

Optimize Linux Boot Time



Optimizing boot time on Linux based systems

Sanjeev Premi

Contents

Introduction

Objectives

Use-case 1: Boot from SDHC Card

Use-case 2: Boot from NAND

Understanding the boot sequence

Measuring boot time

Identify markers

Profiling the bootloaders

Profiling the Linux Kernel

Enable timestamps on kernel prints

Instrument kernel initialization

Areas of optimization

Techniques for optimization

Shedding the bulk

x-loader

u-boot

Linux kernel

Deferred initialization

Build for product

x-loader

u-boot

Linux kernel

Filesystem

Be quiet

x-loader

u-boot

Linux kernel

Avoid duplication

Leverage the SoC speed

Don't forget to remove instrumentation

Doing it yourself

Getting sources

x-loader

U-Boot

Linux Kernel

Building sources

Build x-loader

Build u-boot

Build Linux kernel

Build Linux kernel modules

Install Linux kernel modules

Pre-built binaries

What is current status?

Comparing size of binaries

Comparing boot times

Before and After Demo

What next?

Short term

Long term

More things to try

Frequently Asked Questions

References

Introduction

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

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

- Appearance of *home screen* for devices containing a display e.g. cellphone, media player
- An audible tone / LED turning on or changing color for devices without display
- Appearance of shell prompt on development systems with console.

As these examples suggest, there isn't an objective measure that can be used across. This article doesn't attempt to define boot-time. It recognizes the fact that boot-time must be measured in context of the device, its intended usage and associated user expectations.

Objectives

Specific usage of term 'optimizing' - instead of 'reducing' - sets the overall direction for this article. The reduction can be achieved by adding hacks/ quirks/ taking custom shortcuts - that are difficult to maintain across component versions. Optimizations are generic and easily maintainable.

This article describes a typical boot sequence and identifies the opportunities for optimization. It also walks through different techniques that can be used to optimize the boot time. These optimizations are illustrated with specific patches that *currently* apply against the latest PSP 04.02.00.07 release for OMAP35x and Sitara AM37x devices.

This article is augmented with actual patches across the boot-loader(s) and Linux kernel to illustrate the techniques discussed here. Therefore, discussion is limited to select techniques and changes. The commit messages in the individual patches provide more detailed description. The differences between SDK and an end-product will be visible in these patches as well.

- The SDK tends to be generic and inclusive. It is intended to demonstrate maximum features available on the platform and tools to leverage these features.
- An end-product is specific and exclusive. It implements only a defined set of use-cases.

To provide a direction to the optimizations, these use-cases are being defined:

Use-case 1: Boot from SDHC Card

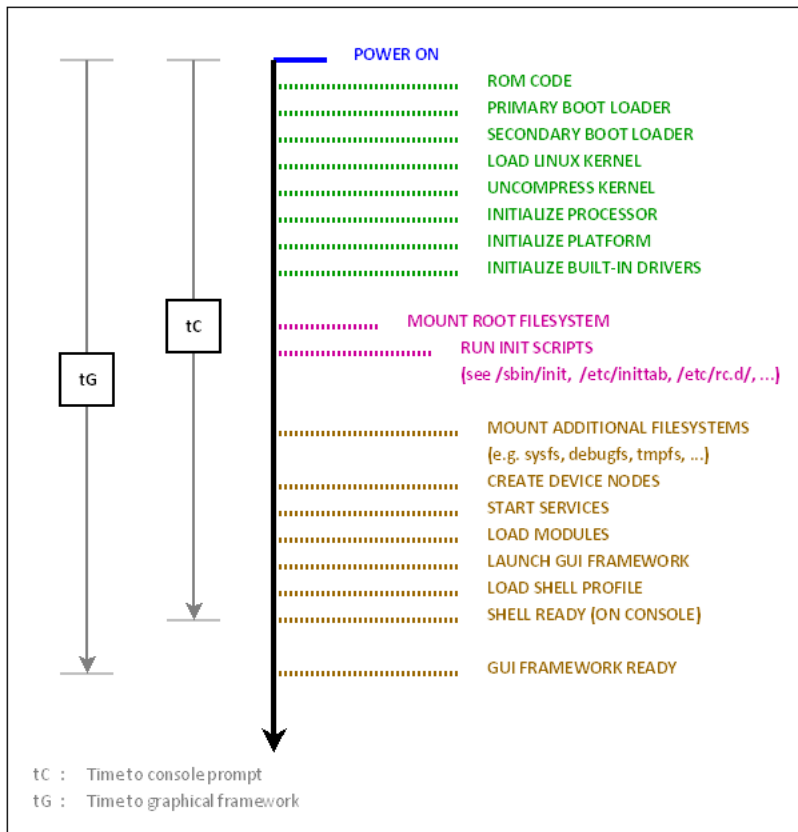
- Boot from SDHC card
- Mount the filesystem from SDHC card itself
- Shell on serial console.
- A graphical application on the LCD

