密级状态：绝密（　）　秘密（　）　内部（　）　公开（ √ ）

RK 平台 PMIC 框架介绍以及开发说明

文件状态： [　] 正在修改 [√] 正式发布	当前版本：	V1.0
	作　　者：	杨永忠
	完成日期：	2016-06-10
	审　　核：	
	完成日期：	2016-6-10

福州瑞芯微电子股份有限公司

Fuzhou　Rockchips　Semiconductor　Co . , Ltd

版 本 历 史

版本号	作者	修改日期	修改说明	备注
1.0	杨永忠	2016-06-10	创建	

1. 概述

PMIC 是保证系统正常运行工作的重要部分，主要功能是提供给主控 IC 相关模块的工作电源，并且提供接口给 DVFS 机制调用以满足系统的功耗需求。PMIC 的配置是否正确直接关系到系统的稳定性，需要重点关注。

2. 重要概念

2.1 Regulator，中文名翻译为“稳定器”，是 Voltage Regulator（稳压器）或者 Current Regulator（稳流器）的简称，指可以自动维持恒定电压（或电流）的装置。

2.2 PMIC(Power Management IC),中文名翻译为“电源管理 IC”,也就是我们通常所说的 PMU(Power Management Unit)。

2.3 DCDC(Direct Current)，直流变到直流,例如：直流电压(3.0V)转换成其它的直流电压(1.5V)。

2.4 LDO(Low Dropout Regulator)，低压差线性稳压器。

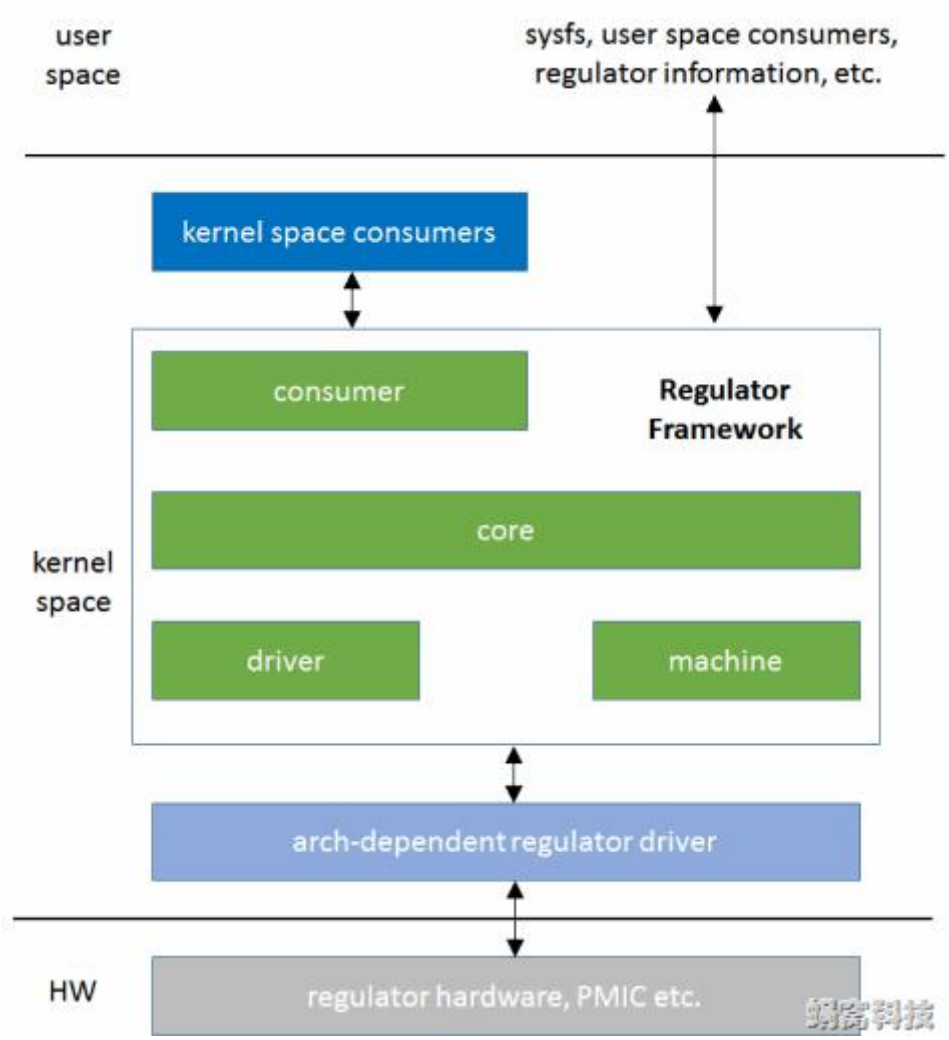
2.5 DVFS(Dynamic Voltage And Frequency Scaling)，动态电压频率调整，动态技术则是根据芯片所运行的应用程序对计算能力的不同需要，动态调节芯片的运行频率和电压(对于同一芯片，频率越高，需要的电压也越高)，从而达到节能的目的。

