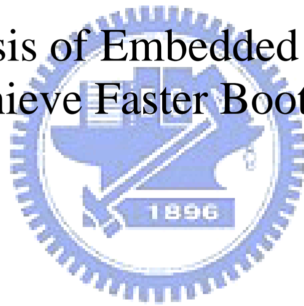
# 國立交通大學

## 電機與控制工程研究所

## 碩士論文

# 利用實驗分析加速嵌入式 Linux 2.6.14 核心的開機時間

# An Empirical Analysis of Embedded Linux Kernel 2.6.14 to Achieve Faster Boot Time




研 究 生：楊志堅

指導教授：黃育綸　博士

中 華 民 國 九 十 五 年 七 月

# 利用實驗分析加速嵌入式 Linux 2.6.14 核心的開機時間

# An Empirical Analysis of Embedded Linux Kernel 2.6.14 to Achieve Faster Boot Time


研　究　生：楊志堅　　　　Student：Chih-Chien Yang

指導教授：黃育綸 博士　　　　Advisor：Dr. Yu-Lun Huang


國 立 交 通 大 學

電 機 與 控 制 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Degree of Electrical Engineering and Control Engineering

College of Electrical and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of Master

in

Electrical and Control Engineering

July 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年七月

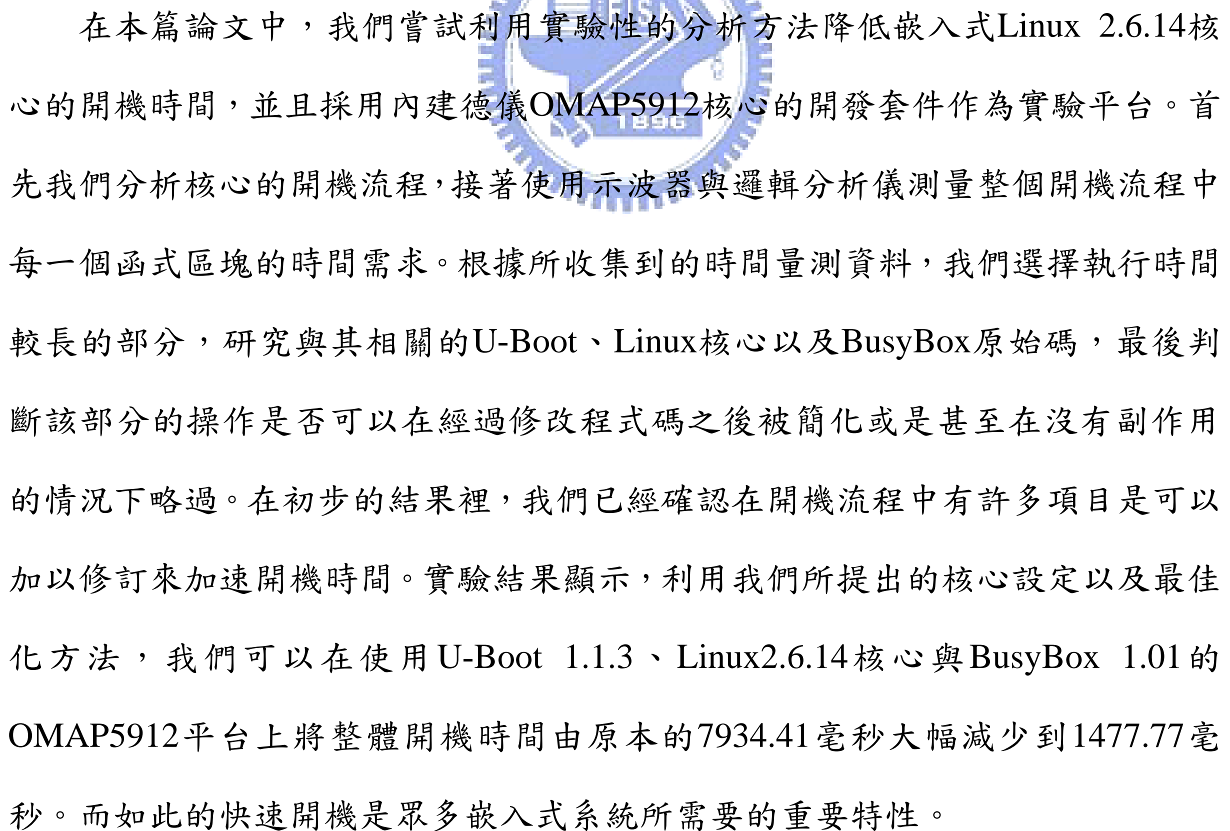
# 利用實驗分析加速嵌入式Linux 2.6.14核心的開機時間

研 究 生：楊志堅　　　　　　　　　指導教授：黃育綸 博士

國 立 交 通 大 學

電 機 與 控 制 工 程 研 究 所

## 摘要



在本篇論文中，我們嘗試利用實驗性的分析方法降低嵌入式Linux 2.6.14核心的開機時間，並且採用內建德儀OMAP5912核心的開發套件作為實驗平台。首先我們分析核心的開機流程，接著使用示波器與邏輯分析儀測量整個開機流程中每一個函式區塊的時間需求。根據所收集到的時間量測資料，我們選擇執行時間較長的部分，研究與其相關的U-Boot、Linux核心以及BusyBox原始碼，最後判斷該部分的操作是否可以在經過修改程式碼之後被簡化或是甚至在沒有副作用的情況下略過。在初步的結果裡，我們已經確認在開機流程中有許多項目是可以加以修訂來加速開機時間。實驗結果顯示，利用我們所提出的核心設定以及最佳化方法，我們可以在使用U-Boot 1.1.3、Linux2.6.14核心與BusyBox 1.01的OMAP5912平台上將整體開機時間由原本的7934.41毫秒大幅減少到1477.77毫秒。而如此的快速開機是眾多嵌入式系統所需要的重要特性。

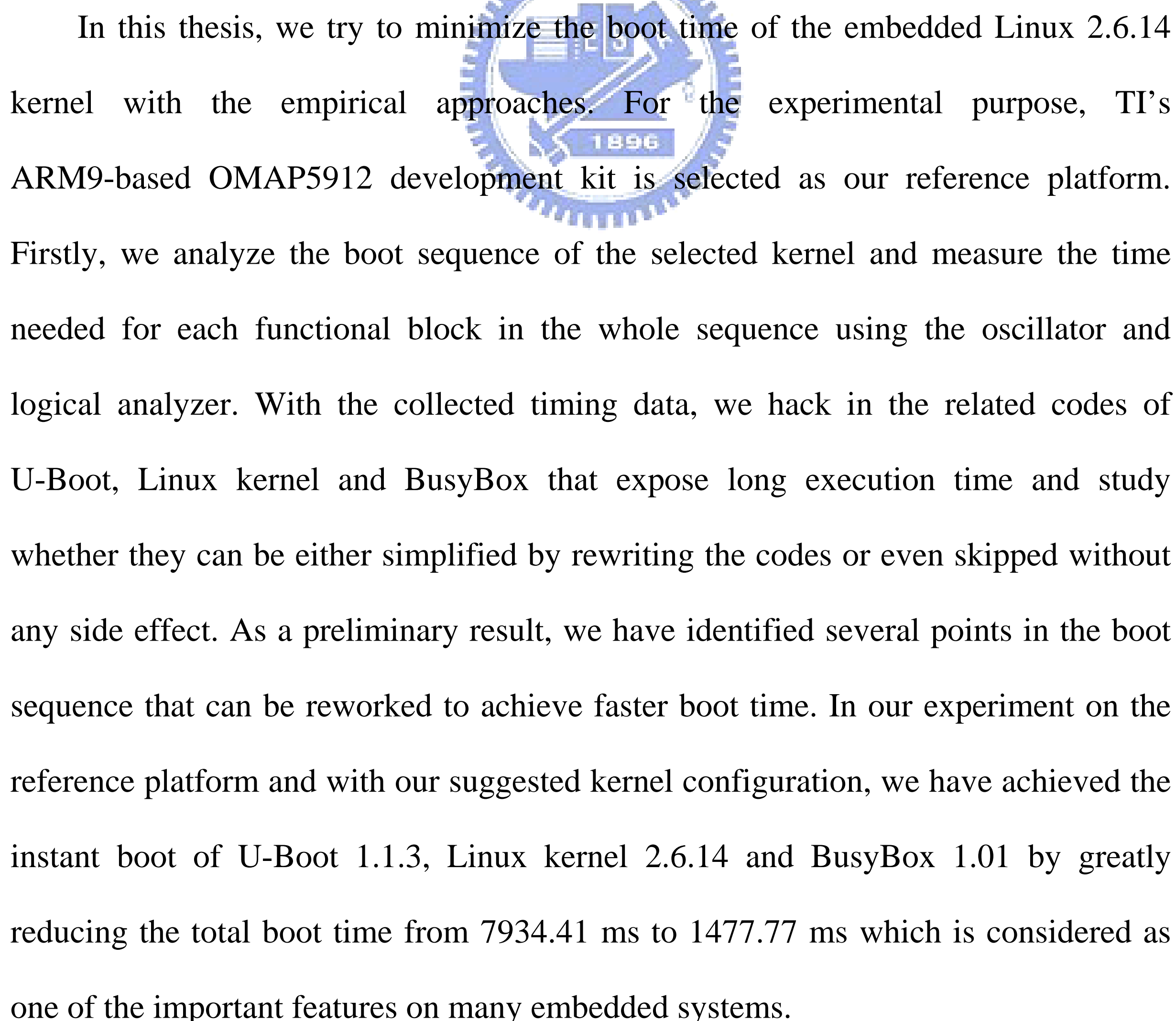
# An Empirical Analysis of Embedded Linux Kernel 2.6.14 to Achieve Faster Boot Time

Student: Chih-Chien Yang　　　Advisor: Dr. Yu-Lun Huang

Department of Electrical and Control Engineering

National Chiao-Tung University

## Abstract



In this thesis, we try to minimize the boot time of the embedded Linux 2.6.14 kernel with the empirical approaches. For the experimental purpose, TI's ARM9-based OMAP5912 development kit is selected as our reference platform. Firstly, we analyze the boot sequence of the selected kernel and measure the time needed for each functional block in the whole sequence using the oscillator and logical analyzer. With the collected timing data, we hack in the related codes of U-Boot, Linux kernel and BusyBox that expose long execution time and study whether they can be either simplified by rewriting the codes or even skipped without any side effect. As a preliminary result, we have identified several points in the boot sequence that can be reworked to achieve faster boot time. In our experiment on the reference platform and with our suggested kernel configuration, we have achieved the instant boot of U-Boot 1.1.3, Linux kernel 2.6.14 and BusyBox 1.01 by greatly reducing the total boot time from 7934.41 ms to 1477.77 ms which is considered as one of the important features on many embedded systems.

# 致謝

首先由衷感激我的指導教授黃育綸老師，在這兩年的研究生生涯裡，總是熱心而絲毫不厭倦的不斷的引導我如何做研究；開導我如何從千頭萬緒之中找出方向；建議我如何將大筆雜亂無章的資料收為己用；教導我如何吸收他人的新觀念並找出加以改進的地方；研究之外，也細心的關懷我脫離情緒的低潮。老師的呵護，讓我如沐春風，在短短的兩年內持續進步，有所成長，終於能寫出自己的論文，成為一位合格的研究生。也感謝口試委員：胡竹生老師與何福軒博士撥冗對我的論文提出建議，讓我的論文更加完整。

其次我也非常感謝即時嵌入式系統實驗室的夥伴們：建銘和精佑對於我的研究提供了很多幫助；嘉良、瀞瑩、欣宜、依文、勁源、恩捷、Newlin、興龍、立穎以及培華充實了我的研究生生活，也讓實驗室天天充滿了歡笑。希望大家都能找到自己的方向，順利的完成學業。

也謝謝我的女朋友猴兒，願意犧牲相處的時間，進而支持鼓勵我專心的學習。最後要感謝我親愛的家人，雖然總是讓他們擔心，但仍然能夠默默的包容我，僅以此論文獻給我最愛的家人。

# Table of Content


**摘要 ........ i**

**Abstract ........ ii**

**致謝 ........ iii**

**Table of Content ........ iv**

**List of Figures ........ vi**

**List of Tables ........ vii**

**Chapter 1 Introduction ........ 1**

1.1 Motivation ........ 1

1.2 Contribution ........ 2

1.3 Synopsis ........ 2

**Chapter 2 Related Work ........ 3**

2.1 Snapshot Technique for NOR Flash ........ 3

2.2 Erase Block Summary ........ 4

2.3 Kernel Execute-In-Place ........ 5

2.4 InitNG ........ 5

2.5 Summary ........ 6

**Chapter 3 Background ........ 7**

3.1 OSK5912 OMAP Starter Kit ........ 7

3.2 U-Boot ........ 9

3.3 Embedded Linux ........ 10

3.4 BusyBox ........ 11

3.5 Summary ........ 11

**Chapter 4 Boot Time Analysis ........ 12**

4.1 Boot Sequence ........ 12

4.2 Boot Time Measurement Tools ........ 13

4.2.1 Kernel Function Trace ........ 13

4.2.2 Printk Times ........ 15

4.2.3 initcall-times patch......15
4.2.4 Expect......16
4.3 Measurement Tools Analysis......17
4.3.1 Kernel Function Trace......17
4.3.2 Printk Times......18
4.3.3 initcall-times patch......18
4.3.4 Expect......19
4.4 Boot Time Measurement......19
4.5 Boot Sequence Analysis......22
4.5.1 Level I: From the Application's Point of View......23
4.5.2 Level II: From the Functional Block's Point of View......23
4.5.3 Measurement result......26
4.6 Summary......27
**Chapter 5 Experiment......28**
5.1 Redundancy Analysis......28
5.2 Comparison......35
5.3 Reduced Functional Ability......49
5.4 Recommendation......50
5.5 Summary......51
**Chapter 6 Conclusion......52**
**Chapter 7 Future Work......53**
**Reference......54**

# List of Figures


Figure 2.1 Snapshot Management of Snapshot Technique for NOR Flash ............4
Figure 4.1 Numeric Trace Data of KFT ............14
Figure 4.2 Symbolic Trace Date of KFT ............14
Figure 4.3 Kernel Routines Date in Nested............15
Figure 4.4 Initcall Log in Kernel Ring Buffer............16
Figure 4.5 The Timestamp Resetting ............16
Figure 4.6 No Timestamp before Some Messages............18
Figure 4.7 Device Reset Timing of OMAP5912............20
Figure 4.8 The Movie Frame of Finish Uncompressing Kernel ............21
Figure 4.9 The Movie Frame of Finish Displaying Linux Banner............21
Figure 4.10 The Records of Logic Analyzer ............22
Figure 5.1 The Patch of Changing Clocking Mode............37
Figure 5.2 The Patch of Simplifying *abortboot* ............38
Figure 5.3 The Patch of Verification Switch............40
Figure 5.4 The Sample Patch of Silent Console............41
Figure 5.5 The Patch of Uncompressed Kernel ............43
Figure 5.6 The Effect of Preset LPJ ............43
Figure 5.7 The Patch of Quick Shell Prompt ............47
Figure 5.8 The NOR Flash Memory Map ............48
Figure 5.9 The Mount Operation of JFFS2 Partition in Background............48

