linux内核中的GPIO系统之（2）：pin control subsystem

作者：linuxer 发布于：2014-7-26 18:24 分类：GPIO子系统

一、前言

在linux2.6内核上工作的嵌入式软件工程师在pin control上都会遇到这样的状况：

（1）启动一个新的项目后，需要根据硬件平台的设定进行pin control相关的编码。例如：在bootloader中建立一个大的table，描述各个引脚的配置和缺省状态。此外，由于SOC的引脚是可以复用的，因此在各个具体的driver中，也可能会对引脚进行的配置。这些工作都是比较繁琐的工作，需要极大的耐心和细致度。

（2）发现某个driver不能正常工作，辛辛苦苦debug后发现仅仅是因为其他的driver在初始化的过程中修改了引脚的配置，导致自己的driver无法正常工作

（3）即便是主CPU是一样的项目，但是由于外设的不同，我们也不能使用一个kernel image，而是必须要修改代码（这些代码主要是board-specific startup code）

（4）代码不是非常的整洁，cut-and-pasted代码满天飞，linux中的冗余代码太多

作为一个嵌入式软件工程师，项目做多了，接触的CPU就多了，摔的跤就多了，之后自然会去思考，我们是否可以解决上面的问题呢？此外，对于基于ARM core那些SOC，虽然表面上看起来各个SOC各不相同，但是在pin control上还有很多相同的内容的，是否可以把它抽取出来，进行进一步的抽象呢？新版本中的内核（本文以3.14版本内核为例）提出了pin control subsystem来解决这些问题。

二、pin control subsystem的文件列表

1、源文件列表

我们整理linux/drivers/pinctrl目录下pin control subsystem的源文件列表如下：

文件名	描述
core.c core.h	pin control subsystem的core driver
pinctrl-utils.c pinctrl-utils.h	pin control subsystem的一些utility接口函数
pinmux.c pinmux.h	pin control subsystem的core driver(pin muxing部分的代码，也称为pinmux driver)
pinconf.c pinconf.h	pin control subsystem的core driver(pin config部分的代码，也称为pin config driver)
devicetree.c devicetree.h	pin control subsystem的device tree代码
pinctrl-xxxx.c	各种pin controller的low level driver。

在pin controller driver文档中，我们以2416的pin controller为例，描述了一个具体的low level的driver，这个driver涉及的文件包括pinctrl-samsung.c，pinctrl-samsung.h和pinctrl-s3c24xx.c。

2、和其他内核模块接口头文件

很多内核的其他模块需要用到pin control subsystem的服务，这些头文件就定义了pin control subsystem的外部接口以及相关的数据结构。我们整理linux/include/linux/pinctrl目录下pin control subsystem的外部接口头文件列表如下：

文件名	描述
consumer.h	其他的driver要使用pin control subsystem的下列接口： a、设置引脚复用功能 b、配置引脚的电气特性 这时候需要include这个头文件
devinfo.h	这是for linux内核的驱动模型模块（driver model）使用的接口。struct device中包括了一个struct dev_pin_info *pins的成员，这个成员描述了该设备的引脚的初始状态信息，在probe之前，driver model中的core driver在调用driver的probe函数之前会先设定pin state
machine.h	和machine模块的接口。

站内搜索

请输入关键词 搜索

功能

留言板
评论列表
支持者列表

最新评论

callme_friend
@pixiandouban：visio
callme_friend
@hit201j：太忙，没时间回复，需要的朋友可统一去以下链...
callme_friend
@icecoder：太忙，没时间回复，需要的朋友可统一去以下...
callme_friend
@就爱吃泡芙：太忙，没时间回复，需要的朋友可统一去以下链接下...
callme_friend
@今雨轩：太忙，没时间回复，需要的朋友可统一去以下链接下载：...
callme_friend
@xtydtc：太忙，没时间回复，需要的朋友可统一去以下链接...

文章分类

Linux内核分析(11)
统一设备模型(15)
电源管理子系统(42)
中断子系统(15)
进程管理(19)
内核同步机制(18)
GPIO子系统(5)
时间子系统(14)
通信类协议(7)
内存管理(27)
图形子系统(1)
文件系统(4)
TTY子系统(6)
u-boot分析(3)
Linux应用技巧(13)
软件开发(6)
基础技术(13)
蓝牙(16)
ARMv8A Arch(13)
显示(3)
USB(1)
基础学科(10)
技术漫谈(12)
项目专区(0)
X Project(28)

随机文章

linux内核中的GPIO系统之（4）：pinctrl驱动的理解和总结

3、Low level pin controller driver接口

我们整理linux/include/linux/pinctrl目录下pin control subsystem提供给底层specific pin controller driver的头文件列表如下：

文件名	描述
pinconf-generic.h	这个接口主要是提供给各种pin controller driver使用的，不是外部接口。
pinconf.h	pin configuration 接口
pinctrl-state.h	pin control state状态定义
pinmux.h	pin mux function接口

三、pin control subsystem的软件框架图

1、功能和接口概述

一般而言，学习复杂的软件组件或者软件模块是一个痛苦的过程。我们可以把我们要学习的那个软件block看成一个黑盒子，不论里面有多么复杂，第一步总是先了解其功能和外部接口特性。如果你愿意，你可以不去看其内部实现，先自己思考其内部逻辑，并形成若干问题，然后带着这些问题去看代码，往往事半功倍。

