<http://www.wowotech.net/tag/regulator>

<http://blog.csdn.net/allen6268198/article/details/7357232>

Linux内核的动态电压和电流控制接口(regulator)

<http://blog.csdn.net/songcdut/article/details/8747039>

<http://blog.csdn.net/lurayvis/article/details/26642641>

## 概述

Regulator，中文名翻译为“稳定器”，在电子工程中，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。

voltage regulator最早应用于功放电路中，主要用于滤除电源纹波（100或者120Hz）和噪声，以及避免“输出电压随负载的变化而变化”的情况。后来，随着IC级别的regulator的出现（便宜了），voltage regulator几乎存在于任何的电子设备中。例如我们常见的嵌入式设备中，基本上每一种电压，都是经过regulator输出的。

相比较voltage regulator的广泛使用，很少见到current regulator的应用场景（相信大多数的嵌入式工程师都没有接触过）。它一般存在于电流源中，除此之外，它广泛存在于近年来新兴的LED照明设备中。current regulator在LED设备中的作用主要有两个：避免驱动电流超出最大额定值，影响其可靠性；获得预期的亮度要求，并保证各个LED亮度、色度的一致性。

虽然原理比较复杂，但从设备驱动的角度看，regulator的控制应该很简单，就是输出的enable/disable、输出电压或电流的大小的控制。那么，linux kernel的regulator framework到底要做什么呢？这就是本文的目的：弄清楚regulator framework背后思考，并总结出其软件架构（和common clock framework类似，consumer/provider/core）。

注1：有关regulator的描述，参考自

“http://sound.westhost.com/articles/vi-regulators.html”。

注2：kernel中有关regulator framework的介绍写的相当好

（Documentation\power\regulator\\*），因此本文大部分内容会参考这些文件。

## 背后的思考

Linux regulator framework的目的很直接：提供标准的内核接口，控制系统的voltage/current regulators，并提供相应的机制，在系统运行的过程中，动态改变regulators的输出，**以达到省电的目的。**

看似简单的背后，有些因素不得不考虑。

1. 最重要的，就是安全性：
   1. 在一个系统中，错误的regulator配置是非常危险的，严重时可以损毁硬件。而无论是regulator的使用者（consumer），还是regulator提供者（provider，即regulator driver），都不一定有足够的知识和能力，避免危险发生。因此必须从machine的角度，小心的设计regulator的输出限值（这一般由产品设计、硬件设计决定的）。
   2. 同时，一旦设计确定下来之后，这些限制必须保存在一些相对固定的地方，不能轻易地被软件修改。
   3. 最后，所有的regulator操作，必须是小心的、在可允许范围内的。

2）系统中大部分的设备，都没有动态更改regulator配置的需求，甚至连enable/disable都懒得关心的，framework需要考虑这种情况，尽量简化接口。

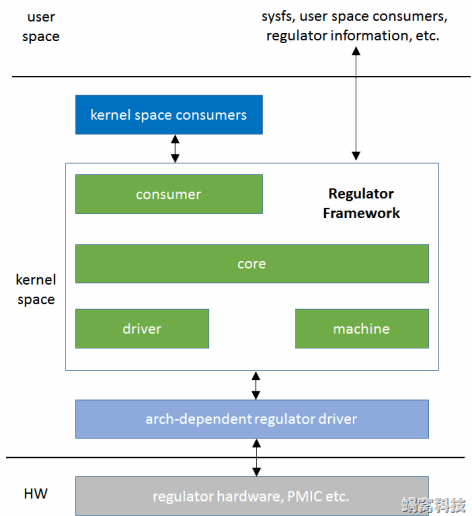
3）会存在同一个regulator向多个设备提供power的情况，如果这些设备的需求不同怎么办？

4）regulator之间是否可以级联？如果可以，怎么处理？

这些思考最终都会反映到软件设计上，具体可参考如下的软件架构。

## 软件架构

基于上面的思考，regulator framework的软件架构如下：



除了machine之外，基本上和[common clock framework](http://www.wowotech.net/linux_kenrel/clk_overview.html)的consumer/provider框架类似。

### **machine**

machine的主要功能，是使用软件语言（struct regulator\_init\_data），静态的描述regulator在板级的物理现状，包括：

1） 前级regulator（即该regulator的输出是另一个regulator的输入，简称supply regulator）和后级regulator（即该regulator的输入是其它regulator的输出，简称consumer regulator）。

