Linux 对 IO 端口资源的管理

本文主要从内核实现的角度分析 linux 2.4.0 内核 IO 子系统中对 IO 端口资源的管理的实现原理。本文是为那些想要深入分析 Linux 的 IO 子系统的读者和设备驱动程序开发人员而写的。

Copyright ?0? 2002 by 詹荣开
E-mail:zhanrk@sohu.com
linux-2.4.0
Version 1.0.0，2002-10-1

关键词：设备管理、驱动程序、I/O 端口、资源

几乎每一种外设都是通过读写设备上的寄存器来进行的。外设寄存器也称为“I/O 端口”，通常包括：控制寄存器、状态寄存器和数据寄存器三大类，而且一个外设的寄存器通常被连续地编址。CPU 对外设 IO 端口物理地址的编址方式有两种：一种是 I/O 映射方式（I/O－mapped），另一种是内存映射方式（Memory－mapped）。而具体采用哪一种则取决于 CPU 的体系结构。

有些体系结构的 CPU（如，PowerPC、m68k 等）通常只实现一个物理地址空间（RAM）。在这种情况下，外设 I/O 端口的物理地址就被映射到 CPU 的单一物理地址空间中，而成为内存的一部分。此时，CPU 可以象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外设 I/O 指令。这就是所谓的“内存映射方式”（Memory－mapped）。

而另外一些体系结构的 CPU（典型地如 X86）则为外设专门实现了一个单独地地址空间，称为“I/O 地址空间”或者“I/O 端口空间”。这是一个与 CPU 地 RAM 物理地址空间不同的地址空间，所有外设的 I/O 端口均在这一空间中进行编址。CPU 通过设立专门的 I/O 指令（如 X86 的 IN 和 OUT 指令）来访问这一空间中的地址单元（也即 I/O 端口）。这就是所谓的“I/O 映射方式”（I/O－mapped）。与 RAM 物理地址空间相比，I/O 地址空间通常都比较小，如 x86 CPU 的 I/O 空间就只有 64KB（0－0xffff）。这是“I/O 映射方式”的一个主要缺点。

linux 将基于 I/O 映射方式的或内存映射方式的 I/O 端口通称为“I/O 区域”（I/O region）。

在讨论对 I/O 区域的管理之前，我们首先来分析一下 Linux 是如何实现“I/O 资源”这一抽象概念的。

3．1 linux 对 I/O 资源的描述

linux 设计了一个通用的数据结构 resource 来描述各种 I/O 资源（如：I/O 端口、外设内存、DMA 和 IRQ 等）。该结构定义在 include/linux/ioport.h 头文件中：

```
struct resource {
const char *name;
unsigned long start, end;
unsigned long flags;
struct resource *parent, *sibling, *child;
};
```

各成员的含义如下：

1. name 指针：指向此资源的名称。
2. start 和 end：表示资源的起始物理地址和终止物理地址。它们确定了资源的范围，也即是一个闭区间[start,end]。
3. flags：描述此资源属性的标志（见下面）。
4. 指针 parent、sibling 和 child：分别为指向父亲、兄弟和子资源的指针。

属性 flags 是一个 unsigned long 类型的 32 位标志值，用以描述资源的属性。比如：资源的类型、是否只读、是否可缓存，以及是否已被占用等。下面是一部分常用属性标志位的定义（ioport.h）：

```
/*
* IO resources have these defined flags.
*/
#define IORESOURCE_BITS 0x000000ff /* Bus-specific bits */

#define IORESOURCE_IO 0x00000100 /* Resource type */
#define IORESOURCE_MEM 0x00000200
#define IORESOURCE_IRQ 0x00000400
#define IORESOURCE_DMA 0x00000800

#define IORESOURCE_PREFETCH 0x00001000 /* No side effects */
#define IORESOURCE_READONLY 0x00002000
#define IORESOURCE_CACHEABLE 0x00004000
#define IORESOURCE_RANGELENGTH 0x00008000
#define IORESOURCE_SHADOWABLE 0x00010000
```

```
#define IORESOURCE_BUS_HAS_VGA 0x00080000

#define IORESOURCE_UNSET 0x20000000
#define IORESOURCE_AUTO 0x40000000
#define IORESOURCE_BUSY 0x80000000
/* Driver has marked this resource busy */
```

指针 parent、sibling 和 child 的设置是为了以一种树的形式来管理各种 I/O 资源。

3．2 linux 对 I/O 资源的管理

linux 是以一种倒置的树形结构来管理每一类 I/O 资源（如：I/O 端口、外设内存、DMA 和 IRQ）的。每一类 I/O 资源都对应有一颗倒置的资源树，树中的每一个节点都是一个 resource 结构，而树的根结点 root 则描述了该类资源的整个资源空间。

