Linux I/O port programming mini-HOWTO

Autore: Riku Saikkonen <Riku.Saikkonen@hut.fi> v, 28 dicembre 1997

Questo HOWTO descrive la programmazione delle porte I/O e come realizzare delle brevi attese di tempo nei programmi che girano in modo utente, su macchine basate sugli Intel x86. Traduzione di *Fabrizio Stefani* , 15 ottobre 1999.

Contents

1 Introduzione **2**

2 Usare le porte I/O nei programmi C **2**

2.1 Il metodo normale 2

2.2 Un altro metodo: `/dev/port` 3

3 Gli interrupt (IRQ) e l'accesso DMA **3**

4 Temporizzazione ad elevata precisione **4**

4.1 Ritardi 4

4.1.1 Pause: `sleep()` e `usleep()` 4

4.1.2 `nanosleep()` 4

4.1.3 Ritardare tramite l'I/O sulle porte 4

4.1.4 Ritardare usando le istruzioni assembler 5

4.1.5 `rdtsc` per i Pentium 5

4.2 Misurare il tempo 6

5 Altri liguaggi di programmazione **6**

6 Alcune utili porte **6**

6.1 La porta parallela 6

6.2 La porta giochi (joystick) 8

6.3 La porta seriale 9

7 Suggerimenti **9**

8 Risoluzione dei problemi **10**

9 Codice d'esempio **10**

10 Crediti **11**

1 Introduzione

Questo HOWTO descrive la programmazione delle porte I/O e come realizzare delle brevi attese di tempo nei programmi che girano in modo utente, su macchine basate sugli Intel x86. Questo documento è derivato dal piccolissimo IO-Port mini-HOWTO dello stesso autore.

Questo documento è Copyright 1995-1997 di Riku Saikkonen. Vedere il

Linux HOWTO copyright `http://sunsite.unc.edu/pub/Linux/docs/HOWTO/COPYRIGHT` per i dettagli.

Se avete delle correzioni o qualcosa da aggiungere, contattatemi liberamente via e-mail (NdT: in inglese) presso (`Riku.Saikkonen@hut.fi`)...

Cambiamenti rispetto alla versione precedente (30 Mar 1997):

- Chiarite le cose a proposito di `inb_p`/`outb_p` e la porta 0x80.
- Tolte le informazioni su `udelay()`, poiché `nanosleep()` fornisce un modo più pulito di usarlo.
- Convertito in Linuxdoc-SGML e riorganizzato.
- Numerose altre modifiche e aggiunte minori.

2 Usare le porte I/O nei programmi C

2.1 Il metodo normale

Le routine per accedere alle porte I/O sono in `/usr/include/asm/io.h` (o `linux/include/asm-i386/io.h` nella distribuzione del sorgente del kernel). Tali routine sono delle macro inline, quindi è sufficiente usare `#include <asm/io.h>`; non vi serve nessuna libreria aggiuntiva.

A causa di una limitazione in gcc (presente fino alla versione 2.7.2.3) e in egcs (tutte le versioni), *dovete* compilare qualunque sorgente che usa tali routine con l'ottimizzazione abilitata (`gcc -O1` o maggiore), oppure `#define extern` deve essere vuoto prima di mettere `#include <asm/io.h>`.

Per il debugging potete usare `gcc -g -O` (almeno con le ultime versioni di gcc), sebbene l'ottimizzazione possa causare, a volte, un comportamento un po' strano del debugger. Se ciò vi dà noia, mettete le routine che accedono alla porta I/O in un file sorgente separato e compilate con l'ottimizzazione abilitata solo quest'ultimo.

Prima di accedere a qualunque porta, dovete dare al vostro programma il permesso per farlo. Ciò si fa chiamando la funzione `ioperm()` (dichiarata in `unistd.h` e definita nel kernel) da qualche parte all'inizio del vostro programma (prima di qualunque accesso ad una porta I/O). La sintassi è `ioperm(from, num, turn_on)`, dove `from` è il primo numero di porta e `num` il numero di porte consecutive a cui dare l'accesso. Per esempio, `ioperm(0x300, 5, 1)` dà l'accesso alle porte da 0x300 a 0x304 (per un totale di 5 porte). L'ultimo argomento è un valore booleano che specifica se dare (true (1)) o togliere (false (0)) al programma l'accesso alle porte. Potete chiamare più volte `ioperm()` per abilitare più porte non consecutive. Vedere la pagina di manuale di `ioperm()` per i dettagli sulla sintassi.

