全志科技
Allwinner Technology

Tina3.0

PINCTRL 模块使用文档 v1.0

文档履历

版本号	日期	制/修订人	制/修订记录
V1.0	1028/5/18		初始版本

目　录

1. 概述

1.1. 编写目的

本文档对 Linux3.4/Linux3.10/Linux4.4/linux4.9 平台的 GPIO 接口使用进行详细的阐述，让用户明确掌握 GPIO 配置、申请等操作的编程方法。

1.2. 适用范围

本文档适用于 R6/R16/R18/R30/R40/R58/R311 平台。

1.3. 相关人员

本文档适用于所有需要在 R6/R16/R18/R30/R40/R58/R311 平台上开发设备驱动的人员。

2. 模块介绍

Pinctrl 框架是 linux 系统为统一各 SOC 厂商 pin 管理，避免各 SOC 厂商各自实现相同 pin 管理子系统而提出的。目的是为了减少 SOC 厂商系统移植工作量。

2.1. 模块功能介绍

许多 SoC 内部都包含 pin 控制器，通过 pin 控制器，我们可以配置一个或一组引脚的功能和特性。在软件上，Linux 内核 pinctrl 驱动可以操作 pin 控制器为我们完成如下工作：

1. 枚举并且命名 pin 控制器可控制的所有引脚；
2. 提供引脚的复用能力
3. 提供配置引脚的能力，如驱动能力、上拉下拉、数据属性等。
4. 与 gpio 子系统的交互
5. 实现 pin 中断

2.2. 相关术语介绍

术语	介绍
sunxi	Allwinner 的 SOC 硬件平台
Pincontroller	是对硬件模块的软件抽象，通常用来表示硬件控制器。能够处理引脚复用、属性配置等功能
Pin	根据芯片不同的封装方式，可以表现为球形、针型等。软件上采用常用一组无符号的整数[0-maxpin]来表示
Pin groups	外围设备通常都不只一个引脚，比如 SPI，假设接在 soc 的{0,8,16,24}管脚，而另一个设备 I2C 接在 SOC 的{24,25}管脚。我们可以说这里有两个 pin groups。很多控制器都需要处理 pin groups。因此管脚控制器子系统需要一个机制用来枚举管脚组且检索一个特定组中实际枚举的管脚
Pinconfig	管脚可以被软件配置成多种方式，多数与它们作为输入/输出时的电气特性相关。例如，可以设置一个输出管脚处于高阻状态，或是“三态”（意味着它被有效地断开连接）。或者可以通过设置将一个输入管脚与 VDD 或 GND 相连(上拉/下拉)，以便在没有信号驱动管脚时可以有个确定的值
Pinmux	引脚复用功能，使用一个特定的物理管脚（ball/pad/finger/等等）进行多种扩展复用，以支持不同功能的电气封装的习惯
Device tree	犹如它的名字，是一颗包括 cpu 的数量和类别，内存基地址，总线与桥，外设连接\中断控制器和、gpio 以及 clock 器等系统资源的树，Pinctrl 驱动支持从 device tree 中定义的设备节点获取 pin 的配置信息
Script 脚本	指的是打包到 img 中的 sys_config.fex 文件. 包含系统各模块配置参数

2.3. 模块配置介绍

在命令行中进入 Tina 根目录，执行命令进入配置主界面：

```
source build/envsetup.sh  (详见①)
lunch                     (详见②)
make kernel_menuconfig (详见③)
```

详注：

① 加载环境变量及 tina 提供的命令

② 输入编号，选择方案，

注意：R40 对应方案如：azalea_m2ultra-tina
R18 对应方案如：tulip-d1-tina
R16 对应方案如：astar_parrot-tina
R30 对应方案如：koto_perf1-tina
R31 对应方案如：mandolin_perf1-tina

③ 进入内核配置主界面(对一个 shell 而言，前两个命令只需要执行一次)

配置路径：

(R16 平台默认选上 pinctrl 相关配置,Pinctrl 子系统配置信息如下:)

```
Device Drivers
  └─>Pin controllers
      └─>Debug PINCTRL calls
```

(R18 平台)

```
Device Drivers
  └─>Pin controllers
      └─>Allwinner SOC PINCTRL DRIVER
          └─>Pinctrl sun50iw1p1 PIO controller
```

(R30 平台)

```
Device Drivers
  └─>Pin controllers
      └─>Allwinner SOC PINCTRL DRIVER
          └─>Pinctrl sun50iw3p1 PIO controller
```

(R40 平台)

```
Device Drivers
  └─>Pin controllers
      └─>Allwinner SOC PINCTRL DRIVER
          └─>Pinctrl sun8iw11p1 PIO controller
```

(R311 平台)

```
Device Drivers
  └─>Pin controllers
      └─>Allwinner SOC PINCTRL DRIVER
          └─>Pinctrl sun8iw15p1 PIO controller
```

操作图示(以 R40 平台为例):

