Linux I/O port programmering mini-HOWTO

Av: Riku Saikkonen <Riku.Saikkonen@hut.fi> Översättning till svenska: Sven Wilhelmsson, <sven.wilhelmsson@swipnet.se>
v ,28 December 1997 , Översatt: 10 Augusti 1998

I detta HOWTO dokument beskrivs hur man programmerar I/O portar samt hur man fördröjer eller mäter korta tidsintervall i *user-mode* Linux program för Intel x86 arkitekturen.

Innehåll

1 Inledning

I detta HOWTO dokument beskrivs hur man når I/O-portar och hur man fördröjer korta tidsintervall i *user-mode* Linux program som körs på Intel x86 arkitekturen. Detta dokumentet är en uppföljare till den mycket lilla *IO-Port mini-HOWTO* från samma författare.

If you have corrections or something to add, feel free to e-mail me (`Riku.Saikkonen@hut.fi`)...

Ändringar från tidigare publicerad version (Mar 30 1997):

- Förklaringar angående `inb_p`/outb_p/ och port 0x80.
- Tagit bort information om `udelay()`, eftersom `nanosleep()` medger ett renare sätt att göra samma sak.
- Konverterat till Linuxdoc-SGML, och ändrat om något.
- Många mindre tillägg och ändringar.

2 Att använda I/O portar i C-program

2.1 Det vanliga sättet

Rutiner för att nå I/O-portar finns i`/usr/include/asm/io.h` (eller `linux/include/asm-i386/io.h` i 'the kernel source distribution'). Rutinerna där är inline makron, så det räcker att göra `#include <asm/io.h>`, inga ytterligare bibliotek behövs.

Beroende på brister i gcc (åtminstone i 2.7.2.3 och lägre) och i egcs (alla versioner), måste du kompilera källkod som använder dessa rutiner med optimeringsflaggan på (`gcc -O1` eller högre), eller alternativt `#define extern` till ingenting, (dvs. `#define extern` på en i övrigt blank rad) innan du gör `#include <asm/io.h>`.

Om du vill avlusa, 'debug', kan du använda `gcc -g -O` (åtminstone med moderna versioner av gcc), även om optimeringen ibland gör att debuggern beter sig lite underligt. Om detta besvärar dig, kan du lägga de rutiner som anropar I/O-portarna i separata filer och kompilera endast dem med optimeringsflaggan på.

Innan du anropar en port, måste du ge ditt program tillstånd till detta. Detta gör man genom att anropa `ioperm()` funktionen (som finns deklarerad i `unistd.h`, och definierad i 'kernel') någonstans i början av ditt program, innan någon I/O-port anropas. Syntaxen är `ioperm(from, num, turn_on)`, där `from` är den första portadressen som ska ges tillstånd och `num` är antalet konsekutiva adresser. Till exempel, `ioperm(0x300, 5, 1)` ger tillstånd till portarna 0x300 till 0x304 (totalt 5 portar). Det sista argumentet är en bool som specificerar om du vill ge accesstillstånd (true(1)) eller ta bort tillståndet (false(0)). Du kan använda `ioperm()` upprepade gånger för att ge tillstånd till ickekonsekutiva portadresser. Se `ioperm(2)` manualen.

`ioperm()` anropet kräver att ditt program har rootprivilegier. Det krävs att du antingen kör som root, eller gör setuid root. Du kan släppa rootprivilegierna så snart du har anropat `ioperm()`. Det är inte nödvändigt att explicit släppa dina accessrättigheter med `ioperm(..., 0)` mot slutet av ditt program. Detta sker automatiskt när processen avslutas.

`Om du gör setuid()` till en *non-root user* förstörs inte de accessrättigheter som är redan givna av `ioperm()`, men `fork()` förstör dem (*child* processen får inga rättigheter, men *parent* behåller dem).

