程序 6 -1 linux/boot/bootsect.S

```
1  !
2  ! SYS_SIZE is the number of clicks (16 bytes) to be loaded.
3  ! 0x3000 is 0x30000 bytes = 196kB, more than enough for current
4  ! versions of linux
   ! SYS_SIZE 是要加载的系统模块长度，单位是节，每节 16 字节。0x3000 共为 0x30000 字节=196KB。
   ! 若以 1024 字节为 1KB 计，则应该是 192KB。对于当前内核版本这个空间长度已足够了。当该值为
   ! 0x8000 时，表示内核最大为 512KB。因为内存 0x90000 处开始存放移动后的 bootsect 和 setup
   ! 的代码，因此该值最大不得超过 0x9000（表示 584KB）。
   ! 这里感叹号'!'或分号';'表示程序注释语句开始。
5  !
   ! 头文件 linux/config.h 中定义了内核用到的一些常数符号和 Linus 自己使用的默认硬盘参数块。
   ! 例如其中定义了以下一些常数：
   ! DEF_SYSSIZE  = 0x3000 - 默认系统模块长度。单位是节，每节为 16 字节；
   ! DEF_INITSEG  = 0x9000 - 默认本程序代码移动目的段位置；
   ! DEF_SETUPSEG = 0x9020 - 默认 setup 程序代码段位置；
   ! DEF_SYSSEG   = 0x1000 - 默认从磁盘加载系统模块到内存的段位置。
6  #include <linux/config.h>
7  SYSSIZE = DEF_SYSSIZE          ! 定义一个标号或符号。指明编译连接后 system 模块的大小。
8  !
9  !       bootsect.s              (C) 1991 Linus Torvalds
10 !       modified by Drew Eckhardt
11 !
12 ! bootsect.s is loaded at 0x7c00 by the bios-startup routines, and moves
13 ! iself out of the way to address 0x90000, and jumps there.
14 !
15 ! It then loads 'setup' directly after itself (0x90200), and the system
16 ! at 0x10000, using BIOS interrupts.
17 !
18 ! NOTE! currently system is at most 8*65536 bytes long. This should be no
19 ! problem, even in the future. I want to keep it simple. This 512 kB
20 ! kernel size should be enough, especially as this doesn't contain the
21 ! buffer cache as in minix
22 !
23 ! The loader has been made as simple as possible, and continuos
24 ! read errors will result in a unbreakable loop. Reboot by hand. It
25 ! loads pretty fast by getting whole sectors at a time whenever possible.
   !
   ! 以下是前面文字的译文：
   !     bootsect.s              (C) 1991 Linus Torvalds
   !     Drew Eckhardt 修改
   !
   ! bootsect.s 被 ROM BIOS 启动子程序加载至 0x7c00 (31KB)处，并将自己移到了地址 0x90000
   ! (576KB)处，并跳转至那里。
   !
   ! 它然后使用 BIOS 中断将'setup'直接加载到自己的后面(0x90200)(576.5KB)，并将 system 加
   ! 载到地址 0x10000 处。
   !
   ! 注意! 目前的内核系统最大长度限制为(8*65536)(512KB)字节，即使是在将来这也应该没有问
   ! 题的。我想让它保持简单明了。这样 512KB 的最大内核长度应该足够了，尤其是这里没有象
   ! MINIX 中一样包含缓冲区高速缓冲。
   !
   ! 加载程序已经做得够简单了，所以持续地读操作出错将导致死循环。只能手工重启。只要可能，
```

```
! 通过一次读取所有的扇区，加载过程可以做得很快。
26
! 伪指令（伪操作符）.globl 或.global 用于定义随后的标识符是外部的或全局的，并且即使不
! 使用也强制引入。 .text、.data 和.bss 用于分别定义当前代码段、数据段和未初始化数据段。
! 在链接多个目标模块时，链接程序（ld86）会根据它们的类别把各个目标模块中的相应段分别
! 组合（合并）在一起。这里把三个段都定义在同一重叠地址范围中，因此本程序实际上不分段。
! 另外，后面带冒号的字符串是标号，例如下面的'begtext:'。
! 一条汇编语句通常由标号（可选）、指令助记符（指令名）和操作数三个字段组成。标号位于
! 一条指令的第一个字段。它代表其所在位置的地址，通常指明一个跳转指令的目标位置。
27 .globl begtext, begdata, begbss, endtext, enddata, endbss
28 .text                                ! 文本段（代码段）。
29 begtext:
30 .data                                ! 数据段。
31 begdata:
32 .bss                                 ! 未初始化数据段。
33 begbss:
34 .text                                ! 文本段（代码段）。
35
! 下面等号'='或符号'EQU'用于定义标识符或标号所代表的值。
36 SETUPLEN = 4                          ! nr of setup-sectors
                                        ! setup 程序代码占用磁盘扇区数(setup-sectors)值；
37 BOOTSEG  = 0x07c0                    ! original address of boot-sector
                                        ! bootsect 代码所在内存原始段地址；
38 INITSEG  = DEF_INITSEG               ! we move boot here - out of the way
                                        ! 将 bootsect 移到位置 0x90000 - 避开系统模块占用处；
39 SETUPSEG = DEF_SETUPSEG              ! setup starts here
                                        ! setup 程序从内存 0x90200 处开始；
40 SYSSEG   = DEF_SYSSEG                ! system loaded at 0x10000 (65536).
                                        ! system 模块加载到 0x10000（64 KB）处；
41 ENDSEG   = SYSSEG + SYSSIZE          ! where to stop loading
                                        ! 停止加载的段地址；
42
43 ! ROOT_DEV & SWAP_DEV are now written by "build".
! 根文件系统设备号 ROOT_DEV 和交换设备号 SWAP_DEV 现在由 tools 目录下的 build 程序写入。
! 设备号 0x306 指定根文件系统设备是第 2 个硬盘的第 1 个分区。当年 Linus 是在第 2 个硬盘上
! 安装了 Linux 0.11 系统，所以这里 ROOT_DEV 被设置为 0x306。在编译这个内核时你可以根据
! 自己根文件系统所在设备位置修改这个设备号。这个设备号是 Linux 系统老式的硬盘设备号命
! 名方式，硬盘设备号具体值的含义如下：
! 设备号=主设备号*256 + 次设备号（也即 dev_no = (major<<8) + minor ）
! （主设备号：1-内存,2-磁盘,3-硬盘,4-ttyx,5-tty,6-并行口,7-非命名管道）
! 0x300 - /dev/hd0 - 代表整个第 1 个硬盘；
! 0x301 - /dev/hd1 - 第 1 个盘的第 1 个分区；
!  …
! 0x304 - /dev/hd4 - 第 1 个盘的第 4 个分区；
! 0x305 - /dev/hd5 - 代表整个第 2 个硬盘；
! 0x306 - /dev/hd6 - 第 2 个盘的第 1 个分区；
!  …
! 0x309 - /dev/hd9 - 第 2 个盘的第 4 个分区；
! 从 Linux 内核 0.95 版后就已经使用与现在内核相同的命名方法了。
44 ROOT_DEV = 0                         ! 根文件系统设备使用与系统引导时同样的设备；
45 SWAP_DEV = 0                         ! 交换设备使用与系统引导时同样的设备；
46
! 伪指令 entry 迫使链接程序在生成的执行程序（a.out）中包含指定的标识符或标号。这里是
```

```
! 程序执行开始点。49 -- 58 行作用是将自身(bootsect)从目前段位置 0x07c0(31KB) 移动到
! 0x9000(576KB) 处，共 256 字（512 字节），然后跳转到移动后代码的 go 标号处，也即本程
! 序的下一语句处。
entry start                         ! 告知链接程序，程序从 start 标号开始执行。
start:
        mov     ax,#BOOTSEG         ! 将 ds 段寄存器置为 0x7C0；
        mov     ds,ax
        mov     ax,#INITSEG         ! 将 es 段寄存器置为 0x9000；
        mov     es,ax
        mov     cx,#256             ! 设置移动计数值=256 字（512 字节）；
        sub     si,si               ! 源地址    ds:si = 0x07C0:0x0000
        sub     di,di               ! 目的地址 es:di = 0x9000:0x0000
        rep                         ! 重复执行并递减 cx 的值，直到 cx = 0 为止。
        movw                        ! 即 movs 指令。从内存[si]处移动 cx 个字到[di]处。
        jmpi    go,INITSEG          ! 段间跳转（Jump Intersegment）。这里 INITSEG
                                    ! 指出跳转到的段地址，标号 go 是段内偏移地址。

! 从下面开始，CPU 在已移动到 0x90000 位置处的代码中执行。
! 这段代码设置几个段寄存器，包括栈寄存器 ss 和 sp。栈指针 sp 只要指向远大于 512 字节偏移
! （即地址 0x90200）处都可以。因为从 0x90200 地址开始处还要放置 setup 程序，而此时 setup
! 程序大约为 4 个扇区，因此 sp 要指向大于（0x200 + 0x200 * 4 +堆栈大小）位置处。这里 sp
! 设置为 0x9ff00 - 12（参数表长度），即 sp = 0xfef4。在此之上位置会存放一个自建的驱动
! 器参数表，见下面说明。实际上 BIOS 把引导扇区加载到 0x7c00 处并把执行权交给引导程序时，
! ss = 0x00，sp = 0xfffe。
! 另外，第 65 行上 push 指令的期望作用是想暂时把段值保留在栈中，然后等下面执行完判断磁道
! 扇区数后再弹出栈，并给段寄存器 fs 和 gs 赋值（第 109 行）。但是由于第 67、68 两语句修改
! 了栈段的位置，因此除非在执行栈弹出操作之前把栈段恢复到原位置，否则这样设计就是错误的。
! 因此这里存在一个 bug。改正的方法之一是去掉第 65 行，并把第 109 行修改成“mov ax,cs”。
go:     mov     ax,cs               ! 将 ds、es 和 ss 都置成移动后代码所在的段处(0x9000)。
        mov     dx,#0xfef4          ! arbitrary value >>512 - disk parm size

        mov     ds,ax
        mov     es,ax
        push    ax                  ! 临时保存段值（0x9000），供 109 行使用。（滑头!）

        mov     ss,ax               ! put stack at 0x9ff00 - 12.