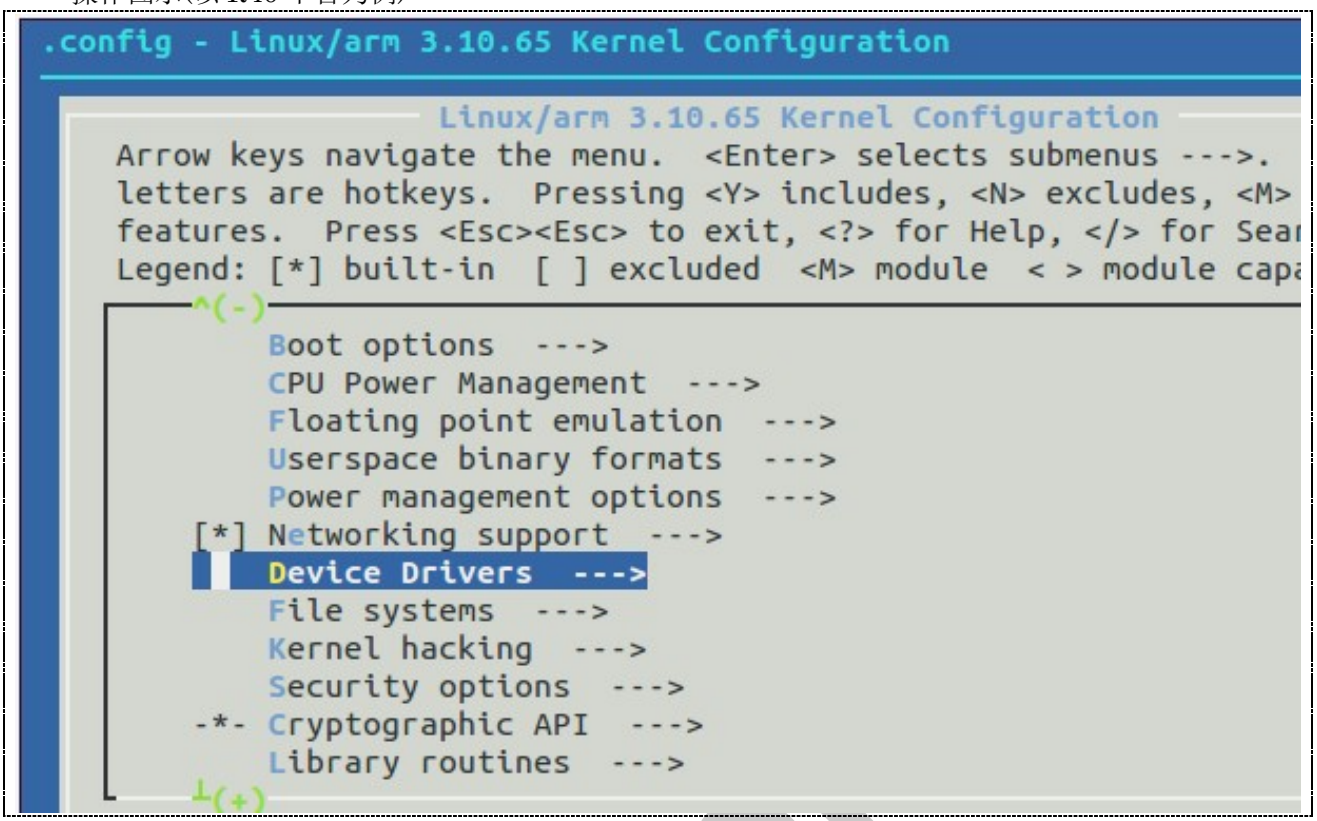

图 2-1 kernel_menuconfig 主界面

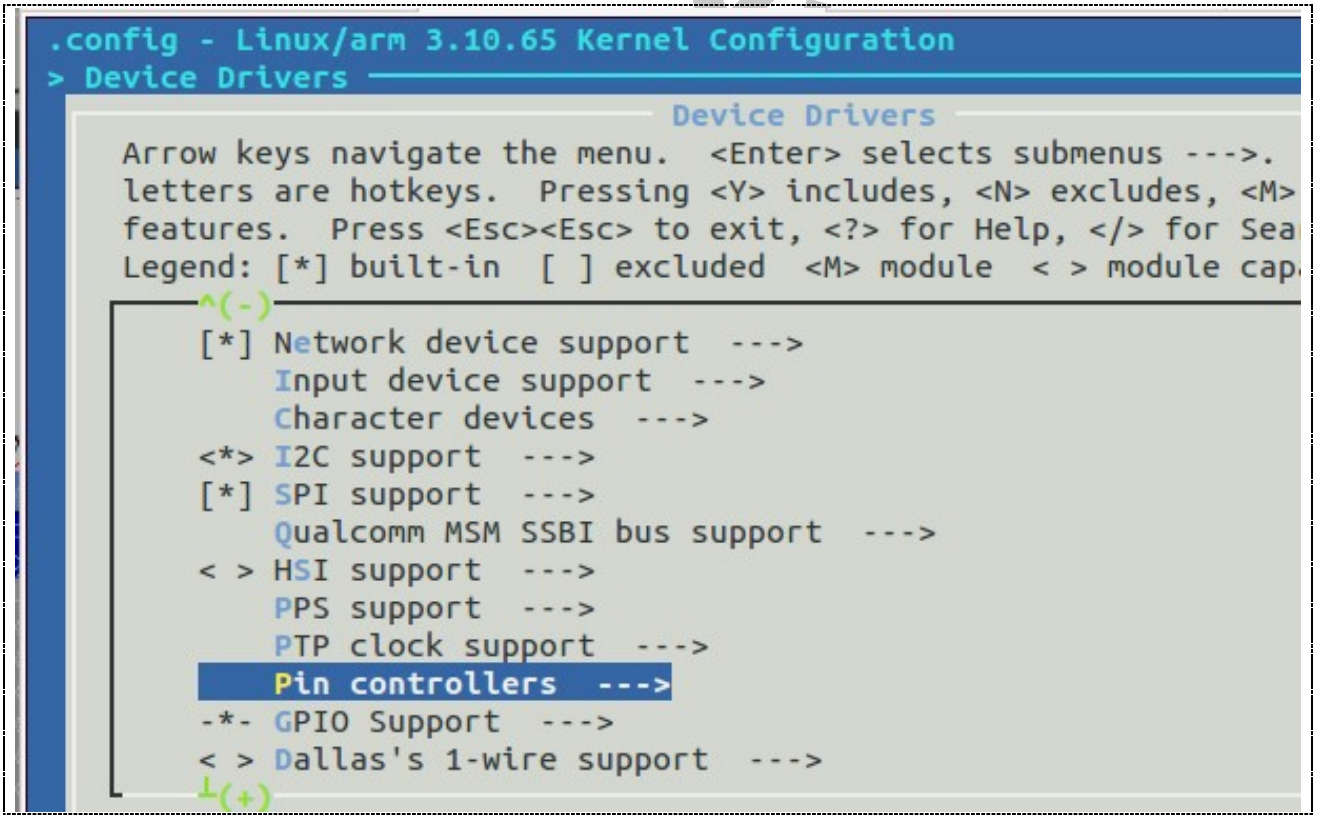

图 2-2 Device Driver 界面

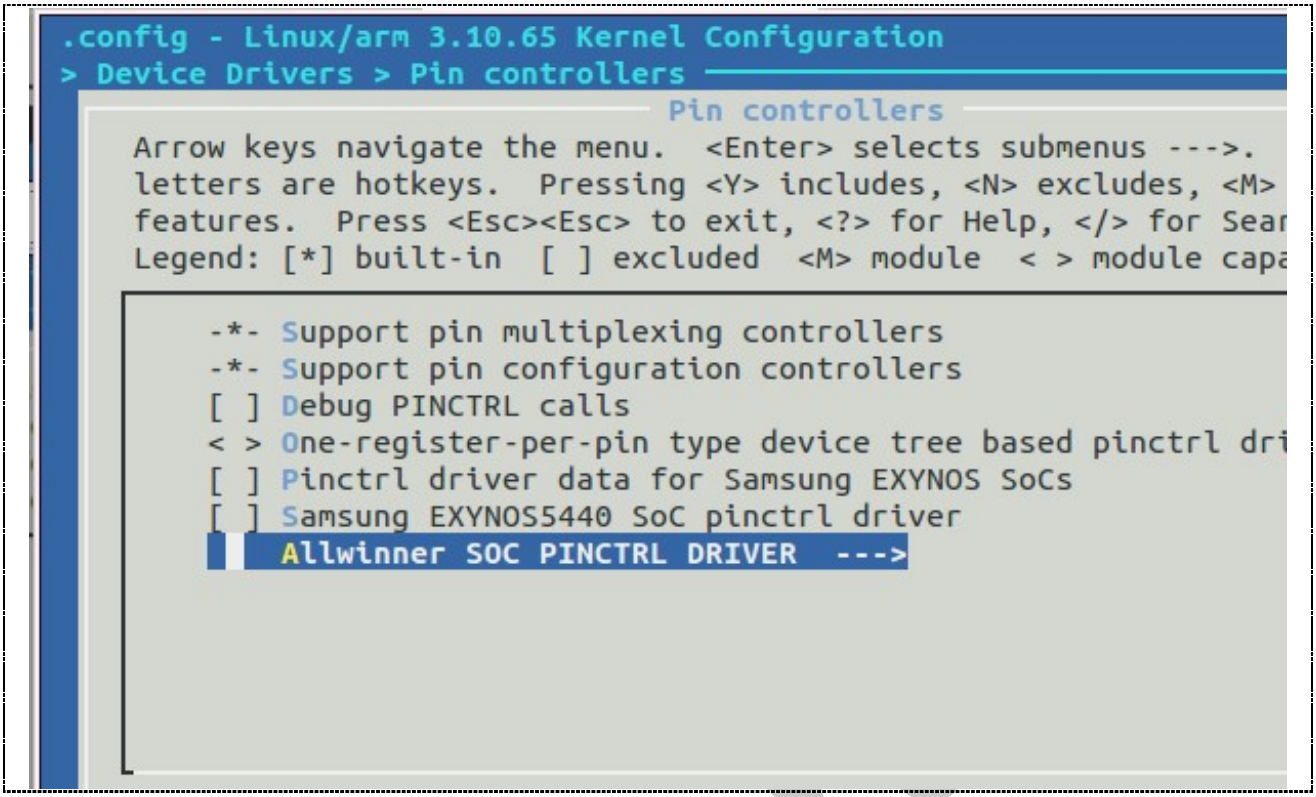

图 2-3 Pin controllers 界面

