CHAPTER 1
An Introduction to Device Drivers

One of the many advantages of free operating systems, as typified by Linux, is that their internals are open for all to view. The operating system, once a dark and mysterious area whose code was restricted to a small number of programmers, can now be readily examined, understood, and modified by anybody with the requisite skills. Linux has helped to democratize operating systems. The Linux kernel remains a large and complex body of code, however, and would-be kernel hackers need an entry point where they can approach the code without being overwhelmed by complexity. Often, device drivers provide that gateway.

Device drivers take on a special role in the Linux kernel. They are distinct "black boxes" that make a particular piece of hardware respond to a well-defined internal programming interface; they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver; mapping those calls to device-specific operations that act on real hardware is then the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and "plugged in" at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available.

There are a number of reasons to be interested in the writing of Linux device drivers. The rate at which new hardware becomes available (and obsolete!) alone guarantees that driver writers will be busy for the foreseeable future. Individuals may need to know about drivers in order to gain access to a particular device that is of interest to them. Hardware vendors, by making a Linux driver available for their products, can add the large and growing Linux user base to their potential markets. And the open source nature of the Linux system means that if the driver writer wishes, the source to a driver can be quickly disseminated to millions of users.

This book teaches you how to write your own drivers and how to hack around in related parts of the kernel. We have taken a device-independent approach; the programming techniques and interfaces are presented, whenever possible, without being tied to any specific device. Each driver is different; as a driver writer, you need to

understand your specific device well. But most of the principles and basic techniques are the same for all drivers. This book cannot teach you about your device, but it gives you a handle on the background you need to make your device work.

As you learn to write drivers, you find out a lot about the Linux kernel in general; this may help you understand how your machine works and why things aren't always as fast as you expect or don't do quite what you want. We introduce new ideas gradually, starting off with very simple drivers and building on them; every new concept is accompanied by sample code that doesn't need special hardware to be tested.

This chapter doesn't actually get into writing code. However, we introduce some background concepts about the Linux kernel that you'll be glad you know later, when we do launch into programming.

The Role of the Device Driver

As a programmer, you are able to make your own choices about your driver, and choose an acceptable trade-off between the programming time required and the flexibility of the result. Though it may appear strange to say that a driver is "flexible," we like this word because it emphasizes that the role of a device driver is providing *mechanism*, not *policy*.

The distinction between mechanism and policy is one of the best ideas behind the Unix design. Most programming problems can indeed be split into two parts: "what capabilities are to be provided" (the mechanism) and "how those capabilities can be used" (the policy). If the two issues are addressed by different parts of the program, or even by different programs altogether, the software package is much easier to develop and to adapt to particular needs.

For example, Unix management of the graphic display is split between the X server, which knows the hardware and offers a unified interface to user programs, and the window and session managers, which implement a particular policy without knowing anything about the hardware. People can use the same window manager on different hardware, and different users can run different configurations on the same workstation. Even completely different desktop environments, such as KDE and GNOME, can coexist on the same system. Another example is the layered structure of TCP/IP networking: the operating system offers the socket abstraction, which implements no policy regarding the data to be transferred, while different servers are in charge of the services (and their associated policies). Moreover, a server like *ftpd* provides the file transfer mechanism, while users can use whatever client they prefer; both command-line and graphic clients exist, and anyone can write a new user interface to transfer files.

Where drivers are concerned, the same separation of mechanism and policy applies. The floppy driver is policy free—its role is only to show the diskette as a continuous

array of data blocks. Higher levels of the system provide policies, such as who may access the floppy drive, whether the drive is accessed directly or via a filesystem, and whether users may mount filesystems on the drive. Since different environments usually need to use hardware in different ways, it's important to be as policy free as possible.

When writing drivers, a programmer should pay particular attention to this fundamental concept: write kernel code to access the hardware, but don't force particular policies on the user, since different users have different needs. The driver should deal with making the hardware available, leaving all the issues about *how* to use the hardware to the applications. A driver, then, is flexible if it offers access to the hardware capabilities without adding constraints. Sometimes, however, some policy decisions must be made. For example, a digital I/O driver may only offer byte-wide access to the hardware in order to avoid the extra code needed to handle individual bits.

