CHAPTER 15

Memory Mapping 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. Many types of driver programming require some understanding of how the virtual memory subsystem works; the material we cover in this chapter comes in handy more than once as we get into some of the more complex and performance-critical subsystems. The virtual memory subsystem is also a highly interesting part of the core Linux kernel and, therefore, it merits a look.

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. Not all devices require *mmap* support, but, for some, mapping device memory can yield significant performance improvements.
- We then look at crossing the boundary from the other direction with a discussion of direct access to user-space pages. Relatively few drivers need this capability; in many cases, the kernel performs this sort of mapping without the driver even being aware of it. But an awareness of how to map user-space memory into the kernel (with *get_user_pages*) can be useful.
- The final section covers direct memory access (DMA) I/O operations, which 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. 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 that 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 the program to play a number of tricks with the process's address space, including mapping the program's memory to 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.

The following is a list of address types used in Linux. Figure 15-1 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 larger 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. Some architectures can provide an I/O memory management unit (IOMMU) that remaps addresses between a bus and main memory. An IOMMU can make life easier in a number of ways (making a buffer scattered in memory appear contiguous to the device, for example), but programming the IOMMU is an extra step that must be performed when setting up DMA operations. Bus addresses are highly architecture dependent, of course.

Kernel logical addresses

These make up the normal address space of the kernel. These addresses map some portion (perhaps 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, therefore, 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 kernel logical address.

Kernel virtual addresses

Kernel virtual addresses are similar to logical addresses in that they are a mapping from a kernel-space address to a physical address. Kernel virtual addresses do not necessarily have the linear, one-to-one mapping to physical addresses that characterize the logical address space, however. All logical addresses *are* kernel virtual addresses, but many kernel virtual addresses are not logical addresses. For example, memory allocated by *vmalloc* has a virtual address (but no direct physical mapping). The *kmap* function (described later in this chapter) also returns virtual addresses. Virtual addresses are usually stored in pointer variables.

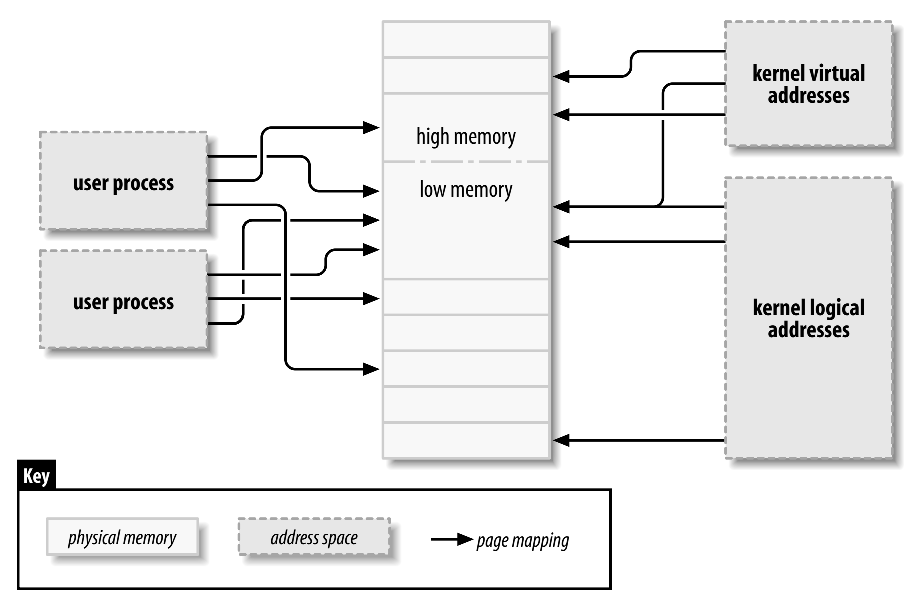

Figure 15-1. Address types used in Linux

If you have a logical address, the macro *__pa()* (defined in *<asm/page.h>*) returns 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 types were explicit, but we have no such luck. In this chapter, we try to be clear on which types of addresses are used where.

Physical Addresses and Pages

Physical memory is divided into discrete units called *pages*. Much of the system's internal handling of memory is done on a per-page basis. Page size varies from one architecture to the next, although most systems currently use 4096-byte pages. The constant `PAGE_SIZE` (defined in *<asm/page.h>*) gives the page size on any given architecture.

If you look at a memory address—virtual or physical—it is divisible into a page number and an offset within the page. If 4096-byte pages are being used, for example, the 12 least-significant bits are the offset, and the remaining, higher bits indicate the page number. If you discard the offset and shift the rest of an offset to the right, the result is called a *page frame number* (PFN). Shifting bits to convert between page frame numbers and addresses is a fairly common operation; the macro `PAGE_SHIFT` tells how many bits must be shifted to make this conversion.

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 kernel (on the x86 architecture, in the default configuration) splits the 4-GB virtual address space between user-space and the kernel; the same set of mappings is used in both contexts. A typical split dedicates 3 GB to user space, and 1 GB for kernel space.[*] The kernel's code and data structures must fit into that space, but the biggest consumer of kernel address space is virtual mappings for physical memory. The kernel cannot directly manipulate memory that is not mapped into the kernel's address space. The kernel, in other words, needs its own virtual address for any memory it must touch directly. Thus, for many years, the maximum amount of physical memory that could be handled by the kernel was the amount that could be mapped into the kernel's portion of the virtual address space, minus the space

---

[*] Many non-x86 architectures are able to efficiently do without the kernel/user-space split described here, so they can work with up to a 4-GB kernel address space on 32-bit systems. The constraints described in this section still apply to such systems when more than 4 GB of memory are installed, however.

needed for the kernel code itself. As a result, x86-based Linux systems could work with a maximum of a little under 1 GB of physical memory.

In response to commercial pressure to support more memory while not breaking 32-bit application and the system's compatibility, the processor manufacturers have added "address extension" features to their products. The result is that, in many cases, even 32-bit processors can address more than 4 GB of physical 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. Before accessing a specific high-memory page, the kernel must set up an explicit virtual mapping to make that page available in the kernel's address space. Thus, many kernel data structures must be placed in low memory; 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 it is beyond the address range set aside for kernel virtual addresses.

On i386 systems, the boundary between low and high memory is usually set at just under 1 GB, although that boundary can be changed at kernel configuration time. This boundary is not related in any way to the old 640 KB limit found on the original PC, and its placement is not dictated by the hardware. 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 limitations on the use of high memory as we come to them in this chapter.

The Memory Map and Struct Page

Historically, the kernel has used logical addresses to refer to pages of physical memory. The addition of high-memory support, however, has exposed an obvious problem with that approach—logical addresses are not available for high memory. Therefore, kernel functions that deal with memory are increasingly using pointers to `struct page` (defined in <*linux/mm.h*>) instead. This data structure is used to keep track of just about everything the kernel needs to know about physical memory;

---

* The 2.6 kernel (with an added patch) can support a "4G/4G" mode on x86 hardware, which enables larger kernel and user virtual address spaces at a mild performance cost.

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 `0`, the page is returned to the free list.

`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. This field does not appear on all architectures; it generally is compiled only where the kernel virtual address of a page cannot be easily calculated. If you want to look at this field, the proper method is to use the *page_address* macro, described below.

`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 that track all of the physical memory on the system. On some 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 does not work with memory from *vmalloc* or high memory.

`struct page *pfn_to_page(int pfn);`
: Returns the `struct page` pointer for the given page frame number. If necessary, it checks a page frame number for validity with *pfn_valid* before passing it to *pfn_to_page*.