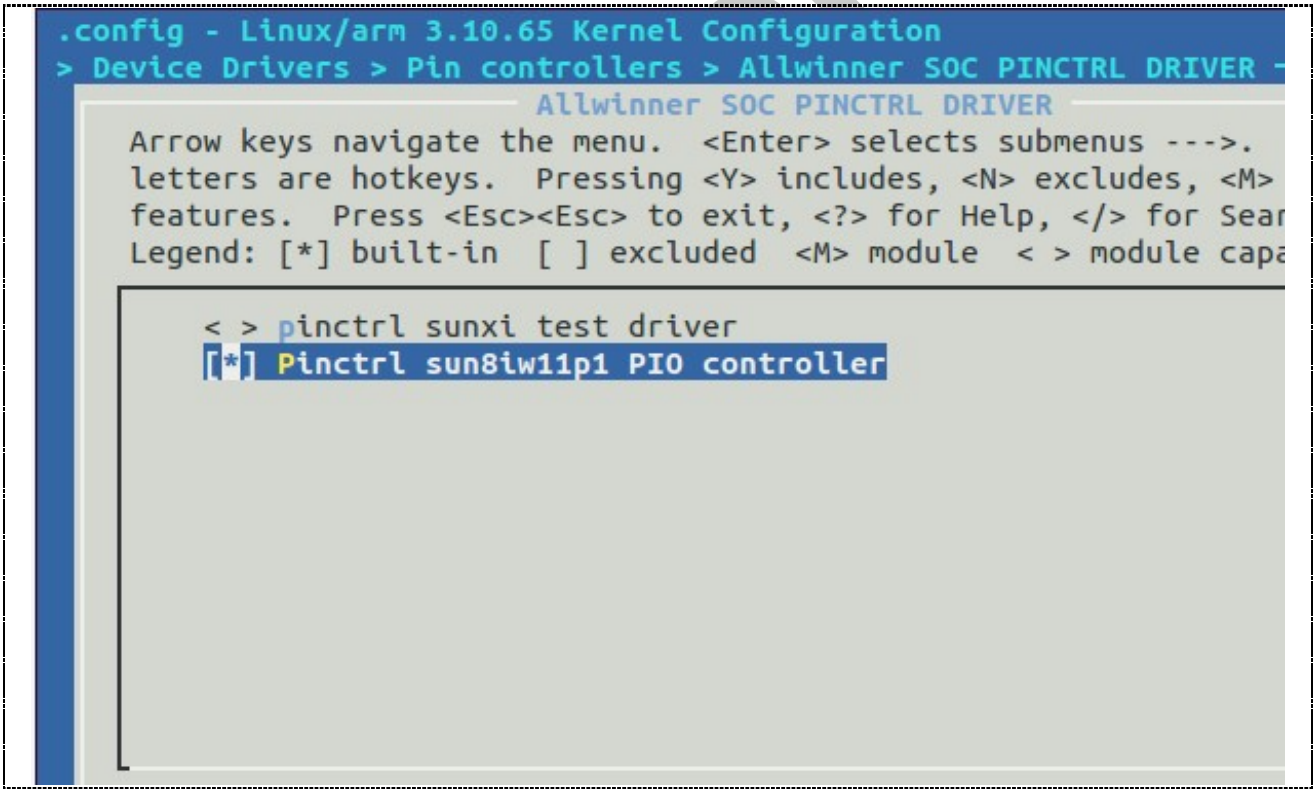

图 2-4 aw pinctrl 界面

2.4. 源码结构介绍

2.4.1. R18/R40 内核路径

tina/lichee/linux-3.4	(R16 平台)
tina/lichee/linux-3.10	(R40 平台)
tina/lichee/linux-4.4	(R18 平台)
tina/lichee/linux-4.9	(R30/R311 平台)

