# 第3天

# 进入32位模式并导入C语言

- 制作真正的IPL
- 试错
- 读到18扇区
- 读入10个柱面
- 着手开发操作系统
- 从启动区执行操作系统
- 确认操作系统的执行情况
- 32位模式前期准备
- 开始导入C语言
- 实现HLT（harib00j）

## 1 制作真正的IPL

到昨天为止我们讲到的启动区，虽然也称为IPL（Initial Program Loader，启动程序装载器），但它实质上并没有装载任何程序。而从今天起，我们要真的用它来装载程序了。

把我们的操作系统叫作hello-os很不给劲，干脆改个名字吧。我们就叫它“纸娃娃操作系统”。所谓纸娃娃，意思就是说那是用纸糊起来的，虚有其表，不是真娃娃，就像拍电影时用的岩石等道具，其实都是中间空空的冒牌货。也就是说我们现在要开发的操作系统，只是看上去像操作系统，而其实是个没有内容的纸娃娃，所以大家不用想得太困难，轻轻松松来做就好了。

虽然今后我们要一直称它为“纸娃娃操作系统”，而且在相当长的一段时间里也只是把它当作一种演示程序，但到最后我们肯定能开发出一个像模像样的操作系统，这一点请大家放心。

稍微扯远点，其实仔细想一想，这种虚有其表的“纸娃娃”又何止是操作系统呢。就说CPU吧，其实它根本就不懂什么“数”的概念，只是我们设计了一个电路，只要同时传给它电信号0011和0110，它就能输出结果为1001的电信号，而这种电路我们就称之为加法电路。只有人才会把这个结果解读为3+6=9，CPU只是处理这些电信号。换句话说，虽然CPU根本就不懂什么数字，但却能给出正确的计算结果，这就是笔者所谓的“纸娃娃”。

开发过游戏程序的人就会明白，比如我们和计算机下象棋的时候，可能会觉得计算机水平很高，但实际上计算机对象棋规则一窍不通，仅仅是在执行一个程序而已。就算计算机走出了一着妙棋，也根本不是因为它下手毫不留情啊，或者聪明啊，或者求胜心切什么的，这

些都是表象，其实它只是按部就班地执行程序而已。也就是说它本身没有内涵，只有一个唬人的外壳，所以才叫“纸娃娃”。“纸娃娃”太厉害了！……操作系统就算是虚有其表、虚张声势又怎样？没什么不好的，这样就可以了！

那么我们先从简单的程序开始吧。因为磁盘最初的512字节是启动区，所以要装载下一个512字节的内容。我们来修改一下程序。改好的程序就是projects/03_day下的harib00a[①]，像以前一样，我们把它复制到tolset里来。

这次添加的内容大致如下。

本次添加的部分

```
MOV     AX,0x0820
MOV     ES,AX
MOV     CH,0            ; 柱面0
MOV     DH,0            ; 磁头0
MOV     CL,2            ; 扇区2

MOV     AH,0x02         ; AH=0x02 : 读盘
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
```

新出现的指令只有JC。真好，讲起来也轻松了。所谓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。这里突然冒出来“进位标志”这么个新词，不过大家不用担心，很快就会明白的。

至于“INT 0x13”这个指令，我们虽然知道这是要调用BIOS的0x13号函数，但还不明白它到底是干什么用的，那就来查一下吧。当然还是来看下面这个（AT）BIOS网页，

http://community.osdev.info/?(AT)BIOS

我们可以找到如下的内容：

- 磁盘读、写，扇区校验（verify），以及寻道（seek）
  - AH=0x02;（读盘）
  - AH=0x03;（写盘）

① harib是日语中haribote（纸娃娃）的前面几个字母。——译者注

- AH=0x04;（校验）
- AH=0x0c;（寻道）
- AL=处理对象的扇区数;（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:BX=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）

我们这次用的是AH=0x02，哦，原来是“读盘”的意思。

返回值那一栏里的FLACS.CF又是什么意思呢？这就是我们刚才讲到的进位标志。也就是说，调用这个函数之后，如果没有错，进位标志就是0；如果有错，进位标志就是1。这样我们就能明白刚才为什么要用JC指令了。

