戏说 BIOS

作者：Peter Hu

来源：http://www.ufoit.com/bbs/thread-672-1-1.html

整理：Daway Huang

发表日期：2009-07-06

整理日期：2009-11-10

目　　录

“我所知道的 EC”系列基本结束了，现在我终于可以有空玩一玩 BIOS 了。可是需要特别强调的是我是一名 EC 而不是 BIOS，所以我看 BIOS 的视角可能会不够专业，也不够正统，这也是为什么我将这个系列取名为”戏说 BIOS”的原因。可能有些朋友会觉得我不务正业，身为 EC 却去玩什么 BIOS(呵呵…我本来就是不务正业)，我觉得 PC 本身是一个非常复杂的系统，从经验来看很多问题都不是孤立的，通常会牵扯到很多的环节，因此如果将视野局限于自己的一亩三分地则很有可能见树不见林，看不清问题的本质。所以我觉得全面细致的理解系统的各个环节非常有必要;而且我的技术目标也是希望通过 n 年的努力能够贯通 PC 系统从 EC->BIOS->OS->DRIVER->APP 的整个链条，把握系统的运作的脉络。

1 戏说 BIOS 之 Hello BIOS

关于这个“戏说 BIOS”系列，我打算先练习一些 BIOS 新人学习作业，如：cmos dump，kbc access，pci scan，smbiosdump 等等；然后在对我感兴趣的一些 BIOS 的领域做一些 study。现在让我向 BIOS 世界打个招呼吧: “Hello BIOS,I am coming!”。

```
data segment
HelloBIOS db 'Hello BIOS,I am coming!$'
data ends

code segment
assume cs:code,ds:data

start:
mov ax,data
mov ds,ax
mov dx,offset HelloBIOS
mov ah,9
int 21h

mov ax,4c00h
int 21h
code ends
end start
```

2 戏说 BIOS 之 CMOS

2.1 Introduction

CMOS 全称为 complementary metal oxide semiconductor, 翻译成中文就是互补金属氧化物半导体，它是由一颗小的纽扣电池供电的 128/256 bytesram(现在的 chipset 通常提供 256 bytes 或者更大的空间)。它主要用于存放 RTC 以及一些 oem 的系统配置信息，所以除了 RTC 等部分其它的很多信息都是 undocumented& non-standard。RTC 标准的(documented&standard) ram bank 如下表 1 所示：

表 1

Index	Name
00h	Seconds
01h	Seconds Alarm
02h	Minutes
03h	Minutes Alarm
04h	Hours
05h	Hours Alarm
06h	Day of Wee
07h	Day of Month
08h	Month
09h	Year
0Ah	Register A
0Bh	Register B
0Ch	Register C
0Dh	Register D
0Eh-7Fh	114 Bytes of User RAM

2.2 Access Cmos

访问 cmos 通常是透过 70h,71h 这两个 IO port 实现的，有些 chipset 支援 256 bytes 的 cmos ram，访问 128bytes 以后的空间需要开启 chipset 的始能 register，有些 chipset 使用 72h，73h 访问扩展的空间如 intel chipset，有些仍然使用 70h，71h 如 sis chipset，因为这部分是非标准的，故后面的练习程序就不去读写这部分 ram space。读写 cmos 的过程非常简单，读特定的 index 的内容只需要将 index 送给 70h，然后就可以从 71h 读出对应的数据，具体过程如下述 code 所示：

```
;--------------------------------------------------------------------------
;read_cmos
;read the contents of a specific CMOS register
;call with
:al = CMOS address to read
;returns
:ah = Contents of register
;used registers: ax
;--------------------------------------------------------------------------
read_cmos proc near
clior   al,80h      ;disable NMI

out    70h, al
call   io_delay
in     al, 71h
```

```
call    io_delay
mov   ah, al
xor     al,al
out     70h,al      ;enable NMI
sti

ret
read_cmos endp
```

写操作和读类似，只是要将待写入的数据送给 71h 即可代码如下所示：

```
;--------------------------------------------------------------------------
;write_cmos
;write the contents of a specific CMOS register
;call with
:al = CMOS address to write
;ah = Contents of register
;returns:NULL
;used registers: ax
;--------------------------------------------------------------------------
write_cmos proc near
clior   al,80h      ;disable NMI
out     70h,al
call    io_delay
mov   al,ah
out     71h,al
call    io_delay
xor     al,al
out     70h,al      ;enable NMI
sti

ret
write_cmos endp
```

另外有些细节需要注意的是：

a. 读写过程中都需要关掉中断，以防止中断处理程序访问 CMOS，以及 RTC 更新过程中可能会导致并发访问。

b. NMI(non-maskable interrupt)是一种中断向量为 2 的中断，但是与常规中断不同的是它不能通过 mask register 屏蔽掉，而且 sti，cli 指令也对它无效；NMI 通常用于一些无法恢复的硬件错误，访问 CMOS 时也可能产生 NMI，所以需要关掉。NMI 可以通过 70h bit7 做开关。

c. 状态寄存器 A bit7 记录了 RTC 是否正在更新，如果正在更新则等到更新结束再去读 RTC（我写的 cmosdump 因为偷懒没有检查这一个 bitJ）。

