Interrupt Handling

張大緯

CSIE, NCKU

The information on the slides are from
"*Linux Device Drivers*, Third Edition, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Copyright 2005 O'Reilly Media, Inc., 0-596-00590-3."

Introduction

- An interrupt
  - a signal that the HW can send when it wants the processor’s attention
- Drivers has to register/install their interrupt handlers

Installing an Interrupt Handler

- Interrupt lines are a limited resource
  - A driver has to explicitly request an interrupt line/IRQ channel
    - int request_irq (irq no., handler, flags, dev_name, dev_id)
      - Irq no.: The interrupt number being requested
      - Handler: pointer to the handling function being installed
      - Flags: a bit mask (described later)
      - Dev_name: the owner device of the interrupt
      - Dev_id: identify which device is interrupting
        - can be set to NULL if the interrupt is not shared

Flags of the request_irq()

- SA_INTERRUPT
  - A “fast” interrupt handle
  - Executed with interrupts disabled on the current processor
- SA_SHIRQ
  - The interrupt can be shared between devices
- SA_SAMPLE_RANDOM
  - Used as the seed of the random number generator

Installing an Interrupt Handler

- The time to install a interrupt handler
  - Driver initialization
    - An unused device may also hold an ISR channel
  - First open on the device
    - The common case
    - free_irq() on the last close

The /proc Interface

- See the */proc/interrupts*
  - Only shows the interrupts with installed handler

IRQ No. (in use)

2 CPUs

```
root@montalcino:/bike/corbet/write/ldd3/src/short# m /proc/interrupts
           CPU0       CPU1
  0:    4848108         34    IO-APIC-edge  timer
  2:          0          0          XT-PIC  cascade
  8:          3          1    IO-APIC-edge  rtc
 10:       4335          1   IO-APIC-level  aic7xxx
 11:       8903          0   IO-APIC-level  uhci_hcd
 12:         49          1    IO-APIC-edge  i8042
```

....

Interrupt controller

Device name (dev_name of request_irq()

# of interrupted delivered to the CPU
Most interrupts are delivered to CPU0 ← **cache locality**

The /proc Interface

- See the intr line of the ***/proc/stat***
  - shows interrupts even when the handler is not installed
    - More useful for drivers that acquire and release the interrupt at each open and close cycle

```
intr 5167833 5154006 2 0 2 4907 0 2 68 4 0 4406 9291 50 0 0
```

....

Number of total interrupt events

interrupt events of IRQ1

interrupt events of IRQ0

interrupt events of IRQ2

Autodetecting the IRQ Number

- The caller of the request_irq() should provide the IRQ number
- How to get the number?
  - Requires the user to specify at load time
    - Not Good…users usually don’t know the number
  - Build the knowledge into the driver

```
if (short_irq < 0) /* not yet specified: force the default on */
    switch(short_base) {
        case 0x378: short_irq = 7; break;
        case 0x278: short_irq = 2; break;
        case 0x3bc: short_irq = 5; break;
    }
```

//select the IRQ # based on base address of the IO region

Autodetecting the IRQ Number

- How to get the number?
  - Have the device tells you
    - Most modern devices announce the IRQ# it uses
      - Including PCI devices
    - The drivers get the IRQ# by reading a status byte from one of the device's I/O ports or PCI configuration space
  - Kernel-assisted probing
    - Not all of the devices announce their IRQ#
    - The drivers should probe the devices to get the #

  *on some platforms (PowerPC, M68k, most MIPS implementations, and both SPARC versions) probing is unnecessary

Autodetecting the IRQ Number -- Kernel-assisted Probing

- Basic idea
  - Driver tells the device to generate interrupts
  - Driver watches what interrupt line is triggered
  - Get the #
- Linux kernel offers a low-level facility for probing the interrupt number
  - works only for non-shared interrupts
  - The API
    - unsigned long probe_irq_on(void);
    - int probe_irq_off(unsigned long);

Autodetecting the IRQ Number -- Kernel-assisted Probing

- Steps
  - unsigned long probe_irq_on(void);
    - Returns a bit mask of unassigned interrupts
      - The mask will be passed to probe_irq_off() later
  - Make the device to generate interrupts
  - int probe_irq_off(unsigned long);
    - The return value is the IRQ number corresponding to the generated interrupt
    - IF more than one line has been triggered, the return value is negative

An Example of Kernel-assisted Probing

```
int count = 0;
do {
    unsigned long mask;

    mask = probe_irq_on( );
    outb_p(0x10,short_base+2); /* enable reporting */
    outb_p(0x00,short_base);   /* clear the bit */
    outb_p(0xFF,short_base);   /* set the bit: interrupt! */
    outb_p(0x00,short_base+2); /* disable reporting */
    udelay(5);  /* give it some time */
    short_irq = probe_irq_off(mask);

    if (short_irq == 0) { /* none of them? */
        printk(KERN_INFO "short: no irq reported by probe\n");
        short_irq = -1;
    }
```

...code skipped...

