Linux Power Management Features, Part 2

Théo Lebrun
theo.lebrun@bootlin.com

Embedded Linux Conference
Europe 2025

Théo Lebrun

- Embedded Linux engineer and trainer at **Bootlin**
- Joined Bootlin in 2022, following an internship
  - Linux **kernel driver development** on embedded systems
    - Suspend-to-RAM for TI J7200 SoC
    - Upstreaming of Mobileye SoCs
  - PipeWire ecosystem
  - Open-source focus
- Current maintainer of elixir.bootlin.com
- Living in *Lyon*, not-south-nor-north east of France
- theo.lebrun@bootlin.com

Agenda

Audience: kernel driver developers

Goal: spread the word about existing device PM APIs

Covered last year:

- System-wide suspend
- Runtime power management
- Interactions between the two

This year:

1. How to reason about PM?
2. Linux PM features
    - Runtime PM features
    - Power domains
    - PM QoS

How to reason about PM?

How to reason about PM?

- What is it?
  - Turning off unneeded hardware capabilities
  - *Or slowing it down*
- Why we care?
  - Mostly to last longer on batteries
- How to do it?
  - It is a tradeoff
  - Either hardware resources are available now
  - Or they are accessible after some latency amount

How to reason about PM?

- Is it complex?
  - Yes
- Throttling makes this more than a one dimension axis
  - HW is running at reduced speed
  - Do you keep it as is when a request comes in?
  - Or you pay the transition cost to get full HW capabilities?
- Decisions are platform dependent
  - What can your hardware do?
  - What is most efficient?
  - Slow and steady? Race to halt?
- Decisions are project dependent
  - What is your workload?
  - Can you measure it?
  - What is acceptable?

How to reason about PM?

- Some more issues?
  - Yes
- Making non-platform specific code is hard
  - For a given device, each resource can be in a few different states
  - Platform state is exponentially large
  - Current solution: under-define power states
  - Example: network switch over PCI. What does "*it is runtime suspended*" mean?
- The "*we aren't alone*" problem
  - Boards aren't single-CPU anymore
  - Sharing resources is hard
  - Don't assume we have full knowledge
  - Don't assume we are making decisions

Runtime PM

Runtime PM: introduction

- Per device operations
- Suspend & resume
- Refcount-based
- Tree device hierarchy

include/linux/pm.h

```
// Device API
struct dev_pm_ops {
        /* Device is active but not needed anymore. */
        int (*runtime_suspend)(struct device *dev);
        /* Device is suspended but needed. */
        int (*runtime_resume)(struct device *dev);
        /* ... */
};
```

include/linux/pm_runtime.h

```
// Consumer API
void pm_runtime_enable(struct device *dev);
void pm_runtime_disable(struct device *dev);
int pm_runtime_get(struct device *dev);
int pm_runtime_get_sync(struct device *dev);
int pm_runtime_put(struct device *dev);
int pm_runtime_put_sync(struct device *dev);
```

Runtime PM: dummy example

```
static int x_probe(struct platform_device *pdev)
{
        pm_runtime_enable(&pdev->dev);
        pm_runtime_get_sync(&pdev->dev);
        return 0;
}

static int x_runtime_suspend(struct device *dev) { /* ... */ }
static int x_runtime_resume(struct device *dev)  { /* ... */ }

static const struct dev_pm_ops x_dev_pm_ops = {
        RUNTIME_PM_OPS(x_runtime_suspend, x_runtime_resume, NULL)
};

static struct platform_driver x_driver = {
        .probe = x_probe,
        .driver.pm = pm_ptr(&x_dev_pm_ops),
};
module_platform_driver(x_driver);
```

Runtime PM without callbacks

- What about runtime PM without callbacks registered?

```
static int x_probe(struct platform_device *pdev)
{
        pm_runtime_enable(&pdev->dev);
        pm_runtime_get_sync(&pdev->dev);
        return 0;
}

static struct platform_driver x_driver = {
        .probe = x_probe,
};
module_platform_driver(x_driver);
```

Runtime PM without callbacks: device model hierarchy

- First reason: waking up the device model hierarchy
- Apart from calling callbacks, runtime PM has other side effects: parent devices might? get resumed as `pm_runtime_get(dev)` calls `pm_runtime_get(dev->parent)`
  - ? not if parent is already active
  - ? not if parent is disabled
  - ? not if parent ignores his children
- The parent refcount, `dev->power.usage_count`, is always incremented however

Runtime PM without callbacks: device model hierarchy

- Example: `dw_spi_pci`, a SPI controller driver on the PCI bus.
- It has no `->runtime_suspend|resume()` callbacks.

drivers/spi/spi-dw-pci.c

```
static SIMPLE_DEV_PM_OPS(dw_spi_pci_pm_ops, dw_spi_pci_suspend, dw_spi_pci_resume);

static struct pci_driver dw_spi_pci_driver = {
        .name =    DRIVER_NAME,
        .id_table = dw_spi_pci_ids,
        .probe =   dw_spi_pci_probe,
        .remove = dw_spi_pci_remove,
        .driver         = {
                .pm     = &dw_spi_pci_pm_ops,
        },
};
module_pci_driver(dw_spi_pci_driver);
```

Runtime PM without callbacks: device model hierarchy

- It asks the SPI subsystem to do runtime PM operations on its behalf.

drivers/spi/spi-dw-core.c

```
struct spi_controller *host = spi_alloc_host(dev, 0);
if (!host)
        return -ENOMEM;

// ...
host->max_speed_hz = dws->max_freq;
host->flags = SPI_CONTROLLER_GPIO_SS;
host->auto_runtime_pm = true;
// ...

ret = spi_register_controller(host);
if (ret) {
        dev_err_probe(dev, ret, "problem registering spi host\n");
        goto err_dma_exit;
}
```

Runtime PM without callbacks: device model hierarchy

- And finishes its probe by enabling runtime PM.

drivers/spi/spi-dw-pci.c

```
static int dw_spi_pci_probe(struct pci_dev *pdev, const struct pci_device_id *ent)
{
        // ...

        pm_runtime_set_autosuspend_delay(&pdev->dev, 1000);
        pm_runtime_use_autosuspend(&pdev->dev);
        pm_runtime_put_autosuspend(&pdev->dev);
        pm_runtime_allow(&pdev->dev);

        return 0;

err_free_irq_vectors:
        pci_free_irq_vectors(pdev);
        return ret;
}
```

Runtime PM without callbacks: device model hierarchy

- Conclusion?
  - `dw_spi_pci` asks its framework subsystem (SPI) to `pm_runtime_get|put()` automatically on requests.
  - This in turn signals to parent devices, ie the PCI bus controller, when the bus can be safely shut down.
  - Autosuspend will kick in to delay the suspend. This minimises wasteful suspend/resume cycles during bursts of operations. Parent might have its own autosuspend delay.
- No `->runtime_suspend|resume()` implementation required inside `dw_spi_pci`.
- Remember the device model is *recursive*.
  Think GPIO expander over I2C over USB over PCI over platform bus.

Runtime PM without callbacks: device links

- Second reason: trigger implicit behavior reacting to runtime PM refcount.
- That is done through `struct device_link` with `DL_FLAG_PM_RUNTIME`.
  See also `DL_FLAG_RPM_ACTIVE`.
- Example usage:

drivers/net/phy/phy_device.c

```
/**
 * If the external phy used by current mac interface is managed by
 * another mac interface, so we should create a device link between
 * phy dev and mac dev.
 */
if (dev && phydev->mdio.bus->parent && dev->dev.parent != phydev->mdio.bus->parent)
        phydev->devlink = device_link_add(dev->dev.parent, &phydev->mdio.dev,
                                          DL_FLAG_PM_RUNTIME | DL_FLAG_STATELESS);
```

Runtime PM without callbacks: device links

- Subsystem using device links with `DL_FLAG_PM_RUNTIME` are:
  - `drivers/net/phy/` for attaching MAC0 to MAC1 if MAC0 uses MAC1's PHY;
  - `pinctrl` supports linking pin controllers to all their consumers; the flag `link_consumers` is used by `pinctrl-stmfx.c` and `stm32/pinctrl-stm32.c`;
  - `pci` for attaching *quirked* multi-function devices together;
  - `dev->dev_pm_domain`;
  - `pmdomain`, ie *Generic PM Domain*;
  - ~36 drivers calling it directly.

> I was starting to pull this, and then tried to figure out what the heck “genpd” is.
>
> — Linus

Power domains

Power domains: dev_pm_domain

- dev_pm_domain is a field inside struct device.
- It got introduced prior to the Git history.
- Of interest to us are dev->dev_pm_domain.ops.runtime_suspend|resume().

include/linux/pm.h

```
struct dev_pm_domain {
        struct dev_pm_ops ops;
        int (*start)(struct device *dev);
        void (*detach)(struct device *dev, bool power_off);
        int (*activate)(struct device *dev);
        void (*sync)(struct device *dev);
        void (*dismiss)(struct device *dev);
        int (*set_performance_state)(struct device *dev, unsigned int state);
};
```

Power domains: dev_pm_domain

Limitations of dev_pm_domain:

- One domain per device.
- Not straight-forward to implement.
- Not fitting the object model.
  - The domain is not a device and,
  - No device is marked as *providing* the domain.
- It therefore cannot fit in with hardware description (ie devicetree).

Power domains: pmdomain

- Limitations are addressed by pmdomain, a subsystem that is implemented by piggybacking on dev->dev_pm_domain. It used to be called genpd.
- Upstreamed by Rafael J. Wysocki in July 2011.
- Providers must register themselves:

include/linux/pm_domain.h

```
int of_genpd_add_provider_simple(struct device_node *np,
                                 struct generic_pm_domain *genpd);
int of_genpd_add_provider_onecell(struct device_node *np,
                                  struct genpd_onecell_data *data);
```

Power domains: pmdomain

- Consumers aren't expected to use an API (other than runtime suspend/resume); binding is done through devicetree properties & phandles (pinctrl style).

arch/arm64/boot/dts/ti/k3-j7200-mcu-wakeup.dtsi

```
k3_pds: power-controller {
        compatible = "ti,sci-pm-domain";
        #power-domain-cells = <2>;
        bootph-all;
};

wkup_vtm0: temperature-sensor@42040000 {
        compatible = "ti,j7200-vtm";
        reg = <0x00 0x42040000 0x00 0x350>,
              <0x00 0x42050000 0x00 0x350>;
        power-domains = <&k3_pds 154 TI_SCI_PD_EXCLUSIVE>;
        #thermal-sensor-cells = <1>;
        bootph-pre-ram;
};
```

Power domains: pmdomain

```
commit 17f88151ff190b9357f473d7704eee7ae3097d11
Author: Franklin S Cooper Jr <fcooper@ti.com>
Date:   Mon Sep 11 15:11:44 2017 -0500

    i2c: davinci: Add PM Runtime Support

    66AK2G has I2C instances that are not apart of the ALWAYS_ON power domain
    unlike other Keystone 2 SoCs and OMAPL138. Therefore, pm_runtime
    is required to insure the power domain used by the specific I2C instance is
    properly turned on along with its functional clock.

    Signed-off-by: Franklin S Cooper Jr <fcooper@ti.com>
    Acked-by: Sekhar Nori <nsekhar@ti.com>
    Signed-off-by: Wolfram Sang <wsa@the-dreams.de>

 drivers/i2c/busses/i2c-davinci.c | 67 +++++++++++++++++++++++++++++++++-------
 1 file changed, 55 insertions(+), 12 deletions(-)
```

Power domains: pmdomain

- Mainly, a pmdomain is two callbacks; power-on and power-off.
- Power-on is called implicitly when going from zero consumer devices active to one.
- Power-off is called whenever all devices inside the domain are runtime suspended.

include/linux/pm_domain.h

```
struct generic_pm_domain {
        struct device dev;
        struct dev_pm_domain domain;    /* PM domain operations */
        // ...
        int (*power_off)(struct generic_pm_domain *domain);
        int (*power_on)(struct generic_pm_domain *domain);
        // ...
};
```

Power domains: pmdomain

Benefits of pmdomain compared to raw dev->dev_pm_domain:

- Straight forward to implement (example: imx/imx93-pd.c).
- Fitting the object model: provider is an identified device.
- Mark resources inside devicetree (as it should be).
- Still one domain per device **but** we can implement domain hierarchy.
  - Kevin Hilman has a series to add support for it in DT.
  - Issue with flat firmware-provided PM domains, as scmi.
- Zero code inside ->runtime_suspend|resume() for consumers.
  - Transparent to consumer drivers, if they do runtime PM.

Power domains: pmdomain

Some more pmdomain features:

- GENPD_FLAG_PM_CLK / pm_clk_* infrastructure. Attaches clocks to a given power domain, and those are enabled/prepared implicitly.
- Callbacks when devices get attached. Example usage: drivers use the PM_CLK infrastructure to attach device clocks coming from devicetree.

```
int (*attach_dev)(struct generic_pm_domain *domain, struct device *dev);
void (*detach_dev)(struct generic_pm_domain *domain, struct device *dev);
```

- GENPD_FLAG_ACTIVE_WAKEUP for handling power domains on the system-wide suspend wakeup path.
- GENPD_FLAG_RPM_ALWAYS_ON for some (broken?) platforms.

PM QoS

PM QoS: introduction

- PM QoS (*Quality of Service*) is about registering performance expectations.
- Per-device PM QoS is about **requests** to respect a maximum resume latency.
- All requests are aggregated into a single resume latency to respect.

include/linux/pm_qos.h

```
enum dev_pm_qos_req_type {
        DEV_PM_QOS_RESUME_LATENCY = 1, // set max PM resume latency
        DEV_PM_QOS_LATENCY_TOLERANCE,  // value interpreted by driver
        DEV_PM_QOS_MIN_FREQUENCY,      // passed to cpufreq
        DEV_PM_QOS_MAX_FREQUENCY,      // passed to cpufreq
        DEV_PM_QOS_FLAGS,
};

int dev_pm_qos_add_request(struct device *dev, struct dev_pm_qos_request *req,
                           enum dev_pm_qos_req_type type, s32 value);
int dev_pm_qos_update_request(struct dev_pm_qos_request *req, s32 new_value);
int dev_pm_qos_remove_request(struct dev_pm_qos_request *req);
s32 dev_pm_qos_read_value(struct device *dev, enum dev_pm_qos_req_type type);
```

PM QoS: impact on runtime suspend

- If `resume_latency` is zero, never runtime suspend; `rpm_check_suspend_allowed()`.
- Otherwise, we might? runtime suspend if we know the operation is quick enough.
  - That is coming from past experience: how slow were the worst runtime suspend/resume?
  - ? only if device belongs to PM domain
  - ? only if PM domain has a governor
- See `default_suspend_ok()` in `drivers/pmdomain/governor.c`.

```
dev.effective_constraint_ns = dev.resume_latency
for child_dev in dev.children:
    if child_dev.effective_constraint_ns < dev.effective_constraint_ns:
        effective_constraint_ns = child_dev.effective_constraint_ns

# worst past measurements
dev.effective_constraint_ns -= dev.suspend_latency_ns
dev.effective_constraint_ns -= dev.resume_latency_ns

if dev.effective_constraint_ns >= 0:
    dev.suspend()
```

PM QoS: impact on runtime suspend

- suspend_latency_ns is time for dev->runtime_suspend() and pmdomain->stop(dev).
- resume_latency_ns is time for pmdomain->stop(dev) and dev->runtime_suspend().
- **Notice:** this does not include the pmdomain->power_on() and pmdomain->power_off() duration, as it isn't per-device.
  - Depending on the platform, it might be the biggest time sink.

PM QoS: PM domains

- PM domains QoS is done through separate measurements: power_on_latency_ns and power_off_latency_ns.
  - Behavior is similar to suspend_latency_ns and resume_latency_ns.
  - Accounts for PM subdomains.
  - Accounts for children devices known suspend/resume timings.
- Values can come from devicetree; in that case, further ->power_on|off() are not timed.
- See __default_power_down_ok() in drivers/pmdomain/governor.c.

Conclusion

More features (in bulk)

- `cpufreq` configures CPU frequencies; it gets configured through QoS APIs.
- `cpuidle` picks from its list of idle states the deepest acceptable state.
  See Monday talk by Dhruva Gole (TI) and Kevin Hilman (BayLibre).
- PM domains provide OPP: "*The set of discrete tuples consisting of frequency and voltage pairs that the device will support per domain are called Operating Performance Points*". Documentation. Contributed in 2010 by Nishanth Menon (TI).
- PM domains have `set_hwmode_dev()` / `get_hwmode_dev()` for toggling hardware controlled PM. Only one user (Qualcomm GDSC).
- Related ongoing work: The Case for an SoC Power Management Driver — Stephen Boyd, Google (2024).

Thank you!

Questions?

Théo Lebrun

theo.lebrun@bootlin.com

https://bootlin.com/pub/conferences/