# List of Tables

Table 4.1 KFT Activates from *start_kernel* to *to_userspace* .............................. 17
Table 4.2 The 18 functional blocks .............................. 23
Table 4.3 A Template of Boot Time Measurement Table .............................. 25
Table 4.4 Boot Time with 18 Function Blocks .............................. 26
Table 5.1 The Time Comparison of Changing Clocking Mode .............................. 36
Table 5.2 The Time Reduced by Skipping *console_init_r* .............................. 37
Table 5.3 The Time Reduced by Simplifying *abortboot* .............................. 38
Table 5.4 The Time Reduced by Verification Switch .............................. 39
Table 5.5 The Time Reduced by Silent Console in U-Boot .............................. 40
Table 5.6 The Time Reduced by Uncompressed Kernel .............................. 41
Table 5.7 The Time of *calibrate_delay* .............................. 43
Table 5.8 The Time of Initcalls .............................. 44
Table 5.9 The Time Reduced by Silent Console in Linux kernel .............................. 45
Table 5.10 The Comparison between Different FS .............................. 47
Table 5.11 Functional Ability Comparison .............................. 49

# Chapter 1

# Introduction

## 1.1 Motivation

With the development and popularity of the mobile device and high-level consumer electronics, there are more and more applications of embedded Linux operating system on them. Boot is the first impression of an electronic product for consumers; therefore the boot time should not be too long to give consumers a good impression. However, the boot time of general embedded Linux operating system on the market is about 8-10 seconds on average at present and most of consumer electronics use Linux kernel 2.4 as the embedded Linux operating system. But with the development of Linux kernel 2.6, using Linux kernel 2.6 as the embedded Linux operation system will be a trend certainly in the future. In order to prevent users from having bad impressions, the developer let product show boot logo using extra graphic chip on screen first during the core processor doing boot. Therefore if we can provide fast boot mechanism of Linux kernel 2.6 and enable users to do the first operation of the product within a shortest time, users will accept this product even more, developers can save the extra hardware cost (the graphic chip) and the product will meet the requirements in the future even more. In addition, we can save more valuable time in critical reboot operating, if boot is faster.

## 1.2 Contribution

At first, we propose the method which measure the exact time needed for each specific function, even for specific instruction in the whole boot sequence using the oscillator, logical analyzer and other assistant records and information.

Secondly, with our optimized U-Boot 1.1.3, suggested Linux kernel 2.6.14 configuration, and optimized BusyBox 1.01, we have achieved the instant boot on the OMAP5912OSK by greatly reducing the total boot time from 10062.94 ms to 1477.77 ms which is considered as one of the important features on many embedded systems. And the optimization methods of U-Boot 1.1.3 and BusyBox 1.01 are also suitable for other platforms, not only on the OMAP5912OSK.

## 1.3 Synopsis

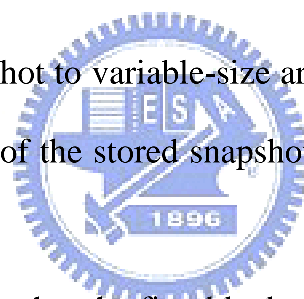
This thesis is organized as follows. In Chapter 2 and Chapter 3, related work and background are surveyed. In Chapter 4, we analyze the issue of boot time measurement tools, use complex tools and oscillator and logical analyzer to measure boot time, and analyze the measurement result to find long execution time operations. In Chapter 5, we implement experiments to optimize the lone execution time operations in Chapter4 to achieve faster boot time. Finally, we the conclusions and further work are given in Chapter 6.

# Chapter 2

# Related Work

There are many exist techniques to improve the boot time of Linux. They improve different parts of full boot process separately. They include the different file system structure of flash storage device [1] [2], the special method to execute kernel [3] [10] and the process control initialization utility [4].

## 2.1 Snapshot Technique for NOR Flash



This technique stores snapshot to variable-size areas managed by linked lists and sequentially record the location of the stored snapshots to prearranged areas by using an ordered tree data structure.

In Figure 2-1, it can be seen that the first block of flash memory is reserved as a root block which sequentially stores pointers to snapshot header blocks. During the *mount_root* operation, the last stored pointer can be found quickly using sequential or binary search algorithms. The binary searching divides the root block into two sub-blocks and reads the boundary pointer of these sub-blocks. If the pointer is null, this searching selects the left sub-block. Otherwise, the other one is selected. With the selected sub-block, the above procedure is repeated until the last stored pointer is found. Since the block size ($B_{size}$) is typically 128KB in NOR flash and the size of a pointer to block is 2B ($P_{size}$), this search algorithm has a better time complexity of $O(lg(B_{size} / P_{size})) = O(16)$.

In summary, this technique only reads $lg(B_{size} / P_{size})$ x $P_{size}$ + $lg(B_{size} / H_{size})$ x

$H_{size}$ (=92) bytes in an average case to fine the location of the last stored snapshot, providing an instant lookup time.

However, the author doesn't release the source code. Therefore we can't try using this technique.

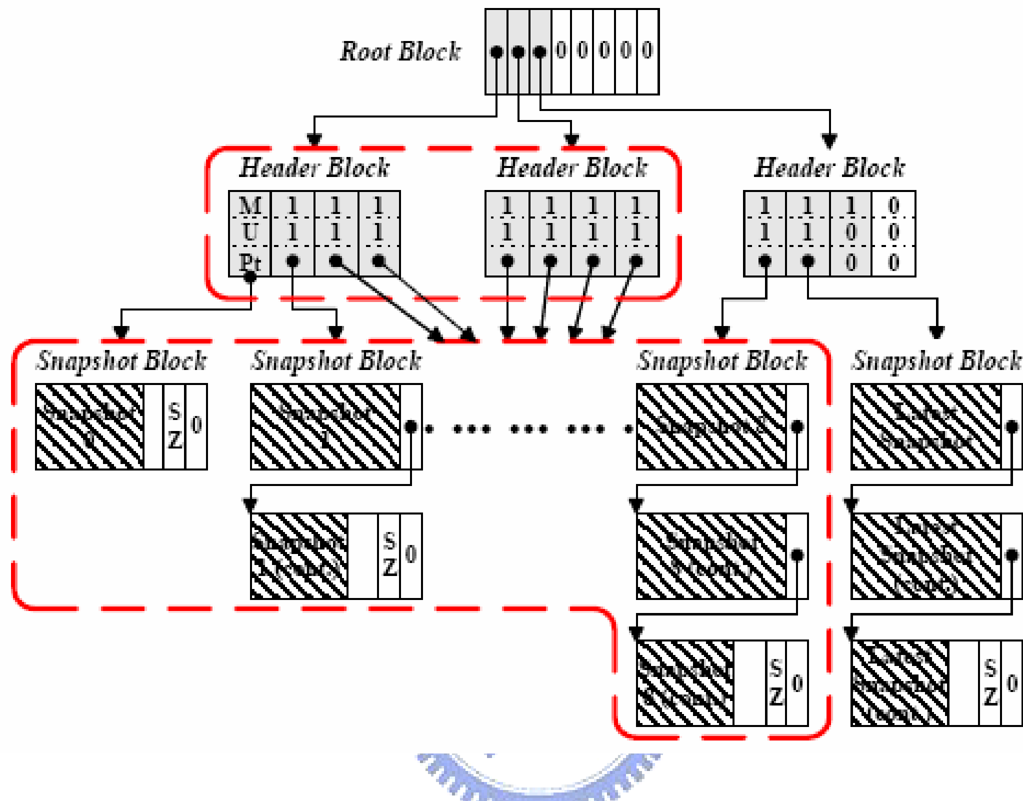



Figure 2.1 Snapshot Management of Snapshot Technique for NOR Flash

## 2.2 Erase Block Summary

Erase Block Summary (EBS) is an improvement to speed up the mount process. EBS stores extra summary information at the end of every (closed) erase block. This information is generated automatically at file system write operations. To make it possible to determine the size of the summary node, there is an 8 byte long summary marker node (*jffs2_sum_marker*) at the end of erase blocks. At mount time *jffs2_scan_eraseblock*() reads the last 8 bytes of the erase block during the scan process. If it finds valid *sum_marker* node, it loads the summary node pointed by the

relative offset stored in *sum_marker*. All information needed at mount time is stored in this node, so scanning the full erase block is not necessary. It can cause a big speedup, especially at NAND devices. If *sum_marker* is not found (or invalid) the normal scan process will be applied.

Known the EBS is only existent in JFFS2 image. That is to say, EBS is only existent in the parts of used space and not existent in the parts of unused space. Therefore, the effect of EBS is limited.

## 2.3 Kernel Execute-In-Place

Execute-In-Place (XIP) allows the kernel run from non-volatile storage directly addressable by the CPU, such as NOR flash. This saves RAM space since the text section of the kernel is not loaded from flash to RAM. Read-write sections, such as the data section and stack, are still copied to RAM. The XIP kernel is not compressed since it has to run directly from flash, so it will take more space to store it. The flash address used to link the kernel object files, and for storing it, is configuration dependent. Therefore, the proper physical address where to store the kernel image depending on specific flash memory usage must be known.

For OMAP-based platform, Kernel XIP is only effective on OMAP Innovator using rrload. Now, Kernel XIP still not support by U-Boot on ARM-based platform.

## 2.4 InitNG

InitNG is a full replacement of the old and in many ways deprecated sysvinit tool (*init*) created by Jimmy Wennlund. It is designed to significantly increase the speed of

booting a UNIX-compatible system by starting processes asynchronously. On boot, *initng* will be invoked as the first process (pid = 1) by the kernel. At first, *initng* will parse configuration files located in */etc/initng* for critical information such as runlevel and service data. After that, all services required by the default runlevel will be started as soon as their dependencies are met, allowing services to run in parallel. This asynchronous execution can dramatically improve boot time by better utilizing the system resources (especially in the case of multiprocessor systems).

The last version of InitNG is 0.6.7, which still not support for ARM-based platform.

## 2.5 Summary

In Chapter 2, we introduce many new techniques for improve the boot time. Some techniques are only absorbed in PPC; some techniques are still not ported to OMAP-based platform and others techniques are only working on specific peripheral and application.

# Chapter 3

# Background

On the market, the choices of hardware and software for development of mobile device and high-level consumer electronics are very many. To choose a good combination for product which is suitable for the function requirement and high return on investment is the most important.

## 3.1 OSK5912 OMAP Starter Kit

The OMAP 5912 multiprocessor platform is available in the OSK5912 OMAP Starter kit by Spectrum Digital. The dual-core architecture provides benefits of both DSP and reduced instruction set computer (RISC) technologies [5].

The MPU core is the ARM926EJ-S reduced instruction set computer (RISC) processor. The ARM926EJ-S is a 32-bit processor core that performs 32-bit or 16-bit instructions and processes 32-bit, 16-bit, or 8-bit data. The core uses pipelining so that all parts of the processor and memory system can operate continuously. The MPU core also incorporates the data and program memory management units (MMUs) with table look-aside buffers. To minimize external memory access time, the ARM926EJ-S includes an instruction cache, data cache, and a write buffer. In general, these are transparent to program execution.

The DSP core of the OMAP5912 device is based on the TMS320C55x DSP generation CPU processor core. The C55x DSP architecture achieves high performance and low power through increased parallelism and total focus on

reduction in power dissipation. The CPU supports an internal bus structure composed of one program bus, three data read buses, two data write buses, and additional buses dedicated to peripheral and DMA activity. These buses provide the ability to perform up to three data reads and two data writes in a single cycle. In parallel, the DMA controller can perform up to two data transfers per cycle independent of the CPU activity. A central 40-bit arithmetic/logic unit (ALU) is supported by an additional 16-bit ALU. Using of the ALU provides the ability to optimize parallel activity and power consumption. The OMAP5912 DSP core also includes a 24K-byte instruction cache to minimize external memory accesses, improving data throughput and conserving system power.

The TMS320C55x DSP core within the OMAP5912 device utilizes three powerful hardware accelerator modules which assist the DSP core in implementing specific algorithms that are commonly used in video compression applications such as MPEG4 encoders/decoders. They are DCT/iDCT Accelerator, Motion Estimation Accelerator and Pixel Interpolation Accelerator. These accelerators allow implementation of such algorithms using fewer DSP instruction cycles and dissipating less power than implementations using only the DSP core. The hardware accelerators are utilized via functions from the TMS320C55x Image/Video Processing Library available from Texas Instruments.

The OMAP5912OSK platform also provides rich user interfaces, high processing performance, and long battery life through the maximum flexibility of a fully integrated mixed processor solution. Therefore, the OMAP5912OSK could meet of requirement of following applications:

- Applications Processing Devices
- Mobile Communications
    - WAN 802.11X

- Bluetooth
- GSM, GPRS, EDGE
- CDMA

- Video and Image Processing (MPEG4, JPEG, Windows Media Video, etc.)
- Advanced Speech Applications (text-to-speech, speech recognition)
- Audio Processing (MPEG-1 Audio Layer3 [MP3], AMR, WMA, AAC, and Other GSM Speech Codecs)
- Graphics and Video Acceleration
- Generalized Web Access
- Data Processing

For the diversified features and applications, we choose OMAP5912OSK as our development platform.

## 3.2 U-Boot

In an embedded system the role of the boot loader is more complicated since these systems do not have BIOS to perform the initial system configuration. The low level initialization of microprocessors, memory controllers, and other board specific hardware must be performed before a Linux kernel image can execute. At a minimum an embedded loader provides the following features:

1. Initializing the hardware, especially the memory controller.
2. Providing boot parameters for the Linux kernel.
3. Starting the Linux kernel.

Additionally, most boot loaders also provide convenience features that simplify development:

1. Reading and writing arbitrary memory locations.
2. Uploading new binary images to the board's RAM via a serial line or Ethernet.
3. Copying binary images from RAM to FLASH memory.

Das U-Boot is a GPL'ed cross-platform boot loader shepherded by Wolfgang Denk [6] and provides the full functions of above-mentioned requirement. It also provides out-of-the-box support for hundreds of embedded boards and a wide variety of CPUs including PowerPC, ARM, XScale, MIPS, Coldfire, NIOS, Microblaze, and x86. The easy configuration of U-Boot strikes the right balance between a rich feature set and a small binary footprint. Therefore, U-Boot 1.1.3 is the best choice of the boot loader on our implementation platform, and supports for Linux kernel 2.6.

