Day 6

Phase 1

SubPhase 1

分割源文件。这里分割源文件的主要方法和我们之前学习c和c++的不太一样：我们暂时还没有使用include语句，我们只是把他们拆分开来，然后编译的时候各自生成目标文件，最后再链接到一起去。

SubPhase 2

Makefile支持使用%作为通配符，因此我们使用%可以将重复相近的生成规则缩成一条来写，以免形成冗长的难以阅读和调试的Makefile。

> https://seisman.github.io/how-to-write-makefile 这是一篇很好的教授如何写makefile的教程

SubPhase 3

整理头文件，我们将定义放在`bootpack.h`头文件中。

其实Phase 1就是简单的拆分一下文件，学习过c语言多文件组织并不是什么难事，这里也没有什么值得多说的地方。

不过我想补充一下，我鼓捣出了一个makefile的修改方法，可以替代之前出错的copy指令

```
haribote.sys : asmhead.bin bootpack.hrb Makefile
    copy /B asmhead.bin+bootpack.hrb haribote.sys
# 替换成
haribote.sys : asmhead.bin bootpack.hrb Makefile
    cmd /c"copy /B asmhead.bin+bootpack.hrb haribote.sys"
```

```
run :
    $(MAKE) img
    $(COPY) haribote.img ..\z_tools\qemu\fdimage0.bin
    $(MAKE) -C ../z_tools/qemu
# 替换成
run :
    $(MAKE) img
    cmd /c"cp haribote.img ..\z_tools\qemu\fdimage0.bin"
    $(MAKE) -C ../z_tools/qemu
```

但实际上为何之前的会出错，这样不会出错，原因不得而知。我觉得应该是make的版本原因。

Phase 2

SubPhase 1

之前我们并不理解`load_gdtr`，今天我们根据书上的解释来理解一波。

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

`lgdt`指令接受一个内存地址，从那里读取6个字节，然后赋值给gdtr寄存器。这6个字节中低16位将作为段上限，高32位将作为gdt的开始地址。

我们要组合出来这6个字节：先将limit存入16位寄存器AX（高位将被丢掉），然后将AX的内容放到ESP+6的位置，这16位的内容和参数addr所处的内存位置正好是我们要的6个字节。于是直接以ESP+6的位置作为lgdt的参数咯。

SubPhase 2

段设置

段上限中一个比较有趣的设置位是Gbit。由于设计上的一些限制，以字节为单位最多只能指定1MB的段大小，设置Gbit后，limit将被解释为页。由于一页是4KB，所以段最多可以指定到4GB了。

段访问权属性

CPU到底是处于系统模式还是应用模式，取决于执行中的应用程序是位于访问权为0x9a的 段，还是位于访问权为0xfa的段。

Phase 3

PIC是programmable interrupt controller，意思是可编程中断控制器。CPU只能单独处理一个中断，这无法满足众多的低速外设的中断需求。所以使用PIC将众多中断信号合成一个中断信号。一个PIC可以监听8个信号，使用两个PIC，其中一个直接与CPU相连，另一个与第一个PIC相连。与CPU直接相连的PIC叫做主PIC，另一个PIC叫做从PIC，这样总共能处理15个中断信号了。

很多设定是要查看PIC文档才能知道如何设置的，我们就先暂时囫囵吞枣一下吧。

从PIC接在主PIC的IRQ2口上。

SubPhase 1

```
void init_pic(void)
/* PICの初期化 */
{
    io_out8(PIC0_IMR, 0xff ); /* 禁止所有中断 */
    io_out8(PIC1_IMR, 0xff ); /* 禁止所有中断 */
    io_out8(PIC0_ICW1, 0x11 ); /* 边沿触发模式 (edge trigger mode)  */
    io_out8(PIC0_ICW2, 0x20 ); /* IRQ0-7由INT20-27接收 */
    io_out8(PIC0_ICW3, 1 << 2); /* PIC1由IRQ2连接 */
    io_out8(PIC0_ICW4, 0x01 ); /* 无缓冲区模式 */
    io_out8(PIC1_ICW1, 0x11 ); /* 边沿触发模式 (edge trigger mode)  */
    io_out8(PIC1_ICW2, 0x28 ); /* IRQ8-15由INT28-2f接收 */
    io_out8(PIC1_ICW3, 2 ); /* PIC1由IRQ2连接 */
    io_out8(PIC1_ICW4, 0x01 ); /* 无缓冲区模式 */
    io_out8(PIC0_IMR, 0xfb ); /* 11111011 PIC1以外全部禁止 */
    io_out8(PIC1_IMR, 0xff ); /* 11111111 禁止所有中断 */
```

```
    return;
}
```

我们一会将要编写0x2c和0x21的中断handler，来实现对鼠标和键盘的响应。

SubPhase 2

中断的处理不能只靠C语言进行，因为要进行中断返回（iretd）。这个指令是C语言无法直接调用的，所以我们还是得用nask来写，但我们还是想用c语言怎么办呢？我们可以先写c语言程序，然后在nask程序中调用它。这样我们就可以兼得nask和c语言的好处了！

```
void inthandler21(int *esp)
/* 来自PS/2键盘的中断 */
{
    struct BOOTINFO *binfo = (struct BOOTINFO *) ADR_BOOTINFO;
    boxfill8(binfo->vram, binfo->scrnx, COL8_000000, 0, 0, 32 * 8 - 1, 15);
    putfonts8_asc(binfo->vram, binfo->scrnx, 0, 0, COL8_FFFFFF, "INT 21 (IRQ-1) : PS/2
keyboard");
    for (;;) {
        io_hlt();
    }
}
```

```
    EXTERN _inthandler21, _inthandler2c
_asm_inthandler21:
    PUSH ES
    PUSH DS
    PUSHAD
    MOV EAX,ESP
    PUSH EAX
    MOV AX,SS
    MOV DS,AX
    MOV ES,AX
    CALL _inthandler21
    POP EAX
    POPAD
    POP DS
    POP ES
    IRETD
```

其中`pushad`相当于

```
PUSH EAX
PUSH ECX
PUSH EDX
PUSH EBX
PUSH ESP
PUSH EBP
PUSH ESI
PUSH EDI
```

而`popad`相当于

```
POP EDI
POP ESI
POP EBP
POP ESP
POP EBX
POP EDX
POP ECX
POP EAX
```

这两个指令相当于保存和回复CPU现场。

对了，PUSH和POP是栈的概念，这个我们早就在数据结构等相关课程上学习过了，书上的看一下也就明白了

```
MOV AX,SS
MOV DS,AX
MOV ES,AX
```

这三个命令是因为C语言自以为是地认为“DS也好，ES也好，SS也好，它们都是指同一个段”，所以如果不按照它的想法设定的话，函数inthandler21就不能顺利执行。

既然改变了DS和ES的值，那么我们在改变之前也应该保存现场和恢复现场。

因此我们有

```
push es
push ds
; ---------
pop ds
pop es
```

然后我们再按照相同方法搞一下0x2c中断服务程序（鼠标）就可以了。

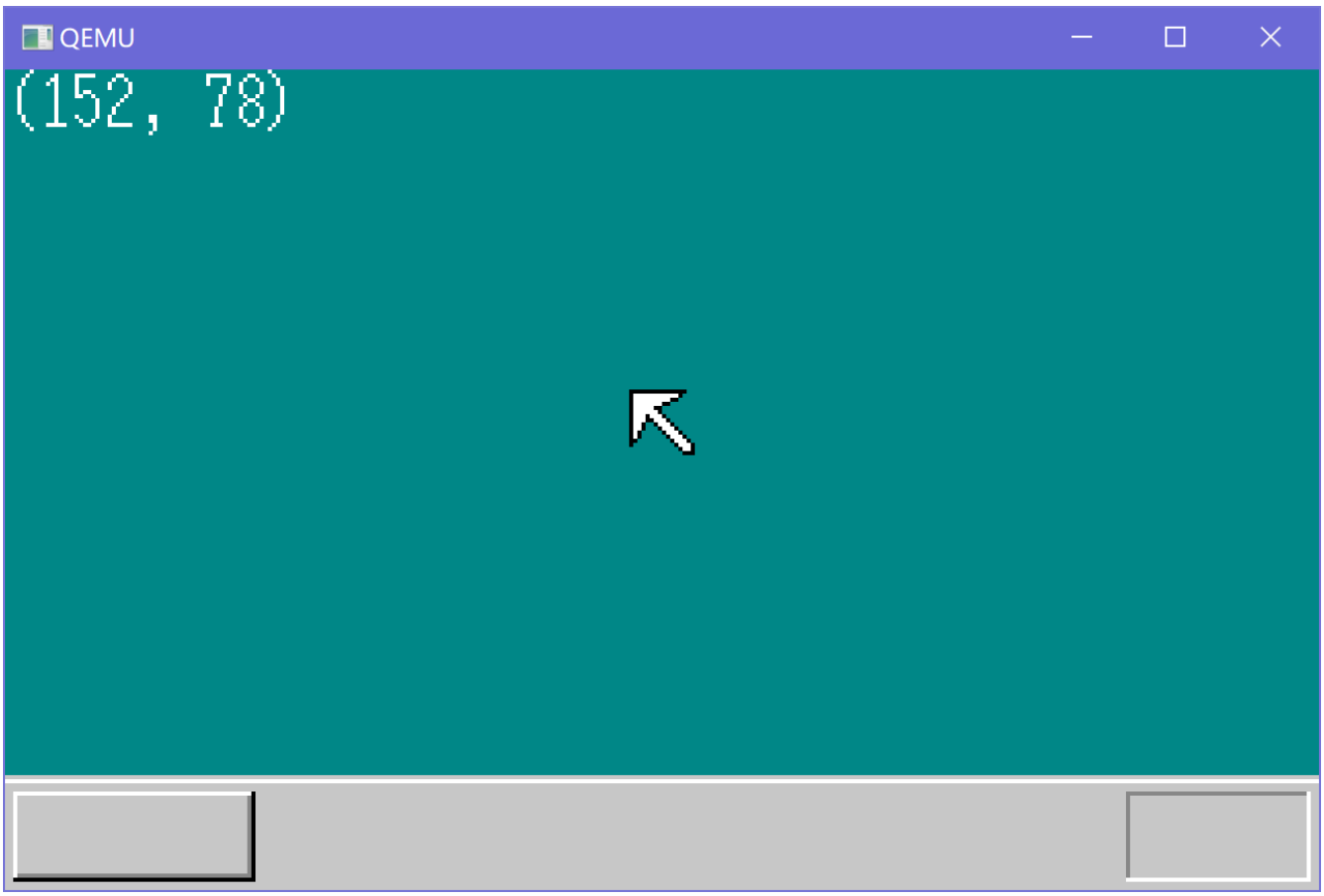
SubPhase 3

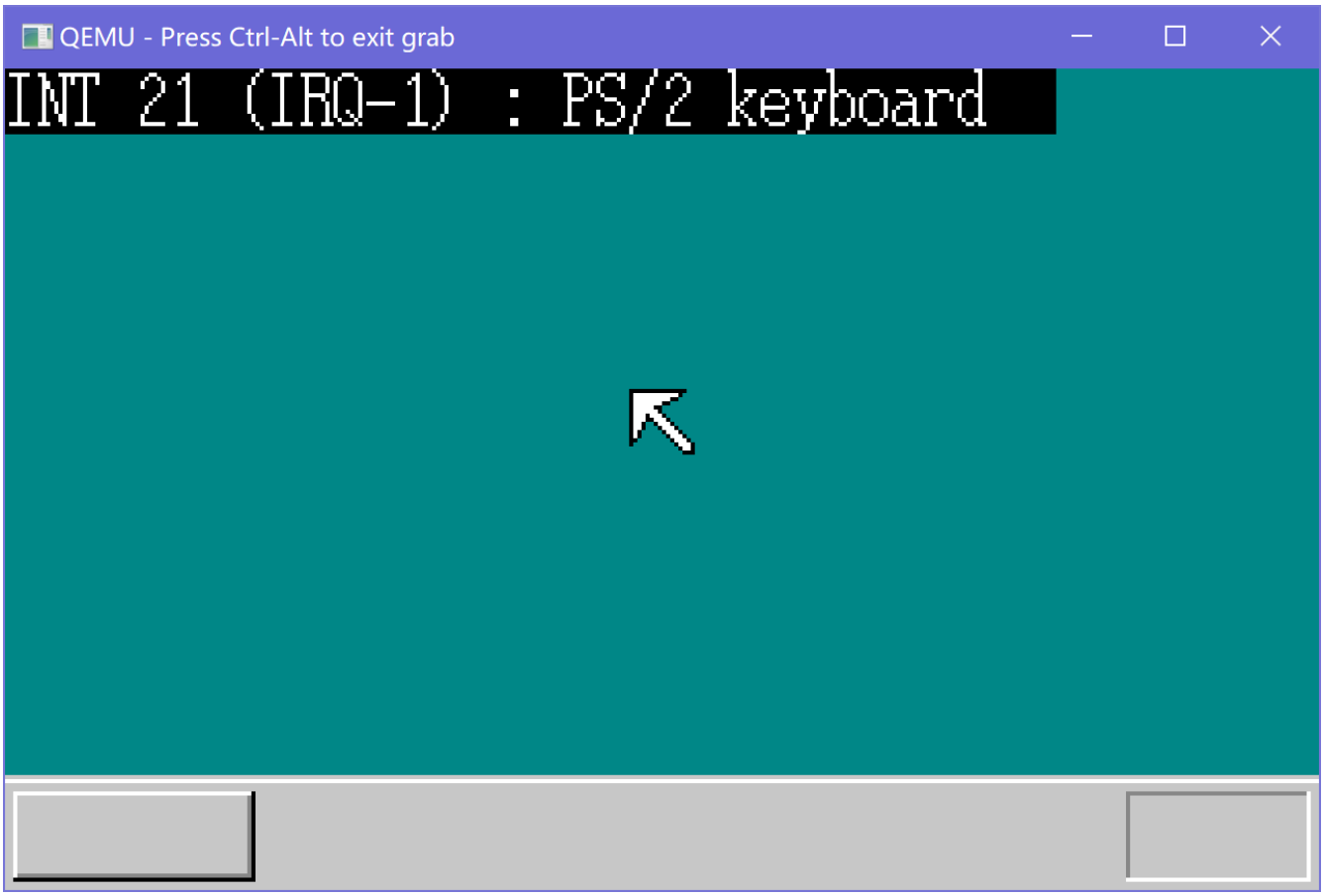
然后我们就可以注册中断处理程序啦

```
set_gatedesc(idt + 0x21, (int) asm_inthandler21, 2 * 8, AR_INTGATE32);
set_gatedesc(idt + 0x2c, (int) asm_inthandler2c, 2 * 8, AR_INTGATE32);
```

2的意思是段号是2，乘8的原因是低3位有别的意思。

然后我们就来试一试吧！





鼠标的中断似乎还不能正常处理。今天就先到这里了。

Day 7

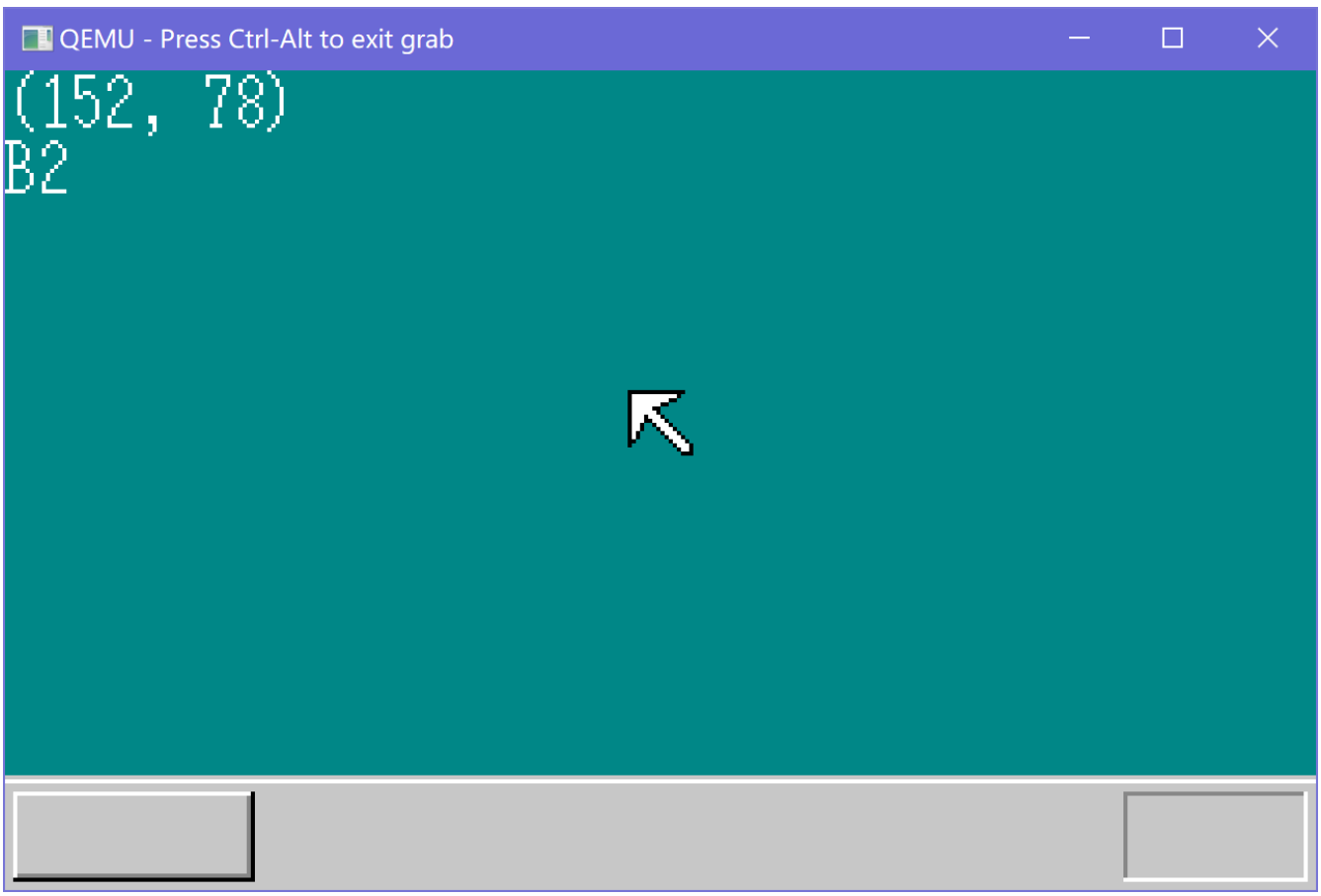
Phase 1

SubPhase 1

获取按键编码。为了获取按键编码，我们需要使用`io_in8`函数从针脚读入键盘提供的数据。读入地址是0x0060

我们要先通知PIC程序已知悉IRQ1中断（相当于重启监视），然后再从0x0060设备获取数据。代码如下

```
void inthandler21(int *esp) {
    struct BOOTINFO *binfo = (struct BOOTINFO *) ADR_BOOTINFO;
    unsigned char data, s[4];
    io_out8(PIC0_OCW2, 0x61);   /* IRQ-01受付完了をPICに通知 */
    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;
}
```



按下`m`键的结果

SubPhase 2

使用队列来处理键盘事件

对于已经学习过数据结构的我们，在只是用数组和整形变量的情况下手撸出来一个循环队列是一个再简单不过的事情了。

```
struct FIFO8 {
    int head, tail;
    char data[QSIZE];
};

void fifo8_init(struct queue *q) {
    q->data[q->head = q->tail = 0] = 0;
}

void fifo8_status(struct queue *q) {
    return (q->tail + QSIZE - q->head) % QSIZE;
}

int fifo8_put(struct queue *q, char dta) {
    if (qsize(q) >= QSIZE - 1) return -1;
    q->data[q->tail] = dta;
    if (++q->tail >= QSIZE) q->tail = 0;
    return 0;
}

int fifo8_get(struct queue *q, char *dta) {
    if (qsize(q) <= 0) return -1;
    *dta = q->data[q->head];
    if (++q->head >= QSIZE) q->head = 0;
    return 0;
}
```

我们稍微修改一下`inthandler21`和主函数的循环就可以在保证不丢数据的情况下加快中断的响应了。

```
// partial content of "int.c"
void inthandler2c(int *esp) {
    struct BOOTINFO *binfo = (struct BOOTINFO *) ADR_BOOTINFO;
    boxfill8(binfo->vram, binfo->scrnx, COL8_000000, 0, 0, 32 * 8 - 1, 15);
    putfonts8_asc(binfo->vram, binfo->scrnx, 0, 0, COL8_FFFFFF, "INT 2C (IRQ-12) : PS/2
mouse");
    for (;;) {
        io_hlt();
    }
}

// partial content of function HariMain in "bootpack.c"
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);
        }
    }
```

注意我们在对队列读取数据之前用`io_cli`暂时关闭了中断，这是因为我们不希望在我们读取队列的时候因为外部中断的打断再次改变队列的内容。这样会造成混乱。我们在读取完队列的内容之后再恢复允许中断。

测试，发现运行正常。但截图上无法体现，故不再重复贴图。

Phase 2

SubPhase 1

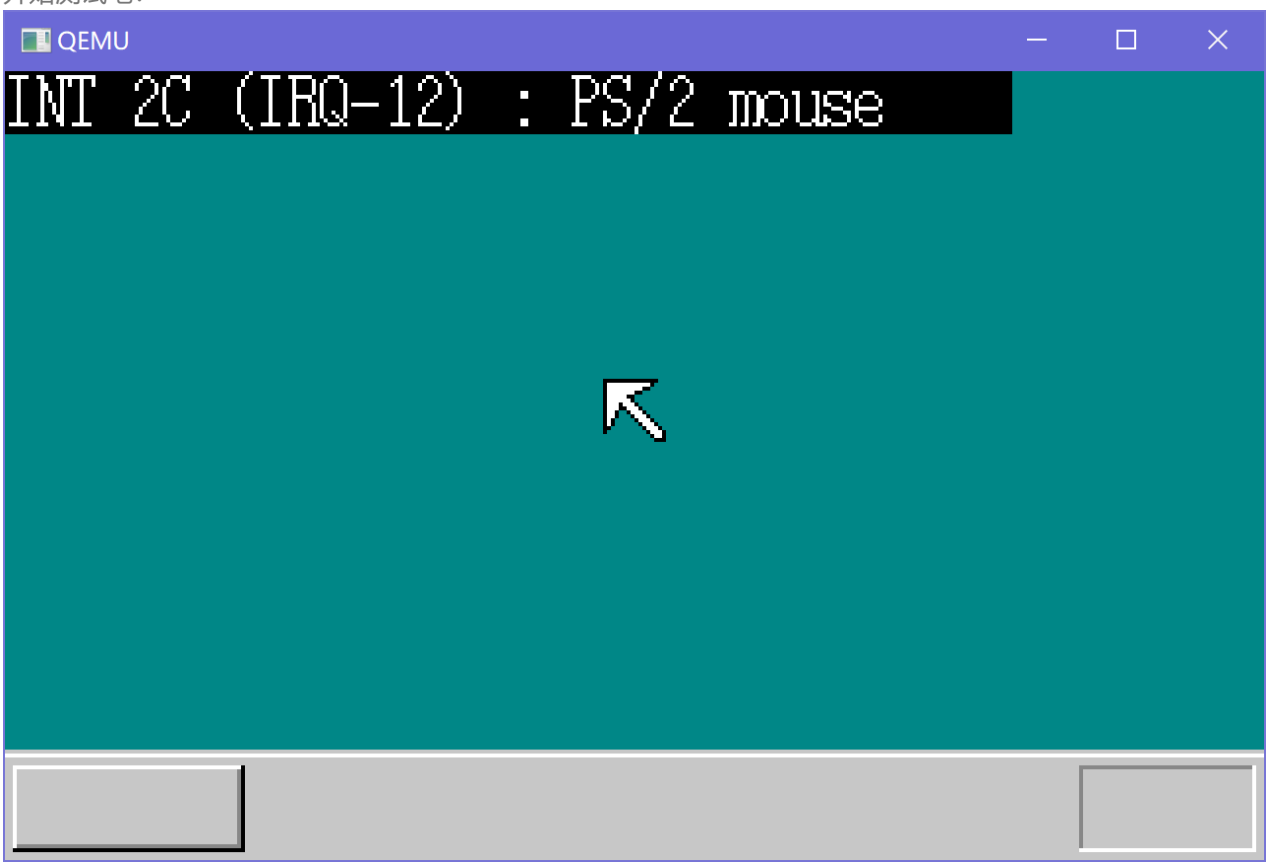
由于历史原因，当鼠标只是接入到机器上之后并不会马上工作，计算机必须先激活鼠标控制电路，然后激活鼠标。完成这些工作之后鼠标才会开始工作，鼠标控制电路才会送入中断和数据。

```
#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) {
    for (;;) {
        if ((io_in8(PORT_KEYSTA) & KEYSTA_SEND_NOTREADY) == 0) {
            break;
        }
    }
    return;
}

void init_keyboard(void) {
    wait_KBC_sendready();
    io_out8(PORT_KEYCMD, KEYCMD_WRITE_MODE);
    wait_KBC_sendready();
    io_out8(PORT_KEYDAT, KBC_MODE);
    return;
}

#define KEYCMD_SENDTO_MOUSE     0xd4
#define MOUSECMD_ENABLE         0xf4

void enable_mouse(void) {
    wait_KBC_sendready();
```

```
    io_out8(PORT_KEYCMD, KEYCMD_SENDTO_MOUSE);
    wait_KBC_sendready();
    io_out8(PORT_KEYDAT, MOUSECMD_ENABLE);
    return;
}
```

`wait_KBC_sendready`这个函数是干什么的呢？他是让键盘控制电路做好准备动作，等待控制指令的到来。其中的道理是，键盘控制电路速度相比CPU比较慢，如果我们在它没有准备好的时候就给他发送指令，就无法保证指令能够执行，从而出现错误，得不到我们想要的结果。

> 书上小错误好多啊
>
> 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。
>
> 这里应该是`0xf4`吧?

> 开始测试吧!
>
> 
>

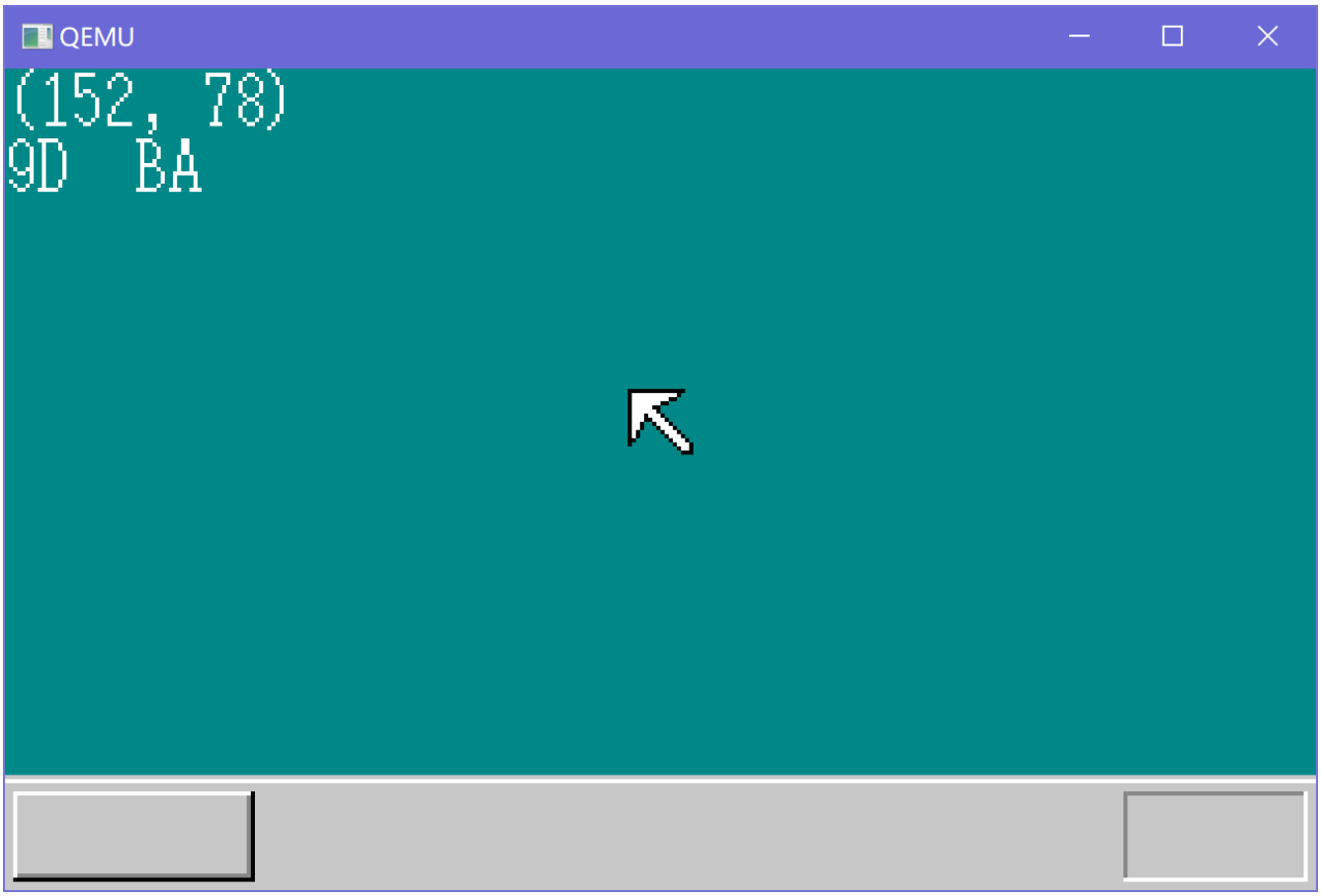
SubPhase 2

从鼠标接收数据的方法和从键盘接收数据的方法大同小异。不同之处是要先通知从PIC中断已经处理，然后再通知主PIC。原因是两个PIC之间的协调不能自动完成，需要CPU来完成。

```
// partial content of "int.c"
void inthandler2c(int *esp) {
    unsigned char data;
    io_out8(PIC1_OCW2, 0x64);
    io_out8(PIC0_OCW2, 0x62);
```

```
    data = io_in8(PORT_KEYDAT);
    fifo8_put(&mousefifo, data);
    return;
}
// partial content of function HariMain in "bootpack.c"
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);
        }
    }
}
```

测试运行一下吧！



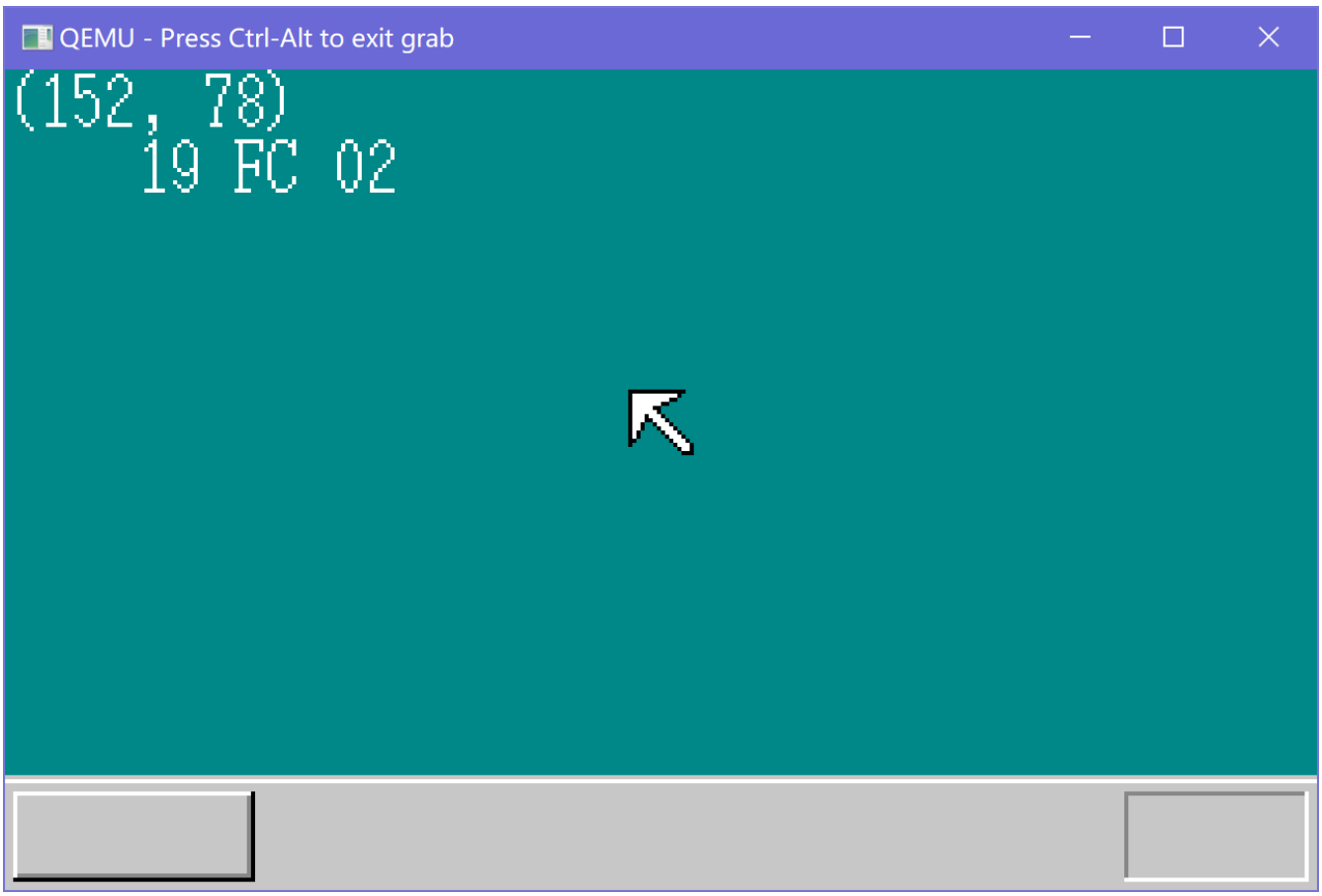
工作正常。今天就先这样吧。

Day 8

Phase 1

SubPhase 1

鼠标发送的数据是三个字节一组的，在连续的若干组前有前导的0xfa。我们先让他一次显示一组的三个字节的数据。程序很好修改，我们只需要记录已经得到了几个字节并且将他们暂时存储起来，存满三个字节然后再打印吧。



按下鼠标左键移动鼠标有如上结果。

SubPhase 2

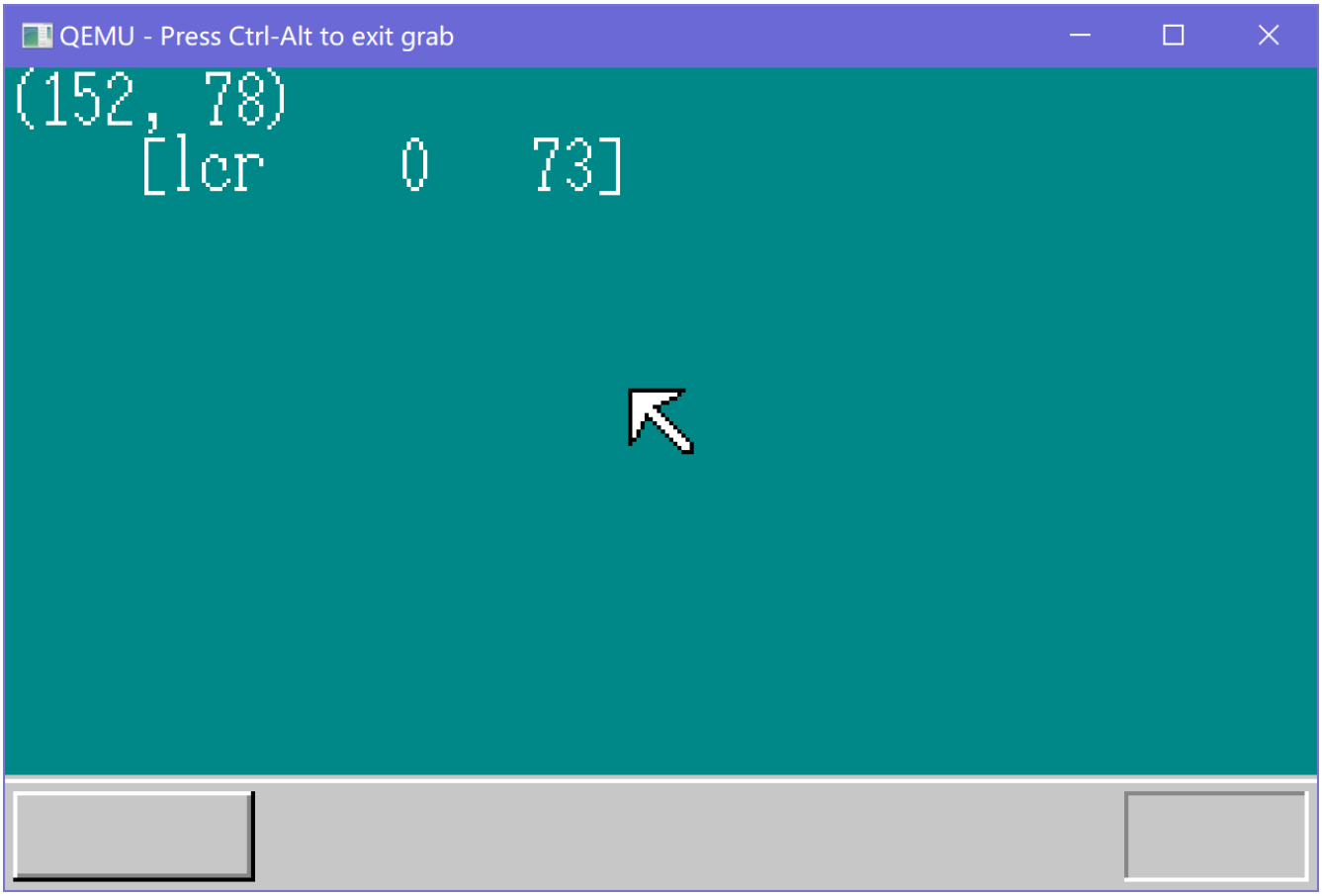
由于鼠标偶尔会断线，断线之后会重新发送0xfa数据。这将使我们之后的每一个数据都产生错位，从而使得鼠标无法正常工作。由于第一字节对移动的反应和对点击的反应都是分别限定在0~3和8~F的范围内的，我们可以以此为依据，适当的扔掉几个字节，这样就可以重新对齐了。

```
if (mdec->phase == 1) {
    if ((dat & 0xc8) == 0x08) { // 1100 1000 使用位运算提高效率
        mdec->buf[0] = dat;
        mdec->phase = 2;
    }
    return 0;
}
```

第一字节低三位表示按键的状态，第0位是左键，第1位是右键，第2位是鼠标中键。

第二字节和第三字节都是有符号数，代表鼠标移动的速度。

我们把处理好的数据以可读的方式打印到屏幕上吧！

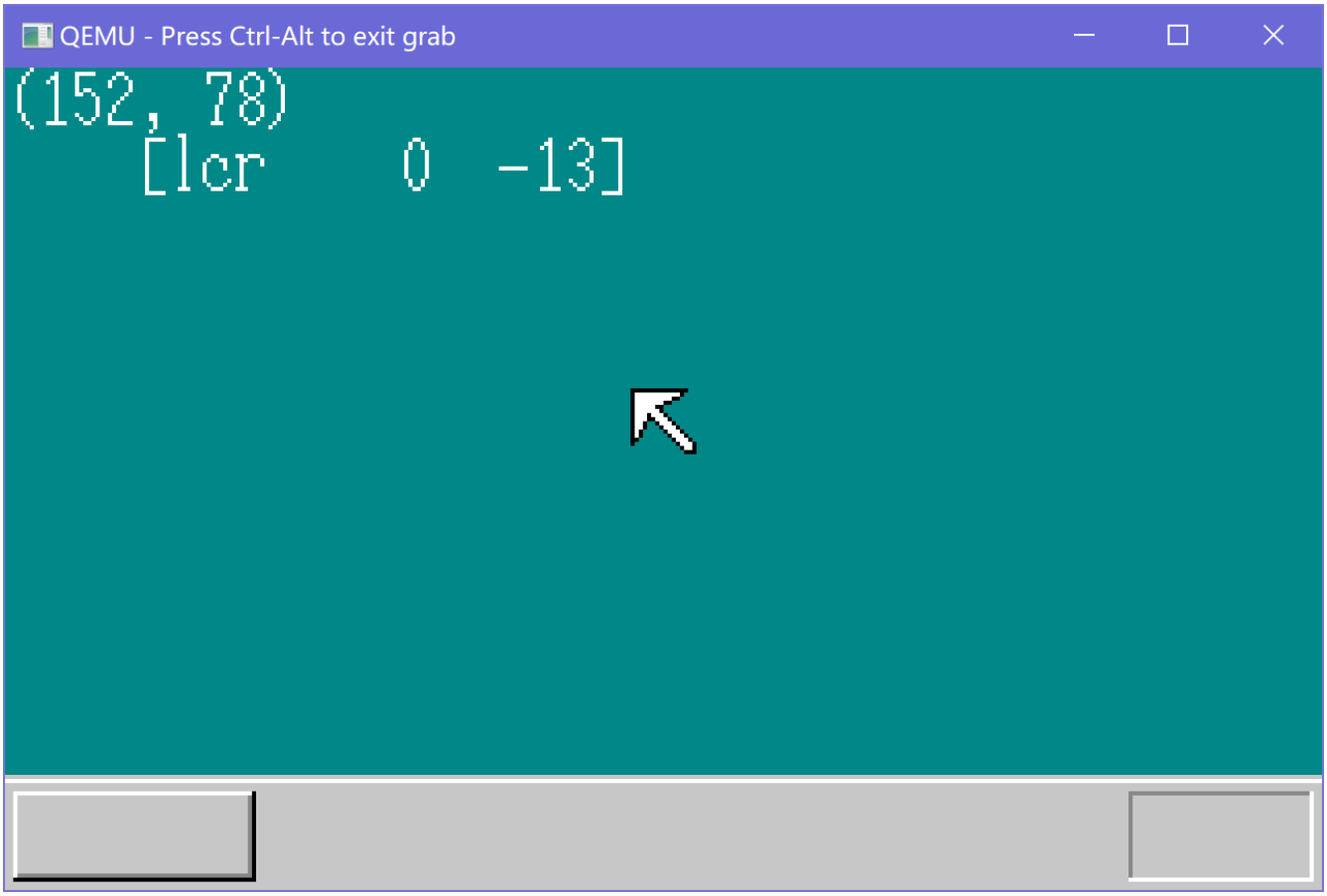


鼠标快速向上移动的时候屏幕显示如上。

注意我们一般在屏幕上做定位都是以下方向为y轴正方向的，而以现在的测试结果看来，鼠标是以上为正方向的，为了符合我们的一般认知，我们把y轴分量取个相反数吧！

```
mdec->y = -mdec->y
```

在`mouse_decode`的phase 3加部分入如上代码即可



结果符合预期！

SubPhase 3

有了鼠标移动的瞬时速度，我们就可以让光标移动起来啦！不过仍然有需要注意的地方，我们需要避免鼠标移动出屏幕之外。具体做法是将鼠标加上速度之后的x、y值对屏幕范围取个min max就好了。

让鼠标移动起来的具体做法是先将原来的光标删除，然后计算光标的新位置，并在新位置上绘制光标。

书上是这么处理的

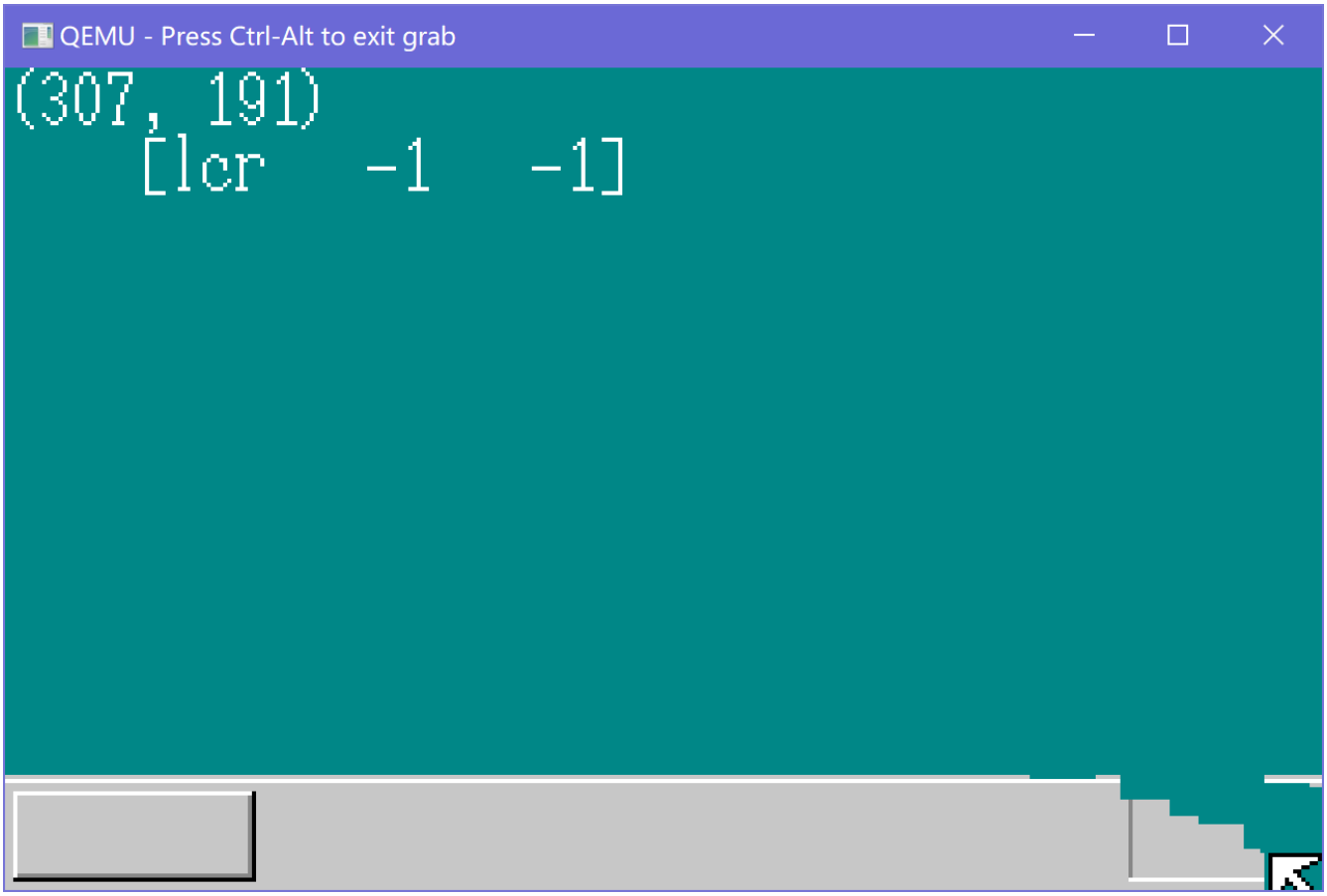
```
boxfill8(binfo->vram, binfo->scrnx, COL8_008484, mx, my, mx + 15, my + 15); /* マウス消す */
mx += mdec.x;
my += mdec.y;
if (mx < 0) {
    mx = 0;
}
if (my < 0) {
    my = 0;
}
if (mx > binfo->scrnx - 16) {
    mx = binfo->scrnx - 16;
}
if (my > binfo->scrny - 16) {
    my = binfo->scrny - 16;
}
sprintf(s, "(%3d, %3d)", mx, my);
boxfill8(binfo->vram, binfo->scrnx, COL8_008484, 0, 0, 79, 15); /* 座標消す */
```

```
putfonts8_asc(binfo->vram, binfo->scrnx, 0, 0, COL8_FFFFFF, s); /* 座標書く */
putblock8_8(binfo->vram, binfo->scrnx, 16, 16, mx, my, mcursor, 16); /* マウス描く */
```

但我觉得这样不够优秀。这和我们平常使用电脑的时候，光标除了左上角不能超出屏幕之外，其他部分是可以超出屏幕不太一样。我们将代码稍加改动也可以实现的。

```
// HariMain
if (mx > binfo->scrnx - 1) {
    mx = binfo->scrnx - 1;
}
if (my > binfo->scrny - 1) {
    my = binfo->scrny - 1;
}
// graphics.c
void boxfill8(unsigned char *vram, int xsize, unsigned char c, int x0, int y0, int x1, int
y1)
{
    struct BOOTINFO *binfo = (struct BOOTINFO *) ADR_BOOTINFO;
    int x, y;
    for (y = y0; y <= y1; y++) {
        if (y >= binfo->scrny) return;
        for (x = x0; x <= x1 && x < binfo->scrnx; x++)
            vram[y * xsize + x] = c;
    }
    return;
}

void putblock8_8(char *vram, int vxsize, int pxsize,
    int pysize, int px0, int py0, char *buf, int bxsize)
{
    struct BOOTINFO *binfo = (struct BOOTINFO *) ADR_BOOTINFO;
    int x, y;
    for (y = 0; y < pysize; y++) {
        if (y + py0 > binfo->scrny - 1) return;
        for (x = 0; x < pxsize; x++) {
            if (x + px0 > binfo->scrnx) break;
            vram[(py0 + y) * vxsize + (px0 + x)] = buf[y * bxsize + x];
        }
    }
    return;
}
```

结果如下



注意到鼠标移动会把任务栏冲掉，作者暂时没有提供解决方案，但是我有一个：绘制屏幕（除鼠标外）不要直接在vram里面做，在内存中另外开辟一片区域B。然后一开始先把这片区域的内容先复制到vram中。当擦除鼠标的时候，我们不要直接填充颜色，而是把擦除对应B中的数据复制到VRAM当中，这样就可以保证不被擦除啦！

Phase 2

我们再回过头来看看`asmhead.nas`吧

SubPhase 1

首先在切换CPU模式之前，要先暂时的关闭CPU的中断，否则如果正在切换模式的时候来了一个中断，就会产生问题。

先禁止主PIC，然后禁止从PIC。

```
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
```

SubPhase 2

内存分布

地址范围	大小	备注
0x00000000 - 0x000fffff	1MB	包含BIOS、VRAM等内容
0x00100000 - 0x00267fff	1440KB	用于保存软盘的内容。
0x00268000 - 0x0026f7ff	30KB	空
0x0026f800 - 0x0026ffff	2KB	IDT
0x00270000 - 0x0027ffff	64KB	GDT
0x00280000 - 0x002fffff	512KB	bootpack.hrb
0x00300000 - 0x003fffff	1MB	栈及其他
0x00400000 -	-	空

SubPhase 3

```
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
```

我们利用它将bootpack.hrb第0x10c8字节开始的0x11a8字节复制到0x00310000号地址去

今天先到这吧。