8259 PIC

From OSDev Wiki
(Redirected from PIC)

The 8259 *Programmable Interrupt Controller* (PIC) is one of the most important chips making up the x86 architecture. Without it, the x86 architecture would not be an interrupt driven architecture. The function of the 8259A is to manage hardware interrupts and send them to the appropriate system interrupt. This allows the system to respond to devices needs without loss of time (from polling the device, for instance).

It is important to note that APIC has replaced the 8259 PIC in more modern systems, especially those with multiple cores/processors.

Contents

What does the 8259 PIC do?

The 8259 PIC controls the CPU's interrupt mechanism, by accepting several interrupt requests and feeding them to the processor in order. For instance, when a keyboard registers a keyhit, it sends a pulse along it's interrupt line (IRQ 1) to the PIC chip, which then translates the IRQ into a system interrupt, and sends a message to interrupt the CPU from whatever it is doing. Part of the kernel's job is to either handle these IRQs and perform the necessary procedures (poll the keyboard for the scancode) or alert a userspace program to the interrupt (send a message to the keyboard driver).

Without a PIC, you would have to poll all the devices in the system to see if they want to do anything (signal an event), but with a PIC, your system can run along nicely until such time that a device wants to signal an event, which means you don't waste time going to the devices, you let the devices come to you when they are ready.

The IBM PC 8259 PIC Architecture

In the beginning (IBM PC and XT), only a single 8259 PIC chip was used, which provided 8 IRQs to the system. These were traditionally mapped by the BIOS to interrupts 8 to 15 (0x08 to 0x0F). It is unlikely that any of these single-PIC machines will be encountered these days.

The IBM PC/AT 8259 PIC Architecture

The IBM PC/AT extended the PC architecture by adding a second 8259 PIC chip. This was possible due to the 8259A's ability to cascade interrupts, that is, have them flow through one chip and into another. This gives a total of 15 interrupts. Why 15 and not 16? That's because when you cascade chips, the PIC needs to use one of the interrupt lines to signal the other chip.

Thus, in an AT, IRQ line 2 is used to signal the second chip... But to confuse things more, IRQ 9 is redirected to IRQ 2. So when you get an IRQ 9, the signal is redirected to IRQ 2. This two-chip architecture is still used and available in modern systems, and hasn't changed (except for the advent of the above-mentioned APIC architecture).

How does the 8259 PIC chip work?

Each of the two 8259 PICs in modern systems have 8 inputs. When any of the inputs is raised, the PIC sets a bit internally telling one of the inputs needs servicing. It then checks whether that channel is masked or not, and whether there's an interrupt already pending. If the channel is unmasked and there's no interrupt pending, the PIC will raise the interrupt line. On the slave, this feeds IRQ 2 to the master, and the master is connected to the processor interrupt line.

When the processor accepts the interrupt, the master checks which of the two PICs is responsible for answering, then either supplies the interrupt number to the processor, or asks the slave to do so. The PIC that answers looks up the "vector offset" variable stored internally and adds the input line to form the requested interrupt number. After that the processor will look up the interrupt address and act accordingly (see Interrupts for more details).

Programming with the 8259 PIC

Each chip (master and slave) has a command port and a data port (given in the table below). When no command is issued, the data port allows us to access the interrupt mask of the 8259 PIC.

Chip - Purpose	I/O port
Master PIC - Command	0x0020
Master PIC - Data	0x0021
Slave PIC - Command	0x00A0
Slave PIC - Data	0x00A1

- Each PIC vector offset must be divisible by 8, as the 8259A uses the lower 3 bits for the interrupt number of a particular interrupt (0..7).
- The only way to change the vector offsets used by the 8259 PIC is to re-initialize it, which explains why the code is "so long" and plenty of things that have apparently no reasons to be here.
- If you plan to return to real mode from protected mode (for any purpose), you really must restore the PIC to its former configuration.

Real Mode

Chip	Interrupt numbers (IRQ)	Vector offset	Interrupt Numbers
Master PIC	0 to 7	0x08	0x08 to 0x0F
Slave PIC	8 to 15	0x70	0x70 to 0x77

These default BIOS values suit real mode programming quite well; they do not conflict with any CPU exceptions like they do in protected mode.

Protected Mode

In protected mode, the IRQs 0 to 7 conflict with the CPU exception which are reserved by Intel up until 0x1F. (It was an IBM design mistake.) Consequently it is difficult to tell the difference between an IRQ or an software error. It is thus recommended to change the PIC's offsets (also known as remapping the PIC) so that IRQs use non-reserved vectors. A common choice is to move them to the beginning of the available range (IRQs 0..0xF -> INT 0x20..0x2F). For that, we need to set the master PIC's offset to 0x20 and the slave's to 0x28. For code examples, see below.