这主要用于描述regulator在板级的级联关系，需要留意的是，它和clock不同，这种级联关系是非常确定的，以至于需要使用静态的方式描述，而不是像clock那样，在注册的时候动态指定并形成。

2）该regulator的物理限制（struct regulation\_constraints），包括：

1. 输出电压的最大值和最小值（voltage regulator）；
2. 输出电流的最大值和最小值（current regulator）；
3. 允许的操作（修改电压值、修改电流限制、enable、disable等等）；
4. 输入电压是多少（当输入是另一个regulator时）；
5. 是否不允许关闭（always\_on）；
6. 是否启动时就要打开（always\_on）；
7. 等等。

这些限制关系到系统安全，因此必须小心配置。配置完成后，在系统运行的整个过程中，它们都不会再改变了。

### **Driver**

driver模块的功能，是从regulator driver的角度，抽象regulator设备。

1）使用struct regulator\_desc描述regulator的静态信息，包括：名字、supply regulator的名字、中断号、操作函数集（struct regulator\_ops）、使用regmap时相应的寄存器即bitmap等等。

2）使用struct regulator\_config，描述regulator的动态信息（所谓的动态信息，体现在struct regulator\_config变量都是局部变量，因此不会永久保存），包括struct regulator\_init\_data指针、设备指针、enable gpio等等。

3）提供regulator的注册接口（regulator\_register/devm\_regulator\_register），该接口接受描述该regulator的两个变量的指针：struct regulator\_desc和struct regulator\_config，并分配一个新的数据结构（struct regulator\_dev，从设备的角度描述regulator），并把静态指针（struct regulator\_desc）和动态指针（struct regulator\_config）提供的信息保存在其中。

### consumer

consumer的功能，是从regulator consumer的角度，抽象regulator设备（struct regulator），并提供regulator操作相关的接口。包括：

### **core**

core负责上述逻辑的具体实现，并以sysfs的形式，向用户空间提供接口。

## 接口汇整

regulator framework向内核空间consumer提供的接口位于“include/linux/regulator/consumer.h”中，包括regulator的获取、使能、修改等接口，如下。

### 1）struct regulator

struct regulator结构用于从consumer的角度抽象一个regulator，consumer不需要关心该结构的细节，当作一个句柄使用即可（类似struct clk）。

### 2）regulator的get/put接口

struct regulator \*\_\_must\_check regulator\_get(struct device \*dev,

const char \*id);

struct regulator \*\_\_must\_check devm\_regulator\_get(struct device \*dev,

const char \*id);

struct regulator \*\_\_must\_check regulator\_get\_exclusive(struct device \*dev,

const char \*id);

struct regulator \*\_\_must\_check devm\_regulator\_get\_exclusive(struct device \*dev,

const char \*id);

struct regulator \*\_\_must\_check regulator\_get\_optional(struct device \*dev,

const char \*id);

struct regulator \*\_\_must\_check devm\_regulator\_get\_optional(struct device \*dev,

const char \*id);

void regulator\_put(struct regulator \*regulator);

void devm\_regulator\_put(struct regulator \*regulator);

根据是否独占regulator、是否可以多次get，regulator get接口分为三类：

正常的get，非独占、可以重复get，regulator\_get/devm\_regulator\_get；

独占性质的get，独占、不可重复get，regulator\_get\_exclusive/devm\_regulator\_get\_exclusive；

optional的get，非独占、不可重复get，regulator\_get\_optional/devm\_regulator\_get\_optional。

get接口的参数为id，会在下一篇文章中详细介绍。

### 3）supply alias相关的接口

int regulator\_register\_supply\_alias(struct device \*dev, const char \*id,

struct device \*alias\_dev,

const char \*alias\_id);

void regulator\_unregister\_supply\_alias(struct device \*dev, const char \*id);

int devm\_regulator\_register\_supply\_alias(struct device \*dev, const char \*id,

struct device \*alias\_dev,

const char \*alias\_id);

void devm\_regulator\_unregister\_supply\_alias(struct device \*dev,

const char \*id);

int devm\_regulator\_bulk\_register\_supply\_alias(struct device \*dev,

const char \*const \*id,

struct device \*alias\_dev,

const char \*const \*alias\_id,

int num\_id);

void devm\_regulator\_bulk\_unregister\_supply\_alias(struct device \*dev,

const char \*const \*id,

int num\_id);