`ioperm()` kan endast ge access rättigheter till portarna 0x000 - 0x3ff. För att komma åt högre portadresser, kan man använda `iopl()`, som ger access till *alla* portar på en gång. Använd nivå 3 (dvs `iopl(3)`) för att ge ditt program tillgång till alla portar. (Men var försiktig - att skriva på fel port kan orsaka allehanda otrevliga saker med din dator). Du behöver rootprivilegier för att anropa `iopl()`. Se `iopl(2)` manualen.

Sedan, för att komma åt portarna... För att läsa in en byte, (8 bitar) från en port, call `inb(port)`, den returnerar den byte den läser. För att ställa ut, call `outb(value,port)` (notera parameterordningen). För att läsa in 16 bitar från port `x` och `x+1` ,en byte från vardera, call inw(x) och för att ställa ut, call `outw(value,x)`. Är du osäker på om du skall använda byte eller word instruktioner, är det troligen `inb()` och `outb()` - flertalet apparater konstrueras för bytevis portaccess. Notera att alla portaccesser tar åtminstone cirka en mikrosekund att utföra.

För övrigt fungerar makroanropen `inb_p()`, `outb_p()`, `inw_p()`, och `outw_p()` på samma sätt som ovannämnda, förutom att de lägger till ca en mikrosekund efter varje portaccess. Du kan göra fördröjningen ännu längre, ca 4 mikrosekunder, med `#define REALLY_SLOW_IO` innan du gör `#include <asm/io.h>`. Dessa makron gör normalt (såvida du inte gör `#define SLOW_IO_BY_JUMPING`, vilket blir mindre noggrant) access till port 0x80 för att skapa delay, så du behöver först ge accessrätt till port 0x80 med `ioperm()`. (Skrivning på port 0x80 påverkar ingenting). För mer flexibla delay-metoder, läs vidare.

Det finns sidor till `ioperm(2)`, `iopl(2)` och ovannämnda makron i någorlunda färska utgåvor av Linux manual.

2.2 En alternativ metod: `/dev/port`

Ett annat sätt att komma åt I/O-portar är `open()` `/dev/port` (en 'character device', major number 1, minor 4) för läsning och/eller skrivning (stdio `f*()` funktionerna har intern buffring, så använd inte dem). Gör sedan `lseek()` till den aktuella byten i filen (fil position 0 = port 0x00, fil position 1 = 0x01, och så vidare), och `read()` eller `write()` en byte eller ett ord till eller från den.

Naturligtvis behöver ditt program accessrättigheter till `/dev/port` för att metoden skall fungera. Denna metod är sannolikt långsammare än den normala metoden enligt ovan, men behöver varken optimeringsflaggan vid kompilering eller `ioperm()`. Det behövs inte heller 'root access', bara du ger 'non-root user' eller 'group' access till `/dev/port` - låt vara att detta är dumt ur systemsäkerhetssynpunkt, eftersom det är möjligt att skada systemet, kanske till och med vinna 'root access', genom att använda `/dev/port` för att komma åt hårddisk, nätverkskort, etc. direkt.

3 Avbrott (IRQs) och DMA

Man kan inte använda IRQ eller DMA direkt i en *usermode* process. Man måste skriva en *kernel driver*; se *The Linux Kernel Hacker's Guide* `<http://www.redhat.com:8080/HyperNews/get/khg.html>` Där finns detaljer och *kernel* källkod som exempel. Man kan heller inte stänga av ett avbrott från ett *user-mode* program.

4 Högupplösande timing

4.1 Fördröjningar

Först och främst måste sägas att det inte går att garantera user mode processer exakt kontroll avseende timing eftersom Linux är ett multiprocess system. Din process kan bli utskyfflad under vad som helst mellan 10 millisekunder upp till några sekunder (om belastningen är hög). Detta spelar emellertid ingen roll för flertalet program som använder I/O-portar. För att reducera effekterna, kan du med hjälp av kommandot *nice* ge din process hög prioritet. Se `nice(2)` manualen eller använd *real-time scheduling* enligt nedan.

