一：

内核：启动应用程序

应用程序：读写文件 获取按键

应用：opne read write 实质是： swi value 会引发异常，也就是中断调用不同的函数。进入内核空间

系统调用 sys\_read sys\_write

VFS

驱动: led\_opne led\_read led\_write 内核

应用程序如何找到对应的驱动程序，因该有自己的驱动框架。

二：字符程序的驱动框架：

写出的驱动告诉内核？把这个file\_operations结构体告诉内核

调用此函数major = register\_chrdev(0, "first\_drv", &first\_drv\_fops); // 注册, 告诉内核

谁调用它这个函数就叫做驱动的入口 first\_drv\_init（）

但是为了区分不同的入口，会有不同的修饰：

module\_init(first\_drv\_init);// module\_init 这个宏 是个结构体 里面的函数指针 指向入口函数。

open(/dev/xxx);//设备属性 ：主设备号 从设备号

内核

应用程序靠什么找到内核的驱动？

设备类型 设备号

内核里面有个数组，根据主设备号找到file\_operations(里面有.open .write)

在入口函数调用register\_chrdev（主设备号，名字， 结构体）//将结构体放入内核数组

对应的出口函数则需要从内核数组中移除

驱动

Makefile:

KERN\_DIR = /work/system/linux-2.6.22.6 // KERN\_DIR内核目录我们的驱动程序肯定的依靠编译好的内核

all:

make -C $(KERN\_DIR) M=`pwd` modules //make –C //转到KERN\_DIR目录的makefile执行

clean:

make -C $(KERN\_DIR) M=`pwd` modules clean

rm -rf modules.order

obj-m += first\_drv.o

20200719

一般的app的调用过程：

app：用 open（/dev/xxx） write（）操作 //打开的文件有属性 和设备号

属性 主设备号

C 库

VFS 0 1 2 3 4 5 6 7 8 9 ………….111 内核

驱动

Led\_open led\_write

VFS 如何找到驱动？

VFS根据内核中的一个数组,找到对应的一项假如是111.里面的file\_operation.

file\_operation这个在我们的写驱动的时候包括：.open .write owner

入口函数用register\_chardev(主设备号（major），名字，file\_operation.结构体)放在内核数组中

一般主设备号如何确定呢？

一般可以cat /prco/devices可以查看系统有哪些是没用的，可以选用

或者我们在调用register\_chrdev的时候写0，系统自动分配

1手工建立设备节点的命令：mknod /dev/xxx c 250 0 //设备节点位置 主设备号 从设备号

删除设备节点：rm /dev/xxx

2.我们不可能根据自动生成的设备节点去手工创建设备,所以有自动生成的方法

我们注册的设备在sys下会生成设备信息。mdev根据设备信息自动创建设备节点

自动创建：udev机制。Busybox对应mdev。

我们在我们的驱动程序中使用

创建一个类。目的是在sys下生成设备信息，mdev会根据设备信息自动创建设备节点

class\_create

第一个参数指定所要创建的设备所从属的类，第二个参数是这个设备的父设备，如果没有就指定为NULL，第三个参数是设备号，第四个参数是设备名称，第五个参数是从设备号。

class\_device\_create创建设备类

为什莫会自动创建设备呢？

因为我们在制作根文件系统在/etc/init.d/rcS中支持热插拔

echo /sbin/mdev > /proc/sys/kernel/hotplug //内核有文件加载或者卸载会自动调用sbin/mdev所指向的proc/sys/kernel/hotplug这个程序

将做好的firstdrv.ko文件

insmod firstdrv.ko

这样我们在ls – l/dev/xyz 会发现会自动生成/dev/xyz

cat /proc/devices 就会发现如下：

252 first\_drv

我们进去sys/class/firstfrv/xyz dev里面就是主设备号和副设备号

**实现点灯驱动：**

1. 框架
2. 看硬件原理图
3. 看2440手册
4. 写代码（不能像单片机一样直接操作物理地址要操作虚拟地址）

首先我们板子是网络文件系统启动：

**第二个驱动：按键驱动：查询的方式**

1. 写出框架

入口函数的操作

给sysfs操作更多的信息: udev可以自动创建设备节点

1. 硬件操作

看原理图

芯片手册

编程（需要地址映射）

GPF0 GPF2

GPG3 GPG4

犯的错误：

static struct file\_operations second\_drv\_fops = {

.owner = THIS\_MODULE, /\* 这是一个宏，推向编译模块时自动创建的\_\_this\_module变量 \*/

.open = second\_drv\_open,

.read = second\_drv\_read, // .read = second\_drv\_read 如果将read 错误写成write 这个实验会疯狂打印

