RK816 开发指南

文件标识：RK-KF-YF-074

发布版本：V1.3.0

日期：2020-07-22

文件密级：□绝密 □秘密 □内部资料 ■公开

免责声明

本文档按“现状”提供，瑞芯微电子股份有限公司（“本公司”，下同）不对本文档的任何陈述、信息和内容的准确性、可靠性、完整性、适销性、特定目的性和非侵权性提供任何明示或暗示的声明或保证。本文档仅作为使用指导的参考。

由于产品版本升级或其他原因，本文档将可能在未经任何通知的情况下，不定期进行更新或修改。

商标声明

“Rockchip”、“瑞芯微”、“瑞芯”均为本公司的注册商标，归本公司所有。

本文档可能提及的其他所有注册商标或商标，由其各自拥有者所有。

瑞芯微电子股份有限公司

Rockchip Electronics Co., Ltd.

地址： 福建省福州市铜盘路软件园A区18号

网址： www.rock-chips.com

客户服务电话： +86-4007-700-590

客户服务传真： +86-591-83951833

客户服务邮箱： fae@rock-chips.com

前言

概述

本文主要指导读者如何在RT-THREAD上开发RK816。

产品版本

芯片名称	RT-Thread 版本
RK816 芯片	

读者对象

本文档（本指南）主要适用于以下工程师：

技术支持工程师

软件开发工程师

修订记录

版本号	作者	修改日期	修改说明
V1.0.0	陈健洪	2020-01-06	初始版本
V1.1.0	陈健洪	2020-05-26	增加电池温度功能
V1.2.0	陈健洪	2020-06-29	完善Charger和NTC章节
V1.3.0	陈健洪	2020-07-22	增加RTC的alarm功能

目录

RK816 是一款高性能 PMIC，RK816 集成 4 个大电流 DCDC、1个升压BOOST、6 个 LDO、1个OTG输出、1 个 RTC、可调上电时序，而且还集成了开关充电，智能功率路径管理，库仑计等功能。

从用户使用的角度，RK816 的功能可以概括为如下几个方面：

- regulator：控制 DCDC、LDO 电源
- rtc：提供时钟计时、定时等功能
- gpio：out1 和 out2 两个推挽输出引脚（只能 output），可当普通 gpio 使用
- pwrkey： power 按键检测
- charger：控制电池充电电流、电池充电电压、系统输入电流
- fuel gauge：电池电量统计
- NTC sensor：电池温度检测

1. PMIC

1.1 驱动文件

```
./bsp/rockchip/common/drivers/drv_pmic.h
./bsp/rockchip/common/drivers/drv_pmic.c
```

1.2 配置使能

```
CONFIG_RT_USING_PMIC
CONFIG_RT_USING_PMIC_RK816

HAL_PWR_MODULE_ENABLED
HAL_PWR_I2C8_MODULE_ENABLED
```

1.3 板级定义

用户需要在 board.c 中定义PMIC的I2C硬件信息。例如：

```
struct pwr_i2c_desc pmic_pwr_i2c_desc =
{
    .name = "i2c0",
    .addr = RK816_I2C_ADDR,
};
```

