**2. windows 7 的 MBR 分析**

|  |
| --- |
| **前话：**   引导磁盘的 MBR 由 int 0x19 加载到 0x7c00，int 0x19 最后的工作是跳到 0x7c00 执行。 |

**一、 我的 bochs 上的 windows 7 的 disk images 介绍**

  在 bochs 上，我为 windows 7 分配了 10g 的磁盘空间， 这 10g 的分空间，bochs 是这样分配的：cylinders = 20805, heads = 16, spt = 63

  cylinders（柱面或道）是 20805，heads（磁头或面）是 16，spt（扇区/柱面 - 每道扇区数）为 63

  那么：磁盘空间为：disk size = cylinders \* 63 \* heads \* 512 = 20805 \* 63 \* 16 \* 512 = 10,737,377,280 bytes = 10g

**二、硬盘的 MBR（主引导记录）**

  MBR 是位于：0 扇区（逻辑扇区） 即：0 柱面（0-cylinder），0 磁头（0-head），1 扇区（1-sector）

  大小为 512 bytes。

整个 MBR 的结构如下：

|  |  |  |
| --- | --- | --- |
| **位置（hex）** | **大小（bytes）** | **描述** |
| 000 - 162 | 354 bytes | 硬盘 MBR 引导记录（代码区） |
| 162 - 1BD | 92 bytes | MBR 数据区域 |
| 1BE - 1CD | 16 bytes | 分区表 1 |
| 1CE - 1DD | 16 bytes | 分区表 2 |
| 1DE - 1ED | 16 bytes | 分区表 3 |
| 1EE - 1FD | 16 bytes | 分区表 4 |
| 1FE - 1FF | 2 bytes | MBR 标志（55AA） |

**1、磁盘分区表（Disk Partition Table）**

  在 MBR 里的后 64 个字节里是磁盘的分区表结构，可定义 4 个分区，每个分区 16 bytes，从 0x1be ~ 0x1fe 共 64 bytes。

表1：磁盘分区表1（DPT1）结构

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **位置（hex）** | **大小（bytes）** | **意义** | **描述** | |
| 1BE | 1 | 分区的启动标志 | 80 = | 可启动分区 |
| 00 = | 不可启动区 |
| 1BF - 1C1 | 3 | 分区的起始扇区 | 1BF = | heads，起始 heads （1 个 bytes） |
| 1C0 = | sector，低 6 bits 表示起始 sector，这里只用该节字的低 6 bits 来表示 sector |
| 1C1 = | cylinder，1C0 的高 2 btis 加上 1C1 的 8 bits 组成 10 bits 表示起始 cylinder |
| 1C2 | 1 | 文件系统 | 如：07 表示 ntfs 系统，详见：[文件系统](http://www.mouseos.com/win7/fs_id.html" \t "_blank) | |
| 1C3 - 1C5 | 3 | 分区的结束扇区 | 其意义和起始扇区一致 | |
| 1C6 - 1C9 | 4 | 此分区前扇区数 | 这 4 bytes 表示此分区前有多少扇区（实际上等于此分区的起始扇区号），以 little-endian 排列的。 | |
| 1CA - 1CD | 4 | 此分区扇区数 | 这 4 bytes 用来表示此分区共有多少扇区，同样是以 little-endian 排列的。 | |

上面是分区1 的结构，与分区2 ~ 分区4 结构一致 。

**2、下面看看我的 bochs 上安装的 windows 7 分区表**

**（1）分区1结构**

|  |
| --- |
| /\*\*\*\*\* DPT1 分区表1：从 0x1be 到 0x1cd  \*\*\*\*\*\*/  000001BE  80          /\* boot indicator \*/                                 /\* 00:  不可启动分区　\*/                                /\* 80:  可启动分区（只可有1个启动分区）\*/  000001BF  20           /\* 起始 header 号 \*/ 000001C0  21           /\* 起始 sector 号 \*/ 000001C1  00           /\* 起始 cylinder 号 \*/  000001C2  07           /\* 系统属性 ID 标记 \*/                         00h：未知操作系统                         01h：DOS FAT12（16位扇区数）                         02h：XENIX                         04h：DOS FAT16（16位扇区数）                         05h：DOS 扩展分区(DOS 3.3+)                         06h：DOS 4.0 (Compaq 3.31), 32位扇区数                         07h：HPFS/NTFS                        0ah：OS/2                        0bh：win95 fat32                        0ch：win95 fat32 (LBA)                        ... ...  000001C3  DF           /\* 结束 header 号 \*/ 000001C4  13           /\* 结束 sector 号 \*/ 000001C5  0C           /\* 结束 cylinder 号 \*/  000001C6  00080000      /\* 此分区前的扇区总数 \*/  000001CA  00200300     /\* 此分区的扇区总数 \*/ |

可以看出分区1是：

★ boot indicator = 80 表示：该分区是可启动的分区。

★ 起始扇区是：0 - cylinder， 0x20 - heads， 0x21 - sector

  1 byte 的 head 最大可表示：0xFF 个 heads，即：255 个 heads

  6 bits 的 sector 最大表示： 0x3F 个 sector 即：63 个 sectors

  10 bits 的 cylinder 最大表示：0x3FF 个 cylinder 即： 1023 个 cylinders

  每个 cylinder 的扇区为：heads \* 63 = 255 \* 63 = 16065 sectors

那么：分区1的起始扇区是第几扇分区呢？ 它的逻辑分区 L 是：

L = cylinder \* 16065 + heads \* 63 + sector - 1

 = 0 \* 16065 + 0x20 \* 63 + 21 - 1

 = 0x20 \* 63 + 0x20

 = 2048

  因此，分区1的起始扇区是第 2048 号扇区，即说明，此分区前面有 2048 个扇区被保留（扇区号从 0 开始编号）

★ 分区1 的文件系统是 ntfs 文件系统。

★ 分区1 的结束扇区是： 0xdf - heads， 0x13 - sectors， 0x0c - cylinder

那么，结束扇区是第几号扇区呢？

L = cylinder \* 16065 + heads \* 63 + sectors - 1

 = 0x0c \* 16065 + 0xdf \* 63 + 0x13 - 1

 = 192780 + 14049 + 18

 = 206847

结束扇区是第 206847 号扇区。

因此：分区的扇区数是：206847 - 2048 + 1 = 204800 个扇区

★ 分区1 前的扇区数是： 00080000

由于是 little-endian 排列，它的 size 是 0x00000800 即：2048 个扇区

正好符合前面说的 2048 个扇区。

★ 分区1 的扇区数是： 00200300

同样是 little-endian 排列，它的 size 是 0x00032000 即： 204800 个扇区

正好符合前面说的 204800 个扇区数。

**由此可见，分区1 共有 100M 的磁盘空间。**

**2、分区 2 的结构**

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\* DPT2 分区表2：从 0x1CE 到 0x1DD \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  000001CE  00              /\* boot indicator \*/  000001CF  DF            /\* 起始 header \*/ 000001D0  14             /\* sector \*/   000001D1  0C             /\* cylinder \*/  000001D2  07            /\* file type flag \*/  000001D3  FE            /\* 结束 header \*/ 000001D4  FF            /\* sector \*/ 000001D5  FF            /\* cylinders \*/  000001D6  00280300                    /\* 分区前扇区数 \*/  000001DA  00C83C01                 /\* 此分区扇区数 \*/ |

可以看出分区2 是:

★ 该分区是不可启动的分区。

★ 起始扇区是：L = 0x0c \* 16065 + 0xdf \* 63 + 0x14 - 1 = 206849 号扇区，正好是分区1的结束扇区的下一个扇区

★ 文件系统同样是 ntfs

★ 结束扇区是：L = 0x3ff \* 16065 + 0xfe \* 63 + 0x3f - 1 = 16450559 号扇区。

★ 分区前扇区数：00280300 = 0x00032800 = 206848 扇区。

★ 分区的扇区数：00c83c01 = 0x013cc800 = 20760576 扇区数

**那么，分区2的大小是 ≈ 10g 约等于 10g， 由于分区 1 的大小是 100M bytes，分区2大小约为 10g**

这里有一个现象：

  前面说的结束分区是 16450559 号扇区，明显是不对的。这是因为，cylinder 最大只能表示 1023 个，超过部分没办法表示，这里就以分区扇数为准。

:(

|  |
| --- |
| **由此可见：**   在 windows 7 的默认分区下，分区1 只是启动分区，分区2 则整个系统的分区。 |

**三、 windows 7 的 MBR 代码分析**

下面是 windwos 7 MBR 的完整结构：

|  |
| --- |
| 00000000  33C0              xor ax,ax 00000002  8ED0              mov ss,ax 00000004  BC007C            mov sp,0x7c00 00000007  8EC0              mov es,ax 00000009  8ED8              mov ds,ax  0000000B  BE007C            mov si,0x7c00 0000000E  BF0006            mov di,0x600 00000011  B90002            mov cx,0x200 00000014  FC                cld 00000015  F3A4              rep movsb  00000017  50                push ax 00000018  681C06            push word 0x61c 0000001B  CB                retf  0000001C  FB                sti 0000001D  B90400            mov cx,0x4 00000020  BDBE07            mov bp,0x7be 00000023  807E0000          cmp byte [bp+0x0],0x0 00000027  7C0B              jl 0x34 00000029  0F850E01          jnz word 0x13b  0000002D  83C510            add bp,byte +0x10 00000030  E2F1              loop 0x23 00000032  CD18              int 0x18  00000034  885600            mov [bp+0x0],dl 00000037  55                push bp 00000038  C6461105          mov byte [bp+0x11],0x5 0000003C  C6461000          mov byte [bp+0x10],0x0 00000040  B441              mov ah,0x41 00000042  BBAA55            mov bx,0x55aa 00000045  CD13              int 0x13 00000047  5D                pop bp 00000048  720F              jc 0x59 0000004A  81FB55AA          cmp bx,0xaa55 0000004E  7509              jnz 0x59 00000050  F7C10100          test cx,0x1 00000054  7403              jz 0x59  00000056  FE4610            inc byte [bp+0x10]  00000059  6660              pushad 0000005B  807E1000          cmp byte [bp+0x10],0x0 0000005F  7426              jz 0x87 00000061  666800000000      push dword 0x0 00000067  66FF7608          push dword [bp+0x8] 0000006B  680000            push word 0x0 0000006E  68007C            push word 0x7c00 00000071  680100            push word 0x1 00000074  681000            push word 0x10 00000077  B442              mov ah,0x42 00000079  8A5600            mov dl,[bp+0x0] 0000007C  8BF4              mov si,sp 0000007E  CD13              int 0x13 00000080  9F                lahf 00000081  83C410            add sp,byte +0x10 00000084  9E                sahf 00000085  EB14              jmp short 0x9b 00000087  B80102            mov ax,0x201 0000008A  BB007C            mov bx,0x7c00 0000008D  8A5600            mov dl,[bp+0x0] 00000090  8A7601            mov dh,[bp+0x1] 00000093  8A4E02            mov cl,[bp+0x2] 00000096  8A6E03            mov ch,[bp+0x3] 00000099  CD13              int 0x13 0000009B  6661              popad 0000009D  731C              jnc 0xbb 0000009F  FE4E11            dec byte [bp+0x11] 000000A2  750C              jnz 0xb0 000000A4  807E0080          cmp byte [bp+0x0],0x80 000000A8  0F848A00          jz word 0x136 000000AC  B280              mov dl,0x80 000000AE  EB84              jmp short 0x34 000000B0  55                push bp 000000B1  32E4              xor ah,ah 000000B3  8A5600            mov dl,[bp+0x0] 000000B6  CD13              int 0x13 000000B8  5D                pop bp 000000B9  EB9E              jmp short 0x59 000000BB  813EFE7D55AA      cmp word [0x7dfe],0xaa55 000000C1  756E              jnz 0x131 000000C3  FF7600            push word [bp+0x0] 000000C6  E88D00            call word 0x156 000000C9  7517              jnz 0xe2 000000CB  FA                cli 000000CC  B0D1              mov al,0xd1 000000CE  E664              out 0x64,al 000000D0  E88300            call word 0x156 000000D3  B0DF              mov al,0xdf 000000D5  E660              out 0x60,al 000000D7  E87C00            call word 0x156 000000DA  B0FF              mov al,0xff 000000DC  E664              out 0x64,al 000000DE  E87500            call word 0x156 000000E1  FB                sti 000000E2  B800BB            mov ax,0xbb00 000000E5  CD1A              int 0x1a 000000E7  6623C0            and eax,eax 000000EA  753B              jnz 0x127 000000EC  6681FB54435041    cmp ebx,0x41504354 000000F3  7532              jnz 0x127 000000F5  81F90201          cmp cx,0x102 000000F9  722C              jc 0x127 000000FB  666807BB0000      push dword 0xbb07 00000101  666800020000      push dword 0x200 00000107  666808000000      push dword 0x8 0000010D  6653              push ebx 0000010F  6653              push ebx 00000111  6655              push ebp 00000113  666800000000      push dword 0x0 00000119  6668007C0000      push dword 0x7c00 0000011F  6661              popad 00000121  680000            push word 0x0 00000124  07                pop es 00000125  CD1A              int 0x1a 00000127  5A                pop dx 00000128  32F6              xor dh,dh 0000012A  EA007C0000        jmp word 0x0:0x7c00 0000012F  CD18              int 0x18 00000131  A0B707            mov al,[0x7b7] 00000134  EB08              jmp short 0x13e 00000136  A0B607            mov al,[0x7b6] 00000139  EB03              jmp short 0x13e 0000013B  A0B507            mov al,[0x7b5] 0000013E  32E4              xor ah,ah 00000140  050007            add ax,0x700 00000143  8BF0              mov si,ax 00000145  AC                lodsb 00000146  3C00              cmp al,0x0 00000148  7409              jz 0x153 0000014A  BB0700            mov bx,0x7 0000014D  B40E              mov ah,0xe 0000014F  CD10              int 0x10 00000151  EBF2              jmp short 0x145 00000153  F4                hlt 00000154  EBFD              jmp short 0x153 00000156  2BC9              sub cx,cx 00000158  E464              in al,0x64 0000015A  EB00              jmp short 0x15c 0000015C  2402              and al,0x2 0000015E  E0F8              loopne 0x158 00000160  2402              and al,0x2 00000162  C3                ret    /\*\*\*\*\*\*\*\*\*\* 下面的区域是 mbr 常量符： mbr\_str　\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* char \*mbr\_str = "Invalid partition table." "Error loading operation system." "Missing operation system." \*/  00000163  496E7661 6C696420 70617274 6974696F  00000173  6E207461 626C6500 4572726F 72206C6F 00000183  6164696E 67206F70 65726174 696E6720 00000193  73797374 656D004D 69737369 6E67206F 000001A3  70657261 74696E67 20737973 74656D  000001B2  00 000001B3  00 000001B4  00 000001B5  63 000001B6  7B 000001B7  9A 000001B8  46 000001B9  69 000001BA  6C 000001BB  65 000001BC  20 000001BD  73  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* disk partion table：从 0x1be 到 0x1fd \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\*\*\*\*\* DPT1 分区表1：从 0x1be 到 0x1cd \*\*\*\*\*\*/  000001BE  80          /\* boot indicator \*/                       /\* 00:  不可启动分区　\*/                       /\* 80:  可启动分区（只可有1个启动分区）\*/  000001BF  20           /\* 起始 header 号 \*/ 000001C0  21           /\* 起始 sector 号 \*/ 000001C1  00           /\* 起始 cylinder 号 \*/  000001C2  07           /\* 系统属性 ID 标记                         00h：未知操作系统                         01h：DOS FAT12（16位扇区数）                         02h：XENIX                         04h：DOS FAT16（16位扇区数）                         05h：DOS 扩展分区(DOS 3.3+)                         06h：DOS 4.0 (Compaq 3.31), 32位扇区数                         07h：HPFS/NTFS                        0ah：OS/2                        0bh：win95 fat32                        0ch：win95 fat32 (LBA)                        ... ...                        \*/ 000001C3  DF           /\* 结束 header 号 \*/ 000001C4  13           /\* 结束 sector 号 \*/ 000001C5  0C           /\* 结束 cylinder 号 \*/  000001C6  00080000      /\* 此分区前的扇区总数 \*/  000001CA  00200300     /\* 此分区的扇区总数 \*/    /\*\*\*\*\*\*\*\*\*\*\*\*\*\* DPT2 分区表2：从 0x1CE 到 0x1DD \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 000001CE  00  000001CF  DF  /\* 起始 header \*/ 000001D0  14            /\* sector \*/   000001D1  0C            /\* cylinder \*/  000001D2  07            /\* file type flag \*/  000001D3  FE            /\* 结束 header \*/ 000001D4  FF            /\* sector \*/ 000001D5  FF            /\* cylinders \*/  000001D6  00280300  000001DA  00C83C01    /\*\*\*\*\*\*\*\*  1DE - 1ED  =  DPT3 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 000001DE  00 000001DF  00 000001E0  00 000001E1  00 000001E2  00 000001E3  00 000001E4  00 000001E5  00 000001E6  00000000 000001EA  00000000  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  1EE - 1FD = DPT4 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 000001EE  00 000001EF  00 000001F0  00 000001F1  00 000001F2  00 000001F3  00 000001F4  00  000001F5  00 000001F6  00000000 000001FA  00000000  000001FE  55AA            /\* MBR 标记 \*/ |

分区3 和分区4 都是空的。

上面的代码是用 nasm 中的 ndisasmw 反汇编出来的结果的。事实上，还有一部分不是代码部分，是数据区域

从 0x00 - 0x162 ：这是 MBR 的主体代码区域

从 0x162 - 0x1bd：这是 MBR 用到的数据区域

从 0x1be - 0x1fd: 这是 MBR 的磁盘分区表区域

从 0x1fe - 0x1ff：这是 MBR 的标志 "55AA"

**1、 mbr 前期准备工作**

|  |
| --- |
| 00000000  33C0              xor ax,ax 00000002  8ED0              mov ss,ax 00000004  BC007C            mov sp,0x7c00 00000007  8EC0              mov es,ax 00000009  8ED8              mov ds,ax  0000000B  BE007C            mov si,0x7c00 0000000E  BF0006            mov di,0x600 00000011  B90002            mov cx,0x200 00000014  FC                cld 00000015  F3A4              rep movsb  00000017  50                push ax 00000018  681C06            push word 0x61c 0000001B  CB                retf |

代码中，把所有 segment 都设为 0，然后从 0x7c00 开始复制 512 bytes 到 0x600 区域，然后转到 0x61c 开始执行

实际上是把 mbr 重新复制到 0x600 处，然后从指令 retf 的下一条指令，即：指令 sti 处开始继续执行下去。

但是此时，ip 是位于 0x61c 处，0x61c 与 0x7c1c 代码是一样的。

|  |
| --- |
| **关于 0x7c00：**   bios 执行完最后的一步工作是：0x19 中断从硬盘 0 扇区（即：MBR）读取 1 个 sector（512 bytes）到内存 0x7c00 处。  再跳到 0x7c00 处交由 MBR 代码继续执行。 |

**2、 读磁盘到 0x7c00**

  MBR 的下一步工作就是读磁盘，下面来看一看 MBR 是如何读磁盘。

**（1）读磁盘前检查分区是否可启动**

代码接着走到：

|  |
| --- |
| 0000001C  FB                sti 0000001D  B90400            mov cx,0x4 00000020  BDBE07            mov bp,0x7be               /\* MBR 的 0x1be 即：DPT1 \*/ 00000023  807E0000          cmp byte [bp+0x0],0x0      /\* 检查 DPT1 的 boot indicator \*/ 00000027  7C0B              jl 0x34                    /\* 如果不是 0 的话，视为 0x80，进行下一次读工作 \*/ 00000029  0F850E01          jnz word 0x13b             /\* 不是，打印出错信息，最后 hlt 停机 \*/ |

  跳到 0x34 处是读磁盘的代码，而跳到 0x13b 处是打印出错信息，最终的结果是 hlt

  打印出错信息，稍后再看，下一步先看一看 MBR 是如何读磁盘。

**（2）读磁盘的方式**

  在 win7 的 MBR 中使用 int 0x13 扩展功能来读取磁盘，代码接着走到：

|  |
| --- |
| **代码接着：**  0000002D  83C510            add bp,byte +0x10 00000030  E2F1              loop 0x23 00000032  CD18              int 0x18  00000034  885600            mov [bp+0x0],dl          /\* 首次的 dl 值是 int 0x19 中断遗留下的，dl = 0x80 \*/  00000037  55                push bp  /\* 如果不成功的反复读 5 次 \*/ 00000038  C6461105          mov byte [bp+0x11],0x5   /\* DPT1[0] <--- 0x05 \*/ 0000003C  C6461000          mov byte [bp+0x10],0x0   /\* DPT0[16] <--- 0x00，此处是一个标志位，用来记录是否支持 int 0x13 扩展 \*/  /\* 测试是否支持 int 0x13 扩展功能 \*/ 00000040  B441              mov ah,0x41 00000042  BBAA55            mov bx,0x55aa 00000045  CD13              int 0x13                /\* 测试 int 0x13 扩展 \*/  00000047  5D                pop bp  00000048  720F              jc 0x59                /\* not support \*/ 0000004A  81FB55AA          cmp bx,0xaa55 0000004E  7509              jnz 0x59               /\* not support \*/ 00000050  F7C10100          test cx,0x1 00000054  7403              jz 0x59                /\* extend disk function \*/  00000056  FE4610            inc byte [bp+0x10]    /\* flags \*/  00000059  6660              pushad 0000005B  807E1000          cmp byte [bp+0x10],0x0      /\* not support \*/ 0000005F  7426              jz 0x87 |

★ 从 0x34 处是开始读磁盘的工作。

★ MBR 使用 [bp + 0x11] 和 [bp + 0x10] 两处内存作为临时变量。

  [bp + 0x11] 实际上是 DPT2 的 boot indicator 处，而 [bp + 0x10] 则是 DPT1 的最后一个 byte

★ 代码接下来测试是否支持 int 0x13 的扩展功能

★ 如果支持，则：置标志位为 1

★ 如果不支持，则跳到 0x87 处，而 0x87 是使用 int 0x13 原有功能来读取磁盘。（非扩展）

**（3）使用 int 0x13 扩展来读取磁盘**

|  |
| --- |
| **代码接着：**  /\* 以下是使用 int 0x13 扩展功能读 disk \*/  00000061  666800000000      push dword 0x0 00000067  66FF7608          push dword [bp+0x8]   /\* start sectors for read \*/ 0000006B  680000            push word 0x0 0000006E  68007C            push word 0x7c00   /\* buffer address \*/ 00000071  680100            push word 0x1 00000074  681000            push word 0x10  00000077  B442              mov ah,0x42 00000079  8A5600            mov dl,[bp+0x0]      /\* hard disk \*/ 0000007C  8BF4              mov si,sp            /\* buffer\_packet 的 address \*/ 0000007E  CD13              int 0x13             /\* 从disk 0x100000 读入\*/  00000080  9F                lahf  00000081  83C410            add sp,byte +0x10    /\* skip buffer \*/ 00000084  9E                sahf 00000085  EB14              jmp short 0x9b |

代码使用 int 0x13 的 0x42 号功能读扇区。

红色标注部分：这部分实际上是一个局部分变量，这个变量是个 struct（结构体），其定义用 c 描述如下：

|  |
| --- |
| **下面是用 c 描述为：**  struct buffer\_packet {     short buffer\_packet\_size;         /\* struct's  size \*/     short sectors;                           /\* 读多少个 sectors \*/     char \*buffer;                            /\* buffer address（segment:offset 形式）\*/     long long start\_sectors;          /\* 从哪个 sector 开始读 \*/ } buffer; |

这个 struct 的大小为 16 bytes 即：0x10 bytes，对就于指令 push word 0x10

它将从 [bp + 0x8] 处得出从哪个 sector 开始读，这个 [bp + 0x8] 从前面分析得知：它是 DPT1 的“分区前面的扇区数”

实际上它是标明分区1从哪个逻辑扇区开始。从前面可知：这个值就是 2048，因此，代码是从扇区 2048 开始读，

这个 2048 就是分区1 的开始扇区，它是个逻辑扇区概念，而非（cylinder/head/sector）表示法。

★ 代码中将从 2048 号扇区开始读 1 个扇区到 0x7c00 处，最后，代码将跳到 0x9b 处

更详细的 int 0x13 / ah = 0x42 用法，见：[int 0x13 扩展读（ax = 0x42）](http://www.mouseos.com/win7/int_0x13.html#42)

**（4）使用 int 0x13 原有功能来读取磁盘**

|  |
| --- |
| **代码接着：**  /\* read sector into memory \*/ 00000087  B80102            mov ax,0x201          /\* al =1 --> 读 1 个 sector \*/ 0000008A  BB007C            mov bx,0x7c00         /\* es:bx = buffer \*/ 0000008D  8A5600            mov dl,[bp+0x0]       /\* hard disk \*/ 00000090  8A7601            mov dh,[bp+0x1]       /\* 从哪个 head 开始读 \*/ 00000093  8A4E02            mov cl,[bp+0x2]       /\* 从哪个 sector 开始读 \*/                                                   /\* 其中低 6 bits 是 sector,高2位是cylinder \*./ 00000096  8A6E03            mov ch,[bp+0x3]       /\* 从哪个 cylinder 开始读 \*/ 00000099  CD13              int 0x13 |

如果不支持 int 0x13 扩展则使用原有功能读磁盘。

原有的读磁盘模式是：指定 head、cylinder 以及 sector

|  |
| --- |
| **使用 int 0x13 原有功能的缺点：**  由于使用 cylinder/head/sector 的模式：     cylinder 最大只支持 0x3ff，head 最大只支持 0xff，sector 最大只支持 0x3f 那么：     最大的访问容量为：(0x3ff \* 16065 + 0xff \* 63 + 0x3f) \* 512 ≈ 8g，超过 8g 的磁盘将无法访问。 |

因此，有必要引入扩展的 int 0x13 功能

|  |
| --- |
| **int 0x13 扩展功能的优点：**     显而易见：int 0x13 扩展功能可访问的容量远远超过原有功能，可以说访问多大都行，     它使用了 struct 的访问模式：提供 64 位的起始扇区值，16 位的一次访问扇区数。     理论上，可以说是非常惊人的访问容量。 |

**（5）读后续处理**

|  |
| --- |
| **代码接着：**  /\*\*\*\* 读完后续处理 \*/ 0000009B  6661              popad 0000009D  731C              jnc 0xbb           /\* succssed jump \*/  /\* 不成功的话，反得读 5 次 \*/ 0000009F  FE4E11            dec byte [bp+0x11] /\* 进行 5 次读 disk \*/ 000000A2  750C              jnz 0xb0  /\* 读完 5 次后，仍旧不成功，打印出错信息，htl \*/ 000000A4  807E0080          cmp byte [bp+0x0],0x80 000000A8  0F848A00          jz word 0x136 000000AC  B280              mov dl,0x80 000000AE  EB84              jmp short 0x34  /\* 读 sector 不成功时，disk controller 复位再重新读 \*/ 000000B0  55                push bp 000000B1  32E4              xor ah,ah 000000B3  8A5600            mov dl,[bp+0x0] 000000B6  CD13              int 0x13 000000B8  5D                pop bp 000000B9  EB9E              jmp short 0x59 |

这段代码功能是处理 int 0x13 后续工作，如果 int 0x13 读失败，则将 disk controller 复位，再重新读，反复读 5 次，

仍旧不成功的，将打印出错信息后 hlt

**（6）成功读取后**

|  |
| --- |
| **代码接着：**  /\* 读 sector 成功后 \*/ 000000BB  813EFE7D55AA      cmp word [0x7dfe],0xaa55 000000C1  756E              jnz 0x131         /\* 不是，打印错信息，hlt \*/  000000C3  FF7600            push word [bp+0x0] 000000C6  E88D00            call word 0x156    /\* 从 64H 读 1 byte \*/  000000C9  7517              jnz 0xe2          /\* 缓冲区满，不能写\*/  000000CB  FA                cli 000000CC  B0D1              mov al,0xd1       /\* 准备写数据到 0x60 端口中\*/ 000000CE  E664              out 0x64,al  000000D0  E88300            call word 0x156 000000D3  B0DF              mov al,0xdf 000000D5  E660              out 0x60,al 000000D7  E87C00            call word 0x156 000000DA  B0FF              mov al,0xff 000000DC  E664              out 0x64,al 000000DE  E87500            call word 0x156 000000E1  FB                sti  000000E2  B800BB            mov ax,0xbb00 000000E5  CD1A              int 0x1a  000000E7  6623C0            and eax,eax 000000EA  753B              jnz 0x127 |

