程序 6-2 linux/boot/setup.S

```
!
!       setup.s         (C) 1991 Linus Torvalds
!
! setup.s is responsible for getting the system data from the BIOS,
! and putting them into the appropriate places in system memory.
! both setup.s and system has been loaded by the bootblock.
!
! This code asks the bios for memory/disk/other parameters, and
! puts them in a "safe" place: 0x90000-0x901FF, ie where the
! boot-block used to be. It is then up to the protected mode
! system to read them from there before the area is overwritten
! for buffer-blocks.
!
! setup.s 负责从 BIOS 中获取系统数据，并将这些数据放到系统内存的适当
! 地方。此时 setup.s 和 system 已经由 bootsect 引导块加载到内存中。
!
! 这段代码询问 bios 有关内存/磁盘/其他参数，并将这些参数放到一个
! “安全的”地方：0x90000-0x901FF，也即原来 bootsect 代码块曾经在
! 的地方，然后在被缓冲块覆盖掉之前由保护模式的 system 读取。

! NOTE! These had better be the same as in bootsect.s!
! 以下这些参数最好和 bootsect.s 中的相同！
#include <linux/config.h>
! config.h 中定义了 DEF_INITSEG = 0x9000; DEF_SYSSEG = 0x1000; DEF_SETUPSEG = 0x9020。

INITSEG  = DEF_INITSEG  ! we move boot here - out of the way ! 原来 bootsect 所处的段。
SYSSEG   = DEF_SYSSEG   ! system loaded at 0x10000 (65536).  ! system 在 0x10000 处。
SETUPSEG = DEF_SETUPSEG ! this is the current segment        ! 本程序所在的段地址。

.globl begtext, begdata, begbss, endtext, enddata, endbss
.text
begtext:
.data
begdata:
.bss
begbss:
.text

entry start
start:

! ok, the read went well so we get current cursor position and save it for
! posterity.
! ok，整个读磁盘过程都正常，现在将光标位置保存以备今后使用（相关代码在 59--62 行）。

! 下句将 ds 置成 INITSEG(0x9000)。这已经在 bootsect 程序中设置过，但是现在是 setup 程序，
! Linus 觉得需要再重新设置一下。
        mov     ax,#INITSEG
        mov     ds,ax

! Get memory size (extended mem, kB)
! 取扩展内存的大小值（KB）。
```

```
! 利用 BIOS 中断 0x15 功能号 ah = 0x88 取系统所含扩展内存大小并保存在内存 0x90002 处。
! 返回：ax = 从 0x100000（1M）处开始的扩展内存大小(KB)。若出错则 CF 置位，ax = 出错码。
41
42          mov     ah,#0x88
43          int     0x15
44          mov     [2],ax              ! 将扩展内存数值存在 0x90002 处（1 个字）。
45
46  ! check for EGA/VGA and some config parameters
    ! 检查显示方式（EGA/VGA）并取参数。
    ! 调用 BIOS 中断 0x10，附加功能选择方式信息。功能号：ah = 0x12，bl = 0x10
    ! 返回：bh =显示状态。0x00 -彩色模式，I/O 端口=0x3dX；0x01 -单色模式，I/O 端口=0x3bX。
    ! bl = 安装的显示内存。0x00 - 64k；0x01 - 128k；0x02 - 192k；0x03 = 256k。
    ! cx = 显示卡特性参数(参见程序后对 BIOS 视频中断 0x10 的说明)。
47
48          mov     ah,#0x12
49          mov     bl,#0x10
50          int     0x10
51          mov     [8],ax              ! 0x90008 = ??
52          mov     [10],bx             ! 0x9000A =安装的显示内存；0x9000B=显示状态(彩/单色)
53          mov     [12],cx             ! 0x9000C =显示卡特性参数。
    ! 检测屏幕当前行列值。若显示卡是 VGA 卡时则请求用户选择显示行列值，并保存到 0x9000E 处。
54          mov     ax,#0x5019          ! 在 ax 中预置屏幕默认行列值（ah = 80 列；al=25 行）。
55          cmp     bl,#0x10            ! 若中断返回 bl 值为 0x10，则表示不是 VGA 显示卡，跳转。
56          je      novga
57          call    chsvga              ! 检测显示卡厂家和类型，修改显示行列值（第 215 行）。
58  novga:  mov     [14],ax             ! 保存屏幕当前行列值（0x9000E，0x9000F）。

    ! 这段代码使用 BIOS 中断取屏幕当前光标位置（列、行），并保存在内存 0x90000 处（2 字节）。
    ! 控制台初始化程序会到此处读取该值。
    ! BIOS 中断 0x10 功能号 ah = 0x03，读光标位置。
    ! 输入：bh = 页号
    ! 返回：ch = 扫描开始线；cl = 扫描结束线；dh = 行号(0x00 顶端)；dl = 列号(0x00 最左边)。
59          mov     ah,#0x03            ! read cursor pos
60          xor     bh,bh
61          int     0x10                ! save it in known place, con_init fetches
62          mov     [0],dx              ! it from 0x90000.
63
64  ! Get video-card data:
    ! 下面这段用于取显示卡当前显示模式。
    ! 调用 BIOS 中断 0x10，功能号 ah = 0x0f
    ! 返回：ah = 字符列数；al = 显示模式；bh = 当前显示页。
    ! 0x90004(1 字)存放当前页；0x90006 存放显示模式；0x90007 存放字符列数。
65
66          mov     ah,#0x0f
67          int     0x10
68          mov     [4],bx              ! bh = display page
69          mov     [6],ax              ! al = video mode, ah = window width
70
71  ! Get hd0 data
    ! 取第一个硬盘的信息（复制硬盘参数表）。
    ! 第 1 个硬盘参数表的首地址竟然是中断向量 0x41 的向量值！而第 2 个硬盘参数表紧接在第 1 个
    ! 表的后面，中断向量 0x46 的向量值也指向第 2 个硬盘的参数表首址。表的长度是 16 个字节。
    ! 下面两段程序分别复制 ROM BIOS 中有关两个硬盘的参数表，0x90080 处存放第 1 个硬盘的表，
```

```
    ! 0x90090 处存放第 2 个硬盘的表。
72
    ! 第 75 行语句从内存指定位置处读取一个长指针值并放入 ds 和 si 寄存器中。ds 中放段地址，
    ! si 是段内偏移地址。这里是把内存地址 4 * 0x41（= 0x104）处保存的 4 个字节读出。这 4 字
    ! 节即是硬盘参数表所处位置的段和偏移值。
73          mov     ax,#0x0000
74          mov     ds,ax
75          lds     si,[4*0x41]         ! 取中断向量 0x41 的值，即 hd0 参数表的地址➔ds:si
76          mov     ax,#INITSEG
77          mov     es,ax
78          mov     di,#0x0080          ! 传输的目的地址: 0x9000:0x0080 ➔ es:di
79          mov     cx,#0x10            ! 共传输 16 字节。
80          rep
81          movsb
82
83  ! Get hd1 data
84
85          mov     ax,#0x0000
86          mov     ds,ax
87          lds     si,[4*0x46]         ! 取中断向量 0x46 的值，即 hd1 参数表的地址➔ds:si
88          mov     ax,#INITSEG
89          mov     es,ax
90          mov     di,#0x0090          ! 传输的目的地址: 0x9000:0x0090 ➔ es:di
91          mov     cx,#0x10
92          rep
93          movsb
94
95  ! Check that there IS a hd1 :-)
    ! 检查系统是否有第 2 个硬盘。如果没有则把第 2 个表清零。
    ! 利用 BIOS 中断调用 0x13 的取盘类型功能，功能号 ah = 0x15；
    ! 输入：dl = 驱动器号（0x8X 是硬盘：0x80 指第 1 个硬盘，0x81 第 2 个硬盘）
    ! 输出：ah = 类型码；00 - 没有这个盘，CF 置位；01 - 是软驱，没有 change-line 支持；
    !                   02 - 是软驱(或其他可移动设备)，有 change-line 支持； 03 - 是硬盘。
96
97          mov     ax,#0x01500
98          mov     dl,#0x81
99          int     0x13
100         jc      no_disk1
101         cmp     ah,#3               ! 是硬盘吗？(类型 = 3 ？)。
102         je      is_disk1
103 no_disk1:
104         mov     ax,#INITSEG         ! 第 2 个硬盘不存在，则对第 2 个硬盘表清零。
105         mov     es,ax
106         mov     di,#0x0090
107         mov     cx,#0x10
108         mov     ax,#0x00
109         rep
110         stosb
111 is_disk1:
112
113 ! now we want to move to protected mode ...