Code Examples

Common Definitions

This is just a set of definitions common to the rest of this section. For the outb(), inb() and io_wait() functions, see this page.

```
#define PIC1            0x20            /* IO base address for master PIC
#define PIC2            0xA0            /* IO base address for slave PIC
#define PIC1_COMMAND    PIC1
#define PIC1_DATA       (PIC1+1)
#define PIC2_COMMAND    PIC2
#define PIC2_DATA       (PIC2+1)
```

End of Interrupt

Perhaps the most common command issued to the PIC chips is the *end of interrupt* (EOI) command (code 0x20). This is issued to the PIC chips at the end of an IRQ-based interrupt routine. If the IRQ came from the Master PIC, it is sufficient to issue this command only to the Master PIC; however if the IRQ came from the Slave PIC, it is necessary to issue the command to both PIC chips.

```
#define PIC_EOI         0x20            /* End-of-interrupt command code

void PIC_sendEOI(unsigned char irq)
{
        if(irq >= 8)
                outb(PIC2_COMMAND,PIC_EOI);

        outb(PIC1_COMMAND,PIC_EOI);
}
```

Initialisation

When you enter protected mode (or even before hand, if you're not using GRUB) the first command you will need to give the two PICs is the initialise command (code 0x11). This command makes the PIC wait for 3 extra "initialisation words" on the data port. These bytes give the PIC:

- Its vector offset. (ICW2)
- Tell it how it is wired to master/slaves. (ICW3)
- Gives additional information about the environment. (ICW4)

```
/* reinitialize the PIC controllers, giving them specified vector offsets
   rather than 8h and 70h, as configured by default */

#define ICW1_ICW4       0x01            /* ICW4 (not) needed */
#define ICW1_SINGLE     0x02            /* Single (cascade) mode */
#define ICW1_INTERVAL4  0x04            /* Call address interval 4 (8) */
#define ICW1_LEVEL      0x08            /* Level triggered (edge) mode */
#define ICW1_INIT       0x10            /* Initialization - required! */

#define ICW4_8086       0x01            /* 8086/88 (MCS-80/85) mode */
#define ICW4_AUTO       0x02            /* Auto (normal) EOI */
#define ICW4_BUF_SLAVE  0x08            /* Buffered mode/slave */
#define ICW4_BUF_MASTER 0x0C            /* Buffered mode/master */
#define ICW4_SFNM       0x10            /* Special fully nested (not) */

/*
arguments:
        offset1 - vector offset for master PIC
                vectors on the master become offset1..offset1+7
        offset2 - same for slave PIC: offset2..offset2+7
*/
void PIC_remap(int offset1, int offset2)
{
        unsigned char a1, a2;

        a1 = inb(PIC1_DATA);                        // save masks
        a2 = inb(PIC2_DATA);
```

```
        outb(PIC1_COMMAND, ICW1_INIT+ICW1_ICW4);  // starts the initializ
        io_wait();
        outb(PIC2_COMMAND, ICW1_INIT+ICW1_ICW4);
        io_wait();
        outb(PIC1_DATA, offset1);                 // ICW2: Master PIC vec
        io_wait();
        outb(PIC2_DATA, offset2);                 // ICW2: Slave PIC vect
        io_wait();
        outb(PIC1_DATA, 4);                       // ICW3: tell Master PI
        io_wait();
        outb(PIC2_DATA, 2);                       // ICW3: tell Slave PIC
        io_wait();

        outb(PIC1_DATA, ICW4_8086);
        io_wait();
        outb(PIC2_DATA, ICW4_8086);
        io_wait();

        outb(PIC1_DATA, a1);   // restore saved masks.
        outb(PIC2_DATA, a2);
}
```

Note the presence of io_wait() calls, on older machines its necessary to give the PIC some time to react to commands as they might not be processed quickly

Disabling

If you are going to use the processor local APIC and the IOAPIC, you must first disable the PIC. This is done via:

```
mov al, 0xff
out 0xa1, al
out 0x21, al
```

Masking

The PIC has an internal register called the IMR, or the Interrupt Mask Register. It is 8 bits wide. This register is a bitmap of the request lines going into the PIC. When a bit is set, the PIC ignores the request and continues normal operation. Note that setting the mask on a higher request line will not affect a lower line. Here is an example of how to do this:

```
void IRQ_set_mask(unsigned char IRQline) {
    uint16_t port;
    uint8_t value;

    if(IRQline < 8) {
```

```
        port = PIC1_DATA;
    } else {
        port = PIC2_DATA;
        IRQline -= 8;
    }
    value = inb(port) | (1 << IRQline);
    outb(port, value);
}

void IRQ_clear_mask(unsigned char IRQline) {
    uint16_t port;
    uint8_t value;

    if(IRQline < 8) {
        port = PIC1_DATA;
    } else {
        port = PIC2_DATA;
        IRQline -= 8;
    }
    value = inb(port) & ~(1 << IRQline);
    outb(port, value);
}
```

ISR and IRR

The PIC chip has two interrupt status registers: the In-Service Register (ISR) and the Interrupt Request Register (IRR). The ISR tells us which interrupts are being serviced, meaning IRQs sent to the CPU. The IRR tells us which interrupts have been raised. Based on the interrupt mask (IMR), the PIC will send interrupts from the IRR to the CPU, at which point they are marked in the ISR.

