嵌入式系统仿真实验第九讲

上一次我们在 LCD 上显示了 ASCII 字符，今天的实验，我们来看如何在 LCD 终端界面上显示汉字字符以及键盘驱动输入字符（上次的实验，不是键盘输入，而是用串口终端模拟输入的）。

1 LCD 汉字字符显示

显示 ASCII 字符，我们用的 ASCII 字符点阵，这是在 bare metal programming(裸机编程)阶段嵌入式系统工程师必备的技能，因为如果只是个简单的产品，我们能节约一点硬件成本那都是很了不起的。如果我们能掌握一点汉字字符的显示原理，不仅能帮助我们深入理解计算机的原理，还说不定在今后的工作中能发挥点作用。

在使用汉字字符之前，我们简单介绍一下汉字的编码。早期的计算机是使用英文的人发明的，所以刚开始是没有中文输入和显示的，后来有很多人研究中文输入法和中文显示，包括联想的起底，也是靠倪光南院士他们当初做的汉卡发家。

要用汉字编码，先看看有哪些常见的汉字字符集编码：

GB2312 编码：1981 年 5 月 1 日发布的简体中文汉字编码国家标准。GB2312 对汉字采用双字节编码，收录 7445 个图形字符，其中包括 6763 个汉字。

BIG5 编码：台湾地区繁体中文标准字符集，采用双字节编码，共收录 13053 个中文字，1984 年实施。

GBK 编码：1995 年 12 月发布的汉字编码国家标准，是对 GB2312 编码的扩充，对汉字采用双字节编码。GBK 字符集共收录 21003 个汉字，包含国家标准 GB13000-1 中的全部中日韩汉字，和 BIG5 编码中的所有汉字。

GB18030 编码：2000 年 3 月 17 日发布的汉字编码国家标准，是对 GBK 编码的扩充，覆盖中文、日文、朝鲜语和中国少数民族文字，其中收录 27484 个汉字。GB18030 字符集采用单字节、双字节和四字节三种方式对字符编码。兼容 GBK 和 GB2312 字符集。

Unicode 编码：国际标准字符集，它将世界各种语言的每个字符定义一个唯一的编码，以满足跨语言、跨平台的文本信息转换。

目前的操作系统一般都支持 Unicode 编码，也就是再不会出现由于字符编码格式造成的乱码问题出现，早期的时候汉字是双 8 位-两个字节的编码，而 ASCII 是 7 位编码，所以很多中文邮件之类的文档收发都会造成乱码出现，那时候出现过很多的转码软件（我在九十年代做自由程序员的时候，就帮别人做过 BIG5 和 GB2312 码的互换程序），拿到现在这些软件就完全没有作用了，但在嵌入式系统领域，低成本、高可靠性、高附加值都得靠软件来实现，即使后面我给大家讲到 bootloader 的移植，操作系统的移植，能否把系统裁剪得越小越高效可靠，是考察一个嵌入式系统工程师基本功的关键，硬件的迭代有摩尔定律，但同等条件下，**嵌入式系统工程师的功底在产品竞争中往往取到了决定性的作用**。

早期，UCDOS 下使用的 HZK16 目前在嵌入式系统里面还有很多人在用。

HZK16 字库是一个符合 GB2312 标准的 16×16 点阵字库，HZK16 的 GB2312-80 支持的汉字有 6763 个，符号 682 个。其中一级汉字有 3755 个，按声序排列，二级汉字有 3008 个，按偏旁部首排列。这个字体文件大概 256K 字节大小。

HZK16 字库用 16×16 共 256 个点来显示一个汉字，也就是需要 32 个字节才能显示一个汉字。在 GB2312 汉字编码表中，一个汉字由两个字节编码，其范围为 A1A1~FEFE。比如 0xCEE4 这个编码，代表汉字“武”。高八位称为区码，从 A1-A9 为符号区，B0 到 F7 才是汉字区。每一个区有 94 个字符（注意：这只是编码的许可范围，不一定都有字型对应，比如符号区就有很多编码空白区域）。低八位为位码，表示该字在位区的位置。