3. 软件框架

RK 平台经常使用到的 RICOH619,ACT8846,RK818,RT5037,RK816,RK808 等等，一般有多路

DCDC 以及多路 LDO 组成，为系统主要的模块提供工作电源,有的 PMIC 带有电量计。从软件角度而言，每个 DCDC 或者 LDO 抽象为一个 Regulator，PMIC 是由多个 Regulator 组成。目前 RK 平台官方支持的主要型号如下列表，客户可以依据具体需求选型。

型号	充电	电量计	RTC	DCDC	LDO	BOOST
RK818	√	√	√	4	9(81 个 switch)	1
RK816	√	√	√	4	6(1 个 switch)	1
RK808			√	4	10（2 个 switch）	1
ACT8846				4	8	0
RICOH619	√	√	√	5	12(2 个 ldo_rtc)	1



RK 平台 PMIC 框架相关代码路径：

kernel/drivers/regulator/

kernel/drivers/mfd/

kernel/drivers/power/

4. PMIC 驱动介绍

4.1.以 rk818 驱动为例，讲解 LINUX 通用 regulator 的驱动框架流程。代码路径：drivers/mfd/rk818.c,其实大部分 regulator 的驱动都会放在此文件夹下:drivers/regulators/.

Rk818 首先是个 I2C 设备，i2c 设备驱动的添加流程这边不细

说，主要关注 regulator 的部分。

4.1.1.STEP1: 解析 dts “regulator”节点。

```
/include/ "../../../arm/boot/dts/rk818.dtsi"
&rk818 {
        gpios =<&gpio0 GPIO_A1 GPIO_ACTIVE_HIGH>,<&gpio0 GPIO_A0 GPIO_ACTIVE_LOW>;
        rk818,system-power-controller;
        //pinctrl-names = "default";
        //pinctrl-0 = <&gpio0_c1>;
        regulators {

                rk818_dcdc1_reg: regulator@0{
                        regulator-name= "vdd_logic";/*vcc logic*/
                        regulator-min-microvolt = <700000>;/*<725000>;*/
                        regulator-max-microvolt = <1500000>;
                        regulator-initial-mode = <0x2>;
                        regulator-initial-state = <3>;
                        regulator-state-mem {
                                regulator-state-mode = <0x2>;
                                regulator-state-enabled;
                                regulator-state-uv =<900000>;
                        };
                };

                rk818_dcdc2_reg: regulator@1 {
                        regulator-name= "vdd2v3";
                        regulator-min-microvolt = <1100000>;
                        regulator-max-microvolt = <1100000>;
                        regulator-initial-mode = <0x2>;
                        regulator-initial-state = <3>;
                        regulator-state-mem {
                                regulator-state-mode = <0x2>;
                                regulator-state-enabled;
                                regulator-state-uv = <1100000>;
                        };
                };
```

注意：定义的 gpios =<&gpio0 GPIO_A1 GPIO_ACTIVE_HIGH>,<&gpio0 GPIO_A0 GPIO_ACTIVE_LOW>;第一个 GPIO 口定义是 PMIC 中断口 PMIC_INT GPIO0_A1，第二个定义 PMIC_SLEEP IO 口 GPIO0_A0.这些口要依据硬件配置。以此类推，如果其它 PMIC 的配置不同，可以回归到驱动去仔细确认下 IO 口的作用，不能模糊配置。

rk818.dtsi:

```
&rk818 {
        compatible = "rockchip,rk818";

        regulators {
                #address-cells = <1>;
                #size-cells = <0>;

                rk818_dcdc1_reg: regulator@0 {
                        reg = <0>;
                        regulator-compatible = "rk818_dcdc1";
                        regulator-always-on;
                        regulator-boot-on;
                };

                rk818_dcdc2_reg: regulator@1 {
                        reg = <1>;
                        regulator-compatible = "rk818_dcdc2";
                        regulator-always-on;
                        regulator-boot-on;
                };
```

