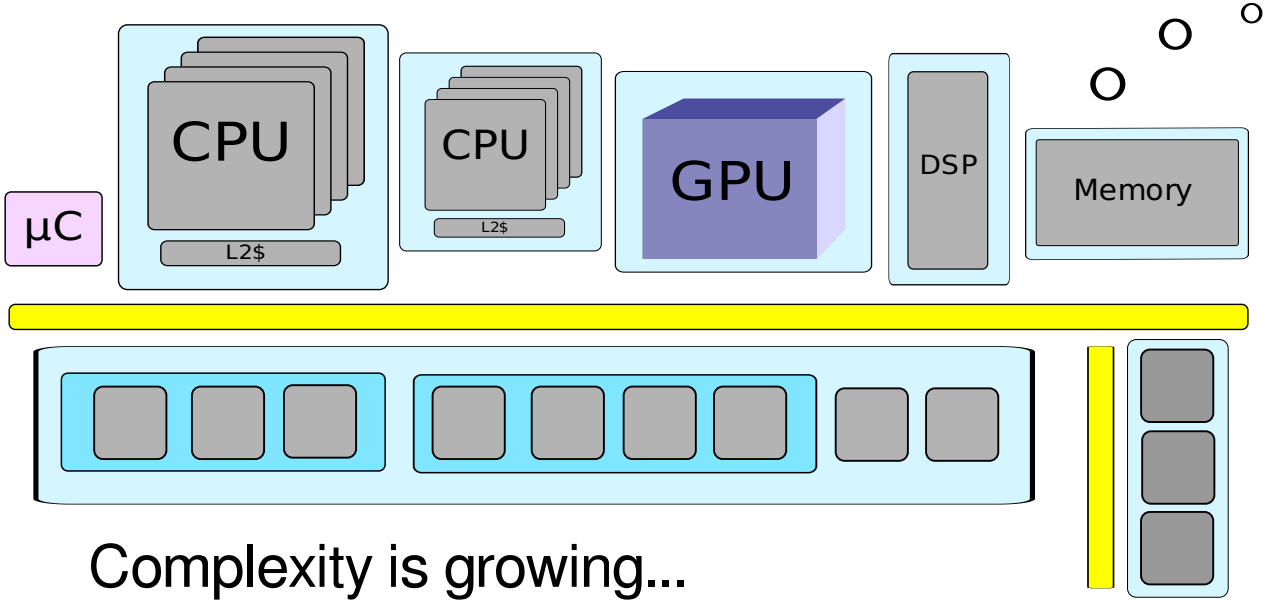
NOTE: For the full experience, use the original SVG at:
http://people.linaro.org/~kevin.hilman/conf/elc2015/Intro_Kernel_PM.svg

