



TURBO C2.0

高级编程与 剖析

編著 朱茂華

成都科技大学出版社

384130

Turbo C 高级编程与剖析

朱茂华



成都科技大学出版社
一九九四年八月

[川]新登字015号

责任编辑:蔡源众 王云川

封面设计:陆春熙

JS205/3207

内容简介

本书通过对 600 多个 Turbo C 源程序进行分析,系统而又详细地介绍了 C 语法、库函数和标头文件的全部内容。对涉及的软硬件环境、编程要点有详尽的中文说明。阐释重点难点,剖析易混淆之处,帮助读者解决在用 C 编程时遇到的各种实际问题。

全书 44 章,包括集成开发环境、程序调试、文件管理、键盘与鼠标、打印机、视频函数和工具程序等。该书集 DOS、中断和程序设计技巧于一体,融进了近年来 C 语言研究的最新成果,揭示了软件测试中发现的一些鲜为人知的内容。该书还具有综合性手册的特点。

本书通俗易懂,资料丰富实用,适合软件人员、程序员、函授学院学生、计算机自学考试人员阅读和 C 爱好者自学用。可供大中专院校计算机及相关专业的师生参考。



TURBOC 高级编程与剖析

朱茂华

编、著

成都科技大学出版社出版发行

四川省新华书店 经销

中国人民解放军成都军区卫校印刷厂印刷

开本:787×1092 1/16 印张 50

1994年8月 第1版1994年8月 第1次印刷

印数:1—2000册 字数:1200千字

ISBN 7-5616-2719-x/TP·59

定 价:50.00元

导 读

C 语言由于自身所具有的特点和其强大的功能,已经在计算机世界的各个领域内广为使用。从 UNIX 系统到 DOS、Windows 系统、从信息管理系统(Management Information System)到多媒体应用(Multimedia)等,都已广泛地使用了 C 语言编程。正是由于 C 语言的高效灵活性,一些程序员才把它称作“真正的程序员语言”。然而 C 语言又是几乎所有高级语言中最难掌握的一门语言,有人甚至用“非打点十二分精神不足以深解”来形容阅读 C 语言书籍的困难性。一些作者也总是想方设法去减少读者在这方面的负担。采用较好的编排技巧就是一个明证,就这一点来说,本书也不例外。

同时我们也深刻的体会到,剖析优秀的程序实例是迅速掌握一门语言的最捷径的方法,通过源程序的阅读及调试,并基于源程序之上开发实用软件包,是国内外优秀程序设计人员的秘诀。因此作者在本书中搜集了大量具有代表性的程序例子和几个大而复杂的实用例程,让读者既能看到树木,又能看到森林,最终达到迅速掌握,迅速提高的目的。

在阅读本书之前,你应当将目录浏览一遍,了解本书的编排顺序,根据自己的需要,阅读相关的章节。如果你是刚刚开始接触 Turbo C,我们建议你按下述方法逐步深入:

1. 先阅读第一章、第二章的前两节、第三章到第十七章。其中第一章、第十六章和第十七章的前三节是必读的;

2. 在阅读时每逢涉及短的例程时最好在集成环境下试一试。在学会编辑程序时,你应当了解菜单结构和每一菜单的含义。一切不必死记硬背,开始也许要逐一过目,以后只在感到有疑问时才查。当有一些基本概念了,再把第十七章“程序调试”一节的一个程序实实在在跟踪一遍;

3. 在集成环境下,你应当重点掌握“调试表达式”的使用方法;

4. 觉得有把握了,可以在集成环境下调试大一点的例程;

5. 每当遇到问题时,可以按前面指出的查询原则按功能全书查找;

6. 现在可以再深入一步,熟悉常用的第三十一章到第三十五章;

7. 对几个工具程序即第四十章到四十四章进行了解。因为有些源程序不能在集成环境下编译连接,而应使用 TCC.EXE,所以第四十章、第四十一章应视为重点;

8. 至此,你完全可以熟悉其余各章了。

在学习中,一定会发现不少内容你可能还未碰到过。所以,你还应多阅读别人编写的程序,看看自己能不能读懂。几经练习后,便可开始解决自己的实际问题了(注意对实际问题的系统分析和相应的算法)。

对高级程序员,可以跳过前面十七章,而把重点放在后面的一些章节上。可以熟悉库函数等位置和使用方法,Turbo C 与 DOS 等关系,不同 C 版本之间的对比分析,补充高版本 C 的相关内容。

一个良好的 C 程序员还应当对常用工具软件有所了解,能综合使用多种软件效果会更好。本书在提供部分重要 DOS 命令的同时,也给出了这方面的例子。

如同学习 Turbo C 时可参考 Turbo Pascal 一样,作为综合性技术手册,本书显然也可以供学习其它计算机语言者参考。

作者简介

江苏省张家港市人,男,1945年生。高级工程师,室主任。1968年毕业于南京大学数学系计算数学专业,同年分配到兵器工业部成都华西光学电子仪器厂。当过机修钳工,后任机械动力处技术员。1981年开始在科研所使用计算机开发民用产品。目前正致力于计算机实用,普及性研究工作。

曾与他人合作,用微机开发成功国内第一个 BXSJ80~200/4.5 变焦距镜头。已开发的微机软件有,光学设计软件、照相镜头光栏机构设计软件、变焦距镜头设计专用程序、模具线切割程序, WSOP 和 DBFOP 工具软件等。

至今已在各种报刊上发表论文 30 多篇。曾编写过《电子数字计算机简介》、《BASIC 语言与扩充》、《符号函数和照相物镜头栏机构凸轮计算》等资料。1993 年编写的《高级汉字自动制表软件 OFFICE 使用技巧》由北京科海培训中心出版。

如需购卖本书或对本书提出批评及建议,请与成都电子科技大学都乐资料服务部联系。

地址:成都建设北路二段四号

邮编:610054

电话:(028)3335219

目 录

引言.....	(1)
第一章 Turbo C 的安装	(3)
1.1 安装 Turbo C 的硬件环境	(3)
1.2 Turbo C 2.0 磁盘上的内容	(3)
1.3 三种安装方法	(6)
1.4 文本阅读器 README.COM	(7)
第二章 80x86 指令和六种存储模式	(12)
2.1 寄存器	(12)
2.2 8086 地址计算	(14)
2.3 8086/80386 指令集简介	(15)
2.4 嵌入汇编	(26)
2.5 Turbo C 的六种存储模式	(29)
2.6 汇编程序的伪指令(摘要)	(34)
第三章 关键字和语句	(41)
3.1 分类	(41)
3.2 详细说明	(42)
3.3 附注	(67)
第四章 变量的存储分类与应用	(68)
4.1 分类	(68)
4.2 初始化	(69)
4.3 详细说明	(70)
4.4 局部变量和全局变量的关系	(76)
4.5 寄存器变量	(77)
4.6 嵌入汇编和寄存器变量	(77)
第五章 数据类型的转换	(78)
5.1 int 类型 and char 类型间的转换	(78)
5.2 int 类型 and enum 类型间的转换	(79)
5.3 指针之间的转换	(79)
5.4 符号扩充、转义符与算术运算转换	(81)
5.5 类型的强制转换	(90)
5.6 字符串与数值之间的转换	(91)
第六章 运算符	(92)
6.1 结合运算符 ()	(93)
6.2 下标运算符 []	(93)
6.3 分量运算符 . 和 ->	(94)
6.4 逻辑运算符 !、&& 和 	(96)
6.5 位运算符 ~、<<、>>、&、^ 和 	(99)
6.6 负值运算符 -	(105)
6.7 递增、递减运算符 ++ 和 --	(106)
6.8 指针运算符 & 和 *	(107)
6.9 强制类型转换运算符(类型名)	(108)

6.10 求字节数运算符 sizeof	(110)
6.11 算术运算符 +、-、*、/ 和 %	(110)
6.12 关系运算符 >、<、==、>=、<= 和 !	(111)
6.13 赋值运算符 = 与 op =	(111)
6.14 条件运算符 ?:	(112)
6.15 逗号运算符 ,	(114)
6.16 综合举例	(115)
第七章 数组与字符串	(116)
7.1 数组	(116)
7.1.1 一维数组	(116)
7.1.2 二维数组和多维数组	(120)
7.2 字符分类	(122)
7.2.1 分类标志常量	(123)
7.2.2 外部字符数组 ctype 的含义	(123)
7.2.3 字符分类宏	(123)
7.2.4 清字符最高位与字符大小写转换	(124)
7.3 字符串操作	(124)
7.3.1 字符串的定义	(124)
7.3.2 串操作函数	(126)
7.4 查找字符串实用程序 GREP.COM	(150)
7.4.1 作用	(150)
7.4.2 语法和帮助	(150)
7.4.3 语法说明	(151)
7.4.4 可能出现的错误信息	(161)
第八章 指针	(163)
8.1 指针的重要性	(163)
8.2 变量的指针和指向变量的指针变量	(163)
8.3 指针的定义	(165)
8.4 指针运算符 * 和 & 的相互关系	(167)
8.5 指针值传递的单向性	(169)
8.6 指向数组的指针	(171)
8.7 指针的运算	(172)
8.8 指针加减时的比例因子	(176)
8.9 指针动态分配和给指针赋初值	(176)
8.10 指针比较大小	(183)
8.11 指针与字符串	(184)
8.12 和存储模式相关的指针修饰符	(186)
8.13 与远地址相关的指针函数	(192)
8.14 指向结构的指针和符号 ->	(194)
8.15 用指向结构的指针作函数的参数	(196)
8.16 结构中有指向自身的指针	(197)
8.17 指向函数的指针(函数指针)	(200)

8.18 返回指针值的函数	(201)
第九章 结构与联合	(202)
9.1 结构	(202)
9.2 结构指针	(206)
9.3 访问结构成员	(206)
9.4 结构数组	(206)
9.5 用 sizeof 求结构的大小	(207)
9.6 联合	(208)
9.7 读取任意 *.DBF 文件中的数据	(211)
第十章 位运算与位域	(216)
10.1 计算机中的位	(216)
10.2 数循环移位	(217)
10.3 位域	(218)
第十一章 预处理指令和编译控制行	(223)
11.1 定义宏指令	(223)
11.1.1 定义不带参数的宏	(224)
11.1.2 定义带参数的宏及标识符的粘接	(224)
11.1.3 有关宏的其它一些说明	(224)
11.1.4 调试宏	(225)
11.1.5 预定义宏	(226)
11.2 取消宏定义指令	(228)
11.3 文件包含(嵌入)指令	(230)
11.3.1 包含指令格式	(230)
11.3.2 标头文件	(230)
11.4 条件编译指令	(232)
11.5 出错指令	(234)
11.6 报告现程序有汇编代码的指令	(235)
11.7 警告处理指令	(235)
11.8 保证 Huge 函数执行时不变寄存器值指令	(236)
11.9 将行号嵌入执行文件指令	(236)
11.10 空编译指令	(236)
11.11 生成列表文件的预处理程序 CPP.EXE	(236)
第十二章 接收自变量个数可变的宏	(243)
12.1 数据类型和宏	(243)
12.2 Turbo C 函数特殊参数 "... "的用法	(248)
12.3 应用实例	(256)
第十三章 函数	(259)
13.1 函数类型标识符	(259)
13.2 函数说明和函数原型	(261)
13.3 函数定义	(264)
13.4 函数参数和函数中的变量	(264)
13.5 函数的返回值	(266)

13.6 函数的调用和调用约定	(267)
13.7 函数说明、定义和调用之间的关系	(274)
13.8 函数的嵌套调用	(276)
13.9 函数的递归	(276)
13.10 内部函数	(277)
13.11 外部函数	(277)
13.12 程序的可执行语句应在函数定义的语句体中	(278)
13.13 函数的种类	(278)
13.14 函数与数组	(279)
13.15 函数和指针	(282)
13.16 汇编语言调用 Turbo C 函数	(282)
第十四章 程序结构和主函数	(283)
14.1 程序结构	(283)
14.1.1 独立的 C 源程序	(283)
14.1.2 源程序由几个子源程序构成	(283)
14.1.3 并立源文件	(285)
14.2 源程序部分内容说明	(286)
14.2.1 文件名	(286)
14.2.2 标识符	(286)
14.2.3 双限界匹配符	(287)
14.2.4 注释	(288)
14.2.5 语句与编译指令	(288)
14.2.6 函数	(288)
14.2.7 关键字	(289)
14.3 主函数 main ()	(289)
14.3.1 主函数在程序中的位置	(289)
14.3.2 参数	(289)
14.3.3 使用关键字 cdecl	(291)
14.3.4 返回值	(291)
14.4 DOS 环境和环境函数	(292)
第十五章 驻留内存的帮助工具文件 THELP.COM	(299)
15.1 语法	(299)
15.2 在 THELP 激活后所能使用的键	(304)
15.3 错误信息	(305)
第十六章 集成开发环境和缺省参数设置	(307)
16.1 怎样进入集成环境	(307)
16.2 集成环境中的热键	(309)
16.3 集成环境中菜单结构	(312)
16.4 用 TCINST.EXE 程序设置 TC.EXE 参数缺省值	(317)
16.5 TCINST.EXE 的菜单结构	(318)
16.6 TC.EXE 与 TCINST.EXE 菜单项详细说明	(322)
16.7 DOS 5.0 的行编辑器 EDLIN.EXE	(385)

第十七章 编译和调试程序	(392)
17.1 静态检查	(392)
17.2 编译查错	(393)
17.3 程序调试	(394)
17.4 DOS 5.0 的调试程序 DEBUG.EXE	(413)
17.5 错误、警告及提示信息	(425)
第十八章 DOS 错误处理函数	(436)
18.1 全局变量与数组	(436)
18.2 库函数	(437)
第十九章 硬盘体系结构和主引导程序	(449)
19.1 主引导扇区的查找	(450)
19.2 主引导扇区中分区内容的说明	(451)
19.3 分区基本输入输出参数块 BPB 的内容	(452)
第二十章 磁盘文件的结构	(455)
20.1 目录项的结构	(455)
20.2 文件分配表 FAT	(458)
20.2.1 逻辑扇区	(458)
20.2.2 簇	(459)
20.2.3 FAT 的表头标志	(459)
20.2.4 DOS 将一个簇分配给新文件的过程	(460)
20.2.5 如何使用 FAT	(460)
20.3 库函数	(461)
第二十一章 程序头前缀 PSP	(474)
21.1 PSP 的作用	(474)
21.2 PSP 在内存中的位置	(474)
21.3 全局变量 — psp 和库函数 getpsp()	(474)
21.4 PSP 的内容	(476)
21.5. COM 文件与 PSP 的关系	(484)
21.6. EXE 文件和 PSP 的关系	(485)
第二十二章 中断和中断函数	(486)
22.1 中断矢量	(486)
22.2 中断过程和中断优先权	(488)
22.3 部分库函数用到的中断	(488)
22.4 BIOS 工作区	(490)
22.5 调用中断库函数	(493)
22.6 端口、内存单元存取函数	(508)
22.6.1 端口地址	(508)
22.6.2 读写端口或内存单元内容	(514)
22.7 内存控制块 MCB	(517)
22.8 interrupt 中断函数修饰符和常驻内存程序	(519)
第二十三章 串行通讯	(530)
23.1 RS-232	(530)

23.2 库函数 bioscom()	(532)
第二十四章 控制内存块函数	(537)
24.1 分类	(537)
24.2 库函数	(537)
第二十五章 动态地址分配函数	(544)
25.1 分类	(545)
25.2 库函数	(546)
第二十六章 数学函数	(555)
26.1 常数和宏说明	(555)
26.2 函数或宏分类	(557)
26.3 详细说明	(559)
第二十七章 80x87 数学协处理器	(595)
27.1 概述	(595)
27.2 数据类型	(598)
27.3 80x87 指令简要说明	(599)
27.4 80x87 函数	(606)
27.5 其它一些说明	(610)
第二十八章 日期与时间函数	(613)
28.1 概述	(613)
28.2 库函数	(614)
第二十九章 目录函数	(628)
29.1 分类	(628)
29.2 库函数	(628)
29.3 一个全盘搜索文件程序	(637)
29.4 DOS 5.0 的 dir 命令	(642)
29.5 功能强于 DOS 5.0 内部命令 dir 的 CDIR	(645)
第三十章 文件管理	(663)
30.1 缓冲型文件系统和非缓冲型文件系统	(663)
30.2 C 语言的 FILE 结构剖析	(663)
30.3 文本流与二进制流	(669)
30.4 标准 I/O 预定义流	(670)
30.5 文件控制块 FCB	(671)
30.6 库函数及设备驱动程序	(673)
第三十一章 格式输入与输出函数	(754)
31.1 格式输出函数	(754)
31.1.1 参数 format 的书写规则	(754)
31.1.2 ... (可变参数表)	(764)
31.1.3 库函数	(765)
31.2 格式输入函数	(768)
31.2.1 参数 format 的书写规则	(768)
31.2.2 函数返回值	(780)
31.2.3 函数说明	(781)

第三十二章 过程控制函数	(783)
32.1 进程管理函数	(783)
32.2 TC.EXE 文件结构剖析	(797)
第三十三章 键盘与鼠标	(802)
33.1 键盘	(802)
33.1.1 键的分类	(803)
33.1.2 接通码和释放码	(804)
33.1.3 换挡键 / 双态键的状态字节	(805)
33.1.4 库函数 bioskey()	(810)
33.1.5 键盘缓冲区	(811)
33.1.6 键码测试程序	(814)
33.1.7 程序中定义键值的方法	(820)
33.1.8 键盘中断	(821)
33.1.9 应用	(831)
33.2 鼠标	(836)
33.2.1 鼠标安装	(836)
33.2.2 使用鼠标的演示程序	(837)
33.2.3 鼠标的图形光标设计	(849)
33.2.4 用鼠标画图	(850)
第三十四章 打印机	(856)
34.1 概述	(856)
34.2 控制打印机函数	(858)
34.3 DOS 5.0 的脱机打印程序 PRINT.EXE	(863)
34.4 设置打印参数	(865)
34.5 图象打印	(868)
第三十五章 视频函数	(880)
35.1 Turbo C 涉及的显示卡	(880)
35.2 显示卡的体系结构	(882)
35.3 视频缓冲区与视频组合	(893)
35.4 屏幕显示方式	(894)
35.5 文本方式	(896)
35.5.1 屏幕的绝对坐标和窗口的相对坐标	(896)
35.5.2 文本方式下的数据格式	(896)
35.5.3 字符属性	(896)
35.5.4 视频页	(900)
35.5.5 光标形状	(900)
35.5.6 文本方式下使用的库函数	(901)
35.6 图形方式	(913)
35.6.1 象素和字节的关系	(913)
35.6.2 变量 directvideo	(920)
35.6.3 使用图形函数的注意事项	(920)
35.6.4 系统控制	(920)

35.6.5 屏幕及视口管理	(937)
35.6.6 颜色控制	(944)
35.6.7 图形方式下的正文输出	(949)
35.6.8 绘图与填充	(958)
35.6.9 图形方式下的错误处理	(971)
35.7 BIOS 中断 INT10H 的功能	(974)
35.8 图形驱动程序和字体转换工具BGIOBJ.EXE	(1006)
35.9 图形演示程序 BGIDEMO.C 注释	(1012)
35.10 在西文操作系统下直接显示汉字	(1035)
第三十六章 发声	(1040)
36.1 计算机发声原理和相关库函数	(1040)
36.2 乐曲构成原理	(1040)
36.3 演奏音乐例程	(1042)
第三十七章 搜索与排序函数	(1048)
第三十八章 对 ANSI 定义信号对应的动作重定义	(1062)
38.1 库函数	(1062)
38.2 关系 ssignal() 和 gsignal() 函数的转换	(1065)
第三十九章 如何用 C 语言访问扩页内存	(1067)
第四十章 命令行编译器 TCC.EXE	(1076)
40.1 TCC 命令行书写语法规则	(1077)
40.2 命令行配置文件 TURBOC.CFG	(1083)
40.3 配置文件转换实用程序 TCCONFIG.EXE	(1083)
40.4 应用举例	(1084)
第四十一章 独立连接程序 TLINK.EXE	(1085)
41.1 使用 TLINK.EXE 的一般语法	(1085)
41.2 连接 TurboC 程序的方法	(1087)
41.3 TCC.EXE 要使用 TLINK.EXE	(1088)
41.4 例子	(1088)
41.5 混合模式的连接	(1089)
41.6 可能产生的错误信息	(1090)
第四十二章 独立管理开发程序 MAKE.EXE	(1091)
42.1 文件间的依赖关系	(1091)
42.2 MAKE 文件	(1092)
42.2.1 注释	(1092)
42.2.2 显式规则	(1093)
42.2.3 隐含规则及部分DOS 命令	(1094)
42.2.4 宏	(1100)
42.2.5 指令	(1102)
42.3 使用 MAKE 的方法	(1104)
42.4 BUILTNS.MAK 文件的使用	(1105)
42.5 MAKE 错误信息	(1105)
第四十三章 库管理程序TLIB.EXE	(1107)

43.1 语法	(1107)
43.2 例	(1109)
43.3 注意事项	(1111)
43.4 可能出现的错误或警告	(1111)
第四十四章 目标模块交叉引用工具 OBJXREF.COM	(1114)
44.1 语法	(1114)
44.2 响应文件选择项	(1125)
44.3 输入文件名	(1127)
44.4 OBJXREF 处理过程	(1128)
44.5 可能出现的警告或错误	(1128)
附 录	(1129)
表 0—1 库函数与宏	(1129)
表 0—2 结构或联合	(1140)
表 0—3 枚举	(1141)
参考资料	(1142)

引 言

C 语言是介于低级语言和高级语言之间的语言,故可称之为中级语言。Turbo C 在国内流行较早,用户很多。它之所以受到用户的普遍欢迎,除了它自身的成功外,在很大程度上还由于有像《用户手册》、《参考手册》、《使用大全》及《运行库函数、源程序与参考大全》等书从不同的角度对 Turbo C 作了介绍,使用户获益匪浅。然而,随着用户层次的变化,在实际应用中用户对此类书渐感不足。例如,过于简要使人不免感到有些地方太难理解;内容繁多让人目不暇接,相同内容分散叙述往往使人不易搞清它们之间的依赖关系;提供的少量例程虽能帮助读者增加感性认识,然而缺乏对执行结果、应用环境及注意要点比较详尽的中文解释。库函数源程序的阅读更使初学者感到力不从心,因为它首先要求读者应有相当的阅读 C 语言的能力。此外,大多数书中内容和 Turbo C 版本并非一一对应(像函数原型中的参数标识符等);另外,一些重要的内容(如鼠标等)未涉及。值得指出的是,结合我国实际情况(例如在汉字操作系统中)的应用,更是不少读者所关心的。换句话说,能沟通各部分内容,较详细地解释 Turbo C,以便使读者不用具备很多程序设计方面的知识就能较快地理解 Turbo C,从而较好地利用 Turbo C 来解决自己的实际问题,这是本书追求的主要目标。

本书讲述的软硬件环境是:

主机以 IBM PC 及其兼容机为主;操作系统是 DOS 3. X 或 DOS 5. 0。

Turbo C 版本为

Turbo C

Version 2. 0

Copyright (c) 1987, 1988 by

Borland International, Inc.

以 Turbo C 集成环境为依托,大部分例程采用缺省环境。

对涉及的标准头文件、库函数原型等书写形式,基本上同原软件版本。

计算机科学是一门发展很快的科学,特别是国外新技术的不断引进,致使它的内容更为丰富,新名词、新概念不断涌现。目前在这方面,往往同一内容可能有多个不相同的名词与之对应,一时还不能统一起来。为了帮助理解,书中常将一些术语和实例结合起来,而不是给它下确切的定义。部分术语用【】括了起来。

本书在系统地介绍 Turbo C 2. 0 时,以 600 多个 Turbo C 源程序作为讲解基础,它们具有以下特点:

1. 大多数例程具有完整性和独立性,换句话说,每个例程均可单独调试和应用。
2. 从一句话编程技巧直到复杂的实用程序,一般都有详细的中文解释和必要的输出结果。
3. 对涉及的软硬件环境作了必要的说明,读者由此可以了解来龙去脉。
4. 注重其实用性、技巧性和编写形式上的多样性。对关键的常用编程方法作了技术性总结。
5. 为便于查阅,按用户使用习惯将所有库函数按功能进行了仔细分类和汇总,指出了它

们之间的区别与应用环境。

6. 收录了部分自编实用函数。

7. 部分内容属于高级程序设计技术。

8. 融进了近年来国内 C 语言专家、学者的部分研究成果, (如西文状态下显示汉字、鼠标编程、功能强于 DOS 5.0 内部命令 DIR 的 CDIR 程序等) 注释后供读者参考。

这些例程既可供初学者起步时演习用, 也可拓宽软件开发人员的思路。另一方面, 它们也回答了读者所关心的许多问题, 读者藉此可以避免对所涉及的内容进行毫无把握的猜测和盲目测试, 从而能透彻理解这些内容, 进而解决初用 Turbo C 编程时无从下手的烦恼。相信初学者在从这些例程中获得许多实用编程技术的同时, 还可了解如何充分利用计算机资源方面的许多知识。

对高级 C 程序员来说, 可从中了解程序设计技巧和风格、函数相互之间的依赖关系; 内中搜集的许多技术资料, 具有相当的实用参考价值。因此, 可把本书作为一本综合性技术手册使用。

随着计算机和计算机技术的不断发展, 不同的机型、不同的操作系统、不同的应用软件可能使同一例程或函数在使用时会产生不同的结果, 读者应在自己的软硬件环境下对它们作必要的跟踪调试, 确定其可靠性, 弄清其可移植性。对重要的结论务必进行验证, 绝不要轻率从事。自然, 读者也可用书中提供的例程为基础进行调试和验证, 必要时应当加以更改或扩充。

选用 Turbo C 2.0 版本作为讲解基础的主要原因之一是该版本在国内出现较早, 流行比较广 (需要此版本的读者可与出版部门或作者联系)。特别是它占用内存小 (在家用电脑上都可用), 灵活方便, 各个层次的用户都可利用。另一个原因是该版中已有一个很好的学习 C 语言的工具, 即集成环境。它集源程序编辑、编译、连接和调试于一体, 尤其是单步断点符号调试以及像调试表达式等, 实在对初学者极有帮助, 它使一些难以理解的概念 (如指针、结构、联合、文件句柄、流、堆栈等等) 很容易得到直观的解释。跟其它书不同, 本书在例程中重点对此作了较详细的介绍。

事实上众多的计算机语言都有其共性的一面, 或者说, 当读者掌握了这部分基础知识后, 对较高版本的 C 和其它语言的学习和应用也许就容易多了。

从某种角度上可以说, 本书相当一部分内容是对 Turbo C 使用测试的报告, 有些内容是其它 Turbo C 参考书中未予提示的 (如 FILE 结构剖析、模式库 CS.LIB 修改、某些库函数应用的特殊要求、“...”参数的使用约束、集成环境中运行结果不同于命令行的情况等等)。书中记载了测试中发现的一些问题, 同时还校正了一些参考书中个别的错误。

对熟悉汇编语言的读者, 在有关库函数中简要指出相关的 DOS 操作系统的内部功能, 如 BIOS 中断或 DOS 中断调用, 以便对照。对重要的 DOS 命令也作了相关性叙述。

本书深入浅出, 层次清晰, 通俗易懂, 资料丰富实用, 适合软件人员、程序员、电大、职大、函授学院学生、计算机自学考试人员阅读和 C 爱好者自学用, 也可作为程序员案头实用参考手册。还可供大中专院校计算机及相关专业的师生参考。

读完本书后读者兴许会发现, Turbo C 涉及的内容既广泛又丰富, 然而学习它也并不比学习 BASIC 复杂多少。另一方面, 编者深知, 限于自己的水平与时间仓促, 在不少地方所作的解释可能是肤浅的、不能令人满意的, 甚至是值得商榷的。本书旨在抛砖引玉, 恳请读者对书中错误和不妥之处批评指正。

作 者

1994 年 4 月于成都

第一章 Turbo C 的安装

1.1 安装 Turbo C 的软硬件环境

1. 使用机型

IBM PC 系列机,包括 XT、AT、PS/2、80×86 系列和兼容机。内存至少要有 448K 的 RAM。

2. 操作系统

要求 DOS 2.0 及其以上版本。

3. 显示器

应为 80 列显示,单色彩色均可。能驱动 CGA、MCGA、EGA、VGA、TVGA、SVGA、Hercules、IBM8514、AT&T400 和 3270 PC 等图形适配器。

4. 至少要有有一个软盘驱动器。

至少要有有一个软盘驱动器,最好还应有一个硬盘,这样可使操作更加方便。

5. 80x87 协处理器芯片为可选。

6. 其它硬件,如打印机、鼠标等用户可自行按需配置。Turbo C 集成环境不支持鼠标操作。

1.2 Turbo C 2.0 软磁盘上的内容

在 Turbo C 2.0 提供的六张 360K 磁盘的第一张盘上有一个文件 README,上面有一些帮助信息。将第一张盘插入一个驱动器,例如 A 驱动器,然后在 DOS 提示符下键入 README

```
C>README
```

便可在 80 列屏幕上看到该文件的内容。实际上,这相当于以下操作:

```
C>README README
```

即执行 README.COM 时装入 README 文件。文件装入后只能阅读而不能修改。注意:以后可以发现,在 Turbo C 集成环境下,只要将 README 文件事先换名为一个有扩展名的文件,例如 README.MYC,就可用集成编辑器直接将 README.MYC 读入观察了,也可修改。当然,一般你不要这样做)。

该文件的最后一部分指出提供的 Turbo C 盘片上有哪些文件和它们的用途,这是很重要的(事实上,读者将来会发现,对制作系统盘而言,有些文件是可要可不要的)。现将 Turbo C 2.0 软盘信息罗列如下:

第一张盘 INSTALL/HELP (安装 / 帮助盘)

INSTALL EXE—安装程序

README COM—用于读文本文件 README 的程序,它只能阅读而不能修改被阅读文件

TCHELP TCH — Turbo C 帮助文件,二进制文件

THELP COM — 接收 TCHelp.TCH 文件内容并弹出的程序,它可常驻内存,也能撤去

THELP DOC — 说明怎样使用 THELP.COM 程序的文本文件

README — 一个说明 Turbo C 2.0 相关内容的文本文件

第二张盘 INTEGRATED DEVELOPMENT ENVIRONMENT(集成开发环境盘)

TC EXE — Turbo C 集成编译器

TCCONFIG EXE — 配置文件(configuration files) 转换程序

MAKE EXE — 规划管理程序

GREP COM — 查找串程序

TOUCH COM — 要强迫重新编译或重建目标文件程序时而修改目标文件时间的程序

第三张盘 COMMAND LINE/UTILITIES (命令行编译器 / 实用工具盘)

TCC EXE — Turbo C 使用命令行的编译器

CPP EXE — Turbo C 预处理程序

TCINST EXE — 设置 TC.EXE 缺省参数的程序

TLINK EXE — Borland Turbo 连接器

HELPME! DOC — 常见问题和答案

第四张盘 LIBRARIES (库程序盘)

C0S OBJ — Small 模式启动代码库

C0T OBJ — Tiny 模式启动代码库

C0L OBJ — Large 模式启动代码库

MATHS LIB — Small 模式数学库

MATHL LIB — Large 模式数学库

CS LIB — Small 模式运行库

CL LIB — Large 模式运行库

EMU LIB — 浮点 8087 仿真库

GRAPHICS LIB — 图形库

FP87 LIB — 8087 库

TLIB EXE — Borland Turbo 库管理程序

第五张盘 HEADER FILES / LIBRARIES (标头文件 / 库程序盘)

???????? H — Turbo C 所有标头文件
<SYS> — 装有 *.H 标头文件的子目录
C0C OBJ — Compact 模式启动代码库
C0M OBJ — Medium 模式启动代码库
MATHC LIB — Compact 模式数学库
MATHM LIB — Medium 模式数学库
CC LIB — Compact 模式运行库
CM LIB — Medium 模式运行库

第六张盘 EXAMPLES/BGI/MISC (例子 /BGI 图形库 / 杂项)

UNPACK COM — 打开 *.ARC(archive file, 档案文件) 的解包文件。当原档案文件用了打包文件(象 PKARC.COM 或 PKXARC.COM 等) 对文件进行了压缩, 则在具体使用时应将文件进行解包即还原。因没有这样做时, 可不用此文件操作。

OBJXREF COM — 目标文件交叉引用工具文件

C0H OBJ — Huge 模式启动代码库

MATHH LIB — Huge 模式数学库

CH LIB — Huge 模式运行库

GETOPT C — 对命令行选择项分析的一个函数

HELLO C — Turbo C 源程序例子

MATHERR C — 数学库例外情况处理源程序

SSIGNAL C — signal 和 gsignal 函数的源代码

CINSTXFR EXE — 将 TC 1.5 版缺省参数拷贝到 TC 2.0 中安装程序

INIT OBJ — 用于连接 Prolog 时的初始化目标文件

BGI ARC — BGI 驱动程序和字体打包文件, 内中包括以下文件:

BGIOBJ EXE — 把图形驱动程序文件或矢量字体文件装入到用户执行文件中去的程序

ATT BGI — ATT400 图形卡驱动程序

CGA BGI — CGA 图形卡驱动程序

EGAVGA BGI — EGA 和 VGA 图形卡驱动程序

HERC BGI — Hercules 图形卡驱动程序

IBM8514 BGI — IBM 8514 图形卡驱动程序

PC3270 BGI — PC3270 图形卡驱动程序

GOTH CHR — 哥特 (gothic) 矢量字体文件

LITT CHR — 小号 (small) 矢量字体文件

SANS CHR — 光滑不修饰 (sans serif) 矢量字体文件

TRIP CHR — 三重 (triplex) 矢量字体文件

BGIDEMO C — 图形演示程序

STARTUP ARC — 有启动源代码和相关文件的打包文件, 内中有:

RULES ASI — 说明和 Turbo C 接口的汇编包含文件的规则和结构, 文本文件

C0 ASM — 启动代码的汇编源代码
 SETARGV ASM — 命令行分析的汇编源代码
 SETENVP ASM — 配置环境的汇编源代码
 BUILD-C0 BAT — 建立启动代码模块的批处理文件
 MAIN C — 说明分别用 TCC.EXE 或 TASM.EXE 编译连接 C 文件的方案比较
 EMUVARS ASI — 说明仿真程序的汇编变量的文本文件
 WILDARGS OBJ — 扩充通配符 (wildcard) 参数模块的目标代码
 EXAMPLES ARC — 各类 C 例程打包文件, 其中有:
 CPASDEMO PAS — 演示 Turbo Pascal 4.0 和 Turbo C 接口的 Pascal 例程
 CPASDEMO C — 演示 Turbo Pascal 4.0 和 Turbo C 接口的 C 例程
 CTOPAS TC — 在与 Pascal 4.0 程序连接时为产生正确格式的 C 模块供 TC.EXE 使用的配置文件
 CBAR C — 供 PBAR.PRO 用的函数例子
 PBAR PRO — 演示 Turbo Prolog 和 Turbo C 接口的例程
 WORDCNT C — 演示源程序级调试例程。该程序虽能经编译通过, 但运行不能得到正确结果, 因为它有一些错误, 需经修改后才能使用。
 WORDCNT DAT — 供 WORDCNT.C 使用的数据文件, 文本文件
 MCALC ARC — 微型计算表格 (MicroCalc) 的源程序打包文件, 包括:
 MCALC DOC — MicroCalc 使用说明文件
 MCALC C — MicroCalc 的主程序
 MCINPUT C — MicroCalc 的输入子程序
 MCOMMAND C — MicroCalc 的输入命令子程序
 MCPARSER C — MicroCalc 的输入分析子程序
 MCUTIL C — MicroCalc 的其它用途的子程序
 MCDISPLY C — MicroCalc 屏幕显示子程序
 MCALC H — MicroCalc 的头文件
 MCALC PRJ — MicroCalc 的规划文件 (project file)

1.3 三种安装方法

Turbo C 2.0 可分三种情况选择安装:

1. 将所有文件安装到硬盘上
将第一张盘插入 A 驱动器, 然后操作

C>INSTALL

按回车出现提示时选

Install Turbo C on a Hard Drive

然后根据屏幕提示一步一步操作: 一张盘安装结束后将此盘抽出, 并插入下一张盘。在硬盘上开一个子目录 (本书假定为 C:\TC) 安装其所有文件, 该子目录里将有 TURBOC.CFG 文件。

2. 原硬盘里装有 Turbo C 1.5 版本, 现升级为 Turbo C 2.0

出现选项时利用光标键选

Update Hard Drive copy of Turbo C 1.5 to Turbo C 2.0

3. 安装到软盘上

首先准备好三张已格式化的 360K 空盘（可利用 DOS 的 FORMAT 命令实现），在选项时选

Install Turbo C to a Floppy Drive

注意：Turbo C 有六种存储模式：Tiny、Small、Medium、Compact、Large 和 Huge。每次运行 INSTALL.EXE 时，让你安装一种存储模式的 Turbo C（要三张盘）。一般三张盘的分工是：

第 1 张盘：Turbo C Work Disk（#1）Turbo c 系统盘

第 2 张盘：Libraries Work Disk（#2）库盘

第 3 张盘：Source File Work Disk（#3）源文件盘

事实上，对集成环境系统主要是 TC.EXE 程序，命令行系统是 TCC.EXE、TLINK.EXE 及程序 MAKE.EXE、TASM.EXE（如果手头只有 MASM.EXE，可用 DOS 的 REN 命令将它换名为 TASM.EXE）等。其它程序（主要是标头文件和库）可在它们运行时根据错误提示，缺什么拷什么的原则补充。

如果是液晶或等离子显示器，为使显示为 80 列，可用 DOS 命令：

C>mode bw80

说明：在不发生混淆的情况下，我们有时也把 Turbo C（或集成开发环境）简称为 TC。为明确起见，有时我们也用“□”表示一个空格符。

1.4 文本阅读器 README.COM

可执行文件 README.COM 是一个文本文件阅读器，即可以读出由 ASCII 码构成的文件，并将其内容在屏幕上显示出来。用户可以用光标移动键上下或左右移动屏幕内容，从而可以浏览整个文本内容。习惯上称这样的执行文件为文本阅读器。在这里，README.COM 主要用于阅读 README 文件。README 文件是一个 ASCII 码文件，即文本文件。TC 希望用户使用文本阅读器来浏览这个文件，以帮助用户在使用 TC 软件前对软件作必要的了解。该文件中有一些很重要的内容值得用户参考，我们将在适当的地方列出它的一些主要内容提醒读者注意。【文本阅读器】除供浏览外，也可将文本整个打印出来，但是不允许对文本文件本身进行修改（这实际是对被阅读文件的一种安全保护）。

一 阅读文本文件 README

如果 README.COM 和 README 两个文件都在 C 当前目录（可以是子目录）里，则用

C>README

回车后便看到 README 文件被读入屏幕上。其第 1 行显示的是文件创建时间和文件名。最底下一行是操作提示状态行（确切地讲，屏幕最顶上一行，即显示时间和文件名的那一行也是状态行）：

Command▲ * * * Top of file * * * Keys: ^ ↓ ←→PgUp PgDn ESC=Eixt F1=Help

例如:按下 F1 键便可得到操作命令提示:

F Find text
C Case-sensitive find
N Find next
F5,6 Color of text
F7,8 Color of Status lines
Home Start of files
End End of file
W Wrap long lines on/off
P Printing on/off
7,8 Strip or leave hi-bit
S Save defaults

ESC Return to file

下面对其作出较为详尽的说明。

1. 按下 F 键 (F 或 f 都可,下同) 不区分大小写查找单词。

状态行显示

Find▲

你可以输入你要查找的单词,例如 how 查到便将有此单词的首次出现的行移到屏幕第一行,随后的各行依次跟上。查找只能是正向查找,即从当前显示的内容查到文尾。不能反向查找;其次,不区分单词字母的大小写,即 how、HOW 和 How 均认为是一样的。

2. 按下 C 键区分大小写查找单词。

状态行显示

Scan▲

它跟 F 键操作不同之处只是区分单词字母中的大小写字母。

3. 按 N 键继续查找下一个单词所在行。

当找到一个单词后,单词所在行便显示在屏幕顶端。要在随后的行中寻找此单词,只要按动 N 键。

4. 按 F5 功能键改变显示文本的背景颜色。

每按一次,改变一种颜色。连续按键可使多种可显颜色周而复始出现。下同。

5. 按 F6 功能键改变显示文本的前景颜色

6. 按 F7 功能键改变状态行的背景颜色。

7. 按 F8 功能键改变状态行的前景颜色。

8. 按 Home 键,文本从头开始显示。

9. 按 END 键,屏幕显示文本尾部及以上内容。

10. 按 PgUp、PgDn 键上下翻页,逐屏显示。

11. 按 W 键后,对正文中长的行你可以用 ←键或 →键来回移动它,以便看到原先不能在屏幕上显示的内容。注意:它是乒乓开关,即再按一下它,←键或 →键就不起作用;又按一下,又可移动。

—

Abstract

二 阅读多个文本文件

README.COM 主要供 TC 用户阅读 README 文件用,但它实际上也可阅读其它文本文件。

例如

```
C>README E:\ZM\*.C
```

它便可以连续阅读 E 盘上子目录 ZM 中扩展名为 .C 的全部文件。具体方法是

假定目录中依次有三个文件:P1.C、P2.C 和 P3.C,那末 README.COM 一开始显示那个在目录中排在最前面的那个文件 P1.C。当你要浏览第二个文件时,按一下 ESC 键便可。随后再按一下 ESC 键便显示 P3.C 的内容。又按 ESC 键便退出阅读器。

事实上,键盘上有些键也能起到某种作用。例如,要立即退出阅读器可按 S、X、Q、F10 或 Alt-C 等键均可。按 ? 或 H 键相当于按 F1 键;按 U 或 D 键相当于 PgUp 或 PgDn 键等等。

从上述所述,可见文本阅读器比之 DOS 的 TYPE 命令是进了一步。

三 可能发生的错误信息

1. Text not found(文本文件未找到)
2. Not enough memory(内存不够)
3. Wrong DOS version(错误的 DOS 版本)
4. Open failed(打开文件失败)
5. File not found(文件未找到)
6. Path not found(路径未找到)
7. Too many files(文件太多)
8. Access denied(拒绝存取)

四 README.COM 的汉化方法

README.COM 的应用环境是西文状态,所以在 CCDOS 下使用要按 Ctrl-F7 键,即转换为西文状态。除了不能阅读汉字外,其余功能正常。要使它阅读汉字,应将它进行汉化。主要是对其显示部分的修改。下面列举一种利用 PCTOOLS 修改 README.COM 有关内容进行汉化的方法(未考虑 11 行显示和标尺显示)。

relative sector 0000016,Clust 01041

0048(0030) 8B FA 4F 48 0B C9 7E 66 8B 16 67 01 3B CA 76 02

23 ←-对应修改处,下同

0080(0050) 69 01 AC 8A 7E 01 3C 20 73 22 3C 00 75 0A 3A 3E

66 01 E8 0A 00 E2 F7 5E 07 5F 5A 59

0096(0060) 2A 01 75 18 47 47 EB 34 F6 06 94 01 08 75 0D 3B

5B 58 C3 53 52 50 89 F8 B7 A0 F6 F7 89 C2 86 D6

0112(0070) 3E 6D 01 73 07 B0 A8 90 8A 3E 2C 01 8A D8 80 3E

D0 EA BB 00 00 B4 02 CD 10 58 5A 51 B9 01 00 88

relative sector 0000017,Clust 01042

0000(0000) 8D 01 01 74 06 8A E7 AB EB 12 90 8A E7 FA EC D0

E3 B4 09 CD 10 59 5B 47 47 C3 90 50 26 8A 65 FF

0016(0010) D8 72 FB EC D0 D8 73 FB 8A C3 AB FB E2 B4 5E 07

E8 D0 FF 58 FE C1 E9 1E 04 90 90 90 90 90 90 90
0032(0020) 5F 5A 59 5B 58 C3 80 3E 98 11 24 75 10 57 56 1E
90 90 90 90 90

relative sector 0000025, Clust 01050
0048(0030) 09 74 0C AA 47 FE C1 EB ED FE C5 B1 08 DB D7 80
E9 D5 FB 90

修改后, F5 ~ F8 键改变颜色不起作用, 改变颜色要用 Ctrl-F6 即 CCDOS 的改变颜色的方法。在 IBM PC /AT 机上, 用 CGA 显示器 11 行汉字显示时, 帮助内容只显示 底下部分, 标尺不能正常显示 正文则能很好地反象显示。打印也可以。

第二章 80x86 指令和六种存储模式

2.1 寄存器

1. 通用寄存器

位 15 位 0

位 7...位 0 | 位 7 位 0

AH	AL
BH	BL
CH	CL
DH	DL

AX 累加器 (数学运算)

BX 基地址寄存器 (基址, 或变址)

CX 计数器 (循环计数器)

DX 数据寄存器 (保存数据)

图 2-1

说明: (1) 许多数学运算只能在 AX 中进行

(2) BX 可以存放 far 类型指针的偏移量部分 (用作指向内存某位置)

(3) CX 用于循环指令

(4) 有些指令用 DX 保存数据

2. 段地址寄存器 (也是 16 位)

CS 代码段指针

DS 数据段指针

SS 栈段指针

ES 附加段指针

图 2-2

说明: (1) 段地址寄存器的 16 位值左移 4 位 (即乘 16) 后才真正得到 20 位段地址。

(2) CS 寄存器指向 64K 内存块的开始处, 即指向代码段, 其中存放着下一个要执行的指令, 地址为 CS:IP。IP 寄存器中存放段偏移量。任何情况下都不能直接装载 CS 寄存器。

(3) DS 寄存器指向数据段的首部。数据段用于存放大多数内存操作数的 64K 大小的内存块。BX、SI 或 DI 中的内存偏移量通常是相对于 DS 而言的, 对内存的寻址也是相对于 DS 寄存器而言的。

(4) ES 寄存器指向附加段的 64K 内存块的首部。附加段只要需要就可用它。有时, 附加段用于设置附加的 64K 的内存块以存储数据。

(5) SS 寄存器指向堆栈段的首部。堆栈段是驻留堆栈的 64K 大小的内存区。SP 仅能对堆栈段内存区进行寻址。

3. 专用寄存器 (也是 16 位)

	SP 栈指针
	BP 基址指针
	SI 源索引(源变址)
	DI 目的索引(目的变址)

图 2-3

说明: (1) SP 指向当前栈顶, 它是相对于栈段的偏移量

(2) BP 是辅助栈指针, 它相对于堆栈段寄存器 SS 而言的。用于取栈中参数。BP 在函数中实现时作为参数和自动变量区的基地址。参数相对于 BP 的偏移量是正的, 大小随存储模式和函数入口时保存的寄存器个数而变。BP 总是指向所保存的旧的 BP 值。无参数及说明无参数的函数无需使用和保存 BP。

4. 重要的寄存器

位 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
	NT	IOPL	OF	DF	IF	TF	SF	ZF		AF		PF		CF	(FLAGS) 标志寄存器
															IP 指令指针
										ET	TS	EM	MP	PE	机器状态字 (MCW)

图 2-4

说明: (1) 虽然重要, 但 Turbo C 并不直接访问它们。

(2) NT —— 嵌套任务标志, 用于当前执行的任务是否嵌套在另一任务内。如果 NT 为 1, 则当前嵌套的任务有一个有效的链连接到前一个任务。

(3) IOPL —— 输入 / 输出特权级标志, 保证指令只完成那些允许完成的操作。

(4) OF —— 溢出 (Overflow) 标志。

(5) DF —— 字符串操作控制方向 (Direction) 标志, 随着 DF 复位为 0, SI/ 或 DI 自动增量; 反之, 自动减量。

(6) IF —— 中断 (Interrupt) 允许标志。

(7) TF —— 陷阱 (Trap) 标志, 当它为 1 时把微处理器置入单步方式, 从而允许对程序调试。

(8) SF —— 符号 (Sign) 标志, 对负的结果为 1, 正的为 0。

(9) ZF —— 零 (Zero) 标志, 结果为 0 时其值为 1。

(10) AF —— 辅助 (Auxiliary Carry) 进位标志, 表示最低有效的 4 位 BCD 数值位是否产生了进位或借位。

(11) PF —— 奇偶 (Parity) 标志, 在数据通讯中奇偶性为奇时为 1。

(12) CF —— 进位 (Carry) 标志, 当用 8 位或 16 位操作数完成的算术操作产生了进位或借位时, 它为 1。它主要用于移位和循环指令, 并包含移出或循环移出寄存器的位。

(13) IP —— 指令指针包含了在当前操作的码段内寻址下一条要执行的指令所必须的偏移量, 这个偏移量产生了下一条顺序的程序指令的全 32 位指针。

(14) ET —— 处理器扩展类型。

- (15) TS —— 任务切换位, 每当完成任务切换操作时自动置位 (为 1), 与此同时协处理器操作码将导致协处理器不可用陷阱。当 80x86 复位或执行 clts 指令后为 0。
- (16) EM —— 如它为 0, 则所有的协处理器操作码都将在实际的 80x87 协处理器上执行; 如为 1, 则表示机器上没装协处理器, 因此不能用协处理器操作码, 而将采用软件仿真操作。
- (17) MP —— 监控协处理器, 它只与 TS 一起使用, 如果 TS=1, WAIT 操作码便产生一个协处理器不可用信息。如机器上装有 80x87 时它为 1。
- (18) PE —— 保护允许位, 用于启动微处理器保护方式 (当其位为 1 时)。
- (19) 80386 的寄存器扩展为 32 位, 为区别于原寄存器, 所有对寄存器的访问都必须用字母 E 开始, 如指令指针是 32 位的, 称 EIP; 累加器为 EAX 等等。另外, 标志寄存器中增加:
- 位 16 RF —— 恢复标志位, 当调试寄存器的断点和单步操作时成功地完成每条指令时它复位为 1; 否则下条指令中的所有调试故障都被忽略。
- 位 17 VM —— 虚拟 8086 方式标志, 如果它为 1, 且 80386 在保护方式下, 微处理器就切换到虚拟的 8086 方式进行操作; 在仿真 8086 期间, 80386 对特权操作码也将产生异常 13 故障。
- (20) 80386 有 3 个 32 位的控制寄存器 CR0、CR2 和 CR3, CR0 的低 16 位就是 8086/80286 的机器状态字, 这样就使 80386 与保护方式下的 80286 兼容。80386 还有 4 个系统地址寄存器: GDT (全局描述符表)、IDT (中断描述符表)、LDT (局部描述符表) 和 TSS (任务状态段表), 用于访问 80386 保护方式所支持的一些表和段。

它还有 8 个 32 位的调试 (debug) 寄存器 DR0 ~ DR7 和 2 个 32 位的测试寄存器 TR6 和 TR7, DR6 用于设置断点, DR7 显示断点现行状态; TR6 用于测试控制, TR7 用于测试状态。

2.2 8086 地址计算

下面介绍 8086 使用段寄存器计算地址的方法。

8086 完整地址由两个 16 位值组成。现设一数据段地址是 2F84H (即 DS 值)。想要知道数据段中偏移量为 0532H 处数据的真实地址【绝对地址】, 可按如下法计算:

第一步 将已知的段地址左移 4 位 ($2F84H \ll 4$)

$$(0010\ 1111\ 1000\ 0100) \ll 4 = 0010\ 1111\ 1000\ 0100\ 0000 = 2F840H$$

第二步 将左移后的值加上偏移量

$$\begin{aligned} 2F840H + 0532H &= 0010\ 1111\ 1000\ 0100\ 0000 + 0000\ 0101\ 0011\ 0010 \\ &= 0010\ 1111\ 1101\ 0111\ 0010 = 2FD72H \end{aligned}$$

这 20 位的数值即为所求。

从上例可以看出, 段的实际地址总是一个 20 位数, 但是由于段寄存器只能保留 16 位值, 因而段址总是用最左边的 16 位构成, 而把右边 4 位总约定为 0000, 这就是为什么计算实际地址时要将段址左移 4 位的原因。

由此可知, 内存中每隔 16 个字节, 才能有一个地址可作为段首地址。所以常把内存中 16 个字节称为【节】(Paragraph), 故段总是从节的边界开始的。

地址的标准写法是,段:偏移量。按照上面的处理方法,可以发现【数对】(段:偏移量)的不唯一性,这是由于不同的数对在段值左移4位后加上偏移量可能是相同的,即不同数对可能均表示着内存中同一单元。

例如,数对 0000:0123 和 0002:0103 的 20 位数均为 00123H。
于是对应着同一单元。

由此也就可以理解为什么整个程序不超过 64K 时段可以重迭,即有同一个开始地址。在同一段内段首址已定而只偏移量不同时,不同偏移量对应的地址是唯一的。

2.3 8286/ 80386 指令集简介

编译器允许任何操作数,错误由汇编检测。操作数的格式编译器没有强制限制。Turbo C 2.0 主要是对 8086/80186/80286,对 80386 未作说明,因此有关内容只供参考。

有关 80x87 协处理器指令参见《80x87 数学协处理器》一章。其中助记符

fdecstp,ffree,fincstp

在使用浮点运算和子程序中使用嵌入 (inline) 汇编时 (TCC -f 选择项),则不能使用它们。

注意:在使用 80186 助记符时,编译时要包括 -L 选择项。对只适用于 80286/80386 的指令在助记符后已指出。

所谓 BCD 码是指二—十进制数。通常在微机中,从键盘输入一个数字 (0 ~ 9) 是以 ASCII 码表示的。若用 ASCII 码表示一个十进制数字,则一个存储单元 (8 位) 只能存储一个十进制数字。而一个十进制数字最多只用低 4 位就够了 (最大数 9=00001001b)。为了有效利用空间,可采用 BCD 码,即一个十进制数用 4 位二进制数表示。这样 8 位可存储两个十进制数字,或者说,可以把相邻两个存储单元中的内容合并到一个存储单元中。一个字节由两个十进制数字组成,一个在高 4 位,另一个在低 4 位,则称这个数为组合 (或压缩)BCD 数。如果一个字节高 4 位为 0,低 4 位表示一个十进制数,则称该数为未组合 (或不压缩)BCD 数。

一 分类

(一) 算术运算

- . 加法 add,adc,inc
- . 减法 dec,neg,sbb,sub
- . 乘法 mul,imul
- . 除法 div,idiv
- . ASCII 码调整 aaa,aad,aam,aas
- . 十进调整 daa,das

(二) 位运算

- . 逻辑位处理 and,not,or,xor
- . 扫描置 1 位 bsf,bsr
- . 移位 shl,shr
- . 循环移位 rol,ror,sal,sar
- . 多字值移位 rcl,rcr
- . 多位移位 shld,shrd

(三) 控制流程

- . 无条件转移 jmp
- . 条件转移 jxxxx
- . 比较转移 cmp

- . 位测试和转移 test
- . 测试和设置位 bt, btc, btr, bts
- . 循环 loop, loope, loopz, loopne, loopnl, jcxz
- . 条件地设置字节 setxxxx
- . 调用过程 call, ret, iret
- . 建立堆栈框架 enter, leave
- . 使用中断 int, into
- . 检查存储范围 bound

(四) 处理字符串

- . 重复前缀 rep, repe, repz, repne, repnz
- . 移动串 movs, movb, movw
- . 搜索串 scas, scasb, scasw
- . 比较串 cmps, cmpsb, cmpsw
- . 填充串 stos, stosb, stosw
- . 把串中值装入寄存器 lods, lodsb, lodsw
- . 从端口把串读入存储器 ins, insb, insw
- . 把串从存储器写入端口 outs, outsb, outsw

(五) 装入、存储和移动数据

- . 复制数据 mov
- . 交换数据 xchg
- . 查找数据 xlat
- . 传输标志 lahf, sahf
- . 扩展带符号值 cbw, cwd
- . 移动并扩展值 movsx, movzx
- . 写入近指针 lea
- . 写入远指针 lds, les, lfs, lgs, lss
- . 向堆栈来回送数据 push, pop
- . 把标志保存在堆栈 pushf, popf
- . 把所有寄存器保存到堆栈 pusha, popa
- . 把数据来回向端口传送 in, out

(六) 控制处理器

- . 控制计时和对准 nop
- . 标志操作 cld, cld, cmc, stc, std, cli, sti
- . 控制处理器 lock, wait, hlt
- . 控制保护模式进程 lar, lsr, lgdt, lidt, sgdt, sidt, ltr, str, lmsw, smsw, arpr, clts, verr, verw

二 指令

1. aaa ; aaa 是指令操作码助记符,下同。add 加法后 al 中的 ASCII 调整。它紧跟在 add 指令后执行。它把 AL 中由两个未组合的十进制数相加后的内容转换为未组合的十进制数和。它将 AL 高 4 位置 0,并检查 AL 中的低 4 位,是否是 0~9 之间合法的 BCD 数。如低 4 位有大于 9 的数或 AF=1,则 AF=1, CF=1,且 AH 加 1, AL 加 6 后让其高 4 位为 0;如有一个十进制进位,则 AF=0, CF=0。

2. aad

除法操作前对 AX 中的 ASCII 调整。它把 AX 中的两个未组合的十进制数在两个数相除以前进行校正,以便相除以后可以得到正确的未组合的十进制结果。

用 AL 中的最低有效数位和 AH 中的最高数位准备进行除法操作。除法后 AL 为 AL+(10*AH), AH 为 0;AX 中最后是原始的未组合的两位数字的二进制等量值。

3. aam

乘法后 AX 中的 ASCII 调整。它把 AX 中的两个未组合的十进制数相乘的结果校正,最后在 AX 中得到正确的未组合的十进制积。

它在 MUL 指令之后执行。用 AL 除以 10 把 AL 结果拆开,把最高有效数位(商)留在 AH 中,余数留在 AL 中。

4. aas

减法后 AL 中的 ASCII 调整。它将 AL 中由两个未组合的十进制数相减的结果校正后,在 AL 中产生一个正确的未组合的十进制数差。

它在 sub 指令之后执行,如 AL 中低 4 位大于 9 或 AF=1,则 AL 减 6, AH 减 1, AF=1, CF=1, AL=(AL & 0FH); 否则, CF=0, AF=0, AL=(AL & 0FH)。为把 AL 中内容转换为一个 ASCII 码数, aas 指令之后必须跟指令 or AL, 30H。

5. adc 目的,源

带进位的整数加法。如果 CF 曾经为 1,则和数加 1,结果返回到目的操作数。常用于多字节(或字)加法操作(中的部分操作)。

6. add 目的,源

两个整数加法操作,相加后保存在目的操作数中。

7. and 目的,源

求两个操作数的逻辑与。即它们对应位上都为 1 时结果位才为 1,否则为 0。指令执行后 OF=0, CF=0。

8. arpl (选择器, CS 选择器) (80286/80386)

调整选择器 RPL 场。如果第一个操作数的 RPL 场(最低的两位)小于第二个操作数的 RPL 场, ZF=1,第一操作数的 RPL 场增加到与第二个操作数的 RPL 场相匹配;否则, ZF=0,第一个操作数不作修改。它常出现在操作系统软件中,而不出现在应用程序中。

9. bound 目的,源

检查数组下标的是否越界。该操作用于保证指定的数组下标是在由两个字的存储器块所定义的界限之内。第一个操作数应大于或等于第一个字,而小于或等于第二个字,否则就会产生中断 INT 5H。如果操作数是 32 位,则每个字应为双字。

10. bsf 目的寄存器,源操作数 (80386)

bsr 目的寄存器,源操作数

bsf 从右向左扫描,起始位的位置索引为 0,向着最高有效位扫描;bsr 则从左向右扫描。如找到了为 1 的位,则 ZF=0。如果没有找到为 1 的位,目标值是不确定的。它只对 16 位或 32 位寄存器操作。也可用于检查寄存器值是否为 0。

11. 测试和设置位指令

bt 寄存器/存储器,选择器

;位测试指令,检查目标值的特定位,把它的值送入进位标志,然后进位标志可用于其它指令,如条件转移等。

bts 寄存器/存储器,选择器

;置位。位测试并设置指令,检查目标值的特定位,把它的值送到进位标志中,然后设置这一位。

btr 寄存器/存储器,选择器

;复位。位测试并重置指令,检查目标的特定位,把它的值送到进位标志中,然后清除这一位。

btc 寄存器/存储器,选择器

;求反。位测试并取反指令,检查目标值的特定位,把它的值送到进位标志,然后把这一位的值取反。

12. call 参数或操作数

调用一个过程,保存下一条指令的地址在堆栈中,然后将程序控制传送给参数。当调用的过程完成后,继续执行紧接在 call 指令下边的指令。

13. cbw

把 AL 中带符号的字节转换成 AX 中带符号的字(AH 的所有位等于 AL 的最高位)。

14. cld

清除进位标志,即 $CF=0$ 。

15. cld

清除方向标志,即 $DF=0$ 。在执行 CLD 之后,字符串操作自动使变址寄存器 SI(或 DI)增加。

16. cli

清除中断标志,即 $IF=0$ 。它禁止除 NMI(不可屏蔽中断)以外的所有中断。

17. cts (80286/80386)

清除任务切换标志,即 $TS=0$ 。这是一条特权指令,只能在 0 级执行而且是保留供操作系统软件使用的。它保持对每次执行 wait 或 esc 的跟踪,如 $MP=1$ 和 $TS=1$,将进入陷阱。它常出现在操作系统软件中,而不出现在应用程序中。

18. cmc

进位标志 CF 求反。

19. cmp 目的,源

比较两个操作数。它从目的操作数中减去源操作数,从而引起 OF、SF、ZF、AF、PF 和 CF 等标志的变化。但它并不修改这两个操作数本身。除了立即方式操作之外,两数必须属于同一类型。在立即方式中,符号扩展的立即数据字节可与存储器字相比较。

20. cmps 源串,目的串

比较串操作数。将 SI 减去 DI,根据其差值设置标志位 CF、AF、PF、OF、SF 和 ZF,注意:虽然进行比较,但不直接改变 SI 和 DI,SI 和 DI 是根据 DF 值决定增或减, $DF=0$,则它们递增,否则递减。注意:cmps 右边的参数是由 DI 检索的操作数,而这个操作数是用 ES 进行寻址的。这一默认的规定是不能更改的。

例如:CMPS DS,BYTE PTR [SI],ES:[DI] 指定字节比较

21. cmpsb

按字节比较 DS:[SI] 和 ES:[DI] 中串操作数。影响标志位同 cmps。

22. cmpsw

按字比较 DS:[SI] 和 ES:[DI] 中串操作数。影响标志位同 cmps。

23. cwd

将 AX 中的带符号字转换为 DX,AX 中带符号的双字,这是通过把 AX 中的最高有效位扩展到 DX 中所有的位上实现的。

24. daa

加法后的 AL 中的十进制调整,它只能在两个组合的 BCD 操作数加法之后使用,以得到正确的组合的十进制和。它的操作过程是,如 AL 低 4 位大于 9 或 $AF=1$,则 AL 加 6 后 AF 仍为 1,否则 $AF=0$;如果上一次操作的结果大于 9FH,或者 $CF=1$,则 AL 加 60H, $CF=1$;否则 $CF=0$ 。

25. das

减法后的 AL 中的十进制调整,它只能在两个组合的 BCD 操作数减法之后使用,以得到正确的组合十进制差。它的操作过程是,如 AL 低 4 位大于 9 或 $AF=1$,则 AL 减 6 后 AF 仍为 1,否则 $AF=0$;如果上一次操作的结果大于 9FH,或者 $CF=1$,则 AL 减 60H, $CF=1$;否则 $CF=0$ 。

26. dec 目的

将目的内容减 1。

27. div 源

无符号除法。被除数是隐含的,在 AX 或 DX,AX 中,除数为源。

(1) 字节除以字节 MOV AL,N—BTE

DIV D—BIT;商在 AL 中,余数在 AH 中

(2) 字除以字 MOV AX,N—WRD

DIV D—BTE；商在 AL 中，余数在 AH 中

(3) 双字除以字 MOV DX,N—MSW

MOV AX,N—LSW

DIV D—WRD；商在 AX 中，余数在 DX 中

(4) 仅 80386 MOV EDX,M—N

MOV EAX,L—N

DIV ECX；商在 EAX 中，余数在 EDX 中

28. enter 立即字，立即字节

第一个操作数确定正在执行的子程序在堆栈中需要设置多少个动态存储器字节，第二个则给出在高级语言源码内子程序的嵌套级别。本指令确定有多少个堆栈帧的指针从前一个帧复制到新的堆栈帧，BP 用作堆栈帧的指针。如果第二个操作数为 0，就压入 BP，设置 BP 为 SP，并且从 SP 中减去第一个操作数。

29. hlt

使程序暂停执行，只有外部中断或复位才能重新启动执行。如用一个中断恢复程序的执行，所保存的 CS,IP 值将指向 hlt 后的下一条指令。

30. idiv 源

带符号整数除法，被除数在 AX 或 DX:AX 中。如果源操作数是字节，则 AX 除以源操作数，商在 AL，余数在 AH 中；当源操作数是字时，则用字除 DX:AX。除数的最高有效的 16 位保存在 DX 中，商在 AX 中，余数在 DX 中。余数的符号同被除数。如果商太大以至于目的寄存器装不下或除数为 0 时，则产生中断 INT 0H。

31. imul 源

带符号整数乘法。

如果源操作数是字节，则以源操作数乘以 AL 寄存器内容，而 16 位带符号的结果留在 AX 中。若 AH 寄存器是 AL 符号扩展，则 CF=0,OF=0，否则 CF=1,OF=1。如果源操作数是一个字，则源操作数乘以 AX 寄存器内容，32 位带符号结果留在 DX:AX 中。DX 包含着最高有效的 16 位。如果 DX 是 AX 的带符号扩展，则 CF=0,OF=0，否则它们都为 1。

32. in 累加器，端口

可以将字节或字传入 AL 或 AX 寄存器。当端口为常数值时，其范围为 0~255；若用 DX 来指定端口，则可以将 0~65535 先放入 DX 中，再用 IN AL,DX 或 IN AX,DX 指令传送数据。

33. inc 目的

加 1 指令。它使有效地址字节、字或字寄存器加 1。

34. ins 目的串，端口

先将 DX 指向一个输入端口，然后用本指令把一个字节或字送到 ES:[DI] 中。内存操作数必须用 ES 来寻址，越段访问是非法的。注意：它不允许用一个立即数作为指定的端口号，端口只能用 DX 寄存器寻址。

在执行这条指令后，DI 的内容会自动改变。若 DF=0，则 DI 增加 1（对字节）或 2（对字），相反则减少。传送字节或字是由目的串决定的。

35. insb

传送一个字节。从 DX 指定的端口输入到 ES:[DI]。

36. insw

传送一个字。从 DX 指定的端口输入到 ES:[DI]。

37. int 中断类型

产生软中断过程调用。中断类型一般为 0~255。

38. into

产生中断 INT 4H，该中断只有在溢出标志位 OF 置位时才进行。int 后面为字母 o。

39. iret

中断返回。在实地址方式下,该指令从堆栈中弹出 IP、CS 和 FLAGS,并恢复被中断了的码。在保护方式下,如 NT=1,将进行的操作与引起任务切换的 call 或 int 指令的操作是相反的。执行 iret 码将修改它在任务状态段中保存的状态。这意味着如果重新进入该任务将执行 iret 下边的指令;如果 NT=0,iret 将从一个中断子程序返回而没有任务切换。要返回到的源码的特权级必须与中断子程序的特权级相同或低于中断子程序的特权级。

40. jxxxx 条件标号

条件转移指令。它把一个或多个标志状态(jcxz 除外)作为它们的条件。设置特定条件标志最常用指令是 cmp 或 test。指令助记符中单个字母的含意是, a(above, 高于)、b(below, 低于)、e(equal, 等于)、g(greater, 大于)、l(less, 小于)、n(not, 不)及 o(标志 OF)、p(PF)、s(SF)、z(ZF)。括号中为测试标志状态。

ja	如果高于则转移 (CF=0,ZF=0)
jae	如果高于或等于则转移 (CF=0)
jb	如果低于则转移 (LF=0)
jbe	如果低于或等于则转移 (CF=1 或 ZF=1)
jc	如果有进位则转移 (CF=1)
cxz	如果 CX=0 则转移,不影响标志位
je	如果等于则转移 (ZF=1)
jg	如果大于则转移 (ZF=0,SF=OF)
jge	如果大于或等于则转移 (SF=OF)
jl	如果小于则转移 (SF<>OF)
jle	如果小于或等于则转移 (ZF=1,SF<>OF)
jna	如果不高于则转移 (CF=1 或 ZF=1)
nae	如果不高于或等于则转移 (CF=1)
jnb	如果不低于则转移 (CF=0)
jnbe	如果不低于或等于则转移 (CF=0,ZF=0)
jnc	如果没有进位则转移 (CF=0)
jne	如果不等于则转移 (ZF=0)
jng	如果不大于则转移 (ZF=1 或 SF<>OF)
jnge	如果不大于或等于则转移 (SF<>OF)
jnl	如果不小于则转移 (SF=OF)
jnle	如果不小于或等于则转移 (ZF=0,SF=OF)
jno	如果无溢出则转移 (OF=0)
jnp	如果奇偶性为奇则转移 (PF=0)
jns	如果无符号则转移 (SF=0)
jnz	如果不为 0 则转移 (ZF=0)
jo	如果溢出则转移 (OF=1)
jp	如果奇偶性为偶则转移 (PF=1)

jpe 如果奇偶性为偶则转移 (PF=1)
jpo 如果奇偶性为奇则转移 (PF=0)
js 如果符号标志为 1 则转移 (SF=1)
jz 如果结果为 0 则转移 (ZF=1)

41. jmp 目标

无条件把控制转移到不同程序段而不保存任何返回信息。

42. lahf

将标志寄存器的低 8 位中的内容 (SF、ZF、AF、PF 和 CF) 传送到 AH 中。

43. lar (存取权字节, 选择器) (80286/80386)

选择器可以是存储器或寄存器字。如果有关的描述符在当前特权级的选择器的 RPL 中可见, 则该描述符的存取权字节被装入第一个操作数的高字节, 而低字节设置为 0。如果完成了装入操作, ZF=1, 否则 (选择器是不可见的或是错误的类型) ZF=0。

44. lds 目的, 源

les 目的, 源

把有效地址中的双字 DW 装入 DS/ES 和字寄存器, 它是写入远指针指令。把设置在第二个操作数指定的存储器中的 4 字节指针装入段寄存器和一个字寄存器。该指针的第一个字 (偏移量) 装入第一个操作数指定的寄存器, 指针的最后一个字装入 DS 寄存器或 ES 寄存器。

对 80386 还可用 lss/lfs/lgs 指令, 字宽为 32 位、指针、操作数是 48 位, 它们分别把段地址写入 SS、FS 和 GS。

45. lea 目的, 源

把源操作数的偏移量传送到目的操作数中。它把近指针写入到指定寄存器。

46. leave

退出高级过程。它重新分配所有的局部变量, 并设置 BP 为 SP, 把在该调用过程之后寄存器的立即值返回。

当 BP 被复制到 SP 时, 由该过程使用的堆栈空间被释放。旧的帧指针现在已复制到 BP, 恢复调用程序帧, 接着的 RET nn 指令按反方向连接, 并删除正退出的过程的堆栈中所压入的任何参数。

47. lgdt 存储器或操作数 (80286/80386)

lidt 存储器或操作数

把存储器内容装入 GDT 或 IDT。它们常出现在操作系统软件中, 而不用在应用程序中。

48. lldt 字或操作数 (80286/80386)

把选择器的内容装入局部描述符表寄存器。它们常出现在操作系统软件中, 而不用在应用程序中。

49. lmsw 源操作数 (80286/80386)

把有效地址装入机器状态字。它们常出现在操作系统软件中, 而不用在应用程序中。50. lock 指令, 操作数类型 (80386)

前缀 lock 使总线锁定, 以待下一个指令的完成。指令 bt, bts, btr, btc, xchg, add, or, adc, sbb, and, sub, xor, not, neg, inc, dec 等可加 lock 前缀。如

```
lock xchg lab, al
```

51. lodsb 源串

lodsb ; 送字节

lodsw ; 送字

把 SI 指向的来自源串的这了或字节送入 AL 或 AX 中, SI 则自动变化。如果 DF=0, 则 SI 增 1 (字节) 或 2 (字), 反之则减。

52. loop 短目标 ; 如果 CX <> 0, CX 减 1, 转移

looppe 短目标 ; 如果 CX <> 0, ZF=1, CX 减 1, 转移

loopne 短目标 ; 如果 CX <> 0, ZF=0, CX 减 1, 转移

loopnz 短目标 ; 如果 CX <> 0, ZF=0, CX 减 1, 转移

用 CX 计数器控制循环,目的操作数必须是在该指令之前的 128 个字节到该指令之后的 127 个字节范围之内。

53. **lsl** 段界限,选择器 (80286/80386)

装入段界限,如装入成功, ZF=1。

54. **ltr** 源操作数 (80286/80386)

装入任务寄存器,它是操作系统指令,在应用程序中不出现。

55. **mov** 目的,源

把源复制到目的中。

56. **movs** 目的串,源串

movsb ,传送字节

movsw ,传送字

把存储在 [SI] 中的字节或字传送到 ES:[DI] 中。在执行指令之后,如 DF=0(执行了 **cld** 指令),则寄存器 SI 和 DI 增量,如 DF=1(执行了 **std** 指令),则寄存器减量。增减 1 或 2 按字节或字而定。目的串必须用 ES 寄存器寻址,不能有段超越。对源串可以使用段超越,缺省寄存器为 DS。

57. **movzx** 目的寄存器,寄存器/存储器 (80286/80386)

movsx 目的寄存器,寄存器/存储器

以字节或字形式读有效地址或寄存器的内容,将其值零扩展 (**movzx**) 或符号扩展 (**movsx**) 到寄存器属性长度 (16/32 位),并将结果存放在目的寄存器中。

58. **mul** 源

无符号乘法。源可以是指定的字节或字。 $AX = AL * \text{指定的字节}$ 如 AH=0,则 CF=0,OF=0;或者, $DX:AX = AX * \text{指定的字}$,如 DX=0,则 CF=0,OF=0。

59. **neg** 目的操作数

用寄存器或内存值的二进制补码取代其原值。用 0 减去目的操作数,并把结果返回到目的操作数中,形成 2 的补码。除非操作数为 0 时,将使 CF=0,否则 CF=1。其它标志位也可能变动。

60. **nop**

空操作。一字节指令,占用空间。

61. **not** 目标

逻辑非,使有效地址字节或字的每一位取反,即 0 变 1,1 变 0。标志位不变。

62. **or** 目的,源

逻辑或操作,即除非两者对应位都为 0 时结果相应位为 0,否则总为 1。一个操作数自身相“或”,不会改变操作数的值,但可使 CF=0。

63. **out** 端口,累加器

把字节或字输出到端口。可先将端口号 (0 ~ 65535) 放在 DX 中,然后用本指令输出。

64. **outs** 端口,源串 (80286/80386)

outsb

outsw

把串 (字节或字) [SI] 或 DS:[SI] 输出到端口 DX (不允许用立即数作端口号) 传送之后,源串的值根据 DF=0 (或 DF=1) 增加 (或减少)。

65. **pop** 目的

从堆栈中弹出字送到目的中。对 16 (或 32) 位操作数,堆栈指针 SP 增加 2 (或 4)。不允许用 CS 作操作数。**pop ss** 指令禁止所有的中断,包括 NMI,直到下一条指令执行完为止。

66. **popa**

弹出所有通用寄存器 (DI、SI、BP、SP、BX、DX、CX 和 AX)。SP 的值被废弃,而不是装入 SP。

67. **popf**

把堆栈顶的字 (或双字) 弹出,并将其值放至标志寄存器。用 SS:SP 指向堆栈顶部,SP 自动增 2 (或

4)。

68. push

可以把 ES、CS、SS、DS、寄存器、立即数压入堆栈，SP 减 2(或 4)。

图 2-5 表示连续执行指令

push ax

push bx

后的堆栈情况。每个指令分两步执行：

(1)将 SP-1 送入 SP,然后把 ah 送入 SP 所指单元；(2)再次将 SP-1 送入 SP,把 al 送入 SP 所指单元。

随着压入堆栈的内容增加，SP 值不断减小，但每次操作完，SP 总是指向堆栈的顶部。堆栈的最大容量即为 SP 的初值与 SS 之间的距离。

执行 pop 指令的过程正好与此相反。

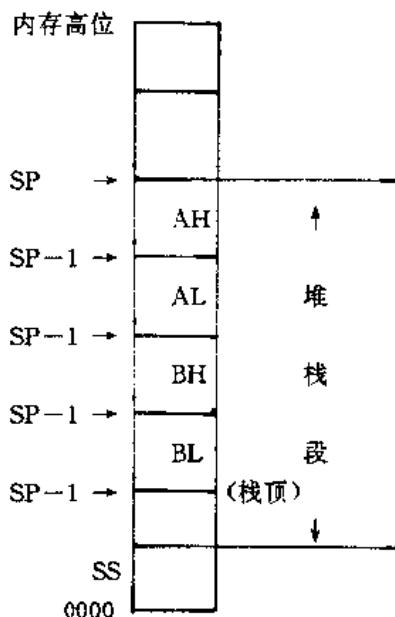


图 2-5 堆栈操作示意图

69. pusha

把所有通用寄存器即 AX、CX、DX、BX、原始的 SP、BP、SI 和 DI 压入堆栈，堆栈指针值减 16(或

32)。

70. pushf

把标志寄存器压入堆栈，SP 减 2(或 4)。

71. 循环移位指令

rci 目的,1 ;左移 1 位

rci 目的,CL ;CL 确定左移次数

rci 目的,计数 ;80286/80386 不允许使移位计数大于 31,如大于 31,则只使用该移位计数的低 5 位。

rcr 目的,1 ;右移

rcr 目的,CL

rcr 目的,计数

rci 把进位标志移进最低有效位,而最高有效位成为进位标志,其余各位左移一位向 左移一位。rcr 则完成相反的传送(CF 中原有内容传送到操作数的最高位,而操作数的最低位送入 CF 中)。

rol 目的,1 ;各位左移 1 位,最高有效位移入 CF 并循环到最低有效位,即移出的最高位值代替最低有效位上的值。

rol 目的,CL

rol 目的,计数

ror 目的,1 ;右移 1 位,最低有效位移出并循环到最高有效位,CF 等于移出的最低有效位上的值。

ror 目的,CL

ror 目的,计数

72. rep 相等时重复

;重复前缀,Turbo C 嵌入汇编允许使用

repz 相等时重复 ;用于 movs,stos,ins,outs 等指令
 repe 相等时重复
 repne 不相等时重复 ;用于 scas,cmps 指令
 repnz 不相等时重复

只要 CX 寄存器不为 0,就重复地进行跟着原始串操作。处理器执行的步骤:

(1) 检查 CX,若 CX=0 则退出,转移到下一条指令。

(2) 执行一次串操作。

(3) 如 DF=0,SI/DI 递增,否则递减。

(4) CX 减 1,标志不改变。

(5) 对 scas 或 cmps 指令,检查 ZF,若重复条件为假便退出。退出时 repe 或 repz 设置标志,或用 repne 或 repnz 清除标志。

(6) 转到下一个循环。

73. ret

返回到 near 或 far 调用程序。它把堆栈顶部的两个字节传送给指令指针,这两个字节提供了要执行的下一条指令的偏移地址。对段间 ret 还从堆栈中把下两个字节弹入 CS,它是要执行的下一条指令的代码段地址。在返回调用处后可将常数加到 SP 而释放堆栈,非 C 的语言大都以此来由堆栈中释放掉参数。C 则利用调用函数来释放参数,故其参数的数目可以是变动的。

74. sahf

将 AH 的值装入标志寄存器 (FLAGS 的低 8 位) 中。

75. sal 目的,计数 ;算术左移,使最高有效位移进 CF,而最低有效位用 0 补 sal 目的,1

sal 目的,CL

shl 目的,计数 ;逻辑左移,它同算术左移

shl 目的,1

shl 目的,CL

sar 目的,计数 ;算术右移,使最低有效位移进 CF,而最高有效位不变

sar 目的,1

sar 目的,CL

shr 目的,计数 ;逻辑右移,使最低有效位移进 CF,而最高有效位用 0 补

shr 目的,1

shr 目的,CL

只有在执行单次移位指令时 OF=1。对左移,如最高有效位与进位标志相同(表示移位后的符号位与移位前的相同),则 OF=0,否则为 1;对 shr 若 OF=0 表示移位后符号位未变。

76. sbb 目的,源

用目的减去源与进位标志的和数(称带借位减),并把结果存入目的中,相应标志被置位。

77. scas 目的串

scasb

scasw

从 AL 或 AX 中减去由 ES:DI 指向的存储器字节或字操作数,其结果被忽略,而相应的标志被置位。不允许段超越。比较之后,如 DF=0,则 DI 增加 1 或 2;否则便减少。

78. setxxxx 目标 (80386)

条件地设置字节指令。当条件(在括号中)满足时目标(8 位)寄存器或存储器被设置为 1,否则设置为 0。在它前面常有 cmp 或 test 指令。

seta ;高于 (CF=0,ZF=0)

setae ;高于或等于 (CF=0)

setb ;低于 (CF=1)

setbe ;低于或等于 (CF=1 或 ZF=1)

setc	;有进位 (CF=1)
sete	;等于 (ZF=1)
setg	;大于 (ZF=0 或 SF=OF)
setge	;大于或等于 (SF=OF)
setl	;小于 (SF<>OF)
setle	;小于或等于 (ZF=1, SF<>OF)
setna	;不高于 (CF=1)
setnae	;不高于或等于 (CF=1)
setnb	;不低于 (CF=0)
setnbe	;不低于或等于 (CF=0, ZF=0)
setnc	;无进位 (CF=0)
setne	;不等于 (ZF=0)
setng	;不大于 (ZF=1 或 SF<>OF)
setnge	;不大于或等于 (SF<>OF)
setnl	;不小于 (SF=OF)
setnle	;不小于或等于 (ZF=1, SF<>OF)
setno	;无溢出 (OF=0)
setnp	;奇偶性为奇 (PF=0)
setns	;无符号 (SF=0)
setnz	;结果非零 (ZF=0)
seto	;溢出 (OF=1)
setp	;奇偶性为偶 (PF=1)
setpe	;奇偶性为偶 (PF=1)
setpo	;奇偶性为奇 (PF=0)
sets	;有符号 (SF=1)
setz	;零 (ZF=1)

79. sgdt 目的 (80286/80386)

sidi 目的

把 GDT 或 IDT 寄存器的内容保存在存储器中。它常出现在操作系统软件中,而不出现在应用程序中。

80. shld 寄存器/存储器,寄存器,[I/L/立即数] ;左移 (80386)

shrd 寄存器/存储器,寄存器,[CL/立即数] ;右移,CL 包括移位数

双精度移位指令。第一个操作数(最左边)包含要移位的值,是 16 或 32 位寄存器;第二个操作数包含要移成这个值的位,它必须是与第一个操作数有同样大小的寄存器;第三个操作数包含移位的位数。方括号表示任选。对 shld,高位位从寄存器操作数移到右边(低位部分);对 shrd,低位位从寄存器操作数移到左边(高位部分)。

```
mov ax,3AF2H ;AX=00111010 11110010
mov bx,9C00H ;BX=10011100 00000000
shld ax,bx,7 ;AX=01111001 01001110 (794EH)
```

81. sldt 目的 (80286/80386)

把 LDT 存储在有效地址字中。它常出现在操作系统软件中,而不出现在应用程序中。

82. smsw 目的 (80286/80386)

把机器状态字存储在有效地址中。它常出现在操作系统软件中,而不出现在应用程序中。

83. stc

设置进位标志为 1,或称 CF 置位。

84. std

置方向标志为 1。

85. sti

置中断标志为 1。

86. stos 目的串

stosb

stosw

把 AL 或 AX 中内容传送到由 ES:DI 指向的存储器字节或存储器字中。不允许段超越。如 DF=0, DI 自动增 1 或 2,反之则减。

87. str 目的 (80286/80386)

把任务寄存器的内容保存在有效地址字中。它常出现在操作系统软件中,而不出现在应用程序中。

88. sub 目的,源

整数减法。目的减源后的差放在目的中,各标志位相应置位。

89. test 目的,源

将两操作数进行“与”操作,但比较结果并不使用,它不改变两个操作数的值,只是将相应的标志修改,其中 OF=0,CF=0。一般它后面跟像 jxxxx 条件转移指令。

90. verr 可读的目的选择器

verw 可写的目的选择器

校验段是否可读写。如果段是可读写的,就置 ZF=1,否则为 0。

91. wait

挂起(暂停)CPU 指令的执行,直到 80x87 协处理器处理好上一指令,以便执行下一相关指令。标志未变。

92. xchg 目的,源

将两个操作数(字节或字)交换。

93. xlat [转换表]

完成查找字节的转换。AL 是由 DS:BX 寻址的无符号表索引(变址)。由 DS:BX 检索的字节被复制到 AL 中。例

LEA BX,MYTABLE ; 表的开始地址

MOV AL,INDEX ; 进入表的偏移量(表的开始偏移为 0)

XLAT MYTABLE ; 数值在 AL 中返回

94. xor 目的,源

逻辑异或操作。所比较的两位数为 1 或 0 时,结果为 0,否则为 1。操作后,OF=0,CF=0。一个操作数对自身“异或”的结果是使操作数为 0。

2.4 嵌入汇编

汇编代码的高速性和低级控制特性能明显地改进程序的性能。

如果将汇编代码直接嵌入 C 语言源程序中,这就是【嵌入汇编】。嵌入汇编可将汇编代码放入 C 程序的任何位置,并可全面访问 C 语言的常量、变量甚至函数。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 C 和汇编之间的接口。

嵌入汇编语句格式

■ e 指令,但那样编译程序一遇到汇编语句就会重新开始。使用选择项或指令的目的只是为了节省编译时间。

在嵌入汇编语句里使用 80186 指令助记符时,必须包括 -L 命令选择项,它使得编译程序在其生成的文件中加入适当的语句,以使得 Turbo 汇编能够识别这些助记符。

5. 嵌入汇编并非完整的汇编程序,所以许多错误不能直接得到检测,这可由 TASM 弥补。但由于 C 语言行号已丢失,所以 TASM 不能标明错误位置。

6. 在嵌入汇编中无须像汇编语言中考虑参数偏移量、标识符的拼写等一些问题,因此为程序员带来了方便。

在 asm 语句中可使用 C 符号(包括自动或局部变量、寄存器变量和函数参数在内的任何符号),Turbo C 会自动转换成汇编语言操作数,并在标识符前增加下划线。

如果嵌入式汇编码中使用了函数名,则函数名前必须以下划线为前缀。

一般说来,C 符号可用以地址操作数出现的地方,而寄存器变量用在寄存器是合法操作数的场合。

函数可以用 register 关键字修饰频繁使用的两个变量为寄存器变量,其它的变量则作为自动变量。如指定的寄存器变量不起作用时,则忽略 register 的作用。

只有 short、int(和相应的 unsigned 类型)或 2 字节指针变量可以作为寄存器变量。SI 和 DI 是用作存放寄存器变量的寄存器。如果函数没有使用寄存器变量,嵌入汇编可以自由使用它们作为暂存寄存器。C 函数的出口、入口代码自动保存和恢复调用者的 SI 和 DI 的数值。

如函数中已说明了一个寄存器变量,嵌入汇编代码可以通过使用 SI 和 DI 以改变或引用寄存器变量,但最好的办法是直接使用 C 语言符号,防止寄存器变量在内部实现时有变化。

虽然对嵌入汇编直接引用标识符就能得到正确的偏移量,但是,在汇编指令中必须使用 word ptr、byte ptr 等其它长度转换符。

在嵌入汇编语句中,假定已定义了整型变量 k(int k),则形如

```
asm mov k,5
asm inc k
```

那样的语句应写成

```
asm mov WORD PTR k,5
asm inc BYTE PTR k
```

7. 在嵌入汇编中还允许使用文字量直接插入到源程序中去(参见 emit() 函数)。

8. 程序段

```
struct my{          /* 定义了一个结构 */
    int a;           /* 整型变量占 2 个字节 */
    int b;
    int c;
}myA;
myfun()              /* 结构的成员名可作为数值常量使用 */
{
    .....           /* 结构的成员名可出现在任何数值常量可出现的地方 */
    asm mov ax,myA.b  /* 把 myA.b 的值赋给 AX */
    asm mov bx,[di].c /* 把 [di]+myA.c 的偏移量赋给 BX */
    .....           /* 这两个汇编语句将产生下述代码: */
                      /* mov ax,DGROUP,myA+A */
                      /* mov bx,[di+4] */
}
```

如果两个结构使用相同成员名,为区别起见,应在成员名前加上结构的类型,并用括号括起。如

```
asm mov bx,[di].(struct tmm)tmm—a
```

9. 程序段中有跳转指令或循环指令是允许的,如

```

int x()
{
    lable,      /* C 的 goto 标号 */
    .....
    asm jmp lable /* 由于规定嵌入汇编必须以 asm 开头,故它不能定义标号 */
    .....      /* 不能产生直接的 far 跳转指令。对允许的间接跳转,可 */
}              /* 以使用寄存器名作为跳转指令的操作数          */

```

但它们只能在函数内部起作用。

10. 在用户编写的任何嵌入式汇编代码的末尾,应使寄存器 BP、SP、CS、DS 和 SS 与进入此嵌入式代码时的值一致。

11. Turbo C 嵌入汇编允许使用的几个汇编指令

(1) 变量名 db 表达式

它定义一个变量或表,以便初始化一个存储单元。它保留字节宽度(8位)的存储单元。

例

```

HEL    DB 23
MY     DB 'S'
LON    DB 'PRON NN'
PERSS  DB 0,1,2,9
HOW    DB 45 DUP('STACK')
AR     DB 50 DUP(03CH)
YOU    DB 'YES',35,0FADH

```

(2) 变量名 dd 表达式

定义双字。

例

```

HLL    DD 0FFFFFFFFH
TEE    DD 0H
YOU    DD 77,66,444,555
LGI    DD 45H+23H
PRESS  DD 2.1345
HOW    DD 4.7E12
ARR    DD 50 DUP(03CH)
TYYZ   DD 0.34E-2

```

(3) 变量名 dw 表达式

定义字。

例

```

TRE    DW 0ABCDH
MMNN   DW 50 DUP(?)

```

(4) extrn 符号或变量名

符号或变量名在当前程序模块中声明为 extrn,而在另一个程序模块中声明为 PUBLIC。extrn 可以放在包含该程序模块的段内或者放在所有的段外,其类型可以是 BYTE、WORD、DWORD、QWORD、TBYTE、NEAR、FAR 或 ABS。

例

```

MYCODE SEGMENT PARA 'CODE'
        PUBLIC V1,VAL2
MYPROC PROC FAR

```

```

ASSUME CS:MYCODE,SS,STACK
PUSH DS
.....
EXTRN V1,VAL2
NUCODE SEGMENT PARA 'CODE'
.....

```

2.5 Turbo C 的六种存储模式

提供多种存储模式 (memory model) 的目的主要是为了使用户能够根据程序处理数据和执行代码的多少合理地使用内存,减少不必要的开销。

一 存储模式选用参考

表 2-1

数据大小 (DATA)	代码大小(CODE)	
	64KB	1MB
64KB	微模式:代码数据重迭,总长 64KB	模式:小数据,大代码
	小模式:代码数据不重迭,总长 128KB	
1MB	紧凑模式:大数据,小代码	大模式:大数据,大代码
		巨模式:大数据,大代码 (但静态数据大于 64KB)

二 存储模式分类

微模式、小模式和紧凑模式称为【小代码模式】;而中模式、大模式和巨模式称为【大代码模式】。微模式、小模式和中模式称为【小数据模式】;而紧凑模式、大模式和巨模式称为【大数据模式】。

三 六种存储模式下 8086 内存分配情况

代码存储区为执行程序转换产生的机器指令。注意:当编译一模块即带有一些子程序的源程序时,所产生的代码不能超过 64KB,因为它们将装入同一个代码段,而代码段只能装 64KB。从它们的内存结构看出,即使对大代码模式(中模式、大模式和巨模式)也是如此。倘若模块太大,无法装入 64KB 代码段,应将该模块的源程序分成几个小一点的源程序,然后分别编译连接。

初始化数据区为全局变量和静态变量存储区。注意:尽管巨模式支持大于 64KB 的静态数据,但各模块的数据仍应小于 64KB。

未初始化区为局部变量存储区。

堆栈区存放函数调用程序相关的内容(函数出入口,暂时用到的局部变量等等)。当堆栈存储量增加时,堆栈指针将指向内存低地址单元。

堆是动态数据区。当堆存储量增加时,它向内存高地址发展。

1. 微 (Tiny) 模式

各段均取相同地址,即 CS=DS=SS; ES= 暂存。代码、数据和数组等总空间为 64KB。全部使用类型为 near 的指针。当内存很小时,可用它。

在这种模式下编译连接后生成的执行文件可以转换成 *.COM 文件。具体步骤是：
在集成环境下，选

Options/Compiler/Model Tiny

编译生成执行文件（例如 model.exe）后，用 DOS 命令（执行 EXE2BIN.EXE 文件）

C>EXE2BIN MODEL.EXE MODEL.COM

便将 model.exe 生成一个 model.com 文件（注意：原 model.exe 仍存在）。如果不是在微模式下编译连接生成的执行文件用此命令时，将出现

File cannot be converted

的错误信息，不能生成 *.COM 文件。比较 model.exe 和 model.com 文件，可以发现后者执行代码小得多。*.COM 文件占用内存少，运行速度快。

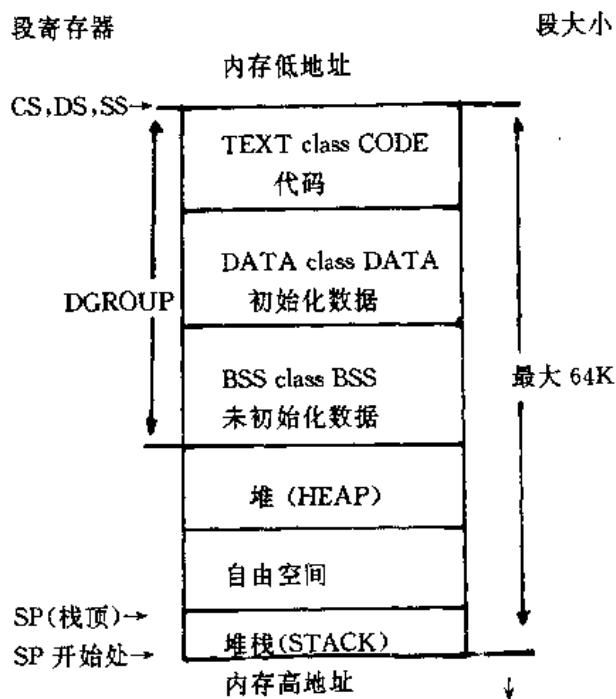


图 2-6 微模式内存结构

2. 小 (Small) 模式

其代码和数据段不同，也不重迭，因而可有 64KB 的代码和 64KB 的静态数据。栈段 (SS) 和数据段 (DS) 都取相同值 (DS=SS)，附加段 ES= 暂存，但 CS ≠ DS。它总使用 near 指针。它是常用模式，也是集成环境使用的缺省模式。

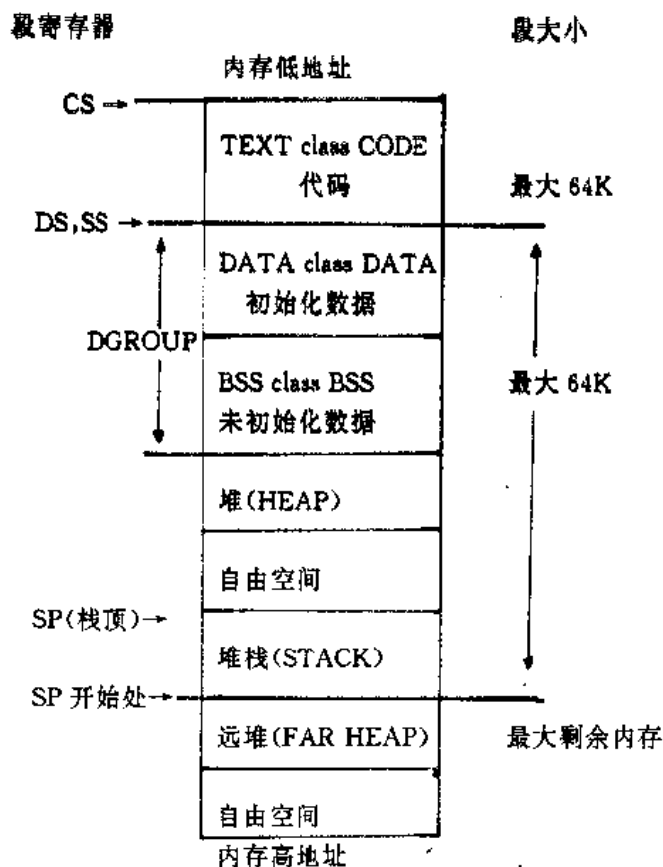
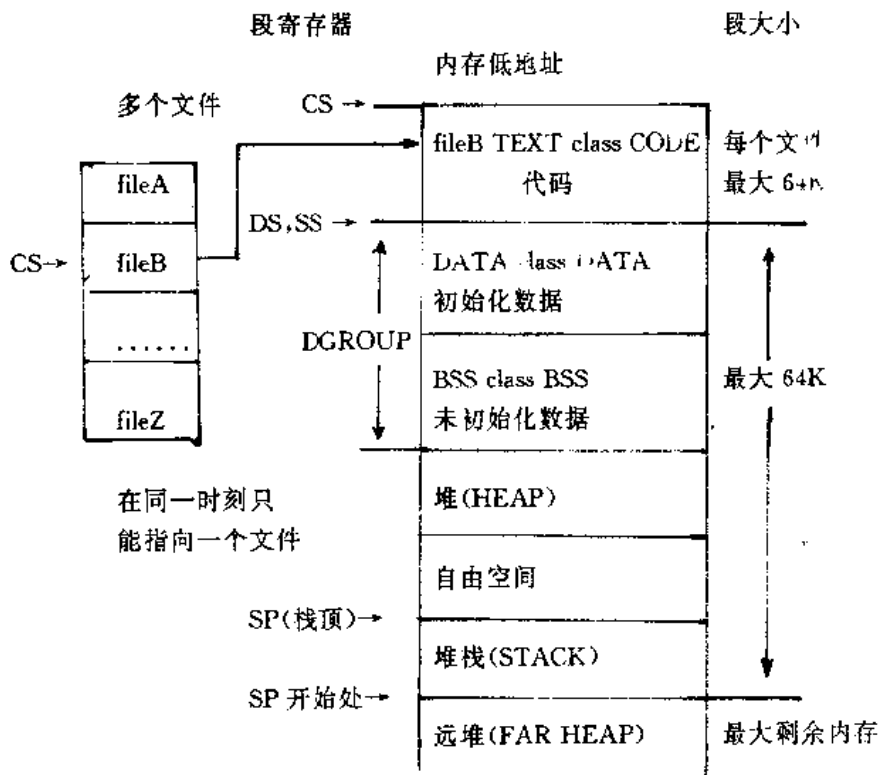


图 2-7 小模式内存结构

3. 中 (Medium) 模式

代码段使用 far 指针, 但数据段不用 far 指针。因而静态数据最多可为 64KB, 而代码最多可达 1MB。适用于数据量较小而代码大的程序。CS 1 = DS, DS=SS, ES= 暂存。



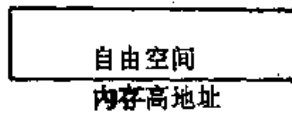


图 2-8 中模式内存结构

4. 紧凑 (Compact) 模式

数据段用 *far* 指针, 而代码段不用 *far* 指针, 代码最多为 64KB, 数据可达 1MB。适用于数据量大而代码短的程序。CS ! = DS ! = SS, ES = 暂存 (各模块有一个 CS)。

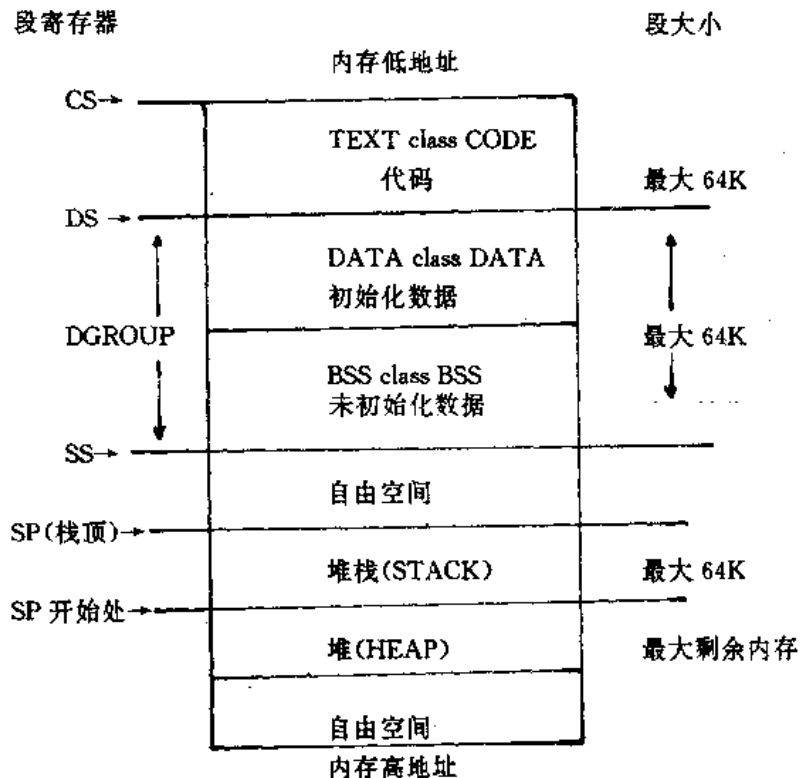
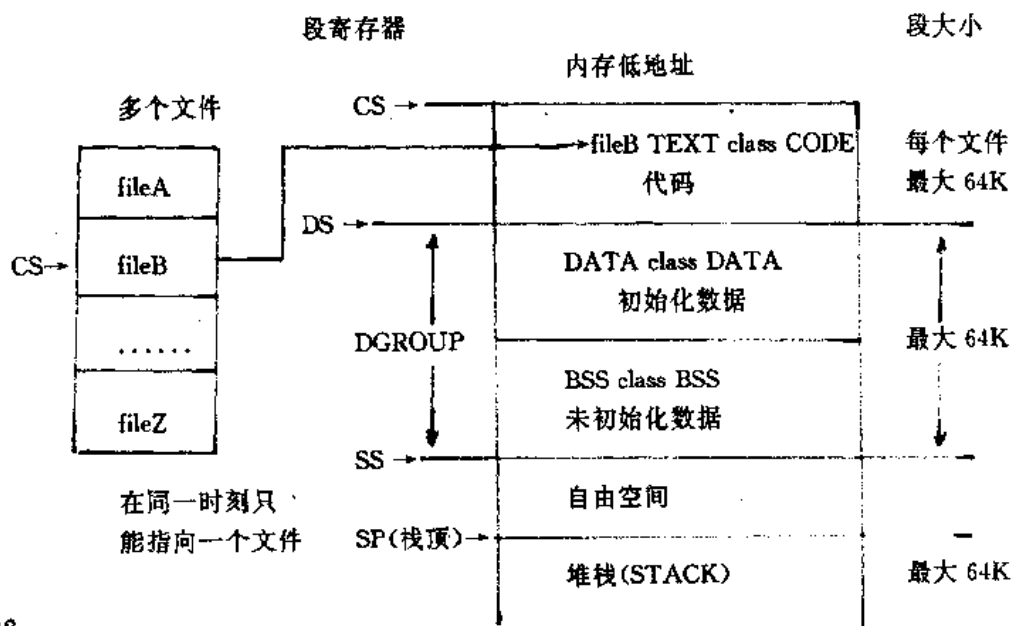


图 2-9 紧凑模式内存结构

5. 大 (Large) 模式

代码和数据均用 *far* 指针, 两者均可达到 1MB。适用于非常大的应用程序。CS ! = DS ! = SS, ES = 暂存 (各模块有一个 CS)。



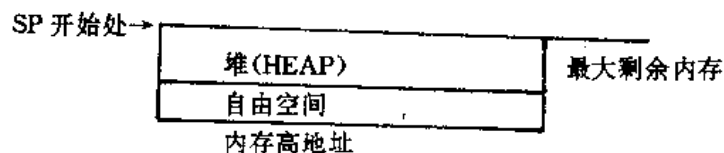


图 2-10 大模式内存结构

6. 巨 (Huge) 模式

代码和数据均用 far 指针。静态数据可多于 64KB, $CS \neq DS = SS$; ES= 暂存 (每个模块有一个 DS 和 CS)。

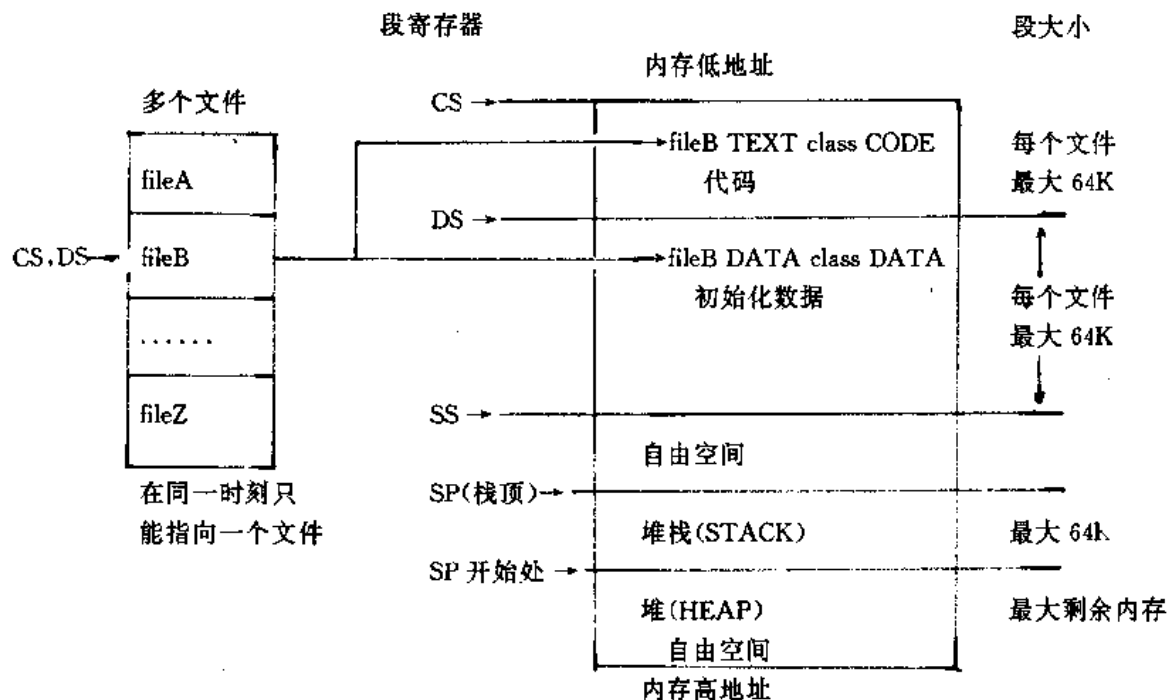


图 2-11 巨模式内存结构

四 和存储模式对应的启动文件和库文件

表 2-2

存储模式 启动文件 80x87 库 数学库文件 运行库文件

微模式	C0T.OBJ	FP87.LIB 或 EMU.LIB	MATHS.LIB	CS.LIB
小模式	C0S.OBJ		MATHM.LIB	CM.LIB
中模式	C0M.OBJ		MATHC.LIB	CC.LIB
紧凑型	C0C.OBJ		MATHL.LIB	CL.LIB
大模式	C0L.OBJ		MATHH.LIB	CH.LIB
巨模式	C0H.OBJ			

如果注意到文件名只有一个字母不同,而该字母即是存储模式英文单词的首字母时,记住它们就很容易了。

在集成环境 (TC.EXE) 或用命令行编译 (TCC.EXE) 时,针对不同的存储模式, Turbo C 会按适当的次

序自动连接适当的库。但如单独使用连接器 (TLINK, EXE) 时, 连接的库应由用户指定, 其中对所选存储模式必须按严格的顺序 (表中已考虑到这种顺序) 连接相应的启动文件和库。除此之外, 当程序中使用浮点子程序时, 如机上装有 80x87 协处理器, 可以连接 FP87.LIB 库。如不用协处理器, 则必须连接 8087 仿真库 EMU.LIB。

图形库只有一个, 即 GRAPHICS.LIB, 它跟存储模式无关。

2.6 汇编程序的伪指令 (摘要)

说明: -I 指 IBM, -M 指 Microsoft 公司, -S 指 Speedware 公司, -A 指 ASM386 汇编程序。

-1.186 -M, -S

允许汇编所有 8086 和非保护方式下的 80186 指令, 它可用 .8086 伪指令取消。

-2.286C -I, -M, -S

允许汇编所有 8086 和非保护方式下的 80286 指令, 它可用 .8086 伪指令取消。

-3.286P -M

允许汇编所有 8086 和非保护方式下的 80286 指令, 它可用 .8086 伪指令取消。

-4.287 -M

允许汇编所有的 80387 指令及

FSETPM ; 设置保护方式

FSTSW AX ; 把状态字保存在 AX 中 (等待)

FNSTSW AX ; 把状态字保存在 AX 中 (不等待)

-5.8086 -I, -M, -S

复位而使用 8088 / 8086 汇编, 此时并不汇编 80286 指令。

-6.8087 -I, -M, -S

允许每个汇编程序对 8087 / 80287 的指令和数据进行汇编。

-7 文本或符号 & -I

这是宏操作符。在字符串中或者没有用分隔符领先的哑变量或参数必须用 & 开头, 以便在宏扩展中替换它。如程序中的宏程序是

```
MY      MACRO    GUESS      ,MACRO 宏程序
RP&T:   MOV DX,01234H
MOV     AX,'&T'
ENDM
```

程序中调用宏的语句是

```
MY      G
```

则上述宏相当于

```
REPG:   MOV     AX,'G'
        MOV     DX,01234H
```

-8 标号=数值 -I, -M, -S

等于伪指令与 EQU 相似, 但前者允许同一标号在定义后还可再用此伪指令重新定义, 而后者不允许。

-9 ! 字符 -I

当惊叹号放在一个字符前时, 允许该字符的文字输入, 它等效于 <字符>。

-10 %符号 -I

它只能用在宏中。跟在百分号后面的符号被转换成以默认的数字基数表示的数字, 在宏扩展时这个数字将取代哑变量。

-11 %OUT 注释字符 -I, -M, -S

在程序汇编时将注释字符在屏幕上显示出来。

—12 ;, 注释 —I,—M,—S

如将语句

```
MY      MACRO  GUESS      ;MACRO 宏程序
```

改成语句

```
MY      MACRO  GUESS      ;,MACRO 宏程序
```

则能节省程序的有效存储空间。

—13 ASSUME 段寄存器,段名,... —I,—M,—S

它使一个段与段寄存器相符。如

```
MYCODE   SEGMENT PARA 'CODE'   ;为宏汇编定义码段
MYPROC    PROC    FRA           ;过程名为 MYPROC
          ASSUME  CS,MYCODE,DS,MYDATA,SS,STACK
          .....
```

有效的段寄存器是 CS、DS、ES 和 SS。

—14 COMMENT 分隔符 注释 分隔符 —I,—M,—S

允许对程序注释时不用分号,这对多行注释特别有用。如

```
COMMENT   @ 注释长度不受限制 @
```

—15 .CREF (或.XCREF) —I,—M,—S

.CREF 在调用交叉参考实用程序时允许输出交叉参考信息的默认条件,而 .XCREF 可以使用一个可选择的操作数来避免已命名的变量在交叉参考输出中出现。如

```
MYDATA    SEGMENT  PARA 'DATA'
ANS        DB      ?
MYDATA     ENDS
.XCREF     ANS
          .....
```

—16 变量 DB 表达式 —I,—M,—S

定义字节伪指令,初始化存储单元。如

```
H1    DB    40
H2    DB    0AH
H3    DB    'KH RR'
H4    DB    4,5
H5    DB    46 DUP('STACK')
H6    DB    54 DUP(04CH)
H7    DB    ?
H8    DB    'YES',34,0FACH
```

—17 变量 DBIT 表达式 —A

初始化一个完整字节。位串必须用字母 B 结束。如

```
FFG    DBIT 100B
```

—18 变量 DD 表达式 —I,—M,—S

定义双字伪指令。

—19 变量 DP 表达式 —I,—M,—S

定义小数点用于定义一个 48 位的 PWORD 整数类型变量。如

AA DP 'ACSDFH' ;最多 6 个字符
 CC DP ?
 WWW DP 33 DUP(0A76543H)

—20 变量 DQ 表达式 —I,—M,—S

定义 4 个字即 64 位的伪指令。

—21 变量 DT 表达式 —I,—M,—S

定义 10 个字节即 80 位的伪指令,为 80x87 协处理器的数字保留 10 个字节组合的十进制存储单元。如

H1 DT 100.2000,6000000
 H2 DT 25 DUP(45.678)
 H3 DT -6.34E-5,+5.002E45
 H4 DT 789*123

—22 变量 DW 表达式 —I,—M,—S

定义 1 个字伪指令。

—23 ELSE —I,—M,—S

它在 IFXX 伪指令之后使用,表示“否则”之意,每条指令占一行。

—24 END 可选择的表达式 —I,—M,—S

在源程序的结尾需要它。如 END 后有表达式,则该表达式就指向这段码的入口点。

—25 ENDIF —I,—M,—S

于 IFXX 条件伪指令之后,表示 IFXX 结束。它占一行。

—26 ENDM —I,—M,—S

表示伪指令 MACRO、REPT、IRP 和 IRPC 终止。它占一行。

—27 过程名 ENDP —I,—M,—S

终止过程,占一行。

—28 段或结构的名称 ENDS —I,—M,—S

终止 SEGMENT 或 STRUC 的伪指令。如

MYCODE ENDS ;结束名为 MYCODE 的码段

—29 变量 EQU 数值 —I,—M,—S

设置变量为确定的值,但它不能重定义,且不能与 STRUC 定义一起使用。

—30 EVEN —I,—M,—S

如果程序计数已在偶数边界,则它不进行任何操作,否则它将增加一个 NOP 操作,以使计数器中的值增 1 后为偶数。

—31 EXITM —I,—M,—S

它与 REPT、IRP、IRPC 和 MACRO 一起使用,从重复操作或以特定条件的伪指令测试结果为基础的宏中退出。允许块的嵌套,EXITM 只结束它所在的块。

—32 EXTRN 符号或变量名 —I,—M,—S

符号或变量在当前程序模块(段)中声明为 EXTRN 时,表明它们在另一个程序模块(段)中早已声明为 PUBLIC。

—33 名字 GROUP 段名 —I,—M,—S

它把一个由该名字标识的段组织在一个实际的 64K 的段内。该标识符在所有的段标号中应是唯一的。段名可用 SEGMENT 伪指令或 SEG 变量或标号操作符指定。

—34 条件伪指令 IFXX —I,—M,—S

IF 操作数 ;如果操作数不等于 0 为真

IFE 操作数 ;如果操作数等于 0 为真

IF1 ;如果汇编程序扫描 1 为真

IF2 ;如果汇编程序扫描 2 为真

IFDEF 符号或变量 ;如果已定义此符号或变量则为真
 IFNDEF 符号或变量 ;如果未定义此符号或变量则为真
 IFB 操作数 ;如果操作数是空白则为真
 IFNB 操作数 ;如果操作数不是空白则为真
 IFIDN 操作数 1,操作数 2 ;如果操作数 1 等于操作数 2 则为真
 IFDIF 操作数 1,操作数 2 ;如果操作数 1 不等于操作数 2 则为真

—35 INCLUDE 驱动器/路径/文件名/扩展名 —I,—M,—S

它在装入 MACRO 的集合 (或库) 时有用。它将另一个文件的未汇编码与当前码相连接。注意, CONFIG.SYS 中设置的文件数应大于 8。如果列出了汇编过的文件, 为易于识别和设置扩展, 来自 INCLUDE 的所有的码都应在第 30 列, 用一个“C”列出。

—36 IRP 哑元, <操作数列表> —I,—M,—S

它用于重复在 IRP 和 ENDM 之间的操作。重复次数是由操作数列表中的操作数的数字控制的。对每一次重复, 操作数列表中的当前项将替换码块中的每一个哑元操作数。如

```
IRP    VAL,<2,4,6,8> ;数字 2, 4, 6, 8 的操作数列表
DW     VAL           ;新的数字替换 VAL
ENDM
```

—37 IRPC 哑元, 串 —I,—M,—S

它用于重复在 IRPC 和 ENDM 之间的操作。重复次数是由操作数中的数字控制的, 对每一次重复, 串中的当前字符将替换码块中的每一个哑元操作数。如

```
IRPC   DATA,2486 ;字符 2, 4, 6 的串
DB     DATA      ;每次通过时新的数字
DB     DATA * 2  ;每次通过时新的数字乘 2
ENDM
```

—38 标号名字 LABEL 类型 —I,—M,—S

定义标号的属性, 类型可以是 BYTE, WORD, DWORD, QWORD 和 TBYTE。

—39 .XALL 列表伪指令 —I,—M,—S

.LALL ;列出所有 MACROS 的宏扩展和条件块 (如果这条伪指令是与 .TF) COND, .LFCOND, .SFCOND 一起使用的)。

.SALL ;取消所有扩展的 MACRO 列表

.XALL ;只列出产生实际目的码的源码

—40 .LFCOND —I,—M,—S

列出任何判断为假的码的条件块, 这个条件可用 .SFCOND 或 .TFCOND 终止。

—41 .LIST —I,—M,—S

允许源和目的码列表。

—42 .XLIST —I,—M,—S

终止列表。

—43 LOCAL 哑元列表 —I,—M,—S

声明哑元列表中的变量为局部变量, 当使用 LOCAL 时, 用一个唯一的符号替换宏内出现的每一个标号。这就允许在一个给定的段内不止一次地使用宏扩展。

```
DELAY    MACRO ;;建立软件延时
          LOCAL P1,P2 ;;声明 P1,P2 为局部变量
          MOV     DX,15H ;;延时约 5 秒
P1:      MOV     CX,0FF00H ;;十进制数 65280
P2:      DEC     CX
```

```

JNZ     P2                ;,不为 0 则转 P2
DEC     DX
JNZ     P1                ;,不为 0 则转 P1
ENDM

```

—44 宏的名字 MACRO 哑元列表 —I,—M,—S

当在一个列表中扩展了宏时,它们的码前边有一个加号,表示出现了一个扩展。哑元列表中各元素的排列位置是很重要的,因为替换是逐项进行的。如果在哑元中的元素在宏体中没有出现,则该元素被忽略。如果包含的元素太少,则用 0 代替。

```

PRINTCHAR MACRO TEXT      ;用传递的“TEXT”显示字符串
MOV        DX,OFFSET TEXT ;字符串在数据段中
MOV        AH,9           ;显示直到“$”字符串
INT        21H            ;从当前光标位置开始显示
ENDM                    ;宏结束

```

—45 .MSFLOAT —S

将所有浮点数都转换为 Microsoft 的浮点格式。

—46 NAME 模块名 —I,—M,—S

命名一个模块。每个模块按下列优先顺序给出名字:

```

NAME
TITLE 的前 6 个字符
源文件名的前 6 个字符

```

—47 ORG 数值或表达式 —I,—M,—S

设置单元计数器从而包含了码的起始地址。美元符号 \$ 可以用作单元计数器的当前值。表达式在扫描 1 时必须是已知的,而且必须等于 16 位的绝对数。

```

ORG 100H
ORG $
ORG $+560H

```

—48 PAGE 行数,页宽 —I,—M,—S

在源文件中设置列表页的长度和宽度。当遇到 PAGE+ 时,章号加 1。该指令并不初始化打印机。

```

PAGE          ;进到下一页
PAGE ,132      ;改默认页宽 80 为新值 132 (许可范围 60~132)
PAGE 20,132    ;改默认页长 58 为新值 20 (许可范围 10~255)

```

—49 过程名 PROC 类型 —I,—M,—S

标识一个源码块(过程)。所有程序至少有一个带有类型为 FAR 的 PROC 伪指令。如果使用外部调用,则 PROC 的过程名必须是 PUBLIC。如果 PROC 是一个 .EXE 文件的入口点,或者 CS 包含有其它数值,则类型必须是 FAR。如果在同一个段内调用 PROC,类型只能是 NEAR。根据类型的不同,RET 将是段返回或段间返回。

—50 PUBLIC 数字、变量或标号 —I,—M,—S

其后跟的数字、变量或标号能传送给其它程序。

—51 PURGE 宏名 —I,—M

删除一个宏定义,从而可以重新使用这个宏原来所占用的空间。

—52 .RADIX 十进制基数值 (2~16,默认为 10) —I,—M,—S

它直接影响 DB 和 DW,但不影响 DD、DQ 或 DT。如

```

.RADIX 16

```


VALUE DB 120 ;VALUE 被设置为十六进制数 120

—53 记录名 RECORD 场名,宽度 —I,—M,—S

用位组合格式化字节和字。记录名为对存储器分配的伪指令。

场是由多个二进制位组成的,并是以右边对齐的,宽度则是场包含的位数(1~16)。

如

MYRECD RECORD H,1,E,2,L,3,P,6

存储器分配为

HELLLLPPPPPP

—54 REPT 数值或表达式 —I,—M,—S

按指定的次数(其后的数值或表达式)重复 REPT 和 ENDM 之间的码。码块不必是在 MACRO 之内。

—55 段名 SEGMENT 定位类型 组合类型 class —I,—M,—S

定位类型:

1. PAGE

使一个段起始于页的边界,即一个能被 256 整除的地址,该地址的最低两位等于 00H。

2. PARA

使段起始于一个段落的边界,该地址能被 16 整除,最低有效位为 0。

3. WORD

使段起始于字边界,即偶数地址,最低有效位等于 0。

4. BYTE

使段起始于字节边界,可以在任何一个地方。

组合类型:

1. PUBLIC

表示连接时该段将与另一段连接。

2. COMMON

表示该段和要被连接的其它同名的段将起始于同一地址。

3. AT

表示该段将开始于由表达式确定的段落的边界。

4. STACK

表示该段在运行时是堆栈的一部分,只有 .EXE 文件需要 STACK。

例

```
STACK      SEGMENT  PARA  STACK
            DB          64 DUP ('MYSTACK')
STACK      ENDS
```

```
MYDATA     SEGMENT  PARA  'DATA'
INFO       DB          1,2,3
ANS        DB          ?
MYDATA     ENDS
```

```
MCODE      SEGMENT  PARA  'CODE'
PUBLIC     V1,V2,SUM
MPROC      PROC      FAR
ASSUME     CS,MCODE,DS,MYDATA,SS,STACK
PUSH       DS
```

.....
—56 .SFCND —I,—M,—S

终止判断为假的条件码块的列表。

—57 结构名 STRUC —I,—M,—S

和 RECORD 相似,只是它具有多字节能力。

```
MYSTRUC    STRUC
H1          DB       8           ,单项可被替换
H2          DB       1,2,4       ,多项不可替换
H3          DB       'SU er'     ,字符串可被替换
MYSTRUC    ENDS
```

.....

BLUE MYSTRUC <4,, 'SU enter'> ;取代 H1 和 H3

—58 SUBTTL 串 —I,—M,—S

允许在紧接着 TITLE 的下面一行列出子标题。串最多为 60 个字符。在一个程序内允许有多个 SUBTTL 出现,但其后串不存在时并不引起跳页。

—59 .TFCOND —I,—M,—S

终止 .SFCND 和 .LFCOND 的操作。

—60 TITLE 串 —I,—M,—S

允许在每个列表页的第二行输入一个标题,串最多有 60 个字符。如

```
TITLE       DISK-FORMATINNG
```

第三章 关键字和语句

TC 的关键字共 59 个,它们有固定的含义,因此程序员在使用它们的时候必须照原样书写,也不得将它们单独作为其它标识符用。《K&R》(指 Brian W. Kernighan 和 Dennis M. Ritchie 合写的《C 程序设计语言》一书)中的两个关键字 entry 和 fortran, TC 未使用。关键字后面括号内的内容含意是,(AN) 指由《ANSI》(美国国家标准化研究所)所扩充的;(TC) 指由 TC 本身扩充的。

3.1 分类

一 变量存储类

. auto	自动的	—2
. extern	外部的	—14
. register	寄存器变量	—26
. static	静态的	—31

二 数据类型

. void	空类型(TC)	—37
. int	基本整型	—21
. short	短整型	—28
. long	长整型	—23
. float	单精度实型	—16
. double	双精度实型	—11
. signed	有符号的(TC)	—29
. unsigned	无符号的	—36
. char	字符型	—6
. enum	枚举(AN)	—13
. struct	结构	—32
. union	联合	—35

三 语句

. break	从循环中退出	—3
. return	函数返回	—27
. continue	继续循环	—8
. goto	转向标号	—18
. do	do 循环	—10
. for	for 循环	—17

. while	while 循环	—39
. if	如果	—20
. else	否则	—12
. switch	开关	—33
. case	某一开关值	—4
. default	其余开关值	—9

四 指针修饰

. near	近指针 (TC)	—24
. __cs __ds __es __ss	相对段寄存器的 near 指针 (TC)	—56~—59
. far	远指针 (TC)	—15
. huge	规格化指针 (TC)	—19

五 语言修饰符

. cdecl	C 标识符 (TC)	—5
. pascal	Pascal 标识符 (TC)	—25

六 伪变量 (全为 TC)

. __AX __AH __AL	—40~—55
. __BX __BH __BL	
. __CX __CH __CL	
. __DX __DH __DL	
. __BP __SP	
. __SI __DI	

七 其它

. const	常量符 (TC)	—7
. volatile	随时可变符 (TC)	—38
. typedef	自定义数据类型	—34
. sizeof	求占用内存字节数	—30
. asm	嵌入汇编语句开始符 (TC)	—1
. interrupt	中断函数修饰符 (TC)	—22

3.2 详细说明

以下按关键字的字母顺序排列先后。

—1 asm

asm 加在汇编语句前面构成【asm 嵌入汇编语句】(简称【asm 语句】),这种语句被允许放在 C 源程序中间。例如

```
int min(int v1,int v2)
{asm mov ax,v1
asm cmp ax,v2
```

```

asm jle minexit
asm mov ax,v2
minexit:
return(ax);
}

```

注意：C 源程序中使用了汇编语句后必须用 TCC.EXE 编译。

—2 auto

用它说明的变量为自动变量（通常是指函数的局部变量）。在程序中常一般省略不写。

—3 break

break 语句常出现在 while，do-while，for 或 switch 等语句的循环语句体内，它被执行时程序将终止循环，转去执行循环语句体后面的语句。这就是说有时不等循环结束，就可用 break 语句从循环中间退出来。

C>TYPE BREAK1.C

```

main()
{
float x,y=20.4;
printf("输入一个数\n");
scanf("%f",&x);
for(;;)
{
x++;
if(x>y)break;
}
printf("%f\n",x);
}

```

在 if...else... 组成的语句中或在函数的主体中使用它时要注意。

如

C>TYPE BREAK2.C

```

main(){
int i=0;
if(i==0)break; /* break 未用在循环中 */
printf("ok! \n");
}

```

将出现

Misplaced break in function main

的错误。

另外，要注意使用了 break 语句后程序执行的新位置。

例如

```

for(.....)
{
    while(.....)
    {
        if(.....)break; /* break 只退出了 while 循环,但没退出 for 循环 */
        .....
    }
    .....
}

```

—4 case

case 语句用于 switch 语句构成的循环体内,其一般形式是

case <常量表达式>;

常量表达式计算后的结果必须是整型值,而且是一个常量。该值通常应是语句

switch (表达式)

中的表达式可能出现的值之一。

每个 case 关键字之后只能有一个常量,且其值是唯一的。

从某种意义上看,case 也可以看成标号。不过,程序是从配对标号开始执行,直到发现 break 语句或 switch 的结尾为止。这时,与每个标号有关的语句不是复合语句,而是语句段。因此,在 case 之后不应出现变量定义语句。

如

```

.....
switch(x)
{
case 1:
    int y;
    .....
}

```

而

```

.....
switch(x)
{
    int y;
case 1:
    .....
}

```

是可以的。相当于在 switch 复合语句开始定义局部变量。

C>TYPE CASE. C

#define YY 12

```

main()
{
int const y=12;
int x=10;
switch(x)
{
case YY; printf("%d\n",x);break; /* 如 x=YY,则打印 x 值后退出 */
/* case y;printf("%d\n",x);break; 不允许,因 y 仍被看成变量 */
/* case 12;printf("%d\n",x);break; 不允许,因 YY=12,重复了 */
printf("no!"); /* 警告:Unreachable code,此句将不被执行。如抑制此警告,则程序无任何输出。正确的做法应将该句纳入 case 或 default 等语句之后。 */
}
}
—5 cdecl

```

cdecl 即 C declaration(C 说明)的意思。编译时如使用选项 `-p` (或在集成环境下用 `O/C/Codegeneration/Calling convention Pascal`),用户可以将程序中所有全局标识符变成 Pascal 类型。但是,有时需要将某些标识符仍然保持它原先的形式,特别是其它文件(如库文件)中所定义的 C 标识符(头部带有下划线),那么可以在源文件中对这些标识符前加上 `cdecl`,这样可以摆脱 `-p` 选择项的约束,把任一调用函数作为正规的 C 函数处理。

如

```

int cdecl FileCount; /* 定义变量 */
far cdecl HisFunc(int x); /* 定义函数 */

```

又例如在编译某程序时要使用 `-p` 选择项,但又想使用 `printf()` 函数,则可以写

`C>TYPE CDECL.C`

```

extern cdecl printf();
putnums(int i, int j, int k);
cdecl main()
{
putnums(1,4,9);
}

putnums(int i,int j,int k)
{
printf("And the answer are,%d, %d, and %d\n",i,j,k);
}

```

该修饰符保证用户程序即使使用了 `-p` 选择项(注意用 `TCC.EXE` 编译时使用的运行时库中的所有函数均需 `cdecl` 说明),编译时也能使用定义的 C 标识符。例如,标头文件 `stdio.h` 中所有函数均为 `cdecl` 类型,因而即使用了 `-p` 选项,仍能与库子程序(都用了 `cdecl` 说明)连接。

注意:当它用于函数头部时,它将通过规约说明 C 类型参数。也即函数表中最后一个参数最先压入堆栈,而调用者将整理堆栈。

附注:TC 允许调用其它语言编写的子程序,反之亦然。程序涉及这种应用时应特别注意两个问题:标识符与参数传递(参见《函数》一章)。

编译一个 TC 程序,程序中所有全局标识符即函数名和全局变量名均被保存在为使用连接的目标码文件(*.OBJ)中,它们以最初的形式存放(小写、大写或大小写混合形式)。若没有选用 -u- (或 O/C/C/Generate underbars 为 Off), TC 必定在标识符前加上下划线。

—6 char

字符数据类型,占 1 字节内存。它和 signed、unsigned 构成两种字符类型:

1. signed char ch; 或即 char ch;

一般用 char ch; 语句,它定义标识符 ch 是一个带符号的变量,其值范围为 -128~127。

2. unsigned char ch;

定义了一种不带符号的字符变量,取值范围为 0 ~ 255。

C>TYPE CHAR1.C

```
main()
{
    int k,x,y;
    x=0xFF81;
    y=(unsigned char)0xFF81;
    printf("%d %d\n",x,y);
} /* 程序输出:-127 129 */
```

—7 const

const 修饰符在 Turbo C 库函数中应用较多,如

```
char * —Cdecl asctime(const struct tm * tblock);
char * —Cdecl ctime(const time_t * time);
int —Cdecl printf(const char * format,...);
void * —Cdecl bsearch(const void * key,const void * base,size_t nelem,
    size_t width,int —Cdecl (* fcmp)(/* const void *,const void * */));
```

等等。const 修饰符在不少 C 语言参考书中没有提到。const 的作用是什么,以及如何在用户自己的程序中正确使用它是我们所关心的主要问题。

const 主要是指出向目标赋值的方法,如增值、减量,以及修改许可方面的约束。

1. const 可以修饰多种数据类型,包括修饰指针变量和非指针变量,也可以修饰结构等。const var; 相当于 const int var; ,它把变量 var 处理为一个常量。因此,不能对它再进行增减值或重新赋值。并且,这种常量又跟用宏定义的符号常量有所不同(参见程序 CONST1.C)。

C>TYPE CONST1.C

```
#include "stdio.h"
#define PVAR      printf("%d %f\n",var,pi)
#define VAR 80
/* #define [VAR] 80 此句不行,产生 Error:Define directive needs an
```



```

        identifier,即 define 指令必须有一个标识符,而 [VAR] 不是 */
main()
{
    const var=80;
    const float pi=3.14159;
    /* char y[var]; 这句不允许,会产生 Error:constant expression required */ char y
[VAR];
    /* var++; 这句不允许,产生 Error:Cannot modify a const object */
    /* var +=1; 或 var=81;pi=3.141592653; 等句重新赋值都不允许 */
    /* VAR++; 这句不允许,产生 Error:Lvalue required */
    PVAR; /* 输出:80 3.141590 */
    * (int *) &var=81; /* 可以通过指针间接修改 */
    PVAR; /* 输出:81 3.141590 */
}

```

2. const char *s 或 char const *s 说明 s 是一个指向一个常量字符串的指针。它虽然在增值、减量或修改方面跟 char *s 没有什么不同 (即是允许的),但是使用它等于明确告诉用户:在函数内你不要对它进行增值、减量或修改操作,否则会带来意外效果! 值得指出的是, Turbo C 不会在编译时对这种操作像对数组那样作出反映,在程序执行时也不阻止这种操作。换言之,这主要靠程序员自己把握好这一环节 (参见程序 CONST2.C、CONST3.C 和 CONST4.C)。

对于库函数,由于用户只要在自己的源程序中包含所用函数的原型就可直接调用它,用户不可能对库函数参数修改,因此对 const 关心是不重要的;然而,用户如果在自己编写的源程序中使用 const,便应注意这一问题了。

C>TYPE CONST2.C

```

#include "stdio.h"
void m(const char *s,char const *t);
main()
{
    const char *s="\n%s,看看指针调用后的变化\n";
    const char *t="这是一个试验";
    m(s,t); /* 注释中为调试表达式的值 */
    printf(s,t); /* s,p: DS:01E4 */
} /* t,p: DS:01EF */
/* 程序输出:ffff
这是一个试验,看看指针调用后的变化 */

```

```

void m(const char *s,char const *t)
{
    /* s,p: DS:01E4 */
    char *f="ffff"; /* t,p: DS:01EF */
    char h[80]="HH"; /* &h[0],p: DS:FF86 */
    s="ok!"; /* s,p: DS:0211 */
    t="no"; /* t,p: DS:0215 */
    s=h; /* s,p: DS:FF86 */
}

```

```

t=h;          /* t,p:    DS:FF86 */
s++;          /* s,p:    DS:FF87 */
t++;          /* t,p:    DS:FF87 */
s="as";       /* s,p:    DS:0218 */
t="qw";       /* t,p:    DS:021B */
s++;          /* s,p:    DS:0219 */
t++;          /* t,p:    DS:021C */
s="MMMMM";    /* s,p:    DS:021E */
t="NNN";      /* t,p:    DS:0224 */
s=f;          /* s,p:    DS:020C */
t=f;          /* t,p:    DS:020C */
printf(s,t);  /* 打印 ffff */
}

```

C>TYPE CONST3.C

```

#include "alloc.h"
main()          /* 如包含 alloc.h 则不会发生警告消息 */
{              /* 原型: void * --cdecl malloc(size--t size); */
const void *s=malloc(256); /* 如不包含标头文件 alloc.h, 编译时出现警告:
                               Warning: Non-portable pointer assignment */

s="使用 const 要细心!";
printf("%s\n",s);
}

```

3. char * const s; 是说明一个常量指针, s 和数组名差不多,但也有细微的区别。这是因为若使用一个指向 char 的指针,如果没有初始化,或没用分配内存函数给它分配内存,则不会给它分配字符串空间;但对数组则要分配空间,且数组变量保存着该空间的首地址。char const * str; 和 char * const s; 是不同的, str 是一个指向常量字符串的指针。两者有本质的区别(参见程序 CONST4.C 和图 3-1)。

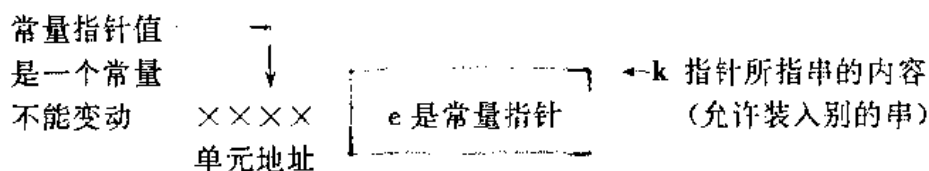


图 3-1 常量指针

C>TYPE CONST4.C

```

#include "stdio.h"
void strttest( const char *s,char const *t);
main()
{
char *s="%s=12S\n";
char *t="串";
strttest(s,t);
}
void strttest( const char *s,char const *t)

```

```

{
char h[80]="试验常量指针: "; /* 在调试时它不是可执行语句 */
char *ptr="正常串指针"; /* 在调试时它是可执行语句 */
char *const e="e是常量指针"; /* 在调试时它是可执行语句 */
/* h=s; 此句不允许,编译时会产生 Error:Lvalue required */
/* h="hh"; 此句也不允许.事实上,数组名h可以被看成一个指向
80个字节块的常量指针,程序员不能更改其值 */
/* e="能重新赋值吗?"; 不允许,产生 Error:Cannot modify a const object */
/* e++,也不允许 */
printf("%s",h);
s=h; /* 这种赋值是允许的 */
s=e; /* 这种赋值是允许的 */
s="不对啊!"; /* 这种赋值是允许的 */
printf(s,t);
strcpy(e,ptr); /* 对常量指针拷贝允许,因拷贝不会更改指针所指串首地址 */
} /* 程序输出:试验常量指针:不对啊!
如把语句 s=h;s=e; 和 s="不对啊!"; 取消,则输出:
试验常量指针:串=12S */

```

—8 continue

有时并不想完全地从循环中退出来,而只是想执行到循环语句体内某一语句时不再执行其后语句,而再次从循环的顶部开始执行,这时可用 continue 语句,如

```

for(i=0; i<20; i++)
{ if(array[i]==0) /* 如果数组元素值为0,则本次循环结束, */
continue; /* 即不再执行其后语句,而开始下一轮循环 */
array[i]=1/array[i]; /* 重新循环时变量i的值已被修正 */
}

```

continue 语句常出现在像 while, do-while 或 for 这类语句的循环语句体最内部,根据判别条件终止本次循环后开始新一轮循环。

—9 default

它是源程序中 switch 语句循环语句体内的一个语句,相当于“其余情况”时的执行。

如

```

switch(x)
{ case 1: /* 当 x=1 时由于本句后续无 break 语句,接着便执行下一句 */
case 2: y=10; break; /* 当 x 值不是 1~3 时,程序跳过这三句, */
case 3: y=100; break; /* 而直接执行 default 语句 */
default: printf("不正确的 x 值! \n"); exit(1);
}

```

当它不存在时,则程序什么也不执行便跳出 switch 语句。注意:不要将此关键字写错,否则它后面的内容将不被执行,Turbc C 对此也不进行检查。

—10 do-while

它和 while 语句一起构成循环语句,一般格式为

do-while

```

    { ..... } /* 语句体 */
while(表达式);
例如计算  $n=1 \times 2 \times 3 \times \dots \times 100$ 
    j=1; n=1; f=100;
    do { n*=j;
        j++;
    } while(j<=f); /* 当 j 小于 101 时继续执行循环 */
                /* 或者可以认为当表达式 j<=f 值非 0 时继续循环 */

```

do... while 语句和 while 语句的区别是, do... while 语句体中的语句至少要被执行一次,而 while 语句是先测试条件表达式的值非 0 时才执行语句体,因此有可能其语句体中一句也不会被执行。

—11 double

定义数据类型为实数的标识符。例如

```

double f; /* f 将占 4 个字长, 值范围=1.7E-308~1.7E+308 */
long double f; /* f 将占 5 个字长, 值范围=3.4E-4932~1.1E+4932 */

```

使用 double 或 float 数需要浮点数数学库 (MATH?. LIB) 支持。在集成环境中,如果程序使用了浮点数,TC 将自动连接浮点数学库。

—12 else

它意指“否则”,用于条件语句中,参见 if 关键字。

—13 enum

枚举类型修饰符。它定义一组离散整数的集合。一般定义格式是

```
enum [<id1>] {<id2>|=<const>},...};
```

例如可发定义

```
enum modes {LASTMODE=-1,BW40=0,C40,BW80,C80,MONO=7};
```

结果在源程序中可接着定义

```

enum modes colorss; /* colorss 只能取花括号内值 */
enum modes s[10]; /* 数组元素只能取花括号内值 */

```

其中 <id1> 是集合的标志名,或称枚举名,用于区分不同的枚举。枚举名必须唯一。

注意:它不能当常量或变量使用。<id2> 是整型常量名,可以加等号对其赋值 <const>; 如果无 <const>,则缺省常量值等于前一常量值加 1。第一个常量值的缺省值是 0。整型常量名相当于符号常量。例如你可以用宏来定义星期:

```

#define sun 0 /* 星期日 */
#define mon 1 /* 星期一 */
#define tues 2 /* 星期二 */
#define wed 3 /* 星期三 */
#define thur 4 /* 星期四 */
#define fri 5 /* 星期五 */
#define sat 6 /* 星期六 */

```

但用枚举则简单得多:

```
typedef enum week {sun,mon,tues,wed,thur,fri,sat}days;
```

这里 days 是变量名,表明它取值范围只能是 sun、mon、...、sat 等值中的一个。

在花括号内的整型常量名应是唯一的。所有枚举变量都应先定义后应用。

注意:在同一作用范围内的枚举型号及整型常量名应是唯一的。

enum 类型变量的数值可以不加修改地转换成 int 类型。除使用强制转换外,TC 将不对其进行类型检查。

—14 extern

一个全程变量在同一个程序中只能定义一次。extern 就是 external(外部的),

它告诉编译程序一个全程变量或一个函数体实际存储(量)和初始值已在某一个文件(程序段,通常指一个不相连的源码模块)中定义过,而不必再为它分配内存。当两个模块连接时,外部变量得到统一。其一般格式为

```
extern <数据类型定义>;  
extern <函数原型>;
```

例如

```
extern int fmode;  
extern void Factorial(int n);
```

外部变量在函数之外定义,它可为许多函数所共用(即相当于公共变量)。一个外部量只要在一个文件中定义,其它文件中只要对它冠以 extern 说明后便可使用。很少将它放在函数内部。

关键字 extern 对函数原型是任选的,因为函数本身一般总是外部的,C语言中不允许函数嵌套定义。在 TC 中,一个函数内的 extern 说明将遵守分程序作用域规则,即在定义它们的分程序作用域之外,说明无效。

```
C>EXTERN.PRJ /* 在集成环境中用此规划文件编译连接生成执行程序 */  
EXTERN1.C /* 程序输出:subx=3 mainx=8 */  
EXTERN2.C
```

```
C>TYPE EXTERN1.C
```

```
int x=2;  
main()  
{  
x++;  
sub();  
x+=5;  
printf("\tmainx=%d\n",x);  
}
```

```
C>TYPE EXTERN2.C
```

```
extern int x; /* 如果此句没有,则 TC 认为变量 x 没有定义 */  
void sub(void)  
{  
printf("subx=%d",x);  
}
```

—15 far

far 修饰符使用格式为两种:

```
<数据类型>□far□<指针定义>;    /* 修饰指针 */
<数据类型>□far□<函数定义>;    /* 修饰函数 */
```

例如

```
char far *s;
void far *p;
int far my—func()
```

这种修饰符常用于微型 (tiny)、小型 (small) 或紧凑型 (compact) 存储模式中形成远指针。当它用于函数说明时,被调用函数是远调用和远返回。

far 指针占 32 位,包括段地址和偏移量。段地址左移后和偏移量相加。使用 far 指针可以访问多个代码段,允许程序大于 64K。使用 far 数据指针也可存取多于 64K 的数据。对大型 (Large) 模式,指针缺省为 far。使用 far 指针可以在 Intel8088/8086 处理器中的 1MB 的存储空间中引用数据。

C>TYPE FAR1.C

```
main()
{
int far ss();
printf("%d\n",ss(2));
}
int far ss(int x)
{ return (x * x); }
```

—16 float

实数数据类型。占两个字长。范围在 $3.4E-38 \sim 3.4E+38$ 。如果程序中用了浮点数 (float 或 double)TC 将自动启动浮点数学库。

—17 for

语句 for 的一般格式是

for([<表达式 1>];[<表达式 2>];[<表达式 3>])<语句体>

1. 表达式 1 常用于初始化控制循环的变量 (或称【索引变量】)。在表达式 2 中常包含索引变量,对表达式 2 的测试结果决定是否继续进行循环。表达式 3 通常要对索引变量的修正。事实上表达式 1 可以有多于一个以上的表达式,各表达式之间用逗号分隔,这就是【多重表达式】。

如

```
for(i=0, t=strin; i<40 && *t; i++, t++)putch(*t);
putch('\n');
```

2. 表达式 2 的值为 0 (为假) 时,不再反复执行语句体;否则不断反复执行之;表达式 2 也可以是多重表达式,例如

C>TYPE FOR1.C

```
#include "stdio.h"
```

```

main(){
int i,j;
for(i=0,j=1;i<2,j<3;i++,j+=5)
printf("i=%d j=%d\n",i,j);
}

```

编译时虽出现警告: Code has no effect in function main,但输出结果为 i=0 j=1 是正确的,不过为了避免语句过于复杂化,很少用。

注意:表达式 2 的值的检验是在循环开始前就进行的。这就是说,如果一开始表达式 2 的值为 0,则循环体将可能不被执行。

3. 表达式 3 主要用于修正索引变量的值。它也可以是一个多重表达式。

4. 所有表达式均是任选的,任何一个表达式均可省略,但分号不能省略。特别当语句体是一空语句时 for(;;); 将产生死循环。这是由于表达式 2 不出现时,TC 总认为它有值 1 (恒为真);

5. 对 for 循环应当注意的几个问题是:

(1)for 循环的次数是确定的,但要注意循环体实际执行次数。

```
for(n=1;n<10;n++)
```

```
{.....} /* 循环体只执行 9 次,而不是 10 次 */
```

(2)在循环体中不能对索引变量赋值,否则结果是不可预料的。

C>TYPE FOR2.C

```

main()          /* 此程序将出现死循环 */
{
int k;
for(k=0;k<100;k++)
{
if(k==4)printf("rrrr");
else {k=4;printf("kkkk");}
}
}

```

(3)for() 后带有分号时你想执行的循环体最多只执行一次。

如

```
for(x=0;x<5;x++);
```

```
{.....} /* 想执行的循环体 */
```

这种错误是很难发现的。

(4)for 循环的条件检验在循环开始时就进行的。如果一开始条件为假 (false),则循环体不被执行。

C>TYPE FOR3.C

```

main()
{
int x=1,y;
for(y=1,y<x,y++)printf("x=%d y=%d,",x,y);
} /* 程序无任何输出 */

```

—18 goto

goto 语句其一般格式是

goto 标号;

goto 使程序无条件地去执行标号处的语句。

如

```
goto Again;
```

```
.....
```

```
Again: printf("+++\\n");
```

标号语句像 case 语句一样,是在一般标识符后紧跟一个冒号。

注意:标号后至少要有有一个语句,那怕是【空语句】即只有一个冒号的语句也行。

一个函数内的 goto 语句转向的标号不应在函数外。标号在定义它的函数内必须是唯一的。

goto 语句可以从多重循环语句体的最内层一下子跳到最外层;但反之则不行。

goto 语句可用 return、break 或 continue 语句代替,从结构化程序设计的角度看,一般应少用 goto 语句。

注意:goto 语句后的标号必须在程序中出现;反过来,设置了标号但可以没有 goto 语句转向它,此时标号相当于一个注释的作用。

另一个要注意的是下述形式:

```
{
```

```
.....
```

```
Label:
```

```
.....
```

```
goto Label;
```

```
; /* 此空语句在此不可少 */
```

```
}
```

—19— huge

huge 修饰符使用格式为两种:

<数据类型>□huge□<指针定义>; /* 修饰指针 */

<数据类型>□huge□<函数定义>; /* 修饰函数 */

huge 指针也是 32 位的,同样包括段地址和偏移量。与 far 指针不同的是, huge 指针总是规格化的,即每隔 16 个字节(或 1 个节)就有一地址作为段开始地址,这表明指针的偏移量在 0~15(十六进制的 0H~FH)。因而它可以进行指针比较。其次, huge 指针增值时也不会发生段重迭现象(参见《指针》一章)。

—20 if

条件语句,有两种基本形式:

1. if (<表达式 1>) /* 如果表达式 1 非 0, */
 <语句体> /* 则执行<语句体> */
2. if (<表达式 1>) /* 如果表达式 1 非 0, */
 <语句体 1> /* 则执行<语句体 1>, */


```

else
<语句体 2>      /* 否则执行<语句体 2> */

```

else 语句之后可以跟上述 if 语句（即嵌套），但 else 与 if 之间不应有其它语句。当存在多个 else 语句时，else 与同一层中前面最接近它的 if 语句配对。可用单步调试观察实际执行情况。如

```

if (x==1)y=12;
else if (x>4)y=1000;
    else if (x==3)y=-1;
    .....