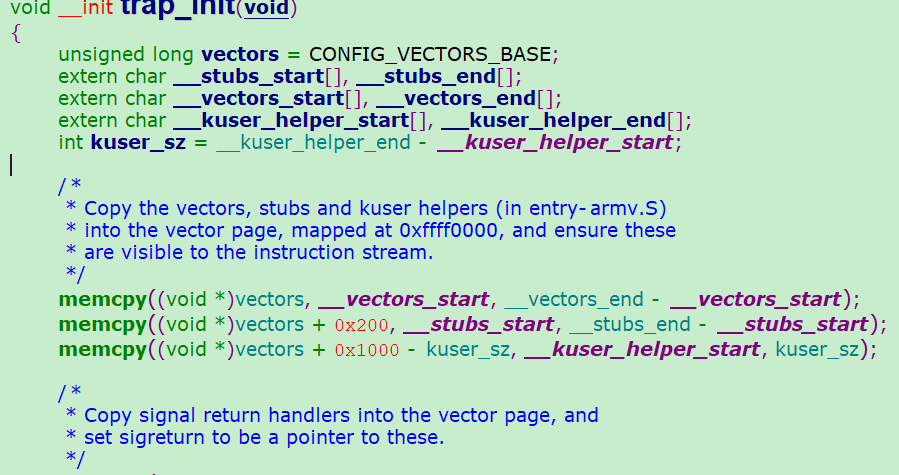
};

**3．第三个程序：中断方式**

1按键按下

1. cpu发生中断
2. 跳转到中断向量入口处执行
3. 函数 。保存中断的现场
4. 执行中断处理程序
5. 恢复现场

linux下 ： trap\_init()（arch/arm/kernel/traps.h） 用来设置各种异常处理的向量，包括中断向量。所谓的向量就是放在固定位置的代码，发生异常时候。cpu自动去执行固定位置的代码。Arm架构的异常向量的基址可以是0x00000000 也可以是0xffff0000.linux内核使用后者。Trap\_init()将异常向量复制到0xffff0000地址（虚拟地址）处。



Vectors就是0xffff0000 ,地址\_\_vectors\_end - \_\_vectors\_start 之间的是异常向量表的位置在在arch/arm/kernel/entry-armv.S定义。

\_\_vectors\_start: 也是一些跳转指令如下：

\_\_vectors\_start:

swi SYS\_ERROR0

b vector\_und + stubs\_offset

ldr pc, .LCvswi + stubs\_offset

b vector\_pabt + stubs\_offset

b vector\_dabt + stubs\_offset

b vector\_addrexcptn + stubs\_offset

b vector\_irq + stubs\_offset

b vector\_fiq + stubs\_offset

.globl \_\_vectors\_end

\_\_vectors\_end:

假如发生了vector\_und 异常。跳到 vector\_und + stubs\_offset地址

其实异常向量表的地只是一些简单的跳转指令，发生异常时cpu自动执行这些指令，跳转到更复杂的代码。比如保存被中断程序的执行环境，调用异常处理函数，恢复被中断程序的环境继续运行。

这些“更复杂的代码”在地址\_\_stubs\_start～\_\_stubs\_end 之间，它们arch/arm/kernel/entry-armv.S 中定义。第722 行将它们复制到地址0xffff0000+0x200 处。

异常向量、异常向量跳去执行的代码都是使用汇编写的，为给读者一个形象概念，下面

讲解部分代码，它们在arch/arm/kernel/entry-armv.S 中。

异常向量的代码如下，其中的“stubs\_offset”用来重新定位跳转的位置（向量被复制到

地址0xffff0000 处，跳转的目的代码被复制到地址0xffff0000+0x200 处）。

1059 .equ stubs\_offset, \_\_vectors\_start + 0x200 - \_\_stubs\_start

1060

1061 .globl \_\_vectors\_start

1062 \_\_vectors\_start:

1063 swi SYS\_ERROR0 /\* 复位时，CPU 将执行这条指令 \*/

1064 b vector\_und + stubs\_offset /\* 未定义异常时，CPU 将执行这条指令 \*/

1065 ldr pc, .LCvswi + stubs\_offset /\* swi 异常 \*/

1066 b vector\_pabt + stubs\_offset /\* 指令预取中止 \*/

1067 b vector\_dabt + stubs\_offset /\* 数据访问中止 \*/

1068 b vector\_addrexcptn + stubs\_offset /\* 没有用到 \*/

1069 b vector\_irq + stubs\_offset /\* irq 异常 \*/

1070 b vector\_fiq + stubs\_offset /\* fiq 异常 \*/

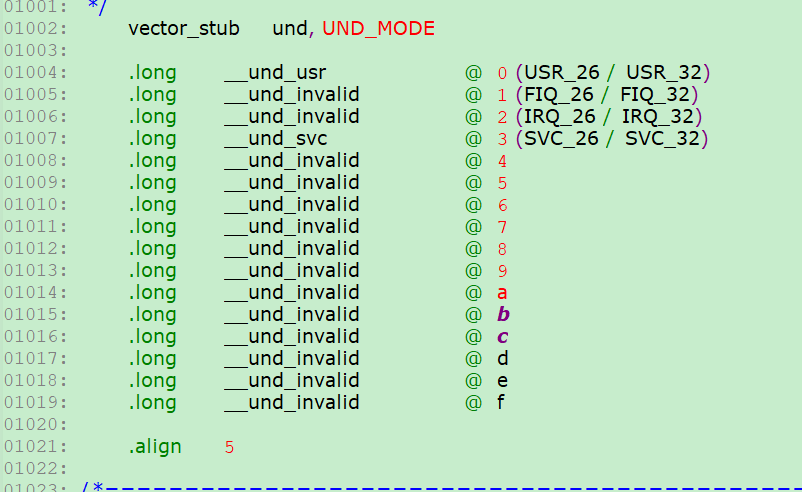
1071

1072 .globl \_\_vectors\_end

1073 \_\_vectors\_end:

其中的 vector\_und、vector\_pabt 等表示要跳转去执行的代码。以vector\_und 为例，它仍

在arch/arm/kernel/entry-armv.S 中，通过vector\_stub 宏来定义，代码如下：



第 1002 行的vector\_stub 是一个宏，它根据后面的参数“und, UND\_MODE”定义了以

“vector\_und”为标号的一段代码。vector\_stub 宏的功能为：计算处理完异常后的返回地址、

保存一些寄存器（比如 r0、lr、spsr），然后进入管理模式，最后根据被中断的工作模式调

用第1004～1019 行中的某个跳转分支。当发生异常时，CPU 会根据异常的类型进入某个

工作模式，但是很快vector\_stub 宏又会强制CPU 进入管理模式，在管理模式下进行后续

处理，这种方法简化了程序设计，使得异常发生前的工作模式要么是用户模式，要么是管

理模式。

第 1004～1019 行中的代码表示在各个工作模式下执行未定义指令时，发生的异常的处

理分支。比如1004 行的\_\_und\_usr 表示在用户模式下执行未定义指令时，所发生的未定义异

常将由它来处理；第1007 行的\_\_und\_svc 表示在管理模式下执行未定义指令时，所发生的未

定义异常将由它来处理。在其他工作模式下不可能发生未定义指令异常，否则使用

“\_\_und\_invalid”来处理错误。ARM 架构CPU 中使用4 位数据来表示工作模式（目前只有7

种工作模式），所以共有16 个跳转分支。

不同的跳转分支（比如\_\_und\_usr、\_\_und\_svc）只是在它们的入口处（比如保存被中断

程序的寄存器）稍有差别，后续的处理大体相同，都是调用相应的C 函数。比如未定义指令

异常发生时，最终会调用C 函数do\_undefinstr 来进行处理。各种的异常的C 处理函数可以分

为5 类，它们分布在不同的文件中

① 在 arch/arm/kernel/traps.c 中。

未定义指令异常的 C 处理函数在这个文件中定义，总入口函数为do\_undefinstr。

② 在 arch/arm/mm/fault.c 中。

与内存访问相关的异常的 C 处理函数在这个文件中定义，比如数据访问中止异常、指令

预取中止异常。总入口函数为do\_DataAbort、do\_PrefetchAbort。

③ 在 arch/arm/mm/irq.c 中。

中断处理函数的在这个文件中定义，总入口函数为asm\_do\_IRQ，它调用其他文件注册

的中断处理函数。

④ 在 arch/arm/kernel/calls.S 中。

在这个文件中，swi 异常的处理函数指针被组织成一个表格；swi 指令机器码的位[23:0]

