Linux I/O port programming mini-HOWTO

Autore: Riku Saikkonen <Riku.Saikkonen@hut.fi> v3.0, 13/12/2000

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 a cura di *Fabrizio Stefani* , 8 giugno 2003, e revisione a cura di *Sandro Cardelli* , 4 luglio 2005.

Indice

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.

Per correzioni o eventuali aggiunte sono raggiungibile via e-mail (`Riku.Saikkonen@hut.fi`)...

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 serve nessuna libreria aggiuntiva.

A causa di una limitazione in gcc (presente in tutte le versioni di cui sono al corrente, compreso egcs), *bisogna* compilare tutti i sorgenti che usano tali routine con l'ottimizzazione abilitata (`gcc -O1` o maggiore), oppure usare `#define extern static` prima di `#include <asm/io.h>` (ricordarsi poi di mettere `#undef extern`).

Per il debugging si può usare `gcc -g -O` (almeno con le ultime versioni di gcc), sebbene l'ottimizzazione possa causare, a volte, un comportamento un pò strano del debugger. È possibile evitare tale inconveniente mettendo le routine che accedono alla porta I/O in un file sorgente separato e compilare solo quest'ultimo con l'ottimizzazione abilitata.

2.1.1 Permessi

Prima di accedere ad una qualsiasi porta, bisogna dare al 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 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. È possibile chiamare più volte `ioperm()` per abilitare più porte non consecutive. Vedere la pagina di man di `ioperm(2)` per i dettagli sulla sintassi.

La chiamata `ioperm()` necessita che il programma abbia i privilegi di root; quindi bisogna eseguirlo da utente root, oppure renderlo suid root. Dopo la chiamata `ioperm` per abilitare le porte che si vogliono usare, si può 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 termina.

Un `setuid()` ad un utente non root non disabilita l'accesso alla porta fornito da `ioperm()`, mentre un `fork()` lo disabilita (il processo figlio non ottiene l'accesso, il genitore invece lo mantiene).

`ioperm()` può fornire l'accesso solo alle porte da 0x000 a 0x3ff; per porte più alte bisogna usare `iopl()` (che, con una sola chiamata, fornisce l'accesso a tutte le porte). Per fornire al programma l'accesso a *tutte* le porte I/O usare 3 come argomento di livello (cioè `iopl(3)`) (quindi attenzione: accedere alle porte sbagliate può provocare un sacco di sgradevoli cose al computer). Di nuovo, per effettuare la chiamata a `iopl()` sono necessari i privilegi di root. Per maggiori dettagli vedere le pagine di man di `iopl(2)`.

2.1.2 Accedere 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(valore, porta)` (notare 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(valore, x)`. Se si hanno dubbi su quali istruzioni (byte o word) usare per le porte, probabilmente servono `inb()` e `outb()` — la maggior parte dei dispositivi sono progettati per gestire l'accesso alle porte a livello di byte. Notare 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 introducono un ritardo, di circa un microsecondo, dopo l'accesso alla porta. Si può allungare tale ritardo a circa quattro microsecondi mettendo `#define REALLY_SLOW_IO` prima di `#include <asm/io.h>`. Queste macro, di solito (a meno non si usi `#define SLOW_IO_BY_JUMPING`, che probabilmente è meno preciso), effettuano una scrittura sulla porta 0x80 per ottenere il ritardo e quindi, prima di usarle, bisogna dargli l'accesso alla porta 0x80 con `ioperm()` (le scritture fatte sulla porta 0x80 non hanno conseguenze su nessuna parte del sistema). Più avanti è spiegato come ottenere dei ritardi con sistemi più versatili.

Nelle raccolte di pagine di man per Linux, nelle versioni ragionevolmente recenti, ci sono le pagine di man di `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 e sono quindi da evitare). Poi posizionarsi con `lseek()` sul byte appropriato nel file (posizione 0 del file = porta 0x00, posizione 1 del file = porta 0x01 e così via...) e leggere (`read()`), o scrivere (`write()`), un byte o una word da, o in, esso.

Ovviamente, perché ciò funzioni, il programma avrà bisogno dell'accesso in lettura/scrittura a `/dev/port`. Questo metodo è probabilmente più lento del metodo normale precedentemente descritto, ma non ha bisogno né di ottimizzazioni in compilazione né della funzione `ioperm()`. Non serve nemmeno l'accesso da root, se si fornisce al gruppo o agli utenti non root l'accesso a `/dev/port` – ma, in termini di sicurezza del sistema, far questo è una pessima idea perché è possibile danneggiare il sistema, e forse anche ottenere l'accesso a root, usando `/dev/port` per accedere direttamente agli hard disk, alle schede di rete, ecc.

