第7天

FIFO与鼠标控制

- 获取按键编码（hiarib04a）
- 加快中断处理（hiarib04b）
- 制作FIFO缓冲区（hiarib04c）
- 改善FIFO缓冲区（hiarib04d）
- 整理FIFO缓冲区（hiarib04e）
- 总算讲到鼠标了（hiarib04f）
- 从鼠标接收数据（hiarib04g）

1 获取按键编码（hiarib04a）

今天我们继续加油吧。鼠标不动的原因已经大体弄清楚了，主要是由于设定不到位。但是，在解决鼠标问题之前，还是先利用键盘多练练手，这样更易于鼠标问题的理解。

现在，只要在键盘上按一个键，就会在屏幕上显示出信息，其他的我们什么都做不了。我们将程序改善一下，让程序在按下一个键后不结束，而是把所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。

我们要修改的，是int.c程序中的inthandler21函数，具体如下：

int.c节选

```
#define PORT_KEYDAT     0x0060

void inthandler21(int *esp)
{
    struct BOOTINFO *binfo = (struct BOOTINFO *) ADR_BOOTINFO;
    unsigned char data, s[4];
    io_out8(PIC0_OCW2, 0x61);   /* 通知PIC"IRQ-01已经受理完毕" */
    data = io_in8(PORT_KEYDAT);

    sprintf(s, "%02X", data);
    boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 0, 16, 15, 31);
    putfonts8_asc(binfo->vram, binfo->scrnx, 0, 16, COL8_FFFFFF, s);
```

```
    return;
}
```

首先请把目光转移到“io_out8(PIC0_OCW2, 0x61);”这句话上。这句话用来通知PIC“已经知道发生了IRQ1中断哦”。如果是IRQ3，则写成0x63。也就是说，将“0x60+IRQ号码”输出给OCW2就可以。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。详情可参阅以下网页：

http://community.osdev.info/?(PIC) 8259A

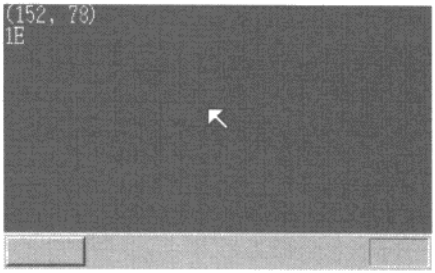
相关内容在最下面的“致偷懒者”（ものぐさなあなたのために）附近。

必须重启对于中断的监视

下面我们应该注意，从编号为0x0060的设备输入的8位信息是按键编码。编号为0x0060的设备就是键盘。为什么是0x0060呀？要想搞懂这个问题，还是得问IBM的大叔们这都是他们定的，笔者也不太清楚原因。不过这个号码是从下面这个网页查到的。

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

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。这里没什么难点……那好，我们运行一下“make run”。然后按下“A”键，哦，按键编码乖乖地显示出来了！



按下“A”之后

大家可以做各种尝试，比如按下“B”键，按下回车键等。键按下去之后，随即就会显示出一个数字（十六进制）来，键松开之后也会显示出一个数字。所以，计算机不光知道什么时候按下了键，还知道什么时候把键松开了。这种特性最适合于开发游戏了。不错不错，心满意足。

2 加快中断处理（hiarib04b）

程序做出来了，大家心情肯定很好，但其实这个程序里有一个问题，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，不再接受别的中断。所以如果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，这都是我们不希望看到的。

另一方面，字符显示是要花大块时间来进行的处理。仅仅画一个字符，就要执行8 × 16=128次if语句，来判定是否要往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，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正在等待键盘中断处理的结束。

那该如何是好呢？结论很简单，就是先将按键的编码接收下来，保存到变量里，然后由HariMain偶尔去查看这个变量。如果发现有了数据，就把它显示出来。我们就这样试试吧。

int.c节选

```
struct KEYBUF {
    unsigned char data, flag;
};

#define PORT_KEYDAT     0x0060

struct KEYBUF keybuf;

void inthandler21(int *esp)
{
    unsigned char data;
    io_out8(PIC0_OCW2, 0x61);    /* 通知PIC IRQ-01已经受理完毕 */
    data = io_in8(PORT_KEYDAT);
    if (keybuf.flag == 0) {
        keybuf.data = data;
        keybuf.flag = 1;
    }
    return;
}
```

我们先完成了上面的程序。考虑到键盘输入时需要缓冲区，我们定义了一个构造体，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是1，就表示缓冲区中存有数据。那么，如果缓冲区中存有数据，而这时又来了一个中断，那该怎么办呢？这没办法，我们暂时不做任何处理，权且把这个数据扔掉。

下面让我们看看bootpack.c的HariMain函数吧。我们对最后的io_halt里的无限循环进行了如下修改。

bootpack.c中HariMain函数的节选

```
    for (;;) {
        io_cli();
        if (keybuf.flag == 0) {
            io_stihlt();
        } else {
            i = keybuf.data;
            keybuf.flag = 0;
            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);
        }
    }
```

开始先用io_cli指令屏蔽中断。为什么这时要屏蔽中断呢？因为在执行其后的处理时，如果有中断进来，那可就乱套了。我们先将中断屏蔽掉，去看一看keybuf.flag的值是什么。

如果flag的值是0，就说明键还没有被按下，keybuf.data里没有值保存进来。在keybuf.data里有值被保存进来之前，我们无事可做，所以干脆就去执行io_hlt。但是，由于已经执行io_cli屏蔽了中断，如果就这样去执行HLT指令的话，即使有什么键被按下，程序也不会有任何反应。所以STI和HLT两个指令都要执行，而执行这两个指令的函数就是io_stihlt[①]。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以后，又回到for语句的开头，再执行io_cli函数。

继续往后读程序，我们能找到else语句。它一定要跟在if语句后面，意思是说只有在if语句中的条件不满足时，才能执行else后面花括号中的语句。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表示把键码值清为空，最后再通过io_sti语句开放中断。虽然如果在keybuf操作当中有中断进来会造成混乱，但现在keybuf.data的值已经保存完毕，再开放中断也就没关系了。最后，就可以在中断已经开放的情形下，优哉优哉地显示字符了。

回过头来看一看，可以发现，其实在屏蔽中断期间所做的处理非常少，中断处理程序本身做的事情也非常少，而这正是我们所期待的。真棒！如果我们坚持这么做，不但中断很少会被遗漏，而且最后完成的操作系统也会非常利索。

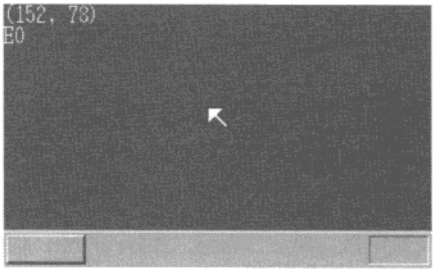
试验键盘

我们赶紧来测试一下吧。运行“make run”。哦，像以前一样，能够顺利执行……但是，发生了一点儿小问题。请按下键盘的右Ctrl键看看。不管是按下，还是松开，屏幕上显示的都是“E0”。哎?我们再试试harib04a，看看情况如何。结果按下去时显示“1D”，松开时显示“9D”。怎么回事？与harib04a结果不一样就意味着哪儿出了问题。

通过查资料[②]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，所以一次按键就会产生两次中断，第一次中断时发送E0，第二次中断时发送1D。

① 可能有人会认为，不做这个函数，而是用“io_sti();io_hlt();”不也行吗？但是，实际上这样写有点问题。如果io_sti()之后产生了中断，keybuf里就会存入数据，这时候让CPU进入HLT状态，keybuf里存入的数据就不会被觉察到。根据CPU的规范，机器语言的STI指令之后，如果紧跟着HLT指令，那么就暂不受理这两条指令之间的中断，而要等到HLT指令之后才受理，所以使用io_stihlt函数就能克服这一问题。

② http://community.osdev.info/?(AT)keyboard



按下右Ctrl键时的情形

在harib04a中，以上两次中断所发送的值都能收到，瞬间显示E0之后，紧接着又显示1D或是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。

这么一说，可能有人会觉得还是以前的harib04a更好。但是在harib04a中，键盘控制器（设备号码0x0060）是在“想去厕所，快要忍不住了”（=屏蔽中断的状态）的情况下，等待着程序的处理，才勉强得到这样看起来还不错的结果。但这对于硬件来讲，实在有点太勉为其难了。在harib04b程序中，硬件没有负担，不会憋得肚子疼，只是笔者这里写的程序还是不够好，好不容易接收到的数据，没能很好地利用起来。

所以，我们来修改一下程序，让它再聪明点儿。

3 制作 FIFO 缓冲区（hiarib04c）

问题到底出在哪儿呢？在于笔者所创建的缓冲区，它只能存储一个字节。如果做一个能够存储多字节的缓冲区，那么它就不会马上存满，这个问题也就解决了。

最简单的解决方案是像下面这样增加变量。

```
struct KEYBUF {
    unsigned char data1, data2, data3, data4, ...
};
```

但这样一来，程序就变长了，所以将它写成下面这样：

```
struct KEYBUF {
    unsigned char data[4];
};
```

当我们使用这些缓冲区的时候，可以写成data[0]、data[1]等。至于创建得是否正常，那就是后话了。

说起缓冲，我们在讲栈的时候，曾讲过FIFO、FILO等，这次我们需要的是FIFO型。为什么呢？如果输入的是ABC，输出的时候，却把顺序搞反了，写成CBA，那可就麻烦了。所以需要按照输入数据的顺序输出数据。

根据这种思路，我们制作了以下程序：

int.c节选

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

void inthandler21(int *esp)
{
    unsigned char data;
    io_out8(PIC0_OCW2, 0x61);    /* 通知PIC IRQ-01已经受理完毕 */
    data = io_in8(PORT_KEYDAT);
    if (keybuf.next < 32) {
        keybuf.data[keybuf.next] = data;
        keybuf.next++;
    }
    return;
}
```

keybuf.next的起始点是“0”，所以最初存储的数据是keybuf.data[0]。下一个数据是keybuf.data[1]，接着是[2]，依此类推，一共有32个存储位置。

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

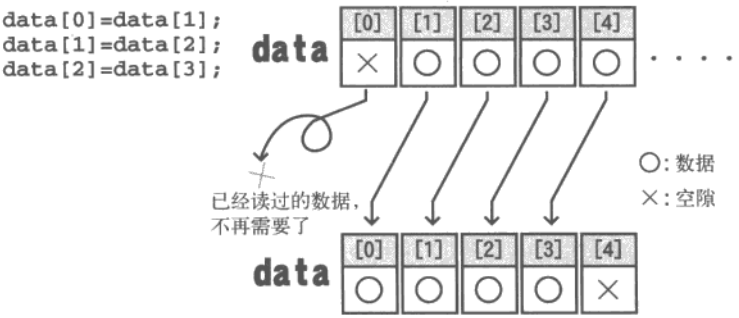
取得数据的程序如下所示。

```
    for (;;) {
        io_cli();
        if (keybuf.next == 0) {
            io_stihlt();
        } else {
            i = keybuf.data[0];
            keybuf.next--;
            for (j = 0; j < keybuf.next; j++) {
                keybuf.data[j] = keybuf.data[j + 1];
            }
            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);
```

```
        }
    }
```

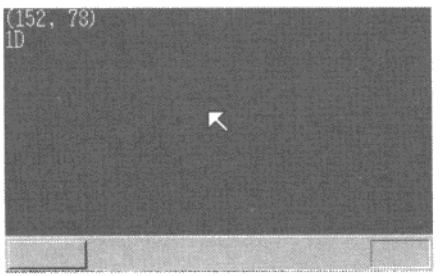
如果next不是0，则说明至少有一个数据。最开始的一个数据肯定是放在data[0]中的，将这个数存入到变量i中去。这样，数就减少了一个，所以将next减去1。

接下来的for语句，我们用下图来说明它所完成的工作。



像上面这样，数据的存放位置全部都向前移送了一个位置。如果不移送的话，下一次就不能从data[0]读入数据了。

那好，我们赶紧测试一下，看看能不能正常运行。“make run”，按下右Ctrl键，哦，运行正常！



按下右边Ctrl键以后的情形

虽然现在想说这个程序已经OK了，但实际上还是有问题。还有些地方还不尽如人意。inthandler21可以了，完全没有问题。有问题的是HariMain。说得具体一点，是从data[0]取得数据后有关数据移送的处理不能让人满意。

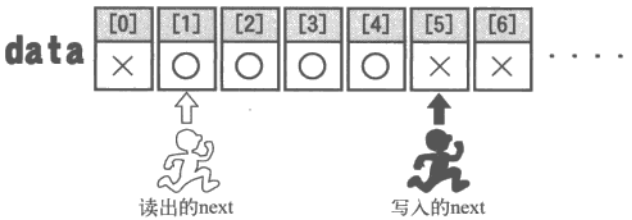
像这种移送数据的处理，一般说来也就不超过3个，基本上没有什么问题。但运气不好的时候，我们可能需要移送多达32个数据。虽然这远比显示字符所需的128个像素要少，但要是有办法避免这种操作的话，当然是最好不过了。

数据移送处理本身并没有什么不好，只是在禁止中断的期间里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。那该怎么办才好呢？接下来的harib04d章节就要讲述这个问题了。

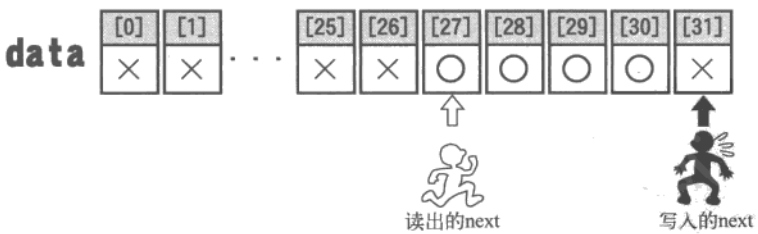
4 改善FIFO缓冲区（hiarib04d）

能不能开发一个不需要数据移送操作的FIFO型缓冲区呢？答案是可以的。因为我们有个技巧可以用。

这个技巧的基本思路是，不仅要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。这就好像数据读出位置在追着数据写入位置跑一样。这样做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。这种方式很好嘛！



但是这样的缓冲区使用了一段时间以后，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。

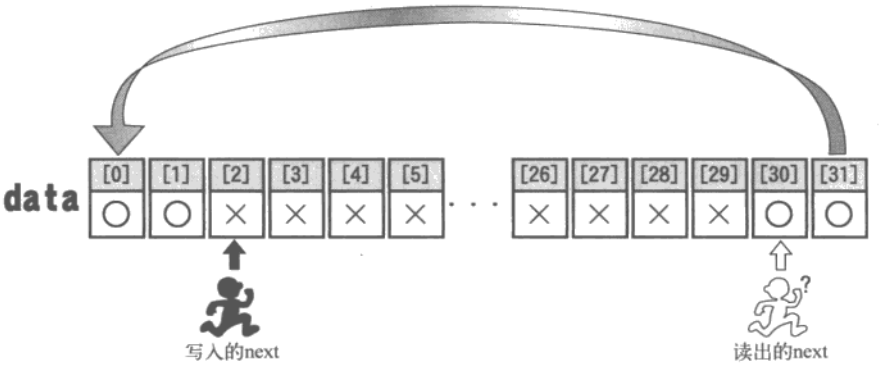


如果当下一个数据写入位置到达缓冲区终点时，数据读出位置也恰好到达缓冲区的终点，也就是说缓冲区正好变空，那还好说。我们只要将下一个数据写入位置和下一个数据读出位置都再置为0就行了，就像转回去从头再来一样。

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。原来每次都要进行数据移送，而现在不用每次都做，当然值得高兴，但问题是这样一来，用户会说：“有时候操作系统的反应不好。这系统不行啊。” 嗯，我们还是想尽可能避免所有的数据移送操作。

如果将缓冲区扩展到256字节，的确可以减少移位操作的次数，但是不能从根本上解决问题。

仔细想来，当下一个数据写入位置到达缓冲区最末尾时，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。打个比方，这就好像打开一张世界地图，一直向右走的话，会在环绕地球一周后，又从左边出来。这样一来，这个缓冲区虽然只有32字节，可只要不溢出的话，它就能够持续使用下去。

如果不是很理解以上说明的话，可以看看程序，一看就能会明白。

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);   /* 通知 IRQ-01已经受理完毕 */
    data = io_in8(PORT_KEYDAT);
    if (keybuf.len < 32) {
        keybuf.data[keybuf.next_w] = data;
        keybuf.len++;
        keybuf.next_w++;
        if (keybuf.next_w == 32) {
            keybuf.next_w = 0;
        }
    }
    return;
}
```

以上无非是将我们的说明写成了程序而已，并没什么难点。倒不如这样说，正是因为看了以上程序，大家才能搞清楚笔者想要说什么。读出数据的程序如下：

```
    for (;;) {
        io_cli();
        if (keybuf.len == 0) {
            io_stihlt();
        } else {
            i = keybuf.data[keybuf.next_r];
            keybuf.len--;
            keybuf.next_r++;
            if (keybuf.next_r == 32) {
                keybuf.next_r = 0;
            }
            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);
        }
    }
```

看吧，没有任何数据移送操作。这个缓冲区可以记录大量数据，执行速度又快，真是太棒啦。我们测试一下，运行“make run”，当然能正常运行。耶!

5 整理 FIFO 缓冲区（hiarib04e）

本来正说着键盘中断的话题，中间却插进来一大段关于FIFO缓冲区基本结构的介绍，不过既然这样，我们就来整理一下，让它具有一些通用性，在别的地方也能发挥作用。所谓“别的地方”，是指什么地方呢？当然是鼠标啦。鼠标只要稍微动一动，就会连续发送3个字节的数据。……事实上这次我们之所以先做键盘的程序，就是想拿键盘来练习一下FIFO和中断。因为如果一上来就做鼠标程序，数据来得太多，又要取出来进行处理，会让人手忙脚乱，不知所措。

首先我们将结构做成以下这样。

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

如果我们将缓冲区大小固定为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，几个字节都行。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。缓冲区的地址当然也必须保存下来，我们把它保存在变量buf里。p代表下一个数据写入地址（next_w），q代表下一个数据读出地址（next_r）。

fifo.c的fifo8_init函数

```
void fifo8_init(struct FIFO8 *fifo, int size, unsigned char *buf)
/* 初始化FIFO缓冲区 */
{
    fifo->size = size;
    fifo->buf = buf;
    fifo->free = size; /* 缓冲区的大小 */
    fifo->flags = 0;
    fifo->p = 0; /* 下一个数据写入位置 */
    fifo->q = 0; /* 下一个数据读出位置 */
    return;
}
```

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。更具体的说明就不用了吧。

fifo.c的fifo8_put函数

```
#define FLAGS_OVERRUN       0x0001

int fifo8_put(struct FIFO8 *fifo, unsigned char data)
/* 向FIFO传送数据并保存 */
{
    if (fifo->free == 0) {
        /* 空余没有了，溢出 */
        fifo->flags |= FLAGS_OVERRUN;
        return -1;
    }
    fifo->buf[fifo->p] = data;
    fifo->p++;
    if (fifo->p == fifo->size) {
        fifo->p = 0;
    }
    fifo->free--;
    return 0;
}
```

fifo8_put是往FIFO缓冲区存储1字节信息的函数。以前如果溢出了，就什么也不做。但这次，笔者想：“如果能够事后确认是否发生了溢出，不是更好吗？”所以就用flags这一变量来记录是

否溢出。至于其他内容，只是写法上稍有变化而已。

啊，要说这里出现的新东西，可能就是return语句后面跟着数字的写法了吧。如果有人调用以上函数，写出类似于“i = fifo8_put（fifo, data）;”这种语句时，我们就可以通过这种方式指定赋给i的值。为了能够简单明了地确认到底有没有发生溢出，笔者将它设定为-1或0，分别表示有溢出和没有溢出这两种情况。

fifo.c的fifo8_get函数

```
int fifo8_get(struct FIFO8 *fifo)
/* 从FIFO取得一个数据 */
{
    int data;
    if (fifo->free == fifo->size) {
        /* 如果缓冲区为空，则返回 -1 */
        return -1;
    }
    data = fifo->buf[fifo->q];
    fifo->q++;
    if (fifo->q == fifo->size) {
        fifo->q = 0;
    }
    fifo->free++;
    return data;
}
```

fifo8_get是从FIFO缓冲区取出1字节的函数。这个应该不用再讲了吧。

fifo.c的fifo8_status函数

```
int fifo8_status(struct FIFO8 *fifo)
/* 报告一下到底积攒了多少数据 */
{
    return fifo->size - fifo->free;
}
```

这是附赠的函数fifo8_status，它能够用来调查缓冲区的状态。status的意思是“状态”。

笔者把以上这几个函数总结后写在了程序fifo.c里。

使用以上函数写成了下面的程序段。

int.c节选

```
struct FIFO8 keyfifo;

void inthandler21(int *esp)
{
```

```
    unsigned char data;
    io_out8(PIC0_OCW2, 0x61);    /* 通知PIC，说IRQ-01的受理已经完成 */
    data = io_in8(PORT_KEYDAT);
    fifo8_put(&keyfifo, data);
    return;
}
```

这段程序看起来非常清晰，12行变成了5行。在fifo8_put的参数里，有一个“&”符号，这可不是AND运算符，而是取地址运算符，用它可以取得结构或变量的地址值。变量名的前面加上&，就成了取地址运算符。这稍微有点复杂。fifo8_put接收的第一个参数是内存地址，与之匹配，这里调用时传递的第一个参数也要是内存地址。

MariMain函数内容如下所示：

```
    char s[40], mcursor[256], keybuf[32];

    fifo8_init(&keyfifo, 32, keybuf);

    for (;;) {
        io_cli();
        if (fifo8_status(&keyfifo) == 0) {
            io_stihlt();
        } else {
            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);
        }
    }
```

这段程序简洁而清晰。for语句的内容被精简掉了5行呀。当然，程序运行肯定也没问题。不信的话，可以用“make run”测试一下（当然，信的话更要试试啦）。看，运行正常！

6 总算讲到鼠标了（harib04f）

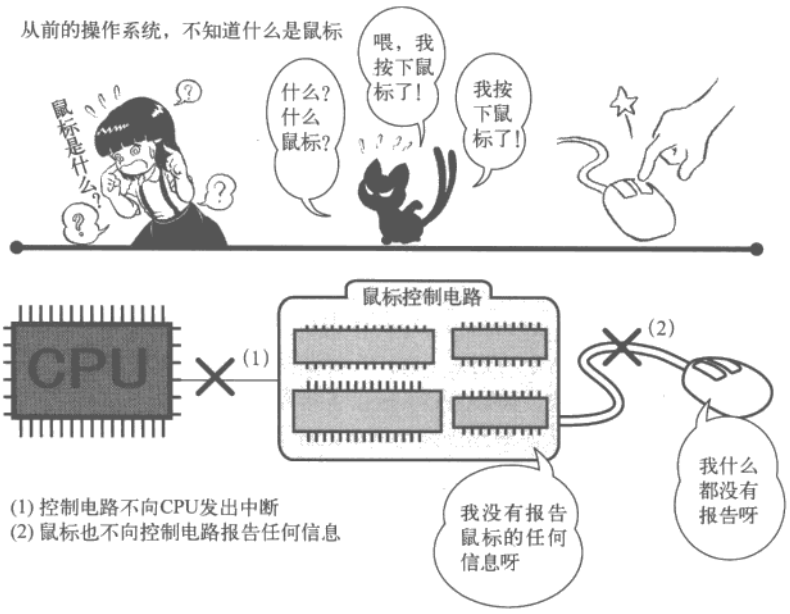
现在到了让大家期待已久的讲解鼠标的时间了。首先说一说，为什么虽然我们的电脑连着有鼠标，却一直不能用的原因。

从计算机不算短暂的历史来看，鼠标这种装置属于新兴一族。早期的计算机一般都不配置鼠标。一个很明显的证据就是，现在我们要讲的分配给鼠标的中断号码，是IRQ12，这已经是一个很大的数字了。与键盘的IRQ1比起来，那可差了好多代了。

所以，当鼠标刚刚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，如果只是稍微动一动鼠标就产生中断的话，那在使用那些操作系统的时候，就只好先把鼠标拔掉了。IBM的大叔们认为，这对于使用计算机的人来说是很不方便的。所以，

虽然在主板上做了鼠标用的电路，但只要不执行激活鼠标的指令，就不产生鼠标的中断信号(1)。

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，鼠标也就没必要送数据了，否则倒会引起电路的混乱。所以，处于初期状态的鼠标，不管是滑动操作也好，点击操作也好，都没有反应(2)。



总而言之，我们必须发行指令，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。通过上面的说明，大家应该已经明白了，要先让鼠标控制电路有效。如果先让鼠标有效了，那时控制电路还没准备好数据就来了，可就麻烦了，因为控制电路还处理不了。

现在来说说控制电路的设定。事实上，鼠标控制电路包含在键盘控制电路里，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。

bootpack.c节选

```
#define PORT_KEYDAT             0x0060
#define PORT_KEYSTA             0x0064
#define PORT_KEYCMD             0x0064
#define KEYSTA_SEND_NOTREADY    0x02
#define KEYCMD_WRITE_MODE       0x60
#define KBC_MODE                0x47

void wait_KBC_sendready(void)
{
```

```
    /* 等待键盘控制电路准备完毕 */
    for (;;) {
        if ((io_in8(PORT_KEYSTA) & KEYSTA_SEND_NOTREADY) == 0) {
            break;
        }
    }
    return;
}

void init_keyboard(void)
{
    /* 初始化键盘控制电路 */
    wait_KBC_sendready();
    io_out8(PORT_KEYCMD, KEYCMD_WRITE_MODE);
    wait_KBC_sendready();
    io_out8(PORT_KEYDAT, KBC_MODE);
    return;
}
```

首先我们来看函数wait_KBC_sendready。它的作用是，让键盘控制电路（keyboard controller, KBC）做好准备动作，等待控制指令的到来。为什么要做这个工作呢？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。

break语句是从for循环中强制退出的语句。退出以后，只有return语句在那里等待执行，所以，把这里的break语句换写成return语句，结果一样。

下面看函数init_keyboard。它所要完成的工作很简单，也就是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，当然这些数值必须通过调查才能知道。我们可以从老地方[①]得到这些数据。

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

现在，我们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。

bootpack.c节选

```
#define KEYCMD_SENDTO_MOUSE     0xd4
#define MOUSECMD_ENABLE         0xf4
```

① http://community.osdev.info/?ifno(AT)keyboard

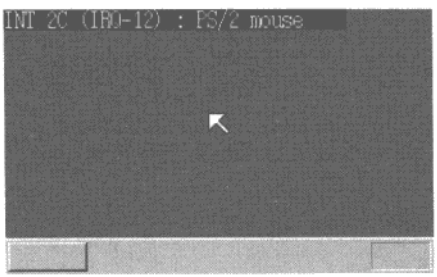
```
void enable_mouse(void)
{
    /* 激活鼠标 */
    wait_KBC_sendready();
    io_out8(PORT_KEYCMD, KEYCMD_SENDTO_MOUSE);
    wait_KBC_sendready();
    io_out8(PORT_KEYDAT, MOUSECMD_ENABLE);
    return; /* 顺利的话，键盘控制其会返送回ACK(0xfa)*/
}
```

这个函数与init_keyboard函数非常相似。不同点仅在于写入的数据不同。如果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。我们根据这一特性来发送激活鼠标的指令。

另一方面，一直等着机会露脸的鼠标先生，收到激活指令以后，马上就给CPU发送答复信息："OK，从现在开始就要不停地发送鼠标信息了，拜托了。"这个答复信息就是0xfa。

因为这个数据马上就跟着来了，即使我们保持鼠标完全不动，也一定会产生一个鼠标中断。

所以，我们将enable_mouse也做成了从HariMain中调用的形式。好，我们马上测试一下。运行"make run"。



鼠标中断终于来了

鼠标中断终于出来了。这可是很了不起的进步哟。

7 从鼠标接受数据（harib04g）

既然中断已经来了，现在就让我们取出中断数据吧。前面已经说过，鼠标和键盘的原理几乎相同，所以程序也就非常相似。

int.c节选

```
struct FIFO8 mousefifo;
void inthandler2c(int *esp)
```

```
/* 来自PS/2鼠标的中断 */
{
    unsigned char data;
    io_out8(PIC1_OCW2, 0x64);    /* 通知PIC1 IRQ-12的受理已经完成 */
    io_out8(PIC0_OCW2, 0x62);    /* 通知PIC0 IRQ-02的受理已经完成 */
    data = io_in8(PORT_KEYDAT);
    fifo8_put(&mousefifo, data);
    return;
}
```

不同之处只有送给PIC的中断受理通知。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。这不是笔者的失误，而是事实。也许是因为键盘控制电路中含有鼠标控制电路，才造成了这种结果。至于传到这个设备的数据，究竟是来自键盘还是鼠标，要靠中断号码来区分。

取得数据的程序如下所示：

bootpack.c节选

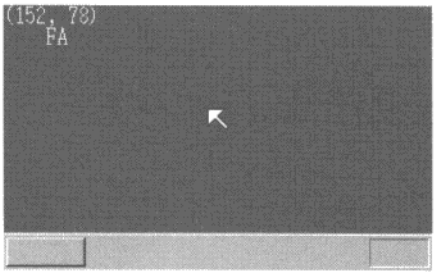
```
    fifo8_init(&mousefifo, 128, mousebuf);

    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();
                sprintf(s, "%02X", i);
                boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 32, 16, 47, 31);
                putfonts8_asc(binfo->vram, binfo->scrnx, 32, 16, COL8_FFFFFF, s);
            }
        }
    }
```

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。很简单吧。

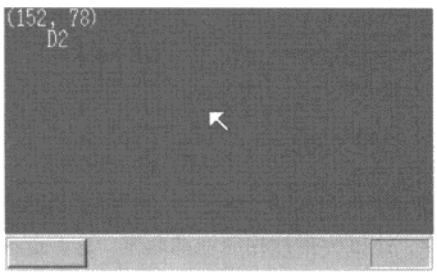
到底能不能执行呢？好紧张呀。我们来测试一下。运行“make run”。



启动刚完成时

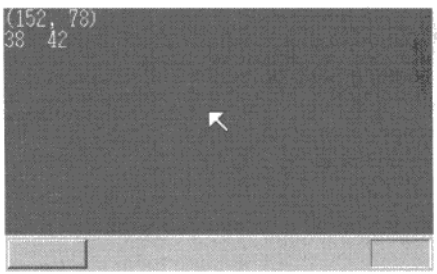
就像上面那样，最初只显示鼠标发送过来的数据，且内容的确是FA。

随便滚动鼠标一下，就会像下面这样显示出各种各样的数据来。



滚动鼠标

如果按下键盘，当然会像以前一样，正常响应。



按下键盘之后

看，运行得很正常很不错呀。

好了，今天我们做的事已经不少了，就先到这吧。明天我们来解读鼠标数据，让鼠标指针在屏幕上动起来。真期待呀。啊，今天就不要再往下读了哦。先睡觉，明天再继续，好吧？