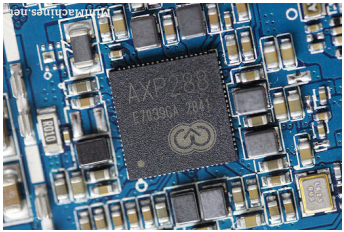
Power Management Integrated Circuits: Keep the power in your hands



Quentin Schulz
Bootlin
quentin.schulz@bootlin.com

Quentin Schulz

- Quentin Schulz
- Embedded Linux and kernel engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Strong open-source focus
  - Linux kernel contributors, ARM SoC support, kernel maintainers
  - Worked on drivers for AXP20X/AXP22X PMICs,

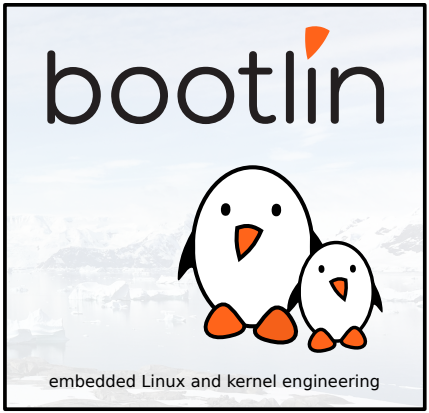
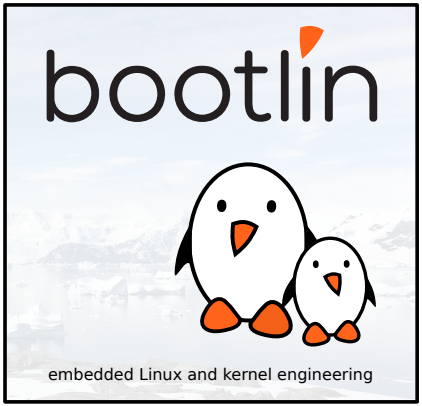


Table of contents

What's a PMIC?

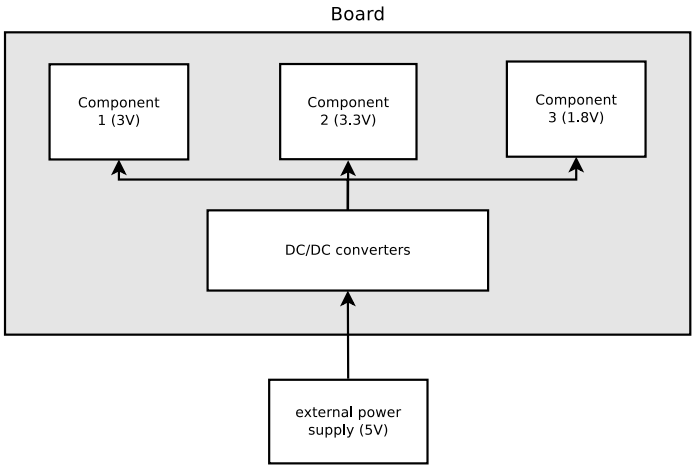
Quentin Schulz
quentin.schulz@bootlin.com



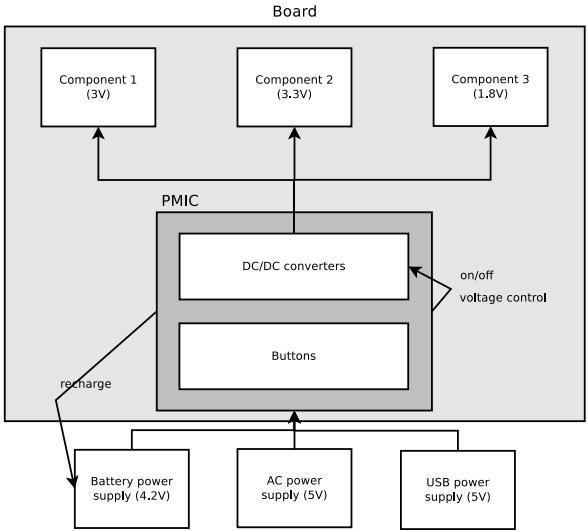
What's a PMIC?