Non è possibile usare `select(2)` oppure `poll(2)` per leggere `/dev/port`, perché l'hardware non ha la capacità di notificare alla CPU il cambiamento del valore in una porta d'ingresso.

3 Gli interrupt (IRQ) e l'accesso DMA

Non è possibile usare gli IRQ o il DMA direttamente da un processo in modo utente, bisogna scrivere un driver per il 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.

È possibile disabilitare gli interrupt da un programma in modo utente, ma può essere pericoloso (anche i driver del kernel lo fanno per il tempo più breve possibile). Dopo la chiamata `iopl(3)` è possibile disabilitare gli interrupt con una semplice chiamata `asm(cli);` e poi riabilitarli con `asm(sti);`.

4 Temporizzazione ad elevata precisione

4.1 Ritardi

Innanzi tutto è bene 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. Un 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 è possibile assegnare al processo un più elevato valore di priorità (vedere la pagina di man di `nice(2)`), oppure si può usare uno scheduling in real time (vedere sotto).

Per temporizzazioni più precise di quelle disponibili 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; per i dettagli vedere la pagina di man di `sched_setscheduler(2)`. 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()`

Incominciamo 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()`. Queste 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 man 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 le architetture x86) lo scheduler generalmente lavora, almeno, dai 10 ai 30 millisecondi prima di restituire il controllo ad un 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 Linux della serie 2.0.x, c'è una nuova chiamata di sistema, `nanosleep()` (vedere la pagina di man di `nanosleep(2)`), che permette di dormire o ritardare per brevi periodi di tempo (pochi microsecondi o più).

Per ritardi fino a 2 ms, se (e solo se) il processo è impostato per lo scheduling in soft real time (usando `sched_setscheduler()`), `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 kernel driver) 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 byte impiega quasi esattamente un microsecondo, indipendentemente dal tipo e dalla velocità del processore. È possibile ripetere tale operazione più volte per ottenere un'attesa di alcuni microsecondi. La scrittura sulla porta non dovrebbe avere controindicazioni su nessuna 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, si sta accedendo direttamente alla porta parallela, si possono semplicemente effettuare degli `inb()` in più per ottenere il ritardo.

4.1.4 Ritardare usando le istruzioni assembler

Se si conosce il tipo e la velocità del processore della macchina su cui girerà il programma, è possibile sfruttare del codice basato sull'hardware, che usa certe istruzioni assembler, per realizzare dei ritardi molto brevi (ma ricordare che lo scheduler può sospendere il processo in qualsiasi momento, quindi i ritardi potrebbero essere impredicibilmente più lunghi del previsto). Nella tabella seguente la velocità interna del processore determina il numero di cicli di clock richiesti. Ad esempio, 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      cicli di clock
                   su un i386          su un i486
xchg %bx,%bx           3                   3
nop                    3                   1
or %ax,%ax             2                   1
mov %ax,%ax            2                   1
add %ax,0              2                   1
```

I cicli di clock dei Pentium dovrebbero essere gli stessi degli i486, ma nei Pentium Pro/II `add %ax, 0` potrebbe richiedere solo 1/2 ciclo di clock. Si può rimediare usando un'altra istruzione (a causa dell'esecuzione fuori ordine non è necessario che tale istruzione sia quella immediatamente successiva nel codice).

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. Per realizzare istruzioni di ritardo `xchg %bx, %bx` è una buona scelta.

Per usarle bisogna chiamare `asm("istruzione")`. La sintassi delle istruzioni è come nella tabella precedente. Se si vogliono mettere più istruzioni in un singolo `asm()` bisogna separarle 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 le architetture Intel x86.

4.1.5 `rdtsc` per i Pentium

Con i Pentium è possibile ottenere il numero di cicli di clock trascorsi dall'ultimo riavvio del sistema con il seguente codice C (che esegue l'istruzione RDTSC):

```
        extern __inline__ unsigned long long int rdtsc()
```

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

Si può sondare tale valore per ritardare del numero di cicli di clock desiderato.

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 vedere 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 si desidera che un processo riceva un segnale dopo un certo quanto di tempo, usare `setitimer()` o `alarm()`. Per i dettagli vedere le pagine di man delle suddette funzioni.

5 Altri linguaggi di programmazione

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

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

6 Alcune utili porte

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

Se si vogliono usare queste o altre porte per il loro scopo originale (cioè controllare una normale stampante o un modem), sarebbe meglio 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 si desidera controllare un dispositivo di largo uso, come uno scanner (che è sul mercato già da un pò), 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 di informazioni sulla connessione di dispositivi ai computer (e sull'elettronica 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`. Per controllare qualcosa che si comporta come una normale stampante, vedere 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 di seguito, 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` . Ricordare che poiché non è possibile usare gli IRQ o il DMA in un programma che gira in modo utente, per usare ECP/EPP probabilmente sarà necessario scrivere un driver per il 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 su 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 modalità di lettura estesa.

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

- Bit 0 e 1, riservati.
- Bit 2 stato dell'IRQ (non è un pin e 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)

La porta `BASE+2` (porta di Controllo) è di sola scrittura (una lettura restituisce l'ultimo dato scritto) e controlla i seguenti segnali di stato:

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

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
```

