CHAPTER THIRTEEN

MMAP AND DMA

This chapter delves into the area of Linux memory management, with an emphasis on techniques that are useful to the device driver writer. The material in this chapter is somewhat advanced, and not everybody will need a grasp of it. Nonetheless, many tasks can only be done through digging more deeply into the memory management subsystem; it also provides an interesting look into how an important part of the kernel works.

The material in this chapter is divided into three sections. The first covers the implementation of the *mmap* system call, which allows the mapping of device memory directly into a user process's address space. We then cover the kernel `kiobuf` mechanism, which provides direct access to user memory from kernel space. The `kiobuf` system may be used to implement "raw I/O" for certain kinds of devices. The final section covers direct memory access (DMA) I/O operations, which essentially provide peripherals with direct access to system memory.

Of course, all of these techniques require an understanding of how Linux memory management works, so we start with an overview of that subsystem.

Memory Management in Linux

Rather than describing the theory of memory management in operating systems, this section tries to pinpoint the main features of the Linux implementation of the theory. Although you do not need to be a Linux virtual memory guru to implement *mmap*, a basic overview of how things work is useful. What follows is a fairly lengthy description of the data structures used by the kernel to manage memory. Once the necessary background has been covered, we can get into working with these structures.

Address Types

Linux is, of course, a virtual memory system, meaning that the addresses seen by user programs do not directly correspond to the physical addresses used by the hardware. Virtual memory introduces a layer of indirection, which allows a number of nice things. With virtual memory, programs running on the system can allocate far more memory than is physically available; indeed, even a single process can have a virtual address space larger than the system's physical memory. Virtual memory also allows playing a number of tricks with the process's address space, including mapping in device memory.

Thus far, we have talked about virtual and physical addresses, but a number of the details have been glossed over. The Linux system deals with several types of addresses, each with its own semantics. Unfortunately, the kernel code is not always very clear on exactly which type of address is being used in each situation, so the programmer must be careful.

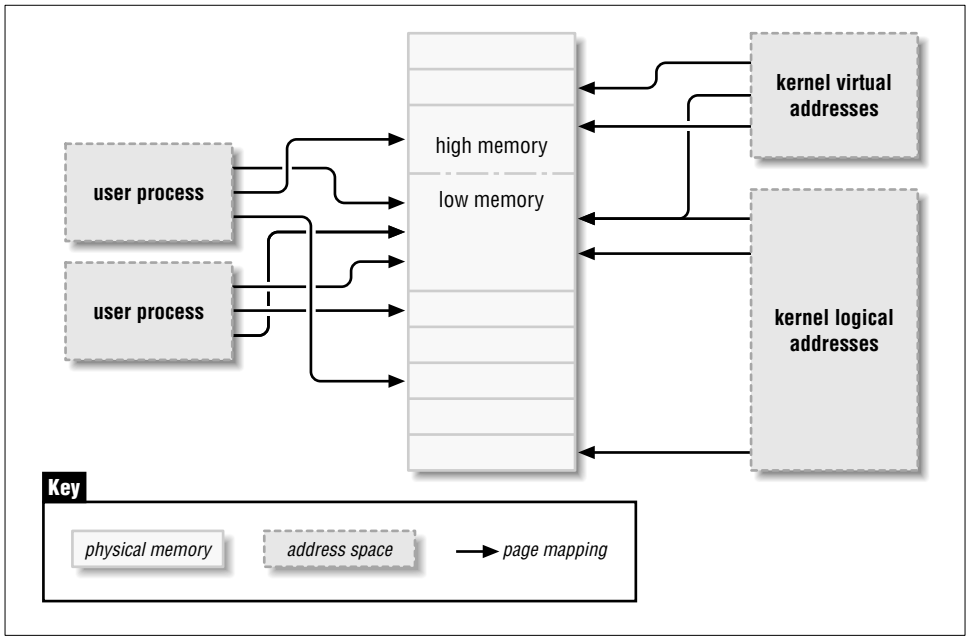

Figure 13-1. Address types used in Linux

The following is a list of address types used in Linux. Figure 13-1 shows how these address types relate to physical memory.

User virtual addresses
: These are the regular addresses seen by user-space programs. User addresses are either 32 or 64 bits in length, depending on the underlying hardware architecture, and each process has its own virtual address space.

Physical addresses
: The addresses used between the processor and the system's memory. Physical addresses are 32- or 64-bit quantities; even 32-bit systems can use 64-bit physical addresses in some situations.

Bus addresses
: The addresses used between peripheral buses and memory. Often they are the same as the physical addresses used by the processor, but that is not necessarily the case. Bus addresses are highly architecture dependent, of course.

Kernel logical addresses
: These make up the normal address space of the kernel. These addresses map most or all of main memory, and are often treated as if they were physical addresses. On most architectures, logical addresses and their associated physical addresses differ only by a constant offset. Logical addresses use the hardware's native pointer size, and thus may be unable to address all of physical memory on heavily equipped 32-bit systems. Logical addresses are usually stored in variables of type `unsigned long` or `void *`. Memory returned from *kmalloc* has a logical address.

Kernel virtual addresses
: These differ from logical addresses in that they do not necessarily have a direct mapping to physical addresses. All logical addresses are kernel virtual addresses; memory allocated by *vmalloc* also has a virtual address (but no direct physical mapping). The function *kmap*, described later in this chapter, also returns virtual addresses. Virtual addresses are usually stored in pointer variables.

If you have a logical address, the macro *__pa()* (defined in `<asm/page.h>`) will return its associated physical address. Physical addresses can be mapped back to logical addresses with *__va()*, but only for low-memory pages.

Different kernel functions require different types of addresses. It would be nice if there were different C types defined so that the required address type were explicit, but we have no such luck. In this chapter, we will be clear on which types of addresses are used where.

High and Low Memory

The difference between logical and kernel virtual addresses is highlighted on 32-bit systems that are equipped with large amounts of memory. With 32 bits, it is possible to address 4 GB of memory. Linux on 32-bit systems has, until recently, been limited to substantially less memory than that, however, because of the way it sets up the virtual address space. The system was unable to handle more memory than it could set up logical addresses for, since it needed directly mapped kernel addresses for all memory.

Recent developments have eliminated the limitations on memory, and 32-bit systems can now work with well over 4 GB of system memory (assuming, of course, that the processor itself can address that much memory). The limitation on how much memory can be directly mapped with logical addresses remains, however. Only the lowest portion of memory (up to 1 or 2 GB, depending on the hardware and the kernel configuration) has logical addresses; the rest (high memory) does not. High memory can require 64-bit physical addresses, and the kernel must set up explicit virtual address mappings to manipulate it. Thus, many kernel functions are limited to low memory only; high memory tends to be reserved for user-space process pages.

The term "high memory" can be confusing to some, especially since it has other meanings in the PC world. So, to make things clear, we'll define the terms here:

Low memory
: Memory for which logical addresses exist in kernel space. On almost every system you will likely encounter, all memory is low memory.

High memory
: Memory for which logical addresses do not exist, because the system contains more physical memory than can be addressed with 32 bits.

On i386 systems, the boundary between low and high memory is usually set at just under 1 GB. This boundary is not related in any way to the old 640 KB limit found on the original PC. It is, instead, a limit set by the kernel itself as it splits the 32-bit address space between kernel and user space.

We will point out high-memory limitations as we come to them in this chapter.

The Memory Map and struct page

Historically, the kernel has used logical addresses to refer to explicit pages of memory. The addition of high-memory support, however, has exposed an obvious problem with that approach—logical addresses are not available for high memory. Thus kernel functions that deal with memory are increasingly using pointers to `struct page` instead. This data structure is used to keep track of just about everything the kernel needs to know about physical memory; there is one `struct page` for each physical page on the system. Some of the fields of this structure include the following:

`atomic_t count;`
: The number of references there are to this page. When the count drops to zero, the page is returned to the free list.

`wait_queue_head_t wait;`
: A list of processes waiting on this page. Processes can wait on a page when a kernel function has locked it for some reason; drivers need not normally worry about waiting on pages, though.

`void *virtual;`
: The kernel virtual address of the page, if it is mapped; `NULL`, otherwise. Low-memory pages are always mapped; high-memory pages usually are not.

`unsigned long flags;`
: A set of bit flags describing the status of the page. These include `PG_locked`, which indicates that the page has been locked in memory, and `PG_reserved`, which prevents the memory management system from working with the page at all.

There is much more information within `struct page`, but it is part of the deeper black magic of memory management and is not of concern to driver writers.

The kernel maintains one or more arrays of `struct page` entries, which track all of the physical memory on the system. On most systems, there is a single array, called `mem_map`. On some systems, however, the situation is more complicated. Nonuniform memory access (NUMA) systems and those with widely discontiguous physical memory may have more than one memory map array, so code that is meant to be portable should avoid direct access to the array whenever possible. Fortunately, it is usually quite easy to just work with `struct page` pointers without worrying about where they come from.

Some functions and macros are defined for translating between `struct page` pointers and virtual addresses:

`struct page *virt_to_page(void *kaddr);`
: This macro, defined in `<asm/page.h>`, takes a kernel logical address and returns its associated `struct page` pointer. Since it requires a logical address, it will not work with memory from *vmalloc* or high memory.

`void *page_address(struct page *page);`
: Returns the kernel virtual address of this page, if such an address exists. For high memory, that address exists only if the page has been mapped.

```
#include <linux/highmem.h>
void *kmap(struct page *page);
void kunmap(struct page *page);
```
: *kmap* returns a kernel virtual address for any page in the system. For low-memory pages, it just returns the logical address of the page; for high-memory pages, *kmap* creates a special mapping. Mappings created with *kmap* should always be freed with *kunmap*; a limited number of such mappings is available, so it is better not to hold on to them for too long. *kmap* calls are

additive, so if two or more functions both call *kmap* on the same page the right thing happens. Note also that *kmap* can sleep if no mappings are available.

We will see some uses of these functions when we get into the example code later in this chapter.

Page Tables

When a program looks up a virtual address, the CPU must convert the address to a physical address in order to access physical memory. The step is usually performed by splitting the address into bitfields. Each bitfield is used as an index into an array, called a *page table*, to retrieve either the address of the next table or the address of the physical page that holds the virtual address.

The Linux kernel manages three levels of page tables in order to map virtual addresses to physical addresses. The multiple levels allow the memory range to be sparsely populated; modern systems will spread a process out across a large range of virtual memory. It makes sense to do things that way; it allows for runtime flexibility in how things are laid out.

Note that Linux uses a three-level system even on hardware that only supports two levels of page tables or hardware that uses a different way to map virtual addresses to physical ones. The use of three levels in a processor-independent implementation allows Linux to support both two-level and three-level processors without clobbering the code with a lot of `#ifdef` statements. This kind of conservative coding doesn't lead to additional overhead when the kernel runs on two-level processors, because the compiler actually optimizes out the unused level.

It is time to take a look at the data structures used to implement the paging system. The following list summarizes the implementation of the three levels in Linux, and Figure 13-2 depicts them.

Page Directory (PGD)
: The top-level page table. The PGD is an array of `pgd_t` items, each of which points to a second-level page table. Each process has its own page directory, and there is one for kernel space as well. You can think of the page directory as a page-aligned array of `pgd_t`s.

Page mid-level Directory (PMD)
: The second-level table. The PMD is a page-aligned array of `pmd_t` items. A `pmd_t` is a pointer to the third-level page table. Two-level processors have no physical PMD; they declare their PMD as an array with a single element, whose value is the PMD itself—we'll see in a while how this is handled in C and how the compiler optimizes this level away.

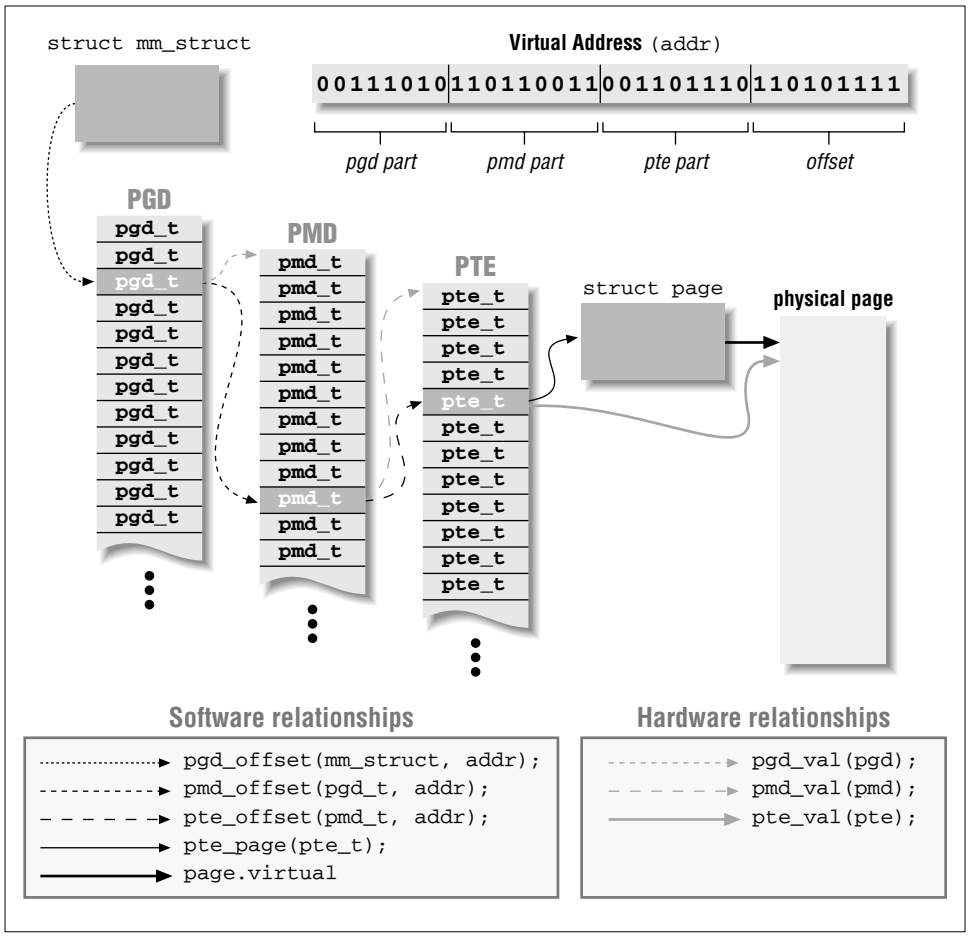

Figure 13-2. The three levels of Linux page tables

Page Table
: A page-aligned array of items, each of which is called a Page Table Entry. The kernel uses the `pte_t` type for the items. A `pte_t` contains the physical address of the data page.

