Linux Power!
(From the perspective of a PMIC vendor)
Matti Vaittinen
Jan 10 2023
ROHM Semiconductors

1. Breath, smile, look at them
2. Eyecontact. They are your friends. On your side
3. Breath
4. Welcome. I am not excited to be here. I am terrified
5. I am afraid of social situations and don't like making myself known
6. So, of course I am climbing up on the stage to give a talk, right? ... How stupid one can be?
7. Well, as I am here, I will talk a little bit about powring a SoC on a Linux based system

Linux Power!

(From the perspective of a PMIC vendor)

Matti Vaittinen
Jan 10 2023
ROHM Semiconductors

Topics
Goal
What is PMIC
Regulator errors and notifications
Functional-safety helpers in regulator subsystem
What and Why is a PMIC?
PMIC drivers
MFD and subdevices
Regulators
Monitoring for abnormal conditions
Severity levels and limit values
Regulator errors and notifications
Helpers and examples
Wrap it up

1. Shallow overview on what is PMIC and why it is needed
2. Linux oriented Short glance of drivers can be needed for a PMIC
3. functional safety and reporting hw issues

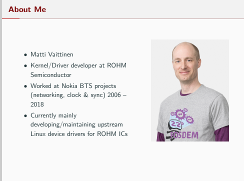
Topics

Goal
What is PMIC
Regulator errors and notifications
Functional-safety helpers in regulator subsystem

What and Why is a PMIC?
PMIC drivers
- MFD and subdevices
- Regulators

Monitoring for abnormal conditions
- Severity levels and limit values
- Regulator errors and notifications
- Helpers and examples

Wrap it up



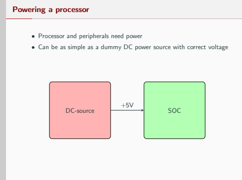
1. This is the formal me
2. Linux developer who started working with Linux at 2005
3. Today I work at ROHM Semiconductors
4. HW vendor, no forest from the trees.
5. Anyone with insight on how notifiers are or could be used - please explain!

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

Following section aims to give an idea about What a PMIC is and Why PMICs are needed?

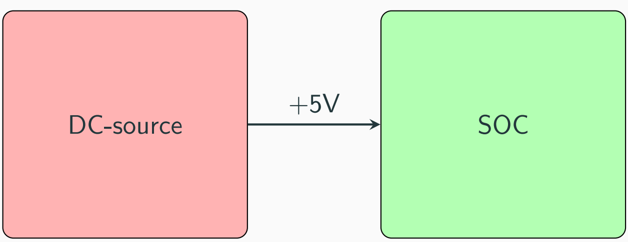
What and Why is a PMIC?

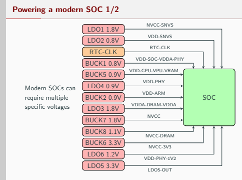


could be this simple. Just passive source

Powering a processor

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

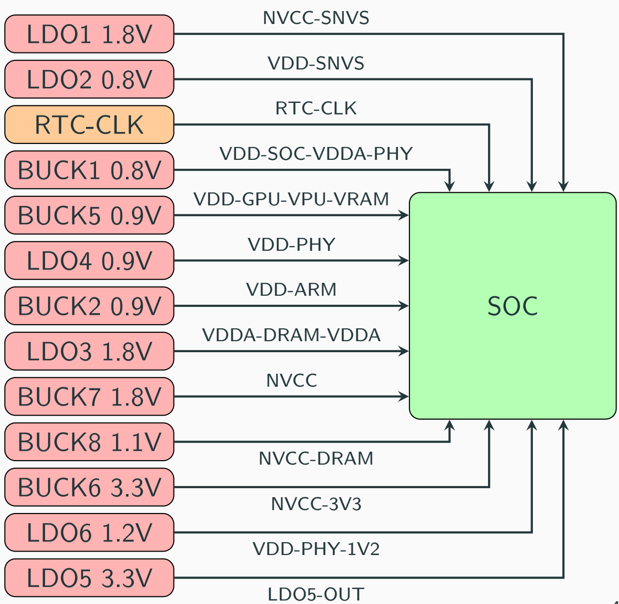


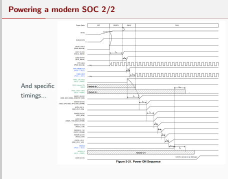


1. almost a real SOC.
2. pic omits state GPIOs

Powering a modern SOC 1/2

Modern SOCs can require multiple specific voltages





1. TIMINGs - explain state transitions
2. from PMIC spec
3. shows also internal changes

Powering a modern SOC 2/2

And specific timings...