区码：汉字的高字节，因为汉字编码是从 0xA0 区开始的，所以文件最前面就是从 0xA0 区开始，要算出相对区码

位码：汉字的低字节，表示该字在位区的位置。

由此，我们可以算出一个汉字在 HZK16 中的绝对偏移位置：

offset=(94*(区码-1)+(位码-1))*32

注: 1、区码减 1 是因为数组是以 0 为开始而区码位码是以 1 为开始的；

2、(94*(区码-1)+位码-1)表示每个区 94 个字符排列，在该区内该字的位码是处在第几个位置。

3、最后乘以 32，因为 hzk16 汉字库要用 16x16=256 位也就是 32 个字节信息记录该字的字模信息，每一位二进制”0” 表示背景色，”1”表示显示色，这样组成一个汉字的显示字模。

我们通过汉字的两字节编码（区码和位码），计算出它在字体文件 HZK16 中存放的偏移地址就可以读取这 32 个字节的点阵信息，然后将它们画出来就可以显示出中文字了。

比如“武”字，GB2312 的编码为 0x CEE4，那么“武”字在 hzk16 文件里面存放的点阵信息的偏移地址就是(94*(0xCE-1)+0xE4-1)*32=0x098520，也就是从 0x098520 开始 32 个字节存放的是“武”的点阵信息。

明白了这个原理，我们就可以编程序了，字符文件我们目前还不能像大家在有操作系统或者基础库的情况下对文件进行操作，我们只能选择两种方式，前面我有介绍，一种用我前面编的二进制转 C 代码程序把 hzk16k（k 是楷体字模）文件转成 C 语言字节数组，另一个方法就是 objcopy 命令生成.o 文件，然后在链接文件数据段指明链接地址。

接着上次的程序，我们在 LCD 驱动文件 vid.c 里面加几个函数。

```
extern char _binary_hzk16s_start;
void lcd_put_pixel(int x, int y, int u_color)
{
    int c;
    c = get_color();
    set_color(u_color);
    set_pix(x,y);
    set_color(c);
}

/*      LCD 显示单个汉字
 *      x : 屏幕 x 轴的坐标
 *      y : 屏幕 y 轴的坐标
 *      word : 需要显示的汉字字符编码（两个字节）
```

```
 *      scale: 显示比例
 */
void lcd_show_single_chinese(int x, int y, unsigned char* word, int scale)
{
    //汉字库 hzk16 的起初地址
    volatile unsigned char *addr = &_binary_hzk16_start;
    //一个汉字占 32 位数据
    unsigned char buffer[32];
    //一个汉字的点阵是 16*16
    unsigned char val[256];
    unsigned int offset,i,j,k,len;
    /* 计算偏移地址 */
    offset = (94*(unsigned int)(word[0]-0xa0-1)+(word[1]-0xa0-1))*32;
    addr += offset;//首地址+偏移地址
    //拷贝 32 个字节数据到 buffer 数组
    for(i=0; i<32; i++,addr++)
    {
    buffer[i] = *addr;
    }

    /* 按顺序取出 32 个字节字模中的每一个二进制位，共 256 位，这个 0 或 1 放到 val 数
组 */

    len = 0;
    for(k=0; k<16; k++)
    {
        for(j=0; j<2; j++)
        {
            for(i=0; i<8; i++)
            {
                val[len++] = buffer[k*2+j]&(0x80 >> i);
            }
        }
    }

    /* 根据 val 数组里面的 0 或 1 点阵来设置对应坐标点象素的颜色 */
    len = 0;
    for (j = y; j < y+16; j++)
    {
        for (i = x; i < x+16; i++)
        {
        /* 根据点阵的某位决定是否描颜色 */
        if (val[len++])
        lcd_put_pixel(scale*i, scale*j, color);
```

```
        }
    }

}

//LCD 打印多个中文字符串
void fb_print_chinese(int x, int y, unsigned char* word, int scale)
{
    int i = 0;

    while(word[i])
    {
    lcd_show_single_chinese(x,y,&(word[i]),scale);
    i+=2;   //一个汉字占两个字节
    x+=16;  //一个汉字显示 16x16，下一个字开始横坐标要加 16
    }
}
```

