Turbo Assembler

（汇编）用户手册

—高级技术篇

丛海莱 译

北京联想计算机集团公司
一九九〇年八月

Turbo Assembler(汇编)用户手册

——高级技术篇

丛海莱　编译

北京联想计算机集团公司

一九九0年八月

目　录

第六章　Turbo Assembler 与 Turbo C 的接口

许多程序设计人员用汇编语言编写所有的程序；而另外一些程序设计人员则更愿意用高级语言完成繁重的工作，只有在需要低级控制或高效率的代码时才使用汇编语言；还有一些程序员主要用汇编语言进行程序设计，但偶尔也使用高级语言库和高级语言结构。

作为一种实际需要，Turbo C 对 C 语言与汇编语言的混合使用提供了良好的支持，它对汇编代码与 C 的内部汇编特征提供了一种简单而明快的方法，使用这种方法可以直接将汇编代码放入 C 程序中。对设计者而言，可将 Turbo Assembler 模块分别汇编，然后与 Turbo C 代码相连接。

本章首先介绍在 Turbo C 中如何使用内部汇编，然后详细讨论如何将单独汇编过的 Turbo Assembler 模块与 Turbo C 相连接，并剖析在 Turbo C 代码中调用 Turbo Assembler 函数的过程，最后介绍如何在 Turbo Assembler 代码中调用 Turbo C 函数。（注意：凡涉及到 Turbo C 时，均指 1.5 或更高版本的 Turbo C。）下面开始具体讨论这几方面的内容。

§6.1　在 Turbo C 中使用内部汇编

如果你想出使用汇编语言改进 C 程序的理想方法，可能会想到：如果能将汇编语言指令插入 C 代码中的关键位置，那么汇编代码的高速性和低级控制特性一定能明显地改进程序的性能。同时，你可能希望避免汇编语言与 C 语言接口传统的复杂性，而且希望不改任何 C 代码就可以做到上述这几点，这样就不必改变已可以正常运行的 C 代码。

Turbo C 的内部汇编特征可以满足你的各种需要。内部汇编仅仅是一种可将任何汇编代码放入 C 程序的任何位置，并可以全面访问 C 语言常量、变量甚至函数。的确，内部汇编可以极大地改进程序的性能，因为它与严格地用汇编语言编写的程序几乎具有同样强大的功能；例如，Turbo C 库中高性能的程序代码就是用内部汇编完成的。使用内部汇编，用户可以在 C 程序中按自己的意愿加入汇编语句，而不必考虑两者之间的接口。

考虑下面的 C 代码，它是内部汇编的一个例子：

```
.
.
.
i=0;                          /*set i to 0 */
asm dec WORD PTR i;           /*decrement i(in assembler) */
i++;                          /* increment i(in c) */
.
.
.
```

第一行和最后一行是正常的 C 语句，但中间行呢？正如你可能猜想的一样，以 asm 为开始的行是内部汇编代码行。如果用调试器查看由该 C 程序的源代码所编译出的可执行码，可以发现：

```
.
.
.
mov WORD PTR [bp-02],0000
dec WORK PTR [bp-02]
inc WORD PTR [bp-02]
.
.
.
```

其中，内部汇编码 DEC 指令在编译出的代码：

```
i=0;
```

和

```
i++;
```

之间。

从根本上看，每当 Turbo C 编译程序碰到标识内部汇编的关键字 asm 时，它就将相关的汇编行直接插到编译后的代码中，只有一点发生变化：对 C 变量的 引用被转换成与之等价的适当的汇编语言形式，正如前一个例子中，对变量 i 的引用被转换成了 WORD PTR[BP-2]一样。简言之，用户可以用 asm 关键字将任意汇编码插到 C 代码任意位置。（对内部汇编码所完成的动作也有一些限制，“内部汇编的限制”一节将讨论这些限制。）

将汇编代码直接插入到 Turbo C 产生的代码中，这似乎有些危险，的确，内部汇编也有其冒险之处。Turbo C 仔细地编译其代码，以避免内部汇编码引起严重的错误。

另一方面，任何编写得很粗糙的汇编语言代码，无论是内部汇编语句还是单独的汇编模块，其运行都具有潜在的盲目性和破坏性；这是汇编语言的高速性和低级控制能力所付出的代价。除此之外，相对于纯汇编码中出现的错误而言，内部汇编码出现的错误并不常见，因为 Turbo C 注意到了许多程序设计的细节，例如进入和退出函数、传递参数、分配变量的地址等。总之，在 C 代码中加入内部汇编码可以方便地改进程序的性能，尽管你不得不花一定的代价去排除偶尔出现的汇编语言错误，但花这种代价仍然是值得的。

内部汇编中重要的几点说明

1. 为了使用内部汇编，你必须激活 Turbo C 的命令行版本，TCC.EXE。而 Turbo C 的集成环境版本 TC.EXE 并不支持内部汇编。

2. 你所拥有的 Turbo Assembler 拷贝中带有的 TLINK 很可能与 Turbo C 拷贝中带有的 TLINK 不是同一个版本。既然为了支持 Turbo Assembler，在 LINK 中增加了一些重要的功能，同时无疑还会进一步增加新的功能，所以重要的是，你最好用自己所拥有的最新版本的 TLINK 连接包含有内部汇编行的 Turbo C 模块。最安全的方法是，务必确保存放连接程序的盘上只有一个 TLINK.EXE 文件；而此 TLINK.EXE 文件应该是用户拥有的 Borland 公司的其它产品随带的 TLINK.EXE 文件版本中最新的那一个。

§6.1.1 内部汇编如何工作

通常情况下，Turbo C 直接将每个源 C 代码文件编译成目标文件，然后激活 TLINK

将这些目标文件连接成可执行的程序。图 6.1 描述了这种编译——连接过程。要开始这种过程，用户可输入命令行

TCC filename

该命令行指示 Turbo C 先将 FILENAME.C 编译成 FILENAME.OBJ，然后激活 TLINK 将 FILENAME.OBJ 连接成 FILENAME.EXE。

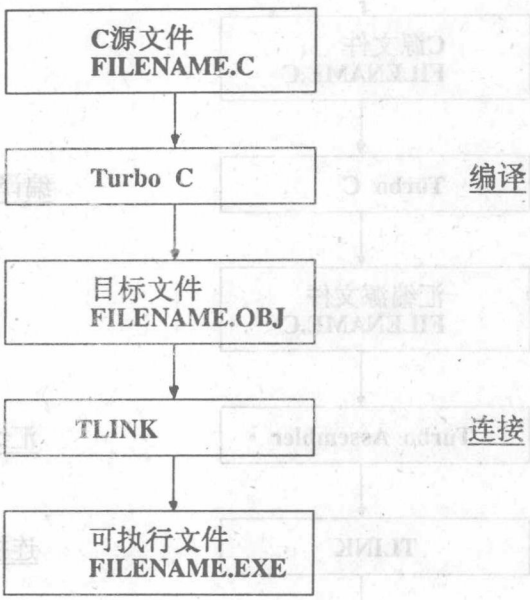

图 6.1 Turbo C 编译和连接过程

但使用内部汇编时，Turbo C 会自动在编译——连接时加入一个步骤。

Turbo C 处理每个含有内部汇编代码的模块时，首先将整个模块编译成汇编语言源文件，然后激活 Turbo Assembler 将产生的汇编码汇编成目标文件，最后激活 TLINK 对这些目标文件进行连接。图 6.2 描述了该过程，即描述了 Turbo C 如何根据含有内部汇编代码的 C 源文件产生一个可执行文件。要开始此过程，你可打入命令行：

TCC -B filename

该命令行指示 Turbo C 先编译产生 FILENAME.ASM，再激活 Turbo Assembler 将 FILENAME.ASM 汇编成 FILENAME.OBJ，最后激活 TLINK 将 FILENAME.OBJ 连接成 FILENAME.EXE。

Turbo C 只是将内部汇编码传递给汇编语言文件。这种机制的精华在于 Turbo C 无需了解怎样汇编内部代码；相反，Turbo C 将 C 代码编译成与内部汇编码同级的代码——汇编语言代码——并让 Turbo Assembler 完成汇编工作。

要了解 Turbo C 究竟怎样处理内部汇编，用户可输入取名为 PLUSONE.C 的下列程序：

```
#include <stdio.h>
int main(void)
{
```

```
    int TestValue;
    scanf("%d",&TestValue);     /* get the value to increment   */
    asm   inc WORD PTR TestValue   /* in crement it (in assembler) */
    printf("%d",TestValue);
}
```

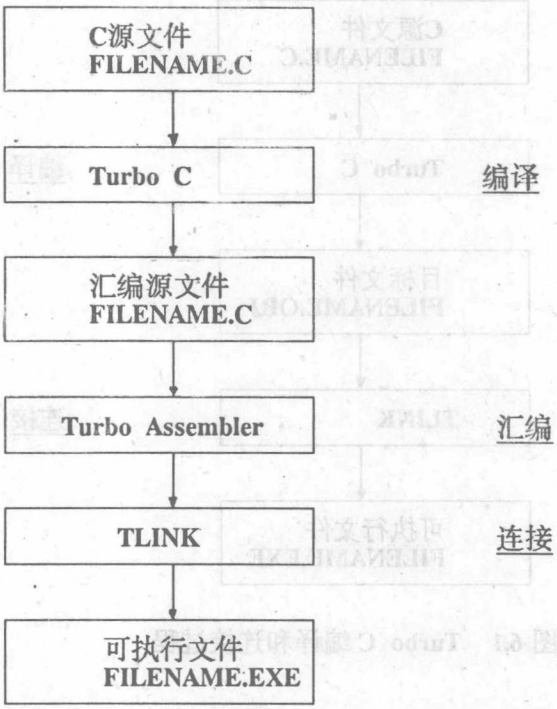

图 6.2　Turbo C 编译、汇编和连接过程

然后用命令行：

```
TCC -S plusone
```

对其进行编译。可选项-S 指示 Turbo C 在将 C 文件编译成汇编代码之后停下，所以文件 PLUSONE.ASM 现在处于用户盘上。你可以发现 PLUSONE.ASM 形式如下：

```
    ifndef  ??version
?debug  macro
    ENDM
    ENDIF
    name  Plusone
_TEXT SEGMENT BYTE PUBLIC 'CODE'
DGROUP  GROUP  _DATA,_BSS
    ASSUME  CS:_TEXT,DS:DGROUP,SS:DGROUP
_TEXT  ENDS
_DATA  SEGMENT WORD PUBLIC 'DATA'
```

```
_D@    LABEL   BYTE
_D@W   LABEL   WORD
_DATA ENDS
_BSS   SEGMENT WORD PUBLIC 'BSS'
_b@    LABEL BYTE
_b@W   LABEL WORD
        ?debug  C  E90156E11009706C75736F6E652E63
        ?debug  C  E90009B9100F696E636C7564655C7374646496F2E68
        ?debug  C  E90009B9100F696E636C7564655C73746464172672E68

_BSS   ENDS
_TEXT SEGMENT BYTE PUBLIC 'CODE'
;    ?DEBUG L 3
_main   PROC   NEAR
      push     bp
      mov      bp,sp
      dec      sp
      dec      sp
; ?debug   L 8
      lea      ax,WORD PRT [bp-2]
      push     ax
      mov      ax,OFFSET DGROUP:_s@
      push     ax
      call     NEAR PTR _scanf
      pop      cx
      pop      cx
;     ?debug    L 9
      inc      WORD PTR [bp-2]
;     ?debug    L 10
      push     WORD PTR [bp-2]
      mov      ax,OFFSET DGROUP:_s@+3
      push     ax
      call     NEAR PTR _printf
      pop      cx
      pop      cx
@1:
;     ?debug L   12
      mov      sp,bp
      pop      bp
      ret
```

```
_main ENDP
_TEXT ENDS
_DATA SEGMENT WORD PUBLIC 'DATA'
_s@   LABEL BYTE
      DB  37
      DB  100
      DB  0
      DB  37
      DB  100
      DB  0
_DATA ENDS
_TEXT SEGMENT BYTE PUBLIC 'CODE'
      EXTRN  _printf:NEAR
      EXTRN  _scanf:NEAR
_TEXT  ENDS
    PUBLIC _main
    END
```

（Turbo C 支持内部汇编，所以为你做了大量的工作。看此代码之后，想必你对 Turbo C 所做的全部工作一定十分欣赏吧！）在注释

```
;debug L 8
```

之下，你可以看到 Scanf 调用的汇编码，其后是

```
;debug L 9
inc WORK PTR [bp-2]
```

这是递增 TestValue 的内部汇编指令。（注意，Turbo C 自动将 C 语言变量 TestValue 转化成该变量的等价汇编地址[BP-2]。）在内部汇编指令之后是 printf 调用的汇编码。

重要的是要了解，Turbo C 将 Scanf 调用编译成汇编语言，将内部汇编码直接插入到输出的汇编文件中，然后将 Printf 调用编译成汇编语言。得到的结果文件是一个有效的汇编源文件，这样就可以用 Turbo Assembler 进行汇编。

如果不用可选项-S，Turbo C 将直接激活 Turbo Assembler 汇编 PLUSONE.ASM，然后激活 TLINK 将得到的目标文件 PLUSONE.OBJ 连接成可执行文件 PLUSONE.EXE。这是 Turbo C 对内部汇编进行处理的一般模式；使用-S 可选项只是为了解释这一过程，以便观察在处理内部汇编时 Turbo C 所使用的中间汇编语言步骤。当编译出将要连接成可执行程序的代码时，可选项-S 并不是特别有用，但它提供了一种方便的手段，通常可以利用它检查内部汇编码中的指令以及由 Turbo C 产生的代码指令。如果用户不清楚内部代码转换后的形式，可以使用-S 可选项检查.ASM 文件。

§6.1.1.1 Turbo C 如何知道使用内部汇编模式

一般情况下，Turbo C 直接将 C 代码编译成目标代码。有多种方法可以告知 Turbo C 支持内部汇编，先将源文件编译成汇编语言，然后激活 Turbo Assembler。

命令行参数-B 指示 Turbo C 将 C 代码编译成汇编代码，再激活 Turbo Assembler 汇编产生的代码，从而得到目标文件。

命令行参数-S 指示 Turbo C 将 C 代码编译成汇编码后停止运行。使用-S 参数时，由 Turbo C 产生的.ASM 文件可以分别汇编，并连接到其它 C 模块或汇编模块上。除了调试和检查之外，在使用了-B 参数后一般不再使用-S。

伪指令#progma

```
#progma inline
```

与命令行可选项-B 有同样的功能，它指示 Turbo C 将 C 代码编译成汇编代码，再激活 Turbo Assembler 汇编得到目标代码。当遇到#progma inline 时，Turbo C 在汇编输出模式下重新启动编译。因而，最好将伪指令#progma inline 尽可能放到 C 语言源代码的首部，因为以#progma inline 开头的任何 C 语言源代码均被编译两次，一次被正常地编译成目标码，一次被编译成汇编码。尽管这对其它任何过程都没有妨碍，但很费时间。

最后，当使用-B、-S 和#progma inline 时，如果 Turbo C 接触到内部汇编码，则编译器给出下列警告：

```
Warning test.c 6:Restarting compile using assembly in function main
```

然后以汇编输出模式重新编译，正如此时使用了伪指令#progma inline 一样。你可以使用-B 或#progma inline 避免这种警告，因为一碰到内部汇编就开始重新编译相对而言要慢得多。

§6.1.1.2 激活 Turbo Assembler 处理内部汇编

Turbo C 要激活 Turbo Assembler，首先必须找到 Turbo Assembler。但究竟怎样找到 Turbo Assembler 则依赖于 Turbo C 的不同版本而有所不同。

版本 1.5 以上的 Turbo C 希望在当前目录或 DOS 环境变量 PATH 所定义的目录之一中找到 TASM.EXE，即找到 Turbo Assembler。Turbo C 基本上可以在同一环境下激活 Turbo Assembler，用户也可以在命令行提示符下打入命令：

```
TASM
```

运行 Turbo Assembler。所以，如果 Turbo Assembler 处于当前目录或命令搜索路径所表示的任一目录中，Turbo C 就可以自动寻找并运行 Turbo Assembler 处理内部汇编。

在这一方面，1.0 和 1.5 版本的 Turbo C 稍有不同，因为这两个版本的 Turbo C 出现于 Turbo Assembler 问世之前，它们通过激活 Microsoft Macro Assembler，即 MASM 来处理内部汇编。所以，这两个版本的 Turbo C 在当前目录或命令搜索路径中寻找文件 MASM.EXE，而不是 TASM.EXM。因而，它们不能自动使用 Turbo Assembler。

注意：用户可阅读 Turbo Assembler 盘上的 README 文件，以了解如何修改这两个版本的 TCC，以便它能使用 TASM。

§6.1.1.3 Turbo C 在何处汇编内部汇编码

内部汇编码可以出现于 Turbo C 的代码段，也可以出现于 Turbo C 的数据段。出现在函数中的内部汇编码被汇编到 Turbo C 的代码段，出现在函数之外的内部汇编码被汇编到 Turbo C 的数据段。

例如，C 代码

```
/*  Table of square Values */
asm  SquareLookUpTable  label  word;
asm  dw  0,1,4,9,16,25,36,49,64,81,100;
/* Function to loop up the square of a value between 0 and 10 */
```

```
int LoopUpsquare(int Value)
{
  asm mov bx,Value ;    /* get the value to square */
  asm shl bx,1;         /* multiply it by 2 to look up in a table of
                           word-sized elemtnes */
  asm mov ax,[SquareLookUpTable+bx];    /* look up the square  */
  return(_AX);  /*  return the result  */
}
```

将为函数 SquareLookUpTable 设置的数据放入 Turbo C 的数据段，将函数 LookUpSquare 内的内部汇编码放到 Turbo C 的代码段。数据也同样可以放到代码段中；考虑下列版本的 LookUpSquare，其中 SquareLookUpTable 处于 Turbo C 的代码段：

```
/*  Function to look up the square of a value between 0 and 10 */
int LookUpSquare(int Value)
{
    asm jmp SkipAroundData  /* jump past the data table */
    /* Table of square values */
    asm SquareLookUpTable label word;
    asm  dw  0,1,4,9,16,25,36,49,64,81,100;
    SkipAroundData:
    asm mov bx,Value ;    /* get the value to square */
    asm shl bx,1;         /* multiply it by 2 to look up in a table
                             of word-sized elemtnes */
    asm mov ax,[SquareLookUpTable+bx];    /* look up the square  */
    return(_AX);  /*  return the result  */
}
```

因为 SquareLookUpTable 位于 Turbo C 的代码段，所以为了读出它，似乎需要用到段前缀 CS。事实上，在访问 SquareLookUpTable 时，这段代码在自动加上前缀 CS:之后才被汇编；Turbo C 将产生正确的汇编代码，以便 Turbo Assembler 得知 SquareLookUpTable 所在的段，再由 Turbo Assembler 产生所需要的段前缀。

§6.1.1.4 将-1 开关用于 80186/80286 指令

如果你希望使用 80186 处理器所独有的汇编语言指令，如：

```
shr ax,3
```

和

```
push 1
```

那么最方便的手段是在 Turbo C 中使用命令行可选项-1，例如：

```
TCC -1 -B heapmgr
```

其中，HEAPMGR.C 是一个包含仅属于 80186 的内部汇编指令的程序。

使用-1 可选项的根本目的是为了在编译时指示 Turbo C 使用 80186 指令集，但-1 可选项也会使得 Turbo C 将.186 伪指令插到输出的汇编文件的头部；这样可以指示 Turbo Assembler 汇编 80186 指令集。如果不使用.186 伪指令，Turbo Assembler 则将标志仅属于

80186 的内部汇编指令为错误。如果未让 Turbo C 使用全 80186 指令集，但用户又希望能汇编 80186 指令，可在包含内部 80186 指令的每个 Turbo C 模块的首部插入伪指令行：

```
asm .186
```

该行被传入汇编文件中并指示 Turbo Assembler 汇编 80186 指令。

尽管 Turbo C 对 80286、80386、80287 及 80387 处理没有提供内部支持，但可以用类似的方式启动支持 80286、80386、80287、80387 的内部汇编。你可以使用关键字 asm 和 Turbo Assembler 伪指令.286、.286C、.286P、.386、.386C、.386P、.287 及.387。

语句行：

```
asm.186
```

说明了内部汇编中很重要的一方面：使用 asm 前缀可以将任何有效的汇编行传入汇编文件，这些汇编行可以是段伪指令、等价符、宏等等。

§6.1.2 内部汇编语句的格式

内部汇编语句与一般的汇编行很相似，但也存在一些不同之处。内部汇编语句的格式是：

```
asm[<label>]<instruction/directive><operands><;or newline>
```

其中：

* 每个内部汇编语句的开头须标以关键字 asm。

* [<label>]是一个有效的汇编标号。正如普通汇编语言格式一样，方括号表示 label 是可选的。（参看后续章节“内存和地址操作数限制”，其中介绍了与 C 对应的汇编语言标号。）

* <instruction/directive>是任何有效的汇编指令或伪指令。

* <operands>指用于指令或伪指令的操作数；它可以引用 C 语言中的常量、变量和标号，但要遵从“内部汇编的限制”一节中描述的各种限制。

* <; or newline>指分号或新的一行，这两者都意味着上一个 asm 语句的结束。

§6.1.2.1 内部汇编中的分号

内部汇编码中为纯 C 代码所不能忽视的一个方面是，在 C 语句中，内部汇编语句不需要用分号作为终结符。分号可以用于终止每条语句，但行的结束也表示了语句的结束。所以，除非你计划在每行放置多条内部汇编语句（这不是一种好的程序设计方法），否则，分号就是可选的，尽管这看上去并不符合 C 语言习惯，但都与基于 UNIX 的许多编译器采用的约定保持了一致。

§6.1.2.2 内部汇编中的注解

前面描述的内部汇编语句格式中缺少了一种关键元素——注释域。可以将分号放到内部汇编语句的末尾，但象其它 C 语言语句一样，它只表示了嵌入式汇编语句的结束；在内部汇编语句中，分号并不代表注释域的开始。

那么怎样注释内部汇编代码呢？的确让人感到有些奇怪，因为采用的是 C 语言的注释形式。事实上这并不奇怪，因为 C 语言的处理器对内部汇编码的处理与对其它 C 代码的处理是一致的。这样，在包含内部汇编码的整个 C 程序中，用户可以采用一致的注释风格，也使得 C 代码和内部汇编码中均可以使用 C 中定义的符号名。例如，在：

```
.
.
.
#define CONSTANT 51
    int i;
.
.
.
.
    i=CONSTANT;  /*Set i to constant value*/
    asm sub WORD PTR i,CONSTANT; /*Subtract constant Value fron i */
.
.
.
```

中，C和内部汇编都可以使用C中定义的符号CONSTANT和i，在内部汇编码中，i被设置成0。

上例揭示了内部汇编的一个重要特征，即操作数域不仅可以直接引用C中定义的符号名，还可以直接使用C中定义的变量。从本章后面部分的内容可以看出，在汇编语言中对C变量的访问通常是一项很繁琐的工作。而且在很多应用程序中通过内部汇编语言的途径将汇编语言和C语言混合使用，这也是其根本原因。

§6.1.2.3 访问结构／联合的元素

内部汇编码可以直接引用结构元素。例如：

```
.
.
.
struct Student {
   char Teacher[30];
   int Grade;
} JohnQPublic
.
.
.
asm mov ax,JohnQPublic.Grade;
.
.
.
```

将Student类型结构JohnQPublic的成员Grade的内容装入AX寄存器。

内部汇编码还可以访问相对于基址或变址寄存器的结构元素。例如：

```
.
.
.
```