ActDuino S500_pinctrl 使用说明书

最新版本号：1.0

2015-12-16

1 目 录

2 引 言

2.1 编写目的

本文为 pinctrl 说明书，便于内核开发人员熟悉 pinctrl 的使用方法，规范化 pinctrl 的使用。

2.2 术语和缩写词

缩写和术语	解 释
S500	炬芯基于 ARM A9 4 核 IC

3 原理与关系

3.1 linux pinctrl 子系统提供的功能

1. 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。
2. 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。pin control subsystem 要管理所有的 pin group。
3. 配置这些 pin 的特性。例如使能或关闭引脚上的 pull-up、pull-down 电阻，配置引脚的 drive strength。

3.2 pinctrl 相关概念

普通 driver 调用 pin control subsystem 的主要目标有两个：
（1）设定该设备的功能复用。
（2）设定该 device 对应的那些 pin 的电气特性。

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 function 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 { C6, C7, C8, C9 }，也可能引出到另外一个 pin group{ C22, C23, C24, C25 }，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。

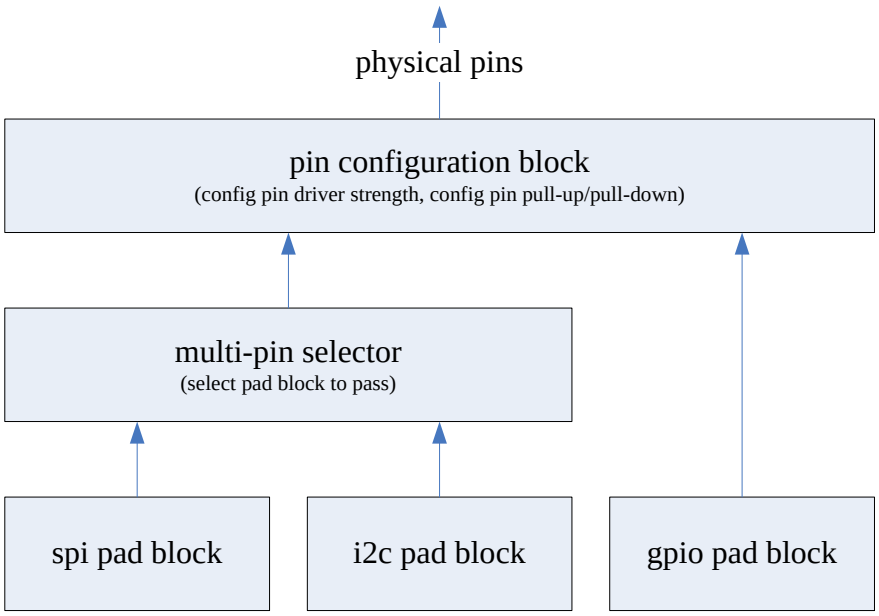
此外，由于电源管理的要求，某个 device 可能处于某个电源管理状态，例如 idle 或者 sleep，这时候，属于该 device 的所有的 pin 就会需要处于另外的状态。

综合上述的需求，就定义了 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，就有了 pin control state holder 的概念，用来管理一个设备的所有的 pin control 状态。

综上所述，普通 driver 调用 pin control subsystem 的接口就只是三个步骤：

（1）驱动加载或是运行时，获取 pin control state holder 的句柄
（2）设定 pin control 状态
（3）驱动卸载或是退出时，释放 pin control state holder 的句柄

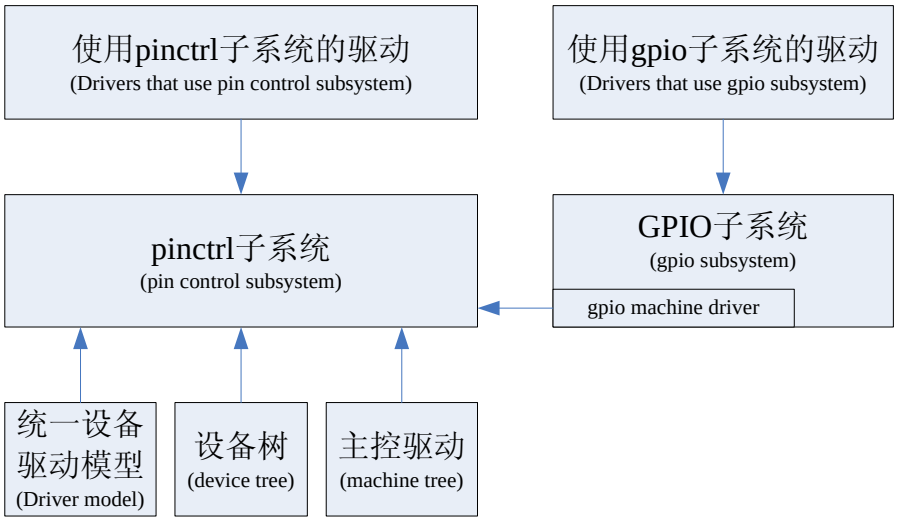
3.3 与 gpio 子系统的关系



multi-pin硬件关系框图

从图上可以看出，在炬芯的主控中，gpio 是凌驾于多功能 pin 选择器之上的，即 gpio block 是 always active 的，不论 multi-pin selector 选择哪一个功能，只要 gpio 使能，pin 就只能作为 gpio 使用，而其它的 pad block 提供的功能即使通过了 multi-pin selector 也无法体现在物理 pin 上。

为了避免 gpio 和 multi-pin selector 之间的冲突，导致不想看到的结果（例如，将某一组 pin 配置成 spi，但又通过 gpio 子系统申请到了这个 pin 上导出的 gpio，结果因为 gpio 优先级高，导致了申请到的 spi 功能无法使用）。我们需要将 gpio 子系统和 pinctrl 子系统的冲突管理起来。



pinctrl子系统关系图

上图显示了 gpio 子系统和 pinctrl 子系统之间的关系，即 pinctrl 子系统实际上也把 gpio 一起管理了起来（即使硬件设计上 gpio 不在 pin multi-selector 之下）。
所有的 gpio 操作也需要透过 pinctrl 子系统来完成，这样一来，如果一个 pin 已经被申请为 gpio，再通过 pinctrl 子系统申请为某个 function 时就会返回错误。

3.4 与统一设备驱动模型的关系

从“pinctrl 子系统关系图”中得知，linux kernel 中的统一设备驱动模型也调用了 pinctrl 子系统的功能。
linux kernel 中的统一设备驱动模型提供了 driver 和 device 的绑定机制，一旦匹配就会调用 driver 的 probe 函数。
而在调用 probe 函数之前，驱动模型实际上已经申请过一次 pin 了（前提是 dts 文件中该驱动节点中有定义名为"default"的 pinctrl 配置）

驱动模型中调用 driver 的 probe 函数的地方：

```
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 函数定义：

```
#define PINCTRL_STATE_DEFAULT "default"
int pinctrl_bind_pins(struct device *dev)
{
        int ret;

        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);
        ret = pinctrl_select_state(dev->pins->p, dev->pins->default_state);
        ……
        return 0;
}
```

从驱动模型的实现中，不难看出在驱动 probe 前，就已经申请到 default 的 pin 配置了，当然 pinctrl 的计数已经+1 了。
Driver 的 probe 函数可以通过 devm_pinctrl_get 获得 pinctrl 的句柄，再自行调用 pinctrl_select_state 设置 pin state

*注：
如果驱动在运行还想释放自己的 pinctrl 句柄（这种情况比较少见）只能采用非常规的手段了，先 devm_pinctrl_get 获得 pinctrl 句柄（这里 pinctrl 计数会再+1）,然后再 devm_pinctrl_put（pinctrl 计数-1），最后还得再调用一次 pinctrl_put（pinctrl 计数再-1，变成 0，并成功释放 pin）。

3.5 与 device tree 的关系

在 device tree source 文件(dts 文件)中，可以在驱动节点中定义该驱动需要用到的 pin 配置。

