Linux I/O port programming mini−HOWTO

Table of Contents

Linux I/O port programming mini−HOWTO

Author: Riku Saikkonen `<Riku.Saikkonen@hut.fi>`

v3.0, 2000−12−13

This HOWTO document describes programming hardware I/O ports and waiting for small periods of time in user−mode Linux programs running on the Intel x86 architecture.

---

1. Introduction

This HOWTO document describes programming hardware I/O ports and waiting for small periods of time in user−mode Linux programs running on the Intel x86 architecture. This document is a descendant of the very small IO−Port mini−HOWTO by the same author.

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

---

2. Using I/O ports in C programs

2.1 The normal method

Routines for accessing I/O ports are in `/usr/include/asm/io.h` (or `linux/include/asm-i386/io.h` in the kernel source distribution). The routines there are inline macros, so it is enough to `#include <asm/io.h>`; you do not need any additional libraries.

Because of a limitation in gcc (present in all versions I know of, including egcs), you *have to* compile any source code that uses these routines with optimisation turned on (`gcc -O1` or higher), or alternatively use `#define extern static` before you `#include <asm/io.h>` (remember to `#undef extern`afterwards).

For debugging, you can use `gcc -g -O` (at least with modern versions of gcc), though optimisation can sometimes make the debugger behave a bit strangely. If this bothers you, put the routines that use I/O port access in a separate source file and compile only that with optimisation turned on.

Permissions

Before you access any ports, you must give your program permission to do so. This is done by calling the `ioperm()` function (declared in `unistd.h`, and defined in the kernel) somewhere near the start of your program (before any I/O port accesses). The syntax is `ioperm(from, num, turn_on)`, where `from` is the first port number to give access to, and `num` the number of consecutive ports to give access to. For example, `ioperm(0x300, 5, 1)` would give access to ports 0x300 through 0x304 (a total of 5 ports). The last argument is a Boolean value specifying whether to give access to the program to the ports (true (1)) or to remove access (false (0)). You can call `ioperm()` multiple times to enable multiple non−consecutive ports. See the `ioperm(2)` manual page for details on the syntax.

The `ioperm()` call requires your program to have root privileges; thus you need to either run it as the root user, or make it setuid root. You can drop the root privileges after you have called `ioperm()` to enable the ports you want to use. You are not required to explicitly drop your port access privileges with `ioperm(..., 0)` at the end of your program; this is done automatically as the process exits.

A `setuid()` to a non−root user does not disable the port access granted by `ioperm()`, but a `fork()` does (the child process does not get access, but the parent retains it).

`ioperm()` can only give access to ports 0x000 through 0x3ff; for higher ports, you need to use `iopl()` (which gives you access to all ports at once). Use the level argument 3 (i.e., `iopl(3)`) to give your program access to *all* I/O ports (so be careful −−− accessing the wrong ports can do all sorts of nasty things to your computer). Again, you need root privileges to call `iopl()`. See the `iopl(2)` manual page for details.

Accessing the ports

To input a byte (8 bits) from a port, call `inb(port)`, it returns the byte it got. To output a byte, call `outb(value, port)` (please note the order of the parameters). To input a word (16 bits) from ports `x` and `x+1` (one byte from each to form the word, using the assembler instruction `inw`), call `inw(x)`. To output a word to the two ports, use `outw(value, x)`. If you're unsure of which port instructions (byte or word) to use, you probably want `inb()` and `outb()` −−− most devices are designed for bytewise port access. Note that all port access instructions take at least about a microsecond to execute.

The `inb_p()`, `outb_p()`, `inw_p()`, and `outw_p()` macros work otherwise identically to the ones above, but they do an additional short (about one microsecond) delay after the port access; you can make the delay about four microseconds with `#define REALLY_SLOW_IO` before you `#include <asm/io.h>`. These macros normally (unless you `#define SLOW_IO_BY_JUMPING`, which is probably less accurate) use a port output to port 0x80 for their delay, so you need to give access to port 0x80 with `ioperm()` first (outputs to port 0x80 should not affect any part of the system). For more versatile methods of delaying, read on.