Om du behöver bättre tidsprecision än vad normala user-mode processer kan ge, så finns vissa förberedelser för 'user-mode real time' support. Linux 2.x kärnor har 'soft real time support', se manualen för `sched_setscheduler(2)`. Det finns en speciell kärna som stöder hård realtid, se `<http://luz.cs.nmt.edu/rtlinux/>` för ytterligare information om detta.

4.1.1 `sleep()` och `usleep()`

Låt oss börja med de lätta funktionsanropen. För att fördröja flera sekunder, är det troligtvis bäst att använda `sleep()`. Fördröjningar på 10-tals millisekunder (ca 10 ms verkar vara minimum) görs med `usleep()`. Dessa funktioner frigör CPU för andra processer, så att ingen CPU-tid går förlorad. Se manualerna `sleep(3)` och `usleep(3)`.

Om fördröjningar är på mindre än 50 ms (beror på din processor och dess belastning), tar det onödigt mycket tid att släppa CPUn, därför att det för Linux *scheduler* (för x86 arkitekturen) vanligtvis tar minst 10-30 millisekunder innan den återger din process kontrollen. Beroende på detta fördröjer `usleep(3)` något mer än vad du specificerar i dina parametrar, och alltid minst ca 10 ms.

4.1.2 nanosleep()

I 2.0.x serien av Linuxkärnor finns ett nytt systemanrop: `nanosleep()` (se `nanosleep(2)` manualen), som möjliggör så korta fördröjningar som ett par mikrosekunder eller mer.

Vid fördröjningar på mindre än 2 ms, om (och endast om) din process är satt till *soft real time scheduling* (med `sched_setscheduler()`), använder `nanosleep()` en vänteloop, i annat fall frigörs CPU på samma sätt som med `usleep()`.

Vänteloopen använder `udelay()` (en intern kernelfunktion som används av många 'kernel drivers'), och loopens längd beräknas med hjälp av BogoMips värdet (det är bara denna sorts hastighet BogoMips värdet mäter noggrant). Se hur det fungerar i `/usr/include/asm/delay.h`

4.1.3 Fördröjningar med port I/O

Ett annat sätt att fördröja ett fåtal mikrosekunder är att använda port I/O. Läsning eller skrivning på port 0x80 (se ovan hur man gör) tar nästan precis 1 mikrosekund oberoende av processortyp och hastighet. Du kan göra det upprepade gånger om du vill vänta ett antal mikrosekunder. Skrivning på denna port torde inte ha några skadliga sidoeffekter på någon standardmaskin och vissa 'kerneldrivers' använder denna metod. det är på detta sättet `{in|out}[bw]_p()` normalt gör sin fördröjning. (se `asmio`.h)/.

Flertalet port I/O instruktioner i adressområdet 0-0x3ff tar nästan exakt 1 mikrosekund, så om du t.ex. använder parallellporten direkt, gör bara några extra `inb()` från porten för att skapa fördröjning.

4.1.4 Att fördröja med assemblerinstruktioner

Om man känner till processortyp och klockhastighet, kan man hårdkoda korta fördröjningar med vissa assemblerinstruktioner (men kom ihåg, processen kan skyfflas ut när som helst, så fördröjningarna kan ibland bli längre). Tabellen nedan ger några exempel. För en 50MHz processor tar en klockcykel 20 ns.

Instruktion	i386 klock cykler	i486 klock cykler
nop	3	1
xchg %ax,%ax	3	3
or %ax,%ax	2	1
mov %ax,%ax	2	1
add %ax,0	2	1

tyvärr känner jag inte till Pentium; förmodligen nära i486. Jag hittar ingen instruktion som tar EN klockcykel i i386. Använd en-cykel instruktioner om du kan, annars kanske *pipelinen* i moderna processortyper förkortar tiden.