```
device-node-name {
```

```
        定义该 device 自己的属性
        pinctrl-names = "sleep", "default", “idle”;
        pinctrl-0 = &xxx_state_sleep;
        pinctrl-1 = &xxx_state_default;
        pinctrl-2 = &xxx_state_idle;
    };
```

pinctrl-0 pinctrl-1 pinctrl-2……表示了该设备的一个个的状态，这里我们定义了三个：pinctrl-0、pinctrl-1、pinctrl-2，数字 0、1、2 就是 pinctrl-names 中对应字符串数组的 index。其中 pinctrl-0 就是“sleep”状态，pinctrl-1 就是“default”状态，pinctrl-2 就是“idle”状态。而 xxx_state_sleep，xxx_state_default，xxx_state_idle 就是驱动具体的 pin 配置项了，需要在 pinctrl 设备节点处定义：

```
    pinctrl@b01b0040 {
            pinctrl-names = "default";
            pinctrl-0 = <&state_default>;

            state_default: pinctrl_default {
            };

            xxx_state_sleep: xxx_sleep {
                    xxx_mfp {
                            actions,groups = "mfp1_2_0", "mfp1_4_3";
                            actions,function = "spi0";
                    };
            };
```

Pinctrl 子系统在加载时，会调用 pinctrl_dt_to_map 函数将 dts 文件中有关 pinctrl 的配置项解析出来，并根据 dts 各驱动节点中对 pinctrl 的引用关系，将 phandle 挂到各个驱动的 device tree 子节点中，各个驱动就可以通过自己的 dev 句柄获得 pinctrl 的配置了。

3.6 与主控驱动的关系

在 kernel 的 machine driver 中，需要将主控的 pinctrl 相关硬件操作抽象成一个符合 linux pinctrl 子系统规范的结构 pinctrl_desc，并调用 pinctrl_register 注册到 pinctrl 子系统中，这样 pinctrl 子系统就可以将上层行为转换成具体的硬件操作了：

```
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;
};
```

其中成员 npins 表示可以控制多少个 pin，成员 pins 则描述每一个 pin 的信息为何。这两个成员就确定了一个 pin controller 所能控制的引脚的信息。

还有操作函数描述：

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 是用来配置引脚的特性（如：pull-up/pull-down）。

pctlops 成员 callback 函数说明：

callback 函数	描述
get_groups_count	该 pin controller 支持多少个 pin group
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	dt_node_to_map 的逆函数

pmxops 成员 callback 函数说明：

callback 函数	描述
request	Pinctrl 子系统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。
free	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	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

confops 成员 callback 函数说明：

callback 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

在 S500 方案中，主控驱动的源码位置如下：
kernel\arch\arm\mach-owl\pinctrl_data-atm7059.c
kernel\arch\arm\mach-owl\pinctrl_common-owl.c
其中，pinctrl_data-atm7059.c 定义了主控的所有 pin、group、mfp、pull-up/pull-down、pad_drive 等 IC 的寄存器及操作 bit 定义。
而 pinctrl_common-owl.c 则定义了所有操作函数，并将整合后的 struct pinctrl_desc 结构通过 pinctrl_register 注册到 pinctrl 子系统中。

4 Dts 文件中的 pinctrl 关键词表

4.1 Pin 命名表

pin 的命名遵循 IC spec 上的命名，以下命名表以每个 pin 默认的功能命名，但实际使用中各个 pin 的功能会随着配置发生变化。

在 dts 中使用关键词“actions,pins”后跟 pin 名字数组来定义需要使用哪些 pin，如：

```
i2c0_over_uart0_pull_up {
    actions,pins = "P_UART0_RX","P_UART0_TX";
    actions,pull = <2>;
};
```

模块	所属 pin 命名			
Ethernet MAC	P_ETH_TXD0	P_ETH_TXD1	P_ETH_TXEN	P_ETH_RXER
	P_ETH_CRS_DV	P_ETH_RXD1	P_ETH_RXD0	P_ETH_REF_CLK
	P_ETH_MDC	P_ETH_MDIO		
SIRQ	P_SIRQ0	P_SIRQ1	P_SIRQ2	
I2S	P_I2S_D0	P_I2S_BCLK0	P_I2S_LRCLK0	P_I2S_MCLK0
	P_I2S_D1	P_I2S_BCLK1	P_I2S_LRCLK1	P_I2S_MCLK1
PCM1	P_PCM1_IN	P_PCM1_CLK	P_PCM1_SYNC	P_PCM1_OUT
KEY	P_KS_IN0	P_KS_IN1	P_KS_IN2	P_KS_IN3
	P_KS_OUT0	P_KS_OUT1	P_KS_OUT2	
LVDS	P_LVDS_OEP	P_LVDS_OEN	P_LVDS_ODP	P_LVDS_ODN
	P_LVDS_OCP	P_LVDS_OCN	P_LVDS_OBP	P_LVDS_OBN
	P_LVDS_OAP	P_LVDS_OAN	P_LVDS_EEP	P_LVDS_EEN
	P_LVDS_EDP	P_LVDS_EDN	P_LVDS_ECP	P_LVDS_ECN
	P_LVDS_EBP	P_LVDS_EBN	P_LVDS_EAP	P_LVDS_EAN
	P_LCD0_D17	P_LCD0_D18		
DSI	P_DSI_DP3	P_DSI_DN3	P_DSI_DP1	P_DSI_DN1
	P_DSI_CP	P_DSI_CN	P_DSI_DP0	P_DSI_DN0
	P_DSI_DP2	P_DSI_DN2		
SD0	P_SD0_D0	P_SD0_D1	P_SD0_D2	P_SD0_D3
	P_SD0_CMD	P_SD0_CLK		
SD1	P_SD1_D0	P_SD1_D1	P_SD1_D2	P_SD1_D3

	P_SD1_CMD	P_SD1_CLK		
SPI	P_SPI0_SCLK	P_SPI0_SS	P_SPI0_MISO	P_SPI0_MOSI
UART0	P_UART0_RX	P_UART0_TX		
UART2	P_UART2_RX	P_UART2_TX	P_UART2_RTSB	P_UART2_CTSB
UART3	P_UART3_RX	P_UART3_TX	P_UART3_RTSB	P_UART3_CTSB
I2C	P_I2C0_SCLK	P_I2C0_SDATA	P_I2C1_SCLK	P_I2C1_SDATA
	P_I2C2_SCLK	P_I2C2_SDATA		
CSI	P_CSI_DN0	P_CSI_DP0	P_CSI_DN1	P_CSI_DP1
	P_CSI_CN	P_CSI_CP	P_CSI_DN2	P_CSI_DP2
	P_CSI_DN3	P_CSI_DP3		
Sensor	P_SENSOR0_PCLK	P_SENSOR0_CKOUT		
NAND	P_DNAND_D0	P_DNAND_D1	P_DNAND_D2	P_DNAND_D3
	P_DNAND_D4	P_DNAND_D5	P_DNAND_D6	P_DNAND_D7
	P_DNAND_WRB	P_DNAND_RDB	P_DNAND_RDBN	P_DNAND_DQS
	P_DNAND_DQSN	P_DNAND_RB0	P_DNAND_ALE	P_DNAND_CLE
	P_DNAND_CEB0	P_DNAND_CEB1	P_DNAND_CEB2	P_DNAND_CEB3

4.2 Mux Group-Function 表

4.2.1 Mux Group 命名表

pin group 按照寄存器的名字和 bits 来命名。比如：MFP_CTL1 bit22：21，定义了 10 个 pin 的 mux：P_LVDS_OEP, P_LVDS_OEN, P_LVDS_ODP, P_LVDS_ODN, P_LVDS_OCP, P_LVDS_OCN, P_LVDS_OBP, P_LVDS_OBN, P_LVDS_OAP, P_LVDS_OAN。该 pin group 的名字为”mfp1_22_21”。

有些 mfp 寄存器的 cell 中，设置某一个值会将多个 pin 配置为不同的功能。那么个 cell 中的 pin 就不能归为同一个 pin group。需要按情况拆解开。那么拆解开的 pin group 的名字还会加上一些后缀。

比如：根据 IC SPEC，mfp1 bit[31:29]可控制 3 根 pin：P_KS_IN0, P_KS_IN1, P_KS_IN2。该 cell 的 0x011 选项会将这 3 根 pin 分别定义为 pwm0, pwm1, pwm0。那么就将其分拆为 3 个 group：分别为“mfp1_31_29_ks_in2”、“mfp1_31_29_ks_in1”、“mfp1_31_29_ks_in0”。

这几个 mfp cell 分割以后，它们之间仍然存在硬件上的互斥关系。比如将 mfp1_31_29_ks_in2 的 function 选为 pwm0，mfp1_31_29_ks_in1 的 function 选为 pwm1，可以工作。但是将 mfp1_31_29_ks_in2 的 function 选为 pwm0，mfp1_31_29_ks_in1 的 function 选为 jtag，就会返回错误。

在 pin group 的命名表中还会看到 “XXX_dummy”的命名，这种 pin group 在 pinmux 设置中可能会用到。这些 pin group 只有一种 mux 功能，所以在 MFP 寄存器中不会表示，但是这些 pin 可能和 GPIO 复用，申请这些 pin 有助于发现它和 GPIO 存在的潜在复用错误。

在 dts 中使用关键词“actions,groups”后跟 Mux Group 名字数组来定义需要使用哪些 Mux Group，如：