There are manual pages for `ioperm(2)`, `iopl(2)`, and the above macros in reasonably recent releases of the Linux manual page collection.

2.2 An alternate method: `/dev/port`

Another way to access I/O ports is to `open() /dev/port` (a character device, major number 1, minor 4) for reading and/or writing (the stdio `f*()` functions have internal buffering, so avoid them). Then `lseek()` to the appropriate byte in the file (file position 0 = port 0x00, file position 1 = port 0x01, and so on), and `read()` or `write()` a byte or word from or to it.

Naturally, for this to work your program needs read/write access to `/dev/port`. This method is probably slower than the normal method above, but does not need compiler optimisation nor `ioperm()`. It doesn't need root access either, if you give a non−root user or group access to `/dev/port` −−− but this is a very bad thing to do in terms of system security, since it is possible to hurt the system, perhaps even gain root access, by using `/dev/port` to access hard disks, network cards, etc. directly.

You cannot use `select(2)` or `poll(2)` to read /dev/port, because the hardware does not have a facility for notifying the CPU when a value in an input port changes.

---

3. Interrupts (IRQs) and DMA access

You cannot use IRQs or DMA directly from a user−mode process. You need to write a kernel driver; see The Linux Kernel Hacker's Guide for details and the kernel source code for examples.

You can disable interrupts from within a user−mode program, though it can be dangerous (even kernel drivers do it for as short a time as possible). After calling `iopl(3)`, you can disable interrupts simply by calling `asm("cli");`, and re−enable them with `asm("sti");`.

4. High−resolution timing

4.1 Delays

First of all, I should say that you cannot guarantee user−mode processes to have exact control of timing because of the multi−tasking nature of Linux. Your process might be scheduled out at any time for anything from about 10 milliseconds to a few seconds (on a system with very high load). However, for most applications using I/O ports, this does not really matter. To minimise this, you may want to nice your process to a high−priority value (see the `nice(2)` manual page) or use real−time scheduling (see below).

If you want more precise timing than normal user−mode processes give you, there are some provisions for user−mode `real time' support. Linux 2.x kernels have soft real time support; see the manual page for `sched_setscheduler(2)` for details. There is a special kernel that supports hard real time; see http://luz.cs.nmt.edu/~rtlinux/ for more information on this.

Sleeping: `sleep()` and `usleep()`

Now, let me start with the easier timing calls. For delays of multiple seconds, your best bet is probably to use `sleep()`. For delays of at least tens of milliseconds (about 10 ms seems to be the minimum delay), `usleep()` should work. These functions give the CPU to other processes (``sleep''), so CPU time isn't wasted. See the manual pages `sleep(3)` and `usleep(3)` for details.

For delays of under about 50 milliseconds (depending on the speed of your processor and machine, and the system load), giving up the CPU takes too much time, because the Linux scheduler (for the x86 architecture) usually takes at least about 10−30 milliseconds before it returns control to your process. Due to this, in small delays, `usleep(3)` usually delays somewhat more than the amount that you specify in the parameters, and at least about 10 ms.

`nanosleep()`

In the 2.0.x series of Linux kernels, there is a new system call, `nanosleep()` (see the `nanosleep(2)` manual page), that allows you to sleep or delay for short times (a few microseconds or more).

For delays <= 2 ms, if (and only if) your process is set to soft real time scheduling (using `sched_setscheduler()`), `nanosleep()` uses a busy loop; otherwise it sleeps, just like `usleep()`.

The busy loop uses `udelay()` (an internal kernel function used by many kernel drivers), and the length of the loop is calculated using the BogoMips value (the speed of this kind of busy loop is one of the things that BogoMips measures accurately). See `/usr/include/asm/delay.h`) for details on how it works.

Delaying with port I/O

Another way of delaying small numbers of microseconds is port I/O. Inputting or outputting any byte from/to port 0x80 (see above for how to do it) should wait for almost exactly 1 microsecond independent of your processor type and speed. You can do this multiple times to wait a few microseconds. The port output should have no harmful side effects on any standard machine (and some kernel drivers use it). This is how `{in|out}[bw]_p()` normally do the delay (see `asm/io.h`).