    ! 现在我们要进入保护模式中了...
114
```

```
115         cli                     ! no interrupts allowed !      ! 从此开始不允许中断。
116
117 ! first we move the system to it's rightful place
    ! 首先我们将 system 模块移到正确的位置。
    ! bootsect 引导程序会把 system 模块读入到内存 0x10000（64KB）开始的位置。由于当时假设
    ! system 模块最大长度不会超过 0x80000（512KB），即其末端不会超过内存地址 0x90000，所以
    ! bootsect 会把自己移动到 0x90000 开始的地方，并把 setup 加载到它的后面。下面这段程序的
    ! 用途是再把整个 system 模块移动到 0x00000 位置，即把从 0x10000 到 0x8ffff 的内存数据块
    ! （512KB）整块地向内存低端移动了 0x10000（64KB）的位置。
118
119         mov     ax,#0x0000
120         cld                     ! 'direction'=0, movs moves forward
121 do_move:
122         mov     es,ax           ! destination segment ! es:di 是目的地址(初始为 0x0:0x0)
123         add     ax,#0x1000
124         cmp     ax,#0x9000      ! 已经把最后一段（从 0x8000 段开始的 64KB）代码移动完?
125         jz      end_move        ! 是，则跳转。
126         mov     ds,ax           ! source segment  ! ds:si 是源地址(初始为 0x1000:0x0)
127         sub     di,di
128         sub     si,si
129         mov     cx,#0x8000      ! 移动 0x8000 字（64KB 字节）。
130         rep
131         movsw
132         jmp     do_move
133
134 ! then we load the segment descriptors
    ! 此后，我们加载段描述符。
    ! 从这里开始会遇到 32 位保护模式的操作，因此需要 Intel 32 位保护模式编程方面的知识了，
    ! 有关这方面的信息请查阅列表后的简单介绍或附录中的详细说明。这里仅作概要说明。在进入
    ! 保护模式中运行之前，我们需要首先设置好需要使用的段描述符表。这里需要设置全局描述符
    ! 表和中断描述符表。
    !
    ! 下面指令 lidt 用于加载中断描述符表（IDT）寄存器。它的操作数（idt_48）有 6 字节。前 2
    ! 字节（字节 0-1）是描述符表的字节长度值；后 4 字节（字节 2-5）是描述符表的 32 位线性基
    ! 地址，其形式参见下面 218--220 行和 222--224 行说明。中断描述符表中的每一个 8 字节表项
    ! 指出发生中断时需要调用的代码信息。与中断向量有些相似，但要包含更多的信息。
    !
    ! lgdt 指令用于加载全局描述符表（GDT）寄存器，其操作数格式与 lidt 指令的相同。全局描述
    ! 符表中的每个描述符项（8 字节）描述了保护模式下数据段和代码段（块）的信息。 其中包括
    ! 段的最大长度限制（16 位）、段的线性地址基址（32 位）、段的特权级、段是否在内存、读写
    ! 许可权以及其他一些保护模式运行的标志。参见后面 205--216 行。
135
136 end_move:
137         mov     ax,#SETUPSEG    ! right, forgot this at first. didn't work :-)
138         mov     ds,ax           ! ds 指向本程序(setup)段。
139         lidt    idt_48          ! load idt with 0,0                  ! 加载 IDT 寄存器。
140         lgdt    gdt_48          ! load gdt with whatever appropriate ! 加载 GDT 寄存器。
141
142 ! that was painless, now we enable A20
    ! 以上的操作很简单，现在我们开启 A20 地址线。
    ! 为了能够访问和使用 1MB 以上的物理内存，我们需要首先开启 A20 地址线。参见本程序列表后
    ! 有关 A20 信号线的说明。关于所涉及的一些端口和命令，可参考 kernel/chr_drv/keyboard.S
    ! 程序后对键盘接口的说明。至于机器是否真正开启了 A20 地址线，我们还需要在进入保护模式
```

```
! 之后（能访问 1MB 以上内存之后）在测试一下。这个工作放在了 head.S 程序中（32--36 行）。
143
144         call    empty_8042              ! 测试 8042 状态寄存器，等待输入缓冲器空。
                                            ! 只有当输入缓冲器为空时才可以对其执行写命令。
145         mov     al,#0xD1                ! command write ! 0xD1 命令码-表示要写数据到
146         out     #0x64,al                ! 8042 的 P2 端口。P2 端口位 1 用于 A20 线的选通。
147         call    empty_8042              ! 等待输入缓冲器空，看命令是否被接受。
148         mov     al,#0xDF                ! A20 on        ! 选通 A20 地址线的参数。
149         out     #0x60,al                ! 数据要写到 0x60 口。
150         call    empty_8042              ! 若此时输入缓冲器为空，则表示 A20 线已经选通。
151
152 ! well, that went ok, I hope. Now we have to reprogram the interrupts :-(
153 ! we put them right after the intel-reserved hardware interrupts, at
154 ! int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
155 ! messed this up with the original PC, and they haven't been able to
156 ! rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
157 ! which is used for the internal hardware interrupts as well. We just
158 ! have to reprogram the 8259's, and it isn't fun.
    !
    ! 希望以上一切正常。现在我们必须重新对中断进行编程 :-( 我们将它们放在正好
    ! 处于 Intel 保留的硬件中断后面，即 int 0x20--0x2F。在那里它们不会引起冲突。
    ! 不幸的是 IBM 在原 PC 机中搞糟了，以后也没有纠正过来。所以 PC 机 BIOS 把中断
    ! 放在了 0x08--0x0f，这些中断也被用于内部硬件中断。所以我们就必须重新对 8259
    ! 中断控制器进行编程，这一点都没意思。
    !