```
sd0_mfp_cmd_clk {
    actions,groups = "mfp2_8_7", "mfp2_6_5";
    actions,function = "sd0";
};
```

Group 命名	**Gpoup 控制的 mux pin**		
mfp0_25_23	P_LCD0_D18		
mfp0_22_20	P_ETH_CRS_DV		
mfp0_18_16_eth_txd0	P_ETH_TXD0		
mfp0_18_16_eth_txd1	P_ETH_TXD1		
mfp0_15_13_rmii_txen	P_ETH_TXEN		
mfp0_15_13_rmii_rxen	P_ETH_RXER		
mfp0_10_8_rmii_rxd0	P_ETH_RXD0		
mfp0_10_8_rmii_rxd1	P_ETH_RXD1		
mfp0_7_6	P_ETH_REF_CLK		
mfp0_5	P_I2S_D0		
mfp0_4_3	P_I2S_LRCLK0	P_I2S_MCLK0	
mfp0_2_1_i2s0	P_I2S_BCLK0		
mfp0_2_1_i2s1	P_I2S_BCLK1	P_I2S_LRCLK1	P_I2S_MCLK1
mfp0_0	P_I2S_D1		
eth_smi_dummy	P_ETH_MDC	P_ETH_MDIO	
sirq0_dummy	P_SIRQ0		
sirq1_dummy	P_SIRQ1		
sirq2_dummy	P_SIRQ2		
mfp1_31_29_ks_in2	P_KS_IN2		
mfp1_31_29_ks_in1	P_KS_IN1		
mfp1_31_29_ks_in0	P_KS_IN0		
mfp1_28_26_ks_in3	P_KS_IN3		
mfp1_28_26_ks_out0	P_KS_OUT0		
mfp1_28_26_ks_out1	P_KS_OUT1		
mfp1_25_23	P_KS_OUT2		
mfp1_22_21	P_LVDS_OEP	P_LVDS_OEN	P_LVDS_ODP
	P_LVDS_ODN	P_LVDS_OCP	P_LVDS_OCN
	P_LVDS_OBP	P_LVDS_OBN	P_LVDS_OAP
	P_LVDS_OAN		
mfp1_20_19	P_DSI_DN0		

mfp1_18_17	P_DSI_DP2		
mfp1_16_14	P_LCD0_D17		
mfp1_13_12	P_DSI_DP3		
mfp1_11_10	P_DSI_DN3		
mfp1_9_7	P_DSI_DP0		
mfp1_6_5	P_LVDS_EEP	P_LVDS_EEN	
mfp1_4_3	P_SPI0_SCLK	P_SPI0_MOSI	
mfp1_2_0	P_SPI0_SS	P_SPI0_MISO	
mfp2_30_29	P_DSI_DP1	P_DSI_CP	P_DSI_CN
mfp2_28_27	P_LVDS_EDP	P_LVDS_EDN	P_LVDS_ECP
	P_LVDS_ECN	P_LVDS_EBP	P_LVDS_EBN
	P_LVDS_EAP	P_LVDS_EAN	
mfp2_26_24	P_DSI_DN2		
mfp2_23	P_UART2_RTSB		
mfp2_22	P_UART2_CTSB		
mfp2_21	P_UART3_RTSB		
mfp2_20	P_UART3_CTSB		
mfp2_19_17	P_SD0_D0		
mfp2_16_14	P_SD0_D1		
mfp2_13_11	P_SD0_D2	P_SD0_D3	
mfp2_10_9	P_SD1_D0	P_SD1_D1	P_SD1_D2
	P_SD1_D3		
mfp2_8_7	P_SD0_CMD		
mfp2_6_5	P_SD0_CLK		
mfp2_4_3	P_SD1_CMD		
mfp2_2_0	P_UART0_RX		
sd1_dummy	P_SD1_CLK		
mfp3_30	P_CLKO_25M		
mfp3_29_28	P_CSI_CN	P_CSI_CP	
mfp3_23_22	P_SENSOR0_CKOUT		
mfp3_21_19	P_UART0_TX		
mfp3_18_16	P_I2C0_SCLK	P_I2C0_SDATA	
mfp3_15_14	P_CSI_DN0	P_CSI_DN1	P_CSI_DN2
	P_CSI_DN3	P_CSI_DP0	P_CSI_DP1
	P_CSI_DP2	P_CSI_DP3	
mfp3_13_12	P_SENSOR0_PCLK		
mfp3_11_10	P_PCM1_IN		
mfp3_9_8	P_PCM1_CLK		
mfp3_7_6	P_PCM1_SYNC		

mfp3_5_4	P_PCM1_OUT		
mfp3_3	P_DNAND_D0	P_DNAND_D1	P_DNAND_D2
	P_DNAND_D3	P_DNAND_D4	P_DNAND_D5
	P_DNAND_D6	P_DNAND_D7	P_DNAND_RDB
	P_DNAND_RDBN		
mfp3_2	P_DNAND_ALE	P_DNAND_CLE	P_DNAND_CEB0
	P_DNAND_CEB1		
mfp3_1_0_nand_ceb2	P_DNAND_CEB2		
mfp3_1_0_nand_ceb3	P_DNAND_CEB3		
dsi_dummy	P_DSI_DN1		
nand_dummy	P_DNAND_WRB	P_DNAND_RB0	P_DNAND_DQS
	P_DNAND_DQSN		
uart2_dummy	P_UART2_RX	P_UART2_TX	
uart3_dummy	P_UART3_RX	P_UART3_TX	
i2c1_dummy	P_I2C1_SCLK	P_I2C1_SDATA	
i2c2_dummy	P_I2C2_SCLK	P_I2C2_SDATA	

4.2.2 Mux Function 命名表

每个 mux pin 可能使用到的功能都做了命名，每个 mux group 可通过使用 mux function 进行配置，以此完成可能的 mux。

在 dts 中使用关键词“actions,function”后跟 Mux Function 名字来定义“actions,groups”指定的 Mux Group 使用哪种功能，如：