You can also look at your driver from a different perspective: it is a software layer that lies between the applications and the actual device. This privileged role of the driver allows the driver programmer to choose exactly how the device should appear: different drivers can offer different capabilities, even for the same device. The actual driver design should be a balance between many different considerations. For instance, a single device may be used concurrently by different programs, and the driver programmer has complete freedom to determine how to handle concurrency. You could implement memory mapping on the device independently of its hardware capabilities, or you could provide a user library to help application programmers implement new policies on top of the available primitives, and so forth. One major consideration is the trade-off between the desire to present the user with as many options as possible and the time you have to write the driver, as well as the need to keep things simple so that errors don't creep in.

Policy-free drivers have a number of typical characteristics. These include support for both synchronous and asynchronous operation, the ability to be opened multiple times, the ability to exploit the full capabilities of the hardware, and the lack of software layers to "simplify things" or provide policy-related operations. Drivers of this sort not only work better for their end users, but also turn out to be easier to write and maintain as well. Being policy-free is actually a common target for software designers.

Many device drivers, indeed, are released together with user programs to help with configuration and access to the target device. Those programs can range from simple utilities to complete graphical applications. Examples include the *tunelp* program, which adjusts how the parallel port printer driver operates, and the graphical *cardctl* utility that is part of the PCMCIA driver package. Often a client library is provided as well, which provides capabilities that do not need to be implemented as part of the driver itself.

The scope of this book is the kernel, so we try not to deal with policy issues or with application programs or support libraries. Sometimes we talk about different policies and how to support them, but we won't go into much detail about programs using the device or the policies they enforce. You should understand, however, that user programs are an integral part of a software package and that even policy-free packages are distributed with configuration files that apply a default behavior to the underlying mechanisms.

Splitting the Kernel

In a Unix system, several concurrent *processes* attend to different tasks. Each process asks for system resources, be it computing power, memory, network connectivity, or some other resource. The *kernel* is the big chunk of executable code in charge of handling all such requests. Although the distinction between the different kernel tasks isn't always clearly marked, the kernel's role can be split (as shown in Figure 1-1) into the following parts:

Process management
: The kernel is in charge of creating and destroying processes and handling their connection to the outside world (input and output). Communication among different processes (through signals, pipes, or interprocess communication primitives) is basic to the overall system functionality and is also handled by the kernel. In addition, the scheduler, which controls how processes share the CPU, is part of process management. More generally, the kernel's process management activity implements the abstraction of several processes on top of a single CPU or a few of them.

Memory management
: The computer's memory is a major resource, and the policy used to deal with it is a critical one for system performance. The kernel builds up a virtual addressing space for any and all processes on top of the limited available resources. The different parts of the kernel interact with the memory-management subsystem through a set of function calls, ranging from the simple *malloc/free* pair to much more complex functionalities.

Filesystems
: Unix is heavily based on the filesystem concept; almost everything in Unix can be treated as a file. The kernel builds a structured filesystem on top of unstructured hardware, and the resulting file abstraction is heavily used throughout the whole system. In addition, Linux supports multiple filesystem types, that is, different ways of organizing data on the physical medium. For example, disks may be formatted with the Linux-standard ext3 filesystem, the commonly used FAT filesystem or several others.

page structures; they are especially useful in any situation where you might be dealing with high memory, which does not have a constant address in kernel space.

The *real* core of the Linux page allocator is a function called *alloc_pages_node*:

```
struct page *alloc_pages_node(int nid, unsigned int flags,
                              unsigned int order);
```

This function also has two variants (which are simply macros); these are the versions that you will most likely use:

```
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);
```

The core function, *alloc_pages_node*, takes three arguments. `nid` is the NUMA node ID* whose memory should be allocated, `flags` is the usual `GFP_` allocation flags, and `order` is the size of the allocation. The return value is a pointer to the first of (possibly many) `page` structures describing the allocated memory, or, as usual, `NULL` on failure.

alloc_pages simplifies the situation by allocating the memory on the current NUMA node (it calls *alloc_pages_node* with the return value from *numa_node_id* as the `nid` parameter). And, of course, *alloc_page* omits the `order` parameter and allocates a single page.