    ! PC 机使用 2 个 8259A 芯片，关于对可编程控制器 8259A 芯片的编程方法请参见本程序后的介绍。
    ! 第 162 行上定义的两个字（0x00eb）是直接使用机器码表示的两条相对跳转指令，起延时作用。
    ! 0xeb 是直接近跳转指令的操作码，带 1 个字节的相对位移值。因此跳转范围是-127 到 127。CPU
    ! 通过把这个相对位移值加到 EIP 寄存器中就形成一个新的有效地址。此时 EIP 指向下一条被执行
    ! 的指令。执行时所花费的 CPU 时钟周期数是 7 至 10 个。0x00eb 表示跳转值是 0 的一条指令，因
    ! 此还是直接执行下一条指令。这两条指令共可提供 14--20 个 CPU 时钟周期的延迟时间。在 as86
    ! 中没有表示相应指令的助记符，因此 Linus 在 setup.s 等一些汇编程序中就直接使用机器码来表
    ! 示这种指令。另外，每个空操作指令 NOP 的时钟周期数是 3 个，因此若要达到相同的延迟效果就
    ! 需要 6 至 7 个 NOP 指令。
159
    ! 8259 芯片主片端口是 0x20-0x21，从片端口是 0xA0-0xA1。输出值 0x11 表示初始化命令开始，
    ! 它是 ICW1 命令字，表示边沿触发、多片 8259 级连、最后要发送 ICW4 命令字。
160         mov     al,#0x11                ! initialization sequence
161         out     #0x20,al                ! send it to 8259A-1  ! 发送到 8259A 主芯片。
162         .word   0x00eb,0x00eb           ! jmp $+2, jmp $+2    ! '$'表示当前指令的地址，
163         out     #0xA0,al                ! and to 8259A-2      ! 再发送到 8259A 从芯片。
164         .word   0x00eb,0x00eb
    ! Linux 系统硬件中断号被设置成从 0x20 开始。参见表 3-2：硬件中断请求信号与中断号对应表。
165         mov     al,#0x20                ! start of hardware int's (0x20)
166         out     #0x21,al                ! 送主芯片 ICW2 命令字，设置起始中断号，要送奇端口。
167         .word   0x00eb,0x00eb
168         mov     al,#0x28                ! start of hardware int's 2 (0x28)
169         out     #0xA1,al                ! 送从芯片 ICW2 命令字，从芯片的起始中断号。
170         .word   0x00eb,0x00eb
171         mov     al,#0x04                ! 8259-1 is master
172         out     #0x21,al                ! 送主芯片 ICW3 命令字，主芯片的 IR2 连从芯片 INT。
                                            ! 参见代码列表后的说明。
173         .word   0x00eb,0x00eb
```

```
174         mov     al,#0x02                ! 8259-2 is slave
175         out     #0xA1,al                ! 送从芯片 ICW3 命令字，表示从芯片的 INT 连到主芯
                                            ! 片的 IR2 引脚上。
176         .word   0x00eb,0x00eb
177         mov     al,#0x01                ! 8086 mode for both
178         out     #0x21,al                ! 送主芯片 ICW4 命令字。8086 模式；普通 EOI、非缓冲
                                            ! 方式，需发送指令来复位。初始化结束，芯片就绪。
179         .word   0x00eb,0x00eb
180         out     #0xA1,al                ! 送从芯片 ICW4 命令字，内容同上。
181         .word   0x00eb,0x00eb
182         mov     al,#0xFF                ! mask off all interrupts for now
183         out     #0x21,al                ! 屏蔽主芯片所有中断请求。
184         .word   0x00eb,0x00eb
185         out     #0xA1,al                ! 屏蔽从芯片所有中断请求。
186
187 ! well, that certainly wasn't fun :-(. Hopefully it works, and we don't
188 ! need no steenking BIOS anyway (except for the initial loading :-).
189 ! The BIOS-routine wants lots of unnecessary data, and it's less
190 ! "interesting" anyway. This is how REAL programmers do it.
191 !
192 ! Well, now's the time to actually move into protected mode. To make
193 ! things as simple as possible, we do no register set-up or anything,
194 ! we let the gnu-compiled 32-bit programs do that. We just jump to
195 ! absolute address 0x00000, in 32-bit protected mode.
    !
    ! 哼，上面这段编程当然没劲:-(，但希望这样能工作，而且我们也不再需要乏味的 BIOS
    ! 了（除了初始加载:-)。BIOS 子程序要求很多不必要的数据，而且它一点都没趣。那是
    ! “真正”的程序员所做的事。
    !
    ! 好了，现在是真正开始进入保护模式的时候了。为了把事情做得尽量简单，我们并不对
    ! 寄存器内容进行任何设置。我们让 gnu 编译的 32 位程序去处理这些事。在进入 32 位保
    ! 护模式时我们仅是简单地跳转到绝对地址 0x00000 处。
196
    ! 下面设置并进入 32 位保护模式运行。首先加载机器状态字(lmsw-Load Machine Status Word)，
    ! 也称控制寄存器 CR0，其比特位 0 置 1 将导致 CPU 切换到保护模式，并且运行在特权级 0 中，即
    ! 当前特权级 CPL=0。此时段寄存器仍然指向与实地址模式中相同的线性地址处（在实地址模式下
    ! 线性地址与物理内存地址相同）。在设置该比特位后，随后一条指令必须是一条段间跳转指令以
    ! 用于刷新 CPU 当前指令队列。因为 CPU 是在执行一条指令之前就已从内存读取该指令并对其进行
    ! 解码。然而在进入保护模式以后那些属于实模式的预先取得的指令信息就变得不再有效。而一条
    ! 段间跳转指令就会刷新 CPU 的当前指令队列，即丢弃这些无效信息。另外，在 Intel 公司的手册
    ! 上建议 80386 或以上 CPU 应该使用指令“mov cr0,ax”切换到保护模式。lmsw 指令仅用于兼容以
    ! 前的 286 CPU。

197         mov     ax,#0x0001      ! protected mode (PE) bit       ! 保护模式比特位(PE)。
198         lmsw    ax              ! This is it!                   ! 就这样加载机器状态字!
199         jmpi    0,8             ! jmp offset 0 of segment 8 (cs) ! 跳转至 cs 段偏移 0 处。
    ! 我们已经将 system 模块移动到 0x00000 开始的地方，所以上句中的偏移地址是 0。而段值 8 已经
    ! 是保护模式下的段选择符了，用于选择描述符表和描述符表项以及所要求的特权级。段选择符长
    ! 度为 16 位（2 字节）；位 0-1 表示请求的特权级 0--3，但 Linux 操作系统只用到两级：0 级（内
    ! 核级）和 3 级（用户级）；位 2 用于选择全局描述符表（0）还是局部描述符表(1)；位 3-15 是描
    ! 述符表项的索引，指出选择第几项描述符。所以段选择符 8（0b0000,0000,0000,1000）表示请求
    ! 特权级 0、使用全局描述符表 GDT 中第 2 个段描述符项，该项指出代码的基地址是 0（参见 571 行），
    ! 因此这里的跳转指令就会去执行 system 中的代码。另外，
```

```
200
201 ! This routine checks that the keyboard command queue is empty
202 ! No timeout is used - if this hangs there is something wrong with
203 ! the machine, and we probably couldn't proceed anyway.
    ! 下面这个子程序检查键盘命令队列是否为空。这里不使用超时方法 -
    ! 如果这里死机，则说明 PC 机有问题，我们就没有办法再处理下去了。
    !
    ! 只有当输入缓冲器为空时（键盘控制器状态寄存器位 1 = 0）才可以对其执行写命令。
204 empty_8042:
205         .word   0x00eb,0x00eb
206         in      al,#0x64        ! 8042 status port          ! 读 AT 键盘控制器状态寄存器。
207         test    al,#2           ! is input buffer full?     ! 测试位 1，输入缓冲器满？
208         jnz     empty_8042      ! yes - loop
209         ret
210
211 ! Routine trying to recognize type of SVGA-board present (if any)
212 ! and if it recognize one gives the choices of resolution it offers.
213 ! If one is found the resolution chosen is given by al,ah (rows,cols).
    ! 下面是用于识别 SVGA 显示卡（若有的话）的子程序。若识别出一块就向用户
    ! 提供选择分辨率的机会，并把分辨率放入寄存器 al、ah（行、列）中返回。
    !