```
sd0_mfp_cmd_clk {
    actions,groups = "mfp2_8_7", "mfp2_6_5";
    actions,function = "sd0";
};
```

模块	**Mux Function 命名**			
Norflash	nor			
Ethernet	eth_rmii	eth_smii		
SPI	Spi0	spi1	spi2	spi3
SENS	sens0	sens1		
UART	uart0	uart1	uart2	uart3
	uart4	uart5	uart6	
I2S	i2s0	i2s1		
PCM	pcm1	pcm0		
KEY	ks			

JTAG	jtag			
PWM	pwm0	pwm1	pwm2	pwm3
	pwm4	pwm5		
SD	sd0	sd1	sd2	
I2C	i2c0	i2c1	i2c2	i2c3
DSI	dsi			
LVDS	lvds			
USB3.0	usb30			
CLKOUT_25M	clko_25m			
MIPI_CSI	mipi_csi			
NAND	nand			
SPDIF	spdif			
SIRQ	sirq0	sirq1	sirq2	
TS	ts			
LCD	lcd0			

4.2.3 Mux Group-Function 对照表

Mux Group 所控制的 pin，可以通过设置 Mux Function 的方式改变 IC 内部连通性，使得同一组 pin 用作不同的功能，例如：

```
sd0_mfp_cmd_clk {
    actions,groups = "mfp2_8_7", "mfp2_6_5";
    actions,function = "jtag";
};
```

每个 mux group 可以设置的 function 如下表：

Mux Group	Mux Function			
mfp0_25_23	nor	sens1	pwm2	pwm4
	lcd0			
mfp0_22_20	eth_rmii	eth_smii	spi2	uart4
	pwm4			
mfp0_18_16_eth_txd0	eth_rmii	eth_smii	spi2	uart6
	pwm4			
mfp0_18_16_eth_txd1	eth_rmii	eth_smii	spi2	uart6
	pwm5			
mfp0_15_13_rmii_txen	eth_rmii	uart2	spi3	pwm0
mfp0_15_13_rmii_rxen	eth_rmii	uart2	spi3	pwm1
mfp0_10_8_rmii_rxd0	eth_rmii	uart2	spi3	pwm3

	uart5			
mfp0_10_8_rmii_rxd1	eth_rmii	uart2	spi3	pwm2
	uart5			
mfp0_7_6	eth_rmii	uart4	spi2	eth_smii
mfp0_5	i2s0	nor		
mfp0_4_3	i2s0	pcm1		
mfp0_2_1_i2s0	i2s0	nor	pcm0	
mfp0_2_1_i2s1	i2s1	nor	pcm0	
mfp0_0	i2s1	nor		
eth_smi_dummy	eth_rmii	eth_smii		
sirq0_dummy	sirq0			
sirq1_dummy	sirq1			
sirq2_dummy	sirq2			
mfp1_31_29_ks_in2	ks	jtag	nor	pwm0
	pwm4	sens1		
mfp1_31_29_ks_in1	ks	jtag	nor	pwm1
	pwm5	sens1	usb30	
mfp1_31_29_ks_in0	ks	jtag	nor	pwm0
	sens1			
mfp1_28_26_ks_in3	ks	jtag	nor	pwm1
	sens1			
mfp1_28_26_ks_out0	ks	uart5	nor	pwm2
	sens1	sd0		
mfp1_28_26_ks_out1	ks	jtag	nor	pwm3
	sens1	sd0		
mfp1_25_23	sd0	ks	nor	pwm2
	uart5	sens1		
mfp1_22_21	lvds	ts	lcd0	
mfp1_20_19	dsi	uart2	spi0	
mfp1_18_17	dsi	uart2	spi0	sd1
mfp1_16_14	nor	sd0	sd1	pwm3
	lcd0			
mfp1_13_12	nor	sd0	sd1	lcd0
mfp1_11_10	dsi	sd1	lcd0	
mfp1_9_7	dsi	sd0	uart2	spi0
mfp1_6_5	lvds	nor	ts	lcd0
mfp1_4_3	spi0	nor	i2c3	pcm0
mfp1_2_0	spi0	nor	i2s1	pcm1
	pcm0			

mfp2_30_29	dsi	sd1	lcd0	
mfp2_28_27	lvds	nor	lcd0	
mfp2_26_24	dsi	sd1	uart2	spi0
mfp2_23	uart2	uart0		
mfp2_22	uart2	uart0		
mfp2_21	uart3	uart5		
mfp2_20	uart3	uart5		
mfp2_19_17	sd0	nor	jtag	uart2
	uart5			
mfp2_16_14	sd0	nor	uart2	uart5
mfp2_13_11	sd0	nor	jtag	uart2
	uart1			
mfp2_10_9	sd0	nor	sd1	
mfp2_8_7	sd0	nor	jtag	
mfp2_6_5	sd0	jtag		
mfp2_4_3	sd1	nor		
mfp2_2_0	uart0	uart2	spi1	i2c0
	pcm1	i2s1		
sd1_dummy	sd1			
mfp3_30	clko_25m			
mfp3_29_28	mipi_csi	sens0		
mfp3_23_22	sens0	nor	sens1	pwm1
mfp3_21_19	uart0	uart2	spi1	i2c0
	spdif	pcm1	i2s1	
mfp3_18_16	i2c0	uart2	i2c1	uart1
	spi1			
mfp3_15_14	mipi_csi	sens0		
mfp3_13_12	sens0	nor	pwm0	
mfp3_11_10	pcm1	sens1	uart4	pwm4
mfp3_9_8	pcm1	sens1	uart4	pwm5
mfp3_7_6	pcm1	sens1	uart6	i2c3
mfp3_5_4	pcm1	sens1	uart6	i2c3
mfp3_3	nand	sd2		
mfp3_2	nand	spi2		
mfp3_1_0_nand_ceb2	nand	pwm5		
mfp3_1_0_nand_ceb3	nand	pwm4		
dsi_dummy	dsi			
nand_dummy	nand			
uart2_dummy	uart2			

uart3_dummy	uart3			
i2c1_dummy	i2c1			
i2c2_dummy	u2c2			

4.3 Drive Group 命名表

paddrv 使用的配置值完全等同于 IC spec 中关于 PAD_DRVx 寄存器的定义。
对于某些 pin 可以设置 pin 的驱动能力（即供电能力），可以通过配置 drive group 的等级对 pin 的驱动能力进行配置。

在 dts 中使用关键词“actions,paddrv”后跟驱动能力等级来定义“actions,groups”指定的 Drive Group 所代表的 pin 组使用哪种驱动能力，如：

