LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

# 设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS



# 第 11 章　内存与 I/O 访问

由于 Linux 系统中提供了复杂的内存管理功能，所以内存的概念在 Linux 系统中变得相对复杂，出现了常规内存、高端内存、虚拟地址、逻辑地址、总线地址、物理地址、I/O 内存、设备内存、预留内存等概念。本章将系统地讲解内存和 I/O 的访问编程，带您走出内存和 I/O 的概念迷宫。

11.1 节讲解内存和 I/O 的硬件机制，主要涉及内存空间、I/O 空间和 MMU。

11.2 节讲解 Linux 的内存管理、内存区域的分布、常规内存与高端内存的区别。

11.3 节讲解 Linux 内存存取的方法，主要涉及内存动态申请以及通过虚拟地址存取物理地址的方法。

11.4 节讲解设备 I/O 内存和 I/O 端口的访问流程，这一节对于编写设备驱动意义非常重大，设备驱动使用此节的方法访问物理设备。

11.5 节讲解设备驱动中的 DMA 与 CACHE 一致性问题以及 DMA 编程方法。



# 11.1 CPU 与内存和 I/O

## 11.1.1 内存空间与 I/O 空间

在 X86 处理器中存在着 I/O 空间的概念，I/O 空间是相对于内存空间而言的，它通过特定的指令 in、out 来访问。端口号标识了外设的寄存器地址。Intel 语法的 in、out 指令格式如下：

```
IN 累加器, {端口号|DX}
OUT {端口号|DX},累加器
```

目前，大多数嵌入式微控制器如 ARM、PowerPC 等中并不提供 I/O 空间，而仅存在内存空间。内存空间可以直接通过地址、指针来访问，程序和程序运行中使用的变量和其他数据都存在于内存空间中。

内存地址可以直接由 C 语言指针操作，例如在 186 处理器中执行如下代码：

```
unsigned char *p = (unsigned char *)0xF000FF00;
*p=11;
```

以上程序的意义为在绝对地址 0xF0000+0xFF00（186 使用 16 位段地址和 16 位偏移地址）写入 11。

而在 ARM、PowerPC 等未采用段地址的处理器中，p 指向的内存空间就是 0xF000FF00，而*p = 11 就是在该地址写入 11。

再如，186 处理器启动后会在绝对地址 0xFFFF0（对应 C 语言指针是 0xF000FFF0，0xF000 为段地址，0xFFF0 为段内偏移）执行，请看下面的代码：

```
typedef void (*lpFunction) ( ); /* 定义一个无参数、无返回类型的函数指针类型 */
lpFunction lpReset = (lpFunction)0xF000FFF0; /* 定义一个函数指针，指向*/
/* CPU 启动后所执行第一条指令的位置 */
lpReset(); /* 调用函数 */
```

在以上程序中，没有定义任何一个函数实体，但是程序中却执行了这样的函数调用：lpReset()，它实际上起到了“软重启”的作用，跳转到 CPU 启动后第一条要执行的指令的位置。因此，可以通过函数指针调用一个没有函数体的“函数”，本质上只是换一个地址开始执行。

即便是在 X86 处理器中，虽然提供了 I/O 空间，如果由我们自己设计电路板，外设仍然可以只挂接在内存空间。此时，CPU 可以像访问一个内存单元那样访问外设 I/O 端口，而不需要设立专门的 I/O 指令。因此，内存空间是必须的，而 I/O 空间是可选的。图 11.1 给出了内存空间和 I/O 空间的对比。

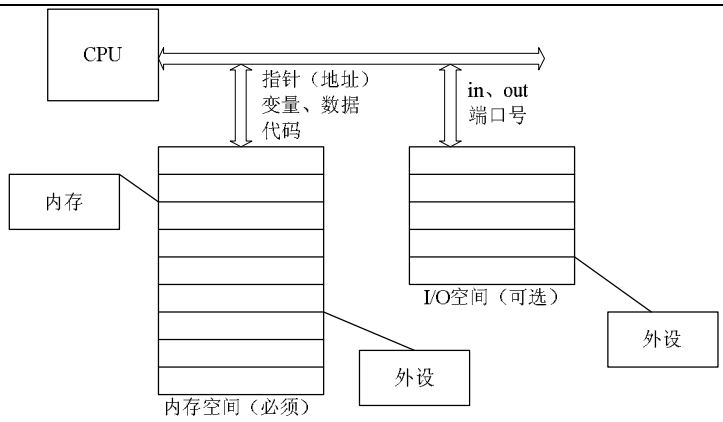


图 11.1　内存空间和 I/O 空间

### 11.1.2　内存管理单元 MMU

高性能处理器一般会提供一个内存管理单元（MMU），该单元辅助操作系统进行内存管理，提供虚拟地址和物理地址的映射、内存访问权限保护和 Cache 缓存控制等硬件支持。操作系统内核借助 MMU，可以让用户感觉到好像程序可以使用非常大的内存空间，从而使得编程人员在写程序时不用考虑计算机中的物理内存的实际容量。

为了理解基本的 MMU 操作原理，需先明晰几个概念。

- TLB：Translation Lookaside Buffer，即转换旁路缓存，TLB 是 MMU 的核心部件，它缓存少量的虚拟地址与物理地址的转换关系，是转换表的 Cache，因此也经常被称为“快表”。
- TTW：Translation Table walk，即转换表漫游，当 TLB 中没有缓冲对应的地址转换关系时，需要通过对内存中转换表（大多数处理器的转换表为多级页表，如图 11.2 所示）的访问来获得虚拟地址和物理地址的对应关系。TTW 成功后，结果应写入 TLB。

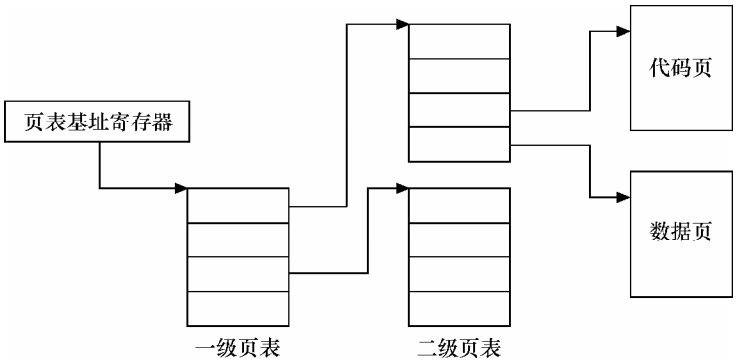


图 11.2　内存中的转换表

图 11.3 给出了一个典型的 ARM 处理器访问内存的过程，其他处理器也执行类似过程。当 ARM 要访问存储器时，MMU 先查找 TLB 中的虚拟地址表。如果 ARM 的结构支持分开的数据 TLB（DTLB）和指令 TLB（ITLB），则除取指令使用 ITLB 外，

其他的都使用 DTLB。ARM 处理器的 MMU 如图 11.3 所示。

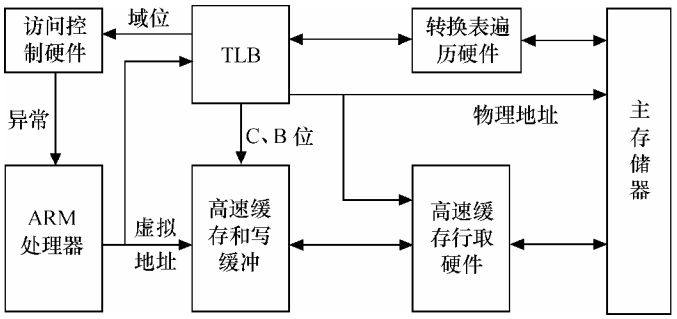


图 11.3　ARM 的内存管理单元

若 TLB 中没有虚拟地址的入口，则转换表遍历硬件从存放于主存储器中的转换表中获取地址转换信息和访问权限（即执行 TTW），同时将这些信息放入 TLB，它或者被放在一个没有使用的入口或者替换一个已经存在的入口。之后，在 TLB 条目中控制信息的控制下，当访问权限允许时，对真实物理地址的访问将在 Cache 或者在内存中发生，如图 11.4 所示。

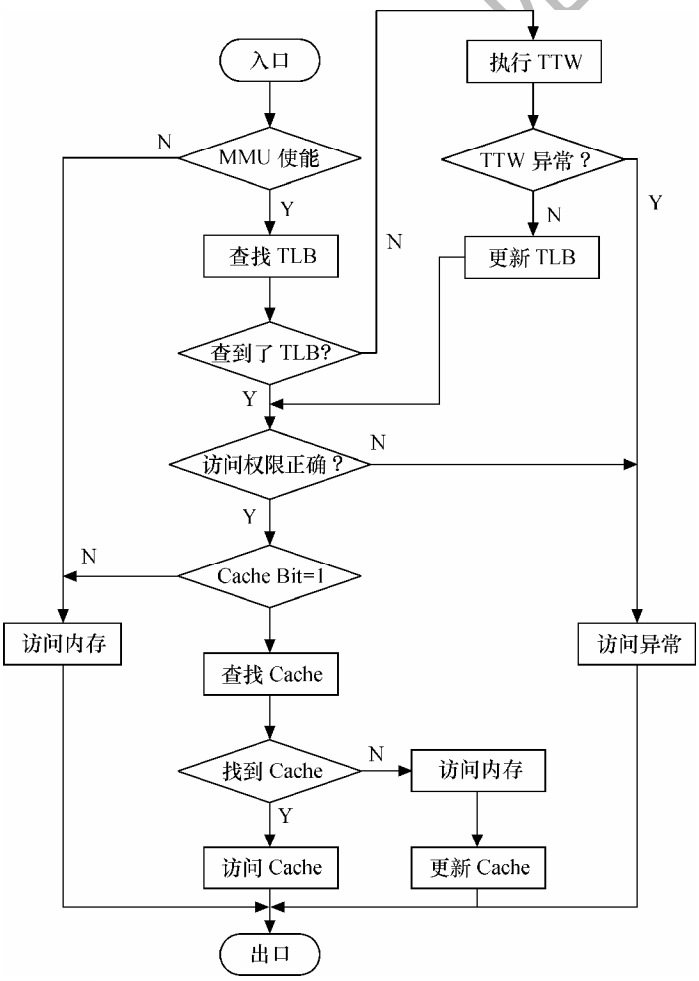

图 11.4　ARM CPU 进行数据访问的流程

ARM 中的 TLB 条目中的控制信息用于控制对对应地址的访问权限以及 Cache 的操作。

- C（高速缓存）和 B（缓冲）位被用来控制对应地址的高速缓存和写缓冲，并决定是否高速缓存。
- 访问权限和域位用来控制读写访问是否被允许。如果不允许，则 MMU 将向 ARM 处理器发送一个存储器异常，否则访问将被允许进行。

上述描述的 MMU 机制针对的虽然是 ARM 处理器，但 PowerPC、MIPS 等其他处理器也均有类似的操作。

MMU 具有虚拟地址和物理地址转换、内存访问权限保护等功能，这将使得 Linux 操作系统能单独为系统的每个用户进程分配独立的内存空间并保证用户空间不能访问内核空间的地址，为操作系统的虚拟内存管理模块提供硬件基础。

