

# Deep Sleep by Design

## Building Linux Products that truly Sleep

Hagen Paul Pfeifer <[hagen@jauu.net](mailto:hagen@jauu.net)>

# Agenda

Module 1: **Physics of Power**

Module 2: **Poor Baseline & Measurement Setup**

Module 3: **Dynamic PM – Throttling** (CPUFreq)

Module 4: **Idle PM – Sleeping** (CPUIdle, Runtime PM, Suspend)

Module 5: **Analysis & Tooling** (powertop, perf, ftrace)

Module 6: **Application & System Design** – “Sleep-First”

Module 7: **Android**

**Wrap-up: Checklist & Resources**

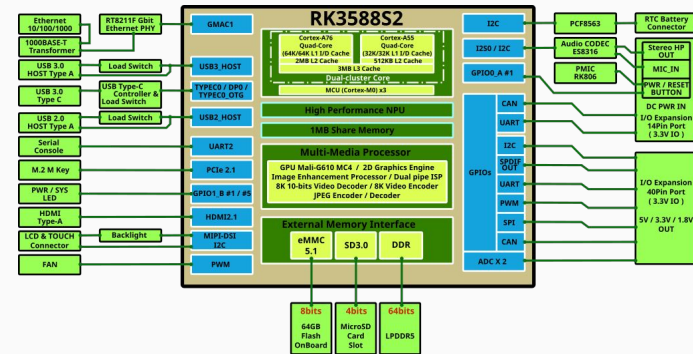
**Duration: 3h, no coffee-breaks**

# Before we start

**Goal: move from “my board idles at >1 W” to “measured deep sleep in the few-mW range” with a repeatable method**

**We'll work with one Arm64 SoC (Odroid M2); the patterns apply to other hardware (Intel, AMD, NXP, TI, Qualcomm, ...) even if the exact numbers differ**

**I assume you are comfortable with shell access, kernel configuration and rebuilding images; we won't cover basic Linux usage**



**Please interrupt me at any time with questions or real-world problems from your products; the more concrete, the better (this workshop is for you!)**

# **Module 1**

## The Physics of Power

# Why Power Is the Tightest Resource

**Power is not just one resource; it's a proxy for three:**

- Battery Life: The most obvious device constraint.
- Thermal Budget: Power (W) = Heat. Throttling is the failure of PM.
- System Cost: Smaller batteries, passive cooling, simpler power delivery.

**Goal is: not just "race to idle", enter deep sleep and stay there most of the time**

**Requires a system-wide view: HW + kernel + userspace together**

**We must combine hardware capabilities with the right software**

# Core Concept I: Throttling vs Shutdown

## Two basic levers: slow down vs turn off

### Throttling (Dynamic PM)

- Scale voltage/frequency to match load (DVFS)
- Analogy: dimming a light
- Linux: CPUFreq, DevFreq

### Shutdown (Idle PM)

- Gate clocks, power-gate blocks
- Analogy: flipping the light switch
- Linux: CPUIdle, Runtime PM, System Suspend

**Key: you cannot throttle to zero – deep sleep needs shutdown**

```
osc_24m: osc-24m {
    compatible = "fixed-clock";
    #clock-cells = <0>;
    clock-frequency = <24000000>;
};

ccm: clock-controller@12340000 {
    compatible = "acme,mychip-ccm";
    reg = <0x12340000 0x1000>;
    #clock-cells = <1>;
    clocks = <&osc_24m>;
    clock-names = "osc_24m";
    * 0 = GPU_AHB_ROOT
    * 1 = GPU_CORE_ROOT
    * 2 = I2C1_BUS_ROOT
    * 3 = I2C1_CORE_ROOT
    */
};

pmc: power-controller@12380000 {
    compatible = "acme,mychip-pmc";
    reg = <0x12380000 0x1000>;
    #power-domain-cells = <1>;

    pd_gpu: power-domain@0 { reg = <0>; };
    pd_periph: power-domain@1 { reg = <1>; };
};

soc: soc@0 {
    compatible = "simple-bus";
    #address-cells = <1>;
    #size-cells = <1>;
    ranges = <0x0 0x0 0x40000000>;

    gpu: gpu@40000000 {
        compatible = "acme,mychip-gpu";
        reg = <0x40000000 0x20000>;
        interrupts = <0 40 4>;

        clocks = <&ccm 0>, /* GPU_AHB_ROOT */
                <&ccm 1>; /* GPU_CORE_ROOT */
        clock-names = "reg", "core";

        power-domains = <&pmc 0>; /* pd_gpu */
        status = "disabled";
    };

    i2c1: i2c@41000000 {
        compatible = "acme,mychip-i2c";
        reg = <0x41000000 0x1000>;
        interrupts = <0 41 4>;

        clocks = <&ccm 2>, /* I2C1_BUS_ROOT */
                <&ccm 3>; /* I2C1_CORE_ROOT */
        clock-names = "bus", "core";

        power-domains = <&pmc 1>; /* pd_periph */
        status = "disabled";
    };
};
```