Actually, a port I/O instruction on most ports in the 0−0x3ff range takes almost exactly 1 microsecond, so if you're, for example, using the parallel port directly, just do additional `inb()`s from that port to delay.

Delaying with assembler instructions

If you know the processor type and clock speed of the machine the program will be running on, you can hard−code shorter delays by running certain assembler instructions (but remember, your process might be scheduled out at any time, so the delays might well be longer every now and then). For the table below, the internal processor speed determines the number of clock cycles taken; e.g., for a 50 MHz processor (e.g. 486DX−50 or 486DX2−50), one clock cycle takes 1/50000000 seconds (=200 nanoseconds).

```
Instruction   i386 clock cycles   i486 clock cycles
xchg %bx,%bx          3                   3
nop                   3                   1
or %ax,%ax            2                   1
mov %ax,%ax           2                   1
add %ax,0             2                   1
```

Clock cycles for Pentiums should be the same as for i486, except that on Pentium Pro/II, `add %ax, 0` may take only 1/2 clock cycles. It can sometimes be paired with another instruction (because of out−of−order execution, this need not even be the very next instruction in the instruction stream).

The instructions `nop` and `xchg` in the table should have no side effects. The rest may modify the flags register, but this shouldn't matter since gcc should detect it. `xchg %bx, %bx` is a safe choice for a delay instruction.

To use these, call `asm("instruction")` in your program. The syntax of the instructions is as in the table above; if you want multiple instructions in a single `asm()` statement, separate them with semicolons. For example, `asm("nop ; nop ; nop ; nop")` executes four `nop` instructions, delaying for four clock cycles on i486 or Pentium processors (or 12 clock cycles on an i386).

`asm()` is translated into inline assembler code by gcc, so there is no function call overhead.

Shorter delays than one clock cycle are impossible in the Intel x86 architecture.

`rdtsc` for Pentiums

For Pentiums, you can get the number of clock cycles elapsed since the last reboot with the following C code (which executes the CPU instrution named RDTSC):

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

You can poll this value in a busy loop to delay for as many clock cycles as you want.

4.2 Measuring time

For times accurate to one second, it is probably easiest to use `time()`. For more accurate times, `gettimeofday()` is accurate to about a microsecond (but see above about scheduling). For Pentiums, the `rdtsc` code fragment above is accurate to one clock cycle.

If you want your process to get a signal after some amount of time, use `setitimer()` or `alarm()`. See the manual pages of the functions for details.

---

5. Other programming languages

The description above concentrates on the C programming language. It should apply directly to C++ and Objective C. In assembler, you have to call `ioperm()` or `iopl()` as in C, but after that you can use the I/O port read/write instructions directly.

In other languages, unless you can insert inline assembler or C code into the program or use the system calls mentioned above, it is probably easiest to write a simple C source file with functions for the I/O port accesses or delays that you need, and compile and link it in with the rest of your program. Or use `/dev/port` as described above.

---

6. Some useful ports

Here is some programming information for common ports that can be directly used for general−purpose TTL (or CMOS) logic I/O.

If you want to use these or other common ports for their intended purpose (e.g., to control a normal printer or modem), you should most likely use existing drivers (which are usually included in the kernel) instead of programming the ports directly as this HOWTO describes. This section is intended for those people who want to connect LCD displays, stepper motors, or other custom electronics to a PC's standard ports.

If you want to control a mass−market device like a scanner (that has been on the market for a while), look for an existing Linux driver for it. The Hardware−HOWTO is a good place to start.

http://www.hut.fi/Misc/Electronics/ is a good source for more information on connecting devices to computers (and on electronics in general).

6.1 The parallel port

The parallel port's base address (called ``BASE'' below) is 0x3bc for `/dev/lp0`, 0x378 for `/dev/lp1`, and 0x278 for `/dev/lp2`. If you only want to control something that acts like a normal printer, see the Printing−HOWTO.