2.3 Msg Based Event Driven

知道了以上的知识，我就有能力写一个类似 RU 中 dump cmos 的工具了。图 2.1 就是我写的 cmosdump：

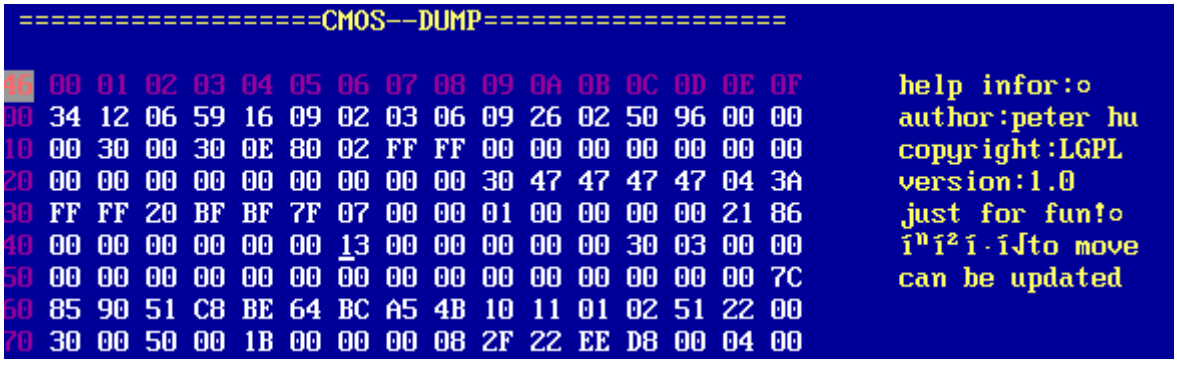

图 2.1

我觉得访问 cmos 本身并不困难，画个 UI 倒是挺费劲的，一个劲 call vbios。另外在完成这支 tool 的过程中我更深刻的体会到知识是相通的了，windows 编程的经验在这里发挥了优势，为了能够动态更新，实时修改，我就借鉴了 windows 下的“基于消息，事件驱动”的机制 Mainloop->GetMsg->TranslateMsg->DispatchMsg 一路下来好不快活！这部分的代码如下所示：

```
mainloop:
call    show_index
call    show_cmos

input_msg:
mov   ah,0
int     16h
cmp   ah,01h     ;esc
je      exit

cmp   ah,48h     ;up arrow
je      up

cmp   ah,50h     ;down arrow
je      down

cmp   ah,4bh     ;left arrowje left
cmp   ah,4dh     ;right arrow
je      right

call    input_byte
cmp   bl,1
jne     msg_loop
mov   ch,ah

mov   ah,0
int     16h
cmp   al,0dh             ;enterje enter
jmp     msg_loop

enter:
call    get_index
mov   ah,ch
mov   al,INDEX
call    write_cmos

msg_loop:
jmp     mainloop

up:
cmp   ROW,MINROW
jbe     roll_up
dec     ROW
jmp     bypass_up

roll_up:
mov   ROW,MAXROW

bypass_up:
call    set_cursor
jmp     mainloop

down:
cmp   ROW,MAXROW
jae     roll_down
```

```
inc     ROW
jmp    bypass_down

roll_down:
mov   ROW,MINROW

bypass_down:
call    set_cursor
jmp    mainloop

left:
cmp   COL,MINCOL
jbe     roll_left
sub    COL,3
jmp    bypass_left

roll_left:
mov   COL,MAXCOL

bypass_left:
call    set_cursor
jmp    mainloop

right:
cmp   COL,MAXCOL
jae     roll_right
add    COL,3
jmp    bypass_right

roll_right:
mov   COL,MINCOL

bypass_right:
call    set_cursor
jmp    mainloop

exit:
call    clr_screen
mov   ax,4c00h
int     21h
```

以上就是 cmosdump.exe 的核心架构 ，完成以后觉得使用 asm 好别扭啊，可能是 c/c++写的太多了，有点适应不过来了，以后还是要多写 asm，增强驾驭 asm 的能力，让我的 asm 和 c/c++一样熟练。最后开放 cmosdump.exe 完整的 source code 供有兴趣的朋友参考，source code 和可执行文件在附件下载。

附件：cmosdump.rar (3.56 KB)

3 戏说 BIOS 之 Keyboard

3.1 Introduction

Keyboard System 看起来好像挺简单，但事实上它远比想象中的复杂，硬件上 Keyboard System 需要两颗 cpu 完成 key stroke 的转换以及和 Host 的通信过程，一颗用于处理 keyboard 的 make&break 过程，另一颗作为 keyboard controller 和 host 交换信息。一次按键过程在软件的层面上也要经过多次转化才能成为最终被用户理解的 ASCII 码。这个过程通常需要经历 ma=>mv=>set2=>Set1=>ASCII。Keyboard System 的架构框图如图 3.1 所示：

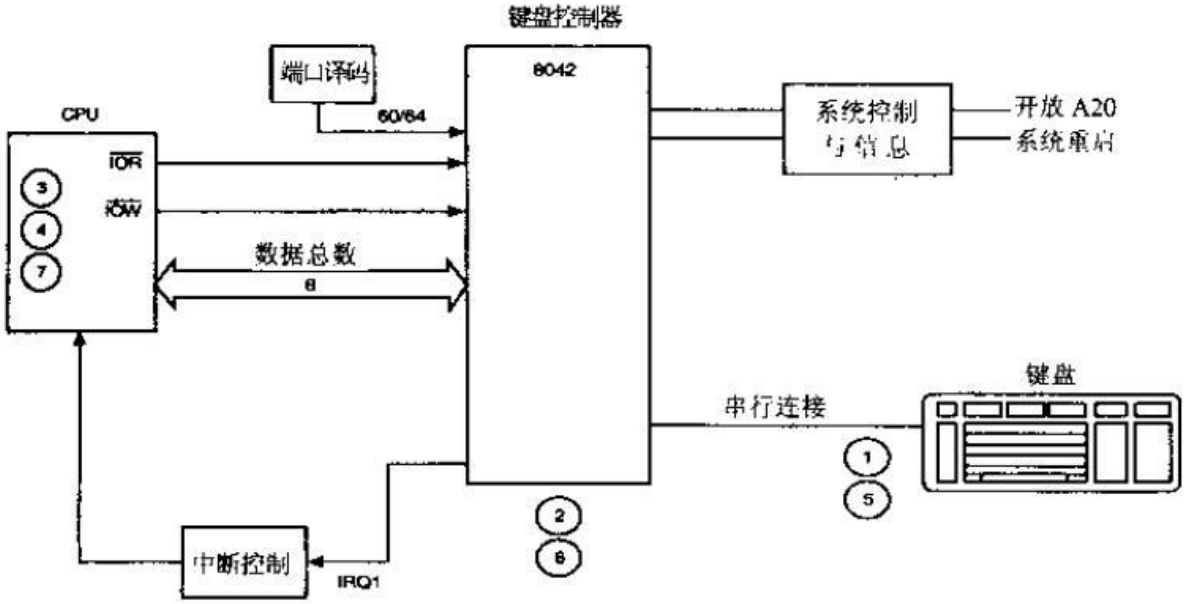

图 3.1

这是 MB 中常见的架构，在 NB 中这部分已经被放入 EC 之中成为 EC 的一个部分 KBC，但是工作原理依旧如此。

3.2 How It Works?

那么当我们按下一个键，需要做哪些动作，才能让我们看到最终的字符呢？听我慢慢道来。

当我们按下一个键‘k’时(make)，键盘内部的 8031 会将 k 的 set2 scan code‘2Ch’ 通过上图 1 的串行连接送给 8042，8042 会查一张 set2 转 set1 的表将该 set2 scan code 转成 set1 的‘14h’，而且 8042 会引发 IRQ1 通知 host，表示有按键事件发生。Host 将会读取 60Port 获取 set1 的 scancode‘14h’，而后 host 会将‘14h’转化为 ASCII 码‘k’；当我们松开一个键时，过程同按下比较像了，不过键盘内部的 8031 会先送‘F0h’，然后再送‘2Ch’给 8042，8042 看到‘f0h’会将 Set1 的‘14h’的 bit7 设置为 1 即 94h，以表示这是一个 break。Host 端也会收到中断 IRQ1，可是 host 通常不处理 break code。Make&Break key 也被称之为通码和断码。最终 host 会将 set1 以及 ASCII 码放在 BDA 之中。

Host 端对于键盘系统处理分为以下几类：

a.字符键

b.功能键

c.控制键

d.双态键

e.特殊功能键

对于这几种不同按键 host 处理方式也会有所不同。对于 c&d host 会在 BDA 中置 flag；对于 a host 会保存 set1 和 ASCII 码在 BDA 之中（大小写根据控制键的 flag 确定）；b 会影响到 set1 的值；对于 e host 可能会通过中断调用相关的 function。

图 3.2 显示 host 的处理流程：

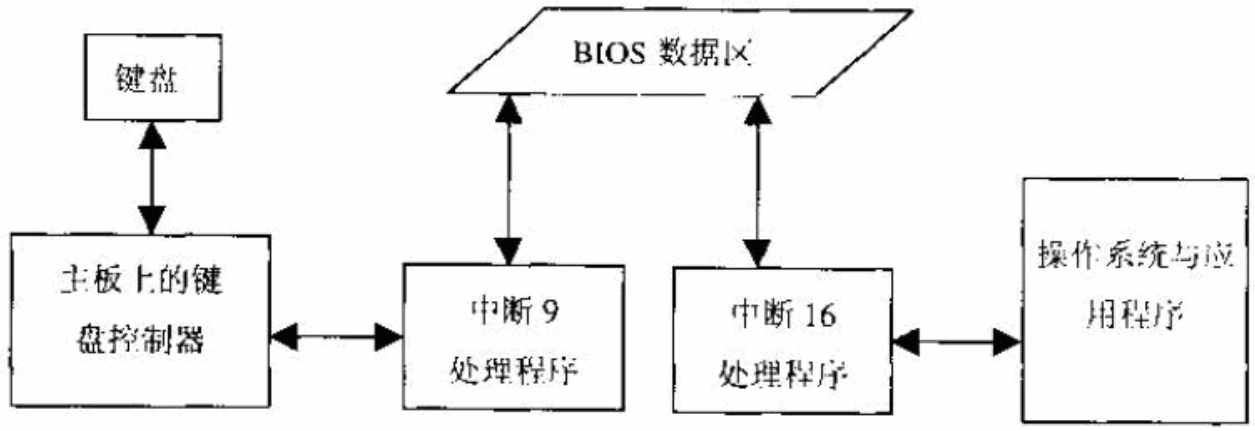

图 3.2

接下来我将分别用 C 和 ASM 演示 BIOS 处理 keyboard system 的大致过程，代码的原理是通过 hook int9 接管 BIOS 的中断处理过程，然后读取 EC 的 6064 port 获得 kbc 的 data 和 status，并转为 ASCII 码显示出来。有一个要注意的地方就是 EOI，EOI 是特指 8259 或者 8259 兼容设备的中断清除指令，需要在中断服务程序结束之前向 8259 发送 EOI 指令。如果在中断程序一开始就发送 EOI 指令的话，中断服务程序一旦比较大，运行时间较长，可能会产生中断嵌套，严重会造成死机。如果中断服务程序结束之后还没有发送 EOI 指令的话，那么以后将屏蔽该 IRQ 以及优先级低于该 IRQ 的所有中断。我最初就没有送 EOI，害得我调试了好久。

C 代码如下所示：

```
#include <dos.h>
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <bios.h>

void interrupt new_int9_handler();          /* interrupt prototype */
void interrupt (*old_int9_handler)();       /* interrupt function pointer */

unsigned char ESC_Press_Flag = 0;
unsigned char fifo[0x10]={0};
unsigned char start=0;
unsigned char stop=0;

int main(void)
{
    printf("Used to test keyboard set1 scancode\n");
    printf("@author:peterhu\t\t@Version 1.0\n");
    printf("Copyright(C) LGPL\t[ESC] to Quit\n");

    /* store old interrupt vector */
    old_int9_handler = getvect(9);
    /* set up new interrupt handler */
    setvect(0x09,new_int9_handler);

    while(1){
        if(ESC_Press_Flag)
            break;
        while(stop != start){
            printf("[%.2x],",fifo[stop]);
            stop = (++stop)%0x10;
```

```
            }
        }
        setvect(0x09,old_int9_handler);
        clrscr();
        return 1;
}

void interrupt new_int9_handler()
{
        unsigned char status;
        unsigned char set1;
        disable();
        status = inportb(0x64);
        if(status & 0x01){
                set1 = inport(0x60);
                fifo[start] = set1;
                start = (++start) % 0x10;
                if(set1 == 0x01)
                        ESC_Press_Flag = 1;
                //printf("[%.2x],",set1);
        }
        outportb(0x20,0x20);
        enable();
}
```

ASM 代码如下所示：

```
data segment
SET1       db   0
H2A        db   ['0','0',']','$'
MSG        db   'Used to test keyboard set1 scancode',0Ah,0Dh,'@author:peterhu',09h,09h,'
                @Version1.0',0Ah,0Dh,'Copyright(C) LGPL',09h,'[ESC] to Quit',0Ah,0Dh,'$'
OLDINT9 dd   0
ESCPR      db   0
data ends

code segment
assume cs:code,ds:data

start:
mov  ax,data
mov  ds,ax
mov  ax,1ch

call   hex2asi
call   show_set1
call   show_title
call   back_int9
call   install_int9

l0:
xor    cx,cx
mov  cl,SET1
jcxz   nokey
xor    ax,ax
mov  al,SET1

call   hex2asi
call   show_set1
mov  SET1,00h
```

```
nokey:
mov  cl,ESCPR
jcxz  l0
call   restore_int9

mov  ax,4c00h
int     21h

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;sub routine for store and show set1 scancode
;;for keybord strok maybe something error :/
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
int9_handler:
cli
in          al,64h
and         al,01h
cmp         al,01h
jne         exit9
in          al,60h
mov         ah,00h
cmp         al,01h
je          escp
jmp         exit9

escp:
mov         ESCPR,01h

exit9:
mov         SET1,al
mov         al,20h
out         20h,al
sti
iret

back_int9:
push        ax
xor         ax,ax
mov         es,ax
mov         ax,es:[24h]
mov         word ptr OLDINT9,ax
mov         ax,es:[26h]
mov         word ptr OLDINT9+2,ax
pop         ax
ret

restore_int9:
push        ax
xor         ax,ax
mov         es,ax
mov         ax,es:[24h]
mov         ax,word ptr OLDINT9
mov         ax,es:[26h]
mov         ax,word ptr OLDINT9+2
pop         ax
ret

install_int9:
push        ax
push        ds
push        bx
```

```
mov     ax,0
mov     ds,ax
mov     bx,24h
cli
mov     word ptr[bx],offset int9_handler
mov     word ptr[bx+2],seg int9_handler
sti
pop     bx
pop     ds
pop     ax
ret

show_title:
push    dx
mov     dx,offset MSG
mov     ah,9
int     21h
pop     dx
ret

show_set1:
push    dx
mov     dx,offset H2A
mov     ah,9
int     21h
pop     dx
ret

hex2asi:
push    dx
push    cx
push    si
push    bx
mov     bx,0
mov     si,offset H2A
mov     byte ptr [si+1],'0'
mov     byte ptr [si+2],'0'

ha1:
mov     cx,10h
mov     dx,0
div     cx
mov     cx,ax
jcxz    ha3
cmp     dx,10d
jnb     ha2
add     dx,30h
push    dx
inc     bx
jmp     short ha1

ha2:
sub     dx,10d
add     dx,'A'
push    dx
inc     bx
jmp     ha1

ha3:
cmp     dx,10d
```

```
jnb     ha4
add     dx,30h
push    dx
inc     bx
mov     cx,bx
jmp     ha5

ha4:
sub     dx,10d
add     dx,'A'
push    dx
inc     bx
mov     cx,bx
jmp     ha5

ha5:
pop     ax
mov     [si+1],al
inc     si
loop    s4

ok:
pop     bx
pop     si
pop     cx
pop     dx
ret

code ends
end start
```

上述程序运行状况如图 3.3 所示，一旦有按键动作该程序就会显示 set1 的 scancode(没有给出相应的 ASCII 码，凑合着用吧)上述代码可以在附件下载。

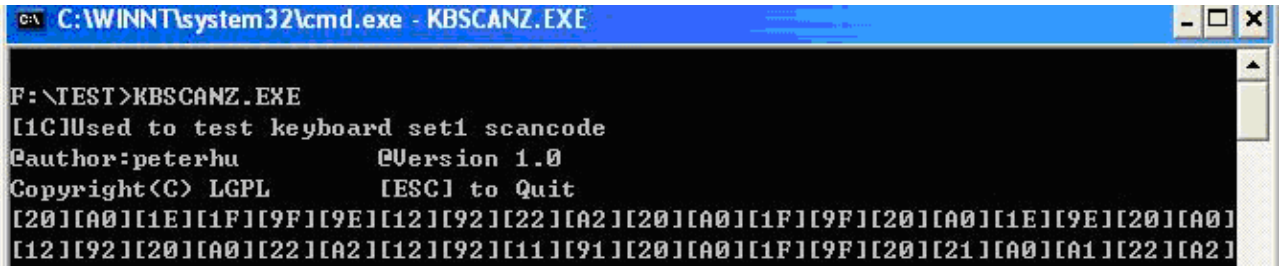

图 3.3

REFF:

<<The Undocumented PC>>

附件：KeyboardTest.rar (2.07 KB)

4 戏说 BIOS 之 Beep

4.1 Introduction

大凡用过电脑的朋友都应该听到过 BIOS 的报警声，有时 PC 开机的时候就会听到嘀的一声，有过修理 PC 经验的话就更清楚了“一短内存刷新失败，二短内存校验错误，一长三短内存错误，一长八短显示错误”等等诸如此类，可能各家的 BIOS 定的规则不同，但目的都是通过报警音获悉系统运行状况，找出病灶对症下药（有点像中医诊断中“望闻问切”中的闻）。

4.2 8253/8254

Beep 声我们都听过，但是有没有想过这是怎么实现的呢？带着疑惑随我开始探索之旅。提到这个 Beep，它可算是历史悠久了，追溯到 IBM 的第一台 PC，那时工程师们可能觉得 pc 功能太过单调枯燥，于是他们就祈求上帝给我点声音吧，于是声音就有了。他们将一个简单的扬声器加入了最初的 pc 硬件之中。光有 speaker 肯定是不足以产生音乐的，因为音乐得有音调和节奏才能组成。虽然我不懂音乐，但是我知道一点就是声音的高低和频率有关，所以还要有能制造频率的东东这就是 8253/8254。既然提到我就大致的讲讲 8253/8254，8253/8254 是可编程的定时器，8254 是 8253 的增强版本差异主要在可以外接 clock 频率不同，其实使用上无差了。8253 有三个独立工作 16 位的计数器 t0、t1、t2 分别使用 40h、41h、42h port 去操纵，除此之外还有一个 43h port 用于设定控制字。三个计数器分别编程，但是在使用之前必须先配置控制字，控制字主要用于选定哪一个计数器，选择计数器的工作模式等。控制字的格式如下表 4.1 所示：

表 4.1

Bit 0	计数值格式 0 表示 binary；1 表示 bcd
Bit1~3	模式选择
Bit4~5	读写指示
Bit6~7	选择计数器

其实这三个计数器在 PC 内部已经规划好了功能，基本上不需要用户参与了。

t0：用于系统时钟提供定时基准，它的输出端与中断控制器的 IRQ0 相连。t1：用于 DRAM 更新的信号，每隔 15.2us 刷新一次。t2：用于控制扬声器发声，作为 speaker 的音频频率。

所以我们知道 t2 用于提供 speaker 的音频频率，驱动 speaker 发生。这个部分早期驱动电路如图 4.1 所示：

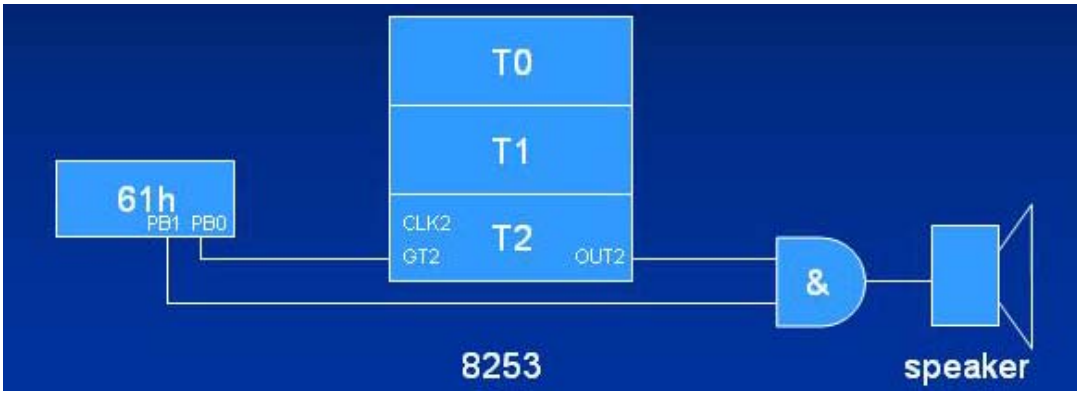

图 4.1

由图 4.1 我们可以看到 61h PB0 控制 T2 的 gate2，也就说只有将 PB0 pull high T2 才能工作。另外 PB1 与 T2 的输出端 OUT2 经过一个与门运算然后再驱动 speaker，所以 PB1 也要 pull high 这样 T2 的输出就可以操纵 speaker 的频率了。图中的 61h 是没有介绍过的，那就再来聊聊 61h port。61h 在 XT 系统中集成在 8255 之中，8255 是一颗可编程的外围接口芯片，61h 对应 8255 的 port B，它是一个 8 bit 的 IO port，每一个 bit 代表的意义如表 4.2 所示：

表 4.2

Bit0	t2 gate2 控制位
Bit1	Speaker 控制位
Bit2	DIP 相关
Bit3	录音马达
Bit4	RAM 同步更新检查位
Bit5	I/0 通道检查
Bit6	Keyboard 电平控制
Bit7	Keyboard 使能控制

AT 以后 8255 已不再使用，port61h 也使用别的 IC 代替了，但是它的主要的 bit 功能还是保留了下来，所以仍然可以使用 I/O 指令读写 61h port。

4.3 Beep~~~~~~

知道了以上的知识，我们就来写一个“一长三短的内存错误”的报警声玩玩咯。需要做的工作有三个：

a.通过操作 61h port 使能 speaker input 和 t2 gate2；

b.操作 8253 控制 beep 音的音调;

c. 音调保持一定的时间（也就是声音的长短）。

我们逐个的实现上述功能。

a 最简单，只要将 61h port 的 bit0&1 pull high/low 即可使能或者禁止。代码如下所示：

```
;--------------------------------------------------------------------------
;speak_set
;en/dis speaker input control&t2 gate2 control
;called with:cx
;used registers: ax
;--------------------------------------------------------------------------
speak_set proc near
push  ax
in    al,61h
jcxz  se_d
jmp   se_e

ss_d:
and   al,0fch
jmp   se_done

ss_e:
or    al,03h

ss_done:
out   61h,al

pop   ax
ret
speak_set endp
```

b 就需要设置 8253 计数器 2 的模式工作频率。操纵 8253 的步骤为：先向 43h port 选择所要使用的计数器以及工作模式参数类型等，然后再向 42h port 装入 t2 的计数初始值。代码如下所示：

```
;--------------------------------------------------------------------------
;t2_set;enable t2 & set work mode & out 2 frequency
;called with
```

```
:di(frequency demanded)
;used registers:ax,dx
;--------------------------------------------------------------------------
t2_set proc near
push  dx
push  ax
mov   al,0b6h           ;t2 lsb,msb,mode 3,binary
out   42h,al
mov   dx,12h
mov   ax,348ch
div   di
out   42h,al
mov   al,ah
out   42h,al

pop   ax
pop   dx
ret
t2_set endp
```

c 可以通过执行 loop 达到延时的目的，可是 loop 延时和处理器的类型频率有关，不同种类的 cpu 执行同样指令所需的时钟周期不同，就算相同种类但是主频不同的 cpu 要达到同样的延时效果计数的基准也会不同。那么有没有精确延时的方法呢？书上给出的答案是通过检测 61h port 的 bit4 ram 刷新检查位，每隔 15us 该 bit 会发生一次变化，所以检测它可以获得比较精确的时间（我猜测这个 bit 会和 8253 t1 同步变化，因为 t1 的输出脉冲用作 DRAM 的刷新定时信号，而该信号要求 15us 刷新一次）。延时的代码如下所示：

```
;--------------------------------------------------------------------------
;delay
;delay time base on 15us unit
;called with:cx (counts of time unit)
;used registers:ax
;--------------------------------------------------------------------------
delay proc near
push  axdloop:
in    al,61h
and   al,10h
cmp   al,ah
je    dloop

mov   ah,al
oop   dloop

pop   ax
ret
delay endp
```

以上就是 beep 的主要代码了，最后开放完整的 source code 供有兴趣的朋友参考。

REFF：

PC硬體元件控制詳解

《IBM-PC 汇编语言程序设计》

附件：beep.rar (945 Bytes)

5 戏说 BIOS 之 PCI Scan

5.1 Introduction

PCI 由 intel 公司在 1990 年前后开发的，后续经过若干年的发展以及标准化，它已然成为 server&pc 上的标准总线。PCI 以其出色的设计以及不错的通信速率在计算机领域攻城掠地，不断的取代诸如：MCA，ISA，EISA，VESA，NuBus 等传统总线。PCI 相对于传统总线有非常多的优点，如：

1，它是数据总线和地址总线分时复用的。这样减少了 pin 脚，节省了空间，而且这样也可以方便实现突发式数据传输。

2，它是即插即用的(plug & play)。当 device 插入系统时，系统会自动对 device 进行资源分配，并加载对应的 driver，而传统的 ISA device 则需要做复杂的手工配置。

3，中断共享。传统的总线有一个致命的缺陷就是它们是中断独占的，本来系统的中断就非常紧缺，所以增加新的 device 会出现中断不够使用的麻烦。而 pci irq routing 机制使得不同的 device irq 共用成为现实。可是技术的发展总是长江后浪推前浪，前浪死在沙滩上! PCI 又逐渐被更好的总线 PCIE 所取代而渐渐退出 PC 的历史舞台，后续我会再去研究一下 PCIE。

5.2 PCI Arch

可能是软体背景的原因，因此我看 PCI spec 也会习惯性的使用软件设计的视角去理解 PCI 的设计(我觉得有关设计、架构的理论应该是相通的，正如软件中经典的 design pattern 的思想来源于建筑学一样)。我的视角里 PCI 同经典的接口编程或者插件式设计非常接近。

接口本质上是一组规则的集合，它是对同类事物行为上的表示，它的主要目标是实现相同类别的不同对象行为上的多态性。面向接口的编程是 OO 思想的精髓所在，它的好处体现在哪里呢？

首先它增强了系统的灵活性，只要遵循接口定义的规则，系统的底层实现部分就可以灵活的替换、扩充。如：PCI 总线定出了设备的统一的硬件接口，这样遵循该接口的 pci device 就可以方便的扩展入系统；另外相同的接口可以接入不同厂家的设备，就像同样的 sata 接口可以接三星的光驱也可以接 LG 的。

其次，规则给出以后，实现该接口的部件就会有共同的接口但是不同的实现，如此系统端就可以通过接口灵活实现对部件的操作配置。PCI 定义出了三种规格的配置空间，根据配置空间提供的信息系统端可以方便的识别设备的种类，功能甚至于厂商和版本号，获得非常丰富的系统端知识；而且该功能也使得设备可以动态的配置资源进而能够做到 plug & play。

5.3 PCI Scan

PCI Configuration Space 是大小为 256 字节的一块空间，它由 header 和 device specific 两部分组成。其中 header 部分是固定的，device specific 部分则是与 device 相关的，不同的 device 会有不同的 layout。配置空间被用于配置、初始化以及灾难性错误处理的功能。图 5.1 是 type 00h Configuration Space Header：

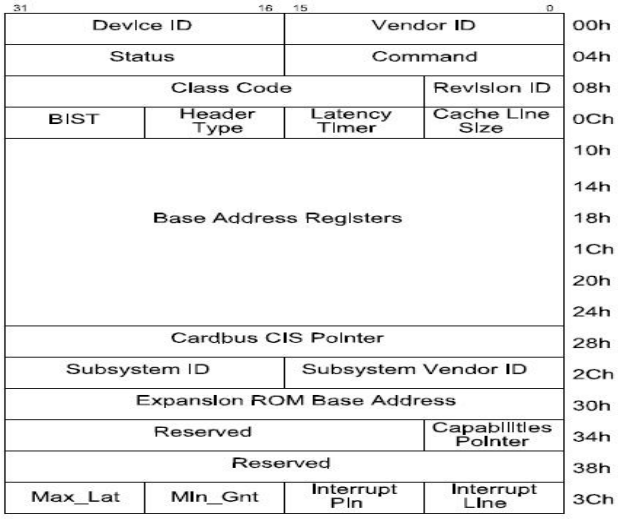

图 5.1

PCI Scan 的重要任务就是读出该 256bytes 配置空间，那么如何读取这部分的信息呢?

有下述两种方法：

1.使用 0CF8-0CFB, 0CFC 这两组 IO port 存取 PCI Configuration Space。将总线号、设备号、功能号和寄存器号组合成一个双字送到配置地址端口(CF8H-CFBH)，然后读写配置数据端口 (CFCH)即可获得配置空间的数据，图 5.2 是配置地址寄存器的格式定义：

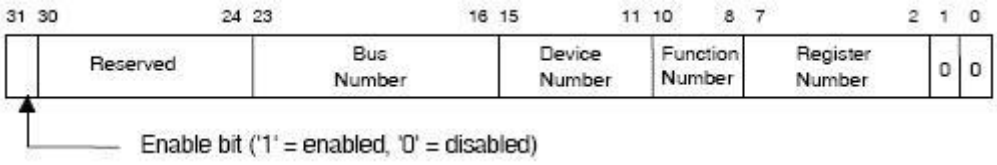

图 5.2

所以我们先要 build 一个 config-address 然后再去透过端口存取配置空间。下述代码用于 build config-address：

```
;---------------------------------------------------------------------------
;build _pci_cfg_add:
;build pci config address
;used registers:eax,ebx
;---------------------------------------------------------------------------
build_pci_cfg_add proc near
push  eax
push  ebx
xor    eax,eax
xor    ebx,ebx
mov   PCI_CFG_ADDRESS,80000000h
mov   al,PCI_BUS_NUM
shl    ax,08h
mov   bl,PCI_DEV_NUM
shl    bx,03h
or     ax,bx
or     al,PCI_FUN_NUM
shl    eax,08h
or     PCI_CFG_ADDRESS,eax

pop   ebx
pop   eax
ret
build_pci_cfg_add endp
```

config-address 准备好以后接下来就是透过 IO port 读取 pci configuration space 了，下述代码演示读取的过程：

```
;---------------------------------------------------------------------------
;read_cfg_space
;read pci config space use io port
;Called with:NULL
;used registers:eax,edx
;returned regs:eax
;---------------------------------------------------------------------------
read_cfg_space proc near
mov   eax,PCI_CFG_ADDRESS
or     eax,edx
mov   dx,PCI_CFG_APORT
out    dx,eax
mov   dx,PCI_CFG_DPORT
in     eax,dx

ret
read_cfg_space endp
```

理论上 PCI bus 支持 256 条总线，每条总线支持 32 个 device，每个 device 又支持 8 个 function，所以我们组合出上面所有的可能就可以遍历出所有的 PCI 设备了。可是实际上 PC 上面 PCI 总线通常只有 1 条，最多也不会超过四条所以我们只扫 4 条总线就可以了，不用做太多的无用功。有了前面的准备，我们就来实现一个类似 RU 中的 PCI scan 吧。图 5.3 就是我写的 pciscan 运行的状况了：

BUS.N	DEV.N	FUNC.N	D.ID	V.ID	L.N	P.N
00	00	00	7910	1002	00	00
00	01	00	7912	1002	00	00
00	04	00	7914	1002	FF	00
00	12	00	4380	1002	0B	01
00	13	00	4387	1002	05	01
00	13	02	4389	1002	0A	03
00	13	03	438A	1002	0F	02
00	13	05	4386	1002	0A	04
00	14	00	4385	1002	00	00
00	14	02	4383	1002	05	01
00	14	03	438D	1002	00	00
00	14	04	4384	1002	00	00
00	18	00	1100	1022	00	00
00	18	01	1101	1022	00	00
00	18	02	1102	1022	00	00
00	18	03	1103	1022	00	00
01	05	00	791F	1002	0A	01
01	05	02	7919	1002	0A	02
02	00	00	8136	10EC	05	01

图 5.3

在该界面下按下 esc 就会退出该程序；移动 ↑ ↓ 键就可以选中 device，然后敲 enter 就会看到该 device 的 Configuration space 如图 5.4 所示：

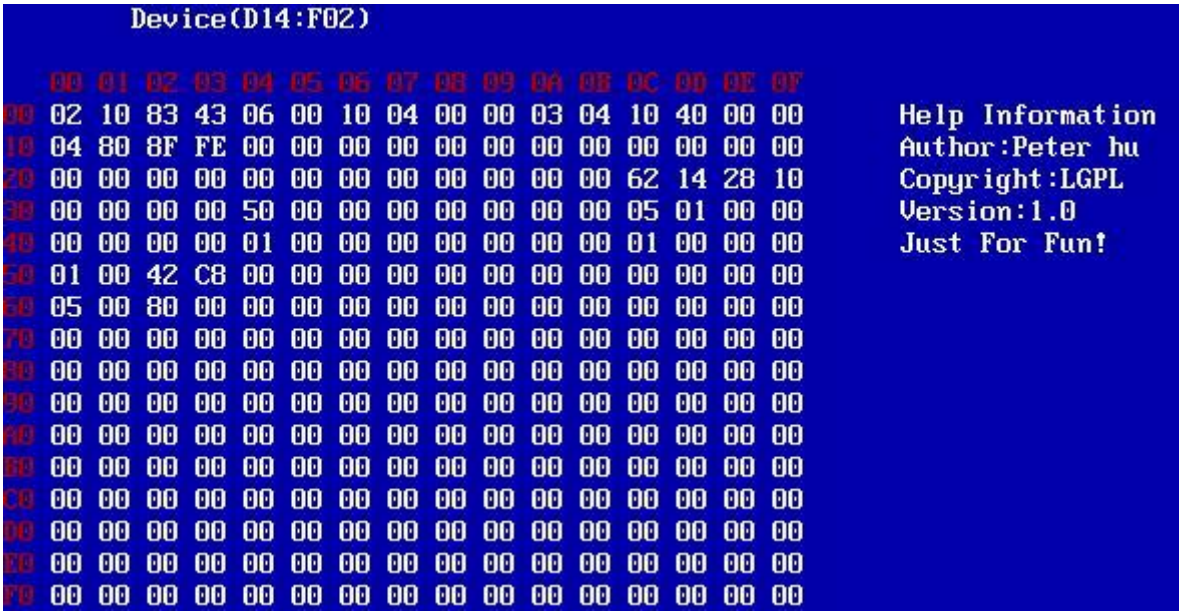

图 5.4

当前界面下如果想返回到上一个界面只需要按下 F6 就会回到图 3 的界面了。

2. Call PCI BIOS int1A 同样也可以获取 PCI device 的信息。其中 AH=B1h，AL=function id，所有的 function id 如下所示：

01h: INSTALLATION CHECK
02h: FIND PCI DEVICE
03h: FIND PCI CLASS CODE
06h: PCI BUS-SPECIFIC OPERATIONS
08h: READ CONFIGURATION BYTE
09h: READ CONFIGURATION WORD

0Ah: READ CONFIGURATION DWORD
0Bh: WRITE CONFIGURATION BYTE
0Ch: WRITE CONFIGURATION WORD
0Dh: WRITE CONFIGURATION DWORD
0Eh: GET IRQ ROUTING INFORMATION
0Fh: SET PCI IRQ
81h: INSTALLATION CHECK (32-bit)
82h: FIND PCI DEVICE (32-bit)
83h: FIND PCI CLASS CODE (32-bit)
86h: PCI BUS-SPECIFIC OPERATIONS (32-bit)
88h: READ CONFIGURATION BYTE (32-bit)
89h: READ CONFIGURATION WORD (32-bit)
8Ah: READ CONFIGURATION DWORD (32-bit)
8Bh: WRITE CONFIGURATION BYTE (32-bit)
8Ch: WRITE CONFIGURATION WORD (32-bit)
8Dh: WRITE CONFIGURATION DWORD (32-bit)
8Eh: GET IRQ ROUTING INFORMATION (32-bit)
8Fh: SET PCI IRQ (32-bit)

我们使用 function id 09h 就可以从 configuration space 中读取出一个字，这样的操作明显简单的多了，只需 call 一次 int1a 中断即可。下述 c 代码演示了读取 Vendor id 的过程，如需读取其它部分只要算出具体 config-address 即可。

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>

int main(int argc,char** argv)
{
	union REGS reg;

	argc = argc;
	argv = argv;

	reg.x.ax = 0xB109;
	reg.x.bx = 0x80000000;
	reg.x.di = 0;

	int86(0x1A, ®, ®);
	if(reg.x.cx != 0xffff){
		printf("Vendor : %4.4X\n", reg.x.cx);
	}
	return 0;
}
```

程序运行结果如图 5.5 所示：

图 5.5

最后依旧是开放完整的 source code 和可执行文件供有兴趣的朋友下载。

附件：PCISCAN.rar (4.69 KB)

6 戏说 BIOS 之 Clock Generator

6.1 Introduction

Clock Generator 是主板上面一颗极为重要的 IC，说它极为重要一点都不为过，因为 Clock generator 负责提供主板上的 clock, 一旦 Clock Generator 除了问题，板子基本上就完蛋了。Clock generator 供给的 clock 部件有 CPU clock，PCI clock，AGP clock，PCIE clock，SATA clock，USB clock 等。

6.2 How to work?

Clock generator 是一颗 IC，它有一颗外接晶振，内部会有锁相环放大调整电路，可以将外接的晶振产生的 clock 放大调整然后再分频输出到各个外围器件和总线，提供器件和总线工作所需的 clock。Clock generator 的工作原理如图 6.1 所示：

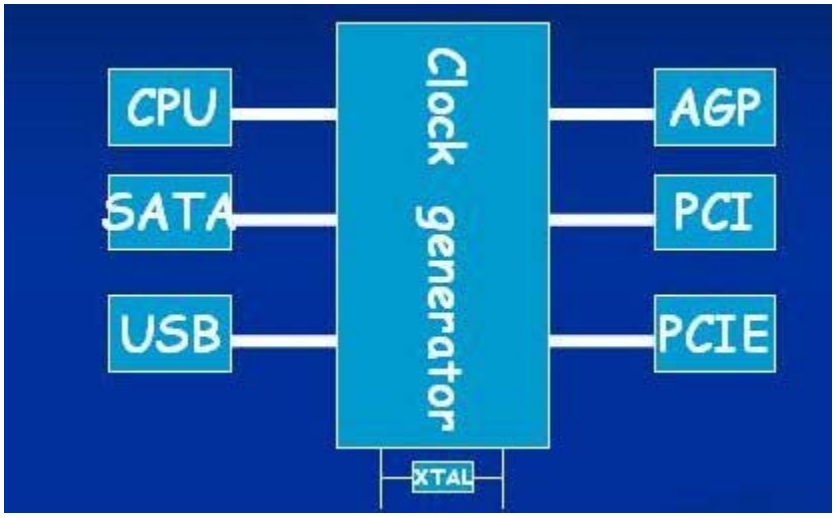

图 6.1

Clock generator 通常是一个 smbus device，接在 SB 的 smbus controller 上所以通过 SB 的 smbus controller，使用标准的 smbus protocol 就可以存取 Clock generator 上的 configure data 从而达到配置各个外围器件 Clock 的目的。鉴于 clock generator 的重要性，所以 BIOS 在非常早的阶段就会配置它(boot block 阶段)，让 CPU、memory 等的工作有一个稳定的 clock。下面我就以 VIA 平台为例演示 Clock generator 读取过程：a, 首先要通过 PCI 配置空间找到 SB smbus controller 的 base address。

b, 查看 Clock generator smbus slave address。ICS9UM700 slave address：42h

c, 透过 SB smbus controller 下达 slave address & protocol 完成 clock generator 数据的读写。

下述 code 演示了获得 smbus contoller base address 的过程：

```
;-------------------------------------------------------------------------------
;get smbus base address
;used registers:eax,edx
;called with:NULL
;-------------------------------------------------------------------------------
get_smbus_bar proc near
push  edx
push  eax
mov   dx,PCI_CFG_ADD
mov   eax,PCI_SMBUS_ADD
out   dx,eax
mov   dx,PCI_CFG_DAT
in    eax,dx
and   eax,0FFFEh
mov   SMBUS_REG_BAR,ax
```

```
pop   eax
pop   edx
ret
get_smbus_bar endp
```

下面的 code 演示使用 read block protocol 读取 clock generator configure data 的过程：

```
;-------------------------------------------------------------------------------
;read via clock gen data by read block protocol
;called with:NULL
;used registers: ax,dx,bx,cx
;-------------------------------------------------------------------------------
read_via_smbus_block proc near
push  dx
push  cx
push  bx
push  ax

call    get_smbus_bar

;reset host status registers
mov   dx,SMBUS_REG_BAR
or      dl,SMBUS_HSTS_REG           ;(00h)
mov   al,05eh
out     dx,al
;set smbus slave address
mov   dx,SMBUS_REG_BAR
or      dl,SMBUS_HADD_REG
mov   al,SLAVE_ADDRESS
or      al,01h
out     dx,al
call    io_delay

;clear smbus status
mov   dx,SMBUS_REG_BAR
or      dl,SMBUS_HSTS_REG           ;(00h)
mov   al,05eh
out     dx,al
call    io_delay

;clear smbus command byte
mov   dx,SMBUS_REG_BAR
or      dl,SMBUS_HCMD_REG   ;(03h)
mov   al,00h
out     dx,al

;block read protocol
mov   dx,SMBUS_REG_BAR
or      dl,SMBUS_HCTL_REG           ;(02h)
mov   al,54h
out     dx,al
call    io_delay

;wait for smbus finished
mov   dx,SMBUS_REG_BAR
or      dl,SMBUS_HSTS_REG           ;(00h)

rvsb_wait_smbus_fi:
in      al, dx
call    io_delay
test    al, 01h
```

```
jnz     rvsb_wait_smbus_fi

;read data count
mov   dx,SMBUS_REG_BAR
or      dl,SMBUS_HDA0_REG
in      al,dx

;read data from smbus block data
mov   cl,14h
mov   bx,offset REGSTR

rvsb_read_data:
mov   dx,SMBUS_REG_BAR
or      dl,SMBUS_BLKDA_REG
in      al,dx
call    hex2asc
dec    cl
inc     bx
inc     bx
cmp   cl,01h
jae     rvsb_read_data

pop    ax
pop    bx
pop    cx
pop    dx
ret
read_via_smbus_block endp
```

图 6.2 演示 ClockGen.exe 运行时的状况：

图 6.2

ClockGen.asm 只演示了 VIA platforma + ICS9UM700 的 dump 过程，至于其它的平台，有兴趣的朋友可以如法炮制。那么这只 tool 可不可以做成通用的呢?当然可以，只要从 PCI configure space 读出 Vendor id 识别出平台然后再根据具体的平台 spec 去获取 smbus controller bar（可能要多看几份 spec 了）然后就可以操作 smbus 了，clock generator slave address 可以让用户输入（我看到的几个都是 42h，但是也会有特例）。我比较懒，有兴趣的朋友可以试着完善它。最后依旧是开放完整的 source code 和可执行文件供有兴趣的朋友下载。

附件：ClockGen.rar (4.1 KB)