如何在 Linux 下撰寫程式來使用 I/O 埠

作者: Riku Saikkonen <Riku.Saikkonen@hut.fi>
譯者: Da-Wei Chiang <dawei@sinica.edu.tw> v, 28 December 1997 翻譯日期: 22 Jul. - 1 Aug. 1998

本文的內容說明了 Intel x86 架構下如何在使用者模式 (user-mode) 中撰寫程式來使用硬體I/O 埠以及等待一小段的時間周期.

目錄

1 介紹

本文的內容說明了 Intel x86 架構下如何在使用者模式 (user-mode) 中撰寫程式來使用硬體 I/O 埠以及等待一小段的時間周期. 內容源自於一篇非常短的文章 IO-Port mini-HOWTO 其作者與本文同.

如果您對本文有任何指教不論是錯誤修正或是內容補述, 都歡迎寄信給我 (Riku.Saikkonen@hut.fi)...

本文對前一次發行的版本 (Mar 30 1997) 作了如下的修正:

- 對於 inb_p/outb_p 和埠位址 0x80 之間的關係做出了澄清.
- 刪除了關於 udelay() 函式的資料, 因爲 nanosleep() 函式 提供了比較明確的使用方法.
- 將內容轉換成 Linuxdoc-SGML 格式, 並且重新作了些許的編排.
- 對很多地方作了些許的補述與修正.

2 如何在 C 語言下使用 I/O 埠

2.1 正規的方法

用來存取 I/O 埠的常式 (Routine) 都放在檔案 /usr/include/asm/io.h 裡 (或放在核心原始碼程式集的 linux/include/asm-i386/io.h 檔案裡). 這些常式是以單行巨集 (inline macros) 的方式寫成的, 所以使用時只要以 #include <asm/io.h> 的方式引用就夠了; 不需要附加任何函式館 (libraries).

譯注: 常式(Routine) 通常是指系統呼叫(System Call)與函式(Function)的總稱.

因爲 gcc (至少出現在 2.7.2.3 和以前的版本) 以及 egcs (所有的版本) 的限制, 你在編譯任何使用到這些常式的原始碼時 必須 打開最佳化選項 (gcc -O1 或較高層次的), 或者是在做 #include <asm/io.h> 這個動作前使用 #define extern 將 extern 定義成空白.

爲了除錯的目的, 你編譯時可以使用 gcc -g -O (至少現在的 gcc 版本是這樣), 但是最佳化之後有時可能會讓除錯器 (debugger) 的行爲變的有點奇怪. 如果這個狀況對你而言是個困擾, 你可以將所有使用到 I/O 埠的常式集中放在一個檔案裡並只在編譯該檔案時纔打開最佳化選項.

在你存取任何 I/O 埠之前, 你必須讓你的程式有如此做的權限. 要達成這個目的 你可以在你的程式一開始的地方 (但是要在任何 I/O 埠存取動作之前) 呼叫 ioperm() 這個函式 (該函式被宣告於檔案 unistd.h , 並且被定義在 核心中). 使用語法是 ioperm(from, num, turn_on), 其中 from 是第一個允許存取的 I/O 埠位址, num 是接著連續存取 I/O 埠位址的數目. 例如, ioperm(0x300, 5, 1) 的意思就是說允許存取埠 0x300 到 0x304 (一共五個埠位址). 而最後一個參數是一個布林代數值用來指定是否 給予程式存取 I/O 埠的權限 (true (1)) 或是除去存取的權限 (false (0)). 你 可以多次呼叫函式 ioperm() 以便使用多個不連續的埠位址. 至於語法的細節請 參考 ioperm(2) 的使用說明文件.

你的程式必須擁有 root 的權限纔能呼叫函式 ioperm() ; 所以你如果不是以 root 的身份執行該程式, 就是得將該程式 setuid 成 root. 當你呼叫過函式 ioperm() 打開 I/O 埠的存取權限後你便可以拿掉 root

的權限. 在你的程式結束之後並不特別 要求你以 ioperm(..., 0) 這個方式拿掉 I/O 埠的存取權限; 因爲當你的程式 執行完畢之後這個動作會自動完成.