        mov     sp,dx
/*
 *      Many BIOS's default disk parameter tables will not
 *      recognize multi-sector reads beyond the maximum sector number
 *      specified in the default diskette parameter tables - this may
 *      mean 7 sectors in some cases.
 *
 *      Since single sector reads are slow and out of the question,
 *      we must take care of this by creating new parameter tables
 *      (for the first disk) in RAM.  We will set the maximum sector
 *      count to 18 - the most we will encounter on an HD 1.44.
 *
 *      High doesn't hurt.  Low does.
 *
 *      Segments are as follows: ds=es=ss=cs - INITSEG,
 *              fs = 0, gs = parameter table segment
 */
```

```
/*
 *      对于多扇区读操作所读的扇区数超过默认磁盘参数表中指定的最大扇区数时，
 *      很多 BIOS 将不能进行正确识别。在某些情况下是 7 个扇区。
 *
 *      由于单扇区读操作太慢，不予以考虑，因此我们必须通过在内存中重创建新的
 *      参数表（为第 1 个驱动器）来解决这个问题。我们将把其中最大扇区数设置为
 *      18 -- 即在 1.44MB 磁盘上会碰到的最大数值。
 *
 *      这个数值大了不会出问题，但是太小就不行了。
 *
 *      段寄存器将被设置成：ds=es=ss=cs - 都为 INITSEG（0x9000），
 *      fs = 0，gs = 参数表所在段值。
 */
85 ! BIOS 设置的中断 0x1E 的中断向量值是软驱参数表地址。该向量值位于内存 0x1E * 4 = 0x78
   ! 处。这段代码首先从内存 0x0000:0x0078 处复制原软驱参数表到 0x9000:0xfef4 处，然后修改
   ! 表中的每磁道最大扇区数为 18。
86
87          push    #0                  ! 置段寄存器 fs = 0。
88          pop     fs                  ! fs:bx 指向存有软驱参数表地址处（指针的指针）。
89          mov     bx,#0x78            ! fs:bx is parameter table address
   ! 下面指令表示下一条语句的操作数在 fs 段寄存器所指的段中。它只影响其下一条语句。这里
   ! 把 fs:bx 所指内存位置处的表地址放到寄存器对 gs:si 中作为原地址。寄存器对 es:di =
   ! 0x9000:0xfef4 为目的地址。
90          seg fs
91          lgs     si,(bx)             ! gs:si is source
92
93          mov     di,dx               ! es:di is destination ! dx=0xfef4，在 61 行被设置。
94          mov     cx,#6               ! copy 12 bytes
95          cld                         ! 清方向标志。复制时指针递增。
96
97          rep                         ! 复制 12 字节的软驱参数表到 0x9000:0xfef4 处。
98          seg gs
99          movw
100
101         mov     di,dx               ! es:di 指向新表，修改表中偏移 4 处的最大扇区数为 18。
102         movb    4(di),*18           ! patch sector count
103
104         seg fs                      ! 让中断向量 0x1E 的值指向新表。
105         mov     (bx),di
106         seg fs
107         mov     2(bx),es
108
109         pop     ax                  ! 此时 ax 中是上面第 65 行保留下来的段值（0x9000）。
110         mov     fs,ax               ! 设置 fs = gs = 0x9000。
111         mov     gs,ax
112
113         xor     ah,ah               ! reset FDC ! 复位软盘控制器，让其采用新参数。
114         xor     dl,dl               ! dl = 0，第 1 个软驱。
115         int     0x13
116
117 ! load the setup-sectors directly after the bootblock.
118 ! Note that 'es' is already set up.
   ! 在 bootsect 程序块后紧根着加载 setup 模块的代码数据。
```

```
    ! 注意 es 已经设置好了。（在移动代码时 es 已经指向目的段地址处 0x9000）。
119
    ! 121--137 行的用途是利用 ROM BIOS 中断 INT 0x13 将 setup 模块从磁盘第 2 个扇区开始读到
    ! 0x90200 开始处，共读 4 个扇区。在读操作过程中如果读出错，则显示磁盘上出错扇区位置，
    ! 然后复位驱动器并重试，没有退路。
    ! INT 0x13 读扇区使用调用参数设置如下：
    ! ah = 0x02 - 读磁盘扇区到内存；al = 需要读出的扇区数量；
    ! ch = 磁道(柱面)号的低 8 位；    cl = 开始扇区(位 0-5)，磁道号高 2 位(位 6-7)；
    ! dh = 磁头号；                   dl = 驱动器号（如果是硬盘则位 7 要置位）；
    ! es:bx →指向数据缓冲区；   如果出错则 CF 标志置位，ah 中是出错码。
120 load_setup:
121         xor     dx, dx                  ! drive 0, head 0
122         mov     cx,#0x0002              ! sector 2, track 0
123         mov     bx,#0x0200              ! address = 512, in INITSEG
124         mov     ax,#0x0200+SETUPLEN     ! service 2, nr of sectors
125         int     0x13                    ! read it
126         jnc     ok_load_setup           ! ok - continue
127
128         push    ax                      ! dump error code ! 显示出错信息。出错码入栈。
129         call    print_nl                ! 屏幕光标回车。
130         mov     bp, sp                  ! ss:bp 指向欲显示的字（word）。
131         call    print_hex               ! 显示十六进制值。
132         pop     ax
133
134         xor     dl, dl                  ! reset FDC ! 复位磁盘控制器，重试。
135         xor     ah, ah
136         int     0x13
137         j       load_setup              ! j 即 jmp 指令。
138
139 ok_load_setup:
140
141 ! Get disk drive parameters, specifically nr of sectors/track
    ! 这段代码取磁盘驱动器的参数，实际上是取每磁道扇区数，并保存在位置 sectors 处。
    ! 取磁盘驱动器参数 INT 0x13 调用格式和返回信息如下：
    ! ah = 0x08     dl = 驱动器号（如果是硬盘则要置位 7 为 1）。
    ! 返回信息：
    ! 如果出错则 CF 置位，并且 ah = 状态码。
    ! ah = 0，  al = 0，           bl = 驱动器类型（AT/PS2）
    ! ch = 最大磁道号的低 8 位，   cl = 每磁道最大扇区数(位 0-5)，最大磁道号高 2 位(位 6-7)
    ! dh = 最大磁头数，            dl = 驱动器数量，
    ! es:di -→ 软驱磁盘参数表。
142
143         xor     dl,dl
144         mov     ah,#0x08            ! AH=8 is get drive parameters
145         int     0x13
146         xor     ch,ch
    ! 下面指令表示下一条语句的操作数在 cs 段寄存器所指的段中。它只影响其下一条语句。实际
    ! 上，由于本程序代码和数据都被设置处于同一个段中，即段寄存器 cs 和 ds、es 的值相同，因
    ! 此本程序中此处可以不使用该指令。
147         seg cs
    ! 下句保存每磁道扇区数。对于软盘来说（dl=0），其最大磁道号不会超过 256，ch 已经足够表
    ! 示它，因此 cl 的位 6-7 肯定为 0。又 146 行已置 ch=0，因此此时 cx 中是每磁道扇区数。
148         mov     sectors,cx
```

```
149         mov     ax,#INITSEG
150         mov     es,ax               ! 因为上面取磁盘参数中断改了 es 值，这里重新改回。
151
152 ! Print some inane message
    ! 显示信息：“'Loading'+回车+换行”，共显示包括回车和换行控制字符在内的 9 个字符。
    ! 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)。
    !
    ! BIOS 中断 0x10 功能号 ah = 0x13，显示字符串。
    ! 输入：al = 放置光标的方式及规定属性。0x01-表示使用 bl 中的属性值，光标停在字符串结尾处。
    ! es:bp 此寄存器对指向要显示的字符串起始位置处。cx = 显示的字符串字符数。bh = 显示页面号；
    ! bl = 字符属性。dh = 行号；dl = 列号。
153
154         mov     ah,#0x03            ! read cursor pos
155         xor     bh,bh               ! 首先读光标位置。返回光标位置值在 dx 中。
156         int     0x10                ! dh - 行（0--24）；dl - 列(0--79)。
157
158         mov     cx,#9               ! 共显示 9 个字符。
159         mov     bx,#0x0007          ! page 0, attribute 7 (normal)
160         mov     bp,#msg1            ! es:bp 指向要显示的字符串。
161         mov     ax,#0x1301          ! write string, move cursor
162         int     0x10                ! 写字符串并移动光标到串结尾处。
163
164 ! ok, we've written the message, now
165 ! we want to load the system (at 0x10000)
    ! 现在开始将 system 模块加载到 0x10000（64KB）开始处。
166
167         mov     ax,#SYSSEG
168         mov     es,ax               ! segment of 0x010000 ! es = 存放 system 的段地址。
169         call    read_it             ! 读磁盘上 system 模块，es 为输入参数。
170         call    kill_motor          ! 关闭驱动器马达，这样就可以知道驱动器的状态了。
171         call    print_nl            ! 光标回车换行。
172
173 ! After that we check which root-device to use. If the device is
174 ! defined (!= 0), nothing is done and the given device is used.