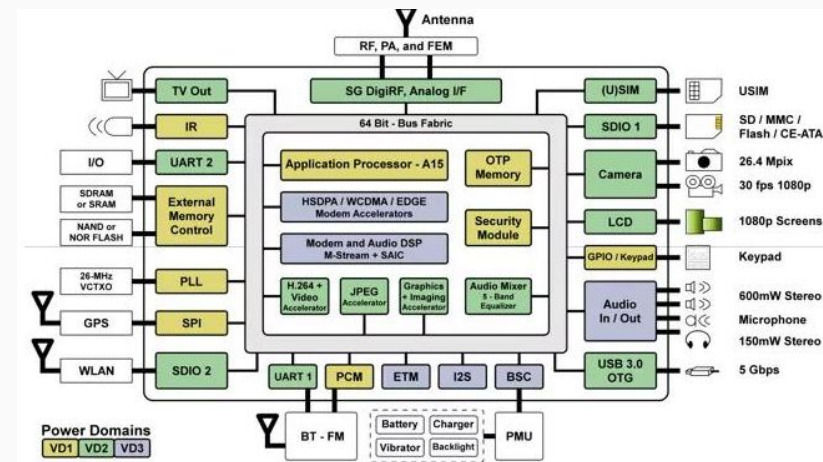
# Core Concept II: Clock vs Power Domains

## Clock domain (clock gating)

- Stop clock to a block (e.g., UART)
- Very fast to enter/exit
- Still leaks static power

## Power domain (power gating)

- Remove supply from a block (e.g., GPU, CPU core)
- Near-zero leakage, but slower, state lost



**Modern SoCs: many independent clock & power domains**

**PM is about using these domains aggressively without breaking UX/requirements**

# Core Concept III: State Retention & Wake Sources

## **State retention problem**

- Power-gated domains lose registers, caches, context

## **Retention solutions**

- Retention flip-flops, retention SRAM, “retention modes”
- Trade-off: less power savings vs faster, easier resume

## **Deeper sleep (e.g. Suspend-to-RAM / S3)**

- Only DRAM retained in self-refresh

## **Wake sources**

- Typical: GPIO button, RTC alarm, Ethernet WoL, USB, modem
- Must explicitly enable only the wake sources you really want



# Platform Divergence: ACPI vs Device Tree

**Kernel frameworks are generic**

**Platform describes actual capabilities**

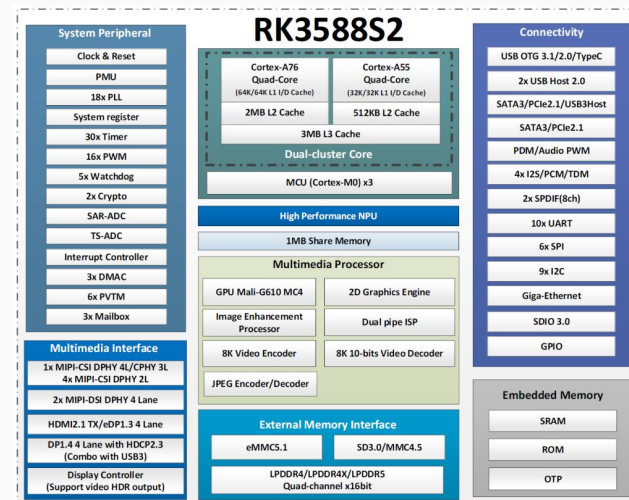
## **x86 / Intel Atom**

- ACPI firmware advertises C-states, P-states, S-states
- Drivers: intel\_idle, intel\_pstate, acpi-cpufreq

## **ARM / ODROID-M2 (RK3588S2)**

- Uses Device Tree + PSCI for idle/suspend
- idle-states and power-domains defined in DTS
- Drivers: cpuidle-psci / cpuidle-dt, cpufreq-dt, genpd

**Takeaway: same kernel concepts, different description layers**



## **Module 2**

### The “Poor Baseline” & Measurement Setup

# Case Study Hardware: ODROID-M2 & Intel Atom

## **ARM board: ODROID-M2 (RK3588S2)**

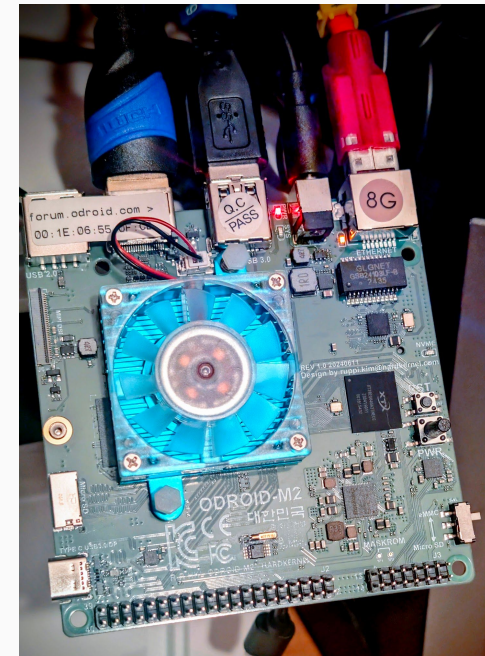
- 4× Cortex-A76 + 4× Cortex-A55, big.LITTLE
- DT-based clocks, power-domains, idle-states

## **x86 board: Intel Atom (E3900-class)**

- ACPI-based C/P/S-states
- intel\_idle and intel\_pstate on modern kernels

**We'll compare patterns on both, not chase board quirks**

**Focus: methodology, not exact numbers**



# Hardware is Destiny I: The SoC

**Software configures, Hardware executes. Your minimum power consumption is set by your hardware choices.**

**Key SoC features for low power:**

- Granular, independent power domains
- Fine-grained clock gating
- An optimized PMIC (Power Management IC) designed for the SoC