- PMIC = Power Management Integrated Circuit,
- handles the power sequence of the board,
- supplies power to the different components inside the board,
- protects the board from unsupported overvoltage and undervoltage,
- might handle different external power supplies,
- can provide other misc features (GPIO, ADC, ...),
- is usually software-controllable (often as an i^2c device),
- is not mandatory (e.g. Raspberry Pi and Orange Pi),

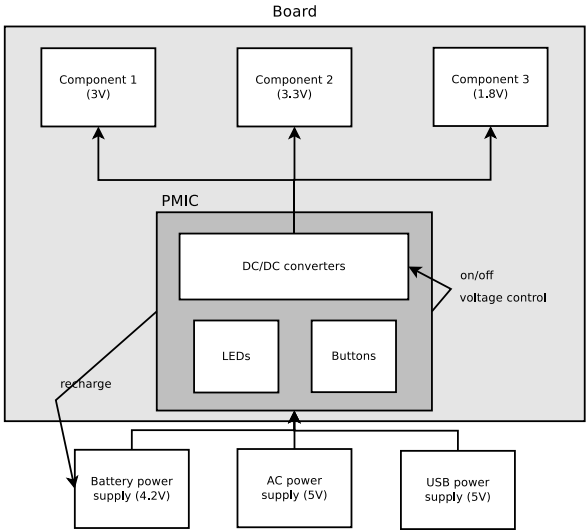
Boards without a PMIC



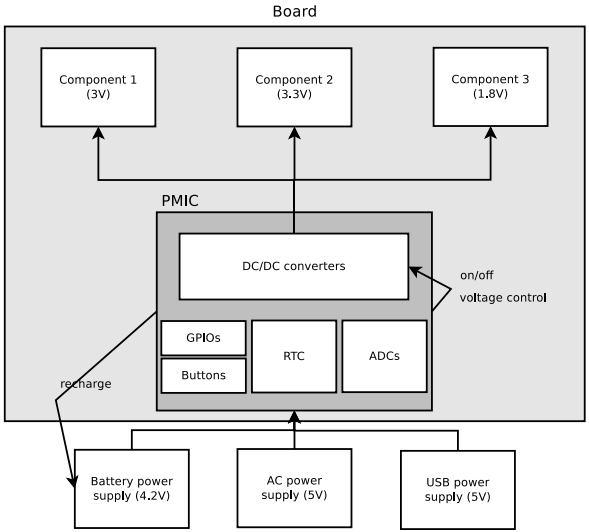
Active Semi ACT8865 (Atmel Sama5d3 Xplained)



TI TPS65217x (BeagleBone Black)

