第十三章　MMAP 和 DMA

这一章介绍 Linux 内存管理和内存映射的奥秘。同时讲述设备驱动程序是如何使用“直接内存访问”（DMA）的。尽管你可能反对，认为 DMA 更属于硬件处理而不是软件接口，但我觉得与硬件控制比起来，它与内存管理更相关。
这一章比较高级；大多数驱动程序的作者并不需要太深入到系统内部。不过理解内存如何工作可以帮助你在设计驱动程序时有效地利用系统的能力。

Linux 中的内存管理

这一节不是描述操作系统中内存管理的理论，而是关注于这个理论在 Linux 实现中的主要特征。本节主要提供一些信息，跳过它不会影响您理解后面一些更面向实现的主题。

页表

当一个程序查一个虚地址时，处理器将地址分成一些位域（bit field）。每个位域被用来索引一个称做页表的数组，以获得要么下一个表的地址，要么是存有这个虚地址的物理页的地址。为了进行虚地址到物理地址的映射，Linux 核心管理三级页表。开始这也许会显得有些奇怪。正如大多数 PC 程序员所知道的，x86 硬件只实现了两级页表。事实上，大多数 Linux 支持的 32 位处理器实现两级，但不管怎样核心实现了三级。
在处理器无关的实现中使用三级，使得 Linux 可以同时支持两级和三级（如 Alpha）的处理器，而不必用大量的#ifdef 语句把代码搅得一团糟。这种“保守编码”方式并不会给核心在两级处理器上运行时带来额外的开销，因为实际上，编译器已经把没用的一级优化掉了。
但是让我们看一会儿实现换页的数据结构。为了跟上讨论，你应该记住大多数用作内存管理的数据都采用 unsigned long 的内部表示，因为它们所表示的地址不会再被复引用。
下述几条总结了 Linux 的三级实现，由图 13-1 示意：

- 一个“页目录（Page Directory,PGD）”是顶级页表。PGD 是由 pgd_t 项所组成的数组，每一项指向一个二级页表。每个进程都有它自己的页目录，你可以认为页目录是个页对齐的 pgd_t 数组。
- 二级表被称做“中级页目录（Page Mid_level Directory）”或 PMD。 PMD 是一个页对齐的 pmd_t 数组。每个 pmd_t 是个指向三级页表的指针。两级的处理器，如 x86 和 sparc_4c，没有物理 PMD；它们将 PMD 声明为只有一个元素的数组，这个元素的值就是 PMD 本身——马上我们将会看到 C 语言是如何处理这种情况以及编译器是如何把这一级优化掉的。
- 再下一级被简单地称为“页表（Page Table）”。同样地，它也是一个页对齐的数组，每一项被称为“页表项（Page Table Entry）”。核心使用 pte_t 类型表示每一项。pte_t 包含数据页的物理地址。

上面提到的类型都在<asm/page.h>中定义，每个与换页相关的源文件都必须包含它。
核心在一般程序执行时并不需要为页表查寻操心，因为这是有硬件完成的。不过，核心必须将事情组织好，硬件才能正常工作。它必须构造页表，并在处理器报告一个页面错时（即当处理器需要的虚地址不在内存中时）查找页表，。
下面的符号被用来访问页表。<asm/page.h>和<asm/pgtable.h>必须被包含以使它们可以被访问。
（Figure 13.1 Linux 的三级页表）
PTRS_PER_PGD
PTRS_PER_PMD

PTRS_PER_PTE

每个页表的大小。两级处理器置 PTRS_PER_PMD 为 1，以避免处理中级。

unsigned long pgd_bal(pgd_t pgd)

unsigned long pmd_val(pmd_t pmd)

unsigned long pte_val(pte_t pte)

这三个宏被用来从有类型数据项中获取无符号长整数值。这些宏通过在源码中使用严格的数据类型有助于减小计算开销。

pgd_t *pgd_offset(struct mm_struct *mm,unsigned long address)

pmd_t *pmd_offset(pgd_t *dir,unsigned long address)

pte_t *pte_offset(pmd_t *dir,unsigned long address)

这些线入函数是用于获取与address相关联的pgd，pmd和pte项。页表查询从一个指向结构mm_struct的指针开始。与当前进程内存映射相关联的指针是current->mm。指向核心空间的指针由init_mm描述，它没有被引出到模块，因为它们不需要它。两级处理器定义pmd_offset(dir,add)为(pmd_t*)dir，这样就把pmd折合在pgd上。扫描页表的函数总是被声明为inline，而且编译器优化掉所有pmd查找。

unsigned long pte_page(pte_t pte)

这个函数从页表项中抽取物理页的地址。使用 pte_val(pte)并不可行，因为微处理器使用 pte 的低位存贮页的额外信息。这些位不是实际地址的一部分，而且需要使用 *pte_page* 从页表中、抽取实际地址。

pte_present(pte_t pte)

这个宏返回布尔值表明数据页当前是否在内存中。这是访问 pte 低位的几个函数中最常用的一个——这些低位被 *pte_page* 丢弃。有趣的是注意到不论物理页是否在内存中，页表始终在（在当前的 Linux 实现中）。这简化了核心代码，因为 *pgd_offse*t 及其它类似函数从不失败；另一方面，即使一个有零“驻留存贮大小”的进程也在实际 RAM 中保留它的页表。

仅仅看看这些列出的函数不足以使你对 Linux 的内存管理算法熟悉起来；实际的内存管理要复杂的多，而且还要处理其它一些繁杂的事，如高速缓存一致性。不过，上面列出的函数足以给你一个关于页面管理实现的初步印象；你可以从核心源码的 i*nclude/asm* 和 *mm* 子树中得到更好的信息。

虚拟内存区域

尽管换页位于内存管理的最低层，你在能有效地使用计算机资源之前还需要一些别的知识。核心需要一种更高级的机制处理进程看到它的内存方式。这种机制在 Linux 中以“虚拟内存区域的方式实现，我称之为“区域”或“VMA”。

一个区域是在一个进程的虚存中的一个同质区间，一个具有同样许可标志的地址的连续范围。它与“段”的概念松散对应，尽管最好还是将其描述为“具有自己属性的内存对象”。一个进程的内存映象由下面组成：一个程序代码（正文）区域；一个数据、BSS（未初始化的数据）和栈区域；以及每个活动的内存映射的区域。一个进程的内存区域可以通过查看 */proc/pid/maps* 看到。*/proc/self* 是*/proc/pid* 的特殊情况，它总是指向当前进程，做为一个例子，下面是三个不同的内存映象，我在#字号后面加了一些短的注释：