但是，MMU 并非对所有处理器都是必须的，例如常用的 SAMSUNG 基于 ARM7TDMI 系列的 S3C44B0X 不附带 MMU，其上无法运行老版的 Linux，而只能运行改版的μcLinux，但是新版的 Linux 2.6 则支持不带 MMU 的处理器。在嵌入式系统中，仍存在大量无 MMU 的处理器，Linux 2.6 为了更广泛地应用于嵌入式系统，融合了μcLinux，以支持 MMU-less 系统，如 Dragonball、ColdFire、Hitachi H8/300 等。

在 S3C2410 的 vivi 这个 Bootloader 中，建立了一个 4GB 物理地址与虚拟地址一一映射的一级页表，我们来追踪一下其创建过程。

vivi 的 main()函数首先会调用 mem_map_init()函数创建页表，其后调用 mmu_init()初始化并使能 MMU。mem_map_init()函数如代码清单 12.1 所示。

代码清单 12.1　vivi Bootloader 的 mem_map_init()函数

```
1  void mem_map_init(void)
2  {
3    #ifdef CONFIG_S3C2410_NAND_BOOT
4      /*CONFIG_S3C2410_NAND_BOOT = y ，在文件 include/autoconf.h 中定义
*/
5      mem_map_nand_boot();
6      /* 最终调用 mem_mepping_linear, 建立页表 */
7    #else
8      mem_map_nor();
9    #endif
10   cache_clean_invalidate(); /* 清空 cache,使无效 cache */
11   tlb_invalidate(); /* 使无效快表 TLB */
12 }
```

在 mem_map_nand_boot()中会调用 mem_mapping_linear()进行页表创建工作，如代码清单 12.2 所示。

代码清单 12.2　vivi Bootloader 中的页表创建

```
1  static inline void mem_mapping_linear(void)
```

```
    2 {
    3   unsigned long pageoffset, sectionNumber;
    4       putstr_hex("MMU table base address = 0x%", (unsigned
long)mmu_tlb_base);
    5
    6   /* 4GB 虚拟地址映射到相同的物理地址，均不能缓存 */
    7   for (sectionNumber = 0; sectionNumber < 4096; sectionNumber++)
    8   {
    9     pageoffset = (sectionNumber << 20);
    10    *(mmu_tlb_base + (pageoffset >> 20)) = pageoffset | MMU_SECDESC;
    11  }
    12
    13  /* 使 DRAM 的区域可缓存 */
    14  /* SDRAM 物理地址 0x3000000-0x33ffffff,
    15  DRAM_BASE=0x30000000,DRAM_SIZE=64M
    16   */
    17  for (pageoffset = DRAM_BASE; pageoffset < (DRAM_BASE + DRAM_SIZE);
    18    pageoffset += SZ_1M)
    19  {
    20      //DPRINTK(3, "Make DRAM section cacheable: 0x%08lx\n",
pageoffset);
    21    *(mmu_tlb_base + (pageoffset >> 20)) =
    22      pageoffset | MMU_SECDESC | MMU_CACHEABLE;
    23  }
    24 }
```

内核为 ARM920T 的 S3C2410 是一个 32 位的 CPU，它的地址空间为 4GB。共有 4 种内存映射模式，即 Fault（无映射）、Coarse Page（粗页表）、Section（段）、Fine Page（细页表）。

代码清单 12.2 使用了 ARM920T 内存映射的 Section 模式（实际可等同于页大小为 1MB 的情况），4GB 的虚拟空间被分成一个一个称为 Section 的单位。4GB 的虚拟内存总共可以被分成 4096 个段（1MB*4096 = 4GB），因此我们必须用 4096 个描述符来对这组段进行描述，每个描述符占用 4 个字节，故这组描述符的大小为 16KB，这 4096 个描述符构成的表格就是转换表。对于 SDRAM 区域，在描述符中使能了 Cache。

mmu_init()函数直接调用 arm920_setup()初始化 MMU，arm920_setup()函数包含一系列内嵌的汇编代码，用于控制 S3C2410 的 CP15 协处理器对 MMU 的处理。

## 11.2 Linux 内存管理

对于包含 MMU 的处理器而言，Linux 系统提供了复杂的存储管理系统，使得进程所能访问的内存达到 4GB。

在 Linux 系统中，进程的 4GB 内存空间被分为两个部分——用户空间与内核空间。用户空间地址一般分布为 0～3GB（即 PAGE_OFFSET，在 0x86 中它等于 0xC0000000），这样，剩下的 3～4GB 为内核空间，如图 11.5 所示。用户进程通常情况下只能访问用户空间的虚拟地址，不能访问内核空间虚拟地址。用户进程只有通过系统调用（代表

用户进程在内核态执行）等方式才可以访问到内核空间。


图 11.5　用户空间与内核空间

每个进程的用户空间都是完全独立、互不相干的，用户进程各自有不同的页表。而内核空间是由内核负责映射，它并不会跟着进程改变，是固定的。内核空间地址有自己对应的页表，内核的虚拟空间独立于其他程序。

Linux 中 1GB 的内核地址空间又被划分为物理内存映射区、虚拟内存分配区、高端页面映射区、专用页面映射区和系统保留映射区这几个区域，如图 11.6 所示。

一般情况下，物理内存映射区最大长度为 896MB，系统的物理内存被顺序映射在内核空间的这个区域中。当系统物理内存大于 896MB 时，超过物理内存映射区的那部分内存称为高端内存（而未超过物理内存映射区的内存通常被称为常规内存），内核在存取高端内存时必须将它们映射到高端页面映射区。

Linux 保留内核空间最顶部 FIXADDR_TOP～4GB 的区域作为保留区。

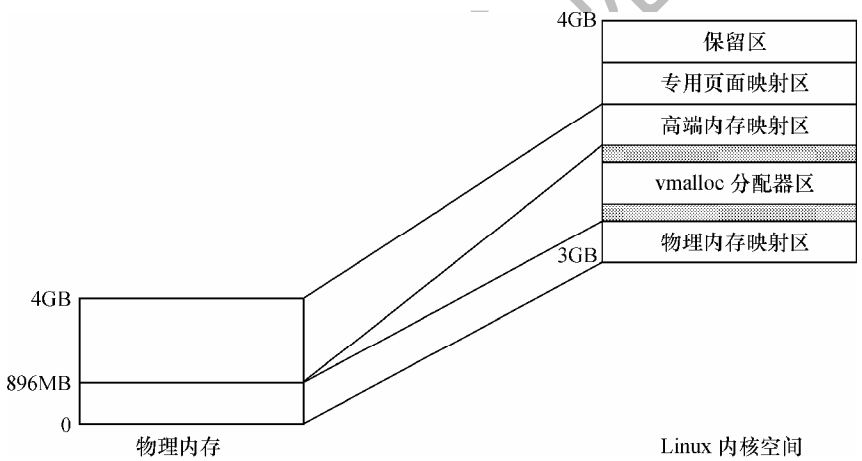


图 11.6　Linux 内核地址空间

紧接着最顶端的保留区以下的一段区域为专用页面映射区（FIXADDR_START～FIXADDR_ TOP），它的总尺寸和每一页的用途由 fixed_address 枚举结构在编译时预定义，用__fix_to_virt(index)可获取专用区内预定义页面的逻辑地址。其开始地址和结束地址宏定义如下：

```
    #define FIXADDR_START      (FIXADDR_TOP - __FIXADDR_SIZE)
    #define FIXADDR_TOP    ((unsigned long)__FIXADDR_TOP)
    #define __FIXADDR_TOP  0xfffff000
```

接下来，如果系统配置了高端内存，则位于专用页面映射区之下的就是一段高端内存映射区，其起始地址为 PKMAP_BASE，定义如下：

```
    #define PKMAP_BASE ( (FIXADDR_BOOT_START - PAGE_SIZE*(LAST_PKMAP + 1))
& PMD_MASK )
```

其中所涉及的宏定义如下：

```
#define FIXADDR_BOOT_START (FIXADDR_TOP - _ _FIXADDR_BOOT_SIZE)
#define LAST_PKMAP  PTRS_PER_PTE
#define PTRS_PER_PTE    512
#define PMD_MASK    (~(PMD_SIZE-1))
# define PMD_SIZE   (1UL << PMD_SHIFT)
#define PMD_SHIFT   21
```

在物理区和高端映射区之间为虚存内存分配区（VMALLOC_START～VMALLOC_END），用于 vmalloc()函数，它的前部与物理内存映射区有一个隔离带，后部与高端映射区也有一个隔离带，vmalloc 区域定义如下：

```
#define VMALLOC_OFFSET (8*1024*1024)
#define VMALLOC_START  (((unsigned long) high_memory +
   vmalloc_earlyreserve + 2*VMALLOC_OFFSET-1) & ~(VMALLOC_OFFSET-1))

#ifdef CONFIG_HIGHMEM   /*支持高端内存*/
# define VMALLOC_END    (PKMAP_BASE-2*PAGE_SIZE)
#else               /*不支持高端内存*/
# define VMALLOC_END    (FIXADDR_START-2*PAGE_SIZE)
#endif
```

当系统物理内存超过 4GB 时，必须使用 CPU 的扩展分页（PAE）模式所提供的 64 位页目录项才能存取到 4GB 以上的物理内存，这需要 CPU 的支持。加入了 PAE 功能的 Intel Pentium Pro 及其后的 CPU 允许内存最大可配置到 64GB，具备 36 位物理地址空间寻址能力。

由此可见，在 3～4GB 之间的内核空间中，从低地址到高地址依次为：物理内存映射区—隔离带—vmalloc 虚拟内存分配器—隔离带—高端内存映射区—专用页面映射区—保留区。

# 11.3 内存存取

## 11.3.1 用户空间内存动态申请

在用户空间动态申请内存的函数为 malloc()，这个函数在各种操作系统上的使用是一致的，malloc()申请的内存的释放函数为 free()。

malloc()的内存一定要被 free()，否则会造成内存泄漏。理想情况下，malloc()和 free()应成对出现，即谁申请，就由谁释放。例如下面的一段程序：

```
char * function(void)
{
  char *p;
  p = (char *)malloc(…);
  if(p==NULL)
    ...;
    ... /* 一系列针对 p 的操作 */
  return p;
}
```

在某处调用 function()，用完 function()中动态申请的内存后将其 free，如下：

```
char *q = function();
…
free(q);
```

上述代码并不合理的，因为违反了 malloc()和 free()成对出现的原则。不满足这个原则，会导致代码的耦合度增大，因为用户在调用 function()函数时需要知道其内部细节。

较好的做法是在调用处申请内存，并传入 function()函数，如下所示：

```
char *p=malloc(…);
if(p==NULL)
...;
function(p);
...
free(p);
p=NULL;
```

而函数 function()则接收参数 p，如下所示：

```
void function(char *p)
{
  ... /* 一系列针对 p 的操作 */
}
```

完全让 malloc()和 free()成对出现有时候很难做到，即便如此，也应尽力将 malloc()申请内存的释放限制在本模块范围之内。如果在 A 模块申请的内存需在 B 模块释放，一般而言，软件结构的设计可能存在问题。