基于上述这个思想，linux 在 kernel/Resource.c 文件中实现了对资源的申请、释放及查找等操作。

3．2．1 I/O 资源的申请

假设某类资源有如下这样一颗资源树：

节点 root、r1、r2 和 r3 实际上都是一个 resource 结构类型。子资源 r1、r2 和 r3 通过 sibling 指针链接成一条单向非循环链表，其表头由 root 节点中的 child 指针定义，因此也称为父资源的子资源链表。r1、r2 和 r3 的 parent 指针均指向他们的父资源节点，在这里也就是图中的 root 节点。

假设想在 root 节点中分配一段 I/O 资源（由图中的阴影区域表示），函数 request_resource() 实现这一功能。它有两个参数：①root 指针，表示要在哪个资源根节点中进行分配；②new 指针，指向描述所要分配的资源（即图中的阴影区域）的 resource 结构。该函数的源代码如下（kernel/resource.c）：

```
    int request_resource(struct resource *root, struct resource *new)
    {
struct resource *conflict;

write_lock(&resource_lock);
conflict = ___request_resource(root, new);
write_unlock(&resource_lock);
return conflict ? -EBUSY : 0;
    }
```

对上述函数的 NOTE 如下：

①资源锁 resource_lock 对所有资源树进行读写保护，任何代码段在访问某一颗资源树之前都必须先持有该锁。其定义如下（kernel/Resource.c）：

```
static rwlock_t resource_lock = RW_LOCK_UNLOCKED;
```

②可以看出，函数实际上是通过调用内部静态函数___request_resource()来完成实际的资源分配工作。如果该函数返回非空指针，则表示有资源冲突；否则，返回 NULL 就表示分配成功。

③最后，如果 conflict 指针为 NULL，则 request_resource()函数返回返回值 0，表示成功；否则返回－EBUSY 表示想要分配的资源已被占用。

函数___request_resource()完成实际的资源分配工作。如果参数 new 所描述的资源中的一部分或全部已经被其它节点所占用，则函数返回与 new 相冲突的 resource 结构的指针。否则就返回 NULL。该函数的源代码如下

（kernel/Resource.c）：

```
/* Return the conflict entry if you can't request it */
static struct resource * ___request_resource
    (struct resource *root, struct resource *new)
{
unsigned long start = new->start;
unsigned long end = new->end;
struct resource *tmp, **p;

if (end < start)
return root;
if (start < root->start)
return root;
if (end > root->end)
return root;
p = &root->child;
for (;;) {
tmp = *p;
if (!tmp || tmp->start > end) {
new->sibling = tmp;
*p = new;
new->parent = root;
return NULL;
}
p = &tmp->sibling;
```

```
if (tmp->end < start)
continue;
return tmp;
}
}
```

对函数的 NOTE：

①前三个 if 语句判断 new 所描述的资源范围是否被包含在 root 内，以及是否是一段有效的资源（因为 end 必须大于 start）。否则就返回 root 指针，表示与根结点相冲突。

②接下来用一个 for 循环遍历根节点 root 的 child 链表，以便检查是否有资源冲突，并将 new 插入到 child 链表中的合适位置（child 链表是以 I/O 资源物理地址从低到高的顺序排列的）。为此，它用 tmp 指针指向当前正被扫描的 resource 结构，用指针 p 指向前一个 resource 结构的 sibling 指针成员变量，p 的初始值为指向 root－>sibling。For 循环体的执行步骤如下：

l 让 tmp 指向当前正被扫描的 resource 结构（tmp = *p）。

l 判断 tmp 指针是否为空（tmp 指针为空说明已经遍历完整个 child 链表），或者当前被扫描节点的起始位置 start 是否比 new 的结束位置 end 还要大。只要这两个条件之一成立的话，就说明没有资源冲突，于是就可以把 new 链入 child 链表中：①设置 new 的 sibling 指针指向当前正被扫描的节点 tmp(new->sibling=tmp)；②当前节点 tmp 的前一个兄弟节点的 sibling 指针被修改为指向 new 这个节点（*p=new）；③将 new 的 parent 指针设置为指向 root。然后函数就可以返回了（返回值 NULL 表示没有资源冲突）。

l 如果上述两个条件都不成立，这说明当前被扫描节点的资源域有可能与 new 相冲突（实际上就是两个闭区间有交集），因此需要进一步判断。为此它首先修改指针 p，让它指向 tmp->sibling，以便于继续扫描 child 链表。然后，判断 tmp->end 是否小于 new->start，如果小于，则说明当前节点 tmp 和 new 没有资源冲突，因此执行 continue 语句，继续向下扫描 child 链表。否则，如果 tmp->end 大于或等于 new->start，则说明 tmp->[start,end]和 new->[start,end]之间有交集。所以返回当前节点的指针 tmp，表示发生资源冲突。