1.4 用户接口

```
rt_uint32_t pmic_get_voltage(struct pwr_i2cbus_desc *desc);
rt_err_t pmic_set_voltage(struct pwr_i2cbus_desc *desc,
                          rt_uint32_t voltUv);
rt_uint32_t pmic_get_suspend_voltage(struct pwr_i2cbus_desc *desc);
rt_err_t pmic_set_suspend_voltage(struct pwr_i2cbus_desc *desc,
                                  rt_uint32_t voltUv);
rt_err_t pmic_set_enable(struct pwr_i2cbus_desc *desc, rt_uint32_t enable);
rt_err_t pmic_set_suspend_enable(struct pwr_i2cbus_desc *desc, rt_uint32_t
enable);
rt_uint32_t pmic_is_enabled(struct pwr_i2cbus_desc *desc);
int pmic_desc_init(struct pwr_i2cbus_desc *descs, uint32_t cnt);
void pmic_desc_deinit(void);
rt_err_t pmic_check_desc_by_pwrId(struct pwr_i2cbus_desc *pdesc, ePWR_ID pwrId);

// 系统关机下电
void pmic_power_off(void);
// 寄存器访问
rt_uint32_t pmic_read(struct rt_i2c_bus_device *pmic_i2c_bus,
                      rt_uint8_t addr, rt_uint16_t reg);
rt_err_t pmic_write(struct rt_i2c_bus_device *pmic_i2c_bus,
                    rt_uint8_t addr, rt_uint16_t reg, rt_uint8_t data);
rt_err_t pmic_update_bits(struct rt_i2c_bus_device *pmic_i2c_bus,
                          rt_uint8_t addr, rt_uint16_t reg,
                          rt_uint8_t mask, rt_uint8_t data);
// 子模块获取pmic板级定义
struct pwr_i2c_desc *pmic_get_i2c_desc(void);
```

2. Regulator

2.1 驱动文件

```
./bsp/rockchip/common/drivers/drv_regulator.c
./bsp/rockchip/common/drivers/drv_regulator.h
```

2.2 配置使能

没有额外配置项，把PMIC的配置使能即可。

2.3 定义接口

定义时请用专门的宏：

```
// 定义RK816的regulator
RK816_BUCK1_2(ID, PWR_ID) // 针对buck1和buck2;
RK816_BUCK4(ID, PWR_ID)   // 针对buck4;
RK816_LDO1_4(ID, PWR_ID)  // 针对ldo1、ldo2、ldo3、ldo4;
RK816_LDO5_6(ID, PWR_ID)  // 针对ldo5和ldo6
// 说明：buck3是通过外部的分压电路得到固定电压，所以没有声明接口。一般外设也不需要引用buck3。

// 定义regulator init状态
REGULATOR_INIT(NAME, ID, VOLTUV, EN, SSPDVOL, SSPDEN)
```

含义如下：

```
// @ID:       枚举类型，表示当前regulator的ID;
// @PWR_ID:   索引ID，即外设通过这个唯一的ID索引到对应的regulator!
#define RK816_BUCK1_2(ID, PWR_ID)

// @NAME:     名字
// @ID:       regulator的唯一pwr_id
// @VOLTUV:   初始化电压
// @EN:       是否使能
// @SSPDVOL:  休眠电压
// @SSPDEN:   休眠是否使能
REGULATOR_INIT(NAME, ID, VOLTUV, EN, SSPDVOL, SSPDEN)
```

2.4 板级定义

用户需要在board.c 定义regulators[]。例如：

```
static struct regulator_desc regulators[] =
{
    ......

    /* BUCK4 */
    {
        .flag = REGULATOR_FLG_I2C8 | REGULATOR_FLG_LOCK,
        .desc.i2c_desc = RK816_BUCK4(RK816_ID_DCDC4, PWR_ID_VCCIO_3V3),
        .desc.i2c_desc.i2c = &pmic_pwr_i2c_desc, // 引用PMIC的板级定义
    },
    /* LDO2 */
    {
        .flag = REGULATOR_FLG_I2C8 | REGULATOR_FLG_LOCK,
        .desc.i2c_desc = RK816_LDO1_4(RK816_ID_LDO2, PWR_ID_VCCIO_1V8),
        .desc.i2c_desc.i2c = &pmic_pwr_i2c_desc,
    },
    /* LDO5 */
    {
        .flag = REGULATOR_FLG_I2C8 | REGULATOR_FLG_LOCK,
        .desc.i2c_desc = RK816_LDO5_6(RK816_ID_LDO5, PWR_ID_VCC_AUDIO),
        .desc.i2c_desc.i2c = &pmic_pwr_i2c_desc,
```

```
    },
    /* LDO6 */
    {
        .flag = REGULATOR_FLG_I2C8 | REGULATOR_FLG_LOCK,
        .desc.i2c_desc = RK816_LDO5_6(RK816_ID_LDO6, PWR_ID_VCC_MIPI),
        .desc.i2c_desc.i2c = &pmic_pwr_i2c_desc,
    },

    ......
};
```

