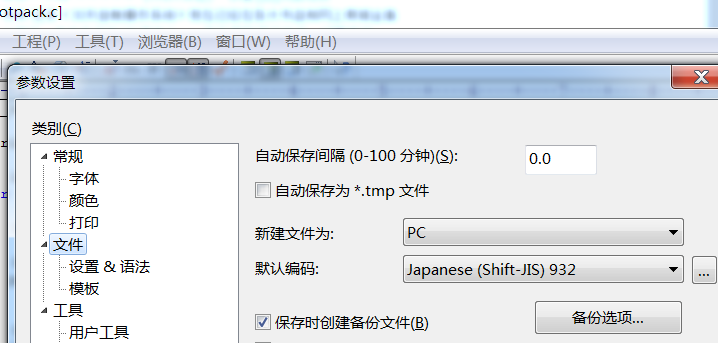
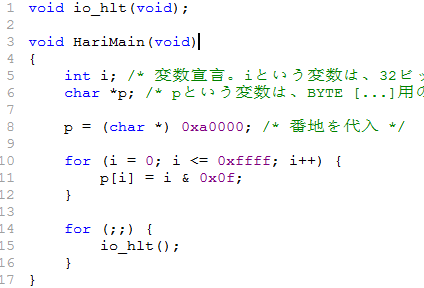
**参考：**

http://community.osdev.info/index.php?%28AT%29BIOS

# (1.19)第一天

编辑器使用EditPlus。

代码里的注释，全部是使用的日文……。因此想要看的话，editplus需要设置编码。**工具――参数设置**。默认编码改成如图：Japanese(Shift-JIS)。然后重新打开文件即可。

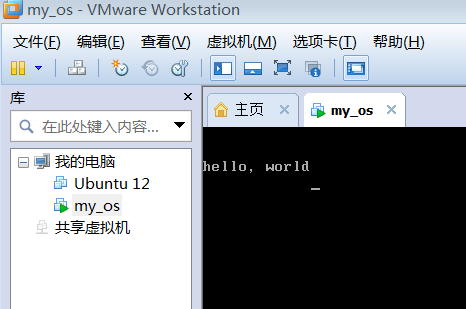
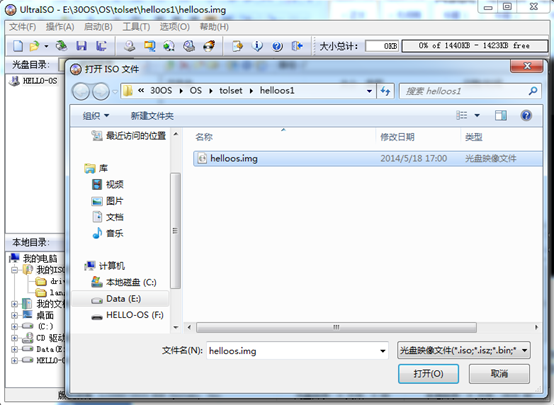
还有种方式是直接将文件拖到Chrome中，再修改Chrome的编码为SHIFT\_JIS。显示出日文后，再copy到百度的在线翻译中，即可转换成中文。

## 启动方式

**虚拟机。**

参考：<http://www.cnblogs.com/bitzhuwei/p/OS-in-30-days-03-use-vmware.html>

使用Floppy，关掉光驱。先创建空的系统，再选择软驱映像文件hello.img。

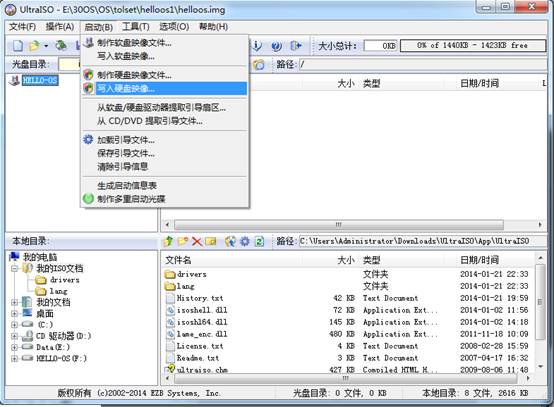
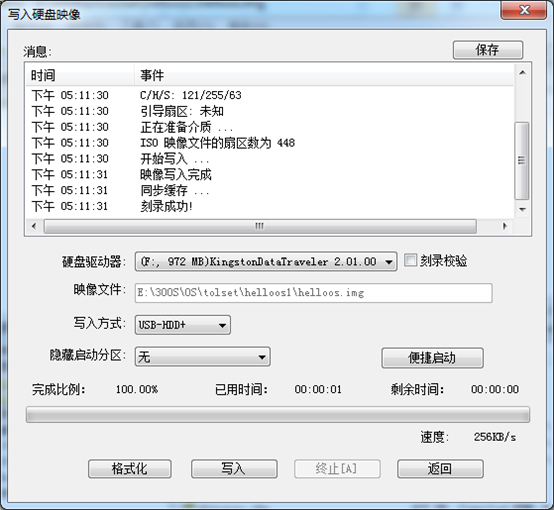
## 製作U盤。

使用**UltraISO**将img文件制作成引导盘。

在ultraiso中打开img文件。

首先，准备一个U盘，保证里面的数据你都不需要了。

然后，下载一个**UltraISO**软件。运行之，打开刚刚生成的helloos.img，如上右图所示。

然后，使用菜单项"**启动-写入硬盘映像…**"，如上左图所示。

在弹出的窗口中，确定"**硬盘驱动器**"中选中的是你准备好的U盘，然后点击"**写入**"按钮，稍候即可，如上右图所示。

注意：如果映像盘启动不成功。可能考虑在写入时，在右边的“**便捷启动**”项中“**写入新的驱动器引导扇区**”，选择合适项。

**U盘启动**

启动时，进入BIOS，做如下2个配置：

1. 在Advanced大项中的USB选项中，将Mass Storage项中，将USB设置成 **Forced FDD**（Floppy Disk Driver）。而不是Auto.
2. 启动顺序。选择从U盘启动，且不要选择UEFI开头的那个U盘项。

这样就可以成功了。

## Helloos1.nas

原始代码与helloos2.nas的解释：

DB 0xeb, 0x4e, 0x90, 0x48, 0x45, 0x4c, 0x4c, 0x4f #name of sector=”helloipl”

DB 0x49, 0x50, 0x4c, 0x00, 0x02, 0x01, 0x01, 0x00 #sector size=512B; cluster size =1

# FAT start selector = 1

DB 0x02, 0xe0, 0x00, 0x40, 0x0b, 0xf0, 0x09, 0x00 #FAT num=2; 根目录大小224

#disk size=2880 sector; disk type; FAT length=9 sector

DB 0x12, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00 #track/sector=18 header num=2

DB 0x40, 0x0b, 0x00, 0x00, 0x00, 0x00, 0x29, 0xff #disk size=2880 sec(again) #unknown

DB 0xff, 0xff, 0xff, 0x48, 0x45, 0x4c, 0x4c, 0x4f #label?? #disk name=”hello-os ”,11B