3．2．2 资源的释放

函数 release_resource()用于实现 I/O 资源的释放。该函数只有一个参数——即指针 old，它指向所要释放的资源。起源代码如下：

```
int release_resource(struct resource *old)
{
int retval;
```

```
write_lock(&resource_lock);
retval = ___release_resource(old);
write_unlock(&resource_lock);
return retval;
}
```

可以看出，它实际上通过调用___release_resource()这个内部静态函数来完成实际的资源释放工作。函数___release_resource()的主要任务就是将资源区域 old（如果已经存在的话）从其父资源的 child 链表重摘除，它的源代码如下：

```
static int ___release_resource(struct resource *old)
{
struct resource *tmp, **p;

p = &old->parent->child;
for (;;) {
tmp = *p;
if (!tmp)
break;
if (tmp == old) {
*p = tmp->sibling;
old->parent = NULL;
return 0;
}
p = &tmp->sibling;
}
return -EINVAL;
}
```

对上述函数代码的 NOTE 如下：

同函数___request_resource()相类似，该函数也是通过一个 for 循环来遍历父资源的 child 链表。为此，它让 tmp 指针指向当前被扫描的资源，而指针 p 则指向当前节点的前一个节点的 sibling 成员（p 的初始值为指向父资源的 child 指针）。循环体的步骤如下：

①首先，让 tmp 指针指向当前被扫描的节点（tmp = *p）。

②如果 tmp 指针为空，说明已经遍历完整个 child 链表，因此执行 break 语句推出 for 循环。由于在遍历过程中没有在 child 链表中找到参数 old 所指定的资源节点，因此最后返回错误值－EINVAL，表示参数 old 是一个无效的值。

③接下来，判断当前被扫描节点是否就是参数 old 所指定的资源节点。如果是，那就将 old 从 child 链表中去除，也即让当前结点 tmp 的前一个兄弟节点的 sibling 指针指向 tmp 的下一

个节点，然后将 old－>parent 指针设置为 NULL。最后返回 0 值表示执行成功。

④如果当前被扫描节点不是资源 old，那就继续扫描 child 链表中的下一个元素。因此将指针 p 指向 tmp－>sibling 成员。

3．2．3 检查资源是否已被占用，

函数 check_resource()用于实现检查某一段 I/O 资源是否已被占用。其源代码如下：

```
int check_resource(struct resource *root, unsigned long start, unsigned long len)
{
struct resource *conflict, tmp;

tmp.start = start;
tmp.end = start + len - 1;
write_lock(&resource_lock);
conflict = ___request_resource(root, &tmp);
if (!conflict)
___release_resource(&tmp);
write_unlock(&resource_lock);
return conflict ? -EBUSY : 0;
}
```

对该函数的 NOTE 如下：

①构造一个临时资源 tmp，表示所要检查的资源[start,start+end-1]。

②调用___request_resource()函数在根节点 root 申请 tmp 所表示的资源。如果 tmp 所描述的资源还被人使用，则该函数返回 NULL，否则返回非空指针。因此接下来在 conflict 为 NULL 的情况下，调用___release_resource()将刚刚申请的资源释放掉。

③最后根据 conflict 是否为 NULL，返回－EBUSY 或 0 值。

3．2．4 寻找可用资源

函数 find_resource()用于在一颗资源树中寻找未被使用的、且满足给定条件的（也即资源长度大小为 size，且在[min,max]区间内）的资源。其函数源代码如下：

```
/*
* Find empty slot in the resource tree given range and alignment.
*/
static int find_resource(struct resource *root, struct resource *new,
unsigned long size,
```

```
unsigned long min, unsigned long max,
unsigned long align,
void (*alignf)(void *, struct resource *, unsigned long),
void *alignf_data)
{
struct resource *this = root->child;

new->start = root->start;
for(;;) {
if (this)
new->end = this->start;
else
new->end = root->end;
if (new->start < min)
new->start = min;
if (new->end > max)
new->end = max;
new->start = (new->start + align - 1) & ~(align - 1);
if (alignf)
alignf(alignf_data, new, size);
if (new->start < new->end && new->end - new->start + 1 >= size)
{
new->end = new->start + size - 1;
return 0;
}
if (!this)
break;
new->start = this->end + 1;
this = this->sibling;
}
return -EBUSY;
}
```

