Tekkaman Ninja

tekkamanninja.blog.chinaunix.net

tekkamanninj

博客访问： 75918
博文数量： 263
博客积分： 15936
博客等级： 上将
技术积分： 13951
用 户 组： 普通用户
注册时间： 2007-03-27 11:22

加关注 短消息
论坛 加好友

个人简介

Fedora-ARM

文章分类

文章存档

Linux设备驱动程序学习（9）-与硬件通信

2007-11-14 21:50:33

分类： LINUX

Linux设备驱动程序学习（9）-与硬件通信

在学习有关I/O总线的内容时，最好先看看相关的知识：从PC总线到ARM的内部总线

I/O 端口和 I/O 内存

每种外设都是通过读写寄存器来进行控制。

在硬件层，内存区和 I/O 区域没有概念上的区别：它们都是通过向在地址总线和控制总线发出电平信号来进行访问，再通过数据总线读写数据。

因为外设要与I\O总线匹配，而大部分流行的 I/O 总线是基于个人计算机模型（主要是 x86 家族：它为读和写 I/O 端口提供了独立的线路和特殊的 CPU 指令），所以即便那些没有单独I/O 端口地址空间的处理器，在访问外设时也要模拟成读写I\O端口。这一功能通常由外围芯片组（PC 中的南北桥）或 CPU 中的附加电路实现（嵌入式中的方法） 。

Linux 在所有的计算机平台上实现了 I/O 端口。但不是所有的设备都将寄存器映射到 I/O 端口。虽然 ISA设备普遍使用 I/O 端口，但大部分 PCI 设备则把寄存器映射到某个内存地址区，这种 I/O 内存方法通常是首选的。因为它无需使用特殊的处理器指令，CPU 核访问内存更有效率，且编译器在访问内存时在寄存器分配和寻址模式的选择上有更多自由。

I/O 寄存器和常规内存

在进入这部分学习的时候，首先要理解一个概念：side effect，书中译为边际效应，第二版译为副作用。我觉得不管它是怎么被翻译的，都不可能精准表达原作者的意思，所以我个人认为记住side effect就好。下面来讲讲side effect的含义。我先贴出两个网上已有的两种说法（在这里谢谢两位高人的分享）：

> 第一种说法：
>
> 3. side effect（译为边际效应或副作用）：是指读取某个地址时可能导致该地址内容发生变化，比如，有些设备的中断状态寄存器只要一读取，便自动清零。I/O寄存器的操作具有side effect，因此，不能对其操作不能使用cpu缓存。
>
> 原文网址：http://qinbh.blog.sohu.com/62733495.html
>
> 第二种说法：
>
> 说一下我的理解：I/O端口与实际外部设备相关联，通过访问I/O端口控制外部设备，“边际效应”是指控制设备（读取或写入）生效，访问I/O口的主要目的就是边际效应，不像访问普通的内存，只是在一个位置存储或读取一个数值，没有别的含义了。我是基于ARM平台理解的，在《linux设备驱动程序》第二版中的说法是“副作用”，不是“边际效应”。
>
> 原文网址：http://linux.chinaunix.net/bbs/viewthread.php?tid=890636&page=1#pid6312646

结合以上两种说法和自己看《Linux设备驱动程序（第3版）》的理解，我个人认为可以这样解释：

> side effect 是指：访问I/O寄存器时，不仅仅会像访问普通内存一样影响存储单元的值，更重要的是它可能改变CPU的I/O端口电平、输出时序或CPU对I/O端口电平的反应等等，从而实现CPU的控制功能。CPU在电路中的意义就是实现其side effect 。

I/O 寄存器和 RAM 的主要不同就是 I/O 寄存器操作有side effect，而内存操作没有。

因为存储单元的访问速度对 CPU 性能至关重要，编译器会对源代码进行优化，主要是：使用高速缓存保

存数值 和 重新编排读/写指令顺序。但对I/O 寄存器操作来说，这些优化可能造成致命错误。因此，驱动程序必须确保在操作I/O 寄存器时，不使用高速缓存，且不能重新编排读/写指令顺序。