Instruktionerna `nop` och `xchg` i tabellen bör inte ha några sidoeffekter. Övriga modifierar statusregistret, men det bör inte betyda något eftersom gcc detekterar detta. `nop` är ett bra val.

Om du vill använda dem, skriv call `asm("instruktion")` i ditt program. Syntaxen ge i tabellen ovan. Vill du göra multipla instruktioner i en `asm()`-sats, så separera med semikolon. Till exempel exekveras i satsen `asm(" nop; nop; nop; nop")` fyra `nop` instruktioner, som fördröjer fyra klockcykler med i486 eller pentium (eller 12 cykler med i386).

`asm()` översätts av gcc till inline assembler kod, så det blir inget *overhead* med funktionsanrop.

Kortare fördröjningar än en klockcykel är inte möjligt med x86 arkitekturen.

4.1.5 rdtsc() för Pentium

Med Pentium kan du erhålla antalet klockcykler som gått sedan senaste uppstart med hjälp av följande C kod:

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

Du kan polla värdet för hur många cykler som helst.

4.2 Att mäta tid

För att mäta tider med en sekunds upplösning, är det nog enklast att använda `time()`. Krävs bättre noggrannhet, ger `gettimeofday()` cirka en mikrosekunds upplösning (men se ovan angående 'scheduling', utskyffling). För Pentium är `rdtsc` kodfragmentet ovan noggrant till en klockcykel.

Om din process skall ha en signal efter en viss tid, så använd `setitimer()` eller `alarm()`. Se manualsidorna.

5 Andra programmeringsspråk

Beskrivningen ovan koncentrerar sig på programmeringsspråket C. Det bör vara tillämpbart även på C++ och Objective C. I assembler får man anropa `ioperm()` eller `iopl()` som i C, och därefter kan man använda I/O-port read/write instruktionerna direkt.

I andra språk, såvida inte du kan infoga inline assembler eller C kod i ditt program eller använda ovannämnda systemanrop, är det nog enklast att skriva en C källkodsfil med funktionerna för I/O-portaccess och separatkompilera och länka den till övriga delar av ditt program. Eller använda `/dev/port` enligt ovan.

6 Några användbara portar

Här följer programmeringsinformation om några portar som direkt kan användas för TTL (eller CMOS) digitala kretsar.

Om du vill använda dessa eller andra gängse portar för det ändamål de är ägnade (t.ex. för att styra en printer eller modem), skall du troligen använda en befintlig drivrutin (vanligtvis inkluderad i kärnan) i stället för att programmera dessa portar direkt som beskrivs i detta HOWTO. Denna sektion är avsedd för dem som vill ansluta LCD-displayer, stegmotorer, eller annan speciell elektronik till en PC's standardport.

Ska du styra en massproducerad produkt som t.ex. scanner (som har funnits på marknaden ett tag), sök efter en befintlig drivrutin. *Hardware-HOWTO* `<http://sunsite.unc.edu/pub/Linux/docs/HOWTO/Hardware-HOWTO>` är ett bra ställe att börja.

`<http://www.hut.fi/Misc/Electronics/>` är en annan bra källa till information om hur man ansluter utrustning till datorer, och om elektronik i allmänhet.

6.1 Parallellporten

Parallellportens basadress (kallad "`BASE`" nedan) är 0x3bc för `/dev/lp0`, 0x378 för `/dev/lp1`, och 0x278 för `/dev/lp2`. Ska du styra något som beter sig som en normal printer, se *Printing-HOWTO* `<http://sunsite.unc.edu/pub/Linux/docs/HOWTO/Printing-HOWTO>`.