```
sd0_d0_d3_cmd_clk_paddrv    {
        actions,groups    = "paddrv1_19_18","paddrv1_17_16";
        actions,paddrv    = <1>; /*level 1，range：0~3*/
};
```

表示"paddrv1_19_18"所代表的 P_SD0_CMD 和"paddrv1_17_16"所代表的 P_SD0_CLK，使用驱动能力 1，来提升数据传输稳定性。

Drive Group 名	设置范围	对应 pin		
paddrv0_29_28	0~2	P_SIRQ0	P_SIRQ1	P_SIRQ2
paddrv0_23_22	0~2	P_ETH_TXD0	P_ETH_TXD1	P_ETH_TXEN
paddrv0_21_20	0~2	P_ETH_RXER		
paddrv0_19_18	0~2	P_ETH_CRS_DV		
paddrv0_17_16	0~2	P_ETH_RXD0	P_ETH_RXD1	
paddrv0_15_14	0~2	P_ETH_REF_CLK		
paddrv0_13_12	0~2	P_ETH_MDC	P_ETH_MDIO	
paddrv0_11_10	0~2	P_I2S_D0		
paddrv0_9_8	0~2	P_I2S_BCLK0		
paddrv0_7_6	0~2	P_I2S_LRCLK0	P_I2S_MCLK0	P_I2S_D1
paddrv0_5_4	0~2	P_I2S_BCLK1	P_I2S_LRCLK1	P_I2S_MCLK1
paddrv0_3_2	0~2	P_PCM1_IN	P_PCM1_CLK	P_PCM1_SYNC
		P_PCM1_OUT		
paddrv0_1_0	0~2	P_KS_IN0	P_KS_IN1	P_KS_IN2
		P_KS_IN3		
paddrv1_31_30	0~2	P_KS_OUT0	P_KS_OUT1	P_KS_OUT2
paddrv1_29_28	0~2	P_LVDS_OEP	P_LVDS_OEN	P_LVDS_ODP
		P_LVDS_ODN	P_LVDS_OCP	P_LVDS_OCN

		P_LVDS_OBP	P_LVDS_OBN	P_LVDS_OAP
		P_LVDS_OAN	P_LVDS_EEP	P_LVDS_EEN
		P_LVDS_EDP	P_LVDS_EDN	P_LVDS_ECP
		P_LVDS_ECN	P_LVDS_EBP	P_LVDS_EBN
		P_LVDS_EAP	P_LVDS_EAN	
paddrv1_27_26	0~2	P_DSI_DP3	P_DSI_DN3	P_DSI_DP1
		P_DSI_DN1	P_DSI_CP	P_DSI_CN
paddrv1_25_24	0~2	P_DSI_DP0	P_DSI_DN0	P_DSI_DP2
		P_DSI_DN2		
paddrv1_23_22	0~3	P_SD0_D0	P_SD0_D1	P_SD0_D2
		P_SD0_D3		
paddrv1_21_20	0~2	P_SD1_D0	P_SD1_D1	P_SD1_D2
		P_SD1_D3		
paddrv1_19_18	0~3	P_SD0_CMD		
paddrv1_17_16	0~3	P_SD0_CLK		
paddrv1_15_14	0~2	P_SD1_CMD		
paddrv1_13_12	0~2	P_SD1_CLK		
paddrv1_11_10	0~2	P_SPI0_SCLK	P_SPI0_SS	P_SPI0_MISO
		P_SPI0_MOSI		
paddrv2_31_30	0~2	P_UART0_RX		
paddrv2_29_28	0~2	P_UART0_TX		
paddrv2_27_26	0~2	P_UART2_RX	P_UART2_TX	P_UART2_RTSB
		P_UART2_CTSB		
paddrv2_24_23	0~2	P_I2C0_SCLK	P_I2C0_SDATA	
paddrv2_22_21	0~2	P_I2C1_SCLK	P_I2C1_SDATA	P_I2C2_SCLK
		P_I2C2_SDATA		
paddrv2_19_18	0~2	P_SENSOR0_PCLK		
paddrv2_13_12	0~2	P_SENSOR0_CKOUT		
paddrv2_3_2	0~2	P_UART3_RX	P_UART3_TX	P_UART3_RTSB
		P_UART3_CTSB		

4.4 Pin Pull Up/Down 对照表

在 dts 中使用关键词“actions,pull”后跟上下拉数据来定义“actions,pins”指定的 pin 组使用哪种上下拉，如：

