个人资料

liuhaoyutz

访问： 80630次
积分： 1673分
排名： 第7877名

原创： 83篇 转载： 0篇
译文： 0篇 评论： 59条

LDD3源码分析之与硬件通信&中断处理

分类： LDD3源码分析 2012-04-11 08:45 2333人阅读 评论(2) 收藏 举报

struct buffer parallel null file 工作

作者：刘昊昱

博客：http://blog.csdn.net/liuhaoyutz

编译环境：Ubuntu 10.10

内核版本：2.6.32-38-generic-pae

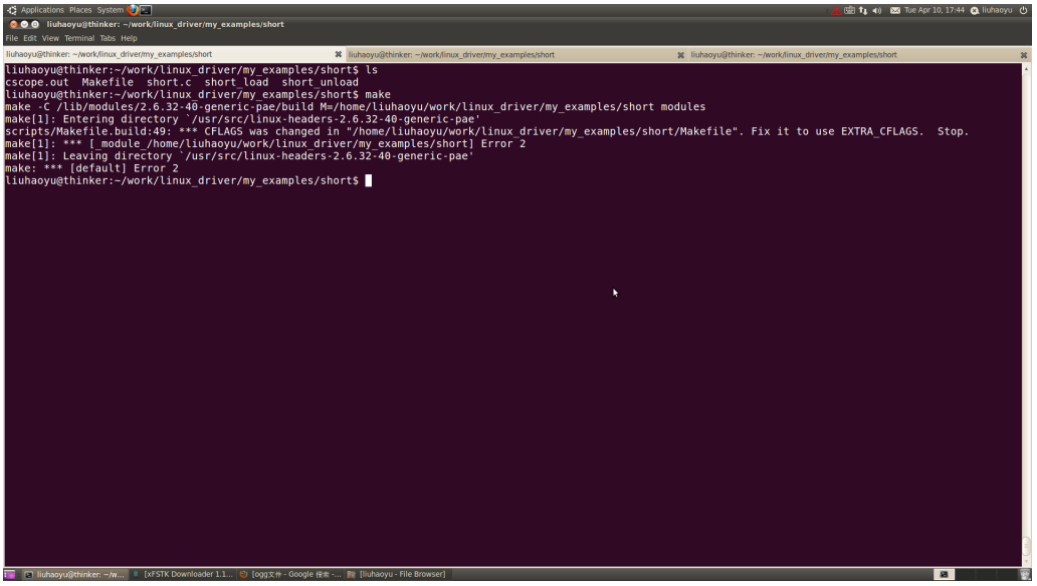
LDD3源码路径：examples/short/

本分析LDD3第9和第10章的示例代码short。short涉及的主要知识点有通过I/O端口或I/O内存操作设备寄存器及设备内存，注册中断处理函数处理中断。本来第9和第10章的代码应该分别进行讨论，但是因为short的代码相互关联比较紧密，所以这里放在同一篇文章中分析。

