Linux设备驱动之I/O端口与I/O内存

一、统一编址与独立编址

该部分来自于：http://blog.chinaunix.net/space.php?uid=21347954&do=blog&id=443670，感谢mason\_hu的分享。

　　从CPU连出来一把线：数据总线、地址总线、控制总线，这把线上挂着N个接口，有相同的，有不同的，名字叫做存储器接口、中断控制接口、DMA接口、并行接口、串行接口、AD接口……一个设备要想接入，就用自己的接口和总线上的某个匹配接口对接……于是总线上出现了各种设备：内存、硬盘，鼠标、键盘，显示器……

对于CPU而言，如果它要发数据到某个设备，其实是发到对应的接口，接口电路里有多个寄存器（也称为端口），访问设备实际上是访问相关的端口，所有的信息会由接口转给它的设备。那么CPU会准备数据到数据总线，但是诸多接口，该发给谁呢？这时就须要为各接口分配一个地址，然后把地址放在地址总线上，需要的控制信息放到控制总线上，就可以和设备通信了。

　　对一个系统而言，通常会有多个外设，每个外设的接口电路中，又会有多个端口，每个端口都需要一个地址，为他们标识一个具体的地址值，是系统必须解决的事，与此同时，你还有个内存条，可能是512M或1G或更大的金士顿、现代DDR2之类，他们的每一个地址也都需要分配一个标识值，另外，很多外设有自己的内存、缓冲区，就像你的内存条一样，你同样需要为它们分配内存……你的CPU可能需要和它们的每一个字节都打交道，所以：别指望偷懒，它们的每一寸土地都要规划好！这听起来就很烦，做起来可能就直接导致脑细胞全部阵亡。但事情总是得有人去做，ARM可能会这样做：他这次设计的CPU是32位的，最多也就能寻址2^32=4G空间，于是把这4GB空间丢给内存和端口，让他们瓜分。但英特尔或许有更好的分配方式……

1、地址的概念

1）物理地址：CPU地址总线传来的地址，由硬件电路控制其具体含义。物理地址中很大一部分是留给内存条中的内存的，但也常被映射到其他存储器上（如显存、BIOS等）。在程序指令中的虚拟地址经过段映射和页面映射后，就生成了物理地址，这个物理地址被放到CPU的地址线上。

物理地址空间，一部分给物理RAM（内存）用，一部分给总线用，这是由硬件设计来决定的，因此在32 bits地址线的x86处理器中，物理地址空间是2的32次方，即4GB，但物理RAM一般不能上到4GB，因为还有一部分要给总线用（总线上还挂着别的许多设备）。在PC机中，一般是把低端物理地址给RAM用，高端物理地址给总线用。

2）总线地址：总线的地址线或在地址周期上产生的信号。外设使用的是总线地址，CPU使用的是物理地址。

物理地址与总线地址之间的关系由系统的设计决定的。在x86平台上，物理地址就是总线地址，这是因为它们共享相同的地址空间——这句话有点难理解，详见下面的“独立编址”。在其他平台上，可能需要转换/映射。比如：CPU需要访问物理地址是0xfa000的单元，那么在x86平台上，会产生一个PCI总线上对0xfa000地址的访问。因为物理地址和总线地址相同，所以凭眼睛看是不能确定这个地址是用在哪儿的，它或者在内存中，或者是某个卡上的存储单元，甚至可能这个地址上没有对应的存储器。

3）虚拟地址：现代操作系统普遍采用虚拟内存管理（Virtual Memory Management）机制，这需要MMU（Memory Management Unit）的支持。MMU通常是CPU的一部分，如果处理器没有MMU，或者有MMU但没有启用，CPU执行单元发出的内存地址将直接传到芯片引脚上，被内存芯片（物理内存）接收，这称为物理地址（Physical Address），如果处理器启用了MMU，CPU执行单元发出的内存地址将被MMU截获，从CPU到MMU的地址称为虚拟地址（Virtual Address），而MMU将这个地址翻译成另一个地址发到CPU芯片的外部地址引脚上，也就是将虚拟地址映射成物理地址。

　　Linux中，进程的4GB（虚拟）内存分为用户空间、内核空间。用户空间分布为0~3GB（即PAGE\_OFFSET，在0X86中它等于0xC0000000），剩下的1G为内核空间。程序员只能使用虚拟地址。系统中每个进程有各自的私有用户空间（0～3G），这个空间对系统中的其他进程是不可见的。