To release pages allocated in this manner, you should use one of the following:

```
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
void free_cold_page(struct page *page);
```

If you have specific knowledge of whether a single page's contents are likely to be resident in the processor cache, you should communicate that to the kernel with *free_hot_page* (for cache-resident pages) or *free_cold_page*. This information helps the memory allocator optimize its use of memory across the system.

vmalloc and Friends

The next memory allocation function that we show you is *vmalloc*, which allocates a contiguous memory region in the *virtual* address space. Although the pages are not consecutive in physical memory (each page is retrieved with a separate call to *alloc_page*), the kernel sees them as a contiguous range of addresses. *vmalloc* returns `0` (the `NULL` address) if an error occurs, otherwise, it returns a pointer to a linear memory area of size at least `size`.

* NUMA (nonuniform memory access) computers are multiprocessor systems where memory is "local" to specific groups of processors ("nodes"). Access to local memory is faster than access to nonlocal memory. On such systems, allocating memory on the correct node is important. Driver authors do not normally have to worry about NUMA issues, however.

We describe *vmalloc* here because it is one of the fundamental Linux memory allocation mechanisms. We should note, however, that use of *vmalloc* is discouraged in most situations. Memory obtained from *vmalloc* is slightly less efficient to work with, and, on some architectures, the amount of address space set aside for *vmalloc* is relatively small. Code that uses *vmalloc* is likely to get a chilly reception if submitted for inclusion in the kernel. If possible, you should work directly with individual pages rather than trying to smooth things over with *vmalloc*.