代码中再次核对，是否 55aa 标志，如果不是将打印出错信息后 hlt

代码中还将测试键盘读/写缓冲区，然后往键盘端口 0x60 写命令字。

call word 0x156 这个子过程就是测试读/写缓冲区过程。

代码最终将转去 0x127 执行

**（7）代码最终会返回到 0x7c00 继续执行**

|  |
| --- |
| **代码接着：**  000000FB  666807BB0000      push dword 0xbb07 00000101  666800020000      push dword 0x200 00000107  666808000000      push dword 0x8 0000010D  6653              push ebx 0000010F  6653              push ebx 00000111  6655              push ebp 00000113  666800000000      push dword 0x0 00000119  6668007C0000      push dword 0x7c00 0000011F  6661              popad 00000121  680000            push word 0x0 00000124  07                pop es 00000125  CD1A              int 0x1a  /\* 最终代码会重新跳会到 0x7c00 处执行 \*/ 00000127  5A                pop dx 00000128  32F6              xor dh,dh 0000012A  EA007C0000        jmp word 0x0:0x7c00 0000012F  CD18              int 0x18 |

在 0x127 处的代码是 MBR 代码的最终出口。

代码显示：MBR 最终的出口是：重新跳会到 0x7c00 继续执行，而此时，0x7c00 处已经是 分区1 的代码。（即：2048 扇区）

很抱歉，我暂时不了解 int 0x1a 第 0xbb00 的作用。如果您了解 int 0x1a 中断第 0xbb00 作用，请您留言。:)

**3、 MBR 将分区1 的 1 个扇区读入 0x7c00 后，跳到 0x7c00 处继续执行。**

下面看看 MBR 剩余的错误处理模块：

|  |
| --- |
| **代码接着：**  /\* 打印出错信息，hlt \*/ 00000131  A0B707            mov al,[0x7b7]            /\* 索引号为 9A \*/ 00000134  EB08              jmp short 0x13e 00000136  A0B607            mov al,[0x7b6]            /\* 索引号为 7B \*/ 00000139  EB03              jmp short 0x13e 0000013B  A0B507            mov al,[0x7b5]            /\* 索引号为 63 \*/  0000013E  32E4              xor ah,ah 00000140  050007            add ax,0x700 00000143  8BF0              mov si,ax 00000145  AC                lodsb 00000146  3C00              cmp al,0x0 00000148  7409              jz 0x153 0000014A  BB0700            mov bx,0x7 0000014D  B40E              mov ah,0xe 0000014F  CD10              int 0x10 00000151  EBF2              jmp short 0x145 00000153  F4                hlt  00000154  EBFD              jmp short 0x153 |

这里需要使用 MBR 的字符串信息：

|  |
| --- |
| **代码接着：**  /\*\*\*\*\*\*\*\*\*\* 下面的区域是 mbr 常量符： mbr\_str　\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ /\* char \*mbr\_str = "Invalid partition table." "Error loading operation system." "Missing operation system." \*/  00000163  496E7661 6C696420 70617274 6974696F  00000173  6E207461 626C6500 4572726F 72206C6F 00000183  6164696E 67206F70 65726174 696E6720 00000193  73797374 656D004D 69737369 6E67206F 000001A3  70657261 74696E67 20737973 74656D  /\* 下面是字符串索引值 \*/  000001B2  00 000001B3  00 000001B4  00 000001B5  63 000001B6  7B 000001B7  9A 000001B8  46 000001B9  69 000001BA  6C 000001BB  65 000001BC  20 000001BD  73 |

在出错处理过程里，根据索引值找到相应的出错信息，然后使用 int 0x10 打印出错信息，最后是 hlt 停机指令。

**4、最后的 1BE ~ 1FD 是磁盘分区表。**

**四、总结 MBR 的作用**

  MBR 的作用主要为：检测分区的合法性， 然后，读活动分区（分区1）的引导记录到 0x7c00，重新返回到 0x7c00 执行。