Dts 与 rk818.dts 配置是包含关系，以 rk818_dcdc1_reg 为例，最终合并而成的 rk818_dcdc1_reg 节点为：

```
rk818_dcdc1_reg: regulator@0{
    //regulator id 号
    reg = <0>;
    //regulator 适配名称
    regulator-compatible = "rk818_dcdc1";
    //regulator 是否常开
    regulator-always-on;
    //regulator 是否启动时打开
    regulator-boot-on;
    //regulator 的名称
    regulator-name= "vdd_logic";
    //regulator 调压范围的最小值
    regulator-min-microvolt=<700000>;
```

```
            //regulator 调压范围的最大值
            regulator-max-microvolt = <1500000>;
            //regulator  初始时的模式，具体驱动的实现
            regulator-initial-mode = <0x2>;
            //regulator  初始时的状态，具体驱动的实现
            regulator-initial-state = <3>;
            //regulator 深度休眠时的状态
            regulator-state-mem {
              //regulator 休眠时的模式
              regulator-state-mode = <0x2>;
//regulator 进入深度休眠时打开，关闭 regulator-state-disabled
              regulator-state-enabled;
//深度休眠时的电压 900mv，依赖 regulator-state-enabled;属性
              regulator-state-uv =<900000>;
            };
          };
```

```
static struct rk818_board *rk818_parse_dt(struct rk818 *rk818)
{
	struct rk818_board *pdata;
	struct device_node *regs,*rk818_pmic_np;
	int i, count;

	rk818_pmic_np = of_node_get(rk818->dev->of_node);
	if (!rk818_pmic_np) {
		printk("could not find pmic sub-node\n");
		return NULL;
	}

	regs = of_find_node_by_name(rk818_pmic_np, "regulators");
	if (!regs)
		return NULL;

	count = of_regulator_match(rk818->dev, regs, rk818_reg_matches,
				   rk818_NUM_REGULATORS);
	of_node_put(regs);
	if ((count < 0) || (count > rk818_NUM_REGULATORS))
		return NULL;

	pdata = devm_kzalloc(rk818->dev, sizeof(*pdata), GFP_KERNEL);
	if (!pdata)
		return NULL;

	for (i = 0; i < count; i++) {
		if (!rk818_reg_matches[i].init_data || !rk818_reg_matches[i].of_node)
			continue;

		pdata->rk818_init_data[i] = rk818_reg_matches[i].init_data;
		pdata->of_node[i] = rk818_reg_matches[i].of_node;
	}
	pdata->irq = rk818->chip_irq;
	pdata->irq_base = -1;

	pdata->irq_gpio = of_get_named_gpio(rk818_pmic_np,"gpios",0);
		if (!gpio_is_valid(pdata->irq_gpio)) {
			printk("invalid gpio: %d\n", pdata->irq_gpio);
			return NULL;
		}

	pdata->pmic_sleep_gpio = of_get_named_gpio(rk818_pmic_np,"gpios",1);
			if (!gpio_is_valid(pdata->pmic_sleep_gpio)) {
				printk("invalid gpio: %d\n", pdata->pmic_sleep_gpio);
		}
	pdata->pmic_sleep = true;
	pdata->pm_off = of_property_read_bool(rk818_pmic_np,"rk818,system-power-controller");

	return pdata;
} ? end rk818_parse_dt ?
```

从 DTS 里解析各个 regulator 的配置信息填入 struct regulator_init_data *init_data;这个结构体，其中 dts 配置的消息会填入 struct regulation_constraints constraints 此容器里面。大家看下结构体就应该清楚了。

```
struct regulator_init_data {
	const char *supply_regulator;	/* or NULL for system supply */

	struct regulation_constraints constraints;

	int num_consumer_supplies;
	struct regulator_consumer_supply *consumer_supplies;

	/* optional regulator machine specific init */
	int (*regulator_init)(void *driver_data);
	void *driver_data;	/* core does not touch this */
};

struct regulation_constraints {

	const char *name;

	/* voltage output range (inclusive) - for voltage control */
	int min_uV;
	int max_uV;

	int uV_offset;

	/* current output range (inclusive) - for current control */
	int min_uA;
	int max_uA;

	/* valid regulator operating modes for this machine */
	unsigned int valid_modes_mask;

	/* valid operations for regulator on this machine */
	unsigned int valid_ops_mask;

	/* regulator input voltage - only if supply is another regulator */
	int input_uV;

	/* regulator suspend states for global PMIC STANDBY/HIBERNATE */
	struct regulator_state state_disk;
	struct regulator_state state_mem;
	struct regulator_state state_standby;
	suspend_state_t initial_state; /* suspend state to set at init */

	/* mode to set on startup */
	unsigned int initial_mode;

	unsigned int ramp_delay;

	/* constraint flags */
	unsigned always_on:1;	/* regulator never off when system is on */
	unsigned boot_on:1;	/* bootloader/firmware enabled regulator */
	unsigned apply_uV:1;	/* apply uV constraint if min == max */
} ?  end regulation_constraints ?  ;
```

