第二十五章 pinctrl 和 gpio 子系统实验

上一章我们编写了基于设备树的 LED 驱动，但是驱动的本质还是没变，都是配置 LED 灯所使用的 GPIO 寄存器，驱动开发方式和裸机基本没啥区别。Linux 是一个庞大而完善的系统，尤其是驱动框架，像 GPIO 这种最基本的驱动不可能采用“原始”的裸机驱动开发方式，否则就相当于你买了一辆车，结果每天推着车去上班。Linux 内核提供了 pinctrl 和 gpio 子系统用于 GPIO 驱动，本章我们就来学习一下如何借助 pinctrl 和 gpio 子系统来简化 GPIO 驱动开发。

25.1 pinctrl 子系统

25.1.1 pinctrl 子系统简介

Linux 驱动讲究驱动分离与分层，pinctrl 和 gpio 子系统就是驱动分离与分层思想下的产物，驱动分离与分层其实就是按照面向对象编程的设计思想而设计的设备驱动框架，关于驱动的分离与分层我们后面会讲。本来 pinctrl 和 gpio 子系统应该放到驱动分离与分层章节后面讲解，但是不管什么外设驱动，GPIO 驱动基本都是必须的，而 pinctrl 和 gpio 子系统又是 GPIO 驱动必须使用的，所以就将 pintrcl 和 gpio 子系统这一章节提前了。

我们先来回顾一下上一章是怎么初始化 LED 灯所使用的 GPIO，步骤如下：

①、修改设备树，添加相应的节点，节点里面重点是设置 reg 属性，reg 属性包括了 GPIO 相关寄存器。

②、 获取 reg 属性中 GPIOI_MODER、GPIOI_OTYPER、GPIOI_OSPEEDR、GPIOI_PUPDR 和 GPIOI_BSRR 这些寄存器的地址，并且初始化它们，这些寄存器用于设置 PI0 这个 PIN 的复用功能、上下拉、速度等。

③、 在②里面将 PI0 这个 PIN 设置为通用输出功能，因此需要设置 PI0 这个 GPIO 相关的寄存器，也就是设置 GPIOI_MODER 寄存器。

④、在②里面将 PI0 这个 PIN 设置为高速、上拉和推挽模式，就要需要设置 PI0 的 GPIOI_OTYPER、GPIOI_OSPEEDR 和 GPIOI_PUPDR 这些寄存器。

总结一下，②中完成对 PI0 这个 PIN 的获取相关的寄存器地址，③和④设置这个 PIN 的复用功能、上下拉等，比如将 GPIOI_0 这个 PIN 设置为通用输出功能。如果使用过 STM32 单片机的话应该都记得，STM32 单片机也是要先设置某个 PIN 的复用功能、速度、上下拉等，然后再设置 PIN 所对应的 GPIO，STM32MP1 是从 STM32 单片机发展而来的，设置 GPIO 是一样。其实对于大多数的 32 位 SOC 而言，引脚的设置基本都是这两方面，因此 Linux 内核针对 PIN 的复用配置推出了 pinctrl 子系统，对于 GPIO 的电气属性配置推出了 gpio 子系统。本小节我们来学习 pinctrl 子系统，下一小节再学习 gpio 子系统。

大多数 SOC 的 pin 都是支持复用的，比如 STM32MP1 的 PI0 既可以作为普通的 GPIO 使用，也可以作为 SPI2 的 NSS 引脚、TIM5 的 CH4 引脚等等。此外我们还需要配置 pin 的电气特性，比如上/下拉、速度、驱动能力等等。传统的配置 pin 的方式就是直接操作相应的寄存器，但是这种配置方式比较繁琐、而且容易出问题(比如 pin 功能冲突)。pinctrl 子系统就是为了解决这个问题而引入的，pinctrl 子系统主要工作内容如下：

①、获取设备树中 pin 信息。

②、根据获取到的 pin 信息来设置 pin 的复用功能

③、根据获取到的 pin 信息来设置 pin 的电气特性，比如上/下拉、速度、驱动能力等。

对于我们使用者来讲，只需要在设备树里面设置好某个 pin 的相关属性即可，其他的初始化工作均由 pinctrl 子系统来完成，pinctrl 子系统源码目录为 drivers/pinctrl。

25.1.2 STM32MP1 的 pinctrl 子系统驱动

1、PIN 配置信息详解

要使用 pinctrl 子系统，我们需要在设备树里面设置 PIN 的配置信息，毕竟 pinctrl 子系统要根据你提供的信息来配置 PIN 功能，一般会在设备树里面创建一个节点来描述 PIN 的配置信息。打开 stm32mp151.dtsi 文件，找到一个叫做 pinctrl 的节点，如下所示：

示例代码 25.1.2.1 pinctrl 节点内容 1

```
1814        pinctrl: pin-controller@50002000 {
1815            #address-cells = <1>;
1816            #size-cells = <1>;
1817            compatible = "st,stm32mp157-pinctrl";
1818            ranges = <0 0x50002000 0xa400>;
1819            interrupt-parent = <&exti>;
1820            st,syscfg = <&exti 0x60 0xff>;
1821            hwlocks = <&hsem 0 1>;
1822            pins-are-numbered;
......
1968        };
```

第 1815~1816 行，#address-cells 属性值为 1 和#size-cells 属性值为 1，也就是说 pinctrl 下的所有子节点的 reg 第一位是起始地址，第二位为长度。

第 1818 行，ranges 属性表示 STM32MP1 的 GPIO 相关寄存器起始地址，STM32MP1 系列芯片最多拥有 176 个通用 GPIO，分为 12 组，分别为：PA0~PA15、PB0~PB15、PC0~PC15、PD0~PD15、PE0~PE15、PF0~PF15、PG0~PG15、PH0~PH15、PJ0~PJ15、PK0~PK7、PZ0~PZ7。

其中 PA~PK 这 9 组 GPIO 的寄存器都在一起，起始地址为 0X50002000，终止地址为 0X5000C3FF。这个可以在《STM32MP157 参考手册》里面找到。PZ 组寄存器起始地址为 0X54004000，终止地址为 0X540043FF，所以 stm32mp151.dtsi 文件里面还有个名为“pinctrl_z”的子节点来描述 PZ 组 IO。pinctrl 节点用来描述 PA~PK 这 11 组 IO，因此 ranges 属性中的 0x50002000 表示起始地址，0xa400 表示寄存器地址范围。

第 1819 行，interrupt-parent 属性值为“&exti”，父中断为 exti。

后面的 gpiox 子节点先不分析，这些子节点都是 gpio 子系统的内容，到后面在去分析。这个节点看起来，根本没有 PIN 先关的配置，别急！先打开 stm32mp15-pinctrl.dtsi 文件，你们能找到如下内容：

示例代码 25.1.2.2 pinctrl 节点内容 2

```
1   &pinctrl {
......
534     m_can1_pins_a: m-can1-0 {
535         pins1 {
536             pinmux = <STM32_PINMUX('H', 13, AF9)>;     /* CAN1_TX */
537             slew-rate = <1>;
538             drive-push-pull;
539             bias-disable;
540         };
541         pins2 {
542             pinmux = <STM32_PINMUX('I', 9, AF9)>;  /* CAN1_RX */
543             bias-disable;
544         };
545     };
......
554     pwm1_pins_a: pwm1-0 {
```

```
555         pins {
556             pinmux = <STM32_PINMUX('E', 9, AF1)>, /* TIM1_CH1 */
557                 <STM32_PINMUX('E', 11, AF1)>,     /* TIM1_CH2 */
558                 <STM32_PINMUX('E', 14, AF1)>;     /* TIM1_CH4 */
559             bias-pull-down;
560             drive-push-pull;
561             slew-rate = <0>;
562         };
563     };
......
1411};
```