对该函数的 NOTE 如下：

同样，该函数也要遍历 root 的 child 链表，以寻找未被使用的资源空洞。为此，它让 this 指针表示当前正被扫描的子资源节点，其初始值等于 root->child，即指向 child 链表中的第一个节点，并让 new->start 的初始值等于 root->start，然后用一个 for 循环开始扫描 child 链表，对于每一个被扫描的节点，循环体执行如下操作：

①首先，判断 this 指针是否为 NULL。如果不为空，就让 new->end 等于 this->start，也即让资源 new 表示当前资源节点 this 前面那一段未使用的资源区间。

②如果 this 指针为空，那就让 new->end 等于 root->end。这有两层意思：第一种情况就是根结点的 child 指针为 NULL（即根节点没有任何子资源）。因此此时先暂时将 new->end 放到

最大。第二种情况就是已经遍历完整个 child 链表，所以此时就让 new 表示最后一个子资源后面那一段未使用的资源区间。

③根据参数 min 和 max 修正 new->[start,end]的值，以使资源 new 被包含在[min,max]区域内。

④接下来进行对齐操作。

⑤然后，判断经过上述这些步骤所形成的资源区域 new 是否是一段有效的资源（end 必须大于或等于 start），而且资源区域的长度满足 size 参数的要求（end－start + 1>=size）。如果这两个条件均满足，则说明我们已经找到了一段满足条件的资源空洞。因此在对 new->end 的值进行修正后，然后就可以返回了（返回值 0 表示成功）。

⑥如果上述两条件不能同时满足，则说明还没有找到，因此要继续扫描链表。在继续扫描之前，我们还是要判断一下 this 指针是否为空。如果为空，说明已经扫描完整个 child 链表，因此就可以推出 for 循环了。否则就将 new->start 的值修改为 this->end+1，并让 this 指向下一个兄弟资源节点，从而继续扫描链表中的下一个子资源节点。

3．2．5 分配接口 allocate_resource()

在 find_resource()函数的基础上，函数 allocate_resource()实现：在一颗资源树中分配一条指定大小的、且包含在指定区域[min,max]中的、未使用资源区域。其源代码如下：

```
/*
* Allocate empty slot in the resource tree given range and alignment.
*/
int allocate_resource(struct resource *root, struct resource *new,
unsigned long size,
unsigned long min, unsigned long max,
unsigned long align,
void (*alignf)(void *, struct resource *, unsigned long),
void *alignf_data)
{
int err;

write_lock(&resource_lock);
err = find_resource(root, new, size, min, max, align, alignf, alignf_data);
if (err >= 0 && __request_resource(root, new))
err = -EBUSY;
write_unlock(&resource_lock);
return err;
}
```

3．2．6 获取资源的名称列表

函数 get_resource_list()用于获取根节点 root 的子资源名字列表。该函数主要用来支持/proc/文件系统（比如实现 proc/ioports 文件和/proc/iomem 文件）。其源代码如下：

```
int get_resource_list(struct resource *root, char *buf, int size)
{
char *fmt;
int retval;

fmt = " %08lx-%08lx : %s
";
if (root->end < 0x10000)
fmt = " %04lx-%04lx : %s
";
read_lock(&resource_lock);
retval = do_resource_list(root->child, fmt, 8, buf, buf + size) - buf;
read_unlock(&resource_lock);
return retval;
}
```

可以看出，该函数主要通过调用内部静态函数 do_resource_list()来实现其功能，其源代码如下：

```
/*
* This generates reports for /proc/ioports and /proc/iomem
*/
static char * do_resource_list(struct resource *entry, const char *fmt,
    int offset, char *buf, char *end)
{
if (offset < 0)
offset = 0;

while (entry) {
const char *name = entry->name;
unsigned long from, to;

if ((int) (end-buf) < 80)
return buf;

from = entry->start;
to = entry->end;
if (!name)
name = "";
```

```
buf += sprintf(buf, fmt + offset, from, to, name);
if (entry->child)
buf = do_resource_list(entry->child, fmt, offset-2, buf, end);
entry = entry->sibling;
}

return buf;
}
```

函数 do_resource_list()主要通过一个 while{}循环以及递归嵌套调用来实现，较为简单，这里就不在详细解释了。

3．3 管理 I/O Region 资源

linux 将基于 I/O 映射方式的 I/O 端口和基于内存映射方式的 I/O 端口资源统称为“I/O 区域”（I/O Region）。I/O Region 仍然是一种 I/O 资源，因此它仍然可以用 resource 结构类型来描述。下面我们就来看看 Linux 是如何管理 I/O Region 的。