`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. This function is

defined in *<linux/mm.h>*. In most situations, you want to use a version of *kmap* rather than *page_address*.

```
#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 in a dedicated part of the kernel address space. 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 maintain a counter, 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.

```
#include <linux/highmem.h>
#include <asm/kmap_types.h>
void *kmap_atomic(struct page *page, enum km_type type);
void kunmap_atomic(void *addr, enum km_type type);
```

kmap_atomic is a high-performance form of *kmap*. Each architecture maintains a small list of slots (dedicated page table entries) for atomic kmaps; a caller of *kmap_atomic* must tell the system which of those slots to use in the `type` argument. The only slots that make sense for drivers are `KM_USER0` and `KM_USER1` (for code running directly from a call from user space), and `KM_IRQ0` and `KM_IRQ1` (for interrupt handlers). Note that atomic kmaps must be handled atomically; your code cannot sleep while holding one. Note also that nothing in the kernel keeps two functions from trying to use the same slot and interfering with each other (although there is a unique set of slots for each CPU). In practice, contention for atomic kmap slots seems to not be a problem.

We see some uses of these functions when we get into the example code, later in this chapter and in subsequent chapters.

Page Tables

On any modern system, the processor must have a mechanism for translating virtual addresses into its corresponding physical addresses. This mechanism is called a *page table*; it is essentially a multilevel tree-structured array containing virtual-to-physical mappings and a few associated flags. The Linux kernel maintains a set of page tables even on architectures that do not use such tables directly.

A number of operations commonly performed by device drivers can involve manipulating page tables. Fortunately for the driver author, the 2.6 kernel has eliminated any need to work with page tables directly. As a result, we do not describe them in any detail; curious readers may want to have a look at *Understanding The Linux Kernel* by Daniel P. Bovet and Marco Cesati (O'Reilly) for the full story.

Virtual Memory Areas

The virtual memory area (VMA) is the kernel data structure used to manage distinct regions of a process's address space. A VMA represents a homogeneous region in the virtual memory of a process: a contiguous range of virtual addresses that have the same permission flags and are backed up by the same object (a file, say, or swap space). 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 (at least) the following areas:

- An area for the program's executable code (often called text)
- Multiple areas 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>* (in which *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 in italics):

```
# cat /proc/1/maps    look at init
08048000-0804e000 r-xp 00000000 03:01 64652      /sbin/init  text
0804e000-0804f000 rw-p 00006000 03:01 64652      /sbin/init  data
0804f000-08053000 rwxp 00000000 00:00 0          zero-mapped BSS
40000000-40015000 r-xp 00000000 03:01 96278      /lib/ld-2.3.2.so  text
40015000-40016000 rw-p 00014000 03:01 96278      /lib/ld-2.3.2.so  data
40016000-40017000 rw-p 00000000 00:00 0          BSS for ld.so
42000000-4212e000 r-xp 00000000 03:01 80290      /lib/tls/libc-2.3.2.so  text
4212e000-42131000 rw-p 0012e000 03:01 80290      /lib/tls/libc-2.3.2.so  data
42131000-42133000 rw-p 00000000 00:00 0          BSS for libc
bffff000-c0000000 rwxp 00000000 00:00 0          Stack segment
ffffe000-fffff000 ---p 00000000 00:00 0          vsyscall page

# rsh wolf cat /proc/self/maps  #### x86-64 (trimmed)
00400000-00405000 r-xp 00000000 03:01 1596291     /bin/cat    text
00504000-00505000 rw-p 00004000 03:01 1596291     /bin/cat    data
00505000-00526000 rwxp 00505000 00:00 0                       bss
3252200000-3252214000 r-xp 00000000 03:01 1237890 /lib64/ld-2.3.3.so
3252300000-3252301000 r--p 00100000 03:01 1237890 /lib64/ld-2.3.3.so
3252301000-3252302000 rw-p 00101000 03:01 1237890 /lib64/ld-2.3.3.so
7fbfffe000-7fc0000000 rw-p 7fbfffe000 00:00 0                 stack
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0       vsyscall
```

The fields in each line are:

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

---

[*] 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.

Each field in */proc/*/maps* (except the image name) corresponds to a field in `struct vm_area_struct`:

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

`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 `0` means that the beginning of the memory area corresponds to the beginning 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.

The vm_area_struct structure

When a user-space process calls *mmap* to map device memory into its address space, the system responds by creating a new VMA to represent that mapping. A driver that supports *mmap* (and, thus, that implements the *mmap* method) needs to help that process by completing the initialization of that VMA. The driver writer should, therefore, have at least a minimal understanding of VMAs in order to support *mmap*.

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. Therefore, VMAs can't be created at will by a driver, or the structures 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 prevents 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 below. 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 are implemented.

`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. This method is invoked any time 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.

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

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.

`int (*populate)(struct vm_area_struct *vm, unsigned long address, unsigned long len, pgprot_t prot, unsigned long pgoff, int nonblock);`

This method allows the kernel to "prefault" pages into memory before they are accessed by user space. There is generally no need for drivers to implement the *populate* method.

The Process Memory Map

The final piece of the memory management puzzle is the process memory map structure, which holds all of the other data structures together. Each process in the system (with the exception of a few kernel-space helper threads) has a `struct mm_struct` (defined in *<linux/sched.h>*) that contains the process's list of virtual memory areas, page tables, and various other bits of memory management housekeeping information, along with a semaphore (`mmap_sem`) and a spinlock (`page_table_lock`). The pointer to this structure is found in the task structure; in the rare cases where a driver needs to access it, the usual way is to use `current->mm`. Note that the memory management structure can be shared between processes; the Linux implementation of threads works in this way, for example.

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 implemented 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
000a0000-000c0000 rwxs 000a0000 03:01 282652      /dev/mem
000f0000-00100000 r-xs 000f0000 03:01 282652      /dev/mem
00400000-005c0000 r-xp 00000000 03:01 1366927     /usr/X11R6/bin/Xorg
006bf000-006f7000 rw-p 001bf000 03:01 1366927     /usr/X11R6/bin/Xorg
2a95828000-2a958a8000 rw-s fcc00000 03:01 282652  /dev/mem
2a958a8000-2a9d8a8000 rw-s e8000000 03:01 282652  /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, four separate mappings of */dev/mem*, which give some insight into how the X server works with the video card. The first mapping is at `a0000`, which is the standard location for video RAM in the 640-KB ISA hole. Further down, we see a large mapping at `e8000000`, an address which is above the highest RAM address on the system. This is a direct mapping of the video memory on the adapter.

These regions can also be seen in */proc/iomem*:

```
000a0000-000bffff : Video RAM area
000c0000-000ccfff : Video ROM
000d1000-000d1fff : Adapter ROM
000f0000-000fffff : System ROM
d7f00000-f7efffff : PCI Bus #01
  e8000000-efffffff : 0000:01:00.0
fc700000-fccfffff : PCI Bus #01
  fcc00000-fcc0ffff : 0000:01:00.0
```

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 manage 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 forces 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. Since the program must know about how the device works, the programmer is not unduly bothered by the need to see to details like page alignment. 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 high-performance 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. Therefore, much of the work has 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_pfn_range` or doing it a page at a time via the *nopage* VMA method. Each method has its advantages and limitations. We start with the "all at once" approach, which is simpler. From there, we add the complications needed for a real-world implementation.

Using remap_pfn_range

The job of building new page tables to map a range of physical addresses is handled by *remap_pfn_range* and *io_remap_page_range*, which have the following prototypes:

```
int remap_pfn_range(struct vm_area_struct *vma,
                     unsigned long virt_addr, unsigned long pfn,
                     unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma,
                        unsigned long virt_addr, unsigned long phys_addr,
                        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:

`vma`
: The virtual memory area into which the page range is being mapped.

