

Time and Space: Optimizing Boot Up and Footprint in Linux-Based CE Devices

William Weinberg, Open Source Development Labs

Abstract: *Linux is making great strides as an OS platform for advanced CE devices. This paper describes how developers can optimize the Linux kernel, drivers, and applications to offer "instant-on" performance and to meet the BOM budgets for this next generation of smart gadgets.*

Introduction

Advances in embedded processor performance and falling prices for RAM and Flash memory over time have boosted the adoption of enterprise-class OS platforms, like Linux, in CE devices. Advancing h/w capabilities have not, however, overcome the need for "instant-on" and nor limited BOM budgets in CE devices.

This paper describes the challenges faced in deploying Linux in CE applications, specifically those related to accelerating boot time and reducing memory footprint. It enumerates the components of each: the steps in the boot sequence and the storage and run-time needs of the Linux kernel and applications. Moreover, it provides paths to achieve rapid device boot-up and to fit advanced CE device applications into available Flash and RAM memory. In particular, the paper details the initialization phases of the Linux kernel, memory management, driver, file system and applications, highlights the size impact of Linux file systems, and examines trade offs in execution time and storage space.

How Fast is Fast Boot?

Developers of intelligent consumer devices face tough demands for "instant on". Rapid power-up expectations arise from experience with appliances like televisions, radios, VCRs, and basic cell phones.

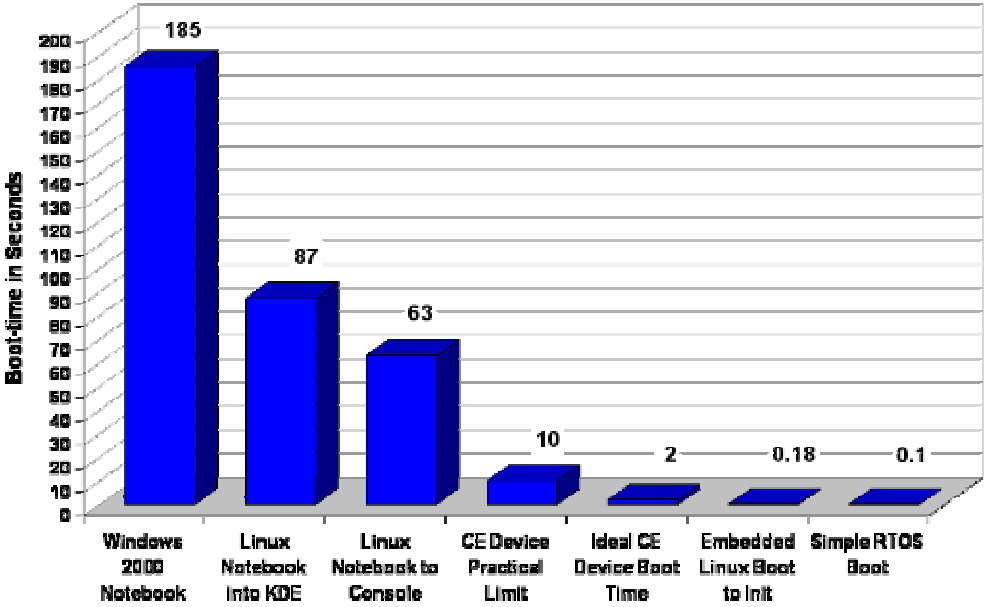


Figure 1 – Comparing Boot Timings of Enterprise Systems and CE Devices¹

¹ Notebook times measured on 800 MHz PIII Sony Vaio 505 systems; CE times from customer interactions; Embedded Linux times from tests at MontaVista Software on 206 MHz StrongARM 1110 h/w.

Boot Sequence - What Happens at Linux Boot Time?

It is instructive to compare the boot-up times for the familiar notebook computers running Windows and Linux with the more aggressive times required in embedded devices. Figure 1 illustrates the large gulf in startup times from the desktop to embedded – a gap of two orders of magnitude!

In workstation applications, the majority of the boot sequence is spent either in the firmware (BIOS) or in reading services and daemons (User Space Initialization)² [Figure 2].