（1）、功能规格。pin control subsystem的主要功能包括：

（A）管理系统中所有可以控制的pin。在系统初始化的时候，枚举所有可以控制的pin，并标识这些pin。

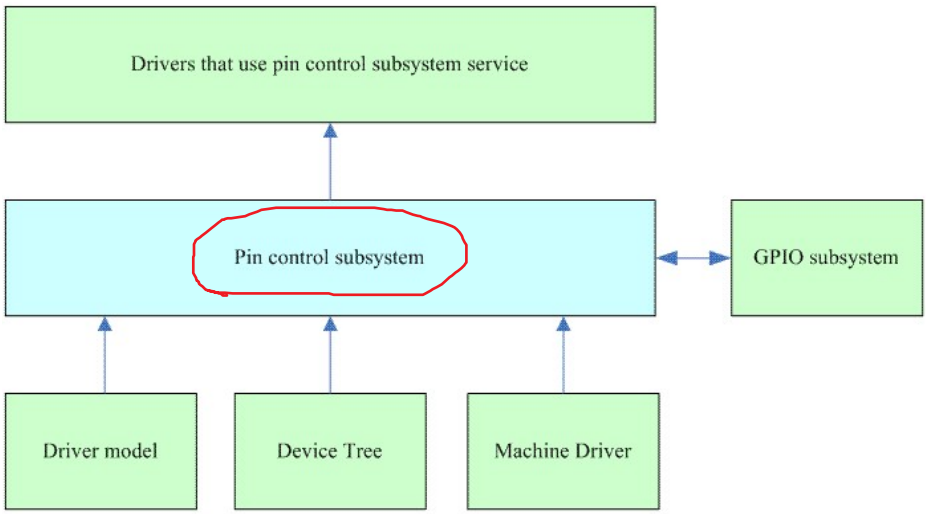
（B）管理这些pin的复用（Multiplexing）。对于SOC而言，其引脚除了配置成普通GPIO之外，若干个引脚还可以组成一个pin group，形成特定的功能。例如pin number是{ 0, 8, 16, 24 }这四个引脚组合形成一个pin group，提供SPI的功能。pin control subsystem要管理所有的pin group。

（C）配置这些pin的特性。例如配置该引脚上的pull-up/down电阻，配置drive strength等

（2）接口规格。linux内核的某个软件组件必须放回到linux系统中才容易探讨它的接口以及在系统中的位置，因此，在本章的第二节会基于系统block上描述各个pin control subsystem和其他内核模块的接口。

（3）内部逻辑。要研究一个subsystem的内部逻辑，首先要打开黑盒子，细分模块，然后针对每一个模块进行功能分析、外部接口分析、内部逻辑分析。如果模块还是比较大，难于掌握，那么就继续细分，拆成子模块，重复上面的分析过程。在本章的第三节中，我们打开pin control subsystem的黑盒子进行进一步的分析。

2、pin control subsystem在和其他linux内核模块的接口关系图如下图所示：



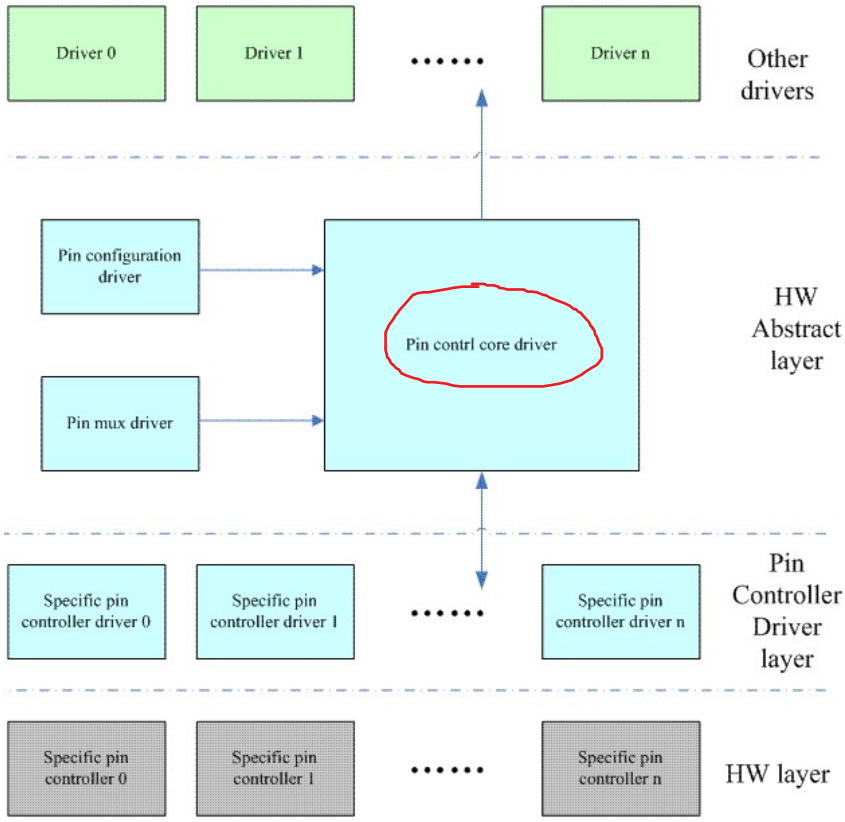
pin control subsystem会向系统中的其他driver提供接口以便进行该driver的pin config和pin mux的设定，这部分的接口在第四章描述。理想的状态是GPIO controll driver也只是象UART,SPI这样driver一样和pin control subsystem进行交互，但是，实际上由于各种源由（后文详述），pin control subsystem和GPIO subsystem必须有交互，这部分的接口在第五章描述。第六章描述了Driver model和pin control subsystem的接口，第七章描述了为Pin control subsystem提供database支持的Device Tree和Machine driver的接口。

3、pin control subsystem内部block diagram

Linux时间子系统之（一）：时间的基本概念
蓝牙协议分析(5)_BLE广播通信相关的技术分析
Device Tree（三）：代码分析
MinGW下安装man工具包