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

`pfn`
: The page frame number corresponding to the physical address to which the virtual address should be mapped. The page frame number is simply the physical address right-shifted by `PAGE_SHIFT` bits. For most uses, the `vm_pgoff` field of the VMA structure contains exactly the value you need. The function affects physical addresses from `(pfn<<PAGE_SHIFT)` to `(pfn<<PAGE_SHIFT)+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_pfn_range* are fairly straightforward, and most of them are already provided to you in the VMA when your *mmap* method is called. You may be wondering why there are two functions, however. The first (*remap_pfn_range*) is intended for situations where `pfn` refers to actual system RAM, while *io_remap_page_range* should be used when `phys_addr` points to I/O memory. In practice, the two functions are identical on every architecture except the SPARC, and you see *remap_pfn_range* used in most situations. In the interest of writing portable drivers, however, you should use the variant of *remap_pfn_range* that is suited to your particular situation.

One other complication has to do with caching: usually, references to device memory should not be cached by the processor. Often the system BIOS sets 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 *pgprot_noncached* function 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_pfn_range* is almost all you really need to do the job. The following code is derived 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):

```
static int simple_remap_mmap(struct file *filp, struct vm_area_struct *vma)
{
    if (remap_pfn_range(vma, vma->vm_start, vm->vm_pgoff,
                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;
}
```

As you can see, remapping memory just a matter of calling *remap_pfn_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 look at providing those operations in a simple way. In particular, we provide *open* and *close* operations for our VMA. These operations are called whenever a process opens or closes the VMA; in particular, the *open* method is 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.

As it turns out, a simple driver such as *simple* need not do any extra processing in particular. So we have created *open* and *close* methods, which print a message to the system log informing the world that they have been called. Not particularly useful, but it does allow us to show how these methods can be provided, and see when they are invoked.

To this end, we override the default `vma->vm_ops` with operations that call *printk*:

```
void simple_vma_open(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "Simple VMA open, virt %lx, phys %lx\n",
            vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}

void simple_vma_close(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "Simple VMA close.\n");
}

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

To make these operations active for a specific mapping, it is necessary to store a pointer to `simple_remap_vm_ops` in the `vm_ops` field of the relevant VMA. This is usually done in the *mmap* method. If you turn back to the *simple_remap_mmap* example, you see these lines of code:

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

Note the explicit call to *simple_vma_open*. Since the *open* method is not invoked on the initial *mmap*, we must call it explicitly if we want it to run.

Mapping Memory with nopage

Although *remap_pfn_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.

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. As it happens, the kernel does not notify drivers directly when a mapped VMA is changed by *mremap*. If the VMA is reduced in size, the kernel can quietly flush out the unwanted pages without telling the driver. If, instead, the VMA is expanded, the driver eventually finds out by way of calls to *nopage* when mappings must be set up for the new pages, so there is no need to perform a separate notification. The *nopage* method, therefore, must be implemented if you want to support the *mremap* system call. Here, we show a simple implementation of *nopage* for the *simple* device.

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

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

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 contains 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 `0`, the kernel knows that the page may be placed on the free list. When a VMA is unmapped, the kernel decrements 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 becomes `0` prematurely, and the integrity of the system is compromised.

The *nopage* method should also store the type of fault in the location pointed to by the `type` argument—but only if that argument is not `NULL`. In device drivers, the proper value for `type` will invariably be `VM_FAULT_MINOR`.

If you are using *nopage*, there is usually very little work to be done when *mmap* is called; our version looks like this:

```
static int simple_nopage_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;

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

The main thing *mmap* has to do is to replace the default (`NULL`) `vm_ops` pointer with our own operations. The *nopage* method then 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 only need to locate and return a pointer to the `struct page` for the desired address. Our *nopage* method looks like the following:

```
struct page *simple_vma_nopage(struct vm_area_struct *vma,
                unsigned long address, int *type)
{
    struct page *pageptr;
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    unsigned long physaddr = address - vma->vm_start + offset;
    unsigned long pageframe = physaddr >> PAGE_SHIFT;

    if (!pfn_valid(pageframe))
        return NOPAGE_SIGBUS;
    pageptr = pfn_to_page(pageframe);
    get_page(pageptr);
    if (type)
        *type = VM_FAULT_MINOR;
    return pageptr;
}
```

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. Therefore, the required sequence of events is to calculate the desired physical address, and turn it into a page frame number by right-shifting it `PAGE_SHIFT` bits. Since user space can give us any address it likes, we must ensure that we have a valid page frame; the *pfn_valid* function does that for us. If the address is out of range, we return `NOPAGE_SIGBUS`, which causes a bus signal to be delivered to the calling process.

Otherwise, *pfn_to_page* gets the necessary `struct page` pointer; we can increment its reference count (with a call to *get_page*) and return it.

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; that is what the *simple* code above does. *nopage* can also return `NOPAGE_OOM` to indicate failures caused by resource limitations.

Note that this implementation works 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 no `struct page` to return a pointer to, *nopage* cannot be used in these situations; you must use *remap_pfn_range* instead.

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 `0` and that is used, for example, to map the BSS segment. Any process referencing the zero page sees exactly that: a page filled with zeroes. If the process writes to the page, it ends up modifying a private copy. Therefore, if a process extends a mapped region by calling *mremap*, and the driver hasn't implemented *nopage*, the process ends up with zero-filled memory instead of a segmentation fault.

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 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 does 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_pfn_range(vma, 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 fails to define a *nopage* method, it is never notified of this extension, and the additional area maps 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 *type);
{ return NOPAGE_SIGBUS; /* send a SIGBUS */}
```

As we have seen, the *nopage* method is called only when the process dereferences an address that is within a known VMA but for which there is currently no valid page table entry. If we have used *remap_pfn_range* to map the entire device region, the *nopage* method shown here is called only for references outside of that region. Thus, it can safely return `NOPAGE_SIGBUS` to signal an error. Of course, a more thorough implementation of *nopage* could check to see whether the faulting address is within the device area, and perform the remapping if that is the case. Once again, however, *nopage* does not work with PCI memory areas, so extension of PCI mappings is not possible.

Remapping RAM

An interesting limitation of *remap_pfn_range* is that it gives access only to reserved pages and physical addresses above the top of physical memory. 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. 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_pfn_range* won't allow you to remap conventional addresses, which include the ones you obtain by calling *get_free_page*. Instead, it maps in the zero page. Everything appears to work, with the exception that the process sees private, zero-filled pages rather than the remapped RAM that it was hoping for. Nonetheless, the function does everything that most hardware drivers need it to do, because it can remap high PCI buffers and ISA memory.

The limitations of *remap_pfn_range* can be seen by running *mapper*, one of the sample programs in *misc-progs* in the files provided on O'Reilly's 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 specified by 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 example 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_pfn_range* to deal with RAM suggests that memory-based devices like *scull* 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 8.

scullp is a 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 in the section "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 `0` 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 (set at module load time) is `0`. The parameter controls how *__get_free_pages* is invoked (see the section "get_free_page and Friends" in Chapter 8). The zero-order limitation (which forces pages to be allocated one at a time, rather than in larger groups) is dictated by the internals of *__get_free_pages*, the allocation function used by *scullp*. To maximize allocation performance, the Linux kernel maintains a list of free pages for each allocation order, and only the reference 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. *scullp*

simply does not know how to properly manage reference counts for pages that are part of higher-order allocations. (Return to the section "A scull Using Whole Pages: scullp" in Chapter 8 if you need a refresher on *scullp* and the memory allocation order value.)

The zero-order limitation 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 the *open*, *close*, and *nopage* VMA methods; 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 = filp->f_dentry->d_inode;

    /* refuse to map if order is not 0 */
    if (scullp_devices[iminor(inode)].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 = filp->private_data;
    scullp_vma_open(vma);
    return 0;
}
```

