# Linux 相关知识

## 平台设备：

### 平台：SC7731C、SC9832 、SC9850

文件系统中的路径：/sys/bus/platform/devices

（1）、module\_init(assistant\_cam\_init);

（2）、

static int \_\_init assistant\_cam\_init(void)

{

    int ret = 0;

    /\* assistant\_cam user space access interface \*/

    ret = platform\_device\_register(&assistant\_cam\_user\_space\_device);

    if (ret) {

        GE\_DBG("\*\*\*\*[assistant\_cam\_init] Unable to device register(%d)\n",

                ret);

        return ret;

    }

    ret = platform\_driver\_register(&assistant\_cam\_user\_space\_driver); **//如果使用dts传入不需要这步**

    if (ret) {

        GE\_DBG("\*\*\*\*[assistant\_cam\_init] Unable to register driver (%d)\n",

                ret);

        return ret;

    }

    return 0;

}

struct platform\_device **assistant\_cam\_user\_space\_device** = {    **//如果使用dts传入不需要这步，**

   .name = "gc0310-user",//这个必须和下面的一样，而且会在目录 /sys/bus/platform/devices/下面生成//gc0310-user文件夹

    .id = -1,

};

static struct platform\_driver **assistant\_cam\_user\_space\_driver** = {

    .probe = assistant\_cam\_user\_space\_probe,

    .driver = {

           .name = "gc0310-user", //这个必须和上面的一样 ,会在devices目录下面创建一个gc0310-user文件夹。

    },

};

static int **assistant\_cam\_user\_space\_probe**(struct platform\_device \*dev)

{

    int ret\_device\_file = 0;

    GE\_DBG("\*\*\*\*\*\*\*\* assistant\_cam\_user\_space\_probe!! \*\*\*\*\*\*\*\*\n");

    ret\_device\_file = device\_create\_file(&(dev->dev), &dev\_attr\_gc0310\_init); //其中&dev\_attr\_gc0310\_init)是由下面的gc0310\_init 前面加上 dev\_attr得来的。

    return 0;

}

（3）、

static DEVICE\_ATTR(gc0310\_init, 0664, show\_assistant\_cam\_init, store\_assistant\_cam\_init);    /\* 664 \*/

**//会在 /sys/bus/platform/devices/gc0310-user/目录下生成一个gc0310\_init的节点。**

static ssize\_t **store\_assistant\_cam\_init**(struct device \*dev, struct device\_attribute \*attr,

                    const char \*buf, size\_t size)

{

    GE\_DBG("[store\_assistant\_cam\_init]\n");

    sensor\_init();

    return size;

}

static ssize\_t **show\_assistant\_cam\_init**(struct device \*dev, struct device\_attribute \*attr, char \*buf)

{

    GE\_DBG("[show\_assistant\_cam\_init]\n");

    sensor\_init();

    return 0;

}

（4）、所得节点的路径：

 /sys/bus/platform/devices/gc0310-user/gc0310\_init

（5）、

读操作：cat   /sys/bus/platform/devices/gc0310-user/gc0310\_init

写操作：echo  /sys/bus/platform/devices/gc0310-user/gc0310\_init

## I2C设备驱动

### 平台：SC7731C、SC9832 、SC9850

文件系统中的路径：/sys/bus/i2c/

好文推荐： **http://blog.csdn.net/snowwupl/article/details/9134145**

重要的结构体：

(1)、I2c\_add\_driver()是一个宏，其原型是：

#define i2c\_add\_driver(driver) \

i2c\_register\_driver(THIS\_MODULE, driver)

（2）、struct i2c\_driver {

unsigned int class;

int (\*attach\_adapter)(struct i2c\_adapter \*) \_\_deprecated;

int (\*detach\_adapter)(struct i2c\_adapter \*) \_\_deprecated;

int (\*probe)(struct i2c\_client \*, const struct i2c\_device\_id \*);

int (\*remove)(struct i2c\_client \*);

void (\*shutdown)(struct i2c\_client \*);

int (\*suspend)(struct i2c\_client \*, pm\_message\_t mesg);

int (\*resume)(struct i2c\_client \*);

void (\*alert)(struct i2c\_client \*, unsigned int data);

int (\*command)(struct i2c\_client \*client, unsigned int cmd, void \*arg);

struct device\_driver driver;

const struct i2c\_device\_id \*id\_table;

int (\*detect)(struct i2c\_client \*, struct i2c\_board\_info \*);

const unsigned short \*address\_list;

struct list\_head clients;

}

实例：

（1）、

        module\_init(assistant\_cam\_init);

        module\_exit(assistant\_cam\_exit);

（2）、

static int \_\_init assistant\_cam\_init(void)

{

    int ret = 0;

    if (i2c\_add\_driver(&assistant\_cam\_driver) != 0) {

        GE\_DBG(

                "[assistant\_cam\_init] failed to register assistant\_cam i2c driver.\n");

    } else {

        GE\_DBG(

                "[assistant\_cam\_init] Success to register assistant\_cam i2c driver.\n");

    }

}

 //如果没有这个DTS，也是要初始化device

#ifdef CONFIG\_OF

static const struct of\_device\_id assistant\_cam\_of\_match[] = {

{.compatible = "sprd,main2",}, //这个值应该和DTS中的值一样。

{},

};

MODULE\_DEVICE\_TABLE(of, assistant\_cam\_of\_match);

#endif

static struct i2c\_driver assistant\_cam\_driver = {

    .driver = {

            .name = "assistant\_cam",

#ifdef CONFIG\_OF

            .of\_match\_table = assistant\_cam\_of\_match,

#endif

    },

    .probe = assistant\_cam\_driver\_probe,

    .id\_table = assistant\_cam\_i2c\_id,

};

static void \_\_exit assistant\_cam\_exit(void)

{

    i2c\_del\_driver(&assistant\_cam\_driver);

}

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

{

new\_client = client;

//这里改i2c地址是为了dts中不去动原先sub摄像头的I2C地址

new\_client->addr=0x21;

assistant\_cam\_hw\_component\_detect();

return 0;

}

(3)、I2C读取

**方法一**

调用：i2c\_read\_data(new\_client,addr,1, (u8\*)&get\_byte);

static int i2c\_read\_data(struct i2c\_client \*client, unsigned char command, int length, unsigned char \*values)

{

int retry;

int err;

struct i2c\_msg msgs[] =

{

{

.addr = client->addr,

.flags = 0,

.len = 1,

.buf = &command,

},

{

.addr = client->addr,

.flags = I2C\_M\_RD,

.len = length,

.buf = values,

},

};

for (retry = 0; retry < 5; retry++)

{

err = i2c\_transfer(client->adapter, msgs, 2);

if (err == 2)

break;

else

mdelay(5);

}

if (retry >= 5)

{

GE\_DBG(KERN\_ERR "%s: i2c read fail, err=%d\n", \_\_func\_\_, err);

return -EIO;

}

return 0;

}

**方法二**

调用：static int xxxx\_i2c\_read( struct i2c\_client\* client,uint8\_t reg,uint8\_t \*data)

static int xxxx\_i2c\_read( struct i2c\_client\* client,uint8\_t reg,uint8\_t \*data)

{

// write reg addr

if( 1!= i2c\_master\_send(client,&reg,1) ) {

printk( KERN\_ERR " xxxx\_i2c\_read fail! \n" );

return -1;

}

// wait

msleep(10);

// read

if( 1!= i2c\_master\_recv(client,data,1) ) {

printk( KERN\_ERR " xxxx\_i2c\_read fail! \n" );

return -1;

}

return 0;

}

其中：i2c\_master\_recv 接口的三个参数：client 为此次与主机通信的从机，buf 为接收的数据指针，count 为接收数据的字节数。

(4)、I2C写入

**方法一**

调用：i2c\_write\_data(new\_client,addr,1,&para);

static int i2c\_write\_data(struct i2c\_client \*client, u16 command, u16 length, u16 \*values)

{

int retry;

int err;

unsigned char data[11];

struct i2c\_msg msg;

int index;

if (!client)

return -EINVAL;

else if (length >= 10)

{

GE\_DBG(KERN\_ERR "%s:length %d exceeds 10\n", \_\_func\_\_, length);

return -EINVAL;

}

data[0] = command;

for (index=1;index<=length;index++)

data[index] = values[index-1];

msg.addr = client->addr;

msg.flags = 0;

msg.len = length+1;

msg.buf = data;

for (retry = 0; retry < 5; retry++)

{

err = i2c\_transfer(client->adapter, &msg, 1);

if (err == 1)

break;

else

mdelay(5);

}

if (retry >= 5)

{

GE\_DBG(KERN\_ERR "%s: i2c write fail, err=%d\n", \_\_func\_\_, err);

return -EIO;

}

return 0;

}

**方法二**

调用：static int xxxx\_i2c\_write( struct i2c\_client\* client,uint8\_t reg,uint8\_t data)

static int xxxx\_i2c\_write( struct i2c\_client\* client,uint8\_t reg,uint8\_t data)

{

unsigned char buffer[2];

buffer[0] = reg;

buffer[1] = data;

if( 2!= i2c\_master\_send(client,buffer,2) ) {

printk( KERN\_ERR " xxxx\_i2c\_write fail! \n" );

return -1;

}

return 0;

}

其中：i2c\_master\_send 接口的三个参数：client 为此次与主机通信的从机，buf 为发送的数据指针，count 为发送数据的字节数。

(5)、文件系统的路径：

/sys/bus/i2c/drivers

和 /sys/bus/i2c/devices  //在这里可以看到匹配到的i2c设备的地址。

**I2C小知识：**



## proc\_create的使用方法

### 平台：SC7731C、SC9832 、SC9850

相关函数讲解：

头文件：#include <linux/proc\_fs.h>

（1）、proc文件夹的创建

struct proc\_dir\_entry \*proc\_mkdir(const char \*name, struct proc\_dir\_entry \*parent);

name就是要创建的文件夹名称。

parent是要创建节点的父节点。也就是要在哪个文件夹之下创建新文件夹，需要将哪个文件夹的proc\_dir\_entry传入。

如果是在/proc目录下创建文件夹，parent为NULL。

例如：

struct proc\_dir\_entry \*mytest\_dir = proc\_mkdir("mytest", NULL);

（2）、proc文件的创建

static inline struct proc\_dir\_entry \*proc\_create(const char \*name, mode\_t mode,

struct proc\_dir\_entry \*parent, const struct file\_operations \*proc\_fops);

name就是要创建的文件名。

mode是文件的访问权限，以UGO的模式表示。

parent与proc\_mkdir中的parent类似。也是父文件夹的proc\_dir\_entry对象。

proc\_fops就是该文件的操作函数了。

例如：

struct proc\_dir\_entry \*mytest\_file = proc\_create("mytest", 0x0644, mytest\_dir, mytest\_proc\_fops);

还有一种方式：

struct proc\_dir\_entry \*mytest\_file = proc\_create("mytest/mytest", 0x0644, NULL, mytest\_proc\_fops);

（3）、mytest\_proc\_fops结构体

Struct file\_operations {

　　struct module \*owner;//拥有该结构的模块的指针，一般为THIS\_MODULES

loff\_t (\*llseek) (struct file \*, loff\_t, int);//用来修改文件当前的读写位置

ssize\_t (\*read) (struct file \*, char \_\_user \*, size\_t, loff\_t \*);//从设备中同步读取数据

ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*);//向设备发送数据

ssize\_t (\*aio\_read) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);//初始化一个异步的读取操作

ssize\_t (\*aio\_write) (struct kiocb \*, const struct iovec \*, unsigned long, loff\_t);//初始化一个异步的写入操作

　　int (\*readdir) (struct file \*, void \*, filldir\_t);//仅用于读取目录，对于设备文件，该字段为NULL

unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*); //轮询函数，判断目前是否可以进行非阻塞的读写或写入

　　int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long); //执行设备I/O控制命令

　　long (\*unlocked\_ioctl) (struct file \*, unsigned int, unsigned long); //不使用BLK文件系统，将使用此种函数指针代替ioctl

　　long (\*compat\_ioctl) (struct file \*, unsigned int, unsigned long); //在64位系统上，32位的ioctl调用将使用此函数指针代替

　　int (\*mmap) (struct file \*, struct vm\_area\_struct \*); //用于请求将设备内存映射到进程地址空间

　　int (\*open) (struct inode \*, struct file \*); //打开

　　int (\*flush) (struct file \*, fl\_owner\_t id);

　　int (\*release) (struct inode \*, struct file \*); //关闭

　　int (\*fsync) (struct file \*, struct dentry \*, int datasync); //刷新待处理的数据

　　int (\*aio\_fsync) (struct kiocb \*, int datasync); //异步刷新待处理的数据

　　int (\*fasync) (int, struct file \*, int); //通知设备FASYNC标志发生变化

　　int (\*lock) (struct file \*, int, struct file\_lock \*);

　　ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);

　　unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);

　　int (\*check\_flags)(int);

　　int (\*flock) (struct file \*, int, struct file\_lock \*);

　　ssize\_t (\*splice\_write)(struct pipe\_inode\_info \*, struct file \*, loff\_t \*, size\_t, unsigned int);

　　ssize\_t (\*splice\_read)(struct file \*, loff\_t \*, struct pipe\_inode\_info \*, size\_t, unsigned int);

　　int (\*setlease)(struct file \*, long, struct file\_lock \*\*);

};

例如：

static const struct file\_operations mytest\_proc\_fops = {

.open = mytest\_proc\_open,

.read = seq\_read,

.write = mytest\_proc\_write,

.llseek = seq\_lseek,

.release = single\_release,

};

解析：

1)、以seq\_和single\_为前缀的函数都是kernel中现成的。可以参考文档：Documentation\filesystems\seq\_file.txt。

在mytest\_proc\_open函数中，**只需要调用single\_open函数**，并传入一个show函数即可。

例如：

static int mytest\_proc\_open(struct inode \*inode, struct file \*file)

{

return single\_open(file, mytest\_proc\_show, inode->i\_private);

}

其中第三个参数，也就是single\_open的data参数，内核中有些地方传入的NULL，有些地方传入的inode->i\_private，也有传入其他值的。来看看data在single\_open函数中如何被使用的：

if (!res)