文章存档

2018年10月(1)
2018年8月(1)
2018年6月(1)
2018年5月(1)
2018年4月(7)
2018年2月(4)
2018年1月(5)
2017年12月(2)
2017年11月(2)
2017年10月(1)
2017年9月(5)
2017年8月(4)
2017年7月(4)
2017年6月(3)
2017年5月(3)
2017年4月(1)
2017年3月(8)
2017年2月(6)
2017年1月(5)
2016年12月(6)
2016年11月(11)
2016年10月(9)
2016年9月(6)
2016年8月(9)
2016年7月(5)
2016年6月(8)
2016年5月(8)
2016年4月(7)
2016年3月(5)
2016年2月(5)
2016年1月(6)
2015年12月(6)
2015年11月(9)
2015年10月(9)
2015年9月(4)
2015年8月(3)
2015年7月(7)
2015年6月(3)
2015年5月(6)
2015年4月(9)
2015年3月(9)
2015年2月(6)
2015年1月(6)
2014年12月(17)
2014年11月(8)
2014年10月(9)
2014年9月(7)
2014年8月(12)
2014年7月(6)
2014年6月(6)
2014年5月(9)
2014年4月(9)
2014年3月(7)
2014年2月(3)
2014年1月(4)



起始理解了接口部分内容，阅读和解析pin control subsystem的内部逻辑已经很简单，本文就不再分析了。

四、pin control subsystem向其他driver提供的接口

当你准备撰写一个普通的linux driver（例如串口驱动）的时候，你期望pin control subsystem提供的接口是什么样子的？简单，当然最好是简单的，最最好是没有接口，当然这是可能的，具体请参考第六章的接口。

1、概述

普通driver调用pin control subsystem的主要目标是：

（1）设定该设备的功能复用。设定设备的功能复用需要了解两个概念，一个是function，另外一个pin group。function是功能抽象，对应一个HW逻辑block，例如SPI0。虽然给定了具体的gunction name，我们并不能确定其使用的pins的情况。例如：为了设计灵活，芯片内部的SPI0的功能可能引出到pin group { A8, A7, A6, A5 }，也可能引出到另外一个pin group{ G4, G3, G2, G1 }，但毫无疑问，这两个pin group不能同时active，毕竟芯片内部的SPI0的逻辑功能电路只有一个。 因此，只有给出function selector（所谓selector就是一个ID或者index）以及function的pin group selector才能进行function mux的设定。

（2）设定该device对应的那些pin的电气特性。

此外，由于电源管理的要求，某个device可能处于某个电源管理状态，例如idle或者sleep，这时候，属于该device的所有的pin就会需要处于另外的状态。综合上述的需求，我们把定义了pin control state的概念，也就是说设备可能处于非常多的状态中的一个，device driver可以切换设备处于的状态。为了方便管理pin control state，我们又提出了一个pin control state holder的概念，用来管理一个设备的所有的pin control状态。因此普通driver调用pin control subsystem的接口从逻辑上将主要是：

（1）获取pin control state holder的句柄

（2）设定pin control状态

（3）释放pin control state holder的句柄

pin control state holder的定义如下：

```
struct pinctrl {
    struct list_head node; - - 系统中的所有device的pin control state holder被挂入到了一个全局链表中
    struct device *dev; - - - 该pin control state holder对应的device
    struct list_head states; - - - - 该设备的所有的状态被挂入到这个链表中
    struct pinctrl_state *state; - - - 当前的pin control state
    struct list_head dt_maps; - - - - mapping table
```

```
    struct kref users; - - - - - - reference count
};
```

系统中的每一个需要和pin control subsystem进行交互的设备在进行设定之前都需要首先获取这个句柄。而属于该设备的所有的状态都是挂入到一个链表中，链表头就是pin control state holder的states成员，一个state的定义如下：

```
struct pinctrl_state {
    struct list_head node; - - - 挂入链表头的节点
    const char *name; - - - - - 该state的名字
    struct list_head settings; - - - 属于该状态的所有的settings
};
```

一个pin state包含若干个setting，所有的settings被挂入一个链表中，链表头就是pin state中的settings成员，定义如下：

```
struct pinctrl_setting {
    struct list_head node;
    enum pinctrl_map_type type;
    struct pinctrl_dev *pctldev;
    const char *dev_name;
    union {
        struct pinctrl_setting_mux mux;
        struct pinctrl_setting_configs configs;
    } data;
};
```

当driver设定一个pin state的时候，pin control subsystem内部会遍历该state的settings链表，将一个一个的setting进行设定。这些settings有各种类型，定义如下：

```
enum pinctrl_map_type {
    PIN_MAP_TYPE_INVALID,
    PIN_MAP_TYPE_DUMMY_STATE,
    PIN_MAP_TYPE_MUX_GROUP, - - - 功能复用的setting
    PIN_MAP_TYPE_CONFIGS_PIN, - - - - 设定单一一个pin的电气特性
    PIN_MAP_TYPE_CONFIGS_GROUP, - - - - 设定单pin group的电气特性
};
```

有pin mux相关的设定（PIN_MAP_TYPE_MUX_GROUP），定义如下：

```
struct pinctrl_setting_mux {
    unsigned group; - - - - - - - - 该setting所对应的group selector
    unsigned func; - - - - - - - - - 该setting所对应的function selector
};
```

有了function selector以及属于该functiong的roup selector就可以进行该device和pin mux相关的设定了。设定电气特性的settings定义如下：

```
struct pinctrl_map_configs {
    const char *group_or_pin; - - - - 该pin或者pin group的名字
    unsigned long *configs; - - - - 要设定的值的列表。这个值被用来写入HW
    unsigned num_configs; - - - - 列表中值的个数
};
```