The purpose of the `if` statement 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 count of active mappings for the device.

open and *close* simply keep track of the mapping count and are defined as follows:

```
void scullp_vma_open(struct vm_area_struct *vma)
{
    struct scullp_dev *dev = vma->vm_private_data;

    dev->vmas++;
}

void scullp_vma_close(struct vm_area_struct *vma)
{
    struct scullp_dev *dev = vma->vm_private_data;

    dev->vmas--;
}
```

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 *type)
{
    unsigned long offset;
    struct scullp_dev *ptr, *dev = vma->vm_private_data;
    struct page *page = NOPAGE_SIGBUS;
    void *pageptr = NULL; /* default to "missing" */

    down(&dev->sem);
    offset = (address - vma->vm_start) + (vma->vm_pgoff << PAGE_SHIFT);
    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);
    if (type)
        *type = VM_FAULT_MINOR;
  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 (0x00000000) to 140 (0x0000008c)
total 232
crw-------    1 root     root      10,  10 Sep 15 07:40 adbmouse
```

```
crw-r--r--    1 root     root      10, 175 Sep 15 07:40 agpgart
morgana% ./mapper /dev/scullp 8192 200
mapped "/dev/scullp" from 8192 (0x00002000) to 8392 (0x000020c8)
d0h1494
brw-rw----    1 root     floppy     2,  92 Sep 15 07:40 fd0h1660
brw-rw----    1 root     floppy     2,  20 Sep 15 07:40 fd0h360
brw-rw----    1 root     floppy     2,  12 Sep 15 07:40 fd0H360
```

Remapping Kernel Virtual Addresses

Although it's rarely necessary, it's interesting to see how a driver can map a kernel virtual address to user space using *mmap*. A true kernel virtual address, remember, is an address returned by a function such as *vmalloc*—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.

Beyond that, there is only one difference between the *nopage* implementations used by *scullp* and *scullv*. Remember that *scullp*, once it found the page of interest, would obtain the corresponding `struct page` pointer with *virt_to_page*. That function does not work with kernel virtual addresses, however. Instead, you must use *vmalloc_to_page*. So the final part of the *scullv* version of *nopage* looks like:

```
  /*
   * After scullv lookup, "page" is now the address of the page
   * needed by the current process. Since it's a vmalloc address,
   * turn it into a struct page.
   */
  page = vmalloc_to_page(pageptr);

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

Based on this discussion, you might also want to map addresses returned by *ioremap* to user space. That would be a mistake, however; addresses from *ioremap* are special and cannot be treated like normal kernel virtual addresses. Instead, you should use *remap_pfn_range* to remap I/O memory areas into user space.

Performing Direct I/O

Most I/O operations are buffered through the kernel. The use of a kernel-space buffer allows a degree of separation between user space and the actual device; this separation can make programming easier and can also yield performance benefits in many situations. There are cases, however, where it can be beneficial to perform I/O directly to or from a user-space buffer. If the amount of data being transferred is large, transferring data directly without an extra copy through kernel space can speed things up.

One example of direct I/O use in the 2.6 kernel is the SCSI tape driver. Streaming tapes can pass a lot of data through the system, and tape transfers are usually record-oriented, so there is little benefit to buffering data in the kernel. So, when the conditions are right (the user-space buffer is page-aligned, for example), the SCSI tape driver performs its I/O without copying the data.

That said, it is important to recognize that direct I/O does not always provide the performance boost that one might expect. The overhead of setting up direct I/O (which involves faulting in and pinning down the relevant user pages) can be significant, and the benefits of buffered I/O are lost. For example, the use of direct I/O requires that the *write* system call operate synchronously; otherwise the application does not know when it can reuse its I/O buffer. Stopping the application until each write completes can slow things down, which is why applications that use direct I/O often use asynchronous I/O operations as well.

The real moral of the story, in any case, is that implementing direct I/O in a char driver is usually unnecessary and can be hurtful. You should take that step only if you are sure that the overhead of buffered I/O is truly slowing things down. Note also that block and network drivers need not worry about implementing direct I/O at all; in both cases, higher-level code in the kernel sets up and makes use of direct I/O when it is indicated, and driver-level code need not even know that direct I/O is being performed.

The key to implementing direct I/O in the 2.6 kernel is a function called *get_user_pages*, which is declared in *<linux/mm.h>* with the following prototype:

```
int get_user_pages(struct task_struct *tsk,
                   struct mm_struct *mm,
                   unsigned long start,
                   int len,
                   int write,
                   int force,
                   struct page **pages,
                   struct vm_area_struct **vmas);
```

This function has several arguments:

`tsk`
: A pointer to the task performing the I/O; its main purpose is to tell the kernel who should be charged for any page faults incurred while setting up the buffer. This argument is almost always passed as `current`.

`mm`
: A pointer to the memory management structure describing the address space to be mapped. The `mm_struct` structure is the piece that ties together all of the parts (VMAs) of a process's virtual address space. For driver use, this argument should always be `current->mm`.

`start`
`len`
: `start` is the (page-aligned) address of the user-space buffer, and `len` is the length of the buffer in pages.

`write`
`force`
: If `write` is nonzero, the pages are mapped for write access (implying, of course, that user space is performing a read operation). The `force` flag tells *get_user_pages* to override the protections on the given pages to provide the requested access; drivers should always pass `0` here.

`pages`
`vmas`
: Output parameters. Upon successful completion, `pages` contain a list of pointers to the `struct page` structures describing the user-space buffer, and `vmas` contains pointers to the associated VMAs. The parameters should, obviously, point to arrays capable of holding at least `len` pointers. Either parameter can be `NULL`, but you need, at least, the `struct page` pointers to actually operate on the buffer.

get_user_pages is a low-level memory management function, with a suitably complex interface. It also requires that the mmap reader/writer semaphore for the address space be obtained in read mode before the call. As a result, calls to *get_user_pages* usually look something like:

```
down_read(&current->mm->mmap_sem);
result = get_user_pages(current, current->mm, ...);
up_read(&current->mm->mmap_sem);
```

The return value is the number of pages actually mapped, which could be fewer than the number requested (but greater than zero).

Upon successful completion, the caller has a `pages` array pointing to the user-space buffer, which is locked into memory. To operate on the buffer directly, the kernel-space code must turn each `struct page` pointer into a kernel virtual address with *kmap* or *kmap_atomic*. Usually, however, devices for which direct I/O is justified are using DMA operations, so your driver will probably want to create a scatter/gather list from the array of `struct page` pointers. We discuss how to do this in the section, "Scatter/gather mappings."

