# 操作系统实验日志

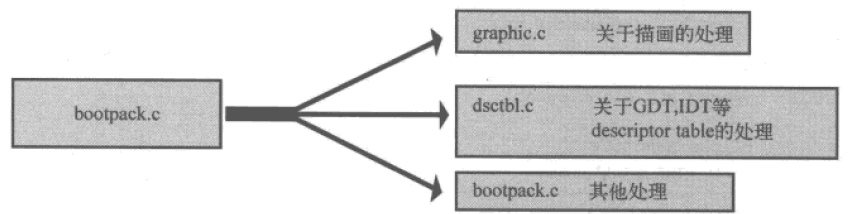
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学号 | 201808010718 | 姓名 | 肖鹏 | 专业年级班级 | 智能1802 |
| 实验日期 | 2020.10.29 | 实验项目 | 第6天：分割编译与中断处理 | | |

## 一、实验主要内容

### 1、简化文件

**内容：**

a）开发到现在，文件数目以及规模都相对开始的时候增大了许多，为了方便管理，需要将一些代码量较大的文件按函数分割为多个子文件，这就需要引入头文件；

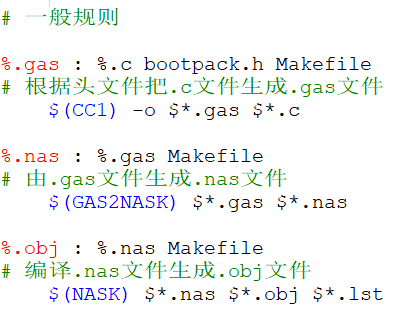


b）此外，Makefile文件中的指令也过于冗余，许多指令只是换了源文件与目标文件，用的还是相同的执行文件，所以对于这种指令可以进行归纳；

**重点总结：**

a）分割编译不适用头文件时，若相关文件代码使用到naskfunc.nas中的函数的话，需要在文件中进行相关函数的声明；使用头文件，则需要在头文件中，把所有会引用该头文件的文件中所使用到的函数都在其中声明。

**关键代码及注释：**

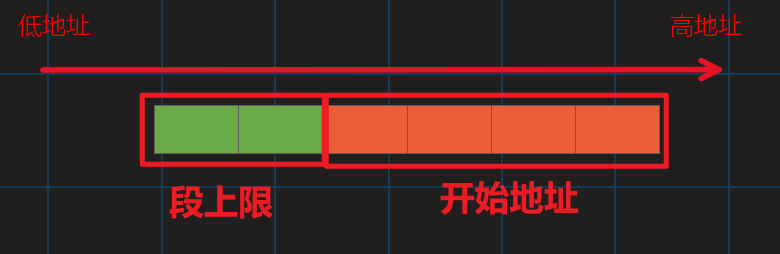


### 2、设置段表条目函数

**内容：**

**a）\_load\_gdtr函数**

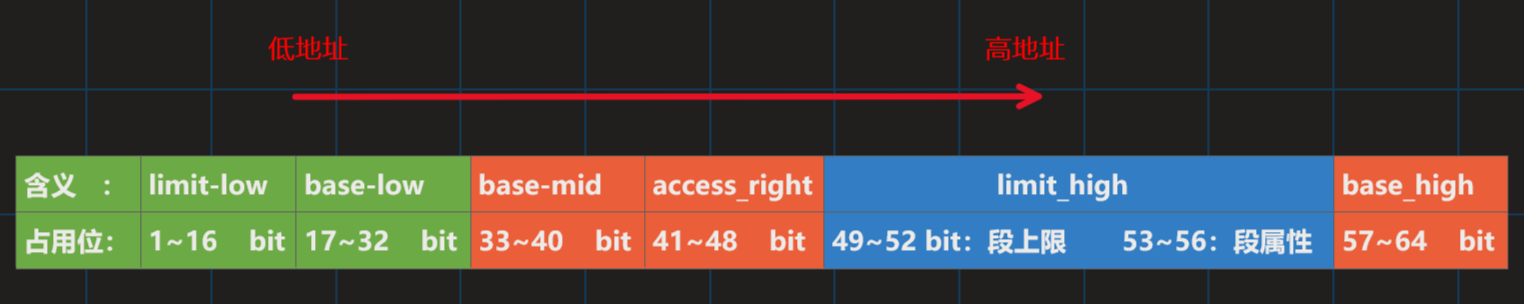
该函数用完成对寄存器GDTR的写入；向GDTR写入只能通过LGDT指令，从指定的地址取6-bytes；函数入参为两个int型，而该寄存器只有48-bits，且低16-bits用于存放段上限（2-bytes）、剩余的用于存放开始地址。



*（GDTR示意图）*

**b）set\_segmdesc函数**

该函数用于设定段表条目中的数据，首先我们在分析代码之前，了解一下该数据结构：



*（SEGMENT\_DESCRIPTION示意图）*

起始地址固定是用32-bits表示的；段上限最大是4GB，本来也该用32-bits表示，但是整个结构体空间只有64-bits，如此一来便没有空间保存段属性了，所以段上限只能使用20-bits；这样空出来的12-bits给段属性使用，在结构体中，12-bits的ar分两部分存储1）高4-bits放在limit\_high的高4-bits；2）低8-bits放在access\_right中；

**c）段属性简介：**

高四位：GD00

G是G bit，1表示段上限是以页面为单位,0以字节；

D是指段的模式,1是指32位模式,0是指16位模式；

这里出现的16位模式主要只用于运行80286的程序；

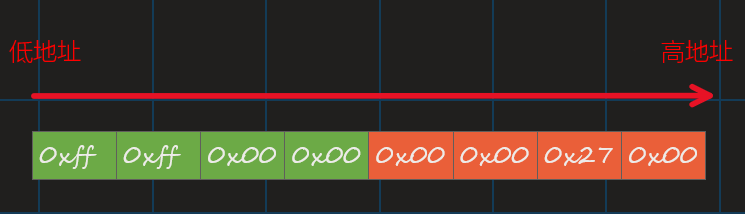
低四位：

0x00 : 未使用的记录表(descriptor table)0x92 (1001 0010): 系统专用，可读写的段、不可执行；0x9a (1001 1010): 系统专用，可执行的段、可读不可写；0xf2 (1111 0010): 应用程序用，可读写的段、不可执行；0xfa (1111 1010): 应用程序用，可执行的段、可读不可写；

**重点总结：**

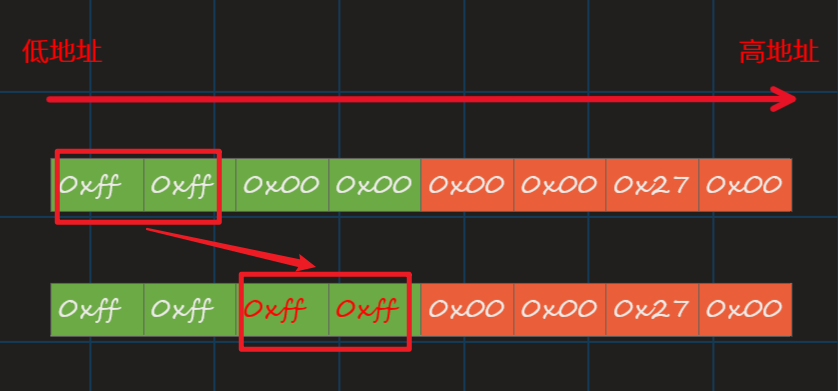
a）如何从栈中提取GDTR的6-bytes（以load\_gdtr(0x0000ffff, 0x270000)为例）：

可知，在栈中，段上限和开始地址的存放如下：



*（小端法）*

由于我们只能提取6-bytes，且只需要limit的低2-bytes，所以希望得到以下的栈图，便能直接通过LGDT指令把相应数据写入GDTR：

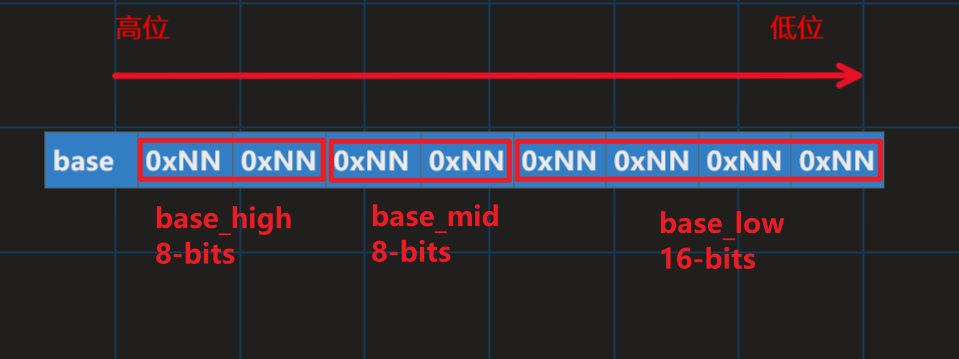


b）如何从set\_segmdesc(limit, base, ar)函数的传参中提取数据到结构体中：

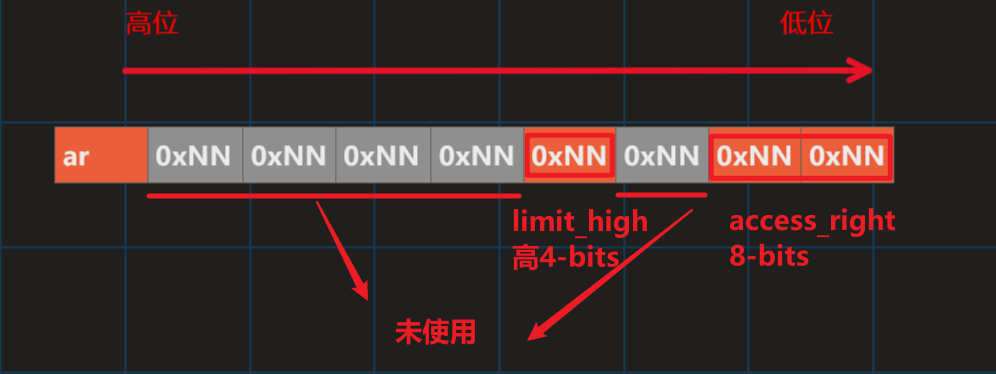
①对于参数limit，如何提取数据：



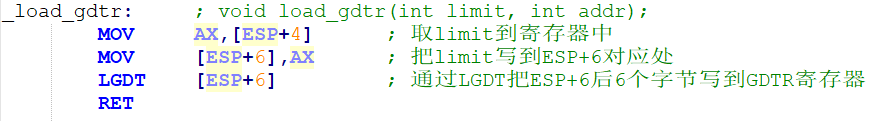
②对于参数base，如何提取数据：



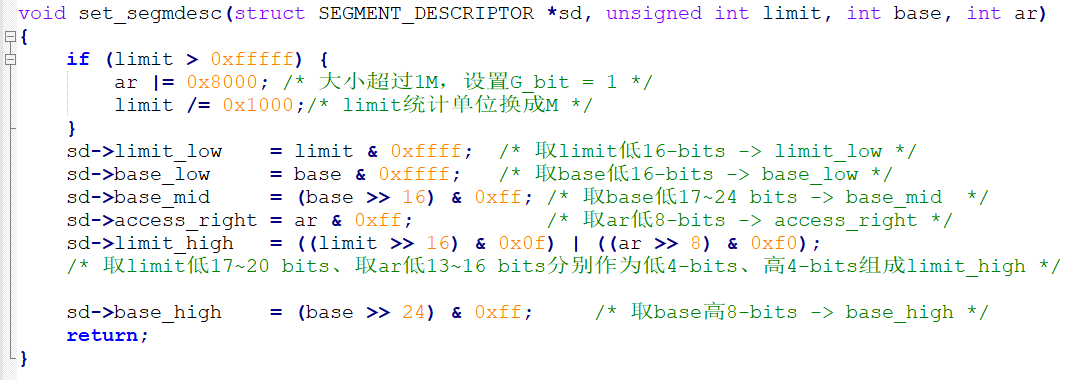
③对于参数ar，如何提取参数：



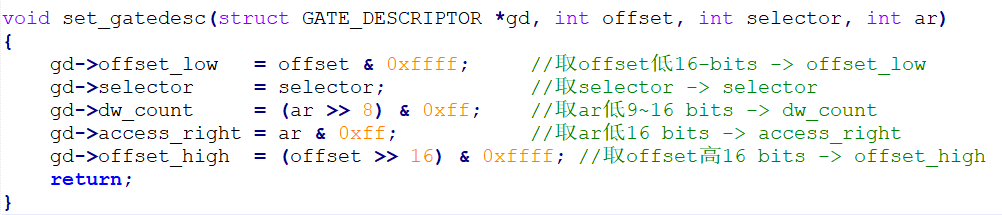
**关键代码及注释：**



*（\_load\_gdtr函数注释）*



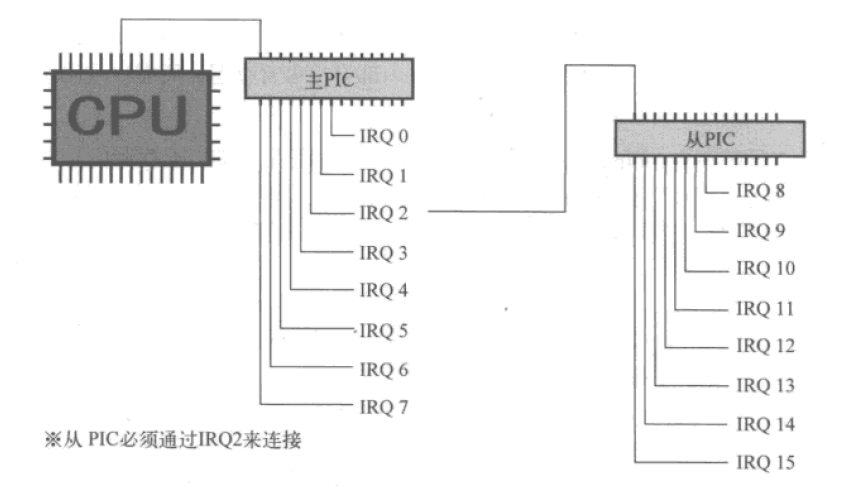
*（set\_segmdesc函数详细注解）*



*(set\_gatedesc函数详细注解)*

### 3、初始化PIC & 中断函数

内容：

a）PIC——可编程中断控制器，监视8个中断信号，1个输出管脚表示是否检测到中断；先有主、从PIC，从PIC连接于主PIC的IRQ2，故现在一共可处理15个中断信号；

b）PIC的寄存器：

①IMR——中断屏蔽寄存器，8-bits，每一位对应一路IRQ信号，若该位为1，则说明这一路的IRQ信号被屏蔽，PIC将忽略该信号；

②ICW——初始化控制数据，一共有4个，每一个8-bits，但是ICW1、ICW3、ICW4都是相对硬件固定的，能够通过软件更改的只有ICW2；ICW3是有关主-从连接的设定，对于主PIC，使用哪个IRQ与从PIC相连，ICW2哪一位就为1；对从PIC同理；

c）ICW2决定了以哪一种中断通知CPU，当CPU与PIC通过IN或OUT进行数据传输时，有数据信号线连在一起，PIC便利用这个信号线发送2-bytes的数据，即“0xcd 0xXX”，对于中断信号IRQ0~15，以INT 0x20~0x2f来接收；

d）中断处理函数，鼠标和键盘的中断处理函数相似，以键盘的为例；首先在int.c中编写inthandler21()函数，表示在中断时要做的事情；然后在naskfunc.nas中编写\_asm\_inthandler21函数，在其中调用inthandler21()函数，之后再将\_asm\_inthandler21注册到中断处理表中，这样在接收到中断INT 0x21时便可以调用该函数。

重点总结：

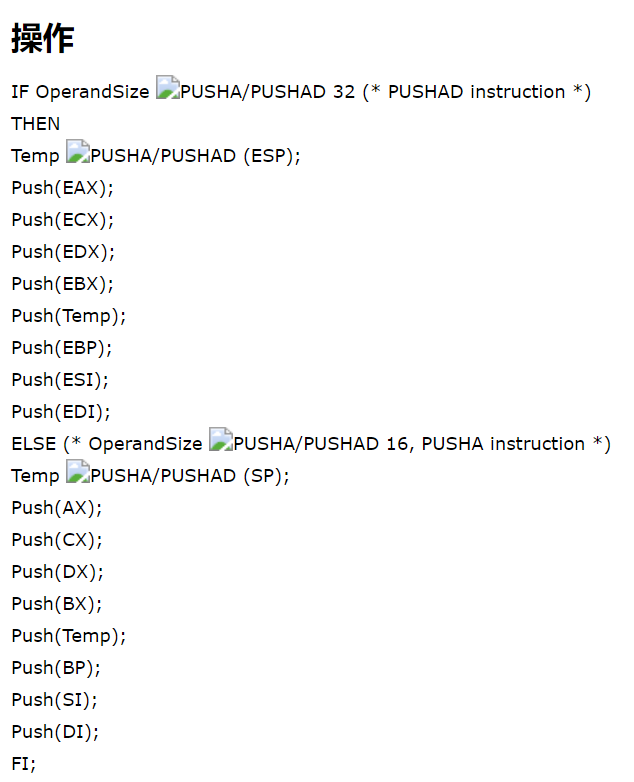
a）IRQ1对应处理键盘中断，中断号为0x21、

IRQ12对应处理鼠标中断，中断号为0x2c；

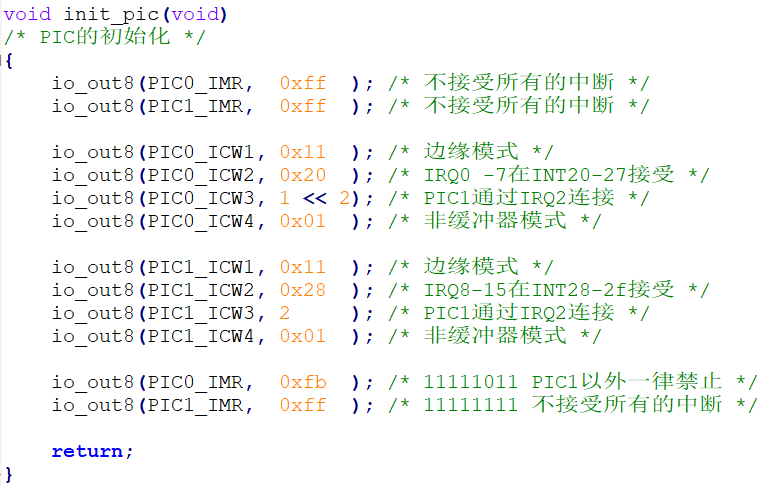
b）asm\_inthandler需要注册到中断表之后才能够在中断发生时触发handler函数;

c）pushad指令：

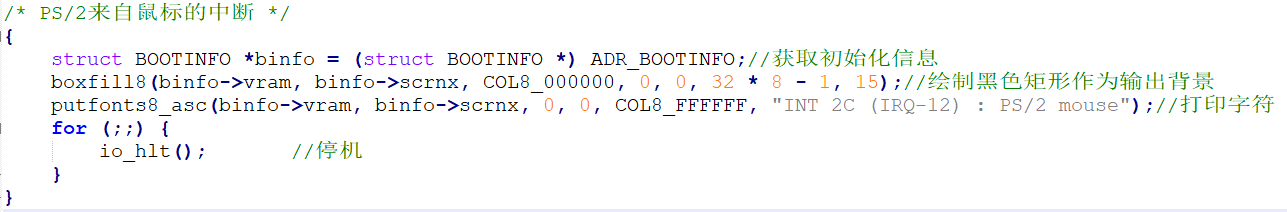
将通用寄存器的内容压入堆栈。这些寄存器按以下顺序存储到堆栈：EAX、ECX、EDX、EBX、EBP、ESP（原始值）、EBP、ESI 及 EDI（如果当前操作数大小属性为 32）；AX、CX、DX、BX、SP（原始值）、BP、SI 及 DI（如果操作数大小属性为 16）。这些指令执行 POPA/POPAD 指令的逆操作。ESP 或 SP 寄存器压入的值是它在第一个寄存器压入之前的值。



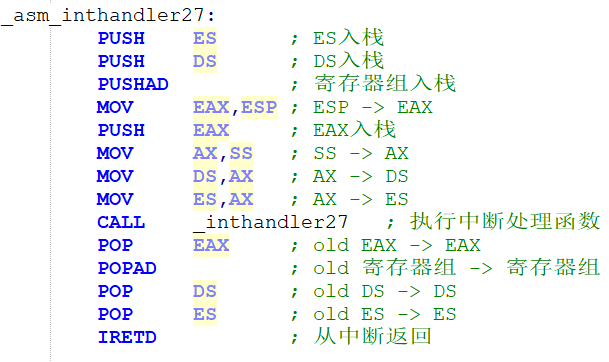
关键代码及注释：



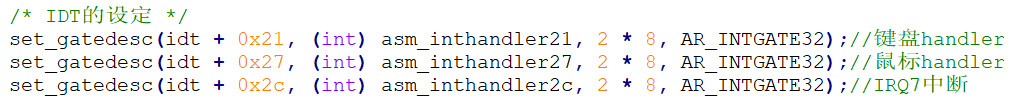
*（init\_pic详细注解）*



*（键盘中断处理函数）*



*（\_asm\_inthandler）*



*（注册asm\_inthandler）*

## 二、遇到的问题及解决方法

1、PIC的级联以及与CPU的中断号传输

**问题描述：**

①书中描述，主PIC通过ICW3设置与从PIC的连接、同理，从PIC也通过ICW3设置与主PIC的连接；那么为什么不能让从PIC的ICW3描述的是‘从PIC’与自己的‘从PIC’的连接方式呢？

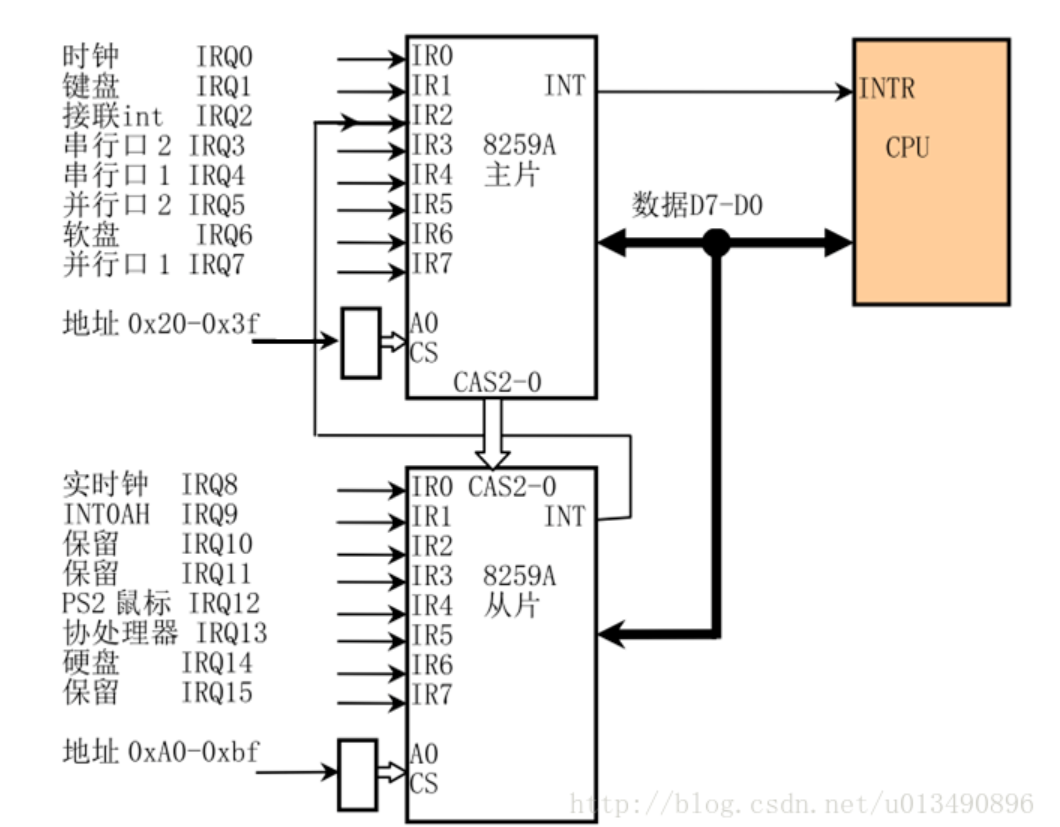
也就是这样的话，可以拓展IRQ的数目，从而处理更多的中断信号；主PIC的ICW3最多是可以把8位都设置成1，也就是最多能够管理64个中断信号、从PIC的ICW3也可根据与主PIC的级联情况而定；

②书中描述，PIC只有一个唯一的输出引脚，但是在数据传输的时候，CPU又能够通过“数据信号线”获取中断号，然后根据中断号做出中断处理函数，这是如何实现的？

**出现原因&解决方法：**

①因为书上无论主PIC、从PIC的ICW3，都是针对主PIC的，这让我产生误解——一个PIC是主或从PIC就是固定了，其实不是，在看了级联电路之后便了解，ICW3只是连接上一级的PIC的IRQ，而连接下一级的是自己的IRQ，这能够描述一个PIC的上、下两级电路；

②书中对于“数据信号线”描述比较模糊，让我产生好奇……；后来通过如下电路图，可以发现，在CPU发出中断查询脉冲之后，8259A片便会把中断号提交到数据总线上，CPU便可获取；



## 三、程序设计创新点

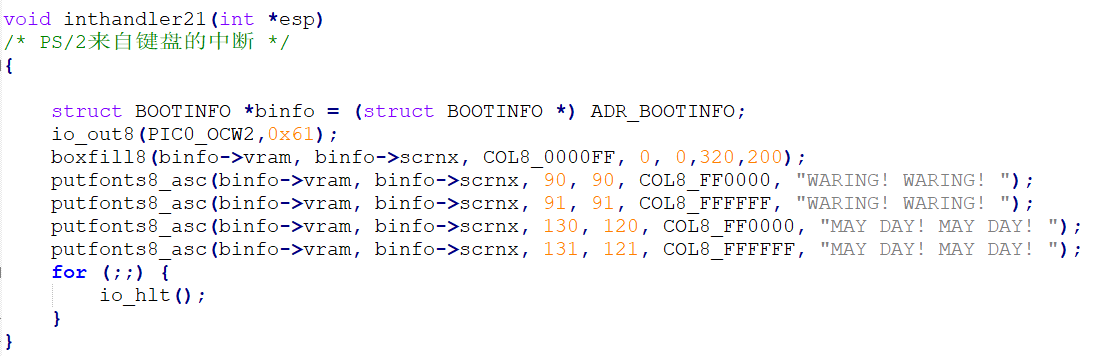
（一个比较鸡肋的“创新点”）

*由于本次实验学的中断处理很少，有关中断退出、获取数据等都在Day 7中，所以代码可修改处不多……*

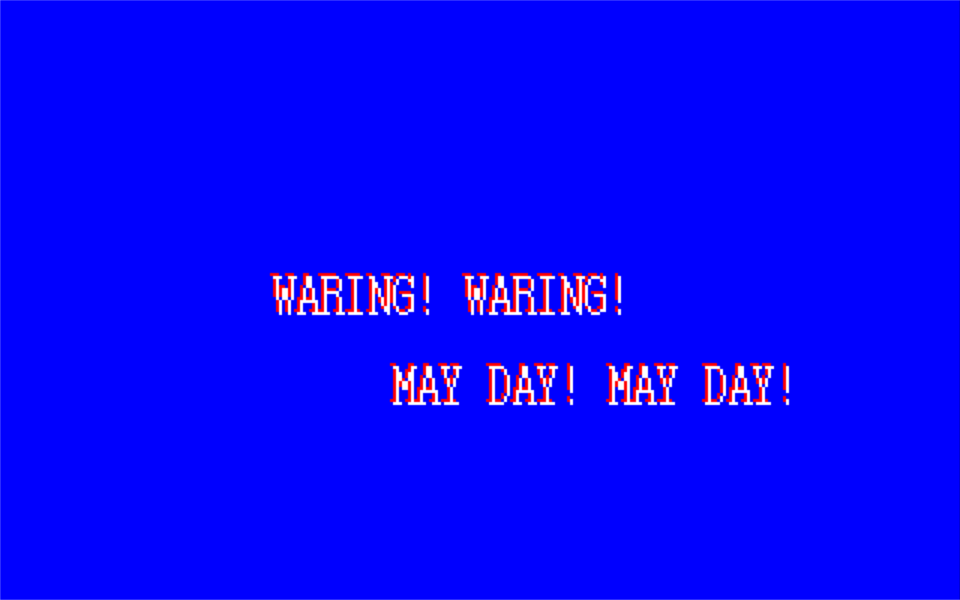
1、修改中断处理函数

创新点：基于Day 6对中断处理函数的学习，对现有的键盘中断处理函数进行修改，原有的中断函数是在接收到键盘的按下后，触发中断信号，然后在屏幕中输出一行信息，然后系统陷入停机状态，不再受理任何中断信息；

关键代码：



结果截图：



## 四、实验心得体会

这次试验的内容处于一个过度的阶段，首先是完成了Day 5留下的GDT和IDT问题，成功实现了中断，但是这个中断完善度比较低，仅仅只能够做到响应一次中断，不能够从中断中返回，暂时只能一直卡在人为设置的循环中。所以这次试验的创新点也很局限，许多需要的东西都要后面才讲，然后最终写了一个比较鸡肋的做法上去。希望Day 7能够实现这次想要做出来的东西。