Introduction to Generic PM domains

http://www.baylibre.com/pub/conferences/2017/KR/PM_genpd.sozi.html

Kevin Hilman, BayLibre
khilman@baylibre.com

Kernel Recipes 2017, Paris

Who am I?

Live / Work

Seattle, WA

(sometimes from Nice)

BayLibre developer

co-founder

bizdev

ergonomics

Kernel (co)maintainer

Amlogic SoCs (ARM)

TI Davinci SoCs (ARM)

Generic PM domains

Adaptive Voltage scaling (AVS)

arm-soc tree (backup)

Pandering...

Je me débrouille en français

BEAR ATTACK?

Clocks

Regulators

Active PM: Underlying Frameworks

Frequency scaling: clock framework
- clk_get_rate()
- clk_set_rate()

Voltage scaling: regulator framework
- regulator_get_voltage()
- regulator_set_voltage()
- Documentation/power/regulator/consumer.txt

Example... drivers/cpufreq/cpufreq-dt.c

CPUfreq

CPU DVFS using CPUFreq

- Select "best" OPP based on requirements

- pluggable governors for selecting "best" OPP
- performance, powersave, ...
- ondemand: heuristics based on load, tunable
- interactive: ondemand++, tuned for latency

Documentation/cpu-freq/core.txt

... what about device DVFS? ... devfreq

OPPs

Operating Performance Points
OPPs

- tuple of frequency, minimum voltage
- Described in DT

```
 cpu0: cpu@0 {
operating-points = <
/* kHz uV */
300000 1025000
600000 1200000
800000 1313000
1008000 1375000
>;
 }
```

c.f. Documentation/power/opp.txt

Active

Driver model: key concept

struct dev_pm_ops
Exists in struct device_driver, struct bus_type, ...

```
struct dev_pm_ops {
 int (*prepare)(struct device *dev);
 void (*complete)(struct device *dev);
 int (*suspend)(struct device *dev);
 int (*resume)(struct device *dev);
 ...
 int (*suspend_late)(struct device *dev);
 int (*resume_early)(struct device *dev);
 ...
};
```

echo mem > /sys/power/state

Platform specific: struct platform_suspend_ops	Per-device: struct dev_pm_ops
->begin()	
	->prepare() ->suspend()
->prepare()	
	->suspend_late() ->suspend_noirq()
->enter() ->wake()	
	->resume_noirq() ->resume_early()
->finish()	
	->resume() ->complete()
->end()	

Suspend
Resume

Wakeups

Wakeup from Suspend

Subsystem / Driver control:
- device_init_wakeup(dev, bool)
- dev_pm_set_wake_irq()
- dev_pm_clear_wake_irq()

When wakeup occurs (e.g. in ISR):
- pm_wakeup_event()

Enable / disable from user space:
- /sys/devices/..../power/wakeup

Static

Static PM, System PM
- traditional suspend/resume
 CONFIG_PM_SLEEP=y
- system wide, all devices
- initiated by userspace
- any device can prevent suspend

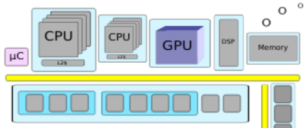
- userspace is "frozen"
 [illegible] Documentation/power/freezing-of-tasks.txt

MUST Read: Documentation/power/devices.txt

Dynamic

Dynamic PM
- based on "activity"
- Active PM
 Save power when doing "something"

- Idle PM
 Save power when doing "nothing"



Complexity is growing...
- more CPUs
- more integrated devices
- more power domains
- micro controllers
- PM firmware, etc.

Kernel is evolving....

Who am I?

Live / Work
 Seattle, WA
 (sometimes from Nice)

BayLibre
developer
 co-founder
 bizdev
 ergonomics

Kernel (co)maintainer

 Amlogic SoCs (ARM)
 TI Davinci SoCs (ARM)
 Generic PM domains
 Adaptive Voltage scaling (AVS)
 arm-soc tree (backup)

Pandering...
 je me débrouille en français

Idle

Idle PM: tickless idle

CONFIG_NOHZ_IDLE=y

- stop periodic tick when idle
- only wakes for next "event" or interrupt

Don't wake up...
 only to press snooze and go back to sleep

NOHZ_IDLE

CPUidle: How deep to sleep?

drivers/cpuidle/governors/menu.c
1) Break even point (based on enter/exit times)
- looks at predictable events (e.g. timers)
- compares against min residency

2) Latency tolerance
- checks QoS (PM_QOS_CPU_DMA_LATENCY)
- compares against min residency

3) Performance Impact:
- black magic "multiplier" based on load
- favor shallower states under heavy load

Limitations:
- not very SMP or multi-cluster aware

Documentation/cpuidle/*.txt

CPUidle

Idle for CPUs

CPU idle states have "depth"
- more power savings
- longer wakeup latency

State Definitions in DT
- legacy: platform-specific driver

State entry
- platform-specific hooks
- based on compatible string

Idle CPUs

OK, but why?
Simplify drivers
 on one SoC, might manage clocks
 on another clocks, regulators, pinctrl

Driver shouldn't have to care
 _get() when busy
 _put() when done

Leave that to the bus_type or domain

Runtime PM: driver callbacks

Use count: 1 --> 0
- ->runtime_suspend()
- prepare for low power state
- ensure wakeups enabled
- save context

Use count: 0 --> 1
- ->runtime_resume()
- restore context
- etc.

Autosuspend -- deferred runtime suspend
- pm_runtime_set_autosuspend_delay()
- pm_runtime_mark_last_busy()
- pm_runtime_put_autosuspend()

Runtime PM API

Tell PM core whether device is in use
"I'm about to use it"
- device: pm_runtime_get(), _sync()
- core: use_count++, pm_runtime_resume()

"I'm done... for now"
- device: pm_runtime_put(), _sync()
- core: use_count-- , pm_runtime_suspend()

Similar to clock framework usage for clock gating
- clk_enable(), clk_disable()

Excellent: Documentation/power/pm_runtime.txt

Runtime PM

Idle for devices: Runtime PM

- per-device idle
- single device at a time
- idleness controlled by driver, based on activity

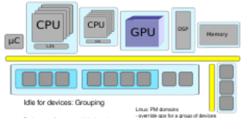
- devices are independent
- one device cannot prevent others from runtime suspending

- does NOT affect user space

```
struct dev_pm_ops {
 ...
 int (*runtime_suspend)(struct device *dev);
 int (*runtime_resume)(struct device *dev);
 int (*runtime_idle)(struct device *dev);
};
```

Bonus: powertop "Device stats"

Idle devices



Idle for devices: Grouping

Devices are often grouped into domains
- power gated as a group
- can be nested
- power gating has latency implications
- external regulator ramp up, etc.

