第5天　结构体、文字显示与GDT/IDT初始化

- 接收启动信息（harib02a）
- 试用结构体（harib02b）
- 试用箭头记号（harib02c）
- 显示字符（harib02d）
- 增加字体（harib02e）
- 显示字符串（harib02f）
- 显示变量值（harib02g）
- 显示鼠标指针（harib02h）
- GDT与IDT的初始化（harib02i）

1　接收启动信息（**harib02a**）

我们今天从哪儿开始讲呢？现在“纸娃娃操作系统”的外观已经有了很大的进步，所以下面做些内部工作吧。

到昨天为止，在bootpack.c里的，都是将0xa0000呀，320、200等数字直接写入程序，而本来这些值应该从asmhead.nas先前保存下来的值中取。如果不这样做的话，当画面模式改变时，系统就不能正确运行。

所以我们就试着用指针来取得这些值。顺便说一下，binfo是bootinfo的缩写，scrn是screen（画面）的缩写。

本次的**HariMain**节选

```
void HariMain(void)
{
    char *vram;
    int xsize, ysize;
    short *binfo_scrnx, *binfo_scrny;
    int *binfo_vram;

    init_palette();
    binfo_scrnx = (short *) 0x0ff4;
    binfo_scrny = (short *) 0x0ff6;
    binfo_vram = (int *) 0x0ff8;
    xsize = *binfo_scrnx;
    ysize = *binfo_scrny;
    vram = (char *) *binfo_vram;
```

这里出现的0x0ff4之类的地址到底是从哪里来的呢？其实这些地址仅仅是为了与asmhead. nas保持一致才出现的。

另外，我们把显示画面背景的部分独立出来，单独做成一个函数init_screen。独立的功能做成独立的函数，这样程序读起来要容易一些。

好了，做完了。执行一下吧。……嗯，暂时好像没什么问题。只是没什么意思，因为画面显示内容没有变化。

2　试用结构体（**harib02b**）

上面的方法倒也不能说不好，只是代码的行数多了些，不太令人满意。而如果采用之前的COLUMN-2里（第4章）的写法：

```
xsize = *((short *) 0x0ff4);
```

程序长度是变短了，但这样的写法看起来就像是使用了什么特殊技巧。我们还是尝试一下更普通的写法吧。

本次的**HariMain**节选

```
struct BOOTINFO {
    char cyls, leds, vmode, reserve;
    short scrnx, scrny;
    char *vram;
};

void HariMain(void)
{
    char *vram;
    int xsize, ysize;
    struct BOOTINFO *binfo;

    init_palette();
    binfo = (struct BOOTINFO *) 0x0ff0;
    xsize = (*binfo).scrnx;
    ysize = (*binfo).scrny;
    vram = (*binfo).vram;
```

我们写成了上面这种形式。struct是新语句。这里第一次出现结构体，或许有人不太理解，如果不明白的话请一定看看后面的专栏。

最开始的struct命令只是把一串变量声明集中起来，统一叫做“struct　BOOTINFO”。最初是1字节的变量cyls，接着是1字节的变量leds，照此下去，最后是vram。这一串变量一共是12字节。有了这样的声明，以后“struct　BOOTINFO”就可以作为一个新的变量类型，用于各种场合，可以像int、char那样的变量类型一样使用。

这里的*binfo就是这种类型的变量，为了表示其中的scrnx，使用了（*binfo）.scrnx这种写法。如果不加括号直接写成*binfo.scrnx，虽然更容易懂，但编译器会误解成*（binfo.scrnx），出现错误。所以，括号虽然不太好看，但不能省略。

COLUMN-5　结构体的简单说明

5.2节[1]里的这种结构体的使用方法，比较特殊。我们先看一个普通的例子。

[1] 第5天的第2小节。——译者注

普通的结构体使用方法

```
void HariMain(void)
{
    struct BOOTINFO abc;
```

```
    abc.scrnx = 320;
    abc.scrny = 200;
    abc.vram  = 0xa0000;
    （以下略）
}
```

先定义一个新结构体变量abc，然后再给这个结构体变量的各个元素赋值。结构体的好处是，可以像下面这样将各种东西都一股脑儿地传递过来。

```
func(abc);
```

如果没有结构体，就只能将各个参数一个一个地传递过来了。

```
func(scrnx, scrny, vram, ...);
```

所以很多时候会将有某种意义的数据都归纳到一个结构体里，这样就方便多了。但如果归纳方法搞错了，反而带来更多麻烦。

为了让程序能一看就懂，要这样写结构体的内部变量：在结构体变量名的后面加一个点（.），然后再写内部变量名，这是规则。

■■■■■

下一步是使用指针。这是5.2节中的使用方法。声明方法如下：

变量类型名 *指针变量名;(回想一下char *p;)

而这次的变量类型是struct BOOTINFO，变量名是binfo，所以写成如下形式：

```
struct BOOTINFO *binfo;
```

这里的binfo表示指针变量。地址用4个字节来表示，所以binfo是4字节变量。

因为是指针变量，所以应该首先给指针赋值，否则就不知道要往哪里读写了。可以写成下面这样：

```
binfo = (struct BOOTINFO *)0x0ff0;
```

本来想写“binfo =0x0ff0;”的，但由于总出警告，很讨厌，所以我们就进行了类型转换。

设定了指针地址以后，这12个字节的结构体用起来就没问题了。这样我们可以不再直接使用内存地址，而是使用*binfo来表示这个内存地址上12字节的结构体。这与“char *p;”中的*p表示p地址的1字节是同样道理。

前面说过，想要表示结构体abc中的scrnx时，就用abc.scrnx。与此类似，这里用(*binfo).scrnx来表示。需要括号的理由在5.2节中已经写了。因此语句写作：

```
xsize = (*binfo).scrnx;
```

3　试用箭头记号（**harib02c**）

事实上，在C语言里常常会用到类似于（*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，我们称之为箭头标记方式。前面也讲到过，a[i]是*（a ＋ i）的省略表现形式所以可以说C语言中关于指针的省略表现形式很充实，很丰富。

使用箭头，可以将“xsize = (*binfo).scrnx;”写成“xsize = binfo—>scrnx；”，简单又方便。不过我们还想更简洁些，即连变量xsize都不用，而是直接以binfo—>scrnx来代替xsize。

本次的**HariMain**节选

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

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

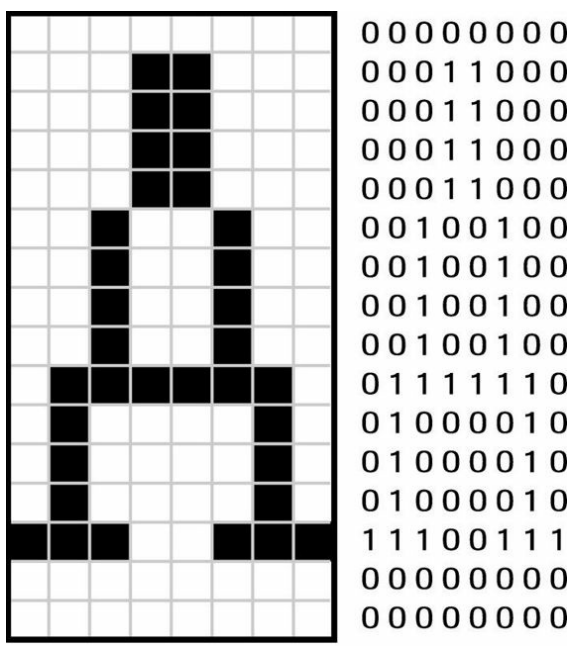
哦，看上去真清爽。我们运行一下“make run”，运行正常。

这次我们想了很多方法，但这些都只是C语言写法的问题，编译成机器语言以后，几乎没有差别。既然没有差别，笔者认为写得清晰一些没什么坏处，所以决定今后积极使用这种写法。讨厌在写法上花工夫的人不使用结构体也没关系，再退一步，还可以不用指针，继续使用write_mem8什么的也没问题。可以根据自己的理解程度和习惯，选择自己喜欢的方式。

4　显示字符（**harib02d**）

内部的处理差不多了，我们还是将重点放回到外部显示上来吧。到昨天为止，我们算是画出了一幅稍微像样的画，今天就来在画面上写字。以前我们显示字符主要靠调用BIOS函数，但这次是32位模式，不能再依赖BIOS了，只能自力更生。

那么怎么显示字符呢？字符可以用8×16的长方形像素点阵来表示。想象一个下图左边的数据，然后按下图右边所示的方法置换成0和1，这个方法好像不错。然后根据这些数据在画面上打上点就肯定能显示出字符了。8“位”是一个字节，而1个字符是16个字节。



字符的原形？

大家可能会有各种想法，比如“我觉得8×16的字太小了，想显示得更大一些”、“还是小点儿的字好”等。不过刚开始我们就先这样吧，一上来要求太多的话，就没有办法往前进展了。

■■■■■

像这种描画文字形状的数据称为字体（font）数据，那这种字体数据是怎样写到程序里的呢？有一种临时方案：

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

其实这仅仅是将刚才的0和1的排列，重写成十六进制数而已。C语言无法用二进制数记录数据，只能写成十六进制或八进制。嗯，读起来真费劲呀。嫌字体不好看，想手动修正一下，都不知道到底需要修改哪儿。但是暂时就先这样吧，以后再考虑这个问题。

数据齐备之后，只要描画到画面上就可以了。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。于是我们制作了下面这个函数。

```
void putfont8(char *vram, int xsize, int x, int y, char c, char *font)
{
    int i;
    char d; /* data */
    for (i = 0; i < 16; i++) {
        d = font[i];
        if ((d & 0x80) != 0) { vram[(y + i) * xsize + x + 0] = c; }
        if ((d & 0x40) != 0) { vram[(y + i) * xsize + x + 1] = c; }
        if ((d & 0x20) != 0) { vram[(y + i) * xsize + x + 2] = c; }
        if ((d & 0x10) != 0) { vram[(y + i) * xsize + x + 3] = c; }
        if ((d & 0x08) != 0) { vram[(y + i) * xsize + x + 4] = c; }
        if ((d & 0x04) != 0) { vram[(y + i) * xsize + x + 5] = c; }
        if ((d & 0x02) != 0) { vram[(y + i) * xsize + x + 6] = c; }
        if ((d & 0x01) != 0) { vram[(y + i) * xsize + x + 7] = c; }
    }
    return;
}
```

if语句是第一次登场，我们来介绍一下。if语句先检查“()”内的条件式，当条件成立时，就执行“{}”内的语句，条件不成立时，什么都不做。

&是以前曾出现过的AND（“与”）运算符。0x80也就是二进制数10000000，它与d进行“与”运算的结果如果是0，就说明d的最左边一位是0。反之，如果结果不是0，则d的最左边一位就是1。“!=”是不等于的意思，在其他语言中，有时写作“<>”。

■■■■■

虽然这样也能显示出“A”来，但还是把程序稍微整理一下比较好，因为现在的程序又长运行速度又慢。

```
void putfont8(char *vram, int xsize, int x, int y, char c, char *font)
{
    int i;
    char *p, d /* data */;
    for (i = 0; i < 16; i++) {
        p = vram + (y + i) * xsize + x;
        d = font[i];
        if ((d & 0x80) != 0) { p[0] = c; }
        if ((d & 0x40) != 0) { p[1] = c; }
        if ((d & 0x20) != 0) { p[2] = c; }
        if ((d & 0x10) != 0) { p[3] = c; }
        if ((d & 0x08) != 0) { p[4] = c; }
        if ((d & 0x04) != 0) { p[5] = c; }
        if ((d & 0x02) != 0) { p[6] = c; }
        if ((d & 0x01) != 0) { p[7] = c; }
    }
    return;
}
```

这样就好多了，我们就用这段程序吧。

下面将这段程序嵌入到bootpack.c中进行整理。大家仔细看看，如果顺利的话，能显示出字符“A”。紧张激动的时刻到了，运行“make run”。哦，“A”显示出来了！

显示出来了，真高兴

5　增加字体（**harib02e**）

虽然字符“A”显示出来了，但这段程序只能显示“A”而不能显示别的字符。所以我们需要很多别的字体来显示其他字符。英文字母就有26个，分别有大写和小写，还有10个数字，再加上各种符号肯定超过30个了。啊，还有很多，太麻烦了，所以我们决定沿用OSASK的字体数据。当然，我们暂时还不考虑显示汉字什么的。这些复杂的东西，留待以后再做。现在我们集中精力解决字母显示的问题。

另外，这里沿用的OSASK的字体，其作者不是笔者，而是平木敬太郎先生和圣人（Kiyoto）先生。事先已经从他们那里得到了使用许可权，所以可以自由使用这种字体。

我们这次就将hankaku.txt这个文本文件加入到我们的源程序大家庭中来。这个文件的内容如下：

hankaku.txt的内容

```
char 0x41
........
...**...
...**...
...**...
...**...
..*..*..
..*..*..
..*..*..
.******.
.*....*.
.*....*.
.*....*.
***..***
........
........
```

这比十六进制数和只有0和1的二进制数都容易看一些。

■■■■■

当然，这既不是C语言，也不是汇编语言，所以需要专用的编译器。新做一个编译器很麻烦，所以我们还是使用在制作OSASK时曾经用过的工具（makefont.exe）。说是编译器，其实有点言过其实了，只不过是将上面这样的文本文件（256个字符的字体文件）读进来，然后输出成16×256=4096字节的文件而已。

编译后生成hankaku.bin文件，但仅有这个文件还不能与bootpack.obj连接，因为它不是目标（obj）文件。所以，还要加上连接所必需的接口信息，将它变成目标文件。这项工作由bin2obj.exe来完成。它的功能是将所给的文件自动转换成目标程序，就像将源程序转换成汇编那样。也就是说，好像将下面这两行程序编译成了汇编：

```
_hankanku:
    DB  各种数据（共4096字节）
```

当然，如果大家不喜欢现在这种字体的话，可以随便修改hankaku.txt。本书的中心任务是自制操作系统，所以字体就由大家自己制作了。

各种工具的使用方法，请参阅Makefile的内容。因为不是很难，这里就不再说明了。

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。这样，C编译器就能够知道它是外部数据，并在编译时做出相应调整。

■■■■■

OSASK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，放在自“hankaku + 0x41 * 16”开始的16字节里。C语言中A的字符编码可以用’A’来表示，正好可以用它来代替0x41，所以也可以写成“hankaku + ‘A’ * 16”。

我们使用以上字体数据，向bootpack.c里添加了很多内容，请大家浏览一下。如果顺利的话，会显示出“ABC 123”。下面就来“make run”一下吧。很好，运行正常。

本次的**HariMain**的内容

```
void HariMain(void)
{
    struct BOOTINFO *binfo = (struct BOOTINFO *) 0x0ff0;
    extern char hankaku[4096];

    init_palette();
    init_screen(binfo->vram, binfo->scrnx, binfo->scrny);
    putfont8(binfo->vram, binfo->scrnx,  8, 8, COL8_FFFFFF, hankaku + 'A' * 16);
    putfont8(binfo->vram, binfo->scrnx, 16, 8, COL8_FFFFFF, hankaku + 'B' * 16);
    putfont8(binfo->vram, binfo->scrnx, 24, 8, COL8_FFFFFF, hankaku + 'C' * 16);
    putfont8(binfo->vram, binfo->scrnx, 40, 8, COL8_FFFFFF, hankaku + '1' * 16);
    putfont8(binfo->vram, binfo->scrnx, 48, 8, COL8_FFFFFF, hankaku + '2' * 16);
    putfont8(binfo->vram, binfo->scrnx, 56, 8, COL8_FFFFFF, hankaku + '3' * 16);

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

各种字符

6　显示字符串（**harib02f**）

仅仅显示6个字符，就要写这么多代码，实在不太好看。

```
putfont8(binfo->vram, binfo->scrnx,  8, 8, COL8_FFFFFF, hankaku + 'A' * 16);
putfont8(binfo->vram, binfo->scrnx, 16, 8, COL8_FFFFFF, hankaku + 'B' * 16);
putfont8(binfo->vram, binfo->scrnx, 24, 8, COL8_FFFFFF, hankaku + 'C' * 16);
putfont8(binfo->vram, binfo->scrnx, 40, 8, COL8_FFFFFF, hankaku + '1' * 16);
putfont8(binfo->vram, binfo->scrnx, 48, 8, COL8_FFFFFF, hankaku + '2' * 16);
putfont8(binfo->vram, binfo->scrnx, 56, 8, COL8_FFFFFF, hankaku + '3' * 16);
```

所以笔者打算制作一个函数，用来显示字符串。既然已经学到了目前这一步，做这样一个函数也没什么难的。嗯，开始动手吧……好，做完了。

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

C语言中，字符串都是以0x00结尾的，所以可以这么写。函数名带着asc，是为了提醒笔者字符编码使用了ASCII。

这里还要再说明一点，所谓字符串是指按顺序排列在内存里，末尾加上0x00而组成的字符编码。所以s是指字符串前头的地址，而使用*s就可以读取字符编码。这样，仅利用下面这短短的一行代码就能够达到目的了。

```
putfonts8_asc(binfo->vram, binfo->scrnx,  8, 8, COL8_FFFFFF, "ABC 123");
```

试试看吧。……顺利运行了。

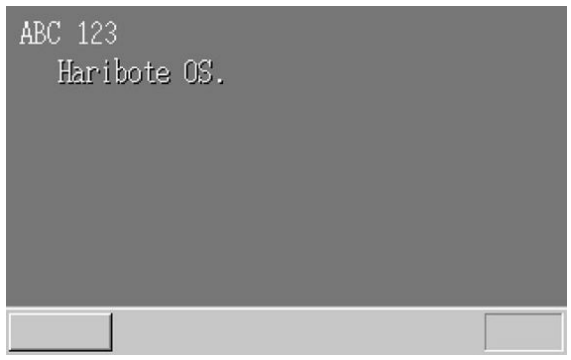
我们再稍微加工一下，……好，完成了。

整理后的**HariMain**

```
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();
    }
}
```



显示出任意字符串

7　显示变量值（**harib02g**）

现在可以显示字符串了，那么这一节我们就来显示变量的值。能不能显示变量值，对于操作系统的开发影响很大。这是因为程序运行与想象中不一致时，将可疑变量的值显示出来是最好的方法。

> 习惯了在Windows中开发程序的人，如果想看到变量的值，用调试器[1]（debugger）很容易就能看到，但是在开发操作系统过程中可就没那么容易了。就像用Windows的调试器不能对Linux的程序进行调试一样，Windows的调试器也不能对我们的“纸娃娃操作系统”的程序进行调试，更不要说对操作系统本身进行调试了。如果在“纸娃娃操作系统”中也要使用调试器的话，那只有自己做一个调试器了（也可以移植）。在做出调试器之前，只能通过显示变量值来查看确认问题的地方。

[1]　指调试程序中的错误（bug）时所用的工具，英文是debugger。另外，调试（动词）是debug。

闲话就说这么多，让我们回到正题。那怎么样显示变量的值呢？可以使用sprintf函数。它是printf函数的同类，与printf函数的功能很相近。在开始的时候，我们曾提到过，自制操作系统中不能随便使用printf函数，但sprintf可以使用。因为sprintf不是按指定格式输出，只是将输出内容作为字符串写在内存中。

这个sprintf函数，是本次使用的名为GO的C编译器附带的函数。它在制作者的精心设计之下能够不使用操作系统的任何功能。或许有人会认为，什么呀，那样的话，怎么不做个printf函数呢？这是因为输出字符串的方法，各种操作系统都不一样，不管如何精心设计，都不可避免地要使用操作系统的功能。而sprintf不同，它只对内存进行操作，所以可以应用于所有操作系统。

■■■■■

我们这就来试试这个函数吧。要在C语言中使用sprintf函数，就必须在源程序的开头写上#include　，我们也写上这句话。这样以后就可以随便使用sprintf函数了。接下来在HariMain中使用sprintf函数。

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

sprintf函数的使用方法是：sprintf（地址，格式，值，值，值，……）。这里的地址指定所生成字符串的存放地址。格式基本上只是单纯的字符串，如果有%d这类记号，就置换成后面的值的内容。除了%d，还有%s，%x等符号，它们用于指定数值以什么方式变换为字符串。%d将数值作为十进制数转化为字符串，%x将数值作为十六进制数转化为字符串。

关于格式的详细说明

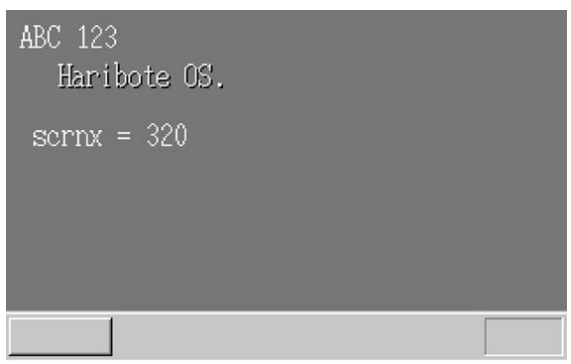
%d	单纯的十进制数
%5d	5位十进制数。如果是123，则在前面加上两个空格，变成"　　123"，强制达到5位
%05d	5位十进制数。如果是123，则在前面加上0，变成"00123"，强制达到5位
%x	单纯的十六进制数。字母部分使用小写abcdef

%X	单纯的十六进制数。字母部分使用大写ABCDEF
%5x	5位十六进制数。如果是456（十进制），则在前面加上两个空格，变成" 1c8"，强制达到5位。还有%5X的形式
%05x	5位十六进制数。如果是456（十进制），则在前面加上两个0，变成"001c8"，强制达到5位。还有%05X的形式

我们来运行一下看看。……运行正常。

说点题外话。因为这本书是在笔者吭哧吭哧写完之后大家才看到的，所以虽然讲到“能显示变量的值了”，“以后调试就容易了”，恐怕大家也很难体会到其中的艰辛。但是，笔者是真刀真枪地编程，在此过程中犯了很多的错（大多都是低级错误）。以前，因为不能显示变量的值，所以发现运行异常的时候，只能拼命读代码，想象变量的值来修改程序，非常辛苦。但从今以后可以显示变量的值就轻松多了。

话说，在分辨率是320×200的屏幕上，8×16的字体可是很大哟（笑）。



可以看见变量的值了！

8　显示鼠标指针（**harib02h**）

估计后面的开发速度会更快，那就赶紧趁着这势头再描画一下鼠标指针吧。思路跟显示字符差不多，程序并不是很难。

首先，将鼠标指针的大小定为16×16。这个定下来之后，下面就简单了。先准备16×16=256字节的内存，然后往里面写入鼠标指针的数据。我们把这个程序写在init_mouse_cursor8里。

```
void init_mouse_cursor8(char *mouse, char bc)
/* 准备鼠标指针（16×16） */
{
    static char cursor[16][16] = {
        "**************..",
        "*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;
            }
            if (cursor[y][x] == 'O') {
                mouse[y * 16 + x] = COL8_FFFFFF;
            }
            if (cursor[y][x] == '.') {
                mouse[y * 16 + x] = bc;
            }
        }
    }
    return;
}
```

变量bc是指back-color，也就是背景色。

要将背景色显示出来，还需要作成下面这个函数。其实很简单，只要将buf中的数据复制到vram中去就可以了。

```
void putblock8_8(char *vram, int vxsize, int pxsize,
    int pysize, int px0, int py0, char *buf, int bxsize)
{
    int x, y;
    for (y = 0; y < pysize; y++) {
        for (x = 0; x < pxsize; x++) {
            vram[(py0 + y) * vxsize + (px0 + x)] = buf[y * bxsize + x];
        }
    }
    return;
```

```
}
```

里面的变量有很多，其中vram和vxsize是关于VRAM的信息。他们的值分别是0xa0000和320。pxsize和pysize是想要显示的图形（picture）的大小，鼠标指针的大小是16×16，所以这两个值都是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。bxsize和pxsize大体相同，但也有时候想放入不同的值，所以还是要分别指定这两个值。

接下来，只要使用以下两个函数就行了。

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

也不知能不能正常运行，试试看。……好，能运行！

鼠标指针出来了

9　GDT与IDT的初始化（harib02i）

鼠标指针显示出来了，我们想做的第一件事就是去移动它，但鼠标指针却一动不动。那是当然，因为我们还没有做出这个功能。……嗯，无论如何想让它动起来。

要怎么样才能让它动呢？……（思考中）……有办法了！首先要将GDT和IDT初始化。不过在此之前，必须说明一下什么是GDT和IDT。

GDT也好，IDT也好，它们都是与CPU有关的设定。为了让操作系统能够使用32位模式，需要对CPU做各种设定。不过，asmhead.nas里写的程序有点偷工减料，只是随意进行了一些设定。如果这样原封不动的话，就无法做出使用鼠标指针所需要的设定，所以我们要好好重新设置一下。

> 那为什么要在asmhead.nas里偷工减料呢？最开始就规规矩矩地设定好不行吗？……嗯，这个问题一下子就戳到痛处了。这里因为笔者希望尽可能地不用汇编语言，而用C语言来写，这样大家更容易理解。所以，asmhead.nas里尽可能少写，只做了运行bootpack.c所必需的一些设定。这次为了使用这个文件，必须再进行设定。如果大家有足够能力用汇编语言编写程序，就不用模仿笔者了，从一开始规规矩矩地做好设定更好。

从现在开始，学习内容的难度要增加不小。以后要讲分段呀，中断什么的，都很难懂，很多程序员都是在这些地方受挫的。从难度上考虑，应该在20天以后讲而不是第5天。但如果现在不讲，几乎所有的装置都不能控制，做起来也没什么意思。笔者不想让大家做没有意思的操作系统。

所以请大家坚持着读下去，先懂个大概，然后再回过头来仔细咀嚼。在一天半以后，内容的难度会回到以前的水平，所以这段时间大家就打起精神加油吧！

■■■■■

先来讲一下分段[1]。回想一下仅用汇编语言编程时，有一个指令叫做ORG。如果不用ORG指令明确声明程序要读入的内存地址，就不能写出正确的程序来。如果写着ORG 0x1234，但程序却没读入内存的0x1234号，可就不好办了。

> [1] 英文是segmentation。

发生这种情况是非常麻烦的。最近的操作系统能同时运行多个程序，这一点也不稀奇。这种时候，如果内存的使用范围重叠了怎么办？这可是一件大事。必须让某个程序放弃执行，同时报出一个“因为内存地址冲突，不能执行”的错误信息。但是，这种错误大家见过吗？没有。所以，肯定有某种方法能解决这个问题。这个方法就是分段。

所谓分段，打个比方说，就是按照自己喜欢的方式，将合计4GB[2]的内存分成很多块（block），每一块的起始地址都看作0来处理。这很方便，有了这个功能，任何程序都可以先写上一句ORG 0。像这样分割出来的块，就称为段（segment）。顺便说一句，如果不用分段而用分页[3]（paging），也能解决问题。不过我们目前还不讨

论分页，可以暂时不考虑它。

[2] GB（Giga Byte，吉字节），指1024MB，可不是Game Boy的省略哟（笑）。

[3] 英文是paging。“分段”的基本思想是将4GB的内存分割；而分页的思想是有多少个任务就要分多少页，　还要对内存进行排序。不过解说到这个程度，大家估计都不懂。以后有机会再详细说明。

需要注意的一点是，我们用16位的时候曾经讲解过的段寄存器。这里的分段，使用的就是这个段寄存器。但是16位的时候，如果计算地址，只要将地址乘以16就可以了。但现在已经是32位了，不能再这么用了。如果写成“MOV　　AL,[DS:EBX]”，CPU会往EBX里加上某个值来计算地址，这个值不是DS的16倍，而是DS所表示的段的起始地址。即使省略段寄存器（segment　register）的地址，也会自动认为是指定了DS。这个规则不管是16位模式还是32位模式，都是一样的。

■■■■■

按这种分段方法，为了表示一个段，需要有以下信息。

- 段的大小是多少
- 段的起始地址在哪里
- 段的管理属性（禁止写入，禁止执行，系统专用等）

CPU用8个字节（=64位）的数据来表示这些信息。但是，用于指定段的寄存器只有16位。或许有人会猜想在32位模式下，段寄存器会扩展到64位，但事实上段寄存器仍然是16位。

那该怎么办才好呢？可以模仿图像调色板的做法。也就是说，先有一个段号[4]，存放在段寄存器里。然后预先设定好段号与段的对应关系。

[4] 英文是segment selector，也有译作“段选择符”的。

调色板中，色号可以使用0~255的数。段号可以用0~8191的数。因为段寄存器是16位，所以本来应该能够处理0~65535范围的数，但由于CPU设计上的原因，段寄存器的低3位不能使用。因此能够使用的段号只有13位，能够处理的就只有位于0~8191的区域了。

段号怎么设定呢？这是对于CPU的设定，不需要像调色板那样使用io_out（由于不是外部设备，当然没必要）。但因为能够使用0~8191的范围，即可以定义8192个段，所以设定这么多段就需要8192×8=65　536字节（64KB）。大家可能会想，CPU没那么大存储能力，不可能存储那么多数据，是不是要写入到内存中去呀。不错，正是这样。这64KB（实际上也可以比这少）的数据就称为GDT。

GDT是“global（segment）descriptor　table”的缩写，意思是全局段号记录表。将这些数据整齐地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内被称作GDTR[5]的特殊寄存器中，设定就完成了。

[5] global (segment) descriptor table register的缩写。

■■■■■

另外，IDT是“interrupt descriptor table”的缩写，直译过来就是“中断记录表”。当CPU遇到外部状况变化，或者是内部偶然发生某些错误时，会临时切换过去处理这种突发事件。这就是中断功能。

我们拿电脑的键盘来举个例子。以CPU的速度来看，键盘特别慢，只是偶尔动一动。就算是重复按同一个键，一秒钟也很难输入50个字符。而CPU在1/50秒的时间内，能执行200万条指令（CPU主频100MHz时）。CPU每执行200万条指令，查询一次键盘的状况就已经足够了。如果查询得太慢，用户输入一个字符时电脑就会半天没反应。

要是设备只有键盘，用“查询”这种处理方法还好。但事实上还有鼠标、软驱、硬盘、光驱、网卡、声卡等很多需要定期查看状态的设备。其中，网卡还需要CPU快速响应。响应不及时的话，数据就可能接受失败，而不得不再传送一次。如果因为害怕处理不及时而靠查询的方法轮流查看各个设备状态的话，CPU就会穷于应付，不能完成正常的处理。

正是为解决以上问题，才有了中断机制。各个设备有变化时就产生中断，中断发生后，CPU暂时停止正在处理的任务，并做好接下来能够继续处理的准备，转而执行中断程序。中断程序执行完以后，再调用事先设定好的函数，返回处理中的任务。正是得益于中断机制，CPU可以不用一直查询键盘，鼠标，网卡等设备的状态，将精力集中在处理任务上。

讲了这么长，其实总结来说就是：要使用鼠标，就必须要使用中断。所以，我们必须设定IDT。IDT记录了0～255的中断号码与调用函数的对应关系，比如说发生了123号中断，就调用○×函数，其设定方法与GDT很相似（或许是因为使用同样的方法能简化CPU的电路）。

如果段的设定还没顺利完成就设定IDT的话，会比较麻烦，所以必须先进行GDT的设定。

■■■■■

虽然说明很长，但程序并没那么长。

本次的***bootpack.c**节选

```
struct SEGMENT_DESCRIPTOR{
short limit_low, base_low;
    char base_mid, access_right;
    char limit_high, base_high;
};

struct GATE_DESCRIPTOR {
    short offset_low, selector;

    char dw_count, access_right;
    short offset_high;
};

void init_gdtidt(void)
{
    struct SEGMENT_DESCRIPTOR *gdt = (struct SEGMENT_DESCRIPTOR *) 0x00270000;
    struct GATE_DESCRIPTOR    *idt = (struct GATE_DESCRIPTOR    *) 0x0026f800;
    int i;

    /* GDT的初始化 */
    for (i = 0; i < 8192; i++) {
        set_segmdesc(gdt + i, 0, 0, 0);
    }
    set_segmdesc(gdt + 1, 0xffffffff, 0x00000000, 0x4092);
    set_segmdesc(gdt + 2, 0x0007ffff, 0x00280000, 0x409a);
    load_gdtr(0xffff, 0x00270000);
```

```
    /* IDT的初始化 */
    for (i = 0; i < 256; i++) {
        set_gatedesc(idt + i, 0, 0, 0);
    }
    load_idtr(0x7ff, 0x0026f800);

    return;
}

void set_segmdesc(struct SEGMENT_DESCRIPTOR *sd, unsigned int limit, int base, int ar)
{
    if (limit > 0xfffff) {
        ar |= 0x8000; /* G_bit = 1 */
        limit /= 0x1000;
    }
    sd->limit_low    = limit & 0xffff;
    sd->base_low     = base & 0xffff;
    sd->base_mid     = (base >> 16) & 0xff;
    sd->access_right = ar & 0xff;
    sd->limit_high   = ((limit >> 16) & 0x0f) | ((ar >> 8) & 0xf0);
    sd->base_high    = (base >> 24) & 0xff;
    return;
}

void set_gatedesc(struct GATE_DESCRIPTOR *gd, int offset, int selector, int ar)
{
    gd->offset_low   = offset & 0xffff;
    gd->selector     = selector;
    gd->dw_count     = (ar >> 8) & 0xff;
    gd->access_right = ar & 0xff;
    gd->offset_high  = (offset >> 16) & 0xffff;
    return;
}
```

SEGMENT_DESCRIPTOR中存放GDT的8字节的内容，它无非是以CPU的资料为基础，写成了结构体的形式。同样，GATE_DESCRIPTOR中存放IDT的8字节的内容，也是以CPU的资料为基础的。

变量gdt被赋值0x00270000，就是说要将0x270000～0x27ffff设为GDT。至于为什么用这个地址，其实那只是笔者随便作出的决定，并没有特殊的意义。从内存分布图可以看出这一块地方并没有被使用。

变量idt也是一样，IDT被设为了0x26f800～0x26ffff。顺便说一下，0x280000～0x2fffff已经有了bootpack.h。“哎？什么时候？我可没听说过这事哦！”大家可能会有这样的疑问，其实是后面要讲到的“asmhead.nas”帮我们做了这样的处理。

■■■■■

现在继续往下说明。

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

请注意一下以上几行代码。gdt是0x270000，i从0开始，每次加1，直到8 191。这样一来，好像gdt+i最大也只能是0x271fff。但事实上并不是那样。C语言中进行指针的加法运算时，内部还隐含着乘法运算。变量gdt已经声明为指针，指向SEGMENT_DESCRIPTOR这样一个8字节的结构体，所以往gdt里加1，结果却是地址增加了8。

因此这个for 语句就完成了对所有8192个段的设定，将它们的上限（limit, 指段的字节数-1）、基址（base）、访问权限都设为0。

再往下还有这样的语句：

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

以上语句是对段号为1和2的两个段进行的设定。段号为1的段，上限值为0xffffffff即大小正好是4GB），地址是0，它表示的是CPU所能管理的全部内存本身。段的属性设为0x4092，它的含义我们留待明天再说。下面来看看段号为2的段，它的大小是512KB，地址是0x280000。这正好是为bootpack.hrb而准备的。用这个段，就可以执行bootpack.hrb。因为bootpack.hrb是以ORG 0为前提翻译成的机器语言。

■■■■■

下一个语句是：

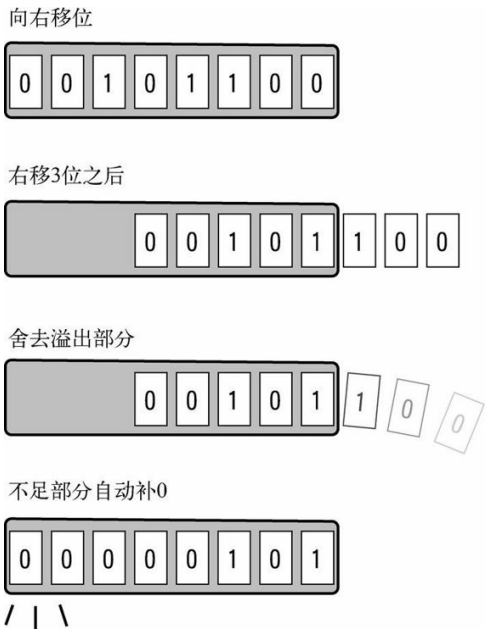
```
load_gdtr(0xffff, 0x00270000);
```

这是因为依照常规，C语言里不能给GDTR赋值，所以要借助汇编语言的力量，仅此而已。

再往下都是关于IDT的记述，因为跟前面一样，所以应该没什么问题。

在set_segmdesc和set_gatedesc中，使用了新的运算符，下面来介绍一下。首先看看语句“ar |= 0x8000;”，它是“ar = ar |0x8000;”的省略表现形式。同样还有“limit /= 0x1000;”，它是“limit = limit/0x1000;”的省略表现形式。“|”是前面已经出现的或（OR）运算符。“/”是除法运算符。

“>>”是右移位运算符。比如计算00101100>>3，就得到00000101。移位时，舍弃右边溢出的位，而左边不足的3位，要补3个0。



■■■■■

今天到这里就差不多了，访问权属性及IDT的详细说明就留到明天吧。总之，使用本程序的操作系统是做成了。能不能正常运行啊？赶紧试一试吧。“make run ”……还好，能运行。这次只是简单地做了初期设定，所以即使运行成功了，画面上也什么都不显示。

现在haribote.sys变成多少字节了呢？哦，光字体就有4KB，增加了不少，到7632字节了。今天就先到这里吧，大家明天见。