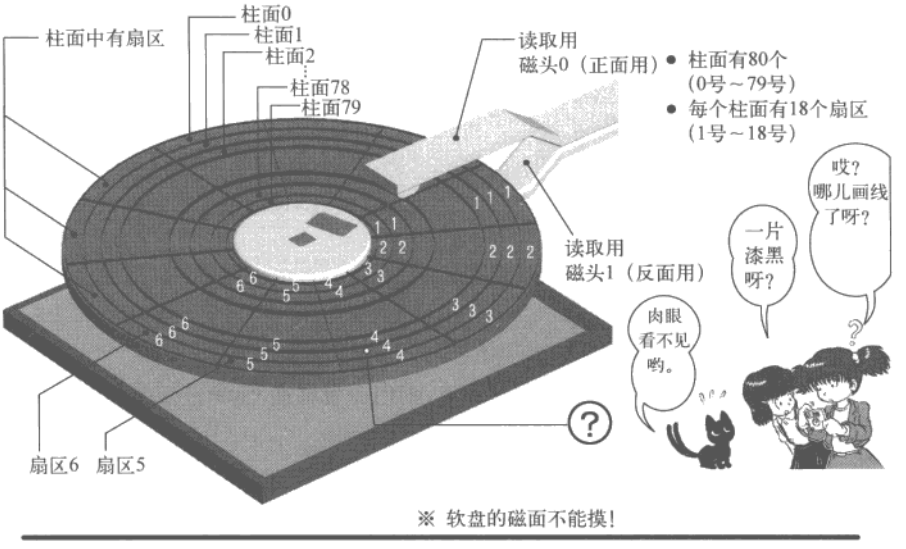
进位标志是一个只能存储1位信息的寄存器，除此之外，CPU还有其他几个只有1位的寄存器。像这种1位寄存器我们称之为标志。标志在英文中为flag，是旗帜的意思。标志之所以叫flag是因为它的开和关就像升旗降旗的状态一样。

所谓进位标志，本来是用来表示有没有进位（carry）的，但在CPU的标志中，它是最简单易用的，所以在其他地方也经常用到。这次就是用来报告BIOS函数调用是否有错的。

其他几个寄存器我们也来依次看一下吧。CH、CL、DH、DL分别是柱面号、扇区号、磁头号、驱动器号，一定不要搞错。在上面的程序中，柱面号是0，磁头号是0，扇区号是2，磁盘号是0。

在有多个软盘驱动器的时候，用磁盘驱动器号来指定从哪个驱动器的软盘上读取数据。现在的电脑，基本都只有1个软盘驱动器，而以前一般都是2个。既然现在只有一个，那不用多想，指定0号就行了。

知道了从哪个软盘驱动器读取数据之后，我们接着看从那个软盘的什么地方来读取数据。





如果手头有不用的软盘，希望大家能把它拆开看看。拆开后可以看到，中间有一个8厘米的黑色圆盘，那是一层薄薄的磁性胶片。从外向内，一圈一圈圆环状的区域，分别称为柱面0，柱面1，……，柱面79。一共有80个柱面。这并不是说工厂就是这样一圈一圈地生产软盘的，只是我们将它作为一个数据存储媒体，是这样组织它的数据存储方式的。柱面在英文中是cylinder，原意是圆筒。磁盘的柱面，尽管高度非常低，但我们可以把它看成是一个套一个的同心圆筒，它正是因此得名的。

下面讲一下磁头。磁头是个针状的磁性设备，既可以从软盘正面接触磁盘，也可以从软盘背面接触磁盘。与光盘不同，软盘磁盘是两面都能记录数据的。因此我们有正面和反面两个磁头，分别是磁头0号和磁头1号。

最后我们看一下扇区。指定了柱面和磁头后，在磁盘的这个圆环上，还能记录很多位信息，按照整个圆环为单位读写的话，实在有点多，所以我们又把这个圆环均等地分成了几份。软盘分为18份，每一份称为一个扇区。一个圆环有18个扇区，分别称为扇区1、扇区2、……扇区18。扇区在英文中是sector，意思是指领域、扇形。

综上所述，1张软盘有80个柱面，2个磁头，18个扇区，且一个扇区有512字节。所以，一张软盘的容量是：

80×2×18×512 = 1 474 560 Byte = 1 440KB

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。这次我们想要装载的就是这个扇区。

剩下的大家还不明白的就是缓冲区地址了吧。这是个内存地址，表明我们要把从软盘上读出的数据装载到内存的哪个位置。一般说来，如果能用一个寄存器来表示内存地址的话，当然会很方便，但一个BX只能表示0～0xffff的值，也就是只有0～65535，最大才64K。大家的电脑起码也都有64M内存，或者更多，只用一个寄存器来表示内存地址的话，就只能用64K的内存，这太可惜了。

于是为了解决这个问题，就增加了一个叫EBX的寄存器，这样就能处理4G内存了。这是CPU能处理的最大内存量，没有任何问题。但EBX的导入是很久以后的事情，在设计BIOS的时代，CPU甚至还没有32位寄存器，所以当时只好设计了一个起辅助作用的段寄存器（segment register）。在指定内存地址的时候，可以使用这个段寄存器。