((struct seq\_file \*)file->private\_data)->private = data;

所以data是seq\_file结构体的private成员。

2)、mytest\_proc\_show函数

传递给single\_open的show函数指针，将在proc file输出时被调用。

例如，执行cat /proc/mytest/mytest时，mytest\_proc\_show函数将会被调用。

static int mytest\_proc\_show (struct seq\_file \*seq, void \*v)

{

seq\_puts(seq, mytest\_flag ? "true\n" : "false\n");

return 0;

}

3）、mytest\_proc\_write函数 //mytest\_proc\_write函数会在写mytest文件时被调用, 功能就是记录写入数据到mytest文件。

static ssize\_t mytest\_proc\_write (struct file \*file, const char \_\_user \*buffer,

size\_t count, loff\_t \*pos)

{

char mode;

if (count > 0) {

if (get\_user(mode, buffer))

return -EFAULT;

mytest\_flag = (mode != '0');

}

return count;

}

应用实例：

（1）、

static int \_\_init zyt\_info\_proc\_init(void) {

proc\_create("zyt\_info", 0, NULL, &zyt\_info\_proc\_fops);

return 0;

}

static void \_\_exit zyt\_info\_proc\_exit(void) {

remove\_proc\_entry("zyt\_info", NULL);

}

MODULE\_LICENSE("GPL");

module\_init(zyt\_info\_proc\_init);

module\_exit(zyt\_info\_proc\_exit);

（2）、

static const struct file\_operations zyt\_info\_proc\_fops = {

.owner = THIS\_MODULE,

.open = zyt\_info\_proc\_open,

.read = seq\_read,

.llseek = seq\_lseek,

.release = single\_release,

};

（3）、

static int zyt\_info\_proc\_open(struct inode \*inode, struct file \*file) {

return single\_open(file, zyt\_info\_proc\_show, NULL);

}

（4）、

static int zyt\_info\_proc\_show(struct seq\_file \*m, void \*v) {

seq\_printf(m, "[LCD] : %s\n[MainSensor] : %s\n[SubSensor] : %s\n%s%s\n",

zyt\_get\_lcm\_info(),

get\_sensor\_info(0),

get\_sensor\_info(1),

zyt\_info\_buffer,

zyt\_battery\_info()

);

return 0;

}

调用方法：cat /proc/zyt\_info

## 4、内核中如何创建一个线程

### 平台：SC7731C、SC9832 、SC9850

函数：kthread\_run

解析：

kthread\_run是一个宏,用来创建一个进程,并且将其唤醒,其定义在头文件include/linux/kthread.h中.

#define kthread\_run(threadfn, data, namefmt, ...) \

({ \

struct task\_struct \*\_\_k \

= kthread\_create(threadfn, data, namefmt, ## \_\_VA\_ARGS\_\_); \

if (!IS\_ERR(\_\_k)) \

wake\_up\_process(\_\_k); \

\_\_k; \

})

先来看看这个宏的参数,threadfn是该线程的执行函数,data是执行函数的传入参数,namefmt是该线程的printf风格的线程名,最后的...类似printf的可变参数列表.从这个宏的实现可以看出kthread\_run是通过kthread\_create来创建一个进程,并返回一个task\_struct,然后使用wake\_up\_process函数将新创建的进程唤醒.下面需要关注一下kthread\_create的实现,其实它也是一个宏,同样也定义在头文件include/linux/kthread.h中.

更详细的解释的链接：<http://blog.csdn.net/cq062364/article/details/39646623>

## 5、队列

## (1)、工作队列

注册工作队列：

INIT\_DELAYED\_WORK(&dev->dispc\_esd\_work, dispc\_esd\_handle);  //参数：触发条件、执行函数

也可以使用宏 INIT\_WORK()

唤醒工作队列：

1. 使用两对函数INIT\_DELAYED\_WORK()对schedule\_delayed\_work(),INIT\_WORK()对schedule\_work()。使这两个函数的时候工作者线程是通过直接使用内核中每个CPU对应的一个 缺省工作者线程envents/n（n代表CPU的序号，从0开始）【扩展：如果是shudule\_work的话，系统中有个events内核线程，这个线程会处理你用shudule\_work接口激活 的所有work。如果是queue\_work的话，一般这种情况都是自己创建了一个单独的处理线程，这样将 你激活的work和这个线程联系起来。至于什么时候运行，那就是events或是你定义的特定线程运行的时】

2、如果缺省的队列不能满足要求，自己创建一个工作者线程也很简单，只需要调用：

struct workqueue\_struct \*create\_workqueue(const char \*name);

调度时使用如下函数：

int queue\_work(struct workqueue\_struct \*wq, struct work\_struct \*work);

或

int queue\_delayed\_work(struct workqueue\_struct \*wq, struct work\_struct

\*work, unsigned long delay);

queue\_delayed\_work(gsl\_monitor\_workqueue, &gsl\_monitor\_work, 1000);

取消工作队列：

cancel\_delayed\_work()

在这下面一般会使用：

flush\_scheduled\_work() // 为了防止有竞争条件的出现, 对于内核模块,如果一个模块使用了工作队列机制,并且利用了events这个缺省队列,那么在卸载这个模块之前,您必须得调用这个函数,这叫做刷新一个工作队列,也就是说,函数会一直等待,直到队列中所有对象都被执行以后才返回.当然,在等待的过程中,这个函数可以进入睡眠.反正刷新完了之后,这个函数会被唤醒,然后它就返回了.

**例子：**

1、头文件路径：include/linux/workqueue.h

2、变量定义：

static struct delayed\_work gsl\_monitor\_work;

static struct workqueue\_struct \*gsl\_monitor\_workqueue = NULL;

3、初始化：

INIT\_DELAYED\_WORK(&gsl\_monitor\_work, gsl\_monitor\_worker);

4、创建一个专用的工作者线程

gsl\_monitor\_workqueue = create\_singlethread\_workqueue("gsl\_monitor\_workqueue");

5、启动队列：

queue\_delayed\_work(gsl\_monitor\_workqueue, &gsl\_monitor\_work, 1000);

6、取消工作队列

cancel\_delayed\_work(&gsl\_monitor\_work)；

7、执行函数：

static void gsl\_monitor\_worker(void)

{

。

。

。

}

## (2)、等待队列

## 6、字符设备驱动

### 平台：SC7731C、SC9832 、SC9850

挂载驱动：insmod globalmem.ko

移除驱动：rmmod globalmem.ko

**（1）、查看主次设备号：**字符设备文件由使用 ls -l 的输出的第一列的“c”标识。如果使用 ls -l 命令, 会看到在设备文件项中有 2 个数(由一个逗号分隔) 这些数字就是设备文件的主次设备编号。

**（2）设备号的作用：**主设备号用来标识与设备文件相连的驱动程序。次编号被驱动用来决定操作的是哪个设备。

**（3）、关于内核设备号：**

1）、struct dev\_t 实质为unsigned int 32位整数，其中高12位为主设备号，低20位为次设备号

2）、如何从kdev\_t结构中分解出主设备号： MAJOR(dev\_t dev)

3）、如何从kdev\_t结构中分解出次设备号： MINOR(dev\_t dev)

4）、如何构造设备号： dev\_t MKDEV(unsigned int major, unsigned int minor);

**（4）、内核中如何分配设备号**

**静态分配：**

1）、根据documentxtation/devices.txt,找到一个没有使用的设备号,并手工指定.

2）、使用register\_chrdev\_region函数注册

dev = MKDEV(devmajor, devminor); // 1 获得设备号

result = register\_chrdev\_region(dev, 1, "led"); // 2 分配设备编号

缺点：设备号容易冲突。

其中：**int register\_chrdev\_region(dev\_t from, unsigned count, const char \*name)**

功能：

申请使用从 from 开始的 count 个设备号(主设备号不变，次设备号增加）

参数：

from：希望申请使用的设备号

count：希望申请使用设备号数目

name：设备名(体现在/proc/devices)

**动态分配**

方法：使用动态分配函数 int alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count,const char \*name) 分配设备号

功能：

请求内核动态分配 count 个设备号，且次设备号从baseminor开始。

参数：

dev：分配到的设备号

baseminor：起始次设备号

count：需要分配的设备号数目

name：设备名(体现在/proc/devices)

例如：

result = alloc\_chrdev\_region(&dev, devminor, 1, “led”);//2 动态分配设备编号

devmajor = MAJOR(dev);

优点：

简单，易于驱动推广

缺点：

无法在安装驱动前创建设备文件,因为还没有分配到主设备号。

解决办法：

一旦驱动程序安装、主设备号分配了, 可从 /proc/devices 中查询到

**（5）、注销设备号：**

1）、定义：不论使用何种方法分配设备号，都应该在不再使用它们时释放这些设备号。

2）、函数：void unregister\_chrdev\_region(dev\_t from,unsigned count)//释放从from开始的count个设备号

**（6）、设备文件**

1）、创建有两种方法

手动：

mknod filename type major minor

Filename:设备文件名

Type: 设备文件类型

Major: 主设备号

Minor: 次设备号

例： mknod /dev/serial0 c 100 0

## 7、关于request\_threaded\_irq----中断线程化

### 平台：SC7731C、SC9832 、SC9850

好文：http://www.wowotech.net/linux\_kenrel/request\_threaded\_irq.html

申请中断函数原型：

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);

函数参数解析：

1>：irq:表示申请的中断号。

2>：handler:表示中断服务例程（相当于中断处理的上半部分）

3.> thread\_fn：中断线程化，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。 （相当于中断处理的下半部分）

4>.irqflags:表示中断标志位。

标志位：

IRQF\_TRIGGER\_RISING ：上升沿触发中断

IRQF\_TRIGGER\_FALLING ：下降沿触发中断

IRQF\_TRIGGER\_HIGH ： 高电平触发中断

IRQF\_TRIGGER\_LOW ：低电平触发中断

IRQF\_ONESHOT ：为保证中断在底半部thread\_fn执行完之后才会继续接受中断

IRQF\_NO\_SUSPEND ：这个flag比较好理解，就是说在系统suspend的时候，不用disable这个中断，如果disable，可能会导致系统不能正常的resume。

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。 共享中断时所用。

## 8、Linux的相关命令

重启命令：

1、reboot

2、shutdown -r now 立刻重启(root用户使用)

3、shutdown -r 10 过10分钟自动重启(root用户使用)

4、shutdown -r 20:35 在时间为20:35时候重启(root用户使用)

如果是通过shutdown命令设置重启的话，可以用shutdown -c命令取消重启

关机命令：

1、halt 立刻关机

2、poweroff 立刻关机

3、shutdown -h now 立刻关机(root用户使用)

4、shutdown -h 10 10分钟后自动关机

如果是通过shutdown命令设置关机的话，可以用shutdown -c命令取消重启

查看CPU详细信息：

cat /proc/cpuinfo 查看CPU的所有信息

cat /proc/cpuinfo| grep "physical id"| sort| uniq| wc -l 查看物理CPU个数

cat /proc/cpuinfo| grep "cpu cores"| uniq 查看每个物理CPU中core的个数

cat /proc/cpuinfo| grep "processor"| wc -l 查看逻辑CPU的个数

grep 'processor' /proc/cpuinfo | sort -u | wc -l 查看总线程数量

cat /proc/cpuinfo | grep name | cut -f2 -d: | uniq -c 查看CPU信息（型号）

查看内存大小

free

查看磁盘大小

df -h

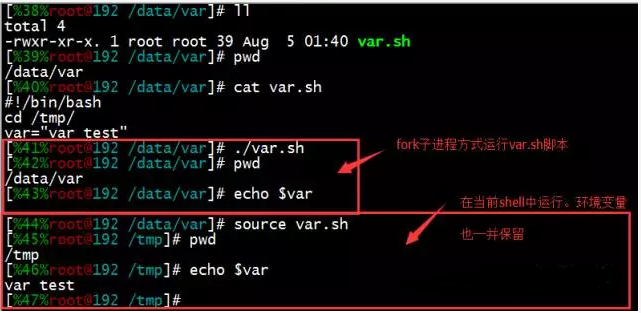
sudo uname --s 显示内核名字s

sudo uname --r 显示内核版本

sudo uname --n 显示网络主机名

sudo uname --p 显示cpu （比如32位和64位）

## 9、source 和 ./ 执行的区别



如图大家可以看到用./方式执行完脚本后，脚本内的所有环境变量均不影响当时的SHELL环境变量，而使用source则完全相反的效果。

1、source命令用法：

source FileName

作用:

在当前bash环境下读取并执行FileName中的命令。该filename文件可以无执行权限”

注：该命令通常用命令“.”来替代。

如：source .bash\_profile

. .bash\_profile两者等效。

source(或点)命令通常用于重新执行刚修改的初始化文档。

source命令(从 C Shell 而来)是bash shell的内置命令。

点命令，就是个点符号，(从Bourne Shell而来)。

3、./的命令用法：

./FileName

作用:

打开一个子shell来读取并执行FileName中命令，需要有执行权限。

注：运行一个shell脚本时会启动另一个命令解释器.

每个shell脚本有效地运行在父shell(parent shell)的一个子进程里. 这个父shell是指在一个控制终端或在一个xterm窗口中给你命令指示符的进程.shell脚本也可以启动他自已的子进程. 这些子shell(即子进程)使脚本并行地，有效率地地同时运行脚本内的多个子任务.