好了，带显示中文字符的 LCD 驱动我们编写完成了，我们可以在 main.c 里面去使用了。

我直接在 main_loop 函数里加了这几句测试一下：

```
    unsigned char incode[10] =
{0xCE,0xE4,0xBA,0xBA,0xBC,0xD3,0xD3,0xCD};//武汉加油
    unsigned char ci[30] =
{0xBB,0xC6,0xBA,0xD7,0xC2,0xA5,0xD6,0xD0,0xB4,0xB5,0xD3,0xF1,
0xB5,0xD1,0xA3,0xAC,0xBD,0xAD,0xB3,0xC7,0xCE,0xE5,0xD4,0xC2,0xC2,0xE4
,0xC3,0xB7,0xBB,0xA8}; //黄鹤楼中吹玉笛，江城五月落梅花
    p = &_binary_image1_start;
    show_bmp(p,54,0);

    set_color(GREEN);
    fb_print_chinese(200,20,ci,1);
    set_color(CYAN);
    fb_print_chinese(120,40,incode,2);

    for(i=0;i<50;i++)
    {
    set_color(GREEN+i*50);
    fb_print_chinese(120,40,incode,2);
    delay(0x00FFFFFF);
    }
    set_color(GREEN);
```

```
fb_print_chinese(120,40,incode,2);
```

show_bmp 函数是前面讲过的，这里新测试的是 fb_print_chinese 函数，大家可以看到我在主屏里显示了一幅图片，然后在屏幕上方打印了一行汉字（”黄鹤楼中吹玉笛，江城五月落梅花”），然后循环打印了武汉加油字样，改变颜色，让其有了一些闪烁的效果而已，我没有仔细去设计，留给你们可以去发挥，比如自己编写画点、画圆、画线、画圆弧、画方块等函数，自己编写 GUI 函数库，设计图形界面等等，包括最近网上比较火的武大的学生用 python 写的武汉加油组成的樱花图案，你们也可以去尝试，寻找编程的乐趣。☺

图 1 汉字显示测试

2 中断与键盘驱动实验

这门课仿真实验做到现在，我想，即使没有实物板子应该对大家的学习也没有什么影响了。外围接口这一章我们实际一直都没讲一个关键的计算机处理机制，中断机制，对于外围接口来说，计算机一般采用中断方式来进行响应。没有讲中

断实验，恐怕是不能算数的。所以我们下面来看看 qemu 是否可以模拟硬件中断。

我们打算以键盘中断来讲中断处理机制。

回到最前面，我们在启动代码 start.S 文件里面，最开始的地方是这么写的：

```
.globl _start
.global reset

_start:
    b   reset
    b . /* Undefined */
    b . /* SWI */
    b . /* Prefetch Abort */
    b . /* Data Abort */
    b . /* reserved */
    b   irq_handler
    b . /* FIQ */
```

这里大家就会问，老师，你没有写中断的异常处理函数呀，这怎么办？

接下来，我们要找出方法来进行中断的异常处理。计算机一上电就会进入复位异常，所以我们都是从复位程序里开始处理所有的事，然后转到 C 语言的。我们要处理键盘的中断，肯定不希望回到汇编去编写键盘的中断服务程序。但是要让计算机相应中断，我们必须对中断寄存器进行设置，所以我们在 reset 段开始进入 SVC32 模式后，就对中断寄存器进行了设置(start.S)：

```
copy_vector:
    LDR sp, =svc_stack   /* set SVC stack*/
    BL copy_vectors     /* copy vector table to 0 */

    MRS r0, cpsr        /* go into IRQ mode*/
    BIC r1, r0, #0x1F
    ORR r1, r1, #0x12
    MSR cpsr, r1
    LDR sp, =irq_stack   /* set IRQ stack */

    BIC r0, r0, #0x80    /* mask in IRQ interrupt I-bit in CPSR */
    MSR cpsr, r0        /* back to SVC mode*/
```

大家可以看到这一小段汇编代码，进入 C 语言的 copy_vectors 函数之前，对 SVC 模式的寄存器环境进行了现场保护 `LDR sp, =svc_stack`，然后使用 BL 指令

