# 概述

pinctrl驱动实现的功能：

1. 枚举并命名所有pin控制器能控制的所有引脚
2. 分组管理，如i2c、spi等接口需多个pin配合使用，并且对于同一个外设控制器可能有多种pin组合，如一个i2c控制器，其引脚组合可能是{11,12}，也可能是{15,16}
3. 设置引脚的功能复用
4. 配置引脚，如上下拉、驱动能力、去抖等。

# 数据结构

pin控制器的信息通过如下结构描述：

struct pinctrl\_desc {

const char \*name;

const struct pinctrl\_pin\_desc \*pins;//所有引脚的描述信息

unsigned int npins;//引脚个数

const struct pinctrl\_ops \*pctlops;//获取group信息的操作方法

const struct pinmux\_ops \*pmxops;//引脚复用的操作方法

const struct pinconf\_ops \*confops;//引脚配置的操作方法

struct module \*owner;

#ifdef CONFIG\_GENERIC\_PINCONF

unsigned int num\_custom\_params;

const struct pinconf\_generic\_params \*custom\_params;

const struct pin\_config\_item \*custom\_conf\_items;

#endif

};

pinctrl\_pin\_desc结构描述了单个引脚的信息，一个pin控制器有多少个引脚就需要定义对应大小的该结构体数组，引脚的定义一般使用#define PINCTRL\_PIN(a, b) { .number = a, .name = b }宏完成：

struct pinctrl\_pin\_desc {

unsigned number;//引脚编号

const char \*name;//引脚名称

void \*drv\_data;

};

一些接口如uart、i2c等需要将多个pin组合在一起使用，即pin group，pinctrl\_ops结构提供了获取pin group信息的方法：

struct pinctrl\_ops {

int (\*get\_groups\_count) (struct pinctrl\_dev \*pctldev);//获取pctldev的group个数

const char \*(\*get\_group\_name) (struct pinctrl\_dev \*pctldev, unsigned selector);//获取group的名称

int (\*get\_group\_pins) (struct pinctrl\_dev \*pctldev, unsigned selector, const unsigned \*\*pins, unsigned \*num\_pins);//获取一个group对应的pin信息及个数

void (\*pin\_dbg\_show) (struct pinctrl\_dev \*pctldev, struct seq\_file \*s, unsigned offset);//debug fs的回调

int (\*dt\_node\_to\_map) (struct pinctrl\_dev \*pctldev, struct device\_node \*np\_config, struct pinctrl\_map \*\*map, unsigned \*num\_maps);//解析一个pin configuration node并将结果保存为mapping table entry，每个entry为一个setting

void (\*dt\_free\_map) (struct pinctrl\_dev \*pctldev, struct pinctrl\_map \*map, unsigned num\_maps);//上面函数的逆操作

};

pinmux\_ops结构提供了引脚复用的操作方法：

struct pinmux\_ops {

int (\*request) (struct pinctrl\_dev \*pctldev, unsigned offset);//引脚申请

int (\*free) (struct pinctrl\_dev \*pctldev, unsigned offset);//释放引脚

int (\*get\_functions\_count) (struct pinctrl\_dev \*pctldev);//获取pctldev的function个数

const char \*(\*get\_function\_name) (struct pinctrl\_dev \*pctldev, unsigned selector);//获取一个function名称

int (\*get\_function\_groups) (struct pinctrl\_dev \*pctldev, unsigned selector, const char \* const \*\*groups, unsigned \*num\_groups);//获取一个function所关联的group

int (\*set\_mux) (struct pinctrl\_dev \*pctldev, unsigned func\_selector, unsigned group\_selector);//引脚复用设置

int (\*gpio\_request\_enable) (struct pinctrl\_dev \*pctldev, struct pinctrl\_gpio\_range \*range, unsigned offset);

int (\*save\_context)(struct pinctrl\_dev \*pctldev);

void (\*restore\_context)(struct pinctrl\_dev \*pctldev);

void (\*gpio\_disable\_free) (struct pinctrl\_dev \*pctldev, struct pinctrl\_gpio\_range \*range, unsigned offset);

int (\*gpio\_set\_direction) (struct pinctrl\_dev \*pctldev, struct

pinctrl\_gpio\_range \*range, unsigned offset, bool input);

bool strict;

};

pinconf\_ops结构提供了引脚配置的操作方法：

struct pinconf\_ops {

#ifdef CONFIG\_GENERIC\_PINCONF

bool is\_generic;