常用的shell的嵌入命令（内部命令）有如下这些，

: 空，永远返回为true

. 从当前shell中执行操作

break 退出for、while、until或case语句

cd 改变到当前目录

continue 执行循环的下一步

echo 反馈信息到标准输出

eval 读取参数，执行结果命令

exec 执行命令，但不在当前shell

exit 退出当前shell

export 导出变量，使当前shell可利用它

pwd 显示当前目录

read 从标准输入读取一行文本

readonly 使变量只读

return 退出函数并带有返回值

set 控制各种参数到标准输出的显示

shift 命令行参数向左偏移一个

test 评估条件表达式

times 显示shell运行过程的用户和系统时间

trap 当捕获信号时运行指定命令

ulimit 显示或设置shell资源

umask 显示或设置缺省文件创建模式

unset 从shell内存中删除变量或函数

wait 等待直到子进程运行完毕

直接执行enable命令也可以得到所有的内部命令列表。

## 10、LINUX 的一些宏的含义

11.1 、EXPORT\_SYMBOL的作用

概念：EXPORT\_SYMBOL标签内定义的函数或者符号对全部内核代码公开，不用修改内核代码就可以在您的内核模块中直接调用，即使用EXPORT\_SYMBOL可以将一个函数以符号的方式导出给其他模块使用。

例子：

/\* LED flash control for high current capture mode\*/

ssize\_t strobe\_VDIrq(void)

{

return 0;

}

EXPORT\_SYMBOL(strobe\_VDIrq);

11.2、

## 11、linux内核自锁旋spinlock常用宏解释

自旋锁与互斥锁有点类似，只是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

　　由于自旋锁使用者一般保持锁时间非常短，因此选择自旋而不是睡眠是非常必要的，自旋锁的效率远高于互斥锁。

　　信号量和读写信号量适合于保持时间较长的情况，它们会导致调用者睡眠，因此只能在进程上下文使用（\_trylock的变种能够在中断上下文使用），而自旋锁适合于保持时间非常短的情况，它可以在任何上下文使用。

　　如果被保护的共享资源只在进程上下文访问，使用信号量保护该共享资源非常合适，如果对共巷资源的访问时间非常短，自旋锁也可以。但是如果被保护的共享资源需要在中断上下文访问（包括底半部即中断处理句柄和顶半部即软中断），就必须使用自旋锁。

　　自旋锁保持期间是抢占失效的，而信号量和读写信号量保持期间是可以被抢占的。自旋锁只有在内核可抢占或SMP的情况下才真正需要，在单CPU且不可抢占的内核下，自旋锁的所有操作都是空操作。

　　跟互斥锁一样，一个执行单元要想访问被自旋锁保护的共享资源，必须先得到锁，在访问完共享资源后，必须释放锁。如果在获取自旋锁时，没有任何执行单元保持该锁，那么将立即得到锁；如果在获取自旋锁时锁已经有保持者，那么获取锁操作将自旋在那里，直到该自旋锁的保持者释放了锁。

　　无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。

　　自旋锁的API有：

spin\_lock\_init(x)

　　该宏用于初始化自旋锁x。自旋锁在真正使用前必须先初始化。该宏用于动态初始化。

DEFINE\_SPINLOCK(x)

　　该宏声明一个自旋锁x并初始化它。该宏在2.6.11中第一次被定义，在先前的内核中并没有该宏。

SPIN\_LOCK\_UNLOCKED

　　该宏用于静态初始化一个自旋锁。

DEFINE\_SPINLOCK(x)等同于spinlock\_t x = SPIN\_LOCK\_UNLOCKED spin\_is\_locked(x)

　　该宏用于判断自旋锁x是否已经被某执行单元保持（即被锁），如果是，返回真，否则返回假。

spin\_unlock\_wait(x)

　　该宏用于等待自旋锁x变得没有被任何执行单元保持，如果没有任何执行单元保持该自旋锁，该宏立即返回，否则将循环在那里，直到该自旋锁被保持者释放。

spin\_trylock(lock)

　　该宏尽力获得自旋锁lock，如果能立即获得锁，它获得锁并返回真，否则不能立即获得锁，立即返回假。它不会自旋等待lock被释放。

spin\_lock(lock)

　　该宏用于获得自旋锁lock，如果能够立即获得锁，它就马上返回，否则，它将自旋在那里，直到该自旋锁的保持者释放，这时，它获得锁并返回。总之，只有它获得锁才返回。

spin\_lock\_irqsave(lock, flags)

　　该宏获得自旋锁的同时把标志寄存器的值保存到变量flags中并失效本地中断。

spin\_lock\_irq(lock)

　　该宏类似于spin\_lock\_irqsave，只是该宏不保存标志寄存器的值。

spin\_lock\_bh(lock)

　　该宏在得到自旋锁的同时失效本地软中断。

spin\_unlock(lock)

　　该宏释放自旋锁lock，它与spin\_trylock或spin\_lock配对使用。如果spin\_trylock返回假，表明没有获得自旋锁，因此不必使用spin\_unlock释放。

spin\_unlock\_irqrestore(lock, flags)

　　该宏释放自旋锁lock的同时，也恢复标志寄存器的值为变量flags保存的值。它与spin\_lock\_irqsave配对使用。

spin\_unlock\_irq(lock)

　　该宏释放自旋锁lock的同时，也使能本地中断。它与spin\_lock\_irq配对应用。

spin\_unlock\_bh(lock)

　　该宏释放自旋锁lock的同时，也使能本地的软中断。它与spin\_lock\_bh配对使用。

spin\_trylock\_irqsave(lock, flags)

　　该宏如果获得自旋锁lock，它也将保存标志寄存器的值到变量flags中，并且失效本地中断，如果没有获得锁，它什么也不做。

　　因此如果能够立即获得锁，它等同于spin\_lock\_irqsave，如果不能获得锁，它等同于spin\_trylock。如果该宏获得自旋锁lock，那需要使用spin\_unlock\_irqrestore来释放。

spin\_trylock\_irq(lock)

　　该宏类似于spin\_trylock\_irqsave，只是该宏不保存标志寄存器。如果该宏获得自旋锁lock，需要使用spin\_unlock\_irq来释放。

spin\_trylock\_bh(lock)

　　该宏如果获得了自旋锁，它也将失效本地软中断。如果得不到锁，它什么也不做。因此，如果得到了锁，它等同于spin\_lock\_bh，如果得不到锁，它等同于spin\_trylock。如果该宏得到了自旋锁，需要使用spin\_unlock\_bh来释放。

spin\_can\_lock(lock)

　　该宏用于判断自旋锁lock是否能够被锁，它实际是spin\_is\_locked取反。如果lock没有被锁，它返回真，否则，返回假。该宏在2.6.11中第一次被定义，在先前的内核中并没有该宏。

　　获得自旋锁和释放自旋锁有好几个版本，因此让读者知道在什么样的情况下使用什么版本的获得和释放锁的宏是非常必要的。

　　如果被保护的共享资源只在进程上下文访问和软中断上下文访问，那么当在进程上下文访问共享资源时，可能被软中断打断，从而可能进入软中断上下文来对被保护的共享资源访问，因此对于这种情况，对共享资源的访问必须使用spin\_lock\_bh和spin\_unlock\_bh来保护。

　　当然使用spin\_lock\_irq和spin\_unlock\_irq以及spin\_lock\_irqsave和spin\_unlock\_irqrestore也可以，它们失效了本地硬中断，失效硬中断隐式地也失效了软中断。但是使用spin\_lock\_bh和spin\_unlock\_bh是最恰当的，它比其他两个快。

　　如果被保护的共享资源只在进程上下文和tasklet或timer上下文访问，那么应该使用与上面情况相同的获得和释放锁的宏，因为tasklet和timer是用软中断实现的。

　　如果被保护的共享资源只在一个tasklet或timer上下文访问，那么不需要任何自旋锁保护，因为同一个tasklet或timer只能在一个CPU上运行，即使是在SMP环境下也是如此。实际上tasklet在调用tasklet\_schedule标记其需要被调度时已经把该tasklet绑定到当前CPU，因此同一个tasklet决不可能同时在其他CPU上运行。

　　timer也是在其被使用add\_timer添加到timer队列中时已经被帮定到当前CPU，所以同一个timer绝不可能运行在其他CPU上。当然同一个tasklet有两个实例同时运行在同一个CPU就更不可能了。

　　如果被保护的共享资源只在两个或多个tasklet或timer上下文访问，那么对共享资源的访问仅需要用spin\_lock和spin\_unlock来保护，不必使用\_bh版本，因为当tasklet或timer运行时，不可能有其他tasklet或timer在当前CPU上运行。

　如果被保护的共享资源只在一个软中断（tasklet和timer除外）上下文访问，那么这个共享资源需要用spin\_lock和spin\_unlock来保护，因为同样的软中断可以同时在不同的CPU上运行。

　　如果被保护的共享资源在两个或多个软中断上下文访问，那么这个共享资源当然更需要用spin\_lock和spin\_unlock来保护，不同的软中断能够同时在不同的CPU上运行。

　　如果被保护的共享资源在软中断（包括tasklet和timer）或进程上下文和硬中断上下文访问，那么在软中断或进程上下文访问期间，可能被硬中断打断，从而进入硬中断上下文对共享资源进行访问，因此，在进程或软中断上下文需要使用spin\_lock\_irq和spin\_unlock\_irq来保护对共享资源的访问。

　　而在中断处理句柄中使用什么版本，需依情况而定，如果只有一个中断处理句柄访问该共享资源，那么在中断处理句柄中仅需要spin\_lock和spin\_unlock来保护对共享资源的访问就可以了。

　　因为在执行中断处理句柄期间，不可能被同一CPU上的软中断或进程打断。但是如果有不同的中断处理句柄访问该共享资源，那么需要在中断处理句柄中使用spin\_lock\_irq和spin\_unlock\_irq来保护对共享资源的访问。

　　在使用spin\_lock\_irq和spin\_unlock\_irq的情况下，完全可以用spin\_lock\_irqsave和spin\_unlock\_irqrestore取代，那具体应该使用哪一个也需要依情况而定，如果可以确信在对共享资源访问前中断是使能的，那么使用spin\_lock\_irq更好一些。

　　因为它比spin\_lock\_irqsave要快一些，但是如果你不能确定是否中断使能，那么使用spin\_lock\_irqsave和spin\_unlock\_irqrestore更好，因为它将恢复访问共享资源前的中断标志而不是直接使能中断。

　　当然，有些情况下需要在访问共享资源时必须中断失效，而访问完后必须中断使能，这样的情形使用spin\_lock\_irq和spin\_unlock\_irq最好。

## 12、Kconfig,Makefile 和 .config

分布在各目录下的Kconfig构成了一个分布式的内核配置数据库，每个Kconfig分别描述了所属目录源文件相关的内核配置菜单。在内核配置make menuconfig(或xconfig等)时，从Kconfig中读出配置菜单，用户配置完后保存到.config(在顶层目录下生成)中。在内核编译时，主Makefile调用这个.config，就知道了用户对内核的配置情况。Kconfig就是对应着内核的配置菜单。假如要想添加新的驱动到内核的源码中，可以通过修改Kconfig来增加对我们驱动的配置菜单，这样就有途径选择我们的驱动，假如想使这个驱动被编译，还要修改该驱动所在目录下的Makefile。

**Kconfig**

先来看下一个相对完整的Kconfig文件：

menuconfig MISC\_DEVICES

bool "Misc devices"

---help---

Say Y here to get to see options for device drivers from various

different categories. This option alone does not add any kernel code.

If you say N, all options in this submenu will be skipped and disabled.

if MISC\_DEVICES

config ST\_L3GD20\_GYR

tristate "L3GD20\_GYR gyroscope sensor support"

depends on I2C=y

help

If you say yes here you get support for ST's

gyroscope sensors L3GD20\_GYR.

choice

prompt "Preemption Model"

depends on SENSORS\_AFA750

default CALI\_NONE

config CALI\_NONE

bool "None"

help

Say yes here to disable calibration function for AFA750

config CALI\_POSITIVE

bool "positive calibration"

help

Say yes here when the afa750 and LCD are laid towared the same direction on your board

endchoice

config SENSORS\_LSM303D

tristate "LSM303 sensor driver"

depends on I2C=y

help

Say yes here to support the sensor

endif

1.语法：

config symbol

options

symbol是一个新的标记的菜单项，options是在这个新的菜单项下的属性和选项。

2.菜单结构：

配置文件描述了菜单选项，每行都是以一关键字开头(除了帮助信息)。下面的关键字结束一菜单选项：

- config ：config是关键字，表示一个配置选项的开始，是菜单menuconfig 的子菜单；紧跟着的TMPFS\_POSIX\_ACL是配置选项的名称，省略了前缀"CONFIG\_

- menuconfig : 表示一个菜单

- choice/endchoice

choice条目将多个类似的配置选项组合在一起，供用户单选或多选

choice

prompt "ARM system type"

default ARCH\_VERSATILE

config ARCH\_AAEC2000

.........

config ARCH\_REALVIEW

.........

endchoice

prompt "ARM system type"给出提示信息“ARM system type”，光标选中

后回车进入就可以看到多个config条目定义的配置选项

- comment ： comment条目用于定义一些帮助信息，出现在界面的第一行，如在arch/arm/Kconifg中有如下代码：

- menu/endmenu

menu条目用于生成菜单，其格式如下：

menu "Floating poing emulation"

config FPE\_NWFPE

..............

config FPE\_NWFPE\_XP

.............

endmenu

menu之后的Floating poing emulation是菜单名，menu和endmenu间有很多config条目，在配置界面中如下所示：

Floating poing emulation--->

[] FPE\_NWFPE

[] FPE\_NWFPE\_XP

- if/endif ：用作判断菜单是否存在

例如：

menuconfig INPUT\_TOUCHSCREEN

bool "Touchscreens"

help

Say Y here, and a list of supported touchscreens will be displayed.

This option doesn't affect the kernel.

If unsure, say Y.

if INPUT\_TOUCHSCREEN

- source ：内核源码目录树中每一个Kconfig都会用source引入其所有子目录下的Kconfig，从而保证了所有的Kconfig项目都被包含进menuconfig中。这个也说明了：如果你自己在linux内核中添加了一个文件夹，一定要在这个文件夹下创建一个Kconfig文件，然后在这个文件夹的上一层目录的Kconfig中source引入这个文件夹下的Kconfig文件。

2.options类型定义：

每个config菜单项都要有类型定义：bool布尔类型、 tristate三态（内建、模块、移除）、 string字符串、 hex十六进制、 integer整型。

例如：

[html] view plain copy

config CALI\_NONE

bool "None"

bool类型的只能选中或不选中，tristate类型的菜单项多了编译成内核模块的选项，如果选择编译成内核模块，则会在.config中生成一个CONFIG\_CALI\_NONEE=m的配置，如果选择内建，就是直接编译成内核影响，就会在.config中生成一个CONFIG\_CALI\_NONE=y的配置.