Introduction to Kernel Power Management

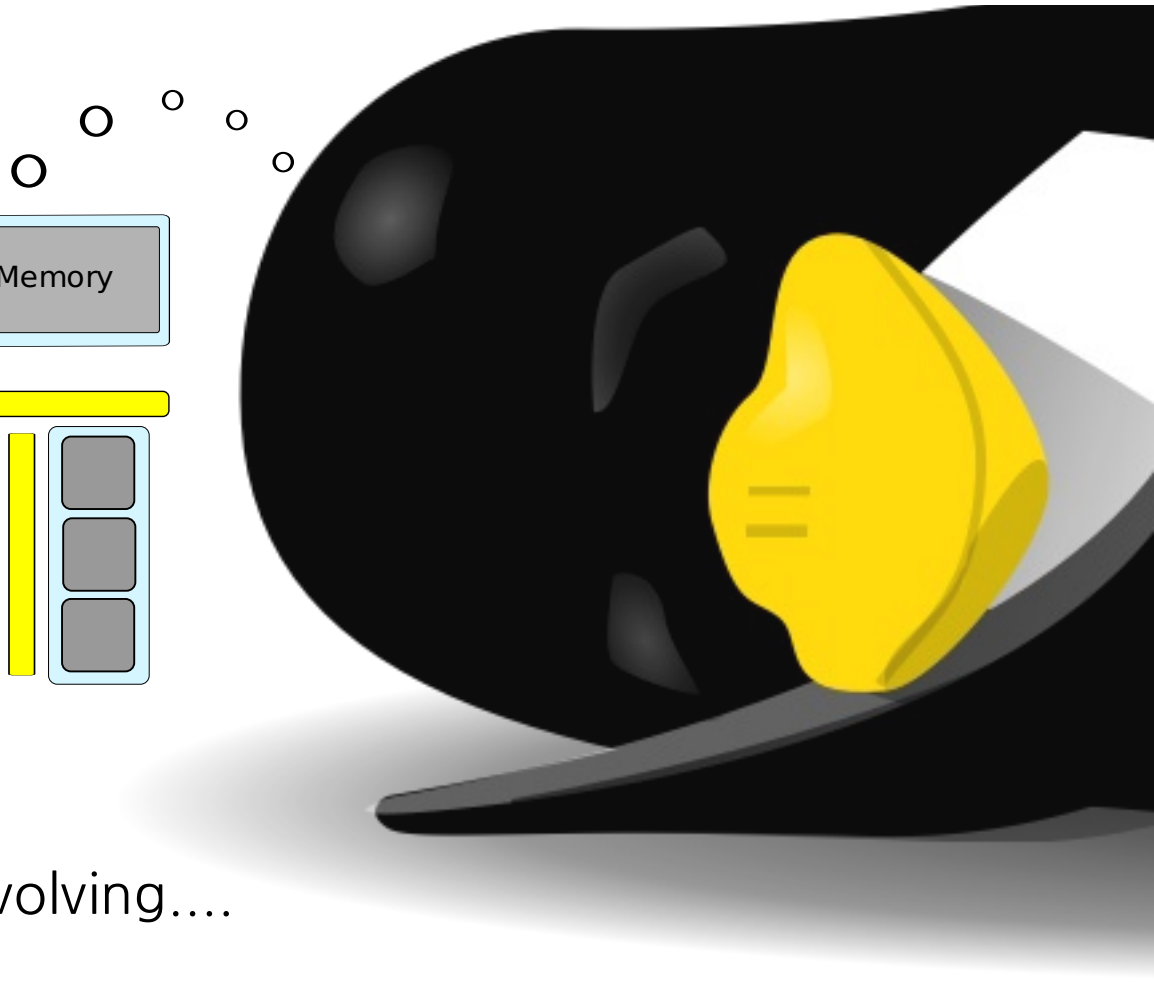
Kevin Hilman, Linaro
khilman@kernel.org

ELC 2015, San Jose CA

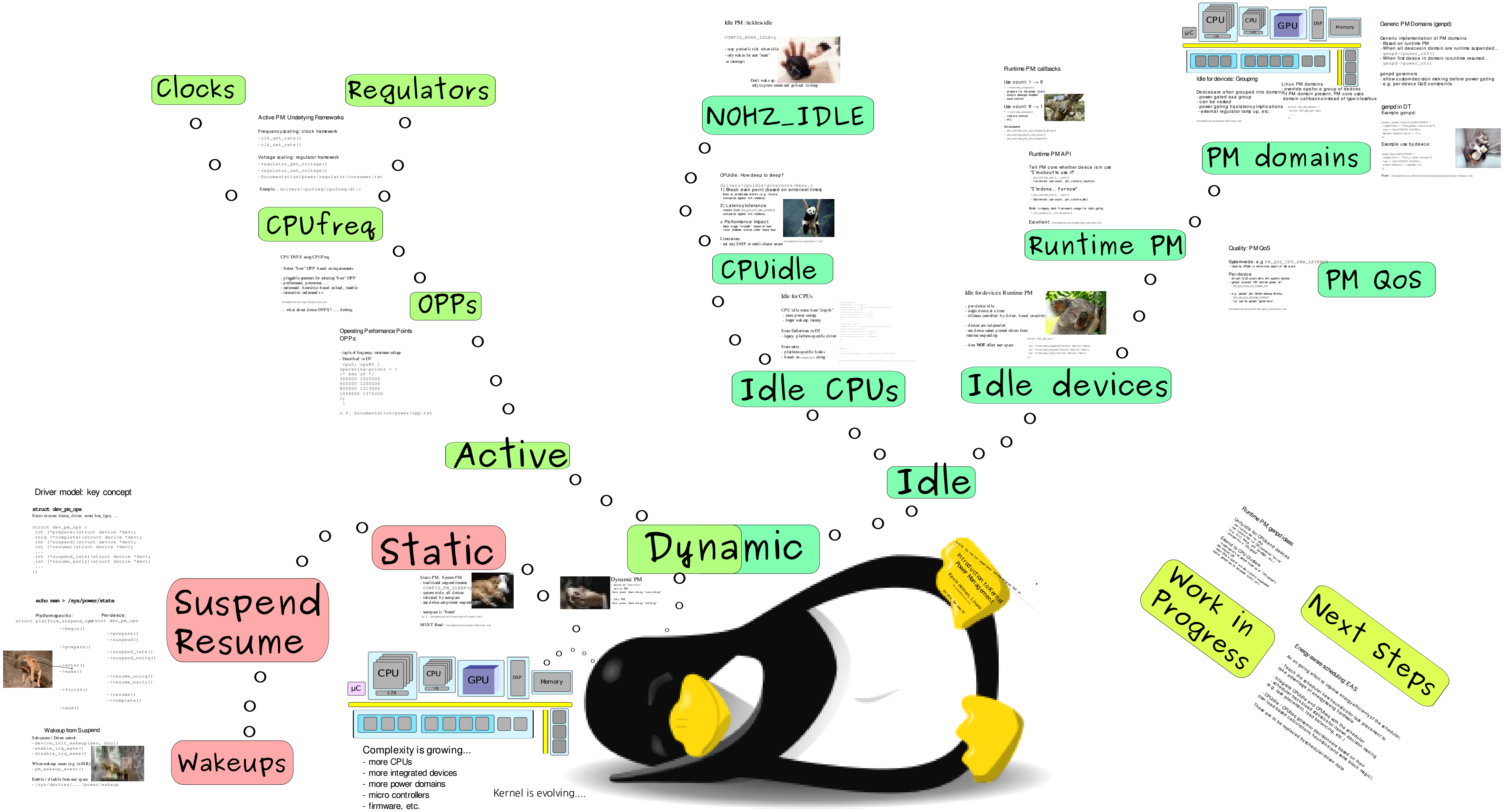


Complexity is growing...

- more CPUs
- more integrated devices
- more power domains
- micro controllers
- firmware, etc.



Kernel is evolving....


Clocks
Regulators
Active PM: Underlying Frameworks
CPUfreq
OPPs
Operating Performance Points
Active
NOHZ_IDLE
Idle PM: tickless idle
CPUidle
Idle CPUs
Idle devices
Runtime PM
Runtime PM callbacks
Runtime PM API
PM domains
Generic PM Domains (genpd)
PM QoS
Quality: PM QoS
Idle
Dynamic
Static
Driver model: key concept
Suspend Resume
Wakeups
Wakeup from Suspend
CPU
GPU
DSP
Memory
µC
Complexity is growing...
- more CPUs
- more integrated devices
- more power domains
- micro controllers
- firmware, etc.
Kernel is evolving....
Work in Progress
Next Steps
Energy-aware scheduling EAS

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/freezing-of-tasks.txt`)

MUST Read: `Documentation/power/devices.txt`

Driver model: key concept

`struct dev_pm_ops`

Exists in struct device_driver, struct bus_type, ...

