保护模式基础

作者：Robert Collins

翻译：coly li

我记得当我第一次学习保护模式的时候，我刚刚自学完了汇编语言，于是我就有了一个疯狂的念头——自学保护模式。我买了一本包括保护模式示例的 80286 汇编语言教材，然后就开始学习了。没过几个小时，我意识到我买的书里没有任何有用的示例，因为书里的例子是介绍如何 EPROM CHIPS 编程的。因此我将那个误导我买此书的海报痛打了一顿。

直到现在，很多年以后，我唯一发现的关于任务切换的示例还是那么的费解和缺少文档说明，虽然我已经无法指出它了。借助 IBM 技术参考手册和我的那本 80286 教材，我坐下来尝试着理解保护模式。在 3 天里花费了大约 40 个小时之后，我最后从 IBM 技术参考手册中复制出来一些源代码，能够进入保护模式了，然后我退回了 DOS。

自从那时起，我学习了很多关于保护模式的知识，以及 CPU 内部是如何处理它的。我发现 CPU 内有一系列应用程序不可访问的隐藏寄存器。我也学习了这些寄存器是如何被装载的，他们在内存管理中的角色，以及更重要的，他们的精确内容。虽然这些寄存器对于应用程序是不可访问的，理解他们在内存管理中的角色的知识也可以被应用到实际编程中。在编程中使用这些知识，可以使用更少的数据，更少的代码，更快的速度来达到我们想要的结果。

保护模式基础

从一个应用的观点来看，保护模式和实模式没有什么太大的区别。都是使用内存段，中断和设备驱动程序去处理硬件。但是有一些细微的区别，使得将 DOS 应用移植到保护模式下并不是一件琐碎的事情（就是说比较麻烦？）。在实模式中，内存段通过与段寄存器结合起来，使用一种内在机制自动处理。这些段寄存器中的内容构成了 CPU 当前地址总线上的部分物理地址（参看图 1a）。

这些物理地址通过段寄存器乘以 16 得到，然后再加上一个 16bit 的偏移量。使用 16bit 偏移量也就暗示了 CPU 使用的段最大尺寸为 64KB。一些程序员通过增加段寄存器中的内容来解决 64K 段的尺寸限制。他们的程序通过将指向 64K 段的指针递增 16 字节的方式来一个段紧接着一个段的方式访问内存。任何在保护模式下使用这种技术访问内存的程序都会产生一个异常错误（CPU 产生的异常中断），因为在保护模式下，段寄存器的使用方法是不同的。

在保护模式下，内存段被一系列的表定义着（这些表成为描述符表），段寄存器被用来保存指向这些表的指针。每一个表项有 8 个字节宽，因此在段寄存器中的数值被定义为 8 的整数倍（如 08h，10h，18h 等等）。段寄存器中的低 3 位被定义了，但是由于一些简单的原因，我们说任何加载了内容不是 8 的倍数的段寄存器的程序，都会引起一个保护错误。

有两种表格被用来定义内存段：全局描述符表（Global Descriptor Table：GDT），和局部描述符表（Local Descriptor Table：LDT）。

GDT 中保存了所有应用程序都可以访问到的段信息，LDT 中保存着为某一个特定的任务或者程序指定的段信息。如前所述，段寄存器在保护模式下不够成物理地址的任何一部分，而是被用作指向 GDT 或者 LDT 的表项的指针（见图 1b）。每一次段寄存器被加载时，基地址从表项中被取出，然后保存在一个内部的、程序员不可见的被称为“段描述符缓冲（segment descriptor cache）”的寄存器中。出现在 CPU 地址总线上的物理地址通过将描述符缓冲中的基址加上 32 位的偏移量而构成。

描述符缓冲寄存器

不论是在实模式，或者是在保护模式下，CPU 将每一个段的基地址保存在一些叫做描述符缓冲寄存器的隐藏寄存器中。

每次 CPU 加载一个段寄存器，段基地址、段大小限制和访问属性（访问权限）信息也被加载，（或者被缓冲）到这些隐藏的寄存器中。为了提高性能，CPU 让之后的内存引用都通过描述符缓冲寄存器来计算，以替代通过查找描述符表来计算物理地址。理解这些隐藏的寄存器的角色和作用，对于采用新的更先进的编程技术和采用未公开的 LOADALL 指令是非常重要的。

图 2a 展示了 80286 上描述符缓冲的结构，图 2b 展示了 80386 和 80486 上的描述符缓冲的结构。

Figure 2 (a) 80286 Descriptor Cache Register

[47..32]	31	[30..29]	28	[27..25]	24	[23..00]
16-bit Limit	P	DPL	S	Type	A	24-bit base address

Figure 2 (b) 80386/80486 Descriptor Cache Register

[31..24]	23	[22..21]	20	[19..17]	16	15	14	[13..00]
0	P	DPL	S	Type	A	0	D	0

[63..32]
32-bit Physical Address

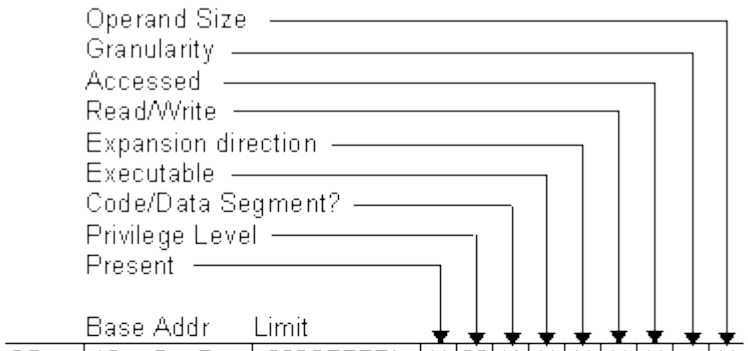
[95..64]
32-bit Limit

在上电时，描述符缓冲寄存器使用固定的缺省值加载，CPU 处于实模式，所有的段都被标记为可读/写的数据段，包括代码段（CS）。依照 Intel 的说法，每一次 CPU 在实模式下 load 一个段寄存器时，基地址将是段值的 16 倍，并且访问权限和尺寸限制属性都是固定的“实模式兼容”值。

这不是真的。实际上，只有段描述符缓冲访问权限在段寄存器每次加载时使用固定值加载——当遇到一个 far jump 的时候，也是如此。在实模式下加载任何其他的段寄存器不会改变存储在描述符缓冲寄存器中的访问权限或者段尺寸限制属性。对于这些段而言，访问权限和段尺寸大小属性都取决与任何先前的设置（查看图 3）。因此，在 80386 的实模式下是有可能拥有一个 4GB，只读的数据段的。但是 Intel 将不会承认，或者支持这种操作模式。

每次 CPU 加载一个段寄存器时，保护模式和实模式是不同的。保护模式会加载全部的描述符缓冲寄存器，不继承原先的数值。CPU 从描述符标中直接加载描述符缓冲。CPU 通过测试描述符表中的访问权限来检查一个段的合法性，非法值将会产生一个异常。任何将代码段加载到一个可读/写的数据段，都水产生一个保护错误。同样，任何尝试将数据段寄存器加载到一个可执行段的尝试都会产生一个异常。（保护错误和异常一样吗？）如果描述符表项通过了所有的检测，CPU 会非常严格的执行这些保护规则，然后 CPU 加载描述符缓冲寄存器。

Figure 3 -- Descriptor Cache Contents (Real Mode)



	Base Addr	Limit	Present	Privilege Level	Code/Data Segment?	Executable	Expansion direction	Read/Write	Accessed	Granularity	Operand Size
CS	16 x SegReg	0000FFFFh	Y	00	Y	N	U	Y	Y	-	H
SS	16 x SegReg	Honored	H	H	Y	H	H	H	Y	-	-
DS	16 x SegReg	Honored	H	H	Y	H	H	H	Y	-	-
ES	16 x SegReg	Honored	H	H	Y	H	H	H	Y	-	-
FS	16 x SegReg	Honored	H	H	Y	H	H	H	Y	-	-
GS	16 x SegReg	Honored	H	H	Y	H	H	H	Y	-	-

Y = Yes
N = No
H = Honored — Whenever the segment gets loaded, the contents of this field don't change.
- = Not Applicable

另一个将实模式应用程序移植到保护模式下的关键点是中断。在实模式下，指向中断处理例

程的双字长指针从物理地址的 0 开始排列（对于 386：除非 IDTR 被修改了，要不然也是这样）。

图 4a 举例说明了实模式下的中断服务例程寻址方式。当产生或调用一个中断时，CPU 在中断向量表中查看中断服务例程（ISR）的地址。当 CPU 将各种标志压到栈中之后，它就远程调用（far call）表中的地址。这些压到堆栈中的信息对于由软件、硬件、CPU 产生的中断而言都是一样的。

Figure 4(a) -- Interrupt service addressing in Real Mode

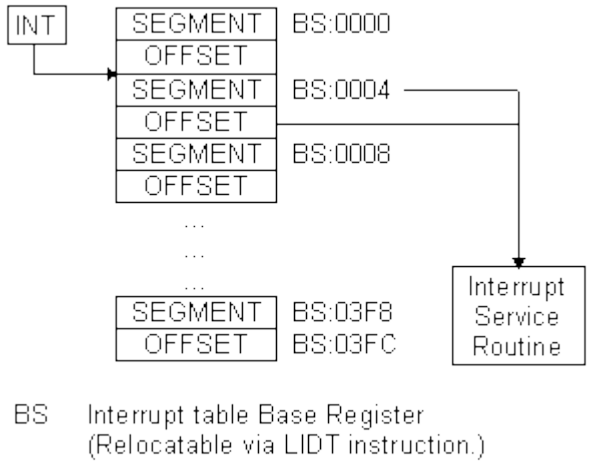
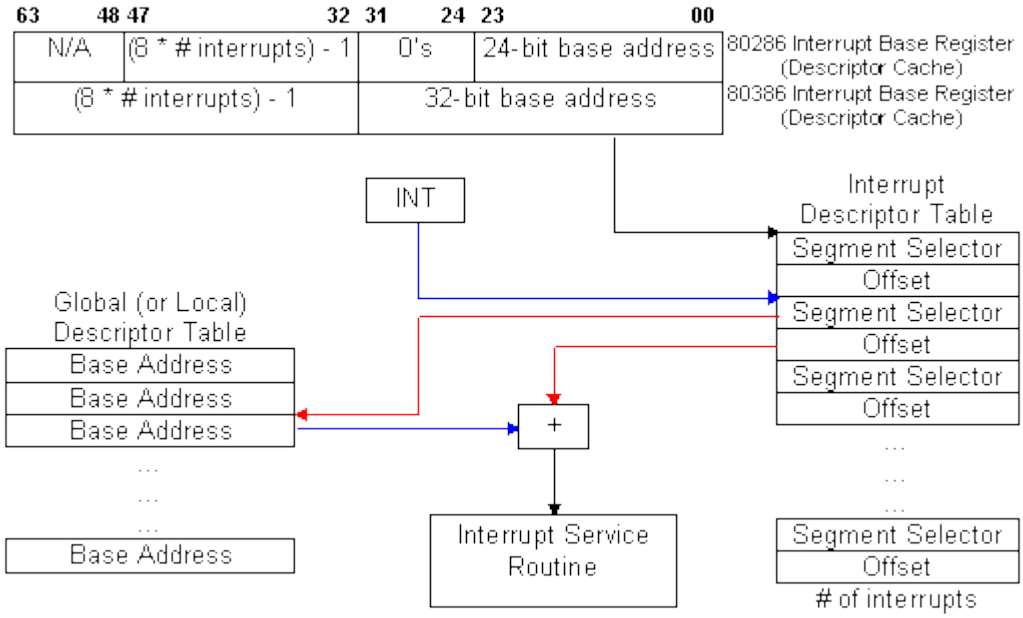


Fig 4(b) Interrupt service addressing in Protected Mode



对于保护模式，压入栈中的信息是可以变化的，就像中断向量的基地址和中断表的大小可以改变一样。保护模式下的中断向量查找机制同实模式下也有很大不同。

图 4b 图示了在保护模式下中断是如何被调用的。

当一个中断产生后，CPU 将中断号同存储在中断描述符寄存器中的中断描述符表的大小进行比较。如果中断号没有超过 IDT 的大小，则这个中断被视为可调用的，

然后就从描述符缓冲中取出 IDT 的基地址；然后就可以从 IDT 中获取中断服务程序的保护模式下地址。这个中断服务例程的地址不是一个物理地址，而是一个保护模式下的段地址。要使用 IDT 中指明的段选择信息，CPU 必须将同样的限制检查过程对 GDT 重新进行一次，以计算中断处理例程的物理地址。一旦物理地址计算出来了，CPU 就将 FLAGS,SEGMENT(选择器)，OFFSET 和可能的错误码压入栈中，然后再跳转到中断服务例程处。

对于软件和硬件的中断服务例程本身，在实模式和保护模式下没有太大的区别。但是针对CPU产生的中断和错误的中断处理例程必定是不同的。

Table 1 -- Exceptions and Interrupts

Description	Int #	Type	Return Addr points to faulting instruction	Error Code	This interrupt first appeared in this CPU
Division by 0	0	Fault	Yes	No	8086
Debug Exception	1	*1	*1	No	8086
NMI	2	*2	No	No	8086
Breakpoint	3	Trap	No	No	8086
Overflow	4	Trap	Yes	No	8086
Bounds	5	Fault	Yes	No	80186
Invalid OP Code	6	Fault	Yes	No	80186
Device not available	7	Fault	Yes	No	80186
Double Fault	8	Abort	No	Yes	80286
Copr. segment overrun	9	Fault	Yes	No	80286 *3
Invalid TSS	10	Fault	Yes	Yes	80286
Segment not present	11	Fault	Yes	Yes	80286
Stack fault	12	Fault	Yes	Yes	80286
General Protection	13	Fault	Yes	Yes	80286
Page fault	14	Fault	Yes	Yes	80386
Floating point error	16	Fault	Yes	No	80386
Alignment check	17	Fault	Yes	Yes	80486
Machine check	18	Abort	No	Yes	Pentium *4
Software interrupts	0-255	Trap	No	No	All

*1	在386级的CPU上，debug异常既可以当作trap，也可当作faults。一个trap是通过在flags image中设置TF(Trap Flag)，或者使用debug寄存器来产生一个数据断点来引起的，在这种情况下，返回地址是紧跟着trap的下一条指令。Faults是通过为代码执行断点设置debug寄存器产生的。对于所有的faults，返回地址指向fault的指令本身。
*2	Non-maskable.
*3	从80486中去掉了，在之后的CPU中不再产生13号异常。
*4	同具体型号相关，对于未来的处理器，处理方法可能不同，也可能去掉。

CPU 产生 3 种类型的中断：traps，faults，和 aborts。对于不同类型，栈内容也是变化的，例如错误码，可能被压入栈中，也可能不压入栈中，这取决于 CPU 产生的中断的类型。Traps 从来不会将错误码压栈；faults 通常会将错误码压栈（就是说有时候也会不压）；aborts 总会将错误码压栈。Traps 非常相似，并且也包括了软件的中断。这类中断的命名非常恰当，正如对当前的一个时间被“圈中（trapping）”了。

在 trap 之前，CPU 是不会知道这个事件发生了的。因此，在向中断发送信号之前一定要首先 trap 这个事件。所以 ISR 的返回地址指向紧跟着这个事件的指令。Traps 包括：被 0 除，数据断点，INT03。Faults 是因为某些错误并需要修改的时候发生了才产生的。CPU 会立即知道错误发生了，并且信号通知给中断产生机制。这一类 ISR 的主要意图，是修改问题然后从刚才发生问题的地方重新运行原指令。正因为如此，此类 ISR 的返回地址指向发生错误的指令——这样就可以使这个指令被重新执行。Aborts 是最严重的中断类型，被认为是不能够重新开始的。这时错误代码被压栈，但通常都是 0。

CPU 的栈段，和状态机，很可能会处于不确定的状态，因此重新执行一个 abort 可能会导致不可预料的结果。Table1 是对保护模式下 CPU 产生的中断的分组列表。在大多数情况下，CPU 也会在实模式下产生同样的中断，但是永远不会有错误代码压入栈中。

我曾经奇怪为什么 BIOS 不能工作在保护模式下。那时，我想编写模式无关的代码也许是比较容易的：只要不进行任何的 Far Jump，或者 Far Call 就可以。但是要做到这点却不是容易的事情。

为了避免使用 far jump 或者 far call，ISR 必须将压入栈中的任何错误代码都删除（为什么必须？）。这就是不可能的开始（要改）。由于错误码只有在保护模式下才会被放入栈中，因此在移去错误码之前，我们必须判断是在实模式下还是在保护模式下。要做到这一点，我们必须访问机器状态字 MSW，或者系统寄存器 CR0。

访问 MSW 可以在任何优先级中进行，但是访问 CR0 只能在最高优先级（level 0）中才能执行。如果用户程序运行于 level 之外的其他优先级，我们也许就没有办法访问这些寄存器。

在调用中断服务例程之前，我们可以通过特定的调用门切换自己的优先级。如果我们使用 SMSW 指令，这就不需要了。但是即使这个问题解决了，让我们想象一个程序在任何的段寄存器中保留有实模式的数值。如果 ISR 将这些寄存器的值压栈，并稍后在推栈，这个推栈指令将会导致 CPU 在 GDT 中查找一个 selector（段选择器？）。这时使用一个实模式值将会导致一个保护错误（protection error）。因此在保护模式中使用 BIOS 例程几乎是不可能的。但是如果有一系列所有程序和操作系统都需要遵守的规则（或标准）时，也许就可以在保护模式中运行 BIOS 了。

进入保护模式

我们的目标是进入保护模式，然后离开保护模式返回 DOS。286 没有退出保护模式的内部机制：一旦你进入了保护模式，你就只能一直呆在那里了。

IBM 认识到这一点就实现了一种解决方案可以通过 reset CPU 让 286 从保护模式返回。286 的 power-on 状态是处于实模式的，因此简单的 reset CPU 就可以让 CPU 返回实模式。

但这导致一个小问题，就是CPU不能继续运行之前的程序了。当reset后，CPU开始运行保存在内存顶部的指令，即BIOS指令代码。由于没有一个协议告诉BIOS我们为了退出保护模式而reset了CPU，因此BIOS没有办法将控制权返回给用户程序。

IBM实现了一种非常简单的协议，将一个代码写入CMOS RAM（CMOS），这样BIOS可以通过检查这个代码决定去做什么。当BIOS从reset向量开始执行后，它立刻在CMOS中检查这个代码以判别是否CPU是为了退出保护模式所以才被reset的。依靠这个保存在CMOS中的代码，BIOS可以将控制权返回给用户程序，使之继续执行。

Reset CPU不会没有副作用；所有CPU寄存器的内容都被破坏了，而有时候可编程中断控制器（PIC）中的中断掩码（interrupt mask）也被BIOS重新设置了（取决于系统shutdown的类型）。因此在进入保护模式之前保存PIC的掩码、栈指针和返回地址是应用程序自己要做的事情。

PIC掩码和栈指针必须被保存在用户数据段中，但是返回地址必须存储在一个预定义在BIOS数据段的固定位置——40:67h。（这我们就知道40：67h这个地址中保存着应用程序从保护模式返回实模式时的返回地址。）

然后，我们设置CMOS中的代码，告诉BIOS我们将从保护模式退出并且返回到用户程序。这个很容易实现——向2个CMOS I/O端口写入数值即可。当CPU被reset后，BIOS检查这个CMOS代码之后，就会清楚这个CMOS代码，这样之后的reset就不会导致预料外的结果了。

设置了 CMOS 中的代码之后，程序必须建立 GDT。（查阅相应的 Intel Programmer’s reference manual 中关于 GDT 的描述 ）。由于访问权限、尺寸限制等是静态值，因此可以通过编译器来填写。但是每一个段的基地址只有在运行中才能知道；因此程序必须将他们填写入 GDT。我们的程序将会建立一个包含这些代码、数据和栈段地址的 GDT（应该是这个程序自己本身的）。最后一个 GDT 项将指向 1M 以示示例（不明白）。

访问位于 1M 的内存可不像建立和使用一个 GDT 项那么简单。8086 具有在超过 1MB 的空间上寻址 64K（减去 16 字节）的潜能，但是它缺少第 21 根地址线（所以不行:-）。8086 仅有 20 根地址线（A00—A19），由于缺少 A20，任何尝试 1M 以上地址的尝试都会绕回到地址 0 的位置。286 具有 24bit 的寻址能力，因此在这方面与 8086 有所不同。

任何尝试对超过 1M 地址（FFFF:0010—FFFF:FFFF）的访问都会声明使用 A20，所以不会转回到地址 0 处。任何使用内存绕回特性的 8086 程序，将会在 286 上运行失败。

作为这个兼容性问题的解决方案，IBM 通过计算机上的某个芯片的可编程输出引脚增加了一个 CPU A20 输出。这个 CPU A20 信号实际上是一个 AND 门，这个 AND 门连接到地址总线上。

基于 CPU A20 的输入，AND 一个外部可编程 source，地址总线 A20 就被 assertred 了。由于可编程控制器下有一些有效的引脚可以被设置为高电平、低电平或者锁定，因此当引脚被设置为高电平，当 CPU 声明使用 A20 时，AND 门的输入就会变高；

当引脚输出被设为低电平时，A20 在地址总线上就总是低电平——即忽略了 CPU A20 的状态。

这样通过控制 A20 是否在地址总线上被声明使用，285 级别的机器就可以模拟 8086 处理器上的内存绕回（memory wrapping）特性了。

注意到只有地址线上的 A20 是被通过门控制的。因此，当没有使能 A20 门的输入时，CPU 只能寻址偶数 MB 的内存，例如：0-1M,2-3M,4-5M 等等。实际上，作为将地址总线 A20 置低电平的结果，这些内存块的内容同 1-2M,3-4M,5-6M 等的范围内的内容对应的都是相同的。

为了使能所有的 24 位的寻址能力，必须向键盘控制器发送一个命令。键盘控制器将会将他的某个输出引脚的输出置高电平，作为 A20 门的输入。一旦设置成功之后，内存将不会再被绕回（memory wrapping），这样我们就可以寻址整个 286 的 16M 内存，或者是寻址 80386 级别机器的所有 4G 内存了。

剩下的为了进入保护模式要做的事情就是改变 CPU 的状态到保护模式，然后执行一个 jump 指令以清楚 CPU 的预读取指令队列（在 Pentium 上就不必要了）。

下表总结了在 286 下进入（还能返回到实模式的）保护模式所需要的步骤。

- 保存 8259 PIC 掩码到程序数据段。
- 保存 SS:SP 到程序数据段。
- 保存从保护模式返回的返回地址到 40:67。

- 设置 CMOS 中的 shutdown 代码，以告诉 BIOS 当 CPU reset 以后我们还要返回原先实模式的用户程序。
- 建立 GDT，使能地址总线的 A20
- 通过 CPU 的机器状态字（MSW）使能保护模式，执行一个 JUMP，以清楚 CPU 的预读取指令队列。

以上 6 部的顺序不分前后。

由于 386 可以不通过复位 CPU 既可以退出保护模式返回实模式，因此在 386 或者 486 上进入保护模式所需要的步骤要比 286 下简单的多。为了兼容的目的，所有 386BIOS 将会识别定义在 286 级别机器上的 CPU shutdown 协议，但是遵循这个协议是没有必要的。

在 386 上退出保护模式，程序只需要简单的清除 CPU 控制寄存器上的一个位即可，不需要保存 PIC 掩码，SS:SP，返回地址和设置 CMOS 代码。因此在 386 上进入保护模式的步骤就简化为：

- 建立 GDT。
- 在地址总线上使能 A20。
- 设置 CPU 控制寄存器（CR0 或 MSW）以使能保护模式
- 执行一个跳转指令以清空 CPU 预取指令队列。

这些必须的步骤中，只有建立 GDT 是不同的。在 386 中基地址扩展为 32 位，大小限制扩展到 20 位，并引入了 2 个新的控制属性。列表 1 列举了进入保护模式需要的所有辅助子程序。

退出保护模式

同进入保护模式一样，退出保护模式在 286 和 386 的机器上也是不同的。386 仅仅是简单的清除 CPU 控制寄存器 CR0 上的一位，而 286 必须 reset CPU。

复位 CPU 也是要花费时间的，大约需要几百个时钟周期（不至于到上千个），才能够使 CPU 从保护模式退回到实模式运行用户程序。IBM 最初采用键盘控制器连接到 CPU RESET 线的一个输出引脚上。为了产生正确的命令，KBC 需要锁住 CPU 的 RESET 线。这种方法是可行的，但是非常慢。

很多新一代 286 芯片组具有一个 FAST RESET 特性。这些芯片组通过向一个 I/O 端口写入简单信息将 RESET 信号线锁住。所以如果允许的话，FAST RESET 是返回实模式更好的方法。

不过这里还有第三种方法，虽然很晦涩，但是确实是一种不使用 KBC 或者 FAST RESET 的一种复位 CPU 的有效方法(见 efficient method for resetting the CPU)。这种方法很精巧，比使用 KBC 快，并且可以在 386 上运行而不必复位 CPU！在退出保护模式返回实模式的各种方法中，这个方法应该是最精巧的——因为它可以工作在 286 和 386 两种 CPU 上，并且还很快。

列表 2 给出了使用 KBC 方式和刚才提高的这种高效方式的必须的代码。

使用KBC去复位CPU是很正规的做法，但是为了理解这个精巧的技术，需要一些说明。回忆一下我们在中断那部分的讨论，CPU通过中断描述符缓冲寄存器中的limit域（即最多有多少项）来检查中断号。如果这个测试通过了，那么下一步就是开始中断处理了。但是如果测试失败，那么CPU就会产生一个DOUBLE FAULT（INT08）信号。例如，让我们假定IDTR的limit域为80h。我们的IDT将会提供16个中断：00—15。如果中断16或者更高中断号的中断产生，CPU就会产生一个DOUBLE FAULT，这是由于在中断调用的初期产生了一个错误（fault）。现在，假定IDTR的limit域为0，这就禁止对所有的中断服务。任何中断的产生都会导致DOUBLE FAULT。但是由于limit域小于40h，因此DOUBLE FAULT自己将会产生一个错误（fault）。这最终会导致一个TRIPLE FAULT，然后CPU会进入shutdown循环。这个shutdown循环不会复位CPU，就像shutdown循环被认为是总选循环一样。外部硬件设备会跟随CPU一起去识别这个shutdown循环信号。一旦这个信号被识别了，外部硬件就会锁定CPU的RESET输入。因此，我们要引起RESET信号所需要做的唯一的事情就是设置IDTR的限制为0（IDTR.LIMIT=0），然后产生一个中断（什么中断都行）。

为了让这个方法看起来更优雅一些，我们不使用INT来中断CPU，我们产生一个无效的操作数。我们的操作数经过精心选择，肯定不会出现在286上，但是却在386上存在。挑选这个操作数方法是为了这个目的：MOV CRO,EAX。这将在286上产生一个期望的无效操作数异常，但是确实在386上退出保护模式的指令序列中的第一个。这样将会在286上产生RESET，而在386上可以没有影响的继续运行下去以体面的退出保护模式。

退出286和386的保护模式的步骤同进入保护模式的反向步骤非常相似，

在 286 上，你必须：

- 复位 CPU 进入实模式
- 用实模式兼容的数值加载段寄存器
- 恢复 SS:SP
- 限制地址线上的 A20（关闭 A20 门）
- 恢复 PIC 掩码

在 386 上，步骤会简单一些：

- 使用实模式兼容的数值加载段寄存器
- 复位 CR0 的 PE 位（Protection Enable bit）
- 使用实模式数值加载段寄存器
- 限制地址线上的 A20（关闭 A20 门）

（列表 3 把扩了退出保护模式后恢复机器状态所需要的子程序）

注意：在 386 下从保护模式退出到实模式时需要加载两次段寄存器。

第一次加载段寄存器是为了确保实模式兼容的值保存在隐藏的描述符缓冲寄存器（们）中，

由于从保护模式退回到实模式时描述符缓冲寄存器会继承访问属性，段大小限制（就是说返回以后，这些数值还在，不变）。

第二次加载段寄存器是为了使用实模式段数值定义段寄存器。

现在我们拥有进入和退出保护模式所需要的所有工具和理论，我们可以通过写一个进入保

护模式的程序来应用这些知识，从扩展内存中移动一个数据块，然后再退出保护模式——返回到DOS。

列表4展示了一个从内存1M处复制1KB数据到我们程序的数据段的程序的基本步骤。

小结

运行在实模式和运行在保护模式下的应用软件并没有太多的不同。在两种模式下我们都是用内存段、中断和设备驱动去支持硬件。无论在实模式或者保护模式中，一系列称为描述符缓冲寄存器的用户不可访问的寄存器们————在内存段和内存管理中扮演了非常重要的角色。

描述符缓冲寄存器保存着定义段基地址、段大小限制和段访问权限属性的信息，并被用于所用的内存引用场合——而忽略在段寄存器中的数值。

进入和退出保护模式需要采用适当的机制即可：进入保护模式需要保存退出保护模式时会用到的当前机器状态。返回实模式的机制依赖于CPU的类型：286需要复位CPU，386可以在程序的控制下进入实模式。

为了对我们的关于CPU内部操作的知识加以应用，我们可以编写一段对应不同CPU类型的退出保护模式源程序来试试。

下面是本文中提到的参考源代码

ftp://ftp.x86.org/source/pmbasics/tspec_a1.asm

ftp://ftp.x86.org/source/pmbasics/tspec_a1.l1

ftp://ftp.x86.org/source/pmbasics/tspec_a1.l2

ftp://ftp.x86.org/source/pmbasics/tspec_a1.l3

http://www.x86.org/ftp/source/pmbasics/tspec_a1.l4

下载所有的源代码：

http://www.x86.org/ftp/dloads/pmbasics.zip