175 ! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending
176 ! on the number of sectors that the BIOS reports currently.
    ! 此后，我们检查要使用哪个根文件系统设备（简称根设备）。如果已经指定了设备(!=0)，
    ! 就直接使用给定的设备。否则就需要根据 BIOS 报告的每磁道扇区数来确定到底使用/dev/PS0
    ! (2,28)，还是 /dev/at0 (2,8)。
    !! 上面一行中两个设备文件的含义：
    !! 在 Linux 中软驱的主设备号是 2(参见第 43 行的注释)，次设备号 = type*4 + nr，其中
    !! nr 为 0-3 分别对应软驱 A、B、C 或 D；type 是软驱的类型（2→1.2MB 或 7→1.44MB 等）。
    !! 因为 7*4 + 0 = 28，所以 /dev/PS0 (2,28)指的是 1.44MB A 驱动器,其设备号是 0x021c
    !! 同理 /dev/at0 (2,8)指的是 1.2MB A 驱动器，其设备号是 0x0208。

    ! 下面 root_dev 定义在引导扇区 508，509 字节处，指根文件系统所在设备号。0x0306 指第 2
    ! 个硬盘第 1 个分区。这里默认为 0x0306 是因为当时 Linus 开发 Linux 系统时是在第 2 个硬
    ! 盘第 1 个分区中存放根文件系统。这个值需要根据你自己根文件系统所在硬盘和分区进行修
    ! 改。例如，如果你的根文件系统在第 1 个硬盘的第 1 个分区上，那么该值应该为 0x0301，即
    ! （0x01, 0x03）。如果根文件系统是在第 2 个 Bochs 软盘上，那么该值应该为 0x021D，即
    ! （0x1D,0x02）。当编译内核时，你可以在 Makefile 文件中另行指定你自己的值，内核映像
```

```
! 文件 Image 的创建程序 tools/build 会使用你指定的值来设置你的根文件系统所在设备号。
177
178         seg cs
179         mov     ax,root_dev         ! 取 508,509 字节处的根设备号并判断是否已被定义。
180         or      ax,ax
181         jne     root_defined
! 取上面第 148 行保存的每磁道扇区数。如果 sectors=15 则说明是 1.2MB 的驱动器；如果
! sectors=18，则说明是 1.44MB 软驱。因为是可引导的驱动器，所以肯定是 A 驱。
182         seg cs
183         mov     bx,sectors
184         mov     ax,#0x0208          ! /dev/ps0 - 1.2Mb
185         cmp     bx,#15              ! 判断每磁道扇区数是否=15
186         je      root_defined        ! 如果等于，则 ax 中就是引导驱动器的设备号。
187         mov     ax,#0x021c          ! /dev/PS0 - 1.44Mb
188         cmp     bx,#18
189         je      root_defined
190 undef_root:                         ! 如果都不一样，则死循环（死机）。
191         jmp undef_root
192 root_defined:
193         seg cs
194         mov     root_dev,ax         ! 将检查过的设备号保存到 root_dev 中。
195
196 ! after that (everyting loaded), we jump to
197 ! the setup-routine loaded directly after
198 ! the bootblock:
! 到此，所有程序都加载完毕，我们就跳转到被加载在 bootsect 后面的 setup 程序去。
! 下面段间跳转指令（Jump Intersegment）。跳转到 0x9020:0000(setup.s 程序开始处)去执行。
199
200         jmpi    0,SETUPSEG          !!!! 到此本程序就结束了。!!!!

! 下面是几个子程序。read_it 用于读取磁盘上的 system 模块。kill_moter 用于关闭软驱马达。
! 还有一些屏幕显示子程序。
201
202 ! This routine loads the system at address 0x10000, making sure
203 ! no 64kB boundaries are crossed. We try to load it as fast as
204 ! possible, loading whole tracks whenever we can.
205 !
206 ! in:   es - starting address segment (normally 0x1000)
207 !
