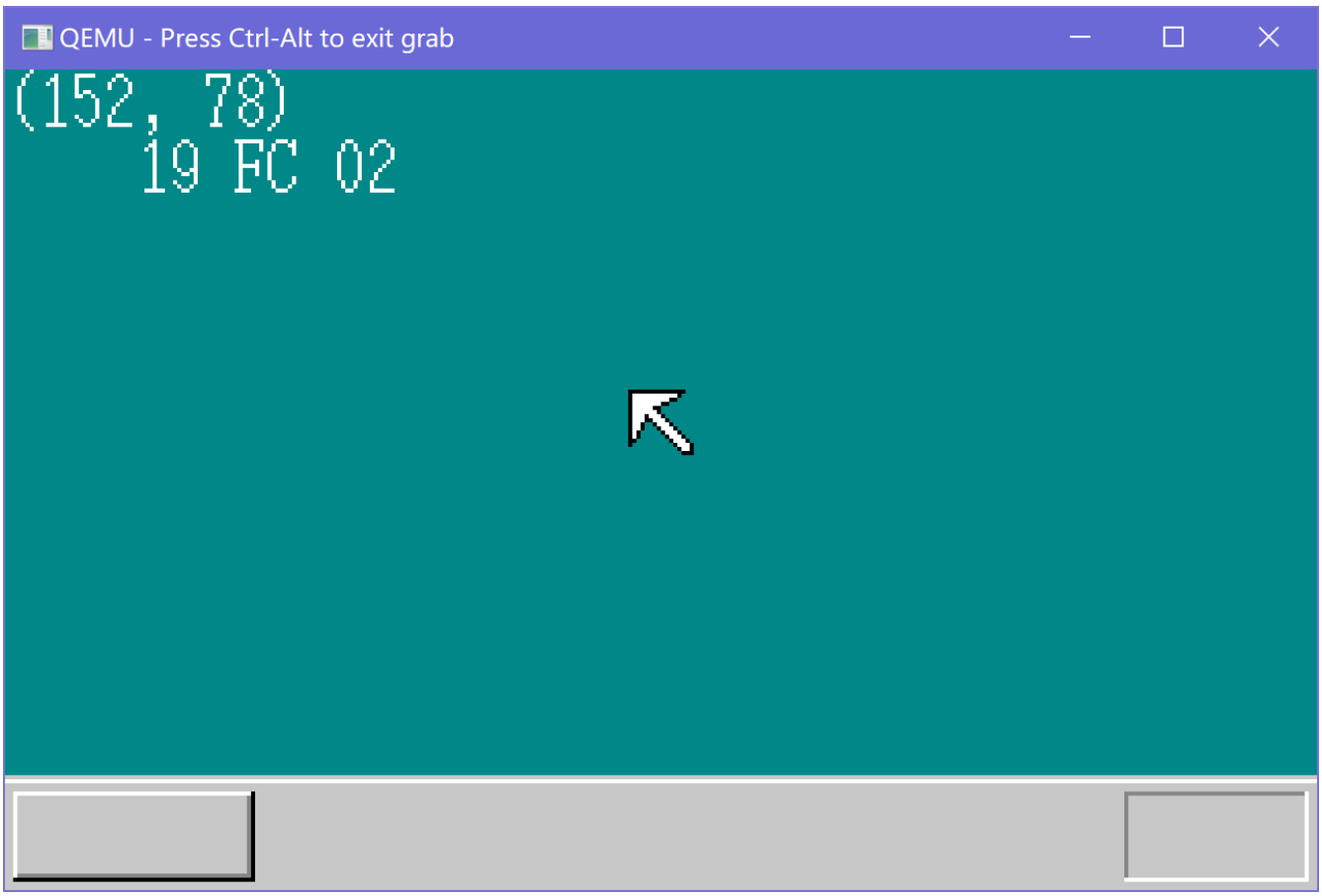
# Day 8

## 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

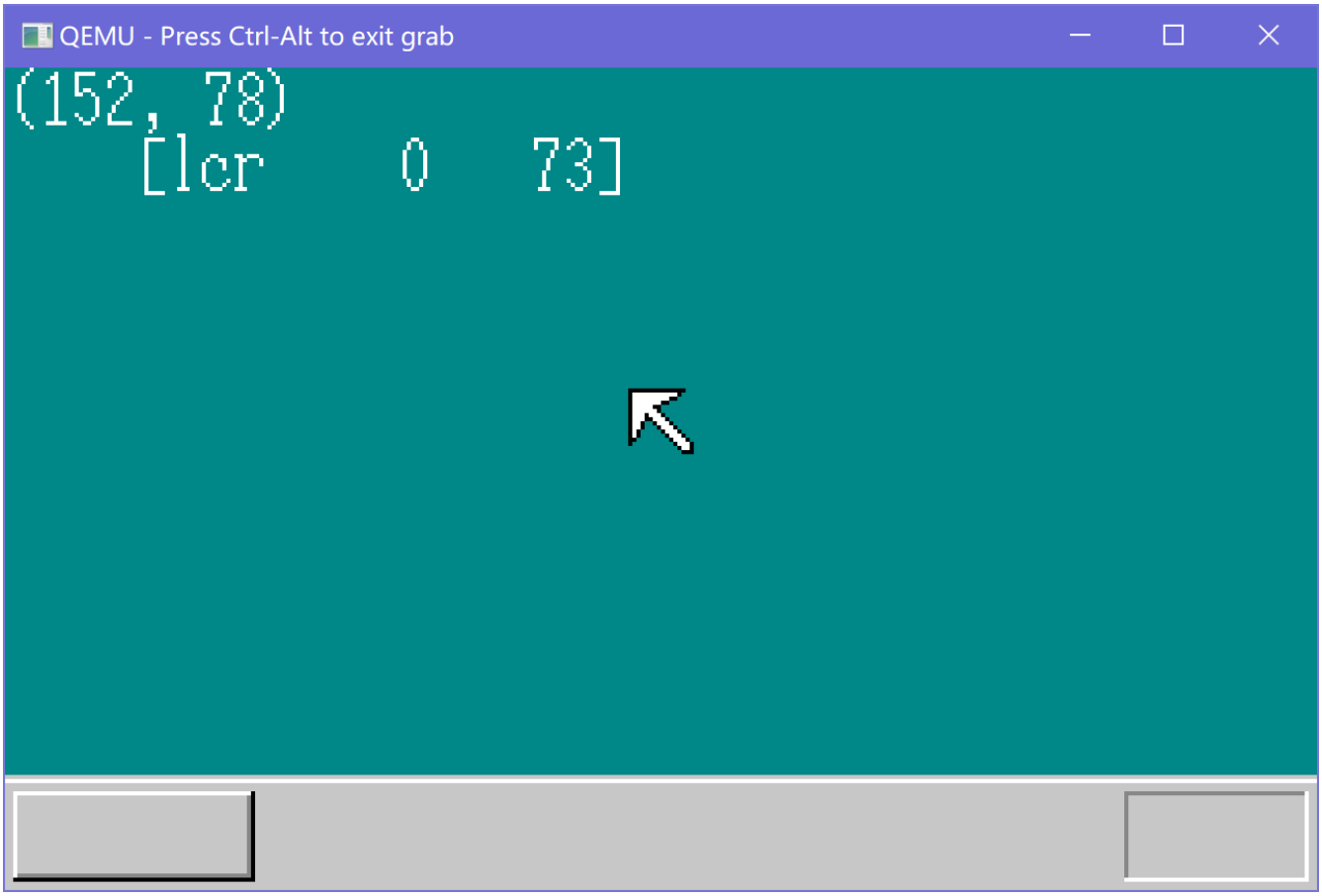
由于鼠标偶尔会断线，断线之后会重新发送0xfa数据。