目录

[隐藏]

简介

GPIO，全称 General-Purpose Input/Output（通用输入输出），是一种软件运行期间能够动态配置和控制的通用引脚。 RK3399 有 5 组 GPIO bank：GPIO0~GPIO4，每组又以 A0~A7, B0~B7, C0~C7, D0~D7 作为编号区分（不是所有 bank 都有全部编号，例如 GPIO4 就只有 C0~C7, D0~D2)。 所有的 GPIO 在上电后的初始状态都是输入模式，可以通过软件设为上拉或下拉 ,也可以设置为中断脚 驱动强度都是可编程的。每个 GPIO 口

除了通用输入输出功能外，还可能有其它复用功能，例如 GPIO2_B2，可以利用成以下功能：

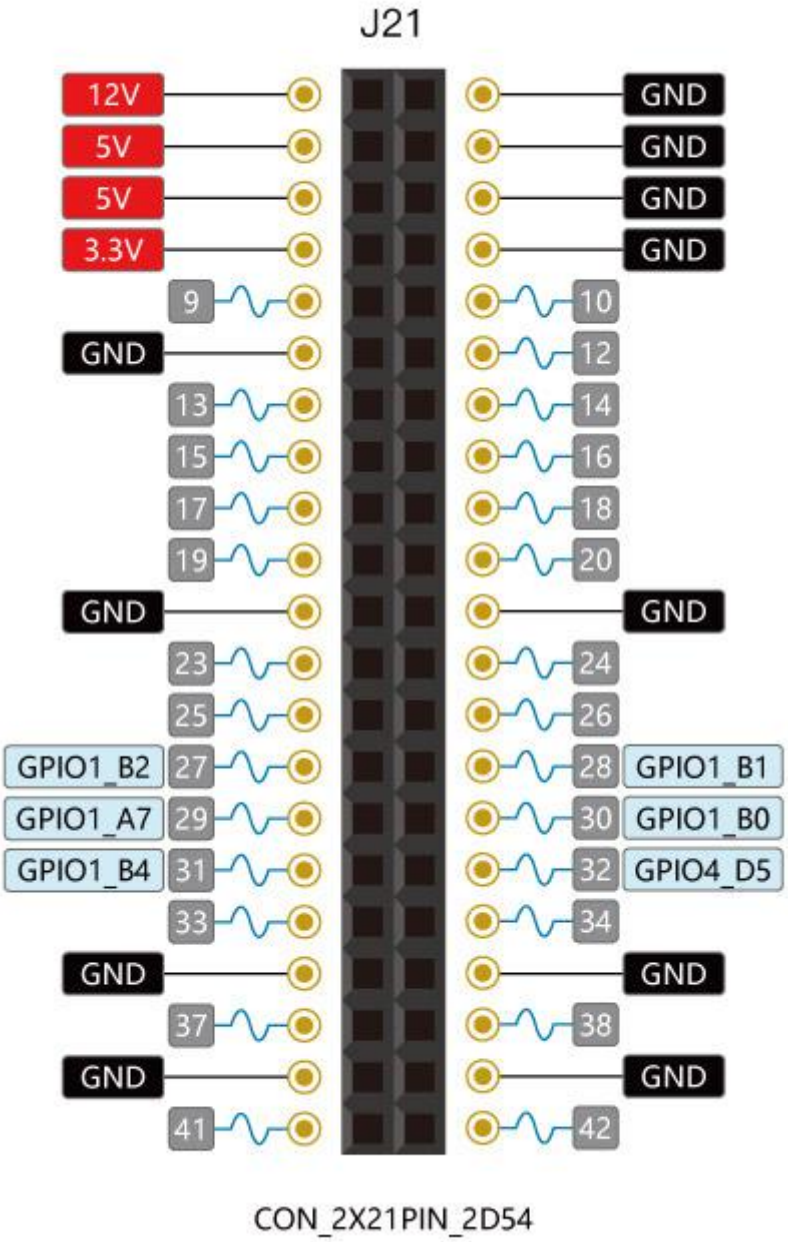
- SPI2_TXD
- CIF_CLKIN
- I2C6_SCL

每个 GPIO 口的驱动电流、上下拉和重置后的初始状态都不尽相同，详细情况请参考《RK3399 规格书》中的 "Chapter 10 GPIO" 一章。 RK3399 的 GPIO 驱动是在以下 pinctrl 文件中实现的：

```
kernel/drivers/pinctrl/pinctrl-rockchip.c
```

