Generic PM Domains and Platform Device Drivers

Rafael J. Wysocki

Renesas Electronics / SUSE Labs / Faculty of Physics U. Warsaw

May 28, 2012

Outline

Generic PM Domains Framework

- Part of the core runtime PM framework.
- Designed for systems with "cluster" low-power states.
- Introduced one year ago (first commit on July 1, 2011, work in progress since then).
- Supports domain hierarchies (subdomains, domains with multiple master domains).
- Supports PM QoS (recently reworked).
- Support for domains containing CPU cores under development.
- Support for bus types other than the platform bus type under development.

Generic PM Domains Framework

- Part of the core runtime PM framework.
- Designed for systems with "cluster" low-power states.
- Introduced one year ago (first commit on July 1, 2011, work in progress since then).
- Supports domain hierarchies (subdomains, domains with multiple master domains).
- Supports PM QoS (recently reworked).
- Support for domains containing CPU cores under development.
- Support for bus types other than the platform bus type under development.

Assumes the availability of single-device low-power states and multi-device (cluster) low-power states.

How It Works (High-Level View)

The goal is to allow the cluster low-power states to be utilized if possible.

How It Works (High-Level View)

The goal is to allow the cluster low-power states to be utilized if possible.

`.runtime_suspend()`, `.runtime_resume()` and system suspend/hibernation callbacks are provided to replace subsystem-level (e.g. bus type) callbacks.

How It Works (High-Level View)

The goal is to allow the cluster low-power states to be utilized if possible.

`.runtime_suspend()`, `.runtime_resume()` and system suspend/hibernation callbacks are provided to replace subsystem-level (e.g. bus type) callbacks.

They use device callbacks (domain-wide or device-specific), if defined, to put devices into single-device low-power states and/or to save/restore their states.

How It Works (High-Level View)

The goal is to allow the cluster low-power states to be utilized if possible.

`.runtime_suspend()`, `.runtime_resume()` and system suspend/hibernation callbacks are provided to replace subsystem-level (e.g. bus type) callbacks.

They use device callbacks (domain-wide or device-specific), if defined, to put devices into single-device low-power states and/or to save/restore their states.

Cluster (domain) low-power states are used if they are available and if putting the system into them doesn't violate PM QoS resume latency constraints.

Generic PM Domain Representation

include/linux/pm_domain.h

```
struct generic_pm_domain {
    char *name;
    struct dev_pm_domain domain;
    struct list_head master_links, slave_links;
    struct list_head dev_list;
    struct dev_power_governor *gov;
    int (*power_off)(struct generic_pm_domain *domain);
    int (*power_on)(struct generic_pm_domain *domain);
    struct gpd_dev_ops dev_ops;
    ...
    struct device_node *of_node;
};
```

Generic PM Domain Representation

include/linux/pm_domain.h

```
struct generic_pm_domain {
    char *name;
    struct dev_pm_domain domain;
    struct list_head master_links, slave_links;
    struct list_head dev_list;
    struct dev_power_governor *gov;
    int (*power_off)(struct generic_pm_domain *domain);
    int (*power_on)(struct generic_pm_domain *domain);
    struct gpd_dev_ops dev_ops;
    ...
    struct device_node *of_node;
};
```

include/linux/pm.h

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

Domain Device Operations

include/linux/pm_domain.h

```
struct gpd_dev_ops {
    /* Runtime PM */
    int (*start)(struct device *dev);
    int (*stop)(struct device *dev);
    int (*save_state)(struct device *dev);
    int (*restore_state)(struct device *dev);
    /* System suspend and hibernation */
    int (*suspend)(struct device *dev);
    int (*suspend_late)(struct device *dev);
    int (*resume_early)(struct device *dev);
    int (*resume)(struct device *dev);
    int (*freeze)(struct device *dev);
    int (*freeze_late)(struct device *dev);
    int (*thaw_early)(struct device *dev);
    int (*thaw)(struct device *dev);
    bool (*active_wakeup)(struct device *dev);
};
```

Auxiliary Data Types

Device data and callbacks (include/linux/pm.h and pm_domain.h)

```
struct generic_pm_domain_data {
    struct pm_domain_data base;
    struct gpd_dev_ops ops;
    struct gpd_timing_data td;
    struct notifier_block nb;
    struct mutex lock;
    bool need_restore;
    bool always_on;
};
```

```
struct pm_domain_data {
    struct list_head list_node;
    struct device *dev;
};
```

Auxiliary Data Types

Device data and callbacks (include/linux/pm.h and pm_domain.h)

```
struct generic_pm_domain_data {
    struct pm_domain_data base;
    struct gpd_dev_ops ops;
    struct gpd_timing_data td;
    struct notifier_block nb;
    struct mutex lock;
    bool need_restore;
    bool always_on;
};
```

```
struct pm_domain_data {
    struct list_head list_node;
    struct device *dev;
};
```

Governor functions (include/linux/pm_domain.h)

```
struct dev_power_governor {
    bool (*power_down_ok)(struct dev_pm_domain *domain);
    bool (*stop_ok)(struct device *dev);
};
```

Domain Initialization and Subdomain Management

```
void pm_genpd_init(struct generic_pm_domain *genpd,
                   struct dev_power_governor *gov, bool is_off);
```

Domain Initialization and Subdomain Management

```
void pm_genpd_init(struct generic_pm_domain *genpd,
                   struct dev_power_governor *gov, bool is_off);
```

- Supposed to be called by the platform.
- Populates `struct generic_pm_domain` objects with defaults.
- Non-default values should be set after executing `pm_genpd_init()` and before adding any devices to the domain.

