IO 类问题排查总结

Author：lml

版本：2.0

日期：2019.04

目录

一、如何通过 IO 读寄存器确认引脚复用问题

1.1 kernel 阶段使用 io 命令确认寄存器方法：

许多客户经常会碰到一些 IO 引脚虽然已经在 dts 中配置了对应的 iomux，但却发现实际示波器测量出来信号不对。一般这种情况出现，可首先排查 IO 引脚是否被复用。在 RK 的 Android 平台，默认有包含 io 工具（源码位置：external\io），linux 系统平台如果没有此源码，可以将 Android 平台此源码打包过去编译即可（linux 平台代码同步最新都已带有 IO 工具，可直接使用命令）。

IO 工具使用方法：

IO 工具所包含的命令参数详解可以在串口或者 adb 输入“io?” 回车后便可罗列出。要查询 io 寄存器首先要有主控芯片详细规格书 TRM，请向对接的 RK 业务邮件申请（注：在 http://opensource.rock-chips.com/可以找到 3399、3288、3328 的 TRM 技术参考手册）。拿到规格书后，如何确认一个 IO 引脚是否被复用呢？ 以下用 RK3399 平台来举例，例如想查询的是 GPIO4_B0 引脚，在原理图上是：

接下来在 TRM 上搜索 gpio4b0（不带下杠，连着输入名称），可以找到寄存器：

1:0	RW	0x0	gpio4b0_sel GPIO4B[0] iomux select 2'b00: gpio 2'b01: sdmmc_data0 2'b10: uart2dbga_sin 2'b11: reserved

此时顺着往上翻滚可知道该寄存器的基地址名称是“GRF”：

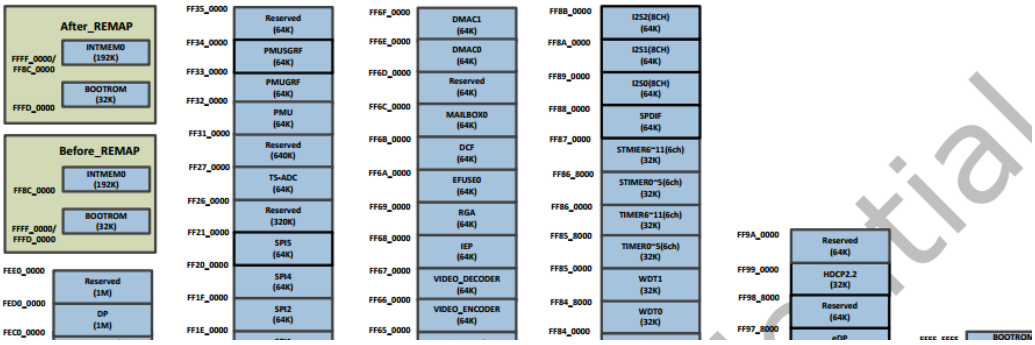
GRF_GPIO4B_IOMUX
Address: Operational Base + offset (0x0e024)
GPIO4B iomux control

