Linux Power!

(From the perspective of a PMIC vendor)

Matti Vaittinen

Jan 10 2023

ROHM Semiconductor

Topics

Goal

What is PMIC

Regulator errors and notifications

Functional-safety helpers in regulator subsystem

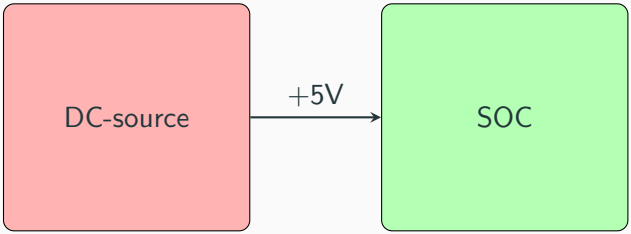
About Me

- Matti Vaittinen
- Kernel/Driver developer at ROHM Semiconductor
- Worked at Nokia BTS projects (networking, clock & sync) 2006 – 2018
- Currently mainly developing/maintaining upstream Linux device drivers for ROHM ICs

What and Why is a PMIC?

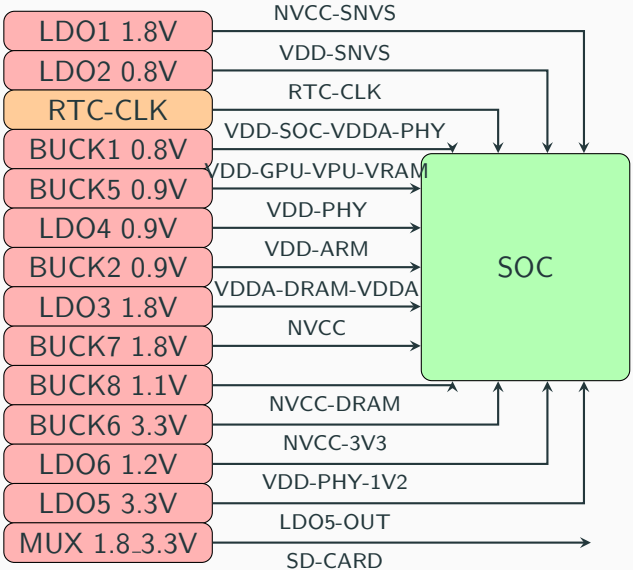
Powering a processor

- Processor and peripherals need power
- Can be as simple as a dummy DC power source with correct voltage



Powering a modern SOC 1/2

Modern SOCs can require multiple specific voltages



Powering a modern SOC 2/2

And specific timings...

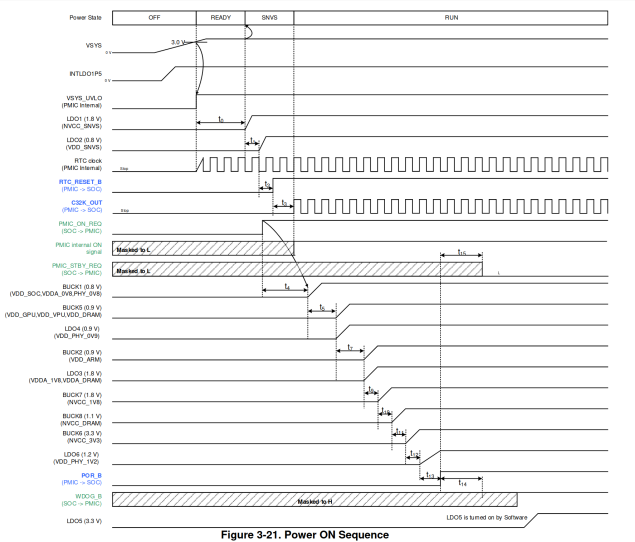


Figure 3-21. Power ON Sequence

More control...

Power savings by:

- Shutting down not needed devices
- Stand-by state(s)
- DVS (Dynamic Voltage Scaling)

Powering-on a system at given time / by an event.

- RTC
- HALL sensor, ...

More functionality

- Battery / charger
- Watchdog
- Functional-safety
  - Voltage monitoring
  - Current monitoring
  - Temperature monitoring

More control...

Power savings by:

- Shutting down not needed devices
- Stand-by state(s)
- DVS (Dynamic Voltage Scaling)

Powering-on a system at given time / by an event.

- RTC
- HALL sensor, ...

More functionality

- Battery / charger
- Watchdog
- Functional-safety
  - Voltage monitoring
  - Current monitoring
  - Temperature monitoring

More control...

Power savings by:

- Shutting down not needed devices
- Stand-by state(s)
- DVS (Dynamic Voltage Scaling)

Powering-on a system at given time / by an event.

- RTC
- HALL sensor, ...