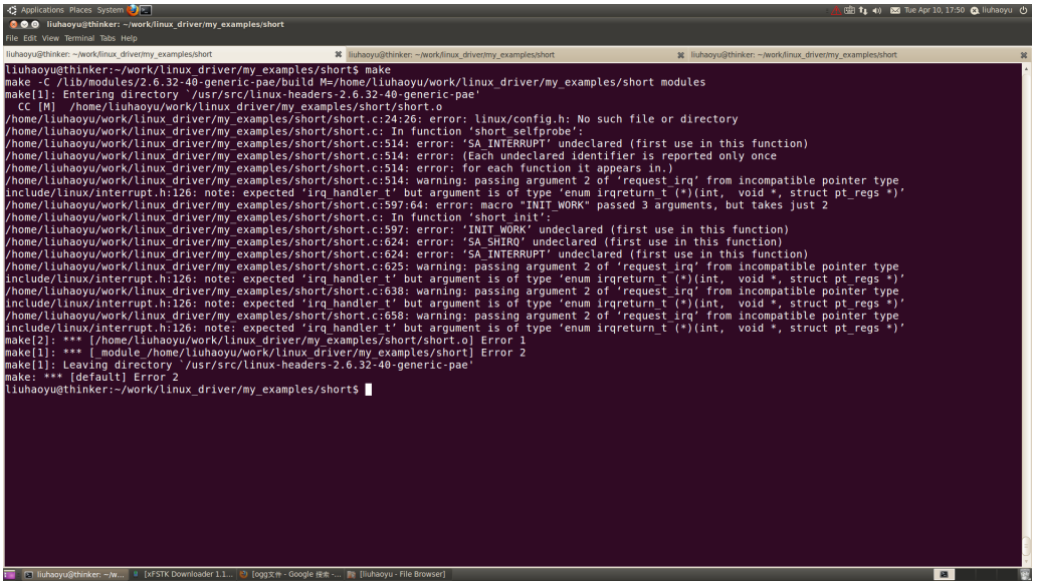
一、short模块编译

在新的内核下，编译short模块时，会遇到一些问题，这里列出遇到的问题及解决方法。

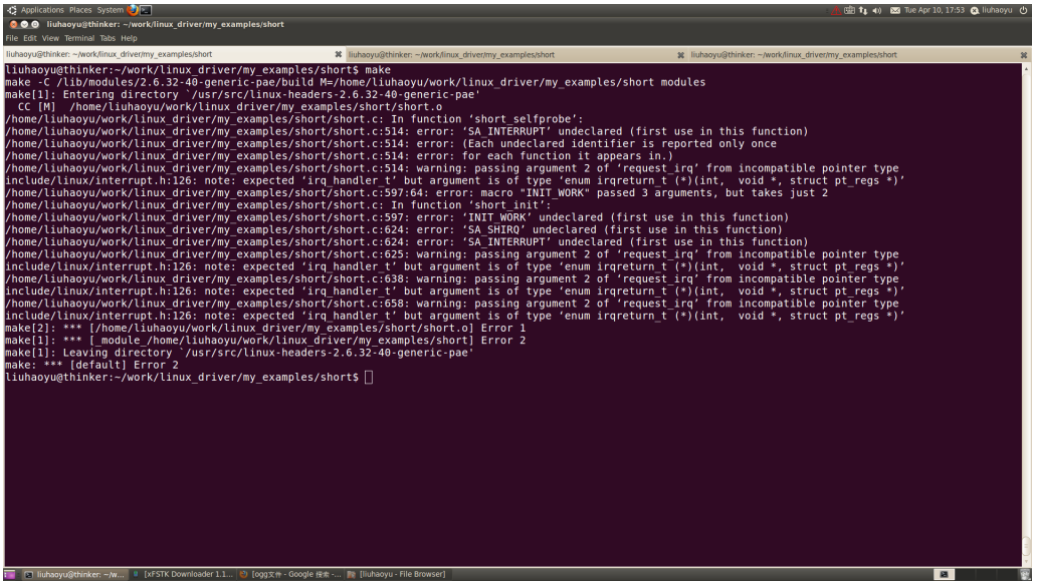
第一次make时，出现如下错误：



修改Makefile的第12，13，35行，将CFLAGS改为EXTRA_CFLAGS，即可解决这个问题。再次make,会出现如下错误：

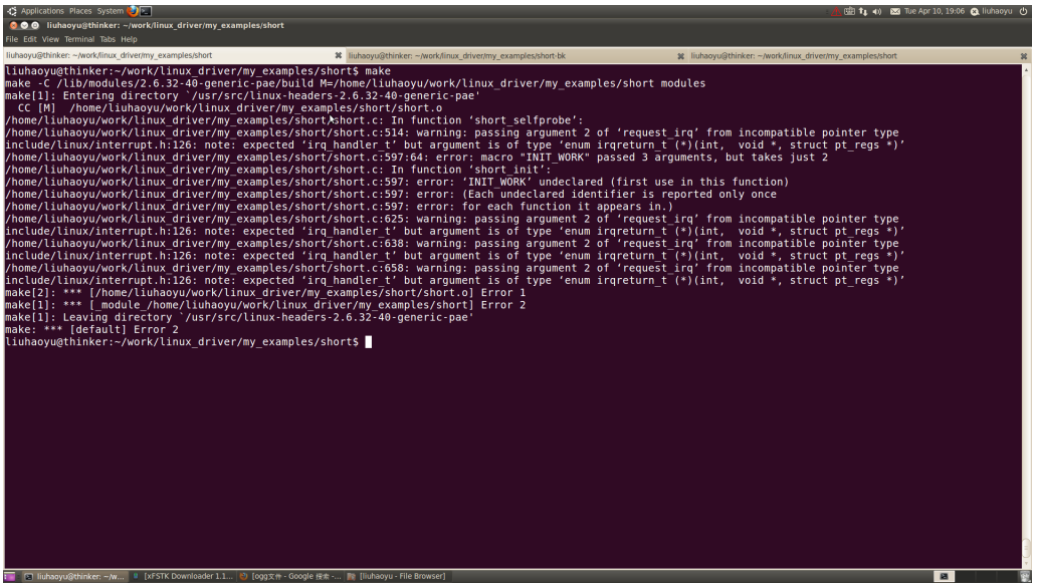


修改short.c，把第24行#include <linux/config.h>屏蔽掉。再次编译出现如下问题：



这是因为SA_INTERRUPT和SA_SHIRQ标志在新内核中发生了变化，SA_INTERRUPT标志已经不存在了，SA_SHIRQ标志位变为IRQF_SHARED。所以做以下修改：