Use-case 2: Boot from NAND

- Boot from NAND
- Mount the filesystem from NAND partition.
- Shell on serial console.
- A graphical application on the LCD

Understanding the boot sequence

A typical (and simplified) boot sequence in Linux based systems is illustrated below:



NOTE

* Actual boot sequence depends upon the device used viz. NAND flash, MMC/SD Card, NOR flash etc.

- Two stage bootloading is required when the bootloader cannot fit into the internal memory of the processor.
- There are multiple inter-dependencies between loading modules and starting services. This picture is presenting a simplified view for illustration purpose.

Measuring boot time

Optimization begins with knowing current boot-time, setting the target and defining the boundary conditions. The boundary conditions are derived from the characteristics of the end-product.

Optimization is an iterative process and we need a reliable mechanism for measure the time. Developers have a variety of options ranging from C programs that can be compiled to scripts that time stamp each line appearing on the serial port. (More details [here \(http://elinux.org/Boot_Time#Measuring_Boot-up_Time\)](http://elinux.org/Boot_Time#Measuring_Boot-up_Time).)

The author used [RealTerm \(http://realterm.sourceforge.net/\)](http://realterm.sourceforge.net/) for the purpose.

- Supports logging in Unix date format - easy for analysis
- Ability to log directly to a file without displaying on the screen - more accurate measurement
- Ability to stop logging after predefined time
- No need to switch terminal for interactive session just for the measurement

Identify markers

As discussed above, the overall boot process involves boot-loader(s), Linux kernel and the filesystem. 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

Developers with intimate knowledge of the source code can point that some of the strings are printed much after the start (e.g. U-boot banner) or much before (e.g. filesystem marker) the actual event. While this is definitely true, these markers serve the purpose well.

Profiling the bootloaders

No built-in infrastructure exists to profile the bootloaders at the moment. Much of profiling is based upon various strings that are printed on the serial console. And it is quite effective!

Looking from the end-product perspective - when all development is complete - the bootloaders are required to perform only these 2 tasks:

1. Perform basic initialization
2. Load binary for the next stage (x-loader loads u-boot and u-boot loads the kernel)

Profiling should be restricted to these actions only.

Profiling the Linux Kernel

Enable timestamps on kernel prints

A timestamp can be added to each string printed by the kernel. Auditing the timestamps can help in calculating actual time spent in processing. To enable timestamp, select the option "Show timing information on prints" when configuring the kernel.

This translates to following in the .config:

```
CONFIG_PRINTK_TIME=y
```

Instrument kernel initialization

When the Linux kernel boot up, all statically linked drivers and subsystems are initialized. This initialization happens via series of **initcalls**. These "initcalls" can be instrumented by adding bootarg **initcall_debug** to the kernel command-line.

Raw data can be extracted by following command at the shell prompt:

```
$ dmesg | grep initcall
```

However, sorting the drivers and subsystems by the time spent in the respective "initcall" is more useful.

- If CONFIG_PRINTK_TIME is enabled:

```
$ dmesg | grep initcall | sort -k8 -n
```

- If CONFIG_PRINTK_TIME is disabled:

```
$ dmesg | grep initcall | sort -k6 -n
```

Same information can be viewed graphically via **bootgraph** script included in the kernel sources. This requires CONFIG_PRINTK_TIME to be enabled.

```
$ dmesg | perl scripts/bootgraph.pl > boot.svg
```

Areas of optimization

Any optimization would fall into either of these categories:

- Size
 - Reduce the size of binaries for each successive component loaded.
 - Remove features that are not required
- 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

Optimization - in both these areas - must be viewed in full context of the target platform and its use cases.

IMPORTANT

Optimization reduces general applicability due to loss of features. This is true for all components - bootloaders, kernel and filesystem.

Reducing the size does help in increasing overall speed - mainly due to gains in time to load the components. But, beyond a certain point, efficiency of implementation gains prominence.

NOTE

This article doesn't attempt to discuss possibility of optimizing the individual kernel subsystems and modules. However, this may be required in the context of the end-product. For example, code meant for handling out-of-protocol quirks, non-conforming devices etc. can be excluded if devices are known to follow defined specs.

Techniques for optimization

Shedding the bulk

The boot-loaders and the Linux kernel contain a *default configuration* that decides which components are included in the build. These default configurations become starting point for creating a product specific configuration. These configurations must be tailored to ensure that unnecessary components aren't being included in the final build.

Some of the features may be required only by developers and don't have much use in the end-product. These features can also be removed.

Here are few examples of simple decisions that can result in a **tailored configuration** for a product.

x-loader

- If NAND support is not required, remove it from the configuration

```
#undef CFG_NAND
```

- If OneNAND support is not required, remove it from the configuration

```
#undef CFG_ONENAND
```

- If MMC/SD support is not required, remove these from the configuration

```
#undef CONFIG_MMC
#undef CFG_CMD_MMC
#undef CFG_CMD_FAT
#undef CONFIG_DOS_PARTITION
```

u-boot

- Avoid long help text for the u-boot commands.

```
#undef CONFIG_SYS_LONGHELP
```

- Use simple parser - instead of hush

```
#undef CONFIG_SYS_HUSH_PARSER
```

- If USB support is not required, remove it from the configuration:

```
#undef CONFIG_USB_OMAP3
#undef CONFIG_MUSB_HCD
#undef CONFIG_MUSB_UDC
```

- If ETHERNET support is not required, remove it from the configuration:

```
#undef CONFIG_CMD_NET
```

Linux kernel

The drivers and subsystems that are not required in the product can simply be left out of the kernel configuration. Ones that are not "essential" at the system boot-up can be built as insertable modules. These modules can be *inserted* into the kernel when they are really needed.

- If the product doesn't support any of MTD devices, the corresponding drivers and support for JFFS2 filesystem can be removed from the kernel configuration.
- Initialization of the capture driver can be delayed until it is really required.

Decision on "What is essential?" and "What is extra?" depends upon the actual product definition.

NOTE

`#undef` is used here to illustrate the configuration options that can be removed in x-loader and u-boot. To exclude them from product specific configuration, not defining them is sufficient.

Deferred initialization

As discussed above, initcalls for some modules may take quite long delaying the boot process. A short patch had been created (against kernel version 2.6.27) to **defer** selected initcalls until after the system is booted - without having to explicitly define them as modules.

- Identify the modules that are not essential for the system at boot-up
- Locate the initcall definition of these modules in the source code.
- Change the declaration of the initcalls from `module_init()` to `deferred_module_init()`
- Once the system has booted, deferred initcalls can be executed by:

```
$ echo 1 >/proc/deferred_initcalls
```

Refer to this (http://elinux.org/Deferred_Initcalls) page for details and link to the original patch.

Build for product

The default configuration of u-boot and the Linux kernel is friendly for development systems - not the end products. This includes, building with debug information, additional code for traceability, etc.

Once the development is complete and the individual components have been well tested, much of the debug infrastructure can be removed.

Here are some suggestions:

x-loader

- Remove option `-g` from the compiler.

u-boot

- Remove option `-g` from the compiler.
- u-boot is build for `armv5` for compatibility reasons. Update compiler flags to build it for `armv7-a`.

Linux kernel

- Remove option `-g` from the compiler.
- Disable `Kernel debugging`
- Disable `Debug Filesystem`
- Disable `Tracers`

When choosing the compiler options, a common dilemma is to optimize for *speed* or *size*. *Here are few data points that should help make the decision faster:*

- Unless executing-in-place, the executable images have to be loaded into the memory from the storage media e.g. while booting from MMC/SD. Bigger means longer time to load the image... which impacts the boot time.
- When optimizing for size, the image would be smaller but the execution would be expected to be *little* slower.

It is common to see the balance shift on either side during the optimization cycles.

Filesystem

A *heavy* filesystem can negate all the efforts in optimizing the components. Most of the filesystems are derived from the *desktop based* systems that may not apply to embedded systems.

IMPORTANT

Initialization scripts need to be reviewed to ensure that only necessary initializations are done.

It is quite common for the filesystems to be built for generic architectures. This may not provide best performance. For example, ARMv7 based system using filesystem built for ARMv5.

Be quiet

In the default configuration, boot-loaders and the Linux kernel print many verbose messages on the serial console. Though quite useful in the development stages, these messages are not required in the end product. Depending upon the number of prints, turning off these messages can save 1 - 2 seconds on the overall boot process.

x-loader

- Add the following line to board specific configuration:

```
#undef CFG_PRINTF
```

u-boot

Turning the prints **off** is a 2 step process:

- Add the following line to board specific configuration:

```
#define CONFIG_SILENT_CONSOLE 1
```

- Set the environment variable `silent` to 1.

```
OMAP3_EVM # set silent 1
OMAP3_EVM # saveenv
```

Linux kernel

The prints in the Linux kernel can be turned off by adding bootarg `quiet` to the kernel command-line.

IMPORTANT

As we have seen above, most of the data used for optimizations is available via these prints. This should, therefore, be delayed till the last iteration. If the desired boot time isn't achieved, enable the prints again and look for more opportunities.

Avoid duplication

Boot-loader(s) and the Linux kernel are independent systems. As generic products, they cannot make assumptions about the state of the devices before using them. This could lead to same device getting initialized - to same settings - across both. Product development happens in a controlled environment, where the responsibilities of each layer can be fixed upfront. This would help avoid unnecessary duplication of initialization code.

As an example, IVA was being initialized in the x-loader, u-boot and later in the Linux kernel for DM37x devices.

- IVA is not being used in either of the boot-loaders - x-loader and u-boot.
- IVA needs to be initialized only for the SoC variants containing IVA e.g. OMAP3530, DM3730.
For devices like OMAP3503 and AM3703, this initialization is just a waste of time!

There could be similar instances for other drivers...

Leverage the SoC speed

Running the SoC - processor and other peripherals - at maximum rated performance is easiest way to reduce the boot time. Since, this is too obvious and doesn't involve any optimization, it is being discussed at the end.

Don't forget to remove instrumentation

In initial sections of the this article, additional instrumentation was added/ enabled to get objective measure to time spent in each software layer. One we are satisfied with the progress made, this instrumentation can be turned off. After a certain stage, amount of time spent in instrumentation becomes proportionally sizable.

Doing it yourself

Starting with the stock PSP release, a series of patches are available for each of x-loader, u-boot and Linux kernel - that indicate the changes done to optimize the boot time.

To illustrate the optimization steps, new configurations - specific to the use cases - were created. Key considerations behind this decision were:

- The default configuration remains untouched for developers to try both configurations without changing branch/ reverting patches.
- This mimics the development flow customers would be taking to create product specific configuration - starting with a known base.

Each patch contains detailed description of the change being done.

Getting sources

x-loader

- Branch `qb_v1.51_OMAPPSP_04.02.00.07` (http://arago-project.org/git/projects/?p=x-load-omap3.git;a=shortlog;h=refs/heads/qb_v1.51_OMAPPSP_04.02.00.07)

U-Boot

- Branch `qb_v2010.06_OMAPPSP_04.02.00.07` (http://arago-project.org/git/projects/?p=u-boot-omap3.git;a=shortlog;h=refs/heads/qb_v2010.06_OMAPPSP_04.02.00.07)

Linux Kernel

- Branch `qb_v2.6.37_OMAPPSP_04.02.00.07` (http://arago-project.org/git/projects/?p=linux-omap3.git;a=shortlog;h=refs/heads/qb_v2.6.37_OMAPPSP_04.02.00.07)

NOTE

* Current set of patches are applicable for the OMAP3EVM.

- In the process of optimizations few generic issues (not related to optimization) were found and fixed on the same branch. Since, these changes apply over a *code freeze* sources, they will be rolled into base PSP package in next cycle.
- Patches for use-case 2 shall be available soon

Building sources

Build x-loader

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- omap3_evm_mmc_config
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

Build u-boot

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- omap3_evm_quick_mmc_config
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

Build Linux kernel

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- omap3_evm_quick_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- uImage
```

Build Linux kernel modules

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- modules
```

Install Linux kernel modules

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- INSTALL_MOD_PATH=<fs-path> modules_install
```

- If same host machine is being used for building the Linux kernel and the filesystem, then **<fs-path>** should point to base of the target filesystem directory.
- If using a pre-built filesystem:
 - Set **<fs-path>** to a local path e.g. **./_install**
 - Create a *tarball* of the **./_install** directory

```
$ cd _install
$ tar cvfz ../modules.tgz .
```
 - Copy this *tarball* to root of the installed filesystem and extract the contents.

```
$ tar xvfz modules.tgz
```
 - Now modules can be installed individually / via scripts e.g.