The types introduced in this list are defined in `<asm/page.h>`, which must be included by every source file that plays with paging.

The kernel doesn't need to worry about doing page-table lookups during normal program execution, because they are done by the hardware. Nonetheless, the kernel must arrange things so that the hardware can do its work. It must build the page tables and look them up whenever the processor reports a page fault, that is,

whenever the page associated with a virtual address needed by the processor is not present in memory. Device drivers, too, must be able to build page tables and handle faults when implementing *mmap*.

It's interesting to note how software memory management exploits the same page tables that are used by the CPU itself. Whenever a CPU doesn't implement page tables, the difference is only hidden in the lowest levels of architecture-specific code. In Linux memory management, therefore, you always talk about three-level page tables irrespective of whether they are known to the hardware or not. An example of a CPU family that doesn't use page tables is the PowerPC. PowerPC designers implemented a hash algorithm that maps virtual addresses into a one-level page table. When accessing a page that is already in memory but whose physical address has expired from the CPU caches, the CPU needs to read memory only once, as opposed to the two or three accesses required by a multilevel page table approach. The hash algorithm, like multilevel tables, makes it possible to reduce use of memory in mapping virtual addresses to physical ones.

Irrespective of the mechanisms used by the CPU, the Linux software implementation is based on three-level page tables, and the following symbols are used to access them. Both `<asm/page.h>` and `<asm/pgtable.h>` must be included for all of them to be accessible.

`PTRS_PER_PGD`
`PTRS_PER_PMD`
`PTRS_PER_PTE`
: The size of each table. Two-level processors set `PTRS_PER_PMD` to 1, to avoid dealing with the middle level.

```
unsigned pgd_val(pgd_t pgd)
unsigned pmd_val(pmd_t pmd)
unsigned pte_val(pte_t pte)
```
: These three macros are used to retrieve the `unsigned` value from the typed data item. The actual type used varies depending on the underlying architecture and kernel configuration options; it is usually either `unsigned long` or, on 32-bit processors supporting high memory, `unsigned long long`. SPARC64 processors use `unsigned int`. The macros help in using strict data typing in source code without introducing computational overhead.

```
pgd_t * pgd_offset(struct mm_struct * mm, unsigned long
      address)
pmd_t * pmd_offset(pgd_t * dir, unsigned long address)
pte_t * pte_offset(pmd_t * dir, unsigned long address)
```
: These inline functions* are used to retrieve the `pgd`, `pmd`, and `pte` entries

* On 32-bit SPARC processors, the functions are not `inline` but rather real `extern` functions, which are not exported to modularized code. Therefore you won't be able to use these functions in a module running on the SPARC, but you won't usually need to.

associated with `address`. Page-table lookup begins with a pointer to `struct mm_struct`. The pointer associated with the memory map of the current process is `current->mm`, while the pointer to kernel space is described by `&init_mm`. Two-level processors define `pmd_offset(dir,add)` as `(pmd_t *)dir`, thus folding the `pmd` over the `pgd`. Functions that scan page tables are always declared as `inline`, and the compiler optimizes out any `pmd` lookup.

`struct page *pte_page(pte_t pte)`
: This function returns a pointer to the `struct page` entry for the page in this page-table entry. Code that deals with page-tables will generally want to use *pte_page* rather than *pte_val*, since *pte_page* deals with the processor-dependent format of the page-table entry and returns the `struct page` pointer, which is usually what's needed.

`pte_present(pte_t pte)`
: This macro returns a boolean value that indicates whether the data page is currently in memory. This is the most used of several functions that access the low bits in the `pte`—the bits that are discarded by *pte_page*. Pages may be absent, of course, if the kernel has swapped them to disk (or if they have never been loaded). The page tables themselves, however, are always present in the current Linux implementation. Keeping page tables in memory simplifies the kernel code because *pgd_offset* and friends never fail; on the other hand, even a process with a "resident storage size" of zero keeps its page tables in real RAM, wasting some memory that might be better used elsewhere.

Each process in the system has a `struct mm_struct` structure, which contains its page tables and a great many other things. It also contains a spinlock called `page_table_lock`, which should be held while traversing or modifying the page tables.

Just seeing the list of these functions is not enough for you to be proficient in the Linux memory management algorithms; real memory management is much more complex and must deal with other complications, like cache coherence. The previous list should nonetheless be sufficient to give you a feel for how page management is implemented; it is also about all that you will need to know, as a device driver writer, to work occasionally with page tables. You can get more information from the *include/asm* and *mm* subtrees of the kernel source.

Virtual Memory Areas

Although paging sits at the lowest level of memory management, something more is necessary before you can use the computer's resources efficiently. The kernel needs a higher-level mechanism to handle the way a process sees its memory. This mechanism is implemented in Linux by means of virtual memory areas, which are typically referred to as areas or VMAs.

An area is a homogeneous region in the virtual memory of a process, a contiguous range of addresses with the same permission flags. It corresponds loosely to the concept of a "segment," although it is better described as "a memory object with its own properties." The memory map of a process is made up of the following:

- An area for the program's executable code (often called text).
- One area each for data, including initialized data (that which has an explicitly assigned value at the beginning of execution), uninitialized data (BSS),* and the program stack.
- One area for each active memory mapping.

The memory areas of a process can be seen by looking in */proc/`pid`/maps* (where `pid`, of course, is replaced by a process ID). */proc/self* is a special case of */proc/pid*, because it always refers to the current process. As an example, here are a couple of memory maps, to which we have added short comments after a sharp sign:

```
morgana.root# cat /proc/1/maps   # look at init
08048000-0804e000 r-xp 00000000 08:01 51297      /sbin/init  # text
0804e000-08050000 rw-p 00005000 08:01 51297      /sbin/init  # data
08050000-08054000 rwxp 00000000 00:00 0          # zero-mapped bss
40000000-40013000 r-xp 00000000 08:01 39003      /lib/ld-2.1.3.so # text
40013000-40014000 rw-p 00012000 08:01 39003      /lib/ld-2.1.3.so # data
40014000-40015000 rw-p 00000000 00:00 0          # bss for ld.so
4001b000-40108000 r-xp 00000000 08:01 39006      /lib/libc-2.1.3.so # text
40108000-4010c000 rw-p 000ec000 08:01 39006      /lib/libc-2.1.3.so # data
4010c000-40110000 rw-p 00000000 00:00 0          # bss for libc.so
bfffe000-c0000000 rwxp fffff000 00:00 0          # zero-mapped stack

morgana.root# rsh wolf head /proc/self/maps  #### alpha-axp: static ecoff
000000011fffe000-0000000120000000 rwxp 0000000000000000 00:00 0     # stack
0000000120000000-0000000120014000 r-xp 0000000000000000 08:03 2844  # text
0000000140000000-0000000140002000 rwxp 0000000000014000 08:03 2844  # data
0000000140002000-0000000140008000 rwxp 0000000000000000 00:00 0     # bss
```

The fields in each line are as follows:

```
start-end perm offset major:minor inode image.
```

Each field in */proc/*/maps* (except the image name) corresponds to a field in `struct vm_area_struct`, and is described in the following list.

`start`
`end`
: The beginning and ending virtual addresses for this memory area.

---

* The name *BSS* is a historical relic, from an old assembly operator meaning "Block started by symbol." The BSS segment of executable files isn't stored on disk, and the kernel maps the zero page to the BSS address range.

`perm`
: A bit mask with the memory area's read, write, and execute permissions. This field describes what the process is allowed to do with pages belonging to the area. The last character in the field is either `p` for "private" or `s` for "shared."

`offset`
: Where the memory area begins in the file that it is mapped to. An offset of zero, of course, means that the first page of the memory area corresponds to the first page of the file.

`major`
`minor`
: The major and minor numbers of the device holding the file that has been mapped. Confusingly, for device mappings, the major and minor numbers refer to the disk partition holding the device special file that was opened by the user, and not the device itself.

`inode`
: The inode number of the mapped file.

`image`
: The name of the file (usually an executable image) that has been mapped.

A driver that implements the *mmap* method needs to fill a VMA structure in the address space of the process mapping the device. The driver writer should therefore have at least a minimal understanding of VMAs in order to use them.

Let's look at the most important fields in `struct vm_area_struct` (defined in `<linux/mm.h>`). These fields may be used by device drivers in their *mmap* implementation. Note that the kernel maintains lists and trees of VMAs to optimize area lookup, and several fields of `vm_area_struct` are used to maintain this organization. VMAs thus can't be created at will by a driver, or the structures will break. The main fields of VMAs are as follows (note the similarity between these fields and the */proc* output we just saw):

`unsigned long vm_start;`
`unsigned long vm_end;`
: The virtual address range covered by this VMA. These fields are the first two fields shown in */proc/*/maps*.

`struct file *vm_file;`
: A pointer to the `struct file` structure associated with this area (if any).

`unsigned long vm_pgoff;`
: The offset of the area in the file, in pages. When a file or device is mapped, this is the file position of the first page mapped in this area.

`unsigned long vm_flags;`
A set of flags describing this area. The flags of the most interest to device driver writers are `VM_IO` and `VM_RESERVED`. `VM_IO` marks a VMA as being a memory-mapped I/O region. Among other things, the `VM_IO` flag will prevent the region from being included in process core dumps. `VM_RESERVED` tells the memory management system not to attempt to swap out this VMA; it should be set in most device mappings.

`struct vm_operations_struct *vm_ops;`
A set of functions that the kernel may invoke to operate on this memory area. Its presence indicates that the memory area is a kernel "object" like the `struct file` we have been using throughout the book.

`void *vm_private_data;`
A field that may be used by the driver to store its own information.

Like `struct vm_area_struct`, the `vm_operations_struct` is defined in `<linux/mm.h>`; it includes the operations listed next. These operations are the only ones needed to handle the process's memory needs, and they are listed in the order they are declared. Later in this chapter, some of these functions will be implemented; they will be described more completely at that point.

```
void (*open)(struct vm_area_struct *vma);
```
The *open* method is called by the kernel to allow the subsystem implementing the VMA to initialize the area, adjust reference counts, and so forth. This method will be invoked any time that a new reference to the VMA is made (when a process forks, for example). The one exception happens when the VMA is first created by *mmap*; in this case, the driver's *mmap* method is called instead.

```
void (*close)(struct vm_area_struct *vma);
```
When an area is destroyed, the kernel calls its *close* operation. Note that there's no usage count associated with VMAs; the area is opened and closed exactly once by each process that uses it.

```
void (*unmap)(struct vm_area_struct *vma, unsigned long
      addr, size_t len);
```
The kernel calls this method to "unmap" part or all of an area. If the entire area is unmapped, then the kernel calls *vm_ops->close* as soon as *vm_ops->unmap* returns.

```
void (*protect)(struct vm_area_struct *vma, unsigned long,
      size_t, unsigned int newprot);
```
This method is intended to change the protection on a memory area, but is currently not used. Memory protection is handled by the page tables, and the kernel sets up the page-table entries separately.

```
int (*sync)(struct vm_area_struct *vma, unsigned long,
      size_t, unsigned int flags);
```
This method is called by the *msync* system call to save a dirty memory region to the storage medium. The return value is expected to be 0 to indicate success and negative if there was an error.

```
struct page *(*nopage)(struct vm_area_struct *vma, unsigned
      long address, int write_access);
```
When a process tries to access a page that belongs to a valid VMA, but that is currently not in memory, the *nopage* method is called (if it is defined) for the related area. The method returns the `struct page` pointer for the physical page, after, perhaps, having read it in from secondary storage. If the *nopage* method isn't defined for the area, an empty page is allocated by the kernel. The third argument, `write_access`, counts as "no-share": a nonzero value means the page must be owned by the current process, whereas 0 means that sharing is possible.

```
struct page *(*wppage)(struct vm_area_struct *vma, unsigned
      long address, struct page *page);
```
This method handles write-protected page faults but is currently unused. The kernel handles attempts to write over a protected page without invoking the area-specific callback. Write-protect faults are used to implement copy-on-write. A private page can be shared across processes until one process writes to it. When that happens, the page is cloned, and the process writes on its own copy of the page. If the whole area is marked as read-only, a `SIGSEGV` is sent to the process, and the copy-on-write is not performed.

```
int (*swapout)(struct page *page, struct file *file);
```
This method is called when a page is selected to be swapped out. A return value of 0 signals success; any other value signals an error. In case of error, the process owning the page is sent a `SIGBUS`. It is highly unlikely that a driver will ever need to implement *swapout*; device mappings are not something that the kernel can just write to disk.

That concludes our overview of Linux memory management data structures. With that out of the way, we can now proceed to the implementation of the *mmap* system call.