2.4.2. 内核头文件目录结构

(下面以 linux-3.10 为例,linux4.4/linux3.4 内核也是同样的路径)

```
linux-3.10/include/linux/pinctrl/
├── consumer.h
├── devinfo.h
├── machine.h
├── pinconf-generic.h
├── pinconf.h
├── pinconf-sunxi.h
├── pinctrl.h
├── pinctrl-state.h
└── pinmux.h
```

2.4.3. 内核驱动源码目录结构

```
linux-3.10/drivers/pinctrl/
├── axp
├── core.c
├── core.h
├── devicetree.c
├── devicetree.h
├── Kconfig
├── Makefile
├── modules.builtin
├── modules.order
├── mvebu
├── pinconf.c
├── pinconf-generic.c
├── pinconf.h
├── pinctrl-ab8500.c
├── pinctrl-ab8505.c
├── pinctrl-ab8540.c
├── pinctrl-ab9540.c
├── pinctrl-abx500.c
├── pinctrl-abx500.h
├── pinctrl-at91.c
├── pinctrl-bcm2835.c
├── pinctrl-coh901.c
├── pinctrl-coh901.h
├── pinctrl-exynos5440.c
├── pinctrl-exynos.c
├── pinctrl-exynos.h
├── pinctrl-falcon.c
├── pinctrl-imx23.c
├── pinctrl-imx28.c
├── pinctrl-imx35.c
├── pinctrl-imx51.c
├── pinctrl-imx53.c
├── pinctrl-imx6dl.c
├── pinctrl-imx6q.c
├── pinctrl-imx6sl.c
├── pinctrl-imx.c
├── pinctrl-imx.h
├── pinctrl-lantiq.c
├── pinctrl-lantiq.h
├── pinctrl-mxs.c
├── pinctrl-mxs.h
├── pinctrl-nomadik.c
├── pinctrl-nomadik-db8500.c
├── pinctrl-nomadik-db8540.c
├── pinctrl-nomadik.h
├── pinctrl-nomadik-stn8815.c
├── pinctrl-s3c64xx.c
├── pinctrl-samsung.c
├── pinctrl-samsung.h
├── pinctrl-single.c
├── pinctrl-sirf.c
```

```
├── pinctrl-tegra114.c
├── pinctrl-tegra20.c
├── pinctrl-tegra30.c
├── pinctrl-tegra.c
├── pinctrl-tegra.h
├── pinctrl-u300.c
├── pinctrl-utils.c
├── pinctrl-utils.h
├── pinctrl-xway.c
├── pinmux.c
├── pinmux.h
├── sh-pfc
├── spear
├── sunxi
│   ├── Kconfig
│   ├── Makefile
│   ├── modules.builtin
│   ├── modules.order
│   ├── pinctrl-sun8iw11p1.c
│   ├── pinctrl-sunxi.c
│   ├── pinctrl-sunxi.h
│   └── pinctrl-sunxi-test.c
└── vt8500
```

3. 驱动框架

3.1. 总体框架

sunxi Pinctrl 驱动模块可以分成 4 个部分：pinctrl api (即 pintrl interfaces,下同)、pinctrl framework、sunxi pinctrl driver 和 board configuration，框架如 图 3-1 驱动框架 所示。

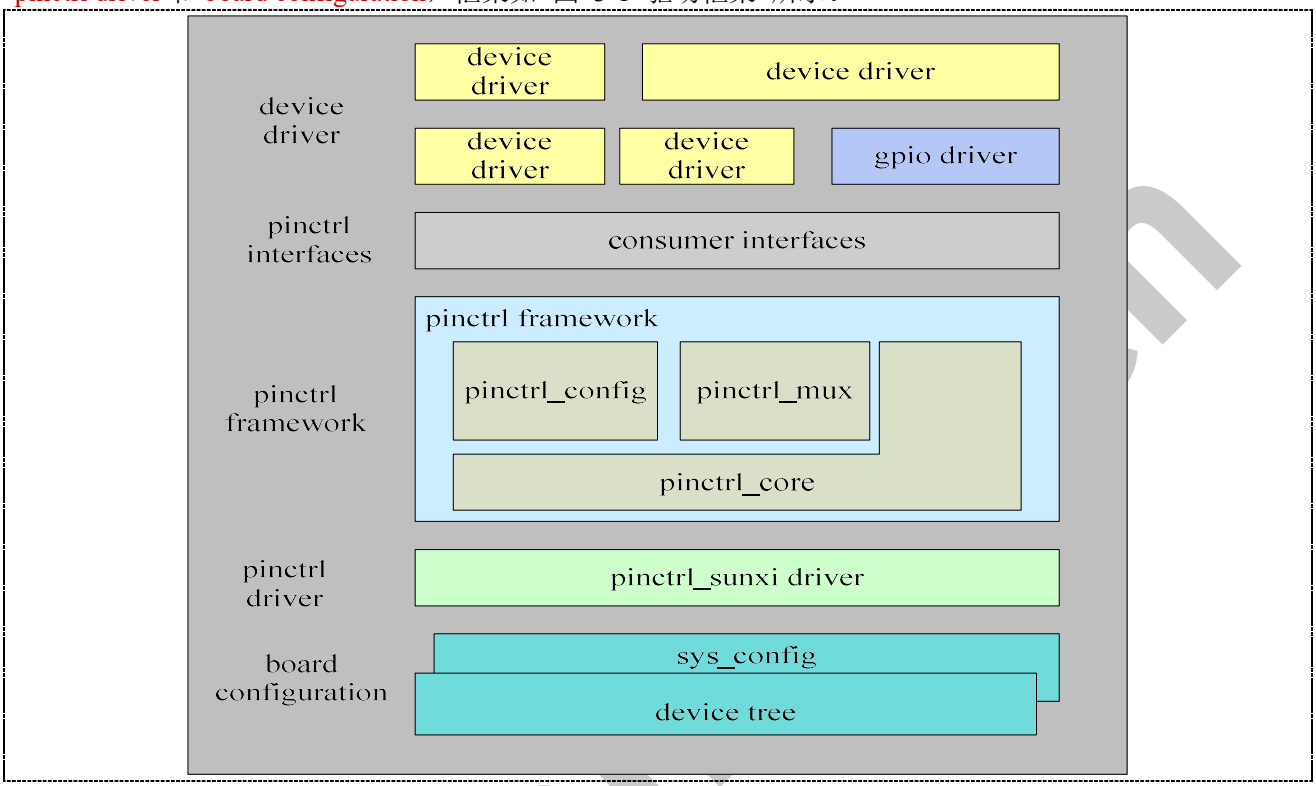

图 3-1 驱动框架

其中：

pinctrl api	pinctrl 提供给上层用户调用的接口
pinctrl framework	Linux 提供的 pinctrl 驱动框架
pinctrl sunxi driver	sunxi 平台需要实现的驱动
board configuration	设备 pin 配置信息，格式 device tree source 或者 sys_config