跳转到一个C语言程序，这个C语言函数将中断异常处理函数的入口地址拷贝到ＡＲＭ的异常向量表里，完成它们的异常响应服务程序的入口地址修改。修改完成后使用 MRS 指令修改当前程序状态字，让 CPU 进入 IRQ 模式，设置中断模式的堆栈指针初始化，设置中断屏蔽位，然后返回 svc 模式。

copy_vectors 我把它写在了 main.c 里面：

```
extern void reset();

/* all other interrupt handlers are while(1) loops */
void __attribute__((interrupt)) undef_handler(void) { while(1); }
void __attribute__((interrupt)) swi_handler(void) { reset(); }
void __attribute__((interrupt)) prefetch_abort_handler(void)
{ while(1); }
void __attribute__((interrupt)) data_abort_handler(void) { while(1); }
void __attribute__((interrupt)) fiq_handler(void) { while(1); }

void copy_vectors(void) {
    extern u32 vectors_start;
    extern u32 vectors_end;
    u32 *vectors_src = &vectors_start;
    u32 *vectors_dst = (u32 *)0;

    while(vectors_src < &vectors_end)
       *vectors_dst++ = *vectors_src++;
}
```

就是将在 start.S 里面写的几个汇编入口标记地址拷贝到异常向量表里面。

这些入口标记在 start.S 里面：

```
.global vectors_start, vectors_end
.global lock, unlock, int_off, int_on

irq_handler:

  sub   lr, lr, #4
  stmfd sp!, {r0-r10, fp, ip, lr}

  bl    IRQ_handler

  ldmfd sp!, {r0-r10, fp, ip, pc}^
```

```
lock:
    mrs r0, cpsr
    orr r0, r0, #0x80   @set CPSR I-bit to 1
    msr cpsr, r0
    mov pc, lr
unlock:
    mrs r0, cpsr
    BIC r0, r0, #0x80   @clear CPSR B-bit to 0
    msr cpsr, r0
    mov pc, lr

int_off:                @int cpsr = int_off()
  MRS r1, cpsr
  MOV r0, r1
  ORR r1, r1, #0x80       @set I-bit to 1
  MSR cpsr, r1           @load into CPSR => IRQ masked out
  mov pc, lr

int_on:                 @int_off(cpsr)
  MSR cpsr, r0           @r0 = original CPSR: load into CPSR
  mov pc, lr


vectors_start:
  LDR PC, reset_handler_addr
  LDR PC, undef_handler_addr
  LDR PC, swi_handler_addr
  LDR PC, prefetch_abort_handler_addr
  LDR PC, data_abort_handler_addr
  B .
  LDR PC, irq_handler_addr
  LDR PC, fiq_handler_addr

reset_handler_addr:         .word reset
undef_handler_addr:         .word undef_handler
swi_handler_addr:           .word swi_handler
prefetch_abort_handler_addr: .word prefetch_abort_handler
data_abort_handler_addr:     .word data_abort_handler
irq_handler_addr:           .word irq_handler
fiq_handler_addr:           .word fiq_handler

vectors_end:
```

其中就 IRQ 是写了处理函数，而其他的几个异常都是搞了个 while(1)循环，

让它死机☺。哦，对了软中断前面我们用到了，所以我把软中断直接跳转到复位入口让计算机直接软复位算了。

enable_irq 和 disable_irq 函数我们在第三章指令集里面讲过，在我的 ppt 第 63 和 64 页，大家可以返回去再看这一节，是开中断和关中断的标准函数。MRS 和 MSR 指令需要配套使用，一定要遵循读－改－写的流程操作，以免误操作发生。

中断响应函数需要拧出来单独说一下：

```
irq_handler:

    sub lr, lr, #4
    stmfd   sp!, {r0-r10, fp, ip, lr}
    bl  IRQ_handler
    ldmfd   sp!, {r0-r10, fp, ip, pc}^
```