其核心是填充 GPIO bank 的方法和参数，并调用 gpiochip_add 注册到内核中。

Firefly-RK3399 开发板为方便用户开发使用，引出了一排通用的 GPIO 口，其对应引脚如下图：



本文以 TP_RST(GPIO0_B4)和 LCD_RST(GPIO4_D5)这两个通用 GPIO 口为例写了一份简单操作 GPIO 口的驱动，在 SDK 的路径为：

```
kernel/drivers/gpio/gpio-firefly.c
```

以下就以该驱动为例介绍 GPIO 的操作。

输入输出

首先在 DTS 文件中增加驱动的资源描述：

```
kernel/arch/arm64/boot/dts/rockchip/rk3399-firefly-demo.dtsi

    gpio_demo: gpio_demo {

        status = "okay";

        compatible = "firefly,rk3399-gpio";

        firefly-gpio = <&gpio0 12 GPIO_ACTIVE_HIGH>;            /*
GPIO0_B4 */

        firefly-irq-gpio = <&gpio4 29 IRQ_TYPE_EDGE_RISING>;  /*
GPIO4_D5 */

    };
```

这里定义了一个脚作为一般的输出输入口：

```
firefly-gpio GPIO0_B4
```

Firefly-RK3399 的 dts 对引脚的描述与 Firefly-RK3288 有所区别，`GPIO0_B4` 被描述为：`<&gpio0 12 GPIO_ACTIVE_HIGH>`，这里的 12 来源于：8+4=12，其中 8 是因为 `GPIO0_B4` 是属于 GPIO0 的 B 组，如果是 A 组的话则为 0，如果是 C 组则为 16，如果是 D 组则为 24，以此递推，而 4 是因为 B4 后面的 4。

GPIO_ACTIVE_HIGH 表示高电平有效，如果想要低电平有效，可以改为：GPIO_ACTIVE_LOW，这个属性将被驱动所读取。

然后在 probe 函数中对 DTS 所添加的资源进行解析，代码如下：

```
static int firefly_gpio_probe(struct platform_device *pdev)

{
```

```
    int ret;
    int gpio;
    enum of_gpio_flags flag;
    struct firefly_gpio_info *gpio_info;
    struct device_node *firefly_gpio_node = pdev->dev.of_node;

    printk("Firefly GPIO Test Program Probe\n");
    gpio_info  =  devm_kzalloc(&pdev->dev,sizeof(struct  firefly_gpio_info  *),
GFP_KERNEL);
    if (!gpio_info) {
        return -ENOMEM;
    }

    gpio = of_get_named_gpio_flags(firefly_gpio_node, "firefly-gpio", 0, &flag);
    if (!gpio_is_valid(gpio)) {
        printk("firefly-gpio: %d is invalid\n", gpio);
        return -ENODEV;
    }
    if (gpio_request(gpio, "firefly-gpio")) {
        printk("gpio %d request failed!\n", gpio);
        gpio_free(gpio);
        return -ENODEV;
```

```
    }
    gpio_info->firefly_gpio = gpio;
    gpio_info->gpio_enable_value = (flag == OF_GPIO_ACTIVE_LOW) ? 0:1;
    gpio_direction_output(gpio_info->firefly_gpio,
gpio_info->gpio_enable_value);
    printk("Firefly gpio putout\n");
    ......
}
```