La chiamata `ioperm()` necessita che il vostro programma abbia privilegi di root; quindi dovete o eseguirlo da root, oppure renderlo suid root. Dopo che avete effettuato la chiamata `ioperm` per abilitare le porte che volete usare, potete rinunciare ai privilegi di root. Alla fine del programma non è necessario abbandonare esplicitamente i privilegi di accesso alle porte con `ioperm(..., 0)`, ciò verrà fatto automaticamente quando il processo esce.

Un `setuid()` fatto ad un utente che non è root non disabilita l'accesso alla porta che gli era stato fornito da `ioperm()`, invece un `fork()` lo disabilita (il processo figlio non acquisisce l'accesso, mentre il padre lo mantiene).

`ioperm()` può fornire l'accesso solo alle porte da 0x000 a 0x3ff; per porte più alte dovete usare `iopl()` (che, con una sola chiamata, vi fornisce l'accesso a tutte le porte). Per fornire al vostro programma l'accesso a *tutte* le porte usate 3 come argomento di livello (cioè `iopl(3)`) (quindi state attenti – accedere alle porte sbagliate può provocare un sacco di sgradevoli cose al vostro computer). Di nuovo, per effettuare la chiamata a `iopl()` dovete avere i privilegi di root. Vedere le pagine di manuale di `iopl(2)` per maggiori dettagli.

Poi, per accedere realmente alle porte... Per leggere (input) un byte (8 bit) da una porta, chiamare `inb(port)`, che restituisce il byte presente all'ingresso. Per scrivere (output) un byte, chiamare `outb(value, port)` (osservate l'ordine dei parametri). Per leggere una word (16 bit) dalle porte `x` e `x+1` (un byte da ognuna per formare una word con l'istruzione assembler `inw`) chiamare `inw(x)`. Per scrivere una word sulle due porte si usa `outw(value, x)`. Se non siete sicuri su quali istruzioni (byte o word) usare per le porte, probabilmente vi servono `inb()` e `outb()` — la maggior parte dei dispositivi sono progettati per gestire l'accesso alle porte a livello di byte. Osservate che tutte le istruzioni per accedere alle porte richiedono, almeno, un microsecondo (circa) per essere eseguite.

Le macro `inb_p()`, `outb_p()`, `inw_p()`, e `outw_p()` funzionano esattamente come le precedenti, ma esse introducono un ritardo, di circa un microsecondo, dopo l'accesso alla porta; potete allungare tale ritardo a circa quattro microsecondi mettendo `#define REALLY_SLOW_IO` prima di `#include <asm/io.h>`. Queste macro, di solito (a meno che inserite `#define SLOW_IO_BY_JUMPING`, che è probabilmente meno preciso), effettuano una scrittura sulla porta 0x80 per ottenere il ritardo; quindi, prima di usarle, dovete dargli l'accesso alla porta 0x80 con `ioperm()` (le scritture fatte sulla porta 0x80 non hanno conseguenze su nessuna parte del sistema). Per ottenere dei ritardi con sistemi più versatili, continuate a leggere.

Nelle raccolte di pagine di manuale per Linux, nelle versioni ragionevolmente recenti, ci sono le pagine di manuale per `ioperm(2)`, `iopl(2)` e per le suddette macro.

2.2 Un altro metodo: `/dev/port`

Un altro modo per accedere alle porte I/O è quello di aprire, in lettura e/o scrittura, con `open()`, il dispositivo a caratteri `/dev/port` (numero primario 1, secondario 4) (le funzioni stdio `f*()` hanno un buffer interno, quindi evitatele). Poi posizionatevi con un `lseek()` sul byte appropriato nel file (posizione 0 del file = porta 0x00, posizione 1 del file = porta 0x01 e così via...) e leggete (`read()`), o scrivete (`write()`), un byte o una word da, o in, esso.