第一条指令调整链接寄存器（r14）以返回到中断点。 stmfd 指令通过将必须保留的 CPU 寄存器压入堆栈在执行中断程序前保存上下文。执行 STMFD 或 LDMFD 指令所需的时间与要传输的寄存器数量成正比，为了减少中断处理延迟，应保存最少数量的寄存器。用高级编程语言（例如 C）编写 ISR 时，了解编译器生成代码的调用约定很重要，因为这会影响到底应将哪些寄存器保存在堆栈中的决定。例如，ARM 编译器生成的代码在函数调用期间会保留 r4－r11，因此，除非中断处理程序将使用这些寄存器，否则无需保存这些寄存器。保存寄存器后，现在可以安全地调用 C 函数来处理中断。在中断处理程序的末尾，ldmfd 指令恢复保存的上下文并从中断处理程序返回。 **ldmfd 指令末尾的'^'符号表示将从保存的 SPSR 恢复 CPSR**。我们在第三章 ARM 指令里面讲过，当且仅当同时加载 PC 时，“ ^”才恢复保存的 SPSR。否则，它仅恢复先前模式的存储区寄存器，不包括保存的 SPSR。在特权模式下，我们可以使用此特殊功能访问用户模式寄存器。

在 main.c 里面，这个实际的 C 语言中断处理函数如下：

```
void IRQ_handler()
{
    int vicstatus, sicstatus;
    int ustatus, kstatus;
```

```
    // read VIC SIV status registers to find out which interrupt
    vicstatus = VIC_STATUS;
    sicstatus = SIC_STATUS;

    if (vicstatus & (1<<31)){
      if (sicstatus & (1<<3)){
         kbd_handler();
       }
    }
}
```

程序从向量中断控制器里面读取中断状态字，判断如果发生了中断，而且中断源是键盘中断，键盘在辅助 VIC 上的 IRQ3 处中断，该中断被映射到主 VIC 上的 IRQ31。如果判断是主 irq31 下的 irq3 中断，那么执行 kbd_handler 程序。

Kbd.c 为键盘的初始化和键值扫描程序。

```
#define NSCAN 58
/* Scan codes to ASCII for unshifted keys */
char unshift[NSCAN] = { // NSCAN=58
  0, 033,'1','2','3','4','5','6','7','8','9','0','-','=','\b','\t',
 'q','w','e','r','t','y','u','i','o','p','[',']', '\r', 0,'a','s',
 'd','f','g','h','j','k','l',';', 0,  0,  0,  0, 'z', 'x','c','v',
 'b','n','m',',','.','/', 0, '*', 0, ' ' };
/* Scan codes to ASCII for shifted keys */
char shift[NSCAN] = {
  0, 033,'!','@','#','$','%','^','&','*','(',')','_','+','\b','\t',
 'Q','W','E','R','T','Y','U','I','O','P','{','}', '\r', 0,'A','S',
 'D','F','G','H','J','K','L',':', 0, '~', 0,  '|','Z','X','C','V',
 'B','N','M','<','>','?', 0, '*', 0, ' ' };
```

图 2 按键扫描码与 ASCII 码映射表

ARM 多功能键盘中断，必须将键盘的控制寄存器初始化为 0x14，其位 2 使能键盘，位 4 使能 Rx（输入）中断。键盘在辅助 VIC 上的 IRQ3 处中断，该中断被连接到主 VIC 上的 IRQ31。键盘会生成扫描码，而不是 ACSII 码。键盘驱动程序中必须包含扫描码的完整列表。扫描码到 ASCII 的转换是通过在软件中映射表来完成的，这就可以将同一键盘用于不同的语言。对于每个键入的按键，键盘都会产生两个中断，一个在按下键时产生，另一个在释放键时产生。释放按键的扫描代码为 0x80 +按下按键的扫描码，即 bit7 为 0（按键按下）和 1（按键释放）。键盘中断时，扫描码位于数据寄存器（0x08）中。中断处理程序必须读取数据寄存器以获取扫描码，这也将清除键盘中断。一些特殊键会生成转义键序列，例如向上箭头键生成 0xE048，其中 0xE0 是转义键本身。图 2 显示了用于将扫描代码

转换为 ASCII 的映射表。键盘有 105 个按键。高于 0x39（57）的扫描码是特殊键，无法直接映射，因此它们不会显示在键映射中，由驱动程序识别并处理此类特殊键。图 2 显示键映射表。