解决方法：

硬件缓存问题:只要把底层硬件配置(自动地或者通过 Linux 初始化代码)成当访问 I/O 区域时(不管内存还是端口)禁止硬件缓存即可。

硬件指令重新排序问题：在硬件(或其他处理器)必须以一个特定顺序执行的操作之间设置内存屏障（memory barrier）。

Linux 提供以下宏来解决所有可能的排序问题:

```
#include <linux/kernel.h>
void barrier(void) /*告知编译器插入一个内存屏障但是对硬件没有影响。编译后的代码会将当前CPU 寄存器中所有修改过的数值保存到内存中，并当需要时重新读取它们。可阻止在屏障前后的编译器优化，但硬件能完成自己的重新排序。其实<linux/kernel.h> 中并没有这个函数，因为它是在kernel.h包含的头文件compiler.h中定义的*/
#include <linux/compiler.h>
# define barrier() __memory_barrier()

#include <asm/system.h>
void rmb(void); /*保证任何出现于屏障前的读在执行任何后续的读之前完成*/
void wmb(void); /*保证任何出现于屏障前的写在执行任何后续的写之前完成*/
void mb(void); /*保证任何出现于屏障前的读写操作在执行任何后续的读写操作之前完成*/
void read_barrier_depends(void); /*一种特殊的、弱些的读屏障形式。rmb 阻止屏障前后的所有读指令的重新排序，read_barrier_depends 只阻止依赖于其他读指令返回的数据的读指令的重新排序。区别微小，且不在所有体系中存在。除非你确切地理解它们的差别，并确信完整的读屏障会增加系统开销，否则应当始终使用 rmb。*/
/*以上指令是barrier的超集*/


void smp_rmb(void);
void smp_read_barrier_depends(void);
void smp_wmb(void);
void smp_mb(void);
/*仅当内核为 SMP 系统编译时插入硬件屏障；否则，它们都扩展为一个简单的屏障调用。*/
```

典型的应用：

```
writel(dev->registers.addr, io_destination_address);
writel(dev->registers.size, io_size);
writel(dev->registers.operation, DEV_READ);
wmb();/*类似一条分界线，上面的写操作必然会在下面的写操作前完成，但是上面的三个写操作的排序无法保证*/
writel(dev->registers.control, DEV_GO);
```

内存屏障影响性能,所以应当只在确实需要它们的地方使用。不同的类型对性能的影响也不同，因此要尽可能地使用需要的特定类型。值得注意的是大部分处理同步的内核原语，例如自旋锁和atomic_t，也可作为内存屏障使用。

某些体系允许赋值和内存屏障组合，以提高效率。它们定义如下:

```
#define set_mb(var, value) do {var = value; mb();} while 0
/*以下宏定义在ARM体系中不存在*/
#define set_wmb(var, value) do {var = value; wmb();} while 0
#define set_rmb(var, value) do {var = value; rmb();} while 0
```

使用do...while 结构来构造宏是标准 C 的惯用方法，它保证了扩展后的宏可在所有上下文环境中被作为一个正常的 C 语句执行。

使用 I/O 端口

I/O 端口是驱动用来和许多设备之间的通讯方式。

I/O 端口分配

在尚未取得端口的独占访问前，不应对端口进行操作。内核提供了一个注册用的接口，允许驱动程序声明它需要的端口：

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n, const char
*name);/*告诉内核：要使用从 first 开始的 n 个端口,name 参数为设备名。若分配成功返回非
NULL，否则将无法使用需要的端口。*/
/*所有的的端口分配显示在 /proc/ioports 中。若不能分配到需要的端口，则可以到这里看看谁
先用了。*/