    ! 注意下面 215--566 行代码牵涉到众多显示卡端口信息，因此比较复杂。但由于这段代码与内核
    ! 运行关系不大，因此可以跳过不看。
    ! 下面首先显示 588 行上的 msg1 字符串（"按<回车键>查看存在的 SVGA 模式，或按任意键继续"），
    ! 然后循环读取键盘控制器输出缓冲器，等待用户按键。如果用户按下回车键就去检查系统具有
    ! 的 SVGA 模式，并在 AL 和 AH 中返回最大行列值，否则设置默认值 AL=25 行、AH=80 列并返回。
214
215 chsvga: cld
216         push    ds              ! 保存 ds 值。将在 231 行（或 490 或 492 行）弹出。
217         push    cs              ! 把默认数据段设置成和代码段同一个段。
218         pop     ds
219         mov     ax,#0xc000
220         mov     es,ax           ! es 指向 0xc000 段。此处是 VGA 卡上的 ROM BIOS 区。
221         lea     si,msg1         ! ds:si 指向 msg1 字符串。
222         call    prtstr          ! 显示以 NULL 结尾的 msg1 字符串。
223 nokey:  in      al,#0x60        ! 读取键盘控制器输出缓冲器（来自键盘的扫描码或命令）。
224         cmp     al,#0x82        ! 如果收到比 0x82 小的扫描码则是接通扫描码，因为 0x82 是
225         jb      nokey           ! 最小断开扫描码值。小于 0x82 表示还没有按键松开。
226         cmp     al,#0xe0        ! 如果扫描码大于 0xe0，表示收到的是扩展扫描码前缀。
227         ja      nokey
228         cmp     al,#0x9c        ! 如果断开扫描码是 0x9c，表示用户按下/松开了回车键，
229         je      svga            ! 于是程序跳转去检查系统是否具有 SVGA 模式。
230         mov     ax,#0x5019      ! 否则把 AX 中返回行列值默认设置为 AL=25 行、AH=80 列。
231         pop     ds
232         ret
    ! 下面根据 VGA 显示卡上的 ROM BIOS 指定位置处的特征数据串或者支持的特别功能来判断机器上
    ! 安装的是什么牌子的显示卡。本程序共支持 10 种显示卡的扩展功能。注意，此时程序已经在第
    ! 220 行把 es 指向 VGA 卡上 ROM BIOS 所在的段 0xc000（参见第 2 章）。
    ! 首先判断是不是 ATI 显示卡。我们把 ds:si 指向 595 行上 ATI 显示卡特征数据串，并把 es:si 指
    ! 向 VGA BIOS 中指定位置（偏移 0x31）处。因为该特征串共有 9 个字符（"761295520"），因此我
    ! 们循环比较这个特征串。如果相同则表示机器中的 VGA 卡是 ATI 牌子的，于是让 ds:si 指向该显
    ! 示卡可以设置的行列模式值 dscati（第 615 行），让 di 指向 ATI 卡可设置的行列个数和模式，
    ! 并跳转到标号 selmod（438 行）处进一步进行设置。
```

```
233 svga:   lea     si,idati        ! Check ATI 'clues'  ! 检查判断 ATI 显示卡的数据。
234         mov     di,#0x31        ! 特征串从 0xc000:0x0031 开始。
235         mov     cx,#0x09        ! 特征串有 9 个字节。
236         repe
237         cmpsb
238         jne     noati           ! 若特征串不同则表示不是 ATI 显示卡。跳转继续检测卡。
239         lea     si,dscati       ! 如果 9 个字节都相同，表示系统中有一块 ATI 牌显示卡。
240         lea     di,moati        ! 于是 si 指向 ATI 卡具有的可选行列值，di 指向可选个数
241         lea     cx,selmod       ! 和模式列表，然后跳转到 selmod（438 行）处继续处理。
242         jmp     cx

    ! 现在来判断是不是 Ahead 牌子的显示卡。首先向 EGA/VGA 图形索引寄存器 0x3ce 写入想访问的
    ! 主允许寄存器索引号 0x0f，同时向 0x3cf 端口（此时对应主允许寄存器）写入开启扩展寄存器
    ! 标志值 0x20。然后通过 0x3cf 端口读取主允许寄存器值，以检查是否可以设置开启扩展寄存器
    ! 标志。如果可以则说明是 Ahead 牌子的显示卡。注意 word 输出时 al➔端口 n，ah➔端口 n+1。
243 noati:  mov     ax,#0x200f      ! Check Ahead 'clues'
244         mov     dx,#0x3ce       ! 数据端口指向主允许寄存器（0x0f➔0x3ce 端口），
245         out     dx,ax           ! 并设置开启扩展寄存器标志（0x20➔0x3cf 端口）。
246         inc     dx              ! 然后再读取该寄存器，检查该标志是否被设置上。
247         in      al,dx
248         cmp     al,#0x20        ! 如果读取值是 0x20，则表示是 Ahead A 显示卡。
249         je      isahed          ! 如果读取值是 0x21，则表示是 Ahead B 显示卡。
250         cmp     al,#0x21        ! 否则说明不是 Ahead 显示卡，于是跳转继续检测其余卡。
251         jne     noahed
252 isahed: lea     si,dscahead     ! si 指向 Ahead 显示卡可选行列值表，di 指向扩展模式个
253         lea     di,moahead      ! 数和扩展模式号列表。然后跳转到 selmod（438 行）处继
254         lea     cx,selmod       ! 续处理。
255         jmp     cx

    ! 现在来检查是不是 Chips & Tech 生产的显示卡。通过端口 0x3c3（0x94 或 0x46e8）设置 VGA 允许
    ! 寄存器的进入设置模式标志（位 4），然后从端口 0x104 读取显示卡芯片集标识值。如果该标识值
    ! 是 0xA5，则说明是 Chips & Tech 生产的显示卡。
256 noahed: mov     dx,#0x3c3       ! Check Chips & Tech. 'clues'
257         in      al,dx           ! 从 0x3c3 端口读取 VGA 允许寄存器值，添加上进入设置模式
258         or      al,#0x10        ! 标志（位 4）后再写回。
259         out     dx,al
260         mov     dx,#0x104       ! 在设置模式时从全局标识端口 0x104 读取显示卡芯片标识值，
261         in      al,dx           ! 并暂时存放在 bl 寄存器中。
262         mov     bl,al
263         mov     dx,#0x3c3       ! 然后把 0x3c3 端口中的进入设置模式标志复位。
264         in      al,dx
265         and     al,#0xef
266         out     dx,al
267         cmp     bl,[idcandt]    ! 再把 bl 中标识值与位于 idcandt 处（第 596 行）的 Chips &
268         jne     nocant          ! Tech 的标识值 0xA5 作比较。如果不同则跳转比较下一种显卡。
269         lea     si,dsccandt     ! 让 si 指向这种显示卡的可选行列值表，di 指向扩展模式个数
270         lea     di,mocandt      ! 和扩展模式号列表。然后跳转到 selmod（438 行）进行设置
271         lea     cx,selmod       ! 显示模式的操作。
272         jmp     cx

    ! 现在检查是不是 Cirrus 显示卡。方法是使用 CRT 控制器索引号 0x1f 寄存器的内容来尝试禁止扩展
    ! 功能。该寄存器被称为鹰标（Eagle ID）寄存器，将其值高低半字节交换一下后写入端口 0x3c4 索
    ! 引的 6 号（定序/扩展）寄存器应该会禁止 Cirrus 显示卡的扩展功能。如果不会则说明不是 Cirrus
```

```
    ! 显示卡。因为从端口 0x3d4 索引的 0x1f 鹰标寄存器中读取的内容是鹰标值与 0x0c 索引号对应的显
    ! 存起始地址高字节寄存器内容异或操作之后的值，因此在读 0x1f 中内容之前我们需要先把显存起始
    ! 高字节寄存器内容保存后清零，并在检查后恢复之。另外，将没有交换过的 Eagle ID 值写到 0x3c4
    ! 端口索引的 6 号定序/扩展寄存器会重新开启扩展功能。
273 nocant: mov     dx,#0x3d4       ! Check Cirrus 'clues'
274         mov     al,#0x0c        ! 首先向 CRT 控制寄存器的索引寄存器端口 0x3d4 写入要访问
275         out     dx,al           ! 的寄存器索引号 0x0c（对应显存起始地址高字节寄存器），
276         inc     dx              ! 然后从 0x3d5 端口读入显存起始地址高字节并暂存在 bl 中，
277         in      al,dx           ! 再把显存起始地址高字节寄存器清零。
278         mov     bl,al
279         xor     al,al
280         out     dx,al
281         dec     dx              ! 接着向 0x3d4 端口输出索引 0x1f，指出我们要在 0x3d5 端口