Ovviamente, perché ciò funzioni, il vostro programma avrà bisogno dell'accesso in lettura/scrittura a `/dev/port`. Questo metodo è probabilmente più lento del metodo normale precedentemente descritto, ma esso non necessita né di ottimizzazioni in compilazione né della funzione `ioperm()`. Non vi serve nemmeno di avere l'accesso da root, se impostate per `/dev/port` l'accesso per utenti, o gruppi, non root — ma far questo è una pessima idea, in termini di sicurezza del sistema, poiché è possibile danneggiare il sistema, forse anche ottenere l'accesso a root, usando `/dev/port` per accedere direttamente ai dischi rigidi, alle schede di rete, ecc.

3 Gli interrupt (IRQ) e l'accesso DMA

Non è possibile usare gli IRQ o il DMA direttamente da un processo in modo utente. Dovete scrivere un driver di kernel; vedere *The Linux Kernel Hacker's Guide* `http://www.redhat.com:8080/HyperNews/get/khg.html` per i dettagli e il codice sorgente del kernel per gli esempi.

Inoltre, non è possibile disabilitare gli interrupt da un programma che gira in modo utente.

4 Temporizzazione ad elevata precisione

4.1 Ritardi

Innanzi tutto devo precisare che, a causa della natura multitasking di Linux, non è possibile garantire che un processo che gira in modo utente abbia un preciso controllo delle temporizzazioni. Il vostro processo potrebbe essere sospeso per un tempo che può variare dai, circa, dieci millisecondi, fino ad alcuni secondi (in un sistema molto carico). Comunque, per la maggior parte delle applicazioni che usano le porte I/O, ciò non ha importanza. Per minimizzare tale tempo potreste assegnare al vostro processo un più alto valore di priorità (vedere la pagina di manuale di `nice(2)`), oppure potreste usare uno scheduling in real time (vedere sotto).

Se volete una temporizzazione più precisa di quella disponibile per i processi che girano in modo utente, ci sono delle "forniture" per il supporto dell'elaborazione real time in modo utente. I kernel 2.x di Linux forniscono un supporto per il soft real time; vedere la pagina di manuale di `sched_setscheduler(2)` per i dettagli. C'è un kernel speciale che supporta l'hard real time; per maggiori informazioni vedere

`http://luz.cs.nmt.edu/~rtlinux/` .

4.1.1 Pause: `sleep()` e `usleep()`

Laciatemi incominciare con le più semplici chiamate di funzioni di temporizzazione. Per ritardi di più secondi la scelta migliore è, probabilmente, quella di usare `sleep()`. Per ritardi dell'ordine delle decine di millisecondi (il ritardo minimo sembra essere di circa 10 ms) dovrebbe andar bene `usleep()`. Tali funzioni cedono la CPU agli altri processi (vanno a dormire: "sleep"), in modo che il tempo di CPU non venga sprecato. Per i dettagli vedere le pagine di manuale di `sleep(3)` e `usleep(3)`.

Per ritardi inferiori a, circa, 50 millisecondi (dipendentemente dal carico del sistema e dalla velocità del processore e della macchina) il rilascio della CPU richiede troppo tempo, ciò perché (per l'architettura x86) lo scheduler generalmente lavora almeno dai 10 ai 30 millisecondi prima di restituire il controllo al vostro processo. Per questo motivo, per i piccoli ritardi, `usleep(3)` in genere ritarda un po' più della quantità specificatagli nei parametri, almeno 10 ms circa.

4.1.2 `nanosleep()`

Nei kernel di Linux della serie 2.0.x, c'è una nuova chiamata di sistema, `nanosleep()` (vedere la pagina di manuale di `nanosleep(2)`), che vi permette di dormire o ritardare per brevi periodi di tempo (pochi microsecondi o poco più).

Per ritardi fino a 2 ms, se (e solo se) il vostro processo è impostato per lo scheduling in soft real time, `nanosleep()` usa un ciclo di attesa, altrimenti dorme ("sleep"), proprio come `usleep()`.