Once your direct I/O operation is complete, you must release the user pages. Before doing so, however, you must inform the kernel if you changed the contents of those pages. Otherwise, the kernel may think that the pages are "clean," meaning that they match a copy found on the swap device, and free them without writing them out to backing store. So, if you have changed the pages (in response to a user-space read request), you must mark each affected page dirty with a call to:

```
void SetPageDirty(struct page *page);
```

(This macro is defined in *<linux/page-flags.h>*). Most code that performs this operation checks first to ensure that the page is not in the reserved part of the memory map, which is never swapped out. Therefore, the code usually looks like:

```
if (! PageReserved(page))
    SetPageDirty(page);
```

Since user-space memory is not normally marked reserved, this check should not strictly be necessary, but when you are getting your hands dirty deep within the memory management subsystem, it is best to be thorough and careful.

Regardless of whether the pages have been changed, they must be freed from the page cache, or they stay there forever. The call to use is:

```
void page_cache_release(struct page *page);
```

This call should, of course, be made *after* the page has been marked dirty, if need be.

Asynchronous I/O

One of the new features added to the 2.6 kernel was the *asynchronous I/O* capability. Asynchronous I/O allows user space to initiate operations without waiting for their completion; thus, an application can do other processing while its I/O is in flight. A complex, high-performance application can also use asynchronous I/O to have multiple operations going at the same time.

The implementation of asynchronous I/O is optional, and very few driver authors bother; most devices do not benefit from this capability. As we will see in the coming chapters, block and network drivers are fully asynchronous at all times, so only char drivers are candidates for explicit asynchronous I/O support. A char device can benefit from this support if there are good reasons for having more than one I/O operation outstanding at any given time. One good example is streaming tape drives, where the drive can stall and slow down significantly if I/O operations do not arrive quickly enough. An application trying to get the best performance out of a streaming drive could use asynchronous I/O to have multiple operations ready to go at any given time.

For the rare driver author who needs to implement asynchronous I/O, we present a quick overview of how it works. We cover asynchronous I/O in this chapter, because its implementation almost always involves direct I/O operations as well (if you are buffering data in the kernel, you can usually implement asynchronous behavior without imposing the added complexity on user space).

Drivers supporting asynchronous I/O should include *<linux/aio.h>*. There are three *file_operations* methods for the implementation of asynchronous I/O:

```
ssize_t (*aio_read) (struct kiocb *iocb, char *buffer,
                     size_t count, loff_t offset);
ssize_t (*aio_write) (struct kiocb *iocb, const char *buffer,
                      size_t count, loff_t offset);
int (*aio_fsync) (struct kiocb *iocb, int datasync);
```

The *aio_fsync* operation is only of interest to filesystem code, so we do not discuss it further here. The other two, *aio_read* and *aio_write*, look very much like the regular *read* and *write* methods but with a couple of exceptions. One is that the `offset` parameter is passed by value; asynchronous operations never change the file position, so there is no reason to pass a pointer to it. These methods also take the `iocb` ("I/O control block") parameter, which we get to in a moment.

The purpose of the *aio_read* and *aio_write* methods is to initiate a read or write operation that may or may not be complete by the time they return. If it *is* possible to complete the operation immediately, the method should do so and return the usual status: the number of bytes transferred or a negative error code. Thus, if your driver has a *read* method called *my_read*, the following *aio_read* method is entirely correct (though rather pointless):

```
static ssize_t my_aio_read(struct kiocb *iocb, char *buffer,
                           ssize_t count, loff_t offset)
{
    return my_read(iocb->ki_filp, buffer, count, &offset);
}
```

Note that the `struct file` pointer is found in the `ki_filp` field of the `kiocb` structure.

If you support asynchronous I/O, you must be aware of the fact that the kernel can, on occasion, create "synchronous IOCBs." These are, essentially, asynchronous operations that must actually be executed synchronously. One may well wonder why things are done this way, but it's best to just do what the kernel asks. Synchronous operations are marked in the IOCB; your driver should query that status with:

```
int is_sync_kiocb(struct kiocb *iocb);
```

If this function returns a nonzero value, your driver must execute the operation synchronously.

In the end, however, the point of all this structure is to enable asynchronous operations. If your driver is able to initiate the operation (or, simply, to queue it until some future time when it can be executed), it must do two things: remember everything it needs to know about the operation, and return `-EIOCBQUEUED` to the caller. Remembering the operation information includes arranging access to the user-space buffer; once you return, you will not again have the opportunity to access that buffer while running in the context of the calling process. In general, that means you will likely have to set up a direct kernel mapping (with *get_user_pages*) or a DMA mapping. The `-EIOCBQUEUED` error code indicates that the operation is not yet complete, and its final status will be posted later.

When "later" comes, your driver must inform the kernel that the operation has completed. That is done with a call to *aio_complete*:

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

Here, `iocb` is the same IOCB that was initially passed to you, and `res` is the usual result status for the operation. `res2` is a second result code that will be returned to user space; most asynchronous I/O implementations pass `res2` as `0`. Once you call *aio_complete*, you should not touch the IOCB or user buffer again.

An asynchronous I/O example

The page-oriented *scullp* driver in the example source implements asynchronous I/O. The implementation is simple, but it is enough to show how asynchronous operations should be structured.

The *aio_read* and *aio_write* methods don't actually do much:

```
static ssize_t scullp_aio_read(struct kiocb *iocb, char *buf, size_t count,
        loff_t pos)
{
    return scullp_defer_op(0, iocb, buf, count, pos);
}

static ssize_t scullp_aio_write(struct kiocb *iocb, const char *buf,
        size_t count, loff_t pos)
{
    return scullp_defer_op(1, iocb, (char *) buf, count, pos);
}
```

These methods simply call a common function:

```
struct async_work {
    struct kiocb *iocb;
    int result;
    struct work_struct work;
};

static int scullp_defer_op(int write, struct kiocb *iocb, char *buf,
        size_t count, loff_t pos)
{
    struct async_work *stuff;
    int result;
```

```
        /* Copy now while we can access the buffer */
        if (write)
            result = scullp_write(iocb->ki_filp, buf, count, &pos);
        else
            result = scullp_read(iocb->ki_filp, buf, count, &pos);

        /* If this is a synchronous IOCB, we return our status now. */
        if (is_sync_kiocb(iocb))
            return result;

        /* Otherwise defer the completion for a few milliseconds. */
        stuff = kmalloc (sizeof (*stuff), GFP_KERNEL);
        if (stuff == NULL)
            return result; /* No memory, just complete now */
        stuff->iocb = iocb;
        stuff->result = result;
        INIT_WORK(&stuff->work, scullp_do_deferred_op, stuff);
        schedule_delayed_work(&stuff->work, HZ/100);
        return -EIOCBQUEUED;
    }
```

A more complete implementation would use *get_user_pages* to map the user buffer into kernel space. We chose to keep life simple by just copying over the data at the outset. Then a call is made to *is_sync_kiocb* to see if this operation must be completed synchronously; if so, the result status is returned, and we are done. Otherwise we remember the relevant information in a little structure, arrange for "completion" via a workqueue, and return `-EIOCBQUEUED`. At this point, control returns to user space.

Later on, the workqueue executes our completion function:

```
static void scullp_do_deferred_op(void *p)
{
    struct async_work *stuff = (struct async_work *) p;
    aio_complete(stuff->iocb, stuff->result, 0);
    kfree(stuff);
}
```

Here, it is simply a matter of calling *aio_complete* with our saved information. A real driver's asynchronous I/O implementation is somewhat more complicated, of course, but it follows this sort of structure.

