30 日でできる! OS 自作入門

30 天自制操作系统

读书笔记

【日】川合秀实　著

周自恒 李黎明 曾祥江 张文旭　译

前　言

本文档为《30 天自制操作系统》（人民邮电出版社）一书的读书笔记。

文档发布在 Github 上，将随着阅读进度不定期更新。请访问 https://github.com/mengyingchina/osask-notes 获取文档最新的版本。

如果发现文档中有文字错误，请到 https://github.com/mengyingchina/osask-notes/issues 提出。

原书的版权声明一节，有：

> 本书中文简体字版由 Mynavi Corporation 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

由于部分内容摘自书中，不清楚这样摘录部分内容会不会存在版权问题，如果有版权问题，我会及时删除相关内容，请知情者告知[1]，谢谢！

[1]Email:mail@wanhu.me

目　　录

第 0 天　着手开发之前

1　前言

阅读本书几乎不需要相关储备知识，这一点稍后还会详述。不管是用什么编程语言，只要是曾经写过简单的程序，对编程有一些感觉，就已经足够了（即使没有任何编程经验，应该也能看懂），因为这本书主要就是面向初学者的。书中虽然有很多 C 语言程序，但实际上并没有用到很高深的 C 语言知识，所以就算是曾经因为 C 语言太难而中途放弃的人也不用担心看不懂。当然，如果具备相关知识的话，理解起来会相对容易一些，不过即使没有相关知识也没关系，书中的说明都很仔细，大家可以放心。

本书以 IBM PC/AT 兼容机（也就是所谓的 Windows 个人电脑）为对象进行说明。

2　何谓操作系统

3　开发操作系统的各种方法

4　无知则无畏

当我们打算开发操作系统时，总会有人从旁边跳出来，罗列出一大堆专业术语，问这问那，像内核怎么做啦，外壳怎么做啦，是不是单片啦，是不是微内核啦，等等。虽然有时候提这些问题也是有益的，但一上来就问这些，当然会让人无从回答。

要想给他们一个满意答复，让他们不再从旁指手画脚的话，还真得多学习，拿出点像模像样的见解才行。但我们是初学者，没有必要去学那些麻烦的东西，费时费力且不说，当我们知道现有操作系统在各方面都考虑得如此周密的时候，就会发现自己的想法太过简单而备受打击没了干劲。如果被前人的成果吓倒，只用这些现有的技术来做些拼拼凑凑的工作，岂不是太没意思了。

所以我们这次不去学习那些复杂的东西，直接着手开发。就算知道一大堆专业术语、专业理论，又有什么意思呢？还不如动手去做，就算做出来的东西再简单，起码也是自己的成果。而且自己先实际操作一次，通过实践找到其中的问题，再来看看是不是已经有了这些问题的解决方案，这样下来更能深刻地理解那些复杂理论。不管怎么说，反正目前我们也无法回答那些五花八门的问题，倒不如直接告诉在一旁指手画脚的人们：我们就是想用自己的方法做自己喜欢的事情，如果要讨论高深的问题，就另请高明吧。

作者苦口婆心地说了这么多就是希望如果你想开发个操作系统，就动手去写吧，到底自己重写个操作系统有什么用倒可以先放着。

如果你到现在还对要不要读这本书，或者读这本书的期望的收获有疑问，推荐你阅读豆瓣上本书的一篇评论[1]后再自行决定。

> 这本书对基础知识要求不高，懂点 C 语言和 CPU 基本知识就可以了，适合初学者。要是奔着了解操作系统原理或内核的期望，就不适宜读这本书了。30 天后也许你真的可以向作者那样做出一个基本的系统模型，但这并不意味着你对内存管理、进程管理、设备管理有着怎样高深的认识。读这本书之前先弄清自己的定位吧，毕竟时间宝贵。

5　如何开发操作系统

6　操作系统开发中的困难

7　学习本书时的注意事项（重要!）

8　各章内容摘要

[1] http://book.douban.com/review/5606888/

第 1 天　从计算机结构到汇编程序入门

1　先动手操作

随书附带了光盘[1]，给出了书中的全部示例程序，以及部分用到的工具。

> 打开附带光盘，里面有一个名为 tolset 的文件夹，把这个文件夹复制到硬盘的任意一个位置上。现在里面的东西还不多，只有 3MB 左右，不过以后我们自己开发的软件也都要放到这个文件夹里，所以往后它会越来越大，因此硬盘上最好留出 100MB 左右的剩余空间。工具安装到此结束，我们既不用修改注册表，也不用设定路径参数，就这么简单。而且以后不管什么时候，都可以把这整个文件夹移动到任何其他地方。用这些工具，我们不仅可以开发操作系统，还可以开发简单的 Windows 应用程序或 OSASK 应用程序等。

示例程序在附带光盘中名为 projects 的目录下，只要需要的示例程序目录复制到 tolset 文件夹里，就可以正常运行示例程序了。

考虑到开发中使用真实的软盘很不方便，作者特意准备了一个模拟器。

[1] 下载链接：http://pan.baidu.com/share/link?shareid=541099&uk=3657658273 或自行搜索“30 天自制操作系统.iso”

我们有了这个模拟器，不用软盘，也不用终止 Windows，就可以确认所开发的操作系统启动以后的动作，很方便呢。

使用模拟器的方法也非常简单，我们只需要在用!cons_nt.bat[2]（或者是!cons_9x.bat）打开的命令行窗口中输入“run”指令就可以了。然后一个名叫 QEMU 的非常优秀的免费 PC 模拟器就会自动运行。

§

在这一节中，作者使用二进制编辑器（十六进制编辑器）做了一个 helloos.img 文件出来。先输入了一些内容，并把内容保存成软盘映像文件格式，将这个文件写入软盘，并用它来启动电脑。画面上会显示出“hello, world”这个字符串。如果有兴趣，希望自己尝试，请参考书中这一小节的内容。

2　究竟做了些什么

简单解释了为什么上一节可以用二进制来写一个所谓的操作系统（虽然只能显示“hello, world”这个字符串）。

3　初次体验汇编程序

好，现在就让我们马上来写一个汇编程序，用它来生成一个跟刚才完全一样的 helloos.img 吧。我们这次使用的汇编语言编译器是笔者自己开发的，名为“nask”，其中的很多语法都模仿了自由软件里

[2]要根据 Windows 的版本决定用哪一个。后缀为 9x 代表是 Windows 9X 系统，后缀为 nt 的代表使用 NT 架构的 Windows 系统，如 Windows XP 及其以后的版本，以后默认为运行 cons_nt.bat，并在后面将其简写为!cons。

PS：作者开发这个系统是 2000 年左右的事情，写书的时间也比较早，所以考虑了这些问题，可以理解哈！

享有盛名的汇编器“NASM”，不过在“NASM”的基础之上又提高了自动优化能力。

†[3] projects\01_day\helloos1\

helloos.nas

```
1     DB    0xeb, 0x4e, 0x90, 0x48, 0x45, 0x4c, 0x4c, 0x4f
2     DB    0x49, 0x50, 0x4c, 0x00, 0x02, 0x01, 0x01, 0x00
3     DB    0x02, 0xe0, 0x00, 0x40, 0x0b, 0xf0, 0x09, 0x00
4     DB    0x12, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00
5     DB    0x40, 0x0b, 0x00, 0x00, 0x00, 0x00, 0x29, 0xff
6     DB    0xff, 0xff, 0xff, 0x48, 0x45, 0x4c, 0x4c, 0x4f
7     DB    0x2d, 0x4f, 0x53, 0x20, 0x20, 0x20, 0x46, 0x41
8     DB    0x54, 0x31, 0x32, 0x20, 0x20, 0x20, 0x00, 0x00
9     RESB    16
10    DB    0xb8, 0x00, 0x00, 0x8e, 0xd0, 0xbc, 0x00, 0x7c
11    DB    0x8e, 0xd8, 0x8e, 0xc0, 0xbe, 0x74, 0x7c, 0x8a
12    DB    0x04, 0x83, 0xc6, 0x01, 0x3c, 0x00, 0x74, 0x09
13    DB    0xb4, 0x0e, 0xbb, 0x0f, 0x00, 0xcd, 0x10, 0xeb
14    DB    0xee, 0xf4, 0xeb, 0xfd, 0x0a, 0x0a, 0x68, 0x65
15    DB    0x6c, 0x6c, 0x6f, 0x2c, 0x20, 0x77, 0x6f, 0x72
16    DB    0x6c, 0x64, 0x0a, 0x00, 0x00, 0x00, 0x00, 0x00
17    RESB    368
```

[3]源代码在随书光盘（见第1节的说明）中的路径; 另外，为了便于查看，给代码中添加了行号，导致复制文中代码不是很方便，请直接使用光盘中的源代码或手动输入。

```
18      DB      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x55, 0xaa
19      DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
20      RESB    4600
21      DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
22      RESB    1469432
```

把 helloos1 文件夹复制粘帖到 tolset 文件夹里，我们只要在用“!cons”打开的命令行窗口里输入“asm”，就可以生成 helloos.img 文件。在用“asm”作成 img 文件后，再执行“run”指令，就可以得到与刚才一样的结果。

§

DB 指令是“data byte”的缩写，也就是往文件里直接写入 1 个字节的指令。

RESB 指令是“reserve byte”的略写，如果想要从现在的地址开始空出 10 个字节来，就可以写成 RESB 10，意思是我们预约了这 10 个字节（大家可以想象成在对号入座的火车里，预订了 10 个连号座位的情形）。而且 nask 不仅仅是把指定的地址空出来，它还会在空出来的地址上自动填入 0x00，所以我们这次用这个指令就可以输出很多的 0x00，省得我们自己去写 18 万行程序了，真是帮了个大忙。

这里还要说一下，数字的前面加上 0x，就成了十六进制数，不加 0x，就是十进制数。这一点跟 C 语言是一样的。

4　加工润色

刚才我们把程序变成了短短的 22 行，这成果令人欣喜。不过还有一点不足就是很难看出这些程序是干什么的，所以我们下面就来稍微改写一下，让别人也能看懂。

†projects\01_day\helloos2

helloos.nas

```
; 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           ; 不使用分区，必须是 0
        DD      2880        ; 重写一次磁盘大小
        DB      0,0,0x29        ; 意义不明，固定
```

```
21          DD      0xffffffff      ;（可能是）卷标号码
22          DB      "HELLO-OS   "   ; 磁盘的名称（11 字节）
23          DB      "FAT12   "      ; 磁盘格式名称（8 字节）
24          RESB    18              ; 先空出 18 字节
25
26  ; 程序主体
27          DB      0xb8, 0x00, 0x00, 0x8e, 0xd0, 0xbc, 0x00, 0x7c
28          DB      0x8e, 0xd8, 0x8e, 0xc0, 0xbe, 0x74, 0x7c, 0x8a
29          DB      0x04, 0x83, 0xc6, 0x01, 0x3c, 0x00, 0x74, 0x09
30          DB      0xb4, 0x0e, 0xbb, 0x0f, 0x00, 0xcd, 0x10, 0xeb
31          DB      0xee, 0xf4, 0xeb, 0xfd
32
33  ; 信息显示部分
34
35          DB      0x0a, 0x0a      ; 2 个换行
36          DB      "hello, world"
37          DB      0x0a            ; 换行
38          DB      0
39
40          RESB    0x1fe-$         ; 填写 0x00, 直到 0x001fe
41          DB      0x55, 0xaa
42
```

```
43 ; 以下是启动区以外部分的输出
44
45         DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
46         RESB    4600
47         DB      0xf0, 0xff, 0xff, 0x00, 0x00, 0x00, 0x00, 0x00
48         RESB    1469432
```

首先是“;”命令，这是个注释命令。

其次是 DB 指令的新用法。我们居然可以直接用它写字符串。在写字符串的时候，汇编语言会自动地查找字符串中每一个字符所对应的编码，然后把它们一个字节一个字节地排列起来。这个功能非常方便，也就是说，当我们想要变更输出信息的时候，就再也不用自己去查字符编码表了。