In addition to the standard output−only mode described below, there is an `extended' bidirectional mode in most parallel ports. For information on this and the newer ECP/EPP modes (and the IEEE 1284 standard in general), see http://www.fapo.com/ and http://www.senet.com.au/~cpeacock/parallel.htm. Remember that since you cannot use IRQs or DMA in a user−mode program, you will probably have to write a kernel driver to use ECP/EPP; I think someone is writing such a driver, but I don't know the details.

The port `BASE+0` (Data port) controls the data signals of the port (D0 to D7 for bits 0 to 7, respectively; states: 0 = low (0 V), 1 = high (5 V)). A write to this port latches the data on the pins. A read returns the data last written in standard or extended write mode, or the data in the pins from another device in extended read mode.

The port `BASE+1` (Status port) is read−only, and returns the state of the following input signals:

- Bits 0 and 1 are reserved.
- Bit 2 IRQ status (not a pin, I don't know how this works)
- Bit 3 ERROR (1=high)
- Bit 4 SLCT (1=high)
- Bit 5 PE (1=high)
- Bit 6 ACK (1=high)
- Bit 7 −BUSY (0=high)

The port `BASE+2` (Control port) is write−only (a read returns the data last written), and controls the following status signals:

- Bit 0 −STROBE (0=high)
- Bit 1 −AUTO_FD_XT (0=high)
- Bit 2 INIT (1=high)
- Bit 3 −SLCT_IN (0=high)
- Bit 4 enables the parallel port IRQ (which occurs on the low−to−high transition of ACK) when set to 1.
- Bit 5 controls the extended mode direction (0 = write, 1 = read), and is completely write−only (a read returns nothing useful for this bit).
- Bits 6 and 7 are reserved.