The mmap Device Operation

Memory mapping is one of the most interesting features of modern Unix systems. As far as drivers are concerned, memory mapping can be used to provide user programs with direct access to device memory.

A definitive example of *mmap* usage can be seen by looking at a subset of the virtual memory areas for the X Window System server:

```
cat /proc/731/maps
08048000-08327000 r-xp 00000000 08:01 55505    /usr/X11R6/bin/XF86_SVGA
08327000-08369000 rw-p 002de000 08:01 55505    /usr/X11R6/bin/XF86_SVGA
40015000-40019000 rw-s fe2fc000 08:01 10778    /dev/mem
40131000-40141000 rw-s 000a0000 08:01 10778    /dev/mem
40141000-40941000 rw-s f4000000 08:01 10778    /dev/mem
     ...
```

The full list of the X server's VMAs is lengthy, but most of the entries are not of interest here. We do see, however, three separate mappings of */dev/mem*, which give some insight into how the X server works with the video card. The first mapping shows a 16 KB region mapped at `fe2fc000`. This address is far above the highest RAM address on the system; it is, instead, a region of memory on a PCI peripheral (the video card). It will be a control region for that card. The middle mapping is at `a0000`, which is the standard location for video RAM in the 640 KB ISA hole. The last */dev/mem* mapping is a rather larger one at `f4000000` and is the video memory itself. These regions can also be seen in */proc/iomem*:

```
000a0000-000bffff : Video RAM area
f4000000-f4ffffff : Matrox Graphics, Inc. MGA G200 AGP
fe2fc000-fe2fffff : Matrox Graphics, Inc. MGA G200 AGP
```

Mapping a device means associating a range of user-space addresses to device memory. Whenever the program reads or writes in the assigned address range, it is actually accessing the device. In the X server example, using *mmap* allows quick and easy access to the video card's memory. For a performance-critical application like this, direct access makes a large difference.

As you might suspect, not every device lends itself to the *mmap* abstraction; it makes no sense, for instance, for serial ports and other stream-oriented devices. Another limitation of *mmap* is that mapping is `PAGE_SIZE` grained. The kernel can dispose of virtual addresses only at the level of page tables; therefore, the mapped area must be a multiple of `PAGE_SIZE` and must live in physical memory starting at an address that is a multiple of `PAGE_SIZE`. The kernel accommodates for size granularity by making a region slightly bigger if its size isn't a multiple of the page size.

These limits are not a big constraint for drivers, because the program accessing the device is device dependent anyway. It needs to know how to make sense of the memory region being mapped, so the `PAGE_SIZE` alignment is not a problem. A bigger constraint exists when ISA devices are used on some non-x86 platforms, because their hardware view of ISA may not be contiguous. For example, some Alpha computers see ISA memory as a scattered set of 8-bit, 16-bit, or 32-bit items, with no direct mapping. In such cases, you can't use *mmap* at all. The inability to perform direct mapping of ISA addresses to Alpha addresses is due to the incompatible data transfer specifications of the two systems. Whereas early Alpha processors could issue only 32-bit and 64-bit memory accesses, ISA can do only 8-bit and 16-bit transfers, and there's no way to transparently map one protocol onto the other.

There are sound advantages to using *mmap* when it's feasible to do so. For instance, we have already looked at the X server, which transfers a lot of data to and from video memory; mapping the graphic display to user space dramatically improves the throughput, as opposed to an *lseek/write* implementation. Another typical example is a program controlling a PCI device. Most PCI peripherals map their control registers to a memory address, and a demanding application might prefer to have direct access to the registers instead of repeatedly having to call *ioctl* to get its work done.

The *mmap* method is part of the `file_operations` structure and is invoked when the *mmap* system call is issued. With *mmap*, the kernel performs a good deal of work before the actual method is invoked, and therefore the prototype of the method is quite different from that of the system call. This is unlike calls such as *ioctl* and *poll*, where the kernel does not do much before calling the method.

The system call is declared as follows (as described in the *mmap(2)* manual page):

```
mmap (caddr_t addr, size_t len, int prot, int flags, int fd,
off_t offset)
```

On the other hand, the file operation is declared as

```
int (*mmap) (struct file *filp, struct vm_area_struct *vma);
```

The `filp` argument in the method is the same as that introduced in Chapter 3, while `vma` contains the information about the virtual address range that is used to access the device. Much of the work has thus been done by the kernel; to implement *mmap*, the driver only has to build suitable page tables for the address range and, if necessary, replace `vma->vm_ops` with a new set of operations.

There are two ways of building the page tables: doing it all at once with a function called *remap_page_range*, or doing it a page at a time via the *nopage* VMA method. Both methods have their advantages. We'll start with the "all at once" approach, which is simpler. From there we will start adding the complications needed for a real-world implementation.

Using remap_page_range

The job of building new page tables to map a range of physical addresses is handled by *remap_page_range*, which has the following prototype:

```
int remap_page_range(unsigned long virt_add, unsigned long phys_add,
                     unsigned long size, pgprot_t prot);
```

The value returned by the function is the usual 0 or a negative error code. Let's look at the exact meaning of the function's arguments:

`virt_add`
: The user virtual address where remapping should begin. The function builds page tables for the virtual address range between `virt_add` and `virt_add+size`.

`phys_add`
: The physical address to which the virtual address should be mapped. The function affects physical addresses from `phys_add` to `phys_add+size`.

`size`
: The dimension, in bytes, of the area being remapped.

`prot`
: The "protection" requested for the new VMA. The driver can (and should) use the value found in `vma->vm_page_prot`.

The arguments to *remap_page_range* are fairly straightforward, and most of them are already provided to you in the VMA when your *mmap* method is called. The one complication has to do with caching: usually, references to device memory should not be cached by the processor. Often the system BIOS will set things up properly, but it is also possible to disable caching of specific VMAs via the protection field. Unfortunately, disabling caching at this level is highly processor dependent. The curious reader may wish to look at the function *pgprot_noncached* from *drivers/char/mem.c* to see what's involved. We won't discuss the topic further here.

A Simple Implementation

If your driver needs to do a simple, linear mapping of device memory into a user address space, *remap_page_range* is almost all you really need to do the job. The following code comes from *drivers/char/mem.c* and shows how this task is performed in a typical module called *simple* (Simple Implementation Mapping Pages with Little Enthusiasm):

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

int simple_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

    if (offset >= _ _pa(high_memory) || (filp->f_flags & O_SYNC))
        vma->vm_flags |= VM_IO;
    vma->vm_flags |= VM_RESERVED;

    if (remap_page_range(vma->vm_start, offset,
           vma->vm_end-vma->vm_start, vma->vm_page_prot))
        return -EAGAIN;
    return 0;
}
```

The */dev/mem* code checks to see if the requested offset (stored in `vma->vm_pgoff`) is beyond physical memory; if so, the `VM_IO` VMA flag is set to mark the area as being I/O memory. The `VM_RESERVED` flag is always set to keep the system from trying to swap this area out. Then it is just a matter of calling *remap_page_range* to create the necessary page tables.

Adding VMA Operations

As we have seen, the `vm_area_struct` structure contains a set of operations that may be applied to the VMA. Now we'll look at providing those operations in a simple way; a more detailed example will follow later on.

Here, we will provide *open* and *close* operations for our VMA. These operations will be called anytime a process opens or closes the VMA; in particular, the *open* method will be invoked anytime a process forks and creates a new reference to the VMA. The *open* and *close* VMA methods are called in addition to the processing performed by the kernel, so they need not reimplement any of the work done there. They exist as a way for drivers to do any additional processing that they may require.

We'll use these methods to increment the module usage count whenever the VMA is opened, and to decrement it when it's closed. In modern kernels, this work is not strictly necessary; the kernel will not call the driver's *release* method as long as a VMA remains open, so the usage count will not drop to zero until all references to the VMA are closed. The 2.0 kernel, however, did not perform this tracking, so portable code will still want to be able to maintain the usage count.

So, we will override the default `vma->vm_ops` with operations that keep track of the usage count. The code is quite simple—a complete *mmap* implementation for a modularized */dev/mem* looks like the following:

```
void simple_vma_open(struct vm_area_struct *vma)
{ MOD_INC_USE_COUNT; }

void simple_vma_close(struct vm_area_struct *vma)
{ MOD_DEC_USE_COUNT; }

static struct vm_operations_struct simple_remap_vm_ops = {
    open:  simple_vma_open,
    close: simple_vma_close,
};

int simple_remap_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long offset = VMA_OFFSET(vma);

    if (offset >= __pa(high_memory) || (filp->f_flags & O_SYNC))
        vma->vm_flags |= VM_IO;
    vma->vm_flags |= VM_RESERVED;
```

```
    if (remap_page_range(vma->vm_start, offset, vma->vm_end-vma->vm_start,
                vma->vm_page_prot))
        return -EAGAIN;

    vma->vm_ops = &simple_remap_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

This code relies on the fact that the kernel initializes to `NULL` the `vm_ops` field in the newly created area before calling `f_op->mmap`. The code just shown checks the current value of the pointer as a safety measure, should something change in future kernels.

The strange `VMA_OFFSET` macro that appears in this code is used to hide a difference in the `vma` structure across kernel versions. Since the offset is a number of pages in 2.4 and a number of bytes in 2.2 and earlier kernels, `<sysdep.h>` declares the macro to make the difference transparent (and the result is expressed in bytes).

Mapping Memory with nopage

Although *remap_page_range* works well for many, if not most, driver *mmap* implementations, sometimes it is necessary to be a little more flexible. In such situations, an implementation using the *nopage* VMA method may be called for.

The *nopage* method, remember, has the following prototype:

```
struct page (*nopage)(struct vm_area_struct *vma,
                      unsigned long address, int write_access);
```

When a user process attempts to access a page in a VMA that is not present in memory, the associated *nopage* function is called. The `address` parameter will contain the virtual address that caused the fault, rounded down to the beginning of the page. The *nopage* function must locate and return the `struct page` pointer that refers to the page the user wanted. This function must also take care to increment the usage count for the page it returns by calling the *get_page* macro:

```
get_page(struct page *pageptr);
```

This step is necessary to keep the reference counts correct on the mapped pages. The kernel maintains this count for every page; when the count goes to zero, the kernel knows that the page may be placed on the free list. When a VMA is unmapped, the kernel will decrement the usage count for every page in the area. If your driver does not increment the count when adding a page to the area, the usage count will become zero prematurely and the integrity of the system will be compromised.

One situation in which the *nopage* approach is useful can be brought about by the *mremap* system call, which is used by applications to change the bounding addresses of a mapped region. If the driver wants to be able to deal with *mremap*, the previous implementation won't work correctly, because there's no way for the driver to know that the mapped region has changed.

The Linux implementation of *mremap* doesn't notify the driver of changes in the mapped area. Actually, it *does* notify the driver if the size of the area is reduced via the *unmap* method, but no callback is issued if the area increases in size.

The basic idea behind notifying the driver of a reduction is that the driver (or the filesystem mapping a regular file to memory) needs to know when a region is unmapped in order to take the proper action, such as flushing pages to disk. Growth of the mapped region, on the other hand, isn't really meaningful for the driver until the program invoking *mremap* accesses the new virtual addresses. In real life, it's quite common to map regions that are never used (unused sections of program code, for example). The Linux kernel, therefore, doesn't notify the driver if the mapped region grows, because the *nopage* method will take care of pages one at a time as they are actually accessed.

In other words, the driver isn't notified when a mapping grows because *nopage* will do it later, without having to use memory before it is actually needed. This optimization is mostly aimed at regular files, whose mapping uses real RAM.

The *nopage* method, therefore, must be implemented if you want to support the *mremap* system call. But once you have *nopage*, you can choose to use it extensively, with some limitations (described later). This method is shown in the next code fragment. In this implementation of *mmap*, the device method only replaces `vma->vm_ops`. The *nopage* method takes care of "remapping" one page at a time and returning the address of its `struct page` structure. Because we are just implementing a window onto physical memory here, the remapping step is simple—we need only locate and return a pointer to the `struct page` for the desired address.

An implementation of */dev/mem* using *nopage* looks like the following:

```
struct page *simple_vma_nopage(struct vm_area_struct *vma,
                unsigned long address, int write_access)
{
    struct page *pageptr;
    unsigned long physaddr = address - vma->vm_start + VMA_OFFSET(vma);
    pageptr = virt_to_page(__va(physaddr));
    get_page(pageptr);
    return pageptr;
}

int simple_nopage_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long offset = VMA_OFFSET(vma);
```

```
        if (offset >= __pa(high_memory) || (filp->f_flags & O_SYNC))
            vma->vm_flags |= VM_IO;
        vma->vm_flags |= VM_RESERVED;

        vma->vm_ops = &simple_nopage_vm_ops;
        simple_vma_open(vma);
        return 0;
    }
```

Since, once again, we are simply mapping main memory here, the *nopage* function need only find the correct `struct page` for the faulting address and increment its reference count. The required sequence of events is thus to calculate the desired physical address, turn it into a logical address with *__va*, and then finally to turn it into a `struct page` with *virt_to_page*. It would be possible, in general, to go directly from the physical address to the `struct page`, but such code would be difficult to make portable across architectures. Such code might be necessary, however, if one were trying to map high memory, which, remember, has no logical addresses. *simple*, being simple, does not worry about that (rare) case.

If the *nopage* method is left `NULL`, kernel code that handles page faults maps the zero page to the faulting virtual address. The zero page is a copy-on-write page that reads as zero and that is used, for example, to map the BSS segment. Therefore, if a process extends a mapped region by calling *mremap*, and the driver hasn't implemented *nopage*, it will end up with zero pages instead of a segmentation fault.

