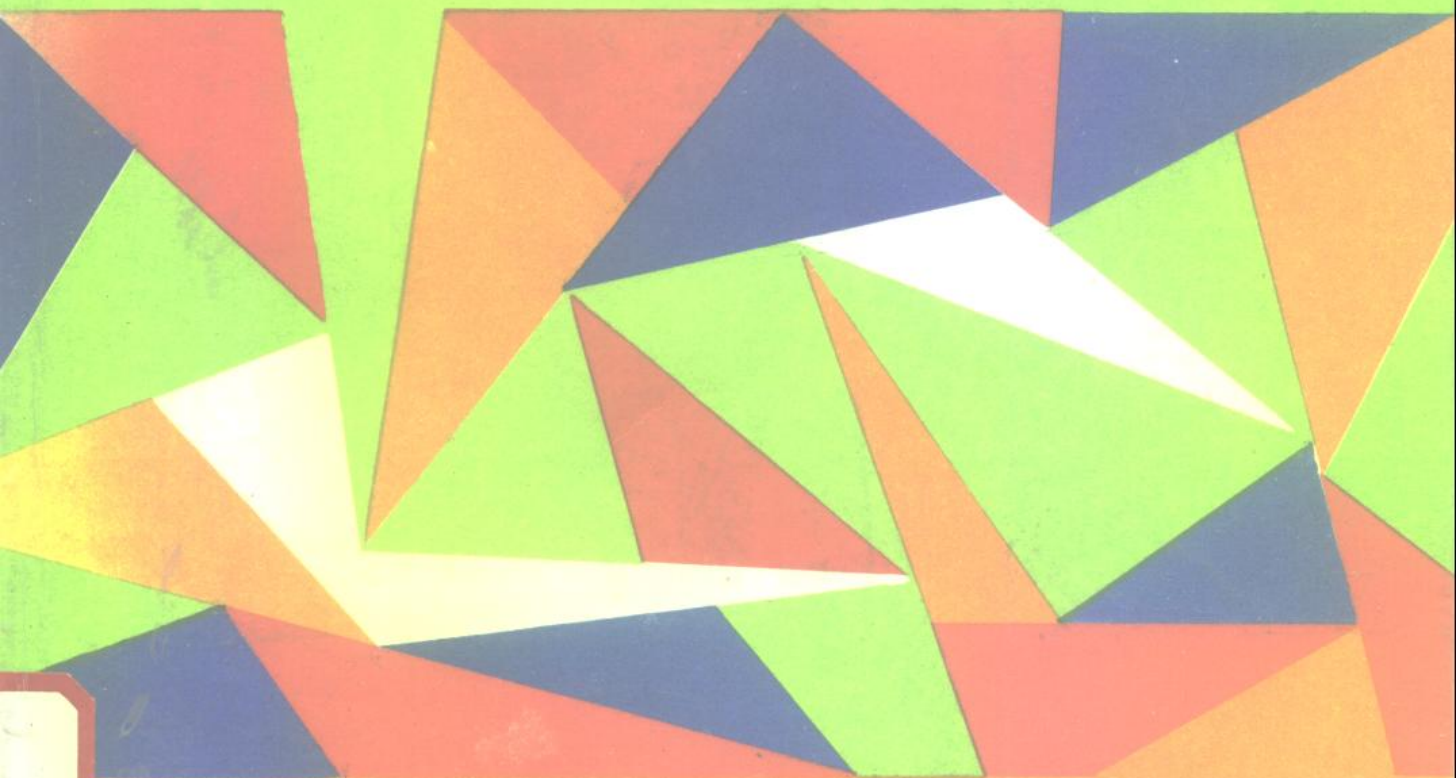
微机高级语言
与
汇编语言
接口技术和实例

天方图书创作室　李振格　编著



北京航空航天大学出版社

微机高级语言与汇编语言

接口技术和实例

天方图书创作室　　李振格　　编 著

北京航空航天大学出版社

(京)新登字 166 号

内 容 提 要

接口就是搭起一座资源共享的桥梁。Turbo C、Turbo Pascal、Turbo Prolog 等高级语言与汇编语言接口能直接使用 BIOS、DOS 的功能，直接对串行口、视频、游戏棒与鼠标等硬件进行存取，能进行内存驻留程序、设备驱动程序编程，提高了处理的速度(特别是图形处理速度)，扩展了控制。高级语言之间的接口能充分发挥各种语言的独特优势，使编程更具灵活性；高级语言与数据库管理语言的接口能减少繁琐的数据项管理(数据项插入、删除、检索、排序)；高级语言与 BIOS 和 DOS 的接口扩展了高级语言的低级功能，减少使用汇编带来的繁琐的负担。

本书针对以上问题，介绍了混合编程的基础、技巧、调试以及工程管理等，适合于编程人员使用。

书　　名：微机高级语言与汇编语言接口技术和实例
WEIJI GAOJI YUYAN YU HUIBIAN JISHU HE SHILI
作　　者：天方图书创作室　　李振格　编著
责任编辑：许传安
出　　版：北京航空航天大学出版社(北京市学院路 37 号，100083)
发　　行：新华书店总店科技发行所
销　　售：本社及各地新华书店
排　　版：天方科技公司
印　　制：北京朝阳区科普印制厂
开　　本：789×1092　1/16
印　　张：22
字　　数：563 千字
印制日期：1994 年 5 月第 1 次印刷
印　　数：8000 册
书　　号：ISBN 7-81012-491-9/TP・120
定　　价：18.00 元

前　言

Borland 公司推出众多风靡世界的 Turbo 系列语言 Turbo Pascal、Turbo C、Turbo Prolog、Turbo Basic、Turbo C++等。这些年轻的语言以其优良的性能和用户界面，很快获得世界各地用户的欢迎。Borland 率先使用的集成环境、联机帮助与热键驱动已成为当今软件用户界面的标准。独有的内部调试器使编程和调试效率倍增，开发周期骤缩。此外，Borland 辅助开发实用程序 Turbo Debugger、Turbo Profiler、MAKE、TLINK、TLIB、TCREF、GREP、TOUCH、OBJXREF 和 TC(P)HELP 等，更使 Turbo 系列语言大放光彩，形成了编程(集成环境)、调试(内部源级调试器和 Turbo Debugger)、运行(集成环境中模拟)、剖视(Turbo Profiler)和工程管理(MAKE、Project 性能)一体化的良好环境。

Turbo 系列语言都有其解决问题的方向，如果综合各自的优点，进行混合编程，那么编程水平就将更上一层楼。

现在世界软件市场异彩纷呈，Microsoft 公司开发的语言 Microsoft C、QuickC、Microsoft Pascal、Quick Pascal、Microsoft Fortran 和 Microsoft Basic、Quick Basic 也很受用户欢迎。如果能跨越 Borland 和 Microsoft 公司之间的障碍，把两者的优点结合在一起编程，那将无往而不胜。

Turbo C、Turbo Pascal、Turbo Prolog 等高级语言与汇编语言接口能直接使用 BIOS、DOS 的功能，直接对串行口、视频、游戏棒与鼠标等硬件进行存取，能进行内存驻留程序、设备驱动程序编程，提高了处理速度(特别是图形处理速度)，扩展了控制功能。

高级语言之间的接口能充分发挥各种语言的独特优势，使编程更具灵活性；高级语言与数据库管理语言的接口能减少繁琐的数据项管理(数据项插入、删除、检索、排序)；高级语言与 BIOS 和 DOS 的接口扩展了高级语言的低级功能，减少使用汇编带来的繁琐负担。

顾名思义，接口就是联络，搭起一座进行资源共享的桥梁。

接口的要素在于参数传递协议、函数返回协议、寄存器协议、数据的内部格式的差异、不同编译器生成的 OBJ 的兼容性(特别是相应的汇编语言的兼容性)、生成的 OBJ 文件所用的汇编级的选项的差别、启动代码的主次、重复的屏幕输入/输出和文件操作等相同功能的扬弃和内存管理的处理等。

本书针对以上难题进行探讨，介绍了混合编程的基础、混合编程的技巧、混合编程的调试、混合编程的工程管理等。虽然如此，但是接口技术属于高级编程的领域，加上此专题的资料不多，因此本书只是结合长期实践经验的一次写作尝试，如有不当，请读者不吝指教。

编　者

1994 年 2 月

目　录

第0章　概　述 …… 1

0.1　适合高级程序设计语言调用的汇编语言子程序的编写格式 …… 2
0.1.1　建立过程 …… 3
0.1.2　进入过程，建立一个参数表的固定基点 …… 3
0.1.3　分配局部数据空间 …… 3
0.1.4　保存调用代码的寄存器值 …… 4
0.1.5　存取参数，编写对参数的具体处理过程 …… 5
0.1.6　送返回值 …… 5
0.1.7　恢复寄存器，退出过程返回调用程序 …… 5
0.2　各高级程序设计语言调用汇编语言子程序的具体约定 …… 5
0.3　对于 Fortran 和 Pascal 的长返回值问题 …… 5

第一部分　Turbo C 与其它语言的接口

第一章　Turbo C 与汇编语言的接口 …… 8

1.1　在 Turbo C 中使用嵌入式汇编 …… 8
1.1.1　嵌入式汇编如何工作 …… 10
1.1.1.1　Turbo C 如何知道使用嵌入式汇编模式 …… 13
1.1.1.2　激活 Turbo Assembler 处理嵌入式汇编 …… 14
1.1.1.3　Turbo C 在何处汇编嵌入式汇编码 …… 14
1.1.1.4　将-1 开关用于 80186/80286 指令 …… 15
1.1.2　嵌入式汇编语句的格式 …… 16
1.1.2.1　嵌入式汇编中的分号 …… 16
1.1.2.2　嵌入式汇编中的注解 …… 16
1.1.2.3　访问结构/联合的元素 …… 17
1.1.3　嵌入式汇编示例 …… 18
1.1.4　嵌入式汇编的限制 …… 22
1.1.4.1　内存和地址操作数限制 …… 22
1.1.4.2　嵌入式汇编中缺少隐含的自动变量大小 …… 23
1.1.4.3　必须保存寄存器 …… 24
1.1.5　嵌入式汇编码相对于纯 C 代码的缺点 …… 25
1.1.5.1　降低了可移植性和可维护性 …… 25
1.1.5.2　降低了编译速度 …… 25
1.1.5.3　仅可由 TCC 使用 …… 25
1.1.5.4　损失了优化能力 …… 25
1.1.5.5　限制了对错误的反跟踪 …… 26

1.1.5.6 调试限制 …… 26
1.1.5.7 用C开发而用嵌入式汇编编译最终代码 …… 26
1.2 从Turbo C中调用Turbo Assembler函数 …… 27
1.2.1 Turbo C与Turbo Assembler的接口机制 …… 27
1.2.1.1 内存模式和段 …… 28
1.2.1.2 公共量和外部量 …… 34
1.2.1.3 链接器命令行 …… 38
1.2.2 Turbo Assembler与Turbo C的交互性 …… 38
1.2.2.1 参数传递 …… 39
1.2.2.2 保存寄存器 …… 45
1.2.2.3 返回值 …… 45
1.2.3 从Turbo C中调用Turbo Assembler函数 …… 46
1.2.4 Pascal调用约定 …… 49
1.3 在Turbo Assembler中调用Turbo C …… 50
1.3.1 链入C的启动码 …… 50
1.3.2 确保已正确设置了段 …… 51
1.3.3 执行调用 …… 51
1.3.4 在Turbo Assembler调用Turbo C函数 …… 52

第二章 Turbo C与DOS、BIOS的接口 …… 54

2.1 寄存器 …… 54
2.2 中断 …… 55
2.2.1 使用DOS中断的注意事项 …… 55
2.3 利用功能调度器实现中断 …… 56
2.4 使用BIOS中断 …… 93
2.5 小结 …… 100

第二部分 Turbo Pascal与其它语言的接口

第三章 Turbo Pascal与汇编语言的接口 …… 102

3.1 扩展Turbo Pascal …… 102
3.2 嵌入指令 …… 104
3.3 外部过程 …… 105
3.3.1 外部函数 …… 105
3.3.2 使用全程数据和过程 …… 107
3.3.3 使用Turbo Assembler …… 109
3.4 嵌入代码与外部过程的比较 …… 112
3.5 使用Turbo Debugger …… 112

第四章　再论与 Turbo Pascal 和汇编语言的接口 …… 117

4.1　Turbo Pascal 内存映象 …… 117
4.1.1　程序段前缀 …… 117
4.1.2　代码段 …… 118
4.1.3　全局数据段 …… 118
4.1.4　堆栈 …… 119
4.1.5　堆 …… 119
4.2　Turbo Pascal 中寄存器的用法 …… 119
4.3　近调用还是远调用？ …… 119
4.4　与 Turbo Pascal 共享信息 …… 120
4.4.1　$L 编译伪指令和外部子程序 …… 120
4.4.2　PUBLIC 伪指令：使 Turbo Pascal 可利用 Turbo Assembler 的信息 … 121
4.4.3　EXTRN 伪指令：使 Turbo Assembler 可利用 Turbo Pascal 的信息 … 121
4.4.4　使用段定位 …… 124
4.4.5　无效代码的消除 …… 124
4.5　Turbo Pascal 参数传递约定 …… 124
4.5.1　值参 …… 125
4.5.1.1　标量类型 …… 125
4.5.1.2　实型 …… 125
4.5.1.3　单精度、双精度、扩展的和复合型：8087 类型 …… 125
4.5.1.4　指针 …… 125
4.5.1.5　串 …… 125
4.5.1.6　记录和数组 …… 125
4.5.1.7　集合 …… 126
4.5.2　变量参数 …… 126
4.5.3　栈的维护 …… 126
4.5.4　存取参数 …… 126
4.6　Turbo Pascal 中的函数结果 …… 129
4.6.1　标量函数结果 …… 129
4.6.2　实型函数结果 …… 129
4.6.3　8087 函数结果 …… 129
4.6.4　串函数结果 …… 129
4.6.5　指针函数结果 …… 129
4.7　为局部数据分配空间 …… 129
4.7.1　分配私有静态存贮区 …… 130
4.7.2　分配动态存贮区 …… 130
4.8　由 Turbo Pascal 调用汇编语言子程序的例子 …… 131
4.8.1　通用 16 进制转换子程序 …… 131

4.8.2 交换两个变量 …… 134
4.8.3 扫描 DOS 环境 …… 137

第五章 Turbo Pascal 与 DOS 和 BIOS 的接口 …… 142

5.1 8088 寄存器 …… 142
5.2 DOS 单元 …… 143
5.3 寄存器集 …… 144
5.4 磁盘驱动功能调用 …… 146
5.4.1 报告磁盘空闲空间 …… 146
5.4.2 读取和设置文件属性 …… 147
5.4.3 目录列表 …… 151
5.5 视频功能调用 …… 155
5.5.1 报告当前视频模式 …… 155
5.5.2 设置光标大小 …… 156
5.5.3 从屏幕读字符 …… 157
5.6 时间和日期功能 …… 159
5.6.1 获取系统日期 …… 159
5.6.2 设置系统日期 …… 160
5.6.3 获取和设置系统时间 …… 161
5.6.4 获取和设置文件的时间和日期 …… 164
5.6.5 报告换档键状态 …… 169
5.7 Turbo Pascal DOS 单元 …… 171
5.7.1 DOS 单元常量 …… 171
5.7.2 DOS 单元数据类型 …… 172
5.7.2.1 DateTime 类型 …… 173
5.7.2.2 SearchRec 类型 …… 173
5.7.3 DosError 变量 …… 173
5.7.4 DOS 单元过程与函数 …… 174
5.7.4.1 中断支持子程序 …… 174
5.7.4.2 日期和时间例程 …… 174
5.7.4.3 磁盘和文件例程 …… 174
5.7.5 进程例程 …… 175

第三部分 Turbo Basic 与其它语言的接口

第六章 Turbo Basic 与 Turbo Assemlber 的接口 …… 188

6.1 传递参数 …… 188
6.1.1 不在当前数据段的变量 …… 190
6.1.2 什么类型的调用? …… 190

6.2 弹出堆栈 …… 191
6.3 为 Turbo Basic 创建一个汇编程序 …… 191
6.4 调用一个在线汇编子程序 …… 191
6.5 在内存中安装一个 Turbo Basic 子程序 …… 193
6.5.1 隐藏串 …… 194
6.5.2 绝对调用(CALL ABSOLUTE) …… 195
6.5.2.1 到一固定内存位置作 CALL ABSOLUTE …… 196
6.5.2.2 到内存不定位置作 CALL ABSOLUTE …… 196
6.5.2.3 CALL ABSOLUTE 的其他问题 …… 197
6.6 调用中断 …… 197
6.7 样本程序 …… 198

第四部分 Turbo Prolog 与其它语言的接口

第七章 Turbo Prolog 与 Turbo C 的接口 …… 202

7.1 声明外部谓词 …… 202
7.2 调用约定和参数压栈顺序 …… 202
7.3 命名约定 …… 203
7.4 Turbo Prolog 调用 Turbo C 过程 …… 204
7.4.1 说明外部谓词 …… 204
7.4.2 建立 C 函数源程序 …… 204
7.4.3 Turbo C 编译选项和连接 …… 204
7.4.4 动态存贮分配 …… 205
7.4.5 传递复合对象到其它语言的程序 …… 206
7.4.6 例子 …… 207
7.5 Turbo C 调用 Turbo Prolog …… 210

第八章 Turbo Prolog 与 Turbo Assembler 的接口 …… 213

8.1 声明外部谓词 …… 213
8.2 调用约定和参数压栈 …… 213
8.3 命名约定 …… 214
8.4 编写汇编语言谓词 …… 214
8.5 用多重流模式实现谓词 …… 219
8.6 从汇编函数调用 Turbo Prolog 谓词 …… 220
8.7 表和函子 …… 222

第九章 Turbo Prolog 与 MS-Fortran 4.0 的接口 …… 226

9.1 系统设置 …… 226
9.2 Turbo Prolog 调用 MS-Fortran 过程 …… 226

9.2.1 在 Prolog 中说明外部谓词 …… 226
9.2.2 定义 Fortran 子程序并建立源程序 …… 227
9.2.2.1 命名约定 …… 227
9.2.2.2 参数约定 …… 227
9.2.2.3 屏幕输出 …… 227
9.2.3 连接步骤 …… 228
9.2.4 例子 …… 228
9.3 Fortran 调用 Turbo Prolog …… 230
9.4 常用接口例程库、预处理程序的建立以及 Fortran 库的改造 …… 231
9.4.1 常用接口例程库的建立 …… 231
9.4.2 预处理程序 …… 232
9.4.3 Fortran 库的改造 …… 234

第十章 Turbo Prolog 访问 dBASE Ⅲ 数据文件 …… 240

10.1 Prolog 事实与 dBASE Ⅲ 记录 …… 240
10.2 dBASE Ⅲ 中 DBF 的存贮结构 …… 240
10.3 把 DBF 记录转换成 Turbo Prolog 事实 …… 241
10.4 利用 Turbo Prolog 工具库访问 dBASE Ⅲ 数据文件 …… 242
10.4.1 一次读出 dBASE Ⅲ 文件的所有记录 …… 242
10.4.2 一次读出一个 dBASE Ⅲ 记录 …… 243

第十一章 Turbo Prolog 与 DOS 系统级的接口 …… 250

11.1 访问 DOS …… 250
11.1.1 system/1 …… 250
11.1.2 system/3 …… 250
11.1.3 envsymbol/2 …… 251
11.1.4 date/3 和 time/4 …… 252
11.1.5 comline/1 …… 252
11.2 访问硬件：低级支撑 …… 253
11.2.1 bios/3 和 bios/4 …… 253
11.2.2 ptr-dword/3 …… 254
11.2.3 membyte/3 和 memword/3 …… 254
11.2.4 port-byte/2 …… 255
11.3 例子： …… 255

第五部分 混合编程程序的调试

第十二章 Turbo Debugger 调试的一个快速示例 …… 258

12.1 演示程序 …… 258

12.2 使用 Turbo Debugger …… 259
12.2.1 菜单(The menus) …… 259
12.2.2 状态行(The status line) …… 260
12.2.3 窗口(The windows) …… 260
12.3 使用 C 演示程序 …… 261
12.3.1 设置断点(Setting breakpoints) …… 262
12.3.2 利用监视(Using watches) …… 262
12.3.3 考察简单的 C 数据对象 …… 263
12.3.4 考察复杂的 C 数据的对象 …… 264
12.3.5 改变 C 数据值 …… 265
12.4 使用 Pascal 示例程序 …… 266
12.4.1 设置断点(Setting breakpoints) …… 267
12.4.2 使用监视(Using watches) …… 268
12.4.3 考察简单的 Pascal 数据对象 …… 268
12.4.4 考察复杂的 Pascal 数据对象 …… 269
12.4.5 改变 Pascal 数据值 …… 269

第十三章 启动 Turbo Debugger …… 272

13.1 准备待调试的程序 …… 272
13.1.1 准备 Turbo C 程序 …… 272
13.1.2 准备 Turbo Pascal 程序 …… 272
13.1.3 准备 Turbo 汇编程序 …… 273
13.1.4 准备 Microsoft 程序 …… 273
13.2 运行 Turbo Debugger …… 273
13.3 命令行选择项 …… 274
13.3.1 装载配置文件(-c) …… 274
13.3.2 显示更新方式(-d) …… 274
13.3.3 获取帮助(-h 与-?) …… 274
13.3.4 进程 ID 转换(-i) …… 274
13.3.5 击键记录(-k) …… 274
13.3.6 汇编模式启动(-l) …… 275
13.3.7 设置堆大小(-m) …… 275
13.3.8 鼠标器支持(-p) …… 275
13.3.9 远程调试(-r) …… 275
13.3.10 源代码处理(-s) …… 275
13.3.11 视频硬件(-v) …… 276
13.3.1.2 覆盖池大小(-y) …… 276
13.4 配置文件 …… 276
13.5 选项菜单 …… 277

13.5.1 语言命令 …… 277
13.5.2 宏菜单 …… 277
13.5.2.1 创建(Create) …… 277
13.5.2.2 停止记录(Stop Recording) …… 277
13.5.2.3 删除(Remove) …… 278
13.5.2.4 全清(Delete All) …… 278
13.5.3 显示选择命令 …… 278
13.5.3.1 显示切换 …… 278
13.5.3.2 整数格式 …… 278
13.5.3.3 屏幕行数 …… 278
13.5.3.4 制表键大小 …… 279
13.5.4 源命令路径 …… 279
13.5.5 保存选择项命令 …… 279
13.5.6 恢复选择项命令 …… 280
13.6 在 Turbo Debugger 中运行 DOS …… 280
13.7 返回 DOS …… 280

第六部分 混合编程的参考资料

附录 A TASM 命令行参考 …… 282
A.1 在 DOS 中启动 Turbo Assembler …… 282
A.2 命令行选择项 …… 284
附录 B 混合编程实用程序 …… 295
B.1 独立的 MAKE 实用程序 …… 295
B.1.1 一个快速示例 …… 295
B.1.1.1 创建一个 make 文件 …… 296
B.1.1.2 使用一个 make 文件 …… 297
B.1.1.3 步进 …… 298
B.1.2 创建 make 文件 …… 298
B.1.2.1 Make 文件的组成 …… 298
B.1.3 使用 MAKE …… 309
B.1.3.1 命令行语法 …… 309
B.1.3.2 中止 MAKE 的说明 …… 310
B.1.3.3 BUILTINS.MAK 文件 …… 310
B.1.3.4 MAKE 是如何查找 make 文件的 …… 310
B.1.3.5 TOUCH 实用程序 …… 311
B.1.3.6 MAKE 命令行选择项 …… 311
B.1.4 MAKE 出错信息 …… 311
B.1.4.1 致命错 …… 311
B.1.4.2 一般错 …… 312

B.2 Turbo Link …… 313
B.2.1 调用 TLINK …… 313
B.2.2 使用应答文件 …… 314
B.2.3 TLINK 选择项 …… 315
B.2.3.1 /x,/m,/s 选择项 …… 315
B.2.3.2 /l 选择项 …… 316
B.2.3.3 /i 选择项 …… 316
B.2.3.4 /n 选择项 …… 316
B.2.3.5 /c 选择项 …… 316
B.2.3.6 /d 选择项 …… 317
B.2.3.7 /e 选择项 …… 317
B.2.3.8 /t 选择项 …… 317
B.2.3.9 /v 选择项 …… 317
B.2.3.10 /s 选择项 …… 317
B.2.4 一些限制 …… 317
B.2.5 出错消息 …… 318
B.2.5.1 致命错 …… 318
D.2.5.2 非致命错 …… 319
B.2.5.3 警告 …… 319
B.3 TLIB:Turbo 库管理员 …… 320
B.3.1 使用目标模块库的优点 …… 320
B.3.2 TLIB 命令行的组成 …… 320
B.3.3 操作表(Operations) …… 321
B.3.4 使用应答文件 …… 322
B.3.5 改进的操作:/c 选择项 …… 322
B.3.6 例子 …… 323
B.3.7 创建一扩展词典:/E 选择项 …… 323
B.4 GREP:一种文件查找实用程序 …… 324
B.4.1 GREP 选择项 …… 324
B.4.1.1 优先级次序 …… 325
B.4.2 查找串 …… 325
B.4.2.1 正则表达式中的操作符 …… 325
B.4.3 文件说明 …… 326
B.4.4 带说明的例子 …… 326
B.5 OBJXREF:目标模块交叉引用实用程序 …… 328
B.5.1 OBJXREF 命令行 …… 328
B.5.1.1 命令行选择项 …… 329
B.5.2 应答文件 …… 329
B.5.2.1 自由形式的应答文件 …… 330

B.5.2.2 连接器应答文件 …… 330
B.5.2.3 /D命令 …… 330
B.5.2.4 /O命令 …… 330
B.5.2.5 /N命令 …… 330
B.5.3 OBJXREF报告样本 …… 331
B.5.3.1 按公用名报告(/RP) …… 332
B.5.3.2 按模块报告(/RM) …… 332
B.5.3.3 按引用报告(/RR)(缺省方式) …… 332
B.5.3.4 按外部引用报告(/RX) …… 333
B.5.3.5 按模块长度报告(/RS) …… 333
B.5.3.6 按类报告(/RC) …… 333
B.5.3.7 按未引用符号名报告(/RV) …… 334
B.5.3.8 冗长报告(/RV) …… 334
B.5.4 使用OBJXREF的例子 …… 334
B.5.5 OBJXREF出错信息和警告 …… 335
B.5.5.1 出错信息 …… 335
B.5.5.2 警告 …… 335
B.6 TCREF:源模块交叉引用实用程序 …… 335
B.6.1 应答文件 …… 336
B.6.2 与TLINK的兼容 …… 336
B.6.2.1 开关 …… 336
B.6.2.2 全局(或连接器级)报告 …… 336
B.6.2.3 局部(或模块级)报告 …… 337

参考文献 …… 338

第0章 概 述

随着微型机应用的不断普及，在各类应用软件开发过程中，为了利用各种高级程序设计语言之间的混合编程技术，人们一直关注着高级程序设计语言与汇编程序设计语言之间的接口技术问题。

众所周知，高级语言具有如下的特点：

- 数据结构丰富，具有现代化语言的各种数据结构。高级语言一般都包括整型、实现、字符型、数组类型、指针类型、结构类型、联合类型等。
- 具有结构化的控制语句如 if...else 语句、while 语句、case 语句、for 语句、repeat...until 语句、do...while 语句等等。用函数作为程序作为程序模块以实现的模块化，结构化的语言，符合现化编程风格要求。
- 丰富的函数，如字符类型判定函数、目录控制函数、图形函数、输入/输出函数、接口函数、数学函数、进行程控制函数、文本窗口显示函数和时间与日期函数等等，一般的语言都具有上百个函数，多的可达几百个函数。标准的函数可以充分加快程序的开发。
- 运算符丰富。高级语言的运算符包罗万象，如加减乘除、括号、赋值和强制类型转换等等。
- 用语言使用表意的关键字编写的程序可读性强。
- 用高级语言编写的程序可移植性好。

而汇编程序设计语言是一种除机器外最靠近机器的面向机器的语言，除伪指令与宏指令外，它与机器指令是一一对应的。程序设计者可以充分利用机器指令的各种各种特有功能，发挥编程技巧，编写出占空间小、运行效率高的高质量程序模块来。

采用汇编语言编写程序存在下列问题：

- 降低了可移植性和可维护性
- 降低了编译速度
- 损失了优化能力
- 限制了对错误的反跟踪
- 调试限制

总之，汇编语言存在着不易掌握，编程困难且费时，程序可读性差，且其程序质量直接到编程人员的技术水平的影响的缺点。与汇编语言相比，面向问题的高级程序设计语言，如 Fortran，Pascal 和 C 等具有易掌握、编写程序既容易又省时、程序可读性好、容易移植等优点。但是，一般来说，各高级程序设计语言很难充分利用硬件所具有的全部功能。

基于上述原因和各类程序设计语言的特点，权衡利弊，在开发应用程序时，人们自然而

然地将注意力转向高级程序设计语言与汇编程序设计语言的混合编程技术上。设想用汇编程序设计语言来编写应用软件中少量的、最低层的、调用频率最高的且直接影响系统效率的或用高级语言难以实现的程序模块，而用高级程序设计语言来编写其它的大量的上层模块。这样，两者综合使用，充分利用两类程序设计语言的长处，既提高了软件的质量，又不影响软件的开发周期。

当然，要实现这一全理的设想，其关键在于必须搞清高级程序设计语言与汇编程序设计语言之间的接口技术和约定。

另外，各种高级语言各有专攻，优劣之处各不相同。一个大系统软件可能存在着使用不同语言的众多程序员；另外，一个程序员可能改变了使用的语言，比如从 Pascal 转到 C 或从 C 转到 Pascal，而想重用已前的编写的函数。因而也有必要在高级语言之间能互通有无，互相调用函数和过程。

近年来，作者在教学与科研中经常遇到这个问题。在项目开发中集中对 Borland 公司高级程序设计语言，如 C、Pascala、C、Basic 和其宏汇编程序设计语言(Turbo Assmbler、TASM)之间的接口技术与约定作了深入的探讨与实践，进而又对 Microsoft 系统的高级程序设计语言和汇编语言之间的接口技术进行对比与分析，摸索出一套较完整的接口技术与规则，有效地解决了上述问题。现将它综述于下，希望它有助于促进软件开发。

0.1 适合高级程序设计语言调用的汇编语言子程序的编写格式

无论是 Microsoft 公司的，还是 Borland 公司的系统语言，调用程序与被调用程序之间都是通过栈来传递参数的，这是这两个公司的产品的共同点，所以，一般要编写一个能被高级程序设计调用的汇编语言子程序都应按下现规则与格进行编写。就具体的程序设计语言而言，只要在使用这些规则与格式时注意到具体语言的某些特殊约定就可以了。

这种汇编子程序的格式与编写步骤如下：

- 建立过程；
- 进入过程；
- 分配局部数据存储区；
- 保留寄存器的值；
- 按各高级程序语言的参数传递规则存、取各参数值，编写处理这些数据的子程序体；
- 传送返回值；
- 恢复寄存器值并退出过程。返回时，根据不同的高级程序设计语言的约定，删除(或不删除)栈中的参数，恢复 sp 到调用它之前的值。

其具体框架如下：

```
        .MODEL[small |large]            ;建立过程
        CODE
                PUBLIC procname
procname    PROC [near|far]
            PUSH    BP                  ;进入过程
```

```
                MOV     BP,SP
                SUB     SP,Napace       ;分配局部数据空间
                PUSH    SI              ;保存调用者的寄存值
                PUSH    DI
                .   .   .               ;根据约定取参数值、过程体
                POP     DI              ;恢复保留的各寄存器之值
                POP     SI
                MOV     SP,BP           ;释放局部数据空间
                POP     BP              ;恢复 BP
                RET     [size]          ;恢复(或不恢复)SP,并返回
procname        ENDP
                END
```

下面逐一解释框架中各步骤的作用：

0.1.1 建立过程

以 MASM(或 TASM)5.0为例，可用简化段指示符来说明. MODEL 定义存储模式，它可有 SMALL(小模式)、MEDIUM(中模式)、LARGE(大模式)和 HUGE(巨模式)等几种选择，要注意的是它必须与高级语言所采用的存储模式相一致。通常，FORTRAN 常用 LARGE 模式，而 C 常用 SMALL 模式. CODE 定义代码段，用 PUBLIC 指示符说明过程名 procname 为公用的。过程名的命名规则应与相应的高级语言相一致，具体见表一。

0.1.2 进入过程，建立一个参数表的固定基点

指令

```
PUSH    BP
MOV     BP,SP
```

就是为此目的而设立的。首先保留调用代码的 BP 值，然后将栈指针传送给 BP。这样，汇编子程序中可以以 BP 为基准的来存取位于栈中的各参数的值(或地址)了。这时，各参数都可以编址 BP 为基准的一个固定的偏移位置上，便于后面代码的引用。以 FORTRAN 程序中的调用语句为例，CALL SUBI(A,B,C)执行后，它将实在参数的远地址(段址:偏移)顺序压入栈(STACK)之中，再压入一个返回地址(段址:偏移)，接着将控制转到了程序 SUBI 的入口处。执行 PUSH BP 和 MOV BP、SP 之后，其栈的状况如图0.1所示。

从图中看到，实参 A、B、C 在栈中的地址均可用 BP 加一常数偏移量，分别用 BP+14、BP+10、BP+6来表示。

0.1.3 分配局部数据空间

这一步可以根据子程序中的需要而定，不是必须的。通常在栈中保留一段空间，用 SUB SP、Napace 指令来实现。建议 Napace 最好为偶数，这是因为栈的 PUSH 和 POP 操作指令都是以2字节为单位的。假设 Nspace=4、那么指令:SUBSP,4执行后，变将 SP 就改变了，并保留了4个字节的空间，这个局部空间的位置可用相对于 BP 的负位移值来确定，分别可用 BP-2和 BP-4来表示。如有必要初始化，则可用指令序列：

```
MOV     WORD PTR [BP-2],  0
MOV     WORD PTR [BP-4],  0
```

将局部数据空间初始化为零

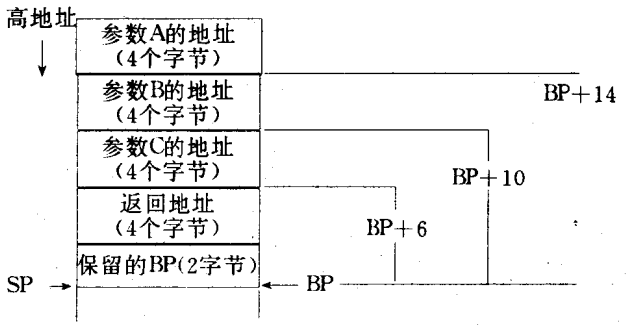


图0.1 存放参数的堆栈

表0.1 参数传递与返回值约定

高级语言	MS FORTRAN	MS PASCAL	MS C	TURBO C
过程PROC的属性	一般FORTRAN采用大存储模式，应使用FAR。	一般使用FAR。	对应小模式用NEAR；对应中、大、巨大模式用FAR	MSC
过程命名约定	FORTRAN仅识别前6个字符，且大、小写不敏感。汇编过程名最好不超过6个字符。	PASCAL识别前8个字符大、小写而敏感，汇编过程名最好不超过8个字符。	C编译将与C公用的标识符前均加一下划线前缀，且对大、小写字符敏感，仅识别前8个字符，汇编过程名应注意这些约定	同MSC
参数传递规则	1)按远引用传递。 2)参数按FORTRAN调用代码中出现的顺序，由左至右压入栈中，即第一个参数在栈的最高位置。	1)参数值传送，变量参数按地址传送。 2)参数按PASCAL调用代码中出现的顺序，由左至右逐个压入栈中	1)按值传送；指针和数组按地址传送。 2)参数按C调用代码中出现的逆序，由右至左逐个压入栈中，即最右边的参数在栈的最高位置。	同MSC
子程序返回调用程序的约定	必须重置SP，使其值恢复到参数压栈前之值，即要用RET size指令返回其中size＝n＊4或size＝n＊4＋2。	同FORTRAN一样也要用RET size来返回。	不必重置SP值，这由C编译程序在调用代码中完成，子程序中只要用指令RET返回	同MSC
函数值返回的约定	1)返回值数据类型为简单型且长度不大于4字节的通过寄存器返回如下： 1个字节 AL 2个字节 AX 4个字节 DX:AX 2)长返回值问题见三	同FORTRAN。	返回值根据类型如下： char unsigned char AX int short unsigned AX enum AX long unsigned DX:AX doblue float fac 指针:AX near * AX far * DX:AX	float double：在8087的TOS寄存器ST(O)之中 struct：静态存储器中，指针在AX或DX:AX之中

0.1.4 保存调用代码的寄存器值

对于一个被高级语言调用的汇编语言子程序来说，除BP已在进入过程时保留外，一般还应免SI、DI之值。DS，SS，CS之值自然不能随意破坏，而AX、BX、CX、DX和ES之值，在过程中可以改变。用PUSH指令保留各寄存器之值。

```
PUSH    SI
PUSH    DI
```

0.1.5 存取参数，编写对参数的具体处理过程

如图4所示，各参数的位置相对于 BP 的位移随该子程序的参数个数和顺序而确定而确定(对于各高级语言具体有点细节上不同)，这样，变可以用 BP+D 来引用这样，其中 D 为相对于 BP 的偏移。压入栈中的是每个参数远地址(段址:偏移)，所以在编写过程体时，一定要记住这一点，先取出参数的地址，再按地址存取参数值。一般使用两条指令达到将某个参数值传送到寄存器 AX 之中的目的：

```
LES     BX,[BP+14]      ; 提取参数 A 的地址 ES:BX
MOV     AX,ES:[BX]      ; 将参数 A 之值传送到 AX
```

掌握好这一点，其它问题就是具体编程问题了。为了程序可读，也可以利用 EQU 伪指令将各参数用相应符号来表示，如

```
A EQU [BP+14]
B EQU [BP+10]
C EQU [BP+6]
```

0.1.6 送返回值

对于通过参数送返回值(如 FORTRAN 中的 SUBROUTINE 和 PASCAL 中的 PROCEDURE)的问题只要掌握第5步中的存取规则就可以解决了。而对于函数值(FORTRN，PASCAL 和 C 中的 FUNCTION)的返回值问题，情况比较复杂，将在下面单独叙述；

0.1.7 恢复寄存器，退出过程返回调用程序

恢复寄存器，释放局部数据空间，这一步是2、3、4的 逆过程。用指令序列

```
POP     DI
POP     SI
MOV     SP, BP
POP     BP
```

来完成。

至于返回调用程序所使用的指令形式有 RET size 和 RET 两种。它随高级语言不同而不同。对于 FORTRAN 和 PASCAL 要用 RET size，其中 size 为参数所占栈的字节总数；而对于 C 语言只要简单地使用 RET 指令即可。

0.2 各高级程序设计语言调用汇编语言子程序的具体约定

各高级程序设计语言调用汇编语言子程序时，总的来说，都使用栈来传递参数，但就具体特定的语言而言，还有若干细节上的差异，在表一列表加以对比。

0.3 对于 FORTRAN 和 PASCAL 的长返回值问题

对于 FORTRAN 和 PASCAL 函数返回实型数时，其函数的返值机制与返回字符型、整型或逻辑型(布乐型)不同。其具体约定如下：

当 FORTRAN(或 PASCAL)在调函数子程序段之前,采取如下特殊动作:

- 在调用代码的栈中,分配一块能容纳函数返回值的数据空间;
- 在调用子程序时,除如前所述将实在参数的地址(或值)逐个压栈之外,还额外多压入一个参数,它只占2个字节。这个参数包含有保存实际函数返回值的存储空间的偏移地址 。所以在 FORTRAN(或 PASCAL)带长返回值与不带长返回值的栈的布局稍有不同,它直接影响到对各参数的相对于 BP 的偏移量,应引起注意。具体示意如图0.2、图0.3。

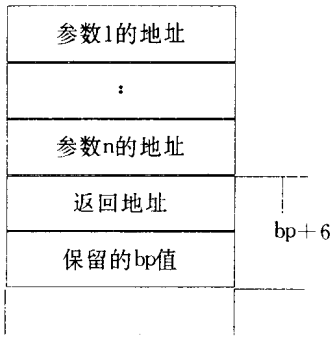


图0.2 不带长返回值的栈布局

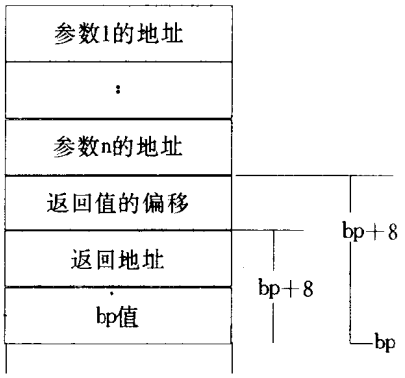


图0.3 带长返回值的栈布局

根据上述约定,如果返回实型函数值时,应完成如下动作:

- 将函数子程序段获得的函数值存储到按返回值偏移指定的存储空间中去;
- 将返回值的偏移(即图(b)中的 bp+6中之值)复制到 AX 之中,将 SS 之值复制到 DX 之中。
- 用 RET size 返回时应注意多加2个字节,即 size=n * 4+2,其中 n 为参数个数。

第一部分

Turbo C 与其它语言的接口

Turbo C 与汇编语言的接口

Turbo C 与 DOS、BIOS 的接口

第一章

Turbo C 与汇编语言的接口

许多程序设计者能够——而且的确——用汇编语言编写所有的程序；而另外一些程序设计者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；还有一些程序设计者主要用汇编语言进行程序设计，只偶而用到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 代码的嵌合使用提供了不只一种，而是两种支持机制。Turbo C 的嵌入式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 语言函数中。对于宁愿全部使用汇编语言设计各独立模块的汇编程序设计者而言，可将 TurboAssembler 模块分别汇编，然后与 Turbo C 代码相链接。

本章首先介绍在 Turbo C 中如何使用嵌入式汇编，然后详细讨论如何将分别汇编过的 Turbo Assembler 模块与 Turbo C 相链接，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。

1.1 在 Turbo C 中使用嵌入式汇编

如果试图想象出使用汇编语言改进 C 语言程序的理想方法，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，用户可能希望避免汇编语言与 C 语言接口传统的复杂性，而且希望不改变任何 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 的嵌入式汇编特征可以满足用户的各种愿望。嵌入式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，甚至函数的能力。的确，嵌入式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嵌入式汇编完成的。使用嵌入式汇编，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，它是嵌入式汇编的一个例子：

```
.
.
```

```
.
i=0;                          /* set i to 0 (in C)              */
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？正如用户可能猜想的一样，以 asm 为开始的行是嵌入式汇编代码行。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 WORD PTR [bp-02]
inc WORD PTR [bp-02]
.
.
.
```

其中，嵌入式汇编码 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 编译程序碰到标识嵌入式汇编的关键字 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 WORDPTR [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（对嵌入式汇编码所完成的动作也有一些限制，“嵌入式汇编的限制”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，而事实上，嵌入式汇编也的确有其冒险之处。尽管 Turbo C 仔细地编译其代码，以避免与嵌入式汇编码交互时所潜在的危险，但毫无疑问，功能性错误的嵌入式汇编码肯定会引起严重的错误。

另一方面，任何编写得很粗糙的汇编语言代码，无论是嵌入式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，嵌入式汇编码中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 代码中加入嵌入式汇编码可以方便地改进程序的性能，尽管用户不得不花一定的代价去排除偶而的汇编语言错误，但花这种代价仍然是值得的。

嵌入式汇编中重要的几点说明：

1. 为了使用嵌入式汇编，用户必须激活 Turbo C 的命令行版本，即 TCC. EXE。而 Trubo C 的用户接口版本 TC. EXE 并不支持嵌入式汇编。

2. 用户所拥有的 Turbo Assembler 拷贝盘中带有的 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 TINK 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，用户最好用自己所拥有的最新版本的 TLINK 链接包含有嵌入式汇编行的 Turbo C 模块。最安全的方法是，用户务必确保存放链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 文件中版本号最新的那一个。

1.1.1　嵌入式汇编如何工作

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，将这些目标文件链接成可执行的程序。图1.1描述了这种编译——链接过程。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 链接成 FILENAME. EXE。

但使用嵌入式汇编时，Trubo C 会自动在编译——链接链中加入一个步骤。

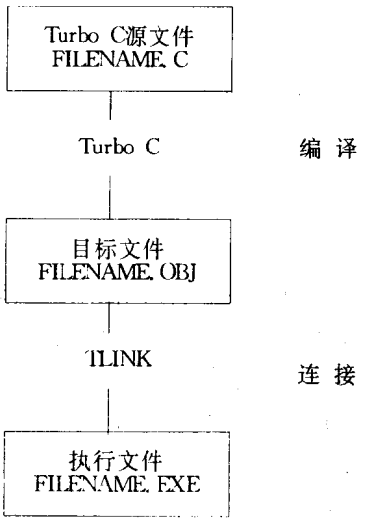


图1.1　Turbo C 编译和连接循环

Turbo C 在处理每个含有嵌入式汇编代码的模块时，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链接。图1.2描述了该过程，即描述了 Turbo C 如何根据含有嵌入式汇编代码的 C 源文件产生一个可执行文件。要开始此过程，用户可打入命令行

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 链接成 FILENAME. EXE。

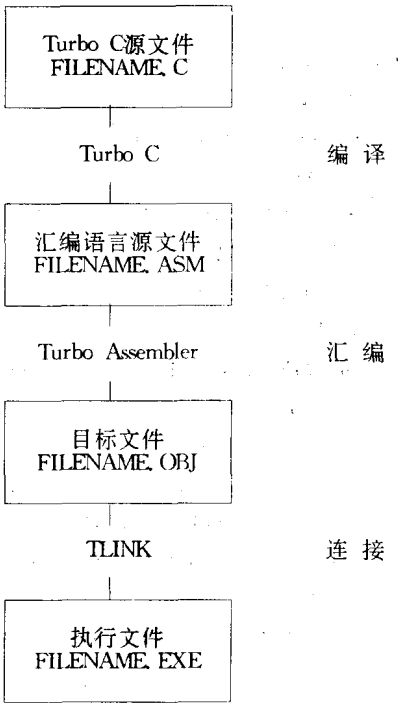


图1.2　Turbo C 编译、汇编和连接

Turbo C 只是将嵌入式汇编码传递给汇编语言文件。这种机制的精华在于 Turbo C 无须了解怎样汇编嵌入式代码；相反，Turbo C 将 C 代码编译成与嵌入式汇编码同级的代码——汇编语言代码——并让 Turbo Assembler 完成汇编工作。

要了解 Turbo C 究竟怎样处理嵌入式汇编，用户可输入取名为 PLUSONE. C 的下列程序：

```
#include <stdio.h>
int main(vvid)
{
  int TestValue;
  scanf("%d",&TestValue);          /* get the value to increment    */
  asm   inc WORD PTR TestValue  /* in crement it (in assembler) */
  printf("%d",TestValue);          /* print the incremented value  */
}
```

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 文件在编译成汇编代码之后停下，所以文件 PLUSDNE. ASM 现在处于用户盘上。用户可以发现 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 E90009B9100F696E636C7564655C737464696F2E68
       ?debug       C E90009B91010696E636C7564655C7374646172672E68
 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 PTR [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 支持嵌入式汇编，所以为用户做了大量的工作。看此代码之后，想必用户对 TurboC 所做的全部工作一定十分欣赏吧！在注释行

```
;debug L 8
```

之下，用户可以看到 Scanf 调用的汇编码，其后是

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

这是递增 TestValue 的嵌入式汇编指令(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])。在嵌入式汇编指令之后是 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，将嵌入式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 链接成可执行文件 PLUSONE. EXE。这是 Turbo C 对嵌入式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，以便观察在处理嵌入式汇编时 Turbo C 所使用的中间汇编语言步骤。当编译出将要链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嵌入式汇编码中的指令以及由 Turbo C 产生的代码指令。如果用户不清楚嵌入式代码转换后的形式，可以使用－S 可选项检查. ASM 文件。

1.1.1.1 Turbo C 如何知道使用嵌入式汇编模式

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 支持嵌入式汇编，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，并链接到其它 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＃pragma

＃pragma 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 汇编已得到的代码。当遇到＃pragma 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＃pragma inline 尽可能放到 C 语言源代码的首部，因为以＃pragma 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＃pragma inline 时，如果 Turbo C 接触到嵌入式汇编码，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＃pragma inline 一样。用户可以使用－B 或＃pragma inline 避免这种警告，因为一碰到嵌入式汇编就开始重新编译相对而言要慢得多。

1.1.1.2　激活 Turbo Assembler 处理嵌入式汇编

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 处理嵌入式汇编。

在这一方面，1.0和1.5版本的 Turbo C 稍有些不同，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 Maro Assembler，即 MASM 来处理嵌入式汇编。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. EXE。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。

1.1.1.3　Turbo C 在何处汇编嵌入式汇编码

嵌入式汇编码可以出现于 Turbo C 的代码段，也可以出现于 Turbo C 的数据段。出现在函数中的嵌入式汇编码被汇编到 Turbo C 的代码段，出现在函数之外的嵌入式汇编码被汇编到 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 内的嵌入式汇编码放到 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。

1.1.1.4 将－1开关用于80186/80286指令

如果用户希望使用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的嵌入式汇编指令的程序。

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 Tur-

boAssembler 汇编全80186指令集。如果不使用. 186伪指令，Turbo Assembler 则将仅属于80186的嵌入式汇编指令作上出错标志。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，可在包含嵌入式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的嵌入式汇编。用户可以使用关键字 asm 和 Turbo Assembler 伪指令. 286、. 286C、. 286P、. 386、. 386C、. 386P、. 287及. 387。

语句行

```
asm  .186
```

阐述了嵌入式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。

1.1.2 嵌入式汇编语句的格式

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

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

其中，

- 每个嵌入式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 Turbo 汇编中有关"内存和地址操作数限制"的内容)。
- <instruction/directive>是任何有效的汇编语言指令或伪指令。
- <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"嵌入式汇编的限制"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。

注意：有关 label(标号)的重要说明，请参看"内存和地址操作数限制"一节。

1.1.2.1 嵌入式汇编中的分号

嵌入式汇编码中为纯 C 代码所不能忽视的一个方面是，在 C 语句中，嵌入式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，除非用户计划在每行放置多条嵌入式汇编语句(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，但却与基于 UNIX 的许多编译器采用的约定保持了一致。

1.1.2.2 嵌入式汇编中的注解

前面描述的嵌入式汇编语句格式中缺少了一种关键元素——注释域。可以将分号放到嵌入式汇编语句的末尾，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；在嵌入式汇编语句中，分号并不代表注释域的开始。

那么怎样注释嵌入式汇编代码呢？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 语言的预处理器对嵌入式汇编码的处理与对其它 C 代

码的处理是一致的。这样，在包含嵌入式汇编码的整个 C 程序中，用户可以采用一致的注释风格，也使得在 C 代码和嵌入式汇编代码中均可以使用 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 和嵌入式汇编都可以使用 C 中定义的符号 CONSTANT 和 i，在嵌入式汇编码中，i 被设置成0。

上例揭示了嵌入式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嵌入式汇编语言的途径将汇编语言和 C 语言混合使用，这也是其根本的原因。

1.1.2.3 访问结构/联合的元素

嵌入式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。

嵌入式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，

```
.
.
.
asm mov bx,OFFSET JohnQPublic;
```

```
asm mov ax,[bx].Grade;
.
.
.
```

同样将 JohnQPublic 的成员 Grade 装入 AX。因为 Grade 在结构 Student 中的偏移量为30，所以上例事实上就是

```
.
.
.
asm mov bx, OFFSET JohnQPublic;
asm mov ax, [bx]+30;
.
.
.
```

相对于指针寄存器访问结构元素的能力是非常强大的，因它允许嵌入式汇编码处理结构数组并传递指向结构的指针。

然而，如果用嵌入式汇编码访问的两个或多个结构具有相同的成员名，则必须插入：

```
asm mov bx, [di].(struct tm) tm   hour > alt
```

例如，

```
.
.
.
struct Student {
        char   Teacher[30];
        int   Grade;
} JohnQpublic
.
.
.
struct Teacher {
        int   Grade;
        long   Income;
};
.
.
.
asm   mov   ax,JohnQPublic.(struct student) Grade
.
.
.
```

1.1.3　嵌入式汇编示例

到此为止，本书中已出现了使用嵌入式汇编的若干代码片段，但仍没有列举一个可以工

作的嵌入式汇编程序。本节将介绍一个使用嵌入式汇编的程序，该程序通过对嵌入汇编的使用，极大地加快了将正文转换成大写形式的过程。作为一个例子，本节列出的代码既介绍了嵌入式汇编的功能，同时可以作为用户开发自己的嵌入式汇编码的范例。

先花一点篇幅分析样例程序所要解决的程序设计方面的问题。要设计一个名为 StringToUpper 的函数将一个串复制成另一个串，在复制过程中将所有的小写字符转换成大写字符。还要使该函数能对所有内存模式中的所有字符串进行处理。要做到这一点，最好的方法是传递指向串的远指针，因为可以将指向串的近指针强制成指向串的远指针，而反之则不成立。

很遗憾，这里涉及到程序性能方面的问题。尽管 Turbo C 能够很好地处理远指针，但 Turbo C 中远指针的处理比近指针的处理要慢得多。这不是 Turbo C 独有的缺陷，而是在8086中使用高级语言编程时所不可避免的。

另一方面，串和远指针处理是汇编语言的特长。从逻辑上看，可以用嵌入式汇编处理远指针和串的复制，而让 Turbo C 完成其它工作。程序 STRINGUP.C 如下：

```
/* Program to demonstrate the use of stringToUpper(). It
   Calls StrintToUpper to convert TestString to uppercase in
   UpperCaseString, then prints UpperCaseString and its length. */
#pragma inline
#include <stdio.h>
/* Function prototype for StringToUpper() */
extern unsigned int StringToUpper(
unsigned char far * DestFarString,
unsigned char far * SourceFarString);

#define MAX  STRING  LENGTH 100

char * TestString = "This Started Out As Lowerase!";

char UpperCaseString[MAX  STRING  LENGTH];

main()
{
    unsigned int StringLength;

    /* Copy an uppercase version of TestString to UpperCaseString */
    StringLength = StringToUpper(UpperCaseString, TestString);

  /* Display the rusults of the conversion */
    printf("Origianl string:\n%s\n\n", TestString);
    printf("Uppercase String:\n%s\n\n", UpperCaseString);
    pritnf("Number of characters: %d\n\n", StringLength);
}
```

```
/* Function to perform high-speed translation to uppercase from
one far string to another

Input:
    DestFarString          -array in which to store uppercased
                           string (will be zero-terminated)
    SourceFarString        -string containing characters to be converted
                           to all uppercase (must be zero  terminated)

    Returns:               The length of the source string in characters,
                           not counting  the terminating zero.  */

unsigned int StringToUpper(    unsigned char far *DestFarString,
                               unsigned char far *SourceFarString)
{
    unsigned int CharacterCount;

    #define LOWER  CASE  A 'a'
    #define LOWER  CASE  Z 'z'
      asm ADJUST  VALUE  EQH  20h;     /* amount to subtract from lowercase
                                          letters to make them uppercase */
      asm  cld;
      asm  push ds;                    /* save C;s data segment          */
      asm  lds si,SourceFarString;     /* loead far pointer to source string */
      asm  les di,DestFarString;       /* load far pointer to souce string */
      CharacterCount = 0;              /* count of characters            */
  StringToUpperLoop:
      asm  lodsb;                      /* get the next character         */
      asm  cmp al,LOWER  CASE  A;      /* if <a the it's not a lowercase letter */
      asm  jb SaveCharacter;
      asm  cmp al,LOWER  CASE  Z;      /* if >z then it's not a lowercase letter */
      asm  ja SaveCharacter;
      asm  sub al,ADJUST  VALUE
  SaveCharacter:
      asm  stosb;                      /* save the character             */
      CharacterCount++;                /* coutn thi character            */
      asm  and al,al                   /* is this the ending erao?       */
      asm  jnz StringToUpperLoop       /* no, process the nest character, if any */
      CharacterCount--;                /* don't count the terminating zero */
      asm  pop ds;                     /* restore C's data segment       */
}
```

运行时,STRINGGUP.C 给出如下输出:

```
Original string:
```

```
This Started Out As lowercase!

Uppercase string:
THIS STARTED OUT AS LOWERCASE!

Number of characters: 30
```

的确将小写字符转换成了大写字符。

STRINGUP.C 的核心是函数 StringToUpper，它承担着拷贝字符串和转换成大写字符的全过程。函数 StringToUpper 是用 C 和嵌入式汇编书写的，并接受两个远指针作为其参数。一个远指针指向正文串；一个远指针指向另一个正文串，第一个正文串被复制到第二个正文串中，并且其中的小写字符全部变成了大写字符。函数声明和参数定义是用 C 处理的，StringToUpper 的函数类型出现在程序的首部。主程序将 StringToUpper 作为纯 C 代码进行调用。简言之，尽管 StringToUpper 中包含有嵌入式汇编代码，但仍可以体现出 TurboC 程序设计的优点。

StringToUpper 函数体是用 C 和嵌入式汇编混合书写的。汇编语言用于从源串中读出每个字符，检查是否需要转换，将字符转换成大写字符，将字符写入目的串中。在 StringToUpper 的嵌入式汇编码中使用了功能强大的 LODSB 和 STOSB 串指令读取和写入字符串。

在编写 StringToUpper 时，因为可以预见并不需要访问 Turbo C 数据段中的任何数据，所以在函数的开始便将 DS 压入堆栈，然后设置新的 DS，使之指向源串，并在函数的其余部分保持不变。相对于纯 C 代码而言，嵌入式汇编的一大优点是能够在函数的开始一次性地装载远指针，在函数结束之前不必再次装载。相反，每次使用时，Trubo C 和其它高级语言通常需要重新装载远指针。仅装载一次远指针，这意味着 StringToUpper 处理远指针的速度就与处理近指针的速度一样快。

StringToUpper 中另外一个令人感兴趣的方面是 C 语句与汇编语句的混合使用。#define 用于设置 LOWER　CASE　A 和 LOWER　CASE　Z，汇编伪指令 EQU 用于设置 ADJUST　VALUE，但所有的这三个符号在嵌入式汇编码中的使用方式是相同的。Turb C 预处理器替换 C 中定义的符号，而 ADJUST　VALUE 则由 Turbo Assembler 进行替换，但两者都可用于嵌入式汇编码中。

在 StringToUppe 中处处可见处理 CharacterCount 的 C 语句，这说明 C 代码和嵌入式汇编代码可以相互交隔。可以很方便地将 CharacterCount 直接用嵌入式汇编码保存在自由的寄存器，如 CX 或 DX 中；这样处理的话，StringToUpper 的运行速度将更快。

如果用户不了解 Turbo C 在嵌入式汇编语句之间究竟会产生什么代码，那么将 C 代码与嵌入式汇编代码随意混合也会带来危险。剖析混合的 C 代码和嵌入式汇编代码所产生的结果的最好方法是使用 Turbo C 编译器的－S 可选项。例如，用户可以加－S 可选项编译 STRINGUP.C 并检查输出文件 STRINGUP.ASM，以了解 StringToUpper 中的 C 代码与嵌入式汇编码究竟如何协调起来。

STRINGUP.C 生动地阐述了嵌入式汇编所带来的好处。在 StringToUpper 中，仅用了 13 条嵌入式汇编语句就近似地将串处理的速度在相应的 C 代码基础上翻了一番。

1.1.4　嵌入式汇编的限制

对嵌入式汇编的使用，限制极少；一般说来，嵌入式汇编语句被原样地传给 Turbo Assembler。但是，在某些内存和地址操作数的使用上也存在着一些值得注意的限制；另外，对于寄存器的使用规则和嵌入式汇编中使用的自动 C 变量的隐含大小也有一些限制。

1.1.4.1　内存和地址操作数限制

Turbo C 对嵌入式汇编语句所作的唯一修改，就是将内存和内存地址引用，如变量名、转移标号等从 C 语言的表达形式转换成为等价的汇编语言表达形式。这种修改引入了两种限制：嵌入式汇编码中的转移指令只能引用 C 中定义的标号，而嵌入式汇编码中的非转移指令可以引用除标号以外的任何 C 元素。例如，

```
        .
        .
        .
        asm   jz   NoDec;
        asm   dec cx;
NoDec:
        .
        .
        .
```

是可行的，而

```
        .
        .
        .
        asm   jz   NoDec;
        asm   dec cx;
        asm   NoDec:
        .
        .
        .
```

将不能顺利通过汇编。同样，嵌入式汇编码中的转移指令也不能以函数名作为其操作数。非转移的嵌入式汇编指令可以带除 C 标号以外的任何操作数。例如，

```
        .
        .
        .
        asm   BaseValue DB '0';
        .
        .
        .
        asm   mov al,BYTE PTR BaseValue;
        .
        .
```

```
.
```

可以通过汇编，而

```
.
.
.
BaseValue;
    asm    DB '0';
.
.
.
    asm    mov al,BYTE PTR BaseValue;
.
.
.
```

则会出错。

注意，调用语句并不作为转移语句，所以嵌入式汇编中调用语句的合法操作数可以是C函数名和汇编标号，但不能是C标号。如果在嵌入式汇编码中使用了函数名，则函数名前必须加以下划线作为前缀；可参看“下划线”一节。

1.1.4.2　嵌入式汇编中缺少隐含的自动变量大小

在嵌入式汇编语句中，当 Turbo C 用形如[BP－02]的操作数代替对于自动变量的引用时，它并不在修改过的语句中插入 WORD PTR 或 BYTE PTR 等确定大小的算符。这意味着

```
.
.
.
    int    i;
.
.
.
    asm    mov ax, i;
.
.
.
```

将以

```
    movxx,[bp－02]
```

的形式输出到汇编文件中。本例当然不会出现问题，因为 AX 的使用已经告知 TurboAssembler 使用16位的内存引用。更进一步，在嵌入式汇编码中不使用确定操作数大小的算符，可以让用户自由地控制操作数大小。但是，考虑

```
.
.
.
```

```
int   i;
.
.
.
asm   mov i,0
asm   inc i;
.
.
.
```

它将被编译成如下形式：

```
.
.
.
mov   [bp-02],0
inc   [bp-02]
.
.
.
```

这两条指令均没有固定大小的操作数，因而 Turbo Assembler 不能正确地汇编。所以，如果不用寄存器作为源操作数或目的操作数，而希望引用自动变量时，务必要使用确定操作数大小的算符。要让上例正确地运行，必须将它修改成：

```
.
.
.
int   i;
.
.
.
asm   mov   WORD   PTR i,0;
asm   inc   BYTE   PTR i;
.
.
.
```

1.1.4.3 必须保存寄存器

在用户编写的任何嵌入式汇编代码的末尾，下列寄存器中的值必须与进入此嵌入式代码时的值一致：BP、SP、CS、DS 和 SS。违反了这条原则，将频繁地导致程序崩溃和系统重启。嵌入式汇编代码可以自由地改变 AX、BX、CX、DX、SI、DI、ES 及标志的值。

1. 保存调用函数和寄存器变量

Turbo C 要求在调用函数以后不破坏 SI 和 DI 的值，SI 和 DI 可以用作寄存器变量。很幸运，如果在嵌入式汇编代码中使用 SI 和 DI，那么用户不必显式地保存它们。如果 Turbo C 检测出用户在嵌入式汇编代码中使用了 SI 或 DI，它会在函数开始时保存它们的值，再在函数结束时将其恢复——这也是使用嵌入式汇编所带来的另一便利。

2. 抑制内部寄存器变量

因为寄存器变量是用 SI 和 DI 进行保存的，所以给定模块中的寄存器变量和同一模块中使用 SI 和 DI 的嵌入式汇编码之间似乎存在着潜在的冲突。这种冲突同样由 Turbo C 来解决；在嵌入式汇编码中使用 SI 和 DI 将关闭对保存寄存器变量的寄存器的使用。

Turbo C 1.0并不能确保在寄存器变量和嵌入式汇编码之间不发生冲突。如果用户使用的是1.0版本的 Turbo C，那么，要么在嵌入式汇编代码中使用 SI 和 DI 之前显式地保存它们，要么改换成更高版本的编译器。

1.1.5 嵌入式汇编码相对于纯 C 代码的缺点

上面已花了很大篇幅剖析嵌入式汇编码的工作过程和其潜在优点。对许多应用而言，尽管嵌入式汇编是一种很好的特征，但它的确存在着某些缺点。以下将介绍嵌入式汇编的缺点，以便于用户决定何时应在程序中使用嵌入式汇编。

1.1.5.1 降低了可移植性和可维护性

正是直接对8086处理器进行编程的能力使得嵌入式汇编码的效率极高，但同时，也正是这种能力降低了 C 语言的基本属性——可移植性。使用嵌放式汇编时，在不改动代码的情况下，用户最好不要将它移植到另一种处理器或编译器上。

同样，嵌入式汇编码不如 C 代码那样具有清晰明朗的格式，而且通常是非结构化的。相对于 C 代码而言，嵌入式汇编码通常难以阅读和维护。

用户使用嵌入式汇编码时，最好将它分离到一个自包含模块中，并用大量的注释仔细将其结构化。这样就很容易维护代码，需要将程序移植到一个不同环境中时，相对而言也很容易查阅和用 C 语言重写它。

1.1.5.2 降低了编译速度

相对于纯 C 代码而言，编译包含有嵌入式汇编代码的 C 模块的速度要慢得多，主要是因为嵌入式汇编代码要经过两次有效的编译，一次由 Turbo C，一次由 Turbo Assembler。如果因为没有使用可选项－B、－S 或 # progma inline，从而导致 Turbo C 不得不重新启动编译，那么嵌入式汇编的编译时间还会变得更长。幸运的是，同过去相比，编译包含有嵌入式汇编码的模块的低速性现在已不成其为问题，因为 Turbo Assembler 的汇编速度比早期的汇编器要快得多。

1.1.5.3 仅可由 TCC 使用

前面提到过，嵌入式汇编特征为 Turbo C 的命令行版本——TCC. EXE 所仅有，而 Turbo C 的集成开发环境版本——TC. EXE 并不支持嵌入式汇编。

1.1.5.4 损失了优化能力

使用嵌入式汇编时，Turbo C 丧失了对程序代码的一些控制，因为用户可以直接将任何汇编语句插入到任何 C 代码中。作为嵌入式汇编程序设计者，用户在某种程度上必须弥补这一点，避免某些破坏性的动作，如没有保存 DS 寄存器或对内存区进行错误的写入。

另一方面，用嵌入式汇编语言编程时，Turbo C 并不要求遵从所有的内部规则；否则，从嵌入式汇编的使用中得不到任何好处，还不如用 C 进行程序设计并让 Turbo C 产生同样的代码。Turbo C 不对含有嵌入式汇编语句的函数进行优化处理，所以可以相对自由地用嵌入式汇编语言编码。例如，使用嵌入式汇编时，一些转移指令的优化器被关闭；如果嵌入式汇编

码中用到了 SI 和 DI，寄存器变量也被关闭。如果用户可能是为了最大限度地提高代码质量才使用嵌入式汇编，那么这种优化方面的损失就值得考虑了。

如果希望用嵌入式汇编产生快速而紧凑的代码，用户可能会完全用嵌入式汇编书写包含嵌入式汇编码的函数——亦即，在同一函数中不将 C 与嵌入式汇编混合使用。这样，可控制嵌入式汇编函数中的代码，Turbo C 可以控制 C 语言函数中的代码，而用户和 Turbo C 都能自由地产生最少限制的代码。

1.1.5.5 限制了对错误的反跟踪

Turbo C 对嵌入式汇编语句只作少量的错误检查，嵌入式汇编语句中的错误通常由 Turbo Assembler 而不由 Turbo C 进行检测。但遗憾的是，有时很难将 Turbo Assembler 产生的错误信息与原来的 C 源代码相联系，因为输出的错误信息和行号是以 Turbo C 产生的. ASM 文件为基准的，而不是以 C 代码本身为基准。

例如，在编译 TEST. C ——一个包含嵌入式汇编代码的程序的过程中，Turbo Assembler 可能检查出第23行有一个错误大小的操作数；不幸的是，"23"指的是 TEST. ASM 中出错行的行号。TEST. ASM 是由 Turbo C 产生的，有待 Turbo Assembler 汇编的中间汇编文件。用户只能自己去检查 TEST. C 中究竟哪一行会引发错误。

在这种情况下，用户最好先在. ASM 中间文件中定位出错的行。只要 Turbo Assembler 报告汇编错误，Turbo C 就将. ASM 文件保存在用户盘上。. ASM 文件包含有特殊的注解，它标识出每块汇编语句由 C 源文件中的哪一行产生。例如，下面的汇编行

```
;?debug L 15
```

是由 C 源文件的第15行产生的。一旦用户在. ASM 文件中定位了出错行，就可以将注释的行号对应到出错的源 C 文件行上。

1.1.5.6 调试限制

1.5和1.5以下版本的 Turbo C 对包含有嵌入式汇编代码的模块并不能产生源级调试信息（调试时，用户需要这些信息以查阅 C 源代码）。使用嵌入式汇编时，1.5和1.5以下版本的 Turbo C 产生的是不包括调试信息的汇编码。所以，损失了源级调试能力，而只保持了对含有嵌入式汇编码的模块的汇编级调试能力。 当与 Turbo Debugger 一起调试包含有嵌入式汇编代码的模块（当然还有纯 C 模块）时，1.5以上版本的 Turbo C 可以得益于 Turbo Assembler 的特殊性能，支持源级调试过程。

1.1.5.7 用 C 开发而用嵌入式汇编编译最终代码

鉴于前面讨论过的嵌入式汇编的缺点，看来应当尽量减少对嵌入式汇编的使用。其实不然，使用嵌入式汇编的技巧在于将其用于开发过程的末尾。

嵌入式汇编的绝大多数缺点都可以归结于一点：嵌入式汇编会极大地减慢编辑/编译/调试过程。低速度的编译，不能使用集成化的用户接口环境，难以找出编译错误，这三者都意味着包含有嵌入式汇编代码的代码开发比纯 C 代码的开发要慢得多。但是，合理地运用嵌入式汇编代码也可以导致代码质量的动态提高。那么，到底怎样使用嵌入式汇编呢？

答案是简单的。首先，完全用 C 开发每个程序，这样可获益于 TC. EXE 所提供的良好的开发环境。当程序完成了全部功能，并能通过调试和正常运行后，再转换到 TCC. EXE 并将程序中重要的部分改换成嵌入式汇编代码。这种方法可以让用户高效率地开发和调试整个程序，再在需要改进程序性能时分离出并强化选出的部分代码。

1.2 从 Turbo C 中调用 Turbo Assembler 函数

将 C 语言和汇编语言混合使用的传统方式是，先完全用 C 语言或汇编语言写出单独的模块，编译 C 语言模块，并汇编出汇编语言模块，然后将分别编译过的模块链接到一起。按这种方式，Turbo C 模块和 Turbo Assembler 模块可以很方便地进行链接。图1.3描述了整个过程。

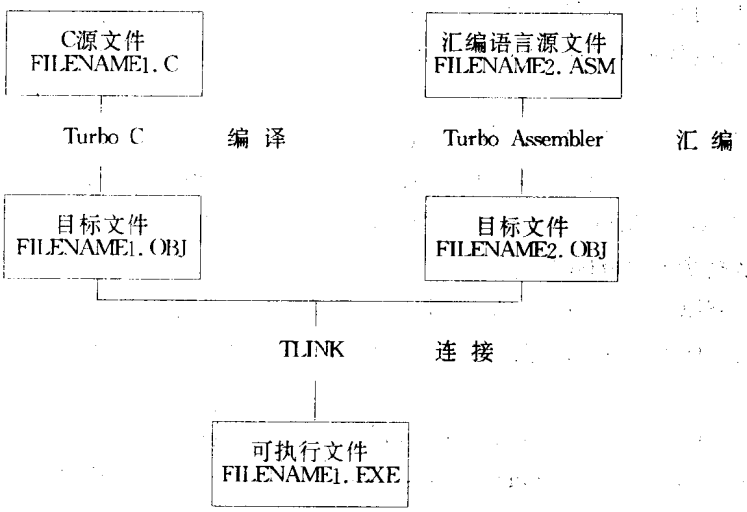


图1.3 用 Turbo C、Turbo Assembler 和 TLINK 编译、汇编和连接

可执行文件由 C 源文件与汇编源文件混合产生。用户可以采用以下命令：

TCC filenam1 filenam2.asm

该命令指示 Turbo C 首先将 FILENAM1.C 编译成 FILENAM.1.OBJ，再激活 TurboAssembler，将 FILENAM2.ASM 汇编成 FILENAM2.OBJ，最后激活 TLINK，链接 FILENAM1.OBJ 与 FILENAM2.OBJ 并产生 FILENAM1.EXE。

分别编译对于可确定大小的汇编代码程序极为有用，因为它可以让用户使用 Turbo Assembler 的全部功能，在纯汇编语言环境中进行汇编语言程序设计，不需要 asm 关键字、额外的编译时间以及嵌入式汇编中与 C 相关的辅助工作。

分别编译也有一定的代价：汇编语言程序设计者必须清楚 C 代码与汇编语言代码的详细接口。Turbo C 可以自处理段说明、参数传递、对 C 变量的引用、保留寄存器变量以及嵌入式汇编等方面，而分别编译的汇编函数则必须显式地完成这些工作，甚至还要完成更多的工作。

Turbo C 与 Turbo Assembler 的接口中，有两个重要的方面值得注意。首先，C 代码与汇编码的各个不同部分必须恰当地链接到一起，其它代码中必须能够使用代码每部分的函数和变量。其次，汇编码必须能恰当地处理 C 风格的函数调用，包括访问传递过来的参数、返回值以及遵从 C 函数所要求的寄存器保存规则。

以下首先开始分析 Turbo C 代码与 Turbo Assembler 代码的链接规则。

1.2.1 Turbo C 与 Turbo Assembler 的接口机制

为了将 Turbo C 与 Turbo Assembler 模块链接到一起，必须完成以下三项工作：Turbo

Assembler 模块必须使用与 Turbo C 兼容的段命名方案，Turbo C 与 Turbo Assembler 模块必须以 Turbo C 可接受的方式共享适当的函数和变量名，必须用 TLINK 将这些模块链接成可执行程序。这时并没有指出 Turbo Assembler 实际上要完成什么工作；而只注意到如何创建一种模式，以便可以按此模式编写与 C 兼容的 Turbo Assembler 函数。

1.2.1.1 内存模式和段

对于一个给定的可以由 C 调用的汇编语言函数，该函数必须采用与 C 程序一致的内存模式，同时必须使用与 C 兼容的代码段。同样，为了让 C 代码能够访问在汇编语言模块中定义的数据（或让汇编代码能够访问 C 模块中定义的数据），汇编语言代码务必遵从 C 语言的数据段命名约定。

内存模式和段处理在汇编语言中实现起来很复杂。幸运的是，通过对简化的段伪指令的使用，在实现与 Turbo C 兼容的内存模式和段这两方面，Turbo Assembler 事实上已为用户完成了全部工作（参看 Turbo 汇编中有关“标准的段伪指令”的内容）。

1. 简化的段伪指令与 Turbo C

伪指令 DOSSEG 指示 Turbo Assembler 根据 Intel 公司的段排序约定对段进行排序，Turbo C 也遵从同样的约定（其它许多通用的语言产品，包括 Microsoft 公司的产品，也遵从这种约定）。

伪指令.MODEL 告诉 Turbo Assembler，用简化的段伪指令创建的段应当与选定的内存模式（tiny、small、compact、medium、large 或 huge）兼容，并控制用 PROC 伪指令创建的过程的隐含类型。由.MODEL 伪指令定义的内存模式与具有同样命名的 Turbo C 模式是相互兼容的。

最后，.CODE、.DATA、.DATA?、.FARDATA、.FARDATA?及.CONST 等简化的段伪指令产生的段与 Turbo C 兼容。

例如，考虑下面的 Turbo Assembler 模块，该模块命名为 DOTOTAL.ASM：

```
; select Inter—convertion segment ordering
    .MODEL small                    ;select small model (near code and date)
    .DATA                           ;TC—compatible initialized data segment
EXTRN   Repetitions: WORD           ;externally defined
    PUBLIC   StartingValue          ;available to other modules
 StartinValue  DW  0
    .DATA?                          ;TC—compatible uniniialized data segment
RunningTotal  Dw  ?
    .CODE                           ;TC—compatible code segment
    PUBLIC   DoTotal
 DoTotal  Proc                      ;function (near—callable in small mode)
          mov  cx,  [ Repetitions]  ; # of counts to do
          mov  ax,  [ StartingValue]
          mov  [RunningTotal],  ax  ;set initial value
TotalLoop:
          inc  [RunningTotal]       ;RunningTotal++
          loop  TotalLoop
```

```
        mov   ax,[RunningTotal]        ;return final total
        ret
DoTotal ENDP
        END
```

在 DoTotal 过程中，许多标号的前面有下划线( )前缀，这仅仅是为了满足 Turbo C 的需要。"下划线"一节将详细介绍这方面的内容。

可以用语句

```
DoTotal();
```

在具有 Small 模式的 Turbo C 程序中调用汇编语言过程 DoTotal。

注意，过程 DoTotal 希望在程序的其它部分定义外部变量 Repetitions。同样，变量 StartingValue 被定义成公有变量，以便程序的其余部分可以对其进行访问。

下面的 Turbo C 模块 SHOWTOT. C 将访问 DOTOTAL. ASM 中的公有数据并将外部数据提供给 DOTOTAL. ASM：

```
extern int StartingValue;
extern int DoTotal(void);
int Repetitions;
main()
{
    int i;
    Repetitions = 10 ;
    StartingValue = 2;
    printf("%d\n",doTotal());
}
```

为了根据 SHOWTOT. C 和 DOTOTAL. ASM 创建可执行程序 SHOWTOT. EXE，可输入以下命令行：

```
tcc showtot dototal. asm
```

如果希望将 Dototal 链接到具有 Compact 模式的 C 程序中，只须简单地将. MODEL 伪指令改为. MODEL COMPACT 即可。需要在 DOTOTAL. ASM 中使用具有 far 类型的段时，用户可以运用. FARDATA 伪指令。

简单地说，用简化的段伪指令产生正确的段排序、内存模式和段名字以便与 Turbo C 链接是一种很简便的方法。

2. 过时风格的段伪指令与 Turbo C

简言之，使用过时风格的段伪指令设计 Turbo Assembler 代码与 Turbo C 代码的接口只是在自找麻烦。例如，如果用过时风格的段伪指令代替 DOTOTAL. ASM 中简化的段伪指令，可以得到

```
DGROUP  GROUP   DATA, BSS
 DATA   SEGMENT WORD PUBLIC 'DATA'
     EXTRN   Repetitions: WORD      ;externally defined
     PUBLIC  StartingValue          ;avaliable to other moudles
 StartingValue DW  0
 DATA   ENDS
```

```
 BSS   SEGMENT   WORD PUBLIC 'BSS'
RunningTotal   DW ?
 BSS   ENDS
 TEXT   SEGMETN BYTE PUBLIC 'CODE'
      ASSUME cs: TEXT,ds:DGROUP,ss:DGROUP
      PUBLIC   DoTotal
 DoTotal   PROC                          ;function (near-callable
                                         ; in small model)
          mov   cx,  [ Repetitions]      ;# of counts to do
          mov   as,  [ StartingValue]
          mov   [RunningTotal],  ax      ; set initial value
TotalLoop:
          inc   [RunningTotal]           ;RunningTotal++
          loop   TotalLoop
          mov   ax,  .[RunningTotal]     ;Retrun final   total
          ret
 DoTcal   ENDP
 TEXT     ENDS
          END
```

使用过时风格段伪指令的 DOTOTAL.ASM 版本不仅很冗长，而且难以阅读，难以修改成与不同的 C 内存模式相匹配的形式。与 Turbo C 接口时，使用过时风格的段伪指令通常没有任何好处。如果用户仍然希望采用过时风格的段伪指令设计与 Turbo C 的接口，那么，用户必须为 C 代码所使用的内存模式标识出正确的段。有关 Turbo C 中段的用法，请参考《Turbo C 用户手册》。

为了与给定的 Turbo C 程序链接，确定恰当的过时风格的段伪指令的最简单的方法，就是用 -S 可选项以期望的内存模式编译 Turbo C 程序的主模块，-S 可选项将指示 Turbo C 产生 C 代码的汇编版本。在 C 代码中，用户可以看到 Turbo C 使用的所有过时风格的段伪指令；可将这些伪指令拷贝到汇编语言代码中。例如，如果打入命令

```
tcc -S showtot.c
```

则产生 SHOWTOT.ASM 文件，其内容为：

```
        ifndef   ??version
?debug   macro
                 ENDM
                 ENDIF
                 NAME   showtot
 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  E91481D5100973686F77746F742E63
BSS           ENDS
TEXT  SEGMENT BYTE PUBLIC  'CODE'
;       ?debug  L  3
main          PROC  NEAR
;       ?debug L 6
        mov       WORD PTR DGROUP:  Repetitions,  10
;       ?debug L  7
        mov       WORD PTR DGROUP:  StartingValue,  2
;       ?debug L  8
        call      NEAR PTR  DoTotal
        push      ax
        mov       ax,  offset DGROUP:  s@
        push      ax
        call      NEAR PTR  printf
        pop       cx
        pop       cx
@1:
;       ?debug L 9
        ret
main          ENDP
TEXT          ENDS
BSS SEMENT WORD PUBLIC 'BSS'
Repetitions LABEL WORD
        DB  2  dup(?)
        ?debug  C  E9
BSS ENDS
DATA SEGMETN WROD PUBLIC 'DATA'
s@ LABLE BYTE
        DB        37
        DB        100
        DB        10
        DB        0
DATA  ENDS
        EXTRN     StartingValue:WORD
TEXT          SEGMENT BYTE PUBLIC 'CODE'
        EXTRN      DoTotal:NEAR
        EXTRN      printf:NEAR
TEXT          ENDS
```

```
PUBLIC  Repetitions
PUBLIC  main
END
```

