The Power Supply Subsystem

Sebastian Reichel

Collabora

October 24, 2018
Sebastian Reichel

- Embedded Linux engineer at Collabora
  - Open Source Consultancy
- Based in Oldenburg, Germany
- Open Source contributor
  - Debian Developer
  - HSI and power-supply subsystem maintainer
  - Cofounder of Oldenburg’s Hack(er)/Makerspace
The power-supply subsystem

- batteries / fuel gauges
- chargers
- (board level poweroff/reset)
- Originally written and maintained by Anton Vorontsov (2007-2014)
- Temporarily maintained by Dmitry Eremin-Solenikov (2014)
Userspace Interface

root@localhost# ls /sys/class/power_supply/
AC  BAT0  BAT1
root@localhost# ls /sys/class/power_supply/BAT0
alarm  energy_full_design  status
capacity  energy_now  subsystem
capacity_level  manufacturer  technology
charge_start_threshold  model_name  type
charge_stop_threshold  power  uevent
cycle_count  power_now  voltage_min_design
...
root@localhost# cat /sys/class/power_supply/BAT0/capacity
65
Userspace Interface

root@localhost# udevadm info /sys/class/power_supply/BAT0
E: POWER_SUPPLY_CAPACITY=79
E: POWER_SUPPLY_ENERGY_FULL=15200000
E: POWER_SUPPLY_ENERGY_FULL_DESIGN=23200000
E: POWER_SUPPLY_ENERGY_NOW=12010000
E: POWER_SUPPLY_POWER_NOW=5890000
E: POWER_SUPPLY_STATUS=Discharging
E: POWER_SUPPLY_VOLTAGE_MIN_DESIGN=11100000
E: POWER_SUPPLY_VOLTAGE_NOW=11688000
...
Userspace Interface

- one power-supply device = one physical device
- All values are in uV, uA, uAh, uW, uWh, ...
- capacity can be exposed as
  - ENERGY_* - uWh
  - CHARGE_* - uAh
  - CAPACITY - percent (from 0 - 100)
  - CAPACITY_LEVEL - critical, low, normal, high, full
Power Supply Types

- Battery
- UPS (no mainline user)
- Mains
- USB
- USB_DCP, USB_CDP, USB_ACA, USB_TYPE_C, ... (deprecated)
Smart Batteries

- power-supply battery = smart battery
- smart battery = dumb battery + fuel gauge (+ static data)
- dumb battery = a bunch of battery cells
- fuel gauge = chip to measure battery status
static int my_battery_probe(struct platform_device *pdev) {
    return 0;
}
static struct platform_driver my_battery_driver = {
    .driver = {
        .name = "my-battery",
    },
    .probe = my_battery_probe,
};
module_platform_driver(my_battery_driver);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("My Name <e-mail@address>");
MODULE_DESCRIPTION("My Battery Driver");
```c
#define CONFIG_OF
static const struct of_device_id my_battery_id_table[] = {
    { .compatible = "vendor,my-battery" },
    { }
};
MODULE_DEVICE_TABLE(of, my_battery_id_table);
#endif

static struct platform_driver my_battery_driver = {
    .driver = {
        .of_match_table = of_match_ptr(my_battery_id_table),
        ...
    },
};
```
Device Tree

/ {
    some-controller {
        compatible = "vendor,my-controller";

        battery@42 {
            reg = <0x42>;
            compatible = "vendor,my-battery";
            vendor,some-property;
        }
    }
};
static int my_battery_probe(struct platform_device *pdev)
{
    struct power_supply_desc *psy_desc;
    struct power_supply_config psy_cfg = {};
    struct my_battery_ddata *ddata;
    int err;

    ddata = devm_kzalloc(ddata->dev, sizeof(*ddata), GFP_KERNEL);
    if (!ddata)
        return -ENOMEM;
    psy_desc = devm_kzalloc(ddata->dev, sizeof(*psy_desc), GFP_KERNEL);
    if (!psy_desc)
        return -ENOMEM;

    psy_cfg.of_node = pdev->dev.of_node;
    psy_cfg.drv_data = ddata;
    ...

... psy_desc->name = "battery";
psy_desc->type = POWER_SUPPLY_TYPE_BATTERY;
psy_desc->properties = my_battery_props;
psy_desc->num_properties = ARRAY_SIZE(my_battery_props);
psy_desc->get_property = my_battery_get_property;

ddata->psy = devm_power_supply_register(&pdev->dev, psy_desc, &psy_cfg);
err = PTR_ERR_OR_ZERO(ddata->psy);
if (err) {
    dev_err(ddata->dev, "failed to register power supply\n");
    return err;
}

return 0;

static enum power_supply_property my_battery_props[] = {
    POWER_SUPPLY_PROP_STATUS,
    POWER_SUPPLY_PROP_HEALTH,
    POWER_SUPPLY_PROP_TECHNOLOGY,
    POWER_SUPPLY_PROP_VOLTAGE_NOW,
    POWER_SUPPLY_PROP_CURRENT_NOW,
    POWER_SUPPLY_PROP_CAPACITY,
    POWER_SUPPLY_PROP_TEMP,
};
static int my_battery_get_property(struct power_supply *psy, enum power_supply_property psp,
    union power_supply_propval *val)
{
    struct my_battery_ddata *ddata = power_supply_get_drvdata(psy);

    switch (psp) {
        case POWER_SUPPLY_PROP_STATUS: ... return 0;
        case POWER_SUPPLY_PROP_HEALTH: ... return 0;
        case POWER_SUPPLY_PROP_TECHNOLOGY: ... return 0;
        case POWER_SUPPLY_PROP_VOLTAGE_NOW: ... return 0;
        case POWER_SUPPLY_PROP_CURRENT_NOW: ... return 0;
        case POWER_SUPPLY_PROP_CAPACITY: ... return 0;
        case POWER_SUPPLY_PROP_TEMP: ... return 0;
        default: return -EINVAL;
    }
}

Open First
case POWER_SUPPLY_PROP_HEALTH:
    int status;
    ret = regmap_read(ddata->regmap, MY_BATTERY_REG_STATUS, &status);
    if (ret)
        return ret;
    val->intval = POWER_SUPPLY_HEALTH_GOOD;
    if (status & MY_BATTERY_STATUS_COLD)
        val->intval = POWER_SUPPLY_HEALTH_COLD;
    if (status & MY_BATTERY_STATUS_VOLTAGE)
        val->intval = POWER_SUPPLY_HEALTH_OVERVOLTAGE;
    if (status & MY_BATTERY_STATUS_HOT)
        val->intval = POWER_SUPPLY_HEALTH_OVERHEAT;
    return 0;
Power Supply property getter - VOLTAGE

case POWER_SUPPLY_PROP_VOLTAGE_NOW:
    int voltage;
    ret = regmap_read(ddata->regmap, MY_BATTERY_REG_VOLT, &voltage);
    if (ret)
        return ret;
    val->intval = mV * 1000; /* mV -> uV */
    return 0;
static int my_battery_prop_is_writable(struct power_supply *psy,
    enum power_supply_property psp)
{
    switch (psp) {
        case POWER_SUPPLY_PROP_TEMP_ALERT_MIN:
        case POWER_SUPPLY_PROP_TEMP_ALERT_MAX:
            return 1;
        default:
            return 0;
    }
}

static int my_battery_probe(struct platform_device *pdev) {
    ...
    psy_desc->set_property = my_battery_set_property;
    psy_desc->property_is_writeable = my_battery_prop_is_writable;
    ...
}
static int my_battery_set_property(struct power_supply *psy,
    enum power_supply_property psp,
    const union power_supply_propval *val)
{
    struct my_battery_ddata *ddata = power_supply_get_drvdata(psy);

    switch (psp) {
        case POWER_SUPPLY_PROP_TEMP_ALERT_MIN:
            return regmap_write(ddata->regmap, MY_BATTERY_REG_ALARM_MIN, val->intval);
        case POWER_SUPPLY_PROP_TEMP_ALERT_MAX:
            return regmap_write(ddata->regmap, MY_BATTERY_REG_ALARM_MAX, val->intval);
        default:
            return -EINVAL;
    }
}
Power Supply - DONE

- Basic Driver construct
- DT initialization
- Driver Probe function
- Power Supply registration
- Get Property function
- Set Property function
power-supply documentation