完整的键盘驱动程序 kbd.c 如下*：

```
#include "keymap2"
#include "mystring.h"

/**********************************************************************
************
//0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
char ltab[] = {
  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
  0,   0,   0,   0,   0,  'q', '1',  0,   0,   0,  'z', 's', 'a', 'w', '2',
0,
  0,  'c', 'x', 'd', 'e', '4', '3',  0,   0,  ' ', 'v', 'f', 't', 'r', '5',
0,
  0,  'n', 'b', 'h', 'g', 'y', '6',  0,   0,   0,  'm', 'j', 'u', '7', '8',
0,
  0,  ',', 'k', 'i', 'o', '0', '9',  0,   0,  '.', '/', 'l', ';', 'p', '-',
0,
  0,   0,  '\'', 0,  '[', '=',  0,   0,   0,   0, '\r', ']',  0, '\\',  0,
0,
  0,   0,   0,   0,   0,   0,  '\b', 0,   0,   0,   0,   0,   0,   0,   0,
0
};

char utab[] = {
  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
0,
  0,   0,   0,   0,   0,  'Q', '!',  0,   0,   0,  'Z', 'S', 'A', 'W', '@',
0,
  0,  'C', 'X', 'D', 'E', '$', '#',  0,   0,  ' ', 'V', 'F', 'T', 'R', '%',
0,
  0,  'N', 'B', 'H', 'G', 'Y', '^',  0,   0,   0,  'M', 'J', 'U', '&', '*',
0,
  0,  '<', 'K', 'I', 'O', ')', '(',  0,   0,  '>', '?', 'L', ':', 'P', '_',
0,
  0,   0,  '"',  0,  '{', '+',  0,   0,   0,   0,  '\r','}',  0,  '|',  0,
0,
```

```
  0,   0,   0,   0,   0,   0, '\b', 0,   0,   0,   0,   0,   0,   0,   0,
0
};
*************************************************************************
*************/
// KBD registers from base address
#define KCNTL 0x00
#define KSTAT 0x04
#define KDATA 0x08
#define KCLK  0x0C
#define KISTA 0x10

typedef struct kbd {
    char *base;
    char buf[128];
    int head, tail, data, room;
}KBD;

KBD kbd;

int kputc(char);

int shifted = 0;
int release = 0;
int control = 0;

int kbd_init()
{
    KBD *kp = &kbd;
    kp->base = (char *)0x10006000;
    //*(kp->base + KCNTL) = 0x11;    // bit4=Enable   bit0=INT on
    *(kp->base + KCNTL) = 0x14;     // 00010100=INTenable, Enable
    *(kp->base + KCLK)  = 8;       // KBD internal clock setting
    kp->head = kp->tail = 0;       //Index to buffer
    kp->data = 0; kp->room = 128;   // Counter

    release = 0;  // key release flag
    shifted = 0;  // shift key down flag
    control = 0;  // control key down flag
}

// kbd_handler() for scan code set 2

/****************************************
```

```
  key press    =>       scanCode
  key release  =>   0xF0 scanCode
Example:
 press   'a'   =>        0x1C
 release 'a'   =>   0xF0   0x1C
****************************************/

void kbd_handler()
{
    unsigned char scode, c;

    KBD *kp = &kbd;
    color = YELLOW;   // int color in vid.c file
    scode = *(kp->base + KDATA);  // get scan code from KDATA reg => clear
IRQ

    //printf("scanCode = %x\n", scode);

    if (scode == 0xF0){      // key release
    release = 1;          // set flag
    return;
    }

    if (scode == 0x12) { //L Shift is being pressed
    shifted = 1;
    }

    if (scode == 0x14) { //L Ctrl is being pressed
    control = 1;
    }

    if (scode == 0x12 && release == 1) { //L Shift is released
    shifted = 0;
    }

    if (scode == 0x14 && release == 1) { //L Ctrl is released
    control = 0;
    }

    if (release && scode){    // next scan code following key release
    release = 0;          // clear flag
    return;
    }
```

```
    if (shifted && scode)
    c = utab[scode]; //Lowercase
    else                // ONLY IF YOU can catch LEFT or RIGHT shift key
    c = ltab[scode]; //Uppercase
    if (control && scode) {
    if (scode == 0x21)
    kprintf("PROGRAM IS TERMINATED\n");
    if (scode == 0x23) {
    c = 0x04;
    kprintf("FILE STREAMING IS TERMINATED\n");
    }
    }

    //kprintf("c=%x %c\n", c, c);
    kprintf("%c",c);
    //if(c=='\r')kprintf("\n");

    kp->buf[kp->head++] = c;  // Enter key into CIRCULAR buf[]
    kp->head %= 128;
    kp->data++;
    kp->room--;
}

int kgetc()
{
    char c;
    KBD *kp = &kbd;
    //printf("%d in kgetc\n", running->pid);

    while(kp->data == 0);

    disable_irq();
    c = kp->buf[kp->tail++];
    kp->tail %= 128;
    kp->data--; kp->room++;
    enable_irq();
    return c;
}

int kgets(char s[ ])
{
    char c;
    while( (c = kgetc()) != '\r'){
       if (c=='\b'){
```

```
            s--;
            continue;
            }
            *s++ = c;
        }
        *s = 0;
        return my_strlen(s);
}
```