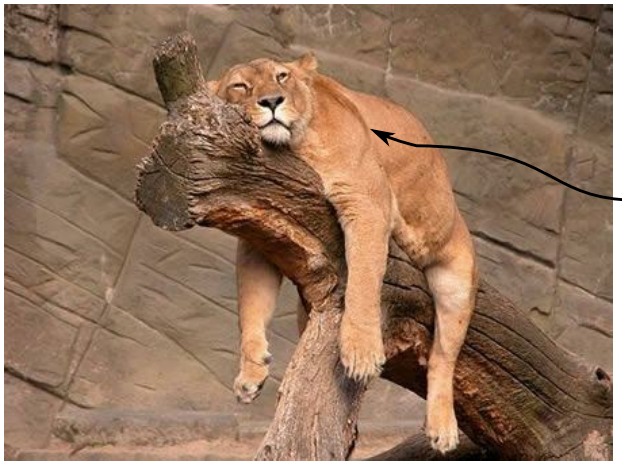
```
struct dev_pm_ops {
 int (*prepare)(struct device *dev);
 void (*complete)(struct device *dev);
 int (*suspend)(struct device *dev);
 int (*resume)(struct device *dev);
 ...
 int (*suspend_late)(struct device *dev);
 int (*resume_early)(struct device *dev);
 ...
};
```