282         mov     al,#0x1f        ! 访问读取“Eagle ID”寄存器内容。
283         out     dx,al
284         inc     dx
285         in      al,dx           ! 从 0x3d5 端口读取“Eagle ID”寄存器值，并暂存在 bh 中。
286         mov     bh,al           ! 然后把该值高低 4 比特互换位置存放到 cl 中。再左移 8 位
287         xor     ah,ah           ! 后放入 ch 中，而 cl 中放入数值 6。
288         shl     al,#4
289         mov     cx,ax
290         mov     al,bh
291         shr     al,#4
292         add     cx,ax
293         shl     cx,#8
294         add     cx,#6           ! 最后把 cx 值存放入 ax 中。此时 ah 中是换位后的“Eagle
295         mov     ax,cx           ! ID”值，al 中是索引号 6，对应定序/扩展寄存器。把 ah
296         mov     dx,#0x3c4       ! 写到 0x3c4 端口索引的定序/扩展寄存器应该会导致 Cirrus
297         out     dx,ax           ! 显示卡禁止扩展功能。
298         inc     dx
299         in      al,dx           ! 如果扩展功能真的被禁止，那么此时读入的值应该为 0。
300         and     al,al           ! 如果不为 0 则表示不是 Cirrus 显示卡，跳转继续检查其他卡。
301         jnz     nocirr
302         mov     al,bh           ! 是 Cirrus 显示卡，则利用第 286 行保存在 bh 中的“Eagle
303         out     dx,al           ! ID”原值再重新开启 Cirrus 卡扩展功能。此时读取的返回
304         in      al,dx           ! 值应该为 1。若不是，则仍然说明不是 Cirrus 显示卡。
305         cmp     al,#0x01
306         jne     nocirr
307         call    rst3d4          ! 恢复 CRT 控制器的显示起始地址高字节寄存器内容。
308         lea     si,dsccirrus    ! si 指向 Cirrus 显示卡的可选行列值，di 指向扩展模式个数
309         lea     di,mocirrus     ! 和对应模式号。然后跳转到 selmod 处去选择显示模式。
310         lea     cx,selmod
311         jmp     cx
    ! 该子程序利用保存在 bl 中的值（第 278 行）恢复 CRT 控制器的显示起始地址高字节寄存器内容。
312 rst3d4: mov     dx,#0x3d4
313         mov     al,bl
314         xor     ah,ah
315         shl     ax,#8
316         add     ax,#0x0c
317         out     dx,ax           ! 注意，这是 word 输出！！ al ➔0x3d4，ah ➔0x3d5。
318         ret

    ! 现在检查系统中是不是 Everex 显示卡。方法是利用中断 int 0x10 功能 0x70（ax =0x7000，
```

```
    ! bx=0x0000）调用 Everex 的扩展视频 BIOS 功能。对于 Everes 类型显示卡，该中断调用应该
    ! 会返回模拟状态，即有以下返回信息：
    ! al = 0x70，若是基于 Trident 的 Everex 显示卡；
    ! cl = 显示器类型：00-单色；01-CGA；02-EGA；03-数字多频；04-PS/2；05-IBM 8514；06-SVGA。
    ! ch = 属性：位 7-6：00-256K，01-512K，10-1MB，11-2MB；位 4-开启 VGA 保护；位 0-6845 模拟。
    ! dx = 板卡型号：位 15-4：板类型标识号；位 3-0：板修正标识号。
    !      0x2360-Ultragraphics II；0x6200-Vision VGA；0x6730-EVGA；0x6780-Viewpoint。
    ! di = 用 BCD 码表示的视频 BIOS 版本号。
319 nocirr: call    rst3d4          ! Check Everex 'clues'
320         mov     ax,#0x7000      ! 设置 ax = 0x7000, bx=0x0000，调用 int 0x10。
321         xor     bx,bx
322         int     0x10
323         cmp     al,#0x70        ! 对于 Everes 显示卡，al 中应该返回值 0x70。
324         jne     noevrx
325         shr     dx,#4           ! 忽律板修正号（位 3-0）。
326         cmp     dx,#0x678       ! 板类型号是 0x678 表示是一块 Trident 显示卡，则跳转。
327         je      istrid
328         cmp     dx,#0x236       ! 板类型号是 0x236 表示是一块 Trident 显示卡，则跳转。
329         je      istrid
330         lea     si,dsceverex    ! 让 si 指向 Everex 显示卡的可选行列值表，让 di 指向扩展
331         lea     di,moeverex     ! 模式个数和模式号列表。然后跳转到 selmod 去执行选择
332         lea     cx,selmod       ! 显示模式的操作。
333         jmp     cx
334 istrid: lea     cx,ev2tri       ! 是 Trident 类型的 Everex 显示卡，则跳转到 ev2tri 处理。
335         jmp     cx

    ! 现在检查是不是 Genoa 显示卡。方式是检查其视频 BIOS 中的特征数字串（0x77、0x00、0x66、
    ! 0x99）。注意，此时 es 已经在第 220 行被设置成指向 VGA 卡上 ROM BIOS 所在的段 0xc000。
336 noevrx: lea     si,idgenoa      ! Check Genoa 'clues'
337         xor     ax,ax           ! 让 ds:si 指向第 597 行上的特征数字串。
338         seg es
339         mov     al,[0x37]       ! 取 VGA 卡上 BIOS 中 0x37 处的指针（它指向特征串）。
340         mov     di,ax           ! 因此此时 es:di 指向特征数字串开始处。
341         mov     cx,#0x04
342         dec     si
343         dec     di
344 l1:     inc     si              ! 然后循环比较这 4 个字节的特征数字串。
345         inc     di
346         mov     al,(si)
347         seg es
348         and     al,(di)
349         cmp     al,(si)
350         loope   l1
351         cmp     cx,#0x00        ! 如果特征数字串完全相同，则表示是 Genoa 显示卡，
352         jne     nogen           ! 否则跳转去检查其他类型的显示卡。
353         lea     si,dscgenoa     ! 让 si 指向 Genoa 显示卡的可选行列值表，让 di 指向扩展
354         lea     di,mogenoa      ! 模式个数和模式号列表。然后跳转到 selmod 去执行选择
355         lea     cx,selmod       ! 显示模式的操作。
356         jmp     cx

    ! 现在检查是不是 Paradise 显示卡。同样是采用比较显示卡上 BIOS 中特征串（“VGA=”）的方式。
357 nogen:  lea     si,idparadise   ! Check Paradise 'clues'
358         mov     di,#0x7d        ! es:di 指向 VGA ROM BIOS 的 0xc000:0x007d 处，该处应该有
```

```
359         mov     cx,#0x04          ! 4 个字符“VGA=”。
360         repe
361         cmpsb
362         jne     nopara            ! 若有不同的字符，表示不是 Paradise 显示卡，于是跳转。
363         lea     si,dscparadise    ! 否则让 si 指向 Paradise 显示卡的可选行列值表，让 di 指
364         lea     di,moparadise     ! 向扩展模式个数和模式号列表。然后跳转到 selmod 处去选
365         lea     cx,selmod         ! 择想要使用的显示模式。
366         jmp     cx

    ! 现在检查是不是 Trident（TVGA）显示卡。TVGA 显示卡扩充的模式控制寄存器 1（0x3c4 端口索引
    ! 的 0x0e）的位 3--0 是 64K 内存页面个数值。这个字段值有一个特性：当写入时，我们需要首先把
    ! 值与 0x02 进行异或操作后再写入；当读取该值时则不需要执行异或操作，即异或前的值应该与写
    ! 入后再读取的值相同。下面代码就利用这个特性来检查是不是 Trident 显示卡。
367 nopara: mov     dx,#0x3c4         ! Check Trident 'clues'
