CHAPTER 10
Interrupt Handling

Although some devices can be controlled using nothing but their I/O regions, most real devices are a bit more complicated than that. Devices have to deal with the external world, which often includes things such as spinning disks, moving tape, wires to distant places, and so on. Much has to be done in a time frame that is different from, and far slower than, that of the processor. Since it is almost always undesirable to have the processor wait on external events, there must be a way for a device to let the processor know when something has happened.

That way, of course, is interrupts. An *interrupt* is simply a signal that the hardware can send when it wants the processor's attention. Linux handles interrupts in much the same way that it handles signals in user space. For the most part, a driver need only register a handler for its device's interrupts, and handle them properly when they arrive. Of course, underneath that simple picture there is some complexity; in particular, interrupt handlers are somewhat limited in the actions they can perform as a result of how they are run.

It is difficult to demonstrate the use of interrupts without a real hardware device to generate them. Thus, the sample code used in this chapter works with the parallel port. Such ports are starting to become scarce on modern hardware, but, with luck, most people are still able to get their hands on a system with an available port. We'll be working with the *short* module from the previous chapter; with some small additions it can generate and handle interrupts from the parallel port. The module's name, *short*, actually means *short int* (it is C, isn't it?), to remind us that it handles *int*errupts.

Before we get into the topic, however, it is time for one cautionary note. Interrupt handlers, by their nature, run concurrently with other code. Thus, they inevitably raise issues of concurrency and contention for data structures and hardware. If you succumbed to the temptation to pass over the discussion in Chapter 5, we understand. But we also recommend that you turn back and have another look now. A solid understanding of concurrency control techniques is vital when working with interrupts.

Preparing the Parallel Port

Although the parallel interface is simple, it can trigger interrupts. This capability is used by the printer to notify the *lp* driver that it is ready to accept the next character in the buffer.

Like most devices, the parallel port doesn't actually generate interrupts before it's instructed to do so; the parallel standard states that setting bit 4 of port 2 (`0x37a`, `0x27a`, or whatever) enables interrupt reporting. A simple *outb* call to set the bit is performed by *short* at module initialization.

Once interrupts are enabled, the parallel interface generates an interrupt whenever the electrical signal at pin 10 (the so-called ACK bit) changes from low to high. The simplest way to force the interface to generate interrupts (short of hooking up a printer to the port) is to connect pins 9 and 10 of the parallel connector. A short length of wire inserted into the appropriate holes in the parallel port connector on the back of your system creates this connection. The pinout of the parallel port is shown in Figure 9-1.

Pin 9 is the most significant bit of the parallel data byte. If you write binary data to */dev/short0*, you generate several interrupts. Writing ASCII text to the port won't generate any interrupts, though, because the ASCII character set has no entries with the top bit set.

If you'd rather avoid wiring pins together, but you do have a printer at hand, you can run the sample interrupt handler using a real printer, as shown later. However, note that the probing functions we introduce depend on the jumper between pin 9 and 10 being in place, and you need it to experiment with probing using our code.

Installing an Interrupt Handler

If you want to actually "see" interrupts being generated, writing to the hardware device isn't enough; a software handler must be configured in the system. If the Linux kernel hasn't been told to expect your interrupt, it simply acknowledges and ignores it.

Interrupt lines are a precious and often limited resource, particularly when there are only 15 or 16 of them. The kernel keeps a registry of interrupt lines, similar to the registry of I/O ports. A module is expected to request an interrupt channel (or IRQ, for interrupt request) before using it and to release it when finished. In many situations, modules are also expected to be able to share interrupt lines with other drivers, as we will see. The following functions, declared in *<linux/interrupt.h>*, implement the interrupt registration interface:

```
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
                    const char *dev_name,
                    void *dev_id);

    void free_irq(unsigned int irq, void *dev_id);
```

The value returned from *request_irq* to the requesting function is either `0` to indicate success or a negative error code, as usual. It's not uncommon for the function to return `-EBUSY` to signal that another driver is already using the requested interrupt line. The arguments to the functions are as follows:

`unsigned int irq`
: The interrupt number being requested.

`irqreturn_t (*handler)(int, void *, struct pt_regs *)`
: The pointer to the handling function being installed. We discuss the arguments to this function and its return value later in this chapter.

`unsigned long flags`
: As you might expect, a bit mask of options (described later) related to interrupt management.

`const char *dev_name`
: The string passed to *request_irq* is used in */proc/interrupts* to show the owner of the interrupt (see the next section).

`void *dev_id`
: Pointer used for shared interrupt lines. It is a unique identifier that is used when the interrupt line is freed and that may also be used by the driver to point to its own private data area (to identify which device is interrupting). If the interrupt is not shared, `dev_id` can be set to `NULL`, but it a good idea anyway to use this item to point to the device structure. We'll see a practical use for `dev_id` in the section "Implementing a Handler."

The bits that can be set in `flags` are as follows:

`SA_INTERRUPT`
: When set, this indicates a "fast" interrupt handler. Fast handlers are executed with interrupts disabled on the current processor (the topic is covered in the section "Fast and Slow Handlers").

`SA_SHIRQ`
: This bit signals that the interrupt can be shared between devices. The concept of sharing is outlined in the section "Interrupt Sharing."

`SA_SAMPLE_RANDOM`
: This bit indicates that the generated interrupts can contribute to the entropy pool used by */dev/random* and */dev/urandom*. These devices return truly random numbers when read and are designed to help application software choose secure keys for encryption. Such random numbers are extracted from an entropy pool that is contributed by various random events. If your device generates interrupts at truly random times, you should set this flag. If, on the other hand, your interrupts are

predictable (for example, vertical blanking of a frame grabber), the flag is not worth setting—it wouldn't contribute to system entropy anyway. Devices that could be influenced by attackers should not set this flag; for example, network drivers can be subjected to predictable packet timing from outside and should not contribute to the entropy pool. See the comments in *drivers/char/random.c* for more information.

The interrupt handler can be installed either at driver initialization or when the device is first opened. Although installing the interrupt handler from within the module's initialization function might sound like a good idea, it often isn't, especially if your device does not share interrupts. Because the number of interrupt lines is limited, you don't want to waste them. You can easily end up with more devices in your computer than there are interrupts. If a module requests an IRQ at initialization, it prevents any other driver from using the interrupt, even if the device holding it is never used. Requesting the interrupt at device open, on the other hand, allows some sharing of resources.

It is possible, for example, to run a frame grabber on the same interrupt as a modem, as long as you don't use the two devices at the same time. It is quite common for users to load the module for a special device at system boot, even if the device is rarely used. A data acquisition gadget might use the same interrupt as the second serial port. While it's not too hard to avoid connecting to your Internet service provider (ISP) during data acquisition, being forced to unload a module in order to use the modem is really unpleasant.