3.依赖型定义depends on或requires

指此菜单的出现与否依赖于另一个定义

config SENSORS\_LSM303D

tristate "LSM303 sensor driver"

depends on I2C=y

这个例子表明SENSORS\_LSM303D这个菜单项只I2C有效。

4.select与depends on是相反的逻辑关系。

A depends on B

那么只有在B选中才能选A

A select B

那么只要选中A就会选中B

5.帮助性定义

只是增加帮助用关键字help或者---help---，"---help---" 和 "help" 在实现的作用上没有区别，"---help---" 有助于将文件中的配置逻辑与给开发人员的提示分开。

6.prompt --输入提示

**Makefile**

1.顶层的Makefile文档读取 .config文档的内容，并总体上负责build内核和模块。

2.Arch Makefile则提供补充体系结构相关的信息。

3.scripts目录下的Makefile文档包含了任何用来根据kbuild Makefile 构建内核所需的定义和规则。

其中.config的内容是在make menuconfig的时候，通过Kconfig文档配置的结果，在/Documentation/kbuild目录下有详细的介绍有关kernel makefile的知识。

**举个例子：**

假设想把G-sensor LSM303D驱动code加载到工程中，配置内核时该怎么办呢？

1：将您写的lsm303d.c 文档添加到/driver/misc/ 目录下。

2：修改/driver/misc/ 目录下的kconfig文档：

[html] view plain copy

config SENSORS\_LSM303D

tristate "LSM303 sensor driver"

depends on I2C=y

help

Say yes here to support the sensor

3：修改该目录下makefile文档。

添加code：

[html] view plain copy

obj-$(CONFIG\_SENSORS\_LSM303D) += lsm303d.o

从上述分析知道CONFIG\_SENSORS\_LSM303D 是从.config 中读出的。

4.配置kernel下configs/XXXX\_defconfig文件

添加code：

[html] view plain copy

CONFIG\_SENSORS\_LSM303D=y

当您编译内核时，将会读取.config文档，当发现CONFIG\_SENSORS\_LSM303D=y，系统在调用/driver/misc下的makefile 时，将会把 lsm303d.o 加入到内核中。即可达到您的目的。

Foreach :

Makefile中的foreach函数几乎是仿照于Unix标准Shell （/bin/sh）中的for语句，或是C-Shell（/bin/csh）中的foreach语句而构建的。它的语法是：

$(foreach <var>,<list>,<text>)

这个函数的意思是，把参数<list>;中的单词逐一取出放到参数<var>;所指定的变量中，然后再执行< text>;所包含的表达式。每一次<text>;会返回一个字符串，循环过程中，<text>;的所返回的每个字符串会以空格分隔，最后当整个循环结束时，<text>;所返回的每个字符串所组成的整个字符串（以空格分隔）将会是foreach函数的返回值。

所以，<var>;最好是一个变量名，<list>;可以是一个表达式，而<text>;中一般会使用<var>;这个参数来依次枚举<list>;中的单词。举个例子：

names := a b c d

files := $(foreach n,$(names),$(n).o)

上面的例子中，$(name)中的单词会被挨个取出，并存到变量“n”中，“$(n).o”每次根据“$(n)”计算出一个值，这些值以空格分隔，最后作为foreach函数的返回，所以，$(files)的值是“a.o b.o c.o d.o”。

注意，foreach中的<var>;参数是一个临时的局部变量，foreach函数执行完后，参数<var>;的变量将不在作用，其作用域只在foreach函数当中。

## 13、编译报错

（1）、

extern void \_\_init xprj\_earlyyyy\_debug(void);

static void earlycon\_owl\_write(struct console \*con, const char \*s, unsigned n)

{

#if 1

xprj\_earlyyyy\_debug();

while (1);

#endif

错误信息为：

WARNING: modpost: Found 2 section mismatch(es).

To see full details build your kernel with:

make CONFIG\_DEBUG\_SECTION\_MISMATCH=y'

FATAL: modpost: Section mismatches detected.

Set CONFIG\_SECTION\_MISMATCH\_WARN\_ONLY=y to allow them.

make[3]: \*\*\* [vmlinux.o] Error 1

make[2]: \*\*\* [vmlinux] Error 2

原因是我们在非\_\_init的section中（earlycon\_owl\_write）调用了\_\_init section（xprj\_earlyyyy\_debug）的代码，按照提示说的，打开CONFIG\_SECTION\_MISMATCH\_WARN\_ONLY配置项即可编译通过：

## 14、Linux相关网站

Linux源码下载：https://www.kernel.org/

## 15、Linux内核空间内存申请函数kmalloc、kzalloc、vmalloc的区别【转】

我们都知道在用户空间动态申请内存用的函数是 malloc()，这个函数在各种操作系统上的使用是一致的，对应的用户空间内存释放函数是 free()。注意：动态申请的内存使用完后必须要释放，否则会造成内存泄漏，如果内存泄漏发生在内核空间，则会造成系统崩溃。

　　那么，在内核空间中如何申请内存呢？一般我们会用到 kmalloc()、kzalloc()、vmalloc() 等，下面我们介绍一下这些函数的使用以及它们之间的区别。

kmalloc()

函数原型：void \*kmalloc(size\_t size, gfp\_t flags)；

　　kmalloc() 申请的内存位于物理内存映射区域，而且在物理上也是连续的，它们与真实的物理地址只有一个固定的偏移，因为存在较简单的转换关系，所以对申请的内存大小有限制，不能超过128KB。

较常用的 flags（分配内存的方法）：

GFP\_ATOMIC —— 分配内存的过程是一个原子过程，分配内存的过程不会被（高优先级进程或中断）打断；

GFP\_KERNEL —— 正常分配内存；

GFP\_DMA —— 给 DMA 控制器分配内存，需要使用该标志（DMA要求分配虚拟地址和物理地址连续）。

flags 的参考用法：

　|– 进程上下文，可以睡眠　　　　　GFP\_KERNEL

　|– 进程上下文，不可以睡眠　　　　GFP\_ATOMIC

　|　　|– 中断处理程序　　　　　　　GFP\_ATOMIC

　|　　|– 软中断　　　　　　　　　　GFP\_ATOMIC

　|　　|– Tasklet　　　　　　　　　GFP\_ATOMIC

　|– 用于DMA的内存，可以睡眠　　　GFP\_DMA | GFP\_KERNEL

　|– 用于DMA的内存，不可以睡眠　　GFP\_DMA |GFP\_ATOMIC

对应的内存释放函数为：

void kfree(const void \*objp);

kzalloc()

　　kzalloc() 函数与 kmalloc() 非常相似，参数及返回值是一样的，可以说是前者是后者的一个变种，因为 kzalloc() 实际上只是额外附加了 \_\_GFP\_ZERO 标志。所以它除了申请内核内存外，还会对申请到的内存内容清零。

/\*\* \* kzalloc - allocate memory. The memory is set to zero. \* @size: how many bytes of memory are required. \* @flags: the type of memory to allocate (see kmalloc). \*/static inline void \*kzalloc(size\_t size, gfp\_t flags){ return kmalloc(size, flags | \_\_GFP\_ZERO);}

kzalloc() 对应的内存释放函数也是 kfree()。

vmalloc()

函数原型：

void \*vmalloc(unsigned long size);

　　vmalloc() 函数则会在虚拟内存空间给出一块连续的内存区，但这片连续的虚拟内存在物理内存中并不一定连续。由于 vmalloc() 没有保证申请到的是连续的物理内存，因此对申请的内存大小没有限制，如果需要申请较大的内存空间就需要用此函数了。

对应的内存释放函数为：

void vfree(const void \*addr);

注意：vmalloc() 和 vfree() 可以睡眠，因此不能从中断上下文调用。

总结

kmalloc()、kzalloc()、vmalloc() 的共同特点是：

1、用于申请内核空间的内存；

2、内存以字节为单位进行分配；

3、所分配的内存虚拟地址上连续；

kmalloc()、kzalloc()、vmalloc() 的区别是：

1、kzalloc 是强制清零的 kmalloc 操作；（以下描述不区分 kmalloc 和 kzalloc）

2、kmalloc 分配的内存大小有限制（128KB），而 vmalloc 没有限制；

3、kmalloc 可以保证分配的内存物理地址是连续的，但是 vmalloc 不能保证；

4、kmalloc 分配内存的过程可以是原子过程（使用 GFP\_ATOMIC），而 vmalloc 分配内存时则可能产生阻塞；

5、kmalloc 分配内存的开销小，因此 kmalloc 比 vmalloc 要快；

一般情况下，内存只有在要被 DMA 访问的时候才需要物理上连续，但为了性能上的考虑，内核中一般使用 kmalloc()，而只有在需要获得大块内存时才使用 vmalloc()。例如，当模块被动态加载到内核当中时，就把模块装载到由 vmalloc() 分配的内存上。

## 16、Linux内核中的pinctrl子系统应用实例

转：https://blog.csdn.net/hanp\_linux/article/details/72818437

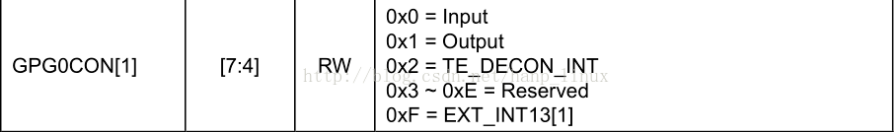
由于近期在做一个项目用到了pinctrl子系统，但是对pinctrl子系统了解又不是很多，所以遇到了麻烦，但是找度娘发现很少有同行对pinctrl的具体用法做出说明，所以只能自己去搞了，在经过一段时间对Linux内核源码的折腾，最终搞定，并将我所应用的实例给展示一下，希望对大家有所帮助。

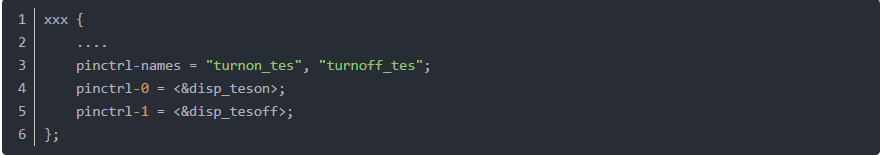
关于pinctrl是什么，为什么要用pinctrl，源码深度剖析我在这就不赘述了，有位博友总结的非常好，大家可以参考http://www.wowotech.net/sort/gpio\_subsystem。

下面我介绍一下如何去使用内核中的pinctrl子系统以device tree设备树为例，当你需要控制某些pin的时候，你首先要在devicetree中去按照pinctrl的规则去描述它，然后才能在driver中去使用：

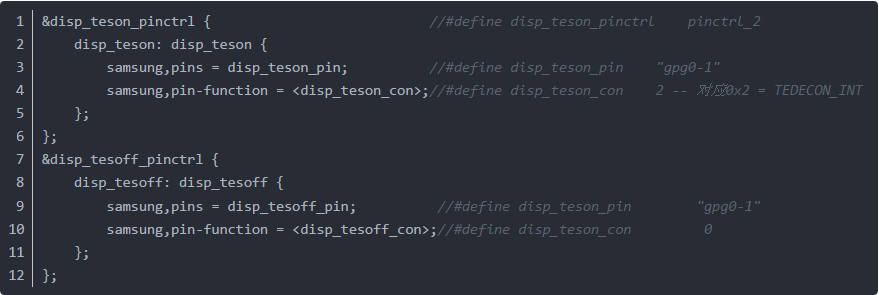
案例1：

xxx这个设备要用到gpg0\_1这个pin的TE\_DECON\_INT功能，并分别将这两个状态取了个名字turnon\_tes和turnoff\_tes.这个名字是随便起的。重点是看pinctrl-0和pinctrl-1，根据示例，它们分别引用了disp\_teson和disp\_tesoff这两个节点。



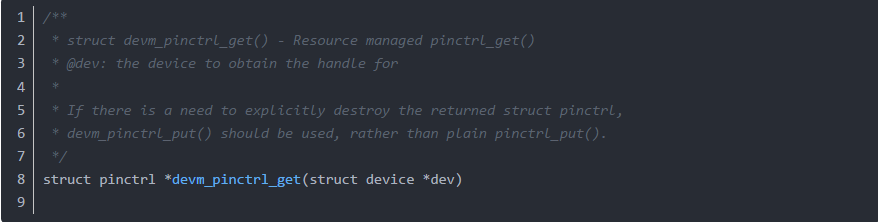


两个重要的属性必须有：pins 和 pin-function分别是pin的名字和要把pin配置成什么功能，还有gpg0属于pinctrl\_2,所以这个地方引用的是pinctrl\_2而不是其他。

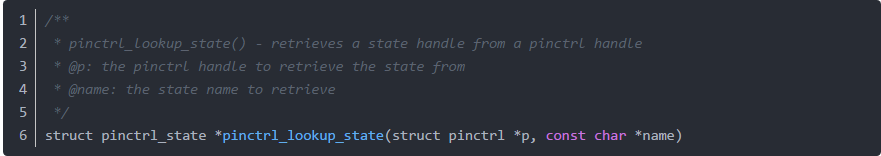


那么driver如何去操作这个pin呢？首先需要大家熟悉几个内核的API：

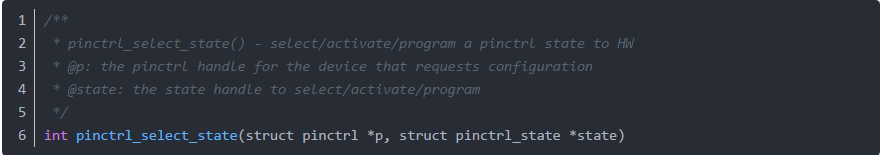
1. 获取一个pinctrl句柄，参数是dev是包含这个pin的device结构体即xxx这个设备的device



