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。

§

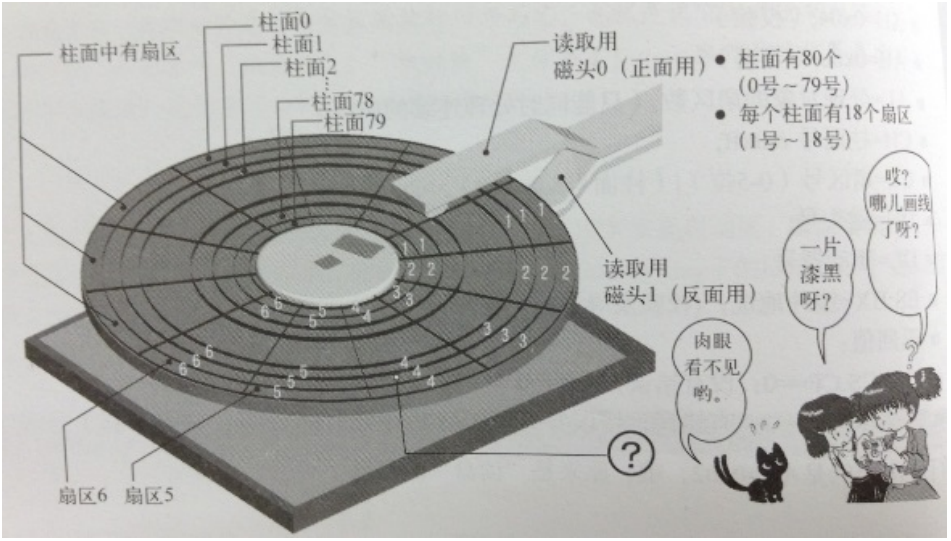

图 3.1: 软盘的结构

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：

```
1  struct KEYBUF {
2      unsigned char data[32];
3      int next;
4  };
5
6  void inthandler21(int *esp)
7  {
8      unsigned char data;
9      io_out8(PIC0_OCW2, 0x61);    /* IRQ-01 受付完了を PIC に通知 */
10     data = io_in8(PORT_KEYDAT);
11     if (keybuf.next < 32) {
12         keybuf.data[keybuf.next] = data;
13         keybuf.next++;
14     }
15     return;
16 }
```

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。

第 8 天　鼠标控制与 32 位模式切换

1　鼠标解读（1）（harib05a）

在上一章得到了鼠标的数据，现在要解读这些数据，然后结合鼠标的动作，让鼠标指针动起来。

```
1     unsigned char mouse_dbuf[3], mouse_phase;
2     enable_mouse();
3     mouse_phase = 0; /* マウスの 0xfa を待っている段階へ */
4
5     for (;;) {
6         io_cli();
7         if (fifo8_status(&keyfifo) + fifo8_status(&mousefifo) == 0) {
8             io_stihlt();
9         } else {
10            if (fifo8_status(&keyfifo) != 0) {
```

```
11              i = fifo8_get(&keyfifo);
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          } else if (fifo8_status(&mousefifo) != 0) {
17              i = fifo8_get(&mousefifo);
18              io_sti();
19              if (mouse_phase == 0) {
20                  /* マウスの 0xfa を待っている段階 */
21                  if (i == 0xfa) {
22                      mouse_phase = 1;
23                  }
24              } else if (mouse_phase == 1) {
25                  /* マウスの 1 バイト目を待っている段階 */
26                  mouse_dbuf[0] = i;
27                  mouse_phase = 2;
28              } else if (mouse_phase == 2) {
29                  /* マウスの 2 バイト目を待っている段階 */
30                  mouse_dbuf[1] = i;
31                  mouse_phase = 3;
32              } else if (mouse_phase == 3) {
```

```
33                      /* マウスの 3 バイト目を待っている段階 */
34                      mouse_dbuf[2] = i;
35                      mouse_phase = 1;
36                      /* データが 3 バイト揃ったので表示 */
37                      sprintf(s, "%02X %02X %02X", mouse_dbuf[0], mouse_dbuf[1], mouse_dbuf[2]);
38                      boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 32, 16, 32 + 8 * 8 - 1, 31);
39                      putfonts8_asc(binfo->vram, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
40                  }
41              }
42          }
43      }
```

首先要把最初读到的 0xfa 舍弃掉。之后，每次从鼠标那里送过来的数据都应该是 3 个字节一组的，所以每当数据累积到 3 个字节，就把它显示在屏幕上。

变量mouse_phase 用来记住接收鼠标数据的工作进展到了什么阶段（phase）。接收到的数据放在mouse_dbuf[0~2]内。

§

运行程序，点击鼠标或者滚动鼠标，可以看到各种反应。

鼠标的 3 个字节的数据分别表示点击（左击，右击，滚轮）数据，鼠标左右移动，鼠标上下移动。

2　稍事整理（harib05b）

修改后的 bootpack.c 节选

```
/* bootpack のメイン */

#include "bootpack.h"
#include <stdio.h>

struct MOUSE_DEC {
    unsigned char buf[3], phase;
};

void enable_mouse(struct MOUSE_DEC *mdec);
int mouse_decode(struct MOUSE_DEC *mdec, unsigned char dat);

void HariMain(void)
{
    （中略）
    struct MOUSE_DEC mdec;
    （中略）

    enable_mouse(&mdec);

```

```
21      for (;;) {
22          io_cli();
23          if (fifo8_status(&keyfifo) + fifo8_status(&mousefifo) == 0) {
24              io_stihlt();
25          } else {
26              if (fifo8_status(&keyfifo) != 0) {
27                  i = fifo8_get(&keyfifo);
28                  io_sti();
29                  sprintf(s, "%02X", i);
30                  boxfill8(binfo->vram, binfo->scrnx, COL8_008484,  0, 16, 15, 31);
31                  putfonts8_asc(binfo->vram, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
32              } else if (fifo8_status(&mousefifo) != 0) {
33                  i = fifo8_get(&mousefifo);
34                  io_sti();
35                  if (mouse_decode(&mdec, i) != 0) {
36                      /* データが 3 バイト揃ったので表示 */
37                      sprintf(s, "%02X %02X %02X", mdec.buf[0], mdec.buf[1], mdec.buf[2]);
38                      boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 32, 16, 32 + 8 * 8 - 1, 31);
39                      putfonts8_asc(binfo->vram, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
40                  }
41              }
42          }
```

```
43      }
44  }
45
46  void enable_mouse(struct MOUSE_DEC *mdec)
47  {
48      /* マウス有効 */
49      wait_KBC_sendready();
50      io_out8(PORT_KEYCMD, KEYCMD_SENDTO_MOUSE);
51      wait_KBC_sendready();
52      io_out8(PORT_KEYDAT, MOUSECMD_ENABLE);
53      /* うまくいくと ACK(0xfa) が送信されてくる */
54      mdec->phase = 0; /* マウスの 0xfa を待っている段階 */
55      return;
56  }
57
58  int mouse_decode(struct MOUSE_DEC *mdec, unsigned char dat)
59  {
60      if (mdec->phase == 0) {
61          /* マウスの 0xfa を待っている段階 */
62          if (dat == 0xfa) {
63              mdec->phase = 1;
64          }
```

```
65          return 0;
66      }
67      if (mdec->phase == 1) {
68          /* マウスの 1 バイト目を待っている段階 */
69          mdec->buf[0] = dat;
70          mdec->phase = 2;
71          return 0;
72      }
73      if (mdec->phase == 2) {
74          /* マウスの 2 バイト目を待っている段階 */
75          mdec->buf[1] = dat;
76          mdec->phase = 3;
77          return 0;
78      }
79      if (mdec->phase == 3) {
80          /* マウスの 3 バイト目を待っている段階 */
81          mdec->buf[2] = dat;
82          mdec->phase = 1;
83          return 1;
84      }
85      return -1; /* ここに来ることはないはず */
86  }
```

87

整理了程序。创建了一个结构体MOUSE_DEC，把解读鼠标所需要的变量都归总到一块儿。

在函数enable_mouse 的最后，添加了将 phase 归零的处理。之所以要舍去读到的 0xfa，是因为鼠标已经激活了。因此我们进行归零处理也不错。

将鼠标的解读从函数 HariMain 的接收信息处理中剥离出来，放到了mouse_decode 函数里，Harimain 又回到了清晰的状态。3 个字节凑齐后，mouse_decode 函数执行“return 1;”，把这些数据显示出来。

3　鼠标解读（2）（harib05c）

对mouse_decode 函数略加修改。

```
1 struct MOUSE_DEC {
2     unsigned char buf[3], phase;
3     int x, y, btn;
4 };
5
6 int mouse_decode(struct MOUSE_DEC *mdec, unsigned char dat)
7 {
8     if (mdec->phase == 0) {
9         /* マウスの 0xfa を待っている段階 */
10        if (dat == 0xfa) {
```

```
11              mdec->phase = 1;
12          }
13          return 0;
14      }
15      if (mdec->phase == 1) {
16          /* マウスの 1 バイト目を待っている段階 */
17          if ((dat & 0xc8) == 0x08) {
18              /* 正しい 1 バイト目だった */
19              mdec->buf[0] = dat;
20              mdec->phase = 2;
21          }
22          return 0;
23      }
24      if (mdec->phase == 2) {
25          /* マウスの 2 バイト目を待っている段階 */
26          mdec->buf[1] = dat;
27          mdec->phase = 3;
28          return 0;
29      }
30      if (mdec->phase == 3) {
31          /* マウスの 3 バイト目を待っている段階 */
32          mdec->buf[2] = dat;
```

```
33          mdec->phase = 1;
34          mdec->btn = mdec->buf[0] & 0x07;
35          mdec->x = mdec->buf[1];
36          mdec->y = mdec->buf[2];
37          if ((mdec->buf[0] & 0x10) != 0) {
38              mdec->x |= 0xffffff00;
39          }
40          if ((mdec->buf[0] & 0x20) != 0) {
41              mdec->y |= 0xffffff00;
42          }
43          mdec->y = - mdec->y; /* マウスでは y 方向の符号が画面と反対 */
44          return 1;
45      }
46      return -1; /* ここに来ることはないはず */
47  }
```

结构体里增加的几个变量用于存放解读结果。这几个变量是 x、y 和 btn，分别用于存放移动信息和鼠标按键状态。

还修改了 if（mdec—>phase==1）语句。这个 if 语句，用于判断第一字节对移动有反应的部分是否在 0 ～ 3 的范围内；同时还要判断第一字节对点击有反应的部分是否在 8 ～ F 的范围内。如果这个字节的数据不在以上范围内，它就会被舍去。

最后的 if（mdec—>phase==3）部分，是解读处理的核心。鼠标键的状态，放在buf[0] 的低 3 位，我们只取出这 3 位。十六进制的 0x07 相当于二进制的 0000 0111，因此通过与运算（&），可以很顺利地取出低 3 位的值。

x 和 y，基本上是直接使用buf[1] 和buf[2] ，但是需要使用第一字节中对鼠标移动有反应的几位（参考第一节的叙述）信息，将 x 和 y 的第 8 位及第 8 位以后全部都设成 1，或全部都保留为 0。这样就能正确地解读 x 和 y。

在解读处理的最后，对 y 的符号进行了取反的操作。这是因为，鼠标与屏幕的 y 方向正好相反，为了配合画面方向，就对 y 符号进行了取反操作。

4　移动鼠标指针（harib05d）

修改图形显示部分，让鼠标在屏幕上动起来。

HariMain 节选

```
1            } else if (fifo8_status(&mousefifo) != 0) {
2                i = fifo8_get(&mousefifo);
3                io_sti();
4                if (mouse_decode(&mdec, i) != 0) {
5                    /* データが 3 バイト揃ったので表示 */
6                    sprintf(s, "[lcr %4d %4d]", mdec.x, mdec.y);
7                    if ((mdec.btn & 0x01) != 0) {
8                        s[1] = 'L';
9                    }
```

```
10                  if ((mdec.btn & 0x02) != 0) {
11                      s[3] = 'R';
12                  }
13                  if ((mdec.btn & 0x04) != 0) {
14                      s[2] = 'C';
15                  }
16                  boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 32, 16, 32 + 15 * 8 - 1, 31);
17                  putfonts8_asc(binfo->vram, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
18                  /* マウスカーソルの移動 */
19                  boxfill8(binfo->vram, binfo->scrnx, COL8_008484, mx, my, mx + 15, my + 15); /* マウス消す */
20                  mx += mdec.x;
21                  my += mdec.y;
22                  if (mx < 0) {
23                      mx = 0;
24                  }
25                  if (my < 0) {
26                      my = 0;
27                  }
28                  if (mx > binfo->scrnx - 16) {
29                      mx = binfo->scrnx - 16;
30                  }
31                  if (my > binfo->scrny - 16) {
```

```
32                          my = binfo->scrny - 16;
33                      }
34                      sprintf(s, "(%3d, %3d)", mx, my);
35                      boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 0, 0, 79, 15); /* 座標消す */
36                      putfonts8_asc(binfo->vram, binfo->scrnx, 0, 0, COL8_FFFFFF, s); /* 座標書く */
37                      putblock8_8(binfo->vram, binfo->scrnx, 16, 16, mx, my, mcursor, 16); /* マウス描く */
38                  }
39              }
```

程序中会检查 mdec.btn 的值，用 3 个 if 语句将 s 的值置换成相应的字符串。

隐藏掉鼠标指针，然后在鼠标指针的坐标上，加上解读得到的位移量。“mx += mdec.x;”是“mx = mx + mdec.x;”的省略形式。因为不能让鼠标指针跑到屏幕外面去，所以进行了调整，调整后重新显示鼠标坐标，鼠标指针也会重新描画。

此时鼠标已经可以正常动起来了。不过只要鼠标一接触到装饰在屏幕下部的任务栏，就会有擦除效果。这是因为我们没有考虑到叠加处理，所以画面就出问题了。

5　通往 32 位模式之路

本节说明 asmhead.nas 中的如同谜一样的大约 100 行程序。

在没有说明的这段程序中，最开始做的事情如下：

asmhead.nas 节选

```
1 ; PIC が一切の割り込みを受け付けないようにする
2 ;    AT 互換機の仕様では、PIC の初期化をするなら、
```

```
;    こいつを CLI 前にやっておかないと、たまにハングアップする
;    PIC の初期化はあとでやる

        MOV     AL,0xff
        OUT     0x21,AL
        NOP                     ; OUT 命令を連続させるとうまくいかない機種があるらしいので
        OUT     0xa1,AL

        CLI                     ; さらに CPU レベルでも割り込み禁止
```

这段程序等同于以下内容的 C 程序。

```
io_out(PIC0_IMR, 0xff); /* 禁止主 PIC 的全部中断 */
io_out(PIC1_IMR, 0xff); /* 禁止从 PIC 的全部中断 */
io_cli(); /* 禁止 CPU 级别的中断 */
```

如果当 CPU 进行模式转换时进来了中断信号，那可就麻烦了。而且，后来还要进行 PIC 的初始化，初始化时也不允许有中断发生。所以，要把中断全部屏蔽掉。

§

asmhead.nas 节选（续）

```
; CPU から 1MB 以上のメモリにアクセスできるように、A20GATE を設定

```

```
3         CALL    waitkbdout
4         MOV       AL,0xd1
5         OUT       0x64,AL
6         CALL    waitkbdout
7         MOV       AL,0xdf             ; enable A20
8         OUT       0x60,AL
9         CALL    waitkbdout
```

这里的 waitkbdout，等同于wait_KBC_sendready。这段程序在 C 语言里的写法大致如下：

```
1 #define KEYCMD_WRITE_OUTPORT    0xd1
2 #define KBC_OUTPORT_A20G_ENABLE 0xdf
3
4     /* A20GATE 的设定 */
5     wait_KBC_sendready();
6     io_out8(PORT_KEYCMD, KEYCMD_WRITE_OUTPORT);
7     wait_KBC_sendready();
8     io_out8(PORT_KEYDAT, KBC_OUTPORT_A20G_ENABLE);
9     wait_KBC_sendready(); /* 这句话是为了等待完成执行指令 */
```

程序的基本结构与init_keyboard 完全相同，功能仅仅是往键盘控制电路发送指令。

这里发送的指令，是指令键盘控制电路的附属端口输出 0xdf。这个附属端口，连接着主板上的很多地方，通过这个端口发送不同的指令，就可以实现各种各样的控制功能。

这次输出 0xdf 所要完成的功能，是让 A20GATE 信号线变成 ON 的状态。它能使内存的 1MB 以上的部分变成可使用状态。最初出现电脑的时候，CPU 只有 16 位模式，所以内存最大也只有 1MB。后来 CPU 变聪明了，可以使用很大的内存了。但为了兼容旧版的操作系统，在执行激活指令之前，电路被限制为只能使用 1MB 内存。A20GATE 信号线正是用来使这个电路停止从而让所有内存都可以使用的东西。

最后还有一点，“wait_KBC_sendready();”是多余的。在此之后，虽然不会往键盘送命令，但仍然要等到下一个命令能够送来为止。这是为了等待 A20GATE 的处理切实完成。

§

asmhead.nas 节选（续）

```
; プロテクトモード移行

[INSTRSET "i486p"]                ; 486 の命令まで使いたいという記述

        LGDT    [GDTR0]           ; 暫定 GDT を設定
        MOV     EAX,CR0
        AND     EAX,0x7fffffff    ; bit31 を 0 にする（ページング禁止のため）
        OR      EAX,0x00000001    ; bit0 を 1 にする（プロテクトモード移行のため）
        MOV     CR0,EAX
        JMP     pipelineflush
pipelineflush:
        MOV     AX,1*8            ;  読み書き可能セグメント 32bit
        MOV     DS,AX
        MOV     ES,AX
```

```
15          MOV         FS,AX
16          MOV         GS,AX
17          MOV         SS,AX
```

INSTRSET 指令，是为了能够使用 386 以后的 LGDT，EAX，CR0 等关键字。

LGDT 指令，不管三七二十一，把随意准备的 GDT 给读进来。对于这个暂定的 GDT，以后还要重新设置。然后将 CR0 这一特殊的 32 位寄存器的值代入 EAX，并将最高位置为 0，最低位置为 1，再将这个值返回给 CR0 寄存器。这样就完成了模式转换，进入到不用分页的保护模式。CR0，也就是 control register 0，是一个非常重要的寄存器，只有操作系统才能操作它。

保护模式与先前的 16 位模式不同，段寄存器的解释不是 16 倍，而是能够使用 GDT。这里的“保护”，来自英文的“protect”。在这种模式下，应用程序既不能随便改变段的设定，又不能使用操作系统专用的段。操作系统受到 CPU 的保护，所以称为保护模式。

在保护模式中，有带保护的 16 位模式，和带保护的 32 位模式两种。我们要使用的，是带保护的 32 位模式。

讲解 CPU 的书上会写到，通过代入 CR0 而切换到保护模式时，要马上执行 JMP 指令。所以我们也执行这一指令。为什么要执行 JMP 指令呢？因为变成保护模式后，机器语言的解释要发生变化。CPU 为了加快指令的执行速度而使用了管道这一机制，就是说，前一条指令还在执行的时候，就开始解释下一条甚至是再下一条指令。因为模式变了，就要重新解释一遍，所以加入了 JMP 指令。

而且在程序中，进入保护模式以后，段寄存器的意思也变了（不再是乘以 16 后再加算的意思了），除了 CS 以外所有段寄存器的值都从 0x0000 变成了 0x0008。CS 保持原状是因为如果 CS 也变了，会造成混乱，所以只有 CS 要放到后面再处理。0x0008，相当于“gdt + 1”的段。

§

asmhead.nas 节选（续）

```
; bootpack の転送

        MOV     ESI,bootpack    ; 転送元
        MOV     EDI,BOTPAK      ; 転送先
        MOV     ECX,512*1024/4
        CALL    memcpy

; ついでにディスクデータも本来の位置へ転送

; まずはブートセクタから

        MOV     ESI,0x7c00      ; 転送元
        MOV     EDI,DSKCAC      ; 転送先
        MOV     ECX,512/4
        CALL    memcpy

; 残り全部

        MOV     ESI,DSKCAC0+512    ; 転送元
        MOV     EDI,DSKCAC+512     ; 転送先
        MOV     ECX,0
```

```
22          MOV         CL,BYTE [CYLS]
23          IMUL    ECX,512*18*2/4    ; シリンダ数からバイト数/4 に変換
24          SUB         ECX,512/4         ; IPL の分だけ差し引く
25          CALL    memcpy
```

简单来说，这部分程序只是在调用 memcpy 函数。我们将这段程序写成了 C 语言形式。

```
1 memcpy(bootpack,    BOTPAK,     512*1024/4);
2 memcpy(0x7c00,      DSKCAC,     512/4     );
3 memcpy(DSKCAC0+512, DSKCAC+512, cyls * 512*18*2/4 - 512/4);
```

函数 memcpy 是复制内存的函数，语法如下：

memcpy(转送源地址, 转送目的地址, 转送数据的大小);

转送数据大小是以双字为单位的，所以数据大小用字节数除以 4 来指定。在上面 3 个 memcpy 语句中，我们先来看看中间一句。

memcpy(0x7c00, DSKCAC, 512/4);

DSKCAC 是 0x00100000，所以上面这句话的意思就是从 0x7c00 复制 512 字节到 0x00100000。这正好是将启动扇区复制到 1MB 以后的内存去的意思。下一个 memcpy 语句：

memcpy(DSKCAC0+512, DSKCAC+512, cyls * 512*18*2/4-512/4);

它的意思就是将始于 0x00008200 的磁盘内容，复制到 0x00100200 那里。

上文中“转送数据大小”的计算有点复杂，因为它是以柱面数来计算的，所以需要减去启动区的那一部分长度。这样始于 0x00100000 的内存部分，就与磁盘的内容相吻合了。

现在我们还没说明的函数就只有有程序开始处的 memcpy 了。bootpack 是 asmhead.nas 的最后一个标签。haribote.sys 是通过 asmhead.bin 和 bootpack.hrb 连接起来而生成的（可以通过 Makefile 确认），所以 asmhead 结束的地方，紧接着串连着 bootpack.hrb 最前面的部分。

memcpy(bootpack, BOTPAK, 512*1024/4);

从 bootpack 的地址开始的 512KB 内容复制到 0x00280000 号地址去。

这就是将 bootpack.hrb 复制到 0x00280000 号地址的处理。为什么是 512KB 呢？这是我们酌情考虑而决定的。内存多一些不会产生什么问题，所以这个长度要比 bootpack.hrb 的长度大出很多。

§

asmhead.nas 节选（续）

```
; asmhead でしなければいけないことは全部し終わったので、
;    あとは bootpack に任せる

; bootpack の起動

        MOV     EBX,BOTPAK
        MOV     ECX,[EBX+16]
        ADD     ECX,3               ; ECX += 3;
        SHR     ECX,2               ; ECX /= 4;
        JZ      skip              ; 転送するべきものがない
        MOV     ESI,[EBX+20]      ; 転送元
        ADD     ESI,EBX
```

```
13          MOV         EDI,[EBX+12]     ; 転送先
14          CALL    memcpy
15  skip:
16          MOV         ESP,[EBX+12]     ; スタック初期値
17          JMP         DWORD 2*8:0x0000001b
```

结果我们仍然只是在做 memcpy。它对 bootpack.hrb 的 header 进行解析，将执行所必需的数据传送过去。EBX 里代入的是 BOTPAK，所以值如下：

[EBX + 16].....bootpack.hrb 之后的第 16 号地址。值是 0x11a8

[EBX + 20].....bootpack.hrb 之后的第 20 号地址。值是 0x10c8

[EBX + 12].....bootpack.hrb 之后的第 12 号地址。值是 0x00310000

上面这些值，是我们通过二进制编辑器，打开 harib05d 的 bootpack.hrb 后确认的。这些值因 harib 的版本不同而有所变化。

SHR 指令是向右移位指令，相当于“ECX ≫=2;”，与除以 4 有着相同的效果。因为二进制的数右移 1 位，值就变成了 1/2；左移 1 位，值就变成了 2 倍。这可能不太容易理解。还是拿我们熟悉的十进制来思考一下吧。十进制的时候，向右移动 1 位，值就变成了 1/10（比如 120->12）；向左移动 1 位，值就变成了 10 倍（比如 3 -> 30）。二进制也是一样。所以，向右移动 2 位，正好与除以 4 有着同样的效果。

JZ 是条件跳转指令，根据前一个计算结果是否为 0 来决定是否跳转。在这里，根据 SHR 的结果，如果 ECX 变成了 0，就跳转到 skip 那里去。在 harib05d 里，ECX 没有变成 0，所以不跳转。

而最终这个 memcpy 到底用来做什么事情呢？它会将 bootpack.hrb 第 0x10c8 字节开始的 0x11a8 字节复制到 0x00310000 号地址去。必须要等到“纸娃娃系统”的应用程序讲完之后才能讲清楚，以后还会说明的。

最后将 0x310000 代入到 ESP 里，然后用一个特别的 JMP 指令，将 2×8 代入到 CS 里，同时移动到 0x1b

号地址。这里的 0x1b 号地址是指第 2 个段的 0x1b 号地址。第 2 个段的基地址是 0x280000，所以实际上是从 0x28001b 开始执行的。这也就是 bootpack.hrb 的 0x1b 号地址。

这样就开始执行 bootpack.hrb 了。

§

下面介绍一下“纸娃娃系统”的内存分布图。

0x00000000 - 0x000fffff : 虽然在启动中会多次使用，但之后就变空。（1MB）

0x00100000 - 0x00267fff : 用于保存软盘的内容。（1440KB）

0x00268000 - 0x0026f7ff : 空（30KB）

0x0026f800 - 0x0026ffff : IDT （2KB）

0x00270000 - 0x0027ffff : GDT （64KB）

0x00280000 - 0x002fffff : bootpack.hrb（512KB）

0x00300000 - 0x003fffff : 栈及其他（1MB）

0x00400000 - : 空

这个内存分布图其实也没有什么特别的理由，觉得这样还行，跟着感觉走就决定了。另外，虽然没有明写，但在最初的 1MB 范围内，还有 BIOS，VRAM 等内容，也就是说并不是 1MB 全都空着。

从软盘读出来的东西，之所以要复制到 0x00100000 号以后的地址，就是因为我们意识中有这个内存分布图。同样，前几天，之所以能够确定正式版的 GDT 和 IDT 的地址，也是因为这个内存分布图。

§

asmhead.nas 节选（续）

```
1 waitkbdout:
2         IN          AL,0x64
```

```
3         AND         AL,0x02
4         IN        AL,0x60         ; 空读（为了清空数据接收缓冲区中的垃圾数据）
5         JNZ         waitkbdout        ; AND の結果が 0 でなければ waitkbdout へ
6         RET
```

这就是 waitkbdout 所完成的处理。基本上，如前面所说的那样，它与wait_KBC_sendready 相同，但也添加了部分处理，就是从 OX60 号设备进行 IN 的处理。也就是说，如果控制器里有键盘代码，或者是已经累积了鼠标数据，就顺便把它们读取出来。

§

下面是 memcpy 程序。

asmhead.nas 节选（续）

```
1 memcpy:
2         MOV         EAX,[ESI]
3         ADD         ESI,4
4         MOV         [EDI],EAX
5         ADD         EDI,4
6         SUB         ECX,1
7         JNZ         memcpy            ; 引き算した結果が 0 でなければ memcpy へ
8         RET
```

这是复制内存的程序。

§

asmhead.nas 节选（续）

```
        ALIGNB    16
GDT0:
        RESB    8                ; ヌルセレクタ
        DW        0xffff,0x0000,0x9200,0x00cf    ; 読み書き可能セグメント 32bit
        DW        0xffff,0x0000,0x9a28,0x0047    ; 実行可能セグメント 32bit （bootpack 用）

        DW        0
GDTR0:
        DW        8*3-1
        DD        GDT0

        ALIGNB    16
bootpack:
```

ALIGNB 指令的意思是，一直添加 DBO，直到时机合适的时候为止。什么是“时机合适”呢？大家可能有点不明白。ALIGNB 16 的情况下，地址能被 16 整除的时候，就称为“时机合适”。如果最初的地址能被 16 整除，则 ALIGNB 指令不作任何处理。

如果标签 GDT0 的地址不是 8 的整数倍，向段寄存器复制的 MOV 指令就会慢一些。所以插入了 ALIGNB 指令。但是如果这样，“ALIGNB 8”就够了，用“ALIGNB 16”有点过头了。最后的“bootpack:”之前，也是“时机合适”的状态，所以就适当加了一句“ALIGNB 16”。

GDT0 也是一种特定的 GDT。0 号是空区域（null sector），不能够在那里定义段。1 号和 2 号分别由下式设定。

`set_segmdesc(gdt + 1, 0xffffffff,   0x00000000, AR_DATA32_RW);`

`set_segmdesc(gdt + 2, LIMIT_BOTPAK, ADR_BOTPAK, AR_CODE32_ER);` 我们用纸笔事先计算了一下，然后用 DW 排列了出来。

GDTR0 是 LGDT 指令，意思是通知 GDT0 说“有了 GDT”。在 GDT0 里，写入了 16 位的段上限，和 32 位的段起始地址。

§

到此为止，关于 asmhead.nas 的说明就结束了。就是说，最初状态时，GDT 在 asmhead.nas 里，并不在 0x00270000　0x0027ffff 的范围里。IDT 连设定都没设定，所以仍处于中断禁止的状态。应当趁着硬件上积累过多数据而产生误动作之前，尽快开放中断，接收数据。

因此，在 bootpack.c 的 HariMain 里，应该在进行调色板（palette）的初始化以及画面的准备之前，先赶紧重新创建 GDT 和 IDT，初始化 PIC，并执行“`io_sti();`”。

bootpack.c 节选

```
1 void HariMain(void)
2 {
3     struct BOOTINFO *binfo = (struct BOOTINFO *) ADR_BOOTINFO;
4     char s[40], mcursor[256], keybuf[32], mousebuf[128];
5     int mx, my, i;
6     struct MOUSE_DEC mdec;
7
```

```
8     init_gdtidt();
9     init_pic();
10    io_sti(); /* IDT/PIC の初期化が終わったので CPU の割り込み禁止を解除 */
11    fifo8_init(&keyfifo, 32, keybuf);
12    fifo8_init(&mousefifo, 128, mousebuf);
13    io_out8(PIC0_IMR, 0xf9); /* PIC1 とキーボードを許可 (11111001) */
14    io_out8(PIC1_IMR, 0xef); /* マウスを許可 (11101111) */
15
16    init_keyboard();
17
18    init_palette();
19    init_screen8(binfo->vram, binfo->scrnx, binfo->scrny);
```

第 9 天　内存管理

1　整理源文件（harib06a）

鼠标指针的叠加处理留待以后处理，先转向内存管理。

把 bootpack.c 里面的一些函数单独择出去。

函数名	移动前	移动后
wait_KBC_sendready	bootpack.c	keyboard.c
init_keyboard	bootpack.c	keyboard.c
enable_mouse	bootpack.c	mouse.c
mouse_decode	bootpack.c	mouse.c
inthandler21	init.c	keyboard.c
inthandler2c	init.c	mouse.c

2　内存容量检查（1）（harib06b）

在进行内存管理之前，必须要做的事情是搞清楚内存究竟到底有多大，范围是到哪里。

在最初启动时，BIOS 肯定要检查内存容量，所以只要我们问一问 BIOS，就能知道内存容量有多大。但问题是，如果那样做的话，一方面 asmhead.nas 会变长，另一方面，BIOS 版本不同，BIOS 函数的调用方法也不相同，麻烦事太多了。所以，作者自己去检查内存。

内存检查时，要往内存里随便写入一个值，然后马上读取，来检查读取的值与写入的值是否相等。如果内存连接正常，则写入的值能够记在内存里。如果没连接上，则读出的值肯定是乱七八糟的。但是，如果 CPU 里加上了缓存会导致写入和读出的不是内存，而是缓存。结果，所有的内存都“正常”，检查处理不能完成。

所以，只有在内存检查时才将缓存设为 OFF。具体来说，就是先查查 CPU 是不是在 486 以上，如果是，就将缓存设为 OFF。按照这一思路，我们创建了以下函数 memtest。

bootpack.c 节选

```
1 #define EFLAGS_AC_BIT        0x00040000
2 #define CR0_CACHE_DISABLE    0x60000000
3
4 unsigned int memtest(unsigned int start, unsigned int end)
5 {
6     char flg486 = 0;
7     unsigned int eflg, cr0, i;
8
9     /* 386 か、486 以降なのかの確認 */
10    eflg = io_load_eflags();
```

```
11      eflg |= EFLAGS_AC_BIT; /* AC-bit = 1 */
12      io_store_eflags(eflg);
13      eflg = io_load_eflags();
14      if ((eflg & EFLAGS_AC_BIT) != 0) { /* 386 では AC=1 にしても自動で 0 に戻ってしまう */
15          flg486 = 1;
16      }
17      eflg &= ~EFLAGS_AC_BIT; /* AC-bit = 0 */
18      io_store_eflags(eflg);
19
20      if (flg486 != 0) {
21          cr0 = load_cr0();
22          cr0 |= CR0_CACHE_DISABLE; /* キャッシュ禁止 */
23          store_cr0(cr0);
24      }
25
26      i = memtest_sub(start, end);
27
28      if (flg486 != 0) {
29          cr0 = load_cr0();
30          cr0 &= ~CR0_CACHE_DISABLE; /* キャッシュ許可 */
31          store_cr0(cr0);
32      }
```

```
33
34      return i;
35  }
```

最初对 EFLAGS 进行的处理，是检查 CPU 是 486 以上还是 386。如果是 486 以上，EFLAGS 寄存器的第 18 位应该是所谓的 AC 标志位；如果 CPU 是 386，那么就没有这个标志位，第 18 位一直是 0。这里，我们有意识地把 1 写入到这一位，然后再读出 EFLAGS 的值，继而检查 AC 标志位是否仍为 1。最后，将 AC 标志位重置为 0。

将 AC 标志位重置为 0 时，用到了 AND 运算，那里出现了一个运算符“ ”，它是取反运算符，就是将所有的位都反转的意思。所以，~EFLAGS_AC_BIT 与 0xfffbffff 一样。

为了禁止缓存，需要对 CR0 寄存器的某一标志位进行操作，需要用到函数load_cr0 和store_cr0，这个函数存在 naskfunc.nas 里。

naskfunc.nas 节选

```
1  _load_cr0:        ; int load_cr0(void);
2          MOV       EAX,CR0
3          RET
4
5  _store_cr0:       ; void store_cr0(int cr0);
6          MOV       EAX,[ESP+4]
7          MOV       CR0,EAX
8          RET
```

§

memtest_sub 函数，是内存检查处理的实现部分。

```
1  unsigned int memtest_sub(unsigned int start, unsigned int end)
2  {
3      unsigned int i, *p, old, pat0 = 0xaa55aa55, pat1 = 0x55aa55aa;
4      for (i = start; i <= end; i += 0x1000) {
5          p = (unsigned int *) (i + 0xffc);
6          old = *p;            /* いじる前の値を覚えておく */
7          *p = pat0;            /* ためしに書いてみる */
8          *p ^= 0xffffffff;    /* そしてそれを反転してみる */
9          if (*p != pat1) {    /* 反転結果になったか？ */
10 not_memory:
11             *p = old;
12             break;
13         }
14         *p ^= 0xffffffff;    /* もう一度反転してみる */
15         if (*p != pat0) {    /* 元に戻ったか？ */
16             goto not_memory;
17         }
18         *p = old;            /* いじった値を元に戻す */
19     }
20     return i;
```

```
21 }
```

调查从 start 地址到 end 地址的范围内，能够使用的内存的末尾地址。首先如果 p 不是指针，就不能指定地址去读取内存，所以先执行“p=i;”。紧接着使用这个 p，将原值保存下来（变量 old）。接着试写 0xaa55aa55，在内存里反转该值，检查结果是否正确。如果正确，就再次反转它，检查一下是否能回复到初始值。最后，使用 old 变量，将内存的值恢复回去。如果在某个环节没能恢复成预想的值，那么就在那个环节终止调查，并报告终止时的地址。

for 语句中 i 的增值部分以及 p 的赋值部分。每次只增加 4，就要检查全部内存，速度太慢了，所以改成了每次增加 0x1000，相当于 4KB，这样一来速度就提高了 1000 倍。p 的赋值计算式也变了，这是因为，如果不进行任何改变仍写作“p=i;”的话，程序就会只检查 4KB 最开头的 4 个字节。所以要改为“p=i + 0xffc;”，让它只检查末尾的 4 个字节。

§

修改 HariMain 程序，添加以下部分：

bootpack.c 节选

```
1    i = memtest(0x00400000, 0xbfffffff) / (1024 * 1024);
2    sprintf(s, "memory %dMB", i);
3    putfonts8_asc(binfo->vram, binfo->scrnx, 0, 32, COL8_FFFFFF, s);
```

暂时先使用以上程序对 0x00400000 ～ 0xbfffffff 范围的内存进行检查。这个程序最大可以识别 3GB 范围的内存。0x00400000 号以前的内存已经被使用了（参考 8.5 节的内存分布图），没有内存，程序根本运行不到这里，所以我们没做内存检查。以 MB 为单位。

如果在 QEMU 上运行，根据模拟器的设定，内存应该为 32MB。不过运行结果显示的是 3072MB。有错误。

3　内存容量检查（2）（harib06c）

真正的原因是编译器对程序进行了优化，导致写的程序中有一些直接被「优化」掉了。

使用“make -r bootpack.nas”来运行的话，就可以确认 bootpack.c 被编译成了什么样的机器语言。用文本编辑器看一看生成的 bootpack.nas 会发现，最下边有memtest_sub 的编译结果。

memtest_sub 的编译结果

```
_memtest_sub:
    PUSH    EBP                 ; C 编译器的固定语句
    MOV EBP,ESP
    MOV EDX,DWORD [12+EBP]      ; EDX = end;
    MOV EAX,DWORD [8+EBP]       ; EAX = start; /* EAX 是 i */
    CMP EAX,EDX                 ; if (EAX > EDX) goto L30;
    JA  L30
L36:
L34:
    ADD EAX,4096                ; EAX += 0x1000;
    CMP EAX,EDX                 ; if (EAX <= EDX) goto L36;
    JBE L36
L30:
    POP EBP                     ; 接收前文中 PUSH 的 EBP
    RET                         ; return;
```

与原始程序相比会发现，编译后没有 XOR 等指令，而且，好像编译后只剩下了 for 语句。

§

编译器优化过程：

首先将内存的内容保存到 old 里，然后写入 pat0 的值，再反转，最后跟 pat1 进行比较。肯定相等，if 语句不成立，得不到执行，所以把它删掉，将比较 *p 和 pat0 这部分也删掉。

```
1  unsigned int memtest_sub(unsigned int start, unsigned int end)
2  {
3      unsigned int i, *p, old, pat0 = 0xaa55aa55, pat1 = 0x55aa55aa;
4      for (i = start; i <= end; i += 0x1000) {
5          p = (unsigned int *) (i + 0xffc);
6
7          old = *p;           /* 先记住修改前的值 */
8          *p = pat0;          /* 试写 */
9          *p ^= 0xffffffff;   /* 反转 */
10         *p ^= 0xffffffff;   /* 再次反转 */
11         *p = old;           /* 恢复为修改前的值 */
12     }
13     return i;
14 }
```

反转了两次会变回之前的状态，所以这些处理也可以不要。

```
1 unsigned int memtest_sub(unsigned int start, unsigned int end)
2   {
3       unsigned int i, *p, old, pat0 = 0xaa55aa55, pat1 = 0x55aa55aa;
4       for (i = start; i <= end; i += 0x1000) {
5           p = (unsigned int *) (i + 0xffc);
6           old = *p;           /* 先记住修改前的值 */
7           *p = pat0;          /* 试写 */
8           *p = old;           /* 恢复为修改前的值 */
9       }
10      return i;
11  }
```

还有，“*p = pat0;”本来就没有意义。反正要将 old 的值赋给 *p。

```
1 unsigned int memtest_sub(unsigned int start, unsigned int end)
2   {
3       unsigned int i, *p, old, pat0 = 0xaa55aa55, pat1 = 0x55aa55aa;
4       for (i = start; i <= end; i += 0x1000) {
5           p = (unsigned int *) (i + 0xffc);
6           old = *p;           /* 先记住修改前的值 */
7           *p = old;           /* 恢复为修改前的值 */
8       }
```

```
9       return i;
10  }
```

*p 里面实际没写进任何内容，删。

```
1 unsigned int memtest_sub(unsigned int start, unsigned int end)
2   {
3       unsigned int i, *p, old, pat0 = 0xaa55aa55, pat1 = 0x55aa55aa;
4       for (i = start; i <= end; i += 0x1000) {
5           p = (unsigned int *) (i + 0xffc);
6       }
7       return i;
8   }
```

这里的地址变量 p，虽然计算了地址，却一次也没有用到，old、pat0、pat1 也都是用不到的变量。全部都舍弃掉。

```
1  unsigned int memtest_sub(unsigned int start, unsigned int end)
2   {
3       unsigned int i;
4       for (i = start; i <= end; i += 0x1000) { }
5       return i;
6   }
```

根据以上编译器的思路，我们可以看出，它进行了最优化处理。但其实这个工作本来是不需要的。用于应用程序的 C 编译器，根本想不到会对没有内存的地方进行读写。

如果更改编译选项，是可以停止最优化处理的。可是在其他地方，我们还是需要如此考虑周密的最优化处理的，所以不想更改编译选项。于是决定memtest_sub 也用汇编来写。

naskfunc.nas 节选

```
1  _memtest_sub:   ; unsigned int memtest_sub(unsigned int start, unsigned int end)
2          PUSH    EDI                         ; （EBX, ESI, EDI も使いたいので）
3          PUSH    ESI
4          PUSH    EBX
5          MOV     ESI,0xaa55aa55          ; pat0 = 0xaa55aa55;
6          MOV     EDI,0x55aa55aa          ; pat1 = 0x55aa55aa;
7          MOV     EAX,[ESP+12+4]          ; i = start;
8  mts_loop:
9          MOV     EBX,EAX
10         ADD     EBX,0xffc               ; p = i + 0xffc;
11         MOV     EDX,[EBX]               ; old = *p;
12         MOV     [EBX],ESI               ; *p = pat0;
13         XOR     DWORD [EBX],0xffffffff  ; *p ^= 0xffffffff;
14         CMP     EDI,[EBX]               ; if (*p != pat1) goto fin;
15         JNE     mts_fin
16         XOR     DWORD [EBX],0xffffffff  ; *p ^= 0xffffffff;
17         CMP     ESI,[EBX]               ; if (*p != pat0) goto fin;
```

```
18          JNE     mts_fin
19          MOV     [EBX],EDX               ; *p = old;
20          ADD     EAX,0x1000              ; i += 0x1000;
21          CMP     EAX,[ESP+12+8]          ; if (i <= end) goto mts_loop;
22
23        JBE     mts_loop
24          POP     EBX
25          POP     ESI
26          POP     EDI
27          RET
28 mts_fin:
29          MOV     [EBX],EDX               ; *p = old;
30          POP     EBX
31          POP     ESI
32          POP     EDI
33          RET
```

删除 bootpack.c 中的`memtest_sub` 函数，运行程序。内存容量显示正常。

4　挑战内存管理（harib06d）

内存管理的基础，一是内存分配，一是内存释放。[1]

内存管理程序：

bootpack.c 节选

```
#define MEMMAN_FREES        4090    /* 偙傟偱栺 32KB */

struct FREEINFO {    /* あき情報 */
    unsigned int addr, size;
};

struct MEMMAN {        /* メモリ管理 */
    int frees, maxfrees, lostsize, losts;
    struct FREEINFO free[MEMMAN_FREES];
};

void memman_init(struct MEMMAN *man)
{
    man->frees = 0;            /* あき情報の個数 */
    man->maxfrees = 0;        /* 状況観察用：frees の最大値 */
    man->lostsize = 0;        /* 解放に失敗した合計サイズ */
```

[1]建议找本操作系统的书看看各种内存管理方法。这里不想写了。

```
17      man->losts = 0;             /* 解放に失敗した回数 */
18      return;
19  }
20
21  unsigned int memman_total(struct MEMMAN *man)
22  /* あきサイズの合計を報告 */
23  {
24      unsigned int i, t = 0;
25      for (i = 0; i < man->frees; i++) {
26          t += man->free[i].size;
27      }
28      return t;
29  }
30
31  unsigned int memman_alloc(struct MEMMAN *man, unsigned int size)
32  /* 確保 */
33  {
34      unsigned int i, a;
35      for (i = 0; i < man->frees; i++) {
36          if (man->free[i].size >= size) {
37              /* 十分な広さのあきを発見 */
38              a = man->free[i].addr;
```

```
39              man->free[i].addr += size;
40              man->free[i].size -= size;
41              if (man->free[i].size == 0) {
42                  /* free[i] がなくなったので前へつめる */
43                  man->frees--;
44                  for (; i < man->frees; i++) {
45                      man->free[i] = man->free[i + 1]; /* 構造体の代入 */
46                  }
47              }
48              return a;
49          }
50      }
51      return 0; /* あきがない */
52  }
```

一开始的 struct MEMMAN，创建了 4000 组，留出不少余量，管理空间大约是 32KB。其中还有变量 maxfrees、lostsize、losts 等，这些变量与管理本身没有关系，不用在意它们。如果特别想了解的话，可以看看函数memman_init 的注释，里面有介绍。

函数memman_init 对 memman 进行了初始化，设定为空。主要工作，是将 frees 设为 0，而其他的都是附属性设定。

函数memman_total 用来计算可用内存的合计大小并返回。

最后的memman_alloc 函数，功能是分配指定大小的内存。除了free[i].size 变为 0 时的处理以外的部分，

在前面已经说过了。

memman_alloc 函数中free[i].size 等于 0 的处理，与 FIFO 缓冲区的处理方法很相似，要进行移位处理。希望大家注意以下写法：

```
1 man->free[i].addr = man->free[i+1].addr;
2
3 man->free[i].size = man->free[i+1].size;
```

我们在这里将其归纳为了：

```
1 man->free[i] = man->free[i+1];
```

这种方法被称为结构体赋值，其使用方法如上所示，可以写成简单的形式。

§

释放内存函数，也就是往 memman 里追加可用内存信息的函数，稍微有点复杂。

bootpack.c 节选

```
1 int memman_free(struct MEMMAN *man, unsigned int addr, unsigned int size)
2 /* 解放 */
3 {
4     int i, j;
5     /* まとめやすさを考えると、free[] が addr 順に並んでいるほうがいい */
```

```
6       /* だからまず、どこに入れるべきかを決める */
7       for (i = 0; i < man->frees; i++) {
8           if (man->free[i].addr > addr) {
9               break;
10          }
11      }
12      /* free[i - 1].addr < addr < free[i].addr */
13      if (i > 0) {
14          /* 前がある */
15          if (man->free[i - 1].addr + man->free[i - 1].size == addr) {
16              /* 前のあき領域にまとめられる */
17              man->free[i - 1].size += size;
18              if (i < man->frees) {
19                  /* 後ろもある */
20                  if (addr + size == man->free[i].addr) {
21                      /* なんと後ろともまとめられる */
22                      man->free[i - 1].size += man->free[i].size;
23                      /* man->free[i] の削除 */
24                      /* free[i] がなくなったので前へつめる */
25                      man->frees--;
26                      for (; i < man->frees; i++) {
27                          man->free[i] = man->free[i + 1]; /* 構造体の代入 */
```

```
28                      }
29                  }
30              }
31              return 0; /* 成功終了 */
32          }
33      }
34      /* 前とはまとめられなかった */
35      if (i < man->frees) {
36          /* 後ろがある */
37          if (addr + size == man->free[i].addr) {
38              /* 後ろとはまとめられる */
39              man->free[i].addr = addr;
40              man->free[i].size += size;
41              return 0; /* 成功終了 */
42          }
43      }
44      /* 前にも後ろにもまとめられない */
45      if (man->frees < MEMMAN_FREES) {
46          /* free[i] より後ろを、後ろへずらして、すきまを作る */
47          for (j = man->frees; j > i; j--) {
48              man->free[j] = man->free[j - 1];
49          }
```

```
50          man->frees++;
51          if (man->maxfrees < man->frees) {
52              man->maxfrees = man->frees; /* 最大値を更新 */
53          }
54          man->free[i].addr = addr;
55          man->free[i].size = size;
56          return 0; /* 成功終了 */
57      }
58      /* 後ろにずらせなかった */
59      man->losts++;
60      man->lostsize += size;
61      return -1; /* 失敗終了 */
62  }
```

如果可用信息表满了，就按照舍去之后带来损失最小的原则进行割舍。但是在这个程序里，我们并没有对损失程度进行比较，而是舍去了刚刚进来的可用信息，这只是为了图个方便。

§

最后，将这个程序应用于 HariMain，结果就变成了下面这样。

bootpack.c 节选

```
1 #define MEMMAN_ADDR         0x003c0000
2
```

```
3 void HariMain(void)
4 {
5     （中略）
6     unsigned int memtotal;
7     struct MEMMAN *memman = (struct MEMMAN *) MEMMAN_ADDR;
8     （中略）
9     memtotal = memtest(0x00400000, 0xbfffffff);
10    memman_init(memman);
11    memman_free(memman, 0x00001000, 0x0009e000); /* 0x00001000 - 0x0009efff */
12    memman_free(memman, 0x00400000, memtotal - 0x00400000);
13    （中略）
14    sprintf(s, "memory %dMB   free : %dKB",
15            memtotal / (1024 * 1024), memman_total(memman) / 1024);
16    putfonts8_asc(binfo->vram, binfo->scrnx, 0, 32, COL8_FFFFFF, s);
```

memman 需要 32KB，我们暂时决定使用自 0x003c0000 开始的 32KB（0x00300000 号地址以后，今后的程序即使有所增加，预计也不会到达 0x003c0000，所以我们使用这一数值），然后计算内存总量 memtotal，将现在不用的内存以 0x1000 个字节为单位注册到 memman 里。最后，显示出合计可用内存容量。在 QEMU 上执行时，有时会注册成 632KB 和 28MB。$632 + 28672 = 29304$，所以屏幕上会显示出 29304KB。

第 10 天　叠加处理

1　内存管理（续）（harib07a）

为了以后使用起来更加方便，我们还是把这些内存管理函数再整理一下。memman_alloc 和memman_free 能够以 1 字节为单位进行内存管理，这种方式虽然不错，但是有一点不足——在反复进行内存分配和内存释放之后，内存中就会出现很多不连续的小段未使用空间，这样就会把man—>frees 消耗殆尽。

编写一些总是以 0x1000 字节为单位进行内存分配和释放的函数，它们会把指定的内存大小按 0x1000 字节为单位向上舍入（roundup），0x1000 字节的大小正好是 4KB。

memory.c 节选

```
1 unsigned int memman_alloc_4k(struct MEMMAN *man, unsigned int size)
2 {
3     unsigned int a;
4     size = (size + 0xfff) & 0xfffff000;
5     a = memman_alloc(man, size);
6     return a;
```

```
}

int memman_free_4k(struct MEMMAN *man, unsigned int addr, unsigned int size)
{
    int i;
    size = (size + 0xfff) & 0xfffff000;
    i = memman_free(man, addr, size);
    return i;
}
```

增加的部分使用了向上舍入。`i = (i + 0xfff) & 0xfffff000;` 是“加上 0xfff 后进行向下舍入”的运算。

由于十六进制不易理解，所以以十进制运算为例来说明。如使用这个方法以 100 为单位对 456 进行向上舍入，就相当于先加上 99 再进行向下舍入。456 加上 99 是 555，向下舍入后就是 500 了。那么如果对 400 进行向上舍入呢？先加上 99，得到 499，再进行向下舍入，结果是 400。看，400 向上舍入的结果还是 400。

2　叠加处理（harib07b）

在画面上进行叠加显示，类似于将绘制了图案的透明图层叠加在一起。

实际上，我们并不是仅仅把两张大小相同的图层重叠在一起，而是要从大到小准备很多张图层。

最上面的小图层用来描绘鼠标指针，它下面的几张图层是用来存放窗口的，而最下面的一张图层用来存放桌面壁纸。同时，我们还要通过移动图层的方法实现鼠标指针的移动以及窗口的移动。

§

```
struct SHEET {
    unsigned char *buf;
    int bxsize, bysize, vx0, vy0, col_inv, height, flags;
}
```

暂时先写成这样就可以了。程序里的 sheet 这个词，表示“透明图层”的意思。笔者觉得英文里没有和“透明图层”接近的词，就凭感觉选了它。buf 是用来记录图层上所描画内容的地址（buffer 的略语）。图层的整体大小，用 bxsize*bysize 表示。vx0 和 vy0 是表示图层在画面上位置的坐标，v 是 VRAM 的略语。`col_inv` 表示透明色色号，它是 color （颜色）和 invisible（透明）的组合略语。height 表示图层高度。flags 用于存放有关图层的各种设定信息。

创建一个管理多重图层信息的结构。

```
#define MAX_SHEETS      256

struct SHTCTL {
    unsigned char *vram;
    int xsize, ysize, top;
    struct SHEET *sheets[MAX_SHEETS];
    struct SHEET sheets0[MAX_SHEETS];
};
```

创建了 SHTCTL 结构体，其名称来源于 sheet control 的略语，意思是“图层管理”。MAX_SHEETS 是能够管理的最大图层数，这个值设为 256 应该够用了。

变量 vram、xsize、ysize 代表 VRAM 的地址和画面的大小，但如果每次都从 BOOTINFO 查询的话就太麻烦了，所以在这里预先对它们进行赋值操作。top 代表最上面图层的高度。sheets0 这个结构体用于存放准备的 256 个图层的信息。而 sheets 是记忆地址变量的领域，所以相应地也要先准备 256 份。由于 sheets0 中的图层顺序混乱，所以把它们按照高度进行升序排列，然后将其地址写入 sheets 中，这样就方便多了。

§

```
1 struct SHTCTL *shtctl_init(struct MEMMAN *memman, unsigned char *vram, int xsize, int ysize)
2 {
3     struct SHTCTL *ctl;
4     int i;
5     ctl = (struct SHTCTL *) memman_alloc_4k(memman, sizeof (struct SHTCTL));
6     if (ctl == 0) {
7         goto err;
8     }
9     ctl->vram = vram;
10    ctl->xsize = xsize;
11    ctl->ysize = ysize;
12    ctl->top = -1; /* シートは一枚もない */
13    for (i = 0; i < MAX_SHEETS; i++) {
14        ctl->sheets0[i].flags = 0; /* 未使用マーク */
```

```
15      }
16  err:
17      return ctl;
18  }
```

程序首先使用memman_alloc_4k 来分配用于记忆图层控制变量的内存空间
接着，给控制变量赋值，给其下的所有图层变量都加上“未使用”标签。做完这一步，这个函数就完成了。

§

再做一个函数，用于取得新生成的未使用图层。

```
1  #define SHEET_USE        1
2
3  struct SHEET *sheet_alloc(struct SHTCTL *ctl)
4  {
5      struct SHEET *sht;
6      int i;
7      for (i = 0; i < MAX_SHEETS; i++) {
8          if (ctl->sheets0[i].flags == 0) {
9              sht = &ctl->sheets0[i];
10             sht->flags = SHEET_USE; /* 使用中マーク */
11             sht->height = -1; /* 非表示中 */
```

```
12              return sht;
13          }
14      }
15      return 0;     /* 全てのシートが使用中だった */
16  }
```

在sheets0[] 中寻找未使用的图层，如果找到了，就将其标记为“正在使用”，并返回其地址就可以了，这里没有什么难点。高度设为 -1，表示图层的高度还没有设置，因而不是显示对象。

§

```
1  void sheet_setbuf(struct SHEET *sht, unsigned char *buf, int xsize, int ysize, int col_inv)
2  {
3      sht->buf = buf;
4      sht->bxsize = xsize;
5      sht->bysize = ysize;
6      sht->col_inv = col_inv;
7      return;
8  }
```

这是设定图层的缓冲区大小和透明色的函数。

§

写设定底板高度的函数。

```
1 void sheet_updown(struct SHTCTL *ctl, struct SHEET *sht, int height)
2 {
3     int h, old = sht->height; /* 設定前の高さを記憶する */
4
5     /* 指定が低すぎや高すぎだったら、修正する */
6     if (height > ctl->top + 1) {
7         height = ctl->top + 1;
8     }
9     if (height < -1) {
10        height = -1;
11    }
12    sht->height = height; /* 高さを設定 */
13
14    /* 以下は主に sheets[] の並べ替え */
15    if (old > height) {    /* 以前よりも低くなる */
16        if (height >= 0) {
17            /* 間のものを引き上げる */
18            for (h = old; h > height; h--) {
19                ctl->sheets[h] = ctl->sheets[h - 1];
20                ctl->sheets[h]->height = h;
```

```
21              }
22              ctl->sheets[height] = sht;
23          } else {    /* 非表示化 */
24              if (ctl->top > old) {
25                  /* 上になっているものをおろす */
26                  for (h = old; h < ctl->top; h++) {
27                      ctl->sheets[h] = ctl->sheets[h + 1];
28                      ctl->sheets[h]->height = h;
29                  }
30              }
31              ctl->top--; /* 表示中の下じきが一つ減るので、一番上の高さが減る */
32          }
33          sheet_refresh(ctl); /* 新しい下じきの情報に沿って画面を描き直す */
34      } else if (old < height) {    /* 以前よりも高くなる */
35          if (old >= 0) {
36              /* 間のものを押し下げる */
37              for (h = old; h < height; h++) {
38                  ctl->sheets[h] = ctl->sheets[h + 1];
39                  ctl->sheets[h]->height = h;
40              }
41              ctl->sheets[height] = sht;
42          } else {    /* 非表示状態から表示状態へ */
```

```
43              /* 上になるものを持ち上げる */
44              for (h = ctl->top; h >= height; h--) {
45                  ctl->sheets[h + 1] = ctl->sheets[h];
46                  ctl->sheets[h + 1]->height = h + 1;
47              }
48              ctl->sheets[height] = sht;
49              ctl->top++; /* 表示中の下じきが一つ増えるので、一番上の高さが増える */
50          }
51          sheet_refresh(ctl); /* 新しい下じきの情報に沿って画面を描き直す */
52      }
53      return;
54  }
```

§

在sheet_updown 中使用了sheet_refresh 函数，这个函数会从下到上描绘所有的图层。

```
1  void sheet_refresh(struct SHTCTL *ctl)
2  {
3      int h, bx, by, vx, vy;
4      unsigned char *buf, c, *vram = ctl->vram;
5      struct SHEET *sht;
6      for (h = 0; h <= ctl->top; h++) {
```

```
7           sht = ctl->sheets[h];
8           buf = sht->buf;
9           for (by = 0; by < sht->bysize; by++) {
10              vy = sht->vy0 + by;
11              for (bx = 0; bx < sht->bxsize; bx++) {
12                  vx = sht->vx0 + bx;
13                  c = buf[by * sht->bxsize + bx];
14                  if (c != sht->col_inv) {
15                      vram[vy * ctl->xsize + vx] = c;
16                  }
17              }
18          }
19      }
20      return;
21  }
```

对于已设定了高度的所有图层而言，要从下往上，将透明以外的所有像素都复制到 VRAM 中。由于是从下开始复制，所以最后最上面的内容就留在了画面上。

§

不改变图层高度而只上下左右移动图层的函数——sheet_slide。

```
1 void sheet_slide(struct SHTCTL *ctl, struct SHEET *sht, int vx0, int vy0)
2 {
3     sht->vx0 = vx0;
4     sht->vy0 = vy0;
5     if (sht->height >= 0) { /* もしも表示中なら */
6         sheet_refresh(ctl); /* 新しい下じきの情報に沿って画面を描き直す */
7     }
8     return;
9 }
```

最后是释放已使用图层的内存的函数sheet_free。

```
1 void sheet_free(struct SHTCTL *ctl, struct SHEET *sht)
2 {
3     if (sht->height >= 0) {
4         sheet_updown(ctl, sht, -1); /* 表示中ならまず非表示にする */
5     }
6     sht->flags = 0; /* 未使用マーク */
7     return;
8 }
```

§

将以上与图层相关的程序汇总到 sheet.c 中，所以就要改造 HariMain 函数了。

```
1  void HariMain(void)
2  {
3      (中略)
4      struct SHTCTL *shtctl;
5      struct SHEET *sht_back, *sht_mouse;
6      unsigned char *buf_back, buf_mouse[256];
7
8      （中略）
9
10     init_palette();
11     shtctl = shtctl_init(memman, binfo->vram, binfo->scrnx, binfo->scrny);
12     sht_back  = sheet_alloc(shtctl);
13     sht_mouse = sheet_alloc(shtctl);
14     buf_back  = (unsigned char *) memman_alloc_4k(memman, binfo->scrnx * binfo->scrny);
15     sheet_setbuf(sht_back, buf_back, binfo->scrnx, binfo->scrny, -1); /* 透明色なし */
16     sheet_setbuf(sht_mouse, buf_mouse, 16, 16, 99);
17     init_screen8(buf_back, binfo->scrnx, binfo->scrny);
18     init_mouse_cursor8(buf_mouse, 99);
19     sheet_slide(shtctl, sht_back, 0, 0);
20     mx = (binfo->scrnx - 16) / 2; /* 画面中央になるように座標計算 */
```

```
21      my = (binfo->scrny - 28 - 16) / 2;
22      sheet_slide(shtctl, sht_mouse, mx, my);
23      sheet_updown(shtctl, sht_back,  0);
24      sheet_updown(shtctl, sht_mouse, 1);
25      sprintf(s, "(%3d, %3d)", mx, my);
26      putfonts8_asc(buf_back, binfo->scrnx, 0, 0, COL8_FFFFFF, s);
27      sprintf(s, "memory %dMB   free : %dKB",
28              memtotal / (1024 * 1024), memman_total(memman) / 1024);
29      putfonts8_asc(buf_back, binfo->scrnx, 0, 32, COL8_FFFFFF, s);
30      sheet_refresh(shtctl);
31
32      for (;;) {
33          io_cli();
34          if (fifo8_status(&keyfifo) + fifo8_status(&mousefifo) == 0) {
35              io_stihlt();
36          } else {
37              if (fifo8_status(&keyfifo) != 0) {
38                  i = fifo8_get(&keyfifo);
39                  io_sti();
40                  sprintf(s, "%02X", i);
41                  boxfill8(buf_back, binfo->scrnx, COL8_008484,  0, 16, 15, 31);
42                  putfonts8_asc(buf_back, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
```

```
43                  sheet_refresh(shtctl);
44              } else if (fifo8_status(&mousefifo) != 0) {
45                  i = fifo8_get(&mousefifo);
46                  io_sti();
47                  if (mouse_decode(&mdec, i) != 0) {
48                      /* データが 3 バイト揃ったので表示 */
49                      sprintf(s, "[lcr %4d %4d]", mdec.x, mdec.y);
50                      if ((mdec.btn & 0x01) != 0) {
51                          s[1] = 'L';
52                      }
53                      if ((mdec.btn & 0x02) != 0) {
54                          s[3] = 'R';
55                      }
56                      if ((mdec.btn & 0x04) != 0) {
57                          s[2] = 'C';
58                      }
59                      boxfill8(buf_back, binfo->scrnx, COL8_008484, 32, 16, 32 + 15 * 8 - 1, 31);
60                      putfonts8_asc(buf_back, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
61                      /* マウスカーソルの移動 */
62                      mx += mdec.x;
63                      my += mdec.y;
64                      if (mx < 0) {
```

```
65                          mx = 0;
66                      }
67                      if (my < 0) {
68                          my = 0;
69                      }
70                      if (mx > binfo->scrnx - 16) {
71                          mx = binfo->scrnx - 16;
72                      }
73                      if (my > binfo->scrny - 16) {
74                          my = binfo->scrny - 16;
75                      }
76                      sprintf(s, "(%3d, %3d)", mx, my);
77                      boxfill8(buf_back, binfo->scrnx, COL8_008484, 0, 0, 79, 15); /* 座標消す */
78                      putfonts8_asc(buf_back, binfo->scrnx, 0, 0, COL8_FFFFFF, s); /* 座標書く */
79                      sheet_slide(shtctl, sht_mouse, mx, my); /* sheet_refresh を含む */
80                  }
81              }
82          }
83      }
84  }
```

我们准备了 2 个图层，分别是sht_back 和sht_mouse，还准备了 2 个缓冲区buf_back 和buf_mouse，用于在

其中描绘图形。以前我们指定为`binfo —> vram` 的部分，现在有很多都改成了`buf_back`。

运行。

3　提高叠加处理速度（1）（haribb07c）

那么怎样才能提高速度呢？首先，我们从鼠标指针的移动，也就是图层的移动来思考一下。

鼠标指针虽然最多只有 16×16=256 个像素，可根据 harib07b 的原理，只要它稍一移动，程序就会对整个画面进行刷新，也就是重新描绘 320×200=64 000 个像素。而实际上，只重新描绘移动相关的部分，也就是移动前后的部分就可以了，即 256×2=512 个像素。这只是 64 000 像素的 0.8% 而已，所以有望提速很多。现在我们根据这个思路写一下程序。

§

```
1 void sheet_refreshsub(struct SHTCTL *ctl, int vx0, int vy0, int vx1, int vy1)
2 {
3     int h, bx, by, vx, vy;
4     unsigned char *buf, c, *vram = ctl->vram;
5     struct SHEET *sht;
6     for (h = 0; h <= ctl->top; h++) {
7         sht = ctl->sheets[h];
8         buf = sht->buf;
9         for (by = 0; by < sht->bysize; by++) {
```

```
10              vy = sht->vy0 + by;
11              for (bx = 0; bx < sht->bxsize; bx++) {
12                  vx = sht->vx0 + bx;
13                  if (vx0 <= vx && vx < vx1 && vy0 <= vy && vy < vy1) {
14                      c = buf[by * sht->bxsize + bx];
15                      if (c != sht->col_inv) {
16                          vram[vy * ctl->xsize + vx] = c;
17                      }
18                  }
19              }
20          }
21      }
22      return;
23  }
```

这个函数几乎和sheet_refresh 一样，唯一的不同点在于它能使用 vx0～vy1 指定刷新的范围。

§

现在我们使用这个 refreshsub 函数来提高sheet_slide 的运行速度。

```
1  void sheet_slide(struct SHTCTL *ctl, struct SHEET *sht, int vx0, int vy0)
2  {
```

```
3     int old_vx0 = sht->vx0, old_vy0 = sht->vy0;
4     sht->vx0 = vx0;
5     sht->vy0 = vy0;
6     if (sht->height >= 0) { /* もしも表示中なら、新しい下じきの情報に沿って画面を描き直す */
7         sheet_refreshsub(ctl, old_vx0, old_vy0, old_vx0 + sht->bxsize, old_vy0 + sht->bysize);
8         sheet_refreshsub(ctl, vx0, vy0, vx0 + sht->bxsize, vy0 + sht->bysize);
9     }
10    return;
11 }
```

这段程序所做的是：首先记住移动前的显示位置，再设定新的显示位置，最后只要重新描绘移动前和移动后的地方。

§

我们所说的在图层上显示文字，实际上并不是改写图层的全部内容。假设我们已经写了 20 个字，那么 8×16×20=2560，也就是仅仅重写 2560 个像素的内容就应该足够了。但现在每次却要重写 64 000 个像素的内容，所以速度才那么慢。

这么说来，这里好像也可以使用 refreshsub，那么我们就来重新编写函数`sheet_refresh` 吧。

```
1 void sheet_refresh(struct SHTCTL *ctl, struct SHEET *sht, int bx0, int by0, int bx1, int by1)
2 {
3     if (sht->height >= 0) { /* もしも表示中なら、新しい下じきの情報に沿って画面を描き直す */
```

```
4           sheet_refreshsub(ctl, sht->vx0 + bx0, sht->vy0 + by0, sht->vx0 + bx1, sht->vy0 + by1);
5       }
6       return;
7   }
```

所谓指定范围，并不是直接指定画面内的坐标，而是以缓冲区内的坐标来表示。这样一来，HariMain 就可以不考虑图层在画面中的位置了。

相应修改sheet_updown 函数。做了改动的只有sheet_refresh（ctl）这部分（有两处）。

改写 HariMain，改写了其中的sheet_refresh，变更点共有 4 个。只有每次要往buf_back 中写入信息时，才进行sheet_refresh。

4　提高叠加处理速度（2）（harib07d）

sheet_refreshsub 即使不写入像素内容，也要多次执行 if 语句，这一点不太好，如果能改善一下，速度应该会提高不少。

按照上面这种写法，即便只刷新图层的一部分，也要对所有图层的全部像素执行 if 语句，判断“是写入呢，还是不写呢”。而对于刷新范围以外的部分，就算执行 if 判断语句，最后也不会进行刷新，所以这纯粹就是一种浪费。既然如此，最初就应该把 for 语句的范围限定在刷新范围之内。

```
1   void sheet_refreshsub(struct SHTCTL *ctl, int vx0, int vy0, int vx1, int vy1)
2   {
3       int h, bx, by, vx, vy, bx0, by0, bx1, by1;
```

```
4     unsigned char *buf, c, *vram = ctl->vram;
5     struct SHEET *sht;
6     for (h = 0; h <= ctl->top; h++) {
7         sht = ctl->sheets[h];
8         buf = sht->buf;
9         /* vx0 ～ vy1 を使って、bx0 ～ by1 を逆算する */
10        bx0 = vx0 - sht->vx0;
11        by0 = vy0 - sht->vy0;
12        bx1 = vx1 - sht->vx0;
13        by1 = vy1 - sht->vy0;
14        if (bx0 < 0) { bx0 = 0; }
15        if (by0 < 0) { by0 = 0; }
16        if (bx1 > sht->bxsize) { bx1 = sht->bxsize; }
17        if (by1 > sht->bysize) { by1 = sht->bysize; }
18        for (by = by0; by < by1; by++) {
19            vy = sht->vy0 + by;
20            for (bx = bx0; bx < bx1; bx++) {
21                vx = sht->vx0 + bx;
22                c = buf[by * sht->bxsize + bx];
23                if (c != sht->col_inv) {
24                    vram[vy * ctl->xsize + vx] = c;
25                }
```

```
26              }
27          }
28      }
29      return;
30  }
```

改良的关键在于，bx 在 for 语句中并不是在 0 到 bxsize 之间循环，而是在 bx0 到 bx1 之间循环（对于 by 也一样）。而 bx0 和 bx1 都是从刷新范围“倒推”求得的。倒推其实就是把公式变形转换了一下，计算 vx0 的坐标相当于 bx 中的哪个位置，然后把它作为 bx0。其他的坐标处理方法也一样。

运行一下！