全 志 科 技
Allwinner Technology

A64

PINCTRL 模块使用说明文档

/V1.2

文档履历

版本号	日期	制/修订人	制/修订记录
V1.0	2013-07-05	Hsr	Init version
V1.1	2014-11-22	Hsr	Add support a64
V1.2	2015-05-15	Hsr	Update for a64

目　　录

1. 概述.....4
1.1. 编写目的.....4
1.2. 适用范围.....4
1.3. 相关人员.....4
2. 模块介绍.....5
2.1. 模块功能介绍.....5
2.2. 相关术语介绍.....5
2.3. 模块配置介绍.....6
2.4. 源码结构介绍.....8
3. 驱动框架.....9
3.1. 总体框架.....9
3.2. state/pinmux/pinconfig.....9
4. 外部接口.....11
4.1. pinctrl.....11
4.1.1. pinctrl_get.....11
4.1.2. pinctrl_put.....11
4.1.3. devm_pinctrl_get.....11
4.1.4. devm_pinctrl_put.....11
4.1.5. pinctrl_lookup_state.....12
4.1.6. pinctrl_select_state.....12
4.1.7. devm_pinctrl_get_select.....12
4.1.8. devm_pinctrl_get_select_default.....12
4.1.9. pin_config_get.....13
4.1.10. pin_config_set.....13
4.1.11. pin_config_group_get.....13
4.1.12. pin_config_group_set.....13
4.2. gpio.....14
4.2.1. gpio_request.....14
4.2.2. gpio_free.....14
4.2.3. gpio_direction_input.....14
4.2.4. gpio_direction_output.....14
4.2.5. __gpio_get_value.....15
4.2.6. __gpio_set_value.....15
4.2.7. of_get_named_gpio.....15
5. 使用例子.....16
5.1. 配置.....16
5.1.1. 场景一.....16
5.1.2. 场景二.....17
5.1.3. 场景三.....17
5.2. 接口使用示例.....18
5.2.1. 配置设备引脚.....18
5.2.2. 获取 GPIO 号.....19
5.2.3. GPIO 属性配置.....19
5.3. 设备驱动如何使用 pin 中断.....21

1. 概述

1.1. 编写目的

本文档对Linux3.10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linux3.10内核，1689平台。

1.3. 相关人员

本文档适用于所有需要在 Linux3.10+1689 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：

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

2.2. 相关术语介绍

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。

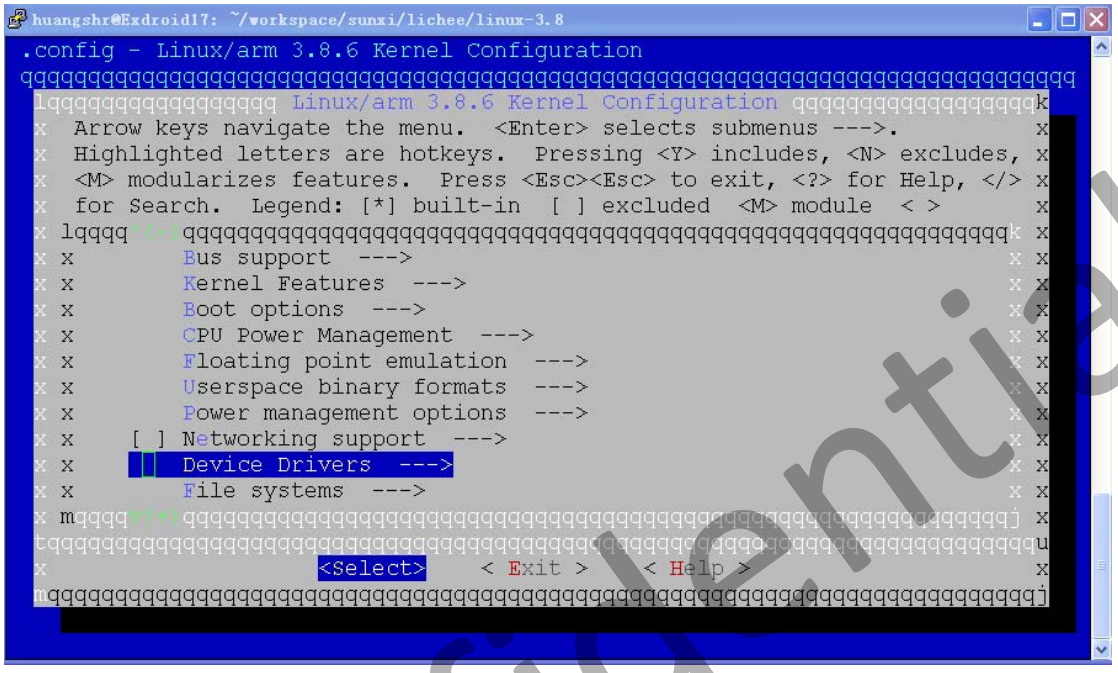
Device tree: 犹如它的名字，是一颗包括 cpu 的数量和类别，内存基地址，总线与桥，外设连接\中断控制器和、gpio 以及 clock 器等系统资源的树，Pinctrl 驱动支持从 device tree 中定义的设备节点获取 pin 的配置信息。

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。

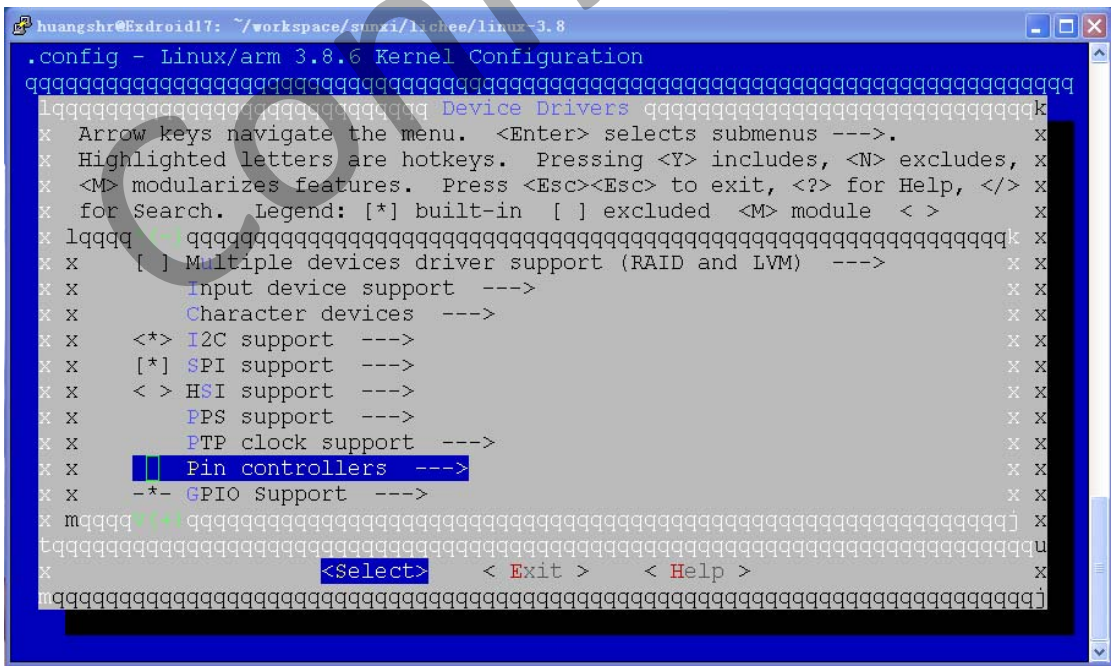
2.3. 模块配置介绍

在命令行中进入内核根目录，执行 make ARCH=arm64 menuconfig 进入配置主界面，并按以下步骤操作：

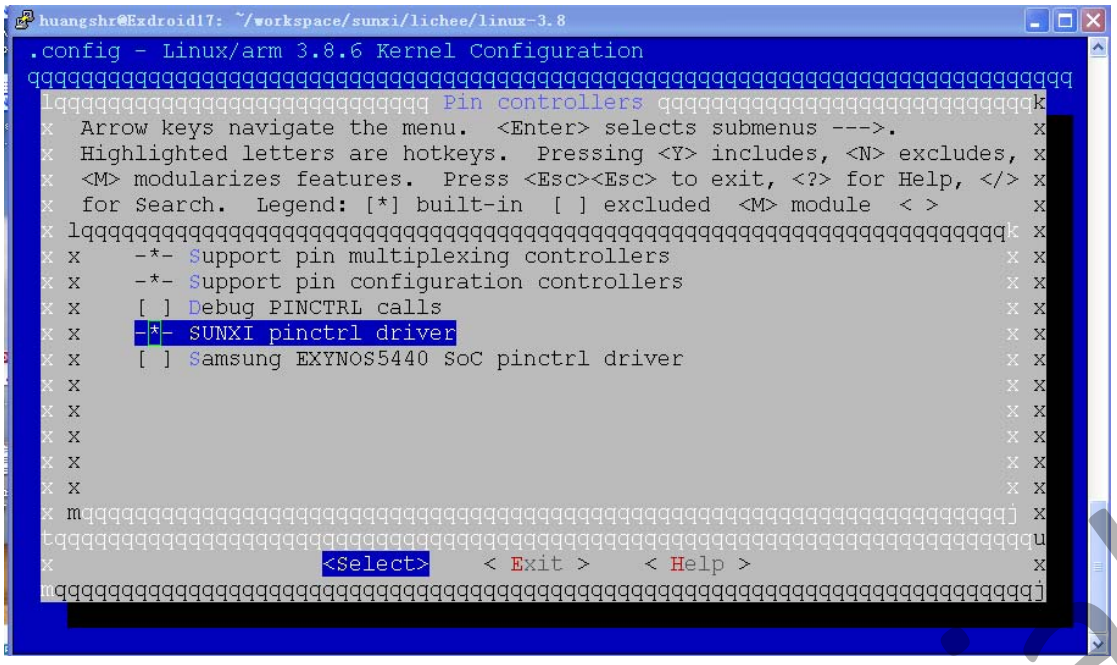
首先，选择 Device Drivers 选项进入下一级配置，如下图所示：



选择 Pin controllers,进入下级配置，如图所示：



Sunxi pinctrl driver 默认编译进内核，如图所示：



2.4. 源码结构介绍

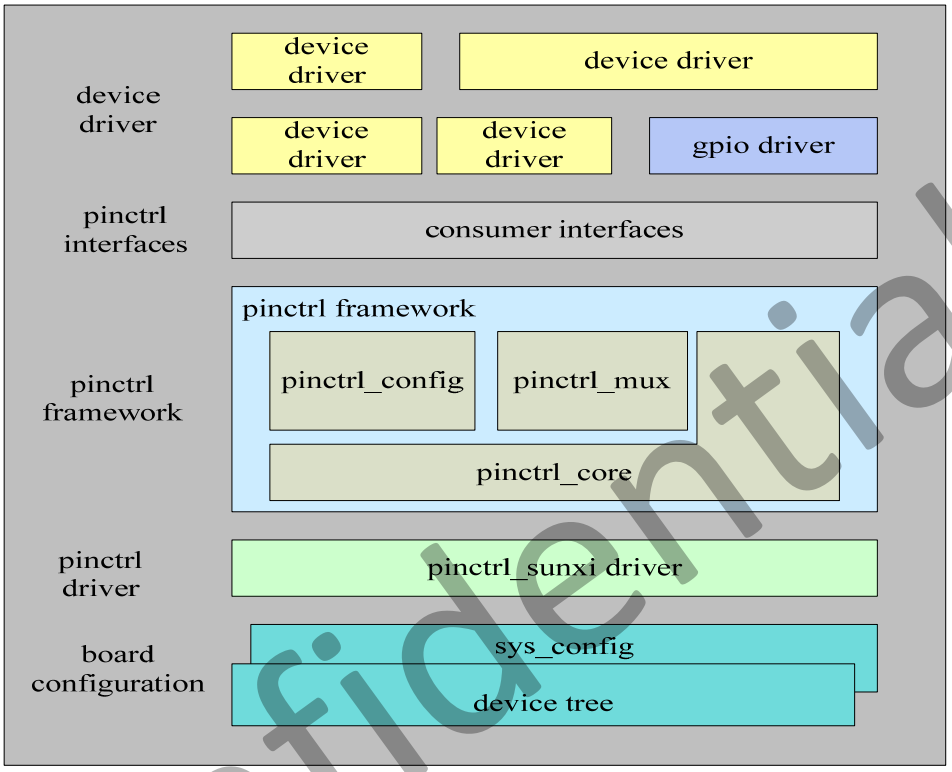
```
linux3.10

|-- drivers
|       |-- pinctrl
|       |           |-- Kconfig
|       |           |-- Makefile
|       |           |-- core.c
|       |           |-- core.h
|       |           |-- devicetree.c
|       |           |-- devicetree.h
|       |           |-- pinconf.c
|       |           |-- pinconf.h
|       |           |-- pinmux.c
|       |           `-- pinmux.h
|       `-- sunxi
|                   |-- pinctrl-sunxi-test.c
|                   |-- pinctrl-sun50iw1p1.c
|                   |-- pinctrl-sun50iw1p1-r.c
`-- include
        `-- linux
                `-- pinctrl
                            |-- consumer.h
                            |-- devinfo.h
                            |-- machine.h
                            |-- pinconf-generic.h
                            |-- pinconf.h
                            |-- pinctrl-state.h
                            |-- pinctrl.h
                            `-- pinmux.h
```

3. 驱动框架

3.1. 总体框架

Sunxi Pinctrl 驱动模块的框架如下图所示，整个驱动模块可以分成 4 个部分：pinctrl api、pinctrl common frame、sunxi pinctrl driver and board configuration。



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。

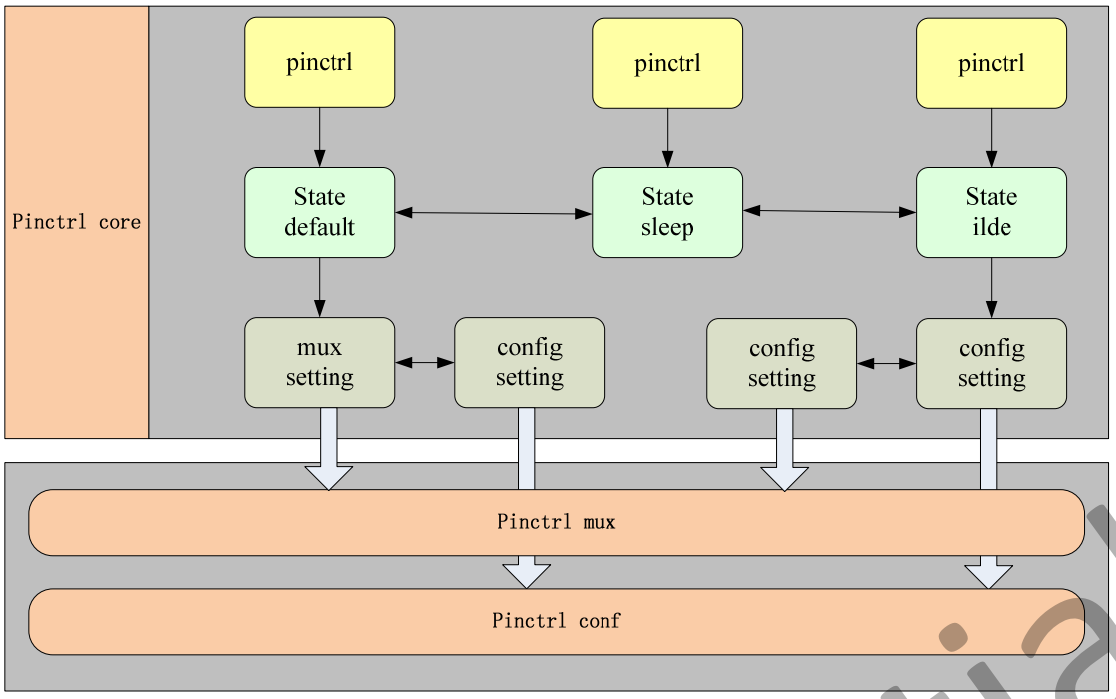
Pinctrl sunxi driver： sunxi 平台需要实现的驱动。

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 映射关系如下图所示。

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。



4. 外部接口

4.1. pinctrl

4.1.1. pinctrl_get

原型： struct pinctrl *pinctrl_get(struct device *dev);

功能： 获取设备的pin操作句柄，所有pin操作必须基于此pinctrl句柄；

输入： 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。

输出： pinctrl句柄。

4.1.2. pinctrl_put

原型： void pinctrl_put(struct pinctrl *p);

功能： 释放pinctrl句柄，必须与pinctrl_get配对使用。

输入： pinctrl句柄。

输出： 无。

4.1.3. devm_pinctrl_get

原型： struct pinctrl *devm_pinctrl_get(struct device *dev);

功能： 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。

输入： 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。

输出： pinctrl句柄。

4.1.4. devm_pinctrl_put

原型： void devm_pinctrl_put(struct pinctrl *p);