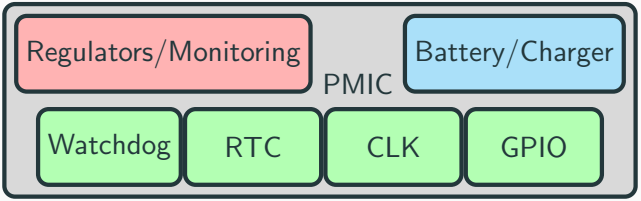
More functionality

- Battery / charger
- Watchdog
- Functional-safety
  - Voltage monitoring
  - Current monitoring
  - Temperature monitoring

PMICs

PMIC - Power Management Integrated Circuit

- Multiple DC sources with specific start-up / shut-down sequence
- Voltage control
- Functional-safety
- Auxiliary blocks to support various needs



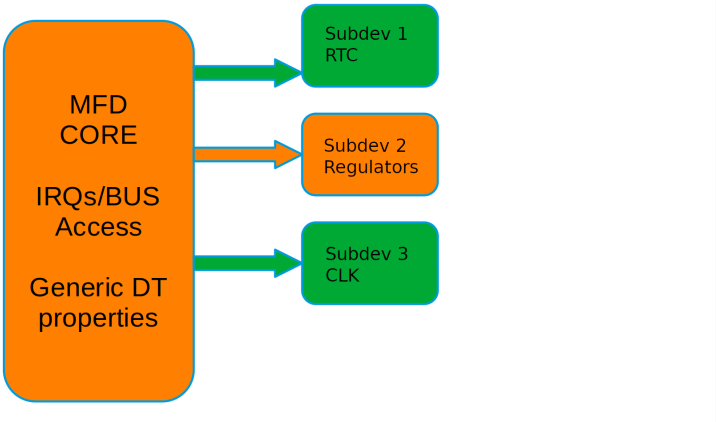
PMIC drivers

Multi Function Devices

Why? (I have 1 reason on mind, may be more)

Often MFD drivers

- **Regulator**
- RTC
- Power supply
- Watchdog
- GPIO
- CLK ...

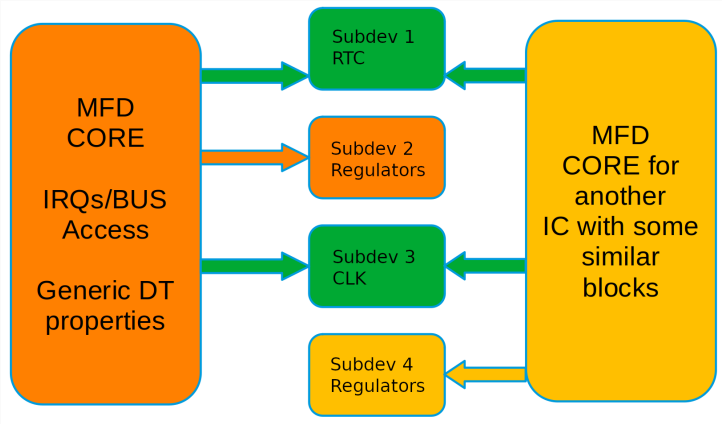


Multi Function Devices

Often MFD drivers

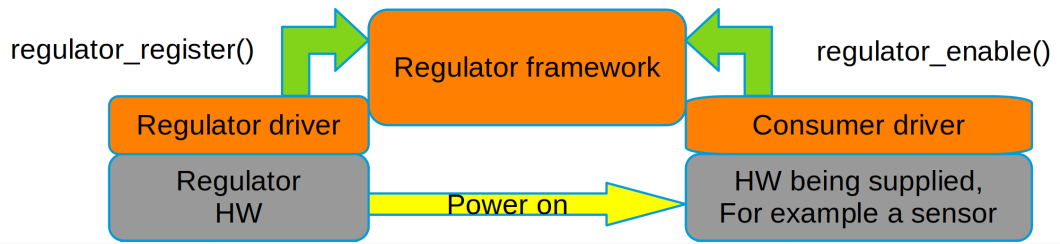
- **Regulator**
- RTC
- Power supply
- Watchdog
- GPIO
- CLK ...

Allows re-use



Regulator (provider) and consumer

- Provider is driver interfacing the hardware. Eg, sits "below" the regulator framework. Between regulator framework and HW
- Consumer is driver who wishes to control the regulator using the regulator framework. Eg, sits "on top of" the regulator framework
- PMIC driver is the provider driver (usually just referred as a regulator driver)



Regulator driver ops

Regulator driver relies on callbacks

Regulator (provider) registers callbacks to regulator framework. Framework handles regulators using these ops.

include/linux/regulator/driver.h

```
struct regulator_ops {
        // snip
        int (*enable) (struct regulator_dev *);
        int (*disable) (struct regulator_dev *);
        int (*is_enabled) (struct regulator_dev *);
        int (*set_voltage_sel) (struct regulator_dev *, unsigned selector);
        int (*get_voltage_sel) (struct regulator_dev *);
        // snip
};
```

