第 9 章　中断和动态时钟显示

在享受计算机给我们带来的便利和乐趣的同时，我仍然会时不时地说它的坏话。人们都说处理器是整个计算机的大脑，可是，处理器是一个非常精确，速度又快的傻子。

在计算机上执行的程序通常需要一些输入，输入可能来自于键盘、鼠标、硬盘、话筒、数码相机等，同时，处理后还需要输出，要送到输出设备，如显示器、硬盘、打印机、网络设备等。

一个程序只做自己的事，当它等待输入，或者等待输出时，它面对的是比处理器慢得多的外部设备。典型的情况下，硬盘的工作速度比处理器至少慢几千万甚至几亿倍，像打印机这类设备就更不用说了。在等待的时候，处理器唯一所能做的，就是不停地观察外部设备的状态变化。

计算机革命的早期，硬件资源极其昂贵和稀少。据说 20 世纪 60 年代，一台计算机的价格抵得上 300 辆野马跑车，月租金超过一万美金。这么昂贵的东西，不好好利用它就是一种罪过。

为了分享计算能力，处理器应当能够为多用户多任务提供硬件一级的支持。在单处理器的系统中，允许同时有多个程序在内存中等待处理器的执行。当一个程序正在等待输入输出时，允许另一个程序从处理器那里得到执行权。

如何把多个程序调入内存，是操作系统的事情，这个可以先放一放。现在的问题是，当一个程序执行时，它是不会知道还有别的程序正眼巴巴地等着执行。在这种情况下，中断（Interrupt）这种工作机制就应运而生了。

中断就是打断处理器当前的执行流程，去执行另外一些和当前工作不相干的指令，执行完之后，还可以返回到原来的程序流程继续执行。这就好比是你正在用手机听歌，突然来电话了。处理器（当然，手机也是有处理器的）必须中断歌曲的播放，来处理这件更为重要的事件。

自从中断这种工作机制产生之后，它就一直是各种处理器必须具备的机制。中断是怎么发生的，处理器又是怎么处理中断的，在这个过程中，我们又能做些什么，这都是本章将要告诉你的。总起来说，本章的任务是：

1. 了解中断的原理和分类，用两个具体的实例来学习如何在中断机制下工作，包括如何使用 BIOS 中断工作。
2. 学会在 Bochs 中观察中断向量表和中断标志位 IF 的变化。
3. 学习一些新的 x86 处理器指令，包括 into、int3、int *n*、iret、cli、sti、hlt、not 和 test 等。

9.1　外部硬件中断

顾名思义，外部硬件中断，就是从处理器外面来的中断信号。当外部设备发生错误，或者有数据要传送（比如，从网络中接收到一个针对当前主机的数据包），或者处理器交给它的事情处理完了（比如，打印已经完成），它们都会拍一下处理器的肩膀，告诉它应当先把手头上的事情放一放，来临时处理一下。

如图 9-1 所示，外部硬件中断是通过两个信号线引入处理器内部的。从很早的时候起，也就

是 8086 处理器的时代，这两根线的名字就叫 NMI 和 INTR。

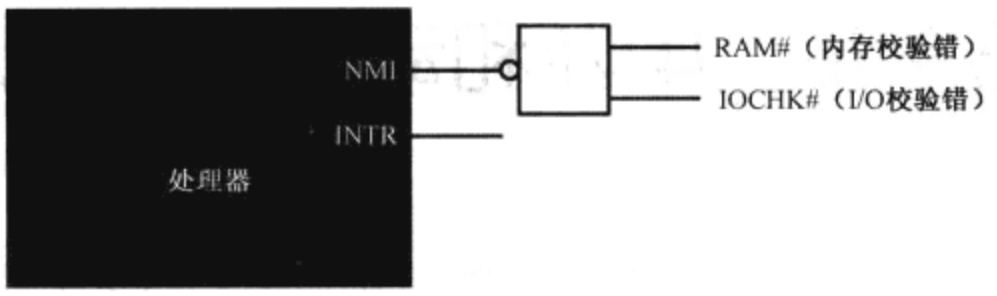

图 9-1　Intel 处理器上的不可屏蔽中断示意图

9.1.1　非屏蔽中断

在某些具有怀疑精神的人眼里，用两根信号线来接受外部设备中断可能是多余的，也许只需要一根就可以了。这似乎有此道理，但是，来自外部设备的中断很多，也不是每一个中断都是必须处理的。有些中断，在任何时候都必须及时处理，因为事关整个系统的安全性。比如，在使用不间断电源的系统中，当电池电量很低的时候，不间断电源系统会发出一个中断，通知处理器快掉电了。再比如，内存访问电路发现了一个校验错误，这意味着，从内存读取的数据是错误的，处理器再努力工作也是没有意义的。在所有这些情况下，处理器必须针对这些中断采取必要的措施，隐瞒真相必然会对用户造成不可挽回的损失。除此之外，更多的中断是可以被忽略或者延迟处理的，如果某个程序希望不被打扰的话。

在这种情况下，处理器的设计者希望通过两个引脚来明确区分不同性质的中断，这是很自然的事。首先，所有的严重事件都必须无条件地加以处理，这种类型的中断是不会被阻断和屏蔽的，称为非屏蔽中断（Non Maskable Interrupt，NMI）。

中断信号的来源，或者说，产生中断的设备，称为中断源。如图 9-1 所示，在传统的兼容模式下，NMI 的中断源通过一个与非门连接到处理器。处理器的 NMI 引脚是高电平有效的，而中断信号是低电平有效的。当不存在中断的时候，与非门的所有输入都为高，因此处理器的 NMI 引脚为低电平，这意味着没有中断发生。

当有任何一个非屏蔽的中断产生时，与非门的输出为高。Intel 处理器规定，NMI 中断信号由 0 跳变到 1 后，至少要维持 4 个以上的时钟周期才算是有效的，才能被识别。

注意，不要把这幅图当成是不变的真理，这是一个简化的示意图，不是真正的设备连接图。

当一个中断发生时，处理器将会通过中断引脚 NMI 和 INTR 得到通知。除此之外，它还应当知道发生了什么事，以便采取适当的处理措施。每种类型的中断都被统一编号，这称为中断类型号、中断向量或者中断号。但是，由于不可屏蔽中断的特殊性——几乎所有触发 NMI 的事件对处理器来说都是致命的，甚至是不可纠正的。在这种情况下，努力去搞清楚发生了什么，通常没有太大的意义，这样的事最好留到事后，让专业维修人员来做。

也正是这个原因，在实模式下，NMI 被赋予了统一的中断号 2，不再进行细分。一旦发生 2 号中断，处理器和软件系统通常会放弃继续正常工作的“念头”，也不会试图纠正已经发生的问题和错误，很可能只是由软件系统给出一个提示信息。

9.1.2　可屏蔽中断

和 NMI 不同，更多的时候，发往处理器的中断信号通常不会意味着灾难。当然，有时候也会非常紧急，比如，在一个由计算机控制的车床上，当零件快速通过铣具时，处理器应当立即处理中断，并向铣具发送信号，告诉它应当如何切削。

这类中断有两个特点，第一是数量很多，毕竟有很多外部设备；第二是它们可以被屏蔽，这样处理器就像是没听见、没看见一样，不会对它们进行处理。所以，这类硬件中断称为可屏蔽中断。尽管不处理中断就会把零件铣坏，但是否允许处理器看见该中断，是你自己的事，这是处理器赋予你的权利。

可屏蔽中断是通过 INTR 引脚进入处理器内部的，像 NMI 一样，不可能为每一个中断源都提供一个引脚。而且，处理器每次只能处理一个中断。在这种情况下，需要一个代理，来接受外部设备发出的中断信号。还有，多个设备同时发出中断请求的几率也是很高的，所以该代理的任务还包括对它们进行仲裁，以决定让它们中的哪一个优先向处理器提出服务请求。

如图 9-2 所示，在个人计算机中，用得最多的中断代理就是 8259 芯片，它就是通常所说的中断控制器，从 8086 处理器开始，它就一直提供着这种服务。即使是现在，在绝大多数单处理器的计算机中，也依然有它的存在。

Intel 处理器允许 256 个中断，中断号的范围是 0～255，8259 负责提供其中的 15 个，但中断号并不固定。之所以不固定，是因为当初设计的时候，允许软件根据自己的需要灵活设置中断号，以防止发生冲突。该中断控制器芯片有自己的端口号，可以像访问其他外部设备一样用 in 和 out 指令来改变它的状态，包括各引脚的中断号。正是因为这样，它又叫可编程中断控制器（Programmable Interrupt Controller，PIC）。

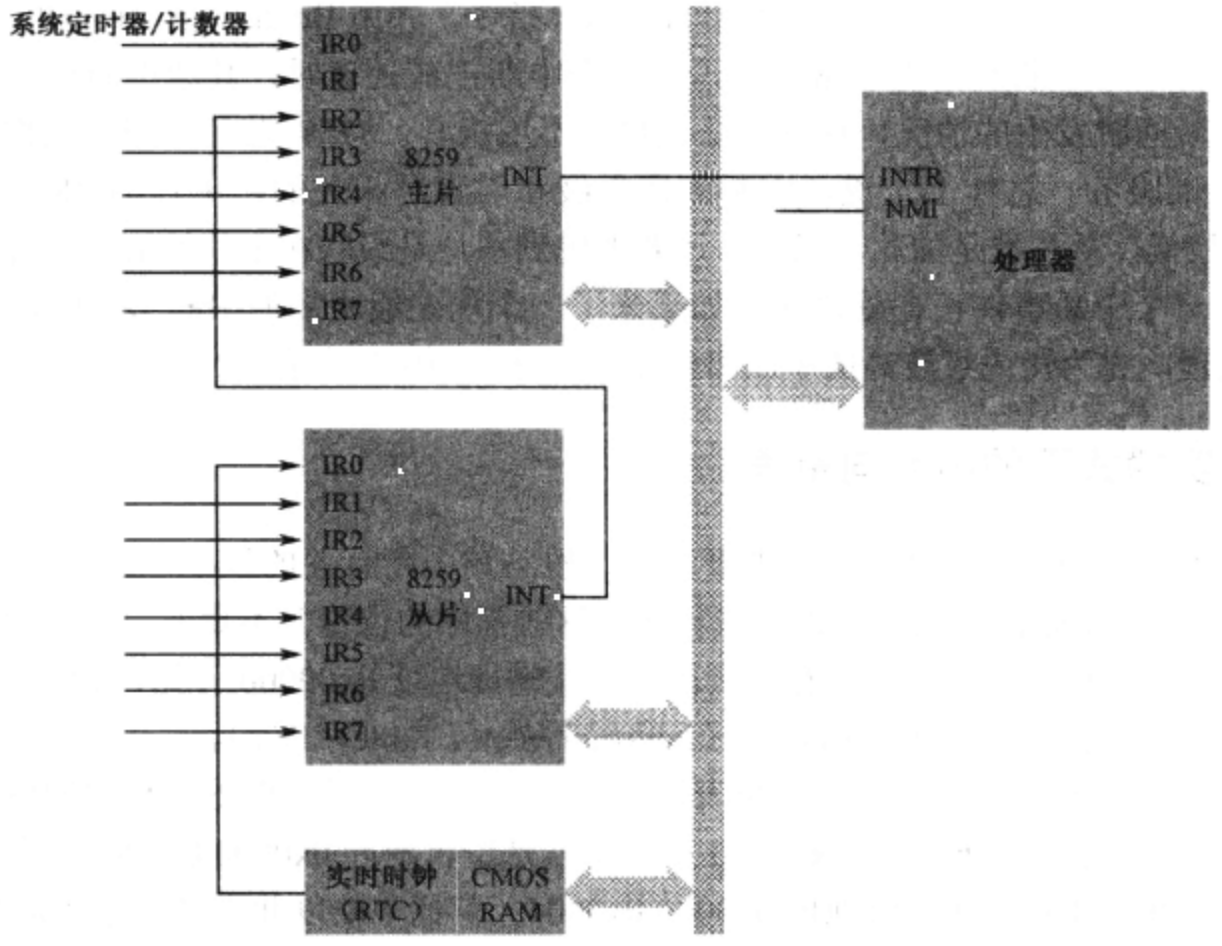

图 9-2 单处理器系统的中断机制

不知道是怎么想的，反正每片 8259 只有 8 个中断输入引脚，而在个人计算机上使用它，需要两块。如图 9-2 所示，第一块 8259 芯片的代理输出 INT 直接送到处理器的 INTR 引脚，这是主片（Master）；第二块 8259 芯片的 INT 输出送到第一块的引脚 2 上，是从片（Slave），两块芯片之间形成级联（Cascade）关系。

如此一来，两块 8259 芯片可以向处理器提供 15 个中断信号。当时，接在 8259 上的 15 个设备都是相当重要的，如 PS/2 键盘和鼠标、串行口、并行口、软磁盘驱动器、IDE 硬盘等。现在，这些设备很多都已淘汰或者正在淘汰中，根据需要，这些中断引脚可以被其他设备使用。

如图 9-2 所示，8259 的主片引脚 0（IR0）接的是系统定时器/计数器芯片；从片的引脚 0（IR0）接的是实时时钟芯片 RTC，该芯片是本章的主角，很快就会讲到。总之，这两块芯片的固定连接即使是在硬件更新换代非常频繁的今天，也依然没有改变。

在 8259 芯片内部，有中断屏蔽寄存器（Interrupt Mask Register，IMR），这是个 8 位寄存器，对应着该芯片的 8 个中断输入引脚，对应的位是 0 还是 1，决定了从该引脚来的中断信号是否能够通过 8259 送往处理器（0 表示允许，1 表示阻断，这可能出乎你的意料）。当外部设备通过某个引脚送来一个中断请求信号时，如果它没有被 IMR 阻断，那么，它可以被送往处理器。注意，8259 芯片是可编程的，主片的端口号是 0x20 和 0x21，从片的端口号是 0xa0 和 0xa1，可以通过这些端口访问 8259 芯片，设置它的工作方式，包括 IMR 的内容。

中断能否被处理，除了要看 8259 芯片的脸色外，最终的决定权在处理器手中。回到前面第 6 章，参阅图 6-2，你会发现，在处理器内部，标志寄存器有一个标志位 IF，这就是中断标志（Interrupt Flag）。当 IF 为 0 时，所有从处理器 INTR 引脚来的中断信号都被忽略掉；当其为 1 时，处理器可以接受和响应中断。

IF 标志位可以通过两条指令 cli 和 sti 来改变。这两条指令都没有操作数，cli（CLear Interrupt flag）用于清除 IF 标志位，sti（SeT Interrupt flag）用于置位 IF 标志。

◆ **检测点 9.1**

写一个小的主引导程序，在程序中使用 sti 和 cli 指令，并用 Bochs 观察 IF 位的变化。

在计算机内部，中断发生得非常频繁，当一个中断正在处理时，其他中断也会陆续到来，甚至会有多个中断同时发生的情况，这都无法预料。不用担心，8259 芯片会记住它们，并按一定的策略决定先为谁服务。总体上来说，中断的优先级和引脚是相关的，主片的 IR0 引脚优先级最高，IR7 引脚最低，从片也是如此。当然，还要考虑到从片是级联在主片的 IR2 引脚上。

最后，当一个中断事件正在处理时，如果来了一个优先级更高的中断事件时，允许暂时中止当前的中断处理，先为优先级较高的中断事件服务，这称为中断嵌套。

9.1.3 实模式下的中断向量表

所谓中断处理，归根结底就是处理器要执行一段与该中断有关的程序（指令）。处理器可以识别 256 个中断，那么理论上就需要 256 段程序。这些程序的位置并不重要，重要的是，在实模式下，处理器要求将它们的入口点集中存放到内存中从物理地址 0x00000 开始，到 0x003ff 结束，共 1KB 的空间内，这就是所谓的中断向量表（Interrupt Vector Table，IVT）。

如图 9-3 所示，每个中断在中断向量表中占 2 个字，分别是中断处理程序的偏移地址和段地址。中断 0 的入口点位于物理地址 0x00000 处，也就是逻辑地址 0x0000:0x0000；中断 1 的入口点位于物理地址 0x00004 处，即逻辑地址 0x0000:0x0004；其他中断以此类推，总之是按顺序的。

当中断发生时，如果从外部硬件到处理器之间的道路都是畅通的，那么，处理器在执行完当前的指令后，会立即着手为硬件服务。它首先会响应中断，告诉 8259 芯片准备着手处理该中断。接着，它还会要求 8259 芯片把中断号送过来。

在 8259 芯片那里，每个引脚都赋予了一个中断号。而且，这些中断号是可以改变的，可以对 8259 编程来灵活设置，但不能单独进行，只能以芯片为单位进行。比如，可以指定主片的中断号从 0x08 开始，那么它每个引脚 IR0～IR7 所对应的中断号分别是 0x08～0x0e。

中断信号来自哪个引脚，8259 芯片是最清楚的，所以它会把对应的中断号告诉处理器，处理器拿着这个中断号，要顺序做以下几件事。

① 保护断点的现场。首先要将标志寄存器 FLAGS 压栈，然后清除它的 IF 位和 TF 位。TF 是陷阱标志，这个以后再讲。接着，再将当前的代码段寄存器 CS 和指令指针寄存器 IP 压栈。

② 执行中断处理程序。由于处理器已经拿到了中断号，它将该号码乘以 4（毕竟每个中断在中断向量表中占 4 字节），就得到了该中断入口点在中断向量表中的偏移地址。接着，从表中依次取出中断程序的偏移地址和段地址，并分别传送到 IP 和 CS，自然地，处理器就开始执行中断处理程序了。

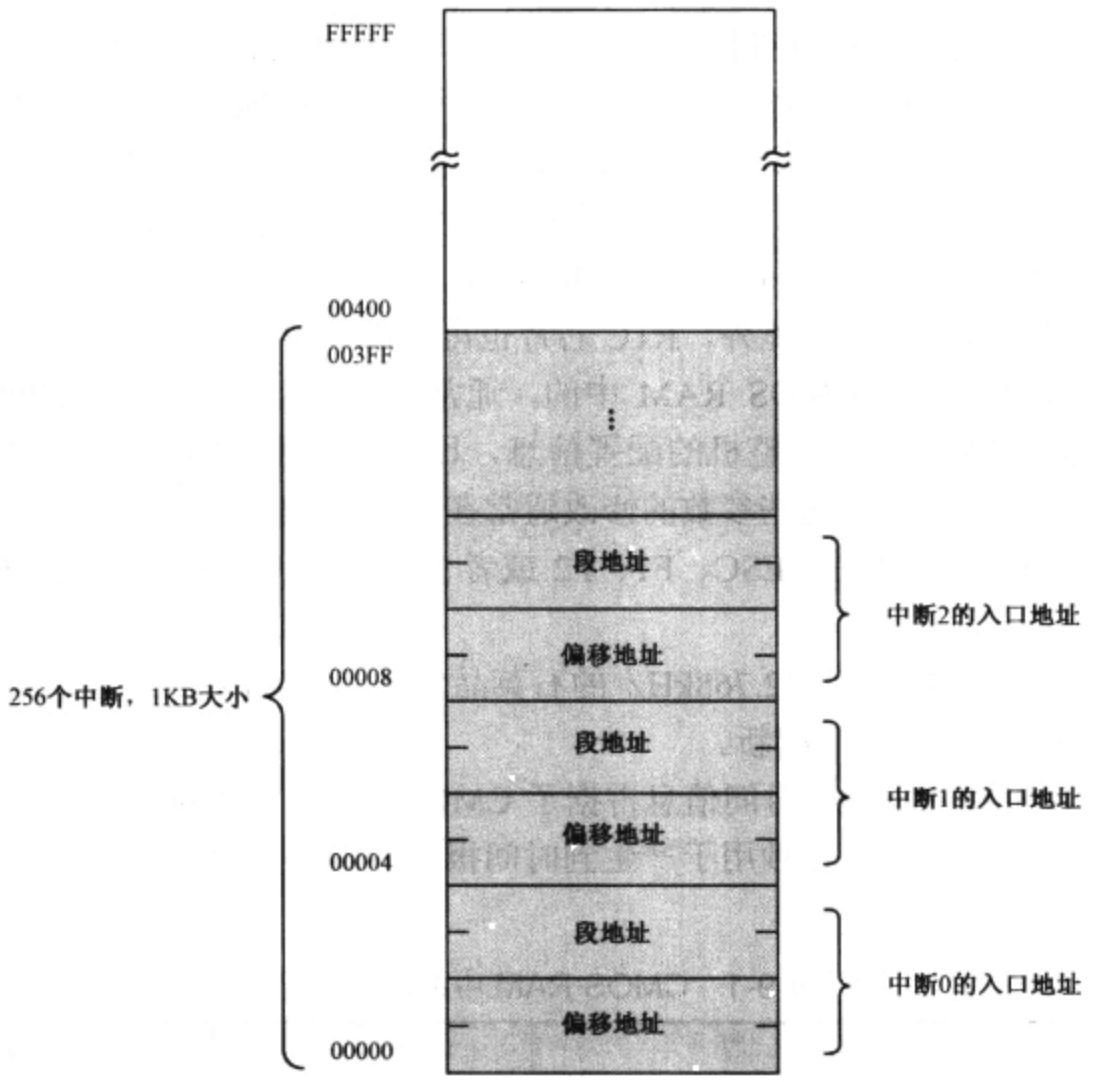

图 9-3　实模式下的中断向量表

注意，由于 IF 标志被清除，在中断处理过程中，处理器将不再响应硬件中断。如果希望更高优先级的中断嵌套，可以在编写中断处理程序时，适时用 sti 指令开放中断。

③ 返回到断点接着执行。所有中断处理程序的最后一条指令必须是中断返回指令 iret。这将导致处理器依次从栈中弹出（恢复）IP、CS 和 FLAGS 的原始内容，于是转到主程序接着执行。

iret 同样没有操作数，执行这条指令时，处理器依次从栈中弹出数值到 IP、CS 和标志寄存器。

顺便提醒一句，由于中断处理过程返回时，已经恢复了 FLAGS 的原始内容，所以 IF 标志位也自动恢复。也就是说，可以接受新的中断。

和可屏蔽中断不同，NMI 发生时，处理器不会从外部获得中断号，它自动生成中断号码 2，其他处理过程和可屏蔽中断相同。

中断随时可能发生，中断向量表的建立和初始化工作是由 BIOS 在计算机启动时负责完成的。BIOS 为每个中断号填写入口地址，因为它不知道多数中断处理程序的位置，所以，一律将它们指向一个相同的入口地址，在那里，只有一条指令：iret。也就是说，当这些中断发生时，只做一件事，那就是立即返回。当计算机启动后，操作系统和用户程序再根据自己的需要，来修改某些中断的入口地址，使它指向自己的代码。马上你就会看到，我们在本章也是这样做的。

9.1.4 实时时钟、CMOS RAM 和 BCD 编码

也许你曾经觉得奇怪，为什么计算机能够准确地显示日期和时间？原因很简单，如图 9-2 所示，在外围设备控制器芯片 ICH 内部，集成了实时时钟电路（Real Time Clock，RTC）和两小块由互补金属氧化物（CMOS）材料组成的静态存储器（CMOS RAM）。实时时钟电路负责计时，而日期和时间的数值则存储在这块存储器中。

实时时钟是全天候跳动的，即使是在你关闭了计算机的电源之后，原因在于它由主板上的一个小电池提供能量。它为整台计算机提供一个基准时间，为所有需要时间的软件和硬件服务。不像 8259 芯片，有关 RTC CMOS 的资料相当少见，很不容易完整地找到，而 8259 的内容则铺天盖地，到处都是。所以，本章只是简要地介绍 8259，而尽量多说一些和 RTC 有关的知识。

早期的计算机没有 ICH 芯片，各个接口单元都是分立的，单独地焊在主板上，并彼此连接。早期的 RTC 芯片是摩托罗拉（Motorola）MS146818B，现在直接集成在 ICH 内，并且在信号上与其兼容。除了日期和时间的保存功能外，RTC 芯片也可以提供闹钟和周期性的中断功能。

日期和时间信息是保存在 CMOS RAM 中的，通常有 128 字节，而日期和时间信息只占了一小部分容量，其他空间则用于保存整机的配置信息，比如各种硬件的类型和工作参数、开机密码和辅助存储设备的启动顺序等。这些参数的修改通常在 BIOS SETUP 开机程序中进行。要进入该程序，一般需要在开机时按 DEL、ESC、F1、F2 或者 F10 键。具体按哪个键，视计算机的厂家和品牌而定。

RTC 芯片由一个振荡频率为 32.768kHz 的石英晶体振荡器（晶振）驱动，经分频后，用于对 CMOS RAM 进行每秒一次的时间刷新。

如表 9-1 所示，常规的日期和时间信息占据了 CMOS RAM 开始部分的 10 字节，有年、月、日和时、分、秒，报警的时、分、秒用于产生到时间报警中断，如果它们的内容为 0xC0～0xFF，则表示不使用报警功能。

表 9-1 CMOS RAM 中的时间信息

偏移地址	内　容	偏移地址	内　容
0x00	秒	0x07	日
0x01	闹钟秒	0x08	月
0x02	分	0x09	年
0x03	闹钟分	0x0A	寄存器 A
0x04	时	0x0B	寄存器 B
0x05	闹钟时	0x0C	寄存器 C
0x06	星期	0x0D	寄存器 D

CMOS RAM 的访问，需要通过两个端口来进行。0x70 或者 0x74 是索引端口，用来指定 CMOS RAM 内的单元；0x71 或者 0x75 是数据端口，用来读写相应单元里的内容。举个例子，以下代码用于读取今天是星期几：

```
mov al,0x06
out 0x70,al
in al,0x71
```

不得不说的是，从很早的时候开始，端口 0x70 的最高位（bit 7）是控制 NMI 中断的开关。当它为 0 时，允许 NMI 中断到达处理器，为 1 时，则阻断所有的 NMI 信号，其他 7 个比特，即 0～6 位，则实际上用于指定 CMOS RAM 单元的索引号，这种规定直到现在也没有改变。

如图 9-4 所示，尽管端口 0x70 的位 7 不是中断信号，但它能控制与非门的输出，决定真正的

NMI 中断信号是否能到达处理器。

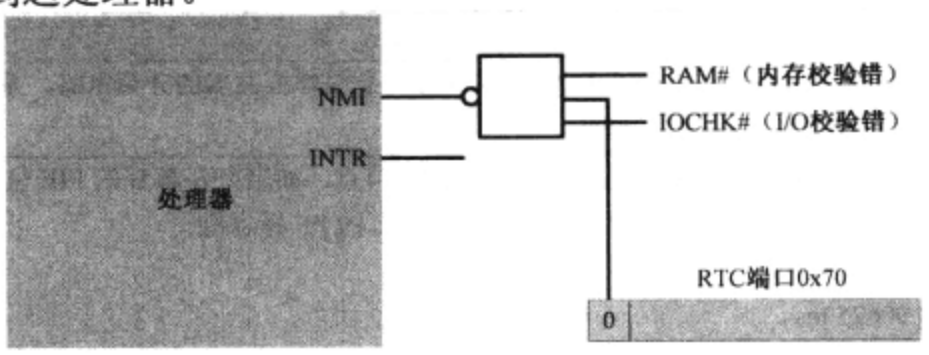

图 9-4 端口 0x70 的位 7 用于禁止或允许 NMI（仅为示意图）

通常来说，在往端口 0x70 写入索引时，应当先读取 0x70 原先的内容，然后将它用于随后的写索引操作中。但是，该端口是只写的，不能用于读出。在早期的系统中，计算机的制造成本很高，为了最大化地利用硬件资源，导致出现很多稀奇古怪的做法，这就是一个活生生的例子。

为了解决这个问题，同时也为了兼容以前的老式硬件，ICH 芯片允许通过切换访问模式来临时取得那些只写寄存器的内容，但这涉及更高层次的知识，已经超出了当前的话题范畴。现在，我们只想把问题搞得简单些，这么说吧，NMI 中断应当始终是允许的，在访问 RTC 时，我们直接关闭 NMI，访问结束后，再打开 NMI，而不管它以前到底是什么样子。

在早期，CMOS RAM 只有 64 字节，而最新的 ICH 芯片内则可能集成了 256 字节，新增的 128 字节称为扩展的 CMOS RAM。当然，在此之前，要先确保 ICH 内确实存在扩展的 CMOS RAM。

CMOS RAM 中保存的日期和时间，通常是以二进制编码的十进制数（Binary Coded Decimal，BCD），这是默认状态，如果需要，也可以设置成按正常的二进制数来表示。要想说明什么是 BCD 编码，最好的办法是举个例子。比如十进制数 25，其二进制形式是 00011001。但是，如果采用 BCD 编码的话，则一个字节的高 4 位和低 4 位分别独立地表示一个 0 到 9 之间的数字。因此，十进制数 25 对应的 BCD 编码是 00100101。由此可以看出，因为十进制数里只有 0～9，故用 BCD 编码的数，高 4 位和低 4 位都不允许大于 1001，否则就是无效的。

单元 0x0A～0x0D 不是普通的存储单元，而被定义成 4 个寄存器的索引号，也是通过 0x70 和 0x71 这两个端口访问的。这 4 个寄存器用于设置实时时钟电路的参数和工作状态。

寄存器 A 和 B 用于对 RTC 的功能进行整体性的设置，它们都是 8 位的寄存器，可读可写，其各位的用途如表 9-2 和表 9-3 所示。

表 9-2 寄存器 A 各位功能说明

比 特 位	功 能
7	正处于更新过程中（Update In Progress，UIP）。该位可以作为一个状态进行监视。CMOS RAM 中的时间和日期信息会由 RTC 周期性地更新，在此期间，用户程序不应当访问它们。对当前寄存器的写入不会改变此位的状态 0：更新周期至少在 488 微秒内不会启动。换句话说，此时访问 CMOS RAM 中的时间、日历和闹钟信息是安全的 1：正处于更新周期，或者马上就要启动 如果寄存器 B 的 SET 位不是 1，而且在分频电路已正确配置的情况下，更新周期每秒发生一次。在此期间，会增加保存的日期和时间、检查数据是否因超出范围而溢出（比如，31 号之后是下月 1 号，而不是 32 号），还要检查是否到了闹钟时间，最后，更新之后的数据还要写回原来的位置 更新周期至少会在 UIP 置 1 后的 488μs 内开始，而且整个周期的完成时间不会多于 1984 μs，在此期间，和日期时间有关的存储单元（0x00～0x09）会暂时脱离外部总线。为避免更新和数据遭到破坏，可以有两次安全地从外部访问这些单元的机会：当检测到更新结束中断发生时，可以有差不多 999ms 的时间用于读写有效的日期和时间数据；如果检测到寄存器 A 的 UIP 位为低（0），那么这意味着在更新周期开始前，至少还有 488μs 的时间

续表

比 特 位	功 能
6～4	分频电路选择（Division Chain Select）。这 3 位控制晶体振荡器的分频电路。系统将其初始化到 010，为 RTC 选择一个 32.768kHz 的时钟频率
3～0	速率选择（Rate Select，RS）。选择分频电路的分节点。如果寄存器 B 的 PIE 位被设置的话，此处的选择将产生一个周期性的中断信号，否则将设置寄存器 C 的 PF 标志位 0000：从不触发中断 0001：3.90625 ms 0010：7.8125 ms 0011：122.070 μs 0100：244.141 μs 0101：488.281 μs 0110：976.5625 μs 0111：1.953125 ms 1000：3.90625 ms 1001：7.8125 ms 1010：5.625 ms 1011：1.25 ms 1100 = 62.5 ms 1101：125 ms 1110：250 ms 1111：500 ms

表 9-3　寄存器 B 各位功能说明

比 特 位	功 能
7	更新周期禁止（Update Cycle Inhibit，SET）。允许或者禁止更新周期 0：更新周期每秒都会正常发生 1：中止当前的更新周期，并且此后不再产生更新周期。此位置 1 时，BIOS 可以安全地初始化日历和时间
6	周期性中断允许（Periodic Interrupt Enable，PIE） 0：不允许 1：当达到寄存器 A 中 RS 所设定的时间基准时，允许产生中断
5	闹钟中断允许（Alarm Interrupt Enable，AIE） 0：不允许 1：允许更新周期在到达闹点并将 AF 置位的同时，发出一个中断
4	更新结束中断允许（Update-Ended Interrupt Enable，UIE） 0：不允许 1：允许在每个更新周期结束时产生中断
3	方波允许（Square Wave Enable，SQWE） 该位空着不用，只是为了和早期的 Motorola 146818B 实时时钟芯片保持一致
2	数据模式（Data Mode，DM） 该位用于指定二进制或者 BCD 的数据表示形式 0：BCD 1：Binary
1	小时格式（Hour Format，HOURFORM） 0：12 小时制。在这种模式下，第 7 位为 0 表示上午（AM），为 1 表示下午（PM） 1：24 小时制
0	老软件的夏令时支持（Daylight Savings Legacy Software Support，DSLSWS） 该功能已不再支持，该位仅用于维持对老软件的支持，并且是无用的

寄存器 C 和 D 是标志寄存器，这些标志反映了 RTC 的工作状态，寄存器 C 是只读的，寄存器 D 则可读可写，它们也都是 8 位寄存器，其各位的含义如表 9-4 和表 9-5 所示。特别是寄存器

C，因为 RTC 可以产生中断，当中断产生时，可以通过该寄存器来识别中断的原因，比如，是周期性的中断，还是闹钟中断。

表 9-4 寄存器 C 各位功能说明

比 特 位	功 能
7	中断请求标志（Interrupt Request Flag，IRQF） IRQF＝（PF×PIE）＋（AF×AIE）＋（UF×UFE） 以上，加号表示逻辑或，乘号表示逻辑与。该位被设置时，表示肯定要发生中断。对寄存器 C 的读操作将导致此位清零
6	周期性中断标志（Periodic Interrupt Flag，PF）。 若寄存器 A 的 RS 位为 0000，则此位是 0，否则是 1。对寄存器 C 的读操作将导致此位清零 注：程序可以根据此位来判断 RTC 的中断原因
5	闹钟标志（Alarm Flag，AF）。 当所有闹点同当前时间相符时，此位是 1。对寄存器 C 的读操作将导致此位清零 注：程序可以根据此位来判断 RTC 的中断原因
4	更新结束标志（Update-Ended Flag，UF） 紧接着每秒一次的更新周期之后，RTC 电路立即将此位置 1。对寄存器 C 的读操作将导致此位清零 注：程序可以根据此位来判断 RTC 的中断原因
3～0	保留，总是报告 0

表 9-5 寄存器 D 各位功能说明

比 特 位	功 能
7	有效 RAM 和时间位（Valid RAM and Time Bit，VRT） 在写周期，此位应当始终写 0。不过，在读周期，此位回到 1。在 RTC 加电正常时，此位被硬件强制为 1
6	保留。总是返回 0。并且在写周期总是置 0
5～0	日期闹钟（Date Alarm），这些位保存着闹钟的月份数值

讲了这么多和 8259 以及 RTC 有关的内容，现在，我们想让 RTC 芯片定期发出一个中断，当这个中断发生的时候，还能执行我们自己编写的代码，来访问 CMOS RAM，在屏幕上显示一个动态走动的时钟。

9.1.5 代码清单 9-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。
本章代码清单：9-1（被加载的用户程序），源程序文件：c09_1.asm

9.1.6 初始化 8259、RTC 和中断向量表

本章提供的代码清单中，没有加载器程序。这是因为可以利用上一章提供的加载器来加载用户程序，只要符合规则，加载器是通用的。

用户程序的入口点在代码清单 9-1 的第 119 行，从这一行开始，到第 124 行，用于初始化各个段寄存器的内容。下面开始在中断向量表中安装实时时钟中断的入口点。既然本章的主题是中断，那么就很有必要强调一件事。当处理器执行任何一条改变栈段寄存器 SS 的指令时，它会在下一条指令执行完期间禁止中断。

栈无疑是很重要的，不能被破坏。要想改变代码段和数据段，只需要改变段寄存器就可以了。但栈段不同，因为它除了有段寄存器，还有栈指针。因此，绝大多数时候，对栈的改变是分

两步进行的：先改变段寄存器 SS 的内容，接着又修改栈指针寄存器 SP 的内容。

想象一下，如果刚刚修改了段寄存器 SS，在还没来得及修改 SP 的情况下，就发生了中断，会出现什么后果，而且要知道，中断是需要依靠栈来工作的。

因此，处理器在设计的时候就规定，当遇到修改段寄存器 SS 的指令时，在这条指令和下一条指令执行完毕期间，禁止中断，以此来保护栈。换句话说，你应该在修改段寄存器 SS 的指令之后，紧跟着一条修改栈指针 SP 的指令。

就代码清单 9-1 来说，在第 121、122 行执行期间，处理器禁止中断。再比如以下指令：

```
push cs
pop ss
mov sp,0
```

在后面两行指令执行期间，处理器禁止中断。

RTC 芯片的中断信号，通向中断控制器 8259 从片的第 1 个中断引脚 IR0。在计算机启动期间，BIOS 会初始化中断控制器，将主片的中断号设为从 0x08 开始，将从片的中断号设为从 0x70 开始。所以，计算机启动后，RTC 芯片的中断号默认是 0x70。尽管我们可以通过对 8259 编程来改变它，但是没有必要。

◆ 检测点 9.2

在 Bochs 中使用“xp”命令显示实模式下的中断向量表，并找出 0x70 号中断处理过程的段地址和偏移地址。

在安装中断向量之前，应该先显示些什么。第 126～130 行，显示两行提示信息，表明正在安装中断向量。这两个字符串位于第 286 行的数据段中。对于过程 put_string 没有什么好说的，它的代码和上一章相同，工作过程更没有区别。

为了修改某中断在中断向量表中的登记项，需要先找到它。第 132～135 行，将中断号 0x70 乘以 4，就是它在中断向量表内的偏移。

第 137 行，修改中断向量表时，需要先用 cli 指令清中断。当表项信息只修改了一部分时，如果发生 0x70 号中断，则会产生不可预料的问题。

第 139～141 行，将段寄存器 ES 压栈暂时保存，并使它指向中断向量表（所在的段）。

接着，第 142～145 行，访问中断向量表内 0x70 号中断的表项，分别写入新中断处理过程的偏移地址和段地址。新的中断处理过程是从标号 new_int_0x70 处开始的，而且位于当前代码段内。所以，该中断处理过程的偏移地址就是标号 new_int_0x70 的汇编地址（注意，段 code 的定义中带有 vstart=0 子句），段地址就是当前段寄存器 CS 的内容。表项修改完毕，从栈中恢复段寄存器 ES 的原始内容。

接下来，我们要设置 RTC 的工作状态，使它能够产生中断信号给 8259 中断控制器。

RTC 到 8259 的中断线只有一根，而 RTC 可以产生多种中断。比如闹钟中断、更新结束中断和周期性中断（参见表 9-3 和表 9-4）。RTC 的计时（更新周期）是独立的，产生中断信号只是它的一个赠品。所以，如果希望它能产生中断信号，需要额外设置。

以上所说的三种中断，我们只要设置一种就可以了。其实，最简单的就是设置更新周期结束中断。每当 RTC 更新了 CMOS RAM 中的日期和时间后，将发出此中断。更新周期每秒进行一次，因此该中断也每秒发生一次。

为了设置该中断，代码清单 9-1 第 147 行，将 RTC 寄存器 B 的索引 0x0b 传送到寄存器 AL。在访问 RTC 期间，最好是阻断 NMI，因此，第 148、149 行，先用 or 指令将 AL 的最高位置 1，再写端口 0x70。

第 150、151 行，用于通过数据端口 0x71 写寄存器 B。写的内容是 0x12，其二进制形式为

00010010，对照表 9-3，其意义不难理解：允许更新周期照常发生，禁止周期性中断，禁止闹钟功能，允许更新周期结束中断，使用 24 小时制，日期和时间采用 BCD 编码。

每次当中断实际发生时，可以在程序（中断处理过程）中读寄存器 C 的内容来检查中断的原因。比如，每当更新周期结束中断发生时，RTC 就将它的第 4 位置 1。该寄存器还有一个特点，就是每次读取它后，所有内容自动清零。而且，如果不读取它的话（换句话说，相应的位没有清零），同样的中断将不再产生。

为此，第 153～155 行，读一下寄存器 C 的内容，使之开始产生中断信号。注意，在向索引端口 0x70 写入的同时，也打开了 NMI。毕竟，这是最后一次在主程序中访问 RTC。

当然，如果采用周期性中断而不是更新周期结束中断，则稍微麻烦一些，因为要设置分频电路的分节点。以下代码片断用于产生 2 次/秒的周期性中断：

```
mov al,0x0a
or al,0x80
out 0x70,al
in al,0x71
or al,0x0f                    ;设置 RTC 寄存器 A，使其每秒发生 2 次中断
out 0x71,al
```

除此之外，还要设置寄存器 B 的 PIE 位，以允许周期性中断。

RTC 芯片设置完毕后，再来打通它到 8259 的最后一道屏障。正常情况下，8259 是不会允许 RTC 中断的，所以，需要修改它内部的中断屏蔽寄存器 IMR。IMR 是一个 8 位寄存器，位 0 对应着中断输入引脚 IR0，位 7 对应着引脚 IR7，相应的位是 0 时，允许中断，为 1 时，关掉中断。

8259 芯片是我见过的芯片中，访问起来最麻烦，也是我最讨厌的一个。好在有关它的资料非常好找，这里就简单地进行讲解。代码清单 9-1 第 157～159 行，通过端口 0xa1 读取 8259 从片的 IMR 寄存器，用 and 指令清除第 0 位，其他各位保持原状，然后再写回去。于是，RTC 的中断可以被 8259 处理了。

第 161 行，sti 指令将标志寄存器的 IF 位置 1，开放设备中断。从这个时候开始，中断随时都会发生，也随时会被处理。

9.1.7　使处理器进入低功耗状态

RTC 更新周期结束中断的处理过程可以看成另一个程序，是独立的处理过程，是额外的执行流程，它随时都会发生，但和主程序互不相干。关于它的执行过程，马上就要讲到，现在继续来看主程序。

在为中断过程做了初始化工作之后，主程序还是要继续执行的。代码清单 9-1 第 163～167 行，用于显示中断处理程序已安装成功的消息。

接着，第 169～171 行，使段寄存器 DS 指向显示缓冲区，并在屏幕上的第 12 行 33 列显示一个字符“@”，该位置差不多是整个屏幕的中心。表达式 12*160 + 33*2 是在指令编译阶段计算的，是该字符在显存中的位置。每个字符在显存中占 2 字节的位置，每行 80 个字符。

在此之后，主程序就无事可做了。第 174 行，hlt 指令使处理器停止执行指令，并处于停机状态，这将降低处理器的功耗。处于停机状态的处理器可以被外部中断唤醒并恢复执行，而且会继续执行 hlt 后面的指令。

所以，第 174～176 行用于形成一个循环，先是停机，接着某个外部中断使处理器恢复执行。一旦处理器的执行点来到 hlt 指令之后，则立即使它继续处于停机状态。

第 175 行，使用 not 指令将字符@的显示属性反转。not 是按位取反指令，其格式为

```
not r/m8
not r/m16
```

not 指令执行时，会将操作数的每一位反转，原来的 0 变成 1，原来的 1 变成 0。比如：

```
mov al,0x1f
not al          ;执行后，AL 的内容为 0xe0
```

从显示效果上看，循环将显示属性反转将取得一个动画效果，可以很清楚地看到处理器每次从停机状态被唤醒的过程。not 指令不影响任何标志位。

相对于 jmp $指令，使用 hlt 指令会大大降低处理器的占用率。Windows 7 操作系统有一个叫做 CPU 仪表盘的小工具，当使用 jmp $指令时，你会看到处理器占用率是 100%；而在一个循环中使用 hlt 指令时，该占用率马上降到 10%左右，这还是在虚拟机环境下，毕竟宿主操作系统还要占用处理器时间。

9.1.8 实时时钟中断的处理过程

主程序就是这样了，停机、执行，接着停机。与此同时，中断也在不停地发生着，处理器还要抽出空来执行中断处理过程，下面就来看看 RTC 的更新周期结束中断处理，该中断处理过程从代码清单 9-1 的第 27 行开始。

第 28～32 行，先保护好现场，将后面用到的寄存器压栈保存。这一点特别重要，中断处理过程必须无痕地执行，你不知道中断会在什么时候发生，也不知道中断发生时，哪一个程序正在执行，所以，必须保证中断返回时，能还原中断前的状态。

第 34～40 行，用于读 RTC 寄存器 A，根据 UIP 位的状态来决定是等待更新周期结束，还是继续往下执行。UIP 位为 0 表示现在访问 CMOS RAM 中的日期和时间是安全的。注意第 36 行，用于把寄存器 AL 的最高位置 1，从而阻断 NMI。当然，这是不必要的，当 NMI 发生时，整个计算机都应当停止工作，也不在乎中断处理过程能否正常执行。

第 38 行从数据端口读取寄存器 A 的内容；第 39 行，test 指令用于测试寄存器 AL 的第 7 位是否为 1。

“test”的意思是“测试”。顾名思义，可以用这条指令来测试某个寄存器，或者内存单元里的内容是否带有某个特征。

test 指令在功能上和 and 指令是一样的，都是将两个操作数按位进行逻辑“与”，并根据结果设置相应的标志位。但是，test 指令执行后，运算结果被丢弃（不改变或破坏两个操作数的内容）。

test 指令需要两个操作数，其指令格式为

```
test r/m8,imm8
test r/m16,imm16
test r/m8,r8
test r/m16,r16
```

和 and 指令一样，test 指令执行后，OF＝CF＝0；对 ZF、SF 和 PF 的影响视测试结果而定；对 AF 的影响未定义。对于 test 指令的应用，这里有一个例子，比如，我们想测试 Al 寄存器的第 3 位是“0”还是“1”，可以这样编写代码：

```
test al,0x08
```

0x08 的二进制形式为 00001000，它的第 3 位是“1”，表明我们关注的是这一位。不管寄存器 AL 中的内容是什么，只要它的第 3 位是“0”，这条指令执行后，结果一定是 00000000，标志位 ZF＝1；

相反，如果寄存器 AL 的第 3 位是“1”，那么结果一定是 00001000，ZF＝0。于是，根据 ZF 标志位的情况，就可以判定寄存器 AL 中的第 3 位是“0”还是“1”。

第 40 行，如果 UIP 位是 0，那么测试的结果是 ZF＝1，继续往下执行第 42 行；否则，说明 UIP 位是 1，需要返回到第 34 行继续等待 RTC 更新周期结束。

正常情况下，访问 CMOS RAM 中的日期和时间，必须等待 RTC 更新周期结束，所以上面的判断过程是必需的，而这些代码也适用于正常的访问过程。但是，当前中断处理过程是针对更新周期结束中断的，而当此中断发生时，本身就说明对 CMOS RAM 的访问是安全的，毕竟留给我们的时间是 999 毫秒，这段时间非常充裕，这段时间能执行千万条指令。所以，在这种特定的情况下，上面的判断过程是不必要的。当然，加上倒也无所谓。

第 42～52 行，分别访问 CMOS RAM 的 0、2、4 号单元，从中读取当前的秒、分、时数据，按顺序压栈等待后续操作。

第 60～62 行，读一下 RTC 的寄存器 C，使得所有中断标志复位。这等于是告诉 RTC，中断已经得到处理，可以继续下一次中断。否则的话，RTC 看到中断未被处理，将不再产生中断信号。RTC 产生中断的原因有多种，可以在程序中通过读寄存器 C 来判断具体的原因。不过这里不需要，因为除了更新周期结束中断外，其他中断都被关闭了。

现在，终于可以在屏幕上显示时间信息了。

第 64、65 行，临时将段寄存器 ES 指向显示缓冲区。

第 67、68 行，首先从栈中弹出小时数，调用过程 bcd_to_ascii 来将用 BCD 码表示的“小时”转换成 ASCII。该过程是在第 105 行定义的，调用该过程时，寄存器 AL 中的高 4 位和低 4 位分别是“小时”的十位数字和个位数字。

第 108 行，将寄存器 AL 中的内容复制一份给 AH，以方便下一步操作。

第 109、110 行，将寄存器 AL 中的高 4 位清零，只留下“小时”的个位数字。接着，将它加上 0x30，就得到该数字对应的 ASCII 码。

十位上的数字在寄存器 AH 的高 4 位。第 112 行，用右移 4 位的方法，将它“拉”到低 4 位，高 4 位在移动的过程中自动清零。

接着，第 113、114 行，用同样的办法来得到十位数字的 ASCII 码。此时，寄存器 AH 中是十位数字的 ASCII 码，AL 中是个位数字的 ASCII 码，它们将作为结果返回给调用者。

最后，第 116 行用于返回调用者。

接着回到第 69 行，为了连续在屏幕上显示内容，最好是采用基址寻址来访问显存。这一行用于指定显示的内容位于显存的什么位置。实际上，这里指定的是第 12 行 36 列。同以前一样，每个字符在显存中占两个字节，每行 80 个字符，所以这里使用了表达式 12*160 + 36*2，该表达式的值是在编译阶段计算的。

第 71、72 行，分别将“小时”的两个数位写到显存中，段地址在 ES 中，偏移地址分别是由寄存器 BX 和 BX+2 提供的。这里没有写入显示属性，这是因为我们希望采用默认的显示属性（屏幕是黑的，默认的显示属性是 0x07，即黑底白字）。

第 74、75 行，用于在下一个屏幕位置显示冒号“:”，这是在显示时间时都会采用的分隔符。当然，通过寄存器 AL 中转是多余的，这两句可以直接写成

```
    mov byte [es:bx+4],':'
```

遗憾的是，等我发现这个问题时，本章已经快要写完了，重新排版实在太费工夫。其实，这不算是个问题，无伤大雅，难道不是吗？

为了验证 RTC 更新结束中断是每秒发生一次的，第 76 行，将冒号的显示属性（颜色）用 not 指令反转。就像手掌的两面一样，每次发生中断时，冒号的颜色将和上一次相反，但永远在两个属

性之间来回变化。到程序运行的时候你就会发现，变化的频率是每秒一次。

剩下的指令都很好理解，因为它们的工作是按相同的方法显示分钟数和秒数。第 78～90 行，依次从栈中弹出分钟和秒的数值，并转换成 ASCII 码，然后显示在屏幕上，中间用冒号间隔。

在 8259 芯片内部，有一个中断服务寄存器（Interrupt Service Register，ISR），这是一个 8 位寄存器，每一位都对应着一个中断输入引脚。当中断处理过程开始时，8259 芯片会将相应的位置 1，表明正在服务从该引脚来的中断。

一旦响应了中断，8259 中断控制器无法知道该中断什么时候才能处理结束。同时，如果不清除相应的位，下次从同一个引脚出现的中断将得不到处理。在这种情况下，需要程序在中断处理过程的结尾，显式地对 8259 芯片编程来清除该标志，方法是向 8259 芯片发送中断结束命令（End Of Interrupt，EOI）。

中断结束命令的代码是 0x20。代码清单 9-1 第 92～94 行就用来做这件事。需要注意的是，如果外部中断是 8259 主片处理的，那么，EOI 命令仅发送给主片即可，端口号是 0x20；如果外部中断是由从片处理的，就像本章的例子，那么，EOI 命令既要发往从片（端口号 0xa0），也要发往主片。

最后，第 96～102 行，从栈中恢复被中断程序的现场，并用中断返回指令 iret 回到中断之前的地方继续执行。iret 的意思是 Interrupt Return。

9.1.9 代码清单 9-1 的编译和运行

本章的代码不包括加载器，也就是负责加载用户程序的主引导扇区代码，因为第 8 章已经提供了一个加载器，它同样可以加载本章的用户程序。

在完全理解了代码清单 9-1 的基础上，可以自行编辑和编译它，生成二进制文件。然后，使用 FixVhdWr 工具将其写入虚拟硬盘。和第 8 章一样，写入时的起始逻辑扇区号是 100，毕竟加载器每次要从这个地方读取和加载用户程序。

一旦所有工作都准备停当，即可启动虚拟机来观察运行结果。通常情况下，运行结果会如图 9-5 所示。

图 9-5 代码清单 9-1 编译和运行后的显示效果

在你欣赏程序的运行结果时，你一定会发现时间每秒更新一次，这从冒号的显示属性每秒反转一次可以看出来。与此不同的是，字符“@”却以很快的速度在闪烁。这意味着，把处理器从停机状态唤醒的不单单是实时时钟的更新周期结束中断，还有其他硬件中断，只不过我们不知道是谁而已。

9.2 内 部 中 断

和硬件中断不同，内部中断发生在处理器内部，是由执行的指令引起的。比如，当处理器检测到 div 或者 idiv 指令的除数为零时，或者除法的结果溢出时，将产生中断 0（0 号中断），这就是除法错中断。

再比如，当处理器遇到非法指令时，将产生中断 6。非法指令是指指令的操作码没有定义，或者指令超过了规定的长度。操作码没有定义通常意味着那不是一条指令，而是普通的数。

内部中断不受标志寄存器 IF 位的影响，也不需要中断识别总线周期，它们的中断类型是固定的，可以立即转入相应的处理过程。

9.3 软 中 断

软中断是由 int 指令引起的中断处理。这类中断也不需要中断识别总线周期，中断号在指令中给出。int 指令的格式如下：

```
int3
int imm8
into
```

int3 是断点中断指令，机器指令码为 CC。这条指令在调试程序的时候很有用，当程序运行不正常时，多数时候希望在某个地方设置一个检查点，也称断点，来查看寄存器、内存单元或者标志寄存器的内容，这条指令就是为这个目的而设的。

指令都是连续存放的，因此，所谓的断点，就是某条指令的起始地址。int3 是单字节指令，这是有意设计的。当需要设置断点时，可以将断点处那条指令的第 1 字节改成 0xcc，原字节予以保存。当处理器执行到 int3 时，即发生 3 号中断，转去执行相应的中断处理程序。中断处理程序的执行也要用到各个寄存器，这会破坏它们的内容，但 push 指令不会。我们可以在该程序内先压栈所有相关寄存器和内存单元，然后分别取出予以显示，它们就是中断前的现场内容。最后，再恢复那条指令的第 1 字节，并修改位于栈中的返回地址，执行 iret 指令。

注意，int3 和 int 3 不是一回事。前者的机器码为 CC，后者则是 CD 03，这就是通常所说的 int n，其操作码为 0xCD，第 2 字节的操作数给出了中断号。举几个例子：

```
int 0x00          ;引发 0 号中断
int 0x15          ;引发 0x15 号中断
int 0x16          ;引发 0x16 号中断
```

into 是溢出中断指令，机器码为 0xCE，也是单字节指令。当处理器执行这条指令时，如果标志寄存器的 OF 位是 1，那么，将产生 4 号中断。否则，这条指令什么也不做。

9.3.1 BIOS 中断

可以为所有的中断类型自定义中断处理过程，包括内部中断、硬件中断和软中断。特别是考虑到处理器允许 256 种中断类型，而且大部分都没有被硬件和处理器内部中断占用。

编写自己的中断处理程序有相当大的优越之处。不像 jmp 和 call 指令，int 指令不需要知道目标程序的入口地址。远转移指令 jmp 和远调用指令 call 必须直接或者间接给出目标位置的段地址和偏移地址，如果所有这一切都是自己安排的，倒也不成问题，但如果想调用别人的代码，比如操作系统的功能，这就很麻烦了。举个例子来说，假如你想读硬盘上的一个文件，因为操作系统有这样的功能，所以就不必在自己的程序中再写一套代码，直接调用操作系统例程就可以了。

但是，操作系统通常不会给出或者公布硬盘读写例程的段地址和偏移地址，因为操作系统也是经常修改的，经常发布新的版本。这样一来，例程的入口地址也会跟着变化。而且，也不能保证每次启动计算机之后，操作系统总待在同一个内存位置。

因为有了软中断，这是个利好条件。每次操作系统加载完自己之后，以中断处理程序的形式提供硬盘读写功能，并把该例程的地址填写到中断向量表中。这样，无论在什么时候，用户程序需要该功能时，直接发出一个软中断即可，不需要知道具体的地址。

最有名的软中断是 BIOS 中断，之所以称为 BIOS 中断，是因为这些中断功能是在计算机加电之后，BIOS 程序执行期间建立起来的。换句话说，这些中断功能在加载和执行主引导扇区之前，就已经可以使用了。

BIOS 中断，又称 BIOS 功能调用，主要是为了方便地使用最基本的硬件访问功能。不同的硬件使用不同的中断号，比如，使用键盘服务时，中断号是 0x16，即

```
int 0x16
```

通常，为了区分针对同一硬件的不同功能，使用寄存器 AH 来指定具体的功能编号。举例来说，以下指令用于从键盘读取一个按键：

```
mov ah,0x00          ;从键盘读字符
int 0x16             ;键盘服务。返回时，字符代码在寄存器 AL 中
```

在这里，当寄存器 AH 的内容是 0x00 时，执行 int 0x16 后，中断服务例程会监视键盘动作。当它返回时，会在寄存器 AL 中存放按键的 ASCII 码。

BIOS 中断很多，它们是在 BIOS 执行期间安装的，当主引导程序开始执行时，就可以在程序中使用了。本准备给出一张 BIOS 功能调用列表，但是考虑到现在网络技术很发达，上网很方便，大家可以自行从互联网上寻找相关的 BIOS 功能调用资料，然后在自己的程序中做实验。

你可能觉得奇怪，BIOS 是怎么建立起这套功能调用中断的？它又是怎么知道如何访问硬件的？毕竟，即使是它，要访问硬件也得通过端口一级的途径。

答案是，BIOS 可能会为一些简单的外围设备提供初始化代码和功能调用代码，并填写中断向量表，但也有一些 BIOS 中断是由外部设备接口自己建立的。

首先，每个外部设备接口，包括各种板卡，如网卡、显卡、键盘接口电路、硬件控制器等，都有自己的只读存储器（Read Only Memory，ROM），类似于 BIOS 芯片，这些 ROM 中提供了它自己的功能调用例程，以及本设备的初始化代码。按照规范，前两个单元的内容是 0x55 和 0xAA，第三个单元是本 ROM 中以 512 字节为单位的代码长度；从第四个单元开始，就是实际的 ROM 代码。

其次，我们知道，从内存物理地址 A0000 开始，到 FFFFF 结束，有相当一部分空间是留给外围设备的。如果设备存在，那么，它自带的 ROM 会映射到分配给它的地址范围内。

在计算机启动期间，BIOS 程序会以 2KB 为单位搜索内存地址 C0000～E0000 之间的区域。当它发现某个区域的头两个字节是 0x55 和 0xAA 时，那意味着该区域有 ROM 代码存在，是有效

的。接着，它对该区域做累加和检查，看结果是否和第三个单元相符。如果相符，就从第四个单元进入。这时，处理器执行的是硬件自带的程序指令，这些指令初始化外部设备的相关寄存器和工作状态，最后，填写相关的中断向量表，使它们指向自带的中断处理过程。

9.3.2 代码清单 9-2

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：9-2（被加载的用户程序/BIOS 中断演示程序），源程序文件：c09_2.asm

9.3.3 从键盘读字符并显示

代码清单 9-2 在框架上和前面的用户程序是一致的，差别在于代码段的功能上。

代码清单 9-2 第 28～32 行用于初始化各个段寄存器，这和以前的做法是相同的。

第 34～42 行用于在屏幕上显示字符串，采用的是循环的方法。循环用的是 loop 指令，为此，第 34 行用于计算字符串的长度，并传送到寄存器 CX 中，以控制循环的次数。第 35 行用于取得字符串的首地址。

向屏幕上写字符使用的是 BIOS 中断，具体地说就是中断 0x10 的 0x0e 号功能，该功能用于在屏幕上的光标位置处写一个字符，并推进光标位置。第 38～40 行分别按规范的要求准备各个参数，执行软中断。

第 41、42 行将递增寄存器 BX 中的偏移地址，以指向下一个字符在数据段中的位置。然后，loop 指令将寄存器 CX 的内容减 1，并在其不为零的情况下返回到循环体开始处，继续显示下一个字符。

剩下的工作内容既复杂，又简单。复杂是指，从键盘读取你按下的那个键，并把它显示在屏幕上，很复杂，需要访问硬件，写一大堆指令。简单是指，因为有了 BIOS 功能调用，这只需几条语句就能完成。

第 45、46 行使用软中断 0x16 从键盘读字符，需要在寄存器 AH 中指定 0x00 号功能。该中断返回后，寄存器 AL 中为字符的 ASCII 码。

第 48～50 行又一次使用了 int 0x10 的 0x0e 号功能，把从键盘取得的字符显示在屏幕上。

第 52 行，执行一个无条件转移指令，重新从键盘读取新的字符并予以显示。

9.3.4 代码清单 9-2 的编译和运行

将代码清单 9-2 编辑并编译后，用 FixVhdWr 程序将生成的二进制文件写入虚拟硬盘，起始的逻辑扇区号同样为 100。

如图 9-6 所示，启动虚拟机后，会看到一段欢迎的话。现在，你可以按下任何按键，它们将原样显示在“->”之后。慢慢试验，细细体会，你会发现某些按键的特点。比如，回车键（Enter）仅仅是将光标移到行首，退格键（Backspace）仅仅是将光标退后，并不破坏该位置上的字符。

图 9-6　代码清单 9-2 编译并运行后的效果

本章习题

1．修改代码 9-1，对 8259 芯片编程，屏蔽除 RTC 外的其他所有中断，观察字符“@”的变化速度。

2．修改代码 9-1，使之用一种新的方法来产生中断信号。建议的方法是采用周期性中断。不过，这涉及选择分频电路的分节点，比如，你可以选择 250ms 或者 500ms，它们分别会在 1 秒种内产生 4 次或 2 次中断。