ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 5, May 2013

LINUX BOOT TIME OPTIMIZATION – FTP SERVER

Rahul Tiwari[1], Maulik Patel[2]

M.Tech Student (VLSI and EMBEDDED SYSTEM), Dept. of E.C, U.V.Patel college of Engineering, Mehsana, Gujarat, India[1]

M.Tech Student (VLSI and EMBEDDED SYSTEM), Dept. of E.C, U.V.Patel college of Engineering, Mehsana, Gujarat, India[2]

ABSTRACT*:* One of the problems about operating system is the speed at which it boots. Linux is a general purpose operating system. Today it can be used everywhere. It can be used as client or server right out of the box. Many variations of Linux are currently being used in various real time mission critical systems which require a very high degree of availability and minimal downtime during system upgrades and in consumer electronic products in which user except their devices to be available for use very soon after being turned on . This leads to optimization of boot time for Linux. In this paper we represent all techniques available for optimization. We describe the boot process under the Linux, initial boot time and by using some of available technique how to reduce boot time for particular application e.g. FTP Server in our case.

I. INTRODUCTION

Linux meets the requirements of everyone in all fields such as embedded, real time, personal computer in terms of functionality, scalability and cost. Linux supports all these features because of its configurable nature. Linux is world‘s biggest and longest evolved software system.

Boot time i.e. the time taken by the system to show its "availability" since the power button was pushed on, is a becoming a key differentiator in the usability factor.

The definition of availability varies across the devices.
For example:

Appearance of home screen for devices containing a display e.g. cell phone, media player.

An audible tone / LED turning on or changing colour for devices without display Appearance of shell prompts on development systems with the console.

The important point to understand is that optimization of boot time should not compromise the system’s existing functionality and stability by any degree but in turn help the system to enhance its booting process for faster system upgrade. Before optimizing, first we have to understand the boot process, measure the initial time and optimize it by using different reduction techniques.

II. LINUX BOOT PROCESS

A. Boot Process Flow

The process of booting a system running the Linux consists of multiple stages. Much of the flow of this process is similar whether we are booting an x86 based personal computer or an embedded system based on any architecture. The entire boot process of Linux can be roughly divided into 3 main areas firmware (boot-loaders), kernel, and user space. The beginning of the boot process varies depending on the hardware platform being used. However, once the kernel is found and loaded by the boot loader, the default boot process is identical across all architectures.

ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 5, May 2013

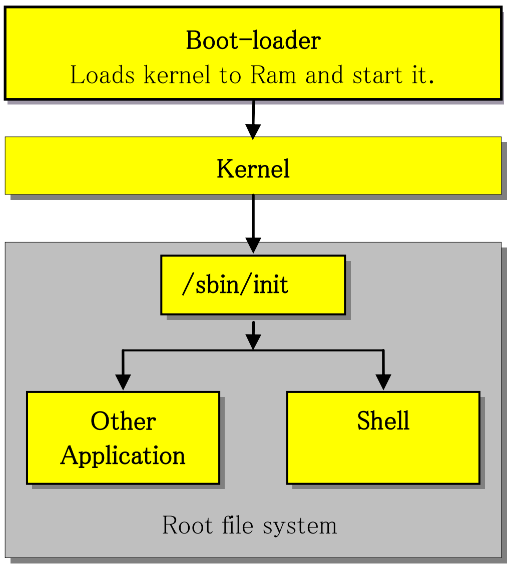


Fig. 1 Boot Process Flow

The following is a list of events during a typical boot sequence:

1. Power on
2. Firmware (boot-loader) starts
3. The kernel decompression starts
4. The kernel start
5. User space start
6. RC script start
7. Application start

B. Booting Components

1) Boot-loader: Depending on the system's architecture, the boot process may differ slightly.

An embedded platform uses a bootstrap program (for example: U-Boot, Red-Boot and Micro Monitor from Lucent). These programs shipped along with the embedded platforms and stay in a specific region of flash memory on the target hardware. On the other hand, in a PC, BIOS resides at address 0xFFFF0 and the first step of BIOS is Power on Self Test (POST). POST can vary from manufacture to manufacture (Asus, Mercury and Intel) and version to version, because there is no standard exists.

There are several stages of boot-loaders that perform different levels of initialization on an OMAP platform, in order to eventually load and run the file system.