我们使用段寄存器时，以ES:BX这种方式来表示地址，写成“MOV AL,[ES:BX]”，它代表ES×16+BX的内存地址。我们可以把它理解成先用ES寄存器指定一个大致的地址，然后再用BX来指定其中一个具体地址。

这样如果在ES里代入0xffff，在BX里也代入0xffff，就是1 114 095字节，也就是说可以指定1M以内的内存地址了。虽然这也还是远远不到64M，但当时英特尔公司的大叔们，好像觉得这就足够了。在最初设计BIOS的时代，这种配置已经很能满足当时的需要了，所以我们现在也还是要遵从这一规则。因此，大家就先忍耐一下这1MB内存的限制吧。

这次，我们指定了ES=0x0820，BX=0，所以软盘的数据将被装载到内存中0x8200到0x83ff的地方。可能有人会想，怎么也不弄个整点的数，比如0x8000什么的，那多好。但0x8000～0x81ff这512字节是留给启动区的，要将启动区的内容读到那里，所以就这样吧。

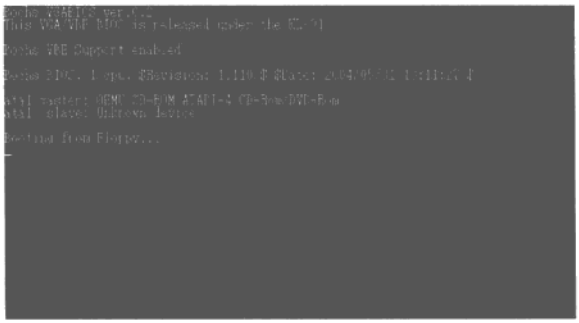
那为什么使用0x8000以后的内存呢？这倒也没什么特别的理由，只是因为从内存分布图上看，这一块领域没人使用，于是笔者就决定将我们的“纸娃娃操作系统”装载到这一区域。0x7c00～0x7dff用于启动区，0x7e00以后直到0x9fbff为止的区域都没有特别的用途，操作系统可以随便使用。

到目前为止我们开发的程序完全没有考虑段寄存器，但事实上，不管我们要指定内存的什么

地址，都必须同时指定段寄存器，这是规定。一般如果省略的话就会把“DS:”作为默认的段寄存器。

以前我们用的“MOV CX,[1234]”，其实是“MOV CX,[DS:1234]”的意思。“MOV AL,[SI]”，也就是“MOV AL,[DS:SI]”的意思。在汇编语言中，如果每回都这样写就太麻烦了，所以可以省略默认的段寄存器DS。

因为有这样的规则，所以DS必须预先指定为0，否则地址的值就要加上这个数的16倍，就会读写到其他的地方，引起混乱。



非常成功

好，我们来执行这个程序看看吧。如果程序有什么错，它就会显示错误信息。但估计不会出什么问题吧。没错的话，它就什么都不做（笑）。所以，如果屏幕上不显示任何错误信息的话，我们就成功了。

哎呀，有件事差点忘了，Makefile中可以使用简单的变量，于是笔者用变量改写了这次的Makefile。怎么样，是不是比之前稍微容易理解一些？

## 2 试错

软盘这东西很不可靠，有时会发生不能读数据的状况，这时候重新再读一次就行了。所以即使出那么一、两次错，也不要轻易放弃，应该让它再试几次。当然如果让它一直重试下去的话，要是磁盘真的坏了，程序就会陷入死循环，所以我们决定重试5次，再不行的话就真正放弃。改良后的程序就是projects/03_day下的harib00b。

本次添加的部分

```
;读磁盘

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; 柱面0
```

```
        MOV     DH,0            ; 磁头0
        MOV     CL,2            ; 扇区2

        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
        JNC     fin             ; 没出错的话跳转到fin
        ADD     SI,1            ; 往SI加1
        CMP     SI,5            ; 比较SI与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
```

还是从新出现的指令开始讲吧。JNC是另一个条件跳转指令，是“Jump if not carry”的缩写。也就是说进位标志是0的话就跳转。JAE也是条件跳转，是“Jump if above or equal”的缩写，意思是大于或等于时跳转。

现在说说出错时的处理。重新读盘之前，我们做了以下的处理，AH=0x00，DL=0x00，INT 0x13。通过前面介绍的（AT）BIOS的网页我们知道，这是“系统复位”。它的功能是复位软盘状态，再读一次。剩下的内容都很简单，只要读一读程序就能懂。