The *nopage* method normally returns a pointer to a `struct page`. If, for some reason, a normal page cannot be returned (e.g., the requested address is beyond the device's memory region), `NOPAGE_SIGBUS` can be returned to signal the error. *nopage* can also return `NOPAGE_OOM` to indicate failures caused by resource limitations.

Note that this implementation will work for ISA memory regions but not for those on the PCI bus. PCI memory is mapped above the highest system memory, and there are no entries in the system memory map for those addresses. Because there is thus no `struct page` to return a pointer to, *nopage* cannot be used in these situations; you must, instead, use *remap_page_range*.

Remapping Specific I/O Regions

All the examples we've seen so far are reimplementations of */dev/mem*; they remap physical addresses into user space. The typical driver, however, wants to map only the small address range that applies to its peripheral device, not all of memory. In order to map to user space only a subset of the whole memory range, the driver needs only to play with the offsets. The following lines will do the trick for a driver mapping a region of `simple_region_size` bytes, beginning at physical address `simple_region_start` (which should be page aligned).

```
unsigned long off = vma->vm_pgoff << PAGE_SHIFT;
unsigned long physical = simple_region_start + off;
unsigned long vsize = vma->vm_end - vma->vm_start;
unsigned long psize = simple_region_size - off;

if (vsize > psize)
    return -EINVAL; /*  spans too high */
remap_page_range(vma_>vm_start, physical, vsize, vma->vm_page_prot);
```

In addition to calculating the offsets, this code introduces a check that reports an error when the program tries to map more memory than is available in the I/O region of the target device. In this code, `psize` is the physical I/O size that is left after the offset has been specified, and `vsize` is the requested size of virtual memory; the function refuses to map addresses that extend beyond the allowed memory range.

Note that the user process can always use *mremap* to extend its mapping, possibly past the end of the physical device area. If your driver has no *nopage* method, it will never be notified of this extension, and the additional area will map to the zero page. As a driver writer, you may well want to prevent this sort of behavior; mapping the zero page onto the end of your region is not an explicitly bad thing to do, but it is highly unlikely that the programmer wanted that to happen.

The simplest way to prevent extension of the mapping is to implement a simple *nopage* method that always causes a bus signal to be sent to the faulting process. Such a method would look like this:

```
struct page *simple_nopage(struct vm_area_struct *vma,
                           unsigned long address, int write_access);
{ return NOPAGE_SIGBUS; /* send a SIGBUS */}
```

Remapping RAM

Of course, a more thorough implementation could check to see if the faulting address is within the device area, and perform the remapping if that is the case. Once again, however, *nopage* will not work with PCI memory areas, so extension of PCI mappings is not possible. In Linux, a page of physical addresses is marked as "reserved" in the memory map to indicate that it is not available for memory management. On the PC, for example, the range between 640 KB and 1 MB is marked as reserved, as are the pages that host the kernel code itself.

An interesting limitation of *remap_page_range* is that it gives access only to reserved pages and physical addresses above the top of physical memory. Reserved pages are locked in memory and are the only ones that can be safely mapped to user space; this limitation is a basic requirement for system stability.

Therefore, *remap_page_range* won't allow you to remap conventional addresses—which include the ones you obtain by calling *get_free_page*. Instead, it will map in the zero page. Nonetheless, the function does everything that most hardware drivers need it to, because it can remap high PCI buffers and ISA memory.

The limitations of *remap_page_range* can be seen by running *mapper*, one of the sample programs in *misc-progs* in the files provided on the O'Reilly FTP site. *mapper* is a simple tool that can be used to quickly test the *mmap* system call; it maps read-only parts of a file based on the command-line options and dumps the mapped region to standard output. The following session, for instance, shows that */dev/mem* doesn't map the physical page located at address 64 KB—instead we see a page full of zeros (the host computer in this examples is a PC, but the result would be the same on other platforms):

```
morgana.root# ./mapper /dev/mem 0x10000 0x1000 | od -Ax -t x1
mapped "/dev/mem" from 65536 to 69632
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
001000
```

The inability of *remap_page_range* to deal with RAM suggests that a device like *scullp* can't easily implement *mmap*, because its device memory is conventional RAM, not I/O memory. Fortunately, a relatively easy workaround is available to any driver that needs to map RAM into user space; it uses the *nopage* method that we have seen earlier.

Remapping RAM with the nopage method

The way to map real RAM to user space is to use `vm_ops->nopage` to deal with page faults one at a time. A sample implementation is part of the *scullp* module, introduced in Chapter 7.

scullp is the page oriented char device. Because it is page oriented, it can implement *mmap* on its memory. The code implementing memory mapping uses some of the concepts introduced earlier in "Memory Management in Linux."

Before examining the code, let's look at the design choices that affect the *mmap* implementation in *scullp*.

- *scullp* doesn't release device memory as long as the device is mapped. This is a matter of policy rather than a requirement, and it is different from the behavior of *scull* and similar devices, which are truncated to a length of zero when opened for writing. Refusing to free a mapped *scullp* device allows a process to overwrite regions actively mapped by another process, so you can test and see how processes and device memory interact. To avoid releasing a mapped device, the driver must keep a count of active mappings; the `vmas` field in the device structure is used for this purpose.

- Memory mapping is performed only when the *scullp* `order` parameter is 0. The parameter controls how *get_free_pages* is invoked (see Chapter 7, "get_free_page and Friends"). This choice is dictated by the internals of *get_free_pages*, the allocation engine exploited by *scullp*. To maximize allocation performance, the Linux kernel maintains a list of free pages for each allocation order, and only the page count of the first page in a cluster is incremented by *get_free_pages* and decremented by *free_pages*. The *mmap* method is disabled for a *scullp* device if the allocation order is greater than zero, because *nopage* deals with single pages rather than clusters of pages. (Return to "A scull Using Whole Pages: scullp" in Chapter 7 if you need a refresher on *scullp* and the memory allocation order value.)

The last choice is mostly intended to keep the code simple. It *is* possible to correctly implement *mmap* for multipage allocations by playing with the usage count of the pages, but it would only add to the complexity of the example without introducing any interesting information.

Code that is intended to map RAM according to the rules just outlined needs to implement *open*, *close*, and *nopage*; it also needs to access the memory map to adjust the page usage counts.

This implementation of *scullp_mmap* is very short, because it relies on the *nopage* function to do all the interesting work:

```
int scullp_mmap(struct file *filp, struct vm_area_struct *vma)
{
    struct inode *inode = INODE_FROM_F(filp);

    /* refuse to map if order is not 0 */
    if (scullp_devices[MINOR(inode->i_rdev)].order)
        return -ENODEV;

    /* don't do anything here: "nopage" will fill the holes */
    vma->vm_ops = &scullp_vm_ops;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_private_data = scullp_devices + MINOR(inode->i_rdev);
    scullp_vma_open(vma);
    return 0;
}
```

The purpose of the leading conditional is to avoid mapping devices whose allocation order is not 0. *scullp*'s operations are stored in the `vm_ops` field, and a pointer to the device structure is stashed in the `vm_private_data` field. At the end, `vm_ops->open` is called to update the usage count for the module and the count of active mappings for the device.

open and *close* simply keep track of these counts and are defined as follows:

```
void scullp_vma_open(struct vm_area_struct *vma)
{
    ScullP_Dev *dev = scullp_vma_to_dev(vma);

    dev->vmas++;
    MOD_INC_USE_COUNT;
}

void scullp_vma_close(struct vm_area_struct *vma)
{
    ScullP_Dev *dev = scullp_vma_to_dev(vma);

    dev->vmas--;
    MOD_DEC_USE_COUNT;
}
```

The function *sculls_vma_to_dev* simply returns the contents of the `vm_private_data` field. It exists as a separate function because kernel versions prior to 2.4 lacked that field, requiring that other means be used to get that pointer. See "Backward Compatibility" at the end of this chapter for details.

Most of the work is then performed by *nopage*. In the *scullp* implementation, the `address` parameter to *nopage* is used to calculate an offset into the device; the offset is then used to look up the correct page in the *scullp* memory tree.

```
struct page *scullp_vma_nopage(struct vm_area_struct *vma,
                               unsigned long address, int write)
{
    unsigned long offset;
    ScullP_Dev *ptr, *dev = scullp_vma_to_dev(vma);
    struct page *page = NOPAGE_SIGBUS;
    void *pageptr = NULL; /* default to "missing" */

    down(&dev->sem);
    offset = (address - vma->vm_start) + VMA_OFFSET(vma);
    if (offset >= dev->size) goto out; /* out of range */

    /*
     * Now retrieve the scullp device from the list, then the page.
     * If the device has holes, the process receives a SIGBUS when
     * accessing the hole.
     */
    offset >>= PAGE_SHIFT; /* offset is a number of pages */
    for (ptr = dev; ptr && offset >= dev->qset;) {
        ptr = ptr->next;
        offset -= dev->qset;
    }
    if (ptr && ptr->data) pageptr = ptr->data[offset];
    if (!pageptr) goto out; /* hole or end-of-file */
    page = virt_to_page(pageptr);

    /* got it, now increment the count */
```

```
        get_page(page);
    out:
        up(&dev->sem);
        return page;
    }
```

scullp uses memory obtained with *get_free_pages*. That memory is addressed using logical addresses, so all *scullp_nopage* has to do to get a `struct page` pointer is to call *virt_to_page*.

The *scullp* device now works as expected, as you can see in this sample output from the *mapper* utility. Here we send a directory listing of */dev* (which is long) to the *scullp* device, and then use the *mapper* utility to look at pieces of that listing with *mmap*.

```
morgana% ls -l /dev > /dev/scullp
morgana% ./mapper /dev/scullp 0 140
mapped "/dev/scullp" from 0 to 140
total 77
-rwxr-xr-x    1 root     root        26689 Mar  2  2000 MAKEDEV
crw-rw-rw-    1 root     root      14,  14 Aug 10 20:55 admmidi0
morgana% ./mapper /dev/scullp 8192 200
mapped "/dev/scullp" from 8192 to 8392
0
crw----    1 root     root     113,   1 Mar 26  1999 cum1
crw----    1 root     root     113,   2 Mar 26  1999 cum2
crw----    1 root     root     113,   3 Mar 26  1999 cum3
```

Remapping Virtual Addresses

Although it's rarely necessary, it's interesting to see how a driver can map a virtual address to user space using *mmap*. A true virtual address, remember, is an address returned by a function like *vmalloc* or *kmap*—that is, a virtual address mapped in the kernel page tables. The code in this section is taken from *scullv*, which is the module that works like *scullp* but allocates its storage through *vmalloc*.

Most of the *scullv* implementation is like the one we've just seen for *scullp*, except that there is no need to check the `order` parameter that controls memory allocation. The reason for this is that *vmalloc* allocates its pages one at a time, because single-page allocations are far more likely to succeed than multipage allocations. Therefore, the allocation order problem doesn't apply to *vmalloc*ed space.

Most of the work of *vmalloc* is building page tables to access allocated pages as a continuous address range. The *nopage* method, instead, must pull the page tables back apart in order to return a `struct page` pointer to the caller. Therefore, the *nopage* implementation for *scullv* must scan the page tables to retrieve the page map entry associated with the page.

The function is similar to the one we saw for *scullp*, except at the end. This code excerpt only includes the part of *nopage* that differs from *scullp*:

```
pgd_t *pgd; pmd_t *pmd; pte_t *pte;
unsigned long lpage;

/*
 * After scullv lookup, "page" is now the address of the page
 * needed by the current process. Since it's a vmalloc address,
 * first retrieve the unsigned long value to be looked up
 * in page tables.
 */
lpage = VMALLOC_VMADDR(pageptr);
spin_lock(&init_mm.page_table_lock);
pgd = pgd_offset(&init_mm, lpage);
pmd = pmd_offset(pgd, lpage);
pte = pte_offset(pmd, lpage);
page = pte_page(*pte);
spin_unlock(&init_mm.page_table_lock);

/* got it, now increment the count */
get_page(page);
out:
up(&dev->sem);
return page;
```

The page tables are looked up using the functions introduced at the beginning of this chapter. The page directory used for this purpose is stored in the memory structure for kernel space, `init_mm`. Note that *scullv* obtains the `page_table_lock` prior to traversing the page tables. If that lock were not held, another processor could make a change to the page table while *scullv* was halfway through the lookup process, leading to erroneous results.

The macro `VMALLOC_VMADDR(pageptr)` returns the correct `unsigned long` value to be used in a page-table lookup from a *vmalloc* address. A simple cast of the value wouldn't work on the x86 with kernels older than 2.1, because of a glitch in memory management. Memory management for the x86 changed in version 2.1.1, and `VMALLOC_VMADDR` is now defined as the identity function, as it has always been for the other platforms. Its use is still suggested, however, as a way of writing portable code.

Based on this discussion, you might also want to map addresses returned by *ioremap* to user space. This mapping is easily accomplished because you can use *remap_page_range* directly, without implementing methods for virtual memory areas. In other words, *remap_page_range* is already usable for building new page tables that map I/O memory to user space; there's no need to look in the kernel page tables built by *vremap* as we did in *scullv*.

The kiobuf Interface

As of version 2.3.12, the Linux kernel supports an I/O abstraction called the *kernel I/O buffer*, or `kiobuf`. The kiobuf interface is intended to hide much of the complexity of the virtual memory system from device drivers (and other parts of the system that do I/O). Many features are planned for kiobufs, but their primary use in the 2.4 kernel is to facilitate the mapping of user-space buffers into the kernel.