再有就是 DW 指令和 DD 指令，它们分别是“data word”和“data double-word”的缩写，是 DB 指令的“堂兄弟”。word 的本意是“单词”，但在计算机汇编语言的世界里，word 指的是“16 位”的意思，也就是 2 个字节。“double-word”是“32 位”的意思，也就是 4 个字节。对了，差点忘记说 RESB 0x1fe-$ 了。这个美元符号的意思如果不讲，恐怕谁也搞不明白，它是一个变量，可以告诉我们这一行现在的字节数（如果严格来说，有时候它还会有别的意思，关于这一点我们明天再讲）。在这个程序里，我们已经在前面输出了 132 字节，所以这里的 $ 就是 132。因此 nask 先用 0x1fe 减去 132，得出 378 这一结果，然后连续输出 378 个字节的 0x00。

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, world”变成“this is a pen.”的话，中间要输出 0x00 的字节数也会随之变化。换句话说，我们必须保证软盘的第 510 字节（即第 0x1fe 字节）开始的地方是 55 AA。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地改写输出信息了。

第 2 天　汇编语言学习与 Makefile 入门

1　介绍文本编辑器

如中文译者所注，推荐使用 Notepad++ 文本编辑器。

使用这个软件打开光盘中提供的源代码会出现日语注释乱码，选择 格式 –> 编码字符集 –> 日文 –>Shift-JIS 即可正常显示代码中原书作者的日语注释。但是，关闭文件之后重新打开又会恢复原状，解决方法为在选择 Shift-JIS 编码后复制内容到一个新的文件中去，保存替换原先的文件即可，需要提示的是，新建的文件要使用 ANSI 编码格式保存，不能是 UTF-8，否则会导致后面编译时报错。

原书作者推荐的文本编辑器 TeraPad 在中文系统中会出现软件本身的文本乱码，如菜单栏工具栏的文字，但是无需设置就可以正常显示代码中的日语注释。

2　继续开发

选讲程序的核心部分，核心程序之前和启动区以外的内容需要具备软盘方面的相关知识，后面讲。

†projects\02_day\helloos3

helloos.nas

```
; hello-os
; TAB=4

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

; 以下的记述用于标准 FAT12 格式软盘

        JMP         entry
        DB         0x90
---中略---
; 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:
```

```
22          MOV         AL,[SI]
23          ADD         SI,1              ; 给 SI 加 1
24          CMP         AL,0
25          JE          fin
26          MOV         AH,0x0e             ; 显示一个文字
27          MOV         BX,15              ; 指定字符颜色
28          INT         0x10              ; 调用显卡 BIOS
29          JMP         putloop
30  fin:
31          HLT                           ; 让 CPU 停止，等待指令
32          JMP         fin                 ; 无限循环
33
34  msg:
35          DB          0x0a, 0x0a         ; 换行两次
36          DB          "hello, world"
37          DB          0x0a              ; 换行
38          DB          0
```

§

ORG 指令：程序从指定的这个地址开始，也就是把程序装载到内存中的指定地址。这里是 0x7c00。

JMP 指令：无条件跳转。配合下面的标签"entry" 等，可指定跳转的目的地。

“entry”：标签声明，用于指定 JMP 指令的跳转地址。

MOV 指令：赋值。“MOV AX,0”相当于“AX=0”这样一个赋值语句。同样，“MOV SS,AX”相当于“SS=AX”。

§

相关寄存器：

16 位寄存器：AX、CX、DX、BX、SP、BP、SI、DI；

其中，前四个的高低 8 位可当 8 位寄存器用，AL、CL、DL、BL、AH、CH、DH、BH；

32 位寄存器：16 位寄存器可以扩展为 32 位寄存器，EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI；

段寄存器：ES、CS、SS、DS、FS、GS；

§

“MOV SI,msg”：这里是可以“将标号赋值给寄存器”。在汇编语言中，所有的标号都仅仅是单纯的数字。每个标号对应的数字，是由汇编器根据 ORG 指令计算出来的。编译器计算出的“标号的地方对应的内存地址”就是那个标号的值。这里，msg 的地址是 0x7c74，所以这个指令就是把 0x7c74 带入到 SI 寄存器去。

“MOV AL,[SI]”：MOV 指令的数据传送源和传送目的地不仅可以是寄存器或常数，也可以是内存地址。这个时候，我们使用方括号 ([]) 来表示内存地址。另外，可以用来指定内存地址的寄存器只有 BX、BP、SI、DI 这几个。

可以用下面指令将 SI 地址的 1 字节内容读入到 AL：

```
MOV AL, BYTE [SI]
```

由于 MOV 指令的规则，即源数据和目的数据必须位数相同，也就是向 AL 里代入的只能是 BYTE，这样以来就可以省略 BYTE，即可以写成：

```
MOV AL, [SI]
```

§

ADD 指令是加法指令。若以 C 语言的形式改写“ADD SI,1”的话，就是“SI=SI=1”。

CMP 是比较指令。“CMP AL,0”，是将 AL 中的值与 0 进行比较。

JE 是条件跳转指令之一。所谓的条件跳转指令，就是根据比较的结果决定跳转或者不跳转。就 JE 指令而言，如果比较结果相等，则跳转到指定的地址；而如果比较结果不等，则不跳转，继续执行下一条指令。

```
CMP AL, 0
JE fin
```

这两条指令相当于：

```
if(AL == 0){ goto fin;}
```

§

INT 是软件中断指令。在这里是调用显卡 BIOS，不解释。

§

HLT 指令，让 CPU 停止动作，进入待机状态。

§

用 C 语言改写 helloos.nas 程序节选。

```
1 entry:
2     AX = 0;
3     SS = AX;
```

```
4     SP = 0x7c00;
5     DS = AX;
6     ES = AX;
7     SI = msg;
8  putloop:
9     AL = BYTE [SI];
10    SI = SI+1;
11    if (AL ==0 ){goto fin;}
12    AH = 0x0e;
13    BX = 15;
14    INT 0x10;
15    goto putloop;
16 fin:
17    HLT;
18    goto fin;
```

就是有了这个程序，我们才能把 msg 里写的数据，一个字符一个字符地显示出来，并且数据变成 0 以后，HLT 指令就会让程序进入无限循环，“hello,world”就是这样显示出来的。

§

程序中的 ORG 后地址 0x7c00 是因为目前约定内存的0x00007c00-0x00007dff 地址为启动区内容的装载地址，不能随便改成其它地址。

3　先制作启动区

考虑到以后的开发，不要一下子用 nask 来制作整个磁盘映像，而是先用它来制作 512 字节的启动区，剩下的部分我们用磁盘映像管理工具来做。

先把 helloos.nas 的后半部分截掉，这是因为启动区只需要最后的 512 字节。现在这个程序仅仅用来制作启动区，所以把文件名改为 ipl.nas。

然后改造 asm.bat，将输出的文件名改成 ipl.bin。另外，也顺便输出列表文件 ipl.lst。这是一个文本文件，可以用来简单地确认每个指令是怎么翻译成机器语言的。

另外还增加了一个 makeimg.bat 文件。它是以 ipl.bin 为基础，制作磁盘映像文件 helloos.img 的批处理文件。利用作者开发的磁盘映像管理工具 edimg.exe，先读入一个空白的磁盘映像文件，然后在开头写入 ipl.bin，最后输出名为 helloos.img 的磁盘映像文件。

这样，从编译到测试的步骤为双击!cons，然后在命令行窗口中按顺序输入 asm –>makeimg –>run 这 3 个命令。

下一节有更简单的编译方式。

4　Makefile 入门

作者编写了一个 Makefile 文件，这样就可以方便的生成所需要的文件。

†projects\02_day\helloos5

Makefile

```
1
2 # デフォルト動作
```

```
3
4 default :
5     ../z_tools/make.exe img
6
7 # ファイル生成規則
8
9 ipl.bin : ipl.nas Makefile
10    ../z_tools/nask.exe ipl.nas ipl.bin ipl.lst
11
12 helloos.img : ipl.bin Makefile
13    ../z_tools/edimg.exe   imgin:../z_tools/fdimg0at.tek \
14        wbinimg src:ipl.bin len:512 from:0 to:0   imgout:helloos.img
15
16 # コマンド
17
18 asm :
19    ../z_tools/make.exe -r ipl.bin
20
21 img :
22    ../z_tools/make.exe -r helloos.img
23
24 run :
```

```
25      ../z_tools/make.exe img
26      copy helloos.img ..\z_tools\qemu\fdimage0.bin
27      ../z_tools/make.exe -C ../z_tools/qemu
28
29  install :
30      ../z_tools/make.exe img
31      ../z_tools/imgtol.com w a: helloos.img
32
33  clean :
34      -del ipl.bin
35      -del ipl.lst
36
37  src_only :
38      ../z_tools/make.exe clean
39      -del helloos.img
```

使用方法为：用!cons 打开命令行窗口，然后就可以通过输入 `make img` 来生成映像文件，输入 `make run` 来运行，等等。可以使用的参数在 Makefile 文件中写明了。另外，当执行不带参数的 make 命令时，相当于执行 `make img`。

第 3 天　进入 32 位模式并导入 C 语言

作者给开发的操作系统起名字叫 纸娃娃操作系统——haribote os。

1　制作真正的 IPL

制作一个可以称为真正的 IPL（启动程序装载器），让启动区真正的开始装载程序。

§

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。程序是在上一天的基础上修改的，添加了以下内容：

†projects\03_day\harib00a

ipl.nas 本次添加的部分

```
1        MOV     AX,0x0820
2        MOV     ES,AX
3        MOV     CH,0            ; シリンダ 0
```

```
4       MOV     DH,0            ; ヘッド 0
5       MOV     CL,2            ; セクタ 2
6
7       MOV     AH,0x02           ; AH=0x02 : ディスク読み込み
8       MOV     AL,1            ; 1 セクタ
9       MOV     BX,0
10      MOV     DL,0x00           ; A ドライブ
11      INT     0x13            ; ディスク BIOS 呼び出し
12      JC      error
```

INT 0x13 是调用 BIOS 的 0x13 号函数。
下面是 BIOS 13 中断的简单说明（功能有磁盘的读、写、扇区校验、寻道）

- AH=0x02 读盘/0x03 写盘/0x04 校验/0x0c 寻道
- AL= 处理连续扇区数
- 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

返回值：FLAGS.CF=0，没有错误 AH=0；FLAGS=1 有错误，AH 保存错误码

这里，JC（jump if carry）是条件跳转指令，如果进位标志为 1，就跳转。跳转条件看调用函数的返回值 FLAG.CF。

对照程序和 BIOS 函数参数说明，可以知道我们这次使用的是读盘，柱面号是 0，磁头号是 0，扇区号是 2，磁盘号是 0。

§

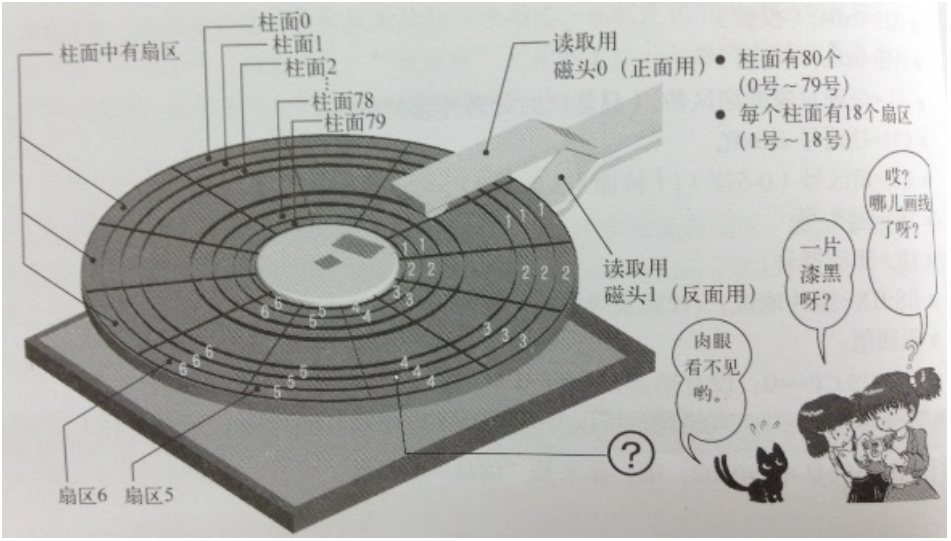

图 3.1: 软盘的结构

一张软盘有 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 $80 \times 2 \times 18 \times 512 = 1474560Byte = 1440KB$。