DB 0x2d, 0x4f, 0x53, 0x20, 0x20, 0x20, 0x46, 0x41 # disk format type=’fat12 ‘

DB 0x54, 0x31, 0x32, 0x20, 0x20, 0x20, 0x00, 0x00 #

RESB 16

DB 0xb8, 0x00, 0x00, 0x8e, 0xd0, 0xbc, 0x00, 0x7c # **main body**

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 # two LF(line feed)

DB 0x6c, 0x6c, 0x6f, 0x2c, 0x20, 0x77, 0x6f, 0x72 # “hello, world”

DB 0x6c, 0x64, 0x0a, 0x00, 0x00, 0x00, 0x00, 0x00 # LF(line feed)

RESB 368

DB 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x55, 0xaa # boot sec END(55AA)(addr:1FE,1FF)

DB 0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00 # 启动区以外部分的输出

RESB 4600

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

RESB 1469432

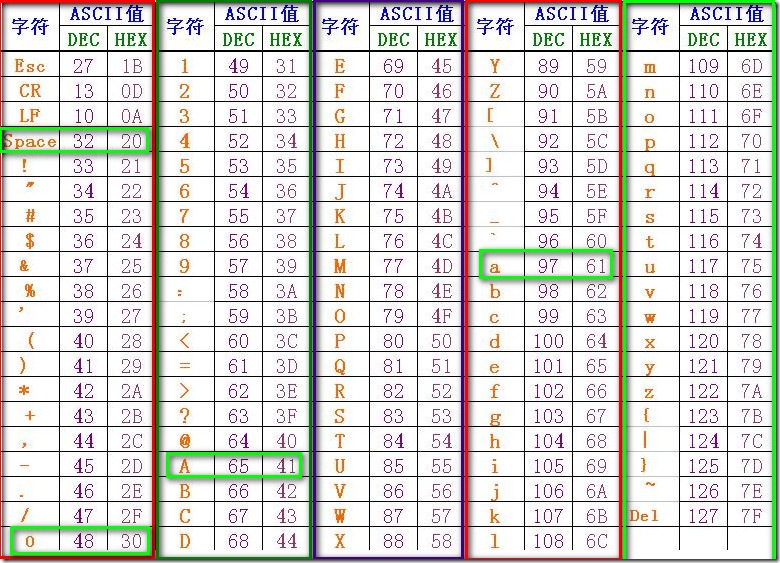
名词：

Boot sector: 盘分为512字节大小，称为扇区。软盘的第一个扇区称为启动区。启动扇区的最后两个字节必须是55AA。软盘为1440KB。有2880个扇区。

IPL：initial program loader。启动程序加载器。用于加载第一扇区之外的内容。

Boot: bootstrap的缩写。是自力更生的完成任务的意思。

## ASCII码表



hello: 0x48, 0x45, 0x4c, 0x4c, 0x4f

HELLO: 0x68, 0x65, 0x6c, 0x6c, 0x6f,

# (1.20)第二天

## Helloos3.nas

**; hello-os**

**; TAB=4**

**ORG 0x7c00 ; origin of 程序装载到内存的该地址**

**; 以下的记录用于标准的FAT12格式的软盘。**

**JMP entry**

**DB 0x90**

**DB "HELLOIPL" ; 启动区的名称，8B**

**DW 512 ; 扇区大小=512B**

**DB 1 ; cluster size = 1sector**

**DW 1 ; FAT start sector = 1**

**DB 2 ; number of FAT?? =2**

**DW 224 ; num of root path = 224**

**DW 2880 ; disk size = 2880 sector**

**DB 0xf0 ; disk type**

**DW 9 ; FAT length = 9 sec**

**DW 18 ; sector per track=18**

**DW 2 ; header=2**

**DD 0 ; not used=0**

**DD 2880 ; disk size = 2880 sector(backup)**

**DB 0,0,0x29 ; unknow 3B**

**DD 0xffffffff ; ???**

**DB "HELLO-OS " ; 11B, name of disk =**

**DB "FAT12 " ; 8B, disk format type**

**RESB 18 ;**

**; main body**

**entry:**

**MOV AX,0 ; initial reg， AX=0**

**MOV SS,AX**

**MOV SP,0x7c00**

**MOV DS,AX**

**MOV ES,AX**

**MOV SI,msg ; msg mean it’s mem addr(base on ORG)**

**putloop:**

**MOV AL,[SI] ; [SI] mean addr pointered by SI**

**ADD SI,1 ; SI=SI+1**

**CMP AL,0 ; 为什么是AL与0比较？hello, world后面有一个回车，**

**;以及一个值为0的字符。该0用于判断输出结束**

**JE fin**

**MOV AH,0x0e ; 显示一个字符; character code in AL.**

**MOV BX,15 ; 指定字符颜色**

**INT 0x10 ; 调用显卡BIOS**

**JMP putloop**

**fin:**

**HLT ; 让CPU等待，睡眠状态**

**JMP fin ; 无限循环**

**msg:**

**DB 0x0a, 0x0a ; 换行两次**

**DB "hello, world"**

**DB 0x0a ; 换行**

**DB 0 ; 用于判断输出结束**

**RESB 0x7dfe-$** ; 填写0x00，直到0x7dfe地址**前**（相对于0x7c00，

; 此即在启动扇区保留1FE,1FF地址，）

**DB 0x55, 0xaa**

**; 以下是启动区外部分的输出**

**DB 0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00**

**RESB 4600**

**DB 0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00**

**RESB 1469432**

第一个扇区的内容，被加载到0x7c00-0x7dff这512B 的空间内。

## Helloos5.nas

仅保留原来文件中，启动扇区部分。并准备新的make.bat及Makefile文件。

**type make.bat:**

..\z\_tools\make.exe %1 %2 %3 %4 %5 %6 %7 %8 %9

**type Makefile:**

# default action for make, means **make img**

default :

../z\_tools/make.exe img

# gen bin file: ipl.bin based on ipl.nas, ane output ipl.lst for transfer info.

ipl.bin : ipl.nas Makefile

**../z\_tools/nask.exe ipl.nas ipl.bin ipl.lst**

# gen image file: helloos.img based on ipl.bin.

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

# gen ipl.bin

asm :

../z\_tools/make.exe -r ipl.bin

# gen helloos.img

img :

../z\_tools/make.exe -r helloos.img

# start to run the 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

# delete tmp file and output file

src\_only :