The kiobuf Structure

Any code that works with kiobufs must include `<linux/iobuf.h>`. This file defines `struct kiobuf`, which is the heart of the kiobuf interface. This structure describes an array of pages that make up an I/O operation; its fields include the following:

`int nr_pages;`
The number of pages in this kiobuf

`int length;`
The number of bytes of data in the buffer

`int offset;`
The offset to the first valid byte in the buffer

`struct page **maplist;`
An array of `page` structures, one for each page of data in the kiobuf

The key to the kiobuf interface is the `maplist` array. Functions that operate on pages stored in a kiobuf deal directly with the `page` structures—all of the virtual memory system overhead has been moved out of the way. This implementation allows drivers to function independent of the complexities of memory management, and in general simplifies life greatly.

Prior to use, a kiobuf must be initialized. It is rare to initialize a single kiobuf in isolation, but, if need be, this initialization can be performed with *kiobuf_init*:

```
void kiobuf_init(struct kiobuf *iobuf);
```

Usually kiobufs are allocated in groups as part of a *kernel I/O vector*, or *kiovec*. A kiovec can be allocated and initialized in one step with a call to *alloc_kiovec*:

```
int alloc_kiovec(int nr, struct kiobuf **iovec);
```

The return value is 0 or an error code, as usual. When your code has finished with the kiovec structure, it should, of course, return it to the system:

```
void free_kiovec(int nr, struct kiobuf **);
```

The kernel provides a pair of functions for locking and unlocking the pages mapped in a kiovec:

```
int lock_kiovec(int nr, struct kiobuf *iovec[], int wait);
int unlock_kiovec(int nr, struct kiobuf *iovec[]);
```

Locking a kiovec in this manner is unnecessary, however, for most applications of kiobufs seen in device drivers.

Mapping User-Space Buffers and Raw I/O

Unix systems have long provided a "raw" interface to some devices—block devices in particular—which performs I/O directly from a user-space buffer and avoids copying data through the kernel. In some cases much improved performance can be had in this manner, especially if the data being transferred will not be used again in the near future. For example, disk backups typically read a great deal of data from the disk exactly once, then forget about it. Running the backup via a raw interface will avoid filling the system buffer cache with useless data.

The Linux kernel has traditionally not provided a raw interface, for a number of reasons. As the system gains in popularity, however, more applications that expect to be able to do raw I/O (such as large database management systems) are being ported. So the 2.3 development series finally added raw I/O; the driving force behind the kiobuf interface was the need to provide this capability.

Raw I/O is not always the great performance boost that some people think it should be, and driver writers should not rush out to add the capability just because they can. The overhead of setting up a raw transfer can be significant, and the advantages of buffering data in the kernel are lost. For example, note that raw I/O operations almost always must be synchronous—the *write* system call cannot return until the operation is complete. Linux currently lacks the mechanisms that user programs need to be able to safely perform asynchronous raw I/O on a user buffer.

In this section, we add a raw I/O capability to the *sbull* sample block driver. When kiobufs are available, *sbull* actually registers two devices. The block *sbull* device was examined in detail in Chapter 12. What we didn't see in that chapter was a second, char device (called *sbullr*), which provides raw access to the RAM-disk device. Thus, */dev/sbull0* and */dev/sbullr0* access the same memory; the former using the traditional, buffered mode and the second providing raw access via the kiobuf mechanism.

It is worth noting that in Linux systems, there is no need for block drivers to provide this sort of interface. The raw device, in *drivers/char/raw.c*, provides this capability in an elegant, general way for all block devices. The block drivers need not even know they are doing raw I/O. The raw I/O code in *sbull* is essentially a simplification of the raw device code for demonstration purposes.

Raw I/O to a block device must always be sector aligned, and its length must be a multiple of the sector size. Other kinds of devices, such as tape drives, may not have the same constraints. *sbullr* behaves like a block device and enforces the alignment and length requirements. To that end, it defines a few symbols:

```
#  define SBULLR_SECTOR 512  /* insist on this */
#  define SBULLR_SECTOR_MASK (SBULLR_SECTOR - 1)
#  define SBULLR_SECTOR_SHIFT 9
```

The *sbullr* raw device will be registered only if the hard-sector size is equal to `SBULLR_SECTOR`. There is no real reason why a larger hard-sector size could not be supported, but it would complicate the sample code unnecessarily.

The *sbullr* implementation adds little to the existing *sbull* code. In particular, the *open* and *close* methods from *sbull* are used without modification. Since *sbullr* is a char device, however, it needs *read* and *write* methods. Both are defined to use a single transfer function as follows:

```
ssize_t sbullr_read(struct file *filp, char *buf, size_t size,
                    loff_t *off)
{
    Sbull_Dev *dev = sbull_devices +
                    MINOR(filp->f_dentry->d_inode->i_rdev);
    return sbullr_transfer(dev, buf, size, off, READ);
}

ssize_t sbullr_write(struct file *filp, const char *buf, size_t size,
                loff_t *off)
{
    Sbull_Dev *dev = sbull_devices +
                    MINOR(filp->f_dentry->d_inode->i_rdev);
    return sbullr_transfer(dev, (char *) buf, size, off, WRITE);
}
```

The *sbullr_transfer* function handles all of the setup and teardown work, while passing off the actual transfer of data to yet another function. It is written as follows:

```
static int sbullr_transfer (Sbull_Dev *dev, char *buf, size_t count,
                loff_t *offset, int rw)
{
    struct kiobuf *iobuf;
    int result;

    /* Only block alignment and size allowed */
    if ((*offset & SBULLR_SECTOR_MASK) || (count & SBULLR_SECTOR_MASK))
        return -EINVAL;
    if ((unsigned long) buf & SBULLR_SECTOR_MASK)
        return -EINVAL;

    /* Allocate an I/O vector */
    result = alloc_kiovec(1, &iobuf);
```

```
    if (result)
        return result;

    /* Map the user I/O buffer and do the I/O. */
    result = map_user_kiobuf(rw, iobuf, (unsigned long) buf, count);
    if (result) {
        free_kiovec(1, &iobuf);
        return result;
    }
    spin_lock(&dev->lock);
    result = sbullr_rw_iovec(dev, iobuf, rw,
                    *offset >> SBULLR_SECTOR_SHIFT,
                    count >> SBULLR_SECTOR_SHIFT);
    spin_unlock(&dev->lock);

    /* Clean up and return. */
    unmap_kiobuf(iobuf);
    free_kiovec(1, &iobuf);
    if (result > 0)
        *offset += result << SBULLR_SECTOR_SHIFT;
    return result << SBULLR_SECTOR_SHIFT;
}
```

After doing a couple of sanity checks, the code creates a kiovec (containing a single kiobuf) with *alloc_kiovec*. It then uses that kiovec to map in the user buffer by calling *map_user_kiobuf*:

```
int map_user_kiobuf(int rw, struct kiobuf *iobuf,
                    unsigned long address, size_t len);
```

The result of this call, if all goes well, is that the buffer at the given (user virtual) `address` with length `len` is mapped into the given `iobuf`. This operation can sleep, since it is possible that part of the user buffer will need to be faulted into memory.

A kiobuf that has been mapped in this manner must eventually be unmapped, of course, to keep the reference counts on the pages straight. This unmapping is accomplished, as can be seen in the code, by passing the kiobuf to *unmap_kiobuf*.

So far, we have seen how to prepare a kiobuf for I/O, but not how to actually perform that I/O. The last step involves going through each page in the kiobuf and doing the required transfers; in *sbullr*, this task is handled by *sbullr_rw_iovec*. Essentially, this function passes through each page, breaks it up into sector-sized pieces, and passes them to *sbull_transfer* via a fake `request` structure:

```
static int sbullr_rw_iovec(Sbull_Dev *dev, struct kiobuf *iobuf, int rw,
                int sector, int nsectors)
{
    struct request fakereq;
    struct page *page;
    int offset = iobuf->offset, ndone = 0, pageno, result;
```

```
        /* Perform I/O on each sector */
        fakereq.sector = sector;
        fakereq.current_nr_sectors = 1;
        fakereq.cmd = rw;

        for (pageno = 0; pageno < iobuf->nr_pages; pageno++) {
            page = iobuf->maplist[pageno];
            while (ndone < nsectors) {
                /* Fake up a request structure for the operation */
                fakereq.buffer = (void *) (kmap(page) + offset);
                result = sbull_transfer(dev, &fakereq);
                kunmap(page);
                if (result == 0)
                    return ndone;
                /* Move on to the next one */
                ndone++;
                fakereq.sector++;
                offset += SBULLR_SECTOR;
                if (offset >= PAGE_SIZE) {
                    offset = 0;
                    break;
                }
            }
        }
        return ndone;
    }
```

Here, the `nr_pages` member of the `kiobuf` structure tells us how many pages need to be transferred, and the `maplist` array gives us access to each page. Thus it is just a matter of stepping through them all. Note, however, that *kmap* is used to get a kernel virtual address for each page; in this way, the function will work even if the user buffer is in high memory.

Some quick tests copying data show that a copy to or from an *sbullr* device takes roughly two-thirds the system time as the same copy to the block *sbull* device. The savings is gained by avoiding the extra copy through the buffer cache. Note that if the same data is read several times over, that savings will evaporate—especially for a real hardware device. Raw device access is often not the best approach, but for some applications it can be a major improvement.

Although kiobufs remain controversial in the kernel development community, there is interest in using them in a wider range of contexts. There is, for example, a patch that implements Unix pipes with kiobufs—data is copied directly from one process's address space to the other with no buffering in the kernel at all. A patch also exists that makes it easy to use a kiobuf to map kernel virtual memory into a process's address space, thus eliminating the need for a *nopage* implementation as shown earlier.

Direct Memory Access and Bus Mastering

Direct memory access, or DMA, is the advanced topic that completes our overview of memory issues. DMA is the hardware mechanism that allows peripheral components to transfer their I/O data directly to and from main memory without the need for the system processor to be involved in the transfer. Use of this mechanism can greatly increase throughput to and from a device, because a great deal of computational overhead is eliminated.

To exploit the DMA capabilities of its hardware, the device driver needs to be able to correctly set up the DMA transfer and synchronize with the hardware. Unfortunately, because of its hardware nature, DMA is very system dependent. Each architecture has its own techniques to manage DMA transfers, and the programming interface is different for each. The kernel can't offer a unified interface, either, because a driver can't abstract too much from the underlying hardware mechanisms. Some steps have been made in that direction, however, in recent kernels.

This chapter concentrates mainly on the PCI bus, since it is currently the most popular peripheral bus available. Many of the concepts are more widely applicable, though. We also touch on how some other buses, such as ISA and SBus, handle DMA.

Overview of a DMA Data Transfer

Before introducing the programming details, let's review how a DMA transfer takes place, considering only input transfers to simplify the discussion.

Data transfer can be triggered in two ways: either the software asks for data (via a function such as *read*) or the hardware asynchronously pushes data to the system.

In the first case, the steps involved can be summarized as follows:

1. When a process calls *read*, the driver method allocates a DMA buffer and instructs the hardware to transfer its data. The process is put to sleep.
2. The hardware writes data to the DMA buffer and raises an interrupt when it's done.
3. The interrupt handler gets the input data, acknowledges the interrupt, and awakens the process, which is now able to read data.

The second case comes about when DMA is used asynchronously. This happens, for example, with data acquisition devices that go on pushing data even if nobody is reading them. In this case, the driver should maintain a buffer so that a subsequent *read* call will return all the accumulated data to user space. The steps involved in this kind of transfer are slightly different:

1. The hardware raises an interrupt to announce that new data has arrived.
2. The interrupt handler allocates a buffer and tells the hardware where to transfer its data.
3. The peripheral device writes the data to the buffer and raises another interrupt when it's done.
4. The handler dispatches the new data, wakes any relevant process, and takes care of housekeeping.

A variant of the asynchronous approach is often seen with network cards. These cards often expect to see a circular buffer (often called a *DMA ring buffer*) established in memory shared with the processor; each incoming packet is placed in the next available buffer in the ring, and an interrupt is signaled. The driver then passes the network packets to the rest of the kernel, and places a new DMA buffer in the ring.

The processing steps in all of these cases emphasize that efficient DMA handling relies on interrupt reporting. While it is possible to implement DMA with a polling driver, it wouldn't make sense, because a polling driver would waste the performance benefits that DMA offers over the easier processor-driven I/O.

Another relevant item introduced here is the DMA buffer. To exploit direct memory access, the device driver must be able to allocate one or more special buffers, suited to DMA. Note that many drivers allocate their buffers at initialization time and use them until shutdown—the word *allocate* in the previous lists therefore means "get hold of a previously allocated buffer."

Allocating the DMA Buffer

This section covers the allocation of DMA buffers at a low level; we will introduce a higher-level interface shortly, but it is still a good idea to understand the material presented here.

The main problem with the DMA buffer is that when it is bigger than one page, it must occupy contiguous pages in physical memory because the device transfers data using the ISA or PCI system bus, both of which carry physical addresses. It's interesting to note that this constraint doesn't apply to the SBus (see "SBus" in Chapter 15), which uses virtual addresses on the peripheral bus. Some architectures *can* also use virtual addresses on the PCI bus, but a portable driver cannot count on that capability.

Although DMA buffers can be allocated either at system boot or at runtime, modules can only allocate their buffers at runtime. Chapter 7 introduced these techniques: "Boot-Time Allocation" talked about allocation at system boot, while "The Real Story of kmalloc" and "get_free_page and Friends" described allocation at

runtime. Driver writers must take care to allocate the right kind of memory when it will be used for DMA operations—not all memory zones are suitable. In particular, high memory will not work for DMA on most systems—the peripherals simply cannot work with addresses that high.

Most devices on modern buses can handle 32-bit addresses, meaning that normal memory allocations will work just fine for them. Some PCI devices, however, fail to implement the full PCI standard and cannot work with 32-bit addresses. And ISA devices, of course, are limited to 16-bit addresses only.

