第**8**天　鼠标控制与**32**位模式切换

- 鼠标解读（1）（harib05a）
- 稍事整理（harib05b）
- 鼠标解读（2）（harib05c）
- 移动鼠标指针（harib05d）
- 通往32位模式之路

1　鼠标解读（1）（harib05a）

好，现在我们已经能从鼠标取得数据了。紧接着的问题是要解读这些数据，调查鼠标是怎么移动的，然后结合鼠标的动作，让鼠标指针相应地动起来。这说起来简单，但做起来呢……事实上编起程序来，也很简单。（笑）

我们要先来对bootpack.c的HariMain函数进行一些修改。

这次**HariMain**的修改部分

```
unsigned char mouse_dbuf[3], mouse_phase;

enable_mouse();
mouse_phase = 0; /* 进入到等待鼠标的0xfa的状态 */

for (;;) {
    io_cli();
    if (fifo8_status(&keyfifo) + fifo8_status(&mousefifo) == 0) {
        io_stihlt();
    } else {
        if (fifo8_status(&keyfifo) != 0) {
            i = fifo8_get(&keyfifo);
            io_sti();

            sprintf(s, "%02X", i);
            boxfill8(binfo->vram, binfo->scrnx, COL8_008484,  0, 16, 15, 31);
            putfonts8_asc(binfo->vram, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
        } else if (fifo8_status(&mousefifo) != 0) {
            i = fifo8_get(&mousefifo);
            io_sti();
            if (mouse_phase == 0) {
                /* 等待鼠标的0xfa的状态 */
                if (i == 0xfa) {
                    mouse_phase = 1;
                }
            } else if (mouse_phase == 1) {
                /* 等待鼠标的第一字节 */
                mouse_dbuf[0] = i;
                mouse_phase = 2;
            } else if (mouse_phase == 2) {
                /* 等待鼠标的第二字节 */
                mouse_dbuf[1] = i;
                mouse_phase = 3;
            } else if (mouse_phase == 3) {
                /* 等待鼠标的第三字节 */
                mouse_dbuf[2] = i;
                mouse_phase = 1;
                /* 鼠标的3个字节都齐了，显示出来 */
                sprintf(s, "%02X %02X %02X", mouse_dbuf[0], mouse_dbuf[1], mouse_dbuf[2]);
                                                        boxfill8(binfo->vram,    binfo-
>scrnx, COL8_008484, 32, 16, 32 + 8 * 8 - 1, 31);
                putfonts8_asc(binfo->vram, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
            }
        }
    }
}
```

这段程序要做什么事情呢？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。

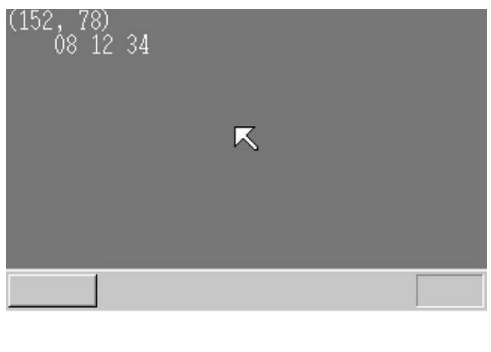
其他地方没有什么难点。不过为了让大家看得更清楚，还是在这里写一下。

```
if (mouse_phase == 0) {
    各种处理;
} else if (mouse_phase == 1) {
    各种处理;
} else if (mouse_phase == 2) {
    各种处理;
} else if (mouse_phase == 3) {
    各种处理;
}
```

对于不同的mouse_phase值，相应地做各种不同的处理。

■■■■■

我们赶紧运行一下试试看吧。“make　run”，然后点击鼠标或者是滚动鼠标，可以看到各种反应。



显示出**3**个字节

屏幕上会出现类似于“08　12　34”之类的3字节数字。如果移动鼠标，这个“08”部分（也就是mouse_dbuf[0]）的“0”那一位，会在0～3的范围内变化。另外，如果只是移动鼠标，08部分的“8”那一位，不会有任何变化，只有当点击鼠标的时候它才会变化。不仅左击有反应，右击和点击中间滚轮时都会有反应。不管怎样点击鼠标，这个值会在8～F之间变化。