2、具体的接口

（1）devm_pinctrl_get和pinctrl_get。devm_pinctrl_get是Resource managed版本的pinctrl_get，核心还是pinctrl_get函数。这两个接口都是获取设备（设备模型中的struct device）的pin control state holder（struct pinctrl）。pin control state holder不是静态定义的，一般在第一次调用该函数的时候会动态创建。创建一个pin control state holder是一个大工程，我们分析一下这段代码：

```
static struct pinctrl *create_pinctrl(struct device *dev)
{

   分配pin control state holder占用的内存并初始化
    p = kzalloc(sizeof(*p), GFP_KERNEL);
    p->dev = dev;
    INIT_LIST_HEAD(&p->states);
    INIT_LIST_HEAD(&p->dt_maps);

mapping table这个database的建立也是动态的，当第一次调用pin control state holder的get函数的时候，就会通过调用pinctrl_dt_to_map来建立该device需要的mapping entry。具体请参考第七章。

    ret = pinctrl_dt_to_map(p);

    devname = dev_name(dev);

    mutex_lock(&pinctrl_maps_mutex);
    for_each_maps(maps_node, i, map) {
        /* Map must be for this device */
        if (strcmp(map->dev_name, devname))
            continue;

        ret = add_setting(p, map); - - - - 分析一个mapping entry，把这个setting的代码加入到holder中

    }
    mutex_unlock(&pinctrl_maps_mutex);

    kref_init(&p->users);

    /* 把这个新增加的pin control state holder加入到全局链表中 */
    mutex_lock(&pinctrl_list_mutex);
    list_add_tail(&p->node, &pinctrl_list);
    mutex_unlock(&pinctrl_list_mutex);

    return p;
}
```

（2）devm_pinctrl_put和pinctrl_put。是（1）接口中的逆函数。devm_pinctrl_get和pinctrl_get获取句柄的时候申请了很多资源，在devm_pinctrl_put和pinctrl_put可以释放。需要注意的是多次调用get函数不会重复分配资源，只会reference count加一，在put中referrenct count减一，当count = = 0的时候才释放该device的pin control state holder持有的所有资源。

（3）pinctrl_lookup_state。根据state name在pin control state holder找到对应的pin control state。具体的state是各个device自己定义的，不过pin control subsystem自己定义了一些标准的pin control state，定义在pinctrl-state.h文件中：

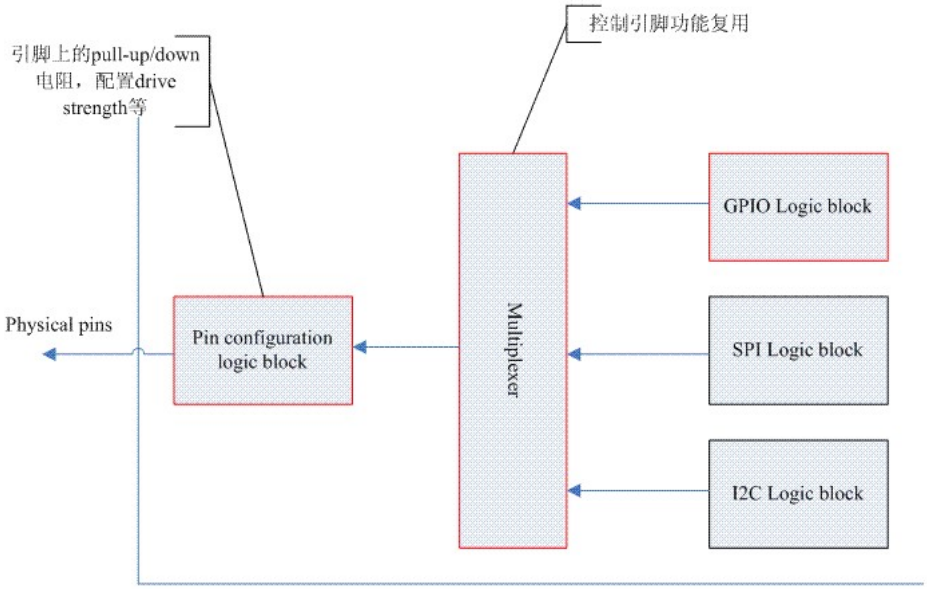
```
#define PINCTRL_STATE_DEFAULT "default"
#define PINCTRL_STATE_IDLE "idle"
#define PINCTRL_STATE_SLEEP "sleep"
```

（4）pinctrl_select_state。设定一个具体的pin control state接口。

五、和GPIO subsystem交互

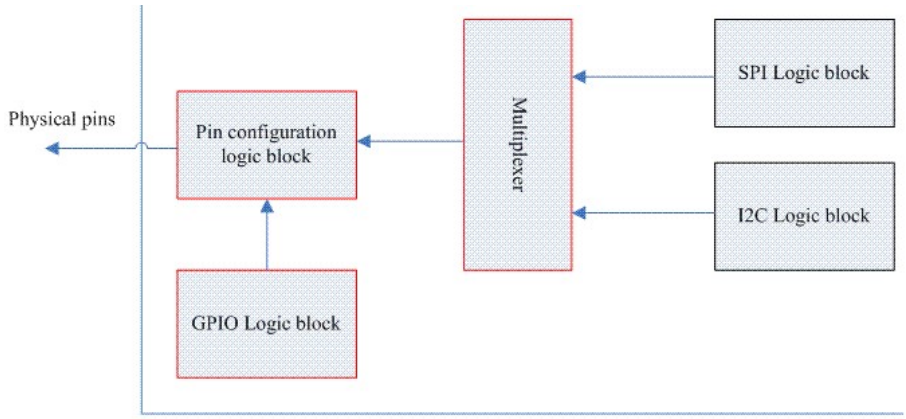
1、为何pin control subsystem要和GPIO subsystem交互？