of_get_named_gpio_flags 从设备树中读取 firefly-gpio 和 firefly-irq-gpio 的 GPIO 配置编号和标志，gpio_is_valid 判断该 GPIO 编号是否有效，gpio_request 则申请占用该 GPIO。如果初始化过程出错，需要调用 gpio_free 来释放之前申请过且成功的 GPIO 。在驱动中调用 gpio_direction_output 就可以设置输出高还是低电平，这里默认输出从 DTS 获取得到的有效电平 GPIO_ACTIVE_HIGH，即为高电平，如果驱动正常工作，可以用万用表测得对应的引脚应该为高电平。 实际中如果要读出 GPIO，需要先设置成输入模式，然后再读取值：

```
int val;
gpio_direction_input(your_gpio);
val = gpio_get_value(your_gpio);
```

下面是常用的 GPIO API 定义：

```
#include <linux/gpio.h>
#include <linux/of_gpio.h>
```

```
enum of_gpio_flags {
        OF_GPIO_ACTIVE_LOW = 0x1,
};

int of_get_named_gpio_flags(struct device_node *np, const char *propname,
                            int index, enum of_gpio_flags *flags);

int gpio_is_valid(int gpio);

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

void gpio_free(unsigned gpio);

int gpio_direction_input(int gpio);

int gpio_direction_output(int gpio, int v)
```

中断

在 Firefly 的例子程序中还包含了一个中断引脚，GPIO 口的中断使用与 GPIO 的输入输出类似，首先在 DTS 文件中增加驱动的资源描述：

```
kernel/arch/arm64/boot/dts/rockchip/rk3399-firefly-port.dtsi
  gpio {
```

```
        compatible = "firefly-gpio";
        firefly-irq-gpio = <&gpio4 29 IRQ_TYPE_EDGE_RISING>;   /*
GPIO4_D5 */
    };
```

IRQ_TYPE_EDGE_RISING 表示中断由上升沿触发，当该引脚接收到上升沿信号时可以触发中断函数。 这里还可以配置成如下：

```
IRQ_TYPE_NONE           //默认值，无定义中断触发类型
IRQ_TYPE_EDGE_RISING    //上升沿触发
IRQ_TYPE_EDGE_FALLING //下降沿触发
IRQ_TYPE_EDGE_BOTH      //上升沿和下降沿都触发
IRQ_TYPE_LEVEL_HIGH     //高电平触发
IRQ_TYPE_LEVEL_LOW      //低电平触发
```

然后在 probe 函数中对 DTS 所添加的资源进行解析，再做中断的注册申请，代码如下：

```
static int firefly_gpio_probe(struct platform_device *pdev)
{
    int ret;
    int gpio;
    enum of_gpio_flags flag;
    struct firefly_gpio_info *gpio_info;
    struct device_node *firefly_gpio_node = pdev->dev.of_node;

```

```
    ......
    gpio_info->firefly_irq_gpio = gpio;
    gpio_info->firefly_irq_mode = flag;
    gpio_info->firefly_irq = gpio_to_irq(gpio_info->firefly_irq_gpio);
    if (gpio_info->firefly_irq) {
        if (gpio_request(gpio, "firefly-irq-gpio")) {
            printk("gpio %d request failed!\n", gpio);
            gpio_free(gpio);
            return IRQ_NONE;
        }

        ret = request_irq(gpio_info->firefly_irq, firefly_gpio_irq,
                flag, "firefly-gpio", gpio_info);
        if (ret != 0)
            free_irq(gpio_info->firefly_irq, gpio_info);
            dev_err(&pdev->dev, "Failed to request IRQ: %d\n", ret);
    }
    return 0;
}
static irqreturn_t firefly_gpio_irq(int irq, void *dev_id) //中断函数
{
    printk("Enter firefly gpio irq test program!\n");
```

```
    return IRQ_HANDLED;
}
```