X-loader:

The x-loader is a small first stage boot-loader derived from the u-boot base code. It is loaded into the internal static RAM by the OMAP ROM code. Due to the small size of the internal static RAM, the x-loader is stripped down to the essentials. The x-loader configures the pin muxing, clocks, DDR, and serial console, so that it can access and load the second stage boot-loader (u-boot) into the DDR.

U-boot:

The u-boot is a second stage boot-loader that is loaded by the x-loader into DDR. It comes from Das U-Boot. The u-boot can perform CPU dependent and board dependent initialization and configuration not done in the x-loader.

ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 5, May 2013

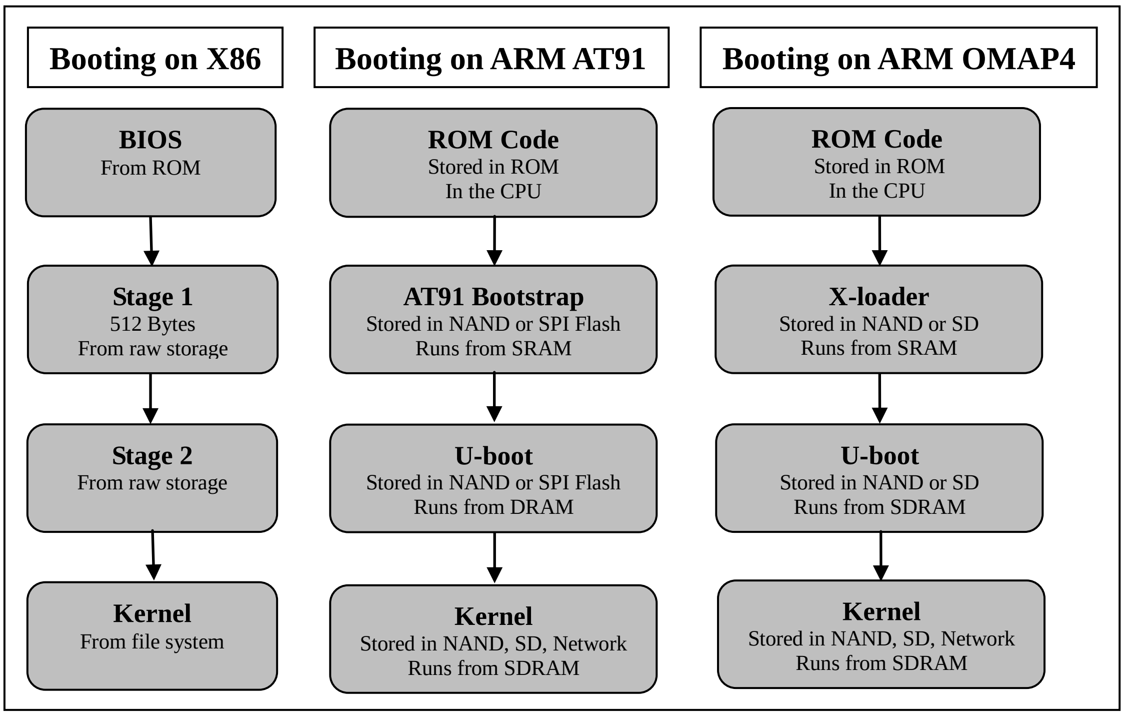


Fig. 2 Boot Sequence on Different Platform

2) Kernel: A kernel is a central component of an operating system. It acts as an interface between the user applications and the hardware. The sole aim of the kernel is to manage the communication between the software (user level applications) and the hardware (CPU, disk memory, etc.).
   The main tasks of the kernel are: Process management, Device management, and Memory management, interrupt handling, I/O communication, File system, etc. During boot process, The kernel initializes devices, mounts the root file system specified by the boot loader as read only, and runs Init (/sbin/init) which is designated as the first process run by the system (PID = 1).

3) File system: A file system is an organization of data and metadata on a storage device. There are many types of file system and media. With all of this variation you can except the Linux file system interface is implemented as a layered architecture.
   A file system must be present in the kernel to start successfully. It can be an in memory file system, network file system and can be attached to the kernel image loaded into memory.

FTP Server:
What is a server?
The server is nothing but a common place (e.g. A common folder on the computer), where shared data is available. We can configure it as our requirement. E.g. all users can read and download data but only one user can upload on it.

