a small program that runs before the main kernel. The expectations of boot loader are to initialize various devices, and eventually call the Linux kernel, passing relevant message to the kernel or core of an operating system.

So, the boot loader should meet (as a minimum) the below mentioned requirements:

1. Setup and initialize the RAM.
2. Initialize one serial port.
3. Detect the machine type.
4. Setup the kernel tagged list.
5. Load initramfs.
6. Call the kernel image.

1. Setup and initialize RAM

The boot loader is expected to detect and properly initialize all RAM that the kernel will use for volatile data storage in the system. It performs this in a machine dependent manner.

2. Initialize one serial (COM) port

The boot loader is also responsible to initialize and enable the serial port with desired baud, stop bits, parity bits etc., so that the kernel serial driver to automatically detect which serial port it should use for the kernel console. The initialized serial port can also be used for debugging the kernel or any other application, to understand if anything is blocking or stopped further execution. Also, the bootloader passes the "console = " option to the kernel with serial port configuration details.

3. Detect the machine type

The boot loader should detect the machine type. Whether it is a hard coded or any other algorithm.

The boot loader should always be able to provide a MACH_TYPE_xxx value to the kernel. (see Linux/arch/arm/tools/Mach-types).It should be passed to the kernel in the register r1.the machine type can be determined by using a device tree. the machine type should always set to ones (~0). This is not mandatory but assures that it will never match any existing type.

4. Setup boot data

The boot loader should provide a tagged list or a dtb image for sending configuration data to the kernel. The physical address of the boot data is passed to the kernel in register r2.

4a. Setup the kernel tagged list

The boot loader should create and initialize the kernel tagged list. A tagged list must always start with ATAG_CORE and it should end with ATAG_NONE. The ATAG_CORE tag may or may not be empty. An empty tag has the size field is '2' (0x00000002). To set the size field to zero ATAG_NONE should be used. In the list any number of tags can be placed .it is undefined whether a repeated tag appends to the information which is carried by the previous tag. Few tags behave as the latter and few tags as former.

The boot loader must pass at a minimum size and location of the system memory, and root filesystem location. Therefore, the minimum tagged list should be as follows:

```
        +-----------+
base -> | ATAG_CORE |   |
        +-----------+   |
        | ATAG_MEM  |   | increasing
address
        +-----------+   |
        | ATAG_NONE |   |
        +-----------+   v
```

4b. Setup the device tree

The device tree image should load into the system Ram at a 64bit address and also it should initialize with the boot data. The kernel will look for the dtb magic value of 0xd00dfeed at the dtb physical address to determine whether a dtb has been passed instead of a tagged list.

The boot loader should pass at a minimum size and location of the system memory and the root filesystem location. A safe location is present just above the 128MiB from starting of RAM.

5. Load initramfs.

If an initramfs is in use with the dtb, then it must be placed in a region of memory where the kernel decompressor will not overwrite it and also with the region which will be covered by the low-memory mapping. A safe location is present above the device tree blob which itself will be loaded just above the 128MiB boundary from the start of RAM.

6. Calling the kernel image

For calling the kernel zImage there are two options.

1. It is said to be legal for the boot loader to call the zimage in flash directly only when the zimage is stored in flash and then it should be linked to be run from flash

2. The zImage may also be placed in system RAM and called there. The kernel should be placed in the first 128MiB of RAM. It is recommended that it is loaded above 32MiB in order to avoid the need to relocate

So that it will make the boot process slightly faster. while booting a non-Zimage the constraints are tighter. For this case the kernel should be loaded at an offset from system equal to TEXT_OFFSET – PAGE_OFFSET

In any case, it should meet the following conditions:

- Quiesce all DMA capable devices so that memory does not get corrupted by bogus network packets or disk data. This will save you many hours of debug.

- CPU register settings

r0 = 0,

r1 = machine type number.

r2 = physical address of device tree block (dtb) in system RAM or physical

address of tagged list in system RAM

- CPU mode

All forms of interrupts must be disabled (IRQs and FIQs) For CPUs which do not include the ARM virtualization extensions, the CPU must be in SVC mode. CPUs which include support for the virtualization extensions can be entered in HYP mode in order to enable the kernel to make full use of these extensions. This is the recommended boot method for such CPUs, for any reason if the kernel is not entered in HYP mode it must be entered in SVC mode.

- Caches, MMUS

Instruction cache may be on or off but the MMU and Data cache must be off. If the kernel is entered in HYP mode, the above requirements apply to the HYP mode configuration in addition to the ordinary PL1 (privileged kernel modes) configuration. In addition, all traps into the hypervisor must be disabled, and PL1 access must be granted for all peripherals and CPU

resources for which this is architecturally possible. Except for entering in HYP mode, the system configuration should be such that a kernel which does not include support for the virtualization extensions can boot correctly without extra help.

- The boot loader is expected to call the kernel image by jumping directly to the first instruction of the kernel image. On CPUs supporting the ARM instruction set, the entry must be made in ARM state, even for a Thumb-2 kernel. On CPUs supporting only the Thumb instruction set such as Cortex-M class CPUs, the entry must be made in Thumb state.

III HARDWARE AND SOFTWARE PLATFORM

This section describes the ARM hardware and software background used for this project. While the first section introduces the hardware details and the second part is focused on the software platform.

A. ARM Cortex Board

ARM development boards are proven platforms for accelerating the development and reducing the risk of new SoC designs. The technology in ARM boards delivers an optimal solution in terms of speed, accuracy, flexibility and cost.

As a hardware platform for this project, ARM Cortex architecture based board, a particular variant of the Samsung platform has been chosen.