#endif

int (\*pin\_config\_get) (struct pinctrl\_dev \*pctldev, unsigned pin, unsigned long \*config);//获取一个引脚的当前配置

int (\*pin\_config\_set) (struct pinctrl\_dev \*pctldev, unsigned pin,unsigned long \*configs,unsigned num\_configs);//配置引脚

int (\*pin\_config\_group\_get) (struct pinctrl\_dev \*pctldev, unsigned selector,unsigned long \*config);//获取一个group的引脚配置

int (\*pin\_config\_group\_set) (struct pinctrl\_dev \*pctldev, unsigned selector, unsigned long \*configs, unsigned num\_configs);//对一个group的所有引脚进行配置

int (\*pin\_config\_dbg\_parse\_modify) (struct pinctrl\_dev \*pctldev, const char \*arg, unsigned long \*config);

void (\*pin\_config\_dbg\_show) (struct pinctrl\_dev \*pctldev, struct seq\_file \*s, unsigned offset);

void (\*pin\_config\_group\_dbg\_show) (struct pinctrl\_dev \*pctldev, struct seq\_file \*s, unsigned selector);

void (\*pin\_config\_config\_dbg\_show) (struct pinctrl\_dev \*pctldev, struct seq\_file \*s, unsigned long config);

};

完成pin控制器的注册之后，会得到如下结构：

struct pinctrl\_dev {

struct list\_head node;//所有pcltdev都会挂到全局的pinctrldev\_list链表下

struct pinctrl\_desc \*desc;//描述信息

struct radix\_tree\_root pin\_desc\_tree;//该pcltdev所有引脚的信息都会存储于此基数树下

#ifdef CONFIG\_GENERIC\_PINCTRL\_GROUPS

struct radix\_tree\_root pin\_group\_tree;//该pcltdev的group信息存储于此基数树下

unsigned int num\_groups;

#endif

#ifdef CONFIG\_GENERIC\_PINMUX\_FUNCTIONS

struct radix\_tree\_root pin\_function\_tree;//该pcltdev的function信息存储于此基数树下

unsigned int num\_functions;

#endif

struct list\_head gpio\_ranges;

struct device \*dev;

struct module \*owner;

void \*driver\_data;

struct pinctrl \*p;

struct pinctrl\_state \*hog\_default;

struct pinctrl\_state \*hog\_sleep;

struct mutex mutex;

#ifdef CONFIG\_DEBUG\_FS

struct dentry \*device\_root;

#endif

};

当使用pinctrl时，也就是设备驱动对pin做出控制时，struct pinctrl用于管理一个设备的所有pin的控制状态，内核描述为pin control state holder：

struct pinctrl {

struct list\_head node;//系统中所有设备的pinctrl都会挂到全局的pinctrl\_list链表上

struct device \*dev;//该pinctrl所属device

struct list\_head states;//该设备支持的所有引脚控制状态

struct pinctrl\_state \*state;//当前的引脚控制状态

struct list\_head dt\_maps;//从dts中解析的信息

struct kref users;//引用计数

};

struct pinctrl\_state表示了设备的引脚控制状态：

struct pinctrl\_state {

struct list\_head node;//链到pinctrl->states上

const char \*name;//该state的名称

struct list\_head settings;//该状态的所有settings

};

struct pinctrl\_setting表示引脚控制状态的setting：

struct pinctrl\_setting {

struct list\_head node;//链到pinctrl\_state->settings上

enum pinctrl\_map\_type type;//控制信息的类型

struct pinctrl\_dev \*pctldev;

const char \*dev\_name;

union {

struct pinctrl\_setting\_mux mux;

struct pinctrl\_setting\_configs configs;

} data;

};

可用控制信息的类型如下：

enum pinctrl\_map\_type {

PIN\_MAP\_TYPE\_INVALID,

PIN\_MAP\_TYPE\_DUMMY\_STATE,

PIN\_MAP\_TYPE\_MUX\_GROUP,

PIN\_MAP\_TYPE\_CONFIGS\_PIN,

PIN\_MAP\_TYPE\_CONFIGS\_GROUP,

};

MAP\_TYPE\_MUX\_GROUP类型的设置信息：

struct pinctrl\_setting\_mux {

unsigned group;//group selector

unsigned func;/function selector

};