FTP server is a File server. FTP server is used to transfer files between server and clients. All major operating system supports FTP. FTP is the most used protocol over internet to transfer files. Like most Internet operations, FTP works on a client/ server model. FTP client programs can enable users to transfer files to and from a remote system running an FTP server program.

III. HARDWARE AND SOFTWARE PLATFORM

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

A. Panda Board

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

As a hardware platform for this project, Panda board, a particular variant of the OMAP4 platform has been chosen. The small and support of many peripherals makes it an ideal object for this project. The main processing core is a

ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 5, May 2013

powerful Cortex-A9 clocked at 1 GHz which complies with the ARM Architecture. The OMAP4430 system-on-chip (SoC) microprocessor has a variety of typical embedded system, like a real-time clock (RTC), DSP sub-system, Display sub-system, and Audio back-end (ABE) sub-system. The device also integrates On-chip memory; External memory interfaces System and connecting peripherals such as on-board Ethernet, LCD display controller, Zig-bee, Bluetooth, HDMI and DVI ports. In addition to these on-chip components, it has 1 GB low power DDR2 RAM and General purpose expansion header (I2C, GPMC, USB, MMC, DSS and ETM).

B. Software Platform

The software platform for this project was built with the help of the build-root. This includes everything needed to work with a Linux based computer system. It contains GCC-compiler, boot-loaders, kernel, and lots of additional libraries and useful tools like busy-box. We can also build manually by using make utility. First download the general source coder for boot-loaders and kernel and then cross-compile it to generate boot-loader images e.g. MLO, U-boot.bin and kernel image uImage.

Typical Starting Point:
Boot-Loader: U-Boot (2011.12)
OS: Linux (3.5.4)
File system: Build-root (2012.13)
Application: FTP Server

IV. MEASUREMENT AND OPTIMIZATION

A. Initial Measurement:

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

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

X-loader
First newline character received on the serial console indicates start of x-loader.
U-boot
The banner containing the U-Boot version indicates the start of u-boot:
```
U-Boot 2010.06 (Apr 16 2011 - 15:22:19)
```
Linux kernel
First line after this indicates the start of Linux kernel:
```
Uncompressing Linux... done, booting the kernel.
```
File System
```
INIT: version 2.86 booting
```

Applicability of different boot time measurement techniques can be classified into three stages: System wide time Measurement, Kernel invocation time Measurement, User space time measurement.

In the system wide time measurement stage, the boot time of the entire booting process is measured. System wide measurement tools are **grabserial** and **uptime**. The kernel measurement tools determine the time spent in various kernel functions and subroutines during kernel invocation**.** Major tools for determining kernel invocation time are **Printk–Times**, **Initcall_debug** and **KFT.** User space time measurement depicts the total time spent from starting of init process to the state where the system is operational. The user space measurement tools are **Boot chart** and **Strace.**

We are use following two techniques for measuring total boot time.

1) Grabserial

There is a very useful program called grabserial, written by Tim Bird and available from http://elinux.org/Grabserial. It is a Python script that adds a time stamp to each string received from a serial port. The target I am using has a serial port for the console and general debugging, so We will use that. Most embedded devices have such a thing.

Usage:

```
grabserial -v -d "/dev/ttyS0" -b 115200 8  30 -t -m  "/#*"
-d        <serial device>
-b        <baudrate>
-m        <match pattern that will reset time stamps>
-t        <time in seconds indicating howmuch time it will run >
```

2) PRINTK_TIME
   Enable following option in kernel configuration:
   Kernel Hacking → Show timing information on printk.

B. Optimization:

Areas of optimization would fall into either of these categories:

1. Size
   Reduce the size of binaries for each successive component loaded.
   Remove features that are not required.
2. Speed
   Optimize for target processor.
   Use faster medium for loading primary, secondary boot loaders and kernel.
   Reduce number of tasks leading to the boot.
   Remove features that are not required.

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

Boot-loader speedups

Kernel XIP: kernel image to be executed in-place in ROM or FLASH.
DMA Copy of Kernel on Start-up: Copy kernel image from Flash to RAM using DMA.
Uncompressed kernel: use uncompressed kernel image because uncompressed kernel might boot faster.
Fast Kernel Decompression: use fast decompression technique.