含有 IPL 的启动区，位于 C0-H0-S1（柱面 0，磁头 0，扇区 1），下一个扇区是 C0-H0-S2，这次我们要装载的扇区就是这个。

ES:BX= 缓冲地址 是个内存地址，表明要把软盘上读出的数据装载到内存的哪个位置。由于一个 BX 只能表示 0∼0xffff 的值，即 64K，太小了，使用段寄存器以 ES:BX 这种方式来表示地址，写成“MOV AL,[ES:BX]”，代表“ES×16+BX”的内存地址。程序中指定了 ES=0x0820，BX=0，所以软盘的数据会被装载到 0x8200∼0x83ff 的位置。

§

作者使用变量的方式改写了 Makefile 文件，可以看一下。

2　试错

鉴于软盘的不可靠性，有时候需要在软盘读不出来的时候多读几次，这里重读 5 次。

†projects\03_day\harib00b

ipl.nas 本次添加的部分

```
; ディスクを読む

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; シリンダ 0
        MOV     DH,0            ; ヘッド 0
        MOV     CL,2            ; セクタ 2
```

```
8
9           MOV         SI,0            ; 失敗回数を数えるレジスタ
10 retry:
11          MOV         AH,0x02           ; AH=0x02 : ディスク読み込み
12          MOV         AL,1            ; 1 セクタ
13          MOV         BX,0
14          MOV         DL,0x00           ; A ドライブ
15          INT         0x13            ; ディスク BIOS 呼び出し
16          JNC         fin               ; エラーがおきなければ fin へ
17          ADD         SI,1            ; SI に 1 を足す
18          CMP         SI,5            ; SI と 5 を比較
19          JAE         error            ; SI >= 5 だったら error へ
20          MOV         AH,0x00
21          MOV         DL,0x00           ; A ドライブ
22          INT         0x13            ; ドライブのリセット
23          JMP         retry
```

其中，JNC（jump if not carry），进位标志为 0 的话跳转；JAE（jump if above or equal），大于或等于时跳转。

在重新读盘之前，做了如下处理，AH=0x00，DL=0x00，INT=0x13，即完成系统复位。

3　读到 18 扇区

往后多读几个扇区，读完柱面 0 的 18 个扇区。

†projects\03_day\harib00c

ipl.nas 本次添加的部分

```
; ディスクを読む

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; シリンダ 0
        MOV     DH,0            ; ヘッド 0
        MOV     CL,2            ; セクタ 2
readloop:
        MOV     SI,0            ; 失敗回数を数えるレジスタ
retry:
        MOV     AH,0x02           ; AH=0x02 : ディスク読み込み
        MOV     AL,1            ; 1 セクタ
        MOV     BX,0
        MOV     DL,0x00           ; A ドライブ
        INT     0x13            ; ディスク BIOS 呼び出し
        JNC     next            ; エラーがおきなければ next へ
        ADD     SI,1            ; SI に 1 を足す
```

```
18          CMP         SI,5            ; SI と 5 を比較
19          JAE         error           ; SI >= 5 だったら error へ
20          MOV         AH,0x00
21          MOV         DL,0x00           ; A ドライブ
22          INT         0x13            ; ドライブのリセット
23          JMP         retry
24  next:
25          MOV         AX,ES           ; アドレスを 0x200 進める
26          ADD         AX,0x0020
27          MOV         ES,AX           ; ADD ES,0x020 という命令がないのでこうしている
28          ADD         CL,1            ; CL に 1 を足す
29          CMP         CL,18           ; CL と 18 を比較
30          JBE         readloop        ; CL <= 18 だったら readloop へ
```

JBE(jump if below or equal)，小于等于则跳转。

要读入下一个扇区只需给 CL 加 1，给 ES 加上 0x20（512/16）。CL 是扇区号，ES 指定读入地址。

这里使用循环的方式读入各个扇区，而不是在开始的时候设置读入的扇区数 AL=17，因为磁盘 BIOS 读盘函数有一些“补充说明”：

> 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。）

经过上面这些读盘处理，已经把磁盘上 C0-H0-S2 到 C0-H0-S18 的 $512 \times 17 = 8704$ 个字节的内容，装载到了内存的 0x8200∼0xa3ff。

4　读入 10 个柱面

C0-H0-S18 扇区的下一个扇区是磁盘反面的 C0-H1-S1，按顺序读到 C0-H1-S18，接着读 C1-H0-S1，最后一直读到 C9-H1-S18。

†projects\03_day\harib00d

```
; ディスクを読む

        MOV     AX,0x0820
        MOV     ES,AX
        MOV     CH,0            ; シリンダ 0
        MOV     DH,0            ; ヘッド 0
        MOV     CL,2            ; セクタ 2
readloop:
        MOV     SI,0            ; 失敗回数を数えるレジスタ
retry:
        MOV     AH,0x02           ; AH=0x02 : ディスク読み込み
        MOV     AL,1            ; 1 セクタ
        MOV     BX,0
        MOV     DL,0x00           ; A ドライブ
        INT     0x13            ; ディスク BIOS 呼び出し
        JNC     next            ; エラーがおきなければ next へ
```

```
17          ADD         SI,1                ; SI に 1 を足す
18          CMP         SI,5                ; SI と 5 を比較
19          JAE         error                ; SI >= 5 だったら error へ
20          MOV         AH,0x00
21          MOV         DL,0x00               ; A ドライブ
22          INT         0x13                ; ドライブのリセット
23          JMP         retry
24  next:
25          MOV         AX,ES                ; アドレスを 0x200 進める
26          ADD         AX,0x0020
27          MOV         ES,AX                ; ADD ES,0x020 という命令がないのでこうしている
28          ADD         CL,1                ; CL に 1 を足す
29          CMP         CL,18                ; CL と 18 を比較
30          JBE         readloop            ; CL <= 18 だったら readloop へ
31          MOV         CL,1
32          ADD         DH,1
33          CMP         DH,2
34          JB          readloop            ; DH < 2 だったら readloop へ
35          MOV         DH,0
36          ADD         CH,1
37          CMP         CH,CYLS
38          JB          readloop            ; CH < CYLS だったら readloop へ
```

JB(jump if below), 如果小于就跳转。

在程序开头使用了 EQU 指令来声明常数，即`CYLS EQU 10`。现在已经能够把软盘最初的 $10 \times 2 \times 18 \times 512 = 184320byte = 180KB$ 的内容完整的装载到内存中了。

5　着手开发操作系统

编写一个短小的程序，只让它 HLT。

`projects\03_day\harib00e`

haribote.nas

```
fin:
    HLT
    JMP fin
```

使用 nask 编译，输出成 harbote.sys。用“make img”指令来生成映像文件。

使用作者最开始提到的二进制编辑器查看 haribote.img 和 haribote.sys 内容，可以发现 0x002600 附近保存着文件名，0x004200 位置保存着文件的内容，这里分别是 haribotesys 和编译后的 harbote.sys 里面的内容“`F4 EB FD`”。

这样，要做的工作就是将操作系统的本身的内容写到名为 harbote.sys 的问卷中，再把它保存到磁盘映像里，然后从启动区执行这个 harbote.sys 就行了。

6　从启动区执行操作系统

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，所以磁盘映像上位于 0x004200 号地址的程序位于内存的 0x8000+0x4200=0xc200 号地址。

修改 haribote.nas，加上 ORG 0xc200，然后在 ipl.nas 处理的最后加上 JMP 0xc200 这个指令。详细程序见“projects/03_day/harib00f”。

通过运行“make run”来运行程序。

7　确认操作系统的执行情况

通过切换以下画面模式，让画面变成一片漆黑，证明程序正常运行。

†projects/03_day/harib00g

```
1 ; haribote-os
2 ; TAB=4
3
4         ORG         0xc200              ; このプログラムがどこに読み込まれるのか
5
6         MOV         AL,0x13              ; VGA グラフィックス、320x200x8bit カラー
7         MOV         AH,0x00
8         INT         0x10
9 fin:
```

```
10          HLT
11          JMP         fin
```

设置显卡模式：

- AH=0x00
- AL= 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
  - 扩展 VGA 图形模式，800×600×4 位彩色模式，独特的 4 面存储模式
- 返回值：无

程序的变动：将 ipl.nas 改名为 ipl10.nas，提醒这个程序只能读入 10 个柱面。

想把磁盘装载内容的结束地址告诉给 haribote.sys，在 ipl10.nas 文件中“JMP 0xc200”之前加入了一行代码，将 CYLS 的值写到内存地址 0x0ff0 中。

运行“make run”查看效果，应该是一片漆黑画面。

8　32 位模式前期准备[1]

考虑到系统以后会支持各种不同的画面模式，就需要把现在的设置信息（BOOT_INFO）保存起来以备后用。

†projects\03_day\harib00h

haribote.nas

```
; haribote-os
; TAB=4

; BOOT_INFO 関係
CYLS    EQU     0x0ff0          ; ブートセクタが設定する
LEDS    EQU     0x0ff1
VMODE    EQU     0x0ff2          ; 色数に関する情報。何ビットカラーか?
SCRNX    EQU     0x0ff4          ; 解像度の X
SCRNY    EQU     0x0ff6          ; 解像度の Y
VRAM    EQU     0x0ff8          ; グラフィックバッファの開始番地

        ORG     0xc200          ; このプログラムがどこに読み込まれるのか

        MOV     AL,0x13          ; VGA グラフィックス、320x200x8bit カラー
        MOV     AH,0x00
```

[1]书中作者先讲述了为什么用 32 位模式，自己看下书吧。

```
16          INT         0x10
17          MOV         BYTE [VMODE],8      ; 画面モードをメモする
18          MOV         WORD [SCRNX],320
19          MOV         WORD [SCRNY],200
20          MOV         DWORD [VRAM],0x000a0000
21
22  ; キーボードの LED 状態を BIOS に教えてもらう
23
24          MOV         AH,0x02
25          INT         0x16                ; keyboard BIOS
26          MOV         [LEDS],AL
27
28  fin:
29          HLT
30          JMP         fin
```

[VRAM] 里保存的是 0xa0000。VRAM 是显卡内存，它的各个地址对应画面上的像素。不同的画面模式对应不同的 VRAM，因此这里将使用的 VRAM 地址保存在 BOOT_INFO 里。这种画面模式下 VRAM 是“0xa000~0xaffff 的 64KB”。画面的像素数、颜色数以及从 BIOS 取得的键盘信息都保存在内存 0x0ff0 位置附近。

9　开始导入 C 语言

现在，直接切换到 32 位模式，然后运行 C 语言写程序。

程序做了很大的改动，haribote.sys 的前半部分使用汇编语言编写的，后半部分是用 C 语言编写的，所以将 haribote.nas 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。这些汇编代码作者在后面再讲解，这里直接跳过，分析 C 语言部分。

C 语言部分写在 bootpack.c 文件中。

†projects\03_day\harib00i

bootpack.c

```
1 void HariMain(void)
2 {
3
4 fin:
5     /* ここに HLT を入れたいのだが、C 言語では HLT が使えない！ */
6     goto fin;
7
8 }
```

§

bootpack.c 变成机器语言的过程：

1. 使用 ccl.exe 从 bootpack.c 生成 bootpack.gas；

2. 使用 gas2nask.exe 从 bootpack.gas 生成 bootpack.nas；
3. 使用 nask.exe 从 bootpack.nas 生成 bootpack.obj；
4. 使用 obj2bim.exe 从 bootpack.obj 生成 bootpack.bim；
5. 使用 bim2hrb.exe 从 bootpack.bim 生成 bootpack.hrb；
6. 使用 copy 指令将 asmhead.bin 与 bootpack.hrb 单纯结合起来就生成了 haribote.sys。

10　实现 HLT（harib00j）

†projects\03_day\harib00j

naskfun.nas

```
1  ; naskfunc
2  ; TAB=4
3
4  [FORMAT "WCOFF"]                ; オブジェクトファイルを作るモード
5  [BITS 32]                        ; 32 ビットモード用の機械語を作らせる
6
7
8  ; オブジェクトファイルのための情報
9
10 [FILE "naskfunc.nas"]            ; ソースファイル名情報
```

```
11
12          GLOBAL    _io_hlt             ; このプログラムに含まれる関数名
13
14
15 ; 以下は実際の関数
16
17 [SECTION .text]          ; オブジェクトファイルではこれを書いてからプログラムを書く
18
19 _io_hlt:     ; void io_hlt(void);
20          HLT
21          RET
```