作为软件工程师，我们期望的硬件设计应该如下图所示：



GPIO的HW block应该和其他功能复用的block是对等关系的，它们共同输入到一个复用器block，这个block的寄存器控制哪一个功能电路目前是active的。pin configuration是全局的，不论哪种功能是active的，都可以针对pin进行电气特性的设定。这样的架构下，上图中红色边框的三个block是完全独立的HW block，其控制寄存器在SOC datasheet中应该是分成三个章节描述，同时，这些block的寄存器应该分别处于不同的地址区间。

对于软件工程师，我们可以让pin control subsystem和GPIO subsystem完全独立，各自进行初始化，各自映射自己的寄存器地址空间，对于pin control subsystem而言，GPIO和其他的HW block没有什么不同，都是使用自己提供服务的一个软件模块而已。然而实际上SOC的设计并非总是向软件工程师期望的那样，有的SOC的设计框架图如下：



这时候，GPIO block是alway active的，而红色边框的三个block是紧密的捆绑在一起，它们的寄存器占据了一个memory range（datasheet中用一个章节描述这三个block）。这时候，对于软件工程师来说就有些纠结了，本来不属于我的GPIO控制也被迫要参与进来。这时候，硬件寄存器的控制都是pin controller来处理，GPIO相关的操作都要经过pin controller driver，这时候，pin controller driver要作为GPIO driver的back-end出现。

2、具体的接口形态

（1）pinctrl_request_gpio。该接口主要用来申请GPIO。GPIO也是一种资源，使用前应该request，使用完毕后释放。具体的代码如下：

```
int pinctrl_request_gpio(unsigned gpio) - - - - 这里传入的是GPIO 的ID
{
    struct pinctrl_dev *pctldev;
    struct pinctrl_gpio_range *range;
    int ret;
    int pin;

    ret = pinctrl_get_device_gpio_range(gpio, &pctldev, &range); - - - A
    if (ret) {
        if (pinctrl_ready_for_gpio_range(gpio))
            ret = 0;
        return ret;
    }
```

```
    mutex_lock(&pctldev->mutex);
    pin = gpio_to_pin(range, gpio);  - - - 将GPIO ID转换成pin ID

    ret = pinmux_request_gpio(pctldev, range, pin, gpio);  - - - - - - B

    mutex_unlock(&pctldev->mutex);

    return ret;
}
```

毫无疑问，申请GPIO资源本应该是GPIO subsystem的责任，但是由于上一节描述的源由，pin control subsystem提供了这样一个接口函数供GPIO driver使用（其他的内核driver不应该调用，它们应该使用GPIO subsystem提供的接口）。多么丑陋的代码，作为pin control subsystem，除了维护pin space中的ID，还要维护GPIO 的ID以及pin ID和GPIO ID的关系。

A：根据GPIO ID找到该ID对应的pin control device（struct pinctrl_dev）和GPIO rang（pinctrl_gpio_range）。在core driver中，每个low level的pin controller device都被映射成一个struct pinctrl_dev，并形成链表，链表头就是pinctrldev_list。由于实际的硬件设计（例如GPIO block被分成若干个GPIO 的bank，每个bank就对应一个HW GPIO Controller Block），一个pin control device要管理的GPIO ID是分成区域的，每个区域用struct pinctrl_gpio_range来抽象，在low level 的pin controller初始化的时候（具体参考samsung_pinctrl_register的代码），会调用pinctrl_add_gpio_range将每个GPIO bank表示的gpio range挂入到pin control device的range list中（gpio_ranges成员）。pinctrl_gpio_range 的定义如下：

```
struct pinctrl_gpio_range {
    struct list_head node;
    const char *name;
    unsigned int id; - - - - - - - - - - - GPIO chip ID
    unsigned int base; - - - - - - 该range中的起始GPIO IDD
    unsigned int pin_base; - - - 在线性映射的情况下，这是起始的pin base
    unsigned const *pins; - - - 在非线性映射的时候，这是table是pin到GPIO的lookup table
    unsigned int npins; - - - - 这个range有多少个GPIO引脚
    struct gpio_chip *gc;------每个GPIO bank都是一个gpio chip，对应一个GPIO range
};
```

pin ID和GPIO ID有两种映射关系，一种是线性映射（这时候pin_base有效），也就是说，对于这个GPIO range，GPIO base ID是a，pin ID base是b，那么a<--->b，a + 1<--->b + 1，a + 2<--->b + 2，以此类推。对于非线性映射（pin_base无效，pins是有效的），我们需要建立一个lookup table，以GPIO ID为索引，可以找到对于的pin ID。

B：这里主要是进行复用功能的设定，毕竟GPIO也是引脚的一个特定的功能。pinmux_request_gpio函数的作用主要有两个，一个是在core driver中标记该pin已经用作GPIO了，这样，如果有模块后续request该资源，那么core driver可以拒绝不合理的要求。第二步就是调用底层pin controller driver的callback函数，进行底层寄存器相关的操作。

（2）pinctrl_free_gpio。有申请就有释放，这是pinctrl_request_gpio的逆函数

（3）pinctrl_gpio_direction_input和pinctrl_gpio_direction_output。为已经指定为GPIO功能的引脚设定方向，输入或者输出。代码很简单，不再赘述。