　　CPU发出取指令请求时的地址是当前上下文的虚拟地址，MMU再从页表中找到这个虚拟地址的物理地址，完成取指。同样读取数据的也是虚拟地址，比如mov ax, var. 编译时var就是一个虚拟地址，也是通过MMU从也表中来找到物理地址，再产生总线时序，完成取数据的。

2、编址方式

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。

　　统一编址：外设接口中的IO寄存器（即IO端口）与主存单元一样看待，每个端口占用一个存储单元的地址，将主存的一部分划出来用作IO地址空间，如，在PDP-11中，把最高的4K主存作为IO设备寄存器地址。端口占用了存储器的地址空间，使存储量容量减小。

　　统一编址也称为“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”）。

　　如，Samsung的S3C2440，是32位ARM处理器，它的4GB地址空间被外设、RAM等瓜分：

　　　　0x8000 1000 LED 8\*8点阵的地址

　　　　0x4800 0000 ~ 0x6000 0000 SFR（特殊暂存器）地址空间

　　　　0x3800 1002 键盘地址

　　　　0x3000 0000 ~ 0x3400 0000 SDRAM空间

　　　　0x2000 0020 ~ 0x2000 002e IDE

　　　　0x1900 0300 CS8900

　　独立编址（单独编址）：IO地址与存储地址分开独立编址，I/O端口地址不占用存储空间的地址范围，这样，在系统中就存在了另一种与存储地址无关的IO地址，CPU也必须具有专用与输入输出操作的IO指令（IN、OUT等）和控制逻辑。独立编址下，地址总线上过来一个地址，设备不知道是给IO端口的、还是给存储器的，于是处理器通过MEMR/MEMW和IOR/IOW两组控制信号来实现对I/O端口和存储器的不同寻址。如，intel 80x86就采用单独编址，CPU内存和I/O是一起编址的，就是说内存一部分的地址和I/O地址是重叠的。

　　独立编址也称为“I/O端口”方式，外设寄存器位于“I/O（地址）空间”。

　　对于x86架构来说，通过IN/OUT指令访问。PC架构一共有65536个8bit的I/O端口，组成64K个I/O地址空间，编号从0~0xFFFF，有16位，80x86用低16位地址线A0-A15来寻址。连续两个8bit的端口可以组成一个16bit的端口，连续4个组成一个32bit的端口。I/O地址空间和CPU的物理地址空间是两个不同的概念，例如I/O地址空间为64K，一个32bit的CPU物理地址空间是4G。如，在Intel 8086+Redhat9.0 下用“more /proc/ioports”可看到：

　　　　0000-001f : dma1

　　　　0020-003f : pic1

　　　　0040-005f : timer

　　　　0060-006f : keyboard

　　　　0070-007f : rtc

　　　　0080-008f : dma page reg

　　　　00a0-00bf : pic2

　　　　00c0-00df : dma2

　　　　00f0-00ff : fpu

　　　　0170-0177 : ide1

　　　　……

　　不过Intel x86平台普通使用了名为内存映射（MMIO）的技术，该技术是PCI规范的一部分，IO设备端口被映射到内存空间，映射后，CPU访问IO端口就如同访问内存一样。看Intel TA 719文档给出的x86/x64系统典型内存地址分配表：

　　　　系统资源 占用

------------------------------------------------------------------------

　　　　BIOS 1M

　　　　本地APIC 4K

　　　　芯片组保留 2M

　　　　IO APIC 4K

　　　　PCI设备 256M

　　　　PCI Express设备 256M

　　　　PCI设备（可选） 256M

　　　　显示帧缓存 16M

　　　　TSEG 1M

　　对于某一既定的系统，它要么是独立编址、要么是统一编址，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，存在IO空间的概念。目前，大多数嵌入式微控制器如ARM、PowerPC等并不提供I/O空间，仅有内存空间，可直接用地址、指针访问。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()。

二、linux I/O端口与I/O内存

IO端口：当一个寄存器或者内存位于IO空间时；

IO内存：当一个内存或者寄存器位于内存空间时；

　　在一些CPU制造商在其芯片上实现了一个单地址空间（统一编址）的同时，其它的CPU制造商认为外设不同于内存，应该有一个独立的地址空间给外设（单独编址），其生产处理器(特别是x86家族)的I/O端口有自己的读写信号线和特殊的CPU指令来存取端口。因为外设要与外设总线相匹配，并且大部分流行的I/O总线都是以个人计算机(主要是x86家族)作为模型，所以即便那些没有单独地址空间给I/O端口的处理器，也必须在访问外设时模拟成读写端口。这通常通过外部芯片组（PC中的南北桥）或者在CPU核中附加额外电路来实现（基于嵌入式应用的处理器）。

　　由于同样的理由，Linux在所有计算机平台上都实现了I/O端口，甚至在那些单地址空间的CPU平台上（模拟I/O端口）。但并不是所有的设备都会将其寄存器映射到I/O端口。虽然ISA设备普遍使用I/O端口，但大部分PCI设备将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存取内存也更有效率，并且编译器在存取内存时在寄存器分配和寻址模式的选择上有更多自由。

1.IO寄存器和常规内存

　　I/O寄存器和RAM的主要不同是I/O操作有边际效应（side effect），而内存操作没有:访问内存只是在内存某一位置存储数值。因为内存存取速度严重影响CPU的性能，编译器可能会对源码进行优化，主要是：使用高速缓存和重排读/写指令的顺序。对于传统内存(至少在单处理器系统)这些优化是透明有益的，但是对于I/O 寄存器，这可能是致命错误，因为它们干扰了那些"边际效应"（驱动程序存取I/O 寄存器就是为了获取边际效应）。因此，驱动程序必须确保在存取寄存器时，不能使用高速缓存并且不能重新编排读写指令的顺序。

　　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 。举个例子，有些设备的中断状态寄存器只要一读取，便自动清零。

　　硬件缓冲的问题是最易解决的：只要将底层硬件配置(或者自动地或者通过Linux 初始化代码)为当存取I/O区时，禁止任何硬件缓冲(不管是I/O 内存还是I/O 端口)。

　　编译器优化和硬件重编排读写指令顺序的解决方法是：在硬件或处理器必须以一个特定顺序执行的操作之间安放一个内存屏障（memory barrier）。

2.操作IO端口（申请，访问，释放）：

　　I/O 端口是驱动用来和很多设备通讯的方法。

（1）申请I/O 端口：

　　在驱动还没独占设备之前，不应对端口进行操作。内核提供了一个注册接口，以允许驱动声明其需要的端口：

/\* request\_region告诉内核：要使用first开始的n个端口。参数name为设备名。如果分配成功返回值是非NULL；否则无法使用需要的端口(/proc/ioports包含了系统当前所有端口的分配信息，若request\_region分配失败时，可以查看该文件，看谁先用了你要的端口) \*/

struct resource \*request\_region(unsigned long first, unsigned long n, const char \*name);

（2）访问IO端口：

　　在驱动成功请求到I/O 端口后，就可以读写这些端口了。大部分硬件会将8位、16位和32位端口区分开，无法像访问内存那样混淆使用。驱动程序必须调用不同的函数来访问不同大小的端口。

　　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来存取I/O端口:

复制代码

/\* inb/outb:读/写字节端口(8位宽)。有些体系将port参数定义为unsigned long；而有些平台则将它定义为unsigned short。inb的返回类型也是依赖体系的 \*/

unsigned inb(unsigned port);

void outb(unsigned char byte, unsigned port);

/\* inw/outw:读/写字端口(16位宽） \*/

unsigned inw(unsigned port);

void outw(unsigned short word, unsigned port);

/\* inl/outl:读/写32位端口。longword也是依赖体系的，有的体系为unsigned long；而有的为unsigned int \*/

unsigned inl(unsigned port);

void outl(unsigned longword, unsigned port);

复制代码

（3）释放IO端口：

/\* 用完I/O端口后(可能在模块卸载时)，应当调用release\_region将I/O端口返还给系统。参数start和n应与之前传递给request\_region一致 \*/

void release\_region(unsigned long start, unsigned long n);

3.操作IO内存（申请，映射，访问，释放）：

　　尽管 I/O 端口在x86世界中非常流行，但是用来和设备通讯的主要机制是通过内存映射的寄存器和设备内存，两者都称为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 内存仅仅是一个类似于RAM 的区域，处理器通过总线访问该区域，以实现对设备的访问。同样，读写这个区域是有边际效应。

　　根据计算机体系和总线不同，I/O 内存可分为可以或者不可以通过页表来存取。若通过页表存取，内核必须先重新编排物理地址，使其对驱动程序可见，这就意味着在进行任何I/O操作之前，你必须调用ioremap；如果不需要页表，I/O内存区域就类似于I/O端口，你可以直接使用适当的I/O函数读写它们。

　　由于边际效应的缘故，不管是否需要 ioremap，都不鼓励直接使用I/O内存指针，而应使用专门的I/O内存操作函数。这些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。

（1）申请I/O 内存：

　　I/O 内存区在使用前必须先分配。分配内存区的函数接口在<linux/ioport.h>定义中：

/\* request\_mem\_region分配一个开始于start,len字节的I/O内存区。分配成功，返回一个非NULL指针；否则返回NULL。系统当前所有I/O内存分配信息都在/proc/iomem文件中列出，你分配失败时，可以看看该文件，看谁先占用了该内存区 \*/

struct resource \*request\_mem\_region(unsigned long start, unsigned long len, char \*name);

（2）映射：

　　在访问I/O内存之前，分配I/O内存并不是唯一要求的步骤，你还必须保证内核可存取该I/O内存。访问I/O内存并不只是简单解引用指针，在许多体系中，I/O 内存无法以这种方式直接存取。因此，还必须通过ioremap 函数设置一个映射。

/\* ioremap用于将I/O内存区映射到虚拟地址。参数phys\_addr为要映射的I/O内存起始地址，参数size为要映射的I/O内存的大小，返回值为被映射到的虚拟地址 \*/

void \*ioremap(unsigned long phys\_addr, unsigned long size);

（3）访问IO内存：

　　经过 ioremap之后，设备驱动就可以存取任何I/O内存地址。注意，ioremap返回的地址不可以直接解引用；相反，应当使用内核提供的访问函数。访问I/O内存的正确方式是通过一系列专门用于实现此目的的函数：

复制代码

#include <asm/io.h>

/\* I/O内存读函数。参数addr应当是从ioremap获得的地址(可能包含一个整型偏移); 返回值是从给定I/O内存读取到的值 \*/

unsigned int ioread8(void \*addr);

unsigned int ioread16(void \*addr);

unsigned int ioread32(void \*addr);

/\* I/O内存写函数。参数addr同I/O内存读函数，参数value为要写的值 \*/

void iowrite8(u8 value, void \*addr);

void iowrite16(u16 value, void \*addr);

void iowrite32(u32 value, void \*addr);

/\* 以下这些函数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。参数count表示要读写的数据个数，而不是字节大小 \*/

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,，onst void \*buf,，nsigned long count);

/\* 需要操作一块I/O 地址时，使用下列函数(这些函数的行为类似于它们的C库类似函数): \*/

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);

/\* 旧的I/O内存读写函数，不推荐使用 \*/

unsigned readb(address);

unsigned readw(address);

unsigned readl(address);

void writeb(unsigned value, address);

void writew(unsigned value, address);

void writel(unsigned value, address);

复制代码

（4）释放IO内存步骤：

void iounmap(void \* addr); /\* iounmap用于释放不再需要的映射 \*/

void release\_mem\_region(unsigned long start, unsigned long len); /\* iounmap用于释放不再需要的映射 \*/

4、像IO内存一样使用端口

　　一些硬件有一个有趣的特性: 有些版本使用 I/O 端口；而有些版本则使用 I/O 内存。不管是I/O 端口还是I/O 内存，处理器见到的设备寄存器都是相同的，只是访问方法不同。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:

/\* ioport\_map重新映射count个I/O端口，使它们看起来I/O内存。此后，驱动程序可以在ioport\_map返回的地址上使用ioread8和同类函数。这样，就可以在编程时，消除了I/O 端口和I/O 内存的区别 \*/

void \*ioport\_map(unsigned long port, unsigned int count);

void ioport\_unmap(void \*addr);/\* ioport\_unmap用于释放不再需要的映射 \*/

注意，I/O 端口在重新映射前必须使用request\_region分配分配所需的I/O 端口。

5、ARM体系的IO操作接口

　　s3c24x0处理器使用的是I/O内存，也就是说：s3c24x0处理器使用统一编址方式，I/O寄存器和内存使用的是单一地址空间，并且读写I/O寄存器和读写内存的指令是相同的。所以推荐使用I/O内存的相关指令和函数。但这并不表示I/O端口的指令在s3c24x0中不可用。如果你注意过s3c24x0关于I/O方面的内核源码，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。注意以下几点：

　　1）所有的读写指令（I/O操作函数）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，如在include/asm-arm/arch-s3c2410/regs-xxx.h中定义了s3c2410处理器各外设寄存器地址（其他处理器芯片也可在类似路径找到内核定义好的外设寄存器的虚拟地址；另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（资源busy）。但是不推荐这么做，这样的代码也不规范，可能会引起并发问题（很多时候我们都需要独占设备）。

　　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。

　　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。