```
$ modprobe snd
```

NOTE

Steps to prepare the MMC card and NAND flash are not covered in this article. These steps are described in the User Guide corresponding to each release.

Pre-built binaries

TODO : Links to download pre-built binaries to be added

What is current status?

As described in initial sections, AM37x EVM was chosen as the first target for the optimization.

Comparing size of binaries

This table compares the size of binaries generated with *optimized* configurations against *default* configurations.

Component/ File	Default Config (in bytes)	Opt for MMC (in bytes)	Opt for NAND (in bytes)
x-loader.bin	18,932	12,460	7,712
u-boot.bin	219,280	64,808	88,908

NOTE

Depending upon the compiler version used, there may be minor differences in the size of binaries generated. These numbers correspond to Codesourcery Lite v2009q1-203.

Comparing boot times

Usage scenario	Before (in secs)	Opt for MMC (in secs)	Opt for NAND (in secs)
Time to console prompt	TBD	TBD	TBD
Time to graphical app	TBD	TBD	TBD

TODO : Fill the table with actual values

IMPORTANT

Being a wiki these tables are open for editing. Request users to change add notes/comments and additional links bottom of the page. All values here will be backed up by actual patches. Please do not link individual website from the core article.

Before and After Demo

TODO : Will be added soon...

What next?

As discussed earlier in the article, optimizations must be viewed in the context of the end use of product. In the continuum, it would be unfair to suggest/ believe that no further optimizations are possible.

This section describes additional steps that are planned in near future. It also lists alternatives that can be tried out.

Short term

- Specific patches shall be added soon to:
 - Implement the use-case 2 described above
 - Boost the MPU frequency early in the boot process

Long term

- Support for other platforms

More things to try

- Developers would also notice that U-Boot sources in this PSP release are based-off 2010.06 release. Cache support was added to ARM architectures in 2010.09. Migration to this U-Boot version can speed up the U-Boot execution and kernel decompression.
- Barebox** (<http://barebox.org/index.html>) is another bootloader that can be used instead of U-Boot.
- UBIFS** (<http://www.linux-mtd.infradead.org/doc/ubifs.html>) is a new filesystem that may be considered instead of JFFS2. Since it doesn't need to scan the media while mounting, filesystem can be mounted in few milliseconds.
- Use LZO compression instead of default GZIP for the Linux kernel. It is expected to speed-up kernel decompression.
 - If NOR flash is used then kernel compression isn't required.

IMPORTANT

Developers should identify the right set of patches to (rework and) apply in their products. Then, use the learnings from the discussion above to further optimize the boot time.

Frequently Asked Questions

Q: What does the prefix **qb_** in the branch names indicate?

A: This prefix is derived from **quick boot**. Since **Fastboot** (<http://en.wikipedia.org/wiki/Fastboot>) is already being used in entirely different context, this was next best choice.

Q: Where are the patches corresponding to use case 2 defined above?

A: They are work-in-progress. Will soon be posted to these same branches...

Q: Some of the patches are quite generic. Shouldn't they be up-streamed into respective projects?


A: Very true. Some of these patches were already posted upstream. They will be posted soon.

- Q:** Why don't we see compiler option `-O3` being turned on?
- A:** Overall time for execution includes two components:
 - Time to load binary from the storage medium
 - Actual execution timeAdding option `-O3` causes significant increase in the size of generated binaries (as against `-Os`). Additional time taken to load the binary offsets any gains achieved through `-O3`. Based on current experiments, `-O3` would make more sense for platforms that use NOR flash as boot medium.
- Q:** Can it get me to 1 second boot?
- A:** Depends on the target system. If the product functionality can be met by a very small kernel and filesystem, then same techniques can help in achieving a 1-sec boot as well. May be less... because we may be able to eliminate a bootloader stage altogether.
- Q:** When should we add a splash?
- Q:** Splash is added to indicate that system is *getting ready* - quite useful if system takes long to boot. Earliest opportunity (in current implementation) to show a splash screen is u-boot. Size of image being used for splash will impact the boot time.
This can be a useful mechanism for *branding* as well. To use / avoid a splash screen is a compromise between these factors.

References

- http://elinux.org/Boot_Time
- http://elinux.org/Deferred_Initcalls
- <http://www.linux-mtd.infradead.org/doc/ubifs.html>
- <http://free-electrons.com/blog/lzo-kernel-compression/>

Keystone=		C2000=For technical support on the C2000 technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments about the article Optimize Linux Boot Time here.		MSP430=For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the article Optimize Linux Boot Time here.		OMAP35x=For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article Optimize Linux Boot Time here.		OMAPL1=For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article Optimize Linux Boot Time here.		MAVRK=For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum. Please post only comments about the article Optimize Linux Boot Time here.	
1. switchcategory:MultiCore=		■ For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum		■ For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum							
Please post only comments related to the article Optimize Linux Boot Time here.		Please post only comments related to the article Optimize Linux Boot Time here.									

Links			
	Amplifiers & Linear Audio	DLP & MEMS High-Reliability Interface	Processors
	Broadband RF/IF & Digital Radio	Logic	<ul style="list-style-type: none">ARM ProcessorsDigital Signal Processors (DSP)Microcontrollers (MCU)OMAP Applications Processors
	Clocks & Timers	Power Management	Switches & Multiplexers
	Data Converters		Temperature Sensors & Control ICs
			Wireless Connectivity

Retrieved from "https://processors.wiki.ti.com/index.php?title=Optimize_Linux_Boot_Time&oldid=114351"

This page was last edited on 24 July 2012, at 20:17.

Content is available under [Creative Commons Attribution-ShareAlike](#) unless otherwise noted.