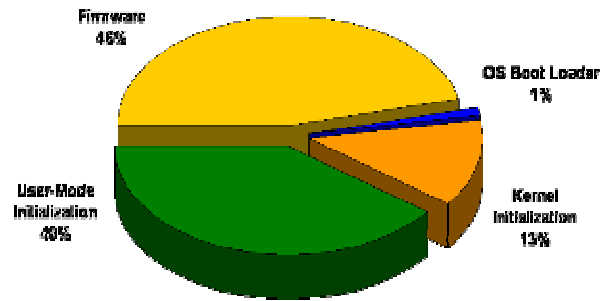


Figure 2 – Distribution of Boot Sequence Among Firmware and Software

This sequence can be further analyzed as five sub-sequences:

	Power-On	BIOS/ Monitor	Kernel Load	Kernel Init	User-space Init
Duration	< 1 second	0-15 seconds	0-5 seconds	0-15 seconds	2-60 seconds
Tasks	<ul style="list-style-type: none"> • Debounce • Hold-off 	<ul style="list-style-type: none"> • POST: • Memory Test • H/W Init • Bus Discovery 	<ul style="list-style-type: none"> • Decompress • Copy to RAM • RAMdisk init 	<ul style="list-style-type: none"> • Timebase • Page Tables • Bus Discovery • Dev Drivers • File Systems 	<ul style="list-style-type: none"> • Daemons • Logging • Console • Java • Graphics, etc.
Trade-offs	<ul style="list-style-type: none"> • Power Source 	<ul style="list-style-type: none"> • BIOS yes/no • BIOS Options • Memory size • Bus No./Depth 	<ul style="list-style-type: none"> • RAM vs. XIP • Memory Size • Kernel Size • FileSys Size 	<ul style="list-style-type: none"> • Hard coded vs. discovered • No. Pages • No. Buses • Drivers in Kernel • FileSys Init 	<ul style="list-style-type: none"> • Sequenced or Parallel exec. • Appl. In RAM vs. XIP • No. execs • Daemon init times

Figure 3 – Key Phases in the Boot Sequence

Power-On

For most systems, turning on the power supply is effectively instantaneous, but a small finite time must pass from “cold” power on until available voltage stabilizes, usually a matter of a few tens of milliseconds.

BIOS/Monitor

In a PC, a BIOS supports h/w initialization, memory test, bus discovery, and other cold-start tasks. In custom designs, a boot monitor performs a subset of these chores, and defers others to kernel initialization or user-space processing. In both cases, duration of this phase can vary greatly, from under 1 second to over 15 seconds. Factors include:

- Physical memory size and scope/complexity of memory testing
- Number and type of devices to initialize
- Number and depth of buses (e.g., PCI) to discover, traverse and map
- Other timeouts built into the firmware by default (e.g., console checking)

² From PFIFFER [2003].

A BIOS usually supports disabling or scaling back memory tests and POST functions; custom firmware, offers the luxury of whether to perform the checks at all (embedded applications may need MORE not LESS power-up testing than a PC).

Loading the Kernel

In a Linux desktop or server, a boot loader (e.g., GRUB or LILO) locates a compressed kernel image (vmlinuz) in a file system and copies it into RAM, decompressing during copy. Once the image is in RAM, the execution transfers (jumps) to a preset start address to begin kernel initialization. In many embedded systems (and some desktops/servers), the loader must also decompress and copy a RAMdisk-based root file system to RAM.

The time needed to copy kernel and file system images to RAM increases more or less linearly with the size of the file. For small kernel and file system images, time spent is trivial; larger images may impact instant-on requirements. For example, if a system requires 0.5 microseconds to read, decompress, and write a byte of kernel code to RAM, then a 2 MB image will take 1,048,576 microseconds, or just over one second to load the kernel or a like-sized file system image; a 10 MB image will take five times as long, contributing five seconds to the total boot sequence.

For larger images, using Execute In Place (XIP) from flash instead of copying eliminates this entire phase – running directly from flash/ROM removes the need to decompress and copy. Depending of flash memory access speed, XIP may also slow down kernel initializing and subsequent execution.