- Documentation/power/power_supply_class.txt
- power-supply sysfs properties are now properly documented (Kudos to Adam Thomson)
- Documentation/ABI/testing/sysfs-class-power
Always check if there is a generic attribute first
Don't call `sysfs_create_*()`
  - It's racy!
  - Udev will not properly pick up the added attributes
New feature arriving shortly (expected to arrive in 4.21)
Power Supply - Custom sysfs attributes 2/2

```c
static ssize_t custom_attribute_show(struct device *dev, struct device_attribute *attr, char *buf) {
    return -EINVAL;
}
static DEVICE_ATTR_RO(custom_attribute);

static struct attribute *my_battery_sysfs_attrs[] = {
    &dev_attr_custom_attribute.attr,
    NULL
};
ATTRIBUTE_GROUPS(my_battery_sysfs);

struct power_supply_config psy_cfg = {
     ...
     .attr_grp = my_battery_sysfs_groups,  /* power supply core will (de)register this */
};

devm_power_supply_register(dev, my_battery_desc, &psy_cfg);
```
Power Supply - Supply Chain 1/2

Documentation/devicetree/bindings/power/supply/power_supply.txt

usb_charger: power@1 {
    compatible = "vendor,usb-charger";
    ...
};
ac_charger: power@2 {
    compatible = "vendor,ac-charger";
    ...
};

battery@3 {
    compatible = "vendor,battery";
    power-supplies = &usb_charger, &ac_charger;
    ...
};

Open First
static void my_battery_ext_pwr_changed(struct power_supply *psy)
{
    struct my_battery_ddata *ddata = power_supply_get_drvdata(psy);
    /* inform userspace, that battery status changed */
    power_supply_changed(psy);
}

static int my_battery_probe(struct platform_device *pdev) {
    struct power_supply_config psy_cfg = {
        .of_node = pdev->dev.of_node, /* provide DT info to power-supply core */
        ...
    };
    ...
    psy_desc->external_power_changed = my_battery_ext_pwr_changed;
    ...
    devm_power_supply_register(&pdev->dev, psy_desc, &psy_cfg);
}
Battery Info: `power_supply_battery_info`

- structure to describe battery characteristics
  - max. capacity
  - max. charge current
  - min/max. voltage
  - min/max. temperature
  - internal resistance
  - WIP: OCV to percentage table

- core provides function to acquire this information from DT
- consumed by fuel-gauge or charger
- Authored by Liam Breck
/* structure to hold battery characteristics */
struct power_supply_battery_info info = {};

/* read dumb battery characteristics from DT/ACPI/... */
if (power_supply_get_battery_info(psy, &info) < 0)
    return -EINVAL;

if (info.energy_full_design_uwh == -EINVAL)
    return -EINVAL;

return -EINVAL;
Battery Info: Discharge Curves

[Graph showing discharge curves at different temperatures: -20°C, 0°C, 23°C, 45°C]
<table>
<thead>
<tr>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ cat .../BAT0/type USB_CDP</td>
<td>$ cat .../BAT0/type USB</td>
</tr>
<tr>
<td>$ cat .../BAT0/usb_type DCP [CDP] ACA</td>
<td>DCP [CDP] ACA</td>
</tr>
</tbody>
</table>

- **type** property is supposed to be static and is read-only
- **usb_type** displays all supported modes
- **usb_type** marks current mode with square brackets
- **usb_type** may allow write access
- not all drivers have been converted (gpio-charger, isp1704, max14656)
New 'usb_type' property

```c
static enum power_supply_usb_type my_charger_usb_types[] = {
    POWER_SUPPLY_USB_TYPE_UNKNOWN,
    POWER_SUPPLY_USB_TYPE_SDP,
    POWER_SUPPLY_USB_TYPE_DCP,
    POWER_SUPPLY_USB_TYPE_CDP
};
static enum power_supply_property my_charger_props[] = {
    ..., POWER_SUPPLY_PROP_USB_TYPE, ...
};
static int my_charger_probe(struct platform_device *pd) {
    ...
    psy_desc->properties = my_charger_props;
    psy_desc->num_properties = ARRAY_SIZE(my_charger_props);
    psy_desc->usb_types = my_charger_usb_types;
    psy_desc->num_usb_types = ARRAY_SIZE(my_charger_usb_types);
    ...
    return devm_power_supply_register(&pd->dev, psy_desc, &psy_cfg);
}
```
Shortcoming: Battery with multiple fuel gauges

- Each fuel-gauge driver stands for a smart battery
- Sometimes hardware has two ways to measure battery properties
- This cannot be described with the power-supply subsystem at the moment
Shortcoming: Charger Manager

- Some embedded system require tight monitoring of charging process
- On x86 this is usually done via ACPI/embedded controller
- There is a charger-manager driver from Samsung doing this
- Proper support should exist in the core
Thanks!

Q & A