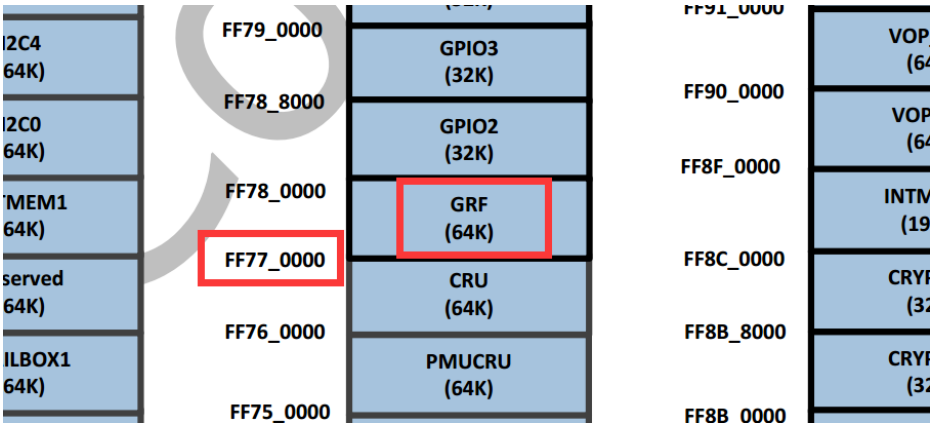
Bit	Attr	Reset Value	

下一步就是在 datasheet 中搜索 address mapping，并在其中找到 GRF 的基地址：

2.1 Address Mapping

RK3399 supports to boot from internal bootrom, which supports remap function by softw programming. Remap is controlled by SGRF_PMU_CON0[15]. When remap is set to 0, th 0xFFFF0000 address is mapped to bootrom. When remap is set to 1, the 0xFFFF0000 addr is mapped to INTMEM0.





明确地址后，在串口或者 adb 输入：

io -4 -l 0x100 0xff77e024

（注：这里显示 0x100 这么多个是因为比较直观，有时 gpio 的 clk 被锁也可直观看出）

回车可得到输出结果:

```
ff77e024:    00000555 0000855f 00000000 00000000
ff77e034:    00000000 00000000 00000000 000002a5
ff77e044:    00005450 0000ff03 00000143 00000000
ff77e054:    00004010 00000001 00000000 0000aa80
ff77e064:    00000455 00002040 0000aaa1 00000000
ff77e074:    00000000 00000000 00000000 00000000
ff77e084:    00000000 00000000 00000000 00000000
ff77e094:    00000000 00000000 00000000 00000000
ff77e0a4:    000000ff 00000000 00000000 00000000
ff77e0b4:    00000000 00000000 00000000 00000000
ff77e0c4:    00000000 00000000 00000000 00000000
ff77e0d4:    00000000 00000000 00000000 00000000
ff77e0e4:    00000000 00000000 00000000 00000000
ff77e0f4:    00000000 00000000 00000000 00000000
```

ff77e104: 00000000 00000000 00000000 0000b01b

ff77e114: 00000001 00003000 00000000 00000018

第一行的结果为：ff77e024: 00000555 0000855f 00000000 00000000

四个值分别对应地址：0xff77e024、0xff77e028、0xff77e02c、0xff77e030。那么 0xff77e024 输出结果就是 0x555，怎么看呢？ 因为 0x555 是十六进制，我们把它转化为二进制就是：0000 0000 0000 0000 0000 0101 0101 0101 ，这里写出完整的转化后的 32bit 方便大家观看，转换后的结果从右至左为低位到高位，即最右边的 bit 为第 0 bit，最左边的 bit 为第 31bit。那么对应到之前查询到的寄存器 ：GRF_GPIO4B_IOMUX 便可以一一对应来查看结果了。在 GRF_GPIO4B_IOMUX 寄存器中，gpio4b3 的功能寄存器是第 7 bit 和第 6 bit[7:6] :

7:6	RW	0x0	gpio4b3_sel GPIO4B[3] iomux select 2'b00: gpio 2'b01: sdmmc_data3 2'b10: cxcsjtag_tms 2'b11: hdcpjtag_tdo

而查看上面串口输出结果转换二进制后结果来看，[7:6] bit 就是 01，也就是对应该寄存器的 01：sdmmc_data3 这个功能。由此，我们知道了 gpio4b3 这个 IO 引脚当前的复用情况。如果这不是自己设置的状态，则去代码中查找哪里被复用即可。

如何查找代码中的复用？

打个比方，比如上面的 gpio4b3 这个引脚，已经在 dts 中设置了对应的 iomux，作为一个 gpio 去拉高拉低，但上面查询结果是 sdmmc，那么这时候我们首先要去 dts 中查找 sdmmc 节点先确认节点里是否有 pinctrl 里填错了 gpio，导致 mux 错了。绝大部分情况的 IO 复用问题都可以在 dts 中找到问题点。