六、和驱动模型的接口

前文已经表述过，最好是让统一设备驱动模型（Driver model）来处理pin 的各种设定。与其自己写代码调用devm_pinctrl_get、pinctrl_lookup_state、pinctrl_select_state等pin control subsystem的接口函数，为了不让linux内核自己的框架处理呢。本章将分析具体的代码，这些代码实例对自己driver调用pin control subsystem的接口函数来设定本device的pin control的相关设定也是有指导意义的。 linux kernel中的驱动模型提供了driver和device的绑定机制，一旦匹配会调用probe函数如下：

```
static int really_probe(struct device *dev, struct device_driver *drv)
{
    ……
    ret = pinctrl_bind_pins(dev);  - - - 对该device涉及的pin进行pin control相关设定
    ……

    if (dev->bus->probe) { - - - - - - 下面是真正的probe过程
        ret = dev->bus->probe(dev);
        if (ret)
            goto probe_failed;
    } else if (drv->probe) {
        ret = drv->probe(dev);
```

```
        if (ret)
            goto probe_failed;
    }

……

}
```

pinctrl_bind_pins的代码如下：

```
int pinctrl_bind_pins(struct device *dev)
{
    int ret;

    dev->pins = devm_kzalloc(dev, sizeof(*(dev->pins)), GFP_KERNEL); - - - （1）

    dev->pins->p = devm_pinctrl_get(dev); - - - - - - - - - - - - - - - - - - （2）

    dev->pins->default_state = pinctrl_lookup_state(dev->pins->p, - - - - - - - （3）
                PINCTRL_STATE_DEFAULT);

    ret = pinctrl_select_state(dev->pins->p, dev->pins->default_state); - - - - - （4）


    dev->pins->sleep_state = pinctrl_lookup_state(dev->pins->p, - - - - - - （3）
                PINCTRL_STATE_SLEEP);

    dev->pins->idle_state = pinctrl_lookup_state(dev->pins->p, - - - - - - - （3）
                PINCTRL_STATE_IDLE);

    return 0;
}
```

（1）struct device数据结构有一个pins的成员，它描述了和该设备相关的pin control的信息，定义如下：

```
struct dev_pin_info {
    struct pinctrl *p; - - - - - - - - - - - - 该device对应的pin control state holder
    struct pinctrl_state *default_state; - - - - 缺省状态
    struct pinctrl_state *sleep_state; - - - - - 电源管理相关的状态
    struct pinctrl_state *idle_state; - - - - - 电源管理相关的状态
};
```

（2）调用devm_pinctrl_get获取该device对应的 pin control state holder句柄。

（3）搜索default state，sleep state，idle state并记录在本device中

（3）将该设备设定为pin default state

七、和device tree或者machine driver相关的接口

1、概述

device tree或者machine driver这两个模块主要是为 pin control subsystem提供pin mapping database的支持。这个database的每个entry用下面的数据结构表示：

```
struct pinctrl_map {
    const char *dev_name; - - - 使用这个mapping entry的设备名
    const char *name; - - - - - - 该名字表示了该mapping entry
    enum pinctrl_map_type type; - - - 这个entry的mapping type
    const char *ctrl_dev_name; - - - - - pin controller这个设备的名字
    union {
        struct pinctrl_map_mux mux;
        struct pinctrl_map_configs configs;
    } data;
};
```

2、通过machine driver静态定义的数据来建立pin mapping database

machine driver定义一个巨大的mapping table，描述，然后在machine初始化的时候，调用pinctrl_register_mappings将该table注册到pin control subsystem中。

3、通过device tree来建立pin mapping database

pin mapping信息定义在dts中，主要包括两个部分，一个是定义在各个具体的device node中，另外一处是定义在pin controller的device node中。

一个典型的device tree中的外设node定义如下（建议先看看pin controller driver的第二章关于dts的描述）：

```
device-node-name {
        定义该device自己的属性

        pinctrl-names = "sleep", "default";
        pinctrl-0 = ;
        pinctrl-1 = ;
    };
```

对普通device的dts分析在函数pinctrl_dt_to_map中，代码如下：

```
int pinctrl_dt_to_map(struct pinctrl *p)
{
    of_node_get(np);
    for (state = 0; ; state++) { - - - - - - - - - - - - - - - - - - - - （1）
        /* Retrieve the pinctrl-* property */
        propname = kasprintf(GFP_KERNEL, "pinctrl-%d", state);
        prop = of_find_property(np, propname, &size);
        kfree(propname);
        if (!prop)
            break;
        list = prop->value;
        size /= sizeof(*list); - - - - - - - - - - - - - - - （2）

        /* Determine whether pinctrl-names property names the state */
        ret = of_property_read_string_index(np, "pinctrl-names", - - - - - - - （3）
                        state, &statename);

        if (ret < 0) {
            /* strlen("pinctrl-") == 8 */
            statename = prop->name + 8; - - - - - - - - - - - - - - （4）
        }

        /* For every referenced pin configuration node in it */
        for (config = 0; config < size; config++) { - - - - - - - - - - - - （5）
            phandle = be32_to_cpup(list++);

            /* Look up the pin configuration node */
            np_config = of_find_node_by_phandle(phandle); - - - - - - - （6）

            /* Parse the node */
            ret = dt_to_map_one_config(p, statename, np_config); - - - - （7）
            of_node_put(np_config);
            if (ret < 0)
                goto err;
        }

        /* No entries in DT? Generate a dummy state table entry */
        if (!size) {
            ret = dt_remember_dummy_state(p, statename); - - - - - - - - （8）
            if (ret < 0)
                goto err;
        }
    }

    return 0;
```

```
err:
    pinctrl_dt_free_maps(p);
    return ret;
}
```