Förutom den vanliga output-only moden som beskrivs nedan, finns det en 'extended' bidirektionell mod i flertalet parallellportar. Information om detta och om de nyare ECP/EPP moderna (och om IEEE 1284 standarden allmänt), se `<http://www.fapo.com/>` och `<http://www.senet.com.au/cpeacock/parallel.htm>`. Kom ihåg att eftersom du inte kan använda IRQs eller DMA i ett user-mode program, så kommer du nog att behöva skriva en 'kernel-driver' för att använda ECP/EPP. Jag tror att någon håller på att skriva en sådan driver men jag känner inte till några detaljer.

Porten `BASE+0` (Data port) styr data signalerna (D0 till D7 för bitarna 0 to 7, respektive; tillstånden: 0 = låg (0 V), 1 = hög (5 V)). Skrivning på denna port ställer ut data till donet. En läsning returnerar senast skrivna data i standard- eller extended-moden, eller data på stiften från en ansluten apparat i 'extended read mode'.

Portarna `BASE+1` ('Status port') är 'read-only', och returnerar tillståndet på följande insignaler:

- Bits 0 och 1 är reserverade.
- Bit 2 IRQ status (inget stift, vet inte hur detta fungerar)
- Bit 3 ERROR (1=hög)
- Bit 4 SLCT (1=hög)
- Bit 5 PE (1=hög)
- Bit 6 ACK (1=hög)
- Bit 7 -BUSY (0=hög)

(Vet inte vilka spänningar som motsvarar hög respektive låg.)

Porten `BASE+2` ('Control port') är 'write-only' (läsning returnerar senast inskrivna data), och styr följande status signaler:

- Bit 0 -STROBE (0=hög)
- Bit 1 AUTO_FD_XT (1=hög)
- Bit 2 -INIT (0=hög)
- Bit 3 SLCT_IN (1=hög)

- Bit 4 aktiverar ('enables') parallell portens IRQ (vilket sker på uppflanken hos ACK) när den sätts till 1.
- Bit 5 styr 'extended mode direction' (0 = skriv, 1 = läs), den är 'write-only' (läsning på denna bit returnerar ingenting meningsfullt).
- Bits 6 och 7 är reserverade.

(Återigen, är inte säker på vad som är hög och låg.)