../z\_tools/make.exe clean

-del helloos.img

### 测试运行

打开!cons\_nt.bat命令，执行命令如下：

**make –r ipl.bin** # gen ipl.bin and ipl.lst

**make –r helloos.img** # gen helloos.img

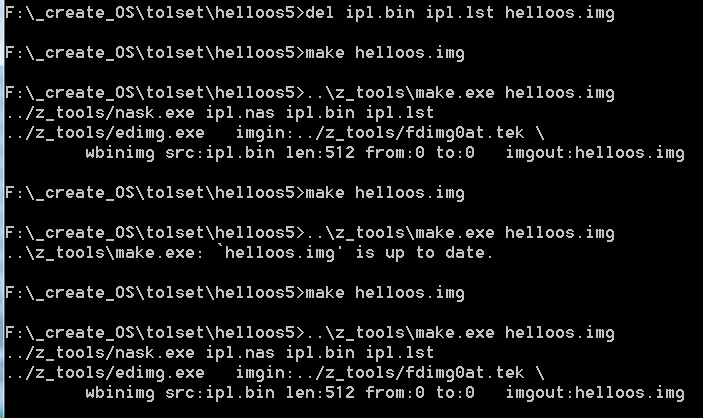
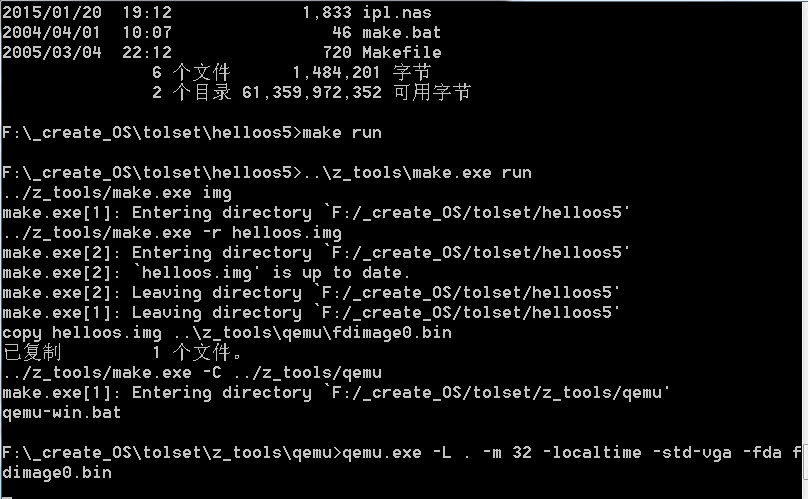
**del ipl.bin ipl.lst helloos.img**  # for try agian

**make –r helloos.img** # gen helloos.img directly

**make –r helloos.img** # ‘helloos.img’ is up to date.

# modify ipl.nas file. “hello, world” -> “how are you?”

**make –r helloos.img** # gen helloos.img again

使用封装命令：

# 只保留ipl.nas, make.bat, Makefile三个文件

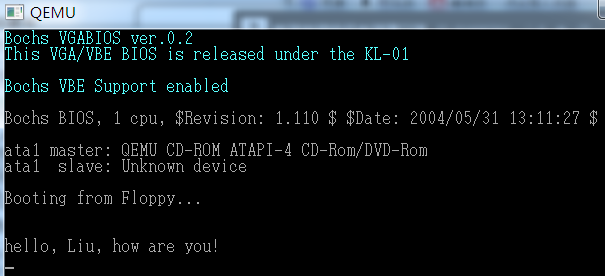
make img

dir # 已经多了ipl.bin and ipl.lst, helloos.img

make clean # delete ipl.bin and ipl.lst

make # use default

make src\_only # delete ipl.bin and ipl.lst, helloos.img



### ipl.lst信息

1 00000000 ; hello-os

2 00000000 ; TAB=4

3 00000000 ;

4 ORG 0x7c00 ;

5 00007C00 ;

6 00007C00 ;

7 00007C00 ;

8 00007C00 **EB 4E** JMP entry ;

9 0000**7C02** 90 DB 0x90 ;

10 00007C03 48 45 4C 4C 4F 49 50 4C DB "HELLOIPL" ;

11 00007C0B 0200 DW 512 ;

12 00007C0D 01 DB 1;

13 00007C0E 0001 DW 1;

14 00007C10 02 DB 2;

15 00007C11 00E0 DW 224 ;

16 00007C13 0B40 DW 2880 ;

17 00007C15 F0 DB 0xf0 ;

18 00007C16 0009 DW 9;

19 00007C18 0012 DW 18 ;

20 00007C1A 0002 DW 2;

21 00007C1C 00000000 DD 0;

22 00007C20 00000B40 DD 2880 ;

23 00007C24 00 00 29 DB 0,0,0x29 ;

24 00007C27 FFFFFFFF DD 0xffffffff ;

25 00007C2B 48 45 4C 4C 4F 2D 4F 53 20 20 DB "HELLO-OS " ;

00007C35 20

26 00007C36 46 41 54 31 32 20 20 20 DB "FAT12 " ;

27 00007C3E 00 00 00 00 00 00 00 00 00 00 RESB 18 ;

00007C48 00 00 00 00 00 00 00 00

28 00007C50 ;

29 00007C50 ;

30 00007C50 ;

31 00007C50 entry: ;

32 00007C50 B8 0000 MOV AX,0 ;

33 00007C53 8E D0 MOV SS,AX

34 00007C55 BC 7C00 MOV SP,0x7c00 ;

35 00007C58 8E D8 MOV DS,AX ;

36 00007C5A 8E C0 MOV ES,AX ;

37 00007C5C ;

38 00007C5C BE 7C74 MOV SI,msg ;

39 0000**7C5F** putloop: ;

40 00007C5F 8A 04 MOV AL,[SI] ;

41 00007C61 83 C6 01ADD SI,1 ;

42 00007C64 3C 00 CMP AL,0 ;

43 00007C66 74 09 JE fin ;

44 00007C68 B4 0E MOV AH,0x0e ;

45 00007C6A BB 000F MOV BX,15 ;

46 00007C6D CD 10 INT 0x10 ;

47 0000**7C6F** EB EE JMP putloop ;

48 0000**7C71** fin: ;

49 00007C71 F4 HLT ;

50 00007C72 EB FD JMP fin ;

51 00007C74

52 0000**7C74** msg: ;

53 00007C74 0A 0A DB 0x0a, 0x0a ;

54 00007C76 68 65 6C 6C 6F 2C 20 4C 69 75 DB "hello, Liu, how are you!"

00007C80 2C 20 68 6F 77 20 61 72 65 20

00007C8A 79 6F 75 21

55 00007C8E 0A DB 0x0a ;

56 00007C8F 00 DB 0

57 00007C90 ;

58 00007C90 00 00 00 00 00 00 00 00 00 00 RESB 0x7dfe-$ ;

59 00007DFE

60 0000**7DFE** 55 AA DB 0x55, 0xaa ;

为什么开头的两个字节是**0xeb, 0x4e？**。

该语句是：JMP entry ;。其中EB是JMP的指令，且是无条件短跳转。第二个字节是跳转时IP的偏移量，实际上entry是在内存的**0x7c50**地址。关于IP需要注意的是，每条指令取出后，IP的值会指向下一条指令的地址。当取完该跳转全集后，IP指向了0x7c02地址。然后发现是跳转，需要偏移0x4e，故IP被修改成**0x7c50**地址。

为什么47行的跳转指令偏移是EE？

47 0000**7C6F** EB EE JMP putloop ;