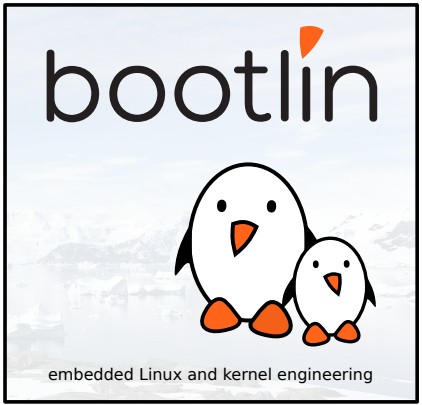


Boards with an X-Powers AXP20X PMIC



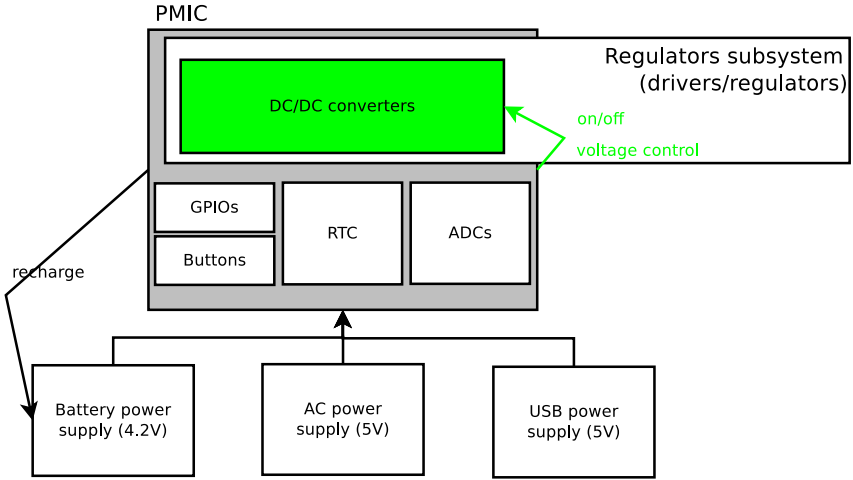
Commonly integrated features

Quentin Schulz
quentin.schulz@bootlin.com



Regulators

Regulators


PMIC
Regulators subsystem
(drivers/regulators)
DC/DC converters
on/off
voltage control
GPIOs
Buttons
RTC
ADCs
recharge
Battery power
supply (4.2V)
AC power
supply (5V)
USB power
supply (5V)

Regulators

- PMIC supplies power to components requiring different input voltages (5V, 3V3, 1V8, ...),
- DC-DC converters and LDO regulators handle the different voltages,
- to save power, regulators can stop supplying power to their unused components,
- some components support a range of input voltages,
- PMIC handles all that,
- their regulator adapts its voltage depending on some parameters (e.g. load, thermal throttling),
- variable regulators allow to reduce power consumption (undervolting) and increase power (overvolting),
  - allows CPU/GPU DVFS (Dynamic Voltage and Frequency Scaling),
  - is the core of battery life and power consumption,
- regulators are part of the regulator framework (`drivers/regulators/`),

Regulator driver example: AXP20X regulators driver

drivers/regulators/axp20x-regulator.c

```
static struct regulator_ops axp20x_ops = {
        .set_voltage_sel        = regulator_set_voltage_sel_regmap,
        .get_voltage_sel        = regulator_get_voltage_sel_regmap,
        .list_voltage           = regulator_list_voltage_linear,
        .enable                 = regulator_enable_regmap,
        .disable                = regulator_disable_regmap,
        .is_enabled             = regulator_is_enabled_regmap,
};
```

Regulator driver example: AXP20X regulators driver

drivers/regulators/axp20x-regulator.c

```
static const struct regulator_desc axp20x_regulators[] = {
        [AXP20X_DCDC2] = {
                .name           = "dcdc2",
                .supply_name    = "vin2",
                .of_match       = of_match_ptr("dcdc2"),
                .regulators_node = of_match_ptr("regulators"),
                .type           = REGULATOR_VOLTAGE,
                .id             = AXP20X_DCDC2,
                .n_voltages     = (2275 - 700) / (25 + 1),
                .owner          = THIS_MODULE,
                .min_uV         = 700 * 1000,
                .uV_step        = 25 * 1000,
                .vsel_reg       = AXP20X_DCDC2_V_OUT,
                .vsel_mask      = 0x3f,
                .enable_reg     = AXP20X_PWR_OUT_CTRL,
                .enable_mask    = 0x10,
                .ops            = &axp20x_ops,
        },
};
```

Regulator driver example: AXP20X regulators driver

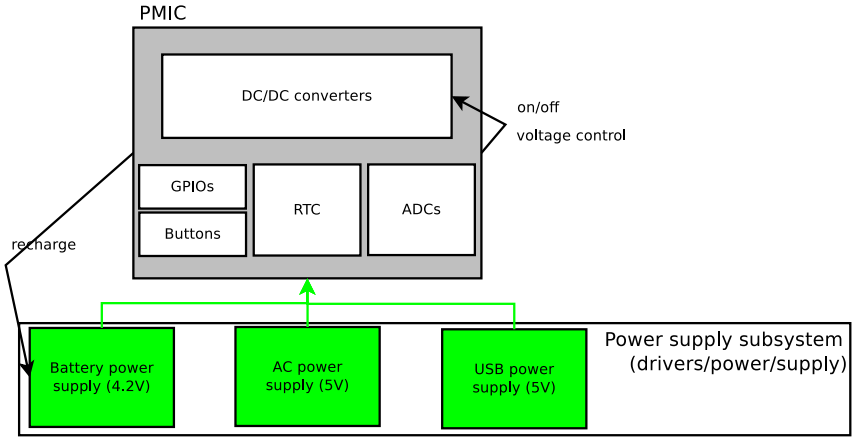
drivers/regulators/axp20x-regulator.c

```
static int axp20x_regulator_probe(struct platform_device *pdev)
{
        struct regulator_dev *rdev;
        struct axp20x_dev *axp20x = dev_get_drvdata(pdev->dev.parent);
        const struct regulator_desc *regulators;
        struct regulator_config config = {
                .dev = pdev->dev.parent,
                .regmap = axp20x->regmap,
                .driver_data = axp20x,
        };

        [...]
        for (i = 0; i < ARRAY_SIZE(axp20x_regulators); i++) {
                rdev = devm_regulator_register(&pdev->dev, axp20x_regulators[i], &config);
                if (IS_ERR(rdev)) {
                        dev_err(&pdev->dev, "Failed to register %s\n",
                                regulators[i].name);

                        return PTR_ERR(rdev);
                }
        }
        return 0;
}
```

