

16. GPIO 和 Pinctrl 子系统的使用

参考文档：

a. 内核 Documentation\devicetree\bindings\Pinctrl\ 目录下：

Pinctrl-bindings.txt

b. 内核 Documentation\gpio 目录下：

Pinctrl-bindings.txt

c. 内核 Documentation\devicetree\bindings\gpio 目录下：

gpio.txt

注意：本章的重点在于“使用”，深入讲解放在“驱动大全”的视频里。

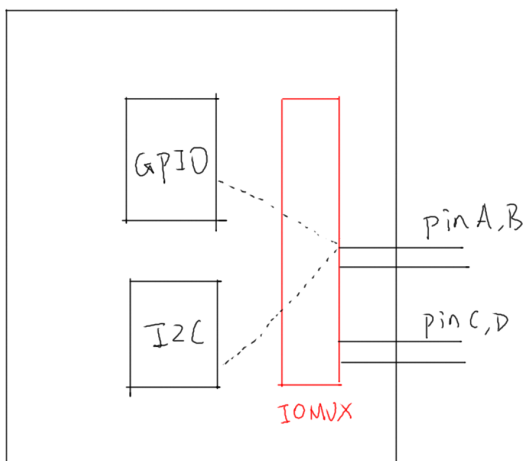
前面的视频，我们使用直接操作寄存器的方法编写驱动。这只是为了让大家掌握驱动程序的本质，在实际开发过程中我们可不这样做，太低效了！如果驱动开发都是这样去查找寄存器，那我们就变成“寄存器工程师”了，即使是做单片机的都不执着于裸写寄存器了。

Linux 下针对引脚有 2 个重要的子系统：GPIO、Pinctrl。

16.1 Pinctrl 子系统重要概念

16.1.1 引入

无论是哪种芯片，都有类似下图的结构：



要想让 pinA、B 用于 GPIO，需要设置 IOMUX 让它们连接到 GPIO 模块；

要想让 pinA、B 用于 I2C，需要设置 IOMUX 让它们连接到 I2C 模块。

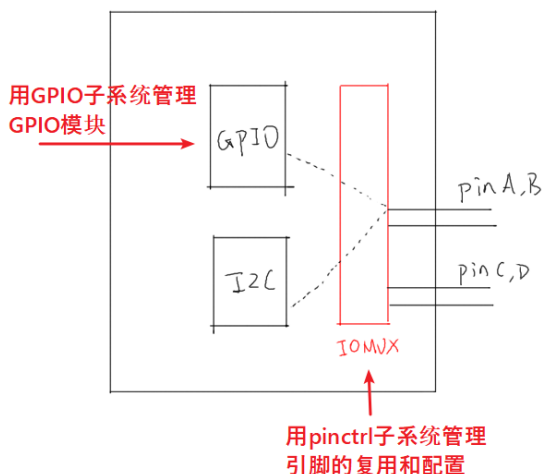
所以 GPIO、I2C 应该是并列的关系，它们能够使用之前，需要设置 IOMUX。有时候并不仅仅是设置 IOMUX，还要配置引脚，比如上拉、下拉、开漏等等。

现在的芯片动辄几百个引脚，在使用到 GPIO 功能时，让你一个引脚一个引脚去找对应的寄存器，这要疯掉。术业有专攻，这些累活就让芯片厂家做吧——他们是 BSP 工程师。我们在他们的基础上开发，我们