用户需要在board.c 定义regulator_inits[]。例如：

```
const struct regulator_init regulator_inits[] =
{
    REGULATOR_INIT("dcdc1", PWR_ID_VCCIO_1V8,    1250000, 0, 1400000, 1),
    REGULATOR_INIT("dcdc2", PWR_ID_BUCK_1V8,     1250000, 0, 1400000, 1),
    REGULATOR_INIT("ldo1",  PWR_ID_VCCIO_1V8_PMU,1200000, 0, 1400000, 1),
    REGULATOR_INIT("ldo3",  PWR_ID_MEMORY,       1200000, 0, 1400000, 1),
    REGULATOR_INIT("ldo4",  PWR_ID_TOP,          1200000, 0, 1400000, 1),
    REGULATOR_INIT("ldo5",  PWR_ID_VDD_1V1,      1200000, 0, 1400000, 1),
    REGULATOR_INIT("ldo6",  PWR_ID_VCCIO_3V3,    1200000, 0, 1400000, 1),
}
```

2.5 用户接口

```
struct regulator_desc *regulator_get_desc_by_pwrid(ePWR_ID pwrId);
rt_err_t regulator_set_voltage(struct regulator_desc *desc, int volt);
uint32_t regulator_get_voltage(struct regulator_desc *desc);
rt_err_t regulator_set_suspend_voltage(struct regulator_desc *desc, int volt);
uint32_t regulator_get_suspend_voltage(struct regulator_desc *desc);
uint32_t regulator_get_real_voltage(struct regulator_desc *desc);
rt_err_t regulator_enable(struct regulator_desc *desc);
rt_err_t regulator_disable(struct regulator_desc *desc);
void regulator_desc_init(struct regulator_desc *descs, uint32_t cnt);
```

3. RTC/Alarm

RTC驱动实现了如下功能：

- 实时时间的配置、获取

  时间范围：年月日时分秒。

- 闹钟(alarm)时间的配置、获取

  时间范围：年月日时分秒，但是不支持设置过期的闹钟；触发方式：仅单次触发。

3.1 驱动文件

```
./bsp/rockchip/common/drivers/pmic/rk816_rtc.c

// 用户层接口
./components/drivers/rtc/rtc.c
./components/drivers/include/drivers/rtc.h
./components/drivers/rtc/alarm.c
./components/drivers/include/drivers/alarm.h
```

3.2 配置使能

```
CONFIG_RT_USING_PMIC
CONFIG_RT_USING_RTC
CONFIG_RT_USING_ALARM
CONFIG_RT_USING_ALARM_CMD
CONFIG_RT_USING_RTC_RK816
```

3.3 用户接口

实时时间和日期的设置

```
rt_err_t set_time(rt_uint32_t hour, rt_uint32_t minute, rt_uint32_t second);
rt_err_t set_date(rt_uint32_t year, rt_uint32_t month, rt_uint32_t day);
```

实时时间和日期的获取：RTC时间一般通过时间管理框架的接口获取。例如：

```
struct tm* localtime(const time_t* t)
```

Alarm时间的操作

```
rt_alarm_t rt_alarm_create(rt_alarm_callback_t callback, struct rt_alarm_setup
*setup);
rt_err_t rt_alarm_control(rt_alarm_t alarm, int cmd, void *arg);
void rt_alarm_update(rt_device_t dev, rt_uint32_t event);
rt_err_t rt_alarm_delete(rt_alarm_t alarm);
rt_err_t rt_alarm_start(rt_alarm_t alarm);
rt_err_t rt_alarm_stop(rt_alarm_t alarm);
```

