向宇

王振祥 编著　李向宇 审校

微机实用技术技巧及实用程序

Computer

国防工业出版社

(京)新登字 106 号

内容提要

本书从作者的实践经验出发，揭示并解决了用户实际应用中经常遇到但又非常棘手的技术难题，揭示并深入讨论了 DOS 系统中鲜为人知的重要数据和参数。本书将编程人员编制软件时经常遇到的难题浓缩到 40 多个问题之中，其中大部分为实用技术问题，如采用 anti-aliasing 技术平滑 VGA 图形；引导区传染的计算机病毒的根治；可执行文件传染的计算机病毒的免疫等。详细讨论了问题出现的原因及其解决方法，并对大多数问题给出了实现的程序。同时，作者根据自己的实践经验，对 DOS 系统不完善的部分进行了弥补。书中给出的近 40 个实用程序可以直接作为系统的外部命令来使用，如文件搜索、目录树的删除等。编程人员在实际编程中也可以将书中的程序作为自己的一个过程直接调用，从而达到大大地提高编程速度的目的。

本书适合于软件编程人员，大中专院校的师生以及计算机软件爱好者阅读。

微机实用技术技巧及实用程序

王振祥　编著

李向宇　审校

责任编辑　陈子玉

国防工业出版社出版发行

（北京市海淀区紫竹院南路 23 号）

（邮政编码　100044）

新华书店经售

北京大地印刷厂印刷

787×1092 毫米　16 开本　印张 23 $\frac{1}{4}$　550 千字

1993 年 1 月第一版　1993 年 1 月第一次印刷　印数：0 001—8 000 册

ISBN 7—118—01134/TP・146　定价：19.60 元

前　言

在软件开发过程中，程序员难免会遇到许多技术问题，如：莫名其妙的编译错误，程序实际运行的结果与预想的结果不同，而在源程序中又不能发现错误等，对有些常见于资料的问题，查阅资料即可得到解决，而对另外的一些技术问题由于找不到有关资料而无法得以解决。笔者在多年的软件开发工作中，碰到过许多类似问题。在无资料可查的情况下，笔者不得不通过对系统及有关的软件进行分析研究来寻找解决方法。比如笔者曾经分析研究了 IBM PC XT BIOS、Super AT BIOS、BASIC 编译程序、DEBUG、SYMDEB、COMMAND、MORE、DISKCOPY、ANSI、EXE2BIN 等，通过这种分析研究，不但找到了解决问题的方法，而且发现并总结出了许多新的实用技术，1992 年年初以来，笔者将有关实用技术及相关程序整理成文，在至今不到一年的时间里，先后在《计算机世界报》、《计算世界月刊》、《中国计算机报》、《中国计算机用户》、《软件报》、《微型计算机》、《微型机与应用》、《计算机工程》等多种报刊杂志上发表了 20 余篇实用技术性文章，并有多篇文章被许多学术会议采用，如第七次全国计算机安全学术会议，中国计算机用户协会微机分会 1992 年年会，第四届全国青年计算机会议等，在发表这些文章之后，笔者收到来自全国各地同行们的几千封来信，探讨、咨询各种有关编程的技术问题。有鉴于此，笔者愿将这些多年劳作总结的实用技术成果汇集成册，以期能对读者有所帮助。作为第一手资料，这些文章及文章中的实用程序可作为工具备用。

除了上述实用技术之外，笔者在工作过程中为提高工作效率曾编写过许多实用工具，如文件快速全盘搜索，快速删除误拷贝文件，利用空格为文件、目录名加密，自动演示程序，分区保护器，不用驱动程序直接使用扩展存储器等。在此，笔者也将这些集技术与实用于一体的程序贡献出来与读者共享。

另外，为进一步方便读者，笔者已将本书中出现的所有实用程序（除演示举例用以外）编译连接成可执行文件，免费提供给广大读者。

在本书的编写过程中，得到了国防工业出版社的大力帮助与指导，对本书内容的完善及程序调试等方面，李向宇同志提出了许多宝贵意见，在此一并表示感谢。

能向读者奉献这本集技术与实用于一体的书籍，笔者感到很高兴，但也深深感到，由于作者水平所限，加之编写时间仓促，错误之处在所难免，恳请大方之家不吝指正，以求互相促进、共同发展。

王振祥
于北京向宇计算机技术公司

目　　录

第一部分　实用技术及其实现

1.　汇编语言程序定义缓冲区的新方法

采用汇编语言编程的人员，在程序中定义缓冲区是时常遇到的问题。普通的做法是采用 Buffer　db XXXX dup(0)这类指令，这种做法会使运行文件中出现连续 XXXX 个 0，这样不但运行文件较大，浪费磁盘空间，而且程序装载时间较长。有没有更好的定义缓冲区的方法呢？回答是肯定的。

第一种方法是：采用分配空间的方法(用 INT 21H 的 48H 号功能)。当需要缓冲区时，可用此功能分配相应的缓冲区，只需要几条指令即可，如：

```
        MOV   AH,48H
        MOV   BX,BUFSIZE
        INT   21H
        JB    ALLOC_ERROR
        MOV   ALLOC_SEG,AX
        ...
ALLOC_ERROR:
        ...
```

但这种方法有一大缺点，就是空间的分配是以节(1 节＝16 字节)为单位进行的，INT 21H 的 48H 子功能调用完成之后，在 AX 中返回的值是所分配空间的段址。这样在使用该缓冲区时，就需要经常进行段寄存器的转换。不仅麻烦，而且稍不注意就可能出现错误。这里笔者介绍另外一种定义缓冲区的方法，使用该方法定义缓冲区不需要任何额外的工作，而又具有和使用指令 db XXXX dup(0)定义缓冲区同样的效果。在介绍该方法之前我们先从 .COM 型文件谈起：从文件结构上讲，.COM 文件和 .EXE 文件相比有以下优点：(1)文件小，占用磁盘空间小；(2)装载速度快。此外，读完本文后读者会发现 .COM 型文件还有定义缓冲区却不占用磁盘空间的优点。但有一得必有一失，.COM 文件的优点是以其编程的限制为代价的。这些编程的限制有：(1)不能有堆栈段；(2)代码段必须从 100H 开始即程序中必须有指令：org 100h；(3)代码及数据之和不能超过 64KB。一般情况下，绝大多数用户编制的程序均可满足这些条件，鉴于 .COM 文件的优点，笔者建议在用汇编语言编程时尽可能编制 .COM 型文件。

对.EXE 文件和 .COM 文件，系统加载时的处理是不同的：

对.COM 文件，系统加载时，入口处的四个段寄存器 DS、ES、CS、SS 均指向程序段前缀 PSP，各段共用这一个段(这就是 .COM 文件的第一个限制)，SP 指针指向该段的最高端(SP＝0FFFEH)，而一个段最大只有 64KB(这就是 .COM 文件的第三个限制)。PSP 的大小为 100H 个字节，代码紧跟在 PSP 之后，即从 100H 开始(IP 为 100H，此即 .COM 文件第二个限制)。虽然 .COM 只使用了 64KB 的内存，但 DOS 为保持其与 CP/M 的兼容性，将全部的可用内存分配给 .COM 文件，这样，在程序有效代码之后的所有空间都是可用的，并可以将这些可用空间作为缓冲区使用。在使用中我们只需要记住缓冲区的位置即可。这种定义缓冲区的方法我们将其称为虚定义方法，即并不真正定义缓冲区。相应地，将其它真正定义缓冲区的方法称为实定义方法。

对 .EXE 文件，系统加载时，系统以各段的实际大小为其分配空间。这样要获得一个字节的空间就必须在程序中定义一个字节。因此，上述虚定义缓冲区的方法不适合于 .EXE型文件。图 1-1-1 比较了 .COM 文件和 .EXE 文件的内存布局。

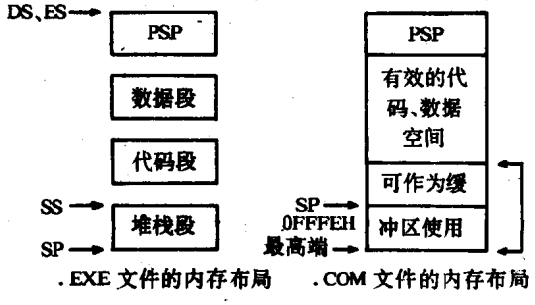

图 1-1-1 .EXE 文件和.COM 文件的内存分布

在编程过程中要使用虚定义的缓冲区(下称虚缓冲区)，必须注意以下问题：

(1)在程序入口处，SP 指向段的最高端(0FFFEH)，而要使用虚缓冲区时，必须将堆栈设置在一个安全的地方。比如将 SP 的值前移即可。

(2)虽然对 .COM 文件来说，内存中所有的可用空间都归它所有，但当使用超出其代码段的空间时，就需要进行段变换。实践中，对绝大多数的情况而言，该段中有效的代码、数据之后的空间即可满足我们定义缓冲区的需要，这里我们也主要是讨论这部分空间的使用。

(3)使用虚缓冲区的关键是记住缓冲区的位置，这可以通过编译器中的单元计数器($)实现。在编程中需要将某处作为虚缓冲区时，只要将此处的单元计数器的值赋予相应的变量即可。该变量需要定义多大，就将单元计数器的值增加多大，下一个变量可接着进行定义。

下面以具体的例子来说明虚缓冲区的使用方法，本例中，程序 PROGRAM1 将一个大小为 30000 字节的文件“SAMPLE . TXT”读入内存，程序如下：

```
;PROGRAM1. ASM
code        segment  public'code'
            assume   cs:code,ds:code
            org      100h
begin:      jmp      start
```

```
filename    db      'sample.txt',0
open_err    db      'Open file error$'
start:
            lea     ax,stk
            mov     sp,ax              ;将堆栈放在安全处
            lea     dx,filename
            mov     ax,3d00h
            int     21h                ;打开文件
            jnc     open_ok
            lea     dx,open_err
            mov     ah,9
            int     21h
            jmp     exit
open_ok:
            mov     bx,ax
            mov     cx,30000
            mov     ah,3fh
            lea     dx,buffer
            int     21h                ;读文件
            mov     ah,3eh
            int     21h                ;关文件
exit:
            mov     ah,4ch
            int     21h
pc          =       $
pc          =       pc+256             ;定义堆栈大小为256个字节
stk         =       pc
buffer      =       pc
pc          =       pc+30000           ;定义buffer的大小为30000个字节
code        ends
            end     begin
```

对 PROGRAM1 程序编译、连接，并通过 EXE2BIN 转换成. COM 型文件之后会发现，可执行文件 PROGRAM1. COM 只有 79 个字节。

为了明确比较起见，我们将其写成 .EXE 型文件(PROGRAM2 . EXE)：

```
;PROGRAM2.ASM
stack       segment public'stack'
            db      256 dup(0)
stack       ends
data        segment public'data'
filename    db      'sample.txt',0
open_err    db      'Open file error$'
buffer      db      30000 dup(0)
data        ends
```

```
code        segment  public'code'
            assume   cs:code,ds:data
begin       proc     far
            push     ds
            xor      ax,ax
            push     ax
            mov      ax,data
            mov      ds,ax
            lea      dx,filename
            mov      ax,3d00h
            int      21h                ;打开文件
            jnc      open_ok
            lea      dx,open_err
            mov      ah,9
            int      21h
            ret
open_ok:
            mov      bx,ax
            mov      cx,30000
            mov      ah,3fh
            lea      dx,buffer
            int      21h                ;读文件
            mov      ah,3eh
            int      21h                ;关文件
            ret
begin       endp
code        ends
            end      begin
```

将此程序编译、连接后，我们会发现 PROGRAM2 . EXE 的大小为 30847 个字节，其中 30000 个字节为缓冲区。

为进一步理解该方法，读者可在 . COM 型文件中用实定义的方法定义 buffer(即在. COM 型文件中也用 buffer db 30000 dup (0)来定义缓冲区，限于篇幅，在此不列出源程序，感兴趣者只要简单修改 PROGRAM1. ASM 即可)，将其编译、连接、转换后，比较一下结果，就会感到本文所介绍方法的优点所在。

2. 汇编语言如何调用高级语言

关于高级语言与汇编语言的混合编程，已有各种资料详细介绍了各种高级语言调用汇编语言的规则及方法，但对于那些喜欢用汇编语言编程的人，他们则可能希望在汇编语言中调用某些高级语言。但至今未见到介绍汇编语言调用高级语言的资料，这里笔者根据

自己的实践总结出了有关汇编语言调用高级语言的一些规则及方法，以期能受益于采用汇编语言编程的人员。

一. 参数的传递

高级语言中均是通过堆栈传递参数的，因此汇编语言调用高级语言时必须遵从这种规则，也使用堆栈传递参数，对不同的高级语言，传递参数时也有不同的约定，实际应用中，编程人员同样必须遵守这些约定。下面分别介绍 BASIC、C、FORTRAN、PASCAL 语言中有关参数传递的约定。

1. BASIC 语言

(1)BASIC 函数的调用均为远调用。

(2)BASIC 传递的均为参数的地址，因此必须两次操作才能得到真正的参数值。

(3)BASIC 参数压栈的顺序与各参数在源程序中出现的顺序相同。

BASIC 语言传递参数后的堆栈结构如图 1-2-1 所示。

2. C 语言(见图 1-2-2)

(1)C 语言函数的调用视所用模式的不同而不同，如果是小、中模式则为近调用(NEAR)，其它模式则为远调用(FAR)。

(2)与 BASIC 语言不同 C 语言传递的是参数的数值。

(3)C 语言传递参数的顺序与它们在源程序中出现的顺序相反。

3. FORTRAN 语言(见图 1-2-3)

(1)与 BASIC 语言相同，FORTRAN 语言函数的调用为远调用。

(2)FORTRAN 语言传递的为参数的地址，这与 BASIC 语言相同，与 BASIC 语言不同的是：如果 FORTRAN 语言采用的是大或巨模式，则传递的是远地址，其它则是传递近地址。FORTRAN4 . 0 以前的版本总是采用大模式。

(3)FORTRAN 参数压栈的顺序与它们在源程序中的顺序相同。

4. PASCAL 语言(见图 1-2-4)

(1)与 BASIC 语言相同，PASCAL 语言的函数调用为远调用。

(2)与 BASIC 语言不同，PASCAL 语言传递的是参数的数值。

(3)PASCAL 参数压栈的顺序与它们在源程序中的顺序相同。

需要特别指出的是：对于远地址，需要先压 SEGMENT，后压 OFFSET。这种作法便于汇编语言使用 LES 指令快速取得参数的地址。对于长整数，需要先压高位，后压低位。

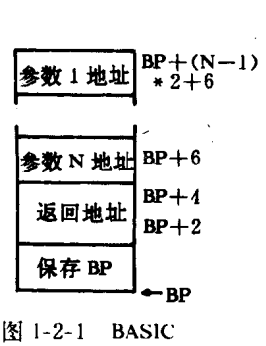

图 1-2-1 BASIC

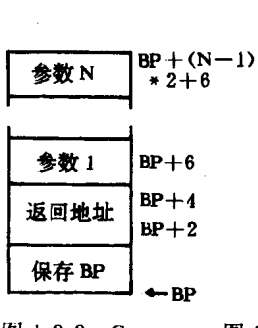

图 1-2-2 C

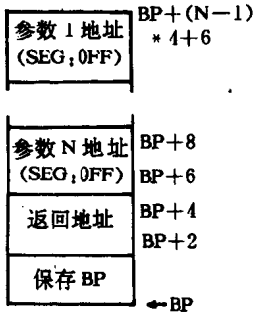

图 1-2-3 FORTRAN

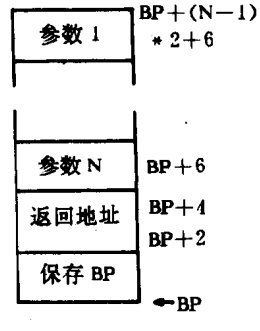

图 1-2-4 PASCAL

可以看出：除了 C 语言的小、中模式外，其他的调用均为远调用，即在声明外部函数

时，用下面语句：

EXTRN　　函数名：FAR（对C语言的小、中模式）

EXTRN　　函数名：NEAR(对其他情况)

二. 平衡堆栈

关于堆栈的平衡，C语言不同于其他语言，其它语言函数的堆栈平衡在函数内部完成，即函数在返回时采用如下指令：

RET ARG _ SIZE

其中ARG _ SIZE为压入堆栈的所有参数的总长度，因此调用函数后自动平衡堆栈，而C语言则不同，C语言函数只是采用简单的RET指令返回，堆栈的平衡留给调用者去完成，因此当调用者采用C语言函数时，必须在调用之后用指令ADD SP，ARG _ SIZE来平衡堆栈。

三. 关于命名的约定

BASIC语言以大写字符作为符号名字，这也是汇编语言的缺省约定，BASIC语言识别的名字长度最多可达40个，而汇编语言识别的名字的最大长度为31个(绝大多数情况满足此约定)。C语言识别的名字长度仅为8个，另外如果使用/NOI(GNORCASE)(考虑大、小写)选项，则C语言对大、小写敏感。与C语言公用的名字需要在名字前面加一下划线，如：_ TEXT。FORTRAN语言仅能识别名字的前6个字符，而PASCAL则识别名字的前8个字符(和C语言相同)。

四. 连接时需要LINK上所需要的库

高级语言的函数假定前面已经执行某些初试化工作，因此从高级语言启动程序，可确保初始化代码的执行，然后调用汇编模块，当需要时汇编模块又可以调用高级语言函数，如图1-2-5所示。

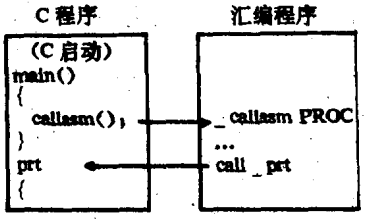

图1-2-5　汇编语言模块调用高级语言函数

下面以C语言的小模式为例，从汇编语言中调用C的PRT函数。下面是printf. c源程序：

```
#include    <io. h>
    #include    <stdio. h>
    extern      callasm();
    main()
    {
        callasm();
    }
    prt(format,long _ int)
    char     * format;
    long     long _ int;
```

```
{
    printf(format,long _int);
}
```

下面是 CALLASM. ASM 的源程序：

```
STACK           SEGMENT  STACK PARA'STACK'
                DW       100     DUP(0)
STACK           ENDS

_DATA           SEGMENT PUBLIC WORD'DATA'
OUT _FORMAT     DB  'THE LONG INTEGER IS %u!',10,0
UNSIGNINT       DW  50000
_DATA           ENDS

DGROUP          GROUP STACK,_DATA
                EXTRN    _prt:NEAR
                PUBLIC   _callasm
_TEXT           SEGMENT PUBLIC WORD'CODE'
ASSUME          CS:_TEXT,DS:DGROUP,ES:DGROUP,SS:DGROUP
_callasm        PROC
                PUSH     BP
                MOV      BP,SP
                PUSH     DS
                PUSH     ES
                PUSH     BX
                MOV      AX,DGROUP
                MOV      DS,AX
                MOV      ES,AX
                LEA      BX,UNSIGNINT
                MOV      AX,[BX]
                PUSH     AX
                LEA      AX,OUT _FORMAT
                PUSH     AX
                CALL     _prt
                ADD      SP,4
                POP      BX
                POP      ES
                POP      DS
                POP      BP
                RET
_callasm        ENDP
_TEXT           ENDS
                END
```

程序编辑完成后做：

(1)cl/c printf. c

(2)masm callasm

(3)link printf+callasm

形成可执行文件 printf . exe 即可运行。

3. C语言中定义数据类型需注意的问题

在用C语言编程的过程中，笔者曾写过一个程序，该程序用于将中西文混杂文件中的汉字输出，结果在程序执行后未输出一个汉字。但从程序表面却实在难以看出问题所在。无奈，笔者用 CODEVIEW 对编译结果进行跟踪，结果发现了一个定义数据类型时容易忽略的问题。本程序失败的原因就在于 BUFFER 的类型定义得不对，导致了程序的执行结果不符合要求。

问题的提出及对问题的分析

为了说明此问题，这里我们先将出现上述问题的源程序写出，程序如下：

```
/*          该程序输出中英文混杂文件中的汉字          */
#include <STDIO. H>
#include <IO. H>
#include <FCNTL. H>
#define  MAXSIZE 48000
char     BUFFER[MAXSIZE];
main()
{
    unsigned     NUMBER,i;
    int          HANDLE;
    HANDLE=open("SAMPLE. BIN",O _ BINARY|O _ RDONLY);
    if (HANDLE==-1)
    {  printf(" OPEN FILE ERROR! \n"); exit(1);  }
    NUMBER=read(HANDLE,BUFFER,MAXSIZE);
    for(i=0;i<NUMBER;i++)
       if(BUFFER[i]>=0XB0)
          {  printf("%C%C\n",BUFFER[i],BUFFER[i+1]); i++; }
}
```

上面的C语言源程序，在编译、连接的过程中均未发现错误，但运行时却未输出任何信息，也就是 printf("%C%C…)语句根本未得到执行，即条件 BUFFER[i]=0XB0 从未得到满足，而扫描的文件中又的确有汉字，为此笔者将 i 值直接设置到汉字的位置。执行时，条件仍然不成立。由此可以确定条件语句的编译结果不符合要求。于是笔者在 CODEVIEW 中用<F3>键转入汇编语言跟踪，结果发现了错误之所在。条件语句和 printf 语句的编译结果如下：

```
;       if(BUFFER[i]>=0XB0)
MOV     BX,Word Ptr [i]
MOV     AL,Byte Ptr [BX+_BUFFER]
CBW
MOV     SI,AX
CMP     SI,_abrktb+44(00B0)
JL      _main+7e(008E)
;         { printf("%C%C\n",BUFFER[i],BUFFER[i+1]); i++ }
MOV     AL,Byte Ptr [BX+0791]
CBW
PUSH    AX
PUSH    SI
MOV     AX,0060
PUSH    AX
CALL    _printf(062E)
...
```

通过上面的编译结果我们可以发现问题出在指令 CBW 和 JL _main+7e 上。编译程序在进行字符比较时，使用了 16 位数据而不是使用 8 位数据，为此它先将字符取入 AL 寄存器，然后用 CBW 指令将 8 位数据扩展成 16 位数据，CBW 指令的用法如下：

入口：AL

出口：AL 不变，AH=0，如果第 7 位为 0；AH=0FFH，如果 AL 的第 7 位为 1。

对汉字而言，字符的第 7 位为 1，因此扩展后成为 FFXXH。从编译结果看出，扩展后的数值与常数 00B0H 相比较，而这时，编译程序使用的比较指令是 JL，因为 JL 指令是将两个相比较的数值当做有符号数进行比较，因此显然 FFXXH(为负数)小于 00B0H(为正数)，因此对汉字来说条件永远得不到满足。那么对普通的 ASCII 码是否会出错呢？对普通的 ASCII 码，其值小于 80H，其第 7 位为 0，因此扩展后其值为 00XXH，与常数 00B0H 相比较，显然条件也得不到满足，因此对普通的 ASCII 码不发生错误，这样总的运行结果是不输出任何信息。那么为什么编译程序会使用 CBW 指令及有符号数比较指令 JL 呢？我们再看一下 BUFFER 的类型定义，这里我们将其定义为 CHAR，该类型为有符号字符，其表示的范围为－128 到 127，因此编译程序将其作为有符号数处理。而 CBW 和 JL 指令正是处理有符号数据的指令，所以编译程序使用了它们。同样 printf 语句也使用了指令 CBW。分析至此我们已经知道了产生上述问题的原因，这里只要我们将 BUFFER 定义为 UNSIGNED CHAR，编译、连接后运行结果正确，为比较起见，我们将新的编译结果也列在下面：

```
;       if(BUFFER[i]>=0XB0)
MOV     BX,Word Ptr [i]
CMP     Byte Ptr [BX+_BUFFER],B0
JB      _main+7c(008E)
;       { printf("%C%C\n",BUFFER[i],BUFFER[i+1]); i++ }
MOV     AL,Byte Ptr [BX+0791]
SUB     AH,AH
```

```
PUSH  AX
MOV   AL,Byte Ptr [BX+BUFFER]
PUSH  AX
MOV   AX,0060
PUSH  AX
CALL  _printf(062A)
...
```

与原来的编译结果比较，我们发现这时比较指令使用的是无符号比较指令 JB，另外 PRINTF 语句中也将 CBW 指令改为 SUB AH，AH，这样将字符作为无符号字符处理。