MAP\_TYPE\_CONFIGS\_\*类型的设置信息：

struct pinctrl\_setting\_configs {

unsigned group\_or\_pin;//group selector或者pin ID

unsigned long \*configs;//需要写入HW中的配置值列表

unsigned num\_configs;//列表中值的个数

};

# pinctrl注册

填充好pinctrl\_desc结构后，就可以注册pinctrl了：

struct pinctrl\_dev \*pinctrl\_register(struct pinctrl\_desc \*pctldesc, struct device \*dev, void \*driver\_data)

{

//初始化pin控制器设备

pctldev = pinctrl\_init\_controller(pctldesc, dev, driver\_data);

ret = pinctrl\_register\_pins(pctldev, pctldesc->pins, pctldesc->npins);

list\_add\_tail(&pctldev->node, &pinctrldev\_list);

}

将所有pinctrl\_pin\_desc遍历注册：

static int pinctrl\_register\_pins(struct pinctrl\_dev \*pctldev, struct pinctrl\_pin\_desc const \*pins, unsigned num\_descs)

{

for (i = 0; i < num\_descs; i++) {

ret = pinctrl\_register\_one\_pin(pctldev, &pins[i]);

}

根据pinctrl\_pin\_desc引脚描述信息得到struct pin\_desc，添加到pctldev->pin\_desc\_tree进行统一管理：

static int pinctrl\_register\_one\_pin(struct pinctrl\_dev \*pctldev,

const struct pinctrl\_pin\_desc \*pin)

{

struct pin\_desc \*pindesc;

pindesc = pin\_desc\_get(pctldev, pin->number);

pindesc = kzalloc(sizeof(\*pindesc), GFP\_KERNEL);

if (pin->name) {

pindesc->name = pin->name;

} else {

pindesc->name = kasprintf(GFP\_KERNEL, "PIN%u", pin->number);

pindesc->dynamic\_name = true;

}

pindesc->drv\_data = pin->drv\_data;

radix\_tree\_insert(&pctldev->pin\_desc\_tree, pin->number, pindesc);

}

pinctrl的注销：

void pinctrl\_unregister(struct pinctrl\_dev \*pctldev);

带有资源管理的注册与注销方法：

struct pinctrl\_dev \*devm\_pinctrl\_register(struct device \*dev, struct pinctrl\_desc \*pctldesc, void \*driver\_data);

void devm\_pinctrl\_unregister(struct device \*dev, struct pinctrl\_dev \*pctldev);

# pinctrl映射

pinctrl的每个功能对应的引脚复用和配置信息由如下结构表示：

struct pinctrl\_map {

const char \*dev\_name;//使用此mapping的设备名

const char \*name;//该mapping的名称

enum pinctrl\_map\_type type;

const char \*ctrl\_dev\_name;//控制此mapping的设备名称

union {

struct pinctrl\_map\_mux mux;//mux设置

struct pinctrl\_map\_configs configs;//config设置

} data;

};

pinctrl\_map的注册在内核不支持设备树时，通过调用pinctrl\_register\_mappings()/pinctrl\_register\_map()接口完成，如果内核支持设备树，则调用pinctrl\_dt\_to\_map()完成，当然他最终也是调用了pinctrl\_register\_map()。pinctrl\_register\_map()会将pinctrl\_map信息添加到全局链表pinctrl\_maps中。

# pinctrl使用

使用pinctrl当然就是设置pin的state。首先需要先获取struct pinctrl，可以通过devm\_pinctrl\_get()和pinctrl\_get()获取：

struct pinctrl \*pinctrl\_get(struct device \*dev)

{

struct pinctrl \*p;

p = find\_pinctrl(dev);

return create\_pinctrl(dev, NULL);

}

如果dev没有创建过pinctrl就需要进行创建：

static struct pinctrl \*create\_pinctrl(struct device \*dev, struct pinctrl\_dev \*pctldev)

{

struct pinctrl \*p;

p = kzalloc(sizeof(\*p), GFP\_KERNEL);

INIT\_LIST\_HEAD(&p->states);

INIT\_LIST\_HEAD(&p->dt\_maps);

ret = pinctrl\_dt\_to\_map(p, pctldev);

for\_each\_maps(maps\_node, i, map) {//遍历pinctrl\_maps

if (strcmp(map->dev\_name, devname))

continue;

//根据map信息将setting信息添加到pinctrl->state->settings

ret = add\_setting(p, pctldev, map);

}

list\_add\_tail(&p->node, &pinctrl\_list);

}

获取到struct pinctrl后在设置state前还需要检查是否存在此state:

struct pinctrl\_state \*pinctrl\_lookup\_state(struct pinctrl \*p, const char \*name)

{

list\_for\_each\_entry(state, &p->states, node)

if (!strcmp(state->name, name))

return state;

}

最后就可以设置pin的state了：

int pinctrl\_select\_state(struct pinctrl \*p, struct pinctrl\_state \*state)

{

//遍历该状态下的所有设置，逐一设置

list\_for\_each\_entry(setting, &state->settings, node) {

switch (setting->type) {

case PIN\_MAP\_TYPE\_MUX\_GROUP:

ret = pinmux\_enable\_setting(setting);

break;

case PIN\_MAP\_TYPE\_CONFIGS\_PIN:

case PIN\_MAP\_TYPE\_CONFIGS\_GROUP:

ret = pinconf\_apply\_setting(setting);

break;

}

}

# pinctrl与device的bind

上面介绍了pinctrl的映射和使用，其实这部分工作一般无需我们参与，设备驱动模型已经处理了pin的设置，驱动和设备匹配过程以前已经讲过，匹配成功后会执行driver\_probe\_device将设备和驱动bind，内部会调用pinctrl\_bind\_pins():

int driver\_probe\_device(struct device\_driver \*drv, struct device \*dev)

{

ret = really\_probe(dev, drv);

ret = pinctrl\_bind\_pins(dev);

}

pinctrl\_bind\_pins其实就是上面讲的pinctrl使用过程，但是它会获取多种pin state，在设备切换到不同状态时进行不同的设置：

int pinctrl\_bind\_pins(struct device \*dev)

{

dev->pins = devm\_kzalloc(dev, sizeof(\*(dev->pins)), GFP\_KERNEL);

dev->pins->p = devm\_pinctrl\_get(dev);

dev->pins->default\_state = pinctrl\_lookup\_state(dev->pins->p,

PINCTRL\_STATE\_DEFAULT);

dev->pins->init\_state = pinctrl\_lookup\_state(dev->pins->p,

PINCTRL\_STATE\_INIT);

ret = pinctrl\_select\_state(dev->pins->p, dev->pins->default\_state);

dev->pins->sleep\_state = pinctrl\_lookup\_state(dev->pins->p,

PINCTRL\_STATE\_SLEEP);

dev->pins->idle\_state = pinctrl\_lookup\_state(dev->pins->p,

PINCTRL\_STATE\_IDLE);

}

# pinctrl与gpio

gpio和i2c/spi等这些接口一样都是pinctrl的consumer，但是gpio又有所不同，上面讲的pinctrl与device的bind，如果是i2c/spi等控制器可以在驱动probe时进行pin的bind，但是gpio控制器不能这么做，比如一个gpio控制器能控制32个pin，我们不能直接将这32个pin直接bind，因为这32个pin一般只有部分作为gpio，所以pinctrl对于gpio的处理还是有一些特殊之处的。

我们在实现gpio驱动时需要向pinctrl申请pin：

int pinctrl\_request\_gpio(unsigned gpio);

void pinctrl\_free\_gpio(unsigned gpio);

如果pinctrl实现了设置gpio方向的回调，还可以使用如下函数设置gpio方向：

int pinctrl\_gpio\_direction\_input(unsigned gpio);

int pinctrl\_gpio\_direction\_output(unsigned gpio);

但是注意gpio number和pinctrl的pin number是不一致的，所以还需要进行两种number的map，map关系通过如下结构表示，也就是gpio controller(gc)的gpio[base]到gpio[base+npins- 1]和pin controller的pin[pin\_base]到pin[pin\_base+npins-1]是一一对应的。

struct pinctrl\_gpio\_range {

struct list\_head node;

const char \*name;

unsigned int id;

unsigned int base;

unsigned int pin\_base;

unsigned const \*pins;

unsigned int npins;

struct gpio\_chip \*gc;

};