Autodetecting the IRQ Number -- DIY Probing

- Drivers can perform the probing by themselves
  - Rarely happened
- Basic idea
  - Register the IRQ handler for all the possible IRQ numbers
  - Trigger interrupts
  - See which IRQ line is triggered

Autodetecting the IRQ Number -- DIY Probing

assumes that 3, 5, 7, and 9 are the only possible IRQ numbers

```
int trials[ ] = {3, 5, 7, 9, 0};
int tried[ ] = {0, 0, 0, 0, 0};
int i, count = 0;
/*
* install the probing handler for all possible lines. Remember
* the result (0 for success, or -EBUSY) in order to only free
* what has been acquired
*/
for (i = 0; trials[i]; i++)                 // register all the possible lines
        tried[i] = request_irq( trials[i], short_probing, SA_INTERRUPT, "short probe", NULL);
do {
        short_irq = 0; /* none got, yet */
        outb_p(0x10,short_base+2); /* enable */
        outb_p(0x00,short_base);
        outb_p(0xFF,short_base); /* toggle the bit, trigger interrupt…. */
        outb_p(0x00,short_base+2); /* disable */
        udelay(5); /* give it some time; the ISR will set the short_irq variable */
```

Autodetecting the IRQ Number -- DIY Probing

```
    /* the value has been set by the handler */
    if (short_irq = = 0) { /* none of them? */
        printk(KERN_INFO "short: no irq reported by probe\n");
    }
    /*
    * If more than one line has been activated, the result is
    * negative. We should service the interrupt (but the lpt port
    * doesn't need it) and loop over again. Do it at most 5 times
    */
} while (short_irq <=0 && count++ < 5);

/* end of loop, uninstall the handler */
for (i = 0; trials[i]; i++)
    if (tried[i] = = 0)  free_irq(trials[i], NULL);
if (short_irq < 0)
    printk("short: probe failed %i times, giving up\n", count);
```

Autodetecting the IRQ Number -- DIY Probing

The interrupt handler, just set the variable **short_irq**

```
irqreturn_t short_probing (int irq, void *dev_id, struct pt_regs *regs)
{
        if (short_irq = = 0) short_irq = irq;       /* found */
        if (short_irq != irq) short_irq = -irq;     /* ambiguous */
        return IRQ_HANDLED;
}
```

If you don't know the possible IRQ numbers

-- probe all the free interrupts from IRQ 0 to IRQ NR_IRQS-1

Fast and Slow Handlers

- Fast interrupts
  - requested with the SA_INTERRUPT flag
  - executed with all other interrupts disabled on the current processor
    - other processors can still handle interrupts
  - Should be used only for use in a few, specific situations
- Slow interrupts
  - executed with interrupts enabled

The Internals of Interrupt Handling on the x86

- *entry.S*
  - lowest level of interrupt handling
    - assembly code
  - pushes the interrupt number on the stack and jumps to the interrupt dispatcher
- *do_IRQ() in irq.c*
  - Acknowledge the interrupt
    - the interrupt controller can go on to other things
  - Obtains a spinlock for the given IRQ number
    - preventing any other CPU from handling this IRQ
  - Clears a couple of status bits (including IRQ_WAITING)
    - We will see later
  - Call handle_IRQ_event()
  - Releases the spinlock
  - …

The Internals of Interrupt Handling on the x86

- *handle_IRQ_event()*
  - Enable interrupts (only for slow interrupts)
  - Invoke the handler
- After the invocation of the *handle_IRQ_event()*
  - Running software interrupts (e.g., tasklets…)
  - Back to regular work
    - May reschedule since a high-priority process may have be awaken

Kernel-assisted Probing Internals

- probe_irq_on() sets up the IRQ_WAITING flag for all the non-handler IRQs
- do_IRQ() clear the IRQ_WAITING flag
- probe_irq_off() checks all the non-handler IRQs with IRQ_WAITING flag cleared

Implementing a Handler

- Restrictions (the same as timers)
  - can’t transfer data to or from user space,
  - cannot do anything that would sleep
    - *wait_event, lock semaphore…*
  - cannot call *schedule*
- The jobs of an interrupt handler
  - Do everything that you want to or have to
    - Clear the pending bit
      - Depends on the device
    - Wakeup some processes
    - Schedule a tasklet or workqueue for long computation
    - ….