使用汇编语言编写了一个函数，io_hlt。将输出设置为 WCOFF 模式，可以编译成目标文件，与 bootpack.obj 链接。

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。需要先在函数名前面加上“_”，否则不能很好地与 C 语言函数链接。需要链接的函数名都需要 GLOBAL 指令声明。

下面写一个实际的函数。先写一个与用 GLOBAL 声明的函数名相同的标号，从此处开始写代码就可以了。

†projects\03_day\harib00j

bootpack.c

```
1 /* 他のファイルで作った関数がありますと C コンパイラに教える */
2
3 void io_hlt(void);
4
```

```
5 /* 関数宣言なのに、{} がなくていきなり; を書くと、
6     他のファイルにあるからよろしくね、という意味になるのです。 */
7
8 void HariMain(void)
9 {
10
11 fin:
12     io_hlt(); /* これで naskfunc.nas の _io_hlt が実行されます */
13     goto fin;
14
15 }
```

“make run”运行程序。

第 4 天　C 语言与画面显示的练习

1　用 C 语言实现内存写入（harib01a）

在让画面黑屏的基础上，通过写 VRAM 的值，在画面上画出些“花”来。

†projects\04_day\harib01a

naskfunc.nas

```
1 _write_mem8:    ; void write_mem8(int addr, int data);
2         MOV         ECX,[ESP+4]         ; [ESP+4] に addr が入っているのでそれを ECX に読み込む
3         MOV         AL,[ESP+8]          ; [ESP+8] に data が入っているのでそれを AL に読み込む
4         MOV         [ECX],AL
5         RET
```

C 语言部分：

bootpack.c

```
1 void io_hlt(void);
2 void write_mem8(int addr, int data);
```

```
3
4 void HariMain(void)
5 {
6     int i; /* 变数宣言。i という変数は、32 ビットの整数型 */
7
8     for (i = 0xa0000; i <= 0xaffff; i++) {
9         write_mem8(i, 15); /* MOV BYTE [i],15 */
10    }
11
12    for (;;) {
13        io_hlt();
14    }
15 }
```

VRAM 中都写入了 15，意思是全部像素的颜色都是第 15 种颜色，即白色，因此运行程序后画面会变成白色。

2　条纹图案（harib01b）

†projects\04_day\harib01b

bootpack.c

```
1
2     for (i = 0xa0000; i <= 0xaffff; i++) {
```

```
3         write_mem8(i, i&0x0f);
4     }
```

通过与运算，将 15 改成特殊的值，低 4 位保持不变，高 4 位全部变成 0。这样每隔 16 个像素，色号就反复一次。

3　挑战指针（harib01c）

使用 C 语言的指针，修改上面程序，实现内存写入。

†projects\04_day\harib01c

bootpack.c

```
1 void io_hlt(void);
2
3 void HariMain(void)
4 {
5     int i; /* 变数宣言。i という変数は、32 ビットの整数型 */
6     char *p; /* p という変数は、BYTE [...] 用の番地 */
7
8     for (i = 0xa0000; i <= 0xaffff; i++) {
9
10        p = i; /* 番地を代入 */
11        *p = i & 0x0f;
```

```
12
13          /* これで write_mem8(i, i & 0x0f); の代わりになる */
14      }
15
16      for (;;) {
17          io_hlt();
18      }
19  }
```

4　指针的应用（1）（harib01d）

本节和下面一个小节做的事情和上面一样，只是换了种 C 语言的写法。

†projects\04_day\harib01d

bootpack.c

```
1       p = (char *) 0xa0000; /* 番地を代入 */
2
3       for (i = 0; i <= 0xffff; i++) {
4           *(p + i) = i & 0x0f;
5       }
6
```

5　指针的应用（2）（harib01e）

†projects\04_day\harib01e

bootpack.c

```
1    p = (char *) 0xa0000; /* 番地を代入 */
2
3    for (i = 0; i <= 0xffff; i++) {
4        p[i] = i & 0x0f;
5    }
```

6　色号设定（harib01f）

在描绘一个操作系统模样的画面之前，需要先处理颜色问题。这次使用的是 320×200 的 8 位颜色模式，色号使用 8 位（二进制）数，也就是只能使用 0～255 的数。8 位的彩色模式由程序员指定 0～255 的数字对应的颜色，比如 25 号颜色对应 #ffffff，26 号颜色对应 #123456 等。这种方式叫做调色板（palette）。

这里指定 0～15 这 16 种颜色。

†projects\04_day\harib01f

bootpack.c

```
1 void io_hlt(void);
2 void io_cli(void);
3 void io_out8(int port, int data);
4 int io_load_eflags(void);
```

```
5  void io_store_eflags(int eflags);
6
7  /* 実は同じソースファイルに書いてあっても、定義する前に使うのなら、
8      やっぱり宣言しておかないといけない。 */
9
10 void init_palette(void);
11 void set_palette(int start, int end, unsigned char *rgb);
12
13 void HariMain(void)
14 {
15     int i; /* 変数宣言。i という変数は、32 ビットの整数型 */
16     char *p; /* p という変数は、BYTE [...] 用の番地 */
17
18     init_palette(); /* パレットを設定 */
19
20     p = (char *) 0xa0000; /* 番地を代入 */
21
22     for (i = 0; i <= 0xffff; i++) {
23         p[i] = i & 0x0f;
24     }
25
26     for (;;) {
```

```
27          io_hlt();
28      }
29  }
30
31  void init_palette(void)
32  {
33      static unsigned char table_rgb[16 * 3] = {
34          0x00, 0x00, 0x00,    /*  0: 黒 */
35          0xff, 0x00, 0x00,    /*  1: 明るい赤 */
36          0x00, 0xff, 0x00,    /*  2: 明るい緑 */
37          0xff, 0xff, 0x00,    /*  3: 明るい黄色 */
38          0x00, 0x00, 0xff,    /*  4: 明るい青 */
39          0xff, 0x00, 0xff,    /*  5: 明るい紫 */
40          0x00, 0xff, 0xff,    /*  6: 明るい水色 */
41          0xff, 0xff, 0xff,    /*  7: 白 */
42          0xc6, 0xc6, 0xc6,    /*  8: 明るい灰色 */
43          0x84, 0x00, 0x00,    /*  9: 暗い赤 */
44          0x00, 0x84, 0x00,    /* 10: 暗い緑 */
45          0x84, 0x84, 0x00,    /* 11: 暗い黄色 */
46          0x00, 0x00, 0x84,    /* 12: 暗い青 */
47          0x84, 0x00, 0x84,    /* 13: 暗い紫 */
48          0x00, 0x84, 0x84,    /* 14: 暗い水色 */
```

```
49          0x84, 0x84, 0x84    /* 15: 暗い灰色 */
50      };
51      set_palette(0, 15, table_rgb);
52      return;
53
54      /* static char 命令は、データにしか使えないけど DB 命令相当 */
55  }
56
57  void set_palette(int start, int end, unsigned char *rgb)
58  {
59      int i, eflags;
60      eflags = io_load_eflags();    /* 割り込み許可フラグの値を記録する */
61      io_cli();                     /* 許可フラグを 0 にして割り込み禁止にする */
62      io_out8(0x03c8, start);
63      for (i = start; i <= end; i++) {
64          io_out8(0x03c9, rgb[0] / 4);
65          io_out8(0x03c9, rgb[1] / 4);
66          io_out8(0x03c9, rgb[2] / 4);
67          rgb += 3;
68      }
69      io_store_eflags(eflags);    /* 割り込み許可フラグを元に戻す */
70      return;
```

```
71 }
```

函数 init_palette 开头一段以 static 开始的语句，声明了一个 table_rgb。
函数 set_palette 中使用了 0x03c8、0x03c9 之类的设备号码，这些设备号码来自于 VGA 的文档。
调色板的访问步骤：

- 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，紧接着，按 R，G，B 的顺序写入 0x03c9。如果还想继续设定下一个调色板，则省略调色板号码，再按照 RGB 的顺序写入 0x03c9 即可。
- 如果想要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 读取三次。读出的顺序就是 R，G，B。如果要继续读出下一个调色板，同样是省略调色板号码，按 RGB 的顺序读出。
- 如果最初执行了 CLI，那么最后要执行 STI。

这里提到的 CLI（clear interrupt flag）是将中断标志置为 0 的指令，STI（set interrupt flag）是将中断标志置为 1 的指令。

set_palette 中想要做的事情是在设定调色板之前首先执行 CLI，处理结束后恢复中断标志，通过谢了函数 io_load_eflags 来读取最初的 eflags 值，处理结束后直接用 io_store_eflags 恢复。

naskfunc.nas

```
1 ; naskfunc
2 ; TAB=4
3
4 [FORMAT "WCOFF"]                ; オブジェクトファイルを作るモード
```

```
[INSTRSET "i486p"]                ; 486 の命令まで使いたいという記述
[BITS 32]                         ; 32 ビットモード用の機械語を作らせる
[FILE "naskfunc.nas"]             ; ソースファイル名情報

        GLOBAL    _io_hlt, _io_cli, _io_sti, _io_stihlt
        GLOBAL    _io_in8,  _io_in16,  _io_in32
        GLOBAL    _io_out8, _io_out16, _io_out32
        GLOBAL    _io_load_eflags, _io_store_eflags

[SECTION .text]

_io_hlt:    ; void io_hlt(void);
        HLT
        RET

_io_cli:    ; void io_cli(void);
        CLI
        RET

_io_sti:    ; void io_sti(void);
        STI
        RET
```

```
27
28 _io_stihlt:    ; void io_stihlt(void);
29         STI
30         HLT
31         RET
32
33 _io_in8:    ; int io_in8(int port);
34         MOV        EDX,[ESP+4]        ; port
35         MOV        EAX,0
36         IN        AL,DX
37         RET
38
39 _io_in16:    ; int io_in16(int port);
40         MOV        EDX,[ESP+4]        ; port
41         MOV        EAX,0
42         IN        AX,DX
43         RET
44
45 _io_in32:    ; int io_in32(int port);
46         MOV        EDX,[ESP+4]        ; port
47         IN        EAX,DX
48         RET
```

```
49
50 _io_out8:    ; void io_out8(int port, int data);
51         MOV         EDX,[ESP+4]         ; port
52         MOV         AL,[ESP+8]         ; data
53         OUT         DX,AL
54         RET
55
56 _io_out16:    ; void io_out16(int port, int data);
57         MOV         EDX,[ESP+4]         ; port
58         MOV         EAX,[ESP+8]         ; data
59         OUT         DX,AX
60         RET
61
62 _io_out32:    ; void io_out32(int port, int data);
63         MOV         EDX,[ESP+4]         ; port
64         MOV         EAX,[ESP+8]         ; data
65         OUT         DX,EAX
66         RET
67
68 _io_load_eflags:    ; int io_load_eflags(void);
69         PUSHFD        ; PUSH EFLAGS という意味
70         POP         EAX
```

```
71          RET
72
73  _io_store_eflags:    ; void io_store_eflags(int eflags);
74          MOV         EAX,[ESP+4]
75          PUSH    EAX
76          POPFD         ; POP EFLAGS という意味
77          RET
```

程序中，PUSHFD 是“push flags double-world”的缩写，POPFD 是“pop flags double-world”的缩写。

7　绘制矩形（harib01g）

开始绘制一些图形。

首先从 VRAM 与画面上的“点”的关系开始说起。

在当前画面模式中，画面上有 320×200（64000）个像素。假设左上点的坐标是（0,0），右下点的坐标是（319319），那么像素坐标（x,y）对应的 VRAM 地址按下式计算：

$$0xa0000 + x + y * 320$$

按照上式计算像素的地址，往该地址的内存里存放某种颜色的号码，那么画面上该像素的位置就出现相应的颜色。这样就画了一个点。继续增加 x 的值，循环以上操作，就能画一条直线，再向下循环这条直线，就能画出很多直线，组成一个有填充色的长方形。