Il ciclo di attesa usa `udelay()` (una funzione interna del kernel usata da parecchi driver del kernel) e la lunghezza del ciclo viene calcolata usando il valore di BogoMips (la velocità di questo tipo di cicli di attesa è una delle cose che BogoMips misura accuratamente). Per i dettagli sul funzionamento vedere `/usr/include/asm/delay.h`.

4.1.3 Ritardare tramite l'I/O sulle porte

Un altro modo per realizzare un ritardo di pochi microsecondi è di effettuare delle operazioni di I/O su una porta. La lettura dalla, o la scrittura sulla (come si fa è stato descritto precedentemente), porta 0x80 di un qualsiasi byte impiega quasi esattamente un microsecondo, indipendentemente dal tipo e dalla velocità del processore. Potete ripetere tale operazione più volte per ottenere un'attesa di pochi microsecondi. La

scrittura sulla porta non dovrebbe avere controindicazioni su una qualsiasi macchina standard (e infatti qualche driver del kernel usa questa tecnica). Questo è il metodo normalmente usato da {in|out}[bw]_p() per realizzare il ritardo (vedere asm/io.h).

In effetti una istruzione di I/O su una qualunque delle porte nell'intervallo 0-0x3ff impiega quasi esattamente un microsecondo; quindi se, per esempio, state accedendo direttamente alla porta parallela, potete semplicemente effettuare degli inb() in più su essa per ottenere il ritardo.

4.1.4 Ritardare usando le istruzioni assembler

Se conoscete il tipo e la velocità del processore della macchina su cui girerà il programma, potete sfruttare un codice basato sull'hardware, che usa certe istruzioni assembler, per realizzare dei ritardi molto brevi (ma ricordate, lo scheduler può sospendere il vostro processo in qualsiasi momento, quindi i ritardi potrebbero essere impredicibilmente più lunghi del previsto). Nella tabella sotto, la velocità interna del processore determina il numero di cicli di clock richiesti; cioè, per un processore a 50 MHz (tipo un 486DX-50 o un 486DX2-50) un ciclo di clock dura 1/50.000.000 di secondo (pari a 200 nanosecondi).

Istruzione	cicli di clock su un i386	cicli di clock su un i486
nop	3	1
xchg %ax,%ax	3	3
or %ax,%ax	2	1
mov %ax,%ax	2	1
add %ax,0	2	1

(Mi dispiace ma non conosco quelli dei Pentium, probabilmente sono vicini a quelli del i486. Non ho trovato una istruzione che impieghi un ciclo di clock su un i386. Usate le istruzioni che impiegano un solo ciclo di clock, se possibile, altrimenti le tecniche di pipeling dei moderni processori potrebbero abbreviare i tempi.)

Le istruzioni nop e xchg, indicate nella tabella, non dovrebbero avere effetti collaterali. Le altre potrebbero modificare i flag dei registri, ma ciò non dovrebbe essere un problema poiché gcc dovrebbe accorgersene. nop è una buona scelta.

Per usarle nel vostro programma chiamate asm("istruzione"). La sintassi delle istruzioni è come nella tabella sopra. Se volete mettere più istruzioni in un singolo statement asm() separatele con dei punti e virgola. Ad esempio asm("nop ; nop ; nop ; nop") esegue quattro istruzioni nop, generando un ritardo di quattro cicli di clock sui processori i486 o Pentium (o 12 cicli di clock su un i386).

gcc traduce asm() in codice assembler inline, per cui si risparmiano i tempi per la chiamata di funzione.

Ritardi più brevi di un ciclo di clock sono impossibili con l'architettura Intel x86.

4.1.5 rdtsc per i Pentium

Per i Pentium, potete ottenere il numero di cicli di clock trascorsi dall'ultimo avvio del sistema con il seguente codice C:

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

Potete sondare tale valore per ritardare di quanti cicli di clock vi pare.

4.2 Misurare il tempo

Per tempi della precisione dell'ordine del secondo è probabilmente più facile usare `time()`. Per tempi più precisi, `gettimeofday()` è preciso fino a circa un microsecondo (ma vedete quanto già detto riguardo lo scheduling). Con i Pentium, il frammento di codice sopra (`rdtsc`) ha una precisione pari a un ciclo di clock.