1.2 Uboot 阶段寄存器状态确认方法：

如果是想要确认 uboot 阶段寄存器状态，可以使用 IO 命令或者 md 命令，一般 SDK 里都有带，不过默认没打开。

1.2.1 使用 IO 命令需打开以下宏：

```
diff --git a/include/configs/rk_default_config.h
b/include/configs/rk_default_config.h
index 9852917d4f..305de26347 100755
--- a/include/configs/rk_default_config.h
+++ b/include/configs/rk_default_config.h
@@ -304,7 +304,7 @@
/* rk io command tool */
-#undef CONFIG_RK_IO_TOOL
+#define CONFIG_RK_IO_TOOL
```

1.2.2 使用 MD 命令需添加以下宏：

```
diff --git a/include/configs/rk33plat.h b/include/configs/rk33plat.h
index 0906c0c091..3f94e9eabe 100755
--- a/include/configs/rk33plat.h
+++ b/include/configs/rk33plat.h
@@ -140,6 +140,7 @@
#define CONFIG_RK_GPIO_EXT_FUNC
#define CONFIG_CHARGE_LED
#define CONFIG_POWER_FUSB302
+ #define CONFIG_CMD_MEMORY
#endif
#if defined(CONFIG_RKCHIP_RK322XH)
```

然后 make distclean 重新选择配置编译即可，此时发现 u-boot/common/cmd_mem.c 被编译。IO 命令的话使用方法和上面介绍的 kernel 部分一样,这里不再做重复。

下面就来介绍 md 命令使用方法：

在打开 MD 的宏后， 在 uboot 代码：

```
diff --git a/common/autoboot.c b/common/autoboot.c
index c27cc2c751..fe28adb2fd 100644
--- a/common/autoboot.c
+++ b/common/autoboot.c
@@ -146,6 +146,7 @@ static int abortboot_normal(int bootdelay)
{
int abort = 0;
unsigned long ts;
+ bootdelay = 3; //此处添加时间可自由定义，单位：秒
#ifdef CONFIG_MENUPROMPT
printf(CONFIG_MENUPROMPT);
```

这样修改后编译烧录 uboot.img 在机器起来至打印 uboot 的：
Hit any key to stop autoboot:
这句会停下来倒数，倒数时间就是上面”bootdelay = 3;”设置的时间，设置多少就是几秒。只要在倒数时间内敲 PC 键盘任意按键便可让机器停在 uboot 阶段，这时候可使用 md 命令（io 命令同理）：
md 0x????????（这里后面的问号即是寄存器地址，想查询什么寄存器就填对应地址）
来读取想要查看的寄存器值。

二、 如何通过 IO 写寄存器

如果客户想要通过 IO 命令临时写一个寄存器做实验，可以通过 io –w 去写。
打个比方，已经通过命令： io -4 -r 0xff77e024 读出了寄存器的值，那么此时想对 0xff77e024 这个寄存器的第 0 个 bit 写入 1，那么可以如下操作：
Io -4 –w 0xff77e024 0x00010001
注：为什么寄存器地址后面的十六进制值的第 16 bit 要写 1？因为该寄存器的 16bit 至 31 bit 是写有效位，默认为 0，即不可写。因为要往第 0 bit 写 1，所以 0 bit 对应的写有效位 16 bit 也要对应置 1 才可写入，这个是根据寄存器描述而定的：

GRF_GPIO4B_IOMUX
Address: Operational Base + offset (0x0e024)
GPIO4B iomux control

Bit	Attr	Reset Value	Description
31:16	RW	0x0000	write_enable bit0~15 write enable When bit 16=1, bit 0 can be written by software . When bit 16=0, bit 0 cannot be written by software; When bit 17=1, bit 1 can be written by software . When bit 17=0, bit 1 cannot be written by software; ..... When bit 31=1, bit 15 can be written by software . When bit 31=0, bit 15 cannot be written by software;

需要注意的是，通过 io 写的寄存器值 reboot 后不会保留。

三、如何确认 IO 电源域软硬件是否匹配

一般来说，电源域软件配置跟硬件实际不匹配，会导致该路的整排 IO 口不受控制，出来的信号紊乱、不达标。如果在实际开发过程中，遇到某一路 IO 异常，查看复用状态没错也没别处调用，但是测量出来的信号就是不对。软件无法控制输出高低等。这时候就要去查一下这路的电源域寄存器配置了。