Regulator descriptor

include/linux/regulator/driver.h

```
struct regulator_desc {
        /* Plenty of regulator properties */
        /* Also information for the helpers */
        /* Finally the ops */
        const struct regulator_ops *ops;
}

struct regulator_dev *
regulator_register(struct device *dev,
                   const struct regulator_desc *regulator_desc,
                   const struct regulator_config *cfg)
```

Regulator descriptor

include/linux/regulator/driver.h

```
struct regulator_desc {
        /* Plenty of regulator properties */
        /* Also information for the helpers */
        /* Finally the ops */
        const struct regulator_ops *ops;
}

struct regulator_dev *
regulator_register(struct device *dev,
                   const struct regulator_desc *regulator_desc,
                   const struct regulator_config *cfg)
```

Regulator constraints

Regulators can have constraints.

Not to be mixed with limits discussed at the end of the presentation.

- struct regulation_constraints
  include/linux/regulator/machine.h
- hard limits forced by the regulator framework.
- can be given by driver in dynamic init data
- can be given via device-tree
- voltage / current range, prevent disabling, step size ...

Image: *Peggy und Marco Lachmann-Anke, Pixabay*

Monitoring for abnormal conditions

Image: *Gerhard, Pixabay*

Detecting unexpected

Linux has 3 severity categories

The categories - PROTECTION, ERROR, WARNING - inform the hardware state.

PROTECTION

- Unconditional shutdown by HW

ERROR

- Irrecoverable error, system not expected to be usable. Error handling by software.

WARNING - NEW(ish)

- Something is off-limit, system still usable but a recovery action should be taken to prevent escalation to errors

Linux has 3 severity categories

The categories - PROTECTION, ERROR, WARNING - inform the hardware state.

PROTECTION

- Unconditional shutdown by HW

ERROR

- Irrecoverable error, system not expected to be usable. Error handling by software.

WARNING - NEW(ish)

- Something is off-limit, system still usable but a recovery action should be taken to prevent escalation to errors

Detecting unexpected

Linux has 3 severity categories

The categories - PROTECTION, ERROR, WARNING - inform the hardware state.

PROTECTION

- Unconditional shutdown by HW

ERROR

- Irrecoverable error, system not expected to be usable. Error handling by software.

WARNING - NEW(ish)

- Something is off-limit, system still usable but a recovery action should be taken to prevent escalation to errors

Detecting unexpected

Linux has 3 severity categories

The categories - PROTECTION, ERROR, WARNING - inform the hardware state.

PROTECTION

- Unconditional shutdown by HW

ERROR

- Irrecoverable error, system not expected to be usable. Error handling by software.

WARNING - NEW(ish)

- Something is off-limit, system still usable but a recovery action should be taken to prevent escalation to errors

Safety limits, device-tree

Property format:

- regulator-<event >-<severity >-<unit >= value

Over current:

- regulator-oc-protection-microamp
- regulator-oc-error-microamp
- regulator-oc-warn-microamp

Similar for over voltage (ov), under voltage (uv) and temperature (temp)

- 0 =>disable
- 1 =>enable
- other =>limit value

Safety limits, device-tree

Property format:

- regulator-<event >-<severity >-<unit >= value

Over current:

- regulator-oc-protection-microamp
- regulator-oc-error-microamp
- regulator-oc-warn-microamp

Similar for over voltage (ov), under voltage (uv) and temperature (temp)

- 0 =>disable
- 1 =>enable
- other =>limit value

What if hardware does not support given limit?

Image: *Pete Linforth, Pixabay*

Callbacks for configuring the limits

include/linux/regulator/driver.h

```
struct regulator_ops {
        // snip
        int (*set_over_current_protection)(struct regulator_dev *,
                int lim_uA, int severity, bool enable);
        int (*set_over_voltage_protection)(struct regulator_dev *,
                int lim_uV, int severity, bool enable);
        int (*set_under_voltage_protection)(struct regulator_dev *,
                int lim_uV, int severity, bool enable);
        int (*set_thermal_protection)(struct regulator_dev *,
                int lim, int severity, bool enable);
};

struct regulator_desc {};
struct regulator_dev *[devm_]regulator_register(...,
                const struct regulator_desc *regulator_desc, ...);
```

Callbacks for configuring the limits

include/linux/regulator/driver.h

```
struct regulator_ops {
        // snip
        int (*set_over_current_protection)(struct regulator_dev *,
                int lim_uA, int severity, bool enable);
        int (*set_over_voltage_protection)(struct regulator_dev *,
                int lim_uV, int severity, bool enable);
        int (*set_under_voltage_protection)(struct regulator_dev *,
                int lim_uV, int severity, bool enable);
        int (*set_thermal_protection)(struct regulator_dev *,
                int lim, int severity, bool enable);
};

struct regulator_desc {};
struct regulator_dev *[devm_]regulator_register(...,
                const struct regulator_desc *regulator_desc, ...);
```

