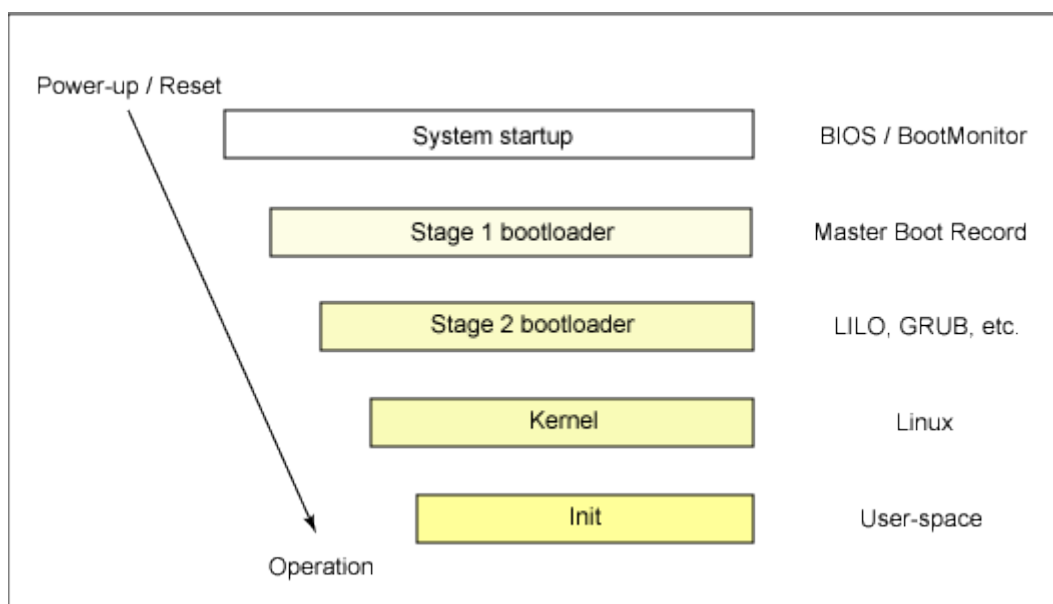


# GRUB 启动分析之 stage1

## 引言

玩 Linux 的人，肯定会听说过 Grub 这个神奇的东西，就是开机启动时候下拉一个菜单让我们选操作系统的那个东东。自己比较懒，一直没深入琢磨这个 Grub 的工作原理流程。最近工作遇到了 Grub 相关的问题，就花了一些时间学习了一下 Grub。

闲言少叙。我们首先看下 Linux 的启动过程流程图：



这个流程图是大牛 M. Tim Jones 在 Inside Linux boot process 中绘制的。清楚的 show 出了我们从摁开机键开始，计算机所做的事情。今天我要分享的是 stage1 bootloader 部分，以及 stage2 bootloader 的部分。

需知，刚刚进入 stage1 bootloader 的时候，操作系统还没有开始运行（废话，GRUB 他老人家就是召唤操作系统的），也没有文件系统的概念，认为直接运行操作系统可执行文件的，就可以洗洗睡了，GRUB 他老人家面临的是一穷二白的局面啊。

下面我开始学习 GRUB。当然了，现在主流的桌面都已经使用 GRUB 2, GRUB 0.9x 系统的目前统称为 GRUB legacy，只修复 bug，不再开发新的 feature 了。但是很多公司的服务器上跑的还是 GRUB。比如我们产品用的还是 GRUB。下面我做的实验都是在基于 2.6.24 内核的操作系统上做的，和家用的 Ubuntu 操作系统不同。

