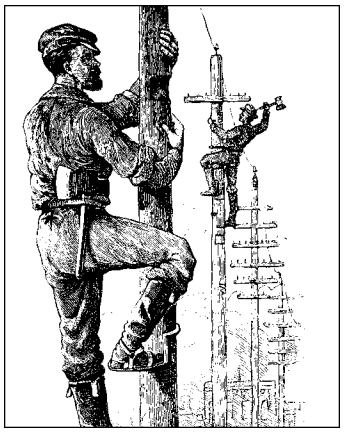

CHAPTER EIGHT

HARDWARE MANAGEMENT

Although playing with *scull* and similar toys is a good introduction to the software interface of a Linux device driver, implementing a *real* device requires hardware. The driver is the abstraction layer between software concepts and hardware circuitry; as such, it needs to talk with both of them. Up to now, we have examined the internals of software concepts; this chapter completes the picture by showing you how a driver can access I/O ports and I/O memory while being portable across Linux platforms.

This chapter continues in the tradition of staying as independent of specific hardware as possible. However, where specific examples are needed, we use simple digital I/O ports (like the standard PC parallel port) to show how the I/O instructions work, and normal frame-buffer video memory to show memory-mapped I/O.

We chose simple digital I/O because it is the easiest form of input/output port. Also, the Centronics parallel port implements raw I/O and is available in most computers: data bits written to the device appear on the output pins, and voltage levels on the input pins are directly accessible by the processor. In practice, you have to connect LEDs to the port to actually *see* the results of a digital I/O operation, but the underlying hardware is extremely easy to use.

I/O Ports and I/O Memory

Every peripheral device is controlled by writing and reading its registers. Most of the time a device has several registers, and they are accessed at consecutive addresses, either in the memory address space or in the I/O address space.

At the hardware level, there is no conceptual difference between memory regions and I/O regions: both of them are accessed by asserting electrical signals on the

address bus and control bus (i.e., the *read* and *write* signals)* and by reading from or writing to the data bus.

While some CPU manufacturers implement a single address space in their chips, some others decided that peripheral devices are different from memory and therefore deserve a separate address space. Some processors (most notably the x86 family) have separate *read* and *write* electrical lines for I/O ports, and special CPU instructions to access ports.

Because peripheral devices are built to fit a peripheral bus, and the most popular I/O buses are modeled on the personal computer, even processors that do not have a separate address space for I/O ports must fake reading and writing I/O ports when accessing some peripheral devices, usually by means of external chipsets or extra circuitry in the CPU core. The latter solution is only common within tiny processors meant for embedded use.

For the same reason, Linux implements the concept of I/O ports on all computer platforms it runs on, even on platforms where the CPU implements a single address space. The implementation of port access sometimes depends on the specific make and model of the host computer (because different models use different chipsets to map bus transactions into memory address space).

Even if the peripheral bus has a separate address space for I/O ports, not all devices map their registers to I/O ports. While use of I/O ports is common for ISA peripheral boards, most PCI devices map registers into a memory address region. This I/O memory approach is generally preferred because it doesn't require use of special-purpose processor instructions; CPU cores access memory much more efficiently, and the compiler has much more freedom in register allocation and addressing-mode selection when accessing memory.

I/O Registers and Conventional Memory

Despite the strong similarity between hardware registers and memory, a programmer accessing I/O registers must be careful to avoid being tricked by CPU (or compiler) optimizations that can modify the expected I/O behavior.

The main difference between I/O registers and RAM is that I/O operations have side effects, while memory operations have none: the only effect of a memory write is storing a value to a location, and a memory read returns the last value written there. Because memory access speed is so critical to CPU performance, the no-side-effects case has been optimized in several ways: values are cached and read/write instructions are reordered.

---

* Not all computer platform use a *read* and a *write* signal; some have different means to address external circuits. The difference is irrelevant at software level, however, and we'll assume all have *read* and *write* to simplify the discussion.

The compiler can cache data values into CPU registers without writing them to memory, and even if it stores them, both write and read operations can operate on cache memory without ever reaching physical RAM. Reordering can also happen both at compiler level and at hardware level: often a sequence of instructions can be executed more quickly if it is run in an order different from that which appears in the program text, for example, to prevent interlocks in the RISC pipeline. On CISC processors, operations that take a significant amount of time can be executed concurrently with other, quicker ones.

These optimizations are transparent and benign when applied to conventional memory (at least on uniprocessor systems), but they can be fatal to correct I/O operations because they interfere with those "side effects" that are the main reason why a driver accesses I/O registers. The processor cannot anticipate a situation in which some other process (running on a separate processor, or something happening inside an I/O controller) depends on the order of memory access. A driver must therefore ensure that no caching is performed and no read or write reordering takes place when accessing registers: the compiler or the CPU may just try to outsmart you and reorder the operations you request; the result can be strange errors that are very difficult to debug.

The problem with hardware caching is the easiest to face: the underlying hardware is already configured (either automatically or by Linux initialization code) to disable any hardware cache when accessing I/O regions (whether they are memory or port regions).

The solution to compiler optimization and hardware reordering is to place a *memory barrier* between operations that must be visible to the hardware (or to another processor) in a particular order. Linux provides four macros to cover all possible ordering needs.

```
#include <linux/kernel.h>
void barrier(void)
```

This function tells the compiler to insert a memory barrier, but has no effect on the hardware. Compiled code will store to memory all values that are currently modified and resident in CPU registers, and will reread them later when they are needed.

```
#include <asm/system.h>
void rmb(void);
void wmb(void);
void mb(void);
```

These functions insert hardware memory barriers in the compiled instruction flow; their actual instantiation is platform dependent. An *rmb* (read memory barrier) guarantees that any reads appearing before the barrier are completed prior to the execution of any subsequent read. *wmb* guarantees ordering in write operations, and the *mb* instruction guarantees both. Each of these functions is a superset of *barrier*.

A typical usage of memory barriers in a device driver may have this sort of form:

```
writel(dev->registers.addr, io_destination_address);
writel(dev->registers.size, io_size);
writel(dev->registers.operation, DEV_READ);
wmb();
writel(dev->registers.control, DEV_GO);
```

In this case, it is important to be sure that all of the device registers controlling a particular operation have been properly set prior to telling it to begin. The memory barrier will enforce the completion of the writes in the necessary order.

Because memory barriers affect performance, they should only be used where really needed. The different types of barriers can also have different performance characteristics, so it is worthwhile to use the most specific type possible. For example, on the x86 architecture, *wmb()* currently does nothing, since writes outside the processor are not reordered. Reads are reordered, however, so *mb()* will be slower than *wmb()*.

It is worth noting that most of the other kernel primitives dealing with synchronization, such as spinlock and `atomic_t` operations, also function as memory barriers.

Some architectures allow the efficient combination of an assignment and a memory barrier. Version 2.4 of the kernel provides a few macros that perform this combination; in the default case they are defined as follows:

```
#define set_mb(var, value)  do {var = value; mb();}  while 0
#define set_wmb(var, value) do {var = value; wmb();} while 0
#define set_rmb(var, value) do {var = value; rmb();} while 0
```

Where appropriate, `<asm/system.h>` defines these macros to use architecture-specific instructions that accomplish the task more quickly.

The header file *sysdep.h* defines macros described in this section for the platforms and the kernel versions that lack them.

Using I/O Ports

I/O ports are the means by which drivers communicate with many devices out there—at least part of the time. This section covers the various functions available for making use of I/O ports; we also touch on some portability issues.

Let us start with a quick reminder that I/O ports must be allocated before being used by your driver. As we discussed in "I/O Ports and I/O Memory" in Chapter 2, the functions used to allocate and free ports are:

```
#include <linux/ioport.h>
int check_region(unsigned long start, unsigned long len);
struct resource *request_region(unsigned long start,
       unsigned long len, char *name);
void release_region(unsigned long start, unsigned long len);
```

After a driver has requested the range of I/O ports it needs to use in its activities, it must read and/or write to those ports. To this aim, most hardware differentiates between 8-bit, 16-bit, and 32-bit ports. Usually you can't mix them like you normally do with system memory access.*

A C program, therefore, must call different functions to access different size ports. As suggested in the previous section, computer architectures that support only memory-mapped I/O registers fake port I/O by remapping port addresses to memory addresses, and the kernel hides the details from the driver in order to ease portability. The Linux kernel headers (specifically, the architecture-dependent header `<asm/io.h>`) define the following inline functions to access I/O ports.

From now on, when we use `unsigned` without further type specifications, we are referring to an architecture-dependent definition whose exact nature is not relevant. The functions are almost always portable because the compiler automatically casts the values during assignment—their being unsigned helps prevent compile-time warnings. No information is lost with such casts as long as the programmer assigns sensible values to avoid overflow. We'll stick to this convention of "incomplete typing" for the rest of the chapter.

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

Read or write byte ports (eight bits wide). The `port` argument is defined as `unsigned long` for some platforms and `unsigned short` for others. The return type of *inb* is also different across architectures.

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

These functions access 16-bit ports (word wide); they are not available when compiling for the M68k and S390 platforms, which support only byte I/O.

* Sometimes I/O ports are arranged like memory, and you can (for example) bind two 8-bit writes into a single 16-bit operation. This applies, for instance, to PC video boards, but in general you can't count on this feature.

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

These functions access 32-bit ports. `longword` is either declared as `unsigned long` or `unsigned int`, according to the platform. Like word I/O, "long" I/O is not available on M68k and S390.

Note that no 64-bit port I/O operations are defined. Even on 64-bit architectures, the port address space uses a 32-bit (maximum) data path.

The functions just described are primarily meant to be used by device drivers, but they can also be used from user space, at least on PC-class computers. The GNU C library defines them in `<sys/io.h>`. The following conditions should apply in order for *inb* and friends to be used in user-space code:

- The program must be compiled with the *-O* option to force expansion of inline functions.
- The *ioperm* or *iopl* system calls must be used to get permission to perform I/O operations on ports. *ioperm* gets permission for individual ports, while *iopl* gets permission for the entire I/O space. Both these functions are Intel specific.
- The program must run as root to invoke *ioperm* or *iopl** Alternatively, one of its ancestors must have gained port access running as root.

If the host platform has no *ioperm* and no *iopl* system calls, user space can still access I/O ports by using the */dev/port* device file. Note, though, that the meaning of the file is very platform specific, and most likely not useful for anything but the PC.

The sample sources *misc-progs/inp.c* and *misc-progs/outp.c* are a minimal tool for reading and writing ports from the command line, in user space. They expect to be installed under multiple names (i.e., *inpb*, *inpw*, and *inpl* and will manipulate byte, word, or long ports depending on which name was invoked by the user. They use */dev/port* if *ioperm* is not present.

The programs can be made setuid root, if you want to live dangerously and play with your hardware without acquiring explicit privileges.

String Operations

In addition to the single-shot in and out operations, some processors implement special instructions to transfer a sequence of bytes, words, or longs to and from a single I/O port or the same size. These are the so-called *string instructions*, and they perform the task more quickly than a C-language loop can do. The following

---

* Technically, it must have the `CAP_SYS_RAWIO` capability, but that is the same as running as root on current systems.

macros implement the concept of string I/O by either using a single machine instruction or by executing a tight loop if the target processor has no instruction that performs string I/O. The macros are not defined at all when compiling for the M68k and S390 platforms. This should not be a portability problem, since these platforms don't usually share device drivers with other platforms, because their peripheral buses are different.

The prototypes for string functions are the following:

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```
Read or write `count` bytes starting at the memory address `addr`. Data is read from or written to the single port `port`.

```
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```
Read or write 16-bit values to a single 16-bit port.

```
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```
Read or write 32-bit values to a single 32-bit port.

Pausing I/O

Some platforms—most notably the i386—can have problems when the processor tries to transfer data too quickly to or from the bus. The problems can arise because the processor is overclocked with respect to the ISA bus, and can show up when the device board is too slow. The solution is to insert a small delay after each I/O instruction if another such instruction follows. If your device misses some data, or if you fear it might miss some, you can use pausing functions in place of the normal ones. The pausing functions are exactly like those listed previously, but their names end in `_p`; they are called *inb_p*, *outb_p*, and so on. The functions are defined for most supported architectures, although they often expand to the same code as nonpausing I/O, because there is no need for the extra pause if the architecture runs with a nonobsolete peripheral bus.

Platform Dependencies

I/O instructions are, by their nature, highly processor dependent. Because they work with the details of how the processor handles moving data in and out, it is very hard to hide the differences between systems. As a consequence, much of the source code related to port I/O is platform dependent.

You can see one of the incompatibilities, data typing, by looking back at the list of functions, where the arguments are typed differently based on the architectural

differences between platforms. For example, a port is `unsigned short` on the x86 (where the processor supports a 64-KB I/O space), but `unsigned long` on other platforms, whose ports are just special locations in the same address space as memory.

Other platform dependencies arise from basic structural differences in the processors and thus are unavoidable. We won't go into detail about the differences, because we assume that you won't be writing a device driver for a particular system without understanding the underlying hardware. Instead, the following is an overview of the capabilities of the architectures that are supported by version 2.4 of the kernel:

IA-32 (x86)
: The architecture supports all the functions described in this chapter. Port numbers are of type `unsigned short`.

IA-64 (Itanium)
: All functions are supported; ports are `unsigned long` (and memory-mapped). String functions are implemented in C.

Alpha
: All the functions are supported, and ports are memory-mapped. The implementation of port I/O is different in different Alpha platforms, according to the chipset they use. String functions are implemented in C and defined in *arch/alpha/lib/io.c*. Ports are `unsigned long`.

ARM
: Ports are memory-mapped, and all functions are supported; string functions are implemented in C. Ports are of type `unsigned int`.

M68k
: Ports are memory-mapped, and only byte functions are supported. No string functions are supported, and the port type is `unsigned char *`.

MIPS
MIPS64
: The MIPS port supports all the functions. String operations are implemented with tight assembly loops, because the processor lacks machine-level string I/O. Ports are memory-mapped; they are `unsigned int` in 32-bit processors and `unsigned long` in 64-bit ones.

PowerPC
: All the functions are supported; ports have type `unsigned char *`.

S390
: Similar to the M68k, the header for this platform supports only byte-wide port I/O with no string operations. Ports are `char` pointers and are memory-mapped.

Super-H
: Ports are `unsigned int` (memory-mapped), and all the functions are supported.

SPARC
SPARC64
: Once again, I/O space is memory-mapped. Versions of the port functions are defined to work with `unsigned long` ports.

The curious reader can extract more information from the *io.h* files, which sometimes define a few architecture-specific functions in addition to those we describe in this chapter. Be warned that some of these files are rather difficult reading, however.

It's interesting to note that no processor outside the x86 family features a different address space for ports, even though several of the supported families are shipped with ISA and/or PCI slots (and both buses implement different I/O and memory address spaces).

Moreover, some processors (most notably the early Alphas) lack instructions that move one or two bytes at a time.* Therefore, their peripheral chipsets simulate 8-bit and 16-bit I/O accesses by mapping them to special address ranges in the memory address space. Thus, an *inb* and an *inw* instruction that act on the same port are implemented by two 32-bit memory reads that operate on different addresses. Fortunately, all of this is hidden from the device driver writer by the internals of the macros described in this section, but we feel it's an interesting feature to note. If you want to probe further, look for examples in *include/asm-alpha/core_lca.h*.

How I/O operations are performed on each platform is well described in the programmer's manual for each platform; those manuals are usually available for download as PDF files on the Web.

* Single-byte I/O is not as important as one may imagine, because it is a rare operation. In order to read/write a single byte to any address space, you need to implement a data path connecting the low bits of the register-set data bus to any byte position in the external data bus. These data paths require additional logic gates that get in the way of every data transfer. Dropping byte-wide loads and stores can benefit overall system performance.

Using Digital I/O Ports

The sample code we use to show port I/O from within a device driver acts on general-purpose digital I/O ports; such ports are found in most computer systems.

A digital I/O port, in its most common incarnation, is a byte-wide I/O location, either memory-mapped or port-mapped. When you write a value to an output location, the electrical signal seen on output pins is changed according to the individual bits being written. When you read a value from the input location, the current logic level seen on input pins is returned as individual bit values.

The actual implementation and software interface of such I/O ports varies from system to system. Most of the time I/O pins are controlled by two I/O locations: one that allows selecting what pins are used as input and what pins are used as output, and one in which you can actually read or write logic levels. Sometimes, however, things are even simpler and the bits are hardwired as either input or output (but, in this case, you don't call them "general-purpose I/O" anymore); the parallel port found on all personal computers is one such not-so-general-purpose I/O port. Either way, the I/O pins are usable by the sample code we introduce shortly.

An Overview of the Parallel Port

Because we expect most readers to be using an x86 platform in the form called "personal computer," we feel it is worth explaining how the PC parallel port is designed. The parallel port is the peripheral interface of choice for running digital I/O sample code on a personal computer. Although most readers probably have parallel port specifications available, we summarize them here for your convenience.

The parallel interface, in its minimal configuration (we will overlook the ECP and EPP modes) is made up of three 8-bit ports. The PC standard starts the I/O ports for the first parallel interface at `0x378`, and for the second at `0x278`. The first port is a bidirectional data register; it connects directly to pins 2 through 9 on the physical connector. The second port is a read-only status register; when the parallel port is being used for a printer, this register reports several aspects of printer status, such as being online, out of paper, or busy. The third port is an output-only control register, which, among other things, controls whether interrupts are enabled.

The signal levels used in parallel communications are standard transistor-transistor logic (TTL) levels: 0 and 5 volts, with the logic threshold at about 1.2 volts; you can count on the ports at least meeting the standard TTL LS current ratings, although most modern parallel ports do better in both current and voltage ratings.

The parallel connector is not isolated from the computer's internal circuitry, which is useful if you want to connect logic gates directly to the port. But you have to be careful to do the wiring correctly; the parallel port circuitry is easily damaged when you play with your own custom circuitry unless you add optoisolators to your circuit. You can choose to use plug-in parallel ports if you fear you'll damage your motherboard.

The bit specifications are outlined in Figure 8-1. You can access 12 output bits and 5 input bits, some of which are logically inverted over the course of their signal path. The only bit with no associated signal pin is bit 4 (0x10) of port 2, which enables interrupts from the parallel port. We'll make use of this bit as part of our implementation of an interrupt handler in Chapter 9.

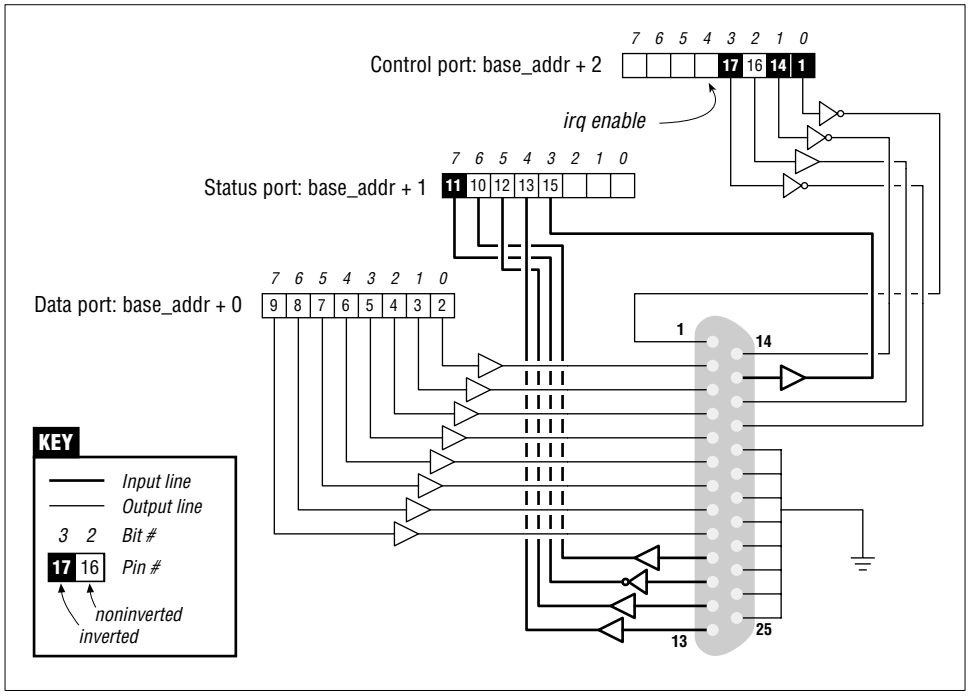

Figure 8-1. The pinout of the parallel port

A Sample Driver

The driver we will introduce is called *short* (Simple Hardware Operations and Raw Tests). All it does is read and write a few eight-bit ports, starting from the one you select at load time. By default it uses the port range assigned to the parallel interface of the PC. Each device node (with a unique minor number) accesses a different port. The *short* driver doesn't do anything useful; it just isolates for external use a single instruction acting on a port. If you are not used to port I/O, you can use *short* to get familiar with it; you can measure the time it takes to transfer data through a port or play other games.

For *short* to work on your system, it must have free access to the underlying hardware device (by default, the parallel interface); thus, no other driver may have allocated it. Most modern distributions set up the parallel port drivers as modules that are loaded only when needed, so contention for the I/O addresses is not usually a problem. If, however, you get a "can't get I/O address" error from *short* (on the console or in the system log file), some other driver has probably already taken the port. A quick look at */proc/ioports* will usually tell you which driver is getting in the way. The same caveat applies to other I/O devices if you are not using the parallel interface.

From now on, we'll just refer to "the parallel interface" to simplify the discussion. However, you can set the `base` module parameter at load time to redirect *short* to other I/O devices. This feature allows the sample code to run on any Linux platform where you have access to a digital I/O interface that is accessible via *outb* and *inb* (even though the actual hardware is memory-mapped on all platforms but the x86). Later, in "Using I/O Memory," we'll show how *short* can be used with generic memory-mapped digital I/O as well.

To watch what happens on the parallel connector, and if you have a bit of an inclination to work with hardware, you can solder a few LEDs to the output pins. Each LED should be connected in series to a 1-KΩ resistor leading to a ground pin (unless, of course, your LEDs have the resistor built in). If you connect an output pin to an input pin, you'll generate your own input to be read from the input ports.

Note that you cannot just connect a printer to the parallel port and see data sent to *short*. This driver implements simple access to the I/O ports and does not perform the handshake that printers need to operate on the data.

If you are going to view parallel data by soldering LEDs to a D-type connector, we suggest that you not use pins 9 and 10, because we'll be connecting them together later to run the sample code shown in Chapter 9.

As far as *short* is concerned, */dev/short0* writes to and reads from the eight-bit port located at the I/O address `base` (0x378 unless changed at load time). */dev/short1* writes to the eight-bit port located at `base + 1`, and so on up to `base + 7`.

The actual output operation performed by */dev/short0* is based on a tight loop using *outb*. A memory barrier instruction is used to ensure that the output operation actually takes place and is not optimized away.

```
while (count--) {
    outb(*(ptr++), address);
    wmb();
}
```

You can run the following command to light your LEDs:

```
echo  -n "any string"  > /dev/short0
```

Each LED monitors a single bit of the output port. Remember that only the last character written remains steady on the output pins long enough to be perceived by your eyes. For that reason, we suggest that you prevent automatic insertion of a trailing newline by passing the *-n* option to *echo*.

Reading is performed by a similar function, built around *inb* instead of *outb*. In order to read "meaningful" values from the parallel port, you need to have some hardware connected to the input pins of the connector to generate signals. If there is no signal, you'll read an endless stream of identical bytes. If you choose to read from an output port, you'll most likely get back the last value written to the port (this applies to the parallel interface and to most other digital I/O circuits in common use). Thus, those uninclined to get out their soldering irons can read the current output value on port 0x378 by running a command like:

```
dd if=/dev/short0 bs=1 count=1 | od -t x1
```

To demonstrate the use of all the I/O instructions, there are three variations of each *short* device: */dev/short0* performs the loop just shown, */dev/short0p* uses *outb_p* and *inb_p* in place of the "fast" functions, and */dev/short0s* uses the string instructions. There are eight such devices, from *short0* to *short7*. Although the PC parallel interface has only three ports, you may need more of them if using a different I/O device to run your tests.

The *short* driver performs an absolute minimum of hardware control, but is adequate to show how the I/O port instructions are used. Interested readers may want to look at the source for the *parport* and *parport_pc* modules to see how complicated this device can get in real life in order to support a range of devices (printers, tape backup, network interfaces) on the parallel port.

Using I/O Memory

Despite the popularity of I/O ports in the x86 world, the main mechanism used to communicate with devices is through memory-mapped registers and device memory. Both are called *I/O memory* because the difference between registers and memory is transparent to software.

I/O memory is simply a region of RAM-like locations that the device makes available to the processor over the bus. This memory can be used for a number of purposes, such as holding video data or Ethernet packets, as well as implementing device registers that behave just like I/O ports (i.e., they have side effects associated with reading and writing them).

The way used to access I/O memory depends on the computer architecture, bus, and device being used, though the principles are the same everywhere. The discussion in this chapter touches mainly on ISA and PCI memory, while trying to convey general information as well. Although access to PCI memory is introduced here, a thorough discussion of PCI is deferred to Chapter 15.

According to the computer platform and bus being used, I/O memory may or may not be accessed through page tables. When access passes though page tables, the kernel must first arrange for the physical address to be visible from your driver (this usually means that you must call *ioremap* before doing any I/O). If no page tables are needed, then I/O memory locations look pretty much like I/O ports, and you can just read and write to them using proper wrapper functions.

Whether or not *ioremap* is required to access I/O memory, direct use of pointers to I/O memory is a discouraged practice. Even though (as introduced in "I/O Ports and I/O Memory") I/O memory is addressed like normal RAM at hardware level, the extra care outlined in "I/O Registers and Conventional Memory" suggests avoiding normal pointers. The wrapper functions used to access I/O memory are both safe on all platforms and optimized away whenever straight pointer dereferencing can perform the operation.

Therefore, even though dereferencing a pointer works (for now) on the x86, failure to use the proper macros will hinder the portability and readability of the driver.

Remember from Chapter 2 that device memory regions must be allocated prior to use. This is similar to how I/O ports are registered and is accomplished by the following functions:

```
int check_mem_region(unsigned long start, unsigned long len);
void request_mem_region(unsigned long start, unsigned long len,
char *name);
void release_mem_region(unsigned long start, unsigned long len);
```

The `start` argument to pass to the functions is the physical address of the memory region, before any remapping takes place. The functions would normally be used in a manner such as the following:

```
if (check_mem_region(mem_addr, mem_size)) {
    printk("drivername: memory already in use\n");
    return -EBUSY;
}
    request_mem_region(mem_addr, mem_size, "drivername");
```

```
    [...]

    release_mem_region(mem_addr, mem_size);
```

Directly Mapped Memory

Several computer platforms reserve part of their memory address space for I/O locations, and automatically disable memory management for any (virtual) address in that memory range.

The MIPS processors used in personal digital assistants (PDAs) offer an interesting example of this setup. Two address ranges, 512 MB each, are directly mapped to physical addresses. Any memory access to either of those address ranges bypasses the MMU, and any access to one of those ranges bypasses the cache as well. A section of these 512 megabytes is reserved for peripheral devices, and drivers can access their I/O memory directly by using the noncached address range.

Other platforms have other means to offer directly mapped address ranges: some of them have special address spaces to dereference physical addresses (for example, SPARC64 uses a special "address space identifier" for this aim), and others use virtual addresses set up to bypass processor caches.

When you need to access a directly mapped I/O memory area, you still shouldn't dereference your I/O pointers, even though, on some architectures, you may well be able to get away with doing exactly that. To write code that will work across systems and kernel versions, however, you must avoid direct accesses and instead use the following functions.

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
```

These macros are used to retrieve 8-bit, 16-bit, and 32-bit data values from I/O memory. The advantage of using macros is the typelessness of the argument: `address` is cast before being used, because the value "is not clearly either an integer or a pointer, and we will accept both" (from *asm-alpha/io.h*). Neither the reading nor the writing functions check the validity of `address`, because they are meant to be as fast as pointer dereferencing (we already know that sometimes they actually expand into pointer dereferencing).

```
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

Like the previous functions, these functions (macros) are used to write 8-bit, 16-bit, and 32-bit data items.

`memset_io(address, value, count);`
: When you need to call *memset* on I/O memory, this function does what you need, while keeping the semantics of the original *memset*.

`memcpy_fromio(dest, source, num);`
`memcpy_toio(dest, source, num);`
: These functions move blocks of data to and from I/O memory and behave like the C library routine *memcpy*.

In modern versions of the kernel, these functions are available across all architectures. The implementation will vary, however; on some they are macros that expand to pointer operations, and on others they are real functions. As a driver writer, however, you need not worry about how they work, as long as you use them.

Some 64-bit platforms also offer *readq* and *writeq*, for quad-word (eight-byte) memory operations on the PCI bus. The *quad-word* nomenclature is a historical leftover from the times when all real processors had 16-bit words. Actually, the *L* naming used for 32-bit values has become incorrect too, but renaming everything would make things still more confused.

Reusing short for I/O Memory

The *short* sample module, introduced earlier to access I/O ports, can be used to access I/O memory as well. To this aim, you must tell it to use I/O memory at load time; also, you'll need to change the base address to make it point to your I/O region.

For example, this is how we used *short* to light the debug LEDs on a MIPS development board:

```
mips.root# ./short_load use_mem=1 base=0xb7ffffc0
mips.root# echo -n 7 > /dev/short0
```

Use of *short* for I/O memory is the same as it is for I/O ports; however, since no pausing or string instructions exist for I/O memory, access to */dev/short0p* and */dev/short0s* performs the same operation as */dev/short0*.

The following fragment shows the loop used by *short* in writing to a memory location:

```
while (count--) {
    writeb(*(ptr++), address);
    wmb();
}
```

Note the use of a write memory barrier here. Because *writeb* likely turns into a direct assignment on many architectures, the memory barrier is needed to ensure that the writes happen in the expected order.

Software-Mapped I/O Memory

The MIPS class of processors notwithstanding, directly mapped I/O memory is pretty rare in the current platform arena; this is especially true when a peripheral bus is used with memory-mapped devices (which is most of the time).

The most common hardware and software arrangement for I/O memory is this: devices live at well-known physical addresses, but the CPU has no predefined virtual address to access them. The well-known physical address can be either hardwired in the device or assigned by system firmware at boot time. The former is true, for example, of ISA devices, whose addresses are either burned in device logic circuits, statically assigned in local device memory, or set by means of physical jumpers. The latter is true of PCI devices, whose addresses are assigned by system software and written to device memory, where they persist only while the device is powered on.

Either way, for software to access I/O memory, there must be a way to assign a virtual address to the device. This is the role of the *ioremap* function, introduced in "vmalloc and Friends." The function, which was covered in the previous chapter because it is related to memory use, is designed specifically to assign virtual addresses to I/O memory regions. Moreover, kernel developers implemented *ioremap* so that it doesn't do anything if applied to directly mapped I/O addresses.

Once equipped with *ioremap* (and *iounmap*), a device driver can access any I/O memory address, whether it is directly mapped to virtual address space or not. Remember, though, that these addresses should not be dereferenced directly; instead, functions like *readb* should be used. We could thus arrange *short* to work with both MIPS I/O memory and the more common ISA/PCI x86 memory by equipping the module with *ioremap*/*iounmap* calls whenever the `use_mem` parameter is set.

Before we show how *short* calls the functions, we'd better review the prototypes of the functions and introduce a few details that we passed over in the previous chapter.

The functions are called according to the following definition:

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void * addr);
```

First of all, you'll notice the new function *ioremap_nocache*. We didn't cover it in Chapter 7, because its meaning is definitely hardware related. Quoting from one of the kernel headers: "It's useful if some control registers are in such an area and write combining or read caching is not desirable." Actually, the function's implementation is identical to *ioremap* on most computer platforms: in situations in which all of I/O memory is already visible through noncacheable addresses, there's no reason to implement a separate, noncaching version of *ioremap*.

Another important feature of *ioremap* is the different behavior of the 2.0 version with respect to later ones. Under Linux 2.0, the function (called, remember, *vremap* at the time) refused to remap any non-page-aligned memory region. This was a sensible choice, since at CPU level everything happens with page-sized granularity. However, sometimes you need to map small regions of I/O registers whose (physical) address is not page aligned. To fit this new need, version 2.1.131 and later of the kernel are able to remap unaligned addresses.

Our *short* module, in order to be backward portable to version 2.0 and to be able to access non-page-aligned registers, includes the following code instead of calling *ioremap* directly:

```
/* Remap a not (necessarily) aligned port region */
void *short_remap(unsigned long phys_addr)
{
    /* The code comes mainly from arch/any/mm/ioremap.c */
    unsigned long offset, last_addr, size;

    last_addr = phys_addr + SHORT_NR_PORTS - 1;
    offset = phys_addr & ~PAGE_MASK;

    /* Adjust the begin and end to remap a full page */
    phys_addr &= PAGE_MASK;
    size = PAGE_ALIGN(last_addr) - phys_addr;
    return ioremap(phys_addr, size) + offset;
}

/* Unmap a region obtained with short_remap */
void short_unmap(void *virt_add)
{
    iounmap((void *)((unsigned long)virt_add & PAGE_MASK));
}
```

ISA Memory Below 1 MB

One of the most well-known I/O memory regions is the ISA range as found on personal computers. This is the memory range between 640 KB (`0xA0000`) and 1 MB (`0x100000`). It thus appears right in the middle of regular system RAM. This positioning may seem a little strange; it is an artifact of a decision made in the early 1980s, when 640 KB of memory seemed like more than anybody would ever be able to use.

This memory range belongs to the non-directly-mapped class of memory.* You

* Actually, this is not completely true. The memory range is so small and so frequently used that the kernel builds page tables at boot time to access those addresses. However, the virtual address used to access them is not the same as the physical address, and thus *ioremap* is needed anyway. Moreover, version 2.0 of the kernel had that range directly mapped. See "Backward Compatibility" for 2.0 issues.

can read/write a few bytes in that memory range using the *short* module as explained previously, that is, by setting `use_mem` at load time.

Although ISA I/O memory exists only in x86-class computers, we think it's worth spending a few words and a sample driver on it.

We are not going to discuss PCI memory in this chapter, since it is the cleanest kind of I/O memory: once you know the physical address you can simply remap and access it. The "problem" with PCI I/O memory is that it doesn't lend itself to a working example for this chapter, because we can't know in advance the physical addresses your PCI memory is mapped to, nor whether it's safe to access either of those ranges. We chose to describe the ISA memory range because it's both less clean and more suitable to running sample code.

To demonstrate access to ISA memory, we will make use of yet another silly little module (part of the sample sources). In fact, this one is called *silly*, as an acronym for Simple Tool for Unloading and Printing ISA Data, or something like that.

The module supplements the functionality of *short* by giving access to the whole 384-KB memory space and by showing all the different I/O functions. It features four device nodes that perform the same task using different data transfer functions. The *silly* devices act as a window over I/O memory, in a way similar to */dev/mem*. You can read and write data, and *lseek* to an arbitrary I/O memory address.

Because *silly* provides access to ISA memory, it must start by mapping the physical ISA addresses into kernel virtual addresses. In the early days of the Linux kernel, one could simply assign a pointer to an ISA address of interest, then dereference it directly. In the modern world, though, we must work with the virtual memory system and remap the memory range first. This mapping is done with *ioremap*, as explained earlier for *short*:

```
#define ISA_BASE    0xA0000
#define ISA_MAX    0x100000  /* for general memory access */

    /* this line appears in silly_init */
    io_base = ioremap(ISA_BASE, ISA_MAX - ISA_BASE);
```

ioremap returns a pointer value that can be used with *readb* and the other functions explained in the section "Directly Mapped Memory."

Let's look back at our sample module to see how these functions might be used. */dev/sillyb*, featuring minor number 0, accesses I/O memory with *readb* and *writeb*. The following code shows the implementation for *read*, which makes the address range `0xA0000-0xFFFFF` available as a virtual file in the range `0-0x5FFFF`. The *read* function is structured as a `switch` statement over the different access modes; here is the *sillyb* `case`:

```
case M_8:
  while (count) {
      *ptr = readb(add);
      add++; count--; ptr++;
  }
  break;
```

The next two devices are */dev/sillyw* (minor number 1) and */dev/sillyl* (minor number 2). They act like */dev/sillyb*, except that they use 16-bit and 32-bit functions. Here's the *write* implementation of *sillyl*, again part of a `switch`:

```
case M_32:
  while (count >= 4) {
      writel(*(u32 *)ptr, add);
      add+=4; count-=4; ptr+=4;
  }
  break;
```

The last device is */dev/sillycp* (minor number 3), which uses the *memcpy_*io* functions to perform the same task. Here's the core of its *read* implementation:

```
case M_memcpy:
  memcpy_fromio(ptr, add, count);
  break;
```

Because *ioremap* was used to provide access to the ISA memory area, *silly* must invoke *iounmap* when the module is unloaded:

```
iounmap(io_base);
```

isa_readb and Friends

A look at the kernel source will turn up another set of routines with names like *isa_readb*. In fact, each of the functions just described has an *isa_* equivalent. These functions provide access to ISA memory without the need for a separate *ioremap* step. The word from the kernel developers, however, is that these functions are intended to be temporary driver-porting aids, and that they may go away in the future. Their use is thus best avoided.

Probing for ISA Memory

Even though most modern devices rely on better I/O bus architectures, like PCI, sometimes programmers must still deal with ISA devices and their I/O memory, so we'll spend a page on this issue. We won't touch high ISA memory (the so-called memory hole in the 14 MB to 16 MB physical address range), because that kind of I/O memory is extremely rare nowadays and is not supported by the majority of modern motherboards or by the kernel. To access that range of I/O memory you'd need to hack the kernel initialization sequence, and that is better not covered here.

When using ISA memory-mapped devices, the driver writer often ignores where relevant I/O memory is located in the physical address space, since the actual address is usually assigned by the user among a range of possible addresses. Or it may be necessary simply to see if a device is present at a given address or not.

The memory resource management scheme can be helpful in probing, since it will identify regions of memory that have already been claimed by another driver. The resource manager, however, cannot tell you about devices whose drivers have not been loaded, or whether a given region contains the device that you are interested in. Thus, it can still be necessary to actually probe memory to see what is there. There are three distinct cases that you will encounter: that RAM is mapped to the address, that ROM is there (the VGA BIOS, for example), or that the area is free.

The *skull* sample source shows a way to deal with such memory, but since *skull* is not related to any physical device, it just prints information about the 640 KB to 1 MB memory region and then exits. However, the code used to analyze memory is worth describing, since it shows how memory probes can be done.

The code to check for RAM segments makes use of *cli* to disable interrupts, because these segments can be identified only by physically writing and rereading data, and real RAM might be changed by an interrupt handler in the middle of our tests. The following code is not completely foolproof, because it might mistake RAM memory on acquisition boards for empty regions if a device is actively writing to its own memory while this code is scanning the area. However, this situation is quite unlikely to happen.

```
unsigned char oldval, newval; /* values read from memory   */
unsigned long flags;          /* used to hold system flags */
unsigned long add, i;
void *base;

/* Use ioremap to get a handle on our region */
base = ioremap(ISA_REGION_BEGIN, ISA_REGION_END - ISA_REGION_BEGIN);
base -= ISA_REGION_BEGIN;  /* Do the offset once */

/* probe all the memory hole in 2-KB steps */
for (add = ISA_REGION_BEGIN; add < ISA_REGION_END; add += STEP) {
   /*
    * Check for an already allocated region.
    */
   if (check_mem_region (add, 2048)) {
          printk(KERN_INFO "%lx: Allocated\n", add);
          continue;
   }
   /*
    * Read and write the beginning of the region and see what happens.
    */
   save_flags(flags);
   cli();
   oldval = readb (base + add);  /* Read a byte */
```

```
    writeb (oldval^0xff, base + add);
    mb();
    newval = readb (base + add);
    writeb (oldval, base + add);
    restore_flags(flags);

    if ((oldval^newval) == 0xff) {  /* we reread our change: it's RAM */
        printk(KERN_INFO "%lx: RAM\n", add);
        continue;
    }
    if ((oldval^newval) != 0) {  /* random bits changed: it's empty */
        printk(KERN_INFO "%lx: empty\n", add);
        continue;
    }

    /*
     * Expansion ROM (executed at boot time by the BIOS)
     * has a signature where the first byte is 0x55, the second 0xaa,
     * and the third byte indicates the size of such ROM
     */
    if ( (oldval == 0x55) && (readb (base + add + 1) == 0xaa)) {
        int size = 512 * readb (base + add + 2);
        printk(KERN_INFO "%lx: Expansion ROM, %i bytes\n",
               add, size);
        add += (size & ~2048) - 2048; /* skip it */
        continue;
    }

    /*
     * If the tests above failed, we still don't know if it is ROM or
     * empty. Since empty memory can appear as 0x00, 0xff, or the low
     * address byte, we must probe multiple bytes: if at least one of
     * them is different from these three values, then this is ROM
     * (though not boot ROM).
     */
    printk(KERN_INFO "%lx: ", add);
    for (i=0; i<5; i++) {
        unsigned long radd = add + 57*(i+1);  /* a "random" value */
        unsigned char val = readb (base + radd);
        if (val && val != 0xFF && val != ((unsigned long) radd&0xFF))
           break;
    }
    printk("%s\n", i==5 ? "empty" : "ROM");
}
```

Detecting memory doesn't cause collisions with other devices, as long as you take care to restore any byte you modified while you were probing. It is worth noting that it is always possible that writing to another device's memory will cause that device to do something undesirable. In general, this method of probing memory should be avoided if possible, but it's not always possible when dealing with older hardware.

Backward Compatibility

Happily, little has changed with regard to basic hardware access. There are just a few things that need to be kept in mind when writing backward-compatible drivers.

Hardware memory barriers didn't exist in version 2.0 of the kernel. There was no need for such ordering instructions on the platforms then supported. Including *sysdep.h* in your driver will fix the problem by defining hardware barriers to be the same as software barriers.

Similarly, not all of the port-access functions (*inb* and friends) were supported on all architectures in older kernels. The string functions, in particular, tended to be absent. We don't provide the missing functions in our *sysdep.h* facility: it won't be an easy task to perform cleanly and most likely is not worth the effort, given the hardware dependency of those functions.

In Linux 2.0, *ioremap* and *iounmap* were called *vremap* and *vfree*, respectively. The parameters and the functionality were the same. Thus, a couple of definitions that map the functions to their older counterpart are often enough.

Unfortunately, while *vremap* worked just like *ioremap* for providing access to "high" memory (such as that on PCI cards), it did refuse to remap the ISA memory ranges. Back in those days, access to this memory was done via direct pointers, so there was no need to remap that address space. Thus, a more complete solution to implement *ioremap* for Linux 2.0 running on the x86 platform is as follows:

```
extern inline void *ioremap(unsigned long phys_addr, unsigned long size)
{
    if (phys_addr >= 0xA0000 && phys_addr + size <= 0x100000)
        return (void *)phys_addr;
    return vremap(phys_addr, size);
}

extern inline void iounmap(void *addr)
{
    if ((unsigned long)addr >= 0xA0000
            && (unsigned long)addr < 0x100000)
        return;
    vfree(addr);
}
```

If you include *sysdep.h* in your drivers you'll be able to use *ioremap* with no problems even when accessing ISA memory.

Allocation of memory regions (*check_mem_region* and friends) was introduced in kernel 2.3.17. In the 2.0 and 2.2 kernels, there was no central facility for the allocation of memory resources. You can use the macros anyway if you include *sysdep.h* because it nullifies the three macros when compiling for 2.0 or 2.2.

Quick Reference

This chapter introduced the following symbols related to hardware management.

```
#include <linux/kernel.h>
void barrier(void)
```

This "software" memory barrier requests the compiler to consider all memory volatile across this instruction.

```
#include <asm/system.h>
void rmb(void);
void wmb(void);
void mb(void);
```

Hardware memory barriers. They request the CPU (and the compiler) to checkpoint all memory reads, writes, or both, across this instruction.

```
#include <asm/io.h>
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);
```

These functions are used to read and write I/O ports. They can also be called by user-space programs, provided they have the right privileges to access ports.

```
unsigned inb_p(unsigned port);
...
```

The statement `SLOW_DOWN_IO` is sometimes needed to deal with slow ISA boards on the x86 platform. If a small delay is needed after an I/O operation, you can use the six pausing counterparts of the functions introduced in the previous entry; these pausing functions have names ending in `_p`.

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

The "string functions" are optimized to transfer data from an input port to a region of memory, or the other way around. Such transfers are performed by reading or writing the same port `count` times.

```
#include <linux/ioport.h>
int check_region(unsigned long start, unsigned long len);
void request_region(unsigned long start, unsigned long len,
      char *name);
void release_region(unsigned long start, unsigned long len);
```

Resource allocators for I/O ports. The *check* function returns 0 for success and less than 0 in case of error.

```
int check_mem_region(unsigned long start, unsigned long
      len);
void request_mem_region(unsigned long start, unsigned long
      len, char *name);
void release_mem_region(unsigned long start, unsigned long
      len);
```

These functions handle resource allocation for memory regions.

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long
      size);
void iounmap(void *virt_addr);
```

ioremap remaps a physical address range into the processor's virtual address space, making it available to the kernel. *iounmap* frees the mapping when it is no longer needed.

```
#include <linux/io.h>
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
memset_io(address, value, count);
memcpy_fromio(dest, source, nbytes);
memcpy_toio(dest, source, nbytes);
```

These functions are used to access I/O memory regions, either low ISA memory or high PCI buffers.