Callbacks for configuring the limits

include/linux/regulator/driver.h

```
struct regulator_ops {
        // snip
        int (*set_over_current_protection)(struct regulator_dev *,
                int lim_uA, int severity, bool enable);
        int (*set_over_voltage_protection)(struct regulator_dev *,
                int lim_uV, int severity, bool enable);
        int (*set_under_voltage_protection)(struct regulator_dev *,
                int lim_uV, int severity, bool enable);
        int (*set_thermal_protection)(struct regulator_dev *,
                int lim, int severity, bool enable);
};

struct regulator_desc {};
struct regulator_dev *[devm_]regulator_register(...,
                const struct regulator_desc *regulator_desc, ...);
```

Simplified example

drivers/regulator/bd9576-regulator.c

```
static int bd9576_set_ocp(struct regulator_dev *rdev, int lim_uA,
                          int severity, bool enable)
{
        /* Return -EINVAL for unsupported configurations */
        if ((lim_uA && !enable) || (!lim_uA && enable))
                return -EINVAL;
        /* Select the correct register and appropriate register-value conversion
         * for given severity and limit.. */
        if (severity == REGULATOR_SEVERITY_PROT) {
                ...
        } else {
                ...
        }
        /* Write configuration to registers */
        return bd9576_set_limit(range, num_ranges, d->regmap,
                                reg, mask, Vfet);
}
```

Informing the unexpected

Two types of information

- ERRORs
- NOTIFICATIONs

ERROR

- set by provider
- queried (polled) by consumer
- regulator_get_error_flags()

NOTIFICATION

- provider invokes consumer callback (blocking notifier call-chain)
- no polling needed
- in some cases IRQ is held active
- regulator_register_notifier()
- can send also other (non error) events

Informing the unexpected

Two types of information

- ERRORs
- NOTIFICATIONs

ERROR

- set by provider
- queried (polled) by consumer
- regulator_get_error_flags()

NOTIFICATION

- provider invokes consumer callback (blocking notifier call-chain)
- no polling needed
- in some cases IRQ is held active
- regulator_register_notifier()
- can send also other (non error) events

Informing the unexpected

Two types of information

- ERRORs
- NOTIFICATIONs

ERROR

- set by provider
- queried (polled) by consumer
- regulator_get_error_flags()

NOTIFICATION

- provider invokes consumer callback (blocking notifier call-chain)
- no polling needed
- in some cases IRQ is held active
- regulator_register_notifier()
- can send also other (non error) events

Regulator error flags

include/linux/regulator/consumer.h

```
#define REGULATOR_ERROR_UNDER_VOLTAGE
#define REGULATOR_ERROR_OVER_CURRENT
#define REGULATOR_ERROR_REGULATION_OUT
#define REGULATOR_ERROR_FAIL
#define REGULATOR_ERROR_OVER_TEMP
#define REGULATOR_ERROR_UNDER_VOLTAGE_WARN
#define REGULATOR_ERROR_OVER_CURRENT_WARN
#define REGULATOR_ERROR_OVER_VOLTAGE_WARN
#define REGULATOR_ERROR_OVER_TEMP_WARN
```

Regulator notifications

include/linux/regulator/consumer.h

```
#define REGULATOR_EVENT_UNDER_VOLTAGE
#define REGULATOR_EVENT_OVER_CURRENT
#define REGULATOR_EVENT_REGULATION_OUT
#define REGULATOR_EVENT_FAIL
#define REGULATOR_EVENT_OVER_TEMP
...
#define REGULATOR_EVENT_UNDER_VOLTAGE_WARN
#define REGULATOR_EVENT_OVER_CURRENT_WARN
#define REGULATOR_EVENT_OVER_VOLTAGE_WARN
#define REGULATOR_EVENT_OVER_TEMP_WARN
#define REGULATOR_EVENT_WARN_MASK
```

Event IRQ helper

A helper provided for IRQ handling and sending the notification

- Supports keeping IRQ disabled for a period of time
- Supports forcibly shutting down the system if accesing the PMIC fails

```
void *regulator_irq_helper(struct device *dev,
                           const struct regulator_irq_desc *d, int irq,
                           int irq_flags, int common_errs,
                           int *per_rdev_errs, struct regulator_dev **rdev,
                           int rdev_amount);
```

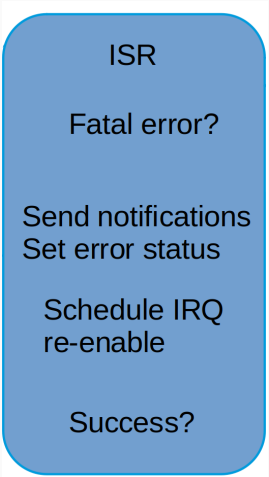
Helper break-out

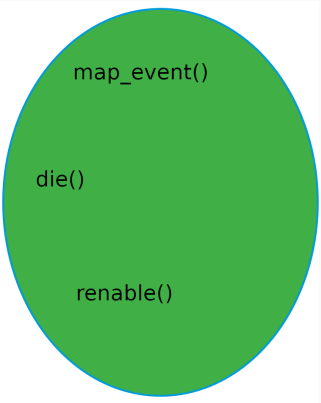
events

helper

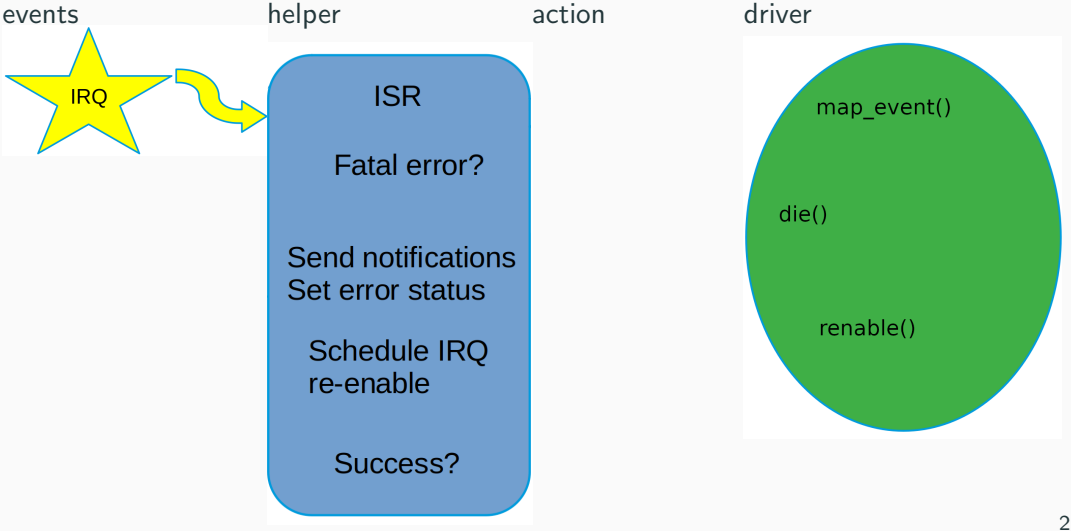
action

driver

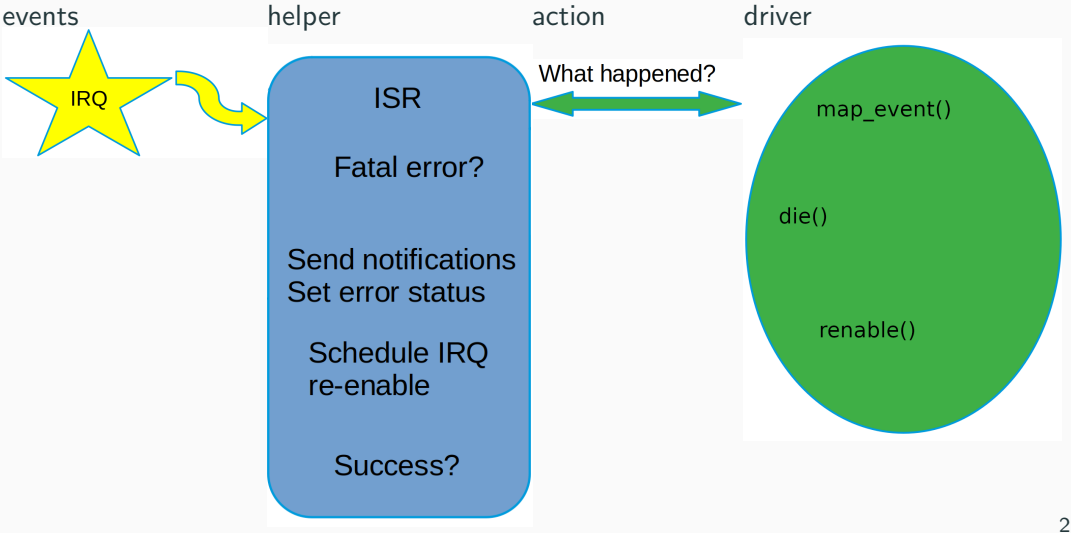




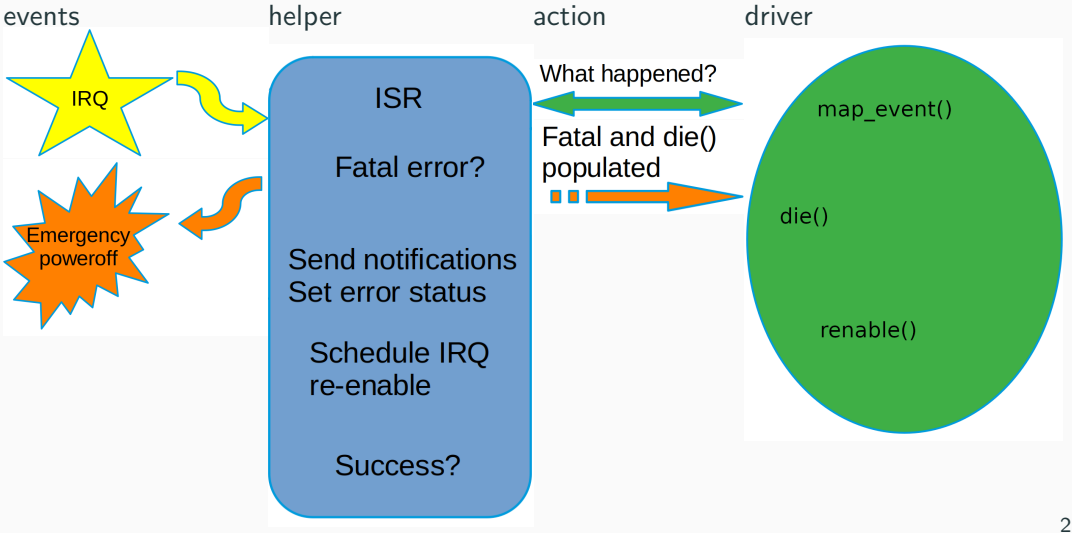
Helper break-out



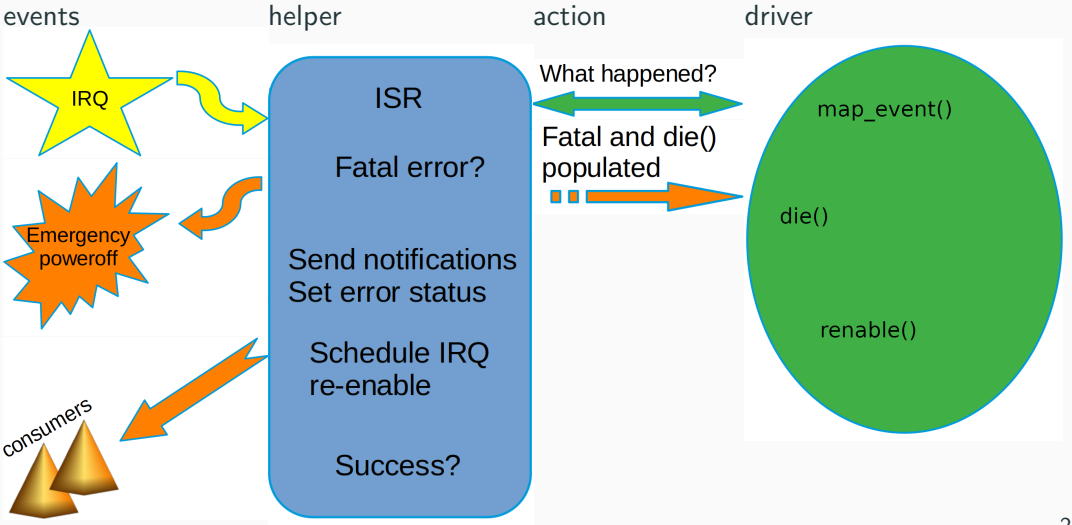
Helper break-out



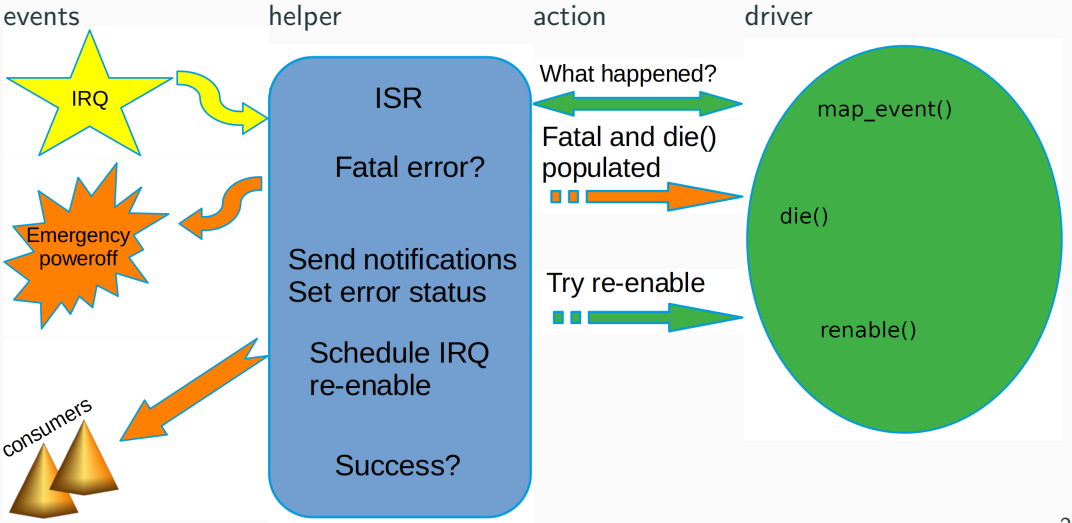
Helper break-out



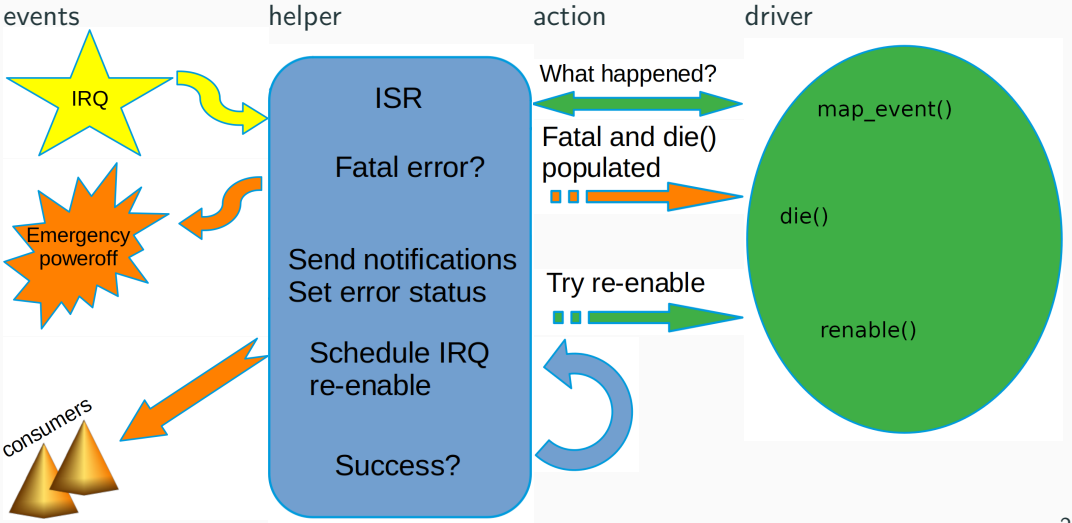
Helper break-out



Helper break-out



Helper break-out



Helper configuration

include/linux/regulator/driver.h

```
struct regulator_irq_desc {
        const char *name;
        int fatal_cnt;
        int reread_ms;
        int irq_off_ms;
        bool skip_off;
        bool high_prio;

        void *data;
        int (*die)(struct regulator_irq_data *rid);
        int (*map_event)(int irq, struct regulator_irq_data *rid,
                         unsigned long *dev_mask);
        int (*renable)(struct regulator_irq_data *rid);
};
```

Helper configuration

include/linux/regulator/driver.h

```
struct regulator_irq_desc {
        const char *name;
        int fatal_cnt;
        int reread_ms;
        int irq_off_ms;
        bool skip_off;
        bool high_prio;

        void *data;
        int (*die)(struct regulator_irq_data *rid);
        int (*map_event)(int irq, struct regulator_irq_data *rid,
                         unsigned long *dev_mask);
        int (*renable)(struct regulator_irq_data *rid);
};
```

Helper Registration

- IRQ information
- Array of regulators
- Events/Errors this IRQ can inform

include/linux/regulator/driver.h

```
void *regulator_irq_helper(struct device *dev,
                           const struct regulator_irq_desc *d,
                           int irq, int irq_flags, int common_errs,
                           int *per_rdev_errs,
                           struct regulator_dev **rdev,
                           int rdev_amount);
```

(or a devm-variant)

Event mapping

include/linux/regulator/driver.h

```
int (*map_event)(int irq, struct regulator_irq_data *rid,
                 unsigned long *dev_mask);

struct regulator_irq_data {
        struct regulator_err_state *states;
        int num_states;
        void *data;
        long opaque;
};

struct regulator_err_state {
        struct regulator_dev *rdev;
        unsigned long notifs;
        unsigned long errors;
        int possible_errs;
};
```

Event mapping

include/linux/regulator/driver.h

```
int (*map_event)(int irq, struct regulator_irq_data *rid,
                 unsigned long *dev_mask);

struct regulator_irq_data {
        struct regulator_err_state *states;
        int num_states;
        void *data;
        long opaque;
};

struct regulator_err_state {
        struct regulator_dev *rdev;
        unsigned long notifs;
        unsigned long errors;
        int possible_errs;
};
```