被用来作为索引。这样，通过不同的“swi index”指令就可以调用不同的swi 异常处理函数，

它们被称为系统调用，比如sys\_open、sys\_read、sys\_write 等。

⑤ 没有使用的异常。

在 Linux 2.6.22.6 中没有使用FIQ 异常。

trap\_init 函数搭建了各类异常的处理框架。当发生异常时，各种C 处理函数会被调用。

这些C 函数还要进一步细分异常发生的情况，分别调用更具体的处理函数。比如未定义指令

异常的C 处理函数总入口为do\_undefinstr，这个函数里还要根据具体的未定义指令调用它的

模拟函数。

除了中断外，内核已经为各类异常准备了细致而完备的处理函数，比如 swi 异常处理函

数为每一种系统调用都准备了一个“sys\_”开头的函数，数据访问中止异常的处理函数为对

齐错误、页权限错误、段翻译错误等具体异常都准备了相应的处理函数。这些异常的处理函

数与开发板的配置无关，基本不用修改

（2）init\_IRQ 函数分析。

中断也是一种异常，之所以把它单独提出来，是因为中断的处理与具体开发板密切相

除一些必须、共用的中断（比如系统时钟中断、片内外设UART 中断）外，必须由驱

动开发者提供处理函数。内核提炼出中断处理的共性，搭建了一个非常容易扩充的中断处

理体系。

init\_IRQ 函数（代码在arch/arm/kernel/irq.c 中）被用来初始化中断的处理框架，设置各

种中断的默认处理函数。当发生中断时，中断总入口函数asm\_do\_IRQ 就可以调用这些函数

作进一步处理。

所以大致如下：

总结：

trap\_init() 构造异常向量 ，也就是是一些跳转

例如 b vectory\_irq //利用宏实现

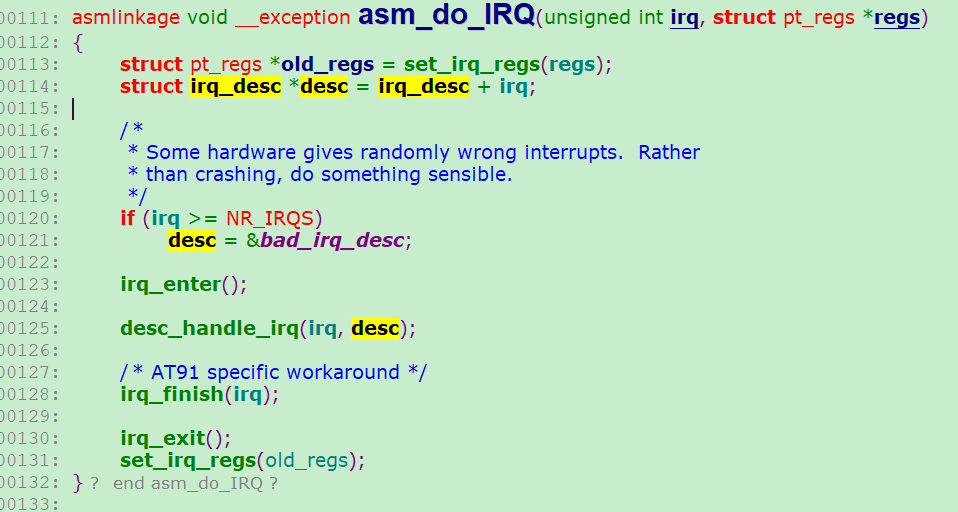
详解可以参考那两份文档entry-armv\_S2 内核entry-armvS 这两个文档

最后到asm\_do\_IRQ（arch/arm/kernel）

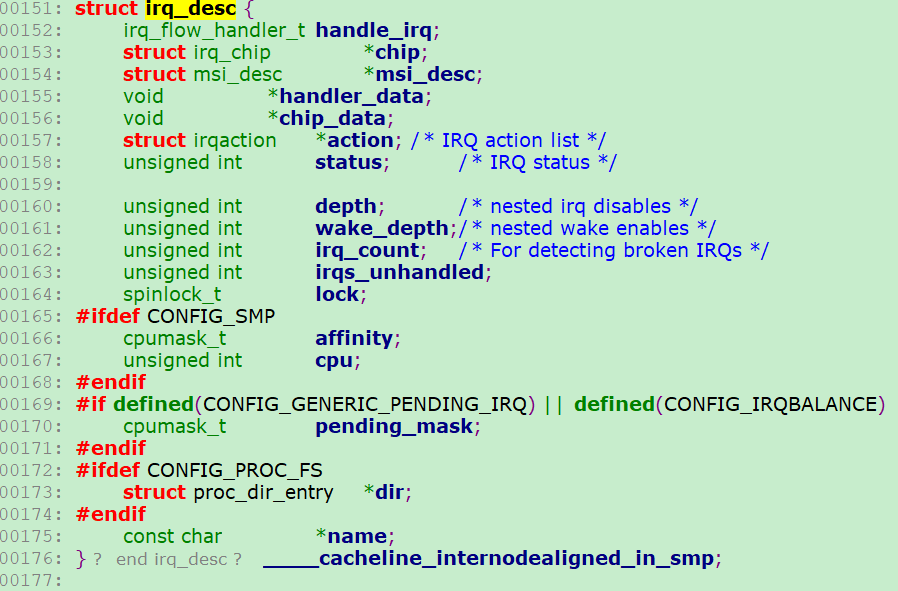
**20200728：linux中断处理函数体系**

asm\_do\_IRQ()//要在带补丁的内核中查找

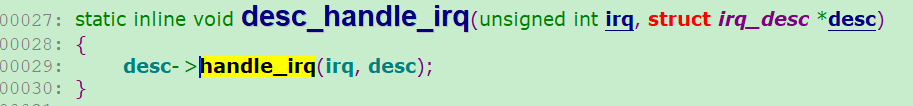
如下图：



Irq\_desc结构体：

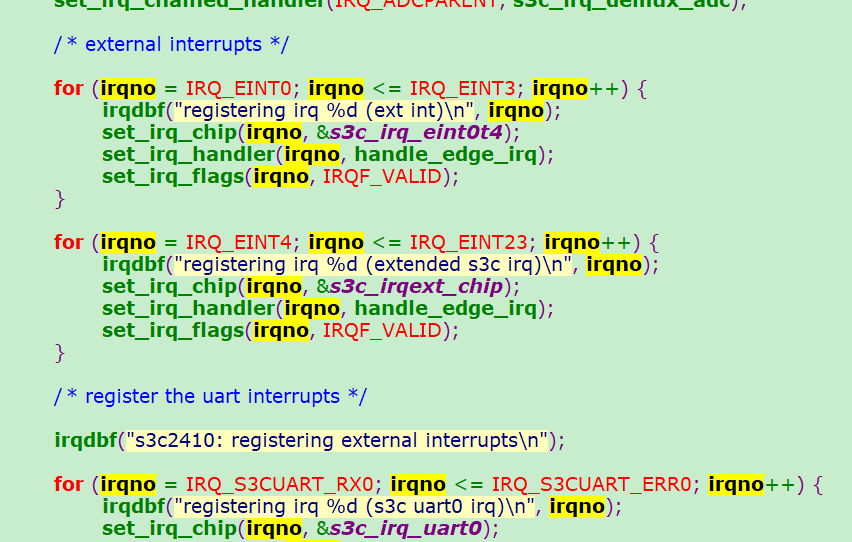


desc\_handle\_irq(irq, desc);

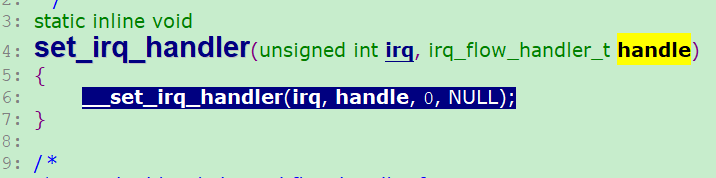


void \_\_init s3c24xx\_init\_irq(void)// 初始化s3c24中断 arch/arm/plat-s3c24xx

set\_irq\_handler(irqno, handle\_edge\_irq);

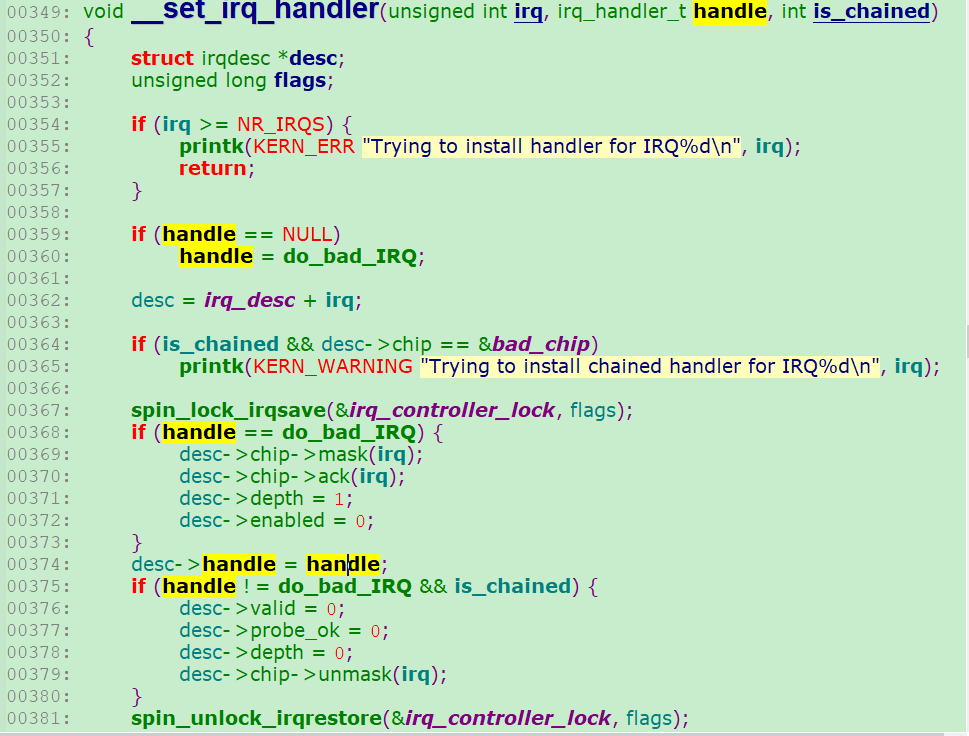


\_\_set\_irq\_handler(irq, handle, 0, NULL);



desc->handle\_irq = handle;//chip.c中

这样就构造了许多的数据在irq\_desc数组中



desc是以中断号为下标的数组中的某一个。Linux 内核将所有的中断统一编号，使用一个irq\_desc 结构数组来描述这些中断。每个

数组项对应一个中断（也有可能是一组中断，它们共用相同的中断号），），里面记录了中断的名称、中断状态、中断标记（比如中断类型、是否共享中断等），并提供了中断的低层硬件访问函数（清除、屏蔽、使能中断），提供了这个中断的处理函数入口，通过它可以调用用户注册的中断处理函数。

通过 irq\_desc 结构数组就可以了解中断处理体系结构，irq\_desc 结构的数据类型在

include/linux/irq.h 中定义，如下所示：

151 struct irq\_desc {

152 irq\_flow\_handler\_t handle\_irq; /\* 当前中断的处理函数入口 \*/

153 struct irq\_chip \*chip; /\* 低层的硬件访问 \*/

⋯⋯

157 struct irqaction \*action; /\* 用户提供的中断处理函数链表 \*/

158 unsigned int status; /\* IRQ 状态 \*/

⋯⋯

175 const char \*name; /\* 中断名称 \*/

176 } \_\_\_\_cacheline\_internodealigned\_in\_smp;

第 152 行的handle\_irq 是这个或这组中断的处理函数入口。发生中断时，总入口函数

asm\_do\_IRQ 将根据中断号调用相应irq\_desc 数组项中的handle\_irq。handle\_irq 使用chip 结

构中的函数来清除、屏蔽或者重新使能中断，还一一调用用户在action 链表中注册的中断处

理函数。

第 153 行的irq\_chip 结构类型也是在include/linux/irq.h 中定义，其中的成员大多用于操作

底层硬件，比如设置寄存器以屏蔽中断、使能中断、清除中断等。这个结构的部分成员如下：

98 struct irq\_chip {

99 const char \*name;

100 unsigned int (\*startup)(unsigned int irq); /\* 启动中断,如果不设置,缺省为

"enable" \*/

101 void (\*shutdown)(unsigned int irq); /\* 关闭中断，如果不设置，缺省为

"disable" \*/

102 void (\*enable)(unsigned int irq); /\* 使能中断，如果不设置，缺省为

"unmask" \*/

103 void (\*disable)(unsigned int irq); /\* 禁止中断，如果不设置，缺省为"mask" \*/

104

105 void (\*ack)(unsigned int irq); /\* 响应中断,通常是清除当前中断使得可以接收

下一个中断\*/

106 void (\*mask)(unsigned int irq); /\* 屏蔽中断源 \*/

107 void (\*mask\_ack)(unsigned int irq); /\* 屏蔽和响应中断 \*/

108 void (\*unmask)(unsigned int irq); /\* 开启中断源 \*/

⋯

126 }

irq\_desc 结构中第157 行的irqaction 结构类型在include/linux/interrupt.h 中定义。用户注

册的每个中断处理函数用一个irqaction 结构来表示，一个中断（比如共享中断）可以有多个处理函数，它们的irqactio 结构链接成一个链表，以action 为表头。irqaction 结构定义如下：

84 struct irqaction {

85 irq\_handler\_t handler; /\* 用户注册的中断处理函数 \*/

86 unsigned long flags; /\* 中断标志，比如是否共享中断、电平触发还是边沿触发等 \*/

87 cpumask\_t mask; /\* 用于SMP(对称多处理器系统) \*/

88 const char \*name; /\* 用户注册的中断名字，"cat /proc/interrupts"时

可以看到 \*/

89 void \*dev\_id; /\* 用户传给上面的handler 的参数，还可以用来区分共

享中断 \*/

90 struct irqaction \*next;

91 int irq; /\* 中断号 \*/

92 struct proc\_dir\_entry \*dir;

93 };

1 按下按键 cpu进入异常。

2 b vector\_irq + stubs\_offset //trap\_init 中的将异常向量表的地址放到一个linux固定基址上然后根据不同向量去偏移。执行跳转指令

3 vector\_irq 宏定义进行下一步

4 \_\_irq\_usr

5 usr\_entry //保存现场堆栈的一些操作

1. irq\_handler//也是一个宏

7 asm\_do\_irq 最终到这里

8 以中断号为下标的数组 desc\_handle\_irq(irq,desc)

9 desc->handle\_irq(irq, desc); //假设 通过set\_irq\_handler(irqno, handle\_edge\_irq); desc->handle\_irq 指向 handle\_edge\_irq

10 set\_irq\_handler(irqno, handle\_edge\_irq);//以这个为例子会实现如下两步

/\* Start handling the irq \*/

desc->chip->ack(irq);//清除中断

action\_ret = handle\_IRQ\_event(irq, action);//取出action链表中的成员执行action-handler

request\_irq函数：对应 free\_irq(unsigned int irq ,void \*dev\_id)

int request\_irq(unsigned int irq,irq\_handler\_t handler,unsigned long irqflags, const char \* devname, void \*dev\_id)

1 分配一个irqaction空间

2 setup\_irq（irq action）

3 struct irq\_desc \*desc = irq\_desc + irq;//在irq\_desc找 加入到链表中

4 if (old) {//判断链接头是否为空

/\*

\* Can't share interrupts unless both agree to and are

\* the same type (level, edge, polarity). So both flag

\* fields must have IRQF\_SHARED set and the bits which

\* set the trigger type must match.

\*/

if (!((old->flags & new->flags) & IRQF\_SHARED) ||/\*链接头挂接有很多项表示是共享中断

\*/

((old->flags ^ new->flags) & IRQF\_TRIGGER\_MASK)) {

old\_name = old->name;

goto mismatch;

}

//判断是否是共享中断

if (!shared) {//之前已有

irq\_chip\_set\_defaults(desc->chip);

desc->chip->set\_type(irq,new->flags & IRQF\_TRIGGER\_MASK);

desc->chip->startup(irq); //使能中断

request\_irq函数：总结：创建一个irqaction 放入irq\_desc action链表里

设置引脚

使能中断

free\_irq(unsigned int irq ,void \*dev\_id)

出链表

禁止中断

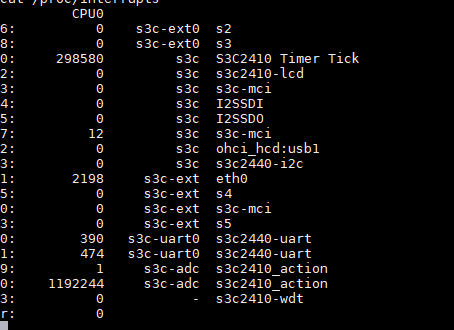
测试程序：third\_button\_driver

cat /proc/interrupts //没有发现

在我们的测试程序中如何打开中断呢？

exec 5</dev/buttons //打开设备/dev/buttons定位到5中

这样我们就可以用中断的方式测试按键了



ls -l /proc/770/fd

就可以看到如下的信息：

lr-x------ 1 0 0 64 Jan 1 00:27 5 -> /dev/button

关闭中断

exec 5<&-

正式的测试代码：目录third\_button\_driver\_1

执行应用程序

挂载驱动

如果rmmod出现rmmod: third\_drv: Resource temporarily unavailable

ps 查看进程号之后 kill -9 id

要想我们运行的程序在后台运行我们要在执行程序./xxx &

**POLL机制：**

poll 中调用 poll\_wait（）

p->qproc(filp, wait\_address, p);// 等效\_\_pollwait（filp, wait\_address, p）

//\_\_pollwait的作用将当前进程放到wait\_address队列中去

Poll //应用程序

Sys\_poll//系统调用

do\_sys\_poll

poll\_initwait(&table);

init\_poll\_funcptr(&pwq->pt, \_\_pollwait);// pt->qproc = qproc;其实就是\_\_pollwait就指向了pwq->pt这个成员

do\_poll(nfds, head, &table, timeout);

for (;;)

