第22章 Linux I/O端口编程

本章介绍有关Linux I/O端口编程的内容，如在C语言下使用I/O端口、硬件中断与IMA存取等方面的内容。

22.1 如何在 C 语言下使用I/O端口

22.1.1 一般的方法

用来存取 I/O 端口的子过程都放在文件 /usr/include/asm/io.h 里(或放在内核源代码程序的linux/include/asm-i386/io.h 文件里)。这些子过程是以嵌入宏的方式写成的，所以使用时只要以#include<asm/io.h> 的方式引用就够了，不需要附加任何函数库。

因为 gcc以及 egcs的限制，你在编译任何使用到这些子过程的源代码时必须打开最优化选项 (gcc -O1或较高层次的)，或者在做 #include <asm/io.h> 这个动作前使用 #define extern 将 extern 定义成空白。

为了除错的目的，你编译时可以使用 gcc -g -O (至少现在的 gcc 版本是这样)，但是最优化之后有时可能会让调试器的行为变得有点奇怪。如果这个状况对你而言是个困扰，你可以将所有使用到 I/O端口的子过程集中放在一个文件里，并只在编译该文件时打开最优化选项。

在你存取任何 I/O 端口之前，你必须让程序有如此做的权限。要完成这个目的，你可以在程序一开始的地方 (但是，要在任何 I/O 端口存取动作之前) 调用 ioperm()这个函数 (该函数在文件 unistd.h中，并且被定义在 内核中)。 使用语法是 ioperm(from，num，turn_on)，其中 from 是第一个允许存取的 I/O 端口地址，num是接着连续存取 I/O 端口地址的数目。 例如，ioperm(0x300，5，1)的意思就是说允许存取端口 0x300 到 0x304 (一共五个端口地址)。

而最后一个参数是一个布尔代数值，用来指定是否给予程序存取 I/O 端口的权限 (true (1)) 或者除去存取的权限 (false (0))。 你可以多次调用函数 ioperm()以便使用多个不连续的端口地址。

你的程序必须拥有 root 权限才能调用函数 ioperm()；所以你如果不是以 root身份执行该程序，就得将该程序设置成 root。 当你调用过函数 ioperm() 打开I/O 端口的存取权限后你便可以拿掉 root 的权限。在你的程序结束之后并不特别要求你以 ioperm(...，0) 这个方式拿掉 I/O 端口的存取权限；因为当你的程序执行完毕之后，这个动作会自动完成。

调用函数 setuid() 将目前执行程序的有效用户识别码 (ID) 设定成非 root的用户，并不影响其先前以 ioperm() 的方式所取得的 I/O 端口存取权限，但是调用函数fork() 的方式却会有所影响 (虽然父进程保有存取权限，但是子进程却无法取得存取权限)。

函数 ioperm() 只能让你取得端口地址 0x000 到 0x3ff 的存取权限；至于较高地址的端口，你得使用函数 iopl() (该函数让你一次可以存取所有的端口地址)。将权限等级参数值设为 3 (例如，iopl(3))，以便你的程序能够存取所有的I/O 端口(因此要小心，如果存取到错误的端口地址将对你的计算机造成各种不可预期的损害。同样地，调用函数 iopl() 你得拥有 root 的权限。

接着，我们来实际地存取 I/O 端口。要从某个端口地址输入一个字节(8位)的信息，你得调用函数 inb(port) ，该函数会传回所取得的一个字节的信息。要输出一个字节的信息，你得调

用函数 outb(value, port) (请记住参数的次序)。要从某两个端口地址 x 和 x+1 (两个字节组成一个字，故使用组合语言指令inw)输入一个字 (16 个 bit) 的信息，你得调用函数 inw(x); 要输出一个字的信息到两个端口地址，你得调用函数 outw(value, x)。如果你不确定使用哪个端口指令(字节或字)，你大概须要 inb() 与 outb() 这两个端口指令，因为大多数的设备都是采用字节大小的端口存取方式来设计的。注意所有的端口存取指令都至少需要大约 1μs 的时间。