Domain Initialization and Subdomain Management

```
void pm_genpd_init(struct generic_pm_domain *genpd,
                   struct dev_power_governor *gov, bool is_off);
```

- Supposed to be called by the platform.
- Populates `struct generic_pm_domain` objects with defaults.
- Non-default values should be set after executing `pm_genpd_init()` and before adding any devices to the domain.

```
int pm_genpd_add_subdomain(struct generic_pm_domain *genpd,
                           struct generic_pm_domain *subdomain);

int pm_genpd_remove_subdomain(struct generic_pm_domain *genpd,
                              struct generic_pm_domain *subdomain);
```

Adding and Removing Devices

```
int __pm_genpd_of_add_device(struct device_node *genpd_node, struct device *dev,
                             struct gpd_timing_data *td);

int pm_genpd_add_device(struct generic_pm_domain *genpd,
                        struct device *dev);

int pm_genpd_remove_device(struct generic_pm_domain *genpd,
                           struct device *dev);
```

Adding and Removing Devices

```
int __pm_genpd_of_add_device(struct device_node *genpd_node, struct device *dev,
                             struct gpd_timing_data *td);

int pm_genpd_add_device(struct generic_pm_domain *genpd,
                        struct device *dev);

int pm_genpd_remove_device(struct generic_pm_domain *genpd,
                           struct device *dev);
```

- Supposed to be called by the platform (calling one of them from a device driver is a layering violation).
- It is recommended to add devices to domains before registering drivers (so that `.probe()` can see that the device is in a PM domain).
- The domain object should be configured entirely before the first device is added to that domain.

Device Trees Support

```
int __pm_genpd_of_add_device(struct device_node *genpd_node,
                             struct device *dev,
                             struct gpd_timing_data *td);

int pm_genpd_of_add_device(struct device_node *genpd_node,
                           struct device *dev);
```

Device Trees Support

```
int __pm_genpd_of_add_device(struct device_node *genpd_node,
                             struct device *dev,
                             struct gpd_timing_data *td);

int pm_genpd_of_add_device(struct device_node *genpd_node,
                           struct device *dev);
```

- The domain to add the device to is found on the basis of its device tree node pointer.
- For this to work, the `of_node` member of `struct generic_pm_domain` has to be set appropriately (this does not happen automatically).

Runtime Suspend Callback Routine

```
int pm_genpd_runtime_suspend(struct device *dev);
```

Runtime Suspend Callback Routine

```
int pm_genpd_runtime_suspend(struct device *dev);
```

1. If `.stop_ok()` returns `true`, the device is "stopped".
   - `.stop()` callback from the `dev_ops` member of `struct generic_pm_domain` (domain-wide, none by default).
   - `.stop()` callback from the `ops` member of `struct generic_pm_domain_data` attached to the given device (device-specific).
2. `pm_genpd_poweroff()` is called for the device's domain.
3. If all devices in the domain are "stopped" and all its subdomains are "off", and `.power_down_ok()` returned `true`:
   1. The states of all devices in the domain are saved (using driver callbacks).
   2. The "power off" operation is carried out for the domain.

Saving Device States

There are two possible ways to save the state of a device before the "power off" operation is carried out for its domain:

- `.save_state()` callback from the `dev_ops` member of `struct generic_pm_domain` (domain-wide).
- `.save_state()` callback from the `ops` member of `struct generic_pm_domain_data` attached to the given device (device-specific).

Saving Device States

There are two possible ways to save the state of a device before the "power off" operation is carried out for its domain:

- `.save_state()` callback from the `dev_ops` member of `struct generic_pm_domain` (domain-wide).
- `.save_state()` callback from the `ops` member of `struct generic_pm_domain_data` attached to the given device (device-specific).

The domain-wide callback takes precedence over the device-specific one.

Saving Device States

There are two possible ways to save the state of a device before the "power off" operation is carried out for its domain:

- `.save_state()` callback from the `dev_ops` member of `struct generic_pm_domain` (domain-wide).
- `.save_state()` callback from the `ops` member of `struct generic_pm_domain_data` attached to the given device (device-specific).

The domain-wide callback takes precedence over the device-specific one.

By default `dev_ops.save_state` points to `pm_genpd_default_save_state()` that executes the device-specific callback or falls back to its driver's `.runtime_suspend()`.

Runtime Resume Callback Routine

```
int pm_genpd_runtime_resume(struct device *dev);
```

Runtime Resume Callback Routine

```
int pm_genpd_runtime_resume(struct device *dev);
```

1. If necessary, the "power on" operation is carried out for the device's domain.
   - This has to abort all instances of `pm_genpd_poweroff()` running for the same domain at that time.
   - It has to be run recursively for all of the "master" domains before.
2. The "start" operation is carried out for the device.
   - `.start()` callback from the `dev_ops` member of `struct generic_pm_domain` (domain-wide, none by default).
   - `.start()` callback from the `ops` member of `struct generic_pm_domain_data` attached to the given device (device-specific).
3. If necessary, the device's state is restored (using driver callback).

Restoring Device States

There are two possible ways to restore the state of a device after the "power on" operation has been carried out for its domain:

- `.restore_state()` callback from the `dev_ops` member of `struct generic_pm_domain` (domain-wide).
- `.restore_state()` callback from the `ops` member of `struct generic_pm_domain_data` attached to the given device (device-specific).

Restoring Device States

There are two possible ways to restore the state of a device after the "power on" operation has been carried out for its domain:

- `.restore_state()` callback from the `dev_ops` member of `struct generic_pm_domain` (domain-wide).
- `.restore_state()` callback from the `ops` member of `struct generic_pm_domain_data` attached to the given device (device-specific).

The domain-wide callback takes precedence over the device-specific one.

Restoring Device States

There are two possible ways to restore the state of a device after the "power on" operation has been carried out for its domain:

- `.restore_state()` callback from the `dev_ops` member of `struct generic_pm_domain` (domain-wide).
- `.restore_state()` callback from the `ops` member of `struct generic_pm_domain_data` attached to the given device (device-specific).

The domain-wide callback takes precedence over the device-specific one.

By default `dev_ops.restore_state` points to `pm_genpd_default_restore_state()` that executes the device-specific callback or falls back to its driver's `.runtime_resume()`.

System Suspend and Hibernation Callbacks

They are organized in analogy with the runtime PM callbacks (i.e. for each supported suspend/resume or hibernation/restore stage there may be a device-specific callback and a domain-wide callback, with the former taking precedence).

System Suspend and Hibernation Callbacks

They are organized in analogy with the runtime PM callbacks (i.e. for each supported suspend/resume or hibernation/restore stage there may be a device-specific callback and a domain-wide callback, with the former taking precedence).

Separate device callbacks are only defined for:

- The “suspend”, “late suspend”, “early resume”, and “resume” phases of system suspend.
- The “freeze”, “late freeze”, “early thaw”, and “thaw” phases of hibernation.

System Suspend and Hibernation Callbacks

They are organized in analogy with the runtime PM callbacks (i.e. for each supported suspend/resume or hibernation/restore stage there may be a device-specific callback and a domain-wide callback, with the former taking precedence).

Separate device callbacks are only defined for:

- The "suspend", "late suspend", "early resume", and "resume" phases of system suspend.
- The "freeze", "late freeze", "early thaw", and "thaw" phases of hibernation.

The suspend callbacks are also used during the last stage of hibernation (i.e. after the image has been saved) and the remaining phases are handled with the help of the "stop" and "start" device callbacks.

Roles of Domain Device Callbacks (Runtime PM)

`.stop()` Put the device into a single-device low-power state (e.g. stop its clock). Configure remote wakeup if necessary. Save some state data if necessary.

Roles of Domain Device Callbacks (Runtime PM)

`.stop()` Put the device into a single-device low-power state (e.g. stop its clock). Configure remote wakeup if necessary. Save some state data if necessary.

`.start()` Put the device back into the full-power state. Restore the state data saved by `.stop()` if necessary.

Roles of Domain Device Callbacks (Runtime PM)

`.stop()` Put the device into a single-device low-power state (e.g. stop its clock). Configure remote wakeup if necessary. Save some state data if necessary.

`.start()` Put the device back into the full-power state. Restore the state data saved by `.stop()` if necessary.

`.save_state()` Prepare the device for putting its domain into a cluster (domain) low-power state (e.g. power removal). Usually, save the device's state.

Roles of Domain Device Callbacks (Runtime PM)

`.stop()` Put the device into a single-device low-power state (e.g. stop its clock). Configure remote wakeup if necessary. Save some state data if necessary.

`.start()` Put the device back into the full-power state. Restore the state data saved by `.stop()` if necessary.

`.save_state()` Prepare the device for putting its domain into a cluster (domain) low-power state (e.g. power removal). Usually, save the device's state.

`.restore_state()` Prepare the device for a transition into the full-power state after it has gone through a domain low-power state. Usually, restore the device's state.

Roles of Domain Device Callbacks (Runtime PM)

`.stop()` Put the device into a single-device low-power state (e.g. stop its clock). Configure remote wakeup if necessary. Save some state data if necessary.

`.start()` Put the device back into the full-power state. Restore the state data saved by `.stop()` if necessary.

`.save_state()` Prepare the device for putting its domain into a cluster (domain) low-power state (e.g. power removal). Usually, save the device's state.

`.restore_state()` Prepare the device for a transition into the full-power state after it has gone through a domain low-power state. Usually, restore the device's state.

Note

`.save_state()` is preceded by `.start()` and followed by `.stop()`.

Default Behavior May Not Be Correct

If the device-specific callback is not present (i.e. the callback pointer in the `ops` member of the `struct generic_pm_domain_data` object attached to the given device is `NULL`), `pm_genpd_default_save_state()` will execute the device driver's `.runtime_suspend()` callback.

Default Behavior May Not Be Correct

If the device-specific callback is not present (i.e. the callback pointer in the `ops` member of the `struct generic_pm_domain_data` object attached to the given device is `NULL`), `pm_genpd_default_save_state()` will execute the device driver's `.runtime_suspend()` callback.

This may not be the right thing to do if the domain provides sufficiently complicated `.stop()` and `.start()` device callbacks.

Default Behavior May Not Be Correct

If the device-specific callback is not present (i.e. the callback pointer in the `ops` member of the `struct generic_pm_domain_data` object attached to the given device is `NULL`), `pm_genpd_default_save_state()` will execute the device driver's `.runtime_suspend()` callback.

This may not be the right thing to do if the domain provides sufficiently complicated `.stop()` and `.start()` device callbacks.

For example, the driver's `.runtime_suspend()` may do something that's duplicated by the domain's `.stop()`.

Default Behavior May Not Be Correct

If the device-specific callback is not present (i.e. the callback pointer in the `ops` member of the `struct generic_pm_domain_data` object attached to the given device is `NULL`), `pm_genpd_default_save_state()` will execute the device driver's `.runtime_suspend()` callback.

This may not be the right thing to do if the domain provides sufficiently complicated `.stop()` and `.start()` device callbacks.

For example, the driver's `.runtime_suspend()` may do something that's duplicated by the domain's `.stop()`.

Analogous observation applies to `pm_genpd_default_restore_state()` and the driver's `.runtime_resume()`.

Universal Runtime Suspend and Resume Callbacks

In principle, it is possible to design the `.runtime_suspend()` and `.runtime_resume()` callbacks of a platform device driver in such a way that they will work with generic PM domains as well as with the platform bus type.

Universal Runtime Suspend and Resume Callbacks

In principle, it is possible to design the `.runtime_suspend()` and `.runtime_resume()` callbacks of a platform device driver in such a way that they will work with generic PM domains as well as with the platform bus type.

Potential problems

- Platform device drivers are supposed to handle device power management entirely by themselves (the bus type does not provide any useful functionality in that respect).

Universal Runtime Suspend and Resume Callbacks

In principle, it is possible to design the `.runtime_suspend()` and `.runtime_resume()` callbacks of a platform device driver in such a way that they will work with generic PM domains as well as with the platform bus type.

Potential problems

- Platform device drivers are supposed to handle device power management entirely by themselves (the bus type does not provide any useful functionality in that respect).
- Universal callbacks may need to make assumptions about the platform that will make the driver platform-specific.

Universal Runtime Suspend and Resume Callbacks

In principle, it is possible to design the `.runtime_suspend()` and `.runtime_resume()` callbacks of a platform device driver in such a way that they will work with generic PM domains as well as with the platform bus type.

Potential problems

- Platform device drivers are supposed to handle device power management entirely by themselves (the bus type does not provide any useful functionality in that respect).
- Universal callbacks may need to make assumptions about the platform that will make the driver platform-specific.

In the end, trying to design universal (working for PM domains as well as for the platform bus type) PM callbacks may not be worth the effort.

Defining Device-Specific Callbacks

The alternative is to make PM domains use special "domain" callbacks instead of the driver's `.runtime_suspend()` and `.runtime_resume()`.

Defining Device-Specific Callbacks

The alternative is to make PM domains use special "domain" callbacks instead of the driver's `.runtime_suspend()` and `.runtime_resume()`.

```
int pm_genpd_add_callbacks(struct device *dev, struct gpd_dev_ops *ops,
                           struct gpd_timing_data *td);
```

Defining Device-Specific Callbacks

The alternative is to make PM domains use special "domain" callbacks instead of the driver's `.runtime_suspend()` and `.runtime_resume()`.

```
int pm_genpd_add_callbacks(struct device *dev, struct gpd_dev_ops *ops,
                           struct gpd_timing_data *td);
```

This routine populates the `ops` member of the `struct generic_pm_domain_data` object attached to the given device.

Defining Device-Specific Callbacks

The alternative is to make PM domains use special "domain" callbacks instead of the driver's `.runtime_suspend()` and `.runtime_resume()`.

```
int pm_genpd_add_callbacks(struct device *dev, struct gpd_dev_ops *ops,
                           struct gpd_timing_data *td);
```

This routine populates the `ops` member of the `struct generic_pm_domain_data` object attached to the given device.

It returns `-EINVAL` if the device is not in a generic PM domain.

Defining Device-Specific Callbacks

The alternative is to make PM domains use special "domain" callbacks instead of the driver's `.runtime_suspend()` and `.runtime_resume()`.

```
int pm_genpd_add_callbacks(struct device *dev, struct gpd_dev_ops *ops,
                           struct gpd_timing_data *td);
```

This routine populates the `ops` member of the `struct generic_pm_domain_data` object attached to the given device.

It returns `-EINVAL` if the device is not in a generic PM domain.

This allows the driver to attach its own set of domain callbacks to the device in case it belongs to a (generic) PM domain.

How That Can Be Done

For example, the driver can do the following:

How That Can Be Done

For example, the driver can do the following:

1. Define an object of type `struct gpd_dev_ops` whose members will point to a set of routines to be used with generic PM domains.

How That Can Be Done

For example, the driver can do the following:

1. Define an object of type `struct gpd_dev_ops` whose members will point to a set of routines to be used with generic PM domains.
2. Define a static object of type `struct dev_pm_ops` whose members will point to a set of routines to be used with the platform bus type.

How That Can Be Done

For example, the driver can do the following:

1. Define an object of type `struct gpd_dev_ops` whose members will point to a set of routines to be used with generic PM domains.
2. Define a static object of type `struct dev_pm_ops` whose members will point to a set of routines to be used with the platform bus type.
3. In its `.probe()` routine, call `pm_genpd_add_callbacks()` passing the pointer to the `struct gpd_dev_ops` object as its second argument.

How That Can Be Done

For example, the driver can do the following:

1. Define an object of type `struct gpd_dev_ops` whose members will point to a set of routines to be used with generic PM domains.
2. Define a static object of type `struct dev_pm_ops` whose members will point to a set of routines to be used with the platform bus type.
3. In its `.probe()` routine, call `pm_genpd_add_callbacks()` passing the pointer to the `struct gpd_dev_ops` object as its second argument.
4. If that returns 0 (success), set its `driver.pm` pointer to `NULL`.

How That Can Be Done

For example, the driver can do the following:

1. Define an object of type `struct gpd_dev_ops` whose members will point to a set of routines to be used with generic PM domains.
2. Define a static object of type `struct dev_pm_ops` whose members will point to a set of routines to be used with the platform bus type.
3. In its `.probe()` routine, call `pm_genpd_add_callbacks()` passing the pointer to the `struct gpd_dev_ops` object as its second argument.
4. If that returns 0 (success), set its `driver.pm` pointer to `NULL`.
5. Otherwise, set its `driver.pm` pointer to the address of the `struct dev_pm_ops` object.

How That Can Be Done

For example, the driver can do the following:

1. Define an object of type `struct gpd_dev_ops` whose members will point to a set of routines to be used with generic PM domains.
2. Define a static object of type `struct dev_pm_ops` whose members will point to a set of routines to be used with the platform bus type.
3. In its `.probe()` routine, call `pm_genpd_add_callbacks()` passing the pointer to the `struct gpd_dev_ops` object as its second argument.
4. If that returns 0 (success), set its `driver.pm` pointer to `NULL`.
5. Otherwise, set its `driver.pm` pointer to the address of the `struct dev_pm_ops` object.

This has to cover system suspend and hibernation too.

Highlights

- The generic PM domains framework allows multi-device (domain) low-power states to be used as well as single-device ones.

Highlights

- The generic PM domains framework allows multi-device (domain) low-power states to be used as well as single-device ones.
- It replaces the subsystem-level (e.g. bus type) PM callbacks with its own routines that take the domain low-power states into account.

Highlights

- The generic PM domains framework allows multi-device (domain) low-power states to be used as well as single-device ones.
- It replaces the subsystem-level (e.g. bus type) PM callbacks with its own routines that take the domain low-power states into account.
- Those routines use callbacks provided by the platform and by device drivers to implement the desired functionality.

Highlights

- The generic PM domains framework allows multi-device (domain) low-power states to be used as well as single-device ones.
- It replaces the subsystem-level (e.g. bus type) PM callbacks with its own routines that take the domain low-power states into account.
- Those routines use callbacks provided by the platform and by device drivers to implement the desired functionality.
- The callbacks provided by device drivers may be either the "standard" ones designed to be used with the platform bus type, or special ones designed specifically with PM domains in mind.

Highlights

- The generic PM domains framework allows multi-device (domain) low-power states to be used as well as single-device ones.
- It replaces the subsystem-level (e.g. bus type) PM callbacks with its own routines that take the domain low-power states into account.
- Those routines use callbacks provided by the platform and by device drivers to implement the desired functionality.
- The callbacks provided by device drivers may be either the "standard" ones designed to be used with the platform bus type, or special ones designed specifically with PM domains in mind.
- If everything is set up correctly by the platform, the driver can decide which set of power management callbacks to use at probe time.

References


R. J. Wysocki, *Why We Need More Device Power Management Callbacks* (https://events.linuxfoundation.org/images/stories/pdf/lfcs2012_wysocki.pdf).


R. J. Wysocki, *Power Management Using PM Domains on SH7372* (https://events.linuxfoundation.org/events/embedded-linux-conference-europe/wysocki).


R. J. Wysocki, *Runtime PM vs System Sleep* (http://www.linuxplumbersconf.org/2011/ocw/system/presentations/27/original/system_sleep_vs_runtime_PM.pdf).


R. J. Wysocki, *Runtime Power Management Framework for I/O Devices in the Linux Kernel* (http://events.linuxfoundation.org/slides/2010/linuxcon2010_wysocki.pdf).

Documentation And Source Code

- Documentation/power/devices.txt
- Documentation/power/runtime_pm.txt
- include/linux/device.h
- include/linux/pm.h
- include/linux/pm_domain.h
- include/linux/pm_runtime.h
- include/linux/pm_wakeup.h
- include/linux/suspend.h
- drivers/base/power/*
- kernel/power/*

Power Domains On SH7372

C5: base (mother) domain, CPG, KEYSC, CMT, RWDT, GPIO

A4LC: LCDC, DSI, MERAM (video)

A4MP: SPU2, FSI (audio)

D4: ARM debug

A4R: SH4AL-DSP, INTCS, DMAC, IIC, TMU, MSIOF, CMT0, CEU, CSI (SH CPU core, I/O)

A3RI: ISP (camera capture unit)

A3RV: VPU (video encode/decode unit)

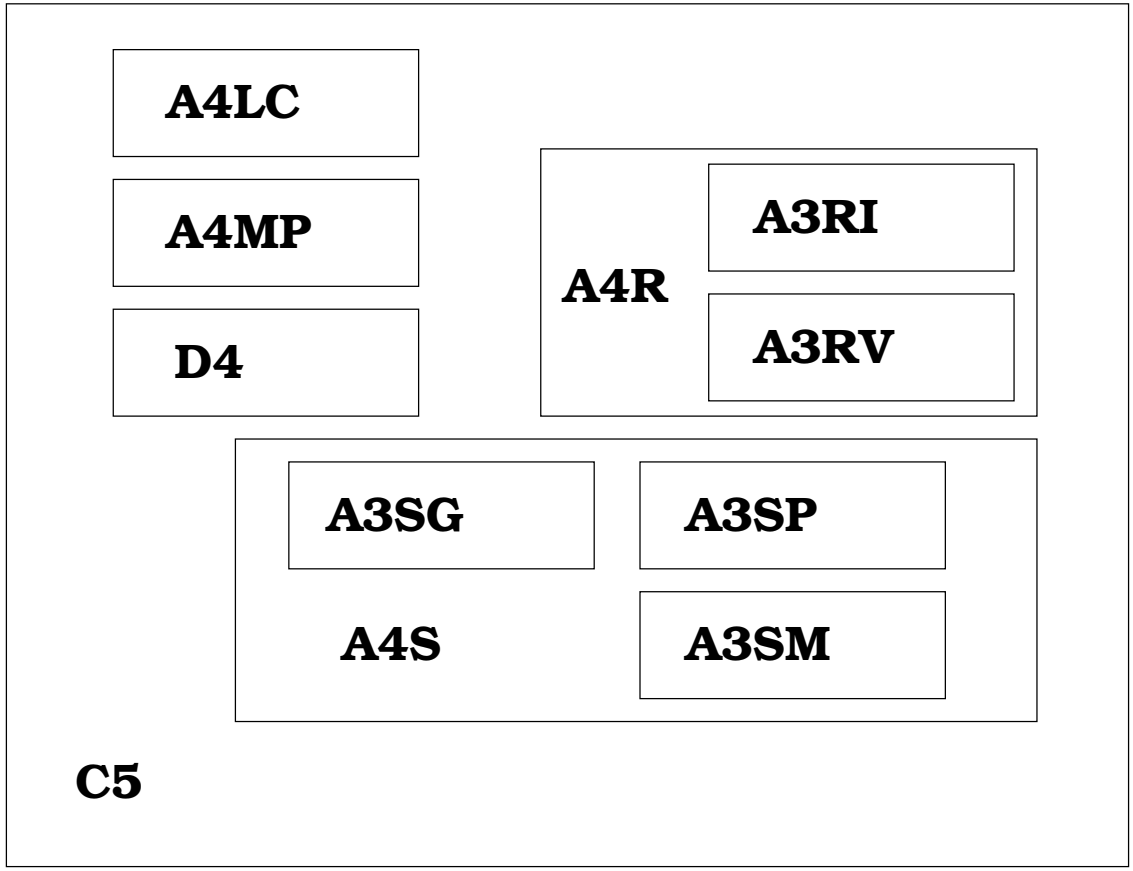
A4S: INTCA, MFI, SBSC (interrupt and SDRAM controllers)

A3SG: SGX (3D graphics)

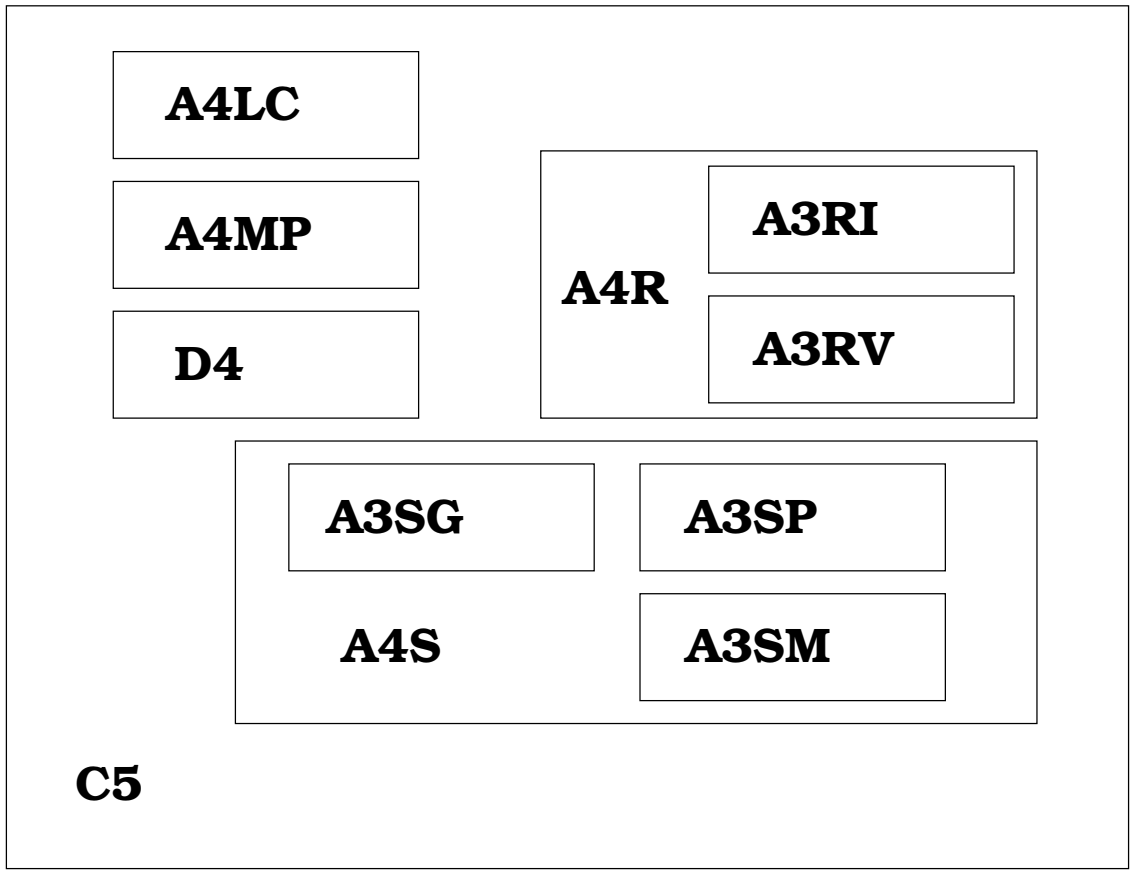
A3SP: SCIF, MSIOF, IIC, USB, SDHI, MMCIF, HDMI (I/O)

A3SM: ARM Cortex-A8 CPU core

Power Domains Hierarchy

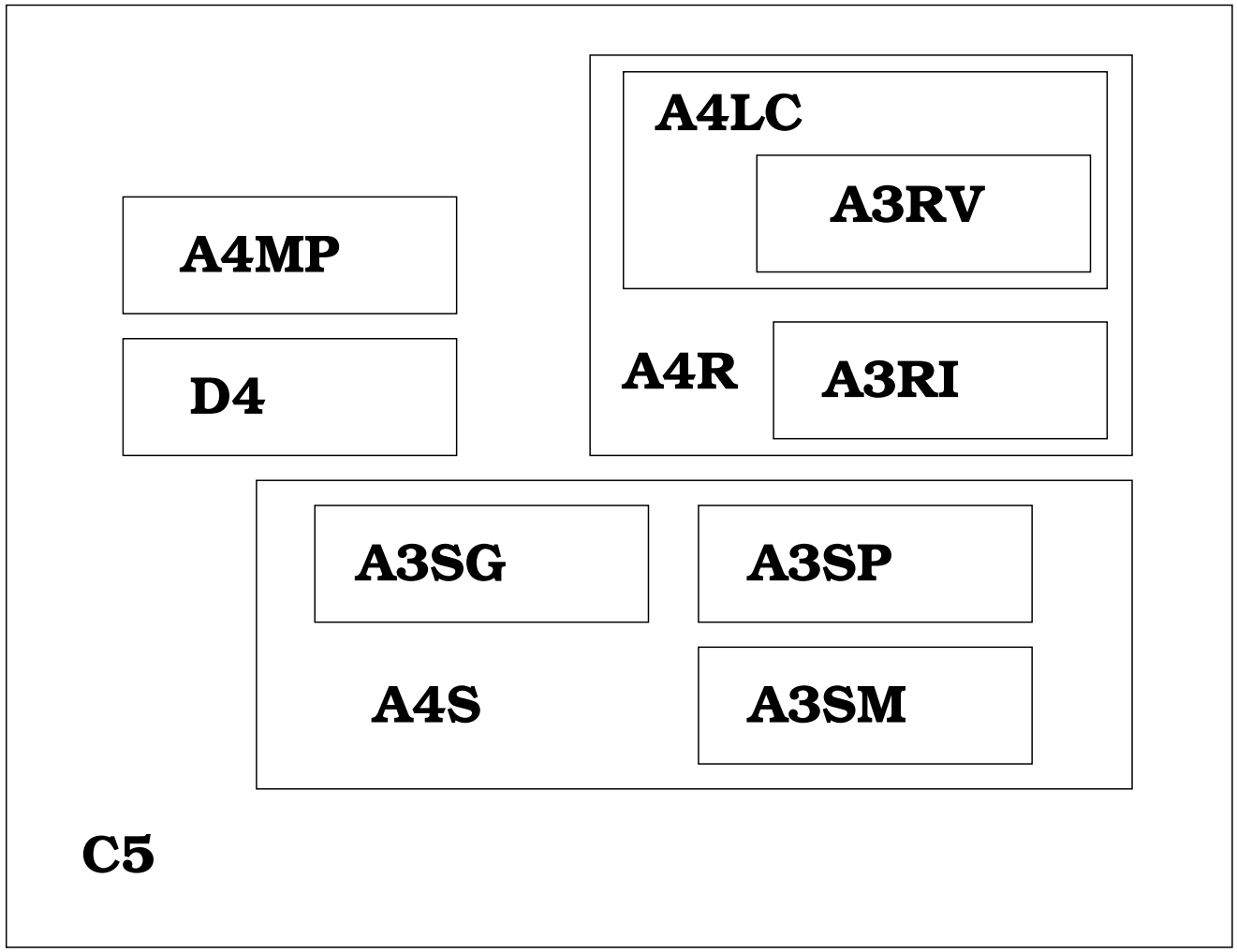


Power Domains Hierarchy



It turns out that A3RV depends on A4LC (because of MERAM) and A4LC depends on A4R (because of the INTCS).

Effective Power Domains Hierarchy



Consequences of the Design

Observations

- Every device is a direct member of one power domain.
- It is desirable to turn off A4LC when A3RV is off.
- It is desirable to turn off A4R when A3RI and A4LC are off.
- It is desirable to turn off A4S when A3SG, A3SP, A3SM are off.

Consequences of the Design

Observations

- Every device is a direct member of one power domain.
- It is desirable to turn off A4LC when A3RV is off.
- It is desirable to turn off A4R when A3RI and A4LC are off.
- It is desirable to turn off A4S when A3SG, A3SP, A3SM are off.

Plan

We will play with a fake (software-only) device added to the A4LC domain, because it is easy to trigger the "power off" and "power on" transitions in it by blanking and unblanking the screen, respectively.

Preliminary Patch

Make the `.runtime_suspend()` and `.runtime_resume()` callbacks in `default_pm_domain` for `shmobile` execute driver callbacks.

Preliminary Patch

Make the `.runtime_suspend()` and `.runtime_resume()` callbacks in `default_pm_domain` for `shmobile` execute driver callbacks.

```
Index: linux/drivers/sh/pm_runtime.c
===================================================================
--- linux.orig/drivers/sh/pm_runtime.c
+++ linux/drivers/sh/pm_runtime.c
@@ -28,10 +28,35 @@ static int default_platform_runtime_idle
        return pm_runtime_suspend(dev);
 }

+static int default_runtime_suspend(struct device *dev)
+{
+       struct device_driver *drv = dev->driver;
+
+       if (drv && drv->pm && drv->pm->runtime_suspend) {
+               int ret = drv->pm->runtime_suspend(dev);
+               if (ret)
+                       return ret;
+       }
+       return pm_clk_suspend(dev);
+}
+
```

Preliminary Patch Continued

```
+static int default_runtime_resume(struct device *dev)
+{
+       struct device_driver *drv = dev->driver;
+       int ret;
+
+       ret = pm_clk_resume(dev);
+       if (ret)
+               return ret;
+
+       return drv && drv->pm && drv->pm->runtime_resume ?
+               drv->pm->runtime_resume(dev) : 0;
+}
+
 static struct dev_pm_domain default_pm_domain = {
        .ops = {
-               .runtime_suspend = pm_clk_suspend,
-               .runtime_resume = pm_clk_resume,
+               .runtime_suspend = default_runtime_suspend,
+               .runtime_resume = default_runtime_resume,
                .runtime_idle = default_platform_runtime_idle,
                USE_PLATFORM_PM_SLEEP_OPS
        },
```

Preliminary Patch Continued

```
+static int default_runtime_resume(struct device *dev)
+{
+       struct device_driver *drv = dev->driver;
+       int ret;
+
+       ret = pm_clk_resume(dev);
+       if (ret)
+               return ret;
+
+       return drv && drv->pm && drv->pm->runtime_resume ?
+               drv->pm->runtime_resume(dev) : 0;
+}
+
 static struct dev_pm_domain default_pm_domain = {
        .ops = {
-               .runtime_suspend = pm_clk_suspend,
-               .runtime_resume = pm_clk_resume,
+               .runtime_suspend = default_runtime_suspend,
+               .runtime_resume = default_runtime_resume,
                .runtime_idle = default_platform_runtime_idle,
                USE_PLATFORM_PM_SLEEP_OPS
        },
```

That is what the platform bus type does by default.

Device Object Definition

```
Index: linux/arch/arm/mach-shmobile/board-mackerel.c
===================================================================
--- linux.orig/arch/arm/mach-shmobile/board-mackerel.c
+++ linux/arch/arm/mach-shmobile/board-mackerel.c
@@ -1285,6 +1285,14 @@ static struct platform_device mackerel_c
        },
 };

+static struct platform_device fake_device = {
+       .name = "fake-device",
+       .id = 0,
+       .dev = {
+               .platform_data = "MY FAKE DEVICE",
+       },
+};
+
 static struct platform_device *mackerel_devices[] __initdata = {
        &nor_flash_device,
        &smc911x_device,
@@ -1307,6 +1315,7 @@ static struct platform_device *mackerel_
        &hdmi_device,
        &hdmi_lcdc_device,
        &meram_device,
+       &fake_device,
 };

 /* Keypad Initialization */
```

Adding Device to PM Domain

The device can be added to the A4LC domain as follows.

Adding Device to PM Domain

The device can be added to the A4LC domain as follows.

```
Index: linux/arch/arm/mach-shmobile/board-mackerel.c
===================================================================
--- linux.orig/arch/arm/mach-shmobile/board-mackerel.c
+++ linux/arch/arm/mach-shmobile/board-mackerel.c
@@ -1592,6 +1601,7 @@ static void __init mackerel_init(void)
 #endif
        sh7372_add_device_to_domain(&sh7372_a3sp, &sdhi2_device);
        sh7372_add_device_to_domain(&sh7372_a4r, &ceu_device);
+       sh7372_add_device_to_domain(&sh7372_a4lc, &fake_device);

        hdmi_init_pm_clock();
        sh7372_pm_init();
```

Adding Device to PM Domain

The device can be added to the A4LC domain as follows.

```
Index: linux/arch/arm/mach-shmobile/board-mackerel.c
===================================================================
--- linux.orig/arch/arm/mach-shmobile/board-mackerel.c
+++ linux/arch/arm/mach-shmobile/board-mackerel.c
@@ -1592,6 +1601,7 @@ static void __init mackerel_init(void)
 #endif
        sh7372_add_device_to_domain(&sh7372_a3sp, &sdhi2_device);
        sh7372_add_device_to_domain(&sh7372_a4r, &ceu_device);
+       sh7372_add_device_to_domain(&sh7372_a4lc, &fake_device);

        hdmi_init_pm_clock();
        sh7372_pm_init();
```

I will demostrate both the case when the device belongs to the PM domain and the case when it does not belong to the PM domain.

Kconfig and Makefile Modifications

```
Index: linux/drivers/misc/Kconfig
===================================================================
--- linux.orig/drivers/misc/Kconfig
+++ linux/drivers/misc/Kconfig
@@ -498,6 +498,13 @@ config MAX8997_MUIC
          Maxim MAX8997 PMIC.
          The MAX8997 MUIC is a USB port accessory detector and switch.

+config MACKEREL_FAKEDEV
+       bool "Mackerel Fake Device Support"
+       depends on MACH_MACKEREL
+       help
+          Enable this if you want to experiment with the demo fake device
+          on the Mackerel board
+
 source "drivers/misc/c2port/Kconfig"
 source "drivers/misc/eeprom/Kconfig"
 source "drivers/misc/cb710/Kconfig"
Index: linux/drivers/misc/Makefile
===================================================================
--- linux.orig/drivers/misc/Makefile
+++ linux/drivers/misc/Makefile
@@ -49,3 +49,4 @@ obj-y += carma/
 obj-$(CONFIG_USB_SWITCH_FSA9480) += fsa9480.o
 obj-$(CONFIG_ALTERA_STAPL) +=altera-stapl/
 obj-$(CONFIG_MAX8997_MUIC) += max8997-muic.o
+obj-$(CONFIG_MACKEREL_FAKEDEV) += fake_device.o
```

Driver Code Part I

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/pm_domain.h>
#include <linux/pm_runtime.h>
#include <linux/slab.h>

struct fake_device_priv {
    bool enabled;
};

static int fake_device_stop(struct device *dev)
{
    dev_info(dev, "%s: stopped\n", __func__);
    return 0;
}

static int fake_device_start(struct device *dev)
{
    dev_info(dev, "%s: started\n", __func__);
    return 0;
}
```

```
static int fake_device_save_state(struct device *dev)
{
    dev_info(dev, "%s: state saved\n", __func__);
    return 0;
}

static int fake_device_restore_state(struct device *dev)
{
    dev_info(dev, "%s: state restored\n", __func__);
    return 0;
}

static int fake_device_runtime_suspend(struct device *dev)
{
    int ret = fake_device_save_state(dev);
    return ret ? : fake_device_stop(dev);
}

static int fake_device_runtime_resume(struct device *dev)
{
    int ret = fake_device_start(dev);
    return ret ? : fake_device_restore_state(dev);
}
```

Driver Code Part II

```
static const struct dev_pm_ops fake_device_pm_ops = {
    .runtime_suspend = fake_device_runtime_suspend,
    .runtime_resume = fake_device_runtime_resume,
};

static bool fake_device_enabled(struct device *dev)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct fake_device_priv *priv = platform_get_drvdata(pdev);

    return priv->enabled;
}

static void fake_device_set_status(struct device *dev, bool enabled)
{
    struct platform_device *pdev = to_platform_device(dev);
    struct fake_device_priv *priv = platform_get_drvdata(pdev);

    if (priv->enabled == enabled)
        return;

    priv->enabled = enabled;
    if (enabled)
        pm_runtime_get_sync(dev);
    else
        pm_runtime_put_sync(dev);
}
```

Driver Code Part III

```
static const char enabled[] = "enabled";
static const char disabled[] = "disabled";

static ssize_t status_show(struct device *dev, struct device_attribute *attr, char *buf)
{
    return sprintf(buf, "%s\n", fake_device_enabled(dev) ? enabled : disabled);
}

static ssize_t status_store(struct device *dev, struct device_attribute *attr,
                            const char *buf, size_t n)
{
    char *cp;
    int len = n;

    cp = memchr(buf, '\n', n);
    if (cp)
        len = cp - buf;

    if (len == sizeof(enabled) - 1 && strncmp(buf, enabled, len) == 0)
        fake_device_set_status(dev, true);
    else if (len == sizeof(disabled) - 1 && strncmp(buf, disabled, len) == 0)
        fake_device_set_status(dev, false);
    else
        return -EINVAL;

    return n;
}
```

Driver Code Part IV

```
static DEVICE_ATTR(status, 0644, status_show, status_store);

static struct attribute *manip_attrs[] = {
    &dev_attr_status.attr,
    NULL,
};

static struct attribute_group manip_attr_group = {
    .name = "manip",
    .attrs = manip_attrs,
};

static int fake_device_remove(struct platform_device *pdev)
{
    struct fake_device_priv *priv = platform_get_drvdata(pdev);

    sysfs_remove_group(&pdev->dev.kobj, &manip_attr_group);
    if (priv->enabled)
        pm_runtime_put_sync(&pdev->dev);

    pm_runtime_disable(&pdev->dev);
    platform_set_drvdata(pdev, NULL);
    kfree(priv);
    return 0;
}
```

Driver Code Part V

```
static int __devinit fake_device_probe(struct platform_device *pdev)
{
    struct sh_mobile_lcdc_info *pdata = pdev->dev.platform_data;
    struct gpd_dev_ops domain_pm_ops = {
        .stop = fake_device_stop,
        .start = fake_device_start,
        .save_state = fake_device_save_state,
        .restore_state = fake_device_restore_state,
    };
    struct fake_device_priv *priv;
    int ret;

    if (!pdata) {
        dev_err(&pdev->dev, "no platform data defined\n");
        return -EINVAL;
    }

    if (strcmp("MY FAKE DEVICE", (char *)pdata))
        return -ENODEV;

    dev_info(&pdev->dev, "Fake device %d found\n", pdev->id);

    priv = kzalloc(sizeof(*priv), GFP_KERNEL);
    if (!priv)
        return -ENOMEM;
```

Driver Code Part VI

```
    platform_set_drvdata(pdev, priv);
    pm_genpd_add_callbacks(&pdev->dev,
                           &domain_pm_ops, NULL);
    pm_genpd_dev_need_restore(&pdev->dev, true);
    pm_runtime_set_suspended(&pdev->dev);
    pm_runtime_enable(&pdev->dev);

    ret = sysfs_create_group(&pdev->dev.kobj,
                             &manip_attr_group);
    if (ret) {
        pm_runtime_disable(&pdev->dev);
        platform_set_drvdata(pdev, NULL);
        kfree(priv);
        return ret;
    }

    return 0;
}
```

```
static struct platform_driver fake_device_driver = {
    .driver = {
        .name   = "fake-device",
        .owner = THIS_MODULE,
        .pm     = &fake_device_pm_ops,
    },
    .probe  = fake_device_probe,
    .remove = fake_device_remove,
};

module_platform_driver(fake_device_driver);

MODULE_DESCRIPTION("Mackerel Fake Device driver");
MODULE_AUTHOR("Rafael J. Wysocki <rjw@sisk.pl>");
MODULE_LICENSE("GPL v2");
```

Thanks!

Thank you for attention!

Thanks!

Thank you for attention!

Special thanks to Renesas Electronics Corp. for covering my travel expenses related to the participation in LinuxCon Japan 2012.