The correct place to call *request_irq* is when the device is first opened, *before* the hardware is instructed to generate interrupts. The place to call *free_irq* is the last time the device is closed, *after* the hardware is told not to interrupt the processor any more. The disadvantage of this technique is that you need to keep a per-device open count so that you know when interrupts can be disabled.

This discussion notwithstanding, *short* requests its interrupt line at load time. This was done so that you can run the test programs without having to run an extra process to keep the device open. *short*, therefore, requests the interrupt from within its initialization function (*short_init*) instead of doing it in *short_open*, as a real device driver would.

The interrupt requested by the following code is `short_irq`. The actual assignment of the variable (i.e., determining which IRQ to use) is shown later, since it is not relevant to the current discussion. `short_base` is the base I/O address of the parallel interface being used; register 2 of the interface is written to enable interrupt reporting.

```
if (short_irq >= 0) {
    result = request_irq(short_irq, short_interrupt,
            SA_INTERRUPT, "short", NULL);
   if (result) {
        printk(KERN_INFO "short: can't get assigned irq %i\n",
                short_irq);
            short_irq = -1;
        }
        else { /* actually enable it -- assume this *is* a parallel port */
            outb(0x10,short_base+2);
        }
    }
```

The code shows that the handler being installed is a fast handler (`SA_INTERRUPT`), doesn't support interrupt sharing (`SA_SHIRQ` is missing), and doesn't contribute to system entropy (`SA_SAMPLE_RANDOM` is missing, too). The *outb* call then enables interrupt reporting for the parallel port.

For what it's worth, the i386 and x86_64 architectures define a function for querying the availability of an interrupt line:

```
int can_request_irq(unsigned int irq, unsigned long flags);
```

This function returns a nonzero value if an attempt to allocate the given interrupt succeeds. Note, however, that things can always change between calls to *can_request_irq* and *request_irq*.

The /proc Interface

Whenever a hardware interrupt reaches the processor, an internal counter is incremented, providing a way to check whether the device is working as expected. Reported interrupts are shown in */proc/interrupts*. The following snapshot was taken on a two-processor Pentium system:

```
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
NMI:          0          0
LOC:    4848187    4848186
ERR:          0
MIS:          0
```

The first column is the IRQ number. You can see from the IRQs that are missing that the file shows only interrupts corresponding to installed handlers. For example, the first serial port (which uses interrupt number 4) is not shown, indicating that the modem isn't being used. In fact, even if the modem had been used earlier but wasn't in use at the time of the snapshot, it would not show up in the file; the serial ports are well behaved and release their interrupt handlers when the device is closed.

The */proc/interrupts* display shows how many interrupts have been delivered to each CPU on the system. As you can see from the output, the Linux kernel generally handles interrupts on the first CPU as a way of maximizing cache locality.* The last two columns give information on the programmable interrupt controller that handles the interrupt (and that a driver writer does not need to worry about), and the name(s) of the device(s) that have registered handlers for the interrupt (as specified in the `dev_name` argument to *request_irq*).

The */proc* tree contains another interrupt-related file, */proc/stat*; sometimes you'll find one file more useful and sometimes you'll prefer the other. */proc/stat* records several low-level statistics about system activity, including (but not limited to) the number of interrupts received since system boot. Each line of *stat* begins with a text string that is the key to the line; the `intr` mark is what we are looking for. The following (truncated) snapshot was taken shortly after the previous one:

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

The first number is the total of all interrupts, while each of the others represents a single IRQ line, starting with interrupt `0`. All of the counts are summed across all processors in the system. This snapshot shows that interrupt number 4 has been used 4907 times, even though no handler is *currently* installed. If the driver you're testing acquires and releases the interrupt at each open and close cycle, you may find */proc/stat* more useful than */proc/interrupts*.

Another difference between the two files is that *interrupts* is not architecture dependent (except, perhaps, for a couple of lines at the end), whereas *stat* is; the number of fields depends on the hardware underlying the kernel. The number of available interrupts varies from as few as 15 on the SPARC to as many as 256 on the IA-64 and a few other systems. It's interesting to note that the number of interrupts defined on the x86 is currently 224, not 16 as you may expect; this, as explained in *include/asm-i386/irq.h*, depends on Linux using the architectural limit instead of an implementation-specific limit (such as the 16 interrupt sources of the old-fashioned PC interrupt controller).

The following is a snapshot of */proc/interrupts* taken on an IA-64 system. As you can see, besides different hardware routing of common interrupt sources, the output is very similar to that from the 32-bit system shown earlier.

```
           CPU0       CPU1
 27:       1705      34141  IO-SAPIC-level  qla1280
 40:          0          0           SAPIC  perfmon
 43:        913       6960  IO-SAPIC-level  eth0
 47:      26722        146  IO-SAPIC-level  usb-uhci
 64:          3          6   IO-SAPIC-edge  ide0
 80:          4          2   IO-SAPIC-edge  keyboard
 89:          0          0   IO-SAPIC-edge  PS/2 Mouse
239:    5606341    5606052           SAPIC  timer
```

* Although, some larger systems explicitly use interrupt balancing schemes to spread the interrupt load across the system.

```
254:      67575      52815          SAPIC  IPI
NMI:          0          0
ERR:          0
```

Autodetecting the IRQ Number

One of the most challenging problems for a driver at initialization time can be how to determine which IRQ line is going to be used by the device. The driver needs the information in order to correctly install the handler. Even though a programmer could require the user to specify the interrupt number at load time, this is a bad practice, because most of the time the user doesn't know the number, either because he didn't configure the jumpers or because the device is jumperless. Most users want their hardware to "just work" and are not interested in issues like interrupt numbers. So autodetection of the interrupt number is a basic requirement for driver usability.

Sometimes autodetection depends on the knowledge that some devices feature a default behavior that rarely, if ever, changes. In this case, the driver might assume that the default values apply. This is exactly how *short* behaves by default with the parallel port. The implementation is straightforward, as shown by *short* itself:

```
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;
    }
```

The code assigns the interrupt number according to the chosen base I/O address, while allowing the user to override the default at load time with something like:

```
insmod ./short.ko irq=x
```

`short_base` defaults to `0x378`, so `short_irq` defaults to `7`.