## 3.3 Embedded Linux

There are many embedded operating system, which are designed to be very compact and efficient, forsaking many functionalities that non-embedded computer operating systems provide and which may not be used by the specialized applications they run. Embedded operating systems include: eCos, Embedded Linux, FreeDOS, FreeRTOS, LynxOS RTOS, NetBSD, OpenBSD, Inferno, OSE, OS-9, QNX, VxWorks, Windows CE and Windows XP Embedded…etc. Among them, Embedded Linux refers to the use of the open source Linux operating system in embedded systems such as cell phones, PDAs, media player handsets, and other consumer electronics devices.

In the past an embedded development was mostly performed using proprietary code written in assembler. Developers had to write all of the hardware drivers and interfaces from scratch. It appeared that the Linux kernel, combined with a small set

of other free software utilities, could be fit into the confines of the limited hardware space of an embedded device. And a typical installation of embedded Linux takes about 2 megabytes. Therefore, we use the embedded Linux kernel 2.6.14 (linux-2.6.14-omap2) [7] [8] as our embedded operating system.

## 3.4 BusyBox

BusyBox [9] combines tiny versions of many common UNIX utilities into a single small executable. It provides replacements for most of the utilities in GNU, which are archival utilities, coreutils, console utilities, editors, finding utilities and init utilities…etc. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins; however, the options that are included provide the expected functionality and behave very much like their GNU counterparts. BusyBox provides a fairly complete environment for any small or embedded system.

BusyBox has been written with size-optimization and limited resources in mind. It is also extremely modular so including or excluding commands (or features) is easy at compile time. This makes it easy to customize specific embedded systems. To create a working system, developers just need to add some device nodes in */dev*, a few configuration files in */etc*, and a Linux kernel. We use BusyBox 1.01 to replace the original big file system of PC running Linux.

## 3.5 Summary

In Chapter 3, we describe the background of our development platform. It includes powerful OMAP5912OSK, universal U-Boot, the open source embedded Linux and tiny BusyBox.

# Chapter 4

# Boot Time Analysis

Before starting reducing the booting time, we should understand the boot sequence first. Then measuring the booting time and analyzing the timing result. Finally, to improve the original embedded operating system as fast booting system.

## 4.1 Boot Sequence

We can summarize the initial boot sequence of Linux kernel as follows [10] [11]:

1. The boot loader arranges for the kernel to be placed at the proper address in memory. This code is external to Linux source code and usually the first code segment executed once the system is powered on. Finally, this boot loader jumps to execute Linux kernel.
2. Architecture-specific assembly code in Linux kernel performs very low-level tasks, such as initializing memory and setting up CPU registers so that C code can run flawlessly. This includes selecting a stack area and setting the stack pointer accordingly. The amount of such code varies from platform to platform; it can range from a few dozen lines up to a few thousand lines.
3. Function *start_kernel* is called. It acquires the kernel lock, prints the banner, and calls function *setup_arch* to configure the system according to the platform's architecture.
4. Architecture-specific C-language code completes low-level initialization,

including interrupt vectors initialization, and retrieves a command line for *start_kernel* to use.

5. *start_kernel* parses the command line and calls the handlers associated with the keyword it identifies.
6. *start_kernel* initializes basic facilities and forks the *init* thread.
7. *init* is the first user space application, it does the process control initialization, runs the initialization script and start daemons. Finally it starts the *getty* processes that put the login prompt.

## 4.2 Boot Time Measurement Tools

The usual way to look at a program is to start where execution begins. As far as Linux is concerned, it's hard to tell where execution begins - it depends on how you define begins. Therefore we need to use some measurement tools to assist us measuring boot time.

### 4.2.1 Kernel Function Trace

Kernel Function Trace (KFT) [10] [12] is a kernel function tracing system, which uses the "*-finstrument-functions*" capability of the *gcc* compiler to add instrumentation callouts to every function entry and exit. The KFT system provides for capturing these callouts and generating a trace of events, with timing details. KFT is excellent at providing a good timing overview of kernel procedures, allowing you to see where time is spent in functions and sub-routines in the kernel.

The STATIC_RUN mode of operation with KFT is doing configuration for a KFT run and is compiled statically into the kernel. This mode is useful for getting a

trace of kernel operation during system boot (before user space is running).

The KFT configuration lets you specify how to automatically start and stop a trace, whether to include interrupts as part of the trace, and whether to filter the trace data by various criteria (for minimum function duration, only certain listed functions, etc.) KFT trace data is retrieved by reading from */proc/kft_data* after the trace is complete.

| Entry | Delta | Function | Caller |
|---|---|---|---|
| 0 | -1 | 0xc002307c | 0xc00231b4 |
| 0 | -1 | 0xc002935c | 0xc00230b4 |
| 0 | 468750 | 0xc0099bc4 | 0xc00293ac |
| 0 | 312500 | 0xc0098c48 | 0xc0099c20 |
| 0 | 312500 | 0xc009dfe0 | 0xc0098c88 |
| 0 | 312500 | 0xc009db18 | 0xc009e1d4 |
| 0 | 312500 | 0xc009cb34 | 0xc009dbdc |
| 0 | 7813 | 0xc009ca58 | 0xc009cf0c |
| 0 | 7813 | 0xc009c50c | 0xc009cadc |
| 0 | 7813 | 0xc00cd148 | 0xc009c5ac |
| 0 | 7813 | 0xc00dacf4 | 0xc00cd298 |
| 0 | 7813 | 0xc00d29ec | 0xc00dad58 |

Figure 4.1 Numeric Trace Data of KFT

| Entry | Delta | Function | Caller |
|---|---|---|---|
| 0 | -1 | run_init_process | init+0xb4 |
| 0 | -1 | execve | run_init_process+0x38 |
| 0 | 468750 | do_execve | execve+0x50 |
| 0 | 312500 | open_exec | do_execve+0x5c |
| 0 | 312500 | path_lookup | open_exec+0x40 |
| 0 | 312500 | link_path_walk | path_lookup+0x1f4 |
| 0 | 312500 | __link_path_walk | link_path_walk+0xc4 |
| 0 | 7813 | do_lookup | __link_path_walk+0x3d8 |
| 0 | 7813 | real_lookup | do_lookup+0x84 |
| 0 | 7813 | jffs2_lookup | real_lookup+0xa0 |
| 0 | 7813 | jffs2_read_inode | jffs2_lookup+0x150 |
| 0 | 7813 | jffs2_do_read_inode | jffs2_read_inode+0x64 |

Figure 4.2 Symbolic Trace Date of KFT

KFT supplies two useful log analysis tools: *addr2sym* is supplied to convert numeric trace data (see Figure 4.1) to kernel symbolic trace data (see Figure 4.2), and *kd* is supplied to process and analyze the data in a KFT trace. By using both tools, the log with function name, execution count, amount execution time and average execution time of kernel routines can be produced. In addition, a log with the trace of

kernel routines in nested (see Figure 4.3) can be produced by using "*kd -c*".

```
 Entry    Delta   PID    Trace
-------- -------- -----  -----------------------------------------------------
    0       -1      1     run_init_process
    0       -1      1     |  execve
    0   468750      1     |  |  do_execve
    0   312500      1     |  |  |  open_exec
    0   312500      1     |  |  |  |  path_lookup
    0   312500      1     |  |  |  |  |  link_path_walk
    0   312500      1     |  |  |  |  |  |  __link_path_walk
    0     7813      1     |  |  |  |  |  |  |  do_lookup
    0     7813      1     |  |  |  |  |  |  |  |  real_lookup
    0     7813      1     |  |  |  |  |  |  |  |  |  jffs2_lookup
    0     7813      1     |  |  |  |  |  |  |  |  |  |  jffs2_read_inode
    0     7813      1     |  |  |  |  |  |  |  |  |  |  |  jffs2_do_read_inode
```

Figure 4.3 Kernel Routines Date in Nested

### 4.2.2 Printk Times

Printk times [13] is a simple technology which adds some code to the standard kernel *printk* routine, to output timing data with each message. While a crude status, this can be used to get an overview of the areas of kernel initialization which take a relatively long time. This feature is used to identify areas of the Linux kernel requiring work.

With printk times turned on, the system emits the timing data as a floating point number of seconds (to microsecond resolution) for the time at which the *printk* started. The utility program shows the time between calls, or it can show the times relative to a specific message. This makes it easier to see the timing for specific segments of kernel code during boot.

### 4.2.3 initcall-times patch

Matt Mackall provided an initcall-times [13] patch which measures times for the

initialization of each driver during *do_initcalls*. This is a special tool to look at the time of initialization of buses and drivers. It times just the initcalls and is enabled by putting "*initcall_debug*" on the command line. The records of device initializations can be read by *dmesg* after boot and use *grep* to find time-consuming initializations (see Figure 4.4).

```
Calling initcall 0xc000ea6c: ptrace_break_init+0x0/0x2c()
 initcall elapsed  0.000000s - ptrace_break_init+0x0/0x2c()
Calling initcall 0xc000f8d4: consistent_init+0x0/0xb4()
 initcall elapsed  0.000061s - consistent_init+0x0/0xb4()
Calling initcall 0xc0013a30: helper_init+0x0/0x48()
 initcall elapsed  0.000427s - helper_init+0x0/0x48()
Calling initcall 0xc0013b88: ksysfs_init+0x0/0x44()
 initcall elapsed  0.000122s - ksysfs_init+0x0/0x44()
Calling initcall 0xc0015958: filelock_init+0x0/0x54()
 initcall elapsed  0.000091s - filelock_init+0x0/0x54()
Calling initcall 0xc0016320: init_script_binfmt+0x0/0x1c()
 initcall elapsed  0.000000s - init_script_binfmt+0x0/0x1c()
```

Figure 4.4 Initcall Log in Kernel Ring Buffer

### 4.2.4 Expect

Wolfgang Denk provides a expect [14] script do start-to-finish timings by filtering every outputted lines of *kermit* [15]. The timestamp is refers to the newline character, i.e. to the end of each line. Because this *expect* script measure the time on host, it depends on clock of host, not the clock of target. Therefore, the time measurement will not make any affection to the target. There is a special parameter called "*start_string*", which can be set to reset the timestamp (see Figure 4.5).

```
   5.837 Starting kernel ...
   5.837
   7.717 Uncompressing Linux..................................................
.......................... done, booting the kernel.
   8.794 Linux version 2.6.14-omap2 (root@phantom.cn.nctu.edu.tw) (gcc version 3.3.2)
#2 Tue Jul 18 16:06:26 CST 2006
   0.008 CPU: ARM926EJ-Sid(wb) [41069263] revision 3 (ARMv5TEJ)
   0.019 Machine: TI-OSK
   0.070 Memory policy: ECC disabled, Data cache writeback
   0.071 OMAP1611b revision 2 handled as 16xx id: 5b058f7948960a0f
```

Figure 4.5 The Timestamp Resetting

## 4.3 Measurement Tools Analysis

We must to check the accuracy of different tools on OMAP5912OSK. In order to obtain exact boot time, we use the oscilloscope to measure signals of RS232_TX which represent the console outputs. So we can compare the time before and after using specific tool, and cross check with the records of oscilloscope.

### 4.3.1 Kernel Function Trace

Since KFT add instrumentation callouts to every function entry and exit. The requirement of system performance will increase in a large amount. Therefore, the execution performance of KFT is limited to the platform. If the performance of specific platform is not enough, KFT will causes huge overhead when doing record. The timing result of KFT is not correct, because the result includes not only original execution time but also overheads.

In Table 4.1, we observe that the boot time will become 2 times because the performance of OMAP5912 can not meet the requirement of KFT. And most of boot time waste on routine *schedule* which reschedules tasks schedule when the usage of MPU is almost 100%.

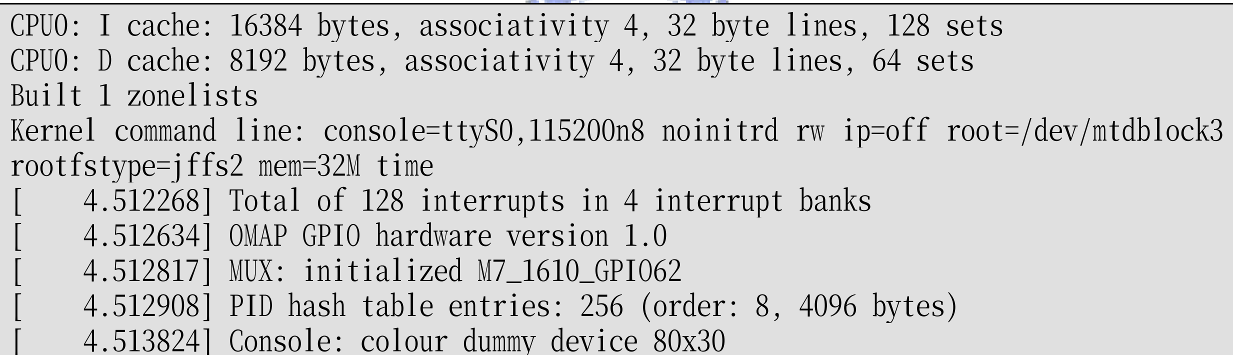
Table 4.1 KFT Activates from *start_kernel* to *to_userspace*

| Configuration | Boot time |
|---|---|
| CONFIG_KFT=n | 8.212 seconds |
| CONFIG_KFT=y, ONFIG_KFT_STATIC_RUN=n | 13.134 seconds |
| CONFIG_KFT=y, ONFIG_KFT_STATIC_RUN=y | 16.010 seconds |

### 4.3.2 Printk Times

At first, the time resolution of printk time is not enough to measure the time below 1 ms. Printk times uses the routine *sched_clock* to get timestamp, but *sched_clock* only has a time resolution of 1 jiffy which is 1/HZ = 1/128 = 7.8125 ms on OMAP5912. Although the value of HZ is 1000 in part of PC, the time resolution is only 1 jiffy = 1/1000 = 1 ms.

Secondly, OMAP5912 will hang when printk times function is compiled in the kernel, therefore we must use it dynamically, i.e. putting "*time*" on the command line. After using printk times dynamically, we observe that not all kernel messages have the timestamp (see Figure 4.6) until the kernel commands have passed.



```
CPU0: I cache: 16384 bytes, associativity 4, 32 byte lines, 128 sets
CPU0: D cache: 8192 bytes, associativity 4, 32 byte lines, 64 sets
Built 1 zonelists
Kernel command line: console=ttyS0,115200n8 noinitrd rw ip=off root=/dev/mtdblock3
rootfstype=jffs2 mem=32M time
[    4.512268] Total of 128 interrupts in 4 interrupt banks
[    4.512634] OMAP GPIO hardware version 1.0
[    4.512817] MUX: initialized M7_1610_GPIO62
[    4.512908] PID hash table entries: 256 (order: 8, 4096 bytes)
[    4.513824] Console: colour dummy device 80x30
```

Figure 4.6 No Timestamp before Some Messages

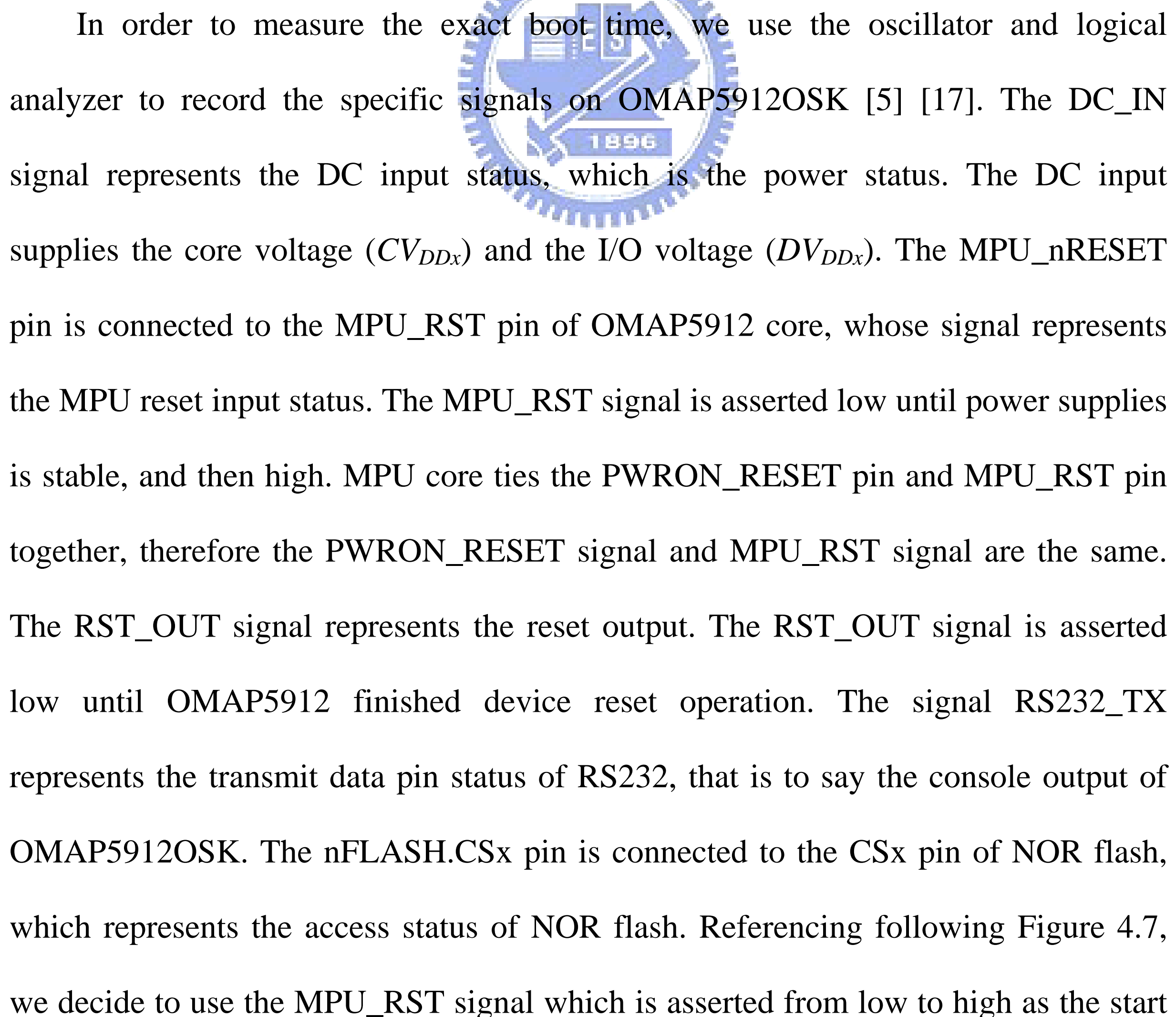
### 4.3.3 initcall-times patch

The default value of CONFIG_LOG_BUF_SHIFT is 14, that is to say, the kernel ring buffer size is $2^{14}$ B = 16 KB [16]. That is not sufficient to hold all the messages with the additional information of initcall-times patch. The kernel ring buffer size must be modified to 128 KB by setting to CONFIG_LOG_BUF_SHIFT to 17 to address the request of initcall-times patch.

### 4.3.4 Expect

This method can't measure the time before console initialized, i.e. the time of the hardware initialization time before U-Boot start can't be measured by *expect* script. Original kernel messages in normal mode are not enough to do accurate analysis; we can't measure the time of specific function or instruction. And the time result will include the delay from the RS232_TX ping to *kermit* doing decode signal to ASCII and outing. The delay time is different of each output and the boot time measured using *expect* is not correct enough.

## 4.4 Boot Time Measurement



In order to measure the exact boot time, we use the oscillator and logical analyzer to record the specific signals on OMAP5912OSK [5] [17]. The DC_IN signal represents the DC input status, which is the power status. The DC input supplies the core voltage ($CV_{DDx}$) and the I/O voltage ($DV_{DDx}$). The MPU_nRESET pin is connected to the MPU_RST pin of OMAP5912 core, whose signal represents the MPU reset input status. The MPU_RST signal is asserted low until power supplies is stable, and then high. MPU core ties the PWRON_RESET pin and MPU_RST pin together, therefore the PWRON_RESET signal and MPU_RST signal are the same. The RST_OUT signal represents the reset output. The RST_OUT signal is asserted low until OMAP5912 finished device reset operation. The signal RS232_TX represents the transmit data pin status of RS232, that is to say the console output of OMAP5912OSK. The nFLASH.CSx pin is connected to the CSx pin of NOR flash, which represents the access status of NOR flash. Referencing following Figure 4.7, we decide to use the MPU_RST signal which is asserted from low to high as the start

point of boot, and the final signal of RS232_TX (BusyBox banner) as the end point of boot.

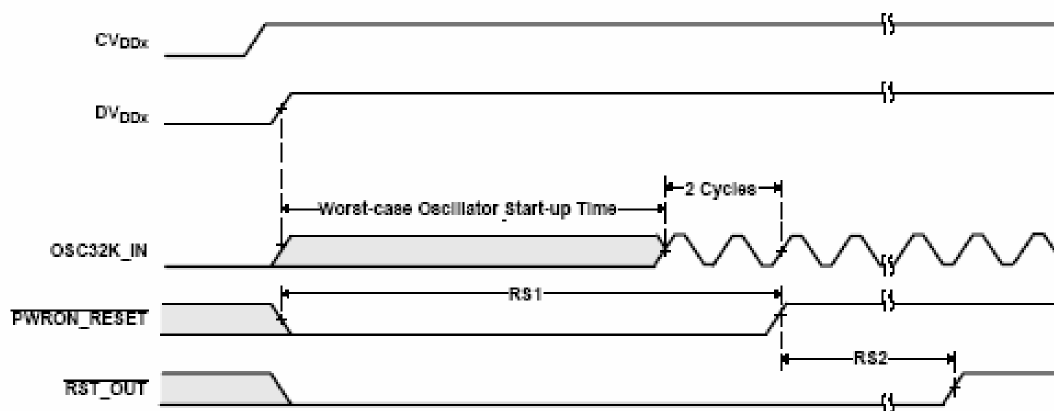



Figure 4.7 Device Reset Timing of OMAP5912

When boot, we capture the screen output of console as a 1000fps (1000 frames per second) movie to assist us to determine specific timestamp.

The records of oscillator and logical analyzer are ambiguous, we need to do cross check of the source code, the signals of MPU_nRESET, the signals of RS232_TX, the signals of nFLASH.CSx, the frames of boot movie and the *dmesg* information to determine which signal is represented for the specific one.

In order to measure further detailed timestamp of specific function, we use different methods and tools at the same time. At first, we trace full source code to obtain detailed boot sequence, i.e. from *start_armboot* of U-Boot to *ash_main* of BusyBox. And then, we hacking the source code to output function names and timings. In the source code of U-Boot, we use *puts* to output U-Boot function names. In the source code of Linux kernel, we use *printk(KERN_EMERG " ")* to output kernel function names and use *sched_clock* to calculate each execution time of function. In the source code of BusyBox, we use *fprintf(stderr, " ")* to output user space function names.

After measurement, we can obtain following useful information:

1. Full console records of boot in 1000fps movie. We can review boot process frame by frame (Every frame represents for 1 millisecond). The movie frame of finish uncompressing kernel and finish displaying Linux banner are show in Figure 4.8 and 4.9.

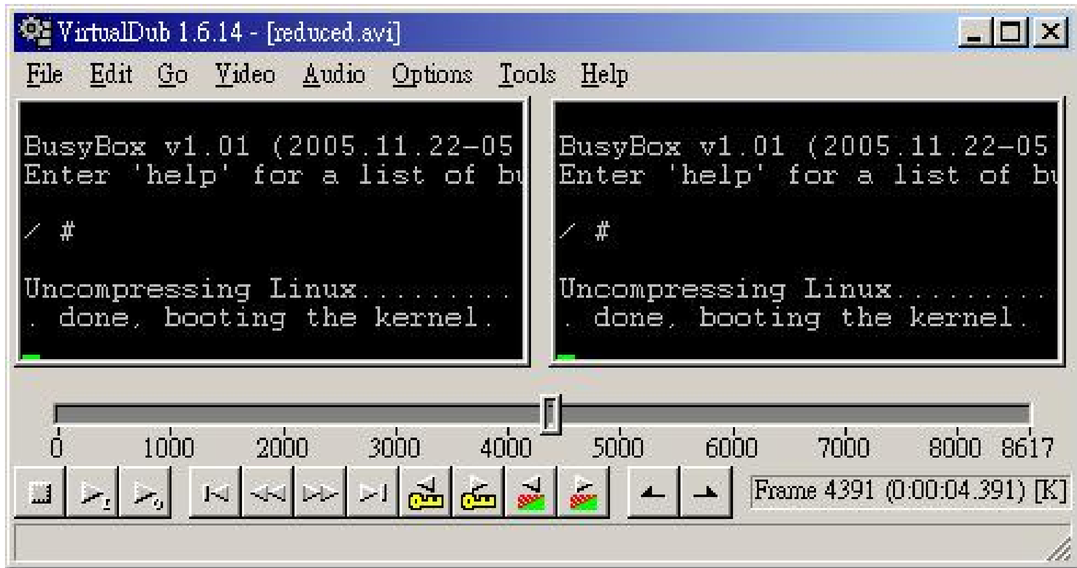



Figure 4.8 The Movie Frame of Finish Uncompressing Kernel

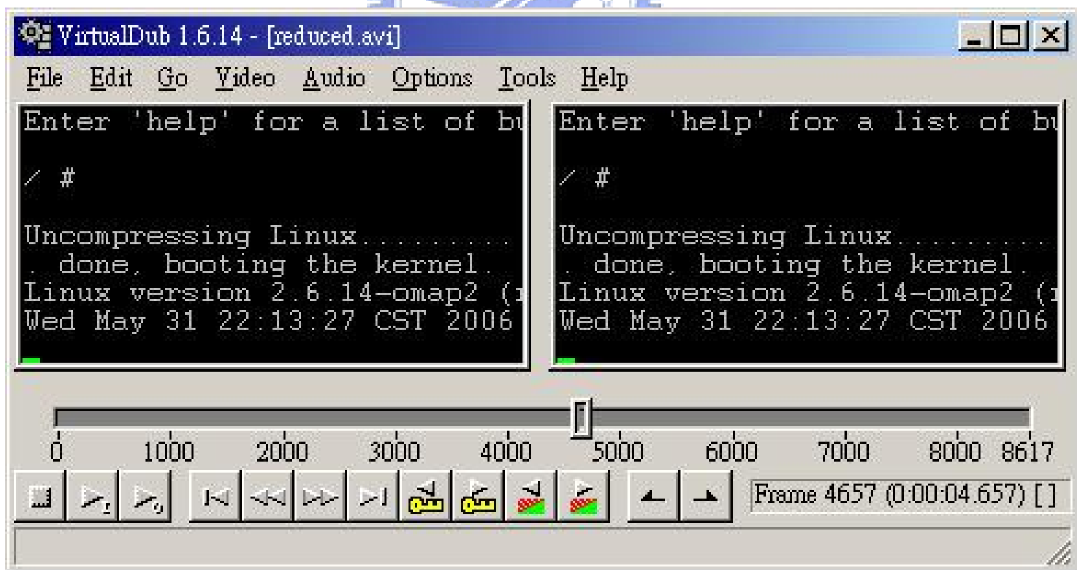



Figure 4.9 The Movie Frame of Finish Displaying Linux Banner

2. Kernel ring buffer. We can use *dmesg* to print the kernel ring buffer to review kernel and initcall_debug messages during kernel phase (see Figure 4.4 and 4.6).

3. The signal records of DC_IN, MPU_nRESET, RST_OUT, RS232_TX, nFLASH.CSx with 100 microsecond's time resolution (see Figure 4.10).

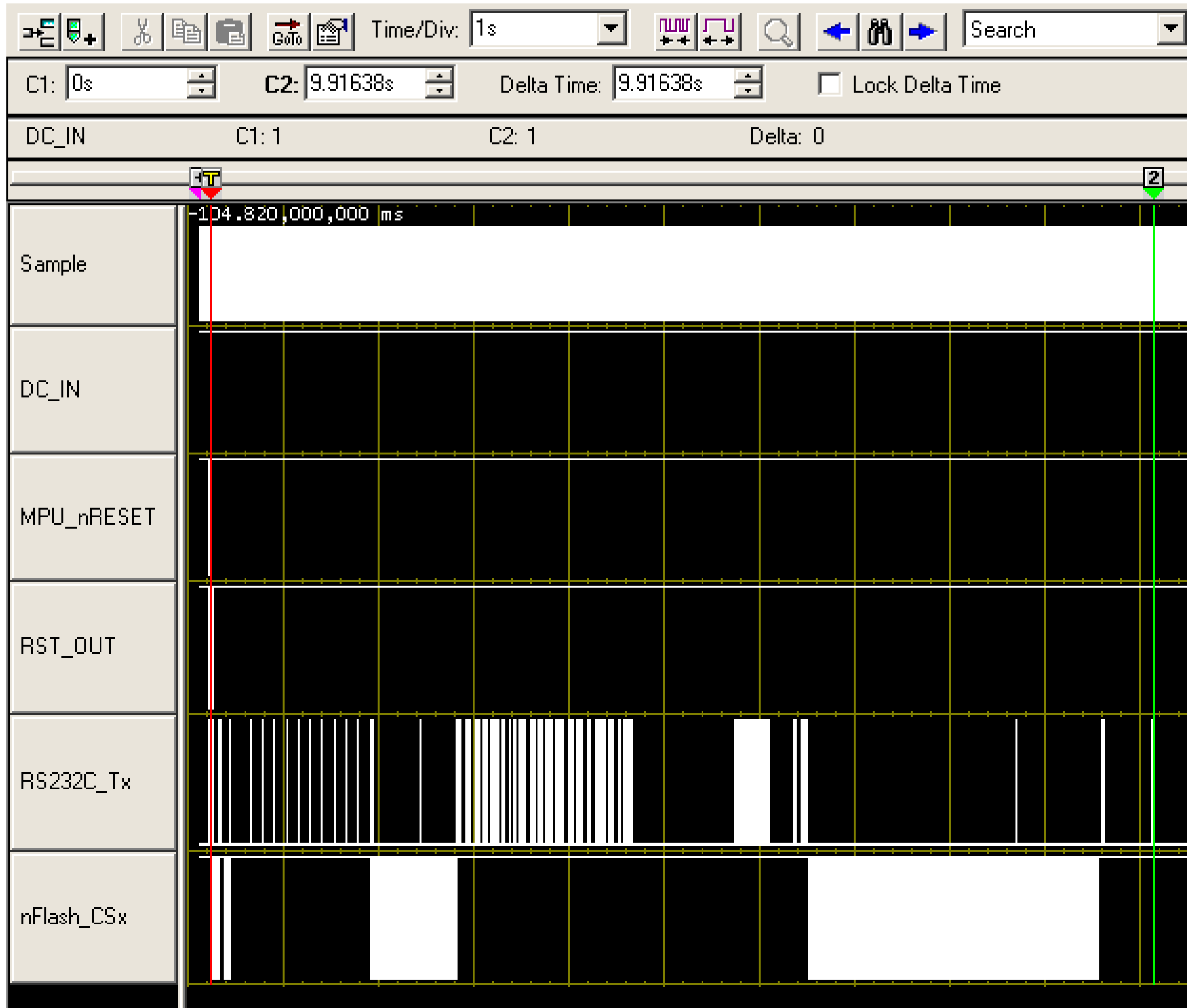



Figure 4.10 The Records of Logic Analyzer

## 4.5 Boot Sequence Analysis

After observing and cross check the original records, we add extra timestamp points and measure repeatedly to divide boot sequence into three levels: represent the level of application's point of view, the level of functional block's point of view and the level of instruction's point of view respectively.

### 4.5.1 Level I: From the Application's Point of View

We divide the boot sequence preliminary from the application's point of view. They are three major phases which art boot loader phase, kernel phase and user space phase. U-Boot 1.1.3 is executed in boot loader phase. Embedded Linux kernel 2.6.14 is executed in the kernel phase. BusyBox v1.01 is executed in user space phase.

### 4.5.2 Level II: From the Functional Block's Point of View

We subdivide the three major phases further by timing result, to enable us to measure more detailed boot time. In order to do subdivision, we need to cross check the boot movie frames, kernel ring buffer and the signal records of oscillator and logical to decide the separate time of the different function block. And subdivide the boot time to 18 functional blocks by the characteristic of different signal of oscillator and logical analyzer, e.g. the flash access status or the console output. The 18 functional blocks is shown at Table 4.2.

Table 4.2 The 18 functional blocks

<table>
<tr><th rowspan="2"></th><th rowspan="2">#</th><th>Start Point</th><th>End Point</th></tr>
<tr><th colspan="2">Description</th></tr>
<tr><td rowspan="6">Boot loader phase</td><td rowspan="2">1</td><td>Device reset start</td><td>Device reset over</td></tr>
<tr><td colspan="2">From the first signal MPU_nRESET becoming high to the first signal RST_OUT becoming high. OMAP5912OSK enable the hardware preliminary.</td></tr>
<tr><td rowspan="2">2</td><td>Device reset over</td><td>MPU read first instruction</td></tr>
<tr><td colspan="2">From the first signal RST_OUT becoming high to the first signal RS232_TX becoming low. OMAP5912OSK start to execute the first instruction.</td></tr>
<tr><td rowspan="2">3</td><td>MPU read first instruction</td><td>env_relocate_spec start</td></tr>
<tr><td colspan="2">From the first signal nFLASH.CSx becoming low to the signal RS232_TX of function env_relocate_spec start (the signal RS232_TX becoming high). U-boot starts and prepares to execute the first function which access flash.</td></tr>
</table>

<table>
<tr><td rowspan="12"></td><td rowspan="2">4</td><td>env_relocate_spec start</td><td>env_relocate_spec over</td></tr>
<tr><td colspan="2">From the signal RS232_TX of function env_relocate_spec start to the signal RS232_TX of function env_relocate_spec over. env_relocate_spec relocates the environment parameters.</td></tr>
<tr><td rowspan="2">5</td><td>env_relocate_spec over</td><td>Image date checksum start</td></tr>
<tr><td colspan="2">From the signal RS232_TX of function env_relocate_spec over to the signal RS232_TX of image date checksum start. This function block does not access flash.</td></tr>
<tr><td rowspan="2">6</td><td>Image date checksum start</td><td>Image date checksum over</td></tr>
<tr><td colspan="2">From the signal RS232_TX of image date checksum start to the signal RS232_TX of image date checksum over. U-Boot verifies the image data checksum.</td></tr>
<tr><td rowspan="2">7</td><td>Image date checksum over</td><td>Copying image to ram start</td></tr>
<tr><td colspan="2">From the signal RS232_TX of image date checksum over to the signal RS232_TX of copying image to ram start. U-Boot finished image data checksum and prepares copy image to DDRRAM.</td></tr>
<tr><td rowspan="2">8</td><td>Copying image to ram start</td><td>Copying image to ram over</td></tr>
<tr><td colspan="2">From the signal RS232_TX of copying image to ram start to the signal RS232_TX of copying image to ram over. U-Boot copies image from flash to DDRRAM.</td></tr>
<tr><td rowspan="2">9</td><td>Copying image to ram over</td><td>Transfer control to Linux</td></tr>
<tr><td colspan="2">From the signal RS232_TX of copying image to ram over to the signal RS232_TX of transferring control to Linux. U-Boot transfers control to Linux kernel.</td></tr>
<tr><td rowspan="9">Kernel phase</td><td rowspan="2">10</td><td>Transfer control to Linux</td><td>Uncompress kernel start</td></tr>
<tr><td colspan="2">From the signal RS232_TX of transferring control to Linux to the signal RS232_TX of uncompressing kernel start. Linux kernel gets the control and prepares to uncompress kernel.</td></tr>
<tr><td rowspan="2">11</td><td>Uncompress kernel start</td><td>Uncompress kernel over</td></tr>
<tr><td colspan="2">From the signal RS232_TX of the signal RS232_TX of uncompressing kernel start to the signal RS232_TX of the signal RS232_TX of uncompressing kernel over. Linux kernel is uncompressed.</td></tr>
<tr><td rowspan="2">12</td><td>Uncompress kernel over</td><td>jffs2_build_filesystem start</td></tr>
<tr><td colspan="2">From the signal RS232_TX of uncompressing kernel over to the signal RS232_TX of jffs2_build_filesystem start. Linux kernel uncompressed and execute routine start_kernel, Linux kernel does not access flash until the routine mount_root invoking jffs2_build_filesystem.</td></tr>
<tr><td rowspan="2">13</td><td>jffs2_build_filesystem start</td><td>jffs2_build_filesystem over</td></tr>
<tr><td colspan="2">From the signal RS232_TX of jffs2_build_filesystem start to the signal RS232_TX of jffs2_build_filesystem over. Linux kernel builds the jffs2 file system.</td></tr>
<tr><td>14</td><td>jffs2_build_filesystem over</td><td>Invoke init</td></tr>
</table>

| | | | |
|---|---|---|---|
| | | From the signal RS232_TX of *jffs2_build_filesystem* over to the signal RS232_TX of invoking *init*. Root file system has been built and Linux kernel invoke the sysvinit tool: */sbin/init*. | |
| User space phase | 15 | Invoke *init* | *init_main* start |
| | | From the signal RS232_TX of invoking *init* to the signal RS232_TX of invoking *init_main* start. Linux kernel still run background routines and *init* wait for start. | |
| | 16 | *init_main* start | RC script start |
| | | From the signal RS232_TX of invoking *init_main* start to the signal RS232_TX of RC script start. *init_main* started for user space and prepares to run RC script. | |
| | 17 | RC script start | RC script over |
| | | From the signal RS232_TX of RC script start to the signal RS232_TX of RC script over. RC script starts several daemons. | |
| | 18 | RC script over | Shell prompt |
| | | From the signal RS232_TX of RC script over to the signal RS232_TX of shell prompt. RC script finished and shell prompt enabled. | |

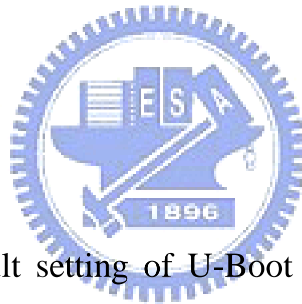
After subdivide the boot sequence to 18 function blocks, each function has the similar characteristic. That is to say, the behaviors of all function in a function block are almost similar. It is useful for our redundancy analysis.

We supply a template of the boot time measurement table. See Table 4.3.

Table 4.3 A Template of Boot Time Measurement Table

| | Level I | Level II | |
|---|---|---|---|
| | | Start Point | End Point |
| | | Time | |
| Total ms | Boot Loader ms | Device reset start | Device reset over |
| | | ms | |
| | | Device reset over | MPU read first instruction |
| | | ms | |
| | | MPU read first instruction | *env_relocate_spec* start |
| | | ms | |
| | | *env_relocate_spec* start | *env_relocate_spec* over |
| | | ms | |
| | | *env_relocate_spec* over | image date checksum start |
| | | ms | |
| | | image date checksum start | image date checksum over |
| | | ms | |

| | | image date checksum over | copy image to ram start |
|---|---|---|---|
| | | ms | |
| | | copy image to ram start | copy image to ram over |
| | | ms | |
| | | copy image to ram over | transfer control to Linux |
| | | ms | |
| | Kernel<br>ms | transfer control to Linux | uncompress kernel start |
| | | ms | |
| | | uncompress kernel start | uncompress kernel over |
| | | ms | |
| | | uncompress kernel over | *jffs2_build_filesystem* start |
| | | ms | |
| | | *jffs2_build_filesystem* start | *jffs2_build_filesystem* over |
| | | ms | |
| | | *jffs2_build_filesystem* over | invoke *init* |
| | | ms | |
| | User Space<br>ms | invoke *init* | *init_main* start |
| | | ms | |
| | | *init_main* start | RC script start |
| | | ms | |
| | | RC script start | RC script over |
| | | ms | |
| | | RC script over | Shell prompt |
| | | ms | |



### 4.5.3 Measurement result

The boot time using default setting of U-Boot 1.1.3, Linux kernel 2.6.14 and BusyBox 1.01 is 7934.41 ms (Do not include the part of wait). Among them, boot loader spends 1111.76 ms, Linux kernel spends 5882.60 ms and user space spends 943.05 ms. The detailed time is in the Table 4.4.

Table 4.4 Boot Time with 18 Function Blocks

| | Level I | Level II | |
|---|---|---|---|
| | | Start Point | End Point |
| | | Time | |
| Total<br>7934.41 ms | Boot Loader<br>1111.76 ms | Device reset start | Device reset over |
| | | 31.38 ms | |
| | | Device reset over | MPU read first instruction |
| | | 0.74 ms | |
| | | MPU read first instruction | *env_relocate_spec* start |
| | | 122.26 ms | |
| | | *env_relocate_spec* start | *env_relocate_spec* over |
| | | 44.98 ms | |
| | | *env_relocate_spec* over | image date checksum start |
| | | 27.62 + 1477.76 ms (wait for user: 1477.76 ms) | |

| | | | |
|---|---|---|---|
| | | image date checksum start | image date checksum over |
| | | 487.92 ms | |
| | | image date checksum over | copy image to ram start |
| | | 0.44 ms | |
| | | copy image to ram start | copy image to ram over |
| | | 395.52 ms | |
| | | copy image to ram over | transfer control to Linux |
| | | 0.90 ms | |
| | Kernel<br>5882.60 ms | transfer control to Linux | uncompress kernel start |
| | | 13.48 ms | |
| | | uncompress kernel start | uncompress kernel over |
| | | 1838.62 ms | |
| | | uncompress kernel over | *jffs2_build_filesystem* start |
| | | 1840.48 ms | |
| | | *jffs2_build_filesystem* start | *jffs2_build_filesystem* over |
| | | 2179.54 ms | |
| | | *jffs2_build_filesystem* over | invoke *init* |
| | | 10.48 ms | |
| | User Space<br>943.05 ms | invoke *init* | *init_main* start |
| | | 818.22 ms | |
| | | *init_main* start | RC script start |
| | | 37.10 ms | |
| | | RC script start | RC script over |
| | | 65.98 ms | |
| | | RC script over | Shell prompt |
| | | 21.75 + 656.79 ms (wait for user: 656.79 ms) | |

## 4.6 Summary

In Chapter 4, we describe the boot sequence and the boot time measurement tools, and then analyze the boot time measurement tools. Finally, we purpose a method to measure the exact time of specific function or instruction, and measure the boot time with 18 function blocks.

# Chapter 5

# Experiment

After obtaining the detailed and exact boot time measurement results, we can find out the redundant operations. Finally, we can rewrite or skip them to achieve faster boot without any side effect.

## 5.1 Redundancy Analysis

By the time measurement result, at first, we can observe many operations of accessing flash and some bad configuration during boot loader phase. Secondly, during kernel phase, we can obtain the execution of all kernel routines by printk useful information. Finally, the choice of file system has huge affection to the boot time. After checking that, we can conclude following redundant works and the methods of fast boot.

- **Boot loader phase**

**METHOD 01: Adjust clocking mode**

**[Original]** The Default setting of U-Boot uses fully synchronous mode as the clocking mode [18]. In fully synchronous mode, the MPU, DSP, and Memory traffic controller (TC) domains run at the same clock frequency derived from DPLL1.

**[Limitation]** The frequency of MPU and DSP are limited by the upper bound of TC [19], i.e. 96 MHz. However, the upper bound frequency of MPU and DSP are 192 MHz. So, the performance of U-Boot is limited because the frequency of each

domain.

**[Modification]** We changed the clocking mode from fully synchronous mode to synchronous scalable mode by setting the value of ARM_SYSST (MPU System Status Register) from 0x0000 to 0x1000 [18] [19]. In synchronous scalable mode, the domains of MPU, DSP, and TC are synchronous and run at different clock speeds.

**[Improvement]** We can ramp up the DPLL1 clock to 192 MHz and let MPU work on 192 MHz by setting ARMDIV to 00, i.e. the frequency of ARM core equals the frequency of DPLL1 divided by $2^0$, and TC work on 96 MHz at the same time by setting TCDIV to 01, i.e. the frequency of TC equals the frequency of DPLL1 divided by $2^1$.

**METHOD 02: Reduce unused console functions**

**[Original]** During U-Boot doing initialization, the initialization of console device is separated into two functions: *console_init_f* and *console_init_r*.

**[Limitation]** After executing the two functions sequentially, the console device will be initialized as a fully console device. However, we do not need U-Boot to provide a fully console device during boot. Therefore fully initialization of console device is redundant.

**[Modification]** After reading the U-Boot source code and doing experiment repeatedly, we know that the function *console_init_r* is useless during boot. Therefore, we skip the execution of *console_init_r*.

**[Improvement]** The execution time of console_init_r can be saved. Although we skip the execution of *console_init_r*, boot is still successful and the output messages still can be shown by console after the function *console_init_f* finished the first stage initialization of console.

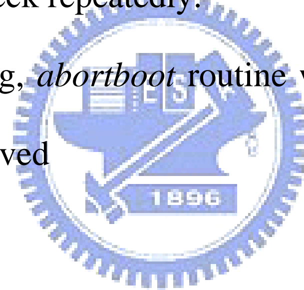
**METHOD 03: Improve abort boot function**

**[Original]** The function *abortboot* will lock U-Boot to wait and check if any key already pressed. If there is any key already pressed, function *abortboot* will abort the boot process, and redirect to U-Boot prompt. Otherwise, after numbers of second, function *abortboot* will unlock U-Boot, and resume the boot process.

**[Limitation]** The time of waiting is *bootdelay* seconds; the default value of *bootdelay* is setting as 10. It will waste 1.25s to wait using U-Boot 1.1.3 (The timer is not accurate; the correct wait time should be 10s. If the timer is accurate, the boot time should add 10-1.25=8.75s more).

**[Modification]** We modified the code of function *abortboot* to reduce the waiting time during U-Boot check if any key already pressed. Original *abortboot* routine will spend numbers of seconds to check repeatedly.

**[Improvement]** After modifying, *abortboot* routine will only check once and waste no time. The wait time can be saved


1896

**METHOD 04: Improve image verification mechanism**

**[Original]** U-Boot provides an image verification mechanism; it will verify both header checksum and data checksum of image at each time during boot.

**[Limitation]** In fact, after burning image, we only need to verify the image checksum once. If the image is correct, doing image verification each time during boot is nonsensical.

**[Modification]** We added a new parameter called *verify* in the U-Boot environment parameters and regard it as a switch of the image verification mechanism. When *verify* is *y*, U-Boot will do header checksum and data checksum same as default. When *verify* is *n*, U-boot will skip the operation of verification.

**[Improvement]** In the practical application, we set *verify* as *y* after burning to verify

the image checksum and set *verify* as *n* if we sure the image is correct. Therefore, we can save the time of image verification.

**METHOD 05: Use silent console in boot loader phase**

**[Original]** U-Boot provides some functions to print the information of devices; the information is useful during development and debug. In U-Boot, most of device initialization and information are deal with separate functions.

**[Limitation]** The execution of information function and every console output by serial port will spend much time.

**[Modification]** We added a new parameter called *quiet* in the U-Boot environment parameters and modified the U-Boot source code to achieve quiet console. When *quiet* is *n*, U-Boot will show full messages of U-Boot banner, dram configuration, flash configuration, function *abartboot*, image verification and invoking Linux kernel. When *quiet* is *y*, U-Boot will show no console messages.

**[Improvement]** By the parameter *quiet*, we can use the silent console to reduce boot time.

- **Kernel phase**

**METHOD 06: Use uncompressed kernel image**

**[Original]** In the past, the cost of flash storage device in embedded product is quite high, so compressed kernel is used to reduce the cost of product. However, the compressed kernel size of optimized embedded Linux is general less than 1 MB, and the uncompressed kernel size is less than 2MB. At present, the cost of 1MB flash storage is not so high. Therefore, using uncompressed kernel becomes an acceptable choice.

**[Modification]** We change the *Makefile* in *linux/arch/arm/boot* to build an uncompressed image for U-Boot.

**[Improvement]** The size of uncompressed kernel is close to 2.1 times of compressed kernel. Therefore the time of coping uncompressed image from flash to ram is close to 2.1 times of compressed kernel, too. However, after comparing the time of coping image and uncompressing kernel between uncompressed and compressed kernel, using uncompressed kernel can save huge proportion of boot time. Although the size of uncompressed kernel is bigger, it is still within the default upper limit (2MB).

**METHOD 07: Eliminate BogoMIPS calibration**

**[Original]** The function *calibrate_delay* [10] [20] [21] can compute an appropriate value for *loops_per_jiffy* and *BogoMIPS* at boot time. The value of *loops_per_jiffy* is used to execute busy wait (non-yielding) delays inside the Linux kernel and primarily dependent on processor speed. *BogoMips* is an unscientific performance of MPU and cache, and the ratio of *loops_per_jiffy*. Its initial value at boot time is expected to be constant for each boot of Linux on the same hardware.

**[Limitation]** The value of *loops_per_jiffy* is primarily dependent on processor speed. Therefore, its initial value at boot time is expected to be constant for each boot of Linux on the same hardware. We don't need to compute the value every system boot.

**[Modification]** Because of the initial value at boot time is expected to be constant, we can preset the initial value in advance.

**[Improvement]** By to preset the initial value in advance, we can avoid the delay associated with dynamically calculating the value, by the kernel, on every system boot.

**METHOD 08: Use device modularization**

**[Original]** Kernel initiates many devices during boot in default setting for different purpose of every kind of product.

**[Limitation]** Because of we might not need all devices as default setting, many settings become non-critical or useless. For example, if we don't need pseudo terminal device (PTY), we should remove or modularize it.

**[Modification]** By reading the *dmesg* information, we can observe the useless, non-critical or time-consumed devices. After understanding the function of those devices, we should decide which the non-critical devices are. In our experiment platform, we should change the setting of shared memory file system, paging of anonymous memory (swap) support, resetting unused clocks, OMAP multiplexing support, PCMCIA/CardBus support, firewall support, loopback device support, initial RAM disk support, ATA/ATAPI/MFM/RLL support, PPP support, frame buffer devices support, second extended file system support and kernel automounter support and NFS file system support…etc, to remove or modularize them.

**[Improvement]** By removing and modularizing device driver, we can save much time in initiating useless, non-critical or time-consumed device.

**METHOD 09: Use silent console in kernel phase**

**[Original]** During boot, Linux kernel provides much information for debug. Because of the printk messages of kernel are quite a lot, they will spend much time by using serial port or VGA [10].

**[Modification]** We can add *quiet* parameter in the kernel command line to changes the loglevel to 4, which suppresses the output of regular (non-emergency) *printk* messages. Even though the messages are not printed to the system console, they are still placed in the kernel *printk* buffer, and can be retrieved after boot using the *dmesg*

command.

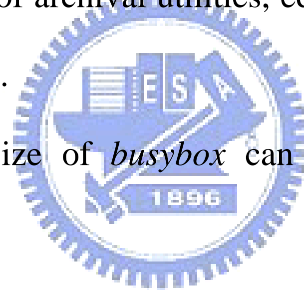
**[Improvement]** We can unable the printk output, and view the message using dmesg. That will save some time.

- **User space phase**

**METHOD 10: Simplify user space utilities**

**[Original]** BusyBox provide many useful utilities for using of user space. However, we should give up some utilities that have similar function or useless in embedded product.

**[Modification]** To reduce the size of *busybox* is to reduce the size of file system. We can give up some similar utilities and some useless utilities which are related with the requirement of a product. Most of archival utilities, editors and console utilities could be gave up in embedded product.



**[Improvement]** The smaller size of *busybox* can reduce the execution time of *busybox*.

**METHOD 11: Accelerate shell prompt start**

**[Original]** For the reason of saving memory, BusyBox will lock and wait for user to press Enter key to activate shell prompt.

**[Limitation]** Generally, user wants to use a product immediately and don't need to press extra key. And the memory using of shell is few comparing full memory size on OMAP5912OSK.

**[Modification]** We skip the wait operation and put shell prompt directly.

**[Improvement]** The time from the "Please press Enter to activate this console" massage shown to user pressing enter is measured as 600 ms in average. That is too long to reduce the boot time.

**METHOD 12: Use complex file system**

**[Original]** By the time measurement result, we can observe function *mount_root* of kernel spend a large amount of time to build the JFFS2 files system. If we can change the file system which has a short mount time, the time can be saving.

**[Limitation]** Generally, we use the JFFS2 (The Journalling Flash File System, version 2) file system which is log-structured and writable on flash storage device in embedded systems. However, for a 32MB NOR flash, kernel always spends 2 to 3 seconds to build the JFFS2 file system. The mount time of JFFS2 file system is too long to make the boot time shorter.

**[Modification]** For flash storage device, CramFS and SquashFS are highly compressed read-only file system, the runtime performance and compression of SquashFS is better than CramFS. No matter CramFS or SquashFS, the mount time is quite short.

In view of the characteristics of writable and read-only file system, we use both writable and read-only file system on a single flash storage device at the same time. First, using appropriate spaces as root file system partition including init and most of routines, then using remaining space as writable file system. Finally we let the function *mount_root* just build the root file system, and build the writable file system in the background after shell prompt.

**[Improvement]** The boot time can be reduced greatly, and we still can do write operation on flash storage.

## 5.2 Comparison

In this section, we compare the time needed of affected function block. In

addition, we will supply the patch file if we modified the source code.

- **Boot loader phase**

**METHOD 01: Adjust clocking mode**

We modified the file *u-boot/board/omap5912/platform.S* (If you use last version of U-Boot, you need to modify the file u-boot/*board/omap5912osk/lowlevel_init.S*) to change the clocking mode. After we change the clocking mode and ramp up the frequency of ARM core to 192 MHz, the timer inaccurate timer become accurate one, and the wait time of *abortboot* is also accurate. Therefore, we skip the time measurement result of the function block which includes the execution of *abortboot*. Because U-Boot works on the upper limited frequency, operations which use MPU to compute data will have the shorter execution time. This part of modification reduces the time needed from 4774.72 ms to 3811.24 ms, i.e. 963.48 ms has been eliminated. The time comparison is shown at Table 5.1 and the patch is shown at Figure 5.1.

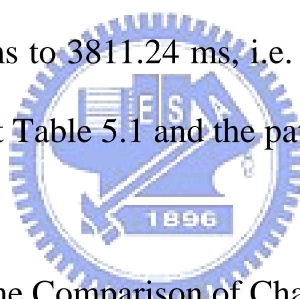


Table 5.1 The Time Comparison of Changing Clocking Mode

| | Function block | | Before (ms) | After (ms) |
|---|---|---|---|---|
| | Start Point | End Point | | |
| Boot Loader | Device reset start | Device reset over | 31.38 | 31.38 |
| | Device reset over | MPU read first instruction | 0.74 | 0.82 |
| | MPU read first instruction | *env_relocate_spec* start | 122.26 | 100.42 |
| | *env_relocate_spec* start | *env_relocate_spec* over | 44.98 | 37.02 |
| | *env_relocate_spec* over | image date checksum start | - | - |
| | image date checksum start | image date checksum over | 487.92 | 423.68 |
| | image date checksum over | copy image to ram start | 0.44 | 0.50 |
| | copy image to ram start | copy image to ram over | 395.52 | 323.38 |
| | copy image to ram over | transfer control to Linux | 0.90 | 0.88 |
| Kernel | transfer control to Linux | uncompress kernel start | 13.48 | 13.32 |
| | uncompress kernel start | uncompress kernel over | 1836.62 | 1040.92 |
| | uncompress kernel over | *jffs2_build_filesystem* start | 1840.48 | 1838.92 |
| Amount | | | 4774.72 | 3811.24 |

```
--- board/omap5912osk/platform.S.old    2006-07-20 21:51:21.000000000 +0800
+++ board/omap5912osk/platform.S        2006-07-20 21:53:38.000000000 +0800
@@ -79,7 +79,7 @@

        /* Set CLKM to Sync-Scalable    */
        /* I supposedly need to enable the dsp clock before switching */
-       mov     r1,     #0x0000
+       mov     r1,     #0x1000
        ldr     r0,     REG_ARM_SYSST
        strh    r1,     [r0]
        mov     r0,     #0x400
@@ -384,9 +384,9 @@
        .word 0x00800002

 VAL_ARM_CKCTL:
-       .word 0x3000
+       .word 0x050f
 VAL_DPLL1_CTL:
-       .word 0x2830
+       .word 0x2810

 #ifdef CONFIG_OSK_OMAP5912
 VAL_TC_EMIFS_CS0_CONFIG:
```

Figure 5.1 The Patch of Changing Clocking Mode

**METHOD 02: Reduce unused console functions**

It is easy to skip the fully console device initialization, we delete the function call *console_init_r* in file *u-boot/lib_arm/board.c*. We also delete the function call *misc_init_r*, because that function is temp one. The time needed is reduced from 219.73 ms to 0 ms; it is shown at Table 5.2.

Table 5.2 The Time Reduced by Skipping *console_init_r*

| Function | Before (ms) | After (ms) |
|---|---|---|
| *console_init_r* | 219.74 | 0.0 |

**METHOD 03: Improve abort boot function**

After we simply the function *abortboot* in *u-boot/common/main.c*, the time needed is reduced from 1704.74ms to 448.12 ms, i.e. 1256.74 ms has been eliminated, and it is shown at Table 5.3 and the patch is shown at Figure 5.2.

Table 5.3 The Time Reduced by Simplifying *abortboot*

| Function block | | Before (ms) | After (ms) |
|---|---|---|---|
| Start Point | End Point | | |
| Device reset start | Device reset over | 31.38 | 31.38 |
| Device reset over | MPU read first instruction | 0.74 | 0.74 |
| MPU read first instruction | *env_relocate_spec* start | 122.26 | 124.18 |
| *env_relocate_spec* start | *env_relocate_spec* over | 44.98 | 45.28 |
| *env_relocate_spec* over | image date checksum start | 1505.38 | 246.54 |
| Amount | | 1704.74 | 448.12 |

```
--- u-boot-1.1.3/common/main.c.old     2006-07-21 01:04:03.000000000 +0800
+++ u-boot-1.1.3/common/main.c         2006-07-21 01:07:06.000000000 +0800
@@ -238,7 +238,6 @@
        printf("Hit any key to stop autoboot: %2d ", bootdelay);
 #endif

-#if defined CONFIG_ZERO_BOOTDELAY_CHECK
        /*
         * Check if key already pressed
         * Don't check if bootdelay < 0
@@ -250,29 +249,6 @@
                        abort = 1;      /* don't auto boot      */
                }
        }
-#endif
-
-       while ((bootdelay > 0) && (!abort)) {
-               int i;
-
-               --bootdelay;
-               /* delay 100 * 10ms */
-               for (i=0; !abort && i<100; ++i) {
-                       if (tstc()) {   /* we got a key press   */
-                               abort  = 1;     /* don't auto boot      */
-                               bootdelay = 0;  /* no more delay        */
-# ifdef CONFIG_MENUKEY
-                               menukey = getc();
-# else
-                               (void) getc();  /* consume input        */
-# endif
-                               break;
-                       }
-                       udelay (10000);
-               }
-
-               printf ("\b\b\b%2d ", bootdelay);
-       }

        putc ('\n');
```

Figure 5.2 The Patch of Simplifying *abortboot*

**METHOD 04: Improve image verification mechanism**

After we added a new parameter called *verify* in the U-Boot environment parameters and modified file *u-boot/common/cmd_bootm.c*. When *verify* is *n*, U-boot will skip the verification of image header checksum and image data checksum. The time needed is reduced from 2193.1 ms to 1702.94 ms, i.e. 490.16 ms has been eliminated, and it is shown at Table 5.4 and the patch is shown at Figure 5.3.