Direct Memory Access

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 to involve the system processor. Use of this mechanism can greatly increase throughput to and from a device, because a great deal of computational overhead is eliminated.

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 into that buffer. 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.*

---

* There are, of course, exceptions to everything; see the section "Receive Interrupt Mitigation" in Chapter 17 for a demonstration of how high-performance network drivers are best implemented using polling.

Another relevant item introduced here is the DMA buffer. DMA requires device drivers 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 introduce a higher-level interface shortly, but it is still a good idea to understand the material presented here.

The main issue that arrises with DMA buffers is that, when they are bigger than one page, they 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 the section "SBus" in Chapter 12), 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 allocate their buffers only at runtime. (Chapter 8 introduced these techniques; the section "Obtaining Large Buffers" covered 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 is used for DMA operations; not all memory zones are suitable. In particular, high memory may not work for DMA on some systems and with some devices—the peripherals simply cannot work with addresses that high.

Most devices on modern buses can handle 32-bit addresses, meaning that normal memory allocations 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 24-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 24 bits is allocated. Alternatively, you can use the generic DMA layer (which we discuss shortly) to allocate buffers that work around your device's limitations.

Do-it-yourself allocation

We have seen how *get_free_pages* can allocate up to a few megabytes (as order can range up to `MAX_ORDER`, currently 11), but high-order requests are prone to fail even when the requested buffer is far 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 the section "Obtaining Large Buffers" in Chapter 8, 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 256 MB, the argument `mem=255M` keeps the kernel from using the top megabyte. Your module could later use the following code to gain access to such memory:

```
dmabuf = ioremap (0xFF00000 /* 255M */, 0x100000 /* 1M */);
```

The *allocator*, part of the sample code accompanying the book, offers a simple API to probe and manage such reserved RAM and has been used successfully on several architectures. However, this trick doesn't work when you have an high-memory system (i.e., one with more physical memory than could fit in the CPU address space).

Another option, of course, is to allocate your buffer with the `GFP_NOFAIL` allocation flag. This approach does, however, severely stress the memory management subsystem, and it runs the risk of locking up the system altogether; it is best avoided unless there is truly no other way.

If you are going to such lengths to allocate a large DMA buffer, however, it is worth putting some thought into alternatives. If your device can do scatter/gather I/O, you can allocate your buffer in smaller pieces and let the device do the rest. Scatter/gather I/O can also be used when performing direct I/O into user space, which may well be the best solution when a truly huge buffer is required.

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 bus 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.

---

* 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.

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*>. The use of these functions is strongly discouraged, because they work properly only on systems with a very simple I/O architecture; nonetheless, you may encounter them when working with kernel code.

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

These functions perform a simple conversion between kernel logical addresses and bus addresses. They do not work in any situation where an I/O memory management unit must be programmed or where bounce buffers must be used. The right way of performing this conversion is with the generic DMA layer, so we now move on to that topic.

The Generic DMA Layer

DMA operations, in the end, come down to allocating a buffer and passing bus addresses to your device. However, the task of writing portable drivers that perform DMA safely and correctly on all architectures is harder than one might think. Different systems have different ideas of how cache coherency should work; if you do not handle this issue correctly, your driver may corrupt memory. Some systems have complicated bus hardware that can make the DMA task easier—or harder. And not all systems can perform DMA out of all parts of memory. Fortunately, the kernel provides a bus- and architecture-independent DMA layer that hides most of these issues from the driver author. We strongly encourage you to use this layer for DMA operations in any driver you write.

Many of the functions below require a pointer to a `struct device`. This structure is the low-level representation of a device within the Linux device model. It is not something that drivers often have to work with directly, but you do need it when using the generic DMA layer. Usually, you can find this structure buried inside the bus specific that describes your device. For example, it can be found as the `dev` field in `struct pci_device` or `struct usb_device`. The `device` structure is covered in detail in Chapter 14.

Drivers that use the following functions should include <*linux/dma-mapping.h*>.

Dealing with difficult hardware

The first question that must be answered before attempting DMA is whether the given device is capable of such an operation on the current host. Many devices are limited in the range of memory they can address, for a number of reasons. By default, the kernel assumes that your device can perform DMA to any 32-bit address. If this is not the case, you should inform the kernel of that fact with a call to:

```
int dma_set_mask(struct device *dev, u64 mask);
```

The `mask` should show the bits that your device can address; if it is limited to 24 bits, for example, you would pass `mask` as `0x0FFFFFF`. The return value is nonzero if DMA is possible with the given `mask`; if *dma_set_mask* returns `0`, you are not able to use DMA operations with this device. Thus, the initialization code in a driver for a device limited to 24-bit DMA operations might look like:

```
if (dma_set_mask (dev, 0xffffff))
    card->use_dma = 1;
else {
    card->use_dma = 0;   /* We'll have to live without DMA */
    printk (KERN_WARN, "mydev: DMA not supported\n");
}
```

Again, if your device supports normal, 32-bit DMA operations, there is no need to call *dma_set_mask*.

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. It is tempting to get that address with a simple call to *virt_to_bus*, but there are strong reasons for avoiding that approach. The first of those is that reasonable hardware comes with an IOMMU that provides a set of *mapping registers* for the bus. The IOMMU can arrange for any physical memory to appear within the address range accessible by the device, and it can cause physically scattered buffers to look contiguous to the device. Making use of the IOMMU requires using the generic DMA layer; *virt_to_bus* is not up to the task.

Note that not all architectures have an IOMMU; in particular, the popular x86 platform has no IOMMU support. A properly written driver need not be aware of the I/O support hardware it is running over, however.

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. Needless to say, use of bounce buffers can slow things down, but sometimes there is no alternative.

DMA mappings must also address the issue of cache coherency. Remember that modern processors keep copies of recently accessed memory areas in a fast, local cache; without this cache, reasonable performance is not possible. If your device changes an area of main memory, it is imperative that any processor caches covering that area be invalidated; otherwise the processor may work with an incorrect image of main memory, and data corruption results. Similarly, when your device uses DMA to read data from main memory, any changes to that memory residing in processor caches must be flushed out first. These *cache coherency* issues can create no end of obscure and difficult-to-find bugs if the programmer is not careful. Some architectures manage cache coherency in the hardware, but others require software support. The generic DMA layer goes to great lengths to ensure that things work correctly on all architectures, but, as we will see, proper behavior requires adherence to a small set of rules.

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. As a bus address, `dma_addr_t` may lead to unexpected problems if used directly by the CPU.

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

Coherent DMA mappings
: These mappings usually exist for the life of the driver. A coherent 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). As a result, coherent mappings must live in cache-coherent memory. Coherent mappings can be expensive to set up and use.

Streaming DMA mappings
: Streaming mappings are usually set up for a single operation. Some architectures allow for significant optimizations when streaming mappings are used, as we 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 coherent mappings whenever possible. There are two reasons for this recommendation. The first is that, on systems that support mapping registers, each DMA mapping uses one or more of them on the bus. Coherent 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 coherent mappings.

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

Setting up coherent DMA mappings

A driver can set up a coherent mapping with a call to *dma_alloc_coherent*:

```
void *dma_alloc_coherent(struct device *dev, size_t size,
                         dma_addr_t *dma_handle, int flag);
```

This function handles both the allocation and the mapping of the buffer. The first two arguments are the device structure and the size of the buffer needed. The function returns the result of the DMA mapping in two places. The return value from the function is a kernel virtual address for the buffer, which may be used by the driver; the associated bus address, meanwhile, is returned in `dma_handle`. Allocation is handled in this function so that the buffer is 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). The `flag` argument is the usual `GFP_` value describing how the memory is to be allocated; it should usually be `GFP_KERNEL` (usually) or `GFP_ATOMIC` (when running in atomic context).

