第15章 GPIO 和 Pinctrl 子系统的使用

参考文档：

- 内核 Documentation\devicetree\bindings\Pinctrl\ 目录下：

```
Pinctrl-bindings.txt
```

- 内核 Documentation\gpio 目录下：

```
Pinctrl-bindings.txt
```

- 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。

15.1 Pinctrl 子系统重要概念

15.1.1 引入

无论是哪种芯片，都有类似图 15.1 的结构：

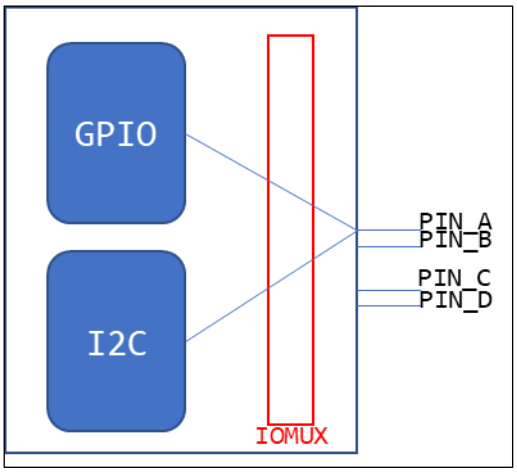


图 15.1 GPIO 与外设连接示意图

- 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？看图 15.2：

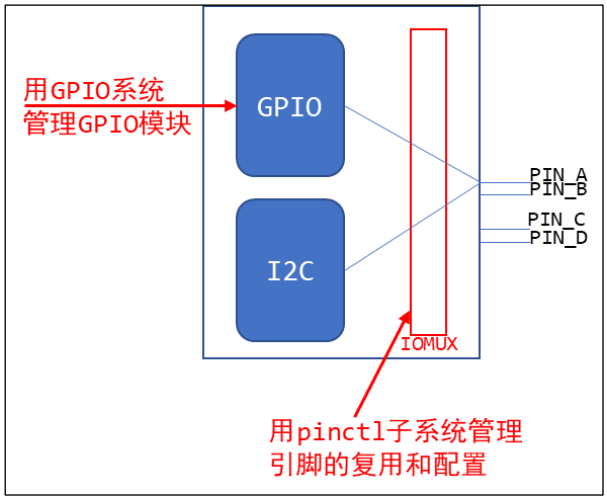


图 15.2 工程师的任务

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。

15.1.2 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。

① 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入、输出等简单的功

能。即先用 pin controller 把引脚配置为 GPIO，再用 GPIO Controler 把引脚配置为输入或输出。

② 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。

图 15.3 就可以把几个重要概念理清楚：

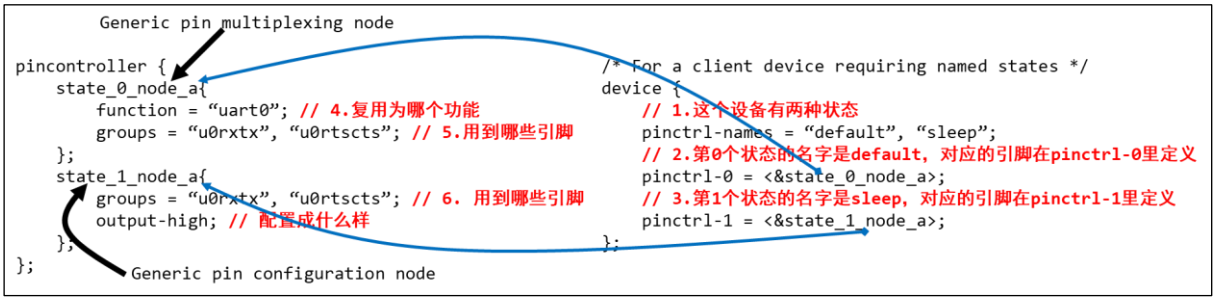


图 15.3 复用节点与配置节点

上图中，左边是 pin controller 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，它有多个“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

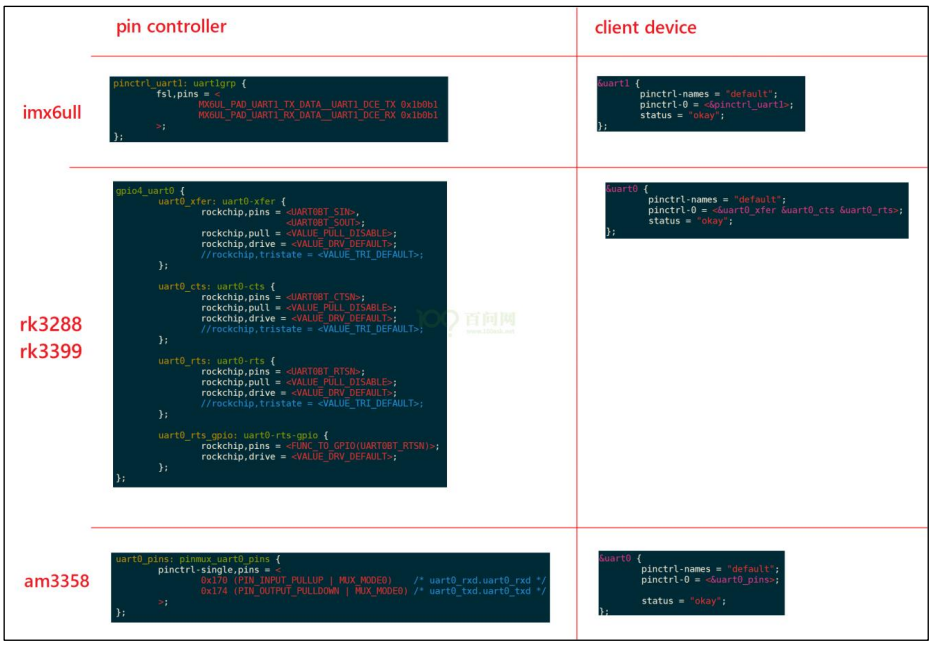
在上图左边的 pin controller 节点中，有子节点或孙节点，它们是给 client device 使用的。

- 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。

15.1.3 示例



15.1.4 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);                    // 1. 引脚被设置为某个状态：根本不用我们自己去调用代码
            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
```