（1）pinctrl-0 pinctrl-1 pinctrl-2……表示了该设备的一个个的状态，这里我们定义了两个pinctrl-0和pinctrl-1分别对应sleep和default状态。这里每次循环分析一个pin state。

（2）代码执行到这里，size和list分别保存了该pin state中所涉及pin configuration phandle的数目以及phandle的列表

（3）读取从pinctrl-names属性中获取state name

（4）如果没有定义pinctrl-names属性，那么我们将pinctrl-0 pinctrl-1 pinctrl-2……中的那个ID取出来作为state name

（5）遍历一个pin state中的pin configuration list，这里的pin configuration实际应该是pin controler device node中的sub node，用phandle标识。

（6）用phandle作为索引，在device tree中找他该phandle表示的那个pin configuration

（7）分析一个pin configuration，具体下面会仔细分析

（8）如果该设备没有定义pin configuration，那么也要创建一个dummy的pin state。

这里我们已经进入对pin controller node下面的子节点的分析过程了。分析一个pin configuration的代码如下：

```
static int dt_to_map_one_config(struct pinctrl *p, const char *statename,
                struct device_node *np_config)
{
    struct device_node *np_pctldev;
    struct pinctrl_dev *pctldev;
    const struct pinctrl_ops *ops;
    int ret;
    struct pinctrl_map *map;
    unsigned num_maps;

    /* Find the pin controller containing np_config */
    np_pctldev = of_node_get(np_config);
    for (;;) {
        np_pctldev = of_get_next_parent(np_pctldev); - - - - - - - （1）
        if (!np_pctldev || of_node_is_root(np_pctldev)) {
            of_node_put(np_pctldev);
            return -EPROBE_DEFER;
        }
        pctldev = get_pinctrl_dev_from_of_node(np_pctldev); - - - - - （2）
        if (pctldev)
            break; - - - - - - - - - - - - - - - - - - - - - - - - - - （3）
        /* Do not defer probing of hogs (circular loop) */
        if (np_pctldev == p->dev->of_node) {
            of_node_put(np_pctldev);
            return -ENODEV;
        }
    }
    of_node_put(np_pctldev);

    /*
     * Call pinctrl driver to parse device tree node, and
     * generate mapping table entries
     */
    ops = pctldev->desc->pctlops;
    ret = ops->dt_node_to_map(pctldev, np_config, &map, &num_maps); - - - - （4）
    if (ret < 0)
        return ret;

    /* Stash the mapping table chunk away for later use */
    return dt_remember_or_free_map(p, statename, pctldev, map, num_maps); - - - - （5）
}
```

（1）首先找到该pin configuration node对应的parent node（也就是pin controler对应的node），如果找不到或者是root node，则进入出错处理。

（2）获取pin control class device

（3）一旦找到pin control class device则跳出for循环

（4）调用底层的callback函数处理pin configuration node。这也是合理的，毕竟很多的pin controller bindings是需要自己解析的。

（5）将该pin configuration node的mapping entry信息注册到系统中

八、core driver和low level pin controller driver的接口规格

pin controller描述符。每一个特定的pin controller都用一个struct pinctrl_desc来抽象，具体如下：

```
struct pinctrl_desc {
    const char *name;
    struct pinctrl_pin_desc const *pins;
    unsigned int npins;
    const struct pinctrl_ops *pctlops;
    const struct pinmux_ops *pmxops;
    const struct pinconf_ops *confops;
    struct module *owner;
};
```

pin controller描述符需要描述它可以控制多少个pin（成员npins），每一个pin的信息为何？（成员pins）。这两个成员就确定了一个pin controller所能控制的引脚的信息。

pin controller描述符中包括了三类操作函数：pctlops是一些全局的控制函数，pmxops是复用引脚相关的操作函数，confops操作函数是用来配置引脚的特性（例如：pull-up/down）。struct pinctrl_ops中各个callback函数的具体的解释如下：

callback函数	描述
get_groups_count	该pin controller支持多少个pin group。pin group的定义可以参考本文关于pin controller的功能规格中的描述。注意不要把pin group和IO port的硬件分组搞混了。例如：S3C2416有138个I/O 端口，分成11组，分别是gpa～gpl，这个组并不叫pin group，而是叫做pin bank。pin group是和特定功能（例如SPI、I2C）相关的一组pin。
get_group_name	给定一个selector（index），获取指定pin group的name
get_group_pins	给定一个selector（index），获取该pin group中pin的信息（该pin group包括多少个pin，每个pin的ID是什么）
pin_dbg_show	debug fs的callback接口
dt_node_to_map	分析一个pin configuration node并把分析的结果保存成mapping table entry，每一个entry表示一个setting（一个功能复用设定，或者电气特性设定）
dt_free_map	上面函数的逆函数

复用引脚相关的操作函数的具体解释如下：

call back函数	描述
request	pin control core进行具体的复用设定之前需要调用该函数，主要是用来请底层的driver判断某个引脚的复用设定是否是OK的。
free	是request的逆函数。调用request函数请求占用了某些pin的资源，调用free可以释放这些资源
get_functions_count	就是返回pin controller支持的function的数目
get_function_name	给定一个selector（index），获取指定function的name
get_function_groups	给定一个selector（index），获取指定function的pin groups信息
enable	enable一个function。当然要给出function selector和pin group的selector
disable	enable的逆函数
gpio_request_enable	request并且enable一个单独的gpio pin
gpio_disable_free	gpio_request_enable的逆函数
gpio_set_direction	设定GPIO方向的回调函数