**Look for SoCs designed for low-power modes**

**Critical: Does the SoC have mainline Linux support?**

# Hardware is Destiny II: Peripherals & Drivers

**A single "bad" peripheral can veto sleep for the entire system**

**All peripherals (Ethernet, WiFi, sensors) must correctly implement:**

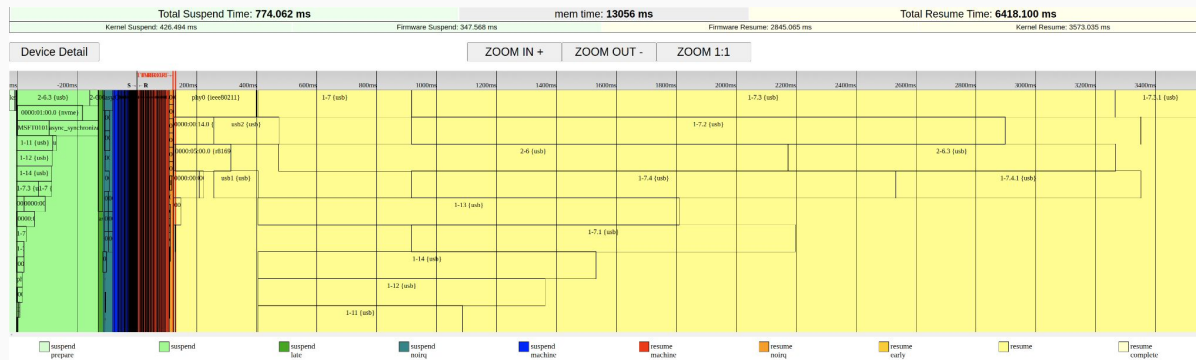
- Runtime PM (autosuspend) for idle savings
- System Suspend hooks for deep sleep.

**Common Problem: USB**

- Many USB devices (or even host controllers) have poor autosuspend support in hardware or drivers

**Choose components with known, high-quality mainline drivers that fully support PM - verify things on reference systems**

sleepgraph.py:



# Lab 1: The Deliberately Bad Baseline

## Kernel config:

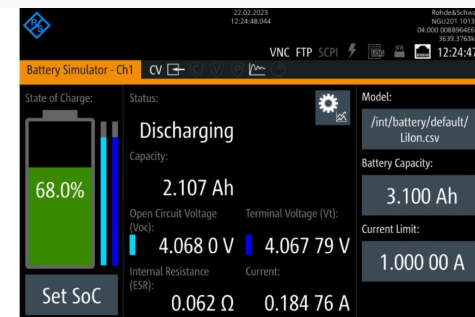
- CONFIG\_PM = n (PM core frameworks off)
- CONFIG\_HZ = 1000 (1000 timer ticks per second)

## Userspace:

- CPUFreq governor (if present) = performance
- Stock distro, no tuning, all services on

## Lab steps:

- Boot, login over serial / UART
- Let system "idle" at shell prompt
- Measure total board power – this is our baseline



# Measurement I: Shunt Resistor “Ground Truth”

**Software estimates (e.g., powertop) are guidance, not truth**

## **Hardware measurement:**

- Insert precise shunt (e.g. 100 mΩ) in main supply path
- Measure voltage drop across shunt with DMM/ADC
- Be

## **Use Ohm's law:**

- $I = V / R$ ,  $P = V_{\text{supply}} * I$

**Use Kelvin (4-wire) connection for accuracy**

**Low-side shunt (between load and GND) is usually simpler**



# Measurement II:

## GPIO Toggle for Resume Latency

**Kernel timestamps are unreliable across suspend/resume**

**Use GPIO + scope/LA for real latency:**

- Select free GPIO pinned out on header
- In IRQ / wake handler: set GPIO high ASAP
- In late resume / userspace ready: set GPIO low

**Measure pulse width on scope  $\Rightarrow$  end-to-end resume time**

**Repeat for:**

- s2idle vs suspend-to-RAM
- Different wake sources (GPIO vs RTC vs network)



# Lab 1 Result: The Bad Idle

## **Example (numbers will differ on your boards):**

- ODROID-M2 idle: ~1.25 W

## **Why so bad?**

- PM frameworks disabled (CONFIG\_PM=n)
- CPU stuck in C0, often at max frequency
- All peripherals powered, clocks running
- Timer tick at 1000 Hz keeps waking CPU

**This is our reference to compare all later improvements**

## **Module 3**

Dynamic PM:  
Throttling  
(CPUFreq)

# Dynamic PM: DVFS via CPUFreq

**First optimistic idea: reduce power while active**

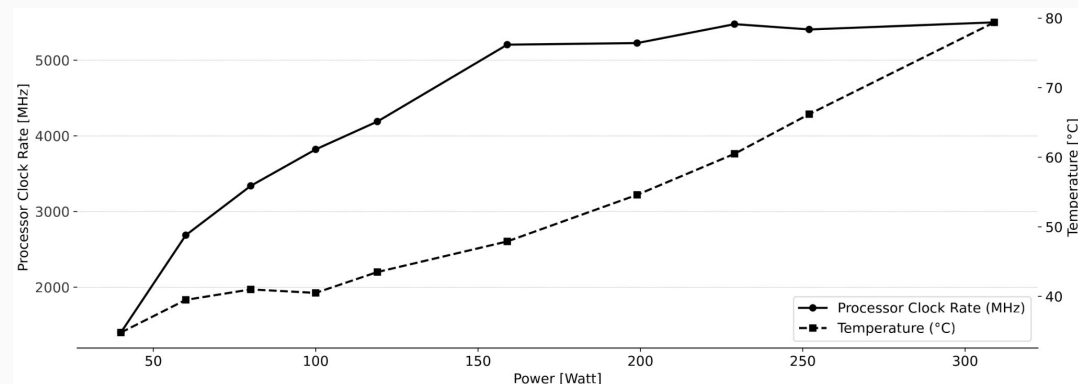
**DVFS = Dynamic Voltage & Frequency Scaling**