Pinout (ett 25-pin D-don , hona) (i=input, o=output):

```
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 specifikationen säger att stiften 1, 14, 16, och 17 ('control- outputs') har 'open-collektor' utgångar dragna till 5 V genom ett 4.7 K motstånd (sänker 20 mA, ger 0.55 mA, högnivå utgång 5.0 V minus eventuellt spänningsfall). Övriga stift sänker 24 mA och ger 15 mA, och deras högnivå spänning är minst 2.4 V. Lågnivå spänningen är i båda fallen minst 0.5 V. Icke-IBM parallell portar avviker troligen från denna standard. För ytterligare information om detta se `<http://www.hut.fi/Misc/Electronics/circuits/lptpower.html>`.

Slutligen en varning: Var noga med jordningen. Jag har förstört flera parallellportar genom att ansluta dem med datorn igång. Det kan vara ett bra alternativ att använda parallellportar som *inte* sitter på moderkortet för sådana här saker. (Du kan antagligen få en *andra* parallellport med ett billigt standard 'multi-I/O' kort; Stäng bara av de portar du inte behöver, och sätt parallellkortets I/O-adress till en ledig adress. Du behöver inte bekymra dig om parallellportens IRQ, eftersom den vanligtvis inte används.)

6.2 Spelporten (joystick)

Spelporten finns på adresserna 0x200-0x207. För att styra normala joystickar finns en *kernel-level joystick driver*, se `<ftp://sunsite.unc.edu/pub/Linux/kernel/patches/>`, filename `joystick-*`.

Pinout (ett 25-pin D-don , hona) (i=input, o=output):

- 1,8,9,15: +5 V (kraftmatning)
- 4,5,12: Ground
- 2,7,10,14: Digitala ingångar BA1, BA2, BB1, och BB2, respektive
- 3,6,11,13: "Analoga" ingångar AX, AY, BX, och BY, respektive

+5 V stiften verkar ofta vara direkt anslutna till moderkortets kraftmatning, så de bör klara ganska mycket ström, beroende på moderkort, kraftaggregat och spelport.

De digitala ingångarna används till de två joystickarna (joystick A och joystick B, med två knappar vardera) som du kan ansluta till porten. De torde vara normala TTL ingångar, och du kan läsa deras status direkt på statusporten (se nedan). En joystick ger låg (0 V) status när knappen är nedtryckt och eljest hög (5 V från matningen via ett 1K motstånd).

De så kallade analoga ingångarna mäter egentligen resistans. Spelportarna har en fyrfaldig one-shot multivibrator (ett 558 chip) anslutet till de fyra ingångarna. På varje ingångsshylsa i kontaktdonet finns ett 2.2K motstånd mellan ingångshylsan och multivibratorns 'open-collector' utgång, och en 0,01 uF 'timing' kondensator mellan multivibratorns utgång och jord. En joystick har en potentiometer för varje axel (X och Y), dragen mellan +5 V och respektive ingångshylsa (AX och AY för joystick A, eller BX och BY för joystick B).

När multivibratorn aktiveras sätts dess utgång hög (5 V) och den inväntar att timing kondensatorn når 3.3 V innan den sänker respektive utgång. På så sätt blir multivibratorns pulslängd proportionell mot potentiometerns resistans (dvs. joystickens position för respektive axel) enligt följande:

R = (t - 24.2) / 0.011,

där R är potentiometerns resistans och t pulsens längd i mikrosekunder.

Således, för att läsa dessa analoga ingångar, skall man först aktivera multivibratorn (med en skrivning på porten; se nedan), sedan polla (göra upprepade läsningar) tillståndet för de fyra axlarna tills de går från hög till låg, och på så sätt mäta pulstiden. Denna pollning tar mycket CPU tid och i ett ickerealtids multiprocess system som Linux blir inte resultatet så tillförlitligt eftersom man inte kan polla kontinuerligt, såvida du inte gör en 'kernel-driver' och stänger avbrottsingångar när du pollar (vilket tar ännu mer CPU tid). Om du vet att signalen kommer dröja tiotals millisekunder så kan du anropa usleep() innan du börjar polla för att ge CPU tid till andra processer.

Den enda I/O-port som du behöver nå är port 0x201 (de andra portarna beter sig precis likadant eller är inaktiva). En skrivning till porten (spelar ingen roll vad) aktiverar multivibratorn. Läsning från porten returnerar signalernas status:

- Bit 0: AX (status (1=high) of the multivibrator output)
- Bit 1: AY (status (1=high) of the multivibrator output)
- Bit 2: BX (status (1=high) of the multivibrator output)
- Bit 3: BY (status (1=high) of the multivibrator output)
- Bit 4: BA1 (digital input, 1=high)
- Bit 5: BA2 (digital input, 1=high)
- Bit 6: BB1 (digital input, 1=high)
- Bit 7: BB2 (digital input, 1=high)

