第十五章 内存映射和DMA

本章深入探讨了Linux内存管理的领域，重点是对设备驱动程序编写者有用的技术。许多类型的驱动程序编程需要一些对虚拟内存子系统如何工作的理解；我们在本章中涵盖的材料在我们进入一些更复杂和性能关键的子系统时会多次派上用场。虚拟内存子系统也是核心Linux内核的一个非常有趣的部分，因此，它值得一看。

本章的材料分为三个部分：

• 第一部分涵盖了mmap系统调用的实现，它允许将设备内存直接映射到用户进程的地址空间。并非所有设备都需要mmap支持，但是，对于一些设备，映射设备内存可以带来显著的性能提升。

• 然后我们从另一个方向探讨了直接访问用户空间页面的问题。相对较少的驱动程序需要这种能力；在许多情况下，内核在驱动程序甚至没有意识到的情况下执行这种映射。但是，了解如何将用户空间内存映射到内核（使用get_user_pages）可能是有用的。

• 最后一部分涵盖了直接内存访问（DMA）I/O操作，这些操作为外设提供了直接访问系统内存的能力。

当然，所有这些技术都需要理解Linux内存管理是如何工作的，所以我们从对那个子系统的概述开始。

15.1. Linux 中的内存管理

这一部分并不是在描述操作系统中内存管理的理论，而是试图指出Linux实现的主要特性。虽然你不需要成为Linux虚拟内存专家就可以实现mmap，但是对事物工作方式的基本概述是有用的。接下来是对内核用于管理内存的数据结构的相当长的描述。一旦涵盖了必要的背景，我们就可以开始使用这些结构。

15.1.1. 地址类型

Linux当然是一个虚拟内存系统，这意味着用户程序看到的地址并不直接对应硬件使用的物理地址。虚拟内存引入了一个间接层，允许进行一些好的操作。有了虚拟内存，系统上运行的程序可以分配远超于物理可用的内存；实际上，即使是单个进程也可以拥有比系统物理内存大的虚拟地址空间。虚拟内存还允许程序对进程的地址空间进行一些操作，包括将程序的内存映射到设备内存。

到目前为止，我们已经讨论了虚拟和物理地址，但是许多细节被忽略了。Linux系统处理几种类型的地址，每种都有自己的语义。不幸的是，内核代码并不总是非常清楚在每种情况下使用的是哪种类型的地址，所以程序员必须小心。

以下是Linux中使用的地址类型的列表。图15-1显示了这些地址类型如何与物理内存相关。

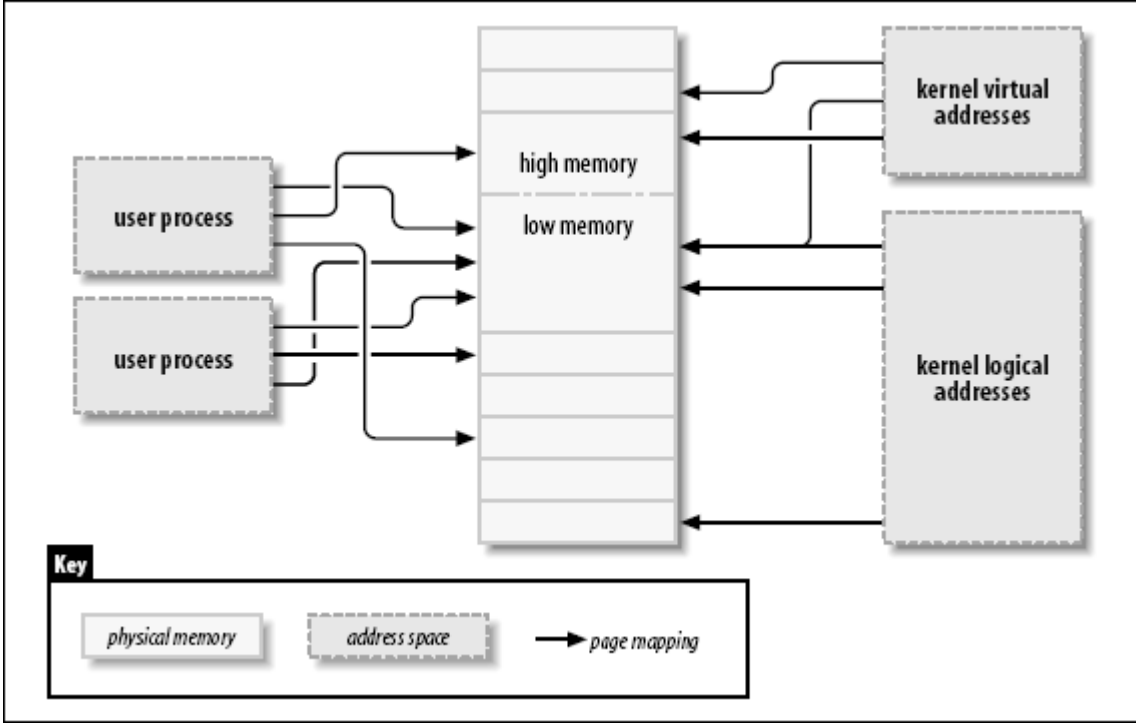
- 用户虚拟地址 这些是用户空间程序看到的常规地址。用户地址的长度是32位或64位，取决于底层硬件架构，每个进程都有自己的虚拟地址空间。

- 物理地址 处理器和系统内存之间使用的地址。物理地址是32位或64位的量；即使是32位系统在某些情况下也可以使用更大的物理地址。

- 总线地址 外设总线和内存之间使用的地址。通常，它们与处理器使用的物理地址相同，但这并不一定是这样。一些架构可以提供一个I/O内存管理单元（IOMMU），在总线和主内存之间重新映射地址。IOMMU可以在许多方面使生活变得更轻松（例如，使在内存中分散的缓冲区对设备看起来是连续的），但是编程IOMMU是设置DMA操作时必须执行的额外步骤。总线地址当然是高度依赖于架构的。

- 内核逻辑地址 这些构成了内核的正常地址空间。这些地址映射了主内存的一部分（可能是全部），并且通常被视为如果它们是物理地址。在大多数架构上，逻辑地址和它们相关的物理地址只相差一个常数偏移。逻辑地址使用硬件的本机指针大小，因此，在装备丰富的32位系统上可能无法寻址所有的物理内存。逻辑地址通常存储在unsigned long或`void *`类型的变量中。从kmalloc返回的内存有一个内核逻辑地址。

- 内核虚拟地址 内核虚拟地址与逻辑地址相似，因为它们是从内核空间地址到物理地址的映射。然而，内核虚拟地址并不一定具有逻辑地址空间特有的线性、一对一的物理地址映射。所有的逻辑地址都是内核虚拟地址，但是许多内核虚拟地址不是逻辑地址。例如，由vmalloc分配的内存有一个虚拟地址（但没有直接的物理映射）。kmap函数（在本章后面描述）也返回虚拟地址。虚拟地址通常存储在指针变量中。



如果你有一个逻辑地址，宏 `__pa()` （在<asm/page.h>中定义）返回其关联的物理地址。物理地址可以通过 `__va()` 映射回逻辑地址，但只适用于低内存页面。

不同的内核函数需要不同类型的地址。如果有不同的C类型定义，使得所需的地址类型明确，那就好了，但我们没有这样的运气。在这一章，我们试图明确在哪里使用哪种类型的地址。

15.1.2. 物理地址和页

物理内存被划分为称为页面的离散单元。系统的大部分内部内存处理都是以每页为基础进行的。页面大小从一个架构到另一个架构有所不同，尽管大多数系统目前使用4096字节的页面。常量PAGE_SIZE（在<asm/page.h>中定义）给出了任何给定架构的页面大小。

如果你看一个内存地址——虚拟的或物理的——它可以被划分为一个页面号和页面内的偏移。例如，如果使用4096字节的页面，那么最不重要的12位是偏移，剩下的更高位表示页面号。如果你丢弃偏移并将地址的其余部分向右移动，结果被称为页面帧号（PFN）。转换页面帧号和地址的位移是一个相当常见的操作；宏PAGE_SHIFT告诉你必须移动多少位来进行这种转换。

15.1.3. 高和低内存High and Low Memory

在配备大量内存的32位系统上，逻辑地址和内核虚拟地址之间的区别被突出显示。使用32位，可以寻址4GB的内存。然而，由于虚拟地址空间的设置方式，Linux在32位系统上直到最近才被限制到远小于这个数量的内存。

内核（在x86架构上，在默认配置中）将4GB的虚拟地址空间在用户空间和内核之间分割；在两种上下文中都使用相同的映射。一个典型的分割将3GB分配给用户空间，1GB用于内核空间。内核的代码和数据结构必须适应这个空间，但是内核地址空间的最大消费者是物理内存的虚拟映射。内核不能直接操作没有映射到内核地址空间的内存。换句话说，内核需要自己的虚拟地址来直接触摸任何内存。因此，多年来，内核可以处理的物理内存的最大量是可以映射到内核虚拟地址空间部分的量，减去内核代码本身所需的空间。结果是，基于x86的Linux系统可以使用最多接近1GB的物理内存。

为了应对商业压力，支持更多的内存，同时不破坏32位应用程序和系统的兼容性，处理器制造商已经在他们的产品中添加了“地址扩展”功能。结果是，在许多情况下，即使是32位处理器也可以寻址超过4GB的物理内存。然而，可以直接用逻辑地址映射多少内存的限制仍然存在。只有最低部分的内存（最多1或2GB，取决于硬件和内核配置）有逻辑地址；其余的（高内存）没有。在访问特定的高内存页面之前，内核必须设置一个明确的虚拟映射，使该页面在内核的地址空间中可用。因此，许多内核数据结构必须放在低内存中；高内存往往被保留给用户空间进程页面。

“高内存”这个术语对一些人来说可能会令人困惑，尤其是因为它在PC世界中有其他的含义。所以，为了让事情清楚，我们在这里定义这些术语： 低内存 内核空间中存在逻辑地址的内存。在你可能遇到的几乎所有系统上，所有的内存都是低内存。 高内存 不存在逻辑地址的内存，因为它超出了为内核虚拟地址预留的地址范围。 在i386系统上，低内存和高内存之间的边界通常设置在刚刚低于1GB，尽管那个边界可以在内核配置时改变。这个边界与原始PC上找到的旧的640KB限制没有任何关系，其位置也不受硬件的指定。相反，这是由内核自身在内核和用户空间之间分割32位地址空间时设置的限制。我们将在本章中指出对高内存使用的限制。

15.1.4. 内存映射和 struct page

历史上，内核使用逻辑地址来引用物理内存的页面。然而，高内存支持的添加暴露了这种方法的一个明显问题——逻辑地址对于高内存来说是不可用的。因此，处理内存的内核函数越来越多地使用指向struct page（在<linux/mm.h>中定义）的指针。这个数据结构用于跟踪内核需要知道的关于物理内存的几乎所有事情——每个系统的物理页面都有一个struct page。这个结构的一些字段包括以下内容：

- `atomic_t count;` 对这个页面的引用数量。当计数降到0时，页面被返回到空闲列表。
- `void *virtual;` 如果页面被映射，那么这是页面的内核虚拟地址；否则，为NULL。低内存页面总是被映射；高内存页面通常不被映射。这个字段并不在所有的架构上出现；它通常只在页面的内核虚拟地址不能被轻易计算的地方编译。如果你想看这个字段，正确的方法是使用下面描述的page_address宏。

- `unsigned long flags;` 一组描述页面状态的位标志。这些包括PG_locked，表示页面已经被锁定在内存中，和PG_reserved，阻止内存管理系统完全处理页面。

在struct page中还有更多的信息，但它是内存管理的更深层的黑魔法的一部分，对驱动程序编写者来说并不关心。

内核维护一个或多个struct page条目的数组，跟踪系统上的所有物理内存。在一些系统上，有一个叫做mem_map的单一数组。然而，在一些系统上，情况更复杂。非均匀内存访问（NUMA）系统和那些具有广泛不连续物理内存的系统可能有多个内存映射数组，所以打算移植的代码应该尽可能避免直接访问数组。幸运的是，通常很容易只使用struct page指针，而不用担心它们来自哪里。

一些函数和宏被定义为在struct page指针和虚拟地址之间进行转换：

```c
struct page *virt_to_page(void *kaddr);
```

这个宏在<asm/page.h>中定义，接受一个内核逻辑地址并返回其关联的struct page指针。由于它需要一个逻辑地址，所以它不能与vmalloc或高内存的内存一起工作。

```c
struct page *pfn_to_page(int pfn);
```

返回给定页面帧号的struct page指针。如果需要，它会用pfn_valid检查一个页面帧号的有效性，然后将其传递给pfn_to_page。

```c
void *page_address(struct page *page);
```

如果存在这样的地址，返回此页面的内核虚拟地址。对于高内存，只有当页面被映射时，该地址才存在。这个函数在<linux/mm.h>中定义。在大多数情况下，你想使用kmap的版本，而不是page_address。

```c
#include <linux/highmem.h>

void *kmap(struct page *page);

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

kmap返回系统中任何页面的内核虚拟地址。对于低内存页面，它只返回页面的逻辑地址；对于高内存页面，kmap在内核地址空间的专用部分创建一个特殊的映射。应该总是用kunmap释放用kmap创建的映射；这样的映射有限，所以最好不要长时间持有它们。kmap调用维护一个计数器，所以如果两个或更多的函数都在同一个页面上调用kmap，会发生正确的事情。还要注意，如果没有映射可用，kmap可以睡眠。

```c
#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是kmap的高性能形式。每个架构都维护一个小的插槽列表（专用的页面表条目）用于原子kmaps；kmap_atomic的调用者必须告诉系统在type参数中使用哪个插槽。对于驱动程序来说，唯一有意义的插槽是KM_USER0和KM_USER1（对于直接从用户空间的调用运行的代码），以及KM_IRQ0和KM_IRQ1（对于中断处理程序）。注意，原子kmaps必须被原子地处理；你的代码不能在持有一个的同时睡眠。还要注意，内核中没有什么可以阻止两个函数试图使用同一个插槽并相互干扰（尽管每个CPU都有一组唯一的插槽）。在实践中，对原子kmap插槽的争用似乎不是问题。

当我们在本章后面和后续章节的示例代码中看到这些函数的一些用途时。

15.1.5. 页表

在任何现代系统上，处理器必须有一种机制来将虚拟地址转换为其对应的物理地址。这种机制被称为页表；它本质上是一个多级树形结构的数组，包含虚拟到物理的映射和一

些相关的标志。即使在不直接使用这样的表的架构上，Linux内核也维护一组页表。

设备驱动程序常常需要执行一些涉及操作页表的操作。幸运的是，对于驱动程序作者来说，2.6内核已经消除了直接操作页表的任何需要。因此，我们不会详细描述它们；好奇的读者可能会想看看Daniel P. Bovet和Marco Cesati的《理解Linux内核》（O'Reilly）以获取完整的故事。

15.1.6. 虚拟内存区Virtual Memory Areas

虚拟内存区域（VMA）是内核用来管理进程地址空间的不同区域的数据结构。VMA代表进程虚拟内存中的一个同质区域：一段连续的虚拟地址范围，它们具有相同的权限标志，并由同一个对象（例如，一个文件或交换空间）支持。它大致对应于“段”的概念，尽管它更好地被描述为“具有自己属性的内存对象”。进程的内存映射由（至少）以下区域组成：

- 一个用于程序的可执行代码的区域（通常称为文本）
- 多个用于数据的区域，包括初始化的数据（执行开始时有明确赋值的数据）、未初始化的数据（BSS）和程序堆栈
- 每个活动内存映射的一个区域

通过查看/proc//maps（其中pid当然是被进程ID替换的）可以看到进程的内存区域。/proc/self是/proc/pid的一个特例，因为它总是指向当前进程。作为示例，这里有一些内存映射（我们在斜体中添加了简短的注释）：

```
# cat /proc/1/maps look at init
08048000-0804e000 r-xp 00000000 03:01 64652
0804e000-0804f000 rw-p 00006000 03:01 64652
0804f000-08053000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 03:01 96278
40015000-40016000 rw-p 00014000 03:01 96278
40016000-40017000 rw-p 00000000 00:00 0
42000000-4212e000 r-xp 00000000 03:01 80290
4212e000-42131000 rw-p 0012e000 03:01 80290
42131000-42133000 rw-p 00000000 00:00 0
bffff000-c0000000 rwxp 00000000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0

/sbin/init text /sbin/init data zero-mapped BSS /lib/ld-
2.3.2.so text /lib/ld-2.3.2.so data BSS for ld.so
/lib/tls/libc-2.3.2.so text /lib/tls/libc-2.3.2.so data
BSS for libc Stack segment 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
```

每行中的字段是：

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

/proc/*/maps中的每个字段（除了图像名称）都对应于struct vm_area_struct中的一个字段：

- start end 这个内存区域的开始和结束虚拟地址。
- perm 一个位掩码，带有内存区域的读、写和执行权限。这个字段描述了进程被允许对属于该区域的页面做什么。字段的最后一个字符是p，表示“私有”，或者s，表示“共享”。
- offset 内存区域在其映射到的文件中的开始位置。偏移量为0意味着内存区域的开始对应于文件的开始。
- major minor 持有已映射文件的设备的主要和次要编号。令人困惑的是，对于设备映射，主要和次要编号指的是持有用户打开的设备特殊文件的磁盘分区，而不是设备本身。
- inode 映射文件的inode号。
- image 已映射的文件（通常是一个可执行图像）的名称。

15.1.6.1. vm_area_struct

当用户空间进程调用mmap将设备内存映射到其地址空间时，系统会创建一个新的VMA来表示该映射。支持mmap的驱动程序（因此，实现了mmap方法）需要通过完成该VMA的初始化来帮助该进程。因此，驱动程序编写者应至少对VMA有最基本的理解，以便支持mmap。

让我们看一下struct vm_area_struct（在<linux/mm.h>中定义）中最重要的字段。设备驱动程序在其mmap实现中可能会使用这些字段。请注意，内核维护VMA的列表和树以优化区域查找，并且vm_area_struct的几个字段用于维护这种组织。因此，驱动程序不能随意创建VMA，否则结构会破裂。VMA的主要字段如下（注意这些字段和我们刚刚看到的/proc输出之间的相似性）：

```c
unsigned long vm_start; unsigned long vm_end;
```

这个VMA覆盖的虚拟地址范围。这些字段是`/proc/*/maps`中显示的前两个字段。

```c
struct file *vm_file;
```

指向与此区域关联的struct file结构的指针（如果有的话）。

```c
unsigned long vm_pgoff;
```

区域在文件中的偏移量，以页面为单位。当文件或设备被映射时，这是在此区域中映射的第一页的文件位置。

```c
unsigned long vm_flags;
```

描述此区域的一组标志。对设备驱动程序编写者最感兴趣的标志是VM_IO和VM_RESERVED。VM_IO将VMA标记为内存映射的I/O区域。除其他事项外，VM_IO标志阻止该区域被包含在进程核心转储中。VM_RESERVED告诉内存管理系统不要试图交换出这个VMA；在大多数设备映射中，它应该被设置。

```c
struct vm_operations_struct *vm_ops;
```

内核可能调用的一组函数，用于操作这个内存区域。它的存在表明内存区域是一个内核“对象”，就像我们在整个书中一直使用的struct file。

```c
void *vm_private_data;
```

驱动程序可以使用的字段，用于存储其自己的信息。

像struct vm_area_struct一样，vm_operations_struct在<linux/mm.h>中定义；它包括下面列出的操作。这些操作是处理进程内存需求所需要的唯一操作，它们按照声明的顺序列出。在本章后面，这些函数中的一些将被实现。

```c
void (*open)(struct vm_area_struct *vma);
```

open方法由内核调用，以允许实现VMA的子系统初始化区域。每次对VMA进行新的引用时（例如，当进程fork时），都会调用此方法。唯一的例外是当VMA首次由mmap创建时；在这种情况下，会调用驱动程序的mmap方法。

```c
void (*close)(struct vm_area_struct *vma);
```

当一个区域被销毁时，内核调用其close操作。注意，VMA没有关联的使用计数；每个使用它的进程都会准确地打开和关闭该区域一次。

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

当进程试图访问属于有效VMA的页面，但当前不在内存中时，会调用nopage方法（如果定义了的话）对相关区域。该方法在可能从二级存储中读取后，返回物理页面的struct page指针。如果没有为区域定义nopage方法，内核会分配一个空页面。

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

此方法允许内核在用户空间访问页面之前将页面“预先故障”到内存中。通常，驱动程序没有必要实现populate方法。

15.1.7. 进程内存映射

内存管理谜题的最后一部分是进程内存映射结构，它将所有其他数据结构绑定在一起。系统中的每个进程（除了一些内核空间的辅助线程）都有一个struct mm_struct（在<linux/sched.h>中定义），它包含进程的虚拟内存区域列表、页表和各种其他内存管理清理信息，以及一个信号量（mmap_sem）和一个自旋锁（page_table_lock）。这个结构的指针在任务结构中可以找到；在驱动程序需要访问它的罕见情况下，通常的方法是使用current→mm。注意，内存管理结构可以在进程之间共享；例如，Linux的线程实现就是这样工作的。

这就结束了我们对Linux内存管理数据结构的概述。有了这个，我们现在可以继续实现mmap系统调用。

15.2. mmap 设备操作

内存映射是现代Unix系统最有趣的特性之一。就驱动程序而言，可以实现内存映射，以便用户程序直接访问设备内存。

可以通过查看X Window系统服务器的虚拟内存区域的子集来看到mmap使用的明确示例：

```
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

...
```

X服务器的VMA完整列表很长，但大多数条目在这里并不感兴趣。然而，我们确实看到了/dev/mem的四个单独映射，这些映射让我们对X服务器如何与视频卡工作有了一些了解。第一个映射在a0000，这是640-KB ISA空洞中视频RAM的标准位置。再往下看，我们看到在e8000000有一个大的映射，这个地址在系统的最高RAM地址之上。这是对适配器上视频内存的直接映射。

这些区域也可以在/proc/iomem中看到：

```
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
```

映射设备意味着将用户空间地址的范围与设备内存关联起来。每当程序在分配的地址范围内读取或写入时，它实际上是在访问设备。在X服务器的例子中，使用mmap允许快速和容易地访问视频卡的内存。对于像这样的性能关键应用，直接访问可以产生很大的差异。

你可能会怀疑，不是每个设备都适合mmap抽象；例如，对于串行端口和其他流向设备，这是没有意义的。mmap的另一个限制是映射是PAGE_SIZE粒度的。内核只能在页表级别管理虚拟地址；因此，映射区域必须是PAGE_SIZE的倍数，并且必须在物理内存中开始，地址是PAGE_SIZE的倍数。如果区域的大小不是页大小的倍数，内核通过使区域稍微变大来强制大小粒度。

这些限制对于驱动程序来说并不是一个大的约束，因为访问设备的程序无论如何都是依赖于设备的。由于程序必须了解设备的工作方式，程序员并不会过分困扰于需要注意像页对齐这样的细节。当在一些非x86平台上使用ISA设备时，存在一个更大的约束，因为它们的硬件视图可能不是连续的。例如，一些Alpha计算机将ISA内存视为一组散布的8位、16位或32位项目，没有直接映射。在这种情况下，你根本不能使用mmap。

无法直接将ISA地址映射到Alpha地址是由于两个系统的数据传输规范不兼容。虽然早期的Alpha处理器只能发出32位和64位的内存访问，但

ISA只能进行8位和16位的传输，而且没有办法将一个协议透明地映射到另一个协议上。

- 总的来说，内存映射是一个强大的工具，可以让用户空间程序直接访问设备内存，从而提高性能。然而，它并不适用于所有类型的设备，特别是那些不支持页对齐的设备，或者那些在硬件级别上不支持连续内存访问的设备。在这些情况下，驱动程序可能需要采用其他方法来实现用户空间程序和设备内存之间的交互。

当可行的时候，使用mmap有明显的优势。例如，我们已经看过X服务器，它将大量数据传输到视频内存和从视频内存中传出；将图形显示映射到用户空间可以显著提高吞吐量，而不是使用lseek/write实现。另一个典型的例子是控制PCI设备的程序。大多数PCI外设将其控制寄存器映射到一个内存地址，高性能应用程序可能更喜欢直接访问寄存器，而不是反复调用ioctl来完成工作。

mmap方法是file_operations结构的一部分，当发出mmap系统调用时会被调用。使用mmap，内核在实际方法被调用之前会做大量的工作，因此，方法的原型与系统调用的原型相当不同。这与ioctl和poll等调用不同，在调用方法之前，内核不会做太多的工作。

系统调用如下声明（如mmap(2)手册页所述）：

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

另一方面，文件操作声明为：

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

方法中的filp参数与第3章中介绍的相同，而vma包含用于访问设备的虚拟地址范围的信息。因此，内核已经完成了大部分工作；为了实现mmap，驱动程序只需要为地址范围构建适当的页表，如果需要，用新的操作集替换vma→vm_ops。

构建页表有两种方式：一次性使用名为remap_pfn_range的函数完成，或者通过nopage VMA方法一次一页地完成。每种方法都有其优点和限制。我们从“一次性”方法开始，这种方法更简单。从那里，我们添加了实际实现所需的复杂性。

15.2.1. 使用 remap_pfn_range

构建新的页表以映射一系列物理地址的任务由remap_pfn_range和io_remap_page_range处理，它们具有以下原型：

```c
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);
```

函数返回的值通常是0或一个负的错误代码。让我们看看函数参数的确切含义：

- vma 正在映射页范围的虚拟内存区域。
- virt_addr 重新映射应该开始的用户虚拟地址。函数为virt_addr和virt_addr+size之间的虚拟地址范围构建页表。
- pfn 对应于虚拟地址应映射到的物理地址的页帧号。页帧号就是物理地址右移PAGE_SHIFT位的结果。对于大多数用途，VMA结构的vm_pgoff字段包含你需要的确切值。函数影响从(pfn<<PAGE_SHIFT)到(pfn<<PAGE_SHIFT)+size的物理地址。
- size 正在重新映射的区域的尺寸，以字节为单位。
- prot 为新的VMA请求的“保护”。驱动程序可以（也应该）使用vma->vm_page_prot中找到的值。

remap_pfn_range的参数相当直接，当你的mmap方法被调用时，它们中的大多数已经在VMA中为你提供了。然而，你可能会想知道为什么有两个函数。第一个（remap_pfn_range）是为了处理pfn指向实际系统RAM的情况，而当phys_addr指向I/O内存时，应该使用io_remap_page_range。在实践中，这两个函数在SPARC之外的所有架构上都是相同的，你会看到在大多数情况下都使用remap_pfn_range。然而，为了编写可移植的驱动程序，你应该使用适合你特定情况的remap_pfn_range的变体。

另一个复杂问题与缓存有关：通常，对设备内存的引用不应由处理器缓存。通常系统BIOS会正确设置，但也可以通过保护字段禁用特定VMA的缓存。不幸的是，禁用这个级别的缓存高度依赖处理器。好奇的读者可能希望看看drivers/char/mem.c中的pgprot_noncached函数，看看涉及什么。我们在这里不再讨论这个话题。

15.2.2. 一个简单的实现

如果你的驱动程序需要将设备内存简单、线性地映射到用户地址空间，那么remap_pfn_range几乎是你真正需要做这项工作的全部内容。以下代码源自drivers/char/mem.c，展示了在一个典型的模块simple（Simple Implementation Mapping Pages with Little Enthusiasm）中如何执行这项任务：

```c
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;

}
```

如你所见，重新映射内存只是调用remap_pfn_range来创建必要的页表的问题。

15.2.3. 添加 VMA 的操作

如我们所见，vm_area_struct结构包含一组可以应用于VMA的操作。现在我们以一种简单的方式提供这些操作。特别是，我们为我们的VMA提供了打开和关闭操作。这些操作在进程打开或关闭VMA时被调用；特别是，每当进程fork并创建一个新的VMA引用时，就会调用open方法。除了内核执行的处理之外，还会调用打开和关闭VMA的方法，所以它们不需要重新实现那里完成的任何工作。它们存在的目的是为了让驱动程序进行任何它们可能需要的额外处理。

事实证明，像simple这样的简单驱动程序并不需要进行任何特别的额外处理。所以我们创建了打开和关闭方法，它们向系统日志打印一条消息，通知世界它们已经被调用。这不是特别有用，但它确实让我们展示了如何提供这些方法，以及何时调用它们。

为此，我们用调用printk的操作覆盖默认的vma→vm_ops：

```c
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,

};
```

为了使这些操作对特定的映射生效，需要在相关VMA的vm_ops字段中存储一个指向simple_remap_vm_ops的指针。这通常在mmap方法中完成。如果你回头看simple_remap_mmap示例，你会看到这些代码行：

```c
vma->vm_ops = &simple_remap_vm_ops;

simple_vma_open(vma);
```

注意对simple_vma_open的明确调用。由于在初始mmap上不调用open方法，所以如果我们想让它运行，我们必须明确调用它。

15.2.4. 使用 nopage 映射内存

虽然remap_pfn_range对许多（如果不是大多数）驱动程序的mmap实现都很有效，但有时候需要更灵活一些。在这种情况下，可能需要使用nopage VMA方法的实现。

nopage方法在一种情况下很有用，那就是通过mremap系统调用引起的情况，应用程序使用mremap来改变映射区域的边界地址。事实上，当通过mremap改变映射的VMA时，内核并不直接通知驱动程序。如果VMA的大小减小，内核可以在不告诉驱动程序的情况下静静地清除不需要的页面。相反，如果VMA扩大，驱动程序最终会通过调用nopage来找出新页面的映射必须设置，所以不需要进行单独的通知。因此，如果你想支持mremap系统调用，必须实现nopage方法。这里，我们展示了一个简单设备的nopage的简单实现。

记住，nopage方法有以下原型：

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

当用户进程试图访问内存中不存在的VMA中的页面时，会调用关联的nopage函数。address参数包含引起故障的虚拟地址，向下舍入到页面的开始。nopage函数必须定位并返回指向用户想要的页面的struct page指针。此函数还必须通过调用get_page宏来增加它返回的页面的使用计数：

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

这一步是必要的，以保持映射页面上的引用计数正确。内核为每个页面维护这个计数；当计数变为0时，内核知道页面可以放在空闲列表上。当一个VMA被取消映射时，内核会减少区域中每个页面的使用计数。如果你的驱动程序在向区域添加页面时不增加计数，使用计数会过早地变为0，系统的完整性就会受到威胁。

nopage方法还应该在type参数指向的位置存储故障的类型——但只有当该参数不为NULL时。在设备驱动程序中，type的适当值总是VM_FAULT_MINOR。

如果你正在使用nopage，那么当调用mmap时通常只需要做很少的工作；我们的版本看起来像这样：

```c
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;

}
```

mmap需要做的主要事情是用我们自己的操作替换默认的（NULL）vm_ops指针。然后nopage方法负责“重新映射”一页一页的，并返回其struct page结构的地址。因为我们只是在这里实现一个对物理内存的窗口，所以重新映射步骤很简单：我们只需要定位并返回所需地址的struct page的指针。我们的nopage方法看起来像下面这样：

```c
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;
}
```

再次强调，我们在这里只是映射主内存，nopage函数只需要找到故障地址的正确的struct page并增加其引用计数。因此，所需的事件序列是计算所需的物理地址，并通过右移PAGE_SHIFT位将其转换为页框号。由于用户空间可以给我们任何它喜欢的地

址，我们必须确保我们有一个有效的页框；pfn_valid函数为我们做了这个。如果地址超出范围，我们返回NOPAGE_SIGBUS，这会导致向调用进程发送总线信号。

否则，pfn_to_page会获取必要的struct page指针；我们可以增加它的引用计数（通过调用get_page）并返回它。

nopage方法通常返回一个指向struct page的指针。如果由于某种原因无法返回正常页面（例如，请求的地址超出了设备的内存区域），可以返回NOPAGE_SIGBUS来表示错误；这就是上面的简单代码所做的。nopage也可以返回NOPAGE_OOM来表示由资源限制引起的失败。

注意，这个实现适用于ISA内存区域，但不适用于PCI总线上的区域。PCI内存映射在最高系统内存之上，系统内存映射中没有这些地址的条目。因为没有struct page可以返回指针，所以在这些情况下不能使用nopage；你必须使用remap_pfn_range代替。

如果nopage方法被留空，处理页面故障的内核代码将零页面映射到故障的虚拟地址。零页面是一个读取为0的写时复制页面，例如，用于映射BSS段。任何引用零页面的进程都会看到完全一样的内容：一个充满零的页面。如果进程写入页面，它最终会修改一个私有副本。因此，如果一个进程通过调用mremap扩展了一个映射区域，并且驱动程序没有实现nopage，那么进程最终会得到充满零的内存，而不是一个段错误。

15.2.5. 重新映射特定 I/O 区

我们迄今为止看到的所有例子都是对/dev/mem的重新实现；它们将物理地址重新映射到用户空间。然而，典型的驱动程序只想映射适用于其外围设备的小地址范围，而不是所有的内存。为了只映射整个内存范围的一个子集到用户空间，驱动程序只需要玩弄偏移量。以下对于驱动程序映射一个区域的简单_region_size字节，开始于物理地址simple_region_start（应该是页对齐的）就可以做到：

```c
unsigned long off = vma->vm_pgoff << PAGE_SHIFT;

unsigned long pfn = page_to_pfn(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, pfn, vsize, vma-
>vm_page_prot);
```

除了计算偏移量，这段代码引入了一个检查，当程序试图映射比目标设备的I/O区域中可用的内存更多的内存时，报告错误。在这段代码中，psize是在指定偏移量后剩余的物理I/O大小，vsize是请求的虚拟内存大小；函数拒绝映射超出允许的内存范围的地址。

注意，用户进程总是可以使用mremap来扩展其映射，可能超过物理设备区域的末尾。如果你的驱动程序没有定义nopage方法，它永远不会被通知这个扩展，额外的区域映射到零页面。作为驱动程序的编写者，你可能希望防止这种行为；将零页面映射到你的区域的末尾并不是一个明确的坏事，但程序员很可能不希望这样做。

防止映射扩展的最简单方法是实现一个简单的nopage方法，总是导致向故障进程发送总线信号。这样的方法看起来像这样：

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

如我们所见，nopage方法只在进程解引用一个已知的VMA内的地址，但当前没有有效的页表条目时被调用。如果我们已经使用remap_pfn_range映射了整个设备区域，那么这里显示的nopage方法只对该区域外的引用被调用。因此，它可以安全地返回NOPAGE_SIGBUS来表示错误。当然，更彻底的nopage实现可以检查故障地址是否在设备区域内，并在这种情况下执行重新映射。然而，再次强调，nopage不能与PCI内存区域一起工作，所以扩展PCI映射是不可能的。

15.2.6. 重新映射 RAM

remap_pfn_range的一个有趣的限制是，它只能访问保留的页面和物理内存顶部以上的物理地址。在Linux中，物理地址的一个页面在内存映射中被标记为“保留”，以表示它不可用于内存管理。例如，在PC上，640 KB到1 MB的范围被标记为保留，托管内核代码的页面也是如此。保留的页面被锁定在内存中，是唯一可以安全映射到用户空间的页面；这个限制是系统稳定性的基本要求。

因此，remap_pfn_range不会允许你重新映射常规地址，这包括你通过调用get_free_page获得的地址。相反，它映射在零页面。一切看起来都在工作，除了进程看到的是私有的、填充了零的页面，而不是它希望的重新映射的RAM。然而，这个函数做了大多数硬件驱动程序需要它做的所有事情，因为它可以重新映射高PCI缓冲区和ISA内存。

通过运行mapper，可以看到remap_pfn_range的限制，mapper是在O'Reilly的FTP站点上提供的文件中的misc-progs中的一个示例程序。mapper是一个简单的工具，可以用来快速测试mmap系统调用；它映射由命令行选项指定的文件的只读部分，并将映射的区域转储到标准输出。例如，以下会话显示/dev/mem没有映射位于地址64 KB的物理页面，相反，我们看到的是一个充满零的页面（这个例子中的主机计算机是一台PC，但在其他平台上的结果会是一样的）：

```
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
```

remap_pfn_range无法处理RAM的问题表明，像scull这样的基于内存的设备不能轻易实现mmap，因为它的设备内存是常规RAM，而不是I/O内存。幸运的是，对于需要将RAM映射到用户空间的任何驱动程序，都有一个相对容易的解决方案；它使用我们之前看到的nopage方法。

15.2.6.1. 使用 nopage 方法重新映射 RAM

将真实RAM映射到用户空间的方法是使用vm_ops→nopage一次处理一个页面错误。一个示例实现是scullp模块的一部分，该模块在第8章中介绍。

scullp是一个面向页面的字符设备。因为它是面向页面的，所以它可以在其内存上实现mmap。实现内存映射的代码使用了在“Linux中的内存管理”一节中介绍的一些概念。

在检查代码之前，让我们看看影响scullp中mmap实现的设计选择：

- scullp在设备被映射的时候不释放设备内存。这是一个政策问题，而不是一个要求，它与scull和类似设备的行为不同，这些设备在打开写入时被截断到长度为0。拒绝释放一个被映射的scullp设备允许一个进程覆盖另一个进程积极映射的区域，所以你可以测试并看到进程和设备内存如何交互。为了避免释放一个被映射的设备，驱动程序必须保持一个活动映射的计数；设备结构中的vmas字段用于此目的。
- 当scullp order参数（在模块加载时设置）为0时，只执行内存映射。该参数控制如何调用**get_free_pages（参见第8章中的“get_free_page和Friends”部分）。零阶限制（强制一次分配一个页面，而不是在更大的组中）是由**get_free_pages的内部决定的，这是scullp使用的分配函数。为了最大化分配性能，Linux内核为每个分配阶维护一个空闲页面列表，只有第一个页面在集群中的引用计数由get_free_pages增加并由free_pages减少。如果分配阶大于零，mmap方法对于scullp设备是禁用的，因为nopage处理的是单个页面而不是页面集群。scullp简单

地不知道如何正确管理高阶分配的页面的引用计数。（如果你需要复习scullp和内存分配阶值，请返回到第8章的“A scull Using Whole Pages: scullp”部分。）

零阶限制主要是为了保持代码的简单。通过玩弄页面的使用计数，可以正确地为多页面分配实现mmap，但这只会增加示例的复杂性，而不会引入任何有趣的信息。

根据刚刚概述的规则，打算映射RAM的代码需要实现open、close和nopage VMA方法；它还需要访问内存映射以调整页面使用计数。

这个scullp_mmap的实现非常简短，因为它依赖nopage函数来做所有有趣的工作：

```c
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;

}
```

if语句的目的是避免映射分配顺序不为0的设备。scullp的操作存储在vm_ops字段中，设备结构的指针被存储在vm_private_data字段中。最后，调用vm_ops→open来更新设备的活动映射计数。

open和close只是跟踪映射计数，定义如下：

```c
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--;
}
```

然后，大部分工作由nopage完成。在scullp的实现中，nopage的地址参数用于计算设备的偏移量；然后使用偏移量在scullp内存树中查找正确的页面。

```c
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使用get_free_pages获取的内存。该内存使用逻辑地址进行寻址，所以scullp_nopage要做的就是调用virt_to_page来获取一个struct page指针。

现在，scullp设备按预期工作，如你可以在mapper实用程序的这个样本输出中看到。在这里，我们将/dev的目录列表（这是很长的）发送到scullp设备，然后使用mapper实用程序通过mmap查看该列表的部分：

```
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
```

15.2.7. 重映射内核虚拟地址

虽然很少需要，但是看到驱动程序如何使用mmap将内核虚拟地址映射到用户空间还是很有趣的。记住，真正的内核虚拟地址是由vmalloc等函数返回的地址，也就是在内核页表中映射的虚拟地址。本节中的代码来自scullv，这是一个像scullp一样工作的模块，但是通过vmalloc分配其存储空间。

scullv的大部分实现与我们刚刚看到的scullp相似，除了不需要检查控制内存分配的order参数。这是因为vmalloc一次分配一个页面，因为单页分配比多页分配更有可能成功。因此，分配顺序问题不适用于vmalloced空间。

除此之外，scullp和scullv使用的nopage实现之间只有一个区别。记住，一旦scullp找到了感兴趣的页面，就会用virt_to_page获取相应的struct page指针。然而，这个函数不能用于内核虚拟地址。相反，你必须使用vmalloc_to_page。所以scullv版本的nopage的最后部分看起来像这样：

```c
  /*
   * 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;
```

基于这个讨论，你可能也想将ioremap返回的地址映射到用户空间。然而，这将是一个错误；来自ioremap的地址是特殊的，不能像普通的内核虚拟地址那样处理。相反，你应该使用remap_pfn_range将I/O内存区域映射到用户空间。

15.3. 进行直接 I/O Performing Direct I/O

大多数I/O操作都通过内核进行缓冲。使用内核空间缓冲区可以在用户空间和实际设备之间提供一定程度的分离；这种分离可以使编程更容易，也可以在许多情况下提供性能优势。然而，有些情况下，直接对用户空间缓冲区进行I/O可能是有益的。如果传输的数据量大，直接传输数据而不通过内核空间进行额外的复制可以加快速度。

2.6内核中直接I/O使用的一个例子是SCSI磁带驱动程序。流式磁带可以通过系统传输大量数据，而且磁带传输通常是记录导向的，所以在内核中缓冲数据的好处很小。所以，当条件适合时（例如，用户空间缓冲区是页对齐的），SCSI磁带驱动程序在不复制数据的情况下执行其I/O。

话虽如此，重要的是要认识到，直接I/O并不总是能提供人们期望的性能提升。设置直接I/O的开销（涉及到故障转入和固定相关的用户页面）可能很大，而且丧失了缓冲I/O的好处。例如，使用直接I/O需要write系统调用同步操作；否则，应用程序不知道何时可以重用其I/O缓冲区。在每次写入完成之前停止应用程序可能会减慢速度，这就是为什么使用直接I/O的应用程序通常也使用异步I/O操作。

无论如何，真正的故事寓言是，在字符驱动程序中实现直接I/O通常是不必要的，甚至可能是有害的。只有当你确定缓冲I/O的开销真的在减慢速度时，你才应该采取这一步。还要注意，块和网络驱动程序不需要担心实现直接I/O；在这两种情况下，内核中的高级代码在需要时设置并使用直接I/O，驱动级代码甚至不需要知道正在执行直接I/O。

在2.6内核中实现直接I/O的关键是一个名为get_user_pages的函数，它在<linux/mm.h>中声明，具有以下原型：

```c
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);
```

这个函数有几个参数：

- tsk 指向执行I/O的任务的指针；其主要目的是告诉内核在设置缓冲区时应由谁负责任何页面错误。这个参数几乎总是作为current传递。
- mm 指向描述要映射的地址空间的内存管理结构的指针。mm_struct结构是将进程的虚拟地址空间的所有部分（VMAs）绑定在一起的部分。对于驱动程序使用，这个参数应该总是current→mm。
- start len start是用户空间缓冲区的（页对齐的）地址，len是缓冲区的长度（以页为单位）。
- write force 如果write非零，那么页面将被映射为写访问（当然，这意味着用户空间正在执行读操作）。force标志告诉get_user_pages覆盖给定页面上的保护以提供请求的访问；驱动程序应该总是在这里传递0。
- pages vmas 输出参数。成功完成后，pages包含指向描述用户空间缓冲区的struct page结构的指针列表，vmas包含指向相关VMAs的指针。这些参数应该，

显然，指向能够至少容纳len个指针的数组。任何一个参数都可以是NULL，但是你至少需要struct page指针来实际操作缓冲区。

`get_user_pages`是一个低级内存管理函数，具有适当复杂的接口。在调用之前，它还需要获取地址空间的mmap读/写信号量的读模式。因此，对get_user_pages的调用通常看起来像这样：

```c
down_read(&current->mm->mmap_sem);

result = get_user_pages(current, current->mm, ...);

up_read(&current->mm->mmap_sem);
```

返回值是实际映射的页面数，这可能少于请求的页面数（但大于零）。

成功完成后，调用者有一个指向用户空间缓冲区的pages数组，该数组被锁定在内存中。要直接操作缓冲区，内核空间代码必须使用kmap或kmap_atomic将每个struct page指针转换为内核虚拟地址。然而，通常，对于直接I/O有理由的设备正在使用DMA操作，所以你的驱动程序可能会想从struct page指针的数组创建一个散列/聚集列表。我们在“散列/聚集映射”一节中讨论如何做到这一点。

15.3.1. 异步 I/O

一旦你的直接I/O操作完成，你必须释放用户页面。然而，在这样做之前，你必须告诉内核你是否改变了这些页面的内容。否则，内核可能会认为页面是“干净的”，意味着它们与交换设备上找到的副本匹配，并在不将它们写出到后备存储器的情况下释放它们。所以，如果你已经改变了页面（响应用户空间的读请求），你必须用以下调用标记每个受影响的页面为脏：

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

（这个宏在<linux/page-flags.h>中定义）。执行此操作的大多数代码首先检查以确保页面不在内存映射的保留部分，该部分永远不会被交换出去。因此，代码通常看起来像这样：

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

由于用户空间内存通常不被标记为保留，这个检查严格来说不是必要的，但是当你在内存管理子系统深处动手脚时，最好是彻底和小心。

无论页面是否已经被改变，它们都必须从页面缓存中释放，否则它们会永远留在那里。要使用的调用是：

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

当然，这个调用应该在页面被标记为脏之后（如果需要的话）进行。

在2.6内核中添加的新功能之一是异步I/O能力。异步I/O允许用户空间启动操作，而不需要等待它们的完成；因此，一个应用程序可以在其I/O在执行过程中做其他处理。一个复杂的、高性能的应用程序也可以使用异步I/O同时进行多个操作。

异步I/O的实现是可选的，很少有驱动程序作者会去做；大多数设备并不从这个能力中受益。正如我们将在接下来的章节中看到的，块和网络驱动程序在所有时候都是完全异步的，所以只有字符驱动程序是显式异步I/O支持的候选对象。如果有充分的理由在任何给定的时间有多于一个的I/O操作未完成，那么字符设备可以从这个支持中受益。一个好的例子是流式磁带驱动器，如果I/O操作没有足够快地到达，驱动器可能会停滞并显著减慢。试图从流式驱动器中获得最佳性能的应用程序可以使用异步I/O在任何给定的时间都有多个操作准备好。

对于那些需要实现异步I/O的少数驱动程序作者，我们提供了一个如何工作的快速概述。我们在这一章中讨论异步I/O，因为它的实现几乎总是涉及到直接I/O操作（如果你在内核中缓冲数据，你通常可以实现异步行为，而不需要在用户空间上增加复杂性）。

支持异步I/O的驱动程序应该包含<linux/aio.h>。异步I/O的实现有三个file_operations方法：

```c
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);
```

aio_fsync操作只对文件系统代码有兴趣，所以我们在这里不再讨论它。其他两个，aio_read和aio_write，看起来非常像常规的read和write方法，但有几个例外。一个是offset参数是通过值传递的；异步操作永远不会改变文件位置，所以没有理由传递一个指向它的指针。这些方法也接受iocb（“I/O控制块”）参数，我们将在稍后讨论。

aio_read和aio_write方法的目的是启动一个读或写操作，这个操作在它们返回时可能已经完成，也可能还没有完成。如果可以立即完成操作，方法应该这样做，并返回通常的状态：传输的字节数或一个负的错误代码。因此，如果你的驱动程序有一个叫做my_read的读方法，以下的aio_read方法是完全正确的（尽管有点无意义）：

```c
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);
}
```

注意，struct file指针在kiocb结构的ki_filp字段中找到。

如果你支持异步I/O，你必须意识到内核有时会创建“同步IOCBs”。这些基本上是必须实际执行的异步操作。人们可能会想知道为什么要这样做，但最好还是按照内核的要求去做。同步操作在IOCB中标记；你的驱动程序应该用以下方法查询该状态：

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

如果这个函数返回一个非零值，你的驱动程序必须同步执行操作。

然而，最后，所有这些结构的目的是为了启用异步操作。如果你的驱动程序能够启动操作（或者，简单地，将其排队，直到未来某个时间可以执行），它必须做两件事：记住关于操作的所有需要知道的事情，并返回-EIOCBQUEUED给调用者。记住操作信息包括安排访问用户空间缓冲区；一旦你返回，你将再也没有机会在调用过程的上下文中访问那个缓冲区。一般来说，这意味着你可能需要设置一个直接的内核映射（使用get_user_pages）或一个DMA映射。-EIOCBQUEUED错误代码表示操作还没有完成，最终的状态将在后面发布。

当“后来”到来时，你的驱动程序必须通知内核操作已经完成。这是通过调用aio_complete完成的：

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

这里，iocb是最初传递给你的同一个IOCB，res是操作的通常结果状态。res2是将返回给用户空间的第二个结果代码；大多数异步I/O实现将res2作为0传递。一旦你调用aio_complete，你就不应该再触摸IOCB或用户缓冲区。

15.3.1.1. 一个异步 I/O 例子

示例源代码中的面向页面的scullp驱动程序实现了异步I/O。实现很简单，但足以显示异步操作应该如何结构化。

aio_read和aio_write方法实际上并不做太多事情：

```c
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);

}
```

这些方法只是简单地调用一个公共函数：

```c
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;

}
```

一个更完整的实现会使用get_user_pages将用户缓冲区映射到内核空间。我们选择通过在一开始就复制数据来简化生活。然后调用is_sync_kiocb来看看这个操作是否必须同步完成；如果是，返回结果状态，我们就完成了。否则我们在一个小结构中记住相关信息，通过工作队列安排“完成”，并返回-EIOCBQUEUED。此时，控制权返回到用户空间。

稍后，工作队列执行我们的完成函数：

```c
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);
}
```

这里，只是简单地用我们保存的信息调用aio_complete。当然，一个真正的驱动程序的异步I/O实现稍微复杂一些，但它遵循这种结构。

15.4. 直接内存访问

直接内存访问（DMA）是我们对内存问题概述的高级主题。DMA是一种硬件机制，允许外围组件直接将其I/O数据传输到主内存和从主内存中传输，无需涉及系统处理器。使用这种机制可以大大提高设备的吞吐量，因为可以消除大量的计算开销。

15.4.1. 一个 DMA 数据传输的概况

在介绍编程细节之前，让我们回顾一下DMA传输是如何进行的，只考虑输入传输以简化讨论。

数据传输可以通过两种方式触发：软件请求数据（通过如read之类的函数）或硬件异步地将数据推送到系统。在第一种情况下，涉及的步骤可以总结如下：

1. 当进程调用read时，驱动程序方法分配一个DMA缓冲区，并指示硬件将其数据传输到该缓冲区。进程被置于睡眠状态。
2. 硬件将数据写入DMA缓冲区，并在完成时引发中断。
3. 中断处理程序获取输入数据，确认中断，并唤醒进程，现在可以读取数据。

第二种情况是当DMA异步使用时。例如，数据采集设备即使没有人读取它们也会继续推送数据。在这种情况下，驱动程序应该维护一个缓冲区，以便后续的read调用将所有累积的数据返回到用户空间。这种传输涉及的步骤略有不同：

1. 硬件引发中断以宣布新数据已经到达。
2. 中断处理程序分配一个缓冲区，并告诉硬件在哪里传输其数据。
3. 外围设备将数据写入缓冲区，并在完成时引发另一个中断。
4. 处理程序分派新数据，唤醒任何相关进程，并进行清理。

异步方法的一个变体经常在网络卡中看到。这些卡通常期望在与处理器共享的内存中建立一个循环缓冲区（通常称为DMA环形缓冲区）；每个传入的数据包都放在环中的下一个可用缓冲区中，并发出一个中断信号。然后驱动程序将网络数据包传递给内核的其余部分，并在环中放置一个新的DMA缓冲区。

在所有这些情况中的处理步骤强调，高效的DMA处理依赖于中断报告。虽然可以用轮询驱动程序实现DMA，但这没有意义，因为轮询驱动程序会浪费DMA相对于更容易的处理器驱动I/O所提供的性能优势。

这里引入的另一个相关项目是DMA缓冲区。DMA要求设备驱动程序分配一个或多个适合DMA的特殊缓冲区。注意，许多驱动程序在初始化时分配他们的缓冲区，并在关闭时使用它们——因此，前面列表中的“分配”一词意味着“获取一个先前分配的缓冲区”。

15.4.2. 分配 DMA 缓冲

本节介绍了在低级别分配DMA缓冲区；我们稍后会介绍一个更高级的接口，但理解这里介绍的材料仍然是个好主意。

与DMA缓冲区相关的主要问题是，当它们大于一个页面时，它们必须在物理内存中占用连续的页面，因为设备使用ISA或PCI系统总线传输数据，这两种都携带物理地址。值得注意的是，这个约束不适用于SBus（参见第12章的“SBus”部分），它在外围总线上使用虚拟地址。一些架构也可以在PCI总线上使用虚拟地址，但是一个可移植的驱动程序不能依赖于这种能力。

虽然DMA缓冲区可以在系统启动时或运行时分配，但模块只能在运行时分配它们的缓冲区。（第8章介绍了这些技术；“获取大缓冲区”部分涵盖了在系统启动时的分配，而“kmalloc的真实故事”和“get_free_page及其朋友们”描述了在运行时的分配。）驱动程序编写者在为DMA操作使用内存时必须小心分配正确类型的内存；并非所有的内存区域都适合。特别是，高内存在某些系统和某些设备上可能无法用于DMA——外围设备简单地无法处理那么高的地址。

大多数现代总线上的设备可以处理32位地址，这意味着正常的内存分配对它们来说工作得很好。然而，一些PCI设备未能实现完整的PCI标准，不能处理32位地址。当然，ISA设备只限于24位地址。

对于具有这种限制的设备，应通过向kmalloc或get_free_pages调用添加GFP_DMA标志从DMA区域分配内存。当这个标志存在时，只分配可以用24位地址的内存。或者，你可以使用通用DMA层（我们稍后会讨论）来分配缓冢区，以解决你的设备的限制。

15.4.2.1. 自己做分配

我们已经看到get_free_pages可以分配多达几兆字节（因为order可以达到MAX_ORDER，目前为11），但即使请求的缓冲区远小于128KB，高阶请求也容易失败，因为系统内存会随着时间的推移变得碎片化。

当内核无法返回请求的内存量或者你需要超过128KB的内存时（例如，PCI帧抓取器常常需要这样），返回-ENOMEM的一种替代方法是在启动时分配内存或者为你的缓冲区保留物理RAM的顶部。我们在第8章的“获取大缓冲区”部分描述了在启动时分配内存，但它对模块不可用。通过在启动时向内核传递一个mem=参数来保留RAM的顶部。例如，如果你有256MB，参数mem=255M会阻止内核使用顶部的1MB。你的模块后来可以使用以下代码来访问这样的内存：

dmabuf = ioremap (0xFF00000 / *255M* /, 0×100000 / *1M* /);

分配器是伴随书籍的示例代码的一部分，提供了一个简单的API来探测和管理这样的保留RAM，并已经在几个架构上成功使用。然而，当你有一个高内存系统（即，物理内存超过CPU地址空间的系统）时，这个技巧就不起作用了。

当然，另一个选择是使用GFP_NOFAIL分配标志来分配你的缓冲区。然而，这种方法严重压力内存管理子系统，并有可能锁定整个系统；除非真的没有其他办法，否则最好避免使用。

然而，如果你要分配一个大的DMA缓冲区，那么考虑替代方案是值得的。如果你的设备可以进行散射/聚集I/O，你可以分配较小的缓冲区，让设备完成剩下的工作。当执行直接I/O到用户空间时，也可以使用散射/聚集I/O，这可能是当需要一个真正巨大的缓冲区时的最佳解决方案。

15.4.3. 总线地址

使用DMA的设备驱动程序需要与连接到接口总线的硬件进行通信，该总线使用物理地址，而程序代码使用虚拟地址。

实际上，情况比这稍微复杂一些。基于DMA的硬件使用总线地址，而不是物理地址。虽然ISA和PCI总线地址在PC上只是物理地址，但这并不适用于每个平台。有时，接口总

线通过桥接电路连接，该电路将I/O地址映射到不同的物理地址。一些系统甚至有一个页面映射方案，可以使任意页面在外围总线上看起来是连续的。

在最低级别（我们稍后会看到一个更高级别的解决方案），Linux内核通过导出以下在<asm/io.h>中定义的函数提供了一个可移植的解决方案。强烈不建议使用这些函数，因为它们只在具有非常简单的I/O架构的系统上工作正常；尽管如此，当你处理内核代码时，你可能会遇到它们。

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

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

这些函数在内核逻辑地址和总线地址之间进行简单的转换。在任何需要编程I/O内存管理单元或需要使用弹跳缓冲区的情况下，它们都不起作用。执行这种转换的正确方法是使用通用DMA层，所以我们现在转到那个主题。

15.4.4. 通用 DMA 层

最后，DMA操作归结为分配一个缓冲区并将总线地址传递给你的设备。然而，编写在所有架构上安全正确地执行DMA的可移植驱动程序的任务比人们想象的要困难。不同的系统对缓存一致性的工作方式有不同的理解；如果你没有正确处理这个问题，你的驱动程序可能会破坏内存。一些系统有复杂的总线硬件，可以使DMA任务更容易或更困难。并不是所有的系统都能从所有部分的内存执行DMA。幸运的是，内核提供了一个独立于总线和架构的DMA层，它隐藏了大部分这些问题，使驱动程序作者无需关心。我们强烈建议你在编写的任何驱动程序中使用这个层进行DMA操作。

下面的许多函数需要一个指向struct device的指针。这个结构是Linux设备模型中设备的低级表示。驱动程序通常不需要直接处理它，但在使用通用DMA层时你需要它。通常，你可以在描述你的设备的特定总线内部找到这个结构。例如，它可以在struct pci_device或struct usb_device的dev字段中找到。设备结构在第14章中详细介绍。

使用以下函数的驱动程序应该包含<linux/dma-mapping.h>

15.4.4.1. 处理困难硬件

在尝试DMA之前，必须首先回答的问题是，给定的设备是否能在当前主机上执行这样的操作。由于许多原因，许多设备在可以寻址的内存范围上有限制。默认情况下，内核假

设你的设备可以对任何32位地址执行DMA。如果不是这样，你应该通过调用以下函数通知内核这个事实：

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

mask应该显示你的设备可以寻址的位；例如，如果它限制为24位，你应该将mask传递为0x0FFFFFF。如果给定的mask可以进行DMA，返回值为非零；如果dma_set_mask返回0，你将无法使用这个设备进行DMA操作。因此，对于一个限制为24位DMA操作的设备的驱动程序的初始化代码可能看起来像这样：

```c
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");
}
```

再次强调，如果你的设备支持正常的32位DMA操作，就没有必要调用dma_set_mask。

15.4.4.2. DMA 映射

DMA映射是分配DMA缓冲区和为设备可访问的该缓冲区生成地址的组合。人们可能会想通过简单地调用virt_to_bus来获取该地址，但有强烈的理由避免这种方法。首先，合理的硬件配备了一个IOMMU，为总线提供了一组映射寄存器。IOMMU可以安排任何物理内存出现在设备可访问的地址范围内，并且它可以使物理上分散的缓冲区在设备看来是连续的。使用IOMMU需要使用通用DMA层；virt_to_bus无法完成这项任务。

注意，并非所有的架构都有IOMMU；特别是，流行的x86平台没有IOMMU支持。然而，正确编写的驱动程序不需要知道它正在运行的I/O支持硬件。

为设备设置一个有用的地址在某些情况下也可能需要建立一个弹跳缓冲区。当驱动程序试图在外围设备无法到达的地址上执行DMA时，例如高内存地址，就会创建弹跳缓冲区。然后根据需要将数据复制到弹跳缓冲区并从中复制出来。不用说，使用弹跳缓冲区可能会减慢速度，但有时没有其他选择。

DMA映射还必须解决缓存一致性问题。记住，现代处理器在快速的本地缓存中保留了最近访问的内存区域的副本；没有这个缓存，就无法获得合理的性能。如果你的设备改变了主内存的一个区域，那么必须使覆盖该区域的任何处理器缓存失效；否则，处理器可能会使用主内存的错误图像工作，导致数据损坏。同样，当你的设备使用DMA从主内存读取数据时，必须首先将驻留在处理器缓存中的对该内存的任何更改刷新出来。如果程序员不小心，这些缓存一致性问题可能会产生无尽的难以发现的错误。一些架构在硬件中管理缓存一致性，但其他架构需要软件支持。通用DMA层竭尽全力确保所有架构上的工作都是正确的，但是，正如我们将看到的，正确的行为需要遵守一小部分规则。

DMA映射设置了一个新的类型，dma_addr_t，用于表示总线地址。驱动程序应将dma_addr_t类型的变量视为不透明的；唯一允许的操作是将它们传递给DMA支持例程和设备本身。作为总线地址，dma_addr_t可能会导致CPU直接使用时出现意外问题。

PCI代码根据DMA缓冲区预期的存在时间，区分了两种类型的DMA映射：

- Coherent DMA映射 这些映射通常在驱动程序的生命周期内存在。一个coherent缓冲区必须同时对CPU和外围设备可用（我们稍后将看到，其他类型的映射在任何给定时间只能对其中一个或另一个可用）。因此，coherent映射必须存在于缓存一致的内存中。设置和使用coherent映射可能会很昂贵。
- Streaming DMA映射 Streaming映射通常为单个操作设置。当使用streaming映射时，一些架构允许进行显著的优化，但这些映射也受到在如何访问它们方面更严格的规则的约束。内核开发人员建议尽可能使用streaming映射而不是coherent映射。这个建议有两个原因。首先，对于支持映射寄存器的系统，每个DMA映射在总线上使用一个或多个映射寄存器。Coherent映射，它们的生命周期很长，即使在不使用的时候也可能长时间占用这些寄存器。另一个原因是，在某些硬件上，streaming映射可以以无法应用于coherent映射的方式进行优化。

这两种映射类型必须以不同的方式操作；现在是时候看看细节了。

Coherent DMA映射是通过以下两个函数创建和销毁的：

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

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

`dma_alloc_coherent`函数分配一个大小为size的缓冲区，并返回一个指向该缓冲区的指针。该缓冲区在CPU和设备之间是一致的，可以被设备用于DMA操作。`dma_handle`参数是一个指向`dma_addr_t`变量的指针，该函数将在其中存储缓冲区的总线地址。`flag`参数是分配标志，通常为`GFP_KERNEL`或`GFP_ATOMIC`。

`dma_free_coherent`函数释放一个由`dma_alloc_coherent`分配的缓冲区。`cpu_addr`和`dma_handle`参数是`dma_alloc_coherent`返回的值。

Streaming DMA映射是通过以下函数创建和销毁的：

```c
dma_addr_t dma_map_single(struct device *dev, void
*cpu_addr, size_t size, enum dma_data_direction
direction);

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

`dma_map_single`函数接受一个CPU地址`cpu_addr`，并返回一个设备可以用于DMA操作的总线地址。`size`参数指定了要映射的数据的大小。`direction`参数指定了数据的传输方向，可以是`DMA_TO_DEVICE`，`DMA_FROM_DEVICE`，`DMA_BIDIRECTIONAL`或`DMA_NONE`。

`dma_unmap_single`函数撤销了由`dma_map_single`创建的映射。`dma_addr`和`size`参数是`dma_map_single`返回的值。`direction`参数应与创建映射时使用的值相同。

注意，使用这些函数时必须遵守一些规则，以确保缓存一致性和正确的DMA操作。在调用 dma_map_single 之后和 dma_unmap_single 之前，驱动程序不应修改数据。此外，驱动程序必须确保在调用 dma_unmap_single 之前，DMA操作已经完成。

15.4.4.3. 建立一致 DMA 映射

驱动程序可以通过调用 dma_alloc_coherent 来设置一个coherent映射：

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

此函数同时处理缓冲区的分配和映射。前两个参数是设备结构和所需缓冲区的大小。函数在两个地方返回DMA映射的结果。函数的返回值是驱动程序可以使用的缓冲区的内核虚拟地址；同时，关联的总线地址在 dma_handle 中返回。此函数处理分配，以便将缓冲区放置在可以与DMA一起工作的位置；通常，内存只是通过 get_free_pages 分配的（但请注意，大小是以字节为单位，而不是顺序值）。 flag 参数是描述如何分配内存的常规 GFP_ 值；通常应该是 GFP_KERNEL （通常）或 GFP_ATOMIC （在原子上下文中运行时）。

当不再需要缓冲区时（通常在卸载模块时），应使用 dma_free_coherent 将其返回给系统：

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

请注意，像许多通用DMA函数一样，此函数要求提供所有的大小、CPU地址和总线地址参数。

15.4.4.4. DMA pools

DMA池是一个用于小型、一致的DMA映射的分配机制。从 dma_alloc_coherent 获取的映射可能有一个最小的页面大小。如果你的设备需要比这更小的DMA区域，你可能应该使用DMA池。在你可能会尝试对嵌入在更大结构中的小区域进行DMA的情况下，DMA池也很有用。一些非常难以理解的驱动程序错误已经被追踪到与结构字段相邻的小

DMA区域的缓存一致性问题。为了避免这个问题，你应该始终显式地为DMA操作分配区域，远离其他非DMA数据结构。DMA池函数在`<linux/dmapool.h>`中定义。

在使用之前，必须通过调用以下函数创建DMA池：

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

这里，`name`是池的名称，`dev`是你的设备结构，`size`是要从这个池分配的缓冲区的大小，`align`是从池分配的所需硬件对齐（以字节表示），`allocation`是，如果非零，分配不应超过的内存边界。例如，如果`allocation`传递为4096，那么从这个池分配的缓冲区不会跨越4KB的边界。

当你完成一个池的使用，可以用以下函数释放它：

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

在销毁池之前，你应该将所有分配返回到池中。

分配是通过`dma_pool_alloc`处理的：

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

对于这个调用，`mem_flags`是常规的`GFP_`分配标志集。如果一切顺利，一个内存区域（在创建池时指定的大小）被分配并返回。与`dma_alloc_coherent`一样，结果DMA缓冲区的地址作为内核虚拟地址返回，并存储在`handle`中作为总线地址。

不需要的缓冲区应该用以下函数返回到池中：

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

15.4.4.5. 建立流 DMA 映射

流式映射比一致映射的接口更复杂，原因有很多。这些映射期望与驱动程序已经分配的缓冲区一起工作，因此，必须处理它们没有选择的地址。在一些架构上，流式映射也可以有多个、不连续的页面和多部分的“散射/聚集”缓冲区。由于所有这些原因，流式映射有自己的一套映射函数。

在设置流式映射时，你必须告诉内核数据移动的方向。一些符号（类型为 `enum dma_data_direction`）已经为此目的定义：

- `DMA_TO_DEVICE`
- `DMA_FROM_DEVICE`

这两个符号应该是相当自解释的。如果数据被发送到设备（可能是作为对写系统调用的响应），应该使用 `DMA_TO_DEVICE`；相反，发送到CPU的数据用 `DMA_FROM_DEVICE` 标记。

- `DMA_BIDIRECTIONAL`

如果数据可以在任一方向移动，使用 `DMA_BIDIRECTIONAL`。

- `DMA_NONE`

这个符号只是作为一个调试工具提供的。尝试使用这个“方向”的缓冲区会导致内核崩溃。

可能会有人想一直选择 `DMA_BIDIRECTIONAL`，但驱动程序作者应该抵制这种诱惑。在一些架构上，这个选择有性能代价。

当你有一个单一的缓冲区要传输时，用 `dma_map_single` 映射它：

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

返回值是你可以传递给设备的总线地址，如果出错则返回NULL。

一旦传输完成，应该用`dma_unmap_single`删除映射：

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

这里，`size`和`direction`参数必须与映射缓冲区时使用的参数匹配。

对流式DMA映射适用一些重要的规则：

- 缓冲区只能用于与映射时给出的方向值匹配的传输。
- 一旦一个缓冲区被映射，它就属于设备，而不是处理器。在缓冲区被取消映射之前，驱动程序不应以任何方式触摸其内容。只有在调用`dma_unmap_single`之后，驱动程序才能安全地访问缓冲区的内容（我们稍后会看到一个例外）。
- 在DMA仍然活动的情况下，不应该取消缓冲区的映射，否则会保证系统的严重不稳定。

你可能会想知道为什么驱动程序在缓冲区被映射后就不能再操作它。实际上，这个规则有两个原因。首先，当一个缓冲区被映射为DMA时，内核必须确保该缓冲区中的所有数据实际上已经被写入内存。当发出`dma_map_single`时，一些数据可能在处理器的缓存中，必须显式地刷新。处理器在刷新后写入缓冲区的数据可能对设备不可见。

其次，考虑一下如果要映射的缓冲区在设备无法访问的内存区域会发生什么。一些架构在这种情况下简单地失败，但其他架构创建一个跳跃缓冲区。跳跃缓冲区只是一个设备可以访问的内存区域的单独区域。如果一个缓冲区以`DMA_TO_DEVICE`的方向映射，并且需要一个跳跃缓冲区，那么原始缓冲区的内容作为映射操作的一部分被复制。显然，复制后对原始缓冲区的更改不会被设备看到。同样，`DMA_FROM_DEVICE`跳跃缓冲区由`dma_unmap_single`复制回原始缓冲区；设备的数据在复制完成之前不会出现。

顺便说一下，跳跃缓冲区是获取正确方向的一个原因。`DMA_BIDIRECTIONAL`跳跃缓冲区在操作前后都被复制，这通常是CPU周期的不必要浪费。

偶尔，驱动程序需要在不取消映射的情况下访问流式DMA缓冲区的内容。提供了一个调用来实现这个功能：

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

在处理器访问流式DMA缓冲区之前，应该调用这个函数。一旦调用了这个函数，CPU就“拥有”了DMA缓冲区，并可以根据需要处理它。然而，在设备访问缓冲区之前，应该用以下方法将所有权转移回设备：

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

再次，处理器在这个调用之后不应访问DMA缓冲区。

15.4.4.6. 单页流映射

偶尔，你可能想要在一个你有`struct page`指针的缓冲区上设置一个映射；例如，使用`get_user_pages`映射的用户空间缓冲区。要使用`struct page`指针设置和撤销流式映射，可以使用以下方法：

```c
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);
```

`offset`和`size`参数可以用来映射页面的一部分。然而，建议除非你真的知道你在做什么，否则应该避免部分页面映射。映射页面的一部分如果分配只覆盖了缓存行的一部分，可能会导致缓存一致性问题；这反过来又可能导致内存损坏和极其难以调试的错误。

15.4.4.7. 散列/聚集映射Scatter/gather mappings

散列/聚集映射是一种特殊类型的流式DMA映射。假设你有几个缓冲区，所有这些缓冲区都需要被传输到设备或从设备传输。这种情况可以通过几种方式产生，包括从`readv`或`writev`系统调用，一个聚集的磁盘I/O请求，或者一个映射的内核I/O缓冲区中的页面列表。你可以简单地依次映射每个缓冲区，并执行所需的操作，但是一次映射整个列表有其优点。

许多设备可以接受一个散列列表的数组指针和长度，并在一个DMA操作中传输它们；例如，如果可以在多个部分中构建数据包，那么“零拷贝”网络就更容易。将散列列表作为一个整体映射的另一个原因是利用总线硬件中有映射寄存器的系统。在这样的系统上，物理上不连续的页面可以从设备的角度组装成一个连续的数组。这种技术只有当散列列表中的条目等于页面大小（除了第一个和最后一个）时才有效，但是当它有效时，它可以将多个操作转换为一个DMA，并相应地加快速度。

最后，如果必须使用跳跃缓冲区，那么将整个列表合并成一个单独的缓冲区是有意义的（因为它正在被复制）。

所以现在你已经相信在某些情况下映射散列列表是有价值的。映射一个散列列表的第一步是创建并填充一个描述要传输的缓冲区的`struct scatterlist`数组。这个结构是依赖于架构的，并在`<asm/scatterlist.h>`中描述。然而，它总是包含三个字段：

```c
struct page *page;
```

对应于要在散列/聚集操作中使用的缓冲区的`struct page`指针。

```c
unsigned int length;

unsigned int offset;
```

该缓冲区的长度和它在页面中的偏移量。

要映射一个散列/聚集DMA操作，你的驱动程序应该为每个要传输的缓冲区在一个`struct scatterlist`条目中设置`page`，`offset`和`length`字段。然后调用：

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

其中`nents`是传入的散列列表条目的数量。返回值是要传输的DMA缓冲区的数量；它可能小于`nents`。

对于输入散列列表中的每个缓冲区，`dma_map_sg`确定要给设备的正确总线地址。作为这个任务的一部分，它还合并了在内存中相邻的缓冲区。如果你的驱动程序运行的系统有一个I/O内存管理单元，`dma_map_sg`也会编程该单元的映射寄存器，可能的结果是，从你的设备的角度看，你能够传输一个单一的，连续的缓冲区。然而，你永远不会知道最终的传输会是什么样子，直到调用之后。

你的驱动程序应该传输`dma_map_sg`返回的每个缓冲区。每个缓冲区的总线地址和长度存储在`struct scatterlist`条目中，但它们在结构中的位置从一个架构到另一个架构是不同的。定义了两个宏，使得可以编写可移植的代码：

```c
dma_addr_t sg_dma_address(struct scatterlist *sg);
```

从这个散列列表条目返回总线（DMA）地址。

```c
unsigned int sg_dma_len(struct scatterlist *sg);
```

返回这个缓冲区的长度。

再次记住，要传输的缓冲区的地址和长度可能与传递给 `dma_map_sg` 的不同。

一旦传输完成，一个散列/聚集映射是通过调用 `dma_unmap_sg` 来解除映射的：

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

注意， `nents` 必须是你最初传递给 `dma_map_sg` 的条目数，而不是函数返回给你的DMA缓冲区的数量。

散列/聚集映射是流式DMA映射，同样的访问规则适用于它们和单一的种类。如果你必须访问一个映射的散列/聚集列表，你必须先同步它：

```c
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);
```

15.4.4.8. PCI 双地址周期映射

通常，DMA支持层使用32位总线地址，可能受到特定设备的DMA掩码的限制。然而，PCI总线也支持一个64位寻址模式，双地址周期（DAC）。通用DMA层不支持这种模式，原因有两个，首先，它是一个PCI特定的特性。其次，许多DAC的实现最好的情况下也是有bug的，而且，因为DAC比常规的32位DMA慢，所以可能有性能成本。即使如此，还是有一些应用场景，使用DAC是正确的选择；如果你有一个设备，可能会与放置在高内存中的非常大的缓冲区一起工作，你可能会考虑实现DAC支持。这种支持只适用于PCI总线，所以必须使用PCI特定的例程。

要使用DAC，你的驱动程序必须包含`<linux/pci.h>`。你必须设置一个单独的DMA掩码：

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

只有当这个调用返回0时，你才能使用DAC寻址。

一个特殊的类型（`dma64_addr_t`）用于DAC映射。要建立这样的映射，调用`pci_dac_page_to_dma`：

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

你会注意到，DAC映射只能从`struct page`指针（毕竟，它们应该存在于高内存中，否则使用它们就没有意义）创建；它们必须一次创建一个页面。`direction`参数是通用DMA层中使用的`enum dma_data_direction`的PCI等价物；它应该是`PCI_DMA_TODEVICE`，`PCI_DMA_FROMDEVICE`，或`PCI_DMA_BIDIRECTIONAL`。

DAC映射不需要外部资源，所以在使用后无需显式释放它们。然而，必须像处理其他流式映射一样处理DAC映射，并遵守关于缓冲区所有权的规则。有一组函数用于同步DMA缓冲区，与通用的类似：

```c
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);
```

15.4.4.9. 一个简单的 PCI DMA 例子

这是一个如何使用DMA映射的简单示例，它展示了一个PCI设备的DMA编码。PCI总线上的DMA操作的实际形式非常依赖于被驱动的设备。因此，这个示例并不适用于任何真实的设备；相反，它是一个假设的驱动程序dad（DMA Acquisition Device）的一部分。这个设备的驱动程序可能会定义一个像这样的传输函数：

```c
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;

}
```

这个函数映射了要传输的缓冲区并启动了设备操作。工作的另一半必须在中断服务例程中完成，它看起来像这样：

```c
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. */

    ...

}
```

显然，这个示例省略了很多细节，包括可能需要防止尝试启动多个，同时的DMA操作的步骤.

15.4.5. ISA 设备的 DMA

ISA总线允许两种类型的DMA传输：原生DMA和ISA总线主DMA。原生DMA使用主板上的标准DMA控制器电路驱动ISA总线上的信号线。另一方面，ISA总线主DMA完全由外围设备处理。后一种类型的DMA很少使用，并且在这里不需要讨论，因为从驱动程序的角度看，它与PCI设备的DMA类似。ISA总线主的一个例子是1542 SCSI控制器，其驱动程序是内核源码中的drivers/scsi/aha1542.c。

就原生DMA而言，ISA总线上的DMA数据传输涉及三个实体：

8237 DMA控制器（DMAC） 控制器保存有关DMA传输的信息，如方向、内存地址和传输大小。它还包含一个计数器，用于跟踪正在进行的传输的状态。当控制器接收到DMA请求信号时，它获得总线的控制权，并驱动信号线，使设备可以读取或写入其数据。

外围设备 设备必须在准备传输数据时激活DMA请求信号。实际的传输由DMAC管理；硬件设备在控制器刺激设备时，顺序地从总线上读取或写入数据。当传输结束时，设备通常会引发中断。

设备驱动程序 驱动程序的工作很少；它为DMA控制器提供方向、总线地址和传输大小。它还与其外围设备进行通信，为其准备传输数据，并在DMA结束时响应中断。

原始的在PC中使用的DMA控制器可以管理四个“通道”，每个通道都与一组DMA寄存器相关联。四个设备可以同时在控制器中存储它们的DMA信息。新的PC包含两个DMAC设备的等价物：第二个控制器（主控制器）连接到系统处理器，第一个（从控制器）连接到第二个控制器的通道0。

通道编号从0-7：通道4不可用于ISA外围设备，因为它在内部用于将从控制器级联到主控制器。因此，可用的通道是从控制器（8位通道）的0-3和主控制器（16位通道）的5-7。任何DMA传输的大小，存储在控制器中，是一个表示总线周期数的16位数字。因此，从控制器的最大传输大小是64 KB（因为它在一个周期中传输八位）和主控制器的128 KB（它进行16位传输）。

因为DMA控制器是系统范围内的资源，内核帮助处理它。它使用DMA注册表提供DMA通道的请求和释放机制，以及一组函数来配置DMA控制器中的通道信息。

15.4.5.1. 注册 DMA 使用

你应该习惯于内核注册表——我们已经看到了它们用于I/O端口和中断线。DMA通道注册表与其他的类似。在包含了<asm/dma.h>之后，可以使用以下函数来获取和释放DMA通道的所有权：

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

void free_dma(unsigned int channel);
```

channel参数是一个0到7之间的数字，或者更准确地说，是一个小于MAX_DMA_CHANNELS的正数。在PC上，MAX_DMA_CHANNELS被定义为8以匹配硬件。name参数是一个标识设备的字符串。指定的名称出现在文件/proc/dma中，可以被用户程序读取。

request_dma的返回值为0表示成功，如果出现错误，则为-EINVAL或-EBUSY。前者表示请求的通道超出范围，后者表示另一个设备正在持有通道。

我们建议你对DMA通道采取与I/O端口和中断线相同的关怀；在打开时请求通道比在模块初始化函数中请求它要好得多。延迟请求允许驱动程序之间进行一些共享；例如，只要它们不同时使用，你的声卡和你的模拟I/O接口就可以共享DMA通道。

我们还建议你在请求中断线之后请求DMA通道，并在中断之前释放它。这是请求这两种资源的传统顺序；遵循这个惯例可以避免可能的死锁。注意，每个使用DMA的设备也需要一个IRQ线；否则，它无法信号数据传输的完成。

在一个典型的情况下，打开的代码看起来像下面这样，它引用了我们假设的dad模块。如所示，dad设备使用一个快速中断处理程序，不支持共享IRQ线。

```c
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;

}
```

与刚刚显示的打开匹配的关闭实现看起来像这样：

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

这是一个安装了声卡的系统上的/proc/dma文件的样子：

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

值得注意的是，默认的声音驱动程序在系统启动时获取DMA通道，并且从不释放它。cascade条目是一个占位符，表示通道4不可用于驱动程序，如前所述。

15.4.5.2. 和 DMA 控制器通讯

在注册之后，驱动程序的主要工作包括配置DMA控制器以进行正确的操作。这项任务并不简单，但幸运的是，内核导出了典型驱动程序所需的所有函数。

驱动程序需要在调用读或写时，或者在准备异步传输时配置DMA控制器。这项任务要么在打开时执行，要么在响应ioctl命令时执行，具体取决于驱动程序和它实现的策略。这里显示的代码通常是由读或写设备方法调用的代码。

这个小节提供了一个关于DMA控制器内部的快速概述，以便你理解这里介绍的代码。如果你想了解更多，我们建议你阅读 `<asm/dma.h>` 和一些描述PC架构的硬件手册。特别是，我们没有处理8位与16位数据传输的问题。如果你正在为ISA设备板写设备驱动程序，你应该在设备的硬件手册中找到相关信息。

DMA控制器是一个共享资源，如果有多个处理器同时试图编程它，可能会产生混淆。因此，控制器受到一个名为 `dma_spin_lock` 的自旋锁的保护。驱动程序不应直接操作锁；然而，已经提供了两个函数来为你做这个：

```c
unsigned long claim_dma_lock();
```

获取DMA自旋锁。这个函数也阻塞本地处理器上的中断；因此，返回值是一组描述之前中断状态的标志；当你完成锁的操作时，必须将它传递给下面的函数，以恢复中断状态。

```c
void release_dma_lock(unsigned long flags);
```

返回DMA自旋锁并恢复之前的中断状态。

在使用下面描述的函数时，应该持有自旋锁。然而，在实际的I/O过程中，不应该持有它。驱动程序在持有自旋锁时，永远不应该睡眠。

需要加载到控制器中的信息包括三项：RAM地址，必须传输的原子项的数量（以字节或字为单位），以及传输的方向。为此， `<asm/dma.h>` 导出了以下函数：

```c
void set_dma_mode(unsigned int channel, char mode);
```

指示通道必须从设备读取（DMA_MODE_READ）还是写入设备（DMA_MODE_WRITE）。还存在第三种模式，DMA_MODE_CASCADE，用于释放总线控制权。级联是第一个控制器连接到第二个控制器顶部的方式，但也可以被真正的ISA总线主设备使用。我们在这里不讨论总线主控。

```c
void set_dma_addr(unsigned int channel, unsigned int
addr);
```

分配DMA缓冲区的地址。函数将addr的24个最低有效位存储在控制器中。addr参数必须是总线地址（参见本章前面的“总线地址”部分）。

```c
void set_dma_count(unsigned int channel, unsigned int
count);
```

分配要传输的字节数。count参数也代表16位通道的字节；在这种情况下，数字必须是偶数。

除了这些函数，还有一些必须在处理DMA设备时使用的管理设施：

```c
void disable_dma(unsigned int channel);
```

可以在控制器内禁用DMA通道。在配置控制器之前，应禁用通道，以防止不正确的操作。（否则，由于控制器通过8位数据传输进行编程，因此，前面的函数都不是原子执行的，可能会发生数据损坏）。

```c
void enable_dma(unsigned int channel);
```

此函数告诉控制器，DMA通道包含有效数据。

```c
int get_dma_residue(unsigned int channel);
```

驱动程序有时需要知道DMA传输是否已完成。此函数返回仍需传输的字节数。成功传输后，返回值为0，而控制器工作时，返回值是不可预测的（但不为0）。不可预测性源于需要通过两个8位输入操作获取16位剩余量。

```c
void clear_dma_ff(unsigned int channel)
```

此函数清除DMA翻转锁存器。翻转锁存器用于控制对16位寄存器的访问。寄存器通过两个连续的8位操作进行访问，翻转锁存器用于选择最低有效字节（当它清除时）或最高有效字节（当它设置时）。当传输了八位时，翻转锁存器会自动切换；程序员必须在访问DMA寄存器之前清除翻转锁存器（将其设置为已知状态）。

使用这些函数，驱动程序可以实现如下函数来准备DMA传输：

```c
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;

}
```

然后，像下面这样的函数用于检查DMA的成功完成：

```c
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);

}
```

剩下要做的只是配置设备板。这个设备特定的任务通常包括读取或写入几个I/O端口。设备在重要的方面有所不同。例如，一些设备期望程序员告诉硬件DMA缓冲区有多大，有时驱动程序必须读取硬编码到设备中的值。对于配置板，硬件手册是你唯一的朋友。

15.5. 快速参考

本章介绍了下列关于内存处理的符号:

15.5.1. 介绍性材料

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

和内存管理相关的大部分函数和结构, 原型和定义在这些头文件.

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

在内核逻辑地址和物理地址之间转换的宏定义.

```
PAGE_SIZE
PAGE_SHIFT
```

常量, 给出底层硬件的页的大小(字节)和一个页面号必须被移位来转变为一个物理地址的位数.

struct page
在系统内存映射中表示一个硬件页的结构.

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

宏定义, 在内核逻辑地址和它们相关的内存映射入口之间转换的. page_address 只用在低地址页或者已被明确映射的高地址页. pfn_to_page 转换一个页面号到它的相关的 struct page 指针.

```
unsigned long kmap(struct page *page);
void kunmap(struct page *page);
```

kmap 返回一个内核虚拟地址, 被映射到给定页, 如果需要并创建映射. kunmap 为给定页删除映射.

```
#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 的高性能版本; 结果的映射只能被原子代码持有. 对于驱动, type 应当是 KM_USER1, KM_USER1, KM_IRQ0, 或者 KM_IRQ1.

struct vm_area_struct;
描述一个 VMA 的结构.

15.5.2. 实现 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);
```

位于 mmap 核心的函数. 它们映射 size 字节的物理地址, 从 pfn 指出的页号开始到虚拟地址 virt_add. 和虚拟空间相关联的保护位在 prot 里指定. io_remap_page_range 应当在目标地址在 I/O 内存空间里时被使用.

```
struct page *vmalloc_to_page(void *vmaddr);
```

转换一个由 vmalloc 获得的内核虚拟地址到它的对应的 struct page 指针.

15.5.3. 实现直接 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);
```

函数, 加锁一个用户空间缓冲到内存并且返回对应的 struct page 指针. 调用者必须持有 mm→mmap_sem.

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

宏定义, 标识给定的页为"脏"(被修改)并且需要写到它的后备存储, 在它被释放前.

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

释放给定的页从页缓存中.

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

宏定义, 返回非零如果给定的 IOCB 需要同步执行.

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

函数, 指示一个异步 I/O 操作完成.

15.5.4. 直接内存存取

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

过时的不好的函数, 在内核, 虚拟, 和总线地址之间转换. 总线地址必须用来和外设通讯.

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

需要来定义通用 DMA 函数的头文件.

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

对于无法寻址整个 32-位范围的外设, 这个函数通知内核可寻址的地址范围并且如果可进行 DMA 返回非零.

```
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);
```

分配和释放一致 DMA 映射, 对一个将持续在驱动的生命周期中的缓冲.

```
#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);
```

创建, 销毁, 和使用 DMA 池来管理小 DMA 区的函数.

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

符号, 用来告知流映射函数在什么方向数据移入或出缓冲.

```
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);
```

创建和销毁一个单使用, 流 DMA 映射.

```
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);
```

同步一个由一个流映射的缓冲. 必须使用这些函数, 如果处理器必须存取一个缓冲当使用流映射时.(即, 当设备拥有缓冲时).

```
#include <asm/scatterlist.h>
struct scatterlist { /* ... */ };
dma_addr_t sg_dma_address(struct scatterlist *sg);
unsigned int sg_dma_len(struct scatterlist *sg);
```

这个散布表结构描述一个涉及不止一个缓冲的 I/O 操作. 宏 sg_dma_address he sg_dma_len 可用来抽取总线地址和缓冲长度来传递给设备, 当实现发散/汇聚操作时.

```
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 映射一个 发散/汇聚 操作, 并且 dma_unmap_sg 恢复这些映射. 如果在这个映射被激活时缓冲必须被存取, dma_sync_sg** 可用来同步.

/proc/dma
包含在 DMA 控制器中的被分配的通道的文本快照的文件. 基于 PCI 的 DMA 不显示, 因为每个板独立工作, 不需要分配一个通道在 DMA 控制器中.

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

定义或者原型化所有和 DMA 相关的函数和宏定义. 它必须被包含来使用任何下面符号.

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

存取 DMA 注册. 注册必须在使用 ISA DMA 通道之前进行.

```
unsigned long claim_dma_lock( );
void release_dma_lock(unsigned long flags);
```

获取和释放 DMA 自旋锁, 它必须被持有, 在调用其他的在这个列表中描述的 ISA DMA 函数之前. 它们在本地处理器上也关闭和重新使能中断

```
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);
```

编程 DMA 信息在 DMA 控制器中. addr 是一个总线地址.

```
void disable_dma(unsigned int channel);
void enable_dma(unsigned int channel);
```

一个 DMA 通道必须被关闭在配置期间. 这些函数改变 DMA 通道的状态.

```
int get_dma_residue(unsigned int channel);
```

如果这驱动需要知道一个 DMA 传送在进行, 它可调用这个函数, 返回尚未完成的数据传输的数目. 在成功的 DMA 完成后, 这个函数返回 0; 值是不可预测的当数据仍然在传送时.

```
void clear_dma_ff(unsigned int channel);
```

DMA flip-flop 被控制器用来传送 16-位值, 通过 2 个 8 位操作. 它必须被清除, 在发送任何数据给处理器之前.