（代码 271）

每一行的域为：

start_end perm offset major:minor inode

---

* 事实中，在sparc上的这些函数并不是inline的，而是实际extern的函数，它们没有被引出到模块化的代码中。因此，你不能在运行在上的模块中使用这些函数，不过实际上一般也用不着那样做。

perm 代表一个位掩码包括读、写和执行许可；它表示对属于这个区域的页，允许进程做什么。这个域的最后一个字符要么是 p 表示私有的，要么是 s 表示共享的。

/proc//maps* 的每个域对应着结构 vm_area_struct 的一个域，我们将在下面描述这个结构。

实现 *mmap* 的方法的驱动程序需要填充在映射设备的进程地址空间中的一个 VMA 结构。因此，驱动程序的作者对 VMA 应该有个最起码的理解以便使用它们。

让我们看一下结构 vm_area_struct（在<linux/mm.h>）中最重要的几个域。这些域可能在设备驱动程序的 *mmap* 实现中被用到。注意核心维护 VMA 的列表和树以优化区域查找，vm_area_struct 的几个域被用来维护这个组织。VMA 不能按照驱动程序的意愿被产生，不然结构将会崩溃。VMA 的几个主要域如下：

unsigned long vm_start

unsigned long vm_end

一个 VMA 描述的虚地址介于 vma->vm_start 和 vma->vm_end 之间。这两个域是 */pro/*/maps* 中显示的最先两个域。

struct inode *vm_inode

如果这个区域与一个 inode 相关联（如一个磁盘文件或一个设备节点），这个域是指向这个 inode 的指针。不然，它为 NULL。

unsigned long vm_offset

inode 中这个区域的偏移量。当一个文件或设备被映射时，这是映射到这个区域的第一个字节的文件的位置（filp->f_ops）。

struct vm_operations_struct *vm_ops

vma->vm_ops 说明这个内存区域是一个核心“对象”，就象我们在本书中一直在用的结构 file。这个区域声明在其内容上操作的“方法”，这个域就是用来列出这些方法。

和结构 vm_area_struct 一样，vm_operations_struct 在<linux/mm.h>中定义；它包括了列在下面的操作。这些操作是处理进程内存需要的所有操作，它们以被声明的顺序列出。列出的原型是 2.0 的，与 1.2.13 的区别在每一项中都有描述。在本章的后面，这些函数中的部分会被实现，那时会更完全地加以描述。

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

在核心生成一个 VMA 后，它就把它打开。当一个区域被复制时，孩子从父亲那里继承它的操作，就区域用 vm->open 打开。例如，当 *fork* 将存在进程的区域复制到新的进程时，vm_ops->open 被调用以打开所有的映象。另一方面，只要 *mmap* 执行，区域在 file->f_ops->mmap 被调用前被产生，此时不调用 vm_ops->open。

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

当一个区域被销毁时，核心调用它的 *close* 操作。注意 VMA 没有相关的使用计数；区域只被打开和关闭一次。

void(*unmap)(struct vm_area_struct *vma,unsigned long addr,size_t len);

核心调用这个方法取消一个区域的部分或全部映射。如果整个区域的映射被取消，核心在 vm_ops->unmap 返回后立即调用 vm_ops->close。

void (*protect)(struct vm_area_struct *vma,unsigned long,size_t,unsigned int new prot);

当前未被使用。许可（保护）位的处理并不依赖于区域本身。

int(*sync)(struct vm_area_struct *vma,unsigned long,size_t,unsigned int flags);

这个方法被 *msync* 系统调用以将一个脏的内存区段保存到存贮介质上。如果成功则返回值为 0 ，如果有错，则返回一个负数。核心版本 1.2 让这个方法返回 void，因为这个函数不被认为会失败。

void(*advise)(struct vm_area_struct *vma,unsigned long,size_t,unsigned int advise);

当前未被使用。

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

当一进程试图访问属于另一个有效 VMA 的某页，而该页当前不在内存时，*nopage* 方法就会被调用，如果它为相关区域定义。这个方法返回该页的（物理）地址。如果这个方法不为这个区域所定义，核心会分配一个空页。通常，驱动程序并不实现 *nopage*，因为被一个驱动程序映射的区段往往被完全映射到系统物理地址。核心版本 1.2 的 *nopage* 具有一个不同的原型和不同的含义。第三个参数 write_access 被当做“不共享”——一个非零值意味着该页必须被当前进程所有，而零则表示共享是可能的。

unsigned long(*wppage)(struct vm_area_struct *vma,unsigned long address,unsigned long page);

这个方法处理“写保护”页面错，但目前不被使用。核心处理所有不调用区域特定的回调函数却往一个被保护的页面上写的企图。写保护被用来实现“写时拷贝（copy_on_write）”。一个私有的页可以被不同进程所共享，直到其中一个进程试图写它时。当这种情况发生时，页面被克隆，进程向自己的页拷贝上写。如果整个区域被称为只读，会有一 SIGSEGV 信息被发送给进程，写时拷贝就未能完成。

int (*swapout)(struct vm_area_struct *vma,unsigned long offset,pte_t *page_table);

这个方法被用来从交换空间取得一页。参数 offset 是相对区域而言（与上面 *swapout* 一样），而 entry 是页面的当前 pte——如果 *swapout* 在这一项中保存了一些信息，那么现在就可以用这些信息来取得该页。

一般说来，驱动程序并不需要去实现 *swapout* 或 *swapin*，因为驱动程序通常映射 I/O 内存，而不是常规内存。I/O 页是一些象访问内存一样访问的物理地址，但被映射到设备硬件而不是 RAM 上。I/O 内存区段或者被标记为“保留”，或者居于物理内存之上，因此它们从不被换出—交换 I/O 内存没什么实际意义。

内存映象

在 Linux 中还有与内存管理相关的第三个数据结构。VMA 和页表组织虚拟地址空间，而物理地址空间则由内存映象概括。

核心需要物理内存当前使用情况的一个描述。由于内存可以被看作是页面数组，因此这个信息也可以组织为一个数组。如果你需要其页面的信息，你就用其物理地址去访问内存映象。下面就是核心代码用来访问内存映象的一些符号：

typedef struct {/*…*/} mem_map_t

extern mem_map_t mem_map[];

映象本身是 mem_map_t 的一个数组。系统中的每个物理页，包括核心代码和核心数据，都在 mem_map 中有一项。

PAGE_OFFSET

这个宏表示由物理地址映射到的核心地址空间中的虚地址。PAGE_OFFSET 在任何用到“物理”地址的地方都必须要考虑。核心认为的物理地址实际上是一个虚拟地址，从实际物理地址偏移 PAGE_OFFSET——这个实际物理地址是在 CPU 外的电气地址线使用。在 Linux2.0.x 中，PAGE_OFFSET 在 PC 上都是零，在大多数其它平台上都不是零。2.1.0 版修改了 PC 上的实现，所以它现在也使用偏移映射。如果考虑到核心代码，将物理空间映射到高的虚拟地址有一些好处，但这已经超出了本书的范围。

int MAP_NR(addr)