嗯，今天进展不错，继续努力吧。

## 3 读到 18 扇区

我们趁着现在这劲头，再往后多读几个扇区吧。下面来看看projects/03_day下的harib00c。

本次添加的部分

```
;读磁盘

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; 柱面0
        MOV     DH,0            ; 磁头0
        MOV     CL,2            ; 扇区2
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
```

```
        JNC     next            ; 没出错时跳转到next
        ADD     SI,1            ; 往SI加1
        CMP     SI,5            ; 比较SI与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
        ADD     AX,0x0020
        MOV     ES,AX           ; 因为没有ADD ES,0x020指令，所以这里稍微绕个弯
        ADD     CL,1            ; 往CL里加1
        CMP     CL,18           ; 比较CL与18
        JBE     readloop        ; 如果CL <= 18 跳转至readloop
```

新出现的指令是JBE。这也是个条件跳转指令，是“jump if below or equal”的缩写，意思是小于等于则跳转。

程序做的事情很简单，只要读一读程序大家马上会明白。要读下一个扇区，只需给CL加1，给ES加上0x20就行了。CL是扇区号，ES指定读入地址。0x20是十六进制下512除以16的结果，如果写成“ADD AX,512/16”或许更好懂。（笔者在写的时候，直接在头脑中换算成了0x20，当然写成512/16也一样。）可能有人会说：往BX里加上512不是更简单吗？说来也是。不过这次我们想练习一下往ES里做加法的方法，所以这段程序就留在这儿吧。

可能有人会想，这里为什么要用循环呢？这个问题很好。的确，这里不是非要用循环才行，在调用读盘函数的INT 0x13的地方，只要将AL的值设置成17就行了。这样，程序一下子就能将扇区2～18共17个扇区的数据完整地读进来。之所以将这部分做成循环是因为笔者注意到了磁盘BIOS读盘函数说明的“补充说明”部分。这个部分内容摘要如下：

- 指定处理的扇区数，范围在0x01～0xff（指定0x02以上的数值时，要特别注意能够连续处理多个扇区的条件。如果是FD的话，似乎不能跨越多个磁道，也不能超过64KB的界限。）

这些内容看起来很复杂。因为很难一两句话说清楚，这里暂不详细解释，就结果而言，这些注意事项目前跟我们还没有关系，就是写成AL=17结果也是完全一样的。但这样的方式在下一次的程序中，就会成为问题，因此为了能够循序渐进，这里特意用循环来一个扇区一个扇区地读盘。

虽然显示的画面没什么变化，但我们已经把磁盘上C0-H0-S2到C0-H0-S18的512×17=8 704字节的内容，装载到了内存的0x8200～0xa3ff处。

## 4 读入 10 个柱面

趁热打铁，我们继续学习下面的内容。C0-H0-S18扇区的下一扇区，是磁盘反面的C0-H1-S1，这次也从0xa400读入吧。按顺序读到C0-H1-S18后，接着读下一个柱面C1-H0-S1。我们保持这个势头，一直读到C9-H1-S18好了。现在我们就来看一看projects/03_day下的harib00d内容。

本次添加的部分

```
;读磁盘

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; 柱面0
        MOV     DH,0            ; 磁头0
        MOV     CL,2            ; 扇区2
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
        JNC     next            ; 没出错时跳转到next
        ADD     SI,1            ; SI加1
        CMP     SI,5            ; 比较SI与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
        ADD     AX,0x0020
        MOV     ES,AX           ; 因为没有ADD ES,0x020指令，所以这里稍微绕个弯
        ADD     CL,1            ; CL加1
        CMP     CL,18           ; 比较CL与18
        JBE     readloop        ; 如果CL <= 18，则跳转至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
```

首先还是说说新出现的指令JB。这也是条件跳转指令，是“jump if below”的缩写。翻译过来就是：“如果小于的话，就跳转。”还有一个新指令，就是在程序开头使用的EQU指令。这相当于C语言的#define命令，用来声明常数。“CYLS EQU 10”意思是“CYLS = 10”。EQU是“equal”的缩写。只将它定义成常数是因为以后我们可能修改这个数字。现在我们先随意定义成10个柱面，以后再对它进行调整（CYLS代表cylinders）。

现在启动区程序已经写得差不多了。如果算上系统加载时自动装载的启动扇区，那现在我们已经能够把软盘最初的10 × 2 × 18 × 512 = 184 320 byte=180KB内容完整无误地装载到内存里了。如果运行“make install”，把程序安装到磁盘上，然后用它来启动电脑的话，我们会发现装载过