注意:

of_regulator_match 是通过 rk818_reg_matches 这个数组去配置要配置具体的 regulator.

```
static struct of_regulator_match rk818_reg_matches[] = {
     { .name = "rk818_dcdc1", .driver_data = (void *)0 },
     { .name = "rk818_dcdc2", .driver_data = (void *)1 },
     { .name = "rk818_dcdc3", .driver_data = (void *)2 },
     { .name = "rk818_dcdc4", .driver_data = (void *)3 },
     { .name = "rk818_ldo1", .driver_data = (void *)4 },
     { .name = "rk818_ldo2", .driver_data = (void *)5 },
     { .name = "rk818_ldo3", .driver_data = (void *)6 },
     { .name = "rk818_ldo4", .driver_data = (void *)7 },
     { .name = "rk818_ldo5", .driver_data = (void *)8 },
     { .name = "rk818_ldo6", .driver_data = (void *)9 },
     { .name = "rk818_ldo7", .driver_data = (void *)10 },
     { .name = "rk818_ldo8", .driver_data = (void *)11 },
     { .name = "rk818_ldo9", .driver_data = (void *)12 },
     { .name = "rk818_ldo10", .driver_data = (void *)13 },
};
```

这里的.name 必须要与 rk818.dtsi 各个 reglator-compatible 一致，否则将会导致此 regulator 注册不上。

4.1.2. STEP2: 通常 PMU 都有两组寄存器控制同一个 DCDC/LDO 的输出，一组是开机之后使用的，一组是深度休眠时的电压，通过 PMIC_SLEEP 引脚切换，但是并不是所有 PMIC 都支持。首先设置 PMIC_SLEEP 引脚的状态：

```
/*****************************set sleep vol & dcdc mode*****************/
#ifdef CONFIG_OF
rk818->pmic_sleep_gpio = pdev->pmic_sleep_gpio;
if (rk818->pmic_sleep_gpio) {
        ret = gpio_request(rk818->pmic_sleep_gpio, "rk818_pmic_sleep");
        if (ret < 0) {
            dev_err(rk818->dev,"Failed to request gpio %d with ret:""%d\n",  rk818->pmic_sleep_gpio, ret);
            return IRQ_NONE;
        }
        gpio_direction_output(rk818->pmic_sleep_gpio,0);
        ret = gpio_get_value(rk818->pmic_sleep_gpio);
        gpio_free(rk818->pmic_sleep_gpio);
        pr_info("%s: rk818_pmic_sleep=%x\n", __func__, ret);
}
#endif
/*************************************************************/
```

4.1.3. STEP3:依据前面解析出来的 init_data 数据，调用 regulator_register( )注册各路 regulator.

```
if (pdev) {
    rk818->num_regulators = rk818_NUM_REGULATORS;
    rk818->rdev = kcalloc(rk818_NUM_REGULATORS,sizeof(struct regulator_dev *), GFP_KERNEL);
    if (!rk818->rdev) {
        return -ENOMEM;
    }
    /* Instantiate the regulators */
    for (i = 0; i < rk818_NUM_REGULATORS; i++) {
    reg_data = pdev->rk818_init_data[i];
    if (!reg_data)
        continue;
    config.dev = rk818->dev;
    config.driver_data = rk818;
    if (rk818->dev->of_node)
        config.of_node = pdev->of_node[i];
    if (reg_data && reg_data->constraints.name)
            rail_name = reg_data->constraints.name;
        else
            rail_name = regulators[i].name;
        reg_data->supply_regulator = rail_name;

    config.init_data =reg_data;

    rk818_rdev = regulator_register(&regulators[i],&config);
    if (IS_ERR(rk818_rdev)) {
        printk("failed to register %d regulator\n",i);
    goto err;
    }
    rk818->rdev[i] = rk818_rdev;
    } ? end for i=0;i<rk818_NUM_REGUL... ?
} ? end if pdev ?
```

regulator_register(const struct regulator_desc *regulator_desc,

const struct regulator_config *config)

需要传入两个参数：

A）.struct regulator_desc *regulator_desc。定义：此 regulator 的类型（.type = REGULATOR_VOLTAGE），调整电压还是调整电流。目前只讨论 REGULATOR_VOLTAGE 此类型；regulator 具体实现的函数集。

```
static struct regulator_desc regulators[] = {

    {
        .name = "RK818_DCDC1",
        .id = 0,
        .ops = &rk818_dcdc_ops,
        .n_voltages = ARRAY_SIZE(buck_voltage_map),
        .type = REGULATOR_VOLTAGE,
        .owner = THIS_MODULE,
    },
```

```
{
        .name = "RK818_LDO1",
        .id =4,
        .ops = &rk818_ldo_ops,
        .n_voltages = ARRAY_SIZE(ldo_voltage_map),
        .type = REGULATOR_VOLTAGE,
        .owner = THIS_MODULE,
},
```

需要实现的 ops 函数：

```
static struct regulator_ops rk818_dcdc_ops = {
        .set_voltage = rk818_dcdc_set_voltage,
        .get_voltage = rk818_dcdc_get_voltage,
        .list_voltage= rk818_dcdc_list_voltage,
        .is_enabled = rk818_dcdc_is_enabled,
        .enable = rk818_dcdc_enable,
        .disable = rk818_dcdc_disable,
        .get_mode = rk818_dcdc_get_mode,
        .set_mode = rk818_dcdc_set_mode,
        .set_suspend_enable =rk818_dcdc_suspend_enable,
        .set_suspend_disable =rk818_dcdc_suspend_disable,
        .set_suspend_mode = rk818_dcdc_set_suspend_mode,
        .set_suspend_voltage = rk818_dcdc_set_sleep_voltage,
        .set_voltage_time_sel = rk818_dcdc_set_voltage_time_sel,
};

static struct regulator_ops rk818_ldo_ops = {
        .set_voltage = rk818_ldo_set_voltage,
        .get_voltage = rk818_ldo_get_voltage,
        .list_voltage = rk818_ldo_list_voltage,
        .is_enabled = rk818_ldo_is_enabled,
        .enable = rk818_ldo_enable,
        .disable = rk818_ldo_disable,
        .set_suspend_enable =rk818_ldo_suspend_enable,
        .set_suspend_disable =rk818_ldo_suspend_disable,
        .set_suspend_voltage = rk818_ldo_set_sleep_voltage,
};
```

B）.struct regulator_config *config，就是之前从 DTS 解析出来的配置数据。

```
struct regulator_config {
        struct device *dev;
        const struct regulator_init_data *init_data;
        void *driver_data;
        struct device_node *of_node;
        struct regmap *regmap;

        int ena_gpio;
        unsigned int ena_gpio_invert:1;
        unsigned int ena_gpio_flags;
};
```

到此一个 PMIC 的各个 regulator 就注册完成了，驱动就可以调用 regulator 接口操作对应的 DCDC/LDO，实现使能，关闭，设置电压等等功能。

4.2.Regulator 常用接口函数说明

//通过 regulator-name 获取 regulator

struct regulator *regulator_get(struct device *dev, const char *id);

//释放 regulator,与 regulator_get 配对使用

void regulator_put(struct regulator *regulator);

//使能 regulator

int regulator_enable(struct regulator *regulator);

//关闭 regulator

int regulator_disable(struct regulator *regulator);

//强制关闭 regulator

int regulator_force_disable(struct regulator *regulator);

//获取

int regulator_list_voltage(struct regulator *regulator, unsigned selector);

//判断当前电压 regulator 是否支持

Int regulator_is_supported_voltage(struct regulator *regulator,int min_uV, int max_uV);

//设置 regulator 输出电压

int regulator_set_voltage(struct regulator *regulator, int min_uV, int max_uV) ;

//设置 regulator 模式

int regulator_set_mode(struct regulator *regulator, unsigned

int mode);

//获取 regulator 模式

unsigned int regulator_get_mode(struct regulator *regulator);

5. 开发指引

5.1 RK 平台 DVFS 与 regulator 的匹配

客户经常遇到 CPU,GPU,DDR 不调频，基本都是因为对应的 regulator 未配置正确。DVFS 机制最基本的原则：提高频率是先抬压后抬频，降低频率是先降频再降电压。运行越高的频率需要越高的电压，如果 regulator 未注册对，调频肯定是无法运行的。

DVFS 一般会调整 vdd_arm(供 CPU 核工作电压)，vdd_logic(供 GPU 控制器，DDR 控制器工作电压)。注意：RK3288 不一样，GPU 控制器工作电压由 vdd_gpu 提供，vdd_logic 只提供 DDR 控制器工作电压。

rk312x.dtsi:

```
dvfs {
    vd_arm: vd_arm {
        regulator_name = "vdd_arm";
        pd_core {
            clk_core_dvfs_table: clk_core {
                operating-points = <
                    /* KHz    uV */
                    312000 1100000
                    504000 1100000
                    816000 1100000
                    1008000 1100000
                    >;
```

```
vd_logic: vd_logic {
        regulator_name = "vdd_logic";
        pd_ddr {
                clk_ddr_dvfs_table: clk_ddr {
                        operating-points = <
                                /* KHz    uV */
                                200000 1200000
                                300000 1200000
                                400000 1200000
                                >;
                        status = "disabled";
                };
        };

        pd_gpu {
                clk_gpu_dvfs_table: clk_gpu {
                        operating-points = <
                                /* KHz    uV */
                                200000 1200000
                                300000 1200000
                                400000 1200000
                                >;
                        status = "okay";
                        regu-mode-table = <
                                /*freq    mode*/
                                200000    4
                                0         3
                        >;
                        regu-mode-en = <0>;
                };
        };
};

vd_logic: vd_logic {
        regulator_name = "vdd_logic";
        pd_ddr {
                clk_ddr_dvfs_table: clk_ddr {
                        operating-points = <
                                /* KHz    uV */
                                200000 1200000
                                300000 1200000
                                400000 1200000
                                >;
                        status = "disabled";
                };
        };

        pd_gpu {
                clk_gpu_dvfs_table: clk_gpu {
                        operating-points = <
                                /* KHz    uV */
                                200000 1200000
                                300000 1200000
                                400000 1200000
                                >;
                        status = "okay";
                        regu-mode-table = <
                                /*freq    mode*/
                                200000    4
                                0         3
                        >;
                        regu-mode-en = <0>;
                };
        };
};
```

由上可以明显看出，regulator_name 就是 DVFS 指定的需要调用的 regulator 名称，因此 regulator 注册时必须指定 regulator-name 为“vdd_arm”,”vdd_logic”的 regulator，否则 DVFS 调压失败，会导致机器变频失败或者开机随机奔溃。

项目的 dts 一般都会引用上述 node 节点，配置项目的频率点以及对应电压。rk3126 dts 设置如下：

```
pd_core {
        clk_core {
                operating-points = <
                        /* KHz    uV */
                        216000 1000000
                        408000 1000000
                        600000 1100000
                        696000 1150000
                        816000 1200000
                        1008000 1350000
                        1200000 1425000
                        >;
```

pd_core 表征此路电源域的配置情况。operating-points 里面设置的就是 CPU 可以运行的频率点以及 CPU 运行的频率点对应的电压。此电压是可以修改的，与硬件此路电源的纹波情况或者主控批次相关。

vdd_logic dts 设置如下：

```
pd_ddr {
        clk_ddr {
                operating-points = <
                        /* KHz    uV */
                        200000 1100000
                        300000 1100000
                        400000 1100000
                        533000 1250000
                        >;

                freq-table = <
                        /*status                  freq(KHz)*/
                        SYS_STATUS_NORMAL         400000
                        SYS_STATUS_SUSPEND        200000
                        //SYS_STATUS_VIDEO_1080P  240000
                        //SYS_STATUS_VIDEO_4K     400000
                        SYS_STATUS_PERFORMANCE    528000
                        //SYS_STATUS_DUALVIEW     400000
                        //SYS_STATUS_BOOST        324000
                        //SYS_STATUS_ISP                  533000
                        >;
                auto-freq-table = <
                        240000
                        324000
                        396000
                        528000
                        >;
                auto-freq=<0>;
                status="okay";
                };
        };
```

```
pd_gpu {
        clk_gpu {
                operating-points = <
                        /* KHz    uV */
                        200000 1100000
                        300000 1100000
                        400000  1150000
                        //480000  1250000
                        >;
                        status = "okay";
                };
        };
```

GPU/DDR 对应的电压都是 vdd_logic 控制，同一时刻以哪个电压为准，原则：同一时刻，取最大电压。如上图：某一时刻，当 DDR 频率为 400M（1100mv）,GPU 频率为 400M(1150mv),则 vdd_logic 电压为 1150mv。

operating-points 里面设置的是可以运行的频率点以及运行的频率点对应的电压，调整的原则：越高的频率需要越高的电压，客户经常抬高频率确不抬高其对应的电压，导致机器不稳定。例如添加 DDR 一档频率点 456M：

```
clk_ddr {
        operating-points = <
                /* KHz    uV */
                200000 1100000
                300000 1100000
                400000 1100000
                456000 1150000
                533000 1250000
                >;
```

RK 平台的 DDR 变频策略是依据场景变频，在不同的场景运行不同的频率，以达到节省功耗的目的。频率点客户可以依据项目的实际情况调整，比如某个场景频率跑低了，可以对应抬高，机器 DDR 运行不了高频率，可以适当降低；可以屏蔽场景档位，建议至少保留 SYS_STATUS_NORMAL，SYS_STATUS_SUSPEND 两档。

```
//机器正常起来运行的 DDR 频率
 SYS_STATUS_NORMAL          400000
//机器进入一级休眠的 DDR 频率
 SYS_STATUS_SUSPEND         200000
//机器播放<=1080P 视频的 DDR 频率
 SYS_STATUS_VIDEO_1080P   240000
//机器播放 4K 视频的 DDR 频率
 SYS_STATUS_VIDEO_4K        400000
```

```
//机器运行跑分软件时的 DDR 频率
SYS_STATUS_PERFORMANCE   528000
//机器插入 HDMI，双显示时候的 DDR 频率
SYS_STATUS_DUALVIEW      400000
//机器滑动界面时候的 DDR 频率
SYS_STATUS_BOOST         324000
//机器进入 CAMERA 运行的频率
SYS_STATUS_ISP           533000
```

5.2 CONFIG 配置

5.2.1 RK818 配置

5.2.2 RK816 配置

5.2.3 RK808 配置

5.2.4 ACT8846 配置

5.2.5 RICOH619 配置

5.3.常见问题 FAQ

5.3.1 使能/关闭 regulator 流程以及参考代码

打开：

regulator_get();

regulator_enable();

regutlator_put();

关闭：

regulator_get();

regulator_disable();

regulator_put();

regulator_get() 获取到 regulator 以后，操作(enable,disable,set_voltage) 等等之后一定要再调用 regulator_put() 释放此 regulator. 否则下次调用同一个 regulator_get 会直接有 warning 报出。参考代码：

kernel/drivers/media/video/rk_camsys/camsys_internal.h

5.3.2 使能 regulator_enable,regulator_disable 打印如下警告：

```
[  386.684691] WARNING: at drivers/regulator/core.c:1683 _regulator_disable+0x30/0xec()
[  386.692322] unbalanced disables for rk818_ldo6
[  386.696701] Modules linked in: mali_kbase
[  386.700663] CPU: 0 PID: 1 Comm: init Tainted: G        W    3.10.0 #3
[  386.707014] [<c0013ec4>] (unwind_backtrace+0x0/0xe0) from [<c001176c>] (show_stack+0x10/0x14)
[  386.715405] [<c001176c>] (show_stack+0x10/0x14) from [<c0036988>] (warn_slowpath_common+0x4c/0x68)
[  386.724218] [<c0036988>] (warn_slowpath_common+0x4c/0x68) from [<c0036a24>] (warn_slowpath_fmt+0x2c/0x3c)
[  386.733628] [<c0036a24>] (warn_slowpath_fmt+0x2c/0x3c) from [<c02df9ec>] (_regulator_disable+0x30/0xec)
[  386.742868] [<c02df9ec>] (_regulator_disable+0x30/0xec) from [<c02dfad4>] (regulator_disable+0x2c/0x58)
```

int regulator_enable(struct regulator *regulator) 与 int regulator_disable(struct regulator *regulator)必须成对使用，也就是不能连续两次 regulator_enable 或者

regulator_disable，导致内部计数出错，下次 regulator_enable 或者 regulator_disable 失效，达不到想要的结果。

5.3.3 调用 regulator_disable 电压关闭无效

A).首先查看打印，相关 regulator 是否有注册上。

B).看下上述 5.3.2 中的问题，是否未成对使用 regulator_enable,regulator_disable.

C).注意 dts 里面 regulator 几点 regulator-always-on;是否配置了，如果配置了，则此 regulator 是关闭不了的。原因：

```
int regulator_enable(struct regulator *regulator)
{
    struct regulator_dev *rdev = regulator->rdev;
    int ret = 0;

    if (regulator->always_on)
        return 0;
```

注意：未配置 regulator-always-on;，regulator/core.c 注册的最后会调用 late_initcall(regulator_init_complete); 把此 regulator disable。相关代码：

```
static int __init regulator_init_complete(void)
{
    struct regulator_dev *rdev;
    struct regulator_ops *ops;
    struct regulation_constraints *c;
    int enabled, ret;

    /*
     * Since DT doesn't provide an idiomatic mechanism for
     * enabling full constraints and since it's much more natural
     * with DT to provide them just assume that a DT enabled
     * system has full constraints.
     */
    if (of_have_populated_dt())
        has_full_constraints = true;

    mutex_lock(&regulator_list_mutex);

    /* If we have a full configuration then disable any regulators
     * which are not in use or always_on.  This will become the
     * default behaviour in the future.
     */
    list_for_each_entry(rdev, &regulator_list, list) {
        ops = rdev->desc->ops;
        c = rdev->constraints;

        if (c && c->always_on)
            continue;

        mutex_lock(&rdev->mutex);

        if (has_full_constraints) {
            /* We log since this may kill the system if it
             * goes wrong. */
            rdev_info(rdev, "disabling\n");
            ret = _regulator_do_disable(rdev);
            if (ret != 0) {
                rdev_err(rdev, "couldn't disable: %d\n", ret);
            }
        } else {
            /* The intention is that in future we will
             * assume that full constraints are provided
             * so warn even if we aren't going to do
             * anything here.
             */
            rdev_warn(rdev, "incomplete constraints, leaving on\n");
        }
```

如果某个电压开机过程中不能关闭，但是需要在其它场合关闭，这里就会导致问题。例如屏的供电，如果直接用 regulator 控制,并没有使用 GPIO(lcd_en)控制，一级休眠就必须关掉（不关闭掉，某些屏漏电会导致屏闪），这里就会存在矛盾。需要修改 regulator/core.c regulator_init_complete 函数。

5.3.4 调用 reglator_set_voltage 设置电压无效

A).首先查看打印，相关 regulator 是否有注册上。

B). 查看 dts 里 regulator 的配置情况，regulator-min-microvolt regulator-max-microvolt 的设置。如果设置成了一样的：

regulator-name= "rk818_dcdc3";

regulator-min-microvolt = <1200000>;

regulator-max-microvolt = <1200000>;

这样要调试设置成其它电压是无效的。相关代码：

```
static void of_get_regulation_constraints(struct device_node *np,
                        struct regulator_init_data **init_data)
{
    const __be32 *min_uV, *max_uV, *uV_offset;
    const __be32 *min_uA, *max_uA, *ramp_delay;
    struct device_node *state;
    struct regulation_constraints *constraints = &(*init_data)->constraints;

    constraints->name = of_get_property(np, "regulator-name", NULL);

    min_uV = of_get_property(np, "regulator-min-microvolt", NULL);
    if (min_uV)
        constraints->min_uV = be32_to_cpu(*min_uV);
    max_uV = of_get_property(np, "regulator-max-microvolt", NULL);
    if (max_uV)
        constraints->max_uV = be32_to_cpu(*max_uV);

    /* Voltage change possible? */
    if (constraints->min_uV != constraints->max_uV)
        constraints->valid_ops_mask |= REGULATOR_CHANGE_VOLTAGE;
```

REGULATOR_CHANGE_VOLTAGE 这个标志位置位上，这个 regulator 才可以进行调压。

5.3.5 深度休眠时电压的设置

深度休眠时，休眠电流要尽量小，因此某些相关电源是可以关闭的或者休眠时设置更低的电压。PMIC_SLEEP 的控制分两部分：

A).PMU 的驱动，配置正常工作时的 PMIC_SLEEP 电平;

B).pm.c 里控制深度休眠时 PMIC_SLEEP 电平;

Dts 配置：

```
regulator-state-mem {
        regulator-state-mode = <0x2>;
        regulator-state-disabled;//深度休眠时关闭此路输出
        regulator-state-uv = <900000>;
};
regulator-state-mem {
        regulator-state-mode = <0x2>;
        regulator-state-enabled;//深度休眠时不关闭此路输出，并且设置电压为 0.9V.
        regulator-state-uv = <900000>;
};
```

5.3.6 RK312X RT5037 休眠无法唤醒

```
arch/arm/mach-rockchip/pm-rk312x.c
static u32 rkpm_slp_mode_set(u32 ctrbits)
{
    ...
    pmu_writel(32 * 30, RK312X_PMU_OSC_CNT);
+   pmu_writel(0x52080, RK312X_PMU_CORE_PWRUP_CNT);
    pmu_writel(pwr_mode_config, RK312X_PMU_PWRMODE_CON);
```

RK312X_PMU_CORE_PWRUP_CNT 这个寄存器设置的值最好是几个毫秒，因为主控不可能一直在等待供电恢复，这个值设大了会造成死机。 PMU 退出 SLEEP 到恢复供电一般是几个毫秒。

5.3.7 RK808 开机死机

```
diff --git a/drivers/mfd/rk808.c b/drivers/mfd/rk808.c
index 592bc3d..ce23680 100755
--- a/drivers/mfd/rk808.c
+++ b/drivers/mfd/rk808.c
@@ -48,7 +48,7 @@

 struct rk808 *g_rk808;
 #define DCDC_RAISE_VOL_BYSTEP 1
-#define DCDC_VOL_STEP 25000  //25mv
+#define DCDC_VOL_STEP 12500  /*12.5mv*/

 static struct mfd_cell rk808s[] = {
   {
@@ -514,7 +514,7 @@ static int rk808_dcdc_set_voltage(struct regulator_dev *dev,
         val = rk808_dcdc_select_min_voltage(dev, vol_temp, vol_temp, buck);
       //  printk("rk808_dcdc_set_voltage buck = %d vol_temp= %d old_voltage= %d
         ret = rk808_set_bits(rk808, rk808_BUCK_SET_VOL_REG(buck), BUCK_VOL_MASK
-       }while(vol_temp != max_uV);
+       } while (vol_temp < max_uV);
      }
      else{
        val = rk808_dcdc_select_min_voltage(dev, min_uV, max_uV, buck);
```

3.0.36 kernel 版本：cat /proc/clocks | grep arm

3.10 cat /d/clk/clk_summary | grep clk_core

查看 DVFS 情况：cat sys/dvfs/dvfs_tree

大家可以继续参考文档：

kernel\Documentation\power\regulator\