### 11.3.2 内核空间内存动态申请

在 Linux 内核空间申请内存涉及的函数主要包括 kmalloc()、__get_free_pages()和 vmalloc()等。kmalloc()和_ _get_free_pages()（及其类似函数）申请的内存位于物理内存映射区域，而且在物理上也是连续的，它们与真实的物理地址只有一个固定的偏移，因此存在较简单的转换关系。而 vmalloc()在虚拟内存空间给出一块连续的内存区，实质上，这片连续的虚拟内存在物理内存中并不一定连续，而 vmalloc()申请的虚拟内存和物理内存之间也没有简单的换算关系。

#### 1．kmalloc()

```
void *kmalloc(size_t size, int flags);
```

给 kmalloc()的第一个参数是要分配的块的大小，第二个参数为分配标志，用于控制 kmalloc()的行为。

最常用的分配标志是 GFP_KERNEL，其含义是在内核空间的进程中申请内存。kmalloc()的底层依赖__get_free_pages()实现，分配标志的前缀 GFP 正好是这个底层函数的缩写。使用 GFP_ KERNEL 标志申请内存时，若暂时不能满足，则进程会睡眠等待页，即会引起阻塞，因此不能在中断上下文或持有自旋锁的时候使用 GFP_KERNE 申请内存。

在中断处理函数、tasklet 和内核定时器等非进程上下文中不能阻塞，此时驱动应当使用 GFP_ATOMIC 标志来申请内存。当使用 GFP_ATOMIC 标志申请内存时，若不存在空闲页，则不等待，直接返回。

其他的相对不常用的申请标志还包括 GFP_USER（用来为用户空间页分配内存，可

能阻塞）、GFP_HIGHUSER（类似 GFP_USER，但是从高端内存分配）、GFP_NOIO（不允许任何 I/O 初始化）、GFP_NOFS（不允许进行任何文件系统调用）、__GFP_DMA（要求分配在能够 DMA 的内存区）、__GFP_HIGHMEM（指示分配的内存可以位于高端内存）、__GFP_COLD（请求一个较长时间不访问的页）、__GFP_NOWARN（当一个分配无法满足时，阻止内核发出警告）、__GFP_HIGH（高优先级请求，允许获得被内核保留给紧急状况使用的最后的内存页）、__GFP_REPEAT（分配失败则尽力重复尝试）、__GFP_NOFAIL（标志只许申请成功，不推荐）和__GFP_NORETRY（若申请不到，则立即放弃）。

使用 kmalloc()申请的内存应使用 kfree()释放，这个函数的用法和用户空间的 free()类似。

### 2．__get_free_pages ()

__get_free_pages()系列函数/宏是 kmalloc()实现的基础，__get_free_pages()系列函数/宏包括 get_zeroed_page()、__get_free_page()和__get_free_pages()。

```
get_zeroed_page(unsigned int flags);
```

该函数返回一个指向新页的指针并且将该页清零。

```
__get_free_page(unsigned int flags);
```

该宏返回一个指向新页的指针但是该页不清零，它实际上为：

```
#define __get_free_page(gfp_mask) \
        __get_free_pages((gfp_mask),0)
```

就是调用了下面的__get_free_pages()申请 1 页。

```
__get_free_pages(unsigned int flags, unsigned int order);
```

该函数可分配多个页并返回分配内存的首地址，分配的页数为 $2^{order}$，分配的页也不清零。order 允许的最大值是 10（即 1024 页）或者 11（即 2048 页），依赖于具体的硬件平台。

__get_free_pages ()和 get_zeroed_page ()的实现中调用了 alloc_pages()函数，alloc_pages()既可以在内核空间分配，也可以在用户空间分配，其原型为：

```
struct page * alloc_pages(int gfp_mask, unsigned long order);
```

参数含义与__get_free_pages()类似，但它返回分配的第一个页的描述符而非首地址。

使用__get_free_pages()系列函数/宏申请的内存应使用下列函数释放：

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

如果申请和释放的 order 不一样，则会引起内存的混乱。

__get_free_pages 等函数在使用时，其申请标志的值与 kmalloc()完全一样，各标志的含义也与 kmalloc()完全一致，最常用的是 GFP_KERNEL 和 GFP_ATOMIC。

### 3．vmalloc()

vmalloc()一般用在为只存在于软件中（没有对应的硬件意义）的较大的顺序缓冲区分配内存，vmalloc()远大于__get_free_pages()的开销，为了完成 vmalloc()，新的页表需要被建立。因此，只是调用 vmalloc()来分配少量的内存（如 1 页）是不妥的。

vmalloc()申请的内存应使用 vfree()释放，vmalloc()和 vfree()的函数原型如下：

```
void *vmalloc(unsigned long size);
void vfree(void * addr);
```

vmalloc()不能用在原子上下文中，因为它的内部实现使用了标志为 GFP_KERNEL 的 kmalloc()。

使用 vmalloc 函数的一个例子函数是 create_module()系统调用，它利用 vmalloc()函数来获取被创建模块需要的内存空间。

### 4．slab 与内存池

在操作系统的运作过程中，经常会涉及大量对象的重复生成、使用和释放问题。在 Linux 系统中所用到的对象，比较典型的例子是 inode、task_struct 等。如果我们能够用合适的方法使得在对象前后两次被使用时分配在同一块内存或同一类内存空间且保留了基本的数据结构，就可以大大提高效率。slab 算法就是针对上述特点设计的。

- 创建 slab 缓存。

```
struct kmem_cache *kmem_cache_create(const char *name, size_t size,
    size_t align, unsigned long flags,
    void (*ctor)(void*, struct kmem_cache *, unsigned long),
    void (*dtor)(void*, struct kmem_cache *, unsigned long));
```

kmem_cache_create()用于创建一个 slab 缓存，它是一个可以驻留任意数目全部同样大小的后备缓存。参数 size 是要分配的每个数据结构的大小，参数 flags 是控制如何进行分配的位掩码，包括 SLAB_NO_REAP（即使内存紧缺也不自动收缩这块缓存）、SLAB_HWCACHE_ALIGN（每个数据对象被对齐到一个缓存行）、SLAB_CACHE_DMA（要求数据对象在 DMA 内存区分配）等。

- 分配 slab 缓存。

```
void *kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags);
```

上述函数在 kmem_cache_create()创建的 slab 后备缓冲中分配一块并返回首地址指针。

- 释放 slab 缓存。

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp);
```

上述函数释放由 kmem_cache_alloc()分配的缓存。

- 收回 slab 缓存。

```
int kmem_cache_destroy(struct kmem_cache *cachep);
```

代码清单 11.3 给出了 slab 缓存的使用范例。

代码清单 11.3　slab 缓存使用范例

```
1  /*创建 slab 缓存*/
2  static kmem_cache_t  *xxx_cachep;
3  xxx_cachep = kmem_cache_create("xxx", sizeof(struct xxx),
4              0, SLAB_HWCACHE_ALIGN|SLAB_PANIC, NULL, NULL);
5  /*分配 slab 缓存*/
6  struct xxx *ctx;
7  ctx = kmem_cache_alloc(xxx_cachep, GFP_KERNEL);
8  ...//使用 slab 缓存
9  /*释放 slab 缓存*/
10 kmem_cache_free(xxx_cachep, ctx);
11 kmem_cache_destroy(xxx_cachep);
```

除了 slab 以外，在 Linux 内核中还包含对内存池的支持，内存池技术也是一种非常经典的用于分配大量小对象的后备缓存技术。

Linux 内核中，与内存池相关的操作包括如下几种。

- 创建内存池。

```
mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn,
     mempool_free_t *free_fn, void *pool_data);
```

mempool_create()函数用于创建一个内存池，min_nr 参数是需要预分配对象的数目，alloc_fn 和 free_fn 是指向内存池机制提供的标准对象分配和回收函数的指针，其原型分别为：

```
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
```

和

```
typedef void (mempool_free_t)(void *element, void *pool_data);
```

pool_data 是分配和回收函数用到的指针，gfp_mask 是分配标记。只有当__GFP_WAIT 标记被指定时，分配函数才会休眠。

- 分配和回收对象。

在内存池中分配和回收对象需由以下函数来完成：

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

mempool_alloc()用来分配对象，如果内存池分配器无法提供内存，那么就可以用预分配的池。

- 回收内存池。

```
void mempool_destroy(mempool_t *pool);
```

mempool_create()函数创建的内存池需由 mempool_destroy()来回收。

### 11.3.3　虚拟地址与物理地址关系

对于内核物理内存映射区的虚拟内存，使用 virt_to_phys()可以实现内核虚拟地址转化为物理地址，virt_to_phys()的定义如代码清单 11.4 所示。

代码清单 11.4　virt_to_phys()函数

```
1 #define __pa(x)  ((unsigned long)(x)-PAGE_OFFSET)
2 extern inline unsigned long virt_to_phys(volatile void * address)
3 {
4   return __pa(address);
5 }
```

上面转换过程中调用的__pa()会将虚拟地址减去 PAGE_OFFSET，通常为 3GB。

与之对应的函数为 phys_to_virt()，它将物理地址转化为内核虚拟地址，phys_to_virt()的定义如代码清单 11.5 所示。

代码清单 11.5　phys_to_virt()函数

```
1 #define __va(x)  ((void *)((unsigned long)(x)+PAGE_OFFSET))
2 extern inline void * phys_to_virt(unsigned long address)
3 {
4   return __va(address);
5 }
```

上面转换过程中调用的__va()会将物理地址加上 PAGE_OFFSET ，通常为 3GB。

注意，上述方法仅适用于常规内存，高端内存的虚拟地址与物理地址之间不存在如此简单的换算关系。

# 11.4 设备 I/O 端口和 I/O 内存的访问

设备通常会提供一组寄存器来用于控制设备、读写设备和获取设备状态，即控制寄存器、数据寄存器和状态寄存器。这些寄存器可能位于 I/O 空间，也可能位于内存空间。当位于 I/O 空间时，通常被称为 I/O 端口，位于内存空间时，对应的内存空间被称为 I/O 内存。

## 11.4.1　Linux I/O 端口和 I/O 内存访问接口

### 1．I/O 端口

在 Linux 设备驱动中，应使用 Linux 内核提供的函数来访问定位于 I/O 空间的端口，这些函数包括如下几种。

- 读写字节端口（8 位宽）。

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

- 读写字端口（16 位宽）。

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

- 读写长字端口（32 位宽）。

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

- 读写一串字节。

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

- insb()从端口 port 开始读 count 个字节端口，并将读取结果写入 addr 指向的内存；outsb()将 addr 指向的内存的 count 个字节连续地写入 port 开始的端口。
- 读写一串字。

```
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```

- 读写一串长字。

```
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

上述各函数中 I/O 端口号 port 的类型高度依赖于具体的硬件平台，因此，只是写出了 unsigned。

### 2．I/O 内存

在内核中访问 I/O 内存之前，需首先使用 ioremap()函数将设备所处的物理地址映射到虚拟地址。ioremap()的原型如下：

```
void *ioremap(unsigned long offset, unsigned long size);
```

ioremap()与 vmalloc()类似，也需要建立新的页表，但是它并不进行 vmalloc()中所执行的内存分配行为。ioremap()返回一个特殊的虚拟地址，该地址可用来存取特定的物理地址范围。通过 ioremap()获得的虚拟地址应该被 iounmap()函数释放，其原型如下：