/*当用完 I/O 端口集(可能在模块卸载时)，应当将它们返回给系统*/
void release_region(unsigned long start, unsigned long n);

int check_region(unsigned long first, unsigned long n);
/*检查一个给定的 I/O 端口集是否可用,若不可用，返回值是一个负错误码。不推荐使用*/
```

操作 I/O 端口

在驱动程序注册I/O 端口后，就可以读/写这些端口。大部分硬件会把8、16和32位端口区分开，不能像访问系统内存那样混淆使用。驱动必须调用不同的函数来存取不同大小的端口。

只支持内存映射的 I/O 寄存器的计算机体系通过重新映射I/O端口到内存地址来伪装端口I/O。为了提高移植性，内核向驱动隐藏了这些细节。Linux 内核头文件(体系依赖的头文件 <asm/io.h>) 定义了下列内联函数(有的体系是宏，有的不存在)来访问 I/O 端口：

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
/*读/写字节端口( 8 位宽 )。port 参数某些平台定义为 unsigned long ，有些为 unsigned
short 。 inb 的返回类型也体系而不同。*/

unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
/*访问 16位 端口( 一个字宽 )*/

unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
/*访问 32位 端口。 longword 声明有的平台为 unsigned long ，有的为 unsigned int。*/
```

在用户空间访问 I/O 端口

以上函数主要提供给设备驱动使用，但它们也可在用户空间使用，至少在 PC上可以。 GNU C 库在 <sys/io.h> 中定义了它们。如果在用户空间代码中使用必须满足以下条件：

（1）程序必须使用 -O 选项编译来强制扩展内联函数。

（2）必须用ioperm 和 iopl 系统调用(#include <sys/perm.h>) 来获得对端口 I/O 操作的权限。ioperm 为获取单独端口操作权限，而 iopl 为整个 I/O 空间的操作权限。 （x86 特有的）

（3）程序以 root 来调用 ioperm 和 iopl，或是其父进程必须以 root 获得端口操作权限。（x86 特有的）

若平台没有 ioperm 和 iopl 系统调用，用户空间可以仍然通过使用 /dev/prot 设备文件访问 I/O 端口。注意：这个文件的定义是体系相关的，并且I/O 端口必须先被注册。

串操作

除了一次传输一个数据的I/O操作，一些处理器实现了一次传输一个数据序列的特殊指令，序列中的数据单位可以是字节、字或双字，这是所谓的串操作指令。它们完成任务比一个 C 语言循环更快。下列宏定义实现了串I/O,它们有的通过单个机器指令实现；但如果目标处理器没有进行串 I/O 的指令，则通过执行一个紧凑的循环实现。 有的体系的原型如下：

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

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

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

使用时注意：它们直接将字节流从端口中读取或写入。当端口和主机系统有不同的字节序时，会导致不可

预期的结果。 使用 inw 读取端口应在必要时自行转换字节序，以匹配主机字节序。

暂停式 I/O

为了匹配低速外设的速度，有时若 I/O 指令后面还紧跟着另一个类似的I/O指令，就必须在 I/O 指令后面插入一个小延时。在这种情况下，可以使用暂停式的I/O函数代替通常的I/O函数，它们的名字以 _p 结尾，如 inb_p、outb_p等等。 这些函数定义被大部分体系支持，尽管它们常常被扩展为与非暂停式I/O同样的代码。因为如果体系使用一个合理的现代外设总线，就没有必要额外暂停。细节可参考平台的 asm 子目录的 io.h 文件。以下是include\asm-arm\io.h中的宏定义：

```
#define outb_p(val,port)     outb((val),(port))
#define outw_p(val,port)     outw((val),(port))
#define outl_p(val,port)     outl((val),(port))
#define inb_p(port)          inb((port))
#define inw_p(port)          inw((port))
#define inl_p(port)          inl((port))

#define outsb_p(port,from,len)     outsb(port,from,len)
#define outsw_p(port,from,len)     outsw(port,from,len)
#define outsl_p(port,from,len)     outsl(port,from,len)
#define insb_p(port,to,len)     insb(port,to,len)
#define insw_p(port,to,len)     insw(port,to,len)
#define insl_p(port,to,len)     insl(port,to,len)
```

由此可见，由于ARM使用内部总线，就没有必要额外暂停，所以暂停式的I/O函数被扩展为与非暂停式I/O同样的代码。

平台相关性

由于自身的特性，I/O 指令与处理器密切相关的，非常难以隐藏系统间的不同。所以大部分的关于端口 I/O 的源码是平台依赖的。以下是x86和ARM所使用函数的总结：