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: attenzione con i collegamenti a massa. Personalmente 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; basta disabilitare le porte di cui non si ha bisogno e impostare l'indirizzo I/O della porta parallela sulla scheda ad un indirizzo libero. Non è necessario conoscere l'IRQ della porta parallela se non lo si usa.)

6.2 La porta giochi (joystick)

La porta giochi è situata agli indirizzi 0x200-0x207. Per controllare i normali joystick è probabilmente meglio usare i driver distribuiti con il kernel di Linux.

Configurazione dei pin (connettore a D femmina a 15 pin):

- 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 scheda madre, quindi dovrebbero poter fornire un bel pò di potenza, a seconda della scheda 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 ed è possibile 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 1 kohm).

I cosìdetti ingressi analogici in effetti misurano una resistenza. La porta giochi ha un quadruplo multivibratore monostabile (un chip 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. Il joystick è dotato di 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 sarà 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 microsecondi).

Quindi, per leggere gli ingressi analogici, bisogna 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 è possibile controllare costantemente la porta (a meno che si usi un driver a livello di kernel e si disabilitino gli interrupt durante il controllo; ma così si spreca ancor più tempo di CPU). Se si sa che il segnale impiegherà parecchio tempo (decine di ms) per tornare basso, si può chiamare usleep() prima di cominciare il controllo, regalando così quel tempo di CPU ad altri processi.

La sola porta di I/O a cui serve di accedere è la porta 0x201 (le altre porte o si comportano allo stesso modo, o non fanno nulla). Qualsiasi scrittura su questa porta (non importa cosa si scrive) 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 d'interesse supporta qualcosa che somiglia alla RS-232, dovrebbe essere possibile usare la porta seriale per comunicare con esso. Il driver di Linux per le porte seriali dovrebbe andar bene per la maggior parte delle applicazioni (non è necessario programmare direttamente la porta seriale, per farlo, probabilmente, bisognerebbe scrivere un driver per il kernel); è piuttosto versatile e quindi, usando velocità di trasmissione (b/s) 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 man di `termios(3)`, il codice sorgente del driver per la porta seriale (`linux/drivers/char/serial.c`) e `http://www.easysw.com/~mike/serial/` .

7 Suggerimenti

Se si desidera un buon I/O analogico, collegare dei chip ADC e/o DAC alla porta parallela (suggerimento: per l'alimentazione usare 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 si abbia un dispositivo a basso consumo e si possa 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, e più lente, vengono controllate dalle porte I/O). Oppure, se ci si accontenta di 1 o 2 canali, se non dà fastidio l'imprecisione e (probabilmente) uno spostamento di fuori zero, dovrebbe bastare (e risulta veramente 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 capita qualcosa del genere, provare ad isolare elettricamente il dispositivo dal computer, usando degli accoppiatori ottici (su *tutti* i segnali tra il computer ed il dispositivo). Per ottenere un migliore isolamento provare a prendere l'alimentazione per gli accoppiatori dal computer (i segnali non usati sulla porta potrebbero fornire potenza sufficiente).

Se si sta cercando un programma per Linux per il progetto di circuiti stampati, c'è un'applicazione per X11, chiamata Pcb, che funziona piuttosto bene, almeno per applicazioni non molto complesse. È inclusa in parecchie distribuzioni Linux ed è disponibile in `ftp://sunsite.unc.edu/pub/Linux/apps/circuits/` (nome del file: `pcb-*`).

8 Risoluzione dei problemi

D1.

Quando accedo alle porte ottengo segmentation faults.

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 hai dimenticato di mettere `#include <asm/io.h>`

D3.

`out*()` non fà nulla, o fà 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 da far funzionare anche i dispositivi non standard, di solito. Vedere le informazioni precedentemente date riguardo le porte standard per indicazioni sulla documentazione.

9 Codice d'esempio

Ecco un esempio di codice 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. Compilarlo 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 pò (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 serve più */
  if (ioperm(BASEPORT, 3, 0)) {perror("ioperm"); exit(1);}

  exit(0);
}

/* fine dell’esempio example.c */
```

10 Ringraziamenti

Ha contribuito troppa gente perché 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.