Some devices are more advanced in design and simply "announce" which interrupt they're going to use. In this case, the driver retrieves the interrupt number by reading a status byte from one of the device's I/O ports or PCI configuration space. When the target device is one that has the ability to tell the driver which interrupt it is going to use, autodetecting the IRQ number just means probing the device, with no additional work required to probe the interrupt. Most modern hardware works this way, fortunately; for example, the PCI standard solves the problem by requiring peripheral devices to declare what interrupt line(s) they are going to use. The PCI standard is discussed in Chapter 12.

Unfortunately, not every device is programmer friendly, and autodetection might require some probing. The technique is quite simple: the driver tells the device to generate interrupts and watches what happens. If everything goes well, only one interrupt line is activated.

Although probing is simple in theory, the actual implementation might be unclear. We look at two ways to perform the task: calling kernel-defined helper functions and implementing our own version.

Kernel-assisted probing

The Linux kernel offers a low-level facility for probing the interrupt number. It works for only nonshared interrupts, but most hardware that is capable of working in a shared interrupt mode provides better ways of finding the configured interrupt number anyway. The facility consists of two functions, declared in *<linux/interrupt.h>* (which also describes the probing machinery):

`unsigned long probe_irq_on(void);`
: This function returns a bit mask of unassigned interrupts. The driver must preserve the returned bit mask, and pass it to *probe_irq_off* later. After this call, the driver should arrange for its device to generate at least one interrupt.

`int probe_irq_off(unsigned long);`
: After the device has requested an interrupt, the driver calls this function, passing as its argument the bit mask previously returned by *probe_irq_on*. *probe_irq_off* returns the number of the interrupt that was issued after "probe_on." If no interrupts occurred, `0` is returned (therefore, IRQ `0` can't be probed for, but no custom device can use it on any of the supported architectures anyway). If more than one interrupt occurred (ambiguous detection), *probe_irq_off* returns a negative value.

The programmer should be careful to enable interrupts on the device *after* the call to *probe_irq_on* and to disable them *before* calling *probe_irq_off*. Additionally, you must remember to service the pending interrupt in your device after *probe_irq_off*.

The *short* module demonstrates how to use such probing. If you load the module with `probe=1`, the following code is executed to detect your interrupt line, provided pins 9 and 10 of the parallel connector are bound together:

```
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;
    }
```

```
        /*
         * if more than one line has been activated, the result is
         * negative. We should service the interrupt (no need for lpt port)
         * and loop over again. Loop at most five times, then give up
         */
    } while (short_irq < 0 && count++ < 5);
    if (short_irq < 0)
        printk("short: probe failed %i times, giving up\n", count);
```

Note the use of *udelay* before calling *probe_irq_off*. Depending on the speed of your processor, you may have to wait for a brief period to give the interrupt time to actually be delivered.

Probing might be a lengthy task. While this is not true for *short*, probing a frame grabber, for example, requires a delay of at least 20 ms (which is ages for the processor), and other devices might take even longer. Therefore, it's best to probe for the interrupt line only once, at module initialization, independently of whether you install the handler at device open (as you should) or within the initialization function (which is not recommended).

It's interesting to note that on some platforms (PowerPC, M68k, most MIPS implementations, and both SPARC versions) probing is unnecessary, and, therefore, the previous functions are just empty placeholders, sometimes called "useless ISA nonsense." On other platforms, probing is implemented only for ISA devices. Anyway, most architectures define the functions (even if they are empty) to ease porting existing device drivers.

Do-it-yourself probing

Probing can also be implemented in the driver itself without too much trouble. It is a rare driver that must implement its own probing, but seeing how it works gives some insight into the process. To that end, the *short* module performs do-it-yourself detection of the IRQ line if it is loaded with `probe=2`.

The mechanism is the same as the one described earlier: enable all unused interrupts, then wait and see what happens. We can, however, exploit our knowledge of the device. Often a device can be configured to use one IRQ number from a set of three or four; probing just those IRQs enables us to detect the right one, without having to test for all possible IRQs.

The *short* implementation assumes that `3`, `5`, `7`, and `9` are the only possible IRQ values. These numbers are actually the values that some parallel devices allow you to select.

The following code probes by testing all "possible" interrupts and looking at what happens. The `trials` array lists the IRQs to try and has `0` as the end marker; the `tried` array is used to keep track of which handlers have actually been registered by this driver.

```
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++)
    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 */
    outb_p(0x00,short_base+2); /* disable */
    udelay(5);  /* give it some time */

    /* 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);
```

You might not know in advance what the “possible” IRQ values are. In that case, you need to probe all the free interrupts, instead of limiting yourself to a few `trials[ ]`. To probe for all interrupts, you have to probe from IRQ `0` to IRQ `NR_IRQS-1`, where `NR_IRQS` is defined in *<asm/irq.h>* and is platform dependent.

Now we are missing only the probing handler itself. The handler’s role is to update `short_irq` according to which interrupts are actually received. A `0` value in `short_irq` means “nothing yet,” while a negative value means “ambiguous.” These values were chosen to be consistent with *probe_irq_off* and to allow the same code to call either kind of probing within *short.c*.

```
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;
    }
```

The arguments to the handler are described later. Knowing that `irq` is the interrupt being handled should be sufficient to understand the function just shown.

Fast and Slow Handlers

Older versions of the Linux kernel took great pains to distinguish between "fast" and "slow" interrupts. Fast interrupts were those that could be handled very quickly, whereas handling slow interrupts took significantly longer. Slow interrupts could be sufficiently demanding of the processor, and it was worthwhile to reenable interrupts while they were being handled. Otherwise, tasks requiring quick attention could be delayed for too long.

In modern kernels, most of the differences between fast and slow interrupts have disappeared. There remains only one: fast interrupts (those that were requested with the `SA_INTERRUPT` flag) are executed with all other interrupts disabled on the current processor. Note that other processors can still handle interrupts, although you will never see two processors handling the same IRQ at the same time.

So, which type of interrupt should your driver use? On modern systems, `SA_INTERRUPT` is intended only for use in a few, specific situations such as timer interrupts. Unless you have a strong reason to run your interrupt handler with other interrupts disabled, you should not use `SA_INTERRUPT`.

This description should satisfy most readers, although someone with a taste for hardware and some experience with her computer might be interested in going deeper. If you don't care about the internal details, you can skip to the next section.

The internals of interrupt handling on the x86

This description has been extrapolated from *arch/i386/kernel/irq.c*, *arch/i386/kernel/apic.c*, *arch/i386/kernel/entry.S*, *arch/i386/kernel/i8259.c*, and *include/asm-i386/hw_irq.h* as they appear in the 2.6 kernels; although the general concepts remain the same, the hardware details differ on other platforms.