3．3．1 I/O Region 的分配

在函数___request_resource()的基础上，linux 实现了用于分配 I/O 区域的函数___request_region()，如下：

```
struct resource * ___request_region(struct resource *parent,
    unsigned long start, unsigned long n, const char *name)
{
struct resource *res = kmalloc(sizeof(*res), GFP_KERNEL);

if (res) {
memset(res, 0, sizeof(*res));
res->name = name;
res->start = start;
res->end = start + n - 1;
res->flags = IORESOURCE_BUSY;

write_lock(&resource_lock);

for (;;) {
struct resource *conflict;

conflict = ___request_resource(parent, res);
if (!conflict)
break;
```

```
if (conflict != parent) {
parent = conflict;
if (!(conflict->flags & IORESOURCE_BUSY))
continue;
}

/* Uhhuh, that didn't work out.. */
kfree(res);
res = NULL;
break;
}
write_unlock(&resource_lock);
}
return res;
}
```

NOTE:

①首先，调用 kmalloc（ ）函数在 SLAB 分配器缓存中分配一个 resource 结构。

②然后，相应的根据参数值初始化所分配的 resource 结构。注意！flags 成员被初始化为 IORESOURCE_BUSY。

③接下来，用一个 for 循环开始进行资源分配，循环体的步骤如下：

l 首先，调用___request_resource()函数进行资源分配。如果返回 NULL，说明分配成功，因此就执行 break 语句推出 for 循环，返回所分配的 resource 结构的指针，函数成功地结束。

l 如果___request_resource()函数分配不成功，则进一步判断所返回的冲突资源节点是否就是父资源节点 parent。如果不是，则将分配行为下降一个层次，即试图在当前冲突的资源节点中进行分配（只有在冲突的资源节点没有设置 IORESOURCE_BUSY 的情况下才可以），于是让 parent 指针等于 conflict，并在 conflict->flags&IORESOURCE_BUSY 为 0 的情况下执行 continue 语句继续 for 循环。

l 否则如果相冲突的资源节点就是父节点 parent，或者相冲突资源节点设置了 IORESOURCE_BUSY 标志位，则宣告分配失败。于是调用 kfree( )函数释放所分配的 resource 结构，并将 res 指针置为 NULL，最后用 break 语句推出 for 循环。

④最后，返回所分配的 resource 结构的指针。

3．3．2 I/O Region 的释放

函数___release_region()实现在一个父资源节点 parent 中释放给定范围的 I/O Region。

实际上该函数的实现思想与___release_resource()相类似。其源代码如下：

```
void ___release_region(struct resource *parent,
        unsigned long start, unsigned long n)
{
struct resource **p;
unsigned long end;

p = &parent->child;
end = start + n - 1;

for (;;) {
struct resource *res = *p;

if (!res)
break;
if (res->start <= start && res->end >= end) {
if (!(res->flags & IORESOURCE_BUSY)) {
p = &res->child;
continue;
}
if (res->start != start' 'res->end != end)
break;
*p = res->sibling;
kfree(res);
return;
}
p = &res->sibling;
}
printk("Trying to free nonexistent resource <%08lx-%08lx>
", start, end);
}
```

类似地，该函数也是通过一个 for 循环来遍历父资源 parent 的 child 链表。为此，它让指针 res 指向当前正被扫描的子资源节点，指针 p 指向前一个子资源节点的 sibling 成员变量，p 的初始值为指向 parent->child。For 循环体的步骤如下：

①让 res 指针指向当前被扫描的子资源节点（res = *p）。

②如果 res 指针为 NULL，说明已经扫描完整个 child 链表，所以退出 for 循环。

③如果 res 指针不为 NULL，则继续看看所指定的 I/O 区域范围是否完全包含在当前资源节点中，也即看看[start,start+n-1]是否包含在 res->[start,end]中。如果不属于，则让 p 指向当前资源节点的 sibling 成员，然后继续 for 循环。如果属于，则执行下列步骤：

l 先看看当前资源节点是否设置了 IORESOURCE_BUSY 标志位。如果没有设置该标志位，则说明该资源节点下面可能还会有子节点，因此将扫描过程下降一个层次，于是修改 p 指针，使它指向 res->child，然后执行 continue 语句继续 for 循环。

l 如果设置了 IORESOURCE_BUSY 标志位。则一定要确保当前资源节点就是所指定的 I/O 区域，然后将当前资源节点从其父资源的 child 链表中去除。这可以通过让前一个兄弟资源节点的 sibling 指针指向当前资源节点的下一个兄弟资源节点来实现（即让*p=res->sibling），最后调用 kfree（）函数释放当前资源节点的 resource 结构。然后函数就可以成功返回了。