Documentation/power/devices.txt

Linux: PM domains
- override ops for a group of devices
- if PM domain present, PM core uses domain callbacks instead of type/class/bus

```
struct dev_pm_domain {
 struct dev_pm_ops ops;
 ...
};
```

PM domains

genpd in DT
Example genpd:

```
power: power-controller@12340000 {
 compatible = "foo,power-controller";
 reg = <0x12340000 0x1000>;
 #power-domain-cells = <1>;
};
```

Example use by device:

```
leaky-device@12350000 {
 compatible = "foo,i-leak-current";
 reg = <0x12350000 0x1000>;
 power-domains = <&power 0>;
};
```

From: Documentation/devicetree/bindings/power/power-domain.txt

Implement a genpd

Main callbacks:
 ->power_off()
 ->power_on()

Sometimes as simple as:
 register write
 clk_disable()

Optional
 ->attach_dev()
 ->detach_dev()

Or maybe...
 turn off regulator
 send cmd to uC

Describe in DT

what's new

statistics and debug
 see which domains are on
 and how long

IRQ-safe domains

always-on domains

Upstream users:
v4.8: 18
v4.13: 24

under discussion: RFC

Unify idle for CPUs and devices
- use runtime PM for CPUs
- use genpd for clusters of CPUs

Better interaction between static & runtime PM
- allow runtime suspended devices to stay that way during system-wide

Devices could be in genpd or more complex domain
- e.g. ACPI domain or PCI (bus_type)
- more complex than "simple" genpd

Support for performance states (not just idle)

Generic PM Domains (genpd)

Generic implementation of PM domains

Goal: do "stuff" when all devices in a domain become newly idle (or active)

Based on runtime PM
- When all devices in domain are runtime suspended...
 genpd->power_off()
- When first device in domain is runtime resumed...
 genpd->power_on()

Generic PM domains

Governors in genpd

Allow custom decision making before cutting power

Cutting power and re-enabling takes time
 ? will it be off long enough to be to be worth it ?

Before power-off, governor is invoked
 genpd->gov->suspend_ok()

Built-in examples:
 Always-on governor: return false
 Simple QoS governor

PM QoS

Quality of Service: PM QoS

System-wide: e.g PM_QOS_CPU_DMA_LATENCY
- Used by CPUidle to determine depth of idle state

Per-device
- attach QoS constraints with specific devices
- genpd: prevent PM domain power off
- PM_QOS_FLAG_NO_POWER_OFF

- e.g. genpd: per-device wakeup latency
- DEV_PM_QOS_RESUME_LATENCY
- for use by genpd "governors"

Documentation/power/pm_qos_interface.txt

Driver model: key concept

`struct dev_pm_ops`

Exists in struct device_driver, struct bus_type, ...

```
struct dev_pm_ops {
 int (*prepare)(struct device *dev);
 void (*complete)(struct device *dev);
 int (*suspend)(struct device *dev);
 int (*resume)(struct device *dev);
 ...
 int (*suspend_late)(struct device *dev);
 int (*resume_early)(struct device *dev);
 ...
};
```

echo mem > /sys/power/state

Platform specific:

`struct platform_suspend_ops`

->begin()

->prepare()

->enter()

->wake()

->finish()

->end()

Per-device:

`struct dev_pm_ops`

->prepare()

->suspend()

->suspend_late()

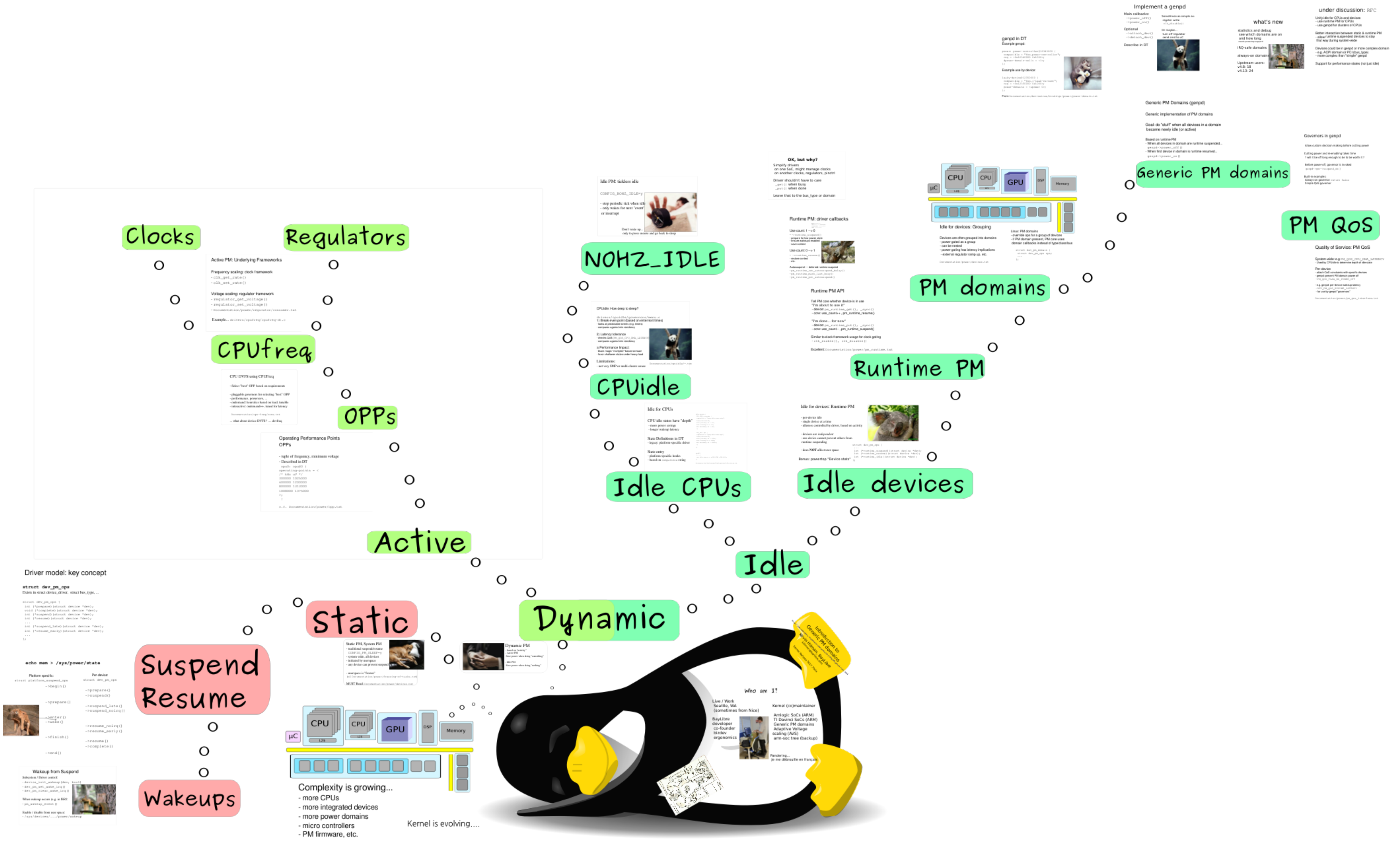
->suspend_noirq()

->resume_noirq()

->resume_early()

->resume()

->complete()



Introduction to Generic PM domains

Kevin Hilman, BayLibre

khilman@baylibre.com

Kernel Recipes 2017, Paris

Who am I?

Live / Work
Seattle, WA
(sometimes from Nice)

BayLibre
developer
co-founder
bizdev
ergonomics

Kernel (co)maintainer

Amlogic SoCs (ARM)
TI Davinci SoCs (ARM)
Generic PM domains
Adaptive Voltage scaling (AVS)
arm-soc tree (backup)

Pandering...
je me débrouille en français

Complexity is growing...
- more CPUs
- more integrated devices
- more power domains
- micro controllers
- PM firmware, etc.

Kernel is evolving....