当程序需要访问一个内存映象时，MAP_NR 返回在与 addr 关联的 mem_map 数组中的索引。参数 addr 可以是 unsigned long，也可以是一个指针。因为这个宏被几个关键的内存管理函数使用多次，所以它不进行 addr 的有效性检查；调用代码在必要的时候必须自己进行检查。

((nr<<PAGE_SHIFT)+PAGE_OFFSET)

没有标准化的函数或者宏可以将一个映象号转译为一个物理地址。如果你需要MAP_NR 的逆函数，这个语句可以使用。

内存映象是用来为每个内存页维护一些低级信息。在核心开发过程中，内存映象结构的准确定义变过几次，你不必了解细节，因为驱动程序不期望查看映象内部。

不过，如果你对了解页面管理的内部感兴趣的话，头文件<linux/mm.h>含有一大段注释解释mem_map_t 域的含义。

mmap 设备操作

内存映象是现代 Unix 系统中最有趣的特征之一。至于驱动程序，内存映射可以提供用户程序对设备内存的直接访问。

例如，一个简单 ISA 抓图器将图象数据保存在它自己的内存中，或者在 640KB-1KB 地址范围，或者在“ISA 洞”（指 14MB-16MB 之间的范围参见第 8 章“硬件管理”中“访问设备板子上的内存”一节）中。

将图象数据复制到常规（并且更快）RAM 中是不定期抓图的合适的方法，但如果用户程序需要经常性地访问当前图象，使用 *mmap* 方法将更合适。

映射一个设备的意思是使用户空间的一段地址空间关联到设备内存上。当程序读写指定的地址范围时，它实际上是在访问设备。

正如你所怀疑的，并不是每个设备都适合 *mmap* 概念；例如，对于串口或其它面向流的设备来说它的确没有意义。*mmap* 的另一个限制是映射是以 PAGE_SIZE 为单位的。核心只能在页表一级处置虚地址，因此，被映射的区域必须是 PAGE_SIZE 的整数倍，而且居于页对齐的物理内存。核心通过使一个区段稍微大一点儿的办法解决了页面粒度问题。对齐的问题通过使用 vma->vm_offset 来处理，但这对于驱动程序并不可行——映射一个设备简化为访问物理页，它必须是页对齐的。

这些限制对驱动程序来说并不是很大的问题，因为不管怎样，访问设备的程序是设备相关的。它知道如何使得被映射的内存区段有意义，因此页对齐不是一个问题。当你 ISA 板子插到一个 Alpha 机器上时，有一个更大的限制，因为 ISA 内存是以 8 位、16 位或 32 位项的散布集合被访问的，没有从 ISA 地址到 Alpha 地址的直接映射。在这种情况下，你根本不能使用 *mmap*。不能进行 ISA 地址到 Alpha 地址的直接映射归因于两种系统数据传送规范的不兼容。Alpha 只能进行 32 位和 64 位的内存访问，而 ISA 只能进行 8 位和 16 位的传送，没有办法透明地从一个协议映射到另一个。结果是你根本不能对插在 Alpha 计算机的 ISA 板子使用 *mmap*。

当可行的时候，使用 *mmap* 有一些好处。例如，一个类似于 X 服务器的程序从显存中传送大量的数据；把图形显示映射到用户空间与 *lseek/write* 实现相比，显著地改善了吞吐率。另一个例子是程序控制 PCI 设备。大多数 PCI 外围设备都将它们的控制寄存映射到内存地址上，一个请求应用更喜欢能直接访问寄存器，而不是反复调用 *ioctl* 来完成任务。

mmap 方法是 file_oprations 结构的一部分，在 *mmap* 系统调用被发出时调用。在调用实际方法之前，核心用 mmap 完成了很多工作，因此，这个方法的原型与系统调用很不一样。这与其它调用如 *ioctl* 和 *select* 不同，它们在被调用之前核心并不做太多的工作。

系统调用如下声明（在 mmap（2）手册中有描述）：

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

另一方面，文件操作如下声明：

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

方法中 inode 和 filp 参数与第三章“字符设备驱动程序”中介绍的一样。vma 会有用以访问

设备的虚拟地址范围的信息。这样，驱动程序只需为这个地址范围构造合适的页表：如果需要，用一组新的操作代替 vma->vm_ops。

一个简单的实现

设备驱动程序的大多数 *mmap* 实现对居于周边设备上的某些 I/O 内存进行线性的映射。*/dev/mem* 和*/dev/audio* 都是这类重映射的例子。下面的代码来自 *drivers/char/mem.c*，显示了在一个被称为 simple（Simple Implementation Mapping Pages with Little Enthusiasm）的典型模块中这个任务是如何完成的：

（代码 277）

很清楚，操作的核心由 *remap_page_range* 完成，它被引出到模块化的驱动程序，因为它做了大多数映射需要做的工作。

维护使用计数

上面给出的实现的主要问题在于驱动程序没有维护一个与被映射区域的连接。这对*/dev/mem* 来说并不是个问题，它是核心的一个完整的部分，但对于模块来说必须有一个办法来保持它的使用计数是最新的。一个程序可以对文件描述符调用 *close*，并仍然访问内存映射的区段。然而，如果关闭文件描述符导致模块的使用计数降为零，那么模块可能被卸载，即使它们仍被通过 *mmap* 使用着。

试图关于这个问题警告模块的使用者是不充分的解决办法，因为可能使用 *kerneld* 装载和卸载你的模块。这个守护进程在模块的使用计数降为零时自动地去除它们，你当然不能警告 *kerneld* 去留神 *mmap*。

这个问题的解决办法是用跟踪使用计数的操作取代缺省的 vma->vm.ops。代码相当简单——用于模块化的*/dev/mem* 的一个完全的 *mmap* 实现如下所示：

（代码 278）

这个代码依赖于一个事实，即核心在调用 f_op->mmap 之前将新产生区域中的 vm_ops 域初始化为 NULL。为安全起见以防止在将来的核心发生什么改变，给出的代码检查了指针的当前值。

给出的实现利用了一个概念，即 *open（vma）*和 *close（vma）*都是缺省实现的一个补充。驱动程序的方法不须复制打开和关闭的内存区域的标准代码；驱动程序只是实现额外的管理。有趣的是注意到，VMA 的 *swapin* 和 *swapout* 方法以另外的方式工作——驱动程序定义的 vm_ops->swap*不是添加而是用完全不同的东西取代了缺省实现。

支持 mremap 系统调用

mremap 系统调用被应用程序用来改变映射区段的边界地址。如果驱动程序希望能支持 *mremap*，以前的实现就不能正确地工作，因为驱动程序没有办法知道映射的区域已经改变了。

Linux 的 *mremap* 实现不提醒驱动程序关于映射区域的改变。实际上，它到是通过 *unmap* 方法在区域减小时提醒驱动程序，但在区域变大时没有回调发出。