写完这个驱动代码，我们来看看如何测试吧。

在 main.c 的主循环函数里面：

```
    kbd_init();
    VIC_INTENABLE |= 1<<31;   // SIC to VIC's IRQ31

    /* enable KBD IRQ */
    SIC_ENSET = 1<<3;  // KBD int=3 on SIC
    SIC_PICENSET = 1<<3;  // KBD int=3 on SIC

    disable_irq();
```

我们进行了键盘的初始化，设置了键盘中断相关的中断允许位，然后关掉了全局中断。Make run 运行测试如图 3 所示。

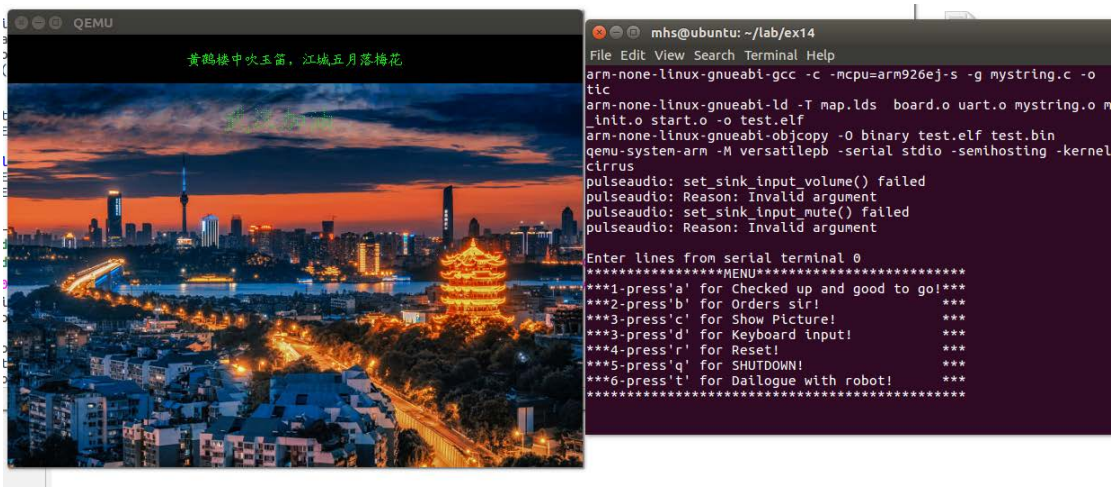

图 3，测试键盘驱动全局中断关闭情况下反应

在全局中断关掉的情况下，键盘中断没有被相应，所以我们在 qemu 的模拟端按键没有任何反应。我们按下 ctrl+alt+g 键，把系统输入光标从 qemu 虚拟机释放出来，到终端里面我们键入 d 回车，调用键盘中断测试程序段：

```
case 0x64:
enable_irq();
clr_scr();
kprintf("Test interrupt-driven KBD driver\n");
while(1){
    set_color(CYAN);
    kprintf("Enter a line from KBD\n");
    kgets(line);
    kprintf("line = %s\n", line);
    if(my_strcmp("exit",line)==0)break;
}
disable_irq();
break;
```