```
void iounmap(void * addr);
```

在设备的物理地址被映射到虚拟地址之后，尽管可以直接通过指针访问这些地址，但是可以使用 Linux 内核的如下一组函数来完成设备内存映射的虚拟地址的读写，这些函数如下所示。

- 读 I/O 内存。

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

与上述函数对应的较早版本的函数为（这些函数在 Linux 2.6 中仍然被支持）：

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
```

- 写 I/O 内存。

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

与上述函数对应的较早版本的函数为（这些函数在 Linux 2.6 中仍然被支持）：

```
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

- 读一串 I/O 内存。

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
```

- 写一串 I/O 内存。

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

- 复制 I/O 内存。

```
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

- 设置 I/O 内存。

```
void memset_io(void *addr, u8 value, unsigned int count);
```

#### 3．把 I/O 端口映射到内存空间

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

通过这个函数，可以把 port 开始的 count 个连续的 I/O 端口重映射为一段“内存空间”。然后就可以在其返回的地址上像访问 I/O 内存一样访问这些 I/O 端口。当不再需要这种映射时，需要调用下面的函数来撤销。

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

实际上，分析 ioport_map()的源代码可发现，映射到内存空间行为实际上是给开发人员制造的一个“假象”，并没有映射到内核虚拟地址，仅仅是为了让工程师可使用统一的 I/O 内存访问接口访问 I/O 端口。

### 11.4.2 申请与释放设备 I/O 端口和 I/O 内存

#### 1．I/O 端口申请

Linux 内核提供了一组函数用于申请和释放 I/O 端口。

```
struct resource *request_region(unsigned long first, unsigned long n,
const char *name);
```

这个函数向内核申请 *n* 个端口，这些端口从 first 开始，name 参数为设备的名称。如果分配成功返回值是非 NULL，如果返回 NULL，则意味着申请端口失败。

当用 request_region()申请的 I/O 端口使用完成后，应当使用 release_region()函数将它们归还给系统，这个函数的原型如下：

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

#### 2．I/O 内存申请

同样，Linux 内核也提供了一组函数用于申请和释放 I/O 内存的范围。

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

这个函数向内核申请 *n* 个内存地址，这些地址从 first 开始，name 参数为设备的名称。如果分配成功返回值是非 NULL，如果返回 NULL，则意味着申请 I/O 内存失败。

当用 request_mem_region()申请的 I/O 内存使用完成后，应当使用

release_mem_region()函数将它们归还给系统，这个函数的原型如下：

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

上述 request_region()和 release_mem_region()都不是必须的，但建议使用。其任务是检查申请的资源是否可用，如果可用则申请成功，并标志为已经使用，其他驱动想再次申请该资源时就会失败。

有许多设备驱动程序在没有申请 I/O 端口和 I/O 内存之前就直接访问了，这不够安全。

### 11.4.3 设备 I/O 端口和 I/O 内存访问流程

综合 11.3 节和本节的内容，可以归纳出设备驱动访问 I/O 端口和 I/O 内存的步骤。

I/O 端口访问的一种途径是直接使用 I/O 端口操作函数：在设备打开或驱动模块被加载时申请 I/O 端口区域，之后使用 inb()、outb()等进行端口访问，最后，在设备关闭或驱动被卸载时释放 I/O 端口范围。整个流程如图 11.7 所示。

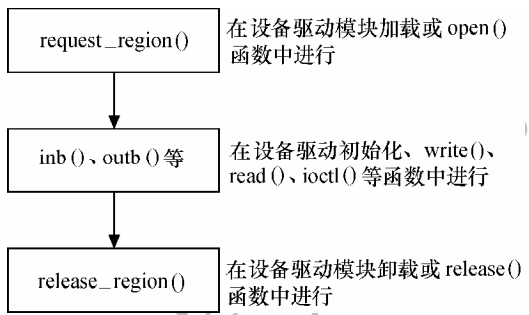


图 11.7 I/O 端口的访问流程（不映射到内存空间）

I/O 端口访问的另一种途径是将 I/O 端口映射为内存进行访问：在设备打开或驱动模块被加载时，申请 I/O 端口区域并使用 ioport_map()映射到内存，之后使用 I/O 内存的函数进行端口访问，最后，在设备关闭或驱动被卸载时释放 I/O 端口并释放映射。整个流程如图 11.8 所示。

I/O 内存的访问步骤如图 11.9 所示，首先是调用 request_mem_region()申请资源，接着将寄存器地址通过 ioremap()映射到内核空间虚拟地址，之后就可以通过 Linux 设备访问编程接口访问这些设备的寄存器了。访问完成后，应对 ioremap()申请的虚拟地址进行释放，并释放 release_mem_ region()申请的 I/O 内存资源。

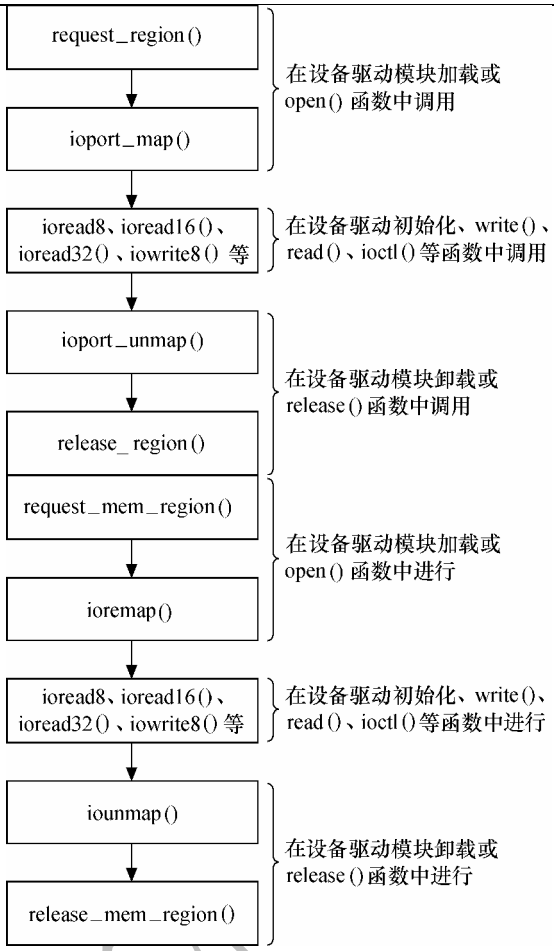


图 11.8　I/O 端口的访问流程（映射到内存空间）　　图 11.9　I/O 内存访问流程

### 11.4.4　将设备地址映射到用户空间

#### 1．内存映射与 VMA

一般情况下，用户空间是不可能也不应该直接访问设备的，但是，设备驱动程序中可实现 mmap()函数，这个函数可使得用户空间能直接访问设备的物理地址。实际上，mmap()实现了这样的一个映射过程：它将用户空间的一段内存与设备内存关联，当用户访问用户空间的这段地址范围时，实际上会转化为对设备的访问。

这种能力对于显示适配器一类的设备非常有意义，如果用户空间可直接通过内存映射访问显存的话，屏幕帧的各点的像素将不再需要一个从用户空间到内核空间的复制的过程。

mmap()必须以 PAGE_SIZE 为单位进行映射，实际上，内存只能以页为单位进行映射，若要映射非 PAGE_SIZE 整数倍的地址范围，要先进行页对齐，强行以 PAGE_SIZE 的倍数大小进行映射。

从 file_operations 文件操作结构体可以看出，驱动中 mmap()函数的原型如下：

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

驱动中的 mmap()函数将在用户进行 mmap()系统调用时最终被调用，mmap()系统调用的原型与 file_operations 中 mmap()的原型区别很大，如下所示：

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

参数 fd 为文件描述符，一般由 open()返回，fd 也可以指定为−1，此时须指定 flags 参数中的 MAP_ANON，表明进行的是匿名映射。

len 是映射到调用用户空间的字节数，它从被映射文件开头 offset 个字节开始算起，offset 参数一般设为 0，表示从文件头开始映射。

prot 参数指定访问权限，可取如下几个值的“或”：PROT_READ（可读）、PROT_WRITE（可写）、PROT_EXEC（可执行）和 PROT_NONE（不可访问）。

参数 addr 指定文件应被映射到用户空间的起始地址，一般被指定为 NULL，这样，选择起始地址的任务将由内核完成，而函数的返回值就是映射到用户空间的地址。其类型 caddr_t 实际上就是 void *。

当用户调用 mmap()的时候，内核会进行如下处理。

① 在进程的虚拟空间查找一块 VMA。

② 将这块 VMA 进行映射。

③ 如果设备驱动程序或者文件系统的 file_operations 定义了 mmap()操作，则调用它。

④ 将这个 VMA 插入到进程的 VMA 链表中。

file_operations 中 mmap()函数的第一个参数就是步骤①中找到的 VMA。

由 mmap()系统调用映射的内存可由 munmap()解除映射，这个函数的原型如下：

int munmap(caddr_t addr, size_t len );

驱动程序中 mmap()的实现机制是建立页表，并填充 VMA 结构体中 vm_operations_struct 指针。VMA 即 vm_area_struct，用于描述一个虚拟内存区域，VMA 结构体的定义如代码清单 11.6 所示。

代码清单 11.6　VMA 结构体

```
1  struct vm_area_struct
2  {
3    struct mm_struct *vm_mm; /* 所处的地址空间 */
4    unsigned long vm_start; /* 开始虚拟地址 */
5    unsigned long vm_end; /* 结束虚拟地址 */
6
7    pgprot_t vm_page_prot; /* 访问权限 */
8    unsigned long vm_flags; /* 标志 */
9    ...
10   /* 操作 VMA 的函数集指针 */
11   struct vm_operations_struct *vm_ops;
12
13   unsigned long vm_pgoff; /* 偏移（页帧号） */
14   struct file *vm_file;
15   void *vm_private_data;
16   ...
17 };
```

VMA 结构体描述的虚地址介于 vm_start 和 vm_end 之间，而其 vm_ops 成员指向

这个 VMA 的操作集。针对 VMA 的操作都被包含在 vm_operations_struct 结构体中，vm_operations_struct 结构体的定义如代码清单 11.7 所示。

代码清单 11.7　vm_operations_struct 结构体

```
1  struct vm_operations_struct
2  {
3    void(*open)(struct vm_area_struct *area); /*打开 VMA 的函数*/
4    void(*close)(struct vm_area_struct *area); /*关闭 VMA 的函数*/
5    struct page *(*nopage)(struct vm_area_struct *area, unsigned long
address,
6      int *type); /*访问的页不在内存时调用*/
7    int(*populate)(struct vm_area_struct *area, unsigned long address,
unsigned
8      long len, pgprot_t prot, unsigned long pgoff, int nonblock);
9    ...
10 };
```

在内核生成一个 VMA 后，它会调用该 VMA 的 open()函数，例如 fork 一个继承父继承资源的子进程时。但是，当用户进行 mmap()系统调用后，尽管 VMA 在设备驱动文件操作结构体的 mmap()被调用前就已产生，内核却不会调用 VMA 的 open()函数，通常需要在驱动的 mmap()函数中显示调用 vma->vm_ops->open()。代码清单 11.8 给出了一个 vm_operations_struct 的操作范例。

代码清单 11.8　vm_operations_struct 操作范例