514，638，658行把flag标志设置为0，624行把flag设置为IRQF_SHARED，修改完成后，再次编译，出现如下错误：



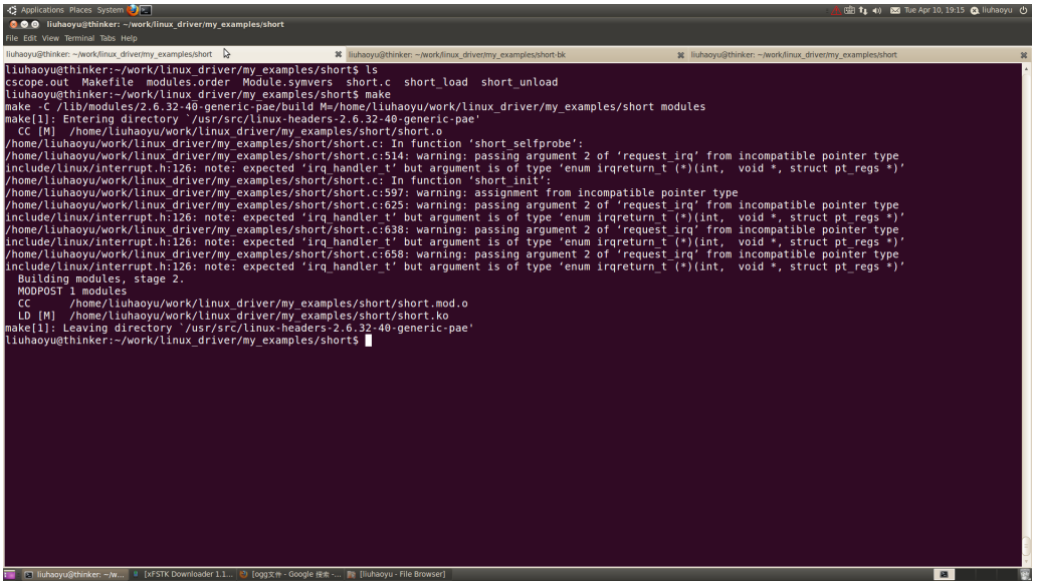
修改597行为INIT_WORK(&short_wq, (void (*)(struct work_struct *)) short_do_tasklet);

2014年01月 (1)
2013年12月 (6)
展开

文章搜索

推荐文章

再次make，编译通过，但还有一些警告信息如下：



这是因为在新的内核版本中中断处理函数的原型只有两个参数，而在2.6.10中有三个参数，这里只要把相应中断处理函数的第三个参数去掉即可，修改后的函数原型如下：

494irqreturn_t short_probing(int irq, void *dev_id)

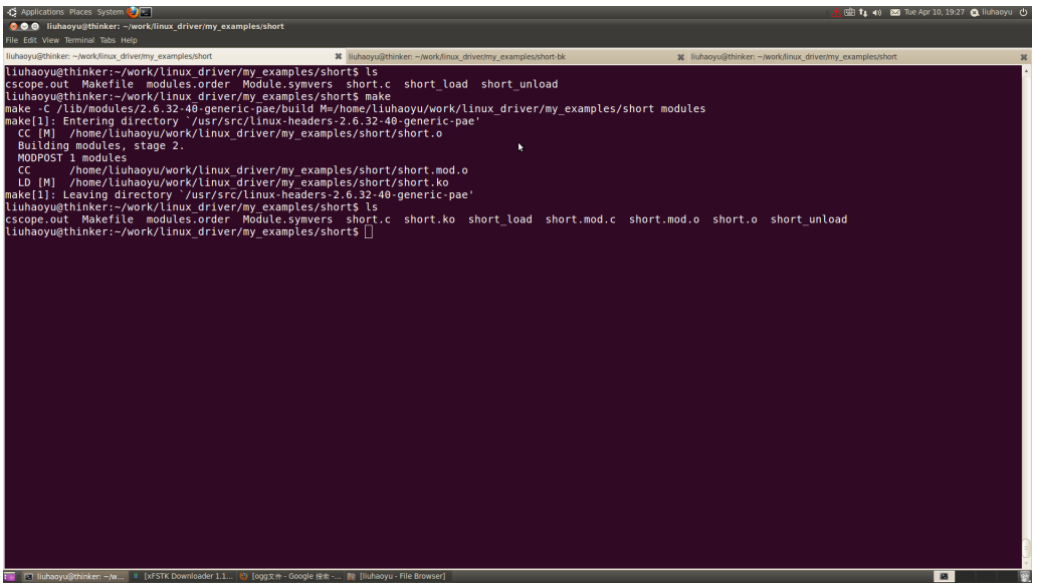
443irqreturn_t short_sh_interrupt(int irq, void *dev_id)

431irqreturn_t short_tl_interrupt(int irq, void *dev_id)

413irqreturn_t short_wq_interrupt(int irq, void *dev_id)

336irqreturn_t short_interrupt(int irq, void *dev_id)

