# 驱动架构

...

用户空间

内

核

空

间

硬件

/sys/class/leds/xxx

leds-xxx.c

leds-gpio.c

leds-xxx.c

led-core.c

led-class.c

led-trigger.c

ledtrig-gpio.c

ledtrig-disk.c

ledtrig-xxx.c

核心

驱动

* 用户空间可以利用内核提供的接口控制led的亮度、trigger策略等。
* 核心层的led-class.c提供了led设备的注册以及用户空间操作led的接口，led-core.c则完成通过定时器控制闪烁以及设置亮度等接口，led-triggers.c则维护所有的触发器，内核提供了很多的触发策略来控制led的亮灭，都以ledtrig-xxx.c来命名。
* 驱动层则是具体led器件的驱动，使用led器件可以节省引脚，当然也可以不使用led器件，此时可以使用leds-gpio.c通过gpio来控制led。
* 硬件层就是led器件，驱动层的leds-gpio.c则对应的就是gpio控制器。

# led的注册

内核使用led\_classdev结构来描述一个led:

struct led\_classdev {

const char \*name; //led名称

enum led\_brightness brightness;//亮度值

enum led\_brightness max\_brightness;//最大亮度值

int flags;//标志，低16位用于表示状态，高16位用于表示控制信息

//设置led亮度，此函数不可睡眠，若led仅可亮灭，则0-灭 非0-亮

void (\*brightness\_set)(struct led\_classdev \*led\_cdev, enum led\_brightness brightness);

//设置led亮度，此函数可睡眠，所以可能造成阻塞

int (\*brightness\_set\_blocking)(struct led\_classdev \*led\_cdev, enum led\_brightness brightness);

//获取当前亮度

enum led\_brightness (\*brightness\_get)(struct led\_classdev \*led\_cdev);

//如果硬件支持闪烁功能可通过此函数设置，如果不设置此函数内核使用定时器进行闪烁，使用定时器闪烁时如果使用brightness\_set()将led关闭，闪烁功能也会被关闭，使用定时器闪烁时如果delay\_on和delay\_off都为0，则默认分别使用500ms也就是默认闪烁频率1Hz，如果只有delay\_on为0，则将led关闭，如果只有delay\_off为0，则led常亮。

int (\*blink\_set)(struct led\_classdev \*led\_cdev, unsigned long \*delay\_on, unsigned long \*delay\_off);

struct device \*dev;//该led\_classdev的设备

const struct attribute\_group \*\*groups;//自定义的sysfs属性文件

struct list\_head node;//会被连接到全局的leds\_list上

const char \*default\_trigger;//trigger类型，可以为timer/gpio等，不设置则为none

unsigned long blink\_delay\_on, blink\_delay\_off;//闪烁时，on和off的时间

struct timer\_list blink\_timer;//闪烁定时器

int blink\_brightness;//闪烁亮度

//有的led分torch和flash两种模式，两者电流不同，如摄像头led的torch模式就是用于录像电流较小，flash模式则用于拍照闪光电流较大，此函数注册flash led时才用到，具体内容参考led-class-flash.txt

void (\*flash\_resume)(struct led\_classdev \*led\_cdev);

struct work\_struct set\_brightness\_work;//设置led亮度时用到的工作队列

int delayed\_set\_value;//阻塞方式设置亮度时使用工作队列方式设置，将亮度值暂存在此成员

#ifdef CONFIG\_LEDS\_TRIGGERS

/\* Protects the trigger data below \*/

struct rw\_semaphore trigger\_lock;

struct led\_trigger \*trigger; //led使用的trigger

struct list\_head trig\_list;//链接到所用trigger的led\_cdevs上

void \*trigger\_data;//trigger私有数据

/\* true if activated - deactivate routine uses it to do cleanup \*/

bool activated;//trigger启动标志

#endif

struct mutex led\_access;

};

结构初始化完成后可向内核注册led设备：

int devm\_led\_classdev\_register(struct device \*parent, struct led\_classdev \*led\_cdev);

如果使用led\_classdev\_register()注册，则需要使用led\_classdev\_unregister()注销。

# led trigger的实现

内核使用led\_trigger结构描述一种触发方式：

struct led\_trigger {

const char \*name;//触发器的名称

void (\*activate)(struct led\_classdev \*led\_cdev);//激活触发器回调，led设置触发器时执行

void (\*deactivate)(struct led\_classdev \*led\_cdev);//关闭触发器回调，led移除或更改触发器时执行

rwlock\_t leddev\_list\_lock;

struct list\_head led\_cdevs;//受此触发器控制的led

struct list\_head next\_trig;//链接到下一个触发器

};

结构初始化完毕后向内核中注册:

int led\_trigger\_register(struct led\_trigger \*trig);

使用如下函数进行注销：

void led\_trigger\_unregister(struct led\_trigger \*trig);

如果无需ativate和deactivate，可通过以下方式注册，指定触发器名称即可注册，并将触发器通过\*\*tp返回：

void led\_trigger\_register\_simple(const char \*name, struct led\_trigger \*\*tp);

对应的注销函数为：

void led\_trigger\_unregister\_simple(struct led\_trigger \*trig);

内核已经提供了很多种的trigger，这些trigger都以ledtrig-xxx.c来命名，下面介绍这些trigger的用法与实现。

## default-on

static void defon\_trig\_activate(struct led\_classdev \*led\_cdev)

{

led\_set\_brightness\_nosleep(led\_cdev, led\_cdev->max\_brightness);

}

static struct led\_trigger defon\_led\_trigger = {

.name = "default-on",

.activate = defon\_trig\_activate,

};

default-on用于将led亮度设为最大。

## timer

static void timer\_trig\_activate(struct led\_classdev \*led\_cdev)

{

//创建delay\_on和delay\_off文件

rc = device\_create\_file(led\_cdev->dev, &dev\_attr\_delay\_on);

rc = device\_create\_file(led\_cdev->dev, &dev\_attr\_delay\_off);

//设置闪烁，如果led\_classdev提供blink\_set方法，则使用此方法控制闪烁，否则使用定时器方式控制。

led\_blink\_set(led\_cdev, &led\_cdev->blink\_delay\_on, &led\_cdev->blink\_delay\_off);

led\_cdev->activated = true;

}

static void timer\_trig\_deactivate(struct led\_classdev \*led\_cdev)

{

if (led\_cdev->activated) {

//删除delay\_on和delay\_off文件

device\_remove\_file(led\_cdev->dev, &dev\_attr\_delay\_on);

device\_remove\_file(led\_cdev->dev, &dev\_attr\_delay\_off);

led\_cdev->activated = false;

}

//关闭led，也就是说将trigger从timer替换为其他方式时先关闭led

led\_set\_brightness(led\_cdev, LED\_OFF);

}

static struct led\_trigger timer\_led\_trigger = {

.name = "timer",

.activate = timer\_trig\_activate,

.deactivate = timer\_trig\_deactivate,

};

如果将某个led的trigger设置为timer，则目录下就会多出delay\_on和delay\_off两个文件，用于设置高低电平的时间(ms)来控制闪烁。

## heartbeat

static void led\_heartbeat\_function(unsigned long data)

{

switch (heartbeat\_data->phase) {

case 0:

//根据负载计算心跳周期

heartbeat\_data->period = 300 + (6720 << FSHIFT) / (5 \* avenrun[0] + (7 << FSHIFT));

heartbeat\_data->period = msecs\_to\_jiffies(heartbeat\_data->period);

delay = msecs\_to\_jiffies(70);

heartbeat\_data->phase++;

if (!heartbeat\_data->invert)

brightness = led\_cdev->max\_brightness;

break;

case 1:

delay = heartbeat\_data->period / 4 - msecs\_to\_jiffies(70);

heartbeat\_data->phase++;

if (heartbeat\_data->invert)

brightness = led\_cdev->max\_brightness;

break;

case 2:

delay = msecs\_to\_jiffies(70);

heartbeat\_data->phase++;

if (!heartbeat\_data->invert)

brightness = led\_cdev->max\_brightness;

break;

default:

delay = heartbeat\_data->period - heartbeat\_data->period / 4 -

msecs\_to\_jiffies(70);

heartbeat\_data->phase = 0;

if (heartbeat\_data->invert)

brightness = led\_cdev->max\_brightness;

break;

}

led\_set\_brightness\_nosleep(led\_cdev, brightness);

mod\_timer(&heartbeat\_data->timer, jiffies + delay);

}

static void heartbeat\_trig\_activate(struct led\_classdev \*led\_cdev)

{

heartbeat\_data = kzalloc(sizeof(\*heartbeat\_data), GFP\_KERNEL);

led\_cdev->trigger\_data = heartbeat\_data;

//创建invert文件

rc = device\_create\_file(led\_cdev->dev, &dev\_attr\_invert);

//初始化heartbeat定时器

setup\_timer(&heartbeat\_data->timer, led\_heartbeat\_function, (unsigned long) led\_cdev);

heartbeat\_data->phase = 0;

//开始执行定时器

led\_heartbeat\_function(heartbeat\_data->timer.data);

led\_cdev->activated = true;

}

static void heartbeat\_trig\_deactivate(struct led\_classdev \*led\_cdev)

{

if (led\_cdev->activated) {

del\_timer\_sync(&heartbeat\_data->timer);

device\_remove\_file(led\_cdev->dev, &dev\_attr\_invert);

kfree(heartbeat\_data);

led\_cdev->activated = false;

}

}

static struct led\_trigger heartbeat\_led\_trigger = {

.name = "heartbeat",

.activate = heartbeat\_trig\_activate,

.deactivate = heartbeat\_trig\_deactivate,

};

如果将触发方式更改为heartbeat，目录下多出invert文件，用于设置电平翻转。heartbeat的实现靠的就是其定时器回调led\_heartbeat\_function()，其4个case控制着时序，最终实现的时序如下：

以上时序是在invert=0的情况下，如果invert=1则时序电平正好与上图相反。

## 其他

除了内核已经提供的trigger方式外，其实内核中还有很多的子系统注册了trigger，如mmc在mmc\_add\_host()时注册了名为mmcx的触发器：

led\_trigger\_register\_simple(dev\_name(&host->class\_dev), &host->led);

然后在mmc开始或结束请求时都会去控制led:

led\_trigger\_event(host->led, LED\_FULL);

led\_trigger\_event(host->led, LED\_OFF);

# API说明

void led\_blink\_set(struct led\_classdev \*led\_cdev, unsigned long \*delay\_on, unsigned long \*delay\_off);

led闪烁设置，优先尝试使用硬件闪烁功能(led\_classdev->blink\_set)，不成功则使用定时器控制闪烁。

void led\_set\_brightness(struct led\_classdev \*led\_cdev, enum led\_brightness brightness);

设置led亮度，在使用定时器控制闪烁时将亮度设为0会结束闪烁。

void led\_trigger\_event(struct led\_trigger \*trig, enum led\_brightness brightness);

统一设置led\_trigger下挂载的所有led\_classdev的亮度。

void led\_trigger\_blink(struct led\_trigger \*trigger, unsigned long \*delay\_on, unsigned long \*delay\_off);

统一设置led\_trigger下挂载的所有led\_classdev的闪烁。

# 总结说明

led子系统除了控制led，其实也可以帮我们完成很多其他事情，比如控制开关，反正都是控制高低电平，又比如对于心跳可以控制心跳指示灯，也可以连到电源系统来指示CPU的活跃状态，对于借助leds-gpio.c通过gpio发出心跳和控制led或者开关，设备树需要配置如下：

leds {

compatible = "gpio-leds";

pinctrl-names = "default";

pinctrl-0 = <&user\_leds>;

led@0 {

label = "heartbeat";

gpios = <&gpio3 7 GPIO\_ACTIVE\_LOW>;

default-state = "on";

linux,default-trigger = "heartbeat";//发出心跳

};

led@1 {

label = "user-led0";

gpios = <&gpio0 20 GPIO\_ACTIVE\_HIGH>;

default-state = "on";//用于控制电平，默认为高

};

};

实际应用中有时也会用到i2c转gpio的芯片以节省引脚，如果这些gpio全部用于输出控制，其实也可以注册为led设备，以利用led子系统完成gpio的输出控制。反正就是led子系统虽然简单，但是只要合理利用其实也可以发挥很大的用处。