上述“12”部分（mouse_dbuf[1]）与鼠标的左右移动有关系，“34”部分（mouse_dbuf[2]）则与鼠标的上下移动有关系。

趁着这个机会，请大家仔细观察一下数字与鼠标动作的关系。我们要利用这些知识去解读这3个字节的数据。

2　稍事整理（**harib05b**）

HariMain有点乱，我们来整理一下。

修改后的**bootpack.c**节选

```
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);

    for (;;) {
        io_cli();
        if (fifo8_status(&keyfifo) + fifo8_status(&mousefifo) == 0) {
            io_stihlt();
        } else {
            if (fifo8_status(&keyfifo) != 0) {
                i = fifo8_get(&keyfifo);
                io_sti();
                sprintf(s, "%02X", i);
                boxfill8(binfo->vram, binfo->scrnx, COL8_008484,  0, 16, 15, 31);
                putfonts8_asc(binfo->vram, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
            } else if (fifo8_status(&mousefifo) != 0) {
                i = fifo8_get(&mousefifo);
                io_sti();
                if (mouse_decode(&mdec, i) != 0) {
                    /* 3字节都凑齐了，所以把它们显示出来*/
                    sprintf(s, "%02X %02X %02X", mdec.buf[0], mdec.buf[1], mdec.buf[2]);
                                                                boxfill8(binfo->vram,   binfo-
>scrnx, COL8_008484, 32, 16, 32 + 8 * 8 - 1, 31);
                    putfonts8_asc(binfo->vram, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
                }
            }
        }
    }
}

void enable_mouse(struct MOUSE_DEC *mdec)
{
    /* 鼠标有效 */
    wait_KBC_sendready();
    io_out8(PORT_KEYCMD, KEYCMD_SENDTO_MOUSE);
    wait_KBC_sendready();
    io_out8(PORT_KEYDAT, MOUSECMD_ENABLE);
    /* 顺利的话，ACK(0xfa)会被送过来 */
    mdec->phase = 0; /* 等待0xfa的阶段 */
    return;
}

int mouse_decode(struct MOUSE_DEC *mdec, unsigned char dat)
{
    if (mdec->phase == 0) {
        /* 等待鼠标的0xfa的阶段 */
        if (dat == 0xfa) {
            mdec->phase = 1;
        }
        return 0;
    }
    if (mdec->phase == 1) {
        /* 等待鼠标第一字节的阶段 */
```

```
            mdec->buf[0] = dat;
            mdec->phase = 2;
            return 0;
        }
        if (mdec->phase == 2) {
            /* 等待鼠标第二字节的阶段 */
            mdec->buf[1] = dat;
            mdec->phase = 3;
            return 0;
        }
        if (mdec->phase == 3) {
            /* 等待鼠标第二字节的阶段 */
            mdec->buf[2] = dat;
            mdec->phase = 1;
            return 1;
        }
        return -1; /* 应该不可能到这里来 */
    }
```

上面几乎没有任何新东西。我们创建了一个结构体MOUSE_DEC。DEC是decode的缩写。我们创建这个结构体，是想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，笔者附加了将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。

我们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。

测试一下，运行“make run”，没有问题，能正常运行。太好了！

3　鼠标解读（2）（harib05c）

程序已经很清晰了，我们继续解读程序。首先对mouse_decode函数略加修改。

bootpack.c 节选

```
struct MOUSE_DEC {
    unsigned char buf[3], phase;
    int x, y, btn;
};

int mouse_decode(struct MOUSE_DEC *mdec, unsigned char dat)
{
    if (mdec->phase == 0) {
        /* 等待鼠标的0xfa的阶段 */
        if (dat == 0xfa) {
            mdec->phase = 1;
        }
        return 0;
    }
    if (mdec->phase == 1) {
        /* 等待鼠标第一字节的阶段 */
        if ((dat & 0xc8) == 0x08) {
            /* 如果第一字节正确 */
            mdec->buf[0] = dat;
            mdec->phase = 2;
        }
        return 0;
    }
    if (mdec->phase == 2) {
        /* 等待鼠标第二字节的阶段 */
        mdec->buf[1] = dat;
        mdec->phase = 3;
        return 0;
    }
    if (mdec->phase == 3) {
        /* 等待鼠标第三字节的阶段 */
        mdec->buf[2] = dat;
        mdec->phase = 1;
        mdec->btn = mdec->buf[0] & 0x07;
        mdec->x = mdec->buf[1];

        mdec->y = mdec->buf[2];
        if ((mdec->buf[0] & 0x10) != 0) {
            mdec->x |= 0xffffff00;
        }
        if ((mdec->buf[0] & 0x20) != 0) {
            mdec->y |= 0xffffff00;
        }
        mdec->y = - mdec->y; /* 鼠标的y方向与画面符号相反 */
        return 1;
    }
    return -1; /* 应该不会到这儿来 */
}
```

■■■■■

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。

另外，笔者还修改了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。

虽说基本上不这么做也行，但鼠标连线偶尔也会有接触不良、即将断线的可能，这时就会产生不该有的数据丢失，这样一来数据会错开一个字节。数据一旦错位，就不能顺利解读，那问题可就大了。而如果添加上对第一字节的检查，就算出了问题，鼠标也只是动作上略有失误，很快就能纠正过来，所以笔者加上了这项检查。

■■■■■

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。

■■■■■

这样，鼠标数据的解读就完成了。现在我们来修改一下显示部分。

HariMain 节选

```
    } else if (fifo8_status(&mousefifo) != 0) {
        i = fifo8_get(&mousefifo);
        io_sti();
        if (mouse_decode(&mdec, i) != 0) {
            /* 数据的3个字节都齐了，显示出来 */

            sprintf(s, "[lcr %4d %4d]", mdec.x, mdec.y);
            if ((mdec.btn & 0x01) != 0) {
                s[1] = 'L';
            }
            if ((mdec.btn & 0x02) != 0) {
                s[3] = 'R';
            }
            if ((mdec.btn & 0x04) != 0) {
                s[2] = 'C';
            }
            boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 32, 16, 32 + 15 * 8 - 1, 31);
            putfonts8_asc(binfo->vram, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
        }
    }
```

虽然程序中会检查mdec.btn的值，用3个if语句将s的值置换成相应的字符串，不过这一部分，暂时先不要管了。这样，程序就变成以下这样。

```
    sprintf(s, "[lcr %4d %4d]", mdec.x, mdec.y);
    boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 32, 16, 32 + 15 * 8 - 1, 31);
    putfonts8_asc(binfo->vram, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
```

这与以前的程序很相似，仅仅用来显示字符串。现在加上刚才的if语句：

```
    if ((mdec.btn & 0x01) != 0) {
        s[1] = 'L';
```

```
    }
```

这行程序的意思是，如果mdec.btn的最低位是1，就把s的第2个字符（注：第1个字符是s[0] ）换成‘L’。这就是将小写字符置换成大写字符。其他的if语句也都这样理解吧。

■■■■■

执行一下看看。

移动鼠标

点击鼠标

反应都很正常，心情大好。

4　移动鼠标指针（**harib05d**）

鼠标的解读部分已经完成了，我们再改一改图形显示部分，让鼠标指针在屏幕上动起来。

HariMain 节选

```
} else if (fifo8_status(&mousefifo) != 0) {
    i = fifo8_get(&mousefifo);
    io_sti();
    if (mouse_decode(&mdec, i) != 0) {
        /* 数据的3个字节都齐了，显示出来  */
        sprintf(s, "[lcr %4d %4d]", mdec.x, mdec.y);
        if ((mdec.btn & 0x01) != 0) {
            s[1] = 'L';
        }
        if ((mdec.btn & 0x02) != 0) {
            s[3] = 'R';
        }
        if ((mdec.btn & 0x04) != 0) {
            s[2] = 'C';
        }
        boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 32, 16, 32 + 15 * 8 - 1, 31);
        putfonts8_asc(binfo->vram, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
        /* 鼠标指针的移动 */
        boxfill8(binfo->vram, binfo->scrnx, COL8_008484, mx, my, mx + 15, my + 15); /* 隐藏鼠
标 */
        mx += mdec.x;
        my += mdec.y;
        if (mx < 0) {
            mx = 0;
        }
        if (my < 0) {
            my = 0;
        }
        if (mx > binfo->scrnx - 16) {
            mx = binfo->scrnx - 16;
        }
        if (my > binfo->scrny - 16) {
            my = binfo->scrny - 16;
        }
        sprintf(s, "(%3d, %3d)", mx, my);
        boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 0, 0, 79, 15); /* 隐藏坐标  */
        putfonts8_asc(binfo->vram, binfo->scrnx, 0, 0, COL8_FFFFFF, s); /* 显示坐标 */
        putblock8_8(binfo->vram, binfo->scrnx, 16, 16, mx, my, mcursor, 16); /* 描画鼠标 */
    }
}
```

这次修改的程序，到/* 鼠标指针的移动 */之前为止，与以前相同，不再解释大家应该也明白。

■■■■■

至于其以后的部分，则是先隐藏掉鼠标指针，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。

好了，我们来测试一下，运行“make run”。然后晃一晃鼠标，结果如下：

动起来了！

太棒了！

鼠标指针总算动起来了！经过长期的艰苦奋战，终于胜利了。为了让鼠标指针能动起来，我们从第5天的下午就开始准备，到第8天下午才完成。

但也正是因为经过这番苦战，我们既完成了GDT/IDT/PIC的初始化，又学会了自由使用栈和FIFO缓冲区，还学会了处理键盘中断。接下来就会轻松很多。

心里实在很高兴，于是多动了几下鼠标。嗯？

只要鼠标一接触到装饰在屏幕下部的任务栏，就会变成下页图那样。这是因为我们没有考虑到叠加处理，所以画面就出问题了。这个话题留到以后再说，今天剩下的时间，笔者想解说一下asmhead.nas。

啊，糟糕！

5　通往32位模式之路

我们一直都没有说明asmhead.nas中的如同谜一样的大约100行程序。等笔者回过神儿来，已经到了可以说明的时候了。现在就是个好机会，我们来具体看看。

在没有说明的这段程序中，最开始做的事情如下：

asmhead.nas节选

```
; PIC关闭一切中断
;    根据AT兼容机的规格，如果要初始化PIC，
;    必须在CLI之前进行，否则有时会挂起。
;    随后进行PIC的初始化。

        MOV     AL,0xff
        OUT     0x21,AL
        NOP                     ; 如果连续执行OUT指令，有些机种会无法正常运行
        OUT     0xa1,AL

        CLI                     ; 禁止CPU级别的中断
```

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，我们要把中断全部屏蔽掉。

顺便说一下，NOP指令什么都不做，它只是让CPU休息一个时钟长的时间。

■■■■■

再往下看，会看到以下部分。

asmhead.nas节选（续）

```
; 为了让CPU能够访问1MB以上的内存空间，设定A20GATE

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

这里的waitkbdout，等同于wait_KBC_sendready（以后还会详细说明）。这段程序在C语言里的写法大致如下：

```
#define KEYCMD_WRITE_OUTPORT    0xd1
#define KBC_OUTPORT_A20G_ENABLE 0xdf

    /* A20GATE的设定 */
```

```
wait_KBC_sendready();
io_out8(PORT_KEYCMD, KEYCMD_WRITE_OUTPORT);
wait_KBC_sendready();
io_out8(PORT_KEYDAT, KBC_OUTPORT_A20G_ENABLE);
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。这条信号线的作用是什么呢？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。和鼠标的情况很类似哟。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。

■■■■■

我们再往下看。

asmhead.nas节选 **(**续**)**

```
; 切换到保护模式

[INSTRSET "i486p"]              ; “想要使用486指令”的叙述

        LGDT    [GDTR0]         ; 设定临时GDT
        MOV     EAX,CR0
        AND     EAX,0x7fffffff  ; 设bit31为0（为了禁止分页）
        OR      EAX,0x00000001  ; 设bit0为1（为了切换到保护模式）
        MOV     CR0,EAX
        JMP     pipelineflush

pipelineflush:
        MOV     AX,1*8          ;  可读写的段 32bit
        MOV     DS,AX
        MOV     ES,AX
        MOV     FS,AX
        MOV     GS,AX
        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，我们以后还要重新设置。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，进入到不用颁的保护模式。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[1]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。

[1] 本来的说法应该是“protected virtual address mode”，翻译过来就是“受保护的虚拟内存地址模式”。与此相对，从前的16位模式称为“real mode”，它是“real address mode”的省略形式，翻译过来就是“实际地址模式”。这些术语中的“virtual”，“real”的区别在于计算内存地址时，是使用段寄存器的值直接指定地址值的一部分呢，还是通过GDT使用段寄存器的值指定并非实际存在的地址号码。

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（pipeline）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。

■■■■■

我们再往下读程序。

asmhead.nas节选（续）

```
; bootpack的转送

        MOV     ESI,bootpack    ; 转送源
        MOV     EDI,BOTPAK      ; 转送目的地
        MOV     ECX,512*1024/4
        CALL    memcpy

; 磁盘数据最终转送到它本来的位置去

; 首先从启动扇区开始

        MOV     ESI,0x7c00      ; 转送源
        MOV     EDI,DSKCAC      ; 转送目的地
        MOV     ECX,512/4
        CALL    memcpy

; 所有剩下的

        MOV     ESI,DSKCAC0+512 ; 转送源
        MOV     EDI,DSKCAC+512  ; 转送目的地
        MOV     ECX,0
```

```
MOV     CL,BYTE [CYLS]
IMUL    ECX,512*18*2/4  ; 从柱面数变换为字节数/4
SUB     ECX,512/4       ; 减去 IPL
CALL    memcpy
```

简单来说，这部分程序只是在调用memcpy函数。为了让大家掌握这段程序的大意，我们将这段程序写成了C语言形式。虽然写法本身可能不很正确，但有助于大家抓住程序的中心思想。

```
memcpy(bootpack,    BOTPAK,     512*1024/4);
memcpy(0x7c00,      DSKCAC,     512/4     );
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。顺便说一下，IMUL[2]是乘法运算，SUB[3]是减法运算。它们与ADD（加法）运算同属一类。

[2] IMUL，来自英文“integer multipule”（整数乘法）。

[3] SUB，来自英文“substract”（减法）。

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。

■■■■■

后面还剩50行程序，我们继续往下看。

asmhead.nas节选（续）

```
; 必须由asmhead来完成的工作，至此全部完毕
;   以后就交由bootpack来完成

; bootpack的启动

        MOV     EBX,BOTPAK
        MOV     ECX,[EBX+16]
        ADD     ECX,3           ; ECX += 3;
        SHR     ECX,2           ; ECX /= 4;
        JZ      skip            ; 没有要转送的东西时
        MOV     ESI,[EBX+20]    ; 转送源
        ADD     ESI,EBX
        MOV     EDI,[EBX+12]    ; 转送目的地
        CALL    memcpy
skip:
        MOV     ESP,[EBX+12]    ; 栈初始值
        JMP     DWORD 2*8:0x0000001b
```

结果我们仍然只是在做memcpy。它对bootpack.hrb的header（头部内容）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，来自英文jump if zero，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。大家可能不明白为什么要做这种处理，但这个问题，必须要等到“纸娃娃系统”的应用程序讲完之后才能讲清楚，所以大家现在不懂也没关系，我们以后还会说明的。

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。

■■■■■

下面要讲的内容可能有点偏离主题，但笔者还是想介绍一下“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

这个内存分布图当然是笔者所做出来的。为什么要做成这呢？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。

如果一开始就制作内存分布图，那么做起操作系统来就会顺利多了。

■■■■■

关于内存分布图就讲这么多，还是让我们回到asmhead.nas的说明上来吧。

asmhead.nas节选（续）

```
waitkbdout:
        IN       AL,0x64
        AND      AL,0x02
        IN       AL,0x60         ; 空读（为了清空数据接收缓冲区中的垃圾数据）
        JNZ     waitkbdout       ; AND的结果如果不是0，就跳到waitkbdout
        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。

JNZ与JZ相反，意思是“jump if not zero”。

■■■■■

只剩下一点点内容了，下面是memcpy程序。

asmhead.nas节选（续）

```
memcpy:
        MOV     EAX,[ESI]
        ADD     ESI,4
        MOV     [EDI],EAX
        ADD     EDI,4
        SUB     ECX,1
        JNZ     memcpy          ; 减法运算的结果如果不是0，就跳转到memcpy
        RET
```

这是复制内存的程序。不用笔者解释，大家也能明白。

■■■■■

最后是剩下来的全部内容。

asmhead.nas节选（续）

```
        ALIGNB  16
GDT0:
        RESB    8               ; NULL selector
        DW      0xffff,0x0000,0x9200,0x00cf ; 可以读写的段（segment）32bit
        DW      0xffff,0x0000,0x9a28,0x0047 ; 可以执行的段（segment）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。所以我们插入了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，所以笔者就适当加了一句“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哟”。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

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

    init_gdtidt();
    init_pic();
    io_sti(); /* IDT/PIC的初始化已经完成，于是开放CPU的中断 */
    fifo8_init(&keyfifo, 32, keybuf);
    fifo8_init(&mousefifo, 128, mousebuf);
    io_out8(PIC0_IMR, 0xf9); /* 开放PIC1和键盘中断(11111001) */
    io_out8(PIC1_IMR, 0xef); /* 开放鼠标中断(11101111)  */

    init_keyboard();

    init_palette();
    init_screen8(binfo->vram, binfo->scrnx, binfo->scrny);
```

■■■■■

夜已经深了，今天就到此为止。在考虑明天要做什么的同时，笔者也决定要睡觉了。晚安！