Kernel speedups

Disable Console: Avoid overhead of console output during system startup.
Disable bug and printk: Avoid the overhead of bug and printk. Disadvantage is that you lose a lot of info.
Preset LPJ: the use of a preset loops_per_jiffy value.
Reordering of driver initialization: Driver bus probing to start as soon as possible.
Deferred Initcalls: Defer non-essential module initialization routines to after primary boot.

User-space and application speedups

Optimize RC Scripts: Reduce overhead of running RC scripts.
Parallel RC Scripts: Run RC scripts in parallel instead of sequentially.
Application XIP: Allow programs and libraries to be executed in-place in ROM or FLASH.
Pre Linking: Avoid cost of runtime linking on first program load.

Statically link applications. This avoids the costs of runtime linking. Useful if you have only a few applications. In that case it could also reduce the size of your image as no dynamic libraries are needed GNU_HASH: ~ 50% speed improvement in dynamic linking.

Avoid udev: it takes quite some time to populate the /dev directory. In an embedded system it is often known what devices are present and in any case you know what drivers are available, so you know what device entries might be needed in /dev. These should be created statically, not dynamically. mknod is your friend, udev is your enemy.

If your device has a network connection, preferably use static IP addresses. Getting an address from a DHCP server takes additional time and has extra overhead associated with it.

Moving to a different compiler version might lead to shorter and/or faster code. Most often newer compilers produce better code. You might also want to play with compiler options to see what works best. If possible move from glibc to uClibc. This leads to smaller executable and hence to faster load times.

1) Boot loader optimization:

In boot loader source code, lots of configuration file available specific to different platform in "**include/configs**". As I use panda board as a platform. A specific configuration file for this platform is also available which is "**omap4_panda.h**".

ISSN (Print) : 2320 – 3765
ISSN (Online): 2278 – 8875

International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering
Vol. 2, Issue 5, May 2013

So whatever changes we want to do, it should be done in this file.

- **Setting Boot delay parameter**
  By default boot delay defined in the configuration file is 3. We can reduce it to 1 or 0; if we put is 0 then Auto boot will happen. If we want to change some boot parameter or boot-args then it should be set to 1 and more. By doing this we save 3.10 second of total boot-time.
- **Setting following environment variable**

  1. Verify
  Before execute image is verified. This should be not necessary in small system so we turned off verification of image by setting environment variable verify to off. By setting this parameter we save **0.05 to 0.06** second.

  ```
  verify = n
  ```

  2. Silent
  Turning the prints off is a two step process.
  Add the following line to board specific configuration.

  ```
  #define CONFIG_SILENT_CONSOLE   1
  ```

  Set the environment variable silent to 1.

  ```
  silent = 1
  ```

  By setting this parameter we can save **0.02 to 0.04** second.

  3. ip_method
  This will bypass the kernel Network configuration while still allowing configuring network in user space.

  ```
  ip_method=off
  ```

  If it is not set then network will be configured during kernel initialization and it take around **7 to 8** second. So by setting this parameter we can save this much of time.
- **Removing unnecessary support**
  U-boot also does some device initialization and it has some devices support.
  E.g. if you want all function except of Network support you can write

  ```
  #include "config_cmd_all.h"
  #undef CONFIG_CMD_NET
  ```

  Here we don't require USB support so we remove USB support.
  Avoid long help text for u-boot commands

  ```
  #undef CONFIG_SYS_LONGHELP
  ```

  By doing this, here we save **0.11** second.

2) Kernel optimization:

After completion of first phase of execution (Boot-loader Phase), at the end of its execution, it call the kernel image to start kernel during boot process.
Kernel image is a binary image which is generated after compiling the kernel.
Like boot-loader, kernel also contains lots of configuration file specific to platform at following path **"arch/arm/configs/omap2plus_defconfig"**.

So we can directly use the configuration file related to our target platform or configure manually using following commands:

```
make menuconfig
make xconfig
```

Kernel contains lots of configuration option e.g. it has several file system support, lot's of generic device driver's support, different network protocols and some general configuration. So it depend on me what we require and what is not.
As we use the panda-board as a target platform. We need to configure kernel for the panda-board. In the kernel configuration files, it has a configuration file named "omap2plus_defconfig" specific to our platform. So we configure the kernel using this file and making kernel image by following way:

```
make omap2plus_defconfig.
make uImage
```

Because of lot's of configurations available in the kernel, the size of the kernel is big or varies according its configuration. For optimization purposes we require a kernel with the small size. The kernel with its small size boots faster than with big size.