! 该子程序将系统模块加载到内存地址 0x10000 处，并确定没有跨越 64KB 的内存边界。
! 我们试图尽快地进行加载，只要可能，就每次加载整条磁道的数据。
! 输入：es - 开始内存地址段值（通常是 0x1000）
!
! 下面伪操作符.word 定义一个 2 字节目标。相当于 C 语言程序中定义的变量和所占内存空间大小。
! '1+SETUPLEN'表示开始时已经读进 1 个引导扇区和 setup 程序所占的扇区数 SETUPLEN。
208 sread:  .word 1+SETUPLEN        ! sectors read of current track !当前磁道中已读扇区数。
209 head:   .word 0                 ! current head    !当前磁头号。
210 track:  .word 0                 ! current track   !当前磁道号。
211
212 read_it:
! 首先测试输入的段值。从盘上读入的数据必须存放在位于内存地址 64KB 的边界开始处，否则
! 进入死循环。清 bx 寄存器，用于表示当前段内存放数据的开始位置。
! 153 行上的指令 test 以比特位逻辑与两个操作数。若两个操作数对应的比特位都为 1，则结果
```

```
    ! 值的对应比特位为 1，否则为 0。该操作结果只影响标志（零标志 ZF 等）。例如若 AX=0x1000，
    ! 那么 test 指令的执行结果是(0x1000 & 0x0fff) = 0x0000，于是 ZF 标志置位。此时即下一条
    ! 指令 jne 条件不成立。
213         mov ax,es
214         test ax,#0x0fff
215 die:    jne die                 ! es must be at 64kB boundary ! es 值必须位于 64KB 边界!
216         xor bx,bx               ! bx is starting address within segment! bx 为段内偏移。
217 rp_read:
    ! 接着判断是否已经读入全部数据。比较当前所读段是否就是系统数据末端所处的段(#ENDSEG)，
    ! 如果不是就跳转至下面 ok1_read 标号处继续读数据。否则退出子程序返回。
218         mov ax,es
219         cmp ax,#ENDSEG          ! have we loaded all yet?    ! 是否已经加载了全部数据?
220         jb ok1_read
221         ret
222 ok1_read:
    ! 然后计算和验证当前磁道需要读取的扇区数，放在 ax 寄存器中。
    ! 根据当前磁道还未读取的扇区数以及段内数据字节开始偏移位置，计算如果全部读取这些未读
    ! 扇区，所读总字节数是否会超过 64KB 段长度的限制。若会超过，则根据此次最多能读入的字节
    ! 数 (64KB – 段内偏移位置)，反算出此次需要读取的扇区数。
223         seg cs
224         mov ax,sectors              ! 取每磁道扇区数。
225         sub ax,sread                ! 减去当前磁道已读扇区数。
226         mov cx,ax                   ! cx = ax = 当前磁道未读扇区数。
227         shl cx,#9                   ! cx = cx * 512 字节 + 段内当前偏移值(bx)。
228         add cx,bx                   !    = 此次读操作后，段内共读入的字节数。
229         jnc ok2_read                ! 若没有超过 64KB 字节，则跳转至 ok2_read 处执行。
230         je ok2_read
    ! 若加上此次将读磁道上所有未读扇区时会超过 64KB，则计算此时最多能读入的字节数：
    ! (64KB – 段内读偏移位置)，再转换成需读取的扇区数。其中 0 减某数就是取该数 64KB 的补值。
231         xor ax,ax
232         sub ax,bx
233         shr ax,#9
234 ok2_read:
    ! 读当前磁道上指定开始扇区（cl）和需读扇区数（al）的数据到 es:bx 开始处。然后统计当前
    ! 磁道上已经读取的扇区数并与磁道最大扇区数 sectors 作比较。如果小于 sectors 说明当前磁
    ! 道上的还有扇区未读。于是跳转到 ok3_read 处继续操作。
235         call read_track             ! 读当前磁道上指定开始扇区和需读扇区数的数据。
236         mov cx,ax                   ! cx = 该次操作已读取的扇区数。
237         add ax,sread                ! 加上当前磁道上已经读取的扇区数。
238         seg cs
239         cmp ax,sectors              ! 若当前磁道上的还有扇区未读，则跳转到 ok3_read 处。
240         jne ok3_read
    ! 若该磁道的当前磁头面所有扇区已经读取，则读该磁道的下一磁头面（1 号磁头）上的数据。
    ! 如果已经完成，则去读下一磁道。
241         mov ax,#1
242         sub ax,head                 ! 判断当前磁头号。
243         jne ok4_read                ! 如果是 0 磁头，则再去读 1 磁头面上的扇区数据。
244         inc track                   ! 否则去读下一磁道。
245 ok4_read:
246         mov head,ax                 ! 保存当前磁头号。
247         xor ax,ax                   ! 清当前磁道已读扇区数。
248 ok3_read:
    ! 如果当前磁道上的还有未读扇区，则首先保存当前磁道已读扇区数，然后调整存放数据处的开
```

```
    ! 始位置。若小于 64KB 边界值，则跳转到 rp_read(217 行)处，继续读数据。
249         mov sread,ax                ! 保存当前磁道已读扇区数。
250         shl cx,#9                   ! 上次已读扇区数*512 字节。
251         add bx,cx                   ! 调整当前段内数据开始位置。
252         jnc rp_read
    ! 否则说明已经读取 64KB 数据。此时调整当前段，为读下一段数据作准备。
253         mov ax,es
254         add ah,#0x10                ! 将段基址调整为指向下一个 64KB 内存开始处。
255         mov es,ax
256         xor bx,bx                   ! 清段内数据开始偏移值。
257         jmp rp_read                 ! 跳转至 rp_read(217 行)处，继续读数据。
258