Power supplies



Power supplies

Power supplies - Overview

The PMIC

- takes care of all possible supported external supplies:
  - AC (socket), USB, battery, ...
- defines the power sequence for the board,
- protects from overvoltage/undervoltage (e.g. X-Powers AXPs are designed for 5V boards but handles 0.3-11V)
- chooses the most suitable one depending on the status of each (low battery, not enough current supplied by a power supply, ...)
- may handle the battery (recharging, handling recharge cycles),

Power Supply subsystem

- is located in `drivers/power/supply`,
- has typically one driver per physical input power supply,
- can expose different data[1], such as current voltage and current, battery capacity, battery type, temperature, ...
- can set as many data, such as minimum and maximum allowed voltage or current, battery voltage when full,
- exposed information is specific to a PMIC (e.g. AXP20X can read current voltage and current values of the AC and USB power supplies unlike AXP22X),

[1]`http://lxr.bootlin.com/source/include/linux/power_supply.h`

Power Supply driver example: AXP20X USB driver

drivers/power/supply/axp20x_usb_power.c

```
static enum power_supply_property axp20x_usb_power_properties[] = {
        POWER_SUPPLY_PROP_PRESENT,
        POWER_SUPPLY_PROP_VOLTAGE_MIN,
        POWER_SUPPLY_PROP_VOLTAGE_NOW,
};

static int axp20x_usb_power_prop_writeable(struct power_supply *psy,
                                           enum power_supply_property psp)
{
        return psp == POWER_SUPPLY_PROP_VOLTAGE_MIN;
}

static const struct power_supply_desc axp20x_usb_power_desc = {
        .name = "axp20x-usb",
        .type = POWER_SUPPLY_TYPE_USB,
        .properties = axp20x_usb_power_properties,
        .num_properties = ARRAY_SIZE(axp20x_usb_power_properties),
        .property_is_writeable = axp20x_usb_power_prop_writeable,
        .get_property = axp20x_usb_power_get_property,
        .set_property = axp20x_usb_power_set_property,
};
```

Power Supply driver example: AXP20X USB driver

include/linux/power_supply.h

```
union power_supply_propval {
    int intval;
    const char *strval;
};
```

Power Supply driver example: AXP20X USB driver

drivers/power/supply/axp20x_usb_power.c

```
static int axp20x_usb_power_get_property(struct power_supply *psy, enum power_supply_property psp,
                                         union power_supply_propval *val)
{
        struct axp20x_usb_power *power = power_supply_get_drvdata(psy);
        switch (psp) {
        case POWER_SUPPLY_PROP_PRESENT:
                return axp20x_usb_power_is_present(power, &val->intval);
        [...]
        }
        return -EINVAL;
}

static int axp20x_usb_power_set_property(struct power_supply *psy, enum power_supply_property psp,
                                         const union power_supply_propval *val)
{
        struct axp20x_usb_power *power = power_supply_get_drvdata(psy);

        switch (psp) {
        case POWER_SUPPLY_PROP_VOLTAGE_MIN:
                return axp20x_usb_power_set_voltage_min(power, val->intval);
        [...]
        }
        return -EINVAL;
}
```