An Interrupt Handler Example

```
static inline void short_incr_bp(volatile unsigned long *index, int delta)
{
        unsigned long new = *index + delta;
        barrier( ); /* Don't optimize these two together */
        /* optimization could expose an incorrect value of the index for a brief period in
        the case where the buffer wraps */
        *index = (new >= (short_buffer + PAGE_SIZE)) ? short_buffer : new; // wrap as needed
}

irqreturn_t short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        struct timeval tv; int written;
        do_gettimeofday(&tv);
        /* Write a 16 byte record. Assume PAGE_SIZE is a multiple of 16 */
        written = sprintf((char *)short_head,"%08u.%06u\n", (int)(tv.tv_sec % 100000000),
                                                                (int)(tv.tv_usec));
        BUG_ON(written != 16);
        short_incr_bp (&short_head, written);  // increase the write (i.e., short_head) pointer
        wake_up_interruptible(&short_queue); /* awake any reading process */
        return IRQ_HANDLED;
}
```

Arguments of an Interrupt Handler

```
static irqreturn_t sample_interrupt(int irq, void *dev_id, struct pt_regs
                                  *regs)
{
    struct sample_dev *dev = dev_id;

    /* now `dev' points to the right hardware item */
    /* .... */
}

 static void sample_open(struct inode *inode, struct file *filp)
 {
     struct sample_dev *dev = hwinfo + MINOR(inode->i_rdev);
     request_irq(dev->irq, sample_interrupt,
                 0 /* flags */, "sample", dev /* dev_id */);
     /*....*/
     return 0;
 }
```

dev_id is recorded in the kernel when request_irq() is invoked, and passed to the interrupt handler when an interrupt happens

Arguments of an Interrupt Handler

- irq
  - The irq number
- dev_id
  - recorded in the kernel when request_irq()
  - passed to the interrupt handler when an interrupt happens
  - driver writers usually use pointers to the device-specific data structures as the dev_id
    - Allow a driver to manage multiple devices
- regs
  - saved processor context when the interrupt happened
  - used for monitoring and debugging
  - rarely used in normal drivers

Return Value of an Interrupt Handler

- Interrupt handlers should return a value indicating whether there was actually an interrupt to handle
  - Return IRQ_HANDLED if you were able to handle the interrupt
  - Return IRQ_NONE for other cases
- Allow kernel to detect and suppress spurious interrupts

Enabling and Disabling Interrupts

- Should be used rarely
- Disabling a single interrupt
  - void disable_irq(int irq);
    - also waits for a currently executing interrupt handler to complete
  - void disable_irq_nosync(int irq);
    - No wait, your driver may suffer from race condition
- Enabling a single interrupt
  - void enable_irq(int irq);

across all
processors

Enabling and Disabling Interrupts

- Disabling all interrupts on the local processor
  - void local_irq_save(unsigned long flags);
  - void local_irq_disable(void);
- Re-enabling all interrupts on the local processor
  - void local_irq_restore(unsigned long flags);
  - void local_irq_enable(void);
- In Linux 2.6, you can **NOT** disable **all** interrupts globally across the entire system

Top and Bottom Halves

- Top Half Interrupt Handler
  - Need to finish up quickly, and not keep interrupts blocked for long
  - The routine actually registered by request_irq()
- Bottom Half Interrupt Handler
  - E.g., tasklets, workqueues…
  - Scheduled by the top half to be executed later
  - Perform whatever other work is required
  - All interrupts are enabled while bottom half is executing
  - Several interrupts may occur before BH is executed

ISR + Tasklet

- Tasklet
  - run in software interrupt context
  - may be scheduled to run multiple times
  - No tasklet ever runs in parallel with itself
  - Can execute in parallel with other tasklets
    - on SMP systems
    - Locking may be needed if your driver uses multiple tasks
  - run on the same CPU as the function that first schedules them
  - declared with
    - DECLARE_TASKLET(name, function, data);

