Achieving Faster Boot Time With Linux

Christopher Hallinan
Technical Marketing Engineer

mentor.com/embedded
Agenda

• System Definitions
• Hardware Considerations
• Typical Embedded Linux System
  – Boot Sequence
• Measurement Methods
• Optimization Techniques
• Summary/Q&A
Define what "Fast Boot" Means

- Fast Boot is not a single technology or "product"
- Many techniques are architecture or platform dependent
- Product/application defines "fast boot"
- Your system requirements determine limits
  - CAN Bus in 50 ms
  - Rear view camera video in 2 seconds?
  - Partial HMI in 3 seconds?
  - Full multimedia plus networking in 4 seconds?
  - Do you need Secure Boot?
What takes so much time?

- Power/Clock Stabilization
  - usually negligible but should be considered
- Low Level CPU Initialization - \(~ 100 \text{ ms}\)
  - Bootloader (often multi-stage, ie secure boot)
- Loading images (kernel, u-boot, rootfs, dtb)
  - Usually these images live on NOR or NAND Flash
  - Even a small reduced kernel can be 1-2 MB (compressed)
  - Often stored compressed (kernel)
- Subsystem (Driver) initialization
- Mounting a root file system
- Userland – System Utilities and Applications
Typical Embedded Linux System

• Freescale i.MX6 SabreLite:
  – Quad-Core ARM® Cortex A9 1 GHz
  – Yocto core-image-sato (~90 MB rootfs)
  – Booting from Class 10 micro-SD

• Stopwatch analysis:

<table>
<thead>
<tr>
<th>From Power On to:</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel FB logo</td>
<td>5</td>
</tr>
<tr>
<td>Userland pspash</td>
<td>13</td>
</tr>
<tr>
<td>Full Mobile Desktop</td>
<td>23</td>
</tr>
</tbody>
</table>
Hardware Considerations

- Hardware architecture makes a difference
  - Goes beyond just clock speeds, etc.
- Power and Clock stabilization should be very fast
- Design choices should support fast boot requirements

Examples:
- Loading u-boot and kernel from SPI NOR takes substantially longer than from parallel NOR or NAND
- NAND flash or SD/MMC requires early software overhead but may be faster overall
- Is your bootloader enabling caches early?
Typical Boot Sequence

Hardware Loader

Initial Boot Loader

Boot Loader

Kernel

“User-land”
Typical Linux Boot

Kernel Init
- Hardware init
- Core init
- Loops per jiffy
- Memory init
- Driver init
- Bus probing

Rootfs
- Filesystem init
- UBI scan
- Rootfs mount

Userland Init
- Init (systemd)
- Subsystem init
- Preload

Apps
- Application Startup
- Prelink
- Optimize
The Bootup Phases (Relative Phase Lengths)

Bootloader
- Early h/w init
- Load/start kernel

Kernel
- Core init
- Module init

Kernel Startup

User-Space Initialization
- Init scripts
- System processes
- Applications

Userspace

Time
Splash Screens – Make it look like it’s booted!

• Indicates system is active, but still booting
• A splash screen can take place:
  – In the bootloader
    – Will get splash up sooner, but...
      – Usually require some porting to bootloader
      – May delay early kernel functionality
  – In the kernel
    – After initialization of the framebuffer driver
  – Early user-space init (psplash)
    – Before system apps initialized
Profiling and Measurement Tools
Boot Time Measurement Methods

• Techniques depend on context:
• Bootloader profiling often requires custom tools
• Several tools are helpful for Linux kernel profiling
  – Some are very easy to use (CONFIG_PRINTK_TIME, etc)
• Userland tools
  – Many tools to chose from
  – Some easy to use, some require investment in learning
• Some portions require custom techniques
Profiling U-Boot

• No clever "out-of-the-box" tools
• Whatever you do here will be custom
• First order: something similar to CONFIG_PRINTK_TIME
  – Enable timestamp values on each line of console output
  – May require architecture-specific timer
  – Can also use "grabserial"
• Second order:
  – Custom time checks around suspect areas
  – Hardware tracing on supported processors
Some Popular Measurement Tools

- CONFIG_PRINTK_TIME
- ftrace
- Bootchart (userspace)
- SystemTap
- LTTng

- initcall_debug
- oprofile (both)
- perf
- strace (userspace)
- uptime
- ...

©2013 Mentor Graphics Corp. Company Confidential
Timing Printk

- A simple method to put a timestamp on every printk
- Useful to identify lengthy init operations
- Activation (use one of the following):
  - Compile kernel with: CONFIG_PRINTK_TIMES=y (in Kernel Hacking)
  - Use “printk.time” on kernel command line
    - Both of these methods can be used during kernel boot
  - Or, dynamically in a run-time system (as root):
    - echo "Y" >/sys/module/printk/parameters/time
    - Obviously only useful after kernel is booted
Using ftrace for profiling

- Requires some "investment" (learning curve)
- Analyze and debug latency and performance issues inside the kernel
- Profile execution time of functions, events, more
- A “framework” of several assorted trace and event utilities
  - Lots of filters and options for fine tuning your measurement
  - Over 800 events in 26 categories (syscalls, sched, irq, module, etc)
- Controlled via /sys/kernel/debug/tracing
- Can be enabled on kernel command line to facilitate early boot profiling. Ex:
  - ftrace=function_graph tracing_thresh=5000
- See “…/Documentation/trace/ftrace.txt for more information
Bootchart

- Very useful for correlating CPU utilization with initialization process
- Helps identify opportunities for parallel init
- Limited to userland initialization profiling
- Very easy to use: init=<path-to-systemd-bootchart>
- Also works with systemd
Typical Bootchart Output – i.mx6 startup

I/O Read
I/O Write
CPU Utilization
CPU Wait

Processes
SystemTap

- Powerful instrumentation framework for tracing, profiling and evaluating kernel behavior
- Command line interface and scripting language
- Ideal for complex tasks that require live analysis without having to recompile the kernel
- Primarily designed for users with in-depth kernel knowledge and experience
- Can provide insight that no other tool can
- Significant learning curve
- Plenty of documentation on-line
- May not be available on every platform
LTTng (Linux Trace Toolkit next gen)

- Kernel and userspace tracing
  - Full system profiling
  - Ideal for profiling multi-core systems
    - Are all the cores utilized efficiently?
    - Are there opportunities for further parallelization?
  - View cpu states, scheduling, cpu utilization, IRQ rates, page fault rates, function call flow, fs and network activity, etc.

- For most modern kernels, LTTng is easy to integrate into your kernel
  - As always, YMMV depending on version and architecture
Using initcall_debug

- Driver initialization calls (initcalls) spend considerable time on kernel bootup

- A kernel flag enables initcall information during startup
  - On the command line, add "initcall_debug=1"

- Notes:
  - Increase the printk log buffer size in kernel config:
    - LOG_BUF_SHIFT=18 (256KB)
  - Enable CONFIG_KALLSYMS (to see function names)

- After booting info is found in bootlog (dmesg)

- May identify:
  - Opportunities for parallelism
  - Opportunities for removing functionality
  - Opportunities for deferring init
Using initcall_debug

Driver initialization calls (initcalls) spend considerable time on kernel bootup.

A kernel flag enables initcall information during startup:

On the command line, add "initcall_debug = 1".

Notes:

- Increase the printk log buffer size in kernel config:
  LOG_BUF_SHIFT=18 (256KB)
- Enable CONFIG_KALLSYMS (to see function names)

After booting info is found in bootlog (dmesg).

May identify:

- Opportunities for parallelism
- Opportunities for removing functionality
- Opportunities for deferring init calls
Oprofile

- Oprofile
  - Statistical profiler which records PC of currently executing program when specified events occur
  - Events include: (machine/processor dependent)
    - Timer, cache refills, interrupts while masked, many more
    - Varies by architecture
  - Useful for finding “hotspots” in kernel and application code
    - Not available on every platform
    - Often requires some customization
Perf

• Much more functionality than oprofile
  – Generally lower overhead, no daemon required
• Statistical profiling of entire system
  – Kernel and userspace
• Integrates hardware performance counters, tracepoints and dynamic probes for advanced profiling
• Userspace app which controls operations
  – perf
• Several subcommands: stat, top, record, report, sched…
• Does not work on all platforms
  – May need porting/patching to your particular platform
Using `strace` to profile applications

- Strace can be used to collect timing information for a process
  - `strace -tt 2>/tmp/strace.log` httpd ...
- Determine where time is being spent in application startup
- Can also collect system call counts (`-c`)
- Can see time spent in each system call (`-T`)
- Great for finding extraneous operations (ubiquitous)
  - scanning invalid paths for files (e.g. dynamic libs, fonts, etc),
  - opening a file multiple times, etc.
- Strace can follow children (`-o -ff`)
- Strace adds SIGNIFICANT overhead to the execution of the program
  - Good for relative timings, not absolute
  - May slow execution so much that it “breaks” interaction with other processes
Optimization Techniques
Optimization Techniques

• There are numerous ways to speed up boot time
• Your mileage may vary depending on many factors
• Identify the longest bootup paths and select these for optimization
• Some techniques are obvious, others not so
• Some are aggressive and intrusive
• Many are simply tweaks and easy to apply
  – Turn off unneeded options, etc.
Optimizing U-Boot

- U-Boot must be relocated from Flash into DRAM
  - Reducing the image size reduces relocation time
- Lots of useful development functionality
  - tftp, pci scan, mem utils, disk utils, load*, dhcp, etc
  - In a production system, many of these features are unnecessary
  - Disabling these features can have a significant impact on boot time
- You want the bootloader to do it’s work and get out of the way as fast as possible
- Look at every CONFIG_* option in your board configuration header
  - include/configs/<board_name>.h
Optimizing the Linux Kernel

• Size matters
  – The kernel needs to be loaded from FLASH into RAM
  – Smaller == faster

• Consider using an uncompressed kernel
  – Decompression can take dozens to hundreds of milliseconds

• Configure as many drivers as possible as modules
  – Mostly a "brute force" approach – trial and error, time consuming

• Consider using deferred initcall patch
  – Defer module init until much later in boot cycle
  – Initcalls deferred until triggered in userspace
Optimization Techniques

- XIP (Your mileage may vary)
- Limit console printk()
- Pre-configure or eliminate udev
- Kernel modules/deferred initcalls
- RTC_nosync
- Checkpoint restart
- Use parallelism for multi-core
- Cache systemd config
- Minimize rootfs size
- ...
Userspace – Optimize init

• Use BusyBox – Very popular in embedded systems

• Consider a custom init for very aggressive boot times
  – Can configure init=myinit on kernel command line
  – Allows complete customization of userland initialization
  – It is always faster to run native code than scripts
    – In general, every line of a script causes fork()/exec()
  – Often used in fixed function types of devices

• If you’re using ready-made startup scripts
  – Eliminate unnecessary stuff (set -x)
  – Run multiple scripts in parallel wherever possible
  – May require adding some synchronization between services you start in parallel if there are dependencies.

• Use SystemD instead of SysV init
Using systemd

- Alternative to SysV Init
- Avoids much fork/exec of typical start up scripts
- Compatible with SysV Init scripts, but
  - Translate to systemd config files for best results
- Solves many of the startup dependencies quite nicely
- Parallelizes as much as it can
  - Big win on multicore
- See http://0pointer.de/blog/projects/systemd.html for an interesting introduction by the author
Application Prelink

• A good portion of application initialization time is spent resolving symbols to dynamic libraries
• Using Prelinking you can cut off a significant portion of application startup time if you have a large/complicated userland
• Tries to assign a preferred address space to each library used by an application – ahead of time
• Prelink is essential to rapid application startup
Designing your Applications - Considerations

- Keep it small
- Prelink
- Be careful adding dependencies on new libraries
  - it can snowball
- Keep fork/exec to minimum
- Some multithreading can help esp. with multicore
- Use Analysis tools – profiling, strace, etc.
- Use only Fast-path I/O Peripherals (e.g. – no Wifi)
- Avoid “discovery” code if possible
Achieving Sub-1 Second Boot
Aggressive Boot Time Reduction

• Mentor’s Adaptive Preloading File System*
  – Profiles boot, stores read block list for faster playback

• U-Boot DMA
  – SPL or KL configuration

• Consider Read Only FS for root - YMMV
  – CRAMFS/SquashFS

• UBI Fastmap

• Avoid Initramfs
  – Apps that require shared libs require the entire lib in initramfs
  – Kernel drivers are more optimized than u-boot
  – Use a preloading FS instead

• Tiny kernel -> kexec full Linux kernel

*This is a technology, not a product
Linux Fast Boot – Mentor Projects

• Lesson Learned
  – Boot time is highly dependent on choices made in HW
  – Hardware architecture and system architecture matter
  – Individual requirements vary

• Mentor has significant experience
  – Kernel optimizations for custom boards in automotive space
  – Architecture analysis for system trade-offs in boot time performance

• Sample Benchmarks Mentor Graphics has met in the past
  • 100 msec boot loader to start kernel loading
  • 750 msec for Linux kernel start
    • Rootfs and read/write filesystem mounted
    • First user process running
  • Audio on within 1.25 sec
  • Safety-critical Camera feeds within 2 sec
  • Home screen available within 10 sec
Thank you!
Your Questions are Welcome

mentor.com/embedded