将减小告诉驱动程序隐含的基本思想法是驱动程序（或是将常规文件映射到内存的文件系统）需要知道区段什么时候被取消映射了，从而采取适应的动作，如将页面刷新到磁盘上。另一方面，映射区域的增大对驱动程序来说意义不大。除非调用 *mremap* 的程序访问新的虚地址。在实际情况中，映射从未使用的区段是很常见的（如未使用过的某些程序代码段）。因此，Linux 核心在映射区段增大时并不告诉驱动程序，因为 *nopage* 方法将会照管这些页。如果它们确实被访问了。

换句话说，当映射区段增大时，驱动程序未被提醒是因为 *nopage* 后来会这样做；从而不必在需要前使用内存。这个优化主要是针对常规文件的，它们使用真正的 RAM 进行映射。

因此，如果你想支持 *mremap* 系统调用，就必须实现 *nopage*。不过，一旦有了 *nopage*，你

可选择广泛地使用它，从而避免从 fops->mmap 调用 *remap_page_range*；这在下一个代码段中给出。在这个 *mmap* 的实现中，设备方法只取代了 vma->vm_fops。*nopage* 方法负责一次重映射一个页并返回其地址。

一个支持 *mremap*（为节省空间，不支持使用计数）的*/dev/mem* 实现如下所示：

（代码 279）

（代码 280）

如果 *nopage* 方法被留为 NULL，处理页面错的核心代码就将零页映射到出错虚地址。零页是一个写时拷贝页，被当作零来读，可以用来映射 BSS 段。因此，如果一个进程通过调用 *mremap* 扩展一个映射区段，并且驱动程序没有实现 *nopage*，你最终会得到一些零页，而不是段错。

注意，给出的实现远远不是最优的；如果内存方法能绕过 *remap_page_range* 而直接返回物理地址会更好。不幸的是，这个技术的正确实现牵涉到一些细节，只能在本章晚些时候搞清楚。而且上面给出的实现在核心 1.2 中并不能工作，因为 *nopage* 的原型在版本 1.2 和 2.0 之间做了修改。在本节中我不打算管 1.2 核心。

重映射特定的 I/O 区段

到目前为止，我们所看到的所有例子都是*/dev/mem* 的再次实现；它们将物理地址重映射到用户空间——或者至少这是它们认为它们所做的。然而，典型的驱动程序只想映射应用于它的外围设备的小地址区间，并非所有内存。

为了能为一个特定的驱动程序自定义*/dev/mem* 的实现，我们需要进一步来研究一下 *remap_page_range* 的内部。这个函数的完整原型是：

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

这个函数的返回值通常为零或为一个负的错误代码。让我们看看它的参数的确切含义。

unsigned long virt_add

重映射开始处的虚拟地址。这个函数为虚地址空间 virt_add 和 virt_add+size 之间的范围构造页表。

unsigned long phys_add

虚拟地址应该映射到的物理地址。这个地址在上面提到的意义下是“物理的”这个函数影响 phys_add 到 phys_add+size 之间的物理地址。

unsigned long size

被重映射的区域的大小，以字节为单位。

pgprot_t prot

为新页所请求的“保护”。驱动程序不必修改保护，而且在 vma->vma_page_prot 中找到的参数可以不加改变地使用。如果你好奇，你可以在<Linux/mm.h>中找到更多的信息。

为了向用户空间映射整个内存区间的一个子集，驱动程序需要处理偏移量。下面几行为映射了从物理地址 simple_region_start 开始的 simple_region_size 字节大小的区段的驱动程序完成了这项工作：

（代码 281）

除了计算偏移量，上面的代码还为错误条件引入了两个检查。第一个检查拒绝将一个在物理空间未对齐的位置映射到用户空间。由于只有完整的页能被重映射，因此映射的区段只能偏移页面大小的整数倍。ENXIO 是这种情况下通常返回的错误代码，它被展开为“无此设备或地址”。

第二个检查在程序试图映射多于目标设备 I/O 区段可获得内存的空间时报告一个错误。代码中 psize 是在偏移被确定后剩下的物理 I/O 大小，vsize 是请求的虚存大小；这个函数拒绝映

射超出允许内存范围的地址。
注意，如果进程调用 *mremap*，它便可以扩展其映射。一个“非常炫耀”的驱动程序可能希望阻止这个发生；达到目的的唯一办法是实现一个 vma->nopage 方法。下面是这个方法的最简单的实现：

```
unsigned long simple_pedantic_nopage(struct vm_area_struct *vma,unsigned long address, int write_access);
{return 0;}        /*发送一个 SIGBUS*/
```

如果 *nopage* 方法返回 0 而不是一个有效的物理地址，一个 SIGBUS（总线错）被发送到当前进程（即发生页面错的进程）。如果驱动程序没有实现 *nopage*，进程在请求的虚地址处得到一个零页；这通常可以接受，因为 *mremap* 是个非常少用的系统调用，而且将零页映射到用户空间也没有安全问题。

重映射 RAM

在 Linux 中，物理地址的一页被标记在内存映象中是“保留的”，表明不被内存管理系统使用。例如在 PC 上，640KB 到 1MB 之间的部分被称为“保留的”，它被用来存放核心代码。*remap_page_range*的一个有趣的限制是，它只能给予对保留的页和物理内存之上的物理地址的访问。保留页被锁在内存中，是仅有的能安全映射到用户空间的页；这个限制是系统稳定性的基本要求*。
因此，*remap_page_range* 不允许你重映射常规地址——包括你通过调用 *get_free_page* 所获得的那些。不过，这个函数做了所有一个硬件驱动程序希望它做的，因为它可以重映射高 PCI 缓冲和 ISA 内存——包括第 1 兆内存和 15MB 处 ISA 洞，如果在第八章“1M 以上的 ISA 内存”中提到的改变发生了的话。另一方面，当对非保留的页使用 *remap_page_range* 时，缺省的 *nopage* 处理程序映射被访问的虚地址处的零页。
这个行为可以通过运行 *mapper* 看到。*mapper* 是在 O’Reilly 的 FTP 站点上提供的文件中 *misc_programs* 里的一个示例程序。它是个可以快速测试 *mmap* 系统调用的简单工具。*mapper* 根据命令行选项映射一个文件中的只读部分，并把映射的区段输出到标准输出上。例如，下面这个交互过程表明*/dev/mem* 不映射位于 64KB 地址处的物理页（本例中的宿主机是个 PC，但在别的平台上结果应该是一样的）：
（代码 283）
remap_page_range 对处理 RAM 的无能为力说明象 *scull*p 这样的设备不能简单地实现 *mmap*，因为它的设备内存是常规 RAM，而不是 I/O 内存。
有两个办法可以绕过 *remap_page_range* 对 RAM 的不可用性。一个是“糟糕”的办法，另一个是干净的。