用户可直接参考RT-Thread官方API手册：

alarm: https://www.rt-thread.org/document/api/group__alarm.html

rtc: https://www.rt-thread.org/document/api/group__rtc.html

3.4 测试命令

`date`命令用于读写rtc时间。实现代码：

```
./components/drivers/rtc/rtc.c
```

范例：

```
msh />date 2018 04 01 12 25 07  // 设置时间：2018-04-01 12:15:07
msh />date                      // 获取当前时间
Sun Apr  1 12:25:07 2018
```

`alarm`命令用于读写闹钟时间。实现代码：

```
./components/drivers/rtc/alarm.c
```

范例：

```
// 闹钟列表(当前无)
msh />alarm
No alarm
// 设置闹钟前先确认当前的时间，避免设置过期闹钟（框架不支持）
msh />date
Sun Apr  1 12:25:07 2018
// 设置3个闹钟
msh />alarm 2018 04 01 12 30 30
msh />alarm 2018 04 01 12 30 40
msh />alarm 2018 04 01 12 30 50
// 查看闹钟列表，其中"*"表示即将触发或最后一个已触发过的闹钟。
// 已触发的闹钟如果不调用rt_alarm_delete()进行删除，则依旧会留在闹钟列表里。
msh />alarm
Alarm list:
      Sun Apr  1 12:30:50 2018
      Sun Apr  1 12:30:40 2018
    * Sun Apr  1 12:30:30 2018

// 设置完闹钟之后等待它们触发，闹钟触发时可以看到打印（仅限alarm命令）：
Alarm is ringing: Sun Apr  1 12:30:30 2018
Alarm is ringing: Sun Apr  1 12:30:40 2018
Alarm is ringing: Sun Apr  1 12:30:50 2018
// 再次查看闹钟列表，"*"指向最后一个已触发闹钟。
msh />alarm
Alarm list:
    * Sun Apr  1 12:30:50 2018
      Sun Apr  1 12:30:40 2018
      Sun Apr  1 12:30:30 2018
```

4. Charger

4.1 驱动文件

```
bsp/rockchip/common/drivers/pmic/rk816_charger.c
bsp/rockchip/common/drivers/pmic/rk816_charger.h

// 用户层接口
components/drivers/pm/charger.c
components/drivers/include/drivers/charger.h
```

用户只能通过`rt_device_control()`访问charger设备：

```
rt_err_t rt_device_control(rt_device_t dev, int cmd, void *arg)

// rt_device_control() 中的cmd 定义，“充电参数”指：充电电压、电流、输入电压等。
#define RT_DEVICE_CTRL_CHAGER_LIMIT_GET     (1) // 获取charger支持的各充电参数的最小、最大值
#define RT_DEVICE_CTRL_CHAGER_STATUS_GET    (2) // 获取当前的充电参数
#define RT_DEVICE_CTRL_CHAGER_BATVOL_SET    (3) // 设置电池充电电压，即满充电压值，单位：mV
#define RT_DEVICE_CTRL_CHAGER_BATCUR_SET    (4) // 设置电池的充电电流，单位：mA
#define RT_DEVICE_CTRL_CHAGER_FNSCUR_SET    (5) // 设置电池的充电截止电流，单位：mA
#define RT_DEVICE_CTRL_CHAGER_SRCCUR_SET    (6) // 设置来自适配器端的输入电流，单位：mA
```