The lowest level of interrupt handling can be found in *entry.S*, an assembly-language file that handles much of the machine-level work. By way of a bit of assembler trickery and some macros, a bit of code is assigned to every possible interrupt. In each case, the code pushes the interrupt number on the stack and jumps to a common segment, which calls *do_IRQ*, defined in *irq.c*.

The first thing *do_IRQ* does is to acknowledge the interrupt so that the interrupt controller can go on to other things. It then obtains a spinlock for the given IRQ number, thus preventing any other CPU from handling this IRQ. It clears a couple of status bits (including one called IRQ_WAITING that we'll look at shortly) and then looks up the handler(s) for this particular IRQ. If there is no handler, there's nothing to do; the spinlock is released, any pending software interrupts are handled, and *do_IRQ* returns.

Usually, however, if a device is interrupting, there is at least one handler registered for its IRQ as well. The function *handle_IRQ_event* is called to actually invoke the handlers. If the handler is of the slow variety (SA_INTERRUPT is not set), interrupts are reenabled in the hardware, and the handler is invoked. Then it's just a matter of cleaning up, running software interrupts, and getting back to regular work. The "regular work" may well have changed as a result of an interrupt (the handler could *wake_up* a process, for example), so the last thing that happens on return from an interrupt is a possible rescheduling of the processor.

Probing for IRQs is done by setting the IRQ_WAITING status bit for each IRQ that currently lacks a handler. When the interrupt happens, *do_IRQ* clears that bit and then returns, because no handler is registered. *probe_irq_off*, when called by a driver, needs to search for only the IRQ that no longer has IRQ_WAITING set.

Implementing a Handler

So far, we've learned to register an interrupt handler but not to write one. Actually, there's nothing unusual about a handler—it's ordinary C code.

The only peculiarity is that a handler runs at interrupt time and, therefore, suffers some restrictions on what it can do. These restrictions are the same as those we saw with kernel timers. A handler can't transfer data to or from user space, because it doesn't execute in the context of a process. Handlers also cannot do anything that would sleep, such as calling *wait_event*, allocating memory with anything other than GFP_ATOMIC, or locking a semaphore. Finally, handlers cannot call *schedule*.

The role of an interrupt handler is to give feedback to its device about interrupt reception and to read or write data according to the meaning of the interrupt being serviced. The first step usually consists of clearing a bit on the interface board; most hardware devices won't generate other interrupts until their "interrupt-pending" bit has been cleared. Depending on how your hardware works, this step may need to be performed last instead of first; there is no catch-all rule here. Some devices don't require this step, because they don't have an "interrupt-pending" bit; such devices are a minority, although the parallel port is one of them. For that reason, *short* does not have to clear such a bit.

A typical task for an interrupt handler is awakening processes sleeping on the device if the interrupt signals the event they're waiting for, such as the arrival of new data.

To stick with the frame grabber example, a process could acquire a sequence of images by continuously reading the device; the *read* call blocks before reading each frame, while the interrupt handler awakens the process as soon as each new frame arrives. This assumes that the grabber interrupts the processor to signal successful arrival of each new frame.

The programmer should be careful to write a routine that executes in a minimum amount of time, independent of its being a fast or slow handler. If a long computation needs to be performed, the best approach is to use a tasklet or workqueue to schedule computation at a safer time (we'll look at how work can be deferred in this manner in the section "Top and Bottom Halves.")

Our sample code in *short* responds to the interrupt by calling *do_gettimeofday* and printing the current time into a page-sized circular buffer. It then awakens any reading process, because there is now data available to be read.

```
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);
    wake_up_interruptible(&short_queue); /* awake any reading process */
    return IRQ_HANDLED;
}
```

This code, though simple, represents the typical job of an interrupt handler. It, in turn, calls *short_incr_bp*, which is defined as follows:

```
static inline void short_incr_bp(volatile unsigned long *index, int delta)
{
    unsigned long new = *index + delta;
    barrier( );  /* Don't optimize these two together */
    *index = (new >= (short_buffer + PAGE_SIZE)) ? short_buffer : new;
}
```

This function has been carefully written to wrap a pointer into the circular buffer without ever exposing an incorrect value. The *barrier* call is there to block compiler optimizations across the other two lines of the function. Without the barrier, the compiler might decide to optimize out the `new` variable and assign directly to `*index`. That optimization could expose an incorrect value of the index for a brief period in the case where it wraps. By taking care to prevent in inconsistent value from ever being visible to other threads, we can manipulate the circular buffer pointers safely without locks.

The device file used to read the buffer being filled at interrupt time is */dev/shortint*. This device special file, together with */dev/shortprint*, wasn't introduced in

Chapter 9, because its use is specific to interrupt handling. The internals of */dev/shortint* are specifically tailored for interrupt generation and reporting. Writing to the device generates one interrupt every other byte; reading the device gives the time when each interrupt was reported.

If you connect together pins 9 and 10 of the parallel connector, you can generate interrupts by raising the high bit of the parallel data byte. This can be accomplished by writing binary data to */dev/short0* or by writing anything to */dev/shortint*.[*]

The following code implements *read* and *write* for */dev/shortint*:

```
ssize_t short_i_read (struct file *filp, char __user *buf, size_t count,
     loff_t *f_pos)
{
    int count0;
    DEFINE_WAIT(wait);

    while (short_head == short_tail) {
        prepare_to_wait(&short_queue, &wait, TASK_INTERRUPTIBLE);
        if (short_head == short_tail)
            schedule( );
        finish_wait(&short_queue, &wait);
        if (signal_pending (current))  /* a signal arrived */
            return -ERESTARTSYS; /* tell the fs layer to handle it */
    }
    /* count0 is the number of readable data bytes */
    count0 = short_head - short_tail;
    if (count0 < 0) /* wrapped */
        count0 = short_buffer + PAGE_SIZE - short_tail;
    if (count0 < count) count = count0;

    if (copy_to_user(buf, (char *)short_tail, count))
        return -EFAULT;
    short_incr_bp (&short_tail, count);
    return count;
}

ssize_t short_i_write (struct file *filp, const char __user *buf, size_t count,
        loff_t *f_pos)
{
    int written = 0, odd = *f_pos & 1;
    unsigned long port = short_base; /* output to the parallel data latch */
    void *address = (void *) short_base;

    if (use_mem) {
        while (written < count)
            iowrite8(0xff * ((++written + odd) & 1), address);
    } else {
```

[*] The *shortint* device accomplishes its task by alternately writing 0x00 and 0xff to the parallel port.

```
        while (written < count)
            outb(0xff * ((++written + odd) & 1), port);
    }

    *f_pos += count;
    return written;
}
```

The other device special file, */dev/shortprint*, uses the parallel port to drive a printer; you can use it if you want to avoid connecting pins 9 and 10 of a D-25 connector. The *write* implementation of *shortprint* uses a circular buffer to store data to be printed, while the *read* implementation is the one just shown (so you can read the time your printer takes to eat each character).