示例代码 25.1.2.2 就是向 pinctrl 节点追加数据，不同的外设使用的 PIN 不同、其配置也不同，因此一个萝卜一个坑，将某个外设所使用的所有 PIN 都组织在一个子节点里面。示例代码 25.1.2.2 中 m_can1_pins_a 子节点就是 CAN1 的所有相关 IO 的 PIN 集合，pwm1_pins_a 子节点就是 PWM1 相关 IO 的 PIN 集合。绑定文档 Documentation/devicetree/bindings/pinctrl/st,stm32-pinctrl.yaml 描述了如何在设备树中设置 STM32 的 PIN 信息：

每个 pincrtl 节点必须至少包含一个子节点来存放 pincrtl 相关信息，也就是 pinctrl 集，这个集合里面存放当前外设用到哪些引脚(PIN)、这些引脚应该怎么配置、复用相关的配置、上下拉、默认输出高电平还是低电平。一般这个存放 pincrtl 集的子节点名字是“pins”，如果某个外设用到多种配置不同的引脚那么就需要多个 pins 子节点，比如示例代码 25.1.2.2 中第 535 行和 541 行的 pins1 和 pins2 这两个子节点分别描述 PH13 和 PI9 的配置方法，由于 PH13 和 PI9 这两个 IO 的配置不同，因此需要两个 pins 子节点来分别描述。第 555~562 行的 pins 子节点是描述 PWM1 的相关引脚，包括 PE9、PE11、PE14，由于这三个引脚的配置是一模一样的，因此只需要一个 pins 子节点就可以了。

上面讲了，在 pins 子节点里面存放外设的引脚描述信息，这些信息包括：

1、pinmux 属性

此属性用来存放外设所要使用的所有 IO，示例代码 25.1.2.2 中第 536 行的 pinmux 属性内容如下：

```
pinmux = <STM32_PINMUX('H', 13, AF9)>;
```

可以看出，这里使用 STM32_PINMUX 这宏来配置引脚和引脚的复用功能，此宏定义在 include/dt-bindings/pinctrl/stm32-pinfunc.h 文件里面，内容如下：

示例代码 25.1.2.3 STM32_PINMUX 宏定义

```
32 #define PIN_NO(port, line)  (((port) - 'A') * 0x10 + (line))
33
34 #define STM32_PINMUX(port, line, mode) (((PIN_NO(port, line)) << 8)
                                           | (mode))
```

可以看出，STM32_PINMUX 宏有三个参数，这三个参数含义如下所示：

port：表示用那一组 GPIO(例：H 表示为 GPIO 第 H 组，也就是 GPIOH)。

line：表示这组 GPIO 的第几个引脚(例：13 表示为 GPIOH_13，也就是 PH13)。

mode：表示当前引脚要做那种复用功能(例：AF9 表示为用第 9 个复用功能)，这个需要查阅 STM32MP1 数据手册来确定使用哪个复用功能。打开《STM32MP157A&D 数据手册》，一个 IO 最大有 16 种复用方法：AF0~AF15，打开第 4 章“Pinouts, pin description and alternate functions”，其中“Table 8. Alternate function AF0 to AF7”描述的是 AF0~AF7 这 8 种复用方

式，“Table 9. Alternate function AF8 to AF15”描述的是 AF8~AF15 这 8 中复用方式。找到图 25.1.2.1 所示内容：

Table 9. Alternate function AF8 to AF15(1) (continued)

Port		AF8	AF9	AF10	AF11	AF12	AF13	AF14	AF15
		SPI6/SAI2/ USART3/ UART4/5/8/ SDMMC1/2/ SPDIFRX	FDCAN1/2/ TIM13/14/ QUADSPI/ SDMMC2/3/ LCD/SPDIFRX	SAI2/4/ QUADSPI/ FMC/ SDMMC2/3/ OTG_FS/ OTG_HS	DFSDM1/ QUADSPI/ SDMMC1/ MDIOS/ETH1/ DSI	SAI4/UART5/ FMC/SDMMC1/ MDIOS	UART7/DCMI/ LCD/DSI/RNG	UART5/LCD	SYS
Port H	PH9	-	-	-	-	-	DCMI_D0	LCD_R3	EVENTOUT
	PH10	-	-	-	-	-	DCMI_D1	LCD_R4	EVENTOUT
	PH11	-	-	-	-	-	DCMI_D2	LCD_R5	EVENTOUT
	PH12	-	-	-	-	-	DCMI_D3	LCD_R6	EVENTOUT
	PH13	UART4_TX	FDCAN1_TX	-	-	-	-	LCD_G2	EVENTOUT
	PH14	UART4_RX	FDCAN1_RX	-	-	-	DCMI_D4	LCD_G3	EVENTOUT

图 25.1.2.1 PH13 引脚复用功能

可以看出，PH13 要设置为 FDCAN1 的 TX 引脚，就要使用 AF9 这个复用配置，这个就是如何在 pinmux 属性中添加一个引脚的复用配置。从图 25.1.2.1 中可以看出，如果 PH13 也可以复用为 UART4 的 TX 引脚，只要设置为 AF8 即可，此时相关的引脚配置如下所示：

示例代码 25.1.2.3 myuart4 pinctrl 节点

```
1  &pinctrl {
2
3      myuart4_pins_a: myuart4-0 {
4          pins{
5              pinmux = <STM32_PINMUX('H', 13, AF8)>;
6          };
7      };
8  };
```

这时候大家可能会奇怪？PH13 做了 FDCAN1 的 TX 功能，还能做 UART4 的 TX 功能，不是冲突了吗？这是不会冲突的，因为 pinctrl 驱动只会把设备树 pinctrl 节点解析出来的数据存储到一个链表里，只有当外设调用这个 pinctrl 节点的时候才会真的使用。但是，如果你要同时使用 FDCAN1 和 UART4 的话就会出问题，因此 PH13 同一时间只能用于一个外设，所以为了方便开发，还是建议大家一个 PIN 最好只能被一个外设使用。

stm32-pinfunc.h 文件里面定义了一个 PIN 的所有配置项 AF0~AF9，如下所示：

示例代码 25.1.2.4 stm32-pinfunc.h 文件

```
10 /*  define PIN modes */
11 #define GPIO      0x0
12 #define AF0       0x1
13 #define AF1       0x2
14 #define AF2       0x3
15 #define AF3       0x4
16 #define AF4       0x5
17 #define AF5       0x6
18 #define AF6       0x7
19 #define AF7       0x8
20 #define AF8       0x9
21 #define AF9       0xa
```

```
22  #define AF10      0xb
23  #define AF11      0xc
24  #define AF12      0xd
25  #define AF13      0xe
26  #define AF14      0xf
27  #define AF15      0x10
28  #define ANALOG    0x11
29  #define RSVD      0x12
```

可以看出，除了 A0~AF15，还有 GPIO、ANALOG 这两个，如果一个 PIN 只是作为最基本的 GPIO 功能，那么就是用“GPIO”；如果这个引脚要用作模拟功能，比如 ADC 采集引脚，那么就设置为“ANALOG”。

2、电气属性配置

接下来了解一下 PIN 的电气特性如何设置，电气特性在 pinctrl 子系统里不是必须的，可以不配置，但是 pinmux 属性是必须要设置的。stm32-pinctrl.yaml 文件里面也描述了如何设置 STM32 的电气属性，如表 25.1.2.1 所示：

电气特性属性	类型	作用
bias-disable	bootlean	禁止使用内部偏置电压
bias-pull-down	bootlean	内部下拉
bias-pull-up	bootlean	内部上拉
drive-push-pull	bootlean	推挽输出
drive-open-drain	bootlean	开漏输出
output-low	bootlean	输出低电平
output-high	bootlean	输出高电平
slew-rate	enum	引脚的速度，可设置：0~3， 0 最慢，3 最高。

表 25.1.2.1 pinctrl 的电气特性

表 25.1.2.1 里的 bootlean 类型表示了在 pinctrl 子系统只要定义这个电气属性就行了，例如：我要禁用内部电压，只要在 PIN 的配置集里添加“bias-disable”即可，这个时候 bias-pull-down 和 bias-pull-up 都不能使用了，因为已经禁用了内部电压，所以不能配置上下拉。enum 类型使用方法更简单跟 C 语言的一样，比如要设置 PIN 速度为最低就可以使用“slew-rate=<0>”。在示例代码 25.1.2.3 里添加引脚的电气特性组合成，如示例代码 25.1.2.6 所示：

示例代码 25.1.2.6 添加电气特性的 myuart4-0 节点

```
1   &pinctrl {
2
3       myuart4_pins_a: myuart4-0 {
4           pins1{
5               pinmux = <STM32_PINMUX('H', 13, AF8)>;
6               bias-pull-up;
7               drive-push-pull;
8             };
9       };
10  };
```

在第 6~7 行里给 myuart4-0 添加了两个电气属性分别为内部上拉和推挽输出，这样我们就设置好一个 PIN 配置了。

2、PIN 驱动程序讲解

本小节会涉及到 Linux 驱动分层与分离、平台设备驱动等还未讲解的知识，所以本小节教程可以不用看，不会影响后续的实验。如果对 Linux 内核的 pinctrl 子系统实现原理感兴趣的话可以看本小节。

所有的东西都已经准备好了，包括寄存器地址和寄存器值，Linux 内核相应的驱动文件就会根据这些值来做相应的初始化。接下来就找一下哪个驱动文件来做这一件事情，pinctrl 节点中 compatible 属性的值为“st,stm32mp157-pinctrl”，在 Linux 内核中全局搜索“pinctrl”字符串就会找到对应的驱动文件。在文件 drivers/pinctrl/stm32/pinctrl-stm32mp157.c 中有如下内容：

示例代码 25.1.2.7 pinctrl-stm32mp157.c 文件代码段

```
2323  static struct stm32_pinctrl_match_data stm32mp157_match_data =
{
2324      .pins = stm32mp157_pins,
2325      .npins = ARRAY_SIZE(stm32mp157_pins),
2326  };
2327
2328  static struct stm32_pinctrl_match_data stm32mp157_z_match_data
= {
2329      .pins = stm32mp157_z_pins,
2330      .npins = ARRAY_SIZE(stm32mp157_z_pins),
2331      .pin_base_shift = STM32MP157_Z_BASE_SHIFT,
2332  };
2333
2334  static const struct of_device_id stm32mp157_pctrl_match[] = {
2335      {
2336          .compatible = "st,stm32mp157-pinctrl",
2337          .data = &stm32mp157_match_data,
2338      },
2339      {
2340          .compatible = "st,stm32mp157-z-pinctrl",
2341          .data = &stm32mp157_z_match_data,
2342      },
2343      { }
2344  };
2345
2346  static const struct dev_pm_ops stm32_pinctrl_dev_pm_ops = {
2347       SET_LATE_SYSTEM_SLEEP_PM_OPS(NULL, stm32_pinctrl_resume)
2348  };
2349
2350  static struct platform_driver stm32mp157_pinctrl_driver = {
2351      .probe = stm32_pctl_probe,
2352      .driver = {
```

```
2353            .name = "stm32mp157-pinctrl",
2354            .of_match_table = stm32mp157_pctrl_match,
2355            .pm = &stm32_pinctrl_dev_pm_ops,
2356        },
2357    };
2358
2359    static int __init stm32mp157_pinctrl_init(void)
2360    {
2361        return
platform_driver_register(&stm32mp157_pinctrl_driver);
2362    }
2363    arch_initcall(stm32mp157_pinctrl_init);
2364
```

第 2334~2344 行， of_device_id 结构体类型的数组，在第二十三章讲解设备树的时候说过，of_device_id 里面保存着这个驱动文件的兼容性值，设备树中的 compatible 属性值会和 of_device_id 中的所有兼容性字符串比较，查看是否可以使用此驱动。stm32mp157_pctrl_match 结构体数组一共有两个兼容，分别为“st,stm32mp157-pinctrl”和“st,stm32mp157-z-pinctrl”，设备树也定义了这两个兼容性值，因此 pinctrl 和 pinctrl_z 节点都会和此驱动匹配，所以 pinctrl-stm32mp157.c 会完成 STM32MP1 的 PIN 配置工作。

第 2350~2357 行，platform_driver 是平台设备驱动，这个是我们后面章节要讲解的内容，platform_driver 是个结构体，有个 probe 成员变量。在这里大家只需要知道，当设备和驱动匹配成功以后 platform_driver 的 probe 成员变量所代表的函数就会执行，在 2351 行设置 probe 成员变量为 stm32_pctl_probe 函数，因此在本章实验中 stm32_pctl_probe 这个函数就会执行，可以认为 stm32_pctl_probe 函数就是 STM32MP157 这个 SOC 的 PIN 配置入口函数。

第 2359~2362 行，就是一个简单的驱动入口函数，platform_driver_register 函数是一个标准的平台设备驱动注册函数，用于向 Linux 内核注册一个 platform_driver，这里就是将 stm32mp157_pinctrl_driver 注册到 Linux 内核总，关于平台设备驱动后面章节会详细讲解。

我们重点来分析一下 stm32_pctl_probe 函数，函数定义在 drivers/pinctrl/stm32/pinctrl-stm32.c 里面，函数内容如下所示：

示例代码 25.1.2.8 stm32_pctl_probe 代码段

```
1452    int stm32_pctl_probe(struct platform_device *pdev)
1453    {
......
1458        struct stm32_pinctrl *pctl;
......
1530
1531        pctl->pctl_desc.name = dev_name(&pdev->dev);
1532        pctl->pctl_desc.owner = THIS_MODULE;
1533        pctl->pctl_desc.pins = pins;
1534        pctl->pctl_desc.npins = pctl->npins;
1535        pctl->pctl_desc.link_consumers = true;
1536        pctl->pctl_desc.confops = &stm32_pconf_ops;
1537        pctl->pctl_desc.pctlops = &stm32_pctrl_ops;
1538        pctl->pctl_desc.pmxops = &stm32_pmx_ops;
```

```
1539        pctl->dev = &pdev->dev;
1540        pctl->pin_base_shift = pctl->match_data->pin_base_shift;
1541
1542        pctl->pctl_dev = devm_pinctrl_register(&pdev->dev,
1543                        &pctl->pctl_desc, pctl);
......
1600};
```

第 1458 行，看它的结构体的名字就知道是 ST 官方自定义的一个结构体类型，用于存放 STM32 相关 PIN 属性集合。第 22.5.1 小节我们驱动代码也添加了自己的结构体，它们都有自己的结构体这样做有啥好处？可以实现一个驱动代码“通杀”多个设备。要想驱动能通用，就要用结构体来保存数据和驱动里面尽量不要使用全局变量(在 pinctrl 驱动里就没有一个全局变量，全部使用结构体来描述一个物体，物体的所有属性都作为结构体成员变量)。接着我们去看下 stm32_pinctrl 结构体是如何定义的，如示例代码 25.1.2.9 所示：

示例代码 25.1.2.9 stm32_pinctrl 结构体代码段

```
100 struct stm32_pinctrl {
......
103     struct pinctrl_desc pctl_desc;
......
107     struct stm32_gpio_bank *banks;
......
120 };
```

第 103 行，pinctrl_desc 结构体用来描述 PIN 控制器，PIN 控制器，PIN 控制器用于配置 SOC 的 PIN 复用功能和电气特性。

第 107 行，这个 stm32_gpio_bank 结构体，是用来注册 GPIO 驱动。到后面 GPIO 子系统在说。

pinctrl_desc 结构体内容如下所示：

示例代码 25.1.2.10 pinctrl_desc 结构体

```
130 struct pinctrl_desc {
131     const char *name;
132     const struct pinctrl_pin_desc *pins;
133     unsigned int npins;
134     const struct pinctrl_ops *pctlops;
135     const struct pinmux_ops *pmxops;
136     const struct pinconf_ops *confops;
137     struct module *owner;
138 #ifdef CONFIG_GENERIC_PINCONF
139     unsigned int num_custom_params;
140     const struct pinconf_generic_params *custom_params;
141     const struct pin_config_item *custom_conf_items;
142 #endif
143     bool link_consumers;
144 };
```

第 134~136 行，这三个“_ops”结构体指针非常重要！因为这三个结构体就是 PIN 控制器的“工具”，这三个结构体里面包含了很多操作函数，Linux 内核初始化 PIN 最终使用的

就是这些操作函数。因此编写一个 SOC 的 PIN 控制器驱动的核心就是实现 pinctrl_desc 里面的 pctlops、pmxops 和 confops，pinctrl_desc 结构体需要由用户提供，结构体里面的成员变量也是用户编写的。但是这个用户并不是我们这些使用芯片的程序员，而是半导体厂商，半导体厂商发布的 Linux 内核源码中已经把这些工作做完了。

示例代码 25.1.2.8 里，第 1536~1538 行，给这三个结构体赋值分别对应 stm32_pconf_ops、stm32_prtrl_ops 和 stm32_pmx_ops，三个结构体如下：

示例代码 25.1.2.11 stm32_pconf_ops、stm32_prtrl_ops 和 stm32_pmx_ops 结构体

```
714  static const struct pinctrl_ops stm32_pctrl_ops = {
715      .dt_node_to_map    = stm32_pctrl_dt_node_to_map,
716      .dt_free_map       = pinctrl_utils_free_map,
717      .get_groups_count  = stm32_pctrl_get_groups_count,
718      .get_group_name    = stm32_pctrl_get_group_name,
719      .get_group_pins    = stm32_pctrl_get_group_pins,
720  };
......
865  static const struct pinmux_ops stm32_pmx_ops = {
866      .get_functions_count = stm32_pmx_get_funcs_cnt,
867      .get_function_name   = stm32_pmx_get_func_name,
868      .get_function_groups = stm32_pmx_get_func_groups,
869      .set_mux             = stm32_pmx_set_mux,
870      .gpio_set_direction  = stm32_pmx_gpio_set_direction,
871      .strict              = true,
872  };
......
1238 static const struct pinconf_ops stm32_pconf_ops = {
1239     .pin_config_group_get   = stm32_pconf_group_get,
1240     .pin_config_group_set   = stm32_pconf_group_set,
1241     .pin_config_dbg_show    = stm32_pconf_dbg_show,
1242     .pin_config_set         = stm32_pconf_set,
1243 };
```

pinctrl_desc 结构体初始化完成以后，需要调用 pinctrl_register 或者 devm_pinctrl_register 函数就能够向 Linux 内核注册一个 PIN 控制器，示例代码 25.1.2.8 中的 1542 行就是向 Linux 内核注册 PIN 控制器。

总结一下，pinctrl 驱动流程如下：

1、定义 pinctrl_desc 结构体。

2、初始化结构体，重点是 pinconf_ops、pinmux_ops 和 pinctrl_ops 这三个结构体成员变量，但是这部分半导体厂商帮我们搞定。

3、调用 devm_pinctrl_register 函数完成 PIN 控制器注册。

25.1.3 设备树中添加 pinctrl 节点模板

我们已经对 pinctrl 有了比较深入的了解，接下来我们学习一下如何在设备树中添加某个外设的 PIN 信息。比如我们需要将 PG11 这个 PIN 复用为 UART4_TX 引脚，pinctrl 节点添加过程如下：

1、创建对应的节点

在 pinctrl 节点下添加一个“uart4_pins”节点：

示例代码 25.1.3.1 uart4_pins 设备节点

```
&pinctrl {
    uart4_pins: uart4-0 {
    /* 具体的 PIN 信息 */
    };
};
```

2、添加“pins”属性

添加一个“pins”子节点，这个子节点是真正用来描述 PIN 配置信，要注意，同一个 pins 子节点下的所有 PIN 电气属性要一样。如果某个外设所用的 PIN 可能有不同的配置，那么就需要多个 pins 子节点，例如 UART4 的 TX 和 RX 引脚配置不同，因此就有 pins1 和 pins2 两个子节点。这里我们只添加 UART4 的 TX 引脚，所以添加完 pins1 子节点以后如下所示：

示例代码 25.1.3.2 uart4_pins 设备节点下的 pins1 属性

```
&pinctrl {
    uart4_pins: uart4-0 {
        pins1{
            /* UART4 TX 引脚的 PIN 配置信息 */
        };
    };
};
```

3、在“pins”节点中添加 PIN 配置信息

最后在“pins”节点中添加具体的 PIN 配置信息，完成以后如下所示：

示例代码 25.1.3.3 完整的 uart4_pins 设备 pinctrl 子节点

```
&pinctrl {
    gpio_led: gpio-led-0 {
        pins1{
            pinmux = <STM32_PINMUX('G', 11, AF6)>; /* UART4_TX */
            bias-disable;
            drive-push-pull;
        };
    };
};
```

按道理来讲，当我们将一个 IO 用作 GPIO 功能的时候也需要创建对应的 pinctrl 节点，并且将所用的 IO 复用为 GPIO 功能，比如将 PI0 复用为 GPIO 的时候就需要设置 pinmux 属性为：<STM32_PINMUX('I', 0, GPIO)>，但是！对于 STM32MP1 而言，如果一个 IO 用作 GPIO 功能的时候不需要创建对应的 pinctrl 节点！

25.2 gpio 子系统

25.2.1 gpio 子系统简介

上一小节讲解了 pinctrl 子系统，pinctrl 子系统重点是设置 PIN(有的 SOC 叫做 PAD)的复用和电气属性，如果 pinctrl 子系统将一个 PIN 复用为 GPIO 的话，那么接下来就要用到 gpio 子系统了。gpio 子系统顾名思义，就是用于初始化 GPIO 并且提供相应的 API 函数，比

如设置 GPIO 为输入输出，读取 GPIO 的值等。gpio 子系统的主要目的就是方便驱动开发者使用 gpio，驱动开发者在设备树中添加 gpio 相关信息，然后就可以在驱动程序中使用 gpio 子系统提供的 API 函数来操作 GPIO，Linux 内核向驱动开发者屏蔽掉了 GPIO 的设置过程，极大的方便了驱动开发者使用 GPIO。

25.2.2 STM32MP1 的 gpio 子系统驱动

1、设备树中的 gpio 信息

首先肯定是 GPIO 控制器的节点信息，以 PI0 这个引脚所在的 GPIOI 为例，打开 stm32mp151.dtsi，在里面找到如下所示内容：

示例代码 25.2.2.1 gpioi 控制器节点

```
1814    pinctrl: pin-controller@50002000 {
1815        #address-cells = <1>;
1816        #size-cells = <1>;
1817        compatible = "st,stm32mp157-pinctrl";
......
1912        gpioi: gpio@5000a000 {
1913            gpio-controller;
1914            #gpio-cells = <2>;
1915            interrupt-controller;
1916            #interrupt-cells = <2>;
1917            reg = <0x8000 0x400>;
1918            clocks = <&rcc GPIOI>;
1919            st,bank-name = "GPIOI";
1920            status = "disabled";
1921        };
1944    };
```

第 1912~1921 行就是 GPIOI 的控制器信息，属于 pincrtl 的子节点，因此对于 STM32MP1 而言，pinctrl 和 gpio 这两个子系统的驱动文件是一样的，都为 pinctrl-stm32mp157.c，所以在注册 pinctrl 驱动顺便会把 gpio 驱动程序一起注册。绑定文档 Documentation/devicetree/bindings/gpio/gpio.txt 详细描述了 gpio 控制器节点各个属性信息。

第 1913 行，“gpio-controller”表示 gpioi 节点是个 GPIO 控制器，每个 GPIO 控制器节点必须包含“gpio-controller”属性。

第 1914 行，“#gpio-cells”属性和“#address-cells”类似，#gpio-cells 应该为 2，表示一共有两个 cell，第一个 cell 为 GPIO 编号，比如“&gpioi 0”就表示 PI0。第二个 cell 表示 GPIO 极性，如果为 0(GPIO_ACTIVE_HIGH)的话表示高电平有效，如果为 1(GPIO_ACTIVE_LOW)的话表示低电平有效。

第 1917 行，reg 属性设置了 GPIOI 控制器的寄存器基地址偏移为 0X800，因此 GPIOI 寄存器地址为 0X50002000+0X800=0X5000A000，大家可以打开《STM32MP157 参考手册》，找到“Table 9. Register boundary addresses”章节的 2.5.2 Memory map and register boundary addresses 小节，如图 25.2.2.1 所示：

0x5000A400 - 0x5000AFFF	3 KB	Reserved	-
0x5000A000 - 0x5000A3FF	1 KB	GPIOI	*GPIO registers*
0x50009400 - 0x50009FFF	3 KB	Reserved	-
0x50009000 - 0x500093FF	1 KB	GPIOH	*GPIO registers*
0x50008400 - 0x50008FFF	3 KB	Reserved	-
0x50008000 - 0x500083FF	1 KB	GPIOG	*GPIO registers*

图 25.2.2.1 GPIOI 寄存器表

从图 25.2.2.1 可以看出，GPIOI 控制器的基地址就是 0X5000A000，这个地址是基于 pinctrl 的地址 0X50002000+0x8000 = 0X5000A000。

第 1918 行，clocks 属性指定这个 GPIOI 控制器的时钟。

示例代码 25.2.2.1 中的是 GPIOI 控制器节点，当某个具体的引脚作为 GPIO 使用的时候还需要进一步设置。ST 官方 EVK 开发板将 PG1 用作 SD 卡的检测(CD)引脚，PG1 复用为 GPIO 功能，通过读取这个 GPIO 的高低电平就可以知道 SD 卡有没有插入。但是，SD 卡驱动程序怎么知道 CD 引脚连接的 PG1 呢？这里肯定需要设备树来告诉驱动，在设备树中的 SD 卡节点下添加一个属性来描述 SD 卡的 CD 引脚就行了，SD 卡驱动直接读取这个属性值就知道 SD 卡的 CD 引脚使用的是哪个 GPIO 了。ST 官方 EVK 开饭的 SD 卡连接在 STM32MP157 的 sdmmc1 接口上，在 stm32mp15xx-edx.dtsi 中找到名为“sdmmc1”的节点，这个节点就是 SD 卡设备节点，如下所示：

示例代码 25.2.2.2 设备树中 SD 卡节点

```
333 &sdmmc1 {
334     pinctrl-names = "default", "opendrain", "sleep";
335     pinctrl-0 = <&sdmmc1_b4_pins_a &sdmmc1_dir_pins_a>;
336     pinctrl-1 = <&sdmmc1_b4_od_pins_a &sdmmc1_dir_pins_a>;
337     pinctrl-2 = <&sdmmc1_b4_sleep_pins_a &sdmmc1_dir_sleep_pins_a>;
338     cd-gpios = <&gpiog 1 (GPIO_ACTIVE_LOW | GPIO_PULL_UP)>;
339     disable-wp;
......
351     status = "okay";
352 };
```

第 338 行，属性“cd-gpios”描述了 SD 卡的 CD 引脚使用的哪个 IO。属性值一共有三个，我们来看一下这三个属性值的含义，“&gpiog”表示 CD 引脚所使用的 IO 属于 GPIOG 组，“1”表示 GPIOG 组的第 1 号 IO，通过这两个值 SD 卡驱动程序就知道 CD 引脚使用了 PG1 这 GPIO。最后一个是“GPIO_ACTIVE_LOW | GPIO_PULL_UP”，Linux 内核定义在 include/linux/gpio/machine.h 文件中定义了枚举类型 gpio_lookup_flags，内容如下：

示例代码 25.2.2.3 gpio_lookup_flag 枚举类型

```
8  enum gpio_lookup_flags {
9       GPIO_ACTIVE_HIGH     = (0 << 0),
10      GPIO_ACTIVE_LOW      = (1 << 0),
11      GPIO_OPEN_DRAIN      = (1 << 1),
12      GPIO_OPEN_SOURCE     = (1 << 2),
13      GPIO_PERSISTENT      = (0 << 3),
14      GPIO_TRANSITORY      = (1 << 3),
15      GPIO_PULL_UP         = (1 << 4),
```

```
16      GPIO_PULL_DOWN          = (1 << 5),
17      GPIO_LOOKUP_FLAGS_DEFAULT   = GPIO_ACTIVE_HIGH |
GPIO_PERSISTENT,
18 };
```

我们可以通过或运算组合不同的配置内容，示例代码 25.2.2.2 中的 338 行，“GPIO_ACTIVE_LOW”表示低电平有效，“GPIO_PULL_UP”表示上拉，所以 PG1 引脚默认上拉，而且电平有效(当 PG1 被拉低的时候表示 SD 卡插入)。

这里也可以看出，把 PG1 用作 GPIO 的时候不需要添加其对应的 pincrl 节点！

2、GPIO 驱动程序简介

在 25.1.2 小节里已经分析过 pinctrl 驱动代码了，前面一小节说过了 STM32MP1 的 pinctrl 驱动和 gpio 驱动是同一个驱动文件，都为 pinctrl-stm32mp157.c，所以他们的入口函数都是 stm32_pctl_probe，找到如下代码所示：

示例代码 25.2.2.3 stm32_pctl_probe 代码段

```
1452    int stm32_pctl_probe(struct platform_device *pdev)
1453    {
......
1585    for_each_available_child_of_node(np, child) {
1586        if (of_property_read_bool(child, "gpio-controller")) {
1587            ret = stm32_gpiolib_register_bank(pctl, child);
1588            if (ret) {
1589                of_node_put(child);
1590                return ret;
1591            }
1592        }
1593    }
......
1600}
```

第 1586 行，判断设备树节点，是否有 gpio-controller。如果存在，那么这个节点就是一个 GPIO 控制器节点。

第 1587 行，stm32_gpiolib_register_bank 函数用来注册 GPIO 驱动，包括生成回调函数，注册的过程是跟 pinctrl 驱动注册是一样的。都是创建自己的结构体，然后初始化结构体，调用内核的注册函数，这样把自己的结构体注册到内核。这边就不去分析了，大家可以试下分析 GPIO 驱动代码。

25.2.3 gpio 子系统 API 函数

对于驱动开发人员，设置好设备树以后就可以使用 gpio 子系统提供的 API 函数来操作指定的 GPIO，gpio 子系统向驱动开发人员屏蔽了具体的读写寄存器过程。这就是驱动分层与分离的好处，大家各司其职，做好自己的本职工作即可。gpio 子系统提供的常用的 API 函数有下面几个：

1、gpio_request 函数

gpio_request 函数用于申请一个 GPIO 管脚，在使用一个 GPIO 之前一定要使用 gpio_request 进行申请，函数原型如下：

```
int gpio_request(unsigned gpio,    const char *label)
```

函数参数和返回值含义如下：

gpio：要申请的 gpio 标号，使用 of_get_named_gpio 函数从设备树获取指定 GPIO 属性信息，此函数会返回这个 GPIO 的标号。

label：给 gpio 设置个名字。

返回值：0，申请成功；其他值，申请失败。

2、gpio_free 函数

如果不使用某个 GPIO 了，那么就可以调用 gpio_free 函数进行释放。函数原型如下：

```
void gpio_free(unsigned gpio)
```

函数参数和返回值含义如下：

gpio：要释放的 gpio 标号。

返回值：无。

3、gpio_direction_input 函数

此函数用于设置某个 GPIO 为输入，函数原型如下所示：

```
int gpio_direction_input(unsigned gpio)
```

函数参数和返回值含义如下：

gpio：要设置为输入的 GPIO 标号。

返回值：0，设置成功；负值，设置失败。

4、gpio_direction_output 函数

此函数用于设置某个 GPIO 为输出，并且设置默认输出值，函数原型如下：

```
int gpio_direction_output(unsigned gpio, int value)
```

函数参数和返回值含义如下：

gpio：要设置为输出的 GPIO 标号。

value：GPIO 默认输出值。

返回值：0，设置成功；负值，设置失败。

5、gpio_get_value 函数

此函数用于获取某个 GPIO 的值(0 或 1)，此函数是个宏，定义所示：

```
#define gpio_get_value    __gpio_get_value
int __gpio_get_value(unsigned gpio)
```

函数参数和返回值含义如下：

gpio：要获取的 GPIO 标号。

返回值：非负值，得到的 GPIO 值；负值，获取失败。

6、gpio_set_value 函数

此函数用于设置某个 GPIO 的值，此函数是个宏，定义如下

```
#define gpio_set_value    __gpio_set_value
void __gpio_set_value(unsigned gpio, int value)
```

函数参数和返回值含义如下：

gpio：要设置的 GPIO 标号。

value：要设置的值。

返回值：无

关于 gpio 子系统常用的 API 函数就讲这些，这些是我们用的最多的。

25.2.4 设备树中添加 gpio 节点模板

本节我们以正点原子 STM32MP157 开发板上的 LED0 为例，学习一下如何创建 GPIO 节点。LED0 连接到了 PI0 引脚上，首先创建一个“led”设备节点。

1、创建 led 设备节点

在根节点“/”下创建 test 设备子节点，如下所示：

示例代码 25.2.4.1 led 设备节点

```
1 led {
2   /* 节点内容 */
3 };
```

2、添加 GPIO 属性信息

在 led 节点中添加 GPIO 属性信息，表明 test 所使用的 GPIO 是哪个引脚，添加完成以后如下所示：

示例代码 25.2.4.2 向 led 节点添加 gpio 属性

```
1 led {
2   compatible = "atk,led";
3   gpio = <&gpioi 0 GPIO_ACTIVE_LOW>;
4   status = "okay";
5 };
```

第 3 行，led 设备所使用的 gpio。

关于 pinctrl 子系统和 gpio 子系统就讲解到这里，接下来就使用 pinctrl 和 gpio 子系统来驱动 STM32MP1 开发板上的 LED 灯。

25.2.5 与 gpio 相关的 OF 函数

在示例代码 25.2.4.2 中，我们定义了一个名为“gpio”的属性，gpio 属性描述了 led 这个设备所使用的 GPIO。在驱动程序中需要读取 gpio 属性内容，Linux 内核提供了几个与 GPIO 有关的 OF 函数，常用的几个 OF 函数如下所示：

1、of_gpio_named_count 函数

of_gpio_named_count 函数用于获取设备树某个属性里面定义了几个 GPIO 信息，要注意的是空的 GPIO 信息也会被统计到，比如：

```
gpios = <0
        &gpio1 1 2
        0
        &gpio2 3 4>;
```

上述代码的“gpios”节点一共定义了 4 个 GPIO，但是有 2 个是空的，没有实际的含义。通过 of_gpio_named_count 函数统计出来的 GPIO 数量就是 4 个，此函数原型如下：

```
int of_gpio_named_count(struct device_node *np, const char  *propname)
```

函数参数和返回值含义如下：

np：设备节点。

propname：要统计的 GPIO 属性。

返回值：正值，统计到的 GPIO 数量；负值，失败。

2、of_gpio_count 函数

和 of_gpio_named_count 函数一样，但是不同的地方在于，此函数统计的是“gpios”这个属性的 GPIO 数量，而 of_gpio_named_count 函数可以统计任意属性的 GPIO 信息，函数原型如下所示：

```
int of_gpio_count(struct device_node *np)
```

函数参数和返回值含义如下：

np：设备节点。

返回值：正值，统计到的 GPIO 数量；负值，失败。

3、of_get_named_gpio 函数

此函数获取 GPIO 编号，因为 Linux 内核中关于 GPIO 的 API 函数都要使用 GPIO 编号，此函数会将设备树中类似<&gpioi 0(GPIO_ACTIVE_LOW | GPIO_PULL_UP)>的属性信息转换为对应的 GPIO 编号，此函数在驱动中使用很频繁！函数原型如下：

```
int of_get_named_gpio(struct device_node *np,
                      const char *propname,
                      int index)
```

函数参数和返回值含义如下：

np：设备节点。

propname：包含要获取 GPIO 信息的属性名。

index：GPIO 索引，因为一个属性里面可能包含多个 GPIO，此参数指定要获取哪个 GPIO 的编号，如果只有一个 GPIO 信息的话此参数为 0。

返回值：正值，获取到的 GPIO 编号；负值，失败。

25.3 硬件原理图分析

本实验的硬件原理参考 21.2 小节即可。

25.4 实验程序编写

本实验对应的例程路径为：开发板光盘→ 1、程序源码→2、Linux 驱动例程→5_gpioled。

本章实验我们继续研究 LED 灯，在第二十四章实验中我们通过设备树向 dtsled.c 文件传递相应的寄存器物理地址，然后在驱动文件中配置寄存器。本章实验我们使用 gpio 子系统来完成 LED 灯驱动。

25.4.1 修改设备树文件

在 stm32mp157d-atk.dts 文件的根节点“/”下创建 LED 灯节点，节点名为“gpioled”，节点内容如下：

示例代码 25.4.1.1 创建 LED 灯节点

```
1  gpioled {
2      compatible = "alientek,led";
3      status = "okay";
4      led-gpio = <&gpioi 0 GPIO_ACTIVE_LOW>;
5  };
```

第 4 行，led-gpio 属性指定了 LED 灯所使用的 GPIO，在这里就是 GPIOI 的 0 号，低电平有效。稍后编写驱动程序的时候会获取 led-gpio 属性的内容来得到 GPIO 编号，因为 gpio 子系统的 API 操作函数需要 GPIO 编号。设备树编写完成以后使用“make dtbs”命令重新编译设备树，然后使用新编译出来的 stm32mp157d-atk.dtb 文件启动 Linux 系统。启动成功以

后进入“/proc/device-tree”目录中查看“gpioled”节点是否存在，如果存在的话就说明设备树基本修改成功(具体还要驱动验证)，结果如图 25.4.1.1 所示：

```
[root@ATK-stm32mp1]:~$ ls /proc/device-tree/
#address-cells                  mlahb
#size-cells                     model
aliases                         name
arm-pmu                         pm_domain
buck1                           psci
chosen                          reboot
compatible                      regulator-3p3v
cpu0-opp-table                  regulator-booster
cpus                            reserved-memory
firmware                        serial-number
gpioled                         soc
interrupt-controller@a0021000   sram@2ffff000
mailbox-0                       stm32mp1_led
mailbox-1                       thermal-zones
memory@c0000000                 timer
```


图 25.4.1.1 gpioled 子节点

25.4.2 LED 灯驱动程序编写

设备树准备好以后就可以编写驱动程序了，本章实验在第四十四章实验驱动文件 dtsled.c 的基础上修改而来。新建名为“5_gpioled”文件夹，然后在 5_gpioled 文件夹里面创建 vscode 工程，工作区命名为“gpioled”。工程创建好以后新建 gpioled.c 文件，在 gpioled.c 里面输入如下内容：

示例代码 25.4.1.2 gpioled.c 驱动文件代码

```
1  #include <linux/types.h>
2  #include <linux/kernel.h>
3  #include <linux/delay.h>
4  #include <linux/ide.h>
5  #include <linux/init.h>
6  #include <linux/module.h>
7  #include <linux/errno.h>
8  #include <linux/gpio.h>
9  #include <linux/cdev.h>
10 #include <linux/device.h>
11 #include <linux/of.h>
12 #include <linux/of_address.h>
13 #include <linux/of_gpio.h>
14 #include <asm/mach/map.h>
15 #include <asm/uaccess.h>
16 #include <asm/io.h>
17 /***************************************************************
18 Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
19 文件名      : gpioled.c
20 作者        : 正点原子 Linux 团队
21 版本        : V1.0
22 描述        : gpio 子系统驱动 LED 灯。
```

```
23  其他        : 无
24  论坛        : www.openedv.com
25  日志        : 初版 V1.0 2020/12/30 正点原子 Linux 团队创建
26  ***************************************************************/
27  #define GPIOLED_CNT      1              /* 设备号个数    */
28  #define GPIOLED_NAME     "gpioled"      /* 名字          */
29  #define LEDOFF           0              /* 关灯          */
30  #define LEDON            1              /* 开灯          */
31
32  /* gpioled 设备结构体 */
33  struct gpioled_dev{
34      dev_t devid;                /* 设备号    */
35      struct cdev cdev;           /* cdev      */
36      struct class *class;        /* 类        */
37      struct device *device;      /* 设备      */
38      int major;                  /* 主设备号  */
39      int minor;                  /* 次设备号  */
40      struct device_node  *nd;    /* 设备节点  */
41      int led_gpio;               /* led 所使用的 GPIO 编号     */
42  };
43
44  struct gpioled_dev gpioled;     /* led 设备  */
45
46  /*
47   * @description   : 打开设备
48   * @param - inode : 传递给驱动的 inode
49   * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
50   *                  一般在 open 的时候将 private_data 指向设备结构体。
51   * @return        : 0 成功;其他 失败
52   */
53  static int led_open(struct inode *inode, struct file *filp)
54  {
55      filp->private_data = &gpioled; /* 设置私有数据 */
56      return 0;
57  }
58
59  /*
60   * @description   : 从设备读取数据
61   * @param - filp  : 要打开的设备文件(文件描述符)
62   * @param - buf   : 返回给用户空间的数据缓冲区
63   * @param - cnt   : 要读取的数据长度
64   * @param - offt  : 相对于文件首地址的偏移
65   * @return        : 读取的字节数，如果为负值，表示读取失败
66   */
```

```
67  static ssize_t led_read(struct file *filp, char __user *buf,
                            size_t cnt, loff_t *offt)
68  {
69      return 0;
70  }
71
72  /*
73   * @description   : 向设备写数据
74   * @param - filp  : 设备文件，表示打开的文件描述符
75   * @param - buf   : 要写给设备写入的数据
76   * @param - cnt   : 要写入的数据长度
77   * @param - offt  : 相对于文件首地址的偏移
78   * @return        : 写入的字节数，如果为负值，表示写入失败
79   */
80  static ssize_t led_write(struct file *filp, const char __user *buf,
                            size_t cnt, loff_t *offt)
81  {
82      int retvalue;
83      unsigned char databuf[1];
84      unsigned char ledstat;
85      struct gpioled_dev *dev = filp->private_data;
86
87      retvalue = copy_from_user(databuf, buf, cnt);
88      if(retvalue < 0) {
89          printk("kernel write failed!\r\n");
90          return -EFAULT;
91      }
92
93      ledstat = databuf[0];       /* 获取状态值 */
94
95      if(ledstat == LEDON) {
96          gpio_set_value(dev->led_gpio, 0);   /* 打开 LED 灯 */
97      } else if(ledstat == LEDOFF) {
98          gpio_set_value(dev->led_gpio, 1);   /* 关闭 LED 灯 */
99      }
100     return 0;
101 }
102
103 /*
104  * @description   : 关闭/释放设备
105  * @param - filp  : 要关闭的设备文件(文件描述符)
106  * @return        : 0 成功;其他 失败
107  */
108 static int led_release(struct inode *inode, struct file *filp)
```

```
109 {
110     return 0;
111 }
112 
113 /* 设备操作函数 */
114 static struct file_operations gpioled_fops = {
115     .owner = THIS_MODULE,
116     .open = led_open,
117     .read = led_read,
118     .write = led_write,
119     .release =  led_release,
120 };
121 
122 /*
123  * @description   : 驱动出口函数
124  * @param         : 无
125  * @return        : 无
126  */
127 static int __init led_init(void)
128 {
129     int ret = 0;
130     const char *str;
131 
132     /* 设置 LED 所使用的 GPIO        */
133     /* 1、获取设备节点：gpioled    */
134     gpioled.nd = of_find_node_by_path("/gpioled");
135     if(gpioled.nd == NULL) {
136         printk("gpioled node not find!\r\n");
137         return -EINVAL;
138     }
139 
140     /* 2.读取 status 属性 */
141     ret = of_property_read_string(gpioled.nd, "status", &str);
142     if(ret < 0)
143         return -EINVAL;
144 
145     if (strcmp(str, "okay"))
146         return -EINVAL;
147 
148     /* 3、获取 compatible 属性值并进行匹配 */
149     ret = of_property_read_string(gpioled.nd, "compatible", &str);
150     if(ret < 0) {
151         printk("gpioled: Failed to get compatible property\n");
152         return -EINVAL;
```

```
153     }
154
155     if (strcmp(str, "alientek,led")) {
156         printk("gpioled: Compatible match failed\n");
157         return -EINVAL;
158     }
159
160     /* 4、 获取设备树中的 gpio 属性，得到 LED 所使用的 LED 编号 */
161     gpioled.led_gpio = of_get_named_gpio(gpioled.nd, "led-gpio", 0);
162     if(gpioled.led_gpio < 0) {
163         printk("can't get led-gpio");
164         return -EINVAL;
165     }
166     printk("led-gpio num = %d\r\n", gpioled.led_gpio);
167
168     /* 5.向 gpio 子系统申请使用 GPIO */
169     ret = gpio_request(gpioled.led_gpio, "LED-GPIO");
170     if (ret) {
171         printk(KERN_ERR "gpioled: Failed to request led-gpio\n");
172         return ret;
173     }
174
175     /* 6、设置 PIO 为输出，并且输出高电平，默认关闭 LED 灯 */
176     ret = gpio_direction_output(gpioled.led_gpio, 1);
177     if(ret < 0) {
178         printk("can't set gpio!\r\n");
179     }
180
181     /* 注册字符设备驱动 */
182     /* 1、创建设备号 */
183     if (gpioled.major) {        /*  定义了设备号 */
184         gpioled.devid = MKDEV(gpioled.major, 0);
185         ret = register_chrdev_region(gpioled.devid, GPIOLED_CNT,
                                         GPIOLED_NAME);
186         if(ret < 0) {
187             pr_err("cannot register %s char driver [ret=%d]\n",
                                GPIOLED_NAME, GPIOLED_CNT);
188             goto free_gpio;
189         }
190     } else {                        /* 没有定义设备号 */
191         ret = alloc_chrdev_region(&gpioled.devid, 0, GPIOLED_CNT,
                                GPIOLED_NAME);    /* 申请设备号 */
192         if(ret < 0) {
193             pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
```

```
                    GPIOLED_NAME, ret);
194             goto free_gpio;
195         }
196         gpioled.major = MAJOR(gpioled.devid); /* 获取分配好的主设备号 */
197         gpioled.minor = MINOR(gpioled.devid); /* 获取分配好的次设备号 */
198     }
199     printk("gpioled major=%d,minor=%d\r\n",gpioled.major,
                gpioled.minor);
200
201     /* 2、初始化 cdev */
202     gpioled.cdev.owner = THIS_MODULE;
203     cdev_init(&gpioled.cdev, &gpioled_fops);
204
205     /* 3、添加一个 cdev */
206     cdev_add(&gpioled.cdev, gpioled.devid, GPIOLED_CNT);
207     if(ret < 0)
208         goto del_unregister;
209
210     /* 4、创建类 */
211     gpioled.class = class_create(THIS_MODULE, GPIOLED_NAME);
212     if (IS_ERR(gpioled.class)) {
213         goto del_cdev;
214     }
215
216     /* 5、创建设备 */
217     gpioled.device = device_create(gpioled.class, NULL,
                gpioled.devid, NULL, GPIOLED_NAME);
218     if (IS_ERR(gpioled.device)) {
219         goto destroy_class;
220     }
221     return 0;
222
223 destroy_class
224     class_destroy(gpioled.class);
225 del_cdev:
226     cdev_del(&gpioled.cdev);
227 del_unregister:
228     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
229 free_gpio:
230     gpio_free(gpioled.led_gpio);
231     return -EIO;
232 }
233
234 /*
```

```
235  * @description   : 驱动出口函数
236  * @param         : 无
237  * @return        : 无
238  */
239 static void __exit led_exit(void)
240 {
241     /* 注销字符设备驱动 */
242     cdev_del(&gpioled.cdev);        /*  删除 cdev   */
243     unregister_chrdev_region(gpioled.devid, GPIOLED_CNT);
244     device_destroy(gpioled.class, gpioled.devid);
245     class_destroy(gpioled.class);   /* 注销类       */
246     gpio_free(gpioled.led_gpio);    /* 释放 GPIO    */
247 }
248
249 module_init(led_init);
250 module_exit(led_exit);
251 MODULE_LICENSE("GPL");
252 MODULE_AUTHOR("ALIENTEK");
253 MODULE_INFO(intree, "Y");
```

第 41 行，在设备结构体 gpioled_dev 中加入 led_gpio 这个成员变量，此成员变量保存 LED 等所使用的 GPIO 编号。

第 55 行，将设备结构体变量 gpioled 设置为 filp 的私有数据 private_data。

第 85 行，通过读取 filp 的 private_data 成员变量来得到设备结构体变量，也就是 gpioled。这种将设备结构体设置为 filp 私有数据的方法在 Linux 内核驱动里面非常常见。

第 96、98 行，直接调用 gpio_set_value 函数来向 GPIO 写入数据，实现开/关 LED 的效果。不需要我们直接操作相应的寄存器。

第 134 行，获取节点“/gpioled”。

第 141~146 行，获取“status”属性的值，判断属性是否“okay”。

第 149~153 行，获取 compatible 属性值并进行匹配。

第 161 行，通过函数 of_get_named_gpio 函数获取 LED 所使用的 LED 编号。相当于将 gpioled 节点中的“led-gpio”属性值转换为对应的 LED 编号。

第 169 行，通过函数 gpio_request 向 GPIO 子系统申请使用 PI0。

第 176 行，调用函数 gpio_direction_output 设置 PI0 这个 GPIO 为输出，并且默认高电平，这样默认就会关闭 LED 灯。

可以看出 gpioled.c 文件中的内容和第二十四章的 dtsled.c 差不多，只是取消掉了配置寄存器的过程，改为使用 Linux 内核提供的 API 函数。在 GPIO 操作上更加的规范化，符合 Linux 代码框架，而且也简化了 GPIO 驱动开发的难度，以后我们所有例程用到 GPIO 的地方都采用此方法。

25.4.3 编写测试 APP

本章直接使用第四十二章的测试 APP，将上一章的 ledApp.c 文件复制到本章实验工程下即可。

25.5 运行测试

25.5.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第二十章实验基本一样，只是将 obj-m 变量的值改为 gpioled.o，Makefile 内容如下所示：

示例代码 25.5.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/my_linux/linux-5.4.31
......
4  obj-m := gpioled.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 gpioled.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“gpioled.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 ledApp.c 这个测试程序：

```
arm-none-linux-gnueabihf-gcc ledApp.c -o ledApp
```

编译成功以后就会生成 ledApp 这个应用程序。

25.5.2 运行测试

将上一小节编译出来的 gpioled.ko 和 ledApp 这两个文件拷贝到 rootfs/lib/modules/5.4.31 目录中，重启开发板，进入到目录 lib/modules/5.4.31 中，输入如下命令加载 gpioled.ko 驱动模块：

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe gpioled        //加载驱动
```

驱动加载成功以后会在终端中输出一些信息，如图 25.5.2.1 所示：

```
[root@ATK-stm32mp1]:/lib/modules/5.4.31$ modprobe gpioled.ko
led-gpio num = 128
gpioled major=241,minor=0
[root@ATK-stm32mp1]:/lib/modules/5.4.31$
```

图 25.5.2.1 驱动加载成功以后输出的信息

从图 25.5.2.1 可以看出，gpioled 这个节点找到了，并且 PI0 这个 GPIO 的编号为 128。驱动加载成功以后就可以使用 ledApp 软件来测试驱动是否工作正常，输入如下命令打开 LED 灯：

```
./ledApp /dev/gpioled 1        //打开 LED 灯
```

输入上述命令以后观察开发板上的红色 LED 灯是否点亮，如果点亮的话说明驱动工作正常。在输入如下命令关闭 LED 灯：

```
./ledApp   /dev/gpioled 0      //关闭 LED 灯
```

输入上述命令以后观察开发板上的红色 LED 灯是否熄灭。如果要卸载驱动的话输入如下命令即可：

```
rmmod gpioled.ko
```