That said, let's see how *vmalloc* works. The prototypes of the function and its relatives (*ioremap*, which is not strictly an allocation function, is discussed later in this section) are as follows:

```
#include <linux/vmalloc.h>

void *vmalloc(unsigned long size);
void vfree(void * addr);
void *ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

It's worth stressing that memory addresses returned by *kmalloc* and *_get_free_pages* are also virtual addresses. Their actual value is still massaged by the MMU (the memory management unit, usually part of the CPU) before it is used to address physical memory.* *vmalloc* is not different in how it uses the hardware, but rather in how the kernel performs the allocation task.

The (virtual) address range used by *kmalloc* and *__get_free_pages* features a one-to-one mapping to physical memory, possibly shifted by a constant `PAGE_OFFSET` value; the functions don't need to modify the page tables for that address range. The address range used by *vmalloc* and *ioremap*, on the other hand, is completely synthetic, and each allocation builds the (virtual) memory area by suitably setting up the page tables.

This difference can be perceived by comparing the pointers returned by the allocation functions. On some platforms (for example, the x86), addresses returned by *vmalloc* are just beyond the addresses that *kmalloc* uses. On other platforms (for example, MIPS, IA-64, and x86_64), they belong to a completely different address range. Addresses available for *vmalloc* are in the range from `VMALLOC_START` to `VMALLOC_END`. Both symbols are defined in *<asm/pgtable.h>*.

Addresses allocated by *vmalloc* can't be used outside of the microprocessor, because they make sense only on top of the processor's MMU. When a driver needs a real physical address (such as a DMA address, used by peripheral hardware to drive the system's bus), you can't easily use *vmalloc*. The right time to call *vmalloc* is when

* Actually, some architectures define ranges of "virtual" addresses as reserved to address physical memory. When this happens, the Linux kernel takes advantage of the feature, and both the kernel and *__get_free_pages* addresses lie in one of those memory ranges. The difference is transparent to device drivers and other code that is not directly involved with the memory-management kernel subsystem.

you are allocating memory for a large sequential buffer that exists only in software. It's important to note that *vmalloc* has more overhead than *__get_free_pages*, because it must both retrieve the memory and build the page tables. Therefore, it doesn't make sense to call *vmalloc* to allocate just one page.

An example of a function in the kernel that uses *vmalloc* is the *create_module* system call, which uses *vmalloc* to get space for the module being created. Code and data of the module are later copied to the allocated space using *copy_from_user*. In this way, the module appears to be loaded into contiguous memory. You can verify, by looking in */proc/kallsyms*, that kernel symbols exported by modules lie in a different memory range from symbols exported by the kernel proper.

Memory allocated with *vmalloc* is released by *vfree*, in the same way that *kfree* releases memory allocated by *kmalloc*.

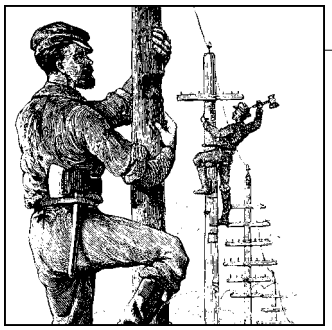
Like *vmalloc*, *ioremap* builds new page tables; unlike *vmalloc*, however, it doesn't actually allocate any memory. The return value of *ioremap* is a special virtual address that can be used to access the specified physical address range; the virtual address obtained is eventually released by calling *iounmap*.

ioremap is most useful for mapping the (physical) address of a PCI buffer to (virtual) kernel space. For example, it can be used to access the frame buffer of a PCI video device; such buffers are usually mapped at high physical addresses, outside of the address range for which the kernel builds page tables at boot time. PCI issues are explained in more detail in Chapter 12.

It's worth noting that for the sake of portability, you should not directly access addresses returned by *ioremap* as if they were pointers to memory. Rather, you should always use *readb* and the other I/O functions introduced in Chapter 9. This requirement applies because some platforms, such as the Alpha, are unable to directly map PCI memory regions to the processor address space because of differences between PCI specs and Alpha processors in how data is transferred.

Both *ioremap* and *vmalloc* are page oriented (they work by modifying the page tables); consequently, the relocated or allocated size is rounded up to the nearest page boundary. *ioremap* simulates an unaligned mapping by "rounding down" the address to be remapped and by returning an offset into the first remapped page.

One minor drawback of *vmalloc* is that it can't be used in atomic context because, internally, it uses *kmalloc(GFP_KERNEL)* to acquire storage for the page tables, and therefore could sleep. This shouldn't be a problem—if the use of *__get_free_page* isn't good enough for an interrupt handler, the software design needs some cleaning up.



CHAPTER 9

Communicating with Hardware

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 until 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 (such as 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 an input/output port. Also, the 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 or a printer 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, 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 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 the 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 platforms 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 the compiler level and at the 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. 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. Therefore, a driver must ensure that no caching is performed and no read or write reordering takes place when accessing registers.

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 stores to memory all values that are currently modified and resident in CPU registers, and rereads them later when they are needed. A call to *barrier* prevents compiler optimizations across the barrier but leaves the hardware free to do its own reordering.

```
#include <asm/system.h>
void rmb(void);
void read_barrier_depends(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*.

read_barrier_depends is a special, weaker form of read barrier. Whereas *rmb* prevents the reordering of all reads across the barrier, *read_barrier_depends* blocks only the reordering of reads that depend on data from other reads. The distinction is subtle, and it does not exist on all architectures. Unless you understand exactly what is going on, and you have a reason to believe that a full read barrier is exacting an excessive performance cost, you should probably stick to using *rmb*.

```
void smp_rmb(void);
void smp_read_barrier_depends(void);
void smp_wmb(void);
void smp_mb(void);
```

These versions of the barrier macros insert hardware barriers only when the kernel is compiled for SMP systems; otherwise, they all expand to a simple *barrier* call.

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 enforces the completion of the writes in the necessary order.

Because memory barriers affect performance, they should be used only where they are 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()* is 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. Also worthy of note is that some peripheral buses (such as the PCI bus) have caching issues of their own; we discuss those when we get to them in later chapters.

Some architectures allow the efficient combination of an assignment and a memory barrier. 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. Note that *set_rmb* is defined only by a small number of architectures. (The use of a `do...while` construct is a standard C idiom that causes the expanded macro to work as a normal C statement in all contexts.)

Using I/O Ports

I/O ports are the means by which drivers communicate with many devices, 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.