In order to support printer operation, the interrupt handler has been slightly modified from the one just shown, adding the ability to send the next data byte to the printer if there is more data to transfer.

Handler Arguments and Return Value

Though *short* ignores them, three arguments are passed to an interrupt handler: `irq`, `dev_id`, and `regs`. Let's look at the role of each.

The interrupt number (`int irq`) is useful as information you may print in your log messages, if any. The second argument, `void *dev_id`, is a sort of client data; a `void *` argument is passed to *request_irq*, and this same pointer is then passed back as an argument to the handler when the interrupt happens. You usually pass a pointer to your device data structure in `dev_id`, so a driver that manages several instances of the same device doesn't need any extra code in the interrupt handler to find out which device is in charge of the current interrupt event.

Typical use of the argument in an interrupt handler is as follows:

```
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 */
    /* .... */
}
```

The typical *open* code associated with this handler looks like this:

```
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;
}
```

The last argument, `struct pt_regs *regs`, is rarely used. It holds a snapshot of the processor's context before the processor entered interrupt code. The registers can be used for monitoring and debugging; they are not normally needed for regular device driver tasks.

Interrupt handlers should return a value indicating whether there was actually an interrupt to handle. If the handler found that its device did, indeed, need attention, it should return `IRQ_HANDLED`; otherwise the return value should be `IRQ_NONE`. You can also generate the return value with this macro:

```
IRQ_RETVAL(handled)
```

where `handled` is nonzero if you were able to handle the interrupt. The return value is used by the kernel to detect and suppress spurious interrupts. If your device gives you no way to tell whether it really interrupted, you should return `IRQ_HANDLED`.

Enabling and Disabling Interrupts

There are times when a device driver must block the delivery of interrupts for a (hopefully short) period of time (we saw one such situation in the section "Spinlocks" in Chapter 5). Often, interrupts must be blocked while holding a spinlock to avoid deadlocking the system. There are ways of disabling interrupts that do not involve spinlocks. But before we discuss them, note that disabling interrupts should be a relatively rare activity, even in device drivers, and this technique should never be used as a mutual exclusion mechanism within a driver.

Disabling a single interrupt

Sometimes (but rarely!) a driver needs to disable interrupt delivery for a specific interrupt line. The kernel offers three functions for this purpose, all declared in *<asm/irq.h>*. These functions are part of the kernel API, so we describe them, but their use is discouraged in most drivers. Among other things, you cannot disable shared interrupt lines, and, on modern systems, shared interrupts are the norm. That said, here they are:

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

Calling any of these functions may update the mask for the specified `irq` in the programmable interrupt controller (PIC), thus disabling or enabling the specified IRQ across all processors. Calls to these functions can be nested—if *disable_irq* is called twice in succession, two *enable_irq* calls are required before the IRQ is truly reenabled. It is possible to call these functions from an interrupt handler, but enabling your own IRQ while handling it is not usually good practice.

disable_irq not only disables the given interrupt but also waits for a currently executing interrupt handler, if any, to complete. Be aware that if the thread calling *disable_irq* holds any resources (such as spinlocks) that the interrupt handler needs, the system can deadlock. *disable_irq_nosync* differs from *disable_irq* in that it returns immediately. Thus, using *disable_irq_nosync* is a little faster but may leave your driver open to race conditions.

But why disable an interrupt? Sticking to the parallel port, let's look at the *plip* network interface. A *plip* device uses the bare-bones parallel port to transfer data. Since only five bits can be read from the parallel connector, they are interpreted as four data bits and a clock/handshake signal. When the first four bits of a packet are transmitted by the initiator (the interface sending the packet), the clock line is raised, causing the receiving interface to interrupt the processor. The *plip* handler is then invoked to deal with newly arrived data.

After the device has been alerted, the data transfer proceeds, using the handshake line to clock new data to the receiving interface (this might not be the best implementation, but it is necessary for compatibility with other packet drivers using the parallel port). Performance would be unbearable if the receiving interface had to handle two interrupts for every byte received. Therefore, the driver disables the interrupt during the reception of the packet; instead, a poll-and-delay loop is used to bring in the data.

Similarly, because the handshake line from the receiver to the transmitter is used to acknowledge data reception, the transmitting interface disables its IRQ line during packet transmission.

Disabling all interrupts

What if you need to disable *all* interrupts? In the 2.6 kernel, it is possible to turn off all interrupt handling on the current processor with either of the following two functions (which are defined in *<asm/system.h>*):

```
void local_irq_save(unsigned long flags);
void local_irq_disable(void);
```

A call to *local_irq_save* disables interrupt delivery on the current processor after saving the current interrupt state into `flags`. Note that `flags` is passed directly, not by pointer. *local_irq_disable* shuts off local interrupt delivery without saving the state; you should use this version only if you know that interrupts have not already been disabled elsewhere.

Turning interrupts back on is accomplished with:

```
void local_irq_restore(unsigned long flags);
void local_irq_enable(void);
```

The first version restores that state which was stored into `flags` by *local_irq_save*, while *local_irq_enable* enables interrupts unconditionally. Unlike *disable_irq*, *local_irq_disable* does not keep track of multiple calls. If more than one function in the call chain might need to disable interrupts, *local_irq_save* should be used.

In the 2.6 kernel, there is no way to disable all interrupts globally across the entire system. The kernel developers have decided that the cost of shutting off all interrupts is too high and that there is no need for that capability in any case. If you are working with an older driver that makes calls to functions such as *cli* and *sti*, you need to update it to use proper locking before it will work under 2.6.

Top and Bottom Halves

One of the main problems with interrupt handling is how to perform lengthy tasks within a handler. Often a substantial amount of work must be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long. These two needs (work and speed) conflict with each other, leaving the driver writer in a bit of a bind.

Linux (along with many other systems) resolves this problem by splitting the interrupt handler into two halves. The so-called *top half* is the routine that actually responds to the interrupt—the one you register with *request_irq*. The *bottom half* is a routine that is scheduled by the top half to be executed later, at a safer time. The big difference between the top-half handler and the bottom half is that all interrupts are enabled during execution of the bottom half—that's why it runs at a safer time. In the typical scenario, the top half saves device data to a device-specific buffer, schedules its bottom half, and exits: this operation is very fast. The bottom half then performs whatever other work is required, such as awakening processes, starting up another I/O operation, and so on. This setup permits the top half to service a new interrupt while the bottom half is still working.

Almost every serious interrupt handler is split this way. For instance, when a network interface reports the arrival of a new packet, the handler just retrieves the data and pushes it up to the protocol layer; actual processing of the packet is performed in a bottom half.

The Linux kernel has two different mechanisms that may be used to implement bottom-half processing, both of which were introduced in Chapter 7. Tasklets are often the preferred mechanism for bottom-half processing; they are very fast, but all tasklet code must be atomic. The alternative to tasklets is workqueues, which may have a higher latency but that are allowed to sleep.

The following discussion works, once again, with the *short* driver. When loaded with a module option, *short* can be told to do interrupt processing in a top/bottom-half mode with either a tasklet or workqueue handler. In this case, the top half executes quickly; it simply remembers the current time and schedules the bottom half processing. The bottom half is then charged with encoding this time and awakening any user processes that may be waiting for data.

Tasklets

Remember that tasklets are a special function that may be scheduled to run, in software interrupt context, at a system-determined safe time. They may be scheduled to run multiple times, but tasklet scheduling is not cumulative; the tasklet runs only once, even if it is requested repeatedly before it is launched. No tasklet ever runs in parallel with itself, since they run only once, but tasklets can run in parallel with other tasklets on SMP systems. Thus, if your driver has multiple tasklets, they must employ some sort of locking to avoid conflicting with each other.

Tasklets are also guaranteed to run on the same CPU as the function that first schedules them. Therefore, an interrupt handler can be secure that a tasklet does not begin executing before the handler has completed. However, another interrupt can certainly be delivered while the tasklet is running, so locking between the tasklet and the interrupt handler may still be required.

Tasklets must be declared with the `DECLARE_TASKLET` macro:

```
DECLARE_TASKLET(name, function, data);
```

`name` is the name to be given to the tasklet, *function* is the function that is called to execute the tasklet (it takes one `unsigned long` argument and returns `void`), and `data` is an unsigned long value to be passed to the *tasklet* function.

The *short* driver declares its tasklet as follows:

```
void short_do_tasklet(unsigned long);
DECLARE_TASKLET(short_tasklet, short_do_tasklet, 0);
```

The function *tasklet_schedule* is used to schedule a tasklet for running. If *short* is loaded with `tasklet=1`, it installs a different interrupt handler that saves data and schedules the tasklet as follows:

```
irqreturn_t short_tl_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    do_gettimeofday((struct timeval *) tv_head); /* cast to stop 'volatile' warning
*/
    short_incr_tv(&tv_head);
    tasklet_schedule(&short_tasklet);
    short_wq_count++; /* record that an interrupt arrived */
    return IRQ_HANDLED;
}
```

The actual tasklet routine, *short_do_tasklet*, will be executed shortly (so to speak) at the system's convenience. As mentioned earlier, this routine performs the bulk of the work of handling the interrupt; it looks like this:

```
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 */
    /*
     * The bottom half reads the tv array, filled by the top half,
         * and prints it to the circular text buffer, which is then consumed
         * by reading processes
         */

        /* First 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);

        /*
         * Then, 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 */
    }
```

Among other things, this tasklet makes a note of how many interrupts have arrived since it was last called. A device such as *short* can generate a great many interrupts in a brief period, so it is not uncommon for several to arrive before the bottom half is executed. Drivers must always be prepared for this possibility and must be able to determine how much work there is to perform from the information left by the top half.

Workqueues

Recall that workqueues invoke a function at some future time in the context of a special worker process. Since the *workqueue* function runs in process context, it can sleep if need be. You cannot, however, copy data into user space from a workqueue, unless you use the advanced techniques we demonstrate in Chapter 15; the worker process does not have access to any other process's address space.

The *short* driver, if loaded with the `wq` option set to a nonzero value, uses a workqueue for its bottom-half processing. It uses the system default workqueue, so there is no special setup code required; if your driver has special latency requirements (or might sleep for a long time in the *workqueue* function), you may want to create your own, dedicated workqueue. We do need a `work_struct` structure, which is declared and initialized with the following:

```
    static struct work_struct short_wq;

        /* this line is in short_init( ) */
        INIT_WORK(&short_wq, (void (*)(void *)) short_do_tasklet, NULL);
```

Our worker function is *short_do_tasklet*, which we have already seen in the previous section.

When working with a workqueue, *short* establishes yet another interrupt handler that looks like this:

```
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;
}
```

As you can see, the interrupt handler looks very much like the tasklet version, with the exception that it calls *schedule_work* to arrange the bottom-half processing.

Interrupt Sharing

The notion of an IRQ conflict is almost synonymous with the PC architecture. In the past, IRQ lines on the PC have not been able to serve more than one device, and there have never been enough of them. As a result, frustrated users have often spent much time with their computer case open, trying to find a way to make all of their peripherals play well together.

Modern hardware, of course, has been designed to allow the sharing of interrupts; the PCI bus requires it. Therefore, the Linux kernel supports interrupt sharing on all buses, even those (such as the ISA bus) where sharing has traditionally not been supported. Device drivers for the 2.6 kernel should be written to work with shared interrupts if the target hardware can support that mode of operation. Fortunately, working with shared interrupts is easy, most of the time.

Installing a Shared Handler

Shared interrupts are installed through *request_irq* just like nonshared ones, but there are two differences:

- The `SA_SHIRQ` bit must be specified in the `flags` argument when requesting the interrupt.
- The `dev_id` argument *must* be unique. Any pointer into the module's address space will do, but `dev_id` definitely cannot be set to `NULL`.

The kernel keeps a list of shared handlers associated with the interrupt, and `dev_id` can be thought of as the signature that differentiates between them. If two drivers were to register `NULL` as their signature on the same interrupt, things might get mixed up at unload time, causing the kernel to oops when an interrupt arrived. For this reason, modern kernels complain loudly if passed a `NULL` `dev_id` when registering shared interrupts. When a shared interrupt is requested, *request_irq* succeeds if one of the following is true:

- The interrupt line is free.
- All handlers already registered for that line have also specified that the IRQ is to be shared.

Whenever two or more drivers are sharing an interrupt line and the hardware interrupts the processor on that line, the kernel invokes every handler registered for that interrupt, passing each its own `dev_id`. Therefore, a shared handler must be able to recognize its own interrupts and should quickly exit when its own device has not interrupted. Be sure to return `IRQ_NONE` whenever your handler is called and finds that the device is not interrupting.