其中，DATA(初始化的数据段)、TEXT(代码段)、BSS(非初始化的数据段)的伪指令以及 GROUP 和 ASSUME 伪指令都处于待汇编的形式，所以用户可以直接使用它们。《Turbo Assembler 参考手册》详细介绍了段伪指令。

3. 段缺省：何时需要装载段？

在某些情况下，为了访问数据，可以由 C 调用的汇编函数可能不得不装载 DS 和/或 ES。在从 C 中调用汇编函数时，了解寄存器设置之间的关系也是很有用的，因为汇编代码有时能够获益于段寄存器之间的等价关系。下面花少量的篇幅介绍从 Turbo C 中调用汇编函数时的段寄存器设置情况、段寄存器间的关系以及汇编函数可能需要装载一到多个段寄存器的情况。

当从 Turbo C 进入汇编函数时，根据使用的内存模式，CS 和 DS 寄存器有如下设置(SS 常被用作堆栈段，ES 常被用作附加段寄存器)。

filename 指汇编模块的名字，calling-filename 指调用汇编模块的模块名。

在 tiny 模式下，TEXT 与 DGROUP 是相同的，所以在进入函数时的 CS 与 DS 相等。在 tiny、small 或 medium 模式下，刚进入函数时，SS 也与 DS 相等。

那么，在可由 C 调用的汇编语言函数中，到底何时有必要装载段寄存器呢？显然，用户不应(也不需要)直接装载 CS 或 SS 寄存器。在远调用、远转移或远返回时，CS 根据需要被自动设置，而不能用其它方式设置它。SS 通常指向堆栈段，在程序中不应直接改变之(除非用户编写的是转换堆栈的代码，此时用户最好知道这样做的后果)。

表1.1　Turbo C 调用汇编时的寄存器设置

模型	CS	DS
微模式	TEXT	DGROUP
小模式	TEXT	DGROUP
紧缩模式	TEXT	DGROUP
中模式	filename　TEXT	DGROUP
大模式	filename　TEXT	DGROUP
巨模式	filename　TEXT	DGROUP

用户通常可以根据自己的需要使用 ES。可以使 ES 指向远数据，也可以将串指令的目的段装入 ES。

现在还剩下 DS 寄存器。在除 huge 模式以外的 Turbo C 的其它模式中，刚进入函数时，DS 指向静态数据段(DGROUP)，用户通常不必改变它。用户通常可以用 ES 访问远数据，但也可能会发现，暂时用 DS 指向要访问的远数据较为理想，这样可以省去代码中的许多段指令。例如，用户可用下列两种方式之一访问一个具有 far(远)类型的段：

```
        .FARDATA
Counter  DW  0
        .
        .
        .
        .CODE
        PUBLIC  AsmFunction
AsmFunction  PROC
        .
        .
        .
        mov ax,@fardata
        mov ex,ax              ;point ES to far data segment
        int  es:[Counter]      ;increment counter variable
        .
        .
        .
AsmFunction  ENDP
        .
        .
        .
```

或

```
        .
        .FARDATA
Counter  DW  0
        .
        .
        .
        .CODE
        PUBLIC  AsmFunction
AsmFunction  PROC
        .
        .
        .
        ASSUME ds:@fardata
        mov ax,  @fardata
        mov ex,  ax            ;point ES to far data segment
        int  [Counter]         ;increment counter variable
        ASSUME  ds:@data
        mov ax,@data
        mov ds,ax              ;point DS back to DGROUP
        .
        .
```

```
    .
AsmFunction  ENDP
    .
    .
    .
```

第二种方法有一个好处，即每次访问远数据段的内存时不必加上 ES:前缀。如果用户的确装载了 DS 并使之指向 far 类型的段，象上例一样，务必在使之恢复之后才能访问 DGROUP 中的任何变量。即便在给定的汇编函数中不访问 DGROUP 中的变量，在退出之前也必须恢复 DS，因为 Turbo C 总是假定函数不会改变 DS 的值。

在具有 huge 模式的可由 C 调用的汇编函数中处理 DS 时，情况有些不同。在 huge 模式下，Turbo C 根本就不使用 DGROUP。相反，每个模块都有自己的数据段，相对于程序中所有其它模块而言，它是远类型的段；不存在共享的近数据段。在进入一个具有 huge 模式的函数中时，应该设置 DS 并使之指向该模块的远类型的段，在函数的其余部分则不再改变 DS 的值。例如，

```
    .
    .
    .FARDATA
Counter  DW  0
    .
    .
    .
    .CODE
    PUBLIC  AsmFunction
AsmFunction PROC
    push    ds
    mov     ax, @fardata
    mov     ds, ax
    .
    .
    .
    pop     ds
    ret
AsmFunction  ENDP
    .
    .
    .
```

注意，在进入 AsmFunction 时用 PUSH 指令保存 DS 的原状态，在退出之前用 POP 指令恢复 DS 的值；甚至在 huge 模式下，Turbo C 也要求所有函数都保存 DS。

1.2.1.2 公共量和外部量

Turbo Assembler 代码可以调用 C 函数并引用 C 中的外部变量，Turbo C 代码也同样可以调用 Turbo Assembler 中的公共函数，并引用 Turbo Assembler 中的公共变量。正如上一

节所描述的一样，一旦在 Turbo Assembler 中创建了与 Turbo C 兼容的段，那么，要让 Turbo C 和 Turbo Assembler 共享函数和变量，只需遵从以下几条简单的规则。

1. 下划线

一般情况下，Turbo C 希望所有的外部标号均以下划线( )开头。在 C 代码中，TurboC 自动给所有的函数名及外部变量名加上下划线前缀，所以，用户只需在汇编码中加上下划线即可。用户必须确保在汇编语言中对 Turbo C 函数和变量的所有引用均加上下划线，同时必须确保由 Turbo C 引用的所有公共的汇编语言函数和变量都以下划线作为开头。

例如，下面的 C 代码：

```
extern int ToggleFlag();
int flag;
main()
{
    ToggleFlag();
}
```

可以与下面的汇编程序正确地链接：

```
.MODEL  small
.DATA
EXTRN     Flag:  WORD
.CODE
PUBLIC    ToggleFlag
ToggleFlag  PROC
   cmp  [ Flag], 0          ;is the flag reset?
   jz   SetFlag             ;yes, set it
   mov  [ Flag],  0         ; no, reset it
   jmp  short EndToggleFlag ;done
SetFlag:
   mov  [ Flag],  1         ;set Flag
EndToggle:
   ret
ToggleFlag  ENDP
END
```

注意，如 SetFlag 等不被 C 代码调用，标号不必具有前导下划线。

顺便指出，可以用命令行可选项－u－告知 Turbo C 不使用下划线。尽管该选择项似乎很有吸引力，但 Turbo C 的实用库函数是在启动下划线的情况下编译的。所以用户不得不从 Borland 公司购买实用库源程序，并在关闭下划线的情况下重新编译该库，这样才能使用－u－选择项("Pascal 调用约定"中有关于－P 选择项的详细介绍。使用－P 可选项时将关闭对下划线的使用，并对大小写同样对待)。

2. 大小写字母的意义

通常情况下，在处理符号名时，Turbo Assembler 对字母的大小写并不敏感，所以并不区别对待大写字母和小写字母。因为 C 语言区别对待大小写，所以最好也使 TurboAssembler 对大小写符号区别对待，至少对于汇编模块和 C 模块所共享的那些符号而言应该如此。/ml

和/mx 可以做到这一点。

命令行开关/ml 使得 Turbo Assembler 对所有符号均按大小写区别对待。命令行开关/mx 使得 Turbo Assembler 只对公共符号(PUBLIC)、外部符号(EXTRN)、全程符号(GLOBAL)和公用符号(COMM)按大小写区别对待。

3. 标号类型

尽管汇编语言程序可以自由地将任何变量作为任意大小的数据(8位、16位、32位等等)来进行访问,但通常情况下应按变量的本来大小去访问它。例如,如果用户试图将字值写到字节变量中去,通常就会引起错误:

```
.
.
.
SmallCount DB 0
.
.
.
mov WORD PTR [SmallCount],0ffffh
.
.
.
```

所以,声明外部 C 变量的汇编语言语句 EXTRN 为这些变量指定合适的大小是很重要的,因为 Turbo Assembler 只能根据用户的声明来决定究竟该以什么大小访问 C 变量。假定在 C 程序中包含有语句

```
char c;
```

汇编代码

```
.
.
.
EXTRN c:WORD
.
.
.
inc [c]
.
.
.
```

将会引起问题,因为每当汇编代码将 C 递增256次时,c 就会发生翻转。而 c 被错误地声明为字变量,所以地址 OFFSET c+1处的字节也会被错误地递增,从而引起不可预料的后果。

C 语言和汇编语言的数据类型对比如1.2表。

4. 远类型的外部量必须在任何段之外

如果使用简化的段伪指令,远段中标号的 EXTRN 声明就不能放在任何段内,因为 Turbo Assembler 认为在给定段中声明的标号都与该段相关。这带来了某些缺点;Tur-

boAssembler 不能检查是否可以寻址在任何段外由 EXTRN 声明的标号，所以既不能产生所需要的段前缀，在没有装入正确的段时，如果用户试图访问这些变量，Turbo Assembler 也不能对用户给出提示。Turbo Assembler 仍能将对这些外部符号的引用汇编成正确代码，但却不能提供正常程度的段可寻址性检查。

表1.2　C 语言汇编语言的数据类型对比

C 数据类型	汇编语言数据类型
unsigned char	byte
char	byte
enum	word
unsigned short	word
short	word
unsigned int	word
int	word
unsigned long	dword
long	dword
fload	dword
double	qword
long double	tbyte
near *	word
far *	dword

如果用户愿意(建议用户最好取消这种念头)，也可以采用过时风格的段伪指令显式地声明每个外部符号所在的段，然后将声明这些符号的 EXTRN 伪指令放入段声明中。这样做也有一点好处；在访问远数据时，如果用户不能确认是否已装载了正确的段，可以方便地只将远符号的 EXTRN 声明放在所有段之外。例如，假定 FILE1. ASM 中包含

```
.
.
.
.FARDATA
File1Variable DB 0
.
.
.
```

而 FILE2. ASM 中包含

```
.
.
.
.DATA
  EXTRN   File1Variable: BYTE
  .CODE
```

```
Start   PROC
        mov     ax,  SEG File1Variable
        mov     dx,  ax
        .
        .
        .
```

如果将 FILE1. ASM 链接到 FILE2. ASM 上,SEG File1Variable 将不能返回正确的段。在 FILE2. ASM 中,EXTRN 伪指令被放在 DATA 伪指令所属的范围内,所以,Turbo Assembler 认为 File1Variable 是在 FILE2. ASM 的近 DATA 段中,而不是在 FARDATA 段中。

在下面的 FILE2. ASM 版本中,SEG File1Variable 可以返回正确的段:

```
        .
        .
        .
        .DATA
@curseg ENDS
        EXTRN File1Variable:BYTE
        .CODE
Start   PROC
        mov     ax,SEG File1Variable
        mov     ds,ax
        .
        .
        .
```

这里的技巧在于用@curseg ENDS 伪指令结束了. DATA 段,所以在将 File1Variable 声明成外部量时,所有的段伪指令均没有生效。

1.2.1.3 链接器命令行

将 Turbo C 模块与 Turbo Assembler 模块相链接的最简单的方法是打入一行 Turbo C 命令,并让 Turbo C 完成全部工作。打入恰当的命令时,Turbo C 将编译 C 代码,激活 Turbo Assembler 完成汇编过程,激活 TLINK 将目标模块链接成可执行文件。例如,假定用户程序由 C 语言文件 MAIN. C、STAT. C 和汇编语言文件 SUMM. ASM、DISPLAY. ASM 组成。命令行

```
tcc main stat summ. asm display. asm
```

将编译 MAIN. C 和 STAT. C,汇编 SUMM. ASM 和 DISPLAY. ASM,把生成的四个目标文件以及 C 的启动码和所需的任何库函数链接到一起,成为 MAIN. EXE 文件。打入汇编文件名时,用户务必记住要打入扩展名. ASM。

如果用户使用单独模式的 TLINK,那么 Turbo Assembler 生成的目标文件是标准的目标模块,可以与 C 语言目标模块一样对待它。

1.2.2 Turbo Assembler 与 Turbo C 的交互性

前面已介绍了如何创建并链接与 C 兼容的汇编语言模块,现在开始讨论用户可将哪类代码放入可由 C 调用的汇编语言函数中。这里分三个方面加以讨论:接受传递过来的参数;

使用寄存器;值返回到调用码中。

1.2.2.1 参数传递

Turbo C 通过堆栈将参数传递给函数。调用函数之前,Turbo C 先将要传给函数的参数压入堆栈,最先压入最右边的参数,最后压入最左边的参数。C 函数调用

```
.
.
.
Test (i,j,1);
.
.
.
```

被汇编成

```
mov  ax,1
push ax
push WORD PTR DGROUP: j
push WORD PTR DGROUP: i
call NEAR PTR  Test
add  sp,6
```

用户可以很清楚地看到,先压入最右边的参数1,再压入 j,最后压入 i。

从函数返回时,先前压入堆栈的参数仍然保留在栈中,但这些参数已没有任何用途。所以,在每次调用函数之后,Turbo C 立即调整堆栈指针,使之指向压入参数前所指的位置,这样就放弃了栈中的参数。在上面的例子中,3个双字节的参数共占6个字节的堆栈空间,所以在调用 Test 之后,Turbo C 给栈指针加6,以删除这些参数。这里,很重要的一点就是,根据 C 调用约定,堆栈中的参数由调用码负责删除(参看"pascal 调用约定"部分的内容)。

汇编语言函数可以相对于 BP 寄存器访问传入堆栈中的参数。例如,假定上例中的函数 Test 具有以下汇编语言形式:

```
    .MODEL small
    .CODE
    PUBLIC   Test
Test        PROC
    push    bp
    mov     bp,  sp
    mov     ax,  [bp+4]     ; get parameter 1
    add     ax,  [bp+6]     ; add parameter 2 to parameter 1
    sub     ax,  [bp+8]     ; subtract parameter 3 form sum
    pop     bp
    ret
Test        ENDP
END
```

可见,函数 Test 可相对于 BP 从栈中取出由 C 代码传入的参数(请记住,BP 寻址堆栈段)。但究竟怎样知道在何处相对于 BP 寻找参数呢?

图1.4绘出了执行 Test 中第一条指令之前堆栈的示意图：

```
i=25;
j=4;
Test(i,j,1);
```

传给 Test 的参数相对于 SP 的位置是确定的。函数调用时，返回地址也被压入堆栈，参数从栈中比返回地址的位置高两个字节的地址处开始存放。将 SP 装入 BP 之后，就可以相对于 BP 访问参数了。但首先必须保存 BP，因为调用码要求在调用之后 BP 保持不变。压入 BP 将修改堆栈中所有数据的偏移量。图1.5绘出了在执行下几行代码之后的堆栈情况：

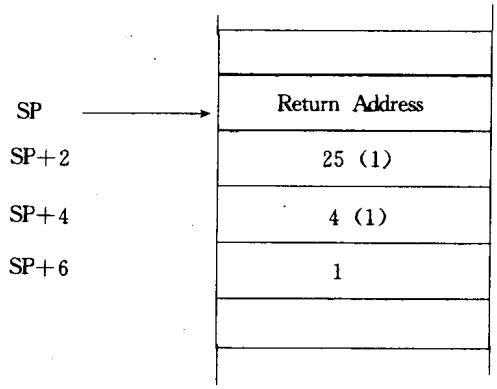


图1.4 执行 TEST 的第一条指令前的堆栈状态

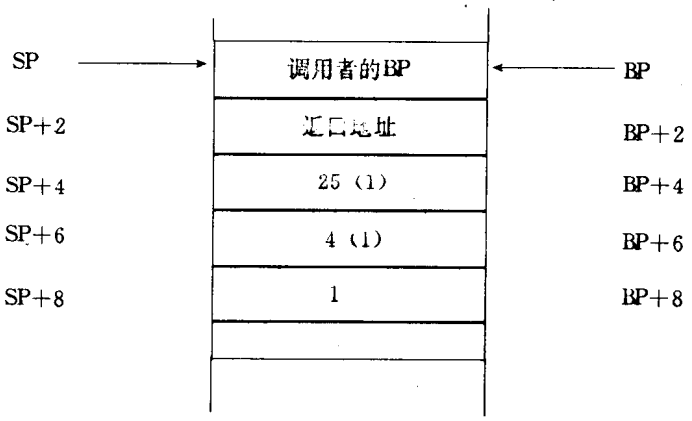


图1.5 PUSH 和 MOV 之后的堆栈状态

.
.
.

push bp

```
mov bp,SP
.
.
.
```

以上是标准的C堆栈模型及函数参数和自动变量在堆栈中的组织情况。正如用户所见到的一样，不管C程序有多少参数，最左边的参数总是保存在被压入的返回地址之上的相邻位置，右边相邻的参数保存在最左边参数之上，如此类推。只要知道了传入参数的顺序和类型，也就知道了在栈中的何处寻找它们。

可以将自动变量空间预留在从SP中减去一定数目字节的位置。例如，在Test函数的前面加上以下几行语句就可以为100个字节的自动数组预留空间，如图1.6所示。

```
.
.
.
push   bp
mov   bp,SP
sub   sp,100
.
.
.
```

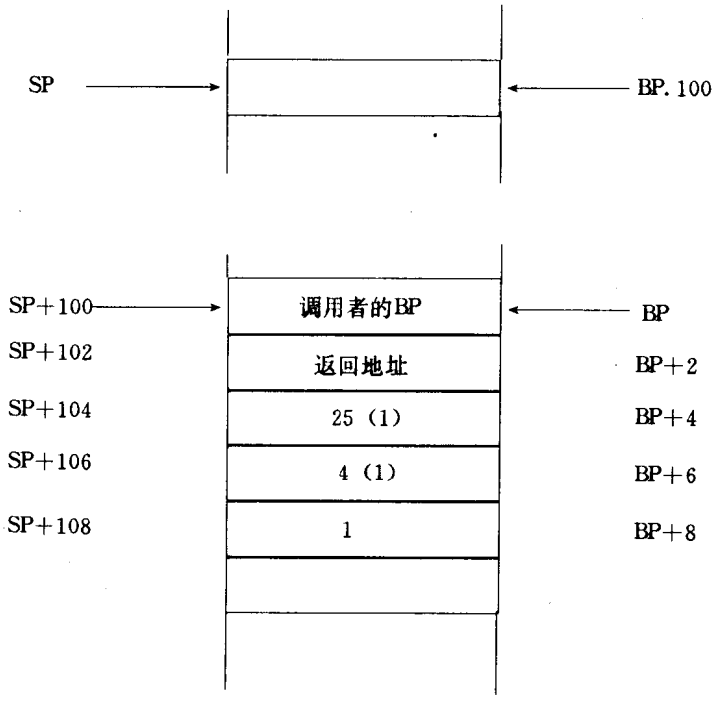


图1.6　PUSH、MOV 和 SUB 之后的堆栈状态

因为保存自动变量的堆栈部分在比BP低的地址处，所以寻址自动变量时要从BP中减去一定的偏移量。例如，

mov BYTE PTR [BP－100],0

将预保留的100个字节的数组的首字节设置为0。另一方面，寻址传入的参数时通常要在 BP 中加上一定的偏移量。

如果用户愿意，用户可以象前面介绍的那样为自动变量分配空间，但 Turbo Assembler 提供了一条特殊的 LOCAL 伪指令，可以使自动变量的空间分配和命名更为简便。在过程中碰到 LOCAL 时，Turbo Assembler 就认为是在定义该过程的自动变量。例如，

LOCAL LocalArray: BYTE: 100, LocalCount: WORD = AUTO　SIZE

定义了自动变量 LocalArray 和 LocalCount。LocalArray 实际上是一个与[BP－100]等价的标号，LocalCount 实际上是一个与[BP－102]等价的标号，但可将它们用作变量名而无需知道它们的值。AUTO　SIZE 是自动存储空间所需的总的字节数；为了找到自动变量的存储空间，必须从 SP 中减去该值。

用户可以从下面的例子中看出 LOCAL 伪指令的用法：

```
        .
        .
        .
  TestSub   PROC
    LOCAL LocalArray:BYTE:100,LocalCount: WORD= AUTO  SIZE
    push   bp                  ;preserve caller's  stack frame pointer
    mov    bp,sp               ;set up our own stack frame pinter
    sub    sp,AUTO  SIZE       ;allocate room for automatic variables
    mov    [LocalCount],10     ;set local count variale to 10
                               ;(LocalCount is actually [BP－102])
        .
        .
        .
    mov    dx,[LocalCount]     ;get count from local variable
    mov    al,'A'              ;we'll fill with character "A"
    lea    bx,[LocalArray]     ;point to local array
                               ;(LocalArray is actually [BP－100])
FillLoop:
    mov    [bx],  al           ;fill next byte
    inc    bx                  ;point to following byte
    loop   FillLoop            ;do next byte ,  if any
    mov    sp,bp               ;deallocate storage for automatic
                               ;variables (add sp,AUTO  SIZE would
                               ;also have wroked)
    pop    bp                  ;restore caller's stack frame pointer
    ret
  TestSub ENDP
        .
        .
        .
```

注意，在该例子中，给定自动变量定义后的第一个域是变量的数据类型：BYTE、WORD、DWORD、NEAR 等等。给定自动变量定义后的第二个域是为该变量预留的该变量类型的元素数目。该域是可选的，使用时定义一个自动数组；省略该域时，则只为指定类型预留一个元素。所以，Local Array 是由100个字节大小的元素组成的数组，而 Local Count 只有一个字大小的元素。

同样值得注意的是，上例中的 LOCAL 行以＝AUTO　SIZE 结尾。该域以等于号作为开始，是可选的；如果选用时，紧跟等于号后的标号被设置成自动存储空间所占的字节数。用户必须使用此标号分配或重新分配自动变量的存储空间，因为伪指令 LOCAL 只产生标号，而实际上并不产生任何代码或数据存储单元。换言之，LOCAL 不为自动变量分配空间，而仅仅产生标号，用户可以用这些标号分配存储空间或访问自动变量。

LOCAL 伪指令有一个极为方便的特点，表示自动变量的标号和表示自动变量空间大小的标号都被限制在用户在其中声明这些标号的过程内，所以，在其它过程中可以重复使用这些自动变量名。

正如用户看到的一样，LOCAL 使得对于自动变量的定义和使用变得简单得多。注意，在宏中使用 LOCAL 伪指令时，其含义完全不同，这部分的内容参见《Turbo 汇编用户手册》（参看《Turbo Assembler 参考手册》第三章，其中有关于 LOCAL 伪指令形式的附加信息）。

顺便指出，Turbo C 仅以此处描述的方式处理堆栈模型。用户可以用可选项－S 编译几个 Turbo C 模块，并查看 Turbo C 产生的汇编代码，可以看出 Turbo C 是如何创建和使用堆栈的。

尽管到此为止，用户所看到的参数传递方式似乎不错，但仍存在更复杂的方面。首先，相对于 BP 以常偏移量访问参数的做法并不理想；不仅仅因为它容易出错，而且，如果增加另外的参数，函数中所有其它的堆栈偏移量都必须改变。例如，假定修改 Test，使之能接受四个参数：

```
Test(flag,i,j,1);
```

这时，i 的偏移量变为6，而不是4；j 的偏移量变为8，而不是6；如此类推。用户可以用等价符代替参数的偏移量：

```
.
.
.
Flag EQU 4
AddParm1 EQU 6
AddParm2 EQU 8
SubParm1 EQU 10

mov  ax,[bp+Addparm1]
add  ax,[bp+Addparm2]
sub  ax,[bp+SubParm1]
.
.
.
```

但对偏移量的计算和维护仍很困难。更为严重的问题是：在远代码模式中，被压入堆栈的返回地址的大小又增加了两个字节；在远代码和远数据模式下，传入的代码指针和数据指针的大小也同样增加两个字节。看来，编写一个在任意地址模式下都能方便合理地访问堆栈模型的函数极为困难。

用不着担心。Turbo Assembler 为用户提供了一条 ARG 伪指令，在汇编语言函数中，用它可以方便地处理传入的参数。

ARG 伪指令可以自动地为用户指定的变量产生正确的堆栈偏移量。例如，

```
arg Fill Array:WORD,Count:WORD,FillValue:BYTE
```

指定了三个参数：Fill Array，字大小的参数；Count，字大小的参数；Fill Value，字节大小的参数。事实上，ARG 将标号 Fill Array 设置成[BP+4]（假定样例代码驻留在近过程中），将标号 Count 设置成[BP+6]，将标号 FillValue 设置成[BP+8]。因为不知道由 ARG 定义的标号所具有的值就可以使用之，所以，使用 ARG 还是比较有价值的。

例如，假定在 C 中调用函数 FillSub。C 函数如下：

```
main()
{
    #define ARRAY  LENGTH 100
    char TestArray[ARRAY  LENGTH];
    FillSub(TestArray,ARRAY  LENGTH,'*');
}
```

在函数 FillSub 中，用户可以使用 ARG 伪指令处理参数：

```
FillSub  PROC NEAR
         ARG FillArray: WORD Count : WORD,FillValue :BYTE
         push  bp              ;preserve caller's stack frame
         mov   bp,sp           ;set out own stack frame
         mov   bx,[FillArray]  ;get pointer to array to fill
         mov   cx,[Count]      ; get length to fill
         mov   al,[FillValue]  ; get value  to fill with
FillLoop:
         mov   [bx],al          ;fill a character
         inc    bx             ;point to next character
         loop  FillLoop        ;do next character
         pop   bp              ;restore caller's  stack frame
         ret
FillSub  ENDP
```

用 ARG 伪指令处理传入的参数，所有要做的工作被示于上例中。ARG 伪指令可以自动处理近返回和远返回时的不同数据大小。使用 ARG 伪指令所带来的另一种方便之处在于，使用局部标号前缀（参看《Turbo Assembler 参考手册》中的"LOCALS"部分）作为声明时，用 ARG 伪指令定义的标号被限制在用户在其中使用这些标号的过程内。所以，用户不用担心不同过程中的参数名发生冲突。

有关 ARG 伪指令更详细的内容，请翻阅《Turbo Assembler 参考手册》第三章。

1.2.2.2 保存寄存器

就 Turbo C 而言，只要保存了寄存器 BP、SP、CS、DS 和 SS，那么可由 C 调用的汇编语言函数可以完成被要求的任何工作。尽管在汇编语言函数中可以改变这些寄存器，但当返回到调用码时，它们的值必须与调用汇编函数时的值一致。可以随意改变 AX、BX、CX、DX、ES 和标志的值。

SI 和 DI 是特殊情况，因为 Turbo C 将其用作寄存器变量。如果在调用汇编语言函数的 C 模块中启动了寄存器变量，就必须保存 SI 和 DI；但若没有启动寄存器变量，就不必保存 SI 和 DI。一种好的经验，就是在由 C 调用的汇编语言函数中总是保存 SI 和 DI，而不管是否启动了寄存器变量。因为用户不知道什么时候可能需要将一个给定的汇编语言模块链接到另一个 C 模块上；或者忘记了需要修改汇编语言模块，而在启动寄存器变量的情况下重新编译了 C 代码。

1.2.2.3 返回值

正如 C 函数一样，可由 C 调用的汇编语言函数也可以返回值。函数值的返回形式表1.3。

表1.3 函数值的返回形式

返回值类型	返回值位置
unsigned char	AX
char	AX
enum	AX
unsigned short	AX
short	AX
unsigned int	AX
int	AX
unsigned long	DX:AX
long	DX:AX
float	8087栈顶(TOS)寄存器(ST(0))
double	8087栈顶(TOS)寄存器(ST)0))
long double	8087栈顶(TOS)寄存器(ST(0))
near *	AX
far *	DX:AX

通常情况下，8位或16位值返回到 AX 中，32位值返回到 DX:AX 中，其中高16位值在 DX 中。浮点值返回到 ST(0)，即8087的栈顶(TOS)寄存器中，使用浮点数仿真器时，则返回到8087仿真器的 TOS 寄存器中。

结构的返回要稍复杂一些。1或2个字节长的结构返回到 AX 中，4字节长的结构返回到 DS:AX 中。3字节的结构或多于4字节的结构必须保存在静态数据区中，返回的是指向静态数据区的指针。正如其它指针一样，指向结构的近指针返回到 AX 中，指向结构的远指针返回到 DX:AX 中。

下面看看具有 Small 模式的可由 C 调用的汇编语言函数 FindLastChar，它返回指向传

入串的最末字符的指针。此函数的 C 形式为

```
extern char * FindLastChar(Char * StringToScan);
```

其中，StringToScan 是非空串，指向其末字符的指针将被返回。

FindLastChar 代码如下：

```
.MODEL small
.CODE
PUBLIC  FindLastChar
FindLastChar PROC
  push    bp
  mov     bp,sp
  cld                         ;we need string instructions to count up
  mov     ax,ds
  mov     es,ax               ;set ES to point to the near data segment
  mov     di,                 ;point ES:DI to start of passed string
  mov     al,0                ;search for the null that ends the strin
  mov     cx,0ffffh           ;search up to 64K～1 bytes
  repnz   scasb               ;look for the null
  del     di                  ;point back to the null
  dec     di                  ;point back to the last character
  mov     ax,di               ;retrun the near pointer in AX
  pop     bp
  ret
FindLastChar ENDP
  END
```

运行的最终结果是，指向传入串末字符的近指针返回到 AX 中。

1.2.3 从 Turbo C 中调用 Turbo Assembler 函数

下面介绍一个由 Turbo C 代码调用 Turbo Assembler 函数的例子。下面的 TurboAssembler 模块——COUNT.ASM 中包含函数 LineCount，它返回传入串的字符数和行数：

```
;Small model C－callable assembler function to count the nuber of
; lines and characters in a zero－terminated string.
;
; Function prototype:
;       extern unsignedt LineCount (char * near StringToCount,
;               unsignedt near * CharacterCountPtr);
; Input:
;       char near * STringToCount: pointer to the  string on which a
;       line count is to be performed
;
;       unsigned int near * CharacterCountPtr:Pointer to the
;               int variable in which the character count is to be stored
;
```

```
NEWLINE     EQU   0ah                    ;the linefeed  character is C's
                                         ; newline character
            DOSSEG
            .MODEL small
            .CODE
            PUBLIC    LineCount
 LineCount  PROC
            push    bp
            mov     bp,sp
            push    si                   ;preserve calling program's register
                                         ;variable, if any
            mov     si,[bp+4]            ;Point SI to the string
            sub     cx,cx                ;set character count to 0
            mov     dx,cx                ;set line count to 0
LineCountLoop:
            lodsb                        ;get the next character
            and     al,al                ;is it null, to end the string?
            jz      EndLineCount         ;yes, we're done
            inc     cx                   ;no,count another character
            cmp     al,NEWLINE           ;is it a newline?
            jnz     LineCountLoop        ;no,check the next character
            inc     dx                   ;yes , count another line
            jmp     LineCountLoop
EndLineCount :
            inc     dx                   ;count the line that ends with the
                                         ;null character
            mov     bx,[bp+6]            ;point to the location at which
                                         ;return the character count
            mov     [bx],cx              ;set the character count variable
            mov     ax,dx                ;return line count as function value
            pop     si                   ;restore calling program's register
                                         ;variable , if any
            pop     bp
            ret
 LintCount  ENDP
            END
```

下面的 C 模块——CALLCT.C 是调用 LineCount 函数的一个例子：

```
char * TestString="Line 1\nline 2\nline3";
extern unsigned int LineCount (char * StringToCount,
        unsigned int * CharacterCountPtr);
main()
{
```

```
        unsigned int LCount;
        unsigned int CCount;
    LCount = LineCount(TestString,&CCount);
    printf("Lines:%d\nCharacters: %d\n",LCount,CCount);
```

用命令行

```
    tcc -ms callct Count.asm
```

编译这两个模块，并将其链接到一起。

正如上面看到的一样，只有与具有 Small 模式的 C 程序链接时，LineCount 才能正常工作。因为在其它模式下，指针的大小和在堆栈模型中的位置会发生变化。以下是另一版本的 LineCount，即 COUNTLG.ASM，它可以由具有 large 模式的 C 程序调用（但不能用于 Small 模式的 C 程序，除非传入远指针，且 LineCount 被声明为远类型）：

```
;Large model C-callable assembler function to count the nuber of
; lines and characters in a zero-terminated string.
;
; Function prototype;
;       extern unsigned int LineCount(char * far StringToCount,
;               unsigned int * far characterCountPtr);
;       char far * StringToCount pointer to the string on which
;                               a line count is to be performed
;       unsigned int far * CharacterCountPtr: pointer to the int variable
                                        in whics the character count
                                        is to be stored
NEWLINE  EQU    0ah                     ;the linefeed character is C's newline
                                        ;character
        .MODEL  large
        .CODE
        PUBLIC   LineCount
LineCount  PROC
        push    bp
        mov     bp,sp
        push    si                      ;preserve calling program's register
                                        ;variable, if any
        push    ds
        lds     si,[bp+6]               ;point DS:SI to the string
        sub     cx,cx                   ;set character count to 0
        mov     dx,cx                   ;set line count to 0
LineCountLoop:
        lodsb                           ;get the next character
        and     al,al                   ;is it null, to end the string?
        jz      EndLineCount            ;yes, we're done
        inc     cx                      ;no,count another character
        cmp     al,NEWLINE              ;is it a newline?
```

```
        jnz         LineCountLoop           ;no,check the next character
        inc         dx                      ;yes , count another line
        jmp         LineCountLoop
EndLineCount :
        inc         dx                      ;count line ending with null character
        les         bx,[bp+10]              ;point ES:BX to the location at which to
                                            ;return the character count
        mov         es:[bx],cx              ;set the character count variable
        mov         ax,dx                   ;return line count as function value
        pop         ds                      ;restore C's standard data seg
        pop         si                      ;restore calling program's register
                                            ;variable , if any
        pop         bp
        ret
LintCount   ENDP
        END
```

可以使用命令

```
tcc -ml callct countlg.asm
```

将 COUNTLG. ASM 链接到 CALLCT. C 上。

1.2.4 Pascal 调用约定

到此为止，已经介绍了 C 如何通过调用码自右至左地将参数压入堆栈，调用函数，在调用之后从栈中删除参数这一系列过程而将参数传递给函数的。Turbo C 也可以遵从 Pascal 程序所采用的约定，即从左至右地传递参数，由被调用函数从堆栈中删除参数。在 Turbo C 中，可以用命令行可选项－P 或关键字 Pascal 启动 Pascal 约定。

下面举一个采用 Pascal 约定的汇编语言函数的例子：

```
        ;
        ; Called    as                  ;TEST(i,j,k);
        ;
i       equ         8                   ;leftmost parameter
j       equ         6
k       equ         4                   ;rightmost parameter
        ;
        .MODE       small
        .CODE
        PUBLIC      TEST
TEST    PROC
        push        bp
        mov         bp,sp
        mov         ax,[bp+i]           ;get i
        add         ax,[bp+j]           ;add j to i
        sub         ax,[bp+k]           ;subtrack k form the sum
```

```
    pop     bp
    ret     6                    ;return,discarding 6 parameter bytes
TEST  ENDP
    END
```

图1.7描述了在执行 MOV BP、SP 之后的堆栈状态：

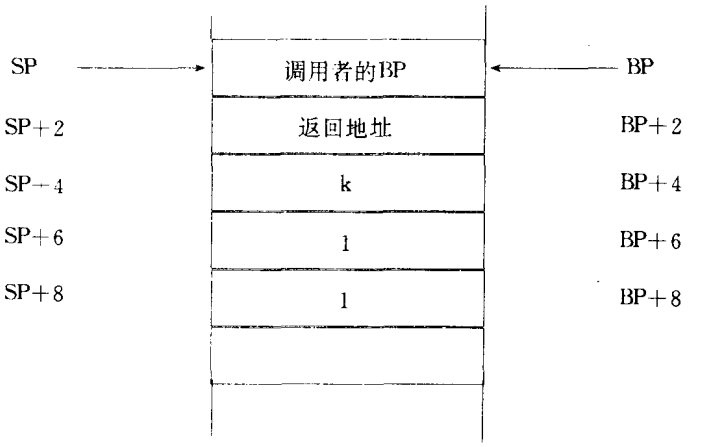


图1.7 MOV BP、SP 后的堆栈状态

注意，被调用函数使用 RET 6从堆栈中删除参数。

Pascal 调用约定也要求所有外部符号和公共符号都是大写形式，且没有前导下划线。为什么可以在 C 程序中采用 Pascal 调用约定呢？

采用 Pascal 约定的代码比正常的 C 代码短一些，执行速度也要快一些，因为每次调用之后不必用指令 ADD SP n 删除参数。有关 Pascal 调用约定中更详细的内容，请参看本书 Pascal 接口部分。

1.3 在 Turbo Assembler 中调用 Turbo C

尽管在 C 中调用汇编语言函数处理特殊问题更为常见，但有时也可能希望在汇编语言中调 C 函数。事实上，从 Turbo Assembler 函数中调用 Turbo C 函数比从 Turbo C 函数中调用 Turbo Assembler 函数还要容易，因为在汇编码中不需要处理堆栈模型。以下将简要介绍如何从汇编语言中调用 Turbo C 函数。

1.3.1 链入 C 的启动码

作为一条通用规则，在将 C 的启动模块作为第一个被链接模块的程序中，最好在汇编语言代码中只调用 Turbo C 的库函数。这类"安全"的程序包括从 TC.EXE 中链接或用命令行 TCC.EXE 链接的所有程序，以及直接用 TLINK 链接，而以 C0T、C0S、C0C、C0M、C0L 或 C0H 作为第一个链接文件的程序。

通常不应在不链入 C 的启动模块的程序中调用 Turbo C 的库函数，因为不链入启动码

时，Turbo C 的一些库函数不能正常运行。如果确实想在这类程序中调用 Turbo C 的库函数，建议用户查阅源启动码(Turbo C 盘上的 C0. ASM 文件)并从 Borland 公司购置 C 库函数的源代码，以便能对所需库函数进行恰当的初始化工作。另外一种可能的途径是，简单地将每个期望的库函数链接到汇编语言程序，如 X. ASM 上。这些汇编语言程序只负责调用每个函数。用下列命令行将其链接到一起：

```
tlink x,x,,cm.lib
```

其中，m 是所期望的内存模式的第一个字母(如 tiny 模式时为 t，small 模式时为 S，等等)。如果 TLINK 报告发现任何未定义的符号，则不能调用该库函数，除非将 C 的启动码链接到此程序上。

注意：如果用户定义的 C 函数调用了 C 中的库函数，那么，调用用户定义的 C 函数与直接调用库函数的过程是一样的；对于直接或间接调用 C 库函数的任何汇编语言程序而言，不链入 C 的启动码都会引起潜在的问题。

1.3.2　确保已正确设置了段

前面已经讲过，用户必须确保 Turbo C 和 Turbo Assembler 使用的是同样的内存模式，而且 Turbo Assembler 所使用的段必须与 Turbo C 所使用的段相匹配。如果有必要重新温习段及内存模式的匹配问题，可翻阅1. 2. 1“Turbo C 与 Turbo Assembler 接口机制”一节。用户同样要记住在所有的段外或在正确的段内对远符号置以 EXTRN 伪指令。

1.3.3　执行调用

在“在 Turbo C 中调用 Turbo Assembler 函数”一节中，已经介绍了 Turbo C 如何准备并执行函数调用的有关内容。下面简要概述 C 函数的调用机制，但这次介绍的是从 Turbo Assembler 中调用 Turbo C 函数。在将参数传给 Turbo C 函数时，所有待完成的工作就是，首先压入最右边的参数，再压入次右边的参数，如此类推，直到压入了最左边的参数为止，然后可以调用 C 函数。例如，如果用 Turbo C 设计一个程序，该程序调用 TurboC 库函数 Strcpy 将 SourceString 复制到 DestString 中，用户可以采用语句

```
strcpy(DestString,SourceString);
```

要在汇编语言中调用同样的函数时，则可以使用

```
lea     ax,SourceString                 ;rightmost parameter
lea     bx,DestString                   ;leftmost parameter
push    ax
push    bx
call      strcpy                        ; copy the string
add     sp,4                            ; discard the parameters
```

在调用之后，别忘了调整 SP 以删除参数。

如果使用 Pascal 调用约定调用 C 函数，则自左至右压入参数，在调用之后也不必调整 SP：

```
lea     ax,DestString                   ;leftmost parameter
push    ax
lea     ax,SourceString                 ;rightmost parameter
```

```
push    ax
call    STRCPY                          ;copy the string
                                        ;don't adjust the stack
```

当然，上例假定用户已用－P 可选项重新编译了 Strcpy，因为标准版本的库函数 strcpy 采用的是 C 而不是 Pascal 调用约定。C 函数对值的返回正如"返回值"一节所描述的一样；8 位和16位值返回到 AX 中，32位值返回到 DS:AX 中，浮点值返回到8087 TOS 寄存器中，而结构的返回方式则因其大小而变化。

C 函数必须保存而且只需保存下列寄存器：SI、DI、BP、DS、SS、SP 和 CS。而对寄存器 AX、BX、CX、DX、ES 和标志则可以自由修改。

1.3.4 在 Turbo Assembler 调用 Turbo C 函数

当需要进行复杂的运算时，用户可能希望在 Turbo Assembler 中调用 Turbo C 函数，因为 C 语言运算比汇编语言运算要容易得多。尤其是整数和浮点数混合运算时更是如此；当然，也可以用汇编语言实现这些运算，但用 C 语言处理类型转换和浮点数运算则更为简单。

下面看一个由汇编语言代码调用 Turbo C 函数以进行浮点数运算的例子。在该例子中，事实上是由 Turbo C 函数将一系列整型数传给 Turbo Assembler 函数，Turbo Assembler 函数计算这些数的和并调用另一个 Turbo C 函数处理浮点数运算，即计算这些数的平均数。

程序的 C 语言部分，即 CALCAVG.C 如下：

```
extern  float Average (int far * ValuePtr,int NumberOfValues);
#define NUMBER  OF  TEST  VALUES 10
int  TestValues[NUMBER  OF  TEST  VALUES] = {
      1,2,3,4,5,6,7,8,9,10
      } ;


main()
{       printf("The average value is: %f\n",
          Average(TestValues,NUMBER  OF  TEST  VALUES));
}
float IntDivide(int Dividend, int Divisor)
{
    return( (float) Dividend / (float) Divisor );
}
```

程序的汇编语言部分即 AVERAGE.ASM 如下：

```
;
; Turbo C－callable small－model function that returns the average
; of a set of integer values. Calls the Turbo C function
; IntDivide() to perform the final division.
;
; Function prototype:
;       extern float Average(int far * ValuePtr,int NumberofValues);
;
```

```
;  Input:
;     int  far   * ValuePtr:      ; the array of values to average
;     int NumberOfValues:         ; the number of values to average

.MODEL     small
EXTRN        IntDivide: PROC
.CODE
PUBLIC       Average
Average  PROC
   push        bp
   mov         bp,sp
   les     bx,[bp+4]              ;point  ES:BX to array of values
   mov     cx,[bp+8]              ;# of values to average
   mov     ax,0                   ;clear the running total
AverageLoop:
   add     ax,es:[bx]             ;add the current value
   add     bx,2                   ;point to the next value
   loop    AverageLoop
   push    WORD PTR  [bp+8]       ;get back the number of values passed to
                                  ;IntDivide as the rightmost parameter
   push    ax                     ;pass the total as the leftmost parameter
   call     IntDivide             ;calcalate the floating-point average
   add     sp,4                   ;discard the parameters
   pop     bp
   ret                            ; the average is in the 8087's TOS register
Average    ENDP
   END
```

C 语言中的 main 函数将指向一个整型数组的指针及数组的长度传给汇编语言函数 Average。函数 Average 计算这些值的和，然后将和及数组元素的个数传给 C 函数 IntDivide。IntDivide 将和及数值个数转换为浮点数并计算平均值，这种过程只需一行 C 代码，但如果用汇编语言则需要好多行。IntDivide 将保存在8087 TOS 寄存器中的平均值返回给 Average，Average 仍将平均值存留在 TOS 寄存器中并返回给 main 函数。

可以用命令行

```
tcc calcavg average. ASM
```

编译 CALCAVG. C 和 AVERAGE. ASM，并将其链接成可执行文件 CALCAVG. EXE。注意，不需要修改任何代码，Average 就可以处理 Small 和 large 数据模式，因为在所有模式下传递的都是远指针。只要使用恰当的. MODEL 伪指令就可以支持 large 代码模式(huge、large 和 medium)。

第二章 Turbo C 与 DOS、BIOS 的接口

通常，用 C 语言编写的源程序与运行程序的操作系统环境是独立的。例如，完成累加计算的 C 源代码，无论是在 Turbo C 这样的 DOS 编译器下运行，还是在 UNIX 或 XENIX 操作系统下运行，结果都相同。

当然，程序在实际执行时依赖于由编译器形成的低级指令在操作环境下完成任务，但你没有必要担心这些不同，这就是高级语言的特点。

然而有时需要利用由特定操作系统或机器本身提供的服务或功能。例如，你的程序可能需要检测当前时间，日期，或检查磁盘的可用存贮空间。这时，需要用到与操作系统或硬件直接通讯的方法。

在 DOS 环境下，利用中断实现通讯。在本章你将学习软中断，还可以学习使用由 DOS 和 PC 机的 ROM-BIOS 提供的函数和服务例程的方法。在讨论如何进行与 DOS 交互之前，先简要地介绍一点背景知识。

2.1 寄存器

基于 Intel 8086系列(包括8088、80286等)处理器的计算机，在各类寄存器中完成了大量工作。这些寄存器是十六位的，在程序运行时，它们用于存贮各种信息。执行程序时，大部分动作或在寄存器内完成或由当时存贮在寄存器内的值控制完成。十四个寄存器中，有四个通用寄存器，它们是 AX、BX、CX 和 DX。在中断函数里用到了这些寄存器。AX 通常被称为累加寄存器，因为通常输入和输出操作经过它才能完成，算术运算也在 AX 内完成。基地址寄存器 BX 用于存贮表的基地址或地址编移量。计数寄存器 CX 通常用于作循环的计数器。最后一个是数据寄存器 DX，用于存贮计算中的数据。

中断还用到其它几个寄存器。标志寄存器 FLAGS 用于说明错误或其它特征信息。虽然它是一个十六位寄存器，但每位代表一个特定的状态或结果。例如，溢出结果为0等等。在很多功能调用中，标志寄存器只能确定有错还是没错。如果有错误，则其它寄存器还会给出错误的详细信息或错误代码。

在一些功能里用到了二个变址寄存器。这两个寄存器通常用于存贮地址，也可以在计算中用于存贮中间结果。这两个寄存器分别为 DI(目的寄存器)和 SI(源寄存器)。例如，在给文件换名和 DOS 服务中，新的名字串的地址放在 DI 寄存器中。

可以一次存取整个寄存器，也可以分别存取寄存器的高八位和低八位。存取整个寄存器

时，可写出整个寄存器名，如 AX；分别存取高、低八位时，可分开，如 AH(AX 的高八位)、AL(AX 的低八位)。

Turbo C 中含有头文件 dos.h，但它并不是 ANSI 标准的一部分。dos.h 提供了调用 DOS 功能所必须的信息和数据结构，文件中包括了 REGS 联合的定义，用它可直接进行寄存器的存取。

一种版本的联合是一个结构，它的成员包括了常用的寄存器，即四个通用寄存器 AX、BX、CX 和 DX，标志寄存器 FLAGS 以及两个变址寄存器(DI 和 SI)。因此，每个成员都是十六位的。在另一种版本的联合定义下，允许存取寄存器的每一个字节，其结构成员是四个通用寄存器的高八位和低八位，即 AH、AL、BH、BL、CH、CL、DH、DL。

当调用 DOS 或 BIOS 完成某一任务时，要用到联合 REGS。请查看 Turbo C 程序包中头文件 dos.h 关于联合 REGS 的定义。

除这个联合的定义外，dos.h 中还包含了在存取寄存器时，所必须有的运行时间库函数的定义。在本章你将学习如何用这些函数实现与 DOS 或 BIOS 的交互。

2.2 中 断

中断实际上是程序或其它地方向 CPU 发送的信号。当响应某中断时，就暂停当前工作，去执行中断。每个中断有相应的中断号，由这个中断号去识别中断源。

有两种软中断方式。第一种是用功能调用直接把中断号传递给操作系统或 BIOS。传递的中断号与所请求的中断相对应。例如，软中断0x25是请求绝对读磁盘，即读出磁盘的每一扇区，不管该扇区是否是文件的一部分。Borland 提供了运行时间库函数 int86()，可直接调用和执行中断。

第二种软中断 DOS 的方法被称为特别软中断0x21，该中断可以调用其它 DOS 服务。有时把这个中断称为功能调度器。由于其职能是调用适当的 DOS 服务函数。使用中断0x21时，还需传递另一个与你所要求的 DOS 服务相应的值。例如，DOS 服务0x36告诉你磁盘的自由空间容量，而服务0x30返回 DOS 的版本号。在调用中断0x21时，一定要把所请求的服务号置入一特定的寄存器。调用运行时间库函数 bdos()或 intdos()可实现这一功能。

在大多数 DOS 中断和服务中，用十六进制数代表中断号，因此，通常看到的是中断0x21，而不是33。这里我们也采用十六进制数。

我们只讨论很少的几种通过功能分配器和直接调用获得的服务。有关 DOS 中断和服务的详细讨论，请查阅有关专用书籍。

2.2.1 使用 DOS 中断的注意事项

调用 DOS 中断时应非常小心，特别是用 DOS 中断改变磁盘内容时。在调用中断重新设置日期以及在磁盘的给定区域写入内容之前，应了解中断的工作原理，以及准确地知道被改变的值是什么，新值又是什么。如果疏忽了这些，可能会引起很多麻烦，最好是查阅有关中断服务的书籍。

注意：中断使得用户可以访问磁盘上的特别区域，如目录区或内部设置区。如不小心，你会无意中改变某些内容，损坏关键文件。在试着用中断写入或改变信息之前，应保护好系统。

同时肯定你所发送的信息是正确无误的，以及这些信息存贮在适当的寄存器中。在使用中断时，应养成一种严谨的作风。

2.3 利用功能调度器实现中断

有两种方法实现 DOS 或 BIOS 服务，即通过功能调度器实现中断和直接中断两种方式。最保险的方法是利用功能调度器。通过功能调度器实现中断，可以保证在不同机型的机器上，程序的运行结果一样，功能调度器可以避免因硬件的差异而影响程序的运行。

采用功能调度器的缺点是执行速度较慢。通过中断0x21发送信息的速度低于直接调用 DOS 或 BIOS 中断发送信息的速度。还有几种服务不宜采用功能调度器，为实现这些中断服务（如视频功能），必须采用直接中断。

本节将介绍如何用功能调度器请求 DOS 服务。在 Turbo C 的运行时间库函数中包括了这一功能的函数 bdos()和 intdos()。函数 bdos()适用于简单服务，而 intdos()用于程序与 DOS 间需大量交换信息的服务请求。

int bdos (int fn _ nr, int dx _ val, int al _ val)

库函数 bdos()调用功能调度器完成由第一个参变量 fn nr 指定的服务，参数 dx val 定义了送入 DX 寄存器的值，如果所用的功能不用到 DX 中的值，则把第二个参数置0，同理第三个参数 al val 给出 AX 低位字节 AL 的值。

因此，bdos()只用到了 DX 和 AX 两个寄存器[功能服务号，即 bdos()的第一个参变量实际上存放在 AH 中]。中断返回后，这个函数返回十六位的整数，下面给出几个调用 bdos()的例子。

此处使用 bdos()的优点在于：你不必为数据结构而担心，只需将 int 型的值传送给函数。但这也正是 bdos()的缺点，函数只适用于使用寄存器 AX 和 DX 的 DOS 中断服务，而对于要求使用 BX、CX 以及其它特殊寄存器的中断服务（如设置错误或其它标志），则不能用 bdos()。

0x1：从键盘读入字符

DOS 服务功能0x1从输入缓冲区（通常是键盘缓冲区）读入一个字符，并在屏幕上显示。函数只简单等待用户将字符输入键盘缓冲区。为调用此服务，需给出调用号。这时，bdos()的第一个参数的值为0x1或为1（在目前的调用下）。由于0x1不再要用其它信息来完成其工作，故把第二个和第三个参变量都置为0。函数自动地把字母回显在屏幕上，你也许还要处理返回值，因此你的程序可以确定所读入的内容。下面是调用 bdos()的一个例子，用到 DOS 服务 0x1：

```
ret val=bdos(0x1,0,0)
```

这个语句完成了以下任务：

1. 它调用 DOS 功能调度器（0x21）完成由第一个参变量指定的服务功能（这时为 0x1）。

2. 启动由第一个参数指定的服务功能。这个服务是根据存在 DX 和 AL 的值去执行

任务，本例为读入字符并回显。

3. 当执行完服务函数后，返回 AX 的值，控制返回到 bdos()。

下述程序允许用户输入希望的内容，按下键 CTRL-D 程序结束。

```
/* Program to illustrat use of selected DOS software interrupts, and to illustrate use of run-time library function bdos().
*/

#include <stdio.h>
#include <dos.h>

void test  0xi ( void); /* routine to illustrate use of service 0x1 */
/* Illustrate use of bdos () for service function 0x1 ———read and echo a character fromstdin.
*/
void test  0x1()
{
        int resp; /* return from call to bdos () */

        printf ( "Type your message, press Ctrl-D when done.\n");

        /* Repeat forever, or until a break.
        NOTE: You can also specify this as a preprocessor macro:
        #define FOREVER for (; ;)
        */
        for (; ; )
        {
              /* call function dispatcher */
              resp = bdos ( 0x1, 0, 0);
              resp &= 0xff;              /* mask high order byte from resp      */

              if ( resp == 0xd)          /* if user typed return                */
                  printf ( "%c", '\12');  /* send linefeed                      */
              else if ( resp == 0x4)     /* if Ctrl-D                           */
                  break;                 /* get out of loop                     */
        } /* end forever */
}

main ()
{
        test  0x1 ();
}
```

在这个例子里有几点需要注意：

首先，返回值 resp 在被用来测试之前，应进行适当处理。一旦调用服务函数，输入字符

就立即回显在屏幕上，返回到 resp 的值实际上是中断完成后，函数 bdos()的返回值。又因 bdos()返回整个 AX 的值(十六位)，然而显示在屏幕上的字符则存贮在 AL 中。AX 的高八位 AH 的值则为前一次使用时的值。因此，在确定刚键入的字符之前，应先屏蔽掉 AX 的高八位。

利用按位与(&)运算，屏蔽掉 AX 的高八位。因此，运算 resp & 0xff 把 resp 中字符值的高八位清零。本例中单独用一条语句屏蔽掉高八位，也可以将调用与屏蔽合在一步中：

```
return=bdos(0x1,0,0) & 0xff
```

第二点是关于 RETURN 键的使用。回车字符(0xd)只简单地发送回车信息，即把光标移到最左列，而实际上还要把光标向下移动一行。因此，还需发送一个辅加的换行字符(0xa)。

在调用高级函数(如运行时间库中的函数)时，你不要做这样的琐事。然而当你调用低级 DOS 中断时，通常需要自己完成许多工作。

最后，control-D 在屏幕上显示为菱形。对于 DOS 功能0x1，这个显示是自动的。你可以在内部过滤掉这一特殊字符而作不返回。读者将会看到另一个获取输入但不回显的功能调用。

0x2：在屏幕上显示一个字符

DOS 服务功能0x2在标准输出设备上(一般指屏幕)显示字符。调用 bdos()时，dx val 的值为准备显示的字符。另外把第三个参数置为0，因为此功能不用 AL 的值，且 bdos()的第一个参数一定为0x2，这是本服务功能的编号。

因此，为了完成与上例中 printf()同样的功能，可用下述方式调用 bdos()的功能0x2，字符'\12'是用八进制表示的换行符。

```
bdos(0x2,'\12',0)
```

这条指令调用 bdos()完成了功能调用0x2功能。

0x3，0x4，0x5：与串行口和打印机的通讯

功能0x3和0x4实现了与串行口的通讯。串行口通常被记为 COML。功能0x3从串行口读入一个字符，其功能与0x1服务相似，但输入源不同。功能0x4向串行口输出字符。

功能0x3有缺陷，特别是当串行口要求以高速操作时。有关此功能的细节及如何选择从 COML 中获取设置信息的方法，参见 DOS 的有关文献。

功能0x5把一个字符发送给打印机。打印机一般标为 LPT1或 PRN。例如，在打印之前可调用这个功能初始化打印机。

下面的程序中包括了实现与串行口通讯的函数以及向打印机发送信息的函数。你可以把这些函数归并在一个程序内，也可以直接调用 bdos()完成相应的功能。后一种方法可以节约调用时间。这几个服务请求用单个函数分别写出，以便于阅读。

```
/* Functions for communicating with serial and printer port,
   using bdos () function as the mediator, and using DOS services.
   If you need to increase speed, dispense with the functions,
   and simply put the calls to bdos() directly into your code.
```

```
   One way to do this, while keeping your code readable is to
   define the DOS service calls as macros. For example:

   #define SERIAL_ RD bdos ( 0x3, 0, 0) & 0xff
 */

/* Read a byte from the serial port, and return this value. */
int serial_ rd ()
{
        int resp;
        resp = bdos ( 0x3, 0, 0) & 0xff;
        return ( resp);
}

/* Send a byte to the serial port. */
void serial_ wrt ( int ch)
{
        ch &= 0xff; /* just to make sure ch is a byte */
        bdos ( 0x4, ch, 0); /* send ch to serial port */
}

/* Send a byte to the printer, usually
LPT1 or PRN. */
void to_ lpt ( int ch)
{
        ch &= 0xff; /* just to make sure ch is a byte */%
        bdos ( 0x5, ch, 0); /* send ch to printer */
}
```

0x7和0x8:无回显读入

功能0x7和0x8从标准输入设备读入字符,且不在屏幕上回显。因此,你可以在内部处理字符,并在需要时再回显。例如下列程序用功能0x8代替0x1,以便在用户使用 CTRL-D 时,避免了显示菱形符号。

```
/* Program to illustrate use of selected DOS software interrupts,
   and to illustrate use of run-time library function bdos().
 */

#include <stdio.h>
#include <dos.h>

void test_0x8 ( void); /* routine to illustrate use of service 0x8 */
/* Illustrate use of bdos () for service function 0x1 ---
```

```
    read and echo a character from stdin.
*/

void test 0x8 ()
{
        int resp; /* return from call to bdos () */
        printf("Type your message , press Ctrl-D when done.\n");
        for ( ; ; )  /*   repeat forever, or until a break   */
        {
                /* call function dispatcher */
                resp = bdos ( 0x8, 0, 0);
                resp &= 0xff; /* mask high order byte from resp */
                if ( resp == 0xd) /* if user typed return */
                {
                                bdos ( 0x2, 0xd, 0); /* "echo" CR */
                                bdos ( 0x2, '/12', 0); /* send linefeed */
                }
                else if ( resp == 0x4) /* if Ctrl  D */
                                break;
                else /* if user types a "normal" character. */
                                bdos ( 0x2, resp, 0); /* echo character */
        } /* end forever */
}

main ()
{
        test 0x8 ();
}
```

函数 test 0x8()象函数 test 0x1()一样，一个字符一个字符地读入输入。然而这个函数不回显所输入的字符。此函数把读入的字符区分类为回车符(0xd)，CTRL-D(0x4)和其它字符。

如输入为回车符，则需发送一个换行符，以便使得光标移到下一行。然而在该函数中你仍需发送一个回车符，此字符象其它字符一样回显。

如果输入为 CTRL-D，则跳出循环。这次函数不回显代表 CTRL-D 的符号，其它字符象调用功能0x2一样回显。

功能0x7和0x8的区别在于它们对 Control-C 和 Control-break 的反应不一样。功能0x8象其它功能一样允许被中断。0x7不允许被打断，且把 Control-C 按一般字符发送。

0x9：写串

到目前为止，你所看到的功能全部处理单个字符。DOS 功能0x9可显示整个串，且串必须以 $ 结束。只有发现了 $ 符，函数才显示串，下列程序调用功能0x9显示当前串。注意，写完

串后，再写回车符和换行符。

```
/* Program to illustrate use of selected DOS software interrupts, and
to illustrate use of run-time library function bdos().
*/

#include <stdio.h>
#include <dos.h>

void test  0x8 ( void); /* routine to illustrate use of service 0x8 */
/* Illustrate use of bdos () for service function 0x1 ——— read and echo a character
from stdin.
*/

void test  0x8 ( )
{
        int resp; /* return from call to bdos () */
        char *message =
                "Type Your message, press Ctrl-D when done.";
        /* write string through bdos (), with 0x9 as service code,
          with address of message as a parameter. */
        bdos ( 0x9, (unsigned) message, 0);
        bdos ( 0x2, 0xd, 0);          /* send CR          */
        bdos ( 0x2, '\12', 0);        /* send linefeed    */
        for ( ; ; ) /* repeat forever, or until a break */
        {
                /* call function dispatcher */
                resp = bdos ( 0x8,0, 0); resp &= 0xff;
                /* mask high order byte from resp */
                if ( resp == 0xd) /* if user typed return */
                {
                        bdos (0x2, 0xd, 0); /* "echo" CR */
                        bdos ( 0x2, '\12', 0); /* send linefeed */
                }
                else if ( resp == 0x4) /* if Ctrl-D */
                        break;
                else /* if user types a "normal" character. */
                        bdos ( 0x2, resp, 0); /* echo character */
        } /* end forever */
}

main ()
{
        test  0x8 ();
```

```
}
```

0xb:检查输入状态

到目前为止,输入服务函数在键盘缓冲区里没有字符时,一直处于等待状态。在一定条件下,这很费时,CPU 可利用这些时间去干别的工作。避免这种情况发生的一种方法是检查键盘缓冲区内是否有字符存在。如果有,则程序可以调用输入处理服务功能读入的这个字符。

DOS 服务功能0xb 可以检查缓冲区。如果键盘缓冲区为空,则 bdos()返回0,否则返回0xff。下列程序调用功能查看用户是否击过键,如果有击键,程序就返回这个字符,并等待下一个字符。如果缓冲区为空,则程序重复显示大写字母。如果要结束程序,可按 CTRL-C。

```
/* Program to illustrate use of bdos () function and, more specifically, use
  of service 0xb ——— check input status.
*/

#include <stdio.h>
#include <dos.h>#define NAX  COUNT 26;
/* used to limit amount written on line */

void test  0xb ( void);

/* Write to screen, until user presses a key.
Then stop until user presses another key.
Continue writing to screen, until ...
*/
void test  0xb ()
{
        int count = 0, out, resp, ch;
        printf ( "Press any key to stop display; Ctrl-C to end.\n");
        resp = bdos ( 0x8, 0, 0); /* get a key, without echo. */
        for ( ; ; )
        {
              resp = bdos ( 0xb, 0, 0) & 0xff; /* keypress? */
              if ( resp == 0xff) /* if yes ... */
              {
                        /* see what's in the keyboard buffer. */
                        ch = bdos ( 0x8, 0, 0) & 0xff;
                        /* write char twice, surrounded by * */
                        bdos( 0x2, '*', 0);
                        bdos(0x2, ch, 0);
                        bdos(0x2, ch, 0);bdos (0x2, '*', 0);
                        /* get another character to start up again */
                        ch = bdos ( 0x8, 0 , 0) & 0xff;
```

```
                } /* end if keyboard buffer had a character */
                out = count % MAX_COUNT; /* to generate letter rows */
                out += 'A';
                bdos ( 0x2, out, 0);
                bdos (0x2, ' ', 0);
                if ( ( count++ % 25) == 24) /* if line is "full" */
                {
                        bdos ( 0x2, 0xd, 0); /* write CR, ... */
                        bdos ( 0x2, 0xa, 0); /* ... and linefeed */
                } /* if time to go to next line */
        } /* end forever */
}

main ()
{
        test_0xb ();
}
```

你的击键将交替地启动或停止显示。显示提示信息后，当你按下第一个键时就开始显示，这时函数 test_0xb()等待输入。注意函数在读入键盘缓冲区的字符前，一直处于等待状态。只要你按下一个键，这个字符就被读入。

函数然后进入无限 for 循环。只要你不按下键，程序就显示字符和空格，每行50个。每循环一次，服务0xb 就检查一次缓冲区。如果缓冲区有字符，则控制传送到功能0x8，读入这个字符。你键入的字符将与其它字符一起显示两次。

由于调用了功能0x8，清除了键入键盘缓冲区的内容，下次调用停止程序的执行，直至键入另一个键。如果缓冲区为空，if 语句中对0x8的第二个调用的条件就得不到满足，因此程序等待输入。一旦再次启动显示，直到你键入另一个键，函数0xb 均使 if 的条件得不到满足。

0x1、0x30：其它服务功能

功能0x30用以确定 DOS 的版本号。用 AX 返回有关信息，低八位 AL 返回 DOS 的主版本号(如1，2，3)，高八位 AH 返回 DOS 的副版本号(如10，11等)。你必须把 bdos()的返回值合并在一起。

功能0x19用以确定缺省的驱动器名。如果缺省为 A 驱动器，则 bdos( )返回的 AH 寄存器的值为0，若为 B 驱动器，返回值为1，若为 C 驱动器，返回值为2。

合并高低八位最简单的方法是用0xff 屏蔽掉高八位，而只留下低八位；用0xff00将低8位置0，然后右移8位，从而得到高八位值。

下列程序说明如何使用这两个功能：

```
/* Program to illustrate use of bdos () function and
DOS services to determine default drive and DOS version.
*/
#include <stdio.h>
#include <dos.h>
```

```
main ()
{
        int resp, drive, major, minor;
        /* determine default drive */
        drive = bdos ( 0x19, 0, 0) & 0xff;
        printf ( "default drive = %c\n", drive + 'a');
        /* determine DOS version */
        resp = bdos ( 0x30, 0, 0);
        major = resp & 0xff; /* major version in low-order byte */
        /* minor version in high order byte */
        minor = (resp * 0xff00) >> 8;
        printf ( "version %d %d\n", major, minor);
}
```

前面这些功能都很简单，他们只用到四个通用寄存器中的两个来传递信息，并且这些寄存器用运行时间库函数 bdos()都能访问到。下一节你会看到一个能调用更复杂服务功能的库函数 intdos()，与 bdos()一样，所有这些功能仍然经过功能调度器实现，即中断0x21。

```
int intdos(union REGS *inreg, union REGS * outreg)
```

与函数 bdos()一样，intdos()也用于软中断并执行服务功能。如果调用时所用寄存器不限于 DX 和 AL 时，必须用 intdos()取代 bdos()。

函数 intdos()也是通过功能调度器激发 DOS 软中断，请求特殊的操作系统服务。然而与 bdos()不同的是，intdos()用包括通用寄存器和其它寄存器的联合传递信息。如前所述这个联合有两种版本，一种使用整个寄存器，另一种使用寄存器中的一个字节。REGS 联合在头文件 dos.h 已有定义。

这里的例子假设 inreg 和 outreg 为外部变量，二者都定义为联合，请求值由 inreg 传递给 DOS 服务，结果则由 outreg 传递给调用函数。

在调用 intdos()之前，应把所希望的服务号置入 inreg.h.ah。根据服务要求，也许还得提供别的信息。在一般情况下，在调用 intdos()之前，这些信息都存贮在相应的联合成员里。在所要求的 DOS 服务完成后，intdos()功能将 AX 的值返回，而第二个指针参数 doutreg 用于返回整个 REGS。

让我们看几个例子，以便了解此函数的工作原理。为了便于阅读，我们采用独立的函数来说明服务请求。

0x36：确定磁盘的可用空间

下列程序告诉你当前磁盘驱动器下的可用存贮空间有多少。函数 get free space()调用 intdos()实现软中断0x21，并请求0x36服务功能，查明并返回磁盘的可用空间数。

```
/* Program to determine available disk space. Program also illustrates use of DOS interrupts.
*/

#include <stdio.h>
#include <dos.h>
```

```
/* used to pass info to and from DOS services */
union REGS inreg, outreg;

long get  free  space ( void);

main ()
{
        long result;
        result = get  free  space ();
        printf ( "%ld total bytes free\n", result);
}

/* Determine available disk space on machine */
long get  free  space ()
{
        long sectors, clusters, bytes, clusters  per  drive;
        long total  free, disk  capacity;
        /* prepare inreg with the required information.
        The AH byte contains the DOS service requested;
        the DL byte contains drive about which you want information
        */
        inreg.h.ah = 0x36;              /* to ask for DOS service 0x36      */
        inreg.h.dl = 0x0;               /* to ask for current drive         */
        /* Call the DOS interrupt 0x21, using intdos                        */
        intdos ( &inreg, &outreg);      /* to use interrupt 0x21            */
        /* results are stored in the registers.
        Total available space can be computed from:
        Sectors per cluster * Bytes per sector * Available Clusters
        */
        sectors = outreg.x.ax;          /* sectors / cluster returned in AX */
        clusters = outreg.x.bx;         /* return available clusters in BX  */
        bytes = outreg.x.cx;            /* bytes per sector returned in CX  */

        /* The following number is not needed to compute FREE space.
        It is useful for computing disk capacity, however.
        */
        clusters  per  drive = outreg.x.dx; /* clusters in DX */

        /* Display information */
        printf ( "sectors == %d/n", sectors);
        printf ("clusters == %d/n", clusters);
        printf ( "bytes == %d/n", bytes);
```

```
        printf ( "clusters  per  drive == %d/n", clusters  per  drive);
        total  free == bytes * sectors * clusters;
        disk  capacity = bytes * sectors * clusters  per  drive;
        printf ( "Total disk capacity == %ld\n", disk  capacity);
        printf ( "Total bytes free == %ld\n", total  free);
        return ( total  free);
    }
```

运行结果为：

```
sectors == 4
clusters == 1636
bytes == 512
clusters  per  drive == 16318
Total disk capacity == 33419264
Total bytes free == 3350528
3350528 total bytes free
```

一旦完成了 DOS 服务，有关磁盘可用空间的信息由联合 outreg 返回。ax 中存放每个簇的扇区数，bx 中为自由簇数，cx 中存放每个扇区的字节数，磁盘的总簇数由 DX 返回（有关磁盘是如何组织扇区和簇的知识，请参见 DOS：The Complete Reference）。

注意：在调用 intdos()之前，每个字节都要控制，返回时则使用了整个寄存器。这在 DOS 中断和服务中很常见。一般 DOS 的服务号存贮在 inreg. h. ah 中，即 AX 的高八位。其余的要用到的信息存贮在其它字节或寄存器里。

0x2a，0x2c：取日期和时间

下列代码是在前例的基础上增加了三个新函数形成的。其中有两个请求 DOS 服务，而第三个函数 how long()用于计算从程序的某一开始点执行到终止点所花费的机时。

```
/* Program to get current date
   and to time how long it takes to do this 1000 times.
   Program also illustrates use of DOS interrupts.
*/

#include <stdio. h>
#include <dos. h>

#define MAX  STR 80
#define FALSE 0
#define TRUE 1
union REGS inreg, outreg;
char *null  str = "";

void get  date ( int *, int *, int * );
double get  time ( void);
double how  long ( double, double);
```

```
/* function to get current date, and to pass information back
in three pieces : month, date, year.
Notice that month and date are stored in bytes, whereas year
is stored in a 16 bit register value.
*/
void get  date ( int *month, int *day, int *year)
{
        /* Ask for DOS service 0x2a */
        inreg.h.ah = 0x2a;
        /* Call DOS interrupt 0x21 */
        intdos ( &inreg, &outreg);
        *month = outreg.h.dh;          /* current month returned in DH      */
        *day = outreg.h.dl;            /* current day returned in DL        */
        *year = outreg.x.cx;           /* current year returned in CX       */
}

/* Function to get the number of seconds elapsed since
midnight on the system clock.
*/
double get  time ()
{
        unsigned int hrs, mins, secs, hundredths;
        /* Request DOS service 0x2c */
        inreg.h.ah = 0x2c;
        /* Call interrupt 0x21 with the appropriate unions */
        intdos ( &inreg, &outreg);
        hrs = outreg.h.ch;             /* hours elapsed returned in CH      */
        mins = outreg.h.cl;            /* minutes elapsed returned in CL    */
        secs = outreg.h.dh;            /* seconds elapsed returned in DH    */
        hundredths = outreg.h.dl;      /* 1/100's secs returned in DL       */
        /* Total elapsed time (in seconds) =
                        3600 (secs / hr) * hrs +
                        60 ( secs / min) * mins +
                        1 ( secs / sec) * secs +
                        .01 ( secs per 1/100th sec) * hundredths.
        */
        return ( (double) hrs * 3600.0 + (double) mins * 60.0 +
        (double) secs + (double) hundredths / 100.0);
}

/* Function to compute the number of seconds elapsed between
  start and finish
```

```
*/

double how  long ( double start, double finish)
{
        * define FULL  DAY 86400.0      /* seconds in a 24 hr day          */
        /* start == first elapsed time measurement;
        finish == second elapsed time measurement.
        If start > finish then the time must have passed midnight
        between start and finish of process.
        In that case, a formula adjustment is necessary.
        */
        if ( start > finish)
              return ( FULL  DAT - start + finish);
        else
              return ( finish - start);
}

main ()
{
        double start, finish;
        int index, month, day, year;
        start = get  time ();              /* start timing                 */
        for ( index = 0; index < 1000; index++)
        {
              get  date ( &month, &day, &year);
        }
        finish = get  time ();             /* stop timing                  */
        printf ( "%d / %d / %d\n", month, day, year);
        printf ( "%10.21f seconds elapsed\n",
        how  long ( start, finish));
}
```

函数 get date()用于确定机器的当前日期。DOS 服务0x2a 提供了这一功能。该服务的返回信息分为月、日、年三部分。注意月和日以一个字节值返回，而年以双字节(16位)返回，同时还可以以数字的形式返回星期。由 AL 返回星期，0表示星期天，1表示星期一。依次类推，修改程序以便能提供这一信息。

函数 get time()调用 DOS 功能服务0x2c，以24小时的计算方式提供系统时间。时间按时、分、秒、秒/10、秒/100的格式提供，用不同的字节返回这些值。小时数在 outreg. h. ch，分钟数在 outreg. h. cl 里，秒数在 outreg. h. dh 里，即百分之一秒数在 outreg. h. dl 里。

要想计算程序中某段的运行时间，可以在程序中要计时部分的开始，调用 get time()，这个调用是系统的当前时间。再在程序中要计时部分的最后，再调用函数 get time()，则两次调用所得时间的差值表明了这段程序的执行时间。假设程序不是在0点左右运行的。

如果第一次对 get time( )的调用在0点之前，而第二次调用则在0点之后，如只作简单

的减法，则结果不正确。因为从午夜开始，时钟又从0开始重新计时。函数 how long()返回两次调用 get time()间的时间差值，并对跨跃0点的情况给予校正。仔细考察代码是如何实现的。

程序调用了上述三个函数，统计对取日期的 DOS 调用执行1000次所花费的时间。

0x2b，0x2d：设置系统时间和日期

可以重新设置系统时间和日期。功能0x2b 允许用户重新设置系统日期。函数 set date ()示例了这一功能，如下所示。

```
/* function to set current date; information is passed
   in three pieces : month, date, year.
   Notice that month and date are stored in bytes, whereas year
   is stored in a 16 bit register value.
*/
int set date ( int month, int day, int year)
{
        /* Ask for DOS service 0x2b */
        inreg.h.ah = 0x2b;
        inreg.h.dh = month;                    /* send required values to service      */
        inreg.h.dl = day;
        inreg.x.cx = year;
        /* Call DOS interrupt 0x21, the function dispatcher                          */
        intdos ( &inreg, &outreg);
        return ( !outreg.h.al);                /* BECOMES true is successful           */
}

/* routine to let user exercise set date () function.                                */
void test set date ()
{
        int outcome, index, month, day, year;
        char info [ MAX STR];
        /* get required information */
        printf ( "month? ");
        gets ( info);
        month = atoi ( info);
        printf ( "day? ");
        gets ( info);
        day = atoi ( info);
        printf ( "year? ");
        gets ( info);
        year = atoi ( info);
        if ( year < 100)                       /* adjust centuries on year.            */
        year += 1900;
```

```
        outcome = set  date ( month, day, year);
        if ( outcome)
                printf ( "Date set successful\n");
        else
                printf ( "Date set UNsuccessful\n");
}
```

日用一个字节存贮，可取1到31间的位。月从1到12，也存在一个字节中。而年则可取1980到2099间的值，用两个字节存贮。其它值被认为非法，并显示有关服务失败的信息。

outreg. h. al 中的值表明了日期的设置是否成功。如果 outreg. h. al 等于0，则说明日期已改变；如果为0xff，则说明调用失败。

为修改原系统时间信息，应把所期望的值先置入适当的寄存器，然后使用 intdos( )调用功能服务0x2d 来进行修改，功能0x2c 和0x2d 也要用到同样的寄存器。函数 set time()表明了如何调用功能0x2d。

```
/* function to set current time; information is passed
  in four pieces : hours, minutes, seconds, hundredths of a second.
*/
int set  time ( int hours, int minutes, int seconds, int hundredths)
{
        /* Ask for DOS service 0x2d */
        inreg. h. ah = 0x2d;
        inreg. h. ch = hours;              /* send required values to service      */
        inreg. h. cl = minutes;
        inreg. h. dh = seconds;
        inreg. h. dl = hundredths;
        /* Call DOS interrupt 0x21, the function dispatcher */
        intdos ( &inreg, &outreg);
        return ( !outreg. h. al);          /* BECOMES true is successful           */
}

/* routine to let user exercise set  time () function. */
void test  set  time ()
{
        int outcome, index, hour, min, sec, hundredth;
        char info [ MAX  STR];
        /* get required information */
        printf ( "hour? ");
        gets ( info);
        hour = atoi ( info);
        printf ( "minute? ");
        gets ( info);
        min = atoi ( info);
        printf ( "second? ");
```

```
        gets ( info);
        sec = atoi ( info);
        printf ( "1/100's of a second? ");
        gets ( info);
        hundredth = atoi ( info);
        outcome = set  time ( hour, min, sec, hundredth);
        if ( outcome)
                printf ( "time set was successful\n");
        else
                printf ( "time set was UNsuccessful\n");
}
```

小时和分钟分别存储在 CX 的高低字节中。CH 的值(小时)从0到23,而 CL(分钟)的值从0到59。同理,DH 的值(秒)从0到59,而存于 DL 中的百分之一秒则从0到99。

象服务0x2b 一样,如果改变成功,则 outreg 中 AL 的值为0,否则为0xff。

0x39、0x3a、0x3b:处理子目录

如果你使用过目录和子目录,则一定熟悉如下 DOS 命令:mkdir 或 md、rmdir 或 rd、以及 chdir 或 cd。这些命令用于子目录的建立,删除以及改变子目录的选择。

服务0x39用于建立子目录,这个服务功能的工作与 mkdir 或 md 相似。在下列代码中的函数 mk dir()说明了如何调用这个服务功能。

```
/* Program to test service for creating a directory.  */

#include <stdio.h>
#include <dos.h>

#define TRUE 1
#define FALSE 0
#define MAX  STR 80

union REGS inreg, outreg;

int mk  dir ( char *);
int test  mk  dir ( void);

/* Create a directory having the specified name.  */
int mk  dir ( char *name)
{
        inreg.h.ah = 0x39;              /* service request code        */
        /* starting address of name string goes in DX register.  */
        inreg.x.dx = (unsigned) &name [ 0];
        intdos ( &inreg, &outreg);      /* call function dispatcher    */
        if ( outreg.x.cflag)            /* cflag is nonzero on error   */
```

```
        {
            printf ( "Error. No subdirectory created. ");
            if ( outreg. x. ax == 3) / * if error code == 3 * /
                printf ( "Path not found. \n");
            else / * other error code * /
                printf ("Not allowed or unable to create. \n");
            return ( FALSE);              / * report failure          * /
        } / * end if directory creation error. * /
        else                              / * if directory created    * /
        return ( TRUE);                   / * report success          * /
}

int test  mk  dir ()
{
        char info [ MAX  STR];
        int outcome;
        printf ( "Name of directory to create? ");
        gets ( info);
        outcome = mk  dir ( info);
        if ( outcome)
            printf ( "Directory created. \n");
        else
            printf ( "No directory created. \n");
        return ( outcome);
}

main ()
{
        int result;
        result = test  mk  dir ();
}
```

为了调用功能0x39，必须传送目录名的地址(包括驱动器和路径的信息)，把它们放在 inreg. x. dx。指定内存单元的内容是以空格符('\0')结束的 ASCII 码。我们把这样的串称为 ASCIIZ 串。

如果在给定的名下服务功能不能创建子目录，则把 outreg. x. cflag 置为非零值，并把错误类型存放在 outreg. x. ax 里。出错的原因可能是所给路径不存在，或给定目录名和路径下已有同名子目录存在，或存贮空间不够。

服务0x3a 和0x3b 具有同样的参数，即包含目录名的 ASCIIZ 串的地址，为删除指定的子目录，调用函数 intdos()的服务功能0x3a，如函数 rm dir()所示。与功能0x39一样，如果调用失败，outreg. x. cflay 的值非零。如果子目录未找到，或者指定要删掉的为当前目录，或者要删掉的目录下还有文件存在，则无法删去该目录。

为选择一个新的子目录，把包含子目录名的 ASCIIZ 串赋给 inreg. x. dx。功能0x3b 把指

定目录变为当前目录或在 outreg. x. cflag 中返回错误值。并把错误代码存于 outreg. x. ax 里，当找不到给定的子目录时，返回的错误代码为3，这是本服务返回的唯一的错误码。

函数 rm dir()和 ch dir()分别说明了功能0x3a 和功能0x3b 的调用方法。

```
/*.Program to test services for removing a directory
and switching directories.
*/

#include <stdio.h>
#include <dos.h>

#define TRUE 1
#define FALSE 0
#define MAX STR 80
union REGS inreg, outreg;

int ch dir ( char * );
int test ch dir ( void);
int mk dir ( char * );
int test mk dir ( void);
int rm dir ( char * );
int test rm dir ( void);

/* Create a directory having the specified name. */
int mk dir ( char *name)
{
        inreg.h.ah = 0x39;                  /* service request code        */
        /* starting address of name string goes in DX register */
        inreg.x.dx = (unsigned) &name [ 0];
        intdos ( &inreg, &outreg);          /* call function dispatcher    */
        if ( outreg.x.cflag)                /* cflag is nonzero on error   */
        {
              printf ( "Error. No subdirectory created. ");
              if ( outreg.x.ax == 3)        /* if error code == 3          */
                    printf ( "Path not found.\n");
              else                          /* other error code            */
                    printf ("Not allowed or unable to create.\n");
              return ( FALSE);              /* report failure              */
        } /* end if directory creation error. */
        else                                /* if directory created        */
        return ( TRUE);                     /* report success              */
}
```

```
int test  mk  dir ()
{
        char info [ MAX  STR];
        int outcome;
        printf ( "directory name? ");
        gets (info);
        outcome = mk  dir ( info);
        if ( outcome)
              printf ( "Directory created. \n");
        else
              printf ( "No directory created. \n");
        return ( outcome);
}

/* Remove the specified directory */
int rm  dir ( char *name)
{
        inreg. h. ah = 0x3a;                    /* service code to remove directory
        inreg. x. dx = (unsigned) &name [ 0];
        intdos ( &inreg, &outreg);              /* call function dispatcher     */
        if ( outreg. x. cflag)                  /* cflag is nonzero on error    */
        {
              printf ( "Error. Subdirectory not deleted. ");
              switch ( outreg. x. ax)           /* which error code?            */
              {
                  case 3:
                        printf ( "Path not found. \n");
                        break;
                  case 6 :
                        printf ( "Can't delete current ");
                        printf ( "directory. \n");
                        break;
                  default :
                        printf ( "Possibly not empty. \n");
                        break;
              }
              return ( FALSE);                  /* report failure               */
        } /* end of error condition                                             */
        else /* if directory deleted                                           */
        return ( TRUE);                         /* report success               */
}

int test  rm  dir ()
```

```
{
        char info [ MAX   STR];
        int outcome;
        printf ( "Name of directory to remove? ");
        gets (info);
        outcome = rm   dir ( info);
        if ( outcome)
              printf ( "Directory removed. \n");
        else
              printf ( "No directory deleted. \n");
        return ( outcome);
}

/* Change to the specified directory */
int ch   dir ( char *name)
{
        inreg. h. ah = 0x3b;                    /* service code to change directories
        inreg. x. dx = (unsigned) &name [ 0];
        intdos ( &inreg, &outreg);              /* call function dispatcher     */
        if ( outreg. x. cflag)                  /* cflag is nonzero on error    */
        {
              printf ( "Error. No move made. ");
              if ( outreg. x. ax == 3)          /* if error code == 3           */
                   printf ( "Path not found. \n");
              else /* if other error code */
                   printf ( "Unable to move. \n");
              return ( FALSE);                  /* report failure               */
        } /* end of error condition */
        else /* if directory created */
        return ( TRUE);                         /* report success               */
}

int test   ch   dir ()
{
        char info [MAX   STR];
        int outcome;
        printf ( "Name of new directory? ");
        gets ( info);
        outcome = ch   dir ( info);
        if (outcome)
              printf ( "Moved to new directory. \n");
        else
              printf ( "No move made. \n");
```

```
        return ( outcome);
}

main ()
{
        int result;
        result = test  ch  dir ();
        result = test  rm  dir ();
}
```

0x3c、0x5b、0x3d、0x3e：处理文件句柄

DOS2.0以上的版本允许用户根据被称作文件句柄的整数值存取文件。对于每一个打开的文件，都有一个与之相连的文件句柄，包括5个系统自动打开的文件：stdin、stdout、sterr、stdaux 和 stdprn。

因此，当 DOS 服务创建或打开一个文件时，函数返回与这个文件相关的文件句柄。在调用服务功能时，这个句柄代替了相应的文件名。虽然最多可使用99个文件句柄，但在单个程序里只能使用20个句柄。另外，20个句柄中的5个被用于打开5个缺省文件。

DOS 提供了两个服务功能0x3c 和0x5b，它们用于创建新文件。功能0x3c 把已存在的文件长度缩短为0，因此，文件中的内容将丢失。另一方面，如果用功能0x5b 创建一个文件，但这个文件已存在，则返回一个错误代码告诉你文件已存在，但并不破坏原文件。因此，用功能0x5b 创建文件较安全，它可以阻止丢失文件。

在调用这两个功能时，需要先把功能号0x3c 或0x5b 置入 inreg. h. ah 里，把文件属性赋给 inreg. x. cx。把包含文件名和路径名的 ASCIIZ 串的起始位置置入 inreg. x. dx。缺省时，这两个功能打开的文件为读/写文件。将 CX 寄存器置为0x2就明确规定了打开的文件为读/写文件；当寄存器 CX 为0x1时，文件为只读文件；当寄存器值为0x2时，文件为隐藏文件；当 CX 值为4时，文件值为系统文件。

如果成功，服务功能把句柄返回给 AX，否则把如下的错误代码之一返回给 AX：

- 0x3：找到给定路径。
- 0x4：已无可用句柄，即打开文件的数目已达到所允许的最大值。
- 0x5：服务功能不能存取文件。例如，已有的同名文件为只读文件。

如果 AX 中的错误代码为0x50，则说明给定文件已存在，故不能再建立同名文件。这个代码只对功能0x5b 有效，而对功能0x3c 无意义。由于后者不进行这个检查。如果已建立好一个文件，则返回时 cflag 为0，如果有错则返回非零。后面的关于这两个服务功能的代码都示例了这些功能。

功能0x3d 可打开一个已存在的文件，除把功能号存放在 AH 中外，还可以把打开文件的属性存于 AL 的后三位(位0～2)，且0x0表示只读，0x1表示只写，而0x2表示读/写。

在调用 intdos()之前，应把文件名的起始位置放入 DX。当打开一个文件时，用于指示下一个操作动作发生位置的文件指针指向文件的头部。请参阅有关参考书籍，看如何用 DOS 服务功能0x42移动文件的指针在第六章中，通过使用运行时间库函数 fseek()完成了类似的动作。

功能0x3d 在 REGS 联合的成员 cflag 和 AX 中返回结果信息。如果 outreg. x. cflag＝0，则说明调用成功，且 outreg. x. ax 的值为句柄。如果 cflag 为非零值，则 outreg. x. ax 中存放如下的错误代码之一：

- 0x2：找不到给定文件；
- 0x3：找不到所给路径；
- 0x4：已打开的文件数已达到最大允许值；
- 0x5：由于某种原因，拒绝存取文件；
- 0xC：存取代码无效。

服务功能0x3e，关闭已知句柄为参数的功能。函数从 BX 中读取被关闭的文件的文件句柄，输出信息由 outreg. x. cflag 和 outreg. x. ax 返回。当前唯一有效的错误代码为0xc，表明指定句柄无效。

下列程序示例了调用上述几个服务功能的函数。

```
/* Program to test services for creating, deleting,
and opening files.
*/

#include <stdio.h>
#include <dos.h>

#define TRUE 1
#define FALSE 0
#define MAX  STR 80

Union REGS inreg, outreg;

char *null  str = "";
int cl  file (int);
int test  cl  file ( void);
it mk  file ( char *, int);
int test  mk  file ( void);
int op  file ( char *, int);
int test  ip  file ( void);
int safe  mk  file ( char *, int);
int test  safe  mk  file ( void);

main ()
{
        int result;
        /* result = test  mk  dir ();
        reslut = test  ch  dir (); */
        do
        result = test  safe  mk  file ();
```

```
        while ( result > 0);
        do
            result = test  cl  file ();
        while ( result > 0);
}

/* create a file and return file handle, if successful.
  WARNING: any existing file with the specified name will be
  overwritten.. To avoid this, use safe  mk  file ();
*/
int mk  file ( char *name, int attrib)
{
        inreg. h. ah = 0x3c;                    /* code for file creation service      */
        inreg. x. cx = attrib;                  /* file attribute -- e. g. read-only */
        /* starting location of file name, including path info                        */
        inreg. x. dx = (unsigned) &name [ 0];
        intdos ( &inreg, &outreg);              /* call function dispatcher           */
        if ( outreg. x. cflag)                  /* cflag is nonzero on error          */
        {
            printf ( "Error code %d: \n", outrge. x. ax);
            if ( outreg. x. ax == 3) /* if error code == 3 8?
                printf ( "path not found. \n");
            else if ( outreg. x. ax == 4)       /* error code == 4                    */
                printf ( "too many files open. \n");
            else /* other error code */
                printf ( "file is read-only. \n");
            return ( FALSE);                    /* report failure                     */
        } /* end of error condition */
        else /* if file was created */
        return ( outreg. x. ax);                /* return file handle                 */
}

int test  mk  file ()
{
        char name [ MAX  STR]. info [ MAX  STR];
        int outcome = 0, attribute;
        printf ( "Name of file to create:");
        gets ( name);
        if ( strcmp ( name, null  str) = 0)
            outcome = -1;
        if ( outcome >= 0)                      /* if file name is not null string.   */
        {
            /* 0 == read/write; 1 == read-only; 2 == hidden;
```

```
                32 == archive;
                */
                printf ( "File attribute? (0, 1, 2, 4, 32)? ");
                gets ( info);
                attribute = atoi ( info);

                outcome = mk  file ( name, attribute);
                if ( outcome)                    /* if successful                    */
                        printf ( "File created, handle == %d. \n", outcome);
                else
                        printf ( "No file created. \n");
        }/* if file name was not empty */
        return ( outcome);
}
/* open an existing file and return file handle, if successful. */
int op  file ( char *name, int attrib)
{
        inreg. h. ah = 0x3d;                     /* code for file opening service    */
        inreg. x. cx = attrib;                   /* file attribute -- e. g. read-only */
        /* starting location of file name, including path info                       */
        inreg. x. dx = (unsigned) &name [ 0];
        intdos ( &inreg, &outreg);               /* call function dispatcher         */
        if ( outreg. x. cflag)                   /* cflag is nonzero on error        */
        {
                printf ( "Error code %d: \n", outreg. x. ax);
                if ( outreg. x. ax == 3)         /* if error code == 3               */
                    printf ( "path not found. \n");
                else if ( outreg. x. ax == 4)    /* error code == 4                  */
                    printf ( "too many files open. \n");
                else if ( outreg. x. ax == 5)
                    printf ( "file is read-only. \n");
                else
                    printf ( "File not found?\n");
                return ( FALSE); /* report failure */
        } /* end of error condition */
        else /* if file was created */
        return ( outreg. x. ax);                 /* return file handle               */
}

int test  op  file ()
{
        char name [ MAX  STR], info [ MAX  STR];
        int outcome = 0, attribute;
```

```
        printf ( "Name of file to open?");
        gets ( name);
        if ( strcmp ( name, null  str) == 0)
                outcome = -1;
        if ( outcome >= 0)                      /* if file name is not null string.      */
        {
                /* 0 == read/write; 1 == read-only; 2 == hidden;
                32 == archive;
                */
                printf ( "File attribute? (0, 1, 2, 4, 32)/ ");
                gets ( info);
                attribute = atoi ( info);
                outcome = op  file ( name, attribute);
                if ( outcome) /* if successful */
                        printf ( "File opened, handle == %d. \n",outcome);
                else
                        printf ( "No file opened. \n");
        } /* if file name was not empty */
        return ( outcome);
}

/* close the file specified by the handle */
int cl  file ( int handle)
{
        inreg. h. ah = 0x3e;                    /* code for file closing service      */
        inreg. x. bx = handle;                  /* file handle, to identify file      */
        intdos ( &inreg, &outreg);
        if ( outreg. x. cflag)                  /* cflag is nonzero on error          */
        {
                printf ( "Error code %d: \n", outreg. x. ax);
                printf ( "Invalid handle or file not open. \n");
                return ( FALSE);                /* report failure                     */
        } /* end error condition */
        else
        return ( TRUE);                         /* report success                     */
}
int test  cl  file ()
{
        char infor [ MAX  STR];
        int outcome = 0, handle;
        printf ( "File handle? \n");
        gets ( info);
        handle = atoi ( info);
```

```
        if ( handle > 0)
            outcome = cl  file ( handle);
        if ( outcome)
            printf ( "File with handle == %d closed. \n",handle);
        else
            printf ( "No file closed. \n);
        return ( outcome);
}

/* create a file and return file handle, if successful. */
int safe  mk  file ( char *name, int attrib)
{
        inreg. h. ah = 0x5b;                    /* code for file creation service      */
        inreg. x. cx = attrib;                  /* file attribte -- e. g. read-only    */
        /* starting location of file name, including path info                        */
        inreg. x. dx = (unsigned) &name [ 0];
        intdos ( &inreg, &outreg);              /* call function dispatcher            */
        if ( outreg. x. cflag)                  /* cflag is nonzero on error           */
        {
            printf ( "Error code %d: \n", outreg. x. ax);
            if ( outreg. x. ax == 3)            /* if error code == 3                  */
                printf ( "path not found. \n");
            else if (outreg. x. ax == 4)        /* error code == 4                     */
                printf ( "too many files open. \n");
            else if ( outreg. x. ax == 5)       /* error code == 5                     */
                printf ( "no room to create file. \n");
            else if ( outreg. x. ax == 0x5o)    /* error code == 80                    */
                printf ( "file alrady exists. \n");
            else
                printf ( "unknow error, no file created. \n");
            return ( FALSE);                    /* report failure                      */
        } /* end of error condition */
        else /* if file was created */
        return ( outreg. x. ax);                /* return file handle                  */
}

int test  safe  mk  file ()
{
        charname [ MAX  STR], info [MAX  STR];
        int outcome = 0, attribute;
        printf ( "Name of file to create?");
        gets ( name);
        if ( strcmp ( name, null  str) == 0)
```

```
        outcome = -1;
    if ( outcome >= 0)                    /* if file name is not null string.    */
    {
        /* 0 == read/write; 1 == read-only; 2 == hidden;
        32 == archive;
        */
        printf ( "File attribute? (0, 1, 2, 4, 32)? ");
        gets ( info);
        attribute = atoi ( info);
        outcome = safe  mk  file ( name, attribute);
        if ( outcome)                     /* if successful                       */
            printf ( "File created, handle == %d. \n",outcome);
        else
            printf ( "No file created. \n");
    } /* if file name was not empty */
    return ( outcome);
}
```

0x3f、0x40：读写文件

可用功能0x3f从一个文件里读取一定数目的字节。此服务功能假定你已调用DOS功能打开一个可读文件。在调用intdos( )完成这个服务之前，须提供如下的信息：

- 给出要读的文件的句柄；
- 要读的字节数；
- 存放读出内容的变量的起始地址。

最后一项通常是串。必须确保有足够的存贮空间存储读出的内容。

当这一功能完成时，根据cflag中的值，决定AX中存贮的是读出的字节数还是错误代码。如果无错误(cflag=0)，AX中的数在0到应读出的字节数之间。

如果AX=0，则说明已达到文件尾，无任何内容可读。如果AX非零，且小于指定要读出的字节数，说明只读了文件的一部分。这种情况可能发生在接近文件末尾时，这时已有的字符太少，无法填写满缓冲区功能0x40用于写一个文件，需要提供以下信息：文件句柄(存于inreg. x. bx)，要写入的字节数(存于inreg. x. cx)，要写入的字节的起始地址(存于inreg. x. dx)。

如果执行成功，此服务功能将写入的字符数或0返回给AX，这时outreg. x. cflag=0。如果outreg. x. ax=0，则说明当前磁盘已满。如果outreg. x. ax小于要写入的字节数，则可能有错。

如果没有写入任何内容，则outreg. x. cflag为非零，且在outreg. x. ax中存有错误码。如果AX的值为5，说明文件为只读文件。AX的值为6，说明文件句柄无效或文件未打开。

下列程序说明如何从文件读出内容或将内容写入一个文件。注意这里用整形参数定义文件，这就指出了与文件相应的文件句柄。此参数的形式与传递给如运行库函数fopen ()、fclose()和fprintf()的参数形式不一样。这些运行库函数以文件指针做参数。尽管这两类参

数提供同样的信息。但文件指针(FILE *)和文件句柄(int)有区别。

```
/* Program to test services for reading and writing files. */

#include <stdio.h>
#include <dos.h>

#define TRUE 1
#define FALSE 0
#define MAX_STR 80
#define BUFF_SIZE 50

union REGS inreg, outreg;

char *null_str = "";
int cl_file(int);
int test_cl_file(void);
int mk_file(char *, int);
int test_mk_file(void);
int op_file(clar *, int);
int test_op_file(void);
int safe_mk_file(char *, int);
int test_safe_mk_file(void);
int rd_file(int, int, char *);
int wr_file(int, int, char *);

main()
{
        int r_handle, w_handle, r_result, w_result;
        char str[MAX_STR];
        r_handle = test_op_file();
        w_handle = test_mk_file();
        if(r_handle && w_handle)              /* if both files were opend          */
        {
                do /* work as long as source file not empty */
                {
                    r_result = rd_file(r_handle, BUFF_SIZE, str);
                    if(r_result < 0)          /* if read error                     */
                    {
                            printf("Error reading file.\n");
                            exit(1);
                    }
                    if(r_result == 0)         /* end of source file                */
                            printf("End of file reached.\n");
```

```
                /* if only a partial read */
                else if ( r  result < BUFF  SIZE)
                        str [r  result] = '\0';
                else /* if a full read */
                        str [BUFF  SIZE] = '\0';
                printf ( "%s\n", str);
                /* Notice r  result is used for write length.
                    Otherwise, garbage characters get written.
                */
                if ( r  result > 0)          /* if something read          */
                        w  result = wr  file ( w  handle,r  result, str);
            }
            while ( r  result > 0);
            cl  file ( r  handle);
            cl  file ( w  handle);
        }
}

/* create a file and return file handle, if successful.
  WARNING: any existing file with the specified name will be
  overwritten. To avoid this, use safe  mk  file ();
*/

int mk  file ( char *name, int attrib)
{
        inreg.h.ah = 0x3c;                  /* code for file creation service   */
        inreg.x.cs = attrib;                /* file attribute -- e.g. read-only */
        /* starting location of file name, including path info */
        inreg.x.dx = (unsigned) &name [0];
        intdos ( &inreg, &outreg);          /* call function dispatcher         */
        if ( outreg.x.cflag)                /* cflag is nonzero on error        */
        {
                printf ( "Error code %d: \n", outreg.x.ax);
                if ( outreg.x.ax == 3)      /* if error code == 3               */
                        printf ( "path not found.\n");
                else if ( outreg.x.ax == 4) /* error code == 4                  */
                        printf ( "too many files open.\n:);
                /* other error code */
                printf ( "file is read-only.\n"
                return ( FALSE)             /* report failure                   */
        } /* end of error condition */
        else  /* if file was created */
        return ( outreg.x.ax);              /* return file handle               */
```

```
}

int test  mk  file ()
{
        char name [ MAX  STR],info [MAX  STR];
        int outcome = 0, attribute;
        printf ( "Name of file to create:");
        gets ( name);
        if ( strcmp ( name, null  str) == 0)
                outcome = -1;
        if ( outcome >= 0)                      /* if file name is not null string.      */
        {
                /* 0 == read/write; 1 == read-only; 2 == hidden;
                32 == archive;
                */
                printf ( "File attribute? (0, 1, 2, 4, 32)? ");
                gets ( info);
                attribute = atoi ( info);
                outcome = mk  file ( name, attribute);
                if ( outcome)                   /* if successful                         */
                        printf ( "File created, handle == %d. \n:,outcome);
                else
                        printf ( "No file created. \n");
        } /* if file name was not empty */
        return ( outcome);
}

/* open as existing file and return file handle, if successful. */
int op  file ( char *name, int attrib)
{
        inreg. h. ah = 0x3d;                    /* code for file opening service         */
        inreg. x. cs = attrib;                  /* file attribute -- e. g. read-only     */
        /* starting location of file name, including path info */
        inreg. x. dx = (unsigned) &name [ 0];
        intdos ( &inreg, &outreg);              /* call function dispatcher              */
        if ( outreg. x. cflag)                  /* cflag is nonzero on error             */
        {
                printf ( "Error code %d: \n", outreg. x. ax);
                if ( outreg. x. ax == 3)        /* if error code == 3                    */
                        printf ( "path not found. \n");
                else if ( outreg. x. ax == 4)   /* error code == 4                       */
                        printf ( "too many files open. \n");
                else if ( outreg. x. ax == 5)
```

```
                printf ( "file is read-only. \n");
        else
                printf ( "File not found:\n");
        return ( FALSE);                    /* report failure                */
    } /* end of error eoncition */
    else /* if file was created */
    return ( outreg. x. ax);                /* return file handle            */
}
int test  op  file ()
{
    char name [ MAX  STR], info [ MAX  STR];
    int outcome = 0, attribute;
    printf ( "Name of file to open? ");
    gets ( name);
    if ( strcmp ( name, null  str) == 0)
        outcome = -1;
    if ( outcome >= 0)                      /* if file name is not null string.  */
    {
        /* 0 == read/write; 1 == read-only; 2 == hidden;
        32 == archive;
        */
        printf ( "File attribute? (0, 1, 2, 4, 32)? ");
        gets ( info);
        attribute = atoi ( info);
        outcome = op  file ( name, attribute);
        if ( outcome)                       /* if successful                 */
            printf ( "File opened, handle == %d. \n",outcome);
        else
            printf ( "No file opened. \n");
    } /* if file name was not empty */
    return ( outcome);
}

/* close the file specified by the handle */
int cl  file ( int handle)
{
    inreg. h. ah = 0x3e;                    /* code for file colsing service  */
    inreg. x. bs = handle;                  /* file handle, to identify file  */
    intdos ( &inreg, &outreg);
    if ( outreg. x. cflag)                  /* cflag is nonzero on error      */
    {
        printf ( "Error code %d: \n", outreg. x. ax);
        printf ( "Invalid handle or file not open. \n");
```

```
                return ( FALSE);                    /* report failure                    */
        } /* end error condition */
        else
        return ( TRUE); /* report success */
}

int test  cl  file ()
{
        char info [ MAX  STR];
        int outcome = 0, handle;
        printf ( "File handle" \n");
        gets ( info);
        handle = atoi ( info);
        if ( handle > 0)
                outcome = cl  file ( handle);
        if ( outcome)
                printf ( "File with handle == %d closed. \n",handle);
        else
                printf ( "No file closed. \n");
        return ( outcome);
}

/* create a file and return file handle, if successful. */
int safe  mk  file ( char *name, int attrib)
{
        inreg. h. ah = 0x5b;                        /* code for file creation service    */
        inreg. x. cx = attrib;                      /* file attribute -- e. g. read-only */
        /* starting location of file name, including path info */
        inreg. x. dx = (unsigned) &name [ 0];
        intdos ( &inreg, &outreg);                  /* call function dispatcher          */
        if ( outreg. x. cflag)                      /* cflag is nonzero on error         */
        {
                printf ( "Error code %d: \n", outreg. x. ax);
                if ( outreg. x. ax == 3)            /* if error code == 3                */
                        printf ( "path not found. \n"0;
                else if ( outreg. x. ax == 4)       /* error code == 4                   */
                        printf ( "too many files open. \n");
                else if ( outreg. x. ax == 5)       /* error code == 5                   */
                        printf ( "no room to create file. \n");
                else if ( outreg. x. ax == 0x50)    /* error code == 80                  */
                        printf ( "file already exists. \n");
                else /* other error code */
                        printf ( "unknown error, no file created. \n");
```

```
            return ( FALSE);                    /* report failure                    */
        } /* end of error condition */
        else /* if file was created */
        return ( outreg.x.ax);                  /* return file handle                */
}

int test  safe  mk  file ()
{
        char name [MAX  STR], info [ MAX  STR];
        int outcome = 0, attribute;
        printf ( "Name of file to create?");
        gets ( name);
        if ( strcmp ( name, null  str) == 0)
                outcome = -1;
        if ( outcome >= 0) /* if file name is not null string. */
        {
                /* 0 == read/write; 1 == read-only; 2 == hidden;
                32 == archive;
                */
                printf ( "File attribute? (0, 1, 2, 4, 32)/ ");
                gets ( info);
                attribute = atoi ( info);
                outcome = safe  mk  file ( name, attribute);
                if ( outcome) /* if successful */
                        printf ( "File created, handle == %d. \n",outcome);
                else
                        printf ( "No file created. \n");
        } /* if file name was not empty */
        return ( outcome);
}

/* Write specified number of bytes to file */
int wr  file ( int handle, int size, char *message)
{
        inreg.h.ah = 0x40;
        inreg.x.bx = handle;
        inreg.x.cx = size;
        inreg.x.dx = (unsigned) &message [ 0];
        intdos ( &inreg, &outreg);
        if ( outreg.x.cflag)
        {
                printf( "Error writing, code == %d\n",outreg.x.ax);
                if ( outreg.x.ax == 5)
```

```
                printf ( "file is read only?\n");
        else
        printf ( "file not open?\n");
        return ( -1);
    }
    else / * if no error * /
    {
        if ( outreg. x. ax == 0)
            printf ( "end of file reached. \n");
        else if ( outreg. x. ax < size)
        {
            printf ( "Partial file write of %d chars. \n",outreg. x. ax);
        }
    }
    return ( outreg. x. ax);
}

/ * Read specified number of bytes from file * /
int rd  file ( int handle, int size, char * message)
{
    inreg. h. ah = 0x3f;
    inreg. x. bx = handle;
    inreg. x. cx = size; / * bytes to read * /
    inreg. x. dx = (unsigned) &message [ 0];
    intdos ( &inreg, &outreg);
    if ( outreg. x. cflag) / * cflag is nonzero on error * /
    {
        printf ( "Error reading, code == %d\n",outreg. x. ax);
        if ( outreg. x. ax == 5)
            printf ( "file is read only?\n");
        else
            printf ( "file not open?\n");
        return ( -1);
    } / * end error condition * /
    else / * if no error * /
    {
        if ( outreg. x. ax == 0)
            printf ( "end of file reached. \n");
        else if ( outreg. x. ax < size)
        {
            printf ( "Partial file read of %d chars. \n",outreg. x. ax);
        }
    }
```

```
        return( outreg.x.ax);
}
```

0x43:文件属性的读取与设置

服务功能0x43可用于确定文件的属性,也可以用于设置文件属性。不象其它服务功能,功能0x43不用文件句柄来标识文件,而是用文件名。

与其它功能一样,inreg.h.ah 中存有功能服务号。如果想读取文件的属性,把0x0赋给 inreg.h.al。如果要设置文件属性,则把 inreg.h.al 置为0x1。重新设置文件属性时,把新属性置入 inreg.x.cx 中。0x0代表只读,0x2表隐藏文件,0x4表示系统文件,0x20代表档案文件。为指示文件,需要把文件名的起始地址送到 inreg.x.dx。

下列程序示例了调用功能0x43:

```
/* Program illustrating use of DOS service for setting
   or reading file attributes.
*/

#include <stdio.h>
#include <string.h>
#include <dos.h>

#define MAX   STR 100

union REGS inreg, outreg;

int attribute ( char *, int, int);
int attribute ( char *name, int action, int setting)
{
        inreg.h.ah = 0x43;                      /* code for attribute service      */
        inreg.h.al = action;                    /* 0 == get; 1 == set              */
        if ( action) /* if set, specify new attributes(s) */
                inreg.x.cx = setting & 0xff;
        inreg.x.dx = (unsigned) &name[ 0];      /* address of file name            */
        intods ( &inreg, &outreg);              /* call function dispatcher        */
        if ( outreg.x.cflag)                    /* cflag is nonzero on error       */
        {
                printf ( "Error code %d\n", outreg.x.ax);
                switch ( outreg.x.ax)
                {
                        case 0x1:
                                printf ( "invalid request.\n");
                                break;
                        case 0x2:
                                printf ( "file not found.\n");
```

```
                        break;
                case 0x3:
                        printf ( "invalid path. \n");
                        break;
                case 0x5:
                        printf ( "access denied. \n");
                        break;
                default:
                        printf ( "unknown error code. \n". );
            } /* switch on outreg. x. ax */
            return ( -outreg. x. ax);
        } /* end error condition */
        else
        {
            if ( !action)                    /* if just reading attributes      */
                return ( outreg. x. ax);
            else /* if setting and no error */
                return ( setting);
        } /* end if no error */
}

main()
{
        int result, action, setting = 0;
        char info [MAX  STR], name [ MAX  STR];
        printf ( "file name?");
        gets ( name);
        printf ( "Set or get? (g to get) ");
        gets ( info);
        if ( ( info [ 0] == 'g') || ( info [ 0] == 'G'))
            action = 0;
        else
        {
            action = 1;
            printf ( "Setting? (0, 1, 2, 4, 32, or sums) ");
            gets ( info);
            setting = atoi ( info);
        }
        result = attribute ( name, action, setting);
        printf ( "result == %d\n", result);
}
```

0x56:重命名文件

功能0x56用于重命名文件。事实上在重命名文件时,可以把一个文件移到同一磁盘的新

子目录下。

为实现这一功能，在调用 intdos( )前需要提供包含当前文件名的串的起始地址(存在 inreg. x. dx 中)，以及包含新文件名的串的起始地址(存在 inreg. x. di 中)。我们还没有用过变址寄存器 DI，通常用它存贮地址。这个寄存器的名字来源于目的变址(destinationindex)。注意联合 REGS 中包括了成员 di 和 si(源变址寄存器)。

对文件名也有一定限制。第一，在文件名中不能采用匹配符。第二，不能把文件从一个驱动器移到另一个驱动器下，但你可以利用新文件中必要的路径信息，把一文件从一个子目录移到另一个子目录。不能重新命名诸如隐含文件和子目录这样的特殊文件。只能用这个功能重命名一个打开的文件。

如果调用失败，此服务把 outreg. x. cflag 置为非零，且 AX 中存贮了错误码，0x2代表找不到已知文件，0x3表示找不到给定路径，0x5表示拒绝存取(例如用新名字的文件已存在)，0x11表示把文件移到另一个驱动器。

下列程序说明如何调用该功能：

```
/* Function for renaming a file */

#include <stdio.h>
#include <string.h>
#include <dos.h>

#define FALSE 0
#define TRUE 1
#define MAX_STR 100

union REGS inreg, outreg;

int ren_file ( char *, char * );
int ren_file ( char *old_name, char *new_name)
{
        inreg.h.ah = 0x56;                          /* rename service code      */
        /* store locations of old and new file names */
        inreg.x.dx = (unsigned) &old_name [ 0];
        inreg.x.di = (unsigned) &new_name [ 0];
        intdos ( &inreg, &outreg);                  /* call function dispatcher */
        if ( outreg.x.cflag)                        /* cflag is nonzero on error */
        {
                printf ( "Error code %d.\n", outreg.x.ax);
                switch ( outreg.x.ax)
                {
                        case 0x2:
                                printf ( "file not found.\n");
                                break;
```

```
                case 0x3:
                        printf ( "path not found. \n");
                        break;
                case 0x5:
                        printf ( "access denied. \n");
                        break;
                case 0x11:
                        printf ( "invalid drive. \n");
                        break;
                default:
                        printf ( "unknown error. \n");
                        break;
            } /* end switch on outreg. x. ax */
            return ( FALSE);                    /* report failure          */
        } /* end error condition */
        else
            return ( TRUE);                     /* report success          */
}

main ()
{
        char old [ MAX  STR], new [ MAX  STR];
        int result;
        printf ( "Current file name?");
        gets ( old);
        printf ( "New file name? ");
        gets ( new);
        result = ren  file ( old, new);
        if ( result)
            printf ( "File name changed: %s --> %s\n",old, new);
        else
            printf ( "Name not changed. \n");
}
```

2.4　使用 BIOS 中断

到目前为止，我们所看到的中断服务功能都用到 DOS 的功能调度器作为媒介。在本节我们将看到几个直接调用 BIOS 中断的例子。为使用这些服务，要调用运行时间库函数 int86 ()。

利用功能调度器的软中断在不同的硬件上工作原理相似，而直接调用 BIOS 则不然。一些直接调用 BIOS 的中断只能在与 IBM PC，XT 和 AT 机完全兼容的机器上运行。

注意：在调用 BIOS 中断时须非常小心，如果你改变某值时，应肯定所用的值是正确的，

并正确地发出调用。如果稍不留神，就可能把磁盘的有用区域重写。

```
int int86(int nr  of  intrpt, union REGS * in reg,
    union REGS *outreg)
```

运行时间库函数 int86()不经功能调度器而直接调用软中断。int86()有三个参数：中断号以及两个 REGS 联合参数。让我们看以下几个简单的例子。

0x11、0x12：确定设备及内存

BIOS 中断0x11用来确定机器的配置。这个中断的调用过程非常简单，仅用适当的参数调用 int86( )。在调用 int86( )之前无须给任何寄存器赋值。下面调用了 BIOS 中断0x11：

```
int(0x11, &inreg, &outreg)
```

由 outreg. r. ax 返回设备的所有信息。ax 的各位表达有关配置的如下信息：

- 如果有磁盘驱动器，则位0(最右位)为1，否则为0。
- 如果 AT 机带有80287协处理器，则位1为1，在非 AT 机中此位无用。
- 在早期的 IBM PC 系列中，位2和位3表明系统板的内存容量，而在新机种里不使用这个信息。
- 位4和位5表明视频模式。01(0X1)表示40列文本方式(CGA)，10(0X2)表示80列文本方式(CGA)，11(0X3)表示80列文本方式(单色适配 器)，00方式没采用。
- 位6和位7表明磁盘驱动器的数目，00表示有一个驱动器。依次类推，11表示有四个驱动器，位0为0时，表明无磁盘驱动器，则这两位无意义。
- 在 PCjr 机中，位8表示有一个直接存取片，在其它机器不用这一位。
- 位9至位11表示串行口数目。
- 如果接有游戏适配器，则位12为1。
- 如果在 PCjr 机上接有一个串行打印机，则位13为1。在其它机器不用这一位。
- 位14和位15表明可连接的打印机台数。

BIOS 中断0x12用于确定机器上安装的 RAM 的容量。与中断0x11一样，在调用之前，无需设置任何值。其执行结果由 outreg. x. ax 返回。AX 中的值表示了 RAM 的总量，以1K 为单位。例如 AX＝384，表示内存容量为384K。

下列程序示例了 BIOS 中断0x11和0x12的使用。

```
/* Program to illustrate use of interrupts to dete==ine
   hardware configuration and available memory.
*/

#include <stdio.h>
#include <dos.h>

#define MAX  STR 80
#define FALSE 0
#define TRUE 1
union REGS inreg, outreg;
```

```
char *null  str = "";

main ()
{
        int86 ( 0x11, &inreg, &outreg);
        printf ( "AX register == %x\n", outreg. x. ax );
        int86 ( 0x12, &inreg, &outreg);
        printf ( "AX register == %d kbytes available\n", outreg. x. ax);
}
```

0x10：视频中断功能

BIOS 中断0x10很重要，因它实现了视频 I/O。实际上这个中断提供了十多种功能请求。每个请求允许你查找或改变视频 I/O 的某些方面。

例如，请求0xf 用于确定当前视频方式。而请求0x0用于设置当前视频方式，各请求需在一定图形适配器和监视器下工作。在执行中断之前，最好阅读有关资料，查看图形板和监视器的信息，在本节你将看到几种有用的视频函数。使用视频函数完成图形功能的示例，可参见《用 C 编写图形程序》。

BIOS 中断0x10的请求0xf 用于确定当前视频模式。下例说明如何调用这个请求。在调用函数 int86()之前，应把请求号置入 AH，即将 inreg. h. ah 为0xf。下列语句示例了一旦把请求号赋给 AH，应如何调用0x10的请求0xf。

```
int86(0x10,&inreg,&outreg)
```

执行中断后，结果返回在 outreg 的三个字节中，outreg. h. ah 表明显示模式，outreg. h. al 表明当前模式下每行的字符数，即为40还是80outreg. h. bh 表明当前活动视频页，根据视频模式，可以有1个、4个或8个显示页：在图形方式下只有1页，而在80列文本模式下可有4页(0～3)。而在40列的文本模式下，可有8页(0～7)。

下面列出各种视频方式。带有行、列信息的是文本模式，其余为图形模式。用有序数对表示每行的列数和每屏的行数。例如，(180，25)表示每行80列，每屏25行，在图形方式下象素数表示为：

＜水平象素数＊垂直象素数＞

例如，640×200表示每行600象素，每列200象素。

- 0x0：CGA，　中分辨率文本模式，　黑白两色，(40，25)
- 0x1：CGA，　中分辨率文本模式，　最多可用16种颜色，(40，25)
- 0x2：CGA，　高分辨率文本模式，　黑白两色，(80，25)
- 0x3：CGA，　高分辨率文本模式，　最多可用16种颜色，(80，25)
- 0x4：CGA，　中分辨率图形方式，　4种颜色，(320×200)
- 0x5：CGA，　中分辨率图形方式，　黑白两色，(320×200)
- 0x6：CGA，　高分辨率图形方式，　黑白两色，(640×200)
- 0x7：单色适配器，高分辨文本模式，　黑白两色，(80，25)
- 0x8：PCjr，　低分辨率图形方式，　最多可用16种颜色，(160×200)

- 0x9:PCjr, 中分辨率图形方式, 最多可用16种颜色,(320×200)
- 0xa:PCjr, 高分辨率图形方式, 最多可用4种颜色,(640×200)
- 0xb:未使用
- 0xc:未使用
- 0xd:EGA, 中分辨率图形方式, 最多可用16种颜色,(320×200)
- 0xe:EGA, 高分辨率图形方式, 最多可用16种颜色,(640×200)
- 0xf:EGA, 超高分辨率图形方式, 最多可用16种颜色,(640×350)

功能0x0可以改变视频模式,可以把机器设置在不同的视频模式下工作。例如,把 EGA 的中分辨率改为高分辨率,可先把 inreg. h. ah 置为0x0,然后把 inreg. h. al 置为0xe,最后以0x10作为第一个参数调用 int86()。

函数不返回有关调用成功的任何信息,因此需调用功能0xf 检验视频是否已发生改变。

功能请求0x1允许改变光标的形状。与其它字符一样,光标也是由点组成,在 CGA 中每个字符占8行(从0到7),而对于单色显示器,则占用14行。在 CGA 中用末两行,即第6、第7两行,以及在单色显示中也用末两行,即12、13两行作为缺省光标。

下列函数允许重新定义光标的形状和位置。第一个参变量表明光标的顶行。在调用中断之前,先将其赋给 inreg. h. ch,第二个参变量表示光标的底行,且存于 inreg. h. cl 中。在顶行与底行间的所有行都将变亮。例如,参变量分别0和4时,调用函数 set cursor(),则字符框的前五行变亮。

```
/* Program to illustrate how to change cursor size and location */

#include <stdio.h>
#include <string.h>
#include <dos.h>

#define MAX STR 80
#define FALSE 0
#define TRUE 1
#define MONO MAX 13
#define COLOR MAX 7

union REGS inreg, outreg;

char *null str = "";

void new cursor (int, int, int);
void new cursor (int top, int bottom, int max val)
{
        inreg.h.ah = 0x01;                      /* request code          */
        /* bring out of range values within bounds */
        if ((top < 0) || (top > max val))
                top = 0;
```

```
        inreg. h. ch = top;
        /* bring out of range values within bounds */
        if ( (bottom < 0) || ( bottom > max  val;
                inreg. h. cl = bottom;
        int86 ( 0x10, &inreg, &outreg);          /* call video interrupt  */
}

main ()
{
        int curr  mode, top, bottom;
        char info [ 80];
        /* get coordinates for top and bottom of cursor */
        printf ( "Top row of cursor?);
        gets ( info);
        top = atoi ( info);
        printf ( "Bottom row of cursor? ");
        gets ( info);
        bottom = atoi ( info);
        new  cursor ( top, bottom, MONO  MAX);
}
```

功能请求0x2和0x3分别用于设置和读取光标的位置。调用功能请求0x2可以控制光标的位置。在调用这一功能时，除功能请求码外还需传递三个值。

必须给出光标的所在的行。一般来说，屏幕顶行为0行，而底行为第24行。把这个值置入 inreg. h. dh。把列数赋给 inreg. h. dl。其值在0到79之间或0与39之间(当然这些值随文本模式的不同而不同)，且最左列为第0列。

最后你还须设定光标所在的视频页号，并把它送到 inreg. h. bh 中。在图形方式下，这个值必须为0。下例表明如何把光标移到任意位置。

```
/* program to illustrate use of function request
   to change cursor location
*/

#include <stdio. h>
#include <string. h>
#include <dos. h>

#define MAX  STR 80
#define FALSE 0
#define TRUE 1
#define MID  ROW 12                    /* middle value in rows 0 —— 24      */
#define MID  COL 39                    /* appr0ximate middle in cols ) —— 79 */

union REGS inreg, outreg;
```

```
char *null_str = "";
voik move_cursor(int, int);
void test_move_cursor(void);

/* move the cursor to the position specified,
using BIOS interrupt 0x10 (video) for access to request.
*/
void move_cursor(int row, int column)
{
        inreg.h.ah = 0x2;                /* function request code        */
        inreg.h.dh = row;                /* new row position             */
        inreg.h.dl = column;             /* new column position          */
        inreg.h.bh = 0; /* page 0 */
        int 86(0x10, &inreg, &outreg);   /* BIOS interrupt               */
}

void test_move_cursor()
{
        int row = MID_ROW, col = MID_COL,
        index, rand_val, seed;
        char info[MAX_STR];
        /* Get seed for generating random walk */
        printf("seed? ");
        gets(info);
        seed = atoi(info);
        srand(seed);
        /* write each character to screen, at random position */
        for(index = 97; index <= 122; index++)
        {
                move_cursor(row, col);
                printf("%c", index);
                /* decide whether to move up (odd #) or down */
                rand_val = rand();
                if(rand_val % 2)
                        row++;
                else
                        row--;
                /* decide whether to move left (odd #) or right */
                rand_val = rand();
                if(rand_val % 2)
                        col--;
                else
```

```
                col++;
        } /* for index == 'a' through 'z' */
}

main ()
{
        test  move  cursor ();
}
```

功能请求0x3不仅能告诉你光标的位置，而且还得给出光标的形状信息。当给定光标所在的页号后，函数返回光标所在的列与行，还返回光标的顶行和底行。

为指定视频号，把视频赋给 inreg. h. bh。检测 outreg. h. ah 的值来确定光标所在的行，其值在0与24之间。与功能请求0x2一样，执行 int86()后，光标的列号放在 outreg. h. al 中。光标的顶行在寄存器 outreg. h. ch 中，底行在 outreg. h. cl 中。

视频页存贮在内存中，并用于建立屏蔽输出。例如，在 CGA 中有16K 的内存用于显示。视频页数依赖于当前视频模式下显示一屏所需的内存容量。例如，在文本模式下，在屏幕上显示一个字符要占用两个字节。一个字节存贮字符本身，而另一个字节用于存贮属性。

在中分辨文本模式下，显示一屏幕需2K 内存。这就意味着可存贮8页信息。而在(80,25)模式下，只能有四个显示页。因每屏占4KB 内存。

调用功能请求0x5，可以以任意次序存储和显示每页的内容。这个功能请求用于在屏幕上显示给定页的内容，在调用 int86()之前，先把页号赋给 inreg. h. al，当然还得把功能号0x5置入 inreg. h. ah。可用下面的函数设置当前活动视频页：

```
/* Function for setting the currently active video page.
  NOTE: Function assumes dos.h header file and external
  variables, inreg and outreg.
  Function passes row and column position on page selected,
  but does nothing with the information about
  cursor's shape that the also returned by request 0x05.
*/

void set  video  page ( int page, int *row, int *col)
{
        inreg.h.ah = 0x5;
        inreg.h.bh = page;
        int86 ( 0x10, &inreg, &outreg);
        *row = outreg.h.dh;
        *col = outreg.h.dl;
}
```

还有其它几十种 DOS 和 BIOS 功能服务例程，它们都可以直接调用或通过功能调度器调用。本章只打算介绍这方面的入门知识和一些简单的调用例子。读者可以自己试着去实现其它功能。但在用到改变磁盘或内存内容的中断服务时，需特别小心。

2.5 小 结

本章学习了如何使用三个运行时间库函数实现 DOS 或 BIOS 中断。调用这些中断，可以请求各种 DOS 或 BIOS 服务。如：输入输出处理、文件处理、查看系统时间、日期以及其它系统信息控制屏幕等等。

中断允许用户在底层与操作系统和硬件打交道。有时这样做很有用，但应小心，因稍有疏忽就会引起灾难性后果。

第二部分

Turbo Pascal 与其它语言的接口

Turbo Pascal 与汇编语言的接口

再论与 Turbo Pascal 与汇编语言的接口

Turbo Pascal 与 DOS 和 BIOS 的接口

第三章

Turbo Pascal 与汇编语言的接口

首台计算机问世之初，还不存在任何程序设计语言的时候，每个程序都必须以“机器语言”的形式直接输入计算机。机器语言是由代表指令的数码组成的，用机器语言形式来编写和维护程序是极其困难的。因此，汇编语言应运而生。

汇编语言使用的不再是数码，而是助记符，这使得编写和维护程序都容易多了。尽管如此，伴随汇编语言编程的仍是一个冗长乏味且容易出错的过程。即使执行一个简单的任务也需要很多行代码。并且在早期，每种型号的计算机都有其自己的汇编语言，这使得将一个程序从一种计算机转移到另一种计算机成为一件不可能的事。

高级语言，例如 COBOL 和 FORTRAN，是在汇编语言基础上迈前的一大步。因为一条高级语言语句可以完成多条汇编指令的功能，所以程序的编写速度加快了。高级语言的另一优点是用高级语言写的程序只要稍加修正，就可以移到其它机器上运行。因此，随着时间推移，汇编语言的应用范围缩小到一些专用程序，而通用程序除了极少例外，一般都用高级语言编写。

3.1 扩展 Turbo Pascal

作为高级语言，Turbo Pascal 提供了强大的功能和很高的速度。但用汇编语言写的过程运行速度更快且能给用户提供对计算机各个细节方面的控制。幸运的是，可将汇编例程嵌入到程序中，那样可兼有 Turbo Pascal 的逻辑结构和汇编语言的高速，从而扩展 Turbo Pascal 的功能。

将汇编语言嵌入 Turbo Pascal 程序的方式有两种：外部过程方式和嵌入代码方式。外部过程是汇编语言例程，把它汇编成. OBJ 文件并在编译时将其链接到 Turbo Pascal 程序上去。嵌入代码包含直接插入 Turbo Pascal 程序的机器语言指令。Turbo Pascal 不在外部过程和嵌入代码中查错，因此一定要十分小心，使嵌入代码和外部过程经过充分调试。

在使用嵌入代码时，由于得摆弄代表机器指令的数码，因此几乎退化到计算机的最原始时期。编写嵌入代码并非易事，它需要关于汇编语言和 Turbo Pascal 语言两者都有坚实牢固的基础知识。

要使用嵌入代码，得从使用 Inline 编译指令开始，它告诉 Turbo Pascal 去正确解释跟在其后的机器指令。机器语言指令由十六进制数码构成，并在数码前加一个 $ 符，后而跟一斜杠，如下所示：

```
Inline( $ 8B/ $ 46/<i/          ( * MOV AX,I      * )
```

```
    $03/$46/<j/              (* ADD AX,J          *)
    $89/$46/$FE/);           (* MOV [BP-2],AX     *)
```

在此例中，注释部分给出了对应于嵌入代码的汇编助记符。这段嵌入代码将变量 I 移入 AX 寄存器，把 AX 中值加上 J 变量，然后将 AX 中的内容移入堆栈中的一个位置。在代码中虽然不明显，I 和 J 其实都看作定位于堆栈中的值参数。如果有全局变量，要使用另外的嵌入代码。

注意宽度(SIZE)操作符“<”和“>”的用法。由于没有更好叫法，我们姑且将“<”称作字节宽度操作符，将“>”称作字宽度操作符。这两个操作符都用来引用变量，如在前面例子中，变量是 WORD 类型的值参数，但由于变量存放的位置是用相对于 BP 寄存器的单字节位移量来表示的，因此使用了字宽度操作符，使用正确宽度的操作符是至关重要的。绝对确保使用了正确宽度的唯一办法是使用一个调试器，例如 Turbo Debugger，来查看嵌入代码的反汇编形式。

在刚给出的例子中，所有嵌入代码都包含在一个延长成三行的语句中。另一种做法是将每行当作一个独立的嵌入语句来声明，如下所示：

```
Inline($8B/$46/<i/          (* MOV AX,I          *)
Inline($03/$46/<j/          (* ADD AX,J          *)
Inline($89/$46/$FE/);       (* MOV [BP-2],AX     *)
```

虽然这种做法要繁琐些，但用 Turbo Debugger 调试起来也要容易些。如果将三行编码编到一个语句中去，Turbo Debugger 只是将第一行当作反汇编列表的一部分列出来。但如果将每一行编成一条语句，Turbo Debugger 将每一行用反汇编的形式列出来，这使得调试容易得多。

下面清单示例了将两个整数相加的简单嵌入功能。

```
Program TestInline;
Uses CRT;
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Function Sum(i,j : Integer) : Integer;
Begin
Inline($8B/$46/<i/)          (* MOV AX,I          *)
Inline($03/$46/<j/)          (* ADD AX,J          *)
Inline($89/$46/$FE/);        (* MOV [BP-2],AX     *)
End;
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Begin
ClrScr;
Write('1 + 2 = ');
Writeln(Sum(1,2));
Writeln;
Write('Press Enter...');
Readln;
End.
```

注意，函数头与在 Turbo Pascal 中一般函数的头是一样的。函数以 Begin 开始，后跟 In

line 语句和左括号，这告诉 Turbo Pascal 后面跟的是机器码。机器码的每个字节以十六进制格式键入并用斜杠隔开。Inline 语句以右括号结束。注释部分中给出的汇编助记符，对程序不影响，但有助于解释代码在做什么。

写了嵌入过程后，也许有兴趣看看机器级指令是什么样子。下面是函数 Sum 的反汇编代码：

```
PUSH    BP
MOV     BP,SP
SUB     SP,+02

MOV     AS,[BP+08]
ADD     AX,[BP+06]
MOV     [BP-02],AX

MOV     AX,[BP-02]
MOV     SP,BP
POP     BP
RETF    0004
```

清单中间三行代码是嵌入语言。Turbo Pascal 在其前后增加了七条指令。前三行建立过程入口堆栈。这三行中前两行对任何过程或函数调用都是统一的。第三行，SUB SP，+02用于那些返回一字节或两字节函数值的函数调用。换句话说，Turbo Pascal 为该函数在堆栈中保留了两个字节来临时存放其结果。

后面四行将函数结果从堆栈中移到 AX 寄存器，将 SP 和 BP 寄存器恢复初值，然后作一个 FAR 返回，同时从堆栈中弹出参数。

在嵌入过程中返回函数结果这一工作可能是很需技巧的。Turbo Pascal 期望在堆栈的特定位置发现函数结果。因而必须确保在过程结束前将函数结果正确安放在堆栈中。

注意，一个 FAR 返回被用作结束函数调用，因为{F+}编译指令在程序编译时被开启了。可以看出，写一个嵌入过程和函数并非易事。因为不仅需要用机器语言做工作，而且还要知道 Turbo Pascal 在汇编级上是如何工作的。

毕竟，使用外部汇编过程汇编成.OBJ 文件并用 External 指令链接到 Turbo Pascal 上去的做法要容易得多因而效率高。然而，有一种情况，嵌入代码还是必不可少的——使用嵌入指令(Inline Directive)。

3.2 嵌入指令

嵌入指令与嵌入过程或嵌入函数是基本相同的，不同点在于 Turbo Pascal 不为嵌入指令附加代码来建立和清除堆栈。嵌入指令的声明亦大致与嵌入过程或函数相同，不同点在于嵌入指令省去了关键字 Begin 和 End。下面清单示例了嵌入函数和嵌入指令的区别：

```
(* Inline  Function  *)
Function Sum(i,j : Integer) : Integer;
Begin
```

```
Inline($8B/$46/<i/)          (* MOV AX,I          *)
Inline($03/$46/<j/)          (* ADD AX,J          *)
Inline($89/$46/$FE/);        (* MOV [BP-2],AX     *)
End;
(*************************************)
(* Inline Directive   *)
Function SumD(i,j : Integer) : Integer;
Inline ($58/          (* POP AX - moves j into AX *)
        $5B/          (* POP BX - moves i into BX *)
        $03/$C3);     (* ADD AX,BX - sums values  *)
```

当编译这代码时，SumD 函数产生如下指令：

```
POP     AX
POP     BX
ADD     AX,BX
```

可以看到，Turbo Pascal 没有增加一条指令。使用嵌入指令时，所见即所得。然而，要注意，嵌入指令不能用名字去访问变量，而且因为 Turbo Pascal 此时不建立堆栈，因而不能用相对于 BP 的位移量来引用函数参数(当然，如果自己建立堆栈，那是另一回事了)。嵌入指令通过直接将堆栈顶端值弹过寄存器来访问参数。

如上例所示那样，用嵌入指令给出的过程和函数是不好编程的，一般说来，嵌入指令用于完成特定功能的短代码是最佳的。例如，下面的嵌入指令将 SP 的值存放在 WORD 型变量 WordVar 中。

```
Inline($89/$26/WordVar);        (* MOV WordVar,SP *)
```

当然，勿须频繁取得 SP 的值，但如果确要取 SP 值时，最有效的方法是使用嵌入指令。随着编程任务不断高级化，将发现这种特殊编程方法的许多好处。

3.3　外部过程

外部过程是用汇编语言写出来，汇编成.OBJ 文件并在编译时与 Turbo Pascal 链接起来的例程。同嵌入代码相比，汇编例程有许多优点。汇编代码在编写、解释和维护方面比起嵌入代码中的机器语言都要容易处理得多。事实上，程序员写嵌入过程时，通常先将其写成一个外部过程然后转换成嵌入代码。既然嵌入过程比起外部过程来无任何主要优点，就没多大理由去走那多余的一步了。

更重要的是，外部例程能直接访问 Turbo Pascal 全局变量、局部变量、参数、过程和函数。换句话说，在汇编中访问 Turbo Pascal 数据和函数就如在 Turbo Pascal 中一样容易。这些强有力的特征使得编写外部汇编例程对 Turbo Pascal 来说是一种很有吸引力的选择，尤其是在须要相当快速度时。

3.3.1　外部函数

在前面，见到了将两个整数相加的函数的嵌入代码。下面用外部例程方式给出的汇编程序清单完成同样的功能：

```
CODE    SEGMENT BYTE PUBLIC
        ASSUME CS:CODE

PUBLIC SUM
SUM     PROC  FAR
        PUSH  BP
        MOV   BP,SP
        MOV   AX,[BP+08]
        ADD   AX,[BP+06]
        POP   BP
        RET   4
SUM     ENDP
CODE    ENDS
        END
```

程序虽然短小，但例程说明了汇编子程序的基本要素。它以定义 CODE 段和说明 Sum 作为 PUBLIC 过程开始(虽然 Assembler 允许使用任意代码段名，但 Turbo Pascal 只使用两个：CODE 和 CSEG)。PUBLIC 很重要，因为它使被说明的例程可被另外的程序模块使用。如果不用 PUBLIC 说明，则 Turbo Pascal 不能存取。

其余汇编代码定义 Sum 例程和任何其它外部过程一样，过程以保存 BP、设置堆栈指针开始。随后的两个语句从堆栈取得参数相加，把结果放于 AX 寄存器。对于返回标量的函数，比如字节、整型、字等等，Turbo Pascal 规定把结果置于 AX 寄存器。过程以 RET 4指令结束，值4是指 Turbo Pascal 在调用例程时压进堆栈的4字节。当过程返回时，必须确保从堆栈移去参数。

要使外部子程序在程序中可用，必须先汇编成. OBJ 文件。假设在本例中的汇编代码在 ADD. ASM 中，其目标文件为 ADD. OBJ。在 Turbo Pascal 中，如下说明外部子程序：

```
{$F+}
{$L ADD}
Function Sum(i,j:Integer) : Integer; External;
```

第一个语句是强制远程调用的编译指令。因为对应的外部例程以 FAR PROC 说明，以 RETF 命令结束，在 Turbo Pascal 中必须作为远程调用对待。下一行编译指令使 TurboPascal 从 ADD. OBJ 中查找外部子程序，因为过程 Sum 包含在 ADD. OBJ 中，连接器可在其中定位外部引用。

最后一条语句是用 External 修饰的函数说明，使 Turbo Pascal 从目标文件中取得过程的代码。完整的程序如下：

```
Program TestExternal;
Uses CRT;
(*********************************)
{$F+}
{$L ADD}
Function Sum(i,j : Integer) : Integer; External;
(*********************************)
```

```
Begin
ClrScr;
Write(' 1 + 2 = ');
WriteLn(Sum(1,2));
WriteLn;
Write('Press ENTER...');
ReaedLn;
End.
```

3.3.2 使用全程数据和过程

Turbo Pascal 的汇编接口的最有价值的特点之一是能在汇编子程序中使用全程过程、变量和类型常量。因为在 Pascal 和汇编之间穿插代码更加灵活，因而编写的代码也更加易于维护。

为说明汇编例程如何使用全程数据、过程，考虑把字符串变成大写的例程。例程接收一字符串参数，并将其变成大写。本程序指出有限制希望不要传递太多的字符，以免死机。汇编例程须要知道字符数的最大值，以及在检测到错误时如何转移控制权。

下面的汇编代码能符合这些标准。Turbo Pascal 全局变量 MaxStrLen 存放表示错误的数字。当然，在过程中设置一数，或把数作为参数传递，但这样的程序冗长而且难于维护。

```
DATA        SEGMENT BYTE PUBLIC
            EXTRN MaxStrLen : BYTE;
DATE        ENDS

CODE        SEGMENT BYTE PUBLIC
            ASSUME CS:CODE, DS:DATA
            EXTRN    StrLenError : FAR
            PUBLIC   UPCASESTR

UPCASESTR       PROC    FAR
            PUSH    BP
            MOV     BP,SP
            LDS     SI,[BP+6]           ; Move string length byte
            MOV     AL,BYTE PTR[SI]     ; into AL.

            XOR     CX,CX               ; Move string length
            MOV     CL,AL               ; into CL
            CMP     CL,MaxStrLen        ; If the string is less than
            JL      LOOP1               ; the maximum length, goon.
            CALL    StrLenError         ; If not, call StrLenError

LOOP1 :     INC     SI                  ; Point to character.
            MOV     AL,BYTE PTR [SI]    ; Load char into AL.
            CMP     AL,97               ; Compare to 'a'.
```

```
        JB      NOTLOW                  ; If lower, jump.
        CMP     AL,122                  ; Compare to 'z'.
        JA      NOTLOW                  ; If higher, jump.
        SUB     AL,32                   ; Uppercase char.
        MOV     BYTE PTR [SI],AL        ; Move char to string.

NOTLOW: LOOP    LOOP1
        POP     BP
        RET     2

UPCASESSTR      ENDP
CODE    ENDS
        END
```

CODE 段有些地方引用了一个叫 StrLenError 的外部过程，这也是 Turbo Pascal 的全局过程，当非法长度字符串被传给汇编例程时，它就被调用。过程开始后，首先查看字符串参数的字节长度。如果长度大于或等于测试值(MaxStrLen)，就将控制权转交给 Turbo Pascal 过程 StrLenError，它打印出错误信息并中止执行。这里给出的程序显示出该例程是如何被测试的：

```
{$F+}
Program TestAsm ;
Uses CRT ;
Const
      MaxStrLen : Byte = 100 ;
Var
      s : String ;

{$L UPCASE}
Procedure UpCaseStr(Var s : String) ; external ;
(*************************************)
Procedure StrLenError
Begin
WriteLn('String length error encountered.');
WriteLn ;
Write(' Press ENTER...') ;
ReadLn;
Halt;
End;
(*************************************)
Begin
ClrScr;
s := 'abcdef';
WriteLn('Lower case = ', s) ;
```

```
UpCaseStr(s) ;
Writeln('Upper Case = ',s);
WriteLn(s) ;
WriteLn;
WriteLn('Force a string-length error condition.');
WriteLn ;
Write('Press ENTER...');
ReadLn;
s[0] := Chr(101);
UpCaseStr(s);
End.
```

因为外部过程被定义成一个 FAR 调用，一定要在编译时让 F 编译指令开启。如果没有开启 F 编译指令，程序在从外部过程返回时，将发生崩溃。

3.3.3 使用 Turbo Assembler

在前面外部例程的例子中，所有对参数的引用都是使用了一个相对于基指针的位移量（例如：[BP+6]）。使用位移量时间开销大且易出纰漏。幸好，Borland 在 Turbo Assembler 中引入了一个解决方案。

Turbo Assembler 是一个十分成熟且高性能的汇编器，增加了一些功能使得汇编与 Turbo Pascal 的链接更容易。考虑下面的汇编例程例子——一个叫 Switch 的过程，它交换两个变量的值。该过程要求三个参数——两个指针（分别指向要交换的变量）和一个 WORD 参数。最后一个参数表明要交换参数的宽度（例如，整数的宽度是2）。正常情况下，汇编例程要定义数据段和代码段，并包括一些建立堆栈的入口和出口以及弹出正确个数参数的代码。通过 Turbo Assembler，许多工作已为用户做好。

```
.MODEL TPASCAL
.DATA
BUFFER DB 256 DUP(?)                    ; Buffer to hold value during switch
.CODE
PUBLIC SWITCH

Switch PROC FAR A : DWORD, B : DWORD , Dsize : WORD
; MOVE A INTO BUFFER
    LDS  SI,A                           ; Load address of A into DS:SI
    LEA  DI,BUFFER                      ; Load address of Buffer in ES:DI
    MOV  CX,Dsize                       ; Move Dsize into CX
; Move contents of A into Buffer
    REP    MOVS BYTE PTR  ES:[DI],DS:SI

; MOVE B INTO A
    LDS  SI,B                           ; Load address of B into DS:[SI]
    LES  DI,A                           ; Load address of A into ES:DI
```

```
        MOV   CX,Dsize                    ; Mov Dsize into CX
; Move contents of B into A
        REP   MOVS BYTE PTR ES:[DI],DS:[SI]

; MOVE BURRER INTO B
        LEA   SI,BUFFER                   ; Load address of Buffer into DS:SI
        LES   DI,B                        ; Load address of B in ES:DI
        MOV   CX,Dsize                    ; Move Dsize into CX
; Move contents of Buffer into B
        REP   MOVS BYTE PTR ES:[DI],DS:[SI]
        RET

SWITCH ENDP
        END
```

汇编例程的第一行.MODEL TPASCAL,告诉 Turbo Assembler 产生与 Turbo Pascal 链接的代码。.DATA 和.CODE 命令取代了其它汇编器所需的繁琐的伪操作码。过程原型

Switch PROC FAR a : DWORD , b: DWORD , Dsize : WORD

告诉 Turbo Assembler 过程名是 Switch,是一个 FAR 调用,过程带三个变量参数——两个地址(a 和 b)和一个数值参数(Dsize)。

注意前面给出汇编程序清单中的过程不包括任何入口或出口代码。事实上,结尾处的 RET 指令甚至不用说明从堆栈弹出多少个参数——Turbo Assembler 填充正确的数字。不仅如此,Turbo Assembler 还能使用 Turbo Pascal 的名字来使用参数和全局变量。可以看出,用 Turbo Assembler 来写外部过程比使用标准汇编器要容易得多。

下面给出的程序告诉用户如何使用上面给出的汇编例程。L 编译指令命名了要链接的目标文件名。程序包含两个交换过程:外部例程 Switch 和 Turbo Pascal 例程 Switch1。Turbo Pascal 过程用来作一个比较,令人惊奇的是,它几乎与汇编例程是一样快的——这是 Turbo Pascal 高效的一个证明。

```
{F+}
Program SwitchTest
Uses CRT;
Var
      a,b : Integer ;
      c,d : Real    ;
      e,f : String  ;
{$L SWITCH }
Procedure Switch(Var a,b;
                 c : Integer) ; External ;
Procedure Switch1(Var a,b ;
                 c : Integer) ;
Var
      Buf : String ;
```

```
Begin
Move(a,Buf,c) ;
Move(b,a,c) ;
Move(Buf,b,c) ;
End;

Begin
ClrScr ;
a := 1 ;
b := 2 ;
c := 12.34 ;
d := 45.67 ;
e := 'ABCDEFG' ;
f := 'HIJKLMN' ;

WriteLn('Using assembler') ;
WriteLn ;
WriteLn(a,' > ',b) ;
Switch(a,b,SizeOf(a)) ;
WriteLn(a,' < ',b) ;
WriteLn ;

WriteLn(c:0:2,' > ',d:0:2) ;
Switch(c,d,SizeOf(c))  ;
WriteLn(c:0:2,' < ',d:0:2) ;
WriteLn ;

WriteLn(e,' > ',f) ;
Switch(c,d,SizeOf(e))  ;
WriteLn(e,' < ',f) ;
WriteLn ;
WriteLn ;

a := 1 ;
b := 2 ;
c := 12.34 ;
d := 45.67 ;
e := 'ABCDEFG' ;
f := 'HIJKLMN' ;

WriteLn('Using Pascal') ;
WriteLn ;
WriteLn(a,' > ',b) ;
```

```
Switch(a,b,SizeOf(a)) ;
WriteLn(a,' < ',b) ;
WriteLn ;
WriteLn(c:0:2,' > ',d:0:2) ;
Switch(c,d,SizeOf(c))  ;
WriteLn(c:0:2,' < ',d:0:2) ;
WriteLn ;

WriteLn(e,' > ',f) ;
Switch(c,d,SizeOf(e))  ;
WriteLn(e,' < ',f) ;
WriteLn ;
WriteLn ;

ReadLn ;
End.
```

这个例子只是触及 Turbo Assembler 强力的“皮毛”。如果要认真地为 Turbo Pascal 写外部例程，必须充分深入 Turbo Assembler 的“肌里”，考虑它能提供的优势。

3.4 嵌入代码与外部过程的比较

将过程写成外部过程形式还是嵌入代码形式主要取决于个人的风格口味。嵌入过程一般说来要快些且与程序一起编译，但较难编写与维护。外部过程写起来简单些，因为勿须将汇编代码翻译成机器码。外部过程还易于归档和管理。

一般说来，最好尽量少地使用嵌入代码，但有时嵌入代码不仅被乐于采用而且是必需的。例如，当想在不单独创建另外过程或函数的情况下在程序中植入代码时，就得使用嵌入代码。

3.5 使用 Turbo Debugger

没有一个调试器就无法用汇编语言编程。不管是使用老牌的 DEBUG.COM 或者是一个更高级的程序，都得有单步追踪功能。Borland 的 Turbo Debugger 对于写汇编例程的程序员来说是一个无价之宝。有了 Turbo Debugger，你可以用一次执行一条机器指令的方式来执行程序，从而可以准确地看出每个寄存器和内存中每一位置所发生的一切。还可以学会 Turbo Pascal 内部工作情况的许多内容——参数是如何传送到堆栈上去；算术运算是如何执行的；在某一时间点寄存器所存的内容是什么，等等。

为了观察 Turbo Debugger 是如何工作的，让我们使用本章前面给出的包含嵌入代码的程序：

```
{$D+,L+}
Program TestInline;
Uses CRT;
```

```
(*************************************)
Function Sum(i,j : Integer) : Integer;
Begin
Inline($8B/$46/<i/)          (* MOV AX,I        *)
Inline($03/$46/<j/)          (* MOV AX,J        *)
Inline($89/$46/$FE/);        (* MOV [BP-2],AX   *)
End;
(*************************************)
Begin
ClrScr;
Write('1 + 2 = ');
WriteLn(Sum(1,2));
WriteLn;
Write('Press Enter...');
ReadLn;
End.
```

在编译该程序时，一定要确保 Standalone Debugging 选项开启，并且 D 和 L 编译指令被开启。这样做给了 Turbo Pascal 信息让它将程序代码匹配到它可执行的代码上去。

当将该程序编译好放在磁盘上后，键入 Turbo Debugger 命令后面跟一个要检查的程序的名字，这就可进入 Turbo Debugger。若程序文件名是 TEST.PAS，则命令为：

C> TD TEST

Turbo Debugger 将 TEST.EXE 文件装入，并读入 TEST.PAS 和 TEST.MAP 文件。Turbo Debugger 由于使用了附加在.EXE 文件后面的调试信息，它可以在显示机器语言指令的同时显示一行源代码。图3.1显示了 TEST.PAS 程序在 Turbo Debugger 中的面貌。注意一个箭头指向了程序中的第一个 Begin 语句。当对程序进行追踪时，该箭头将一直指向下一个要执行的语句。

可以使用 F7和 F8键对整个程序进行追踪。两键都是一次执行一行代码，但 F8键跳过函数调用，而 F7追踪到函数调用内部。使用数字小键盘上的光标键，可以上下翻滚程序来查看程序各部分。如果上滚一些语句并同时按下 F4，Turbo Debugger 就执行当前光标位置以前的所有语句。

为了最好地使用 Turbo Debugger，得好好对待机器级指令，这是容易做到的——只须敲 F10激活主菜单，选取 View 选择项，接着按 C 选择 CPU 项(见图3.2)。这就打开了 CPU 窗口，它包含四个“窗格”(Pane)。左上角窗格显示反汇编码(图3.3)，它右边是寄存器框，显示的是 CPU 寄存器内容，右下角窗格追踪堆栈，左下角窗格显示一小片 RAM。主要看的是代码和寄存器窗格。注意一下包含下面这条 Pascal 的代码窗格：

TESTINLINE.20: WriteLn(Sum(1,2)) ;

这是源程序第20行的代码，它写出一个函数结果。在这一行下面，Turbo Debugger 列出了实现它的机器指令：

```
cs:0059  BF5001      mov    di,0150
cs:005C  1E          mov    ds
cs:005D  57          push   di
```

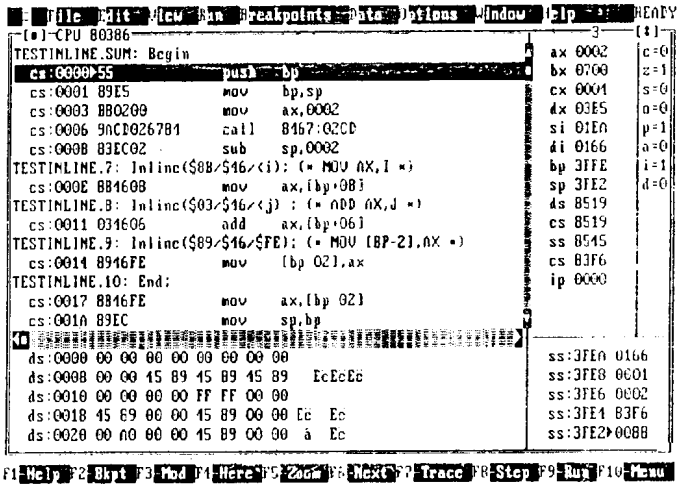


图3.1　Turbo Debugger 上的 Pascal 程序

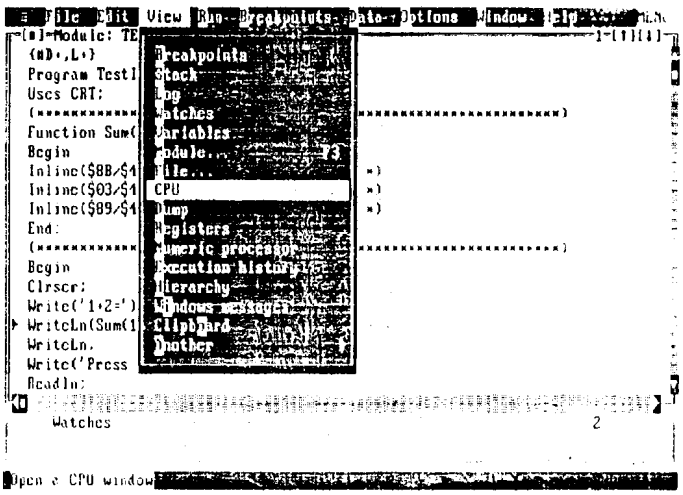


图3.2 从 View 菜单中选取 CPU 屏幕

```
cs:005E  B80100          mov     ax,0001
cs:0061  50              push    ax
```

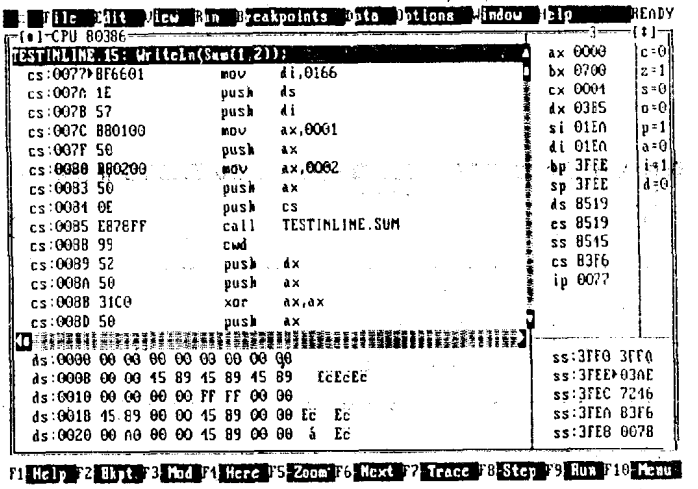


图3.3　Turbo Debugger 的 CPU 屏幕

```
cs:0062  B80200          mov     ax,0002
cs:0065  50              push    ax
cs:0066  E897FF          call    TESTINLINE.SUM
cs:0069  99              cwd
cs:006A  52              push    dx
cs:006B  50              push    ax
cs:006C  31C0            xor     ax,ax
cs:006E  50              push    ax      0
cs:006F  9A7807264C      call    4c26:0778
```

每一行包括三部分——指令在代码中的位置(例如cs:0059)、十六进制形式的机器语言指令(例如:BF5001)和汇编码(例如:mov　di,0150)。可以看出,一行源代码产生了14条机器语言指令,其中两条指令是对其它例程的调用,在代码中间是对过程 Sum 的调用,它包括嵌入代码,按 F7键,直到 Turbo Debugger 进入 Sum 过程。屏幕将呈现图3.4的样子。注意过程开头是:

```
cs:0000  55              push    bp
cs:0001  89E5            mov     bp,sp
cs:0003  83EC02          sub     sp,0002
```

Turbo Pascal 将这些代码加到嵌入代码前以便在过程运行前建立堆栈。后面三行包含嵌入代码:

```
TESTINLINE.10: Inline($8B/$46/<i);        (* MOV AX,I          *)
        cs:0006      8B4606mov   ax,[bp+6]
```

```
TESTINLINE.11: Inline($03/$46/<j);      (* MOV AX,J    .      *)
    cs:0009    034604add   ax,[bp+04]
TESTINLINE.12: Inline($89/$46/$FE/);    (* MOV [BP-2],AX  *)
    cs:000C    8946FEmov   [bp-02],ax
```

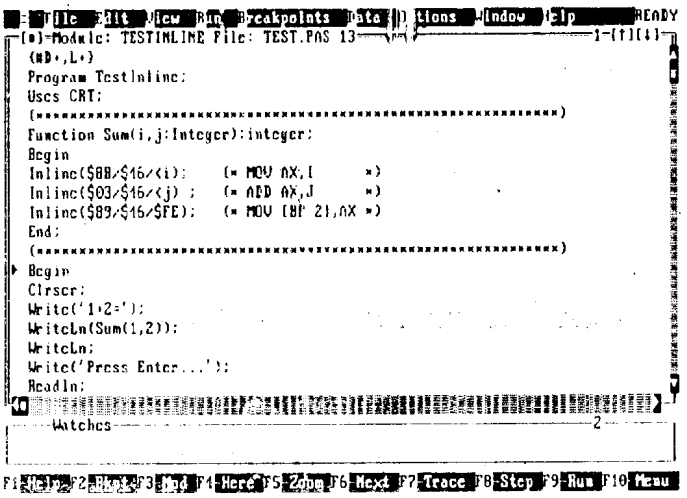


图3.4　Turbo Debugger 中的嵌入代码

注意嵌入代码与反汇编行的机器指令码的比较。通过使用 Turbo Debugger，能检查嵌入代码以保证它将做所期望的事情。

过程结束是将函数结果送 AX 寄存器，清除堆栈并返回入口点：

```
cs:000F  8B46FE        mov     ax,[bp-02]
cs:0012  89EC          mov     sp,bp
cs:0014  5D            pop     bp
cs:0015  c2400         ret     0004
```

嵌入代码是特别难调试的，因为它除了数码化的机器指令之外，别无其它。Turbo Debugger 能看到嵌入代码语句所表示的汇编指令，并逐个执行每条语句以便能定位问题区域。但 Turbo Debugger 并不仅限于调试嵌入代码，它对于外部例程和直接的 Pascal 代码都同样有效。它也是学习汇编语言编程的一个重要方法。使用 Turbo Debugger 可以领会好程序的精妙之处。

Turbo Pascal 产生极高效的代码，但有时想要或必须做得更好。汇编例程和嵌入代码有助于使程序速度快且功能强，外部例程比嵌入代码更易于开发和维护。不论使用哪种方法，Turbo Assembler 将使汇编编程简易，而 Turbo Debugger 有助于追踪和发现汇编代码中的错误。

第四章 再论与 Turbo Pascal 和汇编语言的接口

Turbo Assembler 提供了广泛的、强有力的工具以使用户能将汇编代码嵌入到 Turbo Pascal 程序中。本章将对用户充分运用这些性能(包括许多例子和"机制")所须了解的每一个问题进行介绍。

为什么要交叉使用 Turbo Assembler 和 Turbo Pascal 呢?用户编写的绝大多数程序完全可以用 Turbo Pascal 编写。与其它 Pascal 不同,Turbo Pascal 实际上可以让用户通过 Port[]、Mem[]、MemW[]以及 MemL[]数组直接访问机器资源,用户还可以通过使用 Intr()和 MsDos()函数,使用 BIOS 和操作系统调用。

那么,用户为什么想交叉使用汇编语言和 Turbo Pascal 呢?最可能的两个原因是:为了执行几个 Turbo Pascal 不能直接进行的操作,获得只有汇编语言才能提供的速度(Turbo Pascal 本身速度之所以如此之快,就是因为它是用汇编语言写的)。本章将介绍怎样以及何时利用强有力的汇编语言和 Turbo Pascal 的交叉使用技术。

注意:除非特别声明了版本号,一般所提到的 Turbo Pascal 都是指4.0版或更高版本。

4.1 Turbo Pascal 内存映象

在用户编写与 Turbo Pascal 一起运行的汇编语言代码前,理解编译器是怎样将信息安排在内存中对用户来说相当重要。Turbo Pascal 的存储模型包含中模型和大模型两方面,它们在第四章中介绍过。有一个全局数据段,它允许 DS 对全局变量和类型常量进行快速存取。然而,每个单元都有其自身的代码段,并且堆可占有所有可用内存。Turbo Pascal 中的地址在编译扫描时总是被设置为远指针(32位),因而能引用内存中任何地方的对象。

一个 Turbo Pascal 程序的内存映象如图4.1所示。

4.1.1 程序段前缀

程序段前缀(PSP)是一个在程序加载时由 MS-DOS 创建的256字节的区域。其中包含用于调用程序的命令行参数信息、可用内存的总量以及 DOS 环境。所谓 DOS 环境是指 DOS 所用的串变量表。

在 Turbo Pascal 3.0中,PSP 段地址和其他代码的段地址是一样的。然而,在 Turbo Pascal 4.0以及更高版本中不再如此,主程序、主程序所用单元以及运行时间库都占有不同的段。因此,Turbo Pascal 将 PSP 的段地址存于一个叫做 PrefixSeg 的预先说明的全局变量中,

这使得用户可以访问 PSP 信息。

4.1.2　代码段

每个 Turbo Pascal 程序都有至少两个代码段：一个用于主程序代码，一个用于运行库。另外，每个单元的代码占有一个独立的段。因为每个代码段最大可达64KB，用户程序可占用任意多内存（当然要受到机器可用资源的限制）。程序设计者过去设计超过64KB 的程序要使用覆盖技术，现在他可以将所有代码保持在内存中以获得快速执行功能。从 Turbo Assembler 角度来看，链接进汇编语言模块的代码段具有名字 CODE 或 CSEG。

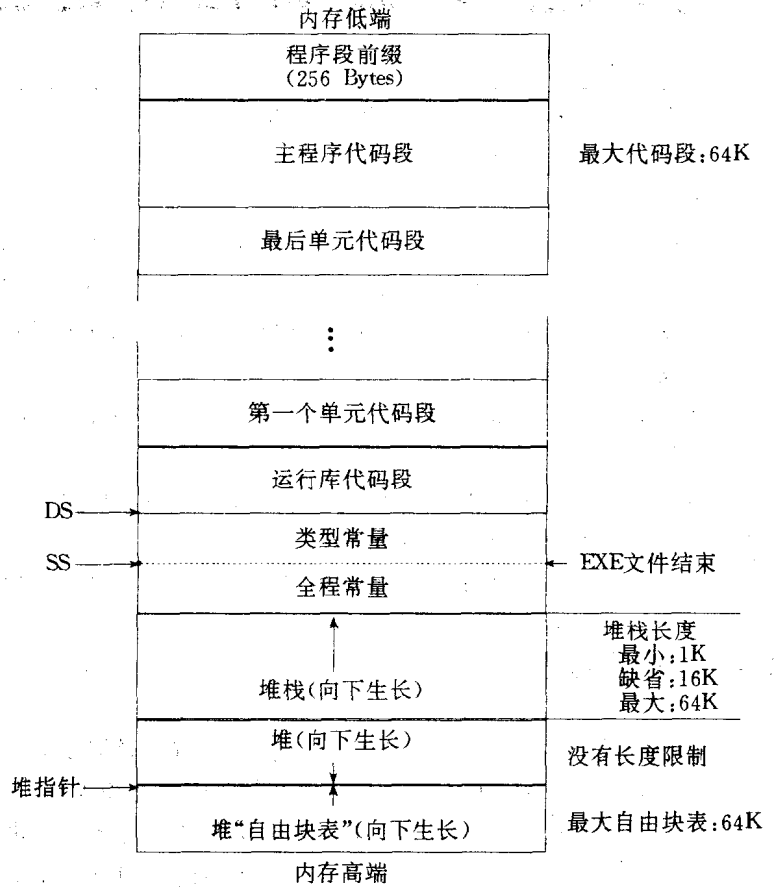


图4.1　Turbo Pascal 5.0程序的内存映象

4.1.3　全局数据段

Turbo Pascal 的全局数据段跟在运行库代码段后。它可包含至多64KB 的初始化和非初始化数据：类型常量和全局变量。与 Turbo Pascal 3.0中一样，类型常量实际上并不完全是常量，而是在程序装入时以一个预初始化值作为开始的变量。但是，与 Turbo Pascal3.0不同的是，Turbo Pascal 4.0不是把类型变量放在代码段。Turbo Pascal 4.0将类型变量放在全局数据段，这样就可用比 Turbo Pascal 3.0更快的速度访问这些类型变量。当从 Turbo Assembler 模块引用时，全局数据段具有名字 DATA 或 DSEG。

4.1.4 堆 栈

在 Turbo Pascal 4.0及其更高版本中，全局数据段在堆栈之上。注意这种安排与 Turbo Pascal 3.0中不同。堆栈和堆不是相向生长的，而是为堆栈分配了一段固定大小的内存。缺省大小为16K，它远远大于绝大多数程序之所需。然而，用户还可声明堆栈大小小到1K(用于短程序)或大到64K(用于有大量递归的程序)。堆栈和堆的大小可由 $M 编译伪指令来选择。

象在大多数80x86程序中一样，堆栈指针从堆栈段的顶部开始向下生长。每一个过程或函数调用时，Turbo Pascal 通常进行检查以确认堆栈没有用完。这种检查可使用{$S-}编译伪指令将其关闭。

4.1.5 堆

在 Turbo Pascal 内存映象的顶部是堆。缺省时，堆占有除分配给代码段、数据段和堆栈段外的所有内存，但是 $M 伪指令可用来限制堆的最大字节数(这也可用于防止当一个最小量的堆空间不够一个程序运行的情况发生)。

每当用户进行一个 New()和 GetMem()操作，就在堆中从底部开始动态分配存贮区。当用户进行一个 Dispose。Release 或 FreeMem 操作，空间就被释放。当使用 Dispose 和 FreeMem 时，Turbo Pascal 4.0利用一个叫做自由块表(free list)的数据结构来保持堆中可用区的信息。自由块表(最大可达64K)从堆区的最顶部向下生长。

4.2 Turbo Pascal 中寄存器的用法

与 Turbo Pascal 3.0相似，Turbo Pascal 4.0对寄存器的使用施加最小的限制。当调用一个函数或过程时，只有三个寄存器的值需要保存，它们是堆栈段寄存器(SS)。数据段寄存器(DS)和段基址指针(BP)。DS 指向全局数据段(所谓的 DATA)，SS 指向堆栈段。BP 被每个过程或函数用来引用它的活动记录(activation record)——它用于参数、局部变量和临时存储的堆栈空间。所有子程序在退出之前都必须调整堆栈指针(SP)，这样，参数就不再保存在堆栈中了。

4.3 近调用还是远调用

因为一个 Turbo Pascal 程序包含多个代码段，它混合使用近调用和远调用来访问过程和函数。这两者有何区别呢?一个近调用只能用于访问一个在同一段中被调用的驻留在相同代码段中的子程序，而一个远调用可访问内存任何地方的子程序。然而，这种灵活性会带来一个小小的损失：一个远调用比一个近调用占用更多一些的时间和空间。

用户的 Turbo Pascal 程序中的每个子程序都能写成近调用或远调用中的一种(不论是由编译写的或是由用户自己写的)。用户应该选择哪一种调用呢?在一个单元的接口(interface)部分说明的子程序总是远调用，以便从其它单元能调用它们。但是在主程序中说明的子程序以及在一个单元的实现(implementation)部分说明的子程序一般是近调用任何子程序可以用{$F+}编译伪指令强制为远调用。

当编写与 Turbo Pascal 接口的汇编语言程序时，用户必须检查以确信自己的程序具有正确的“距离”。如果用户说明一 PROC 在汇编语言中是近调用的，而相应的外部(external)子程序说明成远调用的话，Turbo Pascal 将不报告出错。

4.4 与 Turbo Pascal 共享信息

4.4.1 $L 编译伪指令和外部子程序

{$L}编译伪指令和外部(external)子程序声明是交叉使用 Turbo Assembler 和 Turbo Pascal 的两个关键点。伪指令{$L MYFILE.OBJ}引导 Turbo Pascal 查看 MYFILE.OBJ，并将其链接到用户的 Turbo Pascal 程序中(MYFILE.OBJ 是一标准 MS－DOS 可链接的目标代码格式的文件)。如果{$L}伪指令中的文件名未给出扩展名，就假设为.OBJ。

用户如果要使每个 Turbo Assembler 过程或函数在 Turbo Pascal 程序中是可见的，就必须将其说明为一个 PUBLIC 符号，且在 Turbo Pascal 程序中必须有一个相应的外部说明。

Turbo Pascal 中外部(external)过程或函数说明的语法与提前引用(forward)说明的语法非常相似。

```
procedure AsmProc(a : integer; b : real); external;
function  AsmFunc(c : Word; d : Byte) ; external;
```

这些说明与在下列用户汇编程序中的说明相一致。

```
CODE      SEGMENT BYTE PUBLIC
AsmProc   PROC  NEAR
          PUBLIC AsmProc
          .
          .
          .
AsmProc   ENDP
AsmFunc   PROC FAR
          PUBLIC FAR
          .
          .
          .
AsmFunc   ENDP
CODE ENDS
```

一个 Turbo Pascal 外部(external)过程说明必须在程序或单元的最外层。这就是说，它不可嵌套在另一个子程序中说明。要在任何其它层说明一个外部(external)子程序的企图都将引出一个编译错误。

Turbo Pascal 不检查确认以近调用或远调用属性声明的那些 PROC 是否与用户 Turbo Pascal 程序中相应的近调用或远调用过程一致。事实上，它甚至不检查公共符号 AsmProc 和 AsmFunc 是否是 PROC 的名字。应该由用户自己确认汇编语言和 Pascal 的说明是一致的。

4.4.2 PUBLIC 伪指令:使 Turbo Pascal 可利用 Turbo Assembler 的信息

只有在汇编语言模块中被说明为 PUBLIC 的符号对 Turbo Pascal 是可见的。符号是唯一可由汇编语言输出到 Turbo Pascal 的对象。更进一步说,每个被说明为 PUBLIC 的符号必须在 Turbo Pascal 程序中有一个相应的过程或函数说明,否则编译器将报告一个错误。一个公共符号不必是一个 PROC 说明的一部分。就 Turbo Pascal 而言

```
AsmLabel    PROC FAR
            PUBILC Bar
```

和

```
AsmLabel:
            PUBLIC Bar
```

是等价的。

4.4.3 EXTRN 伪指令:使 Turbo Assembler 可利用 Turbo Pascal 的信息

Turbo Assembler 模块可以访问与之相联的程序或单元最外层声明的任何 Turbo Pascal 过程、函数、变量或类型常量(注意这包括与该模块有关的{$L}编译指令和 external 之后声明的变量)。Turbo Pascal 符号和普通常量对汇编语言是不可见的。

假设用户的 Turbo Pascal 程序说明了以下全局变量:

```
Var
  a : byte ;
  b : word ;
  c : shortInt ;
  d : integer ;
  e : real;
  f : single;
  g : double;
  h : extended;
  i : comp;
  j : pointer;
```

用户可以在汇编语言程序中用 EXTRN 说明访问这些变量:

```
EXTRN A : BYTE        ;1 byte
EXTRN B : WORD        ;2 bytes
EXTRN C : BYTE        ;Assembly language treats signed and unsigned alike
EXTRN D : WORD        ;Ditto
EXTRN E : FWORD       ;6-byte software real
EXTRN F : DWORD       ;4-byte IEEE floating point
EXTRN G : QWORD       ;8-byte IEEE double-precision floating point
EXTRN H : TBYTE       ;10-byte IEEE temporary floating point
EXTRN I : QWORD       ;8087 8-byte signed integer
```

```
EXTRN J : DWORD          ;Turbo Pascal pointer
```

用户可以使用类似方法访问 Turbo Pascal 过程和函数(包括库函数)。假设用户有一个 Turbo Pascal 单元如下所示：

```
unit Sample;
  { Sample unit that defines several pascal procedures that are
    called form an assembly language procedure. }
  interface
  procedure TestSample;
  procedure PublicProc;          { Must be far since it is visible outside }
  implementation
  var
    A : word;
  procedure AsmProc; external;
  {$L ASMPROC.OBJ}
  procedure PublicProc;
    begin  { PublicProc }
      Writeln('In PublicProc');
    end;   { PublicProc }
  procedure NearProc;            { Must be near  }
    begin  { NearProc }
      Writeln('In NearProc');
    end;   { NearProc }
  {$F+}
  procedure FarProc;             { Must be far due to compiler directive  }
    begin  { FarProc }
      Writeln('In FarProc');
    end;   { FarProc }
  {$F-}
  procedure TestSample;
    begin  { TestSample }
      Writeln('In TestSample ');
      A := 10;
      Writeln('Value of A before ASMPROC = ',A);
      AsmProc;
      Writeln('Value of A before ASMPROC = ',A);
    end; { TestSample }
  end.
```

AsmProc 子程序可以以如下方式使用 EXTRN 伪指令调用 PublicProc、NearProc 或 FarProc 子程序：

```
DATA        SEGMENT WORD PUBLIC
            ASSUME DS:DATA
            EXTRN A:WORD                    ;variable form the unit
```

```
DATA    ENDS
CODE    SEGMENT BYTE PUBLIC
        ASSUME CS:CODE
        EXTRN PublicProc : FAR          ;far procedure (exported by the unit)
        EXTRN NearProc : Near           ;near procedure (local to unit)
        EXTRN FarProc : FAR             ;far procedure (local but forced far)
AsmProc PROC NEAR
        PUBLIC AsmProc
        call  FAR PTR PublicProc
        call  NearProc
        call  FAR PTR FarProc
        mov   cx,ds:A                   ;pull in variable A from the unit
        sub   cx,2                      ;do something to change it
        mov   ds:A,cx                   ;store it back
        ret
AsmProc ENDP
CODE    ENDS
        END
```

测试这个 Pascal 单元和汇编代码的主程序如下所示：

```
program TSample;
uses Sample;
begin
  TestSample;
end.
```

要用命令行编译和汇编建立样本程序，可使用如下 batch 文件命令。

```
TASM ASMPROC
TPC /B TSAMPLE
TSAMPLE
```

因为一个外部子程序必须在用户 Turbo Pascal 程序的最外层说明，用户不能使用 EXTRN 说明来访问定位于过程或函数中的对象。然而，当从 Turbo Pascal 调用用户的 Turbo Assembler 子程序时能以值参或 Var 参数形式接收这些对象。

4.4.3.1　使用 EXTRN 对象的限制

Turbo Pascal 的限定标识符语法(qualified identifier syntax)使用一个单元名加一个句点来访问一个绝对单元中的对象，它与 Turbo Assembler 的语法规则不相容，因此将被排斥。说明

```
EXTRN SYSTEM.Assign : FAR
```

将产生一个 Turbo Assembler 错误信息。

还有两个对 Turbo Pascal 使用 EXTRN 对象的微小限制。第一个是对过程和函数的引用不能使用地址运算。因此，如果说明

```
EXTRN PublicProc : FAR
```

用户不能写一个如下语句

```
call PublicProc + 42
```

第二个限制是 Turbo Pascal 链接程序将不识别将字压缩成字节的操作符，因此用户不能将这些操作符作用在 EXTRN 对象上。例如，如果用户声明

```
EXTRN i : WORD
```

用户就不能在 Turbo Assembler 模块中使用 LOW i 或 HIGH i 表达式。

4.4.4 使用段定位

Turbo Pascal 产生可以被装入到用户 PC 机主存任何可用地址的.EXE 文件(可重定位)。因为程序不能预先知道它的一个给定段将被装入到何处，链接程序就会使与段地址相关的操作做适当的修正，以使 CODE 和 DATA 包含正确的值。

在运行时，用户的 Turbo Assembler 代码可以使用一些工具获得段地址。例如，设想用户程序需要改变 DS 的值，但用户不想花费在栈中保存原始内容或将这些内容移到一个临时存贮区所需的机器周期，用户就可使用 Turbo Assembler 的 SEG 操作符如下：

```
.
.
.
mov   ax,SEG DATA    ;get the actual address of Turbo Pascal's global DS
mov   ds,ax          ;put it in DS for Turbo Pascal to use
.
.
.
```

当用户的 Turbo Pascal 程序被装入后，DOS 将正确的 SEG DATA 值立即插入到 MOV 指令的立即操作数域。这是重装段寄存器的最快的方法。这个方法允许中断服务程序在 Turbo Pascal 的全局数据段中保存信息。在中断时，DS 将不一定需要包含 Turbo Pascal 的 DS，但前面的语句可以用来访问 Turbo Pascal 变量和类型常量。

4.4.5 无效代码的消除

Turbo Pascal 有消除无效代码的特点，它是指 Turbo Pascal 在产生最后的.EXE 文件时不包括那些从不执行的子程序的代码。但是，因为 Turbo Pascal 没有关于用户的 Turbo Assembler 模块的完整信息，所以它只能进行有限的代码优化。

Turbo Pascal 将消除一个.OBJ 模块的代码，当在那个模块中没有任何可见的子程序或函数被调用。相反，如果模块中任何一个子程序被引用过，整个模块都将保留。

为了最有效地利用 Turbo Pascal 的无效代码删除特性，用户最好将汇编程语言程序分成每个模块仅包含几个子程序。这样做将允许 Turbo Pascal 修剪冗余。

4.5 Turbo Pascal 参数传递约定

Turbo Pascal 使用 CPU 的栈传递参数(或者当是单精度、双精度、扩展或复合(comp)值参时，使用数字处理器栈)。参数总是以它们在子程序说明中出现的顺序从左向右被求值和

压栈。这一节将解释这些参数是怎样被表达的。

4.5.1 值　参

一个值参是一个其值不能被它要传递到的子程序改变的参数。不象许多其它编译器，Turbo Pascal 不是盲目地将每个值参拷贝到 CPU 栈，处理方法视类型而定。这一节和下面几节对其进行解释。

4.5.1.1 标量类型

所有标量类型的值参(布尔型、字符型、短整型、字节类型、整型、字、长整型、子界类型、枚举类型)以值的形式在 CPU 栈上传递。如果一个对象是1个字节大小，它将以一个全16位字压栈，然而，那个字的高位字节不包含有用信息(这个字节不一定象 Turbo Pascal 3.0版或更早版那样一定是0)。如果目标是2字节大小的，只须简单地将其压栈。如果目标是4字节长(一个长整数)，它将以两个16位字压栈。正如在8088系列处理器上已标准化了的一样，低字先入栈且在堆栈中占据较高地址。

注意复合(comp)类型，尽管是一个整型，但不被认为是一个标量类型。因此，在 Turbo Pascal 4.0中，这种类型的值参被传递到8087栈，而不是 CPU 栈。在 Turbo Pascal 5.0中，复合类型的值被传递到主 CPU 栈。

4.5.1.2 实　型

实数型值参(Turbo Pascal 的6字节软件浮点类型)以6字节在栈上传递。这是唯一在栈上传递的长于4字节的类型。

4.5.1.3 单精度、双精度、扩展的和复合型:8087类型

在 Turbo Pascal 4.0中，8087类型的值参是在协处理器栈而不是在 CPU 栈上传递的。因为8087栈深度只有8层，一个 Turbo Pascal 4.0子程序不得有超过8个以上的8087类型的值参。在子程序返回前，所有8087类型参数必须从数值处理器栈弹出。

Turbo Pascal 5.0使用与 Turbo C 相同的为8087值设计的参数传递约定：它们与其它参数一起在主 CPU 栈上传递。

4.5.1.4 指　针

所有指针类型的值参直接以远指针形式压栈——先压入一个包含段的字，再压入一个包含偏移量的字。段地址占据较高地址与 Intel 协定一致。用户的 Turbo Assembler 程序可使用 LDS 或 LES 指令以获得一个指针参数。

4.5.1.5 串

串参数不论其大小通常从不压栈。Turbo Pascal 代之以压入一个长指针到栈中。调用子程序不得改变由指针引用的串，如果需要改变，子程序必须产生一个该串的拷贝。

这个规定的唯一例外，是当一个覆盖单元 A 中的子程序以一个值参的形式传递一个串常量到一个覆盖单元 B 中的子程序。这里，一个覆盖单元是指任何用{$O+}(覆盖允许 OverlaysAllowed)编译的单元。在这种情况下，在实施调用和段地址被传到单元 B 中子程序以前，将串常量保存在堆栈的临时存贮单元中。要了解更多的信息，请参阅《Turbo Pascal 用户指南(5.0)》中的第六章"覆盖"。

4.5.1.6 记录和数组

恰好是1、2或4个字节长的记录和数组在做为值参传递时直接被复制到栈上。如果一个

数组或记录是其他大小(包括3个字节),将用压入一个指向它的指针来代替。在记录或数组不是1、2或4字节长的情况下,如果子程序要改变它,则必须产生一个该结构的局部拷贝。

4.5.1.7 集 合

集合象串一样,通常从不逐字地压栈。相反,一个指向该集合的指针将被压栈。被子程序接收到的指针将指向一个"标准化的"表示集合的32字节。这个集合的最低字节的第一位将始终表示序数值为0的基类型(或其父类型)的元素。

这个规定的唯一例外,是当一个覆盖单元A中的子程序以值参形式传递一个集合常量到一个覆盖单元B中的子程序。这里一个覆盖单元是指任何用{$O+}(覆盖允许OverlaysAllowed)编译的单元。在这种情况下,在实施调用和段地址被传到单元B中子程序以前,将集合常量存放在堆栈的临时存储单元中。要得到更多的信息,请参阅《Turbo Pascal用户指南(5.0)》中的第六章"覆盖"。

4.5.2 变量参数

所有Var参数以完全相同的方式传递,即以指向它们在内存中实际地址的指针方式。

4.5.3 栈的维护

Turbo Pascal希望在主CPU栈中的所有参数在子程序返回以前都被移走。

有两种方式调整栈。用户可使用RET N指令(N是被压入栈的参数的字节数),或者用户可以把返回地址保存在寄存器中(或主存中)并一个一个地弹出参数。弹出技术在8086和8088(这个系列中最低档的处理器)上当优化以提高速度时有用,这里在每次访问"基址加偏移量"寻址上要花费8个机器周期(最少时间)。弹出技术还能节省空间,因为一个POP指令仅占一个字节。

注意:如果用户使用.MODEL、PROC和ARG伪指令,汇编器将自动地把要被弹出的参数字节数加到所有的RET指令中。

4.5.4 存取参数

当用户的Turbo Assembler子程序取得控制权,栈顶将包含一个返回地址(两个或四个字,根据子程序是近调用还是远调用而定),且在它的上面,传递所有参数。注意:当计算参数地址时,要将寄存器(如BP)考虑在内,因为用户可能已将其内容压栈。

有三种基本技术来存取由Turbo Pascal传递到用户的Turbo Assembler子程序的参数。

- 使用BP寄存器寻址堆栈
- 使用其他基址或变址寄存器以得到参数
- 弹出返回地址,再弹出参数

第一种和第二种方法比较复杂,将在下两节描述。第三种方法涉及把返回地址弹出到一个安全地点再将参数弹出到寄存器。这种方法在用户子程序不需要局部变量空间时使用起来更加方便。

4.5.4.1 使用BP寄存器寻址堆栈

第一种(也是最常用的一种)访问从Turbo Pascal传递到Turbo Assembler的参数的方法是使用BP寄存器来寻址堆栈,如下所示:

```
CODE    SEGMENT
        ASSUME cs:CODE
MyProc  PROC FAR                    ;procedure MyProc(i,j : Integer);
        PUBLIC MyProc
j       EQU  WROD  PTR [bp+6]       ;j above saved BP and return address
i       EQU  WORD  PTR [bp+8]       ;i just above i
        push bp                     ;must preserve caller's BP
        mov  bp,sp                  ;make BP point to the top of the stack
        mov  ax,i                   ;address i via BP
        .
        .
        .
```

当用这种方法计算访问参数的堆栈偏移量时，记住给被保存的 BP 寄存器2个字节。

注意本例中存取参数的"正文相等(text equates)"的用法。这就使得代码更加有助记忆。它们只有一个小小的缺点。因为只有 EQU 伪指令(而不是＝伪指令)能用来做这种相等操作，所以用户不能再在同一 Turbo Assembler 源文件中重定义符号 i 和 j。回避这个缺点的一个方法就是使用更多的描述性参数名以使它们不重复；另一个方法是对每个子程序分别汇编。

1. ARG 伪指令

当用户通过 BP 寄存器访问参数时，Turbo Assembler 提供一个计算栈位移量和执行正文相等的选择——ARG 伪指令。当在 PROC 内使用时，ARG 伪指令自动决定参数相对于 BP 的位移量。它还计算在 RET 指令中使用的参数块的大小。因为由 ARG 伪指令产生的符号只在该 PROC 内有效，用户不需要使每个子程序或函数的参数名唯一。

当用 ARG 伪指令重写时，前一个例子变成以下形式：

```
CODE    SEGMENT
        ASSUME cs:CODE
MyProc  PROC  FAR                   ;procedure MyProc(i,j: Integer);
        PUBLIC MyProc
        ARG j : WROD, i : WROD = RetBytes
        push  bp                    ;must preserve caller's BP
        mov   bp,sp                 ;make BP point to the top of the stack
        mov   ax,i                  ;address i via BP
        .
        .
        .
```

Turbo Assembler 的 ARG 伪指令为参数 i 和 j 创建局部符号。行

ARG j: WORD,i : WORD = RetBytes

在子程序生存期自动地使符号 i 与[WORD PTR BP＋6]相等，使符号 j 与[WORD PTRBP＋8]相等，并使符号 RetBytes 与数4相等(4是参数块的字节数)。值将压栈的 BP 和返回地址的大小都考虑在内了。如果 MyProc 是一个 NEAR PROC(近调用 PROC)，则 i 等于[BP＋4]，j 等于[BP＋6]，RetBytes 仍将具有值4。这样，在两种情况下，MyProc 都将以指令 RET

RetBytes 结束。

当使用 ARG 伪指令时，记住以反序列出参数。用户应将 Turbo Pascal 过程(或函数)中最后一个参数放在 ARG 伪指令的开头第一个，依此类推。

另一个需要注意的是在 Turbo Pascal 中使用 ARG 伪指令的顺序。与其他一些语言不同，Turbo Pascal 总是把一个字节大小的值参以一个16位字形式压栈，并且由用户负责告知 Turbo Assembler 额外字节。例如，设想用户写了一段如下所示的 Pascal 说明函数。

```
function MyProc(i,j : char) : string; external;
```

这个子程序的 ARG 伪指令看来将如下所示：

```
ARG j : BYTE : 2, i : BYTE : 2 = RetBytes RETURNS result : DWORD
```

每个参数后的:2是必须的，它通知 Turbo Assembler 每个字符是以一个2字节数组形式压栈的(在这种情况下，高字节不含有用信息)。

在一个返回字符串的函数中(象前一个函数)，ARG 伪指令中的 RETURNS 选择项让用户定义一个指向栈上临时函数结果的位置的变量(刚刚讨论过)。ARG 的 RETURNS 变量不影响参数块的大小(以字节形式)。参看《参考指南》第三章关于 ARG 伪指令的完整信息。

2. .MODEL 和 Turbo Pascal

.MODEL 伪指令与一个 TPASCAL 联用建立一个简化的分段、存储模型和语言支持。在前面用户已知道怎样建立一个 Pascal 的过程和函数的汇编程序。以下是一个使用.MODEL 和 PROC 伪指令的与前面例子等价的例子：

```
        .MODEL TPASCAL
        .CODE
MyProc  PROC FAR i : BYTE, j : BYTE RETURNS result : DWORD
        PUBLIC  MyProc
        mov     ax,i
        .
        .
        .
        ret
```

注意，现在用户不能以反序声明参数且不需要大量其他语句。.MODEL 伪指令与 TPASCAL 联用建立 Pascal 调用约定、定义段名、完成 PUSH BP 和 MOV BP,SP 操作，它还用 POPBP 和 RET N(这里 N 是参数的字节数)返回。

3. 使用另一个基址或变址寄存器

访问参数的第二种方法是使用另一个基址或变址寄存器(BX,SI 或 DI)以从栈中得到参数。记住，这些寄存器的缺省段是 DS 而不是 SS。要使用它们，用户必须使用一个段指定或改变段寄存器。

以下说明如何使用 BX 来得到用户的参数：

```
CODE    SEGMENT
        ASSUME cs:CODE
MyProc  PROC FAR                    ;procedure MyProc(i,j : integer);
```

```
        PUBLIC MyProc
j       EQU WROD PTR [bp+6]        ;j above return address
i       EQU WROD PTR [bp+8]        ;i just above i
        mov  bx,sp                 ;make BX point to the top of the stack
        mov  ax,i                  ;address i via BX
        .
        .
        .
```

在引用了少量参数的子程序里，这种方法节省时间和空间。为什么呢?因为不象 BP，BX 不须在子程序的最后恢复。

4.6 Turbo Pascal 中的函数结果

Turbo Pascal 中的函数根据不同结果类型以不同方式返回结果。

4.6.1 标量函数结果

标量类型的函数结果在 CPU 寄存器中返回。1个字节的值返回在 AL 中，2个字节的值返回在 AX 中，4个字节的值返回在 DX:AX 中(高字在 DX 中)。

4.6.2 实型函数结果

Turbo Pascal 的6字节软件实型函数结果返回在 CPU 寄存器中。高字在 DX 中，中间的字在 BX 中，而低字在 AX 中。

4.6.3 8087函数结果

8087类型的函数结果返回在8087的“栈顶”寄存器 ST(0)(或 ST)。

4.6.4 串函数结果

串类型函数结果在 Turbo Pascal 调用函数前分配的临时存贮区内返回。在第一个参数压栈前，一个指向这个区的长指针被压入堆栈。这个指针不是参数表的一部分。

注意：因为 Turbo Pascal 在调用函数后要利用函数结果指针，所以用户不能将其移出堆栈。

4.6.5 指针函数结果

指针函数结果返回在 DX:AX 中(段基址:偏移量)。

4.7 为局部数据分配空间

用户的 Turbo Assembler 子程序不能为它们自己的变量分配空间，包括静态的(在调用之间一直存在)和动态的(调用后即消失)。我们将在下两节讨论如何处理这两种情况。

4.7.1 分配私有静态存贮区

Turbo Pascal 允许用户的 Turbo Assembler 在全局数据段(DATA 或 DSEG)为静态变量保留空间。要分配空间,仅需如下使用 DB、DW 等伪指令。

```
DATA    SEGMENT PUBLIC
MyInt   DW  ?               ;Reserve a word
MyByte  DB  ?               ;Reserve a byte
.
.
.
DATA ENDS
```

对由 Turbo Assembler 在全局数据段中分配的变量有两个重要限制。首先,这些变量是"私有"的(它们对用户的 Turbo Pascal 程序是不可见的,尽管用户可向它们传递指针)。其次,它们不能象类型常量那样被初始化。语句

```
MyInt   DM  42          ;this will NOT initialize MyInt to 42
```

在其模块链接到用户的 Turbo 程序中时将不会引起一个错误,但在程序运行时 MyInt 将不是真正从值42开始。

用户可以通过说明 Turbo Pascal 变量或类型常量并使用 EXTRN 伪指令,使得它们对 Turbo Assembler 可见这种方法避开这些限制。

4.7.2 分配动态存贮区

用户的 Turbo Assembler 子程序在每次调用活动期内也能在堆栈上分配动态存贮区(局部变量)。在子程序返回前,这个存贮区必须被回收且 BP 寄存器应被恢复。在下面的例子中,过程 MyProc 为两个整型变量 a 和 b 保留空间。

```
CODE    SEGMENT
        ASSUME cs:CODE
MyProc  PROC FAR
        PUBLIC MyProc
        LOCAL a : WROD, b : WORD = LocalSpace  ;a at [bp-2], b at [bp-4]
i       EQU WROD PTR [bp+6]                    ;parameter i above saved
                                               ;BP and return address
        push  bp                               ;must preserve caller's BP
        mov   bp,sp                            ;   make BP point to the top of the
stack
        sub   sp,LocalSpace                    ;make room for the two words
        mov   ax,42                            ;load  A;s initial value intoAX
        mov   a,ax                             ;and theence into A
        xor   ax,ax                            ;clear AX
        mov   b,ax                             ;and initialize B to 0
```

```
                                        ;do whatever needs to be done
        .
        .
        mov   sp,bp                     ;this restores the original SP
        pop   bp                        ;this restores the original BP
        ret   2                         ;this pops the word parameter
MyProc  ENDP
CODE    ENDS
        END
```

注意 Turbo Assembler 的 LOCAL 伪指令是用来为局部变量产生符号和分配空间的。语句

```
LOCAL   a : WORD, b : WORD = LocalSpace
```

在过程生存期内使符号 a 与[BP－2]相等，符号 b 与[BP－4]相等，而符号 LocalSpace 与数4(局部变量区的大小)相等。没有相应的语句产生引用参数的符号，所以用户仍必须使 i 与[BP＋6]相等。

一个更聪明的初始化局部变量的办法是将它们的值压栈而不是缩减 SP。这样，用户也许可以用下列语句代替 SUB SP,LocalSpace：

```
mov     ax,42       ;get the initial value for A
push    ax          ;put it in A
xor     ax,ax       ;zero AX
push    ax          ;and move the zero into B
```

如果用户用这种方法，要确信仔细保持对堆栈的跟踪！在压栈操作前符号 a 和 b 不能被引用。

其它优化措施包括使用 PUSH CONST 指令初始化局部变量(可用于80186，80286和80386)或将 BP 保存在一个寄存器中而不是将其压栈(如果有一个空闲寄存器的话)。

4.8　由 Turbo Pascal 调用汇编语言子程序的例子

在这一节，将提供给一些用户可从一个 Turbo Pascal 程序调用的汇编语言子程序的例子。

4.8.1　通用16进制转换子程序

在 num 处的字节都被转换成长度为 byteCount ＊ 2的十六进制数字的字符串。由于每个字节产生两个字符，因而 byteCount 的最大值为127。为了提高速度，使用了一个 add－daa－adc－daa 序列来把每半字节转换成一个十六进制数字(每半字节等于4位)。

HexStr 被写成一个远调用。这意味着它必须在 Turbo Pascal 单元的接口部分声明或者用 $F＋编译伪指令声明。

```
CODE            SEGMENT
                ASSUME cs:CODE, ds:NOTHING
;Parameters     (+2 because of push bp)
```

```
byteCount       EQU BYTE PTR ss:[bp+6]
num             EQU DWORD PTR ss:[bp+8]
;Function result address (+2 because of push bp)
resultPtr       EQU DWORD PTR ss:[bp+12]

HexStr          PROC FAR
                PUBLIC HexStr
                push    bp
                mov     bp,sp           ;get pointer into stack
                les     di,resultPtr    ;get address of function result
                mov     dx,ds           ;save Turbo's DS in DX
                lds     si,num          ;get number address
                mov     al,byteCount    ;how many bytes?
                xor     ah,ah           ;make a word
                mov     cx,ax           ;keep track of bytes in CX
                add     si,ax           ;start from MS byte of number
                dec     si
                shl     ax,1            ;how many digits? (2/byte)
                cld                     ;store # digits (going forward)
                stosb                   ;in destination string's length byte
HexLoop:
                std                     ;scan number from MSB to LSB
                lodsb                   ;get next byte
                mov     ah,al           ;save it
                shr     al,1            ;extract high nibble
                shr     al,1
                shr     al,1
                shr     al,1
                add     al,90h          ;special hex conversion sequence
                daa                     ;using ADDs and DAA's
                adc     al,40h
                daa                     ;nibble now converted to ASCII
                cld                     ;store ASCII going up
                stosb
                mov     al,ah           ;repeat conversion for lownibble
                and     al,0Fh
                add     al,90h
                daa
                adc     al,40h
                daa
                stosb
                loop    HexLoop         ;keep going until done
                mov     ds,dx           ;restore Turbo's DS
```

```
        pop   bp
        ret   6                    ;parameters take 6 bytes
HexStr  ENDP
CODE        ENDS
        END
```

使用 HexStr 的 Pascal 样本程序如下所示：

```
Program HexTest
var
  num : word;
{$F+}
function HexStr(var num; byteCount : byte) : string; external;
{$L HEXSTR.OBJ}
{$F-}
begin
  num := $face;
  Writeln('The Converted Hex String is "',HexStr(num,sizeof(num)),'"');
end.
```

可使用如下批处理文件以建立和运行 Pascal 例子和汇编程序：

```
TASM HEXSTR
TPC HEXTEST
HEXTEST
```

用户如果使用.MODEL 伪指令，HexStr 程序将被写为如下形式：

```
        .MODEL TPASCAL
        .CODE
HexStr  PROC FAR num:DWORD, byteCount:BYTE RETURNS resultPtr:DWORD
        PUBLIC HexStr
        les   di,resultPtr         ;get address of function result
        mov   dx,ds                ;save Turbo's DS in DX
        lds   si,num               ;get number address
        mov   al,byteCount         ;how many bytes?
        xor   ah,ah                ;make a word
        mov   cx,ax                ;keep track of bytes in CX
        add   si,ax                ;start form MS byte of number
        dec   si
        shl   ax,1                 ;how many digits? (2/byte)
        cld                        ;store # digits (going forward)
        stosb                      ;in destination string's length byte
HexLoop:
        std                        ;scan number from MSB to LSB
        lodsb                      ;get next byte
        mov   ah,al                ;save it
```

```
        shr     al,1                    ;extract high nibble
        shr     al,1
        shr     al,1
        shr     al,1
        add     al,90h                  ;special hex conversion sequence
        daa                             ;using ADDs and DAA's
        adc     al,40h
        daa                             ;nibble now conerted to ASCII
        cld                             ;store ASCII going up
        stosb
        mov     al,ah                   ;repeat conversion for low nibble
        and     al,0Fh
        add     al,90h
        daa
        adc     al,40h
        daa
        stosb
        loop    HexLoop                 ;keep going until done
        mov     ds,dx                   ;restore Turbo's DS
        ret
HexStr  ENDP
CODE    ENDS
END
```

用户可使用相同的 Pascal 样例程序，并汇编上面程序，用相同的批处理文件命令重新编译样本程序。

4.8.2 交换两个变量

通过下面这个过程，用户可以交换两个大小为 count 的变量，如果 count 为0，处理器将试图交换64K。

```
CODE        SEGMENT
            ASSUME cs:CODE, ds:NOTHING
;Parameters (note that offset are +2 because of push bp)
var1        EQU     DWORD PRT ss:[bp+12]
var2        EQU     DWORD PRT ss:[bp+8]
count       EQU     WORD PRT ss:[bp+6]
Exchange    PROC FAR
            PUBLIC  Exchange
            cld                             ;exchange goes upward
            mov     dx,ds                   ;save DS
            push    bp
            mov     bp,sp                   ;get stack base
            lds     si,var1                 ;get first address
```

```
            les     di,var2                 ;get second address
            mov     cx,count                ;get number of bytes to move
            shr     cx,1                    ;get wrod count (low bit -> carry)
            jnc     ExchangeWords           ;if no odd byte, enter loop
            mov     al,ex:[di]              ;read odd byte from var2
            movsb                           ;move a byte from var1 to var2
            mov     [si-1],al               ;write var2 byte to var1
            jz      Finis                   ;done if only one byte to exchange
ExchangeWords:
            mov     bx,-2                   ;BX is a handy place to keep -2
ExchangeLoop:
            mov     ax,es:[di]              ;read a word from var2
            movsw                           ;do a move from var1 to var2
            mov     [bx][si],ax             ;write var2 word to var1
            loop    ExchangeLoop            ;repeat "count div 2" times
Finis:
            mov     ds,dx                   ;get back Turbo's DS
            pop     bp
            ret     10
Exchange    ENDP
CODE        ENDS
            END
```

使用 Exchange 和 Pascal 样本程序如下所示：

```
Program TextExchange;
type
  EmployeeRecord = record
                     Name          : string[30];
                     Address       : string[30];
                     City          : string[15];
                     State         : string[2];
                     Zip           : string[10];
                   end;
var
  OldEmployee, NewEmployee         : EmployeeRecord;
{$F+}
Procedure Exchange(var var1,var2; count        : Word); external;
{$L XCHANGE.OBJ}
{$F-}
begin
  with OldEmployee do
  begin
    Name := 'John Smith';
    Address := '123 F Street';
```

```
    City := 'Scotts Valley';
    State := 'CA'
    Zip := '90000-0000';
  end;
  with NewEmployee do
  begin
    Name := 'Mary Jones';
    Address := '9471 41st Avenue';
    City := 'New York';
    State := 'NY'
    Zip := '10000-1111';
  end;
  WriteLn('Before: ',OldEmployee.Name,' ',NewEmployee.Name);
  Exchange(OldEmployee,NewEmployee,sizeof(OldEmployee));
  WriteLn('After: ',OldEmployee.Name,' ',NewEmployee.Name);
  Exchange(OldEmployee,NewEmployee,sizeof(OldEmployee));
  WriteLn('After: ',OldEmployee.Name,' ',NewEmployee.Name);
end.
```

要建立和运行 Pascal 例子和汇编程序，可使用如下批处理文件命令：

```
TASM XCHANGE
TPC XCHANGE
XCHANGE
```

使用.MODEL 伪指令，Exchange 汇编语言程序将被写成如下形式：

```
        .MODEL TPASCAL
        .CODE
Exchange  PROC FAR var1:DWORD,var2:DWORD,count:WORD
          PUBLIC Exchange
          cld                          ;exchange goes upward
          mov   dx,ds                  ;save DS
          lds   si,var1                ;get first address
          les   di,var2                ;get second address
          mov   cx,count               ;get number of bytes to move
          shr   cx,1                   ;get word count (low bit -> carry)
          jnc   ExchangeWords          ;if no odd byte, enter loop
          mov   al,es:[di]             ;read odd byte from var2
          movsb                        ;move a byte from var1 to var2
          mov   [si-1],al              ;write var2 byte to var1
          jz    Finis                  ;done if only one byte to exchange
ExchangeWords:
          mov   bx,-2                  ;BX is a handy place to keep -2
ExchangeLoop:
          mov   ax,es:[di]             ;read a word from var2
```

```
                movsw                           ;do a move from var1 to var2
                mov     [bx][si],ax             ;write var2 word to var1
                loop    ExchangeLoop            ;repeat "count div 2" times
        Finis:
                mov     ds,dx                   ;get back Turbo's DS
                ret     10
        Exchange ENDP
        CODE    ENDS
                END
```

用户可使用相同的 Pascal 样本程序，汇编上面程序，用相同的 batch 文件命令重新编译样本程序。

4.8.3　扫描 DOS 环境

使用 EnvString 函数，用户可以扫描一个形为"s=SOMESTRING"的 DOS 环境变量，如果发现就返回 SOMESTRING。

```
        DATA        SEGMENT PUBLIC
                    EXTRN prefixSeg : WORD      ;gives location of PSP
        DATA        ENDS
        CODE        SEGMENT PUBLIC
                    ASSUME cs:CODE,ds:DATA
        EnvString   PROC FAR
                    PUBLIC EnvString
                    push    bp
                    cld                                 ;work upward
                    mov         es,[prefixSeg]          ;look at PSP
                    mov         es,es:[2Ch]             ;ES:DI points at environment
                    xor         di,di                   ;which is paragraph-aligned
                    mov         bp,sp                   ;find the parameter address
                    lds         si,ss:[bp+6]            ;which is right above the return address
                    ASSUME      ds:NOTHING
                    lodsb                               ;look at length
                    or          al,al                   ;is it zero?
                    jz          RetNul                  ;if so, return
                    mov         ah,al                   ;otherwise, save in AH
                    mov         dx,si                   ;  DS:DX contains pointer to first parm
char
                    xor         al,al                   ;make a zero
        Compare:
                    mov         ch,al                   ;we want ch=0 for next count, if any
                    mov         si,dx                   ;get back pointer to string sought
                    mov         cl,ah                   ;get length
                    mov         si,dx                   ;get pointer to string sought
```

```
                repe    cmpsb                   ;compare bytes
                jne     Skip                    ;if compare fails, try next string
                cmp     byte prt es:[di],'='    ;compare succeeded. Is next char '='?
                jne     NoEqual                 ;if not, still no match
Found:
                mov     ax,es                   ;make DS:SI point tostring we found
                mov     ds,ax
                mov     si,di
                inc     si                      ;get past the equal {=} sign
                les     bx,ss:[bp+10]           ;get address of function result
                mov     di,bx                   ;put it in ES:DI
                inc     di                      ;get past the length byte
                mov     cl,255                  ;set up a maximum length
CopyLoop:
                lodsb                           ;get a byte
                or      al,al                   ;zero test
                jz      Done                    ;if zero, we're done
                stosb                           ;put it in the result
                loop    CopyLoop                ;move up to 255 bytes
Done:           not     cl                      ;we've been decrementing CL from
                                                ;255 during save
                mov     es:[bx],cl              ;save the length
                mov     ax,SEG DATA
                mov     ds,ax                   ;restore DS
                ASSUME  ds:NOTHING
Skip:
                dec     di                      ;check for null from this char on
NoEqual:
                mov     cx,7FFFh                ;search a long way if necessary
                sub     cx,di                   ;enviroment never > 32K
                jbe     RetNul                  ;if we're past end,leave
                repne   scasb                   ;look for the next null
                jcxz    RetNul                  ;exit if not found
                cmp     byte ptr es:[di],al     ;second null in a row?
                jne     Compare                 ;if not, try again
RetNul:
                les     di,ss:[bp+10]           ;get address of result
                stosb                           ;store a zero there
                mov     ax.SEG DATA
                mov     ds,ax                   ;restore DS
                ASSUME  ds:DATA
                pop     bp
                ret     4
```

```
EnvString    ENDP
CODE         ENDS
             END
```

使用 EnvString 的 Pascal 样本程序如下：

```
program EnvTest;
  { program looks for environment strings }
var
  EnvVariable : String;
  EnvValue : String;
{$F+}
function EnvString(s:string) : string; external;
{$L ENVSTR.OBJ}
{$F-}
begin
  EnvVariable := 'PROMPT';
  EnvValue := EnvString(EnvVariable);
  if EnvValue='' then EnvValue := '* * * not found * * *';
  Writeln('Environment Variable: ',EnvVariable,'  Value: ',EnvValue);
End.
```

可使用如下批处理文件命令来建立和运行 Pascal 例子和汇编程序：

```
TASM ENVSTR
TPC ENVTEST
ENVTEST
```

如果用户使用.MODEL 伪指令，EnvString 的汇编语言程序将被写成如下形式：

```
        .MODEL TPASCAL
         .DATA
         EXTRN prefixSeg : WORD                ;gives location of PSP
         .CODE
EnvString  PROC FAR EnvVar:DWORD RETURNS EnvVal:DWORD
         PUBLIC EnvString
         cld                                   ;work upward
         mov     es,[prefixSeg]                ;look at PSP
         MOV     es,es:[2Ch]                   ;ES:DI points at environment
         xor     di,di                         ;which is paragraph-aligned
         mov     bp,sp                         ;find the parameter address
         lds     si,EnvVar                     ;which is right above the return address
         ASSUME  ds:NOTHING
         lodsb                                 ;look at length
         or      al,al                         ;is it zero?
         jz      RetNul                        ;if so, return
         mov     ah,al                         ;otherwise, save in AH
         mov     dx,si                         ; DS:DX contains pointer to first parm
```

```
char
                xor     al,al                   ;make a zero
        Compare:
                mov     ch,al                   ;we want ch=0 for next count, if any
                mov     si,dx                   ;get back pointer to string sought
                mov     cl,ah                    ;get length
                mov     si,dx                   ;get pointer to string sought
                repe    cmpsb                   ;compare bytes
                jne     Skip                    ;if compare fails, try next string
                cmp     byte prt es:[di],'='      ;compare succeeded. Is next char '
='?
                jne     NoEqual                 ;if not, still no match
        Found:
                mov     ax,es                   ;make DS:SI point to string we found
                mov     ds,ax
                mov     si,di
                inc     si                      ;get past the equal {=} sign
                les     bx,EnvVar               ;get address of function result
                mov     di,bx                   ;put it in ES:DI
                inc     di                      ;get past the length byte
                mov     cl,255                  ;set up a maximum length
        CopyLoop:
                lodsb                           ;get a byte
                or      al,al                   ;zero test
                jz      Done                    ;if zero, we're done
                stosb                           ;put it in the result
                loop    CopyLoop                ;move up to 255 bytes
        Done:   not     cl                      ;we've been decrementing CL from
                                                ;255 during save
                mov     es:[bx],cl              ;save the length
                mov     ax,SEG DATA
                mov     ds,ax                   ;restore DS
                ASSUME  ds:DATA
                ret
                ASSUME  ds:NOTHING
        Skip:
                dec     di                      ;check for null from this char on
        NoEqual:
                mov     cx,7FFFh                ;search a long way if necessary
                sub     cx,di                   ;enviroment never > 32K
                jbe     RetNul                  ;if we're past end, leave
                repne   scasb                   ;look for the next null
                jcxz    RetNul                  ;exit if not found
```

```
                cmp     byte ptr es:[di],al     ;second null in a row?
                jne     Compare                 ;if not, try again
RetNul:
                les     di,EnvVal               ;get address of result
                stosb                           ;store a zero there
                mov     ax,SEG DATA
                mov     ds,ax                   ;restore DS
                ASSUME  ds:DATA
                ret
EnvString       ENDP
CODE            ENDS
                END
```

用户可使用相同的 Pascal 样本程序，汇编上面的程序，使用相同的批处理文件命令重新编译样本程序。

第五章 Turbo Pascal 与 DOS 和 BIOS 的接口

一般的 PC 机由不同的物理设备组成：一个键盘，一个监视器，几个磁盘驱动器，一台打印机等等。DOS 和 BIOS 包含一些软件例程(例行程序)，这些例程控制上述设备，确保数据准确无误地往返于各个部位间。

Turbo Pascal 程序不断使用 DOS 和 BIOS 服务功能来做如下一些工作：写磁盘文件、在监视器上显示信息、读取当前时间和日期等等。因为 Turbo Pascal 为你代劳了这些，因而在通常情况下，不必知道被调用的 DOS 和 BIOS 服务功能的一些细节。但是仍有两个理由使得你还须了解这些服务功能并知道如何使用它们。

首先，虽然 Turbo Pascal 提供了对许多服务功能的访问调用，但并没有全部用到，如果要使 PC 机服服贴贴地处在你的控制下，还得利用 DOS 和 BIOS 的强有力服务功能；其次，即使从来就用不着这些功能，但学会有关它们的一些内容，无疑将大大加深对个人计算机和操作系统的理解。

5.1 8088 寄存器

8088 微处理器族(包括 8086 和 80286)都有用来执行命令的由 14 个寄存器组成的标准寄存器集或内存地址(LOCATION)，每个寄存器是 16 位字长，这也是 8088 被称作 16 微处理器的原因(在 Turbo Pascal 中，一个 16 位的存贮单元叫一个字)。8088 的寄存器如图 5.1 所示。

前四个寄存器 AX，BX，CX 和 DX 是通用寄存器，临时存放计算和比较与其它操作中所使用的数据。汇编语言程序员以 Pascal 程序员使用变量的方式来使用这些寄存器。这样 AX 由 AH 和 AL 组成。这些通用寄存器是调用 DOS 和 BIOS 功能时最常用到的。

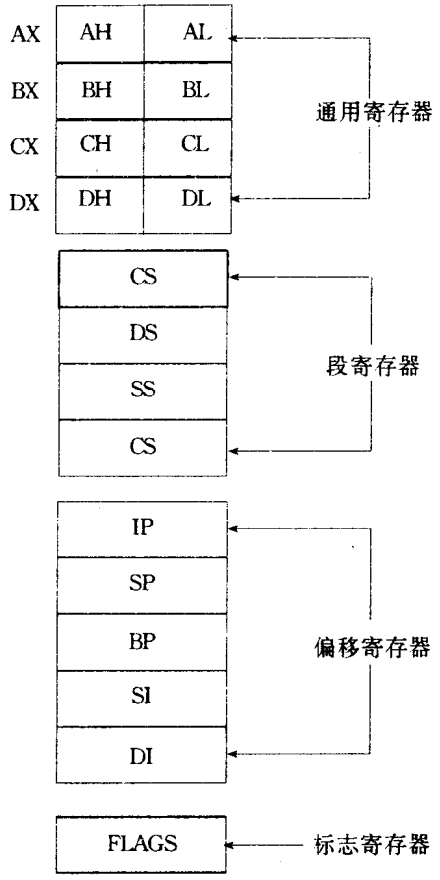


图 5.1　8088 CPU 寄存器

8088 还有四个段寄存器:CS、DS、SS 和 ES。CS 存放代码段地址,DS 存放数据段地址,ES 存放一些特殊操作所用到的临时段地址。CS 和 SS 寄存器存放的关键数据的变化对于程序的完整有很大危险性,因此,不允许用这些寄存器来进行功能调用。但 DS 和 ES 偶尔用来传递段地址。

一个存贮地址由一个段地址和一个内偏移量组成。8088 包含 5 个偏移寄存器:IP、SP、BP、SI 和 DI。这些寄存器与段寄存器一起合用,对内存特殊位置进行寻址。Turbo Pascal 只许访问 SI、DI 和 BP,IP 和 SP 不许在 DOS 和 BIOS 调用中使用。

标志寄存器含有最近执行过的指令所产生的状态信息。尽管并不是全部字位都用到了,但用到的每一位都表示成 CPU 操作所引起的某个条件。标志寄存器主要用于识别错误条件。虽然它也可以在 Turbo Pascal 中使用,但它对 DOS 和 BIOS 调用一般说来是必须的。因为 DOS 和 BIOS 调用的错误代码一般在一个通用寄存器中返回。

5.2　DOS 单元

Turbo Pascal 提供了一个标准单元,叫 DOS 单元。它不仅包含了按用户的意愿调用功能服务所需的数据结构和过程,还包含了调用某些具体 DOS 和 BIOS 服务功能的例程。有了这些例程,可以得到文件信息、目录列表,可以为系统和单个文件设置时间和日期,还可以做

更多工作，本书特别叙述这些新的过程并告诉如何使用它们。

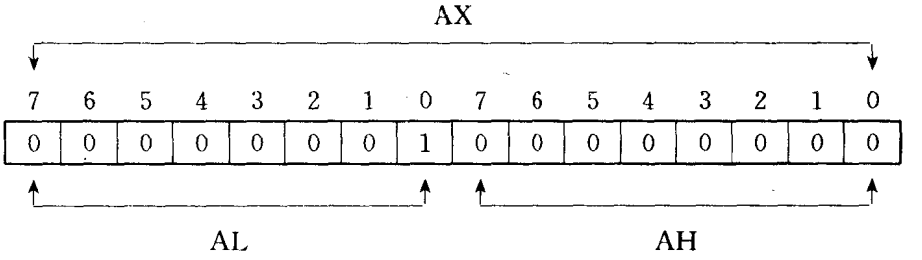
DOS 单元中的一些过程需要一些特别数据类型来定义一些变量。DOS 单元还包括两个过程：MsDos 和 Intr。可以用来调用某个 DOS 或 BIOS 服务功能。两个过程都带有类型为 Registers 的参数变量，有了 MsDos 和 Intr 和 Registers 数据类型，就可以利用由 DOS 和 BIOS 提供的全部功能调用。比较庆幸的是，Borland 公司已经在 DOS 单元中装满了为大多数常用功能调用而提供的易调过程(easy—to—call)。

5.3　寄存器集

Registers 变量是开启 DOS 和 BIOS 功能服务库大门的钥匙。传递给 MsDos 和 Intr 两个过程的 Registers 有着与大多数 8088 寄存器相匹配的域：

```
Type
    Registers = Record
        Case Integer Of
            0:(AX,BX,CX,DX,BP,SI,DI,DS,ES,FLAGS : WORD);
            1:(AL,AH,BL,BH,CL,CH,DL,DH : BYTE);
        End;
```

Registers 数据类型只包含了在 BIOS 和 DOS 服务功能中用到的 CPU 寄存器，该记录有两个变体部分：一部分包含表示全寄存器的 WORD 型变量，另一部分包含只定义通用寄存器高半部或低半部的字节型变量。例如，两个 Byte 型变量 AL 和 AH 合起来是指引用包含 AX 的同一存储地址。低位字节 AL 在高位字节 AH 之前，这是因为 8088 微处理器按反序排放一个字里面的两个字节。这样，如果 AX 中放整数 1，在存贮器里面就如下所示：



在调用 MS DOS 或 Intr 之前，还应将寄存器集变量设置为某个值来让计算机知道需要哪个功能服务和怎样执行它。例如，选择 DOS 服务功能时，可将服务号置入 AH 寄存器。DOS 服务 2BH 是设置系统日期的，其使用方法如下所示：

```
Program SetTime;
Uses DOS,CRT;
Var
    Regs : Registers;
Begin
ClrScr;
FillChar(Regs,SizeOf(Regs),0);
```

```
With Regs Do
    Begin
      AH  :=  $2B;
      DH  :=  12;     (* 月份  *)
      DL  :=  31;     (* 日期  *)
      CX  :=  1990;   (*  年    *)
      End;
MsDos(Regs);          (* call the Dos service *)

If Regs.AL <> 0 Then
   WriteLn('Error!')
Else
   WriteLn('Date has been Set.');
WriteLn;
WriteLn('Press ENTER...');
ReadLn;
End;
```

程序首先用下面语句将寄存器清零：

```
FillChar(Regs,SizeOf(Regs),0)
```

接着将设置系统日期的功能号 2BH 放入 AH 中。然后向寄存器 DH、DL 和 CX 填入日期信息。

MsDos 接受寄存器集变量作为一个参数并调用 DOS 过程将系统日期更新为 1990 年 12 月 31 日。如果 DOS 服务功能检测出一个错误，将返回寄存器集变量，并在 AL 中放入错误码，语句

```
If Regs.AL <> 0 Then
    WriteLn('Error!');
```

检查该代号。如果不为 0，则意味着出错了。

MS DOS 过程是用于 DOS 功能服务，而 Intr 过程是用于 BIOS 服务功能的。Intr 接受两个变量：中断号和寄存器集变量。例如，BIOS 的打印屏幕功能是通过设 AH 为 5，再调用中断 5 来实现的，如下所示：

```
FillChar(Regs,SizeOf(Regs),0);
Regs.AH := 5;
Intr(5,Regs);
```

第一个参数是中断号。上例调用了中断 5，中断 5 能用来为我们做很多事情。在这时将 Regs.AH 设为 5，就将屏幕打印功能调用具体化了（该功能调用也可通过同时按下 SHIFT 键和 PRTSC 键来实现）。在该功能调用中，没有在寄存器集中返回错误指示字。

使用 DOS 和 BIOS 功能调用使得你使用 Turbo Pascal 时如虎添翼，但要学会使用它们是要费点时间的。本章下面的部分将叙述一些程序，它们把 DOS 和 BIOS 功能调用中最有用的精华部分罗列在一起，并作为使用它们的示例。

5.4　磁盘驱动功能调用

磁盘操作系统的主要目标就是管理好计算机的磁盘驱动和文件系统。庆幸的是，Turbo Pascal 标准过程对于与磁盘有关的最棘手的任务如读文件，都注意到了。这一节将说明能改善程序，但 Turbo Pascal 却没有支持的几个 DOS 功能。

5.4.1　报告磁盘空闲空间

DOS 功能调用 36h 能指明磁盘上还有多大空间可用。DL 寄存器选取要检查的磁盘。若为 0 则选取缺省驱动器，若为 1 则选取 A 驱动器，为 2 则选取 B 驱动器等。调用了 MS DOS 后通用寄存器包含有如下信息：

AX　每个分配的簇中扇区数
BX　未用簇数量
CX　每个扇区字节数
DX　总簇数

利用这些值，不难计算出空闲空间的总量为

LongInt(AX) * BX * CX

这里 LongInt 类型强制转换(typecast)是必要的，可避免整数溢出。如果说明的驱动器无效，Turbo Pascal 在 AX 中返回 $FFFF。

下面给出的函数 FreeDiskSpace。报告驱动器上磁盘空闲字节数：

```
Program DiskSpace;
Use Crt,Dos;
Var
    Drive : Char;
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Function FreeDiskSpace(Drive : Char) : LongInt;
Var;
   Regs : Registers;
Begin
FillChar(Regs,SizeOf(Regs),0);
With Regs Do
Begin
    AH := $36;
    DL := Ord(UpCase(Drive)) - 64;
    End;
MsDos(Regs);
  With Regs Do
    If AX = $FFFF Then
       FreeDiskSpace := -1;
    Else
       FreeDiskSpace := Long Int(AX) * BX * CX;
```

```
    End;
(*************************************)
  Begin
  ClrScr;
  Write('Which Drive ? (A/B/C/D):');
  Readln(Drive);
  Drive := UpCase(Drive);

  Writeln (Free Disk Space  (Drive):0',bytes free.');
  Writeln;
  Write('Press ENTER ...');
  Readln;
  End.
```

程序前面函数中的参数是一代表待检驱动器名的字符。为了把驱动器号赋给 DL，函数将参数 Drive 变成大写，转换成 ASCII 码值，然后减去 64。如果 Drive=a，则变换成 A，A 的 ASCII 码是 65。65 减 64 得 1，这就是要赋给 DL 的正确驱动器号。

调用了 MsDos 后，函数检查 AX 寄存器，若 AX 为 $FFFF，意味着程序执行过程中出错，函数返回－1。如果没错，函数就计算出空闲磁盘空间。

5.4.2 读取和设置文件属性

磁盘文件可以具有 DOS 提供的六种属性之一：只读属性、隐含属性、系统属性、卷标属性、子目录和归档属性。文件属性包含在一个单字节中，每个属性由字节位控制。

属性字节位 1 标志文件的只读。只读文件在各种方式下都不能改变。DOS 阻止任何对只读文件的写删除企图，如同软盘写保护缺口上的保护片对软盘实行保护一样。

位 2 告诉文件是否为隐含文件。隐含文件一般包含一些敏感信息，它们被 DOS 忽略，不能由 Dir 命令列出来，也不能被删除或显示等。这样，除非用户有能发现它们的程序，否则隐含文件是不可见的。虽然不承认隐含文件，Turbo Pascal 仍允许将它们用作输入输出。

位 3 控制着系统属性。同隐含文件一样，系统文件也不被 DOS 命令认识。但系统属性并没什么特别用处，只是从 CP/M 继承保留下来的。

位 4 与卷标号有关，卷标号是一个标识软盘或硬盘的说明。在盘被格式化时，这是一个由用户设置的可选项。

位 5 是表明文件是一个子目录文件来跟踪目录和子目录。

位 6 是归档属性位，当文件首次创建时就被置位。当使用 DOS Backup 命令复制此文件时，该位被关闭，直到文件内容有所改变。归档位允许对那些自上次以来已修改过的文件进行备份。7 位和 8 位未用。

要想知道文件属性，或设置所需属性，可使用 DOS 功能调用 43H，由 AL 寄存器中的值控制，AL=1 则设置文件属性。

要是把文件属性报告出来，43h 功能调用在 CL 寄存器中返回属性字节。通过测试各位，就可以定出每个属性位状态。在设置文件属性时，要建立一个属性字节并置入 CL 中，然后再调用 MsDos。

不管是设置还是报告文件属性字节，都要往寄存器集 DS:DX 中装入文件名的 ASCII 字符串的段和偏移量。一个 ASCII 字符串是一个以二进制 0 为结尾的字母串。加上 #0 后可用作 Turbo Pascal 字符串，Turbo Pascal 字符串有一个长度字节，而 ASCII 串则没有。因此，为了把 Turbo Pascal 串用作 ASCII 串，要使用串中的首字符地址。

在下面程序中，DOS 功能调用 43h 使用了 6 个布尔型参数，每一个对应一个文件属性：

```
Program FileAttributes;
Uses Crt,Dos;
Var
      Fname  :  String;
      Ro,Hidden,
      Sys,Vol,
      SubDir,Arch,Error : Boolean;
 (* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Procedure GetFileAttributes(FileName:String;
                            Var RO,
                            Hidden,
                            Sys,
                            Vol,
                            SubDir,
                            Arch,
                            Error:Boolean);
 Regs:Registers;
 Begin
 Fillchar(Regs,SizeOF(Regs),0);
 FileName := FileName + #0;

 With Regs Do
 Begin
     AH := $43;
     DS := Seg(FileName);
     DX := Ofs(FileName)+1;
     End;
     MsDos(Regs);

 Error := (Regs.AL in [2,3,5]);
 Ro := (Regs.CL  And $01 )>0;
 Hidden := (Regs.CL  And $02 )>0;
 Sys := (Regs.CL  And $04 )>0; Vol := (Regs.CL  And $08 )>0;
 SubDir := (Regs.CL And $10 )>0;
 Arch := (Regs.CL And $20) >0;
 (* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
 Begin
```

```
    ClrScr;
    Write('While File?;');
    Readln(Fname);

  GetFileAttributes (Fname,
                     Ro,
                     Hidden,
                     Sys,
                     Vol,
                     SubDir,
                     Arch,
                     Error);
    If Error Then
       Writeln('Error');
    Else
       Begin
       Writeln(Fname,'has these attributes:');
       Writeln('Read Only : 'Ro);
       Writeln ('Hidden: 'Hidden);
       Writeln('System file: 'Sys);
       Writeln(Volume. label: ',Vol);
       Writeln('Subdriectory:',SubDir);
       Writeln('Archive: 'Arch);
       End;

  Writeln;
  Write('Press ENTER...');
  Readln;
  End.
```

DOS 功能调用 43h 在没有找到文件(这时 AL=2),或没找到路径(这时 AL=3),或不允许访问文件(这时 AL=5)时,报告出错。

下面的程序设置文件属性字节。该程序接受对文件四个属性的四个布尔参数。它不包括卷标识别和子目录属性,因为这两个属性不能由 DOS 功能调用设置。

```
Program FileAttributes;
Uses CRT,DOS;
Var
     Ch : char;
     Fname : String;
     RO,Hidden,
     Sys,Arch : Boolean;
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Procedure SetFileAttributes(FileName : String;
```

```
                              Var RO,
                              Hidden,
                              Sys,
                              Arch : Boolean);
Var
    Regs : Registers;
Begin
    FillChar(Regs,SizeOf (Regs),0);
    FillName := FileName + #0;
    With Regs  Do
      Begin
      AH := $43;
      AL := 1;
      DS := Seg(FileName);
      DX := Ofs (FileName)+1;
    If RO Then
      CL := (CL Or $01);
    If Hidden Then
      CL := (CL Or $02);
    If Sys Then
      CL := (CL Or $04);
    If Arch Then
      CL := (CL Or $20);
    End;
    MsDos(Regs);
    End;
(*************************************)
    Begin
      ClrScr;
      Write('Which file? :');
      Readln(Fname);

      Write('Set to read only? (Y/N)');
      Readln(ch);
      RO := UpCase(ch)='Y';

      Write('Set to hidden(Y/N)');
      Readln(ch);
      Hidden := UpCase(ch)='Y';

      Write('Set to archire(Y/N)');
      Readln(ch);
      Arch := UpCase(ch)='Y';
```

```
Write('Set to system file (Y/N)');
Readln(ch);
Sys := UpCase(ch)='Y';

SetFileAttributes(Fname,
                  RO,
                  Hidden,
                  Sys,
                  Arch);

Writeln(Fname,'has been set to these attributes:');
Writeln('Read Only :',RO);
Writeln('Hidden :',Hidden);
Writeln('System file :',Sys);
Writeln;
Write('Press ENTER ...');
Readln;
End.
```

5.4.3 目录列表

显示一个磁盘目录需要三个不同的 DOS 功能调用，并要了解程序段前缀(ProgramSegment PrefixlPsp)和磁盘传送区(Disk-transfer ArealDTA)，后者是前者的一部分。

一个程序开始执行时，DOS 拨出前 256 字节内存为程序的 PSP。因为 PSP 包含了高级技术信息，因此除了 DAT 部分外，其它 PSP 部分通常不能被程序员访问。DAT 是一个 128 字节的缺省缓冲区，用作象读磁盘目录那样的 DOS 操作，这时，DAT 所包含的信息如表 5.1 所示。

表 5.1 DTA 关于目录列表的内容

内 容	偏移地址	长度
DOS 使用的数据	0	21
文件属性	21	1
文件的 Time Stamp	22	2
文件的 date stamp	24	2
文件字节数	26	4
文件名和扩展名	30	13

在从 DAT 获取信息之前，要知道其地址，这可由 DOS 功能调用 2Fh 给出。2Fh 功能调用被引用执行时，将 DTA 段放入 ES，把 DTA 偏移量放入 BX，如下所示：

```
Regs.AH := $2F;
MsDos(Regs);
DTAseg := Regs.ES;
```

```
DTAofs := Regs.BX;
```

以上程序段将 DTA 段存放在变量 DTAseg 中，把位移量存在 DATofs 中。这些变量同 Turbo Pascal 标准数组 Mem 一起使用，从 DTA 中截取某些信息。例如：

```
Mem[DTAseg:DTAofs+21]
```

指向 DAT 中文件属性字节。

DOS 功能调用 4Eh 在目录中搜索第一个匹配文件并把文件信息填入 DTA。但在被调用前，寄存器要含有文件路径名串的段和段内位移量。如下面代码所示：

```
Mask In := Mask In + #0;
With Regs Do
Begin
AH := $4E;
DS := Seg(Mask In);
DX := Ofs(Mask In) + 1;
CL := $00;
End;
MsDos(Regs);

If Regs.AL<> Then Exit;
```

CL 寄存器告诉 DOS 在搜索时应当包含什么类型的文件。如果该寄存器被设为 0，DOS 就查找标准文件。要想在目录列表中包括隐含文件或系统文件，可把 CL 按表 12—2 提供的指导置值。

若 CL 置为 16H(是 2H、4H 和 10H 之和)，那么目录列表中将包括隐含文件、系统文件和子目录文件。若 AL 寄存器返回一个非零值，则表示在目录中未找到与文件说明相匹配的文件项。

程序 Directory 使用过程 DirList 建立一个与用户输入文件说明相对应的文件名。

```
Program Directory;
Uses CRT,DOS;
Type
    Dir Files = Array [1..200] Of String[13];
Var
    FileSpec : String;
    i,
    fc : Integer;
    df : Dir Files;
(*********************************************)
Procedure DirList(Mask In : String;
                  Var Name List : Dir Files;
                  Var File Counter : Integer);
Var
    i : Byte;
```

```
        Regs : Registers;
        DTAseg,
        DTAofs : Word ;
        FileName : String[20];
Begin
FillChar(Regs,SizeOf(Regs),0);
Fill  Counter := 0;

Regs.AH := $2F ;
MsDos(Regs);
With Regs Do
    Begin
    DTAseg := ES ;
    DTAofs := BX ;
    End;

FillChar(Regs,SizeOf(Regs),0);
Mask  In := Mask  In + #0
With Regs Do
    Begin
    AH := $4E;
    DS := Seg(Mask  In);
    DX := Ofs(Mask  In) + 1;
    CL := $00;
    End;
MsDos(Regs);

If Regs.AL<> 0 Then Exit;
i := 1;
    Repeat
    FillName[i] := Chr(Mem[DTAseg:DTAofs+29+i]) ;
    i := i + 1;
    Until (FileName[i-1] < #32) Or (i > 12);
FileName[0] := Chr(i-1);
File  Counter := 1;
Name  list[File  counter] := FileName ;
    Repeat
    FillChar(Regs,SizeOf(Regs),0);
    With Regs Do
        Begin
        AH := $4F;
        CL := $00;
        End;
```

```
        MsDos(Regs) ;
        If Regs.AL = 0 Then
              Begin
              i := 1;
              Repeat
              FillName[i] := Chr(Mem[DTAseg:DTAofs+29+i]);
              i := i + 1;
              Until (FileName[i-1] < #32) Or (i > 12);

        Inc(File  Counter,1);
        FileName[0] := Chr(i-1);
        Name  List [File  Counter] := FileName;
        End;
    Until Regs.AL <> 0;
    End;
    (*************************************)
    Begin
    ClrScr;
      Repeat
      Write('Enter file spec: ');
      ReadLn(FileSpec);
      If FileSpec <> '' Then
          Begin
          DirList(FileSpec,df,fc);
          For i := 1 To Fc Do
                WriteLn(df[i]);
          WriteLn ;
          End;
      Until FileSpec = '';
    End.
```

过程 DirList 接受三个参数:Mask In、Name List 和 File Counter。字符串参数 Mask In 为要匹配的文件说明(例如:test.pas、*.pas 或者???.pas)。字符串数组 Name List 中存放与文件说明匹配的文件名。整数 File Counter 返回 DirList 找到的相匹配文件名个数。

注意,DOS 功能调用 4EH 只找到第一个匹配文件,而 4FH 调用则查找所有后面文件,把匹配的文件逐个找出来,直到 AL 为非零。这时表示文件目录中再也没有相匹配的文件了。

表 5.2 用 CL 寄存器设置 DOS 文件属性

要具有的属性	CL 值
隐含	$02
系统	$04
卷标志	$08
子目录	$10

5.5　视频功能调用

大多数用户几乎完全使用视频标准来评价程序优劣，这主要是因为精心设计的引人入胜的显示使程序易于使用。令人遗憾的是，Turbo Pascal 只提供了有限的屏幕控制功能。本节所讲述的内容有助于用户更好地控制显示器，做更复杂的视频显示花样。

5.5.1　报告当前视频模式

屏幕控制的一个基本方面就是确定计算机视频适配器类型。主要类型有：单显、CGA、PCjr 和 EGA。

BIOS 的中断 10H 报告正在使用的视频适配器类型，下面所示的函数给出了示例：

```
Program VideoMode;
Uses CRT, DOS ;
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Function CurrentVidMode : Char ;
Var
    Regs : Registers ;
Begin
FillChar(Regs,SizeOf(Regs),0) ;
Regs.AH := $0F ;
Intr($10,Regs);
    Case Regs.AL Of
    1..6   : CurrentVidMode := 'C' ;      (* CGA        *)
    7      : CurrentVidMode := 'M' ;      (* Monochrome* )
    7..10  : CurrentVidMode := 'P' ;      (* PCjr       *)
    13..16: CurrentVidMode := 'E' ;       (* EGA        *)
    End;
End;
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Begin
ClrScr ;
WriteLn('Current video mode is : ',CurrentvidMode);
WriteLn;
Write('Press ENTER...') ;
ReadLn;
End.
```

在调用中断服务之前，函数将 AH 设为 0FH。中断程序将屏幕宽度（每行字符个数）放在 AH 中，将视频模式放入 AL，把视频页数放入 BH。上面程序通过检查 AL 内容确定视频

模式。如果该寄存器值为 7，表示屏幕为单显状态，函数返回字母 M。当值为 1 到 6 时，表明是彩显（返回字母 C）。值为 8 到 10 时，表明是 Pcjr（返回字母 P）。值为 13 到 16 时，表明是 EGA（返回字母 E）。

对于第十三章所讲专题，即直接往视频存贮区写数据来说，了解显示器类型是最基本的。

5.5.2　设置光标大小

在有些情况下，程序里最好不让光标显示出来。在另一些情况下，大光标比小光标更有用。在典型的一般情况下，一个光标用两条扫描线组成，但对彩色图形适配器来说，它可显示 8 条扫描线的光标，对单显适配器的情况，可显示 14 线光标。用的扫描线越多，则光标越大，若不用扫描线，则光标消失。

使用 BIOS 中断 10H，将 AH 置为 1，就可设置光标大小。将始扫描线号放入 CH，将尾扫描线号放入 CL。彩色适配器使用 8 线（0～7），单显适配器使用 14 线（0～13），序号低的扫描线出现在上面，序号高的线出现在下面。例如，彩显上的小光标由 6 线和 7 线构成，这两线为靠底部的两线。

```
 Program Cursor ;
Uses DOS, CRT;
(****************************************)
Procedure cursorSize(Stype,Size : Char ) ;
Var
      Regs : Registers;
      i     : Integer;
Begin
Size := UpCase(Size);
If UpCase(Stype) = 'M' Then
      i := 6
Else
      i := 0;

Regs.AH := $01;
      Case Size Of
      '0' :
          Begin
            Regs.CH := $20;
            Regs.CL := $20;
          End;
      'B' :
          Begin
          Regs.CH := $0;
          Regs.CL := $7+i;
          End;
```

```
        'S' :
            Begin
            Regs.CH := $6+i;
            Regs.CL := $7+i;
            End;
        End;
    Intr($10,Regs);
    End;
    (*********************************************)
    Begin
    ClrScr;
    WriteLn('Big cursor');
    CursorSize('C','B');
    WriteLn;
    Write('Press ENTER...');
    ReadLn;

    WriteLn;
    WriteLn;
    WriteLn('No cursor');
    CursorSize('C','O');
    WriteLn;
    Write('Press ENTER...');
    ReadLn;

    WriteLn;
    WriteLn;
    WriteLn('Small cursor');
    CursorSize('C','S');
    WriteLn;
    Write('Press ENTER...');
    ReadLn;

    End.
```

这个程序按传递的参数设置光标大小。参数 STYPE 可为 M(单盘)或 C(彩显)。参数 SIZE 可取 3 个值:B 表示大光标、S 表示小光标、O 表示关掉光标。

如果计算机使用单显,变量 I 置为 6,否则置为 0。简单地将 CH、CL 都置为 20H 可关掉光标;将 CH 置为 0,CL 置为 7 可设置彩显大光标;将 CH 置 0,CL 置 13 可设置单显大光标。将 CH 置 6,CL 置 7,得到彩显小光标;CH 置 12,CL 置 13,得到单显小光标。

5.5.3 从屏幕读字符

可以用 BIOS 中断 10H 从视屏上读字符。下面的 SCREENCHAR 函数示例了如何用中

断从屏幕上读字符。

```
Program ScreenTest;
Uses DOS,CRT;
Var
      s : String;
      i : Integer;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Function ScreenChar : Char;
Var
      Regs : Registers;
Begin
FillChar(Regs,SizeOf(Regs),0);
Regs.AH := 8;
Regs.BH := 0; (* Video page *)
Intr($10,Regs);

ScreenChar := Chr(Regs.AL);
End;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Begin
ClrScr;

WriteLn('ABCDE');

s := '';
For i := 1 To 5 Do
      Begin
      GotoXY(i,1);
      s := S + ScreenChar;
      End;
WriteLn;
WriteLn(S);
WriteLn;
Write('Press ENTER...');
ReadLn;
End.
```

因为 ScreenChar 是从当前光标位置读字符，在调用中断之前将光标正确定位。AH 放入 8，BH 中的数选取所用的视频页，但一般都用 0 页。

在调用了中断后，AL 中返回当前光标位置字符的 ASCII 码。ScreenChar 将 ASCII 码变成一个字符并作为函数值返回。

5.6　时间和日期功能

DOS 有一个内部时钟，对时间和日期保持跟踪。当一个文件建立或改变时，DOS 用当前时钟值为文件置上时间、日期邮戳。表 5.3 给出的 DOS 功能调用让用户控制系统日期和时间。

5.6.1　获取系统日期

DOS 功能调用 2AH，是报告当前系统日期的。它在调用后，在各寄存器中返回日期信息，如表 5.4 所示。

表 5.3　DOS 系统时间、日期功能调用

DOS 功能调用	功能
2AH	报告系统日期
2BH	设置系统日期
2CH	报告系统时间
2DH	设置系统时间

在调用这些功能之前，要在 AH 中放入适当的功能调用号。

表 5.4　DOS 2Ah 调用后寄存器内容

寄存器	返回信息
AL	星期(0 表示星期天)
CX	年
DH	月
DL	日

在过程 GetSystemDate(见下面)给出了示例，它使用 2Ah 功能调用得到系统日期，然后把得到的数据转换成字符串格式，返回给主程序。

```
Program  Date;
Uses  CRT,DOS;
Var
      S:String;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Procedure GetSystemDate(Var date : String);
Var
      Regs : Registers;
      st1,st2,st3,st4 : String[10];
```

```
   Begin
   FillChar(Regs,SizeOf(Regs),0);
   Regs.AH := $2A;
   MsDos(Regs);
   With Regs Do
         Begin
              Case AL Of
              0  :  St1  :='Sunday';
              1  :  St1  :='Monday';
              2  :  St1  :='Tuesday';
              3  :  St1  :='Wednesday';
              4  :  St1  :='Thursday';
              5  :  St1  :='Friday';
              6  :  St1  :='Saturday';
              End;
         Str(CX,st2);      (*  Year   *)
         Str(DH,st3);      (*  Month  *)
         Str(DL,st4);      (*  Date   *)
         End;
   If Length(st3) = 1 Then
         st3 := '0' + st3;
   If Length(st4) = 1 Then
         st4 := '0' + st4;
   date := st1+' '+st3+'-'+st4+'-'+st2;
   End;

   (*************************************)
   Begin
   ClrScr;

   GetSystemDate(s);
   WriteLn('The date is ',s);
   WriteLn;
   WriteLn('Press ENTER...');
   ReadLn;
   End.
```

GetSystemDate 使用 Case 语句决定星期几。把年、月、日转换成字符串,如果年或月由单个数字组成,则在字符串前加前缀 0。

5.6.2 设置系统日期

DOS 服务 2BH 设置系统日期,在调用 MsDos 之前,必须把月份放在寄存器 DH 中,日放在寄存器 DL 中,年放在寄存器 CX 中,下面过程说明 Turbo Pascal 是如何使用上述服务

的：

```
Program Date;
Uses CRT, DOS;
Var
  Error : Boolean;
(*************************************************)
Procedure SetSystemDate( Month, Day, Year : Integer;
                         Var Error : Boolean);
Var
  Regs : Registers;
Begin
FillChar(Regs,SizeOf(Regs), 0);
With Regs Do
Begin
AH := $2B;
DH := Month;
DL := Day;
CX := Year;
End;
MsDos(Regs);
Error := Regs.AL <> 0;
End;

(*************************************************)
Begin
ClrScr;

SetSystemDate(1,1,1990,Error);

If Error Then
  WriteLn('Error!');
Else
  WriteLn('Date has been set.');

WriteLn;
Write('Press ENTER...');
ReadLn;

End.
```

如果键入一个非法日期，那么将出错。此时，寄存器返回一个错误代码。如果 AL 中为非零值，布尔变量 Error 将置为 TRUE。

5.6.3 获取和设置系统时间

DOS 功能调用 2Ch 报告系统时间，功能调用 2DH 则设置系统时间。报告和设置系统时间与系统日期的操作十分类似。以下两个过程介绍如何报告和设置时间：

```
Program SysTime;
Uses DOS, CRT;
Var
      Hour,
      Minute,
      Second : Byte;
      Error : Boolean;

(*********************************************)
Procedure GetSystemTime(Var Time : String);
Var
    Regs : Registers;
    h,m,s : Word;
    st1,st2,st3,st4 : String[10];
Begin
FillChar(Regs,SizeOf(Regs),0);
Regs.AH := $2C;
MsDos(Regs);
With Regs Do
      Begin
      Str(CH,st1);
      Str(CL,st2);
      Str(DH,st3);
      Str(DL,st4);

If Length(st1) = 1 Then
    st1 :='0' + st1;

If Length(st2) =1 Then
    st2 := '0' + st2;

If Length(st3) =1 Then
    st3 :='0' + st3;

If Length(st4) =1 Then
    st4 :='0' + Then

Time := st1+':'+st2+':'+st3+':'+st4;
```

```
End;

(*****************************************)
Procedure SetSystemTime( Hour, Minute, Second : Byte;
                         Var Error : Boolean);
Var
      Regs : Registers;
Begin
FillChar(Regs,SizeOf(Regs),0);
With Regs Do
      Begin
      AH := $2D;
      CH :=Hour;
      CL :=Minute;
      DH :=Second;
      END;
MsDos(Regs);
Error := Regs.AL <> 0;
End;

(*****************************************)
Begin
ClrScr;
Write('Hour : ');
ReadLn(Hour);
Write('Minute: ');
ReadLn(Minute);
Write('Second: ');
ReadLn(Second);

SetSystemTime(Hour,Minute, Second, Error);

GetSystemTime(s);
WriteLn('Time now: ',s);

Write('Press ENTER...');
ReadLn;

End.
```

如果设置系统时间出错，AL 返回错误代码。返回的 AL 值如非零则表明有错。

5.6.4 获取和设置文件的时间和日期

DOS 功能调用 3Dh 能够报告或设置一个文件的时间和日期，由于要用到文件句柄，并

且日期和时间是作为一个数字值编码，因此涉及磁盘文件的时间和日期功能的问题就复杂了。文件句柄是 DOS 用来处理磁盘输入输出的约定。

DOS 功能调用 3Dh 打开一个文件并在寄存器 AX 中返回文件句柄，即可用得到文件的句柄。函数 GetFileHandle 在下面程序中用来接受一个文件名并返回一个文件句柄：

```
Function GetFileHandle( FileName : String;
                        Var Error : Boolean) : Integer;
Var
    Regs : Registers;
    i : Integer;
    Begin
    FileName := FileName + #0;
    FillChar(Regs,SizeOf(Regs),0);
    With Regs Do
        Begin
        AH := $3D;
        AL := $00;
        DS := Seg(FileName);
        DZ := Ofs(FileName)+1;
        End;
    MsDos(Regs);

    i := Regs.AX;

    If (Lo(Regs.Flags) And $01) > 0 Then
        Begin
        Error := True;
        GetFileHandle := 0;
        Exit;
        END;
    GetFileHandle := i;
    End;
```

如果出错，GetFileHandle 返回 0 并把错误参数置为 TRUE。如果无错，则打开文件，可以接受报告或文件时间和日期。由于使用了 DOS 功能调用 3Eh 打开了文件来得到句柄，因此在做这些工作后，要关闭文件，如下所示：

```
Procedure CloseFileHandle(i : Integer);
Var
Regs : Registers;
Begin
With Regs Do
    Begin
    AX := $3E;
    BX := i;
```

```
        End;
    MsDos(Res);
    End;
```

简而言之，报告或设置文件的时间和日期分三步做：

1. 打开文件并保存文件句柄。
2. 使用文件句柄报告或设置文件时间和日期。
3. 关闭文件。

另外两个过程——GetFileTimeAndDate 和 SetFileTimeAndDate ——示例了如使用 57h 功能调用。如果 AL 设为 0，Tubor Pascal 报告时间和日期；如果 AL 设为 1，则设置时间和日期。在其它情况下，BX 存放文件句柄。

```
Program FileStamp;
Uses DOS, CRT;
Var
      Fname,
      Time  st,
      Day  st : String;
      Month, Day,
      Year, Hour,
      Minute,Second : Word;
      Error : Boolean;

(*********************************)
Function GetFileHandle( FileName : String;
                        Var Error : Boolean) : Integer;
Var
      Regs : Registers;
      i: Integer;
Begin
FileName := FileName + #0;
FillChar(Regs,SizeOf(Regs),0);
With Regs Do
      Begin
      AH  :=  $3D;
      AL  :=  $00;
      DS  :=  Seg(FileName);
      DX  :=  Ofs(FileName)+1;
      End;

MsDos(Regs);

i := Regs.AX;
```

```
If (Lo(regs.Flags) And $01 ) >0 Then
        Begin
        Error := True;
        GetFileHandle := 0;
        Exit;
        End;
GetFileHandle := i;
End;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Procedure CloseFileHandle(i : Integer);
Var
        Regs : Registers;
Begin
With Regs Do
        Begin
        AH := $3E;
        BX := i;
        End;
MsDos(Regs);
End;
(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)
Procedure GetFileTimeAndDate( File  Name : String;
                              Var Time  st,
                              Day  st : String;
                              Var Error : Boolean);
Var
        Regs : Registers;
        i : Integer;
        st1,st2,st3 : String[4];
        y,m,d,r,h,s,Time,Day : Word;

Begin
Error := False;
Time  st := ' ';
Day  st := ' ';

i := GetFileHandle(File  Name,Error);
If Error Then Exit;

With Regs Do
       Begin
       AH := $57;
```

```
        AL := $00;
        BX := i;
        End;

MsDos (Regs);
CloseFileHandle(i);

(*Convert Time *)
r := Regs.CX;
h := r Div 2048;
r := r - (h * 2048);
m := r Div 32;
r := r - (m * 32);
s := r * 2;

Str(h:0,st1);
Str(m:0,st2);
Str(s:,st3);

If Length(st1) = 1 Then st1 := '0' + st1;
If Length(st2) = 1 Then st2 := '0' + st2;
If Length(st3) = 1 Then st3 := '0' + st3;
Time  st := st1+':'+st2+':'+st3;

(*  Convert Day *)
r := Regs.DX;
y := (r Div 512) + 1980;
r := r - ((y - 1980) * 512);
m := r Div 32;
r := r - (m * 32);
d := r;
Str(y:0, st1);
Str(m:0, st2);
Str(d:0, st3);
If Length(st1) = 1 Then st1 := '0' + st1;
If Length(st2) = 1 Then st2 := '0' + st2;
If Length(st3) = 1 Then st3 := '0' + st3;
Day  st := st2 + '-' + st3 + '-' + st1;
End;

(*********************************)
Procedure SetFileTimeAndDate(File  Name : String;
                             Month,Day,
```

```
                              Year,Hour,
                              Minute,Second : Word;
                              Var Error : Boolean);
Var
    Regs : Registers;
    i,j,k : Word;
    t,d : Word;
Begin
Error := False;
i := GetFileHandle(file Name,Error);
If Erroe Then Exit;

t := (Hour * 2048 ) + ( Minute * 32 ) + (Second Div 2 );
d := (( Year -1980 ) * 512) + ( Month * 32 ) + Day;
With Regs Do
    Begin
    AH := $57;
    AL := $01;
    BX := i;
    CX := t;
    DX := d;
    End;

MsDos(Regs);
CloseFileHandle(i);
End;

(*****************************************)
Begin
ClrScr;

Write('File: ');
ReadLn(Fname);
Write('Month: ');
ReadLn(Month);
Write('Day: ');
ReadLn(Day);
Write('Year: ');
ReadLn(Year);
Write('Hour: ');
ReadLn(Hour);
Write('Minute: ');
Readlen(Minute);
```

```
Write('Second: ');
ReadLn(Second);

SetFileTimeAndDate(Fname,
                   Month,Day,Year,
                   Hour,Minute,Second,
                   Error);

GetFileTimeAndDate(Fname,Time  st,Day  st,Error);
If Error Then
     WriteLn('Error!')
Else
     WriteLn(Time  st,' ',Day  st);

WriteLn;
Write('Press ENTER...');
ReadLn;
End.
```

GetFileTimeAndDate 调用 MsDOS 过程后，在 CX 和 DX 中分别放置时间和日期，它们这时都是 Word 变量。这两个变量接着通过算术运算分解成几部分：时、分、秒和日、月、年。这些部分组成字符串并在参数 time—St 和 date—St 中返回。

要设置文件的时间和日期，首先必须计算代表时间和日期的数值。在过程 SetFileTimeAndDate 中，输入值(时，分，秒，日，月，年)当作参数传递进去。过程将其转换成两个数，分别存入 CX(时间)和 DX(日期)。然后，MsDos 调用将设置文件时间和日期。

5.6.5　报告换档键状态

Turbo Pascal 不能直接读取一些 PC 机上的功能键：NUMLOCK、SCROLL LOCK、CTRL、ALT、两个 Shift 键 CAPSLOCK 和 INS 键。报告这些键状态的 BIOS 16h 可加强对键盘的控制功能。

可使用中断 16h(AH 设为 2)来检测这些特殊键状态。在执行 16h 中断后，AL 中返回一个状态字节。该字节中的每一位表明了这八个键中一个键的状态。

在下面程序中 16h 中断检测这八个特殊功能键的状态：

```
Porgram Shift;
Uses CRT,DOS ;
Var
     Ins,
     CapsLock,
     NumLock,
     ScrollLock,
     Alt,
```

```
        Ctrl,
        LeftShift,
        RightShift : Boolean;

(*************************************)
Procedure ShiftStatus(Var Ins,
                          CapsLock,
                          NumLock,
                          ScrollLock,
                          Alt,
                          Ctrl,
                          LeftShift,
                          RightShift : Boolean);
Var
        Regs : Registers ;

Begin
Regs.AH := 2;
Intr($16,Regs);

RightShift    :=  (Regs.AL And $01 ) > 0;
LeftShift     :=  (Regs.AL And $02 ) > 0;
Ctrl          :=  (Regs.AL And $04 ) > 0;
Alt           :=  (Regs.AL And $08 ) > 0;
ScrollLock    :=  (Regs.AL And $10 ) > 0;
NumLock       :=  (Regs.AL And $20 ) > 0;
CapsLock      :=  (Regs.AL And $40 ) > 0;
Ins           :=  (Regs.AL And $80 ) > 0;

End;

(*************************************)
Begin
ClrScr;
WriteLn('Press Ins and then Ctrl to stop...');

        Repeat
        ShiftStatus(Ins,CapsLock,NumLock,ScrollLock,
                    Alt,Ctrl,LeftShift,RightShift);

        GotoXY(1,4);
        WriteLn('Ins.........',Ins,' ');
        WriteLn('CapsLock....',CapsLock,' ');
```

```
        WriteLn('NumLock.....',NumLock,' ');
        WriteLn('ScrollLock..',ScrollLock,' ');
        WriteLn('Alt.........',Alt,' ');
        WriteLn('Ctrl........',Ctrl,' ');
        WriteLn('LeftShift...',LeftShift,' ');
        WriteLn('RightShift..',RightShift,' ');
        Until (Ins And Crtl);
    WriteLn;
    Write('Press ENTER...');
    ReadLn;
    End.
```

过程 ShiftStatus 有对应于 8 个特殊功能健的布尔参数，检测 AL 中返回的字节中的每一位。这样，根据状态字节中各位可以置位 8 个参数。

5.7 Turbo Pascal DOS 单元

DOS 和 BIOS 中有的是功能很强的功能调用，它们可用 MsDos 和 Intr 过程调用。Borland 公司使这些工作更加易行，因为他们提供了最常用的 DOS 和 BIOS 功能调用进行访问的特别例程。这些例程，连同使用这些例程所需的数据结构，都包含在 DOS 单元中。

5.7.1 DOS 单元常量

DOS 单元含有帮助你简化编程的许多常量。这些常量可按主题分为三类：标志常量（用来解释 CPU 的标志寄存器）、文件模式常量（用于 Turbo Pascal 文件处理过程）和文件属性常量（用于解释文件属性字节）。这些常量的声明如下所示：

```
{标志常量}
Const
  FCarry      =  $0000;     {* 进位标志 *}
  FParity     =  $0004;     {* 奇偶标志 *}
  FAuxiliary  =  $0010;     {* 辅助标志 *}
  FZero       =  $0000;     {* 零标志   *}
  FSign       =  $0080;     {* 符号标志 *}
  Foverflow   =  $0800;     {* 溢出标志 *}

{文件模式常量}
Const
  fmClose     =  $D7B0;     (*  文件关闭           *)
  fmInput     =  $D7B1;     (*  文件打开成输入     *)
  fmOutput    =  $D7B2;     (*  文件打开成输出     *)
  fmInOut     =  $D7B3;     (*  文件打开成输入输出 *)
```

```
{文件属性常量}
  Const
    ReadOnly   =  $01;
    Hidden     =  $02;
    SysFile    =  $04;
    Volume ID  =  $08;
    Directory  =  $10;
    Archive    =  $20;
    AnyFile    =  $3F;
    End;
```

5.7.2 DOS 单元数据类型

DOS 单元包含几类数据的声明，可用于 DOS 单元中各例程。FileRec 类型用于 typed 和 untyped 文件变量。TextRec 数据类型用于正文文件变量。

```
Type
    {Type and untyped}
    FileRec = Record
        Handle    : Word;
        Mode      : Word;
        RecSize   : Word;
        Private   : Array[1..26] Of Byte;

    {Textfile Rectord}
    TextBuf =  Array[0..127] Of Char;
    TextRec =  Record
        Handle    : Word;
        Mode      : Word;
        Bufsize   : Word;
        Private   : Word;
        BufPos    : Word;
        BufEnd    : Word;
        BufPtr    : ^TextBuf;
        OpenFunc  : Pointer;
        InOutFunc : Pointer;
        FulshFunc : Pointer;
        CloseFunc : Pointer;
        UserData  : Array[1..16] of Byte;
        Name      : Array[0..79] of Char;
        Buffer    : TextBuf;
        End;
```

FileRec 和 TextRec 都包含了一个 Mode 域，这个域可以通过使用前边描述过的文件模

式常数被说明。

Registers 数据类型被用来同 Intr 和 MsDos 例行程序一起完成 DOS 和 BIOS 功能：

```
Type
        Regsiter = Record
        Case Integer Of
                0 : (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : Word);
                1 : (AL,AH,BL,BH,CL,CH,DL,DH :Word);
              End;
```

在 Registers 数据类型中的每个域都引用了一个 CPU 寄存器。通过在这个记录中设置值，可以调用系统级的服务，就象用汇编语言一样。

DOS 单元还包括了用于时间和日期函数的数据类型(DataTime)，以及用于目录操作的数据类型(SearchRec)。数据类型 Date Time 是同过程 GetTime 和 SetTime 一起被使用的，这两个过程读取和设置系统时钟的时间的日期。

5.7.2.1 DateTime 类型

```
Type
        DateTime = Record
        Year,Month,Day,Hour,Min,Sec,Integer;
        End;
```

5.7.2.2 SearchRec 类型

```
Type
        SearchRec = Record
        Fill :  Array[1..21] of Byte;
        Attr:   Byte;
        Time    LongInt;
        Size :  LongInt;
        Name    String[12];
        ENd;
```

SearchRec 记录类型是与两个过程——FindFirst 和 FindNext 一起被使用的。这两个过程读取一个目录中的文件条目。在 SearchRec 记录中的域包括了文件属性(Attr)、时间印记(Time)、文件大小(Size)和文件名(Name)。

DOS 单元还声明了三种串数据类型：DirStr、NameStr 和 ExStr。DirStr 类型用来存储一个文件名的目录路径部分(例如，C:\TP\TEMP)，NameStr 用于文件名，ExtStr 用于文件扩展。

```
Type
        DirStr   = String[67];
        NameStr= String[8];
        ExtStr   = String[3];
```

这些类型是同过程 Fsplit 一起被使用的，该过程接收一个完整的文件指定并分开来返回路径、文件名和扩展。

5.7.3 DosError 变量

在出错时，DOS 单元的许多例程将 DosError 置值以表明发生了哪个错误。DosError 是

整数型变量，可取的值如表 5.5 所示。

表 5.5 DosError 的可取值

0	无错
2	未找到文件
3	未找到路径
5	访问被拒绝
6	无效的文件句柄
8	存贮空间不够
10	无效的环境
11	无效的格式
18	无更多文件

5.7.4 DOS 单元过程与函数

当用 Intr 与 MsDos 过程存取任何 DOS 或 BIOS 功能时，DOS 单元包含许多子程序，使之很容易使用这些功能。这些子程序在下面的段中要说明。

5.7.4.1 中断支持子程序

中断支持子程序是调用任何 DOS 或 BIOS 服务以及安装用户自己的中断服务子程序的工具。GetIntVec 为被一个特定中断执行的子程序返回当前地址；SetIntVec 以用户的中断子程序来代替现存中断子程序。过程 Intr 可执行任何中断服务，而 MsDos 仅执行 DOS 服务。

5.7.4.2 日期和时间例程

系统时钟保持对当前日期和时间的追踪。可以用 GetTime 和 GetDate 过程从系统时钟那里得到日期和时间。与此相仿，可以用 SetTime 和 SetDate 过程设置系统时钟的时间和日期。

文件有其自己的日期戳记，即一个记录了文件建立或最新更新时间和日期的长整数。可用 GetFTime 得到任一文件的时间戳记。要先将文件的时间戳记传给 UnPackTime，它产生一个日期和时间，然后才可以解释它。可以用 PackTime 将此过程反过来。PackTime 接受一个时间和日期并返回一个长整数，可把它用作在 SetFTime 中设置文件的时间和日期。

5.7.4.3 磁盘和文件例程

DOS 单元含有两个磁盘状态例程：DiskFree 和 DiskSize。DiskFree 返回磁盘上空白字节数，DiskSize 返回磁盘上已用和未用的总字节数。

对磁盘文件进行操作的例程是 DOS 单元中最有用的例程。FindFirst 和 FindNext 这两个例程可让用户读任何目录下的文件。正如其名字的含义，FindFirst 读目录下第一个文件的信息，而 FindNext 则读后续文件信息。在搜索文件时可使用 DOS 通配符字符，甚至可以说明文件的类型（例如归档文件、系统文件和隐含文件）。

如果要知道某个文件的属性，可用 GetAttr，它返回文件的属性字节。另一方面，SetFAttr 允许设置一个文件的属性字节。FExpand 接受一个文件名参数并返回一个完整的文件说

明，包括磁盘驱动器名、路径和文件名。FSplit 所做的正相反，接受一个完整文件说明而返回该说明的某些部分：路径、名字、扩展名。Fsearch 寻找目录列表中某个文件。若发现该文件，Fseach 返回其完整文件说明，否则返回一个空白串。

5.7.5 进程例程

Exec、keep 是 DOS 单元中两个更高级的例程。Exec 允许在一个程序里面执行另外的程序或一个 DOS 外壳命令，而 keep 终止执行一个程序但让它驻留在内存里面。SwapVectors 例程同 Exec 一起使用时，提供一个安全度。当在 Turbo Pascal 程序里面执行一个分程序(child program)时，面临着一个危险：分程序可能永久性的改变中断矢量表。如果在调用 Exec 之前和之后调用 SwapVectors，就可确保中断矢量恢复到其初始状态。最后一个进程控制例程是 DosExitCode，它提供了程序结束时的出错情况。出错代码可用于其它程序或批命令文件。

DOS 维护一个包含计算机“环境”信息的内存区(举例说来，这些信息有文件句柄号、COMMAND. COM 的位置等)。这些信息包含在一些“环境串”当中。函数 EnvCount 返回内存中环境串个数，EnvStr 则返回某个环境串。第三个例程 GetEnv 则返回某个环境单元(如 FILES、COMSPEC、PATH)的环境串。

最后，DOS 单元还包括了一些提供关于计算的信息或对计算机进行控制的零杂例程。DosVer 例程返回一个整数，其高位和低位字节包含了所用 DOS 的版本号。个人计算机还具有两个可以开启或关闭的特征：Control-break 检测和 disk-write 验证。CTRL-BREAK 组合键用于中止程序运行。当 CTRL-BREAK 检测被禁止时，DOS 只是在控制台状态、打印状态或通讯 I/O 状态检测 CTRL-BREA/K，而当 CTRL-BREAK 被使用时，DOS 在每个系统调用时都检测 CTRL-BREAK。例程 GETCBREAK 返回一个布尔值指示 CONTROL-BREAK 检测是开还是关。SETCBREAK 接受一个布尔值，可使 Control－Break 检测开启或关闭。与此相仿，GetVerfy 返回一个布尔值指示 disk－write 验证是开着还是关着，而 SetVerify 接受一个布尔值对 disk－write 验证进行开启或关闭操作。当 disk－write 验证能被使用时，DOS 对每次磁盘写操作进行验证；被禁止时，DOS 则不加验证。

这些 DOS 单元中的例程在附录 E 中逐个被详细说明。以下程序则示例了每个例程(keep 除外)，并让用户对在程序中如何使用它们有一个很好的了解。

```
{$F}
{$M 10000, 0, 0}
Program Test;
Uses DOS,CRT;
Var
    OldTimerVec : Pointer;
    Regs : Registers;
    ClockFlag : Word;
    x,y : Byte;
    i :Integer;
    S : String;
```

```
(*************************************)
Procedure CallOldInt(sub : Pointer);
Begin
{  This procedure calls an interrupt service by address  }
Inline($9C/$FF/$5E/$06);
End;

(*************************************)
Procedure XYChar(x,y : Byte;
                 c : Char;
                 fg,bg : Byte);
{
This routine writes a character to the screen using
a BIOS routine.
}
Begin
GotoXY(x,y)
FillChar(Regs,Size0f(Regs),0);
With Regs Do
     Begin
     AH := $09;                  (* Call BISO Service 9        *)
     AL := Ord(C);               (* Character in AL            *)
     BH := 0;                    (* Video page in BH           *)
     BL := (bg shl 4) + fg;      (* Attribute in BL            *)
     CX := 1;                    (* Number of repetitions in CX *)
     End;
Intr($10,Regs);                  (* Call BIOS Interrupt 10h    *)
End;

(*************************************)
Procedure Clock : Interrupt
{
This procedure replaces the original clock interrupt.
}
Begin
callOldInt(OldTimerVec);         (* Call the original routine  *)
Inc(ClockFlag,1);                (* Increment the counter      *)
Str(ClockFlag,s);                (* convert to string          *)
For i := 1 To Length(s) Do       (* Write string with XYChar   *)
     XYChar(i,10,s(i),Yellow,Black);
End;

(*************************************)
```

```
Function PadRight(S :String; L:Word):String;
{
This function adds blank characters to
a String until it reaches length L. If
the String is longer than L to begin with,
this function truncates the String.
}
Begin
If Length(S) >L Then
       S[0] :=Chr(L);
While Length(S) <L Do
       S:=S+' ';
PadRight :=S;
End;

(*************************************)
Procedure InterruptSupportDemo;
Var
  I:Word;
  FileName :String;
  Begin
  ClrScr;

( * Use BIOS interrupt to determine video adapter * )

FillCgar(Regs,SizeOf(Regs),0);
Regs.AH:= $ 0f;
Intr( $ 10,regs);
Case Regs.AL of
       1..6    :  Writeln('CGA');
       7       :  Writeln('Monochrome')
       8..10   :  writeln('PCjr');
       13..16  :  Writeln('EGA');
End;

( * Use DOS Service to determine if file is read—only * )

Writeln;
Write('Enter file name: ');
Readln(FileName);
FillCgar(Regs,SizeOf(Regs),0);

( *   Add null character to String   * )
```

```
( *   So String can be used as        * )
( *   ASCIIZ String.                    * )

FileName := FileName + #0;

With Regs Do
     Begin
     AH := $43;
     DS := Seg(FileName);
     DX := Ofs(FileName)+1;        ( * Skip length byte            * )
     End;
MsDos(Regs);

If ((Regs.CL And $01)> 0) Then
     WritenLn('File is Read-Only');
Else
     Writeln('File is not Read-Only');
Writeln;
Write('Press ENTER...');
Readln;

ClockFlag :=0;
GetIntVec(8,OldTimeVec); ( * Save interrupt address * )
SetIntvec(8,@Clock); ( * Point timer Interrupt to Clock);
Writeln;
Write('Press ENTER  ');
Readln;
SetInteVer(8,OldTimerVec);          ( * Restore Interrupt address    * );
End;

( * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * )
Procedure DateAndTimeProcedures;
Const
     DayName: Array [0..6] of String[10] =('Sunday',
                                           'Monday',
                                           'Tuesday',
                                           'Wednesday',
                                           'Thursday',
                                           'Friday',
                                           'Saturday');
Var
  Year, Month, Day, DayOfWeek,
  Hour,Minute,Second,Sec100:Word;
```

```
  fname : String;
  f : File;
  T : LongInt;
  DT : DateTime;

Begin
ClrScr;

(*  Display current date and time  *)
WriteLn('Current date and time.');
GetDate(Year,Month,Day,DayofWeek);
Writeln('System Date=',DayName[DayOfWeak],' ',Month,'/',Day,'/',Year);
GetTime(Hour,Minute,Second,Sec100);
WriteLn('System time=',Hour,':',Minute,':',Second,':',Sec100);
WriteLn;

WriteLn('Set current date and time.');
(* Set new date and time *)
Write('Enter year (1980 Or later):');
ReadLn(Year);
Write('Enter month:');
Read(Month);
Write('Enter Day');
ReadLn(Day);
Write('Enter Hour:');
ReadLn(Hour);
Write('Enter minute:');
ReadLn(Minute);
Second:=0;
Sec100:=0;
SetDate(Year,Month,Day);
SetTime(Hour,Minute,*second,Sec100);

(* Display new date and time *)
WriteLn('New date and time');
GetDate(Year,Month,Day,DayOfWeek);
Writeln('System Date=',DayName[DayOfWeek],' ',Month,'/',Day,'/',Year);
GetTime(Hour,Minute,Second,Sec100);
WriteLn('System time=',Hour,':',Minute,':',Secind,':',Sec100);
WriteLn('Press ENTER...');
ReadLn;

(*  Get time and date for a file  *)
```

```
ClrScr;
WriteLn('Ger date and time for a file.');
Write('Enter file name:');
ReadLn(fname);
Assign(f,fname);
Reset(f);
GetFTime(f,T);
Unpack Time(T,DT);
With DT Do
        Begin
        WriteLn('File name: ',fname);
        WriteLn('Date:      'Month,'/',Day,'/',Year);
        WriteLn('Time:      'Hour,':',Min,':',Sec);
        End;

(* Set new time and date for file *)
WriteLn('Set file's date and time.');
With DT Do
        Begin
        Write('Year:');
        ReadLn(Year);
        Write('Month:');
        ReadLn(Month);
        Write('Day: ');
        Readln(Day);
        Write('Hour: ');
        ReadLn(Hour);
        Write('Minute: ');
        ReadLn(Minute);

        Second :=0;
        End;
PackTime(DT,T);
SetFTime(f,T);

(* Get new time and date for file *)
WriteLn('Get new time and date for file.');
GetFTime(f,T);
UnPackTime(T,DT);
With DT Do
        Begin
        WriteLn('File name: ',fname);
        WriteLn('Date:      ',Month,'/',Day,'/',Year);
```

```
        WriteLn('Time:        ',Hour,':',Min,':',Sec);
        End;

Close(f);
WriteLn;
Write('Press ENTER...');
ReadLn;
End;

(*********************************************)
Procedure DiskStatusFunctions;
Var
      S:LongInt;
Begin
ClrScr;
WriteLn('Disk status.');
S:=DiskFree(0);
WriteLn('Free space on disk =',s,' bytes/',s Div 1024,
        'Kbytes');
S:=DiskSize(0);
WriteLn('Total space on disk =',s,' bytes/',s Div 1024,
        'Kbytes');
WriteLn;
Write('Press ENTER...');
ReadLn;
End;

(*********************************************)
Procedure FileHandling;
Var
      Attr : Word;
      f : File;
      S : String;
      DT : DateTime;
      Srec : SearchRec;
      PS : PathStr;
      DS : DirStr;
      FN : NameStr;
      EN : ExtStr;

Begin
ClrScr;
WriteLn('Press ENTER for a directory listing.');
```

```
ReadLn;
FindFirst('*.*',AnyFile,Srec);
While DosError = 0 Do
      Begin
      With Srec Do
          Begin
          UnPackTime(Time,DT);
          With DT Do
              Begin
              S := FExpand(Name);
              Fsplit(S,DS,FN,EN);
              WriteLn(PadRight(DS,10),' ',
                      PadRight(FN,9),' ',
                      ParaRight(EN,5),' ',
                      Size:7,' ',Year);
              End;
          End;
      FindNext(Srec);
  End;
WriteLn;
Write('Press ENTER...');
RcadLn;

WriteLn('Search for a file by name.');
ClrScr;
      Repeat
      Write('Enter file name: ');
      ReadLn(S);
      Write('Enter directory: ');
      ReadLn(DS);
      S:= Fsearch(S,DS);
      If S= '' Then WriteLn('File not found...');
      Until S > '' ;
Assign(f,S);
GetFAttr(f,Attr);
If ((Attr And ReadOnly) > 0 ) Then
      WriteLn('File is read only')
Else
      WriteLn('File is not read only');
If ((Attr And Hidden) > 0 ) Then
      WriteLn('File is hidden')
Else
      Writeln('File is not hidden') ;
```

```
If ((Attr And SysFile) > 0 Then
      WriteLn('File is system file');
Else
      WriteLn('File is not system file');
If ((Attr And VolumeID ) > 0 ) Then
      WriteLn('File is Volume ID')
Else
      WriteLn('File is not Volume ID');
If ((Attr And Directory ) > 0 ) Then
      WriteLn('File is Directory')
Else
    WriteLn('File is not Directory');
If ((Attr And Archive) > 0 ) Then
      WriteLn('File is Archive')
Else
    WriteLn('File is not Archive');

WriteLn;
WriteLn('Press ENTER...');
ReadLn;
End;

(*************************************)
Procedure ProcessHandling
Begin
ClrScr;
WriteLn('DOS SHELL : Type EXIT to return to program...');
SwapVectors;
Exec(GetEnv('COMSPEC'),'');
WriteLn('DosExitCode = ',DosExitCode);
SwapVectors;
WriteLn;
Write('Press Enter...');
ReadLn;
End;

(*************************************)
Procedure EnvironmentHandling;
Var
      i : Integer;
Begin
ClrScr;
WriteLn('Environment info: ');
```

```
WriteLn('Number of Environment Strings =',EnvCount);
For i := 1 To EnvCount Do
      WriteLn(i:2,': ',EnvStr(i));
WriteLn;
WriteLn('COMSPEC =',GetEnv('COMSPEC'));
WriteLn('PATH = ',GetEnv('PATH'));
WriteLn;
Write('Press ENTER...');
ReadLn;
End;

(*********************************)

Procedure MiscProcs;
Var
      YN : Char;
      DosVer : Word;
      Verify,
      CBreak : Boolean;
Begin
ClrScr;
WriteLn('Other information.');
DosVer := DosVersion;
WriteLn('DOS Version: ',Lo(DosVer),'.',Hi(DosVer));
WriteLn;

GetCBreak(CBreak);
If CBreak Then
      Begin
      WriteLn('Ctrl-Break checking is ON.');
      Write('Turn Ctrl-Break checking OFF? Y/N: ');
      ReadLn(YN);
      If UpCase(YN) = 'Y' Then
          SetCBreak(False);
      End
  Else
      Begin
      WriteLn('Ctrl-Break checking is OFF.');
      Write('Turn Ctrl-Break checking ON? Y/N: ');
      ReadLn(YN);
      If UpCase(YN) = 'Y' Then
          SetCBreak(True);
      End;
```

```
GetVerify(Verify);
If Verify Then
        Begin
        WriteLn(Disk write verification is ON.');
        Write('Turn disk write verification OFF? Y/N: ');
        ReadLn(YN);
        If UpCase(YN) = 'Y' Then
            SetVerify(FALSE);
        End
Else
        Begin
        WriteLn(Disk write verification is OFF.');
        Write('Turn disk write verification ON? Y/N: ');
        ReadLn(YN);
        If UpCase(YN) = 'Y' Then
            SetVerify(TRUE);
        End;

WriteLn;
GetCBreak(CBreak);
If CBreak Then
        WriteLn('Ctrl-Break checking is ON.')
Else
        WriteLn('Ctrl-Break checking is OFF.');

GetVerify(Verify);
If Verify Then
        WriteLn(Disk write verification is ON.')
Else
        WriteLn(Disk write verification is OFF.');

WriteLn;
Write('Press ENTER...');
ReadLn;
End;

(*********************************************)

Begin
DateAndTimeProcedures;
DiskStatusFunctions;
FileHandling;
```

```
EnvironmentHandling;
MiscProcs;
InterruptSupportDemo;
  ProecssHandling;
End.
```

第三部分

Turbo Basic 与其它语言的接口

Turbo Basic 与 Turbo Assemlber 的接口

第六章
Turbo Basic 与 Turbo Assemlber 的接口

Turbo Assembler 与 Microsoft 宏汇编的高度兼容性提供 Turbo Basic 程序设计员一项非常方便的使用工具。这一章，我们将用 Turbo Basic 手册中的一些实例向用户说明如何使 Turbo Basic 和 Turbo Assembler 相互联用，从而扩展 Turbo Basic 的功能。

注意：在此所说的 Turbo Basic 是指 1.0 版或更高版本。

Turbo Basic 提供用户三种调用汇编程序的方法：

- 可以用 CALL 来完成一在线调用(inline call)
- 可以用 CALL ABSOLUTE 来调用内存中的某一特定地址
- 可以用 CALL INTERRUPT 和 Turbo Basic 所提供的中断处理支援以跳到某一特定子程序。

至于选择用哪一种调用方式，则取决于用户是否能方便且有效地将某些特定寄存器的值保留起来，以便将来能再跳回 Turbo Basic 的主程序中。一般而言，CALL INTERRUPT 是较方便的方式，用户仅须保存 SS(堆栈段)和 SP(堆栈指针)即可。至于其它两种方法，则尚须保存 DS(数据段)和 BP(基指针)两个寄存器的值。

"保存寄存器"并不需要将所有的寄存器压入堆栈，尽管那是确保安全性的最突出的方法。简单的程序可能不修改任何寄存器，在这种情况下可能不必采取任何预防手段。

使用"可能"这个词是因为避免使用假设，它尤其是汇编程序设计所担心的。总的来说是一个好主意。尽管用户的 MS－DOS 手册可能专门声明了一个特殊中断不允许改变栈指针或基指针(或任何一个其他寄存器)的内容，但也许并不总是这样。MS－DOS 一旦改变，那么一些中断的组合可能与 MS－DOS 手册中的信息产生矛盾。在这种情况下，稳妥总比事后后悔的好。考虑到危险和增加对新版 MS－DOS 的可移植性，谨慎地保存所需的寄存器，反而不会影响用户子程序的环境。

6.1 传递参数

Turbo Basic 通过堆栈传递参数到汇编子程序。所有这些调用均是远调用，堆栈中最后 4 个字节存放"返回地址"，以便 Turbo Basic 将来能顺利的跳回到主程序中。传递到用户子程序的第一个参数的地址为[SP＋4]每推入堆栈一个寄存器，SP 便增加 2。记住，堆栈地址是由高位向低位作向下增长。

每推入双数(不包括数组)到堆栈中，堆栈的高度则增加 4 个字节，因为 Turbo Basic 共传了这个双数的 2 个字节的基址和 2 个字节的偏移量。这种方式有其优点，以后再详述。

若传递的参数为值(如常数或表达式),则每传递一个也会使得堆栈增加 4 个字节,因为 Turbo Basic 也是将该参数在内存中所在临时地址传递过去。这看起来似乎有点多此一举,但却有以下两个明显的优点。首先,很明显地,所有汇编子程序可用同样方式处理所传递过来的参数;其次,若子程序错误地更改了参数的值,而把该值传递引用,也不至于改变内存的重要区域。

用这种方法使用堆栈可以加强用户子程序的标准化。下面的例子将体现这种方法的优越性。假设整型变量 x%的值恰好为 4,且用户有一个需要传入整数的汇编子程序 MYROUTINE,那么,无论用户用 CALL MYROUTINE(x%)还是 CALL MYROUTINE(4%)引用它,这个子程序将以同样方式精确地工作。如果子程序由 CALL MYROUTINE(4%)引用并试图修改被传递的参数值,那么它将修改存贮整数值 4 的内存临时单元区且不会造成危害。

请注意在第二种方式(4%)类型被明确地声明。这不是绝对必须的,尽管这是一个好的做法。如果 Turbo Basic 碰巧声明值 4 是一个单精度数,用户子程序就会使用一个错误值(4 字节的单精度数中的 2 个字节),且运行错误。为了确保用户子程序被传入正确的变量类型,最好在子程序每次被调用时说明变量或值的类型。

如果传送一个数组到子程序,堆栈将生长 60 个字节——其中绝大多数被传送的信息用户可能用不着。Turbo Basic 手册推荐用户以整数形式传递相关的数组参数来代替传递整个数组。传递一些经过选择的参数而不是整个数组可以节省栈空间,减少调用用户子程序所需的时间,且可使得用户程序对以后新版 Turbo Basic 可移植。

例如,假设用户有一仅需将基址指针 BP 推入堆栈的简单子程序。这样,第一个参数的值或地址将在[SP+6]处。如果需要推入 2 个寄存器,第一个参数的地址或值将在[SP+8]处。

假设第一个参数是一个整型值且被以值形式传到汇编子程序。这样,用户可以只简单地写出如下程序以将整型值存入 CX 寄存器:

```
push   bp                ;save the base pointer
mov    bp,sp             ;make the base pinter equal the stack pointer
les    di,[bp+6]         ;ES contains the segment, DI the offset of the value
mov    cx,es:[di]        ;now put the value into CX
```

注意:值不象普通变量那样存放在相同段中,用户必须小心使用正确完整的地址以访问值。后面将更多地说明不在当前数据段中的变量。

另一方面,如果用户知道整型变量已经通过引用而不是通过值进行传递,[BP+6]将包含在数据段中变量的偏移量地址。为了将那个整数值存入 CX 寄存器,用户可以写出如下程序:

```
push   bp                ;save the base pointer
mov    bp,sp             ;make the base pointer equal the stack pointer
mov    bx,[bp+6]         ;put the address of the value in BX
mov    cx,bx             ;put the passed value into CX
```

这个子程序假设变量定位于当前数据段,而且只有在该段中变量的位移量需要更新变量的值。

如果假设用户总是进行传值操作,那么传递变量是安全的。如果实际上变量是通过引用

来传递的，用户不会丢失任何信息，完整的变量地址将包括当前数据段。另一方面，如果用户子程序假设变量是通过引用传递的，但事实并非如此的话，用户得到的地址将不会是正确的完整地址，因为段基址是错的。因此，子程序要么检索错误的值，要么当用户试图改变所传递的变量的值时，以不可预测的结果去改变内存的不正确存贮区。

通过引用传递变量是非常简单的，象字符串、数组和浮点数变量。这些变量如果真的通过堆栈传递可能会因过长而引起错误。此外，从堆栈中读一个长的变量，所花费的时间开销与先从堆栈中取得地址再在内存区内对变量进行操作几乎差不多。对串变量(除非实际上很短)来说，不进行任何内存访问操作似乎不可能有足够的可利用寄存器空间对串来进行处理。

6.1.1 不在当前数据段的变量

如果传递的变量不在当前数据段，那么用户在自己的汇编程序中必须同时具有该变量的段地址和位移量，以存取其值。Turbo Basic 总是在堆栈上同时传递每个变量的段基址和位移量；因此，程序员总可以利用每个变量的完整地址。

参数地址的段基址部分放在紧跟偏移量之后的 2 个字节。在用户的汇编程序中欲取得这个信息，最方便的方法是通过指令 LES。

LES 将把变量地址的位移量值装入被说明的寄存器中，并将地址的段地址部分装入 ES 寄存器。这就保证了无论变量在哪个数据段，用户都可获得变量的全地址。

另外，假设用户子程序需要将一个整型变量的值存入 CX 寄存器。用户可使用 LES 指令，这是由于 EX 寄存器不需保存，如下例程。

```
push   bp                    ;save the base pointer
mov    bp,sp                 ;make the base pointer equal the stack pointer
les    di,[bp+6]             ;ES contains the segment, DI the offset
mov    cx,es:[di]            ;put the value of the variable in CX
```

通过传递每个变量的全地址，Turbo Basic 使得汇编程序员编写与其数据存贮区相独立的子程序成为可能。用户如果要重写程序并将变量或数组存入其它不同的数据段，那么只要使用完整的变量地址和 LES 指令，就不必重写该汇编子程序。

6.1.2 什么类型的调用?

有两种类型的 CALL(调用)：远调用(far)和近调用(near)。远调用离开了当前代码段；而近调用却不离开。

在 Turbo Basic 中只有 CALL ABSOLUTE 会引起麻烦，因为它可定位于内存任何位置，因此，Turbo Basic 要求 CALL ABSOLUTE 子程序以一个远调用作为结束，而且当控制转到这样的子程序时，其会自动产生一个远调用。

当使用 CALL INTERRUPT 时，Turbo Basic 多半也会自动帮用户做好一个相对应的远程调用。如果用户自己设计了中断处理程序，需要使用一个 IRET(return from interrupt 从中断返回)指令将控制权交回 Turbo Basic 程序。

在线汇编程序是在用户程序编译时插入到其中的。其代码一般将存于当前代码段，但 Turbo Basic 并不如此假定。这种子程序也返回一个远调用作为结束。Turbo Basic 自动产生

CALL 和返回指令，因此，不需在自己的代码后面使用一个 RET 指令。如果想在代码结束前的某处终止该子程序，只需跳到在代码结束处的一个标号即可。

注意：因为 Turbo Basic 并不是使用 DOS LINK 来链接用户的程序，用户不必关心子程序是否需要声明 PUBLIC，也不必在自己的程序中声明它们为外部的。

6.2 弹出堆栈

在子程序结束前，用户必须确信所有被压入堆栈的寄存器都已从堆栈中弹出。这方面很容易出现错误，特别是当用户子程序有条件地压入(PUSH)和弹出(POP)寄存器时。

如果用户弹出较少的寄存器，则在被调用后，用户的子程序也许永远无法返回，因为 Turbo Basic 假设堆栈中最后一项是其应该返回的地址。如果弹出过多的寄存器，也会发生同样的问题。

务必不要装载或弹出无用值到段寄存器中，因为这会使得用户源代码与将来的 DOS 新版本(例如，保护模式 OS/2)不兼容。

6.3 为 Turbo Basic 创建一个汇编程序

如果用户创建了一个汇编程序，并想将其转换为.COM 文件以供一个 Turbo Basic 程序使用，仍可使用 Turbo Basic 手册中的批处理文件的示例：

```
TASM %1;
Tlink /t %1;
```

这里不能包括堆栈段，因为 Turbo Basic 在运行时会提供适当的堆栈给汇编子程序使用。

如果用户在程序开始处，没有提供一个明确的 ORG 100h 语句，Turbo Assembler 将会预设程序的执行起始地址在 100h。然而，为了以后的引用，最好还是明确地声明 ORG。

如果用户子程序打算在一个 8086、80286 或 80386 处理器上运行，可在汇编代码开始处使用.186、.286 或.386 伪指令标明处理器的种类。然后，Turbo Assembler 将允许用户使用为这些处理机所提供的指令。用户在以后将会看到这样做的好处。

6.4 调用一个在线汇编子程序

假设用户使用 Turbo Assembler 创建了一个汇编子程并已经转换成.COM 文件，那么便可在用户的 Turbo Basic 程序中以插入伪指令 $INLINE COM 或 $INCLUDE 两种方式来做在线调用。

最简单的插入方式是使用“$INLINE COM 文件名”这种方法，让 Turbo Basic 将.COM 文件插入到用户指定的地方。这种方法相对来说比较容易实现，但有几个缺点：

- Turbo Basic 中每个过程最多只能使用 16 个 $INLINE 伪指令。如果用户要编写比较复杂的程序，可能会引出一些问题(但似乎相当少见)。
- 一个更加严重的问题是.COM 文件不包含注释。当然用户也可在相应的调用程序

中注释，但毕竟可读性较低，如果.COM 文件自身包含注释就好多了。

- $INLINE.COM 文件也会增生，故将若干个文件合并为一个文件将很有用，特别是当用户经常一起使用其中若干文件时(这就是使用汇编子程序库的原因；不幸的是产生一个.COM 文件库不太容易)。
- 最后，如果 $INLINE.COM 有稍微的改动，用户便须修改和重新汇编整个文件。所以，改动相当小的话，也会使人恼火。

基于 $INCLUDE.COM 的工作方式，用户也许希望把.COM 文件转换成一个十六进制数的序列，以便用 $INCLUDE 伪指令可将其插入到程序中。象这样的子程序也可使用 Turbo Basic 编辑器的 Read File(读文件)命令(Ctrl-KR)从盘上读到；那样的话，用户的源文件将明确地表明那个文件被包括。这对 Turbo Basic 程序员来说有很大的好处。

用户可编辑十六进制代码文本，以包含或添加注释。用户也可使用 Turbo Basic 编辑器对在线代码做小小的修改而不需进行重新汇编，用户还可将几个子程序并入一个文件。通过结合这几种技术，用户可以有效地创建一个汇编子程序库供一族程序使用。也许用户维护这样的一个库比使用一个正规的库管理程序还容易。

如果子程序太长，相应的十六进制代码文件就会非常大，以致使源文件不能舒适地编辑。用户可编辑的一段文件最大不能超过 64KB。如果发生这个问题，用户可以象一个 $INCLUDE 文件那样把十六进制文件合并到用户程序中(无论如何，过大的文件不可能使用户程序可读性高)。

以下是一个将.COM 文件转化为十六进制文件的 Turbo Basic 小程序：

```
'COM2INC.BAS
'This program converts COM files to $INCLUDE files with the Turbo
'Basic $INLINE meta-command for easy insertion in Basic programs.
DEFINT A-Z
'All variables will be integers
F$=COMMAND$
'Check to see if there's a command line
WHILE F$=""
  PRINT"This program will convert COM files to $INCLUDE files"
  PRINT"for use with Turbo Basic. The default file type of"
  PRINT"the source file is COM. The default file type of the"
  PRINT"output file is INC. You may override either default"
  PRINT"by entering a specific file-type specification."
  PRINT"if you enter no name for the output file, it will be"
  PRINT"named the same as the input file, but will have a file"
  PRINT"type specification of INC."
  LINE INPUT "Enter the name of the file to convert: ";F$
WEND
IF COMMAND$="" THEN
  LINE INPUT"Enter the name of the desired output file: ";O$
END IF
IF INSTR(F$,".")=0 THEN F$=F$+".COM"          'fix input spec
```

```
IF O$ ="" THEN
  O$ =LEFT$(F$ ,INSTR(F$ ,"."))+"INC"              'fix output spec,
  ELSE
    IF INSTR(O$ ,".")=0 THEN O$ =O$ +".INC"   'both ways
ENDIF
OPEN"R",#1,F$ ,1          'input file will be read one byte
FIELD #1,1 AS A$          'at a time into A$
LASTBYTE&=LOF(1)          'end of file position
OPEN"O",2, O$             'output file is opened
FOR I&= 1 TO LASTBYTE&-1
  GET 1,I&
  X%=ASC(A$)
  IF ((I&-1) MOD 5=0) THEN PRINT #2,"";PRINT #2," $INLINE ";
  PRINT #2,"&H";HEX$(X%);
  IF ((I&-1) MOD 5<>4) THEN PRINT #2,",";
NEXT I&
GET 1,LASTBYTE&
PRINT #2,"&H",HEX$(ASC(A$))
PRINT"Conversion complete. ";LASTBYTE&;" bytes read."
PRINT O$ ;" contains "; LOF(2);" bytes."
CLOSE
END
```

这个程序将输出每行最多有 5 个十六进制代码组成的文件。每行的行首是 $INLINE 伪指令，产生的文件有足够的空间，使用户可以加入的注释。如果用户想在文件的每行写入较多或较少的十六进制代码，只要将常式 MOD 5 改为 MOD N 即可，这里 N 是个大于或小于 5 的数。

用户如果想对所写的子程序做小小的改动并将其转换成十六进制代码，且改动足够少而文件又是完备的，就可参照上述方法进行修改而不必重新创建整个子程序，

6.5　在内存中安装一个 Turbo Basic 子程序

大致有三种方法可以确定一个子程序在内存中的位置。

- 可以由子程序自己返回其地址。
- 可以把一系列子程序组成一组，由其中一个子程序返回该组所有子程序都适合的地址。
- 可以检查内存中某段地址的内容。

要创建一个由自己返回地址的子程序，用户可以使用如下所示的代码。

```
xy:  mov  ax,cs                ;move the code segment register to AX
     push bp                   ;save the base pointer
     mov  bp,sp                ;and copy the stack pointer into BP
     les  di,[bp+6]            ;ES contains the segment, DI the offset
```

```
        mov  es:[di],ax          ;store the CS value to first parameter
        mov  dx,offset xy        ;get the current offset
        les  di,[bp+0ah]         ;address of second parameter
        mov  es:[di],dh          ;store offset value to second parameter
        jmp  fin                 ; jump around "real" code real code would be
here    fin:  pop  bp            ;restore BP and return
```

用户需要传递两个整数参数到这个子程序的变量：第一个记录代码段基址，第二个记录偏移量。问题是：所有代码在用完一次后就设用了。事实上，这段程序在“正常”状况下，即当程序真正要运行时必须去掉，所以比较费事。

除非想调用的子程序能从用户所作的修改中获得很高的速度，否则，所作修改的时间很可能比所能节省的时间还要多。修改应是高质量的，除非用户子程序完全是可重定位的，否则工作代码应以许多 NOP 开始。

然而，用户仍可确定子程序的地址。如果把几个子程序组成一组并在 Turbo Basic 程序中设置了标号，则用户可调用其中任何一个，这样不就允许再用其它子程序时包含了一个“告诉用户地址”的子程序了吗？

事实并非如此。记住：Turbo Basic 为用户处理 RET 伪指令。因为子程序被赋以不同的名字，Turbo Basic 将假设每一个子程序都是可重定位代码。无法保证分开的子程序在最终的.EXE 文件会定位于相同的内存区域。即使子程序都在同一内存区域且有相同次序，用户也不会知道 Turbo Basic 在它们之间插入了多少字节的代码，因而不会知道在何处进行修改。

第三个确定子程序地址方法是签名法(signature)。使用这种办法，不断地检查内存各地址的内容，直到该内容就是用户已经作的特殊记号(即签名)时，该地址便是所需要的子程序。

签名法也有缺点。首先，这样的搜索将花费大量的时间。其次，即使找到了签名也不能保证一定找到指定的子程序。第三，每个子程序必须包含一个与众不同的签名，这既浪费空间又浪费时间。

为了编写能修改的子程序，需要一个定位子程序位置的好方法，并需要一个改变子程序指令的简单方法。

阅读下一节，将找到这些问题的解决办法。在该节将讨论一种用户能在自己的 Turbo Basic 程序中修改所用子程序的特殊方法。

6.5.1　隐藏串

Turbo Basic 允许最大为 64KB 的串空间。有时用户要用到这个空间的每个字节。但是引用串常量(如被用于菜单和提示符的引用串常量)也占用串空间。

然而代码空间却最多能有 16 个段，每个段最大为 64KB。如果能允许用户把那些引用串常量存在代码段中就太好了，这样就不会减少动态串数据可利用的空间。幸运得很，这一点并不难做到。

考虑下面子程序：

```
;This routine takes two integer parameters and returns
;the segment and offset of the text in the body of the program.
;
push   sp
mov    bp,sp
mov    dx,offset show              ;location of string
mov    ax,cs                       ;code segment to AX
les    di,[bp+6]                   ;ES:DI point to parameter
mov    es:[di],dx                  ;report string location
les    di,[bp+0ah]                 ;next parameter
mov    es:[di],ax                  ;report the code segment
jmp    fini                        ;and go back show
DB     'Any text we like here and as much as we want'
DB     'For as long as we want, terminated with any'
DB     'character we like. Here, a null.',0
fini pop bp
```

这个子程序的结果与前面所提的可修改程序在线代码有点不同。其中之一就是用户不是存贮代码(尽管 Turbo Basic 会将这个.COM 文件全部视为代码来处理),而是存贮数据。

这个子程序返回它所存入数据的当前地址。如果用户想知道数据的长度,也可再传入第三个整数参数,使它返回数据的长度。

既然用户已经知道文本在内存的地址,因此只要想打印这个信息时,就可以用 PEEK 指令将串数据读入到串空间。打印完后,现在这个串就已经不再需要,可以把它丢弃,如果用户又需要它时,再到代码空间寻找即可。

用户可以决定在这个子程序中可用的字节数,也可以决定在正文前面的字节数。只要将最后一个指令换成自己的代码:因为用户已经知道子程序的位置、大小,所以可用 BLOAD 覆盖它。虽然此时整个子程序已经不太相同,但是 Turbo Basic 的.EXE 文件并不会有任何改动。

通常,并没有必要使用上述技巧,有时将串存贮在代码空间虽然有用,但 CALLABSOLUTE(绝对调用)能更好地用一个子程序代替另一个子程序。

6.5.2　绝对调用(CALL ABSOLUTE)

CALL ABSOLUTE 在一般 Turbo Basic 手册中较少提到。这是因为,第一,Turbo Basic 对这种子程序的控制较弱。其次,这种子程序是为 Basic 解释器的使用而编写的,而 TurboBasic 与 Basic 解释器的不同之处如此之多以至那些子程序可能无法运行。其三,未来的操作系统可能不允许使用 CALL ABSOLUTE 子程序。尤其是明确区分代码空间和数据空间的操作系统可能不允许处理器执行定位于数据空间的指令。其四,被 CALL ABSOLUTE 调用的子程序仅能传递简单的整型变量。从表面上看,这也许不是一种限制,但简单的整型变量也可包含任何类型变量的段基址和偏移量地址。还有,它似乎会使参数传递更费时间。

对于所讨论的,假设用户使用 MS-DOS 2.0 或更高版本的操作系统,它们允许处理器在内存任何地方执行指令。

6.5.2.1 到一固定内存位置作 CALL ABSOLUTE

如果用户有一族程序共享相同的子程序集，可将那些子程序置于一固定位置(fixed location)。那么，每当需要调用那些子程序的程序就可以根据这一位置直接调用。Turbo Basic 允许用户为此目的使用 MEMSET 命令在内存高地址安全保存这些位置。

ENDMEM 经常与 MEMSET 一起使用。ENDMEM 将返回一个表示被编译后的 Turbo Basic 程序可以使用的内存最后位置的长整数。子程序通常存放在该界限下的内存高地址的某个固定位置。

如果用户有这样一个子程序集，就需要用 CALL ABSOLUTE 命令调用它们。可使用 BLOAD 命令把它们放入内存高地址，用 DEF SEG 设置子程序将被装入的段基址，同时需特别声明子程序将被装入的地址偏移量。

当用户用 Turbo Assembler 创建这些子程序时，应该留心以下规则：

1. 除非打算在一个且仅仅一个地址处运行子程序，所有程序中的转移(即所有 JMP 和 CALL)应是完全可重定位的(关于可重定位代码完整的讨论已经超出了本章的范围)。
2. 如果只打算在一个地址处运行程序，用户应在汇编源代码中的 ORG 伪指令中声明这个地址。

6.5.2.2 到内存不定位置作 CALL ABSOLUTE

Turbo Basic 也允许用户在每次运行一个程序都可能改变的那些内存位置使用 CALLABSOLUTE。进行这种操作最典型的方法是将汇编子程序安装到普通数据区外的一个数组中。

考虑下述代码段。

```
DEFINT a~z
$DYNAMIC                                    'arrays will be dynamic
DIM RoutineArray(10000)                     '20,002 bytes allocated
                                            'miscellaneous code here
whereseg% = VARSEG(ROUTINEARRAY(0))         'segment address
whereoffset% = VARPTR(ROUTINEARRAY(0))      'offset address
DEF SEG = whereseg%                         'set default segment
BLOAD "COMFILE", whereoffset%               'read routine in
CALL ABSOLUTE whereoffset%(parameter%)      'call the routine
DEF SEG                                     'return to default
```

如果用户要使用多个子程序，可以依次将它们装入到同一数组中。如果需要使用子程序的某一版本，用户可以选择装入，并把每个不同版本的子程序装入到不同数组中。最后，如果要改变数组的成分，仅需通过改变所选数组元素的值即可。

正如用户所知，利用 CALL ABSOLUTE 所设计的子程序远比利用 $INLINE 所设计的容易寻找和修改。使用 CALL ABLSOLUTE 子程序的困难在于如果要使它们成为通用，就必须使它们成为完全可重定位的。对简短的子程序来说，这也许不成问题，但对较复杂的子程序，编写完全可重定位的代码就不是那么容易了。

用户也可将子程序 BLOAD 到串变量中。此时，用户必须特别小心，如果用户试图将一个比串变量长的子程序 BLOAD 到串变量中，就会覆写一些其他串。如果该串是可修改的，

那么被 BLOAD 的子程序也可能被部分修改。

串变量也可移动。即使子程序已被正确地装入一个串中，在准备调用子程序前，用户也应该小心地使用 VARSEG 和 VARPTR 来建立串的地址，以确保万一。

Turbo Basic 的串存贮方式与数值变量的存贮方式不同。如果执行 VARPTR(A%)操作，可以得到整型变量 A%的地址。如果执行 VARPTR(A$)操作，则可得到 A$的串描述符(descripter)的地址。此内存位置再下去的两个字节就是该串在串空间的实际地址。为了得到和 VARPTR(A%)类似的结果，必须执行下面形式的程序。

```
A% = VARPTR(A$)
A% = A%+2
StringAddress% = CVI(CHR$(PEEK(A%)) + CHR$(PEEK(A%+1))
```

尽管把汇编子程序插入字符串是很普通的，但因为现在整型数组已经可变大小和任意删除，所以它已不那么具有吸引力了。如果在 CALL ABSOLUTE 子程序中使用整型数组，就可避免因经常移动串数据而带来的额外困难。

用户如想避开使用 BLOAD，也可以通过使用二进制文件的 I/O 将.COM 文件装入到串中，即以二进制类型可打开.COM 文件，然后读入正确数量的字节到串中。用同样办法也可将数据读入整型数组中。不过使用 BLOAD 既快又简单！

6.5.2.3 CALL ABSOLUTE 的其他问题

用 CALL ABSOLUTE 从盘上读出的代码有几个重要缺点，其中最大的缺点是代码必须具备我们先前所提到的可重定位的特性。

另一个严重的问题是子程序必须独立于主程序而从盘上个别读出，这就导致了一些潜在的错误。诸如所需的代码可能不在盘上，或是在盘上却已受损。

第三个问题是，必须花费额外的时间从盘上读取代码，而争取时间却正是我们要以汇编语言(而不是 Turbo Basic)来编写子程序的主要原因。

尽管有这些磕磕绊绊的障碍，但由于 CALL ABSOLUTE 的种种优点，仍可由不同子程序内读取；可在程序控制之下修改代码以及可降低在任何时候都必须贮存在内存之代码的数量。这足可以成为考虑使用 CALL ABSOLUTE 指令的理由。

6.6 调用中断

第三个，也是最后一个从 Turbo Basic 中访问汇编子程序的方法，也许它是避开汇编最容易的办法，又是使用汇编最困难的办法。

多数程序员都使用 CALL INTERRUPT 来取得普通的 MS-DOS 服务。在这种情况下，确实不必担心汇编问题，只需记住下页表即可。

要设置寄存器的值，使用 REG 指令：

```
REG  3, @H0F01
```

它把 CX 寄存器值设置为十六进制数 0F01。CH 寄存器值将为十六进制数 0F，而 CL 将为 01。

要读一个寄存器的值，使用 REG 函数：

```
A% = REG(3)
```

它将 CX 寄存器的当前值赋给 A%。

下面的例子使屏幕做反向翻页(reverse scroll),从第 1 行到第 24 行:

```
REG 3,0                    'row zero, column zero for top.
REG 4,&H175F               'row 23, column 79 for bottom
REG 2,&H70                 'color 7,0
REG 1,&H0701               'bios function 7, scroll 1 line
CALL INTERRUPT &H10        'video interrupt 10h
```

名字	寄存器
REG0	Flags
REG1	AX
REG2	BX
REG3	CX
REG4	DX
REG5	SI
REG6	SI
REG7	BP
REG8	DS
REG9	ES

用汇编写同样的子程序要困难得多且运行效率并不如它。事实上,CALL INTERRUPT 形式在需要时更容易修改。

整个过程通常非常简单。然而,对更高级的程序员来说,中断可以有比正常 MS-DOS 服务有更多的其它用途。

中断经常用于管理设备(如温度传感器、遥控记录仪、定时器和采样器)。使用服务于上述目的的这种中断,必须先找到一个未被使用过的中断(很多中断被 MS—DOS 所用,其他一些中断可能被象磁带机和 Bernoulli Box 这样的存贮设备所利用)。

在 Turbo Basic 程序中,用户可将中断向量指向用 Turbo Assembler 写的子程序。象 Turbo Basic 手册中指出的,中断子程序应保存 SS 和 SP 寄存器的值,其它的寄存器均可修改。在子程序的末尾,通过 IRET 指令将控制传回给 Turbo Basic 程序。

可以用已经提到过的方法来确定一个子程序的地址并将这个地址存入中断向量,但最好把中断子程序放在内存的高地址或象 CALL ABSOLUTE 对子程序那样 BLOAD 它们。

通过 $INLINE 命令包含在用户程序内的中断子程序需要被定位。虽存贮于整型数组中被 BLOAD 的子程序可能不时地改变地址,但至少其地址是可知道的。还有,把中断处理子程序装入这样的数组就意味着中断子程序中所有代码必须是完全可重定位的。

由于这个原因,中断子程序通常被放在内存高地址的固定位置。用户如决定使用这种方法,务必在自己的 Turbo Assembler 源代码中包含适当的 ORG 命令。

6.7 样本程序

```
FILLIT2$ = CHR$(&HFC) + CHR$(&HF3) + CHR$(&HAB) + CHR$(&HCB)
```

```
'          cld          rep          stosw          ret
DIM a%(100)                                    'integer array whose elements are all zero
WHERE%=VARPTR(FILLIT2$)                        'this locations stores the length
WHERE%=WHERE%+2                                'and this is where the string location is
CLS:PRINT PEEK(WHERE%),PEEK(WHERE%+1)
HERE%=PEEK(WHERE%)+256*PEEK(WHERE%+1)          'and this is the string location
DEF SEG                                        'not necessary here, but good programming
                                               'practice
WHERE%=PEEK(0)+256*PEEK(1)
DEF SEG=WHERE%                                 'string segment is the first word in
                                               'default DS
REGES%=VARSEG(a%(0))
REGSI%=VARPTR(a%(0))
REG 1,5%                                       'put the fill value into AX
REG 3,101%                                     'number of elements to fill, 0 to 100
                                               'inclusive into CX
REG 9,REGES%                                   'segment of the array to fill into ES
REG 6,REGSI%                                   'offset to first array element into SI
CALL ABSOLUTE HERE%                            'fill the array with the value 5
DEF SEG
FOR i%=0 To 100:PRINT a%(i%);:NEXT i%
PRINT
PRINT REG(1),REG(3),REG(9),REG(6):STOP
CALL FILLIT(a%(0),-1%,101%)                    'fill the array with the value -1
FOR i%=0 TO 100:PRINT a%(i%);:NEXT i%
PRINT
END
SUB FILLIT INLINE
$INLINE  &H55,&H8B,&HEC,&HC4,&H7E
$INLINE  &HE,&H26,&H8B,&HD,&HC4
$INLINE  &H7E,&HA,&H26,&H8B,&H5
$INLINE  &HC4,&H7E,&H6,&HFC,&HF3
$INLINE  &HAB,&H5D
END SUB
;Routine to transfer an arbitrary number of elements with an arbitrary value
;into an integer array for call absolute. Calling syntax is:
;REG 1,FILLVALUE%          'AX has the fill value
;REG 3,FILLCOUNT%          'CX has the number of elements to fill
;REG 9,VARSEG(ARRAY(0))    'ES has the segment of the array
;REG 6,VARPTR(ARRAY(0))    'DI is the offset to first array element
;CALL ABSOLUTE FILLIT2     '
;FILLIT2 is the address of the absolute routine and DEF SEG will have setthe
;default program segment to that of FILLIT2 before the CALL ABSOLUTE.
```

```
PROGRAM   SEGMENT
START     PROC   FAR                ;this will force a far return
    ASSUME cs:PROGRAM
    push   bp                      ;save the base pointer
    cld                            ;clear direction flag
    rep                            ;next instruction repeats until CX is 0
    stosw                          ;store AX to ES:DI and increment DI by 2
    pop    bp                      ;restore base pointer
    ret                            ;intersegment (far) return
START ENDP
PROGRAM ENDS                       ;end of segment
    END
;Routine to transfer an arbitrary number of elements with an
;arbitrary value into an integer array. Calling syntax is:
;CALL FILLIT(ARRAY(0),FILLVALUE,NUMTIMES)
    ORG 100h
PROGRAM SEGMENT
    ASSUME cs:PROGRAM
    push   bp                      ;save the base pointer
    mov    bp,sp                   ;mov stack pointer to BP
    les    di,[bp+0eh]             ;get offset address of # of elements to fill
    mov    cx,es:[di]              ;number of elements to fill into CX
    les    di,[bp+0ah]             ;get offset address of fill value
    mov    ax,es:[di]              ;put fill value in AX
    les    di,[bp+6]               ;offset address of array to fill
    cld                            ;clear direction flag
    rep                            ;next instruction repeats until CX is zero
    stosw                          ;store AX to ES:DI and increment DI by two
    pop    bp                      ;restore base pointer
PROGRAMENDS                        ;end segment -- no RET instruction
    END
```

第四部分

Turbo Prolog 与其它语言的接口

Turbo Prolog 与 Turbo C 接口
Turbo Prolog 与 Turbo Assembler 的接口
Turbo Prolog 与 MS-Fortran 4.0 的接口
Turbo Prolog 访问 dBASE Ⅲ 数据文件
Turbo Prolog 与 DOS 系统级的接口

第七章 Turbo Prolog 与 Turbo C 接口

Turbo Prolog 是基于逻辑程序设计的语言。从 1.0 版、1.1 版到 2.0 版，已有了很大的发展，成为开发专家系统等知识库系统的有力工具。

尽管 Turbo Prolog 在许多方面是一个非常好的工具，但是仍然需要使用其它语言。例如用 Pascal 或 Fortran 语言很容易完成数值积分运算，用汇编语言实现中断处理及低层操作更好。另外，如果用其它语言开发的程序已经解决了某些问题，这些工作不应被抛弃和否定。因此，Turbo Prolog 允许 Turbo Prolog 程序与其它语言连接。下面将介绍 Turbo Prolog 与其它语言的接口约定、Turbo Prolog 与 Turbo C 的接口、Turbo Prolog 与 MS-Fortran 4.0 的接口、Turbo Prolog 与汇编语言的接口。此外，还介绍了 Turbo Prolog 访问 dBASE Ⅲ 文件的方法，以及 Turbo Prolog 与 DOS 系统级的接口谓词。

在调用其它语言写的子程序和函数前，需要在 Turbo Prolog 中把它们说明成外部谓词，还应了解正确的调用约定和参数压栈顺序，以及如何命名外部谓词的不同流变体。下面三节分别叙述这三个问题。

注意：当提到 Turbo Prolog 时都是指 1.0 或更高版本。

7.1 声明外部谓词

Turbo Prolog 允许通过使用一个全局谓词(global predicates)声明与其他语言进行接口。在声明后面附加一个语言说明，以便使 Turbo Prolog 能知道这个全局谓词是用哪一种语言实现的。

```
global predicates
    add(integer,integer,integer) - (i,i,o),(i,i,o) language asm
    scanner(string, token) - (i,o) language Pascal
```

Turbo Prolog 使接口语言明确化能简化活动记录和参数格式、调用和返回约定、段定义、链接和初始化等问题。

7.2 调用约定和参数压栈顺序

8086 系列处理器使程序员能够对近子例程调用和远子例程调用进行选择。Turbo Prolog 创建大存贮模式程序，并要求所有对子例程的调用和从子例程返回都是远调用。

Turbo Prolog 支持多种调用约定，包括 C、Pascal 和 Assembler 等等。当使用 C 调用约定

与子例程接口时，参数以反序压入堆栈，返回后栈指针自动调整；当与其他语言接口时，参数以正常次序入栈，被调函数负责从栈中取走参数。

在许多 8086 系列的语言编译程序中，可以选择 16 位指针或 32 位指针，16 位指针指向的是默认段，而 Turbo Prolog 总是使用 32 位指针来访问所有内存。

Turbo Prolog 的类型以下列方式实现：

整型	2 字节
实型	8 字节(IEEE 格式)
字符型	1 字节(压入堆栈时用 2 字节)
字符串型	4 字节　双字指针指向一个以 NULL 结尾的串
符号型	4 字节　双字指针指向一个记录。

输出参数以指向某地址的 32 位指针形式压入堆栈，返回值必须赋给指针指向的单元。对输入参数，其值直接入栈，并且参数的大小取决于它的类型。

7.3　命名约定

在 Turbo Prolog 中，一个谓词可以有多个类型的变体和多个输入输出流变体，每个类型和流变体有其各自的子程序。为了调用这些不同过程，必须为每一过程赋以唯一的名字，这是通过从 0 开始对具有相同谓词名的不同子程序向上进行编号来实现的。

例如，给出如下声明：

```
global predicates
    add(integer, integer, integer) - (i,i,o),(i,i,i) language asm
```

第一个变体[流模式为(i,i,o)]被命名为 add　0，第二个变体[流模式为(i,i,i)]命名为 add 1。

Turbo Prolog 还允许程序员为全局谓词声明一个明确的名字，这是通过在声明后跟"as-public name"来实现的。下例中，全局谓词 Pred 将用 my　pred 而不是 pred　0 来声明：

```
global predicates
    pred(integer, integer) - (i,o) language asm as "my  pred"
```

这种方法在用户命名只有一种流模式的谓词时采用。如果存在多种流模式，则用户只能为每个变体声明一个名字。以 add 谓词为例，谓词定义可能如下：

```
global predicats
    add(integer,integer,integer) - (i,i,o) language asm as "doadd"
    add(integer,integer,integer) - (i,i,i) language asm as "add  check"
```

第一个变体[流模式为(i,i,o)]被命名为 doadd，第二个变体[流模式为(i,i,i)]被命名为 add　check。注意这种命名方法需要分别对各变体进行声明。

7.4 Turbo Prolog 调用 Turbo C 过程

7.4.1 说明外部谓词

在 Turbo Prolog 主模块中，应说明被调用的 C 函数为全局谓词。下列的 Turbo Prolog 程序说明了如何访问一个用 Turbo C 编写的过程 double。可用如下的 Turbo Prolog 语句调用该过程：

```
double(MyInputInteger, MyOutVar)
```

在调用前 MyInputInteger 约束为一个整数值，调用后 MyOutVar 被约束为该值的两倍。

```
global predicates
    double(integer,integer) - (i,o) language C
goal
    double(5,N),
    write("5 doubled is ",N).
```

在包含调用 double 的 Turbo Prolog 程序中，必须在全局谓词段中指明实现该过程的语言。

7.4.2 建立 C 函数源程序

在 C 函数源程序中，必须按 Turbo Prolog 命名约定为 C 函数命名，即 C 函数名就是 Prolog 的谓词名紧接一下划线和与一流模式对应的整数。

对于上例的 double 有如下的 C 程序：

```
void double_0(int in, int *out)
{
    *out = in + in;
}
```

7.4.3 Turbo C 编译选项和连接

为了连接 Turbo Prolog 与 Turbo C 模块，必须带有下列选择项编译 C 源程序。

(1) 用 TC.EXE 编译：

<1> Options/Compiler/Model/Large

<2> Options/Compiler/Optimization/Jump optimization ... On

<3> Options/Compiler/Code generation/Generate underbars ... Off

(2) 用 TCC.EXE 编译：

-ml -o -u- -z-c

(3) 连接程序模块：

Turbo Prolog 和 Turbo C 应按下列通用命令行连接：

tlink init<pOBJS><cOBJS><.sym>,[exe],[map],[usr]+prolog[+emu+mathl+cl]

该命令行各个参数的意义如下表所示：

参 数	功 能
tlink	调用 TLINK，即 Borland 的 Turbo 连接程序
init	Turbo Prolog 的初始化文件
pOBJS	Turbo Prolog .OBJ 模块
cOBJS	Turbo C .OBJ 模块
.sym	Turbo Prolog 的符号文件须为第一个逗号前的最后一个文件
exe	(可选)可执行文件名
map	(可选)map 文件名
usr	(可选)用户库文件列表
prolog	指 PROLOG.LIB
emu	(可选)C 浮点仿真库 EMU.LIB
mathl	(可选)C 大模式数学库 MATHL.LIB
cl	(可选)C 大模式库 CL.LIB

假设 PDOUB.PRO 已经编译成.OBJ，连接 PDOUB.PRO 和 CDOUB.C 的正确命令为：

```
tcc -ml -o -u- -r- -c cdoub
tlink init pdoub cdoub pdoub.sym,pdoub,,prolog
```

如不用 TLINK，可创建一个工程定义文件 CPROJ.PRJ，在工程文件中给出两个模块名：

```
pdoub
cdoub.obj
```

注：需给 C 目标文件一个扩展名，如果在 C 文件名后不加扩展名，Turbo Prolog 的编译程序将给出错误信息，系统不能成功地进行连接。

创建工程定义文件后要做的是将 Prolog 模块说明为工程的一部分，如果 C 模块已经编译成 CDOUB.OBJ，只要选择 Compile/EXE file(程序 PDOUB.PRO 必须已装入 TurboProlog 编辑器中)，Turbo Prolog 系统将自动完成所有连接。如果需要用到外部库文件，可以在 O/L/Libraries 菜单项中指明。

```
project "cproj"
global predicates
    double(integer, integer) -(i,o) language c
goal
    double(5,N),
    write("5 doubled is ", N).
```

7.4.4 动态存贮分配

用其他语言编写函数经常需要分配动态存贮区。由于 Turbo Prolog 控制了程序的所有存贮资源，所以不能使用 Turbo C 的存贮分配子程序。

但是，在 Turbo Prolog 库中实现了子程序 malloc 和 free(C 函数 malloc 和 free 的 Turbo Prolog 版本)。连接 Turbo Prolog 和 C 时，如果 Turbo Prolog 库 PROLOG. LIB 在其它 C 库前出现，就从 PROLOG. LIB 中取这些子程序，而不是从 C 库中取：

```
void *  malloc(unsigned size);
void *  free(void *);
```

如果想在 Turbo Prolog 的全局栈中分配单元，可以通过调用库子程序 alloc_gstack 实现：

```
void *alloc_gstack(unsigned size);
```

alloc_gstack 返回一个指向长度为 size 的存贮块的指针，存贮区在 Turbo Prolog 全局栈中分配。使用 alloc_gstack 失败时，存贮区将自动被释放，使得 Turbo Prolog 回溯越过存贮分配。

7.4.5　传递复合对象到其它语言的程序

当 Turbo Prolog 和其它语言连接时，可以直接传递简单数据类型的参量(integer、real、char、symbol 和 string)。本节将简要介绍如何传递 Turbo Prolog 的复合对象。

(1) 传递表

在 Turbo Prolog 中，可在 domains 将字符串表说明为 strlist：

```
domains
    strlist = string *
```

这说明了一个 string 类型的表，该表在内存中表示为一链表，链表的每个记录包含三个域：

域	长　度
表函子	1 个字节
表元素	对应于元素的数目
指向下一个的指针	4 个字节

对于 strlist 的域用 C 数据结构可以描述为：

```
struct node
    { unsigned char functor;
      char *value;
      struct node *next;
    } strlist;
```

functor 域指出表记录的类型：如果是表元素为 1；表尾则为 2。

(2)传递结构

Turbo Prolog 复合对象作为记录实现，记录的第一个元素包含与 Turbo Prolog 域说明相对应的函子数，其余对应于复合对象的各个元素。例如，下列代码段中：

```
domains
    mydom = i(integer); c(char); s(string)
```

对第一个 i(integer) 函子数为 1,2 对应于第二个 c(char),3 对应于第三个,等等。

mydom 用 Turbo C 的 typedef 可定义为:

```
typedef struct
    {
      unsigned char type;
      union {
              int i;
              char c;
              char *s;
            }
    } "mydom";
```

以这种方式创建或返回一复合对象时,在对结构的元素赋值前必须先调用 alloc gstack 为对象在全局栈中分配空间。

7.4.6 例　子

1. 两个源程序例子

下面的例子程序给出了在 Turbo C 中创建一个 Turbo Prolog 函数并对 Turbo Prolog 返回其结构代码。该例子也说明了在 Turbo Prolog 的全局栈中如何分配存贮区。

谓词 pack 取一个 integer、一个 char、一个 real 和一个 string 的对象,并把它们合成一个由用户定义的域 mydom 的复合对象。

Turbo Prolog 源程序"MYPACK.PRO"如下:

```
project "mypack"

global domains
  mydom = f(integer, char, real, string)

global predicates
  pack(integer, char, real, string, mydom) - (i,i,i,i,o) language c

goal
  pack(1, 'a', 99.9, "HELLO", X),
  write(X), nl.
```

Turbo C 源程序"pack.c"如下:

```
void *alloc  gstack(unsigned);
typedef struct {
  char functor;   /* Type of functor */
  int ival;       /* Value of the functor */
  char cval;
  double rval;
} mydom;
```

```
pack  0(int P1, char P2, double P3, mydom * *P4 )
{
  mydom *p= *P4=alloc  gstack(sizeof(mydom));
  p->functor=1;
  p->ival=P1;
  p->cval=P2;
  p->rval=P3;
}
```

注意是怎样调用 allog gstack 在全局栈中为结构分配 sizeof(mydom)个字节空间的。

对这两个源程序选择 Compile/Project 和. PRJ 文件 MYPACK. PRJ 进行编译时，Turbo Prolog 系统将连接它们的. OBJ 模块。MYPACK. PRJ 定义如下：

```
mypack
pack. obj
```

2. 表处理例子

下面给出一个更有用的例子说明如何把一个表转换成一个数组，然后再转换成表。

C 子程序 ListToArray 取一整数表，将它转换成存于全局栈中的数组，并返回元素的个数。转换分三步进行：

(1)遍历表以计算元素的个数；

(2)分配所需元素个数的数组空间；

(3)再次遍历表，把元素传送到数组。

C 子程序 ArrayToList 取一整数数组并以数组的大小为参数，然后将之转换成为一整数表。该子程序只做一趟，建立索引数组的表。

外部过程 inclist 说明如何使用 ListToArray 和 ArrayToList。当给定一整数表时，inclist 首先把整数表转换成一个数组，再把数组的元素增加 1，然后把数组再转换回整数表。

```
project "mycnvrt"

global domains
  list = integer *

global predicates
  inclist(list, list) - (i,o) language c
goal
  inclist([1, 2, 3, 4, 5, 6, 7], L), write(L).
```

下面的 C 程序定义了两个 C 过程 ListToArray 和 ArrayToList，以及外部 Turbo Prolog 谓词 inclist：

```
#define alloc  gstack walloc
#define listfno  1
#define nilfno  2

void *alloc  gstack(unsigned);
```

```
typedef struct ilist {
  char Functor;
  int Value;
  struct ilist *Next;
} IntList;

int ListToArray(IntList *List,int **ResultArray)
{  /* Convert a list to an array placed on the global stack */
  IntList *SaveList = List;
  int *Array;
  int  i = 0;

/* Count the number of elements in the list */
  for(i=0; List->Functor==listfno; List=List->Next)
      i++;

  Array = alloc_gstack(i*sizeof(int));  /* Allocate the needed memory */
  List = SaveList; /* Transfer the elements from the list to the array */

  for(i=0; List->Functor==listfno; List=List->Next)
      Array[i++]=List->Value;

  *ResultArray=Array;
  return(i);
}

ArrayToList(int Array[],int n,IntList **List)
              /* Convert an array to a list */
{
  int i;

  for (i=0; i<n; i++)  /* Allocate a record for each element */
  {
      IntList *p=*List=alloc_gstack(sizeof(IntList));
      p->Functor=listfno;
      p->Value=Array[i];
      List=&(*List)->Next;
  }

  {  /* Allocate the last record in the list */
      IntList *p=*List=alloc_gstack(sizeof(char));
      p->Functor=nilfno;
  }
```

```
}

inclist  0(IntList * InList,IntList * * OutList)
{  /* Increment all values in a list */
  int i, n, * Array;
  n=ListToArray(InList,&Array);
  for(i=0; i<n; i++) Array[i]++;
  ArrayToList(Array,n,OutList);
}
```

7.5 Turbo C 调用 Turbo Prolog

下列 Turbo Prolog 程序说明了两个 C 语言全局谓词 message 和 hello_c。message 谓词可以在 C 语言源程序中通过函数名 message_0 调用。

```
global predicates
  message(string) -(i) language c
  hello_c -language c

clauses
  message(S) :-
      makewindow(13,7,7,"",10,10,3,50),
      write(S), readchar( ),
      removewindow.

goal
  message("Hello from Turbo Prolog"),
  hello_c.
```

上述程序的 goal 段调用 Turbo C 函数 hello_c，而 hello_c 调用 Turbo Prolog 谓词 message_0 显示一段信息。

```
void hello_0()
{
    message_0("Hello from Turbo C");
}
```

使用该特征可以很容易从其它语言存取 Turbo Prolog 强功能的库。可以很容易地在 Turbo Prolog 模块中定义自己的库子程序如下：

```
project "dummy"  /* Use your own project name here */

constants
  % Choose the language calling convention, either C or Pascal
  lang = c
```

```
global predicates
  myfail language lang as "fail"
  mymakewindow(integer,integer,integer,string,integer,integer,integer,integer)
    - (i,i,i,i,i,i,i,i) language lang as "makewindow"
  myshiftwindow(integer) - (i) language lang as "shiftwindow"
  myremovewindow language lang as "removewindow"
  write  integer(integer) - (i) language lang as "write  integer"
  write  real(real) - (i) language lang as "write  real"
  write  string(string) - (i) language lang as "write  string"
  myreadchar(char) - (o) language lang as "readchar"
  myreadline(string) - (o) language lang as "readline"

  extprog language lang as "extprog"

clauses
  myfail :- fail.

  mymakewindow(Wno, Wattr, Fattr, Text, Srow, Scol, Rows, Cols):-
      makewindow(Wno, Wattr, Fattr, Text, Srow, Scol, Rows, Cols).

  myshiftwindow(WNO) :- shiftwindow(WNO).

  myremovewindow :- removewindow.

  write  integer(I) :- write(I).

  write  real(R) :- write(R).

  write  string(S) :- write(S).

  myreadchar(CH) :- readchar(CH).

  myreadline(S) :- readln(S).
```

常量 lang 应设置为 pascal,c,Fortran 或 asm。

下面的 C 过程 extprog 演示了这些新的库子程序的使用。extprog 创建一 Turbo Prolog 窗口,然后在窗口中进行读和写操作。

```
extprog()
{
  char dummychar;
  char * Name;

  mymakewindow(1,7,7,"Hello there",5,5,15,60);
```

```
    write  string("\n\nIsn't it easy");
    myreadchar(&dummychar);
    write  string("\nEnter your name: ");
    myreadline(&Name);
    write  string("\nYour name is: ");
    write  string(Name);
    myreadchar(&dummychar);
    myremovewindow();
}
```

从其它语言调用 Turbo Prolog 的唯一限制是 Turbo Prolog 程序必须为主程序，因为 Turbo Prolog 需要设置堆和栈。

第八章 Turbo Prolog 与 Turbo Assembler 的接口

Turbo Prolog 向程序员提供了丰富的谓词，这些谓词提供了从屏幕窗口管理到 B+树数据库管理的一个丰富的高层次函数集。Turbo Assembler 为 Turbo Prolog 增加的是低层程序设计的工具。

本章先概述 Turbo Prolog 与 Turbo Assembler 的接口；接着给出一些汇编例程与 TurboProlog 接口的简单例子；最后讨论从汇编代码调用 Turbo Prolog 谓词，使用 Turbo Prolog 库调用以及传递复合结构。

8.1 声明外部谓词

Turbo Prolog 允许通过使用一个 global predicates(全局谓词)声明与汇编语言进行接口。在声明后面附加一个语言说明，以便使 Turbo Prolog 能知道这个全局谓词是用汇编语言实现的。例：

```
global predicates
    add(integer,integer,integer) - (i,i,o),(i,i,o) language asm
```

8.2 调用约定和参数压栈

Turbo Prolog 创建大存贮模式程序，并要求所有对子例程的调用和从子例程返回都是远调用。在下推参数后，它对外部谓词的调用形式如下：

```
mov ax,SEGMENT data
mov ds,ax
call  FAR PTR external  predicate  implementation
```

因此外部谓词编写的过程在执行时的数据段为 DATA 段。当取接到一个用汇编语言写的过程时，参数以正常次序入栈，并且被调函数负责从栈中取走参数，并在返回时修正堆栈指针。系统在为一外部谓词调用过程前，DS 寄存器被设置成指向 DATA 段。

Turbo Prolog 的数据类型以下列方式实现：

整型	2 字节
实型	8 字节(IEEE 格式)
字符型	1 字节(压入堆栈时用 2 字节)
字符串型	4 字节　双字指针指向一个以 NULL 结尾的串

符号型　　4 字节　双字指针指向一个记录。

输出参数以指向某地址的 32 位指针形式压入堆栈，返回值必须赋给指针指向的单元。对输入参数，其值直接入栈，并且参数的大小取决于它的类型。

8.3 命名约定

在 Turbo Prolog 中，一个谓词可以有多个类型的变体和多个输入输出流变体，每个类型和流变体有其各自的子程序。为了调用这些不同过程，必须为每一过程赋以唯一的名字，这是通过从 0 开始对具有相同谓词名的不同子程序向上进行编号来实现的。同时应注意，对于 i 模式的不同数据类型变体，按照从左到右的高低权数，先说明整型，后说明实型，否则不能实现带整数类型的变体。

8.4 编写汇编语言谓词

最简单的谓词大概是那些仅有输入流模式的谓词了。例如，如果想滚动当前 Prolog 窗口的内容，可以创建谓词 scroll　left 将屏幕的一个区域向左滚动一列。在例子 SCROLLH.PRO 中，scroll　left 包含 4 个整型变量和一种流模式。

Turbo Prolog 模块 SCROUH.PRO 包含谓词 scroll　left 的全局谓词说明。scroll　left 谓词被定义为用汇编语言实现的谓词。

```
/* SCROLLH.PRO */
global predicates
  scroll  left(integer,integer,integer,integer) - (i,i,i,i) language asm
predicates
  scrollh
clauses
  scrollh:-
      makewindow( , , , ,Row,Col,Nrows,Ncols),
      scroll  left(Row,Col,Nrows,Ncols),
      readchar(C),
      char  int(C,CI),
      not(CI = 27),
      scrollh.
goal
  makewindow(1,7,7," A SCROLLING MESSAGE ", 10,20,4,60),
  write("This message will scroll across the window"),nl,
  write("Look at it go!"),
  readchar( ),
  scrollh,
  readchar( ).
```

下面的汇编代码是 scroll　left 谓词的实现。注意赋给该谓词的名字是 SCROLL　LEFT 0，它与前面讨论的命名约定一致（这是必须的）。

```
;  SCROL. ASM
;
     name scrol
;
;      scroll  left(integer,integer,integer,integer) - (i,i,i,i) language asm
;
SCROL  TEXT     SEGMENT BYTE PUBLIC 'CODE'
    ASSUME     CS:SCROL  TEXT
PUBLIC SCROLL  LEFT  0
SCROLL  LEFT  0  PROC FAR
;
; parameters
arg  NCOLS:WORD, NROWS:WORD, COL:WORD, ROW: WORD = ARGLEN
;
; local variable
local     SSEG :WORD = LSIZE
     push  bp
     mov  bp, sp
     sub  sp, lsize              ;room for local variables
     push si
     push di

     mov SSEG, 0B800h

     sub  NCOLS, 3               ;NCOLS = NCOLS - 3

     mov ax, ROW                 ;DEST = ROW * 160+(COL+1) * 2
     mov dx, 160
     mul dx
     mov dx, COL
     inc  dx                     ;added
     shl  dx,1
     add  dx,ax

     push ds
     push es

     mov  bx, NROWS              ;loop NROWS times using BX as counter
     dec  bx                     ;NROWS = NROWS - 2
     dex  bx

Top: cmp bx,0
     je    Done
```

```
        add   dx, 160           ;dest = dest + 160

        mov   ax, NCOLs         ;lastchar = dest + nc * 2
        shl   ax,1
        add   ax,dx
        push  ax                ;push lastchar offset on stack

        mov   ax, SSEG          ;load screen segment into ES, DS
        mov   es, ax
        mov   ds, ax

        mov   di,dx             ;set up SI and DI for movs
        mov   si,di             ;source is 2 bytes above DEST
        add   si,2

        mov   ax,[di]           ;save the char in col 0 in AX

        mov   cx, NCOLS         ;mov NCOLS words
        cld
        rep   movsw

        pop   di                ;pop lastchar offset to DI
        mov   [di],ax           ;put char in ax to last column

        dec   bx
        jmp   TOP
Done:pop es
        pop   ds
        pop   di
        pop   si
        mov   sp,bp
        pop   bp
        ret   ARGLEN
SCROLL  LEFT  0  ENDP
SCROL   TEXT     ENDS
        END
```

要想从 SCROLLH. PRO 和 SCROL. ASM 产生一个可执行文件，必须先用 Turbo Prolog 将 Prolog 文件编译产生一个. OBJ 文件(当 Turbo Prolog 编译一个模块时，它产生一个. OBJ 文件和一个. SYM 文件)。然后用 Turbo Assembler 将 SCROL. ASM 汇编为一个. OBJ 文件。最后用以下 ILINK 命令行将两个模块链接起来。

TLINK init scrolh scrol scrollh. sym,scroll, ,prolog

最终的可执行文件名为 SCROLL. EXE。

[例]:实现 double 谓词

假设一个汇编语言子程序将通过调用语句

double(MyInVar, MyOutVar)

而运行。在调用前 MyInVar 被约束为一个整型值,调用后 MyOutVar 被约束为其值的两倍。

double 被激活时位于栈底的活动记录如图 8.1 所示:

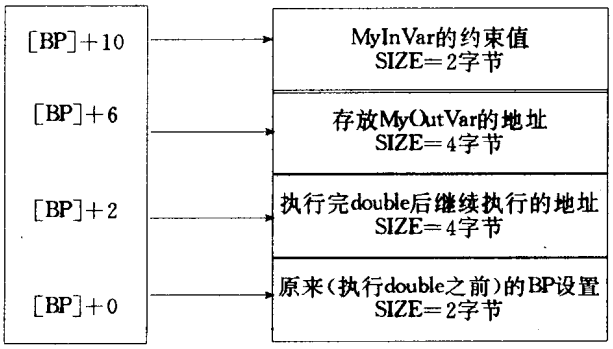


图 8.1　double 的活动记录

下列汇编语言函数实现了 double 谓词:

```
;
; MYASM. ASM
;
A  PROG     SEGMENT BYTE
            ASSUME CS:a  prog
            PUBLIC double  0
double  0   PROC FAR
            push   bp
            mov    bp,sp

            mov    ax, [bp]+6          ;get the value to which MyInVar is bound
            add    ax,ax               ;double that value
            lds    si, DWORD PTR [bp]+10
            mov    [si],ax             ;store the value to which MyOutVar is to
            pop    bp                  ;be bound in the appropriate address
            mov    sp,bp
            ret    6
double  0   ENDP
A  PROG     ENDS
```

调用 double 的 Turbo Prolog 程序必须包含如下全局谓词声明:

```
global predicates
    double(integer,integer) - (i,o) language asm
```

否则,Turbo Prolog 程序与别的程序就没有什么两样了。

下例程序用到了这个 double:

```
/* MYPROLOG.PRO */
global predicates
    double(integer,integer) - (i,o) language asm
goal
    write("Enter an integer"),
    readint(I),
    double(I,Y),
    write(I," doubled is ",Y).
```

如果这个汇编语言程序模块被汇编成文件 MYASM.OBJ，而且调用 Turbo Prolog 目标模块的是 MYPROLOG.OBJ，那么这二者可以通过如下命令行链接起来：

```
TLINK init myprolog myasm myprolog.sym,double,,prolog
```

它将产生一个独立的可执行文件 DOUBLE.EXE(使用 Turbo Prolog 的 PROLOG.LIB 库)。注意 MYPROLOG.SYM 是 TLINK 命令中第一个逗号前最后的一个文件名。

总的来说，活动记录的格式将依赖于调用 Turbo Prolog 谓词的参数个数和与那些参数相应的域类型。例如，如果想定义

```
add(Val1,Val2,Sum)
```

这里 Val1，Val2 和 Sum 属于整数域，活动记录将如图 8.2 所示：

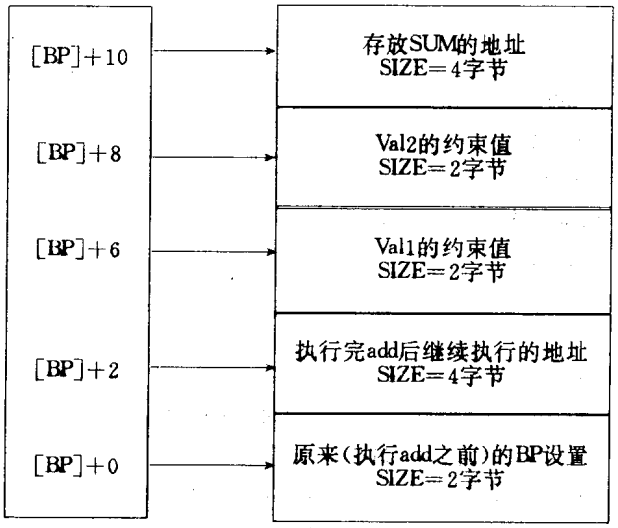


图 8.2 add 的活动记录

注意每个参数占据相应数量的字节：对输出参数，大小总是 4 字节(用于段基址和位移)；对输入参数，大小决定于真正推入堆栈的值，所以它取决于相应的域。

Val1 和 Val2 属于整数域，使用(i)流模式，都占 2 个字节；Sum 使用(o)流模式，占 4 个字节。

还应注意，在 Turbo Prolog 编译器中，对外部谓词的调用采用下列形式：

```
mov   ax,SEGMENT data
mov   ds,ax
call  FAR PTR external  predicate  implementation
```

因此外部谓词编写的过程在执行时的数据段为 DATA 段。

8.5　用多重流模式实现谓词

使用多重流模式实现谓词时，必须留心使汇编语言函数与 Turbo Prolog 的命名约定保持一致。假设要实现具有多重流模式的谓词 add，该 add 在三个变量中的两个被约束时给出等式 X+Y=Z 中另一个未被约束变量的值。

下面的 Turbo Prolog 程序 ADDPRO. PRO 声明了全局汇编语言谓词 add。注意 add 有三个流模式(i,i,o)，(i,o,i)和(o,i,i)。

```
/* ADDPRO.PRO */
global predicates
    add(integer,integer,integer) - (i,i,o),(i,o,i),(o,i,i)
                                        language asm
goal
    add(2,3,X), write("2 + 3 = ",X),nl,
    add(2,Y,5), write("5 - 2 = ",Y),nl,
    add(Z,3,5), write("5 - 3 = ",Z),nl.
```

下面的汇编语言程序 ADD. ASM 包含实现 add 的代码。ADD　0 对应于(i,i,o)流模式，而 ADD　2 对应于(o,i,i)。

```
        name    add
ADD  TEXT       SEGMENT BYTE PUBLIC 'CODE'
        ASSUME  CS:ADD_TEXT

        PUBLIC  ADD_0           ;(i,i,o) flow pattern
ADD_0  PROC     FAR
arg     Z:DWORD, Y:WORD, X:WORD = ARGLEN1
        push bp
        mov  bp,sp
        mov  ax,X
        add  ax,Y
        les  bx,Z
        mov  WORD PTR es:[bx],ax
        pop  bp
        ret  ARGLEN1
ADD_0  ENDP

        PUBLIC  ADD_1           ;(i,o,i) flow pattern
ADD_1  PROC     FAR
arg     Z:WORD, Y:DWORD, X:WORD = ARGLEN2
        push bp
        mov  bp,sp
```

```
        mov  ax,Z
        sub  ax,X
        les  bx,Y
        mov  WORD PTR es:[bx],ax
        pop  bp
        ret  ARGLEN2
ADD 1   ENDP

        PUBLIC    ADD 2          ;(o,i,i) flow pattern
ADD 2   PROC      FAR
arg     Z:WORD, Y:WORD, X:DWORD = ARGLEN3
        push bp
        mov  bp,sp
        mov  ax,Z
        sub  ax,Y
        les  bx,X
        mov  WORD PTR es:[bx],ax
        pop  bp
        ret  ARGLEN3
ADD 2   ENDP

ADD TEXT    ENDS
        END
```

在 ADDPRO.PRO 和 ADD.ASM 被转换为.OBJ 文件后，可以用以下命令行产生.EXE 文件：

```
TLINK init addpro add addpro.sym,addpro,,prolog
```

8.6 从汇编函数调用 Turbo Prolog 谓词

前面已经讨论了从 Turbo Prolog 调用汇编语言函数，现在讨论从汇编语言调用 Turbo-Prolog 谓词。

一个谓词声明为全局谓词后，该谓词就成为可以被其他模块调用的全局函数。命名和调用约定与前面讨论过的约定一样。

下面的 Turbo Prolog 模块定义两个全局谓词：popmessaged 和 from asm。popmessage 定义为 C 语言谓词，from asm 定义为汇编语言谓词。

为了建立 SHOWMESS，从 Turbo Prolog 开发环境将 SHOWMESS.PRO 编译为.OBJ 文件。再用

```
tasm from asm
```

编译 FROM ASM.ASM，并用以下命令进行连接：

```
TLINK init showmess from asm showmess.sym,showmess,,prolog
```

以下是 SHOWMESS：

```
/* SHOWMESS.PRO */
  global predicates
      popmessage(string) - (i) language c   /* predicates called from
                                              assembly language procedure */
      from   asm - language asm          /* assembly language procedure */
  clauses
      popmessage(S):-   /* can be called as a C function named popmessage   0*/
        str_len(S,L),
        LL = L + 4,
        makewindow(13,7,7,"",10,10,3,LL),
        write(S),
        readchar( ),
        removewindow.
  goal
      from   asm.                              /* external */
```

以下汇编代码实现 from　asm 并调用 popmessage：

```
                    EXTRN     popMessage  0:FAR
DGROUP              GROUP      DATA
                    ASSUME    CS:SENDMESS  TEXT, DS:DGROUP
 DATA  SEGMENT  WORD  PUBLIC 'DATA'
mess1               DB        "Report: Condition Red",0
 DATA  ENDS

SENDMESS  TEXT    SEGMENT  BYTE PUBLIC 'CODE'
                  PUBLIC    FROM  ASM  0
FROM  ASM  0        PROC FAR
      push  ds
      mov   ax,OFFSET DGROUP:mess1
      push  ax
      call    FAR PTR PopMessage  0
      pop   cx
      pop   cx
      ret
FROM  ASM  0        ENDP
SENDMESS  TEXT     ENDS
      END
```

下面的程序使用高层汇编扩展来创建相同的可执行程序。要创建这个程序，先从 Turbo Prolog 开发环境将 SHOWNEW.PRO 编译为.OBJ 文件，再用

```
tasm /jmasm51 /jquirks from   new
```

编译 FROM　ASM.ASM，并用以下命令进行连接：

```
TLINK init shownew from   new shownew.sym,show2,,prolog
```

以下是 SHOWNEW：

```
/* SHOWNEW.PRO */
global predicates
    popmessage(string) - (i) language c
            /* predicates called from assembly language procedure */
    from asm - language C as " from asm"
            /* define public name of the assembly language procedure */
clauses
    popmessage(S):-
      str_len(S,L),
      LL = L + 4,
      makewindow(13,7,7,"",10,10,3,LL),
      write(S),
      readchar( ),
      removewindow.
goal
    from asm.                /* call assembly language procedure */
```

下面的汇编码实现 from asm 并调用 popmessage(同前例):

```
; FROM NEW.ASM
      EXTRN    popMessage 0:FAR
      .MODEL    large,c
      .CODE

FROM ASM    PROC
      push ds
      mov ax,OFFSET DGROUP:mess1
      push ax
      call FAR PTR PopMessage 0
      pop cx
      pop cx
      ret
FROM ASM    ENDP

      .DATA
mess1 DB "Report: Condition Red",0
      END
```

8.7 表和函子

这一节将讨论将表和函子传递到汇编语言谓词的方法。如前所述,Turbo Prolog 复合目标不是直接传递的,而是传递一个 4 字节指针到一个结构。

用于表和函子的记录结构简单而又直接。假设有下面的 Turbo Prolog 域:

```
domains
    ilist = integer *
    ifunc = int(integer)
```

相应于 ilist 域的表结点如下所示：

```
STRUC ILIST
    NodeType   DB ?
    Value      DW ?
    NextNode   DD ?
ENDS
```

从这个结构可知，一个表结点有三个部分：

. 结点类型(一个字节)

. 结点值(字节数依赖于类型)

. 指向下一结点的指针(4 字节)

结点类型可以包含两个有意义的值：值 1 表示节点是一个表结点，而值 2 表示结点是一个表的终结点(一个表的终结点不包含其他有用信息)。结点值可为任意 Turbo Prolog 域。

ifunc 函子的相应结构如下所示：

```
STRUNC IFUNC
    FuncType   DB ?
    Value      DW ?
ENDS
```

一个函子的结构有两部分，函子类型和函子记录。函子类型是一个与函子变量在变元选择表中的位置有关的整数。第一个位置是类型 1，第二个位置是类型 2，等等。

以下的 Turbo Prolog 和 Turbo Assembler 模块将实现把一个函子返回到 Turbo Prolog 的谓词：

下面是 Turbo Prolog 模块：

```
/* FUNC.PRO */
domains
    ifunc = int(integer)
global predicates
    makeifunc(integer,ifunc),- (i,o) language C

goal
    makeifunc(4,H),
    write(H).
```

以下是 Turbo Assembler 模块：

```
;
; IFUNC.ASM
;
    EXTRN     alloc  gstack:FAR                    ; alloc  gstack returns
```

```
                                                    ; pointer to memory block

STRUNC IFUNC
    FuncType    DB ?
    Value       DW ?
ENDS

IFUNC  TEXT    SEGMENT  WORD PUBLIC 'CODE'
    ASSUME   CS:IFUNC  TEXT

    PUBLIC    Makeifunc  0
Makeifunc  0    PROC FAR
    arg          inval:WORD,   outp:DWORD
    push         bp
    mov          bp,sp
    mov          ax,3                              ;allocate 3 bytes
    push         ax
    call         FAR PTR  alloc  gstack
    pop          cx
    les          bx,   outp
    mov          [WORD PTR es:bx+2],dx
    mov          [WORD PTR es:bx],ax
    mov          ax,   inval
;;  les          bx,   outp
    les          bx,[DWORD PTR es"bx]
    mov          [(IFUNC PTR es:bx).VALUE],ax      ;value =   inval
    mov          [(IFUNC PTR es:bx).FUNCTYPE],1    ;type  = 1
    pop          bp
    ret
Makeifunc  0    ENDP
IFUNC  TEXT    ENDS
    END
```

本例中 ifunc 只使用了一种函子类型。用户如果如下声明另一个函子：

myfunc = int(integer); char(char); r(real); d2(integer, real)

结构就有点复杂了。这个结构仍将有两部分，但第二部分将是一个用于为 myfunc 定义所有变元的数据结构的联合。下面的结构是用 Turbo Assembler 实现的 myfunc：

```
STRUNC MyFunc
    FuncType DB ?
    UNION
      STRUNC
            int DW  ?
      ENDS
```

```
        STRUNC
                char DB ?
        ENDS
        STRUNC
                real DQ ?
        ENDS
        STRUNC
              v1   DW   ?
              v2   DQ   ?
        ENDS
    ENDS
ENDS
```

与函子选择有关的类型有：

int(integer)	1
char(char)	2
r(real)	3
d2(integer,real)	4

为了帮助理解表和函子，请参阅前面 ilist 的领域声明。为什么有效结点的类型是类型 1 和类型 2？因为 Turbo Prolog 把 ilist 当作一个结构，而这个结构只需简单地声明如下：

```
ilist = listnode(integer, listnode); end  of  list
```

记住当传递复合目标时，是传递一个指针到结构。更特殊地：一个输入流模式或函数是通过指针传递的，一个输出流模式的表或函子是以指向引用一个结构的指针的方式传递的（Turbo Prolog 传递一个指针的地址到返回的结构）。所有返回到 Turbo Prolog 的结构必须使用 Turbo Prolog 的内存分配函数来分配内存（参阅《Turbo Prolog 用户手册》和《Turbo Prolog 参考手册》）。

第九章 Turbo Prolog 与 MS-Fortran 4.0 的接口

Fortran 语言虽然经典，但在科学计算中仍占有举足轻重的地位。Turbo Prolog 虽然提供了与其它语言的接口，但不能直接实现与 Fortran 程序的连接。为了把 Prolog 的自动推理功能与 Fortran 的高效数值计算能力结合起来，解决人工智能领域中出现的需要大量计算的应用问题，就需要做一些准备工作，以实现 Turbo Prolog 与 MS-FORTRAN 4.0 这两个著名的风格各异而又各具特色的程序设计语言之间的接口以及两种程序之间的相互调用。

9.1 系统设置

首先，应把 Turbo Prolog 和 MS-Fortran 4.0 系统以及连接所需的文件装入硬盘的相应目录之下。一般可将 Turbo Prolog 2.0 装入 C 盘的 Prolog20 目录下，MS-FORTRAN 4.0 装入 C 盘的\FOR40 目录。

然后，执行以下命令：

```
path c:\for40; c:\prolog20
set lib=c:\for40
set tmp=c:\for40
```

接着，将以下文件拷贝到目录 prolog20 中：

crt0.asm，prep.asm，f.bat 以及 profor.lib

这些文件的内容在第 4 节中说明。

9.2 Turbo Prolog 调用 MS-Fortran 过程

9.2.1 在 Prolog 中说明外部谓词

被调用的 Fortran 子程序必须在 Prolog 中说明为全局谓词，并注明相应的参数传递特征。

例如：

```
global predicates
    factorial(integer,real) - (i,o) language FORTRAN
```

9.2.2　定义 Fortran 子程序并建立源程序

为了对 FORTRAN 编译器产生的代码进行必要的修整、改正，可调用一个预处理例程。对于下面的 Prolog 声明：

```
old   enough(real)，(i) language Fortran
```

FORTRAN 的实现如下：

```
$NOTRUNCATE                    * * * * * 注意：元命令必须从第一列开始！ * * * * *
subroutine old   enoughf0(age)
real * 8 age[value],age1
    * * * * * 必须有 age1 * * * * *
call prep(age,age1)                * * * * * 调用预处理例程 * * * * *
    * * * * * 以下处理 age1 * * * * *
    . . .
end
```

其中的 prep 子程序为实输入参数的预处理程序，用于解决 MS-Fortran 与 Turbo Prolog 内存分配不一致的问题，其代码在第 4 节给出。

9.2.2.1　命名约定

Fortran 子程序名应与 Prolog 的谓词名保持一种对应关系，即子程序名应在谓词名后加 f0。如上例中：

Prolog 谓词名　　：factorial

Fortran 子程序名　：factorialf0

Fortran 子程序名中的后缀 f0 的意义为：

f　　代表 Fortran；

0　　代表第一种流模式的实现。

注意流模式与实现子程序的对应关系，不要弄错。

9.2.2.2　参数约定

谓词说明中的参数以及子程序中定义的参数在个数、顺序、输入输出模式等方面必须一一对应。

参数传递的类型可以是整型、实型、字符串型等等，但整数应说明为 integer * 2，实数必须用 IEEE 标准浮点格式(8 字节)。Value 表示该值为值参数(输入)，而 reference 表示传地址方式的输出参数(相当于 Pascal 中的 Var 型参数)。

9.2.2.3　屏幕输出

在 Fortran 子程序中不要使用 FORTRAN 的输出子例程，因为 Prolog 的屏幕输出是面向窗口的。但是，可以象下面例子一样使用谓词。新的输出例程很容易编写，这些谓词能够提供比 FORTRAN 的相应部分更好的控制。

(1)写一个整数：

例：writeinteger(1234)

或 Pos = 20

writeinteger(Pos)

(2)写一个(8 字节)实数：

例：writereal(Height).

这里 Height 必须声明如下：

real * 8 Height

注意：writereal(3.3)是不对的；

(3)写字符串：

例：writestring('What a day !'C)

注意：上述结束符'C' 是必不可少的。

9.2.3 连接步骤

(1) 将 Fortran 源程序编译成. OBJ 文件：

FL -Od -c -Fpa Filename. for

(2) 建立工程文件(. PRJ 文件)：

例：工程文件内容大致如下：

P+	(PROLOG 源文件)
prep. obj+	(预处理文件)
test. obj	(FORTRAN 程序的目标文件)

(3) 设置库连接参数

在 Prolog 的主菜单下选择 Option/Link，敲入：

+LIBNAME. LIB

(4) 编译工程生成可执行文件

在 Prolog 的主菜单下选择 Compile/Project(或按 Alt－F9)，对工程中的所有模块编译并连接，生成可执行的. EXE 文件。

9.2.4 例　　子

Turbo Prolog 主程序"p. pro"内容如下：

```
project "p. prj"
global predicates
    test(real,real,real) - (i,o,o) language FORTRAN
    writereal(real) - (i) language C as "writereal"
    newline  language C as "newline"
    writeinteger(integer) - (i) language C as "writeinteger"
    writestring(string) - (i) language C as "writestring"
clauses
    writereal(R):-write(R).
    writeinteger(I):-write(I).
    writestring(S):-write(S).
    newline:-nl.
goal
```

```
    In = 2.10,
    write("          ------- In Prolog main program ----------\n"),
    write("          One input parameter is In = ",In),
    write("          Two output parameters. \n\n\n"),
    nl,
    test(In,Out,Three),
    write("\n\n\n          ---------- Return to Prolog ----------\n"),
    write("          Out(should be 2 * In) = ",Out),
    write("\n          Third argument(return from Fortran) = ",Three).
```

FORTRAN 程序"test. for"内容如下：

```
$FREEFORM
$LARGE

INTERFACE to SUBROUTINE writereal     [c,alias: 'writereal'](N)
real * 8 N[VALUE]
end

INTERFACE to SUBROUTINE writeinteger  [c,alias: 'writeinteger'](N)
integer * 2 N[VALUE]
end

INTERFACE to SUBROUTINE writestring   [alias: 'writestring'](N)
character * ( * ) N[reference]
end

INTERFACE to SUBROUTINE newline       [alias: 'newline']
end

INTERFACE to SUBROUTINE prep(a,b)
real * 8 a,b
end

subroutine testf0(in, out, three)
real * 8 in[value],in1,temp
integer * 2 ii
real * 8 out[reference],three[reference]
call writestring('          Now in Fortran. \nFirst, Preprocessing ! \n'C)
call prep(in, in1)

call writestring('Then examine input parameter. It is 'C)
call writereal(in1)
call writestring(', OK? \n'C)
```

```
call writestring('Now see 3 Fortran intrinsic functions:\n'C)
call writestring('        Sqrt(4.0)='C)
three=sqrt(4.0)
call writereal(three)
call newline

call writestring('        Dsin(pi/2)='C)
three=dsin(3.14/2)
call writereal(three)
call newline

call writestring('        Dmax1(1.4,5.6,7.8)='C)
three=Dmax1(1.4,5.6,7.8)
call writereal(three)
call newline

if (in1.GT.2) then
  temp=3.3333
  call writestring('In IF statement...Is 3.3333='C)
  call writereal(temp)
  call writestring(' ? \n'C)
  ii=12334
  call writestring('        And 12334='C)
  call writeinteger(ii)
  call writestring(' ? \n'C)
endif

out=2*in1
call writestring('Now this value exported to prolog--------'C)
three=3.444
call writereal(three)
call newline
end
```

工程文件"p.prj"内容如下：

```
p
test.obj
```

9.3 Fortran 调用 Turbo Prolog

Fortran 语言也能方便地调用 Turbo Prolog 谓词。为保证软件设计的一致性，与屏幕有关的输入/输出必须由 Turbo Prolog 控制，这样才能充分利用 Turbo Prolog 的漂亮的面向窗口的输入输出功能。

一般说来，Turbo Prolog 标准谓词不能被直接调用，而必须加上一层外壳。另外，被调用的谓词还必须说明为全局的。例如：

```
global predicates predicates
    writestring(string) - language C as "writestring"
clauses
    writestring(S):-write(S).
```

同时，在 Fortran 语言中还要作相应的说明：

```
Interface to subroutine writestring[c,alias:'writestring'](s)
character * ( * ) s[reference]
end
```

Prolog 和 Fortran 中的'C'都是表示 C 语言调用约定；alias 则指出 Turbo Prolog 相应程序中的命名；character * (*) 表示字符串长度在别处标明（实际上可以不标明）；s [reference]表示使用 reference 属性说明字符中的参数。

9.4 常用接口例程库、预处理程序的建立以及 Fortran 库的改造

9.4.1 常用接口例程库的建立

为了简化 Fortran 与 Prolog 的接口，并在日常应用中能够方便地调用接口例程，可以建立一个 Prolog-FORTRAN 接口例程库。例如，可将'writeinteger','writestring','prep'等放在一起，通过 Prolog 的 Compile/EXE 特性或工程特性来连接这个库。注意在应用时要正确地指定该库的访问路径（通过 Option 选单）。

为了用 Prolog 编译器来建立常用接口例程库，可以用 Prolog（使用全局谓词）来写出实现子句，并给出哑目标。例如：

```
global predicates
    writereal(real) - (i) language C as "writereal"
    newline   language C as "newline"
    writeinteger(integer) - (i) language C as "writeinteger"
    writestring(string) - (i) language C as "writestring"
clauses
    writereal(R):-write(R).
    writeinteger(I):-write(I).
    writestring(S):-write(S).
    newline:-nl.
goal
    nl.   /* 这只是为了让 Turbo Prolog 的编译器工作 */
```

再用 Compile/OBJ 来生成文件 Filename.obj 和 Filename.sym，然后使用 tlib 建库：

```
tlib LIBNAME.LIB +FILENAME.OBJ+FILENAME.SYM
```

最后再在 Prolog 的主菜单下选择 Option/Link，敲入：

+LIBNAME. LIB

这就行了。在别的 Prolog 程序中用 include 指令嵌入常用的声明文件(即全局谓词段)，Prolog 就会自动地对 LIBNAME. LIB 进行连接。

例：

COMMON. PRO：

```
global predicates
    writereal(real) - (i) language C as "writereal"
    newline  language C as "newline"
    writeinteger(integer) - (i) language C as "writeinteger"
    writestring(string) - (i) language C as "writestring"
```

用于建库的文件内容如下：

```
include "common. pro"
clauses
    writereal(R):-write(R).
    writeinteger(I):-write(I).
    writestring(S):-write(S).
    newline:-nl.
goal
    In=2.
```

使用以上构建的库的 PROLOG 例子程序——Q. PRO

```
project "q. prj"
include "common. pro"
goal
    writestring("Call the library routine"),
    newline,
    write("Use the Prolog standard predicates"),
    readchar( ).
```

q. prj 的内容为：

```
q
```

重要的是这些库例程可以被 Fortran 子例程使用！并且可以向这个库加入 C 语言的函数！

建　　议

可将所有的接口语句(Prolog 'common. pro' 的 Fortran 等价语句)和元命令都放到一个独立的包括文件的 Fortran 子程序中(见以上例子程序)，然后用以下命令来在 Fortran 程序中嵌入它：

```
$INCLUDE:'includefilename'
```

9.4.2　预处理程序

预处理程序 prep. asm 的源程序如下：

```
        TITLE   prep
        .8087
PREP    TEXT                    SEGMENT  BYTE PUBLIC 'CODE'
PREP    TEXT                    ENDS
 DATA                           SEGMENT  WORD PUBLIC 'DATA'
 DATA                           ENDS
CONST           FSEGMENT        WORD PUBLIC 'CONST'
CONST           ENDS
 BSS            SEGMENT  WORD PUBLIC 'BSS'
 BSS            ENDS
DGROUP          GROUP           CONST,   BSS,   DATA
        ASSUME  CS: PREP  TEXT,  DS: DGROUP,  SS: DGROUP,  ES: DGROUP
EXTRN     acrtused:ABS
PREP    TEXT                    SEGMENT
        PUBLIC  PREP
PREP     PROC   FAR
        push    bp
        mov     bp,sp
        sub     sp,0
        push    di
        push    si
        jmp     \ST14
\CO15:
        les     bx,DWORD PTR [bp+6]
        mov     ax,WORD PTR [bp+10]
        mov     dx,WORD PTR [bp+12]
        mov     di,bx
        mov     si,ax
        push    ds
        mov     ds,ax
        movsw
        movsw
        movsw
        movsw
        pop     ds
        jmp     \EX13
\ST14:
        jmp     \CO15
\EX13:
        pop     si
        pop     di
        mov     sp,bp
        pop     bp
```

```
        ret     8
PREP    ENDP
PREP  TEXT      ENDS
END
```

9.4.3 Fortran 库的改造

为了与 Prolog 接口，需对 FORTRAN 库(llibfora.lib)中的 crt0.obj 进行修改，新的 crt0.asm 源代码如下：

```
title       FORTRAN - FORTRAN start up routine

? DF=       1               ; This is special for c startup
include     version.inc
.xlist
include     cmacros.inc
include     msdos.inc
include     brkctl.inc
.list

            page

createSeg     TEXT,  code,  byte,  public, CODE,  <>
createSeg     c  ETEXT,etext,  byte,  public, ENDCODE,<>

createSeg     DATA,  data,  word,  public, DATA,  DGROUP
createSeg    STACK,  stack,  para,  stack,  STACK,  DGROUP

defGrp      DGROUP

codeOFFSET      equ    offset   TEXT:
dataOFFSET      equ    offset  DGROUP:

page

public        acrtused
              acrtused = 9876h

extrn         acrtmsg:abs

sBegin      stcak
assumes     ds,data
            db      2048 dup (?)
sEnd
```

```
page

externP     main
externP     exit

if          sizeC
extrn         exit:far
else
extrn         exit:near
endif

sBegin      data

extrn         edata:byte
extrn         end:byte

externW       psp
externW       argc
externDP      argv
externDP    environ

globalW       asizds,0
globalW       atopsp,0
globalW       aexit  rtn,<codeoffset   exit>

labelW      <PUBLIC, abrktb>
        dw      ?
        dw      DGROUP
        db      (MAXSEG-1) * (size segrec) dup (?)

labelW  <PUBLIC, abrktbe>
globalW  abrkp,<dataoffset  abrktb>

sEnd

page

externP  cinit
externP  NMSG  TEXT
externP  NMSG  WRITE
externP  FF  MSGBANNER

externP  setargv
```

```
externP   setenvp
externP   nullcheck

sBegin    code
assumes   cs,code
assumes   ds,nothing

labelNP   <PUBLIC, astart>

;         check MS-DOS version for 2.0 or later

          callos    VERSION
          cmp       al,2
          jae       setup

          call       FF  MSGBANNER
          mov       ax,4
          push      ax
          call       NMSG  TEXT
          xchg      dx,ax
          callos    message
          int       20h

setup:
          mov       di,DGROUP
          mov       si,ds:[DOS+MAXPARA]
          sub       si,di
          cmp       si,1000h
          jb        setSP
          mov       si,1000H

setSP:
          cli
          mov       ss,di
          add       sp,dataoffset  end - 2
          sti
          jnc       SPok

          call       FF  MSGBANNER
          xor       ax,ax
          push      ax
          call       NMSG  WRITE
          mov       ax,DOS  terminate shl 8 + 255
```

```
        callos

SPok:
        assumes ss,data

        and     sp,not 1
        mov     [ abrktb].sz,sp
        mov     [ atopsp],sp

        mov     ax,si
        mov     cl,4
        shl     ax,cl
        dec     ax
        mov     [ asizds],ax

        release extra space to DOS

        add     si,di
        mov     ds:[DOS  MAXPARA],si
        mov     bx,es
        sub     bx,si
        neg     bx
        callos  setmem
        mov     [ psp],ds

        zero data areas ( BSS and c  common)

        push    ss
        pop     es
        assumes es,data

        cld
        mov     di,dataOFFSET  edata
        mov     cx,dataOFFSET  end
        sub     cx,di
        xor     ax,ax
        rep     stosb

;       segmentation conventions set up here (DS=SS and CLD)

        push    ss
        pop     ds
        assumes ds,data
```

```
        do necessary initialization BEFORE command line processing !

        call    cinit

        push    ss
        pop     ds
        assumes ds,data

;       process command line and environment

        call    setargv
        call    setenvp

;       call main and exit

        xor     bp,bp

if sizeD
        push    ds
endif
        push    word ptr [environ]

if sizeD
        push    ds
endif
        push    word ptr [  argv]

        push    [  argc]
        call    main

;       use whatever is in ax after returning here from the main program

        push    ax
        call    exit

page

;-----------------------------------------------------------------
;
;       Fast exit fatal errors - die quick and return (255)

labelNP <PUBLIC, cinitDIV>
```

```
        mov     ax,3
        mov     [ aexit  rtn],codeoffset  exit

labelNP <PUBLIC, amsg  exit>
        push    ax
        call      FF  MSGBANNER

        call      NMSG  WRITE

        mov     ax,255
        push    ax
        call    word ptr [ aexit  rtn]

sEnd
        end       astart
        end
```

新的 llibfora.lib 可如下建立：

```
masm crt0 /mx
dosseg crt0.obj
llib llibfora -+crt0.obj
```

第十章
Turbo Prolog 访问 dBASE Ⅲ 数据文件

数据库技术和人工智能技术结合是计算机技术发展的必然结果。在计算机应用不断深入的今天，人们已经开始将人工智能与数据库技术有机地结合起来。演绎数据库、专家系统、知识库等都是数据库技术的扩展或数据库技术与人工智能结合的结晶。数据库和逻辑推理机结合的关键是逻辑推理机与传统数据库间的接口。

本文介绍 Turbo Prolog 访问 dBASE Ⅲ 的思路与方法。

10.1 Prolog 事实与 dBASE Ⅲ 记录

Turbo Prolog 系统所能识别的是事实，而 dBASE Ⅲ 系统产生的是记录。因此，在 Turbo-Prolog 语言中应用 dBASE Ⅲ 数据文件的关键是将 dBASE Ⅲ 数据文件(DBF)的记录转换成 Turbo Prolog 能够识别的事实。事实上，DBF 中的一条记录是直接对应于 Turbo Prolog 中的一个事实的，只是表示形式不同而已，即 DBF 中关系名对应于 Turbo Prolog 的事实名，每一条记录对应于一个事实。表 10.1 给出 DBF 中学生关系与 Turbo Prolog 事实的对应关系，学生关系的数据格式为“学号、姓名、系”。

事　实		学生关系
student(1,Smith,comp.)		(1,Smith,comp.)
student(2,Jones,math.)	<=====>	(2,Jones,math.)
student(3,Blake,comp.)		(3,Blake,comp.)

表 10.1

显然，将 DBF 中的记录转换成 Turbo Prolog 事实无需在量值方面作任何变动，只需在每条记录前冠以事实名即可。要对 DBF 进行操作，首先必须知道 dBASE Ⅲ 中 DBF 的存贮格式。

10.2 dBASE Ⅲ 中 DBF 的存贮结构

DBF 由两部分组成：库结构部分和数据部分。库结构给出了数据(即记录)的结构信息；数据部分以记录形式存放数据。

库结构部分的长度是：(文件字段数+1) * 32+2，其格式见表 10.2。

表 10.2 中记录个数是指 DBF 中当前记录的个数，低位在前，高位在后；数据起始地址是指 DBF 中数据的起始存放地址；记录长度中放的是每条记录的长度，低位在前，高位在后。接下来依次说明 DBF 的属性，属性名占 10 个字节，不足者以空格代替；结构部分第一个字节取 03H 时，类型标志有四种可能：N ——数值型，C ——字符型，D ——日期型，L ——逻辑型；属性长度为该属性值所能取的最大长度，若该属性的类型为实数型，就在小数长度一栏中指出，否则为 0。

1	2～4	5～8	9～10	11～12	13～32						
83H 或 03H	日期		记录个数	数据起始地址	记录长度	空					
属性名					空	类型标志	空	属性长度	小数长度	空	
								17	18		
属性名					空	类型标志	空	属性长度	小数长度	空	

表　10.2

在库结构部分中，最后二个字节为库结构的结束标志 0DH 和 00H。

数据部分紧接在库结构结束标志之后。每个记录的第一个字节为删除标志，若在 dBASE Ⅲ 中用 DEL 删除了记录但并未用 PACK 命令压缩，其内容为 2AH；否则为空格。其后以 ASCII 码形式连续存放记录中各属性值。

10.3　把 DBF 记录转换成 Turbo Prolog 事实

可以用能与 Turbo Prolog 交互的任何语言编写从 DBF 记录到 Turbo Prolog 事实的转换程序，也可以用 Turbo Prolog 语言本身来编写转换谓词。具体的谓词格式是：

record　to　fact(Fact　name, Dbf　name, Start, Amount)

其中 Fact　name 为转换后的 Prolog 事实名；Dbf　name 为转换前的数据库文件名；Start 为转换为 Prolog 事实的第一个记录号；Amount 为转换的记录个数。

转换的主要步骤：

(1) 识别 DBF 并建立中间数据文件，读入数据起始地址、记录长度等；

(2) 读入各属性信息，并利用 Turbo Prolog 提供的回溯技术与表处理功能，将属性内容全部放入属性表中；

(3)　依据提供的初始记录号 Start 来确定第一个转换记录的位置；

(4)　依据属性表，依次读入各个记录，并实现转换，转换的个数由 Amount 来确定，转换后的事实存入中间数据文件；

(5) 把中间数据文件中的事实读入到 Turbo Prolog 系统中。

值得注意的是，每次转换的数据量 Amount 应根据 Turbo Prolog 系统装入应用程序后动态数据库的容量而定。每次转换的记录个数要适中，少则增加了内存与外存的交换次数，多则可能内存溢出。

一般说来，以数据块为单位把数据库的数据转换成 Prolog 事实并进入 Prolog 系统内，是一个好的策略。转换后的数据以动态数据库形式存放有利于搜索与匹配。

可以按照上述思想自己实现转换谓词，当然，也可以利用 Turbo Prolog 工具库中已为我们提供的现成工具谓词来实现存取 dBASE Ⅲ 数据库文件的操作。

10.4　利用 Turbo Prolog 工具库访问 dBASE Ⅲ 数据文件

利用 Turbo Prolog 工具库访问 dBASE Ⅲ 文件，需要包括文件 DBASE3.PRO，还需要模块 REALINTS.OBJ 以进行连接。REALINTS.OBJ 的功能是在 4 个整数和一个实数之间进行相互转换，它的源程序在 REALINTS.C 中定义。要使用这些工具谓词，必须使用 1.10 版（或更高版本）的 Turbo Prolog 系统，因为在这些版本中定义了二进制存取方式。

本章中描述的所有工具都需要文件 READEXT.PRO，其功能是完成一般的转换。

Turbo Prolog 程序访问 dBASE Ⅲ 文件的第一步是调用文件 DBASE3.PRO 中的工具谓词 init　Dbase3，该谓词构造一些描述 dBASE Ⅲ 记录的数据结构，其说明如下：

```
init　Dbase3(REAL, FLDNAMEL, FLDDESCL)
```

其中非标准领域的说明如下：

```
FldDescL     = FldDesc *                    /* description for each field */
FldDesc      = flddesc(Dbase3Type,Integer)
Dbase3Type   = ch;r;l;m;d
FldNameL     = String *
```

根据处理算法的需要，工具库提供了可以一次读出 dBASE Ⅲ 文件所有记录和一次只读一个记录的工具谓词。

10.4.1　一次读出 dBASE Ⅲ 文件的所有记录

使用工具谓词 rd　dBase3 File 可以读出 dBASE Ⅲ 的数据记录，并存放在一个属于工具领域 DBASE3RECL 的表中。其说明如下：

```
Dbase3RecL   = Dbase3Rec *      /* A database is a number of records */
Dbase3Rec    = Dbase3Elem *     /* A record is a number of fields    */
Dbase3Elem   = char(String);    /* Characters                        */
               real(Real);      /* 64-bit IEEE floating point        */
               logical(Bool);   /* Logical                           */
               memo(String);    /* 10 digits rep. a .DBT block no    */
               date(String)     /* format YYYY MM DD                 */
Bool         = Char             /* Y y N n T t F f or Space          */
```

领域 dBASE3 RECL 中的记录表和领域 FLDNAMEL 中的字段名表构成了一个完整地描述 dBASE Ⅲ（1.1 版）数据库文件的数据结构。

工具库文件 XDBASE3.DBF 是一个包含如表 10.3 结构的人事数据 dBASE Ⅲ 文件：

字　段（属 性）	类　型	宽　度
Name 姓名	Char/String	25
Brith　Date(出生日期)	Date	8
Salary(工资)	Numeric	8.2
Age(年龄)	Numeric	2
Memo(备注)	Memo	

相应 Turbo Prolog 记录的格式为：

```
[ string("Frank Borland"),
date("19131205"),
real(10250.95),
real(73),
memo("Frank Borland's memo")]
```

注意，该记录置于其它记录之中。

相应字段名表的形式为：

```
["Name", "Birth  Date", "Salary", "Age", "Memo"]
```

在工具库中，rd Dbase3File 的说明为：

```
rd  Dbase3File(REAL, FILE, FLDDESCL, DBASE3RECL)
```

下面的例子指出了如何使用该谓词以访问 dBASE Ⅲ 文件 XDBASE3.DBF 和 Memo 文件 XDBASE3.DBT：

```
accessAll(DbaseRecs):-
    openread(fp,"xdbase3.dbf"),
    filemode(fp,0),
    readdevice(fp),
    openread(mfp,"xdbase3.dbt"),
    filemode(mfp,0),

    init  dbase3(TotRecs,FldNameL,FldDescL),
    rd  Dbase3File(TotRecs,mfp,FldDescL,RecL),
    makewindow(85,41,36," dBASE III(TM) All Data Records ",0,0,25,40),
    list  recL(FldNameL,RecL), DoPrompt.
```

最后的两个语句(makewindow 和 list recL)不是必需的，但它们能以易读的方式显示 dBASE Ⅲ 文件的内容。

10.4.2　一次读出一个 dBASE Ⅲ 记录

工具谓词 rd Dbase3Rec 使我们能够顺序地读出 dBASE Ⅲ 记录，这样，有了 rd Dbase3Rec，我们就能读一个记录并进行运算，再通过回溯撤消存储分配，然后再读下一个记录。下例说明了这种方法：

```
OneByOne:-
    openread(fp,"xdbase3.dbf"), filemode(fp,0), readdevice(fp),
    openread(mfp,"xdbase3.dbt"),filemode(mfp,0),
    init  dbase3(TotRecs,FldNameL,FldDescL),
    makewindow(85,72,33," dBASE III(TM) Sequential Access ",0,40,25,40),
    rd  Dbase3Rec(TotRecs,mfp,FldDescL,Rec),
    list  recL(FldNameL,[Rec]),fail.
```

谓词 rd Dbase3Rec 也是非确定性的。当 dBASE Ⅲ 文件中有多个记录时，该谓词读出结构 Rec 中的下一个记录，并返回到调用程序。当调用程序失败时，rd Dbase3Rec 产生一个新的结果(即如果存在下一记录的话就读出下一记录)。说明如下：

```
rd  Dbase3Rec(REAL, FILE, FLDDESCL, DBASE3REC)
```

下列完整程序 XDBASE3. PRO 中使用了 rd　Dbase3Rec 和 rd　Dbase3File 这两个谓词：

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

    Turbo Prolog Toolbox

        Access a dBASE III(TM) (V1.1) compatible file from Prolog

        The two datafiles XDBASE3.DBF and XDBASE3.DBT must
        be present at the default directory.

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

project "xdbase3"

Domains
  File            = fp ; mfp

global predicates
  real  ints(REAL,INTEGER,INTEGER,INTEGER,INTEGER) - (o,i,i,i,i) language c

include "readext.pro"
include "dbase3.pro"

PREDICATES
  /* Listning of the database */
  list  recl(FldNameL,Dbase3RecL)
  list  rec(FldNameL,Dbase3Rec)
  list  elem(Dbase3Elem)
  NoofNL(Integer)
  PressAKey
  doPrompt

CLAUSES

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
               List Data from a .DBF & .DBT
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

 list  recL( ,[]) :- !.
 list  recL(FldNameL,[Rec|RecL]) :-
```

```
        nl,nl,
        list  rec(FldNameL,Rec), PressAkey,
        list  recL(FldNameL,RecL).

list  rec([],[]) :- !.
list  rec([FldName|FldNames],[Elem|Elems]) :-
        writef("\n%-12: ",FldName),
        list  elem(Elem),
        list  rec(FldNames,Elems).

list  elem(char(Str))    :- write(Str).
list  elem(real(Real))   :- write(Real).
list  elem(logical(CH)) :- write(CH).
list  elem(memo(Str))   :- write(Str).
list  elem(date(Date))   :- write(Date).

NoofNL(N) :- N<=0,!.
NoofNL(N) :- nl, N2=N-1, NoofNL(N2).

PressAKey :-
        makewindow( , , , ,MinR, ,NoofR, ),
        cursor(R, ), R<=MinR+NoofR-8,!.

PressAKey :- doPrompt,cursor(R,C), scroll(R,0),cursor(0,C).

doPrompt:-
        makewindow(Nr,Att, , ,MinR,MinC,NoofR,NoofC),
        MinR2=MinR+NoofR-1, MinC2=MinC+NoofC/3+1,
        str  len(" Press a key",Len), Len2=Len+1, bitxor(Att,8,Att2),
        makewindow(Nr,Att2,0,"",MinR2,MinC2,1,Len2),
        write(" Press a key"),
        readdevice(FP), readdevice(keyboard), readchar( ), readdevice(FP),
        removewindow.


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                    Goal
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

goal    openread(fp,"xdbase3.dbf"), filemode(fp,0), readdevice(fp),
        openread(mfp,"xdbase3.dbt"),filemode(mfp,0),

        /* Build data structure */
```

```
        init  dbase3(TotRecs,FldNameL,FldDescL),
        /* Remember file positions */
        filepos(fp,Fposfp,0), filepos(mfp,Fposmfp,0),

        /* Read all data records */
        rd  Dbase3File(TotRecs,mfp,FldDescL,RecL),

        /* List all records */
        makewindow(85,41,36," dBASE III(TM) All Data Records ",0,0,25,40),
        list  recL(FldNameL,RecL), DoPrompt,
        window  attr(27),

        /* Read records one by one */
        filepos(fp,Fposfp,0), filepos(mfp,Fposmfp,0),
        makewindow(85,72,33," dBASE III(TM) Sequential Access ",0,40,25,40),
        rd  Dbase3Rec(TotRecs,mfp,FldDescL,Rec),
        list  recL(FldNameL,[Rec]),fail.
```

工具文件 DBASE3.PRO 的源程序清单如下：

```
/*************************************************

    Turbo Prolog Toolbox

        Access a dBASE III(TM) (V1.1) compatible file from Prolog

*************************************************/

Domains
/*************************************************
        Prolog representation of the data base
*************************************************/

  Dbase3RecL    = Dbase3Rec*          /* A database is a number of records */
  Dbase3Rec     = Dbase3Elem*         /* A record is a number of fields    */
  Dbase3Elem    = char(String);       /* Characters                        */
                  real(Real);         /* 64-bit IEEE floating point        */
                  logical(Bool);      /* Logical                           */
                  memo(String);       /* 10 digits rep. a .DBT block no    */
                  date(String)        /* format YYYY MM DD                 */

  Bool          = Char                /* Y y N n T t F f or Space          */

  FldDescL      = FldDesc*            /* description for each field        */
  FldDesc       = flddesc(Dbase3Type,Integer)
```

```
  Dbase3Type      = ch;r;l;m;d

  FldNameL        = String *

PREDICATES
  /* Read predicates */
  Init   Dbase3(Real,FldNameL,FldDescL)
  rd   dbase3   DbaseHeader(Real)
  rd   dbase3   fieldDescL(FldNameL,FldDescL)
  rd   dbase3File(Real,File,FldDescL,Dbase3RecL)
  rd   dbase3   DataRec(File,FldDescL,dBase3Rec)
  rd   dbase3   elem(File,FldDesc,dBase3Elem)

  conv   FldType(Char,dBASE3Type)

  /* Read a single record */
  rd   dbase3   DataRec1(Real,Real,File,FldDescL,dBase3Rec)
  rd   dbase3Rec(Real,File,FldDescL,dBase3Rec)

CLAUSES
  Init   Dbase3(TotRecs,FldNameL,FldDescL):-
         rd   dbase3   DbaseHeader(TotRecs),
         rd   dbase3   fieldDescL(FldNameL,FldDescL).

/*************************************
         Read dBASE III(TM) header
*************************************/

  rd   dbase3   DbaseHeader(TotRecs):-
         ignore(4),                    /* ID & Last update & record size   */
         read   long(TotRecs),         /* 32-bit number                    */
         ignore(24).                   /* Header length, Record length & Reserved

/*************************************
         Read Field descriptors
*************************************/

  rd   dbase3   fieldDescL([FldName|FldNameL],[fldDesc(Type,Len)|FldDescL]):-
         readchar(Ch), Ch<>'\013',!,   /* CR means final array field       */
         read   strArr(10,Name), frontchar(FldName,Ch,Name),
         readchar(T), conv   FldType(T,Type),
         ignore(4),                    /* data address                     */
         readchar(L), char   int(L,Len),
```

```
        ignore(15),                          /* decimal count & Reserved        */
        rd  dbase3  FieldDescL(FldNameL,FldDescL).

    rd  dbase3  FieldDescL([],[]):-readchar( ).

    conv  FldType('C',ch):-!.
    conv  FldType('N',r):-!.
    conv  FldType('L',l):-!.
    conv  FldType('M',m):-!.
    conv  FldType('D',d):-!.

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        Read Data Records
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

    rd  dbase3File(0, , ,[]):-!.
    rd  dbase3File(N,MFP,FldDescL,[Rec|RecL]):-
        rd  dbase3  DataRec1(N,N2,MFP,FldDescL,Rec),
        rd  dbase3File(N2,MFP,FldDescL,RecL).

    rd  dbase3  datarec1(Ni,No,MFP,FldDescL,Rec):-
        readchar(NotDel), NotDel = ' ',!, No = Ni-1,rd  dbase3  DataRec(MFP,FldDescL,
Rec).
    rd  dbase3  datarec1(Ni,No,MFP,FldDescL,Rec):-
        Ni2=Ni-1, rd  dbase3  datarec(MFP,FldDescL, ),
        rd  dbase3  DataRec1(Ni2,No,MFP,FldDescL,Rec).

    rd  dbase3  DataRec( ,[],[]):-!.
    rd  dbase3  DataRec(MFP,[FldDesc|FldDescL],[Elem|ElemL]):-
        rd  dbase3  elem(MFP,FldDesc,Elem), rd  dbase3  DataRec(MFP,FldDescL,ElemL).

    rd  dbase3  elem( ,fldDesc(ch,Len),char(Str)):-!,read  strArr(Len,Str).
    rd  dbase3  elem( ,fldDesc(l,Len),logical(Char)):-!,
        readchar(Char), ToSkip=Len-1, ignore(ToSkip).
    rd  dbase3  elem( ,fldDesc(r,Len),real(Real)):-!,
        read  strArr(Len,Str), str  real(Str,Real).
    rd  dbase3  elem(MFP,fldDesc(m,Len),memo(Memo)):-!,
        read  strArr(Len,BlkNo),
        str  int(BlkNo,P), Pos=P * 512,
        readdevice(FP), readdevice(MFP),
        filepos(MFP,Pos,0), read  strCtrlZ(Memo),
        readdevice(FP).
    rd  dbase3  elem( ,fldDesc(d,Len),date(Date)):-!,read  strArr(Len,Date).
```

```
/********************************************
        Read data records sequentially
********************************************/

rd dbase3Rec(0, , , ):-!,fail.
rd dbase3Rec(N,MFP,FldDescL,Rec):-rd dbase3 datarec1(N, ,MFP,FldDescL,Rec).
rd dbase3Rec(N,MFP,FldDescL,Rec):-N2=N-1, rd dbase3Rec(N2,MFP,FldDescL,Rec).
```

realints. c 的源程序清单如下：

```
real ints 0(P1,P2)
double *P1, P2;
{ *P1=P2; }
```

对读入 Turbo Prolog 系统内的数据库记录，可以与 Turbo Prolog 程序本身的事实一样直接参与运算和逻辑推理。

第十一章 Turbo Prolog 与 DOS 系统级的接口

除了能与各高级语言交互外，Torbo Prolog 还提供了直接访问计算机操作系统和硬件的谓词。本章首先介绍访问 DOS 的谓词，然后介绍访问 BIOS、内存以及为其它硬件设备提供低层支撑的一些谓词。最后举例说明如何在 Turbo Prolog 应用程序中使用这些谓词。

11.1 访问 DOS

利用一些谓词，可以在运行 Turbo Prolog 集成环境时访问 DOS，也可以在运行时刻从用户的 Turbo Prolog 应用程序中访问操作系统。用户可以使用谓词 system 来调用 DOS 命令，也可以用 date 和 time 来查询或重设 DOS 的日期和时间，用 envsymbol 查找 DOS 环境参数，用 comline 读入命令行参数。

本节详细说明这些谓词，并给出使用这些谓词的例子。

11.1.1 system/1

Turbo Prolog 程序通过谓词 system 访问 DOS。

调用格式：system("DOS 命令")。

如果变元为空串("")，则进入 DOS 环境，这要求可以访问到 DOS 的命令解释程序 COMMAND.COM。进入 DOS 状态后即可执行 DOS 命令，键入 exit 则从 DOS 回到 Turbo Prolog。

例子：

1. 在 Turbo Prolog 系统内将文件 B:FILE1.ORG 拷贝到文件 A:FILE2.TRG 中。可以先用 system("")进入 DOS 环境，然后再用以下 DOS 命令进行拷贝：

```
copy b:file1.org a:file2.trg
```

最后再键入 exit 返回到 Turbo Prolog 系统中。

2. 使用以下子句可以不进入 DOS 环境而对文件改名：

```
system("ren oldfile.1 newfile.2").
```

11.1.2 system/3

谓词 system 的扩充版本 system/3 提供了另外两个功能：返回 DOS 错误级别和重置运行时刻的系统视频模式。

调用格式：system(DosCommandString, ResetVideo, DOSErrorLevel) /* (i,i,o)

*/

DOS 错误级别在 DOSErrorLevel 中返回(关于 DOS 错误级别的内容参见有关的 DOS 技术手册)。

ResetVideo 决定是否将用户的程序重置到调用前的视频硬件状态。若 ResetVideo = 1 则重置;ResetVideo = 0 则不重置。当 ResetVideo = 0 时,即使该方式并非 Turbo Prolog 特别支持的模式,程序也在新设置的视频模式中运行(关于设置运行时刻视频模式的内容请参见有关的显示器硬件参考手册)。即如果用户的外部程序把视频方式置为一个 TurboProlog 不特别支持的模式,而且程序中有下述 system 调用(在开发环境中),那么用户程序就可以在这个不特别支持的模式中运行。

```
system("mysetmd",0,DOSErrorLevel)
```

注:外部程序必须至少在 BIOS 级与硬件兼容。

例子:

下列程序是一个窗口驱动的文件拷贝程序。用该程序进行文件拷贝时不须记住源文件和目标文件的先后顺序。

```
goal
  makewindow(1,7,7,"Source",0,0,20,35),
  write("Which file do you want to copy ?"),
  cursor(3,8),readln(X),
  makewindow(2,7,7,"Destination",0,40,20,35),
  write("What is the name of the new copy ?"),
  cursor(3,8),readln(Y),
  concat(X," ",X1),concat(X1,Y,Z),
  concat("copy ",Z,W), /* creates the string W */
  makewindow(3,7,7,"Process",14,15,8,50),
  write(" Copying ",X," to ",Y),cursor(2,3),
  system(W). /* invokes DOS with the string W */
```

11.1.3　envsymbol/2

谓词 enysymbol 在 DOS 环境中查找应用程序的环境符号。这些符号是由 DOS 的 SET 命令设置的。

调用格式:snvsymbol(DOS　env　symbol, Value)　　/* (i,o) */

例子:

以下 DOS 命令把符号 BGIDIR 设置为字符串 C:\tprolog2\bgi

```
SET BGIDIR = c:\tprolog2\bgi
```

而以下 Turbo Prolog 目标则在 DOS 环境中查找符号 BGIDIR,并将 SetValue 约束为 c:\Tprolog2\bgi:

```
/* . . . */
envsymbol("BGIDIR", SetValue),
```

```
initgraph(0,0, , ,SetValue),
/* . . . */
```

如果所查符号不存在，则 envsymbol 失败。

11.1.4 date/3 和 time/4

Turbo Prolog 提供了两个与 DOS 有关的常用标准谓词：date 和 time。这两个谓都能以两种方式使用。使用哪种版本取决于调用时变元是否自由。

下列调用中，所有变元均被约束，time 重置系统内部时钟为指定时间。若所有变元都自由，系统将当前内部时钟设为新的值。

调用格式：time(Hours, Minutes, Seconds, Hundredths)　　/* (i,i,i,i),(o,o,o,o) */

date 与 time 类似，也依赖于系统内部时钟。

调用格式：date(Year, Month, Day)　　/* (i,i,i),(o,o,o) */

例子：

下列程序用 time 来显示列目录操作所需的时间：

```
predicates
  timer
clauses
  timer:-
      time(H1,M1,S1,D1),nl,
      write("Start time is: ",H1,":",M1,":",S1," and ",D1,"/100 sec"),nl,
      /* This is the activity that is being timed */
      system("dir *.*"),
      time(H2,M2,S2,D2),
      Time = (D2-D1) + 100 * ((S2 - S1) + 60 * ((M2 - M1) + 60 * (H2 - H1))),
      write("Elapsed time:  ",Time,"/100 sec"),nl,
      time(H3,M3,S3,D3),
      write("The time now is: ",H3,":",M3,":",S3," and ",D3,"/100 sec"),nl.
goal
  makewindow(1,7,7," Timer ",8,10,12,60),
  write("Press any key to start"),
  readchar( ),
  timer.
```

11.1.5 comline/1

当用 DOS 执行.EXE 程序时，可以用 comline 来读出命令行参数。

调用格式：comline(DOSCommandLine)　　/* (o) */

例子：

下面的程序 EDIT.PRO 实现了一个编辑器，被编辑的文件名作为 DOS 命令行参数。

```
predicates
```

```
  extend(string, string)
clauses
  extend(S,S):-concat( ,".pro",S), !.
  extend(S,S1):-concat(S,".pro",S1).
goal
        comline(X),  /* must be compiled to .exe to function correctly */
        extend(X,X1),
        file_str(X1,S),
        makewindow(1,23,8,"EDITOR",0,0,25,80),
        editmsg(S,S1,"","","",0,"",RET),
        removewindow,
        RET><1,!,
        file_str(X1,S1).
```

把该程序编译成可执行文件后，就可以用下列 DOS 命令把文件 MYFILE. PRO 装入编辑器。

```
C:\>edit myfile.pro
```

11.2 访问硬件：低级支撑

DOS ROM-BIOS(Read Only Memory-Basic Input/Output System，即只读存贮器基本 I/O 系统)提供了程序与操作系统的接口，可以进行磁盘、文件、打印机、显示器 I/O 等各种功能调用。要了解详细的 ROM-BIOS 信息请参阅 DOS 技术手册。

Turbo Prolog 提供了六个内部谓词来对操作系统、I/O 端口和硬件进行低层存取。这些谓词是 bios(两种格式)、ptr_dword、memword、membyte 以及 port_byte。

11.2.1 bios/3 和 bios/4

谓词 bios 访问 PC 的低级 BIOS 例程。它通过预定义的复合对象 reg(...)从 BIOS 例程接受或传送信息。bios 有以下两种格式：

```
bios(InterruptNo, RegistersIn, RegistersOut)          /* (i,i,o) */
bios(InterruptNo, RegistersIn, RegistersOut, OutFlags) /* (i,i,o,o) */
```

这里，RegistersIn 和 RegistersOut 是如下定义的数据结构：

```
/* RegistersIn                                          */
reg(AXi,BXi,CXi,DXi,SIi,DIi,DSi,ESi)                   /* (i,i,i,i,i,i,i,i)    */
/* RegistersOut                                         */
   reg(AXo,BXo,CXo,DXo,SIo,DIo,DSo,ESo)                /* (o,o,o,o,o,o,o,o)    */
```

bios 使用的单元：

. AXi、BXi、CXi、DXi、SIi、DIi、DSi 和 ESi 表示传送给 BIOS 的硬件寄存器值；

. AXo、BXo、CXo、DXo、SIo、DIo、DSo 和 ESo 表示从 BIOS 返回的寄存器值。

bios/3 的流模式为(i,i,o)，而 bios/4 则为(i,i,o,o)。当调用 bios 时，RegistersIn 的各个变元都要被约束，而 RegistersOut 的各个变元均为自由的。

复合对象 RegistersIn 和 RegistersOut 的域是 reg 域，这是 Turbo Prolog 特别为谓词 bios 预定义的数据结构。Turbo Prolog 对该数据结构的定义是：

```
domains
    reg = reg(integer,integer, ..., integer)
```

bios/4 的可选变元 OutFlags 是 8086 标志寄存器的紧凑表示（见图 11.1），OutFlags 是从中断返回时读取的状态内容。标志是以下列方式压缩为整数值的：

U	U	U	U	O	D	I	T	S	Z	U	A	U	P	U	C
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

代码	标志	用途
U	未定义	不用
O	溢出	表示运算后高位溢出
D	方向	指明传递或比较字符串数据时处理的方向
I	中断	表示是否允许中断
T	陷井	允许 CPU 按单步方式操作
S	符号	含有运算结果的符号
Z	零	表示运算或比较操作的结果（0—结果非 0；1—结果为 0）
A	辅助进位	某些特殊运算要求里 8 位数据中低 4 位的进位
P	奇偶校验	运算结果低 8 位数据的奇偶校验（1 表示含偶数个 '1'）
C	进位	运算后从高位的进位值或移位、循环移位的最后一位内容

图 11.1　8086 标志寄存器的整数表示形式

11.2.2　ptr_dword/3

ptr_dword 返回字符串变量 StringVar 的内部地址，或把 StringVar 放到指定存贮单元。

调用格式：ptr_dword(StringVar, Seg, Off)　　/* (o,i,i),(i,o,o) */

当 StrignVar 受约束时，ptr_dword 返回其内部段及偏移地址；当 Seg 和 Off 受约束时，ptr_dword 把 StringVar 存入此处。

Turbo Prolog 中的串是以 '\0' 结尾的 ASCII 码序列，用户可以将本章的低级例程用于异常串（指含有多个 '\0' 的串），但不能将它们插入到数据库中。

11.2.3　membyte/3 和 memword/3

Turbo Prolog 提供了两个谓词用于读取和修改内存的特定单元。membyte 用于访问内存的一个字节；memword 用于访问一个字（双字节）。

调用格式：membyte(Segment, Offset, Byte)　　/* (i,i,i),(i,i,o) */

```
memword(Segment, Offset, Word)    /* (i,i,i),(i,i,o) */
```

Segment、Offset、Byte 和 Word 均定义为整数，许多 bios 调用要求以 Segment:Offset（段:偏移量）形式传送指针。membyte 和 memword 也要求这样形式的指针。内存位置的计

算如下：

```
((Segment * 16) + Offset)
```

11.2.4　port_byte/2

谓词 port　byte 允许从某个 I/O 端口读写一个字节

调用格式：prot　byte(PortAddress, Byte)　　/* (i,i),(i,o) */

其中 PortAddress 和 Byte 定义为整数。关于 I/O 端口的内容请参阅 DOS 技术手册。

11.3　例　　子

1. 下面的子句 set　mode 用 membyte 来读取内存 449h 处的字节，它含有当前显示模式的值。接着，set　mode 用 memword 读取内存 463h 位置处的一个字，该字包含当前显示适配器的端口地址。

本谓词写出当前的模式值，询问用户新的模式(0～7)，并用 port　byte 把显示器置为有效的新模式：

```
set　mode:-
    membyte( $0000, $0449, X),
    memword( $0000, $0463, PortAddress),
    write("Current mode setting is: ", X), nl,
    write("Enter new mode (0～7):"),
    readint(NewMode),
    NewMode <= 7 ,
    port　byte(PortAddress, NewMode).
```

2. 下列程序用 bios 和 ptr　word 定义了四个低级支撑谓词。这些谓词是：

dosver　：返回 DOS 版本号(以实数表示)；

diskspace：返回指定磁盘的总容量和可用字节数。磁盘由下列值指定：

0　表示默认驱动器；

1　表示 A 驱动器；

2　表示 B 驱动器；

3　表示 C 驱动器；

.

.

.

makedir　：创建一个子目录；

removedir：删除一个子目录。

程序清单如下：

```
predicates
  dosver(real)
  diskspace(real, real, real)
  makedir(string)
```

```
  removedir(string)

clauses
  dosver(Version) :- AX=48*256,
  bios(33, reg(AX, 0, 0, 0, 0, 0, 0, 0), reg(VV, , , , , , , )),
  /* You could use hex notation, bios($21...) instead of bios(33...) */
  L=VV div 6, H=VV-256*L, Version=H+L/100.

  diskspace(Disk, TotalSpace, FreeSpace) :-
      AAX = 54*256,
      bios(33, reg(AAX, 0, 0, DISK, 0, 0, 0, 0), reg(AX, BX, CX, DX, , , , )),
      FreeSpace = 1.0*BX*CX*AX, TotalSpace = 1.0*DX*CX*AX.

  makedir(Name) :-
      ptr  dword(Name, DS, DX),
      AX = 256*57,
      bios(33, reg(AX, 0, 0, DX, 0, 0, DS, 0), ).

  removedir(Name) :-
      ptr  dword(Name, DS, DX), AX=256*58,
      bios(33, reg(AX, 0, 0, DX, 0, 0, DS, 0), ).
```

键入以下目标之一即可启动上述程序中的相应谓词：

```
1. dosver(DOSVersionNumber).
2diskspace(DriveNumber, TotalSpace, RemainingSpace).
3makedir("testdir"),
 readchar( ),system("dir"),removedir("testdir"),
 readchar( ),system("dir").
```

第五部分

混合编程程序的调试

Turbo Debugger 调试的一个快速示例

第十二章 Turbo Debugger 调试的一个快速示例

如果用户急于使用 Turbo Debugger，而且不是那种有耐心，首先浏览一下本手册全部内容的人，本章将提供足够的调试知识来应付用户的第一个程序。一旦掌握了本章描述的基本概念，Turbo Debugger 组织合理而且直观的环境和与上下文相关的帮助系统将有助于用户边学边用。

本章介绍 Turbo Debugger 的所有基本特点。在描述了产品磁盘上的演示程序——一个为 C，另一个为 Pascal 的之后，它将告诉：

- 如何运行和中止程序。
- 如何考察程序变量的内容。
- 如何观察复杂的数据对象，如数组和结构。
- 如何改变变量的值。

12.1 演示程序

演示程序（用 C 编写的 TCDEMO.C 和用 Pascal 编写的 TPDEMO.PAS）将向用户介绍调试程序必须知道的两件事情：如何中止和启动用户程序；如何考察程序中变量的数据结构。两个演示程序本身并没有多大意思，其中有些代码和数据结构纯粹是为了显示 Turbo Debugger 的功能才设置的。

每一个演示程序都要求用户键入一些文本行或者一个数据文件名，然后对其中的字和字母进行计数。在程序的最后将显示文本的统计信息，包括每行的平均字数和每一字母出现的频率。

请用户确认在当前目录下包含有获得本章指导的必需的两个文件。对于 C 语言的示例而言，它们是 TCDEMO.C 和 TCDEMO.EXE；对于 Pascal 语言的示例而言，它们是 TPDEMO.PAS 和 TPDEMO.EXE。

为了启动 C 程序，请输入

```
TD TCDEMO
```

为了启动 Pascal 程序，请输入

```
TD TPDEMO
```

Turbo Debugger 将装载演示程序，显示初始屏幕（Startup screen）（见图 12.1），并将光标定位在程序的开始处。

初始屏幕包括菜单条(Menu bar)、模块(Module)窗口和监视(Watch)窗口，以及状态行(Status line)。

为了在任何时候退出指导过程并返回 DOS，可按 Alt－X 键。如果用户遵循指导的过程中完全迷失了方向，可按 Ctrl－F2 键重新装载程序并从头开始学习。然而，Ctrl－F2 键并不清除断点和被监视的变量，用户必须用 Alt－F O 来清除。Alt－B D 当然也可以清除所有断点，但有时用 Alt－F O 重新加载会快一些。

在任何时候都可以按 F1 键获得当前窗口、菜单命令、对话框或者错误信息的有关帮助。浏览一下整个菜单系统，在每个命令下按 F1 键后阅读它的功能概括，这样用户就可以学到不少知识。

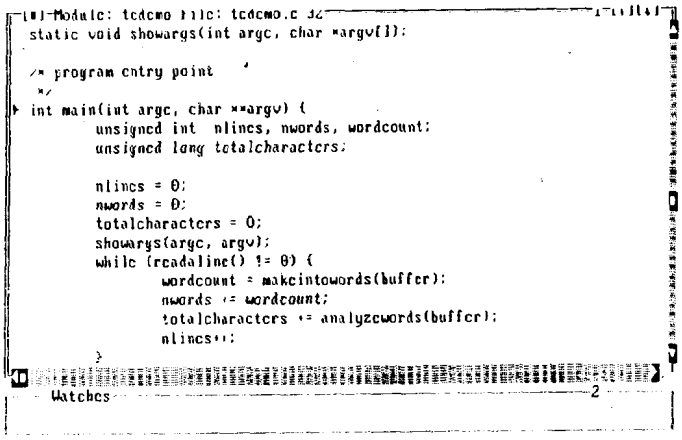


图 12.1　显示 TCDEMO 的初始屏幕

12.2　使用 Turbo Debugger

12.2.1　菜　　单

在屏幕顶部显示的是菜单条(见图 12.2)。为了从菜单条上下拉出一个菜单，先按 F10 键，再用←或→键使想要的选择项变为高亮度，最后按 Enter 键即可。另一种方法是按 Alt 与某菜单名第一个字母的组合键。

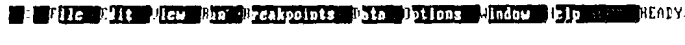


图 12.2　菜单条

现在按 F10 键，注意光标从模块窗口中消失，条形菜单上的 File 命令变为高亮度。这时，屏幕的底行也发生了变化，以显示 File 菜单所包括的命令种类。

用方向键在菜单系统中移动，按↓键可以从菜单条高亮度的条目上下拉出一个菜单。

用鼠标器揿按菜单条中的条目也可以打开一个菜单。

按 Esc 键可以在菜单系统中逐级退出。当菜单条上只有一个菜单条目为高亮度时，按 Esc 键将返回到模块窗口下，这时菜单条就不再为活动状态了。

12.2.2 状态行

屏幕底部的状态行(见图 12.3)显示的是有关功能键及其用途。

状态行的内容随着用户进入的不同地方而发生变化(如菜单命令、对话框中的数据等等)。例如按住 Alt 键一会儿，可以发现状态行的内容改变为用户可以用 Alt 键获取的功能键的信息。

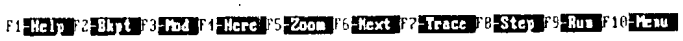


图 12.3 状态行

现在按住 Ctrl 键一会儿，状态行显示的命令是当前窗口区(pane)所对应的局部菜单命令(local menu command)的热键。它们随着用户所在的窗口类型和区的不同而发生变化。关于这一点，以后有更详细的说明。

一旦进入了菜单系统，状态行将再次改变为显示当前高亮度菜单选择的功能。按 F10 键进入菜单条，再使用→键使 File 选择变为高亮度。这时状态行显示“File Orientedfunctions”。使用↓键在 File 菜单的选择项之间滚动，与此同时可以观察到信息的变化。按 Esc 键或用鼠标器揿按模块窗口可以离开菜单系统。

12.2.3 窗 口

窗口占据了大部分屏幕，用户可以通过不同的窗口来考察程序的各个部分。

显示开始时有两个窗口：模块窗口和监视窗口(见图 12.4)。在打开更多的窗口或者调整这两个窗口之前，它们将占满整个屏幕并且不互相重叠(tiled)。新的窗口将自动与现有窗口重叠直至用户移动它们为止。

请注意模块窗口的边框是双线的，并带有一个高亮度的标题，这意味着它是活动窗口。可以使用光标键(方向键、Home、End、PgUp 键等等)在活动窗口中移动光标。现在按 F6 键转到另一个窗口。这时模块窗口变为活动窗口，带有双线边框和高亮度标题。

用户可以使用 View 菜单下的命令来创建新的窗口。例如，选择 View | Stack 将打开一个堆栈(Stack)窗口，该窗口在模块窗口的上面弹出。现在按 Alt—F3 键来删除活动窗口，则堆栈又消失了。

Turbo Debugger 能保存最近一次关闭的窗口，这样就可以在需要时恢复它。如果用户

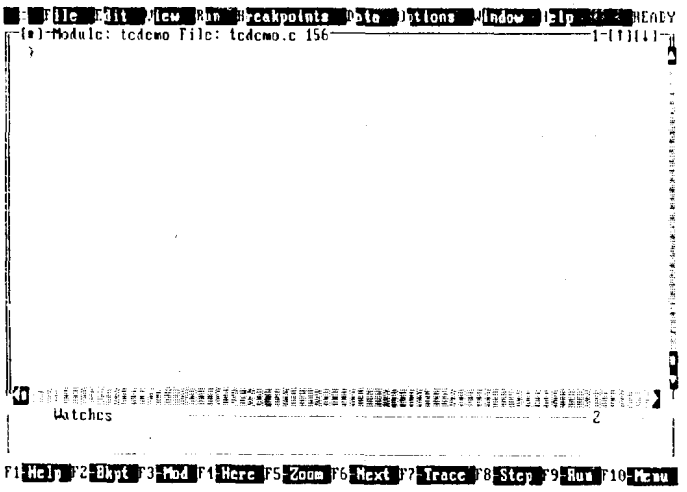


图 12.4　铺满屏幕的模块与监视窗口

不小心关闭了一个窗口，可以选择 Window | Undo Close。在本例中，堆栈窗口又重新出现。用户也可以按 Alt—F6 键来恢复最近一次关闭的窗口。

Window 菜单包括可用于调整已经在屏幕上的窗口的命令。用户可以在屏幕上移动窗口，也可以改变它的大小(使用 Ctrl—F5 键也可达到相同效果)。

选择 Window/Size/Move 并用方向键(arrow key)可在屏幕上重新设置活动窗口(此处为堆栈窗口)的位置。然后按住 Shift 键，这时可用方向键来调整窗口的大小。在用户选择好他所喜欢的大小与位置后，可按 Enter 键。

现在按 Alt—F3 键删除堆栈窗口，准备阅读本手册另一节的内容。如果用户加载了 C 演示程序，那么请接着看下一节——使用 C 演示程序；如果加载的是 Pascal 演示程序，那么就请跳到“使用 Pascal 演示程序”一节。

12.3　使用 C 演示程序

模块窗口左列的箭标( )表示 Turbo Debugger 在该处中止了用户程序。即然用户还没有运行他的程序，该箭头位于程序的第一行。按 F7 键执行单一的一行源程序，则箭标与光标处于下一个可执行的程序行上。

模块窗口标题的右边空白上显示了光标所在行，用方向键上下移动光标可以观察到标题中行号的变化情况。

正象从 Run 菜单中看到的那样，有一些方法可以控制程序的运行过程。现在假定用户想让程序一直执行到第 39 行.

首先把光标定位到第 39 行上，然后按 F4 键。这样做的结果是让程序执行到第 39 行(但不包括该行)为止。现在按 F7 键，其作用是一次执行一行源代码。在本例中按 F7 键使程序执行第 39 行对函数 showargs 的调用，从而使光标跳到第 151 行定义函数 showargs 的地方。

继续按 F7 键将使用户单步走完函数 showargs，然后返回到该调用的下一行——第 40 行。这里我们直接用 Alt－F8 键使程序停留在函数 showargs 的返回处(见图 12.5)，这同样使用户回到第 40 行上。在需要跳过函数的剩余部分时，Alt－F8 是一个很有用的命令。

图 12.5　程序停留在函数 showargs 的返回处

为了让程序运行到一个特定位置，可以直接给出函数名或者行号，而不必把光标先移到源文件中该行上再执行到那一点为止。按 Alt－F9 键可指定一运行暂停标号。这时出现一对话框，输入 readaline 并按 Enter 键，于是程序开始运行，直到停止在函数 readaline (第 142 行)的开头处为止。

12.3.1　设置断点

另一个控制程序停在某处的方法是设置断点。设置断点最简单的办法是用 F2 键把光标移动第 44 行并按 F2 键，Turbo Debugger 使该行成为高亮度，表明其上设了一个断点。

用户也可以用鼠标器来设置或清除断点，其方法是撒按模块窗口的前两列。

现在按 F9 键不间断地执行用户程序。屏幕切换到程序显示，这时演示程序正在运行并且在等待用户输入一行正文。请输入 abc，空格，def，再按 Enter 键。显示又回到 TurboDebugger 屏幕并且箭标位于第 44 行(见图 12.6)，在该处已设置了让程序停下来的断点。现在再按 F2 键把该断点取消掉。

请参阅《Turbo Debugger 用户手册》有关断点的描述，包括条件(conditional)和全局(global)断点。

12.3.2　利用监视(Using watches)

屏幕底部的监视窗口显示用户所指定的变量的值。例如，如果想监视变量 nwords 的值，可把光标移到第 42 行的变量名上并选择模块窗口局部菜单的 Watch 命令(可用 Alt－F10 键获取该命令，或者用状态行上的缩写形式 Ctrl－W)。

有鼠标器时可撒按状态行上的 Ctrl－W。

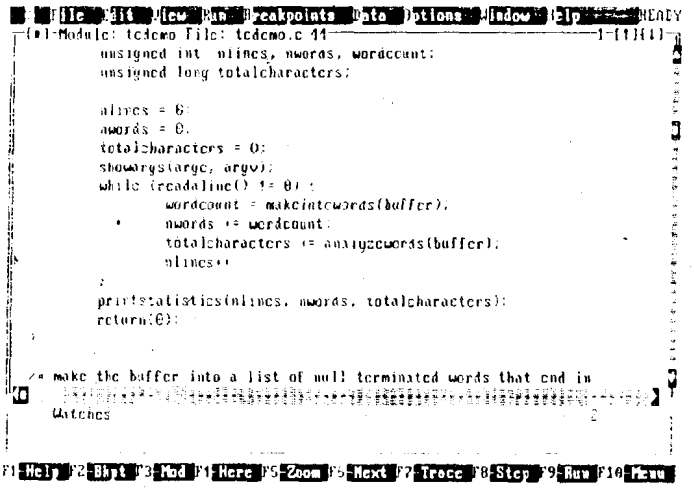


图 12.6　在第 44 行上设置断点

现在 nwords 出现在屏幕底部的监视窗口中，同时带有它的类型(unsigned int)和值。在执行程序时，Turbo Debugger 会更新其中的值以反映变量的当前值。

12.3.3　考察简单的 C 数据对象

一旦程序停止运行，通过 Inspect 命令就可以有一些方法观察数据。这种强有力的设施使用户象在写程序时一样来考察数据结构。

用 Inspect 命令(在各种局部菜单和 Data 菜单下)可以考察任何指定的变量。例如，想查看一下变量 nlines 的值，可把光标移动 nlines 中的一个字母上，从模块窗口局部菜单中选择 Inspect 命令(即按 Ctrl—I 键)，一个考察窗口就被弹出来了(见图 12.7)。

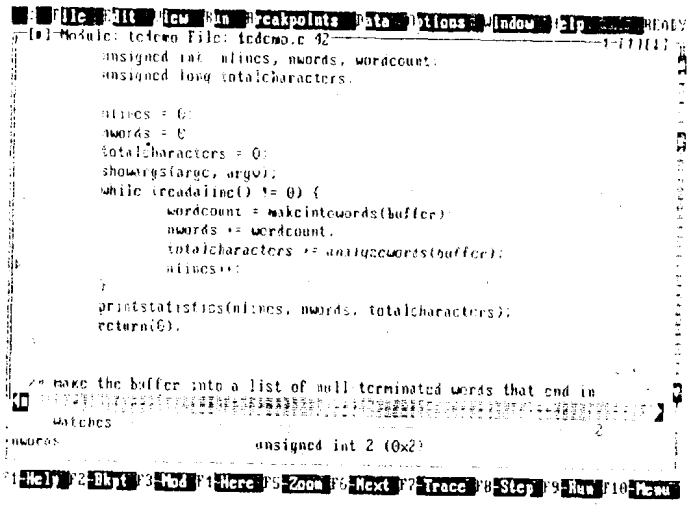


图 12.7　监视窗口中的 C 变量

标题行显示的是变量名，下一行显示的是它的内存地址，第三行表明 nlines 所存贮的数据类型(C 语言的 unsigned int 类型)，它的右边为该变量的当前值。

在考察了变量之后，按 Esc 键关闭考察窗口。用户也可以用 Alt－F3 键删除考察窗口，就象任何其它窗口一样，或者用鼠标器撳按关闭框(close box)。

现在复习一下在这里实际学到的东西。按 Ctrl 键得到模块窗口中局部菜单命令的缩写，再按 I 指定用 Inspect 命令。

为了考察在模块窗口中并不能很方便地显示的数据项，选择 Data/Inspect。这时出现一个对话框，要求用户输入待考察的变量。请键入 letterinfo 并按 Enter 键，此时出现一考察窗口，显示 letterinfo 数组元素的值。考察窗口的标题显示用户正在考察的数据的名称，标题下的第一行是数组 letterinfo 第一个元素在内存中的地址。使用方向键可在构成数组 letterinfo 的 26 个元素之间滚动。下一节将介绍如何考察这一复杂数据对象。

12.3.4 考察复杂的 C 数据的对象

一个复杂的数据对象，如数组或者结构，包含多个元素。移动到数组 letterinfo 的第 4 个元素(由[3]指示)上，按 Alt－F10 键打开考察窗口的局部菜单并按 I 选择 Inspect。这时，出现了一个新的考察窗口，如图 12.8 所示，显示数组中该元素的内容。这个考察窗口(见图 12.9)展示的是类型为 linfo 的结构的内容。

图 12.8 一个考察窗口

当光标放在一个成员名上时，该成员的数据类型就出现在考察窗口底部的区域上。如果这些成员之一又是一个复杂的数据对象，则可以发出一个 Inspect 命令去进一步深入到数据结构之中。

按 Alt－F3 键同时删除两个考察窗口并返回到模块窗口下(Alt－F3 是一次删除多个考察窗口的简便方法。如果用 Esc 键，只有最近的考察窗口才会被删除)。

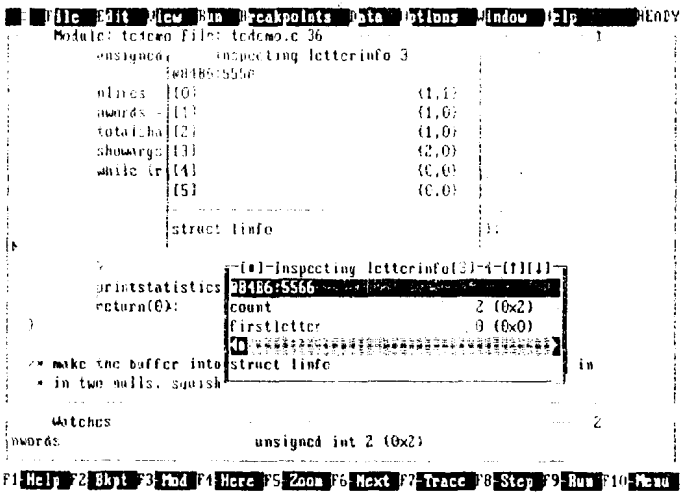


图 12.9　考察一个结构

12.3.5　改变 C 数据值

到现在为止，用户已经学会了如何观看程序中的数据。现在，让我们来改变数据项的值。

使用方向键移到源文件的第 38 行。把光标放在变量 totalcharacters 上面并按 Ctrl－I 考察其值。在考察窗口打开的情况下，按 Alt－F10 键打开考察窗口局部菜单并选择 Change（也可以直接按 Ctrl－C 键来完成这项工作）。这时出现了一个对话框，如图 12.10 所示，要求用户输入新值。

到了这里，用户可以输入任何可计算出一个数值的 C 表达式。键入 totalcharacter＋4 并按 Enter 键，考察窗口就显示出新值 10L(0XA)。

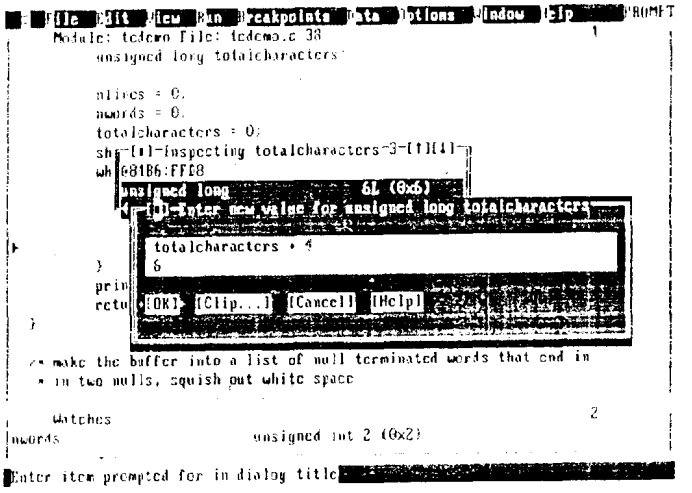


图 12.10　一个修改对话框

图 12.11 求值/修改对话框

为了修改不在模块窗口中显示的数据项的值，选择 Data/Evaluate/Modify。这时出现一对话框(见图 12.11)，在第一个输入框中输入待修改的变量名：键入 argc 并按 Enter 键。然后按 Tab 键两次，移到标有 New Value(新值)的输入框下。输入 123 并按 Enter 键，结果(在第二个框中)就改变为 int 123(0X7B)。 以上是有关如何利用 Turbo Debugger 来调试 Turbo C 程序的快速介绍，更为详细的调试示例请见《Turbo Debugger 用户手册》。

12.4 使用 Pascal 示例程序

模块窗口在列的箭标( )表示 Turbo Debugger 在该处中止了用户程序。即然用户还没有运行他的程序，该箭头位于程序的第一行。按 F7 键执行单一的一行源程序，则箭标与光标处于下一个可执行的程序行上。

模块窗口标题的右边空白上显示了光标所在行，用方向键上下移动光标可以观察到标题中行号的变化情况。

为了让程序一直执行到第 221 行，把光标移到该行上并按 F4 键。这时 TPDEMO 提示用户输入一串字符，键入 ABC、空格、DEF，并按 Enter 键。现在光标仍停留在第 221 行，按 F7 键两次再执行两行源代码。因为执行的第二行是对一个另一个过程的调用，所以现在箭标出现在该函数 processline 的第一行上。若继续使用 F7 键将单步走完函数 processline，然后返回到该调用的下一行——第 224 行。这里，我们直接用 Alt—F8 键使程序停留在函数 processLine 的返回处(见图 12.12)。在需要跳过函数或过程的剩余部分时，命令 Alt—F8 就非常有用。

为了让程序运行到一个特定位置，用户可以直接给出函数名或者行号，而不必把光标先移到源文件中该行上再执行到那一点为止。按 Alt—F9 键可指定一运行暂停标号。这里出现一对话框，输入 GetLine 并按 Enter 键，于是程序开始运行，直到停止在函数 GetLine 开头处为止。

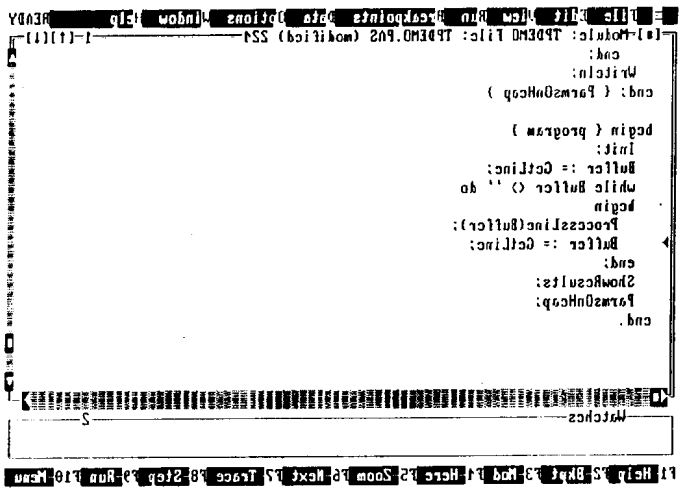


图 12.12　程序停留在过程返回处

12.4.1　设置断点

另一个控制程序停留在某处的方法是设置断点。设置断点最简单的办法是用 F2 键。把光标移到第 121 行并按 F2 键，Turbo Debugger 使该行成为高亮度，表明其上设了一个断点（见图 12.13）。

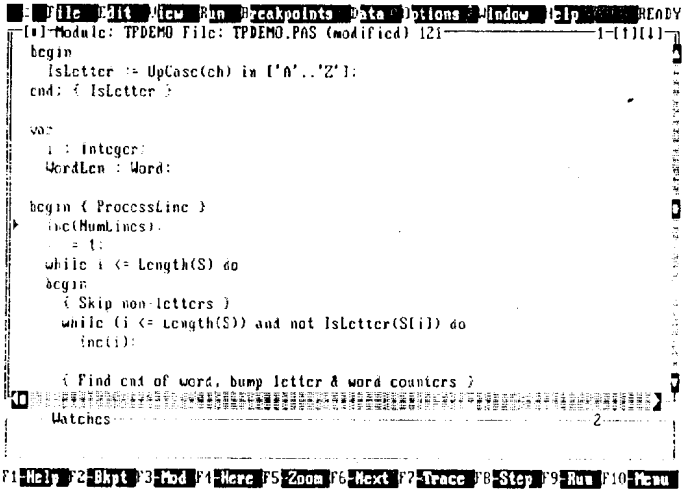


图 12.13　在第 121 行上设置断点

用户也可以用鼠标器来设置或清除断点，其方法是揿按模块窗口的前两列。

现在按 F9 键不间断地执行用户程序。屏幕切换到程序显示，这时演示程序正在运行并且在等待用户输入一行正文。请输入 abc、空格、def，再按 Enter 键。显示又回到 TurboDebugger 屏幕并且箭标位于第 121 行，在该处已设置了让程序停下来的断点。现在再按 F2 键把

该断点取消掉。

12.4.2 使用监视

屏幕底部的监视窗口显示用户所指定的变量的值。例如，如果想监视变量 NumWords 的值，可把光标移到第 144 行的变量名上并选择模块窗口局部菜单的 Watch 命令（可用 Alt—F10 键获取该命令，或者用状态行上的缩写形式 Ctrl—W）。

有鼠标器时可揿按状态行上的 Ctrl—W。

现在 NumWords 出现在屏幕底部的监视窗口中，同时带有它的类型（Word）和值。在执行程序时，Turbo Debugger 会更新其中的值以反映变量的当前值。

12.4.3 考察简单的 Pascal 数据对象

一旦程序停止运行，通过 Inspect 命令就可以有一些方法观察数据。这种强有力的设施使用户象在写程序时一样来考察数据结构。

用 Inspect 命令（在各种局部菜单和 Data 菜单下）可以考察任何指定的变量。例如，想查看一下变量 NumLines 的值，可把光标移动回到第 121 行变量 NumLines 的一个字母上，从模块窗口局部菜单中选择 Inspect 命令（即按 Ctrl—I 键），一个考察窗口就被弹出来了，如图 12.14 所示。

标题行显示的是变量名，下一行显示的是它的内存地址，第三行表明 NumLines 所存贮的数据类型（Passal 语言的 Word 类型），它的右边为该变量的当前值。

在考察了变量之后，按 Esc 键关闭考察窗口。也可以用 Alt—F3 删除考察窗口，就象任何其它窗口一样，或者用鼠标器揿按关闭扭。

现在复习一下这里实际学到的东西。按 Ctrl 键得到模块窗口中局部菜单命令的缩写，再按 I 指定用 Inspect 命令。

图 12.14 监视窗口中 Pascal 的变量

为了考察在模块窗口中并不能很方便地显示的数据项，选择 Data | Inspect。这里出现

图 12.15　一个考察窗口

一个对话框，要求用户输入待考察的变量。请键入 LetterTable 并按 Enter 键，此时出现一考察窗口，显示 LetterTable 的值。使用方向键可在构成数组 LetterTable 的 26 个元素之间滚动。考察窗口的标题显示正在考察的数据的名称和类型，就象用户在源文件中声明该数据完全一样。下一节将介绍如何考察这一复杂数据对象。

12.4.4　考察复杂的 Pascal 数据对象

一个复杂的数据对象，如数组或者结构，包含多个元素。光标移动到数组 letter Table 的第 4 个元素（由['D']指示）上，按 Alt－F10 键打开考察窗口的局部菜单并按 I 选择 Inspect。这时，出现了一个新的考察窗口（见图 12.15），显示数组中该元素的内容。这个考察窗口展示的是类型为 LInfoRec 的结构的内容。

当光标放在一个成员名上时，该成员的数据类型就出现在考察窗口底部的区域上。如果这些成员之一又是一个复杂的数据对象，则可以发出一个 Inspect 命令去进一步深入到数据结构之中。

按 Alt－F3 键同时删除两个考察窗口并返回到模块窗口下（Alt－F3 是一次删除多个考察窗口的简便方法。如果用 Esc 键，只有最上面的考察窗口才会被删除）。

12.4.5　改变 Pascal 数据值

到现在为止，用户已经学会了如何观看程序中的数据。现在，让我们来改变方向数据项的值。

使用键移动源文件的第 103 行，把光标放在变量 NumLetter 上面并按 Ctrl－I 考察其值（见图 12.16）。在考察窗口打开的情况下，按 Alt－F10 键打开考察窗口局部菜单并选择 Change（也可以直接按 Ctrl－C 键来完成这项工作）。这时出现了一个对话框，要求用户输入新值。

到了这里，用户可以输入任何可计算出一个数值的 Passal 表达式。键入 NumLetters＋4

图 12.16 考察一个记录

并按 Enter 键，考察窗口就显示出新值 10。

为了修改不在模块窗口中显示的数据项，选择 Data/Evaluate/Modify。这时出现一对话框(见图 12.17)，在第一个输入框中输入待修改的变量名：键入 NumLetters 并按 Enter 键，结果就就显示在中间区里。然后按 Tab 键两次，输入 123 并按 Enter 键，变量 NumLines 的值就被设置为 123 了(见图 12.18)。

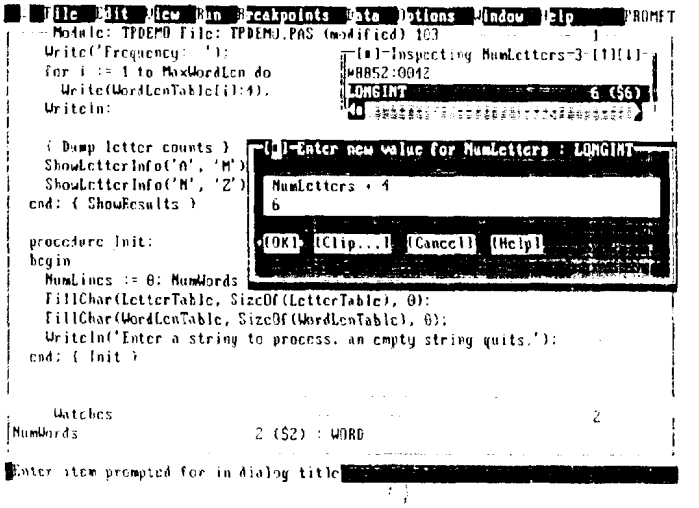


图 12.17 一个修改对话框

上述内容就是如何利用 Turbo Debugger 来调试 Turbo Pascal 程序的快速介绍，《TurboDebugger 用户手册》将提供更为广泛的调试示例。

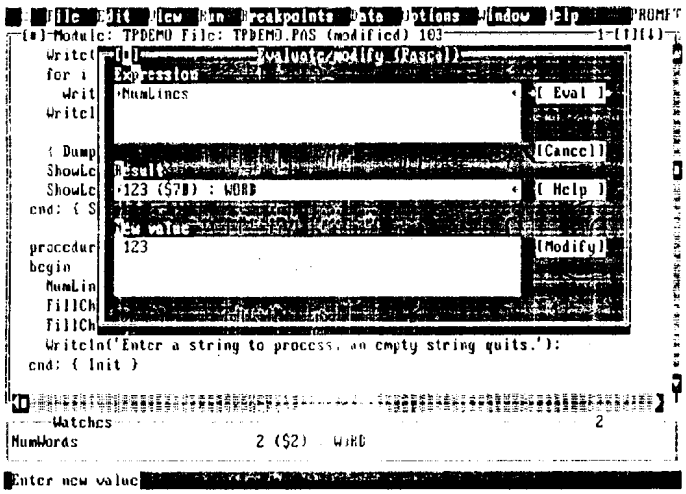


图 12.18　求值/修改对话框

第十三章

启动 Turbo Debugger

本章要介绍的是如何准备待调试的程序。除了引导用户在 DOS 命令行上启动 Turbo Debugger 和裁剪众多的命令行选择项以适应待调试的程序之外，将解释如何把这些命令行选择项永久地保存在配置文件中。在本章中，用户还将学会在 Turbo Debugger 调试过程中运行 DOS 命令处理器的方法以及如何在最后结束工作时返回 DOS。

13.1 准备待调试的程序

当编译和连接一种 Borland 公司的语言时，用户可以告诉编译器产生完全的调试信息。如果用户已经不带任何调试信息地编译了程序的目标模块，那么必须重新编译所有模块，以获得贯穿整个程序的完全的源级调试功能。虽然允许只对特定的模块产生调试信息（在调试一个大程序时也许不得不如此），但用户稍后会发现进入一个不带任何调试信息的模块是很令人恼火的。因此，我们建议用户重新编译所有模块。

13.1.1 准备 Turbo C 程序

如果用户正在使用 Turbo C++集成环境（TC），那么在编译源模块时，打开调试对话框（即选择 Options/Debugger）并且把源级调试无线电按钮（radio button）设置为 Standalone。对于 Trubo C 2.0 来说，则需要把 Debug/Source 设置为 Standalone。

如果用户正在使用命令行编译器（+CC），请指定－V 命令行选择项。

如果用户正在使用 TLINK 作为独立运行的连接器，必须用/v 选择项把调试信息添加到 EXE 文件尾部。

用户还应该留意不使用优化（Optimizing）选择，或者不使用－O 选择项，或者指定－O－来关闭 TURBOC.CFG 文件中的－O 选择。这样做可以消除在单步运行程序时，Turbo Debugger 跳过几行源代码的少数情况。

13.1.2 准备 Turbo Pascal 程序

首先必须确认用户使用的是不低于 5.0 版的 Turbo Pascal，因为早期的版本没有能力把调试信息加进 EXE 文件以致 Turbo Debugger 不使用它。

如果用户正在使用集成环境（TURBO.EXE），请在 Debug 菜单下把 Standalone Debugging 设置为 ON。然后把 Options/Compiler/Debug Information 也设置为 ON 或者使用{$D+}编译器指示。如果想访问局部符号（在过程和函数中申明的任何符号），可以把 Options/

Compiler/Local Symbols 设置为 ON 或把{$L+}(就这样书写,不留空格)加到程序开头处。这时,就可以开始编译用户程序了。

如果用户正在使用命令行版本(TPC.EXE),则必须用/v 选择项进行编译。调试信息和局部符号都是缺省产生的,如果不需要它们,可指定/$命令行选择项。

13.1.3　准备 Turbo 汇编程序

为了调试 Turbo 汇编程序,可指定-zi 命令行选择项以获得完全的调试信息。

在用 TLINK 连接程序时,使用/v 选择项可以把调试信息附加到 EXE 文件后面。

13.1.4　准备 Microsoft 程序

请参见产品磁盘上的文档以了解如何使用实用程序 TDCONVRT.EXE,它将把可用 Code View 调试的程序转换为 Turbo Debugger 格式。

13.2　运行 Turbo Debugger

为了使用 Turbo Debugger 调试程序,在 DOS 提示符下简单地打入 TD 后跟一些命令行参数和程序名,然后按 Enter 键。于是 Turbo Debugger 就装载用户程序并显示源代码,这样用户就可以逐条语句地执行程序了。

命令行启动的一般形式为:

TD [options] [progname [progargs]]

方括号内的条目为可选项。如果使用了一个可选项,在输入时要去掉方括号。progname 是待调试的程序名,程序名后可跟参数。下面是一些命令行的例子:

命令(command)	动作(Action)
td-sc prog1 a b	用-sc 选择项启动调试器,装入程序 prog1,两个命令行参数为 a 与 b
td prog2 -x	以缺省方式启动调试器并装入程序 prog2,程序的一个参数为-x

如果用户简单地输入 TD 并按 Enter 键,Turbo Debugger 使用其缺省选择项启动。

当用户在 Turbo Debugger 中运行程序时,必须同时有 EXE 文件和原始的源文件存在。Turbo Debugger 依次在下列目录中搜索源文件:(1).编译器编译源文件对源文件所在的目录;(2).Options|Path 中指定的 Source 命令目录;(3).当前目录;(4).EXE 文件所在目录。

在使用 Turbo Debugger 调试之前,必须已经把源代码编译为可执行的 EXE 文件,并且带有完全的调试信息。

注意:只有用 Turbo Pascal 5.0、Turbo C 2.0、Turbo 汇编 1.0 或者它们的更高版本编写的程序才能使用 Turbo Debugger 来调试。

如果在 DOS 提示符下运行程序时发现了错误,那么就可以退出该程序并用调试器加载它,这样就可以开始调试了。

13.3 命令行选择项

所有 Turbo Debugger 命令行选择项都以连字符(-)开头,它们和 TD 命令以及彼此之间至少有一个空格相隔。用户可以显式地把一个命令行选择项关闭,其方法是在该选择项后再加一个连字符。例如,-vg-将关闭掉完全图形保存的选择。当一个选择项永久地设置在配置文件中时,用户可以这样来关闭它。用户也可以使用 TDINST 配置程序来修改配置文件。

下面的几节将描述所有的命令行选择项。

13.3.1 装载配置文件(-c)

该选择项装载指定的配置文件。-c 与文件名之间不可以有空格。如果没有使用-c 选择项,将装载 TDCONFIG.TD(如果它存在的话)。例如:

```
TD -cMYCONF.TD TCDEMO
```

本例装载配置文件 MYCONF.TD 和 TCDEMO 的源代码。

13.3.2 显示更新方式(-d)

所有的-d 选择项都影响显示更新(display undating)操作的方式。

-do 在辅助显示(secondary display)中运行 Turbo Debugger,在主要显示(primarydisplay)中观看用户程序的屏幕。

-dp 使用彩色显示的缺省选择项。在一个显示页(display page)中显示调试器,而在另一个中显示正在被调试的程序,这样做最大程度地减少了在两个屏幕之间切换的时间。当被调试的程序本身使用了多个显示页时,不能选择-dp。

-ds 使用单色显示的缺省选择项。它维持调试器与被调试程序的独立的屏幕图象。每次运行用户程序或者调试器重新起动时,都从内存中加载整个屏幕的内容。这是显示两个屏幕图象最费时的方法,但它在任何硬件上都能工作,也能适用于那些对显示采取了特殊处理的程序。

13.3.3 获取帮助(-h 与-?)

在屏幕上显示有关 Turbo Debugger 命令行句法和选择项的帮助信息。

13.3.4 进程 ID 转换(-i)

该选择项允许进程 ID 转换。当用户正在 DOS 内部进行调试或者 DOS 系统调用被激活时,不要使用这个选择项。关于这一特性更为详细的技术资料,请参见附录 B。在调试大多数程序时,用户不必关心这个选择项。

13.3.5 击键记录(-k)

该选择项允许在 Execution History(执行历史)窗口的 Keystroke Recording 区中记录所按过的键。如果使用了该选择项,在调试过程中输入的所有键将被记录到一个磁盘文件中。然后,让 Turbo Debugger 重新加载程序并且重用记录的击键,这样用户就可以恢复到调试

过程以前的某一点上。此项功能同时记录在 Turbo Debugger 和用户程序运行时所按下的键。

13.3.6　汇编模式启动(-l)

这个选择项强迫 Turbo Debugger 以汇编方式起动，显示 CPU 窗口。Turbo Debugger 将不执行程序的起动代码(startup code)，而这些代码在把程序装入调试器时通常是自动被执行的。这意味着用户可以单步执行起动代码。

13.3.7　设置堆大小(-m)

这个选择项把 Turbo Debugger 使用的工作堆(working heap)设置为 NK。其句法为：

-mN 其中 N 为以一千字节为单位的计数。-m 选择项与堆的大小之间不可以存在空格。下面是一个例子：

```
TD -m/0 TCDEMO.EXE
```

缺省的堆大小为 18K，最低边界为 7K。如果用户需要内存，可使用该选择项减少 TurboDebugger 所占用的堆的大小。Turbo Debugger 把一些暂时信息，如命令历史表和断点等保存在堆中。

注意如果用-m 命令行选择(-m0)把堆大小指定为 0，则 Turbo Debugger 将使用它能够使用的最大的堆，通常为 18K。

13.3.8　鼠标器支持(-p)

这个选择项允许获取鼠标器支持。然而，由于在 Turbo Debugger 中缺省的鼠标器支持为 ON，所以-p 选择项对你没有多大用处，除非已使用 TDINST 中把该项缺省选择改变为 OFF。如果用户不想用鼠标器，可使用-p-。

13.3.9　远程调试(-r)

所有-r 选择项都影响远程调试的连接。

-r　允许通过串行连接在远程系统上调试。除非用户在 TDINST 中做了修改，将使用缺省的串行端口(CDM1)和速率(115K 波特)。

-rpN将远程连接端口设置为端口 N。N 值为 1 或 2，分别表示 COM1、COM2.

-rsN设置远程连接速率。N 可取 10(9600 波特)，2(40K 波特)，3(115K 波特)。

13.3.10　源代码处理(-s)

所有-s 选择项都影响 Turbo Debugger 处理源代码和程序标识符的方式。

-sc　在输入符号名时忽略大小写，即使程序在连接时区分了大小写也如此。如果没有-sc 选择，Turbo Debugger 只有在连接程序时忽略了大小写的情况下，才会不区分大小写。本选择对 Pascal 程序无效，因为它本身就与大小写无关。

-sd　设置一个或多个源文件搜索目录。句法为：

```
-sddirname
```

为了设置多个目录，可以反复使用-sd 选择项——每个-sd 选择项只能指定一个目录名。dirname 可以是相对的或者绝对的目录名，并且可以包含一个磁盘字母。如果配置文件中指定了目录，则-sd 声明的目录将附加到后面。

-smN 设置符号表保留内存的大小。在它后面带有一个以一千字节为单位的计数值 N 指明用户想保留的空间。如果想利用 File/Symbol Load 命令手工加载符号表，可以选用-smN。用户也许需要实验一下到底保留多少内存才合适。

13.3.11 视频硬件(-v)

所有-v 选择项都影响 Turbo Debugger 参视频硬件(video hardware)的处理。

-vg 保存程序屏幕的完整的图形映象。这需要额外的 8K 内存，但可以调试使用某些图形显示方式的程序。如果用户程序在 Turbo Debugger 下运行时发生了图形混乱，可试用一下这个选择项。

-vn 不允许使用 43/50 行显示。指定该选择项可以节约一些内存。当在 EGA 或者 VGA 上运行程序并且在 Turbo Debugger 运行时不会使用 43/50 行方式时，可以选用-vn。

-vp 允许保存 EGA 调色板(palette)。

13.3.1.2 覆盖池大小(-y)

不管是在主存(main memory)还是在 EMS 内存中，-Y 选择项都用来设置覆盖池(overlaypool)的大小。

-yN 本选择项用来设置在主存中的覆盖池大小。其中 N 是以一千字节为单位的计数值，它指明用户想保留的覆盖池大小。通常，Turbo Debugger 使用 80K 的代码池(code pool)，而用户可指定的最大与最小值分别为 200K 和 20K。使用 TDINST 可设置永久的覆盖代码池(overlay code poll)大小。如果没有足够的内存在 Turbo Debugger 下装入用户程序，或者在调试小程序时想提高 Turbo Debugger 的性能，都可以使用-yN。代码池越小，Turbo Debugger 从磁盘下装入程序覆盖的次数就越频繁，它反应的速度也越慢。如果代码池较大，被调试程序可占用的空间就要小一些，但 Turbo Debugger 运行得却较快。

-yeN 本选择项用来设置 EMS 内存中的覆盖池大小。如果需要释放一些 EMS 内存供被调试程序使用，可选择-yeN。其中 N 是以 16KEMS 页为单位的计数值，它指明了用户想保留的覆盖池的大小。例如，-ye4 将设置覆盖池大小为 4 页。覆盖池大小的缺省值为 12 个 16KEMS 页。使用-ye0 可取消 EMS 覆盖池。

13.4 配置文件

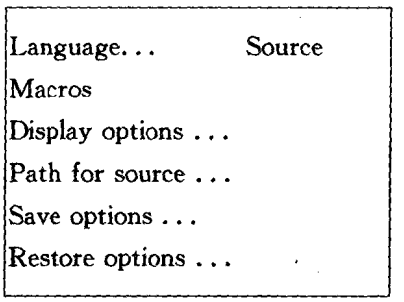
Turbo Debugger 利用配置文件取代命令行选择项的内在缺省值。如果不用配置文件，用户可以通过 TDINST 设置这些选择项的 Turbo Debugger 缺省值。用户也可以通过 TDINST 建立配置文件。

Turbo Debugger 寻找配置文件 TDCONFIG. TD 的顺序是：(1). 在当前目录；(2). 在 TDINST 安装程序中设置的 TURBO 目录；(3). 在包含 TD. EXE 的目录。如果省的 DOS2.

X，TurboDebugger 不会在 TD.EXE 的目录寻找 TDCONFIG.TD。

如果 Turbo Debugger 找到了配置文件，其中的设置将取代原来缺省的值。在启动 Turbo Debugger 时提供的任何命令行选择项将取代相应的缺省值以及 TDCONFIG.TD 中的任何设置。

13.5　选项菜单

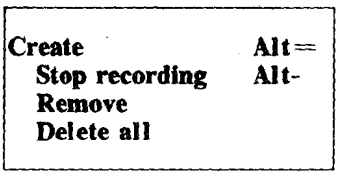


选项(Options)菜单允许用户设置或调整控制 Turbo Debugger 总的外部特性(appearance)和操作(operation)的一些参数。下面的几节将介绍每个菜单命令，并且指明在手册的哪些地方可以找到更为详细的描述。

13.5.1　语言命令

介绍如何设置当前表达式语言，以及它如何影响用户输入表达式的方式，请见《Turbo Debugger 用户手册》。

13.5.2　宏菜单



Macros(宏)命令显示另一个菜单，它允许定义一个新的击键宏或者删除已经赋给一个键的宏。该菜单有如下命令：创建(**Create**)、停止记录(**Stop Recording**)、删除(**Remove**)和全清(**Delete All**)。

13.5.2.1 创建(Create)

开始记录赋给一个键(如 Alt-M)的击键。为了开始一个记录过程，先选择 Options/Macros/Create，它提示用户输入要赋之于宏的键。在记录过程中，屏幕的右上角将显示信息 RECORDING。这时，输入要记录的键。这些键通过 Turbo Debugger 起作用，就好象没有记录宏时一样。一旦结束了记录击键的过程，可发出 Options/Macros/Stop Recording 命令或按其热键 Alt-。用户也可以再按一次赋之于宏的键(此处为 Alt-M)来结束记录过程。

Alt=是开始记录宏的热键。

13.5.2.2　停止记录(Stop Recording)

停止记录赋给一个键的击键。在发出 Options/Macros/Create 命令把一些击键赋给一个

键后使用该命令，也可用结束宏的热键 Alt-。

13.5.2.3　删除(Remove)

删除赋给一个单键的宏。用户将被提示按欲删除宏的键。

13.5.2.4　全清(Delete All)

删除所有击键宏定义并把所有相应键恢复为它们原有的含义。

13.5.3　显示选择命令

本命令打开一对话框，其中选项用以控制调试器显示的外部特性。

13.5.3.1 显示切换

显示切换(Display swapping)选项有三种互斥的选择：

None：　不在两个屏幕之间切换。在调试不向用户屏幕输出信息的程序时选用。

Smart：　只有在显示输出信息时才切换到用户屏幕。当用户跳过一个子程序或者执行打算对视频存储器进行读或写操作的指令与源程序行时，Turbo Debugger 就负责切换屏幕。Smart 是缺省选择。

Always：　每次用户程序运行时都切换到用户屏幕。在 Smart 选择不能捕获到所有程序向屏幕的写操作时使用。如果用户选择了 Always，则每次单步执行程序时屏幕都会闪烁，调试器每执行一条指令都切换屏幕。

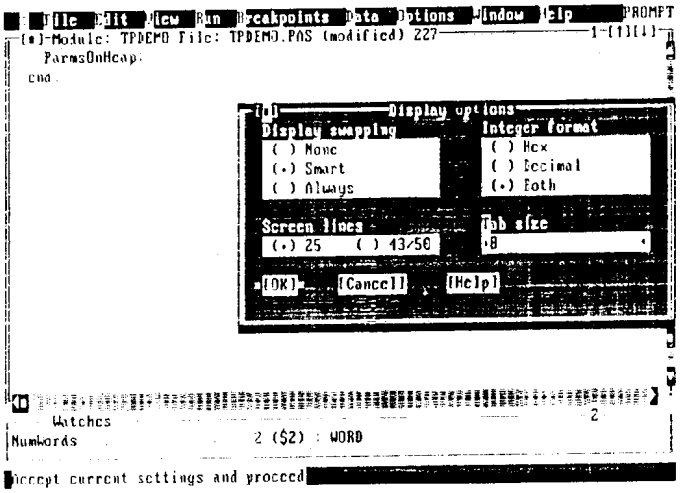


图 13.1　显示选择对话框

13.5.3.2　整数格式

整数格式(Integer Format)的无线电按钮允许用户从 3 种显示整数的格式中选择：

Decimal：　按普通的十进制数显式整数。

Hex：　按十六进制数显示整数，格式与当前语言相适应。

Both：　按十进制数显示，在十进制值后面的圆括弧内再按十六进制数显示一次。

13.5.3.3　屏幕行数

屏幕行数(Screen Lines)无线电按钮用来确定 Turbo Debugger 屏幕是使用通常的 25 行

显示，还是使用 EGA 与 VGA 显示适配器(adapter)支持的 43 行或者 50 行显示。

13.5.3.4　制表键大小

制表键大小(Tab Size)输入框允许你设置每个制表位置所占用的列数。你可以减少制表的列宽度，对于那些有大量用制表键缩进的代码的源文件来说，这样做可以使你看到更多的正文。制表的列宽度可设置在 1 到 32 之间。

13.5.4　源命令路径

设置 Turbo Debugger 搜索源文件的目录。请参阅《Turbo Debugger 用户手册》第八章关于模块窗口的讨论。

13.5.5　保存选择项命令

该命令打开一对话框，从中可以把当前的选择项保存到一个磁盘上的配置文件中去。这些选择项是：

- 击键宏。
- 当前窗口的布局和窗口区的格式。
- 所有在 Options(选项)菜单中的设置。

Turbo Debugger 允许用户以所有这些方式或其中的任一方式保存选择项，这取决于用户选择了保存配置(Save configuration)复选框(见图 13.2)(check box)中的哪一个：

Options：　保存 OPtions(选项)菜单中的所有设置。

Layout：　只保存开窗口的布局(Windowing layout)。

Macros：　只保存当前定义的宏。

用户也可以使用保存到(Save To)输入框改变保存选择项的配置文件的名称。

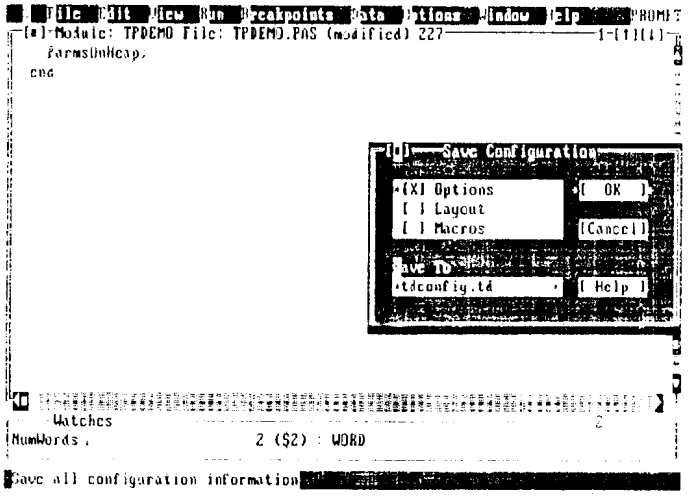


图 13.2　保存选择项对话框

13.5.6 恢复选择项命令

从磁盘文件中恢复用户的选择项。用户可以拥有多个配置文件，包含不同的宏、窗口布局等等，注意必须选择一个由 Save Options 命令或者 TDINST 程序产生的配置文件。

13.6 在 Turbo Debugger 中运行 DOS

在调试程序时，有时需要利用另一个程序或者实用工具。通过 File Dos Shell 命令可以做到这一点。

当用户启动 DOS 命令处理器时，如果有必要的话，正在被调试的程序将被转换到磁盘中。这样就可以让用户执行 DOS 命令。即使正在被调试的程序占用了所有可用内存也一样。当然，这意味着程序在内存和磁盘之间转换时要有几秒钟的延迟。

注意：不要在 DOS Shell 状态下把 TSR(终止并驻留程序)加载到 Turbo Debugger 的上方。

在完成 DOS 命令后，输入 EXIT 和 Enter 键可以返回到调试过程中。

13.7 返回 DOS

用户可以在任何时候按 Alt-X 键结束调试过程并返回 DOS，除非有一对话框处于活动状态(在这种情况下，应先按 Esc 键关闭对话框)。用户也可以选择 File/Quit 来达到同样的目的。

所有最初分配给正在被调试的程序的内存都将被释放掉。如果用户正在调试的程序通过 DOS 内存块分配例程获得了内存，这些内存也将被释放掉。

第六部分

混合编程的参考资料

TASM 命令行参考

混合编程实用程序

附录 A

TASM 命令行参考

本附录主要是让用户熟悉 Turbo Assembler 的命令行可选项。它介绍每个命令行可选项，用户可使用这些可选项改变汇编器的汇编方式；然后介绍怎样使用以及何时使用命令文件；最后讲述配置文件。

A.1 在 DOS 中启动 Turbo Assembler

Turbo Assembler 的命令行语法功能极强，而且十分灵活。如果启动 Tubro Assembler 时不给出任何参数，即打入

TASM

用户会看到全屏幕的提示信息，这些信息描述了许多命令行选择项和用以定义要汇编的文件的语法。图 A.1 列出了用户将看到的内容。

Syntax: TASM	[options] source [,object] [,listing] [,xref]
/a,/s	Alphabetic or Source—code segment ordering
/c	Generate cross—reference in listing
/dSYM[=VAL]	Define symbol SYM = 0, or = value VAL
/e,/r	Emulated or Real floating—point instructions
/h,/?	Display this help screen
/iPATH	Search PATH for include files
/jCMD	Jam in an assembler directive CMD (eg. /jIDEAL)
/kh#,/ks#	Hash table capacity #, String space capacity #
/l,/la	Generate listing: l=normal listing, la=expanded listing
/ml,/mx,/mu	Case sensitivity on symbols: ml=all, mx=globals, mu=none
/mv#	Set maximum valid length for symbols
/m#	Allow # multiple passes to resolve forward references
/n	Suppress symbol tables in listing
/o,/op	Generate overlay object code, Phar Lap—style 32—bit fixups
/p	Check for code segment overrides in protected mode
/q	Suppress OBJ records not needed for linking
/t	Suppress messages if successful assembly
/w0,/w1,/w2	Set warning level: w0=none, w1=w2=warnings on
/w—xxx,/w+xxx	Disable (—) or enable (+) warning xxx
/x	Include false conditionals in listing
/z	Display source line with error message
/zi,/zd	Debug info: zi=full, zd=line numbers only

图 A.1 Turbo Assembler 命令行

使用命令行选择项，用户可以定义一或多个要汇编的文件的名字以及用以控制汇编方

式的任何选择项。

命令行通常具有如下形式：

TASM fileset [;Fileset]...左中括号([)后的分号(;)将文件组隔开，以让用户使用同一命令行汇编多组文件。只要用户愿意，可为每组文件设置不同的选择项；例如，

```
TASM  /eFILE1;/aFILE2
```

按命令行参数/e 汇编 FILE1.ASM 文件，按参数/a 汇编 FILE2.ASM 文件。

按命令行的通常形式，fileset 可以是：

```
[option]... sourcefile [[+] sourcefile]...
          [,[objectfile][,[listfile],[,[xreffile]]]]
```

该语法表明，一组文件可以由希望作用于这些文件的任何选择项开始，后跟要汇编的文件。文件名可以是单个的文件名，也可以使用合法的 DOS 替代符 * 和？规定要汇编的多个文件。如果给出的文件名后不带扩展名，Turbo Assembler 自动以.ASM 为其扩展名。例如，要汇编当前子目录下所有的.ASM 文件，可打入

```
TASM *
```

希望汇编多个文件时，可用加号(+)将文件名隔开：

```
TASM MYFILE1+MYFILE2
```

可以在要汇编的文件名后跟一个可选择的目标文件名、列表文件名和交叉引用文件名。如果不指定目标文件或列表文件，Turbo Assembler 会创建一个与源文件同名的目标文件，并使之具有扩展名.OBJ。

如果不显式地要求列表文件，Turbo Assembler 就不产生它。若希望产生列表文件，可在目标文件名后置一个逗号，再接列表文件名。不显式提供列表文件名时，Turbo Assembler 创建一个与源文件同名的列表文件，且使之具有扩展名.LST。如果用户提供的列表文件后没有扩展名，Turbo Assembler 会自动以.LST 为其扩展名。

如果不显式地要求交叉引用文件，Turbo Assembler 就不产生它。若希望产生交叉引用文件，可在列表文件名后置一逗号“,”，再接交叉引用文件名。不显式提供交叉引用文件名时，Turbo Assembler 创建一个与源文件同名的交叉引用文件，且使之具有扩展名.XRF。如果用户提供的交叉引用文件名后没有扩展名，Turbo Assembler 会自动以.XRF 为其扩展名。可参看《Turbo Assembler 参考手册》的附录 D，其中讲述了如何使用全局交叉引用工具(TCREF)处理交叉引用文件。

如果用户希望使用隐含目标文件名，同时也要求产生列表文件，那么可用逗号分开目标文件名和列表文件名：

```
TASM FILE1,,TEST
```

这样 FILE1.ASM 被汇编成 FILE.OBJ，同时创建列表文件 TEST.LST。

如果希望使用隐含目标文件名和列表文件名，同时也要求产生交叉引用文件，可在这些文件名间置以逗号：

```
TASM MYFILE ,,, MYXREF
```

这样 MYFILE.ASM 被汇编成 MYFILE.OBJ，同时创建列表文件 MYFILE.LST 和交叉引用文件 MYXREF.XRF。

如果在规定要汇编的文件的名字时使用了替代符，那么在指出目标文件名和列表文件

名时亦可使用替代符。例如，假定当前子目录中包含了文件 XX1. ASM 和 XX2. ASM，命令行

TASM xx * ,YY * 汇编所有以 xx 开头的文件，产生以 YY 开关的目标文件，同时抽取原文件名中的一部分形成目标文件名的剩余部分。所以产生的目标文件为 YY1. OBJ 和 YY2. OBJ。

如果不希望产生目标文件而希望产生列表文件，或者希望产生交叉引用文件而不希望产生列表文件或目标文件，那么可指定文件名为空设备(NUL)。例如，

TASM FILE1,,NUL,

将文件 FILE1. ASM 汇编成目标文件 FILE1. OBJ，不产生列表文件，但创建了一个交叉引用文件 FILE1. XRF。

A. 2 命令行选择项

命令行选择项可让用户控制汇编器的行为以及控制如何将信息输出到屏幕、列表文件和目标文件。Turbo Assembler 也提供这样一些可选项，它们不产生任何动作，但使 Turbo Assembler 与当前版本或更低版本的 MASM 兼容：

/b 设置缓冲区大小

/V 显示附加静态量

用户可用大小写字母的任意组合输入选择项。如果不使用多个/i 或/j 选择项，选择项的次序也可以是任意的；这些选择项被依次处理。使用/d 选择项时，在使用后续的/d 可选项前务必仔细定义符号。

注意：在源代码中使用冲突指令可以清除相应命令行选择项。

图 A. 1 汇总了 Turbo Assembler 的命令行选择项；以下对每个选择项进行详细介绍。

/a

功能 指明按字母顺序进行段排序

语法 /a

解释 /a 选择项告诉 Turbo Assembler 在目标文件中按字母顺序存放段。在源文件中使用. ALPHA 伪指令可完成同样的功能。需要汇编为早期版本的 IBM 或 Microsoft 汇编器书写的源文件时，常必须使用该选择项。

/s 选择项与该选择项的效果互逆，它将目标文件恢复成隐含的顺序段排序。如果在源文件中使用. SEQ 伪指令指明顺序段排序，它将忽略命令行上的任何/a 选择项。

举例 TASM /a TEST1

该命令行产生目标文件 TEST1. OBJ，其中的段按字母顺序排序。

/b

语法　/b

解释　为处理兼容性而设置此选择项。它不引起任何附加动作，对汇编过程也不产生影响。

/c

功能　启动列表文件中的交叉引用

语法　/c

解释　/c 选择项启动列表文件中的交叉引用信息。Turbo Assembler 将交叉引用信息加到列表文件尾部的符号表中。意思是，为了查看交叉引用信息，用户可在命令行上显式地指明一个列表文件，也可使用/l 选择项启动列表文件。对每一符号，交叉引用显示出定义该符号的行以及涉及到该符号的所有行。

举例　TASM /l /C TEST1

该命令创建一个列表文件，符号表中包含了交叉引用信息。

/d

功能　定义一个符号

语法　/dsymbol [=value or expression]

解释　/d 选择项为源文件定义一个符号，就象该符号是在源文件第一行中用＝伪指令定义的一样。只要需要，可在命令行上多次使用该选择项。可将某符号定义成与另一符号相等，也可定义成某一常数值。在等号(二)右边不可使用带有算符的表达式。例如可使用/dx＝9 和/dx＝Y，但不允许使用/dx＝Y－4。

举例　TASM /dMAX＝10 /dMIN＝2 TEST1

该命令行定义两个符号，MAX 和 MIN，源文件 TEST1. ASM 的其它语句中可使用这两个符号。

/e

功能　产生浮点仿真器指令

语法　/e

解释　/e 选择项让 Turbo Assembler 产生浮点数指令，这类指令由软件浮点数仿真器执行。如果程序中包括了模拟 80×87 数字协处理器功能的浮点数仿真库时才可用此选择项。通常，如果汇编模块只是用使用了浮点数仿真库的高级语言书写的程序的一部分要用此选择项(Turbo C、Turbo Pascal、Turbo Basic 和 Turbo Prolog 都支持浮点数仿真)。因为仿真库需要编译器的初启代码进行初始化，所以不

能只将汇编程序与仿真库相链接。通过启动实际的浮点数指令，/r 选项可与该选择项的功能互逆。实际的浮点数指令只可由数字协处理器执行。如果在源文件中使用 NOEMUL 伪指令，它将忽略命令行中的/e 选择项。

/e 命令行选择项与源文件首部使用的 EMUL 伪指令功能相同，也与/jEMUL 命令行选项功能相同。

举例 TASM /e SECANT

TCC −f TRIG.C SECANT.OBJ

第一个命令行汇编一个带有浮点数仿真指令的模块，第二个命令行编译一个带有浮点数仿真的 C 语言模块，然后将它与汇编器产生的目标模块链接。

/h 或/?

功能 显示一屏求助信息

语法 /h 或/?

解释 /h 选择项指示 Turbo Assembler 显示一屏求助信息以描述命令行语法。求助信息包括选择项的列表以及用户可输入的各种文件名。/? 选择项与/h 选择项功能相同。

举例 TASM /h

/i

功能 设置 Include 文件的路径

语法 /iPATH

解释 在源文件中使用了 INCLUDE 伪指令时，用户可使用/i 选择项告知 Turbo Assmbler 到何处寻找被包括的文件。可在命令行中放置多个/i 选择项(/i 出现的次数仅受 RAM 的限制)。当 Turbo Assembler 碰到 INCLUDE 伪指令时，它搜寻文件的范围决定于 INCLUDE 伪指令中的文件名，INCLUDE 后的文件名可以含有目录路径，也可以仅是一个简单的文件名。如果目录路径是文件名的一部分，那么 Turbo Assembler 将尝试着搜索此路径，然后按顺序搜索由/i 命令行选择项规定的目录，最后搜索在配置文件中由/i 选择项规定的任何目标。如果 INCLUDE 后的文件名中不包含目录路径，那么 Turbo Assembler 将首先搜索/i 命令行选择项指定的目录，然后搜索配置文件中由/i 选择指定的目录，最后搜索当前目录。

举例 TASM /i\INCLUDE /iD:\ INCLUDE TEST1

如果源文件中存在下列语句

```
INCLUDE MYMACS.INC
```

Turbo Assembler 将首先查找\INCLUDE\MYMACS.INC，然后查找 D:\INCLUDE\MYMACS.INC。如果仍未找到此文件，它将在当前目录中查找 MYMACS.INC。

如果源文件中的语句是

INCLUDE INCS\MYMACS. INC

Turbo Assembler 将首先查找 INCS\MYMACS. INC，然后查找\INCLUDE\MYMACS. INC，最后查找 D：\INCLUDE\MYMACS. INC。

/j

功能 定义 汇编器的初启伪指令

语法 /jdirective

解释 /j 选择项可让用户定义一条伪指令，该伪指令在第一行源文件之前被汇编。directive 可以是任何不带参数的 Turbo Assembler 伪指令，如. 286、IDEAL、%MACS、NOJUMPS 等等。《Turbo Assembler 参考手册》第三章详细描述了所有的 Turbo Assembler 伪指令。可在命令行中放入多个/j 选择项，它们将按从左至右的顺序被处理。

举例 TASM /j. 286 /jIDEAL TEST1

该代码使用启动的 80286 指令和启动的 Ideal 模式表达分析汇编文件 TEST1. ASM。

/kh

功能 设置允许的符号的最大数目

语法 /khnsymbols

解释 /kh 选择项设置用户程序可包含的符号的最大数目。如果不使用该选择项，那么用户程序仅可包含 8 192 个符号；使用该可选项可增加符号的数目，即增加 nsymbols 的值，使之可高达 32 768 个符号。在汇编程序时，如果看到 Out of hash space 信息，则可使用该选择项。也可使用该选择项将可用符号的总数降到低于隐含值 8 192。在试图汇编一个程序而又没有足够的内存空间时，这种设置有助于释放内存。

举例 TASM /kh10000 BIGFILE

此命令告知 Turbo Assembler 在汇编 BIGFILE 时保留 10 000 个符号的内存空间。

/ks

功能 设置 Turbo Assembler 串空间的最大值。

语法 /ksbytes

解释 通常串空间的大小是自动确定的，而且不必调整。但如果在汇编源文件时产生 Out of string space 信息，那么可能就要使用此选择项增加串空间。可以从数值 100 开始设置，一直增加到汇编源文件时不出错为止。bytes 的最大允许值为

255。

举例 TASM /KS150 SFILE

告知 Turbo Assembler 保留 150K 的串空间。

/l

功能 产生列表文件

语法 /l

解释 /l 可选项表明用户希望产生列表文件，即便用户不在命令行上显式地指明列表文件，Turbo Assembler 也将产生列表文件。列表文件与源文件同名，其扩展名为. LST。

举例 TASM /lTEST1

该命令行产生一个名为 TEST1. LST 的列表文件。

/la

功能 显示列表文件中的高级接口代码

语法 /la

解释 /la 选择项告知 Turbo Assembler 显示列表文件中产生的所有代码，包括高级语言接口. MODEL 伪指令产生的代码。

/m1

功能 大小写符号分别对待

语法 /m1

解释 /m1 可选项告知 Turbo Assembler 按大小写分别对待所有的符号名。通常情况下，大小写字母被认为是等价的，所以符号名 ABCxyz. abcxyz 和 ABCXYZ 和 ABCXYZ 指同一个符号。如果指定/m1 选项，这三个符号将被区别对待。既使选用了/m1 可选项，用户也仍可以按大小写输入关键字。关键字指汇编器固有的有特殊含义的符号，例如指令、伪指令和操作符等。

举例 TASM /m1 TEST1

这里，TEST1. ASM 中含有下列语句：

```
abc   DW   0
ABC   DW   1          ;not a duplicate symbol
mov   AX, [Bp]   ;mixed case OK in keywords
```

/mu

功能 将符号转换成大写形式

语法 /mu

解释 /mu 可选项告知 Turbo Assembler 忽略符号的大小写方式。作为缺省情况，如果用户不使用/m1 伪指令时，Turbo Assembler 规定符号中的任何小写字母均将被转换成大写形式。

举例 TASM /MU TEST1

确认所有的符号名都将被转换成大写(隐含)：

```
EXTRN myfunc   ; NEAR
call myfunc    ; don't know if declared as
                 ; MYFUNC, Myfunc,...
```

/mx

功能 按大小写分别对待公共外部符号

语法 /mx

解释 /mx 可选项告知 Turbo Assembler 仅区别对待公共外部符号。源文件中所用的其它所有符号均被当成大写形式处理。当调用编译过或汇编过的其它模块——如由 Turbo C 编译过的模块——中的子程序时，用户应用此选择项以保持大小写分别对待。

举例 TASM /mx TEST1；

这里，TEST1.ASM 中包含如下源语句行：

```
EXTRN   Cfunc : NEAR
myproc  PROC  NEAR
call    Cfunc
.
.
.
```

/n

功能 压缩列表文件中的符号表

语法 /n

解释 /n 可选项表明用户不希望列表文件尾部的常规符号表。一般情况下，一个完整的符号表被列在列表文件的尾部，其中包含所有符号、符号类型、各符号的值。可以在命令行上显式地指明列表文件，或使用/1 可选项要求生成列表文件；否则，/n 将无效。

/p

功能 检查保护模式下的掺杂代码

语法 /p

解释 /p 可选项指明用户希望得到对在保护模式下会产生“掺杂”代码的任何指令的警告信息。在保护模式下，使用 CS：清除手段将数据移入内存的指令就是“掺杂”指令，因为除非采取特殊措施，否则这类指令可能会引起错误。如果用户程序运行于 80286 或 80386 的保护模式下，那么用此可选项即可。

举例 TASM /p TEST1

这里，TEST1.ASM 中含有以下语句：

```
.286p
code    segment
temp    dw    ?
mov     cs:temp,0 ;impure in protected mode
```

/r

功能 产生实际的浮点数指令

语法 /r

解释 /r 可选项告知 Turbo Assembler 产生实际的浮点数指令(而不产生模拟的浮点数指令)。如果用户程序将在装有 80×87 数字协处理器的机器上运行时可用此选择项。

/e 可选项的功能与/r 相反，它产生的是模拟浮点数指令。如果源文件中使用了 EMUL 伪指令，那么它将取代命令行中的/r 可选项。命令行可选项/r 与在源文件的首部使用 NOEMUL 伪指令的功能相同，也与命令行可选项/jNOEMUL 的功能相同。

举例 TASM /r SECANT

TPC /SN+ /SE− TRIG.PAS

第一个命令行汇编一个带有实际浮点数指令的模块。第二个命令行编译一个带有实际浮数指令且要将其目标文件与汇编语言链接的源 pascal 程序代码。

/s

功能 指明段的顺序排列

语法 /s

解释 /s 选择项指示 Turbo Assembler 在目标文件中按其在源文件中所触及的段的顺序进行段的存放。缺省情况下，如果用户不在配置文件中使用/a 选择项予以改变，那么 Turbo Assembler 将对段进行排序。如果源文件中使用了.ALPHA 伪指令指明按字母顺序对段进行排序，它将忽略命令行中的/s 选择项。

举例 TASM /S TEST1

该行命令创建一个目标文件(TEST1.OBJ)，其中段的排列次序与其在源文件中出现的次序一致。

/t

功能　压缩成功汇编的信息量

语法　/t

解释　如果汇编过程中不出现警告或错误信息，/t 选择项将抑制 Turbo Assembler 的任何信息显示。如果要汇编许多模块，而且只希望警告或错误信息显示在屏幕上时，则可使用该选择项。

举例　TASM /t TEST1

/v

语法　/v

解释　只用于兼容性处理。无任何动作，也不对汇编过程产生影响。

/w

功能　控制警告信息的产生

语法　/w

w－[warnclass]

w＋[warnclass]

解释　/w 可选项用于控制 Turbo Assembler 激发哪些警告信息。如果仅选用/w，“软”警告将被启动。软警告只表明用户有待改进代码某些方面的效率。

如果选用不带 warnclass 的/w－选择项，所有的警告均被抑制。选用带 Warhclass 的/w－选择项时，仅指定的警告被抑制。每种警告信息都有一个由 3 个字母组成的标识符：

ALN　段对齐
ASS　假设段是十六位
BRK　缺少方括号
ICG　产生低效的代码
LCO　定位寄存溢出
OPI　IF 条件指令不配对
OPP　过程不完整
OPS　段不完整
OVF　算术溢出
PDC　与编译次序有关的结构
PQK　假设常量警告
PRO　在保护状态下写入内存须指定 CS
RES　关键字警告

TPI　Turbo Pascal 非法警告

如果选用不带 warnclass 的/w+选择项，所有的警告均被启动。选用带 warnclass 的/w+选择项时，仅指定的警告被启动。缺省情况下，除“无效代码产生”(ICG)和“保护模式下写入内存”(PRO)警告之外，Turbo Assembler 在汇编用户文件时将启动所有的警告。在源文件中可使用 WARN 和 NOWARN 伪指令来控制是否启动对某个范围内源程序行的特殊警告。《Turbo Assembler 参考手册》第三章详细介绍了这些伪指令。

举例　TASM /W TEST1

TEST1. ASM 中的下列语句将启动没有/w 可选项时可能不会出现的警告信息：

```
mov bx, ABC    ;inefficient code generation warning
ABC=1
```

如果 TEST2. ASM 中包含语句：

```
dw 1000h * 20h
```

那么命令行

```
TASM /W-OVF TEST2
```

将不会产生警告信息。

/x

功能　列表文件中列出错误的条件句

语法　/x

解释　如果条件句 IF、IFNDEF、IFDEF 等引发错误，/x 选择项可将条件块中的语句归入列表文件。该选择项也使条件伪指令本身被归入列表文件；而一般情况下它们是不被列入列表文件中的。可在命令行上指定列表文件，或通过/1 选择项生成列表文件，否则/x 无效。用户可使用. LFCOND、. SFCOND 和. TFCOND 伪指令抑制/x 可选项的功能。

举例　TASM /X TEST1

/z

功能　选择引发错误信息的源文件行

语法　/z

解释　/z 选择项指示 Turbo Assembler 在产生错误信息时，显示与之对应的源文件行。引发错误的行出现在错误信息的前面。不用此选择项时，TurboAssembler 只显示信息描述所出现的错误。

举例　TASM /Z TEST1

/zd

功能　启动目标文件中的行号信息

语法　/zd

解释　/zd 指示 Turbo Assembler 设置目标文件中的行号信息。这有助于让 Borland 公司的 Turbo Debugger 工具显示源代码中的当前位置，但并不将行号信息放入允许 debugger 访问数据项的目标文件中。如果使用 Turbo Debugger 调试程序时内存空间不够，可用/zd 选定其中一些模块而用/zi 选定其它模块。

举例　TASM /zd TEST1

/zi

功能　启动目标文件中的 debug 信息

语法　/zi

解释　/zi 选择项指示 Turbo Assembler 将全部调试信息输出到目标文件中。调试信息包括与源代码显示同步的行号记录及有助于用户检测和修改程序数据的数据类型信息。/zi 选择项可让用户使用 Turbo Debugger 的全部特征遍历程序并检测或更改数据项。可将/zi 用于全部程序模块，也可只用于用户有兴趣调试的那些模块。由于/zi 开关将信息加入目标文件和可执行程序中，所以当在 Turbo debugger 环境中运行程序而内存不够时，用户可能就不希望用此开关。

举例　TASM /zi TEST1

间接命令文件

在命令行任意位置处，Turbo Assembler 都允许用户指定一个 indirect(间接)文件，间接文件的前面冠以“@”标志。例如：

TASM /dTESTMODE @MYPROJ. TA

将使文件 MYPROJ. TA 的内容成为命令行的组成部分，就如同直接在同一位置打入文件的内容一样。

这一有用的特征可让用户将极常用的命令行和文件列表放在一个独立的文件中。由于用户可以在命令行上使用多个间接文件，也可将命令文件与合法参数混合使用，所以没有必要把整个命令行都放入同一个间接文件中；例如，

TASM @MYFILES @IOLIB /dBUF=1024

用此方法，用户能够把一长串的标准文件和选择项保留在文件中，以便快速方便地改变单独的汇编命令的行为。

可将某一行中的所有文件及所有可选项都保留在命令文件中，也可将它们分解成若干行。

配置文件

Turbo Assembler 也允许用户将极常用的选择项放置在当前目录下的配置文件中。在 DOS 3. X 或更高版本的 DOS 环境下工作时，它也检查 TASM 从中装入的子目录。这种情况下，当运行 Turbo Assembler 时，它将在当前目录中查找 TASM. CFG 文件。如果 Turbo Assembler 找到该文件，它将此文件作为一个间接文件对待，而且在处理命令行中所有其它内容之前首先处理此文件。

当所有基于同一目的的源文件都在同一个目录中时，这种特征极为有用。例如，用户常希望以模拟浮点数指令(/e 可选项)进行汇编，可将该可选项放入 TASM. CFG 文件中，这样就不必在每次启动 Turbo Assembler 时都指定此选择项。

配置文件的内容与间接文件的格式完全相同。该文件可以包含任何有效的命令行选择项，也可包含用户所希望的任意行数。其中的可选择项与出现在命令行上的可选项同样对待。

配置文件的内容在命令行中所有参数之前处理。只通过在命令行中放置相逆功能的可选项，用户就可以清除配置文件中选择项的效果。例如，如果配置文件包含有

```
/a/e
```

而用户以

```
TASM /r/s MYFILE
```

激活 Turbo Assembler，这里 MYFILE 是用户文件。用户文件将被以顺序段排序(/s)和实际浮点数指令(/r)进行汇编，尽管配置文件中含有/a 和/e 选择项，以指定使用按字母顺序进行段排序指令和模拟浮点数指令。

附录 B

混合编程实用程序

Borland 提供了五个功能强大的、独立的实用程序。可对 Turbo 系列语言文件使用这些独立的实用程序，也可以对其它模块来使用它们。

这些对 Turbo 系列语言特别有用的辅助实用程序是：

- MAKE(包括 TOUCH 实用程序；独立的程序管理员)
- TLINK(Turbo 连接器)
- TLIB(Turbo 库管理员)
- GREP(一种文件查找实用程序)
- OBJXREF(一种目标模块交叉引用实用程序)
- TCREF(一种交叉引用实用程序)

本附录将解释每个实用程序是干什么的，并以代码和命令行为例说明如何使用它们。

B.1 独立的 MAKE 实用程序

Turbo 系列语言具有强大的功能和巨大的灵活性。可用它来管理由许多源文件和目标文件构成的大型、复杂程序。但遗憾的是它要求记住哪些文件需要用来产生别的文件，这是因为如果某一文件做了修改，就得进行必要的重编译和连接。当然，一种简单的解决方法就是每当你做了改动时，就把所有模块重新编译一次。但当程序非常庞大时，这将花费许多时间，那么该怎么办呢？

答案很简单：使用 MAKE。Borland 的 MAKE 是一个智能的程序管理员：只要给出合适的指令，它就能做必要的工作使程序更新。而事实上，MAKE 所能做的远远不止这些。它可用来备份文件、从不同子目录中提取文件，甚至当程序使用的数据文件被修改后，它能自动运行程序，随着使用 MAKE 次数的增加，用户就会发现它以各种新的、不同的方式来帮助管理程序开发。

在本节中，将描述如何与 Turbo 系列语言和 TLINK 一起来使用独立的 MAKE。

B.1.1 一个快速示例

让我们用一个例子来开始说明 MAKE 的功能。假设有一显示银河系信息的程序 GETSTARS，它读入一有关银河系信息的文本文件，进行一些处理后，产生一具有结果信息的二进制文件。

GETSTARS 使用了一些定义，存放在 STARDEFS.INC 中；一些例程，存放在 STAR-

LIB. ASM 中(在 STARLIB. INC 中声明)。此外,GETSTARS 本身由三个文件组成:GSPARSE. ASM、GSCOMP. ASM、GETSTARS. ASM。头两个文件 GSPARSE 和 GSCOMP 拥有相应的包含文件(GSPARSE. INC 和 GSCOMP. INC)。第三个文件 GETSTARS 包含了程序的主体。在三个文件中,只有 GSCOMP. ASM 和 GETSTARS. ASM 使用了 STARLIB 中的例程。

下面是每个汇编文件所需要的包含文件:

.ASM 文件	包含文件
STARLIB. ASM	无
GSPARSE. ASM	STARDEFS. INC
GSCOMP. ASM	STRDEFS. INC,STARLIB. INC
GETSTARS. ASM	STARDEFS. INC,STARLIB. INC GSPARSE. INC,GSCOMP. INC

为产生 GETSTARS. EXE(假设为中型数据模式),需打入如下命令行:

```
tasm  /t  /ml  /s  starlib
tasm  /t  /ml  /s  gsparse
tasm  /t  /ml  /s  gscomp
tasm  /t  /ml  /s  getstars
tlink  Starlib gsparse gscomp getstars,getstars,
  getstars,lib\math   lib\io
```

回顾前面的信息,可以得到文件间的一些依赖关系:

- GSPARSE,GSCOMP 和 GETSTARS 均依赖于 STARDEFS. INC,即如果 STARDEFS. INC 做了修改,就必须重新编译这三个文件。
- 同样,对 STARLIB. INC 的任何修改都需要对 GSCOMP 和 GETSTARS 的新编译。
- 对 GSPARSE. INC 的修改意味着 GETSTARS 要重编译,对 GSCOMP. INC 也是一样。
- 当然,对任何源代码文件(STARLIB. ASM、GSPARSE. ASM)的修改意味着对新文件要重新编译。
- 最后,如果进行了重编译,就必须进行重连接。

这样要做许多跟踪。如果你修改了 STRLIB. INC,只重编译了 GETSTARS. ASM 而忘了重编译 GSCOMP. ASM,又会发生什么情况呢?这会导致连接出来的程序运行出问题。当然,可用一个. BAT 文件进行四次重编译和一次连接。但每修改一处,就必须再这样做一遍。我们来看看 MAKE 是如何简化这些工作的。

B. 1. 1. 1 创建一个 make 文件

一个 make 文件由两组信息组成:依赖关系和需满足的命令

例如,把已给的依赖关系和需满足的命令组合起来就可产生如下的 make 文件:

```
getstars exe:  getstars. osj  gscomp. obj  gsparse. obj  starlib. obj
  tlink  starlib  gsparse  gscomp  gestars,getstars,\
        getstars, lib\math  lib\io
```

```
getstars.obj:  getstars.asm  stardefs.inc  starlib.inc  \
               gscomp.inc  gsparse.inc
        tasm    /t  /ml  /s    getstars.asm
gecomp.obj  :  gscomp.asm  stardefs.inc starlib.inc
        tasm    /t  /ml  /s    gscomp.asm
gsparse.obj : gsparse.asm  stardefs.inc
        tasm    /t  /ml  /s    gsparse.asm
starlib.obj : starlib.asm
        tasm    /t  /ml  /s    starlib.asm
```

这正是前面刚刚阐述过的内容，只是次序有些颠倒。MAKE 按下述方式解释该文件：

- 文件 GETSTARS.EXE 依赖于四个文件：GETSTARS.OBJ、GSCOMP.OBJ、GSPARSE.OBJ 和 STARLIB.OBJ。如果这个文件中的任意一个有了变化，GETSTARS.EXE 就被更新。怎么办呢？只要使用所给的 TLINK 命令即可。
- 文件 GETSTARS.OBJ 依赖于五个文件：GETSTARS.ASM、STARDEFS.INC、STARLIB.INC、GSCOMPO.INC 和 GSPAARSE.INC。如果这五个文件中任意一个发生了变化，就用所给的 TASM 命令进行重编译。
- 文件 GSCOMP.OBJ 依赖于三个文件：GSCOMP.ASM，STARDEFS.INC 和 STARLIB.INC。如果其中任一个文件发生了变化，就必须用所给的 TASM 命令进行重新编译。
- 文件 GSPARSE.OBJ 依赖于两个文件：GSPARSE.ASM 和 STARDEFS.INC。如果其中之一发生变化，就必须用所给的 TASM 命令进行重新编译。
- 文件 STARLIB.OBJ 只依赖于一个文件：STARLIB.ASM。如果 STARLIB.ASM 发生了变化，就通过 TASM 重新编译。

把以上信息输入到一文件中，我们暂且称它为 MAKEFILE，现在我们就可以使用 MAKE.EXE 了。

B.1.1.2　使用一个 make 文件

假设我们按上面所述创建了 MAKEFILE 文件，当然还得假设所有的源代码文件都存在，现在就可打入命令：

```
MAKE
```

MAKE 将查找 MAKEFILE 文件（也可改为其它文件名，这将在以后讨论），并读入描述 GETSTARS.EXE 依赖关系的第一行，来检查 GETSTAR.EXE 是否存在且是否最新。

这就要对 GETSTARS.EXE 所依赖的每个文件（GETSTARS.OBJ、GSCOMP.OBJ、GSPARSE.OBJ 和 STARLIB.OBJ）做同样的检查；这些文件所依赖的其它文件也要依次被检查。

通过调用 TASM 命令，可更新各个 OBJ 文件，最后通过 TLINK（如果需要的话）产生 GETSTARS.EXE 的最新版本。

如果 GETSTARS.EXE 和所有.OBJ 文件都已存在，此时 MAKE 将每个.OBJ 文件的最后一次修改日期、时间和它所依赖的文件的日期、时间做比较，如果有一依赖文件比.OBJ 文件更新，则 MAKE 知道上次.OBJ 文件生成以来又已经有了变化，于是执行 TASM 命令。

如果 MAKE 更新了任一.OBJ 文件,则当它把 GETSTARS.EXE 的日期和时间与.OBJ 文件的日期和时间比较时就知道必须执行 TLINK 命令,以产生一最新版本的 GETSTARS.EXE 文件。

B.1.1.3 步 进

下面通过一步一步进行的例子来帮助你弄清上面的描述。假设 GETSTARS.EXE 和所有的.OBJ 都存在,而且 GETSTARS.EXE 比所有.OBJ 文件新,同样,每个.OBJ 文件也比它所依赖的各个文件新。此时,打入以下命令:

```
MAKE
```

将不会发生任何事,因为不需要更新任何文件。

现在,假设修改了 STARLIB.ASM 和 STARLIB.INC,就假定是改变了其中某些常量的值,当打入以下命令:

```
make
```

时,MAKE 看到 STARLIB.ASM 比 STARLIB.OBJ 新,于是执行命令:

```
tasm /t /ml /s starlib.asm
```

STARLIB.INC 也比 GSCOMP.OBJ 新,因此执行下条命令:

```
tasm /t /ml /s gscomp.asm
```

最后,由于执行了以上三条命令,文件 STARLIB.OBJ,GSCOMP.OBJ 和 GETSTARS.OBJ 都比 GETSTARS.EXE 新,因此 MAKE 执行最后一个命令:

```
tlink starlib gsparse gscomp getstars,getstars,getstars,
lib\math lib\io
```

这样,就把所有.OBJ 文件连接起来生成最新版本的 GETSTARS.EXE(注意,TLINK 命令行实际上应在同一行上)。

现在已对 MAKE 有了一些基本的认识:它是干什么的;如何创建一个 make 文件;MAKE 是如何解释该文件的。下面将详细讨论如何建立 Make 文件及如何使用 Make。

B.1.2 创建 make 文件

Make 文件含有 Make 更新程序所需的定义和关系。可创建多个 Make 文件,并按需要来命名。如果运行 Make 时,没有指定一个 make 文件,Make 就查找缺省名为 MAKEFILE 的文件。

所有规则定义,伪指令都以回车换行终结;如果一行过长(见上面例子中的 TLINK 命令),你可为该行最后字符输入反斜杠(\)来使之延续到下一行。

空白键k k 空格和制表符,可用来分开相邻的标识符(比如依赖关系),并用来缩进一条规则内的命令。

B.1.2.1 Make 文件的组成

创建 Made 文件就象写程序一样,有定义、命令和指令。下面就列出 make 文件所允许的结构:

- 注释
- 显式规则
- 隐含规则

- 宏定义
- 伪指令k k 包含文件，条件执行，错误检测，解除宏定义

下面详述各项

注　释

注释由＃号起始，该行＃号后面的字符被 Make 忽略。注释可放在任何位置，不必非得从某一特定列起始。

注释不能用反斜杠(\)来续行，而必须重新用＃号开始。这是因为，如果把它放在＃号前面，反斜杠(\)就不是该行最后字符了，而如果跟在＃号后面，它又是注释的一部分。

下面是 Make 文件中注释的例子：

```
#makefile  for  GETSTARS.EXE
#  does  complete  project  maintenance
getstars.exe:  getstars.obj  gscomp.obj  gsparse.obj  starlib.obj
#  Can't put  a  comment  at  the  end of the next line
tlink starlib gsparse gscomp  getstars,getstars,\
        getstars,lib\math lib\io
#legal comment
              #  can't put a comment between the next two lines
    getstars.obj : getstars.asm  stardefs.inc  starlib.inc
                   gscomp.inc  gsparse.inc
          tasm     /t  /ml  /s  getstars.asm
                   # you can put a comment here
```

显式规则

对显式规则已经熟悉了，因为在前面的 MAKE 文件例子中已用到了显式规则。显式规则采用如下形式：

```
目标  [目标...]  :  [源文件  源文件..]
      [命令]
      [命令]
      ...
```

其中目标是要更新的文件，源文件是目标所依赖的文件，命令是任何有效的 MS－DOS 的命令(包括调用.BAT 文件，执行.COM 和.EXE 文件)。

显式规则可定义一个或多个目标名，零个以上源文件名和一个可选的命令行。目标和源文件名可以含有驱动器和路径，但不能含有匹配符。

语法在这里显得很重要。目标必须从第 1 列开始，命令必须至少缩进一个字符。象前面提到的那样，如果源文件列表或给出的命令长于一行，可用反斜杠(\)来续行。最后，源文件和命令都是可选的，可以含有 目标 [目标…] 后跟冒号(:)的显式规则。

显式规则的作用在于使用所给的源文件，用列出的命令来创建或更新目标。如果 MAKE 遇到显式规则，先检查是否其中的源文件是 MAKE 文件中其它地方的目标文件。如果是，先执行那些规则。

根据别的显式规则，一旦所有的源文件被创建和更新后，MAKE 就检查是否存在目标。

如果不存在，就按给出的顺序调用命令。如果目标存在，就把它的最后一次修改时间与源文件的修改时间相比。如果源文件比目标修改得晚，就执行列出的命令。

对于给定的文件名，在一次 MAKE 执行中在显式规则的左边只能出现一次。

显式规则中的每个命令行必须以空白开始。MAKE 把显式规则后到第一列为非空白的行之间的行看作是命令表的一部分。忽略空白行。

不带命令行的显式规则与带命令行的显式规则有些差别：

- 如果显式规则是目标后带命令的，那么，目标只取决于列出的依赖文件
- 如果显式规则是不带命令的，那么目标不仅依赖于显式给出的文件，而且还依赖于与目标匹配的隐含规则的文件。

参见下一部分对隐含规则的讨论。

下面是显式规则的一些例子：

```
myprog.obj : myprog.asm
  tasm  /t  myprog.asm
prog2.obj : prog2.asm  include\stdio.inc
  tasm  /t  /ml  prog2.asm
prog.exe : myprog.asm prog2.asm include\stdio.inc
  tasm  /t  myprog.asm
  tasm  /t  /ml  prog2.asm
  tlink  myprog  prog2,prog, , lib\io
```

- 第一个显式规则说明 myprog.obj 依赖于 myprog.asm，而 myprog.obj 通过执行给定的 TASM 命令创建。
- 类似地，第二个显式规则说明 prog2.obj 依赖于 myprog.asm 和 stdio.inc（在 INCLUDE 子目录下），而 prog2.obj 是通过给定的 TASM 命令创建的
- 最后一个规则说明 prog.exe 依赖于 myprog.asm，prog2.asm 和 stdio.inc，且如果其中之一发生了变化，prog.exe 就用所给的命令系列来重建。但是，这可能要做不必要的工作。因为即使只有 myprog.asm 变化了，prog2.asm 仍要被重编译。这是由于，每当规则的目标过期了，就要执行该规则下的所有命令。
- 如果把显式规则

```
prog.exe : myprog.obj  prog2.obj
  tlink    myprog  prog2 ,prog , , lib\io
```

作为 MAKE 文件的头一规则，在此规则下面再给出别的规则（关于 myprog.obj 和 prog2.obj 的规则），那么，只有那些需要重编译的文件才被重新编译。

隐含规则

MAKE 也允许你定义隐含规则。隐含规则是显式规则的一般化。那么，这意味着什么？下面通过例子来说明两种类型规则间的关系。考虑一下前面样本程序中的显式规则：

```
starlib.obj : starlib.asm
  tasm  /t  /ml  /s  starlib.asm
```

该规则是条普遍性的规则，因为它遵循一个一般性原则：一个.obj 文件依于同名的.asm 文件且通过执行 tasm 来创建。事实上，在你的 make 文件中，你可能拥有多个（甚至几十

个)具有这种格式的显式规则。

把显式规则重定义为隐含规则,你就可以消除所有相同形式的显式规则了。作为一条隐含规则,它具有如下形式:

```
.asm.obj:
  tasm  /t  /ml  /s  $<
```

这条规则意味着"每个以.obj 结尾的文件依赖于以.asm 结尾的具有相同文件名的文件,且用.obj 文件是通过命令

```
tasm  /t  /ml  /s  $<
```

来创建的,在这里,$<代表文件名及其源(.asm)扩展名。"(符号$<是一特殊宏。将在下面的部分中讨论。)

隐含规则的语法形式是:

```
.源扩展名.目标扩展名
[命令]
[命令]
...
```

和前面一样,这里的命令是可选的且必须要缩进。

源扩展名(必须从第 1 列起始)是源文件的扩展名,即它适用于具有如下格式的任何文件:

```
文件名.源扩展名
```

同样,目标扩展名是指如下文件:

```
文件名.目标扩展名
```

在这里源文件和目标文件的文件名是相同的。换句话说,由该隐含规则代替的所有显式规则具有如下的格式:

```
文件名.目标扩展名  :文件名.源扩展名
{命令}
{命令}
...
```

这里,文件名是任意的。

如果对一给定目标没找到相应的显式规则,或该目标的显式规则不带命令,那么就可用隐含规则。

有问题的文件扩展名可用来确定使用哪条隐含规则。如果对所指定的源扩展名可找到一同名文件为目标,就可应用隐含规则。例如,假设你有一个 MAKE 文件(命名为 MAKEFILE),其内容为:

```
.asm.obj :
  tasm  /t  /ml  /s  $<
```

如果想把汇编程序 raito.asm 编译成 ratio.obj,可用命令:

```
make  ratio.obj
```

MAKE 把 ratio.obj 作为目标。由于没有创建 ratio.obj 的显式规则,MAKE 应用隐含规则,可产生命令:

```
tasm  /t  /ml  /s  ratio.asm
```

这当然就要做必要的编译来创建 ratio.obj 了。

如果给了一条不带命令的显式规则，也可用隐含规则。如上所述，假设在 MAKE 文件起始处有如下的隐含规则：

```
.asm.obj :
  tasm  /t  /ml  /s  $<
```

这时，可在后面将显式规则重写如下：

```
getstars.obj : stardefs.inc  starlib.inc  gscomp.inc  gsparse.inc
gscomp.obj : stardefs.inc  starlib.inc
gsparse.obj : stardefs.inc
```

由于没有显式信息来创建这些.obj 文件，MAKE 就应用前面定义过的隐含规则。由于 starlib.obj 只依赖于 starlib.asm，相应的规则就从本列表中消除。因为 MAKE 会利用隐含规则自动地实现它。

对相同的目标扩展名可写多个隐含规则，但在一特定时间只能应用其中一条。如果对一给定的目标扩展名存在多条隐含规则，就按规则在 MAKE 文件中的出现次序来检查每条规则，直到所有可应用的规则被检查完为止。

MAKE 使用发现源扩展名文件的第一条隐含规则。即使此规则的命令失败，也不再检查其它的隐含规则。在一隐含规则之后，到第一列为非空白的行之间的所有行都被看作是命令表的一部分。忽略空白行。命令行语法将在本章后面部分提供。

与显式规则不同，MAKE 并不知道隐含规则中的文件全名。因此，为 MAKE 提供了特殊的宏，它允许你来包含此规则所创建的文件名（见本小节中关于宏定义的详细讨论）。

下面是隐含规则的一些例子：

```
.c.obj :
 tcc  -c  $<
.asm.obj :
 tasm  /ml  $*
```

在第一条隐含规则中，目标文件是.obj 而源文件为.c 文件。该例在命令表中只有一行；有关命令的语法在后面讲述。

在第二个例子中，MAKE 用带/ml 选择项的 tasm 来汇编所给的以.asm 结尾的源文件。

命令表

我们已讲述了显式和隐含规则及它们如何使用命令表。下面我们就来讲一下这些命令和选择项。

在命令表中的命令必须缩进，即前面至少有一空格或制表等，且采取如下形式：

```
[前缀…]  命令体
```

命令表中每一命令行由一些可选的前缀及单个的命令体组成。

在一命令中允许使用的前缀用来改变 MAKE 对这些命令的处理。前缀或为一 at 符号(@)，或为紧跟着一个数的连字号(-)。

@ 强制 MAKE 在执行一命令前不显示该命令。即使在 MAKE 命令行上不给-S 选择项，显示也是隐藏的。本前缀只有在命令出现时才起作用。

-数 影响 MAKE 如何处理退出代码。如果提供一个数，MAKE 就只在退出

状态超过所给定的数时才退出处理。在下面例中，MAKE 只在退出状态超过 4 时才退出：

```
-4  myprog  sample.x
```

如果不提供“数”前缀，MAKE 就检查命令的退出状态。如果状态为非零，MAKE 将中止并删除当前的目标文件。

- 只有连字号不带数字，MAKE 就不检查退出状态。无论退出状态为何，MAKE 都将继续执行下去。

命令体的处理就像打入一行给 command.com 一样，但重定向和管道不再被支持。

MAKE 可执行如下的内部命令，办法是通过调用 command.com 的一个备份来执行这些命令：

BREAK	CD	CHDIR	CLS	COPY	CTTY	DATE
DEL	DIR	ERASE	MD	MKDIR	PATH	PROMPT
REN	RENAME	SET	TIME	TYPE	VER	VERIFY
VOL						

MAKE 对任何其它命令名采用 MS-DOS 查找算法来查找：

- 首先查找当前目录，然后查找路径中的各个目录。
- 在每个目录中，首先检查带有.COM 扩展名的文件，然后查.EXE，最后查.BAT。
- 如果找到一个.BAT 文件，就调用 COMMAND.OCM 的备份来执行该 BAT 文件。

很显然，在命令行中提供一扩展名，MAKE 就查找相应的扩展名。

下面是一些例子：

本命令将引起 COMMAND.COM 来执行。

```
cd  c:\include
```

本命令将用查找算法来查找：

```
tlink  x  y,z,z,lib\io
```

本命令只查找.com 扩展名：　　myprog.com　geo.xyz

本命令使用所提供的明确的文件名来执行：

```
c:\myprogs\fil.exe  -r
```

宏

有时，某些命令、文件名或选择项在你的 MAKE 文件中反复出现。在本附录起始部分的例子中，所有 TASM 命令都在当前目录下查找源文件。假设想控制一下源文件输入和结果输出的子目录，或者每当修改时就修改 MAKE 文件中的每一行，或者定义一个宏来使处理半自动化。

宏是代表一些字符串的名字。宏定义由宏名的扩展文本组成；这以后，每当 MAKE 遇到宏名，它就用扩展文本来取代宏名。

假设在 make 文件起始处定义了如下的宏：

```
SRC=C:\ASM\
OUT=OBJS\
INC=C:\INC\
```

现在，make 文件其余部分就变成了：

```
getstars.exe :  $(OUT)getstars.obj  $(OUT)gsparse.obj \
                $(OUT)gscomp.obj  $(OUT)starlib.obj
  tlink  $(OUT)starlib  $(OUT)gsparse $(OUT)gscomp\
      $(OUT)getstars,  $(OUT)getstars,  $(OUT)getstars,  lib\math\
      lib\io
getstars.obj:  $(SRC)getstars.asm  $(INC)stardefs.inc  \
               $(INC)starlib.inc  $(INC)gscomp.inc  \
               $(INC)gsparse.inc
  tasm  /t  /ml  /s  /i$(INC)  $(SRC)getstars.asm,  \
               $(OUT)getstars.obj
gscomp.obj:  $(SRC)gscomp.asm  $(INC)stardefs.inc  \
               $(INC)starlib.inc
  tasm  /t  /ml  /s  /i$(INC)  $(SRC)gscomp.asm,  $(OUT)gscomp.obj
gsparse.obj:  $(SRC)gsparse.asm  $(INC)stardefs.inc  \
               $(INC) starlib.inc
  tasm  /t  /ml  /s  /i$(INC)  $(SRC)gsparse.asm,  \
               $(OUT)gsparse.obj
starlib.obj:  $(SRC)starlib.asm
  tasm  /t  /ml  /s  /i$(INC)  $(SRC)starlib.asm,  \
               $(OUT)starlib.obj
```

当运行 MAKE 时，$(SRC)就用扩展文本 C:\ASM\来替代，$(INC)就用扩展文本 C:\INC\来替代，而 $(OUT)就用扩展文本 OBJS\来替代。

从这里获得了什么?灵活性。只要改变宏中的任一个，你就改变了依赖该宏的所有命令。例如，假设要把包含文件移动到被称为 C:\INC\STAR 的新的子目录下。需做的全部工作就是更新 MAKE 文件中的 INC 宏为：

```
INC=C:\INC\STAR\
```

这样，就把所有命令改变为使用这个新的包含子目录。

宏定义采取如下形式：

```
宏名=扩展文本
```

宏名是宏的名字，它为没有空白的字母数字串，当然，你可在宏名和等号(=)间加入空白。扩展文本是一任意串，可包含字母、数字、空白和标点；由一回车换行终结。

如果宏名以前已定义过，不管是通过 make 文件中的宏定义还是通过 MAKE 命令行中的－D 选择项来定义，新的定义都将取代旧的定义。

在宏中，大小写是有意义的：即命名为 mdl、Mdl 和 MDL 的宏被看作是不同的宏。在你的 make 文件中，宏的调用格式为：

```
$(宏名)
```

圆括号是所有调用所必需的，即便宏名只为一个字符也这样，且只对以后我们要讲到的三个特殊预定义宏例外。$(宏名)结构就被看作是宏调用。

当 MAKE 遇到一宏调用时，它就用宏扩展文本取代宏调用。如果宏没有定义，MAKE

就用空串取代它。

下面是一些宏的特殊情况：

在宏中的宏　宏不能嵌入到左边(宏名)而只能用于宏定义的右边(扩展文本)，但只有当被定义的宏在被调用时才能进行宏扩展。换言之，当宏调用被扩展时，嵌在扩展文本中的宏也被扩展。

在规则中的宏　在规则中的宏调用被立即扩展。

在伪指令中的宏　在!if 和!elif 伪指令中的宏调用被立即扩展。如果在!if 和!elif 伪指令间的宏当前无定义，它就扩展为值 0(假)。

在命令中的宏　当命令执行时，在命令中的宏调用就被扩展了。

下面是有关预定义宏的情况：

MAKE 在内部定义了几个特殊宏：$d、$*、$c、$:、$.和$&。第一个是在条件伪指令!if 和!elif 中的定义测试宏；其余几个是用在显式规则和隐含规则中的文件名宏。此外，当前的 SET 环境串被当作宏来自动装入，MAKE 宏被定义为 1。

定义测试宏($d)　如果所给的宏名有定义，定义测试宏$d 就扩展为 1，否则为 0。宏扩展文本的内容不起作用。本特殊宏只允许在!if 和!elif 伪指令间使用。例如，假设想要修改 make 文件，以使它在你未指定内存模式时使用中型内存模式，可以把下面内容放在 make 文件中的起始部分：

```
!if  !$d(INC)      #if INC is not defined
INC = c:\INC\      #define the default path
!endif
```

可用下面的命令行调用 MAKE：

```
make  -D  INC = c:\INC\STAR\
```

这时 INC 就被定义成 c:\INC\STAR\，但如果你用 MAKE 本身来

调用：

```
make
```

那么 INC 就被定义成 C:\INC\，即你的"缺省"内存模式。

不同文件名宏　不同文件名宏按类似的方式工作，它把正被创建的文件扩展成全路径名的某些变体。

基文件名宏($*)　基文件名宏可允许用在显式规则和隐含规则的命令中。本宏($*)扩展成正被创建的文件名，但除去扩展名，如下所示：

文件名为　A:\P\TESTFILE.ASM

$*扩展为　A:\P\TESTFILE

例如，可把显式 GETSTARS.EXE 规则修改成如下形式：

```
getstars.exe: getstars.obj gscomp.obj gsparse.obj\starlib.obj tlink starlib gsparse gscomp $*,$*,$*,\lib\emu lib\math lib\io
```

当执行本规则的命令时，宏$*就被目标文件(不带扩展

名）getstars 所替代。对于隐含规则，本宏也很有用。例如，一TASM 隐含规则可定义如下：

```
.asm.obj:
tasm  /t  $*
```

**全文件名宏（$<）**　全文件名宏也是用在显式规则和隐含规则中。在显式规则中，$<扩展成全目标文件名（包括扩展名），如下：

文件名为　A:\P\TESTFILE.ASM
$<扩展成　A:\P\TESTFILE.ASM

例如，规则

```
starlib.obj:  starlib.asm
  copy  $<  \oldobjs
  tasm  /t  $*
```

将在编译 STARLIB.ASM 前把 STARLIB.OBJ 拷贝到\OLDOBJS 目录下。在隐含规则中，$<代表文件名加上源文件扩展名。例如，前面的隐含规则

```
.obj.asm:
  tasm  /t  $*.asm
```

可重写为

```
.obj.asm:
  tasm  /t  $<
```

文件名路径宏（$:）　本宏扩展成路径名（不含文件名），如下：

文件名为　A:\P\TESTFILE.ASM
$:扩展成　A:\P\

文件名和扩展名宏（$.）　本宏扩展成带扩展名的文件名，如下：

文件名为　A:\P\TESTFILE.ASM
$.扩展成　TESTFILE.ASM

单文件名宏（$&）　本宏只扩展成文件名，不带路径和扩展名，如下：

文件名为　A:\P\TESTFILE.ASM
$&扩展成　TESTFILE

伪指令

Turbo Assembler 的 MAKE 允许做些其它 MAKE 版本不允许做的事情。伪指令就类似于汇编本身所允许使用的伪指令。可用这些伪指令来包含其它 make 文件，生成规则和命令条件式，打印出错信息，消除宏定义。

make 文件中的伪指令用感叹号（!）作为一行的头一个字符，而汇编是用#号。下面是完整的 MAKE 伪指令表：

```
! include
! if
! else
! elif
! endif
```

! error

! undef

文件包含伪指令

文件包含伪指令(! include)指定了从本伪指令这点开始包含一文件到 make 文件中，并解释执行。它采取下面的形式：

! include　"文件名"

这些伪指令可任意嵌套。如果文件包含伪指令试图包含一个已被某外层嵌套包含的文件(这会导致一嵌套循环)，内层的文件包含伪指令就被当作错误来拒绝。那么，如何使用本伪指令呢？假如创建了一个文件 MODEL.MAC，它含有下述内容：

```
! if  ! $d(MDL)
MDL=medium
! endif
```

可通过在任何 make 文件中包含下面伪指令

! include　"MODEL.MAC"

来利用该条件宏定义。当 MAKE 遇到! include 伪指令时，它就打开所指定的文件，读入其中的内容，就象它是 make 文件一样。

条件伪指令

条件伪指令(! if,! elif,! else 和! endif)在构造 make 文件中给程序员提供了一种灵活的方法。规则和宏均可条件化，因此命令行宏定义(使用-D 选择项)就能有选择地使用 make 文件中的各部分。这些伪指令的格式类似于汇编预处理器中的伪指令：

```
! if    表达式
 [行表]
! endif
! if    表达式
 [行表]
! else
 [行表]
! endif
! if    表达式
 [行表]
! elif  表达式
 [行表]
! endif
```

注意：[行表]可选择下述内容组成：

- 宏定义
- 显式规则
- 隐含规则
- 包含伪指令
- if 类伪指令

出错伪指令

消除定义伪指令

条件伪指令构成一组，该组中至少有一个打头的！if 伪指令和用作终结的！endif 伪指令。

！else 伪指令可出现在组中。

！elif 伪指令可出现在！if 和！else 伪指令间。

规则、宏和其它伪指令可在不同的条件伪指令中出现多次。注意，带有命令的完整规则不能用条件伪指令隔开。

条件伪指令组可任意嵌套。

在一源文件中，任何规则、命令或伪指令都必须是完整的。

在同一源文件中！if 伪指令和！endif 伪指令必须匹配。因此，下面的包含文件是非法的，这与文件中所包含的内容无关，因为并没有相匹配的！endif 伪指令：

```
！if  $(FILE  COUNT) > 5
    一些规则
！else
    其它规则
<文件结束符>
```

条件伪指令中允许使用的表达式：！if 或！elif 伪指令允许使用的表达式采用类似于汇编的语法。表达式的求值结果为 32 位带符号的整数表达式。数可为十进制、八进制或十六进制常量。例如，如下是表达式中的合法常量：

```
4336      #十进制常量
0477      #八进制常量
0x23aF    #十六进制常量
```

表达式可使用下列单目操作符：

−	负
~	按位取补
!	逻辑“非”

表达式可使用下列双目操作符：

+	加
−	减
*	乘
/	除
%	求余数
>>	右移
<<	左移
&	按位“与”
|	按位“或”
^	按位“异或”
&&	逻辑“与”
||	逻辑“或”

>	大于
<	小于
>=	大于等于
<=	小于等于
==	相等
! =	不等

表达式含有如下三目操作符：

?:　在? 以前的操作数被看成是测试。如果该操作数值为非零，那么第一个操作数(在? 与:之间)就是结果。如果第一个操作数为 0，结果就是第三个操作数的值(在:部分后面)。

在表达式中可用圆括号把操作数括成一组。如果不用圆括号，双目操作符按与汇编中给出的优先级分成组。跟在汇编中一样，相同优先级的操作符从左到右分组，只有三目操作符(?:)例外，它从右向左分组。

可在表达式中调用宏，且特殊宏 $d()能被识别。在所有宏扩展之后，表达式必须具有适当的语法。在扩展过的表达式中的任何单词都被看作是错误。

出错伪指令

出错伪指令(! error)使 MAKE 中止，并打印出! error 后面跟着的错误诊断文本。它采取如下形式：

! **error**　　任何文本

本伪指令可设计在条件伪指令中来允许使用用户定义的出错条件。例如，你可在第一条显式规则前插入如下代码：

```
! if ! $d(MDL)
#如果 MDL 未定义 •
! error MDL not defined
! endif
```

如果到达这点而 MDL 没有定义过，则 MAKE 将中止，并产生如下的出错信息：

```
Fatal makefile 5: Error directive: MDL not defined
```

消除定义伪指令

消除定义伪指令(!undef)会引起消除掉相应宏名的定义。如果宏未被定义，本伪指令不起作用。语法形式如下：

```
! undef　　宏名
```

B.1.3　使用 MAKE

已经知道了有关写 make 文件的许多东西，现在该是学习如何使用 MAKE 的时候了。

B.1.3.1 命令行语法

使用 MAKE 的最简单方法是在 MS-DOS 提示符处打入命令

```
make
```

MAKE 就开始查找 MAKEFILE；如果找不到，它就查找 MAKEFILE.MAK；如果找不到的话，就中止并打印出错信息。

如果要查找 MAKEFILE 和 MAKEFICE. MAK 以外的文件名，可提供文件选择项(-F)如下：

```
make      -fstars. mak
```

MAKE 的一般语法是：

```
make    选择项    选择项  ...  目标  目标...
```

其中选择项为 MAKE 选择项(在后面讨论)，目标为由显式规则处理的目标文件名。

下面是语法规则：

- make 单词后接一个空格，然后是 make 选择项表。
- 每个 make 选择项与其相邻选择项间用空格隔开。选择项放的次序是任意的，输入的选择项数目也是任意的(只要还有命令行空间就行)。
- make 选择项表之后是一空格，然后是可选的目标表。
- 每个目标与其相邻目标间用空格隔开。MAKE 按目标文件的次序来执行，必要时可重编译其子部分。

如果命令行中不包含任何目标名，MAKE 就使用一显式规则中所指定的第一个目标文件。如果在命令行中指定各个目标，它们就按需要来创建。

下面是 MAKE 命令行的一些例子：

```
make -n -fstars. mak
make -s
make -Iinclude   -DMDL=compact
```

B. 1. 3. 2 中止 MAKE 的说明

可用 Ctrl Break 来中止 MAKE 的任何命令及 MAKE 本身。同样 Ctrl-C 也会中止当前执行的命令和 MAKE 本身。

B. 1. 3. 3 BUILTINS. MAK 文件

当使用 MAKE 时，常常会发现有许多宏和规则(通常是隐含规则)被你不断使用。已有了处理它们三种方法。第一，把它们放在创建的每个 make 文件。第二，把它们放在一文件中并在创建的每个 make 文件中使用! include 伪指令。第三，把它们都放到一个名叫 BUILTINS. MAK 的文件中。

每当运行 MAKE 时，它都查找 BUILTINS. MAK 文件；如果找到了该文件，MAKE 就在处理 MAKEFILE(或其它要处理的 make 文件)之前把该文件读入。

BUILTINS. MAK 文件中可含有任何规则(通常是隐含规则)或宏，它们能普遍地用于计算机的各处文件中。

并不要求 BUILTINS. MAK 文件一定存在。如果 MAKE 找到了 BUILTINS. MAK 文件，就先解释执行它。如果 MAKE 没找到 BUILTINS. MAK 文件，它就直接处理 MAKEFILE(或其它你所指定的 make 文件)。

B. 1. 3. 4 MAKE 是如何查找 make 文件的

MAKE 将在当前目录下或在路径中的任何目录下查找 BUILTINS. MAK。你应该把该文件放在 MAKE. EXE 文件所在的目录下。

MAKE 只在当前目录下查找 make 文件。该文件包含有一些规则，这些规则用于创建特殊的可执行程序文件。BUILTINS. MAK 与 make 文件有相同的语法规则。

MAKE 也查找当前目录下的任何 include 文件。如果你使用-I(包含)选择项,它也在指定的目录下查找。

B.1.3.5 TOUCH 实用程序

有时,想要强制重编译或重创建一特殊目标文件,而该目标文件所依赖的源文件却没有改动。解决这个问题的一种方法就是使用 Turbo Assembler 中所包含的 TOUCH 工具。TOUCH 把若干文件的日期和时间改成当前的日期和时间,这样,就使相应文件比它所依赖的文件更"新"。

要强制重建一目标文件,就要对该目标所依赖的文件使用 TOUCH,用法如下,在 DOS 提示符下打入

touch 文件名 [文件名 ...]

TOUCH 这时将更新文件的创建日期。

这样做之后,你就可用 MAKE 来重建 TOUCH 过的目标文件了(你可在 TOUCH 中使用 DOS 匹配符 * 和?)。

B.1.3.6 MAKE 命令行选择项

我们在前面还提到过 MAKE 的一些命令行选择项,现在给出一个完整的表。注意:大小写是有意义的。

-a	产生自动依赖性检查。
-D 标识符	定义命名的标识符。
-D 标识符=串	把命名标识符定义为等号后的串。串中不能含有空格和制表符。
-I 目录	MAKE 将查找所指定目录下的包含文件(也包括当前目录)。
-U 标识符	消除以前的命名标识符定义。
-S	通常情况下,MAKE 在要执行一命令时打印出该命令。带-S 选择项后,在执行前就不打印命令了。
-n	使 MAKE 打印命令,但并不实际执行这些命令。这对 make 文件的调试是很有用的。
-f 文件名	把给出的文件名作为 MAKE 文件名。如果给出的文件不存在,且不带扩展名,则试用"文件名.MAK"。
-? 或-h	打印帮助信息。

B.1.4 MAKE 出错信息

MAKE 的诊断信息可分成两类:致命错和一般错。当发生致命错时,编译就立即停止了。你必须采取适当的措施来重启编译。一般错是指明一 make 源文件中某些类语法和语义错误。MAKE 将完成对 make 文件的解释,然后中止。

B.1.4.1 致命错

Don't know how to make xxxxxxxx 本信息是 MAKE 在遇到创建序列中不存在的文件时产生的,也可能是不存在相应规则来允许相应文件的创建。

Error directive: xxxx 当 MAKE 处理源文件中的 #error 伪指令时产生本信息。该伪指令的文本以信息形式显示。

Incorrect command-line argument: xxx 如果 MAKE 执行时带有不正确的命令行参数,则发生本错误。

Not enough memory 当整个工作空间都耗尽时会发出此信息。可用增加内存的办法来解决这个问题。如果已用了机器上的 640k 内存,就只能简化源文件了。

Unable to execute command 本信息在一命令执行后发出。这可能是因为没有找到命令文件,或命令文件拼写错了。一般不会发生的可能性是命令虽存在,但它有些问题。

Unable to open makefile 在当前目录不包含名为 MAKEFILE 的文件(或其它指定的 make 文件)时,发出本信息。

B. 1. 4. 2 一般错

Bad file name format in include statement 包含文件名必须用引号或尖括号围起来。文件名可能丢失了引号或尖括号。

Bad undef statement syntaxundef 语句必须包含一个标识符,而在语句体内不含任何其它东西。

Character constant too long MAKE 所执行的命令参数最多达 127 个字符——这是 MS-DOS 规定的限制。

Command syntax error 在下面情况下发出本信息

- make 文件的头一个规则行含有打头的空白。
- 隐含规则不是由.扩展名.扩展名:来组成的。
- 显式规则中冒号(:)字符前不含有名字。
- 宏定义中等号(=)字符前不含有名字。

Division by zero 在! if 语句中做除法和求余数时以零做除数。

Expression syntax error in ! if statement ! if 语句中的表达式组成有错**k k** 它可能含有未匹配的圆括号,或额外的操作符,及额外或不存在的常量。

File name too long 在! include 伪指令中给出的文件名过长以致编译器不能处理。在 MS-DOS 中的文件名必须不得超过 64 个字符。

Illegal character in constant expression x MAKE 在一常量表达式中遇到某个不允许使用的字符。如果字符是一字母,这很可能表明是拼写错了的标识符。

Illegal octal digit 在八进制常量中发现含有数字 8 和 9。

Macro expansion too long 宏扩展不能超过 4 096 个字符。如果宏是递归地扩展自己,常会发生本错。一个宏不能对自身进行合法扩展。

Misplaced elif statement 在遇到! elif 伪指令时没有与之相匹配的! if 伪指令。

Misplaced else statement 在遇到! else 伪指令时没有与之相匹配的! if 伪指令。

Misplaced endif statement 在遇到! endif 伪指令时没有与之相匹配的! if 伪指令。

No file name ending 对包含语句中的文件名丢失了用于关闭的引号或尖括号。

Redefinition of target xxxxxxxx 命名文件出现在多个显式规则的左边。

Unable to open include file xxxxxxxx. xxx 没找到命名文件。这也可能是一包含文件包含了自身。检查一下命名文件是否存在。

Unexpeeted end of file in conditional started on line # 在 MAKE 遇到! endif 之前源文件终结。! endif 或者是丢失了或者是拼写错了。

Unknown preprocessor statement 在一行起始处遇到一个! 字符,而跟着的语句名却不是 error,rdef,if,elif,include,else 和 endif。

B.2 Turbo Link

Turbo Link(TLINK)可作为单独的程序来调用,也可用作独立的连接器。

TLINK 短小而精悍。它虽缺乏其它连接器中的一些声响,但却十分快速且紧凑。在本附录中,我们描述如何把 TLINK 作为独立的连接器来使用。

B.2.1 调用 TLINK

可在 DOS 命令行上通过打入带或不带参数的 tlink 来启动 TLINK。

当不带参数来调用 TLINK 时,TLINK 就显示如下的参数和选择项概要:

```
Turbo Link version 2.0 Copyright (c) 1987,1988 Borland International
The Syntax is: TLINK objfiles,exefile,mapfile,libfiles
@xxxx 表明使用应答文件 xxxx
options:
    /m = map file with publics
    /x = no map file at all
    /i = initialize all Segments
    /l = include Source line numbers
    /s = detailed map of Segments
    /n = no default libraries
    /d = warn if duplicate Symbols in libraries
    /c = lowercase Signifieant in Symbols
    /3 = enable 32-bit processing
    /v = include full symbolic debug information
    /e = ignore Extented Dictionary
    /t = Creat COM file
```

在 TLINK 概要显示中,行

```
The syntas is: TLINK objfiles,exefile,mapfile,libfiles
```

指定了要按给定的次序来提供文件名,不同类的文件用逗号隔开。

例如,如果提供命令行

```
tlink /c mainline wd ln tx, fin, mfin, lib\comm lib\support
```

TLINK 将把它解释成:

- 在连接期间,大小写是有意义的(/c)。
- 要连接的.OBJ 文件是 MAINLINE.OBJ,WD.OBJ,LN.OBJ 和 TX.OBJ。
- 可执行程序名为 FIN.EXE
- 映射文件为 MFIN.MAP
- 要连入的库文件是 COMM.LIB 和 SUPPORT.LIB,这两个库文件都是在子目录

LIB 下。

TLINK 对不带扩展名的文件名按下述原则来填加扩展名：

- 对目标文件为.OBJ
- 对可执行文件为.EXE；对带/t 选择项的可执行文件为.COM
- 对映射文件为.MAP
- 对库文件为.LIB

注意，如果没指定.EXE 文件的名字，TLINK 就把列表的第一个目标文件的名字加上.EXE 构成可执行文件名。例如，如果上例中没指定 FIN 为.EXE 文件的名字，TLINK 就创建 MAINLINE.EXE 作为可执行文件名。

TLINK 总是生成一个映射文件，除非在命令行上用/x 选择项来明确地表明不生成映射文件。有关映射文件有下面两种情况：

- 如果给了/m 选择项，映射文件则包含公用符号。
- 如果给了/s 选择项，映射文件为一详细的段映射。

下面是 TLINK 用来确定映射文件名的规则：

- 如果没指定.MAP 文件，TLINK 就把.EXE 文件名加上.MAP 扩展名来构成映射文件名(.EXE 文件的名字可在命令行中给出，也可由应答文件给出，没给出.EXE 文件的名字，TLINK 就从第一个.OBJ 文件名中取出)。
- 如果映射文件名已在命令行中(或在应答文件中)指定，TLINK 就对给出的名字加上.MAP 扩展名。

注意，即使指定了映射文件名，但如果同时也指定了/x 选择项，那么根本就不创建映射文件。

B.2.2 使用应答文件

TLINK 允许提供各种参数。这些参数可放在命令行上，也可放在应答文件中，或是这两个的组合。

应答文件就是含有选择项和/或文件名的文本文件，这些选择项及文件名通常是在命令行上 TLINK 名字后打入的。

与命令行不同，应答文件能延续成几个文本行。可以把目标文件或库文件的长表分成几行，每行用一个加号(+)结束，这样就可把列表延续到下一行。

也可把 TLINK 的四个组成部分用单独的行来起始，即分别为目标文件、可执行文件、映射文件和库文件。这样做时，就必须略去用来分隔各部分的逗号(,)。

为说明这些特性，假设用应答文件来重写前面的命令行例子，设文件名为 FINRESP，内容如下：

```
/c mainline wd+
ln  tx, fin+
mfin +
lib\comm  lib\support+
```

这时，你就可以打入 TLINK 命令

```
tlink @finresp
```

注意，在应答文件名前必须加一个 at 字符(@)，这可用来表明下一个名字是一个应答文件名。同样的，也可把前面的命令分解到多个应答文件中。例如，可把前面的命令分解到下面两个应答文件中：

文件名	内容
LISTOBJS	mainline+
	wd+
	ln　tx
LISTLIBS	lib\comm+
	lib\support

这时，就可以打入 TLINK 命令

```
tlink /c　@Listobjs,fin,mfin,@Listlibs
```

B. 2. 3.　TLINK 选择项

TLINK 选择项可出现在命令行上任意处。选择项由一斜杠(/)开始，后面跟着指定选择项的字符(m,s,v,i,n,d,x,3,v,e,t,或 c)。

如果有多个选择项，空格并没有什么实际意义(/m/c 与/m /c 是一样的)，可把它们放到命令行上的不同地方。下面各小节将分别描述各选择项。

B. 2. 3. 1　/x,/m,/s 选择项

在缺省情况下，TLINK 总是创建可执行文件的映射。这个缺省映射只包括程序中段的表，程序起始地址，及连接期间生成的警告或出错信息。如果根本不想产生一个映射文件，可用/x 选择项。

如果想要创建一个更完备的映射文件，/m 选择项将在映射文件中加入公用符号表，并按地址增加次序来排序。这类映射文件在调试时是很有用的。许多调试器，如 Periscope，可用公用符号表来允许你检索正在调试的符号地址。

/s 选择项不仅象/m 选择项一样可用来创建一带有段、公用符号和程序起始地址的映射文件，也在其中加入详细的段映射。下面是详细段映射的一个例子：

对每一模块中的每一个段，映射包括地址、按字节计数的长度、类、段名、组、模块和 ACBP 信息。

地址	长度(字节数)	类	段名	组	模块	调准/组合
0000:0000	0E5B	C=CODE	S=SYMB	TEXT	G=(none)	M=SYMB. ASM	ACBP=28
00E5:000B	2735	C=CODE	S=QUAL	TEXT	G=(none)	M=QUAL. ASM	ACBP=28
0359:0000	002B	C=CODE	S=SCOPY	TEXT	G=(none)	M=SCOPY	ACBP=28
035B:00013	003A	C=CODE	S=LRSH	TEXT	G=(none)	M=LRSH	ACBP=20
035F:0005	0083	C=CODE	S=PADA	TEXT	G=(none)	M=PADA	ACBP=20
0367:0008	005B	C=CODE	S=PADD	TEXT	G=(none)	M=PADD	ACBP=20
036D:0003	0025	C=CODE	S=PSBP	TEXT	G=(none)	M=PSBP	ACBP=20

```
036F:0008  05CE    C=CODE  S=BRK  TEXT    G=(none)  M=BRK          ACBP=28
03CC:0006  066F    C=CODE  S=FLOAT  TEXT  G=(none)  M=FLOAT        ACBP=20
0433:0006  000B    C=DATA  S=DATA          G=DGOUP   M=SYMB. ASM   ACBP=48
0433:0012  00D3    C=DATA  S= DATA         G=DGROUP  M=QUAL. ASM   ACBP=48
0433:00E6  000E    C=DATA  S= DATA         G=DGROUP  M=BRK          ACBP=48
0442:0004  0004    C=BSS   S= BSS          G=DGROUP  M=SYMB. ASM   ACBP=48
0442:0008  0002    C=BSS   S= BSS          G=DGROUP  M=QUAL. ASM   ACBP=48
0442:000A  000E    C=BSS   S= BSS          G=DGROUP  M=BRK          ACBP=48
```

如果同一段出现在多个模块中，每个模块就作为一个独立行(如 SYMB. ASM)出现。详细段映射的大部分信息是自解释的，只有 ACBP 字段例外。

ACBP 字段把 A(调准)和 C(组合)属性编码成一系列 4 位字段，这是由核心定义的。TLINK 只使用这些字段中的两个，即 A 字段和 C 字段。ACBP 的值在映射文件中以十六进制形式打印的；下面两字段中的值或在一起构成了 ACBP 中打印的值。

字段	值	描述
A 字段	00	绝对段
(调整)	20	字节调整的段
	40	字调整的段
	60	节调整的段
	80	页调整的段
	A0	无名的绝对存储部分
C 字段	00	不能组合
(组合)	08	可公用组合的段

B. 2. 3. 2 /l 选择项

/l 选择项在. MAP 文件中为源代码行数创建一个小节。为了能使用它，必须在创建. OBJ 文件时使用-Y(行数打开)选择项。如果告诉 TLINK 根本不创建映射文件(用/x 选择项)，本选择项就不起作用了。

B. 2. 3. 3 /i 选择项

即使尾段中不含有数据记录，/i 选择项也会使尾段输出到可执行文件中。注意，这通常并不是必需的。

B. 2. 3. 4 /n 选择项

/n 选择项使连接器忽略掉某些编译器所指定的缺省库。如果缺省库在另一个目录下，本选择项是必需的，因为 TLINK 并不支持对库的查找。当连接用其它语言写的模块时可使用本选择项。

B. 2. 3. 5 /c 选择项

/c 选择项强制在公用符号和外部符号中大小写是有意义的。例如，在缺省条件下，TLINK 把 fred，Fred 和 FRED 看作是相同的，而/c 选择项就使它们成为不同的。

B.2.3.6　/d 选择项

在正常情况下，如果一符号在多个库文件中出现，TLINK 并不警告你。如果该符号一定要包含在程序中，TLINK 使用命令行上第一个出现该符号的文件中的拷贝。由于这是普遍性的使用特性，TLINK 通常就不警告重复符号。下面的假想情况将说明如何使用本特性。

假设你有两个库：一个称为 SUPPORT.LIB，一个辅助的称为 DEBUGSUP.LIB。假设在 DEBUGSUP.LIB 中含有 SUPPORT.LIB 中一些例程的重复例程(DEBOGSUP 中的重复例程具有一些稍微不同的功能，如例程的调试版本)。如果在连接命令中首先包含 DEBUGSUP.LIB，得到的将是调试例程而不是 SUPPORT.LIB 中的例程。

如果没使用本特性或不知道哪个例程是重复的，可用/d 项。它将强制 TLINK 列出库中所有的重复符号，即使在程序中并不用到这些符号。

/d 选择项也强制 TLINK 警告同时出现在.OBJ 文件和.LIB 文件中的符号。在这种情况下，由于在命令行文件列表中头一个文件(从左开始)中出现的符号是要连入的，在.OBJ 文件中的相应符号则是要使用的。

B.2.3.7　/e 选择项

和 Turbo C 装在一起的库文件都含有扩展词典信息，它可使 TLINK 对这些库进行快速连接。在 TLIB 中使用/E 选择项(见 TLIB 小节)可把扩展词典加到其它库文件中去。

尽管带有扩展词典的库连接快速，但仍有两个原因要使用/e 开关，/e 不允许使用扩展词典。

- 当使用扩展词典时，程序连接时需要多一些内存。
- TLINK 将忽略掉带有扩展词典的库中的调试信息，除非使用/e。

B.2.3.8　/t 选择项

如果在微型存储模式下连接并触发此开关，TLINK 就不产生通常的.EXE 文件而是产生一个.COM 文件。

使用/t 选择项，可执行文件的缺省扩展名为.COM。

注意：.COM 文件长度不能超过 64K，没有关于段的驻留内容，没定义堆栈段，且起始地址等于 0:100h。当可执行文件不使用.COM 为扩展名(如.BIN)时，起始地址可为 0:0 或 0:100h。

B.2.3.9　/v 选择项

/v 选择项让 TLINK 在可执行文件中包含调试信息。

注意，当用/v 选择项来连接时，TLINK 初始化所有段，如果程序在连入调试信息时运行有差错，一定有个未初始化的变量。

B.2.3.10　/s 选择项

当连接由 TASM 或其他兼容汇编生成的若干目标模块，且含有用于 80386 处理器的 32 位代码时，应该用/s 选择项。

本选择项增加 TLINK 的内存需求，但减慢连接，所以只能在必要时才可使用。

B.2.4　一些限制

如前所述，TLINK 短小而精悍，它不提供过多的选择项。下面只是 TLINK 的一些重要限制：

- 不支持重载。
- 部分支持公用变量:必须提供一公用符号来解决。
- 最多可拥有 4 000 个逻辑段
- 具有相同和类的段应该或者全部能组合,或是都不能。
- TLINK 最后装入 STACK 类的任何段,即使这些段是 DGROUP 的一部分。
- 用 Microsoft C 或 Microsoft Fortran 编译出的代码不能用 TLINK 连接。这是因为 MiCrosoft 语言在其. OBJ 文件中拥有非文本目标记录格式,而 TLINK 目前不支持。TLINK 设计用于 Turbo Assembler、Turbo C(包括集成环境和命令行版本),Turbo Prolog 和其他编译器,但它一般不能取代 MS LINK。

B.2.5 出错信息

TLINK 有三类错误:致命错、非致命错、警告。

- 致命错使 TLINK 立即中止,并删除掉. EXE 和 MAP 文件。
- 非致命错不删除掉. EXE 或. MAP 文件,但不宜试图执行. EXE 文件。
- 警告只是对你可能应固定的条件的警告。当发出警告时,仍建. EXE 和. MAP 文件。

下面是以后各小节中出错信息要用到的一般性的名字和值。如果得到一具体的出错信息,它们就被相应的名字或值来替换。

<sname>	符号名
<mname>	模块名
<fname>	文件名
XXXXh	4 个数字的十六进制数,以"h"结尾

B.2.5.1 致命错

当发生致命错时,TLINK 将中止并删除掉. EXE 和. MAP 文件。

XXXXXXXX. XXX : bad object file 遇到一错误形式的目标文件。这主要是由于命名的源文件或目标文件没有被完全建立起来。其他可能的情况是:在编译期间机器被重新启动或编译器还没删掉其输出的目标文件时就打入了 Ctrl-Break。

XXXXXXXX. XXX: unable to open file 这是由于命名的文件不存在或拼错写了。

Bad charaeter in parameters 在命令行或应答文件中遇到下列字符之一:", *, <, =, >, ?,[,],|或除了水平制表,移行字符,回车换行及 Ctrl-Z 外的其他控制字符。

Msdos error,ax=xxxxh 这在 MS-DOS 调用返回一未预期错时出现。打印的 AX 值是相应的错误代码。这表明是 TLINK 内部错或 MS-DOS 错。TLINK 产生的 MS-DOS 调用错误发生在读,写和关闭期间。

Not enough memory 没有足够的内存来完成连接过程。试着删掉当前装入的终端和驻留应用程序或减小当前激活的 RAM 的大小。然后再运行 TLINK。

Segment exceeds 64K 如果给定的数据或代码段定义了过多的数据,就会发出本信息,这主要是在把不同源文件中同名段组合在一起时。如果把组中的段组合在一起,该组超过 64K 字节时,也会发出此信息。

Symbol limit exceeded 在一单个连接过程中,你最多能定义的公用符号、段名和组名

为 8 182 个。如果超过此限制，就会发出本信息。

Unexpected group definition　目标文件中的组定义必须以一特定序列出现。本消息一般只在编译器生成了有错误的目标文件时才出现。如果本信息发出时，相应文件是由 Turbo Assembler 创建的，试着重编译该文件一次。再有问题，请与 Borland 公司联系。

Unexpected Segment definition　目标文件中的段定义必须以一特定序列出现。本信息一般只在编译器生成了有错误的目标文件时才出现。如果本信息发出时，相应文件是由 Turbo Assembler 创建的，试着重编译该文件一次。

Unknown option　在命令行或应答文件中遇到了斜杠(/)，而后面跟着的不是所允许的选择项。

write failed, disk full?　如果 TLINK 不能写完它要写的内容，就发出本信息。这主要是由于磁盘已满引起的。

D. 2. 5. 2　非致命错

TLINK 只有两个非致命错。如上所述，当出现非致命错时不删掉. EXE 和. MAP 文件。下面是出错信息：

XXX is unresolved in module YYY　命名的符号在给定的模块中引用但它在目标文件中和连接所包含的库中没有定义。检查一下符号的拼写是否正确。

Fixup overflow, fname＝xxxxh, target＝xxxxh

offset＝xxxxh in modulde　xxxxxxxs

这表明目标文件中不正确的数据或代码引用，而在连接期间，TLINK 必须把这些数据或代码约定好。在一约定中，目标文件表明了被引用的内存位置名和内存位置应在的段的名字。fname 值是按目标文件要求的内存位置应处的段。而 target 值是内存位置实际所处的段。offset 字段是内存位置在目标段内的位移量。

本信息的发出主要是由于内存模式的不匹配。近程调用一个不同代码段中的函数是最有可能的原因。如果你对数据变量或对函数的数据引用生成一近程调用，就会导致本错误。为诊断此问题，可产生带有公用符号的映射(/m)。出错信息中目标字段和位移量字段的值应该是所引用符号的地址。如果目标字段和位移量字段不与映射中的符号匹配，可以查找离信息所给地址最近的符号。引用是在命名模块中，所以可在模块的源文件中查找相冲突的引用。

如果用这些办法还不能查明出错的原因，而在用除 Turbo Assembler 外的汇编语言或其他高级语言来进行程序设计，也有可能是别的原因诱发此信息。即便在 TurboAssembler 下，如果在使用与给定内存模式缺省值不同的段或组名，也会产生本信息。

B. 2. 5. 3　警　　告

TLINK 只有三种警告。头两个是有关符号的重复定义，第三个只适用于微型模式程序，表明没有定义栈。下面是这些信息：

warning: xxx is duplicated in module YYY　命名符号在命名模块中被定义两次。这在 Turbo Assembler 的目标文件中可能发生，例如，在源文件中只是大小写不同的两个不同的 Pascal 名字。

warning: xxx defined in module YYY is duplicated in module ZZZ　命名符号在每个命名模块中都有定义。如果在命令行中给出了两个相同目标文件名，或者该符号的两个拷贝之

一被拼写错了，都会导致产生本警告。

warning：no stack 如果在任何目标文件中和任何连接所包含的库中没有堆栈段的定义，就发出本警告。这对 Turbo C 中的微型内存模式或被转换成. COM 文件的任何应用程序通常都会发出本警告。对别的程序，这表明是出错。

如果 Turbo Assembler 程序只对微型内存模式产生本信息，检查一下 COX 起始的目标文件来确定它们是否正确。

B.3 TLIB：Turbo 库管理员

TLIB 是 Borland 公司的 Turbo 库管理员。它是用来管理每个. OBJ 文件(目标模块)中的库。库是把一些目标模块收集起来作为一个单元的很便利的方法。

使用 TLIB，可建造自己的库，也可修改自己的库，或其它程序员的库，及购买的商品化的库。可把 TLIB 用来：

- 从一组目标模块中创建一个新库
- 往现存的库中填加目标模块或别的库
- 从现有的库中消除目标模块
- 替换现有库中的目标模块
- 从现有库中提取目标模块
- 对现有库或新库的内容进行列表

当修改现有的库时，TLIB 总是为原来的库创建一个带有. BAK 扩展名的拷贝。

TLIB 也可创建(包含在库文件中)一扩展词典，可用它来加快连接。详见下面关于/E 选择项的小节。

尽管 TLIB 对 Turbo Assembler 创建可执行程序不是必需的，但它对提高程序员的生产效率却很有用。会发现 TLIB 对大型开发项目来说是不可缺少的。如果在用别人的目标模块库，可在必要时使用 TLIB 来维护这些库。

B.3.1 使用目标模块库的优点

当用汇编语言写程序时，常常会创建出一些有用的汇编函数的集合，这些函数就类似于汇编运行期间库中的函数。由于汇编的模块性，很可能把这些函数分开放在多个编译过的源文件中。在某一指定程序中只使用整个函数集合中的一个子集。去具体指出你在用哪个文件变得很烦琐。如果把所有的源文件都包含进来，程序又变得太长而不易使用。

目标模块库解决了管理汇编函数集合的问题。当在程序中连入库时，连接器就浏览库并自动地选择出当前程序所需的那些模块。此外，库所用的磁盘空间比目标模块文件集要少，特别是对每个目标文件都很小的情况。库也可加速连接器的连接，因为它只打开一个文件而不是为每个目标模块都打开一个文件。

B.3.2 TLIB 命令行的组成

为获得 TLIB 的用法概要，只要在 DOS 提示符下打入 TLIB。

TLIB 命令行采取如下的通用形式，在尖括号(<>)中列出的项是可选的：

tlib libname </c> </E> <operations> <,listfile>

本节将给出每个命令行成分的概要，下面提供使用 TLIB 的详细情况。关于如何使用 TLIB 的例子见"例子"小节。

成　份	描　述
tlib	调用 TLIB 的命令名
libname	要创建或管理的库的 DOS 路径名。每个 TLIB 命令都必须给出一 libname。匹配字符不允许使用。如果不提供扩展名，TLIB 就假定是.LIB。我们建议最好使用.LIB 扩展名，因为 TASM 的目标生成工具需要靠.LIB 扩展名来识别库文件。注意，如果命名库不存在且有加操作，TLIB 将创建库。
/c	大小写标志。本选择项不常使用，见"改进的操作：/C 选择项"小节中的详细解释。
/E	创建扩展词典，见"创建扩展词典：/E 选择项"中的详细解释。
operations	TLIB 执行的操作表。操作可按任何次序出现。如果只想要确定库的内容，就根本不用给出任何操作。
listfile	要列出库内容的文件名。在 listfile 名(如给出的话)前必须加一逗号。如果不给出文件名，就不产生列表。列表包括每个模块的字母表，其后是该模块所定义的每个公用符号的字母表。列表文件的缺省扩展名为.LST。可通过把 listfile 用作 CON 来定向列表到屏幕，通过使用 PRN 把列表定向到打印机。

B.3.2　操作表(Operations)

操作表描述了你要 TLIB 做的动作。它由给定的一系列相继的操作组成。每个操作是由一个或两个字符的动作符号后面跟着一个文件或模块名组成。空白可用在动作符号、文件、模块名周围，但不能出现于某两字符的动作和某名字的中间。

你可按需要在命令行上放置多个操作，一直到 DOS 所要求的一行 127 个字符的限制。操作的次序是不重要的。TLIB 总是按指定的次序来使用操作：

- 所有提取操作首先做。
- 然后是所有的消除操作。
- 所有的填加操作最后做。

替换一个模块可看作是先消除它，然后再填加替换模块。

文件名和模块名

当 TLIB 往库中填加一目标模块文件时，该文件就简称为模块。TLIB 可以这样得到模块名：取给定的文件名，去掉它的驱动器、路径和扩展名信息。(一般情况下，并不给出驱动器、路径和扩展名)。

注意，TLIB 总是假定合理的缺省。例如，从当前目录填加具有.OBJ 扩展名的模块，你只需提供模块名，而不提供路径和.OBJ 扩展名。

在文件或模块名中不许使用匹配符号。

TLIB 的各种操作

TLIB 可识别三种动作符号(-，+，*)，你可单独使用或组合成对来使用，这样总共有五

种不同的操作。对使用一对字符的操作，字符的次序是不重要的。动作符号和它们是做什么的列表如下：

动作符号	名字	描述
+	填加	TLIB 把命名文件填加到库中。如果文件不提供扩展名，TLIB 就假定扩展名为.OBJ。如果文件是库本身(带.LIB 扩展名)，该操作就把命名库中的所有模块都填加到目标库。如果被填加的模块已在库中，TLIB 就显示一个信息，而不填加新的模块。
-	消除	TLIB 从库中消除命名模块。如果在库中无此模块，TLIB 就显示一个信息。
	提取	TLIB 通过拷贝库中相应模块到一文件中的办法来创建命中文件。如果在库中不存在相应模块，TLIB 就显示一个信息而不创建文件。如果命名文件已存在，就被重写。
-+	替换	TLIB 用相应的文件替换命名模块。这是消除操作后再做+-填加操作的省写。
-*	提取并	TLIB 把命名模块拷贝到相应文件中去，然后把它从库中消除这是提取操作后再做消除操作的省写。

消除操作只需一个模块名，TLIB 允许你打入带有驱动器和扩展名的全路径名。但除了模块名外其他名字都被忽略。

对库中的模块进行改名是不可能的。要对一模块改名，你必须从库中提取出它，再把它从库中消除，然后就对刚创建的文件改名，最后再把它填加到库中。

创建库

要创建库，你只需把模块填加到不存在的库中去。

B.3.3 使用应答文件

当你要做很多操作时，或者当你发现自己不断地重复使用某类操作时，你就可能想要开始使用应答文件了。应答文件只是 ASCII 文本文件，它含有 TLIB 命令行的部分或全部。使用应答文件，可以创建比一 DOS 命令行大的 TLIB 命令。

要使用应答文件 pahname，你可在 TLIB 命令行上任意处指定@<pahtname>。

- 一应答文件可由多行文本组成，在每行末尾你可使用"与"字符(&)来表明延续至下一行。
- 在应答文件中，不需放入全部 TLIB 命令，应答文件可用作 TLIB 命令行的一部分，在命令行上打入其余部分。
- 可在单个 TLIB 命令行中使用多个应答文件。

见"例子"小节中的样本应答文件及 TLIB 命令行使用应答文件的例子。

B.3.4 改进的操作：/c 选择项

当往库中填加一模块时，TLIB 会维护在库的模块中所定义的全部公用符号。在库中的所有符号必须是不同的。如果想往库中填加一个会引起重复符号的模块，TLIB 将显示一信息而并不填加新模块。

在正常情况下，当 TLIB 检查库中的重复符号时，大写字母和小写字母不被看作是不同的。例如，符号 lookup 和 LOOKUP 被看作是重复的。由于汇编把大小写字母看作是不同的，

你需用/c 选择项来往库中加一模块，它把库中已有的只是大小写不一样的符号看作是不同的。/c 选择项强制 TLIB 接受一个模块，该模块中符号只与库中已有符号的大小写不同。

不带/c 选择项，TLIB 就拒绝接受只是大小写不同的符号，特别是由于汇编是大小写有关的语言。这看起来是很奇怪的。其原因在于一些连接器不区别库中只是大小写不同的符号。

TLINK 区别大小写符号并没有问题，它会正确地接受库中含有的只是大小写不同的符号。只要只把库与 TLINK 一起使用，使用 TLIB 的/C 选择项就没问题了。

但是，如果想把库与其他连接器来一起使用(或允许其他人使用另的连接器)，为自保护起见，应该用/C 选择项。

B.3.5 例子

下面的一些简单例子可演示用 TLIB 能做什么。

- 创建一个命名为 MYLIB.LIB 库，它带有模块 X.OBJ，Y.OBJ 和 Z.OBJ，可打入

```
tlib  mylib  +x+y+z
```

- 创建如 1 中的库并得到一个列表，可打入

```
tlib  mylib  +x +y +z, mylib.lst
```

- 获得一现有库 CS.LIB 的列表，可打入

```
tlib  cs, cs.lst
```

- 用一新拷贝替换 X.OBJ 模块，加入 A.OBJ，删除 Z.OBJ，可打入

```
tlib  mylib  -+x +a -z
```

- 从 MYLIB.LIB 库中提取 Y.OBJ 模块，并获得一列表，可打入

```
tlib  mylib  *y, mylib.lst
```

- 使用应答文件来创建带有 A.OBJ，B.OBJ，...，G.OBJ 的新库：

首先创建一应答文本文件 ALPHA.RSP 如下

```
+a.obj+b.obj+c.obj &
+d.obj+e.obj+f.obj &
+g.obj
```

然后使用 TLIB 命令

```
tlib alpha @alpha,alpha.lst
```

B.3.6 创建一扩展词典：/E 选择项

为加速对大型文件的连接，你可让 Turbo Assembler 创建一扩展词典并把它加到库文件中。本词典以十分紧凑的形式包含了标准库词典中所没有的信息。本信息可允许 TLINK 快速处理库文件，特别是在这些文件位于软盘或硬盘上时。Turbo Assembler 分配在磁盘上的所有库都含有扩展词典。

为正被修改的库创建一扩展词典，就需要你在调用 TLIB 来填加、消除和替换库中的模块时使用/E 选择项。为不想更改的现有库创建一扩展词典，就要求使用/E 选择项。并要求 TLIB 删除库中不存在的模块。如果指定的模块没有在库中找到，TLIB 就显示一警告，但它

也为所指定的库创建扩展词典。例如，可打入：

```
tlib /E mylib -bogus
```

B.4 GREP：一种文件查找实用程序

GREP 是一强有力的查找实用程序，它可快速查找几个文件中的文本。

GREP 遵循的一般命令行语法如下：

GREP ＜选择项＞ 查找串 文件说明 ＜文件说明 文件说明... 文件说明＞

例如，如果想要看看在你哪个源文件中调用了 Setupmodem 函数，可用 GREP 来查找目录下所有. ASM 文件的内容以寻找 Setupmodem 串，如下所示：

```
grep setupmoidem *.asm
```

B.4.1 GREP 选择项

在命令行中，选择项是以连字符号(-)打头的若干个单字符。每个单独的字符都是一个可打开或关闭的开关：在该字符后打入加号(+)就打开该选择项；打入连字符号(-)就关闭掉该选择项。

缺省状态为打开(隐含为+)，例如，-r 的意义与-r+相同。可把多个选择项一个个地列出(如-i -d -l)，也可把它们组合起来(如-ild 或-il -d 等)，它们对 GREP 来说是相同的。

下面将列出 GREP 所用的选择项字符及它们的意义：

-c 只计数 只打印匹配行的计数。对每一至少含有一匹配行的文件，GREP 就打印出文件名和匹配行数目的计数。并不打印匹配行。

-d 目录 对命令行上所指定的每个文件说明，GREP 将查找与文件说明相区配的所有文件，查找的目录是所指定的目录及所指定目录下面的所有子目录。如果所给的文件说明不带路径，GREP 就假定文件是在当前目录下。

-i 忽略大小写 GREP 忽略大小写的差别。GREP 在所有情况下都把 a～z 的所有字母看作与 A～Z 的相应字母是相同的。

-l 匹配文件列表 只打印出相匹配的每一个文件名字。在 GREP 找到一个匹配后，它就打印出文件名并接着处理下一个文件。

-n 数 GREP 打印的每个匹配行前加上其行数。

-o UNIX 输出格式 改变匹配行的输出形式以方便地支持 UNIX 型的命令行管道。所有输出行前都加上含有匹配行的文件的名字。

-r 正则表达式查找 由查找串定义的文本被看作是正则表达式而不是文字串。

-u 更新选择项 GREP 把命令行所给的选择项与其缺省选择项组合在一起写到 GREP. COM 文件来作为新的缺省。(换言之，GREP 在自配置)。本选择项允许你按自己的嗜好设置缺省选择项。

-v 不匹配 只打印出不匹配的行。只有那些不含有匹配中的行被看作是不匹配的行。

-w 词查找 与正则表达式相匹配的文本被看作是一个匹配的条件是仅当其紧前字符和紧后字符不是词的一部分。缺省的词字符集包括 A～Z，9～0 和下划线( )。本选择项的可选形式可允许指定合法的词字符。其形式是-w[集合]，集合是任何有效正则表达式集合

的定义。如果字母字符用来定义本集合，集合就自动地被定义成含有集合中的每个字符的大小写值，这与它是如何打入的无关，即便查找是大小写有关的。如果把-w 选择项与-u 选择项组合在一起使用，新的合法字符集就被保留作缺省集。

-z　冗长　GREP 打印出所查到的每个文件名。每个文件名前是其行数。每个文件中匹配行的行数也给出，即便该计数为 0。

B.4.1.1　优先级次序

GREP 的每个选择项都是一个开关：它的状态反映了上次触发它的方式。在任何给定的时间，每个选择项只能打开或关闭。给定选择项在命令行上的每次出现都覆盖掉其原来的定义。例如，

```
grep -r -i- -d -i -r-  main(  my*.asm
```

在命令行上给出，则 GREP 在运行时，-d 选择项打开，-i 选择项打开，-r 选择项关闭。

可用-u 选择项在 GREP.COM 中按嗜好装入对每个选择项的缺省设置。例如，如果总想让 GREP 做冗长查找(-z 打开)，可用如下命令来安装：

```
grep -u -z
```

B.4.2　查找串

查找串的值定义了 GREP 的查找模式。查找串既可为正则表达式也可为文字串。在正则表达式中，某些字符具有特殊意义：它们是用于查找的操作符。在文字串中没有操作符，每个字符都被看作是一个文字。

可把查找串用引号引起，以免把空格和制表符看成是分隔符。匹配不越过行边界(即一匹配必须包含于一单个行中)。

表达式或为单个字符或为由方括号([ ])括起来的多个字符。把多个表达式合在一起构成了正则表达式。

B.4.2.1　正则表达式中的操作符

当使用-r 选择项，查找串就被看作是正则表达式(不是文字表达式)，下面的字符就有了特殊意义：

^　在表达式起始处的声调符号匹配一行的起始。

$　在表达式结尾处的美元符号匹配一行的结尾。

.　句点可与任何字符匹配。

*　一表达式后面跟着星号匹配符可匹配该表达式的零个或多个出现。例如，对于 fo*，* 号作用于表达式 o)，它可匹配 f，fo，foo 等。但它并不匹配 fa。

+　一表达式后面跟着加号(+)可匹配该表达式的一个以上的出现：fo+可匹配 fo，foo 等，但并不匹配 f。

用方括号括起的串可与该串中任何字符匹配，但不能是其他的。如果串中的第一个字符是声调符号(^)，表达式就与除了表达式中字符外的任何字符匹配。例如，[XYXZ]匹配 x，y，或 z，而[^xyz]可匹配 a，和 b，但不能是 x，y 或 z。可用把两字符用连字号(-)隔开的办法来指定一系列字符。这些可组合成表达式(如[a-b d-z?]可匹配?和除了 c 的任何小写字母)。

反斜杠换码字符让 GREP 查找其后的文字字符。例如，\. 匹配一个句点而不是“任何字符”了。

注意:前四个字符($,.,*和+)当用在中括号内时就没有特殊意义了。此外,声调字符^如果紧接着中括号就被特殊看待,不再被当做与句子起始匹配的字符了。

在上面表中没提到的任何普通字符都与相应字符匹配。(>匹配>,#匹配#等。)

B.4.3 文件说明

GREP 命令行的第三项是文件说明,它告诉 GREP 去查找哪些文件。文件说明可为显式文件名,或为带有 DOS 中?和*匹配符的一般文件名。此外,可打入路径(驱动器和目录信息)作为文件说明的一部分。如果给出没有路径的文件说明,GREP 只查找当前目录。

B.4.4 带说明的例子

在下面的例子中,假定所有 GREP 选择项的缺省值为关闭。

例 1

命令行: grep start: *.asm

匹配: start:

restart:

不匹配: restarted:

clock Start:

查找的文件: 当前目录下的 *.ASM。

说明: 在缺省情况下,大小写是有意义的。

例 2

命令行: grep -r [^a~z]main\ *(*.asm

匹配: main(i:integer)

main(i,j:integer)

if(main())halt;

不匹配: mymain()

MAIN(i,integer);

查找的文件:当前目录下的 *.ASM

说明: GREP 查找词 main,main 前不带有小写字母([^a~z]),后面是零个以上空格(\ *),然后是一左括号。

由于空格和制表符通常被看作是命令行分隔符,如果想要把它们当作正则表达式的一部分就必须把它们用引号引起。在这种情况下,main 后的空格就用反斜杠换码字符引起。也可以用把空格放在双引号间的办法来实现,如:

[^a~z]main" " *

例 3

命令行: grep -ri [a~c]:\\data\.fil *.asm *inc

匹配: A:\data.fil

B:\Data.FIL

C:\Data.Fil

不匹配：　d:\data.fil

a:data.fil

查找的文件：　当前目录下的 *.ASM 和 *.INC。

说明：　因为反斜杠和句点字符(\和.)通常具有特殊的意义，如果想查找它们，必须在其紧前面放上反斜杠换码字符来把它们引起。

例 4

命令行：　grep -ri [^a～z]word[^a～z] *.doc

匹配：　every new word must be on a new line.

MY WORD!

word -smallest unit of speech。

In the beginning there was the WORD，and the WORD

不匹配：　Each file has at least 2000 words。

He misspells toward as toword。

查找的文件：　当前目录前 *.DOC

说明：　本格式基本上定义了如何查找一给定词。

例 5

命令行：　grep -iw word *.doc

匹配：　every new word must be on a new line However，

MY WORD!

word：smallest unit of speech which conveys meaning。

In the beginning there was the WORD，and the WORD

不匹配：　each document Contains at least 2000 words

He seems to Continually was spell "toward" as "toword."

查找的文件：　当前目录下的 *.doc。

说明：　本格式定义了基本的"word"查找。

例 6

命令行：　grep "search String with spaces" *.doc *.asm

a:\work\myfile. *

匹配：　This is a Search string with spaces in it.

不匹配：　THIS IS A SEARCH STRING WITH SPACE IN IT.

This is a search string with many spcaes in it

查找的文件：　*.DOC 和 *.ASM 都在当前目录下，而 MYFILE. * 在驱动器 A:中\WORK 的目录下。

说明：　这是如何查找带有空格的串的例子。

例 7

命令行：　grep -rd "[,.:? \"]" $ *.doc

匹配：　He said hi to me.

where are you going?

Happening in anticipation of a uique situation，

Examples inclued the following:
"Many men Smoke,but for man chu."

不匹配：　He Sold "Hi"to me
where are you going? I' m headed to the beach this

查找的文件：　当前驱动器根目录及其所有子目录下的 *.DOC。

说明：　本例在一行结尾处查找,、.、:、?和"字符。注意,在方括号中的双引号前加上反斜杆换码字符是为了把它作是通常字符而不是串的结束引号。也应注意 $ 是如何出现在引号串外边的。这表明正则表达式是怎样连接成一个更长的表达式的。

例 8

命令行：　grep -ild "the " *.doc
或 grep -i -l -d "the" *.doc
或 grep -il -d "the " *.doc

匹配：　Anymay, this is the time we have
do you thinic? the main reason we are

不匹配：　He said "Hi" to me just when I
Where are you going? I' ll bet you re headed to

查找的文件：　当前驱动器根目录及其所有子目录下的 *.DOC。

说明：　本例忽略大小写,只打印出至少有一个匹配的文件名。三个例子表明了指定多个选择项的不同方式。

B.5　OBJXREF:目标模块交叉引用实用程序

OBJXREF 是一种实用程序,它可用来检查一系列目标文件和库文件,并生成有关这些文件内容的报告。一类报告是列出公用名的定义和对它们的引用。另一类报告是列出目标模块定义的段的长度。

有两类公用名:全局变量和函数名。在"OBJXREF 报告样本"小节中的 TEST1.ASM 和 TEST2.ASM 文件表明了公用名的定义和对它们的外部引用。

目标模块是由 TC,TCC 或 TASM 生成的目标(.OBJ)文件。库(.LIB)文件中含有多个目标模块。由 TASM 产生的目标模块与其相应的.ASM 源文件具有同样的名字,除非在命令行上特别指定了一个不同的输出文件名。

B.5.1　OBJXREF 命令行

OBJXREF 命令行的组成是:词 OBJXREF,其后为一系列命令行选择项和一系列目标和库文件名,其间用空格或制表符隔开。其语法形式如下:

OBJXREF <选择项> 文件名<文件名...>

命令行选择项用来确定产生 OBJXREF 报告的种类和 OBJXREF 提供内容的详细程度。这将在下一小节"命令行选择项"中详述。

每个选择项由一斜杆(/)起始,后面跟着一或二字符的选择项名。

目标文件和库文件既可由命令行指定，也可在应答文件中指定。在命令行上，文件名间用空格或制表符隔开。所有指定为.OBJ 文件的目标模块都包含在报告中。类似于 TLINK、OBJXREF 只包含.LIB 文件中的一些模块，这些模块中或者含有.OBJ 文件所引用的公用名，或者含有从前面.LIB 文件中包含进来的模块所引用的公用名。

作为一条一般规则，应该列出程序正确连接所需的所有.OBJ 文件和.LIB 文件，包括各种库。

文件名可含有驱动器和目录路径。DOS 匹配符？和＊可用来标明多个文件。文件名可指.OBJ 文件或.LIB 库文件(如果没给出扩展名。就假定扩展名为.OBJ)。

选择项和文件名在命令行中的出现次序是任意的。

OBJXREF 报告被写到 DOS 的标准输出。缺省值为屏幕。也可用 DOS 重定向符(>)把报告送到打印机(如用>LPT1:)或一文件(如用>列表文件)。

B.5.1.1 命令行选择项

OBJXREF 命令行选择项可分成两类：控制选择项和报告选择项。

控制选择项

控制选择项可改变 OBJXREF 的缺省行为(缺省的行为是这些选择项均不允许使用)。

/I　在公用名中忽略大小写的差别：如果你使用 TLINK 时不带/C 选择项(该选择项使大小写的差别有意义)，可用本选择项。

/F　包含整个库。指定的.LIB 文件中的所有目标模块均被包含，即使它们并不含有正被 OBJXREF 处理的目标模块所引用的公用名。这提供了库文件全部内容的信息(见"OBJXREF 例子"小节中的例 4)。

/V　冗长输出：列出读入的文件名且显示整个的公用名，模块，段和类。

/Z　包含长度为零的段定义：目标模块可定义没有任何空间的段。列出这些长度为零的段定义通常使模块长度报告更难以使用，但如果你要删除一个段的所有定义，这是很有价值的。

报告选择项

报告选择项控制产生何类报告及 OBJXREF 提供内容的详细程序。

/RC　按类报告：按段的类次序给出模块长度。。

/RM　按模块报告：按定义模块的次序给出公用名。

/RP　按公用名报告：顺序给出公用名并带有定义模块名。

/RR　按引用报告：按名字次序给出公用名定义和引用。(这是没指定报告选择项的缺省方式)。

/RS　按模块长度报告：按段名次序给出模块长度。

/RU　按未引用符号名报告：按定义模块的次序给出未引用符号名。

/RV　冗长报告：OBJXRET 产生每类报告。

/RX　按外部引用报告：按引用模块名次序给出外部引用。

B.5.2 应答文件

DOS 所限制的命令行最多可含有 128 个字符。如果你的选择项和文件名表超过此限制，就必须把文件名放进应答文件中。

应答文件是可用文本编辑器生成的文本文件。由于可能已为别的 Turbo Assembler 程序准备了你自己程序中的一系列文件，OBJXREF 可识别出几类应答文件。

从命令行调用应答文件可使用下述选择项之一。在选择项后的应答文件名不能用空格隔开(/L 应答文件而不是/L 应答文件)。

在命令行上可指定多个应答文件，比其它的.OBJ 和.LIB 文件名可在它们的前边或后面。

B.5.2.1 自由形式的应答文件

可用文本编辑器来创建自由形式的应答文件，只列出生成.EXE 文件所需的所有.OBJ 和.LIB 文件名。

为使 OBJXREF 来使用自由形式的应答文件，可在命令行上打入每个文件名，在其前面加入 at 标志(@)，命令行不同表目间用空格或制表符隔开：

@文件名 @文件名 ...

注意：在应答文件中列出的文件名若不带扩展名就假定为.OBJ 文件。

B.5.2.2 连接器应答文件

OBJXREF 也可使用 Turbo Assembler 应答文件格式的文件。命令行调用的连接器应答文件前须加/L：

/L 文件名

对于如何使用这类应答文件，可参阅"B.5.4 使用 OBJXREF 的例子"小节中的例 2。

B.5.2.3 /D 命令

如果想要 OBJXREF 不在当前目录下查找.OBJ 文件，就可把目录名包含在命令行上，前缀以/D：

C:OBJXREF/D 目录 1[;目录 2][;目录 3]

或 C:>OBJXREF/D 目录 1[/D 目录 2][/D 目录 3]

OBJXREF 将按指定的次序在每个目录下查找所有的目标文件和库文件。如果不用/D 选择项，就只有在当前目录下查找。但如果用了/D 选择项就不查找当前目录了，除非它包含在目录表中。例如，首先在 BORLAND 目录下查找文件，然后再在当前目录下查找，可打入：

C:>OBJXREF/D borland;

如果指定多个查找目录，并找到与文件说明相匹配的文件，OBJXREF 就把新文件作为交叉引用的一部分。如果文件说明中含有匹配符，OBJXREF 只继续在别的目录下查找同样的文件说明。

B.5.2.4 /O 命令

/O 选择项允许指定输出文件，OBJXREF 将把产生的报告发送到新文件中。它具有如下形式的语法：

C>OBJXREF myfile.obj /RU /O 文件名.扩展名

在缺省情况下，所有输出都送到屏幕。

B.5.2.5 /N 命令

可限制 OBJXREF 报告的模块、段、类和公用名，只要在命令行上打入合适的名字，并前缀以/N 命令。例如，

OBJXREF <文件表> /RM /N 测试

将告诉OBJXREF 产生一报告，该报告只列出命名为测试的模块的信息。

B.5.3 OBJXREF 报告样本

假设在 Turbo Assembler 的目录下拥有两个源文件，希望产生由它们汇编出的目标文件的 OBJXREF 报告。源文件为 TEST1.ASM 和 TEST2.ASM，内容如下：

```
; TEST1.ASM
        .MODEL small
        STACK 200h
        EXTRN GOOODBYTE:BYTE                ;引用 Goodbye
        EXTRN SAYHELLO:NEAR                 ;引用 SayHello
        PUBLIC HELLO                        ;使 Hello 成为公用的
        PUBLIC  NOTUSED                     ;使 NotUsed 成为公用的
        PUBLIC  NOTUSED                     ;使 NotUsed 成为公用的
.DATA
HELLO DB 'Hello',10,13,'$'                  ;定义 Hello
NOTUSED DW ?
HIDDEN DW ?
.CODE
SAYBYE PROC NEAR                            ;定义 Saybye
        mov dx,OFFSET GOODBYE
        mov ah, 9
        int 21h
        ret
SAYBYE ENDP
START PROC NEAR                             ;定义 start
        mov ax, data
        mov ds, ax
        Call SAYHELLO                       ;引用 SayHello
        Call SAYBYE                         ;引用 SayBye
EXIT:
        mov ax,04cooh
        int 21h
START ENDP
      END START

;TEST2.ASM
    .MODEL small
      EXTRN HELLO:BYTE                      ;引用 Hello
      PUBLIC GOODBYTE                       ;使 GOODBYE 成为公用的
      PUBLIC SAYHELLO                       ;使 SayHello 成为公用的
.DATA
GOODBYE DB 'Goodbye',10,13,'$'              ;定义 Goodbye
```

```
.COPE
SAYHELLO PROC NEAR                    ;定义 SayHello
        mov dx,OFFSET HELLO           ;引用 Hello
        mov ah,9
        int 21h
        ret
SAYHELLO ENDP
        END
```

由这两个文件汇编出的目标模块为 TEST1.OBJ 和 TEST2.OBJ。你可告诉 OBJXREF 为这些.OBJ 文件产生何类报告,办法是在命令行上打入文件名,在后面跟上/R 和指明报告类型的字母。

注意:下列例子只显示输出的一部分。

B.5.3.1 按公用名报告(/RP)

按公用名报告指列出要报告的目标模块中定义的每个公用名,其后定义相应公用名的模块名。

如果在命令行上打入如下内容:

```
OBJXREF  /RP test1 test2
```

OBJXREF 将产生如下形式的报告:

```
                符号        定义于
               GOODBYE     TEST2
               HELLO       TEST1
               NOTUSED     TEST1
               SAYHELLO    TEST2
```

B.5.3.2 按模块报告(/RM)

按模块报告指列出要报告的每个目标模块,然后是在相应模块中定义的公用名。

如果在命令行上打入下述内容:

```
OBJXREF  /RM  test1 test2
```

OBJXREF 将产生如下形式的报告:

```
Module: TEST1 defines the following symbols:
 GOODBYE
 SAYHELLO
```

B.5.3.3 按引用报告(/RR)(缺省方式)

按引用报告指列出每个公用名,并在同一行用圆括号括起相应公用名定义所在的模块。引用此公用名的模块从左边缩进列于下一行。

如果你在命令行上打入下述内容:

```
OBJXREF  /RR test1 test2
```

OBJXREF 将产生如下形式的报告:

```
GOODBYE   (TEST2)
              TEST1
HELLO     (TEST1)
```

```
            TEST2
  NOTUSED   (TEST1)
  SAYHELLO  (TEST2)
            TEST1
```

B.5.3.4　按外部引用报告(/RX)

按外部引用报告指列出每个模块及其所含的外部引用。

如果在命令行打入下述内容：

```
OBJXREF  /RX  test1  test2  CS.LIB
```

OBJXREF 将产生如下形式的报告：

```
module:TEST1 references the following symbols:
  GOODBYE
  SAYHELLO
module: TEST2 references the following:
  HELLO
```

B.5.3.5　按模块长度报告(/RS)

按模块长度报告指列出段名，并随后列出定义相应段的模块。长度按字节计数，以十进制和十六进制两种形式给出。如果段中所定义的任何符号均未赋初始值，则出现字 uninitialized。在.ASM 文件中定义于绝对地址的段在段长度的左边标上 Abs。

如果你在命令行打入下述内容：

```
OBJXREF  /RS  test1 test2
```

OBJXREF 将产生如下形式的报告：

```
;Module sizes by Segment
STACK
          512 (00200h)   TEST1,uninitialized
          512 (00200h)   total
-DATA
          12  (0000ch)   TEST1
          10  (0000Ah)   TEST2
          22  (00016h)   total
-TEXT
          24  (00018h)   TEST1
           8  (00008h)   TEST2
          32  (00020h)   tltal
```

B.5.3.6　按类报告 (/RC)

按类报告指列出按段所属类给出的段长度。CODE 类含有指令，DATA 类含有初始化数据，而 BSS 类含有非初始化数据。不属上述类的段以"No class type"形式列出。

如果在命令行上打入下述内容：

```
OBJXREF  /RC  test1 test2
```

OBJXREF 将产生如下形式的报告：

```
; Module sizes by class
  CODE
        24          (0018h)    TEST1
        8           (00008h)   TEST2
        32          (00020h)   total
  DATA
        12          (0000ch)   TEST1
        10          (0000Ah)   TEST2
        22          (00016h)   total
  STACK
        512 (00200h)   TEST1, uninitial;real
        512 (00200h)   total
```

B.5.3.7 按未引用符号名报告 (/RV)

按未引用符号报告指列出定义了未被其他模块引用的公用名的模块。这样的符号包括下述两种情况：

- 只在定义模块中被引用而无需定义成公用符号(对这样情况，如果模块为C模块，应在定义前加关键字static；如果模块为TASM模块，只需删除相应公用定义)。
- 从未使用过(因此，应删除它以节省代码或数据空间)。如果在命令行上打入下述内容：

```
OBJXREF   /RU test1 test2
```

OBJXREF将产生如下形式的报告：

```
Module; TEST1 defines the following unreferenced symbols:
  NOUSED
```

B.5.3.8 冗长报告 (/RV)

如果在命令行打入/RV，产生上述各类报告。

B.5.4 使用OBJXREF的例子

在下面的例子中，假定使用的文件是在缺省驱动器的当前目录上，而库文件是在\LIB目录下。

例1 C>OBJXREF test1 test2 \lib\io.lib

除了TEST1.OBJ和TEST2.OBJ外，还指定了库文件/LIB/IO.LIB。由于没指定报告类型，所得报告就是缺省方式下的按引用报告，即列出公用名和引用它们的模块。

例2 C>OBJXREF /RV /Ltest1.arf

TLINK应答文件TEST1.ARF中含有与例1命令行同样的内容。/RV选择项指定产生一个冗长报告。

TEST1.ARF内容如下：

```
test1 test2
test1.exe
test1.map
```

\lib\io

例 3　C>OBJXREF　/F /RV \lib\io.lib

本例将产生库文件 IO.LIB 中所有模块的报告；OBJXREF 可产生有用的报告，即使所指定的文件并不构成一个完整程序。/F 使 iO.LIB 文件中的所有模块包含在报告中。

B.5.5　OBJXREF 出错信息和警告

OBJXREF 可产生两类诊断信息：出错信息和警告。

B.5.5.1　出错信息

Out of memory

OBJXREF 在 RAM 内存执行它的交叉引用，可能会发生内存不够用，即使是 TLINK 可以对相同的文件表进行成功地连接。当发生这种情况时，OBJXREF 就退出了。删除内存驻留程序以获得更多空间或增加 RAM。

B.5.5.2　警　　告

WARNING：Unable to open input file rrrr　输出文件 rrrr 可能不存在或未打开。OBJXREF 继续处理下一个文件。

WARNING Unknown option -oooo　OBJXREF 不识别选择项 oooo。OBJXREF 将忽略此选择项。

WARNING：Unresolved symbol nnnn in module mmmm　在模块 mmmm 中引用的公用名 nnnn 没有在指定的.OBJ 或.LIB 文件中定义。OBJXREF 将在任何产生新符号的报告中把它标记为引用但未定义。

WARNING：Invalid file specification ffff　文件名 ffff 的某部分是非法的。OBJXREF 继续处理下一个文件。

WARNING：No files matching ffff　在令行上或应答文件中命名为 ffff 的文件不存在或未被打开。OBJXREF 将跳到下一个文件。

WARNING：Symbol undefined in mmmm1 duplicated in mmmmm2　公用名 nnnn 整模块 mmmm1 和 mmmm2 中均有定义。OBJXREF 将忽略第二个定义。

B.6　TCREF：源模块交叉引用实用程序

TCREF 是设计用来产生两类报告：定义和使用所有全局符号的交叉引用表和模块及其使用符号表

TCREF 把一组 TASM 产生的.XRF 文件作为输入。这些文件含有每个模块的交叉引用信息。从这些输入文件中，产生含有一个或多个 ASCII 文本形式报告的单个.REF 文件。命令格式如下：

TCREF ＜由"+"号隔开的 XRF 文件＞ "，"

　　＜REF 文件名＞　＜开关＞

例如，以 F001.XRF，F002.XRF 和 F003.CXRF 为输入文件，产生 F00.REF 可采取如下形式：

TCREF f001 + f002 + f003, f00

B.6.1 应答文件

TCREF 也可接受 ASCII 文件作为命令串。只要在文件名前加一 at 标志(@)就可在命令串中包含文件。例如,

```
TCREF   @dof00
```

在 DOF00 中含有

```
f001 + f002 + f003,f00
```

这就与前面的例子作用相同。

B.6.2 与 TLINK 的兼容

TCREF 接受 TLINK 接受的命令串,TCREF 忽略任何不相关的开关和字段,如库文件和 MAP 文件或只执行连接功能的开关。类似地,如果不存在.XRF 文件,TCREF 只是忽略它。

注意! 当使用 TLINK 应答文件时,不能明确地指定扩展名,因为如果这样做就会覆盖掉 TCREF 的内部缺省,可能导致灾难性的结果。例如,如果在应答文件中输入

```
f001+f002+f003,f00.exe
```

就不能让 TCREF 不加修改地使用本文件,这是因为产生的.REF 文件被命名为 F00.EXE,这样就重写你的程序了。

B.6.2.1 开关

TCREF 接受 TLINK 中的所有开关,但这些开关中的大多数被废弃不用。TCREF 只使用如下这些开关:

- /C 使 GOLBAL 报告成为大小写有关的。
- /Y 为所有指定的模块产生 LOCAL 报告。
- /P# 将报告页的长度设置为#行。
- /W#将报告页的宽度设置为#列。

TCREF 很注意产生符号的语义意义。当把具有相同名但不同意义的符号放在一块时,交叉引用信息就不起作用了。因此 TCREF 在产生其报告时考虑符号的应用范围。交叉引用信息总是列出相应符号所在的源文件及在源文件中的行数。

B.6.2.2 全局(或连接器级)报告

TCREF 的全局报告指列出对连接器可见的全局符号的交叉引用信息。如果要想产生大小写有关的报告可用/C 开关。

在本报告中,全局符号从左边的列开始以字母序出现。源文件中的引用列于右边的列上。带#号的地方表明定义是出现在新行。

下面是符号打印输出的例子:

```
Global  symbols Cref   #=definition
BAR                TEST.ASM: 1 3 6 9 12 15 18 + 21 23 29
                                  #TEST2.ASM:2 4 6   #8
```

那么,这告诉了你什么? TEST2.ASM 前面的#表明 BAR 是在该模块中某处定义的。对每个源文件,列出了引用相应符号的行。列出的行表可多于一行。如例子中 TEST.ASM

中的行列表，最后，在 8 前面的 # 号表明 BAR 的定义是在 TEST2.ASM 中的第 8 行上。

B.6.2.3　局部(或模块级)报告

如果在命令行上指定了/R，就为每个模块产生一个局部报告。它包含有在相应模块中使用的所有符号，并以字母顺序列出。/C 开关对这些报告没什么作用，因为在汇编期间就确定了合适的大小写相关性。类似于全局报告，源文件中的引用组织在右边的列中。打印输出的样本如下所示：

```
Module TEST.ASM Symbols   Cref #=definition
UGH                       TEST.ASM 1 3 6 9 12 15 18+
                                   21 23 29
                          #UGH.INC: #2
```

参 考 文 献

1. 李振格,Turbo Pascal 大全,航空工业出版社,1991
2. Borland,Turbo Assembler 用户手册,Borland,1988
3. Borland, Turbo Assembler 参考手册,Borland,1988
4. Borland, Turbo Debugger 用户手册,Borland,1988
5. Borland, Turbo Prolog 用户手册,Borland, 1988
6. Borland, Turbo Prolog 参考手册,Borland, 1988

[General Information]
书名=微机高级语言与汇编语言 接口技术和实例
作者=李振格
页数=338
SS号=10203843
出版日期=1994年5月第1版