Se volete che il vostro processo riceva un segnale dopo un certo quanto di tempo, usate `setitimer()` o `alarm()`. Per i dettagli vedere le pagine di manuale delle suddette funzioni.

5 Altri liguaggi di programmazione

La descrizione precedente era relativa specificatamente al linguaggio C. Dovrebbe valere inalterata per il C++ e l'Objective C. In assembler, dovete effettuare la chiamata a `ioperm()` o `iopl()` come in C, ma dopo di ciò potete usare direttamente le istruzioni di lettura/scrittura per l'I/O nella porta.

In altri linguaggi, a meno che possiate inserire nel programma codice assembler o C inline, oppure possiate usare le chiamate di sistema precedentemente menzionate, è probabilmente più facile scrivere un semplice file sorgente C contenente le funzioni per l'accesso in I/O alle porte o i ritardi che vi servono, e compilarlo e linkarlo con il resto del programma. Oppure usare `/dev/port` come descritto precedentemente.

6 Alcune utili porte

Vengono ora date delle informazioni per la programmazione delle porte più comuni che possono essere usate per l'I/O delle logiche TTL (o CMOS) general purpose.

Se volete usare queste o altre porte per il loro scopo originale (cioè controllare una normale stampante o un modem), fareste meglio ad usare i driver esistenti (che, di solito, sono inclusi nel kernel) piuttosto che programmare direttamente le porte come descritto in questo HOWTO. Questa sezione è indirizzata a quelli che vogliono connettere alle porte standard di un PC degli schermi LCD, dei motori passo passo, o altri componenti specifici.

Se volete controllare un dispositivo di largo uso, come uno scanner (che è sul mercato già da un po'), cercate se c'è già un driver di Linux che lo riconosce. L' *Hardware-HOWTO* `http://sunsite.unc.edu/pub/Linux/docs/HOWTO/Hardware-HOWTO` è un buon posto da cui iniziare la ricerca.

`http://www.hut.fi/Misc/Electronics/` è una buona fonte per informazioni sulla connessione di dispositivi ai computer (e sugli apparecchi elettronici in generale).

6.1 La porta parallela

L'indirizzo base della porta parallela (detto "`BASE`" nel seguito) è 0x3bc per `/dev/lp0`, 0x378 per `/dev/lp1` e 0x278 per `/dev/lp2`. Se volete solo controllare un qualcosa che si comporta come una normale stampante, vedete il *Printing-HOWTO* `http://sunsite.unc.edu/pub/Linux/docs/HOWTO/Printing-HOWTO` .

Nella maggior parte delle porte parallele, oltre al modo standard di sola scrittura descritto qui sotto, esiste un modo bidirezionale 'esteso'. Per maggiori informazioni su tale argomento e sui nuovi modi ECP/EPP, vedere `http://www.fapo.com/` e `http://www.senet.com.au/~cpeacock/parallel.htm` . Ricordate che poiché non è possibile usare gli IRQ o il DMA in un programma che gira in modo utente, per usare ECP/EPP dovrete probabilmente scrivere un driver kernel. Credo che qualcuno stia già scrivendo un tale driver, ma non conosco i dettagli della cosa.

La porta `BASE+0` (porta dati) controlla i segnali dei dati della porta (da D0 a D7 per i bit da 0 a 7, rispettivamente; stati: 0 = basso (0 V), 1 = alto (5 V)). Una scrittura in tale porta fissa i dati sui pin. Una lettura restituisce i dati che sono stati scritti per ultimi, in modo standard (oppure esteso), oppure restituisce i dati provenienti dai pin di un altro dispositivo che lavora in modo reale esteso.