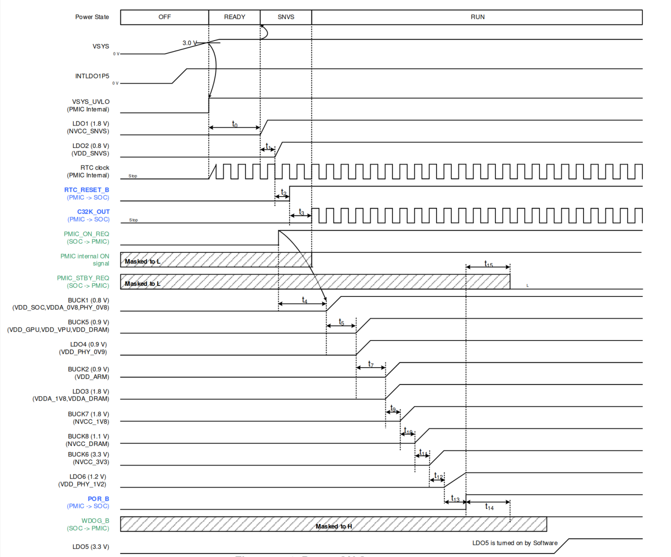


Figure 3-21. Power ON Sequence

More control...

Power savings by:

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

1. Importance of power saving increases
2. Toggling outputs on/off
3. Changing voltages
4. Predefined states changed by GPIO (avoid I2C shut-down races I2C depends on some other block - other can't be shut down?)

More control...

Power savings by:

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

Automated power on
Powering-on a system at given time...
• RTC
...Or by an event
• HALL sensor, ...

1. Monitor turn-on input when SOC is shut down
2. For example RTC / HALL sensor (lid)

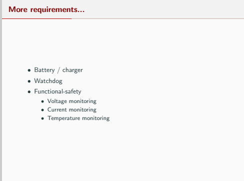
Automated power on

Powering-on a system at given time...

- RTC

...Or by an event

- HALL sensor, ...

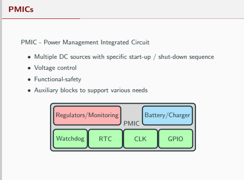


More needs

1. battery powered devices everywhere =¿ charging logic
2. Watchdog cut power - can be external or in power-supply
3. montor abnormal events (temp, over voltage, ocp)

More requirements...

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

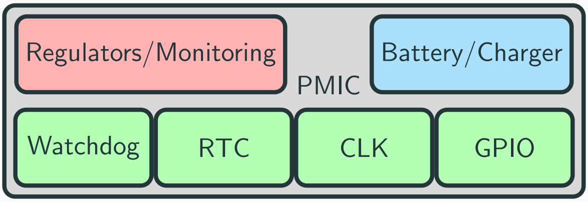


1. PMICs created to support previous use-cases
2. Often very SOC specific
3. Still also generic ones done - very customizable (amount of outputs, voltages, sequences)

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



1. What some typical PMIC drivers look like
2. MFD
3. Regulators

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 ...

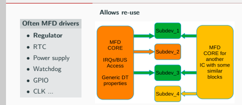
1. MFD "core" driver (Lee says there is no such "thing" as MFD)
2. core driver often provides bus access and IRQ controller code
3. core driver is created as any "standard driver" on that bus
4. Sub devices are (seemingly independent) platform devices
5. mfd-cell array describes subdevices (drivers)
6. mfd registration instantiates subdevices and runs driver probes
7. MODALIAS for module loading
8. Ask why MFD? (spoiler, re-use)

Multi Function Devices

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

Often MFD drivers

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



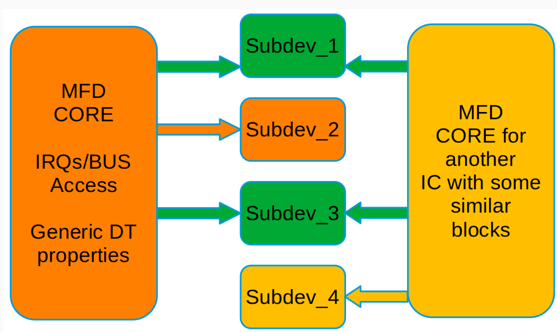
1. Many devices re-use digital blocks from previous generations while adding something new
2. MFD sub-devices can be re-used and new drivers written only for new blocks (ideally)

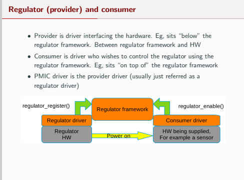
Multi Function Devices

Often MFD drivers

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

Allows re-use

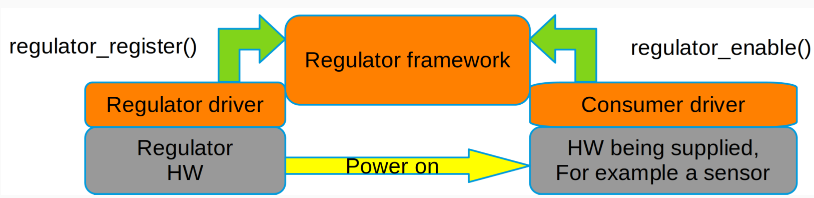




1. regulator framework sits between hardware driver and regulator user
2. provides control/information interface to consumer drivers - regulator API
3. hardware driver interfaces PMIC and translates regulator framework requests to register reads/writes
4. example, regulator consumer can be sensor driver, enabling sensor power when sensor is needed
5. sensor requests and enables the regulator via regulator API
6. regulator framework calls correct callbacks from regulator driver

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)



1. rest of the show explains how regulator framework can be used to deliver errors. From PMIC vendor perspective - sorry

Monitoring for abnormal conditions

Detecting undexpected

Linux has 3 severity categories

- 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 undexpected

Linux has 3 severity categories

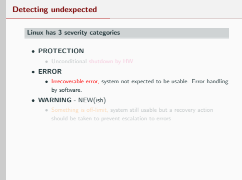
- **PROTECTION**
  - Unconditional shutdown by HW
- **ERROR**
  - Irrecoverable error, system not expected to be usable. Error handling by software.
- **WARNING** - NEW(ish)
  - Something is off-limit, system still usable but a recovery action should be taken to prevent escalation to errors

1. PROTECTION example, severe overvoltage - PMIC shuts down all or offending outputs
2. Could be also temperature-error or over-current

Detecting undexpected

Linux has 3 severity categories

- **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

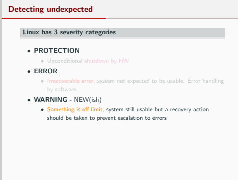


1. Some PMICs may not automatically shut down outputs - SW should do it
2. ERROR still indicate fatal issues - HW not working

Detecting undexpected

Linux has 3 severity categories

- **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

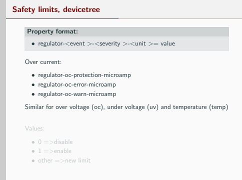


1. WARNING included in kernel 5.14
2. Intended to be used for invoking corrective action(s)
3. Has been requested from us - there probably are use-cases
4. I have no insight as to what they could be - very interested in learning any concrete examples - ask from audience

Detecting undexpected

Linux has 3 severity categories

- **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



1. Often board specific - Can be provided via device-tree
2. property format from box
3. pause

Safety limits, devicetree

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 (oc), under voltage (uv) and temperature (temp)

Values:

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

Safety limits, devicetree
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 (oc), under voltage (uv) and temperature (temp)
Values:
- 0 =>disable
- 1 =>enable
- other =>new limit

1. value sets new limit
2. 1 / 0 special values - they are used to indicate enable / disable
3. pause

Safety limits, devicetree

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 (oc), under voltage (uv) and temperature (temp)

Values:

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

Safety limits, devicetree

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 (oc), under voltage (uv) and temperature (temp)

Values:

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

What if hardware does not support given limit?

1. I have no good answer.
2. What happens if silently ignored?
3. What happens if regulator registration fails?
4. If limit is silently ignored - problems for example when changing a component and new driver does not support limits
5. If registration fails, system may not boot, boot without display etc.
6. Currently just log a warning
7. In general, is there a way to mark NOT supported properties to binding docs? Would it help? Including ALL regulator bindings does hide this at avalidation
8. pause
9. I know you are eager to see the code :) So, let's look at some

Safety limits, devicetree

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 (oc), under voltage (uv) and temperature (temp)

Values:

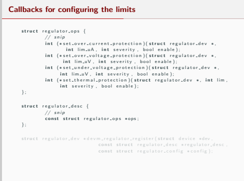
- 0 =>disable
- 1 =>enable
- other =>new limit

What if hardware does not support given limit?

1. Callbacks for over-current, over- and under-voltage, over temperature
2. Arguments include the limit, severity (PROT, ERR, WARN) and enable/disable

Callbacks for configuring the limits

```
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 {
        // snip
        const struct regulator_ops *ops;
};

struct regulator_dev *devm_regulator_register(struct device *dev,
                          const struct regulator_desc *regulator_desc,
                          const struct regulator_config *config);
```



1. Callbacks are amongst the other regulator ops which is pointed from the regulator description

Callbacks for configuring the limits

```
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 {
        // snip
        const struct regulator_ops *ops;
};

struct regulator_dev *devm_regulator_register(struct device *dev,
                          const struct regulator_desc *regulator_desc,
                          const struct regulator_config *config);
```

Callbacks for configuring the limits

1. The description with ops is passed to regulator registration

Callbacks for configuring the limits

```
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 {
        // snip
        const struct regulator_ops *ops;
};

struct regulator_dev *devm_regulator_register(struct device *dev,
                                const struct regulator_desc *regulator_desc,
                                const struct regulator_config *config);
```

1. Example of over-current protection example using pieces of ROHM BD9576 driver as example
2. Regulator framework parses dt properties and invokes callback with values found from DT
3. Driver typically checks for parameter sanity/support
4. Driver decides the register and values to write based on severity and limit
5. for example BD9576 has different register and limit ranges for warning and protection
6. From the PMIC operation POV there is no difference between ERROR and WARNING
7. Finally if limits were sane, the driver updates new limits to registers and returns Ok

Simplified example

```
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
  - sent by provider (usually) from interrupt
  - no polling needed
  - regulator_register_notifier()
  - can send also other events

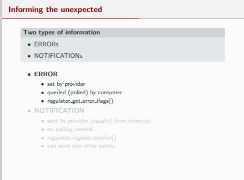
1. So, when an abnormal event is detected by HW (something exeeds the limit) we need to inform consumers
2. The regulator framework can use ERRORs or NOTIFICATIONs
3. Errors visible via sysfs

Informing the unexpected

Two types of information

- ERRORs
- NOTIFICATIONs

- **ERROR**
  - set by provider
  - queried (polled) by consumer
  - regulator_get_error_flags()
- **NOTIFICATION**
  - sent by provider (usually) from interrupt
  - no polling needed
  - regulator_register_notifier()
  - can send also other events



1. Errors are simple status which is stored in framework
2. Errors need to be polled by consumers
3. Polling is not always preferred

Informing the unexpected

Two types of information

- ERRORs
- NOTIFICATIONs

- **ERROR**
  - set by provider
  - queried (polled) by consumer
  - regulator_get_error_flags()
- **NOTIFICATION**
  - sent by provider (usually) from interrupt
  - no polling needed
  - regulator_register_notifier()
  - can send also other events

1. Notifications are sent by HW when problem occurs
2. No polling needed - except to know when condition is back to normal
3. Not all notifications are errors...

Informing the unexpected

Two types of information

- ERRORs
- NOTIFICATIONs

- **ERROR**
  - set by provider
  - queried (polled) by consumer
  - regulator_get_error_flags()
- **NOTIFICATION**
  - sent by provider (usually) from interrupt
  - no polling needed
  - regulator_register_notifier()
  - can send also other events

Regulator error flags

#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

include/linux/regulator/consumer.h

1. The available ERROR definitions
2. I've used REGULATION-OUT for over-voltage
3. Not sure where REGULATOR-ERROR-FAIL is used

Regulator error flags

```
#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
```

include/linux/regulator/consumer.h

Regulator notifications
#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
include/linux/regulator/consumer.h

1. The available EVENT definitions
2. Note, not all of the notificatin types listed as some are not failures
3. Last one (WARN-MASK) is not event but a mask for consumers to allow checking if notification was a warning

Regulator notifications

```
#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
```

include/linux/regulator/consumer.h

Notifications

Usually IRQ backed

1. PMIC detects error and generates IRQ
2. IRQ handler sends notification
3. Regulator consumer action is executed

In some (many) cases IRQ is held active for whole duration of error

- Maybe because these IRQs are considered as a last thing?
- Maybe because there is need to ensure IRQ is not missed?
- Does not play well with all systems

1. Usually an IRQ is generated by HW when error is observed
2. Regulator notifications use blocking call-chain, notifiers should fire from process context
3. Often the IRQ is held active for the duration of a problem
4. Need special handling to avoid IRQ storm which would not exactly help mitigating issues
5. The event executes the action registered by the consumer

Notifications

Usually IRQ backed

1. PMIC detects error and generates IRQ
2. IRQ handler sends notification
3. Regulator consumer action is executed

In some (many) cases IRQ is held active for whole duration of error

- Maybe because these IRQs are considered as a last thing?
- Maybe because there is need to ensure IRQ is not missed?
- Does not play well with all systems

Notifications
Usually IRQ backed
1. PMIC detects error and generates IRQ
2. IRQ handler sends notification
3. Regulator consumer action is executed
In some (many) cases IRQ is held active for whole duration of error
• Maybe because these IRQs are considered as a last thing?
• Maybe because there is need to ensure IRQ is not missed?
• Does not play well with all systems

1. Usually an IRQ is generated by HW when error is observed
2. Regulator notifications use blocking call-chain, notifiers should fire from process context
3. Often the IRQ is held active for the duration of a problem
4. Need special handling to avoid IRQ storm which would not exactly help mitigating issues
5. The event executes the action registered by the consumer

Notifications

Usually IRQ backed

1. PMIC detects error and generates IRQ
2. IRQ handler sends notification
3. Regulator consumer action is executed

In some (many) cases IRQ is held active for whole duration of error

- Maybe because these IRQs are considered as a last thing?
- Maybe because there is need to ensure IRQ is not missed?
- Does not play well with all systems

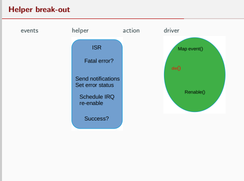
1. helper provided
2. read the box
3. helper is registered using below API - we will see it in more details later

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

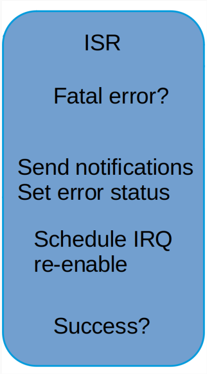
```
void *regulator_irq_helper(struct device *dev,
                          const struct regulator_irq_desc *d, int irq,
                          int irq_flags, int common_errs,
                          int *per_rdev_errs, struct regulator_dev **rdev,
                          int rdev_amount);
```

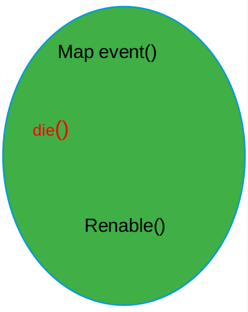


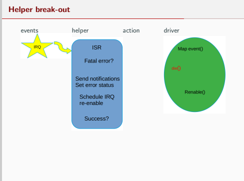
1. But first, let's go through the helper operation
2. Box on the left represent functionality in helper
3. ellipse represent functionality implemented in driver
4. Let's walk through the notification process

Helper break-out

events helper action driver

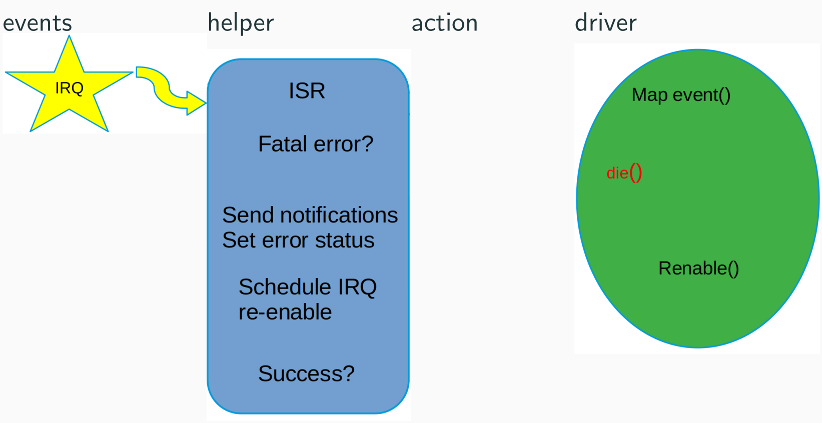


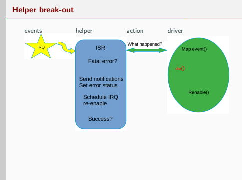




1. abnormal event is detected by HW

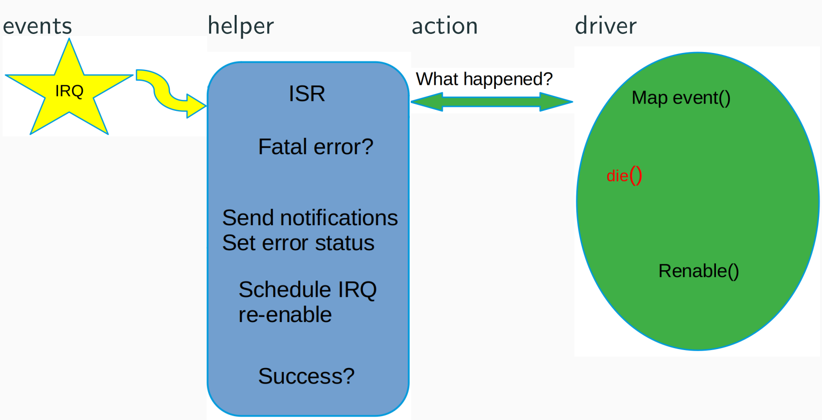
Helper break-out

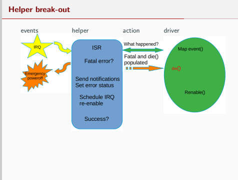




1. helper runs ISR and calls map_event() from driver to know what happened

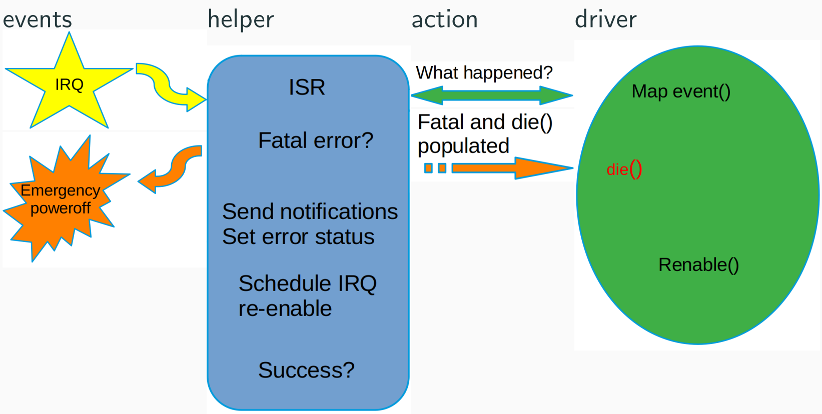
Helper break-out

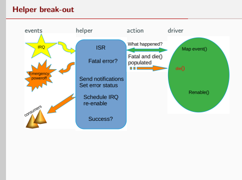




1. map_event() returns information about fatal access failure
2. If error was fatal, helper will call die() callback from driver or try execute emergency power-off
3. The die() callback can be used to provide emergency operation like turning off the failing power

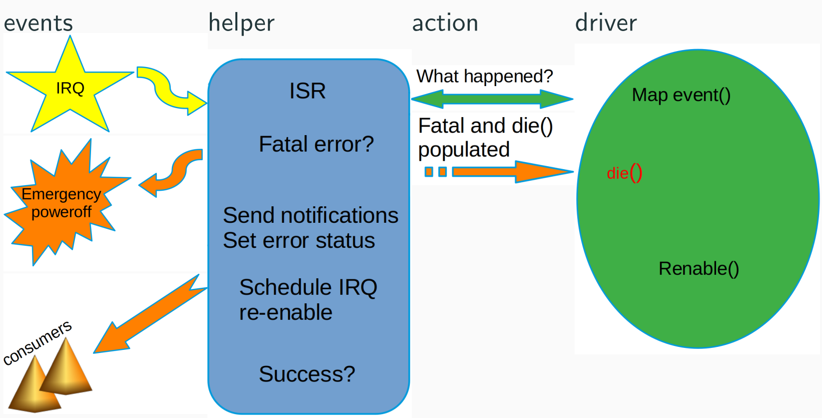
Helper break-out

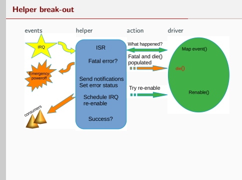




1. Helper will send the notifications to consumers and update the errpr flags
2. Helper will schedule the IRQ re-enable

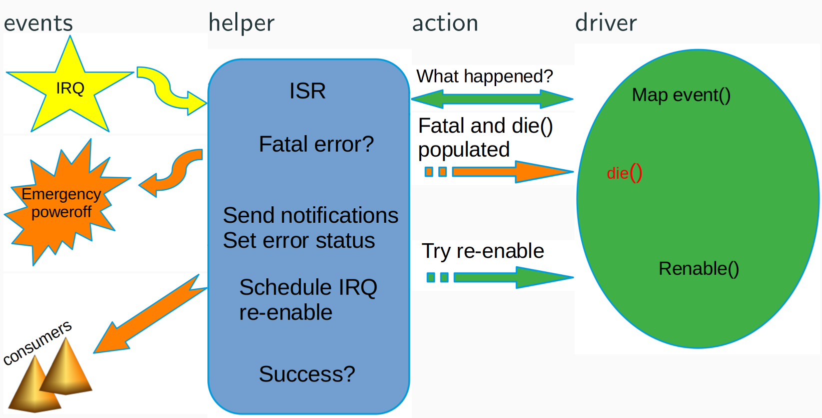
Helper break-out

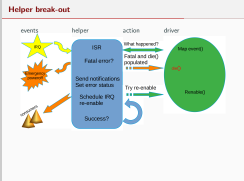




1. When it is time to re-enable the IRQ, helper will ask driver if problem is over and IRQ can be re-enabled

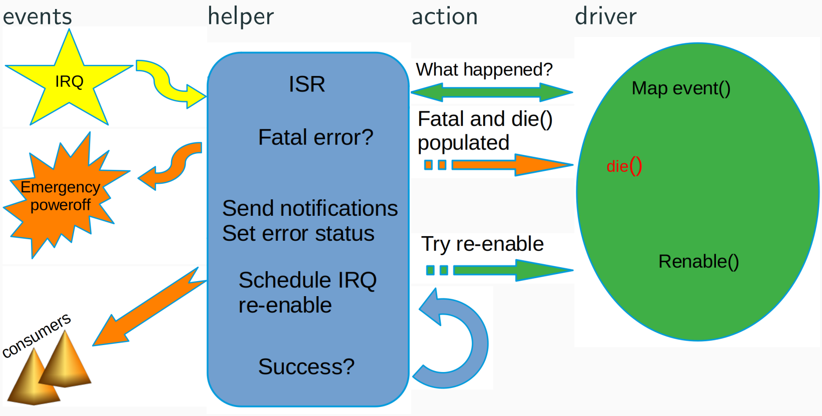
Helper break-out





1. If IRQ can be re-enabled, the helper unmasks it. If not, the re-enable is scheduled again

Helper break-out

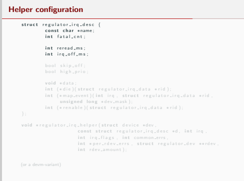


1. Let's get back to helper registration.
2. Helper config information is given in struct regulator_irq_desc
3. fatal_cnt can be used to mark failures to handle event as fatal. Eg. after checking this many times for whether the condition is resolved (attempts to re-enable), die() is called or emergency power-off is executed

Helper configuration

```
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);
};

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)



1. reread_ms is time to wait after failed event mapping until retrying
2. irq_off_ms is time to keep IRQ disabled (before calling renable() if provided

Helper configuration

```
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);
};

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)

1. skip_off and high_prio are sometihng that were used in existing event implementations
2. skip_off can be specified to not handle IRQs for regulators that are disabled
3. high_prio can be set to use high_priority work-queue for trying to re-enable the IRQ

Helper configuration

```
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);
};

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)

Helper configuration

```
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);
};

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)

1. data is pointer which can be used to store data needed in call-backs - like the driver private data
2. Callbacks for die(), mapping event and seeing if IRQ can be re-enabled. We will see the callbacks later

Helper configuration

```
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);
};

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);
```

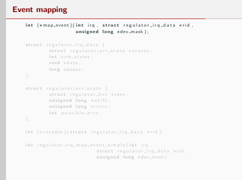
(or a devm-variant)

1. On top of this on the helper registration we provide
2. device pointer and IRQ information
3. errors that are common to all regulator devices this IRQ can be indicating errors for
4. errors specific to only some of the regulator devices
5. and finally an array of the regulator devices for which this IRQ can indicate problems

Helper configuration

```
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);
};

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)



1. in order to find the reason for the IRQ, the helper is invoking a driver callback map_event()
2. The driver should fill in what regulators were sending notification and what events/errors were detected

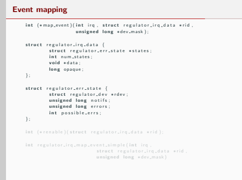
Event mapping

```
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;
};

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

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

1. struct regulator_irq_data is used for this
2. contains array of regulator states for all regulators which can be informing events via this IRQ
3. number of the regulators
4. data pointer as was passed at helper registration
5. opaque integer. Mainly for delivering information about detected events to renable. Usually status register value which can be compared to see if situation has changed

Event mapping

```
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;
};

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

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



1. The status for each regulator is filled in regulator_err_state struct
2. pointer to regulator device which state this struct is informing
3. notifs is the regulator notification flags
4. errors is the regulator error flags
5. possible_errs should contain the errors this IRQ can be informing. Used to clear error statuses when re-enabling the IRQ

Event mapping

```
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;
};

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

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

Event mapping

```
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;
};

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

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


Event mapping

```
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;
};

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

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

Event mapping

Event mapping

```
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;
};

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

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

Event mapping example

Event mapping example

```
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;

        *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;
}
```

Event mapping example

```
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;

        *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;
}
```

Event mapping example

```
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;

        *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 1/2

Fill the helper configuration

```
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 1/2
Fill the helper configuration
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 1/2

Fill the helper configuration

```
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 2/2

Helper registration 2/2

Create an array of regulators this IRQ may concern

```
struct regulator_dev *rdevs[BD9576_NUM_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;
}
```

Fill possible errors this IRQ may indicate and register the helper

```
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);
```

Helper registration 2/2

Create an array of regulators this IRQ may concern

```
struct regulator_dev *rdevs[BD9576_NUM_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;
}
```

Fill possible errors this IRQ may indicate and register the helper

```
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);
```

Helper registration 2/2

Helper registration 2/2

Create an array of regulators this IRQ may concern

```
struct regulator_dev *rdevs[BD9576_NUM_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;
}
```

Fill possible errors this IRQ may indicate and register the helper

```
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);
```

Helper registration 2/2

Create an array of regulators this IRQ may concern

```
struct regulator_dev *rdevs[BD9576_NUM_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;
}
```

Fill possible errors this IRQ may indicate and register the helper

```
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

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

No answers guaranteed
Questions?

No answers guaranteed

Questions?

No answers guaranteed

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

No answers guaranteed

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)

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)
```