{

for (; pfd != pfd\_end; pfd++)//查询多个文件

{

if (do\_pollfd(pfd, pt)) {

count++;//如果pull的驱动程序返回非0那莫count++

pt = NULL;

}

}

do\_pollfd: mask = file->f\_op->poll(file, pwait)//返回mask，poll就是调用我们的驱动函数，在里面调用

//braek 条件 count非零 超时 有信号在等待处理

if (count || !\*timeout || signal\_pending(current))

break;

//如果上面不成立休眠\_\_timeout

\_\_timeout = schedule\_timeout(\_\_timeout);//休眠结束后继续for 此时timeout为0

}

至此我们的按键有如下的机制：

1. 查询 //耗资源

2．中断方式 //read()会阻塞等待

3. poll 机制 //指定超时时间

驱动提醒应用程序：

**异步机制：**

信号的要点:

1. 注册信号处理函数
2. 发信号
3. 谁发发给谁
4. 怎末发

按下按钮驱动通知应用程序：

应用程序：注册信号处理函数

谁发：驱动

发给谁：app.app要告诉驱动pid

怎莫发：kill\_fasync

为了使设备支持异步通知机制，驱动程序中涉及以下3项工作：

1. 支持F\_SETOWN命令，能在这个控制命令处理中设置filp->f\_owner为对应进程ID。

不过此项工作已由内核完成，设备驱动无须处理。

2. 支持F\_SETFL命令的处理，每当FASYNC标志改变时，驱动程序中的fasync()函数将得以执行。

驱动中应该实现fasync()函数。

3. 在设备资源可获得时，调用kill\_fasync()函数激发相应的信号

应用程序：

应用程序：

fcntl(fd, F\_SETOWN, getpid()); // 告诉内核，pid发给谁

//改变Oflags标记 调用fith\_drv\_fasync 初始化结构体struct fasync\_struct

Oflags = fcntl(fd, F\_GETFL);

fcntl(fd, F\_SETFL, Oflags | FASYNC); // 改变fasync标记，最终会调用到驱动的faync > fasync\_helper：初始化/释放fasync\_struct

同一时刻只能有一个app 打开驱动 ：

首先我们了解下同步 互斥 阻塞

1. 原子操作

指在执行过程中不会被别的代码路径打断操作

常用的原子操作函数：

atomic\_t v = ATOMIC\_INIT(0); //定义原子变量v并初始化为0

atomic\_read(atomic\_t \*v); //返回原子变量的值

void atomic\_inc(atomic\_t \*v); //原子变量增加1

void atomic\_dec(atomic\_t \*v); //原子变量减少1

int atomic\_dec\_and\_test(atomic\_t \*v); //自减操作后测试其是否为0，为0则返回true，否则返回false。

2. 信号量 //当我们打开ps 命令时 假如一个进程的状态是D 表明他的状态时将死状态 不可被打断

信号量（semaphore）是用于保护临界区的一种常用方法，只有得到信号量的进程才能执行临界区代码。

当获取不到信号量时，进程进入休眠等待状态。

定义信号量

struct semaphore sem;

初始化信号量

void sema\_init (struct semaphore \*sem, int val);

void init\_MUTEX(struct semaphore \*sem);//初始化为0

static DECLARE\_MUTEX(button\_lock); //定义互斥锁

获得信号量

void down(struct semaphore \* sem);

int down\_interruptible(struct semaphore \* sem); //获取不到休眠

int down\_trylock(struct semaphore \* sem);//获取不到信号量就返回

释放信号量

void up(struct semaphore \* sem);

3. 阻塞

阻塞操作

是指在执行设备操作时若不能获得资源则挂起进程，直到满足可操作的条件后再进行操作。

被挂起的进程进入休眠状态，被从调度器的运行队列移走，直到等待的条件被满足。

非阻塞操作

进程在不能进行设备操作时并不挂起，它或者放弃，或者不停地查询，直至可以进行操作为止。

fd = open("...", O\_RDWR | O\_NONBLOCK);// O\_NONBLOCK 非阻塞默认是阻塞