```

表达式一是任一可以分解（或转换）成一个整数的表达式。【语句体】又称为【复合语句】，其一般形式为

```

{ <语句 1>
.....
<语句 n>
}

```

如

```

C>TYPE F1.C
main()
{
int k;
for(k=0;k<2;k++)
{
/* 复合语句开始 */
int x=4;          /* 变量 x 在复合语句中有效 */
printf("%d ",x+k);
}
/* 复合语句结束 */
/* printf("%d",x); 此句不能用,因这时变量 x 已无定义 */
}

```

当只有一个语句时，也是允许的，这时花括号对可略去。如

```

if(b==0.0)y=1;
或
if( b | ==0.0);
else y=1;

```

也是一样的，但一般不要这样写。#if 和 #else 语句有点类似 if 和 else 语句，但它们是不同的。后者是用于控制编译源文件行，而它们本身是不会产生执行代码的。

—21 int

整形数据类型。整形变量占一个字长（16 位）。它们可以带 signed（缺省值）或 unsigned 修饰符。带 signed 时，数值范围是 -32768 ~ 32767；带 unsigned 时，为 0 ~ 65535。

—22 interrupt

中断函数修饰符的一般使用形式为

interrupt<函数定义>;

如

```
void interrupt myhandler(){};
```

将一个函数定义成中断子程序,它要和 8088/8086 中断向量联合使用。TC 在编译被说明为 interrupt 的函数时,将产生附加的函数入口和出口代码以保护寄存器 AX、BX、CX、DX、SI、DI、ES 和 DS,其它寄存器 SP、BP、SS、CS 与 IP 作为 C 调用序列的一部分或作为中断处理的一部分来保护,换句话说,所有 CPU 寄存器内的值被存储。该函数为一个 IRET 指令所终结。当使用 interrupt 修饰符时,O/L/Stack warning 和 O/C/O/Use register variables 开关都要关闭(置成 Off)。

—23 long

长整形数据类型,即 long int。一个此类型变量占 32 位。

如

```
long x;          /* 相当于 signed long int x */
unsigned long x;
```

—24 near

定义指针为近指针。其一般形式是

```
<数据类型>□near□<指针定义>;      /* 修饰指针 */
<数据类型>□near□<函数定义>;      /* 修饰函数 */
```

如

```
char near *s;
int (near *ip)[20];
```

被定义的指针指向 64K 范围内的一个字。当编译成中 (medium)、大 (large) 和巨 (huge) 型存储模式时为形成 near 指针而常用它。

当 near 用于一个函数说明时,如

```
int near my—func(){};
```

TC 产生近调用和近返回函数码。

—25 pascal

pascal 修饰符说明一个变量或函数使用 Pascal 命名规约(将标识符转换成大写且前面没有下划线,这与源程序标识符是否使用了大小写无关)。

注意:当一个函数头部使用它时,如

```
int pascal FilecCount;
far pascal HisFunc(int x, char *s);
```

它通过规约来说明 Pascal 型参数,即函数表中第一个参数首先被压入堆栈,调用函数将整理堆栈(参见《函数》一章)。

—26 register

register<数据定义>;

为优化存取,register 类型修饰符告诉编译程序存储变量被说明在 CPU 寄存器内(如果可能的话)。

如

```
register int x;
```

—27 return

return 语句的一般形式是

```
return [<表达式>;
```

它使执行程序立即从当前执行函数返回调用该函数的函数,返回时可以返回表达式的值,该值可传送给调用函数。如

```
double sqr(double x)
{ return (x * x); }
```

return 也可以从函数某一点而不是在函数结尾处退出来。

例如

C>TYPE RETURN1.C

```
#include "stdio.h"
main(){
if(m(0)==-1){printf("-----\n");exit(1);}
printf("+++++++\n"); /* 此句前可以不写 else 关键字 */
}
```

```
int m(int x)
{if(x==0)return(-1);
return(1); }
```

结果输出:

若函数类型为 void,用 return 语句返回时函数将不带回任何值给调用函数。

return 中的表达式可以是相当复杂的表达式,它是一个常用语句。

程序 RETURN2.C 是一个有问题的程序,它说明用 void 说明的函数不应有返回;其次,对函数 s() 中的局部变量 k 用 return (&k); 是危险的,因为当函数返回时它已无效,所以很有可能它原有的数据被其它堆栈数据所覆盖,从而原 &k 中的数据已发生变化。

C>TYPE RETURN2.C

```
void sign();
m(int tok,int w)
{
int s[10]={1,2,3,4};
printf("%d\n",s[0]+tok+w);
}
int *s(int n)
{
int k,h=8; /* &k,DS:FFD4 */
k=n/2;
printf("h=%d",h);
sign(h);
```

```

m(20,2);
return (&k);
}
main()
{
int y=100; /* 单步调试时出现 &y; Must take address of memory location */
int * ptr;
/* y=sign(y); 不允许, Error: Not an allowed type */
ptr=s(y);
printf("%d\n", * ptr);
} /* 本程序在集成环境下调试时输出: h=8,23
                                     -30123

```

而单独执行时有, h=8,23

```

50                                     */
void sign(int x)
{
int u;
if(x>0){u=1;return (1);} /* Warning: Void functions may */
else if(x<0){u=-1;return (-1);} /* not return a value */
else {u=0;return 0;}
printf("%d", u); /* Warning: Unreachable code */
} /* Warning: Both return and return of a value used */

```

注意:若函数不是以 return 加上一个表达式结尾,即只有 return 而无表达式(例如,对 void 函数就可这样做),就会返回一个不确定的值(指多次返回时结果不一样)。

—28 short

短整形数据类型。占 16 位。TC 保证 short 型的变量不会大于 long 型的变量,即不比它占用更多的字节。

—29 signed

无符号数修饰符。

—30 sizeof

它一般应用形式为:

sizeof(表达式)或 sizeof(数据类型)

它返回一个表达式或一个数据类型(作为无符号整形)在内存中所占字节数。

如

```

memset(buff, sizeof(buff),0);
nitems=sizeof(table)/(sizeof(table[0]));

```

如果 ch 是一个变量,则 sizeof 后面可不加括号。如 sizeof ch; 是允许的,否则一定要用括号。

使用 sizeof 的目的是要增强程序的可移植性,免受不同计算机固有字长的影响。因此,它也被称为“编译状态运算符”。

—31 static

如果定义的数据变量或函数头部使用了 static,

如

```
static int i;  
static void printnewline(void){}
```

则被定义的数据变量或函数只在当前编码范围内有效。Static 将根据变量的类型分为静态全局变量和静态局部变量。(参见第四章—《变量的存储分类与应用》)

—32 struct

结构修饰符,它表明多个变量组合成一个记录。其一般形式是

```
struct [结构名]  
{  
[<数据类型> <变量名>];  
.....  
}[结构变量名];
```

例如

```
struct my—struct {  
char name[80], phone—number[80];  
int age,height;  
}my—friend;
```

结构名是一个任选的标志名,它表示结构的类型。结构变量名是数据的定义,也是任选的。注意:仅管两者都是任选的,但对一个结构来说,它们中间至少有一个被选用。记录内的元素靠数据类型后跟变量名定义,同类型变量之间用逗号分隔。不同类型数据之间用分号分隔。

接收结构的元素可使用记录选择符(即一个小数点),例如

```
strcpy(my—friend.name, "zhang fiang");
```

为说明其它同类型的结构,可用

```
struct <结构名> <变量名>;
```

如

```
struct my—struct my—friends[200];
```

数组 my—friends[] 被说明成类型为 my—struct 那样的结构数组。

—33 switch

switch 语句的一般形式是

```
switch(<表达式>)  
{  
case 常量表达式值 1: [<语句 1>; break;] /* 方括号内的语句是任选的 */  
case 常量表达式值 2: [<语句 2>; break;] /* case 后必有 break 终结,以求 */  
..... /* 退出 switch,否则它将接着执行后续语句 */  
default : <语句 n>  
}
```

例如

```

switch (operand){
case MULTIPLY:    x *= y; break;    /* 当 operand=MULTIPLY 时    */
case DIVIDE:      x /= y; break;    /* 当 operand=DIVIDE 时    */
case ADD:         x += y; break;    /* 当 operand=ADD 时      */
case SUBTRACT:    x -= y; break;    /* 当 operand=SUBTRACT 时  */
case INCREMENT2:  x++;              /* 当 operand=INCREMENT2 时 */
case INCREMENT1:  x++; break;       /* 当 operand=INCREMENT1 时 */
case EXPONENT:    /* 当 operand=EXPONENT 时    */
case ROOT:        /* 当 operand=ROOT 时      */
case MOD: printf("Not done\n");     /* 当 operand=MOD 时      */
break;
default: printf("Bug! \n");         /* 当 operand= 其它值时    */
exit(1);
}

```

例中像 MULTIPLY 等整常量必须先用 #define 语句定义,并且各常量的 ASCII 值不应相同。

C>TYPE SWITCH1.C

```

#include "stdio.h"
#define T1 1
#define T2 T1
main(){
    int i,x;
    i=1;x=0;
    switch (i){
        case T1: x++;
        case T2: x++; break;
        default: printf("Bug! \n");
                exit(1);
    }
    printf("x=%d\n",x); /* 如 T1 被定义为 1, T2 被定义为 T1+1 */
    getch();           /* 则当 i=1 时输出 x=2; i=2 时输出 x=1 */
}

```

由于 T1 等于 T2, 故在编译时将出现

Duplicate case in function main

错误。这就是说,它们在 switch 中必须是唯一的值。从改正后的输出结果可知,程序未遇 break 语句将往下继续执行。为避免意外,switch 语句中最好有 default 语句,以免发生意外时能及时发现错误。显然,default 语句应在所有 case 语句之后。

现在将上例稍作修改为

C>TYPE SWITCH2.C

```

#include "stdio.h"
#define T1 1
#define T2 21

```

```

main(){
    float i,x;
    i=1.0;x=0.0;
    switch (i){
        case T1: x++;printf("xx=%f\n",x);
        case T2: x++;printf("xxx=%f\n",x); break;
        default: printf("Bug! \n");
                exit(1);
    }
    printf("x=%f\n",x);
    getch();
}

```

输出结果为

```

xx=1.000000
xxx=2.000000
x=2.000000

```

这就说明 switch 里的表达式可以是非整形量,但如将 T1 改定义为 1.2,则编译将出错。这就是说,常量表达式值 1、常量表达式值 2、... (称【switch 语句的选择表达式】) 的值必须与整数兼容,即它必须能转换为一个整数。所以它可以是 char、enum 或 int 及其变型,但不可以是实数(浮点数,双精度数)、指针、字符串或其它数据结构(但可以用某一数据结构内与整数相容的成员,如 union 的成员,struct 的成员)。选择表达式也可以是字符(实际使用其 ASCII 码值),

如

C>TYPE SWITCH3.C

```

#include <ctype.h>
main(){
    do—main—menu();
}
do—main—menu(short *done)
{
    char cmd;
    *done=0;
    do{
        cmd=getchar();
        switch(cmd){
            case 'f': /* 'f' 或 'F' 是字符 */
            case 'F': do—file—menu(done);break;
            .... /* done 可能在 do—file—menu() 中被赋 0 值 */
            default ;exit(1);
        }while(! *done); /* done 被赋 0 值时结束循环 */
    }
}

```

—34 typedef

除了 TC 用一些规定的关键字作为数据类型的标识外,程序员也可以用自己认为满意的标识符定义数据类型,为此可用 typedef(即 type definition)。

该修饰符应用的一般形式为

typedef <原数据类型> <新数据类型标识符>

则新数据类型标识符就表示原数据类型。

例

```
typedef int *intptr;      /* intptr 与 int * 同义,下类同 */
typedef char  namestr[30];
typedef enum   {male, female, unknown }sex; /* 男性, 女性, 未知 */
typedef struct {
    namestr last,first; /* char last[30], char first[30] */
    char   ssn[9]; /* specification serial number 序号 */
    sex    gender; /* sex,gender 性别,gender=0 为男,gender=1 为女 */
    short  age;     /* age 年龄 */
    float  gpc;     /* 加仑/人 */
}student;          /* student 学生 */
typedef student class[100]; /* class 数组的类型为 student */
class hist104,ps102;      /* hist104[100], pr102[100] */
student valedictorian;
intptr intr;             /* int *intr */
```

用了 typedef 程序可读性和维护性都加强了。例如,只修改一处,即某个类型实际定义的地方,就能使整个程序与之相关的地方全部得到修改。

—35 union

联合的一般形式是

```
union [<联合名>]{
    <数据类型><变量名>;
    .....
}[<联合变量名>];
```

例如

```
union int—or—long {
    int i;
    long k;
} a—number;
```

联合允许一些变量共享存储空间。TC 将对联合 a—number 中最大元素分配足够的存储空间。但是它不像 struct 那样,它的元素并不是占用内存不同的空间,元素 a—number.i 和 a—number.k 将占用内存中同一空间。因此当其中一个元素占用这一空间时将冲掉该空间内原有的另一个元素。换句话说,上例中的两个元素在同一时刻只能有一个元素存在于这个空间内。

对联合元素的访问方法跟结构是一样的。

—36 unsigned

无符号数据类型修饰符。它将某些类型(char、short、int、long)说明为无符号,即这些类

型只存有非负值（大于或等于 0）。

—37 void

空数据类型。当将它作为一个函数的返回类型时，void 表示函数不返回任何实际值。

如

```
void hello(char * name){printf("Hello, %s.",name);}
```

如果 void 出现在函数表内，则表明该函数没有任何参数。如

```
int init(void) /* 函数无任何参数，或称函数有【空参数】 */
```

```
{return 1;} /* 函数返回数值 1 */
```

指针也可以用 void 说明。void 指针不能非显式地重定位，这是因为编译程序不能决定指针所指对象的大小。参见下例对指针 p 的赋值。

C>TYPE VOID1.C

```
#include "stdio.h"
int x;
float r;
void *p=&x; /* 指向变量 x */
main(){
    printf("p1=%p\n",p); /* 输出 p1=0444 */
    *(int *)p=2; /* 注意此格式 */
    printf("p2=%p\n",p); /* 输出 p2=0444 */
    p=&r; /* 指向变量 r */
    printf("p3=%p\n",p); /* 输出 p3=0440 */
    *(float *)p=1.1; /* 注意此格式 */
    printf("p4=%p\n",p); /* 输出 p4=0440 */
}
```

指向 void 类型的指针，并不是指向空无一物的指针，而是指向一个可以是任何数据类型的指针，只不过数据类型还不清楚。由于前一点，用户可以将任何指针不加转换地赋给 void 指针（反之亦然）；而对后一点，使用间接运算符 * 时必须象上例所示，而不能直接用象 *p=2; 那样的语句。

注意：许多运行时内存分配函数，如 malloc()，它们的类型均被说明为 void * 型，即它们返回一个无类型的指针，可以不经类型强制转换（在 TC 中），就将它赋给任何类型的指针（但为了保证可移植性，建议使用类型强制转换）。

作为一特别构造，可以在一表达式中加上 void，显式地表示忽略函数返回值。例如，要想以键入一键方式暂停一下，以便看清某一输出结果，但是你又想忽略键值，以免引起意外，则可用下列语句

```
(void) getch();
```

void 前后的括号是必须的，否则会出现

Expression syntax in function s

那样的语法错误。

C>TYPE VOID2.C

```
#include "stdio.h"
```

```

#define P1 printf("\n 请输入一个数")
main()
{
    void ch='A';
    char cg='B';
    /* 开始有:—streams[0]:{0,521,'\x0','\x0',0,NULL,NULL,0,540) */
    fprintf(stdout,"%c\n",cg);
    /* 没有此句则没有—streams[0]的信息(No type information) */
    /* fprintf(stdout,"%c\n",ch); 此句不允许(Not an allowed type) */
    P1; getch(); /* 执行此句后—streams[0]未变,表明它与预定义流 stdin 无关 */
    P1; (void)getch(); /* —streams[0]也未变 */
    /* ch=(void)getch(); 此句不允许,主要是(void)getch();不会返回值 */
}

```

注意:如果一个函数没有被说明类型,则缺省地认为其是 int 型;如果函数返回一个没有类型的指针,则缺省地认为它返回一个指向 char 类型的指针。

—38 volatile

它表示被修饰的变量可能被后台子程序改变。每一次访问变量都是从内存取出,而不是寄存器的拷贝值。

例如

```

volatile int ticks;
interrupt timer()
{
    ticks++;
}

wait(int interval) /* 按常规 ticks 进入循环前被赋 0 值,进入循环后 */
{
    /* 其值从程序看来未被改变过,那么当 interval 值 */
    ticks=0; /* 大于 0 时将出现死循环,其实不然,因 ticks */
    while(ticks<interval); /* 中间可为硬件时钟中断子程序所变更。 */
}

```

这里 timer 是一硬件时钟中断子程序,以上子程序将完成给定 interval 时间间隔的 ticks 等待。注意:高度优化的编译器绝不会在 while 循环内部加载 ticks 数值,因为循环中不改变它的值。

从上面的例子中看到,volatile 几乎与 const 修饰符相反,它表示目标不但可为用户程序所用,而且也可以为中断子程序、I/O 子程序等外部程序使用。一个目标(象上例中的 ticks)加上 volatile,就是告诉编译器,在编译时不要运算包含此目标的表达式,因为它的值任何时候都可能改变!这种修饰还阻止编译生成寄存器型的变量。

—39 while

while 语句的一般形式是

while(<表达式>)<语句体>

如

```
while(*p=='□')p++;
```

只要 <表达式> 的值非 0,就执行语句体中的语句。注意,一开始执行该语句时是先测

试 < 表达式 > 的值,然后才决定是否执行语句体中语句。语句体中一般有涉及更改 < 表达式 > 值的语句,当语句体执行完后马上又去测试 < 表达式 > 值,若非 0,则又执行语句体;如此反复,直至 < 表达式 > 值为 0(或称“不为真”)。

当表达式为一个变量时,它必须有确定的值,否则由于循环次数的不确定引起不可预料的结果。

如

```
dotime()  
{  
    int n;  
    while(n<100)  
        { n++;printf("%d",n); }  
}
```

由于 n 是局部变量,因而其值在开始时是不确定的,这可以通过调试表达式观察出来。

while 循环体也允许为空语句,如执行语句

```
while( (ch=getch()) != 'Q' );
```

时,只有当你按了大写的 Q 键后才结束循环,转去执行它后面的语句。

—40—AH

伪变量,类型: unsigned char, 对应寄存器为 AH, 即 AX 的高位字节
在进行低级编程时,有时需要用 C 语言直接访问 CPU 寄存器,以达到:

1. 在调用一系统子程序之前,将某些值装入寄存器;
2. 查看寄存器的值。

为此可以使用【伪变量】,即以下划线开头,紧跟寄存器名的一些标识符,如 —AH。

1. 伪变量可看成适当类型 (unsigned int 和 unsigned char) 的全局变量;
2. 在伪变量之前不能使用地址操作符 (&), 因为伪变量没有地址;
3. 由于编译器在不断地产生代码,而这些代码要经常使用寄存器,所以无法保证伪变量的值能够保存那怕是很短暂的时间。为此,向寄存器赋值仅在它使用之前,而在寄存器取得所需值后应立即读出数据。

如

```
void readchar(unsigned char page,unsigned char *ch,unsigned char *attr);  
{ —AH=8; —BH=page; /* 调用中断 INT 10H 子功能 8,—BH 为可见页号 */  
  geninterrupt(0x10); /* 产生中断 0x10 */  
  *ch=—AL; /* 得字符 */  
  *attr=—AH; /* 得属性 */  
}
```

而在 geninterrupt() 与下一句之间不能插入任何操作,以防 —AH, —AL 中值可能变化;

4. 经过一函数调用后,寄存器的值可能会改变。如

.....

```
—CX=18;  
myFunc(); /* 调用函数 */  
i=—CX; /* —CX 的值未必再是 18! */
```

在调用前后,数值不变的寄存器仅有 —CS、—BP、—SI 和 —DI。

5. 修改某些寄存器得非常小心,因为这可能带来意想不到的效果!例如,直接将值存入 —CS、—SS、—SP 或 —BP 中几乎注定要出错,因为 TC 编译器所产生的机器码常以各种方式使用它们。

—41 —AL

伪变量,类型:unsigned char,对应寄存器 AL,AX 的低位字节

—42 —AX

伪变量,类型:unsigned int,对应寄存器 AX,通用寄存器/累加器

—43 —BH

伪变量,类型:unsigned char,对应寄存器 BH,BX 的高位字节

—44 —BL

伪变量,类型:unsigned char,对应寄存器 BL,BX 的低位字节

—45 —BX

伪变量,类型:unsigned int,对应寄存器 BX,通用寄存器/变址寄存器

—46 —BP

伪变量,类型:unsigned int,对应寄存器 BP,基地址指针

—47 —CH

伪变量,类型:unsigned char,对应寄存器 CH,CX 的高位字节

—48 —CL

伪变量,类型:unsigned char,对应寄存器 CL,CX 的低位字节

—49 —CX

伪变量,类型:unsigned int,对应寄存器 CX,通用寄存器/循环控制变量

—50 —DH

伪变量,类型:unsigned char,对应寄存器 DH,DX 的高位字节

—51 —DL

伪变量,类型:unsigned char,对应寄存器 DL,DX 的低位字节

—52 —DX

伪变量,类型:unsigned int,对应寄存器 DX,通用寄存器/保留数据

—53 —DI

伪变量,类型:unsigned int,对应寄存器 DI,寄存器变量

—54 —SI

伪变量,类型:unsigned int,对应寄存器 SI,寄存器变量

—55 —SP

伪变量,类型:unsigned int,对应寄存器 SP,堆栈指针(SS 的栈顶指针)

—56 —cs

—57 —ds

—58 —es

—59 —ss

—cs、—ds、—es 和 —ss 是 4 个特定的 near 数据指针修饰符,它们修饰的指针均为 16 位,分别对应于各段寄存器 CS(代码段地址)、DS(数据段地址)、ES(附加段地址)和 SS(堆栈段地址)。例如,

```
char —ss *p;
```

p 为相对于栈段的 16 位偏移量。

3.3 附注

顾名思义,关键字就是很重要的意思。从这一点考虑,其实还有一些也很关键的字或词,像一些宏名 (SEEK—CUR)、结构名或结构 (如 FILE)、五个预定义流名 (stdin, stdout 等)、标准库函数名及编译指令等等,它们都有固定的含义,用户不要去改变它们,也不要把它标识符定得和它们相同。

第四章 变量的存储分类与应用

4.1 分类

表 4-1

作用域	局部 (变量)				全局 (变量)		
存储类	动态类			静态类	静态类		
分类 修饰符	自动的 auto (缺省)	寄存器 register TC 很少用	(形参)	静态的 static	外部的 extern (缺省)	静态的 static	
非显式初始化值	不确定		(实参)	程序开始执行时有 0 值			
值保留(可见性) 及非其所属函数 能引用其值情况	离开所在函数后值不保留			离开函数后其值保留			
	别的函数也不能引用其值			不能	能		
在所属函数内	有效						
在所属函数外	无效						
在所属文件内					有效		
在所属文件外	无效				有效	无效	

1. 可见性与作用域

变量在一个程序的某一范围内,如果能被该范围内的语句引用,或者直观地说,在集成环境里对此范围内用 F7 键单步跟踪该变量时,监视窗口内不会出现像

. 变量名 : Undefined symbol ' 变量名 ' in function 模块名

这样的提示信息,则称该变量在此范围内具有【可见性】(注意,这种说法有时可能被认为是不严格的)。

如果变量在某范围内具有可见性,便说该范围是【变量的作用域】。作用域是从空间角度对变量的描述。作用域可以是一个文件,也可以是函数。

从作用域角度分,可分成局部变量和全局变量两种。局部变量的作用域只限定在它出现的模块中,即使该模块被嵌套在另一模块内也如此。这里讲的模块,是指 {} 内的程序块。

2. 变量的值可在某一时刻内保留不动的属性称【变量的存在性】。存在性是从时间角度来描述变量的。

从存在性看,变量可分为动态存储和静态存储两种。静态存储是在整个程序运行时都存在,而动态存储只是在调用函数时才临时分配单元,函数调用结束便不再存在。

3. 表 4-1 中寄存器泛指寄存器变量。注意,寄存器变量和寄存器是不同的概念。寄存器变量和具体的计算机系统相关。

4. 函数的形式参数一般被定义为局部动态变量或寄存器变量。

5. 变量一般在函数的开头定义或说明,很少放到其它地方。注意,变量的定义和变量的说明也是不同的概念。

4.2 初始化

1. 非显式初始化

格式: <类型> □ <变量名>;

这一格式虽是变量的定义格式,但也包含它非显式地初始化了。确切地说,对静态存储类变量程序在编译此语句时给其赋 0 值,故程序开始执行时就有 0 值,以后它不再理会此语句,即只初始化一次;对动态类变量得到一个不确定的值。

如

```
int x, y;
char s;
static char s;    /* s 有 0 值 */
char s[];
char t[][4];
```

2. 显式初始化

格式: <类型> □ <变量名> = <值>;

对静态存储类变量程序在编译此语句时给其赋相应值,故程序开始执行时变量就有这个值,以后它不再理会此语句,即只初始化一次;对动态类变量得到一个确定的值,变量被压入堆栈,其所在函数每被调用一次,就要重新初始化一次。

如

```
int x=1, y=2;          /* 整型数 x 被初始化后有值 1, y 有值 2 */
char s=78;             /* 此句和上句相同, 78 为 N 的 ASCII 码 */
char s[]="CH";         /* 数组 s 有值 CH, s[0]='C', s[1]='H', s[2]='\0' */
char t[][4]={"Fi", "Sec"}; /* t[0][0]='F', t[0][1]='i', t[0][2]='\0' */
                        /* t[0][3]='\x0', t[1][0]='S', t[1][1]='e' */
                        /* t[1][2]='c', t[1][3]='\x0' */
```

3. 静态变量和动态变量初始化的异同

注意: 对动态类变量, 语句

```
int x=8; /* 即 auto int x=8; 此处缺省的存储类是 auto, 它常被省略 */
```

相当于两个语句

```
int x; x=8;
```

即它们是等价的; 但对静态类语句,

如

```
static int x=8;
```

就不等价于语句

```
static int x; x=8;
```

对静态或外部变量初值也可以是常量表达式,对指针变量也可以是地址表达式;但对于局部变量或寄存器变量,初始化表达式还可以是前面已定义过的变量或函数等。

C>TYPE V1.C

```
s(int t)
{
    return (t*t);
}

main()
{
    /* 单步调试时 static 语句是非执行语句,即调试时被跳过 */
    static int k;          /* 允许 */
    static int u=9+8;       /* 允许 */
    int y=8;               /* 允许 */
    int x=y;               /* 允许 */
    /* static int z=y+x; 非法初始化,不允许 */
    int *p=&y;             /* 允许 */
    /* static int *ptr=&x; 非法初始化,不允许 */
    int w=y>>4;           /* 允许 */
    /* static int v=y>>4; 非法初始化,不允许 */
    int m=s(2);            /* 允许 */
    /* static int n=s(7); 非法初始化,不允许 */
}
```

4.3 详细说明

4.3.1 局部变量

被定义在函数内部或定义在函数内用 {} 括起来的复合语句 (它们被当作分程序处理) 内部的变量是局部变量。为简述起见,以后我们只笼统地讲“函数内部”时通常就包含 此类复合语句。这里要注意的是:

- (1) 主函数 main 也是函数,所以它内部的定义的变量也和其它函数一样,是局部变量;
 - (2) 函数的形式参数也是局部变量;
 - (3) 局部变量可以作实参传给其它函数;
 - (4) 因为函数定义都是独立的,所以在不同函数内的同名局部变量之间互相不受影响;
- 例如:变量 x, y 在两个函数 (main 和 pingfang) 中都有定义:

C>TYPE V2.C

```
pingfang(int z)
{int x=3,y=2;
    return (x*y*z);
}

main(){
    int x=1,y=5;
```



```

y+=pingfang(80);
printf("x=%d y=%d\n",x,y);    /* 输出结果为 x=1 y=485 */
}

```

用 F7 单步调试刚进入函数 pingfang 时, 输入的调试表达式的值是,

```

. z: 80
y: Undefined symbol 'y'    /* main() 中的 y 或 x 在 pingfang() 中无定义 */
x: Undefined symbol 'x'    /* 表明它们是 main() 中的局部变量 */

```

刚返回 main 函数时有

```

. z: Undefined symbol 'x'    /* 表明 z 是 pingfang 中的局部变量 */
y: 485                        /* 5+2*3*80=485, 其中 5 是 main 中的, 2 与 3 */
x: 1                          /* 是 pingfang 中的。1 是 main 中的, 现恢复 */

```

(5) 局部变量只在复合语句中有效。

C>TYPE V3.C

```

/* 1: */ main(){
/* 2: */ int x=100,y=1,k;
/* 3: */ for (k=1;k<3;k++)
/* 4: */ {                      /* 复合语句开始 */
/* 5: */     int y=4,z=80;
/* 6: */     y*=2,x*=2;
/* 7: */     printf("x=%d y=%d z=%d\n",x,y,z);
/* 8: */ }                      /* 复合语句结束 */
/* 9: */     printf("x=%d y=%d\n",x,y);
/* 10: */ }

```

程序输出

```

x=200 y=8 z=80                /* y,z 首次初始化后进行计算 */
x=400 y=8 z=80                /* y,z 重新初始化后进行计算 */
x=400 y=1                      /* x 在 main 内整个起作用 */

```

如要将第 9 句改成

```
/* 9: */ printf("x=%d y=%d z=%d\n",x,y,z);
```

即输出 z 值, 则会出现

Undefined symbol 'z' in function main

的错误, 因在复合语句外变量 z 未定义。

由于局部变量只对本函数有效, 出了函数就在内存中不再存在 (或称【释放】), 而且其它函数中即使有同名变量 (包括全局变量) 也不能影响它, 所以这种函数可以整个地直接用到其它的源程序中去而不必作任何修改, 而要调用函数时只要通过将实参传给其形参即可。换句话说, 这种函数具有通用性, 具有固定的功能。另一方面, 局部变量也可以一直保持其固有的含义, 再加上调试简单明确, 调试正确后应用时也无后顾之忧, 因此在模块化设计时它可作为独立的模块。

1. 局部动态变量

函数中的局部变量如未指明存储类均认为是动态的。通常把 auto 修饰符省略。函数中说

明的变量其缺省存储方式为自动的。该变量只要一离开它所定义的函数便不存在。

局部动态变量不占用固定存储空间,它只在其所属函数被调用时为其分配存储空间,函数调用结束后其所占用的空间即被释放。释放的空间可作它用。

在定义局部动态变量时如未赋初值,则它的初值是一个不确定的值。

2. 局部静态变量

局部变量前有 static 修饰符。该变量离开其所在的函数后,在内存中其值仍保留,但该值不能为其它函数使用,也即是说其作用域也局限于函数内。局部静态变量占用的存储空间在程序整个运行期间都不会被释放,即该空间不能作其它用途。因此使用局部静态变量将使执行程序占用较多内存空间。

在定义局部静态变量时如未赋初值,则它的初值是 0。如在定义时要赋初值,则当其所属函数被多次调用时,只在首次调用时才赋初值(实际是在编译时赋的初值),以后调用时局部静态变量不再赋初值,而是使用上次其所属函数调用结束时它本身具有的值。换句话说,函数调用结束时,用静态局部变量的终值取代了它的初值,并且该值被保留在静态区内。当下次又调用此函数时,保留的终值便被当初值使用。因此,使用局部静态变量将降低程序的可读性,因为调用次数一多就可能搞不清变量的当前值。

万不得已或确实需要时才使用局部静态变量。例如,当发生堆栈溢出(Stack Overflow)时可考虑将局部变量设置为局部静态变量。

```
C>TYPE V).C
```

```
st(int x)
{
    auto z=1; /* z 是局部动态变量,函数被调用时它被置于堆栈中,每次函数调用时本句都起作用,即函数每被调用一次 z 就要重新赋一次初值本句与 auto z;z=1; 两句等效 */
    static y=2; /* 首次调用时本句起作用,以后再次调用本函数时,本句不起作用
    所以,本句不等于这样两句: static y; y=2; */
    z+=1; /* 本函数执行一次,z=2,即初值 1 再加 1 */
    y+=1; /* 本函数调用一次,y=上次调用后值 +1 */
    return x+y+z;
}

main(){
    int i,j=10;
    for(i=0; i<3;i++)
        printf("i=%d sum=%d\n",i,st(j));
} /* 程序输出结果: i=0 sum=14
                    i=1 sum=15
                    i=2 sum=16 */
```

用 F7 热键跟踪调试可以清楚地看到局部动态变量 z 和局部静态变量 y 的值的变化的。

4.3.2 全局变量(外部变量,extern)

在函数之外定义的变量称为外部变量,又称全局变量。

1. 全局变量从它开始定义的位置开始,直到程序结束都起作用。即在这个作用域内的函数都可用此全局变量的值;

2. 从程序头部到一个全局变量之间的函数不能直接用此变量值,如要应用,必须在函数

内部用修饰符 extern 作一次说明。

```
C>TYPE V5.C
int max(int x,int y)
{
return (x>y? x:y);
}
main(){
extern int a,b;    /* 说明使用外部变量 */
printf("max=%d\n",max(a,b));
}/* 程序输出:18 */
int a=10,b=18;    /* 定义外部变量 a,b */
```

一般情况下 extern 修饰符是很少在函数内部使用的。从此例也可见变量说明与变量定义的概念。

3. 如果全局变量未被初始化,它(包括指针变量)自动初始化有 0 值。

4. 虽然可以定义全局用指针变量,但是对全局指针变量不能在函数外用动态分配内存函数给它分配空间。

```
C>TYPE V6.C
#include "alloc.h"
void *p=malloc(200); /* 不允许 */
main(){}
```

5. 当全局变量分属于几个源文件时,一个源文件中要引用另一源文件中的全局变量时,要在全局变量类型前加上 extern 修饰符。

```
C>TYPE V7.C
int a;    /* 暂称语句①, a 是全局变量 */
main(){
a=3;
printf("%d %d\n",a,pl(a));
}
```

```
C>TYPE V8.C
extern int a;    /* 暂称语句②, a 也是全局变量 */
pl(int x)
{int y=10;
return x*y+a; }
```

在集成环境里使用规划文件 V7. PRJ 编译:

```
C>TYPE V7. PRJ
C:V7.c
C:V8.c
```

结果输出: 3 33。

(1) 若将语句①②改成

```
int a;    /* 语句①不变 */
```

```
static int a; /* 新语句② */
```

则输出：3 30。这就是说，a 在程序 V8.C 中已变成全局静态变量，它有初值 0，并且它只能被本源文件的函数引用。

注意：这里的 static 的意思已不只是静态的意思，它还有使变量局部于它所属文件的意思。

如将语句①②改成

```
static int a; /* 新语句① */  
int a; /* 新语句②，a 在 V8.C 中为全局变量，故有 0 值 */
```

也有类似输出结果。

(2) 若将语句①②改成

```
int a; /* 语句①不变 */  
int a; /* 新语句② */
```

或

```
int a; /* 语句①不变 */  
extern int a=1; /* 新语句②。用 extern int a; 不会出问题 */
```

则编译时将出现

—a defined in module V7.C is duplicatited in module V8.C

的错误。

对于函数外的全局变量的定义只能进行一次。修饰符 extern 只是说明外部变量，而不能兼之对变量进行初始化。在任何需要使用外部变量的函数内可以再用 extern 说明。所以，如把 V8.C 改成

```
extern int a;  
pl(int x)  
{int y=10;  
return x * y + a + mm(a);  
}  
mm(int x)  
{extern int a;  
return x + a;  
}
```

则程序输出：3 39。因此，对外部变量定义和说明是有区别的。

(3) 现在改语句

```
static int a; /* 新语句① */  
extern int a; /* 语句②不变 */
```

则会出现

Undefined symbol "a" in module V8.C

的错误。全局变量使用 static 修饰后，其作用域只在它定义的源程序中。

6. 可将外部变量说明放在头文件中,则在包含该头文件的源程序中外部变量均起作用。

```
C>TYPE V9.PRJ
```

```
V10.c(V9.H)
```

```
V11.c(V9.H)
```

```
C>TYPE V9.H
```

```
extern int a; /* 只一行 */
```

```
C>TYPE V10.C
```

```
C>TYPE V11.C
```

```
#include "V9.H"          #include "V9.H" /* 涉及程序都包含 */
int a; /* 定义全局变量 a */ pl(int x)
main(){                  {int y=10;
a=3;                    return x*y+a;
printf("%d %d\n",a,pl(a)); }}
```

或将 int a; 语句置于 V11.C 中也可。但只能在一个程序中定义。

```
C>TYPE V10.C
```

```
C>TYPE V11.C
```

```
#include "V9.H"          #include "V9.H"
int a;
pl(int x)
main(){                  {int y=10;
a=3;                    return x*y+a;
printf("%d %d\n",a,pl(a)); }
}
```

本例给出在头文件中说明全局变量的示例,这样既能得到正确答案,又避免了在每一个文件中都来说明外部变量的麻烦。

假定我们不用规划文件 V9.PRJ,而将语句 extern int a; 去掉,并在 V10.C 中增加一句 #include "V11.C",然后对 V10.C 编译执行,仍得正确结果: 3 33。这里使用了包含文件后,可以认为变量 a 实际只是在一个文件内,而不是在两个文件内。

在标头文件中说明了不少外部变量,如

```
extern int far          —Cdecl ATT—driver—far[]
extern int far          —Cdecl CGA—driver—far[]
extern int far          —Cdecl EGAVGA—driver—far[]
extern int far          —Cdecl Herc—driver—far[]
extern int far          —Cdecl IBM8514—driver—far[]
extern int far          —Cdecl PC3270—driver—far[]
extern int              —Cdecl —8087;
extern int              —Cdecl —argc;
extern char             * * —Cdecl —argv;
extern char             —Cdecl —ctype[];
```

extern int	—Cdecl —doserrno;
extern int	—Cdecl —fmode;
extern unsigned	—Cdecl —heaplen;
extern double	—Cdecl —huge—dble;
extern float	—Cdecl —huge—flt;
extern long double	—Cdecl —huge—ldble;
extern unsigned int	—Cdecl —openfd[];
extern unsigned char	—Cdecl —osmajor;
extern unsigned char	—Cdecl —osminor;
extern unsigned	—Cdecl —psp;
extern unsigned	—Cdecl —stklen;
extern long double	—Cdecl —tiny—ldble;
extern unsigned	—Cdecl —version;
extern int	—Cdecl daylight;
extern int	—Cdecl directvideo;
extern char	* * —Cdecl environ;
extern int	—Cdecl errno;
extern int far	—Cdecl gothic—font—far[];
extern int far	—Cdecl sansserif—font—far[];
extern int far	—Cdecl small—font—far[];
extern char	* —Cdecl sys—errlist[];
extern int	—Cdecl sys—neerr;
extern long	—Cdecl timezone;
extern int far	—Cdecl triplex—font—far[];

你要在自己的源程序中引用某一变量,除了包含标头文件,还应定义该变量。我们将在有关章节中对它们分别讨论。

7. 全局变量如在函数内发生了变化,则可以认为它是函数“间接返回值”。

8. 静态外部变量由于其作用域仅仅局限于它所在的模块,因此为多人并行设计提供了方便,此时不必考虑变量之间的冲突。

4.4 局部变量和全局变量的关系

局部变量和全局变量同名时,局部变量有效。因此局部变量最好不要和全局变量同名,否则有可能出现不可预料的结果。

C>TYPE V12.C

```

/* 1:  */ int y=1;          /* y 是全局变量 */
/* 2:  */ main(){
/* 3:  */ ;                /* 空语句      */
/* 4:  */ ;
/* 5:  */ printf("%d\n",y);
/* 6:  */ }

```

用 F7 单步追踪,则调试表达式 y 的值依次是,1,1,1,1;而将第三句改成 int y; 时,其值依次是,1,28028,28028,1,这个 28028 就是一个“不可预料的值”。编译程序将发出警告:

Possible use of 'y' before definition in function main

显然,这个警告是不可忽略的。如再把第 4 句改成 y=2;,则有 1,30,2,2,1,这里 30 也是一个不可预料的值,它是由主函数内非显式初始化语句 int y;引起的。

4.5 寄存器变量 (register)

1. 寄存器变量的使用与机器的硬件特性有关,不同的计算机系统对这类变量处理的方法可能不同。TC 把它当作自动变量处理,有时还是对它分配单元,并不真正使用寄存器。所以使用虽合法,但有可能达不到利用它提高执行速度的目的。

2. 静态类变量不能作为寄存器变量。如,不能用

```
register static int x;
```

这样的语句。

4.6 嵌入汇编 (inline) 和寄存器变量

在函数所说明的所有寄存器类型的变量中,选出使用频繁的两个作为寄存器变量,其它的则作为自动变量处理。如果 register 关键字所修饰的变量不能作为寄存器变量,则忽略该关键字的作用。

只有 short、int(和相应的 unsigned 类型)或两字节指针变量可以作为寄存器变量。SI 和 DI 是用作存放寄存器变量的寄存器。如果函数没有使用寄存器变量,inline 汇编可自由使用 SI 和 DI 作为暂存寄存器。C 函数的出口、入口代码自动保存和恢复调用者的 SI 和 DI 的数值。

如函数中有一寄存器变量说明,嵌入汇编代码可以通过使用 SI 和 DI 以改变或引用寄存器变量,但最好的办法是直接使用 C 语言符号,以防止寄存器变量在内部实现时有变化。

第五章 数据类型的转换

TC 支持数据类型之间的自动转换。

5.1 int 类型和 char 类型间的转换

将一个字符放到一个内存单元中去,并不是将字符本身放到内存单元中,而是将它的 ASCII 码值以二进制方式存放于内存单元中。因此一个内存单元(8 位)如存放了一个二进制数 01100001(相当于十进制数 $1 \times 2^6 + 1 \times 2^5 + 1 = 97$),它既可以认为是字母 a,也可以认为是数 97,因为它们都以这种方式存储。进一步说,该内存单元中的数据,既可按字母 a 输出,也可按整数 97 输出。以字符输出时,需要先将 ASCII 码转换为相应字符后才能输出。按整数输出时,直接将 ASCII 码作为整数值输出。

无论单字符常量还是双字符常量,均能表示成 16 位数值。TC 支持双字符。例如可定义

```
int x="An";
```

这时变量有值 0X6E41,即前一字符 A 在低八位字节,后一字符 n 在高八位字节。字符以其 ASCII 码值存储。

注意:下述定义

```
char x='An';
```

仅管允许,但只有前一个字符被转换,因为只分配给 x 一个字节(8 位)。还应注意,其它 C 编译器可能并不支持双字符常量。值得指出,定义

```
char x="An";
```

是不允许的。

signed char 或 signed 类型的变量转换成 int 类型时使用符号扩充;

unsigned char 类型常量将值赋给 int 类型变量时,总是将高位字节置成 0。

编译前可以用开关 -k 或 -k- (或集成环境中的 O/C/C/Default char type) 把源程序中缺省的 char 类型按 unsigned char 或 signed char 类型编译。

```
C>TYPE CHANGE1.C
```

```
main()
{
    /* 单字符常量 */
    unsigned char a1='A',a2='\n',a3='\x7f',a4='\x80',a5='\x81',a6='\x82';
    char b1='A',b2='\n',b3='\x7f',b4='\x80',b5='\x81',b6='\x82';
    unsigned char c1='An',c2='\n\t',c3='\005\007';          /* 双字符常量 */
    char d1='An',d2='\n\t',d3='\005\007';                  /* 双字符常量 */
    /* char f1='Anb'; Error: Character constant too long */
    /* char f2="A"; Warning: Non-portable pointer assignment */
}
```



```

int e1='An',e2='\n\t',e3='\005\007';           /* 双字符常量      */
/* int g1='Anb'; Error: Character constant too long */
/* int g2="A"; Warning: Non-portable pointer assignment */
int x1=a1,x2=a2,x3=a3,x4=a4,x5=a5,x6=a6;
int y1=b1,y2=b2,y3=b3,y4=b4,y5=b5,y6=b6;
int z1=c1,z2=c2,z3=c3,w1=d1,w2=d2,w3=d3;
printf("x=%d %d %d %d %d %d\n",x1,x2,x3,x4,x5,x6);/* 十进制打印 */
printf("y=%d %d %d %d %d %d\n",y1,y2,y3,y4,y5,y6);
printf("z=%d %d %d\n",z1,z2,z3);
printf("w=%d %d %d\n",w1,w2,w3);
printf("e=%d %d %d\n",e1,e2,e3);
printf("x=%x %x %x %x %x %x\n",x1,x2,x3,x4,x5,x6);/* 十六进制打印 */
printf("y=%x %x %x %x %x %x\n",y1,y2,y3,y4,y5,y6);
printf("z=%x %x %x\n",z1,z2,z3);
printf("w=%x %x %x\n",w1,w2,w3);
printf("e=%x %x %x\n",e1,e2,e3);
}
/* 输出结果
x=    65    10    127    128    129    130
y=    65    10    127   -128   -127   -126
z=    65    10     5
w=    65    10     5
e=28225 2314 1797
x=    41    a  7f        80    81    82  高8位为0,低8位为字符 ASCII 码
y=    41    a  7f ff80 ff81 ff82  符号扩充。字符值大于127(10进制),高位字节置成-1(0xff)
z=    41        a     5      定义为char,只印出前一字符
w=    41    a     5
e=6e41    90a   705      定义为int,前一字符表示高位,后一为低位
                           如6e相当字母n,41相当字母A
*/

```

5.2 int 类型 and enum 类型间的转换

两者可不经转换而直接使用。

5.3 指针之间的转换

在 TC 中,用户源程序中不同指针有不同长度。例如, far 与 huge 指针为 32,而 其余指针为 16,这依赖于存储模式和用户使用的【地址修饰符】(指 near、far、huge、—cs、—ds、—ss 或 —es 等)。

一个指针必须指向某一特定类型,其中包括 void 类型。

指针说明为 void 后,并不是说它不指向任何数据,只不过指向数据的类型还不清楚。此后用户可以将任何指针不加转换地赋给 void 指针,反之亦然。不过应当注意到由此可能产生某些意想不到的效果(参见程序 CHANGE2.C)。因此最好对同类型指针操作,否则如果前后发生矛盾的话,你应当重视在编译时检查出指针这种重新赋值是有疑问的警告。另外,数据类

型指针不能与函数类型指针相互转换。

C>TYPE CHANGE2.C

```
fx(x)
{ int a=2; return (x+a); }
main(){
int s=2,t;
int *p=&s;          /* p 指向 int 型数据          */
int (*w)();          /* w 指向函数，这里它不能为 void，
                      因它所指函数 fx() 为 int 型 */
w=fx; /* 不能有 w=p 或 p=w，将警告：Suspicious pointer conversion */
t=(*w)(30); /* Not an allowed type, 因为一个指向函数，一个指向数据 */
printf("%d\n", *p);
printf("%d\n", t);
}
```

C>TYPE CHANGE3.C

```
void vx(double x)
{ printf("%f\n", x); }
double fx(double y)
{ return y+1; }
main(){
void (*v)(double);
void (*f)(double);
double (*ff)(double);
double fn;
ff=fx;
fn=(*ff)(3.14);
printf("%f\n", fn);
/* f=fx; Warning: Suspicious pointer conversion */
/* fn=(*f)(3.14); Error: Not an allowed type */
f=vx;
(*f)(3.14); /* fn=(*f)(3.14); Error: Not an allowed type */
v=f;
(*v)(3.14+2);
/* f=&fn; Warning: Suspicious pointer conversion */
}
```

C>TYPE CHANGE4.C

```
main(){
int s=2, *ss;
void *p=&s;
char t='A', *tt;
double x=3.14, *xx;
ss=p;
/* printf("%d\n", *p); Error: Not an allowed */
printf("s=%d\n", *ss); /* s=2 */
}
```

```

p=&t;
tt=p;
/* printf("t=%c\n", *p); Error: Not an allowed      */
printf("t=%c\n", *tt);                               /* t=A      */
p=&x;
xx=p;
/* printf("x=%f\n", *p); Error: Not an allowed      */
printf("x=%f\n", *xx);                               /* X=3.140000 */
}

```

5.4 符号扩充、转义符与算术运算转换

TC 允许不同类型的数值型数据进行混合运算,运算时是按一定的规则进行的。下面主要说明如何进行类型自动转换及注意事项。

任一复杂的表达式总可以分解成多个形如

$X \text{ op } Y$

那样的简单表达式,可以把它记为 fXY ,所以

$FX Y = X \text{ op } Y$;

就是赋值语句

$FX Y = fXY$;

它包含着数据转换!为便于叙述,我们将上述 X 和 Y 称为操作数, op 为运算符, $FX Y$ 为赋值变量。下面讨论各类数值型数据间混合运算方法。

1. 数据类型转换级别

在进行运算时,当 X 与 Y 类型不同时,首先要将它们转换为同一类型。最后表达式 fXY 的类型跟它们的类型相同。 X 与 Y 转换后的类型为两者中级别较高的。表 5-1 列出了【数据类型转换级别】的高低顺序。其中横向箭头表示必定的转换,如 float 一定转换为 double 后参加运算;纵向箭头表示运算对象转换方向。

不难看出,数据类型级别和长度相关,因而下述的表 5-7 实际上也将从上到下表示了类型的级别的由低到高。

表 5-1 数据类型转换级别高低和转换方向

高	long double	
↑	double	←float
	unsigned long	
	long	
↓	unsigned	
低	int	←char, short

long double 类型是很少用的,对这种类型使用 $\text{printf}()$ 函数时有可能不如意。

2. 符号扩充

表 5-2 列出了 char、short、float、enum、int 与 double 间类型转换的方法。

表 5-2 算术运算中类型转换方法

转换前数据类型	转换后	转 换 方 法
char 或 signed char	int	符号扩充
unsigned char	int	高八位填 0
short	int	若原数无符号, 就转换为无符号整数
enum	int	值保持不变
float	double	尾数填 0
double	float	double 四舍五入裁尾

注 所谓【符号扩充】是指字符最高位为 0, 则整型变量的高 8 位各位全用 0 填充; 若字符最高位为 1, 则高 8 位各位全用 1 填充。

例 1 意外的符号扩充

```
int x=0xFFFE;
long len;
len=x;
if(len & 0x80000000)
{ ..... } /* 这部分内容永远不会被执行 */
```

例 2 意外的截断

```
int x;
long y=0x10000;
x=y; /* 高 16 位被截去 */
while(x>0)
{ ..... } /* 这部分内容永远不会被执行 */
```

事实上, 例如要搞清 char 类型转换为 int 类型, 只要在源程序中采用语句

```
char ch='\20'; /* 也可以写成 char ch='\020'; */
int c=ch;
```

或

```
unsigned char ch='\200';
int c=ch;
```

然后再用单步跟踪方式观察调试表达式

ch, s, ch, x, ch, d 和 c, x, c, d

即可。表 5-3 列出了一般性结论。

表 5-3 char 或 unsigned char 类转为 int 类时符号扩充

char 或 unsigned char 类			转为 int 类
10 进制	8 进制	16 进制	16 进制
0	\0	0x0	0x0000
1~127	\1~\177	0x01~0x7F	0x0001~0x007F
char 类			转为 int 类
10 进制	8 进制	16 进制	16 进制
-128~-1	\200~\377	0x80~0xFF	0xFF80~0xFFFF
unsigned char 类			转为 int 类
10 进制	8 进制	16 进制	16 进制
128~255	\200~\377	0x80~0xFF	0x80~0xFF

3. 转义符

TC 支持【转义符】。转义符是一些由反斜杠 (\) 引导的字符和串常量值。有时也称为【换码序列 (Escape sequence)】或【非图形字符扩展表示法】。

注意：像转义符 '\t' 初看起来好像是两个字符，实际上只起一个字符作用；其次，不要将 '\0' 与 '0' 混淆起来，后者并不是转义符，而是字符 0，它对应的转义符是 '\x30' (ASCII 值)。表 5-4 列出了转义符的书写格式。

表 5-4 TC 使用的转义符

序 列	值	字 符	功 能
* \a	0x07	BEL	响铃
\b	0x08	BS	退格
\t	0x09	HT	横向跳格
\n	0x0A	LF	换行
* \v	0x0B	VT	纵向跳格
\f	0x0C	FF	换页
\r	0x0D	CR	回车
* \"	0x22	"	西文双引号
\'	0x27	'	西文单引号
* \?	0x3F	?	问号
\\	0x5C	\	反斜杠
\DDD			DDD 表示 1 到 3 个 8 进制数
* \xHHH	0xHHH		HHH 表示 1 到 3 个 16 进制数
* 表示《ANSI》对《K&R》的扩充			

表 5-4 中对十六进制的 \x 不要写成 \X(大写字母) 或 \0x, 斜杠和小 x 之间不应当有数字 0。TC 对字母大小写是敏感的, 即对同一个字母而言, 大小写字母被认为是两个不相同的字母。

由于 TC 对反斜杠作了这种规定, 而它又可能出现在字符串中, 例如, 语句

```
file=fopen("c:\new\tools.dat", "r");
```

中, 由于 \n 和 \t 是转义符, 因此 TC 把它们处理为换行符 (LF) 和横向跳格 (HT), 显然这违反文件名的规定, 所以 DOS 将拒绝接受。即使你把上述语句改成

```
file=fopen("c:\New\Tools.dat", "r");
```

也不正确。运行程序 CHANGE5.C 并输入进行单步跟踪, 便会发现 TC 将 \N 或 \T 处理成字母 N 或 T 了!

```
C>TYPE CHANGE5.C
```

```
#include "stdio.h"
main() {
FILE * fp;
char a[]="e:\New\Tools.dat"; /* 调试表达式: a, s: "e:\NewTools.dat" */
fp=fopen(a, "r");
if (fp!=NULL)
printf("successful!");
else
printf("failure!");
getch();
}
```

要想打开 C 盘上目录 new 中的文件 tools.dat, 正确的做法应使用语句

```
file=fopen("c:\\new\\tools.dat", "r");
```

尽管转义符中常用十进制、十六进制或字符表示, 但是由于 C 最早是用八进制表示二进制的机器上开发的, 因而对它也应正确掌握。对表 5-4 中 \DDD 八进制数应注意几点:

(1) 转义符中的八进制整数可以写成一般数值中的八进制整数, 即可以在八进制整数前加上数字 0。

(2) 因为 TC 允许双字符常量, 由于数字 8 和 9 在八进制中不允许, 因而常量 '\258' 被认为是双字符常量 '\25' 与 '\8', 而 '\8' 又是数字 8, 即这种 '\8' 相当于 '\56' 或 '\x38'。所以对语句

```
char ch='\258';
```

因 ch 在内存中只有一个字节, 所以 ch 实际只接收前一个字符 '\25' (相当于十进制数 21 或十六进制数 0x15); 而对语句

```
int c='\258';
```

则 c=0x3815。

(3) 三个八进制数并不是可以无穷的。从十进制数来看, 它局限于 0~179、200~279 和 300~379

这个范围内。超出范围时会出现

Character constant too long

的错误。而且对 char 类型,在这个范围内,凡以数字 8 和 9 结尾的数字,TC 可以接受,但是 TC 将把它们处理成另一些值,如表 5-5 所示。

表 5-5

转义符 变成 16 进制值 对应的 10 进制数

\8	0x38	56	
\9	0x39	57	
\18~\19	0x1	1	
\28~\29	0x2	2	
.....			
\78~\79	0x7	7	
\80~\89	0x8	56	
\90~\99	0x9	57	
\108~\109	\b	8	/* 只接受 8 进制数 \10 */
\118~\119	\t	9	
\128~\129	\n	10	
\138~\139	\v	11	
\148~\149	\f	12	
\158~\159	\r	13	
\168~\169	\xE	14	
\178~\179	\xF	15	
\208~\209	\x10	16	
.....			
\278~\279	\x17	23	
\308~\309	\x18	24	
.....			
\378~\379	\x1F	31	

转义符是常用的,特别在进行控制时(如对矩阵打印机的控制),有些值是不能通过键盘输入或对某些值(如回车等)不能用可打印字符表示时,常在程序中使用转义符实现。而对可打印字符,你可直接使用而无须使用转义。例如,下面的定义是等价的。

```
char ch1='\A\B';
char ch2='\AB';
char ch3='AB';
char ch4='\101';
```

```

char ch5='\x41';
char ch6=65;
char ch7=0101;
char ch8=0x41;
int ch0='\A\B';
int ch1='\AB';
int ch2='AB';
int ch30='\101\102';
int ch3='\101B';
int ch31='\A\102';
int ch32='\101\B';
int ch4='\x41\x42'; /* 不是 '\x42\x41' */
int ch41='\A\x42';
int ch42='\x41\B';
int ch43='\101\x42';
int ch44='\x41\102';
int ch5=16961;
int ch6=041101; /* 16961 的 8 进制值 */
int ch7=0x4241; /* 不是 0x4142 */

```

是起同样作用的，而下述则不一样。

```

int ch41='\x42\x41'; /* (值不对) */
int ch42='\x41B'; Numeric constant too long
int ch43='\x4241'; Hexadecimal or octal constant too large
int ch44='\x41\X42'; Character constant too long (这里 x 不能大写)
int ch45='\X41\x42';
int ch=16706; /* (值不对) */

```

从上可见对双字符可以用不同进制数混合书写，但应严格按照要求进行。

4. 数据类型转换过程

FXY 的类型 TYPE 是由定义时确定的，所以 fXY 最终将数值转换成 FXY 的类型而后赋给 FXY。这就是说，执行语句

$FXY = fXY = X \text{ op } Y;$

时，数据类型转换的过程见图 5-1。例子见表 5-6。

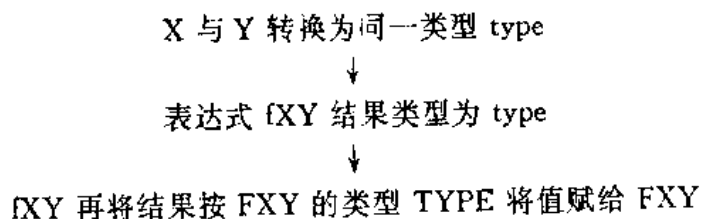


图 5-1

表 5-6

X	Y	XY	TYPE	FXY
10	3	10/3	int	3
			double	3
10.0	3	10.0/3	int	3
			double	3.33333333333333

注: double 型数转换为 int 型数时, 小数部分都被舍去, 但不是四舍五入。

5. TC 规定的数据类型、长度、范围和有效位数

尽管类型转换是由 TC 自动完成的, 但用户必要时应当事先估计转换后的数值范围。当超出规定时可能出错。

C>TYPE CHANGE6.C

```
main()
{
    short x1;
    char x2;
    int x3;
    unsigned int x4;
    unsigned x5;
    long x6;
    float x7;          /* 注释中为跟踪的变量值 */
    double x8;
    x1=32767;          /* 32767=0x7FFF */
    x1=32768;          /* -32768=0x8000 */
    x1=32769;          /* -32767=0x8001 */
    x1=65534;          /* -2=0xFFFE */
    x1=65535;          /* -1=0xFFFF */
    x1=65536;          /* 0=0x0 */
    x1=65537;          /* 1=0x1 */
    /* 1+'A'+1.5/3+8e2*'a'=1+65+0.5+800*97=77666.5
    /* 数据超出类型规定的范围, 导致错误结果
    x1=1+'A'+1.5/3+8e2*'a'; /* 12130=0x2F62 */
    x2=1+'A'+1.5/3+8e2*'a'; /* 'b'=0x62 为低八位值 */
    x3=1+'A'+1.5/3+8e2*'a'; /* 12130 */
    x4=1+'A'+1.5/3+8e2*'a'; /* 12130 */
    x5=1+'A'+1.5/3+8e2*'a'; /* 12130 */
    x1=1+'A'+1.5/3+2*'a'; /* 260=0x104 */
```

```

x2=1+'A'+1.5/3+2*'a';    /* 'b'=0x4 取低八位值 */
x3=1+'A'+1.5/3+2*'a';    /* 260=0x104 */
x4=1+'A'+1.5/3+2*'a';    /* 260=0x104 */
x5=1+'A'+1.5/3+2*'a';    /* 260=0x104 */
x6=1+'A'+1.5/3+8e2*'a';   /* 77666=0x12F62 */
x7=1+'A'+1.5/3+8e2*'a';   /* 77666.5 */
x8=1+'A'+1.5/3+8e2*'a';   /* 77666.5 */
}

```

从上述程序可见,当表达式的值超出赋值变量的范围时将出错。

注意:长度、范围和有效位随机器而异。

表 5-7 TC 规定的数据类型、长度、范围和有效位数

数 据 类 型	长 度	范 围	有效位
unsigned char	8	0~255	
char	8	-128~127	
enum	16	-32768~32767	
unsigned short	16	0~65535	
short	16	-32768~32767	
unsigned int	16	0~65535	
int	16	-32768~32767	
unsigned long	32	0~4294967295	
long	32	-2147483648~2147483647	
float	32	3.4E-38~3.4E+38	7
double	64	1.7E-308~1.7E+308	15~16
long double	80	3.4E-4932~1.1E+4932	
near 指针	16	(包括 near, -cs, -ds, -es, -ss)	
far 指针	32	(包括 far, huge)	

在你的机器上可试用程序 CHANGE7.C 测试数据类型长度。

C>TYPE CHANGE7.C

```

#include "math.h"
main(){
    /* 注释中为 F7 单步跟踪获得的对应变量的值 */
    unsigned char x1=M-PI; /* \x3 */
    char x2=M-PI; /* \x3 */
    unsigned short x3=M-PI; /* 3 */
    short x4=M-PI; /* 3 */
    unsigned int x5=M-PI; /* 3 */
    int x6=M-PI; /* 3 */
}

```

```

unsigned long x7=M-PI; /* 3 */
long x8=M-PI; /* 3 */
float x9=M-PI; /* 3.141593 */
double x10=M-PI; /* 3.14159265358979 */
long double x11=M-PI; /* 3.14159265358979324 */
int z1;
x8=0x8000; /* 32768 */
x8=(int)0x8000; /* -32768 */
x8=0x8000F; /* 524303 x8,xm;0F 00 08 00 */
printf("%d\n",2*3); /* 6 */
printf("%d\n",2F*3); /* 0 */
printf("%f\n",2F*3); /* 6.000000 */
x8=32768*2; /* 65536 */
x8=32768u*2; /* 0 */
x8=32768l*2; /* 65536 */
x8=3276*20000; /* -16000 */
x8=3276L*2000; /* 6552000 */
x8=3276U*2000; /* 63936 */
x8=3276UL*2000; /* 6552000 */
x8=3276LU*2000; /* 6552000 */
z1=sizeof(unsigned char); /* 1 (字节=8 位) */
z1=sizeof(char); /* 1 (字节=8 位) */
z1=sizeof(unsigned short); /* 2 (字节=16 位) */
z1=sizeof(short); /* 2 (字节=16 位) */
z1=sizeof(unsigned int); /* 2 (字节=16 位) */
z1=sizeof(int); /* 2 (字节=16 位) */
z1=sizeof(unsigned long); /* 4 (字节=32 位) */
z1=sizeof(long); /* 4 (字节=32 位) */
z1=sizeof(float); /* 4 (字节=32 位) */
z1=sizeof(double); /* 8 (字节=64 位) */
z1=sizeof(long double); /* 10 (字节=80 位) */
}

```

6. 长整数常量后缀 L 和无符号类型后缀 U

十进制整数常量的允许范围为 $0 \sim 4294967295$ ，或十六进制的 $0x0 \sim 0xFFFFFFFF$ 。负数常量被看成这种不带符号的常量前面加上一个单目的减运算符（-）。TC 也接受这一范围内的数的八进制或十六进制的表示形式。

TC 允许整数常量后面单独带上后缀（suffix）符号 L（或小写字母 l）、U（或小写字母 u），也允许一个整数常量后面同时带上 L 和 U。不允许对变量或宏名后带这种后缀。

带后缀 L 表示将该常量强制为 long 类型，它将按 long 常量参加运算。

带后缀 U 表示将该常量强制为 unsigned 类型，它将按这种类型参加运算。

在表达式中，当一个操作数为常量时，它的类型总是按其实际书写形式确定的，特别当带有这些后缀时更是如此。当一个整数常量本身的实际值大于 65535 时，则不管它是按何种数进制书写的，只要带上后缀 U，便被认为是 unsigned long 类型。

表 5-8 不带 L 或 U 后缀的整数常量的类型

十进制常量	类 型
0~32767	int
32768~65535	long /* 注意! */
65536~2147483647	long
2147483648~4294967295	unsigned long
>4294967295 发生溢出,但不会产生警告信息.其结果是实际值的低字节的内容	

八进制常量	相当十进制数	类 型
00~077777	0~32767	int
0100000~0177777	32768~65535	unsigned int
0200000~01777777777	65536~2147483647	long
020000000000~037777777777	2147483648~4294967295	unsigned long
>037777777777 发生溢出,但不会产生警告信息.其结果是实际值的低字节的内容		

十六进制常量	相当十进制数	类 型
0x0000~0x7FFF	0~32767	int
0x8000~0xFFFF	32768~65535	unsigned int
0x10000~0x7FFFFFFF	65536~2147483647	long
0x80000000~0xFFFFFFFF	2147483648~4294967295	unsigned long
>0xFFFFFFFF 发生溢出,但不会产生警告信息.其结果是实际值的低字节的内容		

7. 浮点数常量后缀 F

TC 一般将浮点数按 double 类型处理,但是如果在浮点数后面加上后缀 F(或小写字母 f),则便强制转为 float 类型。

C>TYPE CHANGE8.C

```
main()
{
    /* 输出: */
    printf("%d\n",sizeof(3));      /* 2 */
    printf("%d\n",sizeof(3.8));    /* 8 */
    printf("%d\n",sizeof(3.4e39)); /* 8 */
    printf("%d\n",sizeof(3.4e39F)); /* 4 */
}
```

5.5 类型的强制转换

在表达式 fXY (或变量及常量等)前面加上

(数据类型标识符)

即

(数据类型标识符)fXY

fXY 结果便有这种数据类型。对变量或常量则以这种类型参与运算。需要指出的是,此后,变

量在参与其它运算时仍是原定义类型。换言之,这种强制转换只对附有这种类型标识符的表达式有效,离开表达式便无效。或者说,变量只以强制转换后的值参与运算,而不改变量在内存中的原有值。

C>TYPE CHANGE9.C

```
main(){
double x=3.1415,xx;
int y1,y2,y3=0;
y1=(int)x+1;
y2=(int)3.1415+1;
y3+=x;
xx=x+1;
printf("y1=%d\n",y1);    /* y1=4      */
printf("y2=%d\n",y2);    /* y2=4      */
printf("y3=%d\n",y3);    /* y3=3      */
printf("xx=%f\n",xx);    /* xx=4.141500 */
printf("x=%f\n",x);      /* x=3.141500  变量原有值不变 */
}
```

5.6 字符串与数值之间的转换

由数字构成的字符串可以转换为数值,反之亦然(参见《数组与字符串》一章)。

第六章 运算符

【运算符, operator】是把操作数连接起来构成表达式的符号。【操作数】可以是常量、变量名、函数或表达式。表 6-1 中列出 TC 的 44 个运算符,也是一般 C 语言采用的。

在构成表达式时各运算符使用的操作数的个数可以有 1~3 个,并且操作数的类型也有一定的要求。根据要求操作数个数的多少,可把运算符分为【单目、双目或三目运算符】。各运算符的名称或分类,各人说法不一,这里只是粗略地按其功能划分。

【运算符的优先级, precedence】是 TC 规定的,当不同级的运算符在一个表达式中同时出现时,优先运算级别高的先起作用,其大体相当于算术四则运算中的口诀:“先乘除后加减”,因为乘除级别高于加减。对同一级别的运算符,按其【结合性 associativity】(或【结合方向】)进行。算术四则运算中的结合性实际只有从左往右,而 C 中增加了从右往左的结合性。为便于记忆,可先记牢这样一点:单目、三目和赋值的结合性均从右往左,其余从左往右。

为帮助阅读,你可以多用优先级最高的圆括号对。特别当发生疑问时,将你认为优先要运算的部分加上圆括号。当你熟练时则应尽量减少这种多余的圆括号,以便生成高质量的代码。

C 语言中有些运算符具有【可换性, commutative】,如 *、+、&、^ 和 | 等,即交换运算符两边的操作数与原来一样,可获得相同结果。

当你要查询某种运算符的作用时,可以在集成环境下用 Ctrl-F4 键打开估算窗,然后输入一个只含常量和该运算符的表达式,从结果窗你很快可知其结果。此法常使人豁然开朗。

表 6-1 运算符的优先级和结合性一览表

优先级	运算符	目次	功 能	运算符类型	结合性
(最高) 15	()		结合	结合	左→右
	[]		标志下标	下标	
	.		取结构成员	分量	
	->		指向结构成员		
14	!	单目	逻辑非	逻辑	左←右
	~		按位求反	字位	
	-		取负	算术	
	++		加 1		
	--		减 1		
	&		取变量地址	指针	
	*		取指针值		
	(类型名)		强制类型	类型转换	
sizeof	求字节数	字节			
13	*		乘	算术	
	/		除		
	%		整数取模		
12	+		加		
	-		减		
11	<<		位左移	字位	
	>>		位右移		

10	<	双目	小于	关系	→
	<=		小于等于		
	>		大于		
	>=		大于等于		
9	==		(恒)等于		
	!=		不等于		
8	&		按位与	字位	
7	^		按位异或		
6			按位或	逻辑	
5	&&		逻辑与		
4			逻辑或		
3	?:	三目	条件表达式	条件	
2	=		赋值	赋值	
	*=		运算后赋值		
	/=				
	%=				
	+=				
	-=				
	>>=				
	<<=				
	&=				
	^=				
=					
1 (最低)	.		顺序求值	逗号	→

在表 6-1 中我们看到,同一运算符可能有几种功能,例如,*,&,-,等等,TC 会正确处理它们,而程序员最好应事先了解可能产生的结果。

6.1 结合运算符 ()

圆括号对级别最高,常用于

将某部分括起来先运算;

将函数参数表与函数名区别开,故圆括号有时也称【函数调用运算符】。

6.2 下标运算符 []

计算数组下标。

注意:一维以上数组下标都应用 [] 括起。如二维数组

```
char a[12][200];
```

是正确的,而

```
char a[12,200];
```

是错误的。

6.3 分量运算符. 和—>

1. 圆点运算符.(外形同小数点符号)是取结构或联合的成员。

C>TYPE OPER1.C

```
main(){
    struct ff{                                /* 定义结构类型 ff          */
        double x,y;
    };
    struct gg{                                /* 定义结构类型 gg          */
        char    f—a;
        int     f—b;
        struct ff ff—2;                      /* gg 的一个成员为 ff 类型的结构 */
    } gg2;                                    /* 定义 gg2 为 gg 类型的结构    */
    gg2.f—a='\n';                            /* 不能用 f—a="\n"; 那样赋值    */
    gg2.f—b=12;
    gg2.ff—2.x=8.0;                          /* 两次引用圆点符            */
    gg2.ff—2.y=9;
    printf("%x\n",gg2.f—a);                  /* a                            */
    printf("%d\n",gg2.f—b);                  /* 12                           */
    printf("%%.1f %.1f\n",gg2.ff—2.x,gg2.ff—2.y); /* 8.0 9.0                    */
}
```

注意:输入调试表达式时也应采用圆点符才能取出结构成员。

```
gg2.ff—2.x; 8.0
gg2.ff—2; { 8.0, 9.0 }
gg2; {'\n', 12, {8.0, 9.0} }
```

把 OPER1.C 修改成 OPER2.C, 增加了数据类型联合。

C>TYPE OPER2.C

```
main(){
    struct ff{
        double x,y;
    };
    struct gg{
        char    f—a;
        int     f—b;
        struct ff ff—2;
    } gg2;
    union {
        int k;
        struct gg gg—3;
    } hh;
    gg2.f—a='\n';
    gg2.f—b=12;
    gg2.ff—2.x=8.0;
```



```

gg2.ff-2.y=9;
hh.k=1;
hh.gg-3.f-a='M';
hh.gg-3.f-b=100;
hh.gg-3.ff-2.x=88.0;
hh.gg-3.ff-2.y=3.14;
printf("%x\n",gg2.f-a);           /* a      */
printf("%d\n",gg2.f-b);           /* 12     */
printf("%.1f %.1f\n",gg2.ff-2.x,gg2.ff-2.y); /* 8.0 9.0 */
printf("%d\n",hh.k);              /* 25677  */
printf("%c\n",hh.gg-3.f-a);       /* M       */
printf("%d\n",hh.gg-3.f-b);       /* 100     */
printf("%f\n",hh.gg-3.ff-2.x);    /* 88.000000 */
printf("%f\n",hh.gg-3.ff-2.y);    /* 3.140000 */
}
/* hh,r: {k:25677, gg-3:{ f-a:'M', f-b:100, ff-2:{ x:88.0, y:3.14 } } }
   gg2,r: { f-a:'\n', f-b:12, ff-2:{ x:8.0, y:9.0 } }
   hh.p: { 25677, { 'M', 100, { 88.0, 3.14 } } }
   gg-2.p: { '\n', 12, { 8.0, 9.0 } }
*/

```

这里需要说明一下为什么 hh.k 最后输出 25677，而不是输出 1？

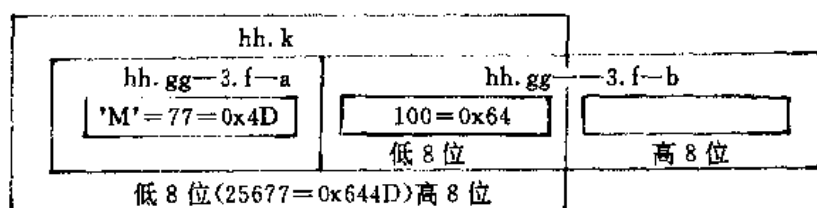


图 6-1 联合 hh 在内存中的前三个字节

因为联合就是将几个不同类型的变量存放同一段内存单元中，由图 6-1 不难看出：
 (1) 整型变量 hh.k 的低 8 位和一字节字符变量 hh.gg-3.f-a 重叠；(2) hh.k 高 8 位与 hh.gg-3.f-b 的低 8 位重叠；(3) 由程序赋值顺序看出，最后 2 字节值为 25677。用单步跟踪很容易看到 hh.k 中值的变化情况。

2. 指向结构成员运算符—>

当一指针指向某结构时，可以用

(* 指针名). 结构成员名

的方法访问结构成员，但这样书写较繁琐，而这类访问又比较多，因而 C 语言中引进专用运算符—>来代替它，即用

指针名—> 结构成员名

所以有人把此运算符称为【指向运算符】。

注意：你如用

(* 指针名). 结构成员名

访问结构成员，则其中的圆括号是必须的，因为圆点运算符的级别高于星号运算符。

C>TYPE OPER3.C

```
main(){
struct ff{
    double x,y;
    }gg={3.14,8};          /* 定义 gg 为 ff 类型的结构 */
struct ff * ptr;           /* 定义 ptr 为 ff 类型的指针 */
ptr=&gg; /* ptr 指向 gg 的首址 */
printf("%f %f\n",ptr->x,ptr->y); /* 3.140000 8.000000 */
printf("%f %f\n",(*ptr).x,(*ptr).y); /* 3.140000 8.000000 */
ptr->x=3.1415;             /* 利用指针向结构成员赋值 */
ptr->y=8e-1;
printf("%f %f\n",ptr->x,ptr->y); /* 3.141500 0.800000 */
(*ptr).x=3.1415;
(*ptr).y=8e-1;
printf("%f %f\n",(*ptr).x,(*ptr).y); /* 3.141500 0.800000 */
}
```

6.4 逻辑运算符!、&& 和||

1. 逻辑非!

其主要是对表达式进行逻辑非运算:

! 表达式

结果得到一个整型数 0 或 1,当表达式值非 0 时运算后有值 0,否则为 1。

操作数的类型可以是字符型、整型、实型或指针型,但 TC 最终以它们为 0 或非 0 来判别它们的真(true)或假(false)。运算结果必为整型。

C>TYPE OPER4.C

```
main(){
int x=12,y=0,z,x0=x;
z=sizeof(! x);
x=! x;
y=! y;
printf("z=%d x=%d y=%d\n",z,x,y); /* z=2 x=0 y=1 */
printf("x=%d ",x0);
if(! x0)
printf("! x<>0\n");
else printf("! x=0\n"); /* x=12! x=0 */
if(x0==0)printf("! x<>0\n");
else printf("x=%d ! x=0\n",x0); /* x=12! x=0 */
}
```

2. 逻辑与运算符&&

【逻辑与运算符】也称【逻辑合取运算符】,只有当逻辑与运算符两边关系表达式均为真(值非 0)时,结果才为真(整型数 1);否则为假(数 0)。

运算顺序严格遵循从左到右的原则,一旦发现左边表达式(也称【左操作数】)为 0,便

不再判断右表达式（或称【右操作数】）的值，结果便为 0。从这一点看，左右操作数的位置是不能互换的，交换后的结果有可能是不一样的。在编译时会出现

Warning: Possibly incorrect assignment

这样的警告信息。由此可见，“结合性”的含义是广泛的，并不只指运算的“顺序”，还包括运算的“中断”。

左右操作数的类型可以不同（字符型、整型、实型或指针型），但 TC 最终以它们为 0 或非 0 来判别它们的真（true）或假（false）。运算结果必为整型。

C>TYPE OPER5.C

```
#define PR(I) printf("%d",y##I),tf(y##I) /* ## 为宏扩展粘结符 */
#define PZ1Z2 printf("z1=%d.1f z2=%d.1f\n",z1,z2)
tf(int x) /* 打印函数 */
{printf("t%s\n",x?"true":"false");}
main(){
int x1=2,x2=3,x3=0,x4=0,y1,y2,y3,y4,y5,y6,y7,y8;
double z1=2,z2=2.0,z3=0,z4=0;
char s1='A',s2=2,s3='\x33',s4=0;
y1=x1 && x2;PR(1); /* 1 true */
y2=x1 && x3;PR(2); /* 0 false */
y3=x1 && z1;PR(3); /* 1 true */
y4=x1 && z2;PR(4); /* 1 true */
y5=x1 && z3;PR(5); /* 0 false */
y6=x2 && x2;PR(6); /* 1 true */
y7=z1 && z2;PR(7); /* 1 true */
y8=z3 && z4;PR(8); /* 0 false */
y1=x1 && s1;PR(1); /* 1 true */
y2=x1 && s2;PR(2); /* 1 true */
y3=x1 && s3;PR(3); /* 1 true */
y4=x1 && s4;PR(4); /* 0 false */
y5=x3 && s4;PR(5); /* 0 false */
y6=x4 && '\0';PR(6); /* 0 false */
y7=(s2-1) && (s3-1);PR(7); /* 1 true */
y8=z2 && s2;PR(8); /* 0 false */
x1=1,x2=2,x3=3,x4=4;z1=z2=1;
PZ1Z2; /* z1=1.0 z2=1.0 */
y1=(z1=x1>x2) && (z2=x3>x4);PR(1); /* 0 false */
/* Warning: Possibly incorrect assignment */
PZ1Z2; /* z1=0.0 z2=1.0 */
x1=1,x2=2,x3=3,x4=4;z1=z2=1;
PZ1Z2; /* z1=1.0 z2=1.0 */
y1=(z1=x2>x1) && (z2=x3>x4);PR(1); /* 0 false */
/* Warning: Possibly incorrect assignment */
PZ1Z2; /* z1=1.0 z2=0.0 */
x1=1,x2=2,x3=3,x4=4;z1=z2=1;
PZ1Z2; /* z1=1.0 z2=1.0 */
y1=(z1==x2>x1) && (z2==x3>x4);PR(1); /* 0 false */
```

```

PZ1Z2;                                /*      z1=1.0 z2=1.0   */
y2=x1==1 && x3==3 && z1+z2==2;PR(2); /* 1      true      */
}

```

3. 逻辑或运算符 ||

【逻辑或运算符】也称【逻辑析取运算符】，只要逻辑或运算符两边关系表达式有一边为真（值非 0）时，结果便为真（整型数 1）；否则为假（数 0）。

像逻辑与运算符一样，运算顺序严格遵循从左到右的原则，一旦发现左操作数非 0，便不再判断右操作数，结果便为 1。左右操作数的位置交换的后的结果有可能是不一样的。在编译时也会出现

Warning: Possibly incorrect assignment

这样的警告信息。

左右操作数的类型可以不同（字符型、整型、实型或指针型），但 TC 最终以它们为 0 或非 0 来判别它们的真（true）或假（false）。运算结果必为整型。

判断闰年的条件有两点：能被 4 整除，但不能被 100 整除；或者能被 400 整除。利用逻辑与和逻辑或运算符，就可以写出判断闰年的语句。

```

C>TYPE OPER6.C
#define PR(I) printf("%d",y##I);tf(y##I)
#define PZ1Z2 printf("z1=%d.1f z2=%d.1f\n",z1,z2)
#define LEAPYEAR(year) (year%4==0 && year%100!=0) || year%400==0
tf(int x)
{printf("\t%s\n",x?"true":"false");}
main(){
int x1=2,x2=3,x3=0,x4=0,y1,y2,y3,y4,y5,y6,y7,y8;
double z1=2,z2=2.0,z3=0,z4=0;
char s1='A',s2=2,s3='\x33',s4=0;
int year;
y1=x1 || x2;PR(1); /* 1      true   */
y2=x1 || x3;PR(2); /* 1      true   */
y3=x1 || z1;PR(3); /* 1      true   */
y4=x1 || z2;PR(4); /* 1      true   */
y5=x1 || z3;PR(5); /* 1      true   */
y6=x2 || x2;PR(6); /* 1      true   */
y7=z1 || z2;PR(7); /* 1      true   */
y8=z3 || z4;PR(8); /* 0      false  */

y1=x1 || s1;PR(1); /* 1      true   */
y2=x1 || s2;PR(2); /* 1      true   */
y3=x1 || s3;PR(3); /* 1      true   */
y4=x1 || s4;PR(4); /* 1      true   */
y5=x3 || s4;PR(5); /* 0      false  */
y6=x4 || '\0';PR(6); /* 0      false  */
y7=(s2-1) || (s3-1);PR(7); /* 1      true   */
y8=z2 || s2;PR(8); /* 1      true   */
x1=1;x2=2;x3=3;x4=4;z1=z2=1;

```

```

PZ1Z2;          /*      z1=1.0 z2=1.0      */
y1=(z1=x1>x2) || (z2=x3>x4);PR(1);      /* 0      false      */
/* Warning: Possibly incorrect assignment      */
PZ1Z2;          /*      z1=0.0 z2=0.0      */
x1=1,x2=2,x3=3,x4=4;z1=z2=1;
PZ1Z2;          /*      z1=1.0 z2=1.0      */
y1=(z1=x2>x1) || (z2=x3>x4);PR(1);      /* 1 true      */
/* Warning: Possibly incorrect assignment      */
PZ1Z2;          /*      z1=1.0 z2=1.0      */
x1=1,x2=2,x3=3,x4=4;z1=z2=1;
PZ1Z2;          /*      z1=1.0 z2=1.0      */
y1=(z1==x2>x1) || (z2==x3>x4);PR(1);      /* 1      true      */
PZ1Z2;          /*      z1=1.0 z2=1.0      */
y2=x1==1 || x3==3 || z1+z2==2;PR(2);      /* 1      true      */
year=1992;      /* 1992 年是闰年, 1993 年则不是闰年      */
y2=LEAPYEAR(year);PR(2);      /* 1      true      */
y2=LEAPYEAR(year+1);PR(2);      /* 0      false      */
}

```

6.5 位运算符 ~、<<、>>、&、^ 和 |

1. 按位取反运算符~又称【位非】，它是将数的各二进制位取反：是0的变成1，是1的变成0。

```

C>TYPE OPER7.C
#define PXY(I) printf("x=%d %x y=%d %x\n",x##I,x##I,y##I,y##I)
#define PZ printf("z1=%lx %ld z2=%lx %ld\n",z1,z1,z2,z2)
main()
{
int x1,x2,x3,x4,y1,y2,y3,y4;
long z1,z2;
x1=0xFF;
y1=~x1,PXY(1);      /* x=255 ff y=-256 ff00      */
x2=x1-1;
y2=~x2,PXY(2);      /* x=254 fe y=-255 ff01      */
x3=32767;
y3=~32767,PXY(3);   /* x=32767 7fff y=-32768 8000 */
x4=-1;
y4=~(-1),PXY(4);    /* x=-1 ffff y=0 0      */
y4=~-1,PXY(4);      /* x=-1 ffff y=0 0      */
z1=1;z2=~z1,PZ;      /* z1=1 1 z2=ffffffe -2      */
z1=0;z2=~z1,PZ;      /* z1=0 0 z2=ffffff -1      */
z1=-1;z2=~z1,PZ;     /* z1=ffffff -1 z2=0 0      */
}

```

2. 左移运算符<<是把整数扩大2的倍数，向左每移一位，数扩大2倍。左移后右边空位上补0。

注意:像

$x = 3 << 1 + 1$

的值是 12 而不是 7。这是因为 + 运算符优先于 << 运算符。要使其值为 7, 应用表达式

$x = (3 << 1) + 1$

3. 右移运算符 >> 是把整数缩小 2 的倍数, 向右每移一位, 正整数缩小 2 倍。对正数 (或无符号数), 右移后左边空位上补 0, 这称【逻辑右移】; 对负数, 符号位原来为 1, 右移后左边有可能用 1 填充, 也有可能用 0 填充, 这取决于所用的计算机系统。

一般情况下, 有符号数右移后左边空位上补上数原符号, 这称【算术右移】。

C>TYPE OPER8.C

```
#define PX printf("      k=%d y1=%d %x y2=%d %x\n", k, y1, y1, y2, y2)
#define PZ printf("      k=%d y3=%ld %lx y4=%ld %lx\n", k, y3, y3, y4, y4)

main()
{
    int k;
    int x1, x2, y1, y2;
    long z1, z2, y3, y4;
    x1=0xFF; x2=-0xFF;
    for(k=0; k<18; k++)
    {
        y1=x1<<k, y2=x2<<k;
        PX;
    }
    x1=0xFF; x2=-0xFF;
    for(k=0; k<10; k++)
    {
        y1=x1>>k, y2=x2>>k;
        PX;
    }
    z1=0xFFFF1111; z2=-0xFFFF1111;
    printf("      z1=%ld z2=%ld\n", z1, z2);
    for(k=0; k<18; k++)
    {
        y3=z1<<k, y4=z2<<k;
        PZ;
    }
    printf("      z1=%ld z2=%ld\n", z1, z2);
    for(k=0; k<18; k++)
    {
        y3=z1>>k, y4=z2>>k;
        PZ;
    }
}

/*
k=0  y1=255 ff    y2=-255 ff01
```

k=1 y1=510 1fe y2=-510 fe02
 k=2 y1=1020 3fc y2=-1020 fc04
 k=3 y1=2040 7f8 y2=-2040 f808
 k=4 y1=4080 ff0 y2=-4080 f010
 k=5 y1=8160 1fe0 y2=-8160 e020
 k=6 y1=16320 3fc0 y2=-16320 c040
 k=7 y1=32640 7f80 y2=-32640 8080
 k=8 y1=-256 ff00 y2=256 100
 k=9 y1=-512 fe00 y2=512 200
 k=10 y1=-1024 fc00 y2=1024 400
 k=11 y1=-2048 f800 y2=2048 800
 k=12 y1=-4096 f000 y2=4096 1000
 k=13 y1=-8192 e000 y2=8192 2000
 k=14 y1=-16384 c000 y2=16384 4000
 k=15 y1=-32768 8000 y2=-32768 8000
 k=16 y1=0 0 y2=0 0
 k=17 y1=0 0 y2=0 0
 k=0 y1=255 ff y2=-255 ff01
 k=1 y1=127 7f y2=-128 ff80
 k=2 y1=63 3f y2=-64 ffc0
 k=3 y1=31 1f y2=-32 ffe0
 k=4 y1=15 f y2=-16 fff0
 k=5 y1=7 7 y2=-8 fff8
 k=6 y1=3 3 y2=-4 fffc
 k=7 y1=1 1 y2=-2 fffe
 k=8 y1=0 0 y2=-1 ffff
 k=9 y1=0 0 y2=-1 ffff
 z1=-61167 z2=61167
 k=0 y3=-61167 ffff1111 y4=61167 eeef
 k=1 y3=-122334 fffe2222 y4=122334 lddde
 k=2 y3=-244668 fffc4444 y4=244668 3bbbc
 k=3 y3=-489336 fff88888 y4=489336 77778
 k=4 y3=-978672 fff11110 y4=978672 eeef0
 k=5 y3=-1957344 ffe22220 y4=1957344 lddde0
 k=6 y3=-3914688 ffc44440 y4=3914688 3bbbc0
 k=7 y3=-7829376 ff888880 y4=7829376 777780
 k=8 y3=-15658752 ff111100 y4=15658752 eeef00
 k=9 y3=-31317504 fe222200 y4=31317504 lddde00
 k=10 y3=-62635008 fc444400 y4=62635008 3bbbc00
 k=11 y3=-125270016 f8888800 y4=125270016 7777800
 k=12 y3=-250540032 f1111000 y4=250540032 eeef000
 k=13 y3=-501080064 e2222000 y4=501080064 lddde000
 k=14 y3=-1002160128 c4444000 y4=1002160128 3bbbc000
 k=15 y3=-2004320256 88888000 y4=2004320256 77778000
 k=16 y3=286326784 11110000 y4=-286326784 eeef0000
 k=17 y3=572653568 22220000 y4=-572653568 ddde0000

```

z1=-61167 z2=61167
k=0 y3=-61167 ffff1111 y4=61167 eef
k=1 y3=-30584 ffff8888 y4=30583 7777
k=2 y3=-15292 ffffc444 y4=15291 3bbb
k=3 y3=-7646 ffffe222 y4=7645 lddd
k=4 y3=-3823 fffff111 y4=3822 eee
k=5 y3=-1912 fffff888 y4=1911 777
k=6 y3=-956 fffffc44 y4=955 3bb
k=7 y3=-478 fffffe22 y4=477 ldd
k=8 y3=-239 ffffff11 y4=238 ee
k=9 y3=-120 fffffff88 y4=119 77
k=10 y3=-60 fffffffc4 y4=59 3b
k=11 y3=-30 ffffffe2 y4=29 ld
k=12 y3=-15 fffffff1 y4=14 e
k=13 y3=-8 fffffff8 y4=7 7
k=14 y3=-4 fffffffc y4=3 3
k=15 y3=-2 fffffffe y4=1 1
k=16 y3=-1 ffffffff y4=0 0
k=17 y3=-1 ffffffff y4=0 0
*/

```

4. 按位与运算符 & 又称【位积】，它是将数的各对应二进制位进行与。位与位之间进行与运算的原则是全为 1，其它为 0：

$0 \& 0 = 0$; $0 \& 1 = 0$; $1 \& 0 = 0$; $1 \& 1 = 1$

它经常用于把特定位清 0，或称将【位屏蔽】。要想把某个数的指定位（假定为 n 位）屏蔽，只要将这个数与一个 n 位为 0 而其余位为 1 的数进行与运算即可。

C>TYPE OPER9.C

```

#define PO(N) printf("%x\n",oldnum##N)
#define PN(N) printf("%x\n",newnum##N)
main(){
int oldnum0=0xff3f; /* 1111 1111 0011 1111 =0xff3f */
int oldnum1=0x1f3f; /* 0001 1111 0011 1111 =0x1f3f */
int bit-mask0=128; /* 0000 0000 1000 0000 =0x80 */
int bit-mask1=127; /* 0000 0000 0111 1111 =0x7f */
int newnum0,newnum1;
newnum0=oldnum0&bit-mask0;PN(0); /* 0000 0000 0000 0000 =0x0000 */
newnum1=oldnum1&bit-mask0;PN(1); /* 0000 0000 0000 0000 =0x0000 */
newnum0=oldnum0&bit-mask1;PN(0); /* 0000 0000 0011 ffff =0x003f */
newnum1=oldnum1&bit-mask1;PN(1); /* 0000 0000 0011 fff =0x003f */
oldnum0=oldnum0>>2;PO(0); /* ffff ffff 1100 ffff =0xffcf */
oldnum1=oldnum1>>2;PO(1); /* 0000 0111 1100 ffff =0x07cf */
newnum0=oldnum0&bit-mask0;PN(0); /* 0000 0000 1000 0000 =0x0080 */
newnum1=oldnum1&bit-mask0;PN(1); /* 0000 0000 1000 0000 =0x0080 */
newnum0=oldnum0&bit-mask1;PN(0); /* 0000 0000 0100 fff =0x004f */
newnum1=oldnum1&bit-mask1;PN(1); /* 0000 0000 0100 fff =0x004f */
}

```


}

5. 按位或运算符 $|$ 又称【位或】，它是将数的各对应二进制位进行或。位与位之间进行或运算的原则是全 0 为 0，其它为 1：

$$0 | 0 = 0; 0 | 1 = 1; 1 | 0 = 1; 1 | 1 = 1$$

它经常用于把特定位置成 1。要想把某个数的指定位（假定为 n 位）置成 1，只要将这个数与一个 n 位为 1 而其余位为 0 的数进行或运算即可。

注意：1 和 1 进行或运算无进位，所以或的概念和二进制加法的概念有区别，后者是要向左进位的。

```
C>TYPE OPER10.C
main()
{
    /* 本例说明位操作和类型关系 */
    int x=0x5154;
    char y=0x11;
    char z1=x|y;          /* 调试表达式 (char)x|y, x:0x55 */
    int z2=x|y;           /* x|y:0x5155 */
    printf("z1=%c z2=%x\n", z1, z2); /* z1=U z2=5155 */
}
```

6. 按位异或运算符 \wedge 又称【位差】，它是将数的各对应二进制位进行异或。位与位之间进行异或运算的原则是相同为 0，相异为 1：

$$0 \wedge 0 = 0; 0 \wedge 1 = 1; 1 \wedge 0 = 1; 1 \wedge 1 = 0$$

与二进制数位加比较

$$0 + 0 = 0; 0 + 1 = 1; 1 + 0 = 1; 1 + 1 = 10$$

不难看出，两数进行位异或运算，相当于对它们的二进制数做特定的加法：当位加中如有进位则舍去进位。因此可以说，这是“不进位的二进制加法”。例如

前 4 位	后 4 位
1 1 0 1	1 0 0 1
\wedge 0 0 0 0	1 1 1 1
1 1 0 1	0 1 1 0

中可见，两数进行异或运算的结果，前 4 位未变，后 4 位是 0 变 1，1 变 0，称为【位翻转】。究其原因，可见只要某位和 1 进行异或运算，该位便被翻转。要想把某个数的指定位（假定为 n 位）翻转，只要将这个数与一个 n 位为 1 而其余位为 0 的数进行异或运算即可。

将一个数与另一个数连续进行两次异或运算，原数不变。这一结论对于处理屏幕上的象点显示是有用的。例如，第一次异或后改变象点颜色，第二次异或后又恢复象点原有颜色。

7. 例

说明直接写显示 RAM 时涉及的位操作。程序详细说明了如何改变屏幕上一象点的颜色的过程。

```
C>TYPE OPER11.C
main()
{
    union mask {
        char c[2];          /* 对 CGA 显示器 */
    };
}
```

```

    int z;          /* 各象点在屏幕上从左到右的顺序存放的 */
    }bit=mask;      /* 在显示方式 4 下, 每字节存 4 个象点 */
int old0=0x0080; /* 0000 0000 1000 0000 象点 0 低字节占第 7,6 位, 颜色 10 */
int old1=0x0020; /* 0000 0000 0010 0000 象点 1 低字节占第 5,4 位, 颜色 10 */
int old2=0x000c; /* 0000 0000 0000 1100 象点 2 低字节占第 3,2 位, 颜色 11 */
int old3=0x0001; /* 0000 0000 0000 0001 象点 3 低字节占第 1,0 位, 颜色 01 */
int old; /* old 表示一字节原 4 个象点:old0,old1,old2,old3. 背景色 00 */
int get=best=bit=0x80; /* 128 */
int mask=bit=0x7f; /* 127 */
int bit=position=1; /* bit=position 表示象点号:1 象点 */
int color=0x0003; /* 0000 0000 0000 0011 指定改变一象点的颜色 11 */
    /* int color=0x0083; 0000 0000 1000 0011 颜色高位非 0 */
int left=2*(3-bit=position); /* 4 需左移次数 */
int right=2*bit=position; /* 2 需右移次数 */
int xor;
int new;
bit=mask.z=0xff3f; /* 1111 1111 0011 1111 屏蔽变量值 */
old=old0 | old1 | old2 | old3; /* 0000 0000 1010 1101 原一字节颜色值 */
xor=color & get=best=bit; /* 0 取得指定颜色的高位值 (bit) */
color=color & mask=bit; /* 0000 0000 0000 0011 指定颜色值被屏蔽了高位 */
color=color<<left; /* 0000 0000 0011 0000 指定颜色值移到 1 象点位 */
    /* 0xffcf=1111 1111 1100 1111 */
    /* 将 1 象点位上置 00, 其余位为 1 */
    /* bit=mask.c[0]=1100 1111 */
if(! xor) /* 联合 mask 使数组元素 c[0] 得低 8 位 */
{ /* 如果指定颜色高位为 0 */
    new=old & bit=mask.c[0]; /* 0000 0000 1000 1101 象点 1 清 0, 其余象点不变 */
    new=new | color; /* 0000 0000 1011 1101 象点 1 换成指定颜色 */
}
else /* 如果指定颜色高位为 1 */
{ /* 如果令 color=0x0083 则 xor=1 */
    new=old | (char)0; /* 0x009D=0000 0000 1001 1101 这是规定 */
    new=new ^ color; /* 象点 1 的颜色为原象点 1 颜色与指定颜色异或 */
}
}

```

程序 OPER12.C 说明数据类型和位操作的关系。

C>TYPE OPER12.C

```

main()
{
    int x=0x81;
    char y0=0x71,y=0x81,y1,y2;
    unsigned char zx=x,zy=y,z0;
    z0=y|(char)0; /* 1000 0001 | 0000 0000 */
    y1=z0^y; /* 1000 0001 ^ 1000 0001 */
    y2=z0^y0; /* 1000 0001 ^ 0111 0001 */
    printf("x=%x %d zx=%x %d\n",x,x,zx,zx); /* x=81 129 zx=81 129 */
}

```

```

printf("y=%x %d zy=%x %d\n",y,y,zy,zy); /* y=ff81 -127 zy=81 129 */
printf("z0=%x %d\n",z0,z0); /* z0=81 129 */
printf("y1=%x %d\n",y1,y1); /* y1=0 0 */
printf("y2=%x %d\n",y2,y2); /* y2=fff0 -16 */
} /* 0xf0=1111 0000 */
/* 调试表达式和 printf 输出的比较:
x,m: 81 00
zx,m: 81
y,m: 81
zy,m: 81
y0,m: 71
y1,m: 00
y2,m: F0
z0,m: 81
*/

```

6.6 负值运算符 —

它是单目运算符，用法是

—操作数

结果是操作数的负值。若操作数是一个无符号数，实际运算结果是 2 的 n 次方减去操作数的差。这里 n 是 CPU 的字长，或者说机器 int 类型的二进制位数。对 IBM 机而言， $n=16$ ，所以 2 的 16 次方为 65536。求一个无符号数的负值，就是求其对 2 的补码。

```

C>TYPE OPER13.C
main()
{
int w1=-1,w2=-w1; /* w2=1 */
unsigned int x=4;
int y=-x,y1=0200000-x; /* 0200000,d: 65536 */
int z=+x;
} /* x=0000 0000 0000 0100 的补码: 对 x 各位求反加 1 */
/* x 的补码=1111 1111 1111 1100 -0xFFFC */
/* y,m: FC FF 对调试表达式，低 8 位在前
y,d: -4
y,x: 0xFFFC
y 与 y1 的结果相同！
y1,m: FC FF
y1,d: -4
y1,x: 0xFFFC
z,m: 04 00
z,d: 4
z,x: 0x4
*/

```

6.7 递增、递减运算符 ++、--

1. 递增运算符 ++ 是将变量的值加 1, 它分两种情形:

(1) 变量名 ++

是先使用变量, 使用过后立即将变量的值加 1;

(2) ++ 变量名

是在使用变量前先将变量值加 1, 然后才使用变量。

程序 OPER14.C 中指出这种加法相当于普通加法。

注意: 对某些类型变量使用这种运算符时可能要受到限制。

注意: 其结合性是从右往左。当你对运算顺序没有把握时, 尽量加上括号试一试。

2. 递减运算符 -- 的情形与递增运算符相仿, 只是将变量的值减 1。

【递增、递减运算符】也称为【自增、自减运算符】。

C>TYPE OPER14.C

main()

```
int x1=4,x2=4;
char y1='a',y2='a';
double z1=3.8,z2=3.8;
unsigned char s1=2,s2=2;
struct a{char ax,bx; }a1={'B','B'};
char *p1=&y1,*p2=&y1;
x1++; /* x1: 5 使用单步跟踪所得值 */
x2=x2+1; /* x2: 5 */
y1++; /* y1: 'b' */
y2=y2+1; /* y2: 'b' */
z1++; /* z1: 4.8 */
z2=z2+1; /* z2: 4.8 */
s1++; /* s1: '\x3' */
s2=s2+1; /* s2: '\x3' */
a1.ax++; /* a1.ax: 'C' */
a1.bx=a1.bx+1; /* a1.bx: 'C' */
p1++; /* p1,p: DS|FFCD */
p2=p2+1; /* p2,p: DS|FFCD */
x1=1,x2=(x1++)+(x1++); /* x1=3, x2=2 */
/* 参加运算的 x1 始终为 1, x1 使用后算第一个
++ , x1=2; 再算第二个 ++ , x1=3 */
x1=1,x2=x1+++x1++; /* x1=3, x2=2 */
/* 编译程序从左到右按优先级将若干字符组成运算符, 故与上句同 */
x1=1,x2=-x1++; /* x1=2, x2=-1 */
/* 负号与 ++ 同级, 故按从右往左结合性, 上句相当于 x2=-(x1++); */
x1=1,++x1; /* x1=2 */
x1=1,x2=(++x1)+(++x1); /* x1=3, x2=6 */
/* x1=1,x2=++x1++x1; Error: Lvalue required */
/* 编译程序不能识别 ++x1++ 这样的表达式 */
```

```

x1=1;x2=-1+x1; /* x1=2, x2=-2 */
/* 以下4行, 仅管执行 printf 后, x1 都为 2, 但有差别 */
x1=1;printf("%d, %d\n", x1, x1++); /* 2, 1 */
/* TC 对函数参数求值顺序是自右而左, 故先执行 x1++, 后执行 x1 */
x1=1;printf("%d, %d\n", x1++, x1); /* 1, 1 */
x1=1;printf("%d, %d\n", x1, ++x1); /* 2, 2 */
x1=1;printf("%d, %d\n", ++x1, x1); /* 2, 1 */
/* 5++; Error: Lvalue required 不能对常量用++ */
/* x2=(x1+1)++; Error: Lvalue required 表达式也不行 */
}

```

6.8 指针运算符 & 和 *

它们都是单目运算符。

对于非数组变量, 有以下结论:

【取地址运算符】& 的一般形式是

指针变量名 = & 变量名;

即把变量的地址赋给指针变量, 或者说指针变量的值是某变量在内存中的地址。

【间接运算符】* 的一般形式是

* 指针变量名 = 变量名;

假定这里指针变量名为 ptr, 它所指的变量为 y, 等号后的变量名为 x, 则上句相当于把变量 x 的值间接赋给指针变量 ptr 所指的变量 y 内。* ptr 即相当于 y, 或 ptr=&x; , y = * ptr。

它们之间的关系可用恒等语句

y = * &x; /* 根据同级时自右向左结合性, 它相当于 y = * (&x); */

即

y = x;

运算符 & 不能用于常量或表达式。如 &8, &(x+4) 等都不合法。

对于数组, 有

指针变量名 = 数组名; /* 与此等价的是 指针变量名 = 数组名[0]; */

或

指针变量名 = & 数组名[i]; /* 取数组第 i+1 个元素的地址 */

注意: 编译程序在遇到这种运算符时如发现有疑问便发出警告。你应当尽量修改源程序使编译不发出这种警告, 以免意外。

```

C>TYPE OPER15.C
main(){
int x=100,y;
int *x0=&x;
char a='AB',b;
char *a0=&a;

```

```

char p[]="AB",q[2];
char *p0=p,*q0=q;
struct mask{
    char c[2];
    int j;
    }mask—bit={"BC",5},mask—bit0;
struct mask *mask0=&mask—bit;
struct ss{int s;}ss1={8};
struct ss *ss0;
y=*x0;printf("y=%d *x0=%d\n",y,*x0); /* y=100 *x0=100 */
y=&x;printf("y=%d *x=%d\n",y,&x); /* y=100 *x=100 */
/* y=&*x; Error: Invalid indirection */
y=99; *x0=y;printf(" *x0=%d\n",*x0); /* *x0=99 */
printf("y+*x0=%d\n",y+*x0); /* y+*x0=18 */
b=*a0;printf("b=%c\n",b); /* b=A */
printf("%s %s\n",p0,p); /* AB AB */
printf("%c %c %c %c\n",p0[0],p0[1],p[0],p[1]); /* A B AB */
printf("q0,p:%p p0,p:%p\n",q0,p0); /* q0,p: FFD6 p0,p: FFD6 */
p0[0]='w';printf("%s\n",p0); /* wB */
p0=&p[1];printf("%s\n",p0); /* B */
p0=&p[0];printf("%s\n",p0); /* wB */
/* q=p; Error: Lvalue required */
/* q=p0; Error: Lvalue required */
q0=p0;
printf("q0,p:%p p0,p:%p\n",q0,p0); /* q0,p: FFD6 p0,p: FFD6 */
printf("q0=%s q=%s\n",q0,q); /* q0=wB q= */
strcpy(q,p);/* 注意,拷贝引起原指针值发生变化! 结果不能预料 */
printf("q0,p:%p p0,p:%p\n",q0,p0); /* q0,p: FF00 p0,p: FFD6 */
printf("q0=%s q=%s\n",q0,q); /* q0= ] q=wB */
printf("%d\n",ss1.s); /* 8 */
ss0=&ss1;
printf("mask0=%p ss0=%p\n",mask0,ss0); /* mask0=FFDE ss0=FFE4 */
printf("%d %d\n",(*ss0).s,ss0->s); /* 8 8 */
ss0=&mask—bit; /* Warning: Suspicious pointer conversion */
printf("mask0=%p ss0=%p\n",mask0,ss0); /* mask0=FFDE ss0=FFDE */
printf("%d %d\n",(*ss0).s,ss0->s); /* 17218 17218 */
printf("%x\n",17218); /* 4342 */
printf("%d\n",ss1.s); /* 8 */
ss0=&x; /* Warning: Suspicious pointer conversion */
printf("%p %d\n",ss0,ss1.s); /* FFCC 8 */
}

```

6.9 强制类型转换运算符 (类型名)

在 C 语言中,可以在任何表达式中将一个(常量、变量或表达式的)值强行转换为另一

种类型的值,以便参加其它运算。其一般形式是

(强制转换类型名)表达式

如 x 原为整型,则语句

```
y=log10((double)(x+100)); /* 注意:它不是 y=log10((double)x+100); */
```

把 $x+100$ 的结果转为双精度数参与运算。值得指出,这样做并不影响变量 x 的原值,也没有改变 x 的原类型。

之所以要这样做,有的是出于移植性考虑,因为这种形式在不同的机器上都会有相同的结果。例如,对求模运算符 $\%$ 规定只对整数进行,假定变量 x 为 `double` 型,譬如定义

```
double x=10;  
int y;
```

要进行求模运算必须用形如

```
y=(int)x % 4;
```

而不能用

```
y=x % 4;
```

另外,表达式中的常数本身也被认为隐含着某种数据类型。如

```
2.0/4;  
2/4;
```

的结果是不同的,前者结果为浮点数 0.5,后者则为整数 0。同时

```
(float)2/4;  
(float)(2/4);
```

结果也不同,这可以用调试表达式测试。更多的情况是,暂时要将某变量按某种类型参加运算,在其它许多场合又不需要这样做时就可使用类型强制转换。

```
C>TYPE OPER16.C  
#include <stdio.h>  
#include <stdlib.h>  
typedef struct {  
    char a[2];  
    int f;  
}OBJECT;  
OBJECT *NewObject()  
{  
    return((OBJECT *)malloc(sizeof(OBJECT)));  
}  
void FreeObject(OBJECT *obj)  
{  
    free(obj);  
}  
main(){
```

```

OBJECT *obj;
obj=NewObject();
obj->a[0]='A';
obj->a[1]='B';
obj->a[2]='\0';
obj->f=100; /* 注意,不是输出 a=AB f=100,因只分配 4 个连续字节 */
printf("a=%s f=%d\n",obj->a,obj->f); /* a=ABd f=100 */
printf("%d\n",sizeof(OBJECT)); /* 4 */
if(obj==NULL){
    printf("failed to creat a new object \n");
    exit(1);
}
printf("OK! obj=%p\n",obj); /* 分配成功,输出 OK! obj=04E0 */
free(obj); /* 释放占用内存,但还未改变内存中内容 */
printf("a=%s f=%d\n",obj->a,obj->f); /* a=ABd f=100 */
}

```

6.10 求字节数运算符 sizeof

它是单目运算符,把它放在变量名前,即

sizeof(变量名)

就求出该变量所占字节数。特别将它放在类型名前,即

sizeof(类型名)

就可求出该类型在一种机器上所占字节数。类型名也可以用用户自定义的。

6.11 算术运算符 +、-、*、/和%

+ 加法运算符或正值运算符

- 减法运算符或负值运算符

* 乘法运算符

/ 除法运算符

%【求模运算符】或【求余运算符】

1. 它们(下面简称为 op)都是双目运算符,应用形式为:

表达式 op 表达式

2. 求模运算符要求左右表达式的值为整型。

3. 运算时涉及的类型转换参见《数据类型的转换》一章。

C>TYPE OPER17.C

```

main(){
    /* 注释中为调试表达式的值 */
    /* double x1=5.0%3; Error: Illegal use of floating point */
    double x2=5%3; /* 2.0 运算符%只适用于整数 */
    double x3=-5%3; /* -2.0 余数符号同被除数 */
}

```



```

double x4=-5.0/3;    /* -1.66666666666667    */
double x5=5.0/3;     /* 1.66666666666667    */
double x6=-5/3;      /* -1.0 商向下取整    */
double x7=5/3;       /* 1.0                */
/* double x8=5/0; Error: Division by zero */
double x9=5/-3;      /* -1.0                */
double x10=5.0/+3;   /* 1.66666666666667    */
}

```

6.12 关系运算符

> 大于
 < 小于
 == 等于
 >= 大于等于
 <= 小于等于
 != 不等于

它们是双目运算符,当关系式成立时,其值为 1; 否则为 0。

注意:语句

```
x=y;
```

与

```
x==y;
```

是两种不同类型的语句。特别不要将

```
if(x==y)u=4;
```

写成

```
if(x=y)u=4;
```

因为对 `if(x=y)` 是先把 `y` 值赋给 `x`,然后在 `x` 非 0 时执行 `u=4;` 语句。所以它和 `if(y=x)` 的含义也是不一样的;而对 `if(x==y)` 则可以写成 `if(y==x)`,因为它们只比较值,而不赋值。

同样,`if((x=y) != u)` 与 `if(x=y != u)` 也是不一样的,这是运算符的优先级造成的。

6.13 赋值运算符

1. 基本赋值运算符

= 等号 /* 赋值与等于是有区别的 */

其一般使用的形式是

操作数 1 = 操作数 2;

程序 OPER18.C 中具体指出它的由左向右的结合性。

```
C>TYPE OPER18.C
```

```
#define PR printf("x=%d y=%d z=%d\n",x,y,z)
```

```

main(){
int x=0,y=0,z=0;
/* int x=y=z=0; Error: Undefined symbol 'y' */
PR;          /* x=0 y=0 z=0 */
x=y=8;PR;    /* x=8 y=8 z=0 */
y=8;x=y;PR;  /* x=8 y=8 z=0 */
x=2+(y=1);PR; /* x=3 y=1 z=0 */
/* x=2+y=1;PR; Error: Lvalue required */
x=(y=2)+(z=1);PR; /* x=1 y=2 z=0 */
y=2;x=y+z-1;PR; /* x=1 y=2 z=0 */
x=(y=5)/(z=2);PR; /* x=2 y=5 z=2 */
y=5;z=2;x=y/z;PR; /* x=2 y=5 z=2 */
x=10;x+=x-=x*x;PR; /* x=-180 y=5 z=2 */
x=10;x-=x*x;x+=x;PR; /* x=-180 y=5 z=2 */
}

```

2. 复合运算符 当有

操作数 1 = 操作数 1 op 操作数 2;

时,可简写成

操作数 1 op= 操作数 2; /* 注意:op 与 = 之间不允许有空格 */

这里 op 可以是 *、/、% 等等。在最简单的情况,假定有变量 x 和 y,则下面式子等价:

x *= y;	等价于 x=x * y;
x /= y;	x=x / y;
x %= y;	x=x % y;
x += y;	x=x + y;
x -= y;	x=x - y;
x >>= y;	x=x >> y;
x <<= y;	x=x << y;
x &= y;	x=x & y;
x ^= y;	x=x ^ y;
x = y;	x=x y;

复合运算符具有书写简单,容易产生高效率的代码(这种写法与编译原理中讲到的“逆波兰式”一致),故常用。

6.14 条件运算符 ?:

它是三目运算符,即有

表达式 1 ? 表达式 2 : 表达式 3

当有多个条件运算符出现时,按自右向左结合性运算。

对只有单个条件运算符的式子而言,总是先算表达式 1,若其值不等于 0,即为真,结果便是表达式 2 的值。表达式 3 不予执行;否则当表达式 1 的值为 0,则为假,结果为表达式 3 的值,表达式 2 不予执行。

注意:允许表达式之间类型不同,但自然应是合乎道理的。类型之间的自动转换参见《数据类型转换》一章。

C>TYPE OPER19.C

```
#define P printf("%d s=%d %d %d %d %d\n",u,s,t,w,x,y,z)
#define Z u=0;s=11;t=2;w=3;x=4;y=5;z=6
main(){
int u,s,t,w,x,y,z;
Z;P; /* 0 s=11 2 3 4 5 6 */
u=s>t?w:x>y?y;z;P; /* 3 s=11 2 3 4 5 6 */
Z; /* 按自右向左结合的证明 */
u=x>y?y;z;P; /* 6 s=11 2 3 4 5 6 */
u=s>t?w;u;P; /* 3 s=11 2 3 4 5 6 */
Z;
u=(s>t?w:x>y)?y;z;P; /* 5 s=11 2 3 4 5 6 */
s=2;t=11;
u=(s>t?w;x>y)?y;z;P; /* 6 s=2 11 3 4 5 6 */
u=x>y;P; /* 0 s=2 11 3 4 5 6 */
u=x<y;P; /* 1 s=2 11 3 4 5 6 */
u=x=y;P; /* 5 s=2 11 3 5 5 6 */
}
```

C>TYPE OPER20.C

```
#define P printf("w=%d u=%d %d %d %d\n",w,u,x,y,z)
#define U u=w!=0?(w>0?x;z):y;P /* 当 w>0,u=x;当 w=0,u=y;当 w<0,u=z */
main(){
int u,w,x=11,y=22,z=33;
w=1; U; /* w=1 u=11 11 22 33 */
w=0; U; /* w=0 u=22 11 22 33 */
w=-1;U; /* w=-1 u=33 11 22 33 */
}
```

C>TYPE OPER21.C

```
#define P printf("%d s=%d %d %d %d %d\n",u,s,t,w,x,y,z)
#define Z u=0;s=0;t=11;w=0;x=22;y=0;z=33
main(){
int u,s,t,w,x,y,z;
Z;P; /* 0 s= 0 11 0 22 0 33 */
s=1;u=(s>0)?(w=x=2):(y=z=3);P; /* 2 s= 1 11 2 2 0 33 */
Z;u=(s>0)?(w=x=2):(y=z=3);P; /* 3 s= 0 11 0 22 3 3 */
Z;s=-1;u=(s>0)?(w=x=2):(y=z=3);P; /* 3 s=-1 11 0 22 3 3 */
}
/* s=1;u=(s>0,t=1)?(w=x=2):(y=z=3); Warning: Code has no effect */
/* s=1;u=(s=t=0)?(w=x=2):(y=z=3);
Warning: Possibly incorrect assignment */
```

C>TYPE OPER22.C

```
#define PRINT(S,T) printf("%" S "\n",T#1)
/* #define PRINT("S",T) printf("%" S "\n",T#1)
```

```

Error;Macro argument syntax error */
/*注意:TC 不在串和字符常量内部扩展宏参数 */
#define N 1;2.0/4
main(){
int x1,x2=11,x3=-2;
double y1,y2;
x1=x2>0? N; PRINT("d",x); /* 1 */
x1=x3>0? N; PRINT("d",x); /* 0 */
y1=x2>0? N; PRINT("f",y); /* 1.000000 */
y1=x3>0? N; PRINT("f",y); /* 0.500000 */
/*.注意跟上面的区别 */
x1=x2? N; PRINT("d",x); /* 1 */
x1=x3? N; PRINT("d",x); /* 1 */
y1=x2? N; PRINT("f",y); /* 1.000000 */
y1=x3? N; PRINT("f",y); /* 1.000000 */
}

C>TYPE OPER23.C
#define PTR ptrxy=ptrx? ptrx:pty;printf("%d\n",*ptrxy);xy=ptrx? x:y;\
printf("%d\n",xy); /* 注意:宏内容太长时用续行符(\) 较好 */
main(){
int x=44,y=88,*ptrx=&x,*pty=&y;
int xy,*ptrxy=&xy;
PTR; /* 44
44 */
ptrx=0;
PTR; /* 88
88 */
}

```

6.15 逗号运算符，

逗号在 C 中经常出现,但并不都是运算符。例如，

- (1)定义变量时起分隔符和运算符的作用；
 - (2)在函数参数表中分隔参数；
 - (3)在 printf 函数的格式串中作一般字符处理,既非分隔符,也不是运算符。
- 只有在逗号表达式中才起运算符的作用。例如，

表达式 1,表达式 2,...,表达式 n

根据结合性,从左向右依次计算每个表达式.对同一个变量因计算多个表达式而有一个以上值时,以最后得到的值为准。即是说逗号运算符起着“顺序求值的作用”。

它的优先级最低。

C>TYPE OPER24.C

```

#define P printf("x=%d, y=%d\n", x, y) /* 宏中串内逗号原样输出 */
main()
{
    int x, y=20;
    x=2+3, x=x*3, P; /* x=15, y=20 */
    x=5, y=y+5, x=8, P; /* x=8, y=25 */
    x=(y=3, 8+2), P; /* x=10, y=3 x取最后一项值 */
    x=(x=3, 8+2), P; /* x=10, y=3 x取最后一项值 */
    for(x=1, y=1, x<3, x++, y++, y++, y+=2) P; /* x=1 y=1 */
                                           /* x=2 y=5 */
    /* 以下语句将产生警告: Code has no effect, 尽量注意不用 */
    y=1, x=2+3, x*3, P; /* x=5 y=1 */
    x=(8+2, y=3), P; /* x=3 y=3 */
    x=(1*2, 2*3), P; /* x=6 y=3 */
    printf("%d %d\n", (x, y)); /* 3 1253 */
    /* 因 (x, y) 被看作一个表达式, 故缺少一个打印目标项 */
}

```

6.16 综合举例

1. 清整数 x 的某一位为 0

```
#define FBUF 0x0004
```

```
x &= ~FBUF;
```

2. 将整数 x 左边 n 个二进制位清 0

```
x &= ((unsigned)~0 >> n);
```

3. 将整数 x 右边 n 个二进制位清 0

```
x &= ~0 << n;
```

4. 求机器字长函数

```
wordlength(void)
```

```
{
```

```
    int k;
```

```
    unsigned v=~0;
```

```
    for(k=1; (v=v>>1)>0; k++); /* 注意此循环体是一个分号 */
```

```
    return (k); /* 返回的 k 值便是字长 */
```

```
}
```

第七章 数组与字符串

7.1 数组

数组是一个具有相同数据类型的链接表,或者说它是同类有序数据的集合。它的每一个数据被称为元素。元素是由数组名和下标唯一确定。下标必须用方括号括起的正整型数表示。数组的下标从 0 开始,相邻两个元素的下标相差 1。数组的数据类型是由其元素的数据类型决定的。数组必须先定义后应用。使用数组时是使用它的元素。

Turbo C 数组有整型、字符型和枚举等。

7.1.1 一维数组

一 定义

格式:存储类型□数据类型□数组名[数组长度];

说明:1. 存储类型有三种:static(静态的)、extern(外部的)和 auto(自动的)。缺省是自动的。

2. 基本的数据类型有整型(int)、浮点型(float, double)、字符型(char)等。

3. 数组名按 C 标识符规定。

4. 数组长度必须用方括号括起(注意:不是圆括号!)。

5. 数组长度可以是 0 或一个正整数,也可以是常量表达式,但不可以是变量。例如,

```
int x=10;
char s[x];
```

是错误的。数组长度和下标的关系是,如果数组长度为 N,则数组元素的下标依次为 0,1,2,...,N-1。或者说数组有 N 个元素,如 s[0]、s[1]、...、s[N-1]。

例

```
C>TYPE ARRAY1.C
```

```
#define SS 23
```

```
#define TT 50
```

```
main(){
    /* 下面注释中为错误定义形式和原因说明 */
    int a=1; /* int a(10); 下标用圆括号 */
    int b[10]; /* int a[10]; 数组名与变量 a 同名, */
    int c[SS+10]; /* int c[]; 数组未给尺寸 */
    int d[SS+TT]; /* int d[SS+a] 常量表达式中有变量 a */
    /* int e[0],e[1],e[2]; 元素分开定义不允许 */
    printf("a=%d\n",a); /* 只定义不赋值允许,编译时也不会产生警告信息 */
}
```

二 数组初始化

数组初始化可以使数组的元素在程序运行前即在编译时得到初值。注意,初始化元素的个

数不宜超过已明确的元素个数。

1. 整型或浮点型数组初始化

C>TYPE ARRAY2.C

```
#define SS 23
#define TT 50
main(){
static int a[2]={1,2};      /* 初值必须用花括号括起,初值和元素严 */
static int b[3]={3,4,5};    /* 格按顺序对应;初值之间应用逗号分隔 */
static int c[3]={6,7};      /* 所有初值构成【初值表】 */
static int d[]={8,9,10};
int e[2]={11,12};           /* e[0]=11,e[1]=12 */
int f[3]={13,14,15};        /* f[0]=13,f[1]=14,f[2]=15 */
int g[3]={16,17};           /* g[0]=16,g[1]=17,而 g[2]=0 */
                             /* 可以只给一部分元素赋初值 */
                             /* 未给初值的元素得值 0 */
int h[]={18,19,20};         /* h[0]=18,h[1]=19,h[2]=20 */
                             /* 可以省写数组长度,其值由初值个数确定 */
                             /* 当给出的初值个数与需要的长度不等时, */
                             /* 应给出长度,或修改初值个数 */
    /* static int e[0]=9,e[1]=10; */ /* 不能对数组元素初始化 */
printf("");
/* 用 F7 热键单步调试,有调试表达式与值:
a[0],15: 1, 2, 3, 4, 5, 6, 7, 0, 8, 9, 10,11,12,13,14
e[0],15: 11,12,13,14,15,16,17,0, 18,19,20,-14,285,1,-16 7 */
```

从中可见未定义处是不确定值。

初值个数不能超过已给定的数组长度。

某些 C 语言中规定只能对静态和外部变量才能赋初值,而对自动变量不行。但是在这里可以看出对自动变量也行。一般有问题时,编译时 TC 会及时提醒你的。

2. 字符数组初始化

下面 ARRAY3.C 中一组字符初始化结果显示在调试表达式中。

C>TYPE ARRAY3.C

```
main(){
/* 注释中为错误的初始化及原因 */
/* char a[2]={ 'a','b','c' }; */ /* 错误:初值个数太多,即: */
/* char b[2]={ 'a','b','\0' }; */ /* Error:Too many initializers */
/* char c[2]={ "a","b" }; */ /* Error:Initializer syntax error */
/* Error:Declaration syntax error */
/* Error:Declaration missing */

char d[2]="ab";
char e[2]='a','b'; /* 用单个字符对数组元素赋初值 */
char f[2]="cd"; /* 用西文双引号括起的字符串称【字符串常量】 */
char g[2]='E'; /* 'c', 'd', '\0' */
char h[2]='F','\0';
char i[2]="G";
/* char j[2]={}; */ /* Error:Initializer syntax error */
```

```

char k[2]={'\0'};
char l[2]="";
/* char m[2]='a'; */ /* Error:Incompatible type conversion */
char a1[]={'a','b','c'};
char b1[]={'a','b','\0'};
/* char c1[]={"a","b"}; */ /* Error:Initializer syntax error */
/* Error:Declaration syntax error */
/* Error:Declaration missing */

char d1[]={"ab"};
char e1[]={'a','b'};
char f1[]="cd";
char g1[]={'E'}; /* 用单个字符赋初值未必在最后加上 '\0' 标志 */
char h1[]={'F','\0'};
char i1[]="G";
/* char j1[]={}; */ /* Error:Initializer syntax error */
char k1[]={'\0'};
char l1[]="";
/* char m1[]='a'; */ /* Error:Incompatible type conversion */
printf();
}

```

现在用 F7 热键对它进行调试（不管警告信息），并且将调试表达式中的显示的重复次数比数组的长度加 1，显示的最后一个值不是数组元素的值，在此仅供参考，以弄清内存中实际存储内容将产生的影响。

调试表达式	值
d[0],3	'a', 'b', 'a'
e[0],2	'a', 'b', 'c'
f[0],3	'c', 'd', 'E'
g[0],3	'E', '\x0', 'F'
h[0],3	'F', '\x0', 'G'
i[0],3	'G', '\x0', '\x0'
k[0],3	'\x0', '\x0', '\x0'
l[0],3	'\x0', '\x0', 'a'
a1[0],4	'a', 'b', 'c', '\x0'
b1[0],4	'a', 'b', '\x0', '\x0'
d1[0],3	'a', 'b', '\x0'
e1[0],3	'a', 'b', 'c'
f1[0],3	'c', 'd', '\x0'
g1[0],2	'E', '□'
h1[0],3	'F', '\x0', 'G'
i1[0],2	'G', '\x0'
k1[0],2	'\x0', '\x1'
l1[0],2	'\x0', 'm'

从上已经可以看出哪些可用的初始化形式，此外还可以得到如下结论：

(1) 初值表中一个字符用一对西文单引号括起；也可用西文双引号将字符串括起后放到花括号内。

(2)TC 规定用转义符 '\0' 表示【字符串结束标志】。'\0' 实际上代表 ASCII 码为 0 的字符,它是一个不可显示字符。有时也称它是【空字符】。

(3)用字符串常量赋初值时,如果数组的最后一个元素未给初值,则 TC 自动在最后一个有效字符后面加上字符串结束标志,这是在编译时完成的。

尤其是在未给定数组长度而用字符串常量赋初值时,总有字符串结束标志存在,所以数组的实际长度为字符串中字符个数加 1。这与用单个字符对数组元素赋初值是不同的。

字符串结束标志要占一个数组元素,因此数组长度为 N 时,如要在数组尾加上结束标志,那么最多只能对前 N-1 个元素(下标为 0,1,...,N-2)赋初值。

(4)初始化时可以对字符串最后一个字符串后面加上空字符,这只要使字符串的字符个数等于数组长度就能实现。是否需要由程序员决定(虽然字符数组最后一个字符不是字符串结束符时也是合法的,但是一般还是必须有的,这样有利于程序对字符串的判别或测定它的长度等)。

(5)可用函数 printf() 检查字符数组初始化后字符串有否结束标志,因为该函数遇字符串结束标志便停止打印。其一般使用格式是

```
printf("%s", 字符数组名);
```

如果打出的字符等于或大于数组长度,或者打出的字符中有其它非初值字符时,均可间接地表明无结束标志。了解这一点是有意义的,因为只要前边打错,后边也就跟着出错。例如

```
printf("%s %d\n", s1, num);
```

如 s1 出错,后面的 num 也往往出错。这跟用 BASIC 的 PRINT 语句的程序员想象中的情形是不一样的。

假定已定义 int y[2]={1,2};, 如用

```
printf("%d", y);
```

以为可以将数组中所有元素的值打印出来,那是错误的。正确的写法是

```
printf("%d,%d", y[0], y[1]);
```

因为数组名并不代表它的全部元素。

(6)允许将转义符作为初始化字符;

(7)初始化值的个数不允许超过元素个数。

(8)注意:字符变量初始化或赋值跟字符数组是不同的。例如,

```
char ch='A'; char c='AC';
```

是可以的,但

```
char m[2]; m="A";
```

```
char n[2]; n='A';
```

```
char s="A";
```

都是错误的。

三 定义与初始化的关系

数组初始化的同时也就对数组进行了定义。在定义方面,字符数组与整型等没有区别,但是在初始化时有区别。在非初始化数组时必须给出数组长度,而在初始化时,既可以给出数组

长度,也可以不给出。在不给出数组长度时必须注意条件和结果。最后,在给定数组长度时初始化值的个数不能超过其元素个数(或不大于数组长度),少一点是可以的。

```
C>TYPE ARRAY4.C
main(){
char a[3];
char b[3]="abc";
char c[3]="ABC";
/* 上两句产生 Warning: 'b' is assigned a value which is never used */
char d[8]={3,4,65,'B','C',68,0};
printf("%s\n",d);          /* 输出结果同监视表达式 d,c */
}
调试表达式      值
a                Undefined symbol 'a'
b[0],4          'a','b','c','\x0'      /* b 数组以 '\0' 结尾 */
c[0],4          'A','B','C','\0'    /* c 数组以空格结尾 */
d,m            03 04 41 42 44 00 00
d,c            " ABCD"              /* 有两个打印机不可打印字符 */
d,s            "\x3\x4ABCD"
d              "x3\x4ABCD"
d[0],8         '\x3','\x4','A','B','C','D','\x0','\x0'
```

7.1.2 二维数组和多维数组

多维数组与二维数组类同。下面只介绍二维数组。

一 二维数组的定义

格式: 存储类型□数据类型□数组名 [数组行数][数组列数];

例

```
#define SS
int a[2][3];
int b[][3];      /* 行数可以省略,但列数不能省略! */
char c[SS][5*6]; /* 下标象一维数组一样定义和标识 */
```

对多维数组,可以为

```
int a[][2][4];    /* 只允许最左边第一个下标可省 */
```

二 二维数组的初始化

初始化的一些约定同一维数组,下面只列出正确的初始化例子。

```
C>TYPE ARRAY5.C
main(){
int a[2][3]={ {1,2,3},{4,5,6}}; /* 不能写成 int a[2,3] */
int b[][3]={1,2,3,4,5,6};      /* 因为逗号是运算符 */
int c[2][3]={ {1},{0,8,1}};
int d[][3]={ {1},{0,8,1}};
int e[][3]={1,0,0,0,8,1};      /* 行数由列数决定 */
```

```

char f[][3]={{"ab"}, {"AB"}};
char g[2][3]={"c", "de"};
char h[2][3]={"XY", "xyz"};
/* char j[2][3]={"XYxyz"}; */ /* Error: Too many initializers */
/* char k[][3]={"XYZxyz"}; */ /* Error: Too many initializers */
printf();
}

```

调试表达式	值
a[0][0],6	1, 2, 3, 4, 5, 6
b[0][0],6	1, 2, 3, 4, 5, 6
c[0][0],6	1, 0, 0, 0, 8, 1
d[0][0],6	1, 0, 0, 0, 8, 1
e[0][0],6	1, 0, 0, 0, 8, 1
f[0][0],6	'a', 'b', '\x0', 'A', 'B', '\x0'
g[0][0],6	'c', '\x0', '\x0', 'd', 'e', '\x0'
h[0][0],6	'x', 'Y', '\x0', 'x', 'y', 'z'

从上对整型二维数组可以得到结论:

(1) 二维数组先将第一行的所有列依次初始化,然后初始化第二行的列,如此直至所有行初始化完毕。

(2) 为清楚起见,同行的初值可用一个花括号括起,不同行间用逗号分隔。使用这种方式时,可以对部分元素赋初值(若某个元素后的初值均为0时,则这些初值可不写出)。

(3) 也可以将所有元素的初值依次摆放,但当某个元素后的初值均为0时,则这些0可不写出。

```

C>TYPE ARRAY6.C
main()
{
int a[][3]={1,2,3,4};
printf("%5d",a[0][0]);
}

```

从下面调试表达式可以看出每个元素均有一个初值与之对应。

调试表达式	值
a[0],2	{1,2,3},{4,0,0}
a	{{1,2,3},{4,0,0}}
a[0][0],6	1,2,3,4,0,0
a[0][1],5	2,3,4,0,0

对二维字符数组也可以有和一维字符数组相似的结论,但它不能象二维整型数组那样将所有初值连起来摆放。为说明其初始化结果,下面再举一例。

```

C>TYPE ARRAY7.C
main()
{
char a[][3]={"", "as"};
char b[][3]={"AS"};
char c[3][3]={"", "my"};
}

```

```

char d[3][3]={"MY"},           /* 输出:      */
printf("a[][3]=%s\n",a);       /*  a[][3]=    */
printf("a[0]=%s a[1]=%s\n",a[0],a[1]); /*  a[0]= a[1]=as */
printf("b[][3]=%s\n",b);       /*  b[][3]=AS   */
printf("c[3][3]=%s\n",c);      /*  c[3][3]=    */
printf("c[1]=%s\n",c[1]);      /*  c[1]=my     */
printf("d[3][3]=%s\n",d);      /*  d[3][3]=MY  */
}

```

调试表达式	值
a	{"", "as"}
a[0],2	"", "as"
a[0][0],6	'\x0', '\x0', '\x0', 'a', 's', '\x0'
a,m	00 00 00 61 73 00
b	{"AS"}
b[0],1	"AS"
b[0][0],3	'A', 'S', '\x0'
b,m	41 53 00
c	{"", "my", ""}
c[0],3	"", "my", ""
c[0][0],9	'\x0', '\x0', '\x0', 'm', 'y', '\x0', '\x0', '\x0', '\x0'
c,m	00 00 00 6D 79 00 00 00 00
d	{"MY", "", ""}
d[0],3	"MY", "", ""
d[0][0],9	'M', 'Y', '\x0', '\x0', '\x0', '\x0', '\x0', '\x0', '\x0', '\x0'
d,m	4D 59 00 00 00 00 00 00 00

上例中清楚地看到 printf 函数遇 '\x0' 便终止打印输出的结果。

实际上常用的是一维字符数组,二维的用得很少。

数组一般不要大于 64K,因为大于 64K 时会在堆外。若实在需要,可采用下述方法(相当于说明了一个大于 64K 的数组 char array[1024][128];):

```

#include <alloc.h>
char (huge *array)[128];
.....
main()
{
.....
array=faralloc (sizeof(*array), 1024);
.....
}

```

这里修饰符 huge 是必须的,以保证动态分配内存的唯一性。

```
/* array.02z */
```

7.2 字符分类

在标头文件 ctype.h 中定义了一些宏,主要是通过查表按 ASCII 码值对单个字符进行分类,所以每个宏都是【谓词】,为真 (ture) 时返回非零值,为假 (false) 时返回零。

有关字符的说明参见《数据类型转换》一章。

7.2.1 分类标志常量

ctype.h 中定义了一些常量,它们可作为分类标志(注释中用的 ASCII 码表示方法为,十进制—十六进制数)。

```
#define __IS_SP      1    /* 是横向制表(9—0x09)、换行(10—0xA)、纵向制 * /
                          /* 表(11—0xB)、换页(12—0xC)、回车(13—0xD) * /
                          /* 和空格(32—0x20)等 * /
#define __IS_DIG    2    /* 是数字 0~9 (48~57—0x30~0x39) * /
#define __IS_UPP     4    /* 是大写字母(65~90—0x41~0x5A) * /
#define __IS_LOW     8    /* 是小写字母(97~122—0x61~0x7A) * /
#define __IS_HEX    16    /* 是字母 A~F 或 a~f * /
#define __IS_CTL    32    /* 是控制字符(0~31—0x0~0x1F, 或 127—0x7F) * /
#define __IS_PUN   64    /* 是标点符号等; ! " # $ % & ' ( ) * + , - . / : ;
                          /* (33~47—0x21~0x2F), , ; < = > ? @ (58~64—0x
                          /* 3A~0x40), [ \ ] ^ _ ` ' (91~96—0x5B~0x60) * /
                          /* { | } ~ (123~126—0x7B~0x7E) * /
```

7.2.2 外部字符数组——ctype 的含义

ctype.h 中还定义了一个外部字符数组——ctype：

```
extern char —Cdecl —ctype[];
```

在 TC 中预置有它们的十六进制值,下面列出从 `--ctype[0]` 开始到 `--ctype[127]` 为止的元素值;从 `--ctype[128]` 以后值均为 00:

00 20 20 20 20 20 20 20 20 20	21 21 21 21 21 20 20 20 20 20
20 20 20 20 20 20 20 20 20 20	20 20 20 01 40 40 40 40 40 40
40 40 40 40 40 40 40 40 40 02	02 02 02 02 02 02 02 02 20 40
40 40 40 40 40 40 14 14 14 14	14 14 04 04 04 04 04 04 04 04
04 04 04 04 04 04 04 04 04 04	04 40 40 40 40 40 40 18 18 18
18 18 18 08 08 08 08 08 08 08	08 08 08 08 08 08 08 08 08 08
08 08 08 40 40 40 40 20	

7.2.3 字符分类宏

不难从宏定义看出宏的功能,下面只列出非零即为真的情况。

```

-1 #define isalnum(c) ((ctype[(c) + 1] & (-IS-DIG | -IS-UPP | -IS-LOW))

```

宏,如 c 是数字 0~9、A~Z 或 a~z,则非零.

```

-2 #define isalpha(c)  (ctype[(c) + 1] & (ISUPP | ISLOW))

```

宏,如 c 是 A~Z 或 a~z 即是英文字母,则非零。

```

-3 #define isascii(c) ((unsigned)(c) < 128)

```

宏,如 $c < 128$ ($0 \sim 127 - 0x0 \sim 0x7F$), 则非零。

```
—4 #define isctrl(c) (—ctype[(c) + 1] & —IS—CTL)
```

宏,如 c 是控制字符 (0~31—0x0~0x1F 或 127—0x7F),非零。

```
—5 #define isdigit(c) (—ctype[(c) + 1] & —IS—DIG)
```

宏,如 c 是数字 0~9,则非零。

- 6 #define isgraph(c) ((c) >= 0x21 && (c) <= 0x7e)
宏,如 c 是除空格符外的所有可打印字符 (33~126—0x21~0x7E),则非零。
- 7 #define islower(c) (—ctype[(c) + 1] & —IS—LOW)
宏,如 c 是字母 a~z,则非零。
- 8 #define isprint(c) ((c) >= 0x20 && (c) <= 0x7e)
宏,如 c 是所有可打印字符 (32~126—0x20~0x7E),则非零。
- 9 #define ispunct(c) (—ctype[(c) + 1] & —IS—PUN)
宏,如 c 是标点符号等(除控制符、数字、空格及大小写字母外的字符),则非零。
- 10 #define isspace(c) (—ctype[(c) + 1] & —IS—SP)
宏,如 c 是制表符、换行、换页、回车或空格符,则非零。
- 11 #define isupper(c) (—ctype[(c) + 1] & —IS—UPP)
宏,如 c 是字母 A~Z,则非零。
- 12 #define isxdigit(c) (—ctype[(c) + 1] & (—IS—DIG | —IS—HEX))
宏,如 c 是十六进制数 0~9、A~F 或 a~f,则非零。

7.2.4 清字符最高位与字符大小写转换

- 1 #define toascii(c) ((c) & 0x7f)
它将 c 最高位(第 7 位)清 0,结果使 c 的最大值是 0x7F,即其值范围为 0~127—0x0~0x7F。它是为与其它系统兼容而设计的。
- 2 #define —toupper(c) ((c) + 'A' - 'a')
宏,将 c 转换成大写字母。仅用于已知 c 是小写字母的情况,否则不能返回需要值。这很容易从宏扩展看出,实质返回的是 c+97-65=c+32。
- 3 int —Cdecl toupper(int ch);
函数,只将字母 ch(不管大小写)转换成大写字母。ch 可以从 -1 到 255 的任一整数,当 ch 不是字母的 ASCII 码值时,函数返回 ch 原值。
- 4 #define —tolower(c) ((c) + 'a' - 'A')
宏,将 c 转换成小写字母。仅用于已知 c 是大写字母的情况,否则不能返回需要值。
- 5 int —Cdecl tolower(int ch);
函数,只将字母 ch(不管大小写)转换成小写字母。ch 可以从 -1 到 255 的任一整数,当 ch 不是字母的 ASCII 码值时,函数返回 ch 原值。CH1.C 中给出了它的定义。

C>TYPE CH1.C

#include "ctype.h"

int tolower(int c)

```
{
    if (c == -1) return (-1);          /* -1 常常用 EOF 来定义 */
    if (isupper((unsigned char)c) ) return (—tower((unsigned char) c));
    else return ((unsigned char)c);
}
```

7.3 字符串操作

7.3.1 字符串的定义

Turbo C 并没有把字符串(或简称【串】)定义成独立的数据类型,而将串定义成字符数组或者一个指向字符的指针。例如,

```
char * string;          /* 指向单个串首址,未赋初值。在内存中只存放指向字符类型
```

```

                                的指针 string    */
char string[80];                /* 内存中留出 80 个存储单位存放串 */
char * string="ABCD";          /* 指向单个串,赋了初值 */
char string[]="ABCD";
char ** string;                /* 指向多个串,未赋初值 */
char * string[]={"abc","ABCDEF"},指向多个串,赋了初值 */

```

在 Turbo C 中,串的长度没有用单独的存储单元存放(可用库函数 strlen() 求其长度)。在串的结尾处一定有一个串终结符('\0' 即 0x0),表明串的结束。这种定义串的方法消除了对串长度的任何限制,因此,只要内存空间允许,一个字符串就可以为任何长度。

一个串可由西文双引号括起来的正文组成,如 "acd"。也可以由多个串连接起来构成,例如,串 "asd" "fghj" 和串 "asdfghj" 是一样的;而串 "asd\" "fghj" 和串 "asd" "fghj" 是一样的。这里反斜杠(\) 和它后面的西文双引号是 Turbo C 的转义符。

可用反斜杠作串正文的续行符。注意,除串正文分成多行或宏定义涉及分行要用反斜杠作续行符外,其它分行情形可不写续行符。

```

C>TYPE STR1.C
main()
{
char * p1, * p2;
p1="asdfgh \
  qwerrrrr"; /* 正文后无续行符不行! 如 q 前无空格,则 p1 和 p2 输出相同 */
p2="asdfgh"
  "qwerrrrr";
printf(p1);
printf("\n");
printf(p2);
}/* 当 q 前有三个空格符时程序输出:asdfgh  qwerrrrr
                                   asdfghqwerrrrr */

```

1. 用数组定义串

如用

```

#define NUM 256
char string[NUM];

```

定义串时,串本身从第 0 个元素(string[0]) 开始,string 只能存放 NUM-1 个“实实在在”的字符,因为最后一个字节要放串终结符。

不能对数组 string 直接赋值,如

```

char string[NUM];
string="as";

```

是错误的。有几种方法可对 string 赋值:

(1) 初始化

```

char string[NUM]="as";
char string[]="as";

```

(2) 串拷贝

```
char string[NUM];
strcpy(string, "as");
(3)将值读入串中
```

C>TYPE STR2.C

```
#include "stdio.h"
#define NUM 6
main()
{
char string[NUM]="as";
/* string 后面的 6( NUM 值 )告诉编译器留出 5 个字符的空间,给一个有 5 个 char 型变量的数组
(第 6 个空间给一个空字符 '\0',以表示串结束)。变量 string 自己并不保存一个字符值,而是保
存着这 5 个 char 变量中第一个在内存中的地址 */
scanf("%s",string);
fflush(stdin); /* 如无此语句清键盘缓冲区,则上次输入的 '\n' 将自动为 gets() 函数接收,结果
用户不能调用 gets() 输数据 */
gets(string); /* 可用调试表达式 string 观察串接收结果 */
}
```

2. 用指针定义串

C>TYPE STR3.C

```
#include "alloc.h"
#define NUM 6
#define PS printf("%s\n",string); /* 注意宏定义以冒号结尾产生的效果 */
main()
{
char * string, * s="WT"; /* 可初始化 */
/* string 或 s 前面的 * 星号告诉编译器,它们是一个指向字符串的指针;或者说它
们保存某个字符的地址,但并未一定留出空间来存储字符。string 与 s 不同的是,
未初始化任何特殊的值;但是串 "WT" 开头地址即字符 W 的地址赋给了指针 s
*/
string="as"; /* 可直接赋值 */
PS /* 这种写法不好,虽然也可用 */
string=(char *)calloc(NUM,1); /* 分配内存空间,使 string 指向自己的串这样它不
会指向别的串。否则像用 scanf() 接收时是很危险的 */
printf("=");PS
strcpy(string,s); /* 也可像数组那样拷贝 */
PS
string=s; /* 用调试表达式 string,p 和 s,p 可以发现至此两指针都有 DS:0194 即
两个指针指向同一个串 */
PS
}
```

7.3.2 串操作函数

对内存块的串操作参见《控制内存块》一章。

一 输出串(写串,参见《文件管理》与《格式输入与输出函数》两章)

- 把串送到预定义流 stdout 中 puts()
- 把串送到流中 fputs()
- 写串到控制台 cputs()
- 格式化输出串 ... printf()

二 串赋值(读串,参见《文件管理》与《格式输入与输出函数》两章)

- 从预定义流 stdin 中读串 gets()
- 从流中读串 fgets()
- 从控制台读串 cgets()
- 格式化输入串 ... scanf()

三 求串长

- 计算串中有效字符的个数,但不包括串结束符('\0') —1 strlen()

四 串合并

- 源串全部字符加到目的串末尾,目的串长度为两串长度之和 —2 strcat()
- 源串最多有 n 个字符加到目的串尾,目的串长度增加 n —3 strncat()

五 串修改

- 将串中所有大写字母转为小写字母 —4 strlwr()
- 将串中所有小写字母转为大写字母 —5strupr()
- 将串中所有字符变成指定字符 —6 strset()
- 将串中前 n 个字符变成指定字符 —7 strnset()
- 将串中所有字符按相反顺序排列(颠倒串) —8 strrev()

六 串比较

- 两个整串比较,串中字母大小写被看成是有区别的 —9 strcmp()
- 两个整串比较,不区分串中字母大小写 —10 stricmp()
- —11 strcmpi()
- 两个串前 n 个字符比较,区分串中字母大小写 —12 strncmp()
- 两个串前 n 个字符比较,不区分串中字母大小写 —13 strnicmp()
- —14 strncmpi()

七 数值与串相互转换(参见《数学函数》一章)

(一)串转换成数值

- 串转换成浮点数,但不记录不能转换字符位置 atof()
- 串转换为双精度值,记录不能转换字符位置 strtod()
- 串转换成整型数 atoi()
- 串转换成长整型值,但不记录不能转换字符位置 atol()
- 串转换为长整型值,记录不能转换字符位置 strtol()

· 串转换为无符号长整数	strtoul()
(二)数值转换成串	
· e 格式浮点数转换成串	ecvt()
· f 格式浮点数转换成串	fcvt()
· 优先生成 f 格式串,否则生成 e 格式串	gcvt()
· 整数转换成串	itoa()
· 长整数转换成串	ltoa()
· 无符号长整数转换成串	ultoa()

八 串拷贝

设源串为 s (串长为 L),目的串为 d,则有

· 源串整个拷到目的串中,返回指针指向 d 首址	—15 strcpy
· 源串整个拷到目的串中,返回指针指向(d 首址 + L)	—16 strcpy
(即 d“尾”)	
· 将源串前 n 个字符拷到目的串中,返回指针指向 d 首址	—17 strncpy
· 为指定串分配空间并复制串到该空间,返回指向复制串的指针	—18 strdup
· 串 s1 头部的 n 个字节拷贝到串 s2 中,相邻偶数和奇数字节位置被交换	—19 swab()

九 搜索串

· 正向搜索串,从串中找出首次出现字符 c 的位置	—20 strchr()
· 反向搜索串,从串中找出首次出现字符 c 的位置	—21 strrchr()
· 正向搜索串 s1,在 s1 中找出首次不在串 s2 中出现的字符的位置	—22 strpbrk()
· 正向搜索串 s1,返回从 s1 首字符开始连续在串 s2 中出现的字符个数	—23 strspn()
· 正向搜索串 s1,返回从 s1 首字符开始连续不在串 s2 中出现的字符个数	—24 strcspn()
· 正向搜索串 s1,返回 s1 中出现串 s2 的位置	—25 strstr()
· 正向搜索串 s1,用串 s2 中的字符分隔串 s1	—26 strtok()

十 错误信息串 (参见《DOS 错误处理函数》一章)

· 返回一个与错误号相对应的串指针	strerror()
-------------------	------------

十一 BASIC 串函数 (自定义函数)

· V\$=MID\$(X\$,n,m)	—27 strMID()
· MID\$(V\$,n,m)=Y\$	—28 MIDstr()
· V\$=LEFT\$(X\$,n)	—29 strLEFT()
· V\$=RIGHT\$(X\$,n)	—30 strRIGHT()
· V\$=STRING\$(n,m)	—31 strSTRN()
· V\$=STRING\$(n,X\$)	—32 strSTRA()
· S\$=SPACE\$(n)	—33 strSPC()

—1 size_t —Cdecl strlen(const char *s);

得到字符串的长度,即串中字符个数,但不包括串尾的空字符('\0')。字符串应有空字符作终止符。无空字符作终止符时,使用本函数将产生不可预料的结果。

本函数相当于 BASIC 中的 LEN(X\$) 函数。

—2 char * —Cdecl strcat(char *dest, const char *src);

将目的串 dest 尾的空字符去掉后加上源串 src 的全部字符及串尾的空字符,结果目的串的长度为原目的串的长度加上源串的长度。由此可见,目的串应有足够大的空间(或对数组应定义有足够多的元素),否则会出问题(占用内存其它空间)。同时原目的串和源串应以空字符作结束标志,否则也会发生意料不到的问题。

函数返回的指针指向目的串首址。

适用于 UNIX 系统。

C>TYPE STR4.C

```
#include "stdio.h"
#include "string.h"
main()
{
char a0[20];
char a1[]="10/8/45 Zhu";
char a2[]="Dec. 12, 1946 Xu";
char a3[]="April 12, 1973 Ji";
char a4[]="Mar. 21, 1977 Huan";
char a5[]={'h','x','i'};
char a6[]="1990";
char a7[]={'H','X','I','\0','h','x','i'};
char *p0=a0, *p1=a0, *p3=a3;
strcpy(a0,""); /* 此句将 a0 清 0, 非常重要。未清 0
前 a0 中可能有不可预料值 */

strcat(a0,a1);
printf("A=%s\n",a0); /* A=10/8/45 Zhu */
printf("len:a1=%d a2=%d\n",strlen(a1),strlen(a2)); /* len:a1=11 a2=14 */
printf("a3: address=%p len=%d\n",a3,strlen(a3));
/* 输出: a3: address=FFAC len=16 */
/* 调试表达式: a3,s:"April 12,1973 Ji" */
/* a3,sm: "April 12,1973 Ji\x0" */
/* 用格式符 s 时,遇空格符打印终止,故相当于 printf 函数 */
/* 用格式符 sm 时,显示数组长度所占存储单元中的内容 */

p1=strcat(a2,a1);
printf("len:a1=%d a2=%d\n",strlen(a1),strlen(a2)); /* len:a1=11 a2=25 */
printf("a3: address=%p len=%d\n",a3,strlen(a3));
/* a3: address=FFAC len=9 */
/* 调试表达式: a3,s:"/8/45 Zhu" */
/* a3,sm: "/8/45 Zhu\x0973 Ji\x0" */
/* 由于数组 a2 原定义空间容纳不了新字符串,便占用了 a3 部分空间! */

printf("B=%s\n",a2); /* Dec. 12,1946 Xu10/8/45 Zhu */
printf("C=%s\n",p1); /* Dec. 12,1946 Xu10/8/45 Zhu */
```

```

/* 返回指针指向目的串 */
printf("D=%s\n",a3); /* /8/45 Zhu */
strcat(a3,a5); /* a5 未以空字符结尾,引起问题! */
printf("E=%s\n",a3); /* /8/45 Zhuhxi 1990 */
strcpy(a0,"");
strcat(a0,a5);
printf("F=%s\n",a0); /* hxi 1990 */
strcat(a0,a7); /* a7 中空字符后的字符未被拷贝过去 */
printf("G=%s\n",a0); /* hxi 1990HXI */
}

```

—3 char * —Cdecl strncat (char *dest, const char *src, size_t maxlen);

类似于 strcat(), 不过只将目的串 dest 尾的空字符去掉后加上源串 src 中的 maxlen 个字符, 串尾再加上空字符, 结果目的串的长度为原目的串的长度加上 maxlen。

函数返回的指针指向目的串首址。

适用于 UNIX 系统。

C>TYPE STR5.C

```

#include "stdio.h"
#include "string.h"
main()
{
    size_t maxlen=18; /* maxlen 超过源串长度 */
    char a0[20]; char a1[]="10/8/45 Zhu";
    char a2[]="Dec. 12, 1946 Xu";
    char a3[]="April 12, 1973 Ji";
    char a4[]="Mar. 21, 1977 Huan";
    char a5[]={'h','x','i'};
    char a6[]="1990";
    char a7[]={'H','X','I','\0','h','x','i'};
    char *p0=a0, *p1=a0, *p3=a3;
    strcpy(a0,"");
    strncat(a0,a6,maxlen); /* 调试表达式: a0,s,"1990" */
    strncat(a1,a6,maxlen); /* a1,s:"10/8/45 Zhu1990"a1,sm,"10/8/45 Zhu1" */
    strcpy(a0,"");
    strncat(a0,a5,maxlen);
    printf("A=%s\n",a0); /* A=hxi */
    strncat(a0,a7,maxlen);
    printf("B=%s\n",a0); /* B=hxiHXI */
    maxlen=2; /* maxlen 小于源串长度 */
    strcpy(a0,"");
    strncat(a0,a6,maxlen); /* a0,s:"19" */
    strcpy(a0,"");
    strncat(a0,a5,maxlen);
    printf("C=%s\n",a0); /* C=hx */
    strncat(a0,a7,maxlen);
    printf("D=%s\n",a0); /* D=hxHX */
}

```

—4 char *—Cdecl strlwr(char *s);

把串 s 中所有大写字母转为小写字母。适用于 UNIX 系统。

—5 char *—Cdecl strupr(char *s);

把串 s 中所有小写字母转为大写字母。适用于 UNIX 系统。

—6 char *—Cdecl strset(char *s, int ch);

把串 s 中所有字符转为字符 ch。适用于 UNIX 系统。

—7 char *—Cdecl strnset(char *s, int ch, size_t n);

把串 s 中前 n 个字符转为字符 ch。如果 n 大于串长度 (strlen(s)), 则 n 相当于串长度。适用于 UNIX 系统。

—8 char *—Cdecl strrev(char *s);

颠倒串中字符顺序,即将串中除空字符及其后续字符外,其余所有字符按相反顺序排列。无错误返回。适用于 UNIX 系统。

C>TYPE STR6.C

```
#include "stdio.h"
```

```
#include "string.h"
```

```
main()
```

```
{
```

```
char a0[20];
```

```
char a1[]="10/8/45 Zhu";
```

```
char a5[]={'h','x','i','s','t'};
```

```
char a7[]={'H','X','I','\0','h','x','i'};
```

```
char *p0=a0, *p1=a1, *p5=a5, *p7=a7;
```

```
strcpy(a0,a1);
```

```
p1=strlwr(a1);
```

```
printf("A=%s\n",a1);          /* A=10/8/45 zhu    */
```

```
printf("B=%s\n",p1);          /* B=10/8/45 zhu    */
```

```
strupr(a1);                  /* 证明 p1 指向 a1  */
```

```
printf("C=%s\n",a1);          /* C=10/8/45 ZHU    */
```

```
strcpy(a1,a0);
```

```
strset(a1,'T');
```

```
printf("D=%s\n",a1);          /* D=TTTTTTTTTTTT */
```

```
strcpy(a1,a0);
```

```
strset(a1,80);                /* 字母 P 的 ASCII 码值是 80 */
```

```
printf("E=%s\n",a1);          /* E=PPPPPPPPPPPP  */
```

```
strcpy(a1,a0);
```

```
strnset(a1,'T',4);
```

```
printf("F=%s\n",a1);          /* F=TTTT/45 zhu    */
```

```
strcpy(a1,a0);
```

```
strnset(a1,'T',strlen(a1)+2);
```

```
printf("G=%s\n",a1);          /* G=TTTTTTTTTTTTT */
```

```
strcpy(a1,a0);
```

```
strrev(a1);
```

```
printf("H=%s\n",a1);          /* H=uhz 54/8/01    */
```

若将上述函数中的 a1 分别用 a5 和 a7 代替则输出如下:

用 a5 代 a1	用 a7 代 a1
A=hxist	A=hxi
B=hxist	B=hxi
C=HXIST	C=HXI
D=TTTTT	D=TTT
E=PPPPP	E=PPP
F=TTTTt	F=TTT
G=TTTTT	G=TTT
H=tsixh	H=IXH

—9 int —Cdecl strcmp (const char *s1, const char *s2);

将串 s1 与串 s2 中字符逐个比较, 结果返回整型值: 如果 s1 小于 s2, 整型值小于 0; 相等, 为 0; 否则整型值大于 0。

比较的原则是根据从左到右, 依次比较字符的 ASCII 码值, 而且串中对同一个英文字母而言, 大写或小写被认为是不等的。适用于 UNIX 系统。

```
C>TYPE STR7.C
#include "stdio.h"
#include "string.h"
main()
{
char a0[]="10/8/45 ZHU",
char a1[]="10/8/45 Zhu",
char a11[]="10/8/45 Zhu",
char a2[]="Dec. 12, 1946 Xu",
char a3[]="April 12, 1973 Ji",
char a4[]="Mar. 21, 1977 Huan",
char a5[]={'h','x','i'},
char a55[]={'h','x','i'},
char a6[]={'h','x'},
char a7[]={'h','x','i','\0','H','X','I'},
char a77[]={'h','x','i','\0','H','X','I'},
char a8[]={'x','i','\0','H','X','I'},
char a88[]={'h','x','i','\0','X','I'},
printf("A=%d\n",strcmp(a0,a1));          /* A=-32 */
printf("B=%d\n",strcmp(a1,a0));          /* B=32 */
printf("C=%d\n",strcmp("10/8/45 ZHU",a1)); /* C=-32 */
printf("D=%d\n",strcmp(a0,"10/8/45 ZHU")); /* D=0 */
printf("E=%d\n",strcmp("10/8/45 ZHU","10/8/45 ZHU")); /* E=0 */
printf("F=%d\n",strcmp(a1,a11));         /* F=0 */
printf("G=%d\n",strcmp(a0,a2));          /* G=-19 */
printf("H=%d\n",strcmp(a3,a4));          /* H=-12 */
printf("I=%d\n",strcmp(a0,a5));          /* I=-55 */
}
```

```

printf("J=%d\n",strcmp(a0,a7));          /* J=-55 */
printf("K=%d\n",strcmp(a5,a55));         /* K=0 */
printf("L=%d\n",strcmp(a5,a6));          /* L=1 */
printf("M=%d\n",strcmp(a5,a7));          /* M=1 */
printf("N=%d\n",strcmp(a5,a8));          /* N=-16 */
printf("O=%d\n",strcmp(a7,a77));         /* O=0 */
printf("P=%d\n",strcmp(a7,a8));          /* P=-16 */
printf("Q=%d\n",strcmp(a7,a88));         /* Q=0 */
printf("R=%d\n",strcmp("X","Y"));        /* R=-1 */
printf("S=%d\n",strcmp("X","YZ"));       /* S=-1 */
printf("T=%d\n",strcmp("XY","Y"));       /* T=-1 */
printf("U=%d\n",strcmp("X","XY"));       /* U=-89 */
}