Kernel Initialization

Understanding all components of Linux kernel initialization entails comprehending large portions of the kernel itself, which is beyond the scope of this document. Nonetheless, key kernel initializations deserve examination:

- **Driver Initialization** – For fixed-function devices, developers often link all drivers to the kernel at build time. However, statically bound drivers must also be initialized at boot time. Since this process is serialized and since some devices present long start-up latencies, they can add significant delays to kernel initialization. Consider migrating drivers for slow-start devices into loadable modules.
- **File Systems** – Many file system types preferred in consumer electronics applications are kernel-based (e.g., CramFS, JFFS2), removing choice about when to initialize. Initialization of drivers for flash is fairly trivial – more CPU-intensive activities occur at mount time. Avoid mounting “slow” file systems during kernel init. Choose root file system type wisely (RAMfs is a good choice). Defer mounting others until user space init or later.
- **Interconnects and Buses** – Most CE devices have fixed on-chip interconnects and few off-chip buses, but can have peripherals, like USB or FireWire, that demand discovery. With a large peripheral population, consider deferring discovery until after user space init.
- **MMU Page Tables** – with large amounts of system memory, initializing management structures mapped RAM can be time-consuming. A system with 128 MB of DRAM and a 4 KB page-size must map 32,768 pages and place them into a free page list, and then allocate pages for use by the kernel, drivers, and for storing page tables themselves. On systems with fixed memory size, page tables could be pre-initialized (requires kernel hacking).
- **Timing Loops / Delays** – the kernel needs to establish time bases and calibrated operation to match discovered hardware. Time spent in calibration slows the initialization process. In many cases, empirically-derived calibration values can be used instead of run-time calibration.

User Space Initialization – Init

Once the system has powered up, the kernel gets copied to RAM and initialized, then control is passed to the “init” process, with the following steps:

- The kernel looks in `/sbin` for `init`

- `init` runs the `/etc/rc.d/rc.sysinit`
- `init` runs all the scripts for the default run-level (as specified in `/etc/inittab` and by the contents of run-level directories `rc0.d...rc6.d` in directory `/etc/rc.d/`)
- `init` runs `/etc/rc.d/rc.local` (and your application start-up)

Linux features the notion of “run-levels” that determine the type of operation and imply the types of services available to users and applications.

Workstations typically boot up into run level 5 with full graphics. A simple embedded application would likely execute in Run Level 2.

Level	Dir	Description
1	rc1.d	Single User Mode
2	rc2.d	Single User Mode with Networking
3	rc3.d	Multi-User Mode - boot up in text mode
4	rc4.d	Not yet Defined
5	rc5.d	Multi-User Mode - boot up in X Windows
6	rc6.d	Shutdown

Figure 4 – Summary of Linux Run Levels and Directories

Each run level has its own directory of start-up activities, which you can customize.

```

K06splash_late K10fbset      K21hotplug   S03pcmcia    S13cups
K07cron        K14resmgr    K21lisdn    S05network   S13kbd
K07hwscan     K14splash_early K21random  S06syslog    S13powersaved
K09cups       K16syslog    S01hotplug  S08resmgr    S13splash
K09powersaved K17network   S01lisdn    S08splash_early S15cron
K09splash     K19pcmcia    S01random   S12alsasound S15hwscan
K10alsasound  K20coldplug  S02coldplug S12fbset     S16splash_late

```

Figure 5. –Default Run Level 2 initialization in SuSE 9.2 (contents of `/etc/init.d/rc2.d`)

Note that items prefixed with “K” are for termination of services from other levels, while those with “S” start new services. Execution order is set based on the number that follows. You can learn more about each `init` entry by reading the source code (most are shell scripts). You can also optimize this sequence by removing or deferring items from `inittab`, from the run-level directory.

Achieving Instant On

There are two approaches to enhancing “instant on” from the user perspective – (1) tuning and reconfiguring the Linux kernel, drivers, and user-space code to optimize the boot sequence, and (2) taking steps to improve the end user’s perception of how long that sequence takes.

Tuning the Start-Up Sequence

Improving actual start-up time starts with analyzing the components of the boot sequence and measuring the duration of each. You can do this yourself by adding timestamps to key start-up scripts or by increasing the amount of system logging performed (which itself slows down booting) or by using tools from commercial vendors and the open source community. On the commercial side:

- MontaVista Software supplies the System Timing Tool as part of MontaVista DevRocket for MontaVista Linux Consumer Electronics Edition
- TimeSys Corporation offers some level of timing analysis in their TimeStorm product
- MetroWerks CodeWarrior Linux Platform Edition includes a family of bring-up tools
- Hardware Assisted Debug from suppliers like American Arium, EST, Lauterbach and Wind River Systems offer execution tracing, from which timing information can be extracted
- Real-Time Innovations (RTI) and Viosoft offer performance analysis tools for embedded Linux

With the metrics in hand, you can make informed decisions about kernel configuration, sequencing, pre-initializing and migrating components and functions in or out of the kernel and into user space.

The “User Experience”

The need for rapid boot can exist in opposition to the many tasks an intelligent device must perform during start-up. If the actual total time cannot be reduced, then a variety of techniques can be used to improve the end-user’s perception of start-up time:

- Put up a splash screen early in the boot sequence, (like the penguin image in server and desktop Linux consoles)
- Design an early splash screen that mimics real functional screens – e.g., draw non-functioning buttons and controls that are drawn over later with active versions
- Sequence critical services (like emergency dialing on a handset) to start early; defer others until boot and user space init have completed
- Defer time-consuming operations (like mounting journaling file systems) until after init

Scaling Embedded Linux Memory Footprint

Before embarking upon a discussion of Linux footprint scaling, remember that Linux is not a tiny RTOS and was never intended to become one. Linux is a very efficient enterprise operating system whose application space now includes a large swath of intelligent, connected devices. For its part, the Linux kernel does not today and will likely never compete on size with 50-100K byte embedded executives – but that doesn’t mean that Linux is “fatware”

While the minimum footprint for a Linux system will dwarf the 50-100K byte profiles of “classic” RTOSes and executives. The reasons for this size differential are several:

- Size is as Size does –small memory footprints in RTOSes represent much less functionality than the minimum found in Linux. With TCP/IP, file systems, security, memory protection and a standards-based APIs, a once-tiny RTOS will consume as much if not more memory than embedded Linux.
- Linux is a Platform, Not a Library – RTOSes are built as run-time libraries. RTOS applications link in just the services they really use. If you don’t use semaphores, then semaphore code is not linked in. By contrast, Linux provides a standard services repertoire to applications as deployed today, updated tomorrow, and to new applications beyond. Scaling the Linux kernel configures in or out SMP support, kernel-based file systems, etc. but not core APIs/PCs. *Ad hoc* code removal limits the ability to run COTS s/w and future versions of application code.
- Size is Relative – some legacy RTOSes do fit into under 100K, and embedded Linux into under 500K. Realize, however, that the minimum footprint of WindowsCE.net variants like PocketPC is 24-27 megabytes, larger again by almost fifty times!

Application Profiles

To give developers a sense of resource usage, let’s examine a series of application profiles³ that correspond to basic embedded applications:

Minimum	Console application -- boots and reports memory usage. No networking or login
Basic	TCP/IP-enabled system with telnet daemon and login shell
Router	Basic Profile plus Zebra routing package.

Memory use divides into two components, kernel and file system, and presents one value at boot-time (compressed) and another at run-time (expanded into RAM). The kernel image includes only code from

³ WEINBERG/MontaVista [2004]

the kernel itself and device drivers, RAMdisk file system code (RAMfs), and TCP/IP (where applicable). File system images include a root file system and the daemons and programs that run in the profile. Boot-time images are compressed and are expanded into RAM for execution and access.

RAMdisks are used in these profiles for simplicity, but consume 3x their content size: once in compressed form, once in expanded form, and up to a third time as programs are loaded to run. With flash file systems, no decompression to RAM need take place at boot-time; an XIP-capable flash file system reduces program memory use to a single (uncompressed) image.

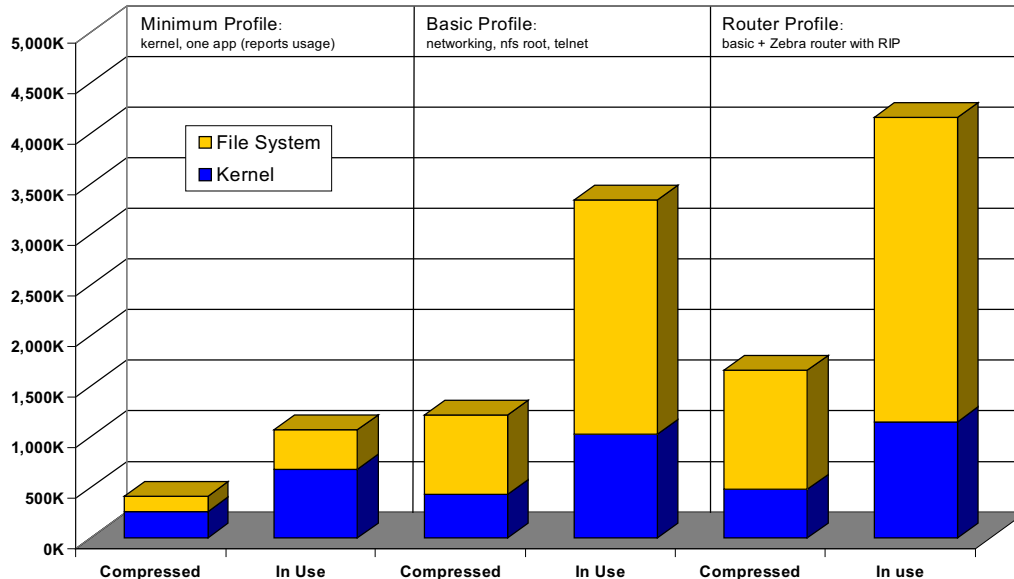


Figure 6 – Compressed and In-use Memory Footprints for Three Application Profiles (ARM Arch.)⁴

Analysis

In the Minimum profile, the ARM kernel takes up less than 250K bytes of boot storage (and grows to about 500K at run-time). The richer Basic and Router profiles trend to just under 400K at boot and run in less than 1 MB. RAMdisk image size varies with program and data content. Larger profiles contain more code because they do more. Nonetheless, a robust routing application can be implemented in around 1.5 MBs flash and will run in approximately 4 MBs of system memory.

Further Scaling Opportunities in Time and Space

Linux offers other opportunities to reduce boot-time and run-time memory usage:

- Use compressing flash file systems (JFFS2 and CramFS) to save space
- Reduce run-time RAM by experimenting with kernel command-line switches. To get the smallest possible RAM used, step available memory down in 16K increments at boot-time (in LILO, GRUB, etc.) and try to boot the system until you reach a value that won't boot. Use a value larger than this to suit your needs, but smaller than the default.
- Pull device drivers from the kernel boot image and load them later as modules – you will save on kernel image size at the expense of (compressed) file system space.
- Strip symbolic information from executables, after you have thoroughly debugged your application

⁴ Ibid

- Carefully audit the contents of `/bin`, `/sbin`, `/usr/bin`, and `/etc` and remove unneeded code and configuration files
- Build your application and selectively rebuild the kernel and libraries with `gcc` optimizations biased towards size over execution speed

Conclusion

TANSTAAFL – there ain't no such thing as a free lunch. – Robert Heinlein.

Embedding Linux brings huge advantages to embedded systems developers in terms of development options, available software components for re-use, robustness, networking, security, and acquisition cost reduction. At the same time it carries with it the burden of coming from enterprise computing roots, where functionality, robustness and throughput trump resource footprint, boot time, and the embedded tradition of painstaking byte-for-byte control of application size and speed.

Thinking Forward

When you consider designing an application to use Linux, you will have set aside many of the hard-won sensibilities (or prejudices) that you have about embedded software. Linux is not an RTOS. Linux will not fit into on-chip RAM on most SoC hardware. You will likely have to budget more RAM and flash into your product BOM than with your last design's kernel. You will benefit from using a 32-bit MMU-enabled processor or microcontroller core. But these are investments you are likely to make anyway when designing long-lived product platforms, not just "one-off" application designs.

Linux will, however, give your application "peer status" with other modern computing devices (even your workstation). It will give you equal or greater choice in choosing your tools and processor family than any proprietary RTOS for CE devices. It will let you leverage vast amounts of available application code and middleware from an ecosystem that dwarfs the embedded marketplace by an order of magnitude or more. It will at last let your company's management build a top-to-bottom IT strategy without being beholden to a single proprietary technology supplier.

Embedding Linux is not a fad. Just ask D-Link, Motorola, NEC, PalmOne, Panasonic, Philips, Sony, Samsung, Tivo, Yamaha, and dozens of other name-brand CE device OEMs around the world.

References

- HEINLEIN, Robert [1966]. *The Moon is a Harsh Mistress*. Tor Books.
- HUNT, James [2003]. "Boot Linux Faster - Parallelize Linux system services to improve boot speed". IBM DeveloperWorks
- LEHRBAUM, Rick [2000]. "My Linux is smaller than your Linux". LinuxDevices.com.
- PIFFER, Andrew [2003]. "Reducing System Reboot Time With kexec". Open Source Development Labs.
- WEINBERG, William [2004]. *Building Intelligent Devices with MontaVista Linux Consumer Electronics Edition*. White Paper Version 2.1. MontaVista Software, Inc.
- WEINBERG, William [2003]. "Optimizing Linux for Handheld Devices with Execute in Place". *Intel Wireless Solutions Magazine*, April, 2003 edition.