3.2. state/pinmux/pinconfig

Pinctrl framework 主要处理 pinstate、pinmux 和 pinconfig 三个功能，pinstate 和 pinmux、pinconfig 映射关系如 图 3-2 pinstate 和 pinmux、pinconfig 映射关系 所示。

系统运行在不同的状态，pin 配置有可能不一样，比如系统正常运行时，设备的 pin 需要一组配置，但系统进入休眠时，为了节省功耗，设备 pin 需要另一组配置。Pinctrl framwork 能够有效管理设备在不同状态下的引脚配置。

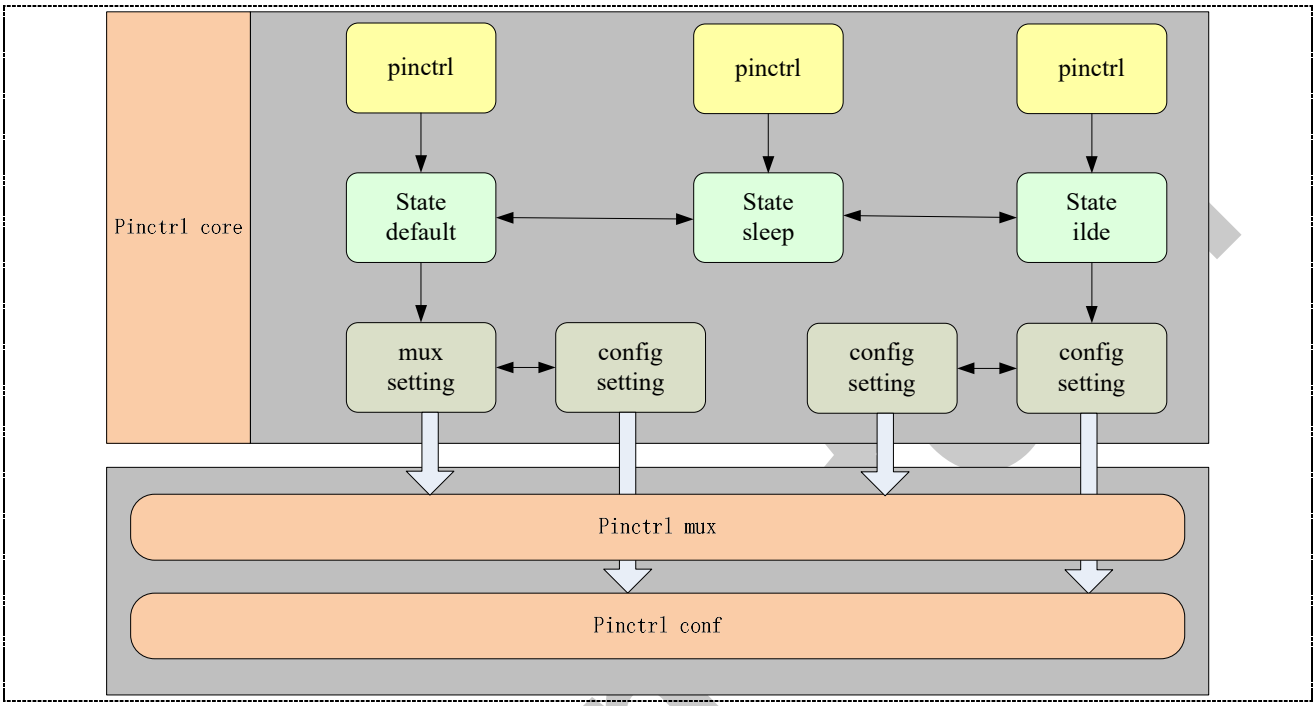

图 3-2 pinstate 和 pinmux、pinconfig 映射关系

4. 外部接口

(下面列举的是常用接口,旁边会注释那个平台有该接口)

4.1. pinctrl

4.1.1. pinctrl_get (R18/R40)

类别	介绍
函数原型	struct pinctrl *pinctrl_get(struct device *dev)
参数	dev：使用 pin 的设备,pinctrl 子系统通过设备名与 pin 配置信息匹配,获取 pin 配置信息
返回	pinctrl 句柄
功能描述	获取设备的 pin 操作句柄，所有 pin 操作必须基于此 pinctrl 句柄

4.1.2. pinctrl_put (R18/R40)

类别	介绍
函数原型	void pinctrl_put(struct pinctrl *p)
参数	p：pinctrl 句柄
返回	无
功能描述	释放 pinctrl 句柄，必须与 pinctrl_get 配对使用

4.1.3. devm_pinctrl_get (R16/R18/R40)

类别	介绍
函数原型	struct pinctrl *devm_pinctrl_get(struct device *dev)
参数	dev：使用 pin 的设备，pinctrl 子系统通过设备名与 pin 配置信息匹配
返回	pinctrl 句柄
功能描述	根据设备获取 pin 操作句柄，所有 pin 操作必须基于此 pinctrl 句柄，与 pinctrl_get 功能完全一样，只是 devm_pinctrl_get 会将申请到的 pinctrl 句柄做记录，绑定到设备句柄信息中。设备驱动申请 pin 资源，推荐优先使用 devm_pinctrl_get 接口

4.1.4. devm_pinctrl_put (R16/R18/R40)

类别	介绍
函数原型	void devm_pinctrl_put(struct pinctrl *p)
参数	p：pinctrl 句柄
返回	无
功能描述	释放 pinctrl 句柄，必须与 devm_pinctrl_get 配对使用

4.1.5. pinctrl_lookup_state (R16/R18/R40)

类别	介绍
函数原型	struct pinctrl_state *pinctrl_lookup_state(struct pinctrl *p, const char *name)
参数	p：pin 句柄 name：state name(R16 平台上只有 default 一种状态)
返回	state 状态句柄
功能描述	根据 pin 操作句柄，查找 state 状态句柄

4.1.6. pinctrl_select_state (R16/R18/R40)

类别	介绍
函数原型	int pinctrl_select_state(struct pinctrl *p, struct pinctrl_state *s)
参数	p：pin 句柄 s：state 句柄
返回	成功：0 失败：其它
功能描述	将 pin 句柄对应的 pinctrl 设置为 state 句柄对应的状态