ISR + Tasklet

The ISR

perform the time-critical jobs and defer the remaining jobs to the tasklet

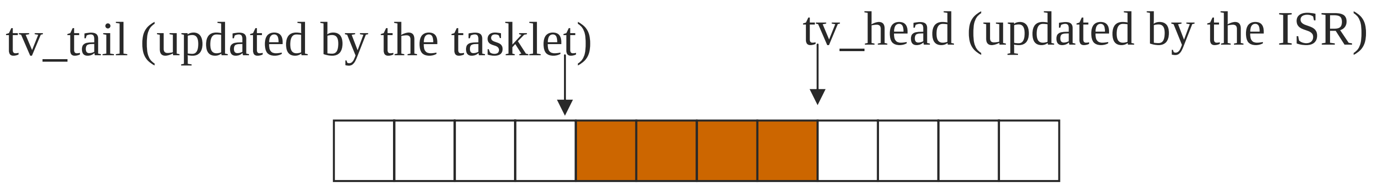
```
irqreturn_t short_tl_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        do_gettimeofday((struct timeval *) tv_head);

        short_incr_tv(&tv_head);                    //update the head pointer of the buffer

        tasklet_schedule( &short_tasklet );

        short_wq_count++;                           /* record that an interrupt arrived */

        return IRQ_HANDLED;
}
```



ISR + Tasklet

The tasklet

print the # of interrupts that have happened since the last running of the tasklet
print the time values recorded by the ISRs

```
void short_do_tasklet (unsigned long unused)
{
	int savecount = short_wq_count, written;
	short_wq_count = 0; /* we have already been removed from the queue */

	/* write the number of interrupts that occurred before this bh */
	written = sprintf((char *)short_head,"bh after %6i\n",savecount);
	short_incr_bp(&short_head, written);
	/* write the time values. Write exactly 16 bytes at a time, so it aligns with PAGE_SIZE*/
	do {
		written = sprintf((char *)short_head,"%08u.%06u\n",
		(int)(tv_tail->tv_sec % 100000000),  (int)(tv_tail->tv_usec));
		short_incr_bp(&short_head, written);
		short_incr_tv(&tv_tail);
	} while (tv_tail != tv_head);
	wake_up_interruptible(&short_queue);		/* awake any reading process */
}
```

ISR+Workqueue

- Workqueue function
  - Run in the process context
    - Can sleep
  - Cannot access the user space
  - Init a workqueue element
    - static struct work_struct short_wq;
    - INIT_WORK(&short_wq, (void (*)(void *)) short_do_tasklet, NULL);
  - Schedule the work queue element
    - schedule_work(&short_wq);

ISR+Workqueue

The ISR

```
irqreturn_t short_wq_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        /* Grab the current time information. */
        do_gettimeofday((struct timeval *) tv_head);
        short_incr_tv(&tv_head);

        /* Queue the bh. Don't worry about multiple enqueueing */
        schedule_work(&short_wq);

        short_wq_count++; /* record that an interrupt arrived */
        return IRQ_HANDLED;
}
```

Interrupt Sharing

- Allow more than one devices use the same IRQ #
- Reduce the chances of interrupt conflicts

Installing a Shared Handler

- Notes on the request_irq()
  - SA_SHIRQ must be specified in the flags argument
    - All drivers for the IRQ# should specify SA_SHIRQ
  - dev_id argument MUST be unique
    - cannot be set to NULL
    - It is used to differentiate between different devices
    - Sometimes you can poll the device to see if it generated an interrupt or not ➔ you can omit the use of dev_id here
      - However, not all the hardware devices support that
- Notes on the free_irq()
  - Pass your unique dev_id to that function
    - Allow kernel to know which ISR is going to leave

Installing a Shared Handler

- All the ISRs for the same IRQ are linked in a list by the kernel
- Kernel invokes all the ISRs on the list when the interrupt happens
  - Each ISR checks the dev_id argument
    - For me: do the interrupt handling jobs
    - Not for me: return IRQ_NONE immediately

Installing a Shared Handler

- Kernel-assisted probing doesn't work here
  - You can only probe free IRQs, not the IRQs that have already been used by others
  - Fortunately, most hardware designed for interrupt sharing is also able to tell the processor which interrupt it is using
    - No explicit probing is required
- Do **NOT** call *enable_irq* or *disable_irq*
  - You will disable others ISRs…
  - Remember that the IRQ is shared, be polite…