2. 获取这个pin对应pin\_state（引脚状态-turnon\_tes/turnoff\_tes）



3. 设置引脚为为某个stata -- turnon\_tes/turnoff\_tes



具体操作：

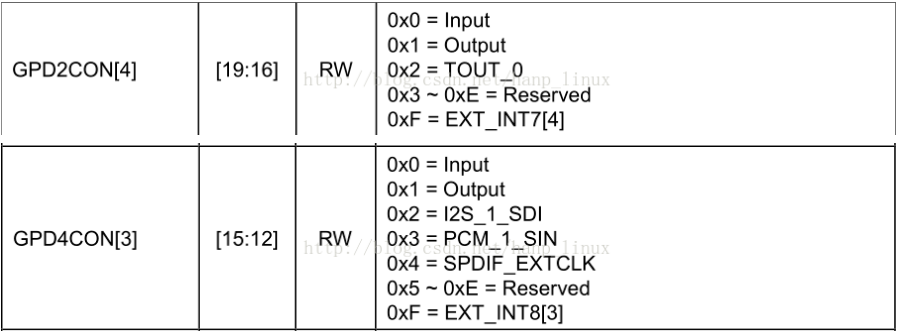


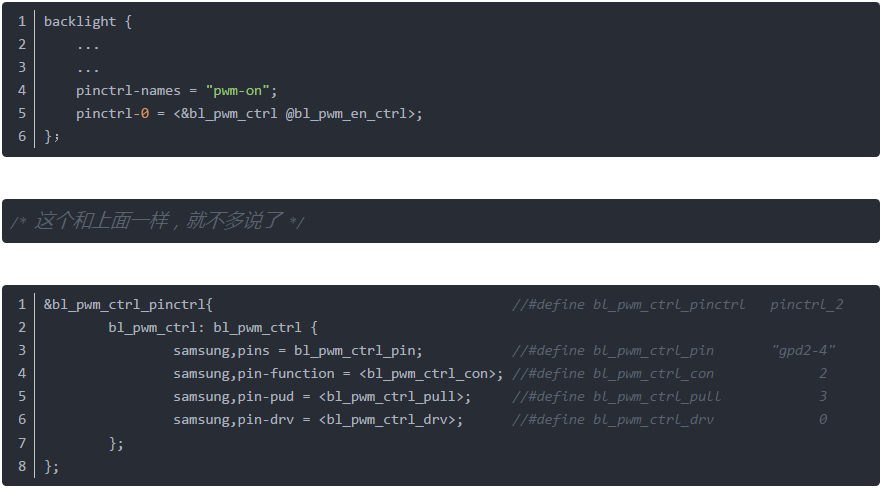
经过以上操作，gpg\_1引脚对应的con寄存器的对应的位域被配置成2，即0x2 = TE\_DECON\_INT功能。同理，根据此方法也可以设置turnoff\_tes的状态。

案例2 -- 一个背光灯device需要使用pwm的输出pin：

device tree：

背光系统中要用到gpd2\_4这个pin的TOUT\_0功能和gpd4\_3这个pin的输出功能并输出1，需要在backlight这个node中做以下描述，这两个pin只有一个状态（pwm-on），同样，这个名字也是可以随便起的。bl\_pwm\_ctrl和bl\_pwm\_en\_ctrl分别是对这两个pin的描述。

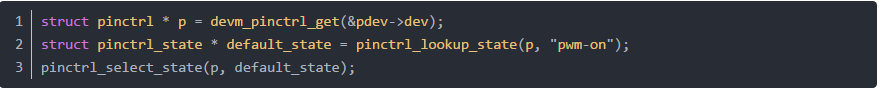




这个描述比上面多了个pin-val，因为这个引脚不仅要配置成输出功能，还要输出1，所以pin-val = 1。

driver的操作：

在backlight的driver的probe中：



执行完以上操作，可以发现gpd2\_4引脚被配置成了TOUT\_0功能，gpd4\_3引脚被配置成为了输出功能，并且输出1（高电平）。

以上就是pinctrl子系统的应用实例。如果有解释不太正确的地方请指教。

应用实例：6739 指纹 tp

## 18、ADB查看I2C挂载情况

使用ADB，当不确定I2C设备是否挂载成功可以使用以下命令：

adb shell

cd /sys/bus/i2c

cd devices

ls

以上命令均可在CMD中输入，以上命令输入结束后可以看到I2C各条总线上挂载的各个设备，

接着使用以下命令可以查看设备名：

cd 0-0001

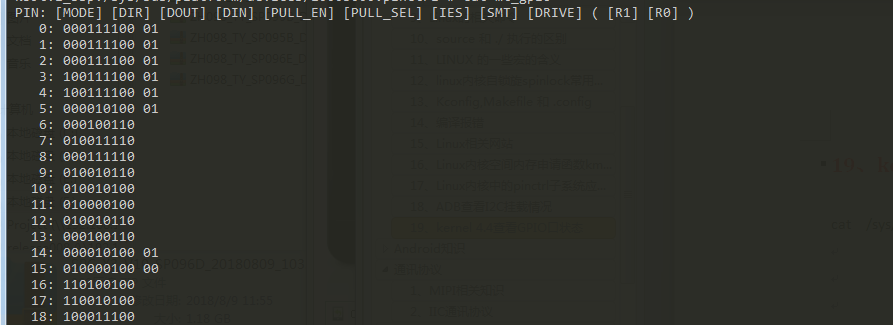
ls

cat name

至此可以看到设备名。

## 19、kernel 4.4查看GPIO口状态

cat /sys/bus/platform/devices/1000b000.pinctrl/ mt\_gpio



cat mt\_gpio

echo "mode 60 0” > mt\_gpio //set gpio60 mode0

echo "dir 60 1” > mt\_gpio //set gpio60 0:input;1,output

echo "out 60 1” > mt\_gpio //set gpio60 output value: 0,low;1,high

echo “pullen 60 1” > mt\_gpio //set gpio60 1,pull enable;0, pull disable

echo “pullsel 60 0” > mt\_gpio //set gpio60 1,pullup;0,pull down

20、阅读字符设备驱动的开发与详解笔记

## 20、linux内核几种打印调试信息的方法

**（1）、pr\_debug需要怎样设置才会打印消息（一般打印调试时候才需要的信息）**

1）在Makefile里面增加 EXTRA\_CFLAGS += -DDEBUG 即可

可以有如下用法“

#define fpsensor\_debug(fmt, args...) pr\_debug("[DBG] "fmt, ##args)

**（2）、pr\_info**

#define pr\_info(fmt, ...) \

printk(KERN\_INFO pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)

**（3）、pr\_err**

#define pr\_err(fmt, ...) \

printk(KERN\_ERR pr\_fmt(fmt), ##\_\_VA\_ARGS\_\_)

## 21、SELinux学习笔记

**小知识：**

1、 SELinux：SELinux(Security-Enhanced Linux) 是美国国家安全局（NSA）对于强制访问控制的实现，是 Linux历史上最杰出的新安全子系统。NSA是在Linux社区的帮助下开发了一种访问控制体系，在这种访问控制体系的限制下，进程只能访问那些在他的任务中所需要文件。SELinux 默认安装在 Fedora 和 Red Hat Enterprise Linux 上，也可以作为其他发行版上容易安装的包得到。

2、除了SELinux之外还有另一种对权限的控制（自主访问控制(DAC)），其实这个就是我们创建文件的时候设置的权限，还可以手动使用chmod 777 文件 修改权限。系统的访问权限是由DAC和SElinux共同完成缺谁都访问不了。

**3、参考博客：https://blog.csdn.net/gulinxieying/article/details/78677139**

**使用adb命令开关SELinux：**

adb shell setenforce 0 关闭SElinux

adb shell setenforce 1 开启SElinux

**使用adb命令确认SElinux是否开启：**

adb shell getenforce 返回 Enforcing 表示开启

返回 Permissive 表示关闭

其中enforcing,permissive 分别对应 1, 0

**语法规则：**

**（1）、访问向量规则**，访问向量规则又称AV规则，属于SElinux策略语言的控制部分。常用的通用AV规则有如下4个(详细解释https://blog.csdn.net/gulinxieying/article/details/78677139)：

allow：表示允许主体对客体执行允许的操作

dontaudit：表示不记录违反规则的决策信息，且违反规则不影响运行(允许操作且不记录)

auditallow：表示允许操作并记录访问决策信息(允许操作且记录)

neverallow：表示不允许主体对客体执行指定的操作

完整的基本安全控制语句格式为：

AV规则 主体 客体：客体类别 许可

eg: allow netd proc:file write

**（2）、SElinux策略文件（SElinux的策略文件主要集中在external\sepolicy下 展讯：device\sprd\sharkl3\common\sepolicy）：**

这里也是我们添加修改安全策略的主要地方我们会发现里面有很多.te文件，这些te文件就是我们定义的策略文件，除了策略文件之外，还有很多没有后缀的，这些没有后缀的文件就是专门定义策略语句元素的。由于所有的te文件中的规则语句都是基于这些语句元素组成的，这里我们先来看下这些定义性文件

所有的安全策略在编译后最终都会汇集到，out/target/product/generic/obj/ETC/sepolicy\_intermediates/policy.conf文件中（Android 8.0的位置变掉），

1）、定义性文件

users ：该文件中定义了SElinux中唯一的用户u：

roles ：该文件定义了SElinux中唯一的角色r：

security\_classes ：该文件定义了所有通用客体类别：

attributes ：该文件定义了所有SContext中的通用type：

access\_vectors ：该文件定义了资源的所有通用许可：

file.te ：该文件自定义了file\_contexts中的所有文件类型的type：

device.te ：该文件自定义了file\_contexts中的所有设备节点的type：

genfs\_contexts ：定义了虚拟文件的SContext：

fs\_use ：定义了所有文件系统类型的SContext：

service.te :：定义了所有service\_contexts中SContext的type：

service\_contexts ：定义了所有所有系统service的SContext：

mac\_permissions.xml ：与seapp\_contexts一起负责为接下来安装的不同类型的app添加SContext。

seapp\_contexts ：与mac\_permissions.xml一起负责为接下来安装的不同类型的app添加SContext。

成对出现的有：

\*device类型：

-- 类型定义: external/sepolicy/device.te; device/mediatek/common/sepolicy/device.te

-- 类型绑定: external/sepolicy/file\_contexts;device/mediatek/common/sepolicy/file\_contexts

\*File类型:

-- 类型定义: external/sepolicy/file.te;device/mediatek/common/sepolicy/file.te

-- 绑定类型: external/sepolicy/file\_contexts;device/mediatek/common/sepolicy/file\_contexts

\*虚拟File类型:

-- 类型定义: external/sepolicy/file.te;device/mediatek/common/sepolicy/file.te

-- 绑定类型: external/sepolicy/genfs\_contexts;device/mediatek/common/sepolicy/genfs\_contexts

\*Service类型:

-- 类型定义: external/sepolicy/service.te; device/mediatek/common/sepolicy/service.te

-- 绑定类型：external/sepolicyservice\_contexts;device/mediatek/common/sepolicy/service\_contexts

\*Property类型:

-- 类型定义: external/sepolicy/property.te;device/mediatek/common/sepolicy/property.te

-- 绑定类型: external/sepolicy/property\_contexts;device/mediatek/common/sepolicy/property\_contexts;

2）、其他te文件

剩余的.te文件中主要是各个进程的具体策略控制。

**（3）、如何配置SElinux**

在支持SElinux的Android版本中，有关selinux策略文件有两处，一个是google原生的，另一个是产品平台配置的，他们分别对应如下目录：

google原生目录：s912/external/sepolicy

平台配置目录：s912/device/amlogic/common/sepolicy

在编译时, 系统会以合并的方式，将平台配置目录下的policy 附加到Google 原生的policy 上，而非替换。一般情况下，不建议修改google原生目录下的策略控制。接下来看下如何配置平台的selinux。

找到平台下的BoardConfig.mk，其相关变量配置就是BOARD\_SEPOLICY\_DIRS，目的就是指定平台sepolicy的目录。BoardConfig.mk中直接包含了sepolicy的mkfile文件： include device/amlogic/common/sepolicy.mk打开该sepolicy.mk文件可以看到BOARD\_SEPOLICY\_DIRS变量的配置：BOARD\_SEPOLICY\_DIRS := device/amlogic/common/sepolicy

**（4）、SElinux调试方法**

**1）、selinux的编译**

首先我们关注下selinux的编译，大家知道编译整个Android工程是个很耗时的工作。为了避免耗时，在我们修改或者添加了selinux策略之后，建议先独自对sepolicy模块进行一次编译，如果通过了，然后再进行整个Android的编译。具体指令如下：如：mmm external/sepolicy

**2）、如何快速确认是selinux引起？**

目前所有的SELinux check 失败，在kernel log 或者android log(L版本后)中都有对应的" avc: denied"的LOG 与之对应。

标志性 log: avc: denied { 操作权限 } for pid=7201 comm=“进程名” scontext=u:r:源类型:s0 tcontext=u:r:目标类型:s0 tclass=访问类型 permissive=0

其中：

进程的scontext 结构： user:role:type[:range]

注释：

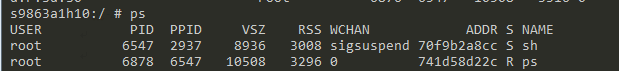
u——user，用户，SEAndroid中仅定义了一个SELinux用户，值为u。

r——role，role是角色之意，它是SELinux中一种比较高层次，更方便的权限管理思路，即Role Based Access Control（基于角色的访问控制，简称为RBAC）。简单点说，一个u可以属于多个role，不同的role具有不同的权限。RBAC我们到最后再讨论。

init——tyoe，代表该进程所属的Domain为init。MAC的基础管理思路其实不是针对上面的RBAC，而是所谓的Type Enforcement Accesc Control（简称TEAC，一般用TE表示）。对进程来说，Type就是Domain。比如init这个Domain有什么权限，都需要通过[例子1]中allow语句来说明。

S0——和SELinux为了满足军用和教育行业而设计的Multi-Level Security（MLS）机制有关。简单点说，MLS将系统的进程和文件进行了分级，不同级别的资源需要对应级别的进程才能访问。限于篇幅，这里不做过多探讨。

ps -Z



文件的scontext 结构： user:role:type[:range]

其中：

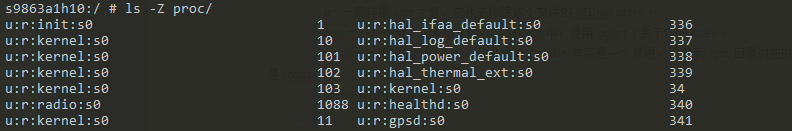
u——同样是user之意，它代表创建这个文件的SELinux user。

object\_r——role，这里指文件，在SELinux中，使用object\_r表示它的role。

rootfs——死的东西的Type，和进程的Domain其实是一个意思。它表示root目录对应的Type是rootfs。

s0——MLS的级别。

ls –Z



在相应源类型.te 文件，添加如下格式的一行语句：(结尾别忘了分号) 格式：“allow 源类型 目标类型:访问类型 {操作权限};” 或者叫做 “AV规则 主体 客体：客体类别 许可”

