iMX51 GPIO前后端驱动架构

余旭,邓飞
预研及软件工程中心
January 28, 2013

任务目标:

- 在jzj项目gpio驱动的基础上，分析整理GPIO前后端驱动架构

GPIO前端驱动架构:

1. jzj项目gpio驱动中指纹模块电源控制的部分代码如下:

```
1
2  #define FINGER_PWR (2*32 + 20) /*GPIO_3_20*/
3
4  static long fingerprint_gpio_setup(int value)
5  {
6      long ret = 0;
7      switch (value) {
8      case CMD_FIG_POWERON:
9          ret = gpio_request(FINGER_PWR, "on");
10         if (ret)
11             goto error;
12
13         gpio_direction_output(FINGER_PWR, 0);
14         gpio_set_value(FINGER_PWR, 0);
15         gpio_free(FINGER_PWR);
16         break;
17     ......
18 }
```

2. 从上面代码可以知道设置GPIO的基本流程是：先request，判断该gpio是否已使用；然后设置该gpio是作为输入还是输出；再设置gpio的值，设置完gpio之后，要释放。若没有释放，下次再设置该gpio，在request的时候就会因为该gpio已使用，而报错。
3. 上面引出一个疑问： 设置指纹模块为power on是通过宏FINGER_PWR来实现的。而宏FINGER_PWR是一个整数值，又不是gpio的地址，为什么能成功设置相应的gpio呢？后续分析会回答这个问题。

✤ GPIO后端驱动架构:

初始化流程:

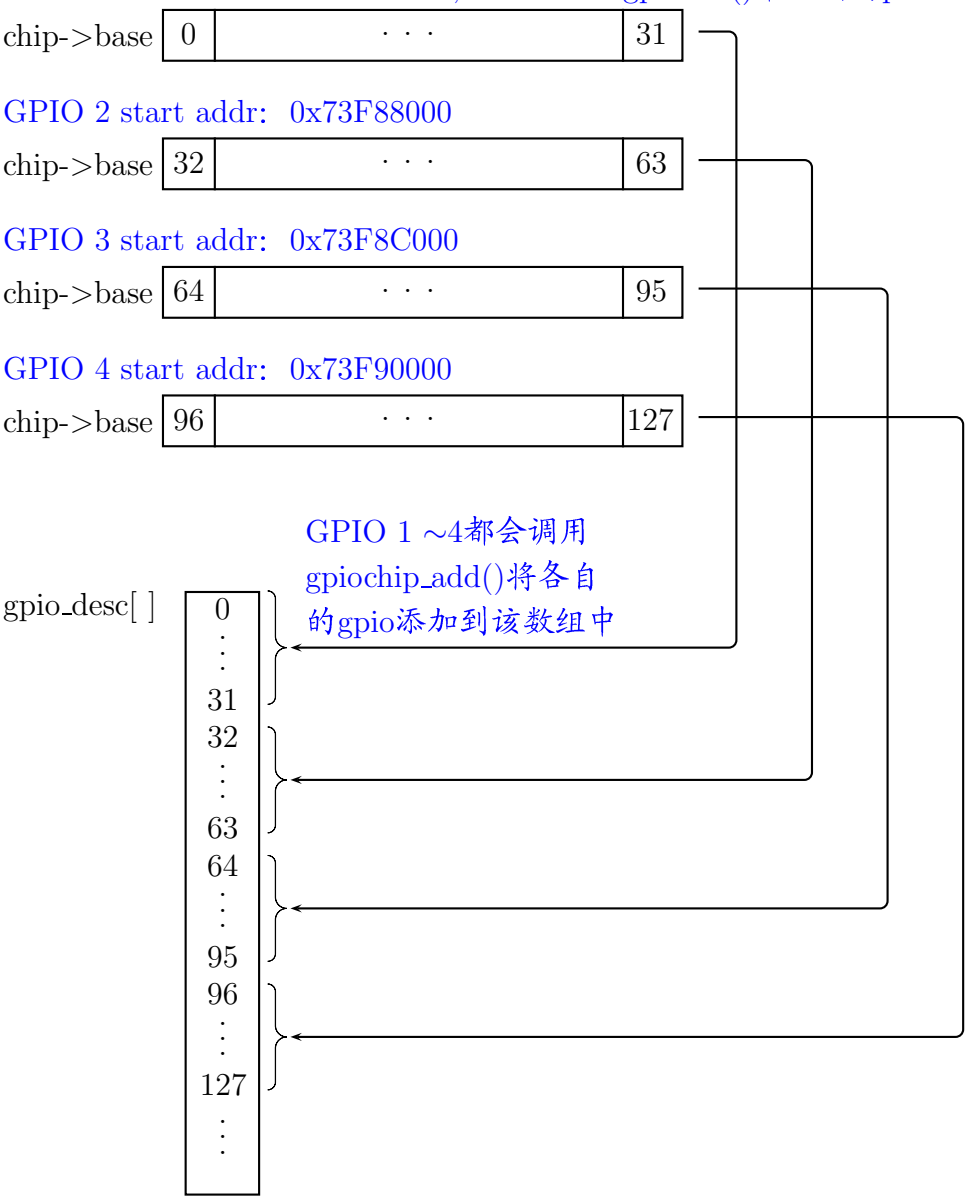
```
void _init start_kernel(void)
{  . . .
   setup_arch(&command_line);  ──→  void _init setup_arch(char **cmdline_p)
                                    {  . . .
                                       init_arch_irq = mdesc->init_irq;
                                       . . .
                                    }
   init_IRQ();                 ──→  init_arch_irq();
   . . .
}

MACHINE_START(MX51_TRICYCLE, ...)
   . . . . . .
   .init_irq = mx5_init_irq,   ──→  mxc_tzic_init_irq()
   . . . . . .                      mxc_register_gpio()
MACHINE_END

mxc_gpio_init(mxc_gpio_ports, 4)
{  . . .                          依次初始化mxc_gpio_ports结构体数组中的4组gpio
   for(i=0; i<4; i++){            port[0]~port[3]
      port[i].chip.direction_input = mxc_gpio_direction_input;
      port[i].chip.direction_output = mxc_gpio_direction_output;   在设置gpio pin时
      port[i].chip.get = mxc_gpio_get;                             实际上最终是调此
      port[i].chip.set = mxc_gpio_set;                             处挂接的函数指针

      port[i].chip.base = i * 32;      //设置每组gpio在数组中的起始位置
      port[i].chip.ngpio = 32;         //每组gpio的数量是32个
      gpiochip_add(&port[i].chip);     //关键函数解决上面疑问的地方
   }                                   //四组gpio,每组都会调用这个函数来完成将gpio
}                                      //添加到gpio_desc[]结构体数组中的工作

int gpiochip_add(struct gpio_chip *chip)
{  . . .
   unsigned id;
   int base = chip->base;   //chip的初始化在上面mxc_gpio_init()中
   for (id = base; id < base + chip->ngpio; id++)
      gpio_desc[id].chip = chip; //依次将每个gpio都添加到gpio_desc[]结构体数组中
   . . .                         //详见下图:
}
```