Shared Handler (Example)

```
irqreturn_t short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        int value, written;
        struct timeval tv;
        /* If it wasn't for this driver, return immediately */
        value = inb(short_base);
        if (!(value & 0x80))            return IRQ_NONE;

        /* clear the interrupting bit */
        outb(value & 0x7F, short_base);

        do_gettimeofday(&tv);
        written = sprintf((char *)short_head,"%08u.%06u\n",
        (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
        short_incr_bp(&short_head, written);
        wake_up_interruptible(&short_queue);            /* awake any reading process */
        return IRQ_HANDLED;
}
```

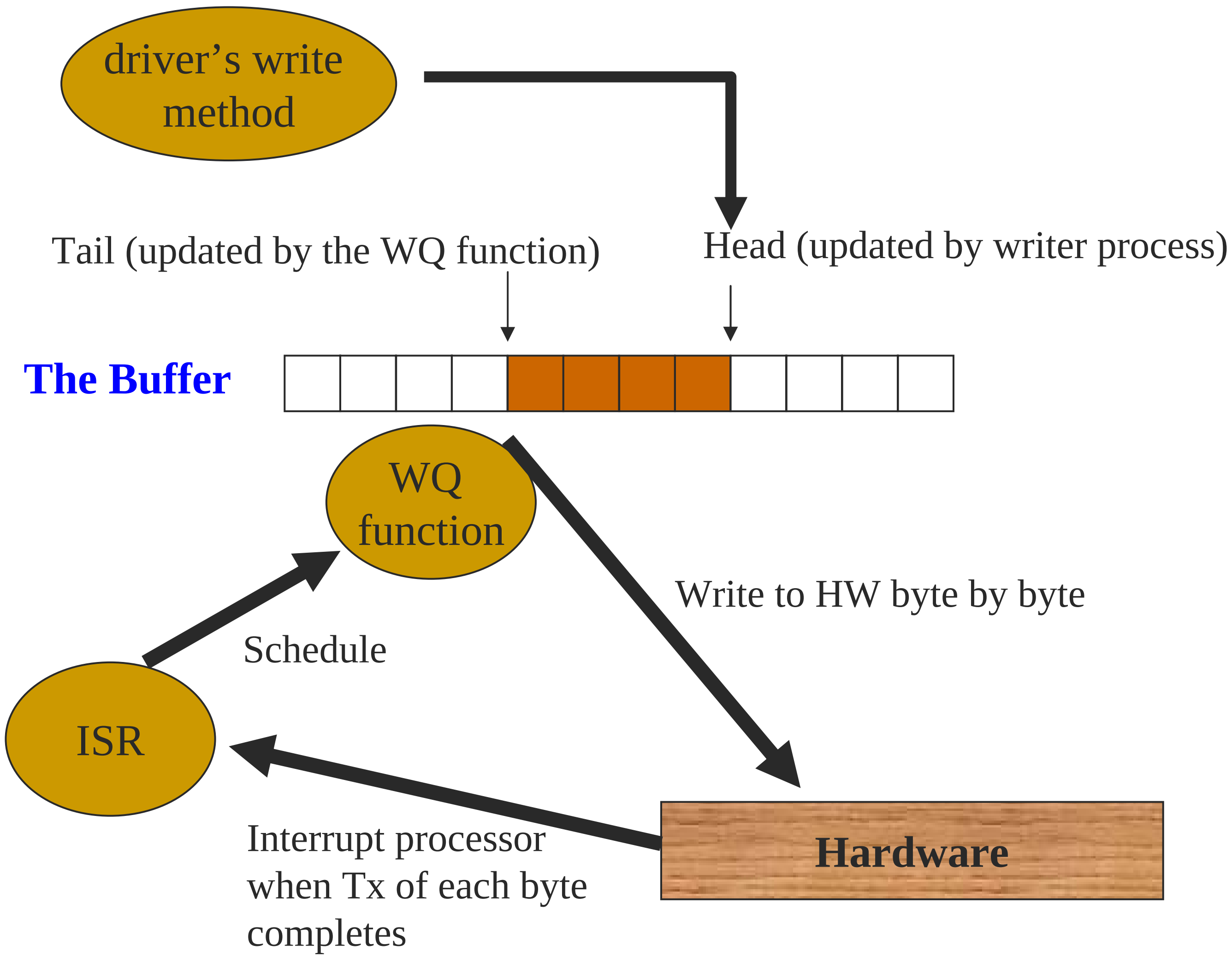
Shared Interrupts and /proc/interrupts

```
        CPU0
 0:  892335412      XT-PIC  timer
 1:     453971      XT-PIC  i8042
 2:          0      XT-PIC  cascade
 5:          0      XT-PIC  libata, ehci_hcd
 8:          0      XT-PIC  rtc
 9:          0      XT-PIC  acpi
10:   11365067      XT-PIC  ide2, uhci_hcd, uhci_hcd, SysKonnect SK-98xx, EMU10K1
11:    4391962      XT-PIC  uhci_hcd, uhci_hcd
12:        224      XT-PIC  i8042
14:    2787721      XT-PIC  ide0
15:     203048      XT-PIC  ide1
```