As we make an FTP server on Panda board. We require only network related stuff not all the options available on panda board. So we configure the kernel with the smallest configuration which it must require.
We know that small kernel will load faster so we have to reduce the size of kernel as small as possible. Here we try to reduce the size of kernel by using following technique from the available ones to reduce size.

We use following techniques for kernel optimization:

Fast Kernel Decompression:

We have three decompression techniques available: GZIP, LZO, and LZMA.

The following table show timing for different decompression technique for an optimized kernel.

TABLE 1

COMPRESSION TECHNIQUE'S RESULTS

Compression Technique	GZIP	LZMA	LZO
Size(MB)	2	1.5	2.2
Un-compress time(ms)	13.6	10.60	15
Kernel initial Time(ms)	912.55	1463.60	655
Total Time(ms)	926.15	1474.2	670

Disable Console:

Here we disable serial console on which we get the kernel messages. This can be done by defining parameter "quiet" in kernel command line.

By defining this we save **0.13 to 0.15** second.

Avoiding Calibration Delay:

This is one type of delay which is calculated each time the kernel boot. To avoid this calculation every time we define "lpj= ", the value which we get first time. So next time when it boot, it doesn't require calculating again.

By defining this parameter we save around **200 to 300** millisecond.

Reordering of driver initialization:

In a Linux tree, **54%** code of its tree is related to drivers. Similarly a nearly **20%** code is related to Architecture specific and **7%** code is related to file system. Here by removing unnecessary driver support or if require taking it as a module.

By doing this we reduce the size of kernel.

After applying all above techniques we reduce the kernel size from 4.0MB to around 2.2MB which causes the reduction in time around **1 to 1.5** second.

3) File system optimization:

The kernel initializes devices, mounts the root file system specified by the boot loader as read only, and runs Init (/sbin/init) which is designated as the first process run by the system (PID= 1). A message is printed by the kernel upon mounting the file system, and by Init upon starting the Init process.

Init is the father of all processes. Its primary role is to create processes from a script stored in the file /etc/inittab. This file usually has entries which cause init to spawn gettys on each line that users can log in. It also controls autonomous processes required by any particular system.

Init's job is "to get everything running the way it should be" once the kernel is fully running. Essentially it establishes and operates the entire user space. This includes checking and mounting file systems, starting up necessary user services, and ultimately switching to a user-environment when system start-up is completed.

File system can be made from scratch or we can customize an available minimal file system. We can also make file system using utility e.g. build root or from the busy-box. Both busy-box generated and build-root generated file system uses the busy box init. I am using file system generated by build-root.

First we download the minimal file system. Then put it into our board and test it but we don't get console. So we set the console, e.g. ttyO2 in place of ttyS0 in etc/inittab file again test it also we observe the time consuming process in it.

It takes more time in following areas:

mounting the file system

A file system which has a small size and type of read only e.g. cramfs, jffs, ubifs has mount faster compared to other file system e.g. ext3, ext4.

In etc/fstab file do proper mounting by uncommenting some lines and commenting not require lines.

By doing this we save **0.5 to 1** second.

In udev daemon

udev is a generic kernel device manager. It runs as a daemon on a Linux system and listens (via Net-link socket) to events the kernel sends out if a new device is initialized or a device is removed from the system. The system provides a set of rules that match against exported values of the event and the properties of the discovered device.

While it is possible to run a Linux system without udev, it is not recommended and is usually only done in mobile or embedded Linux implementations to speed up booting and go easy on memory.

mdev is a light-weight alternative to udev for used in embedded devices. Both handle the creation of device files in /dev and starting of actions when certain events happen with mdev we observe that it takes less time compared to udev

By replacing udev with mdev we save **1.5 to 2.5** second.

Using necessary services:

In systemVinit, it starts some service before login console and after entering run-level. Suppose it enters into run-level 5 then it execute services defined in rc5 directory. Then swan a getty.

E.g. tarting network, Starting telnet daemon, Starting syslogd/klogd.

Here we can start service specific to our requirement not all services should be started. Here the services which require immediately after boot can be started during boot so we can use it immediately after login. We can also start service later means after booting.

As we want to make an FTP server so our networking services should be useable after login. Here we need to start network services during boot time.