```
1  static int xxx_mmap(struct file *filp, struct vm_area_struct *vma)
2  {
3    if (remap_pfn_range(vma, vma->vm_start, vm->vm_pgoff, vma->vm_end
- vma
4      ->vm_start, vma->vm_page_prot))  /* 建立页表 */
5      return  - EAGAIN;
6    vma->vm_ops = &xxx_remap_vm_ops;
7    xxx_vma_open(vma);
8    return 0;
9  }
10
11 void xxx_vma_open(struct vm_area_struct *vma)   //VMA 打开函数
12 {
13   ...
14     printk(KERN_NOTICE  "xxx  VMA  open,  virt  %lx,  phys  %lx\n",
vma->vm_start,
15     vma->vm_pgoff << PAGE_SHIFT);
16 }
17
18 void xxx_vma_close(struct vm_area_struct *vma)    //VMA 关闭函数
19 {
20   ...
21   printk(KERN_NOTICE "xxx VMA close.\n");
22 }
23
24 static struct vm_operations_struct xxx_remap_vm_ops =   //VMA 操作
```

```
结构体
    25 {
    26   .open = xxx_vma_open,
    27   .close = xxx_vma_close,
    28   ...
    29 };
```

第 2 行调用的 remap_pfn_range()创建页表，以 VMA 结构体的成员（VMA 的数据成员是内核根据用户的请求自己填充的）作为 remap_pfn_range()的参数，映射的虚拟地址范围是 vma->vm_ start 至 vma->vm_end。

remap_pfn_range()函数的原型如下：

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

其中的 addr 参数表示内存映射开始处的虚拟地址。remap_pfn_range()函数为 addr～addr+size 之间的虚拟地址构造页表。

pfn 是虚拟地址应该映射到的物理地址的页帧号，实际上就是物理地址右移 PAGE_SHIFT 位。若 PAGE_SIZE 为 4KB，则 PAGE_SHIFT 为 12，因为 PAGE_SIZE 等于 1<<PAGE_SHIFT。

prot 是新页所要求的保护属性。

在驱动程序中，我们能使用 remap_pfn_range()映射内存中的保留页（如 X86 系统中的 640KB～1MB 区域）和设备 I/O 内存，另外，kmalloc()申请的内存若要被映射到用户空间可以通过 mem_map_ reserve()设置为保留后进行。代码清单 11.9 给出了映射 kmalloc()申请的内存到用户空间的典型范例。

**代码清单 11.9　映射 kmalloc()申请的内存到用户空间范例**

```
    1  /*内核模块加载函数*/
    2  int __init kmalloc_map_init(void)
    3  {
    4    ...//申请设备号、添加 cdev 结构体
    5    buffer = kmalloc(BUF_SIZE, GFP_KERNEL); //申请 buffer
    6
    7    for (page = virt_to_page(buffer); page < virt_to_page(buffer +
BUF_SIZE);
    8      page++)
    9    {
    10     mem_map_reserve(page); //置页为保留
    11   }
    12 }
    13 /*mmap()函数*/
    14 static int kmalloc_map_mmap(struct file *filp, struct vm_area_struct
*vma)
    15 {
    16   unsigned long page, pos;
```

```
17  unsigned long start = (unsigned long)vma->vm_start;
18  unsigned long size = (unsigned long)(vma->vm_end - vma->vm_start);
19  printk(KERN_INFO "mmaptest_mmap called\n");
20  /* 用户要映射的区域太大 */
21  if (size > BUF_SIZE)
22    return  - EINVAL;
23
24  pos = (unsigned long)buffer;
25  /* 映射 buffer 中的所有页 */
26  while (size > 0)
27  {
28    /* 每次映射一页 */
29    page = virt_to_phys((void*)pos);
30    if (remap_page_range(start, page, PAGE_SIZE, PAGE_SHARED))
31      return  - EAGAIN;
32    start += PAGE_SIZE;
33    pos += PAGE_SIZE;
34    size -= PAGE_SIZE;
35  } return 0;
36 }
```

第 30 行调用 remap_page_range(start, page, PAGE_SIZE, PAGE_SHARED)的第 4 个参数 PAGE_ SHARED 实际上是_PAGE_PRESENT | _PAGE_USER | _PAGE_RW，表明可读写并映射到用户空间。

通常，I/O 内存被映射时需要是 nocache 的，这时候，我们应该对 vma->vm_page_prot 设置 nocache 标志之后再映射，如代码清单 11.10 所示。

代码清单 11.10　以 nocache 方式将内核空间映射到用户空间

```
1  static int xxx_nocache_mmap(struct file *filp, struct vm_area_struct
*vma)
2  {
3      vma->vm_page_prot  =  pgprot_noncached(vma->vm_page_prot);// 赋
nocache 标志
4    vma->vm_pgoff = ((u32)map_start >> PAGE_SHIFT);
5    //映射
6          if  (remap_pfn_range(vma,  vma->vm_start,  vma->vm_pgoff,
vma->vm_end - vma
7      ->vm_start, vma->vm_page_prot))
8      return  - EAGAIN;
9    return 0;
10 }
```

上述代码第 3 行的 pgprot_noncached()是一个宏，它高度依赖于 CPU 体系结构，

ARM 的 pgprot_noncached()定义如下：

```
    #define pgprot_noncached(prot) __pgprot(pgprot_val(prot)          &
~(L_PTE_CACHEABLE | L_PTE_ BUFFERABLE))
```

另一个比 pgprot_noncached()稍微少一些限制的宏是 pgprot_writecombine()，它的定义如下：

```
    #define   pgprot_writecombine(prot)   __pgprot(pgprot_val(prot)   &
~L_PTE_CACHEABLE)
```

pgprot_noncached()实际禁止了相关页的 Cache 和写缓冲（write buffer），pgprot_writecombine()则没有禁止写缓冲。ARM 的写缓冲器是一个非常小的 FIFO 存储器，位于处理器核与主存之间，其目的在于将处理器核和 Cache 从较慢的主存写操作中解脱出来。写缓冲区与 Cache 在存储层次上处于同一层次，但是它只作用于写主存。

### 2．nopage()函数

除了 remap_pfn_range()以外，在驱动程序中实现 VMA 的 nopage()函数通常可以为设备提供更加灵活的内存映射途径。当访问的页不在内存，即发生缺页异常时，nopage()会被内核自动调用。这是因为，当发生缺页异常时，系统会经过如下处理过程。

① 找到缺页的虚拟地址所在的 VMA。

② 如果必要，分配中间页目录表和页表。

③ 如果页表项对应的物理页面不存在，则调用这个 VMA 的 nopage()方法，它返回物理页面的页描述符。

④ 将物理页面的地址填充到页表中。

实现 nopage()后，用户空间可以通过 mremap()系统调用重新绑定映射区域所绑定的地址，代码清单 11.11 给出了一个设备驱动中使用 nopage()的典型范例。

代码清单 11.11　nopage()函数使用范例

```
 1  static int xxx_mmap(struct file *filp, struct vm_area_struct *vma)
 2  {
 3    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
 4    if (offset >= __pa(high_memory) || (filp->f_flags &O_SYNC))
 5      vma->vm_flags |= VM_IO;
 6    vma->vm_flags |= VM_RESERVED; //预留
 7    vma->vm_ops = &xxx_nopage_vm_ops;
 8    xxx_vma_open(vma);
 9    return 0;
10 }
11
12 struct page *xxx_vma_nopage(struct vm_area_struct *vma, unsigned
long
13   address, int *type)
14 {
15   struct page *pageptr;
16   unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
17   unsigned long physaddr = address - vma->vm_start + offset; /* 物
理地址 */
```

```
18   unsigned long pageframe = physaddr >> PAGE_SHIFT; /* 页帧号 */
19   if (!pfn_valid(pageframe))    /* 页帧号有效? */
20     return NOPAGE_SIGBUS;
21   pageptr = pfn_to_page(pageframe);  /* 页帧号->页描述符 */
22   get_page(pageptr);    /* 获得页，增加页的使用计数 */
23   if (type)
24     *type = VM_FAULT_MINOR;
25   return pageptr;   /*返回页描述符 */
26 }
```

上述函数对常规内存进行映射，返回一个页描述符，可用于扩大或缩小映射的内存区域。

由此可见，nopage()与 remap_pfn_range()的一个较大区别在于 remap_pfn_range()一般用于设备内存映射，而 nopage()还可用于 RAM 映射。

大多数设备驱动都不需要提供设备内存到用户空间的映射能力，因为，对于串口等面向流的设备而言，实现这种映射毫无意义。

# 11.5 I/O 内存静态映射

在将 Linux 移植到目标电路板的过程中，通常会建立外设 I/O 内存物理地址到虚拟地址的静态映射，这个映射通过在电路板对应的 map_desc 结构体数组中添加新的成员来完成，map_desc 结构体的定义如代码清单 11.12 所示。

代码清单 11.12　map_desc 结构体

```
1 struct map_desc
2 {
3   unsigned long virtual;//虚拟地址
4   unsigned long pfn ;// __phys_to_pfn(phy_addr)
5   unsigned long length; //大小
6   unsigned int type;//类型
7 };
```

例如，S3C2440 ARM 评估板 SMDK2440 的 map_desc 结构体数组定义如代码清单 11.13 所示，位于文件/arch/ arm/mach-smdk2440.c。

代码清单 11.13　SMDK2440 评估板的 map_desc 数组

```
1  static struct map_desc smdk2440_iodesc[] __initdata = {
2   /* ISA IO空间映射 */
3   {
4       .virtual    = (u32)S3C24XX_VA_ISA_WORD,
5       .pfn        = __phys_to_pfn(S3C2410_CS2),
6       .length     = 0x10000,
7       .type       = MT_DEVICE,
8   }, {
9       .virtual    = (u32)S3C24XX_VA_ISA_WORD + 0x10000,
10      .pfn        = __phys_to_pfn(S3C2410_CS2 + (1<<24)),
```

```
    .length     = SZ_4M,
    .type       = MT_DEVICE,
}, {
    .virtual    = (u32)S3C24XX_VA_ISA_BYTE,
    .pfn        = __phys_to_pfn(S3C2410_CS2),
    .length     = 0x10000,
    .type       = MT_DEVICE,
}, {
    .virtual    = (u32)S3C24XX_VA_ISA_BYTE + 0x10000,
    .pfn        = __phys_to_pfn(S3C2410_CS2 + (1<<24)),
    .length     = SZ_4M,
    .type       = MT_DEVICE,
}
};
```

Linux 操作系统移植到特定平台上，MACHINE_START 到 MACHINE_END 宏之间的定义针对特定电路板而设计，其中的 map_io()成员函数完成 I/O 内存的静态映射，代码清单 11.14 给出了 SMDK2440 电路板的 MACHINE_START、MACHINE_END 宏的例子。

代码清单 11.14　SMDK2440 MACHINE_START、MACHINE_END 宏