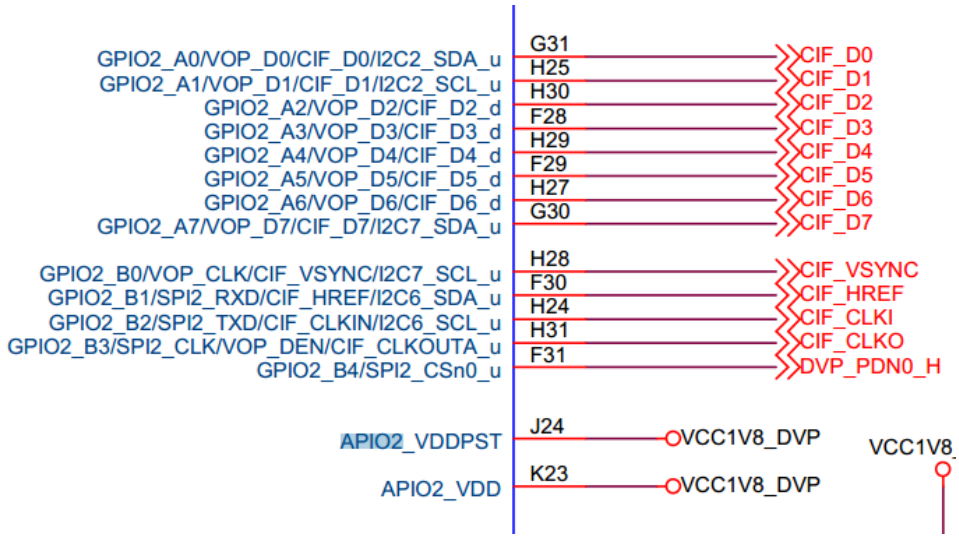
打开主控芯片对应的 TRM 芯片规格书中的 GRM/PMUGRF 章节搜索”VSEL”或者”volsel”、”vsel”这些关键字来找到 io-domain 寄存器。这里以 3399 为例：

GRF_IO_VSEL

Address: Operational Base + offset (0x0e640)

Bit	Attr	Reset Value	
			write_enable bit0~15 write
15:4	RO	0x0	reserved
3	RW	0x0	gpio1833_gpio4cd_ms
2	RW	0x0	sdmmc_gpio4b_ms
1	RW	0x0	audio_gpio3d4a_ms
0	RW	0x0	bt656_gpio2ab_ms

搜索关键字就能找到对应的寄存器。随后我们再来看原理图上引脚对应的电源，比如，我们发现 gpio2 这路的整排引脚控制不正常：



那么它们这排引脚对应的电源是 APIO2，接下来再去 sdk 代码的 kernel 目录下找到文档：kernel$ vim Documentation/devicetree/bindings/power/rockchip-io-domain.txt

这里面就有详尽的描述告知软件中是什么定义对应的 APIO2 这一路电源：

```
Possible supplies for rk3399:
- bt656-supply:  The supply connected to APIO2_VDD.
- audio-supply:  The supply connected to APIO5_VDD.
- sdmmc-supply:  The supply connected to SDMMC0_VDD.
- gpio1830       The supply connected to APIO4_VDD.

Possible supplies for rk3399 pmu-domains:
- pmu1830-supply:The supply connected to PMUIO2_VDD.
```

于是可以在 dts 目录下搜索“io_domain”找到：

```
&io_domains {
        status = "okay";

        bt656-supply = <&vcc1v8_dvp>;
        audio-supply = <&vcca1v8_codec>;
        sdmmc-supply = <&vcc_sd>;
        gpio1830-supply = <&vcc_3v0>;
};
```

确认软件配置里对应的项电源设置跟硬件一样即可：

bt656-supply = <&vcc1v8_dvp>;

那么此处设置的 1.8V 跟硬件原理图一致才是正确，如果不一样就改成一致，一般来说确认过电源域后可以解决绝大部分的 IO 引脚控制异常或者信号异常问题了。

四、 查看 GPIO 寄存器

客户想查看当前某 GPIO 状态，从而确认当前 GPIO 到底是输入还是输出、输出高还是输出低等问题，也可以通过 IO 命令实现。以下以 gpio4b0 举例：
通过搜索“ADDRESS MAPPING”在 TRM 上找到 gpio4 的基地址：