I/O Port Allocation

As you might expect, you should not go off and start pounding on I/O ports without first ensuring that you have exclusive access to those ports. The kernel provides a registration interface that allows your driver to claim the ports it needs. The core function in that interface is *request_region*:

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n,
                                const char *name);
```

This function tells the kernel that you would like to make use of `n` ports, starting with `first`. The `name` parameter should be the name of your device. The return value is non-`NULL` if the allocation succeeds. If you get `NULL` back from *request_region*, you will not be able to use the desired ports.

All port allocations show up in */proc/ioports*. If you are unable to allocate a needed set of ports, that is the place to look to see who got there first.

When you are done with a set of I/O ports (at module unload time, perhaps), they should be returned to the system with:

```
void release_region(unsigned long start, unsigned long n);
```

There is also a function that allows your driver to check to see whether a given set of I/O ports is available:

```
int check_region(unsigned long first, unsigned long n);
```

Here, the return value is a negative error code if the given ports are not available. This function is deprecated because its return value provides no guarantee of whether an allocation would succeed; checking and later allocating are not an atomic operation. We list it here because several drivers are still using it, but you should always use *request_region*, which performs the required locking to ensure that the allocation is done in a safe, atomic manner.

Manipulating I/O ports

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 end, 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:

```
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 (one word wide); they are not available when compiling for the S390 platform, which supports only byte I/O.

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

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

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 stick to this convention of "incomplete typing" throughout this chapter.

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.

* 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 generally, you can't count on this feature.

I/O Port Access from User Space

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 of these functions are x86-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, however, that the meaning of the file is very platform-specific and not likely 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 (e.g., *inb*, *inw*, and *inl* and manipulates byte, word, or long ports depending on which name was invoked by the user). They use *ioperm* or *iopl* under x86, */dev/port* on other platforms.

The programs can be made setuid root, if you want to live dangerously and play with your hardware without acquiring explicit privileges. Please do not install them setuid on a production system, however; they are a security hole by design.

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 macros implement the concept of string I/O either by 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 S390 platform. This should not be a portability problem, since this platform doesn't usually share device drivers with other platforms, because its peripheral buses are different.