其目标地址是0000**7C5F**，执行7C6F地址的全集时，IP中的地址是7C71。

EE=11101110，最高位是1，表示是负跳转。EE取反加1后得10010即0x12(表示-18)。则0x7C71-0x12=**0x7C5F**。

# (1.21)第三天

软盘知识：

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

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。

## hariboa

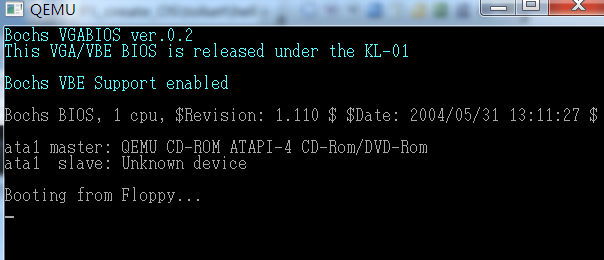
这个程序正常状态下，是先从第二个扇区读入数据到0x8200-0x83ff地址空间，然后根据读数据后的状态，使用JC做判断，读取正常后就进入到HALT状态，如果出错，则输出“load error”错误。通常是没有输出的，即Booting from Floppy下行为空。

需要明确**缓冲区**与段寄存器。

缓冲区。此处将ES置为0x0820，ES:BX组合成缓冲区地址。

ES是辅助段寄存器（Extra segment），用于指定大致范围，BX是基址，用于确定偏移，真实缓冲区地址＝ES\*16+BX。当ES=0X0820, BX=0，真实内存地址为0X8200。

后面两个例子，分别增加读取磁盘5次试错，以及读到18扇区（连读17个sec）。使用了SI控制出错，CL控制扇区数。



## 读到10个柱面

此处需要注意，书上P48页上的图，也就是扇区的编号顺序。C: cylinder, H: heade, S:

Sector，具体是这样的：

C0-H0-S1 --> C0-H0-S2 --> … C0-H0-S17 --> C0-H0-S18 --> # C0-H0共18个sector（正面）

C0-H1-S1 --> C0-H1-S2 --> … C0-H1-S17 --> C0-H1-S18 --> # C0-H1共18个sector（反面）

C1-H0-S1 --> C1-H0-S2 --> … C1-H0-S17 --> C1-H0-S18 --> # C1-H0共18个sector（正面）

C1-H1-S1 --> C1-H1-S2 --> … C1-H1-S17 --> C1-H1-S18 --> # C1-H1共18个sector（反面）

使用的寄存器：

缓冲区：ES:BX

driver no: DL=0x00 ( floppy A) cylinder: CH, header: DH, sector: CL

read cmd: AH=0x02, read sector num: AL=1

retry count: SI sector count: CL

**软盘空间与内存空间对应位置。**

根据书上P49页知识，IPL从第2个扇区开始读软盘，最开始的地址，根据ES=0X0820及BX=0，地址为0X8200.

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 扇区位置 | 内存位置 | 备注 |
| 1 | C0-H0-S1 | **0x07C00 ~ 0x07DFF** | 启动引导扇区 |
| 2 | C0-H0-S2 | 0x08200 ~ 0x083FF | 2号扇区 |
| 3 | C0-H0-S3 | 0x08400 ~ 0x085FF | 3号扇区 |
| 4 | C0-H0-S18 | **0x0A200 ~ 0x0A3FF** | 18号扇区 |
| 5 | C0-H1-S1 | 0x0A400 ~ 0x0A5FF | 19号扇区 |
| 6 | C0-H1-S2 | 0x0A600 ~ 0x0A7FF | 20号扇区 |
| 360 | C9-H1-S18 | 0x34E00 ~ 0X34FFF | 360号扇区 |

**流程图：**



**改进**

由于原代码在读取正常后，没有任何输出。本次增加以正常输出部分。

在CMP CH,CYLS代码两行后，与fin之间增加：

success:

MOV SI,success\_msg

suc\_putloop:

MOV AL,[SI]

ADD SI,1 ; SI偵1傪懌偡

CMP AL,0

JE fin

MOV AH,0x0e ;

MOV BX,15 ;

INT 0x10 ;

JMP suc\_putloop

success\_msg:

DB 0x0a, 0x0a ;

DB "hello, my OS!"

DB 0x0a ;

DB 0

## 开发haribote.sys

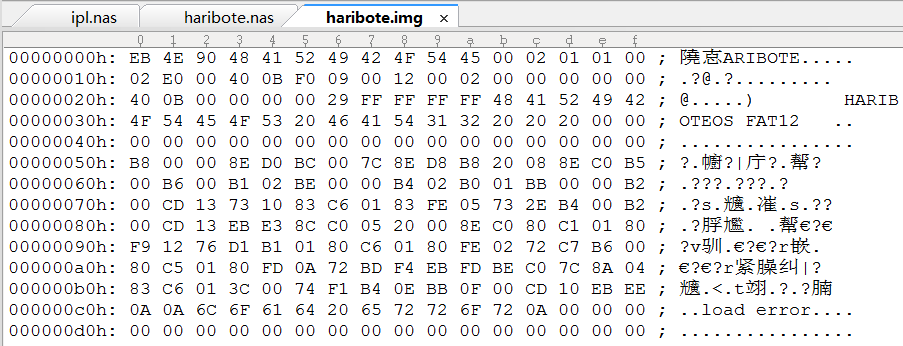
此处使用haribof示例。

从现在开始，将操作系统文件独立成haribote.nas，原来的引导程序继续名为ipl.nas。

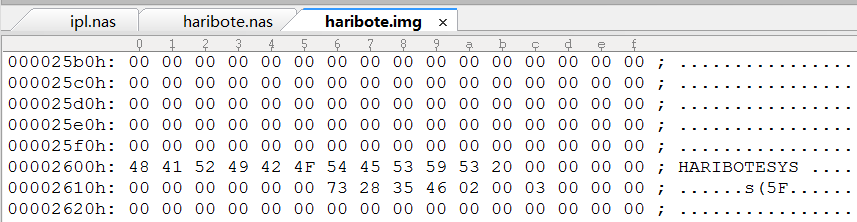
ipl.nas ---- > ipl.bin

haribote.nas ---- > haribote.sys （再将ipl.bin + haribote.sys写成映像haribote.img）

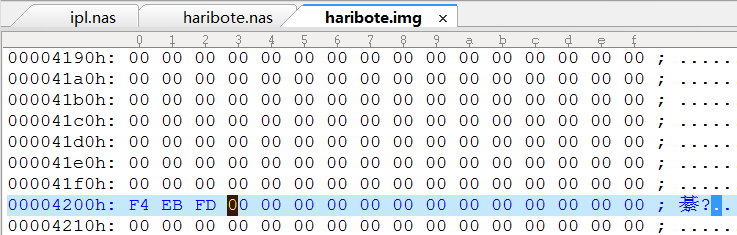
查看映像文件内容：



引导扇区开始部分



0x2600地址——文件名



0x4200地址——文件内容（只有6个字节：F4 EB FD）

0x4200地址的内容。即：

1 00000000 fin:

2 00000000 **F4**  HLT

3 00000001 **EB FD**  JMP fin # FD=0b11111101 = **-3**

上面的程序启动后，启动区会加载10个柱面的数据到内。其中第一个扇区加载到0x7c00开始的512字节，第二个扇区加载到0x8200开始的地址（第一个扇区并不是我们写的IPL程序加载的，相当于第一个扇区是从0x8000地址开始的）。

这样，上述0x4200软盘地址的代码，被加载到内存的0x8000+0x4200=**0xC200**地址。因此，IPL在加载完10个柱面的数据后，直接JMP到C200地址。

## 设置显卡模式

代码如haribog

使用VGA显卡，320\*200\*8位彩色。

注意，JMP前，多了条MOV指令如下：

MOV [0x0ff0],CH ; IPL将加载完后CH中的cylinder数告诉给haribote.sys

JMP 0xc200

# 不明白为什么要写向**0x0FF0地址**？

前后差别，分别在模拟器及虚拟器中测试：

|  |  |  |
| --- | --- | --- |
|  | 以前 | 有显卡 |
| 虚拟机 |  |  |
| 模拟器 |  |  |

## boot\_info

hariboh目录

boot info，本处主要包括：

启动区柱面信息；画面颜色位数；分辨率（x, y）；VRAM。

VRAM指显卡内存开始地址。本处为**0xA0000**（到0xAFFFF,共64K）。不同分辨率的VRAM是不同的。

## use C program

代码目录hariboi。没有使用HLT指令，整个系统一起循环跳转，会占掉一个CPU核。

hariboj则增加了io\_hlt函数，通过HLT指令，系统可以睡眠，不耗CPU。

编译关系



gas2nask：C使用gcc编译，其基于gas汇编。此工具将其转为nasm汇编。

obj2bim：bim是binary image，obj2bim用于将多个obj文件链接成一个机器语言文件。

bim2hrb：hrb is haribote，将obj及头文件等整合在一起。

io\_hlt()，这个函数在naskfunc.nas中使用汇编定义的。只包含两条命令HLT, RET。语法格式不明白。似乎是NAS的语法要求global声明的串，需要以”\_”开关，这样链接器才能找到。

# (1.22)第四天

## 1 写显存

harib01a目录。

**naskfunc.nas:**

GLOBAL \_io\_hlt,\_write\_mem8

[SECTION .text]

**\_io\_hlt**: ; void io\_hlt(void);

HLT

RET

**\_write\_mem8**: ; void write\_mem8(int addr, int data);

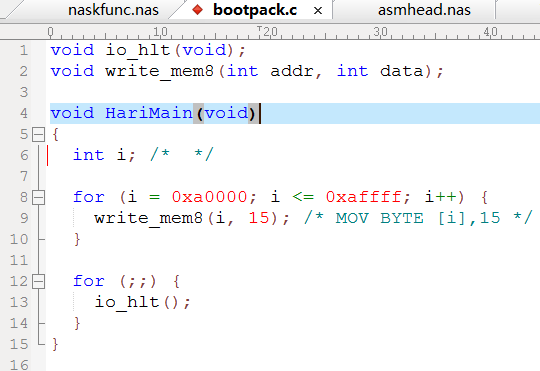
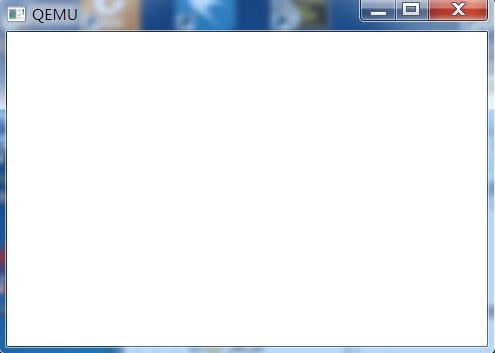
MOV ECX,[ESP+4] ; [ESP+4] addr

MOV AL,[ESP+8] ; [ESP+8] data

MOV [ECX],AL

RET

**bootpack.c**

左图为代码，右图为运行结果，屏幕由之前的全黑变成全白色了。

## 2 条纹图案

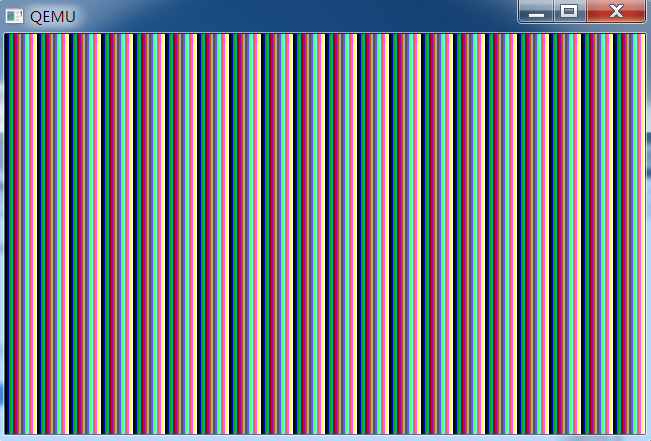
将bootpack.c中的输出for循环改成如下：

for (i = 0xa0000; i <= 0xaffff; i++) {

write\_mem8(i, i & 0x0f); // 15 -- > I & 0x0f

}

结果：



PS：CPU除了做加法，还可以做逻辑操作，常用的三个：

置0： OR bit1

置1：AND bit0

反转：XOR bit1

## 3 挑战指针

将bootpack.c中的输出for循环改成如下：

for (i = 0xa0000; i <= 0xaffff; i++) {

// write\_mem8(i, i & 0x0f); // 15 -- > I & 0x0f

**char \*p;**

**p = i;**

**\*p = i & 0x0f;**

}

//使用char \*p，是因为我们要按字节写显存，char指针正好是BYTE型的。

// **char \*p; BYTE型地址; short \*p; WORD型地址; int \*p; DWORD型地址**。

本次也可以正常输出条纹了。但是，编译时多了条警告。

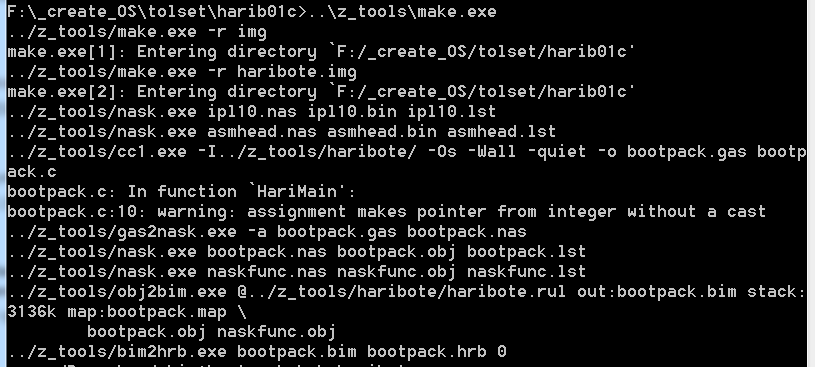
编译HariMain时出现了：

**warning: assignment makes pointer from integer without a cast.**

赋值语句直接使用整数生成了指针，没有经过转换（cast）。问题出在: **p = i;**

p是指针，i是整数。需要进行一次类型转换，正确写法： **p = ( char \* ) i;**

这是告诉编译器转换类型。



### 关于指针

**char \*p = (char \*) i;**

**\*p = i & 0x0f**

这两行其实等于： **\*(** **(char \*) i ) = i & 0x0f**

也就是可以不用变量p的出现。其实质就是汇编里的： **BYTE [i] = i & 0x0f**

关于理解指针左边的C代码与右边的汇编指令等价。

|  |  |
| --- | --- |
| **int i = 0xa000;**  **char \*p ;**  **p = (char \*) i;**  **\*p = i & 0x0f;** | **MOV ECX, I**  **MOV BYTE [ECX] (i & 0x0f)** |

p=这句，是给ECX寄存器赋值，在CPU内操作，

\*p这句，是给[ECX]号内存地址赋值。在内存中操作。两者执行顺序不能轻换

那么，当定义char x;时，我们可以使用一个变量x，但当定义char \*p;时，我们即可以使用p，也可以使用\*p，这是两个变量吗？ 其实回到上面的汇编指令，只有一个变量ECX，\*p并不是变量，p才是变量，\*p是汇编中 BYTE [p]的代替。

那么**char \*p;** 定义的是\*p or p？是p。也可以定义成**char\* p;** 只是要同时定义多个变量时后者不方便理解。如char\* p, q; 此时q是1字节变量，p是地址变量，若两都都要定义成地址变量，应该char\* p, \*q。故通常\*写变量相邻，不与类型char,int等相邻。

当要改变地址变量的值以指向新的地址，并赋值时，有几种方式是等价的。

|  |
| --- |
| char \*p = **0xa000; int i;**   1. **\*( p + i ) = 0xff;** 2. **\*( i + p ) = 0xff;** 3. **p[i] = 0xff;** 4. **i[p] = 0xff;** |

其中数组形式的写法，不改理解成数组，其实对于数组，**a[3]=0xff，与3[a]=0xff**是一样等价的。神奇吧。对于数组的定义，;其等价的汇编是：

|  |  |
| --- | --- |
| **char a[10];** | **a:**  **RESB 10** |

所以，a其实就是个常量，是一个汇编的label，代表的是一个地址。或者说是一段内存空间的起始地址。

下面做一个实验：

|  |  |
| --- | --- |
|  |  |

## 4 设定色号

本次使用的320x200的8位彩色模式。只有255种颜色。颜色使用RGB方式，6个16进制数，即24bit表示RGB。

**调色板（palette）**定义色号（8bit）与颜色（24bit）的对应关系。

代码在**harib01f**目录。

直接运行的局部结果（左为原始条纹图案，中为本次调色板）：

|  |  |  |
| --- | --- | --- |
|  |  |  |

最右图是将C代码中的io\_out8(0x03c9, rgb[0] / 4); 除4去掉后的效果。即：io\_out8(0x03c9, rgb[0]);

还是不太明白调色板定义的颜色的意义。

### bootpack.c文件：

**init\_palette**函数：

static unsigned char table\_rgb[48]定义的颜色，static相当于汇编的DB指令。

调用set\_palette(0, 15, table\_rgb);。

**set\_palette**(int start, int end, unsigned char \*rgb)：

eflags = io\_load\_eflags() //保存FLAGS寄存器

io\_cli(); // 关中断

//后面的io\_out8相关操作是写显存。写过程为：

// 将调色板号码写入0x03c8， 然后按R,G,B的顺序写入0X03C9。

// 若顺序写，后面可以省略调色板号，直接按RGB写入0X03C9即可。

io\_out8(0x03c8, start); // io\_out8在naskfunc.nas中定义，向指定的地址传送数据，

# 向0x03c8地址写入0号调色板号

io\_out8(0x03c9, rgb[0] / 4); # 没明白这里为什么要除4呢？

io\_store\_eflags(eflags); // 恢复FLAGS寄存器中的现场，

**关于中断。**

关中断：CLI（clear interrupt flag）

开中断：STL（set interrupt flag）

### naskfunc.nas文件

问题：为什么每个汇编函数中，读堆栈地址时，都要+4呢？这应该涉及到函数的调试与现场环境的保存和恢复？

\_io\_out8: ; void io\_out8(int port, int data);

MOV EDX,[ESP+4] ; port, also means address

MOV AL,[ESP+8] ; data

OUT DX,AL

RET

注意，数据输入输出时，地址放在DX，数据放在AX(or AL)。

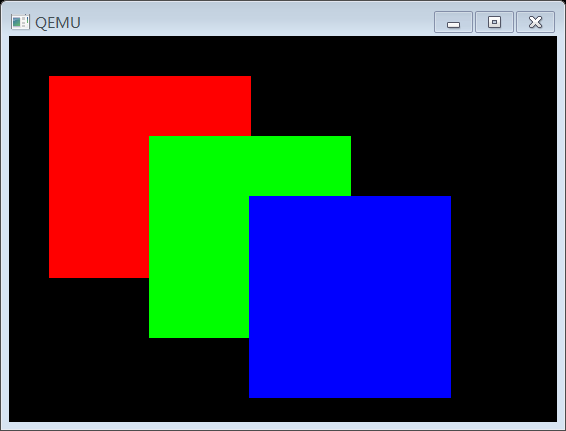
IN: IN AX,DX

OUT: OUT DX,AX

## 5 绘制矩阵

视频像素点的坐标关系。如下左图。320\*200＝64000个像素。

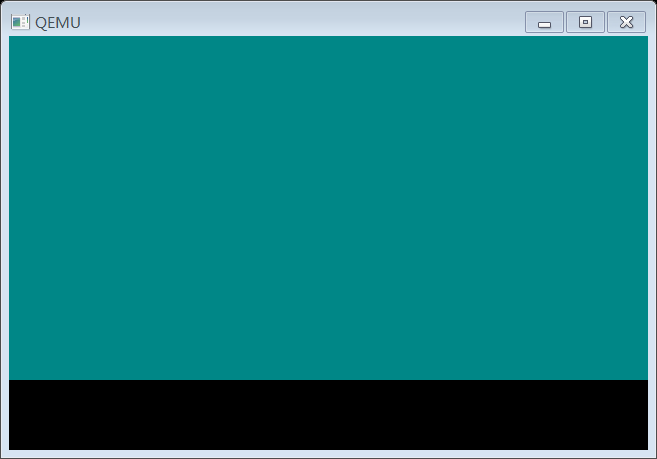
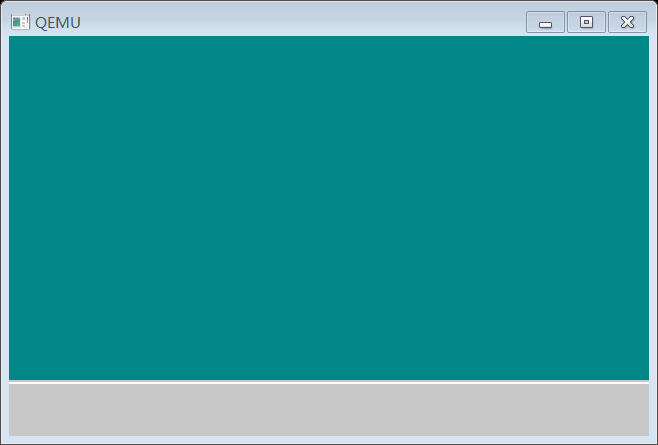
像素(x,y)对应的VRAM地址＝**0xa000 + x + y \* 320**。

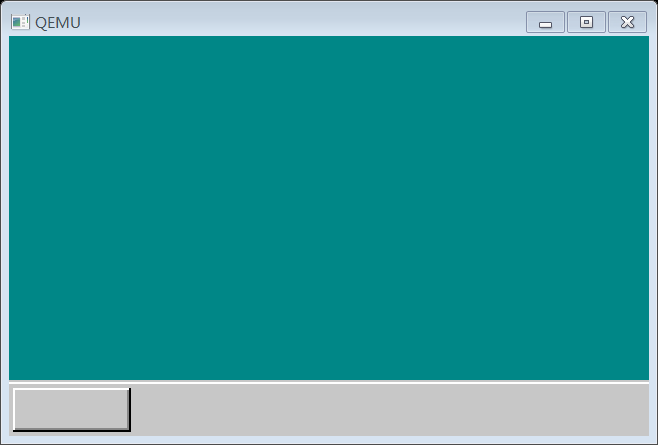
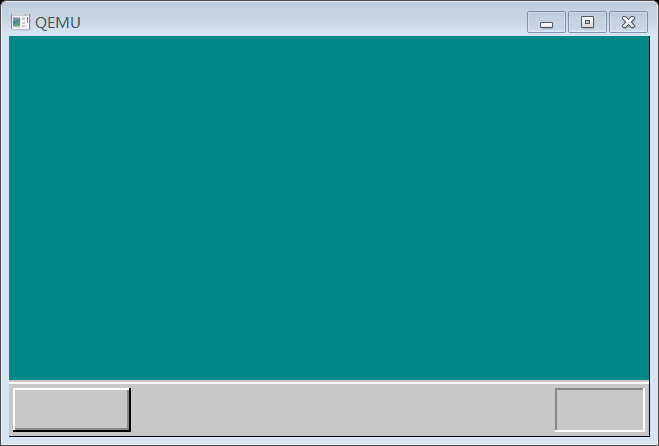
绘制矩阵，实际上是重复画线条。使用函数来完成。

|  |
| --- |
| 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;  } |

## 6 滚动条

先画最大的背景， 画下面的条

画下面条左边的滚动条 画下边条右下角凹的印

# 附录

## X86 Register

EAX, AX, AH, AL

## 常用windows命令

AT 计划在计算机上运行的命令和程序。  
**ATTRIB 显示或更改文件属性。**BREAK 设置或清除扩展式 CTRL+C 检查。  
CACLS 显示或修改文件的访问控制列表(ACLs)。  
CALL 从另一个批处理程序调用这一个。  
**CD 显示当前目录的名称或将其更改。**CHCP 显示或设置活动代码页数。  
**CHDIR 显示当前目录的名称或将其更改。  
CHKDSK 检查磁盘并显示状态报告。**CHKNTFS 显示或修改启动时间磁盘检查。  
**CLS 清除屏幕。  
CMD 打开另一个 Windows 命令解释程序窗口。**COLOR 设置默认控制台前景和背景颜色。  
**COMP 比较两个或两套文件的内容。**COMPACT 显示或更改 NTFS 分区上文件的压缩。  
CONVERT 将 FAT 卷转换成 NTFS。您不能转换当前驱动器。  
**COPY 将至少一个文件复制到另一个位置。  
DATE 显示或设置日期。**

**DEL 删除至少一个文件。  
DIR 显示一个目录中的文件和子目录。**DISKCOMP 比较两个软盘的内容。  
DISKCOPY 将一个软盘的内容复制到另一个软盘。  
DOSKEY 编辑命令行、调用 Windows 命令并创建宏。  
**ECHO 显示消息，或将命令回显打开或关上。**ENDLOCAL 结束批文件中环境更改的本地化。  
ERASE 删除至少一个文件。  
**EXIT 退出 CMD.EXE 程序(命令解释程序)。**FC 比较两个或两套文件，并显示不同处。  
FIND 在文件中搜索文字字符串。  
FINDSTR 在文件中搜索字符串。  
FOR 为一套文件中的每个文件运行一个指定的命令。  
FORMAT 格式化磁盘，以便跟 Windows 使用。  
FTYPE 显示或修改用于文件扩展名关联的文件类型。  
GOTO 将 Windows 命令解释程序指向批处理程序中某个标明的行。  
GRAFTABL 启用 Windows 来以图像模式显示扩展字符集。  
**HELP 提供 Windows 命令的帮助信息。**IF 执行批处理程序中的条件性处理。  
LABEL 创建、更改或删除磁盘的卷标。  
**MD 创建目录。  
MKDIR 创建目录。**MODE 配置系统设备。  
MORE 一次显示一个结果屏幕。  
**MOVE 将文件从一个目录移到另一个目录。**PATH 显示或设置可执行文件的搜索路径。  
PAUSE 暂停批文件的处理并显示消息。  
POPD 还原 PUSHD 保存的当前目录的上一个值。  
PRINT 打印文本文件。  
PROMPT 更改 Windows 命令提示符。  
PUSHD 保存当前目录，然后对其进行更改。  
**RD 删除目录。**RECOVER 从有问题的磁盘恢复可读信息。  
REM 记录批文件或 CONFIG.SYS 中的注释。  
REN 重命名文件。  
RENAME 重命名文件。  
REPLACE 替换文件。  
**RMDIR 删除目录。  
SET 显示、设置或删除 Windows 环境变量。**SETLOCAL 开始批文件中环境更改的本地化。  
SHIFT 更换批文件中可替换参数的位置。  
SORT 对输入进行分类。  
START 启动另一个窗口来运行指定的程序或命令。  
SUBST 将路径跟一个驱动器号关联。  
TIME 显示或设置系统时间。  
TITLE 设置 CMD.EXE 会话的窗口标题。  
TREE 以图形模式显示驱动器或路径的目录结构。  
**TYPE 显示文本文件的内容。**VER 显示 Windows 版本。  
VERIFY 告诉 Windows 是否验证文件是否已正确写入磁盘。  
VOL 显示磁盘卷标和序列号。  
XCOPY 复制文件和目录树。