†projects\04_day\harib01g

bootpack.c

```
1 #define COL8_000000        0
2 #define COL8_FF0000        1
3 #define COL8_00FF00        2
4 #define COL8_FFFF00        3
5 #define COL8_0000FF        4
6 #define COL8_FF00FF        5
7 #define COL8_00FFFF        6
8 #define COL8_FFFFFF        7
9 #define COL8_C6C6C6        8
10 #define COL8_840000        9
11 #define COL8_008400        10
12 #define COL8_848400        11
13 #define COL8_000084        12
14 #define COL8_840084        13
15 #define COL8_008484        14
16 #define COL8_848484        15
17
18 void HariMain(void)
19 {
20     char *p; /* p という変数は、BYTE [...] 用の番地 */
21
```

```
22      init_palette(); /* パレットを設定 */
23
24      p = (char *) 0xa0000; /* 番地を代入 */
25
26      boxfill8(p, 320, COL8_FF0000,  20,  20, 120, 120);
27      boxfill8(p, 320, COL8_00FF00,  70,  50, 170, 150);
28      boxfill8(p, 320, COL8_0000FF, 120,  80, 220, 180);
29
30      for (;;) {
31          io_hlt();
32      }
33  }
34
35  void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int y0, int x1, int y1)
36  {
37      int x, y;
38      for (y = y0; y <= y1; y++) {
39          for (x = x0; x <= x1; x++)
40              vram[y * xsize + x] = c;
41      }
42      return;
43  }
```

上面的程序写了一个 boxfill8 的函数，被调用 3 次，绘制了 3 个矩形。

8　今天的成果（harib01h）

绘制一个简单的带任务条的系统桌面，简单到简陋……竟然有凹凸效果！

†projects\04_day\harib01h

bootpack.c

```
1  void HariMain(void)
2  {
3      char *vram;
4      int xsize, ysize;
5
6      init_palette();
7      vram = (char *) 0xa0000;
8      xsize = 320;
9      ysize = 200;
10
11     boxfill8(vram, xsize, COL8_008484,  0,         0,          xsize -  1, ysize - 29);
12     boxfill8(vram, xsize, COL8_C6C6C6,  0,         ysize - 28, xsize -  1, ysize - 28);
13     boxfill8(vram, xsize, COL8_FFFFFF,  0,         ysize - 27, xsize -  1, ysize - 27);
14     boxfill8(vram, xsize, COL8_C6C6C6,  0,         ysize - 26, xsize -  1, ysize -  1);
15
```

```
16      boxfill8(vram, xsize, COL8_FFFFFF,  3,         ysize - 24, 59,         ysize - 24);
17      boxfill8(vram, xsize, COL8_FFFFFF,  2,         ysize - 24,  2,         ysize -  4);
18      boxfill8(vram, xsize, COL8_848484,  3,         ysize -  4, 59,         ysize -  4);
19      boxfill8(vram, xsize, COL8_848484, 59,         ysize - 23, 59,         ysize -  5);
20      boxfill8(vram, xsize, COL8_000000,  2,         ysize -  3, 59,         ysize -  3);
21      boxfill8(vram, xsize, COL8_000000, 60,         ysize - 24, 60,         ysize -  3);
22
23      boxfill8(vram, xsize, COL8_848484, xsize - 47, ysize - 24, xsize -  4, ysize - 24);
24      boxfill8(vram, xsize, COL8_848484, xsize - 47, ysize - 23, xsize - 47, ysize -  4);
25      boxfill8(vram, xsize, COL8_FFFFFF, xsize - 47, ysize -  3, xsize -  4, ysize -  3);
26      boxfill8(vram, xsize, COL8_FFFFFF, xsize -  3, ysize - 24, xsize -  3, ysize -  3);
27
28      for (;;) {
29          io_hlt();
30      }
31  }
```

第 5 天　结构体、文字显示与 GDT/IDT 初始化

1　接收启动信息（harib02a）

在 bootpack.c 里面中直接将 0xa0000、320、200 这种数字直接写入程序，而本来这些值应该从 asmhead.nas 先前保存下来的值中取。如果不这样，当画面模式改变时，系统就不能正常运行。

使用指针来取得这些值。

†projects\05_day\harib02a

bootpack.c 节选

```
1 void HariMain(void)
2 {
3     char *vram;
4     int xsize, ysize;
5     short *binfo_scrnx, *binfo_scrny;
6     int *binfo_vram;
7
```

```
8     init_palette();
9     binfo_scrnx = (short *) 0x0ff4;
10    binfo_scrny = (short *) 0x0ff6;
11    binfo_vram = (int *) 0x0ff8;
12    xsize = *binfo_scrnx;
13    ysize = *binfo_scrny;
14    vram = (char *) *binfo_vram;
```

这里的 0x0ff4 之类的地址仅仅是为了与 asmhead.nas 保持一致才出现的。
另外，把显示画面背景的部分独立出来，单独做成一个函数 init_screen。

2　试用结构体（harib02b）

使用结构体的方式重写主程序。†projects\05_day\harib02b

bootpack.c 节选

```
1 struct BOOTINFO {
2     char cyls, leds, vmode, reserve;
3     short scrnx, scrny;
4     char *vram;
5 };
6
7 void HariMain(void)
```

```
8 {
9     char *vram;
10    int xsize, ysize;
11    struct BOOTINFO *binfo;
12
13    init_palette();
14    binfo = (struct BOOTINFO *) 0x0ff0;
15    xsize = (*binfo).scrnx;
16    ysize = (*binfo).scrny;
17    vram = (*binfo).vram;
```

3　试用箭头记号（harib02c）

C 语言中常常会用到类似于 (*binfo).scrnx 的表现手法，因此出现了一种不使用括号的省略表现方式，即 binfo->scrnx，称之为箭头标记方式。†projects\05_day\harib02c

bootpack.c 节选

```
1 void HariMain(void)
2 {
3     struct BOOTINFO *binfo = (struct BOOTINFO *) 0x0ff0;
4
5     init_palette();
6     init_screen(binfo->vram, binfo->scrnx, binfo->scrny);
```

上面几小节都是 C 语言的写法问题，编译成机器语言以后几乎没什么差别。只是后面的这些写法更简洁清晰一些。

4　显示字符（harib02d）

使用 8×16 的长方形像素点阵来表示字符。

像这种描述文字形状的数据称为字体数据，通过数组来保存，写入程序。†projects\05_day\harib02d

bootpack.c 节选

```
1    static char font_A[16] = {
2        0x00, 0x18, 0x18, 0x18, 0x18, 0x24, 0x24, 0x24,
3        0x24, 0x7e, 0x42, 0x42, 0x42, 0xe7, 0x00, 0x00
4    };
```

用 for 语句将画 8 个像素的程序循环 16 遍，就可以显示出一个字符了。

bootpack.c 节选

```
1 void putfont8(char *vram, int xsize, int x, int y, char c, char *font)
2 {
3     int i;
4     char *p, d /* data */;
5     for (i = 0; i < 16; i++) {
6         p = vram + (y + i) * xsize + x;
```

```
7         d = font[i];
8         if ((d & 0x80) != 0) { p[0] = c; }
9         if ((d & 0x40) != 0) { p[1] = c; }
10        if ((d & 0x20) != 0) { p[2] = c; }
11        if ((d & 0x10) != 0) { p[3] = c; }
12        if ((d & 0x08) != 0) { p[4] = c; }
13        if ((d & 0x04) != 0) { p[5] = c; }
14        if ((d & 0x02) != 0) { p[6] = c; }
15        if ((d & 0x01) != 0) { p[7] = c; }
16    }
17    return;
18 }
```

运行“make run”，可以显示大写字符“A”出来。

5　增加字体（harib02e）

使用已经做好的字体库 hankaku.txt（程序目录下有）。

§

由于字库仅是简单的字符，需要专用的“编译器”来使字库可以被程序调用。

makefont.exe 将上面的文本文件（256 个字符的字体文件）读进来，然后输出成 16×256=4096 字节的 hankaku.bin 文件。使用 bin2obj.exe 加上链接所必要的接口信息，将它变成目标文件。

如果在 C 语言中使用这种字体数据，只需要写上以下语句就可以了。

```
extern char hankaku[4096];
```

像这种在源程序以外准备的数据，都需要加上 extern 属性。

§

OSAKA 的字体数据，依照一般的 ASCII 字符编码，含有 256 个字符。A 的字符编码是 0x41，所以 A 的字体数据放在在“hankaku+0x41*16”开始的 16 个字节里。C 语言中的字符编码可以用‘A’来表示，即可以写成“hankaku+'A'*16”。

程序添加了“ABC 123”，运行以下试试看。

†projects\05_day\harib02e

bootpack.c

```
1  void HariMain(void)
2  {
3      struct BOOTINFO *binfo = (struct BOOTINFO *) 0x0ff0;
4      extern char hankaku[4096];
5
6      init_palette();
7      init_screen(binfo->vram, binfo->scrnx, binfo->scrny);
8      putfont8(binfo->vram, binfo->scrnx,  8, 8, COL8_FFFFFF, hankaku + 'A' * 16);
9      putfont8(binfo->vram, binfo->scrnx, 16, 8, COL8_FFFFFF, hankaku + 'B' * 16);
10     putfont8(binfo->vram, binfo->scrnx, 24, 8, COL8_FFFFFF, hankaku + 'C' * 16);
11     putfont8(binfo->vram, binfo->scrnx, 40, 8, COL8_FFFFFF, hankaku + '1' * 16);
```

```
12      putfont8(binfo->vram, binfo->scrnx, 48, 8, COL8_FFFFFF, hankaku + '2' * 16);
13      putfont8(binfo->vram, binfo->scrnx, 56, 8, COL8_FFFFFF, hankaku + '3' * 16);
14
15      for (;;) {
16          io_hlt();
17      }
18  }
```

6　显示字符串（harib02f）

可以直接显示字符串，而不是一个个字符写在程序里。

†projects\05_day\harib02f

bootpack.c

```
1  void putfonts8_asc(char *vram, int xsize, int x, int y, char c, unsigned char *s)
2  {
3      extern char hankaku[4096];
4      for (; *s != 0x00; s++) {
5          putfont8(vram, xsize, x, y, c, hankaku + *s * 16);
6          x += 8;
7      }
8      return;
9  }
```

使用显示字符串的方法来炫一下。

bootpack.c

```
void HariMain(void)
{
    struct BOOTINFO *binfo = (struct BOOTINFO *) 0x0ff0;

    init_palette();
    init_screen(binfo->vram, binfo->scrnx, binfo->scrny);
    putfonts8_asc(binfo->vram, binfo->scrnx,  8,  8, COL8_FFFFFF, "ABC 123");
    putfonts8_asc(binfo->vram, binfo->scrnx, 31, 31, COL8_000000, "Haribote OS.");
    putfonts8_asc(binfo->vram, binfo->scrnx, 30, 30, COL8_FFFFFF, "Haribote OS.");

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

7　显示变量值（harib02g）

显示变量的值，这个很有用，因为这里没有调试器，调试系统不是很容易，所以还是打印变量值来看看到底哪里不对了。由于在自制操作系统中不能随便使用 printf 函数，但可以使用 sprintf，它不是按制定格式输出，只

是将输出的内容作为字符串写在内存中，所以可以应用于所有操作系统。

这里用的 sprintf 函数是本次使用的名为 GO 的 C 编译器附带的函数。

§

添加头文件`#include<stdio.h>`。

sprintf 函数的使用方法是：sprintf(地址，格式，值，值，值，……)。其中，格式和 C 语言中的定义一样，`%d,%x` 这种。

```
1    sprintf(s, "scrnx = %d", binfo->scrnx);
2    putfonts8_asc(binfo->vram, binfo->scrnx, 16, 64, COL8_FFFFFF, s);
```

8　显示鼠标指针（harib02h）

将鼠标指针的大小定为 16×16。

†projects\05_day\harib02h

bootpack.c

```
1 void init_mouse_cursor8(char *mouse, char bc)
2 /* マウスカーソルを準備（16x16） */
3 {
4     static char cursor[16][16] = {
5         "**************..",
```