If you need to probe for your device before requesting the IRQ line, the kernel can't help you. No probing function is available for shared handlers. The standard probing mechanism works if the line being used is free, but if the line is already held by another driver with sharing capabilities, the probe fails, even if your driver would have worked perfectly. Fortunately, most hardware designed for interrupt sharing is also able to tell the processor which interrupt it is using, thus eliminating the need for explicit probing.

Releasing the handler is performed in the normal way, using *free_irq*. Here the `dev_id` argument is used to select the correct handler to release from the list of shared handlers for the interrupt. That's why the `dev_id` pointer must be unique.

A driver using a shared handler needs to be careful about one more thing: it can't play with *enable_irq* or *disable_irq*. If it does, things might go haywire for other devices sharing the line; disabling another device's interrupts for even a short time may create latencies that are problematic for that device and it's user. Generally, the programmer must remember that his driver doesn't own the IRQ, and its behavior should be more "social" than is necessary if one owns the interrupt line.

Running the Handler

As suggested earlier, when the kernel receives an interrupt, all the registered handlers are invoked. A shared handler must be able to distinguish between interrupts that it needs to handle and interrupts generated by other devices.

Loading *short* with the option *shared=1* installs the following handler instead of the default:

```
irqreturn_t short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int value, written;
    struct timeval tv;

    /* If it wasn't short, return immediately */
    value = inb(short_base);
    if (!(value & 0x80))
        return IRQ_NONE;

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

    /* the rest is unchanged */

    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;
}
```

An explanation is due here. Since the parallel port has no “interrupt-pending” bit to check, the handler uses the ACK bit for this purpose. If the bit is high, the interrupt being reported is for *short*, and the handler clears the bit.

The handler resets the bit by zeroing the high bit of the parallel interface’s data port—*short* assumes that pins 9 and 10 are connected together. If one of the other devices sharing the IRQ with *short* generates an interrupt, *short* sees that its own line is still inactive and does nothing.

A full-featured driver probably splits the work into top and bottom halves, of course, but that’s easy to add and does not have any impact on the code that implements sharing. A real driver would also likely use the `dev_id` argument to determine which, of possibly many, devices might be interrupting.

Note that if you are using a printer (instead of the jumper wire) to test interrupt management with *short*, this shared handler won’t work as advertised, because the printer protocol doesn’t allow for sharing, and the driver can’t know whether the interrupt was from the printer.

The /proc Interface and Shared Interrupts

Installing shared handlers in the system doesn’t affect */proc/stat*, which doesn’t even know about handlers. However, */proc/interrupts* changes slightly.

All the handlers installed for the same interrupt number appear on the same line of */proc/interrupts*. The following output (from an x86_64 system) shows how shared interrupt handlers are displayed:

```
           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
NMI:      41234
LOC:  892193503
ERR:        102
MIS:          0
```

This system has several shared interrupt lines. IRQ 5 is used for the serial ATA and IEEE 1394 controllers; IRQ 10 has several devices, including an IDE controller, two USB controllers, an Ethernet interface, and a sound card; and IRQ 11 also is used by two USB controllers.

Interrupt-Driven I/O

Whenever a data transfer to or from the managed hardware might be delayed for any reason, the driver writer should implement buffering. Data buffers help to detach data transmission and reception from the *write* and *read* system calls, and overall system performance benefits.

A good buffering mechanism leads to *interrupt-driven I/O*, in which an input buffer is filled at interrupt time and is emptied by processes that read the device; an output buffer is filled by processes that write to the device and is emptied at interrupt time. An example of interrupt-driven output is the implementation of */dev/shortprint*.

For interrupt-driven data transfer to happen successfully, the hardware should be able to generate interrupts with the following semantics:

- For input, the device interrupts the processor when new data has arrived and is ready to be retrieved by the system processor. The actual actions to perform depend on whether the device uses I/O ports, memory mapping, or DMA.
- For output, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful data transfer. Memory-mapped and DMA-capable devices usually generate interrupts to tell the system they are done with the buffer.

The timing relationships between a *read* or *write* and the actual arrival of data were introduced in the section "Blocking and Nonblocking Operations" in Chapter 6.

A Write-Buffering Example

We have mentioned the *shortprint* driver a couple of times; now it is time to actually take a look. This module implements a very simple, output-oriented driver for the parallel port; it is sufficient, however, to enable the printing of files. If you chose to test this driver out, however, remember that you must pass the printer a file in a format it understands; not all printers respond well when given a stream of arbitrary data.

The *shortprint* driver maintains a one-page circular output buffer. When a user-space process writes data to the device, that data is fed into the buffer, but the *write* method does not actually perform any I/O. Instead, the core of *shortp_write* looks like this:

```
        while (written < count) {
            /* Hang out until some buffer space is available. */
            space = shortp_out_space( );
            if (space <= 0) {
                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)) {
                up(&shortp_out_sem);
                return -EFAULT;
            }
            shortp_incr_out_bp(&shortp_out_head, space);
            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( );
            spin_unlock_irqrestore(&shortp_out_lock, flags);
        }

    out:
        *f_pos += written;
```

A semaphore (`shortp_out_sem`) controls access to the circular buffer; *shortp_write* obtains that semaphore just prior to the code fragment above. While holding the semaphore, it attempts to feed data into the circular buffer. The function *shortp_out_space* returns the amount of contiguous space available (so there is no need to worry about buffer wraps); if that amount is 0, the driver waits until some space is freed. It then copies as much data as it can into the buffer.

Once there is data to output, *shortp_write* must ensure that the data is written to the device. The actual writing is done by way of a *workqueue* function; *shortp_write* must kick that function off if it is not already running. After obtaining a separate spinlock that controls access to variables used on the consumer side of the output buffer (including shortp_output_active), it calls *shortp_start_output* if need be. Then it's just a matter of noting how much data was "written" to the buffer and returning.

The function that starts the output process looks like the following:

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

    /* Set up our 'missed interrupt' timer */
    shortp_output_active = 1;
    shortp_timer.expires = jiffies + TIMEOUT;
    add_timer(&shortp_timer);

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

The reality of dealing with hardware is that you can, occasionally, lose an interrupt from the device. When this happens, you really do not want your driver to stop forevermore until the system is rebooted; that is not a user-friendly way of doing things. It is far better to realize that an interrupt has been missed, pick up the pieces, and go on. To that end, *shortprint* sets a kernel timer whenever it outputs data to the device. If the timer expires, we may have missed an interrupt. We look at the timer function shortly, but, for the moment, let's stick with the main output functionality. That is implemented in our *workqueue* function, which, as you can see above, is scheduled here. The core of that function looks like the following:

```
    spin_lock_irqsave(&shortp_out_lock, flags);

    /* Have we written everything? */
    if (shortp_out_head == shortp_out_tail) { /* empty */
        shortp_output_active = 0;
        wake_up_interruptible(&shortp_empty_queue);
        del_timer(&shortp_timer);
    }
    /* Nope, write another byte */
    else
        shortp_do_write( );

    /* If somebody's waiting, maybe wake them up. */
    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);
```

Since we are dealing with the output side's shared variables, we must obtain the spinlock. Then we look to see whether there is any more data to send out; if not, we note that output is no longer active, delete the timer, and wake up anybody who might have been waiting for the queue to become completely empty (this sort of wait is done when the device is closed). If, instead, there remains data to write, we call *shortp_do_write* to actually send a byte to the hardware.