Table 5.4 The Time Reduced by Verification Switch

| | Function block | | Before (ms) | After (ms) |
|---|---|---|---|---|
| | Start Point | End Point | | |
| Boot Loader | Device reset start | Device reset over | 31.38 | 31.34 |
| | Device reset over | MPU read first instruction | 0.74 | 0.74 |
| | MPU read first instruction | *env_relocate_spec* start | 122.26 | 122.90 |
| | *env_relocate_spec* start | *env_relocate_spec* over | 44.98 | 45.10 |
| | *env_relocate_spec* over | image date checksum start | 1505.38 | 1502.86 |
| | image date checksum start | image date checksum over | 487.92 | |
| | image date checksum over | copy image to ram start | 0.44 | |
| Amount | | | 2193.10 | 1702.94 |

```
--- u-boot-1.1.3/common/cmd_bootm.c.old 2005-08-14 07:53:35.000000000 +0800
+++ u-boot-1.1.3/common/cmd_bootm.c     2006-07-21 01:45:33.000000000 +0800
@@ -191,16 +191,18 @@
        }
        SHOW_BOOT_PROGRESS (2);

-       data = (ulong)&header;
-       len  = sizeof(image_header_t);
+       if (verify) {
+               data = (ulong)&header;
+               len  = sizeof(image_header_t);

-       checksum = ntohl(hdr->ih_hcrc);
-       hdr->ih_hcrc = 0;
+               checksum = ntohl(hdr->ih_hcrc);
+               hdr->ih_hcrc = 0;

-       if (crc32 (0, (char *)data, len) != checksum) {
-               puts ("Bad Header Checksum\n");
-               SHOW_BOOT_PROGRESS (-2);
-               return 1;
+               if (crc32 (0, (char *)data, len) != checksum) {
+                       puts ("Bad Header Checksum\n");
+                       SHOW_BOOT_PROGRESS (-2);
+                       return 1;
+               }
        }
```

```
        SHOW_BOOT_PROGRESS (3);
```

Figure 5.3 The Patch of Verification Switch

**METHOD 05: Use silent console in boot loader phase**

There are five files need to be modified. In the file *u-boot/common/cmd_bootm.c*, we need to modify the code of U-Boot banner, *print_image_hdr*. In the file *u-boot/common/main.c*, we need to modify the code of *abortboot* message. In the file *u-boot/include/configs/omap5912osk.h*, we set CFG_CONSOLE_INFO_QUIET=1. In the file *u-boot/lib_arm/armlinux.c*, we modify the code of transfer control to Linux. Finally, in the file *u-boot/lib_arm/board.c*, we modify the code of *display_banner, dram_init*, *display_dram_config* and *display_flash_config*. This part of silent console reduces the time needed from 2603.00 ms to 2557.80 ms, i.e. 45.20 ms has been eliminated, and it is shown at Table 5.5 and the sample patch for *u-boot/lib_arm/armlinux.c* is shown at Figure 5.4.

Table 5.5 The Time Reduced by Silent Console in U-Boot

| | Function block | | Before (ms) | After (ms) |
|---|---|---|---|---|
| | Start Point | End Point | | |
| Boot Loader | Device reset start | Device reset over | 31.38 | 31.38 |
| | Device reset over | MPU read first instruction | 0.74 | 0.74 |
| | MPU read first instruction | *env_relocate_spec* start | 122.26 | 109.48 |
| | *env_relocate_spec* start | *env_relocate_spec* over | 44.98 | 45.24 |
| | *env_relocate_spec* over | image date checksum start | 1505.38 | 1474.04 |
| | image date checksum star | image date checksum over | 487.92 | 884.72 |
| | image date checksum over | copy image to ram start | 0.44 | |
| | copy image to ram start | copy image to ram over | 395.52 | |
| | copy image to ram over | transfer control to Linux | 0.90 | 0.78 |
| Kernel | transfer control to Linux | uncompress kernel start | 13.48 | 11.42 |
| Amount | | | 2603.00 | 2557.80 |

```
diff -Nur u-boot-1.1.3/lib_arm/armlinux.c u-boot-1.1.3o/lib_arm/armlinux.c
--- u-boot-1.1.3/lib_arm/armlinux.c     2005-08-14 07:53:35.000000000 +0800
+++ u-boot-1.1.3o/lib_arm/armlinux.c    2006-07-21 03:43:12.000000000 +0800
```

```
@@ -85,11 +85,15 @@
        void (*theKernel)(int zero, int arch, uint params);
        image_header_t *hdr = &header;
        bd_t *bd = gd->bd;
+       int quiet;
+       char *s;

 #ifdef CONFIG_CMDLINE_TAG
        char *commandline = getenv ("bootargs");
 #endif

+       s = getenv ("quiet");
+       quiet = (s && (*s == 'y')) ? 0 : 1;
        theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep);

        /*
@@ -256,7 +260,11 @@
 #endif

        /* we assume that the kernel is in place */
-       printf ("\nStarting kernel ...\n\n");
+       if (quiet) {
+               printf ("\nStarting kernel at %08lx...",
+                               (ulong) theKernel);
+       }
+       printf("\n");

 #ifdef CONFIG_USB_DEVICE
        {
```

Figure 5.4 The Sample Patch of Silent Console

- **Kernel phase**

**METHOD 06: Use uncompressed kernel image**

For uncompressed kernel, we need to increase the size of mtdblock2 to put the uncompressed kernel, which is assigned in *linux/arch/arm/mach-omap1/board-osk.c* and *linux/include/asm-arm/sizes.h*. If the kernel has been optimized, the original size of mtdblock2 is enough to put the optimized uncompressed kernel. After using of uncompressed kernel, the time needed is reduced from 6282.1 ms to 5369.46 ms, i.e. 912.64 ms has been eliminated, and it is shown at Table 5.6 and the patch is shown at Figure 5.5. The noteworthy one is if we use uncompressed kernel with no image verification, the time will be reduced for 912.64+1026.1=1938.74 ms.

Table 5.6 The Time Reduced by Uncompressed Kernel

|  | Function block |  | Before (ms) | After (ms) |
|---|---|---|---|---|
|  | Start Point | End Point |  |  |
| Boot Loader | Device reset start | Device reset over | 31.38 | 31.38 |
|  | Device reset over | MPU read first instruction | 0.74 | 0.74 |
|  | MPU read first instruction | *env_relocate_spec* start | 122.26 | 122.22 |
|  | *env_relocate_spec* start | *env_relocate_spec* over | 44.98 | 45.18 |
|  | *env_relocate_spec* over | image date checksum start | 1505.38 | 1505.20 |
|  | image date checksum start | image date checksum over | 487.92 | 1026.10 |
|  | image date checksum over | copy image to ram start | 0.44 | 0.64 |
|  | copy image to ram start | copy image to ram over | 395.52 | 831.58 |
|  | copy image to ram over | transfer control to Linux | 0.90 | 0.90 |
| Kernel | transfer control to Linux | uncompress kernel start | 13.48 | 1805.52 |
|  | uncompress kernel start | uncompress kernel over | 1838.62 |  |
|  | uncompress kernel over | *jffs2_build_filesystem* start | 1840.48 |  |
| Amount |  |  | 6282.1 | 5369.46 |

```
diff -ruN linux-2.6.14.orig/arch/arm/boot/Makefile
linux-2.6.14/arch/arm/boot/Makefile
--- linux-2.6.14.orig/arch/arm/boot/Makefile    2006-06-15 21:57:19.000000000
+0800
+++ linux-2.6.14/arch/arm/boot/Makefile 2006-06-15 22:20:09.000000000 +0800
@@ -46,6 +46,10 @@
 $(obj)/Image: vmlinux FORCE
        $(call if_changed,objcopy)
        @echo '  Kernel: $@ is ready'
+       $(CONFIG_SHELL) $(MKIMAGE) -A arm -O linux -T kernel \
+       -C none -a $(ZRELADDR) -e $(ZRELADDR) -n 'Linux-$(KERNELRELEASE)' \
+       -d arch/arm/boot/Image arch/arm/boot/uImage-uncompress
+       @echo '  Kernel: arch/arm/boot/uImage-uncompress is ready'

 $(obj)/compressed/vmlinux: $(obj)/Image FORCE
        $(Q)$(MAKE) $(build)=$(obj)/compressed $@
diff -ruN linux-2.6.14.orig/Makefile linux-2.6.14/Makefile
--- linux-2.6.14.orig/Makefile  2006-06-15 22:02:41.000000000 +0800
+++ linux-2.6.14/Makefile       2006-06-15 22:29:54.000000000 +0800
@@ -984,7 +984,8 @@
 # Directories & files removed with 'make clean'
 CLEAN_DIRS  += $(MODVERDIR)
 CLEAN_FILES += vmlinux System.map \
-                .tmp_kallsyms* .tmp_version .tmp_vmlinux* .tmp_System.map
+                .tmp_kallsyms* .tmp_version .tmp_vmlinux* .tmp_System.map \
```

```
+                arch/arm/boot/uImage-uncompress

 # Directories & files removed with 'make mrproper'
 MRPROPER_DIRS  += include/config include2
```

Figure 5.5 The Patch of Uncompressed Kernel

**METHOD 07: Eliminate BogoMIPS calibration**

By last boot, we can obtain the value of *loops_per_jiffy* by *dmesg*, which is 373760. By put "*lpj*=373760" (passing 373760 to kernel as the value of *loops_per_jiffy*) in command line, the boot time is reduced from 154.785157 ms to 0.061036 ms, i.e. 154.724121 ms has been eliminated, and it is shown at Table 5.7 and the effect is shown at Figure 5.6.

Table 5.7 The Time of *calibrate_delay*

| normal boot | preset *loops_per_jiffy* |
|---|---|
| 154.785157 ms | 0.061036 ms |

```
Before presetting:
Calibrating delay loop... 74.75 BogoMIPS (lpj=373760)
After presetting:
Calibrating delay loop (skipped)... 74.75 BogoMIPS preset
```

Figure 5.6 The Effect of Preset LPJ

**METHOD 08: Use device modularization**

After using the unofficial patch for OMAP5912, we find still many choices can be modified. We remove these options: Code maturity level options, Support for paging of anonymous memory (swap), Reset unused clocks during boot, OMAP multiplexing support, IP kernel level autoconfiguration, Initial RAM disk (initrd)

support, ATA/ATAPI/MFM/RLL support, Mouse interface, Keyboards, Virtual terminal, Unix98 PTY support, Legacy (BSD) PTY support, Second extended fs support, Inotify file change notification support, Dnotify support, Kernel automounter support, MSDOS fs support and VFAT (Windows-95) fs support. And then we modularize these options: PCCard (PCMCIA/CardBus) support, Unix domain sockets, INET (socket monitoring interface), Loopback device support, PPP (point-to-point protocol) support, Texas Instruments TLV320AIC23 Codec, Hardware Monitoring support ,Support for frame buffer devices ,Kernel automounter version 4 support and NFS file system support. Finally, we choose the option: Configure standard kernel features (for small systems) to finish this part of works. Finally, by the initcall-times patch, we can obtain the value of initcalls. The time measured by initcall-times is accurate, which is the same as the value measured by oscilloscope and logic analyzer. The boot time is reduced from 1574.645998 ms to 448.913571 ms, i.e. 1125.732427 ms has been eliminated, and it is shown at Table 5.8.

Table 5.8 The Time of Initcalls

| Inincall name | Before (ms) | After (ms) |
|---|---|---|
| *customize_machine* | 4.516601 | 4.364014 |
| *omap_init_devices* | 1.983643 | 1.983642 |
| *init_bio* | 1.037598 | 0.946044 |
| *i2c_init* | 1.220703 | 1.373291 |
| *omap_i2c_init_driver* | 2.380371 | 2.380371 |
| *tps_init* | 11.138916 | 11.016846 |
| *chr_dev_init* | 8.331299 | 8.392334 |
| *param_sysfs_init* | 16.387940 | 9.460449 |
| *init_jffs2_fs* | 1.312256 | 1.220703 |
| *omapfb_init* | 32.745361 | - |
| *tty_init* | 70.251465 | 1.922607 |
| *pty_init* | 626.342774 | - |

| | | |
|---|---|---|
| *serial8250_init* | 245.086670 | 189.575195 |
| *noop_init* | 2.777100 | 2.899170 |
| *as_init* | 3.814697 | 3.723144 |
| *deadline_init* | 3.295898 | 3.234863 |
| *cfq_init* | 3.112793 | 2.960205 |
| *rd_init* | 23.284912 | 22.033691 |
| *loop_init* | 10.803223 | - |
| *net_olddevs_init* | 1.586914 | 1.403808 |
| *ppp_init* | 4.394531 | - |
| *smc_init* | 17.425537 | 17.211915 |
| *i2c_dev_ini* | 4.577637 | 4.394531 |
| *omapflash_init* | 59.356690 | 57.983399 |
| *omap_cf_init* | 273.498535 | - |
| *mousedev_init* | 5.187988 | - |
| *omap_kp_init* | 3.845215 | - |
| *omap_ts_init* | 2.441407 | 2.044677 |
| *inet_init* | 100.341797 | 96.527100 |
| *bictcp_register* | 1.861572 | 1.861572 |
| *af_unix_init* | 3.570556 | - |
| *omap1_late_clk_reset* | 26.733399 | - |
| Amount | 1574.645998 | 448.913571 |

**METHOD 09: Use silent console in kernel phase**

After using silent console, the time needed is reduced from 5882.6 ms to 5499.6 ms, i.e. 383 ms has been eliminated, and it is shown at Table 5.9.