例如：

07-10 14:06:32.780 9043 9043 W com.filemanager: type=1400 audit(0.0:104): avc: denied { read } for name="u:object\_r:curef\_set\_prop:s0" dev="tmpfs" ino=1334 scontext=u:r:platform\_app:s0:c512,c768 tcontext=u:object\_r:curef\_set\_prop:s0 tclass=file permissive=0

可以得到如下 sepolicy：添加到 platform\_app.te 文件 allow platform\_app curef\_set\_prop : file { read };

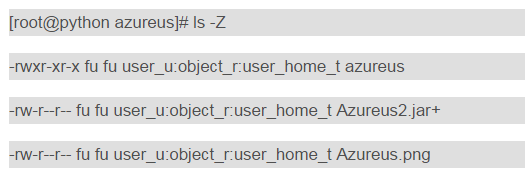
**（5）、常用指令**

SELinux 是个经过安全强化的Linux操作系统，实际上，基本上原来的运用软件没有必要修改就能在它上面运行。真正做了特别修改的RPM包只要50多个。像文件系统EXT3都是经过了扩展。对于一些原有的命令也进行了扩展，另外还增加了一些新的命令，接下来我们就来看看这些命令。

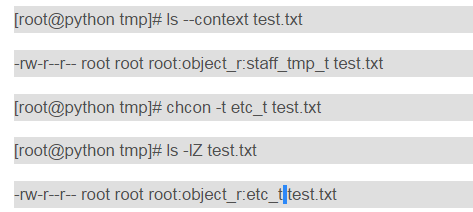
1）、文件操作

**ls命令（可以看到文件或者文件夹的标签）：**

在命令后加个 －Z 或者加 –context



**chcon**更改文件的标签



**restorecon** 当这个文件在策略里有定义是，可以恢复原来的 文件标签。

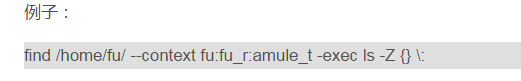
**setfiles**  跟chcon一样可以更改一部分文件的标签，不需要对整个文件系统重新设定标签。

**fixfiles**  一般是对整个文件系统的， 后面一般跟 relabel,对整个系统 relabel后，一般我们都重新启动。如果，在根目录下有.autorelabel空文件的话，每次重新启动时都调用 fixfiles relabel

**star**就是tar在SELinux下的互换命令，能把文件的标签也一起备份起来。

**cp**可以跟 -Z, --context=CONTEXT 在拷贝的时候指定目的地文件的security context

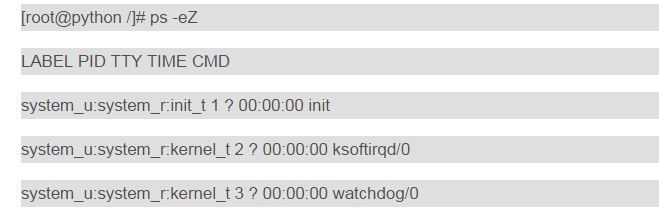
**find**可以跟 –context 查特定的type的文件。



**run\_init**在sysadm\_t里手动启动一些如Apache之类的程序，也可以让它正常进行，domain迁移。

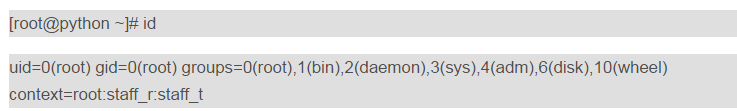
2）、进程domain的确认

程序现在在那个domain里运行，我们可以在ps 命令后加 －Z

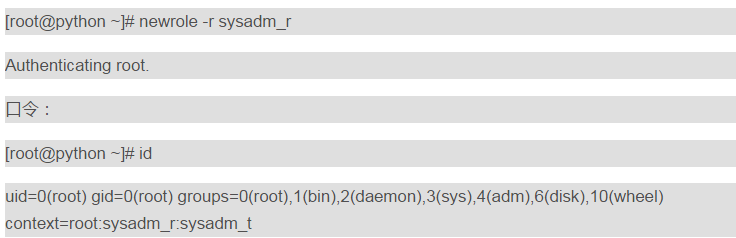


3）、ROLE的确认和变更

命令id能用来确认自己的 security context

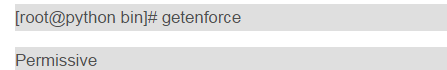


这里，虽然是ROOT用户，但也只是在一般的ROLE和staff\_t里运行，如果在enforcing模式下，这时的ROOT对于系统管理工作来说什么也干不了。



4）、模式切换

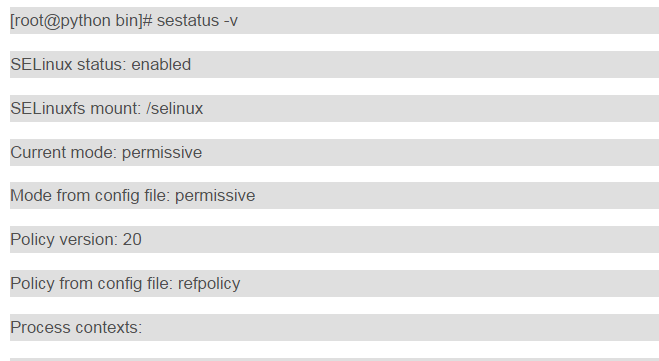
**getenforce**得到当前的SELINUX值



**setenforce**更改当前的SELINUX值 ，后面可以跟 enforcing,permissive 或者 1, 0。



**sestatus**显示当前的 SELinux的信息

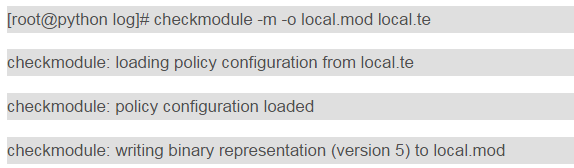


5）、其他重要命令

**Audit2allow**很重要的一个以python写的命令，主要用来处理日志，把日志中的违反策略的动作的记录，转换成 access vector，对开发安全策略非常有用。在refpolicy里，它的功能比以前有了很大的扩展。



**checkmodule**编译模块



**semodule\_package**创建新的模块



**semodule**可以显示，加载，删除 模块



**semanage**  这是一个功能强大的策略管理工具，有了它即使没有策略的源代码，也是可以管理安全策略的。因为我主要是介绍用源代码来修改策略的，详细用法大家可以参考它的man页。

**（6）、参考文献**

1. Security-Enhanced Linux in Android

<https://source.android.com/security/selinux/index.html>

2. 深入理解SELinux SEAndroid

<http://blog.csdn.net/innost/article/details/19299937>

3.SElinux详解.pdf

4.类型强制（TE编写规则）

http://blog.csdn.net/myarrow/article/details/10105961

# Android知识

## 开机的时候一直在动画界面开不了机。

### 平台：SC7731C、SC9832 、SC9850

查找问题方法：通过usb进去adb，通过打印底层log或者上层的log，看是否有异常的问题存在。

解决方法：通过查看上层log发现一直在报错说找不到任何的PackageInstaller，所以app当然也就装不上；所以Config.mk中加上 ZYT\_GMS\_PACKAGES=GooglePackageInstaller （当然不一定要是谷歌的，360或者原生的皆可以。）既可以解决问题。

## 2、Android 编译与下载

### 编译

修改哪个文件夹就往外退，直到看到Android.mk文件的目录，在这个目录下面执行 mm进行编译。

### 编译生成路径

idh.code\out\target\product\sp9850ka\_1h10\system\app

idcode\out\target\product\sp9850ka\_1h10\system\priv-app

一般情况下是在这两个目录中的一个apk所在目录的名字和刚刚编译的时候Android.mk所以的目录一致。

### 下载

adb root

adb remount

adb push apk路径

adb reboot

## 3、发送广播

adb shell am broadcast -a com.dfl.receive.on

adb shell am broadcast -a com.dfl.receive.off

## 4、Android 编译环境搭建——build/envsetup.sh

准备好编译环境后，编译Rom的第一步是 source build/envsetup.sh，该步骤把e\nvsetup.sh里的函数声明为当前会话终端可用的命令。这些命令能让我们切换目录，提交代码，编译Rom更方便。如果记不住所有命令,只要你记住hmm就可以了，也可通过hmm命令看到支持的命令列表。

**1. 命令分类:**

1.1 编译用的命令



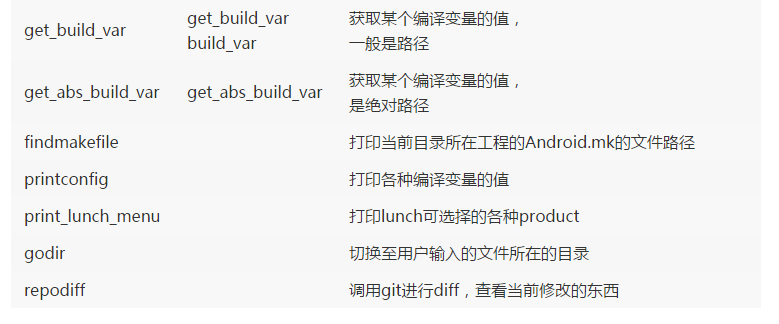






1.2 查看代码时的辅助命令





1.3 辅助函数

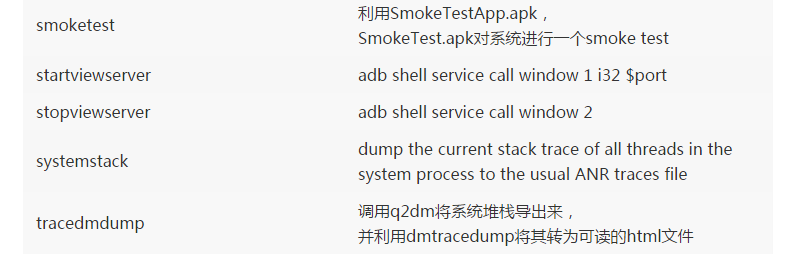






1.4 调试相关





1.5 提交代码相关命令



**2. source build/envsetup.sh 执行流程**

envsetup.sh 定义了很多函数，除此之外还执行了其它操作，代码如下:

VARIANT\_CHOICES=(user userdebug eng)，# TARGET\_BUILD\_VARIANT变量的可能值

#LUNCH\_MENU\_CHOICES是供用户选择的prodcut列表,

#每次source build/envsetup.sh时需重置变量LUNCH\_MENU\_CHOICES

#不然后续的include vendor/cm/vendorsetup.sh时会继续添加产品至变量LUNCH\_MENU\_CHOICES里，

#导致出现很多重复产品

unset LUNCH\_MENU\_CHOICES

add\_lunch\_combo full-eng #默认添加full-eng，full\_mips-eng等4个产品

add\_lunch\_combo full\_x86-eng

add\_lunch\_combo vbox\_x86-eng

add\_lunch\_combo full\_mips-eng

alias bib=breakfast#给breakfast起别名

complete -F \_lunch lunch #给lunch添加tab提示

case `uname -s` in #定义sgrep函数 在所有工程类型代码里搜索

Darwin)

function sgrep()

{

find -E . -name .repo -prune -o -name .git -prune -o -type f \

-iregex '.\*\.(c|h|cpp|S|java|xml|sh|mk)' -print0 \

| xargs -0 grep --color -n "$@"

}

;;

\*)

function sgrep()

{

find . -name .repo -prune -o -name .git -prune -o -type f \

-iregex '.\*\.\(c\|h\|cpp\|S\|java\|xml\|sh\|mk\)' -print0 \

| xargs -0 grep --color -n "$@"

}

;;

esac

export -f cmremote

export -f aospremote

alias mmp='dopush mm' #定义更多编译后push到设备的函数的操作

alias mmmp='dopush mmm'

alias mkap='dopush mka'

alias cmkap='dopush cmka'

if [ "x$SHELL" != "x/bin/bash" ]; then #只支持Bash终端

case `ps -o command -p $$` in

\*bash\*)

;;

\*)

echo "WARNING: Only bash is supported," \

"use of other shell would lead to erroneous results"

;;

esac

fi

#Execute the contents of any vendorsetup.sh files we can find.