Interrupt-Driven I/O

- Buffering
  - Improving the performance for slow IO
    - Detach data Tx/Rx from the *write/read* system calls
- Interrupt-driven Buffering I/O
  - Input
    - device interrupts the processor when new data has arrived
    - input buffer is filled at interrupt time
    - input buffer is emptied by processes that read the device
  - Output *(we will see an example later …)*
    - an interrupt happens
      - When it is ready to accept new data
      - To acknowledge a successful data transfer
    - output buffer is filled by processes that write to the device
    - output buffer is emptied at interrupt time

A Write-Buffering Example



The Write Method

```
while (written < count) {                // while the currently written size is less than the user-specified one
        /* Hang out until some buffer space is available. */
        space = shortp_out_space( );
        if (space <= 0) {                // no free space, wait…
                if (wait_event_interruptible(shortp_out_queue, (space = shortp_out_space( )) > 0))
                                goto out;
        }
        /* Move data into the buffer. */
        if ((space + written) > count)          space = count - written;
        if (copy_from_user((char *) shortp_out_head, buf, space)) {   // copy the user data to the buffer head
                up(&shortp_out_sem);            return -EFAULT;
        }
        shortp_incr_out_bp(&shortp_out_head, space);     // update the buffer head
        buf += space;     written += space;
        /* If no output is active, make it active. */
        spin_lock_irqsave(&shortp_out_lock, flags);
        if (! shortp_output_active)  shortp_start_output( );     // this function triggers the output WQ function
        spin_unlock_irqrestore(&shortp_out_lock, flags);
}
out:
*f_pos += written;
```

Scheduling the WorkQueue Function

```
static void shortp_start_output(void)
{
        if (shortp_output_active) /* Should never happen */
        return;

        //setup an timer to handle the case of interrupt missing…
        shortp_output_active = 1;
        shortp_timer.expires = jiffies + TIMEOUT;
        add_timer(&shortp_timer);

        /* And get the process going. */
        queue_work(shortp_workqueue, &shortp_work);
}
```

You can, occasionally, lose an interrupt from the device

-- the timer prevent halting your driver when an interrupt is missed

The WorkQueue Function

```
spin_lock_irqsave(&shortp_out_lock, flags);

/* Have we written everything? */
if (shortp_out_head = = shortp_out_tail) { /* empty, do not need to send bytes to the device */
        shortp_output_active = 0;
        wake_up_interruptible(&shortp_empty_queue);    // wake up tasks that wait for the
                                                       // emptiness of the buffer.
                                                       // e.g., someone who wants to close the device
        del_timer(&shortp_timer);
}
/* Nope, write another byte */
else
        shortp_do_write( );                            // write a byte to the device HW

/* wakeup writer if the free space is larger than a threashold: SP_MIN_SPACE */
if (((PAGE_SIZE + shortp_out_tail -shortp_out_head) % PAGE_SIZE) > SP_MIN_SPACE)
{
        wake_up_interruptible(&shortp_out_queue);
}
spin_unlock_irqrestore(&shortp_out_lock, flags);
```

The ISR

An interrupt happens when the **Tx** of **each byte** is **completed**...

```
static irqreturn_t shortp_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        if (! shortp_output_active)
                return IRQ_NONE;

        /* Remember the time, and farm off the rest to the workqueue function */
        do_gettimeofday(&shortp_tv);

        queue_work(shortp_workqueue, &shortp_work);  //schedule the WQ to send another byte

        return IRQ_HANDLED;
}
```

The Timeout Function

- Timer expires
  - The Tx complete interrupt is delayed? or missed?
- Query the hardware to see if the interrupt is delayed due to the busyness of the device
  - Device busy ➔ wait
  - Device ready ➔ the interrupt is missed
    - Call the WQ function manually to send the next byte