6.3 Serieporten

Om den apparat som du vill kommunicera med stöder något som liknar RS-232 bör du kunna använda en serieport för att tala med den. Linux drivrutin för serieportar bör räcka för nästan alla tänkbara tillämpningar (du ska inte behöva programmera serieportarna direkt, och skulle så vara måste du skriva en 'kernel driver'); den är mycket flexibel, så att använda ickestandardiserade bithastigheter torde inte vara något problem.

Se `termios(3)` manualen, eller seriedriverns källkod, (`linux/drivers/char/serial.c`), och `<http://www.easysw.com/mike/serial/index.html>` för mer info om hur man programmerar serieportar på Unix system.

7 Tips

Om du behöver bra analog I/O kan du ansluta ett ADC- och/eller DAC-chip till parallellporten (tips: kraft kan du ta från anslutningsdonet till spelporten eller från ett don till en yttre diskenhet eller använda ett separat kraftaggregat. Har du strömsnåla kretsar kan du ta kraftmatning från parallellporten. Du kan också köpa ett AD/DA-kort (de flesta äldre/långsammare typerna ansluts till I/O-portar. Eller, om det räcker med

1 eller 2 kanaler och måttlig noggrannhet, köp ett billigt ljudkort som har stöd från Linux *sound driver*. Ett sådant är också tämligen snabbt.

Noggranna analoga apparater störs lätt om jordningen är bristfällig. Om du får problem av detta slag, kan du pröva att isolera din utrustning från datorn med hjälp av optokopplare. (på *alla* signaler mellan datorn och din utrustning. Försök att få matning till optokopplarna från datorn (lediga signaler från porten kan ge tillräckligt med kraft) för bästa isolation från störning.

Letar du efter Linux mjukvara för mönsterkortframtagning, så finns det en fri sådan som kallas Pcb. Den torde göra ett bra jobb, åtminstone om inte du gör något alltför komplicerat. Den finn med i många Linux distributioner, och den finns tillgänglig i `<ftp://sunsite.unc.edu/pub/Linux/apps/circuits/>` (filename `pcb-*`).

8 Felsökning

Q1.

Jag får *segmentation faults* när jag adresserar portar.

A1.

Antingen har ditt program inte rootprivilegier, eller har `ioperm()` falerat av någon annan orsak. Testa det returnerade värdet från `ioperm()`. Testa också att du verkligen adresserar de portar som du gett tillstånd till med `ioperm()` (se Q3).

Om du använder 'delaying macros' (`inb_p()`, `outb_p()`, osv.), kom ihåg att du då måste anropa `ioperm()` för att ge accesstillstånd till adress 0x80.

Q2.

Jag kan inte hitta var `in*()`, `out*()` funktionerna definieras, och gcc klagar över odefinierade referenser.

A2.

Du kompilerade inte med optimeringsflaggan på (`-O1 eller högre`), och därför kunde inte gcc lösa upp de makron som finns i `asm/io.h`. Eller glömde du kanske `#include <asm/io.h>`.

Q3.

`out*()` gör ingenting, eller gör något konstigt.

A3.

Kolla ordningen på parametrarna; set skall vara `outb(value, port)`, inte `outb(port, value)` som förekommer i MS-DOS.

Q4.

Jag vill köra en standard RS-232 device/parallel printer/joystick...

A4.

Då är det nog bäst att använda en befintlig driver (i Linux kernel eller en X-server eller någon annanstans) för detta. Drivrutinerna är vanligtvis mycket flexibla, så att även en ickestandard apparat fungerar vanligtvis med dem. Se info om standard portar ovan efter hänvisningar till dokumentation.

9 Kod exempel

Här följer ett enkelt exempel på kod för I/O-port access:

```
/*
 * example.c: very simple example of port I/O
 *
 * This code does nothing useful, just a port write, a pause,
 * and a port read. Compile with 'gcc -O2 -o example example.c',
 * and run as root with './example'.
 */

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

#define BASEPORT 0x378 /* lp1 */

int main()
{
  /* Get access to the ports */
  if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}

  /* Set the data signals (D0-7) of the port to all low (0) */
  outb(0, BASEPORT);

  /* Sleep for a while (100 ms) */
  usleep(100000);

  /* Read from the status port (BASE+1) and display the result */
  printf("status: %d\n", inb(BASEPORT + 1));

  /* We don't need the ports anymore */
  if (ioperm(BASEPORT, 3, 0)) {perror("ioperm"); exit(1);}

  exit(0);
}

/* end of example.c */
```

10 Erkännande

Alltför många har bidragit till artikeln för att jag skall kunna räkna upp alla, men tack allesammans. Jag har inte besvarat alla bidrag som jag fått; ledsen för det, men återigen tack för all hjälp.