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()
```

Wakeup from Suspend

Subsystem / Driver control:

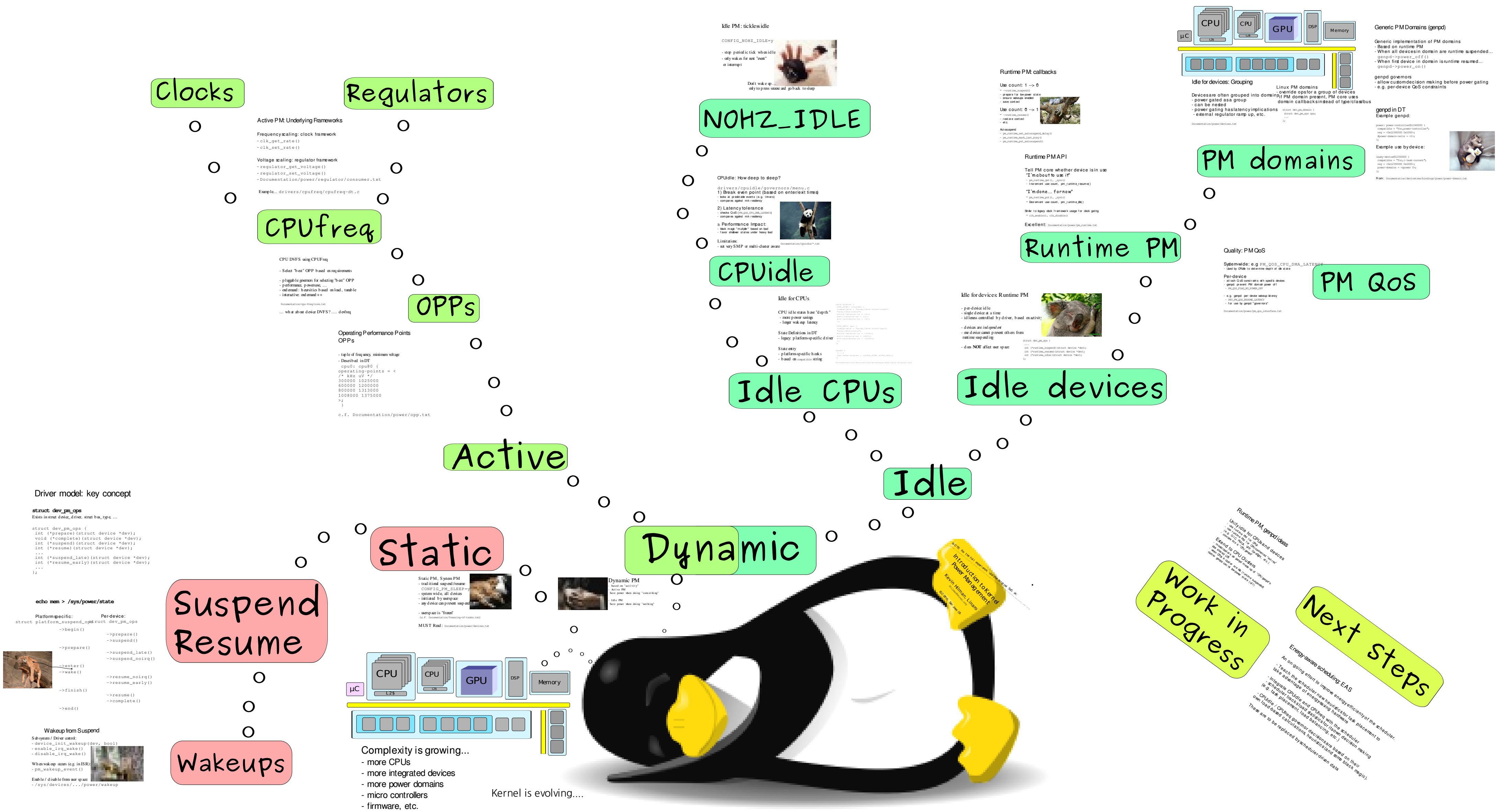
- `device_init_wakeup(dev, bool)`
- `enable_irq_wake()`
- `disable_irq_wake()`

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

- `pm_wakeup_event()`

Enable / disable from user space:

- `/sys/devices/.../power/wakeup`


Clocks
Regulators
Active PM: Underlying Frameworks
CPUfreq
OPPs
Operating Performance Points
Active
Idle PM: tickless idle
NOHZ_IDLE
CPUidle
Idle CPUs
Runtime PM callbacks
Runtime PM API
Runtime PM
Idle for devices: Runtime PM
Idle devices
Idle for devices: Grouping
PM domains
Generic PM Domains (genpd)
Quality: PM QoS
PM QoS
Idle
Dynamic
Static
Driver model: key concept
Suspend
Resume
Wakeups
Wakeup from Suspend
CPU
GPU
DSP
Memory
µC
Complexity is growing...
- more CPUs
- more integrated devices
- more power domains
- micro controllers
- firmware, etc.
Kernel is evolving....
Work in Progress
Next Steps

Dynamic PM

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

- Idle PM
  Save power when doing "nothing"

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: heursitics based on load, tunable
- interactive: ondemand++

`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

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

CPU DVFS using CPUFreq

- Select "best" OPP based on requirements

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

`Documentation/cpu-freq/core.txt`

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

Active PM: Underlying Frameworks

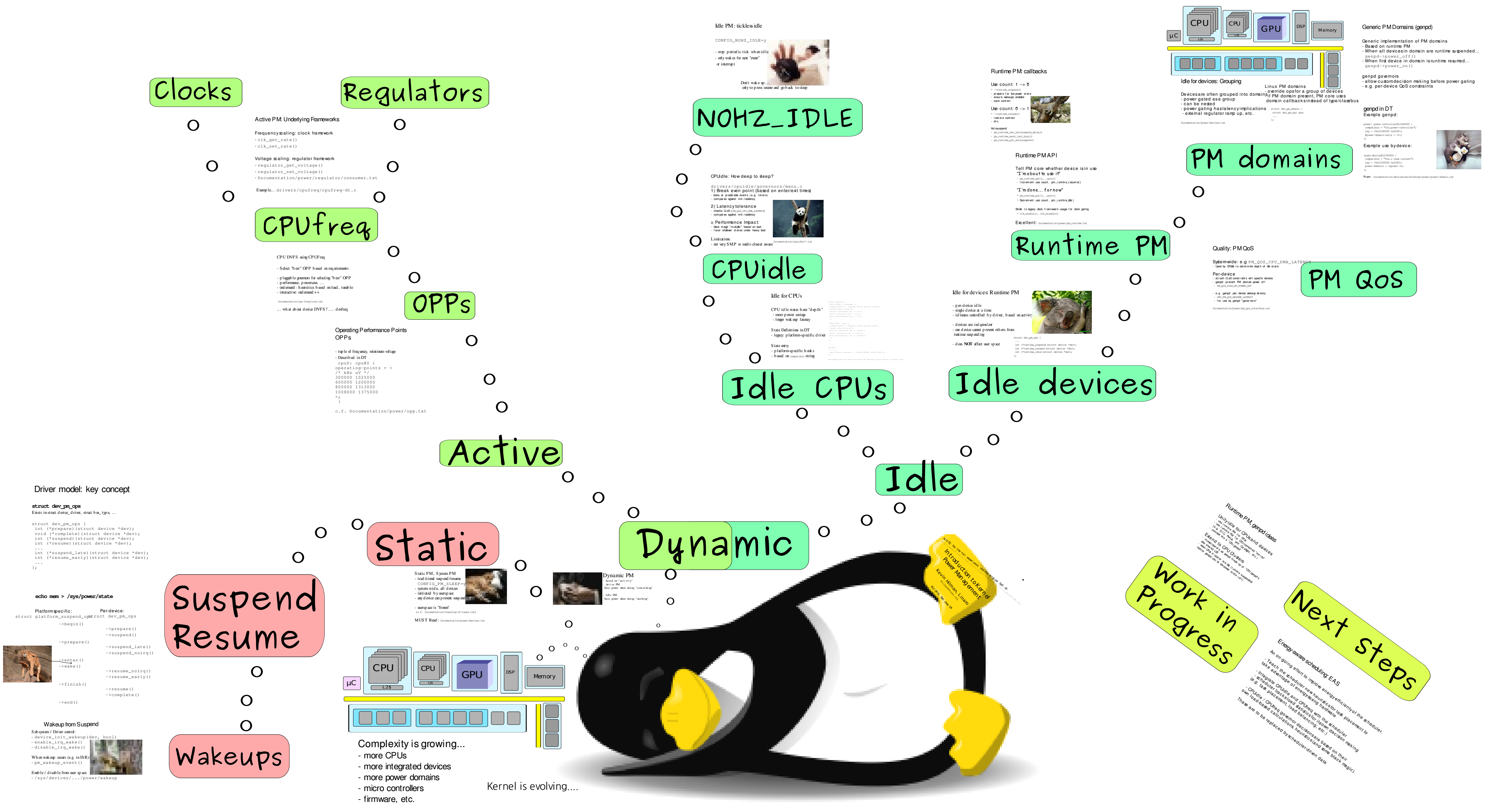
Frequency scaling: clock framework

- `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`


Clocks
Regulators
Active PM: Underlying Frameworks
CPUfreq
OPPs
Operating Performance Points
OPPs
c.f. Documentation/power/opp.txt
Active
NOHZ_IDLE
Idle PM: tickless idle
CONFIG_NO_HZ_IDLE=y
CPUidle
CPUidle: How deep to sleep?
1) Break even point (based on enter/exit times)
2) Latency tolerance
Idle CPUs
Runtime PM
Runtime PM callbacks
Runtime PM API
Idle devices
Idle for devices: Runtime PM
PM domains
Generic PM Domains (genpd)
Idle for devices: Grouping
Linux PM domains
PM QoS
Quality: PM QoS
Idle
Dynamic
Static
Driver model: key concept
struct dev_pm_ops
echo mem > /sys/power/state
Suspend Resume
Wakeups
Wakeup from Suspend
CPU
GPU
DSP
Memory
μC
Complexity is growing...
- more CPUs
- more integrated devices
- more power domains
- micro controllers
- firmware, etc.
Kernel is evolving....
Introduction to Kernel Power Management
Kevin Hilman, Linaro
Work in Progress
Next Steps
Energy-aware scheduling: EAS

Idle

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

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

Limitations:
- not very SMP or multi-cluster aware

Documentation/cpuidle/*.txt

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

Runtime PM

Runtime PM: 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
- 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"
- pm_runtime_get(), _sync()
- Increment use count, pm_runtime_resume()

"I'm done... for now"
= pm_runtime_put(), _sync()
- Decrement use count, pm_runtime_idle()

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

Excellent: Documentation/power/pm_runtime.txt

Idle devices

Idle for devices: Runtime PM

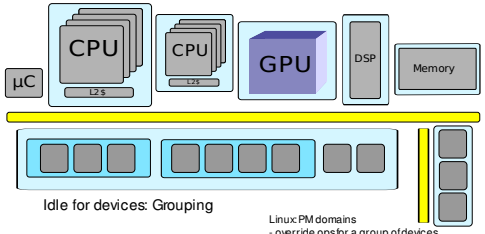
- per-device idle
- single device at a time
- idleness controlled by driver, based on activity

- 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);
};
```