La porta `BASE+1` (porta di Stato) è di sola lettura e restituisce lo stato dei seguenti segnali d'ingresso:

- Bit 0 e 1, sono riservati.
- Bit 2 stato dell'IRQ (non è un pin, non so come funziona)
- Bit 3 ERROR (1 = alto)
- Bit 4 SLCT (1 = alto)
- Bit 5 PE (1 = alto)
- Bit 6 ACK (1 = alto)
- Bit 7 -BUSY (0 = alto)

(Non sono sicuro degli stati alto e basso)

La porta `BASE+2` (porta di Controllo) è di sola scrittura (una lettura restituisce gli ultimi dati scritti) e controlla i seguenti segnali di stato:

- Bit 0 -STROBE (0 = alto)
- Bit 1 AUTO_FD_XT (1 = alto)
- Bit 2 -INIT (0 = alto)
- Bit 3 SLCT_IN (1 = alto)
- Bit 4, quando impostato ad 1, abilita l'IRQ della porta parallela (che si verifica nella transizione da basso ad alto di ACK)
- Bit 5 controlla la direzione del modo esteso (0 = scrittura, 1 = lettura) ed è assolutamente di sola scrittura (una lettura di questo bit non restituisce nulla di utile).
- Bit 6 e 7, sono riservati.

(Di nuovo, non sono sicuro degli stati alto e basso)

Configurazione dei pin (connettore a "D" femmina a 25-pin sulla porta) (i = input, ingresso; o = output, uscita):

```
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 (Massa)
```

Le specifiche IBM dicono che i pin 1, 14, 16 e 17 (le uscite di controllo) hanno i driver dei collettori aperti connessi a 5 V attraverso resistori da 4,7 Kohm (pozzo 20 mA, fonte 0,55 mA, uscita a livello alto pari a 0,5 V meno il pullup). I rimanenti pin hanno il pozzo a 24 mA, la fonte a 15 mA, e la loro uscita a livello alto è di 2,4 V (minimo). Per entrambi, lo stato basso è di 0,5 V (massimo). Le porte parallele non IBM probabilmente si discostano da questo standard. Per maggiori informazioni a tal riguardo vedere `http://www.hut.fi/Misc/Electronics/circuits/lptpower.html` .

In ultimo un avvertimento: state attenti con i collegamenti a massa. Io ho rotto diverse porte parallele collegandoci qualcosa mentre il computer era acceso. Per giochetti del genere sarebbe buona cosa usare una porta parallela che non sia integrata sulla piastra madre. (Di solito è possibile ottenere una seconda porta parallela, per la propria macchina, tramite una economica e standard scheda 'multi-I/O'; semplicemente disabilitate le porte di cui non avete bisogno e impostate l'indirizzo I/O, della porta parallela sulla scheda, ad un indirizzo libero. Non dovete preoccuparvi dell'IRQ della porta parallela visto che, normalmente, non viene usato.)

6.2 La porta giochi (joystick)

La porta giochi è situata agli indirizzi 0x200-0x207. Per controllare i normali joystick c'è un apposito driver a livello di kernel, vedere

`ftp://sunsite.unc.edu/pub/Linux/kernel/patches/` , nome del file `joystick-*`.

Configurazione dei pin (connettore a "D" femmina a 15 pin sulla porta):

- 1, 8, 9, 15: +5 V (alimentazione)
- 4, 5, 12: Massa
- 2, 7, 10, 14: ingressi digitali BA1, BA2, BB1 e BB2, rispettivamente
- 3, 6, 11, 13: ingressi "analogici" AX, AY, BX e BY, rispettivamente

I pin +5 V sembra che siano spesso collegati direttamente alle linee di alimentazione sulla piastra madre, quindi dovrebbero poter fornire un bel po' di potenza, a seconda della piastra madre, dell'alimentatore e della porta giochi.

Gli ingressi digitali sono usati per i pulsanti dei due joystick (joystick A e joystick B, con due pulsanti ciascuno) collegabili alla porta. Dovrebbero usare i normali livelli d'ingresso TTL e potete leggerne lo stato direttamente dalla porta di stato (vedere sotto). Quando il pulsante è premuto, il joystick restituisce uno stato basso (0 V), altrimenti restituisce uno stato alto (i 5 V del pin dell'alimentazione attraverso un resistore di un Kohm).