呼叫函式 setuid() 將目前執行程式的有效使用者識別碼 (ID) 設定成非 root 的使用者並不影響其先前以 ioperm() 的方式所取得的 I/O 埠存取權限, 但是呼叫函式 fork() 的方式卻會有所影響 (雖然父行程 (parent process) 保有存取權限, 但是子行程 (child process) 卻無法取得存取權限).

函式 ioperm() 只能讓你取得埠位址 0x000 到 0x3ff 的存取權限; 至於 較高位址的埠, 你得使用函式 iopl() (該函式讓你一次可以存取所有的埠位址). 將權限等級參數值設爲 3 (例如, iopl(3)) 以便你的程式能夠存取 所有的 I/O 埠 (因此要小心 — 如果存取到錯誤的埠位址將對你的電腦造成各種不可預期的損害. 同樣地, 呼叫函式 iopl() 你得擁有 root 的權限.至於語法的細節請參考 iopl(2) 的使用說明文件.

接著, 我們來實際地存取 I/O 埠... 要從某個埠位址輸入一個 byte (8 個 bits) 的資料, 你得呼叫函式 inb(port) , 該函式會傳回所取得的一個 byte 的資料. 要輸出一個 byte 的資料, 你得呼叫函式 outb(value, port) (請記住參數的次序). 要從某二個埠位址 x 和 x+1 (二個 byte 組成一個 word, 故使用組合語言 指令 inw) 輸入一個 word (16 個 bits) 的資料, 你得呼叫函式 inw(x) ; 要輸出一個 word 的資料到二個埠位址, 你得呼叫函式 outw(value, x) . 如果你不確定使用那個埠指令 (byte 或 word), 你大概須要 inb() 與 outb() 這二個埠指令 — 因爲大多數的裝置都是採用 byte 大小的埠存取方式來設計的. 注意所有的埠存取指令都至少需要大約一微秒的時間來執行.

如果你使用的是 inb_p(), outb_p(), inw_p(), 以及 outw_p() 等巨集指令, 在你對埠位作址存取動作之後只需很短的(大約一微秒)延遲時間就可以完成; 你也可以讓延遲時間變成大約四微秒方法是在使用 #include <asm/io.h> 之前使用 #define REALLY_SLOW_IO. 這些巨集指令通常 (除非你使用的是 #define SLOW_IO_BY_JUMPING, 這個方法可能較不準確) 會利用輸出資料到埠位址 0x80 以便達到延遲時間的目的, 所以你得先以函式 ioperm() 取得埠位址 0x80 的使用權限 (輸出資料到埠位址 0x80 不應該會對系統的其他其他部分造成影響). 至於 其他通用的延遲時間的方法, 請繼續讀下去.

ioperm(2), iopl(2) 等函式, 和上面所述及的巨集指令的使用說明會收錄在 最近出版的 Linux 使用說明文件集中.

2.2 另一個替代的方法: /dev/port

另一個存取 I/O 埠的方法是以函式 open() 開啓檔案 /dev/port (一個字元裝置,主要裝置編號爲 1, 次要裝置編號爲 4) 以便執行讀且/或寫的動作 (注意標準輸出入 (stdio) 函式 f*() 有內部的緩衝 (buffering), 所以要避免使用). 接著使用 lseek() 函式以便在該字元裝置檔案中找到某個 byte 資料的正確位置 (檔案位置 0 = 埠位址 0x00, 檔案位置 1 = 埠位址 0x01, 以此類推), 然後你可以使用 read() 或 write() 函式對某個埠位址做讀或寫一個 byte 或 word 資料的動作.

這個替代的方法就是在你的程式裡使用 read/write 函式來存取 /dev/port 字元裝置檔案. 這個方法的執行速度或許比前面所講的一般方法還慢, 但是不需要編譯器 的最佳化功能也不需要使用函式 ioperm() . 如果你允許非 root 使用者或群組存取 /dev/port 字元設裝置案, 操作時就不需擁有 root 權限 – 但是對於系統安全而言 是個非常糟糕的事情, 因爲他可能傷害到你的系統, 或許會有人因而取的 root 的權限, 利用 /dev/port 字元裝置檔案直接存取硬碟, 網路卡, 等設備.