调用 gpio_to_irq 把 GPIO 的 PIN 值转换为相应的 IRQ 值 ,调用 gpio_request 申请占用该 IO 口，调用 request_irq 申请中断，如果失败要调用 free_irq 释放，该函数中 gpio_info-firefly_irq 是要申请的硬件中断号，firefly_gpio_irq 是中断函数，gpio_info->firefly_irq_mode 是中断处理的属性，"firefly-gpio"是设备驱动程序名称，gpio_info 是该设备的 device 结构，在注册共享中断时会用到。

复用

如何定义 GPIO 有哪些功能可以复用，在运行时又如何切换功能呢？以 I2C4 为例作简单的介绍。

查规格表可知，I2C4_SDA 与 I2C4_SCL 的功能定义如下：

在 kernel/arch/arm64/boot/dts/rockchip/rk3399.dtsi 里有：

Pad#	func0	func1
I2C4_SDA/GPIO1_B3	gpio1b3	i2c4_sda
I2C4_SCL/GPIO1_B4	gpio1b4	i2c4_scl

```
    i2c4: i2c@ff3d0000 {
        compatible = "rockchip,rk3399-i2c";
        reg = <0x0 0xff3d0000 0x0 0x1000>;
        clocks = <&pmucru SCLK_I2C4_PMU>, <&pmucru PCLK_I2C4_PMU>;
        clock-names = "i2c", "pclk";
        interrupts = <GIC_SPI 56 IRQ_TYPE_LEVEL_HIGH 0>;
        pinctrl-names = "default", "gpio";
```

```
        pinctrl-0 = <&i2c4_xfer>;
        pinctrl-1 = <&i2c4_gpio>;    //此处源码未添加
        #address-cells = <1>;
        #size-cells = <0>;
        status = "disabled";
    };
```

此处，跟复用控制相关的是 pinctrl- 开头的属性：

- pinctrl-names 定义了状态名称列表： default (i2c 功能) 和 gpio 两种状态。
- pinctrl-0 定义了状态 0 (即 default) 时需要设置的 pinctrl: &i2c4_xfer
- pinctrl-1 定义了状态 1 (即 gpio)时需要设置的 pinctrl: &i2c4_gpio

这些 pinctrl 在 kernel/arch/arm64/boot/dts/rockchip/rk3399.dtsi 中这样定义：

```
    ......
    pinctrl: pinctrl {
        compatible = "rockchip,rk3399-pinctrl";
        rockchip,grf = <&grf>;
        rockchip,pmu = <&pmugrf>;
        #address-cells = <0x2>;
        #size-cells = <0x2>;
        ranges;
        i2c4 {
            i2c4_xfer: i2c4-xfer {
                rockchip,pins =
```

```
                    <1 12 RK_FUNC_1 &pcfg_pull_none>,
                    <1 11 RK_FUNC_1 &pcfg_pull_none>;
            };
            i2c4_gpio: i2c4-gpio {
                rockchip,pins =
                    <1 12 RK_FUNC_GPIO &pcfg_pull_none>,
                    <1 11 RK_FUNC_GPIO &pcfg_pull_none>;
            };

        };
```

RK_FUNC_1,RK_FUNC_GPIO 的定义在 kernel/include/dt-bindings/pinctrl/rk.h 中：

```
#define RK_FUNC_GPIO    0
#define RK_FUNC_1   1
#define RK_FUNC_2   2
#define RK_FUNC_3   3
#define RK_FUNC_4   4
#define RK_FUNC_5   5
#define RK_FUNC_6   6
#define RK_FUNC_7   7
```

另外，像"1 11"，"1 12"这样的值是有编码规则的，编码方式与上一小节"输入输出"描述的一样，"1 11"代表 GPIO1_B3，"1 12"代表 GPIO1_B4。

在复用时，如果选择了 "default" （即 i2c 功能),系统会应用 i2c4_xfer 这个 pinctrl，最终将 GPIO1_B3 和 GPIO1_B4 两个针脚切换成对应的 i2c 功能；而如果选择了 "gpio" ，系统会应用 i2c4_gpio 这个 pinctrl，将 GPIO1_B3 和 GPIO1_B4 两个针脚还原为 GPIO 功能。