```
        "*OOOOOOOOOOO*...",
        "*OOOOOOOOOO*....",
        "*OOOOOOOOO*.....",
        "*OOOOOOOO*......",
        "*OOOOOOO*.......",
        "*OOOOOOO*.......",
        "*OOOOOOOO*......",
        "*OOOO**OOO*.....",
        "*OOO*..*OOO*....",
        "*OO*....*OOO*...",
        "*O*......*OOO*..",
        "**........*OOO*.",
        "*..........*OOO*",
        ".............*OO*",
        ".............***"
    };
    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;
```

```
28              }
29              if (cursor[y][x] == 'O') {
30                  mouse[y * 16 + x] = COL8_FFFFFF;
31              }
32              if (cursor[y][x] == '.') {
33                  mouse[y * 16 + x] = bc;
34              }
35          }
36      }
37      return;
38  }
```

变量 bc 是指 back-color，也就是背景色。

要将背景色显示出来，只要将 buf 中的数据复制到 vram 中去就可以了。

bootpack.c

```
1  void putblock8_8(char *vram, int vxsize, int pxsize,
2      int pysize, int px0, int py0, char *buf, int bxsize)
3  {
4      int x, y;
5      for (y = 0; y < pysize; y++) {
6          for (x = 0; x < pxsize; x++) {
7              vram[(py0 + y) * vxsize + (px0 + x)] = buf[y * bxsize + x];
8          }
```

```
9       }
10      return;
11  }
```

函数中，vram 和 vxsize 是关于 VRAM 的信息，值分别为 0xa0000 和 320。pxsize 和 pysize 是想要显示的图形的大小，这里是鼠标指针的大小 16。px0 和 py0 指定图形在画面上的显示位置。最后的 buf 和 bxsize 分别指定图形的存放地址和每一行含有的像素数。

调用函数：

bootpack.c

```
1       init_mouse_cursor8(mcursor, COL8_008484);
2       putblock8_8(binfo->vram, binfo->scrnx, 16, 16, mx, my, mcursor, 16);
```

9　GDT 与 IDT 的初始化（harib02i）

上节中，鼠标显示出来了，但是无法移动。需要通过初始化 GDT 和 IDT 来完成移动。

GTD（global (segment) descriptor table），全局段号记录表。将这些数据整齐的排列在内存的某个地方，然后将内存的起始地址和有效设定个数放在 CPU 内被称为 GDTR 的特殊寄存器中，设定就完成了。

IDT（interrupt descriptor table），中断记录表。IDT 记录了 0~255 号中段号码与调用函数的对应关系。

†projects\05_day\harib02i

bootpack.c 节选

```
1   struct SEGMENT_DESCRIPTOR {
2       short limit_low, base_low;
```

```
3     char base_mid, access_right;
4     char limit_high, base_high;
5 };
6
7 struct GATE_DESCRIPTOR {
8     short offset_low, selector;
9     char dw_count, access_right;
10    short offset_high;
11 };
12
13 void init_gdtidt(void)
14 {
15    struct SEGMENT_DESCRIPTOR *gdt = (struct SEGMENT_DESCRIPTOR *) 0x00270000;
16    struct GATE_DESCRIPTOR    *idt = (struct GATE_DESCRIPTOR    *) 0x0026f800;
17    int i;
18
19    /* GDT の初期化 */
20    for (i = 0; i < 8192; i++) {
21        set_segmdesc(gdt + i, 0, 0, 0);
22    }
23    set_segmdesc(gdt + 1, 0xffffffff, 0x00000000, 0x4092);
24    set_segmdesc(gdt + 2, 0x0007ffff, 0x00280000, 0x409a);
```

```
25      load_gdtr(0xffff, 0x00270000);
26
27      /* IDT の初期化 */
28      for (i = 0; i < 256; i++) {
29          set_gatedesc(idt + i, 0, 0, 0);
30      }
31      load_idtr(0x7ff, 0x0026f800);
32
33      return;
34  }
35
36  void set_segmdesc(struct SEGMENT_DESCRIPTOR *sd, unsigned int limit, int base, int ar)
37  {
38      if (limit > 0xfffff) {
39          ar |= 0x8000; /* G_bit = 1 */
40          limit /= 0x1000;
41      }
42      sd->limit_low    = limit & 0xffff;
43      sd->base_low     = base & 0xffff;
44      sd->base_mid     = (base >> 16) & 0xff;
45      sd->access_right = ar & 0xff;
46      sd->limit_high   = ((limit >> 16) & 0x0f) | ((ar >> 8) & 0xf0);
```

```
47      sd->base_high    = (base >> 24) & 0xff;
48      return;
49  }
50
51  void set_gatedesc(struct GATE_DESCRIPTOR *gd, int offset, int selector, int ar)
52  {
53      gd->offset_low   = offset & 0xffff;
54      gd->selector     = selector;
55      gd->dw_count     = (ar >> 8) & 0xff;
56      gd->access_right = ar & 0xff;
57      gd->offset_high  = (offset >> 16) & 0xffff;
58      return;
59  }
```

SEGMENT_DESCRIPTOR 中存放 GDT 的 8 字节的内容，GATE_DESCRIPTOR 中存放 IDT 的 8 字节内容。

变量 gdt 被赋值为 0x00270000，也就是将 0x270000~0x27ffff 设为 GDT（从内存分布图可以看到这一块地方没有被使用）。

变量 idt 被设为 0x26f800~0x26ffff。

§

```
for (i = 0; i < 8192; i++) {
        set_segmdesc(gdt + i, 0, 0, 0);
    }
```

for 循环完成对 8192 个段的设定，将它们的上限（limit，指段的字节数 -1），基址、访问权限都设置为 0。

```
    set_segmdesc(gdt + 1, 0xffffffff, 0x00000000, 0x4092);
    set_segmdesc(gdt + 2, 0x0007ffff, 0x00280000, 0x409a);
```

以上语句对段号为 1 和 2 的两个段进行设定。段号为 1 的段，上限值为 0xffffffff 即大小正好为 4GB，地址是 0，它表示的是 CPU 所能管理的全部内存本身。段属性设置为 0x4092。段号为 2 的段，它的大小是 512KB，地址是 0x280000，这正好是为 bootpack.hrb 准备的，用这个段，就可以执行 bootpack.hrb，因为 bootpack.hrb 是以 ORG 0 为前提翻译成机器语言的。

```
  load_gdtr(0xffff, 0x00270000);
```

因为 C 语言不能给 GDTR 赋值，借助汇编语言来赋值。

后面关于 IDT 的描述跟前面一样。

后面的 set_segmdesc 和 set_gatedesc 函数中用到了一些新的运算符，“|”、“/”、“»” 等，自己看吧。

第 6 天　分割编译与中断处理

1　分割源文件（harib03a）

将 bootpack.c 分割成了 graphic.c、dsctbl.c、bootpack.c。

2　整理 Makefile（harib03b）

将处理相同事情的部分归纳，按照一般规则来处理。

3　整理头文件（harib03c）

将各个源文件重复部分去掉，归纳起来，放到 bootpack.h 的头文件里。在具体的源文件里引用头文件。

4　意犹未尽

介绍上一章 naskfunc.nas 的load_gdtr 函数：

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

函数用来将指定的段上限（limit）和地址赋值给名为 GDTR 的 48 位寄存器。这是一个很特别的 48 位寄存器，并不能用我们常用的 MOV 指令来赋值。给它赋值的时候，唯一的方法就是指定内存地址，从指定的地址读取 6 个字节，然后复制给 GDTR 寄存器。完成这一任务的就是 LGDT。

该寄存器的低 16 位是段上限，它等于“GDT 的有效字节数 -1”。剩下的高 32 位代表 GDT 的开始地址。

在最初执行这个函数的时候，DWORD[ESP+4] 里存放的是段上限，DWORD[ESP+8] 里存放的是地址。具体到实际的数值，就是 0x000fff 和 0x00270000。把它们按字节写出来的话就成了FF FF 00 00 00 27 00(注意地位放在内存地址小的字节里)。为了执行 LGDT，希望把它们排列成FF FF 00 00 00 27 00 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] 里。这样结果就成了[FF FF FF FF 00 27 00 00]，如果从[ESP+6] 开始读 6 字节的话，正好是我们想要的结果。

书中补充了 dsctbl.c 里的set_segmdesc 函数及相关段的知识。具体内容参见书本。

5　初始化 PIC（harib03d）

为了达到鼠标指针移动的目的，必须使用中断，而要使用中断必须将 GDT 和 IDT 正确无误的初始化。此时，需要初始化 PIC。

PIC（programmable interrupt controller），可编程中断控制器。PIC 将 8 个中断信号（IRQ）集合成一个中断信号的装置。PIC 监视输入管脚的 8 个中断信号，只要有一个中断信号进来，就将唯一的输出管脚信号变成 ON，并通知给 CPU。最初设计者希望通过增加 PIC 来处理更多的中断信号，把中断信号设计成 15 个，增设了 2 个 PIC（主从 PIC）。

§

†projects\06_day\harib03d

int.c 的主要组成部分

```
1  void init_pic(void)
2  /* PIC の初期化 */
3  {
4      io_out8(PIC0_IMR,  0xff  ); /* 全ての割り込みを受け付けない */
5      io_out8(PIC1_IMR,  0xff  ); /* 全ての割り込みを受け付けない */
6
7      io_out8(PIC0_ICW1, 0x11  ); /* エッジトリガモード */
8      io_out8(PIC0_ICW2, 0x20  ); /* IRQ0-7 は、INT20-27 で受ける */
9      io_out8(PIC0_ICW3, 1 << 2); /* PIC1 は IRQ2 にて接続 */
10     io_out8(PIC0_ICW4, 0x01  ); /* ノンバッファモード */
```

```
11
12      io_out8(PIC1_ICW1, 0x11  ); /* エッジトリガモード */
13      io_out8(PIC1_ICW2, 0x28  ); /* IRQ8-15 は、INT28-2f で受ける */
14      io_out8(PIC1_ICW3, 2     ); /* PIC1 は IRQ2 にて接続 */
15      io_out8(PIC1_ICW4, 0x01  ); /* ノンバッファモード */
16
17      io_out8(PIC0_IMR,  0xfb  ); /* 11111011 PIC1 以外は全て禁止 */
18      io_out8(PIC1_IMR,  0xff  ); /* 11111111 全ての割り込みを受け付けない */
19
20      return;
21  }
```

以上是 PIC 的初始化程序。从 CPU 的角度看，PIC 是外部设备，CPU 使用 OUT 指令进行操作，程序中的 PIC0 和 PIC1，分别指主 PIC 和从 PIC。PIC 内部有很多寄存器，用端口号码对彼此进行区别，以决定是写入哪一个寄存器。具体的端口号码写在 bootpack.h 里。

§

PIC 寄存器是 8 位寄存器。IMR（interrupt mask register）是中断屏蔽寄存器。8 位分别对应 8 路 IRQ 信号。如果某一位的值是 1，则该位所对应的 IRQ 信号被屏蔽，PIC 就忽略该路信号。这主要是因为，正在对中断设定进行更改时，如果再接受别的中断信号会引起混乱，为了防止这种情况发生，就必须屏蔽中断。还有，如果某个 IRQ 没有连接任何设备的话，静电干扰也可能会引起反应，导致操作系统混乱，所以也要屏蔽掉这类干扰。

ICW（initial control word）是初始化控制数据。ICW 有 4 个，分别编号为 1~4，共有 4 个字节的数据。ICW1 和 ICW4 主板配线方式、中断信号的电气特性等有关，不再叙述。ICW3 是有关主 -从连接的设定，对主

PIC 而言，第几号 IRQ 与从 PIC 相连，是用 8 位来设定的。对从 PIC 而言，该 PIC 与主 PIC 的第几号相连是用 3 位来设定。

§

不同的操作系统可以进行独特设定的只有 ICW2。它决定了 IRQ 以哪一个信号中断通知 CPU。

这次是以`INT 0x20~0x2f` 接收中断信号 IRQ0～15 而设定的，因为 INT 0x00～0x1f 用于应用程序想对操作系统干坏事的时候 CPU 内部系统保护通知，所以直接使用`INT 0x20~0x2f`。

6　中断处理程序的制作（harib03e）[1]

鼠标是 IRQ12，键盘是 IRQ1，编写了用于 INT 0x2c 和 INT 0x21 的中断处理程序（handler），即中断发生时要调用的程序。

int.c 节选