Table 5.9 The Time Reduced by Silent Console in Linux kernel

| Function block | | Before (ms) | After (ms) |
|---|---|---|---|
| Start Point | End Point | | |
| transfer control to Linux | uncompress kernel start | 13.48 | 13.38 |
| uncompress kernel start | uncompress kernel over | 1838.62 | 1838.66 |
| uncompress kernel over | *jffs2_build_filesystem* start | 1840.48 | 1463.88 |
| *jffs2_build_filesystem* start | *jffs2_build_filesystem* over | 2179.54 | |

| *jffs2_build_filesystem* over | invoke *init* | 10.48 | 2183.68 |
|---|---|---|---|
| Amount | | 5882.60 | 5499.60 |

- **User space phase**

**METHOD 10: Simplify user space utilities**

We give up the most of archival utilities, editors and console utilities, because the usage of these utilities is not many. And we remove the utilities for user management, login/password management utilities and system logging utilities for single user mode. Finally, we add the Linux module utilities, web server, telnet server and some daemons. We have built a powerful file system, and the size of pure file system without modules is only 636.4 KB using JFFS2.

**METHOD 11: Accelerate shell prompt start**

After skip the wait for enter, the wait time, 600ms in average is reduced. The patch is shown at Figure 5.7.

```
diff -Nur busybox-1.01/init/init.c busybox-1.01-phantom-v2/init/init.c
--- busybox-1.01/init/init.c    2005-08-17 09:29:16.000000000 +0800
+++ busybox-1.01-phantom-v2/init/init.c 2006-07-11 06:13:37.000000000 +0800
@@ -429,12 +429,14 @@
        char *s, *tmpCmd, *cmd[INIT_BUFFS_SIZE], *cmdpath;
        char buf[INIT_BUFFS_SIZE + 6];  /* INIT_BUFFS_SIZE+strlen("exec ")+1 */
        sigset_t nmask, omask;
+/* skip press_enter */
+/*
        static const char press_enter[] =
 #ifdef CUSTOMIZED_BANNER
 #include CUSTOMIZED_BANNER
 #endif
                "\nPlease press Enter to activate this console. ";
-
+*/
        /* Block sigchild while forking.  */
        sigemptyset(&nmask);
        sigaddset(&nmask, SIGCHLD);
@@ -579,17 +581,18 @@
                        }
                }

+/*
+ * Save memory by not exec-ing anything large (like a shell)
+ * before the user wants it. This is critical if swap is not
+ * enabled and the system has low memory. Generally this will
```

```
+ * be run on the second virtual console, and the first will
+ * be allowed to start a shell or whatever an init script
+ * specifies.
+ */
+/*
 #if !defined(__UCLIBC__) || defined(__ARCH_HAS_MMU__)
                if (a->action & ASKFIRST) {
                        char c;
-                       /*
-                        * Save memory by not exec-ing anything large (like a shell)
-                        * before the user wants it. This is critical if swap is
not
-                        * enabled and the system has low memory. Generally this
will
-                        * be run on the second virtual console, and the first will
-                        * be allowed to start a shell or whatever an init script
-                        * specifies.
-                        */
                        messageD(LOG, "Waiting for enter to start '%s'"
                                                "(pid %d, terminal %s)\n",
                                          cmdpath, getpid(), a->terminal);
@@ -598,7 +601,7 @@
                                ;
                }
 #endif
-
+*/
                /* Log the process name and args */
                message(LOG, "Starting pid %d, console %s: '%s'",
                                  getpid(), a->terminal, cmdpath);
```

Figure 5.7　　The Patch of Quick Shell Prompt

**METHOD 12: Use complex file System**

Firstly, the comparison between JFFS2, CramFS and SquashFS is shown at Table 5.10.

Table 5.10　　The Comparison between Different FS

| | Writable FS | Read-only FS | |
|---|---|---|---|
| | JFFS2 | CramFS | SquashFS |
| Kernel size (KB) | 1721920 | 1627312 | 1660832 |
| FS image size (KB) | 1162272 | 1007616 | 1085440 |
| Mount time (ms) | 2179.54 | 8.62 | 6.98 |

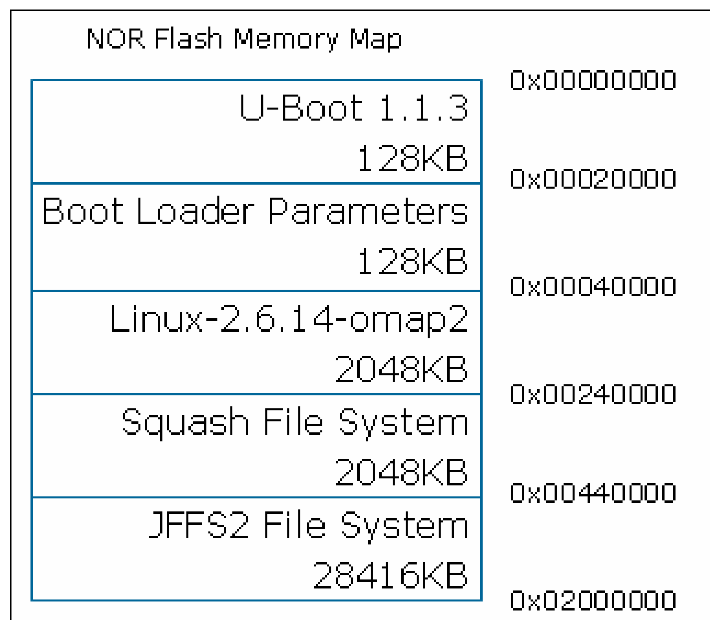



Figure 5.8 The NOR Flash Memory Map

Time/Div: 500ms
Search
C1: 1.59835s
C2: 3.93157s
Delta Time: 2.33322s
Lock Delta Time
DC_IN
C1: 1
C2: 1
Delta: 0
-52.410,000,000 ms
Sample
DC_IN
MPU_nRESET
RST_OUT
RS232C_Tx
nFlash_CSx

Figure 5.9 The Mount Operation of JFFS2 Partition in Background

And we implement a complex file system which includes SquashFS and JFFS2 FS to reduce the boot time greatly, and we still can do write operation on flash storage. The NOR flash memory map is shown at Figure 5.8, and the mount operation of JFFS2 partition in background is shown at Figure 5.9, the shell prompt is put at 1489.59 ms for user, and the JFFS2 partition is mounted at 3744.60 ms in background.

## 5.3 Reduced Functional Ability

We built the Table 5.11 to show the functional ability comparison between original boor and faster boot, the abilities and characteristics will show in it.

Table 5.11 Functional Ability Comparison

| | Original Boot | Faster Boot |
|---|---|---|
| Clocking mode | MPU and DSP work at 96 MHz during U-Boot phase | MPU and DSP work at 192 MHz during U-Boot phase |
| Console functions | U-Boot will provide a fully console device | The command *loadb* and the console outputs work ok |
| Abort boot function | U-Boot will wait for user for 1 second at least | U-Boot will wait for user for 200 ms at most |
| Image verification mechanism | Image verification mechanism is always ON | User can switch the mechanism himself |
| Silent console in U-Boot | U-Boot will output the normal information | U-Boot will output no information |
| Uncompressed Kernel | The compressed image size is less than 1 MB and kernel has been uncompressed before start | The image size is more then 1MB and less than 2.2MB, and kernel hasn't been uncompressed before start |
| BogoMIPS calibration | Kernel compute BogoMIPS and *loops_per_jiffy* every boot | Kernel get the *loops_per_jiffy* from command line |
| Device Modularization | Kernel will initiate all usable device during kernel phase | Some modularized devices have been initiated before using them |

| Silent console in kernel | Kernel will output the normal information | Kernel will output no information during boot, and store information in kernel ring buffer |
|---|---|---|
| User space utilities | Many useful utilities will be supplied, some are similar | Limited utilities for specific requirement |
| Shell prompt start | Saving memory and wait for user to enable shell prompt | Shell prompt will use 84 or 96KB of memory and user can user shell prompt soon |
| File system | It will waste 2 to 3 seconds to build the JFFS2 file system during boot | JFFS2 file system will be built in background and save 2 to 3 seconds during boot |

## 5.4 Recommendation

In Section 5.4, we provide a comparison of original and faster boot. Certainly, some method for faster boot will reduce the boot time and also the functional abilities. Therefore, we need to do some choices to meet the balance between boot time and functional abilities.

We recommend that the methods of U-Boot phase must be used, because the methods of reducing the time during U-Boot doing boot process are functional lossless. Regarding the console information and image verification mechanism, we can use the switch to meet the requirement. You only need to do image verification again to determine if you shall re-burn the kernel image after system crash and the stability of system is not acceptable.

The implement of uncompressed image is according to the total size of boot loader, kernel and file system. Sometimes, you must upgrade the flash because the original flash capability can not meet the requirement of uncompressed kernel image. For example, some specific product which has only 2 MB NOR flash, after reducing useless modules and utilities of user space, the total size of U-Boot, boot loader parameter, uncompressed kernel and root file system can be limited less than 2 MB.

Therefore, using uncompressed kernel is feasible. On the other hand, in order to provide multi-function and support more devices, a 4MB NOR flash is necessary for a complete embedded operating system. At present, the price of 2Mbytes NOR flash (Intel JS28F160C3TD70) is US$ 1.596 (price break is 1000) [22], and the price of 4Mbytes NOR flash (Intel JS28F320C3TD70) is US$ 3.0457 (price break is 1000) [23], you must pay more costs for this method.

Device modularization is limited by the requirement of specific product, if the product is required to provide all useable devices immediately after boot; the effect of device modularization is restricted. For example, a smartphone should let the function of communication usable immediately after boot, and let other functions be initiated in background. And other methods of reducing the time during kernel phase are functional lossless.

If the product need not to store extra data, using a read-only file system is recommendable, otherwise using a complex file system can meet the requirement of faster boot and data storage. If the product need no shell prompt, the method of quicker shell prompt should not be used. If the product is required to provide complete utilities; the effect of the part of user space utilities is restricted.

## 5.5 Summary

In Chapter 5, we implement many methods which reduced the boot time from boot loader phase to user space phase. Our optimized U-Boot 1.1.3, suggested Linux kernel 2.6.14 configuration, and optimized BusyBox 1.01 are usable immediately and modification of hardware is needless.

# Chapter 6

# Conclusion

There are many mobile devices and high-level consumer electronics on the market, such as Smart Phone, Palm PDA, Pocket PC and Camera Phones, etc; they are very convenient and powerful. However, the boot time of them is generally 8~10 seconds. Most of people are impatient; they want the boot time faster and faster.

Therefore, we try to improve the boot time of these products which are based on embedded platform. We choose the OMAP5912OSK running embedded Linux as our experiment platform. OMAP5912OSK based on the dual core processor of ARM926EJ-S MPU and TMS320C55x DSP, 32MB NOR flash and 32MB DDR SDRAM; they are useful or development.

We study the source code of U-Boot, Linux kernel and BusyBox, and try to measure the boot time by software tools and hardware tools, such as KFT, Printk Times, initcall-times patch, expect script, oscilloscope and logic analyzer. After obtaining the time measurement results, we subdivide the total boot time to 18 function blocks, and find out the long execution time operation in each function block. By hack related source code, we either simplified by rewriting the codes or even skipped without any side effect to reduce the execution time.

Finally, we optimize the U-Boot 1.1.3 and BusyBox, and suggest a fast boot Linux kernel 2.6.14 configuration. With read-only SquashFS file system, the boot time is only 1477.77 ms. With our complex writable file system, the boot time is only 1598.35 ms. By our method , the boot time of mobile devices and high-level consumer electronics can also be reduced, people will like these.

# Chapter 7

# Future Work

We already achieved the fast boot on OMAP5912OSK; we believe that our methods can be implemented on other platform. And we will try to build a timing equation for fast boot by analyze more detailed time measurement results. By that timing equation, we can obtain the boot time by related parameter of specific platform.

# Reference

[1] Keun Soo Yim, Jihong Kim, and Kern Koh, "A Fast Start-Up Technique for Flash Memory Based Computing Systems," Proceedings of the ACM Symposium on Applied Computing, 2005

[2] Zoltán Sógor, Ferenc Havasi, "A JFFS2 Analysis (draft version)," University of Szeged, 2005

[3] The Consumer Electronics Linux Forum, "Kernel Execute-In-Place," http://tree.celinuxforum.org/CelfPubWiki/KernelXIP

[4] Jimmy Wennlund, "Next Generation Init System – InitNG," http://www.initng.org/

[5] Texas Instruments, "OMAP5912 Applications Processor (Rev. E)," http://www-s.ti.com/sc/ds/omap5912.pdf

[6] Wolfgang Denk, "Das U-Boot - Universal Bootloader," http://sourceforge.net/projects/u-boot/

[7] Linus Torvalds, "The Linux Kernel Archives," http://www.kernel.org/

[8] Tony Lindgren, "Unofficial OMAP-1510/1610 Linux patches," http://www.muru.com/linux/omap/

[9] Rob Landley, "BusyBox - The Swiss Army Knife of Embedded Linux," http://www.busybox.net/

[10] Tim R. Bird, "Methods to Improve Boot Time in Linux," Proceedings of the Ottawa Linux Symposium, Sony Electronics, 2004

[11] Alessandro Rubini, Jonathan Corbet, "Linux Device Drivers, Second Edition," O'Reilly Media, Inc., 2001

[12] The Consumer Electronics Linux Forum, "Kernel Function Trace,"

http://tree.celinuxforum.org/CelfPubWiki/KernelFunctionTrace

[13] The Consumer Electronics Linux Forum, “Printk Times,”
http://tree.celinuxforum.org/CelfPubWiki/PrintkTimes

[14] Don Libes, “Exploring Expect,” O'Reilly Media, Inc., 1994

[15] Columbia University, “C-Kermit 8.0,”
http://www.columbia.edu/kermit/ck80.html

[16] Robert Love, “Linux Kernel Development (2nd Edition),” Novell Press, 2005

[17] Texas Instruments, “OSK5912 Board Design Guide,”
http://www-s.ti.com/sc/psheets/spru715/spru715.pdf

[18] Texas Instruments, “OMAP5912 Multimedia Processor OMAP3.2 Subsystem Reference Guide (Rev. B),”
http://www-s.ti.com/sc/psheets/spru749b/spru749b.pdf

[19] Texas Instruments, “OMAP5912 Applications Processor Silicon Errata (Rev. I),”
http://focus.ti.com/lit/er/sprz209i/sprz209i.pdf

[20] The Consumer Electronics Linux Forum, “Preset LPJ,”
http://tree.celinuxforum.org/pubwiki/moin.cgi/PresetLPJ

[21] The Consumer Electronics Linux Forum, “Calibrate Delay Avoidance Specification R2,”
http://tree.celinuxforum.org/pubwiki/moin.cgi/CalibrateDelayAvoidanceSpecification_5fR2

[22] Digi-Key Corporation, “Digi-Key Part Number：866265-ND,”
http://www.digikey.com/scripts/dksearch/dksus.dll?Detail?Ref=26333&Row=247793&Site=US

[23] Digi-Key Corporation, “Digi-Key Part Number：864838-ND,”
http://www.digikey.com/scripts/dksearch/dksus.dll?Detail?Ref=26333&Row=247704&Site=US