368         mov     al,#0x0e          ! 首先在端口 0x3c4 输出索引号 0x0e，索引模式控制寄存器 1。
369         out     dx,al             ! 然后从 0x3c5 数据端口读入该寄存器原值，并暂存在 ah 中。
370         inc     dx
371         in      al,dx
372         xchg    ah,al
373         mov     al,#0x00          ! 然后我们向该寄存器写入 0x00，再读取其值➔al。
374         out     dx,al             ! 写入 0x00 就相当于“原值”0x02 异或 0x02 后的写入值，
375         in      al,dx             ! 因此若是 Trident 显示卡，则此后读入的值应该是 0x02。
376         xchg    al,ah             ! 交换后，al=原模式控制寄存器 1 的值，ah=最后读取的值。
    ! 下面语句右则英文注释是“真奇怪...书中并没有要求这样操作，但是这对我的 Trident 显示卡
    ! 起作用。如果不这样做，屏幕就会变模糊...”。这几行附带有英文注释的语句执行如下操作：
    ! 如果 bl 中原模式控制寄存器 1 的位 1 在置位状态的话就将其复位，否则就将位 1 置位。
    ! 实际上这几条语句就是对原模式控制寄存器 1 的值执行异或 0x02 的操作，然后用结果值去设置
    ! （恢复）原寄存器值。
377         mov     bl,al             ! Strange thing ... in the book this wasn't
378         and     bl,#0x02          ! necessary but it worked on my card which
379         jz      setb2             ! is a trident. Without it the screen goes
380         and     al,#0xfd          ! blurred ...
381         jmp     clrb2             !
382 setb2:  or      al,#0x02          !
383 clrb2:  out     dx,al
384         and     ah,#0x0f          ! 取 375 行最后读入值的页面个数字段（位 3--0），如果
385         cmp     ah,#0x02          ! 该字段值等于 0x02，则表示是 Trident 显示卡。
386         jne     notrid
387 ev2tri: lea     si,dsctrident     ! 是 Trident 显示卡，于是让 si 指向该显示卡的可选行列
388         lea     di,motrident      ! 值列表，让 di 指向对应扩展模式个数和模式号列表，然
389         lea     cx,selmod         ! 后跳转到 selmod 去执行模式选择操作。
390         jmp     cx

    ! 现在检查是不是 Tseng 显示卡（ET4000AX 或 ET4000/W32 类）。方法是对 0x3cd 端口对应的段
    ! 选择（Segment Select）寄存器执行读写操作。该寄存器高 4 位（位 7--4）是要进行读操作的
    ! 64KB 段号（Bank number），低 4 位（位 3--0）是指定要写的段号。如果指定段选择寄存器的
    ! 的值是 0x55（表示读、写第 6 个 64KB 段），那么对于 Tseng 显示卡来说，把该值写入寄存器
    ! 后再读出应该还是 0x55。
391 notrid: mov     dx,#0x3cd         ! Check Tseng 'clues'
392         in      al,dx             ! Could things be this simple ! :-)
393         mov     bl,al             ! 先从 0x3cd 端口读取段选择寄存器原值，并保存在 bl 中。
394         mov     al,#0x55          ! 然后我们向该寄存器中写入 0x55。再读入并放在 ah 中。
395         out     dx,al
```

```
396             in      al,dx
397             mov     ah,al
398             mov     al,bl           ! 接着恢复该寄存器的原值。
399             out     dx,al
400             cmp     ah,#0x55        ! 如果读取的就是我们写入的值，则表明是 Tseng 显示卡。
401             jne     notsen
402             lea     si,dsctseng     ! 于是让 si 指向 Tseng 显示卡的可选行列值的列表，让 di
403             lea     di,motseng      ! 指向对应扩展模式个数和模式号列表，然后跳转到 selmod
404             lea     cx,selmod       ! 去执行模式选择操作。
405             jmp     cx

    ! 下面检查是不是 Video7 显示卡。端口 0x3c2 是混合输出寄存器写端口，而 0x3cc 是混合输出寄存
    ! 器读端口。该寄存器的位 0 是单色/彩色标志。如果为 0 则表示是单色，否则是彩色。判断是不是
    ! Video7 显示卡的方式是利用这种显示卡的 CRT 控制扩展标识寄存器（索引号是 0x1f）。该寄存器
    ! 的值实际上就是显存起始地址高字节寄存器（索引号 0x0c）的内容和 0xea 进行异或操作后的值。
    ! 因此我们只要向显存起始地址高字节寄存器中写入一个特定值，然后从标识寄存器中读取标识值
    ! 进行判断即可。
    ! 通过对以上显示卡和这里 Video7 显示卡的检查分析，我们可知检查过程通常分为三个基本步骤。
    ! 首先读取并保存测试需要用到的寄存器原值，然后使用特定测试值进行写入和读出操作，最后恢
    ! 复原寄存器值并对检查结果作出判断。
406 notsen: mov     dx,#0x3cc       ! Check Video7 'clues'
407             in      al,dx
408             mov     dx,#0x3b4       ! 先设置 dx 为单色显示 CRT 控制索引寄存器端口号 0x3b4。
409             and     al,#0x01        ! 如果混合输出寄存器的位 0 等于 0（单色）则直接跳转，
410             jz      even7           ! 否则 dx 设置为彩色显示 CRT 控制索引寄存器端口号 0x3d4。
411             mov     dx,#0x3d4
412 even7:     mov     al,#0x0c        ! 设置寄存器索引号为 0x0c，对应显存起始地址高字节寄存器。
413             out     dx,al
414             inc     dx
415             in      al,dx           ! 读取显示内存起始地址高字节寄存器内容，并保存在 bl 中。
416             mov     bl,al
417             mov     al,#0x55        ! 然后在显存起始地址高字节寄存器中写入值 0x55，再读取出来。
418             out     dx,al
419             in      al,dx
420             dec     dx              ! 然后通过 CRTC 索引寄存器端口 0x3b4 或 0x3d4 选择索引号是
421             mov     al,#0x1f        ! 0x1f 的 Video7 显示卡标识寄存器。该寄存器内容实际上就是
422             out     dx,al           ! 显存起始地址高字节和 0xea 进行异或操作后的结果值。
423             inc     dx
424             in      al,dx           ! 读取 Video7 显示卡标识寄存器值，并保存在 bh 中。
425             mov     bh,al
426             dec     dx              ! 然后再选择显存起始地址高字节寄存器，恢复其原值。
427             mov     al,#0x0c
428             out     dx,al
429             inc     dx
430             mov     al,bl
431             out     dx,al
432             mov     al,#0x55        ! 随后我们来验证“Video7 显示卡标识寄存器值就是显存起始
433             xor     al,#0xea        ! 地址高字节和 0xea 进行异或操作后的结果值”。因此 0x55
434             cmp     al,bh           ! 和 0xea 进行异或操作的结果就应该等于标识寄存器的测试值。
435             jne     novid7          ! 若不是 Video7 显示卡，则设置默认显示行列值（492 行）。
436             lea     si,dscvideo7    ! 是 Video7 显示卡，于是让 si 指向该显示卡行列值表，让 di
437             lea     di,movideo7     ! 指向扩展模式个数和模式号列表。
```

```
    ! 下面根据上述代码判断出的显示卡类型以及取得的相关扩展模式信息（si 指向的行列值列表；di
    ! 指向扩展模式个数和模式号列表），提示用户选择可用的显示模式，并设置成相应显示模式。最后
    ! 子程序返回系统当前设置的屏幕行列值（ah = 列数；al=行数）。例如，如果系统中是 ATI 显示卡，
    ! 那么屏幕上会显示以下信息：
    ! Mode:  COLSxROWS:
    ! 0.     132 x 25
    ! 1.     132 x 44
    ! Choose mode by pressing the corresponding number.