gpio与gpio_desc[]之间的关系:

上面的代码实际上是将每32个gpio组成一个mxc_gpio_port{},而这个mxc_gpio_port{}中含有一个虚拟的gpio_chip 。而为每个chip挂接的gpio操作函数指针都是一样的，只是每个chip在gpio_desc[ ]中的位置不同。

从iMX51 spec p63可以知道共有4组gpio,及每组gpio的基地址。

GPIO 1 start addr: 0x73F84000,也就是mxc_gpio_set()中提到的port->base



所以设置gpio时，传入整数值，作为gpio_desc[]结构体数组的下标，就能获取到相应的gpio，详见后续分析。

```
struct mxc_gpio_port mxc_gpio_ports[ ] = {
    {
        .chip.label = "gpio-0",
        .base = IO_ADDRESS(GPIO1_BASE_ADDR),//0x73F84000 <==iMX51 spec p63
    },{
        .chip.label = "gpio-1",
        .base = IO_ADDRESS(GPIO2_BASE_ADDR),
    },{
        .chip.label = "gpio-2",
        .base = IO_ADDRESS(GPIO3_BASE_ADDR),
    },{
        .chip.label = "gpio-3",
        .base = IO_ADDRESS(GPIO4_BASE_ADDR),
    }
}
```

✤ 相应函数的功能及实现分析:

1. gpio_request(): 用来检查某个gpio是否已经使用了，若已经在用了，返回busy。
   e.g: #define TRICYCLE_3G_PW (1*32 + 4)
   gpio_request(TRICYCLE_3G_PW, "3g_power");

```
int gpio_request(unsigned gpio, const char *label)
{
    struct gpio_desc *desc;
    struct gpio_chip *chip;

    desc = &gpio_desc[gpio]; //gpio_desc[]是在上面mxc_gpio_init()中
    chip = desc->chip;       //调用gpiochip_add()来实现初始化的

    //将desc->flags的第FLAG_REQUESTED位设为1，并返回该位原来的值
    //注意: FLAG_REQUESTED表示第几位，而不是要设置的值
    if (test_and_set_bit(FLAG_REQUESTED, &desc->flags) == 0){
        desc_set_label(desc, label ? : "?"); //实际上该函数内部宏没有开，
        status = 0;                          //其并没有实现设置label的功能
    } else
        status = -EBUSY;//gpio已经在使用，返回busy。

    if (chip->request)//chip在初始化时并没有挂接该函数指针，不会进
        status = chip->request(chip, gpio - chip->base);
}
```

2. gpio_free(): 当request了gpio，且使用完之后，要调用此函数释放。否则下次再request时就会因为该gpio正在使用而报busy错。
   e.g: gpio_free(TRICYCLE_3G_PW);

```
void gpio_free(unsigned gpio)
{
    struct gpio_desc *desc;
    struct gpio_chip *chip;

    desc = &gpio_desc[gpio]; //gpio_desc[]是在上面mxc_gpio_init()中
    chip = desc->chip;       //调用gpiochip_add()来实现初始化的

    //若desc->flags的第FLAG_REQUESTED位为1，则
    if (chip && test_bit(FLAG_REQUESTED, &desc->flags)) {
        desc_set_label(desc, NULL);
        clear_bit(FLAG_REQUESTED, &desc->flags);
    }
}
```

3. gpio_direction_input()/gpio_direction_output(): 设置gpio是作为输入还是输出功能。两者在实现上大致相同，在此仅以设为输入为例。
   e.g: gpio_direction_input(TRICYCLE_3G_PW);

```
int gpio_direction_input(unsigned gpio)
{
    struct gpio_desc *desc = &gpio_desc[gpio];
    struct gpio_chip *chip = desc->chip;

    //获取gpio在gpio_desc[]结构体数组中的下标
    gpio -= chip->base;
    status = chip->direction_input(chip, gpio); //在mxc_gpio_init()中挂接函数指针
}                    |  port[i].chip.direction_input = mxc_gpio_direction_input;
                     |  mxc_gpio_direction_input()又会调用下面这个函数
                     v
static void _set_gpio_direction(struct gpio_chip *chip, unsigned offset,int dir)
{  . . .
    u32 l;                                  set GPIO Direction Register
    l = __raw_readl(port->base + GPIO_GDIR);<==iMx51 spec p1036
    if (dir)                    往寄存器相应位写0,表示设为input，写1为output
        l |= 1 << offset;       这个offset就是gpio_direction_input()中求出的下标
    else                        每组gpio有32个，每个gpio就对应该寄存器中的一位
        l &= ~(1 << offset);
    __raw_writel(l, port->base + GPIO_GDIR);
}
```

4. gpio_set_value()/gpio_get_value(): 设置/获取gpio的值，最终会调用mxc_gpio_init()中挂接的函数指针来实现设置gpio的值。(如: port[i].chip.set = mxc_gpio_set;)
   e.g: gpio_set_value(TRICYCLE_3G_PW, 1);

```
#define gpio_set_value    __gpio_set_value
void __gpio_set_value(unsigned gpio, int value)
{
    struct gpio_chip *chip;
    //通过return gpio_des[gpio].chip来得到相应的chip
    chip = gpio_to_chip(gpio);
    chip->set(chip, gpio - chip->base, value);
}
                ↓
void mxc_gpio_set(struct gpio_chip *chip, unsigned offset, int value)
{
    struct mxc_gpio_port *port = container_of(chip, struct mxc_gpio_port, chip);
    void __iomem *reg = port->base + GPIO_DR;//iMX51 spec p1035

    u32 l = (__raw_readl(reg) & (~(1 << offset))) — (value << offset);
    __raw_writel(l, reg);
}
```

✤ 注意事项:

1. 若需要设置的gpin pin具有复合功能时，必须要根据iMx51 spec，调用mxc_request_iomut() 将该pin设置为gpio功能后，才能进行后续操作。

✤ 致谢:

感谢此次工作中，余工的指导工作。