Power Supply driver example: AXP20X USB driver

drivers/power/supply/axp20x_usb_power.c

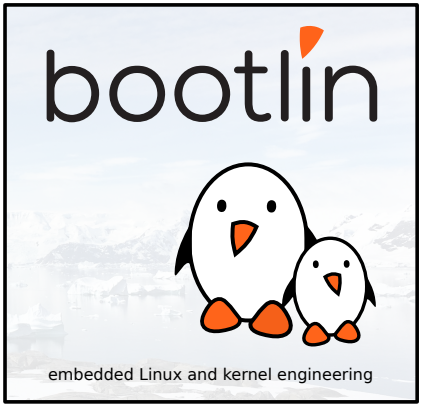
```
static int axp20x_usb_power_probe(struct platform_device *pdev)
{
        /* Custom structure */
        struct axp20x_usb_power *power;
        struct power_supply_config psy_cfg = {};

        power = devm_kzalloc(&pdev->dev, sizeof(*power), GFP_KERNEL);
        if (!power)
                return -ENOMEM;

        /* For use in other functions which call power_supply_get_drvdata */
        psy_cfg.drv_data = power;
        [...]
        power->supply = devm_power_supply_register(&pdev->dev, axp20x_usb_power_desc, &psy_cfg);
        if (IS_ERR(power->supply))
                return PTR_ERR(power->supply);
        [...]
        return 0;
}
```

Miscellaneous - PMIC-specific parts

Quentin Schulz
quentin.schulz@bootlin.com

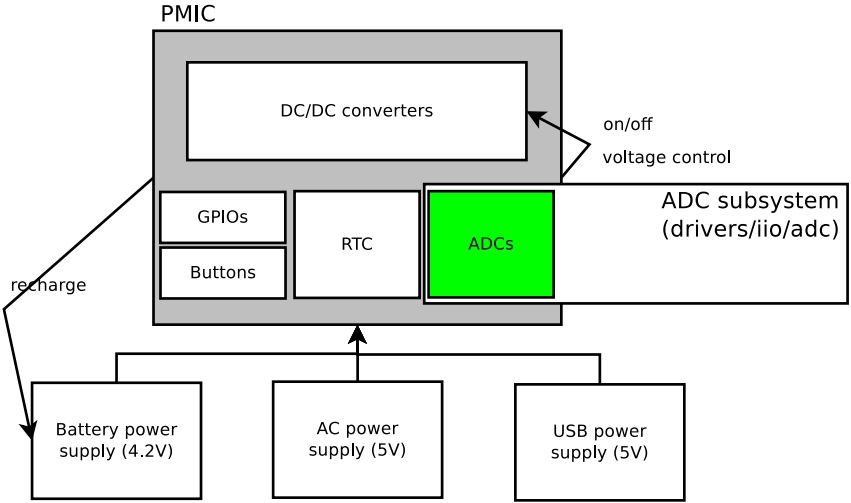


Parts specific to some PMICs

- Buttons: detect when the power reset button is pushed to shutdown the system (`drivers/power/reset`),
- GPIO: e.g. the AXP PMICs have several pins you can use either as GPIO or ADC,
- RTC with backup battery to keep time between reboots,
- Fuel gauge (if logically separated from the battery driver),
- ADC: e.g. AXP PMICs can expose what is the current voltage/current of a power supply,

ADC for current values

ADC driver


PMIC
DC/DC converters
on/off
voltage control
GPIOs
Buttons
RTC
ADCs
ADC subsystem
(drivers/iio/adc)
recharge
Battery power
supply (4.2V)
AC power
supply (5V)
USB power
supply (5V)

ADC - Current data values

- some PMICs can give some data in real time,
  - internal temperature, supplied voltage, consumed current, (dis)charging current, battery percentage, ...
- often stored in registers of an embedded Analog to Digital Converter (ADC),
- proper way: have a driver for this ADC feeding data to the power supply drivers,
  - the subsystem for ADC drivers is Industrial I/O (`drivers/iio/adc`)