```
1  MACHINE_START(S3C2440, "SMDK2440")
2   /* Maintainer: Ben Dooks <ben@fluff.org> */
3   .phys_io    = S3C2410_PA_UART,
4   .io_pg_offst    = (((u32)S3C24XX_VA_UART) >> 18) & 0xfffc,
5   .boot_params    = S3C2410_SDRAM_PA + 0x100,
6
7   .init_irq   = s3c24xx_init_irq,
8   .map_io     = smdk2440_map_io,
9   .init_machine   = smdk2440_machine_init,
10  .timer      = &s3c24xx_timer,
11  MACHINE_END
```

第 8 行赋值给 map_io 的 smdk2440_map_io()函数完成 SMDK2440 电路板 I/O 内存的静态映射，最终调用的是 cpu->map_io()建立 map_desc 数组中物理内存和虚拟内存的静态映射关系。

在一个已经移植好 OS 的内核中，驱动工程师完全可以对非常规内存区域的 I/O 内存（外设控制器寄存器、MCU 内部集成的外设控制器寄存器等）依照电路板的资源使用情况添加到 map_ desc 数组中，代码清单 11.15 的例子给出了内存空间资源的使用情况（注释部分）与 map_desc 数组的对应关系。

代码清单 11.15　根据电路板内存资源情况定义 map_desc

```
1  /*
2   * 逻辑地址 物理地址
3   * e8000000  40000000    PCI memory      PHYS_PCI_MEM_BASE   (max
512M)
4   * ec000000  61000000    PCI 配置空间  PHYS_PCI_CONFIG_BASE    (max
16M)
5   * ed000000  62000000    PCI V3 regs     PHYS_PCI_V3_BASE    (max
64k)
6   * ee000000  60000000    PCI IO          PHYS_PCI_IO_BASE    (max
16M)
7   * ef000000          Cache flush
8   * f1000000  10000000    核心模块寄存器
```

```
 9   * f1100000  11000000    系统控制寄存器
10   * f1200000  12000000    EBI 寄存器
11   * f1300000  13000000    计数器/定时器
12   * f1400000  14000000    中断控制器
13   * f1600000  16000000    UART 0
14   * f1700000  17000000    UART 1
15   * f1a00000  1a000000    调试用 LEDs
16   * f1b00000  1b000000    GPIO
17   */
18
19  static struct map_desc ap_io_desc[] __initdata = {
20   {
21       .virtual    = IO_ADDRESS(INTEGRATOR_HDR_BASE),
22       .pfn        = __phys_to_pfn(INTEGRATOR_HDR_BASE),
23       .length     = SZ_4K,
24       .type       = MT_DEVICE
25   }, {
26       .virtual    = IO_ADDRESS(INTEGRATOR_SC_BASE),
27       .pfn        = __phys_to_pfn(INTEGRATOR_SC_BASE),
28       .length     = SZ_4K,
29       .type       = MT_DEVICE
30   }, {
31       .virtual    = IO_ADDRESS(INTEGRATOR_EBI_BASE),
32       .pfn        = __phys_to_pfn(INTEGRATOR_EBI_BASE),
33       .length     = SZ_4K,
34       .type       = MT_DEVICE
35   }, {
36       .virtual    = IO_ADDRESS(INTEGRATOR_CT_BASE),
37       .pfn        = __phys_to_pfn(INTEGRATOR_CT_BASE),
38       .length     = SZ_4K,
39       .type       = MT_DEVICE
40   }, {
41       .virtual    = IO_ADDRESS(INTEGRATOR_IC_BASE),
42       .pfn        = __phys_to_pfn(INTEGRATOR_IC_BASE),
43       .length     = SZ_4K,
44       .type       = MT_DEVICE
45   }, {
46       .virtual    = IO_ADDRESS(INTEGRATOR_UART0_BASE),
47       .pfn        = __phys_to_pfn(INTEGRATOR_UART0_BASE),
48       .length     = SZ_4K,
49       .type       = MT_DEVICE
50   }, {
```

```
51      .virtual    = IO_ADDRESS(INTEGRATOR_UART1_BASE),
52      .pfn        = __phys_to_pfn(INTEGRATOR_UART1_BASE),
53      .length     = SZ_4K,
54      .type       = MT_DEVICE
55  }, {
56      .virtual    = IO_ADDRESS(INTEGRATOR_DBG_BASE),
57      .pfn        = __phys_to_pfn(INTEGRATOR_DBG_BASE),
58      .length     = SZ_4K,
59      .type       = MT_DEVICE
60  }, {
61      .virtual    = IO_ADDRESS(INTEGRATOR_GPIO_BASE),
62      .pfn        = __phys_to_pfn(INTEGRATOR_GPIO_BASE),
63      .length     = SZ_4K,
64      .type       = MT_DEVICE
65  }, {
66      .virtual    = PCI_MEMORY_VADDR,
67      .pfn        = __phys_to_pfn(PHYS_PCI_MEM_BASE),
68      .length     = SZ_16M,
69      .type       = MT_DEVICE
70  }, {
71      .virtual    = PCI_CONFIG_VADDR,
72      .pfn        = __phys_to_pfn(PHYS_PCI_CONFIG_BASE),
73      .length     = SZ_16M,
74      .type       = MT_DEVICE
75  }, {
76      .virtual    = PCI_V3_VADDR,
77      .pfn        = __phys_to_pfn(PHYS_PCI_V3_BASE),
78      .length     = SZ_64K,
79      .type       = MT_DEVICE
80  }, {
81      .virtual    = PCI_IO_VADDR,
82      .pfn        = __phys_to_pfn(PHYS_PCI_IO_BASE),
83      .length     = SZ_64K,
84      .type       = MT_DEVICE
85  }
86 };
```

此后，在设备驱动中访问经过 map_desc 数组映射后的 I/O 内存时，直接在 map_desc 中该段的虚拟地址上加上相应的偏移即可，不再需要使用 ioremap()。

# 11.6 DMA

DMA 是一种无须 CPU 的参与就可以让外设与系统内存之间进行双向数据传输的硬件机制。使用 DMA 可以使系统 CPU 从实际的 I/O 数据传输过程中摆脱出来，从而大大提高系统的吞吐率。DMA 通常与硬件体系结构特别是外设的总线技术密切相关。

DMA 方式的数据传输由 DMA 控制器（DMAC）控制，在传输期间，CPU 可以并发地执行其他任务。当 DMA 结束后，DMAC 通过中断通知 CPU 数据传输已经结束，然后由 CPU 执行相应的中断服务程序进行后处理。

## 11.6.1 DMA 与 Cache 一致性

Cache 和 DMA 本身似乎是两个毫不相关的事物。Cache 被用做 CPU 针对内存的缓存，利用程序的空间局部性和时间局部性原理，达到较高的命中率从而避免 CPU 每次都一定要与相对慢速的内存交互数据来提高数据的访问速率。DMA 可以用做内存与外设之间传输数据的方式，这种传输方式之下，数据并不需要经过 CPU 中转。

假设 DMA 针对内存的目的地址与 Cache 缓存的对象没有重叠区域（如图 11.10 所示），DMA 和 Cache 之间将相安无事。但是，如果 DMA 的目的地址与 Cache 所缓存的内存地址访问有重叠（如图 11.11 所示），经过 DMA 操作，Cache 缓存对应的内存的数据已经被修改，而 CPU 本身并不知道，它仍然认为 Cache 中的数据就是内存中的数据，以后访问 Cache 映射的内存时，它仍然使用陈旧的 Cache 数据。这样就发生 Cache 与内存之间数据“不一致性”的错误。

所谓 Cache 数据与内存数据的不一致性，是指在采用 Cache 的系统中，同样一个数据可能既存在于 Cache 中，也存在于主存中，Cache 与主存中的数据一样则具有一致性，数据若不一样则具有不一致性。

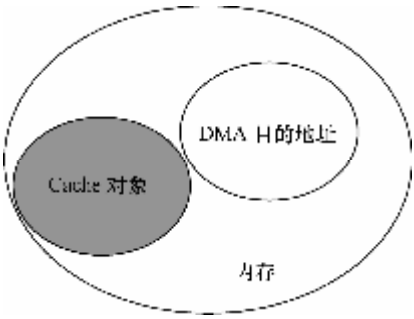


图 11.10　DMA 目的地址与 Cache 对象不交叉

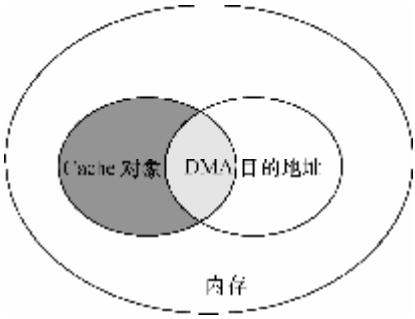


图 11.11　DMA 目的地址与 Cache 对象交叉

需要特别注意的是，Cache 与内存的一致性问题经常被初学者遗忘。在发生 Cache 与内存不一致性错误后，驱动将无法正常运行。如果没有相关的背景知识，工程师几乎无法定位错误的原因，因为看起来所有的程序都是完全正确的。

解决由于 DMA 导致的 Cache 一致性问题的最简单方法是直接禁止 DMA 目标地址范围内内存的 Cache 功能。当然，这将牺牲性能，但是却更可靠，图 11.12 所示为 Cache 和 DMA 在考虑性能和易用两个方面时处于跷跷板两端的比重。

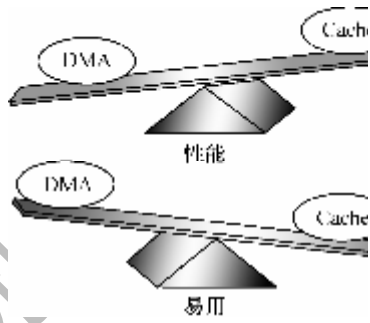


图 11.12　跷跷板两端的 Cache 与 D

Cache 的不一致问题并发只是发生在 DMA 的情况下，实际上，还存在于其他情况。例如，对于带 MMU 功能的 ARM 处理器，在开启 MMU 之前，需要先置 Cache 无效，TLB 也是如此，代码清单 11.16 所给出的一段汇编被用来完成此任务。

代码清单 11.16　置 ARM 的 Cache 无效

```
1 /* 使 cache 无效 */
2 "mov  r0, #0\n"
3 "mcr  p15, 0, r0, c7, c7, 0\n"    /* 使数据和指令 cache 无效 */
4 "mcr  p15, 0, r0, c7, c10, 4\n"   /* 放空写缓冲 */
5 "mcr  p15, 0, r0, c8, c7, 0\n"    /* 使 TLB 无效 */
```

### 11.6.2　Linux 下的 DMA 编程

首先 DMA 本身不属于一种等同于字符设备、块设备和网络设备的外设，它只是外设与内存交互数据的一种方式。因此，本节的标题不是“Linux 下的 DMA 驱动”而是“Linux 下的 DMA 编程”。

内存中用于与外设交互数据的一块区域被称做 DMA 缓冲区，在设备不支持 scatter/gather CSG，分散/聚集操作的情况下，DMA 缓冲区必须是物理上连续的。

对于 ISA 设备而言，其 DMA 操作只能在 16MB 以下的内存中进行，因此，在使用 kmalloc()和_ _get_free_pages()及其类似函数申请 DMA 缓冲区时应使用 GFP_DMA 标志，这样能保证获得的内存是具备 DMA 能力的（DMA-capable）。

内核中定义了__get_free_pages()针对 DMA 的“快捷方式”__get_dma_pages()，它在申请标志中添加了 GFP_DMA，如下所示：