3 硬體中斷 (IRQs) 與 DMA 存取

你的程式如果在使用者模式 (user-mode) 下執行不可以直接使用硬體中斷 (IRQs) 或 DMA. 你必需撰寫一個核心驅動程式; 相關的細節請參考網頁 *The Linux Kernel Hacker's Guide* <http://www.redhat.com:

8080/HyperNews/get/khg.html> 以及拿核心程式原始碼來當範例.

就是說, 你在使用者模式 (user-mode) 中所寫的程式無法抑制硬體中斷的產生.

4 高精確的時序

4.1 延遲時間

首先, 我會說不保證你在使用者模式 (user-mode) 中執行的行程 (process) 能夠精確地控制時序因爲 Linux 是個多工的作業環境. 你在執行中的行程 (process) 隨時會因爲各種原因被暫停大約 10 毫秒到數秒 (在系統負荷非常高的時候). 然而, 對於大多數使用 I/O 埠的應用而言, 這個延遲時間實際上算不了什麼. 要縮短延遲時間, 你得使用函式 nice 將你在執行中的行程 (process) 設定成高優先權(請參考 `nice(2)` 使用說明文件) 或使用即時排程法 (real-time scheduling) (請看下面).

如果你想獲得比在一般使用者模式 (user-mode) 中執行的行程 (process) 還要精確的時序, 有一些方法可以讓你在使用者模式 (user-mode) 中做到 '即時' 排程的支援. Linux 2.x 版本的核心中有軟體方式的即時排程支援; 詳細的說明請參考 `sched_setscheduler(2)` 使用說明文件. 有一個特殊的核心支援硬體的即時排程; 詳細的資訊請參考網頁 <http://luz.cs.nmt.edu/rtlinux/>

4.1.1 休息中 (Sleeping) : `sleep()` 與 `usleep()`

現在, 讓我們開始較簡單的時序函式呼叫. 想要延遲數秒的時間, 最佳的方法大概 是使用函式 `sleep()` . 想要延遲至少數十毫秒的時間 (10 ms 似乎已是最短的 延遲時間了), 函式 `usleep()` 應該可以使用. 這些函式是讓出 CPU 的使用權 給其他想要執行的行程 (processes) ("自己休息去了"), 所以沒有浪費掉 CPU 的時間. 細節請參考 `sleep(3)` 與 `usleep(3)` 的說明文件.

如果讓出 CPU 的使用權因而使得時間延遲了大約 50 毫秒 (這取決於處理器與機器的速度, 以及系統的負荷), 就浪費掉 CPU 太多的時間, 因爲 Linux 的排程器 (scheduler) (單就 x86 架構而言) 在將控制權發還給你的行程 (process) 之前通常至少要花費 10-30 毫秒的時間. 因此, 短時間的延遲, 使用函式 `usleep(3)` 所得到的延遲結果通常會大於你在參數所指定的值, 大約至少有 10 ms.

4.1.2 `nanosleep()`

在 Linux 2.0.x 一系列的核心發行版本中, 有一個新的系統呼叫 (system call), `nanosleep()` (請參考 `nanosleep(2)` 的說明文件), 他讓你能夠 休息或延遲一個短的時間 (數微秒或更多).

如果延遲的時間 <= 2 ms, 若(且唯若)你執行中的行程 (process) 設定了軟體的即時 排程 (就是使用函式 tt/sched_setscheduler()/), 呼叫函式 `nanosleep()` 時 不是使用一個忙碌迴圈來延遲時間; 就是會像函式 `usleep()` 一樣讓出 CPU 的使用權休息去了.

這個忙碌迴圈使用函式 `udelay()` (一個驅動程式常會用到的核心內部的函式) 來達成, 並且使用 BogoMips 值 (BogoMips 可以準確量測這類忙碌迴圈的速度) 來計算迴圈延遲的時間長度. 其如何動作的細節請參考 `/usr/include/asm/delay.h`).