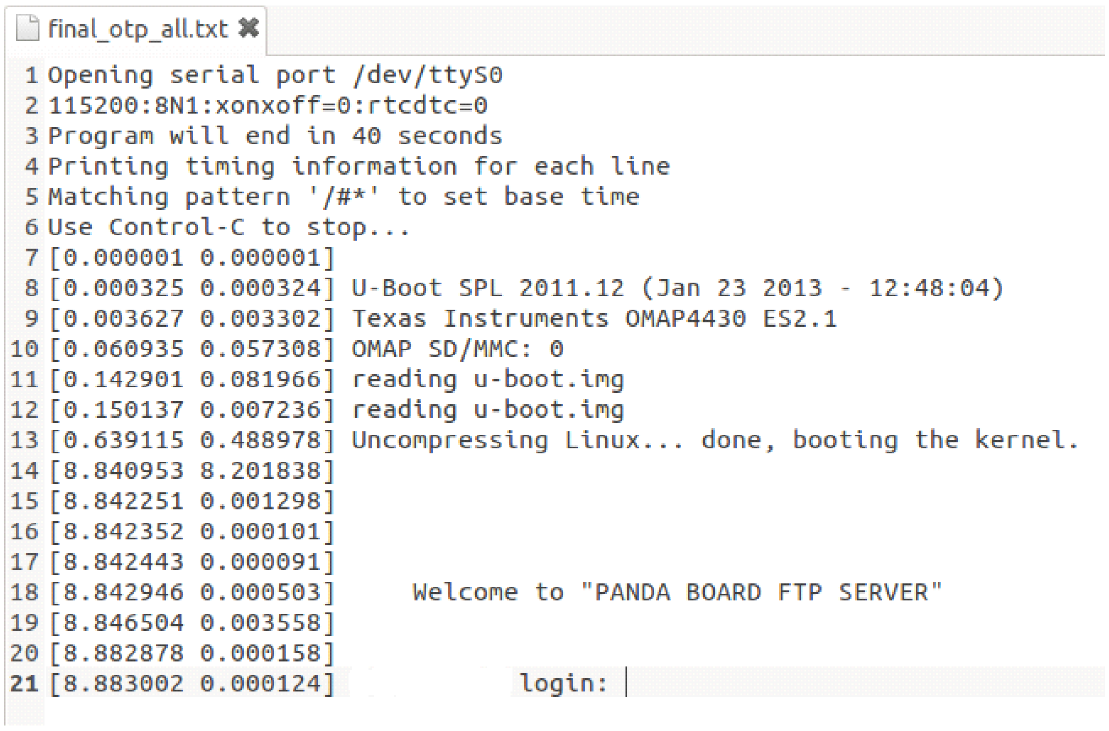
By doing this we save **0.064 to 0.1** second.

Using build-root file system

The file system using build root is generated by following steps

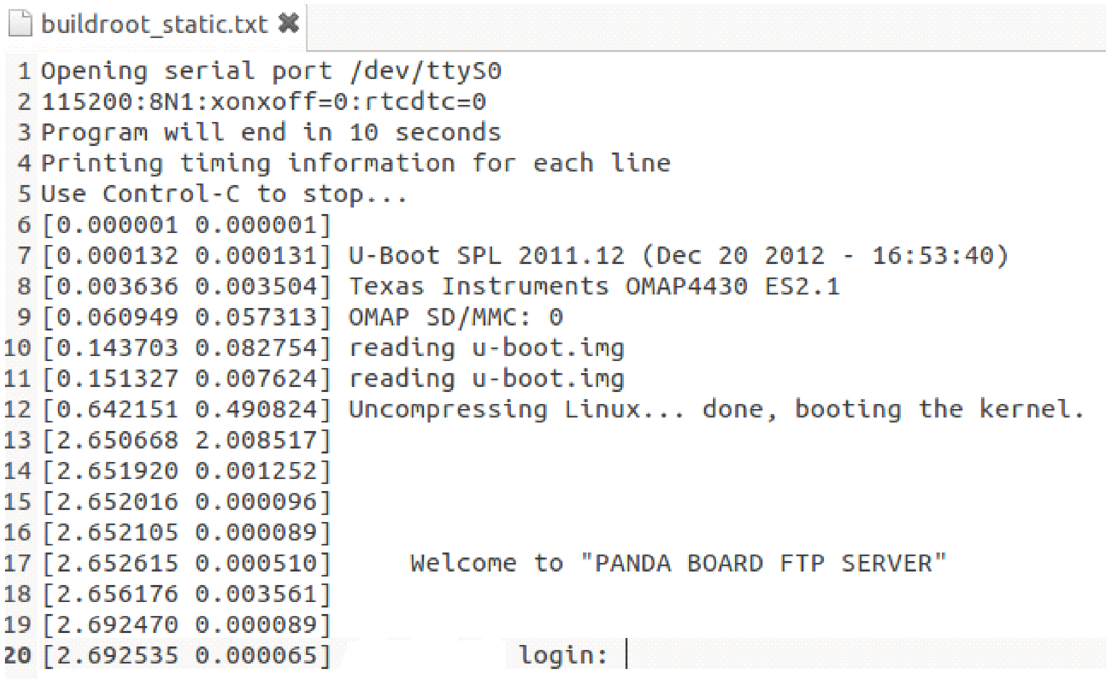
- Download the build root source code.
- Check the board specific configuration file if available then do follow:
  make panda_defconfig
- Otherwise we can configure it manually by following step:
  make menuconfig
- Configure it as our requirement like use busy-box init, use mdev in place of udev , start only necessary application. Configure and install necessary package require for networking especially for a FTP server then save the configuration.
- At last compile it by following step:
  make
- After compilation, file system is available at /output/images/rootfs.tar