设备树中在gpio controller的节点中需要添加如下信息来说明map关系，gpio-ranges可以包含一个或多个，如下信息表明gpio controller的gpio0~9和pinctrl1中的pin10~19一一对应，gpio10~29和pinctrl2中的pin30~49一一对应。

gpio-ranges = <&pinctrl1 0 10 10>, <&pinctrl2 10 30 20>;

gpiochip\_add\_pin\_range()实现了两种number的map，其会在我们注册gpio\_chip时自动调用，过程如下：

gpiochip\_add\_data

of\_gpiochip\_add

of\_gpiochip\_add\_pin\_range

gpiochip\_add\_pin\_range

pinctrl\_find\_and\_add\_gpio\_range

pinctrl\_add\_gpio\_range

list\_add\_tail(&range->node, &pctldev->gpio\_ranges);

# 示例说明

示例虚拟的pinctrl驱动只控制4个pin，这4个pin的复用关系如下：

|  |  |  |  |
| --- | --- | --- | --- |
| pin0 | gpio0 | i2c0 | spi0 |
| pin1 | gpio1 |
| pin2 | gpio2 | i2c1 |
| pin3 | gpio3 |

virt-pinctrl.c是虚拟的pinctrl驱动，对应的设备树信息如下：

virt\_pinctrl: pinctrl{

compatible = "xm,virt-pinctrl";

pinctrl\_i2c0\_default: i2c0-default {

mux {

groups = "i2c0\_group";

function = "i2c0";

};

};

pinctrl\_i2c1\_default: i2c1-default {

mux {

groups = "i2c1\_group";

function = "i2c1";

};

};

pinctrl\_spi0\_default: spi0-default {

mux {

groups = "spi0\_group";

function = "spi0";

};

};

};

pinctrl\_consumer.c没有具体的功能，只是为了匹配driver和device从而完成pin的bind，如果设备树中添加pinctrl-consumer0、pinctrl-consumer1、pinctrl-consumer2中的一个都会得到如下截图中的信息：

pinctrl-consumer0 {

compatible = "xm,pinctrl-consumer";

pinctrl-names = "default";

pinctrl-0 = <&pinctrl\_i2c0\_default>;

};

pinctrl-consumer1 {

compatible = "xm,pinctrl-consumer";

pinctrl-names = "default";

pinctrl-0 = <&pinctrl\_i2c1\_default>;

};

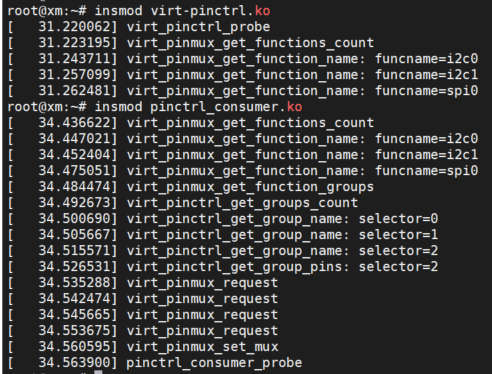
pinctrl-consumer2 {

compatible = "xm,pinctrl-consumer";

pinctrl-names = "default";

pinctrl-0 = <&pinctrl\_spi0\_default>;

};



pinctrl-consumer0和pinctrl-consumer1是可以同时存在的，但是都不能与pinctrl-consumer2共存，因为会出现

引脚冲突：



virt\_gpiochip.c是在以前的虚拟gpio控制器驱动基础上将request/free改为从pinctrl申请/释放，设备树信息如下：

virt\_gpiochip: virt\_gpiochip@0 {

#gpio-cells = <2>;

compatible = "xm,virt-gpiochip";

gpio-controller;

gpio-ranges = <&virt\_pinctrl 0 0 4>;

};

pinctrl\_consumer.c在判断到 compatible为”xm,pinctrl-consumer-gpio”时会获取gpio并设置gpio的value：

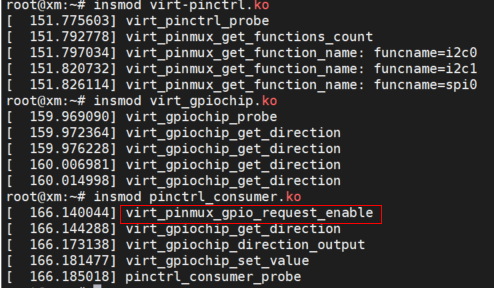
pinctrl-consumer3 {

compatible = "xm,pinctrl-consumer-gpio";

pinctrl-names = "default";

gpios = <&virt\_gpiochip 0 0>;

};



同理，因为申请的是gpio0，设备树中不能存在pinctrl-consumer0和pinctrl-consumer2，否则会出现引脚冲突：

