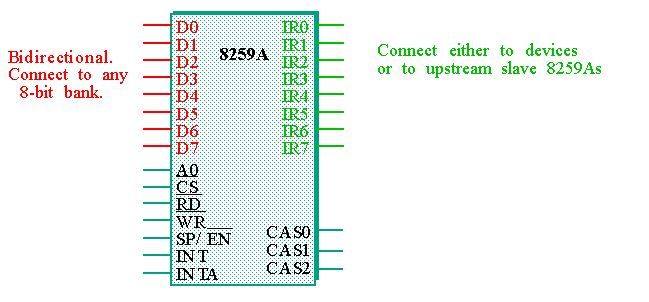
第七章

上一章我们用很大的篇幅讨论了中断的实现，idt的构造，并在entry做了一个中断的演示。这一章我们以上一章内容为基础，来实现一下8259 PIC的硬件中断。

8259微控制器系列是一组可编程中断控制器(PIC)集成电路(IC)。 由于电路限制，一个PIC仅支持8个中断，这个数量实在太少了，但所幸PIC可以通过级联的方式连接。大多数电脑都会有两个PIC(这里说的都是早期的电脑，现在的电脑，特别是多核处理器的电脑，都采用了APIC)，一个PIC在处理器内部，另一个在主板上。

PIC的内容还是蛮多的，讲起了也比较复杂，但这一内容对我们了解计算机的用处也并不是很大，所以我尽量简单说。而这部分内容如果不感兴趣也可以凭个人喜好跳过，不影响我们的实现。如果有不明白的或者感兴趣的，可以查阅附列的参考资料，或者自行谷歌。

用一个简图来看一下双列直插封装的8259A构造：



这里除了GND(接地)和VCC(输入电压)以外，所有的引脚都被标注出来了。

我们来简单介绍下各个引脚：

WR引脚：该引脚连接写入选通信号(Pentium上的8个)

RD引脚：连接到IOCR(输入输出控制例程)信号。

INT引脚：连接到微处理器的INTR引脚。

INTA引脚：连接到微处理器上的INTA引脚。

A0引脚：选择不同的命令字

CS引脚：使芯片进行编程和控制。

SP / EN引脚：从器程序(SP)/使能缓冲器(EN)。

从机程序(1 =主机，0 =从机)

启用缓冲区(缓冲模式时控制数据总线传输)

CAS0，CAS1，CAS2引脚：用于在级联系统中从主器件输出到从属PIC控制器。

D0 - D7引脚：8位数据连接器引脚。相当于一个小的数据总线，外部控制器通过这个来与PIC传递数据。

关于中断，也分好几种模式，这些模式涉及到与PIC连接的问题，这里说一下：

* 电平触发中断(level triggered interrupt)，当PIC上的IR线是高电位时触发，并且直到该中断完成才停止。电平触发中断线可以由多个中断共享，这一设计是首选的模式，因为多个设备可以共享一根线。但有一个问题就在于如果有一个需要维护的优先级较高的中断，所有其他中断将被阻塞。
* 边缘触发中断(Edge Triggered interrupts)，同样在IR线为高电位时触发，但这种情况设备只发送一个脉冲信号，而不是一直维持在高电位状态。边缘触发中断的线同样可以由多个设备共享，并且不存在一个触发阻塞其他中断的情况，但这一模式的问题也显而易见：可能会错过一些关键的中断。这可能会造成CPU锁死的情况。
* 混合模式(hybrid)，由于以上两种模式都有各自的问题，因此大多数系统采用二者混合的模式。通过CPU上的NMI(不可屏蔽中断)引脚来监视它们的活动，防止错过系统的一些重大的问题。
* Message Signaled，这种模式就是之前提到的通过数据线来传送中断的方式，典型的就是PCI线。

关于中断，CPU上有三个引脚，分别是INTR，NMI和INTA。INTR和NMI都是接收中断请求的引脚，而INTA则是中断确认引脚。PIC的INT引脚连接到处理器的INTR引脚，PIC的INTA和处理器的INTA相连。

说完这些引脚后，我们详细说说一个硬件中断执行的全过程：

1. 中断产生

当设备控制器需要产生中断时，需要以某种方式向PIC发送信号。比如说，这个设备是使用中断线0的定时器。

* 定时器控制器通过激活IR0线来通知PIC。这将其状态从0(无电源)改为1(电源正在通过线路)。
* PIC将中断请求寄存器(Interrupt Request Register，IRR)内部的表示IRQ的位置设为1。
* PIC检查中断屏蔽寄存器(Interrupt Mask Register，IMR)以查看中断是否可以被服务。如果中断可以被服务，则PIC确定是否有等待被服务的优先级更高的中断。如果存在，中断请求将被忽略，直到更高优先级的中断被服务为止。如果可以处理中断，并且没有更高优先级的中断，则PIC继续进行下一步。
* PIC通过INTA引脚向处理器通知处理器中断已被触发。

处理器现在知道中断已被触发。

1. 处理器确认中断

* CPU完成当前指令的执行。
* CPU检查RFLAGS中的中断标志(IF)。如果IF置1，CPU通过INTR引脚将中断请求确认回PIC。如果IF被清除，则忽略中断请求。
* PIC通过INTR接收到识别信号。
* PIC将中断向量号放入D0-D7引脚。在PIC初始化期间，该中断向量号从初始化控制字(ICW)2位获得。
* PIC也将IRQ号码放入D0-D7
* PIC在In Service Register(ISR)中设置正确的位。在这种情况下，是0，表示中断0当前被服务。