配置引脚的特性的struct pinconf_ops数据结构的各个成员定义如下：

call back函数	描述
pin_config_get	给定一个pin ID以及config type ID，获取该引脚上指定type的配置。
pin_config_set	设定一个指定pin的配置
pin_config_group_get	以pin group为单位，获取pin上的配置信息
pin_config_group_set	以pin group为单位，设定pin group的特性配置
pin_config_dbg_parse_modify	debug接口
pin_config_dbg_show	debug接口
pin_config_group_dbg_show	debug接口

pin_config_config_dbg_show　　debug接口

原创文章，转发请注明出处。蜗窝科技。http://www.wowotech.net/linux_kenrel/pin-control-subsystem.html

标签: pin control

« Linux kernel的中断子系统之（六）：ARM中断处理过程 | Linux内核中的GPIO系统之（3）：pin controller driver代码分析»

评论：

随风律动
2017-07-19 16:27

去掉pinctrl_bind_pins的话，管脚复用怎么配置？

回复

wowo
2017-07-19 22:06

@随风律动：可以参考一下这篇文章：http://www.wowotech.net/gpio_subsystem/pinctrl-driver-summary.html

回复

thatwas
2016-12-30 15:34

我明白了，哈哈

回复

thatwas
2016-12-30 15:10

请问在解析pinctrl的group的信息的函数里有这么一行代码：
grp->pins[i] = bank->pin_base + (m.mux.goff - 0x0A) * 8 + m.mux.off;
grp->data[i].func = m.mode;
这个pins为什么要这么处理？？
拜托了，一直想不明白

回复

andy01011501
2016-02-22 15:10

有没有移动客户端啊？

回复

wowo
2016-02-22 16:02

@andy01011501：抱歉，还没有。曾经搞过一个微信公众号，但懒懒往上面贴文章的，也就没有公布出来。

回复

andy01011501
2016-02-23 16:06

@wowo：哦，你的网站很不错，顶一下！

回复

wowo
2016-02-24 08:45

@andy01011501：谢谢，欢迎常来~~

回复

cinmun
2015-07-29 11:45

貌似现在是Android天下了 纯嵌入式linux开发越来越少 pin control 在Android里貌似都是厂商自己定义的 不在这个目录下管理

回复

fucker
2015-07-27 10:08

linuxer 大神：

请接收小弟的膜拜~！

```
(__)
 /oo\\_________
 \   /          \---\
 \/        /    \    \
     \\_|___\\_|/      *
```

|| YY|
|| ||

回复

netilovefm

2015-07-25 21:44

hi, I want to ask something // 金

```
for (config = 0; config < size; config++) { - - - - - - - - - - - - (5)
    phandle = be32_to_cpup(list++);
    /* Look up the pin configuration node */
    np_config = of_find_node_by_phandle(phandle); - - - - - - (6)
```

in (6) , usually find gpio controller or processor's gpio controller node is it right ?
in this node, has pin's configure infotrmation is it right?

spi,i2c etc device driver calls pinctrl_dt_to_map() or pinctrl_get() once in their driver source ?
I confused. pinctrl_register() is for gpio controller and is called first , after calling pinctrl_register() , spi,i2c etc other driver source call pinctrl_get(->pincrl_dt_to_map)
then pctl_dev is one. spi, i2c etc uses this pctl_dev is it rigtht?

lastly

```
np_pctldev = of_get_next_parent(np_pctldev); - - - - - - - (1)
```

what is usually np_pctldev's parent ? I try to find this , but I can't find ....

answer this question , using chinese or english plz

回复

linuxer

2015-07-27 10:46

@netilovefm：Sorry, we want to focus on the following items recently:
1. power management
2. device model
3. time subsystem
4. interrupt subsystem

so, I have not too much time to help answer your GPIO subsystem question, maybe others can help.....

回复

随风

2015-07-25 09:02

您好，请问这两个结构体：

```
struct pinctrl_setting_mux {
    unsigned group;
    unsigned func;
};
struct pinctrl_setting_configs {
    unsigned group_or_pin;
    unsigned long *configs;
    unsigned num_configs;
};
```

和这两个结构体：

```
struct pinctrl_map_mux {
    const char *group;
    const char *function;
};
struct pinctrl_map_configs {
    const char *group_or_pin;
    unsigned long *configs;
    unsigned num_configs;
};
```

有什么关联？看起来成员都差不多，谢谢

回复

Bruce

2017-11-20 10:41

@随风：pinctrl_map_mux 是直接从dts解析出来的，group和function直接用dts里的字符表示；
pinctrl_setting_mux 是通过pinctrl_map转换而来的，group和function用index表示，作为pinctrl的pin state 直接配置数据，在配置的时候在全局group和function数据库中很方便通过index获取对应的数据。

pinctrl_map是dts的代码表示
pinctrl_setting是pin state配置的代码表示

以上仅是个人理解，欢迎讨论!

回复

SleepDeXiang
2015-06-10 11:58

呀，刷屏了，不好意思楼主 网不好 我不是来踢馆的

回复

> **wowo**
> 2015-06-10 12:19
>
> @SleepDeXiang：没关系，我已经删掉多余的了。
> PS：好久没见过这么欢乐的工程师了啊，看到评论后笑的我肚子疼啊，哈哈。也希望@常山黄豆和@tong 两位莫介意哈~~
>
> 回复

odriver
2015-05-22 17:29

博主您好！我有一块arm开发板（zynq芯片）。但是我看它的linux系统中没有使用pinctrl这个子系统。在这种情况下您觉得linux系统会怎么处理引脚复用这部分功能呢？我一直没有找到这个linux系统配置引脚复用的代码。希望能给点指导，谢谢！

回复

1 2

发表评论：

昵称

邮件地址 (选填)

个人主页 (选填)

发表评论