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

The prototypes for string functions are:

```
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.

There is one thing to keep in mind when using the string functions: they move a straight byte stream to or from the port. When the port and the host system have different byte ordering rules, the results can be surprising. Reading a port with *inw* swaps the bytes, if need be, to make the value read match the host ordering. The string functions, instead, do not perform this swapping.

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 when the processor is overclocked with respect to the peripheral bus (think ISA here) 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. On the x86, the pause is achieved by performing an `out b` instruction to port 0x80 (normally but not always unused), or by busy waiting. See the *io.h* file under your platform's *asm* subdirectory for details.

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 reasonably modern 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 are, therefore, 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, here is an overview of the capabilities of the architectures supported by the kernel:

IA-32 (x86)
x86_64
: 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`.

Cris
: This architecture does not support the I/O port abstraction even in an emulated mode; the various port operations are defined to do nothing at all.

M68k
M68k-nommu
: Ports are memory-mapped. 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 long`.

PA-RISC
: All of the functions are supported; ports are `int` on PCI-based systems and `unsigned short` on EISA systems, except for string operations, which use `unsigned long` port numbers.

PowerPC
PowerPC64
: All the functions are supported; ports have type `unsigned char *` on 32-bit systems and `unsigned long` on 64-bit systems.

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 separate 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 PDFs on the Web.

* Single-byte I/O is not as important as one may imagine, because it is a rare operation. 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.

An I/O Port Example

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.

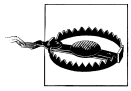
The actual implementation and software interface of such I/O ports varies from system to system. Most of the time, I/O pins are controlled by two I/O locations: one that allows selecting what pins are used as input and what pins are used as output and one in which you can actually read or write logic levels. Sometimes, however, things are even simpler, and the bits are hardwired as either input or output (but, in this case, they're no longer called "general-purpose I/O"); 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 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 2–9 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 9-1. 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 use this bit as part of our implementation of an interrupt handler in Chapter 10.

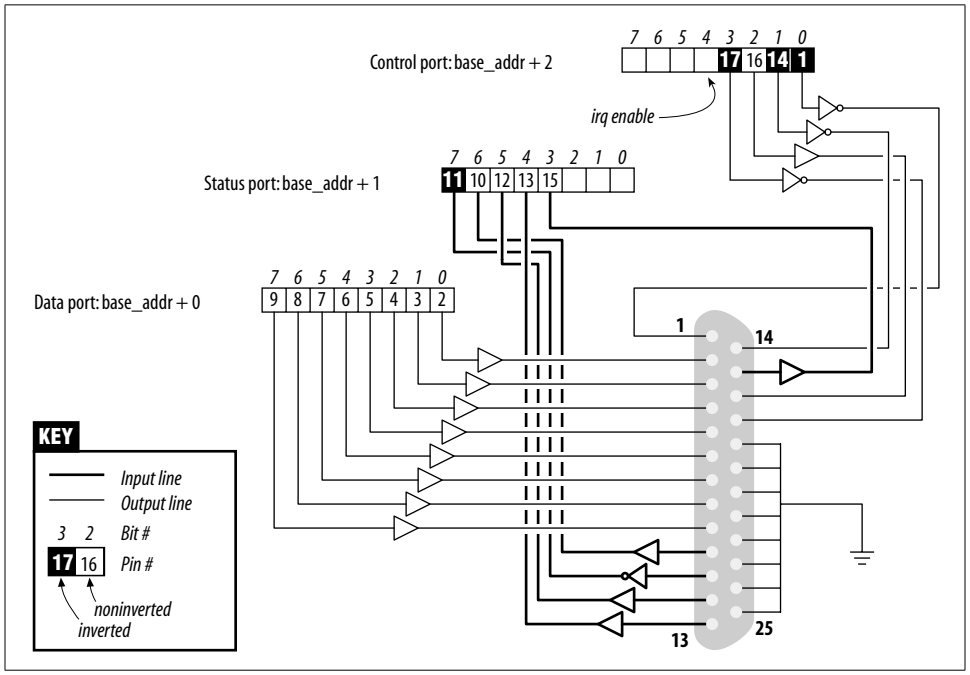


Figure 9-1. The pinout of the parallel port

A Sample Driver

The driver we introduce is called *short* (Simple Hardware Operations and Raw Tests). All it does is read and write a few 8-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 as 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* usually tells 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 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 the section "Using I/O Memory," we 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. In the next chapter, we show a sample driver (called *shortprint*), that is capable of driving parallel printers; that driver uses interrupts, however, so we can't get to it quite yet.

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 connect them together later to run the sample code shown in Chapter 10.

As far as *short* is concerned, */dev/short0* writes to and reads from the 8-bit port located at the I/O address `base` (0x378 unless changed at load time). */dev/short1* writes to the 8-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++), port);
    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 read an endless stream of identical bytes. If you choose to read from an output port, you 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 such as:

```
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 *short*7. 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 to access I/O memory depends on the computer architecture, bus, and device being used, although 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 12.

Depending on 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, and this usually means that you must call *ioremap* before doing any I/O. If no page tables are needed, 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 discouraged. Even though (as introduced in the section "I/O Ports and I/O Memory") I/O memory is addressed like normal RAM at hardware level, the extra care outlined in the section "I/O Registers and Conventional Memory" suggests avoiding normal pointers. The wrapper functions used to access I/O memory are safe on all platforms and are 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 hinders the portability and readability of the driver.

I/O Memory Allocation and Mapping

I/O memory regions must be allocated prior to use. The interface for allocation of memory regions (defined in *<linux/ioport.h>*) is:

```
struct resource *request_mem_region(unsigned long start, unsigned long len,
                                    char *name);
```

This function allocates a memory region of `len` bytes, starting at `start`. If all goes well, a non-`NULL` pointer is returned; otherwise the return value is `NULL`. All I/O memory allocations are listed in */proc/iomem*.