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 (genpd)

Generic implementation of PM domains
- 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 governors
- allow custom decision making before power gating
- e.g. per-device QoS constraints

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 D>;
};
```

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

PM QoS

Quality: 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 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 = <2>;
 };

 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: 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

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

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

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

Limitations:
- not very SMP or multi-cluster aware

`Documentation/cpuidle/*.txt`

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

Runtime PM

Runtime PM: 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
- `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"
- `pm_runtime_get()`, `_sync()`
- Increment use count, `pm_runtime_resume()`

"I'm done... for now"
= `pm_runtime_put()`, `_sync()`
- Decrement use count, `pm_runtime_idle()`

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

Excellent: `Documentation/power/pm_runtime.txt`

Idle devices

Idle for devices: Runtime PM

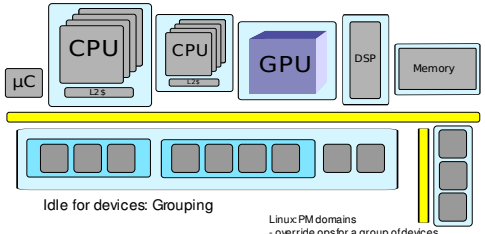
- per-device idle
- single device at a time
- idleness controlled by driver, based on activity

- 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);
};
```

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 (genpd)

Generic implementation of PM domains
- 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 governors
- allow custom decision making before power gating
- e.g. per-device QoS constraints

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`

PM QoS

Quality: 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);
};
```

Runtime PM API

Tell PM core whether device is in use

"I'm about to use it"

- `pm_runtime_get(), _sync()`
- Increment use count, pm_runtime_resume()

"I'm done... for now"

- `pm_runtime_put(), _sync()`
- Decrement use count, pm_runtime_idle()

Similar to legacy clock framework usage for clock gating

- `clk_enable(), clk_disable()`

Excellent: `Documentation/power/pm_runtime.txt`

Runtime PM: 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

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

Idle

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

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

Limitations:
- not very SMP or multi-cluster aware

`Documentation/cpuidle/*.txt`

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 devices

Runtime PM: 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
- `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"
- `pm_runtime_get()`, `_sync()`
- Increment use count, `pm_runtime_resume()`

"I'm done... for now"
= `pm_runtime_put()`, `_sync()`
- Decrement use count, `pm_runtime_idle()`

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

Excellent: `Documentation/power/pm_runtime.txt`

Runtime PM

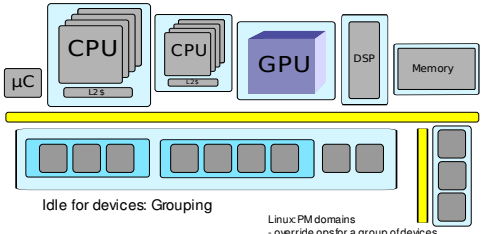
Idle for devices: Runtime PM

- 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);
};
```



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

Generic PM Domains (genpd)

Generic implementation of PM domains
- 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 governors
- allow custom decision making before power gating
- e.g. per-device QoS constraints

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`

Quality: PM QoS

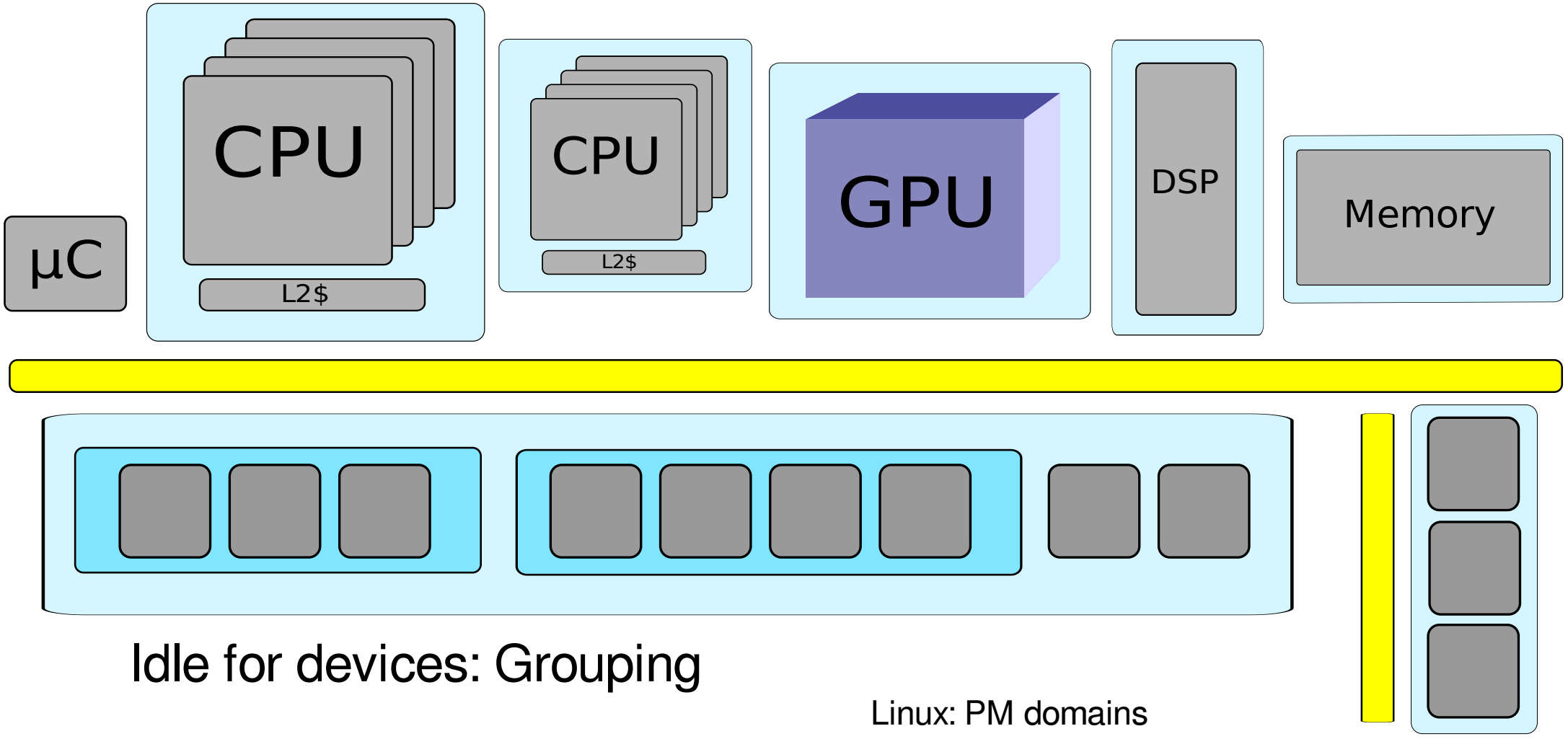
System-wide: e.g. `PM_QOS_CPU_DMA_LATENCY`
- Used by CPUidle to determine depth of idle state