4.1.3 使用 I/O 埠來延遲時間

另一個延遲數微秒的方法是使用 I/O 埠. 就是從埠位址 0x80 輸入或輸出任何 byte 的資料 (請參考前面) 等待的時間應該幾乎只要 1 微秒這要看你的處理器的型別與速度. 如果要延遲數微秒的時間你可以將

這個動作多做幾次. 在任何標準的機器上輸出資料到該 埠位址應該不會有不良的後果纔對 (而且有些核心的設備驅動程式也在使用他). {in|out}[bw]_p() 等函式就是使用這個方法來產生時間延遲的 (請參考檔案 asm/io.h).

實際上, 一個使用到埠位址範圍爲 0-0x3ff 的 I/O 埠指令幾乎只要 1 微秒的時間, 所以如果你要如此做, 例如, 直接使用並列埠, 只要加上幾個 inb() 函式從該 埠位址範圍讀入 byte 的資料即可.

4.1.4 使用組合語言來延遲時間

如果你知道執行程式所在機器的處理器型別與時鐘速度, 你可以執行某些組合語言指令以便獲得較短的延遲時間 (但是記住, 你在執行中的行程 (process) 隨時會被暫停, 所以有時延遲的時間會比實際長). 如下面的表格所示, 內部處理器的速度決定了所要使用的時鐘周期數; 如, 一個 50 MHz 的處理器 (486DX-50 或 486DX2-50), 一個時鐘周期要花費 1/50000000 秒 (=200 奈秒).

指令	i386 時鐘周期數	i486 時鐘周期數
nop	3	1
xchg %ax,%ax	3	3
or %ax,%ax	2	1
mov %ax,%ax	2	1
add %ax,0	2	1

(對不起, 我不知道 Pentiums 的資料, 或許與 i486 接近吧. 我無法在 i386 的資料上找到只花費一個時鐘周期的指令. 如果能夠就請使用花費一個時鐘周期的指令, 要不然就使用管線技術的新式處理器也是可以縮短時間的.)

上面的表格中指令 nop 與 xchg 應該不會有不良的後果. 指令最後可能會 改變旗號暫存器的內容, 但是這沒關係因爲 gcc 會處理. 指令 nop 是個好的選擇.

想要在你的程式中使用到這些指令, 你得使用 asm("instruction"). 指令的語法就如同上面表格的用法 ; 如果你想要在單一的 asm() 敘述中使用多個指令, 可以使用分號將他們隔開. 例如, asm("nop ; nop ; nop ; nop") 會執行四個 nop 指令, 在 i486 或 Pentium 處理器中會延遲四個時鐘周期 (或是 i386 會延遲 12 個時鐘周期).

gcc 會將 asm() 翻譯成單行組合語言程式碼, 所以不會有呼叫函式的負荷.

在 Intel x86 架構中不可能有比一個時鐘周期還短的時間延遲.

4.1.5 在 **Pentiums** 處理器上使用函式 rdtsc

對於 Pentiums 處理器而言, 你可以使用下面的 C 語言程式碼來取得自從上次重新開機 到現在經過了多少個時鐘周期:

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

你可以詢問參考此值以便延遲你想要的時鐘周期數.

4.2 時間的量測

想要時間精確到一秒鐘, 使用函式 `time()` 或許是最簡單的方法. 想要時間更精確, 函式 `gettimeofday()` 大約可以精確到微秒 (但是如前所述會受到 CPU 排程的影響). 至於 Pentiums 處理器, 使用上面的程式碼片斷就可以精確到一個時鐘周期.

如果你要你執行中的行程 (process) 在一段時間到了之後能夠被通知 (get a signal), 你得使用函式 `setitimer()` 或 `alarm()` . 細節請參考函式的使用說明文件.

5 使用其他程式語言

上面的說明集中在 C 程式語言. 他應該可以直接應用在 C++ 及 Objective C 之上. 至於組合語言部分, 雖然你必須先在 C 語言中呼叫函式 `ioperm()` 或 `iopl()` , 但是之後你就可以直接使用 I/O 埠讀寫指令.

至於其他程式語言, 除非你可以在該程式語言中插入單行組合語言或 C 語言之程式碼或者使用上面所說的系統呼叫, 否則倒不如撰寫一個內含有存取 I/O 埠或延遲時間所必需函式之簡單的 C 原始程式碼或許還比較容易, 編譯之後再與你的程式鏈結. 要不然就是使用前面所說的 `/dev/port` 字元裝置檔案.

6 一些有用的 I/O 埠

本節提供一些常用 I/O 埠的程式撰寫資訊這些都是可以直接拿來用的一般目的 TTL (或 CMOS) 邏輯位準的 I/O 埠.

如果你要按照其原始的設計目的來使用這些或其他常用的I/O 埠 (例如, 控制一般的印表機或數據機), 你應該會使用現成的裝置驅動程式 (他通常被含在核心中) 而不會如本文所說地去撰寫 I/O 埠程式. 本節主要是提供給那些想要將 LCD 顯示器, 步進馬達, 或是其他商業電子產品 連接到 PC 標準 I/O 埠的人.

如果你想要控制大眾市場所販賣的裝置像是掃描器 (已經在市場販賣了一段期間), 去找看看是否有現成的 Linux 裝置驅動程式. 網頁 *Hardware-HOWTO* <http://sunsite.unc.edu/pub/Linux/docs/HOWTO/Hardware-HOWTO> 是個好的參考起點.

至於想要知道更多有關如何連接電子裝置到電腦(以及一般的電子學原理)的相關資訊則網頁 <http://www.hut.fi/Misc/Electronics/> 是個好的資料來源.

6.1 並列埠 (parallel port)

並列埠的基本埠位址 (以下稱之爲 "BASE") 之於 `/dev/lp0` 是 0x3bc , 之於 `/dev/lp1` 是 0x378 , 之於 `/dev/lp2` 是 0x278 . 如果你只是想要控制一些像是一般印表機的動作, 可以參考網頁 *Printing-HOWTO* <http://sunsite.unc.edu/pub/Linux/docs/HOWTO/Printing-HOWTO>.

除了下面即將描述的標準僅輸出 (output-only) 模式, 大多數的並列埠都有 '擴充的' 雙向 (bidirectional) 模式. 至於較新的 ECP/EPP 模式 (以及一般的 IEEE 1284 標準) 埠口的相關資料, 可以參考網頁 <http://www.fapo.com/> 以及 <http://www.senet.com.au/cpeacock/parallel.htm>. 因爲在使用者模式 (user-mode) 中的程式無法使用 IRQs 或 DMA, 想要使用 ECP/EPP 模式你或許得撰寫一個核心的裝置驅動程式; 我想應該有人寫了這類的裝置驅動程式, 但是詳情我並不知道.

埠位址 `BASE+0` (資料埠) 用來控制資料埠的信號位準 (D0 到 D7 分別代表著 bits 0 到 7, 位準狀態: 0 = 低位準 (0 V), 1 = 高位準 (5 V)). 一個寫入資料到該埠的動作會將資料信號位準拴住 (latches) 在埠的腳

位 (pins) 上. 一個將該埠的資料讀出的動作會將上一次以標準僅輸出 (output-only) 模式或擴充的寫入模式所拴住的資料信號位準讀回, 或是以擴充讀出模式 從另外一 個裝置將腳位上的資料信號位準讀回.

位址 `BASE+1` (狀態埠) 是個僅讀 (read-only) 的埠, 會將下面的輸入信號位準讀回:

- Bits 0 和 1 保留不用.
- Bit 2 IRQ 的狀態 (不是個腳位 (pin) , 我不知道他的工作原理)
- Bit 3 ERROR (1=高位準)
- Bit 4 SLCT (1=高位準)
- Bit 5 PE (1=高位準)
- Bit 6 ACK (1=高位準)
- Bit 7 -BUSY (0=高位準)

(我不確定高低位準的電壓狀態.)