Pinout (a 25−pin female D−shell connector on the port) (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
```

The IBM specifications say that pins 1, 14, 16, and 17 (the control outputs) have open collector drivers pulled to 5 V through 4.7 kiloohm resistors (sink 20 mA, source 0.55 mA, high−level output 5.0 V minus pullup). The rest of the pins sink 24 mA, source 15 mA, and their high−level output is min. 2.4 V. The low state for both is max. 0.5 V. Non−IBM parallel ports probably deviate from this standard. For more information on this, see http://www.hut.fi/Misc/Electronics/circuits/lptpower.html.

Finally, a warning: Be careful with grounding. I've broken several parallel ports by connecting to them while the computer is turned on. It might be a good thing to use a parallel port not integrated on the motherboard for things like this. (You can usually get a second parallel port for your machine with a cheap standard `multi−I/O' card; just disable the ports that you don't need, and set the parallel port I/O address on the card to a free address. You don't need to care about the parallel port IRQ if you don't use it.)

6.2 The game (joystick) port

The game port is located at port addresses 0x200−0x207. If you want to control normal joysticks, you're probably better off using the drivers distributed with the Linux kernel.

Pinout (a 15−pin female D−shell connector on the port):

- 1,8,9,15: +5 V (power)
- 4,5,12: Ground
- 2,7,10,14: Digital inputs BA1, BA2, BB1, and BB2, respectively
- 3,6,11,13: ``Analog'' inputs AX, AY, BX, and BY, respectively

The +5 V pins seem to often be connected directly to the power lines in the motherboard, so they may be able to source quite a lot of power, depending on the motherboard, power supply and game port.

The digital inputs are used for the buttons of the two joysticks (joystick A and joystick B, with two buttons each) that you can connect to the port. They should be normal TTL−level inputs, and you can read their status directly from the status port (see below). A real joystick returns a low (0 V) status when the button is pressed and a high (the 5 V from the power pins through an 1 Kohm resistor) status otherwise.

The so−called analog inputs actually measure resistance. The game port has a quad one−shot multivibrator (a 558 chip) connected to the four inputs. In each input, there is a 2.2 Kohm resistor between the input pin and the multivibrator output, and a 0.01 uF timing capacitor between the multivibrator output and the ground. A real joystick has a potentiometer for each axis (X and Y), wired between +5 V and the appropriate input pin (AX or AY for joystick A, or BX or BY for joystick B).

The multivibrator, when activated, sets its output lines high (5 V) and waits for each timing capacitor to reach 3.3 V before lowering the respective output line. Thus the high period duration of the multivibrator is proportional to the resistance of the potentiometer in the joystick (i.e., the position of the joystick in the appropriate axis), as follows:

$$R = (t - 24.2) / 0.011,$$

where R is the resistance (ohms) of the potentiometer and t the high period duration (microseconds).

Thus, to read the analog inputs, you first activate the multivibrator (with a port write; see below), then poll the state of the four axes (with repeated port reads) until they drop from high to low state, measuring their high period duration. This polling uses quite a lot of CPU time, and on a non−realtime multitasking system like (normal user−mode) Linux, the result is not very accurate because you cannot poll the port constantly (unless you use a kernel−level driver and disable interrupts while polling, but this wastes even more CPU time). If you know that the signal is going to take a long time (tens of ms) to go down, you can call usleep() before polling to give CPU time to other processes.

The only I/O port you need to access is port 0x201 (the other ports either behave identically or do nothing). Any write to this port (it doesn't matter what you write) activates the multivibrator. A read from this port returns the state of the input signals:

- 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 The serial port

If the device you're talking to supports something resembling RS−232, you should be able to use the serial port to talk to it. The Linux serial driver should be enough for almost all applications (you shouldn't have to program the serial port directly, and you'd probably have to write a kernel driver to do it); it is quite versatile, so using non−standard bps rates and so on shouldn't be a problem.

See the `termios(3)` manual page, the serial driver source code (`linux/drivers/char/serial.c`), and http://www.easysw.com/~mike/serial/ for more information on programming serial ports on Unix systems.

---

7. Hints

If you want good analog I/O, you can wire up ADC and/or DAC chips to the parallel port (hint: for power, use the game port connector or a spare disk drive power connector wired to outside the computer case, unless you have a low−power device and can use the parallel port itself for power, or use an external power supply), or buy an AD/DA card (most of the older/slower ones are controlled by I/O ports). Or, if you're satisfied with 1 or 2 channels, inaccuracy, and (probably) bad zeroing, a cheap sound card supported by the Linux sound driver should do (and it's quite fast).

With accurate analog devices, improper grounding may generate errors in the analog inputs or outputs. If you experience something like this, you could try electrically isolating your device from the computer with optocouplers (on *all* signals between the computer and your device). Try to get power for the optocouplers from the computer (spare signals on the port may give enough power) to achieve better isolation.

If you're looking for printed circuit board design software for Linux, there is a free X11 application called Pcb that should do a nice job, at least if you aren't doing anything very complex. It is included in many Linux distributions, and available in ftp://sunsite.unc.edu/pub/Linux/apps/circuits/ (filename `pcb-*`).

---

8. Troubleshooting

Q1.

I get segmentation faults when accessing ports.

A1.

Either your program does not have root privileges, or the `ioperm()` call failed for some other reason. Check the return value of `ioperm()`. Also, check that you're actually accessing the ports that you enabled with `ioperm()` (see Q3). If you're using the delaying macros (`inb_p()`, `outb_p()`, and so on), remember to call `ioperm()` to get access to port 0x80 too.

Q2.

I can't find the `in*()`, `out*()` functions defined anywhere, and gcc complains about undefined references.

A2.

You did not compile with optimisation turned on (`−O`), and thus gcc could not resolve the macros in `asm/io.h`. Or you did not `#include <asm/io.h>` at all.

Q3.

`out*()` doesn't do anything, or does something weird.

A3.

Check the order of the parameters; it should be `outb(value, port)`, not `outportb(port, value)` as is common in MS−DOS.

Q4.

I want to control a standard RS−232 device/parallel printer/joystick...

A4.

You're probably better off using existing drivers (in the Linux kernel or an X server or somewhere else) to do it. The drivers are usually quite versatile, so even slightly non−standard devices usually work with them. See the information on standard ports above for pointers to documentation for them.

9. Example code

Here's a piece of simple example code for 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. Credits

Too many people have contributed for me to list, but thanks a lot, everyone. I have not replied to all the contributions that I've received; sorry for that, and thanks again for the help.