是驱动工程师。开玩笑的，BSP 工程师是更懂他自家的芯片，但是如果驱动工程师看不懂他们的代码，那你的进步也有限啊。

所以，要把引脚的复用、配置抽出来，做成 Pinctrl 子系统，给 GPIO、I2C 等模块使用。

BSP 工程师要做什么？看下图：



等 BSP 工程师在 GPIO 子系统、Pinctrl 子系统中把自家芯片的支持加进去后，我们就可以非常方便地使用这些引脚了：点灯简直太简单了。

等等，GPIO 模块在图中跟 I2C 不是并列的吗？干嘛在讲 Pinctrl 时还把 GPIO 子系统拉进来？

大多数的芯片，没有单独的 IOMUX 模块，引脚的复用、配置等等，就是在 GPIO 模块内部实现的。

在硬件上 GPIO 和 Pinctrl 是如此密切相关，在软件上它们的关系也非常密切。

所以这 2 个子系统我们一起讲解。

16.1.2 重要概念

从设备树开始学习 Pintrl 会比较容易。

主要参考文档是：内核 Documentation\devicetree\bindings\pinctrl\pinctrl-bindings.txt

这会涉及 2 个对象：pin controller、client device。

前者提供服务：可以用它来复用引脚、配置引脚。

后者使用服务：声明自己要使用哪些引脚的哪些功能，怎么配置它们。

a. pin controller:

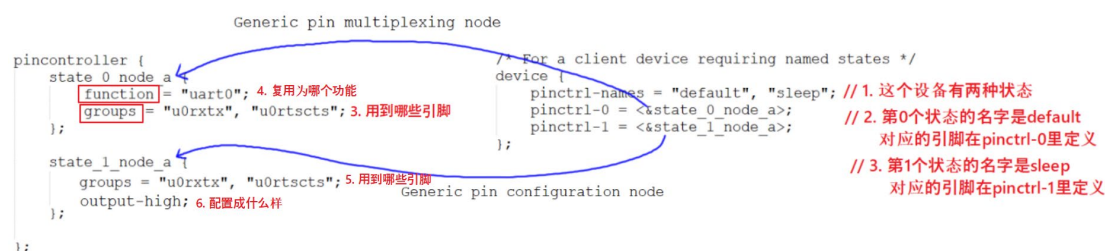
在芯片手册里你找不到 pin controller，它是一个软件上的概念，你可以认为它对应 IOMUX——用来复用引脚，还可以配置引脚(比如上下拉电阻等)。

注意，pin controller 和 GPIO Controller 不是一回事，前者控制的引脚可用于 GPIO 功能、I2C 功能；后者只是把引脚配置为输出、输出等简单的功能。

b. client device

“客户设备”，谁的客户？Pinctrl 系统的客户，那就是使用 Pinctrl 系统的设备，使用引脚的设备。它在设备树里会被定义为一个节点，在节点里声明要用哪些引脚。

下面这个图就可以把几个重要概念理清楚：



上图中，左边是 pincontroller 节点，右边是 client device 节点：

a. pin state:

对于一个“client device”来说，比如对于一个 UART 设备，它有多“状态”：default、sleep 等，那对应的引脚也有这些状态。

怎么理解？

比如默认状态下，UART 设备是工作的，那么所用的引脚就要复用为 UART 功能。

在休眠状态下，为了省电，可以把这些引脚复用为 GPIO 功能；或者直接把它们配置输出高电平。

上图中，pinctrl-names 里定义了 2 种状态：default、sleep。

第 0 种状态用到的引脚在 pinctrl-0 中定义，它是 state_0_node_a，位于 pincontroller 节点中。

第 1 种状态用到的引脚在 pinctrl-1 中定义，它是 state_1_node_a，位于 pincontroller 节点中。

当这个设备处于 default 状态时，pinctrl 子系统会自动根据上述信息把所用引脚复用为 uart0 功能。

当这个设备处于 sleep 状态时，pinctrl 子系统会自动根据上述信息把所用引脚配置为高电平。

b. groups 和 function:

一个设备会用到一个或多个引脚，这些引脚就可以归为一组(group)；

这些引脚可以复用为某个功能：function。

当然：一个设备可以用到多组引脚，比如 A1、A2 两组引脚，A1 组复用为 F1 功能，A2 组复用为 F2 功能。

c. Generic pin multiplexing node 和 Generic pin configuration node

在上图左边的 pin controller 节点中，有子节点或孙节点，它们是给 client device 使用的。

可以用来描述复用信息：哪组(group)引脚复用为哪个功能(function)；

可以用来描述配置信息：哪组(group)引脚配置为哪个设置功能(setting)，比如上拉、下拉等。

注意：pin controller 节点的格式，**没有统一的标准!!!!** 每家芯片都不一样。

甚至上面的 group、function 关键字也不一定有，但是概念是有的。

16.1.3 示例

pin controller	client device
imx6ull <pre>pinctrl_uart1: uart1grp { fsl,pins = < MX6UL_PAD_UART1_TX_DATA__UART1_DCE_TX 0x1b0b1 MX6UL_PAD_UART1_RX_DATA__UART1_DCE_RX 0x1b0b1 >; };</pre>	<pre>&uart1 { pinctrl-names = "default"; pinctrl-0 = <&pinctrl_uart1>; status = "okay"; };</pre>
rk3288 rk3399 <pre>gpio4_uart0 { uart0_xfer: uart0-xfer { rockchip,pins = <UART0BT_SIN>, <UART0BT_SOUT>; rockchip,pull = <VALUE_PULL_DISABLE>; rockchip,drive = <VALUE_DRV_DEFAULT>; //rockchip,tristate = <VALUE_TRI_DEFAULT>; }; uart0_cts: uart0-cts { rockchip,pins = <UART0BT_CTS>; rockchip,pull = <VALUE_PULL_DISABLE>; rockchip,drive = <VALUE_DRV_DEFAULT>; //rockchip,tristate = <VALUE_TRI_DEFAULT>; }; uart0_rts: uart0-rts { rockchip,pins = <UART0BT_RTS>; rockchip,pull = <VALUE_PULL_DISABLE>; rockchip,drive = <VALUE_DRV_DEFAULT>; //rockchip,tristate = <VALUE_TRI_DEFAULT>; }; uart0_rts_gpio: uart0-rts-gpio { rockchip,pins = <FUNC_TO_GPIO(UART0BT_RTSN)>; rockchip,drive = <VALUE_DRV_DEFAULT>; }; };</pre>	<pre>&uart0 { pinctrl-names = "default"; pinctrl-0 = <&uart0_xfer &uart0_cts &uart0_rts>; status = "okay"; };</pre>
am3358 <pre>uart0_pins: pinmux_uart0_pins { pinctrl-single,pins = < 0x170 (PIN_INPUT_PULLUP MUX_MODE0) /* uart0_rxd.uart0_rxd */ 0x174 (PIN_OUTPUT_PULLDOWN MUX_MODE0) /* uart0_txd.uart0_txd */ >; };</pre>	<pre>&uart0 { pinctrl-names = "default"; pinctrl-0 = <&uart0_pins>; status = "okay"; };</pre>

16.1.4 代码中怎么引用 pinctrl

这是透明的，我们的驱动基本不用管。当设备切换状态时，对应的 pinctrl 就会被调用。
比如在 platform_device 和 platform_driver 的枚举过程中，流程如下：

```
really_probe:
/* If using pinctrl, bind pins now before probing */
ret = pinctrl_bind_pins(dev);
dev->pins->default_state = pinctrl_lookup_state(dev->pins->p,
    PINCTRL_STATE_DEFAULT); /* 获得"default"状态的pinctrl */
dev->pins->init_state = pinctrl_lookup_state(dev->pins->p,
    PINCTRL_STATE_INIT); /* 获得"init"状态的pinctrl */

ret = pinctrl_select_state(dev->pins->p, dev->pins->init_state); /* 优先设置"init"状态的引脚 */
ret = pinctrl_select_state(dev->pins->p, dev->pins->default_state); /* 如果没有init状态，则设置"default"状态的引脚 */

.....
ret = drv->probe(dev); // 2. 调用到我们的代码
```

当系统休眠时，也会去设置该设备 sleep 状态对应的引脚，不需要我们自己去调用代码。

非要自己调用，也有函数：

```
devm_pinctrl_get_select_default(struct device *dev); // 使用"default"状态的引脚
pinctrl_get_select(struct device *dev, const char *name); // 根据 name 选择某种状态的引脚
pinctrl_put(struct pinctrl *p); // 不再使用，退出时调用
```