```
1 void inthandler21(int *esp)
2 /* PS/2 キーボードからの割り込み */
3 {
4     struct BOOTINFO *binfo = (struct BOOTINFO *) ADR_BOOTINFO;
5     boxfill8(binfo->vram, binfo->scrnx, COL8_000000, 0, 0, 32 * 8 - 1, 15);
6     putfonts8_asc(binfo->vram, binfo->scrnx, 0, 0, COL8_FFFFFF, "INT 21 (IRQ-1) : PS/2 keyboard");
7     for (;;) {
```

[1]书中讲到了 IRQ1 和 IRQ12 的中断处理程序，光盘里面补充了 IRQ7 的中断处理程序。详细解释见书。

```
8           io_hlt();
9       }
10  }
```

程序只是显示一条信息，然后保持在待机状态。鼠标的程序也几乎完全相同。

§

中断执行后不能执行 return，而是必须执行 IRETD 指令。借助汇编语言修改 naskfunc.nas。

†projects\06_day\harib03e

naskfunc.nas 节选

```
1   _inthandler21, _inthandler27, _inthandler2c
2
3   _asm_inthandler21:
4           PUSH    ES
5           PUSH    DS
6           PUSHAD
7           MOV         EAX,ESP
8           PUSH    EAX
9           MOV         AX,SS
10          MOV         DS,AX
11          MOV         ES,AX
12          CALL    _inthandler21
```

```
13          POP         EAX
14          POPAD
15          POP         DS
16          POP         ES
17          IRETD
```

其中，PUSHAD 相当于：

```
1 PUSH EAX
2 PUSH ECX
3 PUSH EDX
4 PUSH EBX
5 PUSH ESP
6 PUSH EBP
7 PUSH ESI
8 PUSH EDI
```

POPAD 相当于按以上相反的顺序，把它们全都 POP 出来。

§

函数只是将寄存器的值保存到栈里，然后将 DS 和 ES 调整到与 SS 相等，再调用 _inthandler21，返回以后，将所有寄存器的值再返回到原来的值，然后执行 IRETD。

§

将这个函数注册到 IDT 中去，在 dsctbl.c 的`init_gdtidt` 里加入以下语句。

```
1    /* IDT の設定 */
2    set_gatedesc(idt + 0x21, (int) asm_inthandler21, 2 * 8, AR_INTGATE32);
3    set_gatedesc(idt + 0x27, (int) asm_inthandler27, 2 * 8, AR_INTGATE32);
4    set_gatedesc(idt + 0x2c, (int) asm_inthandler2c, 2 * 8, AR_INTGATE32);
```

`asm_inthandler21` 注册在 idt 的第 0x21 号。这里的 2×8 表示的是`asm_inthandler21` 属于哪一个段，即段号是 2，乘以 8 是因为低三位有别的意思，这里低三位必须为 0，相当于写成“2«3”。

号码为 2 的段，正好涵盖了整个 bootpack.hrb。最后的`AR_CODE32_ER` 将 IDT 的属性设定为 0x008e。它表示这是用于中断处理的有效设定。

```
1 set_segmdesc(gdt + 2, LIMIT_BOTPAK, ADR_BOTPAK, AR_CODE32_ER);
```

§

对 bootpack.c 的 Harimain 的补充。“`io_sti();`”仅仅是执行 STI 指令，它是 CLI 的逆指令。在 HariMain 的最后，修改了 PIC 的 IMR，以便接收来自键盘和鼠标的中断。

§

运行程序会发现，键盘的中断响应正常，鼠标的却不行。

第 7 天　FIFO 与鼠标控制

1　获取按键编码（harib04a）

现在，只要在键盘上按一按键，就会在屏幕上显示信息，其他的我们什么也做不了。我们将程序改善一下，让程序在按下一个键后不会结束，而是把按键的编码在画面上显示出来，这样就可以切实完成中断处理程序了。

更改的程序是 init.c 程序中的 inthandler21 函数，具体如下：

```
1 #define PORT_KEYDAT        0x0060
2
3 void inthandler21(int *esp)
4 {
5     struct BOOTINFO *binfo = (struct BOOTINFO *) ADR_BOOTINFO;
6     unsigned char data, s[4];
7     io_out8(PIC0_OCW2, 0x61);    /* IRQ-01 受付完了を PIC に通知 */
8     data = io_in8(PORT_KEYDAT);
```

```
9
10      sprintf(s, "%02X", data);
11      boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 0, 16, 15, 31);
12      putfonts8_asc(binfo->vram, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
13
14      return;
15  }
```

§

程序中`io_out8(PIC0_OCW2, 0x61);`这句话用来通知 PIC 已经知道发生了 IRQ1 中断。如果是 IRQ3，则写成 0x63。执行这句话之后，PIC 继续时刻监视 IRQ1 中断是否发生，反过来，如果忘记了执行这句话，PIC 就不再监视 IRQ1 中断，不管下次由键盘输入什么信息，系统都感知不到了。

§

程序所完成的，是将接收到的按键编码显示在画面上，然后结束中断处理。

2　加快中断处理（harib04b）

程序里有一个问题，那就是字符显示的内容被放在了中断处理程序中。

所谓中断处理，基本上就是打断 CPU 本来的工作，加塞要求进行处理，所以必须完成得干净利索。而且中断处理进行期间，不再接收别的中断。所以如果我们处理键盘的中断速度太慢，就会出现鼠标的运动不连贯、不能从网上接收数据等情况。

另一方面，字符显示要花大块的时间来进行处理。仅仅画一个字符，就要执行 8 ×16=128 次 if 语句，来判断是否要往 VRAM 里描画该像素。如果判定为描画该像素，还要执行内存写入指令。而且为确定具体往内存的哪个地方写，还要做很多地址计算。这些事情，在我们看来，或许只是一瞬间的事情，但在计算机看来，可不是这样。

谁也不知道其他中断会在哪个瞬间到来。事实上，很可能在键盘输入的同时，就有数据正在从网上下载，而 PIC 在等待键盘中断处理的结束。

§

解决方案是先将按键的编码接收下来，保存到变量里去，然后由 HariMain 偶尔去看看这个变量。如果发现有了数据，就把它显示出来。

```
1  struct KEYBUF keybuf;
2
3  void inthandler21(int *esp)
4  {
5      unsigned char data;
6      io_out8(PIC0_OCW2, 0x61);    /* IRQ-01 受付完了を PIC に通知 */
7      data = io_in8(PORT_KEYDAT);
8      if (keybuf.flag == 0) {
9          keybuf.data = data;
10         keybuf.flag = 1;
11     }
```

```
12      return;
13  }
```

考虑到键盘的输入时需要缓冲区，先定义一个构造体，命名为 keybuf。其中的 flag 变量用于表示这个缓冲区是否为空。如果 flag 是 0，表示缓冲区为空；如果 flag 为 1，表示缓冲区中有数据。那么，如果缓冲区有数据，而这时又来了一个中断，那么该怎么办呢？先不管哈

§

bootpack.c 中 HariMain 函数节选

```
1  for (;;) {
2          io_cli();
3          if (keybuf.flag == 0) {
4              io_stihlt();
5          } else {
6              i = keybuf.data;
7              keybuf.flag = 0;
8              io_sti();
9              sprintf(s, "%02X", i);
10             boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 0, 16, 15, 31);
11             putfonts8_asc(binfo->vram, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
12         }
13     }
```

开始先用`io_cli` 指令屏蔽中断。

如果 flag 的值是 0，说明键还没有被按下，keybuf.data 里没有值保存下来。在 keybuf.data 里有值被保存下来之前我们无事可做，所以干脆去执行`io_hlt`。但是，由于已经执行了`io_cli` 屏蔽了中断，如果这样就去执行 HLT 指令的话，即使没有什么键被按下，程序也不会有任何反应。所以 STI 和 HLT 都要执行，而执行这两个指令的函数就是`io_stihlt`。执行 HLT 指令以后，如果收到了 PIC 的通知，CPU 就会被唤醒。这样，CPU 首先会去执行中断处理程序。中断处理程序执行完之后，又回到 for 语句的开头，再执行`io_cli` 函数。

如果通过中断处理函数在 keybuf.data 里存入了按键编码，else 语句就会被执行。先将这个键码（keybuf.data）值保存到变量 i 里，然后将 flag 置为 0 表示键码值清为空，最后再通过`io_sti` 语句开放中断。

§

运行程序，能够顺利执行……但是，右 Ctrl 键的显示是有问题的。

查阅资料得知，当按下右 Ctrl 键时，会产生两个字节的键码值“E0 1D”，而松开这个键之后，会产生两个字节的键码值“E0 9D”。在一次产生两个字节键码值的情况下，因为键盘内部电路一次只能发送一个字节，所以一次按键会产生两次中断，第一次中断时发送 E0，第二次中断发生 1D。

在 harib04a 中，以上两次中断所发送的值都能收到，瞬间显示 E0 后，紧接着又显示 1D 或者 9D。而在 harib04b 中，HariMain 函数在收到 E0 之前，又收到前一次按键产生的 1D 或者 9D，而这个字节被舍弃了。

3　制作 FIFO 缓冲区（harib04c）

问题在于这里创建的缓冲区只存储一个字节，如果做一个能够存储多字节的缓冲区，那么它就不会满，问题也就解决了。

根据这种思路，有一下程序：

```
struct KEYBUF {
    unsigned char data[32];
    int next;
};

void inthandler21(int *esp)
{
    unsigned char data;
    io_out8(PIC0_OCW2, 0x61);    /* IRQ-01 受付完了を PIC に通知 */
    data = io_in8(PORT_KEYDAT);
    if (keybuf.next < 32) {
        keybuf.data[keybuf.next] = data;
        keybuf.next++;
    }
    return;
}
```

keybuf.next 的起点是“0”，所以最初存储的数据是 keybuf.data[0]，共 32 个存储位置。

下一个存储位置用变量 next 来管理。这样就可以记住 32 个数据，而不会溢出，但是为保险起见，next 的值变成 32 之后，就舍去不要了。

§

取得数据的程序如下：

```
1    for (;;) {
2        io_cli();
3        if (keybuf.next == 0) {
4            io_stihlt();
5        } else {
6            i = keybuf.data[0];
7            keybuf.next--;
8            for (j = 0; j < keybuf.next; j++) {
9                keybuf.data[j] = keybuf.data[j + 1];
10           }
11           io_sti();
12           sprintf(s, "%02X", i);
13           boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 0, 16, 15, 31);
14           putfonts8_asc(binfo->vram, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
15       }
16   }
```

如果 next 不是 0，则说明至少有一个数据。最开始的一个数据肯定是放在`data[0]` 中的，将这个数据存入到变量 i 中去。这样，数就减少一个，所以将 next 减去 1。

接下来，for 循环中，数据的存放位置全部都向前移送一个位置。

§

此时，右 Ctrl 的处理运行正常。但从`data[0]` 取得数据后有关数据移送的处理不尽如人意。

数据移送处理本身没有什么不好，只是在禁止中断期间做数据移送处理有问题。但如果在数据移送处理前就允许中断的话，会搞乱要处理的数据，这当然不行。下面解决。

4　改善 FIFO 缓冲区（harib04d）

想开发一个不需要数据移送操作的 FIFO 型缓冲区。基本思路是：不仅维护下一个要写入数据的位置，还要维护下一个要读出数据的位置。这就像数据读出位置在追着数据写入位置跑一样。这样就不需要数据移送操作了。数据读出位置追上数据写入位置的时候，就相当于缓冲区为空，没有数据。

但是这样的缓冲区使用一段时间后，下一个数据写入位置会变成 31，而这时下一个数据读出位置可能已经是 29 或 30 什么的了。当下一个写入位置变成 32 的时候，就走到死胡同了。因为下面没地方可以写入数据了。

如果当下一个数据写入位置到达缓冲区终点时，数据读出位置也恰好到达缓冲区终点，也就是说缓冲区正好变空，那还好说。我们只要将下一个数据写入位置和下一个数据读出位置都再置为 0 就行了，就像转回去从头再来一样。

但是总还是会有数据读出位置没有追上数据写入位置的情况。这时，又不得不进行数据移送操作。原来是每次都要进行数据移送，而现在不用每次都做。

仔细想一下，当下一个数据写入位置到达缓冲区最末尾，缓冲区开头部分应该已经变空了（如果还没有变空，说明数据读出跟不上数据写入，只能把部分数据扔掉了）。因此如果下一个数据写入位置到了 32 以后，就强制性地将它设置为 0. 这样一来，下一个数据写入位置就跑到了下一个数据读出位置的后面，让人觉得怪怪的。但这无关紧要，没什么问题。

对下一个数据读出位置也做同样的处理，一旦到了 32 以后，就把它设置为从 0 开始继续读取数据。这样 32 字节的缓冲区就能一圈一圈地不停循环，长久使用。数据移送操作一次都不需要。

§

相应的代码如下：

bootpack.h 节选