进而通过命令： io -4 -l 0x100 0xFF790000
可以查询到 gpio4 的状态，例如结果为：

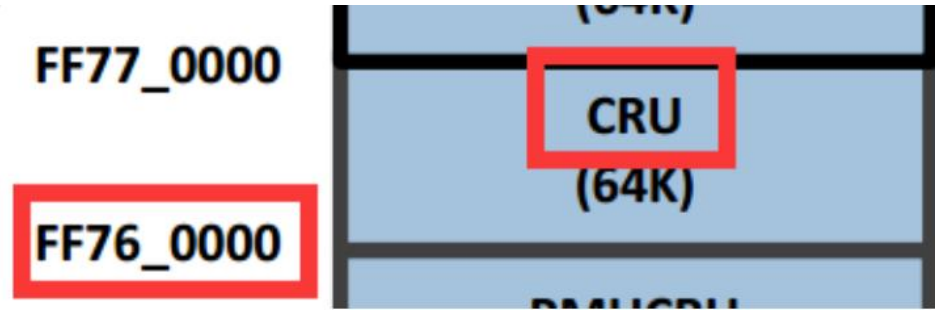
```
ff790000:  42400000 42400000 42400000 42400000
ff790010:  42400000 42400000 42400000 42400000
ff790020:  42400000 42400000 42400000 42400000
ff790030:  42400000 42400000 42400000 42400000
ff790040:  42400000 42400000 42400000 42400000
ff790050:  42400000 42400000 42400000 42400000
ff790060:  42400000 42400000 42400000 42400000
ff790070:  42400000 42400000 42400000 42400000
ff790080:  42400000 42400000 42400000 42400000
ff790090:  42400000 42400000 42400000 42400000
ff7900a0:  42400000 42400000 42400000 42400000
ff7900b0:  42400000 42400000 42400000 42400000
ff7900c0:  42400000 42400000 42400000 42400000
ff7900d0:  42400000 42400000 42400000 42400000
ff7900e0:  42400000 42400000 42400000 42400000
ff7900f0:  42400000 42400000 42400000 42400000
```

从结果中可以看出，所有的值都奇怪的一致。如果用 IO 命令读出来的值部分或者全部都一致的显示为某个奇怪的值，则很有可能这些 gpio 的 clk 被关闭，无法读写。
举个例子，此时想把 RK3399 上的 gpio4 clk 打开：
在 RK3399 TRM datasheet 上逐一搜索 gpio4 ，可以找到如下结果：

5	RW	0x0	pclk_gpio4_en pclk_gpio4 clock disable bit When HIGH, disable clock

往上翻滚可见寄存器基地址名称以及偏移地址：

再去查询 address mapping 找到基地址：



所以我们只要通过 IO 写命令把 gpio4 的 clk 打开即可以正常 IO 命令读写：
首先通过 IO 读命令确认当前的 gpio4 clk 是否为关闭（寄存器说明高为 disable）：

```
io -4 -l 0x100 0xFF76037C
ff76037c:  000001b8 00000010 00000000 0000002a
ff76038c:  00000000 00000000 00000000 00000000
ff76039c:  00000000 00000000 00000000 00000000
ff7603ac:  00000000 00000000 00000000 00000000
ff7603bc:  00000000 00000000 00000000 00000000
ff7603cc:  00000000 00000000 00000000 00000000
ff7603dc:  00000000 00000000 00000000 00000000
ff7603ec:  00000000 00000000 00000000 00000000
ff7603fc:  00000000 00000000 00000000 00000000
ff76040c:  00000000 00000000 00000000 00000000
ff76041c:  00000000 00000000 00004040 00000000
ff76042c:  00000014 00000000 00000000 00000000
ff76043c:  00000000 00000000 00000000 00000000
ff76044c:  00000000 00000000 00000000 00000000
ff76045c:  00000000 00000000 00000000 00000000
ff76046c:  00000000 00000000 00000000 00000000
```

