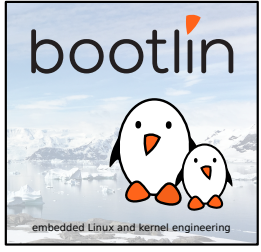
Power Management



ReDS
Reconfigurable & embedded Digital Systems

PM building blocks

- Several power management *building blocks*
  - Clock framework
  - Suspend and resume
  - Runtime power management
  - CPUidle
  - Power domains
  - Frequency and voltage scaling
- Independent *building blocks* that can be improved gradually during development

Clock framework (1)

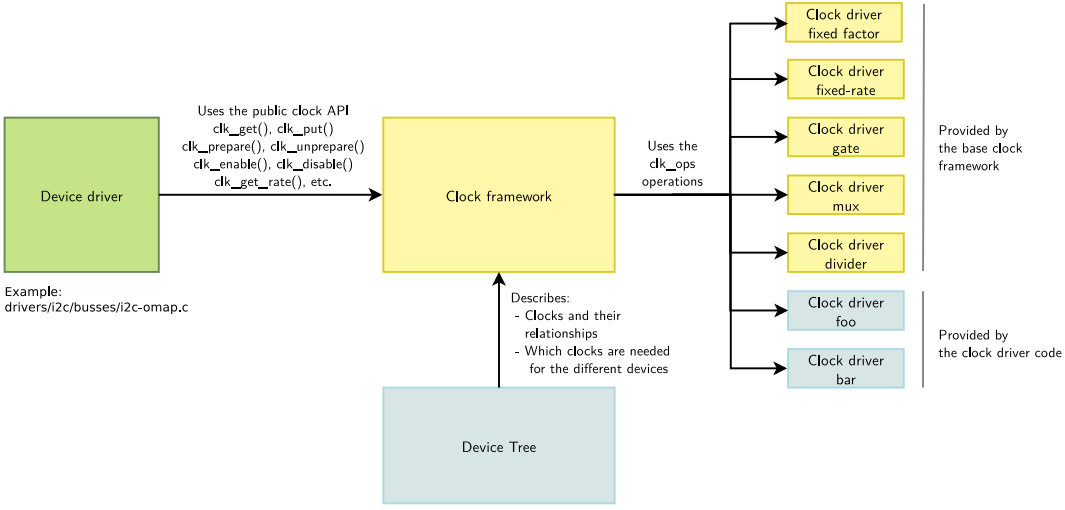
- Generic framework to manage clocks used by devices in the system
- Allows to reference count clock users and to shutdown the unused clocks to save power
- Simple API described in `include/linux/clk.h`.
  - `clk_get()` to lookup and obtain a reference to a clock producer
  - `clk_enable()` to inform the system when the clock source should be running
  - `clk_disable()` to inform the system when the clock source is no longer required.
  - `clk_put()` to free the clock source
  - `clk_get_rate()` to obtain the current clock rate (in Hz) for a clock source
  - `clk_set_rate()` to set the current clock rate (in Hz) of a clock source

Clock framework (2)

The common clock framework

- Allows to declare the available clocks and their association to devices in the Device Tree
- Provides a *debugfs* representation of the clock tree
- Is implemented in `drivers/clk/`

Diagram overview of the common clock framework



Clock framework (3)

The interface of the CCF divided into two halves:

- Common Clock Framework core
  - Common definition of `struct clk`
  - Common implementation of the `clk.h` API (defined in `drivers/clk/clk.c`)
  - `struct clk_ops`: operations invoked by the clk API implementation
  - Not supposed to be modified when adding a new driver
- Hardware-specific
  - Callbacks registered with `struct clk_ops` and the corresponding hardware-specific structures
  - Has to be written for each new hardware clock
  - Example: `drivers/clk/mvebu/clk-cpu.c`

Clock framework (4)

Hardware clock operations: device tree

- The **device tree** is the **mandatory way** to declare a clock and to get its resources, as for any other driver using DT we have to:
  - **Parse** the device tree to **setup** the clock: the resources but also the properties are retrieved.
  - Declare the **compatible** clocks and associate each to an **initialization** function using `CLK_OF_DECLARE()`
  - Example: `arch/arm/boot/dts/armada-xp.dtsi` and `drivers/clk/mvebu/armada-xp.c`

See our presentation about the Clock Framework for more details:
https://bootlin.com/pub/conferences/2013/elce/common-clock-framework-how-to-use-it/

Suspend and resume (to / from RAM)

- Infrastructure in the kernel to support suspend and resume
- System on Chip hooks
  - Define operations (at least `valid()` and `enter()`) `struct platform_suspend_ops` structure. See the documentation for this structure for details about possible operations and the way they are used.
  - Registered using the `suspend_set_ops()` function
  - See `arch/arm/mach-at91/pm.c`
- Device driver hooks
  - `pm` operations (`suspend()` and `resume()` hooks) in the `struct device_driver` as a `struct dev_pm_ops` structure in (`struct platform_driver`, `struct usb_driver`, etc.)
  - See `drivers/net/ethernet/cadence/macb_main.c`
- *Hibernate to disk* is based on suspend to RAM, copying the RAM contents (after a simulated suspend) to a swap partition.

Triggering suspend / hibernate

- `struct suspend_ops` functions are called by the `enter_state()` function. `enter_state()` also takes care of executing the suspend and resume functions for your devices.
- Read `kernel/power/suspend.c`
- The execution of this function can be triggered from user space:
  - `echo mem > /sys/power/state` (suspend to RAM)
  - `echo disk > /sys/power/state` (hibernate to disk)
- Systemd can also manage suspend and hibernate for you, and offers customizations
  - `systemctl suspend` or `systemctl hibernate`.
  - See `https://www.man7.org/linux/man-pages/man8/systemd-suspend.service.8.html`

Runtime power management

- Managing per-device idle, each device being managed by its device driver independently from others.
- According to the kernel configuration interface: *Enable functionality allowing I/O devices to be put into energy-saving (low power) states at run time (or autosuspended) after a specified period of inactivity and woken up in response to a hardware-generated wake-up event or a driver's request.*
- New hooks must be added to the drivers: `runtime_suspend()`, `runtime_resume()`, `runtime_idle()` in the `struct dev_pm_ops` structure in `struct device_driver`.
- API and details on `power/runtime_pm`
- See `drivers/net/ethernet/cadence/macb_main.c` again.

Saving power in the idle loop

- The idle loop is what you run when there's nothing left to run in the system.
- `arch_cpu_idle()` implemented in all architectures in `arch/<arch>/kernel/process.c`
- Example: `arch/arm/kernel/process.c`
- The CPU can run power saving HLT instructions, enter NAP mode, and even disable the timers (tickless systems).
- See also https://en.wikipedia.org/wiki/Idle_loop

Managing idle

Adding support for multiple idle levels

- Modern CPUs have several sleep states offering different power savings with associated wake up latencies
- The *dynamic tick* feature allows to remove the periodic timer tick to save power, and to know when the next event is scheduled, for smarter sleeps.
- CPUidle infrastructure to change sleep states
  - Platform-specific driver defining sleep states and transition operations
  - Platform-independent governors
  - Available in particular for x86/ACPI and most ARM SoCs
  - See `admin-guide/pm/cpuidle` in kernel documentation.

PowerTOP

https://en.wikipedia.org/wiki/PowerTOP

- With dynamic ticks, allows to fix parts of kernel code and applications that wake up the system too often.
- PowerTOP allows to track the worst offenders
- Now available on ARM cpus implementing CPUidle
- Also gives you useful hints for reducing power.
- Try it on your x86 laptop:
  `sudo powertop`

Generic PM Domains (genpd)

- Generic infrastructure to implement power domains based on Device Tree descriptions, allowing to group devices by the physical power domain they belong to. This sits at the same level as bus type for calling PM hooks.
- All the devices in the same PD get the same state at the same time.
- Specifications and examples available at `Documentation/devicetree/bindings/power/power_domain.txt`
- Driver example: `drivers/soc/rockchip/pm_domains.c` (`rockchip_pd_power_on()`, `rockchip_pd_power_off()`, `rockchip_pm_add_one_domain()`...)
- DT example: look for `rockchip,px30-power-controller` (`arch/arm64/boot/dts/rockchip/px30.dtsi`) and find PD definitions and corresponding devices.
- See Kevin Hilman's talk at Kernel Recipes 2017: `https://youtu.be/SctfvoskABM`

Frequency and voltage scaling (1)

Frequency and voltage scaling possible through the `cpufreq` kernel infrastructure.

- Generic infrastructure: `drivers/cpufreq/cpufreq.c` and `include/linux/cpufreq.h`
- Generic governors, responsible for deciding frequency and voltage transitions
  - `performance`: maximum frequency
  - `powersave`: minimum frequency
  - `ondemand`: measures CPU consumption to adjust frequency
  - `conservative`: often better than `ondemand`. Only increases frequency gradually when the CPU gets loaded.
  - `schedutil`: Tightly integrated with the scheduler, making per-policy decisions, RT tasks running at full speed.
  - `userspace`: leaves the decision to a user space daemon.
- This infrastructure can be controlled from `/sys/devices/system/cpu/cpu<n>/cpufreq/`

Frequency and voltage scaling (2)

- CPU frequency drivers are in `drivers/cpufreq/`. Example: `drivers/cpufreq/omap-cpufreq.c`
- Must implement the operations of the `cpufreq_driver` structure and register them using `cpufreq_register_driver()`
  - `init()` for initialization
  - `exit()` for cleanup
  - `verify()` to verify the user-chosen policy
  - `setpolicy()` or `target()` to actually perform the frequency change
- See documentation in `cpu-freq/` for useful explanations

Regulator framework

- Modern embedded platforms have hardware responsible for voltage and current regulation
- The regulator framework allows to take advantage of this hardware to save power when parts of the system are unused
  - A consumer interface for device drivers (i.e. users)
  - Regulator driver interface for regulator drivers
  - Machine interface for board configuration
  - sysfs interface for user space
- See `power/regulator/` in kernel documentation.

BSP work for a new board

In case you just need to create a BSP for your board, and your CPU already has full PM support, you should just need to:

- Create clock definitions and bind your devices to them.
- Implement PM handlers (suspend, resume) in the drivers for your board specific devices.
- Implement runtime PM handlers in your drivers.
- Implement board specific power management if needed (mainly battery management)
- Implement regulator framework hooks for your board if needed.
- Attach on-board devices to PM domains if needed
- All other parts of the PM infrastructure should be already there: suspend / resume, cpuidle, cpu frequency and voltage scaling, PM domains.

Useful resources

- power/ in kernel documentation.
  - Will give you many useful details.
- Introduction to kernel power management, Kevin Hilman (Kernel Recipes 2015)
  - https://www.youtube.com/watch?v=juJJZORgVwI