Then, since we may have freed space in the output buffer, we consider waking up any processes waiting to add more data to that buffer. We do not perform that wakeup unconditionally, however; instead, we wait until a minimum amount of space is available. There is no point in awakening a writer every time we take one byte out of the buffer; the cost of awakening the process, scheduling it to run, and putting it back to sleep is too high for that. Instead, we should wait until that process is able to move a substantial amount of data into the buffer at once. This technique is common in buffering, interrupt-driven drivers.

For completeness, here is the code that actually writes the data to the port:

```
static void shortp_do_write(void)
{
    unsigned char cr = inb(shortp_base + SP_CONTROL);

    /* Something happened; reset the timer */
    mod_timer(&shortp_timer, jiffies + TIMEOUT);

    /* Strobe a byte out to the device */
    outb_p(*shortp_out_tail, shortp_base+SP_DATA);
    shortp_incr_out_bp(&shortp_out_tail, 1);
    if (shortp_delay)
        udelay(shortp_delay);
    outb_p(cr | SP_CR_STROBE, shortp_base+SP_CONTROL);
    if (shortp_delay)
        udelay(shortp_delay);
    outb_p(cr & ~SP_CR_STROBE, shortp_base+SP_CONTROL);
}
```

Here, we reset the timer to reflect the fact that we have made some progress, strobe the byte out to the device, and update the circular buffer pointer.

The *workqueue* function does not resubmit itself directly, so only a single byte will be written to the device. At some point, the printer will, in its slow way, consume the byte and become ready for the next one; it will then interrupt the processor. The interrupt handler used in *shortprint* is short and simple:

```
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);
        return IRQ_HANDLED;
    }
```

Since the parallel port does not require an explicit interrupt acknowledgment, all the interrupt handler really needs to do is to tell the kernel to run the *workqueue* function again.

What if the interrupt never comes? The driver code that we have seen thus far would simply come to a halt. To keep that from happening, we set a timer back a few pages ago. The function that is executed when that timer expires is:

```
    static void shortp_timeout(unsigned long unused)
    {
        unsigned long flags;
        unsigned char status;

        if (! shortp_output_active)
            return;
        spin_lock_irqsave(&shortp_out_lock, flags);
        status = inb(shortp_base + SP_STATUS);

        /* If the printer is still busy we just reset the timer */
        if ((status & SP_SR_BUSY) == 0 || (status & SP_SR_ACK)) {
            shortp_timer.expires = jiffies + TIMEOUT;
            add_timer(&shortp_timer);
            spin_unlock_irqrestore(&shortp_out_lock, flags);
            return;
        }

        /* Otherwise we must have dropped an interrupt. */
        spin_unlock_irqrestore(&shortp_out_lock, flags);
        shortp_interrupt(shortp_irq, NULL, NULL);
    }
```

If no output is supposed to be active, the timer function simply returns; this keeps the timer from resubmitting itself when things are being shut down. Then, after taking the lock, we query the status of the port; if it claims to be busy, it simply hasn't gotten around to interrupting us yet, so we reset the timer and return. Printers can, at times, take a very long time to make themselves ready; consider the printer that runs out of paper while everybody is gone over a long weekend. In such situations, there is nothing to do other than to wait patiently until something changes.

If, however, the printer claims to be ready, we must have missed its interrupt. In that case, we simply invoke our interrupt handler manually to get the output process moving again.

The *shortprint* driver does not support reading from the port; instead, it behaves like *shortint* and returns interrupt timing information. The implementation of an interrupt-driven *read* method would be very similar to what we have seen, however. Data from the device would be read into a driver buffer; it would be copied out to user space only when a significant amount of data has accumulated in the buffer, the full *read* request has been satisfied, or some sort of timeout occurs.

Quick Reference

These symbols related to interrupt management were introduced in this chapter:

`#include <linux/interrupt.h>`
`int request_irq(unsigned int irq, irqreturn_t (*handler)( ), unsigned long flags, const char *dev_name, void *dev_id);`
`void free_irq(unsigned int irq, void *dev_id);`
: Calls that register and unregister an interrupt handler.

`#include <linux/irq.h.h>`
`int can_request_irq(unsigned int irq, unsigned long flags);`
: This function, available on the i386 and x86_64 architectures, returns a nonzero value if an attempt to allocate the given interrupt line succeeds.

`#include <asm/signal.h>`
`SA_INTERRUPT`
`SA_SHIRQ`
`SA_SAMPLE_RANDOM`
: Flags for *request_irq*. `SA_INTERRUPT` requests installation of a fast handler (as opposed to a slow one). `SA_SHIRQ` installs a shared handler, and the third flag asserts that interrupt timestamps can be used to generate system entropy.

`/proc/interrupts`
`/proc/stat`
: Filesystem nodes that report information about hardware interrupts and installed handlers.

`unsigned long probe_irq_on(void);`
`int probe_irq_off(unsigned long);`
: Functions used by the driver when it has to probe to determine which interrupt line is being used by a device. The result of *probe_irq_on* must be passed back to *probe_irq_off* after the interrupt has been generated. The return value of *probe_irq_off* is the detected interrupt number.

`IRQ_NONE`
`IRQ_HANDLED`
`IRQ_RETVAL(int x)`
: The possible return values from an interrupt handler, indicating whether an actual interrupt from the device was present.

`void disable_irq(int irq);`
`void disable_irq_nosync(int irq);`
`void enable_irq(int irq);`
A driver can enable and disable interrupt reporting. If the hardware tries to generate an interrupt while interrupts are disabled, the interrupt is lost forever. A driver using a shared handler must not use these functions.

`void local_irq_save(unsigned long flags);`
`void local_irq_restore(unsigned long flags);`
Use *local_irq_save* to disable interrupts on the local processor and remember their previous state. The `flags` can be passed to *local_irq_restore* to restore the previous interrupt state.

`void local_irq_disable(void);`
`void local_irq_enable(void);`
Functions that unconditionally disable and enable interrupts on the current processor.