    !
    ! 这段程序首先在屏幕上显示 NULL 结尾的字符串信息“Mode:  COLSxROWS:”。
438 selmod: push    si
439         lea     si,msg2
440         call    prtstr
441         xor     cx,cx
442         mov     cl,(di)         ! 此时 cl 中是检查出的显示卡的扩展模式个数。
443         pop     si
444         push    si
445         push    cx
    ! 然后并在每一行上显示出当前显示卡可选择的扩展模式行列值，供用户选用。
446 tbl:    pop     bx              ! bx = 显示卡的扩展模式总个数。
447         push    bx
448         mov     al,bl
449         sub     al,cl
450         call    dprnt           ! 以十进制格式显示 al 中的值。
451         call    spcing          ! 显示一个点再空 4 个空格。
452         lodsw                   ! 在 ax 中加载 si 指向的行列值，随后 si 指向下一个 word 值。
453         xchg    al,ah           ! 交换位置后 al = 列数。
454         call    dprnt           ! 显示列数；
455         xchg    ah,al           ! 此时 al 中是行数值。
456         push    ax
457         mov     al,#0x78        ! 显示一个小“x”，即乘号。
458         call    prnt1
459         pop     ax              ! 此时 al 中是行数值。
460         call    dprnt           ! 显示行数。
461         call    docr            ! 回车换行。
462         loop    tbl             ! 再显示下一个行列值。cx 中扩展模式计数值递减 1。
    ! 在扩展模式行列值都显示之后，显示“Choose mode by pressing the corresponding number.”，
    ! 然后从键盘口读取用户按键的扫描码，根据该扫描码确定用户选择的行列值模式号，并利用 ROM
    ! BIOS 的显示中断 int 0x10 功能 0x00 来设置相应的显示模式。
    ! 第 468 行的“模式个数值+0x80”是所按数字键-1 的松开扫描码。对于 0--9 数字键，它们的松开
    ! 扫描码分别是：0 - 0x8B；1 - 0x82；2 - 0x83；3 - 0x84；4 - 0x85；
    !              5 - 0x86；6 - 0x87；7 - 0x88；8 - 0x89；9 - 0x8A。
    ! 因此，如果读取的键盘松开扫描码小于 0x82 就表示不是数字键；如果扫描码等于 0x8B 则表示用户
    ! 按下数字 0 键。
463         pop     cx              ! cl 中是显示卡扩展模式总个数值。
464         call    docr
465         lea     si,msg3         ! 显示“请按相应数字键来选择模式。”
466         call    prtstr
467         pop     si              ! 弹出原行列值指针（指向显示卡行列值表开始处）。
468         add     cl,#0x80        ! cl + 0x80 = 对应“数字键-1”的松开扫描码。
469 nonum:  in      al,#0x60        ! Quick and dirty...
470         cmp     al,#0x82        ! 若键盘松开扫描码小于 0x82 则表示不是数字键，忽律该键。
471         jb      nonum
```

```
472         cmp     al,#0x8b        ! 若键盘松开扫描码等于 0x8b，表示按下了数字键 0。
473         je      zero
474         cmp     al,cl           ! 若扫描码大于扩展模式个数值对应的最大扫描码值，表示
475         ja      nonum           ! 键入的值超过范围或不是数字键的松开扫描码。否则表示
476         jmp     nozero          ! 用户按下并松开了一个非 0 数字按键。
    ! 下面把松开扫描码转换成对应的数字按键值，然后利用该值从模式个数和模式号列表中选择对应的
    ! 的模式号。接着调用机器 ROM BIOS 中断 int 0x10 功能 0 把屏幕设置成模式号指定的模式。最后再
    ! 利用模式号从显示卡行列值表中选择并在 ax 中返回对应的行列值。
477 zero:   sub     al,#0x0a        ! al = 0x8b - 0x0a = 0x81。
478 nozero: sub     al,#0x80        ! 再减去 0x80 就可以得到用户选择了第几个模式。
479         dec     al              ! 从 0 起计数。
480         xor     ah,ah           ! int 0x10 显示功能号=0（设置显示模式）。
481         add     di,ax
482         inc     di              ! di 指向对应的模式号（跳过第 1 个模式个数字节值）。
483         push    ax
484         mov     al,(di)         ! 取模式号➔al 中，并调用系统 BIOS 显示中断功能 0。
485         int     0x10
486         pop     ax
487         shl     ax,#1           ! 模式号乘 2，转换成为行列值表中对应值的指针。
488         add     si,ax
489         lodsw                   ! 取对应行列值到 ax 中（ah = 列数，al = 行数）。
490         pop     ds              ! 恢复第 216 行保存的 ds 原值。在 ax 中返回当前显示行列值。
491         ret

    ! 若都不是上面检测的显示卡，那么我们只好采用默认的 80 x 25 的标准行列值。
492 novid7: pop     ds              ! Here could be code to support standard 80x50,80x30
493         mov     ax,#0x5019
494         ret
495
496 ! Routine that 'tabs' to next col.
    ! 光标移动到下一制表位的子程序。
497
    ! 显示一个点字符'.'和 4 个空格。
498 spcing: mov     al,#0x2e        ! 显示一个点字符'.'。
499         call    prnt1
500         mov     al,#0x20
501         call    prnt1
502         mov     al,#0x20
503         call    prnt1
504         mov     al,#0x20
505         call    prnt1
506         mov     al,#0x20
507         call    prnt1
508         ret
509
510 ! Routine to print asciiz-string at DS:SI
    ! 显示位于 DS:SI 处以 NULL（0x00）结尾的字符串。
511
512 prtstr: lodsb
513         and     al,al
514         jz      fin
515         call    prnt1           ! 显示 al 中的一个字符。
516         jmp     prtstr
```

```
fin:    ret

! Routine to print a decimal value on screen, the value to be
! printed is put in al (i.e 0-255).