程还是挺花时间的。这证明我们的程序运行正常。画面显示依然没什么变化，但这个程序已经用从软盘读取的数据填满了内存0x08200 ~ 0x34fff的地方。

## 5 着手开发操作系统

总算写到这个题目了，这代表我们终于完成了启动区的制作。

下面，我们先来编写一个非常短小的程序，就只让它HLT。

**最简单的操作系统？**

```
fin:
    HLT
    JMP fin
```

将以上内容保存为haribote.nas，用nask编译，输出成hanbote.sys。到这里没什么难的。

接下来，将这个文件保存到磁盘映像haribote.img里。可能有人不明白什么叫保存到映像里，其实就是像下面这样操作：

- 使用make install指令，将磁盘映像文件写入磁盘。
- 在Windows里打开那个磁盘，把haribote.sys保存到磁盘上。
- 使用工具将磁盘备份为磁盘映像。

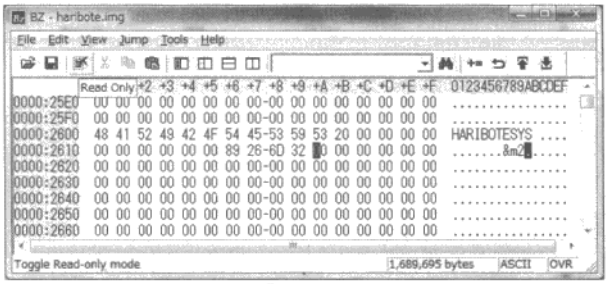
大家仔细看，以上操作以磁盘映像文件开始，最终也是以磁盘映像文件结束。我们再来想像一下，如果不用借助磁盘和Windows就可以得到磁盘映像和文件，那多方便啊。这就是“保存到磁盘映像里”的意思。

能够完成这些工作的工具其实有很多，我们曾经使用过的edimg.exe就是其中之一。所以，这次还用这个工具。

那做这个工作究竟有什么意义呢？我们先做做看，然后再说明。……笔者对程序作了修改，得到了projects/03_day下的harib00e。当然对Makefile也相应做了改动。

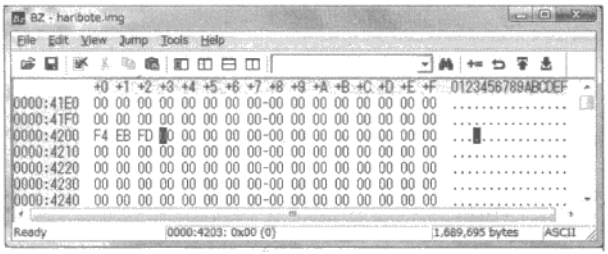
接下来用“make img”指令来做个映像文件。执行完命令，映像文件也就做成了。然后我们用二进制编辑器打开刚做成的映像文件“haribote.img”，看一看“haribote.sys”文件在磁盘中是什么样的。

最先注意到的地方是0x002600附近，磁盘的这个位置好像保存着文件名。




0x002600附近的样子

再往下看，找到0x004200那里，可以看到“F4 EB FD”。




0x004200附近的样子

这是什么呢？这就是haribote.sys的内容。因为我们用二进制编辑器看haribote.sys，它恰好也就是这三个字节。好久没用的二进制编辑器这次又大显身手了。

以上内容可以总结为：一般向一个空软盘保存文件时，

(1) 文件名会写在0x002600以后的地方；
(2) 文件的内容会写在0x004200以后的地方。

这就是我们一直想知道的东西。

了解了这一点，下面要做的事就简单了。我们将操作系统本身的内容写到名为haribote.sys文件中，再把它保存到磁盘映像里，然后我们从启动区执行这个haribote.sys就行了。接下来我们就来做这件事。

## 6 从启动区执行操作系统

那么，要怎样才能执行磁盘映像上位于0x004200号地址的程序呢？现在的程序是从启动区开始，把磁盘上的内容装载到内存0x8000号地址，所以磁盘0x4200处的内容就应该位于内存0x8000+0x4200=0xc200号地址。

这样的话，我们就往haribote.nas里加上ORG　0xc200，然后在ipl.nas处理的最后加上JMP 0xc200这个指令。这样修改后，得到的就是“projects/03_day”下的harib00f。

赶紧运行“make run”，目前什么都没发生。那么程序到底有没有执行haribote.sys呢？大家可能会有点担心。所以，下面我们让haribote.sys跳出来表现一下。

## 7 确认操作系统的执行情况

怎么让它表现呢？如果还只是输出一条信息的话就太没意思了。考虑到将来我们肯定要做成Windows那样的画面，所以这次就来切换一下画面模式。我们这次做成的文件，就是projects/03_day下的harib00g。