The small and support of many peripherals makes it an ideal object for this project. The main processing core is a powerful Cortex-A9 clock runs at 1 GHz which complies with the ARM Architecture. The selected Samsung S33C chip is a system-on-chip (SoC) microprocessor has a variety of typical embedded system, like a UART, SPI, real-time clock (RTC, 5.5" LCD Display sub-system, and Audio back-end (ABE) sub-system. The device also integrates On-chip memory; External memory interfaces System and connecting peripherals such as on-board Ethernet, LCD display controller, Zig-bee, Bluetooth, HDMI and DVI ports. In addition to these on-chip components, it has 1 GB low power DDR2 RAM and General purpose expansion header (I2C, GPMC, USB, MMC, DSS and ETM).

B. Software Platform

The software platform for this project was built with the help of the build-root and u Boot Bool loader. This includes everything needed to work with a Linux based computer system. It has GCC-compiler, boot-loaders, kernel, and most useful additional libraries and useful tools like busy-box etc. We can also build manually by using make utility tool. First download the general source coder for boot-loaders and kernel and then cross-compile it to generate boot-loader images

IV. MEASUREMENT AND OPTIMIZATION

A. Initial Measurement:

Boot time Optimization can be measured by knowing current boot-time, setting the target and defining the boundary conditions. First we need to quantify the problem. We could use a stop-watch. But, that is not very accurate and rather tedious after the first few runs. A better solution is to instrument the code or to monitor the boot from outside. We will describe both techniques, starting with external monitoring.

The overall boot process involves boot-loader(s), Linux kernel and the file system. We must identify the time stampings markers in the boot logs that can be used as delimiters for each stage of the boot process. This helps in finding the time spent in each stage of the boot process.

B. Optimization:

Areas of optimization classified into two categories:

1. Size optimization
   - Reduce the size of binaries for each successive component loaded.
   - Remove features that are not required.
2. Speed Optimization

Optimize for target processor.

- Use faster medium for loading primary, secondary boot loaders and kernel.
- Reduce number of tasks leading to the boot, as many as possible.
- List and remove features that are not required.

Boot time is affected by different factors such as hardware, boot loader configurations, kernel configuration and application profile. We will discuss various Boot time reduction techniques which will be used for optimizing Linux in different steps of booting such as boot loader, Kernel loading, User-space application initialization and so on.

III. RESULTS

After whole optimization:

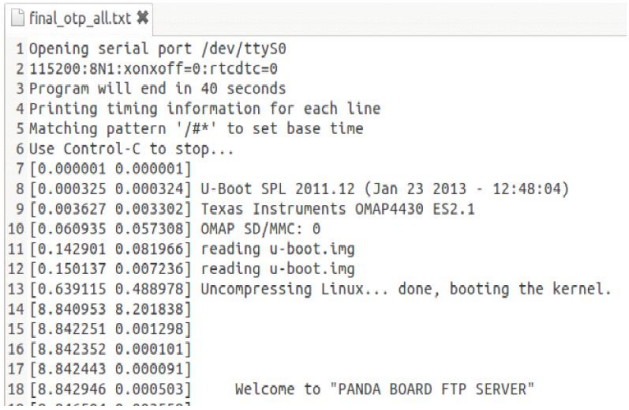

Fig: Boot time optimization.

IV. CONCLUSION

Based on detailed experiments on U-Boot bootloader on ARM cortex architecture based processor, we conclude that Reduction in code size and Boot Time depends on various approaches in reducing the code size which we had used in practical use cases. There are still enough windows where we can even reduce more booting time by applying more techniques which will use on different platform. So, finally we say that we were able to demonstrate the Reduction in Boot Time and code size is achieved for a particular architecture, in our case, it is ARM cortex.

ACKNOWLEDGMENT

We thank to our project guide, Mr. Abdul Rahim and, for providing necessary facilities towards carrying out this work. We are also very thankful to Mr S. Narayana raju, Embedded System Design Specialist, for all the support in successfully completing this project. We are also very much thankful to our project co-ordinator and entire ECE department faculty in giving the freedom in choosing this project, for continuous support and encouragement.

REFERENCES

[1] http://www.comptechdoc.org/os/linux/howlinuxworks/linux_hlbootproc.html.
[2] http://www.linuxinsight.com/proc_uptime.html
[3] http://planet.linaro.org/tag/boot%20time/
[4] http://www.comptechdoc.org/os/linux/howlinuxworks/linux_hlbootproc.html
[5] http://www.omappedia.com/wiki/4AI.1.4_OMAP4_Icecream_Sandwich_Panda_Notes
[6] http://www.thegeekstuff.com/2011/02/linux-boot-process/
[7] http://www.ibm.com/developerworks/linux/library/l-linuxboot/index.html
[8] http://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/ch09s04.html
[9] http://elinux.org/Boot_Time
[10] http://processors.wiki.ti.com/index.php/Optimize_Linux_Boot_Time
[11] http://www.linuxhomenetworking.com/wiki/index.php/Quick_HOWTO_:_Ch15_:_Linux_FTP_Server_Setup#.UZ8F0loW2li
[12] http://free-electrons.com/pub/conferences/2011/genivi/boot-time.pdf
[13] https://www.toradex.com/blog/embedded-linux-boot-time-optimization
[14] http://processors.wiki.ti.com/index.php/Optimize_Linux_Boot_Time

BIOGRAPHY

Sri. KAVITHA, studying Mtech in Embedded Systems, at AITS, Rajampet, INDIA. Currently she is undergoing internship training at EmWare Technologies (INDIA) Pvt Ltd, at Bangalore center. She is expert in embedded systems peripheral drivers and 8/16/32 architectures etc.

Sri Abdul Rahim, Professor - He is having vast experience of Industry as well as Academics. He presented several papers in International and National Conferences and journals.