3．3．3 检查指定的 I/O Region 是否已被占用

函数___check_region()检查指定的 I/O Region 是否已被占用。其源代码如下：

```
int ___check_region(struct resource *parent, unsigned long start, unsigned long n)
{
struct resource * res;

res = ___request_region(parent, start, n, "check-region");
if (!res)
return -EBUSY;

release_resource(res);
kfree(res);
return 0;
}
```

该函数的实现与___check_resource()的实现思想类似。首先，它通过调用___request_region()函数试图在父资源 parent 中分配指定的 I/O Region。如果分配不成功，将返回 NULL，因此此时函数返回错误值－EBUSY 表示所指定的 I/O Region 已被占用。如果 res 指针不为空则说明所指定的 I/O Region 没有被占用。于是调用___release_resource()函数将刚刚分配的资源释放掉（实际上是将 res 结构从 parent 的 child 链表去除），然后调用 kfree（）函数释放 res 结构所占用的内存。最后，返回 0 值表示指定的 I/O Region 没有被占用。

3．4 管理 I/O 端口资源

我们都知道，采用 I/O 映射方式的 X86 处理器为外设实现了一个单独的地址空间，也即“I/O 空间”（I/O Space）或称为“I/O 端口空间”，其大小是 64KB（0x0000－0xffff）。linux 在其所支持的所有平台上都实现了“I/O 端口空间”这一概念。

由于 I/O 空间非常小，因此即使外设总线有一个单独的 I/O 端口空间，却也不是所有的外设都将其 I/O 端口（指寄存器）映射到“I/O 端口空间”中。比如，大多数 PCI 卡都通过内存映

射方式来将其 I/O 端口或外设内存映射到 CPU 的 RAM 物理地址空间中。而老式的 ISA 卡通常将其 I/O 端口映射到 I/O 端口空间中。

linux 是基于“I/O Region”这一概念来实现对 I/O 端口资源（I/O－mapped 或 Memory－mapped）的管理的。

3．4．1 资源根节点的定义

linux 在 kernel/Resource.c 文件中定义了全局变量 ioport_resource 和 iomem_resource，来分别描述基于 I/O 映射方式的整个 I/O 端口空间和基于内存映射方式的 I/O 内存资源空间（包括 I/O 端口和外设内存）。其定义如下：

```
struct resource ioport_resource =
        { "PCI IO", 0x0000, IO_SPACE_LIMIT, IORESOURCE_IO };
struct resource iomem_resource =
        { "PCI mem", 0x00000000, 0xffffffff, IORESOURCE_MEM };
```

其中，宏 IO_SPACE_LIMIT 表示整个 I/O 空间的大小，对于 X86 平台而言，它是 0xffff（定义在 include/asm-i386/io.h 头文件中）。显然，I/O 内存空间的大小是 4GB。

3．4．2 对 I/O 端口空间的操作

基于 I/O Region 的操作函数___XXX_region()，linux 在头文件 include/linux/ioport.h 中定义了三个对 I/O 端口空间进行操作的宏：①request_region()宏，请求在 I/O 端口空间中分配指定范围的 I/O 端口资源。②check_region()宏，检查 I/O 端口空间中的指定 I/O 端口资源是否已被占用。③release_region()宏，释放 I/O 端口空间中的指定 I/O 端口资源。这三个宏的定义如下：

```
#define request_region(start,n,name)
___request_region(&ioport_resource, (start), (n), (name))
#define check_region(start,n)
___check_region(&ioport_resource, (start), (n))
#define release_region(start,n)
___release_region(&ioport_resource, (start), (n))
```

其中，宏参数 start 指定 I/O 端口资源的起始物理地址（是 I/O 端口空间中的物理地址），宏参数 n 指定 I/O 端口资源的大小。

3．4．3 对 I/O 内存资源的操作

基于 I/O Region 的操作函数___XXX__region()，linux 在头文件 include/linux/ioport.h 中定义了三个对 I/O 内存资源进行操作的宏：①request__mem__region()宏，请求分配指定的 I/O 内存资源。②check__ mem__region()宏，检查指定的 I/O 内存资源是否已被占用。③release__ mem__region()宏，释放指定的 I/O 内存资源。这三个宏的定义如下：

```
#define request__mem__region(start,n,name)
    ___request__region(&iomem__resource, (start), (n), (name))
#define check__mem__region(start,n)
___check__region(&iomem__resource, (start), (n))
#define release__mem__region(start,n)
___release__region(&iomem__resource, (start), (n))
```

