# 整体框架

gpio系统的框架可以理解为三层，中间层为linux内核提供的核心实现gpiolib，向下定义了gpio\_chip结构用于描述gpio控制器，向上为设备驱动提供了统一的接口来操作gpio，底层根据SoC的gpio控制器不同进行gpio\_chip的填充与注册，上层就是gpio的使用，使用gpio的设备繁多，包括键盘、按键、led等等。整体框图如下：

gpiolib

gpio\_chip

键盘

键盘

键盘

# gpio chip

## gpio\_chip结构

gpio\_chip结构描述一个GPIO控制器，我们需要根据控制器的属性进行成员的填充。结构定义如下：

struct gpio\_chip {

int (\*request)(struct gpio\_chip \*chip, unsigned offset);//请求

void (\*free)(struct gpio\_chip \*chip,unsigned offset);//释放

Int (\*get\_direction)(struct gpio\_chip \*chip, unsigned offset);//获取方向

int (\*direction\_input)(struct gpio\_chip \*chip, unsigned offset);//设置为输入

int (\*direction\_output)(struct gpio\_chip \*chip, unsigned offset, int value);//设置为输出

int (\*get)(struct gpio\_chip \*chip, unsigned offset);//获取电平

void (\*set)(struct gpio\_chip \*chip, unsigned offset, int value);//设置电平

int (\*set\_debounce)(struct gpio\_chip \*chip, unsigned offset, unsigned debounce);//设置去抖

int (\*to\_irq)(struct gpio\_chip \*chip, unsigned offset);//获取中断号

int base;//首个gpio开始号

u16 ngpio;//gpio个数

}

## gpio\_chip注册

int gpiochip\_add(struct gpio\_chip \*chip)

gpio\_chip的注册

void gpiochip\_remove(struct gpio\_chip \*chip)

gpio\_chip的注销

## 示例说明

示例fake\_gpiochip只是演示gpio\_chip的注册。insmod fake\_gpiochip.ko后/sys/class/gpio/下会多出一个gpiochip504目录，注意并不一定是504，因为我在注册的时候base赋值了-1。

### 设备树

fake\_gpiochip: fake\_gpiochip@0 {

compatible = "fake-gpio-chip";

};

### gpio sysfs使用

echo 504 > export

/sys/class/gpio/目录中有一个用于写export的文件，向其中写入一个数字表示建立一个GPIO端口的子目录，执行此操作，会出现gpio504目录，这个目录就包含了对504号gpio的操作，如下：

direction：可读写，使用"in"和"out"两个值表示输入和输出。

value：可读写，0代表低电平，1（或其他非0）代表高电平。

echo in > direction

echo out > direction

echo 1 > value

cat direction

cat value

我们执行上面这些操作可以通过打印信息看出都能正确调用我们在注册gpio\_chip时填充的那些fake\_gpiochip\_\*。

# gpio使用

## 接口说明

目前的gpio子系统存在2套接口：

* legacy API：integer-based接口，形式为gpio\_xxx()，例如 void gpio\_set\_value(unsigned gpio, int value)，这套接口已弃用，如果你去看接口实现内部也是使用了gpiod\_xxx()，所以这套接口就不要用了。
* 推荐API：descriptor-based接口，形式为gpiod\_xxx()，例如 void gpiod\_set\_value(struct gpio\_desc \*desc, int value)，我们都要试用这套接口，下面也只对这套接口做出介绍。

## 函数说明

所有的接口定义都在include/linux/gpio/consumer.h中。常用的如下：

[devm]\_gpiod\_get\_xxx()

[devm]\_gpiod\_put\_xxx()

获得/释放一个/一组 gpio

gpiod\_direction\_input()

gpiod\_direction\_output()

gpiod\_get\_direction()

设置/查询输入或者输出

gpiod\_get\_value()

gpiod\_set\_value()

gpiod\_get\_value\_cansleep()

gpiod\_set\_value\_cansleep()

gpiod\_get\_array\_value()

gpiod\_set\_array\_value()

读写一个/一组gpio

gpiod\_to\_irq()

获得 gpio 对应的中断号

注意事项：

* 以\_cansleep为后缀的函数是可能会睡眠的 API，不可以在 hard (non-threaded) IRQ handlers 中使用;
* gpiod\_get\_value() 返回的是硬件上的电平值;
* gpiod\_set\_value() 设置的值是逻辑值而非电平值，1 表示active，0 表示inactive，由设备树里的gpio mappings 里的GPIO\_ACTIVE\_XXX 来决定最终的物理电平，总结如下：

|  |  |  |
| --- | --- | --- |
| Function (example) | active-low property | physical line |
| gpiod\_set\_raw\_value(desc, 0); | don't care | low |
| gpiod\_set\_raw\_value(desc, 1); | don't care | high |
| gpiod\_set\_value(desc, 0); | default (active-high) | low |
| gpiod\_set\_value(desc, 1); | default (active-high) | high |
| gpiod\_set\_value(desc, 0); | default (active-high) | high |
| gpiod\_set\_value(desc, 1); | default (active-high) | low |
| gpiod\_set\_value(desc, 0); | open drain | low |
| gpiod\_set\_value(desc, 1); | open drain | high impedance |
| gpiod\_set\_value(desc, 0); | open source | high impedance |
| gpiod\_set\_value(desc, 1); | open source | high |

* 应避免使用set\_raw/get\_raw函数，因为驱动应该更关注逻辑值而不是物理值。

## 示例说明

示例只是简单的获取了一个gpio，然后将这个gpio根据module的输入值进行控制，insmod gpio\_consumer.ko value=1。设备树中目前设置的是GPIO\_ACTIVE\_HIGH，也就是value=1时电平为高，value=0时电平为低，如果设置为GPIO\_ACTIVE\_LOW，则电平状态就会反过来。

### 设备树

&am33xx\_pinmux {//引脚复用

consumer\_gpio: consumer\_gpio {

pinctrl-single,pins = <

AM33XX\_IOPAD(0x814, PIN\_OUTPUT | MUX\_MODE7) /\* (V8) \*/

>;

};

};

gpio\_consumer: gpio\_consumer@0 {

compatible = "gpio-consumer";

gpios = <&gpio1 5 GPIO\_ACTIVE\_HIGH>;

pinctrl-names = "default";

pinctrl-0 = <&consumer\_gpio>;

};