现在处理器具有要执行的IRQ号和中断向量号。

1. 中断

* 处理器中断当前进程。 它在堆栈上push EFLAGS，CS和EIP。
* 处理器使用中断向量号(由PIC给出)。在实模式下，CPU在IVT中偏移。 在保护模式下，处理器在IDT中偏移
* 实模式：CPU偏移到IVT的入口，将中断的基地址加载到CS：IP中，开始中断的具体处理。
* 保护模式：CPU使用加载的IDT，把选择子的gate描述符加载到CS段选择器中。gate描述符的偏移量被加载到EIP中。如果启用了分页，则该地址将从线性地址转换为物理地址。现在，CPU将对当前状态进行架构特定的安全检查。中断例程现在可以从gate描述符+CS：EIP地址处控制。

1. 中断服务程序

* 现在ISR(中断服务程序)正在执行以处理硬件中断。 它可以执行为特定设备提供服务所需的任何操作。 例如，从设备读取或写入数据，读取状态寄存器，发送命令等。
* 在此期间，所有中断由中断屏蔽寄存器(IMR)屏蔽。 换句话说，这将禁用所有硬件中断，直到结束中断，结束中断这需要将中断结束(End of interrupt，EOI)命令发送到PIC。
* 在通过主PIC命令寄存器将EOI信号发送到PIC之后，PIC清除了在在服务寄存器(IRR)中的对应的位，并且现在可以为新的中断服务了。
* 然后中断服务程序执行IRETD指令，弹出EFLAGS，CS和EIP寄存器
* 最后转回一开始的任务中。

上面提到了很多8259的寄存器，所以有必要对这些寄存器也做一个简单的介绍。8259的寄存器都是内部寄存器，包括命令寄存器(Command Register)，状态寄存器(Status register)，中断请求寄存器(Interrupt Request Register,IRR)，服务寄存器(In-Sevice Register,ISR)，中断屏蔽寄存器(Interrupt Mask Register,IMR)。

* Command Register：这是一个只写寄存器，用于向微控制器发送命令。 有很多不同的命令可以发送。 某些命令用于从其他寄存器读取，而其他命令用于初始化和发送数据，例如中断结束(EOI)。
* Status register：这是一个只读寄存器，可以访问以确定PIC的状态。
* Interrupt Request Register (IRR)：该寄存器指定哪些中断正在等待确认：
* In-Sevice Register (ISR)：该寄存器指定哪些中断已被确认，等待中断结束(EOI)信号， EOI信号非常重要，因为它决定了中断的结束。
* 注意：中断完成后，我们需要发送EOI信号，让8259A确认中断。 否则会导致未定义的行为或故障。
* Interrupt Mask Register (IMR)：这个寄存器指定了什么中断被忽略，这允许我们在执行此寄存器指定的中断之前，专注于执行某些更重要的中断。

这些寄存器都是8位寄存器，其中每个位确定中断是否被禁止。 如果该位为0，则它被使能。 如果为1，则中断设备被禁用。

以上几个寄存器的位数设置是相同的，所以这里我只列一个表了：

|  |  |  |
| --- | --- | --- |
| **Interrupt Mask Register (IMR)/ In Service Register (ISR)/ Interrupt Request Register (IRR)** | | |
| Bit Number | IRQ Number (主控制器) | IRQ Number (从控制器) |
| 0 | IRQ0 | IRQ8 |
| 1 | IRQ1 | IRQ9 |
| 2 | IRQ2 | IRQ10 |
| 3 | IRQ3 | IRQ11 |
| 4 | IRQ4 | IRQ12 |
| 5 | IRQ5 | IRQ13 |
| 6 | IRQ6 | IRQ14 |
| 7 | IRQ7 | IRQ15 |

说完了寄存器，再来说说8259的软件端口映射，这个才是我们的实际上要操作的端口，和之前操作一样，也是通过IN和OUT来对端口操作。

|  |  |
| --- | --- |
| **8259A Software Port Map** | |
| Port Address | Description |
| 0x20 | 主PIC命令和状态寄存器 |
| 0x21 | 主PIC中断屏蔽寄存器和数据寄存器 |
| 0xA0 | 从PIC命令和状态寄存器 |
| 0xA1 | 从PIC中断屏蔽寄存器和数据寄存器 |

有了端口之后，我们就可以通过端口来发送我们的控制命令了，8259A的控制命令分为初始化命令和操作命令：

**初始控制字(Initialization Control Words，ICW)**，初始化PIC的目的在于重新映射PIC的中断号，这可以保证发生中断能正常产生。ICW就是我们的初始化命令，这里有四个初始化命令，分别是ICW1，ICW2，ICW3和ICW4。注意如果有多个PIC级联的话，我们对每个PIC都需要设置。

**ICW1**：这个是PIC初始化中主要的控制字，格式如下：

|  |  |  |
| --- | --- | --- |
| **Initialization Control Word (ICW)** | | |
| Bit Number | Value | Description |
| 0 | IC4 | 如果设置为1，PIC在初始化期间会接收IC4。 |
| 1 | SNGL | 如果设置1，系统中只有一个PIC。 如果清零，则PIC与从属PIC级联，同时ICW3必须发送到控制器。 |
| 2 | ADI | 如果设置为1，CALL地址间隔为4，否则8.这个在x86下一般被忽略，默认为0 |
| 3 | LTIM | 如果设置为1，则在电平触发模式下操作。 如果设置为0，则以边沿触发模式运行 |
| 4 | 1 | 初始化位。 如果要初始化PIC，则置1 |
| 5 | 0 | MCS-80/85：中断向量地址。 x86架构：必须为0 |
| 6 | 0 | MCS-80/85：中断向量地址。 x86架构：必须为0 |
| 7 | 0 | MCS-80/85：中断向量地址。 x86架构：必须为0 |

一般设置值如下：

位0 - 设置为1，所以我们可以发送ICW 4

位1 - PIC级联位。 x86架构有2个PIC，因此我们需要主从PIC与从机级联。 保持0

位2 - 呼叫地址间隔。 被x86忽略并保持在8，所以保持0

位3 - 边沿触发/电平触发模式位。 默认情况下，我们处于边缘触发，所以将其保留为0

位4 - 初始化位。 设为1

位5 ... 7 - 在x86上未使用，设置为0。

所以应该是0x11，即0001 0001，指令为outb(0x20,0x11)和outb(0xA0,0x11)

**ICW2**：这个指令用来映射PIC所使用的IVT的基址。格式如下：

|  |  |  |
| --- | --- | --- |
| **Initialization Control Word (ICW) 2** | | |
| Bit Number | Value | Description |
| 0-2 | A8/A9/A10 | 在MCS-80/85模式下，用于IVT的地址位A8-A10。 |
| 3-7 | A11(T3)/A12(T4)/A13  (T5)/A14(T6)/A15(T7) | 在MCS-80/85模式下，用于IVT的地址位A11-A15。 在80x86模式下，指定中断向量地址。 可能在x86模式下设置为0。 |

一旦发送了ICW1指令，ICW2指令也必须使用。注意当前的前31个中断0x0到0x1F都被使用了(在第六章里面有IVT的表)，所以我们从32号(0x20)IRQ开始，给主片PIC分配，从片从40号(0x28)开始分配。

所以指令如下：outb(0x21, 0x20); outb(0xA1, 0x28);

**ICW3**：它用于设定PIC之间连接的的IRQ线路，以便相互之间的通讯。

|  |  |  |
| --- | --- | --- |
| **Initialization Control Word (ICW) 3 - Primary PIC** | | |
| Bit Number | Value | Description |
| 0-7 | S0-S7 | 指定什么中断请求(IRQ)连接到从属PIC |

|  |  |  |
| --- | --- | --- |
| **Initialization Control Word (ICW) 3 - Secondary PIC** | | |
| Bit Number | Value | Description |
| 0-2 | ID0 | 连接到主PIC使用的IRQ号码(以二进制符号表示) |
| 3-7 | 0 | 保留，必须为0 |

一旦ICW1开启了级联的话，ICW3也是必须设置的。PIC使用级联端口来相互之间通信，而使用IR0-IR7相互连接，中断编号对应了位数，就不列表格了。注意这里**主片的设置是每一位对应一个中断。而从片是用3位来表示8个中断**。我们这里使用主片的IR2连到从片：outb(0x21, 0x04);(0x04二进制是0100，也就是第二位被设置，对应IR2)，从片也要设置：outb(0xA1, 0x02);(这里从片的0x02表示IR2，也就是010，从片是从000到111计数)

**ICW4**：这个用来控制操作的方式，ICW第0位设定了的话，这里也要设定。

|  |  |  |
| --- | --- | --- |
| **Initialization Control Word (ICW) 4** | | |
| Bit Number | Value | Description |
| 0 | uPM | 如果设置为1，则为80x86模式。 如果MCS-80/86模式清零 |
| 1 | AEOI | 如果置位，在最后一个中断确认脉冲时，控制器自动执行中断结束(EOI)操作 |
| 2 | M/S | 仅在BUF设置时才使用。 如果设置为1，选择主片缓冲。设置0则是从片缓冲。 |
| 3 | BUF | 如果设置，控制器以缓冲模式运行 |
| 4 | SFNM | 特殊的完全嵌套模式(Fully Nested Mode)。 用于具有大量级联控制器的系统。 |
| 5-7 | 0 | 保留，必须为0 |

在我们这里只需要设置第0位就行，也就是outb(0x21, 0x01);outb(0xA1, 0x01);

到了这里，我们的PIC终于初始化好了。接下来就可以对它进行操作啦。所以下面看一下我们的操作命令指令。

操作命令指令(Operation Command Words，OCW)：用来操作和控制PIC。有两个指令，OCW1和OCW2。

OCW1：OCW 1表示中断屏蔽寄存器(IMR)内部的值。 要获得当前的OCW 1，从IMR读取就行。同时，IMR和状态寄存器映射所在是同一端口。 因为状态寄存器是只读的，所以PIC可以根据这个操作是读还是写来判断要操作的寄存器。

OCW2：这个是用来控制PIC的主要控制字。格式如下：

|  |  |  |
| --- | --- | --- |
| **Operation Command Word (OCW) 2** | | |
| Bit Number | Value | Description |
| 0-2 | L0/L1/L2 | 控制器必须做出反应的中断级别 |
| 3-4 | 0 | 保留，必须为0 |
| 5 | EOI | 中断结束(EOI)请求 |
| 6 | SL | 选择 |
| 7 | R | 旋转选项 |

具体5-7位如何使用的格式更为复杂，这里实在没必要多介绍。仅仅说两个：

000：以自动EOI模式更新(清零)，也就是开始接受中断。

001：非特定EOI命令，用来结束当前中断。

说了这么多了，还没具体给出我们的代码呢。首先在idt.c里面增加了关于8259A PIC的初始化，就是我们上面讨论的内容。大部分内容和上一章是完全一样的，仅仅增加了部分内容(但我还是列出全部内容好了)：

/idt/idt.c

1. #include "common.h"
2. #include "string.h"
3. #include "print.h"
4. #include "idt.h"
5. // 中断描述符表
6. i\_descriptor idt[256];
7. // IDTR
8. idtr idt\_ptr;
9. // 中断处理函数的指针数组
10. interrupt\_handler\_t interrupt\_handlers[256];
11. // 设置中断描述符
12. static void idt\_set\_gate(uint8\_t num, uint32\_t base, uint16\_t sel, uint8\_t flags);
13. // 声明加载 IDTR 的函数
14. extern void idt\_flush(uint32\_t);
15. //声明一个测试函数
16. static void interupt255\_test(pt\_regs \*regs);
17. // 初始化中断描述符表
18. void init\_idt()
19. {
20. // 重新映射 IRQ 表
21. // 两片级联的 Intel 8259A 芯片
22. // 主片端口 0x20 0x21
23. // 从片端口 0xA0 0xA1
25. // 初始化主片、从片
26. // 0001 0001
27. outb(0x20, 0x11);
28. outb(0xA0, 0x11);
29. // 设置主片 IRQ 从 0x20(32) 号中断开始
30. outb(0x21, 0x20);
31. // 设置从片 IRQ 从 0x28(40) 号中断开始
32. outb(0xA1, 0x28);
34. // 设置主片 IR2 引脚连接从片
35. outb(0x21, 0x04);
36. // 告诉从片输出引脚和主片 IR2 号相连
37. outb(0xA1, 0x02);
39. // 设置主片和从片按照 8086 的方式工作
40. outb(0x21, 0x01);
41. outb(0xA1, 0x01);
43. // 设置主从片允许中断
44. outb(0x21, 0x0);
45. outb(0xA1, 0x0);
46. bzero((uint8\_t \*)&interrupt\_handlers, sizeof(interrupt\_handler\_t) \* 256);
48. idt\_ptr.limit = sizeof(i\_descriptor) \* 256 - 1;
49. idt\_ptr.base = (uint32\_t)&idt;
51. bzero((uint8\_t \*)&idt, sizeof(i\_descriptor) \* 256);
52. // 0-32: 用于 CPU 的中断处理
53. idt\_set\_gate( 0, (uint32\_t)isr0, 0x08, 0x8E);
54. idt\_set\_gate( 1, (uint32\_t)isr1, 0x08, 0x8E);
55. idt\_set\_gate( 2, (uint32\_t)isr2, 0x08, 0x8E);
56. idt\_set\_gate( 3, (uint32\_t)isr3, 0x08, 0x8E);
57. idt\_set\_gate( 4, (uint32\_t)isr4, 0x08, 0x8E);
58. idt\_set\_gate( 5, (uint32\_t)isr5, 0x08, 0x8E);
59. idt\_set\_gate( 6, (uint32\_t)isr6, 0x08, 0x8E);
60. idt\_set\_gate( 7, (uint32\_t)isr7, 0x08, 0x8E);
61. idt\_set\_gate( 8, (uint32\_t)isr8, 0x08, 0x8E);
62. idt\_set\_gate( 9, (uint32\_t)isr9, 0x08, 0x8E);
63. idt\_set\_gate(10, (uint32\_t)isr10, 0x08, 0x8E);
64. idt\_set\_gate(11, (uint32\_t)isr11, 0x08, 0x8E);
65. idt\_set\_gate(12, (uint32\_t)isr12, 0x08, 0x8E);
66. idt\_set\_gate(13, (uint32\_t)isr13, 0x08, 0x8E);
67. idt\_set\_gate(14, (uint32\_t)isr14, 0x08, 0x8E);
68. idt\_set\_gate(15, (uint32\_t)isr15, 0x08, 0x8E);
69. idt\_set\_gate(16, (uint32\_t)isr16, 0x08, 0x8E);
70. idt\_set\_gate(17, (uint32\_t)isr17, 0x08, 0x8E);
71. idt\_set\_gate(18, (uint32\_t)isr18, 0x08, 0x8E);
72. idt\_set\_gate(19, (uint32\_t)isr19, 0x08, 0x8E);
73. idt\_set\_gate(20, (uint32\_t)isr20, 0x08, 0x8E);
74. idt\_set\_gate(21, (uint32\_t)isr21, 0x08, 0x8E);
75. idt\_set\_gate(22, (uint32\_t)isr22, 0x08, 0x8E);
76. idt\_set\_gate(23, (uint32\_t)isr23, 0x08, 0x8E);
77. idt\_set\_gate(24, (uint32\_t)isr24, 0x08, 0x8E);
78. idt\_set\_gate(25, (uint32\_t)isr25, 0x08, 0x8E);
79. idt\_set\_gate(26, (uint32\_t)isr26, 0x08, 0x8E);
80. idt\_set\_gate(27, (uint32\_t)isr27, 0x08, 0x8E);
81. idt\_set\_gate(28, (uint32\_t)isr28, 0x08, 0x8E);
82. idt\_set\_gate(29, (uint32\_t)isr29, 0x08, 0x8E);
83. idt\_set\_gate(30, (uint32\_t)isr30, 0x08, 0x8E);
84. idt\_set\_gate(31, (uint32\_t)isr31, 0x08, 0x8E);
85. idt\_set\_gate(32, (uint32\_t)irq0, 0x08, 0x8E);
86. idt\_set\_gate(33, (uint32\_t)irq1, 0x08, 0x8E);
87. idt\_set\_gate(34, (uint32\_t)irq2, 0x08, 0x8E);
88. idt\_set\_gate(35, (uint32\_t)irq3, 0x08, 0x8E);
89. idt\_set\_gate(36, (uint32\_t)irq4, 0x08, 0x8E);
90. idt\_set\_gate(37, (uint32\_t)irq5, 0x08, 0x8E);
91. idt\_set\_gate(38, (uint32\_t)irq6, 0x08, 0x8E);
92. idt\_set\_gate(39, (uint32\_t)irq7, 0x08, 0x8E);
93. idt\_set\_gate(40, (uint32\_t)irq8, 0x08, 0x8E);
94. idt\_set\_gate(41, (uint32\_t)irq9, 0x08, 0x8E);
95. idt\_set\_gate(42, (uint32\_t)irq10, 0x08, 0x8E);
96. idt\_set\_gate(43, (uint32\_t)irq11, 0x08, 0x8E);
97. idt\_set\_gate(44, (uint32\_t)irq12, 0x08, 0x8E);
98. idt\_set\_gate(45, (uint32\_t)irq13, 0x08, 0x8E);
99. idt\_set\_gate(46, (uint32\_t)irq14, 0x08, 0x8E);
100. idt\_set\_gate(47, (uint32\_t)irq15, 0x08, 0x8E);
101. // 255 将来用于实现系统调用
102. idt\_set\_gate(255, (uint32\_t)isr255, 0x08, 0x8E);
103. //注册测试函数到我们的中断中
104. register\_interrupt\_handler(255,interupt255\_test);
105. // 更新设置中断描述符表
106. idt\_flush((uint32\_t)&idt\_ptr);
107. }
108. // 设置中断描述符
109. static void idt\_set\_gate(uint8\_t num, uint32\_t base, uint16\_t sel, uint8\_t flags)
110. {
111. idt[num].base\_lo = base & 0xFFFF;
112. idt[num].base\_hi = (base >> 16) & 0xFFFF;
113. idt[num].sel = sel;
114. idt[num].always0 = 0;
115. // 先留下 0x60 这个魔数，以后实现用户态时候
116. // 这个与运算可以设置中断门的特权级别为 3
117. idt[num].flags = flags; // | 0x60
118. }
119. // 调用中断处理函数
120. void isr\_handler(pt\_regs \*regs)
121. {
122. if (interrupt\_handlers[regs->int\_no]) {
123. interrupt\_handlers[regs->int\_no](regs);
124. } else {
125. printk("Unhandled interrupt: %d\n", regs->int\_no);
126. }
127. }
128. // 注册一个中断处理函数
129. void register\_interrupt\_handler(uint8\_t n, interrupt\_handler\_t h)
130. {
131. interrupt\_handlers[n] = h;
132. }
133. //测试函数的具体内容
134. void interupt255\_test(pt\_regs \*regs){
135. printk("the interrupt test for IRQ\_255\n");
136. }
137. // IRQ 处理函数
138. void irq\_handler(pt\_regs \*regs)
139. {
140. // 发送中断结束信号给 PICs
141. // 按照我们的设置，从 32 号中断起为用户自定义中断
142. // 因为单片的 Intel 8259A 芯片只能处理 8 级中断
143. // 故大于等于 40 的中断号是由从片处理的
144. if (regs->int\_no >= 40) {
145. // 发送重设信号给从片
146. outb(0xA0, 0x20);
147. }
148. // 发送重设信号给主片
149. outb(0x20, 0x20);
150. if (interrupt\_handlers[regs->int\_no]) {
151. interrupt\_handlers[regs->int\_no](regs);
152. }else {
153. printk("Unhandled hardware interrupt: %d\n", regs->int\_no);
154. }
155. }

可以看到，我们在初始化中断描述符表里面增加了关于PIC的初始化代码，结合注释和上面讲的内容，应该可以看懂吧。此外增加了对于新的中断的设置，以及中断处理的函数irq\_handler()

看一下我们在idt.h里面新增的中断吧(同样给出了全部代码啦)：

/include/idt.h

1. #ifndef INCLUDE\_IDT\_H\_
2. #define INCLUDE\_IDT\_H\_
3. #include "types.h"
4. // 初始化中断描述符表
5. void init\_idt();
6. // 中断描述符
7. typedef
8. struct i\_descriptor {
9. uint16\_t base\_lo; // 中断处理函数地址 15～0 位
10. uint16\_t sel; // 目标代码段描述符选择子
11. uint8\_t always0; // 置 0 段
12. uint8\_t flags; // 一些标志，文档有解释
13. uint16\_t base\_hi; // 中断处理函数地址 31～16 位
14. }\_\_attribute\_\_((packed)) i\_descriptor;
15. // IDTR
16. typedef
17. struct idtr{
18. uint16\_t limit; // 限长
19. uint32\_t base; // 基址
20. } \_\_attribute\_\_((packed)) idtr;
21. // 寄存器类型
22. typedef
23. struct pt\_regs {
24. uint32\_t ds; // 用于保存用户的数据段描述符
25. uint32\_t edi; // 从 edi 到 eax 由 pusha 指令压入
26. uint32\_t esi;
27. uint32\_t ebp;
28. uint32\_t esp;
29. uint32\_t ebx;
30. uint32\_t edx;
31. uint32\_t ecx;
32. uint32\_t eax;
33. uint32\_t int\_no; // 中断号
34. uint32\_t err\_code; // 错误代码(有中断错误代码的中断会由CPU压入)
35. uint32\_t eip; // 以下由处理器自动压入
36. uint32\_t cs;
37. uint32\_t eflags;
38. uint32\_t useresp;
39. uint32\_t ss;
40. } pt\_regs;
41. // 定义中断处理函数指针
42. typedef void (\*interrupt\_handler\_t)(pt\_regs \*);
43. // 注册一个中断处理函数
44. void register\_interrupt\_handler(uint8\_t n, interrupt\_handler\_t h);
45. // 调用中断处理函数
46. void isr\_handler(pt\_regs \*regs);
47. // 声明中断处理函数 0-19 属于 CPU 的异常中断
48. // ISR:中断服务程序(interrupt service routine)
49. void isr0(); // 0 #DE 除 0 异常
50. void isr1(); // 1 #DB 调试异常
51. void isr2(); // 2 NMI
52. void isr3(); // 3 BP 断点异常
53. void isr4(); // 4 #OF 溢出
54. void isr5(); // 5 #BR 对数组的引用超出边界
55. void isr6(); // 6 #UD 无效或未定义的操作码
56. void isr7(); // 7 #NM 设备不可用(无数学协处理器)
57. void isr8(); // 8 #DF 双重故障(有错误代码)
58. void isr9(); // 9 协处理器跨段操作
59. void isr10(); // 10 #TS 无效TSS(有错误代码)
60. void isr11(); // 11 #NP 段不存在(有错误代码)
61. void isr12(); // 12 #SS 栈错误(有错误代码)
62. void isr13(); // 13 #GP 常规保护(有错误代码)
63. void isr14(); // 14 #PF 页故障(有错误代码)
64. void isr15(); // 15 CPU 保留
65. void isr16(); // 16 #MF 浮点处理单元错误
66. void isr17(); // 17 #AC 对齐检查
67. void isr18(); // 18 #MC 机器检查
68. void isr19(); // 19 #XM SIMD(单指令多数据)浮点异常
69. // 20-31 Intel 保留
70. void isr20();
71. void isr21();
72. void isr22();
73. void isr23();
74. void isr24();
75. void isr25();
76. void isr26();
77. void isr27();
78. void isr28();
79. void isr29();
80. void isr30();
81. void isr31();
82. // 32～255 用户自定义异常
83. void isr255();
84. // IRQ 处理函数
85. void irq\_handler(pt\_regs \*regs);
86. // 定义IRQ
87. #define IRQ0 32 // 电脑系统计时器
88. #define IRQ1 33 // 键盘
89. #define IRQ2 34 // 与 IRQ9 相接，MPU-401 MD 使用
90. #define IRQ3 35 // 串口设备
91. #define IRQ4 36 // 串口设备
92. #define IRQ5 37 // 建议声卡使用
93. #define IRQ6 38 // 软驱传输控制使用
94. #define IRQ7 39 // 打印机传输控制使用
95. #define IRQ8 40 // 即时时钟
96. #define IRQ9 41 // 与 IRQ2 相接，可设定给其他硬件
97. #define IRQ10 42 // 建议网卡使用
98. #define IRQ11 43 // 建议 AGP 显卡使用
99. #define IRQ12 44 // 接 PS/2 鼠标，也可设定给其他硬件
100. #define IRQ13 45 // 协处理器使用
101. #define IRQ14 46 // IDE0 传输控制使用
102. #define IRQ15 47 // IDE1 传输控制使用
103. // 声明 IRQ 函数
104. // IRQ:中断请求(Interrupt Request)
105. void irq0(); // 电脑系统计时器
106. void irq1(); // 键盘
107. void irq2(); // 与 IRQ9 相接，MPU-401 MD 使用
108. void irq3(); // 串口设备
109. void irq4(); // 串口设备
110. void irq5(); // 建议声卡使用
111. void irq6(); // 软驱传输控制使用
112. void irq7(); // 打印机传输控制使用
113. void irq8(); // 即时时钟
114. void irq9(); // 与 IRQ2 相接，可设定给其他硬件
115. void irq10(); // 建议网卡使用
116. void irq11(); // 建议 AGP 显卡使用
117. void irq12(); // 接 PS/2 鼠标，也可设定给其他硬件
118. void irq13(); // 协处理器使用
119. void irq14(); // IDE0 传输控制使用
120. void irq15(); // IDE1 传输控制使用
121. #endif // INCLUDE\_IDT\_H\_

具体的函数我们还是在汇编代码里实现的：

/idt/idt\_s.s

1. [GLOBAL idt\_flush]
2. idt\_flush:
3. mov eax, [esp+4] ; 参数存入 eax 寄存器
4. lidt [eax] ; 加载到 IDTR
5. ret
6. .end:
7. ; 定义两个构造中断处理函数的宏(有的中断有错误代码，有的没有)
8. ; 用于没有错误代码的中断
9. %macro ISR\_NOERRCODE 1
10. [GLOBAL isr%1]
11. isr%1:
12. cli ; 首先关闭中断
13. push 0 ; push 无效的中断错误代码(起到占位作用，便于所有isr函数统一清栈)
14. push %1 ; push 中断号
15. jmp isr\_common\_stub
16. %endmacro
17. ; 用于有错误代码的中断
18. %macro ISR\_ERRCODE 1
19. [GLOBAL isr%1]
20. isr%1:
21. cli ; 关闭中断
22. push %1 ; push 中断号
23. jmp isr\_common\_stub
24. %endmacro
25. ; 定义中断处理函数
26. ISR\_NOERRCODE 0 ; 0 #DE 除 0 异常
27. ISR\_NOERRCODE 1 ; 1 #DB 调试异常
28. ISR\_NOERRCODE 2 ; 2 NMI
29. ISR\_NOERRCODE 3 ; 3 BP 断点异常
30. ISR\_NOERRCODE 4 ; 4 #OF 溢出
31. ISR\_NOERRCODE 5 ; 5 #BR 对数组的引用超出边界
32. ISR\_NOERRCODE 6 ; 6 #UD 无效或未定义的操作码
33. ISR\_NOERRCODE 7 ; 7 #NM 设备不可用(无数学协处理器)
34. ISR\_ERRCODE 8 ; 8 #DF 双重故障(有错误代码)
35. ISR\_NOERRCODE 9 ; 9 协处理器跨段操作
36. ISR\_ERRCODE 10 ; 10 #TS 无效TSS(有错误代码)
37. ISR\_ERRCODE 11 ; 11 #NP 段不存在(有错误代码)
38. ISR\_ERRCODE 12 ; 12 #SS 栈错误(有错误代码)
39. ISR\_ERRCODE 13 ; 13 #GP 常规保护(有错误代码)
40. ISR\_ERRCODE 14 ; 14 #PF 页故障(有错误代码)
41. ISR\_NOERRCODE 15 ; 15 CPU 保留
42. ISR\_NOERRCODE 16 ; 16 #MF 浮点处理单元错误
43. ISR\_ERRCODE 17 ; 17 #AC 对齐检查
44. ISR\_NOERRCODE 18 ; 18 #MC 机器检查
45. ISR\_NOERRCODE 19 ; 19 #XM SIMD(单指令多数据)浮点异常
46. ; 20~31 Intel 保留
47. ISR\_NOERRCODE 20
48. ISR\_NOERRCODE 21
49. ISR\_NOERRCODE 22
50. ISR\_NOERRCODE 23
51. ISR\_NOERRCODE 24
52. ISR\_NOERRCODE 25
53. ISR\_NOERRCODE 26
54. ISR\_NOERRCODE 27
55. ISR\_NOERRCODE 28
56. ISR\_NOERRCODE 29
57. ISR\_NOERRCODE 30
58. ISR\_NOERRCODE 31
59. ; 32～255 用户自定义
60. ISR\_NOERRCODE 255
61. [GLOBAL isr\_common\_stub]
62. [EXTERN isr\_handler]
63. ; 中断服务程序
64. isr\_common\_stub:
65. pusha ; Pushes edi, esi, ebp, esp, ebx, edx, ecx, eax
66. mov ax, ds
67. push eax ; 保存数据段描述符
69. mov ax, 0x10 ; 加载内核数据段描述符表
70. mov ds, ax
71. mov es, ax
72. mov fs, ax
73. mov gs, ax
74. mov ss, ax
76. push esp ; 此时的 esp 寄存器的值等价于 pt\_regs 结构体的指针
77. call isr\_handler ; 在 C 语言代码里
78. add esp, 4 ; 清除压入的参数
80. pop ebx ; 恢复原来的数据段描述符
81. mov ds, bx
82. mov es, bx
83. mov fs, bx
84. mov gs, bx
85. mov ss, bx
87. popa ; Pops edi, esi, ebp, esp, ebx, edx, ecx, eax
88. add esp, 8 ; 清理栈里的 error code 和 ISR
89. iret
90. .end:
91. ; 构造中断请求的宏
92. %macro IRQ 2
93. [GLOBAL irq%1]
94. irq%1:
95. cli
96. push 0
97. push %2
98. jmp irq\_common\_stub
99. %endmacro
100. IRQ 0, 32 ; 电脑系统计时器
101. IRQ 1, 33 ; 键盘
102. IRQ 2, 34 ; 与 IRQ9 相接，MPU-401 MD 使用
103. IRQ 3, 35 ; 串口设备
104. IRQ 4, 36 ; 串口设备
105. IRQ 5, 37 ; 建议声卡使用
106. IRQ 6, 38 ; 软驱传输控制使用
107. IRQ 7, 39 ; 打印机传输控制使用
108. IRQ 8, 40 ; 即时时钟
109. IRQ 9, 41 ; 与 IRQ2 相接，可设定给其他硬件
110. IRQ 10, 42 ; 建议网卡使用
111. IRQ 11, 43 ; 建议 AGP 显卡使用
112. IRQ 12, 44 ; 接 PS/2 鼠标，也可设定给其他硬件
113. IRQ 13, 45 ; 协处理器使用
114. IRQ 14, 46 ; IDE0 传输控制使用
115. IRQ 15, 47 ; IDE1 传输控制使用
116. [GLOBAL irq\_common\_stub]
117. [EXTERN irq\_handler]
118. irq\_common\_stub:
119. pusha ; pushes edi, esi, ebp, esp, ebx, edx, ecx, eax
121. mov ax, ds
122. push eax ; 保存数据段描述符
124. mov ax, 0x10 ; 加载内核数据段描述符
125. mov ds, ax
126. mov es, ax
127. mov fs, ax
128. mov gs, ax
129. mov ss, ax
131. push esp
132. call irq\_handler
133. add esp, 4
135. pop ebx ; 恢复原来的数据段描述符
136. mov ds, bx
137. mov es, bx
138. mov fs, bx
139. mov gs, bx
140. mov ss, bx
142. popa ; Pops edi,esi,ebp...
143. add esp, 8 ; 清理压栈的 错误代码 和 ISR 编号
144. iret ; 出栈 CS, EIP, EFLAGS, SS, ESP
145. .end:

做了这么多，我来总结总结究竟干了什么事情。上一章我们讲了中断的来龙去脉，并且实现了中断的IDT，做了中断测试，但注意，我们实现的是**软件中断**，而这一章，我们在上一章的基础上，增加了**8259芯片，实现了硬件中断**。类比上一章中断的过程，这一章中断实现的过程如下：

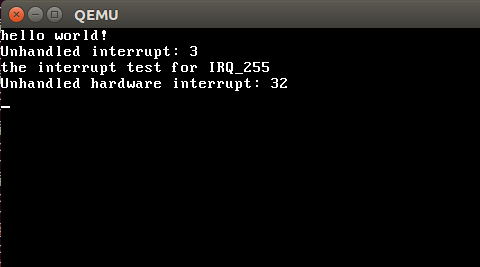
好了，解释的足够清楚了吧。下面我们同样做个小测试，看看我们的硬件中断行不行：

/entry.c

1. #include"console.h"
2. #include"print.h"
3. #include"gdt.h"
4. #include"idt.h"
5. int kern\_entry()
6. {
7. init\_gdt();
8. init\_idt();
9. console\_clear();
10. printk("hello world!\n");
11. asm volatile ("int $0x3");
12. asm volatile ("int $0xFF");
13. asm volatile ("int $0x20");
14. return 0;
15. }

相比起上一章来说，我们增加了一个硬件中断int $0x20,0x20代表32号中断。

运行结果：



嗯，not bad。还算不错。

但总是搞这种没用的中断，似乎没什么意思，我们这次来实打实地实现一个时钟中断好了。

时钟中断对于操作系统内核来说很重要的一种中断，它使得CPU无论在执行任何用户或者内核的程序时，都能定义的将执行权利交还到CPU手中来。除了记录时间之外，时钟中断的处理函数里通常都是对进程的调度处理。

具体的时钟中断源是8253/8254 Timer 产成的，要按照需要的频率产生中断，需要先配置8253/8254 Timer芯片。这又涉及到了硬件的参数，端口等内容。和我们上面讲的8259A类似，Timer芯片也有一大堆要查阅了解的端口什么的。而一来这一章以及足够多了，二来我也不厌其烦了，所以把这部分内容放在附录里好了。

直接给实现的代码：

/drivers/timer.c

1. #include "timer.h"
2. #include "print.h"
3. #include "common.h"
4. #include "idt.h"
5. void timer\_callback(pt\_regs \*regs)
6. {
7. static uint32\_t tick = 0;
8. printk("Tick: %d\n", tick++);
9. }
10. void init\_timer(uint32\_t frequency)
11. {
12. // 注册时间相关的处理函数
13. register\_interrupt\_handler(IRQ0, timer\_callback);
14. // Intel 8253/8254 PIT芯片 I/O端口地址范围是40h~43h
15. // 输入频率为 1193180，frequency 即每秒中断次数
16. uint32\_t divisor = 1193180 / frequency;//
17. // D7 D6 D5 D4 D3 D2 D1 D0
18. // 0 0 1 1 0 1 1 0
19. // 即就是 36 H
20. // 设置 8253/8254 芯片工作在模式 3 下
21. outb(0x43, 0x36);
22. // 拆分低字节和高字节
23. uint8\_t low = (uint8\_t)(divisor & 0xFF);
24. uint8\_t hign = (uint8\_t)((divisor >> 8) & 0xFF);
26. // 分别写入低字节和高字节
27. outb(0x40, low);
28. outb(0x40, hign);
29. }

以及头文件：

/include/timer.h

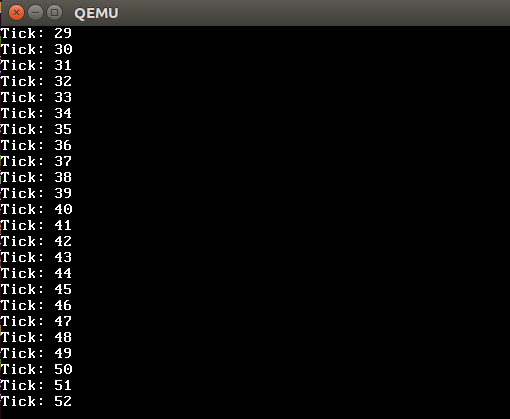
1. #ifndef INCLUDE\_TIMER\_H\_
2. #define INCLUDE\_TIMER\_H\_
3. #include "types.h"
4. void init\_timer(uint32\_t frequency);
5. #endif

把我们的文件放好后，再次修改entry.c如下：

/entry.c

1. #include"console.h"
2. #include"print.h"
3. #include"gdt.h"
4. #include"idt.h"
5. #include"timer.h"
6. int kern\_entry()
7. {
8. init\_gdt();
9. init\_idt();
10. console\_clear();
11. printk("hello world!\n");
12. asm volatile ("int $0x3");
13. asm volatile ("int $0xFF");
14. asm volatile ("int $0x20");
15. init\_timer(200);
17. //开启中断，这样我们设定好的timer就会按照时间给出中断
18. asm volatile("sti");
20. return 0;
21. }

运行结果如下：



此时我们的目录：

jf@ubuntu:~/OS\_jf$ tree

.

├── boot

│   └── boot.s

├── drivers

│   ├── console.c

│   └── timer.c

├── entry.c

├── floppy.img

├── gdt

│   ├── gdt.c

│   └── gdt\_s.s

├── idt

│   ├── idt.c

│   └── idt\_s.s

├── include

│   ├── common.h

│   ├── console.h

│   ├── gdt.h

│   ├── idt.h

│   ├── print.h

│   ├── string.h

│   ├── timer.h

│   ├── types.h

│   └── vargs.h

├── kernel

│   └── print.c

├── libs

│   ├── common.c

│   └── string.c

├── Makefile

└── scripts

├── gdbinit

└── kernel.ld

8 directories, 24 files

<http://www.brokenthorn.com/Resources/OSDevPic.html>

<http://www.brokenthorn.com/Resources/OSDevPit.html>