```
sd0_pull_d0_d3_cmd {
    actions,pins = "P_SD0_CMD", "P_SD0_CLK";
    actions,pull = <1>;
```

};
表示将 P_SD0_CMD 和 P_SD0_CLK 这两个 pin 下拉。

actions,pull 值的含义：
enum owl_pinconf_pull {
    OWL_PINCONF_PULL_DISABLE = 0,   //关闭上下拉功能
    OWL_PINCONF_PULL_DOWN = 1,   //该 pin 下拉
    OWL_PINCONF_PULL_UP = 2,   //该 pin 上拉
};

actions,pull = <0>，表示上下拉功能关闭。
actions,pull = <1>，表示下拉。
actions,pull = <2>，表示上拉。

*请注意，并不是所有的 pin 都有上下拉功能。而且，有上下拉功能的 pin，也可能只有上拉或只有下拉，具体的功能对照表如下：

pin 名	关闭可用	上拉可用	下拉可用
P_PCM1_SYNC	●	●	
P_PCM1_OUT	●	●	
P_KS_OUT2	●	●	
P_LCD0_D17	●	●	
P_DSI_DN3	●	●	
P_ETH_RXER	●		●
P_SIRQ0	●	●	●
P_SIRQ1	●	●	●
P_SIRQ2	●	●	●
P_I2C0_SDATA	●	●	
P_I2C0_SCLK	●	●	
P_KS_IN0	●	●	
P_KS_IN1	●	●	
P_KS_IN2	●	●	
P_KS_IN3	●	●	
P_KS_OUT0	●	●	
P_KS_OUT1	●	●	
P_DSI_DP1	●	●	
P_DSI_CP	●	●	
P_DSI_CN	●	●	
P_DSI_DN2	●	●	
P_DNAND_RDBN	●	●	

P_SD0_D0	●	●	
P_SD0_D1	●	●	
P_SD0_D2	●	●	
P_SD0_D3	●	●	
P_SD0_CMD	●	●	
P_SD0_CLK	●	●	
P_SD1_CMD	●	●	
P_SD1_D0	●	●	
P_SD1_D1	●	●	
P_SD1_D2	●	●	
P_SD1_D3	●	●	
P_UART0_RX	●	●	
P_UART0_TX	●	●	
P_CLKO_25M	●		●
P_SPI0_SCLK	●	●	
P_SPI0_MOSI	●	●	
P_I2C1_SDATA	●	●	
P_I2C1_SCLK	●	●	
P_I2C2_SDATA	●	●	
P_I2C2_SCLK	●	●	
P_DNAND_DQSN	●	●	
P_DNAND_DQS	●	●	●
P_DNAND_D0	●	●	
P_DNAND_D1	●	●	
P_DNAND_D2	●	●	
P_DNAND_D3	●	●	
P_DNAND_D4	●	●	
P_DNAND_D5	●	●	
P_DNAND_D6	●	●	
P_DNAND_D7	●	●	

4.5 GPIO-PIN 对照表

pin 除了可以复用作各种功能外，还可配置成 GPIO 使用，在章节“3.3 与 gpio 子系统的关系”中，有说明 pinctrl 子系统将 GPIO 子系统也管理了起来，因此申请 gpio 时会检查当前该 gpio 所对应的 pin 是否已经被其它驱动申请用作其它功能了，如果已经被申请，则申请 gpio 时会报错。反之亦然。

GPIO	PIN
GPIOA14	P_ETH_TXD0
GPIOA15	P_ETH_TXD1
GPIOA16	P_ETH_TXEN
GPIOA17	P_ETH_RXER
GPIOA18	P_ETH_CRS_DV
GPIOA19	P_ETH_RXD1
GPIOA20	P_ETH_RXD0
GPIOA21	P_ETH_REF_CLK
GPIOA22	P_ETH_MDC
GPIOA23	P_ETH_MDIO
GPIOA24	P_SIRQ0
GPIOA25	P_SIRQ1
GPIOA26	P_SIRQ2
GPIOA27	P_I2S_D0
GPIOA28	P_I2S_BCLK0
GPIOA29	P_I2S_LRCLK0
GPIOA30	P_I2S_MCLK0
GPIOA31	P_I2S_D1
GPIOB0	P_I2S_BCLK1
GPIOB1	P_I2S_LRCLK1
GPIOB2	P_I2S_MCLK1
GPIOB3	P_KS_IN0
GPIOB4	P_KS_IN1
GPIOB5	P_KS_IN2
GPIOB6	P_KS_IN3
GPIOB7	P_KS_OUT0
GPIOB8	P_KS_OUT1
GPIOB9	P_KS_OUT2
GPIOB10	P_LVDS_OEP
GPIOB11	P_LVDS_OEN
GPIOB12	P_LVDS_ODP
GPIOB13	P_LVDS_ODN
GPIOB14	P_LVDS_OCP
GPIOB15	P_LVDS_OCN
GPIOB16	P_LVDS_OBP
GPIOB17	P_LVDS_OBN
GPIOB18	P_LVDS_OAP

GPIOB19	P_LVDS_OAN
GPIOB20	P_LVDS_EEP
GPIOB21	P_LVDS_EEN
GPIOB22	P_LVDS_EDP
GPIOB23	P_LVDS_EDN
GPIOB24	P_LVDS_ECP
GPIOB25	P_LVDS_ECN
GPIOB26	P_LVDS_EBP
GPIOB27	P_LVDS_EBN
GPIOB28	P_LVDS_EAP
GPIOB29	P_LVDS_EAN
GPIOB30	P_LCD0_D18
GPIOB31	P_LCD0_D17
GPIOC0	P_DSI_DP3
GPIOC1	P_DSI_DN3
GPIOC2	P_DSI_DP1
GPIOC3	P_DSI_DN1
GPIOC4	P_DSI_CP
GPIOC5	P_DSI_CN
GPIOC6	P_DSI_DP0
GPIOC7	P_DSI_DN0
GPIOC8	P_DSI_DP2
GPIOC9	P_DSI_DN2
GPIOC10	P_SD0_D0
GPIOC11	P_SD0_D1
GPIOC12	P_SD0_D2
GPIOC13	P_SD0_D3
GPIOC14	P_SD0_D4
GPIOC15	P_SD0_D5
GPIOC16	P_SD0_D6
GPIOC17	P_SD0_D7
GPIOC18	P_SD0_CMD
GPIOC19	P_SD0_CLK
GPIOC20	P_SD1_CMD
GPIOC21	P_SD1_CLK
GPIOC22	P_SPI0_SCLK
GPIOC23	P_SPI0_SS
GPIOC24	P_SPI0_MISO

GPIOC25	P_SPI0_MOSI
GPIOC26	P_UART0_RX
GPIOC27	P_UART0_TX
GPIOC28	P_I2C0_SCLK
GPIOC29	P_I2C0_SDATA
GPIOC31	P_SENSOR0_PCLK
GPIOD10	P_SENSOR0_CKOUT
GPIOD12	P_DNAND_ALE
GPIOD13	P_DNAND_CLE
GPIOD14	P_DNAND_CEB0
GPIOD15	P_DNAND_CEB1
GPIOD16	P_DNAND_CEB2
GPIOD17	P_DNAND_CEB3
GPIOD18	P_UART2_RX
GPIOD19	P_UART2_TX
GPIOD20	P_UART2_RTSB
GPIOD21	P_UART2_CTSB
GPIOD22	P_UART3_RX
GPIOD23	P_UART3_TX
GPIOD24	P_UART3_RTSB
GPIOD25	P_UART3_CTSB
GPIOD28	P_PCM1_IN
GPIOD29	P_PCM1_CLK
GPIOD30	P_PCM1_SYNC
GPIOD31	P_PCM1_OUT
GPIOE0	P_I2C1_SCLK
GPIOE1	P_I2C1_SDATA
GPIOE2	P_I2C2_SCLK
GPIOE3	P_I2C2_SDATA

5 使用 dts 描述 pinctrl 配置

5.1 Dts 中 Pinctrl 配置方法

所有的 pinctrl-state 都定义在 pin controller device 节点中：

```
pinctrl@b01b0040 {
        pinctrl-names = "default";
        pinctrl-0 = <&state_default>;

        state_default: pinctrl_default {
        };

        mmc_share_uart_state: mmc_share_uart {
                ……
        };
        mmc0_state_default: mmc0_default {
                ……
        };
};
```

而各驱动引用定义在 pin controller device 节点中的子节点，即 pinctrl-state 节点。

各驱动使用如下方法引用 pinctrl-state 节点：

pinctrl-N：描述该设备需要使用的 pin 的一个状态(pin state)，相当于上述的 state。N 的数值必须从 0 开始顺序递增。pinctrl-N 属性的值为 pin configuration nodes 的 phandle。pinctrl-N 属性引用的 pin configuration nodes 必须是 pin controller device node 的直接子节点。

pinctrl-names：为每个 pin state 定义一个名字。每个名字顺序对应一个 pin state。比如 pinctrl-0 的名字为“default”，pinctrl-1 的名字对应“idle”。若不指定 pinctrl-names 属性，这样的话，pin state 的名字就是该属性的“N”字符。比如 pinctrl-0 的名字为字符“0”。