具体意义请参考下一篇文章。

### 4）regulator的控制、状态获取接口

int \_\_must\_check regulator\_enable(struct regulator \*regulator);

int regulator\_disable(struct regulator \*regulator);

int regulator\_force\_disable(struct regulator \*regulator);

int regulator\_is\_enabled(struct regulator \*regulator);

int regulator\_disable\_deferred(struct regulator \*regulator, int ms);

int regulator\_can\_change\_voltage(struct regulator \*regulator);

int regulator\_count\_voltages(struct regulator \*regulator);

int regulator\_list\_voltage(struct regulator \*regulator, unsigned selector);

int regulator\_is\_supported\_voltage(struct regulator \*regulator,

int min\_uV, int max\_uV);

unsigned int regulator\_get\_linear\_step(struct regulator \*regulator);

int regulator\_set\_voltage(struct regulator \*regulator, int min\_uV, int max\_uV);

int regulator\_set\_voltage\_time(struct regulator \*regulator,

int old\_uV, int new\_uV);

int regulator\_get\_voltage(struct regulator \*regulator);

int regulator\_sync\_voltage(struct regulator \*regulator);

int regulator\_set\_current\_limit(struct regulator \*regulator,

int min\_uA, int max\_uA);

int regulator\_get\_current\_limit(struct regulator \*regulator);

int regulator\_set\_mode(struct regulator \*regulator, unsigned int mode);

unsigned int regulator\_get\_mode(struct regulator \*regulator);

int regulator\_set\_optimum\_mode(struct regulator \*regulator, int load\_uA);

int regulator\_allow\_bypass(struct regulator \*regulator, bool allow);

struct regmap \*regulator\_get\_regmap(struct regulator \*regulator);

int regulator\_get\_hardware\_vsel\_register(struct regulator \*regulator,

unsigned \*vsel\_reg,

unsigned \*vsel\_mask);

int regulator\_list\_hardware\_vsel(struct regulator \*regulator,

unsigned selector);

控制有关的包括enable、disable、电压设置、电流设置、mode设置等，其中disable又包括normal、强制、退出等类型。

状态获取包括：是否enable；是否可以改变电压；支持的电压列表；是否支持指定范围的电压；当前输出电压；当前电流限制；当前mode；等等。

更为详细的描述，请参考下一篇文章。

### 5）bulk型的操作（一次操作多个regulator）

int regulator\_bulk\_register\_supply\_alias(struct device \*dev,

const char \*const \*id,

struct device \*alias\_dev,

const char \*const \*alias\_id,

int num\_id);

void regulator\_bulk\_unregister\_supply\_alias(struct device \*dev,

const char \* const \*id, int num\_id);

int \_\_must\_check regulator\_bulk\_get(struct device \*dev, int num\_consumers,

struct regulator\_bulk\_data \*consumers);

int \_\_must\_check devm\_regulator\_bulk\_get(struct device \*dev, int num\_consumers,

struct regulator\_bulk\_data \*consumers);

int \_\_must\_check regulator\_bulk\_enable(int num\_consumers,

struct regulator\_bulk\_data \*consumers);

int regulator\_bulk\_disable(int num\_consumers,

struct regulator\_bulk\_data \*consumers);

int regulator\_bulk\_force\_disable(int num\_consumers,

struct regulator\_bulk\_data \*consumers);

void regulator\_bulk\_free(int num\_consumers,

struct regulator\_bulk\_data \*consumers);

### 6）notifier相关的接口

int regulator\_register\_notifier(struct regulator \*regulator,

struct notifier\_block \*nb);

int regulator\_unregister\_notifier(struct regulator \*regulator,

struct notifier\_block \*nb);

如果consumer关心某个regulator的状态变化，可以通过上面接口注册一个notifier。

### 7）其它接口

/\* driver data - core doesn't touch \*/

void \*regulator\_get\_drvdata(struct regulator \*regulator);

void regulator\_set\_drvdata(struct regulator \*regulator, void \*data);

