PC 机常用汇编语言子程序库

李沐荪
费震宇 编译

北京科海培训中心
一九九〇年五月

PC 机常用汇编语言子程序库

李沐荪
费震宇 编译

北京科海培训中心
一九九〇年五月

目　录

第一章 引 言

汇编语言是IBM PC编程用的最快，最强有力的语言。然而，即使是富有经验的程序员，要编写一个真正有效的汇编语言子程序来完成特定的任务，也是不很容易而且费时的。

本书的目的是使所有程序员，不管他们懂不懂汇编语言，能够把快速，有效的汇编语言子程序结合到他们的程序中去。本书中的子程序不论在汇编语言或在高级语言程序中，如BASIC或Pascal程序中，都能同样很好地运行。

本书好像一本“食谱”，可以从中找到解决特定的编程问题的方法。汇编语言程序员可以用这些子程序作为范例，用来开发自己专用的子程序。

书中的子程序涉及许多方面的重要领域，如输入／输出，数制变换，多位算术，图形，声音，字符串及文件操作。在这些领域中，对计算机的直接控制特别重要。汇编语言提供这样的直接控制，因为它允许用户用计算机本身的机器语言，直接访问其中央处理器，从而加快执行速度，它还允许访问计算机硬件，这是高级语言做不到的。

本书第一章说明书的对象、使用条件、书的组成，以及如何使子程序和汇编语言程序以及高级语言程序接口。还讨论了某些基本的着眼点，例如寄存器的使用，以及书中为什么不用宏指令。随后的每一章含有某一不同领域需用的例程。这种组织方法便于按照子程序的功能来查找。

本书的对象

本书可供初学者和专家使用，为IBM PC或有关计算机编写汇编语言程序，或用于高级语言中，取得非用汇编语言得不到的速度和对机器的控制。

汇编语言程序员应当注意到IBM PC使用Intel 8088微处理器，它的指令系统和8086微处理器是一样的。因此，这些子程序也适用于8086 CPU。本书中以后各章的许多子程序具有通用性，只取决于是否有8086/8088 CPU，其中包括数字和字符串子程序，它们对任何8086或8088汇编语言程序员都是有价值的。

为有效地使用本书，读者不必是8088汇编语言的专家。高级语言程序员和初学者不必要理解子程序是如何工作的，更不说如何编写。熟能生巧，阅读本书中的子程序能学会一些技巧，有些则可以通过阅读8088汇编语言的有关书籍而获得（如Robert Lafore的Assembly language Primer for the IBM PC and XT, New York:Plume/Waite, New American Library,1984),有些则只有坐下来自己编写汇编语言程序才能获取。

如果读者是初学者，或高级程序员，可以结合自己的程序使用这些子程序，只需懂得如何将子程序和其他子程序“包”在一起，如何从自己的程序中调用，如何来回传送数据。我们将在本章中随后举例说明这样的过程。前述参考书中也有详尽的解释。

本书的特色

IBM PC有专门的图形和音响设备。针对这些重要领域，各设独立的一章。汇编语言

对图形和音响提供了其他高级语言不可能实现的控制、特别是图形控制需要迅速变换成千上万的象素来产生有用的图象。第七章，图形章中的子程序经过了仔细的优化，以取得最快速度。图形和音响子程序是专为IBM PC上的设备设计的，但可加以修改，以便用于其它图形和音响设备。

本书中许多程序与IBM PC的操作系统DOS接口。我们选用了DOS2.0版本，但许多子程序可以直接用于PC DOS1.0版本，以及C/PM-86。现在DOS已经有3.0以上的版本，但有时仍会遇到要用C/PM-86的情形。特别是第2章的标准I/O子程序，使用了三种系统都能提供的系统调用。

在关于文件操作的第十章中，子程序的设计采用了新的“文件把柄”（file handle），PC DOS版本2.0以上才有的这种新式调用大大地简化了磁盘文件的汇编语言编程，提供了通常必须由汇编语言程序员才能开发的功能。子程序演示了如何利用这些调用来编写程序，用来完成各种有用的任务，例如，从通信线中取文本文件存放在磁盘中。

PC DOS2.0版本以上还备有过滤器程序，这是一种Unix式的特色，主要特点是允许程序员在键盘和屏幕的友好环境下，开发和彻底测试文本处理子程序。然后，一旦找出了最恶劣的错误，可以不加修改地用同一子程序来操作磁盘文件。第十章收入了几个这样的过滤器程序。它们可以直接用来计数字符和净化文件，并且不难修改，以适应读者的特殊需要。

本书中的子程序均经过小心的测试，测试环境是用其他汇编语言程序进行调用，全面考验各方面的特性。

使用本书需要的条件

为最大可能地直接使用本书，应备有IBM PC或使用8088 CPU的兼容机，至少有64K用户内存，PC DOS或MS DOS版本2.0以上。还需要IBM发行的IBM宏汇编（M-ASM）或小汇编（ASM），供8086/8088用。如果使用宏汇编，至少需要96K内存。这两个汇编器都使用Intel的处理器指令助记符，变量和标识符的标识符名字可以长达31个字符，产生浮动的目标代码模块，与PC DOS提供的IBM连结器兼容。IBM推荐使用宏汇编，因为它支持其手册中所有的性能，但有一些附加的性能，其中包括宏指令，是我们不打算使用的。

使用全屏幕文本编辑器是有好处的，但也可以用DOS随带的EDLIN。如果用全屏幕编辑器与IBM宏汇编配合使用，单软盘，即使是双面360K磁盘，是不够用的。要用第二张盘放程序。

图形的第一章需要用IBM PC彩色图形适配器（CGA）。如果用的是其他图形卡，也不难修改子程序，来适应读者自己的系统。

音响的第一章需要IBM PC内装的扬声器和支持它的定时电路。其它“IBM PC兼容机”可能有，也可能没有这一设备。必须检查这样的机器，是否在这方面真正和IBM PC兼容。

本书的结构

从第二章起，每一章开头有一个简单的介绍，说明其范围，目的及收入其中的子程序

的特殊要求。介绍之后就是子程序。

每一个子程序都有仔细的文献，有专门的程序头，给出子程序的功能，其输入/输出，寄存器使用情况，段的使用情况，它调用的其它子程序，和任何特别的说明。这被认为是一种良好的编程习惯，许多使用计算机的公司都以某种形式提出这样的要求，不一定要把所有这些信息包括在读者自己的程序中，因为它占用源程序空间，键入费时费力。然而，放在适当的地方还是有好处的，所以读者可能要引用这本书作为自己程序的文献的一部分。

头段之后跟着每个程序的源码。源码的第一行几乎总是一条PROC指令。这是PROCedure（过程）一词的简写。每个8088子程序被认为属于一类较大的程序结构，称为过程。过程通常是代码块，被设计来完成专门的任务。过程可以被调用，就象本书的子程序的情形；也可以插入行间，如宏指令的情形。IBM PC汇编器要求用PROC指令，来确定从其他子程序调用本子程序时用什么样的机器代码合适（是近或远调用或返回）。任何IBM PC汇编过程源码的最后一行是ENDP（END Procedure过程终了）指令。它确切地告知汇编器，代码的哪些部分（特别是所有的RETurn返回指令）是在过程中。这样一种明确的起始和终结语句在汇编语言中强行形成现代的码块结构，从而提供现代化结构式编程技巧的优点。这些技巧被用来提高设计，编制和维护软件的效率。

在每个子程序中，我们几乎对每一行作了注释，说明其目的，虽然，有些注释如“保存寄存器”是指好几行代码，但仅出现在第一行上。使用空行（只带一个分号）来将子程序分成一些逻辑上互相关联的小块，从而使代码易读。代码愈易读，就愈容易改错，甚至一步就写成功。

子程序按它们互相调用的顺序排列。如用在Pascal中一样，一个子程序应位于所有调用它的子程序之前。这简化了汇编器的任务，并产生更有效的代码。这也是一种自然而又现代化的组成程序的方法。这意味着，作为用户，为理解和实现任一特定的子程序，只需考虑在它以前的子程序。

各章的顺序按类似的方法组成，从第2章的基本输入/输出开始，然后是第3章（二进制），第四章（BCD）和第五章（浮点）数制变换。前面这几章建立了向计算机输入和输出文本及数字信息的必要工具，这是进行几乎任何其他编程工作的前提。第六章，多位算术，含有四种基本数字运算的子程序，可到达任何要求的精度（仅受限于计算机的内存容量）。第七章和第八章分别讨论图形和音响，如前所述。第九章含字符串操作子程序。第十章含文件操作子程序（见上述关于文件的讨论）。

子程序与汇编语言的接口

为使本书中的子程序有用，必须与其它子程序组合，形成完整的程序，这可以全部用汇编语言来完成，也可以和高级语言组合。我们先看一下怎样完全用汇编语言工作；然后讨论如何将这些子程序与高级语言，如BASIC，写成的程序组合在一起。在进入例子之前，让我们预先复习一下内存分段的问题。

段的使用

用汇编语言对8088处理器编程时，必须了解有关段的知识。8088处理器访问主存时，

按64K分片，称为段。有四个寄存器，称为段寄存器，它们在8088 CPU的一兆字节寻址空间中，规定每一个段的起点。因此，在任一时刻，只有四个段在起作用。

四个段寄存器称为CS（代码段），SS（堆栈段），DS（数据段）和ES（附加段）。它们的名字表明了它们的功能。就是说，代码段用来存放处理器指令，堆栈段用来存放堆栈，数据段用来存放变量及其他数据。附加段也用来存放数据。

当使用IP（Instruction Pointer指令指针）来访问内存时，它取CPU指令，因此指向代码段中的一个地址。其它的寻址寄存器，如BX和SI，通常指向数据段内的地址，还有其它，具体地说，SP和BP指向堆栈段内的地址。在字符串操作中，附加段是DI的默认段（见下面的图1-1）。

段寄存器	段寄存器 CS	SS	DS	ES
IP	是	否	否	否
SP	否	是	否	否
BP	是 跨段	缺省	是 跨段	是 跨段
BX	是 跨段	是 跨段	缺省	是 跨段
SI	是 跨段	是 跨段	缺省	是 跨段
DI	是 跨段	是 跨段	缺省 非字符串存栈	缺省 字符串操作

图 1.1 段寄存器的使用

如果许多汇编语言文件被连结在一起，不同的段将以不同的方式结合和互相关联，有不同的特性。特别是不同汇编文件的代码段，常常是保持独立，放在内存中分开的区域中。在一个代码段中的子程序很容易通过远调用（FAR CALL）来调其它段中的子程序（但请注意，没有什么远转移，FAR JuMP）。为实现这类从一个代码段向另一个的切换，需付的代价并不很大，多数情形下由汇编器考虑解决。

堆栈段只需设在整个程序中的一个汇编语言模块内。它应被说明为“STACK”，使操作系统能够自动将它初始化为程序的堆栈。就在开始运行程序之前，操作系统自动设置堆栈段寄存器指向该段的起点。堆栈指针指向该段的终点。当程序运行时，堆栈向下生长，进入段体。

数据段可以用不同方式处理。事实上，对于大多数程序，只需一个数据段，由所有汇编模块共享。这是容易安排的——只需对每个汇编模块的数据段赋予同一名字，并说明每

个为PUBLIC。结果是一大块数据段，由不同汇编模块的分部组成。程序中只含一个数据段，在运行时，不太费事就可以换段访问数据。数据放在自备段中，而不放在代码段中，是一个基本哲理概念，具有极大的价值。它是现代结构式编程的又一例子，结构式编程鼓励“模块化”设计，事事各就其位。此外，由于每段只有64K，数据独自占用一段，使代码段的中代码有更充裕的空间。

用这种安排方法，在汇编时，某一块特定数据的偏移量是未知的，只有在所有模块连结在一起时，才能确定。结果是OFFSET算符不能在横跨几个汇编模块的整个数据段中返回正确的偏移量。然而有一条CPU指令LEA（Load EffectiveAddress装入有效地址），能够在程序实际运行时，将指定变量的偏移量装入指定寄存器。在需要计算复杂的数据结构等类变量的地址时，必须用这条CPU指令，而不用OFFSET算符。

附加段有许多功能。例如，它可以就是数据段，也可以是专用内存区，其中存放操作系统的数据，也或者是视频RAM。读者将在子程序中看到如何使用附加段。

与其它程序接口的两种方法

本书中的子程序可以在源代码这一级，利用文本编辑器与其他汇编语言程序组合在一起。它们也可以分组放入不同的汇编语言文件中，然后与其它子程序，包括读者自己所写在内，连接在一起。也可以将两种方法混合起来使用。

在同一文件内组合子程序

下面是汇编语言程序整个包含在一个文件内的例子。它将第2章中的几个标准I/O子程序组合在一个全程序中，允许读者以交互方式选印信息。程序本身没有什么用，只不过用作范例，说明如何编写简单但完整的汇编语言模块。

```
; EXAMPLE PROGRAM 1     CH1 EX1.ASM
;
; ------------------  equates begin ----------------------
cr          equ         13              ; carriage return
lf          equ         10              ; linefeed
; -------------------- equates end ---------------------+
;
; -------------------- stack area begins -----------------+
stacks segment stack                    ; stack segment starts here
      dw 5 dup (0)                      ; reserve 5 levels of stack with zeros
stacks ends                             ; stack segment ends here
; ---------------------- stack area ends ------------------+
;
;  ---------------------- data area begins --------------- +
datas  segment public              ; date segment starts here
;
; MESSAGES
menu     db cr,lf,'Message Demomstration Program',cr,lf
```

```
        db cr,lf,Press 1 or 2 for messages or CTRL/C to stop:
        db 0
mess1 db cr, lf, 'message number one', cr, lf, 0
mess2 db cr, lf, 'message number two', cr, lf, 0
mess3 db cr, lf', You hit an invalid key', cr lf, 0
datas   ends                    ; data segment ends here
; ----------------------------data area ends-------------- +
;
; ----------------------------code area starts ------------ +
codes   segment
;
    assume cs:codes, ss stacks, ds:datas
;
; --------------------routine begins ---------------------- +
;   ROUTINE FOR STANDARD INPUT WITH ECHO
;
stdin   proc    far
    mov   ah, 1                 ; standard nput
    int   21h                   ; DOS call
    ret                         ; return
stdin   endp
; ------------------------------- routine ends ------------ +
;
; ------------------------------- routine begins ----------- +
; ROUTINE FOR STANDARD OUTPUT
;
stdout  proc    far
    push  dx                    ; save registers
;
    mov   dl, al                ; in DL for DOS call
    mov   ah, 2                 ; standard output
    int   21h                   ; DOS call
;
    pop   dx                    ; restore registers
    ret                         ; return
stdout  endp
; -------------------------- routine ends -------------- +
;
; -------------------------- routine begins ------------ +
; ROUTINE TO SEND MESSAGE TO STANDARD OUTPUT
;
stdmessout  proc    far
    push  si                    ; save registers
```

```
    push    ax
;
stdmessout1:
    mov     al, [si]            ; get byte
    inc     si                  ; point to next byte
    cmp     al, 0               ; done?
    je      stdmessoutexit      ; if so  exit
    call    stdout              ; send it out
    jmp     stdmessout1         ;
;                               ;
stdmessoutexit:
    pop     ax                  ; restore registers
    pop     si
    ret                         ; return
stdmessout endp
; ---------------------------- routine ends ---------------- +
;
; ---------------------------- main program begins --------- +
; PROGRAM TO INTERACTIVLELY DISPLAY MESSAGES
;
main    proc    far
;
start:
    mov     ax, datas           ; get data segment
    mov     ds, ax              ; put into DS
;
main0:
    lea     si, menu            ; point to menu message
    call    stdmessout          ; send the message
;
    call    stdin               ; get key from user
main1:
    cmp     al, '1'             ; message number 1?
    jne     main2               ; skip if not
    lea     si mess1            ; point message 1
    call    stdmessout          ; Sent the message
    jmp     main4               ; exit to bottom of loop
main2:
    cmp     al, '2'             ; message number 2?
    jne     main3               ; skip if not
    lea     si, mess2           ; point to message 2
    call    stdmessout          ; send the message
    jmp     main4               ; exit to bottom of loop
```

```
main3:
    lea     si, mess3           ; point to message 3
    call    stdmessout          ; send the message
;
main4:
    jmp     main0               ; another message?
;
main    endp
; ---------------------------- main program ends---------- +
codes   ends
; ---------------------------- code area ends ----------- +
    end     start
```

以上的一切都放在一个名为CH1EX1，ASM的文件或模块中。可以看出，模块的各个主要组成部分用虚线分隔，虚线中部有标识，说明每节的起始和终了。

第一节含所有的equate伪指令，在此用常数给变量名赋值，用于程序其余部分。equate伪指令必须放在程序开始处，因为汇编器汇编程序其余部分时要引用这些数值。

equate放在前面的另一条理由是只需修改它们的数值，就可以使程序在新环境下运行。如果将equate伪指令放在最前面，并且予以适当的注释，则负责集成各个汇编程序为工作系统的程序就能更有效地工作。

第二节是堆栈，放在名为“STACK”的堆栈段中。注意该段被赋以属性“STACK”。如前所述，操作系统根据这一属性，在程序开始运行之前自动设定堆栈段寄存器和堆栈指针。如果生成堆栈段而不用“STACK”属性，必须用指令来初始化SS和SP寄存器。本例中，为堆栈保留了五个字。其它程序可能需要更多。

接着是数据，放在名为“DATAS”的数据段内(数据段含所有的变量与信息)。按照惯例，我们说明该段为“PUBLIC”，使它可被程序的所有其它部分（如果有的话）所访问，如果在其它模块中还有数据段，“PUBLIC”类型将使它们合并成一个。

最后的几节含有代码。所有代码都放在一个名为“CODES”的代码段中。每个子程序的代码放在自身的子节中，用虚线隔开。子程序的排列顺序是每个子程序放在任何调用它的子程序之前。特别是主程序，它放在最后。

虽然这些子程序在第二章中正式列出清单时都有头段，详细说明它们的功能，输入，输出，寄存器使用情形等，在本章的程序中出现时，已被去掉头段，以求节省空间。

因为主程序是一个无限循环，不需专门提供返回操作系统的途径。在程序运行期间的任何时刻，按Ctrl/C，就可返回操作系统。

程序演示如何使用第二章的STDIN和STDMESSOUT子程序。它用STDIN来检测需要什么信息，用STDMESSOUT来显示指定的信息。本程序所采用的控制结构——检查一系列可能性，看是否可能完成——可以用来编制其他交互式程序，完成有用的任务。

汇编本例

通常将编程工具，如编辑器，汇编器和连接器放在A驱动器中，待开发的汇编语言

程序放在B驱动器中。比较方便的是登录在B驱动器上，然后键入路径命令：

B>PATH A:\

这样就不必再考虑工具是在A驱动器中的问题，键入下面的命令进行汇编：

B>MASM CH1EX1;

于是在B驱动器中可以得到一个名为CH1EX1.OBJ的文件。为运行程序，还需连接，键入

B>LINK CH1EX1;

将生成CH1EX1.EXE文件。现在键入命令：

B>CH1EX1

程序将进入内存，开始工作。

使用分开的文件

如果用分开的文件来存放本书中的子程序，需要分别编辑和汇编多个文件。每个文件应含一组相关的子程序。例如，某一章的全部子程序可以放在同一文件中。于是每个文件形成一个独立的模块，可以完成一个特定的功能。就是说，所有算术子例程放在一个大文件中，所有图形子程序放在另一个中等等。用这样的模块来进行工作是有好处的，因为它节省了编辑和汇编模块需用的时间，已经工作的代码可以放在模块中，汇编成最终的简练的目标代码形式；还不能工作的代码可再进行编辑和汇编，直到正确为止。

放在一个模块中，可供其它模块中的子程序调用的子程序，必须在本身的汇编语言文件中说明为PUBLIC（公用的），在准备调用它们的文件中，必须被定义为EXTRN（即EXTeRNal，外部的）。下面我们用一个例子来说明这是怎样工作的。事实上，它就是和第一个例子一样的程序，不过以不同的方式组成。

先看含有主程序的文件。它被称为CH1EX2.ASM，含有与以前相同的大部分小节，包括equate伪指令，堆栈，数据和代码。然而，这次加了一节外部引用。所以必要，是因为有些子程序现在不在主程序模块中，它们放在自己的模块中，如果不说明它们是外部的，汇编器将认为它们已被遗忘，将给出一系列严重出错信息。

```
; EXAMPLE PROGRAM 2     CH1 EX2.ASM
;
; -------------------- equates begin -------------------- +
cr      equ     13                  ; carriage return
lf      equ     10                  ; linefeed
; ----------------------------; equates end ---------------- +
;
; + + + + + + + + + + + + + + externals begin + + + + + +
        extrn stdin:far, stdout:far, stdmessout:far
+ + + + + + + + + + externals end + + + + + + + + + + + + +
;
;-------------------- stack area begins --------------------
stacks segment stack                ; stack segment starts here
    dw 5 dup (0)                    ; reserve 5 levels of stack with zeros
stacks ends                         ; stack segment ends here
```

```
; ------------------- stack area ends ------------------- +
; ------------------data area begins-------------
datas segment public                ; data segment starts here
;
; MESSAGES
menu, db cr, lf,'Message Demomstration Program', cr, lf
      db cr, lf,'Press 1 or 2 for messages or CTRL/C to stop: '
      db 0
mess1 db cr, lf, 'message number one', cr, lf, 0
mess2 db cr, lf, 'message number two', cr, lf, 0
mess3 db cr, lf, 'you hit an invalid key', cr, lf, 0
;
datas ends                          ; data segment ends here
; -------------------------data area ends ---------------- +
;
; ------------------------code area starts ------------- +
codes segment
;
    assume cs:codes, ss:stacks, ds:datas
;
; ---------------------- main program begins ------------ +
; PROGRAM TO INTERACTIVELY DISPLAY MESSAGES
;
main    proc    far
;
start:
    mov     ax, datas           ; get data segment
    mov     ds, ax              ; put into DS
;
main0:
    lea     si, menu            ; point to menu message
    call    stdmessout          ; send the message
;
    call    stdin               ; get key from user
main1:
    cmp     al, '1'             ; message number 1?
    jne     main2               ; skip if not
    lea     si, mess1           ; point message 1
    call    stdmessout          ; send the message
    jmp     main4               ; exit to bottom of loop
main2:
    cmp     al, '2'             ; message number 2?
    jne     main3               ; skip if not
```

```
    lea     si, mess2           ; point to message 2
    call    stdmessout          ; send the message
    jmp     main4               ; exit to bottom of loop
main3:
    lea     si, mess3           ; point to message 3
    call stdmessout             ; send the message
;
main4:
    jmp     main0               ; another message?
;
main        endp
; ------------------ main program ends ---------------------- +
codes       ends
; ------------------ code area ends ------------------------- +
    end  start
```

外部引用的每一项给出了不在本模块中的子程序或变量名，并赋予一个“类型”。类型是一个重要成份，它确定用什么机器代码来引用子程序或变量，因此必须说明。因为这些子程序位于自己独立的代码段中，它们属于far（远）类型，在机器语言中需要远调用。

本例中，文件CH11O.ASM含子程序。本模块（见下面）还给出了将子程序形成软件包的一般方法。具体地说，所有子程序都放在名为"CODES"的段中。语句

codes segment

后面没有加上public，因此它们不会和其他模块中的同名段结合在一起。请注意，这里谈到的public是所谓“结合类型”（combine type）；它的作用是决定同名代码段是否结合成一个连续的段。

紧接以上语句的另一句

public stdin, stdout, stdmessout

用到的public是所谓伪指令，它说明后面跟随的子程序可被其它模块引用。一个是结合类型，一个是伪指令，不幸名字相同，请不要混淆。

```
; EXAMPLE PROGRAM 3 - I/O MODULE-FILE CH11O.ASM
;
; ----------------------------code area starts ------------------------------------
codes    segment
;
; + + + + + + + + + + + + + public declaration start + + + + + + + + + + + + +
    public stdin, stdout, stdmessout
; + + + + + + + + + + + + + + + + + public declaration end + + + + + + + + + + +
;
    assume cs:codes
```

```
; ------------------------------routine begins------------------------------ +
; ROUTINE FOR STAND INPUT WITH ECHO
;
stdin    proc    far
    mov         ah, 1               ; standard input
    int         21h                 ; DOS call
    ret                             ; return
stdin    endp
; ------------------------------routine ends-------------------------------- +
;
; ------------------------------routine begins------------------------------ +
; ROUTINE FOR STANDARD OUTPUT
;
stdout  proc    far
    push    dx                      ; save registers
;
    mov     dl, al                  ; in DL for DOS call
    mov     ah, 2                   ; standard output
    int     21h                     ; DOS call
;
    pop         dx                  ; restore registers
    ret                             ; return
stdout endp
; ------------------------------ routine ends-------------------------------- +
;
; ------------------------------ routine begins------------------------------ +
; ROUTINE TO SEND MESSAGE TO STANDARD OUTPUT
;
stdmessout proc    far
    push    si                      ; save registers
    push    ax
;
stdmessout1:
    mov     al, [si]                ; get byte
    inc     si                      ; point to next byte
    cmp     al, 0                   ; done?
    je      stdmessoutexit          ; if so exit
    call    stdout                  ; send it out
    jmp     stdmessout1
;
stdmessoutexit:
    pop     ax                      ; restore registers
    pop     si
```

```
    ret                                 ; return
stdmessout endp
; ------------------------------------routine ends-------------------------------+
codes           ends
; ------------------------------code area ends-----------------------------------+
    end
```

在代码模块中作了public说明。不需要说明子程序类型。因为从上下文中可以知道。就是说，子程序和变量就在"PUBLIC"命令所在的模块中，因此汇编器已经知道它们的类型。

为汇编此程序，先键入

B>MASM CH1EX2;

生成目标码CH1EX2.OBJ；再键入

B>MASM CH1IO;

生成I/O模块的目标码CH1IO.OBJ。然后，用下列命令将两个目标码连结在一起：

B>LINK CH1EX2 CH1IO;

为运行程序，键入

B>CH1EX2

应当得到和以前相同的结果。

子程序与BASIC接口

下面的例子演示如何将本书中的子程序通过CALL语句与BASIC接口。

本例置于文件CH1EX3.ASM中。它用本书第三章的BIN16OUT子程序，将内部16位整数变换成ASCII二进制形式在屏幕上显示出来。

```
; EXAMPLE PROGRAM 4 - CH1EX3.ASM
;
; ----------------------------externals begin-----------------------------------+
        extrn   stlout:far
; ------------------------- externals end ------------------------------------+
;
; -------------------------- code area starts ---------------------------------+
codes   segment
        assume  cs:codes
; ------------------------ routine begins ------------------------------------+
;
bin16out    proc    far
;
; a binary number is in DX
;
        push    cx              ; save registers
;
        mov     cx, 16          ; loop for a count of 16
```

```
bin16out1:
        rcl     dx, 1           ; rotate DX left once
        mov     al, dl          ; move into AL
        and     al, 1           ; just keep digit
        add     al, 30h         ; add 30h to AL
        call    stdout          ; send it out
        loop    bin16out1
;
        pop     cx              ; restore registers
        ret
;
bin16out        endp
; ------------------------------ routine ends ------------------------------+
;
; ------------------------------ main program begins ------------------------+
; PROGRAM TO INTERFACE BETWEEN BASIC AND ASSEMBLY LANGUAGE
;
main    proc    far
;
start:
        push    bp              ; save BP register
        mov     bp, sp          ; point BP to stack
;
        mov     bx, [bp+6]      ; get address of data
        mov     dx, [bx]        ; get the data
        call    bin16out        ; convert to binary
;
        pop     bp              ; restore BP
        ret     2               ; return skipping the data
main    endp
; ------------------------------ main program ends--------------------------+
codes   ends
; ------------------------------ code area ends ----------------------------+
        end     start
```

此程序的核心是一个接口子程序，由BASIC中的CALL语句调用。可以使用USR函数，但是CALL语句更灵活和有力。如果用USR函数，接口子程序将不大相同。可以参看本章第一节中提到的参考书。

当使用BASIC的CALL时，数据通过8088的系统堆栈向子程序往来传送。因为堆栈也用来存储返回地址，必须特别小心取出所要的数据而不毁坏或丢失返回地址。使用BP（基底指针）寄存器来完成这一过程是很方便的。堆栈指针（SP）和基底指针通常都指向堆栈段。然而堆栈指针不能用于通常的寻址方式。习惯的做法是将BP的内容压到堆栈