其中，参数 start 是 I/O 内存资源的起始物理地址（是 CPU 的 RAM 物理地址空间中的物理地址），参数 n 指定 I/O 内存资源的大小。

3．4．4 对/proc/ioports 和/proc/iomem 的支持

linux 在 ioport.h 头文件中定义了两个宏：

get__ioport__list()和 get__iomem__list()，分别用来实现/proc/ioports 文件和/proc/iomem 文件。其定义如下：

```
#define get__ioport__list(buf) get__resource__list(&ioport__resource, buf, PAGE__SIZE)
#define get__mem__list(buf) get__resource__list(&iomem__resource, buf, PAGE__SIZE)
```

3．5 访问 I/O 端口空间

在驱动程序请求了 I/O 端口空间中的端口资源后，它就可以通过 CPU 的 IO 指定来读写这些 I/O 端口了。在读写 I/O 端口时要注意的一点就是，大多数平台都区分 8 位、16 位和 32 位的端口，也即要注意 I/O 端口的宽度。

linux 在 include/asm/io.h 头文件（对于 i386 平台就是 include/asm-i386/io.h）中定义了一系列读写不同宽度 I/O 端口的宏函数。如下所示：

⑴读写 8 位宽的 I/O 端口

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

其中，port 参数指定 I/O 端口空间中的端口地址。在大多数平台上（如 x86）它都是 unsigned

short 类型的，其它的一些平台上则是 unsigned int 类型的。显然，端口地址的类型是由 I/O 端口空间的大小来决定的。

⑵读写 16 位宽的 I/O 端口

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

⑶读写 32 位宽的 I/O 端口

```
unsigned int inl ( unsigned port ) ;
void outl ( unsigned int value, unsigned port ) ;
```

3．5．1 对 I/O 端口的字符串操作

除了上述这些“单发”(single－shot) 的 I/O 操作外，某些 CPU 也支持对某个 I/O 端口进行连续的读写操作，也即对单个 I/O 端口读或写一系列字节、字或 32 位整数，这就是所谓的“字符串 I/O 指令”(String Instruction) 。这种指令在速度上显然要比用循环来实现同样的功能要快得多。

linux 同样在 io.h 文件中定义了字符串 I/O 读写函数：

⑴8 位宽的字符串 I/O 操作

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

⑵16 位宽的字符串 I/O 操作

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

⑶32 位宽的字符串 I/O 操作

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

3．5．2 Pausing I/O

在一些平台上（典型地如 X86），对于老式总线（如 ISA）上的慢速外设来说，如果 CPU 读写其 I/O 端口的速度太快，那就可能会发生丢失数据的现象。对于这个问题的解决方法就是在两次连续的 I/O 操作之间插入一段微小的时延，以便等待慢速外设。这就是所谓的“Pausing I/O”。

对于 Pausing I/O，linux 也在 io.h 头文件中定义了它的 I/O 读写函数，而且都以 XXX_p 命名，比如：inb_p()、outb_p()等等。下面我们就以 out_p()为例进行分析。

将 io.h 中的宏定义___OUT(b,”b”char)展开后可得如下定义：

```
extern inline void outb(unsigned char value, unsigned short port) {
___asm___ ___volatile___ ("outb %" "b " "0,%" "w" "1"
: : "a" (value), "Nd" (port));
}

extern inline void outb_p(unsigned char value, unsigned short port) {
___asm___ ___volatile___ ("outb %" "b " "0,%" "w" "1"
___FULL_SLOW_DOWN_IO
: : "a" (value), "Nd" (port));
}
```

可以看出，outb_p()函数的实现中被插入了宏___FULL_SLOWN_DOWN_IO，以实现微小的延时。宏___FULL_SLOWN_DOWN_IO 在头文件 io.h 中一开始就被定义：

```
#ifdef SLOW_IO_BY_JUMPING
#define ___SLOW_DOWN_IO "
jmp 1f
1: jmp 1f
1:"
#else
#define ___SLOW_DOWN_IO "
outb %%al,$0x80"
#endif

#ifdef REALLY_SLOW_IO
#define ___FULL_SLOW_DOWN_IO ___SLOW_DOWN_IO
    ___SLOW_DOWN_IO ___SLOW_DOWN_IO ___SLOW_DOWN_IO
#else
#define ___FULL_SLOW_DOWN_IO ___SLOW_DOWN_IO
#endif
```