```
mmc@b0230000 {
        pinctrl-names = "default","share_uart2_5";
        pinctrl-0 = <&mmc0_state_default>;
        pinctrl-1 = <&mmc_share_uart_state>;
};
```

上面的例子中，mmc 驱动定义的 pinctrl-state 有两个，其中 mmc0_state_default 是定义 pin controller device node 下的直接子节点，表示 sd0 的 pin group 作为 sd 功能使用，而 mmc_share_uart_state 则表示作为 serial 功能使用。其中 mmc0_state_default 对应的

pinctrl-names 属性为 default，而 mmc_share_uart_state 对应的 pinctrl-names 属性为 share_uart2_5。

5.2 pin mapping database 的建立

pin controller device 节点配置好后，内核是如何获取到 pin mapping database 并使用的呢？

通过查看 pinctrl 子系统代码，了解到 pin mapping database 建立有两种情况：

一种情况是主控驱动调用 pinctrl_register 注册 machine 的 struct pinctrl_desc 结构到 pinctrl 子系统时，由 pinctrl_register 调用 pinctrl_get 创建。

另一种就是各驱动在加载时由统一设备驱动模型在 driver 和 device 绑定时调用 devm_pinctrl_get 转而调用 pinctrl_get 创建。（绑定过程在“3.4 与统一设备驱动模型的关系”章节中有描述）

pinctrl_get 获取 pin mapping database 的过程大致如下：

1） pinctrl_get 函数调用 create_pinctrl 来创建 pinctrl 句柄
2） create_pinctrl 函数调用 pinctrl_dt_to_map 来解析 dts 当前设备节点中的命名为 pinctrl-N 的属性信息
3） pinctrl_dt_to_map 搜索当前设备驱动节点所有的 pinctrl-%d，并通过其指向的 phandle 找到定义在 pin controller device node 中的 pinctrl state 节点，然后调用 dt_to_map_one_config 进行解析。
4） dt_to_map_one_config 函数向上搜索 pinctrl state 节点的父亲节点，并与所有注册的 pinctrl 驱动比对，找到匹配的 pinctrl 驱动的句柄后，调用 pinctrl 驱动注册的 dt_node_to_map 函数来解析 pinctrl state 节点数据。
5） 主控 pinctrl 驱动的 owl_pinctrl_dt_node_to_map 收到 pinctrl state 节点句柄后，搜索以下属性："actions,function"、"actions,pull"、"actions,paddrv"、"actions,pins"、"actions,groups"，并将解析的数据打包成 struct pinctrl_map 结构，并回传给调用者。

至此，pinctrl_get 就完成了收集并组装 pin mapping database 工作，并返回了 pinctrl 的控制句柄。设备驱动可以通过这个句柄来完成对 pinctrl 的各项操作。

5.2.1 pin controller device

pin controller device 包含一系列的 pin configuration node 作为子节点。这些子节点描述了 pinctrl_map 中可能用到的所有 pin state。

pinctrl 模块对这些 pin configuration node 定义如下：

1. actions,groups 的属性用于表示 Mux Group 的名字数组。
2. actions,pins 的属性用于表示该组 Pin 的名字数组。
3. actions,function 的属性表示该组 Mux Group 使用功能的名字。

4. actions,paddrv 的属性表示该组 Dirver Group 所代表的 pin 的驱动能力。
5. actions,pull 的属性表示该组 Pin 需要配置的上下拉状态。
6. pin configuration node 的名字任意。
7. pin group、pin、function 的有效名字，paddrv、pull 的有效值定义，请参见“4 Dts 文件中的 pinctrl 关键词表”章节。

pin controller 的 device node 定义为 pinctrl@b01b0040，示例如下：

```
pinctrl@b01b0040 {//-----------------------------------------------------(A)
        pinctrl-names = "default";
        pinctrl-0 = <&state_default>;     //-----------------------------------------------------(B)

        state_default: pinctrl_default {   //-----------------------------------------------------(C)
            serial5_state_default: serial5_default{
                serial_5{
                    actions,groups = "mfp2_19_17","mfp2_16_14";
                    actions,function = "uart5";
                };
            };
        };

        nand_state_default: nand_default {
            nand_mfp {
                actions,groups = "nand_dummy", "mfp3_3", "mfp3_2", "mfp3_1_0_nand_ceb2",
"mfp3_1_0_nand_ceb3";
                actions,function = "nand";
            };
            nand_dqsn_pullup {
                actions,pins = "P_DNAND_DQSN";
                actions,pull = <2>;    //pull up
            };
        };
        nand_state_sleep: nand_sleep {
            nand_dqsn_pullup {
                actions,pins = "P_DNAND_DQSN";
                actions,pull = <0>;    //pull disabled
            };
        };
    };
```

说明：

(A) pinctrl@b01b0040 其实就是一个设备节点，这个设备节点的解析是由 pinctrl 主控驱动完

成的，主控驱动在初始化时会调用 pinctrl_register 函数将自己注册到 pinctrl 子系统中。

(B) pinctrl_register 会调用 pinctrl_get 来获取属于主控驱动的 pin mapping database。所以，pinctrl-names 为"default"的 pinctrl-state 是不是也会被使能呢？答案是肯定的。如同“与统一设备驱动模型的关系”章节中一样，pinctrl_register 在获得 pinctrl 句柄后，会申请 default 的 pin 配置。所以如果有不属于任何其它设备的 pinctrl 配置，但也想使能，就可以统一放到这里进行初始化。

(C) 其它的子节点，则由各驱动在加载时由统一设备驱动模型在 driver 和 device 绑定时调用 pinctrl_get 时被解析，并将解析完成的 pinctrl 句柄存放到驱动设备的 device 中。

5.2.2 驱动设备

如“5.4 与统一设备驱动模型的关系”章节中所描述，统一设备模式在匹配到驱动后，会调用 devm_pinctrl_get 来建立 pinctrl database map。

在具体的驱动设备节点中，通过关键字“pinctrl-N”来引用在 pin controller device 设备节点中定义的 pinctrl state 节点。其中“N”表示序号，是 pinctrl-names 定义的字符串数组下标。

```
nand@b0210000 {
        compatible = "actions,atm7059a-nand";
        reg = <0xB0210000 0xb4>, <0xB0160000 0xfc>, <0xB01B0000 0x90>, <0xB0260000
0xd00>;
        interrupts = < 0 41 0x4 >;
        clock-source = "NANDPLL";
        pinctrl-names = "default", “sleep”;
        pinctrl-0 = <&nand_state_default>;
        pinctrl-1 = <&nand_state_sleep>;
        status = "okay";
};
```

说明：

(A) nand@b0210000 是定义在 root 节点下的一级子节点。

(B) 在驱动调用 platform_driver_register(&PDL_NandDev_platform_driver)时由统一设备模型对“compatible”属性进行匹配，匹配到后就调用 pinctrl_bind_pins 进行绑定，并调用 pinctrl_get 解析“pinctrl-N”引用的 pinctrl 子节点，并将 pinctrl 句柄放到 platform_device 结构的 dev 结构中。

(C) 驱动通过 Pinctrl 驱动提供的 pinctrl_get 接口从 dev 结构中获得 pinctrl 句柄的引用，并调用其它 pinctrl 驱动提供的接口对 pinctrl 进行操作。

6 kernel 接口使用说明

6.1 Pinctrl 驱动提供给内核驱动使用的接口

接口原型	说明
struct pinctrl *pinctrl_get(struct device *dev)	驱动加载成功并绑定后, pinctrl 的控制句柄就存放到 dev 结构中了，可以通过该函数得到 pinctrl 的控制句柄。（pinctrl 句柄是如何生成的，请参看 6.4 “pin mapping database 的建立”章节）
struct pinctrl_state *pinctrl_lookup_state(struct pinctrl *p, const char *name)	通过输入 state 的名字，来找到对应的 pinctrl_state 结构，与 pinctrl_select_state 配合使用。
int pinctrl_select_state(struct pinctrl *p, struct pinctrl_state *state)	选择 state，在这个调用中会将 state 所代表的硬件 pinctrl 配置设置到 IC 对应的 pinctrl 寄存器中。
void pinctrl_put(struct pinctrl *p)	释放 pinctrl 控制句柄。
static inline struct pinctrl * __must_check pinctrl_get_select(struct device *dev, const char *name)	作用等同于 pinctrl_get + pinctrl_lookup_state + pinctrl_select_state
static inline struct pinctrl * __must_check pinctrl_get_select_default(struct device *dev)	作用等同于 pinctrl_get_select(dev, “default”);

Devm 类型接口：

接口原型	说明
struct pinctrl *devm_pinctrl_get(struct device *dev)	作用同 pinctrl_get
void devm_pinctrl_put(struct pinctrl *p)	作用同 pinctrl_put
static inline struct pinctrl * __must_check devm_pinctrl_get_select(struct device *dev, const char *name)	作用同 pinctrl_get_select
static inline struct pinctrl * __must_check	作用同 pinctrl_get_select_default

devm_pinctrl_get_select_default(struct device *dev)	

6.2 驱动使用 pinctrl 接口代码示例

示例 1(基本接口使用)

```
#include <linux/pinctrl/consumer.h>

struct pinctrl *p;
struct pinctrl_state *s_default;
struct pinctrl_state *s_share_uart2_5;

static int __init owl_mmc_probe(struct platform_device *pdev)
{
    int ret;