4.1.7. devm_pinctrl_get_select (R16/R18/R40)

类别	介绍
函数原型	struct pinctrl *devm_pinctrl_get_select(struct device *dev, const char *name)
参数	dev：使用 pin 的设备，pinctrl 子系统通过设备名与 pin 配置信息匹配 name：state name(R16 平台上只有 default 一种状态)
返回	pinctrl 句柄
功能描述	获取设备的 pin 操作句柄，并将句柄设定为指定状态

4.1.8. devm_pinctrl_get_select_default (R16/R18/R40)

类别	介绍
函数原型	struct pinctrl *devm_pinctrl_get_select_default(struct device *dev)
参数	dev：使用 pin 的设备，pinctrl 子系统会通过设备名与 pin 配置信息匹配
返回	pinctrl 句柄
功能描述	获取设备的 pin 操作句柄，并将 pin 句柄对应的 pinctrl 设置为 default 状态

4.1.9. pin_config_get (R16/R18/R40)

类别	介绍
函数原型	int pin_config_get(const char *dev_name, const char *name, unsigned long *config)
参数	dev_name：pinctrl 名称 name：pin 名称 config：pin 配置属性
返回	成功：0 失败：其它
功能描述	获取指定 pin 的属性

4.1.10. pin_config_set (R16/R18/R40)

类别	介绍
函数原型	int pin_config_set(const char *dev_name, const char *name, unsigned long config)
参数	dev_name：pinctrl 名称 name：pin 名称 config：pin 配置属性
返回	成功：0 失败：其它
功能描述	设置指定 pin 的属性

4.1.11. pin_config_group_get (R16/R18/R40)

类别	介绍
函数原型	int pin_config_group_get(const char *dev_name, const char *pin_group, unsigned long *config)
参数	dev_name：pinctrl 名称

	pin_group：group 名称 config：pin 配置属性
返回	成功：0 失败：其它
功能描述	获取指定 group 的属性

4.1.12. pin_config_group_set (R16/R18/R40)

类别	介绍
函数原型	int pin_config_group_set(const char *dev_name, const char *pin_group, unsigned long config)
参数	dev_name：pinctrl 名称 pin_group：group 名称 config：pin 配置属性
返回	成功：0 失败：其它
功能描述	设置指定 group 的属性

4.2. gpio

4.2.1. gpio_request (R18/R40)

类别	介绍
函数原型	int gpio_request(unsigned gpio, const char *label)
参数	gpio：gpio 编号. label：gpio 名称，可以为 NULL
返回	成功：0 失败：其它
功能描述	申请 gpio. 获取 gpio 的访问权

4.2.2. gpio_free (R18/R40)

类别	介绍
函数原型	void gpio_free(unsigned gpio)
参数	gpio：gpio 编号
返回	无
功能描述	释放 gpio

4.2.3. gpio_direction_input (R18/R40)

类别	介绍
函数原型	int gpio_direction_input(unsigned gpio)
参数	gpio：gpio 编号
返回	成功：0 失败：其它
功能描述	将 gpio 设置为 input

4.2.4. gpio_direction_output (R18/R40)

类别	介绍
函数原型	int gpio_direction_output(unsigned gpio, int value)
参数	gpio：gpio 编号. value：gpio 电平值，非 0 表示高，0 表示低
返回	成功：0 失败：其它
功能描述	将 gpio 设置为 output，并设置电平值

4.2.5. __gpio_get_value (R18/R40)

类别	介绍
函数原型	int __gpio_get_value(unsigned gpio)
参数	gpio：gpio 编号
返回	gpio 电平， 1：表示 gpio 电平为高 0：表示 gpio 电平为低
功能描述	获取 gpio 电平值. (gpio 已为 input/output 状态)

4.2.6. __gpio_set_value (R18/R40)

类别	介绍
函数原型	void __gpio_set_value(unsigned gpio, int value)
参数	gpio：gpio 编号. value：gpio 电平值，非 0 表示高，0 表示低
返回	无
功能描述	设置 gpio 电平值. (gpio 已为 output 状态)

4.2.7. of_get_named_gpio (R18/R40)

类别	介绍
函数原型	int of_get_named_gpio(struct device_node *np, const char *propname, int index)
参数	np：要获取 GPIO 信息的节点 propname：节点中包含 gpio 描述信息的属性. index：所要查找的 gpio 在名称为 propname 的属性中的索引号
返回	GPIO 号
功能描述	通过名称获取 GPIO 索引号

5. R18/R40 使用例子

5.1. 配置

(以 R40 的 3.4 内核为例)

总结 linux-3.4 平台上 sys_config.fex 的配置，用户在主键中的管脚配置主要有以下几种情形，针对这几种情形，文档描述了在 devicetree 配置文件中，用户如何实现对应配置.

配置场景：

1. 用户只配置通用 GPIO，即用来做输入、输出、中断
2. 用户只配置设备管脚，如 Uart 设备的引脚、LCD 的引脚等.
3. 用户既要配置通用 GPIO，也要配置设备引脚.

5.1.1. 场景一

场景一：用户只需要配置 GPIO，devicetree 配置 demo 如下所示：

5.1.1.1. sys_config.fex 配置:

```
[Vdevice]
Vdevice_1                = port:PB02<0><1><0><0>
Vdevice_2                = port:PB03<1><1><0><0>
```

5.1.1.2. device_tree 对应配置

```
soc{
vdevie: vdevice@0{
            ...
             Vdevice_1=<&pio PB 2 0 1 0 0>
            Vdevice_2=<&pio PB 3 1 1 0 0>
      .....
}
};
```

说明：

gpio in/gpio out/ eint 采用上边的配置方法，配置参数解释如下：

```
vdevice_1=    <&pio    PB   1    1    1    1    0>;
      |              |     |    |    |    |    |-----------------电平
      |              |     |    |    |    |----------------上下拉
      |              |     |    |    |---------------------驱动力
      |              |     |    |--------------------------复用类型，0-GPIOIN 1-GPIOOUT..
      |              |     |-------------------------pin bank 内偏移.
      |              |-----------------------------哪个 bank
      |-------------------------------------指向哪个 pio，属于 cpus 要用&r_pio
      |-----------------------------------------------属性名字，相当 sys_config 子键名
```

5.1.2. 场景二

场景二：用户只需要配置设备引脚，devicetree 配置 demo 如下所示：

5.1.2.1. sys_config.fex 配置

```
[Vdevice]
Vdevice_0                = port:PB00<2><1><0><0>
Vdevice_1                = port:PB01<2><1><0><0>
Vdevice_2                = port:PB02<2><1><0><0>
Vdevice_3                = port:PB03<2><1><0><0>
```