本次的haribote.nas

```
; haribote-os
; TAB=4

        ORG     0xc200          ; 这个程序将要被装载到内存的什么地方呢?

        MOV     AL,0x13         ; VGA显卡，320x200x8位彩色
        MOV     AH,0x00
        INT     0x10
fin:
        HLT
        JMP     fin
```

设定AH=0x00后，调用显卡BIOS的函数，这样就可以切换显示模式了。我们还可以在支持网页（AT）BIOS里看看。

设置显卡模式（video mode）

- AH=0x00;
- 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，调色板模式
  - 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）
- 返回值：无

参照以上说明，我们暂且选择0x13画面模式，因为8位彩色模式可以使用256种颜色，这一点看来不错。

如果画面模式切换正常，画面应该会变为一片漆黑。也就是说，因为可以看到画面的变化，所以能判断程序是否运行正常。由于变成了图形模式，因此光标会消失。

另外，这次还顺便修改了其他一些地方。首先将ipl.nas的文件名变成了ipl10.nas。这是为了提醒大家这个程序只能读入10个柱面。另外，想要把磁盘装载内容的结束地址告诉给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。这样启动区程序就算完成了。

成功地出现全黑画面

赶紧“make run”看看。

哦哦，画面一片漆黑。运行顺利！真是太好了！

有一点要先说明一下，现在我们把启动区里与haribote.sys没有关系的前后部分也读了进来，所以启动时很慢。可能会有人觉得这样做很浪费时间，但对于我们的纸娃娃操作系统来说，装载启动区这些部分，以后会起大作用的，所以暂时先忍耐一下吧。

## 8 32位模式前期准备

今天还有些时间，再往下讲一点吧。

现在，汇编语言的开发告一段落，我们要开始以C语言为主进行开发了，这是我们当前的目标。

笔者准备的C编译器，只能生成32位模式的机器语言。如果一定要生成16位模式机器语言，虽然也不是做不到，但是很费事，还没什么好处，所以就用32位模式吧。

所谓32位模式，指的是CPU的模式。CPU有16位和32位两种模式。如果以16位模式启动的话，用AX和CX等16位寄存器会非常方便，但反过来，像EAX和ECX等32位的寄存器，使用起来就很麻烦。另外，16位模式和32位模式中，机器语言的命令代码不一样。同样的机器语言，解释的方法也不一样，所以16位模式的机器语言在32位模式下不能运行，反之亦然。

32位模式下可以使用的内存容量远远大于1MB。另外，CPU的自我保护功能（识别出可疑的机器语言并进行屏蔽，以免破坏系统）在16位下不能用，但32位下能用。既然有这么多优点，当然要使用32位模式了。

可是，如果用32位模式就不能调用BIOS功能了。这是因为BIOS是用16位机器语言写的。如果我们有什么事情想用BIOS来做，那就全部都放在开头先做，因为一旦进入32位模式就不能调用BIOS函数了。（当然，也有从32位返回到16位的方法，但是非常费工夫，所以本书不予赘述。）

再回头说说要使用BIOS做的事情。画面模式的设定已经做完了，接下来还想从BIOS得到键盘状态。所谓键盘状态，是指NumLock是ON还是OFF等这些状态。

所以，我们这次只修改了haribote.nas。修改后的程序就是projects/03_day下的harib00h。

**本次的haribote.nas**

```
; haribote-os
; TAB=4

; 有关BOOT_INFO
CYLS    EQU     0x0ff0          ; 设定启动区
LEDS    EQU     0x0ff1
VMODE   EQU     0x0ff2          ; 关于颜色数目的信息。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          ; 这个程序将要被装载到内存的什么地方呢?
        MOV     AL,0x13         ; VGA 显卡，320x200x8位彩色
        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
```

看一下程序就能明白，设置画面模式之后，还把画面模式的信息保存在了内存里。这是因为，以后我们可能要支持各种不同的画面模式，这就需要把现在的设置信息保存起来以备后用。我们暂且将启动时的信息称为BOOT_INFO。INFO是英文information（信息）的缩写。

[VRAM]里保存的是0xa0000。在电脑的世界里，VRAM指的是显卡内存（video RAM），也就是用来显示画面的内存。这一块内存当然可以像一般的内存一样存储数据，但VRAM的功能不仅限于此，它的各个地址都对应着画面上的像素，可以利用这一机制在画面上绘制出五彩缤纷的图案。

其实VRAM分布在内存分布图上好几个不同的地方。这是因为，不同画面模式的像素数也不一样。当画面模式为○×时使用这个VRAM；而画面模式为◇△时可能使用那个VRAM，像这样，不同画面模式可以使用的内存也不一样。所以我们就预先把要使用的VRAM地址保存在BOOT_INFO里以备后用。