    //进入 probe，系统已经默认被设置为 default 的 state（详见 5.4 “与统一设备驱动模型的关系”章节），如果没有设置除 default 外的其它 state 的需求，下面的代码就可以忽略了。如果需要设置为除 default 以外的其它 state，需要显式调用 pinctrl_select_state 进行设置。

    p = pinctrl_get(&pdev->dev);
    if (IS_ERR(p)) {
        return PTR_ERR(p);
    }
    s_default = pinctrl_lookup_state(p, "default");
    if (IS_ERR(s)) {
        pinctrl_put(p);
        return PTR_ERR(s);
    }
    s_share_uart2_5 = pinctrl_lookup_state(p, "share_uart2_5");
    if (IS_ERR(s)) {
        pinctrl_put(p);
        return PTR_ERR(s);
    }
    ret = pinctrl_select_state(p, s_share_uart2_5);
    if (ret < 0) {
        pinctrl_put(p);
        return ret;
    }
```

```
    return 0;
}

static int owl_mmc_remove(struct platform_device *pdev)
{
    pinctrl_put(p);
}
```

示例 2（更精简的方法）

```
#include <linux/pinctrl/consumer.h>

struct pinctrl *p;

static int __init owl_mmc_probe(struct platform_device *pdev)
{
    p = pinctrl_get_select(&pdev->dev, "share_uart2_5");
    if (IS_ERR(p)) {
        return PTR_ERR(s);
    }
    return 0;
}

static int owl_mmc_remove(struct platform_device *pdev)
{
    //这里不需要调用 devm_pinctrl_put，因为驱动模型框架会自动释放
}
```

7 uboot 接口使用说明

在 uboot 阶段，也提供了相应的驱动程序来完成 pinctrl 的功能。这个阶段的 pinctrl 驱动相对简单一些，和 kernel 的驱动程序有一些不同。

1. 提供接口帮助设备驱动完成 mfp、paddrv、pull 的配置。
2. 帮助解析 devicetree
3. 不提供 pin 的互斥功能，也就是说对相同的 pin 进行再一次配置会覆盖之前的配置，不会失败。

pinctrl 在 board_init 时进行了初始化，这时候已经可以使用 devicetree，所以设备驱动只需要将 devicetree 的相关 node 送给 pinctrl，就可以帮助设备驱动完成所有的 pin 的配置。这个阶段提供了 2 个接口：

1、int owl_fdtdec_set_pinctrl(int offset);

该接口需要设备驱动提供代表 pinctrl state 的 pin configuration node。该 pin configuationnode 通常由设备的 pinctrl-N 属性的 phandle 值指定。调用以后，pinctrl 驱动会完成该 pin configuration node 下的相关 pin 配置。

offset：代表一个 pin configuration node

2、int owl_device_fdtdec_set_pinctrl_default(int dev_offset);

由于设备的默认配置（也是大多数设备的唯一配置）会出现在 pinctrl-0，对于这种情况，提供了一个更方便的接口。该接口需要设备驱动提供代表设备本身的 node。调用以后，pinctrl 驱动会从该设备的 node 下找到 pinctrl-0 属性指向的 pin configuration node，进行配置。

dev_offset：代表一个设备的 node

示例：

```
static int acts_mmc_fdt_init(struct owl_mmc_host *host)
{
    const void *blob = gd->fdt_blob;
    int node_list[1], count;
    //找到设备节点
    count = fdtdec_find_aliases_for_id(blob, 0,
        COMPAT_ACTIONS_OWL_SDMMC, node_list, 1);
    if (count < 0 || node_list[0] < 0)
        return -1;
    //在设备节点中找到 pinctrl-0 指向的 state，并设置 pinctrl 配置
    owl_device_fdtdec_set_pinctrl_default(node_list[0]);
    return 0;
```

}

8 版本历史

日期	版本号	注释	作者
2015/12/16	1.0	建立初始版本	ActDuino S500 项目组

9 声 明

Disclaimer

Information given in this document is provided just as a reference or example for the purpose of using Actions' products, and cannot be treated as a part of any quotation or contract for sale.

Actions products may contain design defects or errors known as anomalies or errata which may cause the products' functions to deviate from published specifications. Designers must not rely on the instructions of Actions' products marked "reserved" or "undefined". Actions reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

ACTIONS DISCLAIMS AND EXCLUDES ANY AND ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY AND ALL EXPRESS OR IMPLIED WARRANTIES OF MERCHANTABILITY, ACCURACY, SECURITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND AGAINST INFRINGEMENT OF INTELLECTUAL PROPERTY AND THE LIKE TO THE INFORMATON OF THIS DOCUMENT AND ACTIONS PRODUCTS.

IN NO EVENT SHALL ACTIONS BE LIABLE FOR ANY DIRECT, INCIDENTAL, INDIRECT, SPECIAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING, WITHOUT LIMITATION FOR LOST OF DATA, PROFITS, SAVINGS OR REVENUES OF ANY KIND ARISING FROM USING THE INFORMATON OF THIS DOCUMENT AND ACTIONS PRODUCTS. REGARDLESS OF THE FORM OF ACTION, WHETHER BASED ON CONTRACT; TORT; NEGLIGENCE OF ACTIONS OR OTHERS; STRICT LIABILITY; OR OTHERWISE; WHETHER OR NOT ANY REMEDY OF BUYER IS HELD TO HAVE FAILED OF ITS ESSENTIAL PURPOSE, AND WHETHER ACTIONS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES OR NOT.

Actions' products are not designed, intended, authorized or warranted for use in any life support or other application where product failure could cause or contribute to personal injury or severe property damage. Any and all such uses without prior written approval of an Officer of Actions and further testing and/or modification will be fully at the risk of the customer.

Ways of obtaining information

Copies of this document and/or other Actions product literature, as well as the Terms and Conditions of Sale Agreement, may be obtained by visiting Actions' website at: http://www.actions-semi.com or from an authorized Actions representative.

Trademarks

The word “Actions” and the logo are the trademarks of Actions Semiconductor Co., Ltd, and Actions (Zhuhai) Technology Co., Limited is authorized to use them. Word “炬芯” is the trademark of Actions (Zhuhai) Technology Co., Limited. Names and brands of other companies and their products that may from time to time descriptively appear in this document are the trademarks of their respective holders, no affiliation, authorization, or endorsement by such persons are claimed or implied except as may be expressly stated therein.

Rights Reserved

The provision of this document shall not be deemed to grant buyers any right in and to patent, copyright, trademark, trade secret, know how, and any other intellectual property of Actions or others.

Miscellaneous

Information contained or described herein relates only to the Actions products and as of the release date of this publication, abrogates and supersedes all previously published data and specifications relating to such products provided by Actions or by any other person purporting to distribute such information.

Actions reserves the rights to make changes to information described herein at any time without notice. Please contact your Actions sales representatives to obtain the latest information before placing your product order.

Additional Support

Additional products and company information can be obtained by visiting the Actions website at: http://www.actions-semi.com

支持：

如欲获得公司及产品的其它信息，欢迎访问我公司网站：http://www.actions-semi.com

炬芯（珠海）科技有限公司

地址：珠海市唐家湾镇高新区科技四路 1 号 1#厂房一层 C 区
电话：+86-756-3392353
传真：+86-756-3392251
邮政编码：519085
网址：http://www.actions-semi.com
电子邮件 （业务）: mp-sales@actions-semi.com
（技术支持）: mp-cs@actions-semi.com