如果你使用的是 inb_p()、outb_p()、inw_p()以及 outw_p() 等宏指令，在你对端口地址存取动作之后只需很短的(大约为1μs) 延迟时间就可以完成;你也可以让延迟时间变成大约 4μs ，方法是在使用 #include <asm/io.h> 之前使用#define REALLY_SLOW_IO。 这些宏指令通常 (除非你使用的是 #define SLOW_IO_BY_JUMPING，这个方法可能不太准确) 会利用输出信息到端口地址 0x80以便达到延迟时间的目的，所以你得先以函数 ioperm() 取得端口地址 0x80 的使用权限(输出信息到端口地址 0x80 不应该会对系统的其他部分造成影响)。 至于其他通用的延迟时间的方法，请参考下面的内容。

22.1.2 另一个替代方法: /dev/port

另一个存取 I/O 端口的方法是以函数 open() 打开文件 /dev/port(一个字符设备，主设备编号为 1，次设备编号为 4)， 以便执行读与(/或)写的动作(注意标准输出入 函数 f*() 有内部的缓冲 ，所以要避免使用)。接着使用 lseek() 函数以便在该字符设备文件中找到某个字节信息的正确位置(文件位置 0 = 端口地址 0x00，文件位置 1 = 端口地址 0x01，以此类推)，然后你可以使用read() 或 write() 函数对某个端口地址做读或写一个字节的动作。

这个方法就是在你的程序里使用 read/write 函数来存取 /dev/port字符设备文件。 这个方法的执行速度或许比前面所讲的一般方法还慢，但是不需要编译器的最优化函数，也不需要使用函数 ioperm() 。 如果你允许非 root 用户或群组存取/dev/port 字符设备，操作时就不需拥有 root 权限。但是，对于系统安全而言，这样做非常糟糕，因为它可能伤害到你的系统，或许，会有人因此而取得 root 的权限，利用 /dev/port 字符设备文件直接在硬盘、网络卡等设备上进行存取操作。

22.2 硬件中断 与 DMA 存取

你的程序如果在用户模式下执行，不可以直接使用硬件中断 (IRQ) 或DMA。你必须编写一个内核驱动程序。也就是说，你在用户模式中所写的程序无法控制硬件中断的产生。

22.3 高精确的时间

22.3.1 延迟时间

在用户模式中执行的进程不能精确地控制时间，因为 Linux 是个多用户的操作环境，在执行中的进程随时会因为各种原因被暂停大约 10ms到数秒 (在系统负荷非常高的时候)。 然而，对于大多数使用 I/O 端口的应用程序而言，这个延迟时间实际上算不了什么。要缩短延迟时间，你得使用函数 nice 将你在执行中的进程设定成高优先权(请参考 nice(2)使用说明文件)，或使用即时调度法 (real-time scheduling) (请看下面介绍)。

如果你想获得比在一般用户模式中执行的进程还要精确的时间，有一些方法可以让你在用户模式中做到“ 即时调度 ”的支持。Linux 2.x 版本的内核中有软件方式的即时调度支持。

1. 睡眠: sleep() 与 usleep()

现在，让我们开始进行较简单的时间函数调用。 想要延迟数秒的时间，最佳的方法大概是使用函数 sleep() 。 想要延迟至少数十毫秒的时间 (10 ms 似乎已是最短的延迟时间了)，函数 usleep() 应该可以使用。这些函数把 CPU 的使用权让给其他想要执行的进程，所以没有浪费掉CPU的时间。

如果让出 CPU 的使用权因而使得时间延迟了大约 50ms(这取决于处理器与机器的速度，以及系统的负荷)，那就浪费掉 CPU 太多的时间，因为Linux 的调度器 (scheduler) (单就 x86 结构而言) 在将控制权发还给你的进程之前通常至少要花费 10～30ms的时间。因此，短时间的延迟，使用函数usleep(3) 所得到的延迟结果通常会大于你在参数所指定的值，大约至少有 10 ms。

2. nanosleep()

在 Linux 2.0.x 一系列的内核发行版本中，有一个新的系统调用， nanosleep() (请参考 nanosleep(2) 的说明文件)，它让你能够休息或延迟一个短的时间 (数微秒或更多)。

3. 使用 I/O 端口延迟时间

另一个延迟数微秒的方法是使用 I/O 端口。 就是从端口地址 0x80 输入或输出任何字节的信息 (请参考前面) 等待的时间应该几乎只要 1μs 这要看你的处理器的类型与速度。如果要延迟数微秒的时间，你可以将这个动作多做几次。 在任何标准的机器上输出信息到该端口地址，应该不会有不良的后果 (而且有些内核的设备驱动程序也在使用它)。{in|out}[bw]_p() 等函数就是使用这个方法来产生时间延迟的。

实际上，一个使用到端口地址范围为 0～0x3ff 的 I/O 端口指令，几乎只要 1μs 的时间，所以如果你要如此做，例如，直接使用并行端口，只要加上几个 inb() 函数从该端口地址范围读入字节的信息即可。

4. 使用组合语言来延迟时间

如果你知道执进程序所在机器的处理器类型与时钟速度，你可以执行某些组合语言指令以获得较短的延迟时间 (但是记住，你在执行中的进程随时会被暂停， 所以，有时延迟的时间会比实际长)。如下面列表所示，内部处理器的速度决定了所要使用的时钟周期数；例如，一个 50 MHz 的处理器 (486DX-50或 486DX2-50)，一个时钟周期要花费 1/50 000 000 s (=20ns)。

指令	i386 时钟周期数	i486 时钟周期数
nop	3	1
xchg %ax，%ax	3	3
or %ax，%ax	2	1
mov %ax，%ax	2	1
add %ax，0	2	1

上面的列表中，指令 nop 与 xchg 应该不会有不良的后果。 指令最后可能会改变标志寄存器的内容，但是，这没关系，因为 gcc 会处理。 指令 nop 是个好的选择。

想要在你的程序中使用这些指令，你得使用 asm('instruction')。指令的语法就如同上面列表的用法; 如果你想要在单一的 asm() 叙述中使用多个指令，可以使用分号将它们隔开。 例如，asm('nop ; nop ; nop ; nop') 会执行4个 nop 指令，在 i486 或 Pentium 处理器中会延迟4个时钟周期 (i386 会延迟 12 个时钟周期)。

gcc 会将 asm() 翻译成单行组合语言程序码，所以不会有调用函数的负荷。在 Intel x86 结构中不可能有比1个时钟周期还短的时间延迟。

5. 在 Pentium 处理器上使用函数 rdtsc

对于 Pentium 处理器而言，你可以使用下面的 C 语言程序计算自从上次重新开机到现在经

过了多少个时钟周期:

```
extern __inline__ unsigned long long int rdtsc()
{
   unsigned long long int x;
   __asm__ volatile (".字节 0x0f, 0x31" : "=A" (x));
   return x;
}
```

你可以查询并参考此值以便延迟你想要的时钟周期数。

22.3.2 时间的量测

想要时间精确到 1s, 使用函数 time() 或许是最简单的方法。想要时间更精确, 函数 gettimeofday() 大约可以精确到微秒 (但是如前所述会受到 CPU 调度的影响)。至于Pentium 处理器, 使用上面的程序片断就可以精确到一个时钟周期。

如果要执行中的进程在一段时间到了之后能够被通知 (get a signal), 可以使用函数 setitimer() 或 alarm() 。

22.4 使用其他程序语言

上面的说明集中在 C 程序语言。它应该可以直接应用在 C++ 及 Objective C语言之上。至于组合语言部分, 虽然你必须先在 C 语言中调用函数 ioperm() 或 iopl(), 但是, 随后可以直接使用 I/O 端口读写指令。

至于其他程序语言, 除非你可以在该程序语言中插入单行组合语言或 C语言的程序码, 或者使用上面所说的系统调用, 否则, 倒不如编写一个内含有存取 I/O端口或延迟时间所必须使用的函数的C原始程序, 编译之后再与你的程序连接。要不然就使用前面所说的 /dev/port 字符设备文件。

22.5 一些有用的 I/O 端口

如果你要按照其原始的设计目的来使用这些或其他常用的 I/O 端口(例如, 控制一般的打印机或数据机), 你应该使用现成的设备驱动程序 (它通常含在内核中), 而不会如本文所说的去编写 I/O 端口程序。本节主要是提供给那些想要将液晶显示器 (LCD)、步进电动机或其他商业电子产品连接到PC标准 I/O 端口的人。

22.5.1 并行端口

并行端口的基本端口地址 (以下称为BASE)对于 /dev/lp0 是 0x3bc , 对于 /dev/lp1是 0x378 , 对于 /dev/lp2 是 0x278 。

除了下面即将描述的标准只输出模式, 大多数的并行端口都有扩充的双向模式。因为在用户模式中的程序无法使用 IRQ 或 DMA, 想要使用 ECP/EPP模式, 或许得编写一个内核的设备驱动程序。

端口地址 BASE+0 (信息端口) 用来控制信息端口的信号电平 (D0 到 D7 分别代表着 bit 0 到7, 电平状态: 0 = 低电平 (0 V), 1 = 高电平 (5 V))。一个写入信息到该端口的动作, 会将信息信号电平拴在端口的端脚上。一个将该端口的信息读出的动作会将上一次以标准只输出模式或扩充的写入模式所拴住的信息信号电平读回, 或者以扩充读出模式从另外一个设备将端脚上的信息信号电平读回。

端口地址 BASE+1 (状态端口) 是个只读入的端口，会将下面的输入信号电平读回：

bits 0 和 1 保留不用。

bit 2 IRQ 的状态。

bit 3 ERROR (1=高电平)。

bit 4 SLCT (1=高电平)。

bit 5 PE (1=高电平)。

bit 6 ACK (1=高电平)。

bit 7 -BUSY (0=高电平)。

端口地址 BASE+2 (控制端口) 是个只写入的端口(一个将该端口的信息读出的动作仅会将上一次写入的信息信号电平读回)，用来控制下面的状态信号：

bit 0 -STROBE (0=高电平)。

bit 1 AUTO_FD_XT (1=高电平)。

bit 2 -INIT (0=高电平)。

bit 3 SLCT_IN (1=高电平)。

bit 4 当被设定为 1 时允许并行端口产生 IRQ 信号 (发生在 ACK端脚的电平由低变高的瞬间)。

bit 5 用来控制扩充模式时端口的输出入方向 (0 = 写，1 = 读)，这是个只写的端口 (一个将该端口的信息读出的动作对此 bit 一点用处也没有)。

bits 6和7 保留不用。

端口的端脚排列方式 (该端口是一个 25 只脚 D 字形外壳的母连接器，i=输入，o=输出)如下：

1io —— STROBE，2io —— D0，3io —— D1，4io —— D2，5io —— D3，6io —— D4，7io —— D5，8io —— D6，9io D7，10i —— ACK，11i —— BUSY，12i —— PE，13i —— SLCT，14o —— AUTO_FD_XT，15i —— ERROR，16o INIT，17o —— SLCT_IN，18 —— 25 Ground

22.5.2 游戏端口

游戏端口的端口地址范围为 0x200-0x207。

端口的端脚排列方式 (该端口是一个 15 只脚 D 字形外壳的母连接器)如下：

1、8、9、15: +5 V (电源)。

4、5、12: 接地。

2、7、10、14: BA1、BA2、BB1和 BB2 等数位输入。

3、6、11、13: AX、AY、BX和 BY 等“类比”输入。

+5 V 的端脚似乎通常会直接连接到主机板的电源线上，所以它应该提供相当的电力，这还要看所使用主机板、电源以及游戏端口的类型。

数位输入用于操纵口的按钮可以让你连接两个操纵口的四个按钮 (操纵口A和操纵口B，各有两个按钮) 到游戏端口也就是数位输入的四个端脚。 它们应该是一般 TTL电压电平的输入，你可以直接从状态端口 (参考下面说明) 读出它们的电平状态。一个实际的操纵口在按钮被按下时会传回低电平 (0 V) 状态，否则传回高电平 (5V经由1kΩ的电阻连接到电源端脚) 状态。

所谓的类比输入实际是量测到的阻抗值。 游戏端口有四个单晶体多谐振荡器(一个 558 晶片) 连接到四个类比输入端脚。每个类比输入端脚与多谐振荡器的输出之间连接着一个 2.2k Ω 的电阻，而且多谐振荡器的输出与地之间连接着一个 0.01 μF 的时间电容。一个实际的操纵口的每个坐标 (X 和 Y) 上会有一个可变电阻，连接在 +5 V与每个相对的类比输入端脚之间 (端脚

AX 或 AY 是给操纵口A用的，而端脚 BX 或 BY是给操纵口B用的)。

操作的时候，多谐振荡器将其输出设定为高电平 (5 V)，并且等到时间电容上的电压达到 3.3 V 之后将相对的输出设定为低电平。 因此操纵口中多谐振荡器输出的高电平时间周期与可变电阻的电阻值成正比 (也就是，操纵口在相对坐标的位置)，如下所示:

R = (t - 24.2) / 0.011

其中，R是可变电阻的阻值 (Ω)，而t是高电平时间周期的长度 (s)。

因此，要读出类比输入端脚的数值，首先得启动多谐振荡器 (以端口写入的方式，请看下面)，然后查询四个坐标的信号状态(以持续的端口读出方式)，一直到信号状态由高电平变成低电平，计算其高电平时间周期的长度。这个持续查询的动作花费相当多的 CPU 时间，而且在一个非即时的多用户环境，所得的结果不是非常准确的。因为，你无法以固定的时间来查询信号的状态(除非你使用内核层次的驱动程序而且你得在查询的时候抑制掉中断的产生，但是这样做会浪费更多的 CPU 时间)。如果你知道信号的状态会花费一段不短的时间 (数十毫秒)而成为低电平，可以在查询之前调用函数 usleep() 将 CPU的时间让给其他想要执行的进程。

游戏端口中唯一需要你来存取的端口地址是 0x201 (其他的端口地址不是动作一样就是没用)。任何对这个端口地址所做的写入动作 (不论你写入什么) 都会启动多谐振荡器。对这个端口地址做读出动作会取回输入信号的状态:

bit 0: AX (1=高电平，多谐振荡器的输出状态)
bit 1: AY (1=高电平，多谐振荡器的输出状态)
bit 2: BX (1=高电平，多谐振荡器的输出状态)
bit 3: BY (1=高电平，多谐振荡器的输出状态)
bit 4: BA1 (数位输入，1=高电平)
bit 5: BA2 (数位输入，1=高电平)
bit 6: BB1 (数位输入，1=高电平)
bit 7: BB2 (数位输入，1=高电平)

22.5.3 串行端口

如果你的设备支持RS-232之类的接口，你应该可以使用串行端口。 Linux所提供的串行端口驱动程序应该能够应用在任何地方 (你应该不需要直接编写串行端口程序，或是内核的驱动程序)。它具有相当的通用性，所以如果使用非标准的速率以及其他等等，应该不是问题。