4.2 配置使能

```
CONFIG_RT_USING_CHARGER
CONFIG_RT_USING_CHARGER_RK816
```

4.3 基础概念

- 最大充电电流

  RK816 带有智能电源路径管理功能，来自适配器的供电优先给系统使用，剩余的再给电池充电。软件上配置的允许向电池充电的最大剩余电流值称之为最大充电电流。

  ioctl访问方式：`RT_DEVICE_CTRL_CHAGER_BATCUR_SET`。

- 最大输入电流

  软件上配置可从适配器获取的最大电流，称之为“最大输入电流”。例如 5V/2A 的适配器，我们一般软件配置最大输入电流为 2A（也可以设置为 1.8A...）。

  ioctl访问方式：`RT_DEVICE_CTRL_CHAGER_SRCCUR_SET`。

- 最大充电电压

  电池满充状态的电压。

  ioctl访问方式：`RT_DEVICE_CTRL_CHAGER_BATVOL_SET`。

- 充电截止电流

  PMIC对电池停止充电的最小电流阈值。

  ioctl访问方式：`RT_DEVICE_CTRL_CHAGER_FNSCUR_SET`。

- 采样电阻

板子上电池端正负极端之间的采样电阻，PMIC通过采样电阻完成电压、电流的采集。通过改变采样电阻的阻值，用户可以扩展充电电流和充电截止电流的档位（放大或缩小）。硬件设计上，通常采用20mR采样电阻。该参数需要在板级配置中进行指定（见后续章节）。

4.4 硬件档位

根据采样电阻的阻值不同，用户可以扩展充电电流和充电截止电流的档位（放大或缩小），但是最大充电电压、最大输入电流不受影响。如下是根据寄存器定义顺序列举的硬件档位：

- 最大充电电压档位(7个档位)：

```
[4050, 4100, 4150, 4200, 4250, 4300, 4350]
```

- 最大输入电流档位(7个档位)：

```
[450, 80, 850, 1000, 1250, 1500, 1750, 2000]  // 注意：第2项是80，不是800
```

- 最大充电电流档位(8个档位，不同采样电阻有不同的档位值)：

```
10mR:     [2000, 2400, 2800, 3200, 3600, 4000, 4500, 4800]
20mR:     [1000, 1200, 1400, 1600, 1800, 2000, 2250, 2400]
40mR:     [500,   600,  700,  800,  900, 1000, 1125, 1200]
100mR:    [200,   240,  280,  320,  360,  400,  450,  480]
```

- 充电截止电流档位(4个档位，不同采样电阻有不同的档位值)：

```
10mR:     [300, 400, 600, 800]
20mR:     [150, 200, 300, 400]
40mR:     [75,  100, 150, 200]
100mR:    [30,   40,  60,  80]
```

注意事项:

- 上述不同采样电阻对应的参数档位值是固定的，不允许用户修改；
- 用户只能通过`rt_device_control()`的方式访问设备，并且只能传递上述列出的具体档位值。例如：100mR采样电阻情况下配置最大充电电流，只有8个档位值可以被传递：200，240，280，320，360，400，450，480；40mR下也是8个档位值可以被传递：500，600，700，800，900，1000，1125，1200。
- 用户不需要通过`rt_device_control()`接口传递采样电阻的阻值。这个阻值需要通过板级配置定义(见后续章节)，rk816_charger.c驱动初始化时会自动去获取该配置。

4.5 板级定义

用户只需要根据板子的实际硬件设计，在board.c 定义采样电阻的阻值（单位：毫欧）。比如：

```
struct rk816_charger_platform_data rk816_charger_pdata =
{
    .sample_res = 100, // 当前是100mR
};
```

4.6 用户接口

上述章节已提到用户需要通过`rt_device_control()`接口访问charger设备，如下给出更详细使用说明：

```
rt_err_t rt_device_control(rt_device_t dev, int cmd, void *arg)

// rt_device_control() 中的cmd 定义，“充电参数”指：充电电压、电流、输入电压等。
#define RT_DEVICE_CTRL_CHAGER_LIMIT_GET     (1) // 获取charger支持的各充电参数的最小、最大值
#define RT_DEVICE_CTRL_CHAGER_STATUS_GET    (2) // 获取当前的充电参数
#define RT_DEVICE_CTRL_CHAGER_BATVOL_SET    (3) // 设置电池充电电压，即满充电压值，单位：mV
#define RT_DEVICE_CTRL_CHAGER_BATCUR_SET    (4) // 设置电池的充电电流，单位：mA
#define RT_DEVICE_CTRL_CHAGER_FNSCUR_SET    (5) // 设置电池的充电截止电流，单位：mA
#define RT_DEVICE_CTRL_CHAGER_SRCCUR_SET    (6) // 设置来自适配器端的输入电流，单位：mA
```

- RT_DEVICE_CTRL_CHAGER_LIMIT_GET：

  使用：rt_device_control(device, RT_DEVICE_CTRL_CHAGER_LIMIT_GET, &limit);

  limit为`struct rt_charger_limit`类型变量，保存了从charger读的充电电压、充电电流、输入电流、截止电流的max和min档位。

- RT_DEVICE_CTRL_CHAGER_STATUS_GET：

  使用：rt_device_control(device, RT_DEVICE_CTRL_CHAGER_STATUS_GET, &status);

  status为`struct rt_charger_status`类型变量，保存了从charger读的当前充电电压、充电电流、输入电流、截止电流值。

- RT_DEVICE_CTRL_CHAGER_BATVOL_SET：

  使用：rt_device_control(device, RT_DEVICE_CTRL_CHAGER_BATVOL_SET, &voltage);

  voltage为rt_int32_t类型变量，用于配置充电电压。

- RT_DEVICE_CTRL_CHAGER_BATCUR_SET：
- RT_DEVICE_CTRL_CHAGER_FNSCUR_SET：
- RT_DEVICE_CTRL_CHAGER_SRCCUR_SET：

  类同RT_DEVICE_CTRL_CHAGER_BATVOL_SET的使用。

> 建议阅读bsp/rockchip/common/drivers/pmic/rk816_charger.c中的rk816_charger_control()函数实现。

注意事项

配置的档位值应该严格按照前面章节给出的档位表。否则：

- 配置的档位值超出min，max档位时，返回失败；
- 配置的档位值未超出min，max档位，但是没有准确匹配时则取小一档进行配置；

使用范例

```
struct rt_charger_status status;
struct rt_charger_limit limit;
rt_device_t device;
rt_uint32_t bat_volt = 4350;
rt_uint32_t bat_cur = 240;
```

```
device = rt_device_find("charger");
if (device == RT_NULL)
{
    return -RT_ERROR;
}

// 此处仅供示例参考
ret = rt_device_control(device, RT_DEVICE_CTRL_CHAGER_LIMIT_GET, &limit);
...
ret = rt_device_control(device, RT_DEVICE_CTRL_CHAGER_STATUS_GET, &status);
...
ret = rt_device_control(device, RT_DEVICE_CTRL_CHAGER_BATVOL_SET, &bat_volt);
...
ret = rt_device_control(device, RT_DEVICE_CTRL_CHAGER_BATCUR_SET, &bat_cur);
...
```

5. Fuel Gauge

5.1 驱动文件

```
bsp/rockchip/common/drivers/pmic/rk816_fg.c
bsp/rockchip/common/drivers/pmic/rk816_fg.h

// 用户层接口
components/drivers/pm/fuel_gauge.c
components/drivers/include/drivers/fuel_gauge.h
```

5.2 配置使能

```
RT_USING_PM_FG
RT_USING_FG_RK816
```

5.3 板级定义

用户需要在board.c 定义电量计相配置信息。比如：

```
struct rk816_fg_platform_data rk816_fg_pdata =
{
    .ocv_table = {
        3400, 3650, 3693, 3707, 3731, 3749, 3760,
        3770, 3782, 3796, 3812, 3829, 3852, 3882,
        3915, 3951, 3981, 4047, 4086, 4132, 4182,
    },
    .design_capacity = 4000,
    .design_qmax = 4500,
    .bat_res = 85,
    .sample_res = 10,
```

```
    .power_off_thresd = 3400,
    .zero_algorithm_vol = 3950,
    .virtual_power = 0,
    .max_soc_offset = 80,
    .monitor_sec = 5,
};
```

参数说明：

- `ocv table`
  每款电池都有自己的电池特性曲线，根据 ocv 特定电压对应特定电量的原则，我们将 0%~100%的电量细分成 21 个点，步进 5%电量，由此得到一张/组“电压<-->电量”的表格。这个表的用途就是第一次接电池开机、长时间关机后再开机、长时间休眠后校正库仑计的依据所在。该数据表可以由电池原厂提供，也可以由 RK 深圳分公司进行测量，具体请咨询深圳分公司相关工程师。例如：

  ```
  ocv_table = <3400 3599 3671 3701 3728 3746 3762  ......  4088 4132 4183>;
  // 表格的对应关系为：3400mv： 0%、 3599mv： 5%、  3671mv： 10%、 ……  4183mv： 100%
  ```

- `design_capacity`
  实际电池容量：经实际测量后确定的实际可用容量。例如标称 4000mah，但是实测只有 3850mah，则该值请填写 3850。

- `design_qmax`
  最大容量值，主要用途是作为软件处理的纠错条件之一。目前请该值请填写标称容量的 1.1 倍数值：即标称容量*1.1。

- `bat_res`
  电池内阻。主要在放电算法中会用到，非常重要！该值在测量 ocv_table 时一起获取，所以请注意这个参数的测量，切勿遗漏。

- `power_off_thresd` **请仔细阅读和理解**

  期待的系统关机电压，单位：mV。特别注意：该值指的是**VSYS 的瞬时电压**，而不是 vbat 端的电压（但是电量计采集的是 vbat 端的电压）！原理说明：Vbat 端的电压需要经过一个阻值大约 50 毫欧的 mos 管后（除此外，其实另外还有 PCB 走线带来的阻抗）才转换为 VSYS 供给系统， 所以把 VSYS 作为关机点依据才是正确的。由此我们可知：相同的 vbat 端电压，当前的负载电流越大，则 vsys 端电压就越低；反之，相同 vsys 下，当前负载电流越大，对应的 vbat 电压也就越高。

  RK 的平台不建议 vsys 端的电压低于 3.4v，这样容易导致 VCC_IO（3.3v）等 DCDC/LDO 的供电不稳定。

- `zero_algorithm_vol`

  进入电压+库仑计放电模式的电压值，单位：mV。低于该值，进入电压+库仑计结合的软件放电算法。建议：4.2v 电池设置为 3850mv，4.3v 及其以上的电池设置为 3950mv。

- `sample_res`

  电池端的采样电阻大小，单位：毫欧。跟charger的板级定义保持一致。

- `virtual_power`

  测试模式，供以后扩展使用，目前默认填写0。

- `max_soc_offset`

  开机校正时允许的最大电量误差。如果关机至少 30 分钟，则开机时会进行一次 ocv 表的电量查询，并且对比关机前的电量，如果偏差超过 max_soc_offset，即进行强制校正，把电量设置为 ocv 表对应的真实值。例如：当前显示电量是 20%，但是根据 ocv 电压推算的实际电量为 80%，则此时显示的电量直接显示为 80%。一般在发生死机后会出现这种电量偏差极大的情况，这个值的大小依客户的可接受程度，由客户自己进行设置，不建议这个值小于 60。

- `monitor_sec`

  轮询时间（秒）。电量计驱动需要不停轮询才能正常工作，目前建议 5~10s 比较合适，设置为 5s 最佳。

5.4 自初始化

开机时电量计驱动会通过`INIT_DEVICE_EXPORT()`完成自初始化并且创建一个电量计的线程，每隔`monitor_sec`跑一次电量计算法，更新内部状态。所以用户只要做好板级配置、正常使能驱动即可。

5.5 用户接口

```
rt_err_t rt_device_control(rt_device_t dev, int cmd, void *arg)

// rt_device_control() 中的cmd 定义：
#define RT_DEVICE_CTRL_FG_GET       (1) /* Get fuel gauge status */
```

用户通过此接口可以获取当前的电池电量、电池电压、电池容量等状态。

范例：

```
struct rt_fg_status status;
rt_device_t device;
rt_uint32_t volt = 4350;

device = rt_device_find("fuel_gauge");
if (device == RT_NULL)
{
    return -RT_ERROR;
}

// 此处仅供示例参考（省略返回值判断）
rt_device_control(device, RT_DEVICE_CTRL_FG_GET, &status);
```