显然，___FULL_SLOW_DOWN_IO 就是一个或四个___SLOW_DOWN_IO（根据是否定义了宏 REALLY_SLOW_IO 来决定），而宏___SLOW_DOWN_IO 则被定义成毫无意义的跳转语句或写端口 0x80 的操作（根据是否定义了宏 SLOW_IO_BY_JUMPING 来决定）。

3．6 访问 I/O 内存资源

尽管 I/O 端口空间曾一度在 x86 平台上被广泛使用，但是由于它非常小，因此大多数现代总线的设备都以内存映射方式（Memory－mapped）来映射它的 I/O 端口（指 I/O 寄存器）和外设内存。基于内存映射方式的 I/O 端口（指 I/O 寄存器）和外设内存可以通称为“I/O 内存”资源（I/O Memory）。因为这两者在硬件实现上的差异对于软件来说是完全透明的，所以驱动程序开发人员可以将内存映射方式的 I/O 端口和外设内存统一看作是“I/O 内存”资源。

从前几节的阐述我们知道，I/O 内存资源是在 CPU 的单一内存物理地址空间内进行编址的，也即它和系统 RAM 同处在一个物理地址空间内。因此通过 CPU 的访内指令就可以访问 I/O 内存资源。

一般来说，在系统运行时，外设的 I/O 内存资源的物理地址是已知的，这可以通过系统固件（如 BIOS）在启动时分配得到，或者通过设备的硬连线（hardwired）得到。比如，PCI 卡的 I/O 内存资源的物理地址就是在系统启动时由 PCI BIOS 分配并写到 PCI 卡的配置空间中的 BAR 中的。而 ISA 卡的 I/O 内存资源的物理地址则是通过设备硬连线映射到 640KB－1MB 范围之内的。但是 CPU 通常并没有为这些已知的外设 I/O 内存资源的物理地址预定义虚拟地址范围，因为它们是在系统启动后才已知的（某种意义上讲是动态的），所以驱动程序并不能直接通过物理地址访问 I/O 内存资源，而必须将它们映射到核心虚地址空间内（通过页表），然后才能根据映射所得到的核心虚地址范围，通过访内指令访问这些 I/O 内存资源。

3．6．1 映射 I/O 内存资源

linux 在 io.h 头文件中声明了函数 ioremap（），用来将 I/O 内存资源的物理地址映射到核心虚地址空间（3GB－4GB）中，如下：

```
void * ioremap(unsigned long phys_addr, unsigned long size, unsigned long flags);
void iounmap(void * addr);
```

函数用于取消 ioremap（）所做的映射，参数 addr 是指向核心虚地址的指针。这两个函数都是实现在 mm/ioremap.c 文件中。具体实现可参考《情景分析》一书。

3．6．2 读写 I/O 内存资源

在将 I/O 内存资源的物理地址映射成核心虚地址后，理论上讲我们就可以象读写 RAM 那样直接读写 I/O 内存资源了。但是，由于在某些平台上，对 I/O 内存和系统内存有不同的访问

处理，因此为了确保跨平台的兼容性，linux 实现了一系列读写 I/O 内存资源的函数，这些函数在不同的平台上有不同的实现。但在 x86 平台上，读写 I/O 内存与读写 RAM 无任何差别。如下所示（include/asm-i386/io.h）：

```
#define readb(addr) (*(volatile unsigned char *) __io_virt(addr))
#define readw(addr) (*(volatile unsigned short *) __io_virt(addr))
#define readl(addr) (*(volatile unsigned int *) __io_virt(addr))

#define writeb(b,addr) (*(volatile unsigned char *) __io_virt(addr) = (b))
#define writew(b,addr) (*(volatile unsigned short *) __io_virt(addr) = (b))
#define writel(b,addr) (*(volatile unsigned int *) __io_virt(addr) = (b))

#define memset_io(a,b,c) memset(__io_virt(a),(b),(c))
#define memcpy_fromio(a,b,c) memcpy((a),__io_virt(b),(c))
#define memcpy_toio(a,b,c) memcpy(__io_virt(a),(b),(c))
```

上述定义中的宏__io_virt()仅仅检查虚地址 addr 是否是核心空间中的虚地址。该宏在内核 2.4.0 中的实现是临时性的。具体的实现函数在 arch/i386/lib/Iodebug.c 文件。

显然，在 x86 平台上访问 I/O 内存资源与访问系统主存 RAM 是无差别的。但是为了保证驱动程序的跨平台的可移植性，我们应该使用上面的函数来访问 I/O 内存资源，而不应该通过指向核心虚地址的指针来访问。