```

—10 int —Cdecl strcmp(const char *s1,const char *s2);

将串 s1 与串 s2 比较大小,但对两串中的大小写字母不加区分,例如字母 A 和字母 a 被认为是相等的。适用于 UNIX 系统。

—11 #define strcmpi(s1,s2)strcmp(s1,s2)

宏,作用同 strcmp,要用它必须在源文件中包含标头文件 string.h。使用宏是为了与其它编译器兼容而提供的。这包括两层意思,一是 TC 中可直接采用此函数,二是如果其它 C 中的函数和 TC 中某函数功能相似,则可以通过用宏的办法来实现。

—12 int —Cdecl strncmp (const char *s1,const char *s2,size_t maxlen);

比较两串前 maxlen 个字符,区分字符大小写。适用于 UNIX 系统。将程序 STR7.C 中的函数 strcmp 全改成本函数形式,有

*s1	*s2	maxlen=3	maxlen=18
a0	a1	A=0	A=-32
a1	a0	B=0	B=32
"10/8/45 ZHU"	a1	C=0	C=-32
a0	"10/8/45 ZHU"	D=0	D=0
"10/8/45 ZHU"	"10/8/45 ZHU"	E=0	E=0
a1	a11	F=0	F=0
a0	a2	G=-19	G=-19
a3	a4	H=-12	H=-12
a0	a5	I=-55	I=-55
a0	a7	J=-55	J=-55
a5	a55	K=0	K=0
a5	a6	L=1	L=1
a5	a7	M=0	M=0
a5	a8	N=-16	N=-16
a7	a77	O=0	O=0
a7	a8	P=-16	P=-16
a7	a88	Q=0	Q=0
"X"	"Y"	R=-1	R=-1
"X"	"YZ"	S=-1	S=-1
"XY"	"Y"	T=-1	T=-1
"X"	"XY"	U=-89	U=-89

—13 int —Cdecl strncmp (const char *s1, const char *s2, size_t maxlen);

比较两串前 maxlen 个字符,不区分字符大小写。适用于 UNIX 系统。

—14 #define strncmppi(s1,s2,n) strcmp(s1,s2,n)

宏,功能同 strcmp。

—15 char * —Cdecl strcpy (char *dest, const char *src);

将源字符串 src 拷贝到目的串 dest 中, const 修饰符表示可以直接用字符串代入函数表达式中,即

```
strcpy(a0,"10/8/45");
```

那样的语句也是允许的。

拷贝遇源串中空字符便停止,不过空字符要被拷入目的串中。适用于 UNIX 系统。

```
C>TYPE STR8.C
```

```
#include "stdio.h"
```

```
#include "string.h"
```

```
main()
```

```
{
```

```
char a0[20];
```

```
char a1[]="10/8/45 Zhu";
```

```
char a2[]="Dec. 12, 1946 Xu";
```

```
char a3[]="April 12, 1973 Ji";
```

```
char a4[]="Mar. 21. 1977 Huan";
```

```
char a5[]={'h','x','i'};
```

```
char a6[]="1990";
```

```
char a7[]={'h','x','i','\0','H','X','I'};
```

```
char *p0, *p1=a1, *p2=a2, *p3=a3, *p4=a4, *p5=a5, *p6=a6;
```

```
/* a0=a1; */ /* Error: Lvalue required 也不允许 a0="R" */
```

```
/* 但 a0[0]='A'; a0[1]='B'; ... 等对单 */
```

```
/* 个元素赋值允许 */
```

```
strcpy(a0,a1); /* 常用方法 */
```

```
printf("%s\n",a0); /* 输出: 10/8/45 Zhu, 下同 */
```

```
p0=strcpy(a0,a1); /* 注意: 前次拷贝后 a0 中已有 10/8/45 Zhu */
```

```
/* 指针 p0 指向 strcpy 返回指针所指地址 */
```

```
printf("%s %s\n",a0,p0); /* 10/8/45 Zhu 10/8/45 Zhu */
```

```
strcpy(a0,a5); /* a5 中字符串无结束标志! */
```

```
printf("%s\n",a0); /* hxi\x41990 这里 \x4 是 ASCII 码 4 */
```

```
printf("%s\n",p0); /* hxi\x41990 指针 p0 仍指向 strcpy 返回值 */
```

```
strcpy(a0,a7); /* a7 中有 \0, 拷贝只到 \0 为止 */
```

```
printf("%s\n",a0); /* hxi */
```

```
}
```

从用 F7 热键调试中调试表达式的值可以看出,拷贝时将源字符串的结束标志也拷到目的串中了。

调试表达式	值
p0,p	DS:001E
p0,s	"88 Borland Intl."
p1,p	DS:FF88
strcpy	CS:0B33 strcpy
a0,s	"10/8/45 Zhu"
a0,12mx	31 30 2F 38 2F 34 35 20 5A 68 75 00
p0,p	DS:FF74
p0,s	"10/8/45 Zhu"
p1,p	DS:FF88
strcpy	CS:0B33 strcpy
a0,s	"10/8/45 Zhu"
a0,12mx	31 30 2F 38 2F 34 35 20 5A 68 75 00
p0,p	DS:FF74
p0,s	"hxi\t1990"
p1,p	DS:FF88
strcpy	CS:0B33 strcpy
a0,s	"hxi\t1990"
a0,12mx	68 78 69 09 31 39 39 30 00 68 75 00

数组 a0 元素应有足够多,以容纳得下拷入的字符串,否则,如上述程序中将 a0 定义为 a0[2];,那么程序输出结果将可能出错:

```
10/8/45 Zhu
/8/45 Zhu /8/45 Zhu
hxi      1990
hxi      1990
hxi
```

下列程序和调试表达式的值可以看出 strcpy() 的又一拷贝结果。

C>TYPE STR9.C

```
main(){
int n=10;
char *ptr=(char *)malloc(n);
char a[10]="asdfghj"; /* char a[n]="asdfghj"; 不允许 */
strcpy(a,ptr); /* a=ptr 不允许 */
getch();
}
/* a,s: ""
ptr,s: ""
a[0],10x: 0x0, 0x73, 0x64, 0x66, 0x67, 0x68, 0x6A, 0x0, 0x0, 0x0
ptr[0],10x: 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
*/
```

—16 char *—Cdecl stpcpy (char *dest, const char *src);

它与 strcpy 相比,目的串 dest 中结果一样,只是返回的指针值(地址)不一样。strcpy() 返回的指针指向目的串,而 stpcpy() 返回的指针值等于指向目的串的指针值再加上源串 src 的长度。下面以实例说明之。

适用于 UNIX 系统。

C>TYPE STR10.C

```
#include "stdio.h"
#include "string.h"
main()
{
    int len;
    char a0[20];
    char a1[]="10/8/45 Zhu";
    char a2[]="Dec.12,1946 Xu";
    char a3[]="April 12,1973 Ji";
    char a4[]="Mar.21.1977 Huan";
    char a5[]={'h','x','i'};
    char a6[]="1990";
    char a7[]={'h','x','i','\0','H','X','I'};
    char *p0=a0, *p1=a1, *p2=a2, *p3=a3, *p4=a4;
    len=strlen(a1);
    printf("len=%d\n",len); /* len=11 源串长度值,它不括串尾空字符 */
    strcpy(a2,a1);
    printf("A=%s\n",a2); /* A=10/8/45 Zhu */
    printf("B=%s\n",p2); /* b=10/8/45 Zhu 因一开始 p2 指向 a2 */
    p0=a2; /* p0 指向新 a2 */
    p2=strcpy(a2,a1); /* p2 取得 strcpy 返回指针值 */
    p3=a2; /* p3 指向又一次更新的 a2,p3 即是 strcpy 返回指针 */
    p4=p2-len; /* p4 取得返回指针与 len 的差,即 p3 */
    printf("C=p0=%s %p\n",p0,p0); /* C=p0=10/8/45 Zhu */
    printf("D=p2=%s %p\n",p2,p2); /* D=p2 FFA3 */
    printf("E=p3=%s %p\n",p3,p3); /* E=p3=10/8/45 Zhu FF98 */
    printf("F=p4=%s %p\n",p4,p4); /* F=p4=10/8/45 Zhu FF98 */
    strcpy(a2,a5); /* a5 无字符结束标志,拷贝危险 */
    printf("G=%s\n",a2); /* G=hxin1990 */
    strcpy(a2,a7); /* a7 字符串中有空字符 */
    printf("H=%s\n",a2); /* H=hxi 遇空字符即停拷贝 */
}
```

—17 char *—Cdecl strncpy(char *dest,const char *src,size_t maxlen);

将源串 src 中 maxlen 个字符拷贝到目的串 dest 中。

(1) 定义的目的串应用足够的空间容纳拷贝入的字符;

(2) 类型 size_t 在 string.h 中被定义为无符号整数; typedef unsigned size_t;

(3) maxlen 不能大于目的串长度;

(4) 当源串长度等于或大于 maxlen 时,目的串可能无字符串结束标志 \0; 否则,有字符串结束标志,源串后面用若干个空字符,使拷贝字符数补足到 maxlen 个。

适用于 UNIX 系统。

C>TYPE STR11.C

```
#include "stdio.h"
```

```

#include "string.h"
main()
{
size_t maxlen1=5, m2=14;
char a0[14];
char a1[]="10/8/45 Zhu";
char a5[]={'h','x','i'};
char a6[]="1990";
char a7[]={'H','X','I','\0','h','x','i'};
char *p0=a0;          /* 指针 p0 先指向 a0,然后指向 strncpy 的返回值 */
p0=strncpy(a0,a1,maxlen1); /* 以下为调试表达式 a0,14mx 的逐次值 */
p0=strncpy(a0,a5,maxlen1); /* 31 30 2F 38 2F 20 48 00 00 00 91 00 9E 00 */
p0=strncpy(a0,a7,maxlen1); /* 68 78 69 00 00 20 48 00 00 00 91 00 9E 00 */
p0=strncpy(a0,a1,m2);      /* 48 58 49 00 00 20 48 00 00 00 91 00 9E 00 */
p0=strncpy(a0,a5,m2);      /* 31 30 2F 38 2F 34 35 20 5A 68 75 00 00 00 */
p0=strncpy(a0,a7,m2);      /* 68 78 69 00 00 00 00 00 00 00 00 00 00 00 */
}                          /* 48 58 49 00 00 00 00 00 00 00 00 00 00 00 */

```

—18 char *—Cdecl strdup (const char * s);

复制串 s。它是通过 malloc() 函数分配存储空间,空间大小为串长度加 1(数字 1 实际是指串尾的一个空字符,一般讲串长度未包括它),如果没有足够的存储空间,本函数返回空指针。适用于 UNIX 系统。

```

C>TYPE STR12.C
#include "stdio.h"
#include "string.h"
main()
{
int len;
char a1[]="10/8/45 Zhu";
char *p0;
/* p0=(char *)malloc(strlen(a1)+1); 调用 strdup 相当于先执行此语句 */
p0=strdup(a1);
printf("%s\n",p0); /* 屏幕输出: 10/8/45 Zhu */
}

```

—19 void —Cdecl swab (char * from,char * to, int nbytes);

函数将字符串 from 开始的偶数个字节 nbytes 拷贝到指针 to 所指的字符串中,并且拷贝后的结果是这样的:

1. 将原 from 中相邻的偶数字节和奇数字节交换;
2. from 开始的 nbytes 字节(当 nbytes 大于 from 所指串长度时取 from 所指串实际长度)替换 to 所指串开始的 nbytes 个字节。当 to 所指的原串长度大于 nbytes 时,超过部分不变;

3. 当 bytes 取奇数时,实际起作用的是 nbytes-1。

当需要按不同字节顺序将数据从一台机器送到另一台机器上时可考虑使用本函数。

```

C>TYPE STR13.C
#include "stdlib.h"

```

```

#include "stdio.h"
#define P(X) printf( "#X" = "%s\n", X)
#define PP P(str0);P(s0)
main()
{
static char str[33]="abcdefghijk";
char ss[30]="ZYXWVUT";
char *s="ZYXWVUT";
char str0[33], *s0=ss;
strcpy(str0,str);
strcpy(s0,s);
printf("nbetes=4,偶字节时:\n");
swab(str,s,4);
P(str);
P(s);
printf("nbetes=3,奇字节时:\n");
PP;
swab(str0,s0,3);
PP;
printf("ss=%s\n",ss);
printf("nbetes=20,超字节(清 s0)时:\n");
strcpy(s0,"");
swab(str0,s0,20);
PP;
printf("nbetes=20,超字节(未清 s0)时:\n");
swab(str0,s0,20);
PP;
}

```

/* 程序输出:nbetes=4,偶字节时:

```

str=abcdefghijk
s=badcVUT
nbetes=3,奇字节时:
str0=abcdefghijk
s0=ZYXWVUT
str0=abcdefghijk
s0=baXWVUT
ss=baXWVUT
nbetes=20,超字节(清 s0)时:
str0=abcdefghijk
s0=badcfhgji
nbetes=20,超字节(未清 s0)时:
str0=abcdefghijk
s0=badcfhgji          */

```

—20 char * —Cdecl strchr (const char *s, int c);

【正向搜索】字符串s,即从串的第一个字符开始往后逐个搜索字符c(低字节的ASCII值)。它返回一个指针。如果字符c在串中存在,则指针值是串中首次出现c的地址,即指针

指向由起始字符 c 及以后字符组成的串；如果 c 在串中未找到，则返回 NULL（参见图 7-1）。

注意：当串以空字符作终结符时，数 0 总被认为是串的一部分。当 c=0 时，strchr 返回指向这个空字符的指针，指针并不是 NULL。

当串不以空字符结尾时（如 char a1[]={'a','b'}；），有可能出现不可预料的结果。

→ 正向搜索

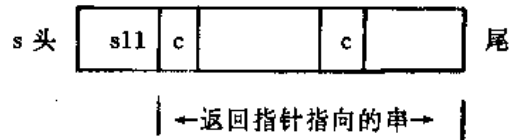


图 7-1 s11 中无字符 c

```
C>TYPE STR14.C
#include "string.h"
#include "stdlib.h"
#define PP(X) p=strchr(a1,X);printf("%d=%s\n",strlen(p),p)
main()
{
char a1[]="abcdefgfh";
int c=90/* 相当于字母 'Z' */，e=101/* 相当于字母 'e' */；
char d='e'；
char *p=(char *)malloc( (strlen(a1)+1)*sizeof(char) )；
PP(c)； /* 程序输出： 0=(null) 相当于 p=0 */
PP(d)； /* 6=edefgh */
PP(e)； /* 6=edefgh */
PP(0)； /* 0= 相当于 p="" */
}
```

—21char *—Cdecl strrchr (const char *s, int c)；

它和 strchr() 类同，不同的是【反向搜索】字符 c，即从串的最后—个字符（通常是空字符）开始往前搜索（参见图 7-2）。

← 反向搜索

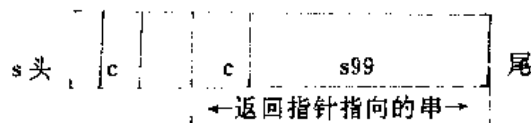


图 7-2 s99 中无字符 c

```
/
C>TYPE STR15.C
#include "string.h"
#include "stdlib.h"
#define PP(X) p=strrchr(a1,X);printf("%d=%s\n",strlen(p),p)
char *strrchr(const char *s,int c)/* 等效函数 */
{
register const char *sl；
for(sl=s+strlen(s);sl>=s;sl--)
```

```

if( *s1==(char)c ) return ( (char *)s1 ); /* 只考虑 c 低字节值 */
return (0);
}
main()
{
char a1[]="abcdefgh";
int c=90 /* 相当于字母 'Z' */ , e=101 /* 相当于字母 'e' */;
char d='e';
char *p=(char *)malloc( (strlen(a1)+1)*sizeof(char) );
PP(c); /* 程序输出:0=(null) 相当于 p=0 */
PP(d); /* 4=efgh */
PP(e); /* 4=efgh */
PP(0); /* 0= */ /* 相当于 p="" */
}

```

—22 char *—Cdecl strpbrk (const char *s1, const char *s2);

每次从 s1 中取出一个字符 *s1, 如 *s1 一旦在 s2 中出现, 则返回 s1 的当前位置; 否则取后一个 *s1 (即 s1++), 然后再在 s2 中寻找, 看 *s1 是否在 s2 中出现。如一次也没找到, 则返回 0 (即 NULL, 参见图 7-3)。

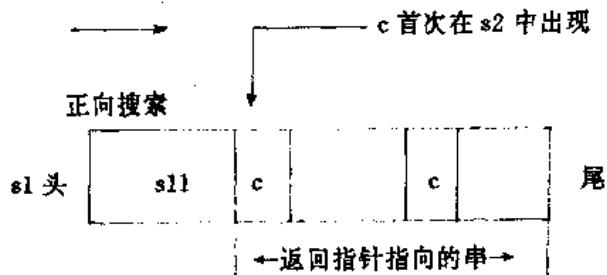


图 7-3 s1 中每一个字符均不在 s2 中出现

C>TYPE STR16.C

```

#include "string.h"
#include "stdlib.h"
#define PP(X) p=strpbrk(a1,X);printf("%d=%s\n",strlen(p),p)
/* 注意:不要将 p=strpbrk(a1,X);写成 p=strpbrk(a1,#X); */
char *strpbrk(const char *s1,const char *s2) /* 等效函数 */
{
register const char *s21;
while( *s1)
{
for(s21=s2; *s21; s21++)
if( *s1== *s21 ) return ( (char *)s1 );
s1++;
}
return (0);
}
main()
{
char a1[]="EFGHAFBCefghafbc";
char *d="ag";

```

```

char *p=(char *)calloc( (strlen(a1)+1)* sizeof(char) ,1);
PP(d);          /* 程序输出:6=ghafbc      */
PP("he");       /*      8=efghafbc      */
PP("fbc");       /*      7=fghafbc       */
PP("afc");       /*      7=fghafbc       */
PP("AFD");       /*      12=AFBCefghafbc */
PP("AD");        /*      12=AFBCefghafbc */
PP(0);          /*      0=(null)        */
}

```

—23 size—t—Cdecl strspn (const char *s1,const char *s2);

设函数返回值为 len,在调用函数前它有值 0。先从 s1 中取出第一个字符 *s1,如果 *s1 在串 s2 中出现,则 len 加 1,然后取 s1 的下一个字符,看它是否在 s2 中出现,假定 *s1 没有在 s2 中出现,则搜索停止,函数返回 len。其过程可参见图 7-4。

注意:搜索后 s1 和 s2 本身未变。

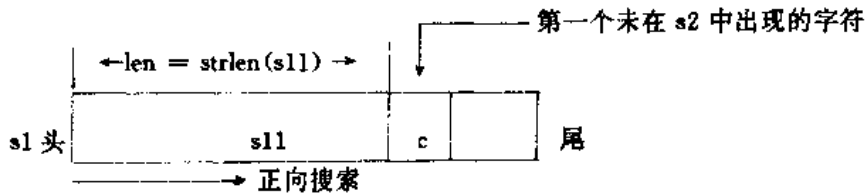


图 7-4 s1 中每一字符都在 s2 中出现过,c 未在 s2 中出现

```

C>TYPE STR17.C
#define STRING—H
#if ! defined (STRING—H) /* 注意:因为用了宏,所以单步调试时不会进入函数 */
typedef unsigned size—t;
size—t strspn(const char *s1,const char *s2)/* 等效函数 */
{
register const char *s21;
int len;
for(len=0; *s1;s1++,len++) /* 从头开始依次取出一个 *s1 和 s2 比较 */
{
for(s21=s2; *s21;s21++) /* 每一次比较开始有 s21=s2 */
if(*s21==*s1)break;
if(*s21==0)break; /* 如 *s1 未在 s2 中出现终止搜索 */
}
return (len);
}
#else
#include "string.h"
#endif
main()
{
char *d[]={ "he", "hef", "hfe", "hgfe", "fg" };

```

```

unsigned int p;
int k;
for(k=0;k<5;k++)
{
    p=strspn("efghafbc",d[k]);
    printf("%d\t",p);
}
}/* 程序输出:12240 */

```

—24 size—t —Cdecl strspn (const char *s1,const char *s2);

设函数返回值为 len,在调用函数前它有值 0。先从 s1 中取出第一个字符 *s1,如果 *s1 没有在串 s2 中出现,则 len 加 1,然后取 s1 的下一个字符,看它是否在 s2 中出现;如果一旦发现了 *s1 在 s2 中出现,则搜索停止,函数返回 len。其过程可参见图 7-5。

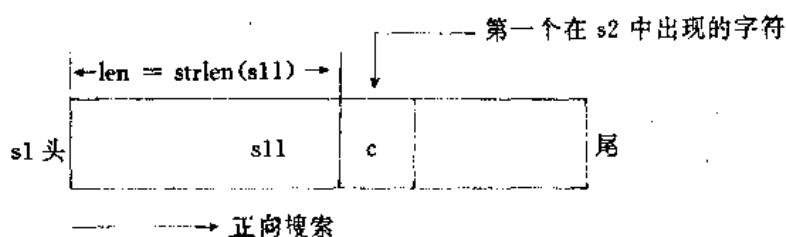


图 7 5 s1 中每一字符都没有在 s2 中出现过,c 首次在 s2 中出现

其等效函数为:

```

typedef unsigned size_t;
size_t strspn(const char *s1,const char *s2)
{register const char *s21;
int len;
for (len=0; *s1;s1++,len++)
    for(s21=s2; *s21;s21++)
        if( *s21==*s1 ) goto bye;
bye:
return (len);
}

```

—25 char *—Cdecl strstr (const char *s1,const char *s2);

如果 s2【非空】,即 s2 不是 ASCII 码 0 时,则在串 s1 正向搜索整个串 s2,如果一旦发现 s1 中有串 s2,便返回 s1 中从包含 s2 首字符开始到 s1 终止符的子串;如找不到,返回 NULL;如果 s2 为空,则返回整个 s1。

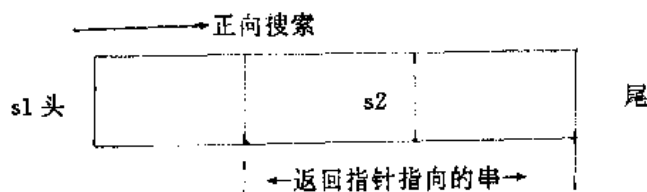


图 7-6(a) s2 非空时



图 7-6(b) s2 为空时

```

C>TYPE STR18.C
#include "string.h"
#define PP(X) p=strstr(a1,X);printf("%d=%s\n",strlen(p),p)
main()
{
char a1[]="EFGHAFBCahefghefbc",
char *d="ag",
char *p;
PP(d);          /* 程序输出:0=(null)          */
PP("he");       /*          9=hefghefbc          */
PP("fbc");       /*          3=fbc                */
PP("aIc");       /*          0=(null)             */
PP("AfD");       /*          0=(null)             */
PP("");         /*          18=EFGHAFBCahefghefbc */
PP(0);          /*          18=EFGHAFBCahefghefbc */
}

```

—26 char * —Cdecl strtok (char * s1,const char * s2);

它搜索时分两个阶段。在第一阶段,将 s1 分成 s10 和 st 两部分。其中 s10 是 strtok() 正向搜索 s2 后的子串,内中每一个字符都在 s2 中出现过;st 也是 s1 的子串,它的第一个字符一定未在 s2 中出现过(参见图 7-7a)。

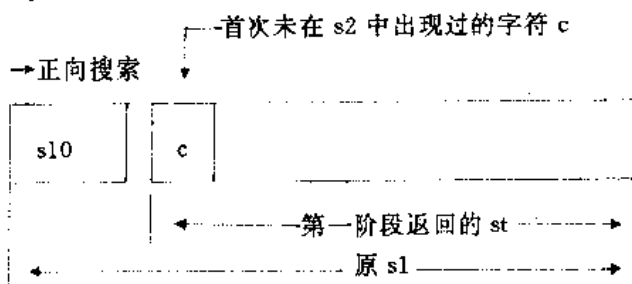


图 7-7(a) s10 中任一字符都在 s2 中出现过

在第二阶段,对 st 正向搜索 s2,结果分成三部分:s11 子串中每一个字符都不会在 s2 中出现;一旦发现 st 中出现 s2 中的字符时,则将该字符处的内存单元置于 ASCII 码 0,即空字符('\0');第三部分是剩余部分,为 st 所指。函数返回 tok(等于 s11+'\0')。值得指出,由于字符串是以空字符作终结标志的,所以原 s1 变成由 s10 和 tok 构成。所以当再次对同一串 a1 调用 strtok() 时(当然 s2 可取与原先不同的内容),a1 串的内容已变化了,而不再是原先的 a1! 这是要注意的。当接着的 strtok(NULL,s2)调用,实际是对剩余的 st 作用(参见图 7-7b)。

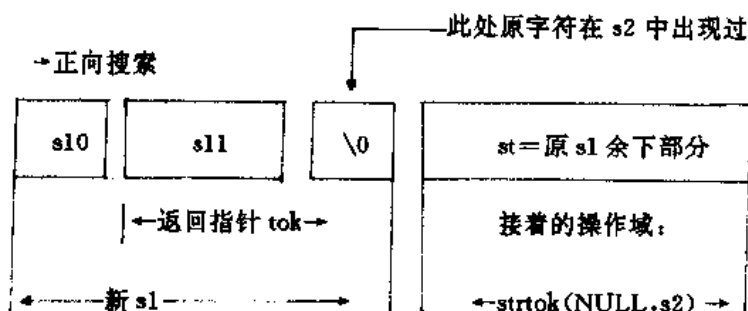


图 7-7(b) s10 中任一字符都在 s2 中出现过, s11 中任一字符均未在 s2 中出现

当 s2 为空字符时, 函数返回 s1; 当首次调用 strtok(NULL, s2) 即 s1 为空字符时, 返回 NULL。

从上可见串 s2 相当于一个由分隔符组成的串。当然, 这种分隔符串是广义的, 也即 s2 不一定是两个字符, 也可以是一个、三个及多个字符, 或者可以是一个单词。而 s1 则可以是零个或多个文本单词序列。

下列程序给出了该函数定义, 可用 F7 键操作进行单步调试跟踪, 理解上述结论。

```
C>TYPE STR19.C
#include "stdio.h"
#include "string.h" /* 如用等效函数可不要此标头文件 */
/*
char *st; /* 全程指针, 初值为 0 */
char *strtok(char *s1, const char *s2) /* 等效函数 */
{
    register const char *sp;
    char *tok;

    /* 第一阶段 */
    if(s1) st = (char *)s1; /* s1 非空, st 指向 s1 */
    while(*st)
    {
        for(sp = s2; *sp; sp++) /* sp 指向 s2 */
            if(*sp == *st) break; /* 如果 *s1 和 s2 中一个字符相同 */
        if(*sp == 0) break; /* 当 *s1 不在 s2 中时终止 while 循环 */
        st++; /* st 表示 s1 当前位置 */
    }
    if(*st == 0) /* 如果 s1 为空返回 NULL */
        return (0);

    /* 第二阶段 */
    tok = st; /* tok 指向 s1 当前位置 */
    while(*st) /* st 仍指向 s1 的当前位置 */
    {
        for(sp = s2; *sp; sp++) /* sp 指向 s2 */
            if(*sp == *st)
            {
                *st++ = 0; /* 首次找到的第一个相同字符处赋 \0 */
                return(tok);
            }
        st++;
    }
}
```

```

    return (tok);
}
*/
main() {
char a1[]="aheisfhgfb";
char a2[]="this is book!";
char a3[]="IT IS A BOOK! THIS IS A BOOK!";
char a4[]="ahfhgfb";
char d[]="is", *e="IS";
char *p;
p= strtok(NULL,d); printf("%s %s\n",p); /* 输出:(null) */
p= strtok(a1,d); printf("%s %s\n",p,a1); /* ahe ahe */
p= strtok(a1,"hh"); printf("%s %s\n",p,a1); /* aa */
p= strtok(a2,"sg"); printf("%s %s\n",p,a2); /* this is book! this is book! */
p= strtok(NULL,"hf"); printf("%s %s\n",p,a2); /* (null) this is book! */
p= strtok(a3,e); printf("%s %s\n",p,a3); /* TIT */
p= strtok(NULL,e); printf("%s %s\n",p,a3); /* A BOOK! TH IT */

p= strtok(a4,""); printf("%s %s\n",p,a3); /* ahfhgfb IT */
p= strtok("0012034","03"); printf("%s\n",p); /* 12 */
p= strtok("", "a"); printf("%s\n",p); /* (null) */
}

```

C>TYPE STR20.C

```

#include "stdio.h"
#include "string.h"
#define PS(P,Y) printf("%s [%s=%s] ",P,Y)

```

```

PP(char *s,char *s5)

```

```

{
char *p1,*p2,*p3;
printf("%s:",s);
p1= strtok(s,s5); PS(p1,s);
p2= strtok(NULL,s5); PS(p2,s);
p3= strtok(NULL,s5); PS(p3,s);
printf("\n");
}

```

```

main()

```

```

{
char *s1="Oct. 04. 1993";
char *s2="10/04/1993";
char *s3="10/. 04. /1993";
char *s5=".,-/" ;
PP(s1,s5); /* 输出:Oct. 04. 1993:Oct [s=Oct] 04 [s=Oct] 1993 [s=Oct] */
PP(s2,s5); /* 10/04/1993:10 [s=10] 04 [s=10] 1993 [s=10] */
PP(s3,s5); /* 10/. 04. /1993:10 [s=10] 04 [s=10] 1993 [s=10] */
}

```

—27 char * —Cdecl strMID(char * x,unsigned char n,unsigned char m); <自定义.h>

在 BASIC 语言中有一个求子串函数 V\$=MID\$(X\$,n,m), V\$ 得到从 X\$ 第 n 个字符开始、返回串长度为 m 的字符串。如果省去 m, 或者第 n 个字符的右端字符个数比 m 小, 则返回以第 n 个字符开始的所有最右边字符。如果 m=0, 或者 X\$="", 或者 n>strlen(X\$)(即 X\$ 的长度), 则 V\$=0。

函数 strMID() 实现相同功能。稍有不同的是函数中 m 不能象 BASIC 中可省略(为此, 必要时你可以令它等于任意一个大于串长的值即可)。

C>TYPE STR21.C

```
#include "stdio.h"
#include "string.h"
#include "mem.h"
#include "alloc.h"
char * strMID(char * x,unsigned char n,unsigned char m)
{
char v[256]; /* 不要用 static char v[256]; 那样语句清内存单元 */
int j,len=strlen(x);
setmem(v,255,'\0'); /* 清内存单元 */
/* strnset(v,'\0',255); 注意, 此句只能保证第一个字符为 '\0',
而不能保证所有的存储单元为 '\0'。这是要注意的。 */
if(m==0 || *x==0 || n>len || n<1) return (v);
if(m>(len-n+1)) m=len-n+1;
x+=n-1;
strncpy(v,x,m);
len=strlen(x);
/* if (len>=m) v[m]='\0'; 此句只在用 strnset(v,'\0',255); 时才用以保证当 len>=m 时串
以空字符结束 */
return (v);
}
main()
{
char * str=(char *)calloc(80,1), * v;
unsigned char n,m;
printf("输入一个串:");
gets(str); /* 输入:ASDFGHJK */
printf("输入开始 n, 串长 m:");
scanf("%u,%u",&n,&m); /* 输入:2,3 */
v=strMID(str,n,m);
printf("%s\n",v); /* 输出:SDF */
}
```

—28char * —Cdecl MIDstr(char * v,unsigned char n,unsigned char m,char * y); <自定义.h>

在 BASIC 语言中有一个将字符串 V\$ 中的部分或全部用另一个串 Y\$ 的子串或全部代替的函数 MID\$(V\$,n,m)=Y\$。注意: 不管怎么替代, 串 V\$ 的长度不变。函数 MID-

set()便相当此功能。除函数形式稍有不同外,函数中 m 不能象 BASIC 中可以省略(为此,必要时你可以令它等于或大于串 y 的长度的任意一个值即可)。

```
C>TYPE STR22.C
#include "stdio.h"
#include "string.h"
#include "alloc.h"
char *MIDstr(char *v,unsigned char n,unsigned char m,char *y)
{
    int lenv=strlen(v),leny=strlen(y);
    char temp[255],dst[255];
    memset(temp,'\0',255); /* 清内存单元 */
    memset(dst,'\0',255);
    if(m==0 || *v==0 || n>lenv || *y==0 || n<1 )return (v); /* 返回原串 */
    if(m> (lenv-n+1) )m=lenv-n+1;
    if(n>1)strncpy(dst,v,n-1); /* 取目的串的第一子串 */
    if(m<leny)strncpy(temp,y,m);
    else {
        strcpy(temp,y);
        m=strlen(y);
    }
    strcat(dst,temp); /* 取目的串的第二子串 */
    v += n+m-1;
    strcat(dst,v); /* 取目的串的第三子串 */
    return (dst);
}
main()
{
    char *vv=(char *)calloc(80,1);
    char yy[80];
    unsigned char n,m;
    printf("输入一个源串 V$:");
    gets(vv); /* 输入:ASDFGHJK */
    printf("输入用于替代的子串 Y$:");
    gets(yy); /* 输入:qw */
    printf("输入开始 n,替代字符数 m:");
    scanf("%u,%u",&n,&m); /* 输入:2,3 */
    vv=MIDstr(vv,n,m,yy);
    printf("%s\n",vv); /* 输出:AqwFGHJK */
}
```

—29char *—Cdecl strLEFT(char *x,unsigned char n) <自定义.h>

相当 BASIC 中 LEFT\$ 函数,即 V\$=LEFT\$(X\$,n),返回字符串 X\$ 最左边 n 个字符到 V\$ 中。

```
C>TYPE STR23.C
#include "stdio.h"
#include "string.h"
```

```

#include "mem.h"
#include "alloc.h"
char *strMID(char *x,unsigned char n,unsigned char m)
{
char v[256];
int j,len=strlen(x);
setmem(v,255,'\0');
if(m==0 || *x==0 || n>len || n<1)return (v);
if(m> (len-n+1) )m=len-n+1;
x +=n-1;
strcpy(v,x,m);
len=strlen(x);
return (v);
}
char *strLEFT(char *x,unsigned char n)
{
return (strMID(x,1,n));
}
main()
{
char *v=(char *)calloc(80,1);
unsigned char nn;
printf("输入一个源串 X$ :");
gets(v); /* 输入:ASDFGHJK */
printf("输入左取字符数 n:");
scanf("%u",&nn); /* 输入:2 */
v=strLEFT(v,nn);
printf("%s\n",v); /* 输出:AS */
}

```

—30char *—Cdecl strRIGHT(char *x,unsigned char n) <自定义.h>

相当 BASIC 中 RIGHT\$ 函数,即 V\$=RIGHT\$(X\$,n),返回字符串 X\$ 最右边 n 个字符到 V\$ 中。

```

C>TYPE STR24.C
#include "stdio.h"
#include "string.h"
#include "mem.h"
#include "alloc.h"
char *strMID(char *x,unsigned char n,unsigned char m)
{
char v[256];
int j,len=strlen(x);
setmem(v,255,'\0');
if(m==0 || *x==0 || n>len || n<1)return (v);
if(m> (len-n+1) )m=len-n+1;
x +=n-1;

```

```

strcpy(v,x,m);
len=strlen(x);
return (v);
}
char *strRIGHT(char *x,unsigned char n)
{
if(n>=strlen(x)) return (x);
return (strMID(x,strlen(x)-n+1,n));
}
main()
{
char *v=(char *)calloc(80,1);
unsigned char nn;
printf("输入一个源串 X$:");
gets(v); /* 输入:ASDFGHJK */
printf("输入右取字符数 n:");
scanf("%u",&nn); /* 输入:2 */
v=strRIGHT(v,nn);
printf("%s\n",v); /* 输出:JK */
}

```

—31 char *—Cdecl strSTRN(unsigned char n,unsigned char m) <自定义.h>

—32 char *—Cdecl strSTRA(unsigned char n, char *x) <自定义.h>

—33 char *—Cdecl strSPC(unsigned int n) <自定义.h>

strSTRN() 相当于 BASIC 中的 V\$=STRING\$(n,m),它得到由指定 ASCII 码值 m 的 n 个拷贝组成的字符串; strSTRA() 相当于 BASIC 中的 V\$=STRING\$(n,X\$),它得到串 X\$ 首字符的 n 个拷贝组成的串;特别地,当 m=32(空格符)时,便得到 BASIC 中 SPACE\$ 函数, S\$=SPACE\$(n),即生成由 n 个空格符组成的字符串,它相当于这里的 strSPC()函数。

C>TYPE STR25.C

```
#include "string.h"
```

```
#include "alloc.h" /* 生成 n 个 ASCII 码 m 组成的串 */
```

```
char *strSTRN(unsigned char n,unsigned char m)
```

```

{
char buf[255];
memset(buf,0,255);
if(n>0)memset(buf,m,n);
return (buf);
}

```

```
char *strSTRA(unsigned char n, char *x) /* 生成 n 个串 x 头字符组成的串 */
```

```

{
char buf[255];
memset(buf,0,255);
memset(buf,*x,n);
return (buf);
}

```

```

char *strSPC(unsigned int n)          /* 生成 n 个空格组成的串 */
{
    return (strSTRA(n," "));          /* 或用 return (strSTRN(n,0x20)); */
}

main()
{
    char *v=(char *)malloc(256);
    char *x="STW";
    int m=101;                        /* 用调试表达式 101,c,可知它低字节为字母 e */
    v=strSTRN(4,m);
    printf("%s\n",v);                 /* 输出:eeee */
    v=strSTRA(5,x);
    printf("%s\n",v);                 /* 输出:SSSSS */
    v=strSPC(2);
    printf("=%s=\n",v);               /* 输出:== */
}

```

7.4 查找字符串实用程序 GREP.COM

7.4.1 作用

GREP.COM 是一个查找文件中指定字符串的实用程序,查找可以同时几个文件中查找。它不但可以从给定的文件中找出有指定字符串的行,也可以找出文件中不含有指定字符串的那些行。此外,通过它还可以一次性知道哪几个文件中有指定字符串及行数等。为方便叙述起见,下面总假定 GREP.COM 程序已被拷到虚拟盘 E 上。为明确起见,必要时用□表示空格符。

7.4.2 语法和帮助

为获得帮助,可用操作:

E>GREP□?

按回车键后屏幕显示:

```

Turbo GREP Version 1.1 Copyright (c) 1987, 1988 Borland International
Syntax: GREP [-rlcnvidzuwo] searchstring file[s]
/* 语法: GREP□[选择项]□被搜索串□文件名□文件名..... */
Options are one or more option characters preceded by "-", and optionally
followed by "+" (turn option on), or "-" (turn it off). The default is "+".
-r+ Regular expression search      -l- File names only
-c- match Count only               -n- Line numbers
-v- Non-matching lines only        -i- Ignore case
-d- Search subdirectories            -z- Verbose
-u- Update default options         -w Word search
-o- UNIX output format             Default set: [0-9A-Z-]

```

A regular expression is one or more occurrences of: One or more characters optionally enclosed in quotes. The following symbols are treated specially:

<code>^</code> start of line	<code>\$</code> end of line
<code>.</code> any character	<code>\</code> quote next character
<code>*</code> match zero or more	<code>+</code> match one or more
<code>[aeiou0-9]</code> match a, e, i, o, u, and 0 thru 9	
<code>[^aeiou0-9]</code> match anything but a, e, i, o, u, and 0 thru 9	

7.4.3 语法说明

一 被搜索串的两形式

选择项由 GREP 开关构成。它是任选项,即可选也可不选。语法说明中的方括号不是键入字符,它只是表示任选的意思。被搜索字符串是必写的,它有两种模式:

1. 文字串是真正的字符串,字符串用双引号括起。例如,

```
"search string with space"
```

2. 是规则表达式,它又有两种形式。

(1) 字符串本身不加双引号,字符串前后可加上上述的具有特殊意义的字符或字符集合。

例如

```
[^a-z]word[^a-z]
```

它表示字符串 word 前后不含有字符 a 到 z 等 26 个小写字母中的任一个字母。当然,其余的字符包括 26 个大写字母是允许的。

(2) 集合加上双引号,又加上前述特殊字符。例如,

```
"[,.:? '\"]" $
```

它表示查找那些行尾有字符 , . : , . ? , ' 和 " 的语句。注意这里没有包括字符 \。要找出那些含有字符 \ 的行,则上述语句应改成

```
"[,.:? '\\"]" $
```

至于文件名可以是具体的文件名,也可以是带有 DOS 通配符 (? 和 *) 的文件名,并且文件名可以带上路径名。如果没有指定路径, GREP 只在当前目录中寻找。

为用具体例子说明应用,下面以虚拟盘 E 上的三个文件 QZ1.C、QZ2.C 和 QZ3.C 为操作对象,它们的内容分别为:

```
E:>TYPE QZ1.C
main()
mymain()
mymainfunc()
MAIN(i;integer);
main(i;integer)
main(i,j;integer);
if(main())Ea(t);
A:\data.fil
C:\Data.fil
B:\DATA.FIL
d:\data.fil
```

```

a:fata.fil
writeln("c:\\data.fil");
every new word must be on a line
MY WORD!
word smallest unit of speech
in the beginning, there was the WORD, end the WORD
Each file has at least 2000 words
He misspells toward as forword
This is a search string with space in it
This is a search string with some spaces in it
He said hi to me
Where are you going?
Happening in anticipation of a unique situation,
Exxample include the falling,
"many men smoke, but for man Chu. "
He said "HI" to me
Where are you gong? I'm headed to the teach this afternoon
Anyway,thiss is the time we have.
Do you think The main reason.
He sait "Hi"to me just when I . .
Where are you going? I'll bet you're headed to. . .

```

E:>TYPE QZ2.C

```

main()
mymain()
mymainfunc()
MAIN(i;integer);
main(i;integer)
main(i,j;integer);
if(main())Ea(t);
A:\data.fil
C:\Data.fil
B:\DATA.FIL
d:\data.fil
a:fata.fil
writeln("c:\\data.fil");
every new word must be on a line
MY WORD!
word smallest unit of speech
in the beginning, there was the WORD, end the WORD
Each file has at least 2000 words
He misspells toward as forword
E:>TYPE QZ3.C
This is a search string with space in it
This is a search string with some spaces in it
He said hi to me
Where are you going?

```

Happening in anticipation of a unique situation,
Exxample include the falling,
"many men smoke, but for man Chu."
He said "HI" to me
Where are you gong? I'm headed to the teach this afternoon
Anyway, thiss is the time we have.
Do you think The main reason.
He sait "Hi" to me just when I...
Where are you going? I'll bet you're headed to...

二 规则表达式(或称“正则表达式”, regular expression)

在规则表达式中,某些字符具有特殊的意义,它们是用于查找的操作符。而在文字串中并没有任何操作符,每个字符都被当作一个文字。

规则表达式是由一个或多个表达式构成的(A regular expression is one or more occurrences of:)。每个表达式由方括号([])括起来的一个或多个字符组成(One or more characters optionally enclosed in quotes)。当选用开关 -r 或 -r+ 时,被查找的串就被看作是正则表达式(不是文字表达式),随后的字符就有了特殊的意义(The following symbols are treated specially:)。这些字符被称为“操作符”。操作符在文字串被当作一般字符处理。

—1 [SET]

它是规则表达式的一种字符集合表示方式。方括号是必须键入的字符,SET 的具体内容有两种表达方式:

(1) 单个指定字符;多个指定字符,字符与字符之间紧靠,不加任何其它字符。

(2) 采用连字符 -, 即把两个字符用连字符隔开的办法来指定多个连续的字符(ASCII 码值)。

这两种方式可连用。例如,

[aeiou0-9] match a, e, i, o, u, and 0 thru 9

表示允许匹配的字符为 a, e, i, o, u 和数字 0 到 9。

[^aeiou0-9] match anythingbut a, e, i, o, u, and 0 thru 9

由于前面加了 ^ 符号,表示只匹配除了 a, e, i, o, u 和 0 到 9 以外的字符。

注意,问号(?)、加号(+)、星号(*)、美元符号(\$)和句号(.)在集合中不再有特殊意义,它们只有原来的含义,即相当于一般字符处理。

—2 ^ start of line

只匹配行首有指定串的行,即只有这些行才被检索出来。或者说规则表达式行首有 ^ 号,则表达式就与除了表达式中字符外的任何字符匹配。如果 ^ 在规则表达式内出现,^ 只有紧跟 [后它才起作用,

—3 \$ end of line

只匹配行尾有指定串的行。

—4 . any character

点号后跟任何字符

—5 \ quote next character

反斜杠后引出紧接着的字符,即只匹配反斜杠后的字符。例如, \. 匹配一个句点(.)而不是匹配“任何字符”了。因为反斜杠和句点字符有特殊的意义,如果想查找它们,就必须在它们

前面放上反斜杠以换码字符的形式来把它们引起。

—6 * match zero or more

* 表示相当于 0 或多于 0 个字符。例如：被搜索字符串写成

f *

则它可以搜索出 f , fo , foo , fopo 等多种形式的字符串。

—7 + match one or more

表示匹配 1 个或 1 个以上字符，它和 * 的特殊意义略有不同。例如：有

E>TYPE P.C

f fo foo fooo

f

fo

foo

fooo

f

fo

foo

fooo

E>GREP fo+ p.c

File P.C:

f fo foo fooo

fo

foo

fooo

f

fo

foo

fooo

E>GREP fo* p.c

File P.C:

f fo foo fooo

f

fo

foo

fooo

f

fo

foo

fooo

实施操作后很容易发现，o+ 与 o* 的区别在于，o+ 不能找出一行上只有单独的 f 行。

—8 空格符等处理

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规则表达式的一部分，就必须用反斜杠换码字符引起，也可以把空格符放在西文双引号间的办法处理。如

[^ a-z]main" " *

或

[^ a-z]main\ * *

三 GREP 的选择项说明

选择项是以连字符（-）打头、后跟 richvidzuwo 中一个或几个单字符构成。每个单字符都被认为是一个可打开或关闭的开关：在字符后有加号（+）时开关就打开，该开关内含的操作便起作用；如后跟减号（-），开关就被关掉，相应的操作就不存在。

在字符后如不写加号或减号，就认为它相当于有加号。如 -r 和 -r+ 是一样的，即开关为开；而 -r- 为关。在任何时刻，一个开关不是开就是关，并且本次选择的开关只在本次起作用，未选任何开关 GREP 自动采用缺省值。选择项允许包括几个开关：或者它们之间用空格隔开，例如

-r -d -l

或者把它们结合起来，例如，

-rdl

两者效果一样。

开关中的字母大小写不分，例如，-r 和 -R 是一样的。

当同一开关多次出现时,则以命令行上最右边的一个为准。

下面列出各开关名称、含义和实例。

—1 ☐ 未选任何开关

例 1 中输出含有指定字符串的文件名 (或称匹配文件名) 和含有指定字符串 main 的行 (或称匹配行) 的内容。如果被查文件中无任何匹配行存在, GREP 则无任何输出。

```
例 1 E>grep ☐ "main" ☐ qz1.c
      File QZ1.C:
      main()
      mymain()
      mymainfunc()
      main(i;integer)
      main(i,j;integer);
      if(main())Ea(t);
      Do you think The main reason.
```

—2 -cmatch Count only

只输出匹配文件名和行数,但不输出具体匹配行的内容。未用本开关时,不输出行数。

```
例 2 E>grep ☐ -c ☐ "main" ☐ qz1.c
      File QZ1.C:
      7 lines match    有 7 个匹配行
```

—3 -d Search subdirectories

查找的目录是命令行上文件名中指定的目录及所指定目录下的所有子目录。如所给文件名未指定路径,则在当前目录下查找。

```
例 3 E>GREP ☐ -d ☐ "main" ☐ qz1.c
      File QZ1.C:
      main()
      mymain()
      mymainfunc()
      main(i;integer)
      main(i,j;integer);
      if(main())Ea(t);
      Do you think The main reason.
```

—4 -i Ignore case

忽略字母大小写的区别。未用本开关 (或 -i) 时,字母大小写是有区别的。注意:它不能影响集合中的大小写字母的定义。

```
例 4 E>grep ☐ -i ☐ "main" ☐ qz1.c
      File QZ1.C:
      main()
      mymain()
      mymainfunc()
      MAIN(i;integer);
```

```

main(i, integer)
main(i, j, integer);
if(main())Ea(t);
Do you think The main reason.

```

—5 —l File names only

只输出相匹配的每一个文件名,或者说,每当在某个文件中找到一个匹配的字符串后便印出该文件名,然后对其它文件进行查找。

例 5 E>grep -n "main" qz1.c
File QZ1.C:

—6 —n Line numbers

输出匹配文件名和匹配行,且匹配行前标有行号。

例 6 E>grep -n "main" qz1.c
File QZ1.C:

```

2      main()
3      mymain()
4      mymainfunc()
6      main(i, integer)
7      main(i, j, integer);
8      if(main())Ea(t);
31     Do you think The main reason.

```

—7 —o UNIX output format

UNIX 输出格式,每一匹配行前有匹配文件名。这种形式以方便地支持 UNIX 型的命令管道。

例 7 E>GREP -o "main" qz1.c

```

QZ1.C  main()
QZ1.C  mymain()
QZ1.C  mymainfunc()
QZ1.C  main(i, integer)
QZ1.C  main(i, j, integer);
QZ1.C  if(main())Ea(t);
QZ1.C  Do you think The main reason.

```

—8 —r Regular expression search

搜索串当作规则表达式查找,而不是当作文字串查找。

例 8 E>grep -r "main" qz1.c
File QZ1.C:

```

main()
mymain()
mymainfunc()
main(i, integer)
main(i, j, integer);
if(main())Ea(t);
Do you think The main reason.

```

—9 —u Update default options

用于修改缺省选择项。用户可用开关 —u 安装偏爱的选择项,例如让 GREP 每次用 —z 开关作缺省选择项,则可用

GREP ☐ —u ☐ —z

显示

Enter output name for new program file [E:\GREP.COM]:

这时可选用缺省文件名,也可另用一文件名(例如 GREP1.COM)而不破坏 GREP.COM,当你 GREP1.COM 来查找字符串时,它自动选了开关 —Z,只有当你选了 —Z— 它才不起作用。—10 —v Non-matching lines only

只输出不匹配行及文件名。不匹配行是指该行中不存在要搜索的串。

例 9 E>GREP ☐ —v ☐ "main" ☐ qz1.c

```
File QZ1.C;
MAIN(i;integer);
A:\data.fil
C:\Data.fil
B:\DATA.FIL
d:\data.fil
a:fata.fil
writeln("c:\data.fil");
every new word must be on a line
MY WORD!
word smallest unit of speech
in the beginning, there was the WORD, end the WORD
Each file has at least 2000 words
He misspells toward as forword
This is a search string with space in it
This is a search string with some spaces in it
He said hi to me
Where are you going?
Happening in anticipation of a unique situation,
Exxample include the falling;
"many men smoke, but for man Chu."
He said "HI" to me
Where are you gong? I'm headed to the teach this afternoon
Anyway,thiss is the time we have.
He sait "Hi"to me just when I...
Where are you going? I'il bet you're headed to...
```

—11 —w Word search

单词查找。

例 10 E>GGREP ☐ —W ☐ "main" ☐ qz1.c

```
File QZ1.C;
main()
```

```

main(i, integer)
main(i, j, integer);
if(main())Ea(t);
Do you think The main reason.

```

我们发现，QZ1.C 中像 mymainfunc() 这样的行未被选出来，这是什么原因？原来对 -W 开关还有一个规定，虽然对其它开关来说，如用

例 11 E>GREP -c[a-z]"main"qz1.c

GREP 会告诉你

Error: Invalid option -[

这样的错误。也即说，开关后不能用集合。但是对于开关 -W 有点例外。例如，

例 12 E>GREP -W[^a-z]"main"qz1.c

则有

```

File QZ1.C:
mymainfunc()

```

又例如，

例 13 E>GREP -w[^(. ,)]"main"qz1.c

有

```

File QZ1.C:
if(main())Ea(t);

```

仔细分析例 11~13，有下列结论：

- <1> -w 后面允许跟集合；
- <2> 使用 -W 时，指定字符串前后的字符不是集合 [A-Z0-9_]

中的字符，换句话说，-W 和 -W[A-Z0-9_] 相当。显然，集合 [A-Z0-9_] 是 GREP 的缺省值。同时选用 -WU 开关时，以后 -W 的集合的缺省值一定 [A-Z0-9_]。

现在回过来看例 11，不难发现 GREP 要求匹配行在匹配字 main 前后不是非 a-z，或者说，其前后允许匹配的字符是 a-z。对例 12 而言，要求匹配字符是 (.,) 等。

-12 -z Verbose

相当于开关 -c、-l 和 -n 的全部功能，即输出含指定字符串的文件名，找出的行前面加上行号，最后给出文件中含指定字符串的行数。

例 14 E>GREP -z"main"qz1.c

```

File QZ1.C:
2      main()
3      mymain()
4      mymainfunc()
6      main(i, integer)
7      main(i, j, integer);
8      if(main())Ea(t);
31     Do you think The main reason.

```


7 lines match

为帮助理解,下面再给出同时对 qz1.c, qz2.c 和 qz3.c 进行查找的例子。

例 15 E:>grep main *.c

File QZ1.C:

```
main()
mymain()
main(i, integer)
main(i, j, integer);
if(main())Ea(t);
```

File QZ2.C:

```
main()
mymain()
main(i, integer)
main(i, j, integer);
if(main())Ea(t);
```

例 16 E:>grep -r [^a-z]main* (*.c

File QZ1.C:

```
main()
main(i, integer)
main(i, j, integer);
if(main())Ea(t);
```

File QZ2.C:

```
main()
main(i, integer)
main(i, j, integer);
if(main())Ea(t);
```

例 17 E:>grep -ri [a-c]:\\data\\.fil *.c *.inc

File QZ1.C:

```
A:\data.fil
C:\Data.fil
B:\DATA.FIL
```

File QZ2.C:

```
A:\data.fil
C:\Data.fil
B:\DATA.FIL
```

No files matching: *.INC

例 18 E:>grep -ri [^a-z]word[^a-z]*.c

File QZ1.C:

```
every new word must be on a line
MY WORD|
word smallest unit of speech
in the beginning, there was the WORD, end the WORD
```

File QZ2.C:

```
every new word must be on a line
```

MY WORD!

word smallest unit of speech

in the beginning, there was the WORD, end the WORD

例 19 E: >grep "search string with space" *.c *.asm \work\myfile. *

File QZ1.C:

This is a search string with space in it

File QZ3.C:

This is a search string with space in it

No files matching: *.ASM

No files matching: E:\WORK\MYFILE. *

例 20 E: >GREP -rd "[.,:?\"]" \$ *.c

File \QZ1.C:

Where are you going?

Happening in anticipation of a unique situation,

"many men smoke, but for man Chu."

Anyway, this is the time we have.

Do you think The main reason.

He said "Hi" to me just when I...

File \QZ3.C:

Where are you going?

Happening in anticipation of a unique situation,

"many men smoke, but for man Chu."

Anyway, this is the time we have.

Do you think The main reason.

He said "Hi" to me just when I...

例 21 E: >GREP -ild "the" *.c

File \QZ1.C:

File \QZ2.C:

File \QZ3.C:

例 22 E: >grep -i -l -d "the" *.c

File \QZ1.C:

File \QZ2.C:

File \QZ3.C:

例 23 E: >grep -il -d "the" *.c

File \QZ1.C:

File \QZ2.C:

File \QZ3.C:

例 24 E: >grep -il -d "the" *.c

Error: No file sets specified

本例是将例 9 的 *.c 前面的空格符删掉后的结果,它告诉操作者发生操作错误。

从上可以看出, QZ?.C 并不是真正的 C 源程序文件,也即是说, GREP 也适用于其它 ASCII 码文件。我们指出的这一点是异常重要的。程序员可以利用这一点对所有头文件(.H)查询他自己设计的程序中的常量、变量、函数、结构和联合是否包含在头文件中。例如,在已知

标识符 size—t 的情况下可立即从标头文件中找出涉及它的全部函数等详细情况。

例 25 E>GREP □size—t □alloc. h

```
File ALLOC. H:
typedef unsigned size—t;
void *—Cdecl calloc(size—t nitems, size—t size);
void *—Cdecl malloc(size—t size);
void *—Cdecl realloc (void * block, size—t size);
```

对 DOS 3.2 以上版本,可使用管道功能:

E>GREP □size—t □alloc. h>MYFILE

结果将查得的结果装入文件 MYFILE 中。

四 开关的优先级次序

当一个开关在命令行上多次出现时,从左到右以最右边的一个为准。例如,

E>GREP -r -i -d -i -r— main(QZ *.C

相当于

E>GREP -d -i -r— main(QZ *.C

7.4.4 可能出现的错误信息

1. Error: File spec too long
错误: 文件名指定得太长。
2. Unable to open file:s
不能打开文件 s。
3. Search string is too long
被搜索串太长。
4. Line too long (file may be binary) — search aborted
行太长(可能是二进制文件) — 搜索失败。
5. Unexpected DOS error reading file
读文件发现意外的 DOS 错误。
6. Expected character before " * "
* 号前应有字符。
7. Expected character before " + "
+ 号前应有字符。
8. Misuse of " * " or " + "
错用 * 或 + 号。
9. No file sets specified
未指定文件。
10. * * * Stack Overflow * * *
堆栈溢出。
11. * * * Out Of Memory * * *
内存不够。
12. No files matching: File
无匹配文件 File。

- 13. Invalid option -xxx
无效选择项 -xxx。
- 14. Missing "["
缺少 [号。
- 15. Missing "]"
缺少] 号。
- 16. Missing matching "Expectedcharacter after "\"
在 \ 号后面缺少“期望字符”。
- 17. Misuse of "-" in range
在范围内缺少连字符 -。
- 18. Null [] set defined
定义了 [], 即集合为空不允许。
- 19. Invalid character in search string
搜索串中出现非法字符。
- 20. No pattern specified
模式未指定。

第八章 指针

8.1 指针的重要性

在 Pascal 语言中有两种类型的参数:传名参数 (pass by address) 和传值参数 (pass by value), 而 C 语言的函数只有【传值参数】, 即在函数调用时, 实参变量将它的值传送给形参变量。若要使用传名参数, 就必须传送地址, 这就要定义指针参数。倘若想修改函数的形参, 进而改变实参值, 程序就必须传递参数的地址, 然后通过一指向实参类型的指针来实施。C 语言并没有字符串这种数据类型。定义字符串可以用一个指向字符的指针来实现的, 即任何字符串操作也需要用到指针。通过改变指针中存放的地址, 可以处理(赋值、恢复或修改)不同信息。在程序运行过程中, 用指针可以建立一个新变量, C 允许在程序中申请一定数量的内存(以字节为单位), 并把可再存储数据的内存地址返回到指针中。利用这种动态内存分配, 程序可以存取计算机中所有可用内存。

利用指针不仅能知道保存数据的值, 还能知道数据存放在什么地方。指针是保存数据存放地址的变量, 而非数据本身。一个指针可以确定地指向内存中由段和偏移量所给出的位置, 但通过指针索引还可以存取其后续的字节。

指针是 C 语言最基本的概念, 也是 C 语言的特点, 在程序设计中将大量应用。

对初学者要正确理解和使用它, 是有一个过程的。但对 TC 而言, 它为初学者创造了一个理解它的优良环境, 即在集成环境下利用调试程序时查看调试表达式就能使程序员迅速理解指针的基本概念, 增加变量值的透明度。像理解

```
char * * ptr;
```

这样说明的指向指针的指针变量, 分别用调试表达式

```
ptr
* ptr
* * ptr
```

就能很快明白其含义(参见《集成开发环境和缺省参数设置》一章)。

8.2 变量的指针和指向变量的指针变量

假定已用语句

```
int x;
```

说明(或称定义)了变量 x 的数据类型为整型, 则程序就要为它在内存中分配存储单元, 或者说给它分配一个地址。该地址实际上是物理地址, C 语言可以使用指针变量来访问它。

从上可见, 变量 x 涉及两个概念: 一是它在内存中存放的单元地址; 一是放在该单元中的内容即变量的值。C 语言把变量的地址称为该变量的指针, 而把存放地址值的变量称为指针变量。

上述定义中包含这样几层意思:

1. 习惯上,讲到整型变量 x ,我们总是指它的值,而很少涉及它的存放地址;同样,讲到指针变量,我们也可以只是指它的值(地址),称【指针(的)值】或【指针地址】。从这个意义上看,在不发生混淆的情况下,常把指针变量 p 简单地说成指针 p 是可以理解的,尽管指针和指针变量是两个不同的概念。为简略起见,以后常将指针变量简单地说成指针。

2. C 语言用符号 $\&$ 后跟变量名表示变量的地址,如 $\&x$ 表示变量 x 的地址。

3. 指针一般是指某变量的指针。或者说,指针常常和变量相关的。因此就有“指向某变量的指针”(即指向该变量的地址)的说法。C 语言中常用星号 ($*$) 表示这种“指向”关系(星号在 C 语言中被称为间接操作符)。

一个指针也可以说明为不指向任何变量的,即 `void` 类型指针(实际此指针可指向任何目标)。当它的值为 `0000H`(或 `NULL`) 时就是这样。指针被说明后,它可以指向一个跟原先在说明中指向的一个不同类型的数据。TC 利用此方法给指针重新赋值。

C>TYPEP 1.C

```
ms(s)
char *s;          /* 采用传统方式定义函数 ms 的参数 */
{
    int k;
    for (k=0;s[k]! =0; k++);
    return k-1;
}

main()
{
    int a[2]={1,2};
    void *q=a;      /* q 指向数组 a */
    q="Hello, word\n"; /* q 重新赋值 */
    printf("%d\n",ms(q));
}
```

此程序即使在 `huge` 模式下也能取得正确值。

对于函数,还有像

```
void far *MK—FP(unsigned long seg, unsigned ofs);
```

注意:数据类型指针不能转换成函数指针,反之亦然。

4. 指针变量也是变量,不过它只存放各种变量的地址值。注意,各种变量本身的值可能是整型的,也可能是浮点数,或者是一个结构等等,但存放的是它们的地址(有多个元素的变量,如结构、数组等,取其第一个元素的地址),而地址的表示方法只有两种。例如,它的值可能是

```
1008H          /* 对近指针 (near) 存放段的偏移量 */
```

或

```
B800,1008    /* 对远指针 (far) 存放段和段的偏移量 */
```

也就是说,指针变量并不保存它所指变量的值。在表达方法上,指针也与通常所说的整数不同。下面我们还将看到指针值与整数在许多方面的不同之处。

5. 事实上我们已经看到,一个指针在源程序中只能指向同一种类型的变量。譬如指向整型的指针和指向结构的指针显然有不同的含义。以后还可以从指针的运算中发现这种规定的重要性。

6. 指针变量既然是变量,所以可以进行地址赋值和运算。例如, p 是一个指向整型变量的指针变量, d 是一个整型变量,则

```
p=&d;
```

语句表示将变量 d 的地址值赋给了指针变量 p ,或称指针变量 p 指向了整型变量 d 。而

```
p+=&d;
```

则是表示指针变量 p 的值加上整型变量 d 的地址后赋给 p 。

8.3 指针的定义

在源程序中要使用一个指针之前,首先要对它进行定义(或称说明)。可以用两种方法定义它,直接定义和间接定义。其中直接定义是基本的,间接定义是在直接定义基础上再用 `typedef` 定义的。

1. 直接定义

格式:指针所指变量的数据类型 □ * 指针名

例如:

```
int *p;           /* p 是整型变量的指针 */
char *ss;         /* ss 是字符型变量的指针 */
float *ptr;       /* ptr 是浮点型变量的指针 */
static char *p;   /* p 是指向静态字符变量的指针 */
int (*x)[8];      /* x 是指向 8 个整数的数组的指针 */
char *p[8];       /* p 是有 8 个元素的指针数组,每个元素是指向 char 型变量的一个指针 */

int (*p)();       /* p 是一个指向返回整型的函数的指针 */
int *p();         /* p 是一个函数,它的返回值是一个整型指针 */
*head.data=0;     /* head 是一个带有域 data 的结构,域 data 指向一个整数 */
head.next=NULL;   /* head 的域 next 是一个指针,所指对象没有说明 */
```

2. 用 typedef 间接定义

传统的 C 语言常用直接定义来定义指针,有时甚至需要嵌套定义,以至这种定义十分难懂,也易出错。TC 用 `typedef` 在一定程度上解决了可读性这个问题。用 `typedef` 可定义一种数据类型。定义后的类型标识符就可以像其它数据类型标识符那样应用。下面用实例说明之。

```
例 1. 直接定义 int *p1;           /* 指向整数的指针 */
      int *f2();                  /* 返回整数指针的函数 */
      间接定义: typedef int *intptr; /* 将 intptr 定义为一个指向整型的指 */
```

```

/* 针类型。它是数据类型而不是指针 */
/* p1, f2()全为 intptr 类型 */
intp trp1, f2();
/* 整型 far 指针 */
例 2. 直接定义: int far * p2;
/* near 型函数, 返回整型的 far 指针 */
int far * f3();
/* far 型函数, 返回整型的 near 指针 */
int * far f4();
间接定义: typedef int far * farptr;
/* farptr 是数据类型标识符 */
farptr p2, f3();
typedef int * intptr;
/* intptr 是数据类型标识符 */
intptr far f4();
例 3. 直接定义: int (* fp1)(int);
/* 接受整型参数, 返回整数的函数指针 */
间接定义: typedef int (* fncptr1)(int);
fncptr1 fp1;
例 4. 直接定义: int (* fp2)(int * ip);
/* 接受整型指针 (fp2), 返回整数 (int)
的函数指针 */
间接定义: typedef int * intptr;
/* 第一次定义类型 */
typedef int (* fncptr2)(intptr); /* 第二次定义类型 */
fncptr2 fp2; /* 注意上两句顺序不能颠倒 */
例 5. 直接定义: int (far * fp3)(int far * ip);
/* fp3 是指向任一函数的 far 指针, 被指函
数接受 far 型整型指针, 返回整数 */
间接定义: typedef int far * farptr;
/* 定义指向整型的 far 指针 */
typedef int (far * ffptr)(farptr);
ffptr fp3;
例 6. 直接定义: int (far * list[5])(int far * ip);
/* 含有 5 个 far 指针的指针数组, 这些指针是指向函数的 far 指针, 这些
函数以指向整数(第二个 int)的 far 指针为参数, 以整数(第一个 int)为返
回值 */
间接定义: typedef int far * farptr;
typedef int (far * ffptr)(farptr);
typedef ffptr ffplist[5]; /* 三次定义类型标识符 */
ffplist list;
例 7. 直接定义:
int (far * gopher(int(far * fp[5])(int far * ip)))(int far * ip);
/* near 函数, 它以 5 个 far 函数指针为元素的数组为参数, 返回一个 far 指针, 而这 5 个 far 函
数本身以整型 far 指针为参数, 返回整数 */
间接定义:
typedef int far * farptr;
/* 定义 farptr 为指向整型值的 far 指针类型 */
typedef int (far * ffptr)(farptr);
/* 定义 ffptr 类型。ffptr 为指向任一函数的 far 指针被指函数为 near 型, 返回
整型值, 参数为 farptr 型。 */
typedef ffptr ffplist[5];
/* 定义 ffplist 数组, 它的类型是 ffptr, 有五个元素组成, 指针的每一个元素都是 far 型指
针, 每个指针指向上述任一函数 */
ffplist list;
/* 定义 list 为 ffplist 类型数组 */

```



```
ffptr gopher(ffplist);
```

/* 定义函数 gopher。它的类型是 ffptr,ffptr 是指向某一函数的 far 指针,gopher 的参数是 ffplist,ffplist 是类型为 ffptr 的数组名,即以五个类型为 ffptr 的指针构成的数组作参数。 */

从上可以看出用 typedef 定义数据类型的一般规律。类型定义层次越多越复杂,给阅读和调用带来不便。特别是结果是函数还是指针?数据的真正类型是什么?这样一类问题很费时费力气去理解。

不过,使用 typedef 语句定义一类数据类型,特别将类型标识符定义成程序员熟悉的字符串,更方便于程序设计、调试和维护。

8.4 指针运算符 * 和 & 的相互关系

1. 符号 * 和 & 是两个与指针相关的运算符,它们的优先级是一样的,按自右往左方向结合运算。

```
C>TYPE P2. C
```

```
main(){
int a=1,b=2,c=3,d=4,j;          /* 整型变量 a,b,c,d 和 j */
int *p, *ptr;                   /* 指针 p 和 ptr */
int **pp;                       /* 指向指针的指针 pp */
char *name[]={"TC","PASCAL"};
p=&a;
b=*p;
ptr=&*p;
c=* &a;
d=d**p;
printf("a=%db=%dc=%dd=%d\n",a,b,c,d);
printf("p=%pptr=%p\n",p,ptr);
for(j=0;j<2;j++){
{pp=name+j;
printf("pp=%s\n",*pp);
}
}/* 程序输出结果为
a=1 b=1 c=1 d=4
p=FFD8 ptr=FFD8
pp=TC
pp=PACAL
(编译时可能有一警告,可不管它)。 */
```

下面均以 P2. C 中的变量为例来说明相关内容。

2. & 运算

```
p=&a;
```

可以说,这时指针 p 指向了变量 a,a 的地址赋给了 p。赋值后,就可以把 &a 等同于 p 来应用。例如返回当前视区信息的函数原型为

```
void far getviewsettings(sstruct viewporttype far *viewport);
```

其使用方法可以是

```
#include "graphics.h"      /* 结构 viewporttype 包含在 graphics.h 中 */
```

```
.....
```

```
struct viewporttype view    /* 定义结构 view */
```

```
getviewsettings(&view);    /* &view 代表指针 */
```

注意:算符 & 不能用于常量或表达式。例如

```
&(a+1);
```

```
&19;
```

是不行的,而只能用于变量或数组等。取寄存器 (register) 变量的地址也是非法的。

3. * 运算

```
b = *p;
```

b 取得 p 指向变量 a 的地址中的值,即 a 的值,故 b=1。

由此可见,*p 即是 p 所指变量 a 的值。

当一个指针 p 并未指向具体的变量时,*p 的值是不可靠的,甚至可能破坏程序的正常执行。

4. &* 运算

根据自右向左结合,语句

```
ptr = &*p;
```

相当于语句

```
ptr = &(*p);
```

*p 是变量 a 的值,或者说就是变量 a,亦即上述语句就是

```
ptr = &(a);
```

或

```
ptr = &a;
```

这就是为什么 p 和 ptr 有同一值 FFD8H 的原因。

由此可见,&*p 相当于 &a,这里 a 是指针 p 指向的变量。

5. *& 的运算

语句

```
c = *&a;
```

中,根据自右向左结合,&a 是变量 a 的地址,可用指针 p 代替,即该句等价于语句

```
c = *p;
```

容易看出,这就是变量 a 的值,所以 c=1。由此可见 *&a 与 a 等价,或者说 *& 相当于空格符,即实际未起任何作用。

6. ** 运算

程序里有两个语句中出现 ** :

```
int **pp;  
d=d**p;
```

对前一语句后面将单独讨论。后一语句根据自右向左结合,*p为a的值,故**p是“乘上a的值”。

8.5 指针值传递的单向性

指针可以作为函数的参数,TC 库函数中就有许多函数带有指针参数。如函数

```
int rewind(FILE *fp);  
int printf(const *format,...);  
void movmem(void *src, void *dest, unsigned length);  
void far getviewsettings(struct viewporttype far *viewport);
```

在调用带有指针参数的函数时应注意【指针值的单向传递性】。看下面两个比较数字大小的程序:P3. C 和 P4. C。程序希望通过比较变量a和b的大小后,*p1和*p2按从大到小输出。

```
C>TYPE P3. C  
main(){  
int a=10,b=22,c;  
int *p1,*p2,*p;  
p1=&a;p2=&b;p=&c;  
printf("---p=%pp1=%pp2=%p\n",p,p1,p2);  
printf(" *p=%d *p1=%d *p2=%d\n",*p,*p1,*p2);  
if(a<b){  
p=p1;  
p1=p2;  
p2=p;  
printf("+++p=%pp1=%pp2=%p\n",p,p1,p2);  
printf(" *p=%d *p1=%d *p2=%d\n",*p,*p1,*p2);  
}
```

```
C>TYPE P4. C  
swap(int *,int *);  
main(){  
int a=10,b=22,c;  
int *p1,*p2,*p;  
p1=&a;p2=&b;p=&c;  
printf("---p=%pp1=%pp2=%p\n",p,p1,p2);  
printf(" *p=%d *p1=%d *p2=%d\n",*p,*p1,*p2);  
if(a<b)swap(p1,p2);  
printf("+++p=%pp1=%pp2=%p\n",p,p1,p2);  
printf(" *p=%d *p1=%d *p2=%d\n",*p,*p1,*p2);  
}  
swap(int *ptr1,int *ptr2)
```

```
{
int *p;
p=ptr1;
ptr1=ptr2;
ptr2=p;
}/* 它们输出的结果是
```

程序 P3. C:

```
——p=FFE2 p1=FFDE p2=FFE0
* P=1222 * P1=10 * P2=22
+++p=FFDE p1=FFE0 p2=FFDE
* P=10 * P1=22 * P2=10
```

程序 P4. C:

```
——p=FFE2 p1=FFDE p2=FFE0
* P=1222 * P1=10 * P2=22
+++p=FFE2 p1=FFDE p2=FFE0
* P=1222 * P1=10 * P2=22 */
```

从上可见程序 P3. C 符合要求。为什么程序 P4. C 得不到正确的结果？原因出在它进行了函数调用。并不是函数调用本身有问题，而是忽略了 C 语言中实参变量（指针 p1, p2）和形参变量（ptr1, ptr2）之间的数据传递是单向的“值传递”，即实参可把值传给形参，而形参不能把值传给实参。

在集成环境下用热键 F7 进行调试操作前先输入调试表达式 p1 和 p2，在进入函数 swap() 后从监视窗口中便看到调试表达式的值为

```
. p2: Undefined symbol'p2'
p1: Undefined symbol'p1'
```

堆栈是

```
swap(DS:FFDE, DS:FFE0)
main()
```

也可通过调试表达式查询各指针的值。可知在函数 swap() 中 ptr1 与 ptr2 指针值是进行了交换，但重进入 main() 函数后，p1 与 p2 的值未交换，而且 ptr1 与 ptr2 在 swap() 调用结束后已不再存在。但若把程序 P4. C 中的 swap() 函数改成

```
swap(int *ptr1, int *ptr2)
{
intp;
p= *ptr1;
* ptr1= * ptr2;
* ptr2=p;
}
```

亦即将地址中的内容交换，则能得正确结果：

```
——p=FFE2 p1=FFDE p2=FFE0
* P=1222 * P1=10 * P2=22
+++p=FFE2 p1=FFDE p2=FFE0
* P=1222 * P1=22 * P2=10
```

内中可见 p1 与 p2 的值并未交换。这就是说，尽管没有改变指针的值（地址），但将它中间所指的内容改变了。

若 p 是一指向整型变量 a 的指针，则 (*p)++ 相当于 a++，而 *(p++) 不与 a++ 等价。

8.6 指向数组的指针

一个数组有多个元素,每个元素在内存中占用地址。当一个指针指向一个数组时,总是指向数组的起始地址(或首址)。

C>TYPE P5.C

```
main()
{
    int k,y[4]={1,2,3,4},*p,*ptr; /* 定义了整数 k,数组 y[4],指针 p,ptr */
    p=&y[0]; ptr=y; /* 两句效果一样,都是将数组 y[4] 的首址赋给指针 */
    printf("\nptr=%p\n",ptr); /* 记住开始指针值 */
    for (k=0; k<4; k++)printf("k=%d\t",y[k]);
    printf("\n");
    for (k=0; k<4; k++)printf("k=%d\t",p[k]);
    printf("\n");
    for (k=0; k<4; k++)printf("k=%d\t",ptr[k]);
    printf("\nptr=%p\n",ptr); /* 检查此时指针值,和开始一样 */
    for (k=0; k<4; k++)printf("k=%d\t",*(ptr+k)); /* 发现输出和上面一样 */
    printf("\nptr=%p\n",ptr); /* 指针值和开始一样 */
    for (k=0; k<4; k++)
    {
        printf("k=%d\t",*ptr);
        ptr+=k;
    }
    printf("\nptr=%p\n",ptr); /* 指针值和开始不一样了 */
} /* 程序输出: ptr=FFDC
k=1 k=2 k=3 k=4
k=1 k=2 k=3 k=4
k=1 k=2 k=3 k=4
ptr=FFDC
k=1 k=2 k=3 k=4
ptr=FFDC
k=1 k=2 k=3 k=4
ptr=FFE8 */
```

从程序输出结果可以看出它们有相同的输出,因此证明上述结论成立。

上述试验也可以在集成环境下在监视窗口内输入调试表达式 `p`、`*p`、`ptr` 和 `*ptr`,再用热键 F7 进行单步跟踪,便可直接从监视窗口内看到结果。

从上面可见,当一个指针指向一个数组时,有如下结论:

1. 通过指针可引用数组元素,这是由带下标的指针实现的,或者说,当指针 `p` 指向数组 `y` 时,`p[k]`与 `y[k]`等价;

2. 当 `ptr` 指向数组 `y` 时,`ptr+k` 即是数组元素 `y[k]`的地址,因为 `ptr` 指向 `y[0]`的地址;

3. 语句 `ptr+=k`;

相当于

```
ptr=ptr+k;
```

即指针可使其本身的值改变。由此可见指针加法的例子。

4. $*(ptr+k)$ 与 $ptr[k]$ 等价, 即 $*(ptr+k)$ 与 $y[k]$ 等价, 只要 ptr 也指向数组 y 。

5. 程序中应注意指针的当前值在何种情况下会有改变。

6. 应当指出, 对指针有 $ptr++$ 这样使指针下移的表达式, 而对数组 $y[]$, 像 $y++$ 这样的表达式是无意义的。数组名 y 可以代表数组的首地址, 是一个固定的值, 但它本身不是一个变量, 因此把 $y++$ 当作后一数组元素的地址是错误的。可以用 $\&y[i]$ 表示数组 y 的第 i 个元素的地址。

8. 7 指针的运算

看下面两个程序。

C>TYPE P6.C

```
/* 1: */ main()
/* 2: */ {
/* 3: */ int a[4]={1,2,3,4};
/* 4: */ int *p,*q;
/* 5: */ p=a;
/* 6: */ printf("-1=%d%p\n",*p,p); /* -1=1 FFDE */
/* 7: */ q=p;
/* 8: */ printf("-2=%d%p\n",*q,q); /* -2=1 FFDE */
/* 9: */ p++;
/* 10: */ printf("-3=%d%p\n",*p,p); /* -3=2 FFE0 */
/* 11: */ p+=1;
/* 12: */ printf("-4=%d%p\n",*p,p); /* -4=3 FFE2 */
/* 13: */ p--;
/* 14: */ printf("-5=%d%p\n",*p,p); /* -5=2 FFE0 */
/* 15: */ p-=1;
/* 16: */ printf("-6=%d%p\n",*p,p); /* -6=1 FFDE */
/* 17: */ p=q;
/* 18: */ printf("-7=%d%p\n",*p++,p); /* -7=1 FFDE */
/* 19: */ printf("-8=%p\n",p); /* -8= FFE0 */
/* 20: */ p=q;
/* 21: */ printf("-9=%d%p\n",*(p++),p); /* -9=1 FFDE */
/* 22: */ printf("-10=%p\n",p); /* -10= FFE0 */
/* 23: */ p=q;
/* 24: */ printf("-11=%d%p\n",*p--,p); /* -11=1 FFDE */
/* 25: */ printf("-12=%p\n",p); /* -12= FFDC */
/* 26: */ p=q;
/* 27: */ printf("-13=%d%p\n",*(p--),p); /* -13=1 FFDE */
/* 28: */ printf("-14=%p\n",p); /* -14= FFDC */
/* 29: */ p=q;
/* 30: */ printf("-15=%d%p\n",*++p,p); /* -15=2 FFDE */
/* 31: */ printf("-16=%p\n",p); /* -16= FFE0 */
/* 32: */ p=q;
/* 33: */ printf("-17=%d%p\n",*(++p),p); /* -17=2 FFDE */
```

```

/* 34: */ printf("-18=%d\n",p); /* -18= FFE0 */
/* 35: */ p=q;
/* 36: */ ++p;
/* 37: */ printf("-19=%d\n",*p,p); /* -19=2 FFE0 */
/* 38: */
/* 39: */ p=q+1;
/* 40: */ printf("-20=%d\n",*--p,p); /* -20=1 FFE0 */
/* 41: */ printf("-21=%d\n",p); /* -21= FFDE */
/* 42: */ p=q+1;
/* 43: */ printf("-22=%d\n",*(--p),p); /* -22=1 FFE0 */
/* 44: */ printf("-23=%d\n",p); /* -23= FFDE */
/* 45: */ p=q+1;
/* 46: */ --p;
/* 47: */ printf("-24=%d\n",*p,p); /* -24=1 FFDE */
/* 48: */ p=q;
/* 49: */ printf("-25=%d\n",(*p)++,p); /* -25=1 FFDE */
/* 50: */ printf("-26=%d\n",*p,p); /* -26=2 FFDE */
/* 51: */ p=q; /* Note: *p=2! */
/* 52: */ printf("-27=%d\n",*p,p); /* -27=2 FFDE */
/* 53: */ printf("-28=%d\n",(*p)--,p); /* -28=2 FFDE */
/* 54: */ printf("-29=%d\n",*p,p); /* -29=1 FFDE */
/* 55: */ p=q; /* Note: *p=1! */
/* 56: */ (*p)++;
/* 57: */ printf("-30=%d\n",*p,p); /* -30=2 FFDE */
/* 58: */ p=q; /* Note: *p=2! */
/* 59: */ (*p)--;
/* 60: */ printf("-31=%d\n",*p,p); /* -31=1 FFDE */
/* 61: */ p=q; /* Note: *p=1! */
/* 62: */ printf("-32=%d\n",++(*p),p); /* -32=2 FFDE */
/* 63: */ printf("-33=%d\n",*p,p); /* -33=2 FFDE */
/* 64: */ p=q; /* Note: *p=2! */
/* 65: */ printf("-34=%d\n",--(*p),p); /* -34=1 FFDE */
/* 66: */ printf("-35=%d\n",*p,p); /* -35=1 FFDE */
/* 67: */ p=q; /* Note: *p=1! */
/* 68: */ ++(*p);
/* 69: */ printf("-36=%d\n",*p,p); /* -36=2 FFDE */
/* 70: */ p=q; /* Note: *p=2! */
/* 71: */ --(*p);
/* 72: */ printf("-37=%d\n",*p,p); /* -37=1 FFDE */
/* 73: */ }

```

程序说明:

1. 程序用指针 q 保存指针 p 的初始地址。p=q 使 p 恢复原值(地址)。
2. 每个整数在内存中要占 2 个字节,一个字节占一个内存单元(8 位),一个单元占有一个地址,故一个整数要占 2 个地址单元。所以前后连续的两个整数的地址数差 2。注意:对

```
int s[100], *p=s; p+=sizeof(p);
```

指针将向前移动了两个数组元素,而不是一个!因为 $\text{sizeof}(p)=2$ 。

3. $p++$ 就是 p 加上 1($P+1$),这个 1 表示 1 个元素,即 $p+1$ 指向 p 后一个元素(p 指向 $a[0]$; $p+1$ 指向 $a[1]$),而不是 p 的值(地址)加 1。所以 $p+1$ 的地址是 p 的地址 FFDE 加 2; $\text{FFDE}+2=\text{FFE0}$ 。

4. $p+=1$ 即是 $p=p+1$,或 $p++$ 都是一样。 $p-=1$ 与 $p=p-1$ 或 $p--$ 又是一码事。 $p--$ 指向 p 前面一个元素。

5. 由于运算符 $++$ 和 $*$ 属同一优先级,所以结合方向是自右向左,故 $*p++$ 等同于 $*(p++)$ 。C 语言规定自增运算符 $++$ 在其作用变量 p 右边时,待作用变量 p 参加运算后再将其加 1,故 $*(p++)$ 相当于先运算 $*(p)$,然后再运算 $p=p+1$ 。注意, $p=p+1$ 要在 $\text{printf}()$ 执行后才实现。

6. 自增运算符 $++$ 在作用变量 p 左边时,先将 p 增 1,然后以 $p+1$ 参加运算,所以 $*(++p)$ 相当于先算 $p=p+1$,后算 $*p$ (注意:此时的 p 的值不是原值,而是新值)。7. $*p--$ 、 $*(--p)$ 的含意和 $*p++$ 、 $*(++p)$ 类同。

8. $(*p)++$ 表示先取 $*p$ 的值,运算结束后再将 $*p$ 的值加 1。这就是说,尽管 p 的值(地址)不变,但其单元内的内容已变,并且在未重新改变之前一直保持着。注意:当前的 $*p$ 值就是程序中类似

```
/* Note: *p=2 */
```

这样的注释语句的真正的含意。

9. 语句 $++(*p)$; 是将 $*p$ 的值加 1。它相当于语句 $*p+=1$; 或语句 $(*p)++$;。注意:语句 $++(*p)$; 与 $(*p)++$; 是不同的。

10. 对指向函数的指针而言,像 $p++$, $p+n$, $p-n$ 等等是无意义的。

11. 当将程序中语句 3 和 4 及有关语句按①~⑧修改后,其输出结果如下所示。注意:语句 3 和 4 前的类型修饰符(int 或 char) 始终应是相同的,即指针只能指向同类型的数据,否则会出现

Suspicious pointer conversion in function main

的警告,输出结果将变得不可靠。

```
① int a[4]={1,2,3,4};
② int *p, *q;
③ char a[4]={1,2,4};
④ char *p, *q;
⑤ char a[4]="1234";
⑥ char a[4]="ABCD";
⑦ %d
⑧ %c
```

输出	①②⑦	③④⑦	⑤④⑦	⑥④⑦	⑥④⑧
-1=	1 FFDE	1 FFE2	49 FFE2	65 FFE2	A FFE2
-2=	1 FFDE	1 FFE2	49 FFE2	65 FFE2	A FFE2
-3=	2 FFE0	2 FFE3	50 FFE3	66 FFE3	B FFE3
-4=	3 FFE2	3 FFE4	51 FFE4	67 FFE4	C FFE4
-5=	2 FFE0	2 FFE3	50 FFE3	66 FFE3	B FFE3
-6=	1 FFDE	1 FFE2	49 FFE2	65 FFE2	A FFE2

-7=	1	FFDE	1	FFE2	49	FFE2	65	FFE2	A	FFE2
-8=		FFE0		FFE3		FFE3		FFE3		FFE3
-9=	1	FFDE	1	FFE2	49	FFE2	65	FFE2	A	FFE2
-10=		FFE0		FFE3		FFE3		FFE3		FFE3
-11=	1	FFDE	1	FFE2	49	FFE2	65	FFE2	A	FFE2
-12=		FFDC		FFE1		FFE1		FFE1		FFE1
-13=	1	FFDE	1	FFE2	49	FFE2	65	FFE2	A	FFE2
-14=		FFDC		FFE1		FFE1		FFE1		FFE1
-15=	2	FFDE	2	FFE2	50	FFE2	66	FFE2	B	FFE2
-16=		FFE0		FFE3		FFE3		FFE3		FFE3
-17=	2	FFDE	2	FFE2	50	FFE2	66	FFE2	B	FFE2
-18=		FFE0		FFE3		FFE3		FFE3		FFE3
-19=	2	FFE0	2	FFE3	50	FFE3	66	FFE3	B	FFE3
-20=	1	FFE0	1	FFE3	49	FFE3	65	FFE3	A	FFE3
-21=		FFDE		FFE2		FFE2		FFE2		FFE2
-22=	1	FFE0	1	FFE3	49	FFE3	65	FFE3	A	FFE3
-23=		FFDE		FFE2		FFE2		FFE2		FFE2
-24=	1	FFDE	1	FFE2	49	FFE2	65	FFE2	A	FFE2
-25=	1	FFDE	1	FFE2	49	FFE2	65	FFE2	A	FFE2
-26=	2	FFDE	2	FFE2	50	FFE2	66	FFE2	B	FFE2
-27=	2	FFDE	2	FFE2	50	FFE2	66	FFE2	B	FFE2
-28=	2	FFDE	2	FFE2	50	FFE2	66	FFE2	B	FFE2
-29=	1	FFDE	1	FFE2	49	FFE2	65	FFE2	A	FFE2
-30=	2	FFDE	2	FFE2	50	FFE2	66	FFE2	B	FFE2
-31=	1	FFDE	1	FFE2	49	FFE2	65	FFE2	A	FFE2
-32=	2	FFDE	2	FFE2	50	FFE2	66	FFE2	B	FFE2
-33=	2	FFDE	2	FFE2	50	FFE2	66	FFE2	B	FFE2
-34=	1	FFDE	1	FFE2	49	FFE2	65	FFE2	A	FFE2
-35=	1	FFDE	1	FFE2	49	FFE2	65	FFE2	A	FFE2
-36=	2	FFDE	2	FFE2	50	FFE2	66	FFE2	B	FFE2
-37=	1	FFDE	1	FFE2	49	FFE2	65	FFE2	A	FFE2

12. 不允许对两个指针进行加、乘、除、移位或屏蔽运算,也不允许 float 或 double 数与指针相加。但在一定的条件下可以将两个指针相减。例如,下列程序求字符串 s 的字符数,执行后得数 4。

C>TYPEP 7.C

main()

{

char *s="ABCD",

printf("%d\n",strlen(s));

}

strlen(char *q)

{

char *t=q; /* 因 while 只测试表达式是否为 0,而 \0 是 0 */

while (*t != '\0') /* 所以将本句改成 while(*t) 效果也一样 */

t++;

```
return(t-q);    /* 两指针相减 */
}
```

在集成环境中,输入六个调试表达式: s,t,q,t-q,t,p 和 q,p,利用热键 F7 进行单步跟踪,在监视窗口内可看到调试表达式的值在变化。当执行 while 一次循环后监视窗口内显示:

```
q,p: DS:0194
.t,p: DS:0195
t-q: 1
q: "ABCD"
t: "BCD"
s: Undefined symbol's'
```

由此很好理解指针减法的含意。显然,使用指针减法应谨慎。

8.8 指针加减时的比例因子

从上可见,当指针 p 指向一个整型变量 (int) 时,p+1 后的新 p 值(地址)等于老的 p 值加上 2。这可以看成 1 乘上 2;当 p 指向一个字符变量 (char) 时,p+1 新值等于老的 p 值加 1,可看成 1 乘上 1。可以验证,一般 p+n 的值(地址)等于 n 乘上一个【比例因子】,该比例因子的值跟指针所指变量的类型相关。例如:

指针说明的类型	比例因子的值	p+n (假定 p 指向变量 a) 的值(地址)
char	1	(&a)+n
int	2	(&a)+n*2
long(float)	4	(&a)+n*4
double	8	(&a)+n*8

事实上,数据的类型是和硬件相关的,或者说对不同的机器同一类型的数据占的数据位是不同的,因此,常用

```
sizeof(type)
```

来表示比例因子,它返回 type 类型变量在内存中所占字节数。其中 type 表示数据类型;char、int、short、long、float 和 double 等各种数据类型。总之,p+n 所指的内存地址是 p+(n * sizeof (type))。

8.9 指针的动态分配和给指针赋初值

```
C>TYPE P8.C
#include "alloc.h"
main(){
int *q;
q=(int *)malloc(sizeof(int));
*q=421;
printf("%d%p\n",*q,q);
}
```

如果将语句

```
q = (int *) malloc(sizeof(int));
```

去掉,编译时将出现警告

Possible use of 'q' before definition

程序运行时有时可能会有正确结果,有时又会有不正确的结果显示,如显示

Null pointer assignment

这说明什么?说明使用指针编写程序时,常见的一个错误就是未给指针分配空间。或者说,违背了【使用指针的规则】,也即是说,一定要在将地址赋给指针之后再使用它。

上述程序中,语句

```
q = (int *) malloc(sizeof(int));
```

就起到这种作用。这句的含意是,先在内存中为一个 int 类型的变量分配足够的空间,然后把这段内存空间的起始地址赋给指针 q,即 q 指向了一个 int 型的变量。其中表达式 (int *) 的意思是,将起始地址看作是一个指向 int 型的指针,这称为类型强制转换。在这里,TC 并不一定要求有这个表达式,但许多其它的编译器需要它。如果将它略去,就会得到

Nonportable pointer assignment

这样的警告信息。另外,库函数 malloc(sizeof(int)) 返回一指向长度为 sizeof(int) (连续的)字节的存储块指针,如果内存没有足够的存储区,将返回 NULL 指针。

应该说,遵循指针使用规则是重要的,否则就不能保证指针一定指向一个未用的内存空间。在未给指针赋值前,指针的值(地址)完全可能是一动态值,即不确定的,很有可能是一已有其它用途的地址。要是用了它,则麻烦了。作为实例现在来看看指针与 scanf() 函数的关系。

一般用

```
char t[80]; scanf("%s", t);
```

来接收字符型数据是可以的,但你想对指针用类似的方法,即用

```
char *t; scanf("%s", t);
```

则就不可靠了,或者说有时程序接收的数据是正确的,但有时(甚至对同一程序反复执行)就不那么正确了。原因何在?主要是未给指针 t 明确指向的单元,于是它可能有一个不可预料的值(地址)。在程序执行时,这个地址很有可能被程序随后的执行部分占用,因此,你原先从键盘上输入的数据便被修改,或者输入的数据改变了原先存放指令或数据的代码,结果使整个程序执行变得很糟糕。这种情况在程序规模较大时时有发生。为避免发生这种地址冲突,上述语句可改成

```
char ts[80], *t=ts; scanf("%s", t);
```

对指针赋初值有两种,一是直接的,即在说明指针时就赋初值;另一是间接的,例如将指针指向某一变量或存储块。

1. 在对指针说明时可对其直接赋初值。例如:

```

(1)int *p=0;
(2)int a[4]={1,2,3,4}, *p=a;
    static int a[2][3]={1,2,3},{4,5,6}}, *p=a;
    static int a[2][3]={1,2,3,4,5,6}, *p=a; /* 后两句效果一样 */
(3)char *p="CD"; /* 等价于两句 char *p; p="CD"; */
    /* 而等价于 char *p; *p="CD"; */

```

C>TYPE P9.C

```

main()
{
char *q="CD";
printf("");
}

```

用 F7 热键调试程序 P9.C, 结果有

调试表达式	值
* &q[2]	'\x0' /* 字符串最后一个字母后自动加上 '\0' */
* &q[1]	'D'
* &q[0]	'C'
&q[2],p	DS:0196
&q[1],p	DS:0195
&q[0],p	DS:0194
* q	'C'
q,p	DS:0194

C>TYPE P10.C

```

main(){
char s[]="AB"; /* 该句不等价于 char s[]; s[]="AB"; */
char *ptr="%s\n"; /* 用指针定义格式字符串 */
char t[]="%s\n"; /* 用数组定义格式字符串 */
printf("%s\n",s); /* 也不等价于 char s[]; s="ab"; */
printf(ptr,s); /* 本句和上句效果相同,即指针代替格式字符串 */
printf(t,s); /* 用数组代替格式字符串 */
}

```

程序输出字母 AB。虽然在输入时可以像指针那样,只用数组名 s 便可整体输出,但这实际上跟库函数 printf() 有关。也即是说,不能因此而得出在程序中可像指针那样用

s+n

来取数组中的元素。这是因为,尽管数组名可以代表地址,但数组名本身不是一个变量。

```

(4) char *q="abc", *p=q; /* 注意:本句应理解为:(1)指针 q 指向字符串 "abc" 的首
    址;(2)指针 p 指向指针 q 当前指的地址 */
(5) char *p[]={"2"}; /* 指针数组赋初值,它接收的是变量的地址 */
(6) char *p[]={"a","b"};
(7) int a,b,c, *p[3]={&a,&b,&c};
(8) char *const str="ABCD"; /* 一个常量指针。注意:它定义后不能在以后的程序中
    再赋值!如 str="EF";但可以在以后的程序中进

```

行函数调用,如 strcpy(str, "EF"); 是合法的,因为它将字符串 "EF" 一个字符一个字符拷贝到 str 所指的存储空间。 */

(9) char const * str1="ABC"; /* 指向常量字符串的指针 str1。它是变量,因而可以在以后的程序中再赋值。 */

(10) const * p; /* const 相当于 const int */

2. 给指针间接赋值。例如,

(1) p=&a;

把变量 a 的地址赋给指针 p。

(2) p=NULL; /* 符号常量 NULL 一般被定义为常量 0,它在 stdio.h 中有定义 */

表示 p 不指向任何变量。注意:它与 p 没赋任何值是不同的,未对 p 赋值不等于没有值,只是它的值是一个不确定的值,也就是说,它可能指向一个事先未估计到的单元,如果把这个单元的内容取出来处理,很可能出问题!因而在引用指针之前一定要赋初值。

注意:存储类型的相互作用。例如,在某函数内部定义

```
static int a[2]={0};  
int * ptr=&a[1];
```

是正确的,但定义

```
int a[2]={0};  
static int * ptr=&a[1];
```

则是错误的。因为当指针定为 static 时,ptr 将自动被初始化为 NULL,所以后一句是非法初始化。

(3) p=&array[i];

把数组第 i 个元素 array[i] 的地址赋给指针 p。

(4) p=array; 或 p=&array[0];

若 array 为数组名,则把数组 array 的首址赋给指针 p。

(5) p=q;

把指针 q 的值赋给(说明为)同类型的指针 p。现在看程序 P11.C,它是测定数组的最大下标值。

C>TYPE P11.C

```
main()  
{  
    int a[4]={1,2,3,4};  
    int *s=a; /* s 指向整型数组 a */  
    a[4]=0; /* 如去了这一句,则不能得到预想的结果 */  
    printf("%d\n",strlen(s)); /* 输出数组最大下标值 4 */  
}  
strlen(int *q)  
{  
    int *t=q;
```

```

while( *t != '\0')
t++;
return(t-q);    /* 指针减法 */
}

```

用 F7 热键调试(进入 strlen 函数后)的部分结果有

调试表达式	一次	二次	三次	四次	五次
*q	1	1	1	1	1
*t	1	2	3	4	0
t-q	0	1	2	3	4
q	DS:FFDE	DS:FFDE	DS:FFDE	DS:FFDE	DS:FFDE
t	DS:FFDE	DS:FFE0	DS:FFE2	DS:FFE4	DS:FFE6

从上可见,地址值按 2 递增,而 t-q 并不按此递增。如果你输入调试表达式

a[0],12

你就可发现,如不加语句 a[4]=0; 则 a[4],a[5],... 等值均可能不是 0 值,那么 strlen 中的循环将继续下去,直至 a 的某个元素为 0 值时才结束。由此可见指针与数组的关系,以及 C 语言对它们的处理方法:不因为定义了维数而加于限制。所以它可能用了别的不可预料的单元中的值,使程序执行出错!

注意:程序中两语句

```
int *s=a; a[4]=0;
```

不能颠倒成

```
a[4]=0; int *s=a;
```

那样会出现

```

Expression syntax in function main
Undefined symbol 's' in function main

```

的错误。这就是说,数据类型定义语句一定要在函数开头即在其它语句之前出现。

(6)注意:虽然不同类型的指针间赋值可以通过强制类型转换实现,但是,对有些转换应十分谨慎!。

```

C>TYPE P12.C
main()
{
int k=100, *pint=&k;
float j=2.34, *pfloat=&j;
pfloat=(float *)pint;
printf("%f\n", *pfloat);
}

```

- 此程序可无错编译运行,但结果却是 -71.50076,而不是 100。读者可自行用调试表达式分析其原因。

(7) 设 p 是指向指针的指针,它的初始化见下面程序:

C>TYPE P13.C

```
main()
{
static int a=10,b=28;
int * num[2]={&a,&b};
int ** p;
p=num;
printf("First: %p\n",p++);
printf("Second: %p\n",p);
}
```

C>TYPE P14.C

```
main()
{
static int a[2]={10,28};
int * num[2]={&a[0],&a[1]};
int ** p;
p=num;
printf("First: %p\n",p++);
printf("Second: %p\n",p);
}
```

以上两个程序只有两句不同,一个用单变量,一个用数组,程序最后输出结果是一样的,即是

First : FFE2

Second : FFE4

但是当用热键 F7 单步调试时有些差异:

调试表达式	单 变 量			数 组		
	一次	二次	三次	一次	二次	三次
**p	3034	10	28	3034	10	28
*p	3838	&a	&b	3838	aa+1	p
DS:001E	DS:FFE2	DS:FFE4	DS:001E	DS:FFE2	DS:FFE4	* num
&a	&a	&a	a	a	a	num
{&a,&b}	{&a,&b}	{&a,&b}	{a,a+1}	{a,a+1}	{a,a+1}	

注意:如将语句

static int a=10,b=28; 或 static int a[2]={10,28};

中的 static 修饰符去了,则会出现

Illegal initialization in function main

即非法初始化的错误。从上还可看到指针未赋初值前已有值,该值非程序员预料的。

如果我们把上述程序中前三句改成

```
static char a='A',b='B';
char * num[2]={&a,&b};
char ** p;
```

或

```
static char a[2]="AB";
char * num[2]={&a[0],&a[1]};
char ** p;
```

则对它们调试表达式的值都为

调试表达式	单变量或数组		
	一次	二次	三次
* * p	'k'	'A'	'B'
* p	"k\v"	"AB..."	"B..."
p	DS:001E	DS:FFE2	DS:FFE4
* num	"AB..."	"AB..."	"AB..."
num	{ "AB...", "B..." } { "AB...", "B..." } { "AB...", "B..." }		

这里三个小数点... 表示还有一些其它内容。内中 p 的值和原来完全相同。

(8) 设 ptr 是指向函数的指针,它的初始化可参见下面程序,程序运行结果为

```
a=30 b=2 c=30
```

```
a=30 b=2 c=2.
```

```
C>TYPE P15.C
```

```
main()
{
int max(int,int),min(); /* 两个函数原型,min 不太完全,但无妨 */
int (* ptr)(); /* ptr 是指向任一函数的指针 */
int a=30,b=2,c;
ptr=max; /* ptr 赋初值,max 是被指向的函数名,ptr 取得函数入口地址 */
c=(* ptr)(a,b); /* 通过指针 ptr 调用函数 max */
printf("a=%db=%dc=%d\n",a,b,c);
ptr=min; /* ptr 又指向另一函数,取得其入口地址 */
c=(* ptr)(a,b); /* 通过 ptr 调用 min 函数 */
printf("a=%db=%dc=%d\n",a,b,c);
}

max(int x, int y)
{
int z;
return (x>y)?x:y; /* 返回大值 */
}

min(int x1,int y1)
{
int z;
return (x1>y1)?y1:x1; /* 返回小值 */
}
```

利用 F7 热键和设置的调试表达式,可以很容易看到调用函数的入口地址。

调试表达式	一次值	二次值	三次值
ptr	main	max	min
ptr,p	CS:01FAmain	CS:024Amax	DS:0261min

* ptr main max min

(9) 指向函数的指针作函数的参数时的赋值方法

指向函数的指针 ptr 作函数 f1 的形式参数,当其它函数调用 f1 时,应将 f1 对应的实参用 ptr 所指的函数名 f2 代替,也即将 f2 的地址传给 f1 的形参。f2 可以是任一函数,不过它的形参应与 ptr 相适应。

```
C>TYPE P16.C
main()
{
    int prn(),sr(),cub();
    int a=3,b=10,c=2;    /* prn 的参数 sr 是一个函数名,函数名 sr 作实参 */
    printf("a=%d b=%d c=%d a*a*b=%d\n",a,b,c,prn(a,b,c,sr));
    printf("a=%d b=%d c=%d a*b*c=%d\n",a,b,c,prn(a,b,c,cub));
}
prn(int x,int y,int t,int (*fun)())    /* fun 是一指向函数的指针 */
{
    return (*fun)(x,t)*y;    /* fun 根据实参 sr 调用函数,x,t 作实参 */
}
/* 调用函数不是固定的,从而增加了灵活性 */
sr(int x) /* 因 C 语言对被调用函数参数个数不作检查,所以少一个参数也无妨 */
{
    /* 相反,如此句写成 sr(int x,int t) 会产生一个参数未被使用的警告 */
    return(x*x);
}
cub(int x1,int t1)    /* 或 cub(int x,int t) */
{
    return x1*t1;    /* 或 return x*t; */
}
```

本例输出为

```
a=3 b=10 c=2 a*a*b=90
a=3 b=10 c=2 a*b*c=60
```

8.10 指针比较大小

1. 若两个指针指向同一个数组元素,则可以进行比较。指向前面的元素的指针小于指向后面的指针。如

```
C>TYPE P17.C
/* #define NULL '\0' */
main()
{
    int a[4]={1,2,3,4};
    /* int *p0=NULL */
    int *p1=&a[2], *p2=&a[3];
    a[4]=0;
```

```

printf("p1=%p p2=%p\n",p1,p2);    /* 输出 p1=FFE2 p2=FFE4 */
if(p1>p2)printf("==p1>p2==");
/* else printf("==p1>NULL=="); */
else printf("--p1<=p2--"); /* 输出 --p1<=p2-- */
}

```

p1<p2 为真 (值为 1), p1>p2 为假 (值为 0)。若用调试表达式, 则有

调试表达式	值
p1<=p2	1
p1>p2	0
&a[3]	{4,0,285,1}
&a[2]	{3,4,0,285}
a,p	{1,2,3,4}
*p2	4
*p1	3
p2,p	DS:FFE4
p1,p	DS:FFE2

2. 此处若两个指针不是指向同一个数组, 则比较就没有意义。

3. 任何指针同 NULL 作相等或不等比较都是有意义的。如在上述程序中将注释语句添入, 则程序输出

```
==p1>NULL==
```

注意: 不能输入像

```
p1>NULL
```

这样的调试表达式, 因为 NULL 被定为宏, 而宏是不能作调试表达式的。当然, 你可以输入像

```
p1>p0
```

这样的表达式, 也能起到同样的作用。

8.11 指针与字符串

TC 并没有将字符串定义成独立的数据类型, 而将它定义成字符数组或者一个指向字符的指针。

```
char *strarr="ABC";
```

使用指向字符类型的指针。它在内存中并没有为存放字符串而预留空间, 而只在内存中留出几字节存放 strarr 指针变量的值 (参见《数组与字符串》一章)。

```

C>TYPE P18.C
#include "string.h"
#define ITEM--NUM 5    /* 元素个数 */
#define PS1(X,Y) printf("\n" #X"="); \
    for(n=0;n<ITEM--NUM;n++)printf(" %s," ,Y)

```

```

#define PS2 PS1(str, str[n]); PS1(s, s[n])
#define PS1P(X, Y) printf("\n 地址" #X "=", Y); \
    for(n=0; n<ITEM-NUM; n++) printf("%p", Y)
#define PS2P PS1P(str, str[n]); PS1P(s, s[n])
char s[][20]={"fde", "fd", "a", "asd", "fd"}; /* 原始字符数组 */
char *str[ITEM-NUM]; /* 指针数组 */
main() /* 本程序用于说明指针数组 */
{
    int j;
    int n=ITEM-NUM;
    PS2, PS2P;
    for(j=0; j<ITEM-NUM; j++)
        str[j]=s[j]; /* 指针获得数组地址 */
    qs(str, 0, n-1); /* 对字符数组排序 */
    PS2;
    PS2P;
    for(j=0; j<ITEM-NUM; j++) /* 想用 s 数组输出排序值 */
        strcpy(s[j], str[j]); /* 这是错误的操作 */
    PS2;
}

qs(char *literal[], int left, int right) /* 参见《搜索与排序函数》一章 */
{
    /* 字母按字典顺序输出 */

    int i, j;
    char *x, *y;
    i=left; j=right;
    x=literal[(left+right)/2];
    do {
        while(strcmp(literal[i], x)<0 && i<right) i++;
        while(strcmp(literal[j], x)>0 && j>left) j--;
        if(i<=j) {
            y=literal[i];
            literal[i]=literal[j];
            literal[j]=y;
            i++; j--;
        }
    } while(i<=j);
    if(left<j) qs(literal, left, j);
    if(i<right) qs(literal, i, right);
} /* 程序输出: str=(null), (null), (null), (null), (null),
    s=fde, fd, a, asd, fd,
    地址 str=0000, 0000, 0000, 0000, 0000,
    地址 s=0194, 01A8, 01BC, 01D0, 01E4,
    str=a, asd, fd, fd, fde,

```

```

a=fdc-fd-f,asd,fd,
地址 s<- 01BC,01D0,01E4,01A8,0194,
地址 s<- 0194,01A8,01BC,01D0,01E4,
str=fd,asd,a,asd,a,
s=a,asd,fd,asd,a,

```

从本例还可看出,如把指针作全局变量,一般应初始化(赋值或指向某变量),否则也许是危险的。应尽量将它放在函数内部,即作局部变量处理,然后用动态地址分配函数给它分配空间较好。 * /

8.12 和存储模式相关的指针修饰符

1. 指针修饰符

与存储模式相关的指针修饰符最基本的有三个: near、far 和 huge。而指针修饰符 —cs、—ds、—es 和 —ss 是特定的 near 型数据指针修饰符。确切地讲,这七个修饰符是地址修饰符,它们也可对函数修饰。

(1) near 指针

定义指针为近指针。其一般形式是

<数据类型>□near□<指针定义>;

例如:

```

char near *s;
int (near *ip)[20];

```

被定义的指针指向 64KB 范围内的一个字。当程序编译成中 (medium)、大 (large) 和巨 (huge) 型模式时为形成 near 指针而常用它。

near 指针地址是 16 位,它的计算依赖于某一寄存器。例如,函数指针的实际地址是将函数的 16 位值(地址)加上 CS 段寄存器左移 4 位后的值;同样,near 数据指针也是相对于数据段 DS 寄存器的偏移量。对 near 指针的任何算术运算均在同一段内进行,因此运算不涉及不同段的问题。

在集成环境下,利用缺省模式 (Options/Compiler/Model Small) 执行程序 P19.C。

```

C>TYPE P19.C
main()
{
int a0[2]={1,2},a1[2]={3,4},a2[2]={5,6},a3[2]={7,8};
int *s0=a0;          /* 缺省指针类型为 near */
int near *s1=a1;      /* 近指针类型标识为 near */
int far *s2=a2;       /* 远指针类型标识为 far */
int huge *s3=a3;      /* huge 指针类型标识为 huge */
printf("s0=%p %d %d\n",s0,s0[0],s0[1]); /* 让热键 F7 单步调试到这句 */

```

```

printf("s1=%p %d %c\n",s1,s1[0],s1[1]); /* 后看调试表达式的值 */
printf("s2=%p %d %d\n",s2,s2[0],s2[1]);
printf("s3=%p %d %d\n",s3,s3[0],s3[1]);
/* 程序输出:s1=FFD2 3 4
s2= 6ECC:FFD6 5 6
s3= 6ECC:FFDA 7 8 */

```

调试前向监视窗口内输入如下调试表达式,有

调试表达式	值
s3	6ECC:FFDA
s2	6ECC:FFD6
s1	DS:FFD2 /* 近指针地址相对于数据段的偏移量 */
s0	DS:FFCE
printf,p	CS:0B7E printf /* 函数 printf 地址相对代码段的偏移量 */
--DI	65490
--SI	65486
--BP	6510
--SP	6548
--ES	28362
--SS	28362
--DS	28362
--CS	28038
--DH * 256 + --DL, D	20041
--DL, D	73
--DH, D	78
--DX, D	20041
--CL, D	0
--CH, D	0
--CX, D	0
--BH * 256 + --BL, D	-38
--BL, D	218
--BH, D	255
--BX, D	65498
--AH * 256 + --AL, D	416
--AL, D	160
--AH, D	1
--AX, D	416

--cs、--ds、--es 和 --ss 是四个特定的 near 型数据指针修饰符,它们均为 16 位,分别对应于各段寄存器。例如,有指针 myptr 说明为:

```
char --ss * myptr;
```

则 myptr 为相对于栈段的 16 偏移量。假定用 F7 热键单步跟踪前输入调试表达式,便有调试表

```
myptr,$S:001E
```

的结果。注意：不要将 `—cs`、`—ds`、`—es` 和 `—ss` 与伪变量 `—CS`、`—DS`、`—ES` 和 `—SS` 混为一谈，它们均是各自独立的关键字。

(2) far 指针

`far` 修饰符使用于指针时，有

`<数据类型>□far□<指针定义>`；

例如：

```
char far *s;
void *far *p;
```

特别是标头文件 `graphics.h` 中定义的库函数中原型指针大多是 `far` 型的。这种修饰符常用于微型 (tiny)、小型 (small) 或紧凑型 (compact) 模式中形成远指针。当它用于函数说明时，被调用函数是远调用和远返回。

`far` 指针占 32 位，包括段地址和偏移量。段地址左移后和偏移量相加形成实际地址。使用 `far` 指针可以访问多个代码段，允许程序大于 64KB。使用 `far` 数据指针也可存取多于 64KB 的数据。对大型 (Large) 模式，指针缺省为 `far`。使用 `far` 指针可以在 Intel8088/8086 处理器中的 1MB 的存储空间中引用数据。

注意：由于对数据比较运算仅使用偏移量，而不考虑其段地址，因此会带来一些问题。

由于内存中同一地址可能有多个不同数对表示，例如有三个变量 `a`、`b`、`c` 装有三个不同的 `far` 指针：0000:0120、0010:0020、0012:0000，尽管它们对应着同一实际地址 00120H，但是由于比较指针时只是偏移量（作为 unsigned 类型）参加比较，而不是 20 位实际地址进行比较，因此表达式

`(a==b)`

将不认为是真值。

由于相等检查只用偏移量，因此任何偏移量为 0000 的指针均与 NULL 指针 (0000:0000) 相等，这显然是用户不期望的，因而必须避免。

另外还须注意的是，当 `far` 指针加上一个数后偏移量超出了 FFFFH (最大值)，则指针将回到段的开始位置。

```
C>TYPE P20.C
#include "stdio.h"
main(){
char far *f1=(char far *)0x5031ffff;
/* (或改成 char far *f1=(char far *)0x50310000;) */
f1+=1;
/* (或改成 f1-=1;) */
printf("%Fp\n",f1); /* 结果显示 5031:0000 */
/* (或显示 5031:fff) */

getch();
}
```

在 P19.C 中，偏移量超出最大值时偏移量回到 0000；当偏移量减到不能再减时变成最大值 ffffH。

所以倘若要用指针比较,最安全的方法是使用 near 指针或 huge 指针。

(3) huge 指针

huge 作指针修饰符时,有

<数据类型>□huge□<指针定义>;

huge 指针也是 32 位的,同样包括段地址和偏移量。与 far 指针不同的是,huge 指针总是规格化的。

所谓【规格化指针】是一个 32 位指针,其段地址部分为最大可能值。每隔 16 个字节就可有一地址作为段开始地址,这表明指针的偏移量在 0~15 (十六进制的 0~F)。

指针的规格化只需要将其转换为 20 位地址,然后取其左边的 16 位为段地址,而将剩下的右边 4 位作为偏移量。例如指针 2F84:0532 的绝对地址是 2FD72H(20 位),左 16 位为 2FD7H 便是段地址,后 4 位即 2H 为偏移量。因此指针规格化后为 2FD7:0002。

huge 将指针规格化的目的是,这样的指针对内存而言是唯一的特大指针。由于规格化的缘故,huge 指针偏移量部分每隔 16 个值将重复一次,而段地址也将跟着作相应的修正。例如,将 811B:000F(20 位为 811BFH)加上 1,结果为 811C:0000;将 811B:000F 减 1,为 811B:000E。huge 指针的这一特性可以使它连续处理大于 64KB 的数据结构。

huge 指针与 far 指针比较:

<1> 关于运算符(==、!=、<、>、<=、>=)都能正确用于 huge 指针,而不能用于 far 指针;

<2> 作用于 huge 指针的算术运算符对段地址和偏移量均起作用,而对 far 指针,只对偏移量起作用;

<3> huge 指针可以通过不断加 1 而寻址 1MB 地址空间,而 far 指针只在 64KB 段头折叠;

<4> 使用 huge 指针需要额外的时间,因为算术运算后需要调用指针规格化子程序,所以它比 near 指针和 far 指针处理要慢。

2. 指针修饰符与存储模式关系

(1) 不同模式下的缺省指针类型修饰符

表 8-1

存储模式	对函数指针	对数据指针
微型模式(Tiny)	near,—cs	near,—ds
小模式(Small)	near,—cs	near,—ds
中模式(Medium)	far	near,—ds
紧凑模式(Compact)	near,—cs	far
大模式(Large)	far	far
巨型模式(Huge)	far	far

下面以同一程序 P21.C 在不同模式下编译调试的调试表达式的值说明之(注意:对每个模式最好在冷启动下进行,不要在集成环境里用改变模式的方式接连使用 F7 热键连续调试;否则很可能得不到下面类似的形式)。

```
C>TYPE P21.C
```

```
ms(char *s)
```

```
{
```

```

int k;
for (k=0;s[k]!=0,k++);
return k-1;
}
main()
{
char *q="Hello, word\n";
printf("%d\n",ms(q));
} /* 程序调试结果参见表 8-2 */

```

表 8-2

存储模式	调 试 表 达 式			
	s,p	main.q,p	main.p	ms.p
Tiny	DS:15DC	DS:15DC	CS:02F1main	CS:02D8ms
Small	DS:0194	DS:0194	CS:0213main	CS:01F8ms
Medium	DS:0194	DS:0194	6DB7(CS):0021main	6DB7(CS):0008ms
Compact	6F30(DS):0094	6F30(DS):0094	CS:01D6main	CS:01BCms
Large	6F39(DS):0094	6F39(DS):0094	6DC1(CS):0006main	6DC1(CS):0006ms
Huge	6F69(DS):000C	6F69(DS):000C	6DC1(CS):002Dmain	6DC1(CS):000Cms

P20. C 在不同存储模式下 (Tiny ~ Huge) 生成的执行文件 (.EXE) 字节数依次为: 9649、9583、9807、11384、11608 和 12728。

(2) 将函数或数据指针说明为非缺省类型的危险性

一般说来,当用 near、far 或 huge 等修饰符说明函数或数据后,则被说明的函数或数据便不管存储模式,而以此为准。现在以程序 P22. C 为例来说明这一问题。P22. C 在小模式 (Small) 下编译后执行完全没有问题,能得到正确结果数 11。但在巨型模式 (huge) 下就不能得到正确结果(结果可能是 33)。

```

C>TYPE P22. C
myputs(s) /* 利用传统的 C 语言风格定义的函数, far 型 */
char *s; /* 在 huge 模式下 s 有缺省的 far 类型 */
{
int k;
for (k=0;s[k]!=0;k++) /* F7 热键调试到此行 */
return k-1;
}
main()
{
char near *mystr; /* 不管在什么模式下, mystr 均为 near 类型 */
mystr="Hello, word\n"; /* 字符串所在段可能和 mystr 不是同一个段 */
/* 因而产生警告: Suspicious pointer conversion in function main */
printf("%d\n",myputs(mystr));
}

```

原因何在? 下面我们利用 F7 热键单步调试并输入调试表达式,有

调试表达式	值
s,p	6F5E:000C
s	" — Copyright (c) 1988 Borland Intl. "
main. mystr ,p	DS:000C
main. mystr	"Hello, word\n"
main. q ,p	6F6B(DS):000C
main. q	"Hello, word\n"
main. p	6DC0(CS):002D main
myputs,p	6DC0(CS):000C myputs

从上可见指针 s 所在的段是 0x6F5E,而指针 mystr 所在的段是数据段 0x6F6B,两者并不相等。因而指针所指变量的内容也就完全不同了!

现在我们将语句

```
char * s;
```

改成

```
char near * s; /* 注意:不能写成 near char * s; */
```

仍用原法调试就有结果

调试表达式	值
s,p	DS:000C
s	"Hello, word\n"
main. mystr ,p	DS:000C
main. mystr	"Hello, word\n"
main. q ,p	6F6E(DS):000C
main. q	"Hello, word\n"
main. p	6DC3(CS):002C main
myputs,p	6DC3(CS):000C myputs

现在我们看到两者的段地址一样了,因而结果正确。

如果把程序前两句改成

```
myputs(char * s)
```

即采用现代 C 语言风格定义函数参数,或者说使用了函数原型(注意:一般在此语句之前还应加一函数原型说明语句,其形式与此句稍有不同之处是在该句后有一分号,但由于此函数在主函数之前说明,故可以省去函数原型说明语句),而用上述方法再调试,有

调试表达式	值
s,p	6F6F(DS):000C
s	"Hello, word\n"
main. mystr ,p	DS:000C
main. mystr	"Hello, word\n"
main. q ,p	6F6F(DS):000C
main. q	"Hello, word\n"
main. p	6DC4(CS):002D main
myputs,p	6DC4(CS):000C myputs

输出结果表明是正确的。

从上述三种方法我们看到,指针传递时问题出在段值上,偏移量没有问题。使用函数原型时编译程序知道指针 *s* 是 *far* 型,所以将段值和偏移量一起压入堆栈(因而显示 6F66F(DS);000C,而不是像第二种方法显示 DS:000C,也不同于第一种方法显示 6F5E:000C。第一种方法中尽管 *s* 也是 *far* 型,但其压入堆栈的段值不对)。由此可知,采用现代风格说明 TC 语句有优点。然而,在相反的情况下,即将 *mystr* 说明成 *far* 型,对这种情况也是最好用函数原型的方法可以避免出错。

使用函数原型还可以解决连接两个事先用不同存储模式编译的模块的连接问题。因为在一个小模式模块中函数希望传递和接受 *near* 指针,而在大模式中则希望接受 *far* 指针。如果不用函数原型,则可能出问题。

现在我们来解决使语句

```
mystr="Hello,word\n";
```

在编译时不产生警告的问题,只要将它改成

```
mystr=(char near *)"Hello, word\n"; /* 不能写成 (char near),即将星号丢了 */
```

就可以了。换句话说,进行了强制类型转换。还可以发现,这样一改,即使采用传统的 C 语言风格,也能得到正确的结果。

还有一种有趣的方法是将指针 *mystr* 说明为 *void* 型:

```
void *mystr="Hello, word\n";
```

即使在传统的 C 语言说明下,在 *huge* 模式下编译既不产生警告,也能得到正确的结果。利用调试表达式检查,发现它在大模式下自动将指针处理成 *far*。

对混合模式的编译连接应小心,搞得不好,会出现难于查找的错误。因此,较好的办法是用模块的编译连接。但由此出现的问题是,例如包含的标头文件 (.H) 中定义的函数或指针类型可能不合要求。譬如,你用 *huge* 模式,则所有的函数与指针应是 *far* 型。为此,你可以将有关的标头文件拷贝,对备份中的函数与指针均修改成 *far* 型。例如对 *stdio.h* 标头文件中的函数原型

```
int _Cdecl printf(const char *format,...);
```

修改为

```
int far _Cdecl printf(const char far *format,...);
```

然后将备份包含文件包含到源程序里去。

8.13 与远地址相关的指针函数

```
—1 void far *MK—FP(unsigned seg, unsigned ofs);
```

设置远指针的宏,变量 *seg* 是段地址,变量 *ofs* 是段内偏移量。它通过段值和偏移量构成一个 *far* 指针。

```
#define MK—FP(seg,ofs) (((void far *)(((unsigned long)(seg)<<16)|  
(unsigned)(ofs))))
```

由此可见它被转换为实际 20 位地址。

—2 unsigned FP—SEG(void far * farptr);

获取远地址段值。它返回一无符号整数代表远指针段值。它在 dos.h 被定义为宏：

```
#define FP—SEG(fp) ((unsigned)((unsigned long)(fp)>>16))
```

—3 unsigned FP—OFF(void far * farptr);

获取远地址偏移量。它返回一无符号整数代表远指针偏移量。它也是宏：

```
#define FP—OFF(fp)((unsigned)(fp))
```

C>TYPE P23.C

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

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

```
main()
```

```
{
```

```
char far * ptr;
```

```
unsigned seg,ofs;
```

```
ptr=MK—FP(0xB800,0x0178);
```

```
seg=FP—SEG(ptr);
```

```
ofs=FP—OFF(ptr);
```

```
printf("far ptr %Fp, segment %04x, offset %0x\n",ptr,seg,ofs);
```

```
/* 程序输出:far ptr B800:0178, segment b800, offset 178 */
```

表明段是 B800H,偏移量为 0178H。

C>TYPE P24.C

```
#include "stdio.h"
```

```
#include "dos.h"
```

```
#define PRINT(X) segment=FP—SEG(X);\
```

```
offset=FP—OFF(X);\ /* 如 p1,p2 定义成 void 也是可 */
```

```
if(*X==7)V—seg=0xb000;\ /* 以的,但是 *X==7 是不允许 */
```

```
else V—seg=0xb800;\
```

```
printf("Far Pointer=%Fp,segment=%04x,offset=%04x,\
```

```
V—segment=%p\n",X,segment,offset,V—seg)
```

```
/* 注意:上行行首的空格在打印时起作用 */
```

```
main()
```

```
{
```

```
char far * p1,far * p2; /* 注意:此句中 p2 前面的 far 不能少 */
```

```
unsigned V—seg,segment,offset;
```

```
p1=(char far *)0x00400049; /* 注意:此句中如少写 far 则输出结果便不一样 */
```

```
PRINT(p1);
```

```
p2=MK—FP(0x0040,0x0049); /* 0040:0049 存放显示方式 */
```

```
PRINT(p2);
```

```
}
```

```
/* 在 MS-DOS 5.0 下,两者输出结果是一样的:
```

```
Far Pointer=0040:0049,segment=0040,offset=0049, V—segment=B800
```

```
Far Pointer=0040:0049,segment=0040,offset=0049, V—segment=B800
```

```
*/
```

/* 利用 Turbo C 提供的 CPP.COM 工具处理

C>CPP P24

后得 P24. I, 将 P24. I 读入 Turbo C 编辑器编辑, 对某几行会显示:

Line too long — CR inserted. Press<ESC>

这是该行太长, 你可先用光标键→将光标右移一些位置, 然后按回车键便可完成对长行的编辑。以下是 P24. I 的部分内容。

```
P24. C 8:
P24. C 9: main()
P24. C 10: {
P24. C 11: char far * p1, far * p2;
P24. C 12: unsigned V—seg, segment, offset;
P24. C 13: p1 = ((void far *)
    (((unsigned long)(0x0040) << 16) | (unsigned)(0x0049)));
P24. C 14: segment = ((unsigned)((unsigned long)(p1) >> 16));
    offset = ((unsigned)(p1));
    if (* p1 == 7) V—seg = 0xb000;
    else V—seg = 0xb800;
    printf("FarPointer = %Fp, segment = %04x, offset = %04x, V—
    segment = %p\n", p1, segment, offset, V—seg);
P24. C 15: p1 = (char *) 0x00400049;
P24. C 16: segment = ((unsigned)((unsigned long)(p1) >> 16));
    offset = ((unsigned)(p1));
    if (* p1 == 7) V—seg = 0xb000;
    else V—seg = 0xb800;
    printf("FarPointer = %Fp, segment = %04x, offset = %04x, V—
    segment = %p\n", p1, segment, offset, V—seg);
P24. C 17: }
P24. C 18:
P24. C 19:
*/
```

8.14 指向结构的指针和符号 —>

在 TC 中经常用到指向结构的指针, 请看程序 P25. C。

C>TYPE P25. C

```
#include "alloc.h"
typedef struct {
    char name[80];
    char class;
    float dist;
} star;

main() {
    star star0;          /* 定义 star0 为 star 型结构 */
}
```

```

star *ms=&star0; /* 指针 ms 指向 star 型结构 star0 的首址 */
strcpy(ms->name,"CHINA"); /* 将 CHINA 作为结构 ms 的成员 name 的内容 */
ms->class='A';
ms->dist=100.98;
printf("name=%s\n",star0.name);
printf("class=%c\n",star0.class);
printf("dist=%f\n",star0.dist);
printf("name=%s\n",(*ms).name); /* 下三句与上三句有同样结果输出 */
printf("class=%c\n",(*ms).class);
printf("dist=%f\n",(*ms).dist);
printf("name=%s\n",ms->name); /* 下三句与上三句有同样结果输出 */
printf("class=%c\n",ms->class);
printf("dist=%f\n",ms->dist);
}

```

程序中出现了【指向运算符 \rightarrow 】,它的意思是指针 ms 指向结构 $star$,要通过指针 ms 来访问结构的成员,就要用符号 \rightarrow (减号后跟一个大于符号)。如果不用指针而直接用结构名来访问其成员,可用

表达式 1: 结构名.成员名 /* 如 $star.name$ 等 */

从程序输出可以看出,

表达式 2: $(*$ 指针名).成员名

或

表达式 3: 指针名 \rightarrow 成员名

与表达式 1 的效果是一样的。常用的表达式 3 是表达式 2 的简写。

如在调试程序时使用调试表达式

$star0, r$

就可获得

{name: "CHINA", class: 'A', dist: 100.98 }

这样清晰的结果。输入另一些调试表达式,有

调试表达式	结果窗内显示内容
$sizeof(star)$	85
$sizeof(star0)$	85
$sizeof(my)$	2
$ms \rightarrow name$	"CHINA"
ms	DS:FF90
$ms+1$	DS:FFE5
$*ms.name$	Illegal structure operation
$star, r$	Improper use of a typedef symbol

调试表达式

0xFFE5-0xFF90

是将 ms+1 的地址与 ms 的地址相减得 85,这是结构 star 的长度。

如将语句

```
star star0;  
  
star *ms=&star0;
```

换成语句

```
star *ms;      /* 将指针 ms 说明为指向 star 型的结构的指针 */  
ms=(star *)malloc(sizeof(star)); /* 为结构 ms 分配为具有结构 star 长度的块 */
```

则有同样的输出结果。

8.15 用指向结构的指针作函数的参数

在 TC 库函数中有这样一类函数,它们的参数是指向某一结构的指针。例如,搜索磁盘目录函数(函数原型在 dir.h)

```
int __cdecl findfirst (const char * path, struct fblk * fblk, int attrib);  
int __cdecl findnext (struct fblk * fblk);
```

中参数 fblk 就是一个指向结构 fblk 的指针(注意:指针名和结构名相同也是可以的!但两者的含义是不一样的)。

```
C>TYPE P26.C  
#include <stdio.h>  
#include <dir.h>      /* 标头文件中包含这两个函数原型 */  
main()                /* 本程序找出 E 盘上全部 .C 文件 */  
{  
    struct fblk fblk;  /* 定义 fblk 为 fblk 型结构。为避免混淆,可将本句改成 struct fblk  
fblknew,下面所有 fblk 也改成 fblknew */  
    int done;  
    printf("Directory listing of E:\n"); /* 扩展名小写也无妨 */  
    done=findfirst("E:*.C",&fblk,0); /* 取结构 fblk 的地址传递,找第一个 .C */  
    while(!done){      /* 两函数找到一个 .C 文件,都返回 0; 否则返回 -1 */  
        printf("%s\n", fblk.ifname);  
        done=findnext(&fblk); /* 在 E 盘上找下一个 .C 文件 */  
    }  
}
```

函数中 *path 指路径名(字符串或字符串变量,可用 DOS 通配符 * 与 ?)。

```
C>TYPE P27.C  
struct p0{  
    int t1;  
    char ch;  
    double x;  
};  
  
main()
```

```

{
struct p0 p={8,'H',3.14};
prn(&p);                      /* 用 &p 作实参 */
}
prn(struct p0 *p1)             /* 指针 p1 为 p0 型结构,作形参 */
{
printf("ch=%c\n",p1->ch);      /* 输出 ch=H */
printf("t1 * x=%0.3f\n",p1->t1 * p1->x); /* 输出 t1 * x=25.120 */
}

```

8.16 结构中有指向自身的指针

C>TYPE P28.C

```

struct person
{
char * name;
char * address;
int num[3];
struct person * next; /* 指针 next 指向结构自身,但不是结构包含自身 */
}; /* 以下三句定义了三个 person 型指针 */
struct person a={"LiFang","ChongQing",{1,28,100}};
struct person b={"WangJi","HuNan",{8,66,99}};
struct person c={"Chuling","ShangHai",{0,45,188}};
main()
{
struct person * q; /* 指针 q 指向 person 型的结构 */
a.next=&b; /* a 的成员 next 取得 b 的地址 */
b.next=&c; /* b 的成员 next 取得 c 的地址 */
c.next=0; /* c 的成员 next 赋予 0 值 (NULL),表示终结 */
for(q=&a;q=q->next) /* 当 q 值为空 (NULL) 时结束循环 */
{ /* 注意,不能将 q=&a 换成 q=0; 或 q=NULL; */
printf("%s,%s",q->name,q->address); /* 单步调试到此语句暂停一下 */
printf("%d,%d,%d\n",q->num[0],q->num[1],q->num[2]);
}
}

```

结构不能把自身和具有相同类型的结构作为自己的成员。但是,由于指针只和地址打交道,因此允许在结构中有指向自身的指针。上述程序说明了其使用过程。对此程序用 F7 单步调试,有

调试表达式	显示内容
a	{"LiFang","ChongQing",{1,28,100},&b}
b	{"Wang Ji","Hu nan",{8,66,99},&c}
c	{"Chu Ling","Shang Hai",{0,45,188},NULL}
a,p	{DS:01B8,DS:01C0,{1,28,100},DS:01A0&b}
b,p	{DS:01CB,DS:01D3,{8,66,99},DS:01AC&c}
c,p	{DS:01DA,DS:01E3,{0,45,188},NULL}

```

c,m    DA 01 E3 01 00 00 2D 00 BC 00 00 00
b,m    CB 01 D3 01 08 00 42 00 63 00 AC 01
a,m    B8 01 C0 01 01 00 1C 00 64 00 A0 01
a,md   184 1 192 1 1 0 28 0 100 0 160 1
*q,m   B8 01 C0 01 01 00 1C 00 64 00 A0 01
q,m    Lvalue required
&q->name,p    DS:0194 &a
&q->address,p  DS:0196(char *)&a+2
&q->num[0],p   DS:0196(char *)&a+4
&q->num[1],p   DS:0196(char *)&a+6
&q->num[2],p   DS:0196(char *)&a+8
&q->num[3],p   DS:0196(char *)&a+10
&q->num[4],p   DS:01A0 &b
&q->next,p     DS:019E(char *)&a+10
&b,p          DS:01A0 &b
sizeof(a)      12
sizeof(a.next) 2
sizeof(a.address) 2
sizeof(a.num)  6
sizeof(a.next) 2
sizeof(q->next) 2
sizeof(person.next) Undefined symbol 'person'

```

及

调试表达式	值
*q	{"LiFang", "Chong Qing", {1, 28, 100}, &b}
q.p	DS:0194 &a
q.p+1	DS:01A0 &b
*q	{"Wang Ji", "Hu Nan", {8, 66, 99}, &c}
q.p	DS:01A0 &b
q.p+1	DS:01AC &c
*q	{"Chu Ling", "Shang Hai", {0, 45, 188}, NULL}
q.p	DS:01AC &c
q.p+1	DS:01B8
*q	{NULL, NULL, {30036, 25202, 11631}, DS:2043}
q.p	DS:000C
q.p+1	NULL

现在再看一个将访问数组元素改成这种形式的程序 P29.C。

C>TYPE P29.C

```

#include "stdio.h"
#include "string.h" /* 包含库函数 strcmp() 原型 */
#define MAX 100 /* 定义最大数组元素 */

struct person
{
    char *name;
    char *address;

```



```

int num[3];          /* 初始化类型为 person 的数组,  */
)a[MAX]='{"Wang Ji", "Hu Nan", {8,66,99}}, /* 即数组的每个元素均是 person  */
{"Pi Ping", "Min Min", {4,33,99}}}; /* 型的结构。 */
getname(char * name2);
/* strcmp(char s[],char t[]);  */
main()
{
    struct person * q, * s(char * name); /* s 为返回一个指针的函数  */
    char name0[40];
    if(getname(name0))
    {
        q=s(name0); /* 注意:不要将本句与下句合成一句: if(q=s(name0))  */
        if(q)
        {
            printf("%s,%s,",q->name,q->address);
            printf("%d,%d,%d\n",q->num[0],q->num[1],q->num[2]);
        }
        else printf("%s not found !\n",name0);
    } /* 如输入: Pi Ping,输出: Pi Ping,Min Min,4,33,99  */
} /* 如输入: Pi Ping,输出: Pi Ping not found !  */
struct person * s(char * name1) /* 从数组 a 中搜索指定的字符串  */
{
    struct person * q1;
    for(q1=a;q1<a+MAX;q1++)
        if(strcmp(q1->name,name1)==0)return(q1);
    return(0);
}
getname(char * name2) /* 从键盘输入字符串;如 Pi Ping  */
{
    /* 滤去非字母与空格符的字符  */
    char c, * q2=name2; /* q2 指向数组 name0[40]首址  */
    while((c=getchar())>='a' && c<='z' || c>='A' &&
        c<='Z' || c==' ') * q2++=c;
    * q2='\0'; /* 此句不能漏掉,因要使 q2 指向一字符串,字符串结尾为 '\0'  */
    return(q2-name2);
}
/*
strcmp(char s[],char t[])
{
    inti=0;
    while(s[i]==t[i])
        if(t[i++]=='\0')return(0);
    return(s[i]-t[i]);
}
*/

```

用 F7 热键调试,有

调试表达式	值
q2,s	""
*q2,d	0
name2,s	"Pi Ping"
*name2,d	80
name2-q2	-7q2-name2?
name2,p	DS:FFBE
q2,p	DS:FFC5

如将语句 `#include "string.h"` 去掉,并将注释中语句利用上,也可得到同样结果。

C>TYPE P30.C

```

struct person          /* 本程序对 P29.C 作了部分修改 */
{
    char * name;
    char * address;
    int num[3];
    struct person * next; /* 增加语句 */
}a[]={{{"Wang Ji","Hu Nan",{8,66,99},&a[1]},{"Pi Ping","Min Min",
{4,33,99},&a[2]},{"Zhu Qi","Jiang Xu",{1968,12,8},NULL}};
/* 上句为修改语句 */
struct person * s(char * name1)
{
    struct person * q1;
    for(q1=a; q1; q1=q1->next)修改语句 */
    if(strcmp(q1->name, name1)==0)return(q1);
    return(0);
}

```

则也能得到同一结果。上述实际上是使用了链表:

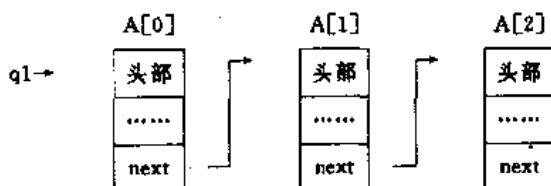


图 8—1

8.17 指向函数的指针(函数指针)

1. 定义

格式: 被指函数返回值的数据类型 `□(*指针名)()`;

该指针名称为【函数指针】。例如,

```
int (*p)();
```

注意: `p` 是一个指针,它指向任一返回值是整型数的函数。它不同于

```
int *p();
```

此处 p 是一个函数,该函数的返回值是一个整型指针。

2. 使用方法

第一步给函数指针赋值:

函数指针名 = 被指函数名;

例如, p 是一个指向返回整型数的函数的函数指针, f1(int x, int y) 和 f2(int z) 是两个返回整型数的函数, 则

```
p=f1;
```

是将函数 f1 的入口地址赋给了函数指针 p (这容易用调试表达式被调用函数名, p 进行验证); 如果接着你又用

```
p=f2;
```

则 f2 的入口地址赋给了 p。也就是说, 同一个函数指针可以指向不同的函数, 只要这些函数的返回类型和函数指针定义的类型相同。

第二步调用指向函数

(* 函数指针名)(被指函数的形参用实参代替)

```
C>TYPE P31.C
```

```
main(){
```

```
int add();
```

```
int x=1,y=2,z;
```

```
int (*p)();
```

```
p=add;          /* p 赋值, 函数只要名字 */
```

```
z=(*p)(x,3,y);   /* 被指向函数形参用实参代替 */
```

```
printf("%d\n",z); /* 上面两句相当于函数调用语句: z=add(x,3,y); */
```

```
}
```

```
add(int x1,int y1,int z1)
```

```
{return x1+y1+z1; }
```

结果输出值 6。

由于函数指针所指函数是不固定的, 因此常用来作另外函数的参数。

8.18 返回指针值的函数

定义格式: 函数返回数据类型 □ * 函数名(函数参数表)

例如库函数

```
void *—Cdecl memcpy(void *dest, const void *str, size_t n)
```

```
char *—Cdecl gets(char *s)
```

等, 返回值是一个指针。

第九章 结构与联合

9.1 结构

【结构, structure】(有些书上称为【结构体】)是一种数据类型,它是一些不同类型的、在使用上紧密相关的数据的集合。它相当于 Pascal 中的记录。有了结构,可以使访问一些相互有联系的数据更方便、清晰,应用更灵活。

一 结构的定义

1. 格式 1

```
struct [结构名]          /* 方括号表示其内容为任选的,下同 */
{
    数据类型 变量名 1;    /* 变量被称为结构的【成员】或【元素】,或【域】*/
    数据类型 变量名 2;    /* 不允许对结构成员直接初始化 */
    .....
    数据类型 变量名 n;
} [结构变量名 [=初值]]; /* 结构名与结构变量名中至少有一个被使用 */
                        /* 结构变量名可以有多个,变量名之间用逗号分开 */
```

注:结构名必须唯一。结构成员名在定义它们的结构内部必须唯一。不同结构之间,成员名与类型允许相同。结构名与结构变量名中至少要有一个被选择。

2. 格式 2

struct 已定义的结构名 结构变量名;

3. 格式 3

已定义的结构类型名 结构变量名;

其中结构类型名由

```
typedef struct [结构名]
{
    数据类型 变量名 1;
    数据类型 变量名 2;
    .....
    数据类型 变量名 n;
} 结构类型名;
```

定义。

例

定义四个结构:

```

struct MYST1                /* 结构 1 */
{
    char c;                  /* char c='A';初始化不允许 */
    int x,y;
    }first={'A',1,8};        /* 给成员赋初值 */
    struct myst2              /* 结构 2 */
    {
        /* Turbo C 标头文件中用得最多的形式 */
        unsigned char ch;
        double z;
        struct MYST1 second; /* 结构内部还可以定义结构 */
    };                        /* 尚未定义具体的结构变量 */
    struct {                  /* 结构 3 */
        float f1;
        long k;
        }three1,three2;      /* 多个结构变量用逗号分隔 */
    typedef struct {          /* 结构 4 */
        unsigned j—sp;
        unsigned j—ss;
        }j—buf[10];
    j—buf jbuffer;           /* 定义 jbuffer 为 j—buf[10] 那样的结构 */

```

三 说明

1. 结构名是可选项,它可以看成是所定义结构详细说明的缩写。当要把其它变量说明为这种结构时,就要用到它,或者说该选项不可少。例如,在结构 2 中把变量 second 说明为结构 MYST1,这就是说 second 也有 MYST1 的所有成员。注意,结构名不是结构变量名,它只能用在关键字 struct 之后,说明某一种形式的结构。不能用它访问结构成员。由于它只是一种结构类型的标识符而不是变量,所以不能用在调试表达式中。

```

C>TYPE STRUN1.C
struct SS{
    char c;
    int x,y;
    }first={'a',1,8};        /* 定义全局结构变量 first */
    struct AS {
        char ch;
        struct SS kk;
        }four;              /* 定义全局结构变量 four */
    struct AS * ptr;         /* 定义类型为 AS 的结构指针 */
    main()
    {
        printf("%d\t",first.x);
        printf("%d\t",four.kk.x); /* 访问多级结构成员的方法 */
        ptr=&four;           /* 结构指针 ptr 指向结构 four */
        ptr->ch='H';          /* 注意,ptr 和 four 类型应相同 */
        printf("%c\n",ptr->ch);
    }

```

```

/* 程序输出:1    0    H
   调试表达式: first: {'a', 1, 8}
               four: {'H', {'\x0', 0, 0}}
               ptr[0]: {'H', {'\x0', 0, 0}} */

```

2. 结构类型名表示用户定义了一种数据类型即结构。结构类型名从此就可以作为像其它数据类型名(char、int等等)一样使用。显然,它也不是变量,从而也不能用在表达式中,当然对它也谈不上初始化。(参见修饰符 typedef 的说明。)

```

C>TYPE STRUN12.C
struct SS{
char c;
int x,y;
}first={'a',1,8};
typedef struct AS{
    char ch;
    struct SS kk;
}four;
four four1; /* four 相当于 struct AS */
struct AS * ptr;
four * pc;
main()
{
printf("%d\t",first.x);
printf("%d\t",four1.kk.x);
ptr=&four1;
pc=&four1;
ptr->ch='H';
printf("%c\t%c\n",ptr->ch,pc->ch);
}
/* 程序输出:1    0    H    H */

```

3. 结构 1 指出了对结构成员初始化的方法。注意:不能对结构成员直接初始化。

例如

```

struct MYST1
{
    char    c='a';
    int     x=1,y=8;
}first;

```

是不允许的。

4. 函数定义时,函数参数也可定义为结构。

```

C>TYPE STRUN13.C
main()
{
    struct {
        int a,b;

```

```

    }arg;
    arg.a=1000;
    fab(arg);
}
fab(parm)      /* 传统式定义函数参数。写成现代式:
                fab(struct{int x,y;}parm) */
struct{
    int x,y;
}parm;        /* 注意 parm 与 arg 要一样 */
{
    printf("%d\n",parm.x); /* 程序输出:1000 */
}

```

四 结构的使用

例如 可将结构成员或整个结构传递给函数。

```

C>TYPE STRUNI4.C
p1(int y)
{
    printf("%d\t",y);
}
p2(char ch)
{
    printf("%c\t",ch);
}
p3(char *w)
{
    printf("%s\t",w);
}
struct yw{
    int x;
    char t[4];      /* 注意,此句如改成 char t[3]; 则不一样 */
};
p4(struct yw uu)    /* 对函数参数只允许用格式 2 或 3 定义结构 */
{                  /* 而不允许用格式 1 来定义结构 */
    printf("%d\t%s\n",uu.x,uu.t);
}
main()
{
    struct yw s;
    s.x=2;
    s.t[0]='B';
    s.t[1]='A';
    s.t[2]='H';
    s.t[3]='\0';
    p1(s.x);
    p2(s.t[1]);
}

```

```

p3(s.t);
p3(&s.t[1]);    /* &s.t[1] 表示结构 s 的成员 t[1] 的地址, 不能写成 s.&t[1] */
p4(s); /* 注意, 整个结构被传递时, 数组最后一个元素即 s.t[3] 不会被传递 */
}
/* 程序输出: 2  A  BAH  AH  2  BAH  */

```

9.2 结构指针

一 格式

指向一个结构的指针称为结构指针, 它可用两种格式定义:

格式 1: struct 结构名 结构指针名

格式 2: 结构类型名 结构指针名

结构指针名是在结构变量名前加上星号(*)。

二 结构指针的使用

当一个结构指针传递给函数时, 实际是将结构地址压入(或退出)栈。函数调用是使用的实际结构, 而不是结构的复制, 因此它能够在调用函数时修改结构中实际成员。

参见《指针》一章。

9.3 访问结构成员

形式 1: 使用圆点符(.)

结构变量名. 结构成员名 /* 圆点符也称【记录选择符, record selector】 */

形式 2: 使用箭头(->)

结构指针->结构成员名

形式 2 相当于

(* 结构变量名). 结构成员名

的专门写法。之所以用括号是因为圆点符比星号具有更高的优先级。

在调试程序时可以使用

结构变量名, R

这样的调试表达式清楚地看到结构成员的内容。参见集成环境中相关章节。

9.4 结构数组

当定义的结构变量为数组时就定义了结构数组。

```

C>TYPE STRUN15.C
#include "string.h"
main()
{

```



```

struct aab{
char xingming[18];
int niangling;
}aa[2]={{"Li Ming",18},{"Zhu Fu-you",16}};    /* 结构数组初始化 */
struct aab dd[3];    /* 这样定义的结构数组不能初始化 */
strcpy(dd[0].xingming,"Jiang Hen");
dd[0].niangling=20;
printf("%s:%d\n",dd[0].xingming,dd[0].niangling);
printf("%s:%d\n",aa[1].xingming,aa[1].niangling);
}
/* 程序输出:Jiang Hen:20
              Zhu Fu-you:16    */

```

9.5 用 sizeof 求结构的大小

```

int length;
length=sizeof( 结构变量名 );

```

由于 Turbo C 允许数据按字节对齐或按字对齐,因此结构各部分相加的和比它实际在机器中占用的尺寸可能要小一些。

```

C>TYPE STRUNI6.C
#define D(X) printf("%d+",sizeof(str. ##X))
main()
{
struct {
int x;
char ch;
double y;
long k;
unsigned char n;
char a;
char *p1;
char far *p2;
}str;
int x;
x=sizeof(str);
D(x);
D(ch);
D(y);
D(k);
D(n);
D(a);
D(p1);
D(p2);
printf("\b=%d\n",x);
}

```

/* 在集成环境下,当选 Options/Compiler/Code generation/Alignment Byte 时

输出: $2+1+8+4+1+1+2+4=23$

当选 Options/Compiler/Code generation/Alignment Word 时,输出:

$2+1+8+4+1+1+2+4=24$, 比实际各成员占用之和多 1

这实际是字边界处理:

- (1) 结构将开始于字 (Word) 边界 (偶数地址);
- (2) 任何非字符成员相对于结构开始都有一个偶数偏移量。
- (3) 结构末尾可能加上一个字节以保证整个结构包含偶数个字节。如果使用命令行编译器,可选项 `-a-` (相当 Byte) 或 `-a` (相当 Word)。

9.6 联合

【联合, union】(有些书上称为【共用体】)也是一种数据结构,在 Pascal 中称为变体记录。它的主要特点是,联合中也包括几个类型不尽相同的变量(包括结构变量等),但这些变量占用内存中相同的一些存储单元,占用的公用存储单元的大小由这些变量中占空间最大的变量决定的。换句话说,这些变量在内存中的地址有可能是相同的(结构的每个变量在内存中占用的地址是不同的)。这有两个好处,一是节省了存储单元,二是当公用存储单元中内容还未变更时,可将其内容赋给联合中的任一变量。或者说,对其中任何一个变量赋值,其它与之相关的变量也可能有相应的值。各变量的值由它在联合中的位置决定的。更确切地说,同一时刻只能对一个变量操作,而操作的结果将立即影响其它变量。这种效果有的类似于相互覆盖的子程序在一指定的内存内运行,同一时刻只有一个子程序被执行。

在集成环境中利用调试表达式很容易看到联合中各变量相互覆盖的情形。

一 定义

1. 格式 1

```
union [联合名]
{
    数据类型 变量名 1; /* 变量被称为联合的【成员】或【元素】,或【域】 */
    数据类型 变量名 2; /* 不允许直接对联合成员初始化 */
    .....
    数据类型 变量名 n;
}[联合变量名]; /* 联合名与联合变量名中至少有一个被使用 */
/* 多个变量名用逗号分开 */
/* 注意:定义时不能对变量赋初值,即初始化。 */
```

注:联合名必须唯一。联合成员名在定义它们的结构内部必须唯一。不同联合之间成员名与类型允许相同。联合名与联合变量名中至少要有一个被选择。

2. 格式 2

union 已定义的联合名 联合变量名;

3. 格式 3

已定义的联合类型名 联合变量名;

其中联合类型名由

```

typedef union [联合名]
{
    数据类型 变量名 1;
    数据类型 变量名 2;
    .....
    数据类型 变量名 n;
}联合类型名;

```

定义。

二 说明

1. 可用 sizeof 求出联合占用空间大小。

C>TYPE STRUNI7.C

```
#define D(X) printf("%d+", sizeof(str. ## X))
```

```
main()
```

```
{
```

```
struct AB{char ch[17];}; /* 结构占用 17 个字节 */
```

```
union {
```

```
int x;
```

```
char ch;
```

```
double y;
```

```
long k;
```

```
unsigned char n;
```

```
char a;
```

```
char *p1;
```

```
char far *p2;
```

```
struct AB m; /* 联合中可以包含结构 */
```

```
}str;
```

```
int x;
```

```
x=sizeof(str);
```

```
D(x);
```

```
D(ch);
```

```
D(y);
```

```
D(k);
```

```
D(n);
```

```
D(a);
```

```
D(p1);
```

```
D(p2);
```

```
D(m);
```

```
printf("b=%d\n", x);
```

```
}
```

```
/* 在集成环境下,当选 Options/Compiler/Code generation/Alignment Byte 时
```

```
输出:2+1+8+4+1+1+2+4+17=17
```

```
当选 Options/Compiler/Code generation/Alignment Word 时,输出:
```

```
2+1+8+4+1+1+2+4+18=18 */
```

2. 最好联合成员的长度相同,以便成员之间相互覆盖。
3. 对联合成员的访问方法同结构。
4. 不能将联合变量名当参数传送。

C>TYPE SRTUNI8.C

```
p1(int y)
{
    printf("%d\t",y);
}
p2(char ch)
{
    printf("%c\t",ch);
}
p3(char *w)
{
    printf("%s\t",w);
}
p4(int y,char *w)
{
    printf("%d\t%s\n",y,w);
}
main()
{
    union {
        int y;
        char t[3];
    } s;
    s.y=2;
    s.t[0]='B';
    s.t[1]='A';
    s.t[2]='\0'; /* 可以从输出看出,s.y=2 已被后面的变量赋值所覆盖 */
    printf("%d: %#x, %c: %c\n",s.y,s.y, s.y>>8 & 0xff, s.y & 0x00ff);
    p1(s.y); /* 允许将联合变量的成员作函数参数 */
    p2(s.t[1]);
    p3(s.t);
    p3(&s.t[1]);
    /* p4(s); 此句不能用,即不能将联合变量名作函数参数 */
}
/* 程序输出:16706,0x4142:A:B
16706  A  BA  A  */
```

5. 联合中可以包含结构,但它们都是互相独立的。

C>TYPE SRTUNI9.C

```
#include "string.h"
main() /* 本程序说明结构和联合的关系 */
{ /* aa,bb 和 cc 是三个独立的变量 */
    struct 体{
```

```

    char s[4],
    int t,
    }aa,
struct ff{
    int x[2],
    int y,
    }bb,
union hh{
    struct gg aa;
    struct ff bb;
    }cc;
aa.t=0xa;
bb.y=0x14;
cc.aa.t=0x64;
cc.bb.y=0xC8;
strcpy(cc.aa.s,"ABC");    /* "ABC" 相当于 0x41,0x42,0x43,'\0' */
strcpy(bb.x,"abc");        /* "abc" 相当于 0x61,0x62,0x63,'\0' */
strcpy(aa.s,"MNO");        /* "MNO" 相当于 0x4D,0x4E,0x4F,'\0' */
} /* 有调试表达式 (0x0'即是 '\0'),
    bb,x: { {0x6261, 0x63 , 0x14 }
    aa,x: { {0x4D, 0x4E, 0x4F, 0x0 }, 0xA }
    cc.bb,x: { { 0x4241, 0x43 }, 0xC8 }
    cc.aa,x: { {0x41, 0x42, 0x43, 0x0 }, 0xC8 }
从中可以看出两点:一是串尾 '\0' 起作用,二是结构和联合的元素间有区别 */

```

9.7 读取任意 *.DBF 文件中的数据

程序 READDBD.C 是一个能读取任意一个由 DBASE III、DBASE IV 或 FOXBASE 生成的数据库文件 *.DBF 中数据的程序。它根据 *.DBF 文件的构成原理使用了结构和联合,从而使读出数据的转换很方便、清晰。

```

C>TYPE READDBF.C
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "ctype.h"
#include "math.h"
#define RECMAX 127    /* 一个记录的最大字段数 */
#define PAGE 21      /* 显示暂停控制行数 */
typedef struct DBF—H—1{ /* 文件一开始的 32 个字节 */
char header; /* 文件头标志 03H(无 memoery 字段即不含 .DBT 文件时)
                或 83H(含 .DBT 文件时) */
char year;    /* 年 (16 进制数) */
char month;   /* 月 (16 进制数) */
char day;     /* 日 (16 进制数) */
unsigned long record—number; /* 记录总数,低位字节在前,高位字节在后。

```

```

        4 个 16 进制数依次为个位、十位、百位和千位 */
unsigned int filehead; /* 文件头长度,低位在前,高位在后。文件头最后一个字节,对 FOXBASE 或
        DBASE N 是 0DH,而对 DBASE III 则是 00H (但它前一个字节是 0DH,即相比之
        下它的文件头要多一个字节)。在文件头之后开始写记录数据 */
unsigned int record—length; /* 每个记录长度,低位在前,高位在后 */
char other[20]; /* 剩余未用字节,保留,一般为 0 */
};
union DBF1{
    char h[32];
    struct DBF—H—1 n;
    }d1;
typedef struct DBF—H—2{ /* 描述一个字段用的 32 字节 */
    char segname[10]; /* 一个字段名,以 ASCII 码存放 */
    char c; /* 未用,保留,一般为 0 */
    char type; /* 字段类型:C、N、D、L 和 M 的 ASCII 码值 */
    int addr—off; /* 首记录中字段对应的内存地址,偏移量,低位在前第一个字段对 DBASE III 是 07H 或
        0FH,对 FOXBASE 是 01H */
    int addr—seg; /* 首记录中字段对应的内存地址,段地址,低位在前。
        第一个字段为 00H */
    unsigned char length; /* 每个字段长度,字节数,最多为 254 字节 */
    unsigned char dec; /* 数字型字段的小数位位数。 */
    char other[15]; /* 剩余未用字节,保留,一般为 0 */
    };
union DBF2{
    char h[32];
    struct DBF—H—2 n;
    }d2[RECMAX];
char begin[1]; /* 记录的第一个字节,为 2AH 表示此记录已删除,否则为 20H */ char name[254];/
* 一个字段内容 */
int page;
main(int argc,char *argv[]) /* 用 C 语言读取 *.DBF 文件结构和记录 */
{ /* 程序利用联合中元素值共享的特点自动将有关字符换成数值 */
    FILE *in;
    char *ptr;
    int k,fn,j;
    int wseg; /* 一个记录中的字段数 */
    double xx;
    long yy;
    if(argc != 2)help(0);
    ptr=strpbrk(argv[1],"."); /* 检查输入文件有扩展名否 */
    if(ptr==NULL)strcat(argv[1],".DBF"); /* 文件名缺省为 .DBF */
    if((in=fopen(argv[1],"rb"))==NULL)help(0);/* 文件应该用二进制打开! */
    if(fread(d1.h,1,32,in) != 32)help(1);
    if(d1.n.header == 0x3 || d1.n.header == '\x83'); /* 核对文件标志字符 */
    else help(2);
    printf("数据库%s",argv[1]);

```

```

printf("建立于%d年%d月%d日\n",d1.n.year,d1.n.month,d1.n.day);
printf("共有%d个记录,每个记录长度为%d字节\n",d1.n.record-number,d1.n.record-length);
wseg=d1.n.filehead/32-1; /* 求出一个记录中的字段个数 */
printf("文件头长度为%d字节,共有%d个字段:\n",d1.n.filehead,wseg);
page +=3;
for(k=0;k<wseg;k++)
{
    if(fread(d2[k].h,1,32,in) !=32)help(1);
    printf("第%d个字段名:%s,类型:%c,长度:%d,小数:%d\n",k+1,
        d2[k].n.segname,d2[k].n.type,d2[k].n.length,d2[k].n.dec);
    pause();
}
if(fseek(in,d1.n.filehead,SEEK-SET))help(1);
for(fn=0;fn<d1.n.record-number;fn++) /* 处理一个记录 */
{
    if(kbhit()) /* 击任一键退出显示 */
    {
        getch();
        printf("查找为用户终止! \n");
        exit(2);
    }
    if(fread(begin,1,1,in) !=1)help(1);
    printf("记录%d",fn+1);
    if(begin[0]==0x2A)printf("已作删除标志,");
    else printf("未作删除标志,");
    for(k=0;k<wseg;k++) /* 处理字段 */
    {
        for(j=0;j<254;j++)name[j]='\0';
        if(fread(name,1,d2[k].n.length,in) !=d2[k].n.length)help(1);
        if(toupper(d2[k].n.type)=='C')printf("%s",name);
        else if (toupper(d2[k].n.type)=='N') printf("%s",name);
        /* 如要将字符转换为数字可用:
        {if(d2[k].n.dec ==0)
        {yy=atol(name);printf("%ld",yy);}
        else {xx=atof(name);printf("%f",xx);}
        }
        */
        else if(toupper(d2[k].n.type)=='D')
        { printf("%c%c%c%c%c年",name[0],name[1],name[2],name[3]);
          printf("%c%c月",name[4],name[5]);
          printf("%c%c日",name[6],name[7]);
        }
        else if(toupper(d2[k].n.type)=='L')printf(", %s.",name);
        else if(toupper(d2[k].n.type)=='M')printf("Memo,");
        /* 为 Memo 型时 name 为 10 个空格符 */
    }
    pause();
}

```

```

printf("\n");
}
printf("查找结束!\n");
fclose(in);
}
help(int x)    /* 帮助 */
{
switch(x)
{
case 0:
printf("正确的命令行:执行文件名 DBF 文件名\n");
exit(1);
case 1:
printf("不能读数据库!\n");
exit(2);
case 2:
printf("非数据库文件!\n");
exit(3);
}
}
pause()    /* 暂停 */
{
if(++page>PAGE)
{
printf("击任一键继续...");
getch();
printf("\n");
page=0;
}
}
/* 假定有数据库文件 AA.DBF,
Record # AA1      AA2 BB1      BB2 CC1 RECORD
      1 Zhangtao    1111.11 07/19/93 .F. Memo    1
      2 * LuoJialing 12.00 10/11/93 .T. Memo    2
用命令
C>DBF1 AA

```

则程序输出:

数据库 aa.DBF 建立于 93 年 11 月 22 日
 共有 2 个记录,每个记录长度为 40 字节
 文件头长度为 225 字节,共有 6 个字段:
 第 1 个字段名:AA1,类型:C,长度:10,小数:0
 第 2 个字段名:AA2,类型:N,长度:8,小数:2
 第 3 个字段名:BB1,类型:D,长度:8,小数:0
 第 4 个字段名:BB2,类型:L,长度:1,小数:0

第 5 个字段名:CC1,类型:M,长度:10,小数:0

第 6 个字段名:RECORD,类型:N,长度:2,小数:0

记录 1 未作删除标志,ZhangTao , 1111.11,1993 年 07 月 19 日, . F. .Memo, 1,

记录 2 已作删除标志,LuoJialing, 12.00,1993 年 10 月 11 日, . T. .Memo, 2,

查找结束! * /

(1000 0000)。

计算机中现已很少用反码运算。

3. 补码

(1)正数的补码就是它本身,即为原码。

(2)负数的补码,等于反码加 1。如求 -8 的补码

$$\begin{array}{r} 1111 \quad 0111 \quad -8 \text{ 的反码} \\ + \quad \quad \quad 1 \\ \hline 1111 \quad 1000 \quad -8 \text{ 的补码} \end{array}$$

-1 的补码等于 1111 1111。

-127 的补码为 1000 0001; 128 的补码为 1000 0000。

(3)+0 和 -0 的补码都是 0000 0000。

如以补码形式在一个字节中存放一个数,其最大值为 127(0111 1111),最小值为 -128 (1000 0000)。

容易用调试表达式立即查得负数的补码。例如输

-3,x

立即有结果

0xFFFD

以及

```
(char)(-28),x; 0x80
(int)(-128),x; 0xFF80
-32768,x; 0xFFFF8000
(int)(-32768); 0x8000
```

位的运算参见《运算符》一章中有关“~、<<、>>、&、^ 和 |”的部分。

例 利用位异或运算(^)的可交换性证明执行下列语句后可交换 x 和 y 的值:

```
x=x^y; y=y^x; x=x^y;
```

从后往前看,

$$\begin{aligned} y &= y^x = y^{\quad} (x^y) = (x^y)^y = x^y^y = x^0 = x \\ x &= x^y = x^{\quad} (y^x) = x^x^y = 0^y = y \end{aligned}$$

从中看出,一个数自己与自己异或运算得 0; 一个数与 0 异或运算还是它本身。

10.2 数循环移位

—1 unsigned —Cdecl —rotl (unsigned value, int count);

—2 unsigned long —Cdecl —lrotl(unsigned long val, int count);

将数值 value 或 val 向左循环移动 count 位,函数返回移位后的值。循环移位参见图 10-

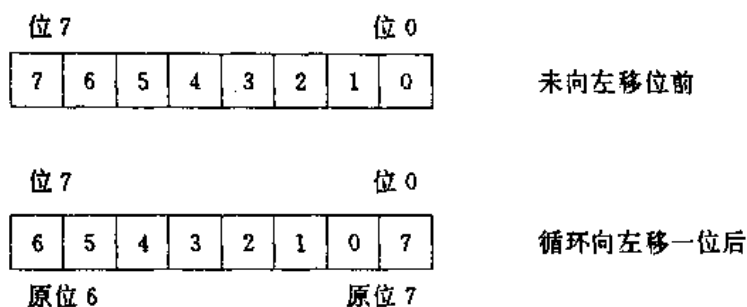


图 10-2

—3 unsigned —Cdecl —rotr (unsigned value, int count);
 —4 unsigned long —Cdecl —lrotr(unsigned long val,int count);
 将数值 value 或 val 向右循环移动 count 位,函数返回移位后的值。循环移位参见 图 10-3。

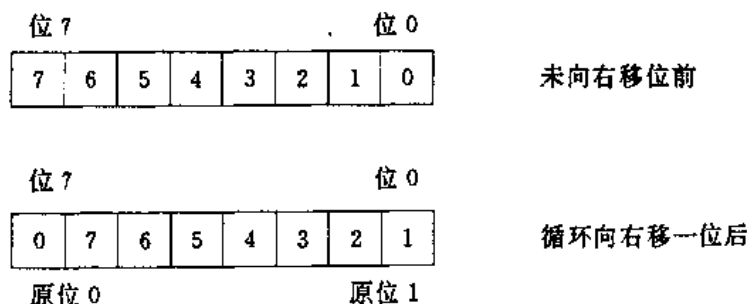


图 10-3

```
C>TYPE BIT1.C
#include "stdlib.h"
#include "stdio.h"
#define PV printf("0xabcd=%s\n",itoa(v,str,2))
main()
{
    unsigned v=0xabcd,v1;
    static char str[33];
    v1=rotr(v,1);
    PV;
    printf("循环左移一位后为:%#x,即%s\n",v1,itoa(v1,str,2));
    v=0xABCD;
    v1=rotr(v,1);
    printf("循环右移一位后为:%#x,即%s\n",v1,itoa(v1,str,2));
}
/* 程序输出, 0xabcd=1010101111001101
   循环左移一位后为:0x579b,即 101011110011011
   循环右移一位后为:0xd5e6,即 1101010111100110 */
```

10.3 位域 (bit—fields)

一个计算机位的值可以是 1 或 0,有些信息直接用计算机位作存储单位也许更经济。

【位域】（或【位段】或【场】）是以计算机位为单位、并由一个或一个以上的位构成的结构成员。位域可以有位域名，也可以没有。通过位域名来引用位域数据。引用方法同结构成员引用。

Turbo C 中一个位域可占 1 至 16 位，可以是 signed int，也可以是 unsigned int。位域在字内从低位到高位分配。它们在字节中的排列位置根据在结构中定义的先后次序依次从字节的低位排到高位，参见程序 BIT2.C 和表 10-1。整数域中的数以二进制补码形式存储。最左一位为正负号。例如一个一位宽的 signed int 位域只能表示 -1 和 0，因为任何非零数均被解释为 -1。位段不是数组，无地址，所以不能对它用 & 运算符。

虽然在一个结构中各个位域所占位数的和不是字节的整数倍也是可以的，但定义为字节的整数倍，并将不用位空起（不涉及引用时也可定义为无名位域）是有好处的，这样可免发生意外。

对位域赋值的方法可以采用位移动赋值（参见《运算符》一章），也可以利用联合中字节覆盖的方法，而使特定位得到相应值。一个具体的应用可参见《日期与时间函数》一章中函数 getftime() 涉及的结构 ftime。

```
C>TYPE BIT2.C
main()
{
    struct bit
    {
        int i:2;
        unsigned j:5;
        int k:1;
        unsigned l:8;
        char a[2];
    }b1;
    union bu
    {
        int x;
        struct bit y;
    }u1;
    u1.x=3;
    u1.y.a[0]='A';
    u1.y.a[1]='\0';
    b1=u1.y;
    printf("%d\n",b1.i);    /* -1 */
    printf("%d\n",b1.j);    /* 0 */
    printf("%d\n",b1.k);    /* 0 */
    printf("%d\n",b1.l);    /* 0 */
    printf("%d\n",u1.x);    /* 3 */
    printf("%s\n",u1.y.a);  /* A */
}
/* b1.rm: 03 00 41 00
   b1.r: {i:-1, j:0, k:0, l:0, a:"A" }
   u1.rm: 03 00 41 00
   u1: {3, {-1, 0, 0, 0, "A" } }
   u1.y.r: {i:-1, j:0, k:0, l:0, a:"A" }
*/
```

C>TYPE BIT3.C

```
#define PI printf("%d %x\n",ul.y,i,ul.x)
#define PK printf("%d %d\n",ul.y,k,z)
#define PM printf("%d %x\n",ul.y,m,ul.x)
main(){
struct bitfields{
int i,3; /* 位域 i 占 2 个位 */
unsigned j,3; /* 位域 j 占 5 个位 */
char ,2; /* 无位域名(无名位段) */
int ,4; /* 无位域名(无名位段) */
int k,1; /* 位域 k 只占一个位 */
unsigned m,3; /* 位域 m 占 2 个位 */
/* char h,2; Error: Not an allowed type */
/* unsigned char m,4; Error: Not an allowed type */
}b1;
union bitu{
int x,x1;
struct bitfields y;
}u1;
int z=0;
/* 依次由低位到高位取数字位给 ul.y.k 赋值.在表达式中引用时自动转为整型数
  因为位域 ul.y.k 只一位,因而它只能表示数 -1(位值为 1 时)或数 0(位值为 0 时) */
ul.y.k=-2;z=ul.y.k+100;PK; /* 0 100 */
ul.y.k=-1;z=ul.y.k+100;PK; /* -1 99 */
ul.y.k=0;z=ul.y.k+100;PK; /* 0 100 */
ul.y.k=2;z=ul.y.k+100;PK; /* 0 100 */
ul.y.k=3;z=ul.y.k+100;PK; /* -1 99 */
printf("\n");
ul.x=0; /* 假定一个字节不是 8 位而是 3 位,最高位为数符号位, */
/* 则补码值范围为 -4~3 */
/* 输出 */ /* 8 位补码 */ /* 3 位补码 */
ul.y.i=-8;PI; /* 0 0 */ /* -8=1111 1000 */ /* 0=000 */
ul.y.i=-7;PI; /* 1 1 */ /* -7=1111 1001 */ /* 1=001 */
ul.y.i=-6;PI; /* 2 2 */ /* -6=1111 1010 */ /* 2=010 */
ul.y.i=-5;PI; /* 3 3 */ /* -5=1111 1011 */ /* 3=011 */
ul.y.i=-4;PI; /* -4 4 */ /* -4=1111 1100 */ /* -4=100 */
ul.y.i=-3;PI; /* -3 5 */ /* -3=1111 1101 */ /* -3=101 */
ul.y.i=-2;PI; /* -2 6 */ /* -2=1111 1110 */ /* -2=110 */
ul.y.i=-1;PI; /* -1 7 */ /* -1=1111 1111 */ /* -1=111 */
ul.y.i=0;PI; /* 0 0 */ /* 0=0000 0100 */
ul.y.i=1;PI; /* 1 1 */ /* 1=0000 0001 */
ul.y.i=2;PI; /* 2 2 */ /* 2=0000 0010 */
ul.y.i=3;PI; /* 3 3 */ /* 3=0000 0011 */
ul.y.i=4;PI; /* -4 4 */ /* 4=0000 0100 */
ul.y.i=5;PI; /* -3 5 */ /* 5=0000 0101 */
ul.y.i=6;PI; /* -2 6 */ /* 6=0000 0110 */
```

```

ul.y.i=7;PI; /* -1 7 *// * 7=0000 0111 */
ul.y.i=8;PI; /* 0 0 *// * 8=0000 1000 */
ul.y.i=9;PI; /* 1 1 *// * 9=0000 1001 */
ul.y.i=10;PI; /* 2 2 *// * 10=0000 1010 */
ul.y.i=11;PI; /* 3 3 *// * 11=0000 1011 */
printf("\n"); /* 注意,ul.y.m 为无符号数,3 位值 0~7 */
ul.y.m=-8;PM; /* 0 3 */
ul.y.m=-7;PM; /* 1 2003 */
ul.y.m=-6;PM; /* 2 4003 */
ul.y.m=-5;PM; /* 3 6003 */
ul.y.m=-4;PM; /* 4 8003 */
ul.y.m=-3;PM; /* 5 a003 */
ul.y.m=-2;PM; /* 6 c003 */
ul.y.m=-1;PM; /* 7 e003 */
ul.y.m=0;PM; /* 0 3 */
ul.y.m=1;PM; /* 1 2003 */
ul.y.m=2;PM; /* 2 4003 */
ul.y.m=3;PM; /* 3 6003 */
ul.y.m=4;PM; /* 4 8003 */
ul.y.m=5;PM; /* 5 a003 */
ul.y.m=6;PM; /* 6 c003 */
ul.y.m=7;PM; /* 7 e003 */
ul.y.m=8;PM; /* 0 3 */
ul.y.m=9;PM; /* 1 2003 */
ul.y.m=10;PM; /* 2 4003 */
ul.y.m=11;PM; /* 3 6003 */
/* 结果参见表 10-1 */
printf("%d %d 0x%x\n",ul.x,ul.x1,ul.x); /* 24579 24579 0x6003 */
ul.x=0x65;
ul.x1=0x01; /* ul.rm: 65 00 */
/* ul.rm: 01 00 */
printf("%d %d\n",ul.x,ul.x1); /* 1 1 */
}

```

表 10-1 结构 bitfields 中位域的空间排列格式

值	0	1	1	0										0	1	1
位	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
名	m			k	未用			未用			j			i		

如果把程序 BIT3.C 中的 bitfields 结构作一些更改:

```

struct bitfields{
int i;3;
unsigned j;3;
char ;0; /* 无名位域长度为 0,有名位域长度不能定义为 0 */
int ;0; /* 否则出现 Error: Structure or union syntax error */
}

```

```

int k:1;
unsigned m:3;
}b1;

```

结果 ul.x 中的输出内容发生变化,由 x003 形式变成 x03。这是什么原因?当遇到 0 长度的位段时,位域便从下一个字节开始存放,(参见表 10-2。)

表 10-2

位 名	第 二 字 节								第 一 字 节							
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
					m			K	未用		j			i		

```

C>TYPE BIT4.C
main(){
struct bit—y{
int y1:3;
unsigned y2:3;      /* 跨字节存放 */
int y3:3;           /* 位域名不能是数组名,即无数组位域 */
int y4:3;
int y5:1;
unsigned y6:5;      /* 超过 2 个字节,便在第 3 个字节中存放 */
}b1;
struct bit—x{
int x1;
char x2;
}b2;
union bit—u{
struct bit—y y;
struct bit—x x;
}u1;
u1.x.x1=0xf5ee;      /* 预置值 */
u1.x.x2=0x7f;
printf("%d\n",sizeof(u1));      /* 3 */
printf("%d\n",sizeof(b1));      /* 3 */
printf("%d\n",u1.y.y1);         /* -2 */
printf("%u\n",u1.y.y2);         /* 5 */
printf("%d\n",u1.y.y3);         /* -1 */
printf("%d\n",u1.y.y4);         /* 2 */
printf("%d\n",u1.y.y5);         /* -1 */
printf("%u\n",u1.y.y6);         /* 31 */
}
/*      7      F      F      5      E      E
        0111   1111   1111   0101   1110   1110      三个字节
                                y6   y5   y4   y3   y2   y1      位域名
        0111   11     11111   1     010   111   101   110
                                31    -1    2     -1    5     -2      位域值

```


第十一章 预处理指令和编译控制行

编译预处理指令主要用来扩展 C 语言的编程环境,一个指令必须独占一个程序行(称编译控制行)。预处理指令以 # 开头, # 前后可以有空格。它有以下几种:

功 能	节 号	指 令	说 明
定义宏指令	11.1.1, 11.1.2	#define	
(调试宏	11.1.4)		
(预定义宏	11.1.5)		
取消宏定义指令	11.2	#undef	
文件包含(嵌入)指令	11.3.1	#include	
(标头文件	11.3.2)		
条件编译指令	11.4.1	#if	如果
	11.4.1	#else	否则
	11.4.1	#elif	否则如果
	11.4.1	#endif	结束如果
	11.4.2	#ifdef	如果定义了
	11.4.2	#if defined()	如果(定义了)
	11.4.3	#ifndef	如果没有定义
出错指令	11.5	#error	
报告现程序有汇编代码指令	11.6	#pragma inline	
警告(Warning)处理指令	11.7	#pragma warn	
保证 huge 函数执行时不变寄存器值指令	11.8	#pragma saveregs	
将行号嵌入执行文件指令	11.9	#line	
空编译指令	11.10	#	
(生成列表文件的预处理程序 CPP.EXE 11.11)			

11.1 定义宏指令

#define 指令用于定义宏。宏是用一个标识符来代替一个正文串,在程序中每次遇到该标识符时就用该正文串进行词法单元替换,称【宏扩展】或【宏代换】。定义的宏一般有两种格式,不带参数的宏和带参数的宏。

使用宏的好处是,使程序书写简洁,修改时可做到一改百改,既省事又不容易出错。程序执行时由于它不象函数调用要涉及进出栈等操作(分配存储单元、保留现场、值传递与返回),因而与实现同等功能的函数相比,它的执行速度将比调用函数执行要快。它的另一个优点是,还可以用宏替代其它 C 语言的函数,因而有利于程序移植。

使用宏的缺点是,它使程序占用空间变大,因为在宏扩展时,有几个地方用这个宏,哪几个地方就有宏代码嵌入;另外,在编译时 Turbo C 不作检查,因此当发生错误结果时不易清查(必要时你可用 CPP.EXE 工具帮助,生成 *.i 文件观察宏扩展结果)。最后,在集成环境里调试时它不被跟踪,宏名也不能用在调试表达式中等。

11.1.1 定义不带参数的宏

格式: #define 宏名 正文串 /* 宏名和正文串间用空格隔开, 句尾无分号。习惯上当正文串为常量时称【字符常量】, 而称宏名为【符号常量】。
*/

例

```
#define FALSE 0          /* 定义常量 */
#define TRUE 1
#define ONE 1            /* 不同宏名可定义有同一个值 */
#define TWO 2
#define THREE ONE+TWO    /* 可以利用前面已定义宏名并运算 */
#define PP printf("%d\n", k); x=2 /* 定义一组程序语句 */
#define ——GRAPHX——DEF—— /* 定义为空即扩展后什么也不出现的宏。在程序中或对调试宏写与不写是不同的。写了表示定义了这个宏, 不写则表示没有定义, 而定义与没有定义是两个完全不同的概念, 它们有不同的用处 */
#define MSG "That is not a legal cell." /* 定义字符串 */
#define FORMFEED '\014' /* 定义八进制数 14 */
```

11.1.2 定义带参数的宏

格式: #define 宏名(参数表) 含参数的正文串

说明: 多个参数间用逗号隔开

例

```
#define MIN(a,b) ((a)<(b)) ? (a):(b) /* 定义a和b的最小值 */
#define var(i,j) (i##j) /* 两个词法单元合并, 如 var(xx,88) 扩展后为 xx88 即将 ## 它前后的空格消去。这又称标识符【粘接】。粘接是在所有宏扩展后才实现的。## 前后可有空格 */
#define TRA(fla) printf("#fla" "%d\n", fla)
/* TRA(1024) 相当于 printf("1024" "%d\n", 1024), 结果印出 10241024。
这可以称为【# 使宏参数字符串化】, 即在宏扩展时将 # 后的参数用像 "实参" 这样的字符串替换。使用它要注意符号 # 应满足两个条件才起作用: 一是 # 在含参数的字符串中, 二是 # 不能在西文双引号中 */
```

11.1.3 有关宏的其它一些说明

1. 描述宏名的标识符只允许 A ~ Z、a ~ z、0 ~ 9 或下划线 (_), 并且必须以字母或下划线开头。
2. 习惯上常将宏名或宏参数用大写字符表示, 这样便于阅读理解, 但这不是必须的。
3. 在同一程序中这两种格式的宏名不允许相同, 即使用某一格式定义一个宏名后, 该宏名不允许再用任一种格式定义。
4. 在被定义的含参数正文串或字符常量内部出现宏参数时, 该参数不能被扩展。宏不能对自身扩展。
5. 当被定义的宏在一行上写不下时可以写到下一行上去, 但行尾一定要加上续行符号即反斜杠 (\)。这跟其它源程序语句不同, 源程序语句只在语句中的字符串在一行写不下而要求续行 (串被分写到两行上) 时才必须使用这种续行符, 在其它情形可以不写。例如,

```
#define Pprint printf("This is a word: %s, But that is a byte: %s\n", \
    ptr1, ptr2)
```

注意:续行上的前面空格即上行回车换行符后的空格被认为是宏定义串的一部分。因上句中 ptr1 和 ptr2 是两个指针变量,故前导空格不起作用,但如将上述宏定义为

```
#define Pprint printf("This is a word: %s, \
    But that is a byte: %s\n", ptr1, ptr2)
```

即截断的字符串必须从行第一个字符开始写起,否则宏扩展时将出现多余的空格符。注意,对宏定义续行必须用续行符“\”,而不能使用在行尾和续行头部加上西文双引号的方法,即

```
#define Pprint printf("This is a word: %s, "
    "But that is a byte: %s\n", ptr1, ptr2)
```

如果不是宏定义而是一般的 C 语句,这是允许的。

6. 宏起作用的范围或称宏作用域是从它开始定义处到它被撤消定义 (#undef) 时为止,或者说,只要它定义后未被撤消,则从定义开始后一直起作用。通常将宏定义成公用的,即在同一个程序各处都起作用,因此常将它定义在程序头部,而不是在程序中随意插入。要使已定义宏在多个程序中都起相同作用,可把它放到一个头文件 (*.h) 中,然后在程序中包含该头文件即可。

C>DEF1.C

```
main() /* 这是一个错误定义宏的例子 */
{
    int x=0;
    if (x>0)
        #define FUYI=1
    else
        /* Error: Misplaced else */
        #define FUYI=-1 /* Warning: Redefinition of 'FUYI' is not identical */
    printf("%d %d\n", x, FUYI); /* Error: Expression syntax */
}
```

7. 为什么定义宏时常加圆括号? 如定义流结束的宏

```
#define feof(f) ((f) -> flags & _FEOF)
```

含参数正文串中用了两层圆括号。这是因为一是 f 可能本身是一个复杂的表达式,如不用圆括号,很可能在涉及运算符优先时出错;其次,宏名 feof(f) 也可能为其它表达式引用,因此也可能遇到运算优先等语法问题。为避免不可预见的麻烦,保证使用可靠,加圆括号是一个办法。当然,也不是绝对必要的。

8. 定义的宏可以在源程序中不被引用。

9. 一个宏扩展后的字符数不能多于 4096。

11.1.4 调试宏

在调试程序时你也可以定义供调试用的宏,但这种宏只在调试时起作用,一离开调试则无效。关于调试用宏可参见集成环境中 Options/Compiler/Defines 命令,它允许一行上最多输入 96 个字符,它们由一个或多个调试宏组成。在定义方面两者也有一些不同,如调试宏可能

要用到等号(=)。可以在宏中包含分号,但必须在分号前放上反斜杠(\)。Turbo C 的调试宏虽然在指出某宏有否定义方面是很方便实用的,但并不那么好用,使用时应小心。注意:对同一个宏不要既用命令定义,同时又在源程序中定义。

11.1.5 预定义宏

【预定义宏】是 Turbo C 预先定义的,该宏名不允许取消或重定义。宏名前后均有两个下划线。当选择好编译环境后,这些预定义宏的值就自行确定了。

有五个预定义宏是 ANSI 标准所要求的,——LINE——、——FILE——、——DATE——、——TIME——和 ——STDC——,称为【标准宏】。除最后一个宏需在集成环境中用 ANSI keywords only On 命令时才有定义外,其余几个在整个程序执行中都起作用。

1. ——LINE——

当前正在处理的程序行的行号,源程序第一行定义为 1(空行也算一行)。它是一个十进制常量。

2. ——FILE——

当前处理的源程序名(是一个字符串)。它在编译处理完 #include、#line 指令或嵌入文件处理完时改变。

```
C>TYPE DEF2.C
```

```
main()
{printf("子程序:%s\n",——FILE——);
}
```

```
C>TYPE DEF3.C
```

```
#include "stdio.h"
#include "process.h"
main()
{
printf("%s\n",——FILE——);
spawnl(P-WAIT,"DEF2.exe",NULL);
printf("%s\n",——FILE——);
}
```

/* 程序输出:

DEF3.C

子程序,DEF2.C 不管 DEF2.C 是否存在都有此输出,当然 DEF2.EXE 必须有 DEF3.C */

3. ——DATE——

预处理程序开始处理源程序时的日期(注意:不是源程序生成或执行程序时的日期),是一个字符串,表示形式为

月 日 年

4. ——TIME——

预处理程序开始处理源程序时的时间(注意:不是源程序生成或执行程序时的时间),是一个字符串,表示形式为:

时 分 秒

5. ——STDC——

当使用命令 Options/Compiler/Source/ANSI keywords only On 或在命令行上用相容标记 -A 时它有常量 1, 否则没有定义。因为它可能没有定义, 所以你不能用

```
if{}
else{}
```

这样的条件语句, 而只能用预处理指令。

```
C>TYPE DEF4.C
main()
{
printf("Turbo C 实现了 ANSI 标准所要求的五个预定义宏,\n");
printf("当前行号(——LINE——),%d\n",——LINE——);
printf("当前处理的源程序名(——FILE——),%s\n",——FILE——);
printf("预处理程序开始处理当前源程序的日期(——DATE——),%s\n",——DATE——);
printf("预处理程序开始处理当前源程序的时间(——TIME——),%s\n",——TIME——);
/* 如写成
if(——STDC——)
printf("——STDC——=1,表示仅使用 ANSI 关键字\n");
else printf("——STDC——无定义,表示使用的是 Turbo C 的关键字\n");
则出现错误 Unreachable code */
#if(——STDC——)
printf("——STDC——=1,表示仅使用 ANSI 关键字\n");
#else printf("——STDC——无定义,表示使用的是 Turbo C 的关键字\n");
#endif
printf("\nTurbo C 自定义的十个宏:\n");
printf("当前 Turbo C 的版本号(——TURBOC——),%#x\n",——TURBOC——);
#if(——PASCAL——)
/* 注意: #if 与 #else 之间语句不能用 {} 括起来! */
printf("——PASCAL——=1,-p 标志已用\n");
printf("——CDECL——未定义\n");
#else
printf("——PASCAL——未定义,-p 标志未用\n");
printf("——CDECL——=1\n");
#endif
printf("——MSDOS 对所有编译程序均为整数常量%d\n",——MSDOS——);
printf("现选 Small 存储模式、Options/Compiler/Source/ANSI only On,"
"Options/Compiler/Code generation/Calling convention C 编译,有\n");
#if defined(——TINY——)
printf("——TINY——=%d\n",——TINY——);
#elif defined(——SMALL——)
printf("——SMALL——=%d\n",——SMALL——);
#elif defined(——MEDIUM——)
printf("——MEDIUM——=%d\n",——MEDIUM——);
#elif defined(——COMPACT——)
printf("——COMPACT——=%d\n",——COMPACT——);
```

```
#elif defined(——LARGE——)
    printf("——LARGE——=%d\n",——LARGE——);
#elif defined(——HUGE——)
    printf("HUGE——=%d\n",——HUGE——);
#endif
}
```

/* 程序输出:

Turbo C 实现了 ANSI 标准所要求的五个预定义宏:

当前行号(——LINE——),4

当前处理的源程序名(——FILE——);DEF3.C

预处理程序开始处理当前源程序的日期(——DATE——);Oct 27 1993

预处理程序开始处理当前源程序的时间(——TIME——);10:53:53

——STDC——=1,表示仅使用 ANSI 关键字

Turbo C 自定义的十个宏:

当前 Turbo C 的版本号(——TURBOC——);0x18d

——PASCAL——未定义,-p 标志未用

——CDECL——=1

——MSDOS 对所有编译程序均为整数常量 1

现选 Small 存储模式,Options/Compiler/Source/ASNI only On,Options/Compiler

/Code generation/Calling convention C 编译,有

——SMALL——=1 */

Turbo C 自己还定义了一些宏(为区别于标准宏,也称它们为 Turbo C【内部宏】):

6. ——TURBOC——

当前使用的 Turbo C 的版本号,一个十六进制的常量。如版本号 1.2 是 0x102 等等。高字节表示主要版本号,低字节表示次要版本号。

7. ——PASCAL——

测试命令行上是否用了 -p 标记,或者相当于在集成环境中使用了测试命令

Options/Compiler/Code generation/Calling convention Pascal

结果生成可和 Pascal 连接的目标文件 *.OBJ。

8. ——MSDOS——

在 MS-DOS 环境下,对所有编译程序均有整常量 1。

9. ——CDECL——

当命令行上没有标记 -p 时它有整型常量 1,否则没有定义。

10. ——TINY——、——SMALL——、——MEDIUM——、——COMPACT——、——LARGE——和——HUGE——

这六个是分别和当前所取的存储模式相对应的存储宏。对某一编译过程只有其中的一个被定义,其余的则没有定义。被定义的宏有宏值(即宏所对应定义的值)1。

11.2 取消宏定义指令

当一个宏在某些程序段内使用后想更改前或删除时可 #undef 指令删除早先定义的宏。

格式: #undef 宏名

说明：Turbo C 提供的库函数中有一些是宏。可同时定义实现同一目标的函数或宏。为减少执行程序代码长度可用函数；若内存允许，为加快运算速度可用宏。当源程序中包含一个标头文件（*.H）时，标头文件往往缺省用宏。为避免使用宏而用函数，最好的办法就是用本指令。例如，在某包含文件中使用了一个宏，则在源程序中涉及该包含文件的包含指令下面使用本指令。

C>TYPE DEF5.C

```
#define ONE 1
demo—1()
{
printf("%d\n",ONE);
}
#undef ONE      /* 无此句将出现错误 */
#define ONE "first"
demo—2()
{
printf(ONE"表示第一\n");
}
main()
{
demo—1();
demo—2();
}
/* 程序输出
1
first 表示第一 */
```

C>DEF6.C

```
#define VAR(X,Y) (X ## Y)
/* #define VAR 100 此句是错误的，不允许用两个一样的名 VAR */
varp(int t)
{
return t * t;
}
main—0()
{
int first=2;
first=varp(VAR(fir,st));
printf("%d\n",first);
}
#undef VAR      /* 也可写成 #undef VAR(X,Y) */
#define VAR first
main—1()
{
int first=3;
first=varp(VAR);
```

```

printf("%d\n",VAR);
}
main()
{
main--0();
main--1();
}
/* 程序输出 4      9      */

```

11.3 文件包含（嵌入）指令

文件包含指令 `#include` 告诉编译程序在编译时将它后面的源文件整个地嵌入到该指令所在行处，然后对它进行编译，而该指令行本身不产生执行代码。

11.3.1 包含指令格式

格式 1: `#include "被包含文件名"`

格式 2: `#include <被包含文件名>`

说明:

1. 被包含文件名不能使用字符常量或字符串变量，也不能使用词法单元的拼合，它必须是一个真正的文件名。文件名可以带路径。

2. 格式 1 和二区别在于寻找被包含文件的方法不同:

它们相同的是，如果文件名带有路径名，编译时只在指定的目录中寻找被包含文件。

在集成环境下，对格式 1，首先要在当前目录中查找，然后在由

`Options/Directories/Include directories`

命令指定的包含目录中寻找；而对格式 2 则不会在当前目录中寻找被包含文件，而只在集成环境指定的包含目录中寻找。如未找到被包含文件，往往显示

`Unable to open include file '被包含文件名'`

对使用命令行编译时最好用 `-I` 开关指定包含文件所在的目录。对有些系统，环境变量 `PATH` 指出的路径有时对 `TCC.EXE` 后面的包含文件不起作用，它们不是包含文件的缺省路径。

值得指出的是，不要将不同的 Turbo C 版本的包含文件同装在一个硬盘上，特别不要将包含文件装在 `TC.EXE` 所在的目录里。由于它们的包含文件虽同名但内容已不全相同，所以可能会产生不易发现的问题。

3. 包含文件一般是头文件（*.h）或 C 源文件（*.c），但被包含的 C 源文件一般不带主函数名 `main`（参见《程序结构和主函数》一章）。

4. 可以用 `CPP.EXE` 分析源文件装入包含文件后的情况，即分析生成的 *.i 文件，以观察预处理后的结果。

5. 包含文件可以嵌套。例如，`stdio.h` 中又包含 `stdarg.h` 文件。

11.3.2 标头文件

包含文件中有一类文件，因它们一般放在源文件头部（但不绝对必要），故称为【头文件】。它们常带有扩展名“.h”（h 是 head 第一个字母）。Turbo C 提供了一些标准的头部文件，为区别于用户自编的头文件，把它简称【标头文件】。

头文件和一般的 C 源文件本质上并没有多少区别，主要是定义一些常用的常量、变量、

宏、函数原型、结构、联合等。相对而言，直接定义函数较少。

一 Turbo C 的标头文件

1. alloc.h	内存管理函数和变量
2. assert.h	专门定义 assert 宏
3. bios.h	MS-DOS 的 ROM BIOS 功能调用
4. conio.h	直接 MS-DOS 控制台输入 / 输出操作
5. ctype.h	字符分类定义的宏
6. dir.h	定义目录与路径的结构、宏和函数
7. dos.h	定义 MS-DOS 和 Intel iAPX86 微处理系列的结构、联合、宏和函数
8. errno.h	定义系统错误变量 errno 的值与对应的助记符
9. fcntl.h	定义在与 open 库子程序连接时的符号常量
10. float.h	定义浮点运算有关的符号常量与同 8087 相关的函数
11. graphics.h	定义有关图形的枚举、结构、宏和函数原型等
12. io.h	低级输入 / 输出函数
13. limits.h	定义字符位、字符或整型数的范围
14. math.h	数字浮点运算函数、宏及数字符号常量
15. mem.h	内存操作（其大部分内容在 string.h 也有）
16. process.h	进程管理
17. setjmp.h	非局部转移
18. share.h	sopen 函数用的共享模式
19. signal.h	ANSI 定义的信号传输
20. stdarg.h	定义接收可变参数的函数中的参数
21. stddef.h	定义公共数据类型、NULL 和变量 errno
22. stdio.h	流输入 / 输出
23. stdlib.h	定义一些公共的类型、变量和函数
24. string.h	内存和字符串函数
25. values.h	一些跟机器相关的与 UNIX V 相兼容的重要常数的符号名
26. time.h	处理时间结构和函数
27. timeb.h	函数 ftime() 及相关的结构
28. types.h	定义时间的数据类型 time_t
29. stat.h	打开和创建文件用到的符号常量和文件状态

二 使用标头文件的注意事项

1. 用户自定义的头文件不要和它同名。为防止重名，可在集成环境下（不管编辑窗内有无装入程序）按两次 F1 键，便显示一个帮助索引小窗：Index。用光标键将亮条移到

Header Files

项上后回车，便可看到显示全部标头文件名的小窗：HEADER FILES。

移动光标键对标头文件名小窗选择某一标头文件名后回车，便看到有关此标头文件的帮助目录项，一般包括：

函数 Functions

常数 Constants
数据类型 data types
全局变量 global variables
宏 Macros

2. 调用库函数一般要包含相应的标头文件。虽然个别库函数（如 printf）不要包含文件也可能不会出错，但对有些函数却是很危险的。例如

C>TYPE DEF7.C

```
#include "math.h" /* 如果不用此句也能无错编译后生成执行文件，但不会得到正确结果。此句也可用语句 double sin(double x); 代替 */
```

```
main()
{
double x=3.0;
x=sin(x);
printf("%f\n",x);
}
```

因此，建议通常引用库函数时要写入库函数原型所在的包含文件。事实上，包含库函数原型所在的包含文件有两个好处，一是编译器将对所有函数的参数的个数和类型进行检查。如果有错，则发出错误信息；其次，如果实参类型不对，还会自动转换为原型提供的类型。

3. 标头文件也可以用集成环境的编辑器读入观察，但应注意：一般不要轻易更改。所以最好你应用 README.COM 读入标头文件，即

C>README 标头文件名

因为该执行文件将标头文件读入后只能阅读而不允许修改，因而不会出差错。

4. 在编译时有可能会出现“Unable to open include file 标头文件名”，此时你应检查 DOS 的 CONFIG.SYS 文件中设置的可同时打开文件数，是否有像 FILES=20 这样的语句。

11.4 条件编译指令

条件编译指令允许你有选择地编译程序中某些部分，而对另一些部分不编译。

一 #if、#else、#elif 和 endif

（一）格式：

```
if 常量表达式 1            /* 如果常量表达式 1 非 0
   程序段语句 1            就执行程序段语句 1
#elif 常量表达式 2        否则如果常量表达式 2 非 0
   程序段语句 2            就执行程序段语句 2
#else
   程序段语句 3            否则就执行程序段语句 3
#endif                      条件编译结束 */
```

如果常量表达式的值为真，则它下面的程序段被编译，否则跳过。常量表达式是由常数和符号常量组成，而不能有变量。

C>TYPE DEF8.C

```
# define FUYI -1 /* 如果已定义此句再用 Options/Compiler/Defines 定义调试宏,则调试宏不起作用。如取消此句,改变调试宏便可得 不同的输出结果 */
main() /* 这是一个将条件指令写入函数内部的情况,这种情况少见 */
{
    #if (FUYI==1 || FUYI== -1) /* 如此句写成 #if FUYI 则调试宏不起作用 */
    printf("%d=", -1);
    #else
    printf("%d=", 0);
    #endif
}
```

说明

1. 它们可以构成嵌套结构, #if、#elif 总是与最近一个 #endif 配对。
2. 可以在常量表达式中使用 sizeof (求类型修饰符的字节长度运算符), 如

C>TYPE DEF9.C

```
main()
{
    #if(sizeof(char *)==2)
    printf("=2\n");
    #else
    printf("! =2\n");
    #endif
}
```

3. 可用 #if 0 (数字 0) 和 #endif 将部分程序括起来而避免嵌套注释。取消的注释 由空格符代替。如在集成环境 O/C/S/Nested comments Off 命令下, 可以用

```
#if 0
/* .....
    /* ..... */
    .....
*/
#endif
```

取消整个嵌套注释,但不能用

```
/* .....
    #if 0
        /* .....
            ..... */
        #endif
    .....
*/
```

取消嵌套注释中的另一部分注释。后者将产生编译错误。

二 #ifdef 和 #if defined()

该指令的意思是“如果定义了”某宏。在 ANSI 标准中使用运算符 defined(宏名), 它的

意思是指“定义了某宏”，注意：宏名一定要加圆括号。如果宏 Macro 已用 #define 指令定义过而后来又未用 #undef 取消，则 defined(Macro) 为真（有值 1），否则为假（值为 0）。它可以在 #if 指令之后的复合表达式中反复使用。如

```
#if defined(宏 1) || defined(宏 2)
```

注意：这两种指令都必须用 #endif 结束。

```
C>TYPE DEF10.C
```

```
#define SET-1
```

```
#define SET-2
```

```
#ifdef SET-2
```

```
#define negative -1
```

```
#endif
```

```
/* 漏了 #endif 将出现
```

```
Unexpected end of file in conditional on line # 错误 */
```

```
#ifdef SET-1 && SET-2
```

```
#define Zero 0
```

```
#endif
```

```
#if !defined(SET-3)
```

```
#define positive 1
```

```
#endif
```

```
main()
```

```
{
```

```
printf("%d %d %d\n", negative, Zero, positive);
```

```
}
```

三 #ifndef

该指令的意思是“如果没有定义”，它与 #if !defined() 指令相当。同样，它们应该用 #endif 结束。

11.5 出错指令

格式：#error 出错信息

出错信息不用加西文双引号括起来。编译中遇到出错指令时编译将停止，并原样印出出错信息。此指令主要用在程序调试阶段。

```
C>TYPE DEF11.C
```

```
#define MY 8
```

```
#if (MY != 0 && MY != 1)
```

```
#error MY 必须是零或壹！
```

```
#endif
```

```
main() /* 此程序在编译时将出现下列信息并停止编译，
```

```
Error 源文件名 行号：Error directive: MY 必须是零或壹！ */
```

```
()
```

11.6 报告现行程序有汇编代码的指令

格式: #pragma inline

当源程序中嵌有汇编代码时不能在集成环境中编译,而应用命令行编译器 TCC.EXE。为了报告程序中有汇编代码,可在程序顶部加上本指令。如果在源程序中未加此指令,则也可以对 TCC.EXE 选用 -B 编译项。如果这两种方法你都不用,则编译程序一遇到汇编语句就会重新开始,结果编译就会很费时间。使用此指令的目的就是节省编译时间。

11.7 警告(Warning)处理指令

格式 1: #pragma warn +xxx /* 警告开放 */

格式 2: #pragma warn -xxx /* 警告关闭 */

格式 3: #pragma warn .xxx /* 警告复原 */

说明:

1. 加号(+)表示如果发生警告,则会显示相应警告信息;用减号(-)时便不显示警告信息;小数点(.)表示警告将恢复程序开始编译前对此警告选定的值;可以显示或不显示。

2. xxx 是三个特定的字母,参见《程序编译和调试》一章中“错误、警告及提示信息”如果 xxx 是编译器不能识别的字符,则该指令被忽略,即不起任何作用。

3. 参见集成开发环境中菜单命令 Options/Compiler/Errors。

4. 一个有效的 pragma 指令的作用只有在它遇到另一个 pragma 指令时才可能改变,否则一直保持原有的作用。

5. 使用 TCC.EXE 编译时也可在命令行上设置 -wxxx 选项,以控制警告信息是否显示。但是,应当注意:编译优先考虑源程序内设置的 pragma 指令。

C>TYPE DEF12.C

```
#pragma warn -sus
main() /* 在编译前设置:O/C/Errors/ANSI violations/ */
{
    /* E,Suspicious pointer conversion 为 On 或 Off */
    int x=1;
    char *ptr=&x; /* 不产生警告 */
    #pragma warn +sus
    char *xx=&x; /* 会产生警告 */
    #pragma warn .sus /* 如开始编译时选 O/C/E/ANSI/E:/On 则产生警告信息 */
    char *yy=&x; /* 如选 O/C/E/ANSI/E:/Off 则不会产生警告信息 */
    printf("%d,%d,%d\n", *ptr, *xx, *yy); /* 输出 1,1,1 */
} /* 在集成环境中:
    O/C/Errors/ANSI violations/
    Display warnings E,Suspicious pointer conversion xxx 警告信息
    On On 或 Off -sus 不显示
    On On +sus 显示
    On Off +sus 不显示 */
```

11.8 保证 huge 函数执行时不变寄存器值指令

格式: #pragma savereg

此指令确保一个 huge 函数执行时不改变其各寄存器的数值。在与编译语言相连接时需
用此指令。指令被置于函数定义之前。每个这种指令只能应用于一个单独的 huge 函数。

11.9 将行号嵌入执行文件指令

格式: #line 开始行号["文件名"]

说明:

1. 开始行号将作为源程序第一行的行号,源程序的以后行号按此递增。
2. 文件名可以和源程序名不同,即是原文件名的一个别名。
3. 本指令常用于程序调试和一些特殊的应用中,用它替换 Turbo C 的预定义宏——
LINE——和 ——FILE——。

```
C>TYPE DEF13.C  
  
#line 100 "DEFTEST.C"  
main()  
{  
printf("当前行号为%d\n",——LINE——);  
}  
/* C>CPP DEF13.C 使用 CPP.EXE 分析 DEF13.C  
C>TYPE DEF13.I  
def13.c 1;  
deftset.c 100; main()  
deftset.c 101; {  
deftset.c 102; printf("当前行号为%d\n",102);  
deftset.c 103; }  
deftset.c 104;  
程序输出:当前行号为 4      */
```

11.10 空编译指令

格式: #

这是一个只有一个字符 # 的指令,它什么也不做。在编译时它总被忽略了。使用它只是出于完整性考虑。

11.11 生成列表文件的预处理程序 CPP.EXE

一 CPP 的作用

其作用是对程序员编写的 C 源程序中的包含文件和宏定义进行扩展处理,结果产生一个

扩展了的源程序的列表文件。列表文件是一个 ASCII 码文件,其内已不再有像 #include、#define、#if、#ifdef、#ifndef、#endif、#else 或 #line 这样的宏定义或条件编译语句,因为这类语句已在扩展或处理后被消去。换句话说,列表文件中不再包含有源文件中以 # 号开始的预处理控制行。下面用 Turbo C 带的 MCALC.C 文件详细说明之。

在 DOS 提示符下键入:

C>CPP -P- MCALC.C

便有一个列表文件 MCALC.I 生成。可在集成环境下用 File/Load F3 命令将它读入编辑窗内观察;也可用 DOS 的 TYPE 命令列出其内容。为简要叙述,下面对该列表文件说明时已将其许多行略去,中文注释是后来加上去的。

C>TYPE MCALC.I

```
/* 列表文件包括 C 源程序的各行、以及所有包含文件的相关行。语句号是连续的 */
/* 处理 MCALC.C 的第 1 句 */
mcalc.c 1;
/* 为简略起见,文中略去了和前句形式上类同的一些连续语句。下同 */
mcalc.c 5;
/* 开始处理 string.h, 行号从 1 开始 */
C:\TC\INCLUDE\string.h 1;
C:\TC\INCLUDE\string.h 16; typedef unsigned size_t;
C:\TC\INCLUDE\string.h 17;
/* 由于——STDC——为假(等于 0),所以下面将包含文件中—Cdecl 变为 cdecl */
C:\TC\INCLUDE\string.h 19; void * cdecl memcpy (void * dest,
const void * src, int c, size_t n);
/* 下列两行在 string.h 中为一行,它表明 CPP 对字符数较多的行的处理方法 */
C:\TC\INCLUDE\string.h 26; void cdecl movedata(unsigned srcseg,
unsigned srcoff, unsigned dstseg,
C:\TC\INCLUDE\string.h 37; size_t cdecl strlen(const char * s);
C:\TC\INCLUDE\string.h 51; char * cdecl strdup (char * s);
C:\TC\INCLUDE\string.h 52;
C:\TC\INCLUDE\string.h 59; char * cdecl strerror (const char * s);
C:\TC\INCLUDE\string.h 60;
/* 从上可见,源程序和包含文件中的空行、注释行以及预处理行均要占列表文
件的行行号。下面开始处理 mcalc.c 第 6 行 */
mcalc.c 6;
/* 开始处理 alloc.h, 行号从 1 开始 */
C:\TC\INCLUDE\alloc.h 1;
/* 由于选择项未选,因而采用了缺省的小型模式 */
C:\TC\INCLUDE\alloc.h 21; typedef int ptrdiff_t;
C:\TC\INCLUDE\alloc.h 22;
C:\TC\INCLUDE\alloc.h 38; int cdecl brk (void * addr);
C:\TC\INCLUDE\alloc.h 40;
C:\TC\INCLUDE\alloc.h 58;
mcalc.c 7;
/* 开始处理 stdarg.h, 行号从 1 开始 */
C:\TC\INCLUDE\stdarg.h 1;
/* 由于没有定义——STDARG) */
```

```

C:\TC\INCLUDE\stdarg.h 18: typedef void *va--list;
C:\TC\INCLUDE\stdarg.h 19:
mcalc.c 8:
    /* 开始处理 dos.h, 行号从 1 开始 */
C:\TC\INCLUDE\dos.h 1:
C:\TC\INCLUDE\dos.h 19: extern int cdecl -8087;
C:\TC\INCLUDE\dos.h 30:
C:\TC\INCLUDE\dos.h 40: struct fcb {
C:\TC\INCLUDE\dos.h 41: char fcb--drive;
C:\TC\INCLUDE\dos.h 42: char fcb--name[8];
C:\TC\INCLUDE\dos.h 43: char fcb--ext[3];
C:\TC\INCLUDE\dos.h 44: short fcb--curblk;
C:\TC\INCLUDE\dos.h 45: short fcb--reclsize;
C:\TC\INCLUDE\dos.h 46: long fcb--filesize;
C:\TC\INCLUDE\dos.h 47: short fcb--date;
C:\TC\INCLUDE\dos.h 48: char fcb--resv[10];
C:\TC\INCLUDE\dos.h 49: char fcb--currec;
C:\TC\INCLUDE\dos.h 50: long fcb--random;
C:\TC\INCLUDE\dos.h 51: };
C:\TC\INCLUDE\dos.h 125: union REGS {
C:\TC\INCLUDE\dos.h 126: struct WORDREGS x;
C:\TC\INCLUDE\dos.h 127: struct BYTEREGS h;
C:\TC\INCLUDE\dos.h 128: };
C:\TC\INCLUDE\dos.h 222: void cdecl ---cli--- (void);
mcalc.c 9:
    /* 开始处理 conio.h, 行号从 1 开始 */
C:\TC\INCLUDE\conio.h 1:
C:\TC\INCLUDE\conio.h 33: enum text --modes { LASTMODE=-1, BW40=0, C40,
                        BW80, C80, MONO=7 };
C:\TC\INCLUDE\conio.h 38: enum COLORS {
C:\TC\INCLUDE\conio.h 39: BLACK,
C:\TC\INCLUDE\conio.h 54: WHITE
C:\TC\INCLUDE\conio.h 55: };
C:\TC\INCLUDE\conio.h 60: extern int cdecl directvideo;
C:\TC\INCLUDE\conio.h 96:
mcalc.c 10:
    /* 开始处理 mcalc.h, 行号从 1 开始 */
mcalc.h 1:
mcalc.h 113: struct CELLREC
mcalc.h 114: {
mcalc.h 115: char attrib;
mcalc.h 116: union
mcalc.h 117: {
    /* 进行宏扩展: MAXINPUT 被 79 代替 */
mcalc.h 118: char text[79 + 1];
mcalc.h 119: double value;

```



```

mcalc.h 120; struct
mcalc.h 121; {
mcalc.h 122; double fvalue;
mcalc.h 123; char formula[79 + 1];
mcalc.h 124; } f;
mcalc.h 125; } v;
mcalc.h 126; };
mcalc.h 127;
mcalc.h 128; typedef struct CELLREC *CELLPTR;
mcalc.h 173; void initcursor(void);
mcalc.c 11;
/* 下面只保留与宏扩展相关的语句, 那些和源程序语句相同的一般被略去 */
mcalc.c 12; CELLPTR cell[100][100], curcell;
mcalc.c 13; unsigned char format[100][100];
mcalc.c 14; unsigned char colwidth[100];
mcalc.c 15; unsigned char colstart[(80 - 3) / 3 + 1];
mcalc.c 17; char changed = 0;
mcalc.c 18; char formdisplay = 0;
mcalc.c 19; char autocalc = 1;
mcalc.c 20; char stop = 0;
mcalc.c 27;
mcalc.c 28; void run()
mcalc.c 29;
mcalc.c 30; {
mcalc.c 31; int input;
mcalc.c 32;
mcalc.c 33; do
mcalc.c 34; {
mcalc.c 35; displaycell(curcol, currow, 1, 0);
mcalc.c 39; switch(input)
mcalc.c 40; {
mcalc.c 41; case '/':
mcalc.c 42; mainmenu();
mcalc.c 43; break;
mcalc.c 44; case 315;
mcalc.c 47; case 316;
mcalc.c 50; case 339;
mcalc.c 51; deletecell(curcol, currow, 1);
mcalc.c 56; case 329;
mcalc.c 67; displayscreen(0);
mcalc.c 69; case 337;
mcalc.c 72; if ((currow >= 100) && (toprow >= 100))
mcalc.c 73; {
mcalc.c 74; currow = 100 - 1;
mcalc.c 75; toprow = 100 - 20;
mcalc.c 76; }

```

```

mcalc.c 77; else if (toprow > (100 - 20))
mcalc.c 78; {
mcalc.c 79; currow -= (toprow + 20 - 100);
mcalc.c 80; toprow = 100 - 20;
mcalc.c 81; }
mcalc.c 83; displayscreen(0);
mcalc.c 85; case 371;
mcalc.c 86; displaycell(curcol, currow, 0, 0);
mcalc.c 94; displayscreen(0);
mcalc.c 97; case 372;
mcalc.c 98; displaycell(curcol, currow, 0, 0);
mcalc.c 99; if (rightcol == 100 - 1)
mcalc.c 106; displayscreen(0);
mcalc.c 109; case 327;
mcalc.c 113; displayscreen(0);
mcalc.c 115; case 335;
mcalc.c 121; displayscreen(0);
mcalc.c 123; case 328;
mcalc.c 126; case 336;
mcalc.c 129; case 331;
mcalc.c 132; case 333;
mcalc.c 141; while (!stop);
mcalc.c 142; }
mcalc.c 143;
mcalc.c 144; void main(int argc, char *argv[])
mcalc.c 145; {
mcalc.c 146; window(1, 1, 80, 25);
mcalc.c 150; setcolor(WHITE);
mcalc.c 151; clrscr();
mcalc.c 152; printf((80 - strlen("MICROCALC - A Turbo C Demonstration Program" )) >> 1,
11, LIGHTCYAN, strlen("MICROCALC - A Turbo C Demonstration Program"), mcalc.c 153; "MICRO-
CALC - A Turbo C Demonstration Program");
mcalc.c 154; printf((80 - strlen("Press any key to continue." )) >> 1, 13, YELLOW,
mcalc.c 155; strlen("Press any key to continue."), "Press any key to continue.");
mcalc.c 158; setcolor(WHITE);
/* 宏扩展为一个表达式 */
mcalc.c 161; memleft = coreleft() - 1000;
mcalc.c 162; redrawscreen();
mcalc.c 163; if (argc > 1)
mcalc.c 164; loadsheet(argv[1]);
mcalc.c 167; setcolor(LIGHTGRAY);
mcalc.c 169; setcursor(oldcursor);
mcalc.c 170; }

```

从列表文件 MCALC.I 中可以看出, 其内的包含文件 MCALC.H 中的条件编译语句
 #if defined(——TINY——) || defined(——SMALL——) || defined(——WEDIUN——
 —)

```
#else
#endif
#if !defined(MAIN)
#endif
```

已消失,同时, CPP 已根据这些语句将相关内容装入 MCALC. I。譬如,由于 MCALC. C 中第一个语句已定义了

```
#define MAIN
(虽然 MAIN 未被定义为具体内容),则源包含文件 MCALC. H 中
#if !defined(MAIN)
extern CELLPTR cell[MAXCOLS][MAXROWS], curcell;
extern unsigned char format[MAXCOLS][MAXROWS];
.....
extern unsigned int oldcursor, shortcursor, tallcursor, nocursor;
#endif
```

被 CPP 处理后没有任何东西装入 MCALC. I。

程序员从列表文件中可以清楚地到宏或包含文件扩展后的结果,凭此可以迅速对源程序中的宏定义错误进行确诊。

CPP 实际上提供其它编译程序中第一趟扫描的功能。此外,如果带上选择参数 -P-, 例如对文件 MCALC. C 使用 CPP

```
C>CPP -P- MCALC. C
```

结果在 CPP. EXE 所在目录产生文件 MCALC. I, 该文件的每一行已不再有行号(包括前面的文件名),并可以直接为 TC 或 TCC 编译(不过,由于它和其它几个源程序相关,所以对它单独编译会出现连接错误)。也即是说,此时 CPP 可作宏预处理器。但当列表文件中有行号时,其不能被编译。

从中还可以看出, C 源程序中可以多包含一些头文件,尽管它们包含的变量和函数也许不被使用,结果也不会发生编译错误。然而,编译程序将给它们分配内存,因此,除非必要,否则不要将无关的头包含文件装入源程序。

注意,对于一般的 C 程序编译,它是不需要的。或者说,它只是在必要时用来分析问题的一种辅助工具,而不是非用不可的。

二 CPP 的参数表

在 DOS 提示符下打入 CPP 后立即回车,屏幕上便显示 CPP 的命令行参数表。对 DOS 3.0 以上版本,键入

```
C>CPP>E:MYFILE
```

回车后则在 E 盘上产生一个内容为 CPP 参数表的 ASCII 文件 MYFILE,你可用 TC 编辑器将其装入并用块打印方法将其在打印机上打印出来。下面对其操作作简单说明。

Syntax: CPP [options] file[s]

语法: C>CPP [选择项][文件名]

说明:

1. 选择项

(1)符号说明

* = default; (下面带 * 者表示缺省值)

-x- = turn switch x off

所有选择项都可认为是一开关,选用时为开,否则为关。开关必须以一短横线开始;-x 表示开关为开,-x- 表示开关为关。

(2) 选择项说明 (参见第四十章对 TCC.EXE 相同命令选择项的说明)

-A	Disable non-ANSI extns.	禁止非 ANSI 扩展
-C	Allow nested comments	允许嵌套注释
-Dxxx	Define macro	定义宏
-Ixxx	Include files directory	包含文件目录
-P	* Include source line info	包含源行号 (缺省选择), 列表文件用源文件名 (或包含文件名) 加行号作为前缀。
-Uxxx	Undefine macro	取消宏
-gnnn	Stop after N warnings	在 N 个警告后停止
-innn	Maximum identifier length N	最大标识符长度为 N
-jnnn	Stop after N errors	在 N 个错误后停止
-mc	Compact model	紧凑存储模式
-mh	Huge Model	巨存储模式
-ml	Large Model	大存储模式
-mm	Medium Model	中存储模式
-ms	* Small Model	小存储模式
-mt	Tiny Model	微存储模式
-nxxx	Output file directory	输出文件目录
-oxxx	Output file name	输出文件名
-p	Pascal calls	PASCAL 调用
-w	Enable all warnings	允许所有警告
-wxxx	Enable warning xxx	允许警告 xxx

注意:它从 TURBO.CFG 文件中读入缺省的选择项。

2. 文件名

CPP 对文件名有两个要求:

(1)磁盘文件必须是 C 源文件 (ASCII 码文件);

(2)磁盘文件必须带有扩展名。

文件名允许使用 DOS 通配符 ? 和 *。例如,

```
C>CPP -ne; c:m*.c
```

与

```
C>CPP -ne; c:m???????c
```

都是合法的。

列表文件名一定带有扩展名 .l。

三 错误、警告及提示信息

(参见《编译和调试程序》一章中“错误、警告及提示信息”。)

第十二章 接收自变量个数可变的宏

在 `stdarg.h` 中定义了一些数据类型和宏：

```
typedef void * va_list;
#define va_start(ap, parmN) (ap=...)
#define va_arg(ap, type)    (*( (type *) (ap) )++)
#define va_end(ap)
#define __va_ptr            (...)
```

有一些函数,如 `vprintf()` 和 `vsprintf()` 等使用的参数个数不是固定的,就可用这些宏来测试可变参数表。

12.1 数据类型和宏

—1 `typedef void * va_list;`

定义了一个相当于 `void *` 的新数据类型 `va_list`,它专门说明指向可变参数表的指针。当调用函数使用一个可变参数表时,就可使用 `va_list`。例如,

```
va_list ap;
```

就把变量 `ap` 说明为一个指向可变参数表的指针。

`void *` 型指针是一个可以指向任何数据类型的指针,不过其所指对象的类型还不清楚。可以将任何指针不加任何转换地赋给 `void` 类型指针,反过来也可以将 `void` 类型指针赋给其它任何类型的指针。不过,对 `void` 型指针不允许使用间接运算符 `*`,因为其对象类型没有定义。

```
C>TYPE STDARG1.C
```

```
#include "stdarg.h"
```

```
main()
```

```
{
```

```
char s[]={'AB','\0'};
```

```
char *p=s;
```

```
char d=*p; /* 如将上句改成 va_list p=s; 则本句出现 Not an allowed type 错误;如再将本句改成 char d=*(char *)p; 则不会有问题了 */
```

```
printf("%c\n",d); /* 程序将输出字母 A */
```

```
}
```

—2 `#define __va_ptr (...)`

该宏定义为一个由三个连续小数点组成的省略号。在 Turbo C 中,该省略号则表示一个可变参数表。程序员也可在自己的程序中使用它,因为在某种意义上,这省略号已被当作这种固定意义在使用。我们将在 12.2 节中详细讨论它。

—3 `#define va_start(ap, parmN) (ap=...)`

在集成环境里按 F1 键后,可在帮助内容中看到

```
void va—start(ap,parmN);
```

这就是说,它不返回任何值。另一方面,从宏定义可以看出,是对指针 ap 赋予一个特殊的值,即省略号。该子程序(用宏实现)使 ap 指向可变参数表的第一个参数。在调用 va—arg() 之前应先调用它。从宏定义可以看出,parmN 纯粹起一个标志作用,暗示可变参数表在它之后,实际未用它。可用 ... 或 —va—ptr 代替它(其实可为任何标识符),也可以不写,但它前面的逗号却不能省略。

```
C>TYPE STDARG2.C
```

```
#include "stdarg.h" /* 此句可以不要,因为 stdio.h 中可包含它 */
#include "stdio.h"

void sum(char *msg,...) /* msg 后的逗号是不能少的 */
/* 注意,不能将上句写成 void sum(char *msg,—va—ptr) */
{
    int total=0,num=0;
    va—list ap;
    int arg;
    va—start(ap,msg);
    printf("%FP\n",ap);
    arg=va—arg(ap,int);
    num++;
    printf("加数是:");
    while(arg != 0) /* 如果可变参数的类型不一致,则不能用循环 */
    { /* 而只能逐个地调用 va—arg() 获取参数值 */
        printf("%d,",arg);
        total+=arg;
        arg=va—arg(ap,int); /* 可变参数类型都为 int (整型), */
        /* 若将它改成 long 便得不到正确结果 */
        num++;
    }
    printf("加数个数为:%d\n%s\n",num-1,msg);
    printf(msg,total);
    printf("%s\n",msg);
}

main()
{
    int start=1,end=80;
    sum("the total of 1+2+3+4+7+80 is %d\n",start,2,3,4,7,end,0);
    /* 可变参数全为整型,如有一个非整型便得不到正确值,且可变参数必须以 */
    /* 数字 0 结尾,否则也不能得正确结果,这是 sum() 中 while() 的要求 */
    printf("End ! \n");
}

/* 程序输出:
0050.01BD
加数是:1,2,3,4,7,80,加数个数为:6
```

the total of 1+2+3+4+7+80 is %d

the total of 1+2+3+4+7+80 is 97

the total of 1+2+3+4+7+80 is %d

End !

由此可知 msg 相当于串:"the total of 1+2+3+4+7+80 is %d\n",而 ... 相当于 可变参数
start,2,3,4,7,end,0。讲是可变参数,就是说,你可以随意增减参数 个数或改变它们的值。 */

—4 #define va—arg(va—list ap,type) (* ((type *) (ap))++)

从宏定义中可以看出,它相当于

* (type)ap++;

即每调用一次,它将获得 ap 当前所指可变参数表中的参数值,随后将 ap 加 1。注意,对于像
va—list 指针不能进行 ap++ 运算,但用 (type) 强制指定类型转换后是可以的,这里 type 是
数据类型。type 是根据实参的数据类型决定的。

在集成环境里,用 F1 键操作可得帮助信息

type va—arg(va—list ap,type);

这就是说,其返回值的类型同圆括号内的 type。在调用它之前,一定要先调用 va—start(),以
便对 ap 赋值,否则结果是不可预料的。

C>TYPE STDARG3.C

#include "stdio.h"

void ff(char *m,double *ft,...)

{

va—list a;

double *k;

double total=0.0,f=10.0,h;

k=va—start(a,ft);

h=*(double *)a;

/* 不能将此句写成 h=*a; ,那样会出现 */

/* Not an allowed type 的错误 */

f=va—arg(a,double);

h=*(double *)a;

/* 可用调试表达式 h 观察 a */

while(f!=0.0)

{total+=f;

f=va—arg(a,double);

}

printf(m,total);

va—end(a);

}

main()

{

double fs=1.2,gl=1.3;

double *ft;

ft=&fs;

```
ff("%f\n",ft,fs,g1,1.5,0.0); /* 最后一个数 0.0 不能写成 0 */
}
/* 在 LX-386/33S 机上,这个程序中的 double 如改成 float 虽能编译通过,*/
/* 但不能得到正确结果。可能在不同的机器上有不同的结论 */
```

——5 #define va—end(ap)

在集成环境里,用 F1 键操作可得帮助信息

```
void va—end(va—list ap);
```

表示不返回值。对一个含有 va—end() 宏 (注意:如使用宏,则圆括号内必须写入一个标识符) 的程序使用 cpp.exe 扩展,不难发现该宏被扩展为一个空语句,即只有一个冒号的语句。单步调试时可以发现它不是执行语句。因此,在大多数情况下,它对指针 ap 不起作用 (至多起一个标识作用)。然而,使用这个宏有时能使堆栈正确复位,以避免不可预料的情况出现。例如,由于 Turbo C 改变了标号 (label) 语法,规定标号后面一定要跟有一个语句,或者说,像

```
{
.....
jump—label;
}
```

这样的复合语句是有问题的,正确的写法应是在标号后面加上一个空语句:

```
{
.....
jump—label; /* 标号 */
; /* 这个空语句是必须的 */
}
```

通常它应在 va—arg() 读完所有可变参数后才调用。

C>TYPE STDARG4.C

```
/* 不能在下句前定义宏 ——STDARG,否则将取消 stdio.h 包含 stdarg.h */
#include "stdio.h"
```

```
void err(char *format,int *p,...)
{
int s1;
va—list aptr;
p=&s1;
printf("%Fp\n",p);
va—start(aptr,p); /* 即使改成 va—start(aptr,format) 也一样 */
vprintf(format,aptr);
va—end(aptr); /* 此句是空语句,表明结束可变参数的接收 */
vprintf(format,aptr);
}
```

```
main()
{
```



```

int v1=-1,v2=-2;
int *y;
y=&v1;
printf("%Fp\n",y);
err("这是一个错误\n",y);
err("无效值%d,%d\n",y,v1,v2);
    /* 不能把此句改成:err("无效值%Fp %d,%d\n",y,v1,v2); */
    /* 因为指针 y 不属于可变参数表,且不是整型数。 */
}
/* 程序输出:04ED,FFE0
           FFE0,FFD2
           这是一个错误
           这是一个错误
           FFE0,FFCE
           无效值-1,-2
           无效值-1,-2
*/

```

C>TYPE STDARG5.C

```
#include "stdio.h"
```

```
main()
```

```

{
double d,sum(); /* 使用传统方法说明函数原型,Turbo C 对参数不会检查 */
d=sum(5,25.0,100,0.2,0.1,0.3125,0.2,0.2);
printf("和数=%f\n",d);
}

```

```
double sum(int num,double y,int x)
```

```

    /* 把此句写成 double sum(int num,...); 或 double sum(int num); */
    /* 并把 d=sum(5,25.0,100,0.2,0.1,0.3125,0.2,0.2); 改成 */
    /*      d=sum(5,0.2,0.1,0.3125,0.2,0.2); 也可得到同样结果 */
    /* 但 sum() 至少应有一个已知的固定的参数,否则不会得到正确结果 */

```

```

{
int num1=num;
double total=0.0,t;
va_list aptr;
va_start(aptr,dd);
for(,num;num--)
{
    t=va_arg(aptr,double);
    va_end(aptr); /* 此句在此不起作用 */
    total += t;
}
printf("num=%d, y=%f, x=%d\n",num1,y,x);
return total;
}

```

/* 程序输出:

num=5, y=25.000000, x=100

和数=1.012500

*/

12.2 Turbo C 函数的特殊参数“...”的用法

Turbo C 中有一个由三个小数点组成的符号“...”(简称为【省略号】),它并不是 C 规定可使用的标识符,因为标识符是由字母或下划线开始,后跟字母、数字或下划线的正文串。当然,它也不是关键字(在帮助文件中也没有省略号的任何帮助信息)。不过,它常在 Turbo C 的库函数(像 `ioctl()` 等)的参数表中出现。换言之,它可以作为函数的参数(表)。从下面的叙述中可以看出,它是一种较为特殊的“参数”,程序员在使用它时应注意其独特的使用方法。

1. 省略号的基本含义

它包含两层意思,一是表示隐含有若干个参数,但参数的确切个数并不知道;其次,每个参数的类型也不清楚。换句话说,它相当于一个【可变参数表】。可变参数表中的参数可多可少,甚至一个没有也是允许的。

类型为 pascal 的函数不允许有可变参数表,因此在这种函数中不允许使用省略号。

```
C>TYPE STDARG6.C
```

```
#include "stdio.h"
```

```
main()
```

```
{
```

```
int x=100;
```

```
char *s="%d";
```

```
pp(s,x);
```

```
}
```

```
pp(char *f,...) /* 如将此句改成 pascal pp(char *f,...) */
```

```
{
```

```
/* 则出现 Conflicting type modifiers 错误 */
```

```
va_list p;
```

```
va_start(p,f);
```

```
printf(f,va_arg(p,int));
```

```
}
```

2. 省略号的特殊性

(1) 它只能出现在现代格式的函数定义中,而不能用在传统格式的函数定义中。如

```
void getmy(char *s,...){}
```

是可以的,而

```
void getmy(s,...)
```

```
char *s;
```

```
{}
```

则是不允许的。事实上,传统格式也无法说明这种数据类型和数据个数都没有确定的“参数”。

(2) 省略号与函数定义的关系

省略号可以出现在参数表中,但函数体中可以不涉及它,即与它无关。这在程序员刚开始

拟订某函数时也许是有用的。例如，

```
fun(int y,...)
{ return (y * y * y); }
```

在编译时不会出错。

(3)涉及省略号的函数定义与函数原型的关系

如果使用了现代格式的函数原型，Turbo C 将把省略号作为显式参数对待，进行匹配检查。如在同一个源文件中有

```
void fun(int y,...); /* 原型 */
.....
void fun(int y){} /* 定义 */
```

或

```
void fun(int y);
.....
void fun(int y,...){}
```

都是不正确的。而当原型使用传统格式时，将不对参数表进行检查。

(4)有省略号的函数至少要有有一个显式参数，且唯一使用的省略号只能在最后一个参数位置上。例如，

```
void fun(...);
void fun(..., int y);
void fun(int y, ..., char c);
void fun(int y, ..., char c, ...);
void fun(int y, ..., ...);
```

都是错误的。

(5)省略号不能出现在调试表达式中
虽然，像

```
void fun(char *f, char c, ...)
{
    int y;
    .....
    y=sizeof(...);
    .....
}
```

是可以的，对小 (Small) 存储模式， $y=2$ ；对大 (Large) 存储模式， $y=4$ 。然而不允许有像 `sizeof(...)` 这样的调试表达式出现在监视窗内。如省略号在调试表达式内，将出现像

Function doesn't take a variable number of arguments

的错误提示。

3. 利用省略号取参数值的方法

一般分三个步骤：定义一个 `va-list` 指针、使该指针指向省略号及利用指针逐个取可变参

数表中的参数值。

(1)在函数内用

```
#include "stdarg.h"
va_list ap;
```

先定义一个 va_list (即 void) 型指针 ap。虽然将 ap 定义成整型 (int) 也是可以的,但定义成其它类型 (char 等) 将被认为值得怀疑的,因此,建议只定义成 va_list。

(2)利用

```
va_start(ap,);
```

使指针 ap 指向可变参数表的第一个参数。

C>TYPE STDARG7.C

```
#include "stdio.h"
int *px, *pz;          /* 定义全程指针,以便利观察变量 x 和 z 的地址 */
main()                 /* 从本程序清楚地看到 ap 所在地址并非原实变量地址 */
{                      /* 因此可变参数表的地址可以认为是临时分配的 */
    int x=100,z=2;
    char *s="%d %d";
    px=&x;
    pz=&z;
    pp(s,x,z);
}
pp(char *f,...)
{
    int y,z;
    int *ap;            /* 为方便用调试表达式观察,将 ap 定义成整型 */
    va_start(ap,f);     /* 此句执行后有 ap:DS:FFD6, *ap: 100 */
    y=va_arg(ap,int);   /* px: DS:FFDC, *px: 100 */
    z=va_arg(ap,int);   /* pz: DS:FFDE, *pz: 2 */
    printf(f,y,z);      /* ap[0],5: 100, 2, 92, 100, 2 */
}
```

虽然从程序 STDARG7.C 看到可变参数表在内存中占有一些连续的单元,并从宏定义

```
#define va_start(ap,paramN) (ap=...)
```

看,省略号似乎也可以看成为一个指针,但是它并不是指针。例如,像

```
...++;
```

这样的式子是无意义的。

(3)利用

```
va_arg(ap,type);
```

获取可变参数表中的一个参数值。从该宏的定义

```
#define va_arg(ap,type) (*((type *) (ap)++)++)
```

可以看出,它每被调用一次,ap 自动增 1,即指向可变参数表的后一参数。由于可变参数是连续顺序存放的,所以如果要断续取可变参数表中的值,或者说要跳过那些暂时不用的参数

值,也应调用它,不过调用获得的值可以不赋给具体的变量而已。这种调用可称为空调用,目的只是让指针 ap 增值,指向后一参数。

定义中 type 表示数据类型,它们可是 char、int、long、float、double 等(也可以在它们前面加上 unsigned),还可以是 char *、struct 结构名、union 联合名等。但是要注意的是,不要乱加括号。例如,将 char * 写成 (char *) 是错误的。

C>TYPE STDARG8.C

```
#include "stdio.h"
#include "stdarg.h"
main(){ /* 可变参数表中有指针时 */
char *d="ABC";
char *h="%s\n";
in(h,d);
}
in(char *f,...)
{
char *u=(char *)malloc(80);
va—list ap;
int y;
va—start(ap,);
u=va—arg(ap,char *);
y=sizeof(...);
printf(f,u);
}
```

C>TYPE STDARG9.C

```
#include "stdarg.h"
main(){ /* 可变参数表中有结构时 */
struct { /* 结构 d 实际要占用 4 个字节 */
char c;
int x;
}d={'e',8};
int r=2;
char *h="%c,%d\n";
in(h,d,r);
}
in(char *f,...)
{
struct aa{
char cc;
int xx;
}u;
int q;
va—list ap;
va—start(ap,);
u=va—arg(ap, struct aa); /* 注意这里 type 的写法 */
printf(f,u.cc,u.xx); /* 取出整个结构,而不是逐个取结构的元素 */
}
```

```

q=va-arg(ap,int);      /* 取下一个参数值          */
printf("%d\n",q);
}

```

/* 在集成环境里必须使用 O/C/C/Alignment Word 编译执行,如用
O/C/C/Alignment Byte 则会出错。原因是结构采用字对齐。 */

4. 调用含省略号函数时实参与形参之间的一般规约

C>TYPE STDARG10.C

```

#include "stdio.h"
#include "stdarg.h"
main(){
int m=1;
double n=2;
char *s="%f\n";
fun(s,m);      /* ← 这两句可能 */
fun(s,m,n);    /* 出问题,因 */
}              /* 为 v 的值可 */
fun(char *f,...) /* 能是不可预 */
{              /* 料的! */
int u;        /* */
double v;     /* */
va-list ap;   /* */
va-start(ap,); /* */
u=va-arg(ap,int); /* */
v=va-arg(ap,double); /* ← */
printf(f,u+v);
}

```

该程序中由于 fun(s,m); (暂称为“实参语句”)中只用了一个可变实参,它表明可变参数表也只有一个参数,所以对 v=va-arg(ap,double); 语句 (暂称为“形参语句”),指针 ap 所指第二个参数是不存在的,因而参数值是不可预料的。这种程序尽管在调试程序时能得到某些正确的结果,但以后实际运行时又可能变得不可靠。

程序 STDARG10.C 提出了一个很重要的问题,即使用 va-arg() 取可变参数值时,实参与形参之间除了传值单向性外,还应有某种规约。为方便叙述,下面称形参所在函数 fun() 为被调用函数,称实参所在函数 main() 为调用函数。同时我们还认为,被调用函数参数表由显式参数和可变参数表组成,相应地称实参语句包括显式实参和可变实参。

(1) 显式实参的个数、数据类型应与显式形参对应。或者说,显式实参是不可缺少的。

(2) 设调用函数中 k 个实参语句所含可变实参的个数分别为 n_1, n_2, \dots, n_k , 则被调用函数中调用 va-arg() 的次数不得多于 $\min(n_1, n_2, \dots, n_k)$ (即 k 个数的最小值)。或者说,设在被调用函数中 va-arg() 调用次数为 n, 则在调用函数中每个实参语句中可变实参的个数至少应为 n 个。进一步说,在被调用函数里,甚至一次也未用 va-arg() 调用可变参数中的值也是可以的。

C>TYPE STDARG11.C

```

#include "stdio.h"
void fun(int y,...)

```

```

{int u1,u2;
double v1,v2,v3,s;
va—list ap;
va—start(ap,);
u1=va—arg(ap,int);      /* 共计调用 va—arg() 4 次 */
v2=va—arg(ap,double);
va—arg(ap,double);      /* 空调用一次          */
v3=va—arg(ap,double);
s=y+u1+v2+v3;
printf("%f\\t",s);
}
main()      /* 本程序可变实参个数等于 va—arg() 调用次数 */
{
int m1=100,m2=12;
double n1=0,n2=8.8,n3=0.2;
fun(m1,m2,n2,n1,n3);
fun(m1+m2,m1,n2,n3,n2*2+n3); /* 可变实参可以是表达式 */
}/* 程序输出:121.000000    238.600000          */

C>TYPE STDARG12.C
#include "stdio.h"
void fun(int xx,...)
{
printf("%d\\n",xx);
}
main()      /* 本程序可变实参个数大于 va—arg() 调用次数 */
{
int x=20,y=10;
fun(x,y);    /* 可变实参 y 未被使用,Turbo C 对此不作检查 */
}

```

(3)va—arg(ap,type) 语句中的数据类型 type 必须和实参语句中对应实参的数据类型完全一致。Turbo C 并不对此作检查。

当实参为常量时尤其要注意。例如,实参为 100,而 type 为 double,或者实参为 100.0,而 type 为 int 等,都会引起错误。

5. 在被调用函数中确定可变实参个数的方法

从上可见,在被调用函数中如果能事先知道可变实参的个数,也就知道了 va—arg() 可调用的次数;或者,可以利用条件语句灵活处理调用 va—arg() 的次数,因而就可使编程更灵活有效。例如,在 BASIC 中有一个取子串函数

MID\$(X\$,n,[m])

程序员可以随意书写其两种形式中的一种:

MID\$(X\$,n) 或 MID\$(X\$,n,m)

程序员自然希望用省略号来编写一个能同时接收这两种形式的函数,然而这是困难的。这是因为按照规约,n 和 m 是两个整数,被调用函数至少要调用一次 va—arg(),从前面的例子可以看出,处理 MID\$(X\$,n) 时将遇到麻烦。

可以用某个显式实参的值来确定调用 `va_arg()` 的次数或某次调用的有效性。例如：库函数

```
—chmod(const char * path, int func, ... /* int attr */);
```

第三个可变形参 `attr` 只有当 `func` 值为 1 时才要用到,而当其为 0 时由于 `attr` 值不用所以可以任意。所以这样的函数允许用户随意书写 `—chmod(path,func)` 或 `—chmod(path,func,attr)`。

由此可知,为了能在被调用函数中知道可变实参的个数,可以用一个可变实参的值决定后面的可变实参的出现。目前常用的一个方法是增加一个可变实参,用它记录可变实参个数或者作为一个(非有效可变实参)标志。被调用函数根据这一标志再作出相应反映。这种仅作为某种判别标志的可变实参在可变实参表中一般是最后一个或第一个。例如,`spaw...()` 类函数中常规定最后一个可变实参为 `NULL` 就是这个道理。

用省略号定义类似上述 BASIC 取子串函数:

```
char * MID(char * x,int n, ... /* int m */) {}
```

从上述分析可以看出,要在不增加可变实参(或显式实参)的情况下是困难的。下面我们给出一个利用宏定义的实现方法,不过实参语句应用形式为

```
((MID(x,n));
```

或

```
((MID(x,n,m));
```

```
C>TYPE STDARG13.C
```

```
#include "stdio.h"
#include "string.h"
#include "mem.h"
#include "alloc.h"
#include "ctype.h"
#include "stdlib.h"
#define MAXC 254
#define MID(F) strset(str,'\0');strset(snew,'\0');strcpy(str,#F);\
    len=strlen(str);j1=j2=k=w=0;\
    strset(t1,'\0');strset(t2,'\0');\
    while(w<len) { if( str[w]=='') k++; \
        if(k==1 && isdigit(str[w])) t1[j1++] = str[w]; \
        if(k==2 && isdigit(str[w])) \
            t2[j2++] = str[w]; w++; } \
    if(isdigit(t1[0])) n=atoi(t1); \
    if(isdigit(t2[0])) m=atoi(t2); \
    if (k==1) snew=strMID(sold,n,strlen(sold)); \
    else snew=strMID(sold,n,m)
```

```
char * strMID(char * x,unsigned char n,...)
```

```
main()
```

```
{
```

```
int w,n=1,m=1,k,len,j1,j2;
```



```

char str[10],t1[5],t2[5];
char sold[]="qwertyASDFG";
char *snew=(char *)calloc(MAXC,1);
MID((sold,n));
printf("%s\n",snew);
MID((sold,n,m));
MID((sold,n,2));
MID((sold,3,4));
}
char *strMID(char *x,unsigned char n,...)
{
char v[MAXC];          /* 不要用 static char v[MAXC]; 那样语句清内存单元 */
int j,len=strlen(x);
va_list ap;
unsigned char m;
va_start(ap,n);
m=va_arg(ap,unsigned char);
setmem(v,255,'\0');          /* 清内存单元 */
/* strnset(v,'\0',255);注意:此句只能保证第一个字符为 '\0',
而不能保证所有的存储单元为 '\0'。这是要注意的。 */
if(m==0 || *x==0 || n>len || n<1)return (v);
if(m>(len-n+1))m=len-n+1;
x+=n-1;
strncpy(v,x,m);
len=strlen(x);
/* if (len>=m)v[m]='\0'; 此句只在用 strnset(v,'\0',255);时才用
以保证当 len>=m 时串以空字符结束 */
return (v);
}

#if 0

main()          /* 用固定形式 strMID(x,n,m) 调用 */
{
char *str=(char *)calloc(80,1),*v;
unsigned char n,m;
printf("输入一个串:");
gets(str);          /* 输入:ASDFGHJK */
printf("输入开始 n,串长 m:");
scanf("%u,%u",&n,&m);          /* 输入:2,3 */
v=strMID(str,n,m);
printf("%s\n",v);          /* 输出:SDF */
}

```

```
#endif
```

6. 利用传统格式定义有可变参数的函数时省略号可不出现

前面我们已经讲到用传统格式定义带可变参数的函数时不能用省略号,实际上在这类函数中不写省略号也照样使用可变参数,因为此时 Turbo C 并不对参数检查。当然,由于其可读性差,建议你最好采用现代格式定义。

```
C>TYPE STDARG14.C
```

```
#include "stdio.h"

void fun()          /* 传统格式函数原型说明 */
main()
{
    enum a{W1,W2}xa;
    int x=20,y=10;
    fun(x,xa,y);
}

void fun(int xx)    /* 定义中未出现省略号 */
{
    int yy;
    enum b{K1,K2}xb;
    va_list ap;
    va_start(ap,);
    xb=va_arg(ap,enum b);
    printf("%d,%d,%d,",xx,K1,K2);
    yy=va_arg(ap,int);
    printf("%d\n",yy);
}/* 程序输出:20,0,1,10 */
```

12.3 应用实例

```
C>TYPE STDARG15.C
```

```
#include <stdio.h>
#include <mem.h>
#include <conio.h>
static unsigned char colortable[256];
void initcolortable(void) /* 初始化颜色表 colortable */
{
    int color, fg, bg, fcolor, bcolor;
    struct text_info ti;
    gettextinfo(&ti);
    if (ti.currmode == C80) /* 如果当前文本模式是 C80: 彩色 80 列 */
    {
        for (color = 0; color <= 255; color++)
            colortable[color] = color;
    }
    else
```

```

(
    /* 其它模式: BW40, C40, BW80, MONO 等 */
    for (fg = BLACK; fg <= WHITE; fg++) /* fcolor 用于前景 */
    {
        if (fg == BLACK) fcolor = BLACK;
        else if (fg <= LIGHTGRAY) fcolor = LIGHTGRAY;
        else fcolor = WHITE;
        for (bg = BLACK; bg <= LIGHTGRAY; bg++) /* bcolor 用于背景 */
        {
            if (bg == BLACK) bcolor = BLACK;
            else
            {
                if (fcolor == WHITE) fcolor = BLACK;
                bcolor = LIGHTGRAY;
            }
            colortable[fg + (bg << 4)] = fcolor + (bcolor << 4);
        }
    }

    for (fg = 128; fg <= 255; fg++)
        colortable[fg] = colortable[fg - 128] | 0x80;
}

void setcolor(int color) /* 用颜色表设置当前颜色 */
{
    /* 在文件 graphics.h 中有此同名函数, 但本程序未将该文件包含 */
    textattr(colortable[color]); /* 此函数原型在 conio.h 中 */
}

void writef(int col, int row, int color, int width, va-list arg-list, ...)
{
    va-list arg_ptr; /* arg_ptr 是一个 void 型指针 */
    char *format;
    char output[81];
    int len;

    /* 在访问任何可变长度参数前, 参数指针必须通过 va-start() 初始化 */
    va-start(arg_ptr, arg-list);
    format = arg-list;
    vsprintf(output, format, arg_ptr); /* 将格式内容输入 output 数组中 */
    /* 此句改成 vsprintf(output, format, ...); 也一样 */
    output[width] = 0; /* 该元素置 '\0', 以便输出 */
    if ((len = strlen(output)) < width) /* width-len = 无字符个数 */
        setmem(&output[len], width - len, ' '); /* 将无字符单元用空格代替 */
    setcolor(color); /* 设置颜色 */
    gotoxy(col, row); /* 移动光标 */
    cprintf(output); /* 输出串到屏幕 */
}

main() /* 本程序列举了一个未用 graphics.h 的非常有用的初始化颜色表函数 */

```

```

{ /* 引用函数可参见 Turbo C 的 MCDISPLY.C 等源程序 */
int x=1,y=4;
initcolortable();          /* 初始化颜色表 colortable */
clrscr();                  /* 清文本屏幕 */
writef(x, y, 3, 24, "从第%d行第%d列开始显示 ok! ",y,x);
                           /* 调用函数:3 是颜色数,24 是显示字符宽度 */
getch();
}/* 程序用兰色输出:从第 4 行第 1 列开始显示 ok! */

```

第十三章 函数

一些高级语言常有变量、函数、过程、子程序和主程序这些逐级渐进的概念,而 TC 像其它 C 语言一样,只有变量、函数(子程序被称为函数)。一个程序中有一个且只有一个名为 main 的【主函数】,由它开始调用其它函数,其它函数之间可以相互调用。同一个函数可被别的函数调用任意次数。

一个 C 程序的执行总是从主函数开始,然后由它调用其它函数,最后又返回到主函数,在主函数终结运行。

13.1 函数类型标识符

格式:

[static|extern][□函数返回的数据类型][□near|□far|□huge]
[□interrupt][□cdecl|□pascal]

例如:定义函数 power:

```
static double near pascal power(double x,int exp)
```

中的 static double near pascal 便是函数 power 的类型标识符。

其中只有数据类型时表明还有缺省值是:extern, near 和 cdecl。如果连数据类型也省略,则指数据类型为 int。

1. static 和 extern 用于区分内部函数和外部函数;
2. 函数返回的数据类型:三种 int 型(int、short int 和 long int)、char、float、double、long double,还可在它前面添加:unsigned(无符号型,即只存在非负值)。

另外还有正文(text),但 text 不是专门的修饰符。正文的返回是通过字符数组或指向字符的指针进行的。

3. near 和 far 区别于近调用和远调用;
4. interrupt 修饰符

它是 TC 特有的修饰符,需和 8088/8086 中断向量联合使用。TC 在编译 interrupt 函数时,将产生附加的函数入口和出口代码以保护寄存器 AX、BX、CX、DX、SI、DI、ES 和 DS。其它的寄存器 SP、BP、SS 与 IP 作为 C 调用序列的一部分或作为中断处理的一部分被保护。

5. pascal 修饰符

只有当函数参数有确定的个数时才能用(即不能在参数中使用省略号“...”)。它为 TC 特有,用于指定使用 Pascal 参数传递序列的函数(和指向函数的指针)。这允许从用其它语言写的程序中调用 C 函数。同样,也允许 C 程序调用其它语言所书写的外部子程序。连接时函数名全部转换为大写。

使用编译选择项 -p 或在集成环境中选

Options/Compiler/Code generation/Calling convention Pascal

时将使所有的函数（和指向这些函数的指针）全当作 Pascal 类型函数处理。如果有

```
C>TYPE F1.PRJ
```

```
f1.c
```

```
f2.c
```

```
C>TYPE F1.C
```

```
pascal putn(int i)
```

```
{printf("%d\n",i);}
```

```
C>TYPE F2.C
```

```
pascal putn(int i); /* 说明 putn 为 pascal 类型函数 */
```

```
main()
```

```
{putn(100);}
```

则 F2.C 便可调用 F1.C 中定义的 pascal 函数,因为 F2.C 中已对它作了说明。在未使用编译选择项 -p 或在集成环境中选

Options/Compiler/Code generation/Calling convention C

时,对 F1.C 或 F2.C 中只要一方用了 pascal,而另一方未用,则此函数便不能调用。

6. cdecl 修饰符

它也是 TC 特有的标识符,主要用于函数和指针,目的是不受编译选择项 -p 的约束,把调用 C 函数当作为正规的 C 函数。

在集成环境里,采用 small 存储模式及

Options/Compiler/Code generation/Calling convention Pascal

在输入调试表达式后,用 F7 热键对规划文件 F2.PRJ 进行调试。

```
C>TYPE F2.PRJ /* 规划文件名 */
```

```
f3.c
```

```
f4.c
```

```
C>TYPE F3.C
```

```
cdecl printf(); /* 此句不用 cdecl,显示 Undefined symbol 'PRINTF' 的错误 */
```

```
putn(int i) /* 函数 putn 未用 cdecl 修饰,故将按 Pascal 规约编译 */
```

```
{
```

```
int x=1;
```

```
printf("x=%d\n",x); /* 调试到函数右 } 后,有调试表达式值: */
```

```
printf("%d\n",i); /* printf,p: CS:0ADC */
```

```
} /* putn,p: CS:01FA */
```

```
C>TYPE F4.C
```

```
putn(int i); /* 用 F7 热键调试此句后便显示调试表达式值: */
```

```
cdecl main() /* printf,p: CS:0ADC printf */
```

```
{putn(100);} /* putn,p: CS:01FA PUTN */
```

从调试表达式的值我们清楚地看到函数名在 Pascal 情况下的大写、无下划线的实例。特别指出,对主函数 main 也应给出 cdecl,否则会出现

Undefined symbol '—main' in modules C0S

的错误。这说明,函数名 main 在 C0S 模块中是按正规 C 书写的。另外,C 启动运行码总是按

照 C 参数传递方法调用 main。

7. interrupt 修饰符

它也是 TC 特有的,它需和 8088/8086 中断向量联合使用。

例如:

```
static void interrupt (*old—int10h)(void); /* 设置中断指针函数原型 */
disable();
setvect(0x10, old—int10h);
```

13.2 函数说明和函数原型

TC 支持函数原型。函数原型也是一种说明。函数原型应在函数调用之前出现。有了函数原型就可对函数的参数及类型等作检查。

对同一个函数的说明可以用多种函数原型表示。例如,说明求两数极大值的函数原型可以是以下几种形式之一:

```
max()                /* 传统格式,编译时不对参数作检查 */
max(int, int);        /* 现代格式,编译时要对参数作检查 */
max(int x, int y);
```

1. 两种函数说明

(1) 传统(或称古典)的格式

函数类型标识符 □ 函数名 ();

注意:其结尾有分号,而函数定义则无分号。

例如:

```
float get—ratio();
void put—ratio();
```

这种说明由于未给出函数的参数信息,因而不容易检查错误,从而会引起一些非常细微及难于跟踪的错误。最好不用这种格式。

(2) 现代格式

函数类型标识符 □ 函数名 (参数表的数据类型 [参数名]);

参数表中多个参数间用逗号隔开(注意:也可以一个参数没有,或有类型 void)。例如,

```
void mm(void);
书写参数表有两种形式,如
float get—f(float, double);
```

或

```
float get—f(float x, double y);
```

换言之,对每一个形式参数,可以只指出其数据类型,也可以再给它起一个名字。如果函数要接受不定量的参数,可以将省略号 (...) 作其最后一个参数。例如标头文件的库函数中有一些就带此参数。

```
int —Cdecl fprintf (FILE *stream, const char *format, ...);
```

```

int  —Cdecl  fscanf    (FILE * stream, const char * format, ...);
int  —Cdecl  printf    (const char * format, ...);
int  —Cdecl  scanf     (const char * format, ...);
int  —Cdecl  sprintf   (char * buffer, const char * format, ...);
int  —Cdecl  sscanf    (const char * buffer, const char * format, ...);
int  —Cdecl  exec... (char * path, char * arg0, ...);
int  —Cdecl  spawn... (int mode, char * path, char * arg0, ...);
int  —Cdecl  chmod     (const char * path, int func, ...);
int  —Cdecl  ioctl     (int handle, int func, ...);
int  —Cdecl  open      (const char * path, int access, ...);
int  —Cdecl  eprintf   (const char * format, ...);
int  —Cdecl  cscanf    (const char * format, ...);

```

省略号 (...) 表示接受任意多个参数,但参数的类型并不清楚。

现代格式是值得推荐的,因为它允许编译器在实际调用函数时,检查参数的数目和类型,并保证函数的说明和定义相一致。还允许编译器在可能时对参数执行相应的类型转换。

2. 什么时候不需用函数原型

(1) 一个函数的返回值如果是整型 (int) 或字符型 (char) 则可以不要函数说明,只要这个函数在文件中有定义,而不管函数定义在调用前或调用后出现。

(2) 如果函数定义出现在该函数调用之前,便可不要另外的函数说明。因为这时编译程序已知此函数细节,或者说已相当于既定义了函数,又说明了函数,因而此时编译程序将对函数作检查,即履行函数说明的功能。

3. 函数说明在程序中的位置

原则上讲函数说明应在函数调用之前,但习惯上常采用两种方法:

(1) 将所有要说明的函数原型放在主函数前面,甚至放到文件的开头。这样的好处是清楚。

(2) 另一种是用包含文件,即将常用函数原型放到一个头文件 (*.h) 中,然后把该头文件包含到程序中来。

4. 标头文件和函数原型

TC 库函数是 TC 的标准函数,无须用户另行定义,只要程序中将其所在的标头文件包含到程序里便可调用它。

```

C>TYPE F5.C
#include "math.h"
main(){
double x;
x=pow10(2.3);
}

```

输出结果为 100。因为函数 pow10() 的原型为

```
double pow10(int p);
```

即函数参数是整形,而实参为 2.3,传给形参时形参获值 2,而不是 2.3。如果去掉包含文件语句,则 x 就不能得到正确值。

5. 用户自定义函数原型与标头文件中函数原型的关系

对用户自定义函数也可以用函数原型,或者将用户常用的函数原型收集于一个头文件

中,当要应用到其中的函数时,将该头文件包含到源文件中(使用 #include 语句)。这样,编译时将对调用函数进行检查,以避免出现差错。

当自定义的函数与标头文件中函数(即 TC 的库函数)同名时是危险的,因为这时在源程序中定义的函数将优先被执行(即原库函数的功能将不会实行),稍有疏忽,便会出问题(当然,当自定义函数和库函数完全等效时是不会有问题的)。当你在源程序中对库函数重定义的同时还包含了库函数原型所在的标头文件时,TC 将对定义函数类型及参数等进行检查。下面以 stdio.h 中的 printf 库函数为例来说明。

```
C>TYPE F6.C
main()
{int y;
y=printf(2);
printf("y=%d\n",y);
}
printf(int x)
{return x * x;}
```

此程序编译时不发生错误信息,用 F7 热键单步调试时再次进入用户定义的 printf() 函数。但是结果在屏幕上什么也没有输出。

```
C>TYPE F7.C
printf(int x)
{return x * x;}
main()
{int y;
y=printf(2);
printf("y=%d\n",y);
}
```

对语句 printf("y=%d\n",y); 出现编译错误:

```
Type mismatch in parameter 'x' in call to 'printf' in function main
Extra parameter in call to printf in function main
```

这时因为函数定义在调用之前相当于作了函数说明。

而如将上述两个程序前加上包含语句

```
#include "stdio.h"      /* 库函数 printf() 原型所在的包含文件 */
```

则都将出现

```
Type mosmatch in redeclaration of 'printf'
Non-portable pointer conversion in function main
```

那样的错误和警告。

从上述可见包含文件中的函数原型优于用户自定义的函数说明。用户在自定义一个函数时,除了要使函数具有固定的功能、参数明确、尽量不用全局变量和返回值清楚外,对函数名的命名也要注意。为防止这类错误出现,可以先将光标移到自定义函数名上,然后按 Ctrl-F1 键。如果该标识符是一个库函数名,则对应该函数的有关信息立即会显示到屏幕上。藉此,你可以经常检查自定义函数名是否与库函数名发生冲突。另一方法是,将自定义函数名中加上一个大写字母,因为一般库函数名中是没有大写字母出现的。

6. 在函数内的函数原型的作用域

只能在所在的函数内部是可见的（即可以起作用）。

在任何函数（包括 main 主函数）之外说明的函数从说明点起便是全局的，也就是说，它们可以被在它们之后的同一程序中任一函数所调用。

13.3 函数的定义

一个源程序中至少有一个函数（main）被定义。除库函数不需定义外，其它函数必须加以定义。函数一旦被定义，函数本身的实际代码就存在了。

对 TC 而言，定义函数可采用两种方式：传统（或称古典）的和现代的。程序员应优选现代的。

1. 现代的定义格式：

```
函数类型标识符 函数名(函数参数说明和定义) /* 注意,结尾无分号 */
{语句体} /* 语句体内包含局部变量 */
```

2. 传统定义格式：

```
函数类型标识符(函数参数表)
函数参数定义
{语句体} /* 语句体内包含局部变量 */
```

这两种定义在大多数情况下是等价的，但是有时也有区别。例如，在涉及 6 种编译模式时，现代风格的定义更有效（参见《指针》一章）。

传统格式由于把参数从函数圆括号中移出来，TC 就不能再进行错误检查，因此最好用现代格式。

在程序中函数定义都是独立的一部分，或者说，函数不允许嵌套定义，即在一个函数内部又定义另一个函数。

```
C>TYPE F8.C
#include "stdio.h"
main(){
double maxx(double x,double y) /* 在函数 main 中定义 maxx 函数 */
{return (x>y)? x:y;}
double x=8.8, y=7.8;
printf("z=%f\n",maxx(x,y));
}
```

将产生 Declaration syntax error 的错误。

13.4 函数的参数和函数中的变量

一 参数

1. 形参

在定义的函数名后面的括号中列出的变量称【形式参数或形参】。用现代格式定义时，必须说明形参的数据类型，多个变量间应用逗号隔开。在定义中指定的形参变量，在未出现函数

调用时,它们并不占内存单元。只有在发生函数调用时才给它分配内存单元。调用结束后,形参所占单元即被释放。这一点,你可以用对定义中变量输入相应的调试或监视表达式证实:未分配存储单元时,涉及形参的表达式无定义。

函数可以无任何形参,在这种情况下,函数后面的圆括号对仍是必写的,尽管圆括号对内无任何内容。

函数的局部变量不应写在形参表中,而只能在函数内部定义。

注意,TC 还有一个特殊的形参 "...",它由三个连续的小数点组成,表示参数个数未定,具体参见《接收自变量个数可变的宏》一章。

2. 实参

在调用函数名后面的括号中的变量或表达式称【实际参数或实参】。实参也可以是常量,也可以是由其它函数调用构成的表达式,如

```
a=min(x,max(y,z));
```

其中 min 和 max 都是函数。

3. 形参和实参应类型和个数相匹配。

4. 可以将实参值传给形参,但形参不能将值传给实参。这就是【值的单向传递性】。这是因为在函数调用结束后,形参所占的内存单元便不再存在,而实参值一般仍保留原值(只要不是全局变量且在被调函数中改变值)。

5. 例

```
C>TYPE F9.C
int x=1;           /* x 是全局变量 */
main(){
  int y=2;
  max(x,y);        /* x 作实参 */
  printf("x=%d\n",x);
}
max(int x,int y)   /* 形参和实参同名 */
{int z;
  z=( >y)? x:y;
  x+=4;            /* 改变 x 值,实际是改变形参值 */
  return z;        /* 返回时因 x 是形参,故其值 5 并不返回 */
}
```

即使把函数 max() 改成

```
max(int x,int y)
{int z;
  x+=4;
  z=(x>y)? x:y;
  return z;
}
```

语句 printf("x=%d\n",x); 输出同样是 x=1,而不是 x=5。除非将它改成

```
max(int t,int y)
{int z;
  x+=4;
```

```

z = (t > y) ? t : y;
return z;
}

```

那将输出 $x=5$ 。这时形参和全局变量不同名,各占不同的内存单元,故全局变量起作用。确切地说,前两个例子中尽管全局变量作为形参,故其值(等于1)被压入堆栈,函数调用结束后又从堆栈中弹出;而在第三种情况下,由于它未作形参,故其值在函数调用时也未压入堆栈,故其值随时可变。

二 变量

在函数内部定义的变量是局部变量,即在函数内部起作用,函数调用结束便不再存在。C语言中每个函数都将被编译成独立的代码块。如果函数中不涉及全局变量,则一个函数内定义的代码或数据不会和另一个函数内的定义的代码或数据发生相互影响,或者说,两个函数的作用域是不同的。

13.5 函数的返回值

当编译器遇到一个它以前未见到的函数时,它将假设这个函数返回一个 int 型的值,如果随后又将它定义成返回其它的值,如 char 型,就会出错。

从理论上讲,每个函数都应该返回一个值,但实际上很多函数并不需要返回任何值。TC 将函数说明为 void 类型,就是表示不返回任何值。

函数返回值大都是靠语句 return 实现的,即用

return 表达式

这个表达式的值就是函数要返回的值。这个值将传送给调用此函数的函数。

一个函数中可能有几个地方出现 return 语句,或者说可以在几个地方返回值。然而,不管有几个返回值,返回的数据类型一定要和函数说明的类型一样。如果函数说明的类型和 return 语句中表达式的值的类型不一致,则 TC 有时能自动将该值转换为函数说明的类型返回。但是应当指出,并不是所有的类型都能进行转换的(如实数与字符数据之间),因而应尽量避免。当然,有时也可以用

return ((type)表达式);

将表达式的值强制转换,这里 type 是函数说明的数据类型。

有时,希望从函数某一点退出来时就使用 return 语句,而将返回值作为退出点的标志。

例如:

```

.....
if (error == 1) return (-1);
.....
if (x >= 1) return (-2);

```

这返回的 -1 或 -2 等就可作为函数不符合某种条件的标志。

在实际应用中,return 语句后面的表达式在形式上可能是比较复杂的。

有时,函数中有 return 语句,但是它后面并不跟任何表达式;或者,有时甚至连 return 语句也没有,实际上它(函数结束标志即“}”)隐含有一个 return 语句。这两种情况,并不是说

它不返回值,而是返回了一个不确定值,因为程序员对返回什么值并不关心,反正用不到返回值。

为了明确不带回确定的值,可将函数定义成 void 型,或称【空类型】。当然,函数定义成 void * 即返回 void 指针的函数跟定义成 void 的函数是不同的。另一方面,定义成 void 的函数也可能通过形参(如指针、结构等)给某些变量赋值。这种值在某种意义上也可以认为是一种通过函数返回的值,因为当退出被调用函数返回调用函数中时,这些值在调用函数中仍起作用。

13.6 函数的调用和调用约定

1. 函数的调试表达式

用 F7 热键调试程序时可用调试表达式

函数名,p

看到函数的入口地址。跟数组名一样,编译器把函数名转换成一个地址。

2. 函数调用时,如不用函数原型,TC 不进行错误检查

它既不检查函数参数的个数,也不检查参数的类型,甚至连函数返回值也不检查。这样做的好处是,调用函数可在其定义前出现,从而使函数定义在程序中的位置可以灵活一些。

为了能对错误检查,你最好使用函数原型,它一般应在被调函数之前。

```
C>TYPE F10.C
```

```
#if 0
s(int xx,int yy)    /* 被调用函数在调用函数之前出现,编译后出错 */
{
    /* 编译通不过,自然不能生成执行文件了 */
    return xx+yy;
}
main()
{
    int x=2,y=3,z;
    z=s(x);          /* Error: Too few parameters in call to 's' */
    printf("%d\n",z);
} /* Warning: 'y' is assigned a value which is never used */
#endif

main()
{
    int x=2,y=3,z;
    z=s(x);
    printf("%d\n",z);
} /* Warning: 'y' is assigned a value which is never used */

s(int xx,int yy)    /* 被调用函数在调用函数之后出现,编译后只发出 */
{
    /* 一个警告,程序实际运行后得不到正确值 */
    return xx+yy;
}
```

在这个程序中,列举了把函数 s() 定义放在被调用之前和调用之后两种情况。对于 s() 放在调用函数之后时,编译程序不作检查,连形参与实参个数不匹配也未指出,编译虽顺利通过,但是这唯一发出的警告也是重要的!因为程序执行后不会得到正确的结果。

由此可见,在定义一个新函数 s() 时,有两种较好的基本形式(参见图 13-1):

形式 1:用函数原型	形式 2:不用函数原型
type s(type1,type2); /* 原型 */	type s(type1,type2) /* 定义 */
main()	{.....}
{	main()
s(); /* 调用 */	{
}	s(); /* 调用 */
type s(type1,type2) /* 定义 */	}
{.....}	

图 13-1

当然,还可以将函数原型或定义放到头文件中,然后将头文件包含到源程序中来。

3. 【传值】

调用函数时把实参值赋给形参变量叫传值。传值是通过堆栈进行的,即实参值先存入堆栈后再把值交给被调用函数(例如 f())。调用结束时来看实参值,其值不变(即使你在 f() 中对形参作过修改也是如此)。这就是传值的单向性,有时也叫函数的【赋值调用】。当实参或形参为指针时,虽然指针涉及地址,但也不可能影响实参。换句话说,TC 中不存在函数的【赋地址调用】。

```
C>TYPE F11.C
#define PP printf("px=%p py=%p\n",px,py); \
    printf("x=%d y=%d\n",x,y);
void swap—value(int *p1, int *p2)
{
    int temp;          /* 改数据 */
    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
void swap—address(int *p1, int *p2)
{
    int *temp;
    temp = p1;          /* 改地址 */
    p1 = p2;
    p2 = temp;
}
main()
{
    int x=10,y=80;
    int *px=&x, *py=&y;
    PP;                  /* px:DS:FFDC py:DS:FFDE x=10 y=80 */
    swap—value(px,py);
```

```

PP;          /* px:DS:FFDC py:DS:FFDE x=80 y=10 */
swap—address(px,py);
PP;          /* px:DS:FFDC py:DS:FFDE x=80 y=10 */
} /* 实参地址始终未变 */

```

4. 三种函数调用语句

调用一个函数可用调用函数语句来实现。根据被调用函数名出现在源程序中的位置，可把调用函数语句粗略地分成三种形式。

(1) 函数语句

格式：函数名(实参)；

例

```

getch();      /* 无实参 */
setbkcolor(1); /* 有实参 */

```

注意：这时把被调用函数作为一个独立的语句处理，初看起来有点像函数原型，但毕竟是有区别的：函数的参数全被实参代替！另一点要注意的是，当无实参时也别忘了写上圆括号对。

对空类型函数，即定义成 void 类型的函数，常用这种方式调用。如库函数 bar3d 就是一个 void 型函数，则用

```
bar3d(100,10,200,100,5,1);
```

语句调用是正确的。而不能用类似下述的函数表达式调用。譬如 x 为一个变量，则语句

```
x=bar3d(100,10,200,100,5,1);
```

是错误的。

(2) 函数表达式语句

格式：函数出现在一个表达式中

例

```

c=getch();      /* 键入字符赋给变量 c */
x=max(3,4);     /* 将调用函数 max 的值赋给变量 x */

```

(3) 函数参数语句

格式：函数语句作为另一个函数调用语句的参数

例

```
printf("p=%d\n",maxx(x,y,z));
```

这里 printf() 和 maxx() 都是函数。

5. near、far 和 huge 调用

一个 C 源程序可能源代码很多（因此其执行代码可能占用很大内存），这时可以分成几个源代码较小的源程序。那么这几个小源程序之间的函数间怎么互相调用？

在大模式下，如果使用递归函数

```

double power(double x, int exp)
{ if (exp<=0)return (0);
  else return (x * power(x,exp-1));
}

```

每次 power 调用本身时，需要用到较多的栈空间，所以必须使用 far 调用。因为一个函数名事

实上也可以看作是一个指针,所以可从 far 指针的概念理解【函数的 far 调用】。

将一函数说明成 far 类型需产生 far 调用。在小代码 (small) 模式中, far 函数 需在首次使用前定义和说明,以便保证产生 far 调用。

将函数说明成 near 类型,例如可以把 power() 说明成 near 型

```
double near power(double x,int exp)
```

可以使函数调用减小一些开销,这意味着函数只能在其本身所在的模块中调用。其它模块有其本身的代码段,因而不能调用另一模块中的 near 函数。而且 near 函数必须在第一次调用以前加以说明和定义,否则编译程序无法知道需产生 near 调用。

huge 调用含义同 far 调用,所不同的是 huge 型函数能够将 DS 置成新值,而 far 函数则不置值(参见《指针》一章)。

与一个程序相关的几个模块最好分别对同一种存储模式编译后再连接。如果在不同存储模式下编译后连接,处理不好会产生预料不到的结果。

6. 调用约定 (Calling convention)

Turbo C 支持函数的两种参数传递方法(也称【调用约定】或【传递顺序】): C 参数传递方法和 PASCAL 参数传递方法。为显式地指明调用约定可用 cdecl 或 pascal 修饰符。注意:使用 pascal 修饰符只是说明调用约定,而不是指 Turbo C 可以调用用 Turbo Pascal 编写的函数! 如果一个函数没有用修饰符 cdecl 或 pascal 指明,且当编译时又没有用 -p 选项(或在集成环境下用 O/C/Calling Convention pascal) 则缺省地认为它是用 cdecl 修饰符。注意,当用了 cdecl 则始终用 C 约定。

下面通过将程序 CDECL.C 和 PASCAL.C 比较就知道它们之间的区别。为此列出了它们在集成环境下的有关调试表达式和通过 TCC.EXE 汇编的源码。

(1) 函数名: CDECL.ASM 带下划线,区分大小写出(—funca),而 PASCAL.ASM 无下划线,也不区分大小写(FUNCA)。

(2) 对 CDECL.ASM 被调用参数无须将参数弹出栈(ret 后无数字),参数出栈是由调用者所完成的。指令

```
add sp,6
```

使编译程序知道栈内压进了多少参数;也知道调用 funca() 时返回地址已在栈内,并且已由函数 funca() 结束时返回指令弹出。而 PASCAL.ASM 无 add sp,6 指令,但它通过 ret 6 指令清除栈内内容,返回主函数。

(3) 可以看出, PASCAL.ASM 是将 funca() 的参数按从左到右的顺序依次压入栈内的,而 CDECL.ASM 则相反。

C>TYPE CDECL.C

```
void cdecl funca(int,int,int);
main()
{
int w=1,x,y;
long z;
x=5,y=7,z=0x1407aa;
funca(x,y,z);
printf("w=%d\n",w);
```

270

C>TYPE PASCAL.C

未标出表示同左边,打×表示没有
void pascal funca(int,int,int);

<pre> } void cdecl funca(int x,int y,int z) { printf("%d %d %d\n",x,y,z); }/* 在主函数 main()内 —SP,x: 0xFFD6 &w: Must take address of memory location /* 从 CDECL.ASM 可以看出, */ &x: Must take address of memory location /* si 和 di 被用来存储 w 和 x */ &y: Undefined symbol 'y' &z: DS:FFDC 在子函数 funca()内 —SP,x: 0xFFCC &w: Undefined symbol 'w' &x: DS:FFD0 &y: DS:FFD2 &z: DS:FFD4 */ C>TCC -S CDECL C>TYPE CDECL.ASM </pre>	<pre> void pascal funca(int x,int y,int z) &x: DS:FFD4 /* 先进栈 */ &y: DS:FFD2 &z: DS:FFD0 C>TCC -S PASCAL C>TYPE PASCAL.ASM </pre>
---	---

```

ifndef ?? version /* 等价于 Turbo Assembler 的版本号 */
? debug macro          /* 宏名: ? debug. 宏体为空          */
    endm              /* 标记宏定义结束          */
    endif
    ? debug S "n41.c"
—TEXT segment byte public 'CODE'
DGROUP group —DATA,—BSS /* 将 group 与多个段 —DATA,—BSS 联络起来,以便在这些段中所用的
                          标号或变量在计算偏移量时均从 group 的开始处算起 */
    assume cs,—TEXT,ds,DGROUP,ss,DGROUP
                          /* 指定段寄存器,以便计算在某给定段或组中定义的所有标号和变量的有
                          效地址 */
—TEXT ends /* 段结束 */
—DATA segment word public 'DATA'
    /* —DATA 为初始化的数据段 */
d@ label byte /* 定义一个具有类型 byte 的符号 */
d@w label word
—DATA ends
—BSS segment word public 'BSS'
    /* —BSS 为非初始化的数据段 */
b@ label byte
b@w label word
? debug C E9EB58441C056E34312E63
? debug C E92959441C056E34322E63
—BSS ends
—TEXT segment byte public 'CODE'
    /* —TEXT 为代码段 */

```

```

; ? debug L 2
-main proc near /* 定义过程 —main
取值为 near 型 */
    push    bp      /* 保存 bp */
    mov     bp,sp    /* bp 保存 sp */
    sub     sp,6     /* sp 减 6 */
    push    si      /* 保存 si,di */
    push    di
; ? debug L 4
    mov     si,1     /* 数 1 送 si */
; ? debug L 6
    mov     di,5     /* 数 5 送 di */
; ? debug L 6
    mov     word ptr [bp-6],7 /* 数 7 */
; ? debug L 6
    mov     word ptr [bp-2],20 /* 0x14 */
    mov     word ptr [bp-4],1962
                                /* 0x7aa */

```

/* si 和 di 寄存器用作内存指针时是相对于 DS 段寄存器的（用于串指令时相对于 ES 段寄存器）。

bp 也可用作内存指针，但它是相对于 SS（堆栈段寄存器）。因为堆栈与数据分别处于不同的段中，而在一般情况下，si 和 di（还有 bx）寄存器指向数据段，所以没有有效的方法使用它们指向堆栈中被传递的参数。因而为 bp 寄存器，它为参数、局部变量和其它基于堆栈的内存寻址提供了支持

```

; ? debug L 7
                                /* 调用函数 funca() 前进栈 */
    push    word ptr [bp-4]
    push    word ptr [bp-6]
    push    di
    call    near ptr —funca
                                push    di
                                push    word ptr [bp-6]
                                push    word ptr [bp-4]
                                call    near ptr FUNCA

```

/* 跟在 call 后面的指令的偏移量被压入栈中。该偏移量将由过程中近（返回）ret 指令弹出。这种 call 指令形式不改变 CS 寄存器。当过程完成时继续执行跟在 call 指令后面的指令。 */

```

    add     sp,6 /* 调整栈顶 */
; ? debug L 8
    push    si
    mov     ax,offset DGROUP:s@
    push    ax
    call    near ptr —printf
    pop     cx
    pop     cx

```

@1:

```

; ? debug L 9
    pop     di /* 恢复 di,si */
    pop     si
    mov     sp,bp
    pop     bp
    ret

```

/* 从过程中返回,将控制传送到存放在栈中的返回地址。该返回地址通常是由 call 指令放在堆栈中的,并且为返回到 call 下一条指令。对于段内(近)返回,栈中的地址是一个段偏移量,将它弹出到指令指针 IP 中。CS 寄存器不变。对于段间返回,栈中地址是一个指针,首先弹出偏移量,然后弹出段值。 */

```

—main endp
;      ? debug L 10
—funca proc      near      FUNCA  proc      near
    push    bp
    mov     bp,sp
;      ? debug L 12
    push    word ptr [bp+8]      push    word ptr [bp+4]
    push    word ptr [bp+6]      push    word ptr [bp+6]
    push    word ptr [bp+4]      push    word ptr [bp+8]
    mov     ax,offset DGROUP:s@+6
    /* offset DGROUP:s@ 返回段 DGROUP 中的偏移量 s@ 送入 ax 中 */
    push    ax
    call    near ptr —printf
    /* near ptr 将一个地址表达式强制为近的代码指针 */
    mov     sp,bp
@2:
;      ? debug L 13
    pop     bp
    ret                                ret 6
    /* ret 后的数字 6 给出了弹出返回地址后所要释放的堆栈字节数或字数,这里是 6 个字节,指进栈
       的 0x14.0x07aa */
—funca endp      FUNCA endp
—TEXT ends
    ? debug C E9
—DATA segment word public 'DATA'
    /* word 表示定义段起始内存边界的类型为字。public 表示将所有同名段组合在一起形成一个
       单一的连续段,然后将 SS 初始化为段首地址,SP 初始化为段的长度。'DATA' 用于控制程
       序连接时段的顺序,把同名段一起装入内存,不管在源文件中顺序如何。 */
s@  label  byte
    db      119    /* 字母 w */
    db      61     /* 字符 = */
    db      37
    db      100
    db      10     /* '\n' */
    db      0
    db      37     /* 字符 % */
    db      100    /* 字母 d */
    db      32     /* 空格 */
    db      37
    db      100
    db      32
    db      37

```

```

db      100
db      10      /* '\n' */
db      0
--DATA  ends
--TEXT  segment byte public 'CODE'
        extrn _printf,near
        /* extrn 表示 _printf 是在另一个模块中定义的符号名字 */
--TEXT  ends
        public  _main
        public  _funca
        end      /* 源文件结束标记 */
        public FUNCA

```

使用 pascal 修饰符的好处是：

- (1) 可以调用使用这种参数传递顺序的汇编子程序；
- (2) 可以调用以其它语言书写的子程序；
- (3) 相对而言，它产生的调用代码较小，因为它无需在调用后清栈。

使用 pascal 修饰符可能引起的麻烦是：

(1) 不能传递可变参数，因为它要根据参数个数对栈作相应调整，参数过多过少均会产生问题；

(2) 如选用 `-p` 选择项，必须引用所有被调用标准 C 函数的相应嵌入文件，否则 Turbo C 将会对这些函数也使用 pascal 调用约定的，这样就会出错！因为那样由于栈未得到清除，且参数的顺序也不会对。

标准嵌入文件将每个函数定义为 `cdecl` 类型，所以引用这些嵌入文件后编译程序将使用 C 调用约定。但是 `cdecl` 的标识符带有下划线，而 pascal 的则没有，除非使用无下划线的编译选择项，且不区分大小写而连接，否则将会引起麻烦。

如在 Turbo C 源程序中使用 pascal 调用约定，则应尽量使用函数原型，并且对函数最好显式地定义成 `cdecl` 或 pascal 类型。

13.7 函数说明、函数定义和函数调用之间的关系

要调用一个函数，则在程序中必然要出现调用此函数的语句，调用函数语句一定出现在某一个函数定义的语句体中；要使用此函数，则此函数的实际代码必须存在，或者说函数应当有定义；为使函数定义有灵活性，并对函数作一定的检查，在函数作用域前应对函数作说明。

按现代风格，函数的定义和函数的说明区别在于函数说明总是以分号结尾的，而定义结尾没有分号。

使用函数说明并不改变所定义的函数，但在编译时要对定义的函数作检查。

如果函数定义在函数调用之前出现，便可不要函数说明。如果你又写了函数说明，则函数定义可以出现在文件中其它地方（当然要注意和函数相关的变量等位置），但两者之间必须匹配。如果函数定义与函数原型不匹配时，将出现编译错误。

函数说明后，程序的后续部分就能识别它的存在，其它的函数就可以调用它了。但是，在通常情况下，进行了函数说明并不意味着该函数的实际代码就存在。对库函数外的函数，只有进行了函数定义，函数本身的实际代码才存在。库函数的代码已存在，要使用它必须进行说明（一般在源程序中包含有库函数原型的标头文件）。

一个函数可在函数内部对形参值重新赋值,但这不影响调用该函数的实参值,因为值是单向传递的。换言之,函数的形参是该函数的局部变量。但对中断函数可能例外。

当没有预先说明函数原型时,TC 根据表达式的算术转换原则将形式参数转换。当作用域内有函数原型时,TC 将定义中的参数按说明中的参数类型转换。

```
C>TYPE F12.C
max(float x,float y){} /* 定义了一个空函数 */
main()
{max(2.3,3.3);}在监视窗口内输入调试表达式
x,f
y,f
```

可见 x, y 都有正确值。现在把函数 max 改成

```
max(int x,int y){}
```

则

```
x,f: 2
y,f: 3
```

然后又把它改成

```
max(float, float);
max(int x,int y){}
```

或

```
max(int, int);
max(float x,float y){}
```

或

```
float max(int, int);
max(float x,float y){}
```

都将出现

```
Type mismatch in redeclaration of 'max'
```

即类型不配的错误。

现再把程序变成

```
C>TYPE F13.C
main()
{max(2.3,3.3);}
max(float x,float y){} /* 定义了一个空函数 */
```

则可以看出 x, y 的值不正确。如把它改成

```
C>TYPE F14.C
max(float, float); /* 函数原型。函数说明后,此后该函数便可应用了 */
main()
{max(2.3,3.3);}
max(float x,float y){} /* 定义了一个空函数 */
```

就正确了。如把此程序改成

```

C>TYPE F15.C
main()
{max(2.3,3.3);          /* 调用 max 语句 */
max(float x,float y){}  /* max 定义语句 */
max(float, float);      /* max 原型 */

```

或

```

C>TYPE F16.C
main()
{max(2.3,3.3);
max(float, float);      /* 原型在定义前,但在调用后 */
max(float x,float y){}

```

都是不正确的。为什么？因为函数原型或函数定义语句均在函数调用语句之后，所以在调用该函数时便出错。

1. 在同一个文件中,如果函数定义出现在被调用函数之前,可以不要函数原型;
2. 为使函数在不同文件都能应用,应将函数说明定义在一个头文件之中,然后在要用的文件中包含这个头文件;
3. 当头文件中有函数说明,而在你的文件中又有此函数说明,两者应完全一样才行。

13.8 函数的嵌套调用

TC 允许嵌套调用函数,即一个函数 A 中可以出现调用函数 B 的语句,而在函数 B 中又可以出现调用函数 C 的语句等等;同一函数还可被多次调用。在函数嵌套调用时应注意各个函数的作用域。

TC 允许在一个函数中对另一个函数进行说明,从而使该函数在这函数中可引用。例如,

```

C>TYPE F17.C
#include "stdio.h"
main(){
double maxx(double,double);
double x=8.8, y=7.8,z;
z=maxx(x,y);          /* 函数 main 调用函数 maxx */
printf("z=%f %f\n",z,maxx(x,y)); /* 输出两个相同值 */
}
double maxx(double x,double y)
{return (x>y)? x:y;}

```

嵌套形式的函数调用与函数的编写顺序无关。或者说,各个函数可按任意顺序放在程序中,且被认为贯穿整个程序的全局变量,包括并未进行函数说明就被使用的函数。但在一定定义或说明之前调用函数时,一定要注意使用它的条件是否成立。

13.9 函数的递归

当调用语句中调用的函数是调用语句所在函数的本身时,这就是函数的递归调用。TC 允许函数递归调用。

```

C>TYPE F18.C
main( )
{
int x=3,y=4,t;
t=power(x,y);
printf("t=%d\n",t);
}
power(int x,int y)
{
int n;
if(y>0)
n=power(x,y-1)*x;
else
n=1;
return n;
}

```

对这个程序可用 F7 热键进行跟踪,并用 Debug/Evaluate Ctrl-F4 检查堆栈,便可以发现堆栈里有像

```

power(3,0)      /* 从调用堆栈可以看到进栈时函数的实参值 3 与 0。下同 */
power(3,1)
power(3,2)
power(3,3)
power(3,4)
main()

```

这样的函数,还可以看到它们先进后出的进出栈情况。

尽管从算法的角度看,递归使程序具有简洁明了的特点,然而由于它占用堆栈空间多,进出栈时间也长,所以尽量少用。

使用函数递归调用还应注意使递归结束的条件,以避免无限递归。

13.10 内部函数

如果希望一个函数只能在其所在的(源)文件中调用,而在其它(源)文件或模块中不起作用,则可以在函数定义时加上修饰符 static,这样的函数就是【内部函数】(或【静态函数】)。如

```
static int max(int x, int y)
```

由于这种函数已被局限于一个文件中,所以即使其它文件中有同名函数,也不会受影响。

13.11 外部函数

【外部函数】的意思和内部函数相反,它可为其它文件引用。其标准格式是在函数类型定义中写上修饰符 extern,许多函数前既无 static,又无 extern,则隐含 extern,即 extern 是缺省值。如

```
extern int max(int x,int y)
```

或

```
int max(int x, int y)

C>TYPE f3.PRJ          /* 编译用的规划文件 */
f20.c
f19.c
f21.c

C>TYPE F19.C
main() {                /* 调用两个外部函数 max, min */
printf("max=%d min=%d\n",max(2,3),min(2,3));
}

C>TYPE F20.C
max(int x,int y)        /* 也可写成 extern max(int x, int y) */
{ return ((x>y)? x:y); }

C>TYPE F21.C
min(int x,int y)        /* 如将此句加上 static 则 F19.C 发现 min 无定义 */
{ return ((x>y)? y:x); }
```

13.12 程序的可执行语句应在函数定义的语句体中

程序的【可执行语句】必须包含在一个函数的语句体中。可执行语句一般包括：

1. 控制语句

if else, for, while, do while, continue, break, switch case, goto, return 等等。

2. 表达式语句

如 $x=3$;

3. 函数调用语句

如 `printf("%d\n",x);`

4. 复合语句

由 {} 括起来的语句体。如

```
{
    x=5; y=6;
    z=max(x,y);          /* 注意:此句的结束符即分号不能省略 */
}
```

程序调试时除可执行语句外的语句是非执行语句。区分一个语句是否为可执行语句可用 F7 热键单步跟踪程序看出,非执行语句会被跳过。

13.13 函数的种类

1. 库函数。TC 定义了数百个库函数,我们将在各有关章节中单独讨论它的应用方法。在一般情况下,使用库函数时应将包括它原型的标头文件包含进来,特别像使用数学函数时别忘了包含 `math.h` 文件!但是,有时如不想对函数检查,又不用标头文件中的符号常量、宏或外部变量等,也可以不包含标头文件,当然,这要在有把握不出问题时才这样做。

2. 用户自定义函数。

3. 空函数

这种函数既无实际参数,语句体内也无任何语句,如

```
zone(){}
```

它的作用是在程序设计初期在程序中预留一个位置,供以后维护时一看就清楚。这种函数不影响程序的执行,因为它什么事也不做。

4. void 型函数的说明

在现代 C 函数中,可以用修饰符 void 将不明确返回数值的函数说明为 void 型。

void 可以认为是一种空类型。例如,一些在程序运行时分配内存的函数(如 malloc)的类型都被说明为 void * 型,即它们返回一个无类型的指针。在 TC 中它们可以不经类型强制转换就将它赋给任何类型的指针。不过,有时为了可移植性,建议使用类型强制转换。

13.14 函数和数组

C>TYPE F22.C

```
int min(int list[], int size)
{
    /* 求出数组各元素中具有最小值的元素下标值 */
    int i, mini=0, min;
    min=list[mini];
    for(i=1; i<size; i++)
    if(list[i]<min) {min=list[i]; mini=i;}
    return mini;
}

main() {
    #define VSIZE 22
    int vector[VSIZE];
    int i;
    for(i=0; i<VSIZE; i++)
    {vector[i]=rand(); /* rand 是随机数发生器,产生 0~215-1 的伪随机数 */
    printf("vector[%2d]=%6d\n", i, vector[i]);
    }
    i=min(vector, VSIZE); /* 调用 min() 函数 */
    printf("mini: vector[%2d]=%6d\n", i, vector[i]);
}
```

仔细观察调用函数语句,可以看出 C 功能强大的一面;在编译时,编译器只关心数组的起始地址,并不关心它在哪里结束。真正送给 min 函数的是 vector 数组的起始地址,而不是复制整个数组。所以在 C 中,能够将不同大小的数组(但必须是同类型)传递给一给定函数。

C>TYPE F23.C

```
int min(int list[], int size)
{
    int i, mini=0, min;
    min=list[mini];
    for(i=1; i<size; i++)
    if(list[i]<min) {min=list[i]; mini=i;}
```

```

return mini;
}

void setrand(int list[],int size)    /* 该函数用来初始化数组的值 */
{ int i;
  for(i=0;i<size;i++)list[i]=rand();
}

main() {
  #define VSIZE 22
  int vector[VSIZE];
  int i;
  setrand(vector,VSIZE);          /* 调用函数后数组 vector 的元素被赋值 */
  for(i=0;i<VSIZE;i++)
  printf("vector[%2d]= %6d\n",i,vector[i]);
  i=min(vector,VSIZE);            /* 求最小值元素的下标 */
  printf("mini: vector[%2d]= %6d\n",i,vector[i]);
}

```

C>TYPE F24. C

```

#define CSIZE 4    /* 注意:二维数组的长度都必须为 CSIZE,而不能空缺 */
int minj=0;        /* 因一个 min 函数一般只能返回一个值,所以利用外部变量 */
int min(int list[][CSIZE], int size) /* 利用二维数组 */
{
  int i,j,mini=0,min;
  min=list[mini][minj];
  for(i=1;i<size;i++)
  for(j=0;j<CSIZE;j++)
  if(list[i][j]<min){min=list[i][j],mini=i,minj=j;}
  return mini;      /* 实际返回 mini 值和 minj 值 */
}

void setrand(int matrix[][CSIZE],int size) /* 二维数组 */
{int i,j;
  for(i=0;i<size;i++)
  for(j=0;j<CSIZE;j++)matrix[i][j]=rand();
}

main() {
  #define VSIZE 22
  int vector[VSIZE][CSIZE];    /* 二维数组 */
  int i,j;
  setrand(vector,VSIZE);        /* 函数调用语句 */
  for(i=0;i<VSIZE;i++)
  for(j=0;j<CSIZE;j++)
  printf("vector[%2d][%2d]= %6d\n",i,j,vector[i][j]);
  i=min(vector,VSIZE);          /* 函数调用语句 */
  printf("mini: vector[%2d][%2d]= %6d\n",i,minj,vector[i][minj]);
}

```

C>TYPE F25. C

```

#define CSIZE 4

```

```

int minj=0;
int lmin(int list[][CSIZE],int size)
{
    int i,j,mini=0,min;
    min=list[mini][minj];
    for(i=1;i<size;i++)
        for(j=0;j<CSIZE;j++)
            if(list[i][j]<min){min=list[i][j];mini=i;minj=j;}
    return mini;
}

void setrand(int matrix[],int size) /* 注意此函数中 matrix 为一维数组 */
{
    int i,j;
    for(i=0;i<size;i++)
        matrix[i]=rand();
}

main()
{
    #define VSIZE 22
    int vector[VSIZE][CSIZE]; /* 数组定义 */
    int i,j;
    setrand(vector,VSIZE*CSIZE); /* 注意此句调用格式 */
    for(i=0;i<VSIZE;i++)
        for(j=0;j<CSIZE;j++)
            printf("vector[%2d][%2d]=%6d\n",i,j,vector[i][j]);
    i=lmin(vector,VSIZE);
    printf("mini: vector[%2d][%2d]=%6d\n",i,minj,vector[i][minj]);
}

```

程序 F25. C 同样可取得与上述相同的结果,尽管在编译时产生

Suspicious pointer conversion in function main

警告,但你可不管它(即强行抑制)。当然,这种函数调用虽然可以,但由于直观性差,还是尽量少用。

在这几个程序里,我们没用函数 rand 的原型(在 stdlib.h 中)也能正确执行,但不要以为都是这样。

```

C>TYPE F26. C
main(){
    double x=8.8, y;
    y=log10(x);
}

```

就不能得到正确的 y 值。只有

```

C>TYPE F27. C
#include "math.h"
main(){
    double x=8.8, y;
}

```

```
y=log10(x);
}
```

或

```
C>TYPE F28.C
double log10(double); /* 此说明和 math.h 中说明完全一样 */
main(){
double x=8.8, y;
y=log10(x);
}
```

才能得到正确值。而将第一句变成（函数说明和 math.h 中说明不一样）

```
log10(double);
```

或

```
double log10(int); 及 log10(int);
```

都不能得到正确结果。而

```
C>TYPE F29.C
main(){
double x=8.8, y;
y=log10(x);
}
double log10(double);
```

将产生一个

```
Type mismatch in redeclaration of 'log10'
```

重定义类型不匹配的误差（因为函数 log10 先被调用，后又被说明为一个非整型，就会产生这种错误）。

13.15 函数和指针

指针可以作为函数的形参；指针数组可以作为主函数的形参；可以定义指向函数的指针，也可以定义指针函数，以及返回指针的函数；并且函数名有时可以作为指针来处理。（参见《指针》一章）。

13.16 汇编语言调用 Turbo C 函数

为了使汇编语言模块能够调用 C 函数，必须在汇编模块中作以下说明：

```
EXTRN <fname>:<fdist>
```

其中 <fname> 为函数名，<fdist> 可以是 near，也可以是 far，这取决于此 C 函数是 near 类型还是 far 类型。若 <fdist> 是 near，此 EXTRN 语句必须出现在汇编模块的代码段中；如果是 far，则此语句应在任何段的外部。所以下面语句

```
EXTRN —myCfunc1:near,—myCfunc2:far
```

允许汇编子程序调用 —myCfunc1 和 —myCfunc2 两个 C 函数。

第十四章 程序结构和主函数

14.1 程序结构

14.1.1 独立的 C 源程序

一个独立的 C 源程序内容如图 14-1 所示,它包含一个主函数 `main()`,在集成环境里能单独经编译连接后生成执行文件。程序开始执行的时候,首先调用主函数,然后通过该函数中的函数调用语句调用其它函数,其它函数之间也可以相互调用。在某种程度上可以说,C 源程序全部由函数组成。

预处理命令 (包含文件语句,宏定义语句等)

类型定义

全局变量说明

函数说明 (函数原型)

`main()`

{

局部说明 (如定义变量) 语句

.....

调用函数 1 语句;

..... /* 其它语句 */

调用函数 n 语句;

.....

}

函数 1 定义;

.....

函数 n 定义;

图 14-1

14.1.2 源程序由几个子源程序构成

当一个源程序太大时可分成几个子源程序,但多个子源程序中只有一个有主函数。如果一个子源文件满足

- (1) 包含有主函数;
- (2) 被调用函数在子源文件内都有定义。

这样两个条件,则称该子源文件为【主文件】(Primary C file),其它子源程序相对主文件而言可称为【支持文件】。这个源程序本身可称为【多文件程序】。例如,下面的程序 M1.C 虽满足第一个条件但不满足第二个条件; M3.C 能满足第二个条件但不满足第一个条件;只有程序 M2.C 能同时满足这两个条件(也可用预处理程序 CPP.EXE 帮助分析),所以,在集成环境中可选

Primary C file M2.C

后进行编译连接,最后生成执行文件。

```
C>TYPE M1.C          /* 支持文件 */
#include "m3.c"
main(){
printf("main");
fun(4);
f('A');
}

C>TYPE M2.C          /* 应选本文件名为主文件名 */
#include "m1.c"        /* 因包含了 m1.c,所以可认为也包含了主函数 */
fun(int y)
{
printf("y=%d\n",y);
}

C>TYPE M3.C          /* 支持文件 */
f(char c)
{printf("%c\n",c);
} /* 用 Compile/Compile to OBJ 对它编译无问题,但用 Compile/Compile/
    Make EXE file 编译时将出现
    Linker Error: Undefined symbol '—main' in module C0S 错误。 */
C>CPP M2.C           /* 用预处理程序 CPP.EXE 生成 M2.I 文件 */
C>TYPE M2.I          /* 可用 C>CPP -P- M2 生成不带行号的 M2.I 文件 */

m2.c 1:
m1.c 1:
m3.c 1: f(char c)
m3.c 2: {printf("%c\n",c);
m3.c 3: }
m3.c 4:
m3.c 5:
m3.c 6:
m3.c 7:
m1.c 2: main(){
m1.c 3: printf("main");
m1.c 4: fun(4);
m1.c 5: f('A');
m1.c 6: }
m1.c 7:
m2.c 2: fun(int y)
```

```

m2.c 3: {
m2.c 4: printf("y=%d\n",y);
m2.c 5: }

```

(1) 子源程序中可以用包含语句包含其它子源程序,包含文件语句如果跟函数执行顺序无关,可放在任一子源程序中(如图 14-2 和图 14-3 所示)。编译时应指定主文件。

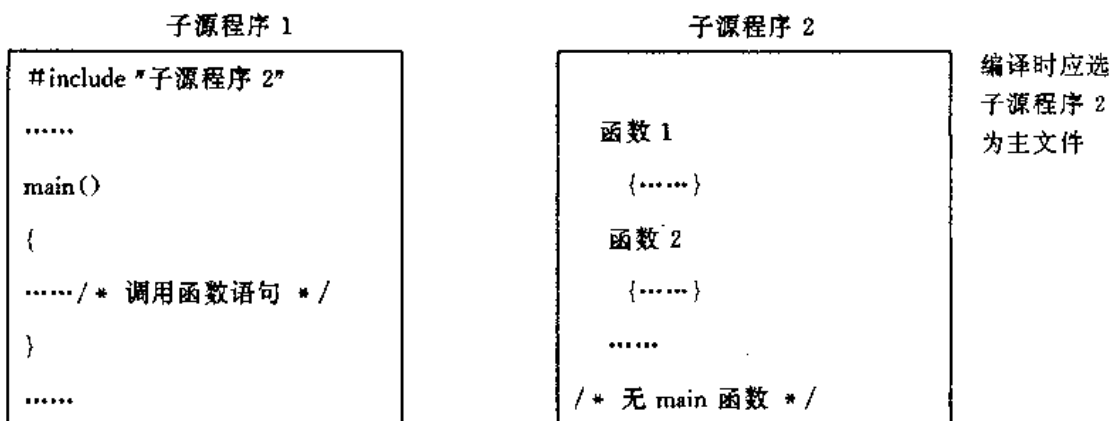


图 14-2

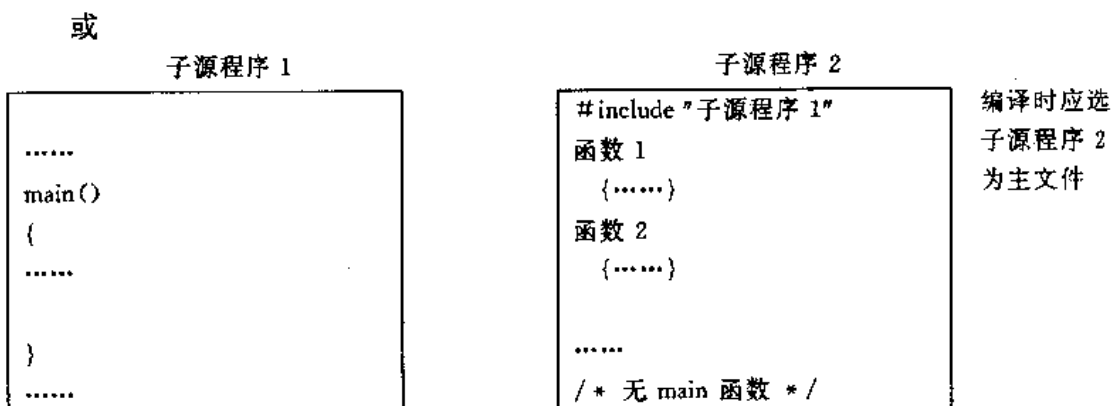


图 14-3

(2) 源程序分成几个子源程序后,没有使用包含指令 #include 将其它子源程序包含(允许包含嵌套),编译时应指定规划文件 *.prj 名进行编译(如图 14-4 所示,参见《集成开发环境和缺省参数设置》一章),此时不用再指定主文件。

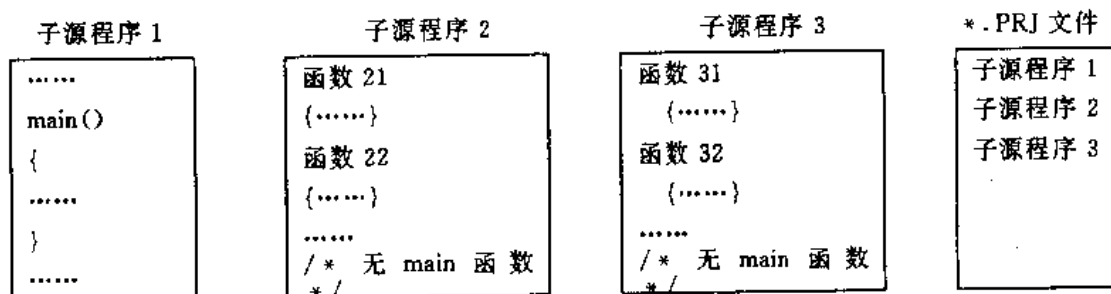


图 14-4

14.1.3 并列源文件

每个源程序都是独立的,即都能单独编译连接生成执行文件(因此每个文件都有主函数

), 生成的执行文件之间可以互相调用执行, 即一个执行程序可以是另一个的子程序 (如图 14-5 所示, 参见《过程控制函数》一章)。

源程序 1 (n1.C)

```
.....
main()
{.....
/* spawn1(P-WAIT,n2.EXE,NULL); */
}
.....
```

源程序 2 (n2.C)

```
.....
main()
{.....
}
.....
```

图 14-5

14.2 源程序部分内容说明

14.2.1 文件名

为方便叙述, 对文件名的成份命名规定如下:

X:\YYYY\YYY\ZZZZZZZZ.WWW

在这个文件全名中, 可称

X: ——驱动器名 /* X 可以是 A、B、C、……等 */

\YYYY\YYY\ ——路径名

ZZZZZZZZ ——基本名 /* 最多 8 个字符 */

.WWW ——扩展名

WWW ——附加名 /* 最多 3 个字符 */

例如, C:\TC\INCLUDE\STDIO.H 是一个完整的文件名。有时也把基本名与扩展名一起称为文件名。参见 Turbo C 库函数 `fnsplit()` 和 `fnmerge()`。

每个 C 源程序常用的缺省扩展名为 .C。扩展名常用于文件分类, 如

- * .PCK 检选文件 (Pick file)
- * .PRJ 规划文件 (Project file)
- * .MAP 映射文件 (Map file)
- * .TC 配置文件 (Config file)
- * .BAK 后备文件 (Backup file)
- * .LIB 库文件
- * .OBJ 目标文件

14.2.2 【标识符】(identifier)

标识符用于表示变量名、函数名、数据类型名或其它目标名。标识符由字母 A ~ Z、a ~ z、0 ~ 9 和下划线 (—) 组成, 并由字母或下划线 (underscore) 开头的字符串。在标识符中, 大小写字母是有区别的。在集成环境里, 命令

Options/Linker/Case-sensitive On

表示标识符的大小写是有区别的; 如置成 Off 则不区分。该开关称为大小写敏感连接开关。注意: 对 Pascal 类型的标识符, 大小写是不加区分的。

尽管标识符可以以下划线开始, 但用户尽量不要这样做, 因为字首或字尾加下划线的标识

符常被用于低级系统函数的标识符。

标识符的长度最大为 32 个,即超过这个长度时只有前 32 个字符有效。注意,对有些语言,有效字符的长度可能没有这样多。这个长度值也是可以改变的,例如在集成环境中选 命令

Options/Compiler/Source/Identifier length

就可改变它。

14.2.3 双限界匹配符

表 14-1 中所列符号常配对使用,可称它们为【双限界符】。

表 14-1

名 称	形 式	带方向?	允许嵌套吗?	搜 索 操 作 键
花括号	{ 及 }	是	允许	Ctrl-Q Ctrl-[或 Ctrl-Q Ctrl-]
尖括号	< 及 >	是	允许	
圆括号	(及)	是	允许	
方括号	[及]	是	允许	
注释符	/* 及 */	不	不允许/允许	(O/C/S/Nested comments Off 时) 向前搜索用 Ctrl-Q Ctrl-[
双引号	" 及 "	不	不允许	(O/C/S/Nested comments On 时)
单引号	' 及 '	不	不允许	向后搜索用 Ctrl-Q Ctrl-]

说明:

1. “带方向”含义

例如在正文中,左花括号 { 必定在右花括号 } 之前,所以说花括号是带方向的。

2. “嵌套”含义

这里讲的嵌套是指双限界符的嵌套,例如

y[m[x]] 或 ((x>0) && (y<0))

3. “搜索方向”的含义

向前搜索是指在正文的当前光标后面(或右边,或下面)处搜索配对的双限界符。

向后搜索的意思与之相反(参见图 14-6)。

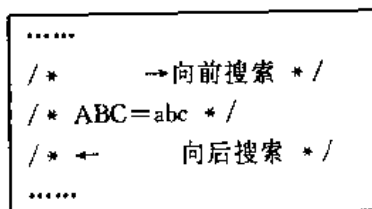


图 14-6

4. 检查双限界符配对时使用的搜索方法

将光标移到一个双限界符上。注意:对 /* 只能移到斜杠(/)上,而对 */ 只能移到星号上(*)。然后按表 14-1 使用搜索操作键。如果另一个配对的双限界符存在,键操作后光标立即移到那个双限界符上;否则光标不动。

14.2.4 注释

Turbo C 允许源程序中添加注释,以提高可读性,但它不会在编译后变成目标码,换句话说,编译时将它忽略。注意:在集成环境下编译时如果有过多的注释,以至于使程序很庞大,则有可能影响编译,因此使用简短、明确的注释也是一项很有意义的工作。

注释可用特殊的符号对

```
/* */
```

括起来(注意:斜杠和星号之间无空格)。当该括号对内又出现注释括号对时,称为【注释嵌套】。在缺省方式下, Turbo C 不允许注释嵌套。但可以选择适当的命令允许注释嵌套。例如,在集成环境下,用

Options/Compiler/Source/Nested comments On

命令实现。在调试程序时应经常加注释,也许要这样做。

另一个允许嵌套注释的方法是用编译指令对 #if 0 和 #endif 将嵌套注释括起来,那样在编译时就不会遇到任何麻烦了。

```
C>TYPE M4.C
#include "stdlib.h"
main()
{
    int x;
    randomize();
    x=random(100);
    #if 0
    /* 产生一个 0~99 的随机数
       /* 如果 x>50 印出 50,否则印出 1 */
       这是一个表演程序 */
    #endif
    if(x>50)printf("%d\n",50);
    else printf("%d\n",1);
}
```

14.2.5 语句与编译指令

Turbo C 语句可以从一行上任一位置开始写,以分号或)(复合语句) 结束。书写格式比较自由,既可以在一行上写几个语句,也可以将一个语句分写到几行上。除字符串和宏体分在几行上时要写续行符(\) 外,其它情况可写可不写。语句没有行号。

在集成环境下,用单步调试很容易看到语句可以分成执行语句和不执行语句。

编译指令是供编译处理用的,它总以符号 # 开头。

14.2.6 函数

函数相当于子程序,每一个函数都完成特定的功能。程序主要由函数构成,一个独立的能生成执行代码的源程序至少包含一个函数(main(), 称【主函数】)。函数可以由用户自己定义,也可以直接使用 Turbo C 提供的【库函数】(大约有 300 多个)。对库函数只要源程序中包含它的函数原型所在的标头文件(*.h) 就可直接调用它了。Turbo C 没有提供库函数的源程序,但无关紧要,因为用户只要知道怎样正确使用它就可以了。

函数一旦定义,其实际代码就存在了。因此,当你在源程序中定义一个和某库函数同名的函数时是危险的,因为库函数的原有函数功能将被改变。当然,如果定义的函数功能和库函数一样,则是可以的,这可称为等效函数。

函数被说明后,程序的后续部分就能识别它的存在,其它函数就可调用它。

详见《函数》一章。

14.2.7 关键字

关键字是指 Turbo C 特有的,用户只能使用不能改变它。关键字可分为两类,一是和标准 ANSI 一样的,另一种是 Turbo C 扩充的。(参见《语句和关键字》一章。)

事实上,从只能严格使用的角度看,还有一些标识符也是很关键的,尽管它们不被认为是关键字。像 FILE、编译指令用到的一些字符串、省略号 (...)、五个预定义流和预定义宏等等。更广义一点,库函数名等也是。

当在编辑窗内写上一个标识符时,你可把编辑光标移到该标识符上,然后按 Ctrl-F1 键,如果这时显示

There is no language help available for this item.

那样的信息,表示不是 Turbo C 已用的标识符,而是用户自定义的标识符。如为 Turbo C 已用标识符(如关键字、库函数名、包含文件名、变量名、宏名、符号常量等等)则有相应的帮助信息显示。

14.3 主函数 main()

每一个独立的可编译生成执行文件的 C 源程序都必须有一个 main() 函数。

14.3.1 主函数在程序中的位置

主函数可以放在源程序中任何地方,但程序执行时总是从主函数开始执行。其它函数相对于主函数的位置是无关紧要的,只要记住一个原则,先定义或说明,后应用。

14.3.2 参数

在 Turbo C 程序启动过程中,主函数可以传递三个参数: argc、argv 和 env。这三个参数中后两个是命令行参数,即在 DOS 提示符后打入的参数。

argc: 整数,自动记录传给命令行参数的个数。

argv: 字符串数组。

在 DOS 3.X 版本中,argv[0] 为执行程序全名,但不包含命令行上的参数;对于 DOS 3.0 以下版本,它为一个空串("")。

argv[1] 为命令行上第一个参数,也是一个字符串;

argv[2] 为命令行上第二个参数,也是一个字符串;

.....

argv[argc] 为空 (NULL)。

env: 字符串数组。

它的每一个元素是环境变量串。

说明:

1. 这几个参数是任选的,但它们遵循三个原则,一是它们的排列顺序是不能颠倒的,二

是一个参数的左边的参数是必选的。最后,参数的类型也是规定的,不能用其它类型。即它们只能有以下几种形式:

```
main()
main(int argc)
main(int argc, char * argv[])
main(int argc, char * argv[], char * env[])
```

2. 习惯上常这样书写这几个参数名,仅管用其它的标识符也是可以的。

3. 它们是主函数的局部变量。

C>TYPE M5.C

```
main(int argc, char * argv[], char * env[])
```

```
/* 为什么数组 argv 没有指定长度? 这是因为数组作函数参数时,函数没有必要知道
   数组的长度。在 C 中能够将不同大小的数组(但必须是同类型)传递给一给定
   函数的。*/
```

```
{
int k;
printf("argc=%d,表示传送%d 个参数\n",argc,argc);
printf("argv[]数组的情况是:\n");
for(k=0;k<=argc;k++)
printf("argv[%d]:%s\n",k,argv[k]);
printf("env[]数组的情况是:\n");
for(k=0;env[k]!="";k++)
printf("env[%d]:%s\n",k,env[k]);
}
```

/* 假定在命令行上打入(注意:命令行参数的最大长度为 128 个字符,包括参数间的空格,这是由 DOS 决定的):

C>M5 first "MY TEST" 8 End

程序输出:argc=5,表示传送 5 个参数

argv[]数组的情况是:

argv[0]:C:\TC\M5.EXE

argv[1]:first

argv[2]:MY TEST

argv[3]:8

argv[4]:End

argv[5]:(null)

env[]数组的情况是:

env[0]:COMSPEC=C:\COMMAND.COM

env[1]:DTP=C:\DTP;

env[2]:PATH=C:\DTP;C:\DOS;C:\LXPC\SYS

env[3]:TEMP=C:\DOS

env[4]:PROMPT=\$p\$g

env[5]:(null)

而在命令行上打入

C>SET

后显示:

COMSPEC=C:\COMMAND.COM

```

DTP=C:\DTP ;
PATH=C:\DTP;C:\DOS;C:\LXPC\SYS
TEMP=C:\DOS
PROMPT= $p $g          */

```

14.3.3 使用关键字 cdecl

在用命令行编译器 (TCC. EXE) 编译连接时如果使用了 -p 选择项,或在集成环境中使用了命令

Options/Compiler/Code convention Pascal

则 main 前面应加上 cdecl 修饰符。这是为了摆脱编译选择项 -p 的约束,把 main() 作为一个正规的 C 函数,因为 C 启动代码库 C0S (Small 存储模式) 总是按照 C 参数传递方法调用 main()。

C>TYPE M6.C

```
main()
```

```
{
```

```
printf("测试何时加 cdecl 修饰符\n");
```

```
}
```

/* 如选用集成环境命令

Options/Compiler/Code convention C

则编译连接没有任何问题。但如选用

Options/Compiler/Code convention Pascal

在编译时没有问题,但在连接时出现

Linker Error:Undefined symbol '-main' in module C0S

Linker Error:Undefined symbol 'PRINTF' in module M6.C

如将程序第一行前增加一句并把 main 前加上修饰符 cdecl,即

```
extern cdecl printf();
```

```
cdecl main()
```

则程序能生成执行文件。

*/

14.3.4 返回值

像一般函数一样,main 也可以有返回值。当一个程序被另一个程序调用时,其返回值可能是有意义的。

C>TYPE M7.C

```
#include "process.h"
```

```
#define PRET(X) if(ret==0)printf("#X"执行正常\n"); \
```

```
else printf("#X"执行不正常\n")
```

```
main() /* 当你键入数字 1 或 2 后执行相应的子程序 */
```

```
{
```

```
int x,ret;
```

```
printf("输入 1:执行程序 test2.exe\n 2:执行程序 test3.exe\n");
```

```
scanf("%d",&x);
```

```
if(x==1)
```

```
{
```

```

ret=spawnl(P—WAIT,"M8.exe","");
PRET(M8.exe);
return (0);
}
else if(x==2)
{
ret=spawnl(P—WAIT,"M9.exe","");
PRET(M9.exe);
return (0);
}
else return (1);
}

C>TYPE M8.C (M9.C 和它相同)
#include "stdlib.h"
main()          /* 子程序根据随机数返回值,也就是 main() 返回值 */
{
int x;
randomize();
x=random(100);
if(x<50)exit(1); /* 此处 exit 也可以用 return 或 —exit 代替 */
else exit(0);
}

```

14.4 DOS 环境和环境函数

环境是 DOS 用于传递父程序与子程序之间信息的一种机制。不论系统处于哪一个层次上, DOS 都为该层次的程序设置一个“环境区域”。在加载程序本身之前,先向内存申请一个存放环境串信息的内存分配块,称【环境块】。在该环境块内,除了继承父程序的环境信息外,还允许各级程序设置本级使用的环境信息。

【环境信息】是指本级运行时所处的系统状况及某些信息。它由一系列以 0 结尾的 ASCII 码字符串组成。每个串的形式为:

环境变量名=环境参数

环境参数又可称为【环境变量内容或环境变量的值】,它也是一个字符串(但在书写时字符串不用西文双引号括起来)。多个环境变量可构成【环境表】。在系统启动时,初始化模块 SYSINIT 在加载 COMMAND.COM 后(此后,系统便在 COMMAND.COM 控制下),首先在常驻内存区申请一内存块,称【主环境块】,存放 DOS 运行的环境信息串。主环境块至少有 3 个环境字符串:

```

'COMSPEC=C:\COMMAND.COM',0
'PROMPT=',0
'PATH=',0

```

第一个由 DOS 设定(若在 A 盘启动,则为 'COMSPEC=A:\COMMAND.COM',0)。它指示命令处理所在的位置。当程序结束时,便可知到何处再次装载 DOS 的暂驻部分。

第二个是设置系统提示符的形式,由 DOS 的 PROMPT 命令设置。对 DOS 5.0 有

```

prompt = $q      /* 设置提示符为 = */
prompt = $ $      /* 设置提示符为 $ */
prompt = $t      /* 设置提示符为当前时间 */
prompt = $d      /* 设置提示符为当前日期 */
prompt = $p      /* 设置提示符为驱动器名和路径 */
prompt = $v      /* 设置提示符为 MS-DOS 版本号 */
prompt = $n      /* 设置提示符为当前驱动器名 */
prompt = $g      /* 设置提示符为 > */
prompt = $l      /* 设置提示符为 < */
prompt = $b      /* 设置提示符为 ! */
prompt = $—      /* 设置提示符为 — */
prompt = $e      /* 设置提示符为 ESC 码,为一小← */
prompt = $h      /* 设置提示符为没有任何显示,光标回退一格 */

```

等式右边也可以是它们的组合,如 `prompt = pg` 提示符依次是显示驱动器名、路径和 > 符号。当然它们的排列顺序是重要的,例如 `pg` 和 `gp` 是不同的。

第三个指出寻找目录的路径,由 DOS 的 PATH 命令设置。DOS 的 PATH 命令用于设置或显示当前的搜索路径,查找外部命令。例如,你键入

```
C>PATH
```

则可能的显示是

```
PATH=C:\DTP;C:\DOS;C:\LXPC\SYS
```

现在你想执行这三个目录中的某一个文件,譬如 `C:\DOS\my.exe`,那末不管当前所在的目录是什么,你只要键入 MY 则该文件就被执行。搜索是先在当前目录中进行,如未找到,则依次在 PATH 指定的目录 `C:\DTP`、`C:\DOS` 或 `C:\LXPC\SYS` 中寻找。如有基本名同名的三个文件 MY.COM、MY.EXE 和 MY.BAT,则按优先次序 .COM, .EXE, .BAT 执行先找到的一个文件。注意:非这三种类型的文件应该用 DOS 的 APPEND 命令。

DOS 允许用 SET 命令向环境内存块中添加新设置的环境串,或者修改已有的环境串。SET 命令有三种形式:

1. SET

它把环境块中各个变量在屏幕上显示出来。

2. SET 环境变量名 =

删除指定的环境变量。

3. SET 环境变量名 = 环境参数

建立(或修改)所指定的环境变量。

当 DOS 加载一个应用程序时,将先向内存申请一个环境内存块。若申请成功,则将 COMMAND.COM 运行的主环境块复制一份副本到该内存块中,并将其段地址送到程序段前缀 PSP 的位移 + 2CH 处。因此,应用程序可通过编程获得所有环境信息,并可对主环境块的拷贝进行修改或添加。如果在加载用 SET 命令设置了一些环境信息,则这些信息也将被传递给应用程序。

当应用程序作为父程序加载另一个子程序时,则父程序的环境块将被子程序继承。此外,在加载前假定设置了新的环境段,在加载后也会逐级依次被传递。在 DOS 命令下,只能通过 SET、PROMPT 和 PATH 命令来存取和修改环境字符串。而在批处理文件 (*.BAT) 中,除此三条命令外,还可用 % 将环境变量括起来的办法来引用环境变量的值。例如,

```
C>TYPE SETPATH.BAT
```

```
echo off
```

```
SET PATH1=C:\TC
```

```
SET PATH2=C:\213
```

```
SET PATH3=%PATH% /* %PATH%相当于PATH的环境参数,下同 */
```

```
C>TYPE SWAPPATH.BAT
```

```
ECHO OFF
```

```
IF %1==1 PATH=%PATH1% /* %1~%9 参数对应你在命令行上键入的参数 */
```

```
IF %1==2 PATH=%PATH2% /* 对应的参数可以是字符串,参见DOS手册 */
```

```
IF %1==3 PATH=%PATH3%
```

如先运行 SETPATH.BAT 设置环境变量,以后只要在 DOS 下键入命令 SWAPPATH 1 (这里数字 1 便是对应 %1 的参数)或 SWAPPATH2 就可将路径转换成 PATH1 或 PATH2,当键入 SWAPPATH 3 时,则恢复到运行 SETPATH.BAT 以前设置的路径。注意:若现在只设了一条路径

```
PATH=C:\213
```

要想增加一个路径 C:\TC,则不能简单地键入

```
PATH=C:\TC
```

因为那样最后只有一个路径,即 PATH=C:\TC,而前一路径 PATH=C:\213 将消失。所以应键入

```
PATH=C:\213;C:\TC
```

这个过程也可这样实现,即在 DOS 下先建立一个文件:

```
C>TYPE ADDPATH.BAT
```

```
PATH=%PATH%;%1
```

然后在 DOS 下执行 ADDPATH:

```
C>ADDPATH C:\TC
```

结果有 PATH=C:\213;C:\TC。

与批处理文件不同,对可执行程序(.COM或.EXE)是不能直接存取主环境块的,而只能存取它的拷贝。程序段前缀 PSP 的偏移量 2CH 处的内容是可执行程序的环境空间的段地址,环境空间的偏移量是 0。各个环境变量以一字节的 00H 相间隔,在最后一个环境变量的后面是两字节的 00H,以表示整个环境的结束。环境串的内容无任何限制,它对操作系统本身没有任何影响。换句话说,环境块的信息甚至可以是 DOS 无任何意义的关键字和参数。然而,用户程序却可编程检测到并解释它。因为,环境块的段地址被复制到程序段前缀 PSP + 2CH 处,这对应用程序是已知的。但应注意:内存控制块及程序段前缀就在程序的环境拷贝的上面,如果程序在修改它的环境时不慎破坏了这两个数据结构,则会造成严重后果。此外,所进行的修改不得超出为用户程序留出的环境空间,以免破坏其它数据结构。DOS 对环境空间的限制是 160 个字节,但 PATH 和 COMSPEC 变量至少要占用 29 字节,剩下的由用户程序使用。如要扩充此空间,对 DOS 3.2 及以上版本可在 CONFIG.SYS 中增加一行:

```
SHELL=COMMAND.COM/E:xxxxx/P
```

这里 SHELL 是 DOS 命令;xxxxx 是十进制整数,以字节为单位,用来指定环境空间的

大小 (160 ~ 32767); /P 是表示在装入 COMMAND.COM 后立即执行 AUTOEXEC.BAT.

—1 char * —Cdecl getenv(const char * name);

它从当前环境表中寻找环境变量名 name, 返回指向环境变量 name 的值的指针。此指针在以后的调用中可以被重写。如果调用成功, 返回非 0 值。如果指定的 name 在环境中没有定义, 它返回 0。

适用于 UNIX 系统。

C>TYPE ENV1.C

```
#include "stdlib.h"
```

```
#define ENV printf("env= %s\n", env)
```

```
main()
```

```
{
```

```
char env1[100];
```

```
char * env;
```

```
env=getenv(env1);
```

```
ENV;
```

```
env=getenv("PATH");
```

```
ENV;
```

```
}
```

/* 假定在 DOS 下用 C>SET 命令后得环境表:

```
COMSPEC=C:\COMMAND.COM
```

```
DTP=C:\DTP;
```

```
PATH=C:\DTP;C:\DOS;C:\LXPC\SYS
```

```
TEMP=C:\DOS
```

```
PROMPT= $p$g
```

则程序输出: env=(null)

```
env=C:\DTP;C:\DOS;C:\LXPC\SYS
```

*/

—2 int —Cdecl putenv (const char * name);

本函数用于对当前环境设置新的环境变量, 其中 name 相当于字符串

环境变量名=环境参数

当等式右边的环境参数为空时, 该环境变量即被删除。如果调用成功, 它返回 0, 否则返回 -1。

它相当于 DOS 的 SET 命令, 但是它又和 SET 命令有所不同。它只对执行程序起作用, 或者说, 它虽然改变了本程序的执行环境, 但不改父进程的环境。可用程序 ENV2.C 作试验。先在 DOS 下键入

```
C>SET AAA=test
```

再键入

```
C>SET
```

便看到环境表中增加了 AAA=test, 执行程序

```
C>ENV2
```

程序输出

1=test

2=

然后再回到 DOS 下用 SET 命令检查（或再执行一次 ENV2），发现环境表中环境变量 AAA 依然存在。

C>TYPE ENV2.C

```
#include "stdlib.h"
```

```
main()
```

```
{
```

```
static char *p;
```

```
p=getenv("AAA");
```

```
printf("1=%s\n",p);
```

```
putenv("AAA=");
```

```
p=getenv("AAA");
```

```
printf("2=%s\n",p);
```

```
}
```

C>TYPE ENV3.C

```
#include "stdio.h"
```

```
#include "dos.h"
```

```
#include "stdlib.h"
```

```
void mode(int mode—code) /* 设置显示模式 */
```

```
{
```

```
union REGS r;
```

```
struct SREGS s;
```

```
—AH=0;
```

```
—AL=mode—code;
```

```
geninterrupt(0x10);
```

```
}
```

```
void setmouse()
```

```
{
```

```
union REGS mr;
```

```
struct SREGS s;
```

```
mr.x.ax=0;
```

```
int86x(0x33,&mr,&mr,&s);
```

```
mode(3);
```

```
/* 文本方式 */
```

```
if(mr.x.ax<=0)
```

```
{
```

```
printf(" 系统没有安装鼠标驱动程序!");
```

```
/* 注意:不管机上有没有装鼠标,鼠标驱动程序都能装入内存! */
```

```
exit(1);
```

```
}
```

```
printf("OK! 已经安装了鼠标驱动程序!");
```

```
}
```

```
main() /* 本程序可测试:鼠标程序安装否,环境变量设置否 */
```

```
/* 但不能测知机上是否有鼠标,由此可决定是否用鼠标 */
```

```

{
char *p; /* getenv() 返回指向环境变量 MOUSE 的指针, */
        /* 如环境中无 MOUSE 变量,则返回空指针 */
        /* 注意:变量名 MOUSE 跟鼠标驱动程序名 MOUSE.COM 无关 */
char *path;
if(((p=getenv("MOUSE")) != NULL) && (! strcmp(p,"YES",3)))
{
printf("一般在机上应装有鼠标,且已装入鼠标驱动程序后,");
printf("才在 DOS 下提示符下键入 SET MOUSE=YES \n");
printf("这里 MOUSE 是环境变量(字母不分大小写),其值=YES(有大小写区别)\n");
delay(2500); /* 延迟 */
mode(18); /* 图形方式 */
setmouse(); /* 安装鼠标驱动程序 */
}
else
{
printf("未在 DOS 下键入 SET MOUSE=YES,故即使有鼠标也没有使用!\n");
exit(1);
}

/* 将环境变量 PATH 加到当前环境中去 */
putenv("PATH=C:\\TC\\INCLUDE;C:\\");
path=getenv("PATH");
printf("\nPATH: %s\n",path);
}

```

—3 全局变量 extern char **environ;

因为 * 运算符的结合性是从右到左,所以 **environ 相当于 *(*environ),故 environ 可以认为是一个指向指针的指针。当一个程序开始运行时,它指向环境块。当 main() 的第三个参数 env 存在时,它的设置和 environ 相同。在调试程序 ENV5.C 时不难用调试表达式

```

env
environ

```

观察,它们都有值 DS:05CA。或者说,参数 env 也可以通过 environ 引用。在执行程序时环境块的大小和位置可能改变,而 environ 总是自动调整以指向正确的位置。

C>TYPE ENV4.C

```

#include "stdio.h"
extern char **environ; /* 不能写成 extern char *environ[] */
main()
{
int k;
for(k=0;environ[k] != NULL;k++)
printf("env[%d]: %s\n",k, environ[k]);
}
/* env[0]:COMSPEC=C:\COMMAND.COM
   env[1]:DTP=C:\DTP;
   env[2]:PATH=C:\DTP;C:\DOS;C:\LXPC\SYS

```

```

    env[3];TEMP=C:\DOS
    env[4];PROMPT=$p$g */

C>TYPE ENV5.C
#include "stdio.h"
#include "process.h"
#include "stdlib.h"
extern char ** environ; /* 仅管 stdlib.h 中已包含了它 */
main(int argc,char * argv[],char * env[])
{
    int k;
    char * e[]={"DUMMY=YES"};
    char * a[]={"rrr","TTTT"};
    for(k=0;env[k] != NULL;k++)
    {
        printf("env[%d]:%s\n",k, environ[k]);
    }
    spawnvpe(P_WAIT,"env6.EXE",a,e);
    putenv("DDDD=TC");
    for(k=0;env[k] != NULL;k++)
    {
        environ[k]=env[k];
        printf("env[%d]:%s {%s}\n",k, environ[k],env[k]);
    }
}
/* env[0];COMSPEC=C:\COMMAND.COM
   env[1];DTP=C:\DTP ;
   env[2];PATH=C:\DTP;C:\DOS;C:\LXPC\SYS
   env[3];TEMP=C:\DOS
   env[4];PROMPT=$p$g
   DUMMY=YES
   rrr
   TTTT
   env[0];COMSPEC=C:\COMMAND.COM {COMSPEC=C;\COMMAND.COM}
   env[1];DTP=C;\DTP ; {DTP=C;\DTP ;}
   env[2];PATH=C;\DTP;C;\DOS;C;\LXPC\SYS {PATH=C;\DTP;C;\DOS;C;\LXPC\SYS}
   env[3];TEMP=C;\DOS {TEMP=C;\DOS}
   env[4];PROMPT=$p$g {PROMPT=$p$g}
   env[5];DDDD=TC {DDDD=TC} */

C>TYPE ENV6.C
#include "stdio.h"
main(int argc,char * argv[],char * env[])
{
    int x;
    for(x=0;env[x] != NULL;x++)
        printf("%s\n",env[x]);
}

```

第十五章 驻留内存的帮助工具文件 THELP.COM

THELP.COM 程序是一驻留内存的 Turbo C 或 Turbo Pascal 在线求助工具。尽管在集成环境下也可使用 THELP(先驻留 THELP,后启动 TC.EXE),但在集成环境里按 F1 键便可得到帮助,因此一般不用它。当使用命令行编译器(或别的编辑器),以及使用独立的 Turbo Debugger 调试器时(先驻留 THELP,后启动 Turbo Debugger),本工具将提供诸多方便。注意:在当前目录里一般应有 THELP.TCH 文件存在。THELP.COM 本身需要约 8K 内存和另加多至 32K 的交换文件。如果不用交换文件则需 40K 内存。

在 DOS 提示符下键入

C>THELP ?

便可得到使用 THELP.COM 的帮助信息: Turbo Help Version 1.0

USAGE: THELP [options]

note: options must be separated by spaces!

/B Use BIOS for video
/C #xx Select color; # =color number, xx=hex color value (see THELP.DOC)
/Dname Full path for disk swapping (implies /S1)
/Fname Full pathname of help file
/H,/?/? Display this help screen
/Kxxyy Change hotkey: xx=shift state, yy=scan code
/Lxx Force number of rows on screen; xx=25,43,50
/M+ ,/M- Display help text: on monochrome screen(+), on default screen(-)
/Px Pasting speed; 0=slow, 1=medium, 2=fast
/R Send options to resident THELP
/Sx Default Swapping Mode: 1=Use Disk, 2=Use EMS, 3=No Swapping
/W Write Options to THELP.COM and exit

15.1 语法

一 用法

USAGE: THELP [options]

用法:在 DOS 提示符下键入:THELP□[选项...]

note: options must be separated by spaces!

注意:各选项之间必须用空格符分开!另外,不宜在汉字操作系统下使用,因它要驻留内存,驻留时如果内存不够会出现死机。

二 选项

■/B

Use BIOS for video (紧跟选项“/B”后的选项英文说明,下面类同)

在显示帮助内容时数据的存取均通过 BIOS 的 INT 10H 功能调用 (参见《视频函数》一章)。一般情况 THELP 直接将显示内容 (文本信息) 写到视频 RAM 中。使用本选项时,如再使用 /M 开关,则 /M 开关无效;在 /B 生效时 (用 /B+, 也是缺省值。/B- 为 /B 失效), 不能使用双监视器。

■ /C#xx

Select color: # = color number, xx = hex color value (see THELP.DOC)

选择帮助屏幕的颜色。对不同的显示器,选用时一般经过试验,找到自己认为满意的颜色。

-- 表示颜色数(1~8):

= 1 彩色、正常文本

= 2 单色、正常文本

= 3 彩色引用页、顶/底描述线

= 4 单色引用页、顶/底描述线

= 5 彩色边框

= 6 单色边框

= 7 彩色、当前参考页选择

= 8 单色、当前参考页选择

xx -- 两位控制前景或背景的十六进制颜色值

第一个 x (前景)

0 = 黑

1 = 兰

2 = 绿

3 = 青

4 = 红

5 = 洋红

6 = 棕

7 = 灰

第二个 x (背景)

0 = 黑

1 = 兰

2 = 绿

3 = 青

4 = 红

5 = 洋红

6 = 棕

7 = 灰

8 = 加强黑

9 = 加强兰

A = 加强绿

B = 加强青

C = 加强红

D = 加强洋红

E = 棕(黄)

F = 灰(白)

把颜色值加上 80H 可能闪烁

注意:使用 /M 将强制单色显示。

■ /Dname

Full path for disk swapping (implies /S1)

使用 THELP.COM 时将产生两个交换文件:THELP.SW1 和 THELP.SW2,它一般在 THELP.COM 所在的目录下 (对 DOS 3.X 版本);或者在 C:\ 目录下 (对 DOS 2.X 版

本)。如用本开关,可指定交换文件的完整路径名。如使用了本选项,则交换文件一定到磁盘,而不管是否使用了 /Sx 开关,也即不再检查是否使用了 EMS。因此,可以说此时相当于选了 /S1 开关。

注意:THELP.SW1 和 THELP.SW2 产生后不要人为地删掉,否则很容易出现死机。

■ /Fname

Full pathname of help file

指定求助文件的完整路径名(缺省值为你拷 TC 系统的目录。例如 C:\TC\TCHELP.TCH)。虽然,对 TC 的求助文件是指 TCHHELP.TCH,但也可以用 PASCAL 的求助文件 TURBO.HEL。

■ /H,/? ,?

Display this help screen

使用开关 /H(或 /?,或 ?)可以在屏幕上显示 THELP 的命令行上的可选项的列表,即显示帮助屏幕。使用本选项时 THELP.COM 不会被装入内存。

■ /Kxxyy

Change hotkey;xx=shift state, yy=scan code

如果你未用本选项,那么在 THELP 启动并驻留内存后显示

To activate THELP, Press 5 on the numeric keypad

即当你按数字键盘上的数字键 5 (Shift 交换状态码 xx = 00h,扫描码 yy = 4ch)后,THELP 便被激活,屏幕显示

TURBO C HELP

Page 100

Help on Help

Menu commands

Keyboard hot keys

C Language

Header Files

Keywords

Precedence

Editor

Cursor movement

Insert & Delete

Block commands

Miscellaneous

Installation(TCINST)

Command—line options

Debugger

Graphics

但使用本选项后,例如,用

C>THELP /K021E

后,就可用 SHIFT+A 键激活 THELP 了。

xx——是变换状态码(十六进制)

xx=01h 右 SHIFT 键

xx=02h 左 SHIFT 键

xx=04h CTRL 键

xx=08h ALT 键

yy——扫描码,其值为:

键名	扫描码	键名	扫描码	键名	扫描码	键名	扫描码
A	1eh	N	31h	0	0bh	F1	3bh
B	30h	O	18h	1	02h	F2	3ch
C	2eh	P	19h	2	03h	F3	3dh
D	20h	Q	10h	3	04h	F4	3eh
E	12h	R	13h	4	05h	F5	3fh
F	21h	S	1fh	5	06h	F6	40h
G	22h	T	14h	6	07h	F7	41h
H	23h	U	16h	7	08h	F8	42h
I	17h	V	2fh	8	09h	F9	43h
J	24h	W	11h	9	0ah	F10	44h
K	25h	X	2dh			F11	85h
L	26h	Y	15h			F12	86h
M	32h	Z	2ch				

■/Lxx

Force number of rows on screen: xx=25,43,50

设置屏幕显示行。xx 可取 25,43,50 三种值。本选项强制 THELP 选用用户指定的显示值,而不用通过调用 BIOS 调用得到的值。

■/M+ ,/M-

Display help text: on monochrome screen(+), on default screen(-)

对于双显示器,用 /M+ 指定 THELP 在单色显示器中显示帮助文件的内容。对彩显一般用缺省值(即 /M-)。注意:/M 与 /B 的关系。

■/Px

Pasting speed: 0=slow, 1=medium, 2=fast

选择 THELP 合适的工作速度,以便编辑程序能正确接收从键盘输入的字符。

x=0 只有当缓冲区空时才接收一单个字符;

x=1 较快,每一时钟周期最多接收 4 个字符;

x=2 最快(缺省值),每一时钟周期接收尽可能多的字符。若是在集成环境下使用本方式时,必须把编辑器内状态行上的自动对齐模式(即 Toggle Autoindent,控制编辑行自动缩进的开关)置为 OFF。

■/R

Send options to resident THELP

传送新的命令选择项到驻留在内存中的 THELP 程序中。一般说来,如果一开始你已用一些选择项将 THELP 驻留在内存,那么一般你不能用同样的方法再启动 THELP,否则会显示

Turbo Languages Resident Help has already been loaded

表示不能重复装入。然而常会碰到要改变驻留的 THELP 中的选项,怎么办?一种方法是用

C>THELP /U

先将内存中的 THELP 清除,然后再启动,这当然是可以的,但是更好的方法是使用本选项。例

如,利用

```
C>THELP /K021E
```

启动 THELP 后还想改显示颜色,可接着用

```
C>THELP /C526 /R
```

屏幕显示

```
New Parameters Sent to Resident THELP
```

表示新的参数已传给驻留 THELP 程序了。结果两种选项同时起作用了。

注意:如 THELP 还未驻留内存,则不能用此开关,否则显示

```
THELP has not been loaded!
```

表示 THELP 没有被装入内存。

■ /Sx

Default Swapping Mode: 1=Use Disk, 2=Use EMS, 3=No Swapping

在指定 /Sx 时给出缺省交换模式,其中 x=1 使用磁盘; x=2 用 EMS; x=3 不交换。

THELP 启动时如果使用了数据交换文件 (*.SW?),则至少要占用 8K 字节内存;如果采用不交换数据形式(不产生 *.SW?),则需要占用 40K 字节的内存。

■ /U

Remove THELP from memory

使用本选项后,屏幕显示

```
THELP has been removed from memory
```

表示内存中已不再有 THELP 程序驻留了,而且交换文件 THELP.SW1 和 THELP.SW2 也被删除。注意:如果在 THELP.COM 驻留内存后又有别的程序驻留内存,那么在将 THELP 从内存移去前必须先将那些程序移去。

如果使用本选项前 THELP.COM 并没有驻留内存,则显示

```
THELP has not been loaded!
```

■ /W

Write Options to THELP.COM and exit

将选择项写到 THELP.COM 文件中,结果这个新的 THELP.COM 启动时便包含了早先的选择项。注意:使用本选项时不包括 /R 选项,其它选项均可用;另外,使用此选项时 THELP.COM 一定未驻留在内存。如已驻留在内存,不起作用(新参数将装入内存中的 THELP,而不是进入磁盘上的 THELP)。例如,THELP 未驻留在内存,键入

```
C>THELP /K021E /W
```

则显示

```
New Parameters Written to THELP.COM
```

表示新参数已装入 THELP。下次用

```
C>THELP
```

驻留内存后,便可用 Shift-A 激活它。

15.2 在 THELP 激活后所能使用的键

—1 Arrow keys: Move Cursor Between Topics

使用移动光标的箭头键能使高亮度光条在当前求助屏幕中的条目间移动。

—2 PgUp/PgDn, Next/Previous Page(if any)

这两个键用于翻屏幕显示页。能否翻页可看看屏幕窗口底端有否这两个键名显示。

—3 F1: Help index

按 F1 键出现帮助索引窗口。

—4 Alt-F1: Review past 20 help pages

按 ALT-F1 键显示将退回到前一帮助窗。每按一次, 往回退一个帮助窗口, 最多可连续退 20 次。

—5 <ENTER>: Select highlighted Topic

按回车键表示高亮度光条所在条目的帮助内容将出现。

—6 <ESC>: End Help

按 ESC 键结束帮助, 退回 DOS 提示符下。

—7 <F>: Select New Help File

按 F 键或 f 键后显示

New Help File:

这时你应输入新的帮助文件名, 例如 TURBO PASCAL 的帮助文件 TURBO.HLP。文件名可带路径名。随后新文件的内容将被读到内存, THELP 将重新对新文件的帮助索引进行初始化。如果新的帮助文件不存在, THELP 将发出两声响后, 返回到原来的帮助屏幕。

—8 <J>: Jump to New Help Page

按 J 或 j 键后显示

Jump To Page:

这时你应输入帮助页号 (页号对 TCHELP.TCH 最大是 1058, 不能输入 1059, 否则可能死机。超过 1059 不起作用。其它帮助文件可用页数也可测试, 最大不超过 9999)。修改页数可用 BackSpace 退格键。按回车键将显示指定页的内容。以下为 TCHELP.TCH 中某些页的内容索引。如

查询内容	页号
Add watch	264
Always	280
Arguments	123
Auto dependencies	256
Backup files option	144
Block commands	1039
.....	
Help on Help	1
..... Keyboard hot keys	1057
..... Main menu	1058
..... short	1019
signal.h	66
signed	1019
.....	

—9 <K>: Select New Keyword

如果你已经知道了帮助文件中某一字（高亮度条目可达到的字），那么当你键入 K 键或 k 键后，显示

Enter Keyword to Find:

此时你输入此字（最多 40 个字符），如果字正确，则立即显示有关此字的帮助内容。

如果未找到，THELP 响两声后返回原屏幕。

—10 <I>: Insert keyword into Application

当键入 I 键或 i 键后，当前屏幕的高亮度条目所在的字将存入键盘缓冲区（字将在屏幕上显示），并退出 THELP（但是 THELP 仍驻留在内存）。例如，先前只在 DOS 情形下激活 THELP，按 I 键后退出 THELP 又返回 DOS 提示符下，且字也显示在提示符后；又例如，先将 THELP 驻留后启动 TC.EXE，按 5 字键激活 THELP，又按 I 键后，高亮条所在字将进入编辑器内；当用 Turbo Debugger 调试程序时，此选项有作用。

—11 <P>: Paste Help Screen into Application

按 P 键或 p 键后，当前屏幕的所有字产生像按 I 键一样（它只是一个字！）一样的结果。

—12 <S>: Save Help Screen to Disk

按一下 S 键或 s 键，当前屏幕的内容便存储在当前目录中的 THELP.SAV 文件中。如果此文件原先没有，则创建它，然后写入；如果此文件已有，则将内容追加到其后。写入的内容一般包括两部分：一是当前屏幕的内容，二是高亮度条目可达的字的可查询的页号。

—13 CTRL-F1: Keystrokes for Turbo Help

按 Ctrl-F1 键后显示以上 1~12 项的英文提示，以指示你操作。

15.3 错误信息

1. Could Not Find Help File:

不能找到帮助文件。

2. Invalid Help File:

无效的帮助用户文件。

3. Wrong help file version:

错误的帮助文件版本。

4. THELP cannot be loaded because another TSR currently resident on this system is using the same hotkey!

THELP 不能装入，原因是另一常驻内存系统用同样的热键。

5. Unknown Operating System!

不认识的操作系统。

6. Invalid Swap Drive Selected:

无效的交换选择。

7. Invalid Command Option

无效的命令选择。

8. Divide error

分隔错误。

9. Abnormal program termination
程序非正常终结。

第十六章 集成开发环境和缺省参数设置

在 TC 集成环境 (Integrated environment) 里,程序员可以直接编辑和建立 C 源程序,跟着进行编译、连接及调试程序,最后执行程序等。换句话说,程序员只要在集成环境下,按照 TC 主屏幕提示按动某些键,TC 便自动进入相应的环境 (启动编辑器、调试器、编译器、连接器或规划组装器等),而程序员无须知道像要完成某操作应启动 TC 系统中某个特定程序这样烦琐的事情。对初学者应先学会利用集成环境开发应用软件。

执行 TC.EXE 程序就可使用集成环境,TCINST.EXE 是用来设置 TC.EXE 缺省参数 (如屏幕大小、编辑命令、菜单颜色或目录等) 的实用程序。

为方便对照,本章将 TC.EXE 与 TCINST.EXE 结合起来叙述。建议读者在实际操作前应先浏览全章,然后边对照 TC 屏幕边熟悉各菜单的功能和使用方法。

16.1 怎样进入集成环境

进入 TC 集成环境 (在不发生误会时我们也说进入 TC,或更简单一点就说 TC 如何如何) 最简单的方法是在 DOS 提示符下键入 C>TC 后便进入了 TC 的集成开发环境。这时屏幕顶端显示

```
File Edit Run Compile Project Options Debug Break/Watch
```

这就是 TC 主菜单。更广泛一些,正确进入 TC 要明确两件事:

(一) 运行磁盘上应有那些文件

若运行盘是硬盘,并且你已用 TC 的安装程序将所有文件装入硬盘,那么只要正确对包含文件 (如 *.H) 目录、库 (如 *.LIB) 文件目录、TC 系统文件 (如 TC.EXE) 目录及输出文件目录等选择好 (后面将详细叙述) 就行了。

若用软盘驱动器运行,重要的是上述几种目录内的文件要在当前磁盘上。应当指出,有些文件是必不可少的,如 TC.EXE,运行某一种存储模式对应的库文件及涉及的包含文件等也是必不可少的。反之,一些暂不用的文件像涉及其它存储模式的文件,未用到的包含文件,帮助文件等可不要。显然,磁盘容量大,例如 1.2 兆盘则可多装一些文件,那么用 1.2 兆驱动器就可能好像在用硬盘一样;而对于 360K 磁盘和用 360K 软驱,由于磁盘容量小,则要受许多限制,使用不怎么方便。当然,如果你的机器有两个软驱,则可以将需用运行文件分装两张盘上,再通过主菜单 Options 项选择正确的目录便行了。值得指出的时,机器有虚拟盘 (指有 1024K 内存而又用 VDISK.SYS 进行管理的虚拟盘可达 384K,相当一张 360K 软盘),也是可以利用的。同样,对具有扩页内存的机器,扩页内存也是可用的。

(二) TC 命令行开关说明

语法:TC [☐编辑文件名] [☐/C 配置文件名] [☐/B 或☐/M] [☐/D]

说明:方括号不是命令行的组成部分,它只是表示其内内容是任选的。

—1 编辑文件名

允许你在启动 TC.EXE 时将编辑文件自动装入编辑器。文件的缺省扩展名是 .C,因此

当文件名未带扩展名时, TC 自动认为有扩展名 .C。文件名也可以带 DOS 通配符 (* 和 ?)。

—2 /C 配置文件名 或 /c 配置文件名

注意/C 和配置文件名之间没有空格符。

如未用 /C 开关, TC.EXE 启动时会自动在当前目录中去查找名为 TCCONFIG.TC 的缺省配置文件; 若未找到, 再到 Turbo C 目录中找它; 若还未找到, 则 TC.EXE 将以其内部缺省值设置启动(有关配置文件的细节参见—6—3—3)。

配置文件的缺省扩展名为 .TC。配置文件是由 TC.EXE 或 TCINST.EXE 生成或指定的, 任何其它方法生成的文件都是无效的。无效的配置文件不会起作用。

—3 /B 或 /b

该开关使 TC 重新编译 project 里的所有文件 (相当于集成环境里 Compile/Build all), 并在标准设备上打印编译信息, 然后返回 DOS。它允许你在批处理文件 (.BAT) 里调用 TC.EXE 对规划文件 (.PRJ) 或主 C 文件处理。下面以一实例说明之。先用 C>TC 进入集成环境, 然后选择

Options/Environment/Config auto save On

Project/Project name MCALC.PRJ

Options/Directories/Output directory : D

尔后用 Alt-x 退出集成环境, C 盘上生成配置文件 TCCONFIG.TC。现在你键入 C>TC □/B 便发现 TC 不断对 C 盘上的文件

MCALC.C、MCPARSER.C、MCDISPLY.C、MCINPUT.C、MCOMMAND.C、MCUTIL.C

重新全部编译和连接, 最后自动退出集成环境, 显示

C>TC /B

Compiling C:\TC\MCALC.C;

Compiling C:\TC\MCPARSER.C;

Compiling C:\TC\MCDISPLY.C;

Compiling C:\TC\MCINPUT.C;

Compiling C:\TC\MCOMMAND.C;

Compiling C:\TC\MCUTIL.C;

Linking D:\MCALC.EXE /* 执行文件和目标文件均在 D 盘上 */

C>

表明 TC 已完全退出而进入 DOS 环境, 现在可以执行 DOS 命令了! 从上可见, 你可以将 TC /B 装入到批处理文件里执行。

上面我们用的是缺省的配置文件, 你也可以用你自己早先用 TC 生成的文件名 (例如可以为 MYCONFIG.TC), 则可用

tc□/cMYCONFIG.TC□/b

命令行。当然, 配置文件里一定要有规划文件名或主 C 文件名 (在集成环境下可用

Compile/Primary C file;

设置)。

—4 /M 或 /m

它和 /B 类似, 不同点在于只编译连接那些过时的文件, 对没有过时文件则不编译。(相当于集成环境里的 Compile/Make EXE file 命令)。

—5 /D 或 /d

它告诉 TC 使用两个显示器。它首先检查是否有配有两个显示器,如没有则本命令不起作用。

在运行或调试程序时可用双监视器模式,或在使用 DOS 命令解释程序时使用。

如果系统配有两个显示器, DOS 只把其中一个当作活动监视器。用 DOS 的 MODE, 命令可以开关这两个显示器(例如:MODE C080, MODE MONO 等)。

处于双监视器模式时, TC 主屏一般出现在不活动显示器上,而程序的输出则在活动显示器上。这样,当在一个显示器上, DOS 提示符下键入 TC /D 时, TC 主屏将出现在另一个显示器上。当想在一个特定的显示器上测试程序时,必须先退出 TC,把活动显示器开关打到想测试用的显示器上,然后再发一次 TC /D 命令,于是程序输出将在键入 TC 命令的那一个显示器上出现。

注意:在 DOS 命令解释程序里不要改变当前活动显示器(例如使用 MODE 命令);用户程序中不能有直接访问不活动显示的视屏口内容,否则可能会产生不可预料的结果;当运行或调试显示使用双显示的程序时,不得使用打开双显示器的开关。

16.2 集成环境中的热键 (hot keys)

【热键】是指这样一些键,在集成环境下,不论在任何地方只要按了这些键中的任一 键,该键事先规定的固定功能调用便立即起作用。

注意:当集成环境要求你按照其提示选择指定键时,只有按了指定键后热键才起作用。

下面只对热键功能作简要叙述,不足之处参见集成菜单部分。

—1 F1

激活帮助窗口,即在编辑窗口内出现帮助窗口。在 TC 任何菜单任何地方,当按了 F1 键,与高亮度菜单项相关的帮助信息便出现;在编辑源程序时按了 F1 键,显示与编辑内容(光标所在处)相关的帮助信息。此时如又按了 F1 键,则出现帮助索引窗口。可以用光标键移动光标到另一关键字上,按回车键便得到与此相关的帮助信息。还可以用 Home 或 End 键将光标移动到屏幕最头上或最后的关键字上,然后按 F1 键获帮助信息(注意:部分帮助信息可能并不正确)。

—2 F2

将编辑器里的源文件无条件**【存盘】**,即写入磁盘。写盘时如果盘上有同名文件,则立即被改写。注意:改写前 TC 不会发出任何提示。一按 F2 键,则不管在任何时候,即使在运行程序(RUN)时也会立即写盘。

—3 F3

将指定文件读入编辑器内,或称**【装入文件】**。

—4 F4

执行程序时程序运行到编辑光标所在行便暂停。

—5 F5

将**【活动窗口】**(或称激活的窗口,例如编辑窗口被激活,则可以编辑窗口内的源程序)放大到整个屏幕,或将活动窗口缩小到其缺省的大小(参见本章稍后部分关于用 TCINST.EXE 设置 TC.EXE 缺省值的叙述)。

—6 F6

是一**【反复键】**,或称**【乒乓开关】**,当首次按它时关闭活动窗,即程序员不能对它进行操作;

接着再按它一次,活动窗被激活,或称打开;接着又按它,活动窗又被关闭,如此等等。TC 当前屏幕上一般只显示两个窗口(未按 F5 键):编辑(Edit)窗与监视(Watch)窗,或编辑窗与消息(Message)窗。在同一时刻,始终只有一个窗口被激活,或者说被打开。

—7 F7

在调试程序时,每按它一次,程序执行一句(可执行的语句)。当一句涉及调用函数时,其可跟踪进入被调用函数内部。

—8 F8

它的功能与 F7 键有点类似,在调试程序时,每按它一次,程序也要执行一句(可执行的语句)。但当一句涉及调用函数时,它并不跟踪进入被调用函数内部。

—9 F9

对那些【过时文件】,即那些与当前源文件相关的文件与源文件相比,在时间(文件生成时间)已过时(引起原因可能是原文件已被修改),进行重新编译和连接,最后生成新的执行文件。

—10 Ctrl—F1 (其意思是,按住 Ctrl 键的同时又按 F1 键,下类同)

编辑器调用上下文的帮助内容(只适用于编辑器)。当编辑源文件时,按它后编辑光标所在的单词的帮助内容便可出现。例如,编辑光标在 #include 的任一字符上,一按该键,则有关 #include 的使用方法便出现在屏幕上;又例如,光标在库函数 printf 上,则有关 printf 的帮助信息便展示出来。

—11 Ctrl—F2

中止当前程序的调试,释放分配给程序的空间,关闭打开的文件。一般将重新进入执行程序前的编辑状态。

—12 Ctrl—F3

程序执行后,用它显示调用堆栈情况。如

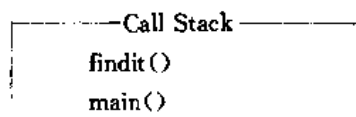


图 16—1

—13 Ctrl—F4

调试程序时评估调试表达式或设置调试变量新值。

—14 Ctrl—F7

增加调试表达式。

—15 Ctrl—F8

设置或消去断点的开关。

—16 Ctrl—F9

编译并运行程序。

—17 Alt—F1

显示上次访问的帮助信息,即回到前级的帮助屏。TC 允许 20 级帮助屏。

—18 Alt—F3

显示【检选表】,允许你从中选择一个文件装入编辑器。

—19 Alt—F6

它的功能分两种情况：一是 TC 屏幕由编辑窗和监视窗组成时，一是由编辑窗和消息窗组成时。

1. 对前者，即在编辑状态时，按该组合键，编辑器内的文件将和检选表中的第二个文件进行交换。也就是说，原为检选表中的第二个文件将进入编辑器，而原在编辑器内的文件将变成检选表的第二个文件。例如有如下检选表

Recent files
P.C
PPP.C
PP.C
NONAME.C
Load file

图 16-2

第一个文件是编辑器里的文件 P.C，第二个文件是 PPP.C。按 Alt-F6 后，PPP.C 便进入编辑器，从而变成检选表中的第一个文件，而 P.C 变成第二个文件。实际上 TC.EXE 在内存中同时存储了这两个文件，因而在交换时并不读盘。

2. 对后者，一般发生在编译或运行程序之后，这时按此组合键使屏幕上信息窗和监视窗交替显示，而编辑窗不动。此时一般要用 F6 键开关激活窗口。当编辑窗被激活时，本组合键能又起前者的作用。

—20 Alt-F7

当编译和连接时发生错误时，按此组合键便把编辑窗内光条定位在当前显示错误的前一错误上（如果前一错误不存在，则高亮度光条消失）。

—21 Alt-F8

当编译和连接时发生错误时，按此组合键便把编辑窗内光条定位在当前显示错误的后一错误上（如果后一错误不存在，则高亮度光条消失）。

—22 Alt-F9

把编辑器内的文件编译成目标文件（*.OBJ）。这种编译并不考虑文件是否过时，也就是说，一按此组合键，编译便开始。

—23 Alt-B

转到 Break/Watch 菜单项。

—24 Alt-C

转到 Compile 菜单项。

—25 Alt-D

转到 Debug 菜单项。

—26 Alt-E

转到 Edit 菜单项。

—27 Alt-F

转到 File 菜单项。

—28 Alt-O

转到 Option 菜单项。

—29 Alt-P

转到 Project 菜单项。

—30 Alt-R

转到 Run 菜单项。

—31 Alt-X

退出 TC, 返回 DOS 状态。

—32 Shift-F10

调用 TC 版本屏幕。版本信息显示后按任一键该消息便消失。

16.3 集成环境中的菜单结构

TC 屏幕顶部显示的是主菜单（一级菜单），高亮度条所在项为激活项，此时一按回车键便进入其子菜单（二级菜单）窗口。利用→←键可以从一个二级菜单移到另一个二级菜单上，而无须退回主菜单再选二级菜单。二级菜单下面的菜单是弹出菜单，可用高亮度显示的大小写字母或利用光标键和回车键来进行菜单选择。按 ESC 键返回上一级子菜单。

符号说明：一个下划线表示一级菜单开始，级数用数字开始。用两个等于号（==）括起来的内容为选择菜单后出现在小窗上的标题，方括号内的内容是可能供选择的缺省值或提示信息。要返回上一级菜单按 ESC 键。

为方便叙述，有时也使用缩写的菜单条目，如

O/C/Errors（顺序按 O、C、E）

表示

Options/Compiler/Errors

对任一窗口和菜单，屏幕底端都有一缺省的功能键操作提示行。

```
—1   File
—1—1   Load      F3
        =Load File Name=
—1—2   Pick Alt-F3
        =Recent files=
—1—3   New
        =Verify['ident' not saved Save? (Y/N)]=
—1—4   Save      F2
—1—5   Write to
        =New Name=
—1—6   Directory
        =Enter File Name=
—1—7   Change dir
        =New Directory=
—1—8   OS Shell
—1—9   Quit Alt-X
        =Verify['ident' not saved Save? (Y/N)]=

—2   Edit Alt-E 或 F10

—3   Run
—3—1   Run          Ctrl-F9
—3—2   Program reset Ctrl-F2
```

- 3—3 Goto cursor F4
- 3—4 Trace into F7
- 3—5 Step over F8
- 3—6 User screen Alt—F5

—4 Compile

- 4—1 Compile to OBJ C;NONAME.OBJ
- 4—2 Make EXE file C;NONAME.EXE
- 4—3 Link EXE file
- 4—4 Build all
- 4—5 Primary C file:
=Primary File=
- 4—6 Get info
=Information=

—5 Project

- 5—1 Project name
=Project Name[*.PRJ]=
- 5—2 Break make on Errors
- 5—2—1 Warnings
- 5—2—2 Errors
- 5—2—3 Fatal errors
- 5—2—4 Link
- 5—3 Auto dependencies Off
- 5—4 Clear project
- 5—5 Remove messages

—6 Options

- 6—1 Compiler
- 6—1—1 Model small
- 6—1—1—1 Tiny
- 6—1—1—2 Small
- 6—1—1—3 Medium
- 6—1—1—4 Compact
- 6—1—1—5 Large
- 6—1—1—6 Huge
- 6—1—2 Defines
=Defined Symbols=
- 6—1—3 Code generation
- 6—1—3—1 Calling convention C [Pascal]
- 6—1—3—2 Instruction set 8088/8086 [80186/80286]
- 6—1—3—3 Floating point Emulation [8087/80287] [None]
- 6—1—3—4 Default char type Signed [unsigned]
- 6—1—3—5 Alignment Byte [word]
- 6—1—3—6 Generate underbars On
- 6—1—3—7 Merge duplicate strings Off
- 6—1—3—8 Standard stack frame On

-6-1-3-9	Test stack overflow	Off	
-6-1-3-10	Line numbers	Off	
-6-1-3-11	OBJ debug information	On	
-6-1-4	Optimization		
-6-1-4-1	Optimize for	Size [Speed]	
-6-1-4-2	Use register variables	On	
-6-1-4-3	Register optimization	Off	
-6-1-4-4	Jump optimization	Off	
-6-1-5	Source		
-6-1-5-1	Identifier length	32	
	=Maximum length=		
-6-1-5-2	Nested comments	Off	
-6-1-5-3	ANSI keywords	Off	
-6-1-6	Errors		
-6-1-6-1	Errors ; stop after	25	
	=Maximum Errors=		
-6-1-6-2	Warnings ; stop after	100	
	=Maximum Warning=		
-6-1-6-3	Display warnings	On	
-6-1-6-4	Portability warnings		
-6-1-6-4-1	A;Non-portable pointer conversion		On
-6-1-6-4-2	B;Non-portable pointer assignment		On
-6-1-6-4-3	C;Non-portable pointer comparison		On
-6-1-6-4-4	D;Conastant out of range in comparison		On
-6-1-6-4-5	E;Constant is long		Off
-6-1-6-4-6	F;Conversion may lose significant digits		Off
-6-1-6-4-7	G;Mixing pointers to signed and unsigned char		Off
-6-1-6-5	ANSI violations		
-6-1-6-5-1	A;'ident' not part of structure		On
-6-1-6-5-2	B;Zero length structure		On
-6-1-6-5-3	C;Void functions may not return a value		On
-6-1-6-5-4	D;Both return and return of a value used		On
-6-1-6-5-5	E;Suspicious pointer conversion		On
-6-1-6-5-6	F;Undefined structure 'ident'		On
-6-1-6-5-7	G;Redefinition of 'ident' is not identical	On	
-6-1-6-5-8	H;Hexadecimal or octal constant too large	On	
-6-1-6-6	Common errors		
-6-1-6-6-1	A;Function should return a value		Off
-6-1-6-6-2	B;Unreachable code		On
-6-1-6-6-3	C;Code has no effect		On
-6-1-6-6-4	D;Possible use of 'ident' before definition		On
-6-1-6-6-5	E;'ident' is assigned a value which is never used	On	
-6-1-6-6-6	F;Parameter 'ident' is never used		On
-6-1-6-6-7	G;Possibly incorrect assignment		On
-6-1-6-7	Less common errors		
-6-1-6-7-1	A;Superfluous & with function or array		Off

-6-1-6-7-2	B; 'ident' declared but never used	Off
-6-1-6-7-3	C; Ambiguous operators need parentheses	Off
-6-1-6-7-4	D; Structure passed by value	Off
-6-1-6-7-5	E; No declaration for function 'ident'	Off
-6-1-6-7-6	F; Call to function with no prototype	Off
-6-1-7	Names	
-6-1-7-1	Code names	
-6-1-7-1-1	Segment name *	
	=CODE Segment Name=	
-6-1-7-1-2	Group name * *	
	=CODE Group Name=	
-6-1-7-1-3	Class name *	
	=CODE Class Name=	
-6-1-7-2	Data names	
-6-1-7-2-1	Segment name *	
	=DATA Segment Name=	
-6-1-7-2-2	Group name * *	
	=DATA Group Name=	
-6-1-7-2-3	Class name *	
	=DATA Class Name=	
-6-1-7-3	BSS names	
-6-1-7-3-1	Segment name *	
	=BSS Segment Name=	
-6-1-7-3-2	Group name * *	
	=BSS Group Name=	
-6-1-7-3-3	Class name *	
	=BSS Class Name=	
-6-2	Linker	
-6-2-1	Map file	Off
-6-2-1-1	Off	
-6-2-1-2	Segments	
-6-2-1-3	Publics	
-6-2-1-4	Detailed	
-6-2-2	Initialize segments	Off
-6-2-3	Default libraries	Off
-6-2-4	Graphics library	On
-6-2-5	Warn duplicate symbols	Off
-6-2-6	Stack warning	On
-6-2-7	Case-sensitive link	On
-6-3	Environment	
-6-3-1	Message Tracking	Current File [Off] [All Files]
-6-3-2	Keep messages	No [Yes]
-6-3-3	Config auto save	Off
-6-3-4	Edit auto save	Off
-6-3-5	Backup files	On
-6-3-6	Tab size	8

```

=Editor Tab Size=
--6-3-7      Zoomed windows      Off
--6-3-8      Screen lines
--6-3-8-1    25 line display
--6-3-8-2    43/50 line display
--6-4        Directories
--6-4-1      Include directories : C:\TC\INCLUDE
=Include Directories=
--6-4-2      Library directories : C:\TC\LIB
=Library Directories=
--6-4-3      Output directory :
=Output Directory=
--6-4-4      Turbo C directory :
=Turbo C Directory=
--6-4-5      Pick file name :
=Pick File Name[TCPICK.TCP]=
--6-4-6      Current pick file :
--6-5        Arguments
=Command Line Parameters=
--6-6        Save options
=Config File[TCCONFIG.TC]=
--6-7        Retrieve options
=Config File[* . TC]=

--7    Debug
--7-1      Evaluate      Ctrl-F4
=Evaluate=
=Result=
=New value=
--7-2      Call stack      Ctrl-F3
=Call Stack=
--7-3      Find function
=Enter subprogram symbol=
--7-4      Refresh display
--7-5      Display swapping  Smart
--7-5-1    None
--7-5-2    Smart
--7-5-3    Always
--7-6      Source debugging  On
--7-6-1    On
--7-6-2    Standalone
--7-6-3    None

--8    Break/Watch
--8-1      Add watch      Ctrl-F7
=Add Watch=

```

—8—2	Delete watch	
—8—3	Edit watch	
	=Edit Watch=	
—8—4	Remove all watches	
—8—5	Toggle breakpoint	Ctrl—F8
—8—6	Clear all breakpoints	
—8—7	View next breakpoint	

16.4 用 TCINST.EXE 程序设置 TC.EXE 参数缺省值

TCINST 工具主要用来改变 TC 集成环境 (TC.EXE) 使用的缺省参数值, 比如屏幕大小、编辑命令、菜单颜色和缺省目录等。

启动 TCINST 的方法有以下几种:

通常在当前子目录 (例如为 C:\TC\) 里有 TCINST.EXE 和 TC.EXE 两个程序, 则在 DOS 提示符下直接键入 TCINST 即可运行 TCINST.EXE 程序。它一般有几种方法启动:

法 1 C>TCINST

或 C>TCINST TC.EXE

或 C>TCINST TC

在 TCINST.EXE 所在目录中应有 TC.EXE 文件, 否则 TCINST 在搜索 TC.EXE (Searching TC.EXE...) 后在屏幕显示找不到 (Can't find 'TC.EXE')。找到 TC.EXE 后它就读

TC.EXE 文件 (Reading TC.EXE...), 然后显示彩色的 (缺省方式) 安装菜单:

Installation Menu	
Compile	编译菜单
Project	规划菜单
Options	选择项菜单
Debug	调试菜单
Editor commands	编辑器命令菜单
Mode for display	显示模式菜单
Set Color	设置颜色菜单
Resize Window	设置窗口大小菜单
Quit/Save	退出 TCINST 菜单

选择菜单命令有两种方法, 一是按菜单项第一个高亮度大写字母后回车, 一是使用 ↑ ↓ 两键上下移动亮条到所需的菜单项后回车, 对应菜单项即被选中。任何时候按 ESC 键从子菜单退回主菜单 (或者退回上一级菜单)。

法 2 C>TCINST /B

此法基本上同法 1, 只是屏幕颜色不是彩色 (缺省模式), 而是单色。

法 3 C>TCINST C:\TC\TC.EXE 或 C>TCINST C:\TC\TC

此法允许 TC.EXE 和 TCINST.EXE 不在一个目录内, 即允许使用目录路径名。

法 4 C>TCINST /B C:\TC\TC

本法是法 2 和法 3 的结合。

为慎重起见,你也可以将 TC.EXE 拷贝一个备份,例如 TCTC.TXE,结果你可以大胆地用 C>TCINST□TCTC

进行试操作。应当指出 TCINST 操作的对象是 TC.EXE,如果不是,则在搜索后显示

Wrong version of TCTC.EXE

另外,你最好在热启动机器后进行,也即内存应尽可能大。切忌在汉字操作系统下进行。

16.5 TCINST.EXE 的菜单结构

有些与 TC.EXE 类同(用相同编号),结构顺序按排出顺序先后判别,—9 开头的编号为 TCINST.EXE 独有。

- 4 Compile
- 4—5 Primary file:
 - =Primary C File=
- 5 Project
- 5—1 Project name
 - =Project file=
- 5—2 Break make on Errors
- 5—2—1 Warnings
- 5—2—2 Errors
- 5—2—3 Fatal errors
- 5—2—4 Link
- 5—3 Auto dependencies Off
- 5—4 Clear project
- 6 Options
- 6—1 Compiler
- 6—1—1 Model small
- 6—1—1—1 Tiny
- 6—1—1—2 Small
- 6—1—1—3 Medium
- 6—1—1—4 Compact
- 6—1—1—5 Large
- 6—1—1—6 Huge
- 6—1—2 Defines
 - =Macros=
- 6—1—3 Code generation
- 6—1—3—1 Calling convention C [Pascal]
- 6—1—3—2 Instruction set 8088/8086 [80186/80286]
- 6—1—3—3 Floating point Emulation [8087/80287] [None]
- 6—1—3—4 Default char type Signed [unsigned]
- 6—1—3—5 Alignment Byte [word]
- 6—1—3—6 Generate underbars On

—6—1—3—7	Merge duplicate strings	Off	
—6—1—3—8	Standard stack frame	On	
—6—1—3—9	Test stack overflow	Off	
—6—1—3—10	Line numbers	Off	
—6—1—3—11	OBJ debug information	On	
—6—1—4	Optimization		
—6—1—4—1	Optimize for	Size [Speed]	
—6—1—4—2	Use register variables	On	
—6—1—4—3	Register optimization	Off	
—6—1—4—4	Jump optimization	Off	
—6—1—5	Source		
—6—1—5—1	Identifier length	32	
	=Specify new Identifier length size: [1..32] 32=		
—6—1—5—2	Nested comments	Off	
—6—1—5—3	ANSI keywords	Off	
—6—1—6	Errors		
—6—1—6—1	Errors ; stop after	25	
	=Specify new Errors size: [0..100] 25=		
—6—1—6—2	Warnings ; stop after	100	
	=Specify new Warning size: [0..100] 100=		
—6—1—6—3	Display warnings	On	
—6—1—6—4	Portability warnings		
—6—1—6—4—1	A;Non-portable pointer conversion		On
—6—1—6—4—2	B;Non-portable pointer assignment		On
—6—1—6—4—3	C;Non-portable pointer comparison		On
—6—1—6—4—4	D;Constant out of range in comparison		On
—6—1—6—4—5	E;Constant is long		Off
—6—1—6—4—6	F;Conversion may lose significant digits		Off
—6—1—6—4—7	G;Mixing pointers to signed and unsigned char		Off
—6—1—6—5	ANSI violations		
—6—1—6—5—1	A;'ident' not part of structure		On
—6—1—6—5—2	B;Zero length structure		On
—6—1—6—5—3	C;Void functions may not return a value		On
—6—1—6—5—4	D;Both return and return of a value used		On
—6—1—6—5—5	E;Suspicious pointer conversion		On
—6—1—6—5—6	F;Undefined structure 'ident'		On
—6—1—6—5—7	G;Redefinition of 'ident' is not identical		On
—6—1—6—5—8	H;Hexadecimal or octal constant too large		On
—6—1—6—6	Common errors		
—6—1—6—6—1	A;Function should return a value		Off
—6—1—6—6—2	B;Unreachable code		On
—6—1—6—6—3	C;Code has no effect		On
—6—1—6—6—4	D;Possible use of 'ident' before definition		On
—6—1—6—6—5	E;'ident' is assigned a value which is never used		On
—6—1—6—6—6	F;Parameter 'ident' is never used		On
—6—1—6—6—7	G;Possibly incorrect assignment		On

```

-6-1-6-7    Less common errors
-6-1-6-7-1    A, Superfluous & with function or array    Off
-6-1-6-7-2    B, 'ident' declared but never used    Off
-6-1-6-7-3    C, Ambiguous operators need parentheses    Off
-6-1-6-7-4    D, Structure passed by value    Off
-6-1-6-7-5    E, No declaration for function 'ident'    Off
-6-1-6-7-6    F, Call to function with no prototype    Off
-6-1-7    Names
-6-1-7-1    Code names
-6-1-7-1-1    Segment name *
                =CODE Segment Name=
-6-1-7-1-2    Group name * *
                =CODE Group Name=
-6-1-7-1-3    Class name *
                =CODE Class Name=
-6-1-7-2    Data names
-6-1-7-2-1    Segment name *
                =DATA Segment Name=
-6-1-7-2-2    Group name * *
                =DATA Group Name=
-6-1-7-2-3    Class name *
                =DATA Class Name=
-6-1-7-3    BSS names
-6-1-7-3-1    Segment name *
                =BSS Segment Name=
-6-1-7-3-2    Group name * *
                =BSS Group Name=
-6-1-7-3-3    Class name *
                =BSS Class Name=
-6-2    Linker
-6-2-1    Map file    Off
-6-2-1-1    Off
-6-2-1-2    Segments
-6-2-1-3    Publics
-6-2-1-4    Detailed
-6-2-2    Initialize segments    Off
-6-2-3    Default libraries    Off
-6-2-4    Graphics library    On
-6-2-5    Warn duplicate symbols    Off
-6-2-6    Stack warning    On
-6-2-7    Case-sensitive link    On
-6-3    Environment
-6-3-1    Message Tracking    Current File [Off] [All Files]
-6-3-2    Keep messages    Off
-6-3-3    Config auto save    Off
-6-3-4    Edit auto save    Off

```

```

-6-3-5    Backup files      On
-6-3-7    Zoomed windows    Off
-9-1      Full graphics save On
-6-3-8    Screen size
-6-3-8-1   25 line display
-6-3-8-2   43/50 line display
-9-2      Options for editor
-9-2-1     Insert mode      On
-9-2-2     Autoindent mode  On
-9-2-3     Use tabs         On
-9-2-4     Optimal fill     On
-9-2-5     Backspace unindents On
-6-3-6     Tab size         8
           =Specify new tab size: [2..16] 8=
-9-2-6     Editor buffer size 65534
           =Specify new editor buffer size: [20000..65534] 65534=
-9-2-7     Make use of EMS memory On
-6-4      Directories
-6-4-1     Include directories : C:\TC\INCLUDE
           =Include Directories=
-6-4-2     Library directories : C:\TC\LIB
           =Library Directories=
-6-4-3     Output directory :
           =Output Directory=
-6-4-4     Turbo C directory :
           =Turbo C Derectory=
-6-4-5     Pick file name :
           =Pick File Name[TCPICK.TCP]=
-9-3      Args
           =Args=
-7        Debug
-7-5      Display swapping Smart
-7-5-1     None
-7-5-2     Smart
-7-5-3     Always
-7-6      Source debugging On
-7-6-1     On
-7-6-2     Standalone
-7-6-3     None
-9-4      Ditor commands
-9-5      Mode for display
-9-5-1     Default
           =Yes the screen was "snowy"=
           =No, always turn off snow detection=
           =Maybe, always check the hardware=
-9-5-2     Color

```

- 9—5—3 Black and white
- 9—5—4 LCD or composite
- 9—5—5 Monochrome
- 9—6 Set colors
 - 9—6—1 Customize colors
 - 9—6—1—1 A: Main menu
 - 9—6—1—2 B: Pulldown menu
 - 9—6—1—3 C: Pop-up menu
 - 9—6—1—4 D: Edit
 - 9—6—1—5 E: Message/Watch
 - 9—6—1—6 F: Compiler status
 - 9—6—1—7 G: Input box
 - 9—6—1—8 H: Error box
 - 9—6—1—9 I: Verify box
 - 9—6—1—10 J: Directory box
 - 9—6—1—11 K: Help
 - 9—6—1—12 L: Status line
 - 9—6—2 Default color set
 - 9—6—3 Turquoise color set
 - 9—6—4 Version 1. X color set
- 9—7 Resize windows
 - = ↑ ↓ resize windows 回车 accept edit ESC—exit=
- 9—8 Quit/save
 - = Save changes to TC. EXE(Y/N)=

16.6 TC.EXE 与 TCINST.EXE 菜单项详细说明

—1 File

主菜单中的文件菜单项,将文件从磁盘读入编辑器,或将编辑器内文件写入磁盘,以及与此相关的一些操作,包括暂时或永久退出集成环境等。

1—1 Load F3

将磁盘文件读入编辑器。选该项后屏幕显示要求你输入文件名的【小窗】(box)

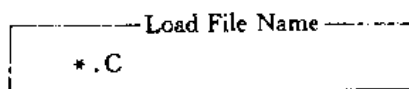


图 16—3

这时你可以输入一个具体的文件名(可带任何扩展名,不一定非是.C),例如 MY.C(或 MY,不输扩展名表示用缺省扩展名.C);也可以使用【DOS 通配符】(指符号 * 或?)。例如,输入 *.* ,则 TC 在一个窗内列出当前目录内的全部文件供你选择,按回车键后【亮条】(即选项后显示有与背景不同的长方形)所在项即被选中。特别,可用 *.C 将全部 C 源文件列表后选择。此外,假定有一个源文件未带任何扩展名,如 PROG,则该文件不能被装入,因为 TC 规定不输扩展名就是带缺省扩展名.C,此时可用 PROG.* 的方法输入,列表后再行选择。

如果在读入新文件时编辑器内已有文件,并且该文件经重新编辑后还未写盘,那么屏幕显示提示

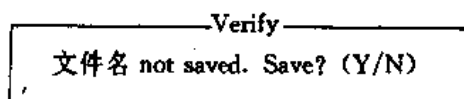


图 16-4

如键入 Y, 则编辑器内的文件先存盘, 然后读入新文件到编辑器内。如果读入文件有具体内容, 则内容便显示在编辑窗内。

如果指定的文件不存在, 则编辑器便自动创建它, 此时编辑器登录其名, 并且编辑窗内无任何源文件内容。

如果给出的文件路径错了, 则显示

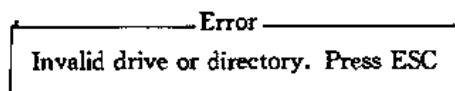


图 16-5

按 ESC 键返回。

Load 后跟的 F3 是热键, 起同样作用(下同, 不再细述)。

—1—2 Pick Alt—F3

在了解本菜单之前应先了解【检选表】。如果在进入集成环境后你编辑了一个源文件(比如叫 MYS1FSST.C), 然后按 F2 键存入磁盘; 接着编辑另一个文件 SSRTOMY2.C, 编好后又存入磁盘, 如此等等。现在你想重新将最先编辑的那个文件(MYS1FSST.C)调出来进行一些修改, 当然可以选

File/Load F3

菜单并输入相应的文件名后, 选中的文件便进入编辑器。但是, 这样做你一是要输入文件名, 当文件名很长或不好记忆时便不方便了(为查询文件名, 这时可多操作一次, 即在输文件名时输入通配符, 例如, *.C, 然后对列出的所有文件名用光条进行选择); 另一方面, 在将该文件装入编辑器后, 你不能在该【文件存盘前的状态】基础上接着进行编辑(其详细说明见下面例子)。因此再好选取集成环境里的菜单

File/Pick Alt—F3

随后出现的小窗的内容是

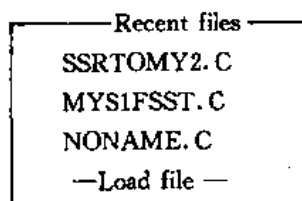


图 16-6

用↑或↓光标键移动亮条到所要选的文件名上, 回车后该文件便被装入编辑器, 即编辑窗内显示该文件供你编辑。早先将此文件存入磁盘前的各种数据状态, 包括编辑光标的原有位置、原先选择的块及标记符, 以及存盘前查找或替换的字符串和替换值等均按存盘前原样恢复, 好像你完全在文件存盘前的状态下接着进行编辑一样。我们把小窗内列出的表简称为检选(Pick)表, 而把记录检选表的文件称为【检选(Pick)文件】。应当注意, 检选文件记录的内容远比小窗内显示的内容多得多, 许多内容有可能没有在小窗内出现。

检选表一直装在编辑器内的(写入磁盘便形成检选文件), 因而只要你没用集成环境下的 File/Quit 命令退出集成环境, 而是用像 File/OS Shell 命令退出集成环境, 尔后用 EXIT 命令重入集成环境时, 原有的检选表还是保留着的。在检选表里, 第一个文件名一定是你最近装

入磁盘的。刚打开检选表时亮条在第二个文件上。它最多可装入 9 个用户文件。最后一个文件名或者是 NONAME.C (用户文件不足 9 个且 NONAME.C 未被编辑时), 或者是最后一个用户文件名。选 —Load file— 项相当于选了 File/Load F3 项。总而言之, 检选表里记录了 you 最近装入编辑器至多 9 个文件的情况。特别要指出的, 所有记录的文件名包括路径。

检选文件产生的方法有三种:

法 1 利用 TCINST.EXE 给 TC.EXE 生成缺省检选文件 (缺省扩展名为 .PCK)

具体方法是, 在 TCINST.EXE 的 O/Directories 菜单里选

Pick file name

输入相应文件名 (否则用缺省名 TCPICK.PCK) 回车后, 在退出 TCINST 时将信息存入 TC.EXE (出现提示

Save changes to TC.EXE? (Y/N)

时键入 Y) 便可。以后进入集成环境时, 查菜单

Options/Directories/Pick file name; TCPICK.PCK

Options/Directories/Current pick file; TCPICK.PCK

可见用了缺省的检选文件。注意: 后一菜单显浅色时, 它表明当前使用的检选文件名。不能通过它来重选检选文件。

法 2 利用集成环境里的 Options/Directories/Pick file name

选菜单后键入相应文件名 (缺省扩展名为 .TCP) 即可。如果磁盘上早先有此文件名, 则此检选文件被装入编辑器 (包括该检选文件对应的检选表中的第一个源文件的内容也自动装入编辑器), 否则 TC 将自动创建它。

法 3 利用配置文件 (*.TC) 装入

配置文件包括检选文件, 因此当你在进入集成环境时当前目录上有 TCCONFIG.TC 文件, 而该文件又记录有检选文件 (例如, TCPICK.TCP), 当前盘上也有 TCPICK.TCP 文件, 那么进入集成环境时编辑器内马上装入 TCPICK.TCP 的第一个文件 (例如, MYFILE.C) 及相应状态。如果 MYFILE.C 不存在, 编辑窗口内除编辑状态行外, 什么也没有, 当前编辑的文件名还是 MYFILE.C。当当前目录内无检选文件 TCPICK.TCP 时, 则编辑器装入 NONAME.C。

当指定了检选文件名, 则每次退出集成环境时对应的检选文件便被更新。注意, 如果有菜单

Options/Directories/Pick file name, Options/Directories/Current pick file; TCPICK.PCK

即前一个菜单值为空, 那么 TC 在退出集成环境时将不保存或更新 TCPICK.PCK 检选文件, 尽管这时后一菜单显示了 TCPICK.PCK。

集成环境如果装入了指定的检选文件, 则它可记住早先编辑过的文件及其存盘前的状态。

注意: 如果磁盘上没有缺省检选文件存在, 那么当你用 C>TC 进入集成环境时显示 (这也是本书选用的 Turbo C 版本)

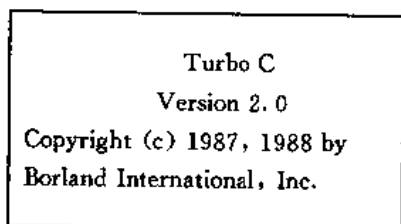


图 16-7

否则无此显示。显示版本的热键是 Shift-F10。

—1—3 New

清编辑窗,缺省源文件名 NONAME.C 被建立。编辑完后如要存盘,TC 将询问是否要 将它换名:

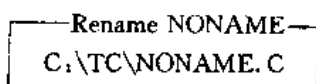


图 16-8

可以换名后存盘。

—1—4 Save F2

不论在何时,按 F2 键后编辑窗内的源文件将无条件存盘(如磁盘已有同名文件,则 被覆 盖,即被改写)。

—1—5 Write to

选该菜单后显示

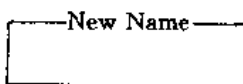


图 16-9

可以输入一个跟编辑窗内文件名不同的文件名,TC 将按新给的名将编辑文件写入磁盘, 而编辑窗内的文件除换成新名外其余内容不受任何影响。实质上这就是将编辑文件拷贝。如 果指定的文件和磁盘上某文件同名,屏幕显示

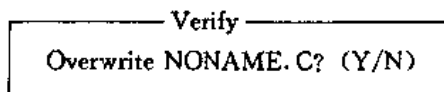


图 16-10

键入 Y 则将已有同名文件改写(先删除,后写入)。

—1—6 Directory

选该项后显示

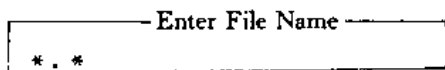


图 16-11

也即让用户通过显示的目录列表来选择要装入编辑器的源文件名。它跟 Load F3 菜单初 看起来作用是一样的,但有两点区别:

1. 假定指定一个磁盘上确已存在的文件(如 MY.C),Load F3 则将该文件立即装入 编辑器;而本命令则不同,它首先显示所有 MY.* 即全部扩展名不同的(特别有扩展名为 BAK 的后备文件)文件列表,供你选用。你必须再次对其选择对应文件后才能使文件进入 编辑器。

2. 假定指定文件不存在,则 Load F3 则立即在编辑器内创建该文件,换句话说,该 文件 已进入编辑器;而本命令则不同,它将显示

Can't find file

拒绝接受! 按 ESC 键退回重选。

—1—7 Change dir

改变【当前目录】名。所谓当前目录即是当前正在使用的目录,也是缺省目录。例如一个文件未给出路径,则认为它在当前目录里。一般情况下,启动 TC.EXE 后当前目录就是 TC.EXE 所在目录。之后你可用本命令对其改变。选择本项后显示当前目录名

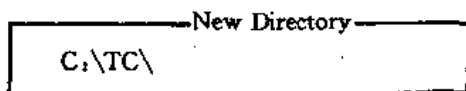


图 16-12

显见,利用本命令你可以查询当前目录。注意:不要将它与 Options/Directories 中的目录混为一谈。

如要改变目录,输入新目录后按回车键确认;否则按 ESC 键取消修改,原当前目录不变。

—1—8 OS Shell

选此项时 TC 将暂时退出集成环境而转到 DOS(DOS 提示符出现)。DOS 的目录为当前目录。这时用户可以执行各种 DOS 命令。事实上此时 TC.EXE 是常驻在内存,因此你不能用诸如

C>TC

那样的命令重入集成环境,那样会出现内存不够。不管在何子目录下,而应键入 EXIT,回车后立即重返集成环境。在调试程序时,视频模式和光标类型被放入调试状态中。

在使用双屏幕时(很少用),暂时退出集成环境后输入的 DOS 命令会出现 TC 所在的屏幕上,而不会出现在用户程序输出的屏幕上。

一般情况下,用本命令返回 DOS 时, DOS 原先的屏幕将恢复再现,除非调试中已改变过屏幕。

—1—9 Quit Alt-X

退出集成环境,返回到 DOS 提示符下。TC.EXE 也不常驻内存。在退出前如编辑器内的文件尚未存盘,将出现

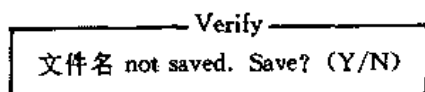


图 16-13

如键 Y,则将当前文件存盘。

—2 Edit Alt-E 或 F10

进入编辑状态的主菜单。当光条在该选项上按回车键,或者按 ESC 键,或按字母键 E 后回车,或按 Alt-E 均可进入编辑。F10 键是进入 / 退出编辑键。

该命令使程序员可以利用编辑窗编辑源程序。编辑的文件可以是 C 源文件,也可以是其它 ASCII 码型文件(和 WORDSTAR 编辑的文件相当)。当信息窗口内有信息显示时,可以用 Alt-F7 或 Alt-F8 将信息窗内的前一错误或后一错误定位在编辑窗口内的【违反行】上,即可能有问题的行上。

当在编辑状态,即编辑窗口被激活时,你可用 Ctrl-F1 键操作得到帮助,它是英文提示,指出编辑光标所在项的对应帮助内容。

编辑窗口的顶上一行,即

Line 1 Col 1 Insetrt Indent Tab Fill Unindent * C:\NONAME.C

称【编辑状态行】(参见 TCINST.EXE 的—9—2 Options for editor)。

在编辑窗口中使用插入模式 (Insert 将显示) 输入代码时,可用回车键来结束一行 (TC 编辑器不会自动换行)。最大行宽为 248 个字符,编辑窗口宽 77 列。如超过 77 列,窗口随着字符的键入而滚动。TC 屏幕上端的状态行的 Line 和 Col 后的数字指出光标所在文件中的行、列位置。在编辑窗口中输完代码后按 F10 键转到主菜单 (参见 —9—x—x 部分内容)。

—3 Run

运行程序主菜单。

被执行程序如带有命令行参数时可用

Options/Argument (—6—5)

选项设置命令行参数,供程序执行时传递。

—3—1 Run Ctrl—F9

TC 将启动集成环境中的【规划组装器】(PROJECT—MAKE,或称【组装器】),对上次编译后发生变动的源代码 (包括源文件、源目标文件和库文件等) 重新进行编译、连接,然后执行程序输出结果。程序执行完毕后立即返回 TC 主屏幕。

若不想调试程序,编译连接前应置

Debug/Source debugging None [Standaione] (—7—6)

否则产生的可执行代码中将包含程序调试信息。

程序执行中的几个问题:

1. 若上次编译后未修改过源代码,则程序执行到事先设置的第一个断点时便暂停;否则便执行到底。

2. 若上次编译后修改过源代码,且已使用了 Run/Step over F8 或 Run/Trace into F7 单步执行程序,现在又来执行程序,屏幕显示

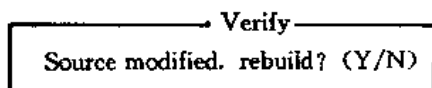


图 16—14

询问是否要从头开始编译、连接和执行程序。按 N 键则不从头开始,而是执行到下一断点,无断点时执行到底。

若没有用过单步执行,则在重新编译连接后重新开始执行。

当无调试信息时显示

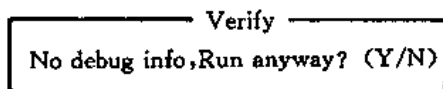


图 16--15

键 Y 继续运行程序,否则终止。

程序执行中暂停或结束时可按 Alt—F5 显示程序输出结果。程序运行结束后选

Compile/Get info

可得到由 main() 函数返回的值。

—3—2 Program reset Ctrl—F2

中止当前程序调试,释放分配给程序的空间,关闭打开的文件。激活编辑窗。编辑窗口内的源文件依然存在,包括已设的断点。

如果你对一个程序执行调试（使用 F7 或 F8 键操作）中间突然不想进行下去了，那么就应用本命令！否则，即使你又重新调入了一个新源程序，你想对此新源程序进行调试，那么你将发现，你实际调试的还是原来你中间放弃调试的那个源程序，而不是刚刚调入编辑窗的源程序！

仅在调试时有效！

—3—3 Goto cursor F4

使程序从已执行的亮条处一直执行到编辑光标所在行后暂停，如果编辑光标以前的可执行行中没有断点的话。否则程序每遇到一个断点便暂停，你可用本命令使它往下执行。这就是说，此时编辑光标本身也被当成一个特殊的“断点”（可以称为【暂时暂停点】）。不过，当编辑光标所在行并不含可执行代码时（如空行，#include 语句，注释行等），显示

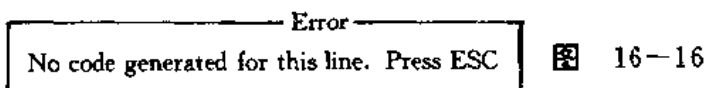


图 16-16

这跟用

Break/watch/Toggle breakpoint Ctrl-F8 (—8—5)

设置了无效断点的显示

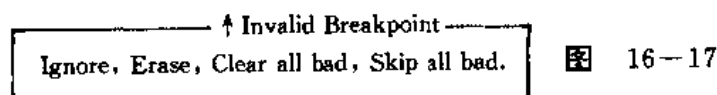


图 16-17

有点不一样（按 I 不管；按 E 删除亮条处断点；按 C 清除所有无效的断点；按 S 则跳过所有无效的断点）。

—3—4 Trace into F7

每按下一次 F7 键便运行当前函数里的下一条语句，若被执行语句里含有调用别的函数的语句，则亮条停在该函数的定义语句上。此后再按 F7 便执行该函数内的语句。该函数内全部语句执行完后便返回执行调用此函数的语句的下一个语句。如果执行语句里没有调用别的函数语句，则执行完后顺序执行下一句。因此，用此调试程序时几乎遍访整个可执行语句。

注意：函数可访问的条件是，在编译源程序前应有

Options/Compiler/Code generation/OBJ debug information On (—6—1—3—11)
Debug/Source debugging On (—7—6)

且调试器能找到其所在的源文件。

—3—5 Step over F8

单步执行程序。它与 Trace into F7 类似，但有一点重要区别：当执行一条语句中有调用另一函数时，它不会进入该函数，而是执行下一语句。为说明两者的区别，可参见下例：

```
C>TYPE MENU1.C
int findit(void)          /* 行号 1 */
{                          /* 行号 2 */
    return 2;             /* 行号 3 */
}                          /* 行号 4 */
main()                   /* 行号 5 */
{                          /* 行号 6 */
    int k;                /* 行号 7 */
```

```

k=findit(); /* 行号 8 */
printf("k=%d findit()=%d\n",k,findit()); /* 行号 9 */
} /* 行号 10 */
/* 用 Trace into 执行语句顺序: 5 8 1 3 4 9 1 3 4 10 */
/* 用 Step over 执行语句顺序: 5 8 9 10 */

```

—3—6 User screen Alt—F5

用于双屏幕管理：一个屏幕为 TC 占用，一个用于程序执行结果输出。为方便叙述，可把前者称为【调试屏幕】（或【TC 主屏幕】），后者称为【用户屏幕】。对用单屏幕的用户，一个屏幕可起两种屏幕的作用：选它一次便使屏幕从 TC 主屏幕进入用户屏幕，或者说可以看到 DOS 状态下的屏幕，从而可以看到程序输出结果。不过这时不能用任何 DOS 命令！按任一非控制键又立即返回 TC 主屏幕。

在调试程序中间常常用此命令看看中间输出结果。当程序输出超出 25 行时只能显示最后 24 行的值。

注意：对有些汉字系统虽然允许直接使用西文 Turbo C，例如联想汉字环境下，由于键功能的冲突而不能使用此选项。

使用双屏幕的用户应当注意到，使用的两个显示器的类型可能被要求为不同的。例如，一个为 EGA，而另一个为单色显示器。因而机器上应装有相应的适配器。

—4 Compile

编译主菜单。

—4—1 Compile to OBJ Alt—F9

当项

Primary C file: (—4—5)

后没有显示主 C 文件名时，TC 便将编辑窗内的源文件（如 MENU2.C）编译成目标文件（MENU2.OBJ）；或者仅管当前编辑窗内有一源文件 SUB.C；

```

C>TYPE Sub.C /* 主 C 文件的相依文件，已在编辑窗内 */
PRINT(int i)
{printf("%d\n",i);
}

```

而在磁盘上有其相关的主文件 MENU2.C；

```

C>TYPE MENU2.C /* 主 C 文件名，在磁盘上 */
#include "stdio.h"
#include "sub.c"
main()
{
PRINT(1);
}

```

则 Compile to OBJ 后显示的是主 C 文件名（MENU2.C），最后编译成一个目标文件 MENU2.OBJ。注意：生成的目标文件只有一个，而不是两个（MENU2.OBJ 和 SUB.OBJ）！目标文件名依主 C 文件名而定。

Compile to OBJ 后显示的文件名是不容程序员修改的，一按回车键编译便马上开始。如觉不对，可按 ESC 键终止。

编译时显示编译窗：

Compiling		
Main file: PRIMARY.C		
Compiling: EDITER → MENU2.C		
	Total	File
Lines compiled:	217	217
Warnings:	0	0
Errors:	0	0
Avarilable memory :273K		
Success	Press any key	

图 16-18

编译如无错误,显示窗内内容。否则,转到消息窗口内第一条错误或警告标志上(亮条)。再按回车后,亮条所在的内容显示在编辑窗的最上端。按任一键后该内容消失,编辑光标出现,编辑窗被激活。

—4—2 Make EXE file F9

TC 将调用规划组装机来生成一个执行文件(.EXE)。Make EXE file 后的文件名也是 TC 根据环境自动生成的,程序员不能直接对其修改。其生成的规则优先顺序是:

首先检查有否规划文件(.PRJ),如有,则生成规划执行文件;否则检查有否主 C 文件。如有,则生成主 C 文件的执行文件;否则生成编辑窗内源文件的执行文件。

编译时将检查依赖文件是否存在,显示

Compiling	
Checking dependencies...	
Ctrl-Break to quit	

图 16-19

编译完后即进行连接(LINK)。

编译或连接时程序员可以看到被调用的目标文件(*.OBJ)和库文件(*.LIB),它们应在指定的目录里。

本命令只对那些已过时的文件(例如最近被修改过的文件)进行操作。例如,有一个规划文件(MCALC.PRJ),首次我们将它用本命令进行了编译连接后组装成执行文件 MCALC.EXE,现在对其规划内的多个文件之一,如 MCLAL.C 进行了一点修改(例如在行尾增加一个空格符),然后再执行本命令时便可发现,编译只对 MCALC.C 进行,而对其它未改动文件一律不进行,因为这些原有文件还未过时(没有被修改,一修改文件存盘时间便会变动)。由此可见,这将给程序员调试程序节约时间,故常用。

注意:这一功能是假定你一直在 TC 集成环境下(包括暂时退出集成环境而后又用 EXIT 返回),而当用 QUIT 退出集成环境后重入时一般就不行了(带配置文件进入可行,但必须恢复其相应环境)。

当执行了一遍本命令后又紧接着执行该命令,则出现

is up to date

表示生成文件已是最新的,不用重来。

—4—3 Link EXE file

把目标文件与库文件连接成执行文件,这些文件既可以是缺省的(在规定的目录里),也可以是定义在当前规划文件里(.PRJ)。只要一按回车键连接便开始,而不管这些文件是否曾被修改过。换言之,对文件不作过时检查。

可以说,Make EXE file 相当于 Compile to OBJ 和 Link EXE file 这两个过程。当跟其它语言接口时,有时需要这种单独的操作过程。

—4—4 Build all

不论是否过时,重建规划里的全部文件。它类似于 Make EXE file,只是它是无条件执行的。执行此命令时,先将规划文件(*.PRJ)的目标文件(*.OBJ)的日期、时间置为0,然后再编译连接生成新的执行文件。在执行本命令时如用 Ctrl-Brak 键中断了命令执行,如要继续执行,可用 Mack EXE file 命令。

—4—5 Primaty C file;

显示 Primary file 时按回车后出现小窗

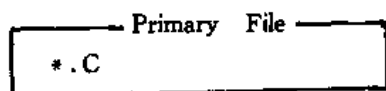


图 16-20

要求输入主 C 文件名(参见《程序结构和主函数》一章)。例如下面两个文件是相关的。

```
C>TYPE MENU3.C
#include "e,sub.c"
main()
{printf("SUB num=%d\n",num);}
```

```
E>TYPE SUB.C
#include "stdio.h"
int num=100;
```

文件 MENU3.C 便是主 C 文件。用 TC 的 CPP 工具(执行 CPP.EXE)可以发现,在预处理时 TC 自动将 E;SUB.C 文件包含进来。事实上 TC 编辑器可编辑 C 源文件的内存缓冲区最多只能是 64K,超过时便不能编辑。但利用包含文件,并指定主 C 文件,就可突破此限制。

当编译过程中发现错误,TC 会自动将含错源文件装入编辑窗内,供程序员修改。但有一点应注意,头文件(*.H)只有在你用

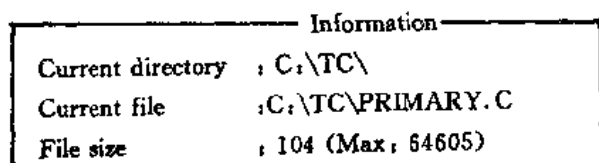
Options/Environment/Message Tracking All files (—6—3—1)

时才会发生错误时将它们中的相关文件装入编辑器。

即使编辑器内无任何包含文件,但只要你指定了主 C 文件(如果没有指定规划文件时),那么按 Alt-F9 键主 C 文件和与它相关的文件将被编译。这说明编译器先编译主文件而不管编辑器中的源文件。如果指定了规划文件(*.PRJ),则不必再指定主 C 文件。

—4—6 Get info

一按回车键,出现一小窗:



EMS usage	: OK
Lines compiled	: 0 No program running
Total warnings	: 0 Program exit code 16
Total errors	: 0 Available memory :112k
Press any key	

图 16-21

它给出了当前目录名 (Current directory)、当前源文件或目标文件名 (Current file)、文件大小 (File size)、EMS (扩页内存) 使用情况 (EMS usage)、编译的行数 (Lines compiled)、发生的警告数 (Total warnings)、发生的错误数 (Total errors)、程序执行情况 (没有程序被执行 No program running, 或程序执行完毕 Program terminated, 或无程序装入 No program loaded)、程序退出码 (Program exit code) 以及可用空间 (Available memory) 等。按任一键返回。

可用空间是指编译后尚存多少内存单元可供利用。

—5 Project

规划主菜单, 通过 Project 菜单上的命令, 就可决定是否在编译时将多个 C 源文件 (*.C) 及目标文件 (*.OBJ) 合起来生成最后可执行文件。

—5—1 Project name

输入缺省规划文件名。例如, TC 系统磁盘上就有一个规划文件名 MCALC.PRJ;

C>TYPE MCALC.PRJ

```
mcalc (mcalc.h)        /* 或者说 mcalc.c 依赖于 mcalc.h */
mcparser (mcalc.h)
mcdisplay (mcalc.h)
mcinput (mcalc.h)
mcommand (mcalc.h)
mcutil (mcalc.h)
```

该文件的特点是扩展名为 .PRJ, 内容为 C 源程序名及非标准头文件 (在源文件名后的括号内, 此头文件内容是用户自定的)。这些文件是将来一次连续编译、连接的。我们把这类涉及多个文件的文件 (象 MCALC.PRJ) 称为【规划文件】(Project file)。典型的规划文件名具有扩展名 .PRJ。另外, 缺省时 *.EXE (可执行文件) 或 MAP (由连接器产生的映射) 文件名也是根据 *.PRJ 取名的。.PRJ 文件的形式同 TC 的 *.C 源文件 有点相似, 但有其自身的特点:

1. 每一行都有一个 C 源程序名, 行尾无任何标点。
2. 各文件的排列顺序是无所谓的, 它只影响编译的顺序。
3. 各文件也可以包括目录路径, 也即是说, 所有的 C 源文件可以在不同的目录中。
4. 说明将被连接到 *.PRJ 文件里的库文件 (*.LIB) 或外部目标文件 (*.OBJ) 时应应用圆括号括起, 构成【依赖表】。依赖表中的文件是 C 源文件依赖的文件。当有多个文件时, 可用空格、逗号或分号分隔。这种依赖关系称为【显式依赖】; 还有一种依赖是当

Auto dependencies On

时, 所包含的 C 源文件都将被检查 (而显式依赖的括号内的文件不被检查), 以确定是否要重新编译, 这就是所谓的【隐式依赖】。

有时候程序需要来自像汇编语言或另外一种编译程序生成的目标文件或库文件, 它们执

行操作的标准库没有提供。在这种情况下,可用两种方法将它们放到 *.PRJ 文件中:

1. 不放入依赖表中,如

```
MYMAIN (MYFUNCS. H)
MYFUNCS (MYFUNCS. H)
SPECIAL. OBJ      /* 不能有依赖表 */
OTHER. LIB        /* 不能有依赖表 */
```

2. 放入依赖表中,如

```
MYMAIN (MYFUNCS. H,SPECIAL. OBJ)
MYFUNCS (MYFUNCS. H,OTHER. LIB)
SPECIAL. OBJ
OTHER. LIB
```

放入依赖表中意味着当某个 .OBJ 或 .LIB 更新了,相应的 C 源文件也将重新被编译。

在某些情况下,可能不得不抑制即不理睬某些标准文件或库(对初学者这样做时要非常谨慎),具体方法为:

1. 抑制启动文件

必须在规划文件一开始写上

```
C0XXXXXX. OBJ
```

这里 XXXXXX 表示最多为 6 个 DOS 文件名字符,例如 C0mine. obj 就是新的启动文件名。当然,该文件必须实际在磁盘上存在。

启动文件同存储模式相关。例如, C0S. OBJ 是小模式启动文件。启动文件主要处理的内容有,存储模式、C 约定、浮点运算、DGROUOP 段址、程序 PSP 段址、环境地址、—8087 浮点仿真、远堆顶端、DOS 版本号、堆栈、错误处理、初始化数据区、程序主参数、BIOS 时间、窗口、关闭流和文件、恢复启动时的中断矢量及 TSR 常驻内存例程等。它是一个比较复杂的模块。初接触 Turbo C 者可不考虑对它改动。可参考 C0. ASM 文件。

2. 抑制标准运行库

要抑制标准库,所要做的只是将库名放到规划文件中(位置任意)。注意:库名必须以 C 开头,后跟存储模式字母(如 S 表示小存储模式),其余 6 个字母可任意,但扩展名 .LIB 必须有。如 CSMYMATH. LIB。

当标准库被抑制后,MAKE 时就不连接在 O/C/C/Floating point 开关指定的数学库(EMU. LIB 或 MATHx. LIB)。如果抑制了标准库又不想将这些库连接进去,就得将它们显式地包括在规划文件中。

Turbo C 提供的磁盘上有一个文件 BGIDEMO. PRJ:

```
C>TYPE BGIDEMO. PRJ
bgidemo. c□graphics. lib
```

如果在当前编辑窗内有文件 MYFILE. C,而集成环境下 Project/Project name 内有文件名,比如 BGIDEMO. PRJ,则编译时编译 BGIDEMO. PRJ,生成 BGIDEMO. EXE 可执行文件,而不管当前文件 MYFILE. C。在用 Run/Run 执行时也一样。

注意:规划文件名和主 C 文件名是有区别的。由规划文件生成的目标文件和执行文件除了扩展名不同外,其余的文件名内容相同。

规划文件里如有 *.OBJ 文件,则其对应的 C 源文件应存在,否则会出现

Can't find source file, enter new name

的提示。如果你对这 OBJ 文件用 Compile to OBJ 命令单独编译后又进行对源文件的修改,然后再对规划文件编译,则编译将不理睬刚才的修改。

—5—2 Break make on Errors

此菜单让程序员预置中止组装器操作的缺省条件。如发生以下 4 种情况之一,组装器中止工作:

—5—2—1 Warnings

当编译遇到一个警告发生时。

—5—2—2 Errors

编译遇到错误发生时。

—5—2—3 Fatal errors

有致命性错误发生时,组装器中止后它将产生一个涉及规划文件中所有文件的错误和警告表。如果没有错误发生,则开始连接。

—5—2—4 Link

组装器工作到进行连接操作之前结束。

—5—3 Auto dependencies Off

自动依赖关系检查开关,可选 On 或 Off(缺省值)。所谓【自动依赖关系检查】,当其设置成 On 时,是指:

1. 自动搜索并打开规划文件中的 C 源文件及对应的目标文件(指早先已被 TC.EXE 或 TCC.EXE 编译而生成的文件 *.OBJ);

2. 将每一个 C 源文件与其对应的目标文件生成时的日期/时间信息进行比较,如不同则重新编译 C 源文件。对日期/时间相同的 C 源文件则不重编译。

当开关设置成 Off 时则不进行这种检查,每次重编译时将对规划文件中的全部 C 源文件进行重编译。设置这种状态的好处是,当重新启动机器时由于时间设置错误而造成不必要的失误。一般情况下,开机后首次对规划文件编译后可将其设置成 On。

—5—4 Clear Project

用 ↑ 或 ↓ 键将光条移到此菜单上并回车后,原先你选的规划文件名及消息窗口内的内容都自动被删掉。操作后你可再选 Project Name 菜单对此检查验证。没有选本菜单,则已选规划文件名在退出集成环境前将一直作为缺省值被保留。

—5—5 Remove messages

清除掉信息窗口内的错误信息。

—6 Options

控制集成环境的编译器(Compiler)、连接器(Linker)、工作环境(Environment)、组装器工作文件目录(Directories)、输入执行程序用到的参数(Arguments)、保存当前配置文件(Save Options)及装入以前设置的配置文件等。

—6—1 Compiler

编译器菜单:选择编译器缺省参数。

当 TC 在编译时,弹出的小窗内显示编译状态:主 C 文件名、编译行、可能发生的错误和警告行、可用内存。发生的错误将自动装入信息窗口。

—6—1—1 Model Small

选择存储模式(缺省存储模式为小存储模式)。

—6—1—1—1 Tiny

微存储模式。执行程序所有代码、数据和堆栈共占 64K 内存。所有函数和数据使用缺省的 **near** 指针。此种模式下生成的执行程序 (.EXE) 可转变为 .COM 型文件。

—6—1—1—2 Small

小存储模式。有 64K 程序代码,另有 64K 数据与堆栈,所有函数和数据使用缺省的 **near** 指针。

—6—1—1—3 Medium

中存储模式。程序代码可达 1MB,数据和堆栈占 64K。所有函数用缺省的 **far** 指针;所有数据用缺省的 **near** 指针。

—6—1—1—4 Compact

紧凑存储模式。64K 代码;堆栈和静态数据占 64K;对【堆】(heap)可容许多至 1MB 的数据;所有函数用缺省的 **near** 指针,数据用 **far** 指针。

—6—1—1—5 Large

大存储模式。代码可多至 1MB;堆栈和静态数据占 64K;对堆可多至 1MB 数据;所有函数和数据均用缺省的 **far** 指针。

—6—1—1—6 Huge

巨存储模式。允许重复段,每个段 64K,代码可达 1MB,堆栈 64K。所有函数和数据指针均分配为 **far** 型。

—6—1—2 Defines

在一般情况下,TC 用 TCINST.EXE 预先定义的缺省宏作临时宏 (Macros)。所谓临时宏就是指只在集成环境下才有效的宏 (但编译时将影响执行文件 *.EXE),一旦退出集成环境该宏便不起作用。缺省情况下 TC.EXE 并没有定义任何临时宏,它允许用户临时定义预处理符号,在调试程序时调试器将自动地使用这些宏。临时宏一旦定义,直到退出 TC 前一直有效。这一功能对用户只想调试某段程序时是非常有用的。例如,对程序 MENU4.C:

C> TYPE MENU4.C

```
main()
{
    int k;
    #ifdef MYK
    k=1;
    #else
    k=0;
    #endif
    printf("%d\n",k);
}
```

调试时选用本命令回车后,出现要求你输入一个或多个宏名的小窗 (Defined Symbols),如你输入一个临时宏名 MYK,程序输出 1,否则输出 0。

可以通过执行 TCINST.EXE 对原先的缺省值进行解除或修改。例如,在定义宏小窗上键入下列字符:

```
BETA-TEST,ONE=1,COMPILER=TURBOC,AS=(ONE+ONE),SIG="ONE=%d\nCOMPIL-
ER=
%d\n\nAS=%d\n"
```

按回车后 TCINST 便缺认,然后在退出 TCINST 时对主菜单选 Quit/Save(按 S 键),所选结

果便被作为缺省值存入 TC.EXE 执行文件内。现在你在集成环境下编译 C 源程序 MENU5.
C: C>TYPE MENU5.C

```
#define TURBOC -1
main()
{
    printf(SIG,ONE,COMPILER,AS);
}
```

执行 MENU5.EXE 便有结果

```
ONE=1
COMPILER=-1;
AS=2
```

在此要说明的是:

1. 上述缺省宏定义了多个符号常量 (BETA—TEST, ONE, COMPILER, AS, SIG), 多个符号常量之间用分号分隔 (如果你想在宏中包含一个分号, 则应在其前面加上一个反斜杠)。所有定义宏全写在一行上, 整行能容纳最多字符数 137 个;
2. 上述定义的 TURBOC 是符号常量, 因此在 C 源程序中应用宏进行定义;
3. 当定义宏窗口内字符显高亮度时, 你一键入一个字符便会将原窗口内已有字符全消去。为了在原有字符基础上进行修改, 可用 TAB 键; 注意: 在集成环境下, 字符一般不显高亮度, 此时应键 DEL 键或 →← 键进入修改;
4. 通常此时常处于字符插入状态 (相当于 INSERT 键一直起作用), 要消去一个不用的字符, 可以用光标键移到该字符处, 键 DEL 键删掉, 然后键入新字符 (也可用退格键将字符删去);
5. 一行开头输入的空格和行尾的空格都会被去掉, 但中间的空格被保留;
6. 从上述例子你应仔细体会和用头文件定义符号常量的异同。至少, 在头文件中定义符号常量时没有用赋值号 (=)。

—6—1—3 Code generation

设置缺省的目标代码生成方式。

—6—1—3—1 Calling convention C [Pascal]

调用约定。当你连续按回车键则反复出现字符 C 或字符 Pascal。一般用 C, 即所有函数和指向这些函数的指针都具有 Turbo C 规定的特性 (例如, 对标识符大小写是有区别的); 而选 Pascal 时, 则把它们当作 Pascal 类型处理。此时允许从其它语言写的程序中调用 C 函数, 也允许 C 程序调用其它语言所书写的外部子程序。连接时函数名全部转为大写。C 和 Pascal 调用约定不同之处在于清栈方式、函数参数个数 (例如, 类型为 pascal 的函数不允许有个数可变的参数)、外部标识符的字体 (字母大小写)、前缀 (即下划线) 等 (参见《函数》一章)。只有当你确实需要而又搞清怎么用时才选 Pascal。

—6—1—3—2 Instruction set 8088/8086 [80186/80286]

说明编译采用的 CPU 指令集。可用 8088/8086 或 80186/80286 (靠按回车选择)。TC 可产生扩展的 80X86 指令, 也可用它来生成在实际模型上运行的 80X86 程序 (如在 MS-DOS 支持下的 IBM PC AT)。

—6—1—3—3 Floating point Emulation [8087/80287] [None]

浮点运算处理。有三种方式: 仿真、8087/80287 或空

—6—1—3—3—1 Emulation

仿真 8087/80287。TC 自动检查机上有没有装 8087/80287 协处理芯片,装了则使用它,否则便进行仿真 8087/80287,只是运算速度比装了 8087/80287 的速度慢一些。仿真生成的目标码可在无协处理器的机器上运行。

—6—1—3—3—2 8087/80287

告诉 TC 装有 8087/80287 协处理器,直接产生 8087/80287 代码。生成的目标码只能在装有相应协处理器的机器上才能应用。

—6—1—3—3—3 None

不使用浮点数。若选了 None 而在程序中又有浮点运算,就会出现连接错误。这是要注意的。

—6—1—3—3—4 Default char type Signed [Unsigned]

设置缺省的字符类型。若选择了 Signed(缺省值),则编译时将说明为 char 的类型均认为是带符号的,反之,选 Unsigned 被认为是无符号的。

—6—1—3—3—5 Alignment Byte [word]

此开关可选择 Byte(字节)或 Word(字)对齐两种形式,它不包括段边界对齐。【字对齐】(word)时,非字符数据不能奇地址对齐,只能偶地址对齐,即段的开始地址一定出现在偶地址上。此时可提高 8086 和 80286 处理机存取数据的速度;字对齐方式也称【字编址】。而用【字节对齐】时,数据对奇偶地址均可对齐,这取决于下一个可用地址。

说明:在给结构分配存储空间时,那些大于一个字符长度的结构元素分配的空间一般都是从整数边界开始的,这就是以字对齐方式摆放。如果为了将结构数据“紧凑”地存放在内存中,就将数据从第一个字节开始连续存放,而不考虑整数边界。不过这样做对大多数机器在存取数据时就要化时间去分解这些紧凑的结构元素,从而导至程序执行速度放慢。

—6—1—3—3—6 Generate underbars On

设置外观 C 符号开始处加上下划线(—),即用下划线作字符前缀。Off 则没有,缺省是有(On)。为使 TC 标准库能正确工作,缺省选择为 On。

使用 Off 选择时应慎重考虑。例如,你选用 Pascal 调用约定(—6—1—3—1)时,而在用包含语句(#include)时包含了 C 的标头文件(*.h),在那些文件里函数大多数用 cdecl 修饰符进行了说明,这就是说标识符编译后要加下划线(本命令为 On),而 Pascal 约定是不加下划线的,因此这时应将命令置为 Off,并且连接程序应与大小写无关。

—6—1—3—3—7 Merge duplicate strings Off

合并重复字符串,在 On 和 Off 之间选择。如选用 On,在编译时将匹配的字符串合并为一个串,从而可以生成小一点的程序,也即起优化作用。

—6—1—3—3—8 Standard stack frame On

是否使用标准堆栈结构(或称【堆栈帧,stack frame】),在 On 和 Off 之间选择。如选用 On,则对每一个函数产生一个标准的栈结构(标准函数入口及退出码)。使用调试器时这是有用的,它简化了反向跟踪调用过程栈的处理。

编译源文件时,若此开关为 Off,那么任何不使用局部变量的有参数的函数都同压缩入口及返回码一起编译,这使得代码更短更快,阻止 Debug/Call Stack“看见”该函数。所以在编译要调试的源文件前应当将开关置为 On。

注意:函数内定义的局部变量是动态的实体,它们所属的内存位置在函数开始时被分配,而在函数执行结束时又被收回,以便供其它函数分配内存用。

编译器在堆栈上分配位置的方式是在函数开头产生程序代码,使堆栈指针指向局部变量,

而如果“跳过”从堆栈指针所指的定义全部局部变量的堆栈区,那末堆栈的其它区域(包括函数其它信息)便称为堆栈帧。指向局部变量区尾部而指向堆栈帧头部的指针称为帧指针(FP, frame pointer, 8086 机上使用 BP)。这个指针被保存在适当的寄存器内,编译器就可利用位移和帧指针相加访问到任何被使用到的局部变量。

—6—1—3—9 Test stack overflow Off

是否进行堆栈溢出检查,在 On 和 Off 之间选择。注意,置成 On 时可能使程序变大。检查要化费时间和空间开销,但堆栈溢出可能成为跟踪的障碍,不安全。如果程序常出现不可解释的问题,可以把它置成 On,再来编译程序,看看是否有栈溢出。

—6—1—3—10 Line numbers Off

是否在【映射文件】(MAP 文件)里放入(符号调试器需用的)行号信息,在 On 和 Off 之间选择。如果选用 On,会使目标文件和映射文件变大,但不影响程序的执行速度。不过,假定 Debug/Source Debugging 开关置为 On,且连接时本命令也选 On,那么可执行文件也会变大,多出来的信息是调试信息。

由于编译器在进行跳转优化时,或组合源文件中的公共代码行,或记录号(这不利于行号跟踪),所以当调试程序时本命令应为 On, O/C/O/Jump Optimization 应为 Off。

—6—1—3—11 OBJ debug information On

是否产生目标调试信息,在 On 和 Off 之间选择。如选用 On,那么调试信息将放入目标文件(*.OBJ)里,以便供集成环境或单独的 TC 调试器使用。置成 Off 时,源码编译后将不能为 TC 集成调试器理解。为使 TC 集成调试器正常工作,当程序连接前应有

Debug/Source debugging On

—6—1—4 Optimization

优化处理方式。有以下四种选择,它改变 TC 码生成策略。

—6—1—4—1 Optimize for Size [Speed]

若选 Size,则编译生成尽可能小的代码;否则可选用 Speed,即对给出作业生成速度较快的目标代码。因此,本项可认为是对空间或时间的优化选择。

—6—1—4—2 Use register variables On

选择 On,编译时寄存器变量自动分配给用户程序,否则即使在你的程序里用了关键字 register,编译程序也无法使用寄存器变量。一般选用 On,除非你用的是不支持寄存器变量的虚拟汇编代码。

用户程序可以使用伪变量(或称【准变量】),如 —AX 等来访问寄存器 AX 等。一个伪变量用相应寄存器名前加下划线标识。

这一选择的主要用处是考虑到有些计算机语言并不使用寄存器变量。

—6—1—4—3 Register optimization Off

寄存器优化开关。当用 On 时,通过记住寄存器的内容和尽可能多的重复使用来抑制过多的取数操作(Load operation)。或者说,使用本命令能避免重复装入已在寄存器里的值,这样可减少指令并把对存储器的访问改为对寄存器的访问。不过,使用它有时会出现问题,因为编译器无法知道寄存器值中间有否被某一指针间接地修改过。因此使用应谨慎。

```
C>TYPE MENU6.C
```

```
main()
{ int a=4,b,*p;
  b=a;
```

```

p=&a;    /* 指针 p 指向变量 a */
*p=b+5;  /* a 的值被间接修改 */
printf("%d\n",a);
}

```

分别按 Register optimization Off 和 On 在 IBM PC AT 机上编译后执行,结果前者得到正确值 9 而后者得到的是错误值 4。这是什么原因? 因为 TC 在调用 printf 语句时不能正确处理语句

```

*p=b+5;

```

的结果(只把 AX 寄存器的值压入了堆栈,而没有把存储器中正确值压入堆栈!),它没有发现指针 p 所指的变量 a 的值已改变,它记的还是原来的值 4! 但如果将该句改成

```

a=b+5;

```

TC 就能发现 a 值已被改变了! (或者你不改这句,而将语句

```

printf("%d\n",a);

```

改成

```

printf("%d\n", *p);

```

结果两者都能得到正确结果)。

此例可以作为开关设置为不同状态时校验程序正确性的辅助方法。

当函数调用或指针到达转移指令能到的地方(如标号、case 语句、循环语句的开始和结束等),寄存器的内容都将丢失,另外 8086 处理器中的寄存器数量较少,多数程序使用这种优化不会出现错误。

—6—1—4—4 Jump optimization Off

跳转优化开关。当置成 On 时,能消去多余的跳转和重新调整循环及开关 (switch) 语句,从而压缩代码。循环调整能加快内循环的执行速度。注意:对集成调试器这可能出错,因为可能有若干源代码行对应同一组执行代码。所以,若要调试,开关置为 Off 较好。

手工优化程序方法举例如下。

例 1 跳转优化

优化前	优化后
void foo(int s, int t)	void foo(int s, int t)
{ if(s>t) return ;	{ begin: /* 增加标号 */
.....	if(s>t) return ;
foo(s,t); /* 递归调用 */	goto begin; /* 改成跳转 */
}	}

这样可避免因递归而执行大量数据重载及堆栈操作。

例 2 开关优化

优化前	优化后
void fun (int c)	void fun (int c)
{ switch(c)	{ switch(c)
{ case 1,	{ case 1:

<pre> printf("ok!"); break; case 2: } return ; } </pre>	<pre> Printf("ok!"); return; case 2: } return ; } </pre>
---	--

将 break 改成 return 可避免 break 语句在编译时产生 JMP 指令, 执行速度加快。

例 3 循环优化

(1) 将不变表达式提到循环外

```

t=0; for(i=0; i<80; i++) s[i]=a*b*c;
可改成
t=a*b*c; for(i=0; i<80; i++) s[i]=t;

```

(2) 循环强度削减

```

int s[80];
for(n=0; n<80; n++) s[n]=n*8;
可改成
int s[80];
register int *p, t;
for(n=0, p=s, t=0; n<80; n++, p++) { *p=t; t+=8; }

```

即用运算快的指令代替慢的指令, 为此可能要增加变量。

(3) 循环变量约减

```

for(n=0, p=s, t=0; n<80; n++, p++) { *p=t; t+=8; }
改成
for(n=0, p=s; n<640; n+=8, p++) *p=n;

```

(4) 循环扩展

```

for(k=0; k<100; k++) x[k]=y[k]*4;
改成
for(k=0; k<100; k+=2)
{ x[k]=y[k]*4; x[k+1]=y[k+1]*4; }

```

通过增加循环语句来减小循环次数。当然, 如改用指针来指向数组, 然后对指针执行循环可能效果更好。

(5) 循环嵌套次序调整

```

x=0; for(i=0; i<100; i++)
for(k=1; k<4; k++) x+=2;
不如改成
x=0; for(k=1; k<4; k++)
for(i=0; i<100; i++) x+=2;

```

多重循环中, 尽量用循环次数多的作内循环, 效果好一些。

—6—1—5 Source

设置 (源程序的) 源代码编辑方式。它将在编译初始化阶段决定处理源码的方式。

—6—1—5—1 Identifier length 32

说明被定义的标识符有效长度。即定义变量、预处理宏名、结构成员名等用的标识符前多少个字符才被认为可以作为区别用。例如,定义了 4,则标识符

FINIS

和

FINISH

被认为是一样的,因为它们前 4 个字符完全相同。当 TCINST 状态行上出现

Specify new Identifier length size [1..32] 32

或在集成环境下有

Maximum length

小窗时,你可以输入 1 到 32 个字符(缺省是 32 个。注意:其它语言可能没有这么慷慨,有的只有 7 个或更少,如考虑程序可移植性,应当注意这一点;否则,可定义长一些,这样能使标识符用较多字符定义,因而更具直观性)。

—6—1—5—2 Nested comments Off

源程序允许进行嵌套注释。TC 的一般注释是不嵌套的(开关置为 Off),即形如

```
/* 注释 */
```

但有时确实需要形如

```
/* 注释一 /* 注释内的注释 */ */
```

这就是注释嵌套。嵌套注释在一般 TC 里是不允许的,且不可移植。但在本命令置为 On 时,TC 可对含有嵌套注释的 C 源文件进行编译。

注意:标准的 C 语言是不支持嵌套注释的,若考虑移植性,最好将该开关置为 off。

—6—1—5—3 ANSI keywords only Off

若想让编译只识别 ANSI 关键字,而把 TC 的扩展关键字(包括 asm、cdecl、far、huge、interrupt、near、pascal、_cs、_ds、_es、_ss、_AX、_AH、_AL、_BX、_BH、_BL、_CX、_CH、_CL、_DX、_DH、_DL、_BP、_SP、_SI、_DI)只当一般标识符对待(即不是关键字),便可将本命令置为 On。

该命令定义为 On 时也定义了编译时用的符号——STDC——。

—6—1—6 Errors

利用本菜单上的七种命令,程序员可以控制 TC 编译器按需处理和响应诊断信息。

—6—1—6—1 Errors ;stop after 25

此任选项使得编译器在连续发现了预定检测出的错误个数(缺省为 25 个)后便停止编译。因此,对错误较多的 C 源程序在试编译时(以便查阅和消去错误),可将此数定小一些,这样既可节省时间,又可避免信息窗内容太多。

当将光条移到此菜单上并回车后,TCINST 状态行显示

Specify new Errors Size[0..100] 25

(或对 TC.EXE 有 Maximum Errors 提示小窗),这时可输入 0 到 100 之间的任一数。数 0 将导致编译器一直编译下去,或达到错误上界(最大值)时停止。

—6—1—6—2 Warnings ;stop after 100

与 Errors ;stop after 25 类同,不过这里指的是警告,而不是错误。警告个数为 0 到 100。

缺省值为 100。

—6—1—6—3 Display warnings On

显示警告开关。如开关为 On,则以下几类警告(共 27 种,见下面详述)。如发生且子菜单开关为 On 则显示,否则即使发生警告也不显示。

1. Portability warnings 移植警告
2. ANSI violations ANSI 关键字违反
3. Common errors 常见错误
4. Less common errors 少见错误

—6—1—6—4 Portability warnings

移植警告信息显示控制。它指出警告语句可能在别的计算机上不一定能正确执行。

—6—1—6—4—1 A:Non-portable pointer conversion On

不可移植的指针转换。

—6—1—6—4—2 B:Non-portable pointer assignment On

不可移植的指针赋值。

—6—1—6—4—3 C:Non-portable pointer comparison On

不可移植的指针比较。

—6—1—6—4—4 D:Constant out of range in comparison On

比较时常量超出了范围。

—6—1—6—4—5 E:Constant is long Off

常量应是 long 类型,可是没定义成 long。

—6—1—6—4—6 F:Conversion may lose significant digits Off

转换可能会丢失有效数字。

—6—1—6—4—7 G:Mixing pointers to signed and unsigned char Off

混淆了 signed 和 unsigned 字符指针。

—6—1—6—5 ANSI violations

ANSI 关键字违反警告信息,有八种情况:

—6—1—6—5—1 A:'ident' not part of structure On

ident 标识符不是结构的一部分。

—6—1—6—5—2 B:Zero length structure On

结构长度为零。

—6—1—6—5—3 C:Void functions may not return a value On

void 型函数不可以返回一个值。

—6—1—6—5—4 D:Both return and return of a value used On

编译程序发现两个返回语句(return)的返回结果类型不一致。

—6—1—6—5—5 E:Suspicious pointer conversion On

值得怀疑的指针转换。

—6—1—6—5—6 F:Underfined structure 'ident' On

结构的标识符 ident 未定义。

—6—1—6—5—7 G:Redefinition of 'ident' is not identical On

标识符 ident 重新定义时不唯一。

—6—1—6—5—8 H:Hexadecimal or octal constant too large On

十六进制或八进制数太大。

—6—1—6—6 Common errors

常见错误,有七种情况。这些警告信息指出,尽管没有违反 C 语言,但会产生错误结果。

—6—1—6—6—1 A;Function should return a value Off

函数应返回一个值。

—6—1—6—6—2 B;Unreachable code On

不可达代码。

—6—1—6—6—3 C;Code has no effect On

代码无效。

—6—1—6—6—4 D;Possible use of 'ident' before definition On

在定义标识符 ident 的前面可能已使用了该标识符。

—6—1—6—6—5 E;'ident' is assigned a value which is never used On

标识符 ident 被赋以一个不使用的值。

—6—1—6—6—6 F;Parameter 'ident' is never used On

参数 ident 从来没有被使用。

—6—1—6—6—7 G;Possibly incorrect assignment On

可能是不正确的赋值。

—6—1—6—7 Less common errors

少见错误,有六种情况。这些错误也将产生错误结果。

—6—1—6—7—1 A;Superfluous & with function or array Off

在函数或数组中有多余的符号 &。

—6—1—6—7—2 B;'ident' declared but never used Off

ident 虽被说明但未使用。

—6—1—6—7—3 C;Ambiguous operators need parentheses Off

二义性操作符需要加括号。

—6—1—6—7—4 D;Structure passed by value Off

现在结构是按值传送,看看是否需按参数传递(结构按值传送或当参数传递均是允许的,此警告不过是一个忠告)。

—6—1—6—7—5 E;No declaration for function 'ident' Off

函数标识符 ident 没有说明。

—6—1—6—7—6 F;Call to function with no prototype Off

调用无原型的函数。

—6—1—7 Names

改变代码、数据或 BSS 的段、组或类名。星号告诉编译器使用缺省名。

一个段 (Segment) 是最多 64K 字节长度的相邻的内存区域,一般可位于内存的任何地方。组 (Group) 是在 64K 字节内存内段的集合。组用于在内存中对段进行寻址。段不必是相邻地构成一组。任一组的地址就是这组中最低的段地址。一个程序可以有一个或多个组。类 (Class) 是段的集合,命名段为类可以控制段在内存中的次序和相对位置。装入类可以越过 64K 字节的边界,而组可以跨越类。

注意:包含程序指令的代码段当名字相同时将作为一块连续的代码区被装配。同样,具有相同名字的数据段也将这样做。

—6—1—7—1 Code names' 设代码段名

—6—1—7—1—1 Segment name * 段名

—6—1—7—1—2 Group name * 组名

—6—1—7—1—3 Class name * 类名

—6—1—7—2 Data names 设数据段名

—6—1—7—2—1 Segment name * 段名

—6—1—7—2—2 Group name * 组名
—6—1—7—2—3 Class name * 类名

—6—1—7—3 BSS names 设 BSS 名
—6—1—7—3—1 Segment name * 段名
—6—1—7—3—2 Group name * 组名
—6—1—7—3—3 Class name * 类名

—6—2 Linker

本菜单是设置连结器缺省参数。

—6—2—1 Map file Off

选择映射 (MAP) 文件 (扩展名为 .MAP) 的类型,有四种。取 Off 以外值时产生的映射文件就会被放进由 Options/Directories/Output directory 指定的目录中。

—6—2—1—1 Off: 不产生映射文件。
—6—2—1—2 Segments: 在映射文件中只包含段
—6—2—1—3 Publics: 在映射文件中包含段和公共符号
—6—2—1—4 Detailed: 在映射文件中包含详细的段映像

下面用实例说明: 现对文件 C>MENU7.C

```
main()  
{  
printf("ok! \n");  
}
```

先将本命令选 Off (其它开关用 TC.EXE 内部缺省值) 编译后,在集成环境下选 File/OS Shell 进入 DOS,则在输出目录内没有发现 MAP 文件产生。然后用

C>EXIT

再进入集成环境,并将本命令选 Segments,则有(段地址是相对地址,不是装入时的绝对地址)

C>TYPE MENU7.MAP

Start	Stop	Length	Name	Class
00000H	0135AH	0135BH	—TEXT	CODE
01360H	01779H	0041AH	—DATA	DATA
0177AH	0177DH	00004H	—EMUSEG	DATA
0177EH	0177FH	00002H	—CRTSEG	DATA
01780H	01781H	00002H	—CVTSEG	DATA
01782H	01787H	00006H	—SCNSEG	DATA
01788H	017CDH	00046H	—BSS	BSS
017CEH	017CEH	00000H	—BSEND	STACK
017D0H	0184FH	00080H	—STACK	STACK

Program entry point at 0000:0000

现在将目录中的 MENU7.* 文件取消后再进入集成环境,并将开关取 Publics,则有

C>TYPE MENU7.MAP

Start	Stop	Length	Name	Class
-------	------	--------	------	-------

..... (注：因此部分内容同选 Segment 一样，故未写出)

Address	Publics by Name	(按名字的字母顺序排列)
0000:01F8	DGROUP@	
0136:0188	emws—adjust	
0136:018C	emws—BPsafe	
0136:0184	emws—control	
0136:018A	emws—fixSeg	
0136:016E	emws—initialSP	
0136:00AE	emws—limitSP	
0136:017E	emws—nmiVector	
0136:017A	emws—saveVector	
0136:018E	emws—stamp	
0136:0182	emws—status	
0136:0186	emws—TOS	
0136:0192	emws—version	
0000:01E2	—abort	
0000:03AD	—atexit	
0000:0578	—brk	
0136:0088	—environ	
0136:0094	—errno	
0000:0243	—exit	
0000:0A3D	—fflush	
0000:0AED	—fputc	
0000:0C1D	—fputchar	
0000:1334	—free	
0000:05DF	—fseek	
0000:0643	—ftell	
0000:06A2	—isatty	
0000:09B0	—itoa	
0000:0906	—lseek	
0000:09F5	—ltoa	
0000:01FA	—main	
0000:04B2	—malloc	
0000:0ABB	—printf	
0000:0586	—sbrk	
0000:06BA	—setvbuf	
0000:09D9	—ultoa	
0000:07AD	—write	
0136:0096	——8087	
0136:0084	——argc	
0136:0086	——argv	
0136:0200	——atexitent	
0136:0428	——atexittbl	
0136:00A6	——brklvl	
0000:0000 Abs	——cvtfak	
0136:019A	——doserrno	

0136:019C	——dosErrorToSV
0136:008A	——envLng
0136:008C	——envseg
0136:008E	——envSize
0000:0121	——exit
0136:01F6	——exitbuf
0136:01F8	——exitfopen
0136:01FA	——exitopen
0000:0AD4	——fputc
0000:0C33	——FPUTN
0136:00A2	——heapbase
0136:01FC	——heaplen
0136:00AA	——heaptop
0136:0074	——Int0Vector
0136:0078	——Int4Vector
0136:007C	——Int5Vector
0136:0080	——Int6Vector
0000:0207	——IOERROR
0000:0931	——LONGTOA
0136:0342	——openfd
0136:0092	——osmajor
0136:0093	——osminor
0136:0090	——psp
0000:0CED	——REALCVT
0136:0420	——RealCvtVector
0000:01A5	——restorezero
0136:0422	——ScanTodVector
0000:0278	——setargv
0000:0363	——setenvp
0136:0098	——StartTime
0136:036A	——stdinStarted
0136:036C	——stdoutStarted
0136:01FE	——stklen
0136:0202	——streams
0136:0092	——version
0000:0D1C	——VPRINTER
0000:08C0	——write
0000:078C	——xfflush
0000:0520	———brk
0136:009E	———brklvl
0136:046C	———first
0136:009C	———heapbase
0136:00A0	———heaptop
0136:0468	———last
0000:03D3	———pull—free—block
0136:046A	———rover

0000:0544	——sbrk	(按实际地址值排列)
Address	Publics by Value	
0000:0000 Abs	——cvt fak	
0000:0121	——exit	
0000:01A5	——restorezero	
0000:01E2	——abort	
0000:01F8	DGROUP@	
0000:01FA	——main	
0000:0207	——IOERROR	
0000:0243	——exit	
0000:0278	——setargv	
0000:0363	——setenvp	
0000:03AD	——atexit	
0000:03D3	——pull——free——block	
0000:04B2	——malloc	
0000:0520	——brk	
0000:0544	——sbrk	
0000:0578	——brk	
0000S:0586	——sbrk	
0000:05DF	——fseek	
0000:0643	——ftell	
0000:06A2	——isatty	
0000:06BA	——setvbuf	
0000:078C	——xflush	
0000:07AD	——write	
0000:08C0	——write	
0000:0906	——lseek	
0000:0931	——LONGTOA	
0000:09B0	——itoa	
0000:09D9	——ultoa	
0000:09F5	——ltoa	
0000:0A3D	——fflush	
0000:0ABB	——printf	
0000:0AD4	——fputc	
0000:0AED	——fputc	
0000:0C1D	——fputchar	
0000:0C33	——FPUTN	
0000:0CED	——REALCVT	
0000:0D1C	——VPRINTER	
0000:1334	——free	
0136:0074	——Int0Vector	
0136:0078	——Int4Vector	
0136:007C	——Int5Vector	
0136:0080	——Int6Vector	
0136:0084	——argc	
0136:0086	——argv	

0136:0088	—environ
0136:008A	——envLng
0136:008C	——envseg
0136:008E	——envSize
0136:0090	——psp
0136:0092	——version
0136:0092	——osmajor
0136:0093	——osminor
0136:0094	——errno
0136:0096	——8087
0136:0098	——StartTime
0136:009C	——heapbase
0136:009E	——brklvl
0136:00A0	——heaptop
0136:00A2	——heapbase
0136:00A6	——brklvl
0136:00AA	——heaptop
0136:00AE	emws—limitSP
0136:016E	emws—initialSP
0136:017A	emws—saveVector
0136:017E	emws—nmiVector
0136:0182	emws—status
0136:0184	emws—control
0136:0186	emws—TOS
0136:0188	emws—adjust
0136:018A	emws—fixSeg
0136:018C	emws—BPsafe
0136:018E	emws—stamp
0136:0192	emws—version
0136:019A	——doserrno
0136:019C	——dosErrorToSV
0136:01F6	——exitbuf
0136:01F8	——exitfopen
0136:01FA	——exitopen
0136:01FC	——heaplen
0136:01FE	——stklen
0136:0200	——atexitcnt
0136:0202	——streams
0136:0342	——openfd
0136:036A	——stdinStarted
0136:036C	——stdoutStarted
0136:0420	——RealCvtVector
0136:0422	——ScanTodVector
0136:0428	——atexittbl
0136:0468	——last
0136:046A	——rover

0136:046C ———first

Program entry point at 0000,0000

最后将开关取成 Detailed,就有

C>TYPE MENU7.MAP

Start Stop Length Name Class
.....(注:因此部分内容同选 Segment 一样,故未写出)
(C=类 S=段 G=组 M=模块)

Detailed map of segments

0000,0000	01FA	C=CODE	S=—TEXT	G=(none)	M=C0S	ACBP=28
0000,01FA	000D	C=CODE	S=—TEXT	G=(none)	M=P.C	ACBP=28
0000,0207	003B	C=CODE	S=—TEXT	G=(none)	M=IOERROR	ACBP=28
0000,0242	0030	C=CODE	S=—TEXT	G=(none)	M=EXIT	ACBP=28
0000,0272	0000	C=CODE	S=—TEXT	G=(none)	M=HEAPLEN	ACBP=28
0000,0272	00F1	C=CODE	S=—TEXT	G=(none)	M=SETARGV	ACBP=28
0000,0363	004A	C=CODE	S=—TEXT	G=(none)	M=SETENV	ACBP=28
0000,03AD	0000	C=CODE	S=—TEXT	G=(none)	M=STKLEN	ACBP=28
0000,03AD	0026	C=CODE	S=—TEXT	G=(none)	M=ATEXIT	ACBP=28
0000,03D3	014D	C=CODE	S=—TEXT	G=(none)	M=MALLOC	ACBP=28
0000,0520	0078	C=CODE	S=—TEXT	G=(none)	M=BRK	ACBP=28
0000,0598	0000	C=CODE	S=—TEXT	G=(none)	M=FILES	ACBP=28
0000,0598	0000	C=CODE	S=—TEXT	G=(none)	M=FILES2	ACBP=28
0000,0598	010A	C=CODE	S=—TEXT	G=(none)	M=FSEEK	ACBP=28
0000,06A2	0018	C=CODE	S=—TEXT	G=(none)	M=ISATTY	ACBP=28
0000,06BA	00D2	C=CODE	S=—TEXT	G=(none)	M=SETVBUF	ACBP=28
0000,078C	0021	C=CODE	S=—TEXT	G=(none)	M=XFFLUSH	ACBP=28
0000,07AD	0113	C=CODE	S=—TEXT	G=(none)	M=WRITE	ACBP=28
0000,08C0	0046	C=CODE	S=—TEXT	G=(none)	M=WRITEA	ACBP=28
0000,0906	002B	C=CODE	S=—TEXT	G=(none)	M=LSEEK	ACBP=28
0000,0931	00EB	C=CODE	S=—TEXT	G=(none)	M=LTOA	ACBP=28
0000,0A1C	0021	C=CODE	S=—TEXT	G=(none)	M=CVTFAK	ACBP=28
0000,0A3D	007E	C=CODE	S=—TEXT	G=(none)	M=FFLUSH	ACBP=28
0000,0ABB	0019	C=CODE	S=—TEXT	G=(none)	M=PRINTF	ACBP=28
0000,0AD4	0219	C=CODE	S=—TEXT	G=(none)	M=PUTC	ACBP=28
0000,0CED	0004	C=CODE	S=—TEXT	G=(none)	M=REALCVT	ACBP=28
0000,0CF1	052C	C=CODE	S=—TEXT	G=(none)	M=VPRINTER	ACBP=28
0000,121D	013E	C=CODE	S=—TEXT	G=(none)	M=FREE	ACBP=28
0136,0000	0194	C=DATA	S=—DATA	G=DGROUP	M=C0S	ACBP=68
0136,0194	0005	C=DATA	S=—DATA	G=DGROUP	M=P.C	ACBP=48
0136,019A	005B	C=DATA	S=—DATA	G=(none)	M=IOERROR	ACBP=48
0136,01F6	0006	C=DATA	S=—DATA	G=DGROUP	M=EXIT	ACBP=48
0136,01FC	0002	C=DATA	S=—DATA	G=DGROUP	M=HEAPLEN	ACBP=48
0136,01FE	0000	C=DATA	S=—DATA	G=DGROUP	M=SETARGV	ACBP=48
0136,01FE	0000	C=DATA	S=—DATA	G=DGROUP	M=SETENV	ACBP=48
0136,01FE	0002	C=DATA	S=—DATA	G=DGROUP	M=STKLEN	ACBP=48
0136,0200	0002	C=DATA	S=—DATA	G=DGROUP	M=ATEXIT	ACBP=48

0136:0202	0000	C=DATA	S=-DATA	G=DGROUP	M=MALLOC	ACBP=48
0136:0202	0000	C=DATA	S=-DATA	G=(none)	M=BRK	ACBP=48
0136:0202	0140	C=DATA	S=-DATA	G=DGROUP	M=FILES	ACBP=48
0136:0342	0028	C=DATA	S=-DATA	G=DGROUP	M=FILES2	ACBP=48
0136:036A	0000	C=DATA	S=-DATA	G=DGROUP	M=FSEEK	ACBP=48
0136:036A	0000	C=DATA	S=-DATA	G=(none)	M=ISATTY	ACBP=48
0136:036A	0004	C=DATA	S=-DATA	G=DGROUP	M=SETVBUF	ACBP=48
0136:036E	0000	C=DATA	S=-DATA	G=DGROUP	M=XFFLUSH	ACBP=48
0136:036E	0000	C=DATA	S=-DATA	G=DGROUP	M=WRITE	ACBP=48
0136:036E	0000	C=DATA	S=-DATA	G=(none)	M=WRITEA	ACBP=48
0136:036E	0000	C=DATA	S=-DATA	G=(none)	M=LSEEK	ACBP=48
0136:036E	0000	C=DATA	S=-DATA	G=(none)	M=LTOA	ACBP=48
0136:036E	0031	C=DATA	S=-DATA	G=DGROUP	M=CVTFAK	ACBP=48
0136:03A0	0000	C=DATA	S=-DATA	G=DGROUP	M=FFLUSH	ACBP=48
0136:03A0	0000	C=DATA	S=-DATA	G=DGROUP	M=PRINTF	ACBP=48
0136:03A0	0001	C=DATA	S=-DATA	G=DGROUP	M=PUTC	ACBP=48
0136:03A2	0077	C=DATA	S=-DATA	G=DGROUP	M=VPRINTER	ACBP=48
0136:041A	0000	C=DATA	S=-DATA	G=DGROUP	M=FREE	ACBP=48
0177:000A	0004	C=DATA	S=-EMUSEG	G=DGROUP	M=COS	ACBP=58
0177:000E	0002	C=DATA	S=-CRTSEGG	G=DGROUP	M=COS	ACBP=58
0178:0000	0000	C=DATA	S=-CVTSEGG	G=DGROUP	M=COS	ACBP=48
0178:0000	0002	C=DATA	S=-CVTSEGG	G=(none)	M=CVTFAK	ACBP=48
0178:0002	0000	C=DATA	S=-CVTSEGG	G=DGROUP	M=REALCVT	ACBP=48
0178:0002	0000	C=DATA	S=-SCNSEGG	G=DGROUP	M=COS	ACBP=48
0178:0002	0006	C=DATA	S=-SCNSEGG	G=(none)	M=CVTFAK	ACBP=48
0178:0008	0000	C=BSS	S=-BSS	G=DGROUP	M=COS	ACBP=48
0178:0008	0000	C=BSS	S=-BSS	G=DGROUP	M=P.C	ACBP=48
0178:0008	0000	C=BSS	S=-BSS	G=DGROUP	M=IOERROR	ACBP=48
0178:0008	0000	C=BSS	S=-BSS	G=DGROUP	M=EXIT	ACBP=48
0178:0008	0000	C=BSS	S=-BSS	G=DGROUP	M=HEAPLEN	ACBP=48
0178:0008	0000	C=BSS	S=-BSS	G=DGROUP	M=STKLEN	ACBP=48
0178:0008	0040	C=BSS	S=-BSS	G=DGROUP	M=ATEXIT	ACBP=48
0178:0048	0006	C=BSS	S=-BSS	G=DGROUP	M=MALLOC	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=BRK	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=FILES	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=FILES2	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=FSEEK	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=ISATTY	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=SETVBUF	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=XFFLUSH	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=WRITE	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=WRITEA	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=LSEEK	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=LTOA	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=FFLUSH	ACBP=48
0178:004E	0000	C=BSS	S=-BSS	G=DGROUP	M=PRINTF	ACBP=48


```

0178:004E 0000 C=BSS S=-BSS G=DGROUP M=PUTC ACBP=48
0178:004E 0000 C=BSS S=-BSS G=DGROUP M=VPRINTER ACBP=48
0178:004E 0000 C=BSS S=-BSS G=DGROUP M=FREE ACBP=48
017C:000E 0000 C=STACK S=-BSENDG=DGROUP M=COS ACBP=28
017D:0000 0080 C=STACK S=-STACK G=(none) M=COS ACBP=74

```

Address Publics by Name

.....(注:因以下部分内容同选 Publics 一样,故未写出)

Program entry point at 0000,0000

—6—2—2 Initialize segments Off

初始化全部未初始化段的开关。选 On 则连接器进行段初始化,结果生成的 *.EXE 文件比必要的要大,因此一般将开关置成 Off。

—6—2—3 Default libraries Off

当连接非 C 编译器产生的模块时,那些编译器可能已在目标文件中放入了一个缺省库表。当本命令置为 On 时,连接器就会试图在这些库和 TC 提供的缺省库中查找未定义的过程。当开关为 Off 时则忽略查找。

—6—2—4 Graphics library On

打开或关闭自动查找 BGI 图形库(和驱动程序 *.BGI 对应的 *.OBJ 文件构成)开关。

1. 置 On 时,连接器除标准 TC 库外,还自动查找并建立源文件中涉及到的 BGI 图形库程序,而不必使用规划文件(*.PRJ)。

2. 置 Off 时,可分两种情况,一是当在规划文件中已写入了与源程序相关的 BGI 图形库名,则 TC 直接建立使用到的 BGI 图形库程序;另一种情况是规划文件中没写入,则不连接任何图形库。总之,使用 Off 可加快连接速度,但是由于一般源程序涉及的图形库次数较少,所以开关的缺省值仍为 On。注意,当你需要用 Turbo C 图形库函数时,如果又没有将 graphics.lib 放入规划文件中,则必须将此项选 On。

—6—2—5 Warn duplicate symbols Off

目标文件(*.OBJ)和库文件(*.LIB)里可能出现相同符号,利用本命令在库中发现重复符号时显示警告(On)或不显示(Off)。当为 Off 时连接程序会自动选择正确的符号。

—6—2—6 Stack warning On

靠连接器产生无堆栈(No Stack)警告信息(On)或不显示(Off)。用微存储模式(Tiny model)编译时容易出现这种警告。

—6—2—7 Case-sensitive Link On

连接时区别大小写字母的开关。因为 C 通常规定大小写字母是有区别的(即 C 语言是对大小写敏感的,Case-sensitive),因此本命令缺省值应是 On。为 Off 时则不区分大小写。

—6—3 Environment

设置 TC 缺省工作环境。

—6—3—1 Message Tracking Current File [Off] [All Files]

TC 编译和调试源程序时出现的错误和警告消息在【消息窗口, Message 窗口】内显示。

1. Current File: 只跟踪在编辑器内的文件。若编译器发现了一个不在当前编辑窗内显示的源文件中的一个错误,则该错误会在消息窗中出现(当前消息上有亮条显示)。如果用户选择有亮条的错误消息(只要按回车键),则与该错误消息对应的非当前编辑文件立即被调入编辑窗内,而原编辑窗内的

文件自行消失(如要对它编辑,只有重新装入)。

如果你对消息窗亮条项没有按回车键,而是按了 F6 键,则激活当前编辑窗,编辑窗内还是原来那个源文件,即发现错误的那个源文件未被装入。

2. All Files: 自动跟踪并装入与错误消息相对应的每个文件,可以看见每一条错误对应的出错位置。

3. Off: 不自动跟踪错误消息。在这种情况下,如果需要可按回车键,则错误消息所在文件会被加载进编辑器。

说明:编译时如果出现错误或警告时,显示的编译窗口内会出现

Press anykey

即按任一键的提示。当你按一键后,消息窗立即被激活,一个亮条出现在第一个错误或警告上。与此同时,编辑窗口内也有一亮条显示,它标志着编译器给出的错误或警告在源代码中的相应位置。当然,有时发生错误的真正位置可能在此之前。

这时可用光标键将消息窗口中的亮条上下移动。可以发现,编辑窗口内的亮条也会随之作相应的移动。如果消息窗口内某条消息内容太长,可用左右光标键滚动消息条,也可用 Home 或 End 键立即移到消息条首部或末尾。当消息条数太多时可用 F5 键放大消息窗口。

可按 F6 键从编辑窗重新进入消息窗,寻找想修改的下一条消息。也可用 Alt-F7 或 Alt-F8 键移到前一消息或后一消息。注意:Alt-F7 或 Alt-F8 键不受 Message Tracking 开关设置的影响,它们总能找出前一错误或后一错误,必要时还会加载源文件。

—6—3—2 Keep Messages Off

当开关为 On(或 Yes)时,则保存当前消息窗口内的错误信息,且把进一步的信息附在后面(编译时只有重新被编译的文件消息被清除,然后被新的消息替代);否则,每次编译前信息窗口内的内容被全部清除。当编译含有多个源文件的程序时,应将它置成 On 较好。

什么时候不要消息了,可以用 Project/Remove messages 命令将消息消去。

—6—3—3 Config auto save Off

本命令自动保存 TC 缺省的工作环境参数(称 TC 配置)。保存此参数的文件称【TC 配置文件】(TC Config file)。注意,TC 配置文件和【命令行配置文件】是两种完全不同的文件。命令行配置文件 TURBOC.CFG 与 TC 配置文件之间可用 TCCONFIG.EXE 文件实行相互转换。

当经常要编辑或调试某一程序时,可给它配置一个配置文件。以后,每当装入该配置文件后,对应程序的原有状态全被自动恢复。因此,配置文件有时是很有用的。

TC 配置文件是二进制文件,通常用扩展名 .TC 标识(当然,必要时你也可以不用此扩展名),缺省的 TC 配置文件名规定为 TCCONFIG.TC。它主要保存供 TC.EXE 调用并替换 TC.EXE 内部某些缺省值的特殊值。这些特殊值和集成开发环境本身有关,涉及象 P/Project Name、O/D/Pick File Name 和 O/Environment 等菜单项。

1. TC.EXE 启动与 TCCONFIG.TC、TCINST.EXE 的关系

首次进入 TC 集成环境时是没有象 TCCONFIG.TC 这样的配置文件的(在目录中你查不到它),TC.EXE 启动时所有菜单开关配置均用其内部缺省值。要想修改这些内部缺省值应靠 TCINST.EXE。事实上,TC.EXE 启动时如果你没有使用 /C 开关,那么 TC.EXE 将自动先在当前目录中寻找 TCCONFIG.TC,找到后则以 TCCONFIG.TC 中的值配置各菜单;如在当前目录中找不到,则在用 TCINST.EXE 设置的(即已装入 TC.EXE 的)TC 目录中去找,找到 TCCONFIG.TC,则用它配置;否则,TC.EXE 便直接用内部缺省值。

如果在启动 TC.EXE 时使用了开关 /C(注意:在开关与文件名之间无空格),例如

```
C>TC□/cmyconfig
```

则 TC.EXE 会在当前目录中去找 MYCONFIG.TC 文件,假定找到,则以 MYCONFIG.TC 的值配置 TC.EXE 的菜单;否则,发出没有找到的警告

Could not access MYCONFIG.TC. Press ESC.

压 ESC 键后,TC.EXE 便不再查找其它 TC 配置文件,而直接以内部缺省值配置菜单。

综上所述,TCINST.EXE 用来设置 TC.EXE 内部的环境缺省值,使用 TC 配置文件启动 TC.EXE 则指定的 TC 配置文件值将优先选用,被配置 TC.EXE 菜单的参数。未用 TC 配置文件,TC.EXE 便用内部缺省值配置菜单参数。

```
/ * tcmenu.03z */
```

2. 怎样建立 TC 配置文件

必须在集成开发环境下建立!在集成环境下,选 Options/Environment/Config Auto Save 菜单(即相当于这里叙述的开关)置为 On,如果你未选 Options/Save Options 菜单,而选用

Run/Run

或

File/OS Shell

或

File/Quit

菜单后,当前设置的配置值便存放到你早先指定的 TC 配置文件中(用开关 /C 或集成菜单中的 Options/Retrieve Options 菜单调入的);如未指定,一定存入 TCCONFIG.TC,如原先 TCCONFIG.TC 不存在,则 TC.EXE 自动创建它)。假定你没有选 Run/Run(或 File/OS Shell 及 File/Quit)菜单,而是选了

Options/Save Options

菜单,那么当输入一个配置文件名(如果不输文件名,TC.EXE 用缺省文件名 TCCONFIG.TC),则当前配置立即存入指定的文件中(此时,与

Options/Environment/Config Auto Save

开关状态无关)。

从上可见,这里讲的 TCINST.EXE 的 Options/Environment/Config Auto Save 菜单就相当于集成环境中的对应菜单,当置成 On 时它的作用已很清楚。

3. 如何装入并修改早先设置的 TC 配置文件

如前所述,装入指定的 TC 配置文件可在 TC.EXE 启动时使用开关 /C,但也可以在集成环境下做到:选

Options/Retrieve Options

菜单,如果你键入一个要调入的 TC 配置文件名,则该文件便被自动调入,修改好菜单参数后按上述方法即可将改后(即当前环境)存入指定 TC 配置文件。

注意:如果对输入文件名时使用了 DOS 通配符 * 或 ?,也是允许的,不过随后将给出列表,你可用按回车从中选一文件名作为输入。

本命令的作用至此已很清楚。

—6—3—4 Edit auto save Off

自动保存当前正在编辑的源文件开关。在集成环境中,当将开关置成 On 时,如果你 是新编辑一个文件 MYFILE.C(早先盘上并无此文件),当选用

File/Quit

时,TC.EXE 将显示

MYFILE.C not Save? (Y/N)

询问你是否要存盘。当选用

Run/Run

时,程序执行前先将当前编辑窗内的文件无条件存盘,然后才开始执行。这样做的好处时以免执行一个有问题的程序时发生诸如死机这样的情况,导至源程序全部丢失,而无法侦查原程序的问题。

当选用

File/OS Shell

时,TC.EXE 自动将 MYFILE.C 存盘后进入 DOS。注意,此时你不能用 C>TC 方式重新进入集成环境,那样将

Program too big fit in memory

程序不能装入内存,而只能用 C>EXIT

命令重入。

事实上,C 编辑器是将当前编辑文件存在内存的一片区域内的,如果不将开关

Edit auto save

置成 On,那么在用 File/OS Shell 时它是不会自动写入磁盘的。

—6—3—5 Backup source files On

自动建立备份文件(扩展名为 .BAK)。如本命令设置为 On,那么在集成环境下,用 File/Save 菜单将当前编辑的源文件存盘的同时,自动建立后备文件。为了节省存储空间,例如磁盘可用空间有限时才将它置成 Off。

—6—3—6 Tab size 8

设定横向制表位置的字符数。当回车后提示行出现

Specify new tab size:[2..16] 8

(或 Editor Tab Size),可输入 2 到 16 之间的数字。当输入数超过 16 时,将显示

Bad value. Legal range 2-16. Press ESC.

按 ESC 键后可重输新数字。

此尺寸可保存到配置文件里。

—6—3—7 Zoomed windows Off F5

放大【编辑窗口】(Edit 窗口,编辑源程序的窗口)或消息窗口到整屏的开关。选 On,放大;否则缩小(即回到原来两个窗口同时显示的状态)。在放大状态,只有【活动窗】可见,活动窗可以是编辑窗,也可以是消息窗(或监视窗),两者切换靠 F6 键。

—6—3—8 Screen Size [or Screen lines]

选择集成环境屏幕显示大小。

—6—3—8—1 25 line display

标准 PC 显示：25 行 80 列，此选择适宜象 MDA (Monochrome Display Adapter 单色显示器) 或 CGA (Color Graphics Adapter 彩色图形适配器)。

—6—3—8—2 43/50 line display

若 PC 配有 EGA 或 VGA 显示器，除 25 行标准显示外，还可选此值便可将文本转为 43 行 80 列 (对 EGA) 或 50 行 80 列 (对 VGA)。

—6—4 Directories

设置包含文件 (象 *.H)、库文件 (*.LIB)、输出文件 (象 *.EXE, *.OBJ)、系统配置文件目录和选择【检选文件】(Pick 文件) 名等。

—6—4—1 Include directories C:\TC\INCLUDE

说明标准【包含文件】(它一般出现在源程序里，形式为

#define □ <包含文件名>;

或

#define □ "包含文件名";

语句中，典型的像 *.H 这样一类的标头文件，也可以是其它 *.C 源文件等) 所在的目录。一行内可输入多个包含文件目录，多个目录间用分号 (;) 分隔。一行最多只能包含 127 个字符。例如

C:\TC\INCLUDE; C:\TC\TC1

注意：所指定的目录里总隐含有当前目录，且优先在当前目录里寻找。

—6—4—2 Library directories C:\TC\LIB

说明编译连接时要用到的 (称为【触发的】) 目标文件 (*.OBJ) 和库文件 (*.LIB 文件) 所在目录。一行上可以说明多个库目录 (但所有字符数不能超出 127 个)，多个目录间用分号 (;) 分隔；分号前后的空格是允许的，但非必要。相对和绝对路径均可。

注意：所指定的目录里总隐含有当前目录，且优先在当前目录里寻找。

—6—4—3 Output directory

编译时生成的 *.EXE、*.OBJ 和 *.MAP 文件均放在这个输出目录里。若为空 (即未给出具体目录名) 则指当前目录。组装机在工作时将在这个目录里寻找相应的 .OBJ 和 .EXE 文件。注意：编译连接时生成的文件将同名文件无条件覆盖，也即在改写前不给出任何提示。

—6—4—4 Turbo C directory C:\TC

该目录内一般有缺省配置文件 (TCCONFIG.TC)、缺省检选文件 (TCPICK.TCP) 和帮助文件 (TCHELP.TCH)。配置文件也可在当前目录上，TC 优先在当前目录上查找。在编辑时按 F1 键时 TC 便要在此目录上寻找 TCHHELP.TCH 文件，如果在该目录上没有这个文件便显示

Can't find file TCHHELP.TCH. Press ESC.

—6—4—5 Pick file name;

输入检选文件名 (参见—6—4)。

—6—4—6 Current pick file;

它显示当前配置的检选文件名。即显示缺省检选文件名或你在 Pick file name: 后输入的

检选文件名。显示内容不能为程序员直接更改。如果你更改了检选文件名或退出集成环境时,当前的检选表的内容被装入此文件中。

—6—5 Arguments

设置传递给执行程序的命令行参数,即设置被执行文件所用到的参数。例如,在 DOS 状态下有

```
C>TCINST /B
```

这 /B 就是命令行参数。选该项后显示

```
KHT5"SS ] Command Line Parameters [或 Args]
```

时输入参数值,文件名可省去。注意,不能输入重定向符号(<, >, <<, >>)和管道等。然而在集成环境下要运行或调试一个需要通过人机对话才能输入的命令行参数,例如,对

```
main(int argc, char *argv[]){.....}
```

要输入的命令行参数 argv[],这时就不能像在 DOS 提示符下直接键入这些参数。这时你可以选用本命令项。当提示你输入命令行参数时,你可输入相应的命令行参数(如 argv[]),但正在调试的文件名本身则不要输入。这样当你调试程序时,这些命令行参数就起作用了。值得指出的是,这些命令行参数只在集成环境里起作用,一旦离开集成环境它们就不存在了。因此,即使在集成环境里你已用本命令设置这些命令行参数而生成某执行文件 *.EXE,但是在 DOS 提示符下执行该程序时还是要重新输入相应的命令行参数的。

—6—6 Save options

将当前 TC 的配置环境(与编译器、连接器、环境、调试及规划等选择)保存到一个配置文件中。选择后显示

```
Config File [TCCONFIG.TC]
```

缺省的配置文件名是 TCCONFIG.TC。TC.EXE 启动时如果你没有指定配置文件,则 TC 先当前目录里寻找 TCCONFIG.TC,如未找到,再到 Turbo C 目录中去寻找,再找不到,便用 TC.EXE 的缺省参数(由 TCINST.EXE 设定)。

—6—7 Retrieve options

选择后显示

```
Config File [*TC]
```

输入以前用

```
Options/Save options
```

命令保存的配置文件名后,该配置文件内的参数便被装入集成环境。

—7 Debug

设置集成调试器缺省参数。跟踪只能跟踪编辑窗口内的文件,这就是说,被跟踪源文件将被装入编辑窗。

在调试程序时注意下列问题也许是有益的:

1. 没有追踪到程序结束而中途退出集成环境,则有可能只生成执行文件(*.EXE)而无目标文件(*.OBJ);
2. 用热键 F7 或 F8 单步调试时,按一次键便追踪一行,因此语句

```
for(i=0;i<100;i++)printf("i=%d",i);
```

和

```
for(i=0;i<100;i++)
```

```
printf("i=%d", i);
```

在调试时产生的操作效果是不一样的。前者按键一次便跳过了，后者则要按键 100 次！当然，在要观察变量 i 的值变化时后者是需要的。

3. 当一个程序未追踪完而中途退出，然后紧接着去追踪另一个程序时，应将追踪环境先用

Run/Program reset Ctrl-F2

命令中止当前程序调试，并消去监视窗内的调试表达式（如果涉及两个程序中的表达式相同时，可以保留）和断点，以免发生意外。另外，这两个程序最好不要用相同的名字。因为跟踪一直记牢原文件名，跟踪时会将一个源文件调入编辑窗内，而不管编辑窗内原来装有什么文件。

4. 单步跟踪时如发现错误便不能进行下去，跟踪自动在出问题行上终止；但如发现警告则可往下进行，而此时消息窗并未激活，所以警告消息可能看不到。注意，有些警告消息是不能忽视的。

—7—1 Evaluate Ctrl-F4

当调试程序时使用该命令能让你看到任一变量或其它数据项的值，若可能的话还允许你修改单个数据项的值，接着的调试便以此输入值作为该数据项的当前值参加调试。

该命令弹出三个小窗：

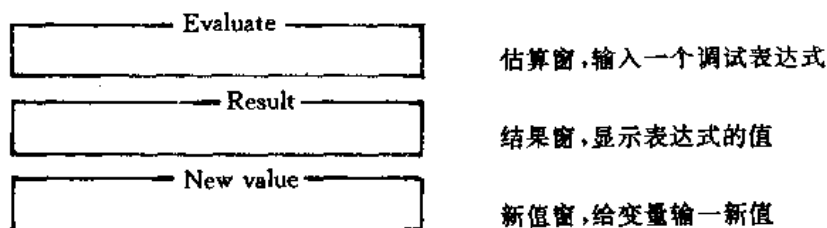


图 16-22

对估算窗你可以直接输入一个且只能是一个合法的【调试表达式】，注意，回车后键入表达式才被确认；如按 ESC 键则取消刚刚的输入。缺省的调试表达式由编辑光标所处的单词组成，这意味着你不必重新输入这个单词，而只须将编辑光标移到这个单词上再按 Ctrl-F4 键便可以让它进入估算窗。如果需要，按光标向右移动键（→）还可以把该单词后面的字符逐个装入估算窗内。显然这对于要输入一个标识符很长的单词或正文时既节省时间又避免了错误。直接按回车键可以计算缺省表达式的值；当然也可以直接输入一个新表达式，其值如果存在，则会立即在结果窗内显示；否则给出相应的出错提示信息。

一行上表达式的内容最多为 127 个字符，超过的字符被忽略。调入缺省表达式后按 End 键（或按 Home、Ins、Del、PgUp、PgDn、←等键也一样）使光条变暗后便可在当前表达式后面直接加上字符，或用 Backspace 退格键删除前面字符（用 Del 键删除光标所在字符）。要迅速移至表达式头部按 Home 键，移至尾部按 End 键。用 Tab 键、↑键或↓键可使光标从一个窗口进入另一个窗口。按 Ctrl-Y 键可立即删除整个表达式。

当估算窗内输入表达式时，结果窗内必有反映，但新值窗内无数据出现。对新值窗也可像对估算窗那样（操作方法相同）输入一个新值（可以为常数、已定义变量值或一个表达式），它用于改变单数据元素，如

```
i          /* 单个变量          */
array[i]   /* 单个数组元素       */
*(ptr+i)   /* 指针所指某内存单元 */
```

结果窗同样会作出反映,或者和新值窗有同样的显示,或者指出估算变量的当前值(新值)。有问题时给出相应的提示,如 Cannot be modified 及某标识符没有定义等。有趣的是,例如在调试程序时当变量 $x=1$, $y=2$ 时,

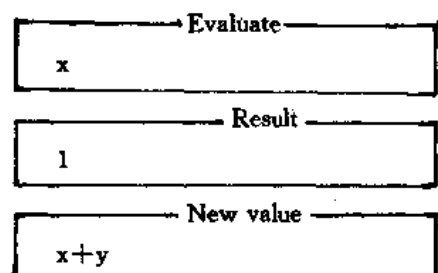


图 16-23

在新值窗输入了 $x+y$ 后,不断按回车键,便发现结果窗内数据不断变化:1,3,5,...。

对结果窗,当然不允许你输入任何值或表达式等。

一旦进入调试或压了 ESC 键,这三个窗自动消失。不过通过新值窗对某变量赋的值将作为该变量当前值保留而参加调试。

一 合法的调试表达式

表达式是指任何合法的 C 表达式。

1. 常量表达式

'd', x 可获得字母 d 的 16 进制的 ASCII 码值;

(unsigned)0xFF88 得值 65416;

(unsigned char)0xFF88, x 得值 0x88;

2. 求字节数

sizeof(char) 得 1;

3. 求内存内容

(char far *)0xF0000000, p 得 F000;0000 BIOS ROM'S

4. 可为

子表达式 op 子表达式

那样的表达式,这里 op 为运算符:!, ~, +, -, *, /, %, &, |, ^, &&, ||, <, <=, ==, >, >=, !=, <<, >>, ?: 等。表达式按既定规则运算。

当表达式中无变量时,观察表达式的值不必进入程序调试也可进行。

5. 表达式中变量可以是各种类型,包括结构、联合和伪变量等。所以这种表达式和普通数学上的表达式的含义是有差别的。因此,表达式的值也可能是多目标的,即不一定只有一个值,如结构就是这样。

6. 对同一个表达式允许带不同的显示格式符,以获得不同的显示结果。

7. 表达式中不应有函数调用,如

sin(x), f5

是不允许的。

8. 表达式中不应有定义类型的符号,如

typedef y

等也是不允许的。

9. 表达式中不应有宏名,如已定义了宏:

```
#define MYNUM 100000
```

则

```
MYNUM+1
```

这样的表达式是不允许的。

10. 表达式中不应有非当前正在执行的函数局部变量、静态变量等,需要查看它们(或不同模块、不同函数中的变量)可用限定变量的方法:

模块名.函数名.变量名

小数点符号用于它们之间分隔。对当前模块或当前函数可以只用变量名,而不必加以模块名或函数名的限定。

例如当主函数 `main(int argc)` 调用一函数 `func1(int i)` 后,当前开始调试的函数是 `func1`,如要观察主函数里的变量 `argc`,应用

```
main. argc
```

这样的表达式。

注意,调用函数时要用堆栈,调用结束该函数相关值将不存在,因此有时不能对它的变量进行观察。这在函数嵌套调用时是要注意的。

二 格式说明符

调试表达式后面可以跟一个逗号分隔符和【显示格式说明符】,或简称格式符。即

调试表达式,格式符

例如,对于数组 `xarray`,有

<code>xarray[0],5</code>	表示以十进制显示 5 个连续的整数
<code>xarray[0],5x</code>	表示以十六进制显示 5 个连续的整数

这逗号后面的数字及 `x` 就是格式符。

1. 格式说明符的种类

你可以在调试表达式后面指定下述几种格式符,表示格式符的字母可以大写也可以小写。

—1 num

`num` 是一个具体的数字,表示显示一系列连续数据元素值的个数。`num` 又称重复次数,它不但特别适用于在一行上显示不下的大数组,而且对其它变量也适用。例如,对变量 `var` 使用表达式

```
var,n
```

它表示从 `var` 变量地址开始连续显示 `n` 个相同类型的变量。注意,若一个表达式并不相当于一个变量,则 `n` 不起作用。判别一个表达式是否相当于一个变量的方法是,看它是否出现在赋值语句的左边,或者能否作为函数的参数。对非字符数组或指针使用时应小心。

数字后面可以有其它格式符,如 `CSDHXM` 等,但它和这些格式符间不能有空格。

—2 C

显示字符。在屏幕上能显示特殊控制字符(ASCII 码 0 ~ 31)。只影响和字符或字符串

相关的变量。

—3 S

显示字符,只影响与字符或字符串有关的变量。但对特殊控制字符的显示形式和 C 格式符不同,它以如下形式显示:

0x0	'\x0'	0xb	'\v'	0x16	'\x16'
0x1	'\x1'	0xc	'\f'	0x17	'\x17'
0x2	'\x2'	0xd	'\r'	0x18	'\x18'
0x3	'\x3'	0xe	'\xE'	0x19	'\x19'
0x4	'\x4'	0xf	'\xF'	0x1a	'\x1A'
0x5	'\x5'	0x10	'\x10'	0x1b	'\x1B'
0x6	'\x6'	0x11	'\x11'	0x1c	'\x1C'
0x7	'\a'	0x12	'\x12'	0x1d	'\x1D'
0x8	'\b'	0x13	'\x13'	0x1e	'\x1E'
0x9	'\t'	0x14	'\x14'	0x1f	'\x1F'
0xa	'\n'	0x15	'\x15'		

由于 S 是缺省的字符及字符串显示格式符, S 格式符只有和格式符 M 联用时才明显起作用。

—4 D

整数值按十进制显示。只适用于简单整数表达式和包含整数的数组、结构等。

—5 H 或 X

所有整数值按十六进制值显示(显示时数前面有 0x 显示)。只适用于简单整数表达式和包含整数的数组、结构等。

—6 Fn

F 表示显示浮点数, n 表示说明有效数字的个数(=n-1),范围为 2~18。例如,对浮点数

```
y=3.1415
```

表达式

```
y,f,y,f0,y,f1 或 y,f18
```

都是允许的, n 超过 18 时当 0 处理。注意:违反规定写法时,如 y,18f 那样的表达式可能会产生意想不到的效果,甚至死机。

—7 M

显示由表达式指定的地址开始的内存存储单元内的值。表达式必须是在赋值语句左边也有效的形式,也即指示地址的标识,否则 M 说明符被忽略。缺省时每一字节以两位十六进制字节显示。M 加 D 格式符,即 MD 表示字节以十进制显示;MX 或 MH 则以十六进制显示;后加 C 或 S 格式符使变量以字符串方式显示(包括特殊字符)。缺省时,显示多少字符由变量大小而定,当然也可以用 num 格式符来说明连续显示的字节数。

—8 P

按段:偏移量(seg:ofs)方式显示远指针,同时还显示指向地址的有关信息。它还能告诉段的内存定位范围,以及那个偏移地址的变量名。像

```
main,p;CS:02F1 main /* 在微存储模式中调试时 */  
main,p;6DC1(CS):0006 main /* 在大存储模式中调试时 */  
&q->name,p;DS:0194 &a
```

```
&a->address,p,DS:0196(char*)>&a+2
```

都是应用 P 格式符后获得的值的例子。

```
C>TYPE MENU8.C
#include "stdio.h"
main() /* 这是一个通过远指针和新值窗输入值来观察内存情况的程序 */
{
float i=4;
char far * ptr=0xf0000000;
printf("i=%d\n",i); /* 用 F7 键操作调试到此句后看结果 */
printf("当前视频模式是%x\n",ptr[0x449]);
}
```

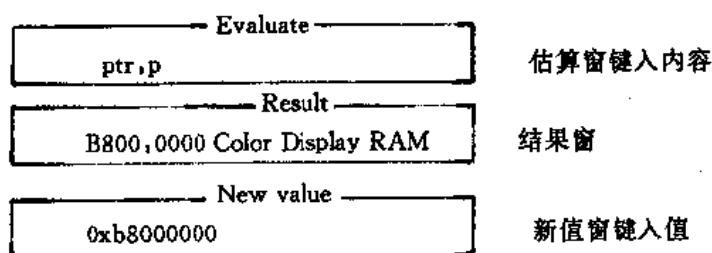


图 16-24

不断改变新值窗的值（注意：不能键入像 B800:0000；当段值不输而只输偏移量时，该偏移量表示相对于当前段的偏移量。偏移量常以 16 进制数键入），可以从结果窗中的显示搞清内存情况。例如有（对不同机器可能有不同结果）：

```
0000:0000      NULL
0000:0001 到 0000:03FF  Interrupt vector table (中断向量表)
0000:0400 到 0000:04FF  BIOS Data area (BIOS 数据区)
0000:0500 到 0000:EC6F  MSDOS/TSR's
0000:EC70 到 6000:DB1F  Turbo C
6000:DB20 到 6000:DC1F  Process PSP
6000:DC20
9000:0000
A000:0000 到 AFFF:FFFF  EGA Vido RAM
B000:0000 到 B7FF:FFFF  Monochrome Display RAM
C000:0000 到 EFFF:FFFF  EMS Page/Adaptor BIOS ROM's
F000:0000 到 FFFF:FFFF  BIOS ROM's
```

程序员可以用各种格式符的组合去试试，看看有什么结果。诸如，

估算窗输入 `ptr,p` 有 `FFFF,00FE`，而输入 `ptr,m` 有 `FE 00 FF EF`
 还可将语句

```
char far * ptr=0xf0000000;
改成
char far * ptr= ("asdf");
```

再调试，看看结果又有什么不同。

—9 R

显示结构的成员名与值。例如，对 C>TYPE MENU9.C

```

#include <conio.h>
struct text—info initial—info;
main()
{
    gettextinfo(&initial—info);
    /* 用 F7 键操作调试到下面这行上,然后将编辑光标移到 initial—info 上 */
    printf("winleft=%d\n wintop=%d\n",initial—info. winleft,
        initial—info. wintop);
    /* 如移到函数名上,即表达式用函数名,结果窗只显示函数名 */
    printf("winright=%d\n winbottom=%d\n",initial—info. winright,
        initial—info. winbottom);
    printf("attribute=%d \n",initial—info. attribute);
    printf("normattr=%d \n",initial—info. normattr);
    printf("currmode=%d \n",initial—info. currmode);
    printf("screenheight=%d \n",initial—info. screenheight);
    printf("screenwidth=%d\n",initial—info. screenwidth);
    printf("curx=%d\n cury=%d\n",initial—info. curx,initial—info. cury);
    textmode(initial—info. currmode);
    getch();
}

```

如在表达式中加上

,r

则有如下显示:

```

{winleft: '\x1', wintop: '\x1', winright: 'P', winbottom: '\x19',
attribute: '\a', normattr: '\a', currmode: '\x3', screenheight: '\x19',
screenwidth: 'P', curx: '\x1', cury: '\x19'}

```

容易将它与程序执行结果比较。这里字母 P 的 ASCII 码为 80,故表示 80 列; \a 是 TC 转义符,相当数字 7。

不用格式符 r,则只有数值而无成员名。对结构也可以加 c 或 d 格式显示符。

当调试表达式为伪变量,可以观察寄存器的情况。例如,

—DH*256+DL,d:20041

2. 格式符的使用规则

—1 未写格式符时用缺省的格式符,这由数据类型决定。

数据类型	优先级顺序 (左边的优先级最高)	缺省格式符
char	CSHD	S
unsigned char	CSHD	S
int	HDCS	D
unsigned int	HDCS	D
long	HDCS	D
unsigned long	HDCS	D
char 指针	CSPH	S
其它指针	PH	P
enum	HDCS	D (后缀成员名)

float	Fn	F7
double	Fn	F15
long double	Fn	F18
char 数组	CSDH	S
其它数组	用逗号分隔括在花括号中元素	
结构	R	R
联合	R	R

对于简单变量或数组元素,由于它们的数据类型是确定的、对多个值也是一样的,所以只要在变量名后加一个格式符就可得到一致的显示格式;而对结构或联合,它的成员的数据类型也许不全一样,因此让表达式带多个格式符。于是对同一种数据就存在按哪一个格式符显示的问题,这就是显示优先级。

—2 格式符只有附于适当的变量之后才能起作用,否则被忽略。例如:对整数 y 用 y,R 则 R 格式符无效。

—3 若表达式求值的结果会导至多个目标的显示,而表达式后又有一个以上的格式符,则 Turbo C 会自动选择合适的格式符显示。例如,对

```
struct {int k;
      double m;
}km={100,3.1415925};
```

则表达式 km,F5H 会使整数 k 按 16 进制显示,实数 m 会以 4 位有效数字显示:

```
{ 0x64, 3.1416 }
```

当然,必要时你可以用像 km.k 或 km.m 那样的表达式显示结构的成员值。

—7—2 Call stack Ctrl—F3

选此命令后显示一个包含调用栈的小窗,其内为程序运行时正在执行的函数的调用函数序列。因为 C 实际上是由一系列函数构成的,所以任一函数至少为 main() 函数调用。而 main() 函数在栈底,正在运行的函数在栈顶。例如,下面看看如何在调试 MENU10.C 时查看堆栈。

```
C>TYPE MENU10.C
#include "stdio.h"
main()
{
  int i=4;
  printf("i=%d\n",i);
  func1(5);
}

func1(int j)
{
  printf("j=%d\n",j);
  func2(8);
}

func2(int k)
{
```

```
printf("k=%d\n",k);
}
```

依次按下列步骤调试它：

第一步，将编辑光标移到程序头部；

第二步，按 F7 键；

第三步，按 Ctrl-F3 键；

第四步，转向第二步。

如此直至调试结束。依次在小窗内出现（序号数为循环数）：

```
1.  main()
2.  main()
3.  main()
4.  main()
5.  func1(5)      ←栈顶
   main()        ←栈底
6.  func1(5)
   main()
7.  func1(5)
   main()
8.  func2(8)
   func1(5)
   main()
9.  func2(8)
   func1(5)
   main()
10. func2(8)
   func1(8)
   main()
11. func1(5)
   main()
12. main()
```

从这里可以清楚地看到函数层层调用的细节：被调用函数名和参数！利用这种函数调用链，你可以发现有些函数不能返回父函数的位置，这对调试疑难程序是很有用的。而且，靠用光标移动键移动小窗内的亮条，则回车后，编辑光标便迅速定位在栈上次下层的那一个函数上。当然，这不会影响程序继续从已调试到的位置上继续往下调试，而不是从编辑光标所在位置处开始往下调试。自然，当把亮条移到栈顶后回车，编辑光标将定位于正在运行函数的当前行上，即你在查堆栈前的执行位置上。

需要指出的是，当不在调试程序方式（用 F7 或 F8 操作）时，该项缺省显示低亮度，也即不容许你查询堆栈的。其次，有些函数（不使用局部变量的列表函数）可能被忽略，如果你在程序编译前将 Options/Compiler/Code generation/Standard stack frame 置成了 Off。最后，小窗内最多只能显示 12 个函数名，超过时只显示从栈顶开始往下的 12 个函数。

为进一步说明堆栈，现把 MENU10.C 改成 STACK.C： C>TYPE STACK.C

```
/* 1 */ #include "stdio.h"
/* 2 */ main()
/* 3 */ {
```

```

/* 4 */ int a=2,b=3,t;
/* 5 */ t=func1(a,b);
/* 6 */ printf("%d",t);
/* 7 */ }
/* 8 */
/* 9 */ func1(int j,int k)
/* 10 */ {
/* 11 */ int c=4;
/* 12 */ return (func2(c,j,k));
/* 13 */ }
/* 14 */
/* 15 */ func2(int w,int x,int y)
/* 16 */ {
/* 17 */ int d=5,e=6,g=7;
/* 18 */ return (w+x+y+d*e*g);
/* 19 */ }

```

并用 TCC.EXE 编译成 STACK.ASM (对汇编的有关说明参见《函数》一章),

```

C>TCC -S STACK.C
C>TYPE STACK.ASM
        ifndef ?? version
? debug macro
        endm
        endif
        ? debug S *n56.c"
--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 E9EA596E1C056E35362E63
? debug C E900101D1115433A5C54435C494E434C5544455C737464696F2E68
? debug C E900101D1116433A5C54435C494E434C5544455C7374646172672E+
? debug C 68
--BSS ends
--TEXT segment byte public 'CODE'
, ? debug L 2
--main proc near
        push bp
        mov bp,sp
        sub sp,2

```

```

        push    si
        push    di
;        ? debug L 4
        mov     si,2
;        ? debug L 4
        mov     di,3
;        ? debug L 5
        push    di
        push    si
        call    near ptr —func1
        pop     cx
        pop     cx
        mov     word ptr [bp-2],ax
;        ? debug L 6
        push    word ptr [bp-2]
        mov     ax,offset DGROUP:s@
        push    ax
        call    near ptr —printf
        pop     cx
        pop     cx
@1:
;        ? debug L 7
        pop     di
        pop     si
        mov     sp,bp
        pop     bp
        ret
—main    endp
;        ? debug L 9
—func1   proc    near
        push    bp
        mov     bp,sp
        push    si
;        ? debug L 11
        mov     si,4
;        ? debug L 12
        push    word ptr [bp+6]
        push    word ptr [bp+4]
        push    si
        call    near ptr —func2
        add     sp,6
        jmp     short @2
@2:
;        ? debug L 13
        pop     si
        pop     bp

```



```

        ret
—func1 endp
;      ? debug L 15
—func2 proc near
        push    bp
        mov     bp,sp
        sub     sp,2
        push    si
        push    di
;      ? debug L 17
        mov     si,5
;      ? debug L 17
        mov     di,6
;      ? debug L 17
        mov     word ptr [bp-2],7
;      ? debug L 18
        mov     ax,si
        mul     di
        mul     word ptr [bp-2]
        mov     dx,word ptr [bp+4]
        add     dx,word ptr [bp+6]
        add     dx,word ptr [bp+8]
        add     ax,dx
        jmp     short @3
@3:
;      ? debug L 19
        pop     di
        pop     si
        mov     sp,bp
        pop     bp
        ret
—func2 endp
—TEXT ends
        ? debug C E9
—DATA segment word public 'DATA'
s@      label byte
        db      37
        db      100
        db      0
—DATA ends
—TEXT segment byte public 'CODE'
        extrn   —printf,near
—TEXT ends
        public —main
        public —func2
        public —func1

```

end

然后用 F7 键单步调试,在 Small 存储模式下,使用调试表达式可得下述结果:

```
main,p: CS:01FA main
func1,p: CS:0224 func1
func2,p: CS:023D func2
—SS,X: 0x777F
```

并用 Ctrl-F4 键操作,输入类似

```
*(int *)0x777fFFE2,X
```

那样的表达式求得存储单元 SS:0XFFE2 中的值(注意:不能输 *(int *)0x777fFFFF,X 表达式,那样会出现死机。除此之外的值一般可输)。

表 16-1 当调试亮条落在 STACK.C 相应语句上时调试表达式和存储单元中的值

语句	2	4	5	9	11	12	15	17	18	13	6
—BP,X	FFEC	FFE0	同左	同左	FFD2	同左	同左	FFC6	同左	FFD2	FFE0
—SP,X	FFE2	FFDA	同左	FFD4	FFD0	同左	FFC8	FFC0	同左	FFD0	FFDA
—SI,X	5C	同左	2	同左	同左	4	同左	同左	5	4	2
—DI,X	4CB	同左	3	同左	同左	3	同左	同左	6	3	3
FFE2			11D	同左	同左	同左	同左	同左	同左	同左	同左
FFE0			FFEC	同左	同左	同左	同左	同左	同左	同左	同左
FFDE			7643	同左	同左	同左	同左	同左	同左	同左	同左
FFDC			5C	同左	同左	同左	同左	同左	同左	同左	同左
FFDA			4CB	同左	同左	同左	同左	同左	同左	同左	同左
FFD8				3	同左	同左	同左	同左	同左	同左	
FFD6				2	同左	同左	同左	同左	同左	同左	
FFD4				20D	同左	同左	同左	同左	同左	同左	
FFD2					FFE0	同左	同左	同左	同左	同左	
FFD0					2	同左	同左	同左	同左	同左	
FFCE							3	同左	同左		
FFCC							2	同左	同左		
FFCA							4	同左	同左		
FFC8							235	同左	同左		
FFC6								FFD2	同左		
FFC4								7	同左		
FFC2								4	同左		
FFC0								3	同左		

现在你将 STACK.C、STACK.ASM 和表 16-1 三者联系起来,就很容易了解 BP 的作用(在子程序入口处压入栈内,在子程序出口处又被弹出)和 SP 的变化情况了。另外两个寄存器 DI 和 SI 常被用作寄存器变量,为使用它们,通常在子程序入口处先将它们原先的值入栈,在出口处再恢复它们。注意:如命令行用了 —r— 选项,即不使用寄存器变量,则无需考虑保存它们。

—7—3 Find function

象 Call stack 命令一样,它只能在调试程序时使用。其对应出现的小窗是

Enter subprogram symbol

图 16-25

输入一个你要了解的函数定义情况。例如,输入了函数名 Print(只输函数名字),则 TC 马上使编辑光标定位于这个函数的定义语句上。输入的函数名应在编辑窗内存在的,或者源码在将选项

Debug Source debugging

和

Options/Compiler/Code generation/OBJ debug information

都置成 On 时编译过,且函数所在的源文件应在磁盘上存在,那么,该源文件将被装入编辑窗,编辑光标定位于该函数定义上。否则会发出

Symbol not found. Press ESC

或

Linker Error: Undefined symbol 'function name' in module xxxx.c

这样的提示,表示没找到。

注意:在编辑窗内也可以用 Ctrl-Q-A 或 Ctrl-Q-F 键操作来寻找指定字符串,当然也可以用它来寻找函数名,但这函数名必须在编辑窗内。

—7—4 Refresh display

刷新显示器命令。万一编辑屏幕作为用户屏幕输出而被重写,可用本命令恢复当前屏幕原有内容。

—7—5 Display swapping Smart

显示转换菜单,有三种方式:

—7—5—1 None

使调试器根本就不进行切换。它用在那些确实不需要含屏输出的代码调试中。

说明:当你采用了两个显示器即双监视器输出时(进入集成环境前使用了 /d 开关),则用户程序在一个屏上,而 TC 的屏在另一个显示器上,此时 TC 不会进行切换,也即是说,Debug/Screen swapping 的选择不起任何作用。

—7—5—2 Smart

(灵活而整洁调试,缺省值),当在调试模式下运行程序用此模式时,调试器就看正在执行的代码中是否产生屏幕输出。若产生(或它调用一函数),则屏幕就从【编辑屏】切换到【用户屏】,其时间足够完成输出。然后又切换回去;否则,不进行切换。

注意:缺省 Smart 在下列情况下并不怎么“灵活”:

一是每次函数调用都要产生切换,即使函数并不产生屏幕输出;其次,在有些情况下,例如用时钟中断程序写屏时,编辑屏可能被修改了但没有切换。例如时钟中断程序写屏。

—7—5—3 Always

设置使得执行每条语句都切换,当编辑屏有可能被运行程序重写时,都应采用此选择。

—7—6 Source debugging On

设置源代码调试方式。有三种方式,它们应在程序编译和连接之前选择。TC 集成调试器用到的源文件一定要在

Options/Compiler/Code generation/OBJ debug information On

状态下编译。

注意:当跟踪信息装入模块后,则调试要小心。例如,你先将程序 C:MY1.C 装入编辑器后换名为 E:MY.C 调试,调试内容若已存在于 E:MY.OBJ、E:MY.EXE 中;现在你又使一个

C:\MY2.C 进入编辑窗,又换名为 E:\MY.C 调试。在调试时,你发现你当前调试的是 C:\MY1.C 的内容,而不是 MY2.C 的内容。TC 这种自动切换会使人恼火,但明白了这个道理,换成不同的文件名就是了。

—7—6—1 On

此方式下,此时跟踪信息被放入 .EXE 文件,供【TC 集成调试器】使用。连接程序可用 TC 集成调试器或【单独的 TC 调试器】(指 Turbo debug)调试;

它可能会在下次你执行程序时影响 Run/Rnu 命令的工作。

—7—6—2 Standalone

该方式也使跟踪信息放入 .EXE 文件,但不能供 TC 集成调试器使用,而只能用单独 TC 调试器调试了。不过程序仍可在 TC 中运行。

—7—6—3 None

对该方式,两种调试器均不能跟踪调试,因为 *.EXE 文件中没有放入调试信息。这时如要用 F7 或 F8 进行调试,程序编译后显示

No debug info. Run anyway? (Y/N)

或

No debug info at program start. Run anyway(Y/N)

选输 Y 则执行程序到底,否则不执行程序。

—8 Break/watch Alt-B

它包括控制断点和对【监视表达式】操作。

【监视表达式】是在程序调试时在监视窗 (watch) 内输入的表达式,用于监视变量等情况。由于它常在程序调试时使用,所以也称【调试表达式】。和估算窗内输入的表达式不同,不管是否进行程序调试,表达式一经输入便一直在监视窗中存在,而估算窗内的表达式只要一开始调试便会消失。

监视窗口内可以容纳多个调试表达式,每当程序调试执行中暂停时,所有的表达式和变量的值将被重算,计算结果在它们的右边(一个冒号后)列出。由此可见,这些表达式的值大都是随程序执行而改变的(只有常量表达式的值不变)。程序员不能够像在新值窗内对调试表达式预置一个值,随后在程序调试中以此值开始往后执行。监视窗口内的表达式是不能预置值的。当监视窗口被激活时,当前表达式用高亮度来标记;当窗口不活动时,当前表达式最左边有一个圆点(即小数点)标记。例如,对下列程序在调试前输入了一个调试表达式

i+1

使用 F7 键连续追踪,得如下结果(内中序号为顺序按 F7 键的次数)

C>TYPE MENU11.C

```
#include "stdio.h"
main()                /* 1.i+1;Undefined symbol 'i' */
{
    int i=4;           /* 2.i+1;31 */
    printf("i=%d\n",i); /* 3.i+1;5 */
    func1(5);          /* 4.i+1;5 */
}                      /* 12.i+1;5 */
func1(int j)          /* 5.i+1;Undefined symbol 'i' */
{
```

```

printf("j=%d\n",j);      /* 6.i+1;Undefined symbol 'i' */
func2(8);                /* 7.i+1;Undefined symbol 'i' */
}                          /* 11.i+1;Undefined symbol 'i' */
func2(int k)              /* 8.i+1;Undefined symbol 'i' */
{
printf("k=%d\n",k);      /* 9.i+1;Undefined symbol 'i' */
}                          /* 10.i+1;Undefined symbol 'i' */

```

向监视窗口内增加表达式时,窗口会变长,一直增加到由 TCINST. EXE 程序中 Resize Windows 说明的值(给 TC. EXE 设定的缺省值,初始化时是半个屏幕)。若继续增加,有些表达式会滚出窗口,但可用 PgUp、PgDn、Up、Down 等键滚动监视窗口来再现滚出的表达式。可用 F6 键激活监视窗口,再用 F5 键放大窗口到整个屏幕,就能看到更多在窗内的表达式。

在集成环境里运行程序或调试程序前,可以在一行或多行上设置断点。【断点,breakpoint】是程序执行暂停(但不是程序终止)的地方。断点所在行有亮条出现(一般显示红色),当程序执行到该行时会暂停,此时亮条变暗;程序离开断点后,断点亮条恢复如旧。程序运行到断点时将会停下来,暂不执行断点的那一行的第一条语句,将调试控制交给用户,此时用户可从多方面来研究程序。如

- 通过调试表达式观察变量值的变化;
- 使用新值窗强行改变变量值,观察运行效果,或者对某些值评估;
- 中途编辑程序,包括在源程序上附加注释等;
- 也可用 F7 或 F8 单步调试到下一个断点,或用 Ctrl-F9 操作直接运行到下一个断点;
- 除去断点或重新设置断点等操作。

断点是在源程序某行设置的标记。当你对于一个程序的故障点位置有所估计时,使用断点调试或许比单步调试效率高得多。断点要设在执行语句上,不要设在空行、解释行、编译指令行、数据说明行或其它非执行行上。

—8—1 Add watch Ctrl-F7

向监视窗口(Watch)内增加一个调试表达。当选择此命令时显示小窗

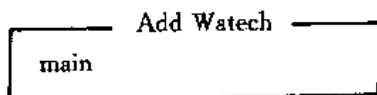


图 16-26

此时可键入一调试表达。缺省的调试表达是编辑光标所在的单词。若表达式有效,则调试器就在监视窗口内增加这个表达式及其它的当前值(显示格式按它定义的数据类型决定)。否则,有可能出现

Out of memory 仅管最多可输 254 个字符的表达式,但一行上无法列出计算结果

Expression on syntax 表达式语法错误

Unterminated string or character constant 表达式中有字符常量等

Undefined symbol 'ident' 表达式中有未定义的符号(常为未调试到而出现)

等错误提示。当然,程序员一般应根据自己需要什么就输什么的原则,大胆操作,当出现错误提示时才作必要的修改;而决不要先去记住那许多限制,限制在多次碰壁后也就容易记住了。

监视窗口被激活时可按 Ins 或 Ctrl-N 键,这时小窗出现监视窗口内亮条所在表达式的内容,按退格键或←、→键后可对其修改,按回车键后这个以亮条所在表达式为基础的又一个表达式被增加到监视窗口中。

监视窗口内允许有两个完全相同的表达式,通常它们附带的格式符不同,常用来对同一变量观察其不同的显示形式。

C>TYPE MENU12.C

```
struct {
    int num;
    char ch[10];
    }u={100,"ok!";
int list[5]={1,2,3,45};
int *ptr=list;
void main()
{
}
/* list , { 1, 2, 3, 45, 0 }
list[2] : 3
list[2],3x : 0x3, 0x2D, 0x0
list,m : 01 00 02 00 03 00 2D 00 00 00
list,2M : 01 00
list,s : ('\x1', '\x2', '\x3', '-', '\x0' )
ptr : list
ptr,r : list
ptr,p : DS:01A0 list
* ptr : 1
* ptr,3 : 1, 2, 3
u : {100,"ok!"}
u,r : { num:100, ch:"ok!" }
u,s : { 'd', "ok!" }
u,x : {0x64, {0x6F, 0x6B, 0x21, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0 } }
u.ch : "ok!" */
```

—8—2 Delete watch

当监视窗口可见时,使用本命令可删除监视窗口内亮条所在表达式。具体操作过程是,按 F6 键激活监视窗,其内亮条所在表达式前应无园点(.)标记,然后选本命令,回车 后亮条所在表达式便消去。每使用一次本命令,消去一个调试表达式。

按 Alt-B-D 后消去当前(刚刚增加的)调试表达式。

要连续消去监视窗内多个表达式,可用 F6 激活监视窗后,移动亮条到所要删除的表达式上,然后按 Del 键或 Ctrl-Y 键,每按一次删除一个,连续按连续删除。

应当注意:未被消去的表达式一直存储在调试器中。例如,你对某程序调试后又装入 另一程序调试,则原先设的表达式如未消去则还存在。

—8—3 Edit watch

修改监视窗口内亮条所在的表达式。方法是先激活监视窗口,将亮条移到要修改的表达式上,然后用本命令,调试器便将亮条所在表达式拷贝到小窗内,供程序员修改。用退格键或光标移动键可在原表达式基础上开始修改,如用新值则直接输入。按 Home 键可使小窗内的小光标立即移至表达式头部,按 End 键可马上移至其尾部。

—8—4 Remove all watches

使用本命令可一次性将监视窗口内的全部调试表达式清除。

—8—5 Toggle breakpoint Ctrl—F8

它是一个反复开关（或乒乓开关），用于设置或消去编辑光标所在行的断点。当一源程序行被设置断点后该行便显示高亮度（有亮条，其颜色由 TCINST.EXE 预置缺省）。编辑源文件时，该亮条不动，表示此断点依然存在。但在下列情况，此断点将消失：

1. 又用本命令一次时，亮条消失，断点不存在了；
2. 用 Break/watch/Clear all breakpoints 命令将所有断点消去时；
3. 删除了断点所在行，断点随之消失；
4. 用 Quit Alt—X 命令退出集成环境。

由于调试程序是在编译、连接之后进行的，所以在调试之前便可发现源程序的编译与连接错误。由此可知，如在调试中间（F7 键操作、F8 键操作，或者设置断点后用 Run/Run 操作）编辑源程序，就有可能失去对断点的跟踪（断点丢失、断点位置错误、对不正确断点不发生警告，或重新开始编译等），因为只有在经过编译与连接之后才能知道断点的正确位置。因此在调试中间编辑了一个含断点的源文件（至少有一处修改）后如有显示

Source modified, rebuild? (Y/N)

时最好键入 Y，从头开始。

在调试中间可以用 Ctrl—F8 操作插入一些新断点或消去一些老断点。如编辑光标在当前执行行上时按了 Ctrl—F8 键，而此时编辑光标所在行上一行又不是断点行，则显示

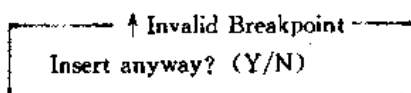


图 16—27

如键入 Y 则使编辑光标所在行上设断点。注意当编辑光标在当前执行行上时，也可用 Ctrl—F8 键操作对其设断点，不过当时断点行的颜色标记要等编辑光标移开后才会显示出来。

总而言之，在调试中间最好不要编辑源程序，可在调试前多设一些断点。调试器能在调试中如遇到一个有问题的无效断点（譬如断点所在语句不是一可执行语句）时会发出

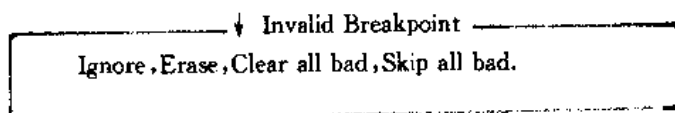


图 16—28

警告。按 I 键忽略；按 E 键删除该断点；按 C 键删除断点；按 S 键则跳过。

注意：当重新从磁盘调入一个程序调试前，一般应结束原先编辑窗内的程序调试，并消去它的所有断点。

—8—6 Clear all breakpoints

一次性清除程序中设置的所有断点。调试中间也可以清除。

—8—7 View next breakpoint

将编辑光标移到程序中下一个断点处。注意：并不是从当前断点处执行到下一个断点处。亦即是说，该命令不执行代码，而只是将编辑光标迅速定位到下一个断点上。如果你在多个文件中设置了断点，那么本命令使你容易对下一个断点所在文件进行编辑。

—9—1 Full graphics save On

自动将屏幕图形保存的开关。为了保存图形，TC 在内存开辟了 8K 内存作缓冲区，要利用这个缓冲区时可将开关置成 On（缺省值）；否则置成 Off，这在屏幕上始终只使用字符文

本模式时可那样做,此时 TC 不会开辟这 8K 内存作缓冲区。注意,由于在集成环境下没有对应的菜单,因此不能在集成环境下修改此值,而只能用 TCINST.EXE 修改。

—9—2 Options for editor

在编辑窗口内,顶端有一描述当前激发的编辑模式的状态行

Line 1 Col 1 Insert Indent Tab Fill Unindent * C:\MYFILE.C

本菜单项就是设置与编辑器状态行等有关的缺省值。其中

Line 1 表示编辑光标处在第 1 行

Col 1 表示编辑光标处在第 1 列

文件名前的星号 * 表示文件装入后已被修改,但修改后还没有存过盘。

—9—2—1 Insert mode On

插入模式开关。在 On 状态,表示可在编辑光标处插入一个字符,光标处原有字符将自动向右移动;在 Off 状态,光标处原有字符将被刚输入的字符取代。

在集成环境里可用按 INS 键或 Ctrl-V 键操作进行切换。当状态行显示 Insert 表示在插入状态,当无显示表示不在插入状态。

在插入模式下,可用回车键来结束一行。

—9—2—2 Autoindent mode On

自动缩进开关。用本命令将 TC.EXE 内部缺省值置为 On 时,在集成环境内里,编辑状态行中便有 Indent 显示,表示当前处在行自动缩进状态;否则,如无 Indent 显示,表示取消行缩进。在集成环境下如使用 TC.EXE 缺省值,设置或取消自动缩进模式可使用键操作 Ctrl-Q-I (反复键操作):即在自动缩进模式按 Ctrl-Q-I,自动缩进模式被取消,接着又按了 Ctrl-Q-I,又进入自动缩进模式)。

【行缩进】是指按动某些特定键后,编辑光标自动一次连续向右移动几个空格。具体地说,在缩进模式时

1. 按回车键后,编辑光标不是到下一行的第 1 列,而是跳到下一行的特定列上,光标所在列的列号和上一行第一个非空格字符所在的列的列号相同,也即从列的角度看,两者在同一列位置上;注意:使用光标键 ↑ 或 ↓ 可以使编辑光标一直处在同一列上,只要行上曾输入过一个字符。

2. 每按一次 TAB 键,编辑光标按 Tab size 菜单设定的字符数界限一级级缩进。如果想改变缩进位置,按空格键 (Space) 或用光标移动键 → (或 ←) 将编辑光标移到所需列后再按 TAB 键;

3. 使用 Ctrl-K-I 可将块缩进一个字符;用 Ctrl-K-U 则块回退一个字符。

使用块缩进或回退时,必须将编辑光标置于块内。使用块缩进或回退的好处是一次可以使正文若干行同时向右或向左移动一个字符。向右移时,各行前面自动补一空格符。

【行回退】的含义和行缩进的含义相反,它是在行缩进模式下,将编辑光标置于行第一个非空格符的字符位置上,按动退格键 (Backspace),则编辑光标一次向左一次移动几个空格符,直到和上一行第一个非空格字符对齐。此后如再按退格键,则按 Tab size 菜单设定的字符数的界限进行一级级回退。

—9—2—3 Use tabs On

是否使用横向制表键 (TAB 键) 开关。置 On,则在集成环境下,按一次 TAB 键编辑光标自动向右移动由 Tab size 菜单设定的字符数,并且文本正文内将插入一个制表符 (ASCII 码为 09H,它在屏幕上不可见)。例如,当前编辑光标位置在第 1 列上,Tab size 设定值为 8

,则按动一次 TAB 键,编辑光标移至第 9 列上;再按一次 TAB 键,移至第 17 列上等等。

在集成环境里,如使用 TC.EXE 缺省值,则编辑状态行显示 Tab Fill,否则无此字符串显示。要改变模式,可用 Ctrl-Q-T 反复键操作。

另外需说明的是,不管当前设置 Tabs 的值是几,存入源文件均以 ASCII 码 09H 表示。因此,假定上次你用 Tabs 的值为 2,而现在刚刚进入集成环境,由于 tabs 缺省值为 8,那末你将看到原先 Tabs 值为 2 的地方均为 8 了。这就是说, Tabs 只管当前的显示,究竟值为多少,要看当前的设置值(而不是原来存盘时的设置值)。事实上,字处理软件也有类似的情况。

注意:在用矩阵打印机打印时,09H 仍起横向制表符作用。

—9—2—4 Optimal fill On

【优化填充】开关。所谓优化填充是指在插入模式(Insert mode 为 On)、自动缩进模式(Autoindent mode 为 On)和使用横向制表模式(Use tabs 为 On)时,每个自动缩进的开始字符处均优化地填以横向制表符(09H)或空格符,从而使正文中使用的字符数最少。例如:对下面两行

A234

B5678

第一行是按了一次 TAB 键后输入字母 A,第二行是在第一行输入数字 4 后回车,紧接着输入字母 B。而实际存储的正文格式为

[09H]A234[0DH][0AH][09H]B5678[1AH]

而不是

□□□□□□□A234[0DH][0AH]□□□□□□□B5678[1AH]

如果现在把这 2 行作为一个块,则用 Ctrl-K-I 让块缩进一个字符,那么存储为

[09H]□A234[0DH][0AH][09H]□B5678[1AH]

从上可见 C 源文件有时并不跟某些版本的 WORDSTAR 字处理生成文件兼容的原因。在 WORDSTAR 里尽管也用 TAB 键进行横向缩进,但生成文件则完全填以屏幕实际显示的空格数。

在集成环境里,如使用 TC.EXE 缺省值,则编辑状态行显示 Fill,否则无此字符串显示。要改变模式,可用 Ctrl-O-F 反复键操作。

—9—2—5 Backspace unindents On

当光标在一行中第一个非空字符上(或在空行上)时,按退格键便回退一级,即按一次回退一个或多个空格。是否使用回退模式开关。在集成环境里,如使用 TC.EXE 缺省值,则编辑状态行显示 Unindent,否则无此字符串显示。要改变模式,可用 Ctrl-O-U 反复键操作。

—9—2—6 Editor buffer size 65534

设置编辑器缓冲区大小。显示

Specify new editor buffer size:[20000..65534] 65534

可选用字节数为 20000~65543。

—9—2—7 Make use of EMS memory On

是否使用扩页内存(EMS 内存)开关。TC 编辑程序有扩页内存支持,系统中含有和 DOS 3.2 以上版本一致的 EMS 驱动程序。启动时,TC 决定是否可使用 EMS 内存。如果能,自动占用扩页内存,释放 64K 内存编译和运行程序。

注意:关闭该功能只能使用 TCINST.EXE,而不能使用集成菜单。

—9—3 Args

Args 是 Arguments(参数) 的意思 (参见 —6—5)。

—9—4 Editor commands

1. 预置的编辑命令和对应的编辑键

注意: 在集成环境里, 主次键起同等作用, 对下列缺省值请优先用次键。键操作 CtrlM 即 Ctrl-M。

固有的命令内容	主键	次键(常用, 优于主键)
(1) New Line 在光标所在行上面插入一行, 光标还在原来行上。	* <CtrlM>	<CtrlM>
(2) Cursor Left 光标向左移动一字符位置。	* <CtrlS>	<Lft> (即←键)
(3) Cursor Right 光标向右移动一字符位置。	* <CtrlD>	<Rgt> (即→键)
(4) Word Left 光标向左移动一个单词位置。	* <CtrlA>	<CtrlLft>
(5) Word Right 光标向右移动一个单词位置。	* <CtrlF>	<CtrlRgt>
(6) Cursor up 光标垂直上移一行, 列号不变。	* <CtrlE>	<Up> (即↑键)
(7) Cursor Down 光标垂直下移一行, 列号不变。	* <CtrlX>	<Dn> (即↓键)
(8) Scroll Up 屏幕向上卷动一行(整屏往下移一行, 顶上第一行由该行的上面一行填充)。	* <CtrlW>	
(9) Scroll Down 屏幕向下卷动一行(整屏往上移一行, 底行由该行的下面一行填充)。	* <CtrlZ>	
(10) Page Up 显示上页(当前屏幕的顶上一页)。	* <CtrlR>	<PgUp>
(11) Page Down 显示下页(当前屏幕的底下一页)。	* <CtrlC>	<PgDn>
(12) Left of Line 光标移到行首。	* <CtrlQ><CtrlS>	<Home>
(13) Right of Line 光标移到行尾。	* <CtrlQ><CtrlD>	<End>
(14) Top of Screen 光标移到本页顶上第一行, 列号不变。	* <CtrlQ><CtrlE>	<CtrlHome>
(15) Bottom of Screen 光标移到本页底行, 列号不变。	* <CtrlQ><CtrlX>	<CtrlEnd>
(16) Top of File 光标移到正在编辑的源程序的第一行。	* <CtrlQ><CtrlR>	<CtrlPgUp>
(17) Bottom of File 光标移到正在编辑的源程序的最后一行。	* <CtrlQ><CtrlC>	<CtrlPgDn>
(18) Move to Block Begin 光标移到块首。	* <CtrlQ><CtrlB>	
(19) Move to Block End 光标移到块尾。	* <CtrlQ><CtrlK>	

- (20) Move to Previous Pos * <CtrlQ><CtrlP>

设光标原来在 XX 处,你现在用一次键操作将它移到另一处,发现不对,想让光标立即返回原先位置时可用本操作。

- (21) Move to Marker 0 * <CtrlQ>0

光标移到设定的“标志 0”处。

- (22) Move to Marker 1 * <CtrlQ>1

- (23) Move to Marker 2 * <CtrlQ>2

- (24) Move to Marker 3 * <CtrlQ>3

- (25) Toggle insert * <CtrlV> <Ins>

控制编辑状态行上插入 (Insert) 状态的【乒乓开关】。所谓乒乓开关,泛指连续按同一键,可使若干种情况周而复始地出现。例如,此处假定按一下开关,状态行上显示出字符 Insert;再按一下,字符消失;又按一下,字符又出现等等。

- (26) Insert Line * <CtrlN>

当光标在行首第一个字符位上,使用本操作后,在光标原所在行上面插入一行,光标移到新插入行行首;如光标不在第一个字符处,则以光标所在位置将原行分成两行,光标右边的字符构成新的一行(插入行),光标本身位置未变。

- (27) Delete Line * <CtrlY>

将光标所在行删除。

- (28) Delete to End of Line * <CtrlQ><CtrlY>

删去光标右边所有字符。

- (29) Delete Word * <CtrlT>

删去光标右边一个单词。

- (30) Delete Char * <CtrlG>

删去光标右边一个字符。

- (31) Delete Char Left * <CtrlBkSp> <CtrlH>

删去光标左边一个字符。

- (32) Set Block Begin * <CtrlK><CtrlB>

设置块首标志。

- (33) Set Block End * <CtrlK><CtrlK>

设置块尾标志。

- (34) Mark Word * <CtrlK><CtrlT>

将一个单词设置为一个块,块首在单词的第一个字符处,块尾在单词的最后一个字符处。注意,单词有时可能只有一个字符。

- (35) Hide Block * <CtrlK><CtrlH>

隐藏或显示块的乒乓开关。注意:只有当块被显示时才认为它能被调用(如进行块拷贝等);否则,它虽已定义但被隐藏时是不能调用的。

- (36) Set Marker 0 * <CtrlK>0

设置标志 0。设置标志的目的便于查找编辑。一经设定,便可用将光标移到标志处的操作迅速使光标移到指定标志处。

- (37) Set Marker 1 * <CtrlK>1

- (38) Set Marker 2 * <CtrlK>2

- (39) Set Marker 3 * <CtrlK>3

- (40) Copy Block * <CtrlK><CtrlC>

拷贝块到光标处。注意:块标志将移动到新位置,即原块所在处的字符已无块标志存在。

- (41) Move Block * <CtrlK><CtrlV>

移动块到光标处。注意：原块处往往留下一空行！

- (42) Delete Block * <CtrlK><CtrlY>

删除一个块，块标志也将被删除。

- (43) Read Block * <CtrlK><CtrlR>

本操作后显示。

Read Block from file

此时你应键入一个磁盘上的已存在的文本文件名，回车后该文件被读出，并且从当前光标处依次装入它的全部内容，即将它插入光标处。文件名也可带通配符。

- (44) Write Block * <CtrlK><CtrlW>

将块写入磁盘，当前块不改变。在执行时显示

Write Block from file

你要输入一个文件名。如果指定文件已经存在，编辑器在重写文件前会提示你是否要重写。输出时如果没有给出扩展名，TC 将自动加上扩展名 .C。

- (45) Print Block * <CtrlK><CtrlP>

将块在打印机上打印出来。显然，块如是整个文件或者没有指定块，则当前编辑的文本将整个打出。

- (46) Exit Editor * <CtrlK><CtrlD> * <CtrlK><CtrlQ>

退出编辑器，返回主菜单。

- (47) Tab * <CtrlI>

行(向右)缩进规定字符，功能与按 TAB 键一样。

- (48) Toggle Autoindent * <CtrlO><CtrlI> * <CtrlQ><CtrlI>

控制编辑器自动缩进标志的乒乓开关。

- (49) Toggle Tabs * <CtrlO><CtrlT> * <CtrlQ><CtrlT>

控制编辑器状态行 Tabs Fill 标志的乒乓开关，这两个标志用这开关时同时出现或消失。

- (50) Restore Line * <CtrlQ><CtrlL>

不管对行如何修改，只要光标没有离开该行，那么当你执行本操作时立即恢复原有行的全部内容。

- (51) Find String * <CtrlQ><CtrlF>

寻找指定字符串。找到后光标停在串的第一个字符处。

- (52) Find and Replace * <CtrlQ><CtrlA>

寻找指定字符串并进行代换。

- (53) Search Again * <CtrlL>

继续搜索

说明：搜索操作选项

1. B 反向搜索

2. G 全程搜索

3. L 在标识块内部搜索

4. n 搜索次数

5. U 搜索时不区分大小写字母

6. W 把搜索串当成一个英文单词对待，因此只搜索出正文中完整的单词

- (54) Insert Control Char * <CtrlP>

在当前光标处插入一个控制字符（下面用 ^ P 代替 CtrlP）；

键操作 显示 键操作 显示 键操作 显示 键操作 显示 键操作 显示

^ P-ESC [^ P-F2 @< ^ P-F3 @ = ^ P-F4 @> ^ P-F5 @?
^ P-F6 @@ ^ P-F7 @A ^ P-F8 @B ^ P-F9 @C ^ P-F10 @D
^ P-Home @F ^ P-↑ @H ^ P-PgUp @I ^ P-← @K ^ P-→ @M
^ P-End @O ^ P-↓ @P ^ P-PgDn @Q ^ P-Ins @R ^ P-Del @S

显示的第一个字符的 ASCII 码为 00H,第二个字符的 ASCII 码为相应字母的 ASCII 码。例如,字母 H 的 ASCII 码为 47H 等(实际插入的是【换码序列(ESCAPE)】)。

(55) Save file * <CtrlK><CtrlS>

将当前编辑的文件存入原来磁盘上同名文件内,当前文件不变。

(56) Match pair * <CtrlQ><Ctrl[>

向右查找配对字符,亦称【双限界符】。配对字符一般指:尖括号对 <>、圆括号对 ()、方括号对 []、花括号对 {}、注释对 /* */、西文双引号对 "" 和西文单引号 '' 对。下面以实例说明查找方法。先对下面两语句

```
return(access(filename,  
0)==0);
```

将光标移到第一行第一个左圆括号上,按 Ctrl-Q-[键,可以看到光标立即移到下行第二个即最后一个右圆括号上,表明这一圆括号和上面配对。现在还将光标移到第一行左边第一个圆括号上,并将第二行上的最右边的圆括号消去,然后按 CtrlQ[键,发现光标不动。(参见《程序结构和主函数》一章。)

(57) match pair backward * <CtrlQ><CtrlJ>

向左查找配对字符。查找方法同向右查找配对字符。

(58) Language Help * <CtrlF1>

将光标放在单词上按动此键,立即得到帮助信息,内容根据光标所在单词的内容(常量、变量、函数等)而定。例如,光标所在单词是一个关键字,则帮助解释该关键字。注意:TCHELP.TCH 文件应在 TC 文件目录里。

(59) Toggle optimal fill * <CtrlO><CtrlF>

优化填充乒乓开关。

(60) Toggle unindent * <CtrlO><CtrlU>

自动缩回乒乓开关。

(61) Block indent * <CtrlK><CtrlI>

块向右缩进一个字符,即块内所有行同时向右移一个字符。

(62) Block unindent * <CtrlK><CtrlU>

块向左回退一个字符。

2. 屏幕的底行提示的编辑驱动键说明

(1)→ ← ↑ ↓ —Select

利用这四个键将光条移到需重新定义的编辑键上(主键或次键上,但不能移到命令上)。

(2)PgUp — PgDn—Page

用这两个键将屏幕上卷或下卷一页。

(3)回车键—Modify

按回车表示确认选用已修改键作对应命令的编辑键。

(4)R—Restore factory defaults

按 R 键,则将所有命令编辑键恢复为 TC 原设定的键。

(5)ESC—exit

按 ESC 键退出修改,返回主菜单。

(6)F4—Key modes, (*)—Wordstar—like (■)—Ignore case (|)—Verbatim

F4 键是乒乓开关,在三种模式之间切换:

<1> (*)—Wordstar—like 模式,简称 WS 模式

它和 Wordstar 字处理软件中对编辑键的使用有点类同。在键前面有星号 (*) 标志的表明使用了此模式。

<2> (■)—Ignore case 模式,简称其它模式

不同于 WS 模式或逐字模式的模式,编辑键前有 ■ 标志。

<3> Verbatim 模式,简称逐字模式

选用该模式的编辑键前面有标志 (|)。编辑键一般可按键本身的含义进行,所以最常用的是使用一个键。如使用多个 (组合) 键 (最多 3 个),则操作时必须顺次按完每个键后命令才被执行。例如,对命令选择

New line ■ <CtrlS><CtrlD><CtrlF>

则在顺次按了 <CtrlS>、<CtrlD> 和 <CtrlF> 须顺次按完每个键后命令才被执行。例如,对命令选择

New line ■ <CtrlS><CtrlD><CtrlF>

则在顺次按了 <CtrlS>、<CtrlD> 和 <CtrlF> 后,才在当前行前插入一行;而对同样的键,在 WS 模式下,即设

New line * <CtrlS><CtrlD><CtrlF>

则既可像逐字模式那样使用编辑键,又可按

<CtrlS><D><F>

方式按键,即只在第一次按 <Ctrl> 键,以后便不必按 <Ctrl> 键,而直接按相应字母键便可。由此可见“逐字”之含意。

三种模式之间的异同见表 16-2。

表 16-2

模 式	W S	其 它	逐 字
首 键	必须是控制键或像 <Ins> 等特殊键		
开头输入控制键方法	按控制键或一个字母	必须先按控制键	
控制键后直接跟字符	A~Z 不允许	A~Z 允许	
	允许 ^ _ \ [] 0~9 等		
控制键最多	6 个	6 个	3 个

说明:

<1> 特殊键指 <Home>、<End>、<PgUp>、<PgDn>、<Ins>、、

<Up>(↑)、<Dn>(↓)、<Lft>(←)、<Rgt>(→)、功能键 F2~F10,控制键主要指 <Ctrl> 与 <Alt> 键。输入时完全可根据程序员要求去试设置。只有当出现

First character in command cannot be alphanumeric. Press<ESC>.

或

Letters not allowed in "WordStar-like" mode. Use control chars.

Press <ESC>.

时才表示输入了不允许的方式,宜换一种键方式(按 ESC 键后重输)。

<2> 对有些键你不能选用,例如对 F1 键,否则会显示

<F1> Can not be overridden. Press <ESC>.

按 ESC 键后重输。但对某些功能键允许选用,例如,这时会出现提示

<AltF3> is a built-in hot key, override it in the Editor? (Y/N)

因为在 TC 集成环境中 Alt-F3 是热键,因而提醒你小心使用,以免对不同功能使用同种键操作产生混淆。所以一般最好不采用!但你觉得可以,那么 TCINST 将接受它。

<3> 有些组合键实际上表示一个键,像 <CtrlE> 相当于 ESC 键,<CtrlH> 键相当于 BackSpace (退格键),<CtrlM> 相当于回车键等。

对由制键与大写字母组成的编辑键,在集成环境下使用时可不管字母的大小写。

—9—5 Mode for display

通常 TC 能正确判定显示模式,只有当需要选择不同的显示模式,或使彩色/图形适配器不出现“雪花”,或发现 TC 对硬件判断不对,或希望“复合彩色屏幕”变成黑白屏幕,或有可移植的 LCD 屏幕不知如何选择时才使用本菜单项进行选择。

—9—5—1 Default

(缺省模式),这是 TC 启动时使用的模式。

—9—5—2 Color

对该模式,TC 不管启动时使用的模式而使用 80 列彩色模式,退出时恢复原模式。

—9—5—3 Black and white

使用 80 列黑白模式,退出时恢复原模式。

—9—5—4 LCD or composite

不管启动模式,采用 80 列黑白方式,退出时恢复原模式。

在 TC 用缺省模式启动,选上述四项回车后,屏幕出现“雪花”,再回车出现小窗,内容是

```
Was There "Snow" on the Screen
Yes, the screen was "snowy"
No, always turn off snow detection
Maybe, always check the hardware
```

图 16—29

当选 Yes 则出现雪花;选 No 则关闭“雪花检查”;选 Maybe 则检测硬件。

—9—5—5 Monochrome

不管启动模式,如果检测到单色适配器,便使用单色适配器,退出时再恢复原方式。

—9—6 Set colors

设置集成环境内各种显示的颜色,以便于观察。TC 菜单窗口涉及的可变颜色项共有 15

种。

—1—	Active title	激活的标题
—2—	Border	窗口边框
—3—	Caps/Num/Scroll lock	窗口状态行右边显示的这三种键
—4—	Current source line	当前光标所在源程序的行
—5—	Disabled entry	可显示项
—6—	Error message	显示的错误信息
—7—	First letter	显示命令字符串的首字母
—8—	Highlighted text	高亮度文本
—9—	Keywords	显示的关键词
—10—	Line breakpoint	设置的断点行
—11—	Normal text	正常亮度文本
—12—	Selected keyword	供选择的关键词
—13—	Selection bar	表示一按回车键即被选中的亮条
—14—	Status line	编辑窗口内的编辑状态行
—15—	Title	窗口标题

可选颜色 16 种：

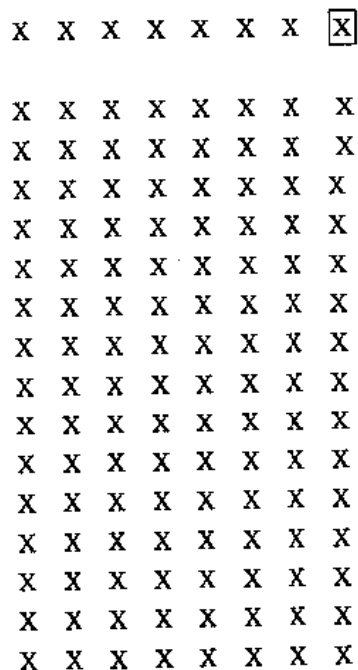


图 16—30

用→、←、↑或↓键移动小方块指示标记便可改变颜色，前景背景自动组合，用户只须观察右边窗口内颜色的变化是否满意，符合要求时按回车键便选定颜色，按 ESC 键返回前一菜单。

—9—6—1 Customize colors

可改变颜色窗口共 16 种（情况 A：～ L：）。

—9—6—1—1 A:Main menu

TC 屏幕顶端一行上显示的主菜单，涉及颜色项：

—11— Normal

—7— First letter

—13— Selection bar

—9—6—1—2 B: Pulldown Menu

从主菜单下拉出的菜单, 涉及颜色项:

- 2— Border
- 11— Normal text
- 7— First letter
- 13— Selection bar
- 5— Disabled entry

—9—6—1—3 C: Pop-up Menus

下拉菜单弹出菜单, 涉及颜色项:

- 2— Border
- 11— Normal text
- 7— First letter
- 13— Selection bar
- 5— Disabled entry

—9—6—1—4 D: Editor

编辑器窗, 涉及颜色项:

- 2— Border
- 15— Title
- 1— Active title
- 11— Normal text
- 8— Highlighted text
- 14— Status line
- 6— Error message
- 10— Line breakpoint
- 4— Current source line

—9—6—1—5 E: Message/watch

信息/监视窗, 涉及颜色项:

- 2— Border
- 15— Title
- 1— Active title
- 11— Normal text
- 13— Selection bar

—9—6—1—6 F: Compiler Status

编译状态窗, 涉及颜色项:

- 15— Title
- 2— Border
- 11— Normal text
- 8— Highlighted text

—9—6—1—7 G: Input Box

输入文件名窗, 涉及颜色项:

- 15— Title
- 2— Border

—11— Normal text
—8— Highlighted text
—9—6—1—8 H:Error Box
错误发生（如文件没找到等）窗，涉及颜色项：

—15— Title
—2— Border
—11— Normal text
—8— Highlighted text

—9—6—1—9 I:Verify Box
确认（如存储文件吗？）窗，涉及颜色项：

—15— Title
—2— Border
—11— Normal text
—8— Highlighted text

—9—6—1—10 J:Directory Box
文件目录窗，涉及颜色项：

—15— Title
—2— Border
—11— Normal text
—8— Highlighted text

—9—6—1—11 K:Help
帮助窗，涉及颜色项：

—15— Title
—2— Border
—11— Normal text
—9— Keywords
—12— Selected keyword

—9—6—1—12 L:Status Line
TC 的状态行，涉及颜色项：

—11— Normal text
—8— Highlighted text
—3— Caps/Num/Scroll lock

从上述看出，可设置的颜色是很多的，有时也许对具体的程序员来说并不需要那么复杂，因此 TC 提供了三套已设置好（各种窗口颜色搭配均是固定的，即颜色已自动搭配好，用户不能更改）的颜色集：

—9—6—2 Default color set
缺省颜色集。

—9—6—3 Turquoise color set
绿兰色颜色集。

—9—6—4 Version 1. X color set
模式 1. X 颜色集。

对这三类颜色集用户只要按 PgUp 或 PgDn 键试翻各种颜色,看看是否合适,可以的话按回车便选定;按 ESC 键便退出对颜色巡视。

—9—7 Resize windows

改变编辑窗口和输出/监视窗口的大小命令。

用 ↑ ↓ 键指定编辑窗口和输出 / 监视窗口的分屏线,改到满意时即回车确认;按 ESC 键便放弃修改,返回主菜单。在编辑窗口和输出 / 监视窗口同时存在时,例如在编译时出现这种情况,那么若你选分屏线时将输出 / 监视窗取在最大位置上,则输出 / 监视窗最多可显示 11 行,编辑窗显示 9 行;反之,在最小状态输出 / 监视窗只能显示 1 行。

注意:使用 43 行或 50 行方式运行时,采用 25 行的比例系数。

—9—8 Quit/Save

退出或存储对 TC.EXE 修改的缺省参数。

当选此菜单时,显示

Save change to TC.EXE? (Y/N)

键入 Y,则刚刚你修改 TC.EXE 缺省参数将存入 TC.EXE 文件中,退出 TCINST 返回 DOS;否则,修改参数不会存入 TC.EXE,而只是退出 TCINST,TC.EXE 的原有缺省参数不变。

不想退出(即突然想起还要修改)可按 ESC 键。

注意:由于 TC 原设置的缺省参数是经过仔细考虑的,为安全起见,在对 TC.EXE 缺省参数作修改前,你最好对它作一个备份,以便必要时可迅速恢复。

16.7 DOS 5.0 的行编辑器 EDLIN.EXE

—1 如何启动编辑器

语法:edlin [drive:][path]filename [/b]

说明:

1. 编辑器只能编辑文本文件(或称正文文件),不能编辑二进制文件。它可以编辑 C 源文件。值得注意的是,通常它在汉字系统下都能运行。相反,在有些汉字操作系统下不能运行 Turbo C 集成编辑器,或者说汉字不能进入集成编辑器。为了使汉字进入 C 源文件,可以分两步走:先在西文状态下利用集成编辑器将要标志汉字处先用足够的英文字母填充,并记牢它们的行号;然后进入汉字系统,改用行编辑器对指定行号编辑,将英文字母改成汉字即可。

2. drive: 是驱动器名,如 a:、b: 等; path 是路径名; filename 是文件基本名和扩展名。当由它们指定的文件在磁盘上存在时,编辑器将它读入内存并编辑,否则编辑后可用 E 命令按此文件名存盘(文件自动创建)。

3. 开关 /b 的使用

通常控制字符 1AH(相当于 Ctrl-Z 键操作)是当作文件结束标志来使用的。如果在编辑的正文中也要使用这个字符,就必须使用 /b 开关。

4. 若指定文件早已存在,则进入编辑后显示(表示已把文件最后一行读入内存)

End of input file

* 或者显示

New file

*

星号标记(asterisk)是等待输入编辑器命令或正文的提示符。当只显示一个星号时,表示要读

入内存的文件太大,所以只有部分文件内容被读入。

—2 行号的使用

编辑器将正文分成行,每行最多可包含 253 个字符。每一行都被分配有一个行号,行号总是连续的。注意,编辑时虽见行号,但它不是正文的一部分,或者说行号不会被写入被编辑的文件内,它只供编辑时使用。增加或删除行时,行号会自行调整。

注意:在结束一行时一定要按回车键。

显示器每屏显示 25 行时,最多每屏可显示 24 行正文。当正文还没有显示完时,编辑器会用

Continue(Y/N)?

如键入 Y 继续显示,否则终止。

常将刚刚编辑过的行的下一行称为【当前行】。

符号 # 被认为是读入的最后行的下一行的行号。

—3 使用模板 (template)

当键入一串字符并回车后该串便被送入一个特殊的内存区内,接着的输入便可利用这个存储区内的串快速输入(对该串复制、修改等)。这个特殊的内存区称为【模板】。

使用模板操作可使编辑速度加快。无论是输入命令或正文都可使用模板操作。对模板的操作可以通过下列键操作交叉进行:

INS 键:

插入/替代(或退出插入)模式。在进入替代模式,每键入一个字符便替代模板中相应字符而显示到当前行上;而在插入模式,键入字符在当前行上显示但不替代模板中字符,而是插入到模板中对应字符前面。该键是一个乒乓开关,即是一个反复键。

F1 键或→键:

每按键一次,依次将模板中一个字符复制到当前行上,并处于替代模式。

F2 键后跟模板中的一个指定字符:

将从模板开头到这个指定字符间的所有字符复制到当前行上,并处于替代模式。

F3 键:

把模板中所有其余字符复制到当前行上,并处于替代模式。

F4 键后跟一个指定字符:

将从指定字符开始到模板尾的所有字符复制到当前行上。如 F4 键后未跟字符,则清除模板中所有字符。

DEL 键:

在使用 F1 键逐个复制字符时,按一下 DEL 键,则模板中的对应一个字符不被复制。

F5 键:

按下键后显示一个 @ 号,它将 @ 前面的输入字符作废,光标移至下一行。这时编辑器处在退出插入模式,而将 @ 前面的字符复制到模板中,同时模板中原有内容被删除;在现在光标所在行状态就象行号后只出现星号一样,可以从头开始输入正文行。

ESC 键:

按下键后显示一个 \ 号,它将 \ 前面的输入字符作废,光标移至下一行;这时编辑器处在退出插入模式,但并没有将 \ 前面的字符复制到模板中,因而模板中原有内容不变。现在光标所在行状态就象行号后只出现星号一样,可以从头开始输入正文。

Backspace 键或←键:

删除当前光标前面一个字符。

另有两个键在编辑时也常用到：

F6 键：

在当前行上加上结束符 1AH,按回车键退出自动连续显示行号编辑。

Tab 键：

一次跳过几个空格。

—4 一行上可键入多个命令

可以在命令行键入多个命令,但一般你不要这样做。一行上键入多个命令时必须用一个分号(;)将行号与其它命令隔开。

—5 查询命令

功能:显示编辑器所有命令。

语法:?

说明:

```
C>EDLIN TURBO.C          /* 编辑一个新文件 TURBO.C */
New file
* ?                        /* 键入命令: ? */
Edit line                 line #
Append                    [# lines]A
Copy                      [startline],[endline],toline[,times]C
Delete                    [startline][,endline]D
End(save file)            E
Insert                    [line]I
List                      [startline][,endline]L
Move                      [startline],[endline],tolineM
Page                      [startline][,endline]P
Quit(throw away changes) Q

Replace                   [startline][,endline][?]R[oldtext][CTRL+Znewtext]
Search                    [startline][,endline][?]Stext
Transfer                  [toline]T[drive:][path]filename
Write                     [# lines]W
*
```

—6 编辑行(Edit line) 命令

功能:显示指定行内容并可修改

语法:[line #]

说明:

1. line 或 # 是指定要显示修改的行号;如省略行号而直接按回车键,则显示当前行内容。

2. 一按回车键便确认修改。注意:当你将一行调出后并未输入什么而直接按了回车键,则编辑器不会改变原行中内容。

—7 附加(Append) 命令

功能:将编辑的大文件未读进内存部分追加到内存中来

语法:[# lines]a

说明:

1. 假定内存可用空间为 100%, 则编辑器能将文件读入内存的最大限度为可用空间的 75%。超过时编辑的文件就有一部分不能装进内存, 而不装进内存是不能编辑的。要编辑就要使用本命令。当将一个文件刚读进内存时显示 (当内存装不下时无此显示)

End of input file

时说明被编辑文件已全部读入内存, 这时你不应使用本命令。

2. 为编辑内存中装不下的剩余文件部分, 必须先用 w 命令把已编辑的若干行写到磁盘上, 然后用本命令把未编辑的行从盘上调入内存。

3. lines 是要读进的总行数。若未指定它, 则编辑器读入最大数值的 lines, 直到占满内存空间 75% 为止。

4. 尽量不要将文件搞得太大, 另外清除内存驻留程序以使内存有最大的可用空间。

—8 拷贝 (Copy) 命令

功能: 将指定范围内的行复制到指定行号处

语法: [startline], [endline], toline[, times]C

说明:

1. startline 为起始行 (缺省为当前行), endline 为终止行 (缺省为当前行)。

它们组成指定的复制范围。它们缺省时逗号不能省略。

2. toline 是指定复制范围开始存放处的前一行的行号。其值不应落在指定复制的范围内, 即不应使行号重迭, 否则编辑器将印出 Entry error 信息, 提示输入出错。

3. times 是重复复制的次数, 若未指定, 其值为 1。

4. 复制中编辑器会自动重编行号。

—9 删除 (Delete) 命令

功能: 删除指定范围内的行

语法: [startline][, endline]D

说明:

1. 当省略起始行 (startline) 时, 缺省为当前行; 当省略终止行 (endline) 时, 只删除起始行。

2. 注意: 每删除一次, 所有的行号将重排。因此, 你不要老记着原来的行号! 必要时在新的操作前应用 L 命令显示一下重排后的行号。

—10 结束 (End) 编辑命令

功能: 把已编辑文件整个写入磁盘后退出编辑器, 返回 DOS。对已有磁盘文件生成后备文件 (*.BAK)。

语法: E

说明:

1. 在使用本命令前你应确信磁盘空间能装下内存中已修改的文件的全部, 否则用本命令将是危险的, 因为可能丢失已编辑文件。

2. 如果在磁盘上已有一个和正在编辑文件基本名相同而又是只读的后备文件存在时, 编辑器显示

Access denied - [drive:][path]filename. BAK

表示不能生成当前编辑文件的后备文件, 结果磁盘上你编辑的文件和那个后备文件将不变。

3. 对刚创建的编辑文件不会产生后备文件。

4. 如果没用本命令与 w 命令, 不管如何对内存中文件编辑修改, 都不会影响磁盘上原有

文件。

—11 插入 (Insert) 命令

功能:在指定行前插入正文

语法:[line]I

说明:

1. line 为行号,缺省时为当前行。当 line 大于实际最大行号或 line 用 # 代替,则最后一行的下一行为当前行,即正文将添加到文件末尾。

2. 当创建一个新文件时,开始必须使用本命令 (一般为 i),然后才能输入。

3. 使用本命令就是进入了“插入方式”,这时每按一次回车键后,光标自动到下一行,且下个行号随之自动出现,等待你继续插入。要退出这种连续插入方式,只要按一下 Ctrl-C 键。当退出插入方式时,紧跟在插入行后面的那一行成为当前行。

4. 每插入一行,编辑器将自动重排行号。

5. 输入控制字符:Ctrl-V 键操作后跟 ASCII 字符。如键入

Ctrl-V-[

后显示

^V[

相当于插入一个 escape 字符 (0x1b)。

—12 列表 (List) 命令

功能:显示指定范围内行的内容

语法:[startline][,endline]L

说明:

1. startline 为指定的起始行。如省略 (但后边的逗号未省略),则指当前行前的 11 行。

2. endline 为终止行。将从起始行开始显示 24 行。

3. 如果起始行与终止行全省略,则显示:当前行前 11 行、当前行和当前行后的行,直到满屏。

4. 当指定的行多于一页时,显示一页后编辑器提示

Continue (Y/N)?

键 Y 则继续。

—13 移动 (Move) 命令

功能:将指定范围内的行搬到另一个地方

语法:[startline],[endline],tolineM

[startline],+n,tolineM

说明:

1. startline 为起始行,省略时为当前行。

2. endline 为终止行,它必须不小于起始行。

3. toline 为指定范围的第一行搬家后的新行号。

4. +n 是指起始行后还有 n 行和起始行一起搬家。

5. 不允许出现指定的行号发生重迭。如有错误,显示“Entry error”。

6. 每搬家一次,编辑器便自动重排一次行号。

—14 显页 (Page) 命令

功能:按页显示行的内容

语法:[startline][,endline]P

说明:

1. startline 是起始行,省略时为当前行。
2. endline 是终止行,省略时显示为从起始行开始直到满页为止。
3. 每显示完一页(该页所占行数由起始行和终止行决定的),终止行的下一行即变成当前行。

—15 退出 (Quit) 命令

功能:不存储当前编辑的文件 (throw away changes) 而退出编辑器,返回 DOS

语法:Q

说明:

1. 当键入 Q 后,编辑器显示

Abort edit (Y/N) ?

当键 Y 则退出编辑,否则继续回到编辑状态。

2. 要在退出将编辑的内容存盘,可用 E 命令。

—16 替换 (Replace) 命令

功能:用另一个指定字符串代替指定范围内的某一字符串

语法:[startline][,endline][?][R][oldtext][CTRL+Znewtext]

说明:

1. startline 为指定范围的起始行,缺省时为当前行。
2. endline 为终止行,缺省时为读入内存文件的最后行。
3. oldtext 是文件中的原有的字符串,它将被替换。缺省时为最近一次你用 r 命令时的 oldtext,或者是最近一次使用 s 命令时用的字符串。如果缺省时你在先前并没有用这两个命令,则本命令终止。

4. newtext 是用户用于替换文件中的串 oldtext 的字符串。缺省时为最近一次你用 r 命令时的 newtext。如果缺省时你在先前并没有使用 r 命令,那末 newtext="" (即为空,结果要删除文件中的 oldtext),这是要注意的。

5. CTRL+Z 是键操作 Ctrl-Z,即输入一个 1AH 字符(行上显示是 ^Z),用于分隔 oldtext 和 newtext。只要你输入 newtext,则在其前必按 Ctrl-Z 键。

6. 字符串可以不用西文双引号括起来。注意,字符串可以是英文单词的一部分。

7. 如使用了问号(?),则当每找到一个 oldtext 时编辑器发一提示

O. K. ?

询问用户是否要用 newtext 替换 oldtext?如键 Y 或回车就替换,否则键 N 就不替换。此过程完后又继续往下寻找与替换。替换时还将在 O. K. ? 的上面显示已被替换后的那一行内容,供你在决定是否要替换时参考。

如果没有使用问号,则每找到一个便自动进行替换。

如果指定串未找到,将显示"Not found"信息。

替换中间要中止,可按 Ctrl-C 键。

8. 每替换一次,替换行的下一行就变成当前行。

—17 搜索 (Search) 命令

功能:在指定范围内搜索一个字符串

语法:[startline][,endline][?][S][text]

说明:

1. startline 为起始行,缺省时为当前行。
2. endline 为终止行,缺省时为最后一行。
3. text 是要搜索的串,一般不应缺省。如果缺省,那末当在此之前你刚用了 r 或 s 命令,则 r 命令中的 oldtext 串或 s 命令中的 text 串可作为这里的 text。如果缺省 时先前你并未使用 r 或 s 命令,则本命令终止。
4. 注意:命令字符 S 和 text 之间没有空格!如果你打了空格,则该空格被认为是 text 的一部分。
5. 如果你用了问号,搜索到一个串后显示

O.K.?

如果你按了 Y 或回车,则搜索终止,显示行的下一行变成当前行。如果你按了 N 键,搜索将继续进行下去。如果最后一个也没找到或到文尾时,显示“Not found”。

—18 传输 (Transfer) 命令

功能:将磁盘上的一个正文文件传送到当前正在编辑的文件中来(合并文件)

语法:[toline]T[drive:][path]filename

说明:

1. drive 是驱动器名, path 是路径名, filename 是文件基本名和扩展名。它们构成传输文件名。
2. 行号 toline 是传输来的文件的第一行所在的位置。如省略,则为当前行。
3. 传输来的文件插入后编辑器将自动重编行号。
4. 当传输进来的文件装不下时显示

Not enough room to merge the entire file

结果文件也没有传进来。

—19 写盘 (Write) 命令

功能:将正在编辑的文件若干行从内存写回磁盘

语法:[#lines]W

说明:

1. 当将一个文件刚读进内存时显示

End of input file

说明被编辑文件已全部读入内存,这时你不使用本命令。仅当读入的文件太大时不能全部读入内存时才用此命令。参见 a 命令。

2. #lines 是指从正在编辑的文件第一行开始的总行数。缺省时写入一些行数后,内存中还存行数占自由内存的 25%。也就是说,不管你怎么写回磁盘,有 25% 的内存装有文件内容。
3. 一些行被写入磁盘后,剩余行将自动重排行号。行号总是从 1 开始的。
4. 当用本命令将从第一行开始的一些行写回磁盘后,可用 a 命令再从磁盘上读一些文件的内容到内存编辑。值得一提的是,编辑器会自动管理这一进一出,即它知道如何正确写回到磁盘相应位置上,然后又怎样正确无误地从文件某一位置处(即未读入内存部分的某处)将未读入部分读入内存。这不用用户操心的。
5. 本命令对整个文件读进内存的情况也是可用的。当然,对此情况你显然不要用此命令。

第十七章 编译和调试程序

一个程序编辑后可以编译、连接,最后生成可执行文件。为了使程序得到预定的正确结果,排除任何潜在的错误,通常要经过静态检查、编译查错和程序调试这样几个过程。事实上,对用 Turbo C 编写的程序和对用其它程序设计语言编写的程序一样,也要经过这些过程,不过在这方面它还有自身的一些特点。

17.1 静态检查

对已编写好的源程序不忙进行编译,而是先进行源码录入、语法、逻辑和总体结构等检查,即所谓的“桌面检查”。静态检查是最能锻炼思维能力、提高编程水平的,初学者坚持这一步是有好处的。

由于 Turbo C 为结构化语言,它适用于自顶向下的方法,即可以从程序的顶层开始,逐渐移动到程序的下层,或者说程序员可以从主函数(main)开始,然后再“缺什么函数就补什么函数”,而每补一个函数只要在程序开始写上它的原型,随后便不管该函数在何处被调用。这样使程序整体结构就十分清楚。对有经验的程序员,源码生成可能是直接在机器上进行的,而不是用笔在纸上生成的。他们常采用自顶向下的设计方法,因为他们往往在编码前已在头脑中形成清晰的程序结构,具体的编码对他们当然也没有什么问题。在大多数情况下,他们是在编译查错或程序调试发生问题时才进行静态检查。

为便于静态检查,可采用所谓的“预防性设计”。例如:

变量名尽可能具有描述性,或者说用户一看到变量名就能从变量名悟出该变量的用处,也可以说不完全依赖记忆来辨识变量。常用的方法是将变量名用英文单词、单词缩写、单词首字符大写等标识,以及使用下划线隔开单词等;

严格检查指针;

函数的实参应和形参在数量、顺序、类型等方面完全一致;

源码必要的缩进使循环嵌套层次清晰;

复杂语句和复杂表达式应有注释;

尽量减少函数要求的数据个数和变动数据的个数;

编写的函数尽量是公用函数,减少特殊函数;

没有把握时多编写单功能函数,调试正确后再考虑合并;

尽量利用现成的经过证明是正确的函数。优先考虑用库函数;

没有把握时不考虑优化;在时间和空间方面,优先满足空间要求。而在空间允许时,优先考虑宏,而不用函数;

少用递归;

发现一个错误后看看其它地方还有类似错误否;

适用机型和移植性考虑;

重点对 Turbo C 不进行检查的问题清查,如数组下标超界,运算符和运算顺序,指针是否分配了一个有效的内存,数据初始化,自定义函数名是否和 Turbo C 库函数同名,函数是否有

说明和原型,文件打开方式等等。

最后,别忘了检查你使用的算法的正确性。

17.2 编译查错

在集成环境下通过编译器进行编译来发现错误。语法错误一般在编译时是比较容易被发现的。应当注意:未通过编译的程序不会产生可执行文件;而只要有一个编译错误发生,就不会有可执行文件产生。编译通不过,就不能进入程序调试。

建议你尽量利用集成环境来进行编译,而不要用命令行编译的方法。其次,在编译时要多看看“错误、警告及提示信息”(参见 17.5 节),对初学者更应该如此。

编译时编译程序将在每个阶段(预处理、语法分析、优化、代码生成等)尽可能多地发现源程序中的错误。当检测到一些错误后,编译将中止,编译器将有关源程序返回到编辑窗内,并停在一个错误行上。按回车键该错误将以(红)亮条显示在编辑窗顶行,当编辑开始后亮条自行消失。当有多个错误出现在信息窗(Message)时,可先按 F6 键使光标进入信息窗,然后上下移动光标键,你将发现编辑窗内的编辑行也同步滚动,再按 F6 键返回编辑窗,便能迅速到达错误行。

有时你反复检查错误发生行,并未发现什么问题。事实上,发生真正错误的行可能在它前面一行或若干行处!一个真正的错误也许会导致多个“错误”(非真正错误)信息出现。因此,你应当先将有明显错误的行改正,而对哪些一时还没有把握是不是真正错误的行暂时不管它。当改正了一处明显错误后立即编译一下,或许有些错误信息会自行消失。

当发生怀疑时编译将发出警告。在源文件中使用了与机器有关的结构,也将产生警告信息。因为有些情况可能是允许的,所以有时警告可被抑制,即不管它。但最好不要有这种情况发生,因为有时也是很危险的。

在集成环境里,有一个 Compile 菜单,它有几个子菜单,分别有不同的功能(详细说明参见《集成开发环境和缺省参数设置》一章)。

1. Compile to OBJ

它只生成 .OBJ 目标文件,而不是生成可执行文件 .EXE。对于不包含 main() 主函数的程序必须用它编译,而用其它几种编译将产生连接错误。生成的 .OBJ 文件可以被单独连接时使用。它不进行连接检查,例如下列程序中有个函数 mainm() 并没有在程序中定义,它也不予理会,照样生成 .OBJ 文件。此外,它也不进行“过时性”检查,即只要选择它便进行编译一次,而不管刚刚是否已编译过。

```
C>TYPE TEST1.C
main()
{
  int y=2;
  mainm();
  printf("ok! %y\n",y);
}
```

2. Make EXE file

生成 .OBJ 和 .EXE 文件。它相当于 Compile to OBJ 和 Link EXE file,但要进行编译查错和过时性检查。

3. Link EXE file

将一些 .OBJ 文件连接起来,生成 .EXE 文件。它也不进行过时效性检查。

4. Build All

它相当于 Make EXE file,只是不进行过时效性检查。当用 Ctrl+Break 键操作中断了该命令时,用 Make EXE file 即可继续。

5. Get info

获得当前集成环境包括装入源文件、编译连接和程序执行等方面的信息。其中,可能有

No program loaded (没有程序装入运行,在纯编译时发生)

No program running (没有程序运行)

Program running (程序正在运行,在调试时发生)

Program terminated (程序正常结束)

程序退出码 (Program exit code) 通常在程序执行后才有。

17.3 程序调试

一个程序即使经过编译、连接后没有发现任何错误或警告,但还不能认为在执行时总能得到预想的正确结果。因此,还应通过集成环境的调试器 (debug) 来调试程序,从而发现程序执行时的错误并加以修改。这一过程有时需要很长时间,具体实施也很复杂,并且往往因人而异。

一 从调试 WORDCNT.C 开始谈起

Turbo C 源盘上提供的这个 C 文件是供调试用的。该程序主要是用来统计一个文本文件中具有不同长度的单词个数及超长单词数。该程序有一些错误,如误将 && 写成 &,但这并不妨碍程序编译通过,或者说编译时不会产生任何警告或错误信息,并且编译连接后能生成执行文件 WORDCNT.EXE。然而当你用命令行执行时 C>WORDCNT 在输入要求读入数据的文件名 WORDCNT.DAT 后只打印出一行数据便发生死机了。当然,如果是在集成环境下执行它,则好一点,也会发生类似死机的现象,不过当你按 Ctrl+Break 键后,显示

```
          -Verify-
User break in wordcnt.c. Press ESC.
```

图 17-1

再按 ESC 键便可返回编辑器。

为了寻找发生死机的原因,可在集成环境里用调试器调试。有关调试器的菜单操作参见《集成开发环境和缺省参数设置》一章。

为方便叙述,重新将 WORDCNT.C 录入并加汉字注解,这有利于读者理解原程序的意图,对调试提供一定的帮助。

```
/* * * * *
 * WORDCNT.C - 调试指导例程
 * 注意: 本例程应和用户手册的程序调试章中的调试指导内容配合阅读
 * 例程中故意设置了一些错误
 * * * * */
/* 1 */ #include <stdio.h>
```

```

/* 2 */ #include <ctype.h>

/* 3 */ #define MAXWORDLEN    16          /* 单词最大长度 */
/* 4 */ #define NUL           ((char)0)    /* ASCII 码 0 */
/* 5 */ #define SPACE         ((char)0x20) /* 空格符 */
/* * * * * *
* 函数功能: 在行缓冲区内寻找下一个单词
* 进入函数: 指针 wordptr 指向单词的第一个字符或单词的前导空格
* 函数返回: 返回指向单词第一个字符的指针。如果行缓冲区中无任何单词, 则返回 NUL
* * * * */
/* 6 */ char * nextword(char * wordptr)
/* 7 */ {
/* 8 */     while ( * wordptr == SPACE )
/* 9 */         wordptr++;          /* 进到第一个非空字符 */
/* 10 */     return(wordptr);
/* 11 */ }
/* * * * * *
* 函数功能: 求单词长度。一个单词被定义为一串字符, 它终止于空格符或 NUL
* 进入函数: wordptr 指向一个单词
* 函数返回: 单词长度
* * * * */
/* 12 */ int wordlen(char * wordptr)
/* 13 */ {
/* 14 */     char * wordlimit;
/* 15 */     wordlimit = wordptr;
/* 16 */     while ( * wordlimit & * wordlimit != SPACE ) /* & 应为 && */
/* 17 */         wordlimit++;
/* 18 */     return( wordlimit - wordptr );
/* 19 */ }
/* * * * * *
* 主函数 main
* * * * */
/* 20 */ void main(void)
/* 21 */ {
/* 22 */     FILE * infile;          /* 输入文件名 */
/* 23 */     char linebfr[1024]; /* 行输入缓冲区, 为安全起见它定义得有足够大 */
/* 24 */     * wordptr;             /* 指向行缓冲区的下一个单词的指针 */
/* 25 */     int i;                 /* 搜索变量 */
/* 26 */     static int wordlenct[MAXWORDLEN],
/* 单词的长度是 1 到 MAXWORDLEN, 因此数组元素 Wordlenct[0]未用。数组
    定义为静态的, 故在执行时数组元素无须设置为 0 */
/* 27 */     overlenct;             /* 超长单词计数 */
/* 28 */     printf("警告: 这是一个供实际调试用的例程。在集成环境(Integrated\n");
/* 29 */     printf("Development Environment)下需中断运行时压 control-break 键\n");
/* 30 */     printf("调试方法详见用户手册中程序调试一章中的说明\n\n");
/* 31 */     printf("输入一个读入数据的文件名: ");

```

```

/* 32 */ gets(linebfr);
/* 33 */ if ( ! strlen(linebfr) ) {
/* 34 */     printf( "你必须指定一个输入文件名! \n" );
/* 35 */     exit(); /* 这句应为 exit(1); */
/* 36 */ }
/* 37 */ infile = fopen( linebfr, "r" ); /* 以文本方式打开, 原文本中回车 */
/* 38 */ if ( ! infile ) { /* 换行符\r\n 读入流中后只剩下\n */
/* 39 */     printf( "不能打开输入文件! \n" );
/* 40 */     exit(); /* 这句应为 exit(2); */
/* 41 */ }

/* 每次循环处理一行。注意: 如果一行字符长度大于输入缓冲区, 程序可能产生无效的结果。使用很
   大的缓冲区是不大可能的 */
/* 42 */ while ( fgets( linebfr, sizeof(linebfr), infile ) ) {
/* 43 */     printf( "%s\n", linebfr ); /* 印出读入行 */
/* 检查缓冲区并去处理新的一行 */
/* 44 */     i = strlen(linebfr);
/* 45 */     if ( linebfr[i-1] != '\n' )
/* 46 */         printf( "太长行开头部分\n\t%70s\n", linebfr );
/* 47 */     else
/* 48 */         linebfr[i-1] = NUL; /* 换行符不作为最近单词的计数字符 */
                                   /* 行缓冲区指针指向 linebfr 所指的第一个单词
                                   ( 即跳过 linebfr 所指串的前导空格 ) */
/* 49 */     wordptr = nextword(linebfr);
/* 每次循环处理一个单词。当 nextword() 返回 NUL 时循环结束, 这时表示已无单词 */
/* 50 */     while ( * wordptr ) {
/* 求该单词长, 长度统计数组的适当元素增加 1, 并指向单词后尾随的空格 */
/* 51 */         i = wordlen(wordptr);
/* 52 */         if ( i > MAXWORDLEN )
/* 53 */             overlenct++;
/* 54 */         else
/* 55 */             ; /* 加此空语句, 将引起十分严重的后果, 应删掉 */
/* 56 */         wordlenct[i]++;
/* 57 */         wordptr += i;
/* 寻找下一个单词 ( 假使有的话 ) */
/* 58 */         wordptr = nextword(wordptr);
/* 59 */     }
/* 60 */ }
/* 打印单词长度和此单词在正文中出现的次数, 每循环一次打印一次 */
/* 61 */ printf( " 单词长度 合计个数\n" );
/* 62 */ for ( i=1; i<MAXWORDLEN; i++ )
/* 63 */     printf( " %5d %5d\n", i, wordlenct[i] );
/* 64 */ printf( "超长单词个数 %5d\n", overlenct );
/* 关闭文件后退出 */
/* 65 */ fclose(infile);
/* 66 */ }

```

/* 为便于观察,不用 WORDCNT.DAT 中的数据,而用用户自编的文件 TEST.DAT,程序输出如下:

警告:这是一个供实际调试用的例程。在集成环境(Integrated Development Environment)下需中断运行时压 control-break 键
调试方法详见用户手册中程序调试一章中的说明

输入一个读入数据的文件名, test. dat

A C33 B2 D444 E5555

F23456789 G23456 H234567890123456

K K23 K2345 U2345678901234567 K2

单词长度 合计个数

1 2

2 2

3 2

4 1

5 2

6 1

7 0

8 0

9 1

10 0

11 0

12 0

13 0

14 0

15 0

超长单词个数 2 , */

二 调试几步曲

通过以下步骤的调试,你可以熟悉利用集成调试器调试程序的一般操作方法。

1. 进入集成环境,并用 File/Load F3 命令(或直接按 F3 键),当出现

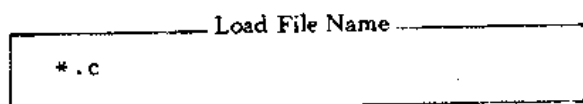


图 17-2

时键入 WORDCNT(扩展名可以不输),WORDCNT.C 便装入编辑窗;

2. 用 Make EXE file 命令编译,没有发现任何编译错误或警告信息,表明编译和连接无任何问题。检查目录,发现 WORDCNT.EXE 文件生成。

3. 在 DOS 提示符下,用命令行方式执行该程序 C>WORDCNT 屏幕显示:

警告:这是一个供实际调试用的例程。在集成环境(Integrated Development Environment)下需中断运行时压 control-break 键
(调试方法详见用户手册中程序调试一章中的说明。)

输入一个读入数据的文件名:WORDCNT.DAT

To be, or not to be, that is the question:

此后便死机了,只有热启动或复位主机。

4. 重复第 1 步。

5. 选择 Run Ctrl-F9 菜单项执行程序。如果程序事先未进行编译连接,则它先自行进行编译连接,然后执行。当执行到第 31 行时要求“输入一个读入数据的文件名:”,可键入 WORDCNT.DAT 或 TEST.DAT。Turbo C 提供有文件 WORDCNT.DAT,不过你也可以改用一个数据简单、便于观察的文本文件 TEST.DAT。例如 C>TYPE TEST.DAT

A C33 B2 D444 E5555

F23456789 G23456 H234567890123456

K K23 K2345 U2345678901234567 K2

现在键入了 TEST.DAT (此文件也可以用集成环境编辑器编辑后用 File/Write to 存盘生成),随后发现程序在输出

A C33 B2 D444 E5555

一行后便锁起了!这表明程序第 43 行后可能有问题。按 Ctrl-Break 键,屏幕出现

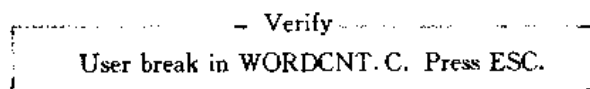


图 17-3

按 ESC 键后返回编辑窗,发现一(绿色)高亮条显示在某一语句上。这表示程序现在处在调试状态,高亮条又称【执行长条】,它指明它所在行是程序将要执行的第一个行。

6. 再选择 Run Ctrl-F9 菜单项执行程序,这回当出现

输入一个读入数据的文件名

时你不是马上键入文件名,而是按 Ctrl-Break 键后回车,屏幕显示

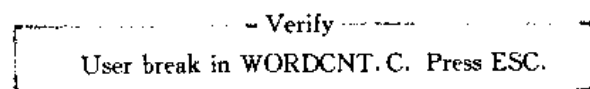


图 17-4

按 ESC 键该窗消失,程序处在调试状态,执行光条停在第 33 行上。如果是连接了两次 Ctrl-Break 键后,屏幕显示

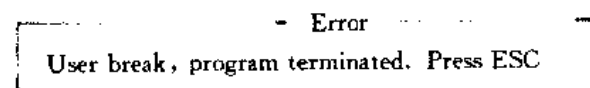


图 17-5

按 ESC 键又返回编辑窗,但程序已结束调试而处在一般编辑状态。

这是什么原因?

如果在程序执行时按了 Ctrl-Break 键,将产生一个【临时断点】,这时程序被中断执行。在集成环境下执行程序时,调试器正在和 DOS、BIOS 打交道,或者说调试器知道当前它正在执行的代码过程是 DOS 过程、BIOS 过程或程序本身。什么时候按了 Ctrl-Break 键后返回编辑器,使执行长条位于源代码相应行上?只有在有关的机器指令执行完后调试器才有可能定位并显示源代码行。有关 Ctrl-Break 的键操作可参见键盘操作及相关章。记住,当程序进入死循环时,必须按两次 Ctrl-Break 键才能退出。可以这样肤浅理解,第一次按 Ctrl-Break 键终止死循环,第二次按 Ctrl-Break 键是终止程序执行。

7. 将 WORDCNT.C 又读入(也叫【加载】)编辑窗,这回选用单步执行命令 Step over F8。所谓单步执行是每执行一次命令(按一次热键 F8),从主函数(main)开始执行主函数

中的一行源代码,而主函数中被调用的函数不被跟踪,或者说执行长条不会进入这些函数。

多次按 F8 键后就到达要求输入文件名。输入文件名后继续按 F8 键,当执行长条停在第 44 行上时,选 User screen Alt-F5 命令(也可直接按 Alt-F5 键),便在屏幕上看到从数据文件读入的一行,这表明 while 和 printf 语句工作正常。显示输出结果的屏幕 又称【用户屏幕】,它只显示程序输出结果等,不显示程序源码及调试表达式等。按任意非 控制键返回到编辑窗口。

8. 当执行长条停在 45 行时,选择 Evaluate Ctrl-F4 命令(也可直接按 Ctrl-F4 键),并在屏幕显示的 Evaluate(可称为【估算窗】)内键入变量名 i,有

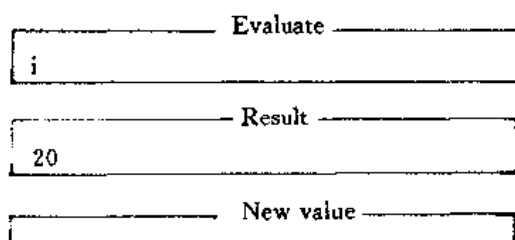


图 17-6

然后又连按 F8 键使执行长条停在 49 行上,并将编辑光标移到数组名 linebfr 上按 Ctrl-F4 键后,linebfr 一下子被拷入估算窗内,计算结果窗(Result)内显示

Result
"A C33 B2 D444 E5555"

图 17-7

从 i=20 到 linebfr 的内容为 19 个字符知道换行符已被删除。至此,程序无什么毛病。

9. 继续按 F8 键进入 while 循环后,每当执行长条停在 52 行时第一步先将编辑光标移到变量 i 上后按回车键,变量 i 被拷入估算窗,结果窗便显示 i 的值;接着按 ESC 键或 F10 键,然后又按 Ctrl-F4 键(此时编辑光标仍在变量 i 上),并按住向右移动光标键→,这时可以看到第 51 行变量 i 后的字符被依次拷入估算窗内

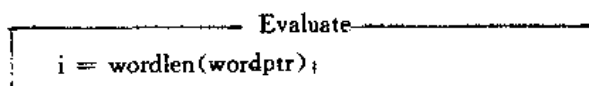


图 17-8

先按退格键将右括号和分号删去,然后按 Home 键窗内光标便一下子移至变量 i 上,按 Del 键便可依次删去不要的字符,使估算窗最后只有指针名 wordptr,按回车键后有

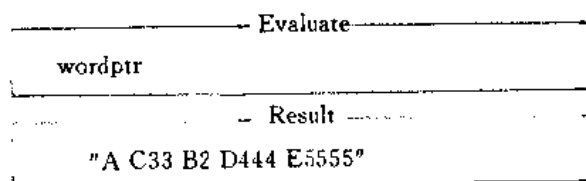


图 17-9

依次循环可以看到 i 的值和 wordptr 的内容在变化。例如第二次循环时对 wordptr 有

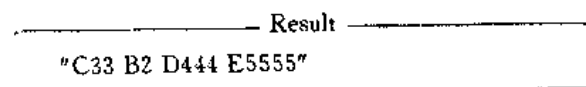


图 17-10

第三次循环时,将发现 i 有 0 值,这显然不对! 因为单词长度不可能为 0 的。从语句

```
/* 56 */      wordlencnt[i]++;  
/* 57 */      wordptr += i;
```

容易看出当 i 为 0 时指针 wordptr 不会移动,这时不管怎么循环,总有

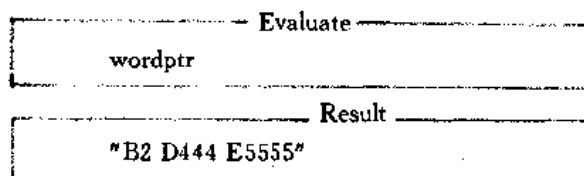


图 17-11

这就是导致死机的原因。

10. 现在在执行第 57 行前按 Ctrl-F4 并输入 i, 自然在结果窗内显示 i 有 0 值。移动向下光标键↓, 将光标移到新值窗 (New value) 内, 并输入一个值, 例如 3 后按回车键 (必须按回车键才确认。如果没有按回车键而是去按了 ESC 键, 则调试器不会改变表达式的值), 就有

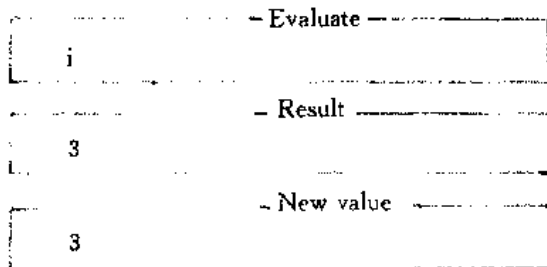


图 17-12

然后再按 F8 进行单步调试, 发现 wordptr 移动了, 有

Result

"D444 E5555"

图 17-13

但随后又出现了 i 为 0, wordptr 又不移动了。换句话说, 现在确信死锁的原因了。

这一步也说明了当新值窗输入一个新值后, 接着的运算 i 将用这新值进行。当然, 它以后还会被程序改变。这种通过改变现行变量值来试探程序运行结果有时是有用的, 它可以观察程序内部行为。例如, 想看看某一函数在接受实参后的情况, 但这种实参值也许很难向函数传送, 那末可以在调用该函数前先修改一些变量值来达到目的。

11. 为观察表达式的情况, 每一次都要按 Ctrl-F4 键并输入相应的表达式这太麻烦了! 有一个比之方便的方法, 就是利用 Break/Watch/Add watch Ctrl-F7 命令 (也可以直接按 Ctrl-F7 键) 向编辑窗口下面的监视 (Watch) 窗口内输入这些表达式或变量。输入的表达式称监视表达式或调试表达式。输入表达式的方法同输入调试表达式或变量大致是一样的 (关系增加、删除和编辑表达式的更详细的说明参见《集成开发环境和缺省参数设置》一章)。每按一次 Ctrl-F7 可以输入一个表达式。例如, 输入调试表达式

i
wordptr

或带限定的变量

main.i
main.wordptr

这回连续按 F8 键就可以直接从监视窗内看到变量的变化了。

什么叫限定表达式或限定变量? 我们来看程序 TEST2.C

```
C>TYPE TEST2.C
px(int x)
{
    static int y=10;          /* y 是静态变量 */
    y += x;
}
```

```

printf("%d %d\n",x,y);
}
limit-1()
{
int y=100;                /* y 是局部变量 */
px(y+8);
printf("%d\n",y+8);
}
limit-2()
{
px(2);
}
main()                    /* 这是一个观察静态变量产生的结果程序 */
{
static int y=2;           /* y 是静态变量 */
printf("y=%d\n",y);
limit-1();
limit-2();
}

```

对这个多层调用函数的程序中的变量 y 如果用一个表达式 y 来观察,当每一个函数源码都很长以至调试时无法看到函数名时,势必会发生这样的疑问:这个 y 是哪一个函数里的 y 呢?因此 Turbo C 允许你在输入变量名时带限定标识符,限定标识符一般指引用该变量的一个或多个函数名,它们放在变量名的左边,函数名与函数名、函数名与变量之间都用标点符号中的句号相隔开。注意:在开头是不能有句号出现的(不过在用 F1 键对 Add watch 获得帮助时,Turbo C 帮助信息中说明这一点时,误认为可以在开头有句号出现)。因此,可以输入调试表达式

```

limit-1.px.y
limit-2.px.y
main.limit-1.px.y
main.limit-2.px.y
limit-1.y
main.y
px.y

```

都是正确的(它们都没有带格式说明符,因此都用缺省格式说明符)。注意,一般只有当执行亮条进入相应的函数内它们才会有正确的值显示。但这里你将发现对有些限定变量当执行光条在其它函数内时它依然有正确值。

这种完整地限定变量名适用于两种情况:

- 需要检查不同模块中的静态变量;
- 需要查看不同函数中的局部变量。

对静态变量一般只在变量名前加上其直接所在函数名即可。

12. 现在选用 Run/Program reset Ctrl-F2 命令(或直接按 Ctrl-F2 键),则执行长条自行消失,这表明结束调试。此时,Turbo C 放弃所有分配给 WORDCNT.EXE 的内存,关闭打开的文件,结束 WORDCNT.EXE 的执行。注意,WORDCNT.C 仍在编辑窗内。

13. 现在重新开始调试。先选 Break/watch/Remove all watches 命令,将监视窗内所有调试表达式一次性全清除了,并按 Ctrl-F7 键依次键入几个新调试表达式

```
wordlencnt[0]
wordlencnt[1]
wordlencnt[0],15
*wordlimit
*wordlimit != 0x20      /* 因为不能用宏 SPACE,但可用它定义的值 */
*wordlimit != ' '      /* 等号后面单引号内为空格符 */
*wordlimit & *wordlimit != ' '
wordlimit-wordptr
```

接着你可以用跟踪进入函数的命令 Run/Trace into F7 (或直接按 F7 键)一步一步调试,可以发现会有这样一些值

```
wordlencnt[0]: 1
wordlencnt[1]: 1
wordlencnt[0],15: 1,1,0,0,0,0,0,0,0,0,0,0,0,0,0
```

及执行长条进入函数 nextword 后

```
*wordlimit: 'B'
*wordlimit != 0x20: 1
*wordlimit != ' ': 1
*wordlimit & *wordlimit != ' ': 0
wordlimit-wordptr: 0
```

这显然有问题。经过分析不难发现将运算符 && 搞错了。因此移动编辑光标键到 16 语句,把它改成

```
/* 16 */ while ( *wordlimit && *wordlimit !=SPACE )
```

改正后按热键 F2,将当前程序无条件存盘,以免再调试时万一程序垮台而意外丢失。

注意:文件存盘并不改变文件仍处于调试状态,这可以用 Compile/Get info 命令证实,此时屏幕显示

Information	
Current directory :	C:\TC
Current file :	C:\TC\WORDCNT.C
File size :	4008(Max: 64507)
EMS usage :	OK
Lines compiled: 368	Program running ←程序正在执行
Total warnings: 0	Program exit code
Total errors : 0	Available memory: 18K
Press any key	

图 17-14

当然从执行长条依然存在也可说明当前仍处于调试状态。

跟踪进入函数内部操作可以看到被执行函数内部变量的情况,因此比单步跟踪 F8 键操作要好一些,但显然要慢一些,多化一些时间。另一方面,F7 键操作也不能跟踪进哪些没有源码的函数内。特别像 Turbo C 的库函数一是没有源码,二是没有它们的调试信息,因此,也不

能进行跟踪进入。为了理解这一点你可以这样做：

- (1) 选 File/OS shell 命令,暂时退出集成环境,进入 DOS。
- (2) C>DEL WORDCNT.OBJ (删除文件操作)
- (3) C>DEL WORDCNT.EXE
- (4) C>EXIT (返回集成环境)
- (5) 用 Options/Compiler/OBJ debug information Off 命令 (原为 On,现改为 Off)
- (6) 按 F8 键进入单步调试,屏幕显示

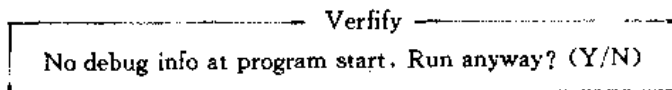


图 17-15

这表明 OBJ 文件中没有调试信息存在。如键 Y 则程序整个执行完,否则没有任何反映,即不进入调试状态,而进入编辑状态。因此,要确保有

Options/Compiler/OBJ debug information On

或者说,在用 Compile/Compile to OBJ 命令时也要在这种状态下进行操作,才能生成供调试用的 *.OBJ 目标文件。当这种目标文件被编译和连接后就带有调试信息,如果又有源码,则可被跟踪。

如果在调试时出现这种无调试信息的提示,你应将此命令设置正确,然后在源程序行上某处无关紧要处键一空格,再按调试键进入调试。

14. 为了验证改正后的情况,移动编辑光标到 26、55 行上,将它们分别改成

```
/* 26 */ static int n, wordlencnt[MAXWORDLEN],
/* 55 */      n++;
```

即增加一个静态变量 n。现在按 F6 键,使光标进入监视窗,按光标上下移动键 ↑ ↓,使光标移到 8 个表达式

```
wordlencnt[0]
wordlencnt[1]
wordlencnt[0],15
* wordlimit
* wordlimit != 0x20
* wordlimit != ' '
* wordlimit & * wordlimit != ' '
wordlimit - wordptr
```

中 * wordlimit != 0x20 上。这时你可用两种方法更改此表达式,一是先按 Ctrl-Y 键,则该表达式立即从监视窗中消失;接着你按 Ctrl-F7 键,并在 Add watch 小窗内输入表达式

main.n

按回车键后该表达式立即添加到监视窗内。第二种方法是当光标在该表达式上时,直接按 Ctrl-F7 键,此时表达式 * wordlimit != 0x20 立即被拷贝到 Add watch 窗内。你可以立即输入表达式 main.n 的每一个字符。当你键入表达式第一个字符时,原表达式的全部内容自动消失,而你刚刚键入的一个字符出现在 Add watch 窗的最左边,这就是立即输入状态。或者你也可以通过修改已拷贝进来的表达式来输入新表达式,即先处于修改状态,然后再改动原表达式。为处于修改状态,你必须先按 PgUp、PgDn、Home、Del、End、→、← 或退格键中的任一键,才

能使当前状态处于表达式修改状态。除 PgUp 和 Home 键能使光标立即移到表达式头部外,其余的都使光标在表达式尾部。处在修改状态后你就可以用编辑键修改表达式。当你要取消 Add watch 窗内的全部内容而用监视窗内的表达式时,只要再按一次 Ctrl-F7 键就可以了。

表达式输入后按 F6 键使光标回到编辑窗。

这一次我们不是直接用 F7 键进行跟踪进入调试,而是先在 15 行上设置一个断点。为此移动编辑光标到该行上,然后用 Break/watch/Toggle breakpoint Ctrl-F8(或按 Ctrl-F8 键),该行上立即出现一个(红)亮条,这表明断点已设置好。注意:断点设在函数 wordlen 内,这就是说断点不一定非设在主函数内。

先按一下 Ctrl-F9 键,出现

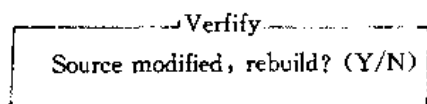


图 17-16

这是告诉你源文件已修改,要否重新编译。如键入 N 表示不管,还是用原来的。如果修改后的程序只是在已有行上增加一点注释之类的说明,自然可键入 N,因为程序没有作实质性的修改。而这种边调试边用注释将调试结果用记录在源程序中是很有用的,可以避免遗忘。不过当程序作了实质性修改(包括增删行)时一般应键 Y,重头开始,否则可能发生“阴差阳错”,到后来越调试越糊涂。

键入 Y 后程序又开始编译并执行。当输入文件名后,又连接两次 Ctrl-F9 键。每次按键后发现执行长条停在断点行上。容易看出,利用设置断点可以一下子执行多行程序,而直接执行到希望单步调试行处。这大大提高了效率。现在按 F7 键后执行长条下移一行,这表明现在进入单步跟踪调试状态(也可以用 F8 键进行单步调试)。调试中可以看到像

```
main.n: 3
* wordlimit : '4'
* wordlimit != ' ': 1
* wordlimit & * wordlimit != ' ': 1
wordlimit-wordptr: 1
```

这样一些正确的结果。这里看到了一个将设置断点和单步调试结合起来使用的例子。

15. 现在想看看 linebfr 的情况,按 Ctrl-F7 键又向监视窗内增加一个表达式

linebfr。该表达式进入后发现原监视窗内的一个表达式消失了。没关系,先按 F5 键,结果使监视窗和编辑窗不再处于“同屏状态”,而是处于“分屏状态”,或者说,当按 F6 键时,或者编辑窗(Edit)占用整个屏幕,或者监视窗(Watch)占用整个屏幕,而不像同屏状态它们共用一个屏幕。注意:F5 键是乒乓开关(Toggle switch),即按一下它,当前屏幕处于“分屏状态”,再按一下便是“同屏状态”,再按一次又回到“分屏状态”,如此交叉出现。现在按一下 F7 键,看到执行光条在编辑窗内的位置,但看不到一个调试表达式;如按一下 F6 键便看到所有调试表达式了。如此反复按 F7 键和 F6 键(也可当需要时才按)便可不断了解调试状态。

16. 将编辑光标移到断点上,按 Ctrl-F8 键后发现断点所在行上的高亮条消失,再按一下 Ctrl-F8 键亮条又重新出现,这表明 Ctrl-F8 键也是一个乒乓开关。现在按 Ctrl-F8 键消去这个断点,并按 F7 键,例如调试到 55 行。再移动编辑光标到 15 行上,选命令 Run/Go to cursor F4(可以直接按 F4 键)后发现程序一下子执行到编辑光标所在行上。

这里分几种情况:

(1) 如果编辑光标在一个非执行语句行,例如第 3 行上,按 F4 键后显示

Error
No code generated for this line. Press ESC.

图 17-17

这表明该行非执行行,不产生执行代码。按 ESC 键后程序进入单步调试状态。还有一些行也可以使程序进入单步调试状态,如 33 行那样的中间执行行。

(2) 如果程序在调试时已输入文件名后,现将编辑光标停在执行不到的行上,例如第 39 行(当键入文件能被打开时它不会被执行)或第 28 行上,那么按 F4 键后程序将一直执行到结束(对本例),从而自动中止调试。

Run/Go to cursor 是一次性操作,并不设立永久性断点。

17. 现在用 Ctrl-F8 键操作在 15 行和 55 行上分别设置一个断点,并且用 Break/watch/Remove all watches 命令一下子删去监视窗内所有调试表达式。按 Ctrl-F9 键,可以看到每按一次,程序执行到 55 行,再按一次执行到 15 行,又按一次又执行到 55 行,如此等等。

18. 程序现在处在调试状态,将光标移到第 3 行上,按 Ctrl-F8 键,在第 3 行下马上出现

↑ Invalid Breakpoint
Insert anyway? (Y/N)

图 17-18

如键 Y,该行即被设为断点行,如键 N,则不设。为什么会出现这种显示?因为这一行不含有直接可执行的调试语句,因此 Turbo C 调试器认为即使在此行设置断点,也是一个无效断点(Invalid Breakpoint)。程序中空行、纯注释行、定义变量未初始化行、预处理行及函数原型说明行等等若设断点,都将被认为是无效断点。显然,利用这种方法可以检查某一行能否置为有效断点。删除任何断点包括删除无效断点不会遇到这种麻烦。

注意:程序不是处在调试状态时是不会有这种显示的,按 Ctrl-F8 键后光标所在行立即设为断点行。为此,按 Ctrl-F2 键中止当前调试,执行光条消失,但原先设置的断点并不会消失。分别将 3 行、4 行、24 行、44 行及 36 行、41 行下面的空行设为断点行。现在按 F8 键进入单步调试,在第 3 行下面出现

↑ Invalid Breakpoint
Ignore, Erase, Clear all bad, Skip all bad.

图 17-19

它表示遇到了一个无效断点。

如按 I 键,表示对这个无效断点不管,程序跳到下一个无效断点上,又显示这样的信息,如此等等,直到所有的无效断处理完才真正开始有效调试。

如按 E 键,程序自动(永久)删除无效断点所在行上的断点标记,即该行不再是断点行。当然,不会删除该行上的源码。此后,程序跳到下一无效断点。

如按 C 键,所有无效断点一次性全被删除,当然所有有效断点不会被删除。

如按 S 键,程序依次跳过所有无效断点行才开始调试,应当指出,那些无效断点并未消去,只是在调试时已不起作用,即遇到这些行时程序不会停下来。

注意:如果 return 后面无表达式即为

return ;

时,该句也将当非执行语句处理,单步调试时会跳过。如果在此句上设置断点,也会出现上述提示。

19. 如果遇到第一个无用断点时键入 C,即清除了所有无效断点,现在连续选用菜单命令 Break/watch/View next breakpoint,则可以看到编辑光标反复在 44 行、15 行和 55 行之间来回移动。该命令是在当前状态下将编辑光标移到下一个断点上,而不是跟踪执行到下一个

断点上。如果要跟踪执行到下一个断点上,用 Ctrl-F9 键操作。如果要快速浏览一下程序中现有断点,以便决定是否要增删某些断点时是很有用的。

20. 当怀疑函数 wordlen 有问题时自然想迅速了解一下它的定义情况,这时可用命令 Debug/Find function,这时显示

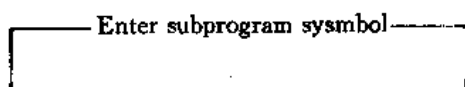


图 17-20

输入函数名 wordlen (注意:别忘了按回车键确认),则编辑光标一下子移到第 12 行上,即该函数定义行上。如果这个函数不存在,则显示

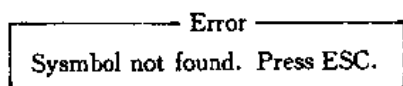


图 17-21

注意:要使用该命令有两个条件,一是调试前要有

Options/Compiler/Obj debug information On

二是必须在调试状态。在没有进入调用状态时可以看到该命令显示为淡颜色,不能用移动光标键进行选择。

21. 如果在调试前设有

Options/Compiler/Code generation/Standard stack frame On

则进入调试后命令 Debug/Call stack Ctrl-F3 可用 (即使在 Tiny 存储模式下编译时编译器发出警告

Linker Warning: No stack

但仍可用)。

如果用 F7 键操作跟踪进函数 nextword 定义的第一行即程序的第 6 行上,按 Ctrl-F3 键,在屏幕右上角弹出

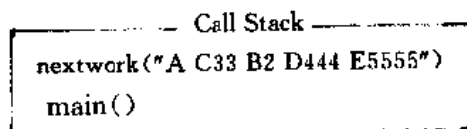


图 17-22

调用栈不仅显示了函数的名称,同时还显示了参数值。继续用 F7 键操作,然后让执行长条停在第 10 行上,可能有

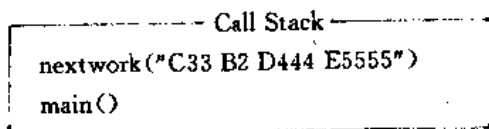


图 17-23

显示了函数 nextword 内当前语句的执行时函数参数情况。

调用堆栈命令同查找函数定义位置一样,也要在进入调试时才能被选择。

结束调用栈访问可按 ESC 键。开始调试时它也会自行结束。

22. 现在把命令

Options/Compiler/Code generation/Standard stack frame On

改成

Options/Compiler/Code generation/Standard stack frame Off

再来调用堆栈,结果发现没有什么不同。但是我们改对程序 TEST2.C 来调用堆栈,将发现对

这两种情况是不同的。例如,在 On 时有

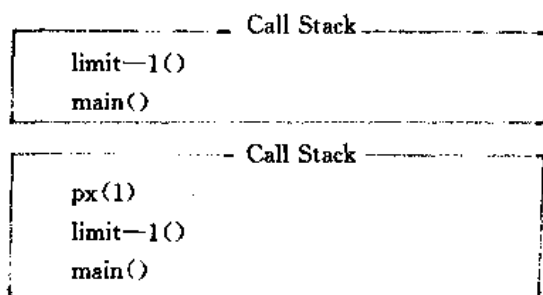


图 17-24

而在 Off 时却只有

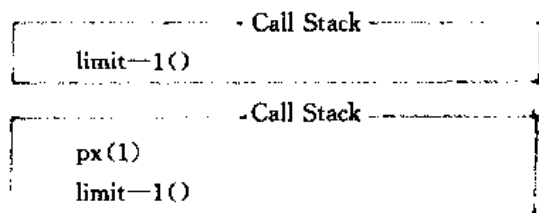


图 17-25

为什么? limit-1() 函数没有参数,这种没有使用参数的函数在编译时压缩了入口和返回码,结果使代码短而执行快,然而简化了反向跟踪调用栈的处理。因此,为了看到所有栈的情况,最好使用 On 开关。

利用观察栈调用,有时还可以发现某些函数调用时栈丢失的情况,以至可以判断函数有某种问题存在。

23. 如果原先我们调入 WORDCNT.C 进入编辑窗后进行调试状态,即执行光条存在于编辑窗内。这时为了验证调用堆栈跟 Standard stack frame 命令的关系,我们在没有终止当前调试状态的情况下利用 File/Load 命令将文件 TEST2.C 装入编辑窗,然后进行调试。这时你将发现,原 WORDCNT.C 又被自动调入编辑窗,调试的是它而不是 TEST2.C! 因此,为了调试一个新程序,你一定要按 Ctrl-F2 键退出调试状态,然后将有关程序装入调试。

24. 当正在调试一个程序时,有时可用 File/OS sell 命令暂时退出集成环境而返回 DOS 状态,以便利用 DOS 命令做一些事情。在大多数情况,不会有什么问题。然而当内存中有一些常驻程序,或者对一些可以在中文状态下运行 Turbo C 软件的操作系统,使用该命令时将出现

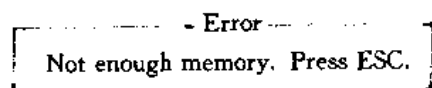


图 17-26

事实上,在调试状态 Turbo C 将占用更多的内存,即使在暂时退出时也不释放常驻内存。为此,只有先用 Ctrl-F2 键操作中止调试状态,从而释放因调试多占用的内存,然后才能使暂时退出成功。

25. 现在假定我们为了记录当前调试结果将编辑光标移到了程序尾部,由于程序可能很长,以至于编辑光标滚出了编辑窗。写出注释后想立即返回执行长条所在行,这时可按 Ctrl-F3 键后接着按回车键(实际选择栈顶上的函数),则执行长条所在行立即被卷入编辑窗内。

注意:不管编辑光标在何处,按 F7 键马上可使执行光标卷入编辑窗,但它与用 Ctrl-F3 键操作不同,它在卷入光标的同时还执行了一步源程序。

26. 现在我们另行编写一个程序 TEST3.C,程序中把 WORDCNT.C 当作一个子进程来执行。 C>TYPE TEST3.C

```
/* 1 */ #include "process.h"
```

```

/* 2 */    #include "stdio.h"
/* 3 */    main()
/* 4 */    {
/* 5 */    char *ptr="执行子程序";
/* 6 */    int no;
/* 7 */    no=spawnl(F-WAIT,"M29.EXE",NULL);
/* 8 */    printf("子程序 WORDCNT.C 退出码=%d\n",no);
/* 9 */    if(no==0)
/* 10 */        printf("%s ok!",ptr);
/* 11 */    else
/* 12 */        printf("没有执行完子程序! \n");
/* 13 */    }

```

用 F8 键操作来调试这个程序。当执行到要求输入文件名时分两种情况输入：

(1) 实际存在的文件名,例如 C:TEST.DAT、C:WORDCNT.DAT 等。

(2) 一个实际上并不存在的文件名,如 A:TEST.DAT(A 驱动器中根本没有插磁盘,或者盘上没有这个文件)等。

观察三个内容：

(1) 用 Alt-F5 键操作看看用户屏幕上程序输出了什么。

(2) 执行长条的移动情况。

(3) 用 Compile/Get info 命令看看 Program exit code 栏。然后把第 35 行和 40 行上的语句改成

```

/* 35 */    exit(1);
/* 40 */    exit(2);

```

再重新调试,看看结果有什么不同。exit() 看起来似乎还允许,但效果不好。

最后把 TEST3.C 的行 9 到行 12 合并成两行：

```

if(no==0) printf("%s ok!",ptr);
else printf("没有执行完子程序! \n");

```

再观察执行长条的移动情况。可以发现,调试是“调试一步是执行一程序行”,因此更改后与更改前是不同的。这一结果对于把

```
for(k=0;k<100;k++) x +=1;
```

改成

```
for(k=0;k<100;k++)
x +=1;
```

来观察变量 x 的值明显是有区别的,尽管这不影响程序的最后输出结果。

用 F7 键操作不会跟踪进入子程序 WORDCNT.C。

27. 现在在 TEST3.C 的第 2 行下增加一行

```
#include "wordcnt.c"
```

与此同时更改行 7

```
/* 7 */    no=mainn();
```

然后又调入 WORDCNT.C,并将行 20 改成

```
/* 20 */ mainn()
```

最后重新编译 TEST3.C。现在用 F7 键操作,当执行第 7 行时发现 WORDCNT.C 自动调入编辑窗。

28. 在调试中如果你把 WORDCNT.C 调入后作了一些修改后用 F2 存盘,然后又返回来调试 TEST3.C,则调试器可能不能识别这种修改,仍按原编译内容进行,这是要注意的。

29. 在调试中,如果将缺省命令 Debug/Display swapping Smart 改成

Debug/Display swapping None

再从头开始用 F8 键操作来调试。当执行到第 32 行要求输入文件名时,这回发现屏幕并不切换到用户屏幕去,你输入的文件名就在当前编辑窗内,它将替换当前屏幕上的字符。直到你按了回车键后,屏幕上刚刚被替换的字符又重新恢复显示,程序继续往下执行。

对有些程序,例如时钟中断程序写屏,编辑屏幕可能被修改而不切换到用户屏幕,这时你最好选用

Debug/Display swapping Always

则执行每一条语句时屏幕都会被切换。另一种方法是万一出现这种情况,使用命令

Debug/Refresh display

重新刷新屏幕。

30. 现在将缺省的命令 Debug/Source debugging On 改成

Debug/Source debugging Standalone

或

Debug/Source debugging None

在开始调试就出现

No debug info, Run any (Y/N)

的信息提示,键 Y 执行整个程序,键 N 则不执行程序也不调试,回到编辑状态。当开关设为 Standalone 时,如果当前盘上装有专用的 Turbo 调试器程序(是独立的不属于集成环境的),则是可以进行调试的。

31. 现在回过头来看看程序输出结果。少了一个单词长度输出,因为只有 15 个,而有一个单词长恰巧是 16。另外怎么会有两个超长单词?明明只有一个。如果把 62 语句改成

```
/* 62 */ for (i=1; i<=MAXWORDLEN; i++)
```

问题得到解决。假如输入三个调试表达式

```
i  
wordlencnt  
wordlencnt[0],18d
```

后重新用 F8 键操作从头开始调试,则不难发现 wordlencnt[17] 中被填入数据,而 wordlencnt[17] 并没有被定义!如果设想对超长单词不用该数组元素统计,则会发现哪个空语句(语句行 55)是完全多余的了。从这个例子也可以看出 Turbo C 不对数组下标越界进行检查。

32. 当对程序几处修改后,便可以输入 WORDCNT.DAT 来进入最后的测试了,因为该数据文件中的数据比 TEST.DAT 一是多,二是数据编排随机性大。如果最后输出结果正确无误,那么程序正确性和耐用性便得到了很好的考验。

三 C语言的缺点

C语言的一些缺点增加了程序调试的困难性。

1. 有无所不在而较难掌握的指针。错误的指针可能使数据写到内存中使你感到意外的地方,因为它可以写到内存中任何位置。例如,它可能将数据写到内存中原来存放程序执行期间的堆栈处,则函数调用期间的联结或局部变量会被破坏,结果调用函数就不能正确返回;如果将数据写到内存中存放标准函数库的模块里,则函数功能可能被改变;如果写到内存中存放操作系统部分,情况将更糟,诸如输入输出可能完全乱了套,程序有可能莫名其妙地终止或无限运行下去。总而言之,执行时出现的奇怪现象使你十分头疼。

2. 运算符类型多,优先级也多,有些还与人们的习惯不怎么相符。特别是容易存取的机器层次操作,稍不小心,便会发生意外。

3. 数据类型和边界检验太弱,转换比较随例。

四 高效调试程序的一些建议

(一)数据准备

(1)有效性

包括有实际意义的数据首先要被测试。

(2)全面性

别忘了对可能误操作引起的数据测试。

(3)代表性

众多的一类数据中找出最有代表性的数据十分重要,因为不可能对所有数据作试验,而只能取其典型性的部分。例如, -1、0 和 1 对某些函数测试是三个具有代表性的整数。

(4)特殊性

一些具有特殊性的数据必须测试一下。例如,对循环中开始和结尾时循环变量引起的结果是值得重视的。又例如一些限制条件等也是值得注意的。

(5)清晰性

数据应是易理解的,最好一目了然,一看就知道将有什么样的测试结果。

(6)渐进性

数据一般分批进行试验,先简单后复杂,先特殊后一般,先少后多。

(7)隐含性

为减少数据输入次数,提高效率,可考虑将某些数据直接装在程序中,即赋给一些变量,程序执行时自动调入。

(8)利用宏

有些参数可以利用定义的宏进行,达到一改百改的目的。这可以考虑用命令

Options/Compiler/Defines

来定义调试用的宏。

(9)数据列表

在最终调试时,按一张功能与数据对应的表测试也许更有效。避免了临时判决和重复,不易发生遗漏,具有整体性。

(10)必要时可由用户准备数据,而不是由程序员自己准备数据。这样有可能发现一些意想不到的问题。

(二)结构性考虑

(1)定义函数尽量不用全局变量,留心静态变量。

(2)先调试单个函数

这样缩小了调试范围,易调试,且基础可靠,在别处调用此函数时一般就可放心不管。必要时,另编一个由 main() 函数和调试函数组成的程序来单独调试也是值得的。

(3)利用添加注释区域和命令

Options/Compiler/Source/Nested comments On

(即允许嵌套注释编译)来减少调试范围。调试范围越小,越易发现问题。对由多个源程序组成的程序一般也宜先单独调试通过后再连起来调试。

(4)多个目标文件最好在同一环境下(如存储模式等)编译,以免条件不同引出连接麻烦。优先考虑利用集成缺省环境。

(5)调试时每一行是否都被执行? 不要忘了这一条。必要时应进行模拟试验。

(6)连环性影响

当一处修改后可能引起另一处出问题,即使该处原来已被证明是无问题的。

(7)程序多方式运行

如程序可连续、反复多次运行。有些程序只在第一次执行时正确,第二次运行时便失败了,应当怀疑数据初始化及以后处理是否正确。

(8)考虑在不同的存储模式下编译连接产生的实际效果。

(三)源程序中用插入调试函数或宏来观察程序运行情况

(1)测试并终止

异常终止进程	abort()
测试一个条件并可能使程序终止	assert()
注册终止函数	atexit()
终止程序执行	exit()
复位错误标志	clearerr()

(2)测试并转移

建立一个硬件错误处理程序	harderr
硬件错误处理函数	hardresume()
	hardretn()
设置 Ctrl-Break 处理程序	ctrlbrk()
非局部转移	setjmp()
执行非局部转移	longjmp()
浮点运算处理程序	—mather()
用户可修改的数学错误处理程序	matherr()
向正在执行程序发送一个信号	raise()
设置某一信号的动作	signal()

(3)获取错误信息

格式化输出函数	... printf()
返回指向错误信息字符串的指针	—strerror()
	strerror()
检测流上错误	ferror()

获取扩展错误信息	dosexterr()
对应于给定错误代码的图形错误信息串	grapherrormsg()
在上次图形操作遇到错误后,给出一错误代码	graphresult()
图形状态查询函数	
全局变量 errno 的值	
编译出错指令	#error

(4)用随机数替代测试

随机数发生器	rand()
	random()
对随机数发生器进行初始化	randomize()

(5)必要时发声告警

初始化随机数发声器	srand()
以指定频率打开 PC 扬声器	sound()

(6)通过延时仔细观察(文本水平移动、屏幕刷新等)

执行挂起一段时间(秒)	sleep()
将程序的执行暂停一段时间(毫秒)	delay()
计算程序执行两个时刻之间的时间差,太长则给出相应处理	difftime()

(四)做好测试记录

对调试程序作一些实质性的记录既有必要,也有价值,因为它为编写文档作了准备。

(1)调试结果记录在源程序中

对小型程序可以用加注释的方法把调试结果记在源文件中,边调边记。因为调试时不影响原编辑功能,换句话说,调试时只要有暂停就可进行编辑!注意到这一点是很有意义的。

(2)利用输出格式函数输出,分析调试结果

除使用调试表达式观察调试结果外,也可以用格式输出函数将调试结果输出到屏幕或打印机。例如,在必要在地方,你可以加一条象 printf() 那样的语句,以便输出调试结果。

对图形程序大多靠直接输出结果来验证。

(3)利用 DOS 改向输出程序执行结果到磁盘

程序输出结果也可以用 DOS 的改向输出功能将结果输出到指定磁盘文件中。例如 C> WORDCNT>RESULT.DAT 便可将 WORDCNT.EXE 的执行结果存到磁盘文件 RESULT.DAT 中。然后在编辑窗内用 Ctrl-R 键操作将 RESULT.DAT 读入到当前编辑窗内。这一办法一般是可行的,但也有例外,如对程序输出结果为图形等一些程序无效。

注意:在某些汉字系统下,对有些程序改向输出与不改向输出可能结果不一样。

(五)辅助工具

(1)当对宏扩展结果发生怀疑时用 CPP.EXE。例如对

```
#define toupper(c) 'a'<=(c) && (c)<='z'? (c)-'a'-'A':(c)
char c,*p="bcD";
c=toupper(*p++);
```

用 CPP.EXE 对它进行宏扩展处理后就会发现问题。因每次取 *p 值后,指针 p 要作 p++ 运算。从扩展后的表达式中可以看出对每一个 c 都要作这种运算,因此原本只做一次的 p++

+ 运算在宏调用一次时结果要做多次了！这显然是错误的。

(2) 当对几个文件中一次性查找指定字符串时用 GREP.COM;

(3) 分析目标模块时用 OBJXREF.COM;

(4) 一次性想强迫重新编译或重建多个目标文件时, 使用 TOUCH.COM;

(5) 想预置集成环境缺省参数时用 TCINST.EXE;

(6) 修改库时可用 TLIB.EXE;

(7) 要将图形驱动程序文件或字体集装入到用户程序中去时用 BGI OBJ.EXE。

17.4 DOS 5.0 的调试程序 DEBUG.EXE

作为对二进制可执行文件专用调试程序, DEBUG.EXE 有其优越性的一面, 现简要列出其使用方法和 Turbo C 与它相似之处。在理解调试程序涉及的一些基本操作知识方面, 读者可以领略到它们的互补性。

(一) DEBUG 的启动

在 DOS 提示符下键入

DEBUG 调试文件名 [调试(可执行)文件名的参数]

文件名可以带有驱动器名、路径名。因一次只能对一个文件调试, 所以文件名中不应有通配符 * 或 ? 出现。

如果没有指定调试文件名及参数, 而直接在 DOS 提示符下键入 DEBUG, 如装载成功, 立即在下一行显示一个连字符 (-), 它是等待用户键入 DEBUG 命令的提示符。

(二) DEBUG 命令

在 DEBUG 命令提示符出现后, 先键入一个或两个命令名字符 (字母大小均可), 然后按下述三种形式中任一种输入参数:

命令名和第一个参数间无空格, 如 ACS:0100;

命令名和第一个参数间有一个空格, 如 A CS:0100;

命令名和第一个参数间有一个逗号, 如 A,CS:0100,110。

参数可能有多个, 多个参数间可用空格或逗号隔开, 回车后一个 DEBUG 命令便输入。

如果命令输入错, 则 DEBUG 发出 ^ Error 提示, ^ 指出错误位置。

—1 ? 命令 (查询命令)

功能: 查询 DEBUG 命令语法说明

语法: 查询命令, 屏幕显示所有 DEBUG 命令语法

参考: 在 Turbo C 集成环境中按 F1 键获得帮助。

—2 A 命令 (汇编命令)

功能: 汇编 (Assemble) 命令。它把 8086/8087/8088 汇编语言语句直接汇编到内存中。

语法: A[address]

说明:

1. address 表示地址参数, 它有以下几种形式, 内中数值均为 16 进制数, 数后面都不必加写表示 16 进制数的字母 H。

例如

指定开始地址

(1)CS:0100 段寄存器: 偏移量

(2)04BA:0100 段名:偏移量

(3)100 段名省略。对 G、L、T、U 和 W 命令默认段为 CS,其它为 DS。

指定地址范围

(4)04ba:0100 10f 从 04ba:100 到 04ba:10f

(5)CS:100 10F 指定段内范围,100 到 10F

(6)CS:100 L 10 指定开始地址和长度,L 后为要处理的字节数

2. 方括号表示括号内的内容是可以任选的。当没有地址时,从最近停止位置开始汇编。

3. 命令输入后就可输入汇编语句。每次输入一条汇编语句,按回车确认。

4. 段超越助记符可用 CS:、DS:、ES: 和 SS:;远程返回助记符用 retf;字符串操作助记符必须指定字符串长度,可用 movsw(传送字串,即 16 位)或 movsb(传送字节,即 8 位)。

5. 汇编转移和调用指令时可加 near 或 far。例如,键入三条指令

```
jmp 502 ;short 转移
```

```
jmp near 505 ;near 转移,此句也可写成 jmp ne 505,即 near 可缩写成 ne
```

```
jmp far 50a ;far 转移
```

6. 为区别操作数是指的一个字节存储单元或一个存储字单元,须用 word ptr 或 byte ptr, word ptr 与 byte ptr 可分别缩写为 wo 与 by。例如,下面两句是一样的

```
dec wo [si]
```

```
dec word ptr [si]
```

7. 为区分立即数和存储单元,存储单元要用方括号括起。例如

```
mov ax, 21 ;将立即数 21H 送入 AX 中
```

```
mov ax,[21] ;将存储单元 21H 中的内容装入 AX 中
```

8. 本命令允许使用 DB 和 DW 两条伪指令,如

```
db 1,2,"this"
```

```
db 'this ' "
```

```
db "this "
```

```
dw 1000,3000,"BATH"
```

9. 本命令也支持所有形式的间接寻址,如

```
add bx,34[bp+2],[si-1]
```

10. 对 8087 操作码,应加 wait 或 fwait 前缀,如

```
fwait fadd st,st(3)
```

参考:Turbo C 的嵌入汇编和 ——emit()—— 内部函数。

—3 C 命令(比较命令)

功能:比较(Compare)两个内存块的命令。

语法:c range address

说明:

1. range 指出第一个块的地址范围, address 指出第二个待比较块的起始地址。这两个块的大小是一样的。

2. 命令输入后,如两块内容相同则无任何显示,否则连续按

块 1 地址 块 1 内容 块 2 内容 块 2 地址

形式显示一字节比较地址中不同的内容。

参考:Turbo C 中库函数 memcmp() 和 memicmp()。

—4 D 命令(显示命令)

功能:显示 (Dump) 内存地址范围内的内容

语法:d[range]

参考: Turbo C 的 peek() 和 peekb() 函数。

通过一个初始化为 0 的远指针和使用 Turbo C 调试表达式可访问内存单元。

—5 E 命令 (修改命令)

功能:输入 (Enter) 而修改内存字节命令。

语法:e address[list]

说明:

1. 假定用了不带 list 选项的命令,即

```
C>DEBUG
```

```
-E 04BA:0100
```

后有显示

```
04BA:0100 EB.
```

面对字节值 EB 你应输入一个新的有效的 16 进制数,按空格键该字节即被修改,并进到修改下一个字节状态;如果你在输入数后又按了连字符(-),则该字节也被修改,但接着是退回到上一字节修改状态;如果你不想修改这个字节,则只要直接按空格键便进到下一步,或者按连字符退回到上一字节修改状态。

2. 任何时候按回车键退回到要求输入命令状态。

3. DEBUG 能自动识别你输入的是否是有效的 16 进制数或连字符,除此之外的数将拒绝接收。

4. 也可以在输入命令时加上选项 list。如

```
-E 04BA:0100 "THIS IS"
```

回车后,从 04BA:0100 开始的 7 个字节内容已自动修改为 THIS IS。或者

```
-E 04BA:0100 54 48 49 53 20 49 53
```

也一样。这就是说,list 是一个立即要修改的串。

参考: Turbo C 的 poke() 和 pokeb() 函数。

例如,键入调试表达式 *(char far *)0x00001234 后,可看到该内存单元中的内容,如在新值窗内键入新值即可改变此单元中的内容。

—6 F 命令 (填充命令)

功能:用 list 的值填充 (Fill) 指定区域 (range)

语法:f range list

说明:

1. 例如,

```
-F 04BA:100L100 41 42
```

则从 04BA:0100 到 04BA:01FF 的 100H 个单元被两个字母 AB 反复填充。

2. 如 list 字节数比 range 少,则用 list 反复填充;否则, list 多余字符被忽略。

参考: Turbo C 中库函数 memset() 和 setmem()。

—7 G 命令 (执行命令)

功能:执行 (Go) 当前内存中的程序

语法:g [=address][addresses]

说明:

1. 如果只打入一个 `g` 而没有后边的选项,则当前内存中的程序就被执行,如同直接对该程序执行一样。

```
C>TEST4.C
```

```
pk(int k)
```

```
{
```

```
while(k-->0)printf("%d,",k);
```

```
}
```

```
main()
```

```
{
```

```
pk(5);
```

```
printf("\n");
```

```
pk(2);
```

```
}
```

```
/* C>DEBUG TEST4.EXE
```

利用 DEBUG 操作

```
-G
```

```
4,3,2,1,0,
```

程序输出

```
1,0,
```

```
Program terminated normally
```

程序正常结束,此后不能恢复程序

```
-G
```

```
Abnormal program termination
```

退出 DEBUG 返回 DOS

```
C>
```

```
*/
```

2. `=address` 是开始执行地址。当上述没指定开始地址时实质上是从默认的起始地址即由当前的 `CS:IP` 寄存器中的值决定的。

3. `addresses` 是临时设置的断点地址。最多可设置 10 个断点,如果多于 10 个,将显示

■一条指令开始执行。注意:执行到的第一个断点未必一定是断点表中的第一个断点。

6. 用户堆栈指针必须是有效的,并且对 `g` 命令必须有 6 个字节可供使用。`g` 命令使用一条 `iret` 指令转移到被测试的程序,调整用户堆栈指针,将所有标志、码段寄存器和指令指针压入用户堆栈(如堆栈无效或太小,则会弄乱操作系统)。DEBUG 将一个中断码(0CCH)放入指定的断点地址(es)中。

参考: Turbo C 集成环境中 `Run Ctrl-F9` 命令和 `Ctrl-F8` 设置断点操作。

—8 H 命令(运算命令)

功能:求两个 16 进制数(Hex)的和与差

语法: `h value1 value2`

说明:

```
C>DEBUG
```

```
-H 19F 10A /* 16 进制数位不应超过 4 位 */
```

```
02A9 0095
```

```
-
```

显示的第一个数为两数之和,第二个数为两数之差。两数取值范围为 0~FFFFH。

在 Turbo C 中可以用 `Ctrl-F4` 键操作,分别输入调试表达式

```
0x19f+0x10a,x 和 0x19f-0x10a,x
```

后立即求得这两个数的和与差。

参考:使用 Turbo C 集成环境中的调试表达式。

—9 I 命令 (输入命令)

功能:从指定端口 (port) 输入 (Input) 一个字节值并显示

语法:i port

说明:

```
C>DEBUG
```

```
-i2f8          端口 2F8H (一个 16 位的值)
```

```
FF
```

```
-
```

参考: Turbo C 库函数 inportb()、inport() 和宏 inp()。

—10 L 命令 (装入命令)

功能:将一个磁盘文件或指定磁盘扇区的内容装入 (Load) 内存

语法:l[address] [drive] [firstsector] [number]

说明:

1. address 是装入文件或扇区内容存放的起始内存单元地址。如果没有指定该地址, DEBUG 用 CS 寄存器当前地址作存放地址。

2. drive

一个 16 进制数,表示驱动器号。0=A 驱动器,1=B 驱动器,... 等等。

3. firstsector 是磁盘上第一个扇区号, number 是连续装入扇区数。如果没有指定这两个参数,从磁盘文件装入的字节数由 BX: CX 寄存器中的值决定。如果绕过文件系统而直接从磁盘指定扇区装入,除必须带这两个参数外,其余两个参数也必须指定。例如,从 C 盘上逻辑扇区 15 (或 0FH) 开始的 109 (或 6DH) 个扇区装入到从 04BA:0100 开始的地址中的命令是

```
-L 04BA:100 2 0f 6d
```

4. 可用 C>DEBUG TEST4.C 即 DEBUG 后带文件名的方法直接装入文件到内存。也可以用

```
C>DEBUG
```

```
-n TEST4.C      使用 n 命令,但 TEST4.C 并未装入内存
```

```
-l              将 TEST4.C 装入内存
```

```
-
```

将文件装入内存。当用 g 命令使程序运行结束后,如要重新运行,可用此方法先装入后运行。

5. 文件装入内存后就可以用 r 命令观察寄存器中的值,这时可发现 BX: CX 中为装入字节数, IP 中为装入开始地址。对非 *.EXE 执行文件,装入字节数同原文件长度,起始地址一般为 CS:100; 而对 *.EXE 可执行文件装入长度一般不等于文件长度,因为装入后要重定位,起始地址为 0, 这时如指定了 address 参数,则该参数被忽略。但是,对同一 EXE 文件如将它的扩展名更改成不是 EXE 或 COM, 则用 DEBUG 装入后便可观察分析整个程序,包括更改其内容后存盘等操作均可进行。

6. 对 MS-DOS 大全 (The MS-DOS Encyclopedia) 描述的带扩展名为 .HEX 的 hex 文件,它是使用内部十六进制的文件也能为 DEBUG 接受。在装入这种文件时一般可以不打入任何参数,即只用一个 i 来装入,文件装入到由 hex 文件指定的地址中;如果你指定了一个起始地址,那么装入的起始地址为你指定的起始地址与 hex 文件所含地址之和。

参考: Turbo C 集成环境里的 File Load 命令。

—11 M 命令 (传送命令)

功能: 将一个内存块 (源块) 中的内容传送 (Move) 到另一个内存块 (目的块) 中

语法: m range address

说明:

1. range 指定要移动块的起始和结束地址 (或起始地址和块长度), address 是要 将块安放的新地址。

2. 根据 address 与 range 值的不同, 有可能发生目的块和源块重迭的现象, 但这不会 影响结果的正确性。

参考: Turbo C 中库函数 memcpy()、memmove() 和 memccpy()。

—12 N 命令 (命名命令)

功能: 为命令 i 或 w 指定 (Name) 文件名, 对执行文件还可同时指定参数

语法: n [pathname] [arglist]

说明:

1. 如果对程序 TEST5.C

```
C>TYPE TEST5.C
```

```
main(int argc,char *argv[])
```

```
{
```

```
if(argc==2)printf("%d = %s = %s\n",argc,argv[0],argv[1]);
```

```
}
```

进行编译连结产生执行文件 TEST5.EXE, 然后在 DOS 提示符下执行:

```
C>TEST5 WWW
```

则程序输出

```
2 = C:\TC\TEST5.EXE = WWW
```

现在利用 DEBUG 执行:

```
C>DEBUG
```

```
-N TEST5.EXE /* 记住: 执行文件名和参数一定要分开输 */
```

```
-L /* 如果用 -N TEST5.EXE WWW 则不能得到预定的结果 */
```

```
-N WWW /* 输参数 WWW */
```

```
-G
```

```
2 = TEST5.EXE = WWW /* 不会输出子目录名 */
```

```
Program terminated normally
```

```
-
```

2. 在使用 L 命令或 W 命令前应用一条 N 命令, 为它们指定文件名, 以防出错。

3. 在使用 n 命令后, 用 d 命令可观察 CS:5C、CS:6C、CS:80 和 CS:81 等处内容可能发生变化。

```
C>DEBUG
```

```
-N TEST5.C
```

```
-D CS:5C
```

```
/* 立即显示相关内容 */
```

```
-D
```

参考: Turbo C 集成环境命令 File/Load 和 Options/Arguments, 以及有关文件控制块部分。

—13 O (输出命令)

功能: 向指定端口输出 (Output) 一个字节值

语法: o port byte

说明:

参考: Turbo C 的库函数 outportb()、outport() 和宏 outp()。

—14 P 命令 (执行子程序等命令)

功能: 执行 (Proceed) 循环、复制串、软中断或子程序, 否则相当于 t 追踪命令。

语法: p [=address] [number]

说明:

1. =address 是指定执行的第一条指令。如果没有指定 address, 缺省的 address 是寄存器 CS:IP 中的值。

2. number 是将控制交给 DEBUG 前执行的指令数, 缺省值为 1。

3. 不能用此命令追踪只读存储器 (ROM) 中内容。

4. 观察调试程序 TEST6.exe

```
C>TETST6.C
```

```
main()
```

```
{
```

```
int k;
```

```
for(k=0;k<2;k++)printf("%d ",k);
```

```
}
```

```
C>DEBUG TEST6.EXE
```

```
-P
```

马上显示追踪一条指令后的内容 (寄存器、标志等)。和 t 命令不同, 它将循环指令 (象 SCAS) 等作为一条指令执行, 而不进行反复循环。从而比 t 命令减少了追踪次数。

参考: Turbo C 集成环境中 Rnu/Step over F8 单步追踪命令。

—15 Q 命令 (退出命令)

功能: 不将文件存盘而退出 DEBUG 后返回 MS-DOS。

语法: q

说明:

退出后显示 MS-DOS 提示符。

参考: Turbo C 集成环境中 Run/Program reset Ctrl-F2 结束调试命令。

—16 R 命令 (寄存器命令)

功能: 显示或修改一个或多个 CPU 寄存器 (Register) 的内容。

语法: r[register]

说明:

1. 看下面操作

```
C>DEBUG TEST6.EXE
```

```
-r
```

```
AX=0000 BX=0000 CX=22E8 DX=0000 SP=0080 BP=0000 SI=0000 DI=0000
```

```
DS=2D09 ES=2D09 SS=2E97 CS=2D19 IP=0000 NV UP EI PL NZ NA PO NC
```

2D19,0000 BA502E MOV DX,2E50

-rfg	查标志键入错误
-br Erroe	显示:键入了错误的寄存器
-rf	查标志, F 是 flag 即标志寄存器
NV UP EI PL NZ NA PO NC—pleicy	键入的 pleicy 三个有效的标志代码
-rf	标志位 PL、EI 未变,NC 变成 CY
NV UP EI PL NZ NA PO CY—ffg	键入的 ffg 是一个无效的标志代码
bf Error	显示:键入了错误的标志
-rf	
NV UP EI PL NZ NA PO CY—PL PL	键入两个一样的标志代码
df Error	显示:键入了错误的标志
-rss	查寄存器 SS
SS 2E97	显示寄存器内容
: 2E97	冒号后可键入新寄存器值
—	凡不修改可直接按回车键

2. 标志识别代码

标志位名称	置位 (1)	复位 (0)
溢出 (Overflow)	OV (是)	NV (否)
方向 (Direction)	DN (减少)	UP (增加)
中断 (Interrupt)	EI (允许)	DI (不允许)
符号 (Sign)	NG (负)	PL (正)
零 (Zero)	ZR (是零)	NZ (非零)
辅助进位 (Auxiliary Carry)	AC (有)	NA (无)
奇偶 (Parity)	PE (偶)	PO (奇)
进位 (Carry)	CY (有)	NC (无)

参考:在 Turbo C 中使用伪变量作调试表达式。

—17 S 命令 (搜索命令)

功能:在指定地址范围内搜索 (Search) 具有指定字节值的所有存储单元地址

语法:s range list

说明:

1. range 是要搜索的范围, list 是一个或多个字节值 (多个字节值之间要用空格或逗号隔开),也可以是一个字符串 (必须用西文双引号括起)。例如,

■ C>DEBUG TEST6.EXE

—S2d09,100 110 41

2D09:0104 找到 2 个

2D09:010E

—scs:100 1a0 "ph" 未找到

—

2. 当 list 是由多个字节值构成时,应把这些字节看成是一些连续的字节,假定这些连续的字节值都是某些字符的 ASCII 值,则这些字节就是字符串。因此,如找到这些连续的字节值,只显示一个地址。

参考: Turbo C 中 Debug/Find function 命令及编辑用命令 Ctrl-Q-F (寻找指定串)和寻找并替换命令 Ctrl-Q-A。

—18 T 命令 (跟踪命令)

功能: 执行一条指令, 并显示所有寄存器、标志位及一条指令的机器码、汇编语句等
语法: t[=address] [value]

说明:

1. =address 指出开始跟踪执行的起始地址, 如果未使用它和 value, 则从 CS:IP 指定值开始跟踪, value 是指定跟踪的指令数目, 是一个 16 进制数, 缺省值为 1。
2. 当指定的 value 大于 1 时, 显示是连续进行的, 要暂停可按 Ctrl-Break 键。
3. 可以跟踪存储在只读存储器 (ROM) 中的指令。

参考: Turbo C 的 Run/Trace into F7 跟踪进入函数内部命令。

—19 U 命令 (反汇编命令)

功能: 对字节进行反汇编, 并显示与之对应的、带有地址和字节值的源语句

语法: u[range]

说明:

如果不使用 range 参数, 则从上一条 U 命令显示之后的第一个字节开始反汇编出 20H 个字节, 否则将反汇编由 range 指定范围内的全部字节。当地址中某些内容用命令修改后 (例如用了 A 命令), 可用此命令观察修改情况。例如, 对 TEST6.EXE 反汇编, 有

```
C>DEBUG TEST6.EXE
```

```
—U
```

```
2D19:0000 BA502E      MOV     DX,2E50
2D19:0003 2E         CS:
2D19:0004 8916F801     MOV     [01F8],DX
2D19:0008 B430      MOV     AH,30
2D19:000A CD21      INT     21
2D19:000C 8B2E0200     MOV     BP,[0002]
2D19:0010 8B1E2C00     MOV     BX,[002C]
2D19:0014 8EDA      MOV     DS,DX
2D19:0016 A39200     MOV     [0092],AX
2D19:0019 8C069000     MOV     [0090],ES
2D19:001D 891E8C00     MOV     [008C],BX
—UCS:21 2D
2D19:0021 892EAC00     MOV     [00AC],BP
2D19:0025 C7069600FFFF     MOV     WORD PTR[0096],FFFF
2D19:002B EB3401     CALL    0162
```

参考: 使用 TCC.EXE 生成汇编文件。例如对 test6.c 要生成 TEST6.ASM 可用

```
C>TCC -S TEST6.C
```

```
C>TYPE TEST6.ASM
```

```
        ifndef      ?? version
? debug  macro
        endm
        endif
        ? debug      S "test6.c"
—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 E94382551B056D33342E63
—BSS      ends
—TEXT     segment byte public 'CODE'
,         ? debug      L 1
—main     proc          near
          push         si
,         ? debug      L 4
          xor          si,si
          jmp          short @5
@4:
,         ? debug      L 4
          push         si
          mov          ax,offset DGROUP,s@
          push         ax
          call         near ptr —printf
          pop          cx
          pop          cx
@3:
          inc          si
@5:
          cmp          si,2
          jl           @4
@2:
@1:
,         ? debug      L 5
          pop          si
          ret
—main     endp
—TEXT     ends
          ? debug      C E9
—DATA     segment word public 'DATA'
s@        label        byte
          db           37
          db           100
          db           32
          db           0

```



```

- DATA    ends
--TEXT     segment byte public 'CODE'
           extrn      _printf: near
           -TEXT      ends
           public     _main
           end

```

—20 W 命令 (写盘命令)

功能: 写 (Write) 一个文件或指定扇区到磁盘

语法: w [address] [drive] [firstsector] [number]

说明:

1. address 是已装入内存的文件或文件的一部分要写入磁盘的起始内存地址。如果没有指定它, 则默认从 CS:100 开始。

2. drive 是要记录文件写入内容的驱动器号, 0=A 驱动器, 1=B 驱动器, ... 等等。

3. firstsector 是磁盘上要记录写入文件的第一个绝对扇区号, number 是要连续写入的扇区数, 它们都是 16 进制数。向绝对扇区写应十分小心。

4. 如果未用任何参数, 在 BX;CX 寄存器中必须设置好将要写入的字节数。注意, 在使用了 g、t、p 或 r 命令后, 很可能要重新设置 BX;CX 中的值。

5. 当装入内存的文件内容被修改后, 但从装入文件到现在并未改变文件的长度、名字和装入的起始地址, 那末 DEBUG 能将文件正确写到磁盘上。

6. 对 .EXE 和 .HEX 文件不能使用本命令。

7. 使用本命令前必要时应使用 N 命令。

参考: Turbo C 中 File/Save F2 命令或 File/Write to 命令。

—21 XA 命令 (分配命令)

功能: 分配扩页内存 (Allocate Expanded Memory) 指定页数

语法: xa[# pages]

说明:

指定分配扩页内存页数 (每页 16KB)。例如分配 8 页扩页内存 C>XA8

```

- XA8
Handle created=0003          分配成功后的显示

```

—22 XD 命令 (重分配命令)

功能: 重新分配扩页内存 (Reallocate Expanded Memory)

语法: xd[handle]

说明:

1. 使用本命令时, 机器上应装有扩页内存驱动程序 (LIMEMS 4.0 版), 否则使用此命令时会显示 EMS not installed 信息。

2. handle 是要重新分配的句柄数。如 C>DEBUG

```

- xd 0003
Handle 0003 dellocated      显示内容

```

—23 XM 命令 (变换命令)

功能: 变换指定句柄的扩页内存逻辑页 (Map Expanded Memory Pages) 为物理页

语法: xm[l.page] [Ppage] [Handle]

说明:

1. Lpage 是想变换成物理页的逻辑页数。
2. Ppage 是物理页数,它用于安排要变换的逻辑页。
3. Handle 是指定句柄数。
4. 将句柄 3 的逻辑页 5 变换成物理页 2 的操作

C>DEBUG

-XM 5 2 0003

Loaical page 05 mapped to physical page 02 显示

-

5. 机器上应装有扩页内存驱动程序 (LIMEMS 4.0 版)。

—24 XS 命令 (显示扩页内存命令)

功能:显示扩页内存状态信息 (Display Expanded—Memory Status)

语法:xs

说明:

例如,显示信息如下:

Handle 0000 has 0000 pages allocated

Handle 0001 has 0002 pages allocated

Physical page 00 = Frame segment C000

Physical page 01 = Frame segment C400

Physical page 02 = Frame segment C800

Physical page 03 = Frame segment CC00

2 of a total 80 EMS pages have been allocated

2 of a total FF EMS handles have been allocated

(三) DEBUG 汉化方法

为使 DEBUG.EXE 执行时能显示汉化信息,可用如下操作对其修改。

C>COPY DEBUG.EXE debugc

C>DEBUG debugc

-E677 7F.FF

-E881 3C.90 7F.90 73.90 04.90

-W

Writing 0509A bytes

-Q

C>DEL DEBUG.EXE

C>REN debugc DEBUG.EXE

现在使用 DEBUG.EXE 就可显示中文信息了。

原
书
缺
页

425-436


```

)/* 如果文件 ww.c 不存在,程序输出:
   文件不存在: No such file or directory
   2      2
   No such file or directory      */

```

18.2 库函数

—1 int —Cdecl doserror(struct DOSERROR *eblkp);

将最近一次调用 DOS 功能失败后的扩展错误信息填入 eblkp 所指的 DOSERROR 结构中。函数返回 exterror(当它为 0 时表示最近一次 DOS 调用未出错)。注意:此结构通过 DOS 功能调用 59H 得到。入口参数是, AH=0x59, BX=0x0。AX 中返回 exterror, BH 中返回 class, BL 中返回 action, CH 中返回 locus。它只适用于 MS-DOS 3.X 及以上版本,对低于它的版本不能用。

DOSERROR 结构在 dos.h 中有定义:

```

struct DOSERROR {
    int exterror; /* 扩展错误代码 */
    char class;   /* 错误种类      */
    char action;  /* 推荐行动      */
    char locus;   /* 错误地点      */
};

```

结构成员详细说明:

1. exterror

值	含义	值	含义
00H	无错误	10H	试图取消当前目录
01H	无效功能号	11H	非同一设备
02H	文件未找到	12H	不再有文件
03H	路径未找到	13H	磁盘写保护
04H	打开文件太多(无可用文件句柄)	14H	不认识的装置
05H	拒绝访问	15H	驱动器未就绪
06H	无效文件句柄	16H	不认识的命令
07H	内存控制块被破坏	17H	数据错
08H	内存不够	18H	错误的请求结构长度
09H	内存块地址无效	19H	寻找错
0AH	无效环境(长度 > 32K)	1AH	非 DOS 磁盘
0BH	无效格式	1BH	扇区未找到
0CH	无效访问码	1CH	打印机无纸
0DH	无效数据	1DH	写故障
0EH	保留	1EH	读故障
0FH	无效驱动器	1FH	一般错误

2. class

值	含义	值	含义
01H	无存储空间或 I/O 通道	08H	未找到

02H 文件或记录锁定	09H 错误格式
03H 被拒绝的访问	0AH 锁定
04H 系统软件故障	0BH 介质故障
05H 硬件故障	0CH 已存在
06H 配置文件丢失或错误	0DH 未知
07H 应用程序错误	

3. action

值	含义	值	含义
01H	再试	05H	立即放弃
02H	再试	06H	忽略
03H	提醒用户重新输入	07H	用户干涉后再试
04H	清除后放弃		

4. locus

值	含义	值	含义
01H	不详或不合适	04H	串行设备(超时)
02H	块设备(磁盘错误)	05H	与内存有关
03H	与网络有关		

下面是在 MS-DOS 5.0 下测试的结果。

```
C>TYPE DOSERR2.C
#include "dos.h"
#include "fcntl.h"
#define VIEW h=dosexterr(&d);\
    printf("h=%d exterror=%d class=%u action=%u locus=%u\n",h,\
        d.exterror,d.class,d.action,d.locus)
#define IFELSE if(handle== -1)printf("Error ! \n");\
    else printf("OK! \n");fclose(handle);VIEW;

main()
{
    int handle,fp,h;
    struct DOSERROR d;
    handle=open("L78.c",O—RDWR);
    IFELSE;
    handle=open("L79.c",O—RDWR);
    IFELSE;
    handle=open("tt.c",O—RDWR);
    IFELSE;
    chsize(handle,1000); /* 改变文件大小 */
    VIEW;
    handle=open("L79.c",O—RDWR);
    IFELSE;
}
/* 如果文件 L78.C 和 L79.C 存在,而文件 tt.c 不存在,在 DOS 5.0 下输出:
    OK!
```



```

h=8 exterror=8 class=1 action=4 locus=2
OK!
h=8 exterror=8 class=1 action=4 locus=2
Error !
h=2 exterror=2 class=8 action=3 locus=2
h=6 exterror=6 class=7 action=4 locus=1
OK!
h=6 exterror=6 class=7 action=4 locus=2
*/

```

附注:中断 INT 24H (严重错误处理例程)

1. 功能:每当遇到严重错误(critical error,通常由硬件引起)发生时它才被 DOS 激发。

注意:它不能被应用程序直接调用。

2. 调用过程:当它被调用时,DOS 将有关寄存器和堆栈置值:

(1) AH = 发生错误的类型和处理标志

位 7 1 = 若为块设备,为坏的 FAT 内存映象;若为字符设备,错误代码在 DI 中
 0 = 磁盘 I/O 错

位 6 未使用

位 5 1 = 若允许忽略 0 = 不允许忽略

位 4 1 = 若允许再试 0 = 不允许再试

位 3 1 = 若允许失败 0 = 不允许

位 2、位 1 磁盘错误区

 00H DOS 区

 01H FAT 区

 10H 根目录区

 11H 数据区

位 0 1 = 读 0 = 写

(2) AL = 若 AH 的位 7 = 0 (复位),则为驱动器号

(3) BP:SI 指向设备驱动程序标题(参见《文件管理》一章 ioctl() 函数)。

(4) 对字符设备,错误代码在 DI 中:

代码	意义	代码	意义
00H	试图破坏写保护	08H	扇区未找到
01H	驱动程序不详单元	09H	打印机无纸
02H	驱动器未准备好	0AH	写错误
03H	驱动程序的命令不详	0BH	读错误
04H	数据错(坏 CRC)	0CH	一般错误
05H	坏的设备驱动程序,请求结构长度	0DH	共享错误
06H	寻道错	0EH	锁定错误
07H	介质类型不详	0FH	无效磁盘更换

(5) 堆栈

双字 INT 24H 调用的返回标志

字 INT 24H 压入的标志

字 INT 24H 的入口的原始 AX

字 BX

字 CX
 字 DX
 字 SI
 字 DI
 字 BP
 字 DS
 字 ES
 双字 INT 21H 调用的返回地址
 字 INT 21H 压入的标志

3. 说明:该处理例程能进行的 DOS 调用仅为 INT 21H 功能 01H ~ 0CH 和 59H。如果处理例程采用弹出堆栈的方法返回到应用,则 DOS 将处于一种不稳定状态,直到用 AH>0CH 进行的第一次调用为止。

4. 处理例程必须返回行动代码于 AL 中:

当 AL = 00H 表示忽略错误并继续处理请求
 01H 表示再试操作
 02H 表示通过中断 INT 23H 终止程序
 03H 表示调用过程中系统调用失败
 SS、SP、DS、ES、BX、CX 和 DX 被保留。

在 DOS 3.1+(这里加号表示等于或高于的意思),对于网络严重错误,忽略(AL=00H)转成失败(AL=03H)。如果指定了忽略但不允许,则它也转入失败。如果指定了再试但不允许,它也转入失败。如果指定了失败但不允许,则它将转入放弃。对 DOS+,如果严重错误发生于严重错误例程内部,则 DOS 调用自动失败。

—2 void —Cdecl harderr(int —Cdecl (* handler)());

本函数为当前程序建立一个硬件错误处理程序。

执行当前程序时,每当发生严重错误 DOS 便调用中断 INT 24H,harderr() 允许此时去执行用户自己编写的 INT 24H 中断服务例程 handler(例程名字),而不是去执行缺省的 INT 24H 中断服务例程(如果用户未编写自己的 INT 24H 例程,则 DOS 将自动执行它)。这实际上是修改了缺省中断 INT 24H 的地址为用户的 INT 24H 例程地址。由于只有当有严重错误发生时 INT 24H 中断才被调用,因此 handler() 函数也只有在有严重错误发生时才被执行,否则它不会被调用,尽管程序中有调用它的语句存在。

在程序中,handler 例程以函数形式(函数名不必一定取 handler,例如在下面的 DOSERR3.C 中用了 my—int24)出现,

handler(unsigned errval,unsigned ax,unsigned bp,unsigned si){}

该函数中的参数具有特定的含义(参见中断 INT 24H):

errval 是 MS-DOS 置于 DI 寄存器的出错代码。

ax 是 MS-DOS 置于 AX 寄存器的值。对于磁盘错误,可由

出错驱动器号 = ax & 0x00FF

求得出错驱动器号(1 = A 驱动器,2 = B 驱动器等)。

bp 和 si 是 MS-DOS 置于 BP 和 SI 寄存器的值。它们一起指向出错驱动器的设备驱动程序标题。bp 为段地址,si 为偏移量,可用 peek() 或 peekb() 从驱动程序标题检索设备信息。注意,驱动程序标题不可用 poke() 或 pokeb() 去改变。

在 handler() 中使用 bdos() 函数调用时,只能使用 DOS 中断 INT 21H 的功能调用 1H

~ 0CH 和 59H,任何其它的 bdos() 调用将使 MS-DOS 崩溃(摧毁 DOS 工作堆栈,使系统陷入一个不可预测状态),特别不能使用 C 标准 I/O 函数和 UNIX 仿真 I/O。

在 hardler() 中可以调用 hardren() 而直接返回到调用 hardier() 的应用程序。

在 hardler() 中也可以调用 hardresume() 而返回到 MS-DOS 子程序,该子程序通过 harderr() 再处理严重错误(即在 hardler() 中又使用 harderr(), harderr() 又调用另一个 hardler(),如此等等)。

在 hardler() 中也可以用 return(2) 或调用 exit() 函数而直接返回到 MS-DOS(abort)。当返回 0 表示忽略错误并继续处理请求(ignore);返回 1 表示操作重试(retry)。

它保留 SS、SP、DS、ES、BX、CX 和 DX 的值。

只适用于 MS-DOS。

C>TYPE DOSERR3.C

```
#include "DOS.H"
#include "process.h"
#define BPSI value=peek(bp,si-);\
    printf("bp:si=0x%x:0x%x value=0x%x\n",bp,si-,value)
/* int cbreak()
{
    printf("如果在 my-int24()中用了 harderr(),当发生严重错误时印出本串\n");
    printf("或用 ctrlbrk()并在程序执行后迅速按了 CTRL+BREAK 键,本串印出\n");
    fopen("k.c","r");
    exit(0);
}
*/
/* 如果文件 k.c 存在,则不用 exit(),cbreak() 调用后将直接返回 main() 中并执行语
   句 fopen("A:MY.C","r") 的下一语句 printf("=====\n");,程序 正常结束
   */
```

```
int my-int24(unsigned errval,unsigned ax,unsigned bp,unsigned si)
{
    /* my-int24() 是用户自编的 INT 24H 中断服务处理程序 */
    int axret=-AX; /* 变量 axret 意为在执行 hardresume() 时先使 -AX=axret */
    int value,si-=si;
    /* harderr(cbrcak); 如果有此句,则不管 axret 值如何,将去执行 cbreak() */
    /* ctrlbrk(cbrcak); 设置用户自己的 Ctrl-Break 处理子程序 */
    printf("errval=0x%x ax=0x%x bp=0x%x si=0x%x\n",errval,ax,bp,si);
    printf("axret=%d\n",axret);
    /* 发生严重错误时 bp:si 指向设备驱动程序的设备标题 */
    BPSI; /* 印出下一个设备标题段,4 个字节 */
    si-=2;
    BPSI;
    si-=2;
    BPSI; /* 印出设备属性,2 个字节 */
    si-=2;
    BPSI; /* 印出设备策略指针,2 字节 */
    si-=2;
    BPSI; /* 印出设备驱动程序中断指针,2 个字节 */
}
```

```

si--+=2;
BPSI; /* 印出设备名,8 个字节 */
si--+=2;
BPSI;
si--+=2;
BPSI;
si--+=2;
BPSI;
/*
printf("请在 A 驱动器中插入软盘后按任一键\n");
getch(); /* 如加上这两行会死机。getch() 将对 DOS 功能调用 */
hardresume(axret); /* 如保留这行且 axret=0 或 1,将不断读 A 盘 */
/* 如用 return(2); 正常终止程序。不能用 return(0) 或 return(1) */
/* 如用 abort(); 结束,则出现 Abnormal program termination */
}
main()
{
harderr(my--int24);
printf("A 驱动器中不要插盘! 按任一键开始\n");
getch();
fopen("A:MY.C","r");
printf("=====\n");
}
/* 输出:bp;
errval=0x2 ax=0x1a00 bp=0x70 si=0x6b
axret=2
bp,si--=0x70:6b value=0x7b 第 1 区段:7b 00 70 00
bp,si--=0x70:6d value=0x70
bp,si--=0x70:6f value=0x8c2 第 2 区段:c2 08
bp,si--=0x70:71 value=0x6f5 第 3 区段:f5 06
bp,si--=0x70:73 value=0x73e 第 4 区段:3e 07
bp,si--=0x70:75 value=0xff03 第 5 区段:03 ff 01 00 00 00 8d 00
bp,si--=0x70:77 value=0x1
bp,si--=0x70:79 value=0x0
bp,si--=0x70:7b value=0x8d
*/

```

注意:在集成环境下用 F7 键调试时,光标不会进入用户定义函数 my--int24(),但可以发现该函数将被执行。

—3 void —Cdecl hardresume(int axret);

它在用户自编的 INT 24H 中断处理函数 hardler() 中作为硬件错误处理函数,axret 是其返回的 MS-DOS 码(参见中断 INT 24H 的返回),值为 0 表示忽略该错误并继续处理请求(ignore),为 1 表示将该操作重试一次(retry),为 2 表示调用 INT 23H 中断结束程序运行(abort)。若为 3 表示过程中系统调用失败。当执行 hardresume() 时如果又发生了严重错误,函数根据 axret 值还可转去执行 hardler() 中设置的另一个 harderr() 处理函数。如果此

时在 `hardler()` 中无 `harderr()` 则当 `axret` 为 0 或 1 时将产生不可预料的结果。如果没有错误发生,则函数返回调用到它的 `harderr()` 语句的下一语句并执行该语句。

它只能在被 `harderr()` 调用的用户 INT 24H 中断处理函数中出现,将它直接用于非 INT 24H 中断处理函数中可能产生不可预测的结果。

注意:它在程序中的位置应在严重错误发生处之前。

C>TYPE DOSERR4.C

```
#include "dos.h"
#define PAX a-x=-AX;printf("a-x=%d\n",a-x)
hardler(unsigned ax)
{
    int retn;
    printf("-AX=%d\n",ax);
    retn=2;          /* 当值为 -1 或 2 时程序正常终止,为 0 或 1 死机 */
    hardresume(retn);
    printf("end\n");  /* 此句将永不被执行 */
}
main()
{
    int a-x;
    /* harderr(hardler); */ /* 如将下面同样两句提到此,则输出 a-x=9508 */
    PAX;                /* -AX=2 程序正常终止 */
    fopen("a:k.c","r"); /* 假定 A 驱动器中未装磁盘 */
    PAX;
    harderr(hardler);    /* 将此句放到发生严重错误的后面,效果不一样了 */
    PAX;
}
/* 屏幕显示:Not ready reading drive A
   Abor,Retry,Fail?
   如键 r, 则又重新显示;键 f,输出:
   a-x=0
   a-x=9508
   后程序终止;如键 a, 程序也正常终止。
*/
```

-4 void cdecl hardretn(int retn);

它在被 `harderr()` 函数调用的用户自编的 INT 24H 中断函数 `hardler()` 中出现,和 `hardresume()` 不同的是,在被调用后,直接返回调用它所在函数的 `harderr()` 的下一句并开始继续执行。调用函数时包括取校验标志、恢复堆栈、置 `-AX=retn` 及进位标志等。在 `DOSERR3.C` 中,如果只将 `hardresume()` 改成 `hardretn()`,则程序最后打印出

=====

`retn` 是一个用于返回到用户程序的值(参见中断 INT 24H 的返回),它代替产生严重错误时从 MS-DOS 功能调用产生的标准值。

C>TYPE DOSERR5.C

```
#include "dos.h"
```

```

#define PAX a—x=—AX,printf("a—x=%d\n",a—x)
harderr(unsigned ax)
{
int retn;
printf("—AX=%d\n",ax);
retn=-1;
hardretn(retn);
printf("end\n"); /* 此句将永不被执行 */
}
main()
{
int a—x;
harderr(harderr);
PAX;
fopen("a,k.c","r"); /* 假定 A 驱动器中未装磁盘 */
PAX;
}
/* 输出, a—x=9508
      —AX=2
      a—x=0
*/

```

C>TYPE DOSERR6.C

```

#include "stdio.h"
#include <dos.h>
#define DISPLAY—STRING 0x09
#define Ignore 0
#define Retry 1
#define Abort 2
int handler (int errval, int ax, int bp, int si)
{
char msg[25];
int drive;
if(ax<0) /* 驱动器错误 */
{
/* 只能用 DOS 功能 0x0~0x0C */
bdosptr(DISPLAY—STRING,"device error $",0); /* $ 符号不能少 */
hardretn(-1); /* 返回调用程序 */
}
drive=(ax & 0xFF);
sprintf(msg,"disk error on drive %c $", 'A'+drive);
bdosptr(DISPLAY—STRING,msg,0);
return(Abort); /* 终止调用程序 */
}
main()
{
harderr(handler); /* handler 是前面的函数名 */
printf("确信在驱动器 A 里没有插磁盘\n");

```

```

printf("击任一键.....\n");
getch();
printf("准备打开 A 驱动器里的文件\n");
fopen("A:\GET.C", "rb");
printf("结束\n");
}

```

—5 void cdecl perror (const char *s)

s 是程序员在源程序中给定的一个字符串（必须小于 95 个字符），其内容由程序员决定，以帮助指出出错信息的较详细的说明。s 既可以象下面 DOSERR7.C 中那样用数组变量 a，也可以放在命令行中（不过主函数要改成

```
main(int argc, char *argv[])
```

并将标识符 a 换成 argv。然后在 DOS 提示符下键入 C>DOSERR7 error!

便可获得同样的结果。源程序中可加入将变量 argc 作为判别是否键入两个参数的语句。

调用这一函数后，标准错误输出设备（stderr，一般指屏幕，但也可重定向）显示如下结果：

s 的内容：与查出的错误号（errno）对应的错误信息字符串

错误号 errno 是在执行程序时发现错误后操作系统对调用相关子程序（函数）返回的值。它在标头文件 errno.h 中被定为

```
extern int cdecl errno;
```

即为外部变量。要用本函数应包含此标头文件。

与本函数相关的还有一个数组 sys—errlist[]，（errno 可作为数组的下标以查找对应错误的字符串。该字符串不包括换行符。参见下列程序），它包括了错误信息字符串。它已由 TC 设定，用户无须更改。下面的函数定义只是帮助理解，在实际调用函数时只须将包含函数原型的标头文件写入源程序中即可。

在标头文件 errno.h 中将外部变量 sys—nerr 定义为 35，下面定义的变量 sys—errno 最大为 36，因指针数组变量 sys—errlist 占 72 字节，每个元素（指针）占 2 个字节。

下面出错信息字符串（串中并未包括换行符）后面注释的是对应的 errno，errno.h 中定义的常量（可以作为错误信息的助记符），以及涉及的函数。当出错时，这些函数将置 errno 为相应值。没有对应函数的 errno 是 MS-DOS 系统调用后返回 AX 的寄存器值。errno.h 中还列出了对应于 UNIX 的一些常量助记符（errno.h 中把它们暂时定义为 -1）。

它适用于 UNIX 系统。

```
C>TYPE DOSERR7.C
```

```

#include "stdio.h"
#include "errno.h"
#include "io.h"
#include "fcntl.h"
char *sys—errlist[]={
"Error 0", /* 0—EZERO */
/* 错误 0,表示没有错误发生. */
"Invalid function number", /* 1—EINVFNC */
/* 无效功能号。getftime,setftime, */

```

```

"No such file or directory", /* 2—ENOENT, 2—ENOFILE */
/* 路径或文件未找到。access, chdir, chmod, creat, exxec..., */
/* findfirst, findnext, mkdir, open, rename, spawn..., stat, */
/* unlink */
"Path not found", /* 3—ENOPATH */
/* 路径没找到 */
"Too many open files", /* 4—EMFILE */
/* 打开文件太多。creat, dup, dup2, exec..., open */
"Permission denied", /* 5—EACCES */
/* 无此存取权限。access, chmod, chsize, creat, exec..., mkdir */
/* open, read, rename, unlink, write */
"Bad file number", /* 6—EBADF */
/* 无效文件号。chsize, close, dup, dup2, eof, filelength */
/* getftime, setftime, ioctl, lseek, tell, read, stat, write */
"Memory arena trashed", /* 7—ECONTR */
/* 内存块破坏 */
"Not enough memory", /* 8—ENOMEM */
/* 无足够存储空间。freemem, allocmem, brk, sbrk, exec... */
/* getcwd, spawn... */
"Invalid memory block address", /* 9—EINVMEM */
/* 无效内存块地址 */
"Invalid environment", /* 10—EINVENV */
/* 无效环境 */
"Invalid format", /* 11—EINVFMT */
/* 无效格式 */
"Invalid access code", /* 12—EINVACC */
/* 无效存取码。open */
"Invalid data", /* 13—EINVDAT */
/* 无效数据。ioctl */
0, /* 返回(null) */
"No such device", /* 15—ENODEV, 15—EINVDRV */
/* 无此设备。getcwd */
"Attempted to remove current directory", /* 16—ECURDIR */
/* 试图删除当前目录 */
"Not same device", /* 17—ENOTSAM */
/* 不是同一设备。rename */
"No more files", /* 18—ENMFILE */
/* 没有更多文件。findfirst, findnext */
"Invalid argument", /* 19—EINVAL */
/* 无效参数。ioctl, lseek, spawn... */
"Arg list too big", /* 20—E2BIG */
/* 参数表太长。exec..., spawn... */
"Exec format error", /* 21—ENOEXEC */
/* 执行格式错误。exec..., spawn... */
"Cross—device link", /* 22—EXDEV */
0,0,0,0,0,

```



```

0,0,0,0,0,
*Math argument
*, /* 33—EDOM */
/* 数学定义域错误。log,pow,sqrt,asin,acos */
"Result too large", /* 34—ERANGE */
/* 结果太大。cabs,exp,pow,getcwd,—matherr,tan */
"File already exists" /* 35—EEXIST */
/* 文件早已存在。 */
},
int sys_nerr=sizeof sys_errlist / sizeof sys_errlist[0];
void perror(const char *s)
{
    char *cp;
    if(errno<sys_nerr && errno>=0) /* 参数 errno 不由程序员指定 */
        cp=sys_errlist[errno]; /* 运行相关程序出错时获得 */
    else cp="Unknown error";
    fprintf(stderr,"%s: %s\n",s,cp);
}
#include "string.h"
char *strerror (int errnum) /* 参数 errnum 由程序员指定,它可 */
{
    char *cp; /* 以和运行实际出错的原因不匹配 */
    int len;
    if(errnum<sys_nerr && errnum>=0)
        cp=sys_errlist[errnum];
    else cp="Unknown error";
    len=strlen(cp);
    cp[len++]='\n'; /* 先有换行符,后有终止符(空字符) */
    cp[len]='\0';
    return cp;
}
main(){
    char a[]="error!";
    char *per=(char *)malloc(100);
    int handle;
    per=strerror(35);
    printf("per=%s",per);
    handle=open("c:pp",O—RDONLY); /* 假设 c 当前目录上无文件 pp */
    perror(a); /* 该函数中有输出语句 */
    per=strerror(2);
    printf("per=%s",per);
    printf("handle=%d\n",handle); /* 出错时函数 open 返回 -1 的同时 */
    /* 置 errno 为 2 */
    per=strerror(a); /* 该函数效果和 perror 相同 */
    printf("per=%s",per);
    getch();
}
/* 程序实际输出结果为

```

```

per=File already exists
error!: No such file of directory
per=No such file of directory
handle=-1
per=error!: No such file of directory
*/

```

```

—6 char * —Cdecl strerror(int errnum);

```

它返回一个与错误号 `errnum` (0 ~ 35) 相对应的错误信息字符串的指针。字符串最后为换行符和空字符。它不象 `perror` 函数一调用便在屏幕上打出错误信息等。其参数 (`errnum`) 值由程序员自己指定。一般在调试程序时用它, 其输出可用调试表达式观察。`perror()` 函数常在程序执行中进行。

```

—7 char * —Cdecl —strerror(const char * s);

```

其使用方法同 `perror()` 函数, 但在效果方面只有一点不同, 它只返回象 `perror()` 向屏幕输出的字符串指针, 而不是直接向屏幕输出。

允许用户生成自己定义的错误信息。如果 `s` 为 `NULL`, 返回指向最近产生的系统错误信息。该字符串以空字符 (`'\0'`) 终止。如果 `s` 不是 `NULL`, 返回包含 `s` (或用户定义的错误信息) 与一个冒号、一个空格和最近产生的系统错误信息及一个换行符。`s` 的长度应等于或小于 94 个字符。

错误信息串被放在一个静态缓冲区中, 每次调用 `perror()` 时都被重写。

它适用于 NUIX 系统。

第十九章 硬盘的体系结构和主引导程序

硬盘由于存储容量大（目前有的硬盘的容量已达数百兆）、存取速度快而为用户喜爱。因为硬盘的品种很多，发展很快，所以下面只介绍与它有关的一般性内容。

为了使用不同的操作系统分享硬盘，可以用 DOS 的 FDISK 程序把硬盘从逻辑上划分为多个【分区】(Partition)，这可用 FDISK.EXE 进行。

MS-DOS 5.0 的 FDISK.EXE 程序具有这样一些功能：

1. 建立主 MS-DOS 分区(称【PRI DOS 分区】，primary MS-DOS partition)；
2. 建立扩展 MS-DOS 分区(称【EXT DOS 分区】，extended MS-DOS partition)；
3. 选择活动分区(active partition，或称【可自举分区】、【引导分区】)；
4. 删除分区；
5. 显示分区数据；
6. 对另一硬盘进行分区操作。

在建立新分区时便可选择分区尺寸（分区最大尺寸为 2 千兆字节）。PRI - DOS 分区必须有且只能有一个，一般应在第一个硬盘 C 上，且应有三个 MS-DOS 系统文件：IO.SYS、MSDOS.SYS 和 COMMAND.COM。为此，在建立分区后先用 DOS 的 FORMAT /S 命令对它格式化，然后将它变成活动分区。此后可建立一个 EXT - DOS 分区，再在这个分区内建立多个逻辑驱动器（为 D:、E:、F:、G: 等盘。根据硬盘容量的不同，最多可分 23 个）。EXT - DOS 分区创建后也要用 DOS 的 format 命令格式化，但可以不要 /s 参数，即分区内没有系统文件。没有格式化的分区是不能使用的。

装入非 MS-DOS 的磁盘操作系统（象 XENIX、CP/M 等）的分区称为【Non-DOS 分区】。但是不能用 FDISK 程序的 DOS 版本建立它。不过，有些系统也笼统地把 EXT-DOS 分区称为 non-DOS 分区。

任何时刻对一个硬盘上只能选择一个活动分区，活动分区上必须有操作系统。机器启动或复位时将引导活动分区内的操作系统。

每个分区内的空间是连续的（DOS 把每个分区都视为从 0 开始的扇区连续块），各分区的大小可以不同（由 FDISK 程序选择分区的编号和空间）。当有多个操作系统时，每个操作系统仅占一个分区，这保证任一操作系统下的系统和应用程序不访问另一操作系统下程序或数据文件。

通常 BIOS 将试从驱动器 A 的 0 面 0 磁道 1 扇区读数据到 0000:7C00。如果失败，而机中装了硬盘，则 BIOS 将读取第一硬盘的 0 面 0 柱 1 扇区，该区含有主自举装入程序（主引导程序）和分区表。主自举扇区的内容装入 0000:7C00 后，控制转移到主自举装入程序，它将扫描分区表，寻找有效的文件分区，然后装入操作系统自举装入程序（在有效分区第一扇区中），并将控制交给它。在真 IBM PC 中，如果软盘或硬盘都没有有效自举区，将发出中断 INT 18H，转向 ROM BASIC，这时你就不能用硬盘了。所以硬盘上这部分数据是不允许轻易更改的，仅管 Turbo C 可以访问到它。

为了管理硬盘的分区，MS-DOS 在硬盘的物理第一扇区存放着一个主引导程序，该程序之后有一个分区表（见图 19-1）：

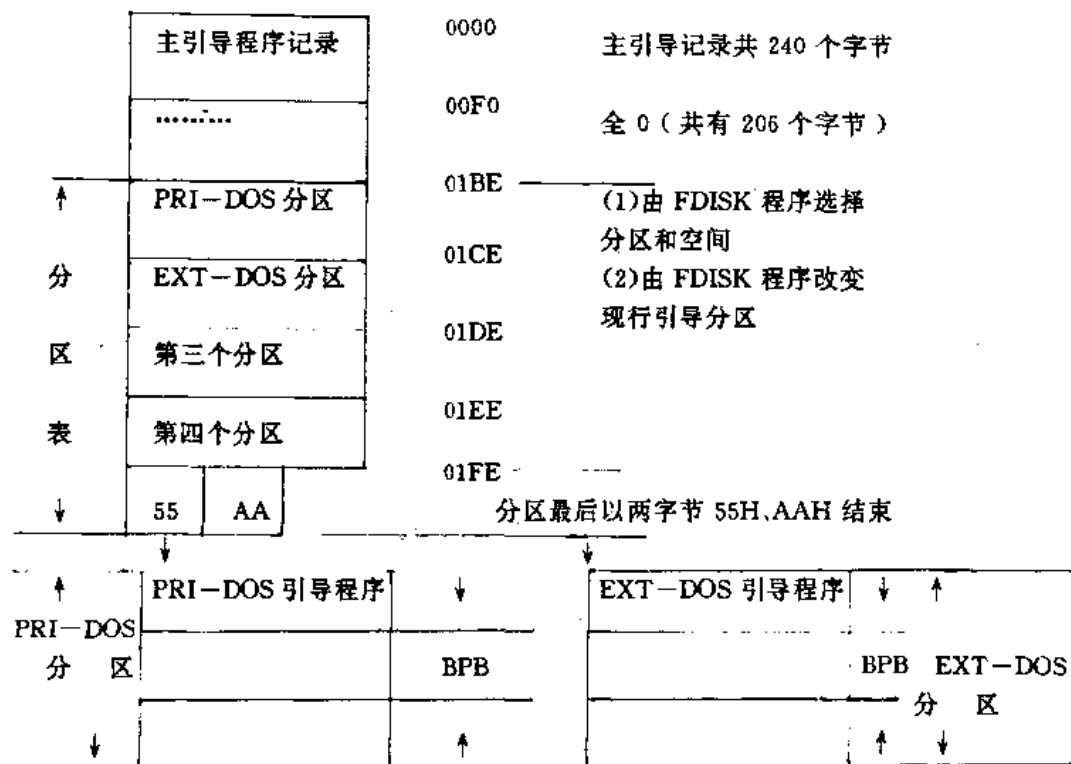


图 19-1 硬磁盘两个引导程序之间的关系

主引导程序由 LowFORM(【低级格式化】，也称预格式化或硬格式化)程序记录在硬盘 0 面 0 柱 1 扇区上，主要用于硬盘自举(其主要功能包括：建立硬盘基数表、软盘基数表、决定是软盘启动还是硬盘启动、读软盘或硬盘主引导记录到内存及检查扇区最后一个字是否为 AA55H 等)，它为各分区所共用。

19.1 主引导扇区的查找

下面以 ST251(40 兆)硬盘分两个分区(PRI-DOS 与 non-DOS，从而硬盘被分为 C 盘和 D 盘)，操作系统为 MS-DOS 3.3A 为例来说明。

要查硬盘 0 面 0 柱 1 扇区(又称【主引导扇区】)可用 DEBUG 程序进行：

```
C>DEBUG
-A
XXXX:0100 MOV DX,0080
XXXX:0103 MOV CX,0001
XXXX:0106 MOV BX,0200 ;将主引导扇区读到 ES:BX 开始的空中
XXXX:0109 MOV AX,0201 ;读出
;如将它改成 MOV AX,0301 便是写回
XXXX:010C INT 13 ;读写硬盘的物理地址
XXXX:010E INT 3 ;断点中断程序执行
XXXX:010F ^ C
-G
-D 200 3FF
XXXX:0200 FA 2B C0 8E D0 8E C0 8E-D8 B8 00 7C 8B E0 FB 8B
```

```

XXXX:0210 F0 BF 00 7E FC B9 00 01-F3 A5 E9 00 02 B9 10 00
XXXX:0220 8B 36 85 7E F6 04 80 75-08 83 EE 10 E2 F6 EB 37
XXXX:0230 90 BF BE 07 57 B9 08 00-F3 A5 5E BB 00 7C 8B 14
XXXX:0240 8B 4C 02 BD 05 00 B8 01 -02 CD 13 73 09 2B C0 CD
XXXX:0250 13 4D 74 19 EB F0 BE FE-7D AD 3D 55 AA 75 14 BE
XXXX:0260 BE 07 EA 00 7C 00 00 8B-36 87 7E EB 0A 8B 36 89
XXXX:0270 7E EB 04 8B 36 8B 7E AC-0A C0 74 FE BB 07 00 B4
XXXX:0280 0E CD 10 EB F2 EE 7F 8D-7E A7 7E C8 7E 0D 0A 49
XXXX:0290 6E 76 61 6C 69 64 20 50-61 72 74 69 74 69 6F 6E
XXXX:02A0 20 54 61 62 6C 65 00 0D-0A 45 72 72 6F 72 20 4C
XXXX:02B0 6F 61 64 69 6E 67 20 4F-70 65 72 61 74 69 6E 67
XXXX:02C0 20 53 79 73 74 65 6D 00-0D 0A 4D 69 73 73 69 6E
XXXX:02D0 67 20 4F 70 65 72 61 74-69 6E 67 20 53 79 73 74
XXXX:02E0 65 6D 00 00 00 00 00 00-00 00 00 00 00 00 00 00
XXXX:02F0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
XXXX:0300 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
.....
XXXX:03B0 00 00 00 00 00 00 00 00-00 00 00 00 00 80 01 ← PRI-DOS
XXXX:03C0 01 00 04 05 51 99 11 00-00 00 4B A3 00 00 00 00 ← non-DOS
XXXX:03D0 41 9A 51 05 D1 32 5C A3-00 00 F6 A2 00 00 00 00
XXXX:03E0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
XXXX:03F0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 55 AA

```

上述程序将主引导扇区读到内存 ES:BX 开始的空间中, ES:0200 ~ 02F0 此 240 个字节为主引导记录区;从 ES:03BE ~ 03FD 这 64 个字节为分区表。容易看出偏移 0200 ~ 03FF 相当于 0000 ~ 01FF。

19.2 主引导扇区中分区内容的说明

每个分区占 16 个字节。表 19-1 以 PRI-DOS 分区中内容为例。

表 19-1

偏移	目的	指示符	磁头号H	扇区数S	柱面CYL
01BE	分区开始	80 (活动分区)	01	01	00
01C2	分区结束	04 (操作系统)	05	51	99
01C6	分区前的扇区数	11 (低字低位) 00 (高位)		00 (高字低位) 00 (高位)	
01CA	本区占扇区数	4B (低字低位) A3 (高位)		00 (高字低位) 00 (高位)	

说明:

1. 活动指示符为 80H 表明该分区为活动分区,即它包含一个可装入的操作系统;为 00H 则不是活动分区。当为 80H 时,它将通知主引导程序装入其后三个字节信息指定扇区的内容——活动分区选择的操作系统的引导程序,并把控制权交给它,再由它装入系统程序。

2. 不同操作系统有不同的操作系统指示符。操作系统指示符为 04H 表明分区的文件分配表 (FAT) 的每簇占 16 位 (2 个字节) 的登记项; 为 01H 表示每簇占 12 位 (1.5 字节); 为 00H 表示不定, 因而不被识别。对 EXT-DOS、non-DOS 分区, 可能为其它值。
3. 分区前的扇区数等于分区的隐藏扇区数 (见下面对分区的 BPB 说明)。
4. 本区占扇区数是指分区实际在磁盘上占有的扇区总数, 它等于分区 BPB 中总扇区数。

19.3 分区的基本输入输出参数块 BPB (BIOS Parameters Block) 的内容

为说明 BPB 先用 PC Shell 工具软件对磁盘 C 编辑, 得 C (PRI-DOS 分区) 的引导程序:

Absolute sector 0000000, System BOOT, Disk Abs Sec 0000000

BPB 开始字节

(JMP 0136)(MSDOS 3.3) ↓

0000(0000) EB 34 90 4D 53 44 4F 53 33 2E 33 00 02 04 01 00

0016(0010) 02 00 02 4B A3 F8 29 00 11 00 06 00 11 00 00 00

↑

BPB 结束字节

0032(0020) 00 00 00 00 00 00 00 00 00 00 00 00 00 00 12

0048(0030) 00 00 00 00 01 00 FA 33 C0 8E D0 BC 00 7C 16 07

.....

0464(01D0) 75 72 65 0D 0A 00 49 4F 20 20 20 20 20 20 53 59

0480(01E0) 53 4D 53 44 4F 53 20 20 20 53 59 53 00 00 00 00

0496(01F0) 00 00 00 00 00 00 00 00 00 00 00 00 00 80 55 AA

得 D (non-DOS 分区) 的引导程序:

Absolute sector 0000000, System BOOT, Disk Abs Sec 0000000

BPB 开始字节

(JMP 0100)(NOSYSTEM) ↓

0000(0000) EB FE 90 4E 4F 53 59 53 54 45 4D 00 02 04 01 00

0016(0010) 02 00 02 F6 A2 F8 29 00 11 00 06 00 5C A3 80 00

↑

BPB 结束字节

.....(为全 0)

0480(01E0) 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

0496(01F0) 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA

BPB 结尾的 55 AA 两个字节表明此分区合法。

表 19-2 DOS 3.30 (DOS 5.0) BPB 构成格式和作用

代号	作用	起始偏移	占用字节	PRI-DOS 分区值	non-DOS 分区值
SS	每扇区字节数	000B	2	0200H = 512 字节	
AU	每个分配簇的扇区数	000D	1	04H	
RS	保留扇区数	000E	2	0001H	
NF	FAT 表的个数	0010	1	02H	
DS	根目录项的个数	0011	2	0200H	
TS	分区所占总扇区数	0013	2	A34BH	A2F6H
MD	磁盘介质说明符	0015	1	F8H (表明是硬盘)	
FS	每个FAT表占扇区数	0016	2	0029H	
ST	每个磁道(柱)的扇区数	0018	2	0011H	
NH	磁头(面)个数	001A	2	0006H	
HS	隐藏扇区的个数	001C	2(4)	0011H	A35CH
	分区中总扇区数	0020	4	可能比CHKDSK 得值大	
	物理驱动器号	0024	2	00H=软盘, 41H=虚拟 80H=1号硬盘等 对	
	扩展引导记录标志	0026	1	如 29H, 盘用 DOS 4. XDOS 5.0 或以后版本 格式化	
	磁盘系列号	0027	4	随机产生BCD 码	
	磁盘卷标号	002B	11	DOS 内部使用卷标	
	文件系统标识	0036	8	FAT12 或FAT16 ↓	

说明:1. 保留扇区数指从逻辑 0 扇区(即为 0 面 0 道 1 扇区的物理扇区)开始的扇区数。一

般逻辑 0 扇区存放引导程序。保留扇区 1 个是指主引导扇区。

2. 对 DOS 5.0 若在偏移 0013H 处为 0, 则总扇区数在偏移 0020H 处, 占 4 个字节。

3. 隐藏扇区是指从磁盘物理 0 扇区到分区第一个扇区前的所有扇区。例如 PRI-DOS 分区前隐藏了 17 个扇区, 即 1 个磁道。从表 19-2 中很容易看出, PRI-DOS 共占用扇区数为 A34BH=41803 字节, 如果将此数加上 17 个扇区(即 1 个保留扇区)便是

$41803 + 17 = 41820 = A35CH$

而 A35CH 恰是 non-DOS 的隐藏扇区数。从表 19-3 可以看出软盘无隐藏扇区。

(表接下页)

表 19-3 不同磁盘的 PRI-DOS 的 BPB (仅供参考)

代 号	360KB 软 盘	1.2MKB 软 盘	1.44MKB 软 盘	10MKB 硬 盘	20MKB 硬 盘
SS	512	512	512	512	512
AU	2	1	1	8	4
RS	1	1	1	1	1
NF	2	2	2	2	2
DS	112	224	224	512	512
TS	720	2400	2880	20723	41531
MD	FD	F9	F0	F0	F8
FS	2	7	9	8	4
ST	9	15	18	17	17
NH	2	2	2	4	4
HS	0	0	0	17	17

第二十章 磁盘文件的结构

文件是一组信息的集合,文件常存储在(软、硬)磁盘上。磁盘文件名的命名一般规则是(方括号内的内容是任选的,参见《程序结构和主函数》一章):

[驱动器名;][路径名] 文件基本名 [. 文件附加名]

(1) 驱动器名可以是 A、B、C, ... 等。

(2) DOS 管理文件是用一个呈树状的目录结构,路径名便是描述这个目录的。DOS 对磁盘文件进行读写操作时,首先要在内存中建立一个文件控制块(FCB),然后根据 FCB 中的文件名和其它信息去访问指定的文件目录,找出相应的文件,随后在磁盘与内存之间进行文件数据传递。

正常情况下,磁盘上只有一个根目录。根目录下属的目录称子目录(可以有多个),子目录中又可包括子目录。例如,

```
C>c:\sub1\sub11\file.exe
```

便描述了 C 驱动器中子目录 SUB1 里的子目录 SUB11 中的文件 FILE.EXE, \sub1\sub11\ 便是路径名。如果路径名省略,则指当前目录(或现行目录)。如果已用 DOS 的 CHDIR 命令进入了某目录,则该目录便是当前目录。如果当前目录是 sub11,则下述命令

```
c>cd\sub\sub11
```

```
c>file.exe
```

与上述命令等价。路径名可以从根目录开始规定,直至文件所在目录;也可以从当前目录开始规定,直至文件所在目录。

路径用“\”符号分隔目录名。注意,在 Turbo C 中,用字符串描述路径时,括在双引号内的路径名要用“ ”即两个反斜杠分隔目录名,而在 DOS 下,路径只用“\”符号分隔便可以了。

(3) 文件基本名有时也简称文件名,它是必须有的。它常有三种形式:

—1 XXXXXXXX 最多用 8 个文件名标识符构成;

—2 * 一个星号(称【通配符】Wild-card Character);

—3 ???????? 每个 ? 也可称通配符,表示任一字符。

(4) 文件附加名常用于标识文件所属类型,也可用通配符。

为方便叙述,下面以 1.2 兆的威宝(Verbatim)软盘为例说明磁盘目录项的结构和 FAT 表的组成。

20.1 目录项的结构

可用 The Norton Utilities 工具软件分析软盘。例如,执行其 NU.EXE 文件后可以 看到磁盘总的结构(参见图 20-1)。

Menu 1.1.5

Select sector

You may select sectors numbered from 0 through 2,399

Starting sector: 15

Ending sector: 28

Outline of Sector Usage on This Disk

0	Boot area	(used by DOS)
1 - 14	FAT area	(used by DOS)
15 - 28	Root Dir. area	(used by DOS)
29 - 2,399	Data area	(where files are stored)

Item type	Drive	Directory name	File name
Directory	A:		Root dir

图20-1 选扇区

如选扇区 15 则显示目录项 (参见图 20-2)。

Sector 15	Directory format
Sector 15 in root directory	Offset 0, hex 0
	Attributes

Filename	Ext	Size	Date	Time	Cluster	Arc	R/O	Sys	Hid	Dir	Vol
N43	EXE	17730	1-20-94	5:38 pm	2	Arc					
QW			1-20-94	5:41 pm		Arc					Vol
ZMH94NEW		20634	1-21-94	9:17 am	37						Dir
unused directory entry											

注:子目录名 ZMH94NEW 为删除文件 DEBUG.EXE 后建立的,该文件长为 20634, 37(十进制数)是它起始簇号。

图 20-2 显示目录项

还可以用它显示目录项内容,现用 PC Sehll 工具软件显示 (选 Disk/View/EditDisk 命令,参见图 20-3)

Disk View/Edit Service

Absolute sector 000015, System ROOT, Disk Abs Sec 000015

0000(0000)	4E 34 33 20 20 20 20 20 45 58 45 20 00 00 00 00	N43	EXE
0016(0010)	00 00 00 00 00 00 DA 8C 34 1C 02 00 42 45 00 00	+4	BE
0032(0020)	51 57 20 20 20 20 20 20 20 20 20 28 00 00 00 00	QW	(
0048(0030)	00 00 00 00 00 00 24 8D 34 1C 00 00 00 00 00 00	\$ 4	
0064(0040)	5A 4D 48 39 34 4E 45 57 20 20 20 10 00 00 00 00	ZMH94NEW	
0080(0050)	00 00 00 00 00 00 38 4A 35 1C 25 00 00 00 00 00	8J5	%
0096(0060)	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00		
0112(0070)	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00		

图 20-3 显示目录项内容

从图 20-1 ~图 20-3 可以看出,磁盘上有三个目录项:一个执行文件 N43. EXE ,一个

卷标 QW，一个子目录 ZMH94NEW。正常情况下，每一个目录项由 32 个字节组成，各字节内容及含意如下：

1. 字节 0 ~ 7 k k 文件基本名

注意：当第一个字节 (0H) 为以下值时，该目录项具有特殊的意义：

—1 00H：表示该目录未使用过；

—2 E5H：表示该目录项已被删除；

—3 2EH：表示该项为子目录内的第一或第二个目录，即在进入子目录后用 DOS 的 DIR 命令列出时有：

. <DIR> 05-20-90 3:09p

.. <DIR> 05-20-90 10:18p

的两个目录，这两个目录的属性字节值为 10H。

注意：子目录名总是作为一个文件名来处理的，或者说 8 个字节便是子目录名，即第一个字节的值并不是 2EH，而是子目录名的第一个字符。与一般文件名不同的是，该目录项的属性字节值也为 10H。

2. 字节 8 ~ 10 k k 文件的类型名

它不包括文件扩展名中的第一个小数点，即是文件附加名。

前 11 个字节实际是文件名，当第 12 个字节即文件属性为卷标位时，这 11 个字节为卷标。

3. 字节 11 k k 文件属性，其含义如下：

位7	位6	位5	位4	位3	位2	位1	位0
共 享	××	档 案	子目录	卷标位	系 统	隐 藏	只 读

图 20-4

例如，当其为下列值时意义如下：

—1 01H：只读 (Read-Only) 文件，此文件不能被写入或删除；

—2 02H：隐藏文件 (Hidden，或称隐含文件)，当用 DOS 的 DIR 命令列目录时，此文件一般不被列出。

—3 04H：系统 (System) 文件，像早期 DOS 的 IBMBIO.COM、IBMDOS.COM 文件、MS-DOS 5.0 的 IO.SYS、MSDOS.SYS 文件都是系统文件，通常它们又是隐藏和只读文件，故实际值为 07H。

—4 08H：卷标位，表示该目录项的前 11 个字节为卷标，而目录项必在根目录中。该目录项除表示卷标名、属性、日期和时间字节外，其它各字节的值无用。卷标又是档案位，故实际值为 28H。

—5 20H：档案位，又称归档 (Archive) 或更改位，只要一个文件被写入且关闭，该位便置成 1；DOS 的 BACKUP 命令使用，一些软件用它表示文件已经备份。

—6 80H：共享 (Shareable) 位。文件允许在网络系统中使用，并告诉操作系统是否两个或两个以上用户可同时存取。

4. 字节 12 ~ 21 —— 保留，未用。

5. 字节 22 ~ 23 —— 建立文件时间。位 15 ~ 位 11 表示小时 (h，0 ~ 23)，位 10 ~ 位 5 表示分 (m，0 ~ 59)，位 4 ~ 位 0 表示秒 (s，它以 2 秒为单位)。数值均用二进制数表示，注意：低位在前，高位在后。

6. 字节 24 ~ 25 —— 建立文件日期。位 15 ~ 位 9 表示年 (y, 0 ~ 119, 表示 1980 年 ~ 2099 年), 位 8 ~ 位 5 表示月 (m, 1 ~ 12), 位 4 ~ 位 0 表示日 (d, 1 ~ 31)。数值均用二进制数表示, 注意: 低位在前, 高位在后。

7. 字节 26 ~ 27 —— 首簇编号, 即该文件的第一簇的簇号。注意: 所有硬盘和软盘上的第一个可分配的簇号总是 002, 簇号从最低字节的第一位开始有效。

8. 字节 28 ~ 31 —— 以字节数表示的文件长度, 注意: 低位在前, 高位在后。

20.2 文件分配表 (FAT)

DOS 用 FAT (文件分配表, File Allocation Table) 来管理磁盘空间的分配, 它也反映磁盘空间当前的使用情况。FAT 对每个文件起着链接表的作用, 它将磁盘文件所占的数据扇区链接成一个整体。在磁盘上有两个 FAT, 它们的内容是相同的。

对前述软盘, 执行 The Norton Utilities 工具软件的 NU.EXE, 并选扇区 1 有 (图 20-5)

Sector 1												FAT format	
Sector 1 in 1st copy of FAT												Cluster 2, hex 2	
3	4	5	6	7	8	9	10	11	12	13	14		
15	16	17	18	19	20	21	22	23	24	25	26		
27	28	29	30	31	32	33	34	35	36	<EF>	<EOF>		
00	00	00	00	00	00	00	00	00	00	00	00		

图 20-5 显示 FAT 表占簇情况

如选扇区 8 可发现它的一个拷贝 (显示 Sector 8 in 2nd copy of FAT)。有时候 需要把 FAT 读到一个 DOS 缓冲区中去 (打开, 分配更大的空间等), 并给那个缓冲区以高的优先权, 以后尽可能长期把它保留在存储器中。现用 PC Sehl 工具软件显示其内容 (参见图 20-6)

Disk View/Edit Service	
Absolute sector 000001, System FAT, Disk Abs Sec 000001	
0000(0000)	F9 FF FF 03 40 00 05 60 00 07 80 00 09 A0 00 0B @ .
0016(0010)	C0 00 0D E0 00 0F 00 01 11 20 01 13 40 01 15 60 + (@ .
0032(0020)	01 17 80 01 19 A0 01 1B C0 01 1D E0 01 1F 00 02 v + v
0048(0030)	21 20 02 23 40 02 FF FF FF 00 00 00 00 00 00 00
0064(0040)	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080(0050)	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

图 20-6 显示 FAT 表内容

20.2.1 逻辑扇区

在文件管理中, 所涉及的扇区的编号为逻辑扇区 (或绝对扇区, Absolute sector), 而在 ROM BIOS 中涉及到扇区的物理地址。

一个【物理扇区】是由驱动器号、面 (或头) 号、道 (或柱面) 号和扇区号等四个参数确定的, 而【逻辑扇区】是以 DOS 区域起始的物理扇区为逻辑 0 扇区, 并按扇区的访问顺序进行连续编号的扇区阵列。

(软盘) 逻辑扇区 0 k 0 道 0 面 1 扇区;

(软盘)逻辑扇区 1 k 0 道 0 面 2 扇区;

(软盘)逻辑扇区 2 k 0 道 0 面 3 扇区;

.....

1 道 0 面 1 扇区;.....; 1 道 1 面 9 扇区; 39 道 1 面 1 扇区等等。

逻辑扇区号的计算方法是

逻辑扇区号=(道号 * 总面数+面号)* 每道扇区数+扇区号-系统隐藏扇区数-1

【系统隐藏扇区数】是指在 DOS 区之前的非 DOS 使用的扇区数,对软磁盘其值为 0,对硬盘为 1(<DOS 3.0 分区)或 17(>=DOS 3.0 分区)。物理扇区和逻辑扇区的对应关系可从表 20-1 看出:

表 20-1

360K 软盘			20M 硬盘	
逻辑扇区号	对应区域	起始物理扇区 面号 柱面号 扇区号	逻辑扇区号	对应区域
0	BOOT 区	0 0 1		主引导分区
1~4	FAT 表	1 0 1	0	BOOT 区
5~11	根目录表	1 0 2	1~82	FAT 表
12~719	数据区	1 1 16	83~114	根目录表
		3 1 14	115~415986	数据区

FAT 总是从逻辑扇区 1 开始的,后边是目录数据。大于一个扇区时,扇区要连续。

20.2.2 簇 (Cluster) 的概念

它是磁盘空间的分配单位。对单面软盘,一个簇对一个逻辑扇区;而对双面软盘,一个簇为两个相邻的逻辑扇区。对双面 5.25 英寸软盘,簇的大小为 1KB,故每个逻辑扇区占 512B。每个簇在 FAT 中占 12 位(即 1.5 字节)。当文件需要空间时,系统将动态地分配一个簇。

20.2.3 FAT 的表头标志

FAT 的头两项(即 FAT 的前三个字节)是它的表头,其中第一个字节为磁盘介质说明符,用于说明磁盘的性质和规格(仅供参考,参见表 19-2、19-3)。

磁盘介质说明符

FFH

FEH

FDH

FCH

F9H

F0H

F8H

磁盘的性质和规格

双面、每道 8 扇区的软盘(320K 盘);

单面、每道 8 扇区的软盘(160K 盘);

双面、每道 9 扇区的软盘(360K 盘);

单面、每道 9 扇区的软盘(180K 盘);

双面、每道 15 扇区的软盘(1.2M 盘);

其它软盘

硬盘

而第二、第三个字节总是 FFH、FFH，这是系统设定的。

从 FAT 的第三项（该项标志 002 簇）开始，每一项用三个 16 进制的数作为一个簇的标志信息，其信息的含义如下：

值	含 义
000H	表示此簇未被使用，但是可用的；
FF0H ~ FF7H	用来标识保留的簇（FF7H 表明此簇在坏扇区上，DOS 将不分配这样的坏簇）。用 CHKDSK 检查磁盘时将返回坏簇的总数，它们不是分配链的一部分）。
FF8H ~ FFFH	标识文件的最后一簇，即用作文件结束标志；
xxxH	除上面三种 16 进制数外的 16 进制数，它指出文件下一个簇的编号。

注意：每一个文件的第一个簇号（又称【首簇号】）存放在该文件的目录项中，目录项的第 26 和第 27 个字节便是首簇号。

20.2.4 DOS 将一个簇分配给新文件的过程

在一个新文件刚建立时，DOS 并没有立即给它分配任何空间，只是在 FAT 中给它建立一个目录项。当第一个文件写入磁盘时，DOS 从可用的自由空间中分配一个簇（其内容为 FF8H ~ FFFH 之一）给这个文件；在这个簇（对应扇区）被写满之前，不再另外分配空间；当一个簇被写满后，再写一个字节时，DOS 则从自由空间中分配另一个簇（注意：此簇不一定与前一簇紧相邻），与此同时，将前一簇设置为下一簇的簇编号。

注意：当 DOS 用删除命令将一个文件删除后，便把 FAT 中该文件目录项的首字节改成 E5H，与此同时释放该文件占用的所有簇号，即在 FAT 中把对应的簇置成 000H，从而这些簇可以重新分配给其它新文件。

20.2.5 如何使用 FAT

1. 从 FAT 中查找文件的所有簇可按下列顺序进行：

—1 找首簇号

从文件目录项的第 26 和第 27 字节中得到文件的首簇号（注意：高位占高字节）。

—2 找首簇后继第一个簇

(1) 将首簇号乘 1.5（因为 FAT 中各项为 1.5 字节）；

(2) 乘积的整数部分就是首簇的第一个后继簇在 FAT 中的相对位移，且该登记项的内容就是下一后继簇的簇编号。

—3 找任一簇编号

(1) 取出 FAT 相对位移上的一个字（2 个字节，注意：高位占高字节）；

(2) 如果相对位移上的簇编号是偶数，则取该字的低 12 位，否则取高 12 位；

(3) 如果作为结果的 12 位是值 FF8H ~ FFFH，则表示该簇为文件的最后一个簇，否则为下一个后继簇的簇编号。

2. 簇转换为逻辑扇区的方法

—1 将簇编号减去 2；

—2 将差数乘于每个簇包含的扇区数（单面软磁盘为每簇一扇区，双面为 2 扇区；对硬盘在 DOS 引导区有说明）；

—3 将积加上开始的逻辑扇区号（即文件开始扇区，见表 20-2，硬盘见 DOS 系统引导

格式说明)。

表 20—2 5.25 英寸磁盘 FAT、簇、扇区关系表

面数	扇区 / 道	第一个FAT占扇区数	第二个FAT占扇区数	盘目录占扇区数	目录个数	每簇占扇区数	文件开始扇区
2	8	1(0.0.2)	1(0.0.3)	4(0.0.4~7)			7

二 绝对磁盘读写函数

- 绝对磁盘读函数 —3 absread()
- 绝对磁盘写函数 —4 abswrite()

三 随机块读写函数 (已过时)

- 随机块读函数 —5 randbrd()
- 随机块写函数 —6 randbwr()

四 取磁盘信息

- 获得磁盘上的自由空间信息 —7 getdfree()

五 取文件分配表 (FAT) 信息

- 取文件分配表信息 —8 getfat()
- 取缺省驱动器中文件分配表的信息 —9 getfatd()

六 低级磁盘 I/O 操作

- BIOS 中断 INT 13H 调用 —10 biosdisk()

七 磁盘验证标志 (参见《中断和中断函数》一章)

- 获得磁盘写验证标志 getverify()
- 设置磁盘写验证标志 setverify()

—1 char far *cdecl getdta(void)

【磁盘传输地址, DTA】用于建立磁盘 I/O 操作和目录搜索的缓冲区地址。

返回指向磁盘传输地址的指针当前值。它相当于调用 INT 21H 的功能 2FH。入口参数是: AH=0x2f。它只适用于 MS-DOS。

—2 void cdecl setdta(char far *dta)

把磁盘传送地址设置为 dta 所指定的值。它相当于调用 INT 21H 的功能 1AH, 入口参数是: AH=0x1a, DX=dta。在调用前先将 DS 压入堆栈。当程序开始时 DTA 被置于 PSP: 0080H 处 (通常为 128 字节)。DOS 在使用 FCB (文件控制块) 实现读写文件操作时利用 DTA 传输数据, 因此, 这时要指定 DTA。但当不用 FCB 进行 DOS 功能调用时, 则不必指定 DTA, 因为 DTA 不起实质性作用 (参见《文件管理》一章)。

它只适用于 MS-DOS。

C>TYPE DISK1.C

```
#include "dos.h"
#include "alloc.h"
#define PRT ptr=getdta();printf("%Fp\n",ptr)
main()
{
char far *ptr;
char far *p;
p=(char *)farmalloc(10241);
```



```

PRT;
setdta(p);
PRT;
}/* 程序输出: 77DF:0080
7979:0008 */

```

函数 `absread()` 和 `abswrite()` 通过 MS-DOS 的 BIOS 功能调用读写磁盘指定扇区的内容, 它们忽略磁盘的逻辑结构, 并不关心文件、FAT 和目录。它们只适用于 MS-DOS。

—3 int —Cdecl `absread(int drive, int nsects, int lsect, void * buffer);`

绝对磁盘读函数。通过 DOS 的绝对磁盘读入中断 INT 25H 将指定磁盘扇区的内容读入内存缓冲区。drive 是要读磁盘驱动器号 (0000 ~ 磁盘最大扇区号), 0 是 A 驱动器, 1 是 B 驱动器等等; nsects 是要读的扇区个数; lsect 是要读的起始逻辑扇区号; buffer 是将数据读入内存中的存储区首地址 (亦称【传递地址】), 它是用于读入数据的缓冲区。

要读的扇区数受段中 buffer 以上存储空间量的限制, 因此调用函数一次最大可读存储空间为 64K 字节。

注意:

1. 它只适用于 MS-DOS。
2. 它只适用于软盘或 32MB 以下硬盘。

DOS 的中断 INT 25H 的入口参数是, `AL = drive`, `CX = nsects`, `DX = lsect`, `DS:BX = buffer`。例如, `absread(0, 1, 0, buf)`; 相当于如下一段汇编程序 (假定定义的 `char buf[50]`; 首址为 `DS:0400`, 可在调试程序时用调试表达式 `&buf[0], p` 观察):

```

MOV AL, 00 ;读 A 盘
MOV BX, 0400 ;置 BUF 地址 DS:0400
MOV DX, 0000 ;读逻辑 0 扇区
MOV CX, 0001 ;共 1 个扇区
INT 25 ;调 INT 25H 读

```

除段寄存器外, 其余寄存器的值经此调用后均可能被改变, 因此, 在发送这个中断前, 一定要注意: 保护有关寄存器 (通过压栈与弹出)。如果调用成功, 函数返回 0 (进位标志 `CF = 0`), 否则返回 -1, 并置全程量 `errno` 为函数调用后返回 AX 寄存器的值。AL 内是 DOS 出错代码, AH 中的内容含义为:

```

80H 设备无响应 (超时)
40H 寻道失败
20H 控制器失败
10H 在磁盘读入时 CRC 错
08H DMA 操作越界
04H 要求的扇区未找到
03H 企图写到贴有写保护的磁盘上 (仅对中断 INT 26H)
02H 地址标志未找到
00H 除上述以外的错误

```

说明: 对容量大于 32MB 的磁盘读 (或写) 时调用中断 INT 25H (或 INT 26H) 的入口参数是, `AL = drive`, `CX = FFFFH`, `DS:BX` 指向磁盘读出包:

偏移量	大小	说明
00H	4 个字节	扇区号
04H	2 个字节	要读 (或写) 的扇区数
06H	4 个字节	传输地址

```

C>TYPE DISK2.C
#include "dos.h"
main()
{
    static char buffer[512];
    register j,k; /* 从 12 扇区开始读 B 盘一个扇区的内容 */
    if( absread(1,1,12,buffer)==-1){
        printf("xxxx!");
        exit(1);
    }
    for(j=0,k=0;j<512;j++) /* 按 16 进制和字符两种形式输出 */
    {
        printf(" %x ",buffer[j] & 0x00ff);
        if( (j%16) && (j>0) )
        {
            for(;k<j;k++)printf(" %c",buffer[k]);
            printf("\n");
        }
    }
}

```

—4 int —Cdecl abswrite(int drive,int nsects,int lsect,void * buffer);

通过 DOS 的绝对磁盘写中断 INT 26H 将内存缓冲区中的数据写入指定驱动器的磁盘上。其参数说明同 absread()。

由于它在写磁盘时并不考虑磁盘上文件、FAT 和目录等具体分布,因此在使用时应非常小心,以免破坏磁盘目录或文件等。

```

C>TYPE DISK3.C
#include "stdio.h"
#include "dos.h"
#include "string.h"
#include "conio.h"
#include "ctype.h"
void display(int,int);
unsigned char buf[53000]; /* 定义缓冲区 */
main(int argc,char * argv[]) /* 对磁盘搜索指定串的程序 */
{ /* 根据刘民的程序改写并注释 */
    /* 在集成环境下可用 Options/Arguments 命令输入命令行参数,例如 */
    /* 为搜索 A 盘上的串 key 可键入: a: s key */
}

```

```

int Onesec,disk,length,c1,c2;
unsigned register i,j;
long Totalsec,start,c1;
union REGS regs;
unsigned char *p,*sp,comp[80];
clrscr();
if(argc==4)
{
printf("磁盘搜索开始");
disk=toupper(argv[1][0])-'A'; /* 先将盘符小写变成大写,disk 得驱动器号 */
regs.h.ah=0x1c; /* DOS 功能调用 1CH 得到指定驱动器信息 */
regs.h.dl=disk+1; /* 驱动器号,0 = 缺省,1 = A, 2 = B, ... */
intdos(&regs,&regs); /* 返回的全是 16 进制值 */
Totalsec=regs.x.dx*regs.h.al; /* dx = 簇的总编号,al = 每簇扇区数 */
/* 得到待读写的总扇区数 Totalsec */
Onesec=51200/regs.x.cx; /* cx = 每扇区字节数 */
/* 得到每一次搜索的扇区数 Onesec */
p=argv[3]; /* 指向待搜索字节 */
if(toupper(argv[2][0])=='S') /* 如果是字符串 */
{
strcpy(comp,argv[3]); /* 将待搜索串拷入 comp,最多 80 个字符 */
c1=strlen(comp); /* 求出待搜索串长度 */
}
else
/* 将待搜索串每两位翻译成一个字符,放入 comp[i]。可用下述调试表达式帮助:
64,c 与 48,c 求出 ASCII 码 64 和 48 对应的字符
'a'>64 与 '0'>64 看结果是 1 或 0 */
for(i=0;i<strlen(argv[3])/2;c1=++i)
/* 同时处理偶数个输入,奇数时最后一个被忽略 */
comp[i]=(toupper(*p)-48-(*p>64)*7)*16+toupper(*(++p))-48-(*p>64)*7;
for(start=0;Totalsec>0;start+=Onesec,Totalsec-=Onesec)
{
length=Totalsec>Onesec? Onesec:Totalsec; /* 得到要读扇区数,16 进制数 */
absread(disk,length,start,buf); /* start 是起始逻辑扇区号 */
printf("搜索扇区:%8ld~%-8ld",start,start+length-1);
p=buf; /* 指向缓冲区头部,现缓冲区中装的是从磁盘读入的内容 */
for(i=0;i<length*regs.x.cx;i++,p++) /* length*regs.x.cx 为读入字节数 */
/* sp 为中间指针 */
{ sp=p;
/* clen 为待搜索串字节数 */
for(j=0;j<c1;j++)
{
if(*sp==comp[j])break; /* 每次从缓冲区取出一个字节与搜索串比较 */
if(j==c1-1) /* 全部相等时 */
{
c1=start+i/regs.x.cx;
c2=i%regs.x.cx;
printf("\n已找到,起始扇区:%#x 偏移:%#x",c1,c2);
}
}
}
}

```

```

    display(i,length * regs.x.cx);
    printf("按任一键继续进行搜索");
    getch();
    }
    }
    }
    printf("\搜索结束");
    }
    else
    {
        printf("\n\a 正确的命令行应为:\n"); /* 响铃提示命令行未输入 */
        printf("\t 本执行程序名 盘符 S 待搜索ASCII 码串\n 或\n");
        printf("\t 本执行程序名 盘符 H 待搜索16 进制串\n");
        printf("ASCII 码串不要用西文双引号括起,16 进制串应有偶数个字符\n");
        printf("压任一键结束程序运行,请重新输入正确的命令行! \n");
        getch();
    }
}

void display(int sta,int num) /* 相当于用 debug 的 D 命令显示 */
{
    int i,j,k=sta/256 * 256;
    num -= sta;
    for(i=0;i<16;i++)
    {
        for(j=0;j<16;j++,num--) /* 左边显示 16 进制值 */
        {
            printf("%02X ",buf[i * 16+j+k]);
            if(j==7)printf(" ");
        }
        printf(" ");
        for(j=0;j<16;j++) /* 右边显示对应的 ASCII 码值 */
        {
            if(isprint(buf[i * 16+j]))printf("%c",buf[i * 16+j+k]);
            else printf(". ");
        }
        printf("\n");
    }
}

--5 int --Cdecl randbrd(struct fcb * fcb,int rent);

```

随机块读函数,已过时,不少 DOS 扩充软件不支持它。它实际相当于 DOS 功能调用 27H,即随机块读,它将 rent 个记录读到当前磁盘传输地址所在的存储区,它们是从 FCB(文件控制块)的随机记录字段指明的磁盘记录中读到的。其入口参数是,AH=0x27,CX=rent(记录数,必须不为0),DX=fcb(DS:DX 指向被打开的 fcb)。

实际读写的记录可通过检查 FCB 的随机记录域而确定,随机记录域值随着实际读写的记录而变。

函数返回值是函数调用后 AL 中的返回值。如果在读完所有记录之前,到达文件结束,那末 AL 为 01 或 03 返回。01 表示到达文件末尾,并且最后一个记录读完成(故是完整的)。03 返回表示跳过文件末尾,最后一个记录是不完整的。如果在磁盘传送段中出现了重迭 0xFFFF 上面的地址(即读记录到 0xFFFF 地址后反绕),以便读出尽可能多的记录,则 AL 以 02 返回。如果成功地读出所有记录,那么 AL 为 00 返回。在任何情况下,返回时 CX 为实际读出记录数,并且将随机记录字段和当前块/记录字段设置成指向下一个记录(没有读出的一个记录)。

结构 fcb 在 dos.h 中定义为:

```
struct fcb{          /* 标准 FCB (参见《文件管理》一章)          */
char fcb—drive;     /* 驱动器号,0 = 缺省驱动器,1 = \ 驱动器,等 */
char fcb—name[8];   /* 文件基本名,8 个字符 */
char fcb—ext[3];    /* 文件扩展名,3 个字符 */
short fcb—curblk;    /* 现行块数 */
short fcb—recsize;   /* 逻辑记录 (Logical record) 长度 (字节数) */
long fcb—filesize;   /* 文件长度 (字节数) */
short fcb—date;     /* 最后改写文件日期 */
char fcb—resv[10];  /* 为 DOS 保留 */
char fcb—currec;    /* 块中现行记录 */
long fcb—random;    /* 随机记录 (Random record) 数 */
};
```

对扩充 FCB 在 dos.h 中定义了另一个结构 xfc_b:

```
struct xfc_b {
char      xfc_b—flag;    /* 常用 0xFF 表示扩充 FCB */
char      xfc_b—rev[5]; /* 为 DOS 保留 */
char      xfc_b—attr;    /* 搜索属性 */
struct fcb xfc_b—fcb;    /* 标准 FCB */
};
```

—6 int —Cdecl randbwr(struct fcb *fcb,int rent);

随机块写函数,已过时,不少 DOS 扩充软件不支持它。除了写和写保护检查外,基本上和 randbrd() 相同。如果函数返回 0,表示所有记录被写入;返回 1 时表示磁盘空间不够;返回 2 时表示写记录时出现重绕情况。

它相当于 DOS 功能调用 28H,入口参数为 AH=0x28, CX=rent, DX=fcb。如果 rent=0,尽管没有记录写入,但文件总是被置到随机记录字段指定的长度,不管它比当前文件长度长还是短(单元被相应地分配或释放)。

—7 void —Cdecl getdfree(unsigned char drive,struct dfree *dtable);

获得磁盘驱动器号为 drive(0= 缺省驱动器,1=A 驱动器,...) 的磁盘上的自由空间信息,信息被存放在由指针 dtable 所指的 dfree 型结构中,dfree 在 DOS.H 中定义为:

```
struct dfree {
unsigned df—avail; /* 可用簇数 (available clusters) */
unsigned df—total; /* 全部簇数 (total clusters) */
unsigned df—bsec;  /* 每扇区字节 (Bytes per sector) */
unsigned df—sclus; /* 每簇扇区数 (sectors per cluster) */
};
```

该函数虽然不返回值,但当其出错时结构成员 `df-sclus=-1`。驱动器未用空间为

`df-sclus * df-avail * df-bsec`

字节。它实际相当于 DOS 中断 INT 21H 的功能调用 36H。入口参数是, `AH=0x36`; `DL=drive`。返回时, `BX` 中为 `df-avail`, `DX` 中为 `df-total`, `CX` 中为 `df-bsec`, `AX` 中为 `df-sclus`, `AX` 为 `0xFFFF` 时出错。它只适用于 MS-DOS。

```
C>TYPE DISK4.C
#include "dos.h"
#include "stdio.h"
main()
{
    struct dfree diskfree;
    char *ch[]={"", "a", "b", "c", "d", "e", "f"};
    int drive;
    if(*ch[3]=='c' || *ch[3]=='C')
        {printf("drive=%c ", *ch[3]); drive=3;}
    else {printf("drive=w 缺省 "); drive=0;}
    getdfree(drive, &diskfree);
    printf("%u %u %u %u\n", diskfree.df-avail,
        diskfree.df-total, diskfree.df-bsec, diskfree.df-sclus);
}
/* 例如输出, drive=c 18339 51033 512 4 */
```

—8 void —Cdecl getfat(unsigned char drive, struct fatinfo *dtable);

取文件分配表 (FAT) 信息。drive 是驱动器号 (0 为缺省, 1 为 A 驱动器等等), 文件分配表信息被填入 dtable 所指的 fatinfo 型结构中。fatinfo 在 DOS.H 中定义为:

```
struct fatinfo {
    char fi-sclus; /* 每簇扇区数 (Sectors per cluster) */
    char fi-fatid; /* FAT 标识字节 (The FAT identical byte) */
    int fi-nclus; /* 总簇数 (Number of clusters) */
    int fi-bysec; /* 每扇区字节数 (Byte per sector) */
};
```

它相当于 DOS 中断 INT 21H 的功能调用 1CH。入口参数是, `AH=0x1C`; `DL=drive`。返回 `AL` 中为 `fi-sclus`, `DX` 中为 `fi-nclus`, `CX` 中为 `fi-bysec`, `BL` 中为 `fi-fatid`。它只适用 MS-DOS, 而且和 DOS 版本有关。

```
C>TYPE DISK5.C
#include "dos.h"
#include "stdio.h"
main()
{
    struct fatinfo d;
    char *ch[]={"", "a", "b", "c", "d", "e", "f"};
    int drive;
```

```

long total;
if(*ch[3]=='c' || *ch[3]=='C')
    {printf("drive=%c ", *ch[3]); drive=3;}
else {printf("drive=缺省 "); drive=0;}
getfat(drive, &d);
printf("%d 0x%x %d %d\n", d.fi-sclus, d.fi-fatid, d.fi-nclus, d.fi-bysec);
total=(long)d.fi-sclus * (long)d.fi-nclus * (long)d.fi-bysec;
printf("bytes of total disk space=%ld\n", total);
}
/* 对软硬盘测试结果有 ( 这里硬盘标识 fi-fatid 为 F8, 1.2 兆软盘为 F9, 360 KB 软盘为 FD 等 ),
drive=b 1 0xffff 2371 512          对 1.2MB 软盘输出完全正确
bytes of total disk space=1213952
drive=c 4 0xffff 15588 512          对 ST251 型 40MB 硬盘输出正确
bytes of total disk space=31924224 但总字节数稍有出入
在 LX-386/33S 的 MS-DOS 5.0 下, 对 100MB 硬盘 fi-nclus 不正确, 100MB 硬盘的规格为:
Hard disk Type Cyls Head WPcom LZone Sect Size
      44   776   8      65535   775   33   100MB
drive=c 4 0xffff -14503 512
bytes of total disk space=-29702144
原因是磁盘总字节数超出 long 类型范围 */

```

—9 void —Cdecl getfatd(struct fatinfo *dtable);

它是取缺省驱动器中文件分配表的信息, 相当于 getfat(0, struct fatinfo *dtable);。
它相当于 DOS 中断 INT 21H 的功能 1BH, 入口参数是 AH=0x1B, 返回 dtable。

—10 int —Cdecl biosdisk(int cmd, int drive, int head, int track,
int sector, int nsects, void *buffer);

本函数是一个使用 BIOS 中断 INT 13H 的操作 (低级磁盘 I/O), 主要把对硬盘或软盘进行输入 / 输出的操作直接转给 BIOS 进行。注意: 因为这些操作主要是对扇区进行, 而忽略磁盘和文件的逻辑结构, 因此, 使用该操作要求程序员必须对磁盘系统和 DOS 有很好的了解, 不要轻易使用它, 以免产生不可预料的结果。对具体的软硬件环境, 只有在验证后才能使用它。

在集成环境下执行磁盘操作时屏幕将转向用户屏幕, 操作结束后又返回 TC 屏幕。

cmd 相当于 INT 13H 的子功能号, 它指示待执行的操作。对于 IBM PC 及兼容机, cmd 可能是 (表 20-3):

表 20-3 形参和调用寄存器对照表

BIOS 的中断 INT 13H 的子功能	Hcmd	nsects	track	sector	head	drive	buffer
	AH	AL	CH	CL	DH	DL	ES, BX
磁盘系统复位	00H					调用	
取上次磁盘操作状态	01H					调用	
读扇区到内存	02H	调用	调用	调用	调用	调用	调用
写磁盘扇区	03H	调用	调用	调用	调用	调用	调用

检验磁盘扇区	04H	调用	调用	调用	调用	调用	调用
格式化磁道	05H	调用	调用	调用	调用	调用	调用
格式化硬盘磁道, 设坏扇区标志	06H	调用	调用	调用	调用	调用	
在指定磁道开始格式化 硬盘	07H	调用	调用	调用	调用	调用	
取当前驱动器参数	08H					调用	
初始化控制器	09H					调用	
读长扇区	0AH	调用	调用	调用	调用	调用	调用
写长扇区	0BH	调用	调用	调用	调用	调用	调用
寻找柱面	0CH		调用	调用	调用	调用	
复位硬盘	0DH					调用	
读扇区缓冲区	0EH					调用	调用
写扇区缓冲区	0FH					调用	调用
检查驱动器准备好?	10H					调用	
重新对准驱动器	11H					调用	
硬盘控制器RAM 诊断	12H					调用	
硬盘驱动器自备诊断	13H					调用	
硬盘控制器内部诊断	14H					调用	

说明:

1. “调用”是指调用时要赋值,未指明者用缺省值。
2. 扇区数 (nsects) 不能等于 0。如对软盘为 1~9 (360KB) 或 1~15 (1.2MB)。在格式化硬盘磁道时它为交叉值。例如,格式化硬盘磁道时,ES: BX 指向一个 512 字节的缓冲区。该缓冲区开始部分是一张扇区交叉表,磁道上每个扇区一项,每项 2 个字节:F, N。第一个字节 F = 00H 表示是好扇区,F = 80H 表示是坏扇区;第二个字节 N = 扇区号。假定每磁道 17 扇区,交叉值为 2,则交叉表的内容为:

00,01,00,0A,00,02,00,0B,00,03,00,0C,00,04,00,0D,00,05,00,0E,
00,06,00,0F,00,07,00,10,00,08,00,11,00,09

如果是软盘格式化磁道,则 ES:BX 是扇区的 ID 信息地址,共有 4n 个字节,其中 n 为每道扇区数。对应每扇区的 4 个字节是 C, H, R, N,分别表示磁道号、磁头号、扇区号和扇区大小指数 (即扇区长度 = $128 * 2^N$)。

3. (软盘)磁道或(硬盘)柱面(track)号低 8 位。当柱面号大于 7 位时,高位值在 CL 中。
4. 扇区号 (sector) 一般占用 0~5 位,6~7 位用于存放硬盘柱面号的高位 (位 8、9)。
5. 磁头号 (head) 对不同的硬盘可能是不同的,可查阅硬盘参数表获得。
6. 驱动器号 (drive) 对软驱是:0 表示第 1 个软驱,1 表示第 2 个,2 表示第 3 个等等;对第 1 个硬盘是 0x30,第 2 个是 0x81,第 3 个是 0x82 等等。软硬盘区分是用位 7 是否为 1 进行的,位 7 值为 1 时表示驱动器是硬盘。

7. 数据一般以每扇区 512 字节读入缓冲区 (buffer), 或相反; 但当进行长读时, 每扇区读 (或写) 入 512 字节要加上额外的 4 个字节 (一些机器上为 4~7 个字节, 主要用作纠错码。数据错误一般不能自动纠正, 在遇到有 ECC 即 error checking and correction 时的第一个扇区后, 便放弃读入)。
8. 对 INT 13H 的子功能 08H, 驱动器返回信息在 buffer 的前 4 个字节中 (只对软盘有效)。其它返回的有:
BL = 驱动设备类型, CH = 最大柱面号的低 8 位, CL = 最大扇区号, DH = 最大磁头号, DL = 驱动设备的数目。
9. 在 100% 兼容 BIOS, 在内存 F000:EFC7 处存放着【磁盘参数】(表 20-4 与表 20-5)。

表 20-4

偏移	作	用
00H 位7~4:	步进率 (或步进速率)	
位3~0:	磁头退出时间 (即卸载时间, 0fh = 240毫秒)	
01H 位7~1:	磁头进给时间 (即加载时间, 01h = 4毫秒)	
位0:	非DMA 模式 (永远是 0)	
02H	延时直到电机关断 (时钟秒摆次数), 即马达等待时间	
03H	每扇区字节数 (扇区长度指数, 00h = 128, 01h = 256, 02h = 512, 03h = 1024)	
04H	每道扇区数	
05H	每扇区间的间隙长度 (扇区间隔字节数)	
06H	数据长度 (每扇区字节数, 如果字段非 0, 则忽略)	
07H	格式化时的间隙长度 (扇区间隔的填充字节)	
08H	格式填充字节 (默认值 f6h)	
09H	寻道后磁头稳定时间 (毫秒)	
0AH	电机起动时间 (以 1/8 秒为单位)	

表 20-5 不同驱动器/介质类型下的盘基数表

偏移	1.2MB/1.2MB	360KB/360KB 1.2MB/360KB	720KB/720KB 1.44MB/720KB	1.44MB/1.44MB
00H	DF	DF	DF	AF
01H	02	02	02	02
02H	25	25	25	25
03H	02	02	02	02
04H	0F	09	09	12
05H	1B	2A	2A	1B
06H	FF	FF	FF	FF
07H	54	50	50	6C
08H	F6	F6	F6	F6
09H	0F	0F	0F	0F

10. INT 13H 的其它子功能可能也有效,但你一定要在查清相关资料后才能去验证,否则可能出现不可预料的结果(也许不同软件版本有差异)。返回状态在 AH 中。该函数如调用成功(CF=0),返回 0,否则返回错误码。(参见表 20-6)

表 20-6

00H 成功	0DH 格式化扇区数无效(硬盘)
01H AH 中的功能(或参数)无效	0EH 检测到控制数据地址标记(硬盘)
02H 找不到地址标识	0FH DMA 判断电平出界(硬盘)
03H 磁盘写保护(软盘)	10H 不可改正的 CRC 或 ECC 读错
04H 找不到扇区	11H 数据 ECC 被改正(硬盘)
05H 复位失败(软盘)	20H 控制器失败
06H 磁盘被更换(软盘)	40H 寻时失败
07H 驱动器参数作用失败(硬盘)	80H 超时(未准备好)
08H DMA 过速	AAH 驱动设备未准备好(硬盘)
09H 试用 DMA 超过 64K 界限	BBH 未定义错(硬盘)
0AH 坏扇区(硬盘)	CCH 写错
0BH 坏磁道(硬盘)	E0H 状态寄存器错(硬盘)
0CH 不被支持的道或无效介质	FFH 传感操作错(硬盘)

11. 【磁盘系统复位】是指恢复磁盘系统条件到刚上电状态,即迫使控制器重新对准驱动头(寻找 0 磁道)。
12. 读写扇区是指绝对扇区。
13. 格式化磁道时,AL 中为交错值。
14. 【寻找柱面】是指将硬盘定位在指定柱面上。
15. 【复位硬盘】是指再初始化硬盘控制器,复位指定驱动器的参数,重新对准磁头(寻 0 磁道)。
16. 【写扇区缓冲区】是指初始化控制器的扇区缓冲区,但不向磁盘写数据。此操作应在格式化之前进行。
17. 【重新对准驱动器】是指使硬盘控制器寻找指定驱动器的 0 柱面。
18. 【CRC】即 Cyclic Redundancy Check,循环冗余码校验。
19. 硬盘的类型一般存在内存 F000:E6A1 处,参数存在 F000:E400 开始的字节中,每个硬盘占 16 个字节。例如,0132H = 306 柱面数,04H = 磁头数,80H = 128 预补偿,0131H = 305 停头位置,11H = 17 扇区数。
20. 本函数只适用于 IBM PC 及其兼容机上。功能 06H~14H 只适用于 XT 或 AT 机。

当硬盘出故障时可用此函数作辅助检测,以便了解故障原因。例如,利用 cmd 等于 00H、02H、03H 和 04H 的操作可以校正软盘驱动器。具体方法是先将一张在无故障机上格式化好了的软盘放入待校验驱动器内(注意:磁头必须是清洁的),然后读写指定磁道。根据测试结果逐渐校正驱动器磁头位置(拧螺钉调节)。

```
C>TYPE DISK6.C
```

```
#include "bios.h"
```

```
#include "alloc.h"
```

```

#define CHECK if(status != 0)printf("错误\n");\
                else printf("通过\n")
main()          /* 本程序可辅助于校正软盘驱动器 */
{
    int drive=0;          /* 选 A 驱动器 */
    int head=1;           /* 对磁头 0 面或 1 面选 1 */
    int track=0;          /* 对 360KB 盘 0~39 磁道选 0 磁道 (对 1.2MB、1.44MB 盘为 0~79)
                          */
    int sector=1;         /* 扇区号 */
    int nsects=8;          /* 扇区数 */
    void *buf=malloc(4096); /* 动态分配存储区 */
    int status;
    printf("磁盘复位测试...");
    status=biosdisk(0,drive,head,track,sector,nsects,buf);
    CHECK;
    printf("磁盘写测试...");
    CHECK;
    status=biosdisk(3,drive,head,track,sector,nsects,buf);
    while(kbhit() == 0)    /* 按任一有效键退出校验 */
    {
        printf("磁盘校验测试...");
        status=biosdisk(4,drive,head,track,sector,nsects,buf);
        CHECK;
    }
    free(buf);             /* 释放分配的存储区 */
    getch();              /* 读取按键值 */
}

```

第二十一章 程序头前缀 PSP

21.1 PSP 的作用

C 是一种过程语言，执行一个 C 程序（编译后的 *.EXE 程序）的全过程是，调入内存，运行，结束后返回（主程序或 DOS），某些程序也许不返回而常驻内存。

对 DOS 而言，所有用户程序都由命令处理器 COMMAND.COM 控制。DOS 建立的 PSP 主要用于沟通 DOS、DOS 命令行和用户程序之间的联系。

21.2 PSP 在内存中的位置

执行程序常有两种形式，带 .COM 扩展名的 COM 文件和带 .EXE 扩展名的 EXE 文件。PSP 是 Program segment prefix 即程序段前缀的缩写，是 DOS 中置于每个程序首的长度为 256 个字节的块。在【COM 文件】（指文件名后缀为 .COM 的文件）中，PSP 位于 CS:0 处，因而在编写 COM 文件的汇编代码时，总是从 100H 空间开始，以便为 PSP 存放留下空间；而在【EXE 文件】（指文件名后缀为 .EXE 的文件）中，代码位于 CS:0 中，DS 和 ES 段寄存器初始化成指向 PSP。

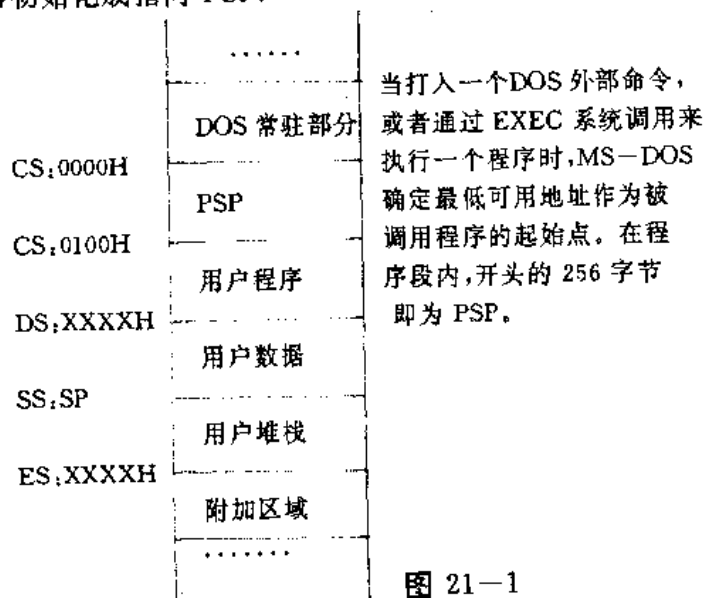


图 21-1

21.3 全局变量 — psp 和库函数 getpsp()

unsigned —Cdecl getpsp(void);

取出当前 PSP 的段地址。它实际是调用 INT 21H 的功能 62H，入口参数是 AH=62H，返回 BX= 当前进程 PSP 段值。注意：当前 PSP 并不一定是调用者的 PSP。该调用只适用于

DOS 3.X 及以上版本,对于 MS-DOS 2.X 和 3.X 低版本则由起动程序 (START-UP) 来设置全程变量 `—psp`。

```
C>TYPE PSP.C
#include "dos.h"
main(){
    unsigned psp;
    psp=getpsp();
    printf("%u %u\n",psp,—psp);
}
```

先将 L5.C 和 L6.C 分别编译,然后执行 L5.EXE,用于了解 PSP 的情况。

```
C>TYPE L5.C
#include "stdio.h"
#include "dos.h"
#include "process.h"
main()
{
    int i=0;
    unsigned x,x1,x2;
    printf("L5.c == == == —psp = %x\n",—psp);
    spawnl(P—WAIT,"l6.exe",NULL);
    x=getpsp();
    x1=FP—SEG(x);
    x2=FP—OFF(x);
    printf("FP—SEG = %x FP—OFF = %x —PSP = %x\n",x1,x2,—psp);
    for(i=0;i<0x100;i+=2)
        {printf(" %x ",peek(x1,i));}
    printf("\n");
}
```

```
C>TYPE L6.C
#include "dos.h"
main()
{
    int i=0;
    unsigned x,x1,x2;
    printf("L6.c — — — — — psp = %x\n",—psp);
    x=getpsp();
    x1=FP—SEG(x);
    x2=FP—OFF(x);
    printf("FP—SEG = %x FP—OFF = %x —PSP = %x\n",x1,x2,—psp);
    for(i=0;i<0x100;i+=2)
        {printf(" %x ",peek(—psp,i));}
    printf("\n");
}
```

/* 程序输出结果为:

```

l5.c=====psp=1478
L6.c-----psp=2694
FP-SEG=0 FP-OFF=2694 -PSP=2694
20cd a000 9a00 fef0 f01d 1f69 1488 14b
1343 156 1343 1478 101 1 ff02 ffff
ffff ffff ffff ffff ffff ffff 268c feda
27e0 14 18 2694 ffff ffff 0 0
5 0 0 0 0 0 0 0
21cd cb 0 0 0 0 2000 2020
2020 2020 2020 2020 0 0 2000 2020
2020 2020 2020 2020 0 0 0 0
0 d00 1488 81 5d4 4f43 534d 4550
3d43 3a43 435c 4d4f 414d 444e 432e 4d4f
4400 5054 433d 5c3a 5444 2050 3b 5250
4d4f 5450 243d 2470 67 4150 4854 433d
5c3a 5444 3b50 3a43 445c 534f 433b 5c3a
584c 4350 535c 5359 5400 4d45 3d50 3a43
445c 534f 0 1 3a43 545c 5c43 364c
452e 4558 200 800 0 ad00 2 0
FP-SEG=0 FP-OFF=1478 -PSP=1478
158 1488 6f4 70 16 12ce 6f4 70
6f4 70 ff54 f000 eb43 f000 eae b f000
3c 12ce 45 12ce 57 12ce 6f 12ce
87 12ce 9f 12ce b7 12ce 6f4 70
d60 b68 f84d f000 f841 f000 774 70
e739 f000 84a 70 e82e f000 efd2 f000
e000 f000 7fb 70 fe6e f000 6ee 70
ff53 f000 f0a4 f000 522 0 4112 c000
40bf 28 40eb 28 1f69 1488 14b 1343
156 1343 42d8 28 435f 28 a164 28
40c5 28 762 70 40c5 28 40c5 28
40c5 28 40c5 28 140 1343 1a0 1344
c6ea 2840 ea00 f000 40c5 28 40c5 28
40c5 28 40c5 28 40c5 28 40c5 28
40c5 28 40c5 28 40c5 28 40c5 28
40c5 28 40c5 28 40c5 28 40c5 28 */

```

21.4 PSP 的内容

由于可执行的 .COM 文件的四个段寄存器 (CS, DS, SS, ES) 共用一个段, 段开始包含 PSP, 所以可用 DOS 的 DEBUG 调试程序来了解 PSP 的结构。GREP.COM 是 Turbo C 提供的工具软件。

```
C>DEBUG GREP.COM L5.C
```

```
-r
```

```
AX=0000 BX=0000 CX=1B43 DX=0000 SP=FFFE BP=0000 SI=0000 DI=0000
```

```

DS=1A3E ES=1A3E SS=1A3E CS=1A3E IP=0100 NV UP EI PL NZ NA PO NC
1A3E:0100 E9260D      JMP     0E29
-D 0 0FF
1A3E:0000 CD 20 00 A0 00 9A F0 FE-1D F0 4F 03 89 14 8A 03
1A3E:0010 89 14 17 03 89 14 78 14-01 01 01 00 02 FF FF FF
1A3E:0020 FF FF FF FF FF FF FF FF-FF FF FF FF 36 1A 4C 01
1A3E:0030 49 19 14 00 18 00 3E 1A-FF FF FF FF 00 00 00 00
1A3E:0040 05 00 00 00 00 00 00 00-00 00 00 00 00 00 00
1A3E:0050 CD 21 CB 00 00 00 00 00-00 00 00 00 00 4C 35 20
1A3E:0060 20 20 20 20 20 43 20 20-00 00 00 00 00 20 20 20
1A3E:0070 20 20 20 20 20 20 20 20-00 00 00 00 00 00 00 00
1A3E:0080 05 20 6C 35 2E 63 0D 20-6C 35 2E 63 0D 0D 53 3B . 15.c. 15.c..S,
1A3E:0090 43 3A 5C 4C 58 50 43 5C-53 59 53 0D 00 00 00 00
1A3E:00A0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
1A3E:00B0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
1A3E:00C0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
1A3E:00D0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
1A3E:00E0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
1A3E:00F0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
-D 1A35:0
1A36:0000 43 4F 4D 53 50 45 43 3D-43 3A 5C 43 4F 4D 4D 41 COMSPEC=C:\COMMA
1A36:0010 4E 44 2E 43 4F 4D 00 44-54 50 3D 43 3A 5C 44 54 ND.COM.DTP=C:\DT
1A36:0020 50 20 3B 00 50 52 4F 4D-50 54 3D 24 70 24 67 00 P :.PROMPT=$p$.
1A36:0030 50 41 54 48 3D 43 3A 5C-44 54 50 3B 43 3A 5C 44 PATH=C:\DTP;C:\D
1A36:0040 4F 53 3B 43 3A 5C 4C 58-50 43 5C 53 59 53 00 54 OS,C:\LXPC\SYS.T
1A36:0050 45 4D 50 3D 43 3A 5C 44-4F 53 00 00 01 00 58 2E EMP=C:\DOS....X.
1A36:0060 43 4F 4D 00 1E B4 00 91-BA 0A 00 F7 E2 0B D2 75 COM.....u
1A36:0070 5A 3E 1A C2 85 91 46 EB-58 00 D8 F8 EB 05 46 F8 Z>....F.X....F.

```

00H	INT 20H			
08H		终止地址		
10H	Ctrl-Break 出口地址		关键错误出口地址	
	DOS 使用			
5CH	第一个格式化FCB			
6CH	第二个格式化FCB			
80H	未格式化参数区			
FFH				

图 21-2 PSP 内容

图 21-2 示意了 PSP 的内容,下面详细说明之。

—1 00 01 = CD 20

等号前为偏移,等号后为值。注意:值是低位在前,高位在后。为方便叙述,笼统地称多个

字节的集合为一个【域】，域中所含字节数称为【域长度】，如本域的长度为 2(下同)。

这是一个 INT 20H 中断指令，即为程序执行结束，返回 DOS。它的目的是支持由 CP/M 转为 DOS 的程序，因为在 CP/M 下程序终止时，将调用或跳转到内存地址中该程序所持的 PSP 段偏移 0 处。

用 EXEC 功能调用并执行一个程序时，程序返回有四种方法：

(1) 长转移到 PSP 段，偏移为 0 处；

(2) 用 CS:0 指向 PSP，再发一个 INT 20 实现；

(3) 用 AH=0 和 CS:0 指向 PSP，再发一个 INT 21H 中断（中断 INT 21H 的功能 0）；或者用 AH=4CH 而不对 CS 限定，而后发 INT 21H 中断；功能 4CH 是结束当前进程，把控制传送到引用进程。

(4) 用 AH=0 或功能调用 4CH，通过长调用到 PSP 的 50H 处（AL=50H）。

—2 02 03 = 00 A0

它是一个段地址，从此段开始后的各段不能为用户程序所用，即程序改变地址时不允许超出这个位置，除非通过中断 INT 21H 的功能 48H 来获得分配内存。

A000H 是段址，其对应的实际地址为(20 位)

A00000H=1010 0000 0000 0000 0000B=640KB

或者说，这一般表示你所用 PC 机当前可用内存结尾的段址。

—3 04 = 00

为DOS保留，未用。

—4 05 06 07 08 09 = 9A F0 FE 1D F0

这是含有一个到 DOS 函数的远 (FAR) 调用，或者说，跳转到 PSP 内的位移 5 处与 INT 21H 作用相同。

该域内这 5 个字节实际相当于这样一条指令，即 CALL F01D:FEF0，或者相当于 CS = F01DH，IP = FEF0H。取指令时，用指令指针 IP 的内容作为偏移地址（该值 FEF0H 是段内可用内存的字节数），用 CS 寄存器的内容作为段地址。如果使用这种方法，AX 中的内容将被冲掉；此法仅对 INT 21H 的 0H ~ 24H 功能有效，且功能号要放在 CL 寄存器内（其它寄存器则按功能要求来设置）。注意：这是为了提供与 CP/M 的相容性而做的努力（在 CP/M 内的 DOS 函数调用是经由到绝对位置 5 的调用来完成的），在 MS-DOS 系统下用户应尽量避免使用此法。

—5 0A 0B 0C 0D = 4F 03 89 14

当前程序之所以被执行是另外一个程序对 DOS 调用的结果，这另外一个程序通常称为【父程序，或称父进程】(Parent program)，当前程序则称为【子程序，或称子进程】。父程序通常是 DOS 的【命令处理器】（即程序 COMMAND.COM），但也可以是别的程序。例如，在 Turbo C 中，当使用 execl、spawnl、... 等进程函数时调用另一个子程序的子程序相对被调用的子程序便是父程序。

当前程序执行时，DOS 将父程序中的中断 INT 22H 的地址拷入当前程序此域中。在当前程序执行中，此域中的数据一般是不让改变的。当当前程序终止时，程序将把控制传送到该地址上，结果该地址处的程序便被执行。这时有两种情况，一种是当一个程序用 EXEC 中断指令功能调用另一个程序时，则地址就是紧跟在 EXEC 指令后的地址。当被调用子程序执行完后用 IRET 指令返回，则又继续执行父程序；另一种是一个返回到 COMMAND.COM 的程序，则控制传给 COMMAND.COM 的常驻部分，地址是常驻部分的 PSP；如果是一个批处理文件，就继续处理。否则，COMMAND.COM 就在瞬间内执行一次总检，确定是否需要重新

装入,然后发出系统提示符并且等待从键盘打入下一个命令。总而言之,该地址就是当前程序执行结束时返回到父程序(COMMAND.COM也是父程序)去的地址。如果没有这个地址,那末当子程序终止时就不知道如何返回到正确的断点上去。

这里正常结束地址是:CS=1489H,IP=034FH。程序L77.C和L78.C的执行结果会使你对此有更多的了解。

```
C>TYPE L77.C      /* 父程序 */
#include "stdio.h"
#include "process.h"
#include "dos.h"
main()
{
    int i;
    printf("parent —psp=%xH\n",—psp);
    for(i=0;i<=0x15;i++)
    printf("%x: %x ",i,(peekb(—psp,i) & 0x00ff));
    printf("\nstart\n");
    spawnl(P-WAIT,"l78.exe",NULL); /* 修改此句,分别按三种情况执行 */
    printf("endl \n");
}
```

```
C>TYPE L78.C /* 子程序 */
#include "stdio.h"
#include "process.h"
#include "dos.h"
main()
{
    int i;
    printf("child —psp=%xH\n",—psp);
    for(i=0;i<=0x15;i++)
    printf("%x: %x ",i,(peekb(—psp,i) & 0x00ff));
    printf("\n");
}
```

使用 spawnl(P-WAIT,"l78.exe",NULL);(父进程挂起直到子进程执行完成)所得输出结果(正常终止地址;父程序CS=1343H,IP=1DCH。而对子程序CS=2753H,IP=1f42H,它表示父程序在此处开始转去执行子程序的):

```
parent —psp=2743H
0:cd 1:20 2:0 3:a0 4:0 5:9a 6:f0 7:fe 8:1d 9:f0 a:dc b:1 c:43 d:13
e:4b f:1 10:43 11:13 12:56 13:1 14:43 15:13
start
child —psp=395dH
0:cd 1:20 2:0 3:a0 4:0 5:9a 6:f0 7:fe 8:1d 9:f0 a:42 b:1f c:53 d:27
e:4b f:1 * 10:43 11:13 12:56 13:1 14:43 15:13
endl
```

使用 spawnl(P-NOWAIT,"l78.exe",NULL);(子进程和父进程同时进行)所得输出结果(可惜子进程无输出,不管怎样,总是先执行父进程):

```
parent —psp=2743H
0:cd 1:20 2:0 3:a0 4:0 5:9a 6:f0 7:fe 8:1d 9:f0 a:dc b:1 c:43 d:13
e:4b f:1 10:43 11:13 12:56 13:1 14:43 15:13
start
end!
```

使用 `spawnl(P—OVERLAY,"l78.exe",NULL)`;(子进程覆盖父进程原来的存储位置)所得输出结果(父程序的“end!”未印出):

```
parent —psp=2743H
0:cd 1:20 2:0 3:a0 4:0 5:9a 6:f0 7:fe 8:1d 9:f0 a:dc b:1 c:43 d:13
e:4b f:1 10:43 11:13 12:56 13:1 14:43 15:13
start
child —psp=2743H
0:cd 1:20 2:f8 3:9f 4:0 5:9a 6:f0 7:fe 8:1d 9:f0 a:dc b:1 c:43 d:13
e:4b f:1 10:43 11:13 12:56 13:1 14:43 15:13
```

—6 0E 0F 10 11 = 8A 03 89 14

当用户在键盘上输入或显示字符串时按了 `Ctrl—Break` 键,屏幕上当前光标位置显示符号 ^ C ,并调用 `INT 23H` 矢量。DOS 提供的 `INT 23H` 使当前程序立即终止。但用户可以更改 `INT 23H` 服务例程(例如用 `ctrbrk()` 函数)。当程序执行时,DOS 将父进程中的 `INT 23H` 的地址保存于此域内,在当前程序终止时又恢复到父程序内设置的 `INT 23H` 服务例程的地址。

如果 `Ctrl—Break` 中断例程能在执行时首先保护所有的寄存器的内容,则它能以 `IRET` 指令结束该中断例程,然后继续执行程序。

Turbo C 的 `abort()` 函数是通过 DOS 中断 `INT 23H` 实现的。

这里保留的父程序里的 `Ctrl—Break` 地址是: `CS=1489H` , `IP=038AH`。

```
C>TYPE L79.C
#include "stdio.h"
#include "dos.h"
#include "process.h"
int c—break(void) /* 按 Ctrl—Break 键后执行的子程序 */
{
    int i;
    printf("PRESS CONTROL BREAK! \n");
    spawnl(P—WAIT,"L80.EXE",NULL);
    return(0); /* 当此句改成 return(1) 则 spawnl() 将不被执行 */
}

main()
{
    int i,i1,i2,j=0;
    ctrlbrk(c—break);
    printf("parent —psp=%x\n",—psp);
    for(i=0;i<=0x15;i++)
        printf("%x: %x ",i,(peekb(—psp,i) & 0x00ff));
    printf("\npress a key for start\n");
}
```

```

getch();
for(i1=0;i1<2;i1++)      /* 在执行此循环中间按 Ctrl-Break 键退出循环 */
    {j++;
        for(i2=0;i2<1000;i2++)
            printf("%d %d\n",i1,i2);      /* 没有此输出语句按 Ctrl-Break 无用 */
    }
printf("end! j=%d\n",j); /* 在执行循环时没按 Ctrl-Break 键此句被执行 */
}

```

C>TYPE L80.C

```

#include "stdio.h"
#include "dos.h"
main()
{
    int i;
    printf("child —psp=%x\n",—psp);
    for(i=0;i<=0x15;i++)
        printf("%x: %x ",i,(peekb(—psp,i) & 0x00ff));
    printf("\n Ctrl-Break, END ! \n");
}

```

/* 两程序分别编译,然后执行 L79.EXE,输出时对父程序的 INT 23H 有 CS=1343H, IP=14BH。而对子程序有 CS=1488H,IP=174EH,它表示在父程序 CS=1488H,IP=174EH 处 中断转去执行子程序。

```

parent —psp=1478
0:cd 1:20 2:0 3:a0 4:0 5:9a 6:f0 7:fe 8:1d 9:f0 a:dc b:1 c:43 d:13
e:4b f:1 10:43 11:13 12:56 13:1 14:43 15:13
press a key for start
0 0
0 1
.....
0 31 ^ C
PRESS CONTROL BREAK!
child —psp=271f
0:cd 1:20 2:0 3:a0 4:0 5:9a 6:f0 7:fe 8:1d 9:f0 a:3f b:27 c:88 d:14
e:4e f:17 10:88 11:14 12:56 13:1 14:43 15:13
Ctrl-Break, END !
*/

```

—7 12 13 14 15 = 17 03 89 14

例如,在磁盘 I/O 期间出现一个严重错误,DOS 则调用 INT 24H 中断矢量。当程序执行时,DOS 将 INT 24H 的地址保存于此域内,在程序终止时又恢复到此地址(可见它是一个严重错误跳出地址,或出口地址)。

这里CS=1489H,IP=0317H。

C>TYPE L81.C /* 父程序 */

```

#include "stdio.h"
#include "dos.h"

```

```

#include "process.h"
handler() /* 错误处理函数 */
{
spawnl(P-WAIT,"l82.exe",NULL); /* 开始调用 L82.EXE */
}
main()
{
int i;
printf("child —psp=%x\n",—psp);
for(i=0;i<=0x17;i++)
    printf("%x,%x ",i,(peekb(—psp,i) & 0x00ff));
printf("\n");
harderr(handler);
printf("这是一个试验:先在A 驱动器内不放磁盘,然后执行本程序\n");
printf("按任一键开始...\n");
getch();
fopen("A:\\FILE","r");
}

```

C>TYPE L82.C /* 子程序 */

```

#include "stdio.h"
#include "dos.h"
#define DISPLAY-STRING-INT21-9 0x09
int ax—x;
main()
{
char msg[25];
int drive,i;
ax—x=—AX;
if(ax—x<0){
    bdosptr(DISPLAY-STRING-INT21-9,"磁盘错! $",0);
    hardretn(-1);
}
drive=(ax—x & 0x00ff);
printf("\n");
sprintf(msg,"在%c 驱动器内无磁盘! \n $",'A'+drive);
bdosptr(DISPLAY-STRING-INT21-9,msg,0);
printf("child —psp=%x\n",—psp);
for(i=0;i<=0x17;i++)
    printf("%x,%x ",i,(peekb(—psp,i) & 0x00ff));
printf("\nHANDERR END ! \n");
return(2); /* 没有返回被调用程序,而是终止程序 */
}

```

/* 分别编译,然后执行 L81.EXE,对父程序输出 INT 24H 地址,CS=775EH,IP=156H,对子程序 CS=781CH,IP=257DH。程序执行完返回 DOS 后发现,功能键 F3 的重现 键入内容的功能被取消,如果不调用 L82 则仍起作用。

child —psp=780c

```
0:cd 1:20 2:0 3:a0 4:0 5:9a 6:f0 7:fe 8:1d 9:f0 a:dc b:1 c:5e d:77
e:4b f:1 10:5e 11:77 12:56 13:1 14:5e 15:77 16:48 17:77
```

这是一个试验:先在A驱动器内不放磁盘,然后执行本程序

按任一键开始...

(这时可看到 A 驱动器灯亮,稍等)

在A驱动器内无磁盘!

```
child —psp=8b19
```

```
0:cd 1:20 2:0 3:a0 4:0 5:9a 6:f0 7:fe 8:1d 9:f0 a:8d b:2d c:1c d:78
e:4b f:1 10:5e 11:77 12:7d 13:25 14:1c 15:78 16:f6 17:77
```

HANDERR END !

*/

—8 16 17 = 78 14

即 1478H,它是当前程序的父程序的 PSP 段址(但对 COMMAND.COM 此域包含它自身的 PSP 段址,这是例外)。

—9 18 ~ 2B = 01 01 01 00 02 FF ~ FF

该域包括 20 个字节,由 DOS 使用。它是一个【作业文件表】(简称 JFT, Job File Table),用户请求计算机完成一个任务可称为一个【作业】。有关文件的操作一般可以通过文件句柄来完成(不过, Turbo C 也用数据流访问文件),而文件句柄则是 JFT 的一个索引,每一个 JFT 的表项又依次是系统文件表(SFT)的一个索引。每个文件句柄对应 JFT 中一个表项,占一个字节。因一个 JFT 只留 20 表项,所以在一个程序中可以同时打开的文件数最多为 20 个,并且已有 5 个句柄已用于标准输入/输出设备文件(stdin、stdout、stderr、stdaux 和 stdprn,参见 STDIO.H),故用户程序实际上只能打开 15 个文件。

—10 2C 2D = 36 1A

被传递的环境的段地址,它是调用处理环境的一个拷贝。所谓拷贝是指在以后的程序中即使你又使用了 DOS 的 SET、PATH 或 PROMPT 命令,拷贝的环境也不会改变。在上例中,用

—D 1A36:0

已列出了环境的内容。

在 DOS 3.0 及以后版本,可执行文件名称紧接着环境变量来放置。

—11 2E 2F ~ 31 = 4C 01 49 19

DOS 使用。

—12 32 33 = 14 00

描述 JFT 的大小的域,它指明程序可用的最多文件句柄数,即同时能打开的文件数,一般为 14 H 个。改变这个值可以支持多于 20 的文件句柄,但需要重新定位 JFT,使之指向一个新的存储空间,该空间包含存放最大数目文件句柄所需的字节空间。

—13 34 35 36 37 = 18 00 3E 1A

JFT 的地址,段:偏移 = 1A3E:0018。通常情况下,段地址为 PSP 的段值,偏移为 18H,它正好指向 PSP 的 JFT。

此域的用处常用于突破文件句柄 20 的限制。方法如下:

1. 设置一内存区域,占用字节为 CONFIG.SYS 文件中

FILES=num

的值 num,它将作为新 JFT;

2. 在本域内填入该内存区域的地址;
3. 将描述 JFT 大小的域内的值设为 JFT 包括的文件数;
4. 将原 JFT 的 20 个字节的值拷贝到新 JFT, 末尾剩余的字节填以 FF, 表示未用字节。

—14 38~54 = FF FF FF FF 00 ~ CD 21 CB 00 00

文件控制块 (FCB) 在用户程序和 DOS 间传递操作文件的信息。它又分为两种形式: 标准文件控制块和扩充文件控制块。根据 FCB 使用情况又可分为未打开的 FCB 和打开的 FCB 两类。未打开的 FCB 仅包括一个驱动器号和可包含通配符 (* 和 ?) 的文件名; 打开的 FCB 包括通过打开 (或建立) 文件系统调用 (中断 INT 21H 的功能 0FH) 所有被填满的字段。

—15 55 56 57 58 59 5A 5B = 00 00 00 00 00 00 00

扩充 FCB 的 7 个字节。扩充 FCB 用于建立或查找磁盘目录项中具有特殊属性的文件。当 FCB 是扩充文件控制块时, 55=FFH, 56~5A 均为 0 (保留), 5B= 文件属性。

对标准文件控制块, 55=00H。

—16 5C 5D 5F ~ 6A 6B = 00 4C ~ 00 00

是未打开的第一个 FCB (文件控制块, 称第一个格式化参数区)。在 DOS 2.0 以前的版本, DOS 利用 FCB 来处理文件。命令处理器会分析命令行的第一个参数 (argv[1]), 参数被转换成适合 FCB 的 OPEN 或 CLEAR 调用形式。其第一个字节表示驱动器名: 01 为 A:, 02 为 B:, 03 为 C: 等, 00 表示缺省驱动器或没有使用 (参见《文件管理》一章)。

—17 6C 6D ~ 7A 7F = 00 20 ~ 00 00

是未打开的第二个 FCB (文件控制块, 称第二个格式化参数区)。命令处理器会命令行的第二个参数 (argv[2]) 被放在此处。若第一个 FCB 被打开, 则它的参数被复盖, 即从 5CH ~ 7FH 成为第一个文件打开的 FCB。

要注意的是, 命令行的全部参数有可能没有全包括在 FCB 中。

—18 80 = 05

磁盘传输区 (DTA) 的第一个字节, 也是非格式化参数区的第一个字节。但通常情况下, 该字节的内容为命令名 (文件名) 后面打入的全部字符的个数 (但输入字符后的回车符虽占一个字节, 但计算的字符个数不包括它)。

注意: 如果使用 PSP 位移 5CH 处的 FCB, 则随机记录的最后一个字节为位移 80H (即本字节)。

—19 81~FF = 20~00

磁盘传输区即非格式化参数区的其余字节。它包括命令名后面的全部字符, 但 DOS 的重定向符号和它后面的文件名将不包括在内。

GREP.COM 程序是利用命令行参数进行工作的, 该程序的第一个指令是通过一个子程序进行取 FCB 中的偏移 81H 开始的参数等操作。

用户程序不能改变 PSP 的 00H ~ 5CH 的任何内容。

21.5 .COM 文件和 PSP 的关系

.COM 文件被设计为在 64K 内执行。当它加载时, DOS 执行以下步骤:

- (1) 所有四个段寄存器 (CS, DS, SS, ES) 的值, 都设定在加载程序所占内存位置的开头;
- (2) IP 寄存器 (指令指针) 指到 100H;
- (3) SP 栈指针寄存器指到段末尾, 以便允许栈长为 PSP 偏移 06H 处的段值减去 100H;

(4) 一个全零值的字放在该栈顶,这是允许用户程序使用 RET 为最后的指令返回到 COMMAND.COM 程序,但是这种假设,用户就要保存用户程序的堆栈段和代码段;

(5) 所有用户内存都分配给程序,如果程序希望通过功能调用 INT 21H 中断的 4BH 功能 (即装入并执行程序 EXEC) 来调用另一程序,则它必须通过设置数据块的功能调用 (INT 21H 的 4 AH) 来释放一些内存,以便为调用的执行程序提供空间。

21.6 .EXE 文件和 PSP 的关系

(1) 一个 PSP 被建立在位移 0 处;

(2) DS 和 ES 寄存器被设置为指向 PSP;

(3) CS、IP、SS 和 SP 应置成 .EXE 头 (header,即 .EXE 最前面的几百个字节) 中所指定的值。

第二十二章 中断和中断函数

22.1 中断矢量

当程序中断 (Interrupts) 当前服务而转去执行一个服务例程以处理某一请求任务的事件称为【中断】。8086 保留内存最前面的 1KB (1024 个字节, 占用 RAM 的 0000:0000 到 0000:03FF) 作为 256 个 far 指针 k k 它们中的每一个称为【中断矢量】(或【中断向量】interrupt vector), 它们分别指向各自的中断服务子程序 (或称为【中断处理程序】), 每个中断服务例程又可以有多个不同的功能, 一般通过 AII 来设置, 最多也可达到 256 个), 或者说, 每 4 个字节存放一个中断服务例程的起始地址。每个中断服务例程的地址先由公式

中断服务例程起始字节 = 中断号 * 4

每个中断代表一种类型的中断, 这些中断用连续的数字编号, 10 进制的 0, 1, 2, ..., 10, 11, ..., 或 16 进制的 0H, 1H, 2H, ..., AH, BH, ... (或 0x0, 0x1, 0x2, ...) 等。中断类型编号简称【中断号】。求出其在 RAM 中的起始字节, 然后根据 0000: 起始字节中的值 (段: 偏移量。在内存中偏移量在段值前面, 低字节在高字节前面) 得到中断服务例程的开始地址。这种相互关联性形象地说明了中断矢量的矢量特性 (参见图 22-1)。

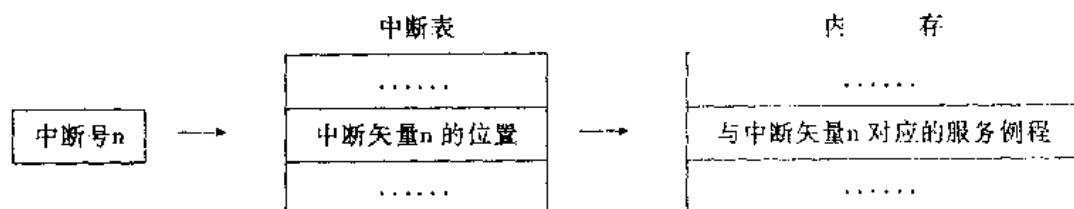


图 22-1

这 1KB 也称为【中断表】, 从 Turbo C 的观点看, 中断表也可以看成由指针构成的数组, 而每一个指针指向一个函数 (中断服务例程), 因此可假想将它定义为

```
void (*int—table[256])();
```

BIOS (基本输入输出系统) 是已固化在机器 ROM (只读存储器) 里的程序的总称, 这些程序在上电时引导机器和提供一些基本的输入 / 输出程序, 处理系统的全部中断。它直接与硬件接口。不同的机器与不同的操作系统可能有不同的 ROM BIOS, 或者说, 对同一个中断号, 其中断的内容可能有差别; 但就 MS-DOS 系统而言, 则大都是兼容的。ROM BIOS 中的程序只能通过中断来调用。以下列出中断表中部分中断矢量的位置和功能:

表 22-1

开始字节	中断类型	说	明
	以下为系统内部中断		
0000	INT 0H	DIV 指令遇到除数为 0 时的处理	

0004	INT 1H 8088 的单步方式 (single-stepping, 由 debug 使用) 等 单步 (陷阱) 中断是当标志寄存器中的单步陷阱标志 TF = 1 时, 微处理机就处于单步方式。此方式下每执行一条指令, 处理机就自动产生一个 INT 1H, 并把标志寄存器压入堆栈, 清除陷阱位 TF 和中断允许位 IF, 转入单步中断的处理 (服务) 过程。此时, TF = 0, 处理机不再是单步工作方式, 也不响应可屏蔽中断的请求。当单步中断处理结束时, 从堆栈中弹出原标志寄存器内容, 处理机又回到单步工作方式。
0008	INT 2H 不可屏蔽中断 (NMI), 出错时它使系统暂停
000C	INT 3H 8088 断点捕获 (trap) 处理 (由 debug 使用)
0010	INT 4H 溢出检测
0014	INT 5H 打印屏幕上的文本数据到打印机 (也称【打印屏幕】)。当装入专用打印驱动程序时也能打印图象。这一中断通常是在按下 Print Screen 键时由 INT 9H 中断处理例程启动, 但也可以被用户应用程序直接激发。中断例程地址在 F000: FF54 中, 所用状态在 0050: 0000 字节中。应用状态字节值的含义是 00H 不作用 01H 正在进行屏幕打印 FFH 上一次屏幕打印时遇到错误
0018	INT 6H 当 CPU 试图执行无效操作时产生, 返回地址指向无效指令起点
001C	INT 7H 没有装协处理器但遇到协处理器指令时, 自动调用可以用来模拟数学协处理器的例程
以下为 ROM BIOS 中断 (其中断矢量值由 BIOS 填入)	
0020	INT 8H 系统时钟, 定时器中断, 硬件中断
0024	INT 9H 键盘中断, 硬件中断。当键盘接收到数据时发生
0028	INT AH MIDI 接口呼唤 (如装了 Roland MIDI 接口) 或 LPT2 打印机呼唤 (如果系统中装有 EGA 或 VGA 卡)
002C	INT BH 异步通信控制 (串行口 COM2), 硬件中断
0030	INT CH 异步通信控制 (串行口 COM1), 硬件中断
0034	INT DH 表明硬盘动作完成或 LPT2 打印机呼唤, 硬件中断
0038	INT EH 软盘控制器完成一次操作后发生 (I/O 口 21H 位 6 必须为 0), 硬件中断
003C	INT FH 当打印机准备好时由 LPT1 (并行打印机) 适配器发生 (但大多数打印机并不可靠地产生这一中断), 硬件中断
0040	INT 10H 视频 I/O
0044	INT 11H 返回系统设备配置信息
0048	INT 12H 返回内存大小 (0040:0013 字中的内容, 单位为 KB) 到 AX 寄存器
004C	INT 13H 软盘或硬盘控制
0050	INT 14H 串行口 I/O
0054	INT 15H 磁带和系统 I/O 服务
0058	INT 16H 键盘 I/O
005C	INT 17H 打印机 I/O
0060	INT 18H 激发 ROM BASIC
0064	INT 19H 系统自举 (Bootstrap, 从软盘或硬盘引导装入)
0068	INT 1AH 时钟管理
006C	INT 1BH Control-Break 中断地址
0070	INT 1CH 计时滴答声 (每秒 18.2 次), 定时器中断调用的程序
0074	INT 1DH 供 6845 初始化用的显示参数
0078	INT 1EH 软盘参数

007C INT 1FH 供显示方式 4,5 和 6 用的图形字符扩展

以下为 DOS 中断

0080	INT 20H 程序正常结束并返回 DOS,为用户常用
0084	INT 21H DOS 调用,为用户常用
0088	INT 22H 调用程序结束地址
008C	INT 23H Control Break 出口地址
0090	INT 24H DOS 关键性错误处理
0094	INT 25H 绝对磁盘读,为用户常用
009C	INT 26H 绝对磁盘写,为用户常用
00A0	INT 27H 结束但驻留内存,为用户常用
00A4	INT 28H DOS 内部使用

22.2 中断过程和中断优先权

【硬件中断】是由设备（磁盘机、键盘或通信转换器等）直接送给微处理器一个请求信号时产生的,这种中断是可以用 cli 指令屏蔽（它将标志寄存器的中断标志位置成 0）,即此时任何硬件中断都不被响应;而用 sti 指令（它将标志寄存器的中断标志位置 1）可以重开硬件中断,即此时微处理机能够接受任何硬件中断的请求。

硬件中断的一个例子是,每当用户敲击或松开一个键时,键盘就产生一个硬件中断,转向一个保存在中断表 INT 9H 入口处的例程。

每个中断的大致过程是:

1. 把 FLAGS、CS 和 IP 寄存器的内容入栈;
2. 清中断允许标志位 IF,禁止进一步发生其它硬件中断;
3. 将指令指针 IP 和代码段寄存器 CS 的内容压入堆栈,保护中断返回地址;
4. 把中断服务例程的入口地址装入 CS 和 IP 寄存器;
5. 通过 far 跳转指令（即按新 CS 和 IP）执行中断服务例程;

6. 如果中断服务例程用 IRET 指令返回,则原挂起程序从被中断起继续执行,并恢复原标志寄存器状态。如果中断服务例程想借助于标志寄存器来返回信息,则不能用 IRET 指令返回。

当两个中断同时发生或者当一个中断正在处理时又发生了另一个中断,这时怎么办? 解决这中断优先权问题由 8259 控制器负责。一般说来,编号较小的中断比编号较大的中断优先处理（注意:有时也未必）。

查找中断表,保留原来的中断矢量,从而达到修改中断服务程序的功能。像 CCDOS 操作系统就是靠修改键盘、屏幕等中断服务例程实现的。值得指出,各厂商为了自身的利益也可能修改中断,扩充其基本功能,其结果使中断内容更丰富,但随之而来的是中断这个大家庭已经变得日趋庞大而又不标准,相互之间可能发生冲突,从而使软件互不兼容,读者应充分注意到这一点。

256 个中断矢量中有些还没有用,程序员可以编写自己的中断处理程序,然后用 far 指针设置中断矢量。自己编写的中断程序必须说明为 interrupt 类型。

22.3 部分库函数用到的中断

Turbo C 提供的一些库函数实际也采用或涉及了某个中断,下面列出它们的一些对照关

系。

中断	库函数	库函数	中断
INT 11H	biosequip()	—chmod()	INT 21H-43H
INT 12H	biosmemory()	—close()	INT 21H-3EH
INT 13H	biosdisk()	—creat()	INT 21H-3CH
INT 14H	bioscom()	—open()	INT 21H-3DH
INT 16H	bioskey()	—read()	INT 21H-3FH
INT 17H	biosprint()	—write()	INT 21H-40H
INT 1AH	biostime()	absread()	INT 25H
INT 21H	bdos()	abswrite()	INT 26H
INT 21H	bdosptr()	allocmem()	INT 21H-48H
INT 21H	intdos()	bdos()	INT 21H
INT 21H	intdosx()	bdosptr()	INT 21H
INT 21H-02H	putch()	bioscom()	INT 14H
INT 21H-07H	getch()	biosdisk()	INT 13H
INT 21H-0BH	kbhit()	biosequip()	INT 11H
INT 21H-0EH	setdisk()	bioskey()	INT 16H
INT 21H-19H	getdisk()	biosmemory()	INT 12H
INT 21H-1AH	setdta()	biosprint()	INT 17H
INT 21H-1AH	findnext()	biostime()	INT 1AH
INT 21H-1BH	getfatd()	chdir()	INT 21H-3BH
INT 21H-1CH	getfat()	country()	INT 21H-38H
INT 21H-25H	setvect()	creatnew()	INT 21H-5BH
INT 21H-27H	randbrd()	creattemp()	INT 21H-5AH
INT 21H-28H	randbwr()	dosexterr()	INT 21H-59H
INT 21H-29H	parsfnm()	dup()	INT 21H-45H
INT 21H-2AH	getdate()	dup2()	INT 21H-46H
INT 21H-2DH	settime()	findfirst()	INT 21H-4EH
INT 21H-2EH	setverify()	findnext()	INT 21H-1AH
INT 21H-2FH	getdta()	findnext()	INT 21H-4FH
INT 21H-31H	keep()	freemem()	INT 21H-49H
INT 21H-33H-00H	getcbrk()	getcbrk()	INT 21H-33H-00H
INT 21H-33H-01H	setcbrk()	getch()	INT 21H-07H
INT 21H-35H	getvect()	getcurdir()	INT 21H-47H
INT 21H-36H	getdfree()	getdate()	INT 21H-2AH
INT 21H-37H-00H	getswitchar()	getdfree()	INT 21H-36H
INT 21H-37H-01H	setswitchar()	getdisk()	INT 21H-19H
INT 21H-38H	country()	getdta()	INT 21H-2FH
INT 21H-39H	mkdir()	getfat()	INT 21H-1CH
INT 21H-3AH	rmdir()	getfatd()	INT 21H-1BH
INT 21H-3BH	setdate()	getftime()	INT 21H-57H-00H

INT 21H-3BH	chdir()	getpsp()	INT 21H-62H
INT 21H-3CH	—creat()	getswitchar()	INT 21H-37H-00H
INT 21H-3DH	—open()	getvect()	INT 21H-35H
INT 21H-3EH	—close()	getverify()	INT 21H-54H
INT 21H-3FH	—read()	harderr()	INT 24H
INT 21H-40H	—write()	intdos()	INT 21H
INT 21H-41H	unlink()	intdosx()	INT 21H
INT 21H-42H	lseek()	ioctl()	INT 21H-44H
INT 21H-43H	—chmod()	isatty()	INT 21H-44H-00H
INT 21H-44H	ioctl()	kbit()	INT 21H-0BH
INT 21H-44H-00H	isatty()	keep()	INT 21H-31H
INT 21H-45H	dup()	lock()	INT 21H-5CH-00H
INT 21H-46H	dup2()	lseek()	INT 21H-42H
INT 21H-47H	getcudir()	mkdir()	INT 21H-39H
INT 21H-48H	allocmem()	parsfnm()	INT 21H-29H
INT 21H-49H	freemem()	putch()	INT 21H-02H
INT 21H-4AH	setblock()	radbrd()	INT 21H-27H
INT 21H-4EH	findfirst()	randbwr()	INT 21H-28H
INT 21H-4FH	findnext()	rename()	INT 21H-56H
INT 21H-54H	getverify()	rmdir()	INT 21H-3AH
INT 21H-56H	rename()	setblock()	INT 21H-4AH
INT 21H-57H-00H	getftime()	setcbk()	INT 21H-33H-00H-01H
INT 21H-57H-01H	setftime()	setdate()	INT 21H-3BH
INT 21H-59H	dosexterr()	setdisk()	INT 21H-0EH
INT 21H-5AH	creattemp()	setdta()	INT 21H-1AH
INT 21H-5BH	creatnew()	setftime()	INT 21H-57H-01H
INT 21H-5CH-00H	lock()	setswitchar()	INT 21H-37H-01H
INT 21H-5CH-01H	unlock()	settime()	INT 21H-2DH
INT 21H-62H	getpsp()	setvect()	INT 21H-25H
INT 24H	harderr()	setverify()	INT 21H-2EH
INT 25H	absread()	unlink()	INT 21H-41H
INT 26H	abswrite()	unlock()	INT 21H-5CH-01H

22.4 BIOS 工作区（或数据区，不同机器可能不同，仅供参考）

内存从 0040:0000 到 0040:FFFF 的 RAM 被作为 ROM BIOS 的工作空间（也有把 0040:0000 ~ 0050:0000 称为 ROM 通讯区，0050:0000 ~ 0060:0000 称为 DOS 通讯区）。必需的子程序所共用的系统变量也放在这个区域内。该区域被称为 BIOS 工作区和数据交换区。用户可从该区域中的一些数据了解 DOS 当前状态。对区域中的数据用户一般不要轻易改动。

用下列程序可印出 BIOS 工作区和数据交换区的内容。

```
C>TYPE TESTBIOS.C
```

```
#include "dos.h"
```

```
main(){
```

```

int i,n;
for(i=0;i<1024;i+=2)
{
    n=peek(0x40,i);
    printf("%#02x=%#06x\n",i,n);
}
}

```

下面为刚进入 DOS 5.0 和联想汉字系统时印出结果,可以看出它们之间的不同之处。有些内存单元中的数据是固定不变的,有些则是随操作不断变动的。

表 22-3

非汉字系统	汉字系统	说 明
00=0x03f8	00=0x03f8	第一个串口 (COM1) 基地址
0x2=0x02f8	0x2=0x02f8	第二个串口 (COM2) 基地址
0x4=0x03e8	0x4=0x03e8	第三个串口 (COM3) 基地址
0x6=000000	0x6=000000	第四个串口 (COM4) 基地址
0x8=0x0378	0x8=0x0378	第一个打印口基地址
0xa=000000	0xa=000000	第二个打印口基地址
0xc=000000	0xc=000000	第三个打印口基地址
0xe=000000	0xe=000000	第四个打印口基地址
0x10=0x4663	0x10=0x4663	0x10 位7~6:已装软盘数减1(如位0为1时) 位5~4:初始显示方式 00 EGA、VGA 或 PGA 01 40×25 彩色 10 80×25 彩色 11 80×25 单色 位3~2:对 PC 机为母板上 RAM16 排数 对 XT 机为母板上 RAM64K 排数 位1: 为1表示已装 80x87 协处理器 位0: 如为1由位7~6指出已装软盘数
		0x11 位7~6:已装并行口数 位5: 已装内部调制解调器或附加串行打印机数 位4: 已装游戏口 位3~1:已装串行口数 位0: 已装 DMA 支撑
0x12=0x80ff	0x12=0x80ff	0x12 为电源自检状态,0x13~0x14 为内存尺寸 0100H=256K,0200H=512K,0280H=640K
0x14=0x0002	0x14=0x0002	0x15 ~0x16 为专用测试区
0x16=000000	0x16=000000	0x17 为键盘状态字节,0x18 为 CapsLock 键状态
0x18=000000	0x18=000000	0x19 为 Alt + 小键盘数字键暂存值
0x1a=0x0020	0x1a=0x002e	指向键盘中断缓冲区头部的指针
0x1c=0x0020	0x1c=0x002e	指向键盘中断缓冲区尾部的指针
0x1e=0x1c0d	0x1e=0x343e	键盘中断缓冲区
.....		

0x3c=0x1474	0x3c=0x0635
0x3e=000000	0x3e=000000
0x40=0x0068	0x40=0x004c
0x42=0x00c0	0x42=0x00c0
0x44=000000	0x44=000000
0x46=000000	0x46=000000
0x48=0x0300	0x48=0x0300
0x4a=0x0050	0x4a=0x0050
0x4c=0x1000	0x4c=0x1000
0x4e=000000	0x4e=000000
0x50=0x1800	0x50=0x0300
0x52=000000	0x52=000000
0x54=000000	0x54=000000
0x56=000000	0x56=000000
0x58=000000	0x58=000000
0x5a=000000	0x5a=000000
0x5c=000000	0x5c=000000
0x5e=000000	0x5e=000000
0x60=0x0d0e	0x60=0x0d0e
0x62=0xd400	0x62=0xd400
0x64=0x2903	0x64=0x2903
0x66=0x8830	0x66=0x8830
0x68=0x8705	0x68=0x8705
0x6a=0x0090	0x6a=0x0090
0x6c=0x9c9b	0x6c=0x0222
0x6e=0x000f	0x6e=0x0010
0x70=000000	0x70=000000
0x72=0x1200	0x72=0x1200
0x74=0x0100	0x74=0x0100
0x76=000000	0x76=000000
0x78=0x1414	0x78=0x1414
0x7a=0x1414	0x7a=0x1414
0x7c=0x0101	0x7c=0x0101
0x7e=0x0101	0x7e=0x0101
0x80=0x001e	0x80=0x001e
0x82=0x003e	0x82=0x003e
0x84=0x1018	0x84=0x1018
0x86=0x6000	0x86=0x6000
0x88=0x1109	0x88=0x1109

键盘中断缓冲区

磁盘寻道和马达状态

0x40 为马达暂停, 0x41 为磁盘驱动器返回码

磁盘驱动器安装状态

磁盘驱动器安装状态

磁盘驱动器安装状态

0x48 为磁盘驱动器安装状态, 0x49 为当前视频模式

屏幕列数

视频缓冲区中显示页的长度(字节数)

视频缓冲区内显示数据的开始(首)地址

CRT 第一页光标位置

CRT 第二页光标位置

CRT 第三页光标位置

CRT 第四页光标位置

CRT 第五页光标位置

CRT 第六页光标位置

CRT 第七页光标位置

CRT 第八页光标位置

当前光标模式(光标的起始行和终止行)

0x62 为活动页开始地址, 0x63 和 0x64 为 CRT 地址寄存器口地址

0x65 为 CRT 控制器模式, 位 5 指出字符闪动或背景增亮状态

0x66 当前调色板颜色, 0x67 为磁带数据

磁带数据

磁带数据

定时器低字(脉冲四字节计数)

定时器高字

0x70 为定时溢出标志, 启动时为 0, 时钟运转后加 1(但不再增加), 0x71 为 Ctrl-Break 键状态

0x72 为 Alt-Ctrl-Del 复位标志状态

硬盘数据区

硬盘数据区

扩展键盘缓冲区头指针(XT)

扩展键盘缓冲区尾指针(XT)

0x85 为当前字体高度

EGA/VGA 在用 INT 10H 功能 1 做光标模拟时 0x87 的位 0 = 0, 如果不做光标模拟, 则在调用该功能前将该位置成 1。对 EGA 必须用程序直接更改此位, 而对 VGA 可用 INT 10H 的功能 12H 更改它。

光标模拟可以让程序调用 INT 10H 功能 1 来改变

光标大小时对 CRTC 里的值作些修改,以至不管字形点阵大小如何都可在各种显示卡上执行。

0x8a=0x810b	0x8a=0x810b
0x8c=0x0150	0x8c=0x0150
0x8e=0x3300	0x8e=0x3300
0x90=0x0202	0x90=0x0202
0x92=000000	0x92=000000
0x94=000000	0x94=000000
0x96=0x1010	0x96=0x1010
0x98=000000	0x98=000000
0x9a=000000	0x9a=000000
0x9c=000000	0x9c=000000
0x9e=000000	0x9e=000000
0xa0=000000	0xa0=000000

0x96 为键盘方式和类型标志,0x97 为 LED 灯标志

.....

22.5 调用中断库函数

一 开关中断

. 屏蔽中断 (或关闭中断)

—1 disable()

—2 —cli—()

. 开放中断

—3 enable()

—4 —sti—()

二 读段寄存器值

. 读当前段寄存器值,放到 SREGS 结构中

—5 segread()

三 产生软中断

. 直接产生一个软中断

—6 —int—()

—7 geninterrupt()

. 通过 REGPACK 类结构产生一个 8086 软中断

—8 intr()

. 产生 8086 软中断,将 segregs 成员拷贝到 DS、ES 寄存器中

—9 int86x()

. 产生 8086 软中断,无 segregs 参数

—10 int86()

. 产生 INT 21H 中断的一个功能,有 segregs 参数

—11 intdosx()

. 产生 INT 21H 中断的一个功能,无 segregs 参数

—12 intdos()

. 直接指定功能号、AL 和 DX 值的 INT 21H 调用

—13 bdos()

常用于小数据内存模式 (Tiny、Small 和 Medium 模式)

. 直接指定功能号、AL 和 DS、DX 指针值的 INT 21H 调用—14 bdosptr()

适用于大数据内存模式 (Compact、Large 和 Huge)

四 ctrl break 处理

. 取得当前 BREAK 开关的设置状态

—15 getcbrk()

- . 设置 BREAK 开关 —16 setcbrk()
- . 建立一个新的 INT 23H 中断矢量 —17 ctrlbrk()

五 开关字符

- . 获得当前 MS-DOS 的开关字符值 —18 getswitchar()
- . 将当前 MS-DOS 开关字符值设置为 ch 的值 —19 setswitchar()

六 磁盘验证标志

- . 获得磁盘写验证标志 —20 getverify()
- . 设置磁盘写验证标志 —21 setverify()

七 机器码放入源程序

- . 将机器码依次放在源程序中 —22 `—emit—()`

八 BIOS 中断

- . 检查系统设备 (BIOS 中断 INT 11H) —23 biosequip()
- . 查询系统内存大小 (BIOS 中断 INT 12H) —24 biosmemory()
- . 低级磁盘 I/O (BIOS 中断 INT 13H) —25 biosdisk()
- . 异步通讯 (BIOS 中断 INT 14H) —26 bioscom()
- . 键盘操作 (BIOS 中断 INT 16H) —27 bioskey()
- . 打印机操作 (BIOS 中断 INT 17H) —28 biosprint()
- . 时钟函数 (BIOS 中断 INT 1AH) —29 biostime()

九 视频中断 (参见《视频函数》一章)

- 1 void —Cdecl disable(void);
- 2 void —Cdecl `—cli—`(void);
- `#define disable() —cli—()`

在 DOS.H 中可以同时看到 disable() 有两个定义,一个是函数,一个是宏。从 DOS.H 的开头可以看到:

```
#if —STDC—          /* 如果 —STDC— 有定义 */
#define —Cdecl       /* 则 —Cdecl 定义为空 */
#else                /* 否则 */
#define —Cdecl cdecl /* 定义 —Cdecl 为 cdecl */
#endif               /* 定义结束 */
```

如果在编译时对 TCC 用了 -A(或对集成环境选了菜单 O/C/Source/ANSI Keyword only) 时,则程序中只有与 ANSI 标准的关键字一样的关键字才被当作关键字处理, Turbo C 扩展的关键字不被认为是关键字。此时, ANSI 标准预定义的宏 `—STDC—` (STDC 可以理解为标准的 C) 有值 1, 否则无定义。

cdecl 是一个修饰符,它并不包括在 ANSI 中。当编译程序时对 TCC 选用了 -p(或对集成环境选择了菜单 O/C/Codegeneration/Calling convention Pascal) 时, cdecl 没有定义, 否则有值 1(未用 -p 或选择 O/C/Codegeneration/Calling convention C)。

C>TYPE INT1.C

```
#include "dos.h"
main()
{ disable(); }
```

用 CPP.EXE 分析时变成

```
#include "dos.h"
main()
{ __cli__(); }
```

说明它是一个【屏蔽中断】（或称【关闭中断】）的宏，清除硬件中断标志；另一方面，它又随 ANSI 和 -p 开关而变化。__cli__() 函数除了不可屏蔽中断 (NMI) 外，将屏蔽其它所有的硬件中断。对 8086 结构它直接转化为机器指令 cli。它没有返回值，只适用于 8086 结构。通常，程序中在它稍后之处应引用 enable()。

```
—3 void __cdecl enable(void);
—4 void __cdecl __sti__(void);
#define enable() __sti__()
```

【开放中断】的宏，设置硬件中断标志。__sti__() 函数允许任何设备中断发生。对 8086 结构，它直接转化为机器指令 sti。它没有返回值，只适用于 8086 结构。通常，程序中在它之前应引用 disable()。—5 void __cdecl segread(struct SREGS *segp);

读当前段寄存器 (CS、ES、SS 和 DS) 的值，放到 segp 所指的结构中。结构 SREGS 在 DOS.H 中定义为：

```
struct {
    unsigned int es; /* 附加段寄存器 */
    unsigned int cs; /* 代码段寄存器 */
    unsigned int ss; /* 栈段寄存器 */
    unsigned int ds; /* 数据段寄存器 */
};
```

本调用可与 intdosx() 或 int86x() 一起使用。它无返回值，只适用于 MS-DOS。

C>TYPE INT2.C

```
#include "dos.h"
#include "errno.h"
main()
{
    struct SREGS s;
    union REGS r;
    unsigned char near *filename="c:\\tc\\file.c";
    int _retu;
    r.h.ah=0x41; /* INT 21H 功能 41H 为删除指定文件 */
    s.ds=-DS; /* 入口:DS:DX = 文件名 */
    /* segread(&s); */ /* 就 -DS 值而言此句同 s.ds=-DS;等效 */
    r.x.dx=(unsigned)filename; /* unsigned 必须要有 */
    _retu=intdosx(&r,&r,&s);
    if(r.x.cflag &= 0)
```

```

    {printf("error=%d errno=%d —doserrno=%d\n",—retu,errno,—doserrno);
      exit(1);}
    printf("0x%x 0x%x 0x%x 0x%x\n",s.es,s.cs,s.ss,s.ds);
}

```

/* 如果文件存在,用 F7 调试输出:0x6fe 0x8d97 0xffd6 0x8a5f.