```
#define __get_dma_pages(gfp_mask, order) \
        __get_free_pages((gfp_mask) | GFP_DMA,(order))
```

如果不想使用 $\log_2$size 即 order 为参数申请 DMA 内存，则可以使用另一个函数 dma_mem_ alloc()，其源代码如代码清单 11.17 所示。

代码清单 11.17　dma_mem_alloc()函数

```
1 static unsigned long dma_mem_alloc(int size)
2 {
3   int order = get_order(size);//大小->指数
4   return __get_dma_pages(GFP_KERNEL, order);
5 }
```

基于 DMA 的硬件使用总线地址而非物理地址，总线地址是从设备角度上看到的内存地址，物理地址则是从 CPU 角度上看到的未经转换的内存地址（经过转换的为虚拟地址）。虽然在 PC 上，对于 ISA 和 PCI 而言，总线地址即为物理地址，但并非每个平台都是如此。因为有时候接口总线通过桥接电路被连接，桥接电路会将 I/O 地址映射为不同的物理地址。例如，在 PReP（PowerPC Reference Platform）系统中，物理地址 0 在设备端看起来是 0x80000000，而 0 通常又被映射为虚拟地址 0xC0000000，所以同一地址就具备了三重身份：物理地址 0、总线地址 0x80000000 及虚拟地址 0xC0000000。还有一些系统提供了页面映射机制，它能将任意的页面映射为连续的外设总线地址。内核提供了如下函数用于进行简单的虚拟地址/总线地址转换：

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

在使用 IOMMU 或反弹缓冲区的情况下，上述函数一般不会正常工作。而且，这两个函数并不建议使用。如图 11.13 所示，IOMMU 的工作原理与 CPU 内的 MMU 非常类似，不过它针对的是外设总线地址和内存地址之间的转化。由于 IOMMU 可以使得外设看到“虚拟地址”，因此在使用 IOMMU 的情况下，在修改映射寄存器后，可以使得 SG 中分段的缓冲区地址对外设变得连续。

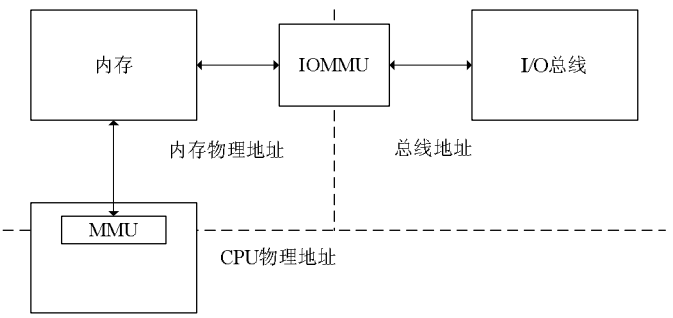


图 11.13　MMU 与 IOMMU

设备并不一定能在所有的内存地址上执行 DMA 操作，在这种情况下应该通过下列函数执行 DMA 地址掩码：

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

例如，对于只能在 24 位地址上执行 DMA 操作的设备而言，就应该调用 dma_set_mask (dev, 0xffffff)。

DMA 映射包括两个方面的工作：分配一片 DMA 缓冲区；为这片缓冲区产生设备可访问的地址。同时，DMA 映射也必须考虑 Cache 一致性问题。内核中提供了以下函数用于分配一个 DMA 一致性的内存区域：

```
    void * dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t
*handle, gfp_t gfp);
```

上述函数的返回值为申请到的 DMA 缓冲区的虚拟地址，此外，该函数还通过参数 handle 返回 DMA 缓冲区的总线地址。handle 的类型为 dma_addr_t，代表的是总线地址。

dma_alloc_coherent()申请一片 DMA 缓冲区，进行地址映射并保证该缓冲区的 Cache 一致性。与 dma_alloc_coherent()对应的释放函数为：

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

以下函数用于分配一个写合并（writecombining）的 DMA 缓冲区：

```
    void * dma_alloc_writecombine(struct device *dev, size_t size, dma_addr_t
*handle, gfp_t gfp);
```

与 dma_alloc_writecombine()对应的释放函数 dma_free_writecombine()实际上就是 dma_free_ coherent()，因为它定义为：

```
    #define dma_free_writecombine(dev,size,cpu_addr,handle) \
         dma_free_coherent(dev,size,cpu_addr,handle)
```

此外，Linux 内核还提供了 PCI 设备申请 DMA 缓冲区的函数 pci_alloc_consistent()，其原型为：

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

对应的释放函数为 pci_free_consistent()，其原型为：

```
    void  pci_free_consistent(struct  pci_dev  *pdev,  size_t  size,  void
*cpu_addr,     dma_addr_t dma_addr);
```

相对于一致性 DMA 映射而言，流式 DMA 映射的接口较为复杂。对于单个已经分配的缓冲区而言，使用 dma_map_single()可实现流式 DMA 映射，该函数原型为：

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

如果映射成功，返回的是总线地址，否则，返回 NULL。第 4 个参数为 DMA 的方向，可能的值包括 DMA_TO_DEVICE、DMA_FROM_DEVICE、DMA_BIDIRECTIONAL 和 DMA_NONE。

dma_map_single()的反函数为 dma_unmap_single()，原型是：

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

通常情况下，设备驱动不应该访问 unmap 的流式 DMA 缓冲区，如果一定要这么做，可先使用如下函数获得 DMA 缓冲区的拥有权：

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

在驱动访问完 DMA 缓冲区后，应该将其所有权返还给设备，通过如下函数完成：

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

如果设备要求较大的 DMA 缓冲区，在其支持 SG 模式的情况下，申请多个不连续的、相对较小的 DMA 缓冲区通常是防止申请太大的连续物理空间的方法。在 Linux 内核中，使用如下函数映射 SG：

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

nents 是散列表（scatterlist）入口的数量，该函数的返回值是 DMA 缓冲区的数量，可能小于 nents。对于 scatterlist 中的每个项目，dma_map_sg()为设备产生恰当的总线地址，它会合并物理上临近的内存区域。

scatterlist 结构体的定义如代码清单 11.18 所示，它包含了 scatterlist 对应的 page 结构体指针、缓冲区在 page 中的偏移（offset）、缓冲区长度（length）以及总线地址（dma_address）。

代码清单 11.18　scatterlist 结构体

```
1 struct scatterlist
2 {
3   struct page *page;
4   unsigned int offset;
5   dma_addr_t dma_address;
6   unsigned int length;
7 };
```

执行 dma_map_sg()后，通过 sg_dma_address()可返回 scatterlist 对应缓冲区的总线地址，sg_dma_len()可返回 scatterlist 对应缓冲区的长度，这两个函数的原型为：

```
    dma_addr_t sg_dma_address(struct scatterlist *sg);
    unsigned int sg_dma_len(struct scatterlist *sg);
```

在 DMA 传输结束后，可通过 dma_map_sg()的反函数 dma_unmap_sg()去除 DMA 映射：

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

SG 映射属于流式 DMA 映射，与单一缓冲区情况下的流式 DMA 映射类似，如果设备驱动一定要访问映射情况下的 SG 缓冲区，应该先调用如下函数：

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

访问完后，通过下列函数将所有权返回给设备：

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

Linux 系统中可以有一个相对简单的方法预先分配缓冲区，那就是同步“mem=”参数预留内存。例如，对于内存为 64MB 的系统，通过给其传递 mem=62MB 命令行参数可以使得顶部的 2MB 内存被预留出来作为 I/O 内存使用，这 2MB 内存可以被静态映射（11.5

节），也可以被执行 ioremap()。

### 1．申请和释放 DMA 通道

和中断一样，在使用 DMA 之前，设备驱动程序需首先向系统申请 DMA 通道，申请 DMA 通道的函数如下：

```
int request_dma(unsigned int dmanr, const char * device_id);
```

同样的，设备结构体指针可作为传入 device_id 的最佳参数。

使用完 DMA 通道后，应该利用如下函数释放该通道：

```
void free_dma(unsigned int dmanr);
```

现在可以总结出在 Linux 设备驱动中 DMA 相关代码的流程，如图 11.14 所示。

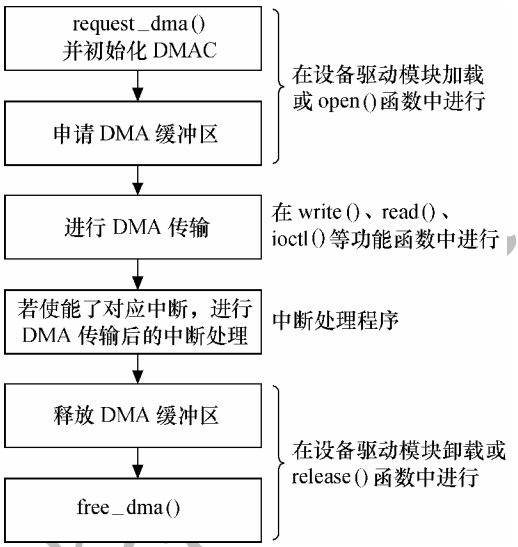


图 11.14　Linux 中 DMA 使用流程

### 2．实例：8237 DMA 控制器

最典型的 DMAC 是 8237，PC 的 DMA 子系统建立在 Intel® 8237 控制器之上。8237 控制器包含了 4 个 DMA 通道，每个通道都可以被独立地编程控制而且任何一个通道在任何时候都可以是活动的。这些通道被编号为 0、1、2 和 3。从 PC/AT 开始，IBM 加入了第二个 8237 芯片，然后把加入的通道命名为 4、5、6 和 7。第 1 个 DMAC 以字节为单位传输，第 2 个 DMAC 以 16 比特数据为单位传输。通过一组寄存器和命令可以控制 8237，代码清单 11.19 给出了 8237 的寄存器及命令定义。

**代码清单 11.19　8237 DMAC 的寄存器和命令定义**

```
1  /* 8237 DMA 控制器 */
2  #define IO_DMA1_BASE 0x00    /* 8 位从 DMA，0～3 通道 */
3  #define IO_DMA2_BASE 0xC0    /* 16 位主 DMA，4～7 通道 */
4
5  /* DMA 控制器 1 控制寄存器 */
6  #define DMA1_CMD_REG     0x08    /* 命令寄存器 (w) */
7  #define DMA1_STAT_REG        0x08    /* 状态寄存器 (r) */
8  #define DMA1_REQ_REG      0x09    /* 请求寄存器 (w) */
```

```
9  #define DMA1_MASK_REG        0x0A    /* 单通道屏蔽 (w) */
10 #define DMA1_MODE_REG        0x0B    /* 模式寄存器 (w) */
11 #define DMA1_CLEAR_FF_REG    0x0C    /* 清除 flip-flop (w) */
12 #define DMA1_TEMP_REG      0x0D    /* 临时寄存器 (r) */
13 #define DMA1_RESET_REG       0x0D    /* 主清除 (w) */
14 #define DMA1_CLR_MASK_REG       0x0E    /* 清除屏蔽 */
15 #define DMA1_MASK_ALL_REG       0x0F    /* 所有通道屏蔽 */
16
17 /* DMA 控制器 2 控制寄存器 */
18 #define DMA2_CMD_REG     0xD0    /* 命令寄存器 (w) */
19 …          /*类似于 DMA 控制器 1*/
20
21 /* DMA0～7 通道地址寄存器 */
22 #define DMA_ADDR_0             0x00
23 …
24
25 /* DMA0～7 通道计数寄存器 */
26 #define DMA_CNT_0              0x01
27 …
28
29 /* DMA0～7 通道页寄存器 */
30 #define DMA_PAGE_0             0x87
31 …
32
33 #define DMA_MODE_READ   0x44    /* I/O 到内存 */
34 #define DMA_MODE_WRITE  0x48    /* 内存到 I/O */
35 …
```

代码清单 11.20 给出的一组函数可以从硬件上设置 8237 DMA 控制器，包括使能和禁止 DMA 通道、设置 DMA 的模式、地址和尺寸等。DMA flip-flop 用来控制对 16 位寄存器（由两个 8 位字节组成）的访问，清除时选中低字节，设置时访问高字节。

代码清单 11.20　8237 DMA 硬件设置