appwiz.cpl------------添加删除程序  
control userpasswords2--------用户帐户设置  
cleanmgr-------垃圾整理  
CMD--------------命令提示符可以当作是 Windows 的一个附件，Ping，Convert 这些不能在图形环境下 使用的功能要借助它来完成。  
cmd------jview察看Java虚拟机版本。  
command.com------调用的则是系统内置的 NTVDM，一个 DOS虚拟机。它完全是一个类似 Virtual PC 的 虚拟环境，和系统本身联系不大。当我们在命令提示符下运行 DOS 程序时，实际上也 是自动转移到 NTVDM虚拟机下，和 CMD 本身没什么关系。  
**calc-----------启动计算器**chkdsk.exe-----Chkdsk磁盘检查  
compmgmt.msc---计算机管理  
conf-----------启动 netmeeting  
control userpasswords2-----User Account 权限设置  
devmgmt.msc--- 设备管理器  
diskmgmt.msc---磁盘管理实用程序  
dfrg.msc-------磁盘碎片整理程序  
drwtsn32------ 系统医生  
dvdplay--------启动Media Player  
dxdiag-----------DirectX Diagnostic Tool  
gpedit.msc-------组策略编辑器  
gpupdate /target:computer /force 强制刷新组策略  
eventvwr.exe-----事件查看器  
explorer-------打开资源管理器  
logoff---------注销命令  
lusrmgr.msc----本机用户和组  
msinfo32---------系统信息  
msconfig---------系统配置实用程序  
net start (servicename)----启动该服务  
net stop (servicename)-----停止该服务  
**notepad--------打开记事本**nusrmgr.cpl-------同control userpasswords，打开用户帐户控制面板  
**Nslookup-------IP地址侦测器**oobe/msoobe /a----检查XP是否激活  
perfmon.msc----计算机性能监测程序  
progman--------程序管理器  
regedit----------注册表编辑器  
regedt32-------注册表编辑器  
regsvr32 /u \*.dll----停止dll文件运行  
**route print------查看路由表**rononce -p ----15秒关机  
rsop.msc-------组策略结果集  
rundll32.exe rundll32.exe %Systemroot%System32shimgvw.dll,ImageView\_Fullscreen----启动一个空白的Windows 图片和传真查看器  
secpol.msc--------本地安全策略  
services.msc---本地服务设置  
sfc /scannow-----启动系统文件检查器  
sndrec32-------录音机  
taskmgr-----任务管理器（适用于2000／xp／2003）  
tsshutdn-------60秒倒计时关机命令  
winchat--------XP自带局域网聊天  
winmsd---------系统信息  
winver-----显示About Windows 窗口  
wupdmgr-----------Windows Update

# 他人笔记

## 高有用

2015年6月8日增加如下2个：

原书的github:

<https://github.com/Tassandar/OSASK>

复旦 刘宇达

<http://lengly.top/?p=85>

<https://github.com/lengly/myOS>

30天自制操作系统 For Linux

<http://www.ituring.com.cn/article/30667>

# 内容见本文档最后。

<https://github.com/zchrissirhcz/osask-linux>

#### end 2015.0608

### 1 scusjs

**### 川大计算机，创新班，杨辉 学生，**

**### 支持windows及linux，有git账号。 博客从第3天开始，到19天。**

<http://blog.csdn.net/scusjs?viewmode=contents>

<http://blog.csdn.net/scusjs/article/details/8947918>

# code

<https://github.com/scusjs/MyOS>

https://github.com/scusjs/MyOS.git

### 2 cherishsir

**### ubuntu平台**

**### cherishsir 16天， QQ 交流Group:159336168**

http://segmentfault.com/blog/cherishsir/1190000000609485?page=1

http://segmentfault.com/blog/cherishsir/1190000000609473

<https://github.com/cherishsir/ubuntu230os.git>

**creat OS**

## 参考资料

http://verynix.com/os-dev-book.html

http://wiki.osdev.org/Main\_Page

## 中可用

###　**祝威** 到**15**天。用**vmware.** bitzhuwei@qq.com

http://www.cnblogs.com/bitzhuwei/p/OS-in-30-days-01-hello-bitzhuwei-OS.html

http://www.cnblogs.com/bitzhuwei/default.html?OnlyTitle=1

### 到6天，用virtualBox

http://www.cnblogs.com/lastavengers/

### Ubuntu 环境

<http://www.cnblogs.com/dyllove98/p/3235363.html>

qq交流群：1**22358078**，此群已经进入，主要是嵌入式开发的。

##第九天链接

http://www.haodaima.net/art/2251717

http://www.tuicool.com/articles/yMVZvy

# 知乎精选

http://www.zhihujingxuan.com/19197.html

一个操作系统的实现

### JamesM's kernel development tutorials

http://www.jamesmolloy.co.uk/tutorial\_html/

# for linux

这个文件是从我github下直接复制下来的,因为图灵社区也支持Markdown所以也在这发一份吧

#### 起因

最近在看30天自制操作系统,准备自己写一个小操作系统当做毕业设计的项目来做, 因为自己日常的环境是Linux, 但是书中用的操作系统是windows,

所以为了方便在Linux下开发,需要对一些东西进行修改

#### 工具

随书光盘中的z\_tools是在windows下环境下用的一些工具,已经有人制作出了对应的Linux工具了

[下载地址 z\_tools](http://hrb.osask.jp/z_tools.tar.bz2)

#### Makefile

在项目中的Makfile 也需要进行相应的修改

TOOLPATH = (工具的目录)

INCPATH = (工具的目录)

MAKE = make -r

NASK = $(TOOLPATH)nask

CC1 = $(TOOLPATH)cc1 -I $(INCPATH) -Os -Wall -quiet

GAS2NASK = $(TOOLPATH)gas2nask -a

OBJ2BIM = $(TOOLPATH)obj2bim

BIN2OBJ = $(TOOLPATH)bin2obj

BIM2HRB = $(TOOLPATH)bim2hrb

RULEFILE = $(TOOLPATH)haribote/haribote.rul

EDIMG = $(TOOLPATH)edimg

IMGTOL = $(TOOLPATH)imgtol.com

MAKEFONT = $(TOOLPATH)makefont

GOLIB = $(TOOLPATH)golib00

COPY = cp

DEL = rm

haribote.sys : asmhead.bin bootpack.hrb Makefile

cat asmhead.bin bootpack.hrb > haribote.sys

$(EDIMG) imgin:$(TOOLPATH)fdimg0at.tek \

哪个makefile中的qemu哪里有了噢？别的都明白！

直接在ubuntu 下安装 安装一个qemu 把Makefile下的run:改为: run : $(MAKE) img qemu-system-i386 -fda haribote.img –  [samael](http://www.ituring.com.cn/users/info/82105) 2013-02-21 19:16

是的，我也在网上找到了，但还是非常感谢你！http://d.hatena.ne.jp/big-eyed-hamster/20081229/1230562654 我在网络上搜索的，它其实也有Mac版的。谢谢大哥！ –  [4lvzhuyiyi4](http://www.ituring.com.cn/users/info/95203) 2013-02-21 19:20

我是用github写博客的, 因为都支持markdown所以就在这里又复制了一篇. 原文在这 http://samael65535.github.com/ –  [samael](http://www.ituring.com.cn/users/info/82105) 2013-02-21 19:18

该怎么用呢。怎么下载还exe文件呢 需要wine吗？

不用,你说的是那个make.exe吧, 那个只是链接到本地的make一个链接.直接 MAKE = make -r 就行 另外 haribote.rul 里的目录也要改 –  [samael](http://www.ituring.com.cn/users/info/82105) 2013-02-15 14:07