出来结果是 0x000001b8，结果转成二进制也就是：0001 1011 1000，那么 gpio4 的 clk 对应的第 5 个 bit ，也就是转换成二进制后的数值从右往左数的第 5bit 为 1（1 即 disable），那么要把它打开， IO 命令才能正常读写。用 IO 写命令来打开：

```
io -4  -w 0Xff76037C 0x00200000
```

这里写入寄存器 0xff76037C 的值 0x00200000 转换成二进制就是：
0000 0000 0010 0000 0000 0000 0000 0000，因为根据该寄存器说明，高 16bit 为写有效位。 那么我们要往第 5bit 写入 0 使能 gpio4 clk，就要对应把第 5bit 对应的写使能位，即第 21bit 写入 1 使能。打开 gpio 4 的 clk 后，就可以使用 IO 命令自由读写 gpio4 的寄存器了。

打开 gpio4 的 pclk 后，再查询 gpio4 的寄存器，结果就正常显示了：

```
io -4 -l 0x100 0xFF790000
ff790000:  42000000 6a400000 00000000 00000000
ff790010:  00000000 00000000 00000000 00000000
ff790020:  00000000 00000000 00000000 00000000
ff790030:  ffffffff ffffffff 00000000 00000000
ff790040:  00000000 9434d0f9 00000000 00000000
ff790050:  438b2f06 00000000 00000000 00000000
ff790060:  00000000 00000000 00000000 3230372a
ff790070:  00000000 00000000 00000000 00000000
ff790080:  42000000 6a400000 00000000 00000000
ff790090:  00000000 00000000 00000000 00000000
ff7900a0:  00000000 00000000 00000000 00000000
ff7900b0:  ffffffff ffffffff 00000000 00000000
ff7900c0:  00000000 9434d0f9 00000000 00000000
ff7900d0:  438b2f06 00000000 00000000 00000000
ff7900e0:  00000000 00000000 00000000 3230372a
ff7900f0:  00000000 00000000 00000000 00000000
```

客户可根据结果对着 TRM 的 GPIO 章节寄存器介绍来对应查看了。比如我们要看 gpio4b0 引脚的状态，根据寄存器描述：

GPIO_SWPORTA_DR	0x0000	W	0x00000000	Port A data register
GPIO_SWPORTA_DDR	0x0004	W	0x00000000	Port A data direction register

0x4 偏移为输入输出状态寄存器，而 0x0 偏移为数据寄存器。

查看输入输出状态：0xff790004 结果为 0x6a400000，结果对应位为：

6a	40	00	00
Gpio4D	Gpio4C	Gpio4B	Gpio4A

而每两位十六进制数字转换为二进制数值后从右到左对应 0 到 7 脚，比如 6a 二进制是： 0110 1010，那么二进制从右到左分别对应 Gpio4D0 到 Gpio4D7 的状态。根据寄存器描述：

GPIO_SWPORTA_DDR
Address: Operational Base + offset (0x0004)
Port A data direction register

Bit	Attr	Reset Value	Description
31:0	RW	0x00000000	gpio_swporta_ddr Values written to this register independently control the direction of the corresponding data bit in Port A. 0: Input (default) 1: Output

0 代表输入，1 代表输出。所以此时 gpio4b0 是输入状态。假如此时它是输出状态，可通过查看 0xFF790000 的 0 偏移的值可确认输出的是高还是低电平。

五、其它驱动调用 GPIO 导致冲突

还有一种常见问题：比如一个 IO 引脚，客户想作为 GPIO 使用，按照上面方法确认了 mux 没错，并且电源域确认也没问题，但是示波器测量波形感觉没有按照自己驱动代码来拉高拉低，有些“不受控”。这时候很可能就是有其它驱动在操作这个 GPIO。很多客户喜欢去 gpio 节点下面查询，但是很多时候节点下面并不能显示具体是被什么驱动调用，显示也并不完善。以下介绍两个常用的找到其它调用处方法：