功能： 释放pinctrl句柄，必须与devm_pinctrl_get配对使用。

输入： pinctrl句柄。

输出： 无。

4.1.5. pinctrl_lookup_state

原型： struct pinctrl_state *pinctrl_lookup_state(struct pinctrl *p, const char *name)

功能： 根据pin操作句柄，查找state状态句柄；

输入： pin句柄

state name

输出： state状态句柄。

4.1.6. pinctrl_select_state

原型：int pinctrl_select_state(struct pinctrl *p, struct pinctrl_state *s)

功能： 将 pin 句柄对应的 pinctrl 设置为 state 句柄对应的状态；

输入： pin句柄

state句柄

输出： state设置结果，0-成功,其他-失败。

4.1.7. devm_pinctrl_get_select

原型： struct pinctrl *devm_pinctrl_get_select(struct device *dev, const char *name)

功能： 获取设备的pin操作句柄，并将句柄设定为指定状态；

输入： 使用pin的设备，pinctrl子系统通过设备名与pin配置信息匹配，

state name

输出： pinctrl句柄。

4.1.8. devm_pinctrl_get_select_default

原型： struct pinctrl *devm_pinctrl_get_select_default(struct device *dev)

功能： 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；

输入： 使用pin的设备，pinctrl子系统会通过设备名与pin配置信息匹配，

输出： pinctrl句柄。

4.1.9. pin_config_get

原型： int pin_config_get(const char *dev_name, const char *name, unsigned long *config)

功能： 获取指定pin的属性；

输入： pinctrl名称

pin名称

pin配置属性

输出： 获取pin属性结果，0-成功，其他-失败。

4.1.10. pin_config_set

原型： int pin_config_set(const char *dev_name, const char *name, unsigned long config)

功能： 设置指定pin的属性；

输入： pinctrl名称

Pin名称

pin配置属性

输出： 设置pin属性结果，0-成功，其他-失败。

4.1.11. pin_config_group_get

原型： int pin_config_group_get(const char *dev_name, const char *pin_group, unsigned long *config)

功能： 获取指定group的属性；

输入： pinctrl名称

group名称

pin配置属性

输出： 获取 group 属性结果，0-成功，其他-失败。

4.1.12. pin_config_group_set

原型： int pin_config_group_set(const char *dev_name, const char *pin_group,

unsigned long config)

功能： 设置指定group的属性；

输入： pinctrl名称

group名称

pin配置属性

输出： 设置 group 属性结果，0-成功，其他-失败。

4.2. gpio

4.2.1. gpio_request

原型： int gpio_request(unsigned gpio, const char *label)

功能: 申请 gpio. 获取 gpio 的访问权.

参数: gpio: gpio 编号.

label: gpio 名称, 可以为 NULL.

输出: 0 表示成功, 否则表示失败.

4.2.2. gpio_free

原型： void gpio_free(unsigned gpio)

功能: 释放 gpio.

参数: gpio: gpio 编号.

输出: 无.

4.2.3. gpio_direction_input

原型： int gpio_direction_input(unsigned gpio)
功能： 将 gpio 设置为 input.
参数： gpio: gpio 编号.
输出： 0 表示成功, 否则表示失败.

4.2.4. gpio_direction_output

原型： int gpio_direction_output(unsigned gpio, int value)
功能： 将 gpio 设置为 output, 并设置电平值.
参数： gpio: gpio 编号.
value: gpio 电平值, 非 0 表示高, 0 表示低.

输出： 0 表示成功，否则表示失败.

4.2.5. __gpio_get_value

原型：int __gpio_get_value(unsigned gpio)
功能： 获取 gpio 电平值. (gpio 已为 input/output 状态)
参数：gpio: gpio 编号.
输出：gpio 电平, 1 表示高, 0 表示低.

4.2.6. __gpio_set_value

原型： void __gpio_set_value(unsigned gpio, int value)
功能： 设置 gpio 电平值. (gpio 已为 output 状态)
参数： gpio: gpio 编号.
value: gpio 电平值，非 0 表示高, 0 表示低.
输出：无.

4.2.7. of_get_named_gpio

原型：int of_get_named_gpio(struct device_node *np, const char *propname, int index)
功能：通过名称获取 GPIO 索引号
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.
输出：GPIO 号.

5. 使用例子

5.1. 配置

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.

（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：

sys_config.fex 配置:

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

device_tree 对应配置

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

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：

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>
```

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：

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>
```

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;
}
```