1.1.1.1 16.2 GPIO 子系统重要概念

16.2.1 引入

要操作 GPIO 引脚，先把所用引脚配置为 GPIO 功能，这通过 Pinctrl 子系统来实现。

然后就可以根据设置引脚方向(输入还是输出)、读值——获得电平状态，写值——输出高低电平。

以前我们通过寄存器来操作 GPIO 引脚，即使 LED 驱动程序，对于不同的板子它的代码也完全不同。

当 BSP 工程师实现了 GPIO 子系统后，我们就可以：

a. 在设备树里指定 GPIO 引脚

b. 在驱动代码中：

使用 GPIO 子系统的标准函数获得 GPIO、设置 GPIO 方向、读取/设置 GPIO 值。

这样的驱动代码，将是单板无关的。

16.2.2 在设备树中指定引脚

在几乎所有 ARM 芯片中，GPIO 都分为几组，每组中有若干个引脚。所以在使用 GPIO 子系统之前，就要先确定：它是哪组的？组里的哪一个？

在设备树中，“GPIO 组”就是一个 GPIO Controller，这通常都由芯片厂家设置好。我们要做的是找到它名字，比如“gpio1”，然后指定要用它里面的哪个引脚，比如<gpio1 0>。

有代码更直观，下图是一些芯片的 GPIO 控制器节点，它们一般都是厂家定义好，在 xxx.dtsi 文件中：

```
imx6ul.dtsi
gpio1: gpio@0209c000 {
    compatible = "fsl,imx6ul-gpio", "fsl,imx35-gpio";
    reg = <0x0209c000 0x4000>;
    interrupts = <GIC_SPI 66 IRQ_TYPE_LEVEL_HIGH>,
               <GIC_SPI 67 IRQ_TYPE_LEVEL_HIGH>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    gpio-ranges = <biomuxc 0 23 16>, <biomuxc 10 17 6>,
                 <biomuxc 16 33 16>;
};

gpio2: gpio@020a0000 {
    compatible = "fsl,imx6ul-gpio", "fsl,imx35-gpio";
    reg = <0x020a0000 0x4000>;
    interrupts = <GIC_SPI 68 IRQ_TYPE_LEVEL_HIGH>,
               <GIC_SPI 69 IRQ_TYPE_LEVEL_HIGH>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    gpio-ranges = <biomuxc 0 49 16>, <biomuxc 16 111 6>;
};

rk3288.dtsi
gpio1: gpio1@ff780000 {
    compatible = "rockchip,gpio-bank";
    reg = <0x00000000 0x0 0x100>;
    interrupts = <GIC_SPI 82 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <scru PCLK_GPIO1>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
};

gpio2: gpio2@ff790000 {
    compatible = "rockchip,gpio-bank";
    reg = <0x00000000 0x0 0x100>;
    interrupts = <GIC_SPI 83 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <scru PCLK_GPIO2>;
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
};

am33xx.dtsi
gpio0: gpio@44e07000 {
    compatible = "ti,omap4-gpio";
    ti,hwmods = "gpio1";
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    reg = <0x44e07000 0x1000>;
    interrupts = <96>;
};

gpio1: gpio@44e0c900 {
    compatible = "ti,omap4-gpio";
    ti,hwmods = "gpio2";
    gpio-controller;
    #gpio-cells = <2>;
    interrupt-controller;
    #interrupt-cells = <2>;
    reg = <0x44e0c900 0x1000>;
    interrupts = <98>;
};
```

我们暂时只需要关心里面的这 2 个属性：

```
gpio-controller;
#gpio-cells = <2>;
```

“gpio-controller”表示这个节点是一个 GPIO Controller，它下面有很多引脚。

“#gpio-cells = <2>”表示这个控制器下每一个引脚要用 2 个 32 位的数(cell)来描述。

为什么要用 2 个数？其实使用多个 cell 来描述一个引脚，这是 GPIO Controller 自己决定的。比如可以

用其中一个 cell 来表示那是哪一个引脚，用另一个 cell 来表示它是高电平有效还是低电平有效，甚至还可以用更多的 cell 来表示其他特性。

普遍的用法是，用第 1 个 cell 来表示哪一个引脚，用第 2 个 cell 来表示有效电平：

GPIO_ACTIVE_HIGH : 高电平有效

GPIO_ACTIVE_LOW : 低电平有效

定义 GPIO Controller 是芯片厂家的事，我们怎么引用某个引脚呢？在自己的设备节点中使用属性 "[<name>-]gpios"，示例如下：

100ask_imx6ull-14x14.dts

```
led0: cpu {
    label = "cpu";
    gpios = <&gpio5 3 GPIO_ACTIVE_LOW>;
    default-state = "on";
    linux,default-trigger = "heartbeat";
};

gt9xx@5d {
    compatible = "goodix,gt9xx";
    reg = <0x5d>;
    status = "okay";
    interrupt-parent = <&gpio1>;
    interrupts = <5 IRQ_TYPE_EDGE_FALLING>;
    pinctrl-names = "default";
    pinctrl-0 = <&pinctrl_tsc_reset &pinctrl_touchscreen_int>;

    reset-gpios = <&gpio5 2 GPIO_ACTIVE_LOW>;
    irq-gpios = <&gpio1 5 IRQ_TYPE_EDGE_FALLING>;
};
```

上图中，可以使用 gpios 属性，也可以使用 name-gpios 属性。

16.2.3 在驱动代码中调用 GPIO 子系统

在设备树中指定了 GPIO 引脚，在驱动代码中如何使用？

也就是 GPIO 子系统的接口函数是什么？

GPIO 子系统有两套接口：基于描述符的(descriptor-based)、老的(legacy)。前者的函数都有前缀“gpiod_”，它使用 gpio_desc 结构体来表示一个引脚；后者的函数都有前缀“gpio_”，它使用一个整数来表示一个引脚。

要操作一个引脚，首先要 get 引脚，然后设置方向，读值、写值。

驱动程序中要包含头文件，

```
#include <linux/gpio/consumer.h> // descriptor-based
或
#include <linux/gpio.h> // legacy
```

下表列出常用的函数：

descriptor-based	legacy	说明
------------------	--------	----

获得 GPIO		
gpiod_get	gpio_request	
gpiod_get_index		
gpiod_get_array	gpio_request_array	
devm_gpiod_get		
devm_gpiod_get_index		
devm_gpiod_get_array		
设置方向		
gpiod_direction_input	gpio_direction_input	
gpiod_direction_output	gpio_direction_output	
读值、写值		
gpiod_get_value	gpio_get_value	
gpiod_set_value	gpio_set_value	
释放 GPIO		
gpio_free	gpio_free	
gpiod_put	gpio_free_array	
gpiod_put_array		
devm_gpiod_put		
devm_gpiod_put_array		

有前缀“devm_”的含义是“设备资源管理”(Managed Device Resource)，这是一种自动释放资源的机制。它的思想是“资源是属于设备的，设备不存在时资源就可以自动释放”。

比如在 Linux 开发过程中，先申请了 GPIO，再申请内存；如果内存申请失败，那么在返回之前就需要先释放 GPIO 资源。如果使用 devm 的相关函数，在内存申请失败时可以直接返回：设备的销毁函数会自动地释放已经申请了的 GPIO 资源。

建议使用“devm_”版本的相关函数。

举例，假设备在设备树中有如下节点：

```
foo_device {
    compatible = "acme,foo";
    ...
    led-gpios = <&gpio 15 GPIO_ACTIVE_HIGH>, /* red */
               <&gpio 16 GPIO_ACTIVE_HIGH>, /* green */
               <&gpio 17 GPIO_ACTIVE_HIGH>; /* blue */

    power-gpios = <&gpio 1 GPIO_ACTIVE_LOW>;
};
```

那么可以使用下面的函数获得引脚：

```
struct gpio_desc *red, *green, *blue, *power;

red = gpiod_get_index(dev, "led", 0, GPIOD_OUT_HIGH);
green = gpiod_get_index(dev, "led", 1, GPIOD_OUT_HIGH);
blue = gpiod_get_index(dev, "led", 2, GPIOD_OUT_HIGH);
power = gpiod_get(dev, "power", GPIOD_OUT_HIGH);
```


要注意的是, `gpiod_set_value` 设置的值是“逻辑值”, 不一定等于物理值。
什么意思?

Function (example)	active-low property	physical line
<code>gpiod_set_raw_value(desc, 0);</code>	don't care	low
<code>gpiod_set_raw_value(desc, 1);</code>	don't care	high
<code>gpiod_set_value(desc, 0);</code>	default (active-high)	low
<code>gpiod_set_value(desc, 1);</code>	default (active-high)	high
<code>gpiod_set_value(desc, 0);</code>	active-low	high
<code>gpiod_set_value(desc, 1);</code>	active-low	low

如果设备树里引脚指定为GPIO_ACTIVE_LOW
那么gpiod_set_value的逻辑值跟引脚的物理值相反

旧的“gpio_”函数没办法根据设备树信息获得引脚, 它需要先知道引脚号。

引脚号怎么确定?

在 GPIO 子系统中, 每注册一个 GPIO Controller 时会确定它的“base number”, 那么这个控制器里的第 n 号引脚的号码就是: base number + n。

但是如果硬件有变化、设备树有变化, 这个 base number 并不能保证是固定的, 应该查看 sysfs 来确定 base number。

16.2.4 sysfs 中的访问方法

在 sysfs 中访问 GPIO, 实际上用的就是引脚号, 老的方法。

a. 先确定某个 GPIO Controller 的基准引脚号(base number), 再计算出某个引脚的号码。

方法如下:

① 先在开发板的/sys/class/gpio 目录下, 找到各个 gpiochipXXX 目录:

```
[root@imx6ull:/sys/class/gpio]# ls
export      gpio30      gpiochip128  gpiochip504  gpiochip96
gpio133     gpiochip0   gpiochip32   gpiochip64   unexport
[root@imx6ull:/sys/class/gpio]#
```

② 然后进入某个 gpiochip 目录, 查看文件 label 的内容

③ 根据 label 的内容对比设备树

label 内容来自设备树, 比如它的寄存器基地址。用来跟设备树(dtsi 文件)比较, 就可以知道这对应哪一个 GPIO Controller。

下图是在 100asK_imx6ull 上运行的结果, 通过对比设备树可知 gpiochip96 对应 gpio4:

```
[root@imx6ull:/sys/class/gpio]# cd gpiochip96/
[root@imx6ull:/sys/class/gpio/gpiochip96]# ls
base      device      label      ngpio      power      subsystem  uevent
[root@imx6ull:/sys/class/gpio/gpiochip96]# cat label
20a8000.gpio
```

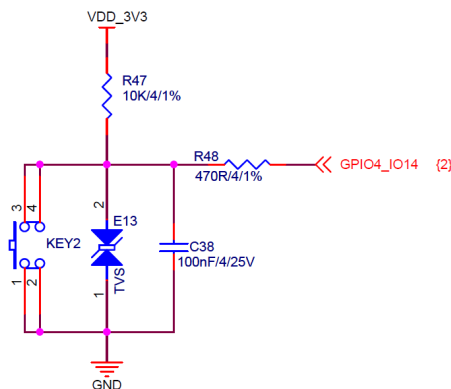
imx6ull.dtsi

```
gpio4: gpio@020a8000 {
    compatible = "fsl,imx6ul-gpio", "fsl,imx35-gpio";
    reg = <0x020a8000 0x4000>;
```

所以 gpio4 这组引脚的基准引脚号就是 96, 这也可以“cat base”来再次确认。

b. 基于 sysfs 操作引脚:

以 100ask_imx6ull 为例, 它有一个按键, 原理图如下:



那么 GPIO4_14 的号码是 96+14=110，可以如下操作读取按键值：

```
echo 110 > /sys/class/gpio/export
echo in > /sys/class/gpio/gpio110/direction
cat /sys/class/gpio/gpio110/value
echo 110 > /sys/class/gpio/unexport
```

注意：如果驱动程序已经使用了该引脚，那么将会 export 失败，会提示下面的错误：

```
-sh: echo: write error: Device or resource busy
```

对于输出引脚，假设引脚号为 N，可以用下面的方法设置它的值为 1：

```
echo N > /sys/class/gpio/export
echo out > /sys/class/gpio/gpioN/direction
echo 1 > /sys/class/gpio/gpioN/value
echo N > /sys/class/gpio/unexport
```