方法一：

思路：驱动里如果想要去操作 GPIO，肯定会调用到 gpio_direction_output、gpio_direction_input、gpiod_direction_output、gpiod_direction_input 这几个接口，他们的定义位置是：vim kernel/drivers/gpio/gpiolib.c

如果我们能够在这些接口里添加判断对应查询的 IO 口条件，如果满足就打印出 dump_stack(); ，那么该 IO 口的调用就一目了然了。

这里需要说明一下的是，在 linux3.10 内核的 sdk 中用的是 gpio_direction_output 和 gpio_direction_input 接口，而在 linux4.4 内核中则是 gpiod_direction_output 和 gpiod_direction_input，详情可查看源码了解。

下面罗列出 linux3.10 版本和 linux4.4 版本添加 dump_stack 条件的方法供参考使用：

这里使用 gpio4b3 来举例。首先，需要计算出代表 gpio4b3 的值，算法如下：

Gpio4_B3 = 4 *32 + (B-A) * 8 + 3 = 3 *32 + 1 * 8 + 3 = 139

（注：最前面和 32 相乘的数字因为是 gpio4，所以是 4*32。如果是 gpio3，那就是 3*32；括号里面的 A、B、C、D 分别代表数值 0、1、2、3，在计算时候分别对应去减即可。这里因为是 B3，所以用 B-A，如果是 C3，就是 C-A；最后的+3 是因为是 GPIO4B3，如果是 GPIO4B2，那么最后就+2。）

Linux3.10 添加方法：

```
int gpio_direction_output(unsigned gpio, int value)
{
+       if (gpio == 139)
+       {
+               printk("dump_stack_start\n");
+               dump_stack();
+               printk("dump_stack_end\n");
+       }

        return gpiod_direction_output(gpio_to_desc(gpio), value);
}
```

Linux4.4 添加方法：

注：在 linux4.4 内核，io 引脚的值有些变化，也就是按照上文算法计算的结果要+1000，所以 GPIO4B3 如果是 linux4.4 内核里要填 1139。

```
int gpiod_direction_output(struct gpio_desc *desc, int value)
{
        if (!desc || !desc->chip) {
                pr_warn("%s: invalid GPIO\n", __func__);
                return -EINVAL;
        }
+         if ( desc_to_gpio(desc) == 1139)
+         {
+                printk("dump_stack_start\n");
+                  dump_stack();
+                printk("dump_stack_end\n");
+         }
        if (test_bit(FLAG_ACTIVE_LOW, &desc->flags))
                value = !value;
        return _gpiod_direction_output_raw(desc, value);
}
```

添加后编译烧录，只要对应判断的引脚有被调用，启动 log 中就会打印出堆栈，可以根据找出结果查看，找到驱动调用函数。

方法二：

```
diff --git a/drivers/pinctrl/pinctrl-rockchip.c
b/drivers/pinctrl/pinctrl-rockchip.c
index c6c04ac..c1dd0bd 100644
--- a/drivers/pinctrl/pinctrl-rockchip.c
+++ b/drivers/pinctrl/pinctrl-rockchip.c
@@ -661,6 +661,11 @@ static int rockchip_set_mux(struct rockchip_pin_bank
*bank, int pin, int mux)
dev_dbg(info->dev, "setting mux of GPIO%d-%d to %d\n", bank->bank_num,
pin,mux);

+ if ((bank->bank_num == 4) && (pin == 11)) {
+ printk("setting mux of GPIO%d-%d to %d\n", bank->bank_num, pin, mux);
+ dump_stack();
+ }

regmap = (bank->iomux[iomux_num].type & IOMUX_SOURCE_PMU)
? info->regmap_pmu : info->regmap_base
```

注：这里“pin ==”后面跟的值计算方式为：
将 A0 至 D7 32 个引脚顺序对应数值 0 至 31。

END：

如果以上方法还是没法解决客户的 IO 问题，请尽快联系 RK FAE 支持，谢谢！

另：关于如何设置 GPIO 默认上拉下拉疑问请参考 RKDocs 目录下的 Pin-Ctrl 文档：

《Rockchip Pin-Ctrl 开发指南 V1.0-20160725.pdf》 。