    ! read_track 子程序。读当前磁道上指定开始扇区和需读扇区数的数据到 es:bx 开始处。参见
    ! 第 67 行下对 BIOS 磁盘读中断 int 0x13，ah=2 的说明。
    ! al - 需读扇区数；es:bx - 缓冲区开始位置。
259 read_track:
    ! 首先调用 BIOS 中断 0x10，功能 0x0e（以电传方式写字符），光标前移一位置。显示一个'.'。
260         pusha                       ! 压入所有寄存器（push all）。
261         pusha                       ! 为调用显示中断压入所有寄存器值。
262         mov ax, #0xe2e              ! loading... message 2e = .
263         mov bx, #7                  ! 字符前景色属性。
264         int 0x10
265         popa
266
    ! 然后正式进行磁道扇区读操作。
267         mov dx,track                ! 取当前磁道号。
268         mov cx,sread                ! 取当前磁道上已读扇区数。
269         inc cx                      ! cl = 开始读扇区。
270         mov ch,dl                   ! ch = 当前磁道号。
271         mov dx,head                 ! 取当前磁头号。
272         mov dh,dl                   ! dh = 磁头号，dl = 驱动器号(为 0 表示当前 A 驱动器)。
273         and dx,#0x0100              ! 磁头号不大于 1。
274         mov ah,#2                   ! ah = 2，读磁盘扇区功能号。
275
276         push dx                     ! save for error dump
277         push cx                     ! 为出错情况保存一些信息。
278         push bx
279         push ax
280
281         int 0x13
282         jc bad_rt                   ! 若出错，则跳转至 bad_rt。
283         add sp,#8                   ! 没有出错。因此丢弃为出错情况保存的信息。
284         popa
285         ret
286
    ! 读磁盘操作出错。则先显示出错信息，然后执行驱动器复位操作（磁盘中断功能号 0），再跳转
    ! 到 read_track 处重试。
287 bad_rt: push ax                     ! save error code
288         call print_all              ! ah = error, al = read
289
290
291         xor ah,ah
292         xor dl,dl
```

```
293         int 0x13
294
295
296         add sp, #10                 ! 丢弃为出错情况保存的信息。
297         popa
298         jmp read_track
299
300 /*
301  *      print_all is for debugging purposes.
302  *      It will print out all of the registers.  The assumption is that this is
303  *      called from a routine, with a stack frame like
304  *      dx
305  *      cx
306  *      bx
307  *      ax
308  *      error
309  *      ret <- sp
310  *
311 */
    /*
     *      子程序 print_all 用于调试目的。它会显示所有寄存器的内容。前提条件是需要从
     *      一个子程序中调用，并且栈帧结构为如下所示：（见上面）
     */
    ! 若标志寄存器的 CF=0，则不显示寄存器名称。
312
313 print_all:
314         mov cx, #5                  ! error code + 4 registers  ! 显示值个数。
315         mov bp, sp                  ! 保存当前栈指针 sp。
316
317 print_loop:
318         push cx                     ! save count left   ! 保存需要显示的剩余个数。
319         call print_nl               ! nl for readability  ! 为可读性先让光标回车换行。
320         jae no_reg                  ! see if register name is needed
321                                     ! 若 FLAGS 的标志 CF=0 则不显示寄存器名，于是跳转。
    ! 对应入栈寄存器顺序分别显示它们的名称“AX：”等。
322         mov ax, #0xe05 + 0x41 - 1   ! ah =功能号（0x0e）；al =字符（0x05 + 0x41 -1）。
323         sub al, cl
324         int 0x10
325
326         mov al, #0x58               ! X     ! 显示字符'X'。
327         int 0x10
328
329         mov al, #0x3a               ! :     ! 显示字符':'。
330         int 0x10
331
    ! 显示寄存器 bp 所指栈中内容。开始时 bp 指向返回地址。
332 no_reg:
333         add bp, #2                  ! next register   ! 栈中下一个位置。
334         call print_hex              ! print it        ! 以十六进制显示。
335         pop cx
336         loop print_loop
337         ret
338
```

```
    ! 调用 BIOS 中断 0x10，以电传方式显示回车换行。
print_nl:
        mov ax, #0xe0d              ! CR
        int 0x10
        mov al, #0xa                ! LF
        int 0x10
        ret

/*
 *      print_hex is for debugging purposes, and prints the word
 *      pointed to by ss:bp in hexadecmial.