这将使我们之后的每一个数据都产生错位，从而使得鼠标无法正常工作。由于第一字节对移动的反应和对点击的反应都是分别限定在0~3和8~F的范围内的，我们可以以此为依据，适当的扔掉几个字节，这样就可以重新对齐了。

```
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！

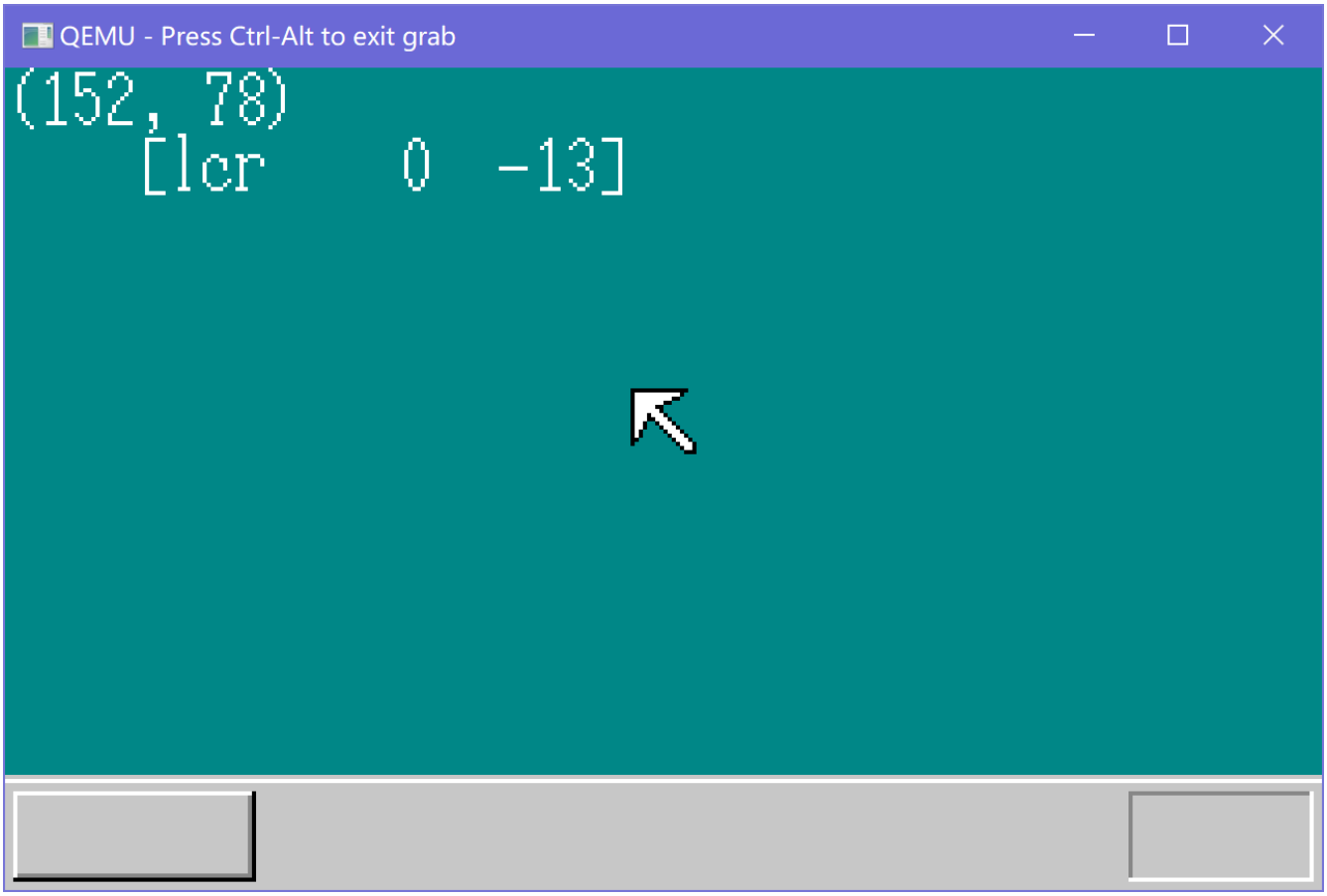



鼠标快速向上移动的时候屏幕显示如上。

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 */
```

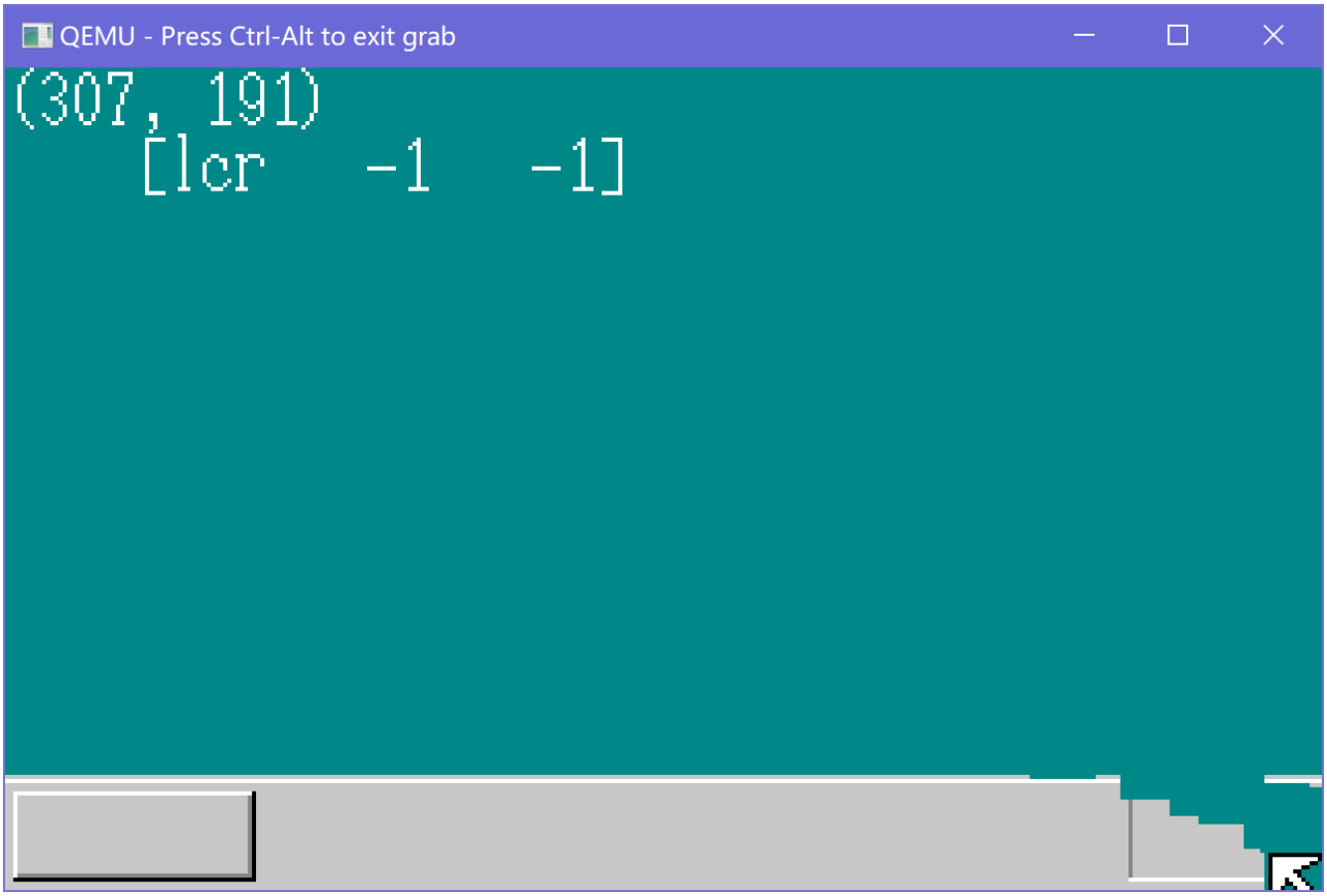
```
putfonts8_asc(binfo->vram, binfo->scrnx, 0, 0, COL8_FFFFFF, s); /* 座標書く */
putblock8_8(binfo->vram, binfo->scrnx, 16, 16, mx, my, mcursor, 16); /* マウス描く */
```

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。