使用预定位

糟糕的办法要为你想映射到用户空间的页在 mem_map[MAP_NR(page)]中置 PG_reserved 位。这样就预定了这些页，而一旦预定了，*remap_page_range* 就可以按期望工作了。设置标志的代码很短很容易，但我不想在这儿给出来，因为另一个方法更有趣。不用说，不释放页面之前，预定的位必须被清除。
有两个原因说明为什么这是个好办法。第一，被标为预定的页永远不会被内存管理所动。核心在数据结构初始化之前系统引导时确定它们，因此不能用在任何其它用途上。而另一方面，通过 *get_free_page*,*vmalloc* 或其它一些方式分配的页都是由内存子系统处理的。即使 2.0 核心在你运行时预定额外的页并不崩溃，这样做可能在将来会产生问题。因而是不鼓励的。不过你可以尝试这种快且脏的技术看看它是任何工作的。

---

* 当某页成为进程内存映象的一部分时，它的使用计数必须增加，因为在取消映射时，它会被减小。 这类锁定不能在活动的RAM页上实施，因为它可能阻止正常的系统操作（象swapping和allocation/deallocation）。

预定页不是个好办法的第二个原因是被预定的页不被算做是整个系统内存的一部分，有的用户在系统 RAM 发生变化时可能会很在意——用户经常留意空闲内存数量，而总的内存量一般总和空闲内存一道显示。

实现 *nopage* 方法

将实际 RAM 映射到用户空间的一个较好的办法是用 vm_ops->nopage 来一次处理一个页面错。作为 *scullp* 模块一部分的一个示例实现在第七章“把握内存”中介绍过。

scullp 是面向页的字符设备。因为它是面向页的，所以可以在它的内存中实现 *mmap*。实现内存映射的代码用了一些以前在“Linux 的内存管理”中介绍过的概念。

在查看代码之前，让我们看一下影响 s*cullp* 中 *mmap* 实现的设计选择。

- 设备为模块更新使用计数
  在卸载模块时为了避免发生问题，内存区域的 *open* 和 *close* 方法被实现去跟踪模块的使用。
- 设备为页更新使用计数
  这是为保证系统稳定是一个严格要求；不能更新这个计数将导致系统崩溃。每个页有其自己的使用计数；当它降为零时，该页被插入到空闲页表。当一个活动映象被破坏掉时，核心会将相关 RAM 页的使用计数减小。因此，驱动程序必须增加它所映射的每个页的使用计数（注意，这个计数在 *nopage* 增加它时不能为零，因为该页已经被 fops->write 分配了）。
- 只要设备是被映射的，*scullp* 就不能释放设备内存
  这与其说是个要求，不如说是项政策，这与 s*cull* 及类似设备的行为不同，因为它们在被因写而打开时长度被截为 0。拒绝释放被映射的 *scullp* 设备允许一个设备一个进程重写正被另一个进程映射的区段，这样你就可以测试并看到进程与设备内存之间是如何交互的。为避免释放一个被映射的设备，驱动程序必须保存一个活动映射的计数；设备结构中的 vma 域被用于这个目的。
- 只有在 *scullp* 的序号 order 参数为 0 时才进行内存影射
  这个参数控制 *get_free_pages* 是如何调用的（见第七章中“get_free_pages 和朋友们”一节）。这个选择是由 *get_free_pages* 的内部机制决定的——*scullp* 利用的分配机制。为了最大化分配性能，Linux 核心为每个分配的序号（order）维护一个空闲页的列表，在一个簇中只有第一页的页计数由 *get-free_pages* 增加和 *free_pages* 减少。如果分配序号大于 0，那么对一个 *scullp* 设备来说 *mmap* 方法是关闭的，因为 *nopage* 只处理单项，而不是一簇页。

最后一个选择主要是为了保证代码的简单。通过处理页的使用计数，也有可能为多页分配正确地实现 *mmap*，但那样只能增加例子的复杂性，而不能带来任何有趣的信息。

如果代码想按照上面提到的规则来映射 RAM，它需要实现 *open*，*close* 和 *nopage*，还要访问 mem_map。

scullp_mmap 的实现非常短，因为它依赖于 *nopage* 来完成所有有趣的工作：

（代码 285　#1）

开头的条件语句是为了避免映射未对齐的偏移和分配序号不为 0 的设备。最后，vm_ops->open 被调用以更新模块的使用计数和设备的活动映射计数。

open 和 *close* 就是为了跟踪这些计数，被定义如下：

（代码 285　#2）

由于模块生成了 4 个 *scullp* 设备并且也没有内存区域可用的 private_data 指针，所以 *open* 和 *close* 取得与 vma 相关联的 *scullp* 设备是通过从 inode 结构中抽取次设备号。次设备号被用

来从设备结构的 scullp_devices 数组取偏移后得到指向正确结构的指针。
大部分工作是由 *nopage* 完成的。当进程发生页面错时，这个函数必须取得被引用页的物理地址并返回给调用者。如果需要，这个方法可以计算 address 参数的页对齐。在 *scullp* 的实现中，address 被用来计算设备里的偏移；偏移又被用来在 *scullp* 的内存树上查找正确的页。
（代码 286　#1）
最后一行增加页计数；这个计数在 atomic_t 中生命，因此可以由一个原子操作更新。事实上，在这种特定的情况下，原子更新并不是严格要求的，因为该页已经在使用中，并且没有与中断处理程序或别的异步代码的竞争条件。
现在 *scullp* 可以按预期的那样工作了，正如你在工具 *mapper* 的示例输出中所看到的：
（代码 286　#2）
（代码 287　#1）

重映射虚地址

尽管很少需要重映射虚地址，但看看驱动程序如何用 *mmap* 将虚地址映射到用户空间是很有趣的。这里虚地址指的是由 *vmalloc* 返回的地址，也就是被映射到核心页表的虚地址。本节的代码取自 *scullv*，这个模块与 *scullp* 类似，只是它通过 *vmalloc* 分配存储。
scullv 的大部分实现与我们刚刚看到的 *scullp* 完全类似，除了不需要检查分配序号。原因是 *vmalloc* 一次只分配一页，因为单页分配比多页分配容易成功的多。因此，使用计数问题在通过 *vmalloc* 分配的空间中不适用。
scullv 的主要工作是构造一个页表，从而可以象连续地址空间一样访问分配的页。而另一方面，*nopage* 必须向调用者返回一个物理地址。因此，*scullv* 的 *nopage* 实现必须扫描页表以取得与页相关联的物理地址。
这个函数与我们在 *scullp* 中看到的一样，除了结尾。这个代码的节选只包括了 *nopage* 中与 *scullp* 不同的部分。
（代码 287　2#）

```
    atomic_inc(&mem_map[MAP_NR(page)]).count;
    return page;
  }
```

页表由本章开始时介绍的那些函数来查询。用于这个目的的页目录存在核心空间的内存结构 init_m 中。
宏 VMALLOC_VMADDR(pageptr)返回正确的 unsigned long 值用于 *vmalloc* 地址的页表查询。注意，由于一个内存管理的问题，这个值的强制类型转换在早于 2.1 的 X86 核心上不能工作。在 X86 的 2.1.1 版中内存管理做了改动，VMALLOC_VMADDR 被定义为一个实体函数，与在其它平台上一样。
最后要提到的一点是 init_mm 是如何被访问的，因为我前面提到过，它并未引出到模块中。实际上，*scullv* 要作一些额外的工作来取得 init_mm 的指针，解释如下。
实际上，常规模块并不需要 init_mm，因为它们并不期望与内存管理交互；它们只是调用分配和释放函数。为 *scullv* 的 *mmap* 实现很少见。本小节中介绍的代码实际上并不用来驱动硬件；我介绍它只是用实际代码来支持关于页表的讨论。
不过，既然谈到这儿，我还是想给你看看 *scullv* 是如何获得 init_mm 的地址的。这段代码依赖于这样的事实：0 号进程（所谓的空闲任务）处于内核中，它的页目录描述了核心地址空间。为了触到空闲任务的数据结构，*scullv* 扫描进程链表直到找到 0 号进程。
（代码 288）
这个函数由 fops->mmap 调用，因为 *nopage* 只在 *mmap* 调用后运行。

基于上面的讨论，你也许还想将由 *vremap*（如果你用 Linux2.1，就是 *ioremap*）返回的地址映射到用户空间。这很容易实现，因为你可以直接使用 *remap_page_range*，而不用实现虚拟内存区域的方法。换句话说，*remap_page_range* 已经可用以构造将 I/O 内存映射到用户空间的页表；并不需要象我们在 *scullv* 中那样查看由 *vremap* 构造的核心页表。

直接内存访问

直接内存访问，或 DMA，是我们内存访问方面讨论的高级主题。DMA 是一种硬件机制，它允许外围组件将 I/O 数据直接从（或向）主存中传送。

为了利用硬件的 DMA 能力，设备驱动程序需要能正确地设置 DMA 传送并能与硬件同步。不幸的是，由于 DMA 的硬件实质，它非常以来于系统。每种体系结构都有它自己管理 DMA 传送的技术，编程接口也互不相同。核心也不能提供一个一致的接口，因为驱动程序很难将底层硬件机制适当地抽象。本章中，我将描述 DMA 在 ISA 设备及 PCI 外围上是如何工作的，因为它们是目前最常用的外围接口体系结构。

不过，我不想讨论 ISA 太多的细节。DMA 的 ISA 实现过于复杂，在现代外围中并不常用。目前 ISA 总线主要用在哑外围接口上，而需要 DMA 能力的硬件生产商倾向于使用 PCI 总线。

DMA 数据传送的概况

在介绍编程细节以前，我们先大致看看 DMA 传送是如何工作的。为简化讨论，只介绍输入传送。

数据传送有两种方式触发：或者由软件请求数据（通过一个函数如 *read*），或者由硬件将数据异步地推向系统。

在第一中情况下，各步骤可如下概括：

- 当一个进程调用一个 *read*，这个驱动程序方法分配一个 DMA 缓冲区，并告诉硬件去传诵数据。进程进入睡眠。
- 硬件向 DMA 缓冲区写数据，完成时发出一个中断。
- 中断处理程序获得输入数据，应答中断，唤醒进程，它现在可以读取数据。

有时 DMA 被异步地使用。例如，一些数据采集设备持续地推入数据，即使没有人读它。这种情况下，驱动程序要维护一个缓冲区，使得接下来的一个 *read* 调用可以将所有累积的数据返回到拥护空间。这种传送的步骤稍有不同：

- 硬件发出一个中断，表明新的数据到达了。
- 中断处理程序分配一个缓冲区，告诉硬件将数据传往何处。
- 外围设备将数据写入缓冲区；当写完时，再次发出中断。
- 处理程序派发新数据，唤醒所有相关进程，处理一些杂务。

上面这两种情况下的处理步骤都强调：高效的 DMA 处理以来于中断报告。尽管可以用一个轮询驱动程序来实现 DMA，这样做并无意义，因为轮询驱动程序会将 DMA 相对于简单的处理器驱动 I/O 获得的性能优势都抵消了。

这里介绍的另一个相关问题是 DMA 缓冲区。为利用直接内存访问，设备驱动程序必须能分配一个特殊缓冲区以适合 DMA。注意大多数驱动程序在初始化时分配它们的缓冲区，一直使用到关机——因此，上面步骤中“分配”一词指的是“获得以前已分配的缓冲区”。

分配 DMA 缓冲区

DMA 缓冲区的主要问题是当它大于一页时，它必须占据物理内存的连续页，因为设备使用 ISA 或 PCI 总线传送数据，它都只携带物理地址。有趣的是注意到，这个限制对 Sbus 并不适用（见第 15 章“外围总线概览”中“Sbus”一节），它在外围总线上适用虚地址。

尽管 DMA 缓冲区可以在系统引导或运行时分配，模块只能在运行时分配其缓冲区。第七章

介绍了这些技术：“Playing Dirty”讲述在系统引导时分配；“*kmalloc* 的真实故事”和“*get_free_page* 和朋友们”讲述运行时分配。如果你用 *kmalloc* 或 *get_free_page*，你必须指定 GFP_DMA 优先级，与 GFP_KERNEL 或 GFP_ATOMIC 进行异或。

GFP_DMA 要求内存空间必须适合于 DMA 传送。核心保证能够进行 DMA 的缓冲区具有以下两个特点。第一，当 *get_free_page* 返回不止一页时，其物理地址必须是连续的（不过，一般情况下的确如此，与 GFP_DMA 无关，因为本身就是以成簇的连续页来组织空闲内存的）。第二，当 GFP_DMA 等设备时，核心保证只有低于 MAX_DMA_ADDRESS 的地址才被返回。宏 MAX_DMA_ADDRESS 在 PC 上被设为 16MB，用以对付马上就会讲到的 ISA 限制。

在 PCI 情况下，没有 MAX_DMA_ADDRESS 的限制，PCI 设备驱动程序在分配它的缓冲区时应避免设置 GFP_DMA。

自行分配

我们已经明白为何*get_free_page*(从而*kmalloc*)不能返回超过 128KB（或更一般地，32 页）的连续内存空间。但这个要求很容易失败，即使当分配的缓冲区小于 128KB时，因为随着时间的推移，系统内存会成为一些碎片*。

如果核心不能返回所需数量的内存，或如果你需要超过 128KB 的内存（例如对于一个 PCI 抓图器来说，这是个很常见的需求），相对于返回-ENOMEM 的一个办法是在引导时分配内存或为你的缓冲区保留物理 RAM 的顶端。我在第七章“Playing Dirty”中讲述了引导时的分配，但这个办法对模块不适用。保留 RAM 顶部可以通过向核心传递一个 mem=参数来完成。例如，如果你有 32M，参数 mem=31M 防止核心适用顶部一兆。你的模块以后可以用下面的代码来获得对该内存的访问：

```
dmabuf=vremap(0x1F00000 /*31MB*/, 0x100000 /* 1MB */ );
```

我自己分配DMA缓冲区的实现在*allocator.c*模块（和一个相配的头文件）中。你可以在*src/misc-modules*的示例文件中找到一个版本，最新版本总可以在我的FTP站点找到：*ftp://ftp.systemy.it/pub/develop*。你也可以找核心补丁*bigphysarea*，它和我的分配程序完成同样的工作。

总线地址

当进行 DMA 时，设备驱动程序必须与连在接口总线上的硬件对话，这里使用物理地址，但程序代码使用虚地址。

事实上，情况还要复杂一些。基于 DMA 的硬件使用总线地址，而不是物理地址。尽管在 PC 上，ISA 和 PCI 地址与物理地址一样，但并不是所有平台都是这样。有时接口总线是通过将 I/O 地址影射到不同物理地址的桥接电路被连接的。

Linux 核心通过引出定义在<*asm/io.h*>中的下列函数来提供一个可移植的解决方案。

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

其中 virt_to_bus 转换在驱动程序需要向一个 I/O 设备（如一个扩展板或 DMA 控制器）发送地址信息时必须使用，而 bus_to_virt 在收到来自连于总线上的硬件地址信息时必须使用。

如果你查看依赖于前面讲的 allocator 机制的代码，你会发现这些函数的使用例子。这些代码也依赖于 *vremap*，因为上下文：

（代码 292）

尽管与 DMA 无关，我们值得再了解两个核心引出的函数：

```
unsigned long virt_to_phys(volatile void * address);
```

---

* “碎片”这个词一般用于磁盘，表示文件在磁介质上不是连续地存放。这个概念同样适用于内存，即当每个虚拟地址空间都散布在整个物理RAM，很难为DMA的缓冲区请求分配连续的空闲页。

void * phys_to_virt(unsigned long address);

这两个函数在虚地址和物理地址之间进行转换；它们在程序代码需要和内存管理单元（MMU）或其它连在处理器地址线上的硬件对话时被用到。在 PC 平台上，这两对函数完成同样的工作；但将它们分开是重要的，既为了代码的清晰，也为了可移植性。

ISA 设备 DMA

ISA 总线允许两类 DMA 传送：“native 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 bus-master DMA”则完全由外围设备控制。后一种 DMA 类型很少用，所以不值得在这里讨论，因为它类似于 PCI 设备的 DMA，至少从驱动程序的角度看是这样的。一个 ISA bus-master 的例子是 1542 SCSI 控制器，它的驱动程序是核心源码中的 *drivers/scsi/aba1542.c*。

关于 native DMA，有三个实体参与了 ISA 总线上的 DMA 数据传送：

8237 DMA 控制器（DMAC）

控制器存有 DMA 传送的信息，如方向、内存地址、传送大小。它还有一个跟踪传送状态的计数器。当控制器收到一个 DMA 请求信号，它获得总线控制并驱动信号线以使设备可以读写数据。

外围设备

设备在准备好传送数据时，必须激活 DMA 请求信号。实际的传送由 DMAC 控制；当控制器通知了设备，硬件设备就顺序地从/往总线上读/写数据。

设备驱动程序

驱动程序要做的很少，它向 DMA 控制器提供方向、RAM 地址、传送大小。它还与外围设备对话准备好传送数据或在 DMA 结束时响应中断。

原先在PC中使用的DMA控制器能管理 4 个通道，每个通道与一组DMA寄存器关联，因此 4 个设备可以同时在控制器中存储它们的DMA信息。新一些的PC有相当于两套DMAC的设备[*]：第二个控制器（主）连向系统处理器，第一个（从）连在第二个控制器的 0 通道上[+]。通道编号为 0~7；4 号通道对ISA外围不可用，因为它是内部用来将从控制器级联到主控制器上。这样从控制器上可用的通道为 0~3（8 位通道），主控制器上为 5~7（16 位通道[♣]）。每次DMA传送的大小存储在控制器中，是一个 16 位数，表示总线周期数。因此从控制器的最大传送大小为 64KB，主控制器为 128KB。

由于 DMA 是系统范围的资源，因此核心协助处理它。它用一个 DMA 注册项提供 DMA 通道的请求和释放机制，并用一组函数配置 DMA 控制器的通道信息。

注册 DMA 的使用

你应该已经熟悉核心注册项了——我们在 I/O 端口和中断线那里见过它们。DMA 通道的注册项与其它类似。在包含了<asm/dma.h>后，下面的函数可以用来获得和释放 DMA 通道的所有权：

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

void free_dma(unsigned int channel);

参数 channel 是 0 到 7 之间的一个数，或更精确地说，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。参数 name 是确定设备的一个字符串。指定的名字出现在文件/proc/dma 中，可由拥护程序读出。

request_dma 在成功时返回 0，有错误时返回-EINVAL 或-EBUSY。前者表明请求的通道出了

---

[*] 这些电路现在是主板芯片组的一部分，但在几年前，它们是两个独立的 8237 芯片。

[+] 最初的PC只有一个控制器；第二个是在 286 平台上开始加上的。第二个控制器以主控制器的身份连接的原因是它能处理 16 位的传送，而第一个控制器一次只传送 8 位，它的存在只是为了后向兼容。

范围，后者表明有其它设备正占有这个设备。
我建议你对待 DMA 通道象对待 I/O 端口和中断线一样认真；在 *open* 时请求通道比从 init_module 中请求要好的多。推迟请求可以允许驱动程序间的一些共享；例如，你的声卡和你的模拟 I/O 接口可以共享 DMA 通道，只要它们不在同时使用。
我同时也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。这是请求这两个资源的常规顺序；依照这个顺序可以避免可能的死锁。注意每个使用 DMA 的设备需要一个 IRQ 线，不然无法表明数据传送的完成。
在典型的情况下，*open* 的代码看起来如下，这是个虚设的 *dad* 模块（DMA 获取设备）。*dad* 设备使用一个快速的中断处理程序，不支持共享 IRQ 线。
（代码 295　#1）
与 *open* 匹配的 *close* 实现如下所示：
（代码 295　#2）
下面是小一个装有声卡的系统上*/proc/dma* 文件的内容：

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

有趣的是注意到缺省的声卡驱动程序在系统引导时获得 DMA 通道，并永不释放。显示的 cascade 项只是占据一个位置，表明通道 4 对驱动程序不可用，如前所述。

与 DMA 控制器对话

注册完成后，驱动程序的主要工作是为正确的操作来配置 DMA 控制器。这项工作并不简单，好在核心引出了所有典型驱动程序所需的函数。
在 *read* 或 *write* 被调用，或者在预备异步传送时，驱动程序都需要配置 DMA 控制器。第二种情况，任务在 *open* 时或在对一个 *ioctl* 命令响应时被执行，这依赖于驱动程序及其实现策略。这里给出的代码一般是由 *read* 或 *write* 设备方法调用。
本小节对 DMA 控制器内部给出一个快速的概览，这样你就可以理解这里介绍的代码。如果你想学更多，我鼓励你阅读<asm/dma.h>和一些介绍 PC 体系结构的硬件手册。特别地，我并不关注 8 位和 16 位数据传输的区别。如果你在为 ISA 设备板子写设备驱动程序，你应该在设备的硬件手册里查找相关信息。
必须装入控制器的信息由三项组成：RAM 地址，必须传送的原子项数目（以字节或字为单位），传送的方向。为了这个目的，下面的函数由<asm/dma.h>引出：

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

说明通道是从设备读（DMA_MODE_READ）还是向设备写（DMA_MODE_WRITE）。还有第三个模式，DMA_MODE_CASCADE，用来释放对总线的控制。级联是第一个控制器连到第二个控制器上的方法，但它也可以由真正的 ISA bus-master 设备使用。我在这里不想讨论 bus-master。

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

分配 DMA 缓冲区的地址。这个函数将 addr 的低 24 位存入到控制器。参数 addr 必须是个总线地址（见“总线地址”）。

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

分配要传送的字节数。参数 count 对 16 位通道仍以字节为单位；在这种情况下，这个数必须是个偶数。

除了这几个函数，还有一些必须用来处理 DMA 设备的杂务工具：

♣ 一个总线I/O周期传送两个字节。

void disable_dma(unsigned int channel);

一个 DMA 通道可以在控制器内被关闭。在 DMAC 被配置之前，通道应该被关闭以防止不正确的操作（控制器通过 8 位数据传送编程，这样前面的函数都不能被原子地执行。）

void enable_dma(unsigned int channel);

这个函数告诉控制器这个 DMA 通道含有效数据。

int get_dma_residue(unsigned int channel);

驱动程序有时需要知道一个 DMA 传送是否结束了。这个函数返回尚待传送的字节数。如果成功传送完，则返回 0；如果控制器还在工作，返回值则不可预知（但不是 0）。这种不可预知性反映一个事实，即这个余数是个 16 位值，通过两个 8 位输入操作获得。

void clear_dma_ff(unsigned int channel);

这个函数清除 DMA flip-flop。flip-flop 用来控制对 16 位寄存器的访问。这些寄存器由两个连续的 8 位操作来访问，flip-flop 用来选中低字节（当它被清除时）或高字节（当它被置时）。flip-flop 在 8 位传输完后自动反转；在访问 DMA 积存器前必须清除一次 flip-flop。

用这些函数，驱动程序可以实现如下所示的一个函数来预备一个 DMA 传送：

（代码 297）

如下的函数用来检查 DMA 的成功完成：

```
int dad_dma_isdone(int channel)
{
    return(get_dma_residue(channel)==0);
}
```

剩下唯一要做的事就是配置设备板子。这个设备特定的任务通常包括读写几个 I/O 端口。设备在很多地方不同。例如，有的设备期望程序员告诉硬件 DMA 缓冲区有多大，有时驱动程序必须从设备中读出被硬写入的数值。为了配置板子，硬件手册是你唯一的朋友。

DMA 和 PCI 设备

DMA 的 PCI 实现比 ISA 上要简单的多。

PCI 支持多个 bus-master，而 DMA 就简化成 bus-mastering。需要读写主存的设备只需要简单地请求获得总线的控制，接着就可以直接控制电信号。PCI 的实现在硬件级更精巧，在设备驱动程序中更容易管理。

编写 PCI 上的 DMA 传送由下列步骤组成：

分配一个缓冲区

DMA 缓冲区在内存中必须是物理连续的，但没有 16MB 寻址能力的限制。一个 get_free_page 调用就足够了。不必在优先级中指定 GFP _DMA。如果你真的需要它，你可以转向（不鼓励）在前面“分配 DMA 缓冲区”中介绍过的更具进攻性的技术。

和设备对话

扩展设备必须被告知 DMA 缓冲区。这通常意味着将缓冲区的地址和大小写入几个设备积存器。有时，DMA 的大小由硬件设备决定，但这是设备相关的。传往 PCI 设备的地址必须是总线地址。

正如你所看到的，不存在为 PCI 设备编写的通用代码。一个典型的实现如下所示，但每个设备都不相同，配置信息量变化也很大。

（代码 298）

快速参考

本章介绍了与内存处理有关的下列符号。下面的列表不包括第一节中介绍的符号，因为其列表太大，而且那些符号在设备驱动程序中也很少用到。

#include <linux/mm.h>

所有与内存管理有关的函数和结构在这个头文件中定义。

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

这个函数居于 mmap 的核心，它将开始于物理地址 phys_add 的 size 字节映射到 virt_add。与虚拟空间相关联的保护位在 port 中指定。

#include <asm/io.h>

unsigned long virt_to_bus(volatile void * address);

void * bus_to_virt(unsigned int address);

unsigned long virt_to_phys(volatile void * address);

void * phys_to_virt(unsigned long address);

这些函数在虚拟和物理地址之间转换。必须使用总线地址来和外围设备对话，物理地址用来和 MMU 电路对话。

/proc/dma

这个文件包括 DMA 控制器中已分配通道的文本形式快照。基于 PCI 的 DMA 并不显示，因为各个板子独立工作，不必在 DMA 控制器中分配一个通道。

#include <asm/dma.h>

这个头文件定义了所有与 DMA 有关的函数和宏。要使用下面的符号就必须包含它。

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

void free_dma(unsigned int channel);

这些函数访问 DMA 注册项。在使用 ISA DMA 通道前必须注册。

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);

这些函数用来将 DMA 信息置入 DMA 控制器。addr 是总线地址。

void disable_dma(unsigned int channel);

void enable_dma(unsigned int channel);

在配置时，DMA 通道必须关闭。这些函数改变 DMA 通道的状态。

int get_dma_residue(unsigned int channel);

如果驱动程序想知道 DMA 传送进行的如何，它可以调用这个函数，返回尚需传送的数据字节数。DMA 成功完成后，函数返回 0；如果还在传送中，这个值是不可预知的。

void clear_dma_ff(unsigned int channel);

DMA flip-flop 被控制器用来用 8 位操作来传送 16 位的值。在传送任何数值到控制器前必须将其清楚。