#source vendor和device下能找到的所有vendorsetup.sh

for f in `/bin/ls vendor/\*/vendorsetup.sh vendor/\*/\*/vendorsetup.sh device/\*/\*/vendorsetup.sh 2> /dev/null`

do

echo "including $f"

. $f

done

unset f

#source目录 sdk/bash\_completion vendor/cm/bash\_completion下的bash脚本，

#能提供tab提示

addcompletions

export ANDROID\_BUILD\_TOP=$(gettop)

2.1 执行的vendorsetup.sh有：

在envsetup.sh里将执行vendor和device目录及各自子目录下所有的vendorsetup.sh，这些vendorsetup.sh做的事情是调用add\_lunch\_combo将它们各自的产品添加到 LUNCH\_MENU\_CHOICES 变量里

#执行cm的vendorsetup.sh将从网上下载cm支持的产品列表，并添加至LUNCH\_MENU\_CHOICES

vendor/cm/vendorsetup.sh

#将添加mini\_armv7a\_neon产品 add\_lunch\_combo mini\_armv7a\_neon-userdebug

device/generic/armv7-a-neon/vendorsetup.sh

#add\_lunch\_combo mini\_armv7a-userdebug

device/generic/armv7-a/vendorsetup.sh

#add\_lunch\_combo mini\_mips-userdebug

device/generic/mips/vendorsetup.sh

#add\_lunch\_combo mini\_x86-userdebug

device/generic/x86/vendorsetup.sh

#add\_lunch\_combo cm\_jflteatt-eng

device/samsung/jflteatt/vendorsetup.sh

# add\_lunch\_combo full\_panda-userdebug

device/ti/panda/vendorsetup.sh

# add\_lunch\_combo zte\_blade-eng

#add\_lunch\_combo zte\_blade-userdebug

device/zte/blade/vendorsetup.sh

2.2 执行的completion bash有：

在envsetup.sh里将执行sdk/bash\_completion和vendor/cm/bash\_completion目录下的bash脚本，这些bash脚本主要是为命令提供tab支持，有了这些tab支持，输入命令后如果某个选项忘记了，只需要敲tab键，就能获得提示，使用命令更加方便

including sdk/bash\_completion/adb.bash

including vendor/cm/bash\_completion/git.bash

including vendor/cm/bash\_completion/repo.bash

分别对应adb，git，repo的tab提示

## 5、logcat为什么可以输出上层的log

背景：在default状态下调用printf等std C/C++接口输出的log不会被打印到eclipse的logcat中，但是android提供了\_\_android\_log\_print这个函数可以将log重定向到eclipse的logcat中。

1)

在对应的mk文件中加入:LOCAL\_LDLIBS := -llog

2)

在要使用LOG的cpp文件中加入：

#include <android/log.h>

使用方法如下;

\_\_android\_log\_print(ANDROID\_LOG\_DEBUG,"Tag", \_\_VA\_ARGS\_\_)

ANDROID\_LOG\_DEBUG: log的level

Tag：module的tag

\_\_VA\_ARGS\_\_:格式化参数列表

例子：

\_\_android\_log\_print(ANDROID\_LOG\_DEBUG,"libAirplay", "service URI : %s", "www.163.com.video.xixi.mp4");

3)

直接使用并不是个好的想法，如果能够可以和eclipse一样实现分等级打印，那还不错；实现如下：

#define LOGV(...)\_\_android\_log\_print(ANDROID\_LOG\_VERBOSE, " Tag", \_\_VA\_ARGS\_\_) // VERBOSE

#define LOGD(...) \_\_android\_log\_print(ANDROID\_LOG\_DEBUG , " Tag ", \_\_VA\_ARGS\_\_) // DEBUG

#define LOGI(...) \_\_android\_log\_print(ANDROID\_LOG\_INFO , " Tag ",\_\_VA\_ARGS\_\_) // INFO

#define LOGW(...) \_\_android\_log\_print(ANDROID\_LOG\_WARN , " Tag ", \_\_VA\_ARGS\_\_) //WARN

#define LOGE(...) \_\_android\_log\_print(ANDROID\_LOG\_ERROR , " Tag ",\_\_VA\_ARGS\_\_) // ERROR

注意：括号中是省略号，不用改为参数.

在需要的地方直接使用

LOGV(“www.163.com.video.xixi.mp4”);

或者LOGD(“www.163.com.video.xixi.mp4”); 就好

# 通讯协议

## 1、MIPI相关知识

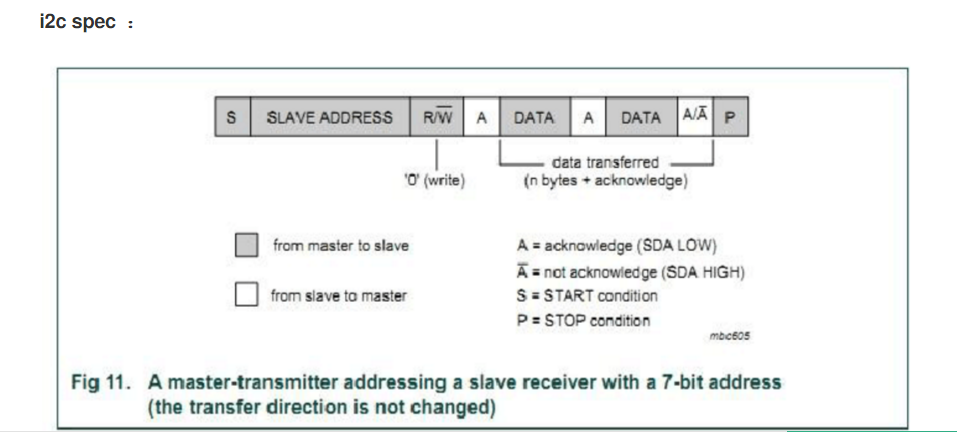
（1）、CSI和DPI分别代表什么：

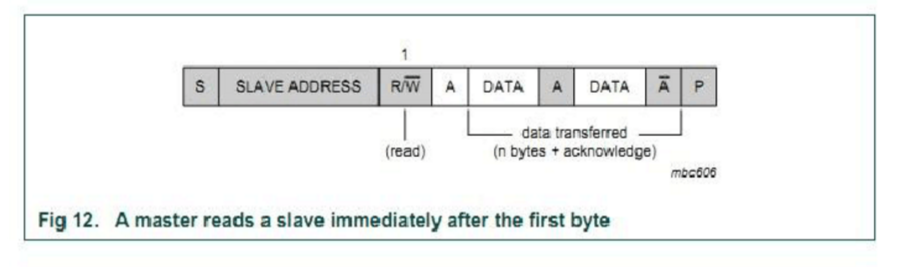
MIPI CSI和DPI是MIPI标准的一种，因为MIPI在移动领域应用范围太广了，各种外围设备都可以用它来传输信息，所以，MIPI联盟给不同的外设接口定义了版本名，CSI是for Camera的，DPI是for Display的。而且这些标准不单包括物理层的时序定义，还包括上层的传输协议/数据处理协议和应用层协议。

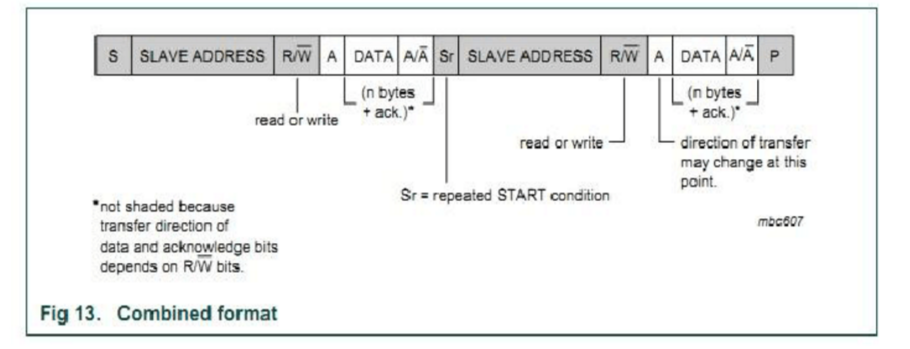
## 2、IIC通讯协议

**1）、I2C通讯及波形讲解**

**概念：**







The Acknowledge signal is defined as follows: the transmitter releases the SDA line (Wei: put SDA at 1 state)  during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it

remains stable LOW during the HIGH period of this clock pulse  (Wei: if the high of clock is hold on, SDA is kept at LOW!! )(see Figure 4). Set-up and

hold times (specified in Section 6) must also be taken into account.

**I2C设备调试及波形分析**

最新在开发OLED屏驱动，需要用到I2C总线，下面大体上讲解一下I2C设备的调试及波形分析，为大家做一些参考，由于刚涉及这部分内容，因此有什么错误的，还请赐教~

 一、概要

I2C总线只需要两条线，一条SDA数据线，一条SCL时钟线；根据这两条线的高低电平、上升沿、下降沿就可以实现主机与I2C设备的通讯；其中有：

（1）I2C总线相关

传输开始条件：SCL处于高电平，SDA下降沿时； 传输接收条件：SCL处于高电平，SDA上升沿时；

传输数据：开始传输后，SCL处于高电平时，SDA的数据为所传输的数据；

回应：当传输完一个字节后，I2C设备需要回应一个ACK，这样主机才继续发送；因此回应信号是在传输完8bit后的下一个数据位（SDA值），当SDA为0表示有回应，为1表示没回应；

正常I2C总线的数据是：Start + I2C devece id + R/W + ACK + Data（first byte）+ ACK + ... + Data（n）+ ACK + Stop

（2）I2C设备相关

设备地址：有7位和10位两种，具体见I2C设备芯片的DataSheet，由于目前用到的是7位，因此下面主要针对7位讲述；在讲I2C设备地址是有可能有两种说法，主要是用8位表示还是用7为表示，比如对于我的OLED来说，当用8位表示时则为0x78地址，当用7位时则为0x3c（即0x78右移1为），在驱动中用0x78还是用0x3c要看具体平台的I2C总线驱动，我在AMLOGIC平台上用的是0x78，而在MV平台上用的是0x3c；

寄存器reg：一般的I2C设备芯片都有带reg，一般在传输正式的数据之前需要先传输reg地址，比如我的OLED来说，在传控制命令时需要先发送0x00的reg地址，在传输数据时需要发送0x40的reg地址；

**二、调试及波形分析**

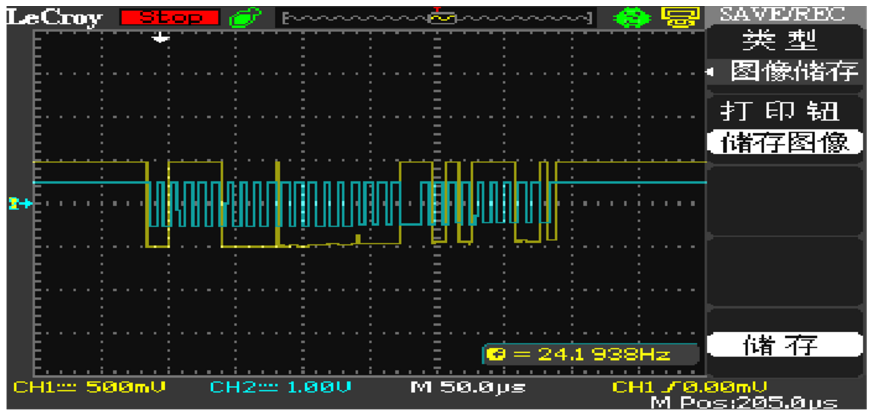
一般当我们拿到一个I2C设备时，就必须涉及到驱动的编写，就比如对于OLED来说，就要用编写OLED驱动，这样我们才能控制它，对于OLED屏来说，第一步也是最重要的一步就是点亮它；当我们做完这一步，那后面剩下的就只是细节问题了；”万事开头难“，这句话真的不假，对于OLED来说，如何才能点亮，我们该怎么调试呢？当我们写完OLED驱动，但OLED屏还是不亮，可能问题会出现在哪？是硬件问题还是软件问题？；若为软件问题，那会是I2C总线驱动问题，还是我们I2C设备驱动有问题？那么我们该如何判断问题出现位置呢？这就需要我们对I2C总线上的数据进行分析；那么下面我将详细讲述如何获取和分析I2C总线上的数据；

1）示波器

对于I2C总线的数据，我们要用到示波器，这样我们才能抓取到信号，而且必须同时采集SDA和SCL的数据；该如何抓取呢？我这边的方式是将示波器调成边下降沿触发模式（因为开始信号是SDA下降沿），并且设置成单次模式（这样抓取完一次就会stop，便于我们数据分析）；

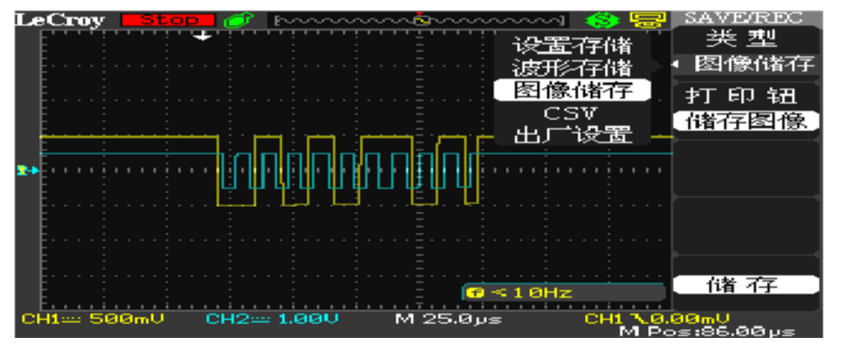
（2）波形

由于我的OLED设备的地址是0x78(8位)，而第一次我必须将OLED设置成off状态，通过命令表可以查到，必须发生0xAE，而刚刚有讲到在发送命令前必须先发送寄存器reg地址，即0x00；于是我发送的数据为0x78+0x00+0xAE；下面是我用示波器抓取到的波形：



分析（其中黄色的为SDA信号，浅蓝色为SCL信号）：从上述波形中我们可以读取到数据依次是：01111000 0 00000000 101011100；即：0111100(7位OLED设备地址) + 0( 读写为，0为写，1为读)+0（ACK回应）+00000000（寄存器reg）+ 10101110(0xAE OLED off命令)；（这里还有一点我还没弄明白就是reg后没有回应，还望大家不吝赐教）

下面当我来随便发送一个地址时，比如当我发送0x56（8位地址），测到的波形如下：



分析：从波形可看出数据依次为：010101101 即0101011（设备地址）+0（读写位）+1（NACK无应答），由于没有相应的I2C设备因此无应答ACK信号，于是就停止传输；

三、总结

由以上可知，可以通过查看I2C总线数据来确认跟踪I2C设备，已更好地确定是软件问题还是硬件问题；

**用示波器对单片机I2C时序进行图形波形分析的试验小结**

I2C要求要有一个主设备，负责发起请求和控制时钟；其它为从设备，通过设备ID地址来识别并响应主设备请求。主从设备要轮流控制SDA。一开始我没搞明白这一点，直接加了写I2C数据代码，然后用示波器在SDA和SCL脚测量，却只能找到些凌乱的波形，没有预期的效果。后来把从设备接上，两边写好代码，互相有了响应，这才在示波器上看到波形。

这里我找了一个主设备往从设备写数据的例子，代码如下：

char buf[128]; int len;

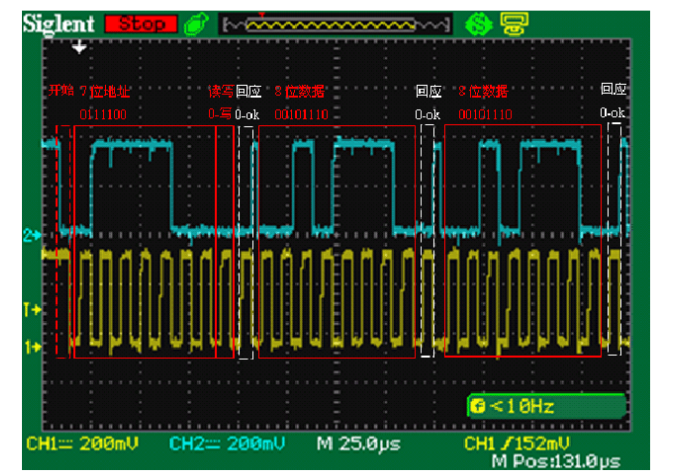
strcpy(buf,"..huz\_hello\_i2c/n"); len=strlen(buf);

//deviceid: 0x3c

write\_i2c(0x3c, buf , len);

接收端的代码比较简单，就不贴了。

将示波器的X和Y分别接到SDA和SCL，得到波形并分析如图：



从图中可知时序如下：

1. 由主机发起，在SCL为高电平时，SDA由高到低切变，形成开始信号；

2.