Static PM, System PM
- traditional suspend/resume
  CONFIG_PM_SLEEP=y
- system wide, all devices
- initiated by userspace
- any device can prevent suspend

- userspace is "frozen"
  (c.f. Documentation/power/freezing-of-tasks.txt)

MUST Read: Documentation/power/devices.txt

Dynamic PM
- based on "activity"
- Active PM
  Save power when doing "something"
- Idle PM
  Save power when doing "nothing"

Driver model: key concept

struct dev_pm_ops

Exists in struct device_driver, struct bus_type, ...

```
struct dev_pm_ops {
 int (*prepare)(struct device *dev);
 void (*complete)(struct device *dev);
 int (*suspend)(struct device *dev);
 int (*resume)(struct device *dev);
 ...
 int (*suspend_late)(struct device *dev);
 int (*resume_early)(struct device *dev);
 ...
};
```

```
echo mem > /sys/power/state
```

Platform specific:
`struct platform_suspend_ops`

```
->begin()

->prepare()

->enter()
->wake()

->finish()

->end()
```

Per-device:
`struct dev_pm_ops`

```
->prepare()
->suspend()

->suspend_late()
->suspend_noirq()

->resume_noirq()
->resume_early()

->resume()
->complete()
```

Wakeup from Suspend

Subsystem / Driver control:
- `device_init_wakeup(dev, bool)`
- `dev_pm_set_wake_irq()`
- `dev_pm_clear_wake_irq()`

When wakeup occurs (e.g. in ISR):
- `pm_wakeup_event()`

Enable / disable from user space:
- `/sys/devices/..../power/wakeup`

Active PM: Underlying Frameworks

Frequency scaling: clock framework
- `clk_get_rate()`
- `clk_set_rate()`

Voltage scaling: regulator framework
- `regulator_get_voltage()`
- `regulator_set_voltage()`
- `Documentation/power/regulator/consumer.txt`

Example... `drivers/cpufreq/cpufreq-dt.c`

CPU DVFS using CPUFreq

- Select "best" OPP based on requirements

- pluggable governors for selecting "best" OPP
- performance, powersave, ...
- ondemand: heuristics based on load, tunable
- interactive: ondemand++, tuned for latency

`Documentation/cpu-freq/core.txt`

... what about device DVFS? ... devfreq

Operating Performance Points
OPPs

- tuple of frequency, minimum voltage
- Described in DT

```
 cpu0: cpu@0 {
operating-points = <
/* kHz uV */
300000 1025000
600000 1200000
800000 1313000
1008000 1375000
>;
 }
```

c.f. Documentation/power/opp.txt

Idle PM: tickless idle

`CONFIG_NOHZ_IDLE=y`

- stop periodic tick when idle
- only wakes for next "event" or interrupt

Don't wake up...
only to press snooze and go back to sleep

CPUidle: How deep to sleep?

`drivers/cpuidle/governors/menu.c`

1) Break even point (based on enter/exit times)
- looks at predictable events (e.g. timers)
- compares against min residency

2) Latency tolerance
- checks QoS (`PM_QOS_CPU_DMA_LATENCY`)
- compares against min residency

3) Performance Impact:
- black magic "multiplier" based on load
- favor shallower states under heavy load

Limitations:
- not very SMP or multi-cluster aware

`Documentation/cpuidle/*.txt`

Idle for CPUs

CPU idle states have "depth"
- more power savings
- longer wakeup latency

State Definitions in DT
- legacy: platform-specific driver

State entry
- platform-specific hooks
- based on `compatible` string

Idle for devices: Runtime PM

- per-device idle
- single device at a time
- idleness controlled by driver, based on activity

- devices are independent
- one device cannot prevent others from runtime suspending

- does NOT affect user space

Bonus: powertop "Device stats"

```
struct dev_pm_ops {
 ...
 int (*runtime_suspend)(struct device *dev);
 int (*runtime_resume)(struct device *dev);
 int (*runtime_idle)(struct device *dev);
};
```

Runtime PM API

Tell PM core whether device is in use
"I'm about to use it"
- device: `pm_runtime_get()`, `_sync()`
- core: use_count++ , pm_runtime_resume()

"I'm done... for now"
- device: `pm_runtime_put()`, `_sync()`
- core: use_count-- , pm_runtime_suspend()

Similar to clock framework usage for clock gating
- `clk_enable()`, `clk_disable()`

Excellent: `Documentation/power/pm_runtime.txt`

Runtime PM: driver callbacks

Use count: 1 --> 0
- `->runtime_suspend()`
- prepare for low power state
- ensure wakeups enabled
- save context

Use count: 0 --> 1
- `->runtime_resume()`
- restore context
- etc.

Autosuspend --- deferred runtime suspend
- `pm_runtime_set_autosuspend_delay()`
- `pm_runtime_mark_last_busy()`
- `pm_runtime_put_autosuspend()`

OK, but why?

Simplify drivers
on one SoC, might manage clocks
on another clocks, regulators, pinctrl

Driver shouldn't have to care
`_get()` when busy
`_put()` when done

Leave that to the bus_type or domain

Idle for devices: Grouping

Devices are often grouped into domains
- power gated as a group
- can be nested
- power gating has latency implications
- external regulator ramp up, etc.

`Documentation/power/devices.txt`

Linux: PM domains
- override ops for a group of devices
- if PM domain present, PM core uses domain callbacks instead of type/class/bus

```
struct dev_pm_domain {
 struct dev_pm_ops ops;
 ...
};
```

Generic PM Domains (genpd)

Generic implementation of PM domains

Goal: do "stuff" when all devices in a domain become newly idle (or active)

Based on runtime PM
- When all devices in domain are runtime suspended...
  `genpd->power_off()`
- When first device in domain is runtime resumed...
  `genpd->power_on()`

genpd in DT

Example genpd:

```
power: power-controller@12340000 {
 compatible = "foo,power-controller";
 reg = <0x12340000 0x1000>;
 #power-domain-cells = <1>;
};
```

Example use by device:

```
leaky-device@12350000 {
 compatible = "foo,i-leak-current";
 reg = <0x12350000 0x1000>;
 power-domains = <&power 0>;
};
```

From: `Documentation/devicetree/bindings/power/power-domain.txt`

Implement a genpd

Main callbacks:
- `->power_off()`
- `->power_on()`

Sometimes as simple as:
register write
`clk_disable()`

Optional:
- `->attach_dev()`
- `->detach_dev()`

Or maybe...
turn off regulator
send cmd to uC

Describe in DT

what's new

statistics and debug
see which domains are on and how long
needs powertop support

IRQ-safe domains

always-on domains

Upstream users:
v4.8: 18
v4.13: 24

under discussion: RFC

Unify idle for CPUs and devices
- use runtime PM for CPUs
- use genpd for clusters of CPUs

Better interaction between static & runtime PM
- allow runtime suspended devices to stay that way during system-wide

Devices could be in genpd or more complex domain
- e.g. ACPI domain or PCI (bus_type)
- more complex than "simple" genpd

Support for performance states (not just idle)

Governors in genpd

Allow custom decision making before cutting power

Cutting power and re-enabling takes time
? will it be off long enough to be to be worth it ?