After whole optimization:


final_otp_all.txt

```
Opening serial port /dev/ttyS0
115200:8N1:xonxoff=0:rtcdtc=0
Program will end in 40 seconds
Printing timing information for each line
Matching pattern '/#*' to set base time
Use Control-C to stop...
[0.000001 0.000001]
[0.000325 0.000324] U-Boot SPL 2011.12 (Jan 23 2013 - 12:48:04)
[0.003627 0.003302] Texas Instruments OMAP4430 ES2.1
[0.060935 0.057308] OMAP SD/MMC: 0
[0.142901 0.081966] reading u-boot.img
[0.150137 0.007236] reading u-boot.img
[0.639115 0.488978] Uncompressing Linux... done, booting the kernel.
[8.840953 8.201838]
[8.842251 0.001298]
[8.842352 0.000101]
[8.842443 0.000091]
[8.842946 0.000503]     Welcome to "PANDA BOARD FTP SERVER"
[8.846504 0.003558]
[8.882878 0.000158]
[8.883002 0.000124]               login:
```

Fig. 3 Linux boot time after optimization (DHCP IP)



```
buildroot_static.txt
Opening serial port /dev/ttyS0
115200:8N1:xonxoff=0:rtcdtc=0
Program will end in 10 seconds
Printing timing information for each line
Use Control-C to stop...
[0.000001 0.000001]
[0.000132 0.000131] U-Boot SPL 2011.12 (Dec 20 2012 - 16:53:40)
[0.003636 0.003504] Texas Instruments OMAP4430 ES2.1
[0.060949 0.057313] OMAP SD/MMC: 0
[0.143703 0.082754] reading u-boot.img
[0.151327 0.007624] reading u-boot.img
[0.642151 0.490824] Uncompressing Linux... done, booting the kernel.
[2.650668 2.008517]
[2.651920 0.001252]
[2.652016 0.000096]
[2.652105 0.000089]
[2.652615 0.000510]     Welcome to "PANDA BOARD FTP SERVER"
[2.656176 0.003561]
[2.692470 0.000089]
[2.692535 0.000065]           login: |
```

Fig. 4 Linux boot time after optimization (Static IP)

V. CONCLUSION

From above result and experiments, we conclude that Reduction and Boot Time depends on different reduction technique which we used for the application. We can even reduce more time by applying more techniques which will use on different platform. So, finally we say that Reduction in Boot Time is varies by different application.

ACKNOWLEDGEMENT

We thank to our project guide, Mr. Hasteen Patel and Mrs. Ekata Mehul, for providing necessary facilities towards carrying out this work. We are also very thankful to all our friends (Hardik Sheth, Kalpik Patel, Chirag Patel, Tejas Patil, Jiju Chako, Samarth Parikh, Shashank Pathak) for all the thoughtful and mind stimulating discussion we had, which prompted us to think beyond the obvious.

REFERENCES

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