我们看看 i2c 的驱动程序 kernel/drivers/i2c/busses/i2c-rockchip.c 是如何切换复用功能的：

```
static int rockchip_i2c_probe(struct platform_device *pdev)
{
    struct rockchip_i2c *i2c = NULL;
    struct resource *res;
    struct device_node *np = pdev->dev.of_node;
    int ret;// ...
        i2c->sda_gpio = of_get_gpio(np, 0);
        if (!gpio_is_valid(i2c->sda_gpio)) {
                dev_err(&pdev->dev, "sda gpio is invalid\n");
                return -EINVAL;
        }
        ret = devm_gpio_request(&pdev->dev, i2c->sda_gpio,
dev_name(&i2c->adap.dev));
        if (ret) {
                dev_err(&pdev->dev, "failed to request sda gpio\n");
                return ret;
```

```
        }
        i2c->scl_gpio = of_get_gpio(np, 1);
        if (!gpio_is_valid(i2c->scl_gpio)) {
                dev_err(&pdev->dev, "scl gpio is invalid\n");
                return -EINVAL;
        }
        ret = devm_gpio_request(&pdev->dev, i2c->scl_gpio,
dev_name(&i2c->adap.dev));
        if (ret) {
                dev_err(&pdev->dev, "failed to request scl gpio\n");
                return ret;
        }
        i2c->gpio_state = pinctrl_lookup_state(i2c->dev->pins->p, "gpio");
        if (IS_ERR(i2c->gpio_state)) {
                dev_err(&pdev->dev, "no gpio pinctrl state\n");
                return PTR_ERR(i2c->gpio_state);
        }
        pinctrl_select_state(i2c->dev->pins->p, i2c->gpio_state);
        gpio_direction_input(i2c->sda_gpio);
        gpio_direction_input(i2c->scl_gpio);
        pinctrl_select_state(i2c->dev->pins->p,
i2c->dev->pins->default_state);// ...}
```

首先是调用 of_get_gpio 取出设备树中 i2c4 结点的 gpios 属于所定义的两个 gpio:

```
gpios = <&gpio1 GPIO_B3 GPIO_ACTIVE_LOW>, <&gpio1 GPIO_B4 GPIO_ACTIVE_LOW>;
```

然后是调用 devm_gpio_request 来申请 gpio，接着是调用 pinctrl_lookup_state 来查找 “gpio” 状态，而默认状态 "default" 已经由框架保存到 i2c->dev-pins->default_state 中了。

最后调用 pinctrl_select_state 来选择是 "default" 还是 "gpio" 功能。

下面是常用的复用 API 定义：

```
#include <linux/pinctrl/consumer.h>

struct device {//...#ifdef CONFIG_PINCTRL

struct dev_pin_info *pins;#endif//...};

struct dev_pin_info {

struct pinctrl *p;

struct pinctrl_state *default_state;#ifdef CONFIG_PM

struct pinctrl_state *sleep_state;

struct pinctrl_state *idle_state;#endif};

struct pinctrl_state * pinctrl_lookup_state(struct pinctrl *p, const char *name);

int pinctrl_select_state(struct pinctrl *p, struct pinctrl_state *s);
```

IO-Domain

在复杂的片上系统（SOC）中，设计者一般会将系统的供电分为多个独立的 block，这称作电源域（Power Domain），这样做有很多好处，例如：

1. 在 IO-Domain 的 DTS 节点统一配置电压域，不需要每个驱动都去配置一次，便于管理；

2. 依照的是 Upstream 的做法，以后如果需要 Upstream 比较方便；

3. IO-Domain 的驱动支持运行过程中动态调整电压域，例如 PMIC 的某个 Regulator 可以 1.8v 和 3.3v 的动态切换，一旦 Regulator 电压发生改变，会通知 IO-Domain 驱动去重新设置电压域。

Firefly-RK3399 原理图上的 Power Domain Map 表以及配置如下表所示：

RK3399 Power Domain Map

Part Port	Domain	Pin name in datasheet	I/O type	Power supply	Power source
Part C	PMUIO1	pmuio1_gpio0ab	1.8V only	VCC1V8_PMUPLL	RK808 VLDO3
Part E	PMUIO2	pmuio2_gpio1abcd	1.8V(Default) 3.0V	VCC_1V5 VCC_3V0	RK808 VLDO6 RK808 VLDO8
Part I	APIO1	gmac_gpio3abc	3.3V only	VCC3V3_S3	RK808 VSW1
Part L	APIO2	bt656_gpio2ab	1.8V(Default) 3.0V	VCC1V8_DVP	RK808 VLDO1
Part G	APIO3	wifi/bt_gpio2cd	1.8V only	VCC1V8_S3	RK808 Buck4
Part K	APIO4	gpio1833_gpio4cd	1.8V 3.0V(Default)	VCC_1V5 VCC3V0_IO	RK808 VLDO6 RK808 VLDO8
Part J	APIO5	audio_gpio3d_gpio4a	1.8V(Default) 3.0V	VCCA1V8_CODEC	RK808 VLDO7
Part F	SDMMC0	sdmmc_gpio4b	1.8V 3.0V(Default)	VCC_SDIO	RK808 VLDO4

通过 RK3399 SDK 的原理图可以看到 bt656-supply 的电压域连接的是 vcc18_dvp, vcc_io 是从 PMIC RK808 的 VLDO1 出来的；

在 DTS 里面可以找到 vcc1v8_dvp， 将 bt656-supply = <&vcc18_dvp>。

其他路的配置也类似，需要注意的是如果这里是其他 PMIC，所用的 Regulator 也不一样，具体以实际电路情况为标准。

调试方法

IO 指令

GPIO 调试有一个很好用的工具，那就是 IO 指令，Firefly-RK3399 的 Android 系统默认已经内置了 IO 指令，使用 IO 指令可以实时读取或写入每个 IO 口的状态，这里简单介绍 IO 指令的使用。 首先查看 io 指令的帮助：

```
#io --help
Unknown option: ?
Raw memory i/o utility - $Revision: 1.5 $

io -v -1|2|4 -r|w [-l <len>] [-f <file>] <addr> [<value>]

    -v          Verbose, asks for confirmation
    -1|2|4      Sets memory access size in bytes (default byte)
    -l <len>    Length in bytes of area to access (defaults to
                one access, or whole file length)
    -r|w        Read from or Write to memory (default read)
    -f <file>   File to write on memory read, or
                to read on memory write
    <addr>      The memory address to access
    <val>       The value to write (implies -w)

Examples:
    io 0x1000                   Reads one byte from 0x1000
    io 0x1000 0x12              Writes 0x12 to location 0x1000
```

```
io -2 -l 8 0x1000              Reads 8 words from 0x1000
io -r -f dmp -l 100 200        Reads 100 bytes from addr 200 to
file
io -w -f img 0x10000           Writes the whole of file to memory

Note access size (-1|2|4) does not apply to file based accesses.
```

从帮助上可以看出，如果要读或者写一个寄存器，可以用：

```
io -4 -r 0x1000 //读从 0x1000 起的 4 位寄存器的值
io -4 -w 0x1000 //写从 0x1000 起的 4 位寄存器的值
```

使用示例：

- 查看 GPIO1_B3 引脚的复用情况

1. 从主控的 datasheet 查到 GPIO1 对应寄存器基地址为：0xff320000

2. 从主控的 datasheet 查到 GPIO1B_IOMUX 的偏移量为：0x00014

3. GPIO1_B3 的 iomux 寄存器地址为：基址(Operational Base) + 偏移量(offset)=0xff320000+0x00014=0xff320014

4. 用以下指令查看 GPIO1_B3 的复用情况：

```
# io -4 -r 0xff320014
ff320014:  00000816a
```

5. 从 datasheet 查到[7:6]：

```
gpio1b3_sel
```

```
GPIO1B[3] iomux select
2'b00: gpio
2'b01: i2c4sensor_sda
2'b10: reserved
2'b11: reserved
```

因此可以确定该 GPIO 被复用为 i2c4sensor_sda。

6. 如果想复用为 GPIO,可以使用以下指令设置：

```
# io -4 -w 0xff320014 0x00000812a
```

GPIO 调试接口

Debugfs 文件系统目的是为开发人员提供更多内核数据，方便调试。 这里 GPIO 的调试也可以用 Debugfs 文件系统，获得更多的内核信息。 GPIO 在 Debugfs 文件系统中的接口为 /sys/kernel/debug/gpio，可以这样读取该接口的信息：

```
# cat /sys/kernel/debug/gpio
GPIOs 0-31, platform/pinctrl, gpio0:
gpio-2   (                    |vcc3v3_3g            ) out hi
gpio-4   (                    |bt_default_wake_host) in  lo
gpio-5   (                    |power                ) in  hi
gpio-9   (                    |bt_default_reset     ) out lo
gpio-10  (                    |reset                ) out lo
gpio-13  (                    |?                    ) out lo
```

```
GPIOs 32-63, platform/pinctrl, gpio1:
gpio-32  (                    |vcc5v0_host           ) out hi
gpio-34  (                    |int-n                 ) in  hi
gpio-35  (                    |vbus-5v               ) out lo
gpio-45  (                    |pmic-hold-gpio        ) out hi
gpio-49  (                    |vcc3v3_pcie           ) out hi
gpio-54  (                    |mpu6500               ) out hi
gpio-56  (                    |pmic-stby-gpio        ) out hi

GPIOs 64-95, platform/pinctrl, gpio2:
gpio-83  (                    |bt_default_rts        ) in  hi
gpio-90  (                    |bt_default_wake       ) in  lo
gpio-91  (                    |?                     ) out hi

GPIOs 96-127, platform/pinctrl, gpio3:
gpio-111 (                    |mdio-reset            ) out hi

GPIOs 128-159, platform/pinctrl, gpio4:
gpio-149 (                    |hp-con-gpio           ) out lo
```

从读取到的信息中可以知道，内核把 GPIO 当前的状态都列出来了，以 GPIO0 组为例，gpio-2(GPIO0_A2)作为 3G 模块的电源控制脚(vcc3v3_3g)，输出高电平(out hi)。

FAQs

Q1：如何将 PIN 的 MUX 值切换为一般的 GPIO？

A1: 当使用 GPIO request 时候，会将该 PIN 的 MUX 值强制切换为 GPIO，所以使用该 pin 脚为 GPIO 功能的时候确保该 pin 脚没有被其他模块所使用。

Q2：为什么我用 IO 指令读出来的值都是 0x00000000？

A2: 如果用 IO 命令读某个 GPIO 的寄存器，读出来的值异常,如 0x00000000 或 0xffffffff 等，请确认该 GPIO 的 CLK 是不是被关了，GPIO 的 CLK 是由 CRU 控制，可以通过读取 datasheet 下面 CRU_CLKGATE_CON* 寄存器来查到 CLK 是否开启，如果没有开启可以用 io 命令设置对应的寄存器，从而打开对应的 CLK，打开 CLK 之后应该就可以读到正确的寄存器值了。

Q3：测量到 PIN 脚的电压不对应该怎么查？

A3: 测量该 PIN 脚的电压不对时，如果排除了外部因素，可以确认下该 pin 所在的 io 电压源是否正确，以及 IO-Domain 配置是否正确。

Q4：gpio_set_value()与 gpio_direction_output()有什么区别？

A4: 如果使用该 GPIO 时，不会动态的切换输入输出，建议在开始时就设置好 GPIO 输出方向 ,后面拉高拉低时使用 gpio_set_value()接口 ,而不建议使用 gpio_direction_output(), 因为 gpio_direction_output 接口里面有 mutex 锁，对中断上下文调用会有错误异常，且相比 gpio_set_value，gpio_direction_output 所做事情更多，浪费。