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

```
void dma_free_coherent(struct device *dev, size_t size,
                        void *vaddr, dma_addr_t dma_handle);
```

Note that this function, like many of the generic DMA functions, requires that all of the size, CPU address, and bus address arguments be provided.

DMA pools

A *DMA pool* is an allocation mechanism for small, coherent DMA mappings. Mappings obtained from *dma_alloc_coherent* may have a minimum size of one page. If your device needs smaller DMA areas than that, you should probably be using a DMA pool. DMA pools are also useful in situations where you may be tempted to perform DMA to small areas embedded within a larger structure. Some very obscure driver bugs have been traced down to cache coherency problems with structure fields adjacent to small DMA areas. To avoid this problem, you should always allocate areas for DMA operations explicitly, away from other, non-DMA data structures.

The DMA pool functions are defined in *<linux/dmapool.h>*.

A DMA pool must be created before use with a call to:

```
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
                                 size_t size, size_t align,
                                 size_t allocation);
```

Here, `name` is a name for the pool, `dev` is your device structure, `size` is the size of the buffers to be allocated from this pool, `align` is the required hardware alignment for allocations from the pool (expressed in bytes), and `allocation` is, if nonzero, a memory boundary that allocations should not exceed. If `allocation` is passed as 4096, for example, the buffers allocated from this pool do not cross 4-KB boundaries.

When you are done with a pool, it can be freed with:

```
void dma_pool_destroy(struct dma_pool *pool);
```

You should return all allocations to the pool before destroying it.

Allocations are handled with *dma_pool_alloc*:

```
void *dma_pool_alloc(struct dma_pool *pool, int mem_flags,
                     dma_addr_t *handle);
```

For this call, `mem_flags` is the usual set of `GFP_` allocation flags. If all goes well, a region of memory (of the size specified when the pool was created) is allocated and returned. As with *dma_alloc_coherent*, the address of the resulting DMA buffer is returned as a kernel virtual address and stored in `handle` as a bus address.

Unneeded buffers should be returned to the pool with:

```
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t addr);
```

Setting up streaming DMA mappings

Streaming mappings have a more complicated interface than the coherent variety, for a number of reasons. These mappings expect to work with a buffer that has already been allocated by the driver and, therefore, 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. For all of these reasons, streaming mappings have their own set of mapping functions.

When setting up a streaming mapping, you must tell the kernel in which direction the data is moving. Some symbols (of type `enum dma_data_direction`) have been defined for this purpose:

`DMA_TO_DEVICE`
`DMA_FROM_DEVICE`

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

`DMA_BIDIRECTIONAL`

> If data can move in either direction, use `DMA_BIDIRECTIONAL`.

`DMA_NONE`

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

It may be tempting to just pick `DMA_BIDIRECTIONAL` at all times, but driver authors should resist that temptation. On some architectures, there is a performance penalty to pay for that choice.

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

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size,
                          enum dma_data_direction 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 *dma_unmap_single*:

```
void dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
                      enum dma_data_direction direction);
```

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

Some important rules 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 *dma_unmap_single* has been called is it safe for the driver to access the contents of the buffer (with one exception that we 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 is in the processor's cache when *dma_unmap_single* is issued, 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 simply fail in this case, but others 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 `DMA_TO_DEVICE`, and a bounce buffer is required, the contents of the original buffer are copied as part of the mapping operation. Clearly, changes to the original buffer after the copy are not seen by the device. Similarly, `DMA_FROM_DEVICE` bounce buffers are copied back to the original buffer by *dma_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. `DMA_BIDIRECTIONAL` bounce buffers are copied both before and after the operation, which is often an unnecessary waste of CPU cycles.

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

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr,
                             size_t size, enum dma_data_direction direction);
```

This function should be called before the processor accesses a streaming DMA buffer. Once the call has been made, the CPU "owns" the DMA buffer and can work with it as needed. Before the device accesses the buffer, however, ownership should be transferred back to it with:

```
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
                                size_t size, enum dma_data_direction direction);
```

The processor, once again, should not access the DMA buffer after this call has been made.

Single-page streaming mappings

Occasionally, you may want to set up a mapping on a buffer for which you have a `struct page` pointer; this can happen, for example, with user-space buffers mapped with *get_user_pages*. To set up and tear down streaming mappings using `struct page` pointers, use the following:

```
dma_addr_t dma_map_page(struct device *dev, struct page *page,
                        unsigned long offset, size_t size,
                        enum dma_data_direction direction);

void dma_unmap_page(struct device *dev, dma_addr_t dma_address,
                    size_t size, enum dma_data_direction direction);
```

The `offset` and `size` arguments can be used to map part of a page. It is recommended, however, that partial-page mappings be avoided unless you are really sure of what you are doing. Mapping part of a page can lead to cache coherency problems if the allocation covers only part of a cache line; that, in turn, can lead to memory corruption and extremely difficult-to-debug bugs.

Scatter/gather mappings

Scatter/gather mappings are a special type of streaming DMA mapping. 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.

Many 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. 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 <*asm/scatterlist.h*>. However, it always contains three fields:

`struct page *page;`
: The `struct page` pointer corresponding to the buffer to be used in the scatter/gather operation.

`unsigned int length;`
`unsigned int offset;`
: The length of that buffer and its offset within the page

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

```
int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents,
               enum dma_data_direction direction)
```

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

For each buffer in the input scatterlist, *dma_map_sg* determines the proper bus address to give to the device. As part of that task, it also coalesces buffers that are adjacent to each other in memory. If the system your driver is running on has an I/O memory management unit, *dma_map_sg* also programs that unit's mapping registers, with the possible result that, from your device's point of view, you are able to transfer a single, contiguous buffer. You will never know what the resulting transfer will look like, however, until after the call.

Your driver should transfer each buffer returned by *pci_map_sg*. The bus address and length of each buffer are 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 *dma_map_sg*.

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

```
void dma_unmap_sg(struct device *dev, struct scatterlist *list,
                  int nents, enum dma_data_direction direction);
```

Note that `nents` must be the number of entries that you originally passed to *dma_map_sg* and not the number of DMA buffers the 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 dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg,
                         int nents, enum dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
                         int nents, enum dma_data_direction direction);
```

PCI double-address cycle mappings

Normally, the DMA support layer works with 32-bit bus addresses, possibly restricted by a specific device's DMA mask. The PCI bus, however, also supports a 64-bit addressing mode, the *double-address cycle* (DAC). The generic DMA layer does not support this mode for a couple of reasons, the first of which being that it is a PCI-specific feature. Also, many implementations of DAC are buggy at best, and, because DAC is slower than a regular, 32-bit DMA, there can be a performance cost. Even so, there are applications where using DAC can be the right thing to do; if you have a device that is likely to be working with very large buffers placed in high memory, you may want to consider implementing DAC support. This support is available only for the PCI bus, so PCI-specific routines must be used.

To use DAC, your driver must include <*linux/pci.h*>. You must set a separate DMA mask:

```
int pci_dac_set_dma_mask(struct pci_dev *pdev, u64 mask);
```

You can use DAC addressing only if this call returns `0`.