**Linux CPUFreq subsystem:**

- Core: policy + sysfs (/sys/devices/system/cpu/.../cpufreq/)
- Driver: platform-specific, talks to PLL / regulators
- Governor: policy algorithm choosing frequency

**Good for reducing active power**

**Spoiler: doesn't fix idle power**



AMD Ryzen 9950X

# CPUFreq Governors in Practice

## **performance**

- Always max freq, our baseline

## **powersave**

- Always min freq, hurts latency/UI

## **ondemand / conservative**

- Legacy, reactive to CPU load, “bouncy”

## **schedutil (default on modern ARM64)**

- Integrated with CFS scheduler
- Uses scheduler load tracking for DVFS

**For most arm64 systems: use schedutil unless you have a reason not to**

# Intel Platform: intel\_pstate vs acpi-cpufreq

## acpi-cpufreq

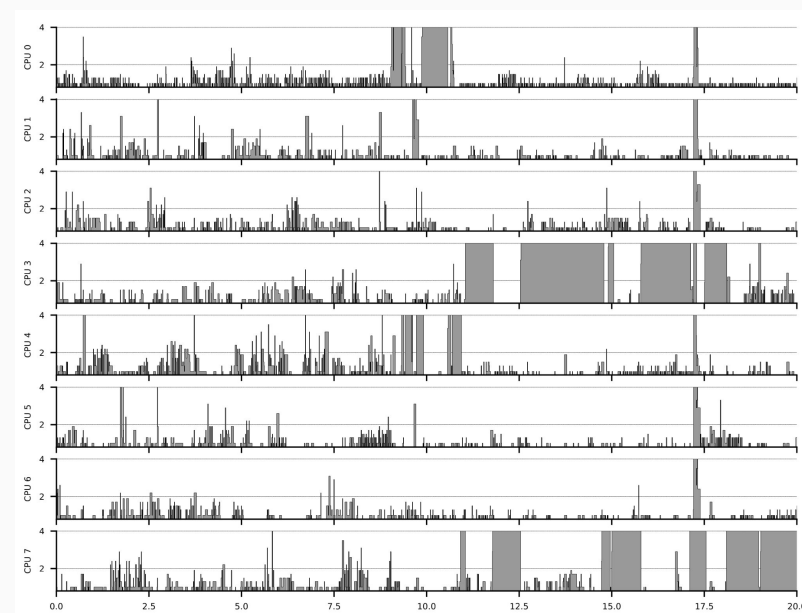
- Trusts ACPI \_PSS tables from firmware
- Exposes standard governors (ondemand, schedutil, ...)

## intel\_pstate (default on modern Intel/Atom)

- Driver and governor combined
- Uses hardware/firmware feedback to choose P-states
- Exposes “performance” and “powersave” policies
- Usually better than acpi-cpufreq

**Override: intel\_pstate=disable on kernel cmdline if you must**

**Note:** modern x86-64 based systems use "HWP" based approaches (hw controlled, not kernel)



# ARM Platform: cpufreq-dt and OPP Tables

## **Generic DT-based CPUFreq driver: cpufreq-dt**

### **Uses Operating Performance Points (OPP):**

- Frequency / voltage pairs
- Defined via operating-points-v2 in DT

### **Example (generic Cortex-A53 SoC, not M2-specific):**

- 1.5 GHz @ 1.10 V
- 1.2 GHz @ 1.05 V
- 667 MHz @ 0.94 V, etc.

**If the OPP table is wrong, your DVFS is wrong**

**For RK3588S2, vendor trees usually provide OPPs – check them**

# DevFreq: DVFS Beyond the CPU

**CPU is not the only power hog**

**DevFreq: generic DVFS for devices (GPU, DDR, ISP...)**

**Same pattern as CPUFreq:**

- Core + driver + governor

**Governors: ondemand, performance, powersave, passive**

**Check `/sys/class/devfreq/` for available devices**

**Under load, DevFreq can matter as much as CPUFreq**

# Lab 2: Enabling CPUFreq

## **Kernel config (simplified):**

- CONFIG\_PM = y
- CONFIG\_CPU\_FREQ = y
- CONFIG\_CPU\_FREQ\_GOV\_SCHEDUTIL = y

## **Steps:**

- Load governor: modprobe cpufreq\_schedutil if modular
- echo schedutil > .../cpufreq/scaling\_governor
- Watch scaling\_cur\_freq under load

## **Measurement:**

- Repeat idle power measurement
- Result: idle stays basically the same

**Lesson: DVFS helps under load, not when “idle”**



## **Module 4**

### Idle PM: Sleeping

# Idle PM: What We Actually Need

**To cut idle power, we must turn things off**

**Three separate frameworks:**

- CPUIdle – idle states for CPUs
- Runtime PM – idle states for individual devices
- System Suspend – whole-system low-power states

**Mixing them up is the #1 design error**

**We'll walk each in turn, then combine them**

# Framework 1: CPUIdle Basics

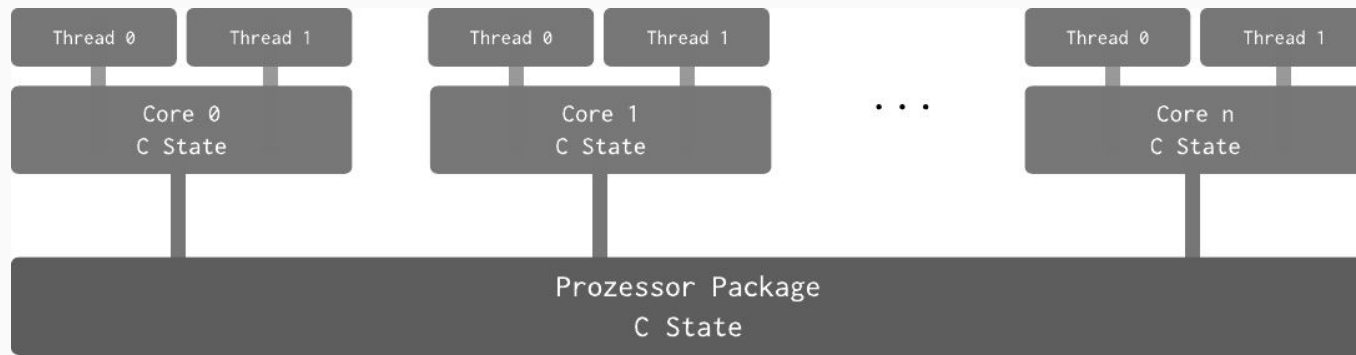
**Trigger: CPU has no runnable tasks (idle loop)**

**CPUIdle components:**

- Core: generic cpuidle framework
- Driver: platform-specific C-states entry/exit
- Governor: chooses which C-state to use

**“Opportunistic sleep”:**

- Sleep whenever the CPU is idle
- Wake on any interrupt or timer



# CPUIdle Deep Dive: C-States & menu Governor

## **ACPI naming: C0, C1...Cn**

- C0: running
- C1: simple halt/wfi, fast, modest savings
- Deeper C-states: clock gating, power gating, big savings, high latency

## **Each state has:**

- Entry/exit latency
- Target residency (time to break even)

## **menu governor:**

- Predicts next wakeup (timers, I/O, scheduler)
- Selects deepest state that fits within latency + PM QoS limits

# Intel: intel\_idle Driver

**intel\_idle bypasses ACPI \_CST and uses Intel tables**

**Enters C-states via MWAIT with hints**

**Exposes C-states under:**

- /sys/devices/system/cpu/cpu0/cpuidle/state\*/

**Debug:**

- state\*/name (C1, C3, C6, ...)
- state\*/time (total residency)

**Tune C-state usage with BIOS/firmware and PM QoS**

# C State Transition Tracking

## Individual C State transitions can be tracked with Perf

```
$ perf record -a -e power:cpu_idle -- sleep 60
$ perf script --gen-script python
$ vim perf-script.py
$ perf script -s ./perf-script.py
  2 36002.863851506 state=4294967295, cpu_id=2
 18 36002.863873513 state=4, cpu_id=18
  2 36002.864113542 state=4, cpu_id=2
 18 36002.864140040 state=4294967295, cpu_id=18
  2 36002.864272308 state=4294967295, cpu_id=2
 16 36002.864349085 state=4294967295, cpu_id=16
 18 36002.864381491 state=4, cpu_id=18
  0 36002.864396402 state=4294967295, cpu_id=0
 12 36002.864403081 state=4294967295, cpu_id=12
  0 36002.864415912 state=4, cpu_id=0
 12 36002.864447521 state=4, cpu_id=12
  2 36002.864472649 state=4, cpu_id=2
 16 36002.864490568 state=4, cpu_id=16
```

# ARM: DT idle-states + PSCI

**Many ARM64 platforms use PSCI for CPU idle/suspend**

**DT idle-states node describes:**

- entry-method = "psci"
- min-residency-us
- entry-latency-us, exit-latency-us

**Driver: cpuidle-psci / cpuidle-dt**

**As on x86, deep states use power gating + longer latency**

**Bad DT = bad idle behaviour, no matter what the kernel wants**

# The Tick Problem: Why CPUIdle Isn't Enough

**CONFIG\_HZ = periodic scheduler tick (100–1000 Hz)**

**Each tick is an interrupt that wakes the CPU**

**Even if idle, CPU keeps popping back to C0 to do nothing**

**Result: cannot stay in deep C-states long enough**

**The periodic tick is often the single worst enemy of low idle power**



# Tickless Idle (CONFIG\_NO\_HZ\_IDLE)

**Enable tickless idle: CONFIG\_NO\_HZ\_IDLE = y**

**Behaviour:**

- When CPU goes idle, kernel stops periodic tick
- Looks ahead to next real timer event
- Programs a one-shot timer for that time
- Lets CPUIdle pick a deep state for the whole interval

**Allows multi-second deep C-state residency**

**Absolutely mandatory for good opportunistic sleep**

**Beware:** majority of these tips guide to a power optimized system - not max-throughput/realtime optimized system

# Framework 2: System Suspend

## **System-wide low-power state (S-states / suspend)**

### **Entry path:**

- Userspace initiates (e.g., `echo mem > /sys/power/state`)
- Freeze userspace tasks
- Suspend devices (top-down prepare, bottom-up suspend)
- Enter platform suspend state (s2idle, S3, etc.)

### **Modes:**

- freeze / s2idle: CPUs use CPUIdle, rest mostly stays on
- mem: Suspend-to-RAM / S3, CPU off, DRAM in self-refresh
- disk: Hibernate / S4

**For deep sleep, we care about mem**

# Framework 3: Runtime PM (RPM)

## **Runtime PM for individual devices**

### **Transparent to userspace**

#### **Based on a usage counter per device:**

- `pm_runtime_get*()` – increment, resume device
- `pm_runtime_put*()` – decrement, maybe idle/suspend device

#### **Autosuspend:**

- Delay before suspend to avoid bouncing the power state

**Goal: if device is not used, it should sleep, even if CPU runs**

# Generic Power Domains (genpd)

**Some devices share a power controller (shared rail)**

**GenPD models power domains in kernel:**

- Devices attach to domains via DT power-domains
- Domain only powered off when all member devices can sleep

**Links Runtime PM with real power rails**

**Without GenPD, Runtime PM can't actually gate leakage-heavy blocks**

# Lab 3: Turning on Idle PM

## Kernel config (simplified):

- CONFIG\_PM = y
- CONFIG\_PM\_SLEEP = y
- CONFIG\_CPU\_IDLE = y
- CONFIG\_NO\_HZ\_IDLE = y
- CONFIG\_PM\_RUNTIME = y
- Platform drivers: CONFIG\_INTEL\_IDLE or CONFIG\_CPUFREQ\_DT, PSCI, GenPD

## Checks:

- `cat cpu0/cpuidle/state*/name` (C-states visible?)
- For a device: `.../power/control = auto`
- `.../power/runtime_status` → `suspended`

## Measurement:

- Idle power should drop significantly vs Lab 1

# **Module 5**

## Analysis & Tooling

# What's Still Keeping Us Awake?

**System can sleep, but maybe it does not**

**Two main reasons:**

- Vetoes – explicit “do not sleep” constraints
  - PM QoS latency limits
  - Wakelocks / wakeup sources
- Wakeups – noisy activity resetting idle timer
  - Chatty apps, kernel threads
  - Interrupt storms, periodic timers

**Our job: find the veto or wakeup, then remove or tame it**

# Veto 1: PM QoS

**PM QoS = Power Management Quality of Service**

**Allows code to say:**

- “I need wakeup within  $N \mu\text{s}$ ”

**Interface (userspace): `/dev/cpu_dma_latency`**

- Open  $\rightarrow$  register a request
- Write s32 latency ( $\mu\text{s}$ )  $\rightarrow$  update request
- Close  $\rightarrow$  remove request

**Effect: CPUIdle will avoid deep C-states with higher latency**

**Debug:**

- Check existing constraints with `pm_qos debugfs / docs`



# Veto 2: Wakelocks / Wakeup Sources

**Wakeup source framework (Android “wakelocks” concept)**

**If a wakeup source is active, system suspend is blocked**

**Two types conceptually:**

- Kernel wakelocks (held in drivers / subsystems)
- Userspace wakelocks via `/sys/power/wake_lock` (if enabled)

**A single stray wakelock often explains “suspend never works”**

# Wakelocks from Userspace

## **Userspace interface (when USER\_WAKELOCK enabled):**

### **Acquire lock:**

```
echo "my_app_lock" > /sys/power/wake_lock
```

### **Release lock:**

```
echo "my_app_lock" > /sys/power/wake_unlock
```

### **Inspect active locks:**

```
cat /sys/power/wake_lock  
cat /sys/kernel/debug/wakeup_sources
```

**Never leave a lock held longer than necessary**

# Tooling I: powertop Overview

## **First responder for power debugging**

### **Run calibration once on battery:**

- `powertop --calibrate`

### **Key tabs:**

- Overview – estimated power, main hogs
- Idle stats – C-state residency per CPU
- Frequency stats – P-state residency
- Device stats – per-device estimates
- Tunables – easy toggles for common settings
- WakeUp – per-process and IRQ wakeups

# powertop Tunables & WakeUp

## **Tunables tab:**

- “Good” vs “Bad” settings
- Typical fixes:
  - USB autosuspend to “Good”
  - Disable NMI watchdog
- `powertop --auto-tune` to flip everything to “Good”

## **WakeUp tab:**

- Sort by wakeups/sec
- Identify top offenders: processes, timers, interrupts
- These are your prime suspects

# Tooling II: perf for Profiling

**powertop says who, perf tells you why**

**perf top**

- Live view of hottest functions
- Use to spot misbehaving code under idle load

**perf record + perf report**

- Record CPU cycles, tracepoints, callgraphs
- Offline analysis of problematic intervals

**We'll use perf with PM tracepoints to see wake paths**

# Using perf to Trace Wakeups

**Goal: catch full stack that wakes the CPU from idle**

**Useful tracepoints:**

- power:cpu\_idle, power:cpu\_frequency, power:cpu\_wakeup

**Example:**

```
perf record -g -a -e power:cpu_idle -e power:cpu_wakeup sleep 10  
perf report --sort=trace
```

**Output: processes + kernel stack causing wakeups**

**Great for hunting “mystery” wakeups**

# Tooling III: ftrace as a Scalpel

**ftrace = built-in kernel tracer via tracefs (back in the days: debugfs)**

## **Mount tracefs:**

```
mount -t tracefs none /sys/kernel/tracing
```

- Interface: /sys/kernel/tracing

## **Modes:**

- function, function\_graph – heavy, but detailed
- events – trace specific subsystems (e.g., power/\*)

## **trace-cmd / kernelshark make life easier**

- raw fs is fine/great for embedded

# ftrace Example: PM Events

## **Simple PM tracing setup:**

```
cd /sys/kernel/debug/tracing  
echo 1 > events/power/enable  
cat trace_pipe
```

## **You see live events:**

- cpu\_idle\_enter / cpu\_idle\_exit
- pm\_runtime\_suspend / pm\_runtime\_resume
- suspend\_resume phases

**Combine with rtcwake tests to understand full PM behaviour**



# Lab 4: Killing Wakeups

**Step 1: powertop --auto-tune**

**Step 2: In WakeUp tab, identify top noise sources:**

- sshd, serial login
- klogd spamming logs
- USB mouse / keyboard polling
- jbd2 / filesystem journal write
- NMI watchdog

**Step 3: Fix one by one, re-measure each time**

**Goal: stable, very low idle in s2idle**

**Typical: down to tens of milliwatts on ARM boards**

## **Module 6**

### Application & System Design ("Sleep-First")

# The “Sleep-First” Design Philosophy

**After Module 5, s2idle idle is reasonably low**

**For an IoT-style device, it’s still not enough**

**Difference:**

- s2idle – cores off, but DRAM + many blocks stay powered
- Suspend-to-RAM – only DRAM in self-refresh, rest off

**Philosophy:**

- System should be in Suspend-to-RAM by default
- Wake only to do useful work
- Immediately go back to deep sleep

# Example Use-Case: “TX Data Every 5 Minutes”

## **Typical battery product pattern:**

- Sample sensors
- Bring up radio / Wi-Fi / LTE
- Send small payload
- Go back to deep sleep

## **Requirements:**

- Stable RTC or equivalent wake source
- All drivers & PM paths working in suspend-to-RAM
- Very predictable wake-to-work latency

# The Wrong Way: sleep 300

## Naive userspace loop:

```
while true; do  
    send_data  
    sleep 300  
done
```

## Problems:

- sleep just blocks the process
- System stays in s2idle / idle C-states
- Idle power maybe ~0.05 W, not  $\mu$ W-level
- Battery dies much sooner than necessary

# The Right(er) Way: rtcwake

## Use hardware RTC alarm + deep suspend

### Pattern:

```
while true; do  
    send_data  
    rtcwake -m mem -s 300  
done
```

-m mem → Suspend-to-RAM (ACPI S3 / equivalent)

-s 300 → RTC wakes system after 300 seconds

**Between wakeups, board sits in true deep sleep**

# rtcwake Deep Dive

**Syntax: `rtcwake [options] -m <mode> {-s <seconds> | -t <time_t>}`**

## **Useful modes:**

- standby – shallow sleep, fast resume
- mem – Suspend-to-RAM
- disk – hibernate to storage
- off – full poweroff with RTC wake if supported
- no – set alarm only, don't suspend

**You can also program RTC directly via `/sys/class/rtc/rtc0/wakealarm`**

**Always test that RTC alarms really wake from your chosen state**

# Lab 5: Implement Periodic Task Properly

## **Implement two versions of “send every 5 minutes”:**

- Version A: sleep 300 loop
- Version B: rtcwake -m mem -s 300 loop

## **Measure both with shunt:**

- A: idles in s2idle, e.g.  $\sim 0.05$  W
- B: idles in suspend-to-RAM, e.g.  $\sim 0.005$  W

## **Use GPIO toggle to measure wake-to-send latency**

**Ask: does latency and energy meet your product spec?**



# **Module 7**

Use Case: Android

# Android-Style Deep Sleep: Big Picture

## **Android goal: be asleep as much as possible**

- Opportunistic suspend: system goes to suspend-to-RAM whenever no WakeLocks are held
- Framework + kernel cooperate: PowerManagerService + suspend blockers in kernel
- Default state for phones: screen off → deep sleep, not just “idle”

## **Key idea: Deep sleep is the normal state, wake is the exception**



# Android-Style Deep Sleep: Big Picture

## **Android goal: be asleep as much as possible**

- Opportunistic suspend: system goes to suspend-to-RAM whenever no WakeLocks are held
- Framework + kernel cooperate: PowerManagerService + suspend blockers in kernel
- Default state for phones: screen off → deep sleep, not just “idle”

**Key idea: Deep sleep is the normal state, wake is the exception**

# WakeLocks & Suspend Blockers

## **WakeLock (framework)**

- App / service tells system: “stay awake until I’m done”
- Partial WakeLock: keep CPU running, screen can be off
- Full WakeLock: keep CPU + display on (rarely used)

## **Suspend blocker / wakeup source (kernel)**

- Kernel-side object that vetoes suspend while held
- Drivers hold it while processing critical work, then release immediately

**Key idea: Suspend is allowed only when no WakeLock / suspend blocker is active**

# Orchestrating Suspend Safely

## **Suspend entry sequence**

- Framework asks kernel for current wakeup\_count
- Framework arms suspend and writes wakeup\_count back
- If count changed in between, kernel rejects suspend (race detected)

## **During suspend**

- Only marked wakeup IRQs stay enabled
- Any wakeup IRQ can abort suspend and bring system back to running

**Key idea: No events are lost between “I want to sleep” and “I am sleeping”**

# Timed Work – AlarmManager, JobScheduler, WorkManager

## **AlarmManager**

- Time-based wakeups (RTC or elapsed realtime)
- “Wakeup” alarms can bring device out of suspend
- Expensive: each alarm = full SoC wake

## **JobScheduler**

- System batches background work from many apps
- Jobs have constraints: charging, unmetered network, idle, etc.
- Scheduler aligns jobs to existing wakeups to avoid extra resumes

## **WorkManager**

- Higher-level API on top (uses JobScheduler / alarms internally)
- Respects system power policy (Doze, App Standby) automatically

# Doze & App Standby – System-Level Policy

## **Doze mode**

- Triggers after long screen-off + device still
- Network access, background jobs, normal alarms are deferred
- System opens short “maintenance windows” to flush queued work

## **App Standby**

- Per-app buckets: active, working set, frequent, rare
- Idle apps get stricter limits on jobs, alarms, network

**Key idea: Global policy layer that throttles background work even when Apps were missed**

# Wakeup Sources & Hardware Policy

## Wakeup-capable devices

- Only selected IRQs allowed to wake the SoC: power button, RTC, modem, selected sensors
- Drivers mark wakeup IRQs explicitly (e.g., `enable_irq_wake`)

## Handling wakeups

- Non-wakeup IRQs fully disabled in suspend
- First wakeup IRQ aborts suspend; handler runs after resume

## Design rule

- Minimize number of wakeup sources
- Prefer co-processors / sensor hubs that pre-filter events and wake SoC only on “real” activity



# Applying This to Embedded Linux

## **From sleep 300 to Android-style design**

- Use hardware wake sources (RTC, GPIO, modem), not busy userspace loops
- Model long-running work as short wake cycles + quick return to mem
- Batch periodic tasks: one wakeup → do all pending sensor reads, logging, uploads

## **Borrow from Android**

- Introduce a simple WakeLock-like API for your daemons (userspace or kernel)
- Add a central “power manager” process that decides when suspend is allowed
- Use `rtcwake -m mem ...` (or direct `/sys/power/state`) only when no internal locks are held
- 

**Key idea: Recreate the pattern: central power arbiter + wakeup-count + minimal wake sources → maximal deep-sleep time**

# **Module 8**

## Conclusion & Checklist

# Summary: Baseline vs Optimized

## **Stepwise improvement path (example numbers):**

- Lab 1 – Bad baseline: ~1.25 W
- Lab 2 – +CPUFreq: ~1.25 W (no idle gain)
- Lab 3 – +CPUIdle + tickless: ~0.35 W
- Lab 3 – +Runtime PM & GenPD: ~0.10 W
- Lab 4 – +Wakeup fixes: ~0.05 W (good s2idle)
- Lab 5 – rtcwake -m mem: ~0.005 W (deep sleep)

**Net improvement: >99% reduction in idle power**

**Same boards, just smarter use of kernel PM features**

# Programming Language Selection

Tier 1

**C / C++ / Rust / Zig**

Tier 2

**Go / Swift / Kotlin / Java / C#**

Tier 3

**Python / JS / Ruby / Perl / Lua**

# The Sleep-First Checklist

## Hardware

- Peripherals with proper Runtime PM support?
- Clean wake sources (RTC, GPIO, maybe WoL)?

## ARM DT / Intel ACPI

- DT: clocks, power-domains, idle-states correctly described?
- ACPI: usable \_CST, \_PSS, \_S3 in firmware?

## Kernel config

- CONFIG\_NO\_HZ\_IDLE, CONFIG\_CPU\_IDLE, CONFIG\_PM\_RUNTIME, CONFIG\_PM\_SLEEP enabled?

## Drivers

- All custom drivers implement dev\_pm\_ops (runtime + system)?

## Userspace

- Periodic work uses rtcwake, not sleep
- No stray PM QoS constraints or wakelocks

## Validation

- Power: shunt resistor + logging
- Latency: GPIO + scope or logic analyzer

# Resources & Further Reading

## **Linux kernel docs (current tree):**

- Documentation/admin-guide/pm/\* (cpuidle, cpufreq, suspend, QoS, runtime PM)

## **Devicetree bindings:**

- idle-states, power-domain bindings, OPP / operating-points-v2

## **Good slide decks / talks:**

- “Linux Kernel Power Management: An Overview” (Linaro)
- Recent PM talks from Bootlin, Linaro, ELCE/EOSS

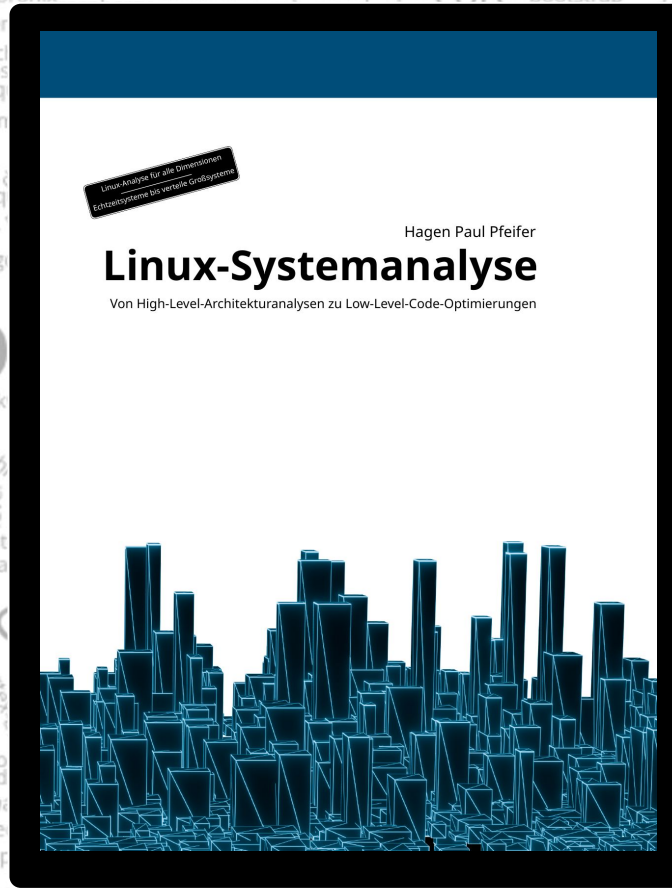
## **Tools:**

- powertop, perf, trace-cmd / kernelshark docs

**For later: bring your own board and repeat this exact pipeline**

# Linux Systemanalyse

Short commercial break:  
If you like this workshop, you might like the book.



SCAN ME

<https://jauu.net/linux-analyse/>

# Thank you very much!

## **Let's Tackle Your Questions!**

Got more questions? Feel free to catch me at the event or email me!  
[hagen@jauu.net](mailto:hagen@jauu.net)



## **Copyright © 2025 Hagen Paul Pfeifer**

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Names and brands may be claimed as the property of others. Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

All statements here are made in a personal capacity and are not affiliated with my employer.

This presentation was created with the highest quality standards, but errors may still be present. No warranties are made for any damages resulting from text or code examples.

License: This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).  
Redistribution is encouraged!