这次VRAM的值是0xa0000。这个值又是从哪儿得来的呢？还是来看看我们每次都参考的（AT）BIOS支持网页。在INT 0x10的说明的最后写着，这种画面模式下“VRAM是0xa0000～0xaffff的64KB”。

另外，我们还把画面的像素数、颜色数，以及从BIOS取得的键盘信息都保存了起来。保存位置是在内存0x0ff0附近。从内存分布图上看，这一块并没被使用，所以应该没问题。



## 9 开始导入C语言

终于准备就绪，现在我们直接切换到32位模式，然后运行用C语言写的程序。这就是projects/03_day下的harib00i。

程序里添加和修改了很多内容。首先是haribote.sys，它的前半部分是用汇编语言编写的，而后半部分则是用C语言编写的。所以以前的文件名haribote.nas也随之改成了asmhead.nas。并且，为了调用C语言写的程序，添加了100行左右的汇编代码。

虽然笔者也很想现在就讲这100行新添的程序，但是很抱歉，还是先跳过这部分吧。等我们再往后多学一点，再回过头来仔细讲解这段程序。其实笔者曾多次对这一部分进行说明，但每次都写得很长很复杂，恐怕大家很难理解。等到后面，大家掌握的内容多了，这一部分再理解起来也就轻松了，所以暂时先不做说明了。

下面讲C语言部分。文件名是bootpack.c。为什么要起这样的名字呢？因为以后为了启动操作系统，还要写各种其他的处理，我们想要把这些处理打成一个包（pack），所以就起了这么一个名字。最重要的核心内容非常非常短，如下所示：

本次的bootpack.c

```
void HariMain(void)
{

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

}
```

这个程序第一行的意思是：现在要写函数了，函数名字叫HariMain，而且不带参数（void），不返回任何值。“{}”括起来的部分就是函数的处理内容。

C语言中所说的函数是指一块程序，在某种程度上可以看作数学中的函数一般，即从变量x

取得值，将处理结果送给y。而上面情况下，既不从变量取得值，也不返回任何值，不太像数学中的函数，但在C语言中这也是函数。

goto指令是新出现的，相当于汇编语言中的JMP，实际上也是被编译成JMP指令。

由“/*”和“*/”括起来的部分是注释，正如这里所写的那样，C语言中不能使用HLT，也没有相当于DB的命令，所以不能用DB来放一句HLT语句。这让喜欢HTL语句的笔者感觉很是可惜。

那么，这个bootpack.c是怎样变成机器语言的呢？如果不能变成机器语言，就是说得再多也没有意义。这个步骤很长，让我们看一看。

- 首先，使用cc1.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。
- 第四部，使用obi2bim.exe从bootpack.obj生成bootpack.bim。
- 最后，使用bim2hrb.exe从bootpack.bim生成bootpack.hrb。
- 这样就做成了机器语言，再使用copy指令将asmhead.bin与bootpack.hrb单纯结合到起来，就成了haribote.sys。

来来去去搞出了这么多种类的文件，那么下面就简单介绍一下吧。

cc1是C编译器，可以将C语言程序编译成汇编语言源程序。但这个C编译器是笔者从名为gcc的编译器改造而来，而gcc又是以gas汇编语言为基础，输出的是gas用的源程序。它不能翻译成nask。

所以我们需要把gas变换成nask能翻译的语法，这就是gas2nask。解释一下这个名字。英语中的“从A到B”说成 “from A to B”，省略一下，就是“A to B”。这里把“to”写成“2”，世界上开发工具的人有时会这么写（这是英语中的谐音，2与to同音）。所以，gas2nask的意思就是“把gas文件转换成nask文件的程序”。

一旦转换成nas文件，它可就是我们的掌中之物了，只要用nask翻译一下，就能变成机器语言了。实际上也正是那样，首先用nask制作obj文件。obj文件又称目标文件，源自英文的“object”，也就是目标的意思。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。

可能会有人想，既然已经做成了机器语言，那只要把它写进映像文件里就万事大吉了。但很遗憾，这还不行，事实上这也正是使用C语言的不便之处。目标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？实际上C语言的作者已经认识到，C语言有它的局限性，不可能只用C语言来编写所有的程序，所以其中有一部分必须用汇编来写，然后链接到C语言写的程序上。