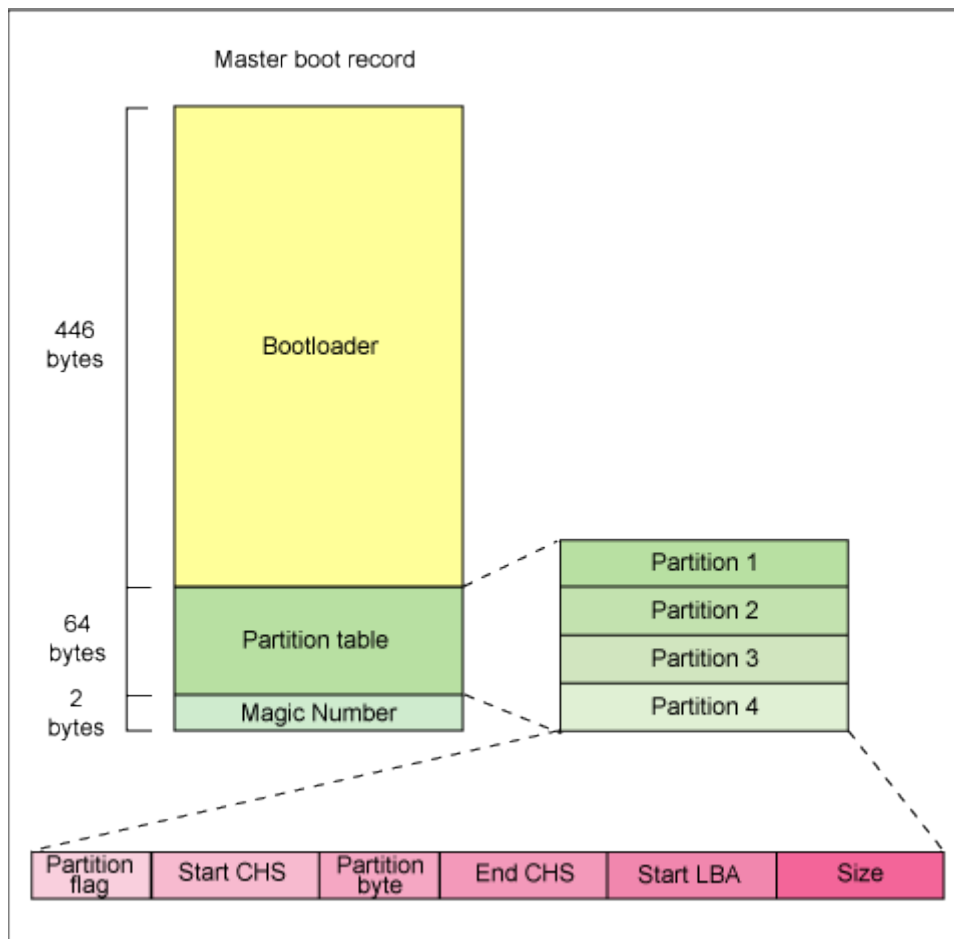
# MBR 和 stage1

MBR 是 master boot record 的缩写，也就是主引导记录。熟悉我博文的知道，我去年写过一篇分析 MBR 里面分区信息的文章。我们跳到/boot/grub/grub 目录下，可以看到有个文件叫 stage1，当然了还有 stage2,稍安勿躁，我们会慢慢提到。我们下载一份 GRUB 0.95 或者 GRUB 0.97 的代码，我们可以看到在 stage1 目录下有个 stage1.S 的汇编文件。他们之间是什么关系。

先说 MBR。MBR 是磁盘的 0 柱面，0 磁道，1 扇区（扇区从 1 开始计数），既然是一个扇区，大小就确定了，就是 512 个字节。MBR 严格的说有三部分组成

Structure of a classical generic MBR				
Address		Description		Size in bytes
Hex	Dec			
+000h	+0	Bootstrap code area		446
+1BEh	+446	Partition entry #1	Partition table (for primary partitions)	16
+1CEh	+462	Partition entry #2		16
+1DEh	+478	Partition entry #3		16
+1EEh	+494	Partition entry #4		16
+1FEh	+510	55h	Boot signature <sup>[nb 1]</sup>	2
+1FFh	+511	AAh		
Total size: 446 + 4*16 + 2				512

- 1. [0,0x1be)前 446 字节是 Bootstrap code area，是一段程序
- 2. ·[0x1be,0x1fe),64 个字节，是分区表信息。我前面的博文有重点分析。
- 3. [0x1fe,0x1ff] ,签名信息，是这两个字节是 55AA。



细心的筒子可以发现，/boot/grub/stage1 文件的大小也是 512 字节一个扇区那么大。我们可以比较下 MBR 和 /boot/grub/stage1 文件的内容。获取 MBR 方法比较简单，dd 就可以了，如下：

```
dd if=/dev/sda of=mbr_0_512 bs=512 count=1
```

这样，我们就获得了磁盘的 0 柱面，0 磁道 1 扇区的内容，即 MBR，存放在了 mbr\_0\_512 文件中：看一下文件的内容：

```

root@manu:~/code/c/classical/grub/grub_test# hexdump -C mbr_0_512
00000000 eb 48 90 00 00 00 00 00 00 00 00 00 00 00 00 |.H.....|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000030 00 00 00 00 00 00 00 00 00 00 00 00 03 02 |.....|
00000040 ff 00 00 20 01 00 00 00 00 02 fa 90 90 f6 c2 80 |... ..|
00000050 75 02 b2 80 ea 59 7c 00 00 31 c0 8e d8 8e d0 bc |u...Y|..1....|
00000060 00 20 fb a0 40 7c 3c ff 74 02 88 c2 52 be 7f 7d |. ..@|<.t...R..}|
00000070 e8 34 01 f6 c2 80 74 54 b4 41 bb aa 55 cd 13 5a |.4....tT.A..U..Z|
00000080 52 72 49 81 fb 55 aa 75 43 a0 41 7c 84 c0 75 05 |RrI..U.uC.A|..u.|
00000090 83 e1 01 74 37 66 8b 4c 10 be 05 7c c6 44 ff 01 |...t7f.L...|.D..|
000000a0 66 8b 1e 44 7c c7 04 10 00 c7 44 02 01 00 66 89 |f..D|....D...f.|
000000b0 5c 08 c7 44 06 00 70 66 31 c0 89 44 04 66 89 44 |\\..D..pf1..D.f.D|
000000c0 0c b4 42 cd 13 72 05 bb 00 70 eb 7d b4 08 cd 13 |..B..r...p..}....|
000000d0 73 0a f6 c2 80 0f 84 ea 00 e9 8d 00 be 05 7c c6 |s.....|. |
000000e0 44 ff 00 66 31 c0 88 f0 40 66 89 44 04 31 d2 88 |D..f1...@f.D.1..|
000000f0 ca c1 e2 02 88 e8 88 f4 40 89 44 08 31 c0 88 d0 |.....@.D.1...|
00000100 c0 e8 02 66 89 04 66 a1 44 7c 66 31 d2 66 f7 34 |...f..f.D|f1.f.4|
00000110 88 54 0a 66 31 d2 66 f7 74 04 88 54 0b 89 44 0c |.T.f1.f.t..T..D.|
00000120 3b 44 08 7d 3c 8a 54 0d c0 e2 06 8a 4c 0a fe c1 |;D.}<.T....L...|
00000130 08 d1 8a 6c 0c 5a 8a 74 0b bb 00 70 8e c3 31 db |...l.Z.t...p..1.|
00000140 b8 01 02 cd 13 72 2a 8c c3 8e 06 48 7c 60 1e b9 |.....r*....H|`..|
00000150 00 01 8e db 31 f6 31 ff fc f3 a5 1f 61 ff 26 42 |...1.1....a.8B|
00000160 7c be 85 7d e8 40 00 eb 0e be 8a 7d e8 38 00 eb ||..}.@.....}.8..|
00000170 06 be 94 7d e8 30 00 be 99 7d e8 2a 00 eb fe 47 |...}.0...}.*...G|
00000180 52 55 42 20 00 47 65 6f 6d 00 48 61 72 64 20 44 |RUB .Geom.Hard D|
00000190 69 73 6b 00 52 65 61 64 00 20 45 72 72 6f 72 00 |isk.Read. Error.|
000001a0 bb 01 00 b4 0e cd 10 ac 3c 00 75 f4 c3 00 00 00 |.....<.u.....|
000001b0 00 00 00 00 00 00 00 00 02 53 fc db 00 00 80 20 |.....S.....|

```

在看下/boot/grub/目录下的是 stage1 文件。（不要被图片误导，我只是将 stage1 文件拷贝到了我的工作目录）

```

root@manu:~/code/c/classical/grub/grub_test# hexdump -C stage1
00000000 eb 48 90 00 00 00 00 00 00 00 00 00 00 00 00 |.H.....|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000030 00 00 00 00 00 00 00 00 00 00 00 00 03 02 |.....|
00000040 ff 00 00 80 01 00 00 00 00 08 fa eb 07 f6 c2 80 |.....|
00000050 75 02 b2 80 ea 59 7c 00 00 31 c0 8e d8 8e d0 bc |u...Y|..1....|
00000060 00 20 fb a0 40 7c 3c ff 74 02 88 c2 52 be 7f 7d |. ..@|<.t...R..}|
00000070 e8 34 01 f6 c2 80 74 54 b4 41 bb aa 55 cd 13 5a |.4....tT.A..U..Z|
00000080 52 72 49 81 fb 55 aa 75 43 a0 41 7c 84 c0 75 05 |RrI..U.uC.A|..u.|
00000090 83 e1 01 74 37 66 8b 4c 10 be 05 7c c6 44 ff 01 |...t7f.L...|.D..|
000000a0 66 8b 1e 44 7c c7 04 10 00 c7 44 02 01 00 66 89 |f..D|....D...f.|
000000b0 5c 08 c7 44 06 00 70 66 31 c0 89 44 04 66 89 44 |\\..D..pf1..D.f.D|
000000c0 0c b4 42 cd 13 72 05 bb 00 70 eb 7d b4 08 cd 13 |..B..r...p.}....|
000000d0 73 0a f6 c2 80 0f 84 ea 00 e9 8d 00 be 05 7c c6 |s.....|.|.|
000000e0 44 ff 00 66 31 c0 88 f0 40 66 89 44 04 31 d2 88 |D..f1...@f.D.1..|
000000f0 ca c1 e2 02 88 e8 88 f4 40 89 44 08 31 c0 88 d0 |.....@.D.1...|
00000100 c0 e8 02 66 89 04 66 a1 44 7c 66 31 d2 66 f7 34 |...f..f.D|f1.f.4|
00000110 88 54 0a 66 31 d2 66 f7 74 04 88 54 0b 89 44 0c |.T.f1.f.t..T..D.|
00000120 3b 44 08 7d 3c 8a 54 0d c0 e2 06 8a 4c 0a fe c1 |;D.}<.T....L...|
00000130 08 d1 8a 6c 0c 5a 8a 74 0b bb 00 70 8e c3 31 db |...l.Z.t...p..1.|
00000140 b8 01 02 cd 13 72 2a 8c c3 8e 06 48 7c 60 1e b9 |....r*....H|`..|
00000150 00 01 8e db 31 f6 31 ff fc f3 a5 1f 61 ff 26 42 |....1.1.....a.8B|
00000160 7c be 85 7d e8 40 00 eb 0e be 8a 7d e8 38 00 eb ||..}.@.....}.8..|
00000170 06 be 94 7d e8 30 00 be 99 7d e8 2a 00 eb fe 47 |...}.0...}.*...G|
00000180 52 55 42 20 00 47 65 6f 6d 00 48 61 72 64 20 44 |RUB .Geom.Hard D|
00000190 69 73 6b 00 52 65 61 64 00 20 45 72 72 6f 72 00 |isk.Read. Error.|
000001a0 bb 01 00 b4 0e cd 10 ac 3c 00 75 f4 c3 00 00 00 |.....<.u.....|
000001b0 00 00 00 00 00 00 00 00 00 00 00 00 24 12 |.....$.|

```

可以看到这两个文件大部分是相同的，其实这部分 code 部分是一样的，至于不一样的地方是

1. [0x1b8, 0x1bb)这个部分叫做 optional disk signature，Windows 系的产品会用到这 4 个字节，对于 Linux 和 grub 是用不到这 4 个字节的。

Address		Description	Size in bytes
Hex	Dec		
000	0	Code	440
1b8	440	Optional disk signature	4
1bc	444	0x0000	2
1be	446	Four 16-byte entries for primary partitions	64
1fe	510	0xaa55	2

2 [0x40 ,0x50]中有部分不同，我暂时不懂

结论是/boot/grub/stage1 文件和主引导记录 MBR 的 code 部分是相同的。事实上这份代码是从 grub 源

代码的 stage1/stage1.S 汇编出来的。stage1.S 是 grub 的第一个文件，便以后编译后产生的代码，正好是 512 字节，不是正好，是必须，否则无法放入 1 个扇区。

这个 MBR 的信息是 grub 安装上去的，方法如下：

```
# 4. 將主程式安裝上去吧！安裝到 MBR 看看！
grub> setup (hd0)
Checking if "/boot/grub/stage1" exists... no <==因為 /boot 是獨立的
Checking if "/grub/stage1" exists... yes <==所以這個檔名才是對的！
Checking if "/grub/stage2" exists... yes
Checking if "/grub/e2fs_stage1_5" exists... yes
Running "embed /grub/e2fs_stage1_5 (hd0)"... 15 sectors are embedded.
succeeded
Running "install /grub/stage1 (hd0) (hd0)1+15 p (hd0,0)/grub/stage2
/grub/grub.conf"... succeeded <==將 stage1 程式安裝妥當囉！
Done.
# 很好！確實有裝起來～這樣 grub 就在 MBR 當中了！
```

## stage1 源码分析

stage1 阶段的源代码就是 grub 源码中 stage/stage1.S，可惜他老人家是爱 at&t 风格的汇编，折磨的我七荤八素，死去活来，看了网上一些前辈的文章，总算有了一些心得体会。还是我常说的那句话，光荣属于前辈！

故事从哪里讲起呢，还是从我们按下电源开关开始讲起。呵呵。

当我们按下开机键，进入系统启动阶段，什么 BIOS，POST（Power-On Self Test 加电自检），反正是一陀名词一陀事，这些我们统统不管，我们就从系统 BIOS 做的最后一件事开始讲起，BIOS 最后一件事：根据用户指定的启动顺序从软盘、硬盘或光驱启动 MBR。在这个过程中会按照启动顺序顺序比较其放置 MBR 的位置的结尾两位是否为 0xAA55，通过这种方式判断从哪个引导设备进行引导。在确定之后，将该引导设备的 MBR 内容读入到 0x7C00 的位置，并再次判断其最后两位，当检测正确之后，进行阶段 1 的引导，从此进入第二阶段 stage1 bootloader 阶段。

简单地说，就是 BIOS 执行 INT 0x19，加载 MBR 内容至 0x7c00，然后跳转执行

且慢，为啥是 0x7c00 位置呢？我边访名医，终于找到了一篇相关的博文《为嘛 BIOS 将 MBR 读入 0x7C00 地址处（x86 平台下）》，这兄弟对系统启动也颇有兴趣，有好多博文写的很优秀，我跟他学了很多。英文好的筒子可以直接看 Why BIOS loads MBR into 0x7C00 in x86？

简单的说，0x7c00=32KB-1024B，是 32K 的最后一个 KB，这个 magic number 不是 intel 决定的，所以我们在 X86 相关的文档中无法找到这个 magic number 的说明，这个 magic number 属于 BIOS

specification。这个 0x7c00 是 IBM PC 5150 BIOS developer team 决定的。

BIOS developer team decided 0x7C00 because:

1. They wanted to leave as much room as possible for the OS to load itself within the 32KiB.
2. 8086/8088 used 0x0 - 0x3FF for interrupts vector, and BIOS data area was after it.
3. The boot sector was 512 bytes, and stack/data area for boot program needed more 512 bytes. So, 0x7C00, the last 1024B of 32KiB was chosen.

```
+----- 0x0
| Interrupts vectors
+----- 0x400
| BIOS data area
+----- 0x5??
| OS load area
+----- 0x7C00
| Boot sector
+----- 0x7E00
| Boot data/stack
+----- 0x7FFF
| (not used)
+----- (...)
```

跑了半天题，我们继续。我们把 MBR 的 code 加载到了 0x7c00,开始执行 MBR 处的代码，下面重点分析 MBR 处的代码，即 grub 源码中的 stage1/stage1.S

```
jmp after_BPB
nop /* do I care about this ??? */

. = _start + 4
```

一开始是个跳转指令，直接跳转到 after\_BPB，后面的 NOP 就执行不到了。ater\_BPB 在后面有定义：对于 mbr 二进制文件而言：

```
root@manu:~/code/c/classical/grub/grub_test# hexdump -C stage1
00000000 eb 48 90 00 00 00 00 00 00 00 00 00 00 00 00 00 |.H.....|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
```

头两个字节 0xeb48,eb 是 JMP 指令，第三个字节是 0x90，这个字节是 NOP 指令。

after\_BPB:

```

/* general setup */
cli      /* we're not safe here! */

/*
 * This is a workaround for buggy BIOSes which don't pass boot
 * drive correctly. If GRUB is installed into a HDD, do
 * "orb $0x80, %dl", otherwise "orb $0x00, %dl" (i.e. nop).
 */
.byte    0x80, 0xca

```

首先是 cli 指令，禁用中断然后是显示 80ca 这个二进制码。看下注释，这个 80ca 的含义是 orb \$0x80,%dl.意思是给 dl 寄存器的赋值 80。

```

DL = 00h 1st floppy disk ( "drive A:" )
DL = 01h 2nd floppy disk ( "drive B:" )
DL = 80h 1st hard disk
DL = 81h 2nd hard disk