A special type (`dma64_addr_t`) is used for DAC mappings. To establish one of these mappings, call *pci_dac_page_to_dma*:

```
dma64_addr_t pci_dac_page_to_dma(struct pci_dev *pdev, struct page *page,
                                 unsigned long offset, int direction);
```

DAC mappings, you will notice, can be made only from `struct page` pointers (they should live in high memory, after all, or there is no point in using them); they must be created a single page at a time. The `direction` argument is the PCI equivalent of the `enum dma_data_direction` used in the generic DMA layer; it should be `PCI_DMA_TODEVICE`, `PCI_DMA_FROMDEVICE`, or `PCI_DMA_BIDIRECTIONAL`.

DAC mappings require no external resources, so there is no need to explicitly release them after use. It is necessary, however, to treat DAC mappings like other streaming mappings, and observe the rules regarding buffer ownership. There is a set of functions for synchronizing DMA buffers that is analogous to the generic variety:

```
void pci_dac_dma_sync_single_for_cpu(struct pci_dev *pdev,
                                     dma64_addr_t dma_addr,
                                     size_t len,
                                     int direction);
```

```
void pci_dac_dma_sync_single_for_device(struct pci_dev *pdev,
                                        dma64_addr_t dma_addr,
                                        size_t len,
                                        int direction);
```

A simple PCI DMA example

As an example of how the DMA mappings might be used, we present a simple example of DMA coding for a PCI device. 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;

    /* Map the buffer for DMA */
    dev->dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE);
    dev->dma_size = count;
    bus_addr = dma_map_single(&dev->pci_dev->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 looks 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 */
    dma_unmap_single(dev->pci_dev->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.

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 eight bits at a time and is there for backward compatibility.

The channels are numbered from 0–7: 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, 0–3 on the slave (the 8-bit channels) and 5–7 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 (because it transfers eight bits in one cycle) and 128 KB for the master (which does 16-bit transfers).

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);
        /* ... */
    }
```

Here's how the */proc/dma* 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 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 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; therefore, the return value is a set of flags describing the previous interrupt state; it must be passed to the following function to restore the interrupt state when you are done with the lock.

`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 consists 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 the section "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. (Otherwise, corruption can occur because the controller is programmed via 8-bit data transfers and, therefore, 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 whether 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 springs from the need to obtain the 16-bit residue through 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 eight 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;
}
```

Then, a function like the next one 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.

Quick Reference

This chapter introduced the following symbols related to memory handling.

Introductory Material

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

Most of the functions and structures related to memory management are prototyped and defined in these header files.

`void *__va(unsigned long physaddr);`
`unsigned long __pa(void *kaddr);`

Macros that convert between kernel logical addresses and physical addresses.

`PAGE_SIZE`
`PAGE_SHIFT`

Constants that give the size (in bytes) of a page on the underlying hardware and the number of bits that a page frame number must be shifted to turn it into a physical address.

`struct page`

Structure that represents a hardware page in the system memory map.

`struct page *virt_to_page(void *kaddr);`
`void *page_address(struct page *page);`
`struct page *pfn_to_page(int pfn);`

Macros that convert between kernel logical addresses and their associated memory map entries. *page_address* works only for low-memory pages or high-memory pages that have been explicitly mapped. *pfn_to_page* converts a page frame number to its associated `struct page` pointer.

`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/highmem.h>`
`#include <asm/kmap_types.h>`
`void *kmap_atomic(struct page *page, enum km_type type);`
`void kunmap_atomic(void *addr, enum km_type type);`
The high-performance version of *kmap*; the resulting mappings can be held only by atomic code. For drivers, `type` should be `KM_USER0`, `KM_USER1`, `KM_IRQ0`, or `KM_IRQ1`.

`struct vm_area_struct;`
Structure describing a VMA.

Implementing mmap

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_add,
  unsigned long pfn, unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma, unsigned long virt_add,
  unsigned long phys_add, unsigned long size, pgprot_t prot);
```

Functions that sit at the heart of *mmap*. They map `size` bytes of physical addresses, starting at the page number indicated by `pfn` to the virtual address `virt_add`. The protection bits associated with the virtual space are specified in `prot`. *io_remap_page_range* should be used when the target address is in I/O memory space.

`struct page *vmalloc_to_page(void *vmaddr);`
Converts a kernel virtual address obtained from *vmalloc* to its corresponding `struct page` pointer.

Implementing Direct I/O

```
int get_user_pages(struct task_struct *tsk, struct mm_struct *mm, unsigned
  long start, int len, int write, int force, struct page **pages, struct
  vm_area_struct **vmas);
```

Function that locks a user-space buffer into memory and returns the corresponding `struct page` pointers. The caller must hold `mm->mmap_sem`.

`SetPageDirty(struct page *page);`
Macro that marks the given page as "dirty" (modified) and in need of writing to its backing store before it can be freed.

`void page_cache_release(struct page *page);`
Frees the given page from the page cache.

```
int is_sync_kiocb(struct kiocb *iocb);
```

Macro that returns nonzero if the given IOCB requires synchronous execution.

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

Function that indicates completion of an asynchronous I/O operation.

Direct Memory Access

```
#include <asm/io.h>
unsigned long virt_to_bus(volatile void * address);
void * bus_to_virt(unsigned long address);
```

Obsolete and deprecated functions that convert between kernel, virtual, and bus addresses. Bus addresses must be used to talk to peripheral devices.

```
#include <linux/dma-mapping.h>
```

Header file required to define the generic DMA functions.

```
int dma_set_mask(struct device *dev, u64 mask);
```

For peripherals that cannot address the full 32-bit range, this function informs the kernel of the addressable range and returns nonzero if DMA is possible.

```
void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t
  *bus_addr, int flag)
void dma_free_coherent(struct device *dev, size_t size, void *cpuaddr,
  dma_handle_t bus_addr);
```

Allocate and free coherent DMA mappings for a buffer that will last the lifetime of the driver.

```
#include <linux/dmapool.h>
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
  size_t size, size_t align, size_t allocation);
void dma_pool_destroy(struct dma_pool *pool);
void *dma_pool_alloc(struct dma_pool *pool, int mem_flags, dma_addr_t
   *handle);
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t handle);
```

Functions that create, destroy, and use DMA pools to manage small DMA areas.

```
enum dma_data_direction;
DMA_TO_DEVICE
DMA_FROM_DEVICE
DMA_BIDIRECTIONAL
DMA_NONE
```

Symbols used to tell the streaming mapping functions the direction in which data is moving to or from the buffer.

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size, enum
  dma_data_direction direction);
void dma_unmap_single(struct device *dev, dma_addr_t bus_addr, size_t size,
  enum dma_data_direction direction);
```

Create and destroy a single-use, streaming DMA mapping.

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr, size_t
  size, enum dma_data_direction direction);
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
  size_t size, enum dma_data_direction direction);
```

Synchronizes a buffer that has a streaming mapping. These functions 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).

```
#include <asm/scatterlist.h>
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.

```
dma_map_sg(struct device *dev, struct scatterlist *list, int nents,
  enum dma_data_direction direction);
dma_unmap_sg(struct device *dev, struct scatterlist *list, int nents, enum
  dma_data_direction direction);
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg, int
  nents, enum dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg, int
  nents, enum dma_data_direction direction);
```

dma_map_sg maps a scatter/gather operation, and *dma_unmap_sg* undoes that mapping. If the buffers must be accessed while the mapping is active, *dma_sync_sg_** may be used to synchronize things.

```
/proc/dma
```

File that 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>
```

Header that 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);`

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);`

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);`

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.