IIO driver example: AXP20X ADC driver

drivers/iio/adc/axp20x_adc.c

```
#define AXP20X_ADC_CHANNEL(_channel, _name, _type, _reg)        \
        {                                                       \
                .type = _type,                                  \
                .indexed = 1,                                   \
                .channel = _channel,                            \
                .address = _reg,                                \
                .info_mask_separate = BIT(IIO_CHAN_INFO_RAW) |  \
                                      BIT(IIO_CHAN_INFO_SCALE), \
                .datasheet_name = _name,                        \
        }

enum axp20x_adc_channel_v {
        AXP20X_ACIN_V = 0,
        AXP20X_VBUS_V,
        [...]
};

enum axp20x_adc_channel_i {
        AXP20X_ACIN_I = 0,
        AXP20X_VBUS_I,
        [...]
};
```

IIO driver example: AXP20X ADC driver

drivers/iio/adc/axp20x_adc.c

```
/* Describe your IIO channels */
static const struct iio_chan_spec axp20x_adc_channels[] = {
        AXP20X_ADC_CHANNEL(AXP20X_VBUS_V, "vbus_v", IIO_VOLTAGE,
                           AXP20X_VBUS_V_ADC_H),
        AXP20X_ADC_CHANNEL(AXP20X_VBUS_I, "vbus_i", IIO_CURRENT,
                           AXP20X_VBUS_I_ADC_H),
        [...]
};

static int axp20x_adc_scale(struct iio_chan_spec const *chan, int *val, int *val2)
{
        switch (chan->type) {
        case IIO_VOLTAGE:
                if (chan->channel == AXP20X_VBUS_I) {
                        *val = 0;
                        *val2 = 375000;
                        return IIO_VAL_INT_PLUS_MICRO;
                }
                return -EINVAL;
        [...]
        }
}
```

IIO driver example: AXP20X ADC driver

drivers/iio/adc/axp20x_adc.c

```
static int axp20x_read_raw(struct iio_dev *indio_dev, struct iio_chan_spec const *chan, int *val,
                           int *val2, long mask)
{
        struct axp20x_adc_iio *info = iio_priv(indio_dev);
        switch (mask) {
        case IIO_CHAN_INFO_RAW:
                *val = axp20x_read_variable_width(info->regmap, chan->address, 12);
                if (*val < 0)
                        return *val;
                return IIO_VAL_INT;
        case IIO_CHAN_INFO_SCALE:
                return axp20x_adc_scale(indio_dev, chan, val);
        default:
                return -EINVAL;
        }
}

/* Specify the functions used when reading or writing to a sysfs entry */
static const struct iio_info axp20x_adc_iio_info = {
        .read_raw = axp20x_read_raw,
        .write_raw = axp20x_write_raw,
        .driver_module = THIS_MODULE,
};
```

IIO driver example: AXP20X ADC driver

drivers/iio/adc/axp20x_adc.c

```
/* Feed a consumer driver via an IIO channel */
static struct iio_map axp20x_maps[] = {
        {
                /* Name of the driver */
                .consumer_dev_name = "axp20x-usb-power-supply",
                /* The name under which the IIO channel will be gotten from the consumer driver */
                .consumer_channel = "vbus_v",
                /* The datasheet_name of the IIO channel to feed */
                .adc_channel_label = "vbus_v",
        }, {
                .consumer_dev_name = "axp20x-usb-power-supply",
                .consumer_channel = "vbus_i",
                .adc_channel_label = "vbus_i",
        }, { /* sentinel */ },
};
```

IIO driver example: AXP20X ADC driver

drivers/iio/adc/axp20x_adc.c

```
static int axp20x_probe(struct platform_device *pdev)
{
        struct axp20x_adc_iio *info;
        struct iio_dev *indio_dev;
        int ret;

        indio_dev = devm_iio_device_alloc(&pdev->dev,
                                          sizeof(*info));
        if (!indio_dev)
                return -ENOMEM;

        /* For use in other functions which call
         * iio_priv */
        info = iio_priv(indio_dev);

        indio_dev->name = "axp20x_ac";
        [...]
        indio_dev->dev.parent = &pdev->dev;
        indio_dev->dev.of_node = pdev->dev.of_node;
        indio_dev->modes = INDIO_DIRECT_MODE;
        indio_dev->info = axp20x_adc_iio_info;
        indio_dev->num_channels =
                ARRAY_SIZE(axp20x_adc_channels);
        indio_dev->channels = axp20x_adc_channels;
        ret = iio_map_array_register(indio_dev,
                                     axp20x_maps);
        if (ret < 0)
                return ret;

        ret = iio_device_register(indio_dev);
        if (ret < 0)
                return ret;
        return 0;
}
```

IIO driver example: AXP20X ADC driver

drivers/power/supply/axp20x_usb_power.c

```
static int axp20x_usb_power_get_property(struct power_supply *psy,
                                         enum power_supply_property psp,
                                         union power_supply_propval *val)
{
        struct axp20x_usb_power *power = power_supply_get_drvdata(psy);

        switch (psp) {
        [...]
        case POWER_SUPPLY_PROP_VOLTAGE_NOW:
                ret = iio_read_channel_processed(power->vbus_v, &val->intval);
                val->intval *= 1000;
                return 0;
        }
        return -EINVAL;
}
```

IIO driver example: AXP20X ADC driver

drivers/power/supply/axp20x_usb_power.c

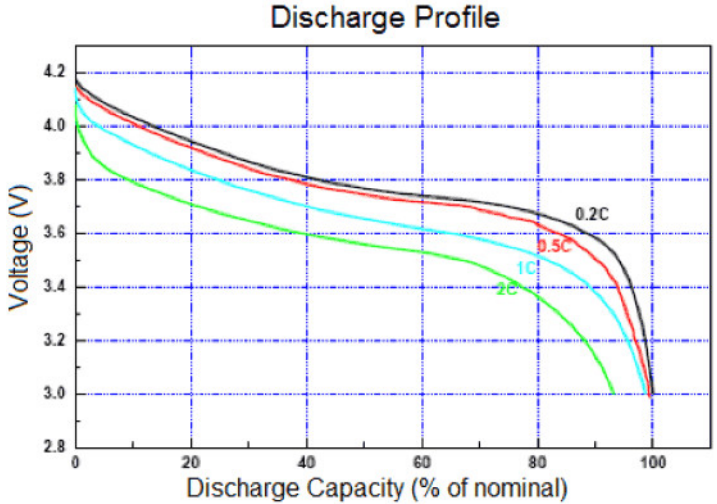
```
static int axp20x_usb_power_probe(struct platform_device *pdev)
{
        struct axp20x_usb_power *power;
        [...]
        power->vbus_v = devm_iio_channel_get(&pdev->dev, "vbus_v");
        if (IS_ERR(power->vbus_v)) {
                if (PTR_ERR(power->vbus_v) == -ENODEV)
                        return -EPROBE_DEFER;
                return PTR_ERR(power->vbus_v);
        }
        [...]
        power->supply = devm_power_supply_register(&pdev->dev, usb_power_desc, &psy_cfg);
        if (IS_ERR(power->supply))
                return PTR_ERR(power->supply);
        return 0;
}
```

Parts specific to boards - Fuel gauge

- battery percentage is approximated from its voltage,
- battery voltage does not linearly decrease in time or load,
- rather follows a curve, called the Open Circuit Voltage (OCV) curve,
- the curve is battery-specific (might be given by the battery vendor),
- the curve depends on several factors (environment, number of charges, age of battery, usage, ...),
- the battery percentage approximation by software must be done in userspace,
- use of POWER_SUPPLY_PROP_VOLTAGE_OCV property:
  - if software approximated, to give points on the OCV curve,
  - if hardware approximated, to get/set the points defining OCV curve used in the PMIC,

Worth reading: `https://training.ti.com/sites/default/files/BatteryMonitoringBasics.ppt`

Parts specific to boards - Fuel gauge

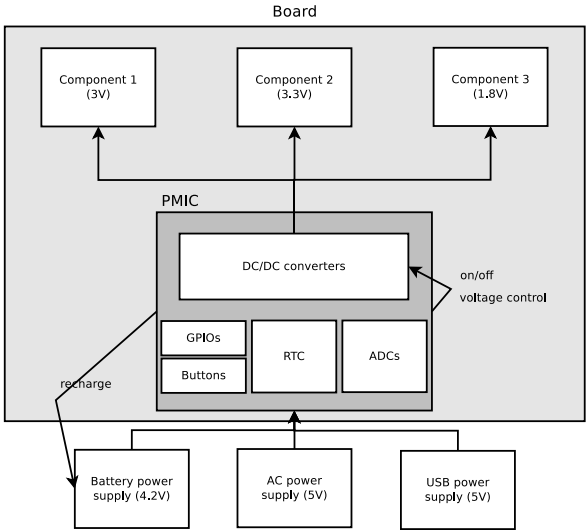


Discharge: 3.0V cutoff at room temperature.

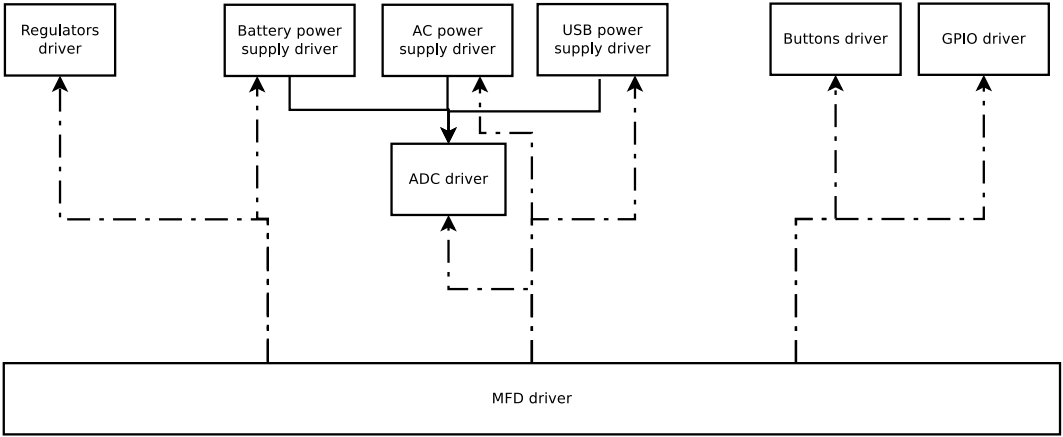
MFD

MFD - The glue between all this

- probes the different drivers of the PMIC (called MFD cells),
- maps the interrupts to the drivers which need them,
- usually passes a regmap to the MFD cells so it makes sure the drivers do not write to and access the same registers at the same time,


Board
Component 1
(3V)
Component 2
(3.3V)
Component 3
(1.8V)
PMIC
DC/DC converters
on/off
voltage control
GPIOs
Buttons
RTC
ADCs
recharge
Battery power
supply (4.2V)
AC power
supply (5V)
USB power
supply (5V)

MFD


Regulators driver
Battery power supply driver
AC power supply driver
USB power supply driver
Buttons driver
GPIO driver
ADC driver
MFD driver

MFD driver example: AXP20X MFD driver

drivers/mfd/axp20x.c

```
static struct resource axp20x_usb_power_supply_resources[] = {
        DEFINE_RES_IRQ_NAMED(AXP20X_IRQ_VBUS_PLUGIN, "VBUS_PLUGIN"),
};

static struct mfd_cell axp20x_cells[] = {
        {
                .name           = "axp20x-usb-power-supply",
                .of_compatible  = "x-powers,axp202-usb-power-supply",
                .num_resources  = ARRAY_SIZE(axp20x_usb_power_supply_resources),
                .resources      = axp20x_usb_power_supply_resources,
        }, [...]
};

int axp20x_device_probe(struct i2c_client *i2c, const struct i2c_device_id *id)
{
       /* Do all the regmap configuration, regmap_irqs included */
       ret = mfd_add_devices(&i2c->dev, -1, axp20x_cells,
                             ARRAY_SIZE(axp20x_cells), NULL, irq_base, NULL);

        if (ret)
                return ret;
        return 0;
}
```

Questions? Suggestions? Comments?

Quentin Schulz
quentin.schulz@bootlin.com

`http://bootlin.com/pub/conferences/2017/elc/schulz-pmics-keep-power-in-your-hands/`