I cosìdetti ingressi analogici in effetti misurano una resistenza. La porta giochi ha un quadruplo multivibratore monostabile (un integrato 558) collegato ai quattro ingressi. Ad ogni ingresso, fra il pin di ingresso e l'uscita del multivibratore, c'è un resistore da 2,2 Kohm e, fra l'uscita del multivibratore e la massa, c'è un condensatore di temporizzazione pari a 0,01 uF. Un joystick (in senso fisico) ha un potenziometro per ogni asse (X e Y), connesso fra +5 V e l'appropriato pin d'ingresso (AX o AY per il joystick A, oppure BX o BY per il joystick B).

Il multivibratore, quando attivato, imposta alte (5 V) le sue linee di uscita ed aspetta che ogni condensatore di temporizzazione raggiunga i 3,3 V prima di abbassare le rispettive linee di uscita. Così facendo, la durata dello stato alto del multivibratore è proporzionale alla resistenza del potenziometro nel joystick (cioè alla posizione della leva sull'asse corrispondente), secondo la relazione:

$$R = (t - 24{,}2) / 0{,}011$$

dove R è la resistenza (in ohm) del potenziometro e t la durata dello stato alto (in secondi).

Quindi, per leggere gli ingressi analogici, dovete prima attivare il multivibratore (con una scrittura sulla porta; vedere sotto), poi controllare (con letture ripetute della porta) lo stato dei quattro assi finché non scendono dallo stato alto a quello basso e quindi misurare la durata del loro stato alto. Tale controllo richiede abbastanza tempo di CPU e, su di un sistema multitasking non in real time come Linux (in modo utente

normale), il risultato non è molto preciso perché non potete controllare costantemente la porta (a meno che usiate un driver a livello di kernel e disabilitiate gli interrupt durante il controllo; ma così si spreca ancor più tempo di CPU). Se sapete che il segnale impiegherà parecchio tempo (decine di ms) per tornare basso, potete chiamare usleep() prima di cominciare il controllo, dando così quel tempo di CPU ad altri processi.

La sola porta di I/O a cui vi serve di accedere è la porta 0x201 (le altre porte o si comportano identicamente, o non fanno nulla). Qualsiasi scrittura su questa porta (non importa cosa scrivete) attiva il multivibratore. Una lettura da questa porta restituisce lo stato dei segnali di ingresso:

- Bit 0: AX (stato dell'uscita del multivibratore (1 = alto))
- Bit 1: AY (stato dell'uscita del multivibratore (1 = alto))
- Bit 2: BX (stato dell'uscita del multivibratore (1 = alto))
- Bit 3: BY (stato dell'uscita del multivibratore (1 = alto))
- Bit 4: BA1 (ingresso digitale, 1 = alto)
- Bit 5: BA2 (ingresso digitale, 1 = alto)
- Bit 6: BB1 (ingresso digitale, 1 = alto)
- Bit 7: BB2 (ingresso digitale, 1 = alto)

6.3 La porta seriale

Se il dispositivo che vi interessa supporta qualcosa che somiglia alla RS-232, allora dovreste poter usare la porta seriale per comunicare con esso. Il driver di Linux per le porte seriali dovrebbe andar bene per quasi tutte le applicazioni (non dovete programmare direttamente la porta seriale, per farlo, probabilmente, dovreste scrivere un driver kernel); è piuttosto versatile, quindi usando velocità di trasmissione (bps) non standard, o cose del genere, non dovrebbero esserci problemi.

Per maggiori informazioni sulla programmazione delle porte seriali sui sistemi Unix, vedere la pagina di manuale di `termios(3)`, il codice sorgente del driver per la porta seriale (`linux/drivers/char/serial.c`) e `http://www.easysw.com/~mike/serial/index.html` .

7 Suggerimenti

Se volete un buon I/O analogico, potete collegare dei chip ADC e/o DAC alla porta parallela (suggerimento: per l'alimentazione usate il connettore della porta giochi o un connettore di alimentazione per i dischi ancora libero che va cablato fino all'esterno del computer, a meno che non abbiate un dispositivo a basso consumo e possiate usare la porta parallela stessa per l'alimentazione, o una fonte di alimentazione esterna), o comprare una scheda AD/DA (la maggior parte di quelle più vecchie, più lente, vengono controllate dalle porte I/O). Oppure, se vi accontentate di 1 o 2 canali, non vi dà fastidio l'imprecisione e (probabilmente) uno spostamento di fuori zero, dovrebbe bastarvi (ed è davvero veloce) una scheda audio economica che sia supportata dai driver audio di Linux.