5.1.2.2. device_tree 对应配置

```
soc{
pio: pinctrl@01c20800 {
        [...]
        vdevice_pins_a: vdevice@0 {
                allwinner,pins = "PB0", "PB1", "PB2", "PB3";
                allwinner,function = "vdevice";
                    allwinner,mutsel = <2>;
                allwinner,drive = <1>;
                allwinner,pull = <0>;
                allwinner,data = <0>;
        };
};
[...]
vdevie: vdevie@0{
    compatible = "allwinner,sun50i-vdevice";
    pinctrl-names = "default";
    pinctrl-0 = <&vdevice_pins_a>;
    status = "okay";
}
};
```

5.1.3. 场景三

场景三：用户既要配置通用 GPIO，也要配置设备引脚，devicetree 配置 demo 如下

5.1.3.1. sys_config.fex 配置

```
[Vdevice]
Vdevice_0               = port:PB00<2><1><0><0>
Vdevice_1               = port:PB01<2><1><0><0>
Vdevice_2               = port:PB02<2><1><0><0>
Vdevice_3               = port:PB03<2><1><0><0>
Vdevice_4               = port:PB04<1><1><0><0>
```

5.1.3.2. device_tree 对应配置

```
soc{
pio: pinctrl@01c20800 {
        [...]
        vdevice_pins_a: vdevice@0 {
                allwinner,pins = "PB0", "PB1", "PB2", "PB3";
                allwinner,function = "vdevice";
                  Allwinner,mutsel = <2>;
                allwinner,drive = <1>;
                allwinner,pull = <0>;
                allwinner,data = <0>;
        };
};
[...]
vdevie: vdevie@0{
    ...
    pinctrl-names = "default";
    pinctrl-0 = <&vdevice_pins_a>;
        Vdevice-4 = <&pio PB 4 1 0 0>
    ...
}
};
```

5.2. 接口使用示例

5.2.1. 配置设备引脚

一般设备驱动只需要使用一个接口 devm_pinctrl_get_select_default 就可以申请到设备所有 pin 资源。

```
static int sunxi_pin_req_demo(struct platform_device *pdev)
{
```

```
        struct pinctrl *pinctrl;
        pr_warn("device [%s] probe enter\n", dev_name(&pdev->dev));
        /* request device pinctrl, set as default state */
        pinctrl = devm_pinctrl_get_select_default(&pdev->dev);
        if (IS_ERR_OR_NULL(pinctrl)) {
                pr_warn("request pinctrl handle for device [%s] failed\n",
                dev_name(&pdev->dev));
                return -EINVAL;
        }
        pr_debug("device [%s] probe ok\n", dev_name(&pdev->dev));
        return 0;
}
```

5.2.2. 获取 GPIO 号

```
static int sunxi_pin_req_demo(struct platform_device *pdev)
{
        int ret;
unsigned int gpio;
unsigned long out_init;
enum of_gpio_flags gpio_flags;
struct device_node *np = dev->of_node;
        struct device *dev = &pdev->dev;

        #get gpio config in device node.
gpio = of_get_named_gpio(np, "vdevice_3", 0);
if (!gpio_is_valid(gpio)) {
        if (gpio != -EPROBE_DEFER)
                dev_err(dev, "Error getting vdevice_3\n");
        return gpio;
}
}
```

5.2.3. GPIO 属性配置

通过 pin_config_set/pin_config_get/pin_config_group_set/pin_config_group_get 接口单独控制指定 pin 或 group 的相关属性。

```
static int sunxi_pin_resource_req(struct platform_device *pdev)
{
unsigned int gpio;
struct gpio_config pin_cfg;
struct device_node *node;
char pin_name[32];
unsigned long config;

pr_warn("device [%s] pin resource request enter\n", dev_name(&pdev->dev));
node = of_find_node_by_name(NULL,"vdevice");
if(!node){
        return -EINVAL;
}
gpio = of_get_named_gpio_flags(node, "vdevice_0", 0, (enum of_gpio_flags *)&pin_cfg);
if (!gpio_is_valid(gpio)) {
        return -EINVAL;
}
if (!IS_AXP_PIN(pin_cfg.gpio)) {

        /* valid pin of sunxi-pinctrl, config pin attributes individually.*/
        sunxi_gpio_to_name(pin_cfg.gpio, pin_name);
        config= SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_FUNC, pin_cfg.mul_sel);
        pin_config_set(SUNXI_PINCTRL, pin_name, config);

        if (pin_cfg.pull != GPIO_PULL_DEFAULT) {
                config = SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_PUD, pin_cfg.pull);
                pin_config_set(SUNXI_PINCTRL, pin_name, config);
        }

        if (pin_cfg.drv_level != GPIO_DRVLVL_DEFAULT) {
            config=SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_DRV,pin_cfg.drv_level);
            pin_config_set(SUNXI_PINCTRL, pin_name, config);
        }

        if (pin_cfg.data != GPIO_DATA_DEFAULT) {
                config = SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_DAT, pin_cfg.data);
                pin_config_set(SUNXI_PINCTRL, pin_name, config);
        }
} else{

        /* valid pin of axp-pinctrl, config pin attributes individually. */
        axp_gpio_to_name(pin_cfg.gpio, pin_name);
        config= SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_FUNC, pin_cfg.mul_sel);
        pin_config_set(AXP_PINCTRL, pin_name, config);

        if (pin_cfg.data != GPIO_DATA_DEFAULT) {
                config = SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_DAT, pin_cfg.data);
                pin_config_set(AXP_PINCTRL, pin_name, config);
        }
}
pr_debug("device [%s] pin resource request ok\n", dev_name(&pdev->dev));
return 0;

}
```

5.3. 设备驱动如何使用 pin 中断

目前 sunxi-pinctrl 使用 irq-domain 为 gpio 中断实现虚拟 irq 的功能，使用 gpio 中断功能时，设备驱动只需要通过 gpio_to_irq 获取虚拟中断号后，其他均可以按标准 irq 接口操作。

```
static int sunxi_gpio_eint_demo(struct platform_device *pdev)
{
	struct device *dev = &pdev->dev;
	int virq;
	int ret;
	/* map the virq of gpio */
	virq = gpio_to_irq(GPIOA(0));
	if (IS_ERR_VALUE(virq)) {
		pr_warn("map gpio [%d] to virq failed, errno = %d\n",
		GPIOA(0), virq);
		return -EINVAL;
	}
	pr_debug("gpio [%d] map to virq [%d] ok\n", GPIOA(0), virq);
	/* request virq, set virq type to high level trigger */
	ret = devm_request_irq(dev, virq, sunxi_gpio_irq_test_handler,
	IRQF_TRIGGER_HIGH, "PA0_EINT", NULL);
	if (IS_ERR_VALUE(ret)) {
		pr_warn("request virq %d failed, errno = %d\n", virq, ret);
		return -EINVAL;
	}
	return 0;
}
```