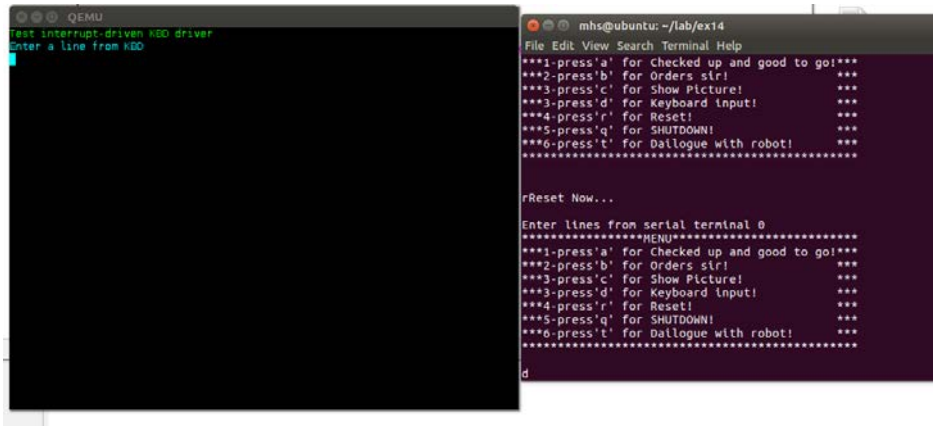

图 4 选择菜单 d 运行测试键盘驱动

打开全局中断后，可以看到我们可以在 qemu 的界面进行键盘输入了，测试见图 5 所示。

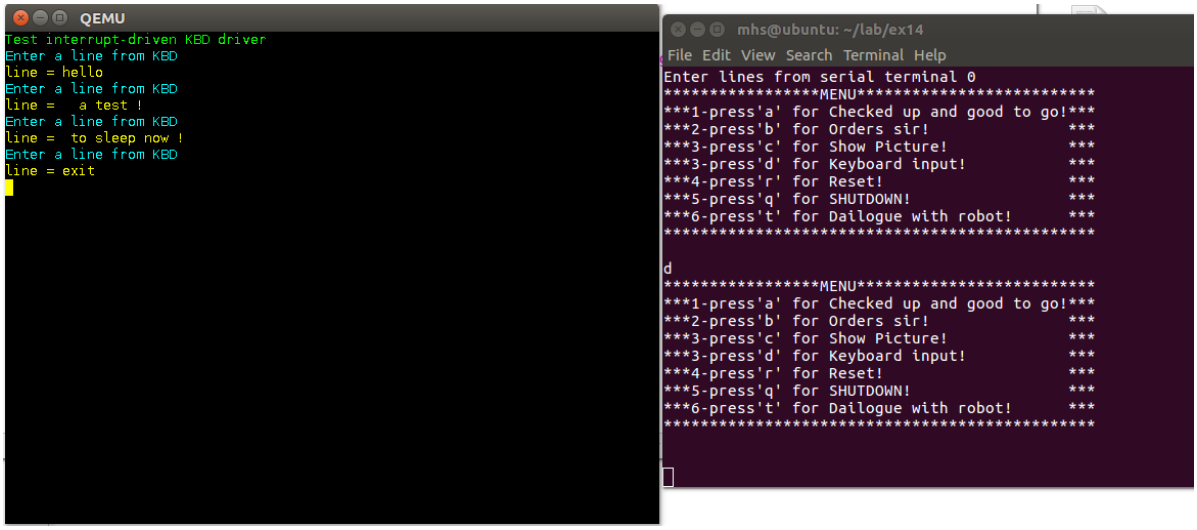

图 5，键盘输入测试

程序里我设置的是键入 exit 时，推出键盘测试，这个时候需要注意系统的光标和鼠标将不会释放，所以需要再按 ctrl+alt+g 释放。退出键盘测试的最后一句我关掉了全局中断 `disable_irq()`，所以如果你再去 qemu 的模拟窗口按键，将不再响应输入了。

注：本实验教程为武汉科技大学机器人与智能系统研究院闵华松老师的网络课程教学文档，可以复制，不做商业用途。

* K.C. Wang, kwang@eecs.wsu.edu，这个 kdb.c 文件代码参考了 WSU 的 K.C. Wang 的教材：

[1] Wang K C. Embedded real-time operating systems[M]. Embedded and Real-Time Operating Systems. Springer, Cham, 2017: 401-475.