```
1  /*使能 DMA 通道*/
2  static __inline__ void enable_dma(unsigned int dmanr)
3  {
4    if (dmanr <= 3)
5      dma_outb(dmanr, DMA1_MASK_REG);
6    else
7      dma_outb(dmanr &3, DMA2_MASK_REG);
8  }
9
10 /*禁止 DMA 通道*/
11 static __inline__ void disable_dma(unsigned int dmanr)
12 {
13   if (dmanr <= 3)
14     dma_outb(dmanr | 4, DMA1_MASK_REG);
15   else
16     dma_outb((dmanr &3) | 4, DMA2_MASK_REG);
17 }
```

```
    18
    19 /* 设置传输尺寸(通道 0~3 最大 64KB, 通道 5~7 最大 128KB */
    20 static __inline__ void set_dma_count(unsigned int dmanr, unsigned
int count)
    21 {
    22   count--;
    23   if (dmanr <= 3)
    24   {
    25     dma_outb(count &0xff, ((dmanr &3) << 1) + 1+IO_DMA1_BASE);
    26     dma_outb((count >> 8) &0xff, ((dmanr &3) << 1) + 1+IO_DMA1_BASE);
    27   }
    28   else
    29   {
    30     dma_outb((count >> 1) &0xff, ((dmanr &3) << 2) + 2+IO_DMA2_BASE);
    31     dma_outb((count >> 9) &0xff, ((dmanr &3) << 2) + 2+IO_DMA2_BASE);
    32   }
    33 }
    34
    35 /* 设置传输地址和页位 */
    36 static __inline__ void set_dma_addr(unsigned int dmanr, unsigned
int a)
    37 {
    38   set_dma_page(dmanr, a >> 16);
    39   if (dmanr <= 3)
    40   {
    41     dma_outb(a &0xff, ((dmanr &3) << 1) + IO_DMA1_BASE);
    42     dma_outb((a >> 8) &0xff, ((dmanr &3) << 1) + IO_DMA1_BASE);
    43   }
    44   else
    45   {
    46     dma_outb((a >> 1) &0xff, ((dmanr &3) << 2) + IO_DMA2_BASE);
    47     dma_outb((a >> 9) &0xff, ((dmanr &3) << 2) + IO_DMA2_BASE);
    48   }
    49 }
    50
    51 /* 设置页寄存器位，当已知 DMA 当前地址寄存器的低 16 位时，连续传输 */
    52 static __inline__ void set_dma_page(unsigned int dmanr, char pagenr)
    53 {
    54   switch (dmanr)
    55   {
```

```
56    case 0:
57      dma_outb(pagenr, DMA_PAGE_0);
58      break;
59    case 1:
60      dma_outb(pagenr, DMA_PAGE_1);
61      break;
62    case 2:
63      dma_outb(pagenr, DMA_PAGE_2);
64      break;
65    case 3:
66      dma_outb(pagenr, DMA_PAGE_3);
67      break;
68    case 5:
69      dma_outb(pagenr &0xfe, DMA_PAGE_5);
70      break;
71    case 6:
72      dma_outb(pagenr &0xfe, DMA_PAGE_6);
73      break;
74    case 7:
75      dma_outb(pagenr &0xfe, DMA_PAGE_7);
76      break;
77  }
78 }
79
80 /*清除 DMA flip-flop*/
81 static __inline__ void clear_dma_ff(unsigned int dmanr)
82 {
83  if (dmanr <= 3)
84    dma_outb(0, DMA1_CLEAR_FF_REG);
85  else
86    dma_outb(0, DMA2_CLEAR_FF_REG);
87 }
88
89 /*设置某通道的 DMA 模式*/
90 static __inline__ void set_dma_mode(unsigned int dmanr, char mode)
91 {
92  if (dmanr <= 3)
93    dma_outb(mode | dmanr, DMA1_MODE_REG);
94  else
95    dma_outb(mode | (dmanr &3), DMA2_MODE_REG);
```

```
96 }
```

假设设备 xxx 使用了 DMA，DMA 相关的信息应该被添加到设备结构体内。在模块加载函数或打开函数中，DMA 通道和中断应该被申请，而 DMA 本身也应被初始化。不论是内存到 I/O 的 DMA，还是 I/O 到内存的 DMA，都应使用代码清单 11.19 给出的硬件设置函数，不过命令不一样。中断服务程序完成 DMA 的善后处理，有时候，正是中断事件的到来才触发了一次 DMA 的传输。

内存到 I/O 的 DMA 发送通常由上层触发，而 I/O 到内存的 DMA 通常由外设收到了数据之后的中断触发。代码清单 11.21 给出了外设使用 8237 DMA 控制器进行数据传输的设备驱动典型范例。

代码清单 11.21　外设使用 8237 DMA 驱动范例

```
1  /*xxx 设备结构体*/
2  typedef struct
3  {
4   ...
5   void *dma_buffer; //DMA 缓冲区
6   /*当前 DMA 的相关信息*/
7   struct
8   {
9       unsigned int direction; //方向
10      unsigned int length;    //尺寸
11         void *target;          //目标
12      unsigned long start_time;  //开始时间
13  } current_dma;
14
15  unsigned char       dma;//DMA 通道
16  } xxx_device;
17
18  static int xxx_open(...)
19  {
20  ...
21  /*安装中断服务程序 */
22  if ((retval = request_irq(dev->irq, &xxx_interrupt, 0, dev->name,
dev))) {
23      printk(KERN_ERR "%s: could not allocate IRQ%d\n", dev->name,
dev->irq);
24      return retval;
25  }
26  /*申请 DMA*/
27  if ((retval = request_dma(dev->dma, dev->name))) {
```

```
    free_irq(dev->irq, dev);
    printk(KERN_ERR "%s: could not allocate DMA%d channel\n", ...);
    return retval;
}
/*申请 DMA 缓冲区*/
dev->dma_buffer = (void *) dma_mem_alloc(DMA_BUFFER_SIZE);
if (!dev->dma_buffer) {
    printk(KERN_ERR "%s: could not allocate DMA buffer\n",
dev->name);
    free_dma(dev->dma);
    free_irq(dev->irq, dev);
    return -ENOMEM;
}
/*初始化 DMA*/
init_dma();
...
}

/*内存到外设*/
static int mem_to_xxx(const byte *buf,int len)
{
...
dev->current_dma.direction = 1; /*DMA 方向*/
dev->current_dma.start_time = jiffies; /*记录 DMA 开始时间*/

memcpy(dev->dma_buffer, buf, len); /*复制要发送的数据到 DMA 缓冲区*/
target = isa_virt_to_bus(dev->dma_buffer);/*假设 xxx 挂接在 ISA 总线
*/

 /*进行一次 DMA 写操作*/
flags=claim_dma_lock();
disable_dma(dev->dma); /*禁止 DMA*/
clear_dma_ff(dev->dma);  /*清除 DMA flip-flop*/
set_dma_mode(dev->dma, 0x48);  /* DMA 内存 -> io */
set_dma_addr(dev->dma, target); /*设置 DMA 地址*/
set_dma_count(dev->dma, len);  /*设置 DMA 长度*/
outb_control(dev->x_ctrl | DMAE | TCEN, dev);/*让设备接收 DMA*/
enable_dma(dev->dma);    /*使能 DMA*/
release_dma_lock(flags);

```

```
66    printk(KERN_DEBUG "%s: DMA transfer started\n", dev->name);
67   ...
68  }
69
70  /*外设到内存*/
71  static void xxx_to_mem(const char *buf, int len,char * target)
72  {
73    ...
74    /*记录 DMA 信息*/
75    dev->current_dma.target = target;
76    dev->current_dma.direction = 0;
77    dev->current_dma.start_time = jiffies;
78    dev->current_dma.length = len;
79
80    /*进行一次 DMA 读操作*/
81    outb_control(dev->x_ctrl | DIR | TCEN | DMAE, dev);
82    flags = claim_dma_lock();
83    disable_dma(dev->dma);
84    clear_dma_ff(dev->dma);
85    set_dma_mode(dev->dma, 0x04); /* I/O->mem */
86    set_dma_addr(dev->dma, isa_virt_to_bus(target));
87    set_dma_count(dev->dma, len);
88    enable_dma(dev->dma);
89    release_dma_lock(flags);
90    ...
91  }
92
93  /*设备中断处理*/
94   static irqreturn_t xxx_interrupt(int irq, void *dev_id, struct
pt_regs *reg_ptr)
95  {
96    ...
97    do
98    {
99      /* DMA 传输完成？ */
100     if (int_type==DMA_DONE)
101     {
102       outb_control(dev->x_ctrl &~(DMAE | TCEN | DIR), dev);
103       if (dev->current_dma.direction)
104       {
```

```
105         /*内存->I/O*/
106         ...
107       }
108       else
109       /*I/O->内存*/
110       {
111         memcpy(dev->current_dma.target, dev->dma_buffer, dev
112           ->current_dma.length);
113       }
114     }
115     else if(int_type==RECV_DATA) /*收到数据*/
116     {
117         xxx_to_mem(...);/*通过 DMA 读接收到的数据到内存*/
118     }
119     ...
120   }
121   ...
122 }
```

# 11.7 总结

外设可处于 CPU 的内存空间和 I/O 空间，除 X86 外，嵌入式处理器一般只存在内存空间。在 Linux 系统中，为 I/O 内存和 I/O 端口的访问提高了一套统一的方法，访问流程一般为“申请资源→映射→访问→去映射→释放资源”。

对于有 MMU 的处理器而言，Linux 系统的内部布局比较复杂，可直接映射的物理内存称为常规内存，超出部分为高端内存。kmalloc()和__get_free_pages()申请的内存在物理上连续，而 vmalloc()申请的内存在物理上不连续。

DMA 操作可能导致 Cache 的不一致问题，因此，对于 DMA 缓冲，应该使用 dma_alloc_coherent()等方法申请。在 DMA 操作中涉及总线地址、物理地址和虚拟地址等概念，区分这 3 类地址非常重要。Linux 内核中对 DMA 通道的申请和释放采用了和中断类似的方法。

- 课程内容：http://www.embedu.org/courses/course1.htm
- 项目实战：http://www.embedu.org/courses/project.htm
- 出版教材：http://www.embedu.org/courses/course3.htm
- 实验设备：http://www.embedu.org/courses/course5.htm

## 推荐课程：华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：
  http://www.farsight.com.cn/courses/TS-LinuxApp.htm
- 嵌入式 Linux 系统开发班：
  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm
- 嵌入式 Linux 驱动开发班：
  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm