第 0 章　常驻(TSR)程序实作初步：使用 TURBO C

市面上有许多有趣的程序，如 Sidekick、抓图程序、DUMP 程序、查错工具、…，这些程序只要执行一次就会常驻于内存，以后随叫随到不需再由磁盘驱动器装入，所以反应速度很快、很受欢迎。这种会常驻于内存的程序，我们称为 TSR(Terminate & Stay Resident)程序。

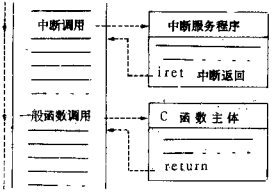
以往，TSR 程序大多为汇编语言所写成的，使用 C 语言的 TSR 程序就很少见了。我们曾在"C 语言实务"一书的最后一章介绍了一个简单的 TSR 程序，现在就以 C 语言为工具，由浅入深地对 TSR 程序来作一全面的讨论。

0.0　80×86 中断(Interrupt)提要

80×86 的中断原理在许多汇编语言的书籍中都有详细的说明，本节只针对和 TSR 程序有关的部分做概要的说明。

0.0.1　80×86 的中断(interrupt)机能

当中断信号出现时 80×86 CPU 便会'中断'(暂停)目前的工作而去执行中断信号所指定的程序，这种由中断所指定执行的程序叫做中断服务程序(Interrupt Service Routine 简称 ISR)。当 ISR 执行完毕，CPU 会回头继续执行刚才被中断的程序，所以 ISR 与一般的子程序有些类似，然而由于中断的特殊性质，ISR 和一般子程序也有些不同。基本上中断发生之初，标志会连同返回地址也推入堆栈，并且 ISR 是用 iret 指令来返回原程序，iret 会由堆栈取原标志值及返回地址。这是中断和一般子程序不同之处。



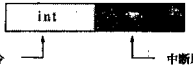
0.0.2　Interrupt 的来源

Interrupt 的来源有两种，一种来自硬件，一种来自软件。

硬件的中断信号可能来自 CPU 内部或由外部的外设发出。外部的硬件中断是指 CPU 外部，如：磁盘驱动器、打印机、键盘…等对 CPU 所发出的中断。我们将介绍的键盘、时钟中

断就是典型的硬件中断。

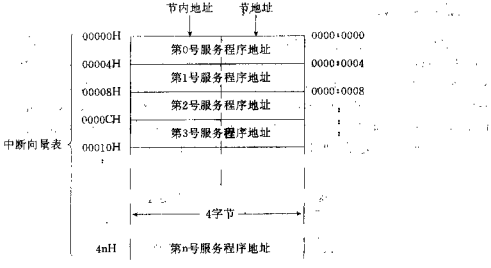
至于软件中断，则如一般指令一样，是用指令的形式存在程序里的。80×86 的中断指令格式如下：



80×86 是用编号来指定中断的服务程序，例如 int 21h 就是指定 CPU 暂停(中断)目前工作而去执行第 21h 号中断服务程序。

0.0.3 中断向量表

光是指定编号，CPU 是无法去执行程序的。必须给定服务程序的地址(称为中断向量)，CPU 才能到该地址去执行服务程序。为此，80×86 的设计者将 00000-003FF 的区间设为中断向量的存放区(称为中断向量表)，这个区间内存放着中断服务程序的进入地址，每一地址占 4 字节。所以第 0 号服务程序的地址在(00000-00003)、第 1 号服务程序地址在(00004-00007)、…第 n 号中断的服务程序地址是存于中断向量表第 4n 字节上。



你可以用 DEBUG 的 D 命令行出 0000:0000-0000:003F 的所有中断向量的内容观察一番。下面我们有一个程序也可以用来列出所有中断向量的内容。有关中断向量及其细节我们在以往的许多书籍中已有详尽的讨论了，相信读者已具备了相当的基础知识才是。此处我们就专注于和 TSR 程序有关的部分来研究。

0.0.4 PC 的硬件中断

下面是 PC 外设的中断编号，我们会在以后的例子中使用到：

表 0.0 外部的硬件中断安排

中断编号	外 设
08h	CLOCK
09h	键盘
0ah	无
0bh	COM2
0ch	COM1
0dh	LPT2
0eh	软盘
0fh	LPT1

由上表我们可以看到 clock 的中断编号为 08h,keyboard 的中断编号为 09h,…依此类推。硬件中断的编号之所以在 08h~0fh 是因为硬件线路的设定所致。基本上由 0~fh 的中断都可自由执行,并且,当我们说某个中断为硬件中断,是说该硬件设备可通过硬件线路来触发此中断,但并不表示此中断为该硬件唯一所有,我们一样可由软件程序来执行该中断服务。例如键盘被发的是第 09h 中断,但我们仍可由程序中执行第 09h 中断。也就是说中断服务程序是一样的,只是中断来源不同而已。在 CPU 上有一个特殊的中断叫 NMI,它就是一个纯硬件的中断,无法由软件来执行。

0.0.5 中断标志

当中断信号发生时,CPU 会视当时 I(Interrupt)标志的值来决定是否执行中断服务,必须 I=1 CPU 才会执行中断服务程序,而当 I=0 时,即使有中断发生,CPU 也不加以理会。

0.1 用 TURBO C 来编写中断服务程序

中断服务程序和一般子程序类似,只不过中断服务程序是以 iret 指令来返回主程序,而一般子程序则用 ret 指令返回主程序。这是二者主要的差别。

0.1.1 interrupt 类型

TURBO C 提供了一种叫 interrupt 的函数类型,用 C 写成的 ISR 函数必须声明成 interrupt 类型,interrupt 类型的函数会在函数执行前后 push、pop 所有的寄存器,并且用 iret 指令返回原程序。你可以在 TCC 命令模式下指定-S 取得 interrupt 类型函数的目标代码,以观察 TURBO C 所生成的 interrupt 类型的函数和一般函数有何不同。

testa.c

```
void test1(void)
{
/* 一般函数 */
}
void interrupt test2(void)
{
/* interrupt 类型的函数 */
}
```

用 tcc -S testa 来编译 testa.c 这个程序，会生成一个叫 testa.asm 的汇编语言程序。这个 testa.asm 就是 testa.c 被 TURBO C 编译后的样子：

testa.asm

```
.....
.....
.....
test1   proc    near←-------test1 被声明为 near
@1:
;       ? debug L 3
        ret←---------使用 ret 返回
_test1 emdp
;       ? debug L 4
_test2 proc     far←--------test2 被声明为 far
        push    ax          }
        push    bx          }
        push    cx          }
        push    dx          }
        push    es          }
        push    ds          } 多了这些
        push    si          }
        push    di          }
        push    bp          }
        mov     bp,DGROUP   }
        mov     ds,bp       }
@2:
;       ? debug L 6
```

```
        pop     bp
        pop     di
        pop     si
        pop     ds
        pop     es          和这些
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        iret←--------使用 iret 返回
_test2 endp
.....................
.....................
        public  _test2
        public  _test1
        end
```

0.1.2　由 C 程序调用 interrupt 类型函数

如果 interrupt 类型的函数是以 iret 返回原程序,那么由一般 C 程序来调用 interrupt 类型的函数,岂不是会因 iret 多 pop 了 flag 寄存器的值而发生错误了吗?

这个问题可以由下列的实验来看出答案:

testb.c

```
void test1(void);
void interrupt test2(void);
main()
{
    test1();
    test2();
}

void test1(void)
{
    /* 一般函数 */
}
void interrupt test2(void)
{
    /* interrupt 类型的函数 */
}
```

用 tcc -S testb 编译会生成 testb.asm 的汇编语言程序。

testb.asm

```
.....
.....
.....
;
; void main(void)
;
                assume  cs:_TEXT
_main           proc    near
                push    bp
                mov     bp,sp
;
;               {
;               test1();
;
                call    near ptr test1
;
;               test2();
;
                pushf←----------多了 pushf
                call    far ptr_test2 ←--------采用 far call
;
;               }
;
                pop     bp
                ret
_main           endp
.....
.....
```

结果我们发现 main()在调用 interrupt 类型函数前，会先执行一个 pushf 指令，这样刚好和 iret 多 pop flag 的动作相抵消，所以由一般的 C 程序调用 interrupt 类型的函数就不会有问题了。但是，为什么我们要由一般的 C 程序中调用 interrupt 类型的函数呢？这种函数不都是做中断服务程序使用的吗？原因，我们会在下一节有关 ISR 的串联方式中加以说明。

0.1.3 setvect()与getvect()

当我们设计好一个 ISR(中断服务程序)后，接下来必须把中断向量指向 ISR 程序的进入地址。但中断向量的值是由 DOS、BIOS 所设定的，Microsoft 建议不要直接更改，否则系统可以会发生问题。在 TURBO C 函数库里有一对函数 setvect()及 getvect()，这两个函数可以很安全的读写中断向量的值：

getvect()　-读取中断向量值
头文件：#include<dos.h>
声　明：void interrupt(*getvect)(int intnum)；
执　行：ivect=getvect(intnum)
返回值：返回 intnum 所指的中断向量地址

说明

getvect(intnum)会返回第 intnum 号中断向量的值，此即 intnum 号中断服务程序的进入地址(4 bytes 的 far 地址)。该地址是以 pointer to function 的形式返回的。getvect()使用的方式为：

```
ivect=getvect(intnum);
```

其中 ivect 是一个 function 的 pointer，存放 function 的地址，其类型必须是 interrupt 类型。由于声明时，此 pointer 无固定值，所以冠以 void。

程序 0.0　vector.c 列出前 16 个中断向量值

```
#include<dos.h>
main()
{
    int i;
    for (i=0;i<=15;i++)
    printf("Vector:%2d=%Fp\n",i,getvect(i));
}
```

此例我们以 getvect(i)列出 i=0～15 共 16 个中断向量值(服务程序地址)。其中 getvect(i)返回的是一个 Far Pointer，所以 Printf 的打印格式必须用%Fp 表示。

你可以改一下程序，列出更多的中断向量来观察，并且可以在 DOS 下用 DEBUG 列出中断向量表的内容加以对比，你会发现在 TURBO C 下，某些中断向量的值和 DOS 下的值不同，虽然 TURBO C 是修改了一些中断。

setvect()——设定中断向量值
头文件：#include<dos.h>
声　明：void setvect(int intnum,void interrupt(*isr)())；
执　行：setvect(intnum,isr)；
返回值：void

说明

setvect(intnum,isr)会把第 intnum 号中断向量指向 isr 所指的函数，也即指向 isr()这个函数，isr()必须是一个 interrupt 类型的函数。

程序 0.1 reboot.c(更改中断向量的例子)

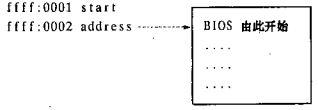
```
#include <dos.h>
void interrupt ( * reboot)();

main()
{
    printf ("Vector: %2d = %Fp\n",5,getvect(5));          /* 设定前 */
    reboot=MK_FP(peek(0xffff,0x0003),peek(0xffff,0x0001));
    setvect(5,reboot);  /* 把 int 05h 向量指向 reboot 地址 */
    printf ("Vector: %2d = %Fp\n",5,getvect(5));          /* 设定后 */
    puts("Press <shift><Prt Sc> OR <Prt Sc> to reboot\n");
    getche();
}
```

上列程序就是以 setvect()来设定 int 05h 的中断向量,以更换其 ISR 程序。此处我们以 ROM BIOS 的 ffff:0001h 地址的内含值做为 int 05h 的 ISR 程序起始地址,fff:0000h 是 BIOS 的进入地址,当 80×86 CPU 启动时,便把 cs 设为 ffff,把 ip 设为 0000,也即以 ffff:0000 做为 CPU 的启动地址。PC 的 BIOS 在此地址上存放它的第一道指令,内容是:

jmp start address → { ffff:0000 jmp
ffff:0001 start
ffff:0002 address

其中 jmp 指令本身占一个 byte,所以由 ffff:0001 地址开始存放的就是 BIOS 启动的进入地址:



jmp start address 就是重新执行启动的动作。现在我们把 int 05h 中断向量指向 start address,所以下一次按 Print-Screen 时,系统便重新启动而不是做 HARD COPY 了!

再看一个例子:

程序 0.2 service.c(自己设计 ISR 的例子)

```
#include <dos.h>
void interrupt isr0();

main()
{
    setvect(5,isr0);
    puts("Press <Prt Sc> to get SERVICE:\n");
```

```
    puts("Press other key to exit.\n");
    getche();暂时不要结束程序
}
void interrupt isr0()
{
    cputs("Yes! Sir! \n");
}
```

这个例子里我们用 C 语言自己来写 ISR 服务程序，我们写了一个叫 isr0()的函数，其类型指定为 interrupt，这样 TURBO C 就会自动把它编译成 ISR 的格式。我们在 main()函数中把第 5 号向量指向 isr0()，所以下一次按 PrtSc 键时，使会依第 5 号中断的指示去执行 isr0()而在屏幕上

显示"Yes! Sir!"的字符串了！

检讨

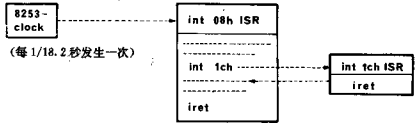
请注意！TURBO C 程序会在结束前自动恢复 int 0、int 1、int 5 等中断向量，所以如果你要在 TURBO C 的窗口模式下执行本程序，则一定要在程序最后执行一个 getche()函数，使程序暂缓结束，以免程序结束返回 TURBO C 后，我们所设的 int 05h 的中断向量又被 TURBO C 改回去了(TURBO C 改了很多向量)。

0.2　Clock 中断

前两个例子我们是按 PrtSc 键来启动 int 05h 的 ISR，只要更改 int 05h 的 ISR，按下 PrtSc 便可做出各种我们指定的动作来。按下 PrtSc 以执行 ISR 的过程，我们称为中断的触发。中断触发的方式很多，本节我们来介绍以 Clock 触发 ISR 的方法。

IBM PC 的计时 clock 每隔 1/18.2 秒会对 CPU 做一次 interrupt，我们也在表 0.0 看到 clock 所触发的是一个 int 08h 中断。

int 08h 中断主要是更新系统的计时、更新磁盘驱动器磁头离开磁盘的延时计数以及执行一个 int 1ch 中断。int 1ch 做什么呢？经我们观察，int 1ch 的 ISR 程序只有一句话：iret；所以 int 1ch 什么也不做。如果我们写一个 int 1ch 的 ISR 程序的话，这个 ISR 就会每隔 1/18.2 秒被执行一次。

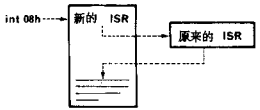


由上图我们可以看到 int 08h 是一个由硬件触发的中断，而 int 1ch 则是由 int 08h 的 ISR 程序中所执行的一个软件中断。

int 1ch 的 ISR 的 int 05h ISR 的写法相同，因此我们便不再多花篇幅来说明了。然而，若

是要写个 int 08h 的 ISR，情况就会比较难一点。

int 1ch 原本的 ISR 是一个空的服务程序，而 int 05h 的 ISR 则是将屏幕拷贝到打印机的程序，这些程序并无太大的重要性，因而我们可以用自己写的 ISR 来直接更换原有的 ISR。但是像 int 08h、int 09h、int 13h、…许多重要的中断服务程序是不能任意更换的。对于这些重要的中断，如果要更动其 ISR 的话，则要采用串联的方式才能保有其原来的功能。



串联的方式

现在我们就以串联的方式来写一个 int 08h 的服务程序：

程序 0.3 clock0.c

```
#include <dos.h>
long far *tickptr=0x0040006c;
int far *scrptr=0xb0000000;
static void interrupt clock();
static void interrupt (*oldclock)();
int seconds;
char digits[]={"0123456789"};
main()
{
    oldclock = getvect(0x08);
    setvect(0x08,clock);
}
void interrupt clock()
{
    oldclock(); ←调用 int 08h 原来的 ISR
    seconds = ((*tickptr)*10/182)%60;
    *scrptr++=0x7000+digits[seconds/10];
    *scrptr =0x7000+digits[seconds%10];
}
```

本程序执行后，屏幕左上角便会出现一个反白的数字秒表，如下：

这个程序和以前的程序有几处不同，首先是我们以 oldclock=getvect(0x08)保留了 int 08h 原来的服务程序地址，请特别注意 oldclock()的声明方式。oldclock()声明成 interrupt 类型是因为 oldclock()是一个中断服务程序，所以是以 iret 返回的，因此必须声明成 inter-

rupt 类型,TURBO C 才会在调用 oldclock()前加入一个 pushf 指令(请参考前一节的 testb.asm)。保留 int 08h 服务程序地址的原因是 int 08h 的 ISR 是重要的计时服务程序,我们不能像 int 05h 的 ISR 一样随便予以换掉不用。

我们把新的 int 08h ISR 叫做 clock(),并以 setvect(0x08,clock)把它设为 0x08 号中断的 ISR。此后每隔 1/18.2 秒 clock()程序便会被调用一次。clock()首先执行原来 oldclock(),所以 int 08h 原来 ISR 功能保留不变,这就是前面提到的串联法则。clock()额外多做的是把一个秒表的数值显示到 scrptr 所指的屏幕地址上。

秒表主要是取自 tickptr 所指的(0040:006c)地址的系统计数值,此计数值是由 oldclock()每隔 1/18.2 秒就加 1,所以此计数值,乘以 10/182(也就是除以 18.2)便成计时的秒数,若再除 60 取余数,则余数就成为一个秒表了。

我们以 seconds/10 及 seconds%10 分别取得秒表计数 seconds 的十位数及个位数,再以此为索引往 digits[]数组取出对应的 ASCII 字符,加上反白属性 0x7000 后,便存至屏幕左上角的地址。

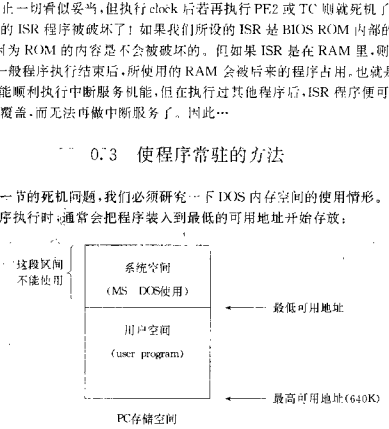
scrptr 是指向屏幕左上角的 Video RAM 地址 0xb0000000,因为屏幕每个字形占 2 bytes,所以我们声明 scrptr 为整数指针,因为整数也正好是占 2 bytes。这样的话 scrptr++便会往前进 2 bytes,正好指向下一字形位置。

死机了!

到目前为止一切看似妥当,但执行 clock 后若再执行 PE2 或 TC 则就死机了!为什么呢?因为我们的 ISR 程序被破坏了!如果我们所设的 ISR 是 BIOS ROM 内部的 Routine 则没什么问题,因为 ROM 的内容是不会被破坏的。但如果 ISR 是在 RAM 里,则情形就大不相同了,因为一般程序执行结束后,所使用的 RAM 会被后来的程序占用。也就是说,我们的 ISR 程序虽然能顺利执行中断服务机能,但在执行过其他程序后,ISR 程序便可能会被后来执行的程序所覆盖,而无法再做中断服务了。因此…

8.3 使程序常驻的方法

要解决前一节的死机问题,我们必须研究一下 DOS 内存空间的使用情形。当 MS-DOS 由磁盘装入程序执行时,通常会把程序装入到最低的可用地址开始存放:

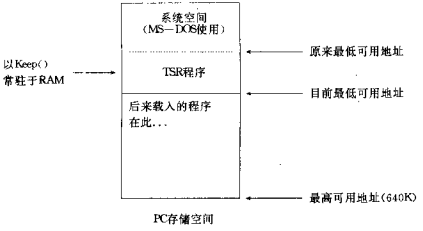


当执行下一个程序时,MS-DOS 还是把程序装入至同样的 RAM 区域执行。所以,后来

的程序会把先前的程序覆盖，这就是前一节发生死机的原因；我们的中断服务程序被覆盖掉了！

0.3.1 使程序常驻的方法

TURBO C 提供了一个叫做 keep()的函数可以使程序永存(常驻)于 RAM。以 keep()结束程序，DOS 会把该程序所用到的空间保留，使得以后执行的程序会装入到该常驻程序后面的地址，因而程序便得以常驻而不被摧毁(覆盖)。



PC存储空间

keep()——结束程序但常驻于 RAM
头文件：#include<dos.h>
声　明：void keep(unsigned char status,unsigned size);
执　行：keep (status,size);
返回值：void

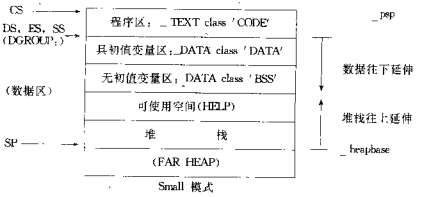
说明

keep()函数使用所谓的 Keep resident 的方式(即常驻方式)来结束程序。执行 Keep(status，size)时，size 是用来指定程序结束时所要保留的内存空间的大小(单位为 para＝16bytes)，并且要指定 status 值作为结束码(Exit Code)以返回给 DOS。以本函数结束程序时，DOS 会依 keep()指定的 size 来保留内存空间给常驻程序，以后其他程序均不能侵入内存区范围内。

0.3.2 计算所应保留的内存

使用本函数时，特别要仔细计算程序真正需要的空间，到底程序真正需要的空间是多少呢？MS-DOS 并未提供这方面的支持，我们只有从 C 程序本身的结构来探讨才能明白…。

TURBO C 程序的内存安排(一般称为内存映射)如下：



其中程序区是存放程序码。程序区以下为数据区，但数据区又细分为几个区域。其中"初值变量区"是存放具有初值的变量(例如：在函数外部以 int k＝100；所定义的变量 k 就会存放在"初值变量区"内。无初值变量区是存放无初值的变量(如 int j；这样的声明就会把 j 存入无初值的变量区)。堆栈区是由 sp 所指的堆栈地址开始往上使用，存放一般堆栈或 auto 变量。至于 heap 区也属于数据区的一部分，是一个可由 C 程序自由取用的内存空间，通常 C 程序的 malloc()等函数就是由 heap 区来取内存的。heap 和堆栈之间并无明显的界限，当堆栈往上堆或是 malloc()使用更多的空间时，heap 所剩的空间便逐渐减少。在堆栈后边的区域叫 far heap，far heap 只有在以 Compact、Large、Huge 等模式编译的程序中才会使用到。

使用 Tiny、Small、Medium 模式时，C 程序的活动就是介于上图程序区至堆栈区之间的范围内，也就是说我们只要 keep 这一范围内的内存，C 程序便能常驻了！TURBO C 特别提供了_heapbase、_psp 等变量，_Heapbase 是记载着 far heap 的开头地址，而_psp 则记载着程序 PSP 的节地址，所以_heapbase-_psp 就是整个 C 程序的活动空间了。我们便是以这种方法来算出 TSR 程序所要 keep 的内存范围。

_heapbase 是一个 far pointer，_psp 是一个节地址，而 keep()函数要求我们指定的是内存空间的 para 数，所以我们要用 FP_SEG 来取得_heaphase 的节地址，二者相减后再传给 keep()，详见下例。

这种方法对 Compact、Large、Huge 等模式也一样可用，只是这些模式在使用 malloc()等函数时会用到 far heap 空间，所以若以前述方式做成 TSR 程序时，要记得不可使用 malloc()等函数，因为这时 far heap 已不能再使用了。

注意！

执行 TSR 程序时要特别小心，如果在 TURBO C 的窗口模式下执行 keep()，则会连同 TURBO C 本身也常驻在 RAM，内存会多被占用 200K 以上，以至于结束 TURBO C 后竟无法再执行 TURBO C，而必须重新启动！这是常见的错误，必须按 Alt X 完全离开 TURBO C 后才能执行以 keep()结束的程序。

请参考下例

程序 0.4　clock1.c

```
#include <dos.h>
long far *tickptr=MK_FP(0x0040,0x006c);
void interrupt clock();
```

```
void interrupt ( * oldclock)();
extern void far * _heapbase;
int far * scrptr = 0xb0000000;
int seconds;
char digits[] = "0123456789";
main()
{
    oldclock = getvect(0x08);
    setvect(0x08,clock);
    keep(0,FP_SEG( _heapbase) - psp);          /* keep 本程序 */
}
void interrupt clock()
{
    oldclock();
    seconds = ((*tickptr) * 10/182)%60;
    *scrptr++ = 0x7000+digits[seconds/10];
    *scrptr = 0x7000+digits[seconds%10];
}
```

这段程序0.3的改良，主要是程序结束前以 keep()使程序常驻于 RAM，此后再执行其他程序都不会死机了。

clock1.c 是更改 int 08h 的向量，但我们也可以直接把 ISR 程序设给 int 1ch 中断，这样就不必在 ISR 中调用 oldclock()了。下面就是一个使用 int 1ch 的钟摆程序(每秒响一声)：

程序 0.5 clock2.c

```
#include <dos.h>
static int count=0;
static void interrupt tick();
extern void far * _heapbase;
main()
{
    setvect(0x1c,tick);
    keep(0,FP_SEG(_heapbase)-_psp);
}
void interrupt tick()
{
    char val;
    int i;
    count++;
    if ((count%18)!=0)
        return;
    count=0;
    val=inportb(0x61);
    outportb(0x61,val|3);                  /* 喇叭打开 */
    for (i=0;i<500;i++)                    /* 延迟一下 */
        ;
    outportb(0x61,val);                    /* 喇叭复原 */
}
```

clock 是 1/18.2 秒触发一次，而程序中的记秒器则是 18/18.2 秒就发声并归 0 一次，所以并不太准确。不过我们是以说明 clock 中断的原理为主，为了准确度而使程序复杂了反而不好。当然，你可以想办法把这个钟校调得更准确吧！

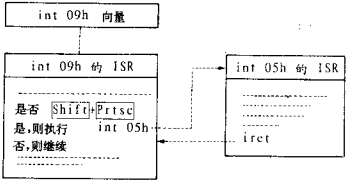
接下来一个有趣的问题是，我们的 CLOCK 是一个不停的在执行的程序，但是某些 TSR 程序是只有在需要时，按下特殊键立即可以执行，这种特殊的键叫做热键(Hot Key)，要了解热键的运行，这就要详细研究一下 PC 键盘的原理了…

0.4　以热键(Hot Key)触发 TSR 的方法

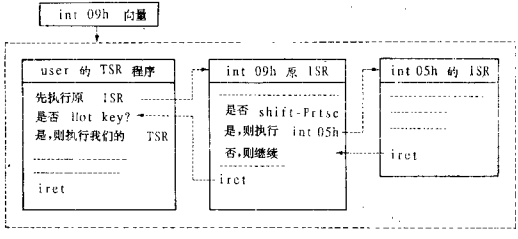
0.4.1　热键(Hot Key)

键盘是 user 与计算机沟通的主要管道，由键盘来触动 TSR 是最自然的想法。键盘的服务程序是 int 09h 及 int 16h，我们可以拦截键盘的服务程序(int 09h 或 16h 的 ISR)，检查按键的值，若 user 按下我们事先定好的某些特别键(叫 Hot Key，热键)，则便执行 TSR 程序。

其实 PC 的 PrtSc 就是一个 Hot Key，只不过这个 Hot Key 原本就在 int 09h 的 ISR 中设定好的。下面是 PrtSc 的运行过程：



