

第1天

知识盲点

- 对CPU而言，内存是外部存储装置，在CPU内核之中，存储装置只有寄存器，全部寄存器的容量加起来也不到1KB
- 本书的讲解顺序：CPU->汇编->C语言
- 有点难，最后找到了关光盘文件

准备工作

一开始我担心自己的电脑是64位的问题，可能和某些工具不兼容，于是又想法是在VM中要不要先安装一个 **WinXP** 的系统，后来，因为书是在图书馆借阅的，当我看书的时候，一些工具我才发现是在光盘上的，可是我借书的时候就没有发现光盘...一定是某同学弄丢了吧。偶然的情况在 **github** 上找到光盘的工具，并开始了正式的编写。

十六进制初始系统

所谓的“初始系统” 实际上是利用十六进制编辑器写的一个在系统运行时候输出 **hello world** 的“系统”。

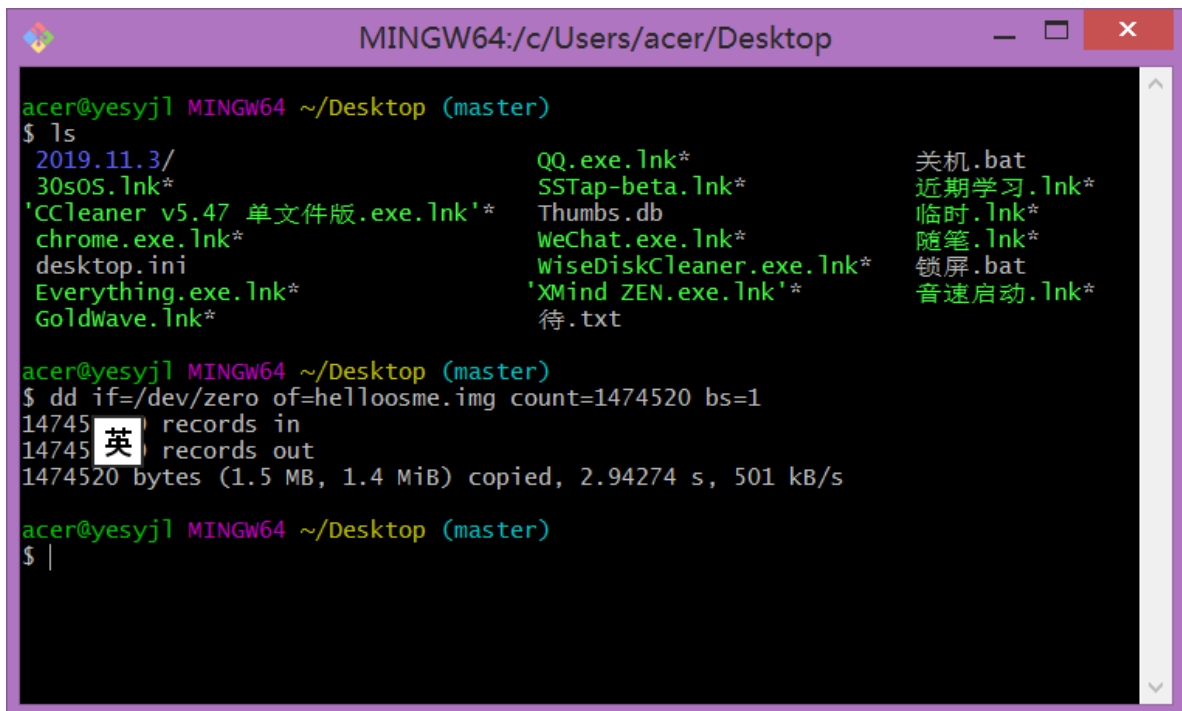
创建 **helloos.img** 镜像

如果照书上来的话，实在是太过于浪费时间，于是采用了一种相对来说简单的方式，因为自己电脑安装了 **git** 这样一来就可以用 **Git Bash** 如下图：



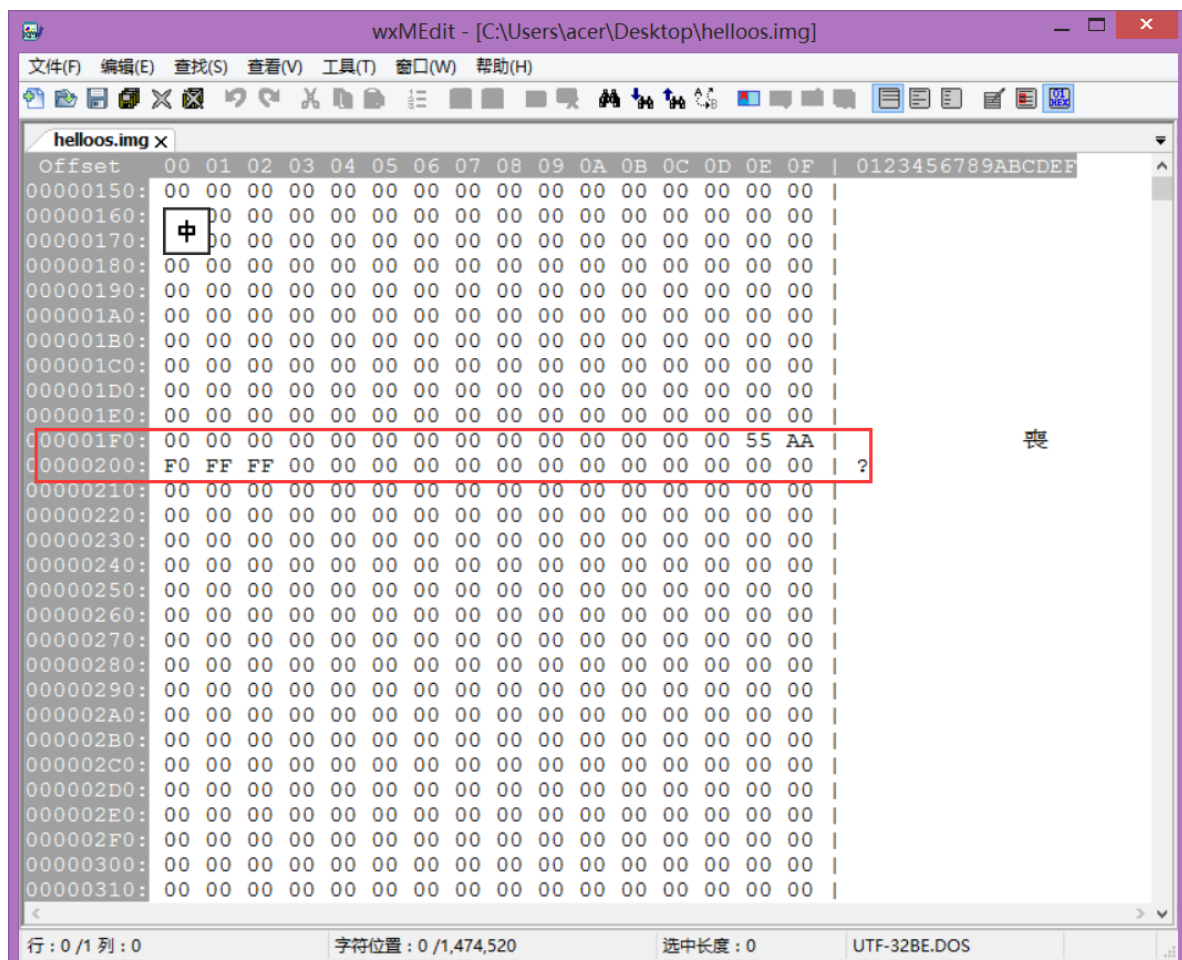
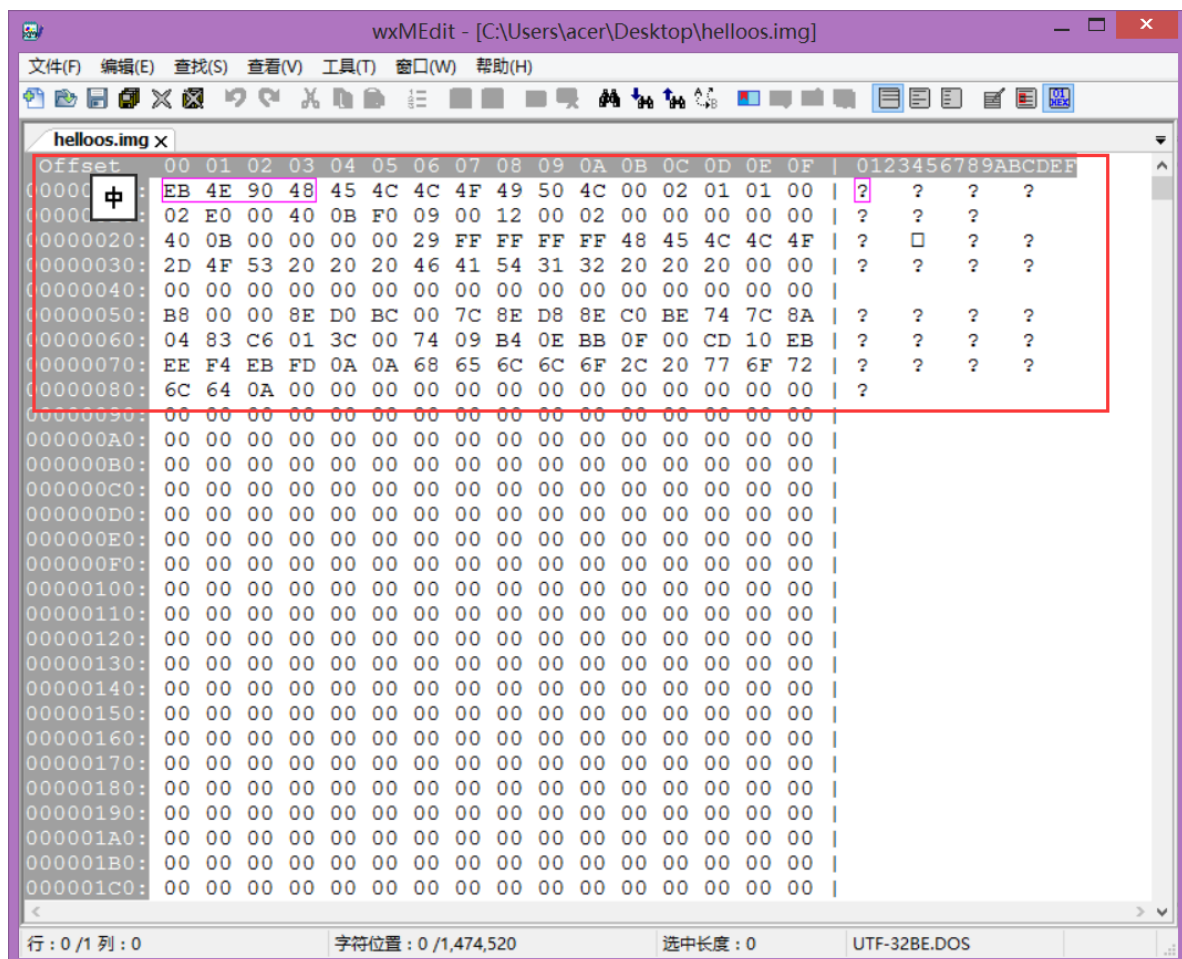
我们随便选在一个地方右键单击，我这里为了方便，选择了 **桌面** 。然后键入如下代码：
其中 **bs** 是单位

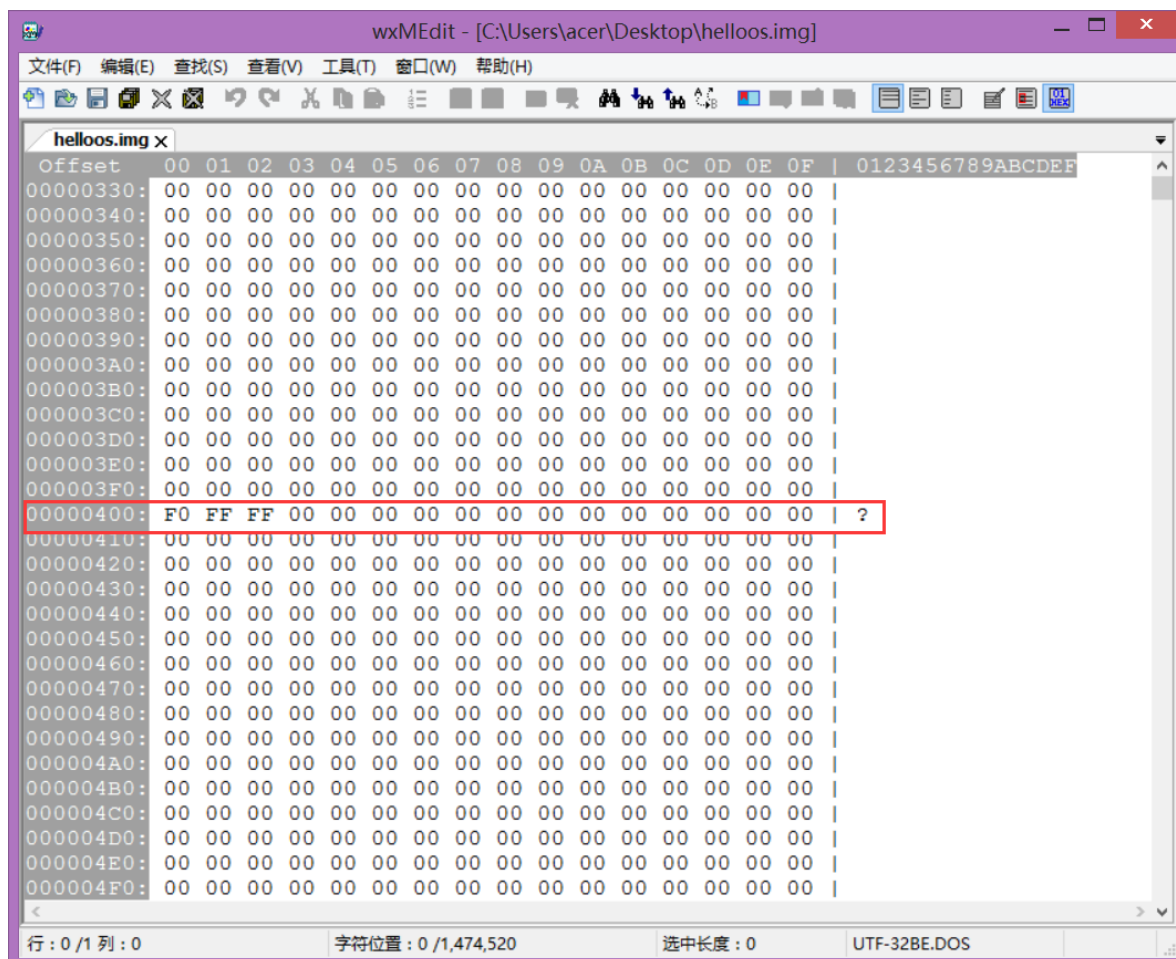
```
dd if=/dev/zero of=helloosme.img count=1474520 bs=1
```



```
MINGW64:/c/Users/acer/Desktop
acer@yesyjl MINGW64 ~/Desktop (master)
$ ls
2019.11.3/          QQ.exe.lnk*        关机.bat
30s0S.lnk*          SSTap-beta.lnk*    近期学习.lnk*
'CCleaner v5.47 单文件版.exe.lnk'*  Thumbs.db         临时.lnk*
chrome.exe.lnk*     WeChat.exe.lnk*    随笔.lnk*
desktop.ini         WiseDiskCleaner.exe.lnk*  锁屏.bat
Everything.exe.lnk* 'XMind ZEN.exe.lnk'*  音速启动.lnk*
GoldWave.lnk*       待.txt
acer@yesyjl MINGW64 ~/Desktop (master)
$ dd if=/dev/zero of=helloosme.img count=1474520 bs=1
14745 英 records in
14745  records out
1474520 bytes (1.5 MB, 1.4 MiB) copied, 2.94274 s, 501 kB/s
acer@yesyjl MINGW64 ~/Desktop (master)
$ |
```

至此，我们创建了一个内容全是 **0** 的镜像，因为 **helloos.img** 的内容只有很少的一部分内容不是0，这样一来我们就节约了很多时间，只需要改很小一部分就好了。这里我们选择一款编辑器 **wxMEdite** 这款十六进制编辑器进行编辑。接下来的内容就是照着书写十六进制数了，写完是这样的。





一共是需要改写 3 个地方。至此，我们的“初始系统”算是完成了。

项目的目录

整个项目的目录树

```
D:.\
|  GPL.txt
|  LGPL.txt
|  license.txt
|
|---helloos0
|    !cons_9x.bat
|    !cons_nt.bat
|    helloos.img
|    install.bat
|    run.bat
|
|---z_new_o
|    !cons_9x.bat
|    !cons_nt.bat
|    make.bat
|    Makefile
|
|---z_new_w
|    !cons_9x.bat
```

```
|      !cons_nt.bat
|      make.bat
|      Makefile
|
|_z_osabin
|      !cons_9x.bat
|      !cons_nt.bat
|      !run_opt.txt
|      run.bat
|
|_z_tools
|      aksa.exe
|      aska.exe
|      bim2bin.exe
|      bim2hrb.exe
|      bin2obj.exe
|      cc1.exe
|      comcom.exe
|      com_mak.txt
|      cpp0.exe
|      doscmd.exe
|      dsar.bat
|      edimg.exe
|      edimgopt.txt
|      esart5.bat
|      fdimg0at.tek
|      fdimg0tw.tek
|      gas2nask.exe
|      golib00.exe
|      imgtol.com
|      ld.exe
|      make.exe
|      makefont.exe
|      nask.exe
|      naskcnv0.exe
|      nothing.com
|      obj2bim.exe
|      osalink1.exe
|      OSASK0.PSF
|      sartol.exe
|      sjisconv.exe
|      t5lzma.exe
|      upx.exe
|      wce.exe
|
|_guigui00
|      errno.h
|      float.h
|      gg00libc.lib
```

- | gg00old0.rul
- | golibc.lib
- | guigui00.h
- | guigui00.rul
- | limits.h
- | math.h
- | setjmp.h
- | stdarg.h
- | stddef.h
- | stdio.h
- | stdlib.h
- | string.h

- |—haribote

- |
 - | errno.h
 - | float.h
 - | golibc.lib
 - | haribote.rul
 - | harilibc.lib
 - | limits.h
 - | math.h
 - | setjmp.h
 - | stdarg.h
 - | stddef.h
 - | stdio.h
 - | string.h

- |—osa_qemu

- |
 - | edimgopt.txt
 - | Makefile
 - | manual.bat
 - | OSAIMGAT.BIN
 - | osalink1.opt
 - | OSASK.EXE
 - | OSASK0.PSF
 - | timerdrv.tek

- |—qemu

- |
 - | bios.bin
 - | fdimage0.bin
 - | Makefile
 - | qemu-win.bat
 - | qemu.exe
 - | SDL.dll
 - | vgabios.bin

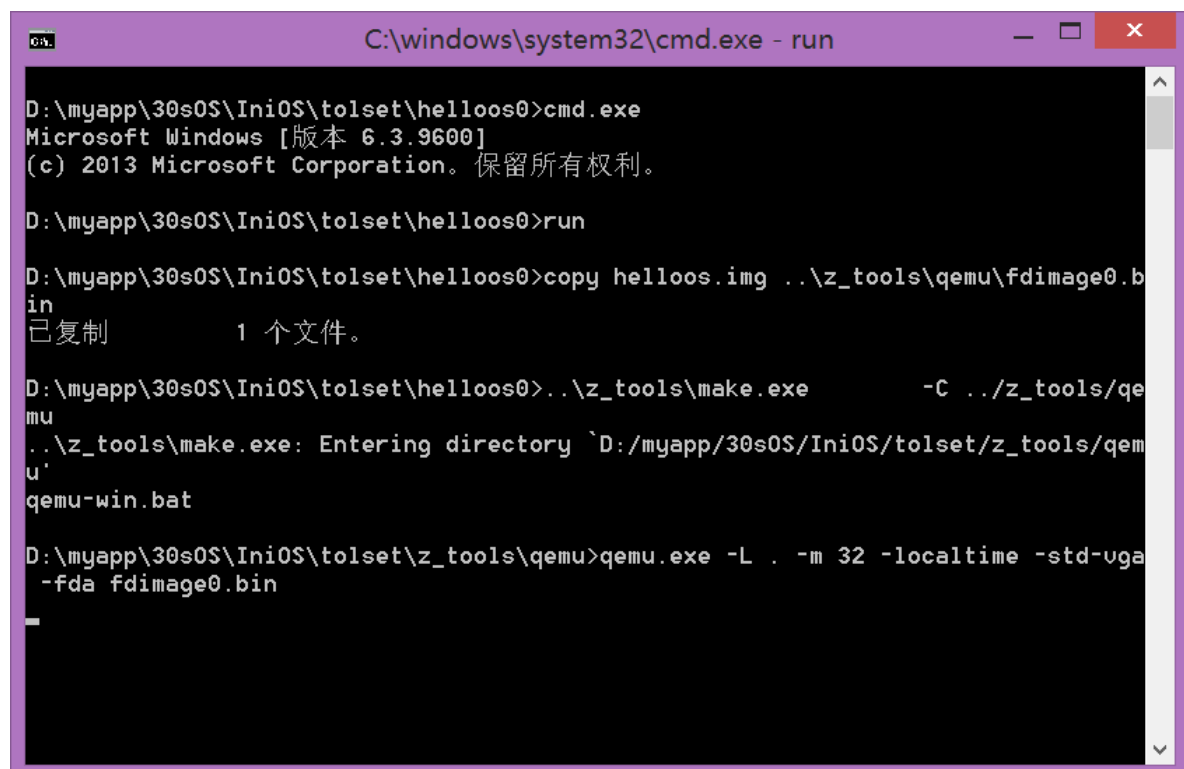
- |—qemu_9x

- |
 - | bios.bin
 - | Makefile

```
|      qemu.exe
|      SDL.dll
|      vgabios.bin
|
└─win32
    errno.h
    float.h
    glibc.lib
    libmingw.lib
    limits.h
    math.h
    setjmp.h
    stdarg.h
    stddef.h
    stdio.h
    stdlib.h
    string.h
    w32clibc.lib
```

系统运行

一切准备就绪，开始运行。点击 `!cons_9x.bat` 或 `!cons_nt.bat`，输入 `run` 命令，查看运行效果。



```
GA. C:\windows\system32\cmd.exe - run
D:\myapp\30s0S\Ini0S\tolset\helloos0>cmd.exe
Microsoft Windows [版本 6.3.9600]
(c) 2013 Microsoft Corporation。保留所有权利。

D:\myapp\30s0S\Ini0S\tolset\helloos0>run

D:\myapp\30s0S\Ini0S\tolset\helloos0>copy helloos.img ..\z_tools\qemu\fdimage0.b
in
已复制          1 个文件。

D:\myapp\30s0S\Ini0S\tolset\helloos0>..\z_tools\make.exe          -C ../z_tools/qe
mu
..\z_tools\make.exe: Entering directory `D:/myapp/30s0S/Ini0S/tolset/z_tools/qem
u'
qemu-win.bat

D:\myapp\30s0S\Ini0S\tolset\z_tools\qemu>qemu.exe -L . -m 32 -localtime -std-uga
-fda fdimage0.bin
```

```
QEMU
Bochs VGABIOS ver.0.2
This VGA/VBE BIOS is released under the KL-01

Bochs VBE Support enabled

Bochs BIOS, 1 cpu, $Revision: 1.110 $ $Date: 2004/05/31 13:11:27 $
ata1 master: QEMU CD-ROM ATAPI-4 CD-Rom/DVD-Rom
ata1 slave: Unknown device

Booting from Floppy...

hello, world
_
```

可以看到，我们的系统成功启动，并且输出“hello,world”，我把这个系统命名为 **Inios**。

汇编程序改版系统

汇编小知识

- **0x** 代表的是十六进制数
- **DB** 是指令 **data byte** 的缩写，也就是往文件里直接写入1个字节的指令。汇编指令不区分大小写。
- **RESB** 是指令 **reserve byte** 的缩写，可以简单的理解为是用来填充 **00**，也就是 **0x00**。比如说 **RESB 16** 就是把16个 **00** 替换了。即：两者等价。

弄明白这些，我们就可以简单的把我们写过的 **helloos.img** 换成用汇编语言进行编写。

汇编源代码

我们已经写好了 **helloos.img**，这时候对于这个文件，用汇编编写应该为：

```
DB 0xeb, 0x4e, 0x90, 0x48, 0x45, 0x4c, 0x4c, 0x4f
DB 0x49, 0x50, 0x4c, 0x00, 0x02, 0x01, 0x01, 0x00
DB 0x02, 0xe0, 0x00, 0x40, 0x0b, 0xf0, 0x09, 0x00
DB 0x12, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00
DB 0x40, 0x0b, 0x00, 0x00, 0x00, 0x00, 0x29, 0xff
(此处省略18万4314行，其实这里省略的大部分是0x00)
DB 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
```

此时，我们再利用 **RESB** 指令对其进行“压缩”，则改写之后的汇编程序为：

```
DB 0xeb, 0x4e, 0x90, 0x48, 0x45, 0x4c, 0x4c, 0x4f
DB 0x49, 0x50, 0x4c, 0x00, 0x02, 0x01, 0x01, 0x00
DB 0x02, 0xe0, 0x00, 0x40, 0x0b, 0xf0, 0x09, 0x00
```



```
DB  0x12, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00
DB  0x40, 0x0b, 0x00, 0x00, 0x00, 0x00, 0x29, 0xff
DB  0xff, 0xff, 0xff, 0x48, 0x45, 0x4c, 0x4c, 0x4f
DB  0x2d, 0x4f, 0x53, 0x20, 0x20, 0x20, 0x46, 0x41
DB  0x54, 0x31, 0x32, 0x20, 0x20, 0x20, 0x00, 0x00
RESB  16
DB  0xb8, 0x00, 0x00, 0x8e, 0xd0, 0xbc, 0x00, 0x7c
DB  0x8e, 0xd8, 0x8e, 0xc0, 0xbe, 0x74, 0x7c, 0x8a
DB  0x04, 0x83, 0xc6, 0x01, 0x3c, 0x00, 0x74, 0x09
DB  0xb4, 0x0e, 0xbb, 0x0f, 0x00, 0xcd, 0x10, 0xeb
DB  0xee, 0xf4, 0xeb, 0xfd, 0x0a, 0x0a, 0x68, 0x65
DB  0x6c, 0x6c, 0x6f, 0x2c, 0x20, 0x77, 0x6f, 0x72
DB  0x6c, 0x64, 0x0a, 0x00, 0x00, 0x00, 0x00, 0x00
RESB  368
DB  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x55, 0xaa
DB  0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
RESB  4600
DB  0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
RESB  1469432
```

至此，我们重新用汇编，对我们的“初始系统进行了改写”，虽然还是原来的样子，但是我们的换一种语言，从十六进制代码换成了汇编语言，不得不说，的确也是一种小小的进步鸭。

系统运行

写完程序了，那么接下来我们看看改写的系统是不是也能正常的运行呢？作者已经为我们提供好了汇编编译器。我们用批处理方便运行，首先运行 `!cons` 在打开的命令行窗口输入 `asm` 就可以生成我们的镜像文件 `helloos.img` 文件。在用 `asm` 在用 `asm` 做成 `img` 文件后，再执行 `run` 指令，就可以运行我们的系统了。

```
C:\windows\system32\cmd.exe
(c) 2013 Microsoft Corporation。保留所有权利。
D:\myapp\30s0S\Ini0S\day_01\tolset\helloos1>asm
D:\myapp\30s0S\Ini0S\day_01\tolset\helloos1>..\z_tools\nask.exe helloos.nas hell
oos.img
D:\myapp\30s0S\Ini0S\day_01\tolset\helloos1>run
D:\myapp\30s0S\Ini0S\day_01\tolset\helloos1>copy helloos.img ..\z_tools\qemu\fdi
mage0.bin
已复制      1 个文件。
D:\myapp\30s0S\Ini0S\day_01\tolset\helloos1>..\z_tools\make.exe -C ../z_tools/qe
mu
..\z_tools\make.exe: Entering directory `D:/myapp/30s0S/Ini0S/day_01/tolset/z_to
ols/qemu'
qemu-win.bat
D:\myapp\30s0S\Ini0S\day_01\tolset\z_tools\qemu>qemu.exe -L . -m 32 -localtime -
std-uga -fda fdimage0.bin
..\z_tools\make.exe: Leaving directory `D:/myapp/30s0S/Ini0S/day_01/tolset/z_to
ols/qemu'
D:\myapp\30s0S\Ini0S\day_01\tolset\helloos1>
```

```
QEMU
Bochs VGABIOS ver.0.2
This VGA/VBE BIOS is released under the KL-01
Bochs VBE Support enabled
Bochs BIOS, 1 cpu, $Revision: 1.110 $ $Date: 2004/05/31 13:11:27 $
ata1 master: QEMU CD-ROM ATAPI-4 CD-Rom/DVD-Rom
ata1 slave: Unknown device
Booting from Floppy...

hello, world
_
```

项目目录

其实项目的结构基本没变，就是把 `helloos1` 文件夹加进来了。

```
D:.\
├─helloos0
├─helloos1
├─z_new_o
├─z_new_w
├─z_osabin
└─z_tools
    └─guigui00
```

```

└─haribote
└─osa_qemu
└─qemu
└─qemu_9x
└─win32

└─helloos1
    !cons_9x.bat
    !cons_nt.bat
    asm.bat
    helloos.img
    helloos.nas
    install.bat
    run.bat

```

加工润色汇编系统

上面写的操作系统，虽然只有短短的22行，但是很难看出程序是干什么的。为了我们自己写的程序，别人也能看懂，我们再把 `helloos.nas` 稍微改一下，就成了 `helloos2`。

```

; hello-os
; TAB=4
; 以下这段是标准FAT12格式软盘专用的代码
    DB      0xeb, 0x4e, 0x90
    DB      "HELLOIPL"      ; 启动区的名称可以是任意的字符串（8字节）
    DW      512              ; 每个扇区（sector）的大小（必须为512字节）
    DB      1                ; 簇（cluster）的大小（必须为1个扇区）
    DW      1                ; FAT的起始位置（一般从第一个扇区开始）
    DB      2                ; FAT的个数（必须为2）
    DW      224              ; 根目录的大小（一般设成224项）
    DW      2880              ; 该磁盘的大小（必须是2880扇区）
    DB      0xf0              ; 磁盘的种类（必须是0xf0）
    DW      9                ; FAT的长度（必须是9扇区）
    DW      18               ; 1个磁道（track）有几个扇区（必须是18）
    DW      2                ; 磁头数（必须是2）
    DD      0                ; 不适用分区，必须是0
    DD      2880              ; 重写一次磁盘大小
    DB      0, 0, 0x29        ; 意义不明，固定
    DD      0xffffffff        ; （可能是）卷标号码
    DB      "HELLO-OS "      ; 磁盘的名称（11字节）注意：引号里的空格可不是
    ; 随便填写的，而是为了补充字节
    DB      "FAT12 "          ; 磁盘格式名称（8字节）
    RESB    18                ; 先空出18字节
; 程序主体
    DB      0xb8, 0x00, 0x00, 0x8e, 0xd0, 0xbc, 0x00, 0x7c
    DB      0x8e, 0xd8, 0x8e, 0xc0, 0xbe, 0x74, 0x7c, 0x8a
    DB      0x04, 0x83, 0xc6, 0x01, 0x3c, 0x00, 0x74, 0x09

```

```

DB      0xb4, 0x0e, 0xbb, 0x0f, 0x00, 0xcd, 0x10, 0xeb
DB      0xee, 0xf4, 0xeb, 0xfd

; 信息显示部分
DB      0x0a, 0x0a      ;    2个换行
DB      "hello, Inios by-YJLAugus"
DB      0x0a            ;    换行
DB      0

RESB    0x1fe-$          ;    填写0x00,直到0x00afe

DB      0x55, 0xaa
; 以下是启动区以外部分的输出
DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
RESB    4600
DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
RESB    1469432

```

以上代码说明

- **DW** 和 **DD** 分别是 **data word** 和 **data double-word** 的缩写。其中 **word** 指16位，也就是 **2个字节**，同理 **data double-word** 是32位，也就是 **4个字节**。
- **RESB 0x1fe-\$** 是一个变量，在这里的意思是告诉我们这一行现在的字节数。在这个程序里我们已经输出132字节（详细下面解释）所以这里的 **\$** 就是132。因此 **nasm** (作者自己写的汇编语言编译器)先用 **0x1fe** 减去132，得出378这一个结果，然后连续输出378字节的 **00x00**。
- 那这里我们为什么不直接写378，而非要用 **\$** 呢？这是因为如果将显示信息从“hello,word”变成“hello,Inios”的话，中间要输出的字节数也会随之变化/欢聚话说，我们必须保证软盘的第510字节（即第0x1fe字节）开始的地方是 **55AA**。如果在程序里使用美元符号（\$）的话，汇编语言会自动计算需要输出多少个 **00**，我们也就可以很轻松的改写输出信息了。

132字节说明

一开始真的是在这里出了问题，代码对着书上的代码比对了很久，不存在写错的情况。但是最后编译出来的镜像文件，我查看了下是不一样的。排查了好久，最后直到读到 **132字节** 的时候才发现了问题所在，所以也就解决了。接下来我们就开始介绍下 **132字节** 到底是怎么来的。当然了这里指的是在刚刚写的程序里，如果是再以前的程序里，相信大家很容易就数出来了。

```

; hello-os
; TAB=4
; 以下这段是标准FAT12格式软盘专用的代码
DB      0xeb, 0x4e, 0x90 ;    3B
DB      "HELLOIPL"      ;    8B(字符串的话，只算引号里面的字符)
DW      512              ;    2B
DB      1                ;    1B
DW      1                ;    2B

```

```

DB      2          ;    1B
DW      224        ;    2B
DW      2880       ;    2B
DB      0xf0       ;    1B
DW      9          ;    2B
DW      18         ;    2B
DW      2          ;    2B
DD      0          ;    4B
DD      2880       ;    4B
DB      0,0,0x29   ;    3B
DD      0xffffffff ;    4B
DB      "HELLO-OS  " ;    11B, 引号里的空格可不是随便填写的, 而是为了补
充字节
DB      "FAT12  "   ;    8B, 引号里的空格可不是随便填写的, 而是为了补充
字节
RESB    18         ;    18B
; 程序主体
DB      0xb8, 0x00, 0x00, 0x8e, 0xd0, 0xbc, 0x00, 0x7c ;8B
DB      0x8e, 0xd8, 0x8e, 0xc0, 0xbe, 0x74, 0x7c, 0x8a ;8B
DB      0x04, 0x83, 0xc6, 0x01, 0x3c, 0x00, 0x74, 0x09 ;8B
DB      0xb4, 0x0e, 0xbb, 0x0f, 0x00, 0xcd, 0x10, 0xeb ;8B
DB      0xee, 0xf4, 0xeb, 0xfd ;4B

; 信息显示部分
DB      0x0a, 0x0a ;    2B
DB      "hello, world" ;    12B(有一个空格占一个字节)
DB      0x0a       ;    1B
DB      0          ;    1B 至此共有132字节。

```

第2天

继续改写 `helloos.nas`

以下内容是在 `helloos3` 中

```

; hello-os
; TAB=4

ORG      0x7c00      ; 指明程序的装载地址

; 以下记述对于标准FAT格式软盘

JMP      entry
DB      0x90
DB      "HELLOIPL"   ; 启动区的名称可以是任意的字符串（8字节）
DW      512          ; 每个扇区（sector）的大小（必须为512字节）

```

```

DB      1          ; 簇（cluster）的大小（必须为1个扇区）
DW      1          ; FAT的起始位置（一般从第一个扇区开始）
DB      2          ; FAT的个数（必须为2）
DW      224        ; 根目录的大小（一般设成224项）
DW      2880       ; 该磁盘的大小（必须是2880扇区）
DB      0xf0       ; 磁盘的种类（必须是0xf0）
DW      9          ; FAT的长度（必须是9扇区）
DW      18         ; 1个磁道（track）有几个扇区（必须是18）
DW      2          ; 磁头数（必须是2）
DD      0          ; 不适用分区，必须是0
DD      2880       ; 重写一次磁盘大小
DB      0,0,0x29   ; 意义不明，固定
DD      0xffffffff ; （可能是）卷标号码
DB      "HELLO-OS  " ; 磁盘的名称（11字节）注意：引号里的空格可不是随便
填写的，而是为了补充字节
DB      "FAT12     " ; 磁盘格式名称（8字节）
RESB    18         ; 先空出18字节

; 程序核心

entry:
    MOV     AX,0          ; 初始化寄存器
    MOV     SS,AX
    MOV     SP,0x7c00
    MOV     DS,AX
    MOV     ES,AX

    MOV     SI,msg

putloop:                    ; 循环
    MOV     AL,[SI]        ; 把SI地址的一个字节的内容读入AL中
    ADD     SI,1          ; 给SI加1
    CMP     AL,0          ; 比较AL是否等于0
    JE      fin           ; 如果比较的结果成立，则跳转到fin,fin是一个标号，
表示“结束”
    MOV     AH,0x0e       ; 显示一个文字
    MOV     BX,15         ; 指定字符颜色
    INT     0x10          ; 调用显卡BIOS，INT  是一个中断指令，这里可以暂时
理解为“函数调用”
    JMP     putloop

fin:
    HLT                     ; 让CPU停止，等待指令，让CPU进入待机
    JMP     fin            ; 无限循环

msg:
    DB      0x0a, 0x0a    ; 换行两次
    DB      "hello, world"
    DB      0x0a         ; 换行
    DB      0

```

```
RESB    0x7dfe-$          ; 从0x7dfe地址开始用0x00填充
```

```
DB      0x55, 0xaa
```

; 以下是引导扇区以外的部分的记述

```
DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
```

```
RESB    4600
```

```
DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
```

```
RESB    1469432
```

以上代码说明

- **ORG** 来源于“origin”，功能是告诉程序要从指定的某个地址开始，如果没有它，有几个指令就不能被正确地翻译和执行，另外，一旦有了这个指令，那么美元 **\$** 的含义也会随之发生改变，它不再是指输出文件的第几个字节，而是代表将要读入的内存地址。
- **JMP** 指令来源于“jump”，相当于C语言中的 **goto** 语句，作用是“跳转”。
- **entry**: 这是标签的声明，用于指定 **JMP** 指令的跳转目的地等。
- **MOV** 是赋值的意思。例如：**MOV SS,AX** 相当于 **SS=AX**。其中，这里的 **SS** 和 **AX** 是寄存器，这个寄存器在机器语言中就相当于 **变量** 的功能。
- **INT** 是 **BIOS** 中调用初始在 **BIOS** 中的指令，其后面跟着数字，使用不同的数字可以调用不同的函数，此次调用的是 **0x10**（即16号）函数，它的功能是控制显卡。

CPU中的重要寄存器（16位）

- **AX**——accumulator，累加寄存器
- **CX**——counter，计数寄存器
- **DX**——data，数据寄存器
- **BX**——base，基址寄存器
- **SP**——stack pointer，栈指针寄存器
- **BP**——base pointer，基址指针寄存器
- **SI**——source index，源地址寄存器
- **DI**——destination index，目的变址寄存器

这些寄存器全是16位寄存器，因此可以存储16位的二进制数。**X** 其实是 **extend** “扩展”的意思，因为在此之前CPU中的寄存器都是8位的，以上的寄存器扩展到了16位。

CPU中8个寄存器（8位）

- **AL**——accumulator low，累加寄存器低位
- **CL**——counter low，计数寄存器低位
- **DL**——data low，数据寄存器低位
- **BL**——base low，基址寄存器低位
- **AH**——accumulator high，累加寄存器高位
- **BP**——base pointer，基址指针寄存器

- SI ——source index , 源地址寄存器
- DI ——destination index , 目的变址寄存器

这8个寄存器其实看以看作是16位寄存器的一个组成部分，而不是单独存在于CPU中的。

CPU中的段寄存器（16位数）

- ES —— 附件段寄存器（extra segment）
- CS —— 代码段寄存器（code segment）
- SS —— 栈段寄存器（stack segment）
- DS —— 数据段寄存器（data segment）
- FS —— 没有名称（segment part 2）
- GS —— 没有名称（segment part 3）

制作启动区

考虑到以后的开发，我们不应该一下就做出整个磁盘映像，而是先只作出 512 字节的启动区。为什么要用512字节呢，因为在前面的代码中 `DW 512` ；每个扇区（sector）的大小（必须为512字节） 我们可以发现，规定的是每个扇区的大小为512字节，这样一以来，我们只需要用第一个扇区来做最初的启动区就好了。对 `helloos.nas` 后半部分截掉，做成 `ipl.nas` 称之为启动区。

然后对 `asm.bat` 进行改造，将输出的文件名称改为 `ipl.bin` 。另外也顺便输出列表文件 `ipl.lst` 。这是一个文本文件，可以用来简单的 **确认每个指令时怎么翻译成机器语言的**。至今，我们还是第一次输出这样的列表文件，原因是当初的“初始系统”足足有 1440KB ,输出列表的话实在是太大了，这次的启动区只有512字节，输出一下就显得容易多了。

另外 `makeimg.bat` 这个文件是以 `ipl.bin` 为基础，制作磁盘映像文件 `helloos.img` 的批处理文件。并利用作者自己开发的磁盘映像管理工具 `edimg.exe` ，先杜宇一个空白的磁盘映像文件，然后再在文件的开头写入 `ipl.bin` 的内容，最后将结果输出为名字为 `helloos.img` 的镜像文件。

通过这些批处理工具，我们的系统从编译到测试就变得很简单了，只需在 `!cons` 中依次输入 `asm - makeimg - run` 即可，此时我们在 `helloos4` 中编写。

ipl.nas

```
; hello-os
; TAB=4

ORG      0x7c00

JMP      entry
DB       0x90
DB       "HELLOIPL"
```



```

DW      512
DB      1
DW      1
DB      2
DW      224
DW      2880
DB      0xf0
DW      9
DW      18
DW      2
DD      0
DD      2880
DB      0,0,0x29
DD      0xffffffff
DB      "HELLO-OS"
DB      "FAT12"
RESB    18

```

; 程序核心

entry:

```

MOV     AX,0
MOV     SS,AX
MOV     SP,0x7c00
MOV     DS,AX
MOV     ES,AX

```

```

MOV     SI,msg

```

putloop:

```

MOV     AL,[SI]
ADD     SI,1
CMP     AL,0
JE      fin
MOV     AH,0x0e
MOV     BX,15
INT     0x10
JMP     putloop

```

fin:

```

HLT
JMP     fin

```

msg:

```

DB      0x0a, 0x0a
DB      "hello,Inios"
DB      0x0a
DB      0

RESB    0x7dfe-$

```

Makefile 入门

Makefile 可以当作非常聪明的批处理文件。有了它就会很聪明的解决一写“重复性”的工作，并加以判别，提高效率。主要是在以下几个方面起作用。

- 可以根据 **Makefile** 中写的代码，根据条件进行一个判断，然后再去执行，这样提高了效率，避免重复工作。
- 可以判断文件是否存在。
- 可以判断文件的输入日期，并依据此来决定是否需要重新生成输出文件。

上面提到的几个方面是略显抽象的，大家可以对照的书上的例子依次尝试一下，看看是不是达到了预期的结果。接下来呢，对 **Makefile** 的内容加以改写，这样以来，我们以前用到的 **run.bat** 和 **install.bat** 就都用不到了，因为我们已经把其中的内容放在了 **Makefile** 中。此时我们就可以“放心”的去只想 **make img** 了。而且就算是直接 **make run** 也可以顺利运行。**make install** 也是一样，只要把磁盘装到驱动器里，这个命令就会自动作出判断，如果已经有了最新 **helloos.img** 就直接安装，没有的话就先自动生成新的 **helloos.img** ,然后安装。

其中在 **Makefile** 中还增加了如下命令：

- **make clean** ：删掉除了最终成果(helloos.img)以外的所有的中间“衍生物”在这里的话就是 **ipl.bin** 和 **ipl.lst** 。
- **make src_only** ：删掉除了源程序 (**ipl.nas**) 外的其他文件，这里指的是 **ipl.bin** 、 **ipl.lst** 和 **helloos.img** 。
- **make** ：执行不带参数的 **make** 时。相当与执行的是 **make img** (这属于默认动作，默认动作放在Makefile的最前头)。

哦，对了，**Makefile** 之所以这么强大，其实是依赖于 **make.exe** ,这个软件是GUN项目组的人开发，特表感谢。

Makefile 源代码

说了这么多，一起来看一下最终的 **Makefile** 文件到底是怎样的吧！说明一点 \ 是续行符号，表示这一行太长写不下去，跳转到 下一行接着写。 **Makefile**

```
default :
    ../z_tools/make.exe img

ipl.bin : ipl.nas Makefile
    ../z_tools/nask.exe ipl.nas ipl.bin ipl.lst

helloos.img : ipl.bin Makefile
    ../z_tools/edimg.exe  imgin:../z_tools/fdimg0at.tek \
        wbinimg src:ipl.bin len:512 from:0 to:0  imgout:helloos.img
```

```

asm :
    ../z_tools/make.exe -r ipl.bin

img :
    ../z_tools/make.exe -r helloos.img

run :
    ../z_tools/make.exe img
    copy helloos.img ..\z_tools\qemu\fdimage0.bin
    ../z_tools/make.exe -C ../z_tools/qemu

install :
    ../z_tools/make.exe img
    ../z_tools/imgtol.com w a: helloos.img

clean :
    -del ipl.bin
    -del ipl.lst

src_only :
    ../z_tools/make.exe clean
    -del helloos.img

```

第3天

制作真正的IPL

前面写到的 `ipl` 实际上并没有装载任何程序，从现在开始我们开始用它来真正的装载程序了。添加了部分内容如下：下面的程序是在 `harib00a` 的 `ipl.nas`

```

; haribote-ipl
; TAB=4

ORG      0x7c00      ; 指明程序的装载地址

; 以下这段是标准FAT12格式软盘专用的代码

DB      0xeb, 0x4e, 0x90
DB      "HELLOIPL"   ; 启动区的名称可以是任意的字符串（8字节）
DW      512          ; 每个扇区（sector）的大小（必须为512字节）
DB      1            ; 簇（cluster）的大小（必须为1个扇区）
DW      1            ; FAT的起始位置（一般从第一个扇区开始）
DB      2            ; FAT的个数（必须为2）

```

```

        DW      224          ; 根目录的大小（一般设成224项）
        DW      2880        ; 该磁盘的大小（必须是2880扇区）
        DB      0xf0        ; 磁盘的种类（必须是0xf0）
        DW      9           ; FAT的长度（必须是9扇区）
        DW      18          ; 1个磁道（track）有几个扇区（必须是18）
        DW      2           ; 磁头数（必须是2）
        DD      0           ; 不适用分区，必须是0
        DD      2880        ; 重写一次磁盘大小
        DB      0,0,0x29    ; 意义不明，固定
        DD      0xffffffff  ; （可能是）卷标号码
        DB      "HELLO-OS"  ; 磁盘的名称（11字节）
        DB      "FAT12"     ; 磁盘格式名称（8字节）
        RESB    18          ; 先空出18字节

```

; 程序主体

entry:

```

        MOV     AX,0        ; 初始化寄存器
        MOV     SS,AX
        MOV     SP,0x7c00
        MOV     DS,AX

```

; 读盘

```

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0        ; 柱面0 从正面读
        MOV     DH,0        ; 磁头0
        MOV     CL,2        ; 扇区2，第一个扇区用作制作启动区了

        MOV     AH,0x02     ; AH=0x02 : 读入磁盘
        MOV     AL,1        ; 1个扇区
        MOV     BX,0
        MOV     DL,0x00     ; A驱动器
        INT     0x13        ; 调用磁盘BIOS
        JC      error

```

fin:

```

        HLT          ; 让CPU停止，等待指令，让CPU进入待机
        JMP     fin   ; 无限循环

```

error:

```

        MOV     SI,msg

```

putloop:

```

        MOV     AL,[SI]
        ADD     SI,1        ; 把SI地址的一个字节的内容读入AL中
        CMP     AL,0        ; 比较AL是否等于0

```

```

JE      fin      ; 如果比较的结果成立，则跳转到fin, fin是一个标号，
表示“结束”

MOV     AH, 0x0e ; 显示一个文字
MOV     BX, 15   ; 指定字符颜色
INT     0x10     ; 调用显卡BIOS, INT 是一个中断指令，这里可以暂时
理解为“函数调用”

JMP     putloop

msg:
DB      0x0a, 0x0a ; 换行两次
DB      "load error"
DB      0x0a       ; 换行
DB      0

RESB    0x7dfe-$   ; 从0x7dfe地址开始用0x00填充

DB      0x55, 0xaa

```

代码相关说明

磁盘读、写，扇区校验 (verify) ,以及寻道 (seek)

- `AH` =0x02 ; (读盘)
- `AH` =0x03 ; (写盘)
- `AH` =0x04 ; (校检)
- `AH` =0x0c ; (寻道)
- `AL` =理对象的扇区数 ; (只能同时处理连续的扇区)
- `CH` =柱面号 &0xff
- `CL` =扇区号 (0-5位) | (柱面号 &0x300) >>2;
- `DH` =磁头号 ;
- `DL` =驱动器号 ;
- `ES:BS` =缓冲地址 ; (校验及寻道时不使用)

返回值：

- `FLACS.CF` ==0 : 没有错误 , `AH` ==0
- `FLAGS.CF` ==1: 有错误 , 错误号码存入 `AH` 内 (与重置 (reset) 功能一样)

指令 `JC` 是“jump if carry”的缩写，意思是如果进位标志 (carry flag) 是1的话，就跳转。所谓 标志 就是只有 一位 信息的寄存器。这里 `BIOS函数` 调用是否有错误，如果没有错误就返回0，如果有错就返回1。

其他的几个寄存器我们也来看一下，`CH` 表示柱面号；`CL` 表示扇区号；`DH` 表示磁头号；`DL` 表示驱动器号。这里我们用的是一个软盘也就是 `DL=0` 了。

思路概括

计算机开机时加载IPL程序（initial program loader，一个nas汇编程序）的情况，包括IPL代码（helloos.nas）、编译生成helloos.img文件、用虚拟机QEMU加载helloos.img、制作U盘启动盘和用物理机加载helloos.img。

计算机启动时会自动加载和执行IPL程序，但IPL程序只能占用512字节。若直接用IPL写OS，空间不够用。所以IPL程序一般用于将真正的OS程序加载到内存某处（记作A），然后跳转到 A。这样计算机就可以执行OS的程序了。

在上一篇中的IPL程序只是个hello world式的试验品，本篇通过修改上一篇的IPL，让它真正实现加载OS程序的功能。同时，将IPL程序代码和OS代码放到不同的源代码文件中；用C语言来编写以后的OS代码；用Makefile来编译源代码。

有了本篇的基础，就算是正式开始编写OS源代码了。

软盘相关预备知识

一张软盘有80个柱面、2个磁头、18个扇区（Cylinder：0~79；Header：0~2；Sector：1~18），1个扇区有512个字节，所以软盘的容量是 $80 \times 2 \times 18 \times 512 = 1440\text{KB}$ 。

向一个软盘保存文件时，文件名会从 0x2600 开始往后存，文件的内容会从 0x4200 开始往后存。含有IPL的启动区，位于C0-H0-S1（柱面0，磁头0，扇区1的缩写），下一个扇区是C0-H0-S2，这次我们要装载的就是这个扇区，也就是说，要装在 第二 扇区。

试错

软盘有时很不靠谱，会发生不能读取数据的情况。那么解决的办法就是多读几次，这里我们让它多试5次，所以上面的程序如果真的是在软盘中的话怕是要出问题的，所以我们再改进一下程序。此时的程序是在 harib00b 。

ipl.nas

```

; haribote-ipl
; TAB=4

ORG      0x7c00      ; 指明程序的装载地址

; 以下这段是标准FAT12格式软盘专用的代码

DB      0xeb, 0x4e, 0x90
DB      "HELLOIPL"      ; 启动区的名称可以是任意的字符串（8字节）
DW      512      ; 每个扇区（sector）的大小（必须为512字节）
DB      1      ; 簇（cluster）的大小（必须为1个扇区）
DW      1      ; FAT的起始位置（一般从第一个扇区开始）
DB      2      ; FAT的个数（必须为2）
DW      224      ; 根目录的大小（一般设成224项）
DW      2880      ; 该磁盘的大小（必须是2880扇区）
DB      0xf0      ; 磁盘的种类（必须是0xf0）
DW      9      ; FAT的长度（必须是9扇区）
DW      18      ; 1个磁道（track）有几个扇区（必须是18）
```

```

DW      2          ; 磁头数（必须是2）
DD      0          ; 不适用分区，必须是0
DD      2880       ; 重写一次磁盘大小
DB      0,0,0x29   ; 意义不明，固定
DD      0xffffffff ; （可能是）卷标号码
DB      "HELLO-OS  " ; 磁盘的名称（11字节）
DB      "FAT12     " ; 磁盘格式名称（8字节）
RESB    18         ; 先空出18字节

```

; 程序主体

entry:

```

MOV     AX,0        ; 初始化寄存器
MOV     SS,AX
MOV     SP,0x7c00
MOV     DS,AX

```

; 读盘

```

MOV     AX,0x0820
MOV     ES,AX
MOV     CH,0        ; 柱面0
MOV     DH,0        ; 磁头0
MOV     CL,2        ; 扇区2，第一个扇区用作制作启动区了

MOV     SI,0        ; 记录失败次数的寄存器

```

retry:

```

MOV     AH,0x02     ; AH=0x02 : 读入磁盘
MOV     AL,1        ; 1个扇区
MOV     BX,0
MOV     DL,0x00     ; A驱动器
INT     0x13        ; 调用磁盘BIOS 判断是否发生错误
JNC     fin         ; 没出错的话就跳转到fin
ADD     SI,1        ; 往SI 加 1
CMP     SI,5        ; 比较SI 和 5
JAE     error       ; SI >= 5时，跳转到error
MOV     AH,0x00
MOV     DL,0x00     ; A 驱动器
INT     0x13        ; 重置驱动器
JMP     retry

```

fin:

```

HLT          ; 让CPU停止，等待指令，让CPU进入待机
JMP     fin  ; 无限循环

```

```

error:
    MOV     SI,msg
putloop:
    MOV     AL,[SI]
    ADD     SI,1           ; 把SI地址的一个字节的内容读入AL中
    CMP     AL,0          ; 比较AL是否等于0
    JE      fin           ; 如果比较的结果成立，则跳转到fin,fin是一个标号，
                          ; 表示“结束”
    MOV     AH,0x0e       ; 显示一个文字
    MOV     BX,15         ; 指定字符颜色
    INT     0x10          ; 调用显卡BIOS，INT 是一个中断指令，这里可以暂时
                          ; 理解为“函数调用”
    JMP     putloop
msg:
    DB      0x0a, 0x0a    ; 换行两次
    DB      "load error"
    DB      0x0a          ; 换行
    DB      0
    RESB    0x7dfe-$      ; 从0x7dfe地址开始用0x00填充
    DB      0x55, 0xaa

```

代码说明

JNC 是另外一条跳转指令，是“Jump if not carry”的缩写。即如果进位标志为0（没有错误）的话就跳转。**JAE** 也是跳转条件，是“Jump if above or equal”的缩写，意思是 **大于或等于** 时发生跳转。

错误时候的处理，重新读盘之前，我们做了以下处理，AH=0x00，DL=0x00，INT 0x13 称之为“系统复位”也就是重新赋值。

读到18扇区

下面写到的内容我们把它放在 **harib00c** 的文件夹中。更新 **ipl.nas** 如下：

ipl.nas

```

; haribote-ipl
; TAB=4

    ORG     0x7c00          ; 指明程序的装载地址

; 以下这段是标准FAT12格式软盘专用的代码

    JMP     entry
    DB      0x90
    DB      "HELLOIPL"      ; 启动区的名称可以是任意的字符串（8字节）

```



```
DW      512      ; 每个扇区（sector）的大小（必须为512字节）
DB      1        ; 簇（cluster）的大小（必须为1个扇区）
DW      1        ; FAT的起始位置（一般从第一个扇区开始）
DB      2        ; FAT的个数（必须为2）
DW      224      ; 根目录的大小（一般设成224项）
DW      2880     ; 该磁盘的大小（必须是2880扇区）
DB      0xf0     ; 磁盘的种类（必须是0xf0）
DW      9        ; FAT的长度（必须是9扇区）
DW      18       ; 1个磁道（track）有几个扇区（必须是18）
DW      2        ; 磁头数（必须是2）
DD      0        ; 不适用分区，必须是0
DD      2880     ; 重写一次磁盘大小
DB      0,0,0x29 ; 意义不明，固定
DD      0xffffffff ; （可能是）卷标号码
DB      "HELLO-OS" ; 磁盘的名称（11字节）
DB      "FAT12"   ; 磁盘格式名称（8字节）
RESB    18       ; 先空出18字节

; 程序主体

entry:
    MOV     AX,0      ; 初始化寄存器
    MOV     SS,AX
    MOV     SP,0x7c00
    MOV     DS,AX

; 读盘

    MOV     AX,0x0820
    MOV     ES,AX
    MOV     CH,0      ; 柱面0
    MOV     DH,0      ; 磁头0
    MOV     CL,2      ; 扇区2，第一个扇区用作制作启动区了

readloop:

    MOV     SI,0      ; 记录失败次数的寄存器

retry:
    MOV     AH,0x02   ; AH=0x02 : 读入磁盘
    MOV     AL,1      ; 1个扇区
    MOV     BX,0
    MOV     DL,0x00   ; A驱动器
    INT     0x13      ; 调用磁盘BIOS 判断是否发生错误
    JNC     fin       ; 没出错的话就跳转到fin
    ADD     SI,1      ; 往SI 加 1
    CMP     SI,5      ; 比较SI 和 5
```

```

JAE    error        ; SI >= 5时，跳转到error
MOV    AH,0x00
MOV    DL,0x00      ; A 驱动器
INT    0x13         ; 重置驱动器
JMP    retry

next:
MOV    AX,ES        ; 把内存地址后移0x200, 往后移动一个扇区512B换成16
进制是0x200
ADD    AX,0x0020
MOV    ES,AX        ; 扇区移动了, 同样的缓冲地址也需要更新, 因为没有
ADD ES,0x200指令, 所以这里以这种方式实现累加ES（更新ES）的目的
ADD    CL,1         ; 往CL里加1, 扇区是第二个扇区了鸭
CMP    CL,18        ; 比较CL与18
JBE    readloop     ; 如果CL <= 18 跳转至readloop

fin:
HLT
JMP    fin          ; 无限循环

error:
MOV    SI,msg
putloop:
MOV    AL,[SI]
ADD    SI,1         ; 把SI地址的一个字节的内容读入AL中
CMP    AL,0         ; 比较AL是否等于0
JE     fin          ; 如果比较的结果成立, 则跳转到fin, fin是一个标号,
表示“结束”
MOV    AH,0x0e      ; 显示一个文字
MOV    BX,15        ; 指定字符颜色
INT    0x10         ; 调用显卡BIOS, INT 是一个中断指令, 这里可以暂时
理解为“函数调用”
JMP    putloop

msg:
DB     0x0a, 0x0a   ; 换行两次
DB     "load error"
DB     0x0a         ; 换行
DB     0

RESB   0x7dfe-$     ; 从0x7dfe地址开始用0x00填充

DB     0x55, 0xaa

```

代码说明

JBE 也是一种跳转指令 是“jump if below or equal”的缩写，意思是小于等于则跳转。

第一个扇区已经做了启动区，所以要读下一个扇区，只需要给 CL (扇区号)加1，给 ES (指定读入地址)，0x200是512转换成16进制的数，根据我们前面讲的**我们在使用段寄存器，以 ES:BX 这种方式来表示地址，写成“MOV AL,[ES:BX]”它表示 $ES * 16 + BX$ 的内存地址**。在这个程序里我们能够知道 BX = 0,故 知道内存地址是0x200的情况下,又因 BX=0，所以512/16 转换成16进制就是 0x20。写到这里，可能又有人会说，程序里不是 0x0020 吗？的确，其实这两个本质是一样的，这两种写法只是在不同的编码环境两种不同的写法，所以，关于这一点，大家就不用纠结了。

程序中的 readloop 等字段，前面也提到过是寄存器的名字，在这里可以理解成是变量，也就是自己定义的名字（我觉得是这样的，后面再看），还有一点，程序中在读到18扇区这里为什么用循环，的确用作者的话说，直接在调用 INT 0x13 这个地方，只要把 AL 的值设置成17就好了。这样就是2-18是一共17个扇区，再往下读就是18 个扇区。因为我们的系统是写在软盘上的（虽然是用模拟器模拟的），如果是软盘的话，好像（作者书中也是用的“似乎一词”）是不能跨多个磁道的，也就是我们必须保证在“读到18扇区”的过程中必须要扇区是连续的，这也就是用循环的原因。至于循环，还有一点，一般是发上跳转的地方都是用到循环的。

到此为止，虽然系统没有什么变化，但是我们已经把磁盘上C0-H0-S2到C0-H0-S18的517 * 17 = 8704字节的内容，装载到了内存的 0x8200~0xa3ff 处，其中的0x8200是怎么来的呢，程序中我们用的是 0x820 是段地址，当cpu 处理时，左移一位：0x200。但是这个 0xa3ff 是 0x8200~0xa3ff 之间一共是8704个字节。0x8200转换成10进制是33280，0xa3ff转换成10进制是41983，从0x8200~0xa3ff（包括端点）共有8704个字节。

读入10个柱面

写到这里，我们的系统已经读入了18个扇区，也就是现在是在 C0-H0-S18 扇区，因为软盘的一面是一共就是18个扇区，所以这个扇区的下一个扇区是软盘的反面的 C0-H1-S1。根据我们前面的“软盘相关预备知识”所讲的，软盘中真正开始寸写的文件的内容是 0xa400 开始的，所以这次也是从这里开始吧。下面我们从反面读完18个扇区后(至此，正反面所有扇区都读完，一个柱面结束)，然后再读入9个柱面，也即从 C0-H1-S1 到 C9-H1-S18 一共是10个柱面。接下来是 hrib00d 中的内容。

ipl.nas

```
; haribote-ipl
; TAB=4
CYLS    EQU    10                ; 定义10个柱面

        ORG    0x7c00            ; 指明程序的装载地址

; 以下这段是标准FAT12格式软盘专用的代码

        JMP    entry
        DB     0x90
        DB     "HELLOIPL"        ; 启动区的名称可以是任意的字符串（8字节）
        DW     512                ; 每个扇区（sector）的大小（必须为512字节）
        DB     1                  ; 簇（cluster）的大小（必须为1个扇区）
```

```
DW      1          ;   FAT的起始位置（一般从第一个扇区开始）
DB      2          ;   FAT的个数（必须为2）
DW      224        ;   根目录的大小（一般设成224项）
DW      2880       ;   该磁盘的大小（必须是2880扇区）
DB      0xf0       ;   磁盘的种类（必须是0xf0）
DW      9          ;   FAT的长度（必须是9扇区）
DW      18         ;   1个磁道（track）有几个扇区（必须是18）
DW      2          ;   磁头数（必须是2）
DD      0          ;   不适用分区，必须是0
DD      2880       ;   重写一次磁盘大小
DB      0,0,0x29   ;   意义不明，固定
DD      0xffffffff ;   （可能是）卷标号码
DB      "HELLO-OS  " ;   磁盘的名称（11字节）
DB      "FAT12     " ;   磁盘格式名称（8字节）
RESB    18         ;   先空出18字节

; 程序主体

entry:
    MOV     AX,0          ;   初始化寄存器
    MOV     SS,AX
    MOV     SP,0x7c00
    MOV     DS,AX

; 读盘

    MOV     AX,0x0820
    MOV     ES,AX
    MOV     CH,0          ;   柱面0
    MOV     DH,0          ;   磁头0
    MOV     CL,2          ;   扇区2，第一个扇区用作制作启动区了

readloop:

    MOV     SI,0          ;   记录失败次数的寄存器

retry:
    MOV     AH,0x02       ;   AH=0x02 : 读入磁盘
    MOV     AL,1          ;   1个扇区
    MOV     BX,0
    MOV     DL,0x00       ;   A驱动器
    INT     0x13          ;   调用磁盘BIOS 判断是否发生错误
    JNC     next          ;   没出错的话就跳转到next
    ADD     SI,1          ;   往SI 加 1
    CMP     SI,5          ;   比较SI 和 5
    JAE     error         ;   SI >= 5时，跳转到error
    MOV     AH,0x00
```

```

MOV     DL,0x00           ; A 驱动器
INT     0x13              ; 重置驱动器
JMP     retry

next:
MOV     AX,ES              ; 把内存地址后移0x200, 往后移动一个扇区512B换成16
进制是0x200
ADD     AX,0x0020
MOV     ES,AX              ; 扇区移动了, 同样的缓冲地址也需要更新, 因为没有
ADD     ES,
                                ; 0x200指令, 所以这里以这种方式实现累加ES (更新
ES) 的目的
ADD     CL,1              ; 往CL里加1, 扇区是第二个扇区了鸭
CMP     CL,18              ; 比较CL与18
JBE     readloop          ; 如果CL <= 18 跳转至readloop
MOV     CL,1
ADD     DH,1
CMP     DH,2
JB      readloop          ; 如果DH < 2, 则跳转到readloop
MOV     DH,0
ADD     CH,1
CMP     CH,CYLS
JB      readloop          ; 如果CH < CYLS, 则跳转到readloop

fin:
HLT
JMP     fin                ; 无限循环

error:
MOV     SI,msg
putloop:
MOV     AL,[SI]
ADD     SI,1              ; 把SI地址的一个字节的内容读入AL中
CMP     AL,0              ; 比较AL是否等于0
JE      fin               ; 如果比较的结果成立, 则跳转到fin, fin是一个标号,
表示“结束”
MOV     AH,0x0e           ; 显示一个文字
MOV     BX,15              ; 指定字符颜色
INT     0x10              ; 调用显卡BIOS, INT 是一个中断指令, 这里可以暂时
理解为“函数调用”
JMP     putloop

msg:
DB      0x0a, 0x0a        ; 换行两次
DB      "load error"
DB      0x0a              ; 换行
DB      0
RESB    0x7dfe-$          ; 从0x7dfe地址开始用0x00填充

```

DB 0x55, 0xaa

代码说明

JB 指令也是跳转指令，是“jump if below”的缩写，即如果小于就跳转。**EQU** 相当于C语言中的 **#define** 用来声明常数。现在启动程序基本写完，现在已经可以把软盘上最初的 $10 * 2 * 8 * 512 = 184320 \text{ byte} = 180\text{KB}$ 内容完整的转载到内存了。运行之后，整个程序还是和以前一样，画面没有什么变化，但这个程序已经用软盘读取的数据填满了内存 **0x08200~0x34fff** 的地方。这里不再进行计算验证，这正好是180KB。

着手开发操作系统

至此，我们启动区制作完毕。此时，我们已经进行到了 **harib00ee**。先写了一个简单的例子：

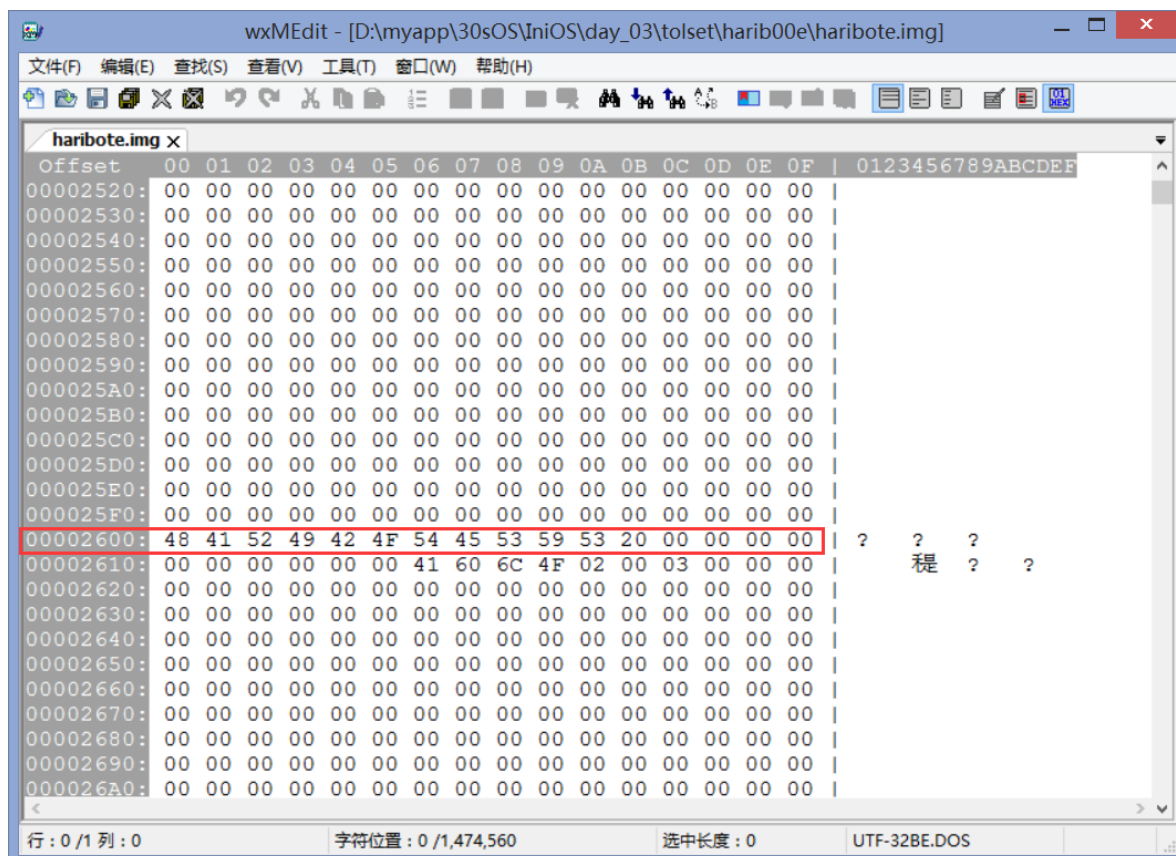
```
fin:
    HLT
    JMP fin
```

将以上的内容保存成 **haribote.nas** 用nask编译,输出 **haribote.sys**。在 **!con** 打开，执行下面的命令。

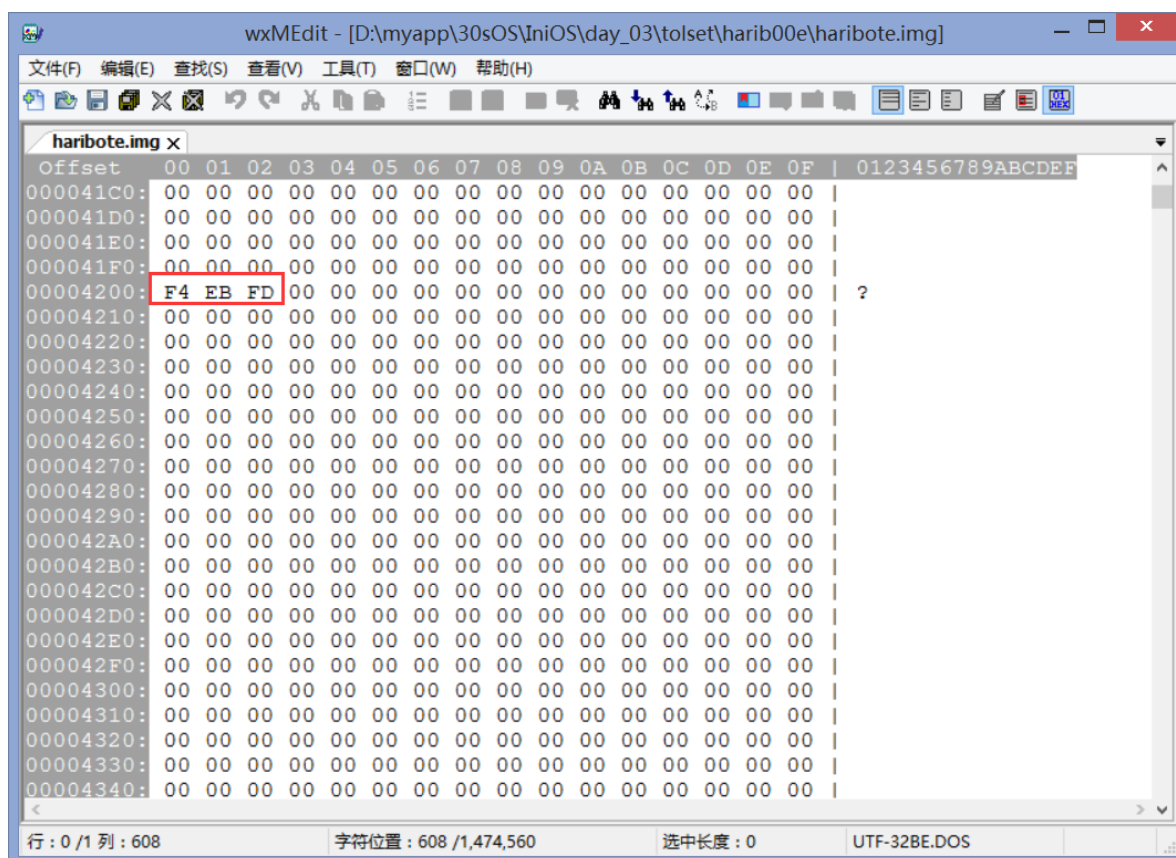
```
..\z_tools\nask.exe haribote.nas haribote.sys
```

执行完这个命令，我们可以看到已经生成了一个 **haribote.sys** 文件。接下来就是将这个文件保存在 **haribote.img**，也也就是“保存到磁盘映像里”的意思。接下来我们需要执行命令 **make img** 来把 **haribote.sys** 保存在 **haribote.img** 中去，也就是说要生成一个新的 **haribote.img**。很简单，新的镜像文件被生成。

注意看0x002600附近,这个地方是 **haribote.sys** 的文件名。



还有一个地方是0x004200，这个地方是 `haribote.sys` 的内容。



小总结

- 文件名会写在 0x002600 以后的地方。
- 文件的内容会写在 0x004200 以后的地方。

关于这一点，以前在讲软盘预备知识的时候有提到过。得到了以上两点，下面要做的事情就简单了。我们将**操作系统本省的内容写到名为 haribote.sys 的文件中，再把它保存到磁盘映像中，然后我们从 启动区 执行这个 haribote.sys 就行了。** 接下来我们要做的就是这件事情。

从启动区执行操作系统

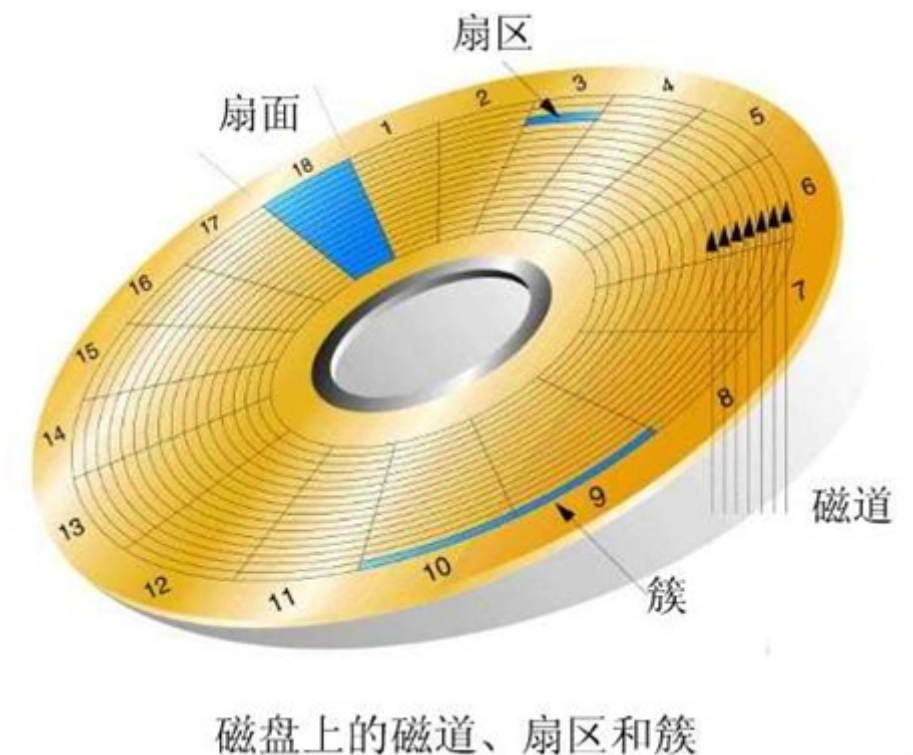
现在的程序是在启动区开始的，简单地说，这个 `ipl.nas` 读了软盘最开始的10个柱面，即C0-H0-S1到C9-H1-S18。那么从软盘（U盘）读到的这些内容放到哪里了呢？答：放到了内存的0x8000到0x34FFF这一段空间，如下表所示。

序号(扇区数)	软盘位置	内存位置	说明
1	C0-H0-S1	0x8000~0x81FF	实际上没有读这一扇区，这一扇区存的是IPL程序
2	C0-H0-S2	0x8200~0x83FF	从软盘（U盘）的512字节到内存的512字节的一一对应。
3	C0-H0-S3	0x8400~0x85FF	同上
...
360	C9-H1-S18	0x34E00~0x34FFF	同上

说明

我们已经知道，一个磁盘是大小是 `512B` 我们以一个扇区为例：0x8000~0x81FF(大小写都可)， $0x81FF-0x8000 = 0x01FF = 511$ 。也就是0x81FF~0x8000之间一共是 `512` 个字节，这也恰好验证了我们一个扇区的大小是512B是正确。至于C0-H0-S2 为什么从0x8200开始，其实这一点，是不是有点不懂的地方就是，为什么会连不在一块？这一点就要在从软盘的结构开始了。如下图：

软盘结构



在上面的图中，我们能看到，是从“外”到“内”的才是扇区。这样我们理解了扇区的结构，这样我们也就能解释扇区不“连续”的问题了。还有就是0x8000~0x81FF 首尾地址分别加上0x200 也正好是下一个扇区。

现在的程序是从启动区开始，把磁盘上的内容装载到内存0x800号地址，所以磁盘0x4200处的内容就应该位于内存0x800 + 0x4200 = 0xc200号地址。此时，我们把上面进行的工作、代码放在 harib00f 中。这时候我们运行 make run ，我们的系统还是老样子，没有什么特变的变化，那程序到底有没有执行 haribote.sys 呢，我们是不得而知的，下来我们就测试一下。

确认操作系统的执行情况

接下来我们将在文件夹 harib00g 中进行。这次我们试试切换画面模式。所以这次我们重写 haribote.nas ,并且 ipl.nas 也做了修改，重新命名为 ipl10.nas ，同时里面代码也稍微做了修改。

haribote.nas

```
; haribote-os
; TAB=4

ORG      0xc200          ; 这个程序将要装载的地址。

MOV      AL, 0x13        ; VGA显卡, 320 * 200 * 8 位色彩。
MOV      AH, 0x00
INT      0x10

fin:
HLT
JMP      fin
```

ipl10.nas

```
; haribote-ipl
; TAB=4
CYLS    EQU    10

        ORG     0x7c00        ; 指明程序的装载地址

; 以下这段是标准FAT12格式软盘专用的代码

        JMP     entry
        DB      0x90
        DB      "HELLOIPL"    ; 启动区的名称可以是任意的字符串（8字节）
        DW      512            ; 每个扇区（sector）的大小（必须为512字节）
        DB      1              ; 簇（cluster）的大小（必须为1个扇区）
        DW      1              ; FAT的起始位置（一般从第一个扇区开始）
        DB      2              ; FAT的个数（必须为2）
        DW      224            ; 根目录的大小（一般设成224项）
        DW      2880           ; 该磁盘的大小（必须是2880扇区）
        DB      0xf0           ; 磁盘的种类（必须是0xf0）
        DW      9              ; FAT的长度（必须是9扇区）
        DW      18             ; 1个磁道（track）有几个扇区（必须是18）
        DW      2              ; 磁头数（必须是2）
        DD      0              ; 不适用分区，必须是0
        DD      2880           ; 重写一次磁盘大小
        DB      0,0,0x29       ; 意义不明，固定
        DD      0xffffffff     ; （可能是）卷标号码
        DB      "HELLO-OS"     ; 磁盘的名称（11字节）
        DB      "FAT12"        ; 磁盘格式名称（8字节）
        RESB    18             ; 先空出18字节

; 程序主体

entry:
        MOV     AX, 0          ; 初始化寄存器
        MOV     SS, AX
        MOV     SP, 0x7c00
        MOV     DS, AX

; 读盘

        MOV     AX, 0x0820
        MOV     ES, AX
        MOV     CH, 0          ; 柱面0
        MOV     DH, 0          ; 磁头0
        MOV     CL, 2          ; 扇区2，第一个扇区用作制作启动区了

readloop:
```

```

MOV     SI, 0                ; 记录失败次数的寄存器

retry:
MOV     AH, 0x02             ; AH=0x02 : 读入磁盘
MOV     AL, 1                ; 1个扇区
MOV     BX, 0
MOV     DL, 0x00             ; A驱动器
INT     0x13                 ; 调用磁盘BIOS 判断是否发生错误
JNC     next                 ; 没出错的话就跳转到next
ADD     SI, 1                 ; 往SI 加 1
CMP     SI, 5                 ; 比较SI 和 5
JAE     error                 ; SI >= 5时, 跳转到error
MOV     AH, 0x00
MOV     DL, 0x00             ; A 驱动器
INT     0x13                 ; 重置驱动器
JMP     retry

next:
MOV     AX, ES                ; 把内存地址后移0x200, 往后移动一个扇区512B换成16
进制是0x200
ADD     AX, 0x0020
MOV     ES, AX                ; 扇区移动了, 同样的缓冲地址也需要更新, 因为没有
ADD     ES,
                                ; 0x200指令, 所以这里以这种方式实现累加ES (更新
ES)的目的
ADD     CL, 1                 ; 往CL里加1, 扇区是第二个扇区了鸭
CMP     CL, 18                ; 比较CL与18
JBE     readloop              ; 如果CL <= 18 跳转至readloop
MOV     CL, 1
ADD     DH, 1
CMP     DH, 2
JB      readloop              ; 如果DH < 2, 则跳转到readloop
MOV     DH, 0
ADD     CH, 1
CMP     CH, CYLS
JB      readloop              ; 如果CH < CYLS, 则跳转到readloop

; ipl的结束地址告诉haribote.sys

MOV     [0x0ff0], CH          ; IPL的结束地址 (将CYLS的值写到内存地址0x0ff0中)
JMP     0xc200

error:
MOV     SI, msg
putloop:
MOV     AL, [SI]
ADD     SI, 1                 ; 把SI地址的一个字节的内容读入AL中
CMP     AL, 0                 ; 比较AL是否等于0

```

```

JE      fin          ; 如果比较的结果成立，则跳转到fin,fin是一个标号，
表示“结束”

MOV     AH,0x0e      ; 显示一个文字
MOV     BX,15        ; 指定字符颜色
INT     0x10         ; 调用显卡BIOS, INT 是一个中断指令，这里可以暂时
理解为“函数调用”

JMP     putloop

fin:
HLT     ; 让CPU停止，等待指令，让CPU进入待机
JMP     fin          ; 无限循环

msg:
DB      0x0a, 0x0a    ; 换行两次
DB      "load error"
DB      0x0a          ; 换行
DB      0

RESB    0x7dfe-$      ; 从0x7dfe地址开始用0x00填充

DB      0x55, 0xaa

```

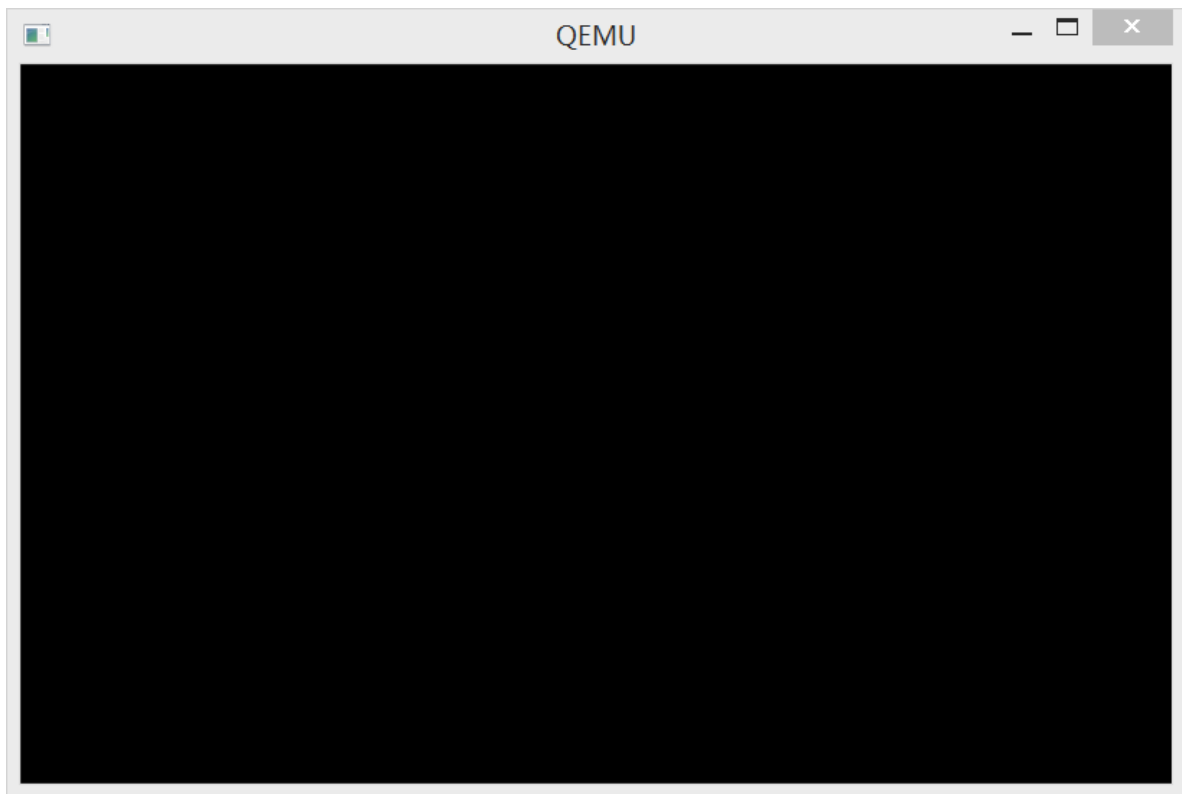
代码说明

关于显卡模式

1. AH = 0x00;
2. AL = 模式：（省略了一些不重要的画面模式）
 - 0x03: 16色字符模式，80×25
 - 0x12: VGA图形模式，640×480×4位彩色模式，独特的4面存储模式
 - 0x13: VGA图形模式，320×200×8位彩色模式，调色版模式
 - 0x6a: 扩展VGA图形模式，800×600×4位彩色模式，独特的4面存储模式（有的显卡不支持这个模式）
3. 返回值：无

上面的程序我们选择了0x13画面模式，因为8位彩色模式可以使用256中颜色，这一点就很友好了。还有就是 `ipl.nas` 更名为 `ipl10` 是因为有10个柱面。另外，想要把磁盘装载内容（ipl）的结束地址告诉给 `haribote.sys`，所以我们在“JMP 0xc200”之前，加入了一行命令，将CYLS的值写到内存地址0x0ff0中，至此，启动区完成，现在我们就用 `make run` 指令来运行一下我们的系统吧！

运行效果



画面是黑的！千万不要以为是程序出错了，这是正确的运行结果，因为我们的设定就是这样的。这里提一点，现在我们把 `haribote.sys` 中没有的关系的前后部分也读取进来了，这样的话，程序会在启动时很慢，但是在后面会起很大的作用，所以先暂时这样。

32位模式前期准备

接下来，我们将用C语言为主进行开发了，这是我们当前的目标。关于C编译器而言，我们选择32位的。这里32位是指的CPU的模式，有16位的，也有32为的，相比16位的，32位的有点比较多。但是也有一点，就是32为的无法调用BIOS功能。所以涉及到BIOS的事情，我们还是要提前做的。我们已经完成了画面的制作，接下来我们从BIOS中获取键盘状态。接下来的事情我们在 `harib00h` 中做。对 `haribote.nas` 进行修改。

`haribote.nas`

```
; haribote-os
; TAB=4

; 有关BOOT_INFO（启动信息）
CYLS    EQU    0x0ff0      ; 设定启动区
LEDS    EQU    0x0ff1
VMODE   EQU    0x0ff2      ; 关于颜色书目的信息。颜色的位数。
SCRNX   EQU    0x0ff4      ; 分辨率的X(screen x)
SCRNY   EQU    0x0ff6      ; 分辨率的Y(screen y)
VRAM    EQU    0x0ff8      ; 图像缓冲区的开始地址

        ORG     0xc200      ; 这个程序将要装载的地址。

        MOV     AL, 0x13     ; VGA显卡，320×200×8 位色彩。
        MOV     AH, 0x00
```

```

INT      0x10
MOV      BYTE [VMODE], 8    ; 记录画面模式
MOV      WORD [SCRNX], 320
MOV      WORD [SCRNY], 200
MOV      DWORD [VRAM], 0x000a0000

; 用BIOS取得键盘上各种LED指示灯的状态

MOV      AH, 0x02
INT      0x16                ; keyboard BIOS
MOV      [LEDS], AL

fin:
HLT
JMP      fin

```

代码说明

VRAM 保存的是0xa0000 ,VRAM 指的是显卡内存 (video RAM) ,也就是用来显示画面的内存。VRAM分布在内存分布图的好几个不同的地方。这是因为，不同的画面模式下的像素也不一样。这次的我们使用的VRAM的值是0xa0000，大家可能对这个值有疑问，其实在这种画面模式下“VRAM是0xa0000~0xfffff的64KB”

另外我们还把画面的像素数，颜色数，以及从BIOS中获得的键盘信息都保存起来。保存位置是在内存0x0ff0附近，从分布图上来看，这一块并没有使用，所以现在我们使用这块地址是没有问题的。

开始导入C语言

到这里，我们前期的准备工作算是正式结束，切换到32位模式下，用C语言写我们的操作系统。这次我们在 `harib00i` 下进行。这次我们更改的文件有：`harbote.sys`，并把其重新命名为 `asmhead.nas` 同时，我们这次创建了一个新的文件 `bootpack.c`。

asmhead.sys

```

; haribote-os boot asm
; TAB=4

BOTPAK   EQU      0x00280000    ; 加载bootpack
DSKCAC   EQU      0x00100000    ; 磁盘缓存的位置
DSKCAC0  EQU      0x00008000    ; 磁盘缓存的位置（实模式）

; BOOT_INFO相关
CYLS     EQU      0x0ff0        ; 引导扇区设置
LEDS     EQU      0x0ff1
VMODE    EQU      0x0ff2        ; 关于颜色的信息
SCRNX    EQU      0x0ff4        ; 分辨率X
SCRNY    EQU      0x0ff6        ; 分辨率Y
VRAM     EQU      0x0ff8        ; 图像缓冲区的起始地址

```

```

    ORG    0xc200          ; 这个的程序要被装载的内存地址

; 画面モードを設定

    MOV    AL, 0x13        ; VGA显卡, 320x200x8bit
    MOV    AH, 0x00
    INT    0x10
    MOV    BYTE [VMODE], 8 ; 屏幕的模式 (参考C语言的引用)
    MOV    WORD [SCRNX], 320
    MOV    WORD [SCRNY], 200
    MOV    DWORD [VRAM], 0x000a0000

; 通过BIOS获取指示灯状态

    MOV    AH, 0x02
    INT    0x16            ; keyboard BIOS
    MOV    [LEDS], AL

; 防止PIC接受所有中断
;   AT兼容机的规范、PIC初始化
;   然后之前在CLI不做什么事就挂起
;   PIC在同意后初始化

    MOV    AL, 0xff
    OUT    0x21, AL
    NOP                    ; 不断执行OUT指令
    OUT    0xa1, AL

    CLI                    ; 进一步中断CPU

; 让CPU支持1M以上内存、设置A20GATE

    CALL    waitkbdout
    MOV    AL, 0xd1
    OUT    0x64, AL
    CALL    waitkbdout
    MOV    AL, 0xdf        ; enable A20
    OUT    0x60, AL
    CALL    waitkbdout

; 保护模式转换

[INSTRSET "i486p"]        ; 说明使用486指令

    LGDT   [GDTR0]        ; 设置临时GDT
    MOV    EAX, CR0
    AND    EAX, 0x7fffffff ; 使用bit31 (禁用分页)
    OR     EAX, 0x00000001 ; bit0到1转换 (保护模式过渡)

```

```

        MOV     CR0,EAX
        JMP     pipelineflush
pipelineflush:
        MOV     AX,1*8           ; 写32bit的段
        MOV     DS,AX
        MOV     ES,AX
        MOV     FS,AX
        MOV     GS,AX
        MOV     SS,AX

; bootpack传递

        MOV     ESI,bootpack     ; 源
        MOV     EDI,BOTPAK      ; 目标
        MOV     ECX,512*1024/4
        CALL    memcpy

; 传输磁盘数据

; 从引导区开始

        MOV     ESI,0x7c00       ; 源
        MOV     EDI,DSKCAC       ; 目标
        MOV     ECX,512/4
        CALL    memcpy

; 剩余的全部

        MOV     ESI,DSKCAC0+512  ; 源
        MOV     EDI,DSKCAC+512  ; 目标
        MOV     ECX,0
        MOV     CL,BYTE [CYLS]
        IMUL    ECX,512*18*2/4   ; 除以4得到字节数
        SUB     ECX,512/4        ; IPL偏移量
        CALL    memcpy

; 由于还需要asmhead才能完成
; 完成其余的bootpack任务

; bootpack启动

        MOV     EBX,BOTPAK
        MOV     ECX,[EBX+16]
        ADD     ECX,3             ; ECX += 3;
        SHR     ECX,2            ; ECX /= 4;
        JZ      skip            ; 传输完成
        MOV     ESI,[EBX+20]     ; 源
        ADD     ESI,EBX
        MOV     EDI,[EBX+12]     ; 目标

```



```

CALL    memcpy
skip:
MOV     ESP, [EBX+12]    ; 堆栈的初始化
JMP     DWORD 2*8:0x0000001b

waitkbdout:
IN      AL, 0x64
AND     AL, 0x02
JNZ     waitkbdout      ; AND结果不为0跳转到waitkbdout
RET

memcpy:
MOV     EAX, [ESI]
ADD     ESI, 4
MOV     [EDI], EAX
ADD     EDI, 4
SUB     ECX, 1
JNZ     memcpy          ; 运算结果不为0跳转到memcpy
RET

; memcpy地址前缀大小

ALIGNB  16
GDT0:
RESB    8                ; 初始值
DW      0xffff, 0x0000, 0x9200, 0x00cf ; 写32bit位段寄存器
DW      0xffff, 0x0000, 0x9a28, 0x0047 ; 可执行的文件的32bit寄存器
      (bootpack用)

      DW      0

GDTR0:
DW      8*3-1
DD      GDT0

ALIGNB  16
bootpack:

```

bootpack.c

```

void HariMain(void)
{

fin:
    /* 这里想写上HLT,但是C语言中不能用HLT*/
    goto fin;

}

```

代码说明

在 `asmhead.nas` 中，写了很多东西，并且为了调用C语言写的程序，写了100行左右的汇编代码，这里先不做解释，后面会慢慢说明。再一个就是C语言部分，`bootpack.c` 名字的含义为一个“包”，因为以后我们还要写很多类似这样的程序，为了方便维护，我们把它打包。`goto` 是C语言中的语法，相当于JMP。

C程序 `bootpack.c` 转换成机器语言的步骤

- 首先，使用 `ccl.exe` 从 `bootpack.c` 生成 `bootpack.gas`。
- 第二步，使用 `gas2nask.exe` 从 `bootpack.gas` 生成 `bootpack.nas`。
- 第三步，使用 `nask.exe` 从 `bootpack.nas` 生成 `bootpack.obj`。
- 第四步，使用 `obj2bim.exe` 从 `bootpack.obj` 生成 `bootpack.bim`。
- 最后，使用 `bim2hrb.exe` 从 `bootpack.bim` 生成 `bootpack.hrb`。
- 经过以上几个步骤就做成了机器语言，再使用 `copy` 指令将 `asmhead.bin` 与 `bootpack.hrb` 单纯结合起来，就成了 `haribote.sys`。也就是我们的二进制文件，最终的机器语言。

工具介绍

- `ccl` 是作者从 `gcc` 改造来的C编译器，可以实现从C语言到汇编语言源程序（`gas` 程序），此时不能直接用 `nask` 进行编译。
- `gas2nask`，就是 `gas to nask` 的意思，把 `nas` 程序转换成 `nask` 能编译的 `nas` 程序。
- 一旦被转换成了，只要用 `nask` 编译一下，就能变成机器语言了。事实就是如此，首先用 `nask` 制作 `obj` 文件。`obj` 文件又称目标文件，程序是用C语言写的，而我们的目标是机器语言，所以这就是“目标文件”这一名称的由来。
- `obj2bim` 的作用是

大家是不是想着，既然已经做成了机器语言了，那只要把他们写进映像文件里就万事大吉了。但是遗憾的是，这还不行事实上这也是使用C语言的不方便之处。目标文件（`obj` 文件）是一种特殊的机器语言文件，必须与其他文件链接（`link`）后才能变成真正可以执行的机器语言。链接是什么意思呢？因为C语言的局限性，所以这个系统一部分需要用C语言编写，一部分需要用汇编编写，然后链接到C语言写的程序上。`bim` 是作者设计的一种文件类型，类似于“镜像文件”，所谓镜像文件，就是“不是本来的状态，而是一种代替的形式”也还不是完成品。只是将各个部分全部链接在一起，做成了一个完整的机器语言。但是却不是能在每个系统上运行的，需要根据不同的系统进行必要的“加工”，这次我们根据我们的“系统”特地写的一个程序 `bim2hrb.exe`，这个程序留到后面介绍。

以前我们写的C程序，最后直接可以编译成可执行性文件，但是我们的为什么就这么复杂呢？因为我们以前使用的编译器已经程序了，也生成了相似的“中间产物”。如果直接一步生成可执行文件的话，那么基本是一步到位了，也就没有了其他的是利用价值，所以，这一步步的生成程序的话，我们可以根据不同的操作系统可以制作适应不同操作系统的程序。这样对于我们接下来的开发是大有裨益的。我们同时也对 `Makefile` 文件进行改动，这里不再进行赘述。我们用 `make run` 运行，还是黑屏，说明程序正常。

实现HLT(harib00j)

尝试实现一下HTL，解决掉我们程序一直耗电的问题。这里我们从 `harib00j` 下开始。因为C语言是不支持HLT的，所以我们写在一个汇编程序中，然后再链接到C语言程序中。首先写 `naskfunc.nas`

naskfunc.nas

```
; naskfunc
; TAB=4

[FORMAT "WCOFF"]           ; 制作目标文件的模式
[BITS 32]                   ; 制作32位模式用的机械语言

; 制作目标文件的信息

[FILE "naskfunc.nas"]       ; 源文件名信息

        GLOBAL  _io_hlt     ; 程序中包含的函数名

; 以下是实际的函数

[SECTION .text]              ; 目标文件中写了这些之后再写程序

_io_hlt:    ; void io_hlt(void);
        HLT
        RET
```

代码说明

上面的函数我们用汇编语言写了一个函数。函数的名字叫 `io_hlt`。因为在CPU中HLT也属于I/O指令，所以我们的函数名字就这样起了。

用汇编写的函数，之后要用与C程序编译生成的 `bootpack.obj` 链接，所以也需要编译成目标文件。因此将输出格式设定为WCOFF模式。另外，还要设定成32位机器语言模式。

在nask目标文件的模式下，**必须设定文件名信息**然后再写明下面程序的函数名。注意要在函数名的前面加上 `_`，否则就不能很好的和C语言衔接。**需要链接的函数名，都要用GLOBAL指令声明。**

实际函数部分，**先写一个与GLOBAL声明的函数名相同的标号**，然后写代码即可。其中 `RET` 指令相当于C语言中的 `return`。下面再看一下C语言中代码。

bootpack.c

```
/* 告诉编译器,有一个函数在别的文件里 */

void io_hlt(void);
```



```

_write_mem8:      ; void write_mem8(int addr, int data);
    MOV     ECX, [ESP+4]      ; [ESP+4]中存放的是地址，将其读入ECX
    MOV     AL, [ESP+8]      ; [ESP+8]中存放的是数据，将其读入AL
    MOV     [ECX], AL
    RET

```

代码说明

我们新写的函数 `_write_mem8` 我们可以看到，如果给这个函数一个循环的话，实现的就是[ESP + 4]的操作，详细如下：

```

第一个数字的存放地址:[ESP + 4]
第二个数字的存放地址:[ESP + 8]
第三个数字的存放地址:[ESP + 12]
...

```

如果C语言联合使用的话，有的寄存器能自由使用，有的寄存器不能自由使用，能自由使用的寄存器只有EAX,ECX,EDX这3个。其他的只能使用其值，而不能改变其值。

程序中新增加的一行 `INSTSET` 指令，用来说明适用的CPU(英特尔系列)是给486型号使用的，但不是说只能486使用，为什么写这个呢?因为如果不写这个的话，因为汇编语言可能会把某些指令识别成标签，比如这个程序中的EAX。

接下来是C语言部分

bootpack.c

```

/* 告诉编译器,有一个函数在别的文件里 */
void io_hlt(void);
void write_mem8(int addr,int data);
/* 是函数声明，却没有函数体，这表示的意思是：函数在别的文件里，你这个编译器自己去找啊！
*/

void HariMain(void)
{
    int i;//声明变量: i是一个32位整数

    for(i=0xa0000;i<=0xffff;i++)
    {
        write_mem8(i,15);//VRAM全部写入15
    }
    for(;;)
    {
        io_hlt();
    }
}

```

make run 一下，发现屏幕变成白色了。其中,0xa0000~0xffff 是VRAM的存储空间地址范围，全部写入15，也就是全部像素的颜色都是第15种颜色，而第15种颜色是白色。所以画面就变成了白色了。



条纹图案

这次我们让他出现条纹图案。在 `hari01b` 中进行，这次我们只需稍微修改一下 `bootpack.c` 就行了

`bootpack.c`

```
/* 告诉编译器,有一个函数在别的文件里 */
void io_hlt(void);
void write_mem8(int addr,int data);
/* 是函数声明,却没有函数体,这表示的意思是:函数在别的文件里,你这个编译器自己去找啊! */

void HariMain(void)
{
    int i;//声明变量: i是一个32位整数

    for(i=0xa0000;i<=0xffff;i++)
    {
        write_mem8(i, i & 0x0f);//进行“与”运算
    }
    for(;;)
    {
        io_hlt();
    }
}
```

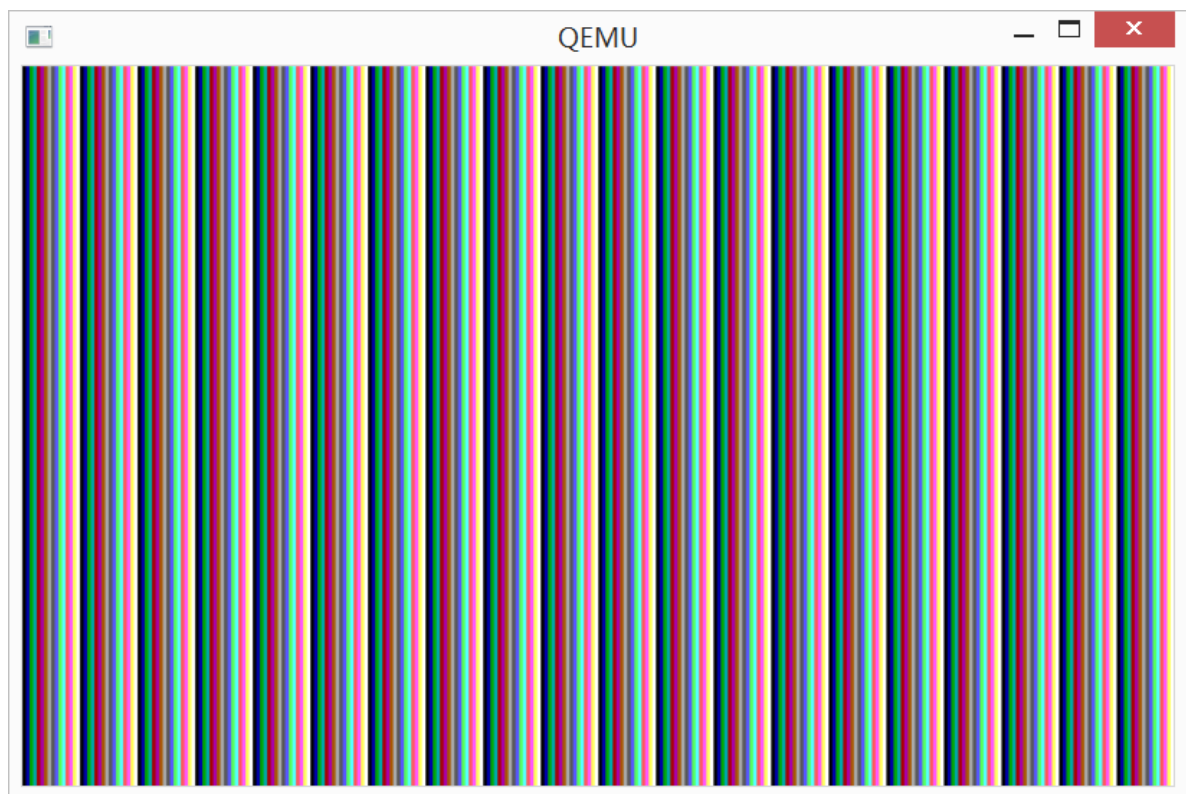
```
}
```

代码说明

这次的程序中，我们在往VRAM中写内容的时候，进行了“与”运算。也就是全部取反的意思，`0x0f` 转化成二进制是 `1111`，全是1，所以取反后全是0，低4位保持不变，而高四位不再随这循环的增加而增加，其实本质是增加的，但是进行了“与”运算之后，高四位就变成了0。故16进制中，写入到VRAM中的值变成了：

```
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 00 01 02 03 04 05 06....
```

这样，每隔16个像素色号就反复一次。这时候我们make run 一下看看，就会出现“条纹图案”。



挑战指针

上面的工作，在向VRAM的指定地址写内容时候，我们是先在汇编语言中写一个函数，然后C语言再进行调用。这次我们用C语言直接实现这一个目的。这次我们在 `harib01c` 中进行。并对 `bootpack.c` 进行了修改。

`bootpack.c`

```
/* 告诉编译器,有一个函数在别的文件里 */
void io_hlt(void);
/* 是函数声明,却没有函数体,这表示的意思是:函数在别的文件里,你这个编译器自己去找啊! */
void HariMain(void)
{
    int i; // 声明变量: i是一个32位整数
```

```

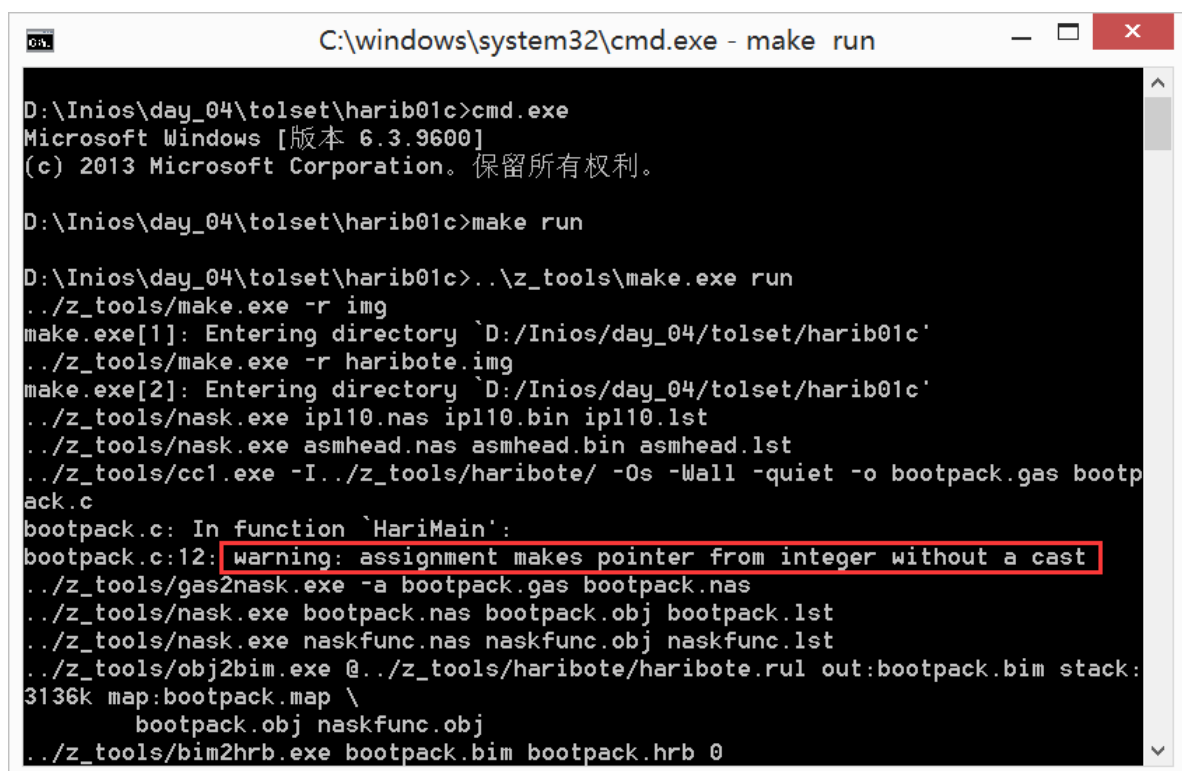
char * p; //指针变量,用来存放BYTE型地址

for(i=0xa0000;i<=0xffff;i++)
{
    //替代了write_mem8(i, i & 0x0f);
    p = i; //带入地址
    *p = i & 0x0f;

}
for(;;)
{
    io_hlt();
}
}

```

同样的，因为这样已经可以替代了我们原来在 `naskfunc.nas` 中写的函数了，所以把其中的汇编函数去就好了，同样的是在C语言程序中，我们也不需要再声明那个汇编程序了。但是仔细看，会有一个警告。



```

C:\windows\system32\cmd.exe - make run

D:\Inios\day_04\tolset\harib01c>cmd.exe
Microsoft Windows [版本 6.3.9600]
(c) 2013 Microsoft Corporation。保留所有权利。

D:\Inios\day_04\tolset\harib01c>make run

D:\Inios\day_04\tolset\harib01c>../z_tools/make.exe run
../z_tools/make.exe -r img
make.exe[1]: Entering directory `D:/Inios/day_04/tolset/harib01c'
../z_tools/make.exe -r haribote.img
make.exe[2]: Entering directory `D:/Inios/day_04/tolset/harib01c'
../z_tools/nask.exe ipl10.nas ipl10.bin ipl10.lst
../z_tools/nask.exe asmhead.nas asmhead.bin asmhead.lst
../z_tools/cc1.exe -I../z_tools/haribote/ -Os -Wall -quiet -o bootpack.gas bootp
ack.c
bootpack.c: In function `HariMain':
bootpack.c:12: warning: assignment makes pointer from integer without a cast
../z_tools/gas2nask.exe -a bootpack.gas bootpack.nas
../z_tools/nask.exe bootpack.nas bootpack.obj bootpack.lst
../z_tools/nask.exe naskfunc.nas naskfunc.obj naskfunc.lst
../z_tools/obj2bim.exe @../z_tools/haribote/haribote.rul out:bootpack.bim stack:
3136k map:bootpack.map \
    bootpack.obj naskfunc.obj
../z_tools/bim2hrb.exe bootpack.bim bootpack.hrb 0

```

这个警告的意思是“赋值语句没有经过类型转换，由整数生成了指针”我们只需把其中的变量 `i` 的数据类型转化一下就可以了，因为我们的指针 `p` 指向的地址寸的是 `char` 类型的数据，因为我们这次是一个字节字节的写入，所以使用了 `char`。且有如下对应关系：

```

char * p;    //用于BYTE类地址
short * p;   //用于WORD类地址
int * p;     //用于DWORD类地址

```


多提一点，“char i”是类似AL的1字节变量，“short i”是类似AX的2字节变量，“int i”是类似EAX的4字节变量。但是不管是“char * p”，还是“short * p”，还是“int * p”，变量p都是4字节。这是因为p是用于记录地址的变量。在汇编语言中，地址也像ECX一样，用4字节的寄存器来指定，所以也是4字节。

至此，我们实现了只有C语言写入内存的功能。

指针应用

绘制条纹图案的C语言代码还可以写成下面的这种形式,为了和以前的代码加以区分，我们在 `harib01d` 中进行。

```
/* 告诉编译器,有一个函数在别的文件里 */
void io_hlt(void);
/* 是函数声明，却没有函数体，这表示的意思是：函数在别的文件里，你这个编译器自己去找啊！ */
void HariMain(void)
{
    int i; //声明变量: i是一个32位整数
    char * p; //指针变量,用来存放BYTE型地址
    p = (char *) 0xa0000; //指针变量赋值
    for(i=0;i<=0xffff;i++)
    {
        *(p+i) = i & 0x0f;

    }
    for(;;)
    {
        io_hlt();
    }
}
```

实现的效果和前面实现的是一样的，这里只是展示一种C语言的另外的一种写法。接下来我们再用另外一种写法来实现。在 `harib01e` 中进行。C语言中，`*(p+i)` 还可以改写成`p[i]`这种形式，所以以上片段也可以写成这样：

```
/* 告诉编译器,有一个函数在别的文件里 */
void io_hlt(void);
/* 是函数声明，却没有函数体，这表示的意思是：函数在别的文件里，你这个编译器自己去找啊！ */
void HariMain(void)
{
    int i; //声明变量: i是一个32位整数
    char * p; //指针变量,用来存放BYTE型地址
    p = (char *) 0xa0000; //指针变量赋值
    for(i=0;i<=0xffff;i++)
    {
        p[i] = i & 0x0f;
    }
}
```

```

    }
    for(;;)
    {
        io_hlt();
    }
}

```

色号设定

前面我们写的操作系统的画面颜色是8位（二进制）数。也就是只能使用0~255的数，颜色太少了，这次我们就写一个6位十六进制数，也就是24位（二进制）数来指定颜色，为什么说是24位二进制呢，因为一个16进制数可以看作是4个二进制数一组组成4组。这样我们的颜色就更加丰富了，也就是RGB方式。接下来的工作我们在 `harb01f` 中进行。

关于调色板的概念,以8位彩色模式为例，由程序员随意指定0~255的数字对应的颜色，比如说25号颜色对应#ffffff,26号颜色对应#123456等。我们把这种方式叫做 **调色板 (palette)**。

这次我们用下面16中颜色就好了，且分别标号0-15。

#000000:黑	#00ffff:浅亮蓝	#000084:暗蓝
#ff0000:亮红	#ffffff:白	#840084:暗紫
#00ff00:亮绿	#c6c6c6:亮灰	#008484:浅暗蓝
#ffff00:亮黄	#840000:暗红	#848484:暗灰
#0000ff:亮蓝	#008400:暗绿	
#ff00ff:亮紫	#848400:暗黄	

这次对 `bootpack.c` 写了很多代码

bootpack.c

```

void io_hlt(void);
void io_cli(void);
void io_out8(int port, int data);
int io_load_eflags(void);
void io_store_eflags(int eflags);

/* 上面是函数声明部分 */

void init_palette(void);
void set_palette(int start, int end, unsigned char *rgb);

void HariMain(void)
{
    int i; /* 声明变量。变量i是32位整数型 */
    char *p; /* 指针变量p是BYTE[...]用到的地址*/

```

```

init_palette(); /* 设定调色板*/

p = (char *) 0xa0000; /* 番地を代入 */

for (i = 0; i <= 0xffff; i++) {
    p[i] = i & 0x0f;
}

for (;;) {
    io_hlt();
}
}

void init_palette(void)
{
    static unsigned char table_rgb[16 * 3] = { //声明常量
        0x00, 0x00, 0x00, /* 0:黑 */
        0xff, 0x00, 0x00, /* 1:亮红 */
        0x00, 0xff, 0x00, /* 2:亮绿 */
        0xff, 0xff, 0x00, /* 3:亮黄 */
        0x00, 0x00, 0xff, /* 4:亮蓝 */
        0xff, 0x00, 0xff, /* 5:亮紫 */
        0x00, 0xff, 0xff, /* 6:浅亮蓝 */
        0xff, 0xff, 0xff, /* 7:白 */
        0xc6, 0xc6, 0xc6, /* 8:亮灰 */
        0x84, 0x00, 0x00, /* 9:暗红 */
        0x00, 0x84, 0x00, /* 10:暗绿 */
        0x84, 0x84, 0x00, /* 11:暗黄 */
        0x00, 0x00, 0x84, /* 12:暗青 */
        0x84, 0x00, 0x84, /* 13:暗紫 */
        0x00, 0x84, 0x84, /* 14:浅暗蓝 */
        0x84, 0x84, 0x84 /* 15:暗灰 */
    };
    set_palette(0, 15, table_rgb);
    return;

    /* C语言中static char 语句只能用于数据,相当于汇编语言中的DB指令 */
}

void set_palette(int start, int end, unsigned char *rgb)
{
    int i, eflags;
    eflags = io_load_eflags(); /* 记录中断许可标志的值 */
    io_cli(); /* 将中断许可标志置为0,禁止中断 */
    io_out8(0x03c8, start);
    for (i = start; i <= end; i++) {
        io_out8(0x03c9, rgb[i] / 4); //颜色值右移两位,这样的结果除了使颜色更亮一些之外,暂时没有发现还有其他的什么用处
    }
}

```

```

        io_out8(0x03c9, rgb[1] / 4);
        io_out8(0x03c9, rgb[2] / 4);
        rgb += 3;
    }
    io_store_eflags(eflags);    /* 复原中断许可标志 */
    return;
}

```

代码说明01

- `init_paette` 函数开头的 `static` 可以理解成“静态”的方法。这样的话比不加 `static` 的话占用的内存会少。所以这里我们加上 `static`。还有一个是 `unsigned`，它的意思是：这里要处理的数据是BYTE (char) 型，但它是没有符号 (sign) 的数 (0或者正整数)。char型的变量有3种模式，分别是signed型，unsigned型，和未指定型。signed型用于处理-128~127的整数。未指定型是指没有特别指定时，可由编译器决定是unsigned还是signed。因为在程序中我们用到了0xff，也就是255，用来表示最大亮度，但有时编译器会认为其是-1(不多做解释)，所以这里咱们用到了这个 `unsigned`。
- `set_palette` 多次调用 `io_out8` 函数，这个 `io_out8` 是往指定装置中传送数据的。详细的待会再说。
- CPU的管脚和内存相连，但同时呢也能和设备相连，对于不同的设备作出不同的反应，为了出错，每个设备都有着自己固定的id，也就是 `设备号`，CPU通过向设备发送电信号来操作。
- 为了能够让调用我们前面提到过的 `调色板`，需要我们安如下步骤来：
 - 首先在一连串的访问中屏蔽中断 (比如CLI)。
 - 将想要设定的调色板号码写入0x03c8端口，紧接着，按RGB的顺序写入0x03c9。若还想继续设定下个调色板，就省略调色板的号码，再按RGB的顺序写入0x03c9就行了。
 - 若想读出当前调色板的状态，首先要将调色板的号码写入0x03c7，再从0x03c9中读取3次，顺序为RGB。若要继续读下一个，则省略调色板号码设定，继续按RGB读出。
 - 若开始执行了CLI，则最后执行STI。

在“调色板的访问步骤中” `CLI` 是将中断标志设置为0的指令；`STI` 是将中断标志设置为1的指令。因为CPU的中断处理的关系。中断的知识，这里不再展开讲解。

`EFLAGS` 是一个特点的寄存器，是存储进位标志和中断标志的寄存器。进位标志可以用 `JC` 或者 `JNC` 来简单的判断到底是0还是1。但是对于中断标志，只能用 `EFLAGS` 再检查第9位是0还是1。

- `io_load_eflags` 读取最初的值，也就是相当于“先记住中断标志”。
- `io_store_eflags` 再判断 `eflags` 的内容，看是否执行。

接下来我们再说下这次写的 `naskfunc.nas`

`naskfunc.nas`

```

; naskfunc
; TAB=4

[FORMAT "WCOFF"]           ; 制作目标文件的模式
[INSTRSET "i486p"]         ; 说明使用的是486命令
[BITS 32]                  ; 制作32位模式用的机械语言
; 制作目标文件的信息
[FILE "naskfunc.nas"]      ; 源文件名信息

GLOBAL _io_hlt, _io_cli, _io_sti, _io_stihlt ; 程序中包含的函数名

GLOBAL _io_in8, _io_in16, _io_in32
GLOBAL _io_out8, _io_out16, _io_out32
GLOBAL _io_load_eflags, _io_store_eflags

; 以下是实际的函数
[SECTION .text]            ; 目标文件中写了这些之后再写程序

_io_hlt:                   ; void io_hlt(void);
    HLT
    RET

_io_cli:                   ; void io_cli(void);
    CLI
    RET

_io_sti:                   ; void io_sti(void);
    STI
    RET

_io_stihlt:                ; void io_stihlt(void);
    STI
    HLT
    RET

_io_in8:                   ; int io_in8(int port);
    MOV     EDX, [ESP+4]    ; port
    MOV     EAX, 0
    IN      AL, DX
    RET

_io_in16:                  ; int io_in16(int port);
    MOV     EDX, [ESP+4]    ; port
    MOV     EAX, 0
    IN      AX, DX
    RET

_io_in32:                  ; int io_in32(int port);
    MOV     EDX, [ESP+4]    ; port

```

```

        IN        EAX,DX
        RET

_io_out8:    ; void io_out8(int port, int data);
        MOV      EDX,[ESP+4]    ; port
        MOV      AL,[ESP+8]    ; data
        OUT      DX,AL
        RET

_io_out16:   ; void io_out16(int port, int data);
        MOV      EDX,[ESP+4]    ; port
        MOV      EAX,[ESP+8]   ; data
        OUT      DX,AX
        RET

_io_out32:   ; void io_out32(int port, int data);
        MOV      EDX,[ESP+4]    ; port
        MOV      EAX,[ESP+8]   ; data
        OUT      DX,EAX
        RET

_io_load_eflags:    ; int io_load_eflags(void);
        PUSHFD      ; 指 PUSH EFLAGS
        POP         EAX
        RET

_io_store_eflags:   ; void io_store_eflags(int eflags);
        MOV         EAX,[ESP+4]
        PUSH        EAX
        POPFD       ; 指 POP EFLAGS
        RET

```

代码说明02

现在再说一下 `EFLAGS` 中的 `PUSHFD` 和 `POPFD` 指令。

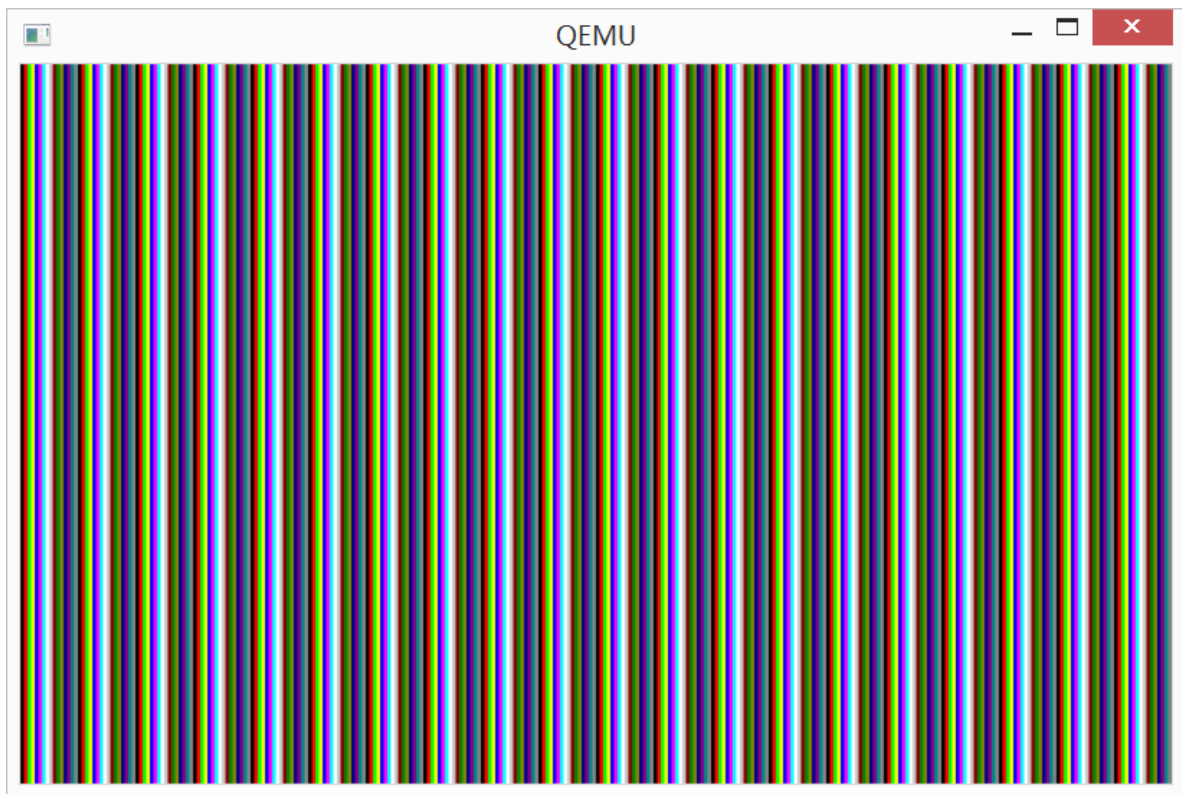
- `PUSHFD` 是“push flags double-word”的缩写，意思是将标志位的值按双字长压入栈。
- `POPFD` 是“pop flags double-word”的缩写，意思是按双字长将标志位从栈中弹出。
`ESP` 是栈指针寄存器，在一开始是有提及的。

也就是说因为 `EFLAGS` 是不能进行“`MOV EAX,EFLAGS`”的，它没有这个操作，所以用“`PUSHFD POP EAX`”代替，其意义是先将 `EFLAGS` 压入栈，再将弹出的值代入 `EAX`。同理，“`PUSH EAX POPFD`”相当于“`MOVE EFLAGS,EAX`”。

还有几个函数这里没做介绍，因为后面还会用到，以后再说。

运行效果

条纹的图案没有变化，但是颜色变了哦！



绘制矩形

颜色已经准备就绪，现在开始画画吧！在当前的画面模式中，有320×240个像素点。假设左上点的坐标是（0，0），右下点的左坐标（319，319），那么像素点坐标（x,y）对应的VRAM地址应按下式子计算。

```
0xa0000 + x + y * 320
```

其他画面模式基本相同，只是起始地址0xa0000和y的系数320有些不同。我们这次是要画矩形的，基本思路是先画一个点，然后增加x坐标，成为一条线，然后增加y值，让线成面，最终成为矩形。并制作了函数 `boxfill8`。我们这次在 `harib01g` 中进行。这次的 `bootpack.c` 如下：

`bootpack.c`

```
void io_hlt(void);
void io_cli(void);
void io_out8(int port, int data);
int io_load_eflags(void);
void io_store_eflags(int eflags);

void init_palette(void);
void set_palette(int start, int end, unsigned char *rgb);
void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int
y0, int x1, int y1);
/* 上面是函数声明部分 */
#define COL8_000000 0
#define COL8_FF0000 1
#define COL8_00FF00 2
```

```

#define COL8_FFFF00      3
#define COL8_0000FF      4
#define COL8_FF00FF      5
#define COL8_00FFFF      6
#define COL8_FFFFFFFF    7
#define COL8_C6C6C6      8
#define COL8_840000      9
#define COL8_008400     10
#define COL8_848400     11
#define COL8_000084     12
#define COL8_840084     13
#define COL8_008484     14
#define COL8_848484     15

void HariMain(void)
{
    int i; /* 声明变量。变量i是32位整数型 */
    char *p; /* 指针变量p是BYTE[...]用到的地址*/

    init_palette(); /* 设定调色板*/

    p = (char *) 0xa0000; /* 将地址赋值进去 */

    boxfill8(p, 320, COL8_FF0000, 20, 20, 120, 120);
    boxfill8(p, 320, COL8_00FF00, 70, 50, 170, 150);
    boxfill8(p, 320, COL8_0000FF, 120, 80, 220, 180);

    for (;;) {
        io_hlt();
    }
}

void init_palette(void)
{
    static unsigned char table_rgb[16 * 3] = {
        0x00, 0x00, 0x00, /* 0:黑 */
        0xff, 0x00, 0x00, /* 1:亮红 */
        0x00, 0xff, 0x00, /* 2:亮绿 */
        0xff, 0xff, 0x00, /* 3:亮黄 */
        0x00, 0x00, 0xff, /* 4:亮蓝 */
        0xff, 0x00, 0xff, /* 5:亮紫 */
        0x00, 0xff, 0xff, /* 6:浅亮蓝 */
        0xff, 0xff, 0xff, /* 7:白 */
        0xc6, 0xc6, 0xc6, /* 8:亮灰 */
        0x84, 0x00, 0x00, /* 9:暗红 */
        0x00, 0x84, 0x00, /* 10:暗绿 */
        0x84, 0x84, 0x00, /* 11:暗黄 */
        0x00, 0x00, 0x84, /* 12:暗青 */
        0x84, 0x00, 0x84, /* 13:暗紫 */

```



```

        0x00, 0x84, 0x84,    /* 14:浅暗蓝 */
        0x84, 0x84, 0x84    /* 15:暗灰 */
    };
    set_palette(0, 15, table_rgb);
    return;

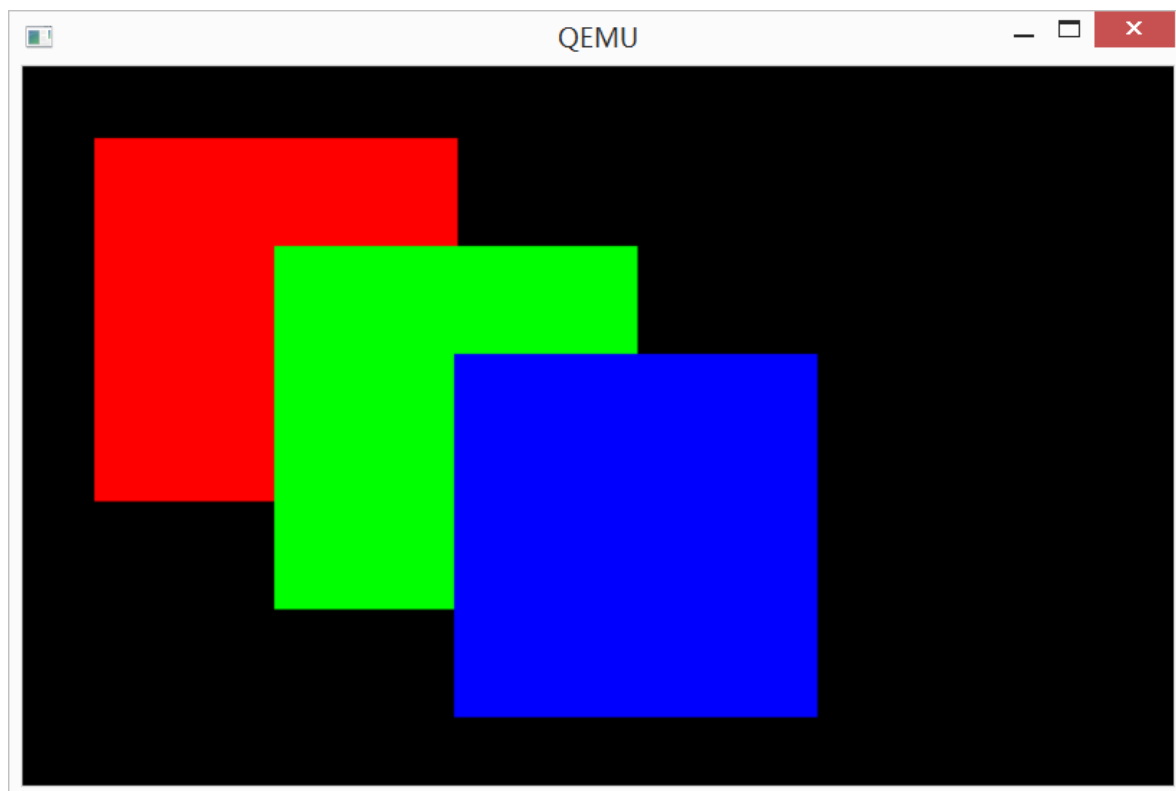
    /* C语言中static char 语句只能用于数据,相当于汇编语言中的DB指令 */
}

void set_palette(int start, int end, unsigned char *rgb)
{
    int i, eflags;
    eflags = io_load_eflags(); /* 记录中断许可标志的值 */
    io_cli();                  /* 将中断许可标志置为0,禁止中断 */
    io_out8(0x03c8, start);
    for (i = start; i <= end; i++) {
        io_out8(0x03c9, rgb[0] / 4); //颜色值右移两位,这样的结果除了使颜色更亮一
        些之外,暂时没有发现还有其他的什么用处
        io_out8(0x03c9, rgb[1] / 4);
        io_out8(0x03c9, rgb[2] / 4);
        rgb += 3;
    }
    io_store_eflags(eflags); /* 复原中断许可标志 */
    return;
}

void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int
y0, int x1, int y1)
{
    int x, y;
    for (y = y0; y <= y1; y++) {
        for (x = x0; x <= x1; x++)
            vram[y * xsize + x] = c;
    }
    return;
}

```

运行结果



成果

我们已经绘制出了矩形，这样简单的绘制一个“窗口”也就没什么了。这次我们在 `harib01h` 中进行。只需修改一下 `HariMain` 函数即可。

`bootpack.c`

```
void io_hlt(void);
void io_cli(void);
void io_out8(int port, int data);
int io_load_eflags(void);
void io_store_eflags(int eflags);

void init_palette(void);
void set_palette(int start, int end, unsigned char *rgb);
void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int
y0, int x1, int y1);
/* 上面是函数声明部分 */
#define COL8_000000    0
#define COL8_FF0000    1
#define COL8_00FF00    2
#define COL8_FFFF00    3
#define COL8_0000FF    4
#define COL8_FF00FF    5
#define COL8_00FFFF    6
#define COL8_FFFFFFFF  7
#define COL8_C6C6C6    8
#define COL8_840000    9
#define COL8_008400   10
#define COL8_848400   11
```

```

#define COL8_000084      12
#define COL8_840084      13
#define COL8_008484      14
#define COL8_848484      15

void HariMain(void)
{
    char *vram; /* 指针变量vram是BYTE[...]用到的地址*/
    int xsize, ysize;; /* 声明变量。变量是32位整数型 */

    init_palette(); /* 设定调色板*/

    vram = (char *) 0xa0000; /* 将地址赋值进去 */
    xsize = 320;
    ysize = 200;

    boxfill8(vram, xsize, COL8_008484, 0, 0, xsize - 1,
ysize - 29);
    boxfill8(vram, xsize, COL8_C6C6C6, 0, ysize - 28, xsize - 1,
ysize - 28);
    boxfill8(vram, xsize, COL8_FFFFFFFF, 0, ysize - 27, xsize - 1,
ysize - 27);
    boxfill8(vram, xsize, COL8_C6C6C6, 0, ysize - 26, xsize - 1,
ysize - 1);

    boxfill8(vram, xsize, COL8_FFFFFFFF, 3, ysize - 24, 59,
ysize - 24);
    boxfill8(vram, xsize, COL8_FFFFFFFF, 2, ysize - 24, 2,
ysize - 4);
    boxfill8(vram, xsize, COL8_848484, 3, ysize - 4, 59,
ysize - 4);
    boxfill8(vram, xsize, COL8_848484, 59, ysize - 23, 59,
ysize - 5);
    boxfill8(vram, xsize, COL8_000000, 2, ysize - 3, 59,
ysize - 3);
    boxfill8(vram, xsize, COL8_000000, 60, ysize - 24, 60,
ysize - 3);

    boxfill8(vram, xsize, COL8_848484, xsize - 47, ysize - 24, xsize - 4,
ysize - 24);
    boxfill8(vram, xsize, COL8_848484, xsize - 47, ysize - 23, xsize - 47,
ysize - 4);
    boxfill8(vram, xsize, COL8_FFFFFFFF, xsize - 47, ysize - 3, xsize - 4,
ysize - 3);
    boxfill8(vram, xsize, COL8_FFFFFFFF, xsize - 3, ysize - 24, xsize - 3,
ysize - 3);

    for (;;) {
        io_hlt();
    }
}

```

```

    }
}

void init_palette(void)
{
    static unsigned char table_rgb[16 * 3] = {
        0x00, 0x00, 0x00,    /* 0:黑 */
        0xff, 0x00, 0x00,    /* 1:亮红 */
        0x00, 0xff, 0x00,    /* 2:亮绿 */
        0xff, 0xff, 0x00,    /* 3:亮黄 */
        0x00, 0x00, 0xff,    /* 4:亮蓝 */
        0xff, 0x00, 0xff,    /* 5:亮紫 */
        0x00, 0xff, 0xff,    /* 6:浅亮蓝 */
        0xff, 0xff, 0xff,    /* 7:白 */
        0xc6, 0xc6, 0xc6,    /* 8:亮灰 */
        0x84, 0x00, 0x00,    /* 9:暗红 */
        0x00, 0x84, 0x00,    /* 10:暗绿 */
        0x84, 0x84, 0x00,    /* 11:暗黄 */
        0x00, 0x00, 0x84,    /* 12:暗青 */
        0x84, 0x00, 0x84,    /* 13:暗紫 */
        0x00, 0x84, 0x84,    /* 14:浅暗蓝 */
        0x84, 0x84, 0x84    /* 15:暗灰 */
    };
    set_palette(0, 15, table_rgb);
    return;

    /* C语言中static char 语句只能用于数据,相当于汇编语言中的DB指令 */
}

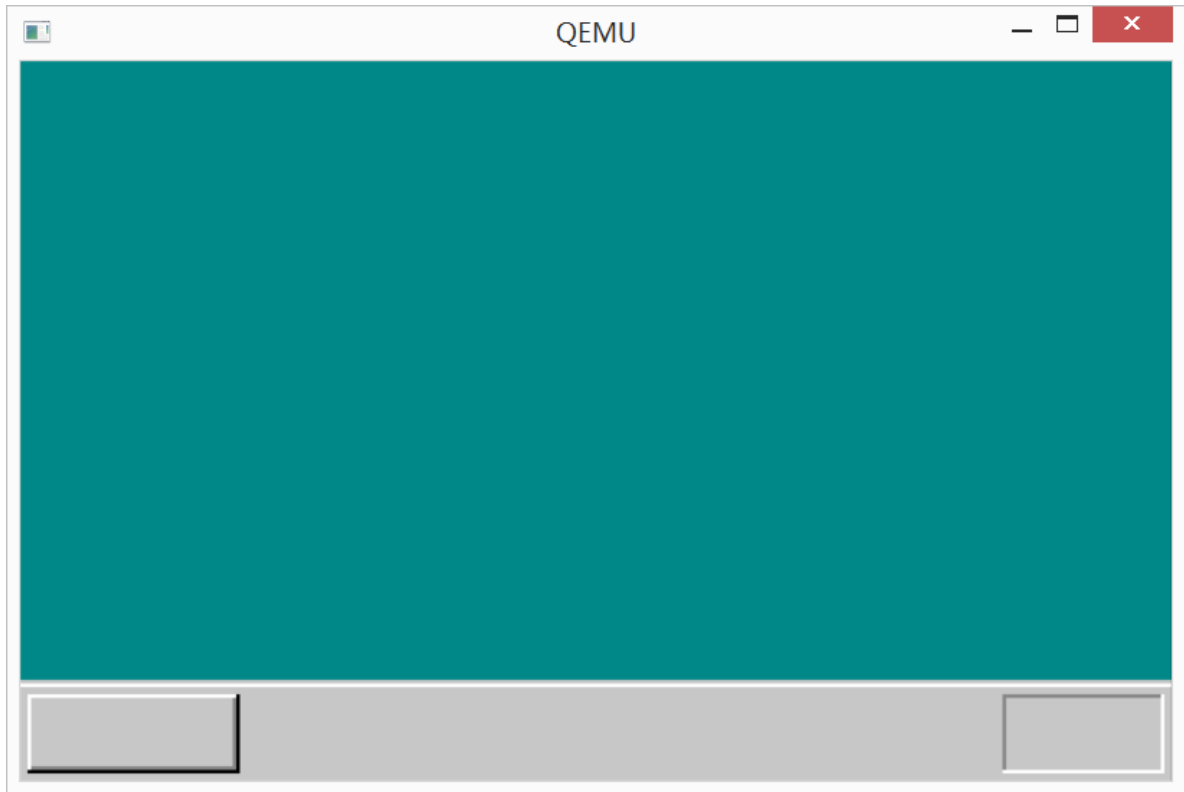
void set_palette(int start, int end, unsigned char *rgb)
{
    int i, eflags;
    eflags = io_load_eflags(); /* 记录中断许可标志的值 */
    io_cli();                  /* 将中断许可标志置为0,禁止中断 */
    io_out8(0x03c8, start);
    for (i = start; i <= end; i++) {
        io_out8(0x03c9, rgb[0] / 4); //颜色值右移两位,这样的结果除了使颜色更亮一些之外,暂时没有发现还有其他的什么用处
        io_out8(0x03c9, rgb[1] / 4);
        io_out8(0x03c9, rgb[2] / 4);
        rgb += 3;
    }
    io_store_eflags(eflags); /* 复原中断许可标志 */
    return;
}

void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int y0, int x1, int y1)
{
    int x, y;

```

```
for (y = y0; y <= y1; y++) {
    for (x = x0; x <= x1; x++)
        vram[y * xsize + x] = c;
}
return;
}
```

运行效果



到现在为止，我们的系统 `haribote.sys` 是1216字节，不到1.2KB。

第5天

接收启动信息

所谓“接受启动信息”就是在用在 `asmhead.nas` 中的信息获取，吧其中的信息提取出来，而不是直接写在C语言里。上次的 `bootpack.c` 中的0xa0000这些类似的，都可以写在 `asmhead.nas` 中。然后再在C语言中根据地址，利用指针获取到这个地址里的值。我们以前写的 `bootpack.c` 是吧“内容”给写死了，这样不便后期的维护与制作，这次我们直接根据地址调用，就灵活了很多。接下来的内容我们在 `harib02a` 中进行。并对 `bootpack.c` 进行更改。

bootpack.c

```
void io_hlt(void);
void io_cli(void);
void io_out8(int port, int data);
int io_load_eflags(void);
```

```

void io_store_eflags(int eflags);

void init_palette(void);
void set_palette(int start, int end, unsigned char *rgb);
void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int
y0, int x1, int y1);
void init_screen(char *vram, int x, int y);

/* 上面是函数声明部分 */
#define COL8_000000      0
#define COL8_FF0000      1
#define COL8_00FF00      2
#define COL8_FFFF00      3
#define COL8_0000FF      4
#define COL8_FF00FF      5
#define COL8_00FFFF      6
#define COL8_FFFFFFFF     7
#define COL8_C6C6C6      8
#define COL8_840000      9
#define COL8_008400     10
#define COL8_848400     11
#define COL8_000084     12
#define COL8_840084     13
#define COL8_008484     14
#define COL8_848484     15

void HariMain(void)
{
    char *vram; /* 指针变量vram是BYTE[...]用到的地址*/
    int xsize, ysize;; /* 声明变量。变量是32位整数型 */
    short *binfo_scrnx, *binfo_scrny;
    int *binfo_vram;

    init_palette(); /* 设定调色板*/
    binfo_scrnx = (short *) 0x0ff4; /*将地址赋值进去*/
    binfo_scrny = (short *) 0x0ff6;
    binfo_vram = (int *) 0x0ff8;
    xsize = *binfo_scrnx;
    ysize = *binfo_scrny;
    vram = (char *) *binfo_vram;

    init_screen(vram, xsize, ysize);

    for (;;) {
        io_hlt();
    }
}

```

```

void init_palette(void)
{
    static unsigned char table_rgb[16 * 3] = {
        0x00, 0x00, 0x00,    /* 0:黑 */
        0xff, 0x00, 0x00,    /* 1:亮红 */
        0x00, 0xff, 0x00,    /* 2:亮绿 */
        0xff, 0xff, 0x00,    /* 3:亮黄 */
        0x00, 0x00, 0xff,    /* 4:亮蓝 */
        0xff, 0x00, 0xff,    /* 5:亮紫 */
        0x00, 0xff, 0xff,    /* 6:浅亮蓝 */
        0xff, 0xff, 0xff,    /* 7:白 */
        0xc6, 0xc6, 0xc6,    /* 8:亮灰 */
        0x84, 0x00, 0x00,    /* 9:暗红 */
        0x00, 0x84, 0x00,    /* 10:暗绿 */
        0x84, 0x84, 0x00,    /* 11:暗黄 */
        0x00, 0x00, 0x84,    /* 12:暗青 */
        0x84, 0x00, 0x84,    /* 13:暗紫 */
        0x00, 0x84, 0x84,    /* 14:浅暗蓝 */
        0x84, 0x84, 0x84    /* 15:暗灰 */
    };
    set_palette(0, 15, table_rgb);
    return;

    /* C语言中static char 语句只能用于数据,相当于汇编语言中的DB指令 */
}

void set_palette(int start, int end, unsigned char *rgb)
{
    int i, eflags;
    eflags = io_load_eflags(); /* 记录中断许可标志的值 */
    io_cli();                  /* 将中断许可标志置为0,禁止中断 */
    io_out8(0x03c8, start);
    for (i = start; i <= end; i++) {
        io_out8(0x03c9, rgb[0] / 4); //颜色值右移两位,这样的结果除了使颜色更亮一
        些之外,暂时没有发现还有其他的什么用处
        io_out8(0x03c9, rgb[1] / 4);
        io_out8(0x03c9, rgb[2] / 4);
        rgb += 3;
    }
    io_store_eflags(eflags); /* 复原中断许可标志 */
    return;
}

void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int
y0, int x1, int y1)
{
    int x, y;
    for (y = y0; y <= y1; y++) {
        for (x = x0; x <= x1; x++)

```

```

        vram[y * xsize + x] = c;
    }
    return;
}
void init_screen(char *vram, int x, int y)
{
    boxfill8(vram, x, COL8_008484, 0, 0, x - 1, y - 29);
    boxfill8(vram, x, COL8_C6C6C6, 0, y - 28, x - 1, y - 28);
    boxfill8(vram, x, COL8_FFFFFFFF, 0, y - 27, x - 1, y - 27);
    boxfill8(vram, x, COL8_C6C6C6, 0, y - 26, x - 1, y - 1);

    boxfill8(vram, x, COL8_FFFFFFFF, 3, y - 24, 59, y - 24);
    boxfill8(vram, x, COL8_FFFFFFFF, 2, y - 24, 2, y - 4);
    boxfill8(vram, x, COL8_848484, 3, y - 4, 59, y - 4);
    boxfill8(vram, x, COL8_848484, 59, y - 23, 59, y - 5);
    boxfill8(vram, x, COL8_000000, 2, y - 3, 59, y - 3);
    boxfill8(vram, x, COL8_000000, 60, y - 24, 60, y - 3);

    boxfill8(vram, x, COL8_848484, x - 47, y - 24, x - 4, y - 24);
    boxfill8(vram, x, COL8_848484, x - 47, y - 23, x - 47, y - 4);
    boxfill8(vram, x, COL8_FFFFFFFF, x - 47, y - 3, x - 4, y - 3);
    boxfill8(vram, x, COL8_FFFFFFFF, x - 3, y - 24, x - 3, y - 3);
    return;
}

```

代码说明

`binfo_scrnx = (short *) 0x0ff4`,以这段函数为例，`binfo` 是 `bootinfo` 的缩写，`scrnx` 是 `screen` 的缩写。这里的 `0x0ff4` 是和 `asmhead.nas` 中的值是一致的，用来获取内存的值。还有一个地方是我们把显示画面的地方单独做成了函数 `init_screen` 执行完还是我们“窗口的样子”。

试用结构体

这次用C语言中的结构体来使我们的程序变得更加易读。下面例子我们在 `harib02b` 中进行。下面是本次的 `bootpack.c`

```

void io_hlt(void);
void io_cli(void);
void io_out8(int port, int data);
int io_load_eflags(void);
void io_store_eflags(int eflags);

void init_palette(void);
void set_palette(int start, int end, unsigned char *rgb);
void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int y0, int x1, int y1);
void init_screen(char *vram, int x, int y);

```



```

/* 上面是函数声明部分 */
#define COL8_000000      0
#define COL8_FF0000      1
#define COL8_00FF00      2
#define COL8_FFFF00      3
#define COL8_0000FF      4
#define COL8_FF00FF      5
#define COL8_00FFFF      6
#define COL8_FFFFFFFF      7
#define COL8_C6C6C6      8
#define COL8_840000      9
#define COL8_008400     10
#define COL8_848400     11
#define COL8_000084     12
#define COL8_840084     13
#define COL8_008484     14
#define COL8_848484     15


struct BOOTINFO{
    char cyls, leds, vmode, reserve; //这里的变量暂时没有用到,后面要用,先写在这里。
    short scrnx, scrny;
    char * vram;
};


void HariMain(void)
{
    char *vram; /* 指针变量vram是BYTE[...]用到的地址*/
    int xsize, ysize; /* 声明变量。变量是32位整数型 */
    struct BOOTINFO * binfo;

    init_palette(); /* 设定调色板*/
    binfo = (struct BOOTINFO *) 0xff0;
    xsize = (*binfo).scrnx;
    ysize = (*binfo).scrny;
    vram = (*binfo).vram;

    init_screen(vram, xsize, ysize);

    for (;;) {
        io_hlt();
    }
}


void init_palette(void)
{

```

```

static unsigned char table_rgb[16 * 3] = {
    0x00, 0x00, 0x00,    /* 0:黑 */
    0xff, 0x00, 0x00,    /* 1:亮红 */
    0x00, 0xff, 0x00,    /* 2:亮绿 */
    0xff, 0xff, 0x00,    /* 3:亮黄 */
    0x00, 0x00, 0xff,    /* 4:亮蓝 */
    0xff, 0x00, 0xff,    /* 5:亮紫 */
    0x00, 0xff, 0xff,    /* 6:浅亮蓝 */
    0xff, 0xff, 0xff,    /* 7:白 */
    0xc6, 0xc6, 0xc6,    /* 8:亮灰 */
    0x84, 0x00, 0x00,    /* 9:暗红 */
    0x00, 0x84, 0x00,    /* 10:暗绿 */
    0x84, 0x84, 0x00,    /* 11:暗黄 */
    0x00, 0x00, 0x84,    /* 12:暗青 */
    0x84, 0x00, 0x84,    /* 13:暗紫 */
    0x00, 0x84, 0x84,    /* 14:浅暗蓝 */
    0x84, 0x84, 0x84     /* 15:暗灰 */
};

set_palette(0, 15, table_rgb);
return;

/* C语言中static char 语句只能用于数据,相当于汇编语言中的DB指令 */
}

void set_palette(int start, int end, unsigned char *rgb)
{
    int i, eflags;
    eflags = io_load_eflags(); /* 记录中断许可标志的值 */
    io_cli();                  /* 将中断许可标志置为0,禁止中断 */
    io_out8(0x03c8, start);
    for (i = start; i <= end; i++) {
        io_out8(0x03c9, rgb[0] / 4); //颜色值右移两位,这样的结果除了使颜色更亮一
        些之外,暂时没有发现还有其他的什么用处
        io_out8(0x03c9, rgb[1] / 4);
        io_out8(0x03c9, rgb[2] / 4);
        rgb += 3;
    }
    io_store_eflags(eflags); /* 复原中断许可标志 */
    return;
}

void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int
y0, int x1, int y1)
{
    int x, y;
    for (y = y0; y <= y1; y++) {
        for (x = x0; x <= x1; x++)
            vram[y * xsize + x] = c;
    }
    return;
}

```

```

}
void init_screen(char *vram, int x, int y)
{
    boxfill8(vram, x, COL8_008484, 0, 0, x - 1, y - 29);
    boxfill8(vram, x, COL8_C6C6C6, 0, y - 28, x - 1, y - 28);
    boxfill8(vram, x, COL8_FFFFFFFF, 0, y - 27, x - 1, y - 27);
    boxfill8(vram, x, COL8_C6C6C6, 0, y - 26, x - 1, y - 1);

    boxfill8(vram, x, COL8_FFFFFFFF, 3, y - 24, 59, y - 24);
    boxfill8(vram, x, COL8_FFFFFFFF, 2, y - 24, 2, y - 4);
    boxfill8(vram, x, COL8_848484, 3, y - 4, 59, y - 4);
    boxfill8(vram, x, COL8_848484, 59, y - 23, 59, y - 5);
    boxfill8(vram, x, COL8_000000, 2, y - 3, 59, y - 3);
    boxfill8(vram, x, COL8_000000, 60, y - 24, 60, y - 3);

    boxfill8(vram, x, COL8_848484, x - 47, y - 24, x - 4, y - 24);
    boxfill8(vram, x, COL8_848484, x - 47, y - 23, x - 47, y - 4);
    boxfill8(vram, x, COL8_FFFFFFFF, x - 47, y - 3, x - 4, y - 3);
    boxfill8(vram, x, COL8_FFFFFFFF, x - 3, y - 24, x - 3, y - 3);
    return;
}

```

试用箭头标记

所谓的“箭头标记”不过是C语言中指针的一种表示方式，详细请看下面的例子。下面的代码我们在 `harib02c` 中完成。

原先的HariMain函数

```

void HariMain(void)
{
    char *vram; /* 指针变量vram是BYTE[...]用到的地址*/
    int xsize, ysize; /* 声明变量。变量是32位整数型 */
    struct BOOTINFO * binfo;

    init_palette(); /* 设定调色板*/
    binfo = (struct BOOTINFO *) 0xff0;
    xsize = (*binfo).scrnx;
    ysize = (*binfo).scrny;
    vram = (*binfo).vram;

    init_screen(vram, xsize, ysize);

    for (;;) {
        io_hlt();
    }
}

```

现在的HariMain函数

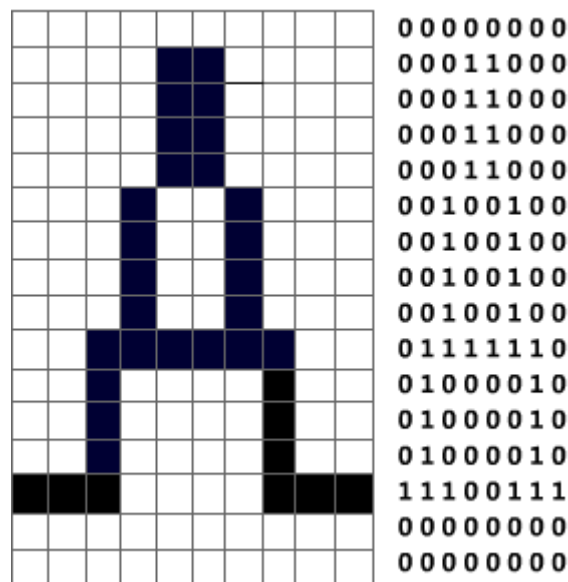
```
void HariMain(void)
{
    struct BOOTINFO * binfo = (struct BOOTINFO *) 0xff0;

    init_palette(); /* 设定调色板*/
    init_screen(binfo->vram, binfo->scrnx, binfo->scrny);

    for (;;) {
        io_hlt();
    }
}
```

显示字符

接着做我们的“显示”，我们已经实现了“画”，这次要实现的是在“画”上写“字”，用8×16的长方形像素点阵来写字。对应的“字符”如书上所示，这里不再进行描绘，本质上这个字符是靠点来组成的，把 1 当作有色点，把 0 当作无色点，这样我们靠16个二进制数就组成了我们的字符，如图所示，8位是一个字节，然后是16个8位，即，16个字节组成了一个字符，我们在 `harib02d`。如下图所示:太难了，纯手画。



上面这种描画文字形状的数据称之为 **字体数据**，我们暂时可以通过下面的方式写到程序里：

```
static char font_A[16] = {
    0x00, 0x18, 0x18, 0x18, 0x18, 0x24, 0x24, 0x24,
    0x24, 0x7e, 0x42, 0x42, 0x42, 0xe7, 0x00, 0x00
};
```

正好是16个16进制数字（因为二进制在C语言中是不支持），我们可以写下下面这个函数。

```
void putfont8(char * vram,int xsize,int x,int y,char c,char * font)
{
    int i;
    char d;//data
    for(i = 0;i < 16;i++){
        d = font[i];
        if((d & 0x80) != 0){ vram[(y+i) * xsize + x + 0] = c}
        if((d & 0x40) != 0){ vram[(y+i) * xsize + x + 1] = c}
        if((d & 0x20) != 0){ vram[(y+i) * xsize + x + 2] = c}
        if((d & 0x10) != 0){ vram[(y+i) * xsize + x + 3] = c}
        if((d & 0x08) != 0){ vram[(y+i) * xsize + x + 4] = c}
        if((d & 0x04) != 0){ vram[(y+i) * xsize + x + 5] = c}
        if((d & 0x02) != 0){ vram[(y+i) * xsize + x + 6] = c}
        if((d & 0x01) != 0){ vram[(y+i) * xsize + x + 7] = c}
    }
    return;
}
```

代码说明

先说一下函数的参数的问题，vram是指针，用来获取VRAM中的真实地址，获取到地址后，才能往里面写数据。xsize是系统画面的横向的所有像素点的数量，相当于前面所讲的 $0xa0000 + x + y * 320$ 中的320，只不过这里是调用asmhead.nas中的相关数据，实现了从静态数据到动态数据的转变；x、y分别是相对于整个系统画面来说的相对坐标。c是将要写入的真实的数据，font也是指针，用来获取font数据。其中的0x80转化成2进制是10000000恰好是8位，它与d进行“与”运算，运算的结果如果是0，就说明最左边一位是0,这样就能判断8位的二进制数字的最高位，也就是最左边的那一位是1还是0.同样的道理，0x40是1000000比10000000少一位，进行同样的“与”运算，便能够判断出最左便第二位是0还是1，依次类推。

vram[(y+i) * xsize + x + 0]这些其实就和 $0xa0000 + x + y * 320$ 是一样的，就是为了计算出要写入“1”的位置（VRAM内存中），后面那的+0、+1是说，前面已经有几位占着了，再写入的话要在后面接着写。

程序优化

为了程序的优化，我们改写成下面的样子。

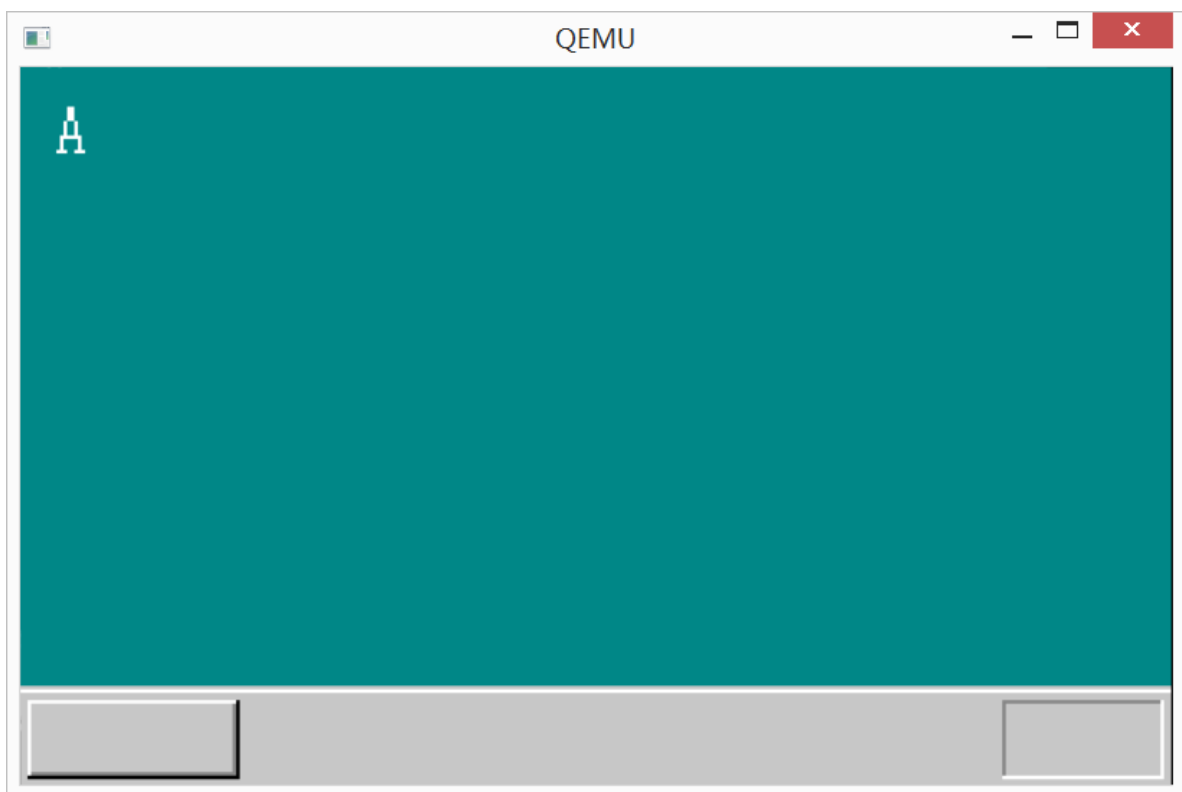
```
void putfont8(char * vram,int xsize,int x,int y,char c,char * font)
{
    int i;
    char * p,d;//data
    for(i = 0; i < 16; i++){
        p = vram + (y + i) * xsize + x;
        d = font[i];
```

```
    if ((d & 0x80) != 0) { p[0] = c; }
    if ((d & 0x40) != 0) { p[1] = c; }
    if ((d & 0x20) != 0) { p[2] = c; }
    if ((d & 0x10) != 0) { p[3] = c; }
    if ((d & 0x08) != 0) { p[4] = c; }
    if ((d & 0x04) != 0) { p[5] = c; }
    if ((d & 0x02) != 0) { p[6] = c; }
    if ((d & 0x01) != 0) { p[7] = c; }

}
return;
}
```

运行结果

不出意外的话，会显示出“A”，而且位置在左上角。看，显示出来了！



增加字体

上一版本我们成功的显示出了“A”，但是对于现实这一个字符，我们就用了一个相当复杂的方式，要是显示更多的字岂不是要很麻烦？这次我们就用另外一种方式显示文字显示的效果，这次用到的关键文件是 `hankaku.txt`。这个的使用权已经得到了原作者的许可，我们直接使用就可以了。

这个不是汇编，也不是C语言，而是一个 `txt` 文本文件。所以需要有一个专门的编译器 `makefont.exe`，这个“编译器”早已经写好，我们使用即可。它起到的作用是把256个字符的字体文件读进来，然后输出成 $16 \times 256 = 4096$ 字节的文件。（这个文本文件是依照一般ASCII字符编码，所以含有256个字符）编译生成 `hankaku.bin`，此时，还不能与 `bootpack.obj` 连接，因为它不是目标文件。所以还要加上连接所必需的接口信息，将它变成目标文件。这个工作由我们前面所提到的 `bin2obj.exe` 来完成。因为用到了“编

译器”makefont.exe,相应的Makefile这个文件，稍作修改。大家可以自行查看，这里不做过多介绍。这次的工作我们在 `harib02e` 中完成。下面的看一下这次的HariMain函数：

```
void HariMain(void)
{

    struct BOOTINFO * binfo = (struct BOOTINFO *) 0x0ff0;
    extern char hankaku[4096];

    init_palette(); /* 设定调色板*/
    init_screen(binfo->vram, binfo->scrnx, binfo->scrny);
    putfont8(binfo->vram, binfo->scrnx, 8, 8, COL8_FFFFFFFF, hankaku + 'Y'
* 16);
    putfont8(binfo->vram, binfo->scrnx, 16, 8, COL8_FFFFFFFF, hankaku + 'G'
* 16);
    putfont8(binfo->vram, binfo->scrnx, 24, 8, COL8_FFFFFFFF, hankaku + 'L'
* 16);
    putfont8(binfo->vram, binfo->scrnx, 32, 8, COL8_FFFFFFFF, hankaku + 'A'
* 16);
    putfont8(binfo->vram, binfo->scrnx, 40, 8, COL8_FFFFFFFF, hankaku + 'U'
* 16);
    putfont8(binfo->vram, binfo->scrnx, 48, 8, COL8_FFFFFFFF, hankaku + 'G'
* 16);
    putfont8(binfo->vram, binfo->scrnx, 56, 8, COL8_FFFFFFFF, hankaku + 'U'
* 16);
    putfont8(binfo->vram, binfo->scrnx, 64, 8, COL8_FFFFFFFF, hankaku + 'S'
* 16);
    putfont8(binfo->vram, binfo->scrnx, 80, 8, COL8_FFFFFFFF, hankaku + '1'
* 16);
    putfont8(binfo->vram, binfo->scrnx, 88, 8, COL8_FFFFFFFF, hankaku + '2'
* 16);
    putfont8(binfo->vram, binfo->scrnx, 96, 8, COL8_FFFFFFFF, hankaku + '3'
* 16);

    for (;;) {
        io_hlt();
    }
}
```

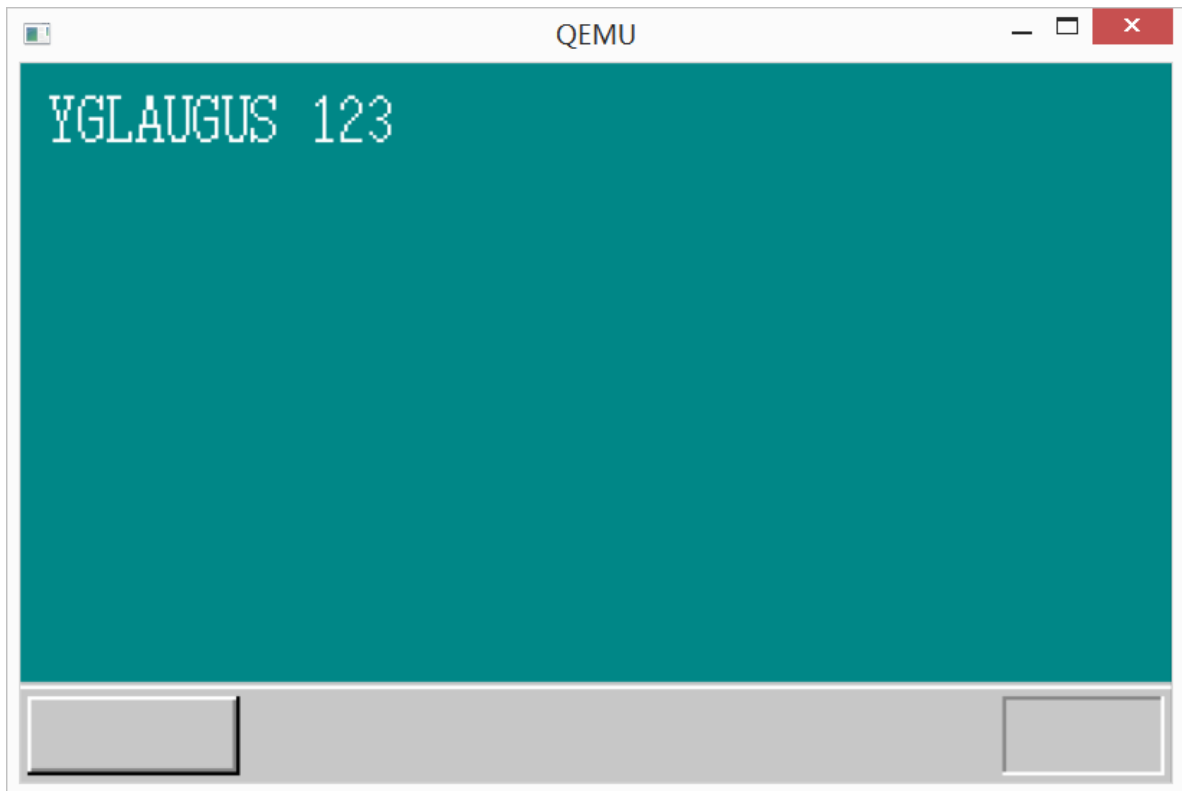
代码说明

在ASCII码中，A的字符编码是0x41，所以A的字体数据，放在从“hankaku + 0x41 * 16”开始的16个字节中（上次已经说明了，设定是16个字节为一个字符），在 `extern char hankaku[4096]` 中，我们可以知道两点信息，一个是hankaku是外部数据，因为我们用到了关键字 `extern`；另外一点是，读入外部数据hankaku，使用的是数组的方式，那么也就是说 `hankaku` 如果单独是使用就是地址，故“hankaku + 0x41 * 16”的hankaku 就是先找到hankaku.txt然后再把其中的内容写入。

C语言中 'A' 可以代替0x41,所以“hankaku + 0x41 * 16”可以写成“hankaku + 'A' * 16”。

运行结果

看，成功了！



显示字符串

上个版本中我们仅仅就显示了几个字符，就写了大量的代码，为了简单，这次我们写一个字符串的函数，这样显示文字就不会那么费劲了，这个函数的思路呢，就是把一个个字符连载一起就组成了字符串，但是值得注意的是，C语言中字符串都是以0x00结尾的，这个函数我们把他命名为 `putfonts8_asc`，后面asc的意思是说字符编码是用的ASCII。下面的工作我们在 `harib02f` 中进行。

函数：

```
void putfonts8_asc(char * vram, int xsize, int x,int y,char c,unsigned
char * s)
{
    extern char hankaku[4096];
    for(; *s != 0x00; s++){
        putfont8(vram, xsize, x, y, c, hankaku + *s * 16);
        x += 8;
    }
    return;
}
```

整理后的HariMain函数


```

void HariMain(void)
{

    struct BOOTINFO * binfo = (struct BOOTINFO *) 0x0ff0;

    init_palette(); /* 设定调色板*/
    init_screen(binfo->vram, binfo->scrnx, binfo->scrny);
    putfonts8_asc(binfo->vram, binfo->scrnx, 8, 8, COL8_FFFFFFFF,
"by:YJLaugus");
    putfonts8_asc(binfo->vram, binfo->scrnx, 31, 31, COL8_000000, "Ini
OS.");
    putfonts8_asc(binfo->vram, binfo->scrnx, 30, 30, COL8_FFFFFFFF, "Ini
OS.");

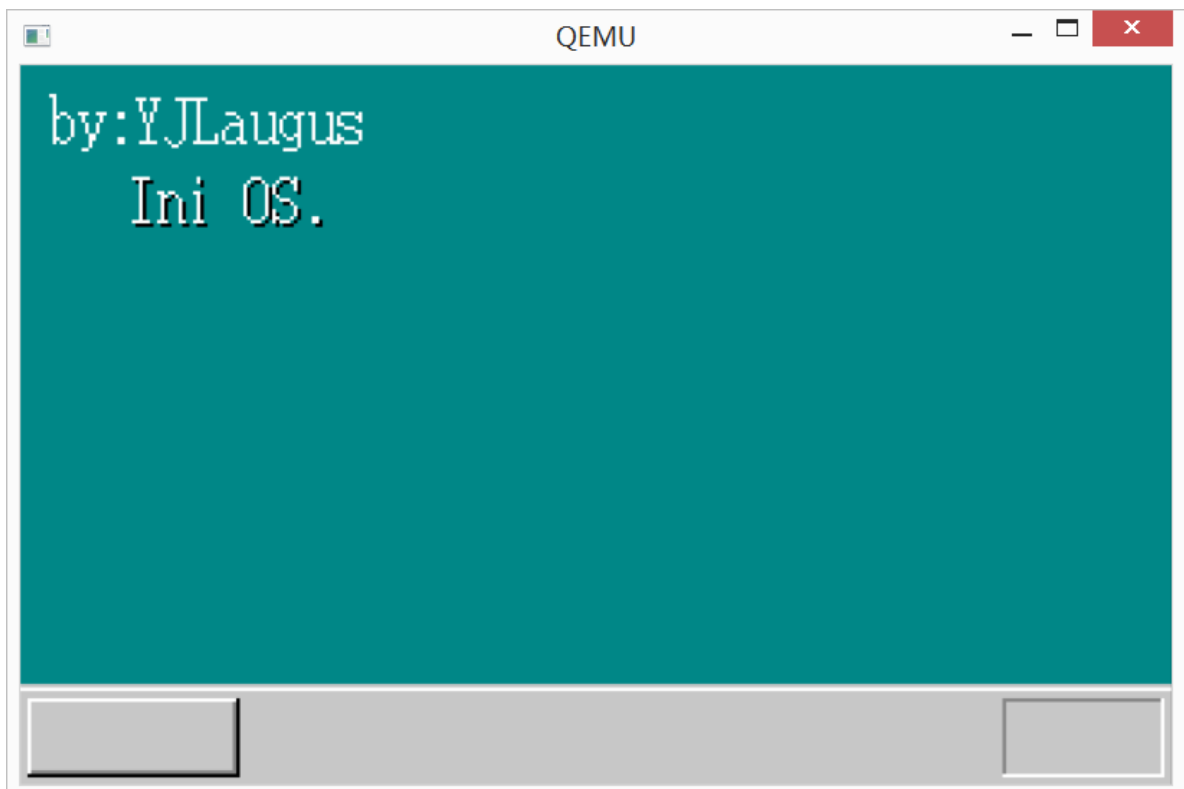
    for (;;) {
        io_hlt();
    }
}

```

代码说明

在函数中我们定义的 `unsigned char * s` , `s`是字符型的指针变量 , 前面的加上 `unsiged` 确保是非负的 , 其实我们的 `s` 指向的就是这一个串的地址 , `*s` 就是串的内容了。

运行结果



显示变量值

显示变量值是为了能够以后能够更好的调试我们的系统程序，我们现在用的windows系统是无法对我们自己写的系统进行调试的，因为我们已经能够让字符串显示在我们的“系统屏幕”上了，那么接下来就是用我们的字符串显示出我们的变量了。

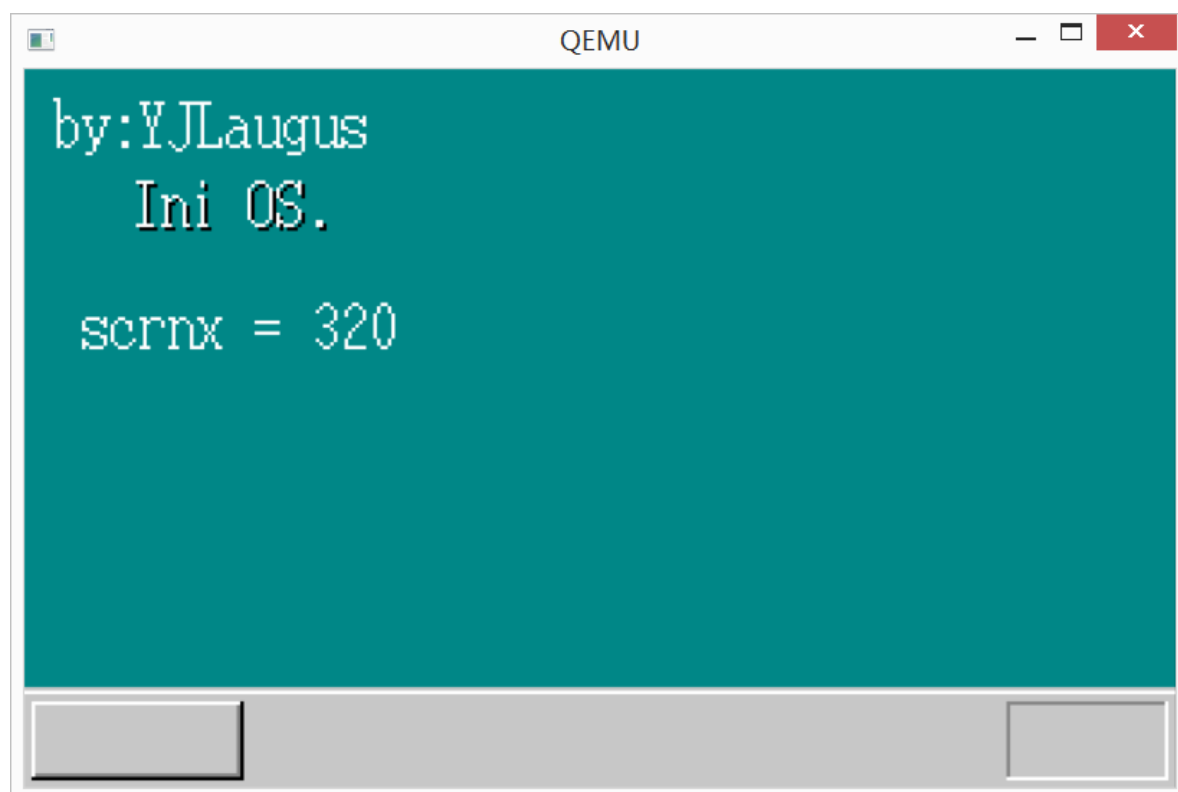
有一点，我们前面做的是 `putfonts8_asc` 函数仅仅的功能显示出字符串到屏幕上，这个函数的最后的参数的是指针，也就是一个地址，之所以能输出一个字符串，也是因为字符串有一个“串地址”。但是能否用这个函数输出变量的 `值` 呢？答案是否定的，`值` 可不是地址，所以这里我们要用到一个函数 `sprintf`，在自己写的操作系统中是一般不能随便使用 `printf` 函数的，但是 `sprintf` 函数可以随便使用，因为其不是按指定格式输出，只是将输出内容作为字符串 `写在内存中`（注意此事并没有输出到系统上哦），所以接下来我们还需要我们的 `putfonts8_asc` 函数，把写到内存中的字符串显示到屏幕上。

写在HariMain函数中：

```
char s[40];
sprintf(s, "scrnx = %d", binfo->scrnx);
putfonts8_asc(binfo->vram, binfo->scrnx, 16, 64, COL8_FFFFFFFF, s);
```

`sprintf` 函数的使用方法：`sprintf(地址, 格式, 值, 值, 值, ...)`，这里的地址是指所生成字符串的存放地址。关于其中的 `%d` 格式的问题，大家可参考书籍的p98。

运行效果



显示鼠标指针

接下来我们开始绘制鼠标指针，并在 `harib02h` 中进行接下来的的工作，我们把鼠标指针设置为16×16=256字节的内存，然后把鼠标指针的数据写入其中。封装成在函数

`init_mouse_cursor8` 中。

init_mouse_cursor8

```
void init_mouse_cursor8(char * mouse, char bc)
//准备鼠标指针 (16×16)
{
    static char cursor[16][16] = {
        "*****..",
        "*0000000000*...",
        "*0000000000*....",
        "*000000000*.....",
        "*00000000*.....",
        "*0000000*.....",
        "*000000*.....",
        "*00000000*.....",
        "*0000*000*.....",
        "*000*..*000*....",
        "*00*....*000*...",
        "*0*.....*000*..",
        "**.....*000*.",
        "*.....*000*",
        ".....*00*",
        ".....***"
    };
    int x, y;

    for (y = 0; y < 16; y++) {
        for (x = 0; x < 16; x++) {
            if (cursor[y][x] == '*') {
                mouse[y * 16 + x] = COL8_000000;
            }
            if (cursor[y][x] == '0') {
                mouse[y * 16 + x] = COL8_FFFFFFFF;
            }
            if (cursor[y][x] == '.') {
                mouse[y * 16 + x] = bc;
            }
        }
    }

    return;
}
```

变量 `bc` 是鼠标的背景色，因为在绘制鼠标的时候，因为是用到的字符数组，是用二维数组完成的，也就是个矩阵的，在几何上说其实是个矩形，这样的话咱的鼠标不是矩形的。所以要给鼠标指针填充背景色（`back-color`）。上面的做到，都是绘制鼠标的工作，但是并没做显示在“系统显示器”上的工作，接下来我们让他显示在上面，这就需要我们再写一个函数 `putblock_8`。

putblock_8

```

void putblock8_8(char *vram, int vxsize, int pxsize, int pysize, int px0,
int py0, char *buf, int bxsiz)
{
    int x, y;
    for (y = 0; y < pysize; y++) {
        for (x = 0; x < pxsize; x++) {
            vram[(py0 + y) * vxsize + (px0 + x)] = buf[y * bxsiz + x];
        }
    }
    return;
}

```

可以看出，同过上面的两个函数，就可以把我们的鼠标指针放在系统桌面上了，还有一点需要提醒一下，以前我们把要显示东西显示在系统上，只需要一个函数就完成，通过 `vram` 指针就能很方便的完成这一点，但是为什么这次我们要用到两个函数呢？其实，这一点也很好说明，原因是我们这次绘制的图案是相对复杂的，参数太多，一个函数不方便。还有一个点，是为了以后方便的对系统加以维护，以后我们绘制的图案会更加的复杂，难不成每次都要写同样的参数对函数进行调用吧！我们写了 `putblock_8` 这个函数就能很方便的以后的“图形块”进行调用了。下面是这次的HairMain函数的调整。

HariMain

```

void HariMain(void)
{

    struct BOOTINFO * binfo = (struct BOOTINFO *) 0xff0;
    char s[40], mcursor[256]; //16 × 16
    int mx, my; //鼠标x,y位置

    init_palette(); /* 设定调色板*/
    init_screen(binfo->vram, binfo->scrnx, binfo->scrny);
    mx = (binfo->scrnx - 16) / 2; /* 为了使鼠标指针居中 */
    my = (binfo->scrny - 28 - 16) / 2;
    init_mouse_cursor8(mcursor, COL8_008484);
    putblock8_8(binfo->vram, binfo->scrnx, 16, 16, mx, my, mcursor, 16);
    sprintf(s, "(%d, %d)", mx, my);
    putfonts8_asc(binfo->vram, binfo->scrnx, 0, 0, COL8_FFFFFFFF, s);

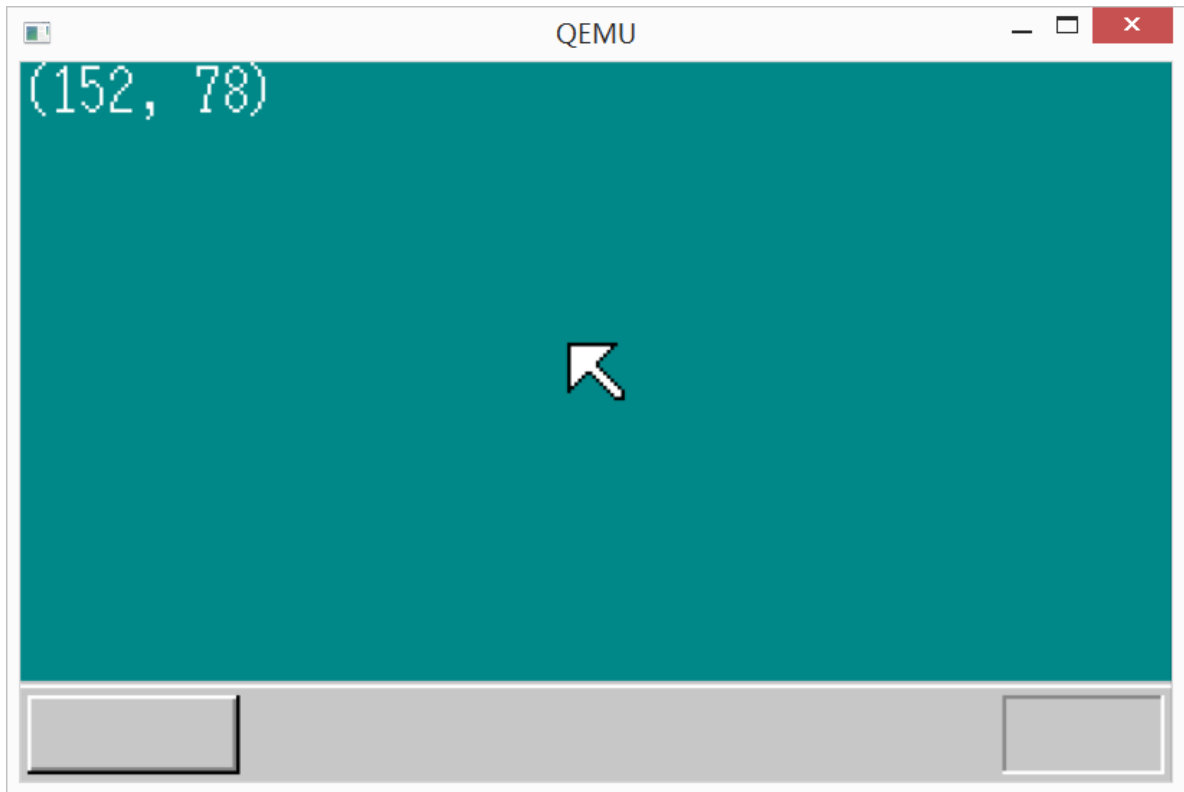
    for (;;) {
        io_hlt();
    }
}

```

代码说明

这次的函数中的变量是比较多的，其中vram和vxsize是关于VRAM的信息。他们的值分别是0xa0000 和320。pxsize和pysize 是要显示的图形（ picture ）的大小，鼠标指针的大小是16×16，所以这两个值都是16。px0和py0指定图形在画面上的显示位置。最后的buf和bxsize分别指定图形存放地址和每一行含有的像素数。bxsize和pxsize是背景图形的大小（ 矩形 ），但也有时候想放入不同的值，所以还是要分别制定这两个值。

运行效果



GDT与IDT的初始化

为什么要讲解这一节呢？上节我们做出了鼠标，但是“中看不中用”，上次做的鼠标是不能移动的，这样的鼠标没有作用，所以呢，接下来就是如何让我们的鼠标动起来。要实现这个功能的前提就是将GDT（ global/segment descriptor table 全局段号记录表 ）和 IDT（ interrupt descriptor table 中断记录表 ）进行初始化。完成这两部才能进行我们“移动鼠标的操作”，涉及到的原理就是“中断机制”。

要实现一系列的“计算机响应”，比如打字、鼠标移动等，就必须要用到中断机制。中断机制必须先进行GDT、IDT的初始化，这次我们就是先完成这个前提工作，接下来的工作都在 `harib02i` 中进行。

现在的操作系统一般都是支持同时运行多个程序的，这样就不可避免的两个或多个程序极有可能运行时抢占同一块内存区域，如果没有相应的处理机制的话，我们系统可能会崩溃或者宕机。处理这个情况的机制 就是分段。

分段就是把内存分为多个块，**每一块的起始地址都看作0**来处理，这很方便，有了这个功能，任何程序都可以先写上一句 `ORG 0`。像这样分割出来的块称为段。需要注意的是，我们用16位的时候曾提到过段寄存器，这里的分段，使用的就是段寄存器。

为了表示一个段，需要以下信息：

- 段的大小
- 段的起始地址
- 段的管理属性（禁止写入，禁止执行，系统专用等）

CPU用8个字节（64位）的数据来表示这些信息，但是用于指定的段寄存器只有16位。或许有人会猜在32位模式下，段寄存器会扩展到64位，但事实上段寄存器仍然是16位。这里我们还是模仿图形调色板的做法，也就是说先有一个段号，存放在段寄存器中，然后预先设定段号与段的对应关系。

因为寄存器是16位的，所以本来应该处理0~65535（2的16次方是65536）范围的数，但是由于CPU设计上的原因，段寄存器低3位数不能使用，故能够使用的实际上只有13位，也就是0~8191（2的13次方）。

下一步就是段号的设定了。这是对于CPU进行的设定，算是“内部”操作，所以，这里不再需要向调色板那样使用 `io_out`，因为不是外部设备嘛。但是我们上面提到，一共是8192个段，一个段是8个字节，也就是8192×8 = 65536字节（64KB。这在CPU中是无法存储的，所以这些数据是写入内存的，并非在CPU中进行存储。

这次的工作主要是对 `bootpack.c` 和 `naskfunc.nas` 进行了更改。

bootpack.c部分

```
void init_gdtidt(void)
{
    struct SEGMENT_DESCRIPTOR *gdt = (struct SEGMENT_DESCRIPTOR *)
0x00270000;
    struct GATE_DESCRIPTOR *idt = (struct GATE_DESCRIPTOR *)
0x0026f800;
    int i;

    /*GDT的初始*/
    for (i = 0; i < 8192; i++) {
        set_segmdesc(gdt + i, 0, 0, 0);
    }
    //先设置两个段
    set_segmdesc(gdt + 1, 0xffffffff, 0x00000000, 0x4092); //上限是
0xffffffff(4GB), 基地址是0, 表示CPU所管理的全部内存
    set_segmdesc(gdt + 2, 0x0007ffff, 0x00280000, 0x409a); //大小512KB, 基地址
是0x80000, 正好为bookpack.hrb准备的
//此处为原来的
bootpack.h写入的地址, 重新做了下调整。
    load_gdtr(0xffff, 0x00270000); //C语言中不能给GDTR(一种特殊寄存器, 用来存放内
存的起始地址, 和有效设定个数)赋值
//所以这里需要借助汇编

    /*IDT的初始化*/
    for (i = 0; i < 256; i++) {
        set_gatedesc(idt + i, 0, 0, 0);
    }
}
```

```

    }
    load_idtr(0x7ff, 0x0026f800);

    return;
}

void set_segmdesc(struct SEGMENT_DESCRIPTOR *sd, unsigned int limit, int
base, int ar)
{
    if (limit > 0xffff) {
        ar |= 0x8000; /* G_bit = 1 */
        limit /= 0x1000;
    }
    sd->limit_low    = limit & 0xffff;
    sd->base_low     = base & 0xffff;
    sd->base_mid     = (base >> 16) & 0xff;
    sd->access_right = ar & 0xff;
    sd->limit_high   = ((limit >> 16) & 0x0f) | ((ar >> 8) & 0xf0);
    sd->base_high    = (base >> 24) & 0xff;
    return;
}

void set_gatedesc(struct GATE_DESCRIPTOR *gd, int offset, int selector,
int ar)
{
    gd->offset_low   = offset & 0xffff;
    gd->selector     = selector;
    gd->dw_count     = (ar >> 8) & 0xff;
    gd->access_right = ar & 0xff;
    gd->offset_high  = (offset >> 16) & 0xffff;
    return;
}

void HariMain(void)
{
    struct BOOTINFO * binfo = (struct BOOTINFO *) 0xff0;
    char s[40], mcursor[256]; //16 × 16
    int mx, my; //鼠标x,y位置

    init_gdtidt();
    init_palette(); /* 设定调色板*/
    init_screen(binfo->vram, binfo->scrnx, binfo->scrny);
    mx = (binfo->scrnx - 16) / 2; /* 为了使鼠标指针居中 */
    my = (binfo->scrny - 28 - 16) / 2;
    init_mouse_cursor8(mcursor, COL8_008484);
    putblock8_8(binfo->vram, binfo->scrnx, 16, 16, mx, my, mcursor, 16);

```

```

sprintf(s, "(%d, %d)", mx, my);
putfonts8_asc(binfo->vram, binfo->scrnx, 0, 0, COL8_FFFFFFFF, s);

for (;;) {
    io_hlt();
}
}

```

naskfunc.nas

```

GLOBAL _load_gdtr, _load_idtr

_load_gdtr:    ; void load_gdtr(int limit, int addr);
    MOV     AX,[ESP+4]        ; limit
    MOV     [ESP+6],AX
    LGDT    [ESP+6]
    RET

_load_idtr:    ; void load_idtr(int limit, int addr);
    MOV     AX,[ESP+4]        ; limit
    MOV     [ESP+6],AX
    LIDT    [ESP+6]
    RET

```

代码说明

对与上述的代码，难理解的地方给出了部分注释说明，其中 `set_segmdesc` 和 `set_gatedesc` 函数作者书上也没过多解释，这里涉及到较多汇编知识，如果读者没有汇编基础的话，建议先略过。回过头来再仔细咀嚼。现在只要知道这两个函数是对段的设定就好了。第一个是段的基本设定，第二个是段的管理属性设定。

还有一点是在 `naskfunc.nas` 中写入函数映射的时候，不要忘记 `GLOBAL _load_gdtr, _load_idtr` 不然C程序还是找不到汇编函数的入口的。最后运行一下，还是那个样子，因为我们只是做了初始化工作，并没有做什么其他的工作。

第6天

分割源文件

先把以前的文件做下整理，简单的分割一下。分割文件有利也有弊，如下分析：

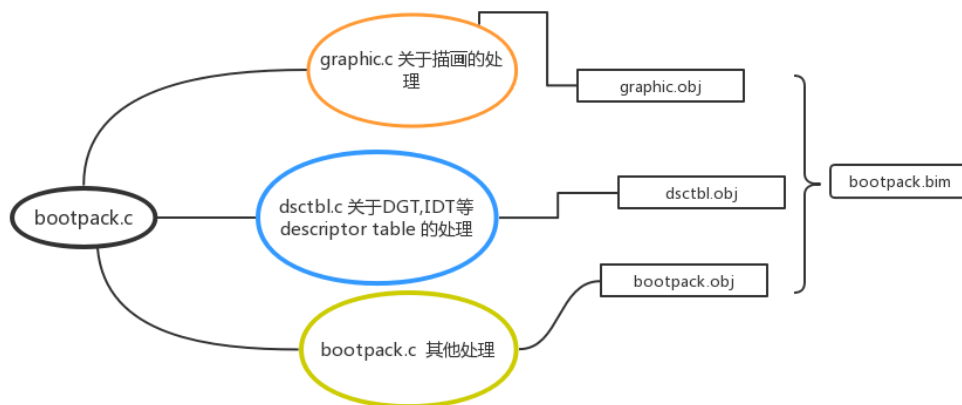
优点

- 按照处理内容进行分类，如果分的好的话，将来进行修改时，容易找到地方。
- 如果Makefile写的好，只需要编译修改过的文件，就可以提高make的速度。
- 耽搁源文件都不长。多个小文件比一个大文件好处理。
- 看起来很酷（qvq）

缺点

- 源文件数量增加
- 分类分得不好的话，修改时不容易找到地方。

分割结构图



小提示

分割完成后，如果 `graphic.c` 也想使用 `naskfunc.nas` 中的函数的时候，必须在使用前先声明。虽然这些声明在 `bootpack.c` 中已经声明了。但编译器在编译 `graphic.c` 的时候根部不知道 `bootpack.c` 存在，所以在 `graphic.c` 中也要有相关函数的声明。

整理Makefile

在整理之前，我们先看下面的程序。

```
bootpack.nas : bootpack.gas Makefile
$(GAS2NASK) bootpack.gas bootpack.nas

graphic.nas : graphic.gas Makefile
$(GAS2NASK) graphic.gas graphic.nas
```

上面的程序都是做的一样的事情，我没可以用下面的改写：

```

%.gas : %.c Makefile
    $(CC1) -o $*.gas $*.c

%.nas : %.gas Makefile
    $(GAS2NASK) $*.gas $*.nas

%.obj : %.nas Makefile
    $(NASK) $*.nas $*.obj $*.lst

```

make.exe 会首先寻找普通的生成规则，如果没有找到，就尝试使用一般规则。所以即使一般规则和普通生成规则有冲突，也没关系的。这样真是方便多了。

整理头文件

这次主要是整理C语言程序，上次我们把 `bootpack.c` 进行了分割，但是不好的是，必须在分割好的文件中进行函数的声明，导致了部分函数声明的重复。这样一来就得不偿失了。这次我们把所有的函数的声明放在 `bootpack.h` 这个头文件中。然后再在各个分割的文件中加上一句 `#include "bootpack.h"` 把这个头文件引入就好了。接下来的工作我们在 `harb03c`。下面附上 `bootpack.h` 的代码。

bootpack.h

```

/* asmhead.nas */
struct BOOTINFO { /* 0xff0-0xffff */
    char cyls; /* 启动区读硬盘读到何处为止 */
    char leds; /* 启动时键盘LED的状态 */
    char vmode; /* 显卡模式为多少位色彩 */ //这些在前面都写到过,但是并没有使用,这里是第一次进行说明
    char reserve;
    short scrnx, scrny; /* 画面分辨率*/
    char *vram;
};
#define ADR_BOOTINFO 0x0000ff0

/* naskfunc.nas */
void io_hlt(void);
void io_cli(void);
void io_out8(int port, int data);
int io_load_eflags(void);
void io_store_eflags(int eflags);
void load_gdtr(int limit, int addr);
void load_idtr(int limit, int addr);

/* graphic.c */
void init_palette(void);
void set_palette(int start, int end, unsigned char *rgb);
void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int y0, int x1, int y1);
void init_screen8(char *vram, int x, int y);

```

```

void putfont8(char *vram, int xsize, int x, int y, char c, char *font);
void putfonts8_asc(char *vram, int xsize, int x, int y, char c, unsigned
char *s);
void init_mouse_cursor8(char *mouse, char bc);
void putblock8_8(char *vram, int vxsize, int pxsize,
    int pysize, int px0, int py0, char *buf, int bsize);
#define COL8_000000      0
#define COL8_FF0000      1
#define COL8_00FF00      2
#define COL8_FFFF00      3
#define COL8_0000FF      4
#define COL8_FF00FF      5
#define COL8_00FFFF      6
#define COL8_FFFFFFFF      7
#define COL8_C6C6C6      8
#define COL8_840000      9
#define COL8_008400     10
#define COL8_848400     11
#define COL8_000084     12
#define COL8_840084     13
#define COL8_008484     14
#define COL8_848484     15

/* dsctbl.c */
struct SEGMENT_DESCRIPTOR {
    short limit_low, base_low;
    char base_mid, access_right;
    char limit_high, base_high;
};
struct GATE_DESCRIPTOR {
    short offset_low, selector;
    char dw_count, access_right;
    short offset_high;
};
void init_gdtidt(void);
void set_segmdesc(struct SEGMENT_DESCRIPTOR *sd, unsigned int limit, int
base, int ar);
void set_gatedesc(struct GATE_DESCRIPTOR *gd, int offset, int selector,
int ar);
#define ADR_IDT          0x0026f800
#define LIMIT_IDT        0x000007ff
#define ADR_GDT          0x00270000
#define LIMIT_GDT        0x0000ffff
#define ADR_BOTPAK       0x00280000
#define LIMIT_BOTPAK     0x0007ffff
#define AR_DATA32_RW     0x4092
#define AR_CODE32_ER     0x409a

```

说明

像这样，仅由函数声明和 `#define` 等组成的文件，我们称之为头文件。以前的时候我们曾提到 `#include<stdio.h>` 这个头文件其实在C语言中是最常见的，双引号 (`"`) 表示该头文件与源文件位于同一个文件夹中。而尖括号 (`<>`) 则表示该文件位于编译器所提供的文件夹里。

意犹未尽

这部分对前面的 `naskfunc.nas` 中未说明的部分进行说明。先从 `_load_gdtr` 函数说起，这个函数的功能是将指定 段上限 (`limit`) 和地址赋值给名为GDTR的48位寄存器。这个寄存器不能用我们常用的MOV进行赋值，而是用用LGDT命令。

```
_load_gdtr:    ; void load_gdtr(int limit, int addr);
              MOV     AX,[ESP+4]          ; limit
              MOV     [ESP+6],AX
              LGDT    [ESP+6]
              RET
```

该寄存器的低16位数 (也就是内存的最初2个字节) 是段上限，它等于“GDT的有效字节数-1”，表示量的大小。剩下的高32位 (即剩下的4个字节)，代表GDT的开始地址。

在最初执行这个函数的话，`DWORD[ESP + 4]`里存放的是段上限，`DWORD[ESP+8]`里存放的是地址。具体到实际数值就是`0x0000ffff`和`0c00270000`。把他们按字节写出来的话就是 `[FF FF 00 00 00 00 27 00]` (要注意，低位要放在内存地址小的字节里，也就是“16进制最右边的数十低位，按照字节写的话就是写在字节的最左边的位置”)，为了执行LGDT，笔者希望把他们排列成`[FF FF 00 00 2700]`的样子，所以就先用“`MOV AX, [ESP + 4]`”读取最初的`0xffff`，然后再写到`[ESP + 6]`中。这样，结果就成了`[FF FF FF FF 00 00 27 00]`的样子，这样，结果就成了`[FF FF FF FF 00 00 27 00]`，如果从`[ESP + 6]`开始读6字节的话，正好是我们想要的结果。

`naskfunc.nas` 中的 `_load_idtr` 函数是设置IDTR的值，IDTR与GDTR结构基本上是一样的，程序也非常相似，这里就不做过多的介绍。

接下来说一下`dsctbl.c`中的 `set_segmdesc` 函数，根据不同的CPU进行不同的设定。

```
/*按照CPU的规格要求，将段的信息归结为8个字节写入内存*/
void set_segmdesc(struct SEGMENT_DESCRIPTOR *sd, unsigned int limit, int base, int ar)
{
    if (limit > 0xffff) {
        ar |= 0x8000; /* G_bit = 1 */
        limit /= 0x1000;
    }
    sd->limit_low    = limit & 0xffff;
    sd->base_low     = base & 0xffff;
    sd->base_mid     = (base >> 16) & 0xff;
    sd->access_right = ar & 0xff;
    sd->limit_high   = ((limit >> 16) & 0x0f) | ((ar >> 8) & 0xf0);
    sd->base_high    = (base >> 24) & 0xff;
```

```
    return;  
}
```

段地址

段的地址用32位来表示，这个地址在CPU中成为 **段基址**，所以用了base这样的一个地址名字。在段里又分为3部分 low(2字节)，mid(1字节)，high(1字节) 一共是4字节（ $4 \times 8 = 32$ 位），上面的函数就是利用位移运算和And运算往各个字节里填入相应的数值。分三个段是为了与80286时代的程序进行兼容。

段上限

段上限，表示一个段有多少个字节。规定段上限只能使用20位，似乎段上限最大也只能是1MB(2的20次方)。但并且规定在段的属性里设定一个标志位，叫做 **Gbit**。当标志位是 **1** 的时候，limit(段上限)的单位不解释成字节（byte），而是解释成 页(page)，一页等于4KB。这样一来 $4KB \times 1M = 4GB$ ，所以可以指定4GB的段。

这20位的段上限分别写到limit_low 和 limit_high 里。看起来它们好像是总共有3字节（上面说过，low 2B,high 1B），即24位，但实际上我们接着要把段属性写入limit_high的高4位中，所以最后段上限还是只有20位。

段属性

段的属性是占12位的，这样和段上限所占的20位一共正好是32位。段属性又称为“段的访问权属性”在程序中用变量名 **access_right** 或 **ar** 表示。因为12位段属性中的高4位放在limit_high的高4位中，所以程序里有意把 **ar** 当作如下的16位构成来处理：

```
xxxx000xxxxxxxxx //其中x是0或者1
```

ar的高4位被称为“扩展访问权”。因为在80286时代还不存在，所以这样说。ar的低8位在80286时代就已经有了。这里不进行详细说明，只进行简单的介绍。

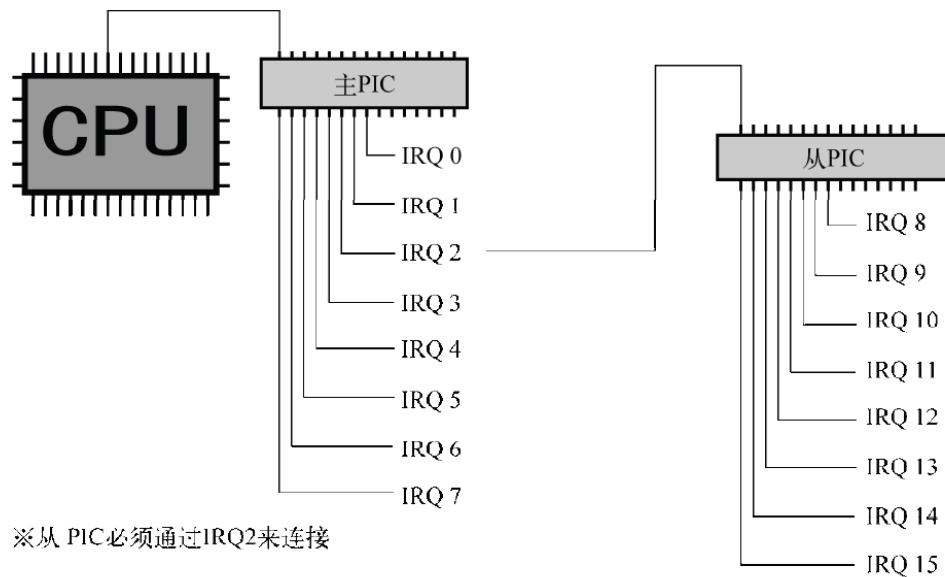
00000000（0x00）：未使用的记录表（descriptor table） 10010010（0x92）：系统专用，可读写的段。不可执行。 10011010（0x9a）：系统专用，可执行的段。可读不可写。 11110010（0xf2）：应用程序用，可读写的段。不可执行。 11111010（0xfa）：应用程序用，可执行的段。可读不可写。

CPU到底是处于系统模式还是应用模式，取决于执行中的应用程序是位于访问权为0x9a的段，还是位于访问权为0xfa的段。

初始化PIC

什么是PIC，为什么要初始化PIC呢？接着上次的程序，要做鼠标的移动，为了这个目的就必须使用中断，而使用中断，首先要做的就是将GDT和IDT正确的设定。但是还没有完，还要完成PIC的初始化。所谓的PIC是“programmable interrupt controller”的缩写，意思是“可编程中断控制器”。为什么呢要用这个PIC呢？因为CPU单独智能处理一个中断，这对计算机来说是远远不够的，所以引入了PIC。

PIC将8个中断信号（interrupt request 缩写IRQ）集成一个中断信号的装置。整个装置如下图所示：



后来的中断信号设置成了15个，并为此增设2个PIC。其中与CPU直接相连的PIC称为主PIC(master PIC)，与主PIC相连的是从PIC(slave PIC)。所谓“主从”是因为如果主PIC不通知CPU，从PIC的信息也就不能传给CPU。主PIC负责处理第0-7号中断，从PIC负责处理第8-15号中断信号。另外从图上看从PIC规定通过第2号IRQ与主PIC相连。

中断处理程序的制作