! 显示十进制数字的子程序。显示值放在寄存器 al 中（0--255）。

dprnt:  push    ax
        push    cx
        mov     ah,#0x00
        mov     cl,#0x0a
        idiv    cl
        cmp     al,#0x09
        jbe     lt100
        call    dprnt
        jmp     skip10
lt100:  add     al,#0x30
        call    prnt1
skip10: mov     al,ah
        add     al,#0x30
        call    prnt1
        pop     cx
        pop     ax
        ret

! Part of above routine, this one just prints ascii al
! 上面子程序的一部分。显示 al 中的一个字符。
! 该子程序使用中断 0x10 的 0x0E 功能，以电传方式在屏幕上写一个字符。光标会自动移到下一个
! 位置处。如果写完一行光标就会移动到下一行开始处。如果已经写完一屏最后一行，则整个屏幕
! 会向上滚动一行。字符 0x07（BEL）、0x08（BS）、0x0A(LF)和 0x0D（CR）被作为命令不会显示。
! 输入：AL -- 欲写字符；BH -- 显示页号；BL -- 前景显示色（图形方式时）。

prnt1:  push    ax
        push    cx
        mov     bh,#0x00        ! 显示页面。
        mov     cx,#0x01
        mov     ah,#0x0e
        int     0x10
        pop     cx
        pop     ax
        ret

! Prints <CR> + <LF>    ! 显示回车+换行。

docr:   push    ax
        push    cx
        mov     bh,#0x00
        mov     ah,#0x0e
        mov     al,#0x0a
        mov     cx,#0x01
        int     0x10
        mov     al,#0x0d
        int     0x10
        pop     cx
```

```
564         pop     ax
565         ret
566
    ! 全局描述符表开始处。描述符表由多个 8 字节长的描述符项组成。这里给出了 3 个描述符项。
    ! 第 1 项无用（568 行），但须存在。第 2 项是系统代码段描述符（570-573 行），第 3 项是系
    ! 统数据段描述符(575-578 行)。
567 gdt:
568         .word   0,0,0,0         ! dummy    ! 第 1 个描述符，不用。
569
    ! 在 GDT 表中这里的偏移量是 0x08。它是内核代码段选择符的值。
570         .word   0x07FF          ! 8Mb - limit=2047  (0--2047，因此是 2048*4096=8Mb)
571         .word   0x0000          ! base address=0
572         .word   0x9A00          ! code read/exec          ! 代码段为只读、可执行。
573         .word   0x00C0          ! granularity=4096, 386   ! 颗粒度为 4096，32 位模式。
574
    ! 在 GDT 表中这里的偏移量是 0x10。它是内核数据段选择符的值。
575         .word   0x07FF          ! 8Mb - limit=2047 (2048*4096=8Mb)
576         .word   0x0000          ! base address=0
577         .word   0x9200          ! data read/write         ! 数据段为可读可写。
578         .word   0x00C0          ! granularity=4096, 386   ! 颗粒度为 4096，32 位模式。
579
    ! 下面是加载中断描述符表寄存器 idtr 的指令 lidt 要求的 6 字节操作数。前 2 字节是 IDT 表的
    ! 限长，后 4 字节是 idt 表在线性地址空间中的 32 位基地址。CPU 要求在进入保护模式之前需设
    ! 置 IDT 表，因此这里先设置一个长度为 0 的空表。
580 idt_48:
581         .word   0               ! idt limit=0
582         .word   0,0             ! idt base=0L
583
    ! 这是加载全局描述符表寄存器 gdtr 的指令 lgdt 要求的 6 字节操作数。前 2 字节是 gdt 表的限
    ! 长，后 4 字节是 gdt 表的线性基地址。这里全局表长度设置为 2KB（0x7ff 即可），因为每 8
    ! 字节组成一个段描述符项，所以表中共可有 256 项。4 字节的线性基地址为 0x0009<<16 +
    ! 0x0200 + gdt，即 0x90200 + gdt。(符号 gdt 是全局表在本程序段中的偏移地址，见 205 行)
584 gdt_48:
585         .word   0x800           ! gdt limit=2048, 256 GDT entries
586         .word   512+gdt,0x9     ! gdt base = 0X9xxxx
587
588 msg1:   .ascii  "Press <RETURN> to see SVGA-modes available or any other key to continue."
589                 db      0x0d, 0x0a, 0x0a, 0x00
590 msg2:           .ascii  "Mode:  COLSxROWS:"
591                 db      0x0d, 0x0a, 0x0a, 0x00
592 msg3:           .ascii  "Choose mode by pressing the corresponding number."
593                 db      0x0d, 0x0a, 0x00
594
    ! 下面是 4 个显示卡的特征数据串。
595 idati:          .ascii  "761295520"
596 idcandt:        .byte   0xa5                    ! 标号 idcandt 意思是 ID of Chip AND Tech.
597 idgenoa:        .byte   0x77, 0x00, 0x66, 0x99
598 idparadise:     .ascii  "VGA="
599
    ! 下面是各种显示卡可使用的扩展模式个数和对应的模式号列表。其中每一行第 1 个字节是模式个
    ! 数值，随后的一些值是中断 0x10 功能 0（AH=0）可使用的模式号。例如从 602 行可知，对于 ATI
    ! 牌子的显示卡，除了标准模式以外还可使用两种扩展模式：0x23 和 0x33。
600 ! Manufacturer:   Numofmodes:   Mode:
```

```
    ! 厂家：            模式数量：      模式列表：
601
602 moati:          .byte   0x02,   0x23, 0x33
603 moahead:        .byte   0x05,   0x22, 0x23, 0x24, 0x2f, 0x34
604 mocandt:        .byte   0x02,   0x60, 0x61
605 mocirrus:       .byte   0x04,   0x1f, 0x20, 0x22, 0x31
606 moeverex:       .byte   0x0a,   0x03, 0x04, 0x07, 0x08, 0x0a, 0x0b, 0x16, 0x18, 0x21, 0x40
607 mogenoa:        .byte   0x0a,   0x58, 0x5a, 0x60, 0x61, 0x62, 0x63, 0x64, 0x72, 0x74, 0x78
608 moparadise:     .byte   0x02,   0x55, 0x54
609 motrident:      .byte   0x07,   0x50, 0x51, 0x52, 0x57, 0x58, 0x59, 0x5a
610 motseng:        .byte   0x05,   0x26, 0x2a, 0x23, 0x24, 0x22
611 movideo7:       .byte   0x06,   0x40, 0x43, 0x44, 0x41, 0x42, 0x45
612
    ! 下面是各种牌子 VGA 显示卡可使用的模式对应的列、行值列表。例如第 615 行表示 ATI 显示卡两
    ! 种扩展模式的列、行值分别是 132 x 25、 132 x 44。
613 !                       msb = Cols    lsb = Rows:
    !                       高字节=列数   低字节=行数：
614
615 dscati:          .word   0x8419, 0x842c                              ! ATI 卡可设置列、行值。
616 dscahead:        .word   0x842c, 0x8419, 0x841c, 0xa032, 0x5042      ! Ahead 卡可设置值。
617 dsccandt:        .word   0x8419, 0x8432
618 dsccirrus:       .word   0x8419, 0x842c, 0x841e, 0x6425
619 dsceverex:       .word   0x5022, 0x503c, 0x642b, 0x644b, 0x8419, 0x842c, 0x501e, 0x641b, 0xa040,
0x841e
620 dscgenoa:        .word   0x5020, 0x642a, 0x8419, 0x841d, 0x8420, 0x842c, 0x843c, 0x503c, 0x5042,
0x644b
621 dscparadise:     .word   0x8419, 0x842b
622 dsctrident:      .word   0x501e, 0x502b, 0x503c, 0x8419, 0x841e, 0x842b, 0x843c
623 dsctseng:        .word   0x503c, 0x6428, 0x8419, 0x841c, 0x842c
624 dscvideo7:      .word   0x502b, 0x503c, 0x643c, 0x8419, 0x842c, 0x841c
625
626 .text
627 endtext:
628 .data
629 enddata:
630 .bss
631 endbss:
```