Hardware Management

We will develop code that is designed to control the parallel port.

First we will look at the kernel functions that allow communication to 8-bit, 16-bit and 32-bit I/O ports.

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

Read and write 8-bit wide ports.

The port argument is defined as unsigned long on some and unsigned short on other platforms.

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

The 16-bit versions.

```
unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);
```

The 32-bit versions in which doubleword is defined as unsigned long or unsigned int, according to the platform.

Parallel Port

On the 80x86, it is possible that the processor tries to transfer data too quickly to and from the bus, particularly if the I/O instructions are back-to-back.

There are `inb_p` and `outb_p` (pause) versions of the above functions.

You can use `SLOW_DOWN_IO` macro to add the delay also.

There are also string functions defined as follows:

```
void insb(unsigned port, void *addr, unsigned long
cnt);
void outsb(unsigned port, void *addr, unsigned long
cnt);
```

Along with the `w` and `l` versions for 16-bit and 32-bit transfers.

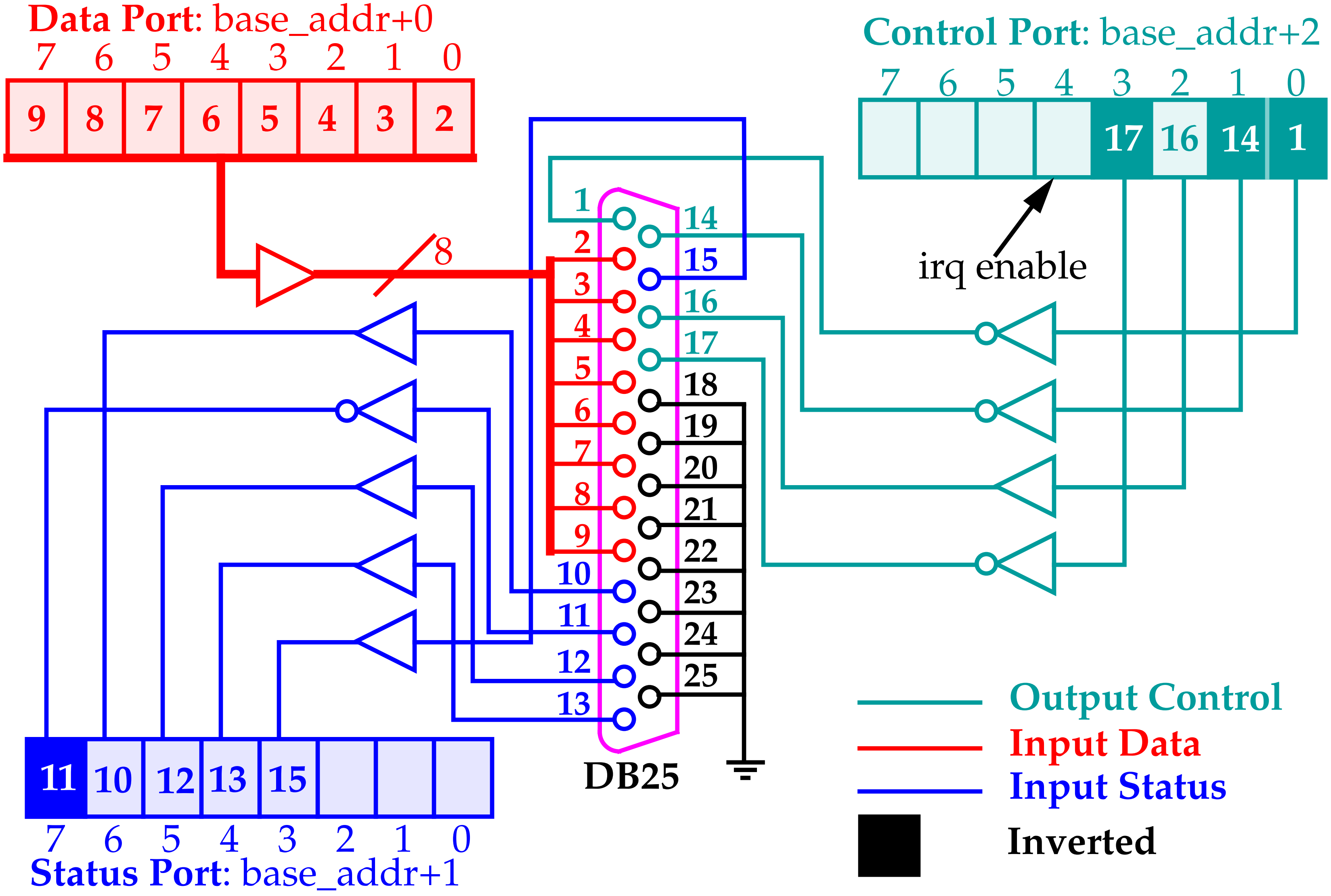
Parallel Port

The parallel port, in its minimal configuration, is made up of a few 8-bit ports.

Data written to the output port shows up on the output pins of the 25-pin connector at standard TTL levels (0 and 5 volts w/ threshold of 1.2 V).

Parallel Port

The parallel port specification.



Interrupt Handling

Control Port bit 4 is not connected to the DB25 connector, but rather it is used to enable interrupts.

`outb` is used at module initialization time to do this.

The device generates an interrupt by raising pin 10 (the “ACK” bit).

Of course, nothing will happen until an interrupt service routine has been installed.

Linux acknowledges and ignores any unexpected interrupts.

There are only 15 or 16 interrupt lines and the kernel keeps a registry of interrupt lines (similar to the I/O port registry).