For devices with this kind of limitation, memory should be allocated from the DMA zone by adding the `GFP_DMA` flag to the *kmalloc* or *get_free_pages* call. When this flag is present, only memory that can be addressed with 16 bits will be allocated.

Do-it-yourself allocation

We have seen how *get_free_pages* (and therefore *kmalloc*) can't return more than 128 KB (or, more generally, 32 pages) of consecutive memory space. But the request is prone to fail even when the allocated buffer is less than 128 KB, because system memory becomes fragmented over time.*

When the kernel cannot return the requested amount of memory, or when you need more than 128 KB (a common requirement for PCI frame grabbers, for example), an alternative to returning `-ENOMEM` is to allocate memory at boot time or reserve the top of physical RAM for your buffer. We described allocation at boot time in "Boot-Time Allocation" in Chapter 7, but it is not available to modules. Reserving the top of RAM is accomplished by passing a `mem=` argument to the kernel at boot time. For example, if you have 32 MB, the argument `mem=31M` keeps the kernel from using the top megabyte. Your module could later use the following code to gain access to such memory:

```
dmabuf = ioremap( 0x1F00000 /* 31M */, 0x100000 /* 1M */);
```

Actually, there is another way to allocate DMA space: perform aggressive allocation until you are able to get enough consecutive pages to make a buffer. We strongly discourage this allocation technique if there's any other way to achieve your goal. Aggressive allocation results in high machine load, and possibly in a system lockup if your aggressiveness isn't correctly tuned. On the other hand, sometimes there is no other way available.

In practice, the code invokes `kmalloc(GFP_ATOMIC)` until the call fails; it then waits until the kernel frees some pages, and then allocates everything once again.

---

* The word *fragmentation* is usually applied to disks, to express the idea that files are not stored consecutively on the magnetic medium. The same concept applies to memory, where each virtual address space gets scattered throughout physical RAM, and it becomes difficult to retrieve consecutive free pages when a DMA buffer is requested.

If you keep an eye on the pool of allocated pages, sooner or later you'll find that your DMA buffer of consecutive pages has appeared; at this point you can release every page but the selected buffer. This kind of behavior is rather risky, though, because it may lead to a deadlock. We suggest using a kernel timer to release every page in case allocation doesn't succeed before a timeout expires.

We're not going to show the code here, but you'll find it in *misc-modules/allocator.c*; the code is thoroughly commented and designed to be called by other modules. Unlike every other source accompanying this book, the allocator is covered by the GPL. The reason we decided to put the source under the GPL is that it is neither particularly beautiful nor particularly clever, and if someone is going to use it, we want to be sure that the source is released with the module.

Bus Addresses

A device driver using DMA has to talk to hardware connected to the interface bus, which uses physical addresses, whereas program code uses virtual addresses.

As a matter of fact, the situation is slightly more complicated than that. DMA-based hardware uses *bus*, rather than *physical*, addresses. Although ISA and PCI addresses are simply physical addresses on the PC, this is not true for every platform. Sometimes the interface bus is connected through bridge circuitry that maps I/O addresses to different physical addresses. Some systems even have a page-mapping scheme that can make arbitrary pages appear contiguous to the peripheral bus.

At the lowest level (again, we'll look at a higher-level solution shortly), the Linux kernel provides a portable solution by exporting the following functions, defined in `<asm/io.h>`:

```
unsigned long virt_to_bus(volatile void * address);
void * bus_to_virt(unsigned long address);
```

The *virt_to_bus* conversion must be used when the driver needs to send address information to an I/O device (such as an expansion board or the DMA controller), while *bus_to_virt* must be used when address information is received from hardware connected to the bus.

DMA on the PCI Bus

The 2.4 kernel includes a flexible mechanism that supports PCI DMA (also known as *bus mastering*). It handles the details of buffer allocation and can deal with setting up the bus hardware for multipage transfers on hardware that supports them. This code also takes care of situations in which a buffer lives in a non-DMA-capable zone of memory, though only on some platforms and at a computational cost (as we will see later).

The functions in this section require a `struct pci_dev` structure for your device. The details of setting up a PCI device are covered in Chapter 15. Note, however, that the routines described here can also be used with ISA devices; in that case, the `struct pci_dev` pointer should simply be passed in as `NULL`.

Drivers that use the following functions should include `<linux/pci.h>`.

Dealing with difficult hardware

The first question that must be answered before performing DMA is whether the given device is capable of such operation on the current host. Many PCI devices fail to implement the full 32-bit bus address space, often because they are modified versions of old ISA hardware. The Linux kernel will attempt to work with such devices, but it is not always possible.

The function *pci_dma_supported* should be called for any device that has addressing limitations:

```
int pci_dma_supported(struct pci_dev *pdev, dma_addr_t mask);
```

Here, `mask` is a simple bit mask describing which address bits the device can successfully use. If the return value is nonzero, DMA is possible, and your driver should set the `dma_mask` field in the PCI device structure to the mask value. For a device that can only handle 16-bit addresses, you might use a call like this:

```
if (pci_dma_supported (pdev, 0xffff))
    pdev->dma_mask = 0xffff;
else {
    card->use_dma = 0;   /* We'll have to live without DMA */
    printk (KERN_WARN, "mydev: DMA not supported\n");
}
```

As of kernel 2.4.3, a new function, *pci_set_dma_mask*, has been provided. This function has the following prototype:

```
int pci_set_dma_mask(struct pci_dev *pdev, dma_addr_t mask);
```

If DMA can be supported with the given mask, this function returns 0 and sets the `dma_mask` field; otherwise, `-EIO` is returned.

For devices that can handle 32-bit addresses, there is no need to call *pci_dma_supported*.

DMA mappings

A *DMA mapping* is a combination of allocating a DMA buffer and generating an address for that buffer that is accessible by the device. In many cases, getting that address involves a simple call to *virt_to_bus*; some hardware, however, requires that *mapping registers* be set up in the bus hardware as well. Mapping registers

are an equivalent of virtual memory for peripherals. On systems where these registers are used, peripherals have a relatively small, dedicated range of addresses to which they may perform DMA. Those addresses are remapped, via the mapping registers, into system RAM. Mapping registers have some nice features, including the ability to make several distributed pages appear contiguous in the device's address space. Not all architectures have mapping registers, however; in particular, the popular PC platform has no mapping registers.

Setting up a useful address for the device may also, in some cases, require the establishment of a *bounce buffer*. Bounce buffers are created when a driver attempts to perform DMA on an address that is not reachable by the peripheral device—a high-memory address, for example. Data is then copied to and from the bounce buffer as needed. Making code work properly with bounce buffers requires adherence to some rules, as we will see shortly.

The DMA mapping sets up a new type, `dma_addr_t`, to represent bus addresses. Variables of type `dma_addr_t` should be treated as opaque by the driver; the only allowable operations are to pass them to the DMA support routines and to the device itself.

The PCI code distinguishes between two types of DMA mappings, depending on how long the DMA buffer is expected to stay around:

Consistent DMA mappings
: These exist for the life of the driver. A consistently mapped buffer must be simultaneously available to both the CPU and the peripheral (other types of mappings, as we will see later, can be available only to one or the other at any given time). The buffer should also, if possible, not have caching issues that could cause one not to see updates made by the other.

Streaming DMA mappings
: These are set up for a single operation. Some architectures allow for significant optimizations when streaming mappings are used, as we will see, but these mappings also are subject to a stricter set of rules in how they may be accessed. The kernel developers recommend the use of streaming mappings over consistent mappings whenever possible. There are two reasons for this recommendation. The first is that, on systems that support them, each DMA mapping uses one or more mapping registers on the bus. Consistent mappings, which have a long lifetime, can monopolize these registers for a long time, even when they are not being used. The other reason is that, on some hardware, streaming mappings can be optimized in ways that are not available to consistent mappings.

The two mapping types must be manipulated in different ways; it's time to look at the details.

Setting up consistent DMA mappings

A driver can set up a consistent mapping with a call to *pci_alloc_consistent*:

```
void *pci_alloc_consistent(struct pci_dev *pdev, size_t size,
                           dma_addr_t *bus_addr);
```

This function handles both the allocation and the mapping of the buffer. The first two arguments are our PCI device structure and the size of the needed buffer. The function returns the result of the DMA mapping in two places. The return value is a kernel virtual address for the buffer, which may be used by the driver; the associated bus address, instead, is returned in `bus_addr`. Allocation is handled in this function so that the buffer will be placed in a location that works with DMA; usually the memory is just allocated with *get_free_pages* (but note that the size is in bytes, rather than an order value).

Most architectures that support PCI perform the allocation at the `GFP_ATOMIC` priority, and thus do not sleep. The ARM port, however, is an exception to this rule.

When the buffer is no longer needed (usually at module unload time), it should be returned to the system with *pci_free_consistent*:

```
void pci_free_consistent(struct pci_dev *pdev, size_t size,
                         void *cpu_addr, dma_handle_t bus_addr);
```

Note that this function requires that both the CPU address and the bus address be provided.

Setting up streaming DMA mappings

Streaming mappings have a more complicated interface than the consistent variety, for a number of reasons. These mappings expect to work with a buffer that has already been allocated by the driver, and thus have to deal with addresses that they did not choose. On some architectures, streaming mappings can also have multiple, discontiguous pages and multipart "scatter-gather" buffers.

When setting up a streaming mapping, you must tell the kernel in which direction the data will be moving. Some symbols have been defined for this purpose:

`PCI_DMA_TODEVICE`
`PCI_DMA_FROMDEVICE`
: These two symbols should be reasonably self-explanatory. If data is being sent to the device (in response, perhaps, to a *write* system call), `PCI_DMA_TODEVICE` should be used; data going to the CPU, instead, will be marked with `PCI_DMA_FROMDEVICE`.

`PCI_DMA_BIDIRECTIONAL`
: If data can move in either direction, use `PCI_DMA_BIDIRECTIONAL`.

`PCI_DMA_NONE`
: This symbol is provided only as a debugging aid. Attempts to use buffers with this "direction" will cause a kernel panic.

For a number of reasons that we will touch on shortly, it is important to pick the right value for the direction of a streaming DMA mapping. It may be tempting to just pick `PCI_DMA_BIDIRECTIONAL` at all times, but on some architectures there will be a performance penalty to pay for that choice.

When you have a single buffer to transfer, map it with *pci_map_single*:

```
dma_addr_t pci_map_single(struct pci_dev *pdev, void *buffer,
                          size_t size, int direction);
```

The return value is the bus address that you can pass to the device, or `NULL` if something goes wrong.

Once the transfer is complete, the mapping should be deleted with *pci_unmap_single*:

```
void pci_unmap_single(struct pci_dev *pdev, dma_addr_t bus_addr,
                      size_t size, int direction);
```

Here, the `size` and `direction` arguments must match those used to map the buffer.

There are some important rules that apply to streaming DMA mappings:

- The buffer must be used only for a transfer that matches the direction value given when it was mapped.
- Once a buffer has been mapped, it belongs to the device, not the processor. Until the buffer has been unmapped, the driver should not touch its contents in any way. Only after *pci_unmap_single* has been called is it safe for the driver to access the contents of the buffer (with one exception that we'll see shortly). Among other things, this rule implies that a buffer being written to a device cannot be mapped until it contains all the data to write.
- The buffer must not be unmapped while DMA is still active, or serious system instability is guaranteed.

You may be wondering why the driver can no longer work with a buffer once it has been mapped. There are actually two reasons why this rule makes sense. First, when a buffer is mapped for DMA, the kernel must ensure that all of the data in that buffer has actually been written to memory. It is likely that some data will remain in the processor's cache, and must be explicitly flushed. Data written to the buffer by the processor after the flush may not be visible to the device.

Second, consider what happens if the buffer to be mapped is in a region of memory that is not accessible to the device. Some architectures will simply fail in this case, but others will create a bounce buffer. The bounce buffer is just a separate region of memory that *is* accessible to the device. If a buffer is mapped with a direction of `PCI_DMA_TODEVICE`, and a bounce buffer is required, the contents of the original buffer will be copied as part of the mapping operation. Clearly, changes to the original buffer after the copy will not be seen by the device. Similarly, `PCI_DMA_FROMDEVICE` bounce buffers are copied back to the original buffer by *pci_unmap_single*; the data from the device is not present until that copy has been done.

Incidentally, bounce buffers are one reason why it is important to get the direction right. `PCI_DMA_BIDIRECTIONAL` bounce buffers are copied before and after the operation, which is often an unnecessary waste of CPU cycles.

Occasionally a driver will need to access the contents of a streaming DMA buffer without unmapping it. A call has been provided to make this possible:

```
void pci_sync_single(struct pci_dev *pdev, dma_handle_t bus_addr,
                     size_t size, int direction);
```

This function should be called *before* the processor accesses a `PCI_DMA_FROMDEVICE` buffer, and *after* an access to a `PCI_DMA_TODEVICE` buffer.

Scatter-gather mappings

Scatter-gather mappings are a special case of streaming DMA mappings. Suppose you have several buffers, all of which need to be transferred to or from the device. This situation can come about in several ways, including from a *readv* or *writev* system call, a clustered disk I/O request, or a list of pages in a mapped kernel I/O buffer. You could simply map each buffer in turn and perform the required operation, but there are advantages to mapping the whole list at once.

One reason is that some smart devices can accept a *scatterlist* of array pointers and lengths and transfer them all in one DMA operation; for example, "zero-copy" networking is easier if packets can be built in multiple pieces. Linux is likely to take much better advantage of such devices in the future. Another reason to map scatterlists as a whole is to take advantage of systems that have mapping registers in the bus hardware. On such systems, physically discontiguous pages can be assembled into a single, contiguous array from the device's point of view. This technique works only when the entries in the scatterlist are equal to the page size in length (except the first and last), but when it does work it can turn multiple operations into a single DMA and speed things up accordingly.

Finally, if a bounce buffer must be used, it makes sense to coalesce the entire list into a single buffer (since it is being copied anyway).

So now you're convinced that mapping of scatterlists is worthwhile in some situations. The first step in mapping a scatterlist is to create and fill in an array of `struct scatterlist` describing the buffers to be transferred. This structure is architecture dependent, and is described in `<linux/scatterlist.h>`. It will always contain two fields, however:

`char *address;`
The address of a buffer used in the scatter/gather operation

`unsigned int length;`
The length of that buffer

To map a scatter/gather DMA operation, your driver should set the `address` and `length` fields in a `struct scatterlist` entry for each buffer to be transferred. Then call:

```
int pci_map_sg(struct pci_dev *pdev, struct scatterlist *list,
               int nents, int direction);
```

The return value will be the number of DMA buffers to transfer; it may be less than `nents`, the number of scatterlist entries passed in.

Your driver should transfer each buffer returned by *pci_map_sg*. The bus address and length of each buffer will be stored in the `struct scatterlist` entries, but their location in the structure varies from one architecture to the next. Two macros have been defined to make it possible to write portable code:

`dma_addr_t sg_dma_address(struct scatterlist *sg);`
Returns the bus (DMA) address from this scatterlist entry

`unsigned int sg_dma_len(struct scatterlist *sg);`
Returns the length of this buffer

Again, remember that the address and length of the buffers to transfer may be different from what was passed in to *pci_map_sg*.

Once the transfer is complete, a scatter-gather mapping is unmapped with a call to *pci_unmap_sg*:

```
void pci_unmap_sg(struct pci_dev *pdev, struct scatterlist *list,
                  int nents, int direction);
```

Note that `nents` must be the number of entries that you originally passed to *pci_map_sg*, and not the number of DMA buffers that function returned to you.

Scatter-gather mappings are streaming DMA mappings, and the same access rules apply to them as to the single variety. If you must access a mapped scatter-gather list, you must synchronize it first:

```
void pci_dma_sync_sg(struct pci_dev *pdev, struct scatterlist *sg,
                     int nents, int direction);
```

How different architectures support PCI DMA

As we stated at the beginning of this section, DMA is a very hardware-specific operation. The PCI DMA interface we have just described attempts to abstract out as many hardware dependencies as possible. There are still some things that show through, however.

M68K
S/390
Super-H
: These architectures do not support the PCI bus as of 2.4.0.

IA-32 (x86)
MIPS
PowerPC
ARM
: These platforms support the PCI DMA interface, but it is mostly a false front. There are no mapping registers in the bus interface, so scatterlists cannot be combined and virtual addresses cannot be used. There is no bounce buffer support, so mapping of high-memory addresses cannot be done. The mapping functions on the ARM architecture can sleep, which is not the case for the other platforms.

IA-64
: The Itanium architecture also lacks mapping registers. This 64-bit architecture can easily generate addresses that PCI peripherals cannot use, though. The PCI interface on this platform thus implements bounce buffers, allowing any address to be (seemingly) used for DMA operations.

Alpha
MIPS64
SPARC
: These architectures support an I/O memory management unit. As of 2.4.0, the MIPS64 port does not actually make use of this capability, so its PCI DMA implementation looks like that of the IA-32. The Alpha and SPARC ports, though, can do full-buffer mapping with proper scatter-gather support.

The differences listed will not be problems for most driver writers, as long as the interface guidelines are followed.

A simple PCI DMA example

The actual form of DMA operations on the PCI bus is very dependent on the device being driven. Thus, this example does not apply to any real device; instead, it is part of a hypothetical driver called *dad* (DMA Acquisition Device). A driver for this device might define a transfer function like this:

```
int dad_transfer(struct dad_dev *dev, int write, void *buffer,
                 size_t count)
{
    dma_addr_t bus_addr;
    unsigned long flags;

    /* Map the buffer for DMA */
    dev->dma_dir = (write ? PCI_DMA_TODEVICE : PCI_DMA_FROMDEVICE);
    dev->dma_size = count;
    bus_addr = pci_map_single(dev->pci_dev, buffer, count,
                              dev->dma_dir);
    dev->dma_addr = bus_addr;

    /* Set up the device */
    writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
    writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
    writel(dev->registers.addr, cpu_to_le32(bus_addr));
    writel(dev->registers.len, cpu_to_le32(count));

    /* Start the operation */
    writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
    return 0;
}
```

This function maps the buffer to be transferred and starts the device operation. The other half of the job must be done in the interrupt service routine, which would look something like this:

```
void dad_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct dad_dev *dev = (struct dad_dev *) dev_id;

    /* Make sure it's really our device interrupting */

    /* Unmap the DMA buffer */
    pci_unmap_single(dev->pci_dev, dev->dma_addr, dev->dma_size,
                     dev->dma_dir);

    /* Only now is it safe to access the buffer, copy to user, etc. */
    ...
}
```

Obviously a great deal of detail has been left out of this example, including whatever steps may be required to prevent attempts to start multiple simultaneous DMA operations.

A quick look at SBus

SPARC-based systems have traditionally included a Sun-designed bus called the SBus. This bus is beyond the scope of this chapter, but a quick mention is worthwhile. There is a set of functions (declared in `<asm/sbus.h>`) for performing DMA mappings on the SBus; they have names like *sbus_alloc_consistent* and

sbus_map_sg. In other words, the SBus DMA API looks almost exactly like the PCI interface. A detailed look at the function definitions will be required before working with DMA on the SBus, but the concepts will match those discussed earlier for the PCI bus.

DMA for ISA Devices

The ISA bus allows for two kinds of DMA transfers: native DMA and ISA bus master DMA. Native DMA uses standard DMA-controller circuitry on the motherboard to drive the signal lines on the ISA bus. ISA bus master DMA, on the other hand, is handled entirely by the peripheral device. The latter type of DMA is rarely used and doesn't require discussion here because it is similar to DMA for PCI devices, at least from the driver's point of view. An example of an ISA bus master is the 1542 SCSI controller, whose driver is *drivers/scsi/aha1542.c* in the kernel sources.

As far as native DMA is concerned, there are three entities involved in a DMA data transfer on the ISA bus:

The 8237 DMA controller (DMAC)
: The controller holds information about the DMA transfer, such as the direction, the memory address, and the size of the transfer. It also contains a counter that tracks the status of ongoing transfers. When the controller receives a DMA request signal, it gains control of the bus and drives the signal lines so that the device can read or write its data.

The peripheral device
: The device must activate the DMA request signal when it's ready to transfer data. The actual transfer is managed by the DMAC; the hardware device sequentially reads or writes data onto the bus when the controller strobes the device. The device usually raises an interrupt when the transfer is over.

The device driver
: The driver has little to do: it provides the DMA controller with the direction, bus address, and size of the transfer. It also talks to its peripheral to prepare it for transferring the data and responds to the interrupt when the DMA is over.

The original DMA controller used in the PC could manage four "channels," each associated with one set of DMA registers. Four devices could store their DMA information in the controller at the same time. Newer PCs contain the equivalent of two DMAC devices:* the second controller (master) is connected to the system processor, and the first (slave) is connected to channel 0 of the second controller.†

---

* These circuits are now part of the motherboard's chipset, but a few years ago they were two separate 8237 chips.

† The original PCs had only one controller; the second was added in 286-based platforms. However, the second controller is connected as the master because it handles 16-bit transfers; the first transfers only 8 bits at a time and is there for backward compatibility.

The channels are numbered from 0 to 7; channel 4 is not available to ISA peripherals because it is used internally to cascade the slave controller onto the master. The available channels are thus 0 to 3 on the slave (the 8-bit channels) and 5 to 7 on the master (the 16-bit channels). The size of any DMA transfer, as stored in the controller, is a 16-bit number representing the number of bus cycles. The maximum transfer size is therefore 64 KB for the slave controller and 128 KB for the master.

Because the DMA controller is a system-wide resource, the kernel helps deal with it. It uses a DMA registry to provide a request-and-free mechanism for the DMA channels and a set of functions to configure channel information in the DMA controller.

Registering DMA usage

You should be used to kernel registries—we've already seen them for I/O ports and interrupt lines. The DMA channel registry is similar to the others. After `<asm/dma.h>` has been included, the following functions can be used to obtain and release ownership of a DMA channel:

```
int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);
```

The `channel` argument is a number between 0 and 7 or, more precisely, a positive number less than `MAX_DMA_CHANNELS`. On the PC, `MAX_DMA_CHANNELS` is defined as 8, to match the hardware. The `name` argument is a string identifying the device. The specified name appears in the file */proc/dma*, which can be read by user programs.

The return value from *request_dma* is 0 for success and `-EINVAL` or `-EBUSY` if there was an error. The former means that the requested channel is out of range, and the latter means that another device is holding the channel.

We recommend that you take the same care with DMA channels as with I/O ports and interrupt lines; requesting the channel at *open* time is much better than requesting it from the module initialization function. Delaying the request allows some sharing between drivers; for example, your sound card and your analog I/O interface can share the DMA channel as long as they are not used at the same time.

We also suggest that you request the DMA channel *after* you've requested the interrupt line and that you release it *before* the interrupt. This is the conventional order for requesting the two resources; following the convention avoids possible deadlocks. Note that every device using DMA needs an IRQ line as well; otherwise, it couldn't signal the completion of data transfer.

In a typical case, the code for *open* looks like the following, which refers to our hypothetical *dad* module. The *dad* device as shown uses a fast interrupt handler without support for shared IRQ lines.

```
int dad_open (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;

    /* ... */
    if ( (error = request_irq(my_device.irq, dad_interrupt,
                              SA_INTERRUPT, "dad", NULL)) )
        return error; /* or implement blocking open */

    if ( (error = request_dma(my_device.dma, "dad")) ) {
        free_irq(my_device.irq, NULL);
        return error; /* or implement blocking open */
    }
    /* ... */
    return 0;
}
```

The *close* implementation that matches the *open* just shown looks like this:

```
void dad_close (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;

    /* ... */
    free_dma(my_device.dma);
    free_irq(my_device.irq, NULL);
    /* ... */
}
```

As far as */proc/dma* is concerned, here's how the file looks on a system with the sound card installed:

```
merlino% cat /proc/dma
 1: Sound Blaster8
 4: cascade
```

It's interesting to note that the default sound driver gets the DMA channel at system boot and never releases it. The `cascade` entry shown is a placeholder, indicating that channel 4 is not available to drivers, as explained earlier.

Talking to the DMA controller

After registration, the main part of the driver's job consists of configuring the DMA controller for proper operation. This task is not trivial, but fortunately the kernel exports all the functions needed by the typical driver.

The driver needs to configure the DMA controller either when *read* or *write* is called, or when preparing for asynchronous transfers. This latter task is performed either at *open* time or in response to an *ioctl* command, depending on the driver and the policy it implements. The code shown here is the code that is typically called by the *read* or *write* device methods.

This subsection provides a quick overview of the internals of the DMA controller so you will understand the code introduced here. If you want to learn more, we'd urge you to read `<asm/dma.h>` and some hardware manuals describing the PC architecture. In particular, we don't deal with the issue of 8-bit versus 16-bit data transfers. If you are writing device drivers for ISA device boards, you should find the relevant information in the hardware manuals for the devices.

The DMA controller is a shared resource, and confusion could arise if more than one processor attempts to program it simultaneously. For that reason, the controller is protected by a spinlock, called `dma_spin_lock`. Drivers should not manipulate the lock directly, however; two functions have been provided to do that for you:

`unsigned long claim_dma_lock();`
: Acquires the DMA spinlock. This function also blocks interrupts on the local processor; thus the return value is the usual "flags" value, which must be used when reenabling interrupts.

`void release_dma_lock(unsigned long flags);`
: Returns the DMA spinlock and restores the previous interrupt status.

The spinlock should be held when using the functions described next. It should *not* be held during the actual I/O, however. A driver should never sleep when holding a spinlock.

The information that must be loaded into the controller is made up of three items: the RAM address, the number of atomic items that must be transferred (in bytes or words), and the direction of the transfer. To this end, the following functions are exported by `<asm/dma.h>`:

`void set_dma_mode(unsigned int channel, char mode);`
: Indicates whether the channel must read from the device (`DMA_MODE_READ`) or write to it (`DMA_MODE_WRITE`). A third mode exists, `DMA_MODE_CASCADE`, which is used to release control of the bus. Cascading is the way the first controller is connected to the top of the second, but it can also be used by true ISA bus-master devices. We won't discuss bus mastering here.

`void set_dma_addr(unsigned int channel, unsigned int addr);`
: Assigns the address of the DMA buffer. The function stores the 24 least significant bits of `addr` in the controller. The `addr` argument must be a *bus* address (see "Bus Addresses" earlier in this chapter).

`void set_dma_count(unsigned int channel, unsigned int count);`
Assigns the number of bytes to transfer. The `count` argument represents bytes for 16-bit channels as well; in this case, the number *must* be even.

In addition to these functions, there are a number of housekeeping facilities that must be used when dealing with DMA devices:

`void disable_dma(unsigned int channel);`
A DMA channel can be disabled within the controller. The channel should be disabled before the controller is configured, to prevent improper operation (the controller is programmed via eight-bit data transfers, and thus none of the previous functions is executed atomically).

`void enable_dma(unsigned int channel);`
This function tells the controller that the DMA channel contains valid data.