IA-32 (x86)

x86_64

这个体系支持所有的以上描述的函数，端口号是 unsigned short 类型。

ARM

端口映射到内存，支持所有函数。串操作 用C语言实现。端口是 unsigned int 类型。

使用 I/O 内存

除了 x86上普遍使用的I/O 端口外，和设备通讯另一种主要机制是通过使用映射到内存的寄存器或设备内存，统称为 I/O 内存。因为寄存器和内存之间的区别对软件是透明的。I/O 内存仅仅是类似 RAM 的一个区域，处理器通过总线访问这个区域，以实现设备的访问。

根据平台和总线的不同，I/O 内存可以就是否通过页表访问分类。若通过页表访问，内核必须首先安排物理地址使其对设备驱动程序可见，在进行任何 I/O 之前必须调用 ioremap。若不通过页表，I/O 内存区域就类似I/O 端口，可以使用适当形式的函数访问它们。因为“side effect”的影响，不管是否需要 ioremap ，都不鼓励直接使用 I/O 内存的指针。而使用专用的 I/O 内存操作函数，不仅在所有平台上是安全，而且对直接使用指针操作 I/O 内存的情况进行了优化。

I/O 内存分配和映射

I/O 内存区域使用前必须先分配，函数接口在 <linux/ioport.h> 定义：

```
struct resource *request_mem_region(unsigned long start, unsigned long len, char
*name);/* 从 start 开始，分配一个 len 字节的内存区域。成功返回一个非NULL指针，否则返回
NULL。所有的 I/O 内存分配情况都 /proc/iomem 中列出。*/

/*I/O内存区域在不再需要时应当释放*/
void release_mem_region(unsigned long start, unsigned long len);

/*一个旧的检查 I/O 内存区可用性的函数，不推荐使用*/
int check_mem_region(unsigned long start, unsigned long len);
```

然后必须设置一个映射，由 ioremap 函数实现，此函数专门用来为I/O 内存区域分配虚拟地址。经过 ioremap 之后，设备驱动即可访问任意的 I/O 内存地址。注意：ioremap 返回的地址不应当直接引用；应使用内核提供的 accessor 函数。以下为函数定义：

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

```
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);/*如果控制寄存器也在该区域，应使用的非缓存版本，以实现side effect。*/
void iounmap(void * addr);
```

访问I/O 内存

访问I/O 内存的正确方式是通过一系列专用于此目的的函数(在 <asm/io.h> 中定义的)：

```
/*I/O 内存读函数*/
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
/*addr 是从 ioremap 获得的地址(可能包含一个整型偏移量)， 返回值是从给定 I/O 内存读取的值*/

/*对应的I/O 内存写函数*/
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);

/*读和写一系列值到一个给定的 I/O 内存地址，从给定的 buf 读或写 count 个值到给定的 addr*/
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);
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);

/*需要操作一块 I/O 地址，使用一下函数*/
void memset_io(void *addr, u8 value, unsigned int count);
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);

/*旧函数接口，仍可工作，但不推荐。*/
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

像 I/O 内存一样使用端口

一些硬件有一个有趣的特性：一些版本使用 I/O 端口，而其他的使用 I/O 内存。为了统一编程接口，使驱动程序易于编写，2.6 内核提供了一个ioport_map函数:

```
void *ioport_map(unsigned long port, unsigned int count);/*重映射 count 个I/O 端口，使其看起来像 I/O 内存。，此后，驱动程序可以在返回的地址上使用 ioread8 和同类函数。其在编程时消除了I/O 端口和I/O 内存的区别。

/*这个映射应当在它不再被使用时撤销:*/
void ioport_unmap(void *addr);

/*注意：I/O 端口仍然必须在重映射前使用 request_region 分配I/O 端口。ARM9不支持这两个函数！*/
```

上面是基于《Linux设备驱动程序（第3版）》的介绍，以下分析 ARM9的s3c2440A的linux驱动接口。

ARM9的linux驱动接口

s3c24x0处理器是使用I/O内存的，也就是说：他们的外设接口是通过读写相应的寄存器实现的，这些寄存器和内存是使用单一的地址空间，并使用和读写内存一样的指令。所以推荐使用I/O内存的相关指令。

但这并不表示I/O端口的指令在s3c24x0中不可用。但是只要你注意其源码，你就会发现：其实I/O端口的

指令只是一个外壳，内部还是使用和I/O内存一样的代码。以下列出一些：

I/O端口

```
#define outb(v,p)        __raw_writeb(v,__io(p))
#define outw(v,p)        __raw_writew((__force __u16) \
                    cpu_to_le16(v),__io(p))
#define outl(v,p)        __raw_writel((__force __u32) \
                    cpu_to_le32(v),__io(p))

#define inb(p)    ({ __u8 __v = __raw_readb(__io(p)); __v; })
#define inw(p)    ({ __u16 __v = le16_to_cpu((__force __le16) \
           __raw_readw(__io(p))); __v; })
#define inl(p)    ({ __u32 __v = le32_to_cpu((__force __le32) \
           __raw_readl(__io(p))); __v; })
```

I/O内存

```
#define ioread8(p)    ({ unsigned int __v = __raw_readb(p); __v; })
#define ioread16(p)    ({ unsigned int __v = le16_to_cpu(__raw_readw(p)); __v; })
#define ioread32(p)    ({ unsigned int __v = le32_to_cpu(__raw_readl(p)); __v; })

#define iowrite8(v,p)    __raw_writeb(v, p)
#define iowrite16(v,p)    __raw_writew(cpu_to_le16(v), p)
#define iowrite32(v,p)    __raw_writel(cpu_to_le32(v), p)
```

我对I/O端口的指令和I/O内存的指令都写了相应的驱动程序，都通过了测试。在这里值得注意的有4点：

（1）所有的读写指令所赋的地址必须都是虚拟地址，你有两种选择：使用内核已经定义好的地址，如S3C2440_GPJCON等等，这些都是内核定义好的虚拟地址，有兴趣的可以看源码。还有一种方法就是使用自己用ioremap映射的虚拟地址。绝对不能使用实际的物理地址，否则会因为内核无法处理地址而出现oops。

（2）在使用I/O指令时，可以不使用request_region和request_mem_region，而直接使用outb、ioread等指令。因为request的功能只是告诉内核端口被谁占用了，如再次request，内核会制止。

（3）在使用I/O指令时，所赋的地址数据有时必须通过强制类型转换为 unsigned long ，不然会有警告（具体原因请看Linux设备驱动程序学习（7）-内核的数据类型） 。虽然你的程序可能也可以使用，但是最好还是不要有警告为妙。

（4）在include\asm-arm\arch-s3c2410\hardware.h中定义了很多io口的操作函数，有需要可以在驱动中直接使用，很方便。

实验源码：

IO_port.tar.gz

IO_port_test.tar.gz

IO_mem.tar.gz

IO_mem_test.tar.gz

两个模块都实现了阻塞型独享设备的访问控制，并通知内核不支持llseek。具体的测试在IO_port中。

测试现象如下：

```
[Tekkaman2440@SBC2440V4]#cd /lib/modules/
[Tekkaman2440@SBC2440V4]#insmod IO_port.ko
[Tekkaman2440@SBC2440V4]#insmod IO_mem.ko
[Tekkaman2440@SBC2440V4]#cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
```

```
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 14 sound
 81 video4linux
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
204 s3c2410_serial
251 IO_mem
252 IO_port
253 usb_endpoint
254 rtc

Block devices:
  1 ramdisk
256 rfd
  7 loop
 31 mtdblock
 93 nftl
 96 inftl
179 mmc
[Tekkaman2440@SBC2440V4]#mknod -m 666 /dev/IO_port c 252 0
[Tekkaman2440@SBC2440V4]#mknod -m 666 /dev/IO_mem c 251 0
[Tekkaman2440@SBC2440V4]#cd /tmp/
[Tekkaman2440@SBC2440V4]#./IO_mem_test
io_addr : c485e0d0
IO_mem: the module can not lseek!
please input the command :1
IO_mem: ioctl 1 ok!
please input the command :8
IO_mem: ioctl STATUS ok! current_status=0X1
please input the command :3
IO_mem: ioctl 3 ok!
please input the command :q
[Tekkaman2440@SBC2440V4]#./IO_porttest_sleep &
[Tekkaman2440@SBC2440V4]#./IO_porttest_sleep &
[Tekkaman2440@SBC2440V4]#./IO_porttest_sleep &
[Tekkaman2440@SBC2440V4]#./IO_port_test
IO_port: the module can not lseek!
please input the command :1
IO_port: ioctl 1 ok!
please input the command :8
IO_port: ioctl STATUS ok! current_status=0X1
please input the command :3
IO_port: ioctl 3 ok!
please input the command :8
IO_port: ioctl STATUS ok!  current_status=0X3
please input the command :q
[1] Done ./IO_porttest_sleep
[Tekkaman2440@SBC2440V4]#ps
  PID Uid VSZ Stat Command
    1 root 1744 S init
    2 root SW< [kthreadd]
```

```
    3 root SWN [ksoftirqd/0]
    4 root SW< [watchdog/0]
    5 root SW< [events/0]
    6 root SW< [khelper]
   61 root SW< [kblockd/0]
   62 root SW< [ksuspend_usbd]
   65 root SW< [khubd]
   67 root SW< [kseriod]
   79 root SW [pdflush]
   80 root SW [pdflush]
   81 root SW< [kswapd0]
   82 root SW< [aio/0]
  709 root SW< [mtdblockd]
  710 root SW< [nftld]
  711 root SW< [inftld]
  712 root SW< [rfdd]
  746 root SW< [kpsmoused]
  755 root SW< [kmmcd]
  773 root SW< [rpciod/0]
  782 root 1752 S -sh
  783 root 1744 S init
  785 root 1744 S init
  787 root 1744 S init
  790 root 1744 S init
  843 root 1336 S ./IO_porttest_sleep
  844 root 1336 S ./IO_porttest_sleep
  846 root 1744 R ps
[Tekkaman2440@SBC2440V4]#ps
  PID Uid VSZ Stat Command
    1 root 1744 S init
    2 root SW< [kthreadd]
    3 root SWN [ksoftirqd/0]
    4 root SW< [watchdog/0]
    5 root SW< [events/0]
    6 root SW< [khelper]
   61 root SW< [kblockd/0]
   62 root SW< [ksuspend_usbd]
   65 root SW< [khubd]
   67 root SW< [kseriod]
   79 root SW [pdflush]
   80 root SW [pdflush]
   81 root SW< [kswapd0]
   82 root SW< [aio/0]
  709 root SW< [mtdblockd]
  710 root SW< [nftld]
  711 root SW< [inftld]
  712 root SW< [rfdd]
  746 root SW< [kpsmoused]
  755 root SW< [kmmcd]
  773 root SW< [rpciod/0]
  782 root 1752 S -sh
  783 root 1744 S init
  785 root 1744 S init
  787 root 1744 S init
  790 root 1744 S init
  847 root 1744 R ps
[3] + Done ./IO_porttest_sleep
[2] + Done ./IO_porttest_sleep
```

程序是针对2440的，若是用2410只需要改改测试的io口就好了！

wangtisheng 2014-01-11 22:45:55

tekkamanninja，IO_port_test.tar.gz中少了个文件啦~~IO_port_test.h~~~~~~~

回复 | 举报

jyan123 2011-04-25 11:39:00

1、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”，而IO端口有两种编址方式：独立编址和统一编制。而具体采用哪一种则取决于CPU的体系结构。 如，PowerPC、m68k等采用统一编址，而X86等则采用独立编址。但对于Linux内核而言，它可能用于不同的CPU，所以它必须都要考虑这两种方式，于是它采用一种新的方法，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），不论你采用哪种方式，都要先申请IO区域：request_resource()，结束时释放它：release_resource()。
对于某一既定的系统，它要么是独立编址，也即“I/O端口”方式，外设寄存器位于“I/O空间”；要么是统一编制，也即“I/O内存”方式，外设寄存器位于“内存空间”（很多外设有自己的内存、缓冲区，外设的寄存器和内存统称“I/O空间”）。
2、对外设的访问分为IO端口访问和IO内存访问。
访问IO内存的流程是：request_mem_region() -> ioremap() -> i

回复 | 举报