笔者还另外发现一个与此类似的问题，就是当使用 CHAR 型变量作数组的下标时，当数组下标为 80H 到 0FFH 时，这种扩展会使数组下标出界。比如下标为 81H 时，扩展结果为 FF81，用 FF81 作下标远远超出了数组的下限。为了详细说明这一问题，笔者将前面程序中的数组下标 i 定义为 CHAR，并将循环语句改为 for(i=0;i<255;i++)，编译结果为：

```
;     for(i=0;i<255;i++)
MOV   Byte Ptr [i],00
JMP   _main+51(0061)
INC   Byte Ptr [i]
MOV   AL,Byte Ptr[i]
CBW
MOV   SI,AX
...
;     if(BUFFER[i]>=0XB0)
CMP   Byte Ptr[SI+_BUFFER],B0
...
```

由此可见，C 语言的各种数据类型间有严格的区别，定义时需根据使用情况精确选择，尤其是类似的有符号数和无符号数的定义必须选择好，在编程时不能想当然地随便取一个，否则您就会同笔者一样费尽周折。

4. 测定 CPU 的类型

尽管当今的许多机器已经是 80286、80386 乃至 80486，但是大部分用户编程时仍采用 8088/8086 的指令集，其目的是为了保证程序能运行于不同的机器之上，然而这种作法虽然兼容性较好，但却不能充分地发挥 80286、80386 乃至 80486 这类机器的性能。而专门为这类机器编制的程序却又影响了兼容性。既保证兼容性又能发挥高级机器性能的方法是：先判断 CPU 之类型，然后据此采用相应的指令集，以尽可能地发挥机器之性能。这种作法的楷模当属 Windows 3.0。它能够在三种方式下运行：基于 8088/8086 的实地址方式、基于 80286 的标准方式和基于 80386 与 80486 的加强方式。无疑要实现此方法，首先必须确定 CPU 的类型。不同的 CPU 因为其结构不同，执行相同的指令却可能得到不同的

结果。以移位指令为例，如指令：SHR AX，CL，该指令是将 AX 寄存器右移，移动的次数在 CL 中。对 8088/8086 而言，CL 的值可以是 0－255 中的任何一个，而对其它类型的 CPU 则对 CL 之值加以限制，比如 80386 不允许 CL 值超过 31。这种限制显然是有道理的，因为移动次数超过要移动寄存器的位数时，完全可以用移动较少次数的指令实现同样的功能，而移动次数少时，指令执行时间短些。因此下面的几条指令能将 8088/8086 同其它类型的 CPU 区分开：

```
MOV     AX,2
MOV     CL,33
SHR     AX,CL
```

对 8088/8086，其执行结果是：AX 为 0，而对其它 CPU，AX 为 1。然后再用其它指令进一步将 CPU 区分开来。据此，笔者编写了确定 CPU 类型的程序 CPU. ASM。它可以区分出下列类型的 CPU：8088、8086、80186、80188、80286 和 80386。

```
;————下面为 CPU. ASM 源程序————
code    segment public'code'
        assume cs:code,ds:code,es:code,ss:code
        org 100h
begin:
        lea     dx,cpu
        call    disp
        call    get_large
        shl     ax,1
        mov     large,ax
        call    get_small
        add     large,ax
        mov     ax,large
        call    trans
        call    disp
        lea     dx,crlf
        call    disp
        mov     ah,4ch
        int     21h

get_large       proc
        xor     ax, ax
        push    ax
        popf
        pushf
        pop     ax
        and     ax, 0f000h
        cmp     ax, 0f000h
        jnz     g_1_5
        xor     al,al
        mov     al,40
        mul     al
        jz      g_1_4
        jmp     g_1_0
g_1_5:
        push    sp
        pop     bx
        cmp     bx,sp
        jnz     g_1_1
        mov     ax,0f000h
        push    ax
        popf
        pushf
        pop     ax
        and     ax,0f000h
        jz      g_1_2
        jmp     g_1_3
g_1_4:
        mov     ax,4
        ret
g_1_3:
        mov     ax,3
        ret
g_1_2:
        mov     ax,2
        ret
g_1_1:
        mov     ax,1
        ret
g_1_0:
```