而我们的 TSR 与 Shift+PrtSc 的道理是完全一样的。我们可以仿这个方式来做：



Hot Key 服务程序

要实现上述的目的，首先我们要写一个 Hot Key 的服务程序，并且把第 9 号中断向量指向 Hot Key 服务程序。这样下一次 int 09h 执行时(按键时)，便会被我们的 Hot Key 程序拦截。在 Hot Key 程序中，首先调用原来的 int 09 服务程序做读键的服务，然后检查按键码(你已知道如何检查吧?)，若是 Hot key，则就执行我们的 TSR 程序，若不是 Hot Key，则结束拦截动作。这种方法不只 int 09h 可用，int 16h 也一样可用。因为 int 09h 的优先性较高，所以我们不使用 int 16h。有人认为 int 16h 较易使用，这是因为使用 int 09h 要具备一点 PC 的知识(详见汇编语言实务一书)，但此处我们先调用 int 09h 的原服务程序，这个服务程序会代我们解决硬件上的问题，所以只要照我们的方法做就没问题。

0.4.2 INTERRUPT 的串接方式

在常驻程序的编写过程中，经常会有拦截、串接中断服务程序的情况发生，在串接的过程中，到底是先串入我们的服务的再接着原有的服务程序，或是先调用原有的服务程序再串接到我们的服务程序呢？为免相互抵触，我们拦截中断的方法都是遵循"先调用原服务程序再执行我们的服务程序"这种做法。

0.4.3 一个简单的 Hot Key 程序

下面是一个具有 Hot Key 功能的简单 TSR 程序：

程序 0.6 Hotkey.c

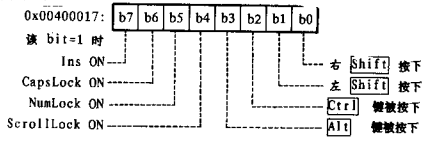
```
#include <dos.h>
extern void far *_heapbase;
void interrupt key();
void interrupt (*oldkey)();
int  active=0;
char far *shift_key=0x00400017;
int  far *scrptr=0xb0000000;
main()
{
    oldkey = getvect(0x09);
    setvect(0x09,key);
    keep(0,FP_SEG(_heapbase)-_psp);
}
void interrupt key()
{
    int i;
    oldkey(); ←先调用原来的 int 09h 服务程序
    if (active)
        return;
    active=1;
    if ((*shift_key & 0x0f) == 0x08)                    /* Alt_key */
    for (i=0;i<=2047;i++) {
        *scrptr&=0x00ff;
```

```
        * scrptr|=0x7000;
        scrptr++;
    }
    else if ((*shift_key & 0x0f) == 0x04)          /* Ctrl key? */
    for (i=0;i<=2047;i++) {
        *scrptr&=0x00ff;
        *scrptr|=0x0700;
        scrptr++;
    }
    active=0;
}
```

程序中我们建立了一个 interrupt 类型的函数 key()，每当按键时 key()便会被执行。key()一开始先执行原来的 int 09h 服务程序，所以有关键盘与主机间的沟通都由 oldkey()处理掉了，我们只要专注于热键的检查即可。另外我们设了一叫 active 的标志，每次 key()执行时要先检查 active，若为 1 则表示目前 key()正在执行中，就 return 以结束本次调用，这样可避免 key()中断了自己，而造成重复调用(reentrant)的现象(见下一章)。每次 key()要进行自己的处理前要先把 active 设 1。

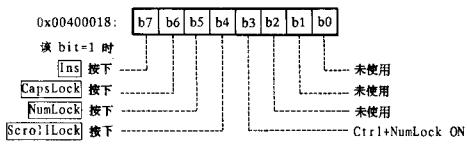
特殊按键地址

PC 的 0x00400017 与 0x00400018 地址(即 0040:0017 和 0040:0018 地址)上会记录着目前各特殊按键的状态：



0x00400017 位址记录着特殊按键的状态

0x00400017 地址记录着特殊按键的状态



0x00400018 地址记录着特殊按键的状态

我们可以读取这个地址的内容来判断某些特殊按键目前是否被按下。