```
struct KEYBUF {
    unsigned char data[32];
    int next_r, next_w, len;
};
```

变量 len 是指缓冲区能记录多少字节的数据。

int.c 节选

```
void inthandler21(int *esp)
{
    unsigned char data;
    io_out8(PIC0_OCW2, 0x61);    /* IRQ-01 受付完了を PIC に通知 */
    data = io_in8(PORT_KEYDAT);
    if (keybuf.len < 32) {
        keybuf.data[keybuf.next_w] = data;
        keybuf.len++;
        keybuf.next_w++;
```

```
10          if (keybuf.next_w == 32) {
11              keybuf.next_w = 0;
12          }
13      }
14      return;
15  }
```

读出数据程序如下：

```
1       for (;;) {
2           io_cli();
3           if (keybuf.len == 0) {
4               io_stihlt();
5           } else {
6               i = keybuf.data[keybuf.next_r];
7               keybuf.len--;
8               keybuf.next_r++;
9               if (keybuf.next_r == 32) {
10                  keybuf.next_r = 0;
11              }
12              io_sti();
13              sprintf(s, "%02X", i);
```

```
14              boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 0, 16, 15, 31);
15              putfonts8_asc(binfo->vram, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
16          }
17      }
```

5　整理 FIFO 缓冲区（harib04e）

将结构做成这样：

```
1 struct FIFO8 {
2     unsigned char *buf;
3     int p, q, size, free, flags;
4 };
```

如果我们将缓冲区大小固定成 32 字节的话，以后改起来就不方便了，所以把它定义成可变的。缓冲区的总字节数保存在变量 size 里。变量 free 用于保存缓冲区里没有数据的字节数。缓冲区地址保存在变量 buf 里。p 代表下一个数据写入地址，q 代表下一个数据读出地址。

```
1 void fifo8_init(struct FIFO8 *fifo, int size, unsigned char *buf)
2 /* FIFO バッファの初期化 */
3 {
4     fifo->size = size;
```

```
5     fifo->buf = buf;
6     fifo->free = size; /* 空き */
7     fifo->flags = 0;
8     fifo->p = 0; /* 書き込み位置 */
9     fifo->q = 0; /* 読み込み位置 */
10    return;
11 }
```

fifo8_init 是结构的初始化函数，用来设定各种初始值，也就是设定 FIFO8 结构的地址以及与结构有关的各种参数。

```
1 #define FLAGS_OVERRUN        0x0001
2
3 int fifo8_put(struct FIFO8 *fifo, unsigned char data)
4 /* FIFO へデータを送り込んで蓄える */
5 {
6     if (fifo->free == 0) {
7         /* 空きがなくてあふれた */
8         fifo->flags |= FLAGS_OVERRUN;
9         return -1;
10    }
11    fifo->buf[fifo->p] = data;
```

```
12      fifo->p++;
13      if (fifo->p == fifo->size) {
14          fifo->p = 0;
15      }
16      fifo->free--;
17      return 0;
18  }
```

fifo8_put 是往 FIFO 缓冲区存储 1 字节信息的函数。用 flags 这一变量来记录是否溢出。

```
1  int fifo8_get(struct FIFO8 *fifo)
2  /* FIFO からデータを一つとってくる */
3  {
4      int data;
5      if (fifo->free == fifo->size) {
6          /* バッファが空っぽのときは、とりあえず-1 が返される */
7          return -1;
8      }
9      data = fifo->buf[fifo->q];
10     fifo->q++;
11     if (fifo->q == fifo->size) {
12         fifo->q = 0;
```

```
13      }
14      fifo->free++;
15      return data;
16  }
```

fifo8_get 是从 FIFO 缓冲区取出 1 字节的函数。

```
1  int fifo8_status(struct FIFO8 *fifo)
2  /* どのくらいデータが溜まっているかを報告する */
3  {
4      return fifo->size - fifo->free;
5  }
```

fifo8_status 用来查看缓冲区状态。

使用以上函数，写成的程序段如下：

```
1  struct FIFO8 keyfifo;
2
3  void inthandler21(int *esp)
4  {
5      unsigned char data;
6      io_out8(PIC0_OCW2, 0x61);    /* IRQ-01 受付完了を PIC に通知 */
```

```
7     data = io_in8(PORT_KEYDAT);
8     fifo8_put(&keyfifo, data);
9     return;
10 }
```

MariMain 函数内容如下：

```
1     char s[40], mcursor[256], keybuf[32];
2
3     fifo8_init(&keyfifo, 32, keybuf);
4
5     for (;;) {
6         io_cli();
7         if (fifo8_status(&keyfifo) == 0) {
8             io_stihlt();
9         } else {
10            i = fifo8_get(&keyfifo);
11            io_sti();
12            sprintf(s, "%02X", i);
13            boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 0, 16, 15, 31);
14            putfonts8_asc(binfo->vram, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
15        }
16    }
```

程序运行正常！

6　总算讲到鼠标了（harib04f）

较计算机的历史来说，鼠标比较新，早起的计算机并不支持，这一点从鼠标的中段号码 IRQ12 这一很大的数字就可以看出。

鼠标作为计算机的一个外部设备开始使用的时候，几乎所有的操作系统都不支持它。在这种情况下，如果只是稍微一动鼠标就产生中断的话，那么使用那些操作系统的时候，就只好把鼠标先拔掉。这是很不方便的。为了不影响鼠标和其它设备的使用，主板上做了鼠标用的电路，但是只要不执行激活鼠标的指令，就不产生鼠标的中断信号。

所谓不产生中断信号，也就是说，即使从鼠标传来了数据，CPU 也不会接收。这样的话鼠标也就没必要传送数据了，否则会引起电路的混乱。所以，处于初期状态的鼠标，不管是滑动操作还是点击操作，都没有反应。

总而言之，我们必须发现指令，让下面两个装置有效，一个是鼠标控制电路，一个是鼠标本身。

§

控制电路的设定。事实上，鼠标控制电路包含着键盘控制电路里，如果键盘控制电路的初始化正常完成，鼠标电路控制器的激活也就完成了。

```
1 #define PORT_KEYDAT                0x0060
2 #define PORT_KEYSTA                0x0064
```

```
3 #define PORT_KEYCMD                0x0064
4 #define KEYSTA_SEND_NOTREADY    0x02
5 #define KEYCMD_WRITE_MODE        0x60
6 #define KBC_MODE                0x47
7
8 void wait_KBC_sendready(void)
9 {
10     /* キーボードコントローラがデータ送信可能になるのを待つ */
11     for (;;) {
12         if ((io_in8(PORT_KEYSTA) & KEYSTA_SEND_NOTREADY) == 0) {
13             break;
14         }
15     }
16     return;
17 }
18
19 void init_keyboard(void)
20 {
21     /* キーボードコントローラの初期化 */
22     wait_KBC_sendready();
23     io_out8(PORT_KEYCMD, KEYCMD_WRITE_MODE);
24     wait_KBC_sendready();
```

```
25      io_out8(PORT_KEYDAT, KBC_MODE);
26      return;
27  }
```

函数wait_KBC_sendready 的作用是让键盘控制电路做好准备动作，等待控制指令的到来。是因为虽然 CPU 的电路很快，但键盘控制电路却没有那么快。如果 CPU 不顾设备接收数据的能力，只是一个劲儿地发指令的话，有些指令会得不到执行，从而导致错误的结果。如果键盘控制电路可以接受 CPU 指令了，CPU 从设备号码 0x0064 处所读取的数据的倒数第二位（从低位开始数的第二位）应该是 0。在确认到这一位是 0 之前，程序一直通过 for 语句循环查询。

init_keyboard 一边确认可否往键盘控制电路传送信息，一边发送模式设定指令，指令中包含着要设定为何种模式。模式设定的指令是 0x60，利用鼠标模式的模式号码是 0x47。

在 HariMain 函数调用init_keyboard 函数，鼠标控制电路的准备就完成了。

§

开始发送激活鼠标的指令。所谓发送鼠标激活指令，归根到底还是要向键盘控制器发送指令。

```
1  #define KEYCMD_SENDTO_MOUSE     0xd4
2  #define MOUSECMD_ENABLE         0xf4
3
4  void enable_mouse(void)
5  {
6      /* マウス有効 */
```

```
7     wait_KBC_sendready();
8     io_out8(PORT_KEYCMD, KEYCMD_SENDTO_MOUSE);
9     wait_KBC_sendready();
10    io_out8(PORT_KEYDAT, MOUSECMD_ENABLE);
11    return; /* うまくいくと ACK(0xfa) が送信されてくる */
12 }
```

往键盘控制电路发送指令 0xd4，下一个数据就会自动发送给鼠标。激活的鼠标发送返回值 oxfa。
在 HariMain 中调用函数，运行程序，鼠标中断可用。

7　从鼠标接受数据（harib04g）

取出中断数据：

```
1 struct FIFO8 mousefifo;
2
3 void inthandler2c(int *esp)
4 /* PS/2 マウスからの割り込み */
5 {
6     unsigned char data;
7     io_out8(PIC1_OCW2, 0x64);    /* IRQ-12 受付完了を PIC1 に通知 */
8     io_out8(PIC0_OCW2, 0x62);    /* IRQ-02 受付完了を PIC0 に通知 */
```

```
9     data = io_in8(PORT_KEYDAT);
10    fifo8_put(&mousefifo, data);
11    return;
12 }
```

IRQ-12 是从 PIC 的第 4 号（从 PIC 相当于 IRQ-08 ～ IRQ-15），首先要通知 IRQ-12 受理已完成，然后再通知主 PIC。这是因为主/从 PIC 的协调不能够自动完成，如果程序不教给主 PIC 该怎么做，它就会忽视从 PIC 的下一个中断请求。从 PIC 连接到主 PIC 的第 2 号上，这么做 OK。

§

下面的鼠标数据取得方法，居然与键盘完全相同。靠中断号码来区分传到这个设备的数据究竟是来自键盘还是鼠标。

取得数据的程序如下：

```
1     fifo8_init(&mousefifo, 128, mousebuf);
2
3     for (;;) {
4         io_cli();
5         if (fifo8_status(&keyfifo) + fifo8_status(&mousefifo) == 0) {
6             io_stihlt();
7         } else {
8             if (fifo8_status(&keyfifo) != 0) {
```

```
9                   i = fifo8_get(&keyfifo);
10                  io_sti();
11                  sprintf(s, "%02X", i);
12                  boxfill8(binfo->vram, binfo->scrnx, COL8_008484,  0, 16, 15, 31);
13                  putfonts8_asc(binfo->vram, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
14              } else if (fifo8_status(&mousefifo) != 0) {
15                  i = fifo8_get(&mousefifo);
16                  io_sti();
17                  sprintf(s, "%02X", i);
18                  boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 32, 16, 47, 31);
19                  putfonts8_asc(binfo->vram, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
20              }
21          }
22      }
```

因为鼠标往往会比键盘更快地送出大量数据，所以我们将它的 FIFO 缓冲区增加到了 128 字节。这样，就算是一下子来了很多数据，也不会溢出。

取得数据的程序中，如果键盘和鼠标的 FIFO 缓冲区都为空了，就执行 HLT。如果不是两者都空，就先检查 keyinfo，如果有数据，就取出一个显示出来。如果 keyinfo 是空，就再去检查 mouseinfo，如果有数据，就取出一个显示出来。

程序运行正常。