埠位址 `BASE+2` (控制埠) 是個僅寫 (write-only) 的埠 (一個將該埠的資料讀出的動作僅會將上一次寫入的資料信號位準讀回), 用來控制下面的狀態信號:

- Bit 0 -STROBE (0=高位準)
- Bit 1 AUTO_FD_XT (1=高位準)
- Bit 2 -INIT (0=高位準)
- Bit 3 SLCT_IN (1=高位準)
- Bit 4 當被設定爲 1 時允許並列埠產生 IRQ 信號 (發生在 ACK 腳位的位準由低變高的瞬間)
- Bit 5 用來控制擴充模式時埠的輸出入方向 (0 = 寫, 1 = 讀), 這是個僅寫 (write-only) 的埠 (一個將該埠的資料讀出的動作對此 bit 一點用處也沒有).
- Bits 6 and 7 保留不用.

(同樣地, 我不確定高低位準的電壓狀態.)

埠的腳位排列 (Pinout) 方式 (該埠是一個 25 隻腳 D 字形外殼 (D-shell) 的母頭連接器) (i=輸入, o=輸出):

```
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
```

IBM 的規格文件上説腳位 1, 14, 16, 和 17 (控制信號的輸出) 採用電晶體的開集極 (open collector) 驅動方式必需使用 4.7 仟歐姆 (kiloohm) 的提升電阻接至 5 V 的電壓 (可流入電流 20 mA, 流出電流 0.55 mA, 高位準的輸出電壓就是 5.0 V 減去提升電阻的電壓). 剩下來的腳位可流入電流 24 mA, 流出電流 15 mA, 高位準的輸出電壓最小 2.4 V. 低位準的輸出電壓二者都是最大 0.5 V. 那些非 IBM 規格的並列埠或許會偏離這個標準. 更多的相關資料請參考網頁 `<http://www.hut.fi/Misc/Electronics/circuits/lptpower.html>`.

最後, 給你一個警告: 留心接地的問題. 我曾經在電腦還是開機的狀況就去連接他因而 弄壞好幾個並列埠. 發生了這種事情你可能會覺得還是不要將並列埠整 合到主機板裡面比較好. (你通常可以拿一片便宜的標準 ‘multi-I/O’ 卡安裝第二個 並列埠; 只要將其他不需要的埠停用, 然後將卡片上並列埠的埠位址設定在 空著的位址即可. 你不需在意並列埠的 IRQ 設定, 因爲通常不會被用到.)

6.2 遊戲 (操縱桿) 埠 (game port)

遊戲埠的埠位址範圍爲 0x200-0x207. 想要控制一般的操縱桿, 有一個核心層次的操縱桿驅動程式, 可參考網址 `<ftp://sunsite.unc.edu/pub/Linux/kernel/patches/>`, 檔名 `joystick-*`.

埠的腳位排列 (Pinout) 方式 (該埠是一個 15 隻腳 D 字形外殼 (D-shell) 的母頭連接器):

- 1,8,9,15: +5 V (電源)
- 4,5,12: 接地
- 2,7,10,14: 分別是 BA1, BA2, BB1, 和 BB2 等數位輸入
- 3,6,11,13: 分別是 AX, AY, BX, 和 BY 等"類比"輸入

+5 V 的腳位似乎通常會被直接連接到主機板的電源線上, 所以他應該能夠提供相當的電力, 這還要看所使用主機板, 電源供給器, 以及遊戲埠的類型.

數位輸入用於操縱桿的按鈕可以讓你連接二個操縱桿的四個按鈕 (操縱桿 A 和 操縱桿 B, 各有二個按鈕) 到遊戲埠也就是數位輸入的四個腳位. 他們應該是一般 TTL 電壓位準的輸入, 你可以直接從狀態埠 (參考下面說明) 讀出他們的位準狀態. 一個實際的操縱桿在按鈕被壓下時會傳回低位準 (0 V) 狀態否則就是高位準 (5V 經由 1 Kohm 的電阻連接到電源腳位) 狀態.

所謂的類比輸入實際是量測到的阻抗值. 遊戲埠有四個單擊多諧振盪器 (one-shot multivibrator) (一個 558 晶片) 連接到四個類比輸入腳位. 每個類比輸入腳位與多諧振盪器的輸出之間連接著一個 2.2 Kohm 的電阻, 而且多諧振盪器的輸出與地之間連接著一個 0.01 uF 的時序電容 (timing capacitor). 一個實際的操縱桿其每個座標 (X 和 Y) 上會有一個可變電阻, 連接在 +5 V 與每個相對的類比輸入腳位之間 (腳位 AX 或 AY 是給操縱桿 A 用的, 而腳位 BX 或 BY 是給操縱桿 B用的).

操作的時候, 多諧振盪器將其輸出設定爲高位準 (5 V) 並且等到時序電容上的電壓達到 3.3 V 之後將相對的輸出設定爲低位準. 因此操縱桿中多諧振盪器輸出的高位準時間周期 與可變電阻的電阻值成正比 (也就是, 操縱桿在相對座標的位置), 如下所示:

R = (t - 24.2) / 0.011,

其中 R 是可變電阻的阻抗值 (ohms) 而 t 是高位準時間周期的長度 (秒).

因此要讀出類比輸入腳位的數值, 首先你得啓動多諧振盪器 (以埠寫入的方式; 請看下面), 然後查詢四個座標的信號狀態(以持續的埠讀出方式)一直到信號狀態由高位準變成低位準, 計算其高位準時間周期的長度. 這個持續查詢的動作花費相當多的 CPU 時間, 而且在一個非即時的多工環境像是 (一般的使用者模式 (user-mode)) Linux, 所得的結果不是非常準確因爲你無法以固定的時間來查詢信號的狀態 (除非你使用核心層次的驅動程式而且你得在你查詢的時候抑制掉中斷的產生, 但是這樣做會浪費更多的 CPU 時間). 如果你知道信號的狀態將會花費一段不短的時間 (數十毫秒) 纔會成爲低位準, 你可以在查詢之前呼叫函式 usleep() 將 CPU 的時間讓給其他想要執行的行程 (processes).

遊戲埠中唯一需要你來存取的埠位址是 0x201 (其他的埠位址不是動作一樣就是沒用). 任何對這個埠位址所做的寫入動作 (不論你寫入什麼) 都會啓動多諧振盪器. 對這個埠位址做讀出動作會取回輸入信號的狀態:

- Bit 0: AX ((1=高位準) 多諧振盪器的輸出狀態)
- Bit 1: AY ((1=高位準) 多諧振盪器的輸出狀態)

- Bit 2: BX ((1=高位準) 多諧振盪器的輸出狀態)
- Bit 3: BY ((1=高位準) 多諧振盪器的輸出狀態)
- Bit 4: BA1 (數位輸入, 1=高位準)
- Bit 5: BA2 (數位輸入, 1=高位準)
- Bit 6: BB1 (數位輸入, 1=高位準)
- Bit 7: BB2 (數位輸入, 1=高位準)

6.3 串列埠 (serial port)

如果你所說的裝置是支援一些像是 RS-232 那類的東西, 你應該可以如你所願地使用串列埠. Linux 所提供的串列埠驅動程式應該能夠應用在任何地方 (你應該不需要直接撰寫串列埠程式, 或是核心的驅動程式); 他相當具有通用性, 所以像是使用非標準的 bps 速率以及其他等等應該不是問題. 請參考 `termios(3)` 說明文件, 串列埠驅動程式原始程式碼 (`linux/drivers/char/serial.c`), 以及網頁 `<http://www.easysw.com/mike/serial/index.html>` 上有更多在 Unix 作業系統撰寫串列埠程式的相關資料.

7 提示

如果你想要有好的 I/O 品值, 你可以在並列埠上自行組裝 ADC 且/或 DAC 晶片 (提示: 電源部分, 可使用遊戲埠上的或將未用到的磁碟電源連接頭接至 機殼之外, 如果你的裝置功率消耗低則可以拿並列埠來充當電源, 不然就是使用外部的電源供給), 或是買 AD/DA 卡片 (大部分較舊型/較低速的產品可由 I/O 埠控制). 或者是 Linux 音效卡驅動程式所支援的便宜音效卡 (速度還相當的快) 上 1 或 2 個不精確, (可能會) 無法歸零的信號通道對你而言就夠了.

使用精確的類比裝置, 不當的接地可能造成類比輸出入信號的誤差. 如果你有這方面的經驗, 你可能會嘗試以光耦合器來隔絕 (電腦與你的裝置之間 所有的 信號) 電子干擾. 試著從電腦上取得光耦合器的電源 (在埠上未用到的信號腳位可以提供足夠的電源) 以求達到最佳的隔絕效果.

如果你現在正在尋找能在 Linux 上使用的印刷電路板設計軟體, 有一個稱爲 Pcb 免費的 X11 應用程式應該能夠勝任, 只要你不要做一些太複雜的事. 許多的 Linux 發行版本 (distributions) 都內含這個程式, 同時他也被放在網址 `<ftp://sunsite.unc.edu/pub/Linux/apps/circuits/>` 上(檔名爲 `pcb-*`).

8 問題排除

Q1.

當我存取 I/O 埠時結果碰到 segmentation faults 這個問題

A1.

不是你的程式沒有 root 權限, 就是因爲某些理由導致函式 `ioperm()` 呼叫失敗. 檢查函式 `ioperm()` 的傳回值. 同時, 檢查你所存取的埠也就是你以 函式 `ioperm()` 所啓用的埠位址 (參考 Q3). 如果你使用的是延遲時間的巨集指令 (`inb_p()`, `outb_p()`, 等等), 記得也要呼叫函式 `ioperm()` 以便存取埠位址 0x80.

Q2.

我無法找到 `in*()`, `out*()` 等函式被定義在何處, 同時 gcc 也抱怨參考到未定義的符號 (undefined references).

A2.

你在編譯程式時沒有打開最佳化選項 (`-O`), 因此 gcc 不能解析 `asm/io.h` 中的巨集指令. 或是你根本就沒有使用 `#include <asm/io.h>`.

Q3.

`out*()` 沒有動作, 或是動作怪怪的.

A3.

檢查參數所放置的次序; 他應該是這樣 `outb(value, port)` , 而不是 MS-DOS 上常用的那樣 `outportb(port, value)`

Q4.

我想要控制一個標準的 RS-232 裝置/連接並列埠的印表機/操縱桿...

A4.

你最好能停止此事而使用現有的驅動程式 (他們存在於 Linux 的核心中或 X 伺服器中或其他的地方) 來達成你的目標. 這些驅動程式通常相當具通用性, 所以就算是有點不標準的裝置, 他們通常都能正常運作. 這些標準 I/O 埠的相關資訊請參考前面說過的文件指引.

9 程式碼範例

這邊是一段用來存取 I/O 埠的簡單的程式碼範例:

```
/*
 * example.c: 一個用來存取 I/O 埠的非常簡單的範例
 *
 * 這個程式碼並沒有什麼用處, 他只是做了埠的寫入, 暫停,
 * 以及埠的讀出幾個動作. 編譯時請使用 `gcc -O2 -o example example.c',
 * 並以 root 的身份執行 `./example'.
 */

#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>

#define BASEPORT 0x378 /* lp1 */

int main()
{
  /* 取得埠位址的存取權限 */
  if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}

  /* 設定埠的輸出資料信號 (D0-7) 全為零 (0) */
  outb(0, BASEPORT);
```

```
  /* 休息一下 (100 ms) */
  usleep(100000);

  /* 從狀態埠 (BASE+1) 讀出資料並顯示結果 */
  printf("status: %d\n", inb(BASEPORT + 1));

  /* 我們不再需要這些埠位址 */
  if (ioperm(BASEPORT, 3, 0)) {perror("ioperm"); exit(1);}

  exit(0);
}

/* 結束 example.c */
```

10 致謝

協助過我的人實在太多無法一一列出, 但還是要跟各位說聲多謝了. 對所有來信協助我的人並沒有一一回覆致上抱歉之意, 並再次謝謝你們的協助.