再次编译，通过。



二、short模块初始化

先来看short模块初始化函数：

```cpp
548int short_init(void)
549{
550    int result;
551
552    /*
553     * first, sort out the base/short_base ambiguity: we'd better
```

```
554     * use short_base in the code, for clarity, but allow setting
555     * just "base" at load time. Same for "irq".
556     */
557    short_base = base;
558    short_irq = irq;
559
560    /* Get our needed resources. */
561    if (!use_mem) {
562        if (! request_region(short_base, SHORT_NR_PORTS, "short")) {
563            printk(KERN_INFO "short: can't get I/O port address 0x%lx\n",
564                    short_base);
565            return -ENODEV;
566        }
567
568    } else {
569        if (! request_mem_region(short_base, SHORT_NR_PORTS, "short")) {
570            printk(KERN_INFO "short: can't get I/O mem address 0x%lx\n",
571                    short_base);
572            return -ENODEV;
573        }
574
575        /* also, ioremap it */
576        short_base = (unsigned long) ioremap(short_base, SHORT_NR_PORTS);
577        /* Hmm... we should check the return value */
578    }
579    /* Here we register our device - should not fail thereafter */
580    result = register_chrdev(major, "short", &short_fops);
581    if (result < 0) {
582        printk(KERN_INFO "short: can't get major number\n");
583        release_region(short_base,SHORT_NR_PORTS);  /* FIXME - use-mem case? */
584        return result;
585    }
586    if (major == 0) major = result; /* dynamic */
587
588    short_buffer = __get_free_pages(GFP_KERNEL,0); /* never fails */  /* FIXME */
589    short_head = short_tail = short_buffer;
590
591    /*
592     * Fill the workqueue structure, used for the bottom half handler.
593     * The cast is there to prevent warnings about the type of the
594     * (unused) argument.
595     */
596    /* this line is in short_init() */
597    INIT_WORK(&short_wq, (void (*)(void *)) short_do_tasklet, NULL);
598
599    /*
600     * Now we deal with the interrupt: either kernel-based
601     * autodetection, DIY detection or default number
602     */
603
604    if (short_irq < 0 && probe == 1)
605        short_kernelprobe();
606
607    if (short_irq < 0 && probe == 2)
608        short_selfprobe();
609
610    if (short_irq < 0) /* not yet specified: force the default on */
611        switch(short_base) {
612            case 0x378: short_irq = 7; break;
613            case 0x278: short_irq = 2; break;
614            case 0x3bc: short_irq = 5; break;
615        }
616
617    /*
618     * If shared has been specified, installed the shared handler
619     * instead of the normal one. Do it first, before a -EBUSY will
620     * force short_irq to -1.
621     */
622    if (short_irq >= 0 && share > 0) {
623        result = request_irq(short_irq, short_sh_interrupt,
624                SA_SHIRQ | SA_INTERRUPT,"short",
625                short_sh_interrupt);
626        if (result) {
627            printk(KERN_INFO "short: can't get assigned irq %i\n", short_irq);
628            short_irq = -1;
629        }
630        else { /* actually enable it -- assume this *is* a parallel port */
631            outb(0x10, short_base+2);
632        }
```

```
633         return 0; /* the rest of the function only installs handlers */
634     }
635
636     if (short_irq >= 0) {
637         result = request_irq(short_irq, short_interrupt,
638                 SA_INTERRUPT, "short", NULL);
639         if (result) {
640             printk(KERN_INFO "short: can't get assigned irq %i\n",
641                     short_irq);
642             short_irq = -1;
643         }
644         else { /* actually enable it -- assume this *is* a parallel port */
645             outb(0x10,short_base+2);
646         }
647     }
648
649     /*
650      * Ok, now change the interrupt handler if using top/bottom halves
651      * has been requested
652      */
653     if (short_irq >= 0 && (wq + tasklet) > 0) {
654         free_irq(short_irq,NULL);
655         result = request_irq(short_irq,
656                 tasklet ? short_tl_interrupt :
657                 short_wq_interrupt,
658                 SA_INTERRUPT,"short-bh", NULL);
659         if (result) {
660             printk(KERN_INFO "short-bh: can't get assigned irq %i\n",
661                     short_irq);
662             short_irq = -1;
663         }
664     }
665
666     return 0;
667}
```

561 - 567行，如果指定使用I/O端口，则调用request_region函数分配I/O端口，这里代码指定要分配从short_base开始的SHORT_NR_PORTS个即8个端口。

568 - 578行，如果指定使用I/O内存，则调用request_mem_region函数分配从short_base开始的SHORT_NR_PORTS个即8个字节的I/O内存。分配I/O内存并不是在使用这些内存之前需要完成的唯一步骤，我们必须首先通过ioremap函数建立映射。ioremap返回用来访问指定物理内存的虚拟地址。

580 - 586行，注册字符设备short″，其文件操作函数集是short_fops。

588行，调用__get_free_pages(GFP_KERNEL,0)分配一个页面保存在 short_buffer中。

597行，调用INIT_WORK初始化一个工作，将来用作中断处理函数的下半部。

604 - 605行，如果short_irq<0并且probe等于1，则调用short_kernelprobe函数由内核探测中断号。该函数的实现我们后面分析。

607 - 608行，如果short_irq<0并且probe等于2，则调用short_selfprobe函数自己手动探测中断号，该函数的实现我们后面分析。

610 - 615行，如果探测没有成功，根据端口地址，强制指定中断号。

622 - 634行，以共享中断的方式注册中断处理函数。需要注意的是631行调用outb(0x10, short_base+2),将并口2号寄存器的第4位置为1，表示启动并口中断报告。

636 - 647行，以非共享中断的方式注册中断处理函数。

653 - 664行，以上半部/下半部的方式注册中断处理函数。

下面我们来看short_kernelprobe函数如何实现由内核自动探测中断号的：

```
[cpp]
466void short_kernelprobe(void)
467{
468     int count = 0;
469     do {
```

```
470        unsigned long mask;
471
472        mask = probe_irq_on();
473        outb_p(0x10,short_base+2); /* enable reporting */
474        outb_p(0x00,short_base);   /* clear the bit */
475        outb_p(0xFF,short_base);   /* set the bit: interrupt! */
476        outb_p(0x00,short_base+2); /* disable reporting */
477        udelay(5);  /* give it some time */
478        short_irq = probe_irq_off(mask);
479
480        if (short_irq == 0) { /* none of them? */
481            printk(KERN_INFO "short: no irq reported by probe\n");
482            short_irq = -1;
483        }
484        /*
485         * if more than one line has been activated, the result is
486         * negative. We should service the interrupt (no need for lpt port)
487         * and loop over again. Loop at most five times, then give up
488         */
489    } while (short_irq < 0 && count++ < 5);
490    if (short_irq < 0)
491        printk("short: probe failed %i times, giving up\n", count);
492}
```

Linux内核提供了探测可用中断号的接口，但这种接口只能在非共享中断模式下使用。内核提供的接口由两个函数组成：

unsigned long probe_irq_on(void);

这个函数返回一个未分配中断的位掩码，驱动程序必须保存返回的位掩码，并将它传递给probe_irq_off函数。调用probe_irq_on函数之后，驱动程序要安排设备产生至少一次中断。

int probe_irq_off(unsigned long);

在请求设备产生中断之后，驱动程序要调用这个函数，并将前面probe_irq_on返回的位掩码作为参数传递给它。probe_irq_off返回probe_irq_on之后发生的中断编号。如果没有中断发生，就返回0。如果产生了多次中断，出现了二义性，就返回负数。

使用内核提供的接口探测中断号时，需要注意在调用probe_irq_on之后启用设备中断，在调用probe_irq_off之前禁用中断。另外，在probe_irq_off之后，需要处理设备上待处理的中断。

472行，调用probe_irq_on函数。

473行，将2号端口的第4位(0x10)设置为1，启用中断。

474行，将0号端口清0。

475行，将0号端口置1，触发中断。

476行，将2号端口的第4位(0x10)设置为0，禁用中断。

477行，延时一会，以保证中断的传递时间。

478行，调用probe_irq_off函数,并把472行probe_irq_on函数返回的位掩码传递给它。

480行，probe_irq_off函数返回0，说明没有中断发生。

489行，probe_irq_off函数返回负值，说明发生了不止一个中断，需要重新探测，这里限定最多探测5次。

下面我们看short_selfprobe函数如何实现DIY探测中断号：

```
[cpp]
501void short_selfprobe(void)
502{
503    int trials[] = {3, 5, 7, 9, 0};
504    int tried[]  = {0, 0, 0, 0, 0};
505    int i, count = 0;
506
507    /*
```

```
508     * install the probing handler for all possible lines. Remember
509     * the result (0 for success, or -EBUSY) in order to only free
510     * what has been acquired
511      */
512    for (i = 0; trials[i]; i++)
513        tried[i] = request_irq(trials[i], short_probing,
514                SA_INTERRUPT, "short probe", NULL);
515
516    do {
517        short_irq = 0; /* none got, yet */
518        outb_p(0x10,short_base+2); /* enable */
519        outb_p(0x00,short_base);
520        outb_p(0xFF,short_base); /* toggle the bit */
521        outb_p(0x00,short_base+2); /* disable */
522        udelay(5);  /* give it some time */
523
524        /* the value has been set by the handler */
525        if (short_irq == 0) { /* none of them? */
526            printk(KERN_INFO "short: no irq reported by probe\n");
527        }
528        /*
529         * If more than one line has been activated, the result is
530         * negative. We should service the interrupt (but the lpt port
531         * doesn't need it) and loop over again. Do it at most 5 times
532         */
533    } while (short_irq <=0 && count++ < 5);
534
535    /* end of loop, uninstall the handler */
536    for (i = 0; trials[i]; i++)
537        if (tried[i] == 0)
538            free_irq(trials[i], NULL);
539
540    if (short_irq < 0)
541        printk("short: probe failed %i times, giving up\n", count);
542}
494irqreturn_t short_probing(int irq, void *dev_id, struct pt_regs *regs)
495{
496    if (short_irq == 0) short_irq = irq;    /* found */
497    if (short_irq != irq) short_irq = -irq; /* ambiguous */
498    return IRQ_HANDLED;
499}
```

DIY探测与内核自动探测的原理是一样的:先启动所有未被占用的中断,然后观察会发生什么。但是，我们要充分发挥对具体设备的了解。通常，设备能使用3或4个IRQ号中的一个来进行配置，探测这些IRQ号，使我们能不必测试所有可能的IRQ就能检测到正确的IRQ号。

并口允许用户选择的IRQ号有3，5，7，9，所以在short中，我们探测这几个中断号。

503行，trials数组列出了以0作为结束标志的需要测试的IRQ。

504行，tried数组用来记录哪个中断号被short驱动程序注册了。

512 - 514行，循环trials数组，为每个要探测的中断号注册中断处理函数short_probing。注意，request_irq函数如果注册成功，返回0保存在tried[i]中。

517 - 522行，触发中断，引起short_probing函数的执行。在short_probing函数中，将发生中断的中断号保存在short_irq中，如果发生多次中断，将设置short_irq值为负数。

525 - 527行，如果short_irq的值为0，说明没有发生中断。

533行，如果short_irq的值小于或等于0，则重新探测，最多探测5次。

536 - 538行，释放IRQ。

完成自动探测或DIY探测后，我们回到short_init函数：

610 - 615行，short_irq小于0，说明没有探测到中断号，short根据端口地址，强制指定默认中断号。

622 - 634行，如果(short_irq >= 0 && share > 0)，则以共享中断方式注册中断处理函数short_sh_interrupt。其中，631行使用outb(0x10, short_base + 2)启动中断报告。

636 – 647行，如果没有指定共享中断，则以非共享中断方式注册中断处理函数short_interrupt。其中645行outb(0x10,short_base+2)启动中断报告。

653 – 663行，注册以顶半部/底半部的方式执行中断处理。如果使用tasklet，对应的中断处理函数是short_tl_interrupt，如果使用工作队列，对应的中断处理函数是short_wq_interrupt。

按照在short_init中出现的顺序，下面我们要看short_sh_interrupt函数了：

```cpp
443irqreturn_t short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
444{
445    int value, written;
446    struct timeval tv;
447
448    /* If it wasn't short, return immediately */
449    value = inb(short_base);
450    if (!(value & 0x80))
451        return IRQ_NONE;
452
453    /* clear the interrupting bit */
454    outb(value & 0x7F, short_base);
455
456    /* the rest is unchanged */
457
458    do_gettimeofday(&tv);
459    written = sprintf((char *)short_head,"%08u.%06u\n",
460            (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
461    short_incr_bp(&short_head, written);
462    wake_up_interruptible(&short_queue); /* awake any reading process */
463    return IRQ_HANDLED;
464}
 93/*
 94 * Atomicly increment an index into short_buffer
 95 */
 96static inline void short_incr_bp(volatile unsigned long *index, int delta)
 97{
 98    unsigned long new = *index + delta;
 99    barrier();  /* Don't optimize these two together */
100    *index = (new >= (short_buffer + PAGE_SIZE)) ? short_buffer : new;
101}
```

注册共享的中断处理程序时，request_irq函数的flag参数必须指定SA_SHIRQ标志，同时dev_id参数必须是唯一的，任何指向模块地址空间的指针都可以使用，但是dev_id不能设置为NULL。

注销共享中断处理程序同样使用free_irq，传递dev_id参数用来从该中断的共享处理程序列表中选择指定的处理程序。这也是dev_id必须唯一的原因。

内核为每个中断维护了一个共享处理程序列表，这些处理程序的dev_id各不相同，就像是设备的签名。

当请求一个共享中断时，如果满足下面条件之一，request_irq就能成功：

1.中断号空闲。

2.任何已经注册了该中断号的处理例程也标识了中断号是共享的。

当共享的中断发生时，内核会调用每一个已经注册的中断处理函数，因此，一个共享中断的中断处理函数必须能识别属于自己的中断，如果不是自己的设备被中断，应该迅速退出。

449 – 451行，读取端口short_base，如果ACK位为1，则报告的中断就是发送给short的。如果为0，则是发给其它中断处理函数的，此时short_sh_interrupt应该立即退出。

454行，清除ACK位。

458行，获取当前时间。

459 – 460行，将时间信息保存在short_head中，在模块初始化函数short_init中，有如下语句：

```
588     short_buffer = __get_free_pages(GFP_KERNEL,0); /* never fails */  /* FIXME */
```

```
589     short_head = short_tail = short_buffer;
```

所以short_head指向缓冲区short_buffer的空闲起始位置。

461行，调用short_incr_bp函数更新空闲缓冲区头指针short_head位置。

462行，唤醒等待队列short_queue上的进程。

如果不是使用共享中断方式，在short_init函数中注册的中断处理函数是short_interrupt，该函数内容如下：

```cpp
336irqreturn_t short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
337{
338    struct timeval tv;
339    int written;
340
341    do_gettimeofday(&tv);
342
343        /* Write a 16 byte record. Assume PAGE_SIZE is a multiple of 16 */
344    written = sprintf((char *)short_head,"%08u.%06u\n",
345            (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
346    BUG_ON(written != 16);
347    short_incr_bp(&short_head, written);
348    wake_up_interruptible(&short_queue); /* awake any reading process */
349    return IRQ_HANDLED;
350}
```

short_interrupt函数的内容和共享中断处理函数short_sh_interrupt的后半部分完全一样，这里不多解释，请参考对short_sh_interrupt函数的分析。

如果指定以顶半部/底半部的方式执行中断处理，在short_init函数中重新注册了中断处理函数，如果采用tasklet，则顶半部是short_tl_interrupt，如果采用工作队列，则顶半部是short_wq_interrupt。这两个函数列出如下：

```cpp
413irqreturn_t short_wq_interrupt(int irq, void *dev_id, struct pt_regs *regs)
414{
415    /* Grab the current time information. */
416    do_gettimeofday((struct timeval *) tv_head);
417    short_incr_tv(&tv_head);
418
419    /* Queue the bh. Don't worry about multiple enqueueing */
420    schedule_work(&short_wq);
421
422    short_wq_count++; /* record that an interrupt arrived */
423    return IRQ_HANDLED;
424}
425
426
427/*
428 * Tasklet top half
429 */
430
431irqreturn_t short_tl_interrupt(int irq, void *dev_id, struct pt_regs *regs)
432{
433    do_gettimeofday((struct timeval *) tv_head); /* cast to stop 'volatile' warning */
434    short_incr_tv(&tv_head);
435    tasklet_schedule(&short_tasklet);
436    short_wq_count++; /* record that an interrupt arrived */
437    return IRQ_HANDLED;
438}
```

在顶半部中，取得当前时间后，调用short_incr_tv函数将时间保存在tv_data数组中，然后调度tasklet或工作稍后执行：

```cpp
372static inline void short_incr_tv(volatile struct timeval **tvp)
373{
```

```
374    if (*tvp == (tv_data + NR_TIMEVAL - 1))
375        *tvp = tv_data;  /* Wrap */
376    else
377        (*tvp)++;
378}
```

short_incr_tv函数用到的几个变量定义如下：

```
[cpp]
357#define NR_TIMEVAL 512 /* length of the array of time values */
358
359struct timeval tv_data[NR_TIMEVAL]; /* too lazy to allocate it */
360volatile struct timeval *tv_head=tv_data;
361volatile struct timeval *tv_tail=tv_data;
```

工作short_wq的初始化在short_init函数中：

```
[cpp]
597    INIT_WORK(&short_wq, (void (*)(void *)) short_do_tasklet, NULL);
```

tasklet short_tasklet定义在第91行，如下：

```
[cpp]
91DECLARE_TASKLET(short_tasklet, short_do_tasklet, 0);
```

由此可见，工作队列和tasklet的处理函数都是short_do_tasklet，它就是所谓的底半部函数：

```
[cpp]
382void short_do_tasklet (unsigned long unused)
383{
384    int savecount = short_wq_count, written;
385    short_wq_count = 0; /* we have already been removed from the queue */
386    /*
387     * The bottom half reads the tv array, filled by the top half,
388     * and prints it to the circular text buffer, which is then consumed
389     * by reading processes
390     */
391
392    /* First write the number of interrupts that occurred before this bh */
393    written = sprintf((char *)short_head,"bh after %6i\n",savecount);
394    short_incr_bp(&short_head, written);
395
396    /*
397     * Then, write the time values. Write exactly 16 bytes at a time,
398     * so it aligns with PAGE_SIZE
399     */
400
401    do {
402        written = sprintf((char *)short_head,"%08u.%06u\n",
403                (int)(tv_tail->tv_sec % 100000000),
404                (int)(tv_tail->tv_usec));
405        short_incr_bp(&short_head, written);
406        short_incr_tv(&tv_tail);
407    } while (tv_tail != tv_head);
408
409    wake_up_interruptible(&short_queue); /* awake any reading process */
410}
```

在底半部函数中，把时间信息从tv_data数组中取出来，写到short_buffer缓冲区中，然后唤醒等待队列short_queue上的进程。这些进程将从short_buffer中读取时间信息。

三、文件操作函数

分析完了模块初始化函数,我们可以看设备文件操作函数了，文件操作函数集是short_fops：

```cpp
270struct file_operations short_fops = {
271    .owner   = THIS_MODULE,
272    .read    = short_read,
273    .write   = short_write,
274    .poll    = short_poll,
275    .open    = short_open,
276    .release = short_release,
277};
```

先看short_open函数：

```cpp
114int short_open (struct inode *inode, struct file *filp)
115{
116    extern struct file_operations short_i_fops;
117
118    if (iminor (inode) & 0x80)
119        filp->f_op = &short_i_fops; /* the interrupt-driven node */
120    return 0;
121}
```

118 – 119行，如果次设备号的第8位为1，重新设置文件操作函数集为short_i_fops。理解这样的设置可以看一下ldd3自带的short_load脚本，该脚本创建的设备节点/dev/shortint和/dev/shortprint的次设备号分别为128和129，如果对这两个节点进行操作，采用short_i_fops，即使用中断。对其它节点的操作，使用非中断操作。

```cpp
328struct file_operations short_i_fops = {
329    .owner   = THIS_MODULE,
330    .read    = short_i_read,
331    .write   = short_i_write,
332    .open    = short_open,
333    .release = short_release,
334};
```

下面看short_read的实现：

```cpp
190ssize_t short_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
191{
192    return do_short_read(filp->f_dentry->d_inode, filp, buf, count, f_pos);
193}

134ssize_t do_short_read (struct inode *inode, struct file *filp, char __user *buf,
135        size_t count, loff_t *f_pos)
136{
137    int retval = count, minor = iminor (inode);
138    unsigned long port = short_base + (minor&0x0f);
139    void *address = (void *) short_base + (minor&0x0f);
140    int mode = (minor&0x70) >> 4;
141    unsigned char *kbuf = kmalloc(count, GFP_KERNEL), *ptr;
142
143    if (!kbuf)
144        return -ENOMEM;
145    ptr = kbuf;
146
147    if (use_mem)
148        mode = SHORT_MEMORY;
149
150    switch(mode) {
151        case SHORT_STRING:
152        insb(port, ptr, count);
153        rmb();
154        break;
155
156        case SHORT_DEFAULT:
157        while (count--) {
```

```
158             *(ptr++) = inb(port);
159             rmb();
160         }
161         break;
162 
163         case SHORT_MEMORY:
164         while (count--) {
165             *ptr++ = ioread8(address);
166             rmb();
167         }
168         break;
169         case SHORT_PAUSE:
170         while (count--) {
171             *(ptr++) = inb_p(port);
172             rmb();
173         }
174         break;
175 
176         default: /* no more modes defined by now */
177         retval = -EINVAL;
178         break;
179     }
180     if ((retval > 0) && copy_to_user(buf, kbuf, retval))
181         retval = -EFAULT;
182     kfree(kbuf);
183     return retval;
184}
```

138行，确定要访问的端口。

139行，确定要访问的内存地址。

注意，对一个设备节点来说，要么是采用I/O端口，要么是采用I/O内存，不可能两个同时用，所以137和138行只有一个起作用，这里只是为减少程序代码而写在一起。理解这两句话，需要联系模块初始化函数short_init中的如下代码：

```
[cpp]
560     /* Get our needed resources. */
561     if (!use_mem) {
562         if (! request_region(short_base, SHORT_NR_PORTS, "short")) {
563             printk(KERN_INFO "short: can't get I/O port address 0x%lx\n",
564                    short_base);
565             return -ENODEV;
566         }
567 
568     } else {
569         if (! request_mem_region(short_base, SHORT_NR_PORTS, "short")) {
570             printk(KERN_INFO "short: can't get I/O mem address 0x%lx\n",
571                    short_base);
572             return -ENODEV;
573         }
574 
575         /* also, ioremap it */
576         short_base = (unsigned long) ioremap(short_base, SHORT_NR_PORTS);
577         /* Hmm... we should check the return value */
578     }
```

回到do_short_read函数：

140行，确定mode值，要理解这句，也要参考LDD3自带的short_load脚本对设备节点次设备号的设置。/dev/short0 - /dev/short7次设备号是0 - 7，对应的mode是0，/dev/short0p - /dev/short7p次设备号是16 - 23，对应的mode是1，/dev/short0s - /dev/short7s次设备号是32 - 39，对应的mode是2。

151 - 153行，使用insb(port, ptr, count)，从port端口一次读count个字节的数据到ptr指向的内存中；

157 - 160行，使用inb(port)一次从port端口读一个位数据，循环count次。

164 - 167行，使用ioread8(address)，从I/O内存address处读一个字节，循环count次。

169 – 173行，使用暂停式I/O函数inb_p(port)，一次从port端口读一个位数据，重复count次。

180行，将读到的数据拷贝到用户空间。

short_write函数的实现与short_read函数类似，只是方向相反而已，这里不再详细分析了。

下面我们来看使用中断的读函数short_i_read：

```cpp
281ssize_t short_i_read (struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
282{
283    int count0;
284    DEFINE_WAIT(wait);
285
286    while (short_head == short_tail) {
287        prepare_to_wait(&short_queue, &wait, TASK_INTERRUPTIBLE);
288        if (short_head == short_tail)
289            schedule();
290        finish_wait(&short_queue, &wait);
291        if (signal_pending (current))  /* a signal arrived */
292            return -ERESTARTSYS; /* tell the fs layer to handle it */
293    }
294    /* count0 is the number of readable data bytes */
295    count0 = short_head - short_tail;
296    if (count0 < 0) /* wrapped */
297        count0 = short_buffer + PAGE_SIZE - short_tail;
298    if (count0 < count) count = count0;
299
300    if (copy_to_user(buf, (char *)short_tail, count))
301        return -EFAULT;
302    short_incr_bp (&short_tail, count);
303    return count;
304}
```

284行，创建等待队列入口wait。

286行，如果short_head等于short_tail，说明short_buffer缓冲区中没有数据可读，需要休眠等待。前面在分析中断处理函数时，我们已经看到在short设备的中断处理函数中，会将数据写入short_buffer缓冲区并唤醒等待队列中的进程。

287 – 289，进入休眠。

290 – 293，被唤醒后执行清理工作。

300行，拷贝short_tail开始的count个数据到用户空间。

302行，更新short_tail位置。

下面我们来看使用中断的写函数short_i_write：

```cpp
306ssize_t short_i_write (struct file *filp, const char __user *buf, size_t count,
307        loff_t *f_pos)
308{
309    int written = 0, odd = *f_pos & 1;
310    unsigned long port = short_base; /* output to the parallel data latch */
311    void *address = (void *) short_base;
312
313    if (use_mem) {
314        while (written < count)
315            iowrite8(0xff * ((++written + odd) & 1), address);
316    } else {
317        while (written < count)
318            outb(0xff * ((++written + odd) & 1), port);
319    }
320
321    *f_pos += count;
322    return written;
323}
```

313 - 315，使用I/O内存，调用iowrite8写数据。

316 - 318，使用I/O端口，调用outb写数据。

更多 0

上一篇 LDD3源码分析之vmalloc
下一篇 LDD3源码分析之内存映射

顶 0 踩 0

主题推荐 源码 通信 硬件 linux内核 structure

猜你在找

Linux系统调用--fcntl函数详解
如何在windows下面编译u-boot （原发于：2012-07-24
国嵌视频学习——Linux内核驱动
STM32 对内部FLASH读写接口函数
开发板如何开启telnet服务
android 手机内存SWAP经验
扩展开放，修改关闭
u-boot编译笔记
find 与 grep
1. Android 2.3用ffmpeg替代stagefright自带的

查看评论

2楼 liuhaoyutz 2012-08-20 21:03发表

你说的对，insb函数是从port读count个字节到ptr指向的内存中。谢谢指正！

1楼 AzRael_AreS 2012-08-18 16:00发表

使用insb(port, ptr, count)，应该是从port端口一次读count个字节数据到ptr指向的内存中，不是count位吧？

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Java VPN Android iOS ERP IE10 Eclipse CRM JavaScript Ubuntu NFC WAP jQuery 数据库 BI HTML5 Spring Apache Hadoop .NET API HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP Spark HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap