Debugging embedded Linux power management

Kernel developer point of view

ELCE, Barcelona, Nov. 7 2012

Tero Kristo <t-kristo@ti.com>
Contents / Introduction

• Debugging tools / methods for PM
• Kernel power management features
• Typical power management problems / bugs
Common PM architecture
Debugging tools / methods for PM
Disabling kernel features

• Disable CONFIG_PM
  – If bug remains, complain to someone else!

• Disable unnecessary drivers
  – Only enable minimal set like timers, console, I2C, etc., depends on arch which are needed

• Disable PM features
  – Only enable a single PM feature at a time and attempt to reproduce the problem

• Pros:
  – Good for isolating a PM related problem if don't have any kind of clue about what is wrong

• Cons:
  – Rather slow and difficult to use
Stress testing

- Scripts that do some PM related operations in a tight loop
  - Example: suspend loop with a wakeup from suspend every 100ms

- Should be random enough so that the bug producing pattern is executed

- Pros:
  - Can be very useful in reproducing some problems that take typically a long time to occur with normal use

- Cons:
  - Difficult to figure out what operations to actually execute in the stress testing script
Tracing (printk / low level UART)

- Populate enough debug printk’s to the code being examined
- If possible, can also dedicate a custom interface for debug traces
- Pros:
  - Easy / quick to use
- Cons:
  - Typically alters code execution time (especially if using serial port), and may hide the actual problem
  - Not usable from very low level code (printk)
  - Printk:s are cached and may not be printed out before a crash happens (e.g. during suspend)
GPIO / LED trace

• Add control to some GPIO / LED signal from certain points in code
  – Example: LED is turned on when CPU is running, disabled when idle

• If possible to use multiple signals, can provide a binary coded trace value from kernel

• Pros:
  – Single GPIO / LED control typically does not consume so much time as to alter execution times drastically (vs. UART)
  – Can even use multiple devices with LEDS, and if using stress testing script, can immediately see if some devices have crashed or not
  – Useful in case debugging code areas where debugger / printk is not usable

• Cons:
  – Most likely only a few available (what to trace?)
Debugger

• Useful in developing new code, and sometimes can see where kernel has crashed

• With PM code, typically need breakpoints
  – Static / dynamic
  – Fake breakpoint (infinite loop in code, re-program PC after stopped)

• Real hackers don't use/need debugger though

• Pros:
  – Well, debugger is always a debugger

• Cons:
  – Communication with CPU is probably blocked during low power operating modes
Buffered traces / statistics

- Trace information collected from kernel side into a ring buffer
- Dumped out with a console operation through e.g. debugfs
- Typical uses: cpuidle tracing, power state usage statistics
- Kernel tracepoints seem to be a good tool for this, and it is easy to add new tracepoints in case something is missing

- Pros:
  - Minor impact to execution times (no slow HW components accessed)

- Cons:
  - Only useful in debugging misbehavior (crash prevents later dump)
Example: tracepoints with idle 1/2

• Execute following script in target device:

```sh
#!/bin/sh
trace-cmd reset
sleep 10
trace-cmd start -e power
sleep 10
trace-cmd stop
trace-cmd extract
```
Example: tracepoints with idle 2/2

- Copy resulting trace.dat over to host and process it with kernelshark
Exporting debug functionality to userspace

• Provide a testing API to userspace to read / write hardware registers directly (on memory mapped registers can use /dev/mem)

• Enhance existing debug interfaces by adding write functionality in addition to existing read-only APIs
  – Example: regulator fw microvolt nodes

• Add completely new interfaces where nothing exists currently

• Pros:
  – Having as much of the functionality available to userspace as possible makes it easy to write test scripts
  – Can dynamically create new test cases

• Cons:
  – Might not be possible to upstream these
Kernel PM features
Suspend

- Executed from command line (echo mem > /sys/power/state)
- Disables all drivers manually
  - Disables also trace!
- Tools for debugging:
  - Trace (limited)
  - Gpios
  - Debugger (with breakpoints)
- Tricks:
  - Prevent low level PM entry so that hardware is mostly taken out from debug process, re-enable once SW works
Cpuidle

• Bit complicated as can execute multiple different C states based on system status

• Tools for debugging:
  – Traces (limited) / gpios / ring buffer
  – Debugger can be used with breakpoints

• Debug information from userspace:
  – /sys/devices/system/cpu/*/cpuidle/*

• Tricks:
  – Export API to userspace to “force” a certain C state always
Regulators

• Userspace API available at /sys/class/regulator

• Easy to check the status of regulator framework against hardware status by using multimeter etc.

• Tricks:
  – Export write capability for microvolt nodes to userspace
  – Export regulator enable / disable to userspace
Clock framework

• Ongoing work within Linaro to get a common clock framework into the kernel

• Part of the code exists already but integration missing to most of the platforms

• Tools for debugging:
  – Traces
  – Register dumps vs. clock framework status

• Userspace interface:
  – /sys/kernel/debug/clk/*
  – Provides info for clock rates, usecounts, flags etc.
  – Easy to tweak to allow manual enable of clocks from userspace
Cpufreq

• Pretty easy to debug, as typically does not block any kernel functionality, and has nice APIs readily available from userspace

• Can usually trace through everything
  – May have a critical section that requires more complex debugging

• Debug information from userspace:
  – /sys/devices/system/cpu/*/cpufreq/*
  – /sys/class/regulator/*
  – /sys/kernel/debug/clk/* if available

• Tricks:
  – Select clock frequency manually with 'ondemand' governor by writing to 'scaling_min/max_freq' nodes
Devfreq

• Device specific DVFS, relatively similar to cpufreq
• Should also be possible to trace through everything
• Adds extra 'devfreq' directory under device sysfs node
Typical PM problems
Bootloader madness insanity

• Everybody uses a different bootloader

• Lots of features inside the bootloaders, which typically leave hardware enabled after use => prevents PM

• Quite often it might not be evident that bootloader is causing PM problems

• First thing to do when someone complains to you about PM issues, ask them what bootloader they are using
  – If not the same you have => its their problem
Device crash

• Device dies completely, either with or without a crash dump
• If with dump, just decipher the crash dump to figure out what happened
• If silent hang, try to pinpoint where the crash happens
  – Disable CONFIG_PM
  – Disable PM options one by one
  – Disable drivers to get a 'minimal kernel'
  – Add traces to code
  – Add breakpoints to potential crash locations etc.
Device malfunction

• Some driver starts misbehaving after a while
• Can take a long time to reproduce
• Maybe difficult to pinpoint the actual problem
• If you are lucky, might provide mysterious crash dumps related to the component in question
• Stress testing scripts might be useful
• Example problems:
  – Device stops responding to serial console after a while, but the kernel / interrupts still work
  – Memory corruption
Increased power consumption (1/3)

• Power source good initial indicator
  – e.g. battery dies too quickly compared to what it should be

• Check if cpuidle / suspend work properly and set the device to proper state
  – Sysfs status for cpuidle
  – timer_stats
  – Regulator status
  – Whatever else is exported to userspace from HW point of view

• Good if you have a working / non-working case where you can compare the system state
Increased power consumption – hardware (2/3)

• Hardware problems usually force higher power use than planned
  – May need to disable some power saving techniques
    • e.g. some regulator must always be 'enabled'
  – Some HW pulls are incorrectly designed and consume extra power

• Might be possible to reduce impact in some cases with software tweaks
  – E.g. align external pull vs. SoC configurable pull config

• May need to re-design hardware
Increased power consumption - userspace (3/3)

• Typical culprit for consuming too much power

• Some process is using too much resources for execution
  – Prevents cpuidle completely (cpu load) or partially (timer usage)

• Check out 'top' or something similar for CPU load

• /proc/timer_stats is good for figuring out timers that are used too often
Some references

- **Powertop**
  - Parses timer + process + interrupt info
  - [https://01.org/powertop/](https://01.org/powertop/)

- **Powerdebug**
  - Parses regulator + clock framework + sensor data

- **Tracepoints**
  - Kernel source: include/trace/events/power.h
  - Debugfs: /sys/kernel/debug/tracing/README
    - Parsers: (target) trace-cmd => (host) kernelshark, pytimechart
Thank you!