The ISR and IRR can be read via the OCW3 command word. This is a command sent to one of the command ports (0x20 or 0xa0) with bit 3 set. To read the ISR or IRR, write the appropriate command to the command port, and then read the command port (not the data port). To read the IRR, write 0x0a. To read the ISR, write 0x0b.

The ISR and IRR are each 8 bits. Here is an example of how to read 16 bits worth of ISR and IRR data from two cascaded PICs:

```
#define PIC1_CMD                    0x20
#define PIC1_DATA                   0x21
#define PIC2_CMD                    0xA0
#define PIC2_DATA                   0xA1
#define PIC_READ_IRR                0x0a    /* OCW3 irq ready next CMD re
#define PIC_READ_ISR                0x0b    /* OCW3 irq service next CMD

/* Helper func */
static uint16_t __pic_get_irq_reg(int ocw3)
{
    /* OCW3 to PIC CMD to get the register values.  PIC2 is chained, and
```

```
     * represents IRQs 8-15.  PIC1 is IRQs 0-7, with 2 being the chain */
    outb(PIC1_CMD, ocw3);
    outb(PIC2_CMD, ocw3);
    return (inb(PIC2_CMD) << 8) | inb(PIC1_CMD);
}

/* Returns the combined value of the cascaded PICs irq request register */
uint16_t pic_get_irr(void)
{
    return __pic_get_irq_reg(PIC_READ_IRR);
}

/* Returns the combined value of the cascaded PICs in-service register */
uint16_t pic_get_isr(void)
{
    return __pic_get_irq_reg(PIC_READ_ISR);
}
```

Note that these functions will show bit 2 (0x0004) as on whenever any of the PIC2 bits are set, due to the chained nature of the PICs. Also note that it is not necessary to reset the OCW3 command every time you want to read. Once you set it for either the IRR or the ISR, future reads of the CMD port will return the appropriate register. The chip remembers what OCW3 setting you used. (Disclaimer: I have not tested this last part, but that's what the spec says.)

Spurious IRQs

When an IRQ occurs, the PIC chip tells the CPU (via. the PIC's INTR line) that there's an interrupt, and the CPU acknowledges this and waits for the PIC to send the interrupt vector. This creates a race condition: if the IRQ disappears after the PIC has told the CPU there's an interrupt but before the PIC has sent the interrupt vector to the CPU, then the CPU will be waiting for the PIC to tell it which interrupt vector but the PIC won't have a valid interrupt vector to tell the CPU.

To get around this, the PIC tells the CPU a fake interrupt number. This is a spurious IRQ. The fake interrupt number is the lowest priority interrupt number for the corresponding PIC chip (IRQ 7 for the master PIC, and IRQ 15 for the slave PIC).

There are several reasons for the interrupt to disappear. In my experience the most common reason is software sending an EOI at the wrong time. Other reasons include noise on IRQ lines (or the INTR line).

Handling Spurious IRQs

For a spurious IRQ, there is no real IRQ and the PIC chip's ISR (In Service Register) flag for the corresponding IRQ will not be set. This means that the interrupt handler must not send an EOI back to the PIC to reset the ISR flag.

The correct way to handle an IRQ 7 is to first check the master PIC chip's ISR to see if the IRQ is a spurious IRQ or a real IRQ. If it is a real IRQ then it is treated the same as any other real IRQ. If it is a spurious IRQ then you ignore it (and do not send the EOI).

The correct way to handle an IRQ 15 is similar, but a little trickier due to the interaction between the slave PIC and the master PIC. First check the slave PIC chip's ISR to see if the IRQ is a spurious IRQ or a real IRQ. If it is a real IRQ then it is treated the same as any other real IRQ. If it's a spurious IRQ then don't send the EOI to the slave PIC; however you will still need to send the EOI to the master PIC because the master PIC itself won't know that it was a spurious IRQ from the slave.

Also note that some operating systems (e.g. Linux) keep track of the number of spurious IRQs that have occurred (e.g. by incrementing a counter when a spurious IRQ occurs). This can be useful for detecting problems in software (e.g. sending EOIs at the wrong time) and detecting problems in hardware (e.g. line noise).

See Also

Articles

- APIC
- IOAPIC

Threads

External Links

- Intel Datasheet (http://pdos.csail.mit.edu/6.828/2005/readings/hardware/8259A.pdf)
- http://www.brokenthorn.com/Resources/OSDevPic.html