5.6 调试开关

用户如果对电量计功能进行拷机，或者在使用过程发现电量异常等问题，都需要打开电量计的调试开关抓取调试信息。打开调试开关后，电量计驱动每隔`monitor_sec`就会打印一次电量计的调试信息。

目前有两种方式打开调试信息：

- 静态编译

  rk816_fg.c 中使能调试开关后重新编译固件：

  ```
  static int dbg_enable = 0;  // 1：使能调试信息  0：关闭
  ```

- 动态配置

  用户在shell命令中执行：

```
fg_rk816 1 // 1：使能调试信息  0：关闭
```

6. NTC Sensor

6.1 驱动文件

```
bsp/rockchip/common/drivers/pmic/rk816_sensor.c
bsp/rockchip/common/drivers/pmic/rk816_sensor.h

// 用户层接口
components/drivers/sensors/sensor.c
components/drivers/include/drivers/sensor.h
```

6.2 配置使能

```
CONFIG_RT_USING_SENSOR
CONFIG_RT_USING_SENSOR_RK816
```

6.3 基础概念

电池温度检测原理：电池在不同的温度下有不同的内阻，用户通过某种方式获取到电池阻值后，根据电池规格书上的"温度-阻值"表获即可取对应的电池温度。一般情况下，温度和阻值成反比关系。

RK816内部会提供一个类似恒流源的模块，通过TS脚对电池输出大小固定的电流脉冲，这样在电池端就会产生一定压降。RK816通过TS脚采集电池NTC脚上内阻压降反推出当前电池的阻值，然后根据"温度-阻值"表获取当前电池温度。

6.4 板级定义

用户需要在board.c 定义电池的NTC配置信息。比如：

```
// 1. 数组内可定义任意多个值；数值为阻值，单位：欧姆
// 2. rk816支持的最大ntc值为110000，请勿超过
static const rt_uint32_t rk816_ntc_table[] = {
    42450, 33930, 27280, 22070, 17960, 14700, 12090, /* -10 ~ 20'C */
    10000, 8310,  6490,  5830,  4910,  4160,  3540,  /* 25  ~ 55'C */
};

struct rk816_sensor_platform_data rk816_sensor_pdata =
{
    .ntc_table = &rk816_ntc_table[0],
    .ntc_num = 14,            // 等价于：ARRAY_SIZE(rk816_ntc_table)
    .ntc_degree_min = -10,    // rk816_ntc_table[0]对应的温度，即最低温度。
    .ntc_degree_step = 5,     // 温度步进，即上述各ntc值之间的步进
```

```
};
```

上面的`rk816_ntc_table`数据需要从电池规格书中获取，规格书上通常会有电阻温度和内阻的表格数据。

6.5 用户接口

```
rt_size_t rt_device_read (rt_device_t dev, rt_off_t pos, void *buffer,rt_size_t size);
```

范例：

```
rt_device_t dev = RT_NULL;
struct rt_sensor_data sensor_data;

device = rt_device_find("temp_rk816");
if (device == RT_NULL)
{
    return -RT_ERROR;
}

// 此处仅供示例参考（省略返回值判断）
// 1. 温度会被保存在：sensor_data.data.temp
// 2. 温度单位：摄氏度*10。
res = rt_device_read(dev, 0, &sensor_data, 1);
...
```

6.6 测试命令

框架提供了cmd实现：

```
./components/drivers/sensors/sensor_cmd.c
```

支持的命令：

```
msh />sensor

sensor  [OPTION] [PARAM]
         probe <dev_name>      Probe sensor by given name
         info                  Get sensor info
         sr <var>              Set range to var
         sm <var>              Set work mode to var
         sp <var>              Set power mode to var
         sodr <var>            Set output date rate to var
         read [num]            Read [num] times sensor
                               num default 5
```

RK816支持的命令：

```
sensor probe temp_rk816  // 必须先执行此命令完成驱动probe，然后才能执行其它命令
sensor info              // sensor信息（意义不大）
sensor read              // 读取温度
```

7. 其他文档

《Rockchip_Developer_Guide_RT-Thread_Power_CN.md》