接着是7位地址和一位读写标志，这里7位地址为0111100，即0x3c，正是我们代码中设置的地址ID；最后一位为0表示写操作；

3. 接着在下一个时钟，主机以高电平状态释放SDA，这时从机响应，将SDA拉低了； 4. 接着是两个8位数据00101110与响应，即0x2E，正是“.”号的ASCII码，符合预期输出； 5.

还有其它数据和最后的停止位，图中被截掉了。

从图中可知，纵向一格是200mV，则SDA和SCL的电平大概就是350mV；由于信号笔上设置了信号x10，因此实际电平应该大概是3.5V（理论上应该是3.3V）。横向一格是25us，10个时钟周期大概用了4格，即4x25us=100us，平均每个时钟周期是10us，可算出传输频率为1/10us=100,000/s，即100k bps。

另外，对于读从设备内容，基本流程是主设备先往从设备写一个命令，然后再输出读取命令，然后才由从设备发送数据。过程类似，不再具体分析了。

下图示例中，主机先向从机写了一个地址命令，然后重新开始并进入读取周期。



# ADB命令

1、adb操作I2C

# C语言知识

## （1）sprintf

注意：会在末尾自动加上\0

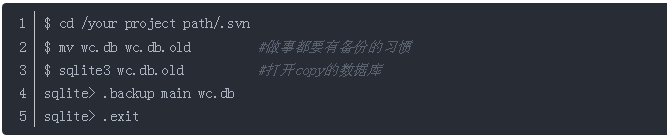
（2）对于宏的操作

1)、#undef A 不定义宏A

# SVN使用

## svn 报错

### （1）、svn: E200033: database is locked



cd /your project path/.svn

mv wc.db wc.db.old

sqlite3 wc.db.old

sqlite> .backup main wc.db

sqlite> .exit

# GitHub知识小计

## 下载方法：

Mac：https://sourceforge.net/projects/git-osx-installer/

Windows：https://git-for-windows.github.io/

Linux：apt-get install git

## 1、相关文档

learn-github-from-zero.pdf

## 知名GIT账号

Google: https://github.com/google

苹果: https://github.com/apple

Facebook: https://github.com/facebook

T witter：https://github.com/twitter

微软：https://github.com/microsoft

Square：https://github.com/square

阿里：https://github.com/alibaba

Linux：https://github.com/torvalds/linux

Rails：https://github.com/rails/rails

Nodejs：https://github.com/nodejs/node

Swift：https://github.com/apple/swift

CoffeeScript：https://github.com/jashkenas/coffeescript

Ruby：<https://github.com/ruby/ruby>

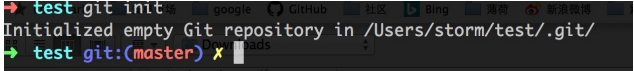
Linux 发明者 Linus T orvalds：https://github.com/torvalds

安卓JakeWharton：https://github.com/JakeWharton

Rails 创始人 DHH：https://github.com/dhh

## 3、相关命令

**git init**  初始化 git 仓库



可以看到初始化成了，至此 test 目录已经是一个 git 仓库了。

**git status** 查看状态 相当于svn的svn status

**git add 文件** 把相应的文件加入到git的本地仓中

**git commit** git commit -m 'first commit' ，这个命令什么意思呢？ commit 是提交的意思，-m 代表是提交信息，执行了以上命令代表我们已经正式进行了第一次提交。

## 4、与远程库之间的操作

**（1）、设置邮箱和用户名（没有设置提交的时候有问题）：**

git config --global user.name "yourname" 注释：这个名字要和GitHub上一样

git config --global user.email“your@email.com" 注释：邮箱设置一个自己的邮箱即可

如果忘记自己的用户名和邮箱可以通过指令查看：

git config --global user.name

git config --global user.email

**（2）、创建公钥私钥**

ssh-keygen -t rsa -C "username" (注：username为你git上的用户名)

注释：

Generating public/private rsa key pair.

Enter file in which to save the key (/Users/username/.ssh/id\_rsa): //这里的username是电脑上的用户名，这个地址也是文件的存储地址，然后我们按

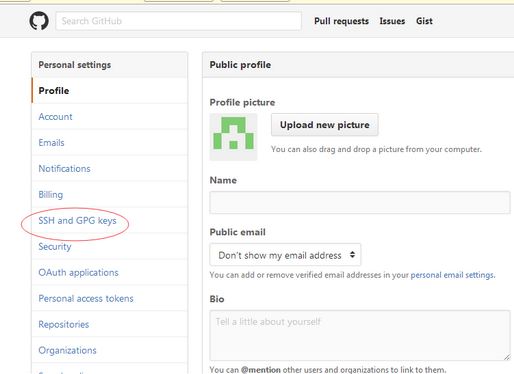
回车，如果你以前有存储地址会返回/Users/your username/.ssh/id\_rsa already exists.Overwrite (y/n)?直接输入y回车。如果以前没有储存地址就会出现

Enter passphrase(empty for no passphrase);也直接回车，两种情况回车后都会出现 Enter same passphrase again 然后接着回车会显示一长串内容其中

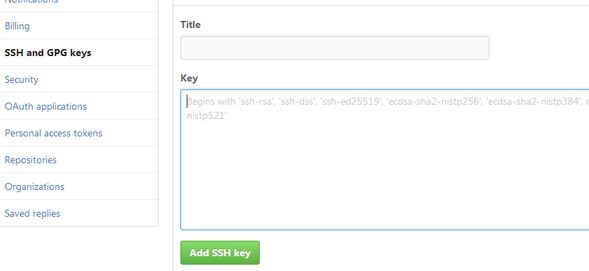
还有一些..o.. o oo .oS. 之类的代码，这说明SSH key就已经生成了。文件目录就是：username/.ssh/id\_rsa.pub.

**（3）、然后找到.ssh文件夹（一般路径：/Users/username/.ssh 也可以看提示）；在.ssh文件夹下面会有两个文件** **id\_rsa和****id\_rsa.pub 其中 id\_rsa是私钥 id\_rsa.pub 是公钥；打开 id\_rsa.pub 复制里面的公钥。**

**（4）、打开https://github.com/，登陆你的账户，进入设置（Settings）找到**



然后将你复制的内容粘贴到key中，其中title可以不用写



再点击Add SSH Key

**（5）、测试是否连接GitHub成功**

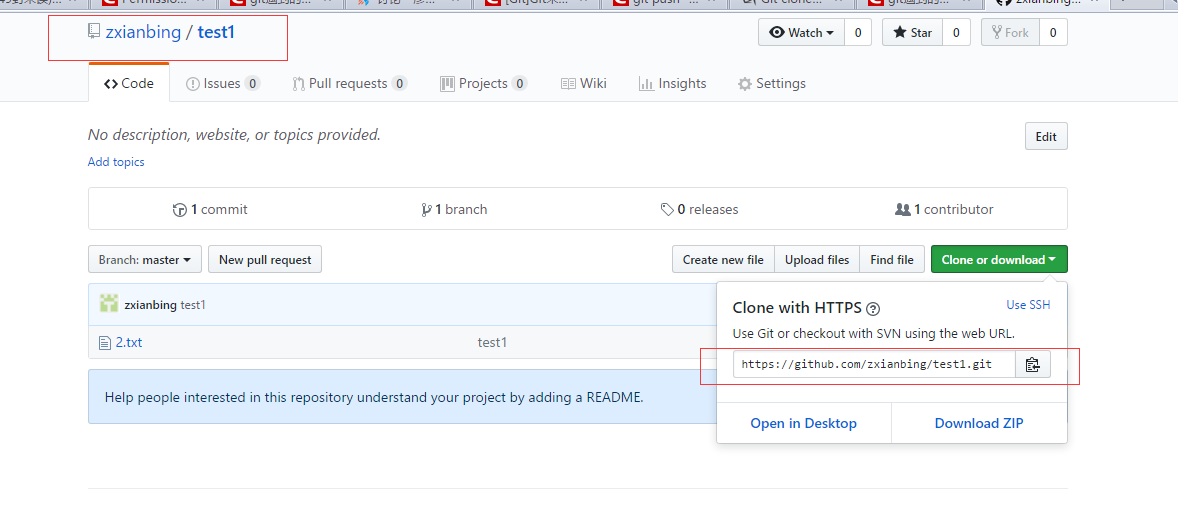
ssh -T [git@github.com](mailto:git@github.com) 注意:执行这个语句有的是直接提示成功，有些会让你输入yes或no 直接输入yes即可。

**（6）、建立本地库和远程库的连接 （执行这步后会在.ssh文件夹下生成****文件）**

1）从GitHub直接克隆一个库到本地

git clone [git@github.com:stormzhang/test.git](mailto:git@github.com:stormzhang/test.git)

注释：其中后面的是相应项目的地址



经过验证从里面得到的地址是<https://github.com/zxianbing/test1.git> 需要手动改为 [git@github.com:zxianbing/test1.git](mailto:git@github.com:zxianbing/test1.git) 格式

2）把本地已有的代码提交到远程库

第一步就是在 GitHub 上建一个 test 项目。

第二步把本地 test2 项目与 GitHub 上的 test 项目进行关联，切换到 test2 目录，执行如下命

令：

git remote add origin [git@github.com:stormzhang/test.git](mailto:git@github.com:stormzhang/test.git) 同样后面的地址和上面的获取方法是一样的。当关联后就可以做push pull的相关操作了。

（7）、如果怀疑自己公钥有问题，需要先删除文件，然后按上面步骤重新再来一次。

## 5、git 的一些配置

（1）、git log 配置为中文

git config --global i18n.commitencoding utf-8

git config --global i18n.logoutputencoding utf-8

（2）、git status 中午乱码

git config --global core.quotepath false

# cmder 使用

## 1、快捷键

（1）、win +Alt+p 打开setting

（2）Ctrl+T 建立新页签；

（3）Ctrl+W 关闭页签;

（4）Ctrl+Tab 切换页签

（5）Alt+F4 关闭所有页签

（6）Alt+Shift+1 开启cmd.exe

（7）Alt+Shift+2 开启powershell.exe

（8）Alt+Shift+3 开启powershell.exe (系统管理员权限)

（9）Ctrl+1 快速切换到第1个页签

（10）Ctrl+n 快速切换到第n个页签( n值无上限)

（11）Alt + enter 切换到全屏状态

（12）Ctr+r  历史命令搜索;

（13）End, Home, Ctrl : Traversing text with as usual on Windows

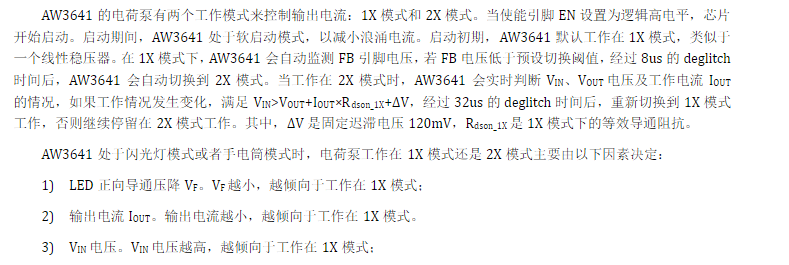
# 展讯MTK通用

## 1、闪光灯 AW3641

### （1）、引脚定义及功能 ：

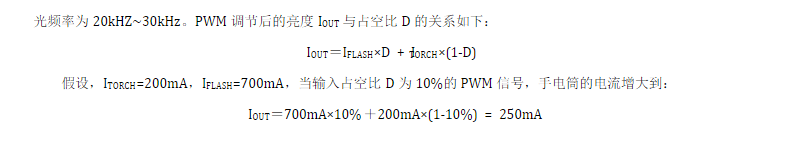


### （2）、自适应电荷泵



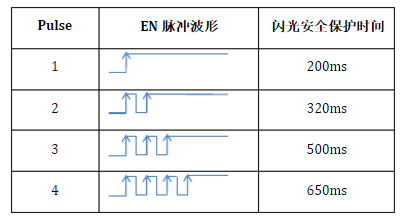
### （3）、PWM调光



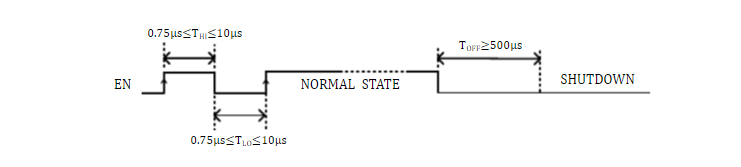


### （4）、闪光安全保护时间设置

在闪光灯模式下，为了防止故障情况下闪光灯持续大电流工作造成可靠性风险，同时保证不同应用条件下闪光灯的曝光时间满足应用要求，AW3641内置了闪光安全保护时间调节功能。闪光灯模式下，默认的闪光时间设置是200ms。在EN引脚输入一线脉冲信号，可以调节闪光安全保护时间。一线脉冲上升沿的个数和闪光安全保护时间的对应关系如下表



下图为一线脉冲的时序要求，其中THI为脉冲的高电平宽度，推荐值为2μs；TLO为脉冲的低电平宽度，推荐值为2μs；为一线脉冲的时序要求，其中THI为脉冲的高电平宽度，推荐值为2μs；TLO为脉冲的低电平宽度，推荐值为2μs；为一线脉冲的时序要求，其中THI为脉冲的高电平宽度，推荐值为2μs；TLO为脉冲的低电平宽度，推荐值为2μs；



### （5）、过温保护