```

因为我们是磁盘加载的 MBR，所以我们 dl 里面存的是 0x80。接下来分析：

```

ljmp     $0, $ABS(real_start)

real_start:

/* set up %ds and %ss as offset from 0 */
xorw     %ax, %ax
movw     %ax, %ds
movw     %ax, %ss

/* set up the REAL stack */
movw     $STAGE1_STACKSEG, %sp

sti      /* we're safe again */
MOV_MEM_TO_AL(ABS(boot_drive)) /* movb ABS(boot_drive), %al */
cmpb     $GRUB_INVALID_DRIVE, %al
je       1f
movb     %al, %dl

```

进入 real\_start 了，ax 清零，ds 赋值 0，ss 赋值 0，将 STAGE1\_STACKSEG (0×2000) 赋值给 sp，这样就设置了实模式下的堆栈段地址（栈顶位置）ss:sp = 0×0000:0×2000。接着置中断允许位。然后将 dl 寄存器中的值拷贝到 al 寄存器，然后将 al 寄存器的值和 0xFF 比较，对于我们的场景来说，我们是 al 里面存的是 0x80,所以，不等于 0xFF，不用跳转，继续执行将 al 的 0x80 拷贝到 dl 寄存器中。



```

/* save drive reference first thing! */
pushw    %dx

/* print a notification message on the screen */
MSG(notification_string)

/* do not probe LBA if the drive is a floppy */
testb    $STAGE1_BIOS_HD_FLAG, %dl
jz    chs_mode

/* check if LBA is supported */
movb     $0x41, %ah
movw     $0x55aa, %bx
int $0x13

```

notification\_string 是 GRUB，这一段截至到 MSG 是显示 GRUB 到屏幕上，因为这个 MSG 是细节，我们按下不表。总是作用是屏幕显示 GRUB。我们还记得，MBR 内容里面有如下信息：

```

00000170 06 be 94 7d e8 30 00 be 99 7d e8 2a 00 eb fe 47 |...}.0...}.*...G|
00000180 52 55 42 20 00 47 65 6f 6d 00 48 61 72 64 20 44 |RUB .Geom.Hard D|
00000190 69 73 6b 00 52 65 61 64 00 20 45 72 72 6f 72 00 |isk.Read. Error.|

```

testb 这部分是探测 drive 是否是硬盘，如果不是硬盘是软盘，直接采用 CHS\_MODE，就不用费事判断了。我们知道，80h 和 81h 是硬盘，所以探测对应 bit 位。如果我们的启动设备是硬盘，按么我们需要检测 LBA 是否支持。

通过 BIOS 调用 INT 0x13 来确定是否支持扩展，LBA 扩展功能分两个子集，如下：

第一个子集提供了访问大硬盘所必须的功能，包括：

```

*****
1.检查扩展是否存在：ah = 41h，bx = 0x55aa，dl = drive( 0x80 ~ 0xff )
2.扩展读：ah = 42h
3.扩展写：ah = 43h
4.校验扇区：ah = 44h
5.扩展定位：ah = 47h
6.取得驱动器参数：ah = 48h
*****

```

第二个子集提供了对软件控制驱动器锁定和弹出的支持，包括：

```

*****
1.检查扩展：ah = 41h
2.锁定/解锁驱动器：ah = 45h
3.弹出驱动器：ah = 46h

```

4.取得驱动器参数 : ah = 48h

5.取得扩展驱动器改变状态: ah = 49h

\*\*\*\*\*

我们采用的是 ah=41h, bx=0x55aa, dl=0x80,所以是检查扩展是否存在。这个操作会改变 CF 标志位的值。如果支持 LBA, 那么 CF=0, 否则 CF=1。

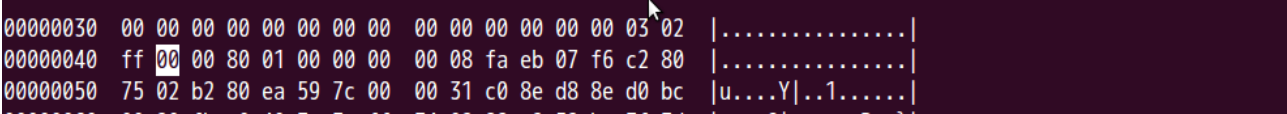
```
/* use CHS if fails */
jc chs_mode
cmpw    $0xaa55, %bx
jne chs_mode

/* check if AH=0x42 is supported if FORCE_LBA is zero */
MOV_MEM_TO_AL(ABS(force_lba)) /* movb ABS(force_lba), %al */
testb   %al, %al
jnz lba_mode
andw     $1, %cx
jz chs_mode
```

我们刚才 BIOS 用 INT 0x13 探查是否采用 LBA 模式。存在下面集中情况：

1. 启动设备不是 0x80, 0x81, 压根不探查, 直接采用 CHS 模式
2. 探查结果 CF=1, 二话不说, 跳转到 CHS 模式
3. CF=0 是否就采用 LBA 呢? 也不一定, 还需要判断 bx==0x55aa, bx==0x55aa, 采用 LBA 模式, 否则 CHS 模式

有一个 FORCE\_LBA Byte, 如果这个位是 1, 那么直接采用 LBA MODE, 这个位是哪个呢?



```
00000030  00 00 00 00 00 00 00 00 00 00 00 00 03 02 |.....|
00000040  ff 00 00 80 01 00 00 00 00 08 fa eb 07 f6 c2 80 |.....|
00000050  75 02 b2 80 ea 59 7c 00 00 31 c0 8e d8 8e d0 bc |u...Y|..1.....|
```

0x41 对应的 00, 表示 FORCE\_LBA 是 zero。

接下来, 就是花开两朵, 各表一枝, 一枝叫 CHS, 另一枝叫 LBA 模式。CHS 已经人老珠黄, 它是硬盘容量很小的那个时代留下的遗产。

C 表示 Cylinders

H 表示 Heads

S 表示 Sectors

其中：

磁头数(Heads)表示硬盘总共有几个磁头,也就是有几面盘片,最大数为 255 (用 8 个二进制位存储)。从 0 开始编号。

柱面数(Cylinders) 表示硬盘每一面盘片上有几条磁道,最大数为 1023(用 10 个二进制位存储)。从 0 开始编号。

扇区数(Sectors) 表示每一条磁道上有几个扇区,最大数为 63(用 6 个二进制位存储)。从 1 始编号。

而现在的硬盘远大于 8.414 GB (按照硬盘厂商常用的单位的计算) , CHS 寻址方式已不能满足要求。可到目前为止,人们常说的硬盘参数还是这古老的 CHS 参数。那么为什么还要使用这些参数?向下兼容。

既然 CHS 已经人老珠黄,我们也没必要在它身上浪费时间了(这话说的,怎么和陈世美这么像?!) 我们关注的重点是 LBA MODE.

```
lba_mode:
    /* save the total number of sectors */
    movl    0x10(%si), %ecx

    /* set %si to the disk address packet */
    movw    $ABS(disk_address_packet), %si

    /* set the mode to non-zero */
    movb    $1, -1(%si)

    movl    ABS(stage2_sector), %ebx

    /* the size and the reserved byte */
    movw    $0x0010, (%si)

    /* the blocks */
    movw    $1, 2(%si)

    /* the absolute address (low 32 bits) */
    movl    %ebx, 8(%si)

    /* the segment of buffer address */
    movw    $STAGE1_BUFFERSEG, 6(%si)

    xorl    %eax, %eax
    movw    %ax, 4(%si)
    movl    %eax, 12(%si)

/*
 * BIOS call "INT 0x13 Function 0x42" to read sectors from disk into
memory
 * Call with    %ah = 0x42
```

```

*          %dl = drive number
*          %ds:%si = segment:offset of disk address packet
*   Return:
*          %al = 0x0 on success; err code on failure
*/

    movb    $0x42, %ah
    int     $0x13

    /* LBA read is not supported, so fallback to CHS. */
    jc      chs_mode

    movw     $STAGE1_BUFFERSEG, %bx
    jmp      copy_buffer

```

然后将标号 disk\_address\_packet 处的地址赋给 si，紧接着将[si-1]内存处置 1（也就是 mode 被置 1，表示 LBA 扩展读；如果是 0，就是 CHS 寻址读）。

movl ABS(stage2\_sector), %ebx,把要加载或拷贝的扇区数传给 ebx 寄存器。

由 si 及其偏移量指向的内存保存着磁盘参数块，如下：

```

*****
偏移量 大小 位数 描述
00h BYTE 8 数据块的大小 (10h or 18h)
01h BYTE 8 保留，必须为 0
02h WORD 16 传输数据块数，传输完成后保存传输的块数
04h DWORD 32 传输时的数据缓存地址
08h QWORD 64 起始绝对扇区号（即起始扇区的 LBA 号码）
*****

```

或者如下图所示：

INT 13h AH=42h: Extended Read Sectors From Drive

[edit]

Parameters:

Registers	
AH	42h = function number for extended read
DL	drive index (e.g. 1st HDD = 80h)
DS:SI	segment:offset pointer to the DAP, see below

DAP : Disk Address Packet		
offset range	size	description
00h	1 byte	size of DAP = 16 = 10h
01h	1 byte	unused, should be zero
02h..03h	2 bytes	number of sectors to be read, (some Phoenix BIOSes are limited to a maximum of 127 sectors)
04h..07h	4 bytes	segment:offset pointer to the memory buffer to which sectors will be transferred (note that x86 is little-endian: if declaring the segment and offset separately, the offset must be declared before the segment)
08h..0Fh	8 bytes	absolute number of the start of the sectors to be read (1st sector of drive has number 0)

Results:

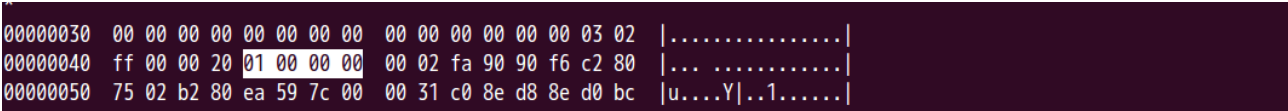
CF	Set On Error, Clear If No Error
AH	Return Code

```
movw    $0x0010, (%si)           执行的结果是 si[0] =10h, si[1]=00h
movw    $1, 2(%si)               执行的结果是 si[2] =01h, si[3]=00h
/* the segment of buffer address */
movw    $STAGE1_BUFFERSEG, 6(%si) 执行的记过是 si·[6]=00h si[7]=07h

movl    ABS(stage2_sector), %ebx
movl    %ebx, 8(%si)

stage2_sector:
    .long    1
```

这个 stage2\_sector 在二进制文件中的偏移量是 0x44



我们 si[8]存储的 long 类型是起始扇区的 LBA 号码，从 1 号扇区也就是 0 柱面，0 磁道，2 扇区。 ， si[2]记录着要传输多少个扇区，值为 1，只传输一个数据块，读取后，将扇区的内容存储到 si 偏移量为 04h 05h 6h、7h 确定的内存区域 0x7000:0x0000 上了。这是 int 13h（42）的作用。

最后一段代码是

```
copy_buffer:
movw    ABS(stage2_segment), %es

/*
 * We need to save %cx and %si because the startup code in
 * stage2 uses them without initializing them.
 */
pusha
pushw    %ds
```

```

movw    $0x100, %cx    //循环 0x100 次, 即 256 次
movw    %bx, %ds
xorw    %si, %si
xorw    %di, %di

cld

rep
movsw                    //每次拷贝 2 字节, 一个 word

popw    %ds
popa

/* boot stage2 */
jmp     *(stage2_address)

```

这段代码的含义是将刚才搬到 0x7000 : 000 的 512 字节, 再次搬到 0x8000 : 0000

OK, 我们很痛苦的跟踪了 stage1.S 的代码, 最后得到的结论是: stage1.S 这汇编出来的 512 个字节代码的作用是将 0 柱面, 0 磁道, 2 扇区的 512 字节 copy 到 0x8000 处。

很失望吧, 费了半天劲, 最后只得到这么一点点结论。人生就是如此, 付出不一定有回报, 对于我们而言, 只管努力, 莫问前程, 才能活得心平气和。

我们读到的 512 字节是干嘛的呢? 啥时候才能看到 GRUB 的选择 OS 的界面呢? 江湖传说的 stage2 到底是怎么回事, 江湖传说的 stage1.5 是怎么回事, 且听下回分解, 我是累了, 不写了。

参考文献:

1 [Stage1.s 源代码分析](#) (这篇文章非常棒, 很多内容都是受惠于这篇博文)

2 维基百科

3 GRUB 源代码分析 (很棒的一个文档)

4 [The mysteries around "0x7C00" in x86 architecture bios bootloader](#)

5 [Linux/Unix 系统的引导过程 \(从加电到操作系统运行\)](#)

作者: trend/bean\_li

博客地址: <http://bean.blog.chinaunix.net/>

Email: [manuscola@163.com](mailto:manuscola@163.com)