Con dispositivi analogici precisi, una cattiva messa a terra può generare degli errori negli input o output analogici. Se vi capita qualcosa del genere, potreste provare ad isolare elettricamente il vostro dispositivo dal computer, usando degli accoppiatori ottici (su *tutti* i segnali tra il computer ed il dispositivo). Per ottenere un migliore isolamento provate a prendere l'alimentazione per gli accoppiatori dal computer (i segnali non usati sulla porta potrebbero fornire potenza sufficiente).

Se state cercando un programma (per Linux) per il progetto di circuiti stampati, c'è un'applicazione per X11, chiamata Pcb, che funziona piuttosto bene, almeno se non dovete fare niente di molto complesso. È inclusa in parecchie distribuzioni di Linux ed è disponibile in `ftp://sunsite.unc.edu/pub/Linux/apps/circuits/` (nomefile `pcb-*`).

8 Risoluzione dei problemi

D1.

Quando accedo alle porte ottengo "segmentation faults" (errori di segmentazione).

R1.

Il tuo programma non ha i privilegi di root, oppure la chiamata `ioperm()` è fallita per qualche altro motivo. Controlla il valore restituito da `ioperm()`. Inoltre, assicurati di stare accedendo proprio alle porte che hai abilitato con `ioperm()` (vedi D3). Se stai usando le macro di ritardo (`inb_p()`, `outb_p()`, e via dicendo), ricordati di effettuare una chiamata a `ioperm()` per ottenere l'accesso anche alla porta 0x80.

D2.

Non riesco a trovare le funzioni `in*()` e `out*()` definite ovunque, e gcc si lamenta per dei riferimenti non definiti (undefined references).

R2.

Non hai compilato con l'ottimizzazione abilitata (`-O`) e quindi gcc non riesce a risolvere le macro contenute in `asm/io.h`. Oppure non hai messo affatto `#include <asm/io.h>`

D3.

`out*()` non fa nulla, o fa qualcosa di strano.

R3.

Controlla l'ordine dei parametri; deve essere `outb(valore, porta)` e non `outportb(porta, valore)` come è in MS-DOS.

D4.

Voglio controllare un dispositivo standard RS-232/una stampante parallela/un joystick...

R4.

Probabilmente ti conviene usare i driver esistenti (nel kernel di Linux, o in un server X, o da qualche altra parte). I driver generalmente sono abbastanza versatili, tanto che anche i dispositivi non standard di solito ci funzionano. Vedere le informazioni precedentemente date riguardo le porte standard per indicazioni sulle relative documentazioni.

9 Codice d'esempio

Ecco un pezzo di un semplice codice d'esempio per l'accesso alla porta I/O:

```
/*
 * example.c: un semplicissimo esempio di I/O su porta
 *
 * Questo codice non fa nulla di utile, solo una scrittura sulla
 * porta, una pausa e una lettura dalla porta. Compilatelo con
```

```
 * ‘gcc -O2 -o example example.c’ ed eseguitelo da root con ‘./example’.
 */

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

#define BASEPORT 0x378 /* lp1 */

int main()
{
  /* Richiede l’accesso alle porte */
  if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}

  /* Imposta bassi (0) i segnali di dati (D0-7) della porta */
  outb(0, BASEPORT);

  /* Va in pausa (dorme) per un po’ (100 ms) */
  usleep(100000);

  /* Legge dalla porta lo stato (BASE+1) e mostra il risultato */
  printf("stato: %d\n", inb(BASEPORT + 1));

  /* La porta non ci serve piu’ */
  if (ioperm(BASEPORT, 3, 0)) {perror("ioperm"); exit(1);}

  exit(0);
}

/* fine dell’esempio example.c */
```

10 Crediti

Ha contribuito troppa gente perché io possa elencarla, ma grazie tante, a tutti. Non ho risposto a tutti i contributi che mi sono giunti; me ne scuso, e grazie ancora per l’aiuto.