Obtaining Interrupt Request Numbers

A module requests and frees an interrupt channel (IRQ) using functions declared in <*linux/sched.h*>:

```
extern int request_irq(unsigned int irq,
   void (*handler)(int, void *, struct pt_regs *),
   unsigned long flags,
   const char *device,
   void *dev_id);

extern void free_irq(unsigned int irq, void *dev_id);
```

- *irq* is the requested IRQ (which may not correspond to the hardware number).
- *handler* is a pointer to the ISR.
- *flags* is a bitmask of options related to interrupt management.
- *device* is the string used in /proc/interrupts to show the owner of the interrupt.
- *dev_id* is a unique identifier used for shared interrupt lines and is usually set to NULL.

Obtaining Interrupt Request Numbers

The bits that can be set in *flags* are:

- SA_INTERRUPT

  When set, this indicates a “fast” interrupt handler (cli issued), otherwise it is “slow” (sti enables processor to handle other types of interrupts).

- SA_SHIRQ

  Indicates that the interrupt can be shared between devices.

- SA_SAMPLE_RANDOM

  Indicates that the generated interrupts contribute to the entropy pool used by application software to choose secure keys for encryption.

A 0 return value from *request_irq* indicates success while a negative number indicates an error.

e.g., -EBUSY indicates another driver is using the IRQ requested.

The interrupt handler is usually not installed from within *init_module*, since the device may never use it.

Because interrupt lines are a limited resource, the installation is usually done on the first *open* call.

Obtaining Interrupt Request Numbers

The request_irq should appear in the open call **before** the hardware is instructed to generate interrupts.

The short module (Simple Hardware Operations and Raw Tests) actually installs the ISR in *init_module* for simplicity.

```
if (short_irq >= 0)
   {
   result = request_irq(short_irq,
      short_interrupt, SA_INTERRUPT, "short", NULL);

   if (result)
      {
       printk(KERN_INFO "short: can't get assigned \
         irq %i\n",short_irq);
       short_irq = -1;
       }
   else /* Enable interrupts(assume parallel port) */
      { outb(0x10, short_base+2); }
   }
```

Obtaining Interrupt Request Numbers

This code shows the handler being installed as a ‘fast”handler , without support for interrupt sharing or contributing to system entropy.

Hardware interrupts are counted, and reported in */proc/interrupts* file.

A sample of mine looks like:

```
           CPU0
  0:   93487704          XT-PIC  timer
  1:     239645          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  9:    4961908          XT-PIC  SMC EPIC/100
 12:    1668143          XT-PIC  PS/2 Mouse
 14:    1550375          XT-PIC  ide0
NMI:          0
```

The first column indicates the interrupt number.

The second the number of times the interrupt occurred.

The third is the chip.

The fourth is the identification string.

Obtaining Interrupt Request Numbers

A second file that reports this (and other) data in a different format is */proc/stat*.

A sample of mine looks like:

```
intr 101942156 93507692 240219 0 2 6 0 3 0 1 4963072 0
0 1680074 1 1551081 5 0 ...
```

Here, a number is given for all interrupt lines (the first number is the total number of interrupts).

This allows you to determine if a driver is working, even if it requests its interrupt line in the *open* call.

Autodetection is the most desirable way to assign interrupt numbers, particularly for jumperless I/O cards.

The *short* module shows examples of the probing required to do this.

Some devices, e.g. parallel port, feature a default behavior that rarely changes, and the driver can use a default value.

Obtaining Interrupt Request Numbers

The parallel port driver implements this choosing a default IRQ based on the base address as:

```
/* not yet specified: force the default on. */
if (short_irq < 0)
   switch(short_base)
      {
      case 0x378: short_irq = 7; break;
      case 0x278: short_irq = 2; break;
      case 0x3bc: short_irq = 5; break;
      }
```

Implementing the handler:

The role of the handler:

- Inform the device (clear a bit in a control port) that the interrupt is being serviced.

  This is not necessary for the parallel port, but is common in many other devices.

- Awaken processes sleeping on the device (blocking on a read operation).

  If a large data transfer is involved, task queues should be used.

Implementing the ISR

Remember the handler can’t transfer data to or from user space, because it doesn’t execute in the context of a process.

The short ISR simply prints the time of day to a page-sized circular buffer:

```
void short_interrupt(int irq, void *dev_id,
   struct pt_regs *regs)
   {
   struct timeval tv;
   do_gettimeofday(&tv);

/* Write a 16 byte record. */
   short_head += sprintf((char *)short_head,
      "%08u.%06u\n",(int)(tv.tv_sec % 100000000),
      (int)(tv.tv_usec));
   if (short_head == short_buffer + PAGE_SIZE)
      short_head = short_buffer; /* wrap */

/* awake any reading process */
   wake_up_interruptible(&short_queue);
   }
```

Implementing the ISR

The example code for *short* requires you to connect pins 9 (high order data bit) and 10 ("ACK"or interr upt request pin) together.

In this way, the act of writing to the port ('cat'ing data to the port) causes an interrupt to be generated and the above code to execute.

The *short_i_write* routine will cause an interrupt to be generated even for ASCII data writes (which normally would not set the high order bit since ASCII data is only 7 bits wide).

The *short_interrupt* routine writes "time of day"data to the device buf fer, *short_buffer*, as shown above.

It also "wakes up"any pr ocess waiting to read data from this buffer.

The *short_i_read* routine will try to read data from the *short_buffer*, and sleep if none is available.

Similar to *scull*, reads and writes are directed to */dev/shortint*