6. R16 使用例子

6.1. 通过 sys_config 申请

设备需要的 pin 资源可以在 sys_confix 文件中配置,在 sunxi-pinctrl 初始化时会解析 sys_config 中的设备 pin 配置信息,并将设备的 pin 配置信息添加到 sunxi-pinctrl 模块中,设备驱动申请设备 pin 资源时只需告诉 pinctrl 设备名称即可,pinctrl 会自动将设备的 pin 资源信息绑定到该设备的 pinctrl 句柄中。

一般设备驱动只需要使用一个接口 devm_pinctrl_get_select_default 就可以申请到设备所有 pin 资源。

示例 1:

```
static int gmac_system_init(struct platform_device *pdev)
{
	int ret = 0;
	struct pinctrl *gmac_pin;
	gmac_pin = devm_pinctrl_get_select_default(&pdev->dev);
	if (IS_ERR_OR_NULL(gmac_pin))
		ret = -EINVAL;
	return ret;
}
```

示例 2:

```
static int twi_request_gpio(struct sunxi_i2c *i2c)
{
	int ret = 0;
	if (!twi_chan_is_enable(i2c->bus_num))
		return -1;
	i2c->pctrl = devm_pinctrl_get(i2c->adap.dev.parent);
	if (IS_ERR(i2c->pctrl)) {
		return -1;
	}
	i2c->pctrl_state = pinctrl_lookup_state(i2c->pctrl, PINCTRL_STATE_DEFAULT);
	if (IS_ERR(i2c->pctrl_state)) {
		return -1;
	}
	ret = pinctrl_select_state(i2c->pctrl, i2c->pctrl_state);
	return ret;
}
static void twi_release_gpio(struct sunxi_i2c *i2c)
{
	devm_pinctrl_put(i2c->pctrl);
}
```

6.2. 设备驱动直接申请

sunxi-pinctrl 可通过以下接口单独控制指定 pin 或 group 的相关属性。

pin_config_set
pin_config_get
pin_config_group_set
pin_config_group_get

目前 sunxi-pinctrl 平台支持配置 pin 以下四种属性。

Mux
Pull
Driver-strength
Output data

示例 1:

```
static int sunxi_pin_resource_req(struct gpio_config *pin_cfg)
{
	char pin_name[SUNXI_PIN_NAME_MAX_LEN];
	unsigned config;
```

```
        if (!IS_AXP_PIN(pin_cfg->gpio)) {
            sunxi_gpio_to_name(pin_cfg->gpio, pin_name);
            config = SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_FUNC, pin_cfg->mul_sel);
            pin_config_set(SUNXI_PINCTRL, pin_name, config);
            if (pin_cfg->pull != GPIO_PULL_DEFAULT) {
                config = SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_PUD, pin_cfg->pull);
                pin_config_set(SUNXI_PINCTRL, pin_name, config);
            }
            if (pin_cfg->drv_level != GPIO_DRVLVL_DEFAULT) {
                config = SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_DRV, pin_cfg->drv_level);
                pin_config_set(SUNXI_PINCTRL, pin_name, config);
            }
            if (pin_cfg->data != GPIO_DATA_DEFAULT) {
                config = SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_DAT, pin_cfg->data);
                pin_config_set(SUNXI_PINCTRL, pin_name, config);
            }
        } else {
            sunxi_gpio_to_name(pin_cfg->gpio, pin_name);
            config = SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_FUNC, pin_cfg->mul_sel);
            pin_config_set(AXP_PINCTRL, pin_name, config);
            if (pin_cfg->data != GPIO_DATA_DEFAULT) {
                config = SUNXI_PINCFG_PACK(SUNXI_PINCFG_TYPE_DAT, pin_cfg->data);
                pin_config_set(AXP_PINCTRL, pin_name, config);
            }
        }
    }
```

6.3. 设备驱动中的使用 pin 中断

目前 sunxi-pinctrl 使用 irq-domain 为 gpio 中断实现虚拟 irq 的功能,使用 gpio 中断功能时,设备驱动只需要通过 gpio_to_irq 获取虚拟中断号后,其他均可按照标准 irq 接口操作。

示例 1:

```
static int sunxi_gpio_eint_test(struct platform_device *pdev)
{
    struct device *dev = &pdev->dev;
    int    virq;
    int    ret;
    /* map the virq of gpio */
    virq = gpio_to_irq(GPIOA(0));
    if (IS_ERR_VALUE(virq)) {
        pr_warn("map gpio [%d] to virq failed, errno = %d\n",
            GPIOA(0), virq);
        return -EINVAL;
    }
    pr_warn("gpio [%d] map to virq [%d] ok\n", GPIOA(0), virq);

    /* request virq, set virq type to high level trigger */
    ret = devm_request_irq(dev, virq, sunxi_gpio_irq_test_handler,
        IRQF_TRIGGER_HIGH, "PA0_EINT", NULL);
    if (IS_ERR_VALUE(ret)) {
        pr_warn("request virq %d failed, errno = %d\n",virq, ret);
        return -EINVAL;
    }
    /* enbale virq */
    //enable_irq(virq);

    return 0;
}
```

6.4. 注意事项

pinctrl 模块兼容 GPIO 的功能,如果 pin 是作为 GPIO input/ouput,仍然可以使用 gpiolib 中的标准接口,但是如果要使用 GPIO 的复用功能,则需要使用 pinctrl 接口

6.5. FAQs

1.调用 devm_pinctrl_get_select_default 返回错误

错误分析:pinctrl 初始化的时候,会去读取 sys_config 中 pin 的配置信息,建立 pin 的复用映射表和属性映射表。对映射表中的设备名,驱动初始化时会去读取有没有存在子健 device_name,如果不存在,则以主键名为设备名。

解决方法:当采用主键名作为设备名时,检查 sys_config 中主键名是否与需要申请 pin 资源的设备名一致。例如:

主键名:[uart_para2],设备名:[uart2],(错误)

主键名:[uart2],设备名:[uart2],(正确)

7. Declaration