Memory regions should be freed when no longer needed:

```
void release_mem_region(unsigned long start, unsigned long len);
```

There is also an old function for checking I/O memory region availability:

```
int check_mem_region(unsigned long start, unsigned long len);
```

But, as with *check_region*, this function is unsafe and should be avoided.

Allocation of I/O memory is not the only required step before that memory may be accessed. You must also ensure that this I/O memory has been made accessible to the kernel. Getting at I/O memory is not just a matter of dereferencing a pointer; on many systems, I/O memory is not directly accessible in this way at all. So a mapping must be set up first. This is the role of the *ioremap* function, introduced in the section

"vmalloc and Friends" in Chapter 1. The function is designed specifically to assign virtual addresses to I/O memory regions.

Once equipped with *ioremap* (and *iounmap*), a device driver can access any I/O memory address, whether or not it is directly mapped to virtual address space. Remember, though, that the addresses returned from *ioremap* should not be dereferenced directly; instead, accessor functions provided by the kernel should be used. Before we get into those functions, we'd better review the *ioremap* prototypes 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 notice the new function *ioremap_nocache*. We didn't cover it in Chapter 8, 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 where all of I/O memory is already visible through noncacheable addresses, there's no reason to implement a separate, noncaching version of *ioremap*.

Accessing I/O Memory

On some platforms, you may get away with using the return value from *ioremap* as a pointer. Such use is not portable, and, increasingly, the kernel developers have been working to eliminate any such use. The proper way of getting at I/O memory is via a set of functions (defined via *<asm/io.h>*) provided for that purpose.

To read from I/O memory, use one of the following:

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

Here, `addr` should be an address obtained from *ioremap* (perhaps with an integer offset); the return value is what was read from the given I/O memory.

There is a similar set of functions for writing to I/O memory:

```
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

If you must read or write a series of values to a given I/O memory address, you can use the repeating versions of the functions:

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
```

```
void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

These functions read or write `count` values from the given `buf` to the given `addr`. Note that `count` is expressed in the size of the data being written; *ioread32_rep* reads `count` 32-bit values starting at `buf`.

The functions described above perform all I/O to the given `addr`. If, instead, you need to operate on a block of I/O memory, you can use one of the following:

```
void memset_io(void *addr, u8 value, unsigned int count);
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

These functions behave like their C library analogs.

If you read through the kernel source, you see many calls to an older set of functions when I/O memory is being used. These functions still work, but their use in new code is discouraged. Among other things, they are less safe because they do not perform the same sort of type checking. Nonetheless, we describe them here:

```
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.

```
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.

Some 64-bit platforms also offer *readq* and *writeq*, for quad-word (8-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 confuse things even more.

Ports as I/O Memory

Some hardware has an interesting feature: some versions use I/O ports, while others use I/O memory. The registers exported to the processor are the same in either case, but the access method is different. As a way of making life easier for drivers dealing with this kind of hardware, and as a way of minimizing the apparent differences between I/O port and memory accesses, the 2.6 kernel provides a function called *ioport_map*:

```
void *ioport_map(unsigned long port, unsigned int count);
```

This function remaps `count` I/O ports and makes them appear to be I/O memory. From that point thereafter, the driver may use *ioread8* and friends on the returned addresses and forget that it is using I/O ports at all.

This mapping should be undone when it is no longer needed:

```
void ioport_unmap(void *addr);
```

These functions make I/O ports look like memory. Do note, however, that the I/O ports must still be allocated with *request_region* before they can be remapped in this way.

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 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.

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

```
while (count--) {
    iowrite8(*ptr++, address);
    wmb( );
}
```

Note the use of a write memory barrier here. Because *iowrite8* likely turns into a direct assignment on many architectures, the memory barrier is needed to ensure that the writes happen in the expected order.

short uses *inb* and *outb* to show how that is done. It would be a straightforward exercise for the reader, however, to change *short* to remap I/O ports with *ioport_map*, and simplify the rest of the code considerably.

ISA Memory Below 1 MB

One of the most well-known I/O memory regions is the ISA range found on personal computers. This is the memory range between 640 KB (`0xA0000`) and 1 MB (`0x100000`). Therefore, it 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.