Before power-off, governor is invoked
`genpd->gov->suspend_ok()`

Built-in examples:
Always-on governor: `return false`
Simple QoS governor

Quality of Service: PM QoS

System-wide: e.g `PM_QOS_CPU_DMA_LATENCY`
- Used by CPUidle to determine depth of idle state

Per-device
- attach QoS constraints with specific devices
- genpd: prevent PM domain power off
- `PM_QOS_FLAG_NO_POWER_OFF`

- e.g. genpd: per-device wakeup latency
- `DEV_PM_QOS_RESUME_LATENCY`
- for use by genpd "governors"

`Documentation/power/pm_qos_interface.txt`

Dynamic

Idle

Idle CPUs

Idle for CPUs

CPU idle states have "depth"
- more power savings
- longer wakeup latency

State Definitions in DT
- legacy: platform-specific driver

State entry
- platform-specific hooks
- based on `compatible` string

```
idle-states {
 CPU_STBY: standby {
 compatible = "qcom,idle-state-stby",
 "arm,idle-state";
 entry-latency-us = <1>;
 exit-latency-us = <1>;
 min-residency-us = <1>;
 };

 CPU_SPC: spc {
 compatible = "qcom,idle-state-spc",
 "arm,idle-state";
 entry-latency-us = <150>;
 exit-latency-us = <200>;
 min-residency-us = <2000>;
 };
};

cpu@0 {
 [...]
 cpu-idle-states = <&CPU_STBY &CPU_SPC>;
};
```

Documentation/devicetree/bindings/arm/idle-states.txt

CPUidle

CPUidle: How deep to sleep?

`drivers/cpuidle/governors/menu.c`

1) Break even point (based on enter/exit times)
- looks at predictable events (e.g. timers)
- compares against min residency

2) Latency tolerance
- checks QoS (`PM_QOS_CPU_DMA_LATENCY`)
- compares against min residency

3) Performance Impact:
- black magic "multiplier" based on load
- favor shallower states under heavy load

`Documentation/cpuidle/*.txt`

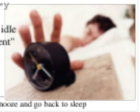
Limitations:
- not very SMP or multi-cluster aware

NOHZ_IDLE

Idle PM: tickless idle

`CONFIG_NOHZ_IDLE=y`

- stop periodic tick when idle
- only wakes for next "event"
 or interrupt



Don't wake up...
only to press snooze and go back to sleep

Idle devices

Idle for devices: Runtime PM

- per-device idle
- single device at a time
- idleness controlled by driver, based on activity

- devices are *independent*
- one device cannot prevent others from runtime suspending

- does **NOT** affect user space

```
struct dev_pm_ops {
 ...
 int (*runtime_suspend)(struct device *dev);
 int (*runtime_resume)(struct device *dev);
 int (*runtime_idle)(struct device *dev);
};
```

Bonus: powertop "Device stats"

Runtime PM

Runtime PM API

Tell PM core whether device is in use

"I'm about to use it"
- device: `pm_runtime_get()`, `_sync()`
- core: use_count++ , pm_runtime_resume()

"I'm done... for now"
- device: `pm_runtime_put()`, `_sync()`
- core: use_count-- , pm_runtime_suspend()

Similar to clock framework usage for clock gating
- `clk_enable()`, `clk_disable()`

Excellent: `Documentation/power/pm_runtime.txt`

Runtime PM: driver callbacks

Use count: 1 --> 0
- `->runtime_suspend()`
- prepare for low-power state
- ensure wakeups enabled
- save context

Use count: 0 --> 1
- `->runtime_resume()`
- restore context
- etc.

Autosuspend --- deferred runtime suspend
- `pm_runtime_set_autosuspend_delay()`
- `pm_runtime_mark_last_busy()`
- `pm_runtime_put_autosuspend()`

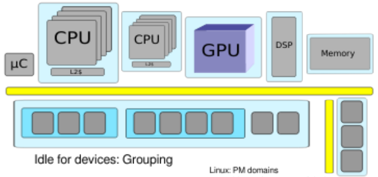
OK, but why?

Simplify drivers
 on one SoC, might manage clocks
 on another clocks, regulators, pinctrl

Driver shouldn't have to care
 `_get()` when busy
 `_put()` when done

Leave that to the bus_type or domain

PM domains



Idle for devices: Grouping

Devices are often grouped into domains
- power gated as a group
- can be nested
- power gating has latency implications
 - external regulator ramp up, etc.

`Documentation/power/devices.txt`

Linux: PM domains
- override ops for a group of devices
- if PM domain present, PM core uses
 domain callbacks instead of type/class/bus

```
struct dev_pm_domain {
 struct dev_pm_ops ops;
 ...
};
```

Generic PM domains

Generic PM Domains (genpd)

Generic implementation of PM domains

Goal: do "stuff" when all devices in a domain
 become newly idle (or active)

Based on runtime PM
- When all devices in domain are runtime suspended...
 `genpd->power_off()`
- When first device in domain is runtime resumed...
 `genpd->power_on()`

Governors in genpd

Allow custom decision making before cutting power

Cutting power and re-enabling takes time
 ? will it be off long enough to be to be worth it ?

Before power-off, governor is invoked
 `genpd->gov->suspend_ok()`

Built-in examples:
 Always-on governor: `return false`
 Simple QoS governor

genpd in DT

Example genpd:

```
power: power-controller@12340000 {
 compatible = "foo,power-controller";
 reg = <0x12340000 0x1000>;
 #power-domain-cells = <1>;
};
```

Example use by device:

```
leaky-device@12350000 {
 compatible = "foo,i-leak-current";
 reg = <0x12350000 0x1000>;
 power-domains = <&power 0>;
};
```

From: `Documentation/devicetree/bindings/power/power-domain.txt`

Implement a genpd

Main callbacks:
 `->power_off()`
 `->power_on()`

Optional
 `->attach_dev()`
 `->detach_dev()`

Describe in DT

Sometimes as simple as:
 regster write
 `clk_disable()`

Or maybe...
 turn off regulator
 send cmd to uC

what's new

statistics and debug
 see which domains are on
 and how long
 needs powertop support!

IRQ-safe domains

always-on domains

Upstream users:
v4.8: 18
v4.13: 24

under discussion: `RFC`

Unify idle for CPUs and devices
- use runtime PM for CPUs
- use genpd for clusters of CPUs

Better interaction between static & runtime PM
- allow runtime suspended devices to stay
 that way during system-wide

Devices could be in genpd or more complex domain
- e.g. ACPI domain or PCI (bus_type)
- more complex than "simple" genpd

Support for performance states (not just idle)

PM QoS

Quality of Service: PM QoS

System-wide: e.g `PM_QOS_CPU_DMA_LATENCY`
- Used by CPUidle to determine depth of idle state

Per-device
- attach QoS constraints with specific devices
- genpd: prevent PM domain power off
 - `PM_QOS_FLAG_NO_POWER_OFF`

- e.g. genpd: per-device wakeup latency
 - `DEV_PM_QOS_RESUME_LATENCY`
 - for use by genpd "governors"

`Documentation/power/pm_qos_interface.txt`

Introduct
Generic P
Kevin
http://www.baylibre.com/

Idle devices

Idle for devices: Runtime PM

- per-device idle
- single device at a time
- idleness controlled by driver, based on activity

- devices are *independent*
- one device cannot prevent others from runtime suspending

- does **NOT** affect user space

Bonus: powertop "Device stats"

```
struct dev_pm_ops {
 ...
 int (*runtime_suspend)(struct device *dev);
 int (*runtime_resume)(struct device *dev);
 int (*runtime_idle)(struct device *dev);
};
```

Runtime PM

Runtime PM API

Tell PM core whether device is in use
 "I'm about to use it"
 - device: `pm_runtime_get(), _sync()`
 - core: use_count++ , pm_runtime_resume()

 "I'm done... for now"
 - device: `pm_runtime_put(), _sync()`
 - core: use_count-- , pm_runtime_suspend()

Similar to clock framework usage for clock gating
 - `clk_enable(), clk_disable()`

Excellent: `Documentation/power/pm_runtime.txt`

Runtime PM: driver callbacks

Use count: 1 --> 0

- `->runtime_suspend()`
- prepare for low-power state
- ensure wakeups enabled
- save context

Use count: 0 --> 1

- `->runtime_resume()`
- restore context
- etc.

Autosuspend --- deferred runtime suspend
- `pm_runtime_set_autosuspend_delay()`
- `pm_runtime_mark_last_busy()`
- `pm_runtime_put_autosuspend()`

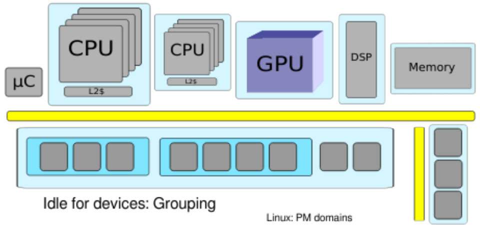
OK, but why?

Simplify drivers
 on one SoC, might manage clocks
 on another clocks, regulators, pinctrl

Driver shouldn't have to care
 `_get()` when busy
 `_put()` when done

Leave that to the bus_type or domain

PM domains



Idle for devices: Grouping

Devices are often grouped into domains
- power gated as a group
- can be nested
- power gating has latency implications
 - external regulator ramp up, etc.

`Documentation/power/devices.txt`

Linux: PM domains
- override ops for a group of devices
- if PM domain present, PM core uses domain callbacks instead of type/class/bus

```
struct dev_pm_domain {
 struct dev_pm_ops ops;
 ...
};
```

Generic PM domains

Generic PM Domains (genpd)

Generic implementation of PM domains

Goal: do "stuff" when all devices in a domain become newly idle (or active)

Based on runtime PM
- When all devices in domain are runtime suspended...
 `genpd->power_off()`
- When first device in domain is runtime resumed...
 `genpd->power_on()`

Governors in genpd

Allow custom decision making before cutting power

Cutting power and re-enabling takes time
 ? will it be off long enough to be to be worth it ?

Before power-off, governor is invoked
 `genpd->gov->suspend_ok()`

Built-in examples:
 Always-on governor: `return false`
 Simple QoS governor

genpd in DT

Example genpd:

```
power: power-controller@12340000 {
 compatible = "foo,power-controller";
 reg = <0x12340000 0x1000>;
 #power-domain-cells = <1>;
};
```

Example use by device:

```
leaky-device@12350000 {
 compatible = "foo,i-leak-current";
 reg = <0x12350000 0x1000>;
 power-domains = <&power 0>;
};
```

From: `Documentation/devicetree/bindings/power/power-domain.txt`

Implement a genpd

Main callbacks:
 `->power_off()`
 `->power_on()`

Optional
 `->attach_dev()`
 `->detach_dev()`

Describe in DT

Sometimes as simple as:
 regster write
 `clk_disable()`

Or maybe...
 turn off regulator
 send cmd to uC

what's new

statistics and debug
 see which domains are on
 and how long
 needs powertop support!

IRQ-safe domains

always-on domains

Upstream users:
v4.8: 18
v4.13: 24

under discussion: `RFC`

Unify idle for CPUs and devices
- use runtime PM for CPUs
- use genpd for clusters of CPUs

Better interaction between static & runtime PM
- allow runtime suspended devices to stay
 that way during system-wide

Devices could be in genpd or more complex domain
- e.g. ACPI domain or PCI (bus_type)
- more complex than "simple" genpd

Support for performance states (not just idle)

PM QoS

Quality of Service: PM QoS

System-wide: e.g `PM_QOS_CPU_DMA_LATENCY`
- Used by CPUidle to determine depth of idle state

Per-device
- attach QoS constraints with specific devices
- genpd: prevent PM domain power off
 - `PM_QOS_FLAG_NO_POWER_OFF`

- e.g. genpd: per-device wakeup latency
 - `DEV_PM_QOS_RESUME_LATENCY`
 - for use by genpd "governors"

`Documentation/power/pm_qos_interface.txt`

Idle for devices: Runtime PM

- per-device idle
- single device at a time
- idleness controlled by driver, based on activity

- devices are *independent*
- one device cannot prevent others from runtime suspending

- does **NOT** affect user space

```
struct dev_pm_ops {
 ...
 int (*runtime_suspend)(struct device *dev);
 int (*runtime_resume)(struct device *dev);
 int (*runtime_idle)(struct device *dev);
};
```

Bonus: powertop "Device stats"

Runtime PM API

Tell PM core whether device is in use

"I'm about to use it"

- device: `pm_runtime_get(), _sync()`
- core: use_count++ , pm_runtime_resume()

"I'm done... for now"

- device: `pm_runtime_put(), _sync()`
- core: use_count-- , pm_runtime_suspend()

Similar to clock framework usage for clock gating

- `clk_enable(), clk_disable()`

Excellent: `Documentation/power/pm_runtime.txt`

Runtime PM: driver callbacks

device: `pm_runtime_put()`

PM core: `usage == 0 ?`, runtime suspend

bus_type (domain): `->runtime_suspend()`

device: `->runtime_suspend()`

Use count: 1 --> 0

- `->runtime_suspend()`
- prepare for low-power state
- ensure wakeups enabled
- save context

Use count: 0 --> 1

- `->runtime_resume()`
- restore context
- etc.

Autosuspend --- deferred runtime suspend

- `pm_runtime_set_autosuspend_delay()`
- `pm_runtime_mark_last_busy()`
- `pm_runtime_put_autosuspend()`

device: `pm_runtime_put()`

PM core: `usage == 0 ?`, runtime suspend

bus_type (domain): `->runtime_suspend()`

device: `->runtime_suspend()`

1 --> 0

Runtime PM: driver callbacks

device: `pm_runtime_put()`

PM core: `usage == 0 ?`, runtime suspend

bus_type (domain): `->runtime_suspend()`

device: `->runtime_suspend()`

Use count: 1 --> 0

- `->runtime_suspend()`
- prepare for low-power state
- ensure wakeups enabled
- save context

Use count: 0 --> 1

- `->runtime_resume()`
- restore context
- etc.

Autosuspend --- deferred runtime suspend

- `pm_runtime_set_autosuspend_delay()`
- `pm_runtime_mark_last_busy()`
- `pm_runtime_put_autosuspend()`

Idle devices

Idle for devices: Runtime PM

- per-device idle
- single device at a time
- idleness controlled by driver, based on activity

- devices are *independent*
- one device cannot prevent others from
 runtime suspending

- does **NOT** affect user space

Bonus: powertop "Device stats"

```
struct dev_pm_ops {
 ...
 int (*runtime_suspend)(struct device *dev);
 int (*runtime_resume)(struct device *dev);
 int (*runtime_idle)(struct device *dev);
};
```

Runtime PM

Runtime PM API

Tell PM core whether device is in use
 "I'm about to use it"
 - device: `pm_runtime_get(), _sync()`
 - core: use_count++ , pm_runtime_resume()

 "I'm done... for now"
 - device: `pm_runtime_put(), _sync()`
 - core: use_count-- , pm_runtime_suspend()

Similar to clock framework usage for clock gating
 - `clk_enable(), clk_disable()`

Excellent: `Documentation/power/pm_runtime.txt`

Runtime PM: driver callbacks

Use count: 1 --> 0

- `->runtime_suspend()`
- prepare for low-power state
- ensure wakeups enabled
- save context

Use count: 0 --> 1

- `->runtime_resume()`
- restore context
- etc.

Autosuspend --- deferred runtime suspend
- `pm_runtime_set_autosuspend_delay()`
- `pm_runtime_mark_last_busy()`
- `pm_runtime_put_autosuspend()`

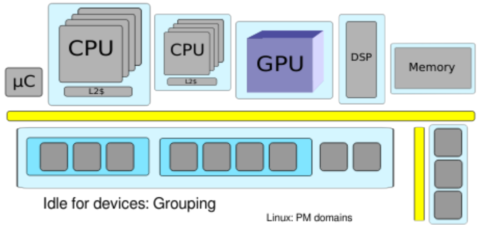
OK, but why?

Simplify drivers
 on one SoC, might manage clocks
 on another clocks, regulators, pinctrl

Driver shouldn't have to care
 `_get()` when busy
 `_put()` when done

Leave that to the bus_type or domain

PM domains



Idle for devices: Grouping

Devices are often grouped into domains
- power gated as a group
- can be nested
- power gating has latency implications
 - external regulator ramp up, etc.

`Documentation/power/devices.txt`

Linux: PM domains
- override ops for a group of devices
- if PM domain present, PM core uses
 domain callbacks instead of type/class/bus

```
struct dev_pm_domain {
 struct dev_pm_ops ops;
 ...
};
```

Generic PM domains

Generic PM Domains (genpd)

Generic implementation of PM domains

Goal: do "stuff" when all devices in a domain
 become newly idle (or active)

Based on runtime PM
- When all devices in domain are runtime suspended...
 `genpd->power_off()`
- When first device in domain is runtime resumed...
 `genpd->power_on()`

Governors in genpd

Allow custom decision making before cutting power

Cutting power and re-enabling takes time
 ? will it be off long enough to be to be worth it ?

Before power-off, governor is invoked
 `genpd->gov->suspend_ok()`

Built-in examples:
 Always-on governor: `return false`
 Simple QoS governor

genpd in DT

Example genpd:

```
power: power-controller@12340000 {
 compatible = "foo,power-controller";
 reg = <0x12340000 0x1000>;
 #power-domain-cells = <1>;
};
```

Example use by device:

```
leaky-device@12350000 {
 compatible = "foo,i-leak-current";
 reg = <0x12350000 0x1000>;
 power-domains = <&power 0>;
};
```

From: `Documentation/devicetree/bindings/power/power-domain.txt`

Implement a genpd

Main callbacks:
 `->power_off()`
 `->power_on()`

Optional
 `->attach_dev()`
 `->detach_dev()`

Describe in DT

Sometimes as simple as:
 regster write
 `clk_disable()`

Or maybe...
 turn off regulator
 send cmd to uC

what's new

statistics and debug
 see which domains are on
 and how long
 needs powertop support!

IRQ-safe domains

always-on domains

Upstream users:
v4.8: 18
v4.13: 24

under discussion: `RFC`

Unify idle for CPUs and devices
- use runtime PM for CPUs
- use genpd for clusters of CPUs

Better interaction between static & runtime PM
- allow runtime suspended devices to stay
 that way during system-wide

Devices could be in genpd or more complex domain
- e.g. ACPI domain or PCI (bus_type)
- more complex than "simple" genpd

Support for performance states (not just idle)

PM QoS

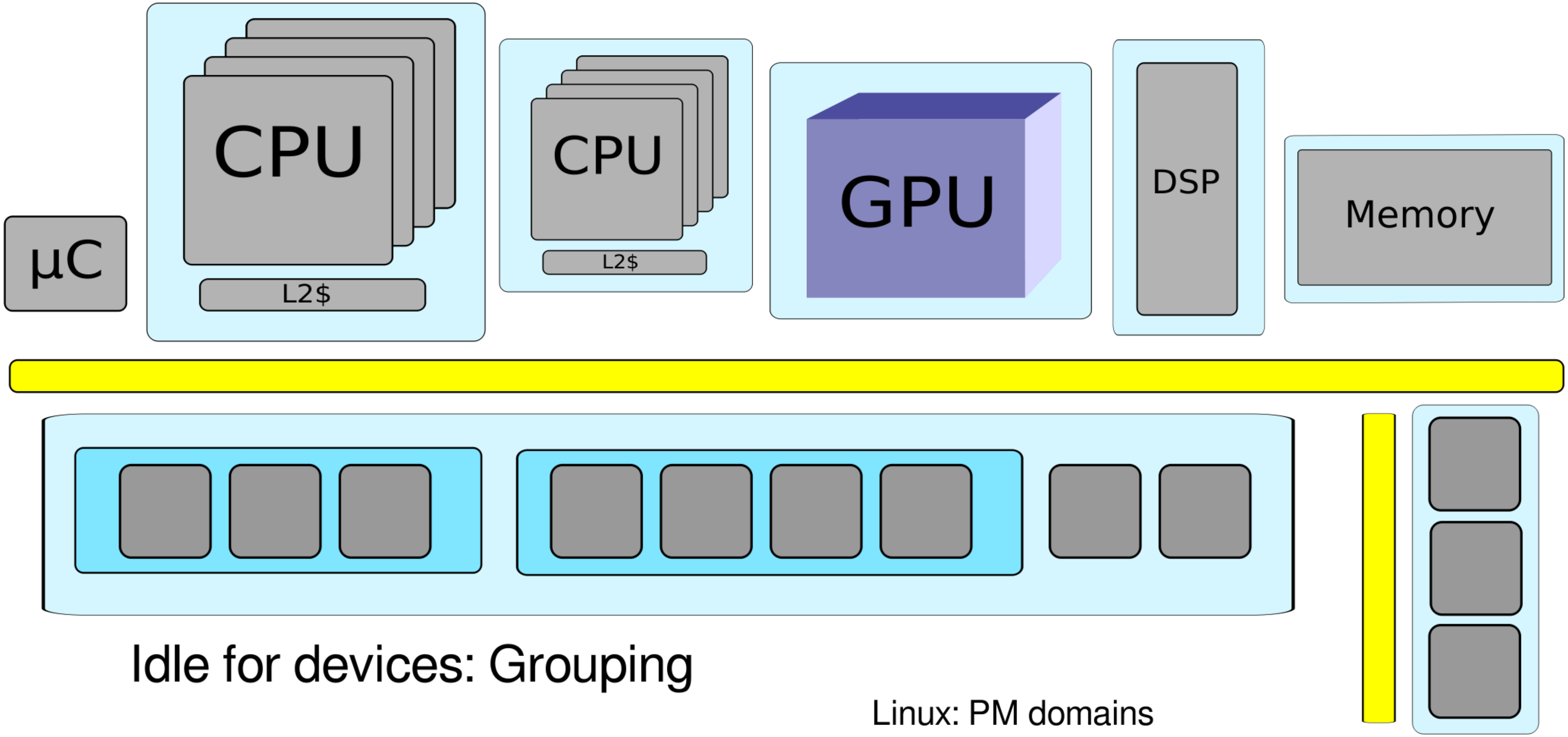
Quality of Service: PM QoS

System-wide: e.g `PM_QOS_CPU_DMA_LATENCY`
- Used by CPUidle to determine depth of idle state

Per-device
- attach QoS constraints with specific devices
- genpd: prevent PM domain power off
 - `PM_QOS_FLAG_NO_POWER_OFF`

- e.g. genpd: per-device wakeup latency
 - `DEV_PM_QOS_RESUME_LATENCY`
 - for use by genpd "governors"

`Documentation/power/pm_qos_interface.txt`



Idle for devices: Grouping

Devices are often grouped into domains
- power gated as a group
- can be nested
- power gating has latency implications
 - external regulator ramp up, etc.

```
Documentation/power/devices.txt
```

Linux: PM domains
- override ops for a group of devices
- if PM domain present, PM core uses
 domain callbacks instead of type/class/bus

```
struct dev_pm_domain {
 struct dev_pm_ops ops;
 ...
};
```

Generic PM Domains (genpd)

Generic implementation of PM domains

Goal: do "stuff" when all devices in a domain become newly idle (or active)

Based on runtime PM

- When all devices in domain are runtime suspended...

  `genpd->power_off()`

- When first device in domain is runtime resumed...

  `genpd->power_on()`

Implement a genpd

Main callbacks:

```
->power_off()
->power_on()
```

Optional

```
->attach_dev()
->detach_dev()
```

Describe in DT

Sometimes as simple as:

regster write

```
clk_disable()
```

Or maybe...

turn off regulator

send cmd to uC

genpd in DT

Example genpd:

```
power: power-controller@12340000 {
 compatible = "foo,power-controller";
 reg = <0x12340000 0x1000>;
 #power-domain-cells = <1>;
};
```

Example use by device:

```
leaky-device@12350000 {
 compatible = "foo,i-leak-current";
 reg = <0x12350000 0x1000>;
 power-domains = <&power 0>;
};
```

From: `Documentation/devicetree/bindings/power/power-domain.txt`

Governors in genpd

Allow custom decision making before cutting power

Cutting power and re-enabling takes time

? will it be off long enough to be to be worth it ?

Before power-off, governor is invoked

`genpd->gov->suspend_ok()`

Built-in examples:

Always-on governor: `return false`

Simple QoS governor

Quality of Service: PM QoS

System-wide: e.g `PM_QOS_CPU_DMA_LATENCY`

- Used by CPUidle to determine depth of idle state

Per-device

- attach QoS constraints with specific devices
- genpd: prevent PM domain power off
  - `PM_QOS_FLAG_NO_POWER_OFF`

- e.g. genpd: per-device wakeup latency
  - `DEV_PM_QOS_RESUME_LATENCY`
  - for use by genpd "governors"

`Documentation/power/pm_qos_interface.txt`

what's new

statistics and debug

see which domains are on and how long

needs powertop support!

IRQ-safe domains

always-on domains

Upstream users:

v4.8: 18

v4.13: 24

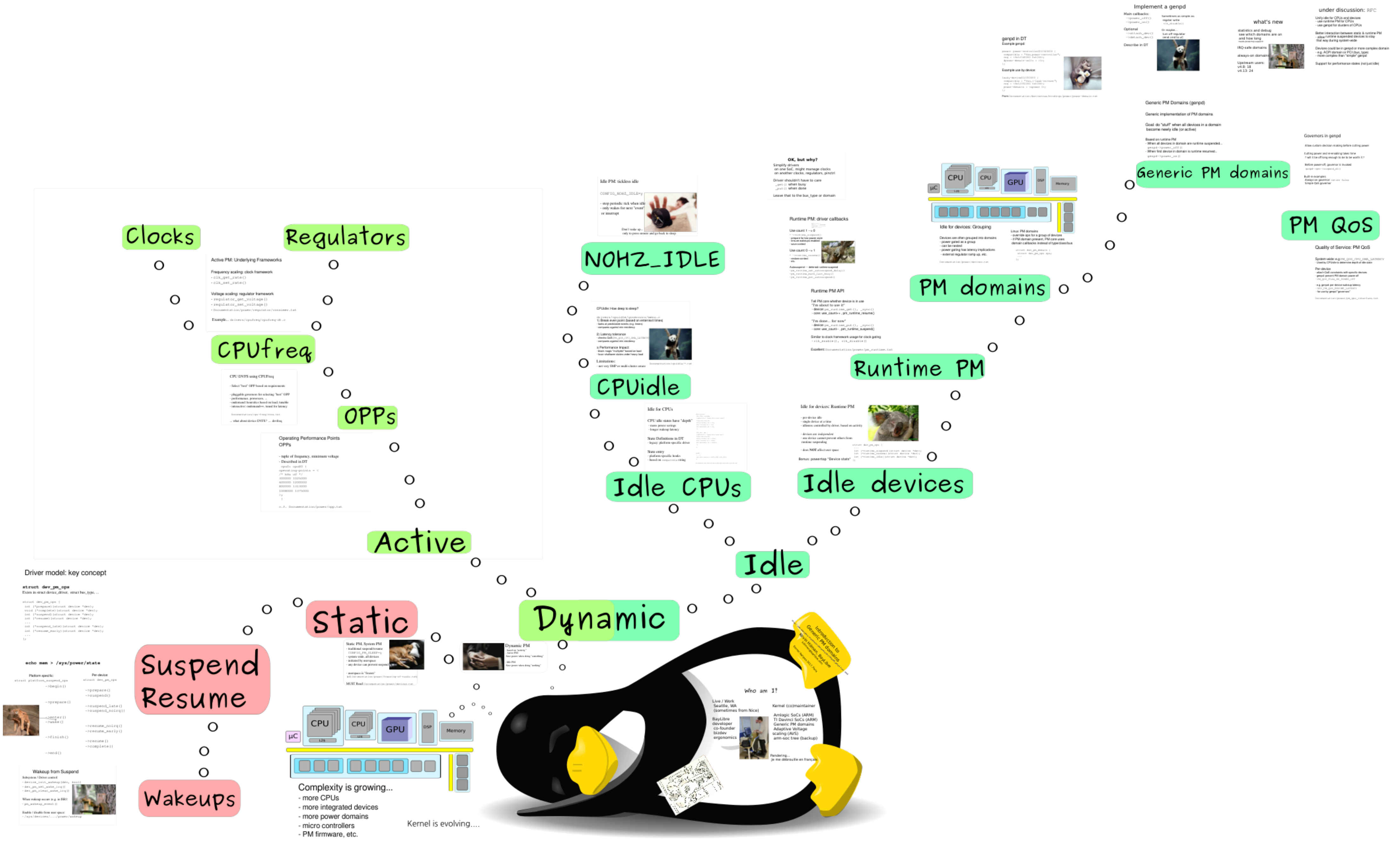
under discussion: RFC

Unify idle for CPUs and devices
- use runtime PM for CPUs
- use genpd for clusters of CPUs

Better interaction between static & runtime PM
- allow runtime suspended devices to stay
 that way during system-wide

Devices could be in genpd or more complex domain
- e.g. ACPI domain or PCI (bus_type)
- more complex than "simple" genpd

Support for performance states (not just idle)



Active PM: Underlying Frameworks

Frequency scaling: clock framework
- clk_get_rate()
- clk_set_rate()

Voltage scaling: regulator framework
- regulator_get_voltage()
- regulator_set_voltage()
- Documentation/power/regulator/consumer.txt

Example... drivers/cpufreq/cpufreq-dt.c

CPU DVFS using CPUfreq

Operating Performance Points
OPPs

Idle PM: tickless idle

CPUidle: How deep to sleep?

Idle for CPUs

Idle for devices: Runtime PM

OK, but why?
Simplify drivers
on one SoC, might manage clocks
on another clocks, regulators, pinctrl

Driver shouldn't have to care
_get() when busy
_put() when done

Leave that to the bus_type or domain

Runtime PM: driver callbacks

Use count: 1 --> 0
- ->runtime_suspend()
- prepare for low power state
- ensure wakeups enabled
- save context

Use count: 0 --> 1
- ->runtime_resume()
- restore context
- etc.

Autosuspend --- deferred runtime suspend
- pm_runtime_set_autosuspend_delay()
- pm_runtime_mark_last_busy()
- pm_runtime_put_autosuspend()

Runtime PM API

Tell PM core whether device is in use
"I'm about to use it"
- device: pm_runtime_get(), _sync()
- core: use_count++, pm_runtime_resume()

"I'm done... for now"
- device: pm_runtime_put(), _sync()
- core: use_count--, pm_runtime_suspend()

Similar to clock framework usage for clock gating
- clk_enable(), clk_disable()

Excellent: Documentation/power/pm_runtime.txt

Idle for devices: Grouping

genpd in DT

Implement a genpd

Main callbacks:
->power_off()
->power_on()

Optional
->attach_dev()
->detach_dev()

Describe in DT

Sometimes as simple as:
register write
clk_disable()

Or maybe...
turn off regulator
send cmd to uC

what's new

statistics and debug
see which domains are on
and how long

IRQ-safe domains

always-on domains

Upstream users:
v4.8: 18
v4.13: 24

under discussion: RFC

Unify idle for CPUs and devices
- use runtime PM for CPUs
- use genpd for clusters of CPUs

Better interaction between static & runtime PM
- allow runtime suspended devices to stay
that way during system-wide

Devices could be in genpd or more complex domain
- e.g. ACPI domain or PCI (bus_type)
- more complex than "simple" genpd

Support for performance states (not just idle)

Generic PM Domains (genpd)

Generic implementation of PM domains

Goal: do "stuff" when all devices in a domain
become newly idle (or active)

Based on runtime PM
- When all devices in domain are runtime suspended...
genpd->power_off()
- When first device in domain is runtime resumed...
genpd->power_on()

Governors in genpd

Quality of Service: PM QoS

Static PM: System PM

Dynamic PM

Driver model: key concept

echo mem > /sys/power/state

Wakeup from Suspend

Complexity is growing...
- more CPUs
- more integrated devices
- more power domains
- micro controllers
- PM firmware, etc.

Kernel is evolving....

Introduction to
Generic PM domains
Kevin Hilman, BayLibre

Who am I?

Live / Work
Seattle, WA
(sometimes from Nice)

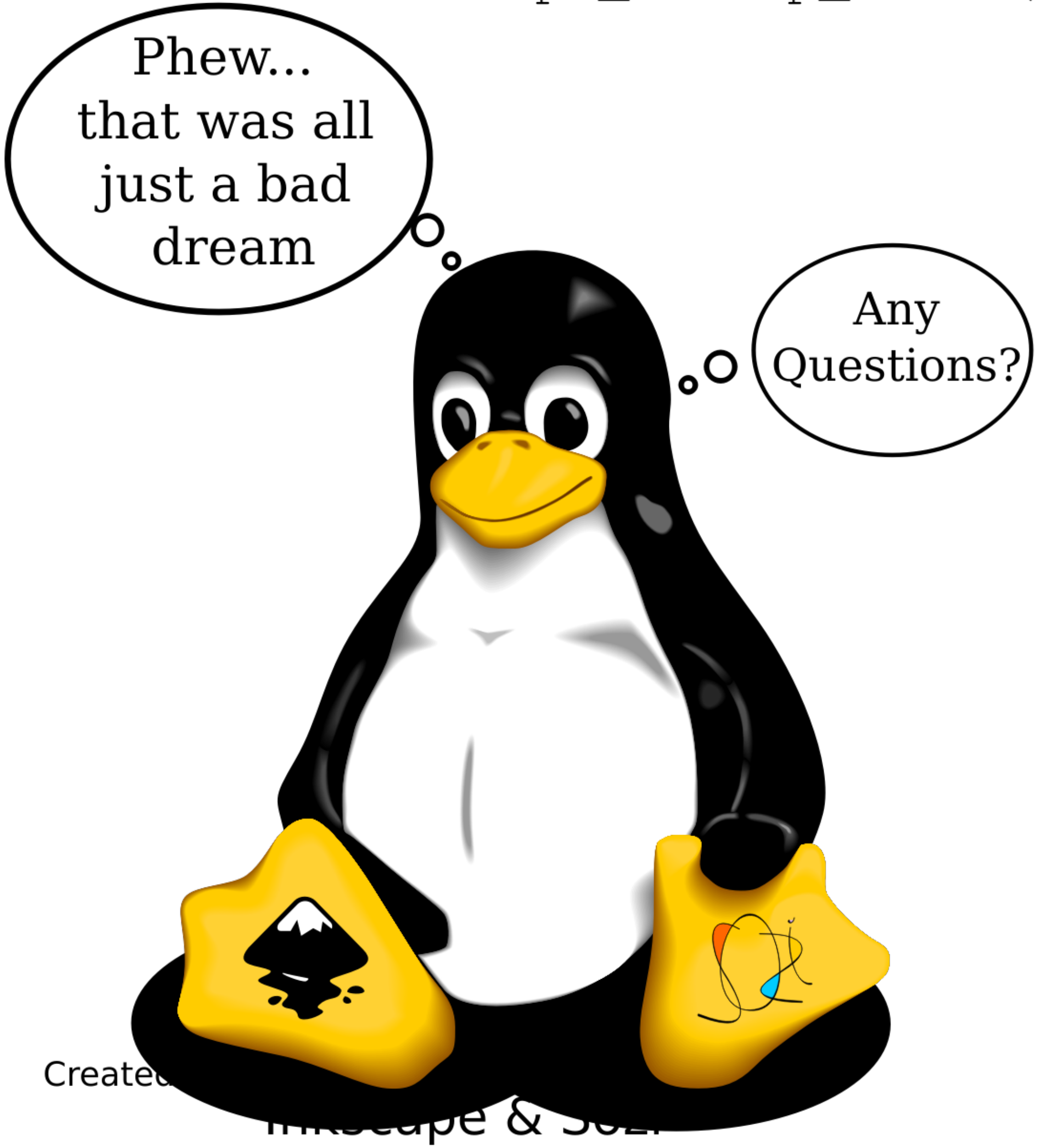
BayLibre
developer
co-founder
bizdev
ergonomics

Kernel (co)maintainer

Amlogic SoCs (ARM)
TI Davinci SoCs (ARM)
Generic PM domains
Adaptive Voltage
scaling (AVS)
arm-soc tree (backup)

Pandering...
je me débrouille en français

pm_wakeup_event()



Created

Inkscape & Sozi

http://www.baylibre.com/pub/conferences/kr2017/