否则输出: error=2 errno=2 —doserrno=2 */

—6 void —Cdecl —int—(int interruptnum);

—7 #define geninterrupt(i) —int—(i)

—int—(interruptnum) 将产生一个软中断 interruptnum,返回值依赖于被调用的中断。geninterrupt(i) 则是具有同样功能的宏。它们只适用8086结构。编译程序将把—int—(interruptnum)直接转化成机器指令int interruptnum,从而产生高效的嵌入代码,并避免函数调用的开销。

DOS 提供了两个便于使用 BIOS 的模块:IBMBIO.COM 和 IBMDOS.COM(或 IO.SYS 和 MSDOS.SYS 等)。这两个模块增加了许多必要的检测,因此使用它们提供的中断即【DOS 中断】要比 BIOS 的对应部分的中断(INT 20H ~ INT 3FH)要好用。

IBMBIO.COM 提供了和 BIOS 的低级接口,是一个 I/O 处理程序,它方便了外设与内存的数据交换;IBMDOS.COM 包含文件管理程序和某些服务功能。当用户程序请求 I/O 时,通过寄存器和控制块把高级信息传送给 IBMDOS.COM。为了完成请求功能,IBMDOS.COM 把信息翻译成一个或多个 IBMBIO.COM 的调用。它们之间的关系见图 22-2。

程序I/O 请求 ↔ IBMDOS.COM ↔ IBMBIO.COM ↔ BIOS ↔ 外部设备

图 22-2

【软中断】(或软件中断)是处理机本身执行 INT 指令而产生的中断,或者说是由程序产生的。例如,下面我们称【DOS 功能调用】都通过单一的软中断k k INT 21H 来产生,因此对它可写出:

—int—(0x21) 或 genterrupt(0x21)

对其它 ROM BIOS 中断调用像

—int—(0x10) 或 genterrupt(16)

等。软中断的含意显然较广,它包括 BIOS 中断和 DOS 中断,而且它并不受中断标志位的状态影响。

C>TYPE INT3.C

#include "dos.h"

main()

{ disable();

 getch();

 enable();

}

程序 INT3.C 中与键盘相关的函数 getch() 依然起作用,实际上它是靠geninterrupt (0x21) 实现的,而不是靠硬件中断实现的。

—8 void —Cdecl intr (int intno,struct REGPACK * preg);

改变软中断接口函数。它产生一个 INT intno 的 8086 软中断。在调用函数前, preg 所指

寄存器作为输入寄存器,中断之后函数将当前寄存器值存入 preg 所指寄存器,包括标志寄存器的值。函数本身不返回任何值,它只适用于 MS-DOS,工作在 8086 系列处理器上。结构 REGPACK(意为 Register pack)在 DOS.H 中定义为:

```
struct REGPACK{
    unsigned r—ax,r—bx,r—cx,r—dx;
    unsigned r—bp,r—si,r—di,r—ds,r—es,r—flags;
};

C>TYPE INT4.C
#include "dos.h"
main()
{
    struct REGPACK p;
    p.r—ax=0x200;
    p.r—dx=0x0a04;
    p.r—bx=0;
    intr(0x10,&p);
    printf("调用中断INT 10H 功能2H,将光标移到第10行第4列\n");
} /* printf()显示的字符串将从第10行第4列开始显示 */
—9 int —Cdecl int86x (int intno,union REGS *inregs,
                        union REGS *outregs,struct SREGS *segregs);
```

通用 8086 软中断接口,即为 ROM BIOS 软中断调用的总入口(即它常用来调用 IBM PC 的 ROM 里的程序),它执行一个由中断号 intno 指定的 8086 软中断。

在执行软中断前,int86x 将联合 inregs 中的内容拷贝到处理器的寄存器中,并将 segregs—>ds 和 segregs—>es 的值拷贝到 DS 和 ES 寄存器中,这样就允许哪些使用远指针或使用大型数据存储模式的程序去指定软中断过程中使用的段地址。换句话说,它允许激活 8086 软中断来获取一个不同于缺省数据段的 DS 值,或取 ES 中的参数。

返回时将当前寄存器的值拷贝到联合 outregs 中,并将系统进位标志(CF)的值拷贝到 outregs—>x.cflag 中,将 8086 标志寄存器的值拷贝到 outregs 的 x.flags 中,并恢复 DS,设置 segregs—>es 和 segregs—>ds 的值为对应段寄存器的值。如果进位标志被置位,就表明发生了错误,AX 寄存器的值被返回。inregs 和 outregs 可以指向同一结构。

它只适用于 MS-DOS,用在 8086 系列处理器上。

联合 REGS 在 DOS.H 中定义为

```
struct WORDREGS{
    unsigned int ax,bx,cx,dx,si,di,cflag,flags;
};
/* ax 是累加器,bx 是基址寄存器,cx 是计数寄存器,dx 是数据寄存器 */
/* si 是源索引寄存器,di 是目的索引寄存器,cflag 是进位标志(Carry) */
/* Flag 即 CF,flags 是 8086 标志寄存器。 */

struct BYTEREGS {
    unsigned char al,ah,bl,bh,cl,ch,dl,dh;
}; /* al 和 ah 分别是 ax 的低位和高位,等等。 */

union REGS {
    struct WORDREGS x;
```

```

    struct BYTEREGS h;
},

```

Turbo C 为了能调用任何一个软中断,调用参数和返回参数都是通过寄存器来传递的,为此它定义了上述数据结构,它们包括了除 IP(指令指针寄存器)以外的寄存器。

—10 int —Cdecl int86(int intno, union REGS *inregs, union REGS *outregs);

也是通用 8086 软中断接口,它与 int86x() 不同之处只是没有参数 segregs,当然也谈不上恢复 DS。

它只适用于 MS-DOS,用在 8086 系列处理器上。

intdos 或 intdosx 均用输入的 cflag 进位标志判断是否出错,但是对 int86x 有些特殊情况,参考下列程序 INT5.C 会看到利用进位标志判别出现的问题:

```

C>TYPE INT5.C
#include "dos.h"
#include "errno.h"
main() /* 本示例纯粹是为了说明函数功能而对应用程序不具实用性 */
{
    struct SREGS s;
    union REGS r;
    struct date d1, *d=&d1; /* 修改此处是为了避免出现编译警告 */
    int —retu,y,xx; /* 增加 xx */
    unsigned char m,t;
    r.h.ah=0x2a;
    —retu=int86x(0x21,&r,&r,&s); /* 修改前为 —retu=intdosx(&r,&r,&s) */
    xx=—AX; /* 增加一句 */
    printf("cflag=%d —AX=0x%x\n",r.x.cflag,xx); /* 增加的一句 */
    if(r.x.cflag != 0)
        (printf("return=%d errno=%d —doserrno=%d\n",—retu,errno,—doserrno),
        exit(1));
    printf("0x%x 0x%x 0x%x 0x%x\n",s.es,s.cs,s.ss,s.ds);
    ((int *)d)[0]=r.x.cx;
    ((int *)d)[1]=r.x.dx;
    y=d->da—year;
    m=d->da—mon;
    t=d->da—day;
    printf("%d 年 %u 月 %u 日\n",y,m,t);
}
/* 用 F7(Turbo C 调试编译器调试)或 Ctrl-F9 执行,输出:

```

```

    cflag=0 —AX=0x2a05
    0xc710 0x9a06 0x8d 0xfe00
    1992 年 7 月 8 日

```

但直接将其编译后执行,有

```

    cflag=1 —AX=0x2a05
    return=10757 errno=19 —doserrno=87

```

两者结果竟绝然相反!这是我们看到的一个调试正确而运行不正确的例子。

errno 是 errno.h 中规定的 DOS 错误码,—doserrno 是 MS-DOS 错误号,它经某种转换后将值存入 errno。这里它们均指“无效参数”。

值得指出的是,如果放弃用 cflag 判别,即不管 cflag 判别结果如何,则函数都能得到正确的结果。

*/

```
—11 int —cdecl intdosx (union REGS *inregs, union REGS *outregs,
                        struct SREGS *segregs);
```

通用 MS-DOS 中断接口,它激活 INT 21H 中断的一个功能。它相当 int86x() 函数的一个特例,即中断号 intno=0x21。inregs 为指向输入寄存器的指针, inregs->h.ah 中放 INT 21H 的功能号。intdosx() 在调用 DOS 中断 INT 21H 的功能前,首先将 inregs 中的寄存器值拷贝到相应的寄存器中。将 segreg->ds 和 segreg->es 的值拷贝到对应的寄存器中。这一特性允许使用远指针或大型数据模式的程序在函数执行过程中指定所使用的段。

返回时将当前寄存器的值拷贝到联合 outregs 中,并将系统进位标志 (CF) 的值拷贝到 outregs->x.cflag 中,将 8086 标志寄存器的值拷贝到 outregs 的 x.flags 中,并恢复 DS,设置 segreg->es 和 segreg->ds 的值为对应段寄存器的值。如果进位标志被置位 (不等于 0),就表明发生了错误,AX 寄存器的值被返回。inregs 和 outregs 可以指向同一结构。

只适用于 MS-DOS。

```
C>TYPE INT6.C
#include "dos.h"
#include "errno.h"
main()
{
    struct SREGS s;
    union REGS r;
    struct date *d;
    int —retu,y;
    unsigned char m,t;
    r.h.ah=0x2a; /* INT 21H 功能 2AH 为取日期 */
    —retu=intdosx(&r,&r,&s);
    if(r.x.cflag != 0)
        {printf("return= %d\n",—retu);
        exit(1);}
    printf("0x%x 0x%x 0x%x 0x%x\n",s.es,s.cs,s.ss,s.ds);
    ((int *)d)[0]=r.x.cx;
    ((int *)d)[1]=r.x.dx;
    y=d->da—year;
    m=d->da—mon;
    t=d->da—day;
    printf("%d 年 %u 月 %u 日 \n",y,m,t);
}
/* 用 F7 调试,在调用 intdosx 前有:
    s,rx:{es,0x903D,cs,0x7500,ss,0xBA11,ds,0x823F}
    —ES,x,0x8A72
    —DS,x,0x8A72
    在调用intdosx 后有:
    s,rx:{es,0x903D,cs,0x7500,ss,0xBA11,ds,0x823F}
```

```

—ES,x,0x903D
—DS,x,0x8A72
*/

```

```

—12 int —Cdecl intdos(union REGS *inregs, union REGS *outregs);

```

它的功能除无 `segregs` 参数外,其余和 `intdosx()` 完全相同。

```

—13 int —Cdecl bdos(int dosfun, unsigned dosdx, unsigned dosal);

```

DOS 中断 INT 21H 的功能调用函数。`dosfun` 是 INT 21H 的功能号, `dosdx` 是 DX 寄存器中存放的值, `dosal` 是 AL 寄存器中存放的值。如果 `dosdx` 或 `dosal` 未用,通常可填 0。它常用在小数据内存模式 (Tiny、Small 和 Medium 模式) 中。返回值在 AX 中,供 DOS 系统使用。

注意:当系统调用要求的参数不只是 DX 和 AL 时,或者返回值也不只是 AX 中的值时,就应用 `intdos` 函数,而不能用 `bdos()` 函数。函数名 `bdos` 可以理解为基本 (basic)DOS 功能调用。

```

C>TYPE INT7.C

```

```

#include "stdio.h"
#include "dos.h"
#include "ctype.h"
main()
{
    int drive;
    char curdrive;
    curdrive='a'+bdos(0x19,0,0); /* INT 21H 的功能 19H 是取当前驱动器号 */
    drive=—AL; /* AL 中返回当前缺省的驱动器号 */
    printf("当前驱动器是%c: 即是%c:\n", toupper(curdrive), toupper(drive));
} /* 输出:当前驱动器是C: 即是C: */

```

```

—14 int —Cdecl bdosptr(int dosfun, void *argument, unsigned dosal);

```

它与 `bdos` 功能相同,也是 DOS 中断 INT 21H 的功能调用,不同的是 `bdos` 使用整型参数,而 `bdosptr` 使用指针参数;此外, `bdosptr` 在大数据模式 (Compact、Large 和 Huge) 中使用特别有效。`dosfun` 是 INT 21H 的功能号;在小数据模式中, `bdosptr` 将参数 `argument` 传给 DX,而在大数据模式中,它给出 DS:DX 的值,供系统调用使用。

调用成功时返回值为 AX 中值,否则返回 -1,并置 `errno` 和 `—doserrno` 为相应值。

```

C>TYPE INT8.C

```

```

#include "stdio.h"
#include "dos.h"
main()
{
    FILE *fp;
    fp=fopen("c:\\g.h", "r"); /* 出错时 fopen 返回 NULL */
    if(fp==NULL)
    { /* INT 21H 的功能 9H 是在屏幕上打印字符串,入口:DS:DX 指 */
        /* 向内存中以 0x24 (即字符 $) 结束的字符串 */
        bdosptr(0x09, "文件打开错误... 文件不存在! $", 0);
        exit(1); /* 退出 */
    }
}

```

```

}
printf("文件已打开,按任一键开始处理文件...\n");
getch();
/* 文件处理语句 */
}

```

—15 int —Cdecl getchbrk(void);

取得当前 BREAK 开关的设置状态,开关为 ON 时, getchbrk() 返回值 1,表示 MS-DOS 系统在每次 DOS 功能调用时都将检测 Ctrl-Break 键;否则,开关为 OFF 时, getchbrk() 返回 0,MS-DOS 系统只对控制台、打印机或通信设备进行 I/O 时检测 Ctrl-Break 键,对其它 DOS 功能调用不检测 Ctrl-Break 键。实际是调用 INT 21H 的功能 33H 的子功能 00H,即

—AX=0x3300;geninterrupt(0x21);return(—DL);

只适用于 MS-DOS。(例子参见《键盘与鼠标》一章)。

—16 int —Cdecl setcbrk(int cbrkvalue);

设置 control break 开关,当 cbrkvalue 值为 1 时,开关为 ON,每次系统调用时都检测 Ctrl-Break 键;为 0 时开关为 OFF,只在控制台、打印机或通讯设备进行 I/O 时检测 Ctrl-Break 键。它实际是 DOS 中断 INT 21H 的功能 33H 的子功能 01H:

—AX=0x3301,—DL=cbrkvalue;geninterrupt(0x21);return(—DL);

只适用于 MS-DOS。(例子参见《键盘与鼠标》一章)。

—17 void —Cdecl ctrlbrk(int —Cdecl(*hander)(void));

它设置一个新的由 hander 所指的 control break 处理函数,这个函数的地址便是 control break 新的出口地址,该函数可以是任一操作或系统调用。它实际修改了中断矢量 INT 23H 的地址为这个函数的入口地址,建立了一个新的 INT 23H 中断矢量。

新的 control break 函数并不直接被调用,一旦发现用户按了 Ctrl-Break 键时才被调用。它并不要求一定要返回,此时没有 return 语句或 return(i),这里 i 是一个非 0 值。如果使用了 return(i),那么该函数可以通过长跳转函数 longjmp() 返回调用它 程序中任一点,这一功能即让 Ctrl-Break 键不起作用有时是有用的。如果它使用了 return(0) 则终止程序执行。

ctrlbrk() 无返回值。为能对整个程序执行中使 ctrlbrk() 起作用,一般应将它放在程序头部。

当用户敲击 Ctrl-Break 键后,键盘中断立即建立一个标志(在内存位置 0040:0071)请求启动 control break 的处理例程。但只在程序使用有识别标志能力的 DOS 功能调用时,控制权才会交给 control break 例程,INT 23H 所指的中断例程才被启动。一般只有标准的 DOS 输入/输出功能(如 INT 21H 的功能 1H~5H、8H~CH)具有这种识别标志。用户可以在 AUTOEXEC.BAT 文件或在 CONFIG.SYS 文件中设置

BREAK=ON

命令,这样当任一 DOS 功能调用时,都要检测 Ctrl-Break 是否被按下。当然,这样会减慢程序执行速度。为此你可以把 BREAK=ON 改成 BREAK=OFF,只在控制台、打印机或通信设备进行 I/O 时检测 Ctrl-Break 键。

在 DOS 命令行上使用 C>BREAK 便会看到

BREAK is on

或

BREAK is off

的结果。Turbo C 用 `ctrlbrk()` 改变缺省的 INT 23H 的服务例程,用 `setcbrk()` 来设置 BREAK 开关状态 (on 或 off),而用 `getcbrk()` 来查询 BREAK 开关状态。

对敲击 Ctrl-Break 中止键的处理有两种方法:启动【BIOS 的 control break 中止处理例程】或【DOS 中止处理例程】。两者有三方面不同:

1. 缺省的 BIOS 的中止处理例程实际上是 BIOS 的键盘中断例程,它在用户任何时候按 Ctrl-Break 键都能检测到中止键的出现,而 DOS 中止处理例程只在 DOS 功能调用时才感受到中止键;

2. 缺省的 BIOS 的中止处理例程什么事也不做。如果希望它检测到中止键时采取一些措施,那就必须装入一个例程(它既涉及 INT 9H,又涉及 INT 23H 中断)。而对一个【DOS 中止处理例程】,是当用户按了 Ctrl-Break 键后,MS-DOS 检测到这种中止键时,会回应字符 ^C,它的缺省措施是立即中止程序。这种异常结束程序是非控制性退出的害处是,写入没关闭的文件中的数据丢失了,或者程序没法恢复原先它发出的中断矢量等。它只涉及 INT 23H 中断。

使用 `ctrlbrk()` 相当于使用一个 DOS 中止处理例程。

3. BIOS 中止处理例程可以只响应 Ctrl-Break 键,而 DOS 中止处理例程可以响应 Ctrl-Break 键和 Ctrl-C 键。

注意:仅管 BIOS 的中止处理例程和 DOS 中止处理例程不能同时装入同一个程序。但是,当 BIOS 的中止处理例程装入后,DOS 中止处理例程将自动失效,此时用户不必担心 MS-DOS 的非控制性程序退出。

```
C>TYPE INT9.C
#include "stdio.h"
#include "dos.h"
int ctr=C(void)    /* 注意:这个函数应是常规函数,而不是 interrupt 函数 */
{
    printf("当你按了Ctrl-Break 键后显示本字符串\n");
    return(0);      /* 这句如果没有写,或者返回的不是 0 值,则执行上句后又    */
                    /* 返回主程序从其中断处继续执行,或者说,对本程序而言,    */
                    /* 按 Ctrl-Break 键将不能终止无穷循环显示。或者说它不起作用 */
}
main()
{
    ctrlbrk(ctr=C);    /* 将 Control break 处理变成处理函数 ctr=C */
    for(;;)printf("按Ctrl-Break 键终止本显示...结束无穷循环显示! \n");
    printf("注意:本句是会被执行的。编译会发出警告:Unreachable code\n");
}

C>TYPE INT10.C
#include "stdio.h"
#include "dos.h"
int ctr=C(void)
{
```



```

printf("\n 当你按了Ctrl-Break 键后显示本字符串\n");
return(1); /* 使用命令行执行程序,按 Ctrl-Break 键,因返回1会继续 */
/* 执行 main() 中内容。但在 Turbo C 集成环境下按 Ctrl-F9 */
/* 执行,执行中按 Ctrl-Break 键,返回集成环境,并显示: */
/* Error:User break,program terminated. Press ESC */
/* 或 User break in xxxxxxxx.C. Press ESC */
/* 按 ESC 键后,Turbo C 自动处于调试状态 */
}
main()
{
for(;;) /* ctrlbrk() 函数在此无穷循环之内 */
{
/* 如果不是无穷循环,则当程序执行中间你 */
int i; /* 始终未按 Ctrl-Break 键,则程序顺利执行 */
for(i=0;i<1000;i++) /* 完,ctr-C() 和 ctrlbrk() 不起任何作用 */
{
if(i%2==0)printf("kk");
else printf("jj");
}
printf("\npress a key when ready...\n");
getch();
ctrlbrk(ctr-C);
for(i=0;i<10000;i++)
{
if(i%2==0)printf("Q");
else printf("T");
}
printf("按一键...无穷循环显示继续!\n");
getch();
}
}

```

—18 int —Cdecl getswitchar(void);

获得当前 MS-DOS 的开关字符值。它是 setswitchar() 相对应函数。

```

void getswitchar(void)
{—AX=0x3700;geninterrupt(0x21);return(—DL);}

```

C>TYPE INT11.C

```

#include "dos.h"
main()
{
char ch;
int f;
ch=getswitchar();
printf("Now,MS-DOS switchar=%c=0x%x\n",ch,ch);
ch='#'; /* 试探性的 */
setswitchar(ch);
printf("New,MS-DOS switchar=%c=0x%x\n",ch,ch);
}

```

```

}
/* 在 MS-DOS 5.0 下输出结果:
Now;MS-DOS switchar=/=0x2f
New;MS-DOS switchar=#=0x23
*/

```

—19 void —Cdecl setswitchar(char ch);

将当前 MS-DOS 开关字符值设置为 ch 的值。开关字符值一般为“/”“#”“—”两种。它实际上是 DOS 的功能调用 37H 的子功能 1, 高于 3.0 的 MS-DOS 版本可能不用, 或者说, 在那些版本中该函数虽然能参加编译运行, 但在程序执行时不起作用。在集成环境下, 按 Ctrl-F1 键时, 尽管当前光标已落在此函数名上, 但没有此函数的相应的帮助信息。

```

void setswitchar(char ch)
{—AX=0x3701;—DL=ch;geninterrupt(0x21);}

```

—20 int —Cdecl getverify(void);

获得磁盘写验证标志 (又称取检验开关)。当其返回值为 0 时, 所有磁盘写都不进行验证; 为 1 时验证, 以保证数据被正确写入。这相当于 DOS 功能调用 54H 的子功能 1, 返回值在 AL 中。只适用于 MS-DOS。

—21 void —Cdecl setverify(int value);

用 value(0 或 1) 设置磁盘写验证标志, 对非关键性记录, 可用它设置检验开关为 off。当 value 大于 1 时均作 1 处理。相当于 DOS 功能调用 2EH。入口参数是, AH = 0x2e; AL = value; (当 value=1 时应使 —DL=0)。只适用于 MS-DOS。

```

C>TYPE INT12.C
#include "dos.h"
#define PP(X) printf(("#X",MS-DOS verify=%d",g),\
                    if(g==0)printf("off\n");\
                    else printf("on\n"))

main()
{
    int g;
    g=getverify();PP(NOW);
    setverify(! g);
    g=getverify();PP(NEW);
}
/* 输出 :NOW;MS-DOS verify=1=on      DOS 缺省状态
        NEW;MS-DOS verify=0=off
*/

```

—22 void —emit—(argument,...);

Turbo C 内部函数, 它将机器码 argument 依次放在源程序中, 源程序编译后它们自动成为机器代码的一部分 (而不再重新编译)。

```

C>TYPE INT13.C
#include "dos.h"
#include "stdio.h"
int —getdisk(void) /* 利用 DOS 功能 19H 取得当前磁盘驱动器号, 注意: 这 */

```

```

/* 里返回 0, 1, ... 分别表示驱动器 A,B,... 跟有些 */
{
/* 函数中定义不一样。函数同 DIR.H 中的 getdisk(), */
——emit——(0xb4,0x19,0xcd,0x21,0x30,0xe4);
}
/* 先用 debug 的 A 命令输入获取驱动器号的汇编程序,然后用 U 命令获得其机器码,并把
它们依次放入 emit 中。
-A 100
2CF0:0100 MOV AH,19
2CF0:0102 INT 21H
2CF0:0104 XOR AH,AH
2CF0:0106
-U 100
2CF0:0100 B419 MOV AH,19
2CF0:0102 CD21 INT 21
2CF0:0104 30E4 XOR AH,AH
-Q
*/

main()
{
int a;
printf("Getdisk!");
a=—getdisk();
printf("%d\n",a);
}
/* 程序输出: Getdisk! 2, 表明当前驱动器是C: */

```

例一个 ROM BIOS 调用通用程序

```

C>TYPE BIOS-ALL.C
/* 本程序名为:BIOS-ALL.C */
#include "dos.h"
#include "stdlib.h"
main(int argc,char *argv[])
{
union REGS r;
int int=no,no-ax=-1,no-bx=-1,no-cx=-1,no-dx=-1;
char c;
if(argc<2)
{
clrscr();
printf(" * * * * * ROM BIOS 调用通用程序命令行参数输入方法 * * * * *\n");
printf(" 在DOS 提示符下依次键入(执行程序名和中断号必输):\n");
printf(" 本执行程序名 中断号 [AX] [BX] [CX] [DX]\n");
printf(" 中断号、AX、BX、CX 和DX 可为10 进制数或16 进制数(如0x10 或10H)\n");
printf(" BIOS 中断入口参数AX、BX、CX 和DX 根据实际可输可不输\n");
printf(" 也可跳跃输入,不起作用参数输-1\n");
printf(" 例如在西文状态下调用BIOS 中断10H 的06H 功能清屏,可键入:\n");
printf(" C>BIOS-ALL 10H 0X0600 0x700 0 184fh\n");
printf(" 注意,不少汉字系统都修改了MS-DOS 原中断10H,在那些系统下本命\n");
}
}

```

```

printf(" 令可能会引起不可预料的结果! 因此在使用本程序时应十分小心。\\n");
exit(1);
}
int—no=hexTOdec(argv[1]);
if(argc>2)no—ax=hexTOdec(argv[2]);
if(argc>3)no—bx=hexTOdec(argv[3]);
if(argc>4)no—cx=hexTOdec(argv[4]);
if(argc>5)no—dx=hexTOdec(argv[5]);
if(no—ax>—1)r.x.ax=no—ax;
if(no—bx>—1)r.x.bx=no—bx;
if(no—cx>—1)r.x.cx=no—cx;
if(no—dx>—1)r.x.dx=no—dx;
int86(int—no,&r,&r);
/* 在此可印出调用后的结果 */
}
#pragma warn —def /* 抑制 Possible use of 'd' before definition 警告 */
hexTOdec(char *h) /* 处理入口参数,统统转换成十进制数 */
{
char **d;
unsigned int x;
if(atoi(h)==—1) return(—1);
x=strlen(h);
if(x<1)exit(1);
if( (h[0]=='0' && (h[1]=='x' || h[1]=='X')) ||
    (x>1 && (h[x—1]=='h' || h[x—1]=='H')) ) x=strtol(h,d,16);
else x=atoi(h);
if(x>0xFFFF)exit(2);
return x;
}

—23 int —Cdecl biosequip(void);

```

它使用了 BIOS 的中断 INT 11H, 主要是检查系统使用了哪些设备。它的返回值在 AX 寄存器中, 为 16 位 (也称【BIOS 设备表字】), 各位含义是:

- | | |
|-------------|--|
| 位0 | 已装软盘, 因此允许从软盘启动 |
| 位1 | 已装数学协处理器 80x87 |
| 位3 和位2 | 母板上 RAM 的大小: 00=16KB, 01=32KB, 10=48KB, 11=64KB, 或不用 |
| 位5 和位4 | 初始化视频模式: |
| | 00 EGA/VGA/PGA |
| | 01 40x25 黑白, 彩色卡 |
| | 10 80x25 黑白, 彩色卡 |
| | 11 80x25 单色卡 |
| 位7 和位6 | 已装软盘驱动器个数: 00=1 个, 01=2 个, 10=3 个, 11=4 个 |
| 位8 | 为 0, 装有 DMA 芯片, 否则未装 |
| 位11, 10 和 9 | 已装串行口数, 如 110=6 个 RS232 端口 |
| 位12 | 已装游戏口 |
| 位13 | 装有串行打印机 |

位15和14 已装并行口(打印机)数,如 01=1 个

本函数只适用于 IBM PC 及其兼容机上,对不同机型可能不同。

注意:在某些 386/486 机上(如 Compaq)已将它扩充到 32 位,另 16 位放到 EAX 寄存器中。

```
C>TYPE INT14.C
#include "bios.h"
main()
{
    long e;
    int k;
    e=biousequip();
    for(k=0;k<32;k++)printf("%d=%#01x\t",k,e>>k & 0x1);
    printf("\n");
}
/* 在386 机上输出:
0=0x1    1=0x1    2=0    3=0    4=0    5=0x1    6=0x1    7=0
8=0     9=0    10=0x1 11=0    12=0    13=0    14=0x1    15=0
16=0    17=0    18=0    19=0    20=0    21=0    22=0    23=0
24=0    25=0    26=0    27=0    28=0    29=0    30=0    31=0 */
```

—24 int —Cdecl biosmemory(void);

本函数通过 BIOS 的中断 INT 12H 的调用返回系统配置的内存大小(单位为 1K),但它不包括扩展内存或扩页内存。该值由主板上的开关设定,存于内存 0040:0013H 开始的字中,它只适用于 IBM PC 及兼容机。

```
C>TYPE INT15.C
#include "bios.h"
main()
{
    printf("由绝对地址0000H 开始的连续内存=%dKB\n",biosmemory());
}
/* 程序输出:由绝对地址0000H 开始的连续内存=640KB */
```

—25 int —Cdecl bioscom(int cmd,char abyte,int port);

用来操作由 port 指定的 I/O 端口上进行各种 RS232 异步通讯。通讯类型取决于 cmd 的值(它实际是中断 INT 14H 的子功能号)。(参见《串行通讯》一章)。

—26 int —Cdecl bioskey(int cmd);

本函数使用 BIOS 的中断 INT 16H 执行各种键盘操作,参见《键盘和鼠标》一章。

—27 int —Cdecl biosprint(int cmd,int abyte,int port);

它调用 BIOS 的中断 INT 17H 对打印机进行操作,参见《打印机》一章。

—28 long —Cdecl biotime(int cmd,long newtime);

实时时钟函数,它调用 BIOS 的中断 INT 1AH,参见《日期和时间函数》一章。

22.6 端口、内存单元存取函数

22.6.1 端口地址

一个程序只能以只读、只写或读/写这三种方式访问硬件。读/写是通过与硬件相连的【端口地址】(或称【口地址】，共有 65535 即 64K 个)进行的。下面列出 286 机 BIOS 一些主要口地址(不同的机器可能不同，仅供参考)：

一 Inter 8237A-5 DMA 控制器

表 22-4

DMA0 DMA1	读寄存器	写寄存器
00H C0H	通道 CH0 当前地址	通道 CH0 基址与当前地址
01H C2H	通道CH0 当前字计数	通道CH0 基址计数与当前字计数
02H C4H	通道CH1 当前地址	通道CH1 基址与当前地址
03H C6H	通道CH1 当前字计数	通道CH1 基址计数与当前字计数
04H C8H	通道CH2 当前地址	通道CH2 基址与当前地址
05H CAH	通道CH2 当前字计数	通道CH2 基址计数与当前字计数
06H CCH	通道CH3 当前地址	通道CH3 基址与当前地址
07H CEH	通道CH3 当前字计数	通道CH3 基址计数与当前字计数
08H D0H	状态寄存器	命令寄存器
09H D2H		请求寄存器
0AH D4H		写屏蔽寄存器(单个位屏蔽)
0BH D6H		工作方式寄存器
0CH D8H		清除先/后触发器
0DH DAH	暂存寄存器	清除指令
0EH DCH		清屏蔽寄存器
0FH DEH		写屏蔽寄存器(全部位)

三个通道用于 I/O 设备与存储器之间的高速数据传送，第四个通道用于对动态存储器进行刷新。

二 8259A 中断控制器

表 22-5

INTA0	INTB0	说明
20H		INTA00=8259 主片(硬中断级别 IRQ0~7)
21H		INTA01=中断起始矢量 08H
	A0H	INTB00=8259 从片(硬中断级别 IRQ8~15)
	A1H	INTB01= 中断起始矢量 70H

三 Intel 8253(或 AT 机上的 8254) 定时器

它的作用是计算系统时钟的脉冲数。若干个时钟周期转换成一个脉冲。脉冲序列可用于计时，也可送入扬声器发出特定的声音。

8253 芯片独立于 CPU 运行，CPU 对它编程后便可去做其它工作，因此 8253 可以像实

时时钟那样运转,而计算机工作对它无影响。然而 8253 最长可编程间隔只有二十分之一秒,而许多工作需要的时间至少为数分钟以上的定时。为此,计算机通过 BIOS 数据区的一个变量来记录 8253 的通道 0 发出的脉冲数(它不是时钟脉冲,时钟每秒可发 1193180 个时钟脉冲,而启动时 BIOS 给通道 0 每秒输出 18.2 个脉冲)。其过程示意图如图 22-2 所示。

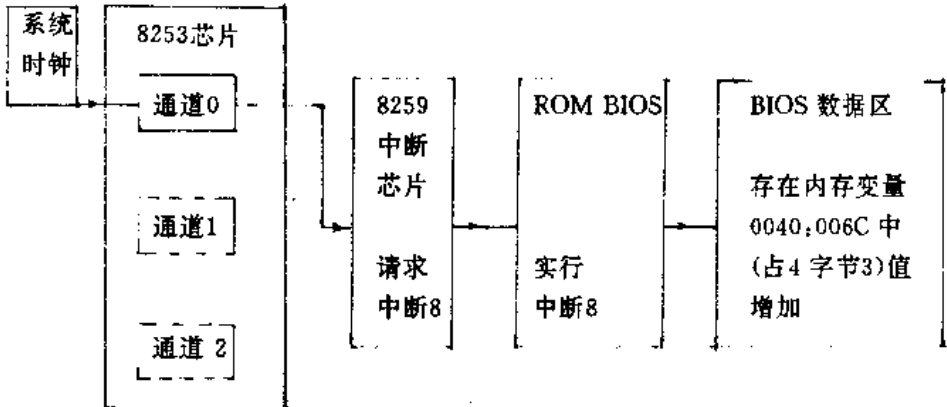


图 22-2 日历计数过程示意图

这一脉冲计数称为【日历计数】。通道 0 每秒发出 18.2 个脉冲,申请 18.2 次定时器中断。一天内的任何时刻(秒)可用计数值除以 18.2 得到。

三个定时/计数器:系统时钟定时器(40H)、刷新请求发生器(41H)和扬声器用音调发生器(42H)。方式控制寄存器(43H)为三个定时/计数器共用。

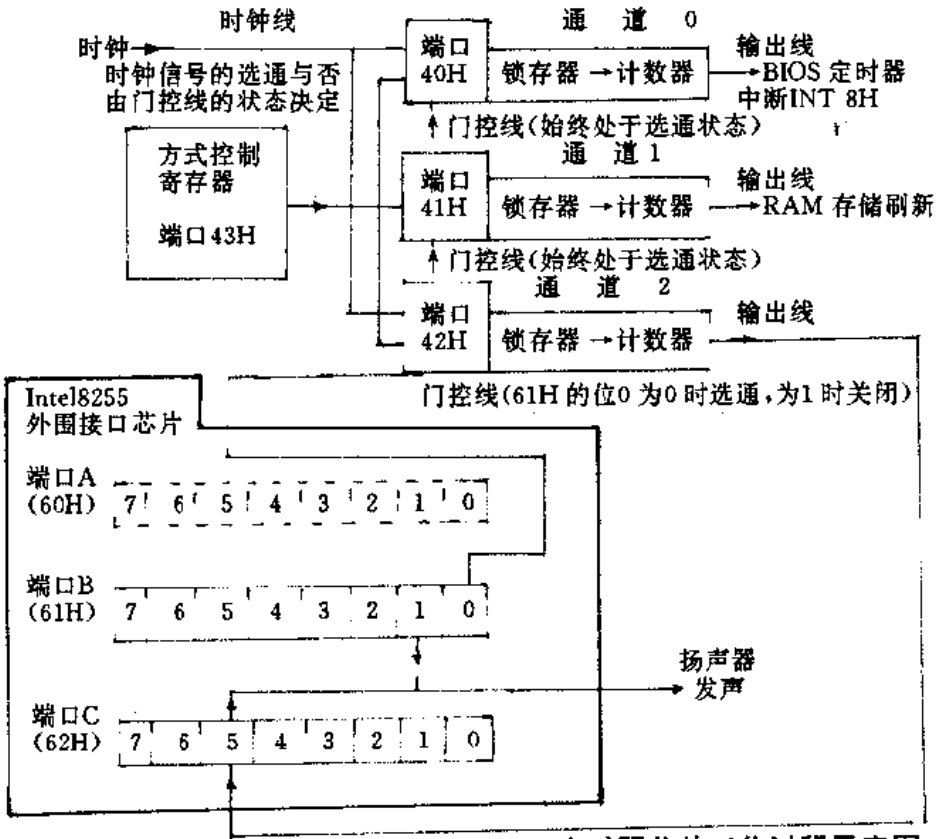


图 22-3 8253 定时器芯片工作过程示意图

每一个通道和一个 8 位方式控制寄存器相连,用于通道的数据存取。它决定送入锁存器

的数(称【锁存值】)方式。各位作用是:

- 位7和位6: 00——选通道 0
01k——选通道 1
10——选通道 2
11——非法,无效
- 位5和位4: 00——计数器锁存操作,并不将锁存值复制给计数器
01——只读写低字节,即只将锁存值的低字节送计数器
10——只读写高字节,即只将锁存值的高字节送计数器
11k 将锁存值的两个字节送计数器,先读写低字节,后读写高字节
- 位3、2、1: 000——方式 0
001——方式 1
010——方式 2
011——方式 3 对通道0用于定时或通道2用于定时和发声时常用此方式,此时锁存值进入锁存器后立即被复制到计数器。
100——方式 4
101——方式 5
- 位0: 0——为二进制码
1——为BCD码(十进制数的二进制代码)

通过口 43H 将一个两字节数(即锁存值,低字节在前)先传到锁存器保留,再将它复制到计数器。以后每当时钟脉冲进入通道,计数器中的数就减1。当计数器内的数减至0时,通道发出一个输出脉冲,与此同时,锁存值又被复制到计数器,计数器又进行减数,发出脉冲,如此不断循环。因此,开始复制给计数的数越小,发出脉冲的速度就越快。三个通道在任何时候都是能接收数据的,CPU 既不能启动也不能关断它们,但 CPU 可以在任何时刻读出任意一个计数器内的当前值。

对 8253 编程有三个步骤:

1. 确定送端口 43H 的数并送入 43H 端口;
2. 如果选通道 2,则应给端口 61H 位 1 和位 0 置数,启动时钟信号。当 61H 的位 1 置 1 时,通道 2 启动扬声器,为 0 用于定时操作;
3. 确定锁存值。例如,要将日历计数速率由每秒 18.2 次变成每秒 1000 次,则锁存值为 $1193180/1000=1193$,并将这个值放入 AX 寄存器,然后分别依次将 AL 和 AH 中的值送入指定通道的端口(注意:使用汇编时送端口是一字节一字节送,且 AH 中的值也要通过 AL 送入)。通常情况下,在程序结束之前应恢复 8253 原来设置的数值,即每秒 18.2 次。

四 8255 芯片

不同机上可能用途有些不同。键盘控制器(8042 和 8048 等)将信号送往 8255,8255 有 5 个口:

- 60H 称端口 A,只能对它进行读操作。当端口 B 的位 7 为 0 时它是键盘接收的 8 位扫描码,故常从该口取键盘扫描码;当端口 B 的位 7 为 1 时,它的位 7 和位 6 表示磁盘驱动器数目,位 5 和位 4 表示显示类型(11 = 单色,10 = 80x25 彩色,01 = 40x25 彩色)。
- 61H 称端口 B,可读可写口,即可从该口读码,也可将改变的码写入该口。

它的位 0 是 8253 定时器芯片通道 2 的控制门;位 1 输出至扬声器;位 2、3 选择端口 C 的内容;位 5 表示扩展槽出错信号;位 6 表示能使用键盘时钟信号;位 7 选择端口 A 的内容。

62H 称端口 C, 只能读。
 63H
 64H 键盘控制端口。其值和意义如下:
 1H 输出缓冲满
 2H 输入缓冲满
 4H 系统标志/自粘
 8H 命令/数据
 10H 键盘禁止
 20H 发送超时
 40H 接收超时
 80H 奇偶在偶位
 60H 写键盘字节 D7 ~ D0
 AAH 自粘无误送 55 码到输出缓冲
 ABH 接口检查正常送 00 码
 E0H 取键盘时钟与数据
 FEH 停工命令

五 RT/CMOS(实时时钟 / 互补金属氧化物半导体) RAM 口地址

(慎用。当需要对系统重新设置时, 一般可在开机时用键盘操作启动 BIOS SETUP 程序对它设置。例如, 对某些 80386 机在内存检测阶段, 可以在屏幕上看到

Hit , if you want to run set up

信息。按 Del 键便可看到进入配置程序的菜单:

STANDARD CMOS SETUP(标准 CMOS 设置)
 ADVANCED CMOS SETUP(高级 CMOS 设置)
 ADVANCED CHIPSET SETUP(高速芯片组设置)

一些机上标准 CMOS 设置是通过 70H 和 71H 端口进行的, 这里 70H 口为地址口, 71H 口为数据口)

对 Motorola MC 146818 实时时钟芯片, 有 64 个 8 位寄存器, 它们可通过 70 口选择。

1. 向 RT/CMOS RAM 写入数据步骤:

- (1) 将要写入 RT/CMOS RAM 的寄存器号送到口地址 70H 中;
- (2) 将要写入 RT/CMOS RAM 的数据送到口地址 71H 中。

2. 从 RT/CMOS RAM 读出数据步骤:

- (1) 将要读出 RT/CMOS RAM 的寄存器号送到口地址 70H 中;
- (2) 从口地址 71H 中获得数据送入 AL 寄存器中。

70H 端口:

选择要读 /

写寄存器号 用 途 (对 80286 机, 仅供参考)

00H	秒
01H	秒报警
02H	分
03H	分报警
04H	时
05H	时报警

06H	星期
07H	日
08H	月
09H	年
0AH	状态寄存器A
	位7 1 表示计时器正在计时 0 表示当前日期和时间可读
	位6~位4 识别正在使用那一个时间基准频率(三个分频器中选一)
	位3~位0 选择分频器的输出频率
0BH	状态寄存器B
	位7 表示按计时器每秒加1的速度计时;1表示停止计时和程序能对14个时间字节初始化
	位6 1表示允许周期中断。它允许中断发生在寄存器A中的速率和分频器位所指定的速率上,0表示禁止中断(缺省值)
	位5 1表示允许报警中断
	位4 1表示允许计时器计时结束中断
	位3 1表示允许由寄存器A中速率选择位所设置的方波频率
	位2 1表示日期和时间使用二进制;0表示用二进制(BCD码)
	位1 1表示24小时方式,0表示12小时方式
	位0 1表示夏令时间,0表示禁止使用夏令时间,采用标准时间
0CH	状态寄存器C
0DH	状态寄存器D
	位7 有效RAM位。1表示电源接通实时时钟,0表示实时时钟已掉电
0EH	诊断状态
	位7 1表示实时时钟芯片掉电
	位6 1表示配置记录的状态和检查不正确
	位5 设备字节检测,1表示配置信息无效
	位4 1表示加电检查所决定的实际存储容量与配置记录中的存储容量不相同
	位3 1表示硬盘适配器或驱动器C初始化失败
	位2 1表示时间状态指示器指示的时间无效(POST有效性检查)
	位1和位0 保留
0FH	停机状态(由POST上电自检程序设置)
	此字节中各位由加电诊断定义:
	=00H 软件或异常中断停工
	=01H RAM容量停工
	=02H RAM测试停工
	=03H RAM出错停工
	=04H 启动装载要求停工
	=05H 中断初始化(要求双地址)
	=06H 保护方式通过
	=07H 或 =08H 保护方式失败
	=09H 模块移动要求
	=0AH 中断初始化输出
10H	软磁盘驱动器类型(如值为0表示没有安装,为1表示360K双面软盘驱动器,为2表示1.2M高密驱动器,3表示720K驱动器,4表示1.44M驱动器等)
	位7~位4 第一个软驱类型(驱动器A)

	位3~位0 第二个软驱类型 (驱动器 B)
12H	硬盘驱动器类型 (如果为 0 表示没有安装, 否则数字表示硬盘类型)
	位7~位4 第一个硬盘类型 (驱动器 C)
	位3~位0 第二个硬盘类型 (驱动器 D)
13H	(由于目前硬盘类型增加, 该保留字节也可能用来表示硬盘类型, 但未见 报告。对有些机上未用)
14H	设备标志字节 (参见 BIOS 工作区 0x10 字节)
15H	系统板存储器 (基本容量低字节)
16H	系统板存储器 (基本容量高字节, 单位:KB)
17H	全部扩展存储器容量 (低字节)
18H	全部扩展存储器容量 (高字节, 单位:KB)
2EH	检查和和高字节
2FH	检查和的低字节 (对地址 10H ~ 20H 检查)
30H	超出 1M 的扩展存储器 (低字节)
31H	超出 1M 的扩展存储器 (高字节, 单位:KB)
32H	日期世纪字节 (BCD 码)
33H	信息标志 (在加电时设置)

71H 端口: 要读/写寄存器的内容。

六 页面寄存器 (74LS612) I/O 地址

80H~8FH

在进行 DMA 操作时, 相应页面寄存器提供高 8 位地址。

七 协处理器 80287 I/O 口地址

F0H 清除口

F1H 复位口

F8H~FFH 数学运算

注: 80286 为 80287 分配了三个 I/O 口地址: F8H、FAH 和 FCH。

八 游戏杆 I/O

201H

九 打印通信口

278H 并行口打印机 2 (LPT 2)

378H 并行口打印机 1 (LPT 1)

2F8H 串行口 2 (COM 2)

3F8H 串行口 1 (COM 1)

十 显示卡 (不同显示卡可能不同, 参见《视频函数》一章)

3B4H 单色显示器 6845 内部索引寄存器

3B8H 单色显示器方式控制寄存器

3BAH 单色显示器状态寄存器
 3D4H 彩色显示器 6845 内部索引寄存器
 3D8H 彩色显示器方式控制寄存器
 3DAH 彩色显示器状态寄存器

十一 软盘接口板

A 盘	B 盘	读	写
3F2H	372H		输出寄存器
3F4H	34H	主状态寄存器	主状态寄存器
3F5H	375H	数据寄存器	软盘数据Reg
3F6H	376H		
3F7H	377H	输入寄存器	软盘控制 Reg

十二 硬盘卡 I/O 地址

C 盘	D 盘	读	写
1F0H	170H	数据寄存器	数据寄存器
1F1H	171H	差错寄存器	设磁道予补偿
1F2H	172H	扇区计数	设扇区计数
1F3H	173H	扇区号	设扇区号
1F4H	174H	低道磁柱	设低道磁柱
1F5H	175H	高道磁柱	设高道磁柱
1F6H	176H	盘号/磁头	设盘号/磁头
1F7H	177H	状态寄存器	设命令寄存器

1F1H 差错寄存器读出码: 1H 数据地址标志错
 2H 0 道回不去
 4H 命令作废
 10H ID 标识未找到
 1F7H 状态寄存器读出码: 04H ECC 校验通过
 10H 寻道成功
 20H 写出故障
 1F7H 命令寄存器命令设置: 1H 命令修改 (再试)
 2H 命令修改 (ECC 方式)
 8H 命令修改 (缓冲器方式)
 10H 磁头回零
 20H 磁头读
 30H 磁头写
 40H 磁头校验
 50H 磁道格式化
 60H 磁道初始化
 70H 寻道
 90H 诊断
 91H 设磁盘参数

22.6.2 读写端口或内存单元内容

一 对端口操作

· 读端口中一个字节

—1 inportb()

- 2——inportb——()
 - 3 inp()
 - 4 inport()
 - 5 outportb()
 - 6 ——outportb——
 - 7 outp()
 - 8 outport()
- 读端口中一个字
- 输出一个字节值到端口中
- 输出一个字到端口中

二 对内存单元直接存取

- 得到由(段:偏移量)指定的内存单元开始的一个字节的内容 —9 peekb()
- 10 peekb(a,b)
- 得到由(段:偏移量)指定的内存单元开始的一个字的内容 —11 peek()
- 12 peek(a,b)
- 把一个字节存放到由(段:偏移量)指定的内存单元中 —13 pokeb()
- 14 pokeb(a,b,c)
- 把一个字存放到由(段:偏移量)指定的内存单元中 —15 poke()
- 16 poke(a,b,c)

—1 unsigned char —Cdecl inportb(int portid);

从硬件端口 portid 中读出一个字节。只适用于 8086 系列。

—2 #define inportb(portid) ——inportb——(portid)

从硬件端口 portid 中读出一个字节。用 Turbo C 的 CPP.COM 工具检查,如程序包含 DOS.H,则用此宏代替函数。——inportb——() 是 Turbo C 内部函数。内部函数由于合理地免去了一些语法检查因而能节省代码,从而使程序执行更快。只适用于 8086 系列。

—3 #define inp(portid) inportb(portid)

inp() 是一个从硬件端口 portid 中读出一个字节的宏,它是另一些编译器使用的函数,这样的宏定义使 Turbo C 能直接使用它。只适用于 8086 系列。

—4 int —Cdecl inport(int portid);

从硬件端口 portid 中读出一个字。只适用于 8086 系列。

—5 void —Cdecl outportb(int portid,unsigned char value);

输出一个字节值 value 到硬件端口 portid 中。使用汇编时一般将 portid 放到 DX 寄存器,value 放到 AL 寄存器,然后使用 OUT 指令(out dx,al)。只适用于 8086 系列。

—6 #define outportb(portid,v) ——outportb——(portid,v)

输出一个字节值 value 到硬件端口 portid 中。用 Turbo C 的 CPP.COM 工具检查,如程序包含 DOS.H,则用此宏代替函数。——outportb——() 是 Turbo C 内部函数。

—7 #define outp(portid,v) outportb(portid,v)

outp() 是一个输出一个字节值 value 到硬件端口 portid 中去的宏,它是另一些编译器使用的函数,这样的宏定义使 Turbo C 能直接使用它。只适用于 8086 系列。

—8 void —Cdecl outport(int portid,int value);

输出一个字(整数值)value 到硬件端口 portid 中。使用汇编时一般将 portid 放到 DX 寄存器,value 放到 AX 寄存器,然后使用 OUT 指令(out dx,ax)。只适用于 8086 系列。

```
C>TYPE INT16.C
```

```
#include "dos.h"
```

```
main()
```

```

{
int far * ahead;
int a1;
ahead=MK—FP(0xc000,0x0025);/* 使用调试表达式 ahead[0],5mc: "PATIB" */
/* 这表明现 SVGA 卡不是 Ahead V5000 系列芯片 */
outport(0x3ce,0x200f);
a1=inp(0x3ce); /* a1:15 表明它既不是 Ahead V500 Version 芯片(a1=20H) */
/* 也不是 Ahead V5000 Version 芯片 (a1=21H) */
ahead=MK—FP(0xc000,0x0040);/* ahead[0],5mc: " C2" 表明它不是 ATI 的 */
/* SVGA,若是 ATI 的 SVGA 卡其值应为 3331H */
outp(0x3C4,0x11);
a1=inp(0x3C4); /* a1:2 表明它是 8800 芯片,仅支持 512K 显示存储区 */
/* 8900 芯片支持 1MB 显示存储区,a1>=3 */
}

```

C>TYPE INT17.C

```

#include "graphics.h"
main()
{
/* int far i; error:Conflicting type modifiers ! */
int graphdriver,graphmode;
char far * pathtodriver="c:\\tc";
detectgraph(&graphdriver,&graphmode);
initgraph(&graphdriver,&graphmode,pathtodriver);
printf("%d %d\n",graphdriver,graphmode);/* 9 (VGA) 2 (VGAHI) */
setrgbpalette(1,0x5,0x4,0x3);
}

```

—9 char —Cdecl peekb(unsigned segment,unsigned offset);

得到由 segment;offset 指定的内存单元开始的一个字节的内容,它是函数。

只适用于 8086 系列。

```

#include "dos.h"
#undef peekb
char peekb(unsigned segment,unsigned offset)
{
—ES=segment;
return (* (unsigned char —es * )offset);
}

```

—10 define peekb(a,b) (* ((char far *)MK—FP((a),(b))))

得到由 segment;offset 指定的内存单元开始的一个字节的内容,但它是一个宏。

—11 int —Cdecl peek(unsigned segment,unsigned offset);

用它可得到由 segment;offset 指定的内存单元开始的一个字的内容。注意:如果在 使用它时没有包含 DOS.H(或者虽然包含了,但用了 #undef 指令)才是函数,否则它被当作宏对待而扩展为插入代码。只适用于 8086 系列。下面是包含了 DOS.H 而作函数时的定义:

```

#include "dos.h"
#undef peek
int peek(unsigned segment,unsigned offset)
{
—ES=segment;

```

```

    return ( * (int —es * )offset);
}

```

—12 #define peek(a,b) (* ((int far *)MK—FP((a),(b))))

得到由 segment:offset 指定的内存单元开始的一个字的内容,但它是一个宏。

—13 void —Cdecl pokeb(unsigned segment,unsigned offset,char value);

—14 #define pokeb(a,b,c) (* ((char far *)MK—FP((a),(b)))=(char)(c))

它把一个字节 value(而不是一个字) 存放到 segment :offset 开始的单元中。

—15 void —Cdecl poke(unsigned segment ,unsigned offset ,int value);

—16 #define poke(a,b,c) (* ((int far *)MK—FP((a),(b),(c)))=(int)(c))

把一个整数值 value 存放到内存单元 segment:offset(段;偏移量) 中。

在调用 poke() 时,如果程序包含了标头文件 dos.h ,则 poke() 将作为可扩展的插入代码的宏看待;否则 (或者虽然包含了,而在随后的调用之前已用了 #undef 编译指令 取消了宏定义) 它将被当作函数,而不是作宏处理。(实例参见《键盘与鼠标》一章。)

它只适用于 8086 系列。

22.7 内存控制块 MCB(Memory Control Block)

DOS 每装入一个程序时就在内存中给这个程序分配相应的【内存块】(或称【存储块】), 内存块中除包含装入的程序信息外,还有 DOS 要用到的一些控制信息 (例如程序的环境)。各个内存块的长度是不一样的,但是它们都是以“节”(16 个字节)为边界的,或者说,它是以 16 字节为一个单位往内存高端方向占用内存的,大于 0 而不足 16 字节时也要占用一个 16 字节。另一方面,在每一个内存块前面都有一个 MCB,每个 MCB 长度为 16 字节。它的前 5 个字节的作用是:

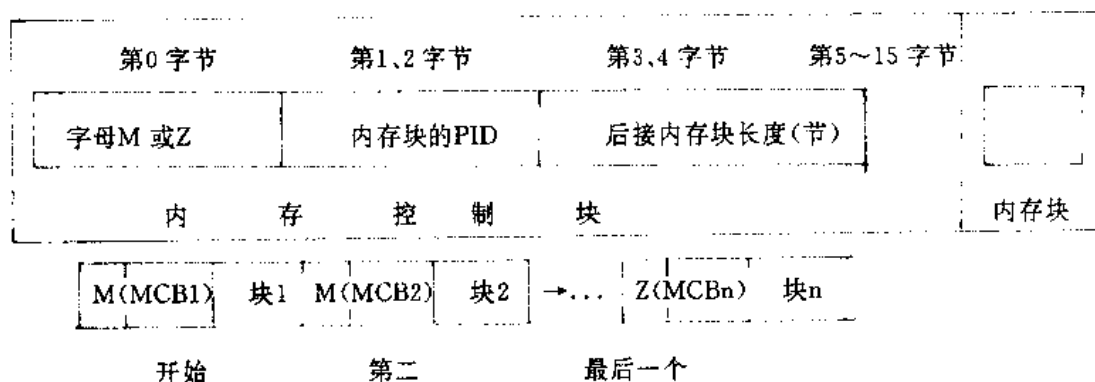


图 22-3

说明:1. MCB 的第 0 个字节除最后一个为字母 Z (或 0x5a) 外,其余均为字母 M (或 0x4d)。

2. 内存块的 PID 有三种情况:或为其后内存块的 PSP 地址,或为 0,这时表示内存块是自由的;若为 0008H 则属于 DOS。

3. 当前内存控制块与下一内存控制块的关系是:

下一内存控制块的地址 = 当前内存块的地址 + 当前内存块长度 + 1

C>TYPE INT18.C

```
#include "dos.h" /* 本程序只能在 DOS 3.0 以上版本才能用 */
```

```

/* 程序描述追踪 MCB 的情况并印出 */
#define PW(X) printf("0x%x ",peek(mcbseg,X));
#define PB(X) printf("0x%x ",peekb(mcbseg,X));
main()
{
union REGS rg;
unsigned mcbseg,i=0;
long int y,x;
rg.h.ah=0x52; /* DOS 功能调用 52H */
intdos(&rg,&rg); /* ES:DX→表指针 (list of lists) */
mcbseg=-ES; /* 即 DOS 内部信息指针 */
printf("0==0x%x;0x%x\n",mcbseg,rg.x.bx-2); /* BX-2 返回第一个内存 */
mcbseg=peek(mcbseg,rg.x.bx-2); /* 控制块 MCB 的段地址 */
printf("0==0x%x\n",mcbseg);
PB(0),PW(1),PW(3),PW(5),PW(7),PW(9),PW(11),PW(13),PB(15);
printf("\n");
i++;
while(peekb(mcbseg,0)==0x4d)
{
x=peek(mcbseg,3)+1;
mcbseg+=x;
printf("i=%d==0x%x\n",i++,mcbseg);
PB(0),PW(1),PW(3),PW(5),PW(7),PW(9),PW(11),PW(13),PB(15);
printf("\n");
}
if(peekb(mcbseg,0)==0x5a);
{
y=peek(mcbseg,3)+x;
printf("\navailable memory:%ld",y*16);
}
else printf("error MCB! \n");
}
/* 输出结果举例:
0==0x125;0x24
0==0xb66
0x4d 0x8 0x7db 0x5e19 0x531f 0x3e44 0x5e2 0x8026 0x4d
i=1==0x1342
0x4d 0x1343 0x116 0x0 0x4300 0x4d4f 0x414d 0x444e 0x0
i=2==0x1459
0x4d 0x0 0x4 0xe902 0xfe14 0x3fba 0xe806 0x265 0xffb8
i=3==0x145e
0x4d 0x1343 0x10 0x885a 0xa526 0x402 0xa241 0x205 0xffff6
i=4==0x146f
0x4d 0x1478 0x7 0xbe05 0x215 0x8e8 0xba01 0x547 0xffe8
i=5==0x1477
0x4d 0x1478 0x1174 0x63a 0x4c44 0x3936 0x5900 0xebff 0x65

```



```
i=6==0x25ec
```

```
0x5a 0x0 0x7a13 0x0 0x0 0x0 0x0 0x0
```

available memory:571520 (当当前程序执行结束后内存可用字节数)

倒数第二个 MCB 为本程序占用的内存块 (大小为 0x1174)。利用本程序你可以对像 内存驻留程序等观察其占用内存情况。*/

22.8 interrupt 中断函数修饰符和常驻内存程序

Turbo C 有一个 interrupt 中断函数修饰符,它不属于 ANSI 标准,它只用于函数说明。被说明的函数会自动保留所有的寄存器的内容,并在函数返回时又恢复这些寄存器的值。被修饰为 interrupt 的函数设置 ds 寄存器为模块的当前数据段,它是通过 IRET 指令而不是通过 RET 指令返回的。

interrupt 函数一般说明格式为:

```
void interrupt my(bp,di,si,ds,cx,bx,ax,ip,cs,flags,...);
```

所有的寄存器都作为参数传递,无须使用寄存器变量就可以使用 and 修改寄存器值。例如,要使用寄存器 AX 你可以对 my() 函数直接用 ax。

8086 的处理器支持 256 个 (BIOS) 中断矢量。中断矢量的含意是,一旦发生某个中断,处理器将从一个中断矢量表中找出该中断矢量的地址 (是内存的某一地址),然后根据这个地址去启动与它对应的中断服务例程 (ISR)。

内存驻留程序 (TSR, 即 Terminate and Stay Resident application) 是指程序运行终止后仍保持驻留在内存中一个受保护的区域 (DOS 保证它不被后来的应用程序覆盖),然后将控制权交回给 DOS,使用户能运行其它的应用程序。它在后台一般处于休眠状态,即不起作用;一当用户按下了某个键 (或组合键,这样的键称为【热键】,该键一般由应用程序预先设置的) 后,TSR 便被激活,它先暂时挂起当前正在运行的应用程序而去运行内存驻留程序,当它从初始激活的函数返回后又继续运行原先挂起的程序。注意,当从 TSR 的入口函数返回到被挂起应用程序前,TSR 程序应恢复其原先状态,为此应考虑事先保留挂起前状态 (如屏幕数据的存储和恢复等)。

TSR 之所以能被激活常利用三个中断:系统定时器中断 INT 8H、键盘中断 INT 9 和测试 DOS 是否忙的中断 INT 28H。简要叙述如下:

一是利用系统定时器中断 INT 8H (也可使用中断矢量 INT 1CH)。因为 PC 每秒由系统定时器中断 18.2 次 (精确的数字是每秒 1193180/65536 次),如果应用程序中先将老定时器中断矢量服务程序入口地址保留,然后给予定时器中断矢量服务程序一个新的入口地址,那末在调用执行定时器中断服务程序时,系统便自动去执行这个新的所谓的定时器中断服务程序。如果我们在新的程序中包含了我们希望要做的事 (例如显示一次时间等等),那么每当定时器中断时这个愿望就可能被实现。当然,另一方面,用户一般很少知道老定时器中断要做哪些事情,因此一般在新定时器中断服务例程中一开始应先调用一次老定时器中断,以避免先去一些重要的服务 (如给可编程中断控制器发送中断结束信号,它将给中断以一个相等或低一些的优先权,然后开中断)。事实上,如果在新程序中不是一开始就调用老程序,那么,BIOS 保持时间的例程将永远接受不到控制权,从而使 BIOS 的时间数据将成为无效;

另一个是利用键盘中断 INT 9H,此中断是在每次敲击或松开一个键时产生的。为了使热

键起作用,就必须修改这个硬件中断。一旦热键被击下,就可以使一个全局变量(例如 hotkey)置标志为 1(未按下时为 0)。这个过程是在新中断键盘服务程序中处理的,如果包括中断服务程序的应用程序常驻内存,或即包括代码、数据和栈等等常驻内存,所以这个变量将起作用,并将其新值传给其它函数使用。和利用定时器中断一样,在此之前应先保留老键盘中断矢量服务程序的入口地址,并在新键盘中断服务程序一开始执行时,要先调用一次老键盘中断服务例程,否则,当程序终止后,敲击或松开一个键时将导致传送控制到含有随机数据的内存中某点,达不到预期目的。

注意:常驻内存程序中有些变量或函数等将不起作用,只要它们未被重新赋值或调用。这是因为,应用程序常驻内存后,它能被激活运行的只是中断服务例程,而不是整个应用程序本身!换言之,执行不再从应用程序开头顺次进行,而是根据被激活的中断服务例程产生的结果去调用函数,常将第一个被调用的(非中断)函数称为 TSR 入口函数。

第三个中断是测试 DOS 是否忙的中断 INT 28H。一般来说, DOS 忙(例如用一个标志 dos-active 来标志,当其值为 1 是表示忙)时不能去激活 TSR 中断入口函数,因为那时是不安全的。但是有一种例外,例如当 DOS 的命令处理器正在显示提示信息并等待用户输入时,这时候很有可能 * dos-active=1,但事实上此时 DOS 还是比较安全的,即弹出 TSR 也不会有问题。这是因为每当 DOS 在等待输入时,它实际上是在反复循环调用 INT 28H,此时 DOS 对某些关键程序片断是绝对不允许打断的。另一方面,当 DOS 处于关键片断时,它也决不会去用中断 INT 28H。所以仅管忙标志为 1,仍不会出问题。虽然 TSR 顺利地装入了内存,但按下热键时有时并不一定能马上收到即时控制权,即立即安全地中止当前进程而激活 TSR,例如 DOS 或 BIOS 正忙于磁盘操作时。

常利用 Turbo C 的扩展功能(中断函数和伪变量)来编写 TSR。

一 全局变量

- | | |
|-------------------------------|--------------|
| · 被分配的堆的大小 | —1 —heaplen |
| · 实际堆字节数或数据段的最小字节数 | —2 —stklen |
| · MS-DOS 版本号 | —3 —version; |
| · MS-DOS 版本的主号(—version 的高字节) | —4 —osmajor; |
| · MS-DOS 版本的次号(—version 的低字节) | —5 —osminor; |
| · 程序段前缀 PSP 的段地址 | —6 —psp |

二 中断函数

- | | |
|-------------------------------------|--------------|
| · 读指定中断号的矢量的当前值(中断服务例程地址) | —7 getvect() |
| · 改写指定中断号的矢量的当前值为新值(新中断服务例程地址) | —8 setvect() |
| · 将当前程序中止运行并驻留在内存 | —9 keep() |
| —1 extern unsigned —Cdecl —heaplen; | |

全局变量。在程序开始执行时,将根据其值确定被分配的堆的大小。它的缺省值为 0,表示为扩展 DS 段用尽可能大的堆。可以对

```
C>TYPE TSR1.C
#include "dos.h"
main()
{printf("%d\n",—heaplen);
}
```

用调试表达式 `--heaplen` 验证,但不能对

```
C>TYPE TSR2.C
#include "dos.h"
main()
{printf("d\n");
}
```

用同一表达式验证,那样会出现

`--heaolen; No type information`

的提示。事实上,这是因为对说明为 `extern` 的变量只有对它赋值或引用时它才被跟踪。对其它说明为 `extern` 的变量也有类似的调试结论。

`--2 extern unsigned --Cdecl --stklen;`

在大数据模式 (Compact、Large 和 Huge) 它是实际堆的字节数;在小数据模式 (Tiny、Small 和 Medium) 可用它来计算数据段的最小字节数,数据段中包含了初始化的全局变量、未初始化的数据和堆栈:

最小数据段大小 = `--DATA` 段大小 + `--BSS` 段大小 + `--stklen` + 最小栈尺寸 (128 个字) 如果当前可用内存小于这个数,启动程序将退出。当然,最大的数据段为 64K。

`--stklen` 的缺省值为 4K (=4096 字节)。也可以用该变量设置新的堆栈大小。

`--3 extern unsigned --Cdecl --version;`

`--4 extern unsigned char --Cdecl --osmajor; --5 extern unsigned char --Cdecl --osminor;`

这三个变量包含 MS-DOS 版本号 (`--version`)、版本的主号 (`--osmajor` 或 `--version` 的低字节) 和版本的次号 (`--osminor` 或 `--version` 的高字节)。应当注意: Turbo C 的有些库函数与版本号相关,包括在不同版本下结果不同,或者只能在某一版本下运行 (如 `--open()`、`creatnew()` 及 `icotl()` 等函数只能在 MS-DOS 3.X 版本下运行)。

```
C>TYPE TSR3.C
#include "dos.h"
main()
{
printf("--version=0x%x\n",--version);
printf("--osmajor=%d\n",--osmajor);
printf("--osminor=%d\n",--osminor);
}
/* 对MS-DOS 3.30 输出:--version=0x1e03
--osmajor=3
--osminor=30 */
--6 extern unsigned --Cdecl --psp;
```

全局变量,程序执行后它包含程序段前缀 PSP 的段地址 (参见《程序头前缀 PSP》一章)。

```
C>TYPE TSR4.C
#include "dos.h"
main()
{
printf("PSP=0x%x\n",--psp);
}
```

```

}
/* 例如输出:PSP=0x77d5 */

```

```

—7 void interrupt (* —Cdecl getvect (int interruptno))();

```

原型中 interruptno 是中断号。它前面的修饰符 interrupt 说明 getvect() 是一个接受整型参数,返回 interrupt 型中断函数指针,因此,如果要使

```

int0x81=getvect(0x81);

```

成立,则 int0x81 必须也说明为 interrupt 型函数指针,即

```

void interrupt (* int0x81)();

```

getvect() 读中断号为 interrupt 的矢量的当前值 (即中断例程或中断函数的远地址,是 4 字节值)。因此又说, getvect() 取得中断矢量入口。

它实际是调用 DOS 功能 INT 35H。入口参数是, AH = 0x35, AL = interruptno (中断号)。结果是该中断执行后把指定中断矢量例程的 CS:IP 值被放入 ES:BX 中返回。因此,它只适用于 MS-DOS。

```

C>TYPE TSR5.C
#include "dos.h"
main()
{
    long bxx,ess;
    void interrupt (* int0x10)();
    int0x10=getvect(0x10);
    ess=—ES;
    bxx=—BX;
    printf("INT 10H 地址=%Fp 或:%lx:%lx\n",int0x10,ess,bxx);
}
/* 例如,在 LX-386/33S 汉字系统下,对集成环境用 Ctrl-F9 执行得

```

```

    INT 10H 地址=2753:1BA0 或;2753:1ba0

```

而在同一汉字系统下用 DOS 命令行执行得

```

    INT 10H 地址=147E:0106 或;147e: 106

```

当然,在西文系统下它们的值也与此不同。由此可见一些汉字系统常可能修改视频中断 10H 例程的入口。

```

*/

```

```

C>TYPE TSR6.C
#include "dos.h"
void interrupt (far *getvect(int intr)()
{
    —AH=0x35;
    —AL=intr;
    geninterrupt(0x21);
    /* #pragma warn —sus */
    return (char —es *)—BX;
    /* #pragma warn .sus */
    /* near 数据指针作修饰符,—BX 为相对于附加段 —es 的 16 位偏移量 */
}

```

```

/* 编译时出现警告,Suspicious pointer conversion,可不管它 */
/* 故在嵌入汇编时可加上,#pragma warn -sus,表示警告变成关 */
/* #pragma warn .sus 则表示警告将保留程序编译前的值 */
}

main()
{
int —INT1,m,n;
for(—INT1=0;—INT1<2;—INT1++)
{
m=—INT1*4;
n=m+2;
printf("INT%d= 0x%x;0x%x\n",—INT1,peek(0,m),peek(0,n));
setvect(—INT1);
printf("%x\n",getvect(—INT1));
}
} /* 经整理后的输出:

```

	初始	调试时	EXIT 退出 Turbo C
INT0=	0x158;0x1488	0x158;0x763a	0x158;0x667a
	158	158	158
INT1=	0x6f4;0x70	0x18f4;0x1488	0x6f4;0x70
	6f4	18f4	6f4

```

*/

```

—8 void —Cdecl setvect(int interruptno,void interrupt(*isr)());

如果原中断号为 interruptno 的中断例程的入口地址为 old,则 setvect() 将对它 赋予一个新的值(假定为 vector),即用 vector 替代 old,该 vector 是一个远地址(4 个字节值), isr 则是指向一个中断函数的指针,该中断函数的地址为 vector。现在启动 interruptno 的中断例程时,用于指向中断例程的 CS:IP 将不是 old,而是 vector!由此可见,setvect()在传送地址。

它实际是 DOS 功能调用 25H。入口参数是,AH=0x25,AL=interruptno,DS:DX 中放 vector:

```

mov ah,25h
mov al,interruptno
push ds
lds dx,dword ptr vector
int 21h
pop ds

```

如果一个函数被定义为 interrupt 类型,则它的地址只能用 setvect() 传送。如果使用了 dos.h 中的原型,则可以在任何模式下简单地把中断函数地址通过 sevect() 传送。setvect() 不返回值,它只适用于 MS-DOS。

由于 BIOS 的缺省中断例程的地址都是精心安排的,因此当你对它们还不熟悉时,不要轻易用 setvect() 改变它们。

—9 void —Cdecl keep (unsigned char status,unsigned size);

它将当前程序驻留在内存中,程序所占空间为 size,内存其余部分被释放。它用于安

个 TSR 程序,返回 MS-DOS,并把出口状态保存在 status 中。status 也称退出码,它由 AL 寄存器传送,能被 keep 的父进程通过 INT 21H 的功能 4DH(检索子进程的返回码)访问。功能 4DH 调用在 AX 中返回另一进程指定的退出码:AL 由退出的程序发送,AH 的值的含义是:对于正常结束为 0;对于由 Ctrl-Break 引起的退出为 1;由于关键性设备出错引起的退出为 2;由于功能调用 31H 引起的结束为 3。

它相当于 DOS 中断 INT 21H 的功能 31H,终止进程并常驻。入口参数是,AH=0x31,AL=status,DX=size。只适用于 MS-DOS。

C>TYPE TSR.C

```

/* DOS TRS (内存驻留程序) 安全使用说明例程 */
/* 根据郭大伟、杨瑞萍的程序改写并注释 */
/* 程序可在集成环境的缺省值 (small 模式) 下编译后执行并驻留内存 */
/* 按 Shift-F1 组合键后被激活,在屏幕右上角显示:年一月一日一时:分:秒 */
/* 时间显示后退出 TSR 重新运行被中断的程序 */

#include "dos.h"
#include "stdio.h"
#include "graphics.h"

#define INT9 0x09 /* 键盘输入中断 9H */
#define ATTR 0x7000 /* 显示文本属性值 */
#define SHIFT-F1 84 /* Shift-F1 组合键值 */
#define STK-SIZE 0x2300 /* 堆栈尺寸 */
extern unsigned __stklen=1024; /* 定义堆栈的长度。缺省值为 0x1000 字节 */
extern void far * __heapbase; /* 定义一个指向栈段顶端之外的内存的地址 */
/* 一个字节的 far 指针,并且标出用来分配 */
/* 给 Turbo C 程序的内存末端。 */

void interrupt (*oldkeyisr)(void); /* 老键盘 TSR 中断矢量 */
void interrupt (*old-int28)(void); /* 老 INT 28H 中断矢量 */
void interrupt (*old-int8)(void); /* 老 INT 8H 中断矢量 */
void interrupt newkeyisr(void); /* 新键盘 TSR 中断矢量 */
void interrupt dos-busy-flag(void); /* DOS 忙标志 */
void interrupt new-int8(void); /* 新 INT 8H 中断矢量 */
typedef void interrupt (*VIFP)(void);
void time-disp(); /* 定义函数与变量 */
VIFP code; /* code 作为判断 TSR 是否早已安装标志 */
char far * dos-active;
struct date dat;
char bf[20];
int vmode,v,hotkey; /* vmode 是视频模式段地址,hotkey 是热键 */
char busy=0; /* 记录 DOS 是否忙,0 为空闲 (idle) */
char tmsk[]="%02d-%02d-%02d-%02x:%02x:%02x"; /* 定义显示时间格式 */
unsigned int sp,ss;
unsigned char stack[STK-SIZE];
main()
{

```

```

union REGS r;
struct SREGS s;
int gd, gm, i, found = 0;
code = (VIFP)0x11111111; /* code=1111:1111,本 TSR 的标识码 */
for(i=0x80;i<=0x81;i++) /* 0x80,0x81 是两个对用户是“自由的”中断矢量 */
if(getvect(i)==code)
    {printf("TSR 早已安装! ");return(0);}
for(i=0x80;i<=0x81;i++)
if(getvect(i)==(VIFP)0) /* 利用中断矢量是否为非 0 值可确定该矢量可用性 */
    {
        setvect(i,code);found=1;break;
    }
if(! found)
    {
        printf("无自由中断矢量可利用,所以TSR 未生成! \n");return(1);
    }
if(--osmajor<2)
    {
        printf("不能生成TSR,DOS 版本应大于2.0 ! \n");
        return(1);
    }
r.h.ah=0x34; /* 测试 DOS 代码是否活动的中断 */
int86x(0x21,&r,&r,&s);
dos--active=MK--FP(s.es,r.x.bx);/* 大于 0 表示 DOS 当前正忙, */
/* 等于 0 为空闲 */
detectgraph(&gd,&gm); /* 测试显示卡的类型 */
if(gd==HERCMONO)vmode=0xb000; /* 单色卡存储段位置 */
else vmode=0xb800; /* 彩色卡存储段位置 */
oldkeyisr=getvect(INT9); /* 取老 INT 9H 中断矢量当前 4 字节值 */
/* 保留在中断矢量 oldkeyisr 中 */
old--int28=getvect(0x28); /* 保留老 INT 28H 中断矢量值 */
old--int8=getvect(8); /* 保留老 INT 8H 中断矢量值 */
/* INT 8H 为定时器中断 */
setvect(INT9,newkeyisr); /* 把新中断函数 newkeyisr 地址赋给 */
/* 中断矢量 INT 9H,即 INT 9H 不再 */
/* 指向 oldkeyisr,而指向 newkeyisr */
setvect(8,new--int8); /* 8 号中断矢量指向新 new--int8 */
setvect(28,dos--busy--flag); /* 28 号中断矢量指向 dos--busy--flag */
getdate(&dat); /* 取得 MS-DOS 日期 */
printf("OK! TSR 被安装! \n");
/* resident--size 函数调用相当于 keep(0,FP--SEG(--heapbase)--psp) */
resident--size(FP--SEG(--heapbase)--psp);/* 整块代码和数据常驻内存 */
/* 驻留尺寸估计;--psp 为程序段前缀 PSP,它是位于内存中程序 */
/* 开始处具有 256 字节的含有系统信息的记录。FP--SEG 析取段值 */
}
void interrupt newkeyisr() /* 新键盘中断服务例程 */

```

```

{
unsigned char far *t;
t=MK-FP(0x0000,0x41A); /* 指向键盘缓冲区头指针 */
(*oldkeyisr()); /* 调用老键盘中断 */
if(*t! = *(t+2))
{
t+=*t-30+5;
if(*t==SHIFT-F1)
hotkey=1; /* Shift-F1 按下 */
}
}

void interrupt new-int8(void) /* 用系统定时器中断服务检测 */
{
(*old-int8());
if(! *dos-active && ! busy && hotkey)activate-sctsr(); /* 弹出 TSR */
}

void interrupt dos-busy-flag(void) /* 新 DOS INT 28H 中断服务检测 */
{
(*old-int28()); /* 此句必须有且应在最前面 */
if(! busy && hotkey)activate-sctsr(); /* 弹出 TSR */
}

activate-sctsr() /* TSR 入口函数。TSR 总是从这个初 */
{ /* 始被激活的函数中自动返回的 */
disable();
ss=--SS; /* 保存栈段 */
sp=--SP; /* 保存栈指针 */
--SS=--DS;
--SP=(unsigned)&stack[STK-SIZE-2]; /* 利用 Turbo C 栈 */
enable(); /* TSR 转到一个局部栈的固定位置,并使用静态 */
if(! busy){ /* 和外部变量来存储数据,以防止代码重进入 */
time=disp();
hotkey=0; /* 清除热键 */
}
disable();
--SP=sp; /* 恢复栈段 */
--SS=ss; /* 恢复栈指针 */
enable();
return 0;
}

/* 注意;不能用 exit() 来终止程序,因为 exit 函数是设计成终止非驻留 */
/* 程序后返回 DOS 的,使用它会把所有文件关闭,缓冲输出内容被写完, */
/* 所有已登记的“出口函数”被调用等。因此,用它从 TSR 中返回很可 */
/* 能会异常结束 TSR 和前台程序,可能产生不可预料的结果。 */
void time=disp() /* TSR 时钟显示程序 */
{
union REGS inr,outr;

```



```

    inr.h.ah=0x02;          /* BIOS 功能调用 */
    int86(0x1A,&inr,&outr);
    sprintf(bf,tmsk,dat.da--year,dat.da--mon,dat.da--day%100,outr.h.ch,
            outr.h.cl,outr.h.dh);
    for(v=0;v<19;v++) /* 如显示:1993-05-31-14:22:26,共 19 个字符 */
        poke(vmode,(60+v)*2,ATTR+bf[v]);
    }
    resident=size(unsigned size) /* DOS 内存驻留功能调用 */
    {
        union REGS r;
        r.h.ah=0x31; /* 功能号 */
        r.h.al=0; /* 退出码 */
        r.x.dx=size; /* 驻留内存大小,分配给程序的段落总数。段的单位为“节” */
        int86(0x21,&r,&r);
        return 0;
    } /* r.h.al 传递的退出码可由其父进程通过 WAIT(INT 21H 的功能 4DH) */
        /* 的访问,且能被批处理命令 ERROR LEVEL 所测试判别。 */
    /* 说明:

```

1. 图 22-4 说明了 TSR 被激活的过程。

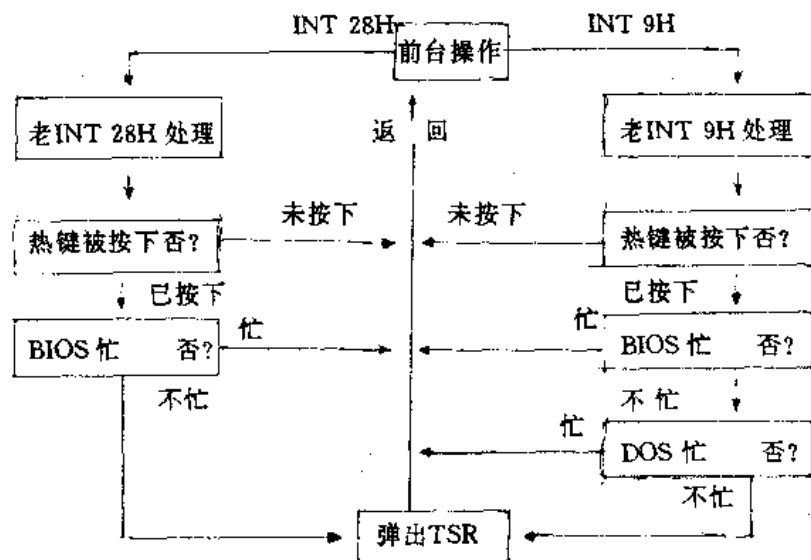
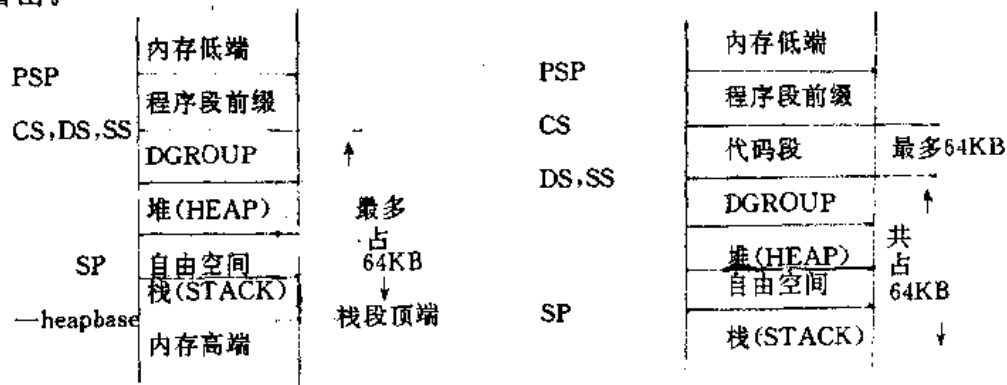


图 22-4 弹出 TSR 的并行机制

2. `--heapbase` 是一个内存管理变量 (Memory mangement variable), 也是一个全局变量。它指向栈段顶端之外的内存的第一个字节的 far 指针。它与内存模式的关系可由图二 22-5 看出。



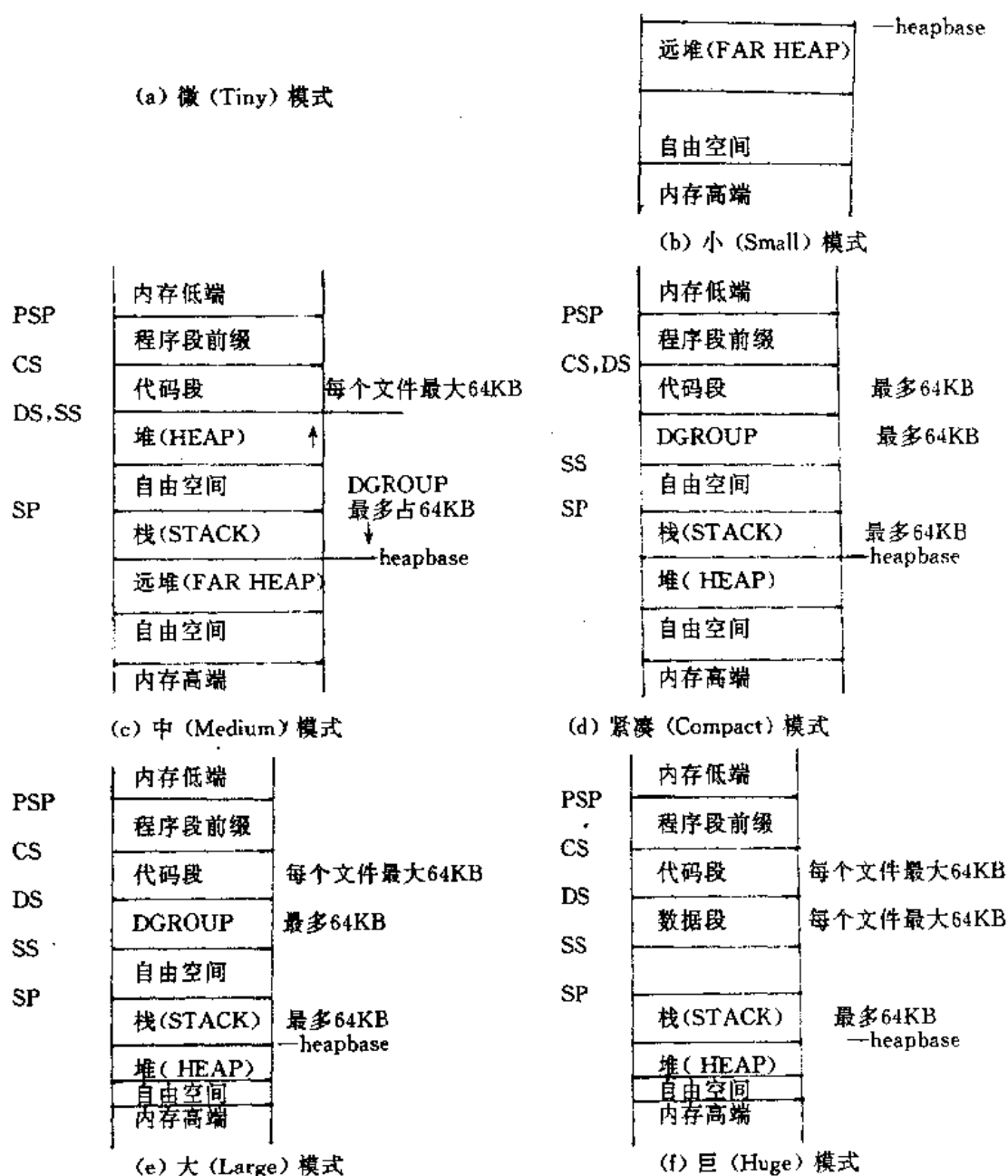


图 22-5 Turbo C 的内存结构中的堆、栈和远堆

从图 22-5 可以看出,前三种存储模式 (称【小数据模式】) 中, `-heapbase` 指向 作为已知道的远堆但又未分配内存的开始处。它们使用的堆 (参见《动态地址分配函数》一章) 位于数据段末端与栈之间,是在已分配的内存区域之内。因此,TSR 可以安全地调用 `malloc()` 等函数,因为函数使用的内存存在驻留块之内。TSR 也可自由地使用全部 Turbo C 栈。一般情况下,这些堆要占 70KB 内存。由此可见,如果 TSR 没有使用堆,将造成内存很大的浪费。为避免浪费,还可适当栈的大小 (它由另一个内存管理变量 `-stklen` 定义,其缺省值是 4KB)。

对后三种模式 (称【大数据模式】), `-heapbase` 指向用来分配动态内存的实际堆。本程序由于 `-heapbase` 指向栈段顶端之外的内存的第一个字节的 `far` 指针,也就是 Turbo C 程序加载时开辟的内存末端绝对地址,因此 TSR 模块将不允许在大数据模式中进行动态分配。如果要动态分配,应给另一个内存管理变量 `-heaplen` 赋值 (它的缺省值为 0),并在计算内存驻留尺寸时考虑它。

利用 MS-DOS 5.0 的 MEM.EXE 程序可测得在 6 种内存模式下本 TSR 驻留内存的大小 分别是: 65936 (Tiny)、87632B (Small)、87840B (Medium)、36784B (Compact)、37040B (Large) 和 37 776B (Huge)。最大是 85KB, 最小是 35KB, 相差甚大。

3. 此程序如在 LX-386/33S 机上编译后使它运行一次驻留内存, 然后在五笔字输入方式下再编辑 (可以进行中西文直接编辑), 选 Alt-F-O 退出时便看到:

Internal Error--Please report to Boland. Press ESC

的错误提示。压 ESC 键后发现

```
exit error=3, winerror=0
```

再键入 exit 便不起作用了。这种错误一般是不易看到的 (当然, 这不是 Turbo C 的问题)。

4. 若用 F7 进行单步调试, 则执行到 resident-size 便终止, 调试结束程序将占用内存, 然而不能被激活。中间可能得到这样一些结果 (使用 Ctrl-F4 调试表达式):

```
oldkeyisr=1488;1952
old-int28=0028;40C5
old-int8=12CE;003C
newkeyisr=-newkeyisr
new-int8=-new-int8
dos-busy-flag=-dos-busy-flag
```

5. 本程序在一定程度上克服了 DOS 的不可重入性 (Non-reentrant)。所谓【不可重入性】是指操作系统核心不可在任意点处中断后使其代码为另一个进程所重新使用 (代码不能被递归地重进入), 或者说, 当 TSR 中断一个 DOS 功能后, 不允许紧接着又调用一个 DOS 功能 (注意: Turbo C 不少库函数是使用 DOS 功能的), 否则系统可能会出问题。本程序把 TSR 操作限制在内存范围和 BIOS 资源内运行 (如直接访问视频缓冲区)。

6. 本程序只能在文本方式下使用, 例如在进入 TC 集成环境前可让它驻留内存。也可以使它与 THELP.COM 同时驻留内存, 然后能分别被激活。THELP.COM 可用 /U 参数释放占用的内存, 而本 TSR 则不能。更进一步, Turbo C 集成环境的 file 主菜单下, 用 OS shell 命令和用 Quit (Alt-X) 是不同的, 前者将继续占用多至 328KB 的内存 (即有部分程序驻留内存), 而后者则全部释放这些内存。

7. 作为试验, 先将上述程序驻留内存, 然后用 DOS 的 TYPE 命令列程序的源程序, 中间如按 Shift-F1 键, 便可看到屏幕右上角显示出日期和时间。

第二十三章 串行通讯

23.1 RS-232

RS232 接口 (RS232 interface) 是美国电子工业协会所规定的一种标准化接口。它是调制解调器与其它相连的数据终端之间的接口。

注意:因为标准 BIOS 接口对数据通信支持不足,为满足不同厂商又开发了许多功能更强的例程。但是,由此可能发生重复与冲突。例如,3 COM BAPI 串行 I/O、FOSSIL 驱动程序,以及网络软件等都有可能使结果不同。

1. 各针信号特性

一些制造厂家提供的 RS-232 信号是:

名称	针号	用途
RTS	4	请求发送 (Request to send)
CTS	5	清除发送 (Clear to send)
DSR	6	数据设备准备好 (Data set ready)
DTR	20	数据终端准备好 (Data terminal ready)
TXD	2	发送数据 (Transmit data)
RXD	3	接收数据 (Receive data)
GDR	7	地 (Ground)
GND	1	地
DCD	8	数据载波检测

对于音频调制解调器,接口引线可以是具有公共地线的单线。该接口规定用双极性信号,负向信号表示“1”,正向信号表示“0”。该接口的最大允许电压为 ± 25 伏。驱动源信号的典型幅度为 $\pm 6 \sim 10$ 伏,内阻为数百欧姆。终端最小信号幅度为 ± 3 伏。该设备和地之间的电压差最多可为2伏,此时驱动源信号幅度必须不小于 ± 5 伏。终端阻抗要求为 $3000 \sim 7000$ 欧姆。

在 TXD、RXD 线上,MARK(表示为1) $= -3V \sim -25V$,SPACE(表示为0) $= +3V \sim +25V$ 。

在 RTS、CTS、DTR 等线上,有效(ON) $= +3 \sim -25V$,无效(OFF) $= -3V \sim -25V$ 。

RS232 任何一脚可直接与任何别的针脚相联,一般不会损坏器件。

RS-232 的许多信号是为了和调制解调器相互通讯而定义的。当通讯是在计算机之间利用串行口直接进行,则可以直接用 TXD、RXD 和 GDR 三根线就够了(但软件中应考虑“软件握手”)。当然,也可用并行口,它比较方便。

2. 微机间通信的接线方式

两台微机间可以通信,其接线方式如图 23-1 所示。

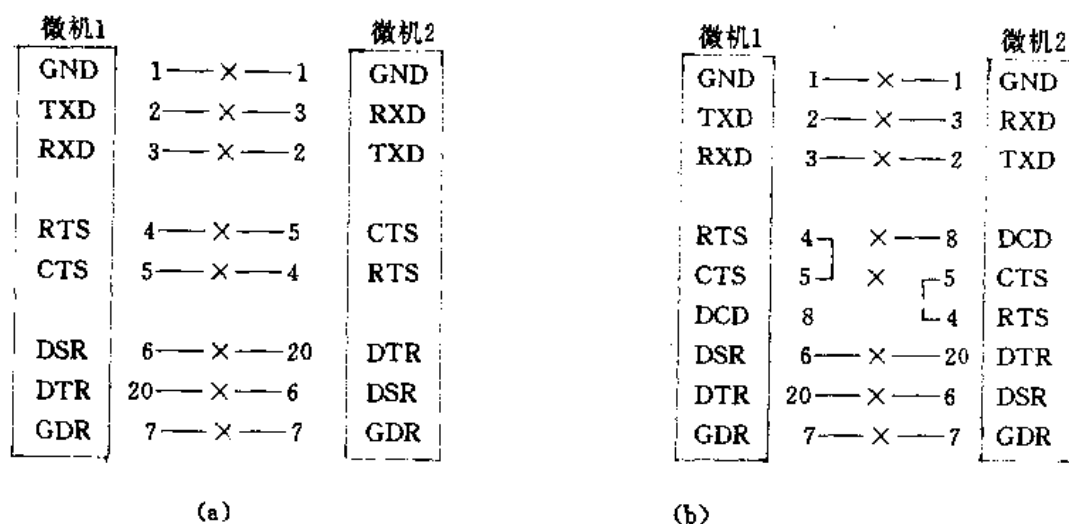


图 23-1 两台微机间接线

3. 异步通信过程中的数据格式

传送一个字符的过程如图 23-2 所示。

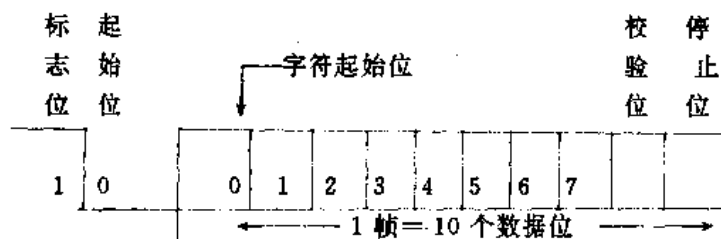


图 23-2 数据格式图

4. 传送方式

- (1) 半双工方式: 每次只能有一个发送端, 一个接收端, 而不能同时接收与发送。
- (2) 双工方式: 一端能同时发送和接收。

5. 传送速度

每秒传送数据位数称波特率 (baud rate)。假定传送一个字符要传 10 个数据位 (1 帧数据), 每秒要传送 960 个字符, 则应采用 $960 \times 10 = 9600$ (位 / 秒) = 9600 波特率。

若有现成的通用传输文件, 如 TDREMOTE.EXE/TDRF.EXE 等就能够很方便地把文件从一台微机传到另一台上。DR DOS 6.0 的 RS232 的阴插头分别插在两机的串行口 COM1 (或 COM2) 上, 使用 9 芯电缆, 长 4 米以内, 使用 FILELINK.EXE 程序。方法如下:

主从两机上均拷入文件 FILELINK.EXE。

先对从机操作: C>FILELINK SET COM1:115200

C>FILELINK SLAVE

从主机向从机发送文件:

C>FILELINK SET COM1:115200

C>FILELINK TRA 要传送的文件名

C>FILELINK QUIT

从机接收文件:

C>FILELINK SET COM1:115200

C>FILELINK REC 要传送的文件名

C>FILELINK QUIT

说明:115200 最高波特率大约 1 秒钟可传送 1.1 万字节左右。

23.2 库函数 bioscom()

int —Cdecl bioscom(int cmd, char abyte, int port);

用来操作由 port 指定的 I/O 端口上进行各种 RS232 异步通讯。

port 为 0 表示 COM1, 为 1 表示 COM2, 等等。

通讯类型取决于 cmd 的值 (它实际是中断 INT 14H 的子功能号);

- 0 初始化端口 port
- 1 按字节发送一个字符到通讯线上 (写字符到端口 port)
- 2 从通讯线上接收一个字符 (从端口 port 读字符)
- 3 返回接口的当前状态

在使用串行接口之前,你想不用其缺省的工作方式,而是把它初始化为某种状态,这就是初始化接口。初始化时指定 cmd 等于 0,并将 abyte 按下述规定设置:

位1 和位0 当位1=1、位0=0 时,使用 7 个数据位,相当于 00000010=0x02
当位1=1、位0=1 时,使用 8 个数据位 =0x03
为其它值时无效

如果要传送一个英文文件,用 7 个数据位就够了,因为英文字母和符号不需最高一位,这样还可提高文件传送速度。对其余文件最好用 8 位传送。为表示文件发送结束,不能使用一般的文件结束标志 (EOF),而应该在发送一个文件以前先把文件的总字节数发送到对方。

位2 当其值为 1 时,使用 2 个停止位,相当于 00000100=0x04
否则用 1 个停止位 =0x00

停止位通常在奇偶校验位之后,它可以占一个或两个数据位的时间宽度。停止位是低电平,它表示一个字节传送的结束。在停止位之后,可以接着又是下一个字节的起始位,也可以在停顿任意长时间之后,才是下一字节的起始位。因此停止位也就确定了相邻两个字节之间的最短停顿时间。究竟用几个停止位不是很主要的,关键是发送方和接收方要一致。

位4 和位3 当位4=0、位3=1 时进行奇校验,或相当于 00001000=0x08
当位4=1、位3=1 时进行偶校验 =0x18
为其它值时不校验

在数据位之后是奇偶校验位。奇偶校验位是用来检查在传送过程中是否有错误。偶校验是将各数据位上的数值再加上校验位,其和将是一个偶数;对奇校验,该和数为一个奇数。

位7、位6 和位5 111=9600 波特 或相当于 11100000=0xE0
110=4800 波特 =0xC0
101=2400 波特 =0xA0
100=1200 波特 =0x80
011= 600 波特 =0x60
010= 300 波特 =0x40
001= 150 波特 =0x20
000= 110 波特 =0x00

数据收发的速度是用波特率来衡量的,波特率是每秒钟传送信息位的数量。它是所传送代码的最短码元占有时间的倒数。例如一个代码的最短信号码元宽度为 20 毫秒,则其波特率

是每秒 50 波特。IBM PC 机的波特率最高可达 9600。像 DMP-42 绘图机则要求 9600 波特。

波特是信号传输速度的一种单位。它等于每秒内离散状态或信号事件的个数。在每个信号事件表示一个二进制位的情况下,波特和每秒比特数一样。在异步传输中,波特是调制率的单位,它是单位间隔的倒数。若单位间隔的宽度是 20 毫秒,则调制率是 50 波特。

例如,语句 `bioscom(0,251,0);` 的意思是,初始化通讯口为 9600 波特、偶校验、1 个停止位和 8 个数据位。因 $251=0xFB$,即 $(0xE0 | 0x18 | 0x00 | 0x03)$,该表达式也可作为监视表达式使用而得值 251。显然,后者书写直观明了。

当 `cmd=1` 时,abyte 为要写的字符。在 `cmd=2` 或 3 时,可将它写 0 (实际该参数被忽略)。

函数总是返回一个 16 位的数值。其高 8 位是描述线路状态 (line status) 位,该字节各位的含义是:

- 位15 超时错误 (Time-out error)
- 位14 发送移位寄存器空 (Transfer shift register empty)
- 位13 发送保持寄存器空 (Transfer holding register empty)
- 位12 中止检测错误 (Break-detect error)
- 位11 结构错误 (或帧错误, Framing error),如果接收双方中有一方时钟发生了偏差,就会发生接收方和发送方不协调而产生帧错误
- 位10 奇偶错误 (Parity error)
- 位 9 【过冲错误】(Overrun error)。

过冲错误的一个具体例子是,假定计算机 A 的速度比计算机 B 的速度快,当计算机 B 还未读完计算机 A 发来的第一个字节,计算机 A 又发来了第二个字节,这就导致“过冲”出错。解决这问题的一个方法是采用所谓的【软件握手】。其工作过程是,发送方发送完第一个字节后,就一直等待着从接收方收到一个“认可”字节。只有在收到这个“认可”字节后,发送方才发送第二个字节。

位 8 数据准备就绪 (data ready)

它的低 8 位随 `cmd` 值而定,常用于描述调制解调器 (Modem Status) 状态:

- `cmd=2` 若没有错误发生,低 8 位中存放由通讯口接收到的数值,否则至少有一高位位置位。
- `cmd=0,1 或 3` 低 8 位含义为:
 - 位7 线路信号被检测 (CD, Line signal detected)
 - 位6 响铃指示 (RI, Ring indicator)
 - 位5 数据设备准备好 (DSR, Data set ready)
 - 位4 清除发送 (CTS, Clear-to-send)
 - 位3 线路信号变化 (Change in line signal)
 - 位2 脉冲后沿响铃检测 (Trailing-edge ring detector)
 - 位1 数据设备准备好信号变化 (Change in data-set-ready)
 - 位0 清除发送信号变化 (Change in clear-to-send)

当使用标准 BIOS 接口时,本函数实际相当于 BIOS 中断 INT 14H 的调用。

```
int bioscom(int cmd,char abyte,int port)
{union REGS r;
 r.h.ah=cmd;
 r.h.al=abyte;
 r.x.dx=port;
 int86(0x14,&r,&r);
}
```

本函数只适用于 IBM PC 及其兼容机上。

C>TYPE PORT.C

```
#include "stdio.h"
#include "bios.h"
#include "conio.h"
#include "ctype.h"
#define PORT 0 /* 本程序通过串行口 0 (第一个串行口) 传送文件 */
void sport(int port,char c) /* 发送一个字节 */
{
    if(bioscom(1,c,port) & 0x80) /* 第 7 位为 1 时出错 */
        {printf("error\n");exit(1);}
}
rport(int port) /* 接收一个字节 (从端口 port 读一个字符) */
{
    int a;
    while( ! (bioscom(3,1,port) & 0x100) ); /* 当“数据就绪”时第 8 位为 1 */
    if(kbhit()) { getch();exit(1); } /* 按键退出 */
    a=bioscom(2,1,port) & 0x80; /* 如果不发生错误,返回的低 8 位为读入字节 */
    /* 否则,至少有一高位被置位。但也可从第 7 */
    /* 位的值检测调用是否成功 (为 0 表示成功) */
    if(a){printf("error\n");}
    return (a & 0xff); /* 返回低 8 位,即读入字节 */
}
void wait(int port) /* 发送方送出一个字节后,等待接收方的应答 (认可) */
    /* 信号,只有在收到“认可”字节后,发送方才送第二个 */
{
    /* 字节,这就是“软件握手”。 */
    if(rport(port) != '.') /* “认可”信号为一个小数点 */
        { printf("error! \n");exit(1); }
}
void send—name(char *f) /* 送文件名 */
{
    printf("等待传送...\n");
    do{
        sport(PORT,'? '); /* 发送一个 ? 号 */
    }while( ! kbhit() && ! (bioscom(3,1,PORT) & 0x100));
    if(kbhit()) { getch();exit(1); }
    wait(PORT); /* 等待接收方信号 */
    printf("传送文件名%s\n\n",f);
    while(*f) /* 传送文件名 */
    { sport(PORT,*f++); /* 送文件名的一个字节 */
      wait(PORT); /* 等待接收方信号 */
    }
    sport(PORT,'\0'); /* 送空字符终结符 */
}
void send—file(char *fname) /* 发送文件名、文件长度和文件本身 */
{

```



```

FILE *fp;
char ch;
int handle;
union{ char c[4]; /* 设置联合的目的是因为串行口一次只能送一个字节 */
      unsigned long count; /* 文件长度占 4 个字节 */
    }cnt;
if( (fp=fopen(fname,"rb"))==NULL) /* 打开文件 */
{ printf("不能打开输入文件! \n");exit(1); }
handle=filenr(fp); /* 取得文件句柄 */
send_name(fname); /* 送文件名 */
wait(PORT); /* 等待接收方信号 */
cnt.count=filelength(handle); /* 取得文件大小并发送 */
sport(PORT,cnt.c[0]); /* cnt.c[0]=cnt.count 低 8 位 */
wait(PORT);
sport(PORT,cnt.c[1]); /* cnt.c[1]=cnt.count 高 8 位 */
wait(PORT);
sport(PORT,cnt.c[2]); /* cnt.count 的 16~28 位 */
wait(PORT);
sport(PORT,cnt.c[3]); /* cnt.count 的 29~31 位 */
do{
    ch=getc(fp); /* 取文件中一个字符 */
    if(ferror(fp)){printf("有错误\n");break;}
    if( ! feof(fp))
    {
        wait(PORT);
        sport(PORT,ch); /* 发送文件的一个字符 */
    }
}while( ! feof(fp));
wait(PORT);
fclose(fp); /* 发送完后关闭文件 */
}

void get_name(char *f) /* 接收文件名 */
{
    printf("接收等待...\n");
    while(rport(PORT) != '? '); /* 检查端口是否发来? */
    sport(PORT,' '); /* 是?后给发送方送应答信号 */
    while(( *f==rport(PORT))) /* 依次送文件名的字符 */
    { /* 文件名最后以空字符 0 结束 */
        if( *f != '? ){ /* 滤去?,使它不进入文件内 */
            f++;
            sport(PORT,' '); /* 给发送方送应答信号 */
        }
    }
}

void rec_file() /* 接收一个文件的文件名、文件长度和本身 */
{

```

```

FILE *fp;
char ch;
char fname[14];
union { char c[4];
        unsigned long count;
    }cnt;
get--name(fname); /* 接收一个文件名 */
printf("接收文件名为:%s\n",fname);
remove(fname); /* 注意:如果当前盘上有和 fname 同名文件存在,将被删掉 */
/* 如文件不存在,函数 remove() 可返回出错信息,但无碍 */
if((fp=fopen(fname,"wb"))==NULL) /* 创建新输出文件 fname */
{ printf("不能打开输出文件\n");exit(1);}
sport(PORT, '.');
cnt.c[0]=rport(PORT); /* 获得 cnt.count 的值,即文件长度 */
sport(PORT, '.');
cnt.c[1]=rport(PORT);
sport(PORT, '.');
cnt.c[2]=rport(PORT);
sport(PORT, '.');
cnt.c[3]=rport(PORT);
sport(PORT, '.'); /* 根据文件长度确定是否接收完毕 */
for(;cnt.count;cnt.count--) /* 相当于 for(;cnt.count>0;cnt.count--) */
{
    ch=rport(PORT);
    putc(ch,fp); /* 写入文件一个字符 */
    if(ferror(fp)) { printf("有错误\n");exit(1); }
sport(PORT, '.');
}
fclose(fp); /* 关闭文件 */

/* 本说明的例程在实用时要作修改 */
main(int argc, char *argv[]) /* 为发送文件,应键: */
{ /* C>本执行文件名 S 被发送文件名 */
    int cmd, port; /* 如接收文件 应键: */
    char abyte; /* C>本执行文件名 R */
    if(argc<2)exit(1);
    bioscom(0,231,PORT); /* 初始化:9600 波特,无奇偶校验,8 位数据,2 停止位 */
    if(tolower(*argv[1])=='s'&&argc==3)send--file(argv[2]); /* 发送文件 */
    else if(tolower(*argv[1])=='r')rec--file(); /* 接收文件 */
    printf("命令行上键入的命令不正确! 请重新键入正确命令\n");
}

```

第二十四章 控制内存块

类似于 DOS 的 DEBUG 的 C(比较)、F(填充)及 M(传送)等命令, Turbo C 允许直接在用户程序中对内存数据块操作。这类库函数原型主要在 mem.h 和 string.h 标头文件中。事实上,串变量的内容也可以看成特殊的内存块,因此,除 movmem() 和 setmem() 外,其余的库函数也可以用于字符串操作(参见《数组与字符串》一章)。

24.1 分类

1. 内存块拷贝

- 从源串中拷贝若干字节到目的串中,返回指向目的串的指针 —1 memcpy()
- 当源中指定字符已拷或已拷了指定个数的字节后拷贝停止 —2 memmove()
- 从源中移动指定字节的块到目的存储单元中,不返回值 —3 memccpy()
- 以段;偏移量方式拷贝,不返回值 —4 movmem()
- 以段;偏移量方式拷贝,不返回值 —5 movedata()

2. 内存块比较

- 精确比较(指大小写字母是不同的)两个串的前 n 个字节 —6 memcmp()
- 忽略大小写字母的区别,比较两个串的前 n 个字节 —7 memicmp()

3. 内存块置值

- 将块前若干字节都设置为指定字符,返回指向块的指针 —8 memset()
- 将块前若干字节都设置为指定字符,但无返回 —9 setmem()

4. 搜索内存块

- 从指定块前若干字节中搜索指定字符 —10 memchr()

24.2 库函数

—1 void * —cdecl memcpy(void *dest, const void *src, size_t n);

—2 void * —cdecl memmove(void *dest, const void *src, size_t n);

这两个函数将源 src 中的 n 个字节拷贝到目的串 dest 中。如果源串与目的串有重迭,函数本身会选择正确的拷贝方向,使重迭处能正确拷贝。注意: dest 应有足够空间容纳 n 个字符, n 不应大于源串 src 的长度。

数据类型 size_t 在 mem.h 中定义为

```
typedef unsigned size_t;
```

返回指向 dest 的指针。注意:函数前 void 修饰符,它表明指针可以指向任一数据类型,因而它所指的结构或数组的尺寸是不知道的(Size of structure or array not known),所以你不能对这种指针进行像增 1(++)或减 1(--)那样的间接运算。

适用于 UNIX 系统 V。

C>TYPE MEM1.C

```
#include "mem.h"
#define COPYOVER  p1=memcpy(y,x,n); /* 拷贝重迭 */ \
                  printf("%s %s %s\n",y,x,p1)
                  /* 注意:注释应写在续行符前 */

main()
{
int u=2.*s0=&u;
char *s="ABCD",t[]="efg";
static char x[8];
char *y=x+1;
void *p1,*p2;
int n=2;
p1=memcpy(x,s,n);
printf("%s %s %s\n",p1,x,s); /* 程序输出:AB AB ABCD */ p1=memcpy(x,t,n);
printf("%s %s %s\n",p1,x,t); /*      ef ef efg */
p2=memcpy(s,s,n);
printf("%s %s\n",p2,s);      /*      ABCD ABCD */
s++;
p2=memcpy(s,s,n);
printf("%s %s\n",p2,s);      /*      BCD BCD */
printf("%s\n",s-1);          /*      ABCD */
COPYOVER;                    /*      ef eef ef */
COPYOVER;                    /*      ee eee ee */
/* printf("%s\n",p2-1); Error:Size of structure or array not known */
p2=memcpy(x,s0,5);
/* 调试表达式值:

                拷贝前                拷贝后
*s0,5mx: 02 00 D2 FF 65      *s0,5mx: 02 00 D2 FF 65
p2,5mx: 99 01 D2 00 EC      p2,5mx: 66 04 02 00 EC
x,5mx: 65 65 66 00 00      x,5mx: 02 00 D2 FF 65

*/
}
```

—3 void *—Cdecl memcpy(void *dest,const void *src,int c,size_t n);

将内存源存储区(src所指)最多n个字节(称【存储块】)拷贝到目的存储区dest中。dest应有足够的空间容纳拷贝的存储块,否则将是不可预料的。在拷贝时如果遇到指定字符c(它可用ASCII码值表示),当c被拷贝入dest后,拷贝便停止。如果c不在源存储区src中,则最多拷贝n个字节后拷贝便停止。

如果c已被拷贝,函数返回指向dest中紧跟c以后的字符的指针;否则返回NULL。

n不应大于源存储区中要拷贝的字符数,否则由于Turbo C不对数组下标检查,而指针只指向内存块头部,因而可能会将不该拷贝的内容拷贝,或者说结果不可预料。为拷贝内存串,可将空字符('\0')作为停止拷贝字符c。

适用于UNIX系统V。

C>TYPE MEM2.C

```

#include "mem. h"
#define MAX 10
void * printclsdest(char * dest,const void * src,int c,size_t n,int flag),
main()
{
char src0[]="ABCDEFGH";
char src1[]="abcdefgh";
char src[6][9];
char d[]={'x','y','z','\0','f','\0'};
char * dest=(char *)calloc(MAX); /* 分配时将 dest 清 0 */
int c='E',k=0;
size_t n=8;
void * p;
for(k=0;k<6;k++) /* 给数组赋初值 */
    if(k%2==0)strcpy(src[k],src0);
    else strcpy(src[k],src1);
printf("dest = %s\n",dest);
strcpy(dest,d);printf("dest = %s\n",dest);
printclsdest(dest,src[5],'e',2,1);
printclsdest(dest,src[0],c,n,1);
printclsdest(dest,src[1],69,8,1);
printclsdest(dest,src[2],'e',8,1);
c='B';
printclsdest(dest,src[3],c,1,0);
printclsdest(dest,src[4],c,8,0);
printclsdest(dest,src[5],'e',3,0);
}
void * printclsdest(char * dest,const void * src,int c,size_t n,int flag)
{ /* 打印及初始化 dest */
int i=MAX;
char * ptr;
if(flag) /* flag 为 1 表示在拷贝前清内存块 dest */
    while(--i>=0) dest[i]='\0';
ptr=(char *)memcpy(dest,src,c,n);
if(ptr != NULL)
    { if (*ptr=='\x0')printf("\n\n");
      else printf("= %s = ",ptr);
    }
else printf("NULL ");
printf(" %s\n",dest);
}
/* 程序输出,dest =
    dest == xyz
    NULL ab
    "" ABCDE
    NULL abcdefgh

```

```

NULL ABCDEFGH
NULL aBCDEFGH
=CDEFGH= ABCDEFGH
NULL abcDEFGH    */

```

—4 void —Cdecl movmem(void *src,void *dest,unsigned length);

它将源内存数据块 src 的前 length 个字节拷贝到目的内存数据块 dest 中。如果源和目的字符串发生重迭,它会选择拷贝方向,使得数据正确地被拷贝。

它跟 memcpy()、memmove() 不同的是它并不返回任何值。前者在 mem.h 和 string.h 中均有原型,而本函数只在 mem.h 中有原型,表明它只用于内存块拷贝。

同样,你应当选择正确的 length,以免拷贝发生意外。

C>TYPE MEM3.C

```

#include "mem.h"
main()
{
char *ptr;
static char dest[]="3456789";
static int y=100;
char src0[]="ABCD",src1[]="abcdefghijk";
int t;
movmem(src0,dest,4);
/* y=(int)movmem(src0,dest,4); Error: Not an allowed type */
printf("%s %s\n",src0,dest);
movmem(src1,dest,2);
printf("%s %s\n",src1,dest);
printf("原来y=%d,地址=%p\n",y,&y);
movmem(src1,dest,10); /* length 超过 dest 长度时 */
printf("%s %s,dest 地址=%p\n",src1,dest,&dest[0]);
printf("新的y=%d,",y);
printf("它相当于两个字母ij=%d 的值\n",'ij');
} /* 程序输出:ABCD ABCD789
          abcdefghijk abCD789
          原来y=100,地址=019C
          abcdefghijk abcdefghijABCD,dest 地址=0194
          新的y=27241,它相当于两个字母ij=27241 的值 */

```

—5 void —Cdecl movedata (unsigned srcseg,unsigned srcoff,unsigned dstseg,
unsigned dstoff,size_t n);

它从源地址 srcseg:srcoff(段;偏移量)拷贝 n 个字节到目的地址 dstseg:dstoff 中。在微型、小型或中型存储模式中,由于数据段的地址没有显式给出,因此用本函数对于移动远(far)地址中的数据是非常有用的。

memcpy() 可以用在紧凑、大型或巨型存储模式中,这时内存段地址是隐式给出的。

C>TYPE MEM4.C

```

#include "mem.h"
#define COLOR—BASE 0xb800
void save—color—screen(char near *buffer)

```

```

{
movedata(COLOR—BASE,0,—DS,(unsigned)buffer,80*25*2);
puts(buffer); /* 在小存储模式下,使用此句便可将原屏幕内容再现,并不断发出喇叭蜂鸣声。buffer
               [0],128mx: 54 07 68 07 65 07 ... */
}
main()
{
char buf[80*25*2];
save—color—screen(buf);
}
—6 int *—Cdecl memcmp(const void *s1,const void *s2,size—t n);

```

逐个比较内存块 s1 和 s2 的前 n 个字节值。即首次先从 s1 中取出第一个字节中的值 *s1 和从 s2 的第一个字节中取出值 *s2 进行比较,有三种情况:

若 *s1 > *s2,则比较停止,返回 *s1 - *s2 的值;

若 *s1 = *s2,如果不是第 n 个字节则继续往下比较,否则比较结束并返回 0;

若 *s1 < *s2,则比较停止,返回 *s1 - *s2 的值。第一个字节比较结束后大家取第二个字节的值进行比较,如此等等,直到比较完 n 个字节。

注意:比较时是把字节值看成是无符号的,因此 memcmp("\xFF","\x7F",1) 将返回一个大于 0 的值。

它适用于 UNIX V。

```

C>TYPE MEM5.C
#include "mem.h"
#include "stdlib.h"
#include "dos.h"
main()
{
char s[][4]={"A","a","AB","AbC","\xFF","\x7F",""};
int j,k,n,m,p;
for(k=0;k<7;k++)
for(j=0;j<7;j++)
{
n=min(strlen(s[k]),strlen(s[j]));
m=memcmp(s[k],s[j],n);
if(m==0) p=C;
else p=m>0? 1:-1;
printf("%d %d: %s %s: %d %d %d\n",k,j,s[k],s[j],n,m,p);
delay(1000); /* 延时供观察输出结果 */
}
}

```

—7 int —Cdecl memicmp(const void *s1,const void *s2,size—t n);

它的功能跟 memcmp() 类同,不同的只是忽略两串 s1 和 s2 中字母的大小写区别,均按大写对待。对非字母一律按它们的 ASCII 码值比较。

适用于 NUIX 系统。

C>TYPE MEM6.C

```

typedef unsigned size_t;          /* 注意,本程序未用 mem.h */
int memicmp(const void *s1,const void *s2,size_t n) /* 等效函数 */
{
    int dif;
    for(;n-->0;((unsigned char *)s1)++,((unsigned char *)s2)++)
        /* 上句中不能去掉 unsigned 等类型强制转换,因 s1 和 s2 为 void */
        {dif=toupper(* (unsigned char *)s1)-toupper(* (unsigned char *)s2);
        /* 类型强制转换是必要的,否则会出现 Error: Not an allowed type */
        if(dif != 0)return (dif); /* 如果字符对不等,停止比较,返回差值 */
        }
    return (0);                  /* 相等,返回 0 */
}
main()
{
    static char *s="ABCD",t[]="ab";
    int n=2,m;
    m=memicmp(s,t,n);
    printf("m=%d\n",m);
} /* 程序输出:m=0 */

```

—8 void *Cdecl memset(void *s,int c,size_t n);

设置内存数据块 s 前 n 个字节都为 c 的低字节值。注意,在使用 n 时应谨慎。例如,当 s 是串时, n 最好不要超过其长度。

用它或下述的 setmem() 可将内存区域初始化一个已知值,以扫除指定内存区域中的“垃圾”,即原有的一些无用的数据。适用于 UNIX V。

C>TYPE MEM7.C

```

#include "mem.h"
main()
{
    void *t,*u="UVWXYZ";
    void *s=(char *)0x0194; /* 用调试表达式 u,p;0194 知 s 指向 u */
    void *w="xyz";
    int n=4,c='A'; /* 注意:c='AB' 与 c='A' 是一样的 */
    t=memset(u,c,n);
    printf("t=%s u=%s\n",t,u);
    t=memset(s,'d',n);
    printf("t=%s s=%s\n",t,s);
    t=memset(w,c,6);          /* n=6 大于串 "xyz" 的长度 3 */
    printf("t=%s w=%s\n",t,w);
} /* 程序输出:t=AAAAYZ u=AAAAYZ

```

t=ddddYZ s=ddddYZ

t=AAAAAA%s u=%s 由于不知指针 w 所指串 "xyz" 后的情况引起 w =
AAAAAA%s u=%s */

—9 void Cdecl setmem(void *dest,unsigned length,char value);

它将 dest 所指的内存数据区前 length 个字节都赋值 value。但它跟 memset() 不同,它

不返回任何值。

```
C>TYPE MEM8.C
#include "mem.h"
main()
{
    char *ptr;
    int len;
    static char dst[6]; /* dst[0]~dst[6] 全为 0 */
    char src[]="ABCD";
    char value='Y';
    setmem(src,2,value);
    /* len=setmem(src,2,value);将引起Error:Not an allowed type */
    printf("%s\n",src); /* 输出 YYCD */
    setmem(dst,2,value);
    printf("%s\n",dst); /* 输出 YY */
}
```

—10 void *—Cdecl memchr(const void *s,int c,size_t n);

在 s 所指的内存块前 n 个字节中搜索指定字符 c(c 的低字节值)。如果搜索成功,返回一指向 s 中首次出现 c 的指针;否则,返回 NULL,表示未找到。

搜索不会改变 s 本身。在选择 n 时应谨慎,例如当 s 为数组时,自然 n 不应超出其长度。适用于 UNIX V。

```
C>TYPE MEM9.C
#include "mem.h"
#define Pptr printf("%s\n",ptr)
main()
{
    char s[]="ABACDEd";
    char t[10];
    char *ptr=t;
    setmem(t,10,'D');
    setmem(&t[10],1,'\0');
    Pptr;
    ptr=memchr(s,'AC',8); /* 'AC' 相当于'A',只有低字节起作用 */
    Pptr;
    ptr=memchr(s,0x45,8); /* 0x45='E' */
    Pptr;
    ptr=memchr(s,'D',10);
    Pptr;
    ptr=memchr(s,'W',10);
    Pptr;
    /* 程序输出:DDDDDDDDDD
                ABACDEd
                EDd
                DEDd
                (null) */
}
```

第二十五章 动态地址分配函数

Turbo C 函数中的局部变量占用内存空间是动态的,即当调用函数时才给它分配空间。当函数调用结束,变量所占空间将自动被释放,从而允许其它变量使用。动态地址分配内存是指使用动态地址分配函数从一块自由内存中(常称【堆】,HEAP,有关堆在内存中的位置和存储模式相关性参见《80x86 指令和六种存储模式》一章)给变量分配需要的空间。当变量不需要使用该空间时,再用动态地址释放函数将该空间释放。空间释放后就成了自由内存,可重新被分配。使用动态内存分配函数不但能使程序员知道是否有自由空间分配、分配的具体位置等情况,而且由于被分配区域未经释放就不会被其它变量侵占,从而保证了该区域的安全。给字符指针动态分配一个指定的区域能使它指向自己的串,避免了 Turbo C 对此不作严格检查而带来的危险。

堆是内存中一块区域,C 可以动态地对它进行存储分配,或者说,在程序执行时堆的大小会改变。堆和栈(或堆栈)是不同的。堆是用于数据分配,程序员可以在设计程序时控制它,而对 Turbo C 来说,栈是由程序自动控制的。数据存在堆中是程序设计的结果,而存在栈中则是函数调用程序的一部分,包括存储实参、局部变量、函数返回地址等(参见《集成开发环境和缺省参数设置》一章)。另外,由于栈的开头被放在栈分段中所允许的最高处,所以当使用 push 或 call 指令时,栈指针 SP 会减小,或者说当栈中压入数据时 SP 指向内存低端位置。而堆则正好和它相反,配置的存储区的地址将逐渐增大(图 25-1)。

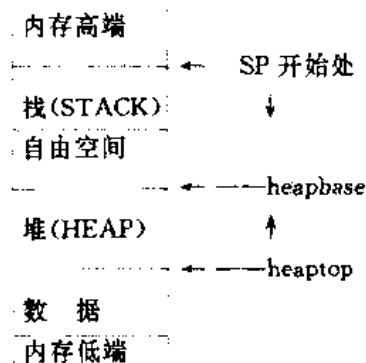


图 25-1 小数据模式中的堆与栈

堆在数据段中。从图 25-1 可以看出,堆和栈相互朝着对方增大,会不会发生相互侵犯而导致程序失败? Turbo C 规定了栈的长度(可用变量 `stklen` 查询,缺省值为 4096 字节,即 4K),当栈超过这个法定的长度时便会发出栈溢出的警告。Turbo C 也不允许堆随便占用内存,例如对小数据模式(Tiny、Small 和 Medium)将其限制在一个特定的范围内(——`heaptop`, ——`heapbase`)。在程序开始执行时可用 `heaplen` 查询将被分配的堆的大小,缺省值是 0,它表示可分配最大的堆。注意它不能用于大数据模式。

除了顶前紧接着的一个 256 字节的边界外,数据段末尾与程序之间的所有空间都在小数据模式中可用。边界用于应用程序扩充堆栈和 MS-DOS 所需的一小量存储。

而对 Small 和 Medium 存储模式,栈和内存高端之间的自由空间也可作为堆。为区别于数据段末尾与栈之间的堆,它被称为【远堆, FAR HEAP】。远堆位于程序默认的数据段外

面。对大型数据模式,栈至内存高端的自由空间都可为堆使用。

Turbo C 还有一些专门从远堆中分配存储区的函数,使用这些函数时一般应注意到:

1. 对微型存储模式 (Tiny) 不能使用这类函数,因为它没有通常由远指针产生的段前缀。

2. 除了在数据段内的 RAM 外,系统中所有可用的 RAM 都能被分配。

3. 被分配的块一般不要大于 64K,但大于 64K 也是可以的。

4. 存取被分配的块只能使用远指针。

5. 函数一般以 unsigned long 为参数。

6. 只适用于 MS-DOS。注意到在包含这类函数的标头文件 alloc.h 中有

```
#if !——STDC——  
void far * —Cdecl farcalloc(unsigned long nunits,unsigned long unitsz);  
.....  
#endif
```

只有使用 ANSI 相容标记 -A(仅使 ANSI 的关键字置 ON)时预定义宏 ——STDC—— 一值才为 1,否则无定义。这就是说,使用这类函数时不应该用相容标记。

动态地址分配中常用的库函数是 malloc() 和 free()。

如果给指针分配内存区域,则该指针应在函数内部,即作为局部变量出现。

注意使用动态地址分配函数时最好要包含标头文件 alloc.h,否则在大数据存储模式下编译程序时可能会遇到麻烦。

25.1 分类

一 分配

- 分配 n 字节块,并指定一个指针指向该分配块 —1 allocmem()
- 分配 n 字节块,函数返回指向该分配块的指针 —2 malloc()
- 从远堆分配 n 字节块,函数返回指向该分配块的远指针 —3 farmalloc()
- 分配 n*m (n 项,每项为 m 字节) 字节块,块清 0, —4 calloc()
函数返回指向该分配块的指针
- 从远堆分配 n*m (n 项,每项为 m 字节) 字节块, —5 farcalloc()
块清 0,函数返回指向该分配块的远指针

二 修改

- 修改先前由 allocmem() 分配的块 —6 setblock()
- 对小数据存储模式调整先分配块 —7 realloc()
- 对大数据存储模式调整已分配块 —8 farrealloc()
- 设置截断值(程序数据段结尾为指针值) —9 brk()
- 把截断值加上 n 字节并改变所分配的数据段存储空间 —10 sbrk()

三 释放

- 释放由前 allocmem() 分配的块 —11 freemem()
- 释放由前 malloc() 或 calloc() 分配的块 —12 free()

· 释放由前 `farmalloc()` 或 `farcalloc()` 分配的块 —13 `farfree()`

四 查询

· 返回未用自由空间的字节数 —14 `coreleft()`
—15 `farcoreleft()`

25.2 库函数

在使用动态存储库函数时,应注意有前缀 `far` 的函数和无 `far` 前缀函数的区别(主要以大、小数据存储模式区分),在分配、修改、释放或查询内存块时应尽量配套使用,而少交叉使用,以免发生意外。此外,特别应注意函数使用的条件要求。

—1 `int` —`Cdecl allocmem(unsigned size, unsigned *segp);`

使用 DOS 功能调用 48H 来分配大小为 `size` 节(1 节 = 16 字节)的自由存储块,指针 `segp` 指向新分配块的起始段地址的地址字。块内所有分配的段是连续的,因而可以通过 `segp` 所指的地址字访问。

如果没有足够的内存空间分配,函数将不对 `segp` 所指的地址字赋值。

DOS 功能 48H 的入口参数是, `AH=0x48`, `BX=size`。返回时, `AX` 指向被分配的内存块的段地址。如果内存不够分配, `CF=1`, `AX` 返回错误代码, `BX` 返回最大可用内存块的节数。

如果调用成功,返回 -1,否则置全局变量 `errno=8(ENOMEM, 无足够的存储空间)` 或 7(`EC ONTR`, 内存控制块损坏)。

只适用于 MS-DOS。

C>DOSMEM1.C

```
#include "dos.h"
```

```
main()
```

```
{
```

```
unsigned size=4,s[2],t[3],*ptr=s;
```

```
int k;
```

```
s[0]=1;s[1]=2;t[0]=3;t[1]=4;t[2]=5;
```

```
k=allocmem(size,ptr); /* 分配块后,k=-1 */
```

```
k=freemem(*ptr); /* 释放块后,k=0 */
```

```
)/ * 调试表达式
```

```
ptr[0],10m;5A 14 02 00 03 00 04 00 05 00
```

```
ptr+1; DS:FFD6
```

```
ptr; DS:FFD4 ptr 本身的地址
```

```
t; {3, 4, 5}
```

```
s,px; {0x145A, 0x2} 分配的段地址的地址字是 0x145A
```

```
s; {5210, 2} 因 ptr 指向 s[0],故 s[0] 内容改变
```

```
k; -1 分配成功
```

```
*/
```

—2 `void *` —`Cdecl malloc (size—t size);`

它企图在堆中分配一个长度为 `size` 字节的存储块,如果分配成功,返回一个指向该存储块的第一个字节的指针。如果没有足够的存储区分配或者 `size=0`,返回 `NULL`(空指针),

块内容不变。适用于 UNIX 系统。size_t 和 NULL 在 alloc.h 中定义为

```
typedef unsigned size_t;
#ifndef NULL
#ifdef (TINY) || defined(SMALL) || defined(MEDIUM)
#define NULL 0
#else
#define NULL 0L
#endif
#endif
```

可见 NULL 和存储模式相关。注意使用空指针常可能发生意外,因此在使用返回值时检查它是否为空指针是很必要的。

—3 void far * —Cdecl farmalloc (unsigned long nbytes);

从远堆 (而不是在数据段) 中分配一块长 nbytes 字节的存储块。返回指向新分配块的指针。如果没有足够的空间分配,返回 NULL。

在小型存储模式 (Small) 和中型存储模式 (Medium) 中,由 farmalloc() 分配的存储块是不能通过 free() 函数来释放的,而只能通过 farfree() 来释放。

—4 void * —Cdecl calloc (size_t nitems, size_t size);

它像 malloc() 一样动态分配存储块,只是块的长度等于 nitems * size 的积,并且将被分配块的每一个存储单元清零 (亦称【块清 0】),即存储单元有值 0。由于它将存储单元清 0,所以也就清除了这些存储单元中的无用数据。这些无用数据往往是由其它程序运行时存放的数据,程序运行结束后一般不会自动将它清除,或者说它成了数据【垃圾】。当当前程序使用这些存储区时如果没有将这些垃圾清除,很可能是不安全的,或者说垃圾会带来麻烦。

如果分配成功,返回指向新分配块的指针。如果没有足够的空间分配给新块,返回 NULL。

C>TYPE DOSMEM2.C

```
#include "alloc.h"
#include "stddef.h"
#include "mem.h"

void * calloc (size_t nitems, size_t size) /* 等效函数 */
{
    unsigned long msize;
    register char * cp;
    msize = (unsigned long)nitems * size; /* 求得块长度 */
    cp = (msize > 0xFFFF) ? NULL : malloc((unsigned)msize);
    if(cp)
        setmem(cp, (unsigned)msize, 0); /* 块清 0 */
    return(cp); /* 如果没有足够内存分配,返回 NULL */
}

main()
{
    unsigned char * ptr;
    int i, j;
    ptr = calloc(2, 3);
    if(ptr != NULL)
    {
```

```

for(i=0;i<2;i++,ptr++) /* 分配的单元很有规律,类似数组那样排列 */
{
    for(j=0;j<3;j++)printf("%x ", *ptr);
    printf("\n");
}
}
/* 输出:000
        000 */

```

—5 void far * —Cdecl farcalloc(unsigned long nunits, unsigned long unitsz);

从远堆中分配有 nunits 个项,每项为 unitsz 字节的一个存储区,即分配的块长度为 nunits * unitsz 字节,并将分配的存储单元都置成 0(或称块清 0,即扫除了已分配块中的垃圾)。返回指向新分配块的指针。如果没有足够的空间,返回 NULL。

—6 int —Cdecl setblock(unsigned segx, unsigned newsize);

修改先前已用 allocmem() 分配的 DOS 存储块的大小为 newsize(单位:节),segx 是该块起始段地址。如果调用成功,返回 -1,否则(CF=1)返回最大可用块的大小(AX 返回出错码),并置 errno 为 7,或 8,或 9。

它相当于 DOS 功能调用 4AH,入口参数是, AH=0x4a, ES=segx, BX=newsize。

只适用于 MS-DOS。

C>TYPE DOSMEM3.C

```

#include "dos.h"
main()
{
    unsigned size=4,s[2],t[3], *ptr=s;
    int k;
    s[0]=1,s[1]=2,t[0]=3,t[1]=4,t[2]=5;
    k=allocmem(size,ptr); /* s,px:{0x145A, 0x2} k: -1 分配 */
    k=freemem(*ptr); /* s,px:{0x145A, 0x2} k: 0 释放 */
    k=setblock(*ptr,10); /* s,px:{0x145A, 0x2} k: 4 未成功 */
    k=allocmem(size+6,s); /* s,px:{0x9935, 0x2} k: -1 重分配 */
    k=setblock(*ptr,10); /* s,px:{0x9935, 0x2} k: -1 成功 */
    k=allocmem(size,s); /* s,px:{0x145A, 0x2} k: -1 再分配 */
    k=setblock(*ptr,10); /* s,px:{0x145A, 0x2} k: 4 未成功 */
}

```

—7 void * —Cdecl realloc(void * block, size_t size);

对小数据存储模式,调整原先分配块即 block 所指块的大小为 size 字节。其一般过程是,先在内存中用 malloc(size) 分配一个新块,尔后将 block 所指块的内容拷入新块。不管怎样,拷入的内容自然不能超过 size,超过部分将丢失。最后将 block 所指块用 free(block); 释放,即原先分配的块已不存在。希望对大数据存储模式用 farrealloc(),以免意外。

如果不能分配,将返回 NULL。适用于 UNIX 系统。

C>TYPE DOSMEM4.C

```

#include "alloc.h"
#define PPP printf("p1=%p p1[0]=%c p1[19]=%c y=%lx\n",p1,p1[0],p1[19],y)
main()

```

```

{
long y;
unsigned char *p1, *p2;
p1=malloc(20);
p1[0]='A';p1[19]='B';
y=coreleft();
PPP;
p2=realloc(p1,30); /* 使用 DOS 重定向与未重定向结果可能会不同 */
y=coreleft();
PPP;
printf("p2=%p p2[0]=%c p2[19]=%c\n",p2,p2[0],p2[19]);
}
/* 在 Small 存储模式下编译后暂时退出集成环境,执行程序输出:
p1=053E p1[0]=A p1[19]=B y=f95c
p1=053E p1[0]=, p1[19]=B y=f934
p2=075E p2[0]=A p2[19]=B
在 Huge 存储模式下编译后暂时退出集成环境,执行程序输出:
p1=6868,0008 p1[0]=A p1[19]=B y=37958
p1=6868,0008 p1[0]= p1[19]=B y=37928
p2=686A,0008 p2[0]=A p2[19]=B */

```

—8 void far * _Cdecl farrealloc(void far * oldblock, unsigned long nbytes);

对大数据存储模式,调整已分配块即 oldblock 所指块为 nbytes 字节。其调整过程与 realloc() 类似,例如,如果调整成功,也会将老块中内容复制到新块中。如果不能分配,返回 NULL。希望对小数据存储模式用 realloc(),以免意外。

C>TYPE DOSMEM5.C

```

#include "alloc.h"
#define PPP y1=coreleft(); y2=farcoreleft();\
printf("p1=%Fp p1[0]=%c p1[19]=%c y1=%lx y2=%lx\n",p1\
,p1[0],p1[19],y1,y2)

main()
{
long y1,y2;
unsigned char far *p1; /* 不允许将这两句写这样的一句,即 */
unsigned char far *p2; /* unsigned char far *p1, *p2 是错误的,但 */
/* 允许写成 unsigned char far *p1, far *p2 */

p1=farmalloc(20);
p1[0]='A';p1[19]='B';
PPP;
p2=farrealloc(p1,30);
PPP;
printf("p2=%Fp p2[0]=%c p2[19]=%c\n",p2,p2[0],p2[19]);
}
/* 在 Small 存储模式下编译后暂时退出集成环境,执行程序输出:
p1=89D7,0008 p1[0]=A p1[19]=B y1=f94a y2=16268
p1=89D7,0008 p1[0]= p1[19]=B y1=f94a y2=16238

```

```
p2=89D9,0008 p2[0]=A p2[19]=B
```

在 Huge 存储模式下编译后暂时退出集成环境,执行程序输出:

```
p1=7B4A,0008 p1[0]=A p1[19]=B y1=24b38 y2=24b38
```

```
p1=7B4A,0008 p1[0]= p1[19]=B y1=24b08 y2=24b08
```

```
p2=7B4C,0008 p2[0]=A p2[19]=B */
```

brk() 和 sbrk() 用于动态地改变分配给调用程序数据段的存储量,这种改变是通过 重置程序的截断值进行的。【截断值, break value】是数据段结尾处以上的第一个位置的地址。分配给数据段的存储空间量随着截断值的增加(或减少)而增加(或减少)。这两个库函数在一般应用程序中不多见,下面的例程也只供参考。

```
—9 int —Cdecl brk(void * addr);
```

它动态地改变数据段所用的内存容量。设置截断值给 addr,即设置程序数据段顶端为指针 addr 所指的存储单元。如果修改导致所分配的存储空间超过允许的范围,这时将不对存储空间作任何改变。

若调用成功返回 0,否则返回 -1(0xFFFF),并置 errno=8(ENOMEM,无足够的存储空间)。

适用于 UNIX 系统。

```
—10 void * —Cdecl sbrk(int incr);
```

把截断值加上 incr 字节,并相应改变数据段存储空间。incr 可以为负,此时所分配的数据段存储空间的地址将减小。如果修改导致所分配的存储空间超过允许的范围,这时将不对存储空间作任何改变。

若调用成功返回老的截断值,否则返回 -1,并置 errno=8(ENOMEM,无足够的存储空间)。适用于 UNIX 系统。

```
C>TYPE DOSMEM6.C
```

```
#include "alloc.h"
```

```
#if defined(—LARGE—) || defined(—HUGE—) || defined(—COMPACT—)
```

```
#define cor printf("coreleft=%lu\n",coreleft())
```

```
#else
```

```
#define cor printf("top=%Fp\t",—heaptop);\
printf("base=%Fp\t",—heapbase);\
printf("len=%d\t",—heaplen);\
printf("brklvl=%Fp\t",—brklvl);\
printf("coreleft=%u\n",coreleft())
```

```
#endif
```

```
#define PHEAP wsp=—SP,wss=—SS,printf("%x,%x\t",wss,wsp);\
printf("p=%Fp\t",p);\
printf("stklen=%d\t",—stklen);cor
```

```
extern char huge * —heapbase; /* 这三个全局变量在大数据模式下无定义 */
```

```
extern char huge * —heaptop; /* 也未在某个标头文件中出现,但可从《 */
```

```
extern char huge * —brklvl; /* 集成开发环境和缺省参数设置》一章的 */
/* —6—2 Linker 中得到一些启发。 */
```

```
extern unsigned —heaplen; /* 在 dos.h 中出现,对大数据模式下无定义 */
```

```
extern unsigned —stklen; /* 在 dos.h 中出现 */
```

```
main()
```

```
{
```



```

int wss, wsp, x, y[100],
int *p=(int *)malloc(100);
PHEAP,
x=brk(p),
printf("x=%d\n",x),
PHEAP,
p=sbrk(0),
PHEAP,
p=sbrk(50),
PHEAP,
p=sbrk(-10),
PHEAP,
}

```

/* 在不同存储模式下（先在集成环境下生成执行文件，然后退出集成环境执行程序）程序输出：

Tiny

```

2753:ffd8 p=1D27:1D44 stklen=4096 top=0000:1CC8 base=1FB0:1CC8 len=0
brklvl=1CC8:1FB0 coreleft=57086

```

x=0

```

2753:ffd8 p=1D27:1D44 stklen=4096 top=0000:1CC8 base=1D44:1CC8 len=0
brklvl=1CC8:1D44 coreleft=57706

```

```

2753:ffd8 p=1D27:1D44 stklen=4096 top=0000:1CC8 base=1D44:1CC8 len=0
brklvl=1CC8:1D44 coreleft=57706

```

```

2753:ffd8 p=1D27:1D44 stklen=4096 top=0000:1CC8 base=1D76:1CC8 len=0
brklvl=1CC8:1D76 coreleft=57656

```

```

2753:ffd8 p=1D27:1D76 stklen=4096 top=0000:1CC8 base=1D6C:1CC8 len=0
brklvl=1CC8:1D6C coreleft=57666

```

Small

```

28b4:ffd8 p=064F:066C stklen=4096 top=0000:05F0 base=08D8:05F0 len=0
brklvl=05F0:08D8 coreleft=62934

```

x=0

```

28b4:ffd8 p=064F:066C stklen=4096 top=0000:05F0 base=066C:05F0 len=0
brklvl=05F0:066C coreleft=63554

```

```

28b4:ffd8 p=064F:066C stklen=4096 top=0000:05F0 base=066C:05F0 len=0
brklvl=05F0:066C coreleft=63554

```

```

28b4:ffd8 p=064F:066C stklen=4096 top=0000:05F0 base=069E:05F0 len=0
brklvl=05F0:069E coreleft=63504

```

```

28b4:ffd8 p=064F:069E stklen=4096 top=0000:05F0 base=0694:05F0 len=0
brklvl=05F0:0694 coreleft=63514

```

Medium

```

28c3:ffd6 p=069D:06BA stklen=4096 top=0000:063E base=0926:063E len=0
brklvl=063E:0926 coreleft=62852

```

x=0

```

28c3:ffd6 p=069D:06BA stklen=4096 top=0000:063E base=06BA:063E len=0
brklvl=063E:06BA coreleft=63472

```

```

28c3:ffd6 p=069D:06BA stklen=4096 top=0000:063E base=06BA:063E len=0
brklvl=063E:06BA coreleft=63472

```

```

28c3,fid6 p=069D,06BA stklen=4096 top=0000,063E base=06EC,063E len=0
brklvl=063E,06EC coreleft=63422
28c3,fid6 p=069D,06EC stklen=4096 top=0000,063E base=06E2,063E len=0
brklvl=063E,06E2 coreleft=63432

```

Compact

```

2963,fde p=2A67,0008 stklen=4096 coreleft=481032
x=0
2963,fde p=2A67,0008 stklen=4096 coreleft=481664
2963,fde p=2A67,0008 stklen=4096 coreleft=481664
2963,fde p=2A67,0008 stklen=4096 coreleft=481614
2963,fde p=2A6A,000A stklen=4096 coreleft=481624

```

Large

```

2975,fdc p=2A79,0008 stklen=4096 coreleftZ
x=0
2975,fdc p=2A79,0008 stklen=4096 coreleft=481376
2975,fdc p=2A79,0008 stklen=4096 coreleft=481376
2975,fdc p=2A79,0008 stklen=4096 coreleft=481326
2975,fdc p=2A7C,000A stklen=4096 coreleft=481336

```

Huge

```

299f,fda p=2AA3,0008 stklen=4096 coreleft=480072
x=0
299f,fda p=2AA3,0008 stklen=4096 coreleft=480704
299f,fda p=2AA3,0008 stklen=4096 coreleft=480704
299f,fda p=2AA3,0008 stklen=4096 coreleft=480654
299f,fda p=2AA6,000A stklen=4096 coreleft=480664 */

```

—11 int —Cdecl freemem(unsigned segx);

释放以前用 allocmem() 所分配的存储块, segx 是该块起始段地址。如果调用成功, 它返回 0, 否则返回 -1。特别注意释放内存时应保证参数 segx 的正确性, 否则会产生不可预料的结果。

它相当于 DOS 功能调用 49H 的部分功能, 入口参数是, AH=0x49, ES=segx, 但本函数对 BX(新要求的块大小) 未要求赋值。

出错时 (CF=1) 置全局变量 errno=8(ENOMEM, 无足够的存储空间) 或 7(ECONTR, 内存控制块损坏), 或 9(EINVMEM, 非法的内存块地址)。

DOS 3.30 在释放内存时并不合并相邻的自由内存空间, 只有当分配块或重新分配时才这样做。

只适用于 MS-DOS。

—12 void —Cdecl free(void *block);

释放先前用 malloc() 或 calloc() 分配的存储块, block 必须包含所分配块的首地址。如果 block 为 NULL, 则也是允许的, 不过不释放任何存储块 (ANSI 有时需要它)。当释放多个存储块时, 尽量考虑先分配后释放。

注意参数 block 应是正确的值, 例如它不应是已经释放了的内存指针等, 否则将产生不可预料的结果。为可靠起见, 不让将要释放的内存中的数据在内存释放后产生坏作用 (因释放后内存中的这部分数据一时不会改变, 将成为“垃圾”), 可以在释放之前先将待释放的内存清 0。

C>TYPE DOSMEM7.C

```
#include "alloc.h"
#define PP n=coreleft(),\
        y=sbrk(0);\
        printf("n=%x y=%p\n",n,y)
        /* n 为自由空间,y 为截断值 */

main()
{
    int *p,*q,*y,n;    /* 也可以将 y 定义为 long y;或unsigned long y; */
    PP;                /* 因为 malloc() 的类型为 void */
    p=malloc(10);
    q=malloc(20);
    PP;
    free(q);
    PP;
    free(p);
    PP;
}
```

/* 输出:Small 存储模式下:

```
n=f9a2 y=050C
n=f97a y=0534
n=f992 y=051C
n=f9a2 y=050C
```

Huge 存储模式下:

```
n=4df8 y=7B20:0000
n=4df8 y=7B24:0000
n=4df8 y=7B22:0000
n=4df8 y=7B20:0000
```

如将 free(q) 和 free(p) 两句排列顺序颠倒,则有

Small 存储模式下:

```
n=f9a2 y=050C
n=f97a y=0534
n=f97a y=0534
n=f9a2 y=050C
```

Huge 存储模式下:

```
n=4df8 y=7B20:0000
n=4df8 y=7B24:0000
n=4df8 y=7B24:0000
n=4df8 y=7B20:0000
```

*/

—13 void —Cdecl farfree(void far * block);

释放远堆中先前用 farcalloc() 或 farmalloc() 分配的块。注意 block 应是一个有效的指针,否则将会出现不可预料的结果。这就是说,free() 和 farfree() 是不能混用的,或者说不能用 free() 释放远指针,也不能用 farfree() 释放一个近指针。

C>TYPE DOSMEM8.C

```
#include "alloc.h"
main()
{
    unsigned long y;
    unsigned long *p1,*p2;
    p1=farmalloc(20);    /* 在 Small 存储模式下编译,对此句出现警告 */
                        /* Warning :Suspicious pointer conversion */
    p1[0]='A';p1[1]='B';    /* 注意这种赋值方法与 printf 的打印不一致问 */
    y=farcoreleft();        /* 题,参见 STDIO.H 中 printf() 的原型 */
    printf("p1=%Fp p1[0]=%c p1[1]=%c p1[2]=%c ", p1,p1[0],p1[1],p1[2]);
    printf("y=%lu\n",y);    /* 不能将本句合并到上一句中,否则会出错 */
}
```

```

p2=farrealloc(p1,30);    /* 在 Small 存储模式下编译,对此句出现警告 */
y=farcoreleft();
printf("p2=%Fp p2[0]=%c p2[1]=%c p2[2]=%c ", p2,p2[0],p2[1],p2[2]);
printf("y=%lu\n",y);
}

```

/* 通过调试表达式 p1[0],10mc,p1[0],10mx,p2[0],10mc,p2[0],10mx 可以观察到具体的变化情况。注意上面的打印方式。

在 Huge 存储模式下输出:

```

p1=7B56,0008 p1[0]=A p1[1]= p1[2]=B y=150136
p2=7B58,0008 p1[0]=A p2[1]= p2[2]=B y=150088      */

```

```

-14 #if defined(——COMPACT——)||defined(——LARGE——)||defined(——HUGE)
    unsigned long —Cdecl coreleft(void);
    #else
    unsigned —Cdecl coreleft(void);
    #endif

```

对小型数据存储模式,它返回堆与栈之间自由空间与 256 字节的差值,256 个字节为边界占用值;对大型数据存储模式,它返回堆与栈之间的未用自由空间的字节数(无符号的长整型值)。

```

-15 unsigned long —Cdecl farcoreleft(void);

```

返回远堆中最高已分配存储区与内存最高端之间未使用的自由空间存储量(字节数)。

第二十六章 数学函数

26.1 常数和宏说明

在 `math.h` 中用宏定义了一些常数,它们有 21 位十进制数。用户在自己的程序中引用这些常数时最好用宏代替它们,而不要直接书写具体的数值。

```
#define M-E          2.71828182845904523536    /* 欧拉常数 e */
#define M-LOG2E      1.44269504088896340736    /* 1/ln2 */
#define M-LOG10E     0.434294481903251827651    /* 1/ln10 */
#define M-LN2        0.693147180559945309417    /* ln2 */
#define M-LN10       2.30258509299404568402    /* ln10 */
#define M-PI         3.14159265358979323846    /* 圆周率 pi */
#define M-PI-2       1.57079632679489661923    /* pi/2 */
#define M-PI-4       0.785398163397448309116    /* pi/4 */
#define M-1-PI       0.318309886183790671538    /* 1/pi */
#define M-2-PI       0.636619772367581343076    /* 2/pi */
#define M-1-SQRTPI   0.564189583547756286948    /* 1/sqrt(pi) */
#define M-2-SQRTPI   1.12837916709551257390    /* 2/sqrt(pi) */
#define M-SQRT2      1.41421356237309504880    /* sqrt(2) */
#define M-SQRT-2     0.707106781186547524401    /* sqrt(2)/2 */
```

此外还定义了几个与数值相关的宏:

```
#define _MATH_H 1 /* TC 缺省情况是没有定义 _MATH_H, 包含 math.h 便定义它 */
#define EDOM 33 /* 见 matherr() */
#define ERANGE 34 /* 见 matherr() */
#define HUGE_VAL _huge_dble /* _huge_dble 是外部变量,常数极值 */
```

用户可自定义另一些重要的常数:

```
#define M-SQRT3      1.73205080756887729352    /* sqrt(3) */
#define M-SQRT5      2.23606797749978969640    /* sqrt(5) */
#define M-2-3        1.25992104989487316476    /* 2 开 3 次方 */
#define M-2-4        1.18920711500272106671    /* 2 开 4 次方 */
#define M-3-3        1.44224957030740838232    /* 3 开 3 次方 */
#define M-PI-180     0.01745329251994329576    /* pi/180 */
#define M-PI2        9.86960440108935861883    /* pi 的平方 */
#define M-SQRTpi     1.77245385090551602729    /* sqrt(pi) */
#define M-E2         7.38905609893065022723    /* e 的平方 */
#define M-LN3        1.09861228866810969139    /* ln3 */
#define M-LNpi       1.14472988584940017414    /* ln(pi) */
#define M-SIN1       0.84147098480789650665    /* sin(1) */
```



```

#define —EXPBASE      2          /* 幂的底数      */
#define —IEEE          1
#define —LENBASE      1          /* 幂用基数      */
#define HIDDENBIT      1          /* 隐含位        */
#define LN—MAXDOUBLE   7.0978E+2 /* double 的最大自然对数 */
#define LN—MINDOUBLE  -7.0840E+2 /* double 的最小自然对数 */

```

在 float.h 中也定义了一些常数,它们主要用于处理浮点数。(参见《80x87 数学协处理器》一章)。

26.2 函数或宏分类

一 绝对值函数

- | | |
|--------|-------------------------------------|
| · 整数 | —1 abs() 函数
abs() 宏
——abs()—— |
| · 长整型数 | —2 labs() |
| · 浮点数 | —3 fabs() |
| · 复数 | —4 cabs() |

二 算术运算

- | | |
|----------|-----------|
| · 两整数相除 | —5 div() |
| · 两长整数相除 | —6 ldiv() |

三 三角函数

- | | |
|------------|-------------|
| · 正弦 | —7 sin() |
| · 余弦 | —8 cos() |
| · 正切 | —9 tan() |
| · 反正弦 | —10 asin() |
| · 反余弦 | —11 acos() |
| · x 的反正切 | —12 atan() |
| · Y/X 的反正切 | —13 atan2() |

四 双曲函数

- | | |
|--------|------------|
| · 双曲正弦 | —14 sinh() |
| · 双曲余弦 | —15 cosh() |
| · 双曲正切 | —16 tanh() |

五 对数函数

- | | |
|--------|-------------|
| · 自然对数 | —17 log() |
| · 常用对数 | —18 log10() |

六 指数函数

- . 底为 e, 指数 x —19 exp()
- . 底为 x, 指数 y —20 pow()
- . 底为 10, 指数为 y —21 pow10()
- . value 分解成 $x * 2^n$ —22 frexp()
- . 计算 $value * 2^n$ —23 ldexp()

七 平方根函数

- . 平方根 —24 sqrt()
- . 直角三角形斜边长 —25 hypot()

八 舍入函数

- . 上舍入 —26 ceil()
- . 下舍入 —27 floor()
- . 四舍五入 —28 round()

九 求余数函数

- . 求 x/y 的余数 (模) —29 fmod()
- . 将 value 分成整数与小数 —30 modf()

十 n 次多项式值

- . 系数为 c, 变元 x —31 poly()

十一 极值

- . 极大值 —32 max()
- . 极小值 —33 min()

十二 符号函数

- . $V = \text{sgn}(x)$ —34 sgn()

十三 数值与字符串转换

(一) 串转换成数值

- . 串转换成浮点数, 但不记录不能转换字符位置 —35 atof()
- . 串转换为双精度值, 记录不能转换字符位置 —36 strtod()
- . 串转换成整型数 —37 atoi()
- . 串转换成长整型值, 但不记录不能转换字符位置 —38 atol()
- . 串转换为长整型值, 记录不能转换字符位置 —39 strtol()
- . 串转换为无符号长整数 —40 strtoul()

(二) 数值转换成串

- . e 格式浮点数转换成串 —41 ecvt()
- . f 格式浮点数转换成串 —42 fcvt()

- 优先生成 f 格式串, 否则生成 e 格式串 —43 gcvt()
- 整数转换成串 —44 itoa()
- 长整数转换成串 —45 ltoa()
- 无符号长整数转换成串 —46 ultoa()

十四 数循环移位 (参见《位运算与位域》一章)

- 整数左移 rotr()
- 长整数左移 lrotr()
- 整数右移 rotr()
- 长整数右移 lrotr()

十五 数字错误处理函数

- 非用户定义 —47 matherr
- 用户可定义 —48 matherr

十六 随机函数

- 产生随机序列的第一个数 (又称“初始化随机数发生器”) —49 srand()
- 由系统时间自动初始化随机数发生器 —50 randize()
- 产生一个 0~32767 的随机数 (又称“随机数发生器”) —51 rand()
- 产生一个 0~n-1 的随机数, n 为用户事先指定 —52 random()

26.3 详细说明

```
—1 int —Cdecl abs(int x);          /* 函数原型 */
   #define abs(x) —abs—(x)         /* 宏 */
   int —Cdecl —abs—(int x);        /* 内部函数 */
```

它返回数 x 的绝对值。只要使用了原型, x 不是整型时将转为整型后再求其绝对值。

注意: x 应在 -32767 到 32767 之间, 此时返回 $0 \sim 32767$; 对 -32768 是例外, 仍返回原值; 对其余超出范围的, 可能非理想值。

在 `math.h` 和 `stdlib.h` 中都有 `abs` 函数的原型

```
int —Cdecl abs(int x);
```

但是, 在 `stdlib.h` 中同时还把 `abs` 定义为宏 (在集成环境下未用 `O/C/S/ANSI keywords` 或对命令行编译器未用 `-A` 相容标记时):

```
int —Cdecl —abs—(int x);
#define abs(x) —abs—(x)
```

这就是说, 如果你将 `stdlib.h` 包含在源文件中, 而又没有用 `#undef abs` 取消宏定义 `abs`, 则定义的宏有效, 即 `abs` 按宏处理; 否则按函数处理。

`—abs—()` 是一个 TC 内部 (in line) 函数, 只要包含了 `stdlib.h`, 就可直接调用它。

适用于 UNIX 系统。

C>TYPE MATH1.C

```

#include "stdlib.h"
#undef abs
main()
{
int x=-3,y=0,z=3;
float a=-3.6,b=0.6,c=3.6;
int x1=abs(x),y1=abs(y),z1=abs(z);
int a1=abs(a),b1=abs(b),c1=abs(c);
float a2=abs(a),b2=abs(b),c2=abs(c);
double d=-0.0000012;
int e1=-32767,e2=-32768;
int f=-32769;
int g1=32767,g2=32768;
int h=-327690;
printf("%d %d %d\n",x1,y1,z1);      /* 3 0 3      */
printf("%d %d %d\n",a1,b1,c1);      /* 3 0 3      */
printf("%.2f %.2f %.2f\n",a2,b2,c2); /* 3.00 0.00 3.00 */
printf("%d\n",abs(d));               /* 0          */
printf("%d\n",abs(e1));              /* 32767      */
printf("%d %d\n",e2,abs(e2));        /* -32768 -32768 */
printf("%d %d\n",f,abs(f));          /* 32767 32767  */
printf("%d %d\n",abs(g1),abs(g2));    /* 32767 -32768 */
printf("%d %d\n",h,abs(h));          /* -10 10     */
}

```

—2 long —Cdecl labs(long x);

给出长整型数绝对值。它定义为：

```

#include <stdlib.h>
long labs(long x)
{return (x<0? -x:x);}

```

适用于 UNIX 系统。

C>TYPE MATH2.C

```

#include "math.h"
#define PL "%ld\n"
#define PX "%lx\n"
#define PO "%lo\n"
main(){
long x=labs(-0.5);
/* long y=labs(-HUGE-VAL); Floating point error; Domain. */
int z=labs(-0x5f);
int w=labs(-017);
long u=labs(-8);
long v=