现在为止，都只有一个源程序，由它来直接生成机器语言文件，这好像是理所当然的，

完全不用考虑什么目标文件的链接。但是这个问题以后要考虑了。下面我们来讲一下用汇编语言做目标文件的方法。

所以，为了将目标文件与别的目标文件相链接，除了机器语言之外，其中还有一部分是用来交换信息的。单个的目标文件还不是独立的机器语言，其中还有一部分是没完成的。为了能做成完整的机器语言文件，必须将必要的目标文件全部链接上。完成这项工作的，就是obj2bim。bim是笔者设计的一种文件格式，意思是“binary image”，它是一个二进制映像文件。

映像文件到底是什么呢？这么说来，磁盘映像也是一种映像文件。按笔者的理解，所谓映像文件即不是文件本来的状态，而是一种代替形式。英文里面说到image file，一般是指图像文件，首先要有一个真实的东西，而它的图像则是临摹仿造出来的，虽然跟它很像，但毕竟不是真的，只是以不同的形式展示出原物的映像。不是常有人这么讲吗，“嗯，搞不懂你在说什么。能不能说得再形象一点儿？”，也就是说 “如果直接说明起来太困难的话，可以找个相似的东西来类比一下。”所谓类比，“不是本来的状态，而是一种代替的形式”。……映像文件大致也就是这个意思。

所以，实际上bim文件也“不是本来的状态，而是一种代替的形式”，也还不是完成品。这只是将各个部分全部都链接在一起，做成了一个完整的机器语言文件，而为了能实际使用，我们还需要针对每一个不同操作系统的要求进行必要的加工，比如说加上识别用的文件头，或者压缩等。这次因为要做成适合 “纸娃娃操作系统”要求的形式，所以笔者为此专门写了一个程序bim2hrb.exe，这个程序留到后面来介绍。

可能有人会想：“我在Windows和Linux上做了很多次C程序了，既没用过那么多工具，也没那么多的中间文件就搞定了，这次是怎么回事呢？”说到底，这是因为那些编译器已经很成熟了。

但是，如果我们的编译器能够直接生成可执行文件，那再想把它用于别的用途可就难了。其实在编译器内部也要做同样的事，只是在外面看不见这些过程而已。这次提供的编译器，是以能适应各种不同操作系统为前提而设计的，所以对内部没有任何隐藏，是特意像这样多生成一些中间文件的。

这样做的好处是仅靠这个编译器，就可以制作Windows、Linux以及OSASK用的可执行文件，当然，还有我们的“纸娃娃操作系统”的可执行文件。

根据以上内容，对Makefile也做了很大改动。如果大家想知道编译时指定了什么样的选项，可以看一看Makefile。

啊，忘了一件大事。函数名HariMain非常重要，程序就是从以HariMain命名的函数开始运行的，所以这个函数名不能更改。

执行这个函数，结果出现黑屏。这表示运行正常。

## 10 实现 HLT（harib00j）

虽然夜已经深了，但笔者现在还不能说“今天就到此结束”。不让计算机处于HALT（HLT），状态心里就不舒服。我们做出的程序这么耗电，不把这个问题解决掉怎么能睡得着呢（笑）。我们来努力尝试一下吧。

首先写了下面这个程序，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

        GLOBAL①  _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
```

也就是说，是用汇编语言写了一个函数。函数名叫io_hlt。虽然只叫hlt也行，但在CPU的指令之中，HLT指令也属于I/O指令，所以就起了这么一个名字。顺便说一句，MOV属于转送指令，ADD属于演算指令。

用汇编写的函数，之后还要与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“_”，否则就不能很好地与C语言函数链接。需要链接的函数名，都要用GLOBAL指令声明。

下面写一个实际的函数。写起来很简单，先写一个与用GLOBAL声明的函数名相同的标号（label），从此处开始写代码就可以了。这次新出现的RET指令，相当于C语言的return，意思就是

① 原意是“全球性的”。在计算机行业中指全局性的（变量，函数等）。其反义词是LOCAL（局部的）。

“函数的处理到此结束，返回吧”，简洁明了。

在C语言里使用这个函数的方法非常简单。我们来看看bootpack.c。

本次的bootpack.c

```
/*告诉C编译器，有一个函数在别的文件里*/

void io_hlt(void);

/*是函数声明却不用{ }，而用;，这表示的意思是：函数是在别的文件中，你自己找一下吧！ */

void HariMain(void)
{

fin:
    io_hlt(); /*执行naskfunc.nas里的_io_hlt*/
    goto fin;

}
```

源程序里的注释写得很到位，请仔细阅读一下。

好了，源程序增加了，Makefile也进行了添加，那么赶紧运行“make run”看看吧。结果虽然还是黑屏，但程序运行肯定是正常的。太好了，这就放心了。大家明天见！