`int get_dma_residue(unsigned int channel);`
The driver sometimes needs to know if a DMA transfer has been completed. This function returns the number of bytes that are still to be transferred. The return value is 0 after a successful transfer and is unpredictable (but not 0) while the controller is working. The unpredictability reflects the fact that the residue is a 16-bit value, which is obtained by two 8-bit input operations.

`void clear_dma_ff(unsigned int channel)`
This function clears the DMA flip-flop. The flip-flop is used to control access to 16-bit registers. The registers are accessed by two consecutive 8-bit operations, and the flip-flop is used to select the least significant byte (when it is clear) or the most significant byte (when it is set). The flip-flop automatically toggles when 8 bits have been transferred; the programmer must clear the flip-flop (to set it to a known state) before accessing the DMA registers.

Using these functions, a driver can implement a function like the following to prepare for a DMA transfer:

```
int dad_dma_prepare(int channel, int mode, unsigned int buf,
                    unsigned int count)
{
    unsigned long flags;

    flags = claim_dma_lock();
    disable_dma(channel);
    clear_dma_ff(channel);
    set_dma_mode(channel, mode);
    set_dma_addr(channel, virt_to_bus(buf));
    set_dma_count(channel, count);
    enable_dma(channel);
    release_dma_lock(flags);
```

```
        return 0;
    }
```

A function like the next one, then, is used to check for successful completion of DMA:

```
int dad_dma_isdone(int channel)
{
    int residue;
    unsigned long flags = claim_dma_lock ();
    residue = get_dma_residue(channel);
    release_dma_lock(flags);
    return (residue == 0);
}
```

The only thing that remains to be done is to configure the device board. This device-specific task usually consists of reading or writing a few I/O ports. Devices differ in significant ways. For example, some devices expect the programmer to tell the hardware how big the DMA buffer is, and sometimes the driver has to read a value that is hardwired into the device. For configuring the board, the hardware manual is your only friend.

Backward Compatibility

As with other parts of the kernel, both memory mapping and DMA have seen a number of changes over the years. This section describes the things a driver writer must take into account in order to write portable code.

Changes to Memory Management

The 2.3 development series saw major changes in the way memory management worked. The 2.2 kernel was quite limited in the amount of memory it could use, especially on 32-bit processors. With 2.4, those limits have been lifted; Linux is now able to manage all the memory that the processor is able to address. Some things have had to change to make all this possible; overall, however, the scale of the changes at the API level is surprisingly small.

As we have seen, the 2.4 kernel makes extensive use of pointers to `struct page` to refer to specific pages in memory. This structure has been present in Linux for a long time, but it was not previously used to refer to the pages themselves; instead, the kernel used logical addresses.

Thus, for example, *pte_page* returned an `unsigned long` value instead of `struct page *`. The *virt_to_page* macro did not exist at all; if you needed to find a `struct page` entry you had to go directly to the memory map to get it. The macro `MAP_NR` would turn a logical address into an index in `mem_map`; thus, the current *virt_to_page* macro could be defined (and, in *sysdep.h* in the sample code, is defined) as follows:

```
#ifdef MAP_NR
#define virt_to_page(page) (mem_map + MAP_NR(page))
#endif
```

The `MAP_NR` macro went away when *virt_to_page* was introduced. The *get_page* macro also didn't exist prior to 2.4, so *sysdep.h* defines it as follows:

```
#ifndef get_page
#  define get_page(p) atomic_inc(&(p)->count)
#endif
```

`struct page` has also changed with time; in particular, the `virtual` field is present in Linux 2.4 only.

The `page_table_lock` was introduced in 2.3.10. Earlier code would obtain the "big kernel lock" (by calling *lock_kernel* and *unlock_kernel*) before traversing page tables.

The `vm_area_struct` structure saw a number of changes in the 2.3 development series, and more in 2.1. These included the following:

- The `vm_pgoff` field was called `vm_offset` in 2.2 and before. It was an offset in bytes, not pages.
- The `vm_private_data` field did not exist in Linux 2.2, so drivers had no way of storing their own information in the VMA. A number of them did so anyway, using the `vm_pte` field, but it would be safer to obtain the minor device number from `vm_file` and use it to retrieve the needed information.
- The 2.4 kernel initializes the `vm_file` pointer before calling the *mmap* method. In 2.2, drivers had to assign that value themselves, using the `file` structure passed in as an argument.
- The `vm_file` pointer did not exist at all in 2.0 kernels; instead, there was a `vm_inode` pointer pointing to the `inode` structure. This field needed to be assigned by the driver; it was also necessary to increment `inode->i_count` in the *mmap* method.
- The `VM_RESERVED` flag was added in kernel 2.4.0-test10.

There have also been changes to the the various `vm_ops` methods stored in the VMA:

- 2.2 and earlier kernels had a method called *advise*, which was never actually used by the kernel. There was also a *swapin* method, which was used to bring in memory from backing store; it was not generally of interest to driver writers.
- The *nopage* and *wppage* methods returned `unsigned long` (i.e., a logical address) in 2.2, rather than `struct page *`.

- The `NOPAGE_SIGBUS` and `NOPAGE_OOM` return codes for *nopage* did not exist. *nopage* simply returned 0 to indicate a problem and send a bus signal to the affected process.

Because *nopage* used to return `unsigned long`, its job was to return the logical address of the page of interest, rather than its `mem_map` entry.

There was, of course, no high-memory support in older kernels. All memory had logical addresses, and the *kmap* and *kunmap* functions did not exist.

In the 2.0 kernel, the `init_mm` structure was not exported to modules. Thus, a module that wished to access `init_mm` had to dig through the task table to find it (as part of the *init* process). When running on a 2.0 kernel, *scullp* finds `init_mm` with this bit of code:

```
static struct mm_struct *init_mm_ptr;
#define init_mm (*init_mm_ptr) /* to avoid ifdefs later */

static void retrieve_init_mm_ptr(void)
{
    struct task_struct *p;

    for (p = current ; (p = p->next_task) != current ; )
        if (p->pid == 0)
            break;

    init_mm_ptr = p->mm;
}
```

The 2.0 kernel also lacked the distinction between logical and physical addresses, so the *__va* and *__pa* macros did not exist. There was no need for them at that time.

Another thing the 2.0 kernel did not have was maintenance of the module's usage count in the presence of memory-mapped areas. Drivers that implement *mmap* under 2.0 need to provide *open* and *close* VMA operations to adjust the usage count themselves. The sample source modules that implement *mmap* provide these operations.

Finally, the 2.0 version of the driver *mmap* method, like most others, had a `struct inode` argument; the method's prototype was

```
int (*mmap)(struct inode *inode, struct file *filp,
            struct vm_area_struct *vma);
```

Changes to DMA

The PCI DMA interface as described earlier did not exist prior to kernel 2.3.41. Before then, DMA was handled in a more direct—and system-dependent—way. Buffers were "mapped" by calling *virt_to_bus*, and there was no general interface for handling bus-mapping registers.

For those who need to write portable PCI drivers, *sysdep.h* in the sample code includes a simple implementation of the 2.4 DMA interface that may be used on older kernels.

The ISA interface, on the other hand, is almost unchanged since Linux 2.0. ISA is an old architecture, after all, and there have not been a whole lot of changes to keep up with. The only addition was the DMA spinlock in 2.2; prior to that kernel, there was no need to protect against conflicting access to the DMA controller. Versions of these functions have been defined in *sysdep.h*; they disable and restore interrupts, but perform no other function.

Quick Reference

This chapter introduced the following symbols related to memory handling. The list doesn't include the symbols introduced in the first section, as that section is a huge list in itself and those symbols are rarely useful to device drivers.

```
#include <linux/mm.h>
```
All the functions and structures related to memory management are prototyped and defined in this header.

```
int remap_page_range(unsigned long virt_add, unsigned long
      phys_add, unsigned long size, pgprot_t prot);
```
This function sits at the heart of *mmap*. It maps `size` bytes of physical addresses, starting at `phys_addr`, to the virtual address `virt_add`. The protection bits associated with the virtual space are specified in `prot`.

```
struct page *virt_to_page(void *kaddr);
void *page_address(struct page *page);
```
These macros convert between kernel logical addresses and their associated memory map entries. *page_address* only works for low-memory pages, or high-memory pages that have been explicitly mapped.

```
void *__va(unsigned long physaddr);
unsigned long __pa(void *kaddr);
```
These macros convert between kernel logical addresses and physical addresses.

```
unsigned long kmap(struct page *page);
void kunmap(struct page *page);
```
kmap returns a kernel virtual address that is mapped to the given page, creating the mapping if need be. *kunmap* deletes the mapping for the given page.

```
#include <linux/iobuf.h>
void kiobuf_init(struct kiobuf *iobuf);
int alloc_kiovec(int number, struct kiobuf **iobuf);
void free_kiovec(int number, struct kiobuf **iobuf);
```
These functions handle the allocation, initialization, and freeing of kernel I/O buffers. *kiobuf_init* initializes a single kiobuf, but is rarely used; *alloc_kiovec*, which allocates and initializes a vector of kiobufs, is usually used instead. A vector of kiobufs is freed with *free_kiovec*.

```
int lock_kiovec(int nr, struct kiobuf *iovec[], int wait);
int unlock_kiovec(int nr, struct kiobuf *iovec[]);
```
These functions lock a kiovec in memory, and release it. They are unnecessary when using kiobufs for I/O to user-space memory.

```
int map_user_kiobuf(int rw, struct kiobuf *iobuf, unsigned
      long address, size_t len);
void unmap_kiobuf(struct kiobuf *iobuf);
```
map_user_kiobuf maps a buffer in user space into the given kernel I/O buffer; *unmap_kiobuf* undoes that mapping.

```
#include <asm/io.h>
unsigned long virt_to_bus(volatile void * address);
void * bus_to_virt(unsigned long address);
```
These functions convert between kernel virtual and bus addresses. Bus addresses must be used to talk to peripheral devices.

```
#include <linux/pci.h>
```
The header file required to define the following functions.

```
int pci_dma_supported(struct pci_dev *pdev, dma_addr_t
      mask);
```
For peripherals that cannot address the full 32-bit range, this function determines whether DMA can be supported at all on the host system.

```
void *pci_alloc_consistent(struct pci_dev *pdev, size_t
      size, dma_addr_t *bus_addr)
void pci_free_consistent(struct pci_dev *pdev, size_t size,
      void *cpuaddr, dma_handle_t bus_addr);
```
These functions allocate and free consistent DMA mappings, for a buffer that will last the lifetime of the driver.

```
PCI_DMA_TODEVICE
PCI_DMA_FROMDEVICE
PCI_DMA_BIDIRECTIONAL
PCI_DMA_NONE
```
These symbols are used to tell the streaming mapping functions the direction in which data will be moving to or from the buffer.

```
dma_addr_t pci_map_single(struct pci_dev *pdev, void
      *buffer, size_t size, int direction);
void pci_unmap_single(struct pci_dev *pdev, dma_addr_t
      bus_addr, size_t size, int direction);
```
Create and destroy a single-use, streaming DMA mapping.

```
void pci_sync_single(struct pci_dev *pdev, dma_handle_t
      bus_addr, size_t size, int direction)
```
Synchronizes a buffer that has a streaming mapping. This function must be used if the processor must access a buffer while the streaming mapping is in place (i.e., while the device owns the buffer).

```
struct scatterlist { /* ... */ };
dma_addr_t sg_dma_address(struct scatterlist *sg);
unsigned int sg_dma_len(struct scatterlist *sg);
```
The `scatterlist` structure describes an I/O operation that involves more than one buffer. The macros *sg_dma_address* and *sg_dma_len* may be used to extract bus addresses and buffer lengths to pass to the device when implementing scatter-gather operations.

```
pci_map_sg(struct pci_dev *pdev, struct scatterlist *list,
      int nents, int direction);
pci_unmap_sg(struct pci_dev *pdev, struct scatterlist *list,
      int nents, int direction);
pci_dma_sync_sg(struct pci_dev *pdev, struct scatterlist
      *sg, int nents, int direction)
```
pci_map_sg maps a scatter-gather operation, and *pci_unmap_sg* undoes that mapping. If the buffers must be accessed while the mapping is active, *pci_dma_sync_sg* may be used to synchronize things.

`/proc/dma`
This file contains a textual snapshot of the allocated channels in the DMA controllers. PCI-based DMA is not shown because each board works independently, without the need to allocate a channel in the DMA controller.

```
#include <asm/dma.h>
```
This header defines or prototypes all the functions and macros related to DMA. It must be included to use any of the following symbols.

```
int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);
```
These functions access the DMA registry. Registration must be performed before using ISA DMA channels.

```
unsigned long claim_dma_lock();
void release_dma_lock(unsigned long flags);
```
These functions acquire and release the DMA spinlock, which must be held prior to calling the other ISA DMA functions described later in this list. They also disable and reenable interrupts on the local processor.

```
void set_dma_mode(unsigned int channel, char mode);
void set_dma_addr(unsigned int channel, unsigned int addr);
void set_dma_count(unsigned int channel, unsigned int
      count);
```
These functions are used to program DMA information in the DMA controller. `addr` is a bus address.

```
void disable_dma(unsigned int channel);
void enable_dma(unsigned int channel);
```
A DMA channel must be disabled during configuration. These functions change the status of the DMA channel.

```
int get_dma_residue(unsigned int channel);
```
If the driver needs to know how a DMA transfer is proceeding, it can call this function, which returns the number of data transfers that are yet to be completed. After successful completion of DMA, the function returns 0; the value is unpredictable while data is being transferred.

```
void clear_dma_ff(unsigned int channel)
```
The DMA flip-flop is used by the controller to transfer 16-bit values by means of two 8-bit operations. It must be cleared before sending any data to the controller.