# Linux 内核函数

目录

[Linux 内核函数 1](#_Toc5089889)

[1 devm\_gpiod\_get\_optional() 1](#_Toc5089890)

[2 container\_of() 2](#_Toc5089891)

[3 cancel\_delayed\_work\_sync() 2](#_Toc5089892)

[4 request\_threaded\_irq() 3](#_Toc5089893)

## 1 devm\_gpiod\_get\_optional()

struct gpio\_desc \*\_\_must\_check devm\_gpiod\_get\_optional(struct device \*dev,

const char \*con\_id,

enum gpiod\_flags flags)

在看显示模块的代码时看到一个函数devm\_gpiod\_get\_optional(), 之前没接触过。

调用过程如下：

devm\_gpiod\_get\_optional

　devm\_gpiod\_get\_index\_optional //index为0

　　devm\_gpiod\_get\_index

　　　gpiod\_get\_index

可以看到devm\_gpiod\_get\_optional只是对gpiod\_get\_index的包装而已，并且index为0。index参数后面会提。

函数参数：

struct gpio\_desc \*\_\_must\_check devm\_gpiod\_get\_index(struct device \*dev,

const char \*con\_id,

unsigned int idx,

enum gpiod\_flags flags);

重点关注第二个和第三个参数。

gpiod\_get\_index()到底用来干什么?

答：gpiod\_get\_index()本质上和gpio\_request()是一个功能，是申请gpio的，只是它是从device tree去查找， 因此看到第二个参数”con\_id”是字符串类型，也就是gpio的名字。

例如在显示驱动看到的去查找名字为”enable”的gpio

panel-simple.c:

panel->enable\_gpio = devm\_gpiod\_get\_optional(dev, " enable-gpios ", 0);

在使用mipi屏幕的主dts就有enable pin的定义

rk3399-firefly-mipi.dts:

enable-gpios = <&gpio1 1 GPIO\_ACTIVE\_HIGH>;

那么index又有什么用呢？

内核文档有个例子，比如gpio如下定义：

led-gpios = <&gpio 15 GPIO\_ACTIVE\_HIGH>, /\* red \*/

<&gpio 16 GPIO\_ACTIVE\_HIGH>, /\* green \*/

<&gpio 17 GPIO\_ACTIVE\_HIGH>; /\* blue \*/

如果index是0,那么对应的就是gpio 15; 　如果index是1,那么对应就是gpio 16，以此类推。

## 2 container\_of()

说明：”由结构体成员地址找到结构体地址。

#define container\_of(ptr, type, member) ({ \

const typeof(((type \*)0)->member) \* \_\_mptr = (ptr); \

(type \*)((char \*)\_\_mptr - offsetof(type, member)); })

Ptr为该成员变量的地址，第二个为结构体，第三个为成员变量

例子1：成员为结构体变量

struct delayed\_work {

struct work\_struct work;

struct timer\_list timer;

};

void mwifiex\_dfs\_cac\_work\_queue(struct work\_struct \*work)

{

struct delayed\_work \*delayed\_work =

container\_of(work, struct delayed\_work, work);

}

例子2：成员变量为指针

client->addr

struct ft6236\_data {

struct i2c\_client \*client;//client地址信息在内核启动时就已经解析

struct input\_dev \*input;

};

static int ft6236\_remove(struct i2c\_client \*client)

{

struct ft6236\_data \*ft6236=container\_of(&client,struct ft6236\_data,addr);

input\_unregister\_device(ft6236->input);

return 0;

}

## 3 cancel\_delayed\_work\_sync()

cancel\_delayed\_work\_sync(&spilcd->delayed\_worker); //完成本次任务后再取消工作队列

cancel\_delayed\_work (&spilcd->delayed\_worker);//立即取消工作队列

struct spilcd{

struct spi\_device \*spidev;//spi设备

struct fb\_info \*fbi;//帧缓冲区

struct spilcd\_data \*data;//从设备树获取的数据

struct delayed\_work delayed\_worker;//延时工作队列

};

## 4 request\_threaded\_irq()

-中断线程化函数

int request\_threaded\_irq(unsigned int irq, irq\_handler\_t handler, irq\_handler\_t thread\_fn,

                                                      unsigned long irqflags,const char \*devname, void \*dev\_id);

分析request\_threaded\_irq()函数中的各个形参：  
1>：irq:表示申请的中断号。  
2>：handler:表示中断服务例程。如果线程化，此处可以传NULL。  
3.> thread\_fn：中断线程化函数。如果此处传递的是NULL。NULL表示没有中断线程化。  
    在 Linux 中，中断具有最高的优先级。不论在任何时刻，只要产生中断事件，内核将立即执行相应的中断处理程序，等到所有挂起的中断和软中断处理完毕后才能执行正常的任务，因此有可能造成实时任务得不到及时的处理。中断线程化之后，中断将作为内核线程运行而且被赋予不同的实时优先级，实时任务可以有比中断线程更高的优先级。这样，具有最高优先级的实时任务就能得到优先处理，即使在严重负载下仍有实时性保证。but,并不是所有的中断都可以被线程化，比如时钟中断，主要用来维护系统时间以及定时器等，其中定时器是操作系统的脉搏，一旦被线程化，就有可能被挂起，这样后果将不堪设想，所以不应当被线程化。   
4>.irqflags:表示中断标志位。  
5>.devname:表示请求中断的设备的名称。

6>.dev\_id: 对应于request\_irq()函数中所传递的第五个参数，可取任意值，但必须唯一能够代表发出中断请求的设备，通常取描述该设备的结构体。 在共享中断时可以用来分辨是哪个设备产生了中断。

devm\_request\_threaded\_irq(dev, client->irq, NULL,

ft6236\_interrupt, IRQF\_ONESHOT|IRQ\_TYPE\_EDGE\_RISING,

client->name, ft6236);

参数解析：

第三个参数NULL表示不使用硬中断函数，硬中断函数用来处理紧急任务。第四个参数ft6236\_interrupt是线程化中断函数，表示如果产生中断，调用该函数。

IRQF\_ONESHOT ： Interrupt is not reenabled after the hardirq handler finished.

\* Used by threaded interrupts which need to keep the

\* irq line disabled until the threaded handler has been run.

意思就是硬中断函数结束后不会重新开启中断，直到线程中断函数结束后才会允许再次中断。

## 5 devm\_input\_allocate\_device()

分配一个输入设备，标识设备内存属于dev,当卸载dev时自动释放内存。

static int ft6236\_probe(struct i2c\_client \*client,const struct i2c\_device\_id \*id)

{

struct device \*dev = &client->dev;

input = devm\_input\_allocate\_device(dev);//分配一个输入设备,参数表示属于哪一个设备，一般是probe函数传入的设备指向的dev

}

## 6 gpio\_to\_irq()

通过io引脚编号来找到对应的中断编号

if (of\_find\_property(np, "interrupts", NULL)) {

ft6236->irq\_pin = of\_get\_named\_gpio\_flags(np, "interrupts", 0, &irq\_flags);

}

gpio\_request(ft6236->irq\_pin, "ft6236");

client->irq = gpio\_to\_irq(ft6236->irq\_pin);//

设备树中配置

ft6236:ft6236@0x38{

compatible = "ft6236";

status = "okay";

reg = <0x38>;

interrupts = <&gpio0 RK\_PC2 IRQ\_TYPE\_EDGE\_RISING>;

};

## 7 ERR\_PTR

static struct infrared\_data \*infrared\_probe\_dt(struct device \*dev)

{

struct device\_node \*np = dev->of\_node;

if (!np) {

return ERR\_PTR(-EINVAL);

}

}

if(!pdata) {

pdata=infrared\_probe\_dt(dev);

if (IS\_ERR(pdata)){

ret = PTR\_ERR(pdata);

goto error;

}

}

return ERR\_PTR (-EINVAL); //该函数将错误码强制转换为指针类型

Ret=PTR\_ERR(pdata); //将指针转换为整型数，此时ret=-EINVAL