Event mapping

include/linux/regulator/driver.h

```
int (*map_event)(int irq, struct regulator_irq_data *rid,
                 unsigned long *dev_mask);

struct regulator_irq_data {
        struct regulator_err_state *states;
        int num_states;
        void *data;
        long opaque;
};

struct regulator_err_state {
        struct regulator_dev *rdev;
        unsigned long notifs;
        unsigned long errors;
        int possible_errs;
};
```

Event mapping

include/linux/regulator/driver.h

```
int (*map_event)(int irq, struct regulator_irq_data *rid,
                 unsigned long *dev_mask);

struct regulator_irq_data {
        struct regulator_err_state *states;
        int num_states;
        void *data;
        long opaque;
};

struct regulator_err_state {
        struct regulator_dev *rdev;
        unsigned long notifs;
        unsigned long errors;
        int possible_errs;
};
```

Re-enabling and simple mapping

Helper for simple IRQs
include/linux/regulator/driver.h

```
int regulator_irq_map_event_simple(int irq,
                                   struct regulator_irq_data *rid,
                                   unsigned long *dev_mask)
```

Optional re-enable:

```
int (*renable)(struct regulator_irq_data *rid);
```

Event mapping example part I

drivers/regulator/bd9576-regulator.c

```
static int bd9576_ovd_handler(int irq, struct regulator_irq_data *rid,
                              unsigned long *dev_mask)
{
        ret = regmap_read(d->regmap, BD957X_REG_INT_OVD_STAT, &val);
        if (ret)
                return REGULATOR_FAILED_RETRY;

        rid->opaque = val & OVD_IRQ_VALID_MASK;
        *dev_mask = 0;

        if (!(val & OVD_IRQ_VALID_MASK))
                return 0;
```

Event mapping example part I

drivers/regulator/bd9576-regulator.c

```
static int bd9576_ovd_handler(int irq, struct regulator_irq_data *rid,
                              unsigned long *dev_mask)
{
        ret = regmap_read(d->regmap, BD957X_REG_INT_OVD_STAT, &val);
        if (ret)
                return REGULATOR_FAILED_RETRY;

        rid->opaque = val & OVD_IRQ_VALID_MASK;
        *dev_mask = 0;

        if (!(val & OVD_IRQ_VALID_MASK))
                return 0;
```

Event mapping example part II

```
        *dev_mask = val & BD9576_xVD_IRQ_MASK_VOUT1TO4;

        for_each_set_bit(i, dev_mask, 4) {
                stat  = &rid->states[i];

                stat->notifs    = rdata->ovd_notif;
                stat->errors    = rdata->ovd_err;
        }

        return 0;
}
```

Helper registration 1/3

Fill the helper configuration

drivers/regulator/bd9576-regulator.c

```
static const struct regulator_irq_desc bd9576_notif_ovd = {
        .name = "bd9576-ovd",
        .irq_off_ms = 1000,
        .map_event = bd9576_ovd_handler,
        .renable = bd9576_ovd_renable,
        .data = &bd957x_regulators,
};
```

Helper registration 2/3

Create an array of regulators this IRQ may concern

drivers/regulator/bd9576-regulator.c

```
struct regulator_dev *ovd_devs[BD9576_NUM_OVD_REGULATORS];

for (i = 0; i < num_rdev; i++) {
        struct bd957x_regulator_data *r = &ic_data->regulator_data[i];
        const struct regulator_desc *desc = &r->desc;

        r->rdev = devm_regulator_register(&pdev->dev, desc, &config);

        rdevs[i] = r->rdev;
        if (i < BD957X_VOUTS1)
                ovd_devs[i] = r->rdev;
}
```

Helper registration 3/3

Fill possible errors this IRQ may indicate and register the helper

drivers/regulator/bd9576-regulator.c

```
int ovd_errs = REGULATOR_ERROR_OVER_VOLTAGE_WARN |
        REGULATOR_ERROR_REGULATION_OUT;

ret = devm_regulator_irq_helper(&pdev->dev, &bd9576_notif_ovd,
                                irq, 0, ovd_errs, NULL,
                                &ovd_devs[0],
                                BD9576_NUM_OVD_REGULATORS);
```

Wrap it up

Summary

- Powering up a modern SOC is not simple
- PMIC is an IC trying to integrate powering related features into single chip
- Many PMICs include functional-safety features
- There is some existing support for indicating abnormal events

Questions?

Thank You for listening!
(or time to wake up) :)

Extras

How to handle notifications?

```
typedef int (*notifier_fn_t)(struct notifier_block *nb,
                        unsigned long action, void *data);

struct notifier_block {
        notifier_fn_t notifier_call;
        struct notifier_block __rcu *next;
        int priority;
};

/**
 * regulator_register_notifier – register regulator event notifier
 * @regulator: regulator source
 * @nb: notifier block
 *
 * Register notifier block to receive regulator events.
 */
int regulator_register_notifier(struct regulator *regulator,
                                struct notifier_block *nb)
```