 */
    /*
     *      子程序 print_hex 用于调试目的。它使用十六进制在屏幕上显示出
     *      ss:bp 指向的字。
     */

    ! 调用 BIOS 中断 0x10，以电传方式和 4 个十六进制数显示 ss:bp 指向的字。
print_hex:
        mov     cx, #4              ! 4 hex digits          ! 要显示 4 个十六进制数字。
        mov     dx, (bp)            ! load word into dx     ! 显示值放入 dx 中。
print_digit:
    ! 先显示高字节，因此需要把 dx 中值左旋 4 比特，此时高 4 比特在 dx 的低 4 位中。
        rol     dx, #4              ! rotate so that lowest 4 bits are used
        mov     ah, #0xe            ! 中断功能号。
        mov     al, dl              ! mask off so we have only next nibble
        and     al, #0xf            ! 放入 al 中并只取低 4 比特（1 个值）。
    ! 加上'0' 的 ASCII 码值 0x30，把显示值转换成基于数字'0' 的字符。若此时 al 值超过 0x39，
    ! 表示欲显示值超过数字 9，因此需要使用'A'--'F'来表示。
        add     al, #0x30           ! convert to 0 based digit, '0'
        cmp     al, #0x39           ! check for overflow
        jbe     good_digit
        add     al, #0x41 - 0x30 - 0xa      ! 'A' - '0' - 0xa

good_digit:
        int     0x10
        loop    print_digit         ! cx--。若 cx>0 则去显示下一个值。
        ret


/*
 * This procedure turns off the floppy drive motor, so
 * that we enter the kernel in a known state, and
 * don't have to worry about it later.
 */
    /* 这个子程序用于关闭软驱的马达，这样我们进入内核后就能
     * 知道它所处的状态，以后也就无须担心它了。
     */
    ! 下面第 377 行上的值 0x3f2 是软盘控制器的一个端口，被称为数字输出寄存器（DOR）端口。它是
    ! 一个 8 位的寄存器，其位 7--位 4 分别用于控制 4 个软驱（D--A）的启动和关闭。位 3--位 2 用于
    ! 允许/禁止 DMA 和中断请求以及启动/复位软盘控制器 FDC。 位 1--位 0 用于选择选择操作的软驱。
    ! 第 378 行上在 al 中设置并输出的 0 值，就是用于选择 A 驱动器，关闭 FDC，禁止 DMA 和中断请求，
    ! 关闭马达。有关软驱控制卡编程的详细信息请参见 kernel/blk_drv/floppy.c 程序后面的说明。
```

```
375 kill_motor:
376         push dx
377         mov dx,#0x3f2               ! 软驱控制卡的数字输出寄存器端口，只写。
378         xor al, al                  ! A 驱动器，关闭 FDC，禁止 DMA 和中断请求，关闭马达。
379         outb                        ! 将 al 中的内容输出到 dx 指定的端口去。
380         pop dx
381         ret
382
383 sectors:
384         .word 0                     ! 存放当前启动软盘每磁道的扇区数。
385
386 msg1:                               ! 开机调用 BIOS 中断显示的信息。共 9 个字符。
387         .byte 13,10                 ! 回车、换行的 ASCII 码。
388         .ascii "Loading"
389
    ! 表示下面语句从地址 508(0x1FC)开始，所以 root_dev 在启动扇区的第 508 开始的 2 个字节中。
390 .org 506
391 swap_dev:
392         .word SWAP_DEV              ! 这里存放交换系统所在设备号(init/main.c 中会用)。
393 root_dev:
394         .word ROOT_DEV              ! 这里存放根文件系统所在设备号(init/main.c 中会用)。

    ! 下面是启动盘具有有效引导扇区的标志。仅供 BIOS 中的程序加载引导扇区时识别使用。它必须
    ! 位于引导扇区的最后两个字节中。
395 boot_flag:
396         .word 0xAA55
397
398 .text
399 endtext:
400 .data
401 enddata:
402 .bss
403 endbss:
404
```