用于设置和获取driver的私有数据。

## consumer模块向用户空间consumer提供的接口

用户空间程序可以通过sysfs接口，使用regulator，就像内核空间consumer一样。这些接口由“drivers/regulator/userspace-consumer.c”实现，主要包括：

sysfs目录位置：/sys/devices/platform/reg-userspace-consumer。

name，读取可以获取该regulator的名字。

state，读取，可以获取该regulator的状态（enabled/disabled）；写入可以改变regulator的状态（enabled或者1使能，disabled或者0禁止）。

## machine模块向regulator driver提供的接口

machine模块主要提供struct regulator\_init\_data、struct regulation\_constraints constraints等数据结构，用于描述板级的regulator配置，具体可参考3.1中介绍。

## driver模块向regulator driver提供的接口

regulator framework向regulator driver提供的接口位于“include/linux/regulator/driver.h”中，包括数据结构抽象、regulator注册等。

1）struct regulator\_desc、struct regulator\_config和struct regulator\_dev

见3.2中的介绍。

2）regulator设备的注册接口

struct regulator\_dev \*

regulator\_register(const struct regulator\_desc \*regulator\_desc,

const struct regulator\_config \*config);

struct regulator\_dev \*

devm\_regulator\_register(struct device \*dev,

const struct regulator\_desc \*regulator\_desc,

const struct regulator\_config \*config);

void regulator\_unregister(struct regulator\_dev \*rdev);

void devm\_regulator\_unregister(struct device \*dev, struct regulator\_dev \*rdev);

见3.2中的介绍。

3）其它接口，请参考后续的文章。

core模块向用户空间提供的sysfs接口

regulator设备在内核中是以regulator class的形式存在的，regulator core通过class->dev\_groups的方式，提供了一些默认的attribute，包括：

name，读取可以获取该regulator的名字；

num\_users，读取可获取regulator的使用者数目；

type，读取可以获取该regulator的类型（voltage或者current）。

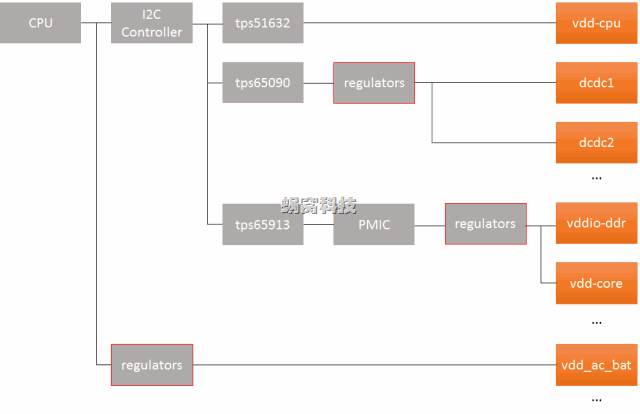
## 概述

本文从regulator driver的角度，描述怎样基于regulator framework编写regulator驱动。同时，以此为契机，学习、理解regulator有关的物理特性，以便能够更好的使用它们。

## regulator driver的实现步骤

### 确定系统中regulator有关的硬件组成

提起硬件，最好能有个例子，好在有device tree，一个活生生的硬件拓扑结构。这里以NVIDIA Tegra Dalmore A04开发板为例（regulator有关的device tree位于“arch\arm\boot\dts\tegra114-dalmore.dts”）：



这里的regulator结构是相当复杂的，其中彩色框代表最终的regulator抽象，它的前一级表示regulator的载体（可以是PMIC、CPU、等等）。下面将会详细说明：

a）CPU通过I2C controller，连接一个降压控制器（TI tps51632），该控制器输出名称为“vdd-cpu”的电压，就称作vdd-cpu regulator吧（因此，在kernel中，regulator是一个虚拟设备）。

b）CPU通过I2C controller，连接一个前端电源管理芯片（TI tps65090），该芯片除了具备充电管理功能外，内置了多个regulator，例如dcdc1、dcdc2等等。

c）CPU通过I2C controller，连接另一个电源管理芯片（TI tps65913），该芯片具有两个功能：GPIO输出和PMIC。PMIC内置了多个regulator，如vddio-ddr、vdd-core等等。

d）CPU内部也集成了一些regulator，如vdd\_ac\_bat等等。

注1：单纯从硬件的角度看，是不存在图中"regulators“、PMIC等实体的，它们的出现，已经包含了软件设计的思路。之所以画在这里，是方便后面的描述。

### 使用DTS，将硬件拓扑呈现出来