Linux Power

Powering a processor

- Processor and peripherals need power
- Can be as simple as a dummy DC power source with correct voltage
- Modern SOCs can have specific requirements
  - Voltages
  - Timings

    => Start-up/shut-down sequence for power sources

More control…

- Power savings by:
  - Shutting down not needed devices
  - Stand-by state(s)
  - DVS (Dynamic Voltage Scaling)
- Powering on system at given time...
  - RTC
- ...Or by event
  - HALL sensor, ...

... more ...

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

PMICs

- PMIC – Powe Management Integrated Circuit
  - Multiple DC sources with specific start-up/shut-down sequence + DVS
  - Auxiliary blocks. (RTC, WDG, Charger, GPIO, CLK, …)
- Often MFD drivers. (Note, allow re-use)
  - **Regulator**, RTC, Power supply, Watchdog, GPIO, CLK ...

Regulator provider / 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.

Detecting unexpected

- PMIC allows configuring safety-limits
  - Severity - Protection, Error, Warning
  - Protection => Unconditional shutdown by HW.
  - Error => Irrecoverable error, system not expected to be usable. Error handling by software.
  - Warning => Something is off-limit, system still usable but a recovery action should be taken to prevent escalation to errors.

Safety limits, devicetree

- Property format regulator-<event>-<severity>-<unit>

  Examples:

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

  regulator-ov-protection-microvolt, regulator-ov-error-microvolt, ...

- Value 0 => disable, 1 => enable (with existing limit)
- others new limit
- What if hardware does not support given limit?

Callbacks for configuring 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)
```

Example

```
static int bd9576_set_ocp(struct regulator_dev *rdev, int lim_uA, int severity,
                          bool enable)
{
      // snip

        /* 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) {
                // snip
        } else {
                // snip
        }

        /* Write configuration to registers */
        return bd9576_set_limit(range, num_ranges, d->regmap,
                                reg, mask, Vfet);
}
```

Informing the unexpected

- Regulator framework supports error (status) and notifications (events)
- Not all notifications are errors/warnings.
- Errors can be set by providers, queried by consumers
  - Polling
- Events can be sent by providers, subscribed by consumers
  - Notifying

Errors

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
```

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

Notifications

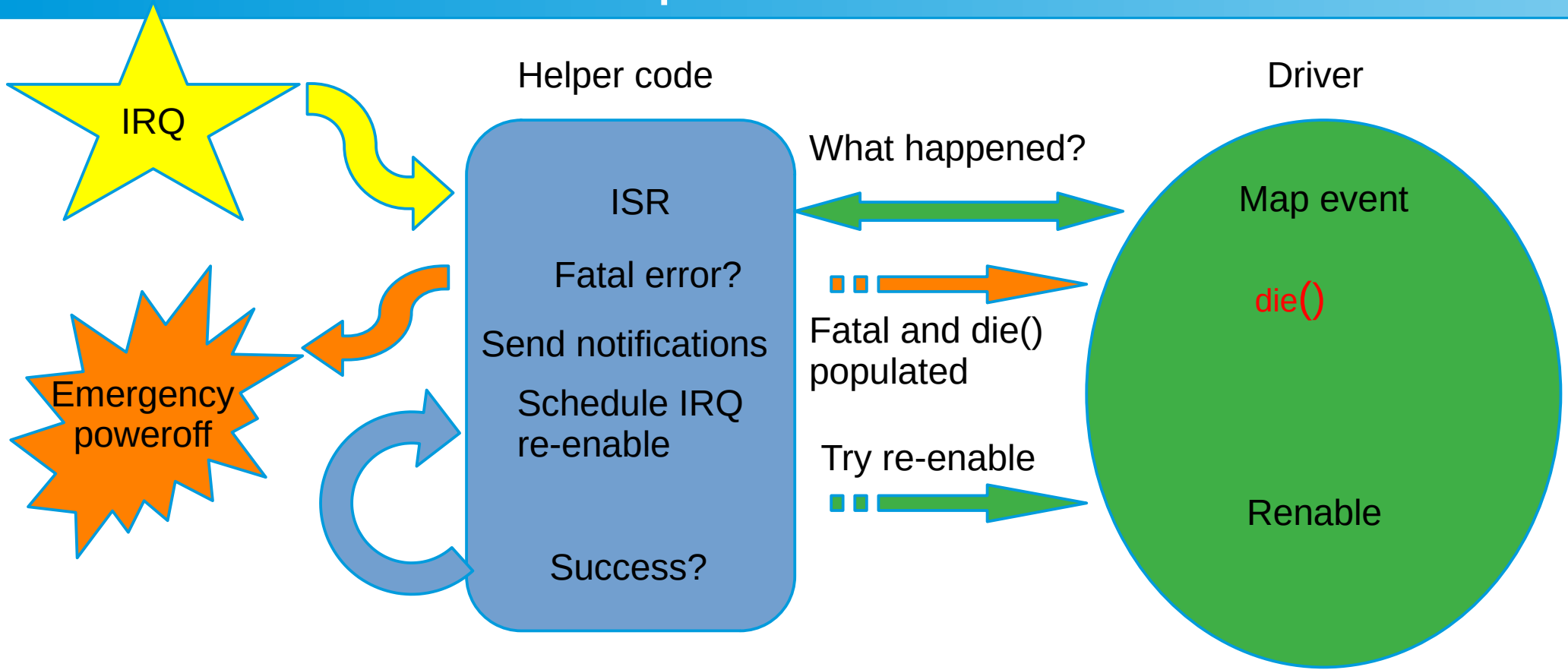
- Usually IRQ backed
  - PMIC detect error => IRQ => notification => consumer action(?)
- In many cases IRQ is held active for whole duration of error
  - Maybe because IRQs are considered as last thing to do(?)
  - Maybe because there is a need to ensure these IRQs are not missed(?)
  - Does not play well with all systems

Event IRQ helper

- Helper provided for IRQ handling and sending the notification.
  - Supports keeping IRQ disabled for period of time
  - Supports forcibly shutting down the system if accessing PMIC fails during error handling

```
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 overview



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 devm_-variant)
```

Event mapping data

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


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


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

```
sstatic 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;
      }

      /* Clear the sub-IRQ status */
      regmap_write(d->regmap, BD957X_REG_INT_OVD_STAT,
                  OVD_IRQ_VALID_MASK & val);

      return 0;
}
```

Example helper registration

```
struct regulator_dev *rdevs[BD9576_NUM_REGULATORS];

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

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

// snip

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);
```