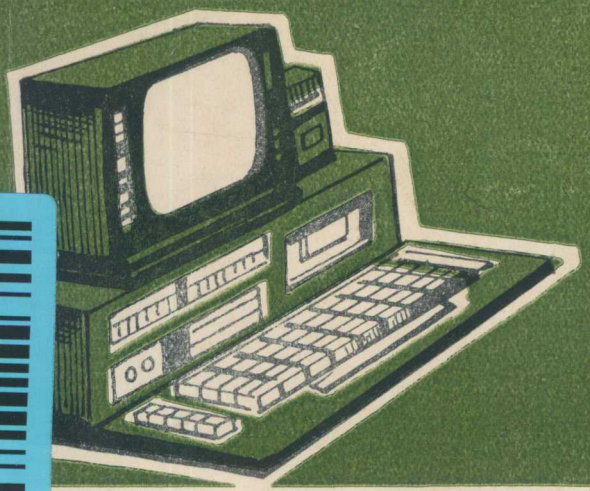
微型计算机

Microcomputer



46

IBM-PC & XT 常用汇编语言程序集

徐飞跃　高　佩
张贻彤　陈奇明　译

徐子亮　校

1986年7月

译　序

近年来，微型计算机技术发展迅速，被誉为第二代个人计算机的IBM PC、PC/XT是美国国际商业机器公司(IBM)于八十年代初开发成功的微型计算机产品，具有极其丰富的系统软件和应用软件。它在事务处理，办公室自动化、教育、通讯、控制、工程设计等许多领域都得到了广泛的应用，也是国际市场畅销的微型计算机系统之一。

目前，IBM PC，PC/XT，PC/AT及其兼容机在我国的推广应用正在逐步深入，为了配合国内微型计算机生产、科研、教学和应用开发工作的开展，我们翻译了由美国莫尔根编写的BLUEBOOK OF ASSEMBLY ROUTINES FOR THE IBM PC & XT"一书。全书共分十章，详细地介绍了IBM PC的各种实用汇编语言子程序，其中包括了输入/输出，数字转换，多位数运算，图形，声音，字符串处理和文件处理子程序等。全书有9种类型共计103条子程序。

在每章的开头列有该章所包含子程序的目录，它包括子程序名和该子程序的功能说明，以便读者迅速选择需要的模块。在各子程序的模块中，都按照汇编语言程序的统一格式对下列项分别作了说明，其中有：功能，输入参数，输出参数，修改的寄存器名称，被访问的段地址，被调用的程序名称等。然后列出了该子程序的清单并有相应的注释。因此读者可方便地引用书中各个子程序模块并变成自己实际程序的一部份。

本书适合广大从事微型计算机科研、生产、教学和应用开发的软件设计人员参考，也可作为高等院校计算机系及有关专业的软件教学参考书。由于翻译时间仓促，错误或不当在所难免，敬请读者批评指正。

译　者

目　录

第一章 概 述

对于使用IBM-PC进行编程的程序员来说，汇编语言是一种速度最快，功能最强的语言。但即使是一个有经验的程序员，写一个确实高效的汇编语言子程序来完成一个特定的工作也是比较困难且费时的。而对于那些程序设计知识和能力局限于诸如BASIC和PASCAL之类高级语言的人来说，写一个这样的汇编语言子程序似乎是不可能的。

本书的目的是：对所有程序设计人员(不管其是否善于使用汇编语言进行编程)来说，都能够将本书中快速，高效的汇编语言子程序结合到其自己的程序（不管是汇编程序还是高级语言)中。需要说明一点，希望使用本书中汇编子程序的人，毋需担心所用的子程序在各种语言程序中的效果，因为无论用在汇编语言程序中还是在诸如BASIC或PASCAL这样的高级语言程序中，它们都有相同的效果。

本书并不需要从头读到尾，除非你对汇编语言有相当兴趣，并希望了解每个子程序的编程技能。本书就象一本菜谱，你可以从中找出"处方"用于解决特定的程序设计问题。汇编程序设计人员也可以将这些主程序作为一个基本模块或出发点，为开发他们自己的实用程序服务。

本书中的主程序包括有输入/输出、数值转换、多位算术运算、图象、声音、字符串和文件处理等不同且重要的领域，在这些领域中，对计算机的直接控制是特别重要的。而汇编语言提供了这些直接控制，它允许你以机器语言直接访问计算机的中央处理器使你的程序达到最大的速度，并能访问计算机的硬件，而这一点在高级语言中是不可能实现的。

本章将对如下几个方面作说明：本书的对象，使用本书需要什么，本书是如何构成的，如何将本书中提到的子程序用到你自己的汇编语言程序和以高级语言编写的程序中去。此外还讨论了一些基本的，原理性的问题，诸如寄存器用途等，以及为什么本书中没有使用宏命令的原因。以后每章按不同的题目范围对子程序进行分类，这种构成方法提供了一种按照子程序实现的功能归类且易于检索的方法。

本书对象

在编写本书过程中，作者在编写方法上立足于初学者。但是，对于那些正在为IBM-PC或其它有关计算机编写汇编语言程序的，或者正在以无法提供足够速度和对这些机器控制的高级语言—只能用汇编语言才有效—进行编程的专家们来说，本书也是他们必备工具。

Intel 8086和8088处理器的汇编语言程序员应该注意：IBM-PC使用8088微处理器，其指令集与8086相同，所以8088微处理器上的子程序，同样可以在8086 CPU上运行。(本书不对这两种芯片进行完整的描述)。本书中许多子程序具有通用特性，仅取决于8086/8088CPU的存在与否。它们是数值和字符串处理子程序，这些子程序对8086和8088汇编语言程序员来说都是有效的，且具有相同的运行结果。

即使你不是一个8088汇编语言程序设计的专家，你也可以方便、迅速、高效地使用

本书。初学者和高级语言程序员既不需要知道子程序是如何工作的，也不用去管如何从头写这些程序。这些子程序中所包含的技能往往产生于经验，一些技能可以通过学习本书的子程序而得到，一些也可以从有关8088汇编语言书籍中得到，而其他技能则必须通过坐下来写你自己的汇编语言子程序才能得到。

如果你是一个初学者或高级语言程序员，你可以通过将本书的子程序合并到你自己的程序中。为此你只需知道这些子程序如何组装到你的程序中去，如何从你的程序中调用它们，并且是如何传递数据的。这些我们将在本章中详细介绍。

本书的特点

IBM-PC有自己特有的图象和声音部件，而图象和声音这二种能力对计算机软件来说是相当重要的，为此本书在这两个方面各有一章。高级语言对声音提供的控制是无法与汇编语言相比的，在图象方面更是如此。常常，由千万个象元组成的图形必须迅速改变其位置或形状以产生有用或有趣的画面。第七章—图象中的子程序已经仔细地优化以达到最大的速度。需要说明的一点是，图象和声音子程序是专门为IBM-PC特别设计的，但修改后这些子程序同样能在其它图象和声音设备上运行。

本书中许多子程序都涉及到IBM公司为支持IBM-PC而提供的PC磁盘操作系统(DOS)。我们选用了DOS版本2，这是因为它具有较强的功能。但是许多子程序也可以在DOS版本1，以及CP/M86上运行。特别是第2章中的标准I/O子程序使用对这三个系统都有效的系统调用。

第10章文件处理中的子程序仅在DOS版本2中有效，因为它们都使用了在DOS版本2中才有的"文件处理"系统子程序。这些新的系统子程序，极大地使得磁盘文件处理的汇编语言程序设计变得简单化。在这些新的系统子程序中，有一些是通常必须由汇编语言程序员开发的文件处理通用子程序。本书把我们的子程序介绍给大家，不但使某些人能使用它们，而且还能使另外一些人了解如何利用这些系统子程序来写一些执行诸如通讯线中将文件存入磁盘之类非常有用的工作程序。

PC DOS版本2也有一个类似于UNIX特性的筛选程序。它允许汇编语言程序员在键盘和屏幕等较好的环境下开发和完全测试文本处理子程序，然后一旦错误消失，用这些未修改过的相同的子程序来处理磁盘文件。第10章中有数个筛选子程序。可以直接将它们用于对文件进行计数和清除，而且也可以很容易地对它们进行修改以满足你自己的特殊要求。

使用本书你需要什么

使用本书的先决条件是你应该有一台IBM个人计算机或一个具有8088 CPU的IBM个人计算机的兼容机，至少有64K用户区，PC DOS2.0或MS DOS。此外，你还将需要IBM公司发行的8086/8088 IBM宏汇编程序(MASM)或小汇编程序(ASM)。如果使用宏汇编，至少要求计算机有96k内存。这两个汇编程序都使用Intel处理器指令的助记符，允许长名(31个字符)作为变量或标号标识符。汇编程序产生与PC DOS支持的IBM LINK兼容的可再定位目标模块。IBM公司建议大家使用宏汇编程序，因为，它提供手册上所有功能，包括一些本书中没有使用的特性，例如宏指令等。

对于有关图象的章节，你还将需要IBM-PC彩色图形适配器。如果你使用的是其它型号的图象板，则不能直接使用本书中的子程序，必须对它们略作修改才能在你的系统上工作。

关于声音章节，我们要求IBM-PC必须有内装喇叭且提供时钟电路。其他IBM-PC兼容机有的可能有，有的可能没有这套装置。如果你是使用兼容机，必须检查一下你的计算机在这个问题上是否与IBM-PC真正兼容。

本书是如何构成的

从第2章开始，每章一开始都有一个简短的章节说明，该说明解释该章中子程序的范围、目的和一些特殊要求。子程序则紧跟在章节介绍之后。

每个子程序都有一个特殊引头，它仔细地注释了子程序的功能，它的输入输出，所用寄存器，引用段，调用程序和注释等。在国际上一些较好的程序例子都采用了这种形式。许多使用计算机的公司都要求这种形式的程序。在你使用这些子程序时，你可以不将这些说明信息包含在你的程序中，因为它们将占用你源程序空间，并需要时间和人力输入它们。但有时候这些信息是非常有用的。当你使用本书中子程序时，你可以将其说明作为你程序说明的一部分。

每个子程序的源代码都紧跟在引头部分的后面，而源代码的第一行几乎都是一条proc指令。proc是英语单词"过程"(procedure)的缩写形式。每个8088子程序都认为是属于被称为过程的程序大类中的一部分。一般来说，过程是一个代码块，用于实现特定的工作。而从子程序角度来讲，过程可以被调用，从主程序角度来讲，过程可以插在某个程序之中。为了使其它子程序能正确有效地调用一个子程序，IBM-PC汇编程序需proc指令来确定适当的机器代码(不管它们是near型还是Call调用或Ret返回)。IBM-PC汇编语言过程源代码的最后一行是ENDP指令(endprocedure的简写形式)。通过PROC和ENDP能确切地使汇编程序知道程序中哪一部分是属于过程的。有了这些清晰的开始和结束语句，使得汇编语言程序具有块结构，为提供结构程序设计实践提供了方便。这种实践的目的在于增加设计人员对软件的设计、编程和维修的效率。

在每个子程序实体中，几乎每行都加有注释，以说明其用途。某些诸如"寄存器保护"之类的简单注释可能会涉及到连续数行源代码，我们仅在这类源代码的首行注释一次，所以注释的方法是极其自由的。又如，利用空行(仅有分号"；")以结构上将子程序分成小块逻辑相关代码行，因而使源代码更易读。易读的源代码较容易纠错和一次生成功。

有些较大的子程还给出了程序流程图，给出流程图的目的在于让那些对子程序逻辑流程有兴趣的程序员能够详细了解子程序结构细节。当然有一定程序设计经验的程序员根据子程序本身就能了解子程序框图要点，但对那些没有汇编语言程序设计经验的程序员来说，流程图无疑将有助由他们对子程序的了解。

子程序以什么次序形成一较大的程序呢？它们是以互相调用的先后次序组织的。正如PASCAL，每个子程序出现在所有其它调用它的子程序之前。以这样的次序构成的程序可以使汇编器需做的工作变得简单化，且能产生效率较高的目标代码。这也是一种自然的和先进的程序构成方法。

本书也以类似的方法组织章节的次序。从第二章的基本输入输出，第三章(二进制)，

第四章(BCD 码)，第五章(浮点数)的数值转换开始。前面这几章是为以后从计算机输出或输入到计算机里的文字或数值信息建立必要的工具，是一个几乎是做其它各种程序设计的先决条件。在第六章多位算术运算中，包含有执行各种精度要求的四种基本数值运算的子程序(精度极限是由计算机内存大小来确定)。第七和第八章分别为图象和声音，这在前面已经讨论过。第九章包含有处理字符串的子程序，而第十章为文件处理子程序（见上面关于文件的讨论)。

子程序与汇编语言接口

为了使用本书提供的通用子程序，使它们发挥作用，我们必须把所需要使用的子程序与其它子程序相结合形成一个完整的程序。这种结合可以在纯汇编语言的程序中实现，也可以在高级语言程序中实现。我们将首先介绍如何在纯汇编语言中实现这种结合，然后再讨论如何将子程序结合进由 BASIC 之类高级语言编写的程序中。在开始讨论这些之前，我们先初步地简单介绍一下存贮器段的概念。

段的用途

当用汇编语言对 8088 微处理器进行编程时，必须对段(segment)有一个基本的了解。

8088 处理器可以直接访问 64k 主存，这 64k 区域称为一段。有四个寄存器称为段寄存器。它们指出在一个 8088 CPU1 兆字节寻址空间中每个段的开始地址。在某一时刻，仅有四个段处于活跃状态。

上面提到的四个段寄存器分别是：CS(代码段)，SS(栈段)，DS(数据段)和 ES（附加段)。每个段寄存器的名字实际上已经指出了他们的功能。也就是说，代码段用于存放处理器指令，堆栈段用于存放堆栈，数据段用于存放变量值和其它数据，附加段也是用于存放数据。

当使用 IP(指令指针)访问内存时，它表示取 CPU 指令，则指向代码段中的一个地址。其他诸如 BX和 SI 之类的寻址寄存器通常指向数据段中的一个地址，而其他SP 和 BP 则指向堆栈段中的一个地址。附加段对 DI 在字符串操作中来说是一个缺省段（见图 1-1)。

如果多个汇编语言文件用 LINK 连接在一起，则各种段将结合起来，且以不同的方法相互访问，各段各不相同。特别是不同汇编语言文件中的代码段常常是分离的，存放在不同的内存区域中。某个代码段中的子段序可以容易地通过 FAR CALL 调用其它段中的子程序(请不要用 FAR JMP 转)。将开关以一个代码段拨向另一个代码段所需的开销是相当少的，而且这些多半由汇编程序来实现。

栈段的需求取决于整个程序中的汇编语言模块。它应该声明“STACK”，使得操作系统能自动地将它作为你程序需要的栈进行初始化。在程序运行之前，操作系统自动地设置栈段寄存器来指出该段的开始位置，而栈指针指到该段的结束位置。

汇编语言程序设计员可以用不同的方式处理数据段。对大多数程序员来说都喜欢使用一个数据段，由所有汇编语言模块来共享。这样做的目的是使数据段便于管理。实现这种管理也是很容易的，只需在每个汇编模块中给数据级一个相同的名字，并声明为 PUBLIC。由此而产生的结果是在一个大的数据段中包含有各汇编模块的数据部分。程序中只有一个

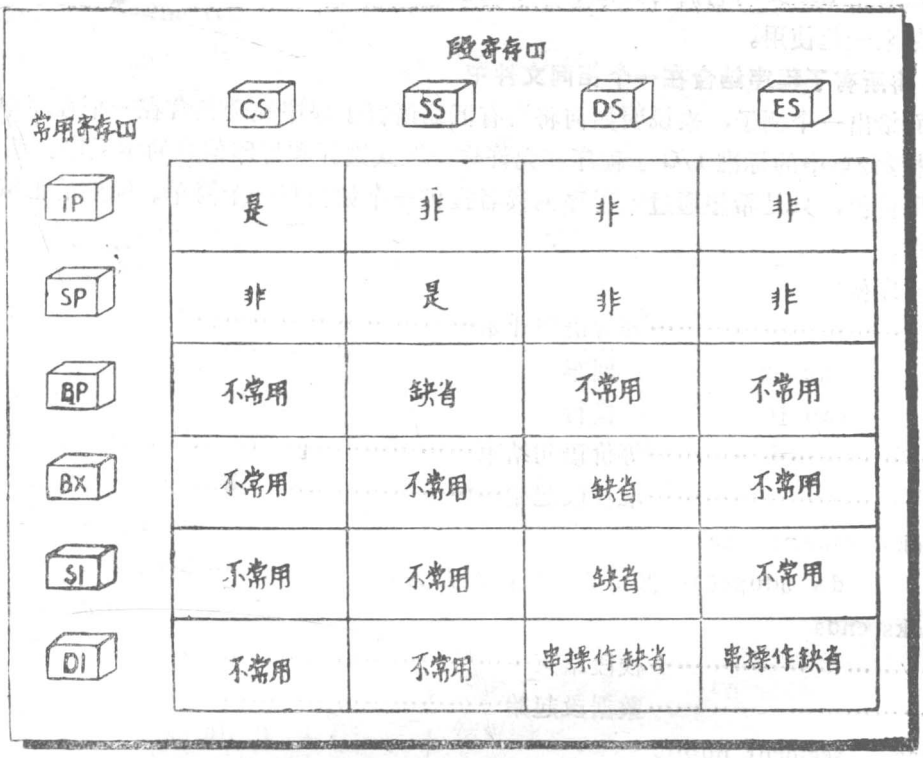


图 1-1 段寄存器的使用

数据段会减少当程序运行时要求存取数据的段开关开销。数据存放在它自己的段中，而不是存放在代码段中是一种较哲理性的概念，具有许多的优点。这是另外一个现代结构程序设计的例子，这种程序设计方法鼓励程序员使用一种“模块化”设计，这种设计思想是将所有东西放在各自的空间中。此外，由于每段只有64k字节，所以数据存放在与代码段分离的数据段中，会使代码段有更多的空间存放代码。

由于这种安排，对于任何一个特定的一片数据的偏移，在汇编时是无法知道的，只有在所有模块用 LINK 连接在一起以后才知道。所以汇编语言命令函数的 OFFSET 操作符无法返回延伸到数个汇编模块的整个数据段中正确的偏移量。但是有一个 CPU 指令 LEA（装入有效地址），它能在程序实际运行时将指定变量的偏移装入到指定的寄存器。当你需要计算信息和复杂数据结构的变量地址时不能用 OFFSET 操作符，只能用这 CPU 指令。

附加段有一个功能，例如，它可以等同于数据段；它可以是一个用于存贮操作系统数据的特殊内存区域，或者它可以作为诸如显示器 RAM 之类的特殊存贮器。在书中以后的子程序中，你可以看到我们是如何使用它的。

与其他子程序接口的二种方法

有二种使用本书中提供的子程序的方法，一种是利用文本编辑程序，在代码级上有其它汇编语言文件进行合并，另一种是将子程序分门别类放入到各个分离独立的汇编语言文

本中，然后用 LINK 将它们与其他子程序连接在一起，包括你自己写的子程序。这二种方法可合起来一起使用。

1．将所有子程序结合在一个相同文件中

下面给出一个例子，来说明如何将所有汇编语言子程序完全包含在一个程序中。这个程序包括第2章中的标准 I/O 子程序和允许你交互式选择和打印信息的主程序。该程序本身无使用价值，只是希望通过本程序向读者提供一个如何写一个简单、完整的汇编语言模块的模式。

```
; 程序例1
; ……等价语句开始……
cr      equ 13          ; 回车
lf      equ 10          ; 换行
; ……等价语句结束……
; ……堆栈段起始……
stacks segment stack
        dw 5dup(0); 保留 5 层栈，初始为 0
stacks ends
; ……栈段结束……
; ……数据段起始……
datas   segment public
menu    db cr, lf, 'Message Demonstration Program', cr, lf
        db cr, lf, 'Press 1 or 2 for messages or CTRL/C to stop!
        db 0
mess1   db cr, lf, 'messagenumber 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
; ……数据段结束……
; ……代码段开始……
codes   segment
        assume cs : codes, ss : stacks, ds : datas
; ……程序起始……
; 标准输入程序
stdin   proc far
        mov ah,1        ; 标准输入
        int 21h         ; 调用 DOS 中断
        ret             ; 返回
stdin   endp
; ……程序结束……
; ……程序开始……
```

```
; 标准输出程序
stdout      proc far
            push dx              ; 保护寄存器
            mov dl, al
            mov ah, 2            ; 标准输出
            int 21h              ; 调用 DOS 中断
            pop dx               ; 恢复寄存器
            ret
stdout      endp
; ……程序结束……
; ……程序开始……
; 信息输出标准程序
stdmessout  proc far
            push si              ; 保护寄存器
            push ax
stdmessout1:
            mov al, [si]         ; 取字节
            inc si               ; 指向下一字节
            cmp al, 0            ; 结束吗
            je stdmessoutexit    ; 是则退出
            call stdout          ; 输出
            jmp stdmessout1      ; 作循环
stdmessoutexit:
            pop ax               ; 恢复寄存器
            pop si
            ret
stdmessout endp
; ……程序结束……
; ……主程序开始……
; 显示信息选择程序
main        proc far
start:
            mov ax, datas        ; 取数据段
            mov ds, ax           ; 送到 ds
main0:
            len si, menu         ; 指向菜单信息
            call stdmessout      ; 输出信息
            call stdin           ; 取用户输出键
main1:
```

```
                cmp al,'1'          ; 为'1'
                jne main2           ; 若不是，则转
                lea si,mess1        ; 指向信息1
                call stdmessout     ; 输出
                jmp main4           ; 作循环
main2:
                cmp al,'2'          ; 为'2'
                jne main3           , 若不是，则转
                lea si, mess2       ; 指向信息3
                call stdmessout,    ; 输出
                jmp mess4           ; 作循环
main3:
                lea si, mess3       ; 指向信息3
                call stdmessout     ; 输出
main4:
                jmp main0           ; 循环
main            endp
; ……主程序结束……
code            ends
; ……代码段结束……
                end start
```

我们已经将执行一个程序需要的所有东西都放在称为CH1EX1.ASM汇编文件或模块中。从源代码中可以看到，该模块的主要成分都用省略号线分隔开，每行的中间都有一个标识指出每部分的开始和结束。

第一部分的实体是替换式，其作用是给以后程序中使用的常数一个标识名。将所有替换式放在程序的一开始是很重要的，因为在汇编该程序时，汇编程序必须一开始就知道要进行的那些替换，以便在以后程序汇编过程中的必要时刻进行替换。当然，不是一定要将替换式放在程序的一开始，只是程序员本身在编程过程中不要搞错当前哪个替换式有效。

使用替换式的一个理由是：为了使用汇编语言程序也能够在新的环境下工作，程序员往往有意用替换式来替换在其环境下可能要作变动的值。如果换了环境，只要对替换式作些变动，无须对程序本身作修改。所以将替换式放在程序一开始的另一个理由是：这样做可使修改变得很方便。另外如果替换放在前面且较适当地进行了说明，则希望将各汇编程序汇集成一个工作系统的程序可以更有效地做他自己的工作。

接下来一部分是包含有称为STACKS堆栈段的栈。注意我们已经给这个段一个属性"STACK"。正如以前说明过；在程序运行以前，操作系统通过这个属性知道堆栈段寄存器和栈指针应该建立在哪儿。对于现在这个简单的程序，我们只保留5层栈，其他程序可以要求更多些。

再接下来是数据部分。它包含有称为DATAS的数据段。数据段中装有所有需要的变

量和信息。为了与我们一般实际情况相一致，我们将该段定义为 PUBLIC，使得程序中所有其他部分都能访问它(如果有这个需要)，也能与其它模块中的数据段进行归并。(同样如果需要的话)。

最后几部分包含有代码。所有代码都放在称为 codes 这个段中。每个子程序的源代码都放在它自己的由破折号线定界的部分中。子程序的排列次序是：每个子程序都放在调用它的子程序之前，特别是主程序放在最后。

尽管这些子程序在第 2 章中正式给出时有一个引头详细描述其功能、输入、输出、所有寄存器等，但是在本章程序中使用它们时为节省空间而除去了每个子程序的引头。

由于主程序是永远循环的，所以我们不需要提供到操作系统的返回。如果你希望在程序运行时中断程序的执行，则在程序过程中任一时刻，你可以通过按 CTRL-C 回到操作系统。

该程序演示如何使用第 2 章中的STDIN和STDMESSOUT子程序。该程序使用STDIN子程序来确定你需要哪些信息，而用 STDMESSOUT 子程序来显示特定的信息。这种检查一系列可能性和根据某种选择执行的程序控制结构可以与嵌入的用户程序一起执行一个有实际意义的工作。

如何进行汇编呢？

通常，人们总是将诸如文本编辑程序(EDLIN)，汇编程序(MASM 或 ASM)和连接器(LINK)拷贝在一个软盘上，放在驱动器 A 中，而将正在开发的汇编语言程序拷贝在另一个数据软盘上，放在驱动器 B 中。

如果你是按这种约定安排的话，则欲对上述程序进行行汇编，你可以键入命令 A>MASM B：CH1EX1；

提示符"A>"出现以后，在 B 驱动器上得到一个名为 CH1EX1.OBJ 的文件。在可以运行该程序以前你还需要连接它们。这个可以通过键入下列命令来实现：

A>LINK B：CH1EX1；

同样，提示符"A>"出现以后，在 B 驱动器上则得到一个可运行程序 CH1EX1·EXE 文件。现在，可以用下列命令来运行：

A>B：CH1EX1

于是，程序将开始为你工作。

2．使用不同的汇编文件

如果你打算采用不同的文件来存放本书中的子程序，则你必须分别编辑源代码，然后汇编各个文件。每个这样的文件应该包含有一组相关的子程序。例如，某个章节中子程序可以放在相同的文件中，也就是说，所有算术子程序都放在一个大的文件中，所有图象子程序都放在另一个文件中，等等。这样每个汇编文件就形成一个可以用于特殊用途的独立的汇编模块。在软件开发过程中，这样做有许多优点，例如，可以减少编辑和汇编代码所需时间。已经证明是正确的代码可以以最终的、简洁的和目标代码形式保存起来，而需要编辑和重新汇编的代码是还有错误的模块，直至它们也变成正确为止。每当程序被测试的时候，所有目标代码模块通过 DOS 支持的连接器(LINK)连接在一起。

如果一个模块的子程序被其它子程序调用，则被调用的子程序必须在它自己的汇编语

言文件中声明 PUBLIC，而在包含有发出调用的子程序的文件中，必须将被调用的子程序声明为 EXTRN。这些指令在 IBM-PC宏汇编语言手册中第五章中已有叙述。

下面我们举一个例子说明如何实现这里将要介绍的另一种方法。事实上，此例与上例基本上是相同的，但是以不同的方法构成源代码。

让我们首先看一下包含有主程序的文件，名为 CH1EX2.ASM 其中大部分是与前面相同，包括替换式、堆栈、数据和代码。但此时存在一些外部调用。程序员常常需用到一些外部调用，因为可能有某种原因，程序员有时不将某些程序放在子程序中，而另设一个源代码文件。如果你未将这些子程序声明成外部的(EXTRN)，则汇编程序在汇编过程中将认为你把它们忘记了，指出你犯了一串严重错误。

```
; 程序例2
; ……等价语句起始……
cr          equ 13              ; 回车
lf          equ 10              ; 换行
; ……等价语句结束……
; + + + + + + + + + + 外部说明开始 + + + + + + + + + +
            extrn stdin : far, stdout : far,stdmessout : far
; + + + + + + + + + + 外部说明结束 + + + + + + + + + +
; ……栈段起始……
stacks segment stack
            dw 5dup(0)          ; 保留 5 层栈且初始值为零
stacks      ends
; ……栈段结束……
; ……数据段起始……
datas       segment public
menu        db lr,lf,'message Demonstration Program',cr,lf
            db lr,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
; ……数据段结束……
; ……代码段起始……
codes       segment
            assume cs : codes,ss : stacks,ds : datas
; ……主程序……
; 选择显示信息程序
main        proc far
start:
```

```
                mov ax,datas            ; 取数据段
                mov ds,ax               ; 放入 DS
main0
                lea si,menu             ; 指向菜单信息
                call stdmessout         ; 显示
                call stdin              ; 取用户输入键
main1:
                cmp al,'1'              ; 信息1?
                jne main2               ; 若不是则转
                lea si,mess1            ; 指向信息1
                call stdmessout         ; 显示
                jmp main4
main2:
                cmp al,'2'              ; 信息2?
                jne main3               ; 若不是则转
                lea si,mess2            ; 指向信息2
                call stdmessout         ; 显示
                jmp main4
main3:
                lea si,mess3            ; 指向信息3
                call stdmessout         ; 显示
main4:
                jmp main0
main            endp
; ……主程序结束……
codes ends
; ……代码段结束……
                end start
```

每个外部说明给出一个本汇编模块代码中未提供的子程序或变量的名字，并且赋给它一个类型。在为访问子程序和变量而确定适当的机器代码中，类型是一个基本的成分。因此，类型必须作声明。由于这些子程序是在它们自己的代码文件中，所以它们具有 far 类型且在机器语言中要求 far 调用。

在我们的例子中，CH1IO.ASM 汇编文件包含有所需的子程序。通过观察这个模块(下面)，你可以确切地知道，一般情况下组装子程序需要什么。特别是注意所有子程序都放在称为 CODES 的段中。这些段虽然名字相同，但不是公用的，也就是说，一个模块中的 CODES 段不与其他模块中相同名字的段相合并。

```
; 程序例3——I/O 模块——文件 CH1IO.ASM
; ……代码段起始……
codes           segment
```

```
; ++++++++++公用说明++++++++++
        public stdin; stdout,stdmessout
; ++++++++++公用说明结束++++++++
        assume cs : codes
; ……子程序起始……
; 标准输入子程序
stdin   proc far
        mov ah,1        ; 标准输入
        inf 21h         ; 调用 DOS 中断
        ret
stdin   endp
; ……子程序结束……
; ……子程序起始……
; 标准输出子程序
stdout  proc far
        push dx         ; 保护寄存器
        mov de,al       ;
        mov ah,2;       ; 标准输出
        int 21h         ; DOS 中断
        pop dx          ; 恢复寄存器
        ret
stdout  endp
; ……子程序结束……
; ……子程序起始……
; 从标准输出设备上输出信息
stdmessout  proc far
        push si         ; 保护寄存器
        push ax
stdmessout1:
        mov al, [si]    ; 取字节
        inc si          ; 指向下一字节
        cmp al,0        ; 结束吗
        je stdmessoutexit   ; 是则退出
        call stdout     ; 输出
        jmp stdmessout1
stdmessoutexit
        pop ax          ; 恢复寄存器
        pop si
        ret
```

```
stdmessout   endp
; ……子程序结束……
code         ends
; ……代码段结束……
             end
```

在本代码模块中，子程序声明成是公共的。我们不需要明显地声明类型，因为以上下文关系就可以知道其类型，也就是说，子程序和变量在 PUBLIC 命令出现的模块中，所以汇编程序已经知它的类型是什么。

所有这些子程序都可以在第 2 章中找到。再一次说明，为节省空间而忽略了引头。

为了汇编此程序，你应该先打入命令：

A>MASM B：CH1EX2;

来汇编主程序，然后打入命令：

A>MASM B：CH1IO

来汇编 I/O 子程序模块。现在在 B 驱动器上已经得到二个目标文件 CH1EX2.OBJ 和CH1IO.OBJ。然后你将通过下列命令将这二个目标文件连接在一起：

A>LINK B：CH1EX2 B：CH1IO;

若要运行程序则键入：

A>B：CH1EX2

则你将得到与前面例子相同的结果。

子程序与 BASIC 并用

下一个例子将说明另外一个问题，即如何在 BASIC 程序中通过 CALL 语句访问子程序。

这个例子的源代码放在 CH1EX3.ASM 文件中，作用是将内部 16 位整型数转换成 ASCⅡ二进制形式，然后通过本书第三章中的 BIN16OUT 将结果显示在屏幕上。

```
; 程序例 4
; ……外部说明……
             extsn stdout：far
; .. ……外部说明结束……
; ……代码段起始……
Codes        segment
             assume cs：Codes
; …… ……子程序起始……
; 16 位二进制转换成 asc Ⅱ 二进制程序
bin16out     proc far
; 二进制数在 DX 中
             push cx                ; 保护寄存器
             mov cx,16              ; 设置计数器
bin16out1:
```