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`

PM QoS



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

- 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 governors

- allow custom decision making before power gating
- e.g. per-device QoS constraints

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`

Quality: PM QoS

System-wide: e.g `PM_QOS_CPU_DMA_LATENCY`

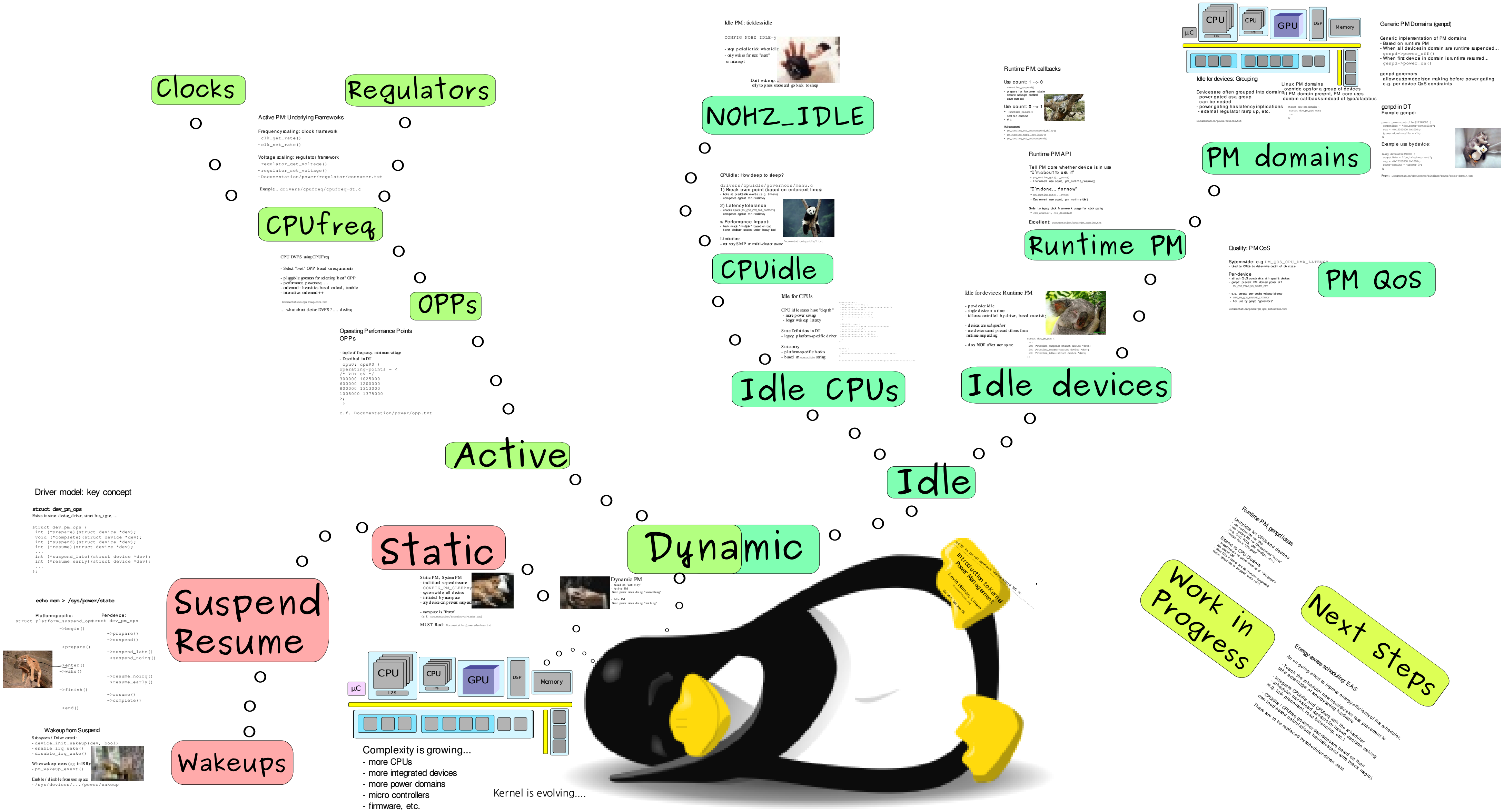
- Used by CPUidle to determine depth of idle state

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`


Clocks
Regulators
Active PM: Underlying Frameworks
CPUfreq
OPPs
Operating Performance Points
Active
NOHZ_IDLE
Idle PM: tickless idle
CPUidle
CPUidle: How deep to deep?
Idle CPUs
Runtime PM
Runtime PM callbacks
Runtime PM API
PM domains
Generic PM Domains (genpd)
Idle for devices: Grouping
PM QoS
Quality: PM QoS
Idle devices
Idle for devices: Runtime PM
Idle
Dynamic
Static
Suspend
Resume
Driver model: key concept
echo mem > /sys/power/state
Wakeups
Wakeup from Suspend
CPU
GPU
DSP
Memory
µC
Complexity is growing...
- more CPUs
- more integrated devices
- more power domains
- micro controllers
- firmware, etc.
Kernel is evolving....
Work in Progress
Next Steps
Energy-aware scheduling EAS
Runtime PM, genpd ideas

Runtime PM, genpd ideas

Unify idle for CPUs and devices

- use runtime PM for CPUs
- use runtime PM for CPU-connected "extras"
  (e.g. GIC, PMUs, VFP, CoreSight, etc.)
- combine into a "CPU genpd"

Extend to CPU Clusters

- model clusters as genpd made up of "CPU genpd"s
  plus shared L2$
- when CPUs in cluster are idle (runtime suspended)
  cluster genpd can hit low-power state (off)

Next Steps

Work in Progress

Energy-aware scheduling: EAS

An on-going effort to improve energy efficienty of the scheduler.

- Teach the scheduler new heuristics for task placement to take advantage of energy-saving hardware

- Integrate CPUidle and CPUfreq with the scheduler
  - scheduler tracks load statistics for its own decision making (e.g. task placement, load balancing, etc.)

- CPUidle / CPUfreq governor decisions are based on their own load-based calculations, heuristics (and some black magic).

These are to be replaced by scheduler-driven data

Runtime PM, genpd ideas

Unify idle for CPUs and devices

- use runtime PM for CPUs
- use runtime PM for CPU-connected "extras"
  (e.g. GIC, PMUs, VFP, CoreSight, etc.)
- combine into a "CPU genpd"

Extend to CPU Clusters

- model clusters as genpd made up of "CPU genpd"s
  plus shared L2$
- when CPUs in cluster are idle (runtime suspended)
  cluster genpd can hit low-power state (off)

Energy-aware scheduling: EAS

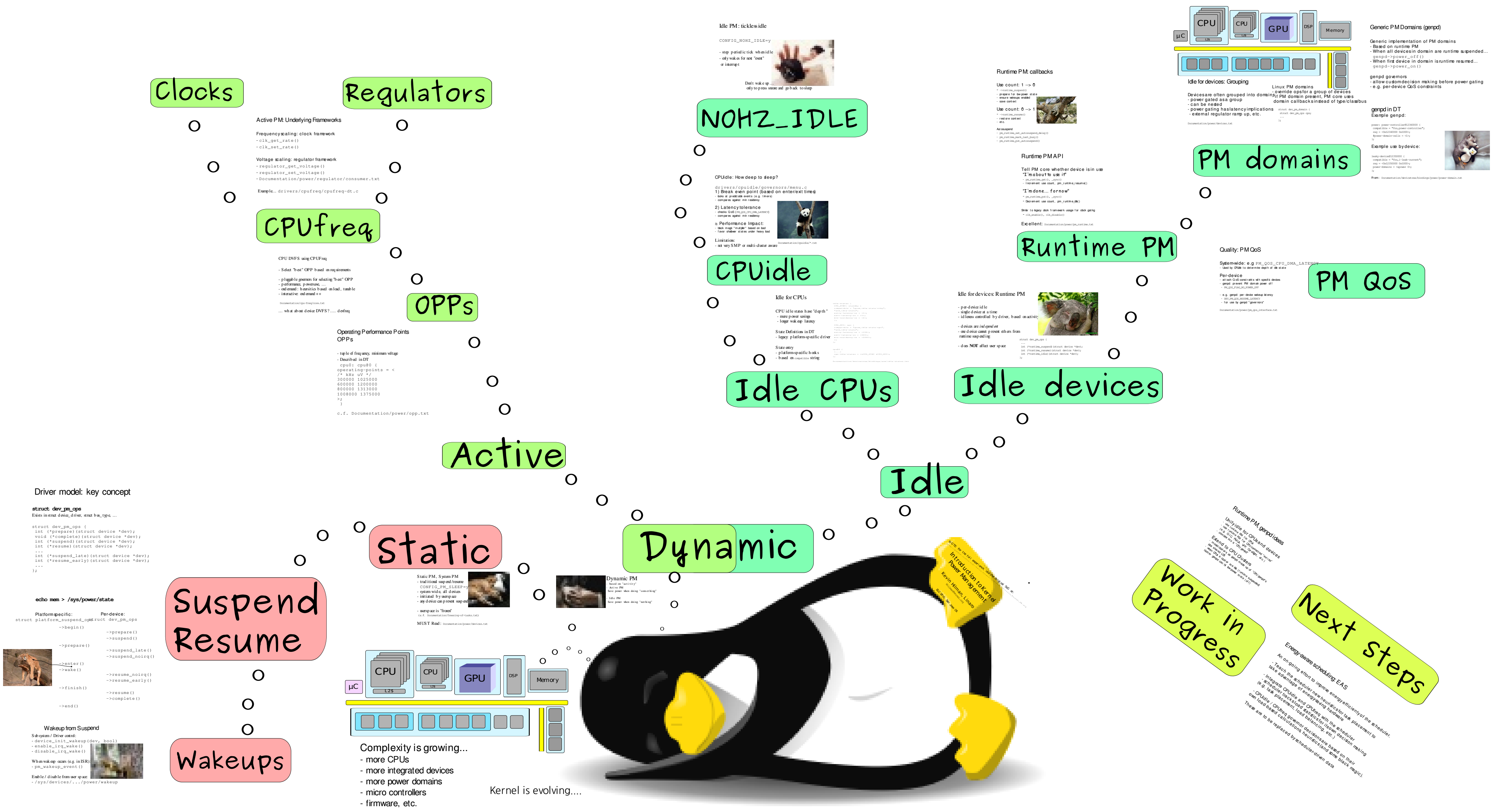
An on-going effort to improve energy efficienty of the scheduler.

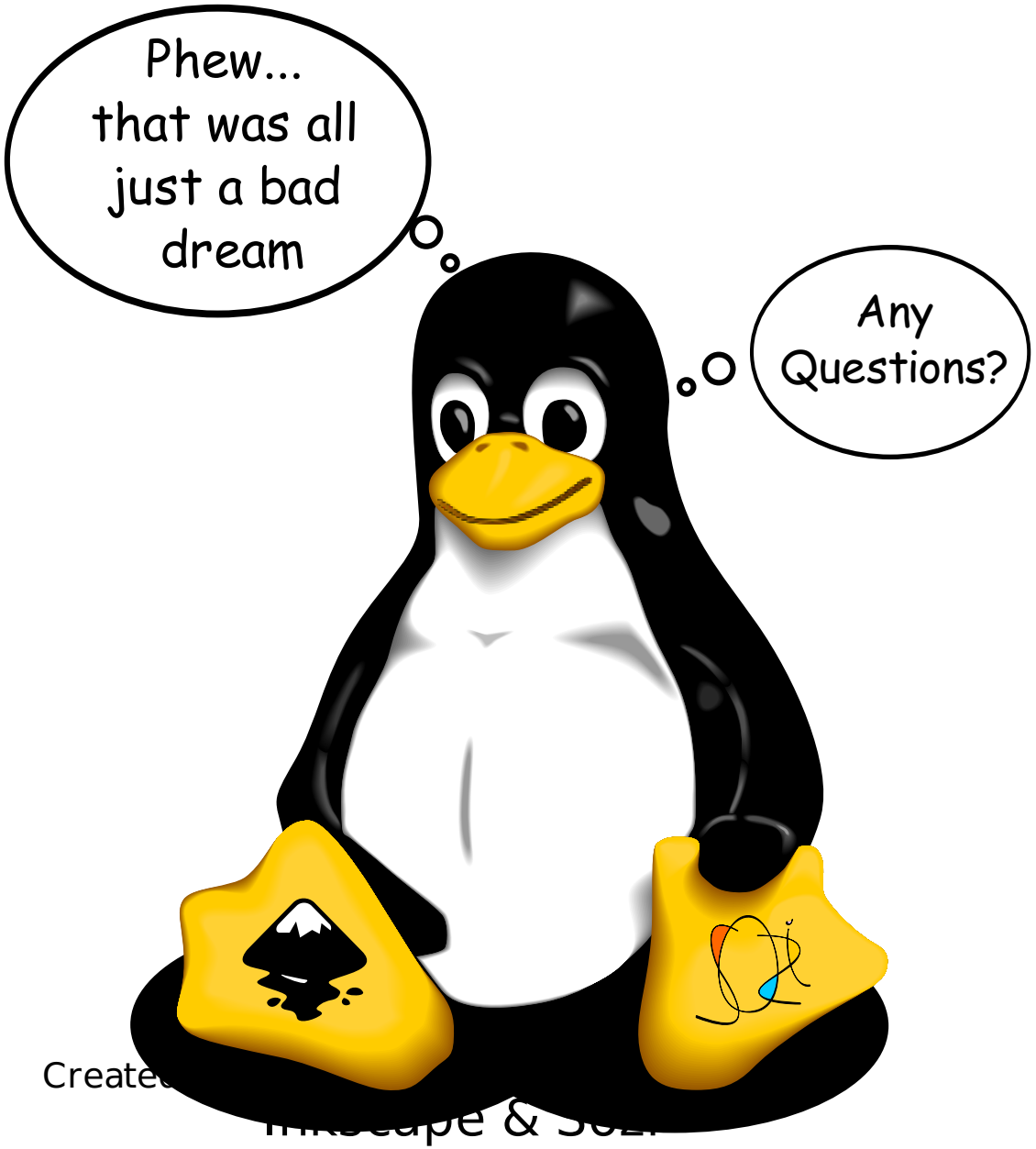
- Teach the scheduler new heuristics for task placement to take advantage of energy-saving hardware

- Integrate CPUidle and CPUfreq with the scheduler
 - scheduler tracks load statistics for its own decision making (e.g. task placement, load balancing, etc.)

 - CPUidle / CPUfreq governor decisions are based on their own load-based calculations, heuristics (and some black magic).

These are to be replaced by scheduler-driven data


Clocks
Regulators
Active PM: Underlying Frameworks
CPUfreq
OPPs
Operating Performance Points
Active
Static
Dynamic
Suspend Resume
Wakeups
Driver model: key concept
Wakeup from Suspend
NOHZ_IDLE
Idle PM: tickless idle
CPUidle
CPUidle: How deep to sleep?
Idle CPUs
Idle for CPUs
Idle
Idle devices
Idle for devices: Runtime PM
Runtime PM
Runtime PM callbacks
Runtime PM API
PM domains
Generic PM Domains (genpd)
Idle for devices: Grouping
PM QoS
Quality: PM QoS
CPU
GPU
DSP
Memory
µC
Complexity is growing...
- more CPUs
- more integrated devices
- more power domains
- micro controllers
- firmware, etc.
Kernel is evolving....
Work in Progress
Next Steps
Energy-aware scheduling: EAS
Introduction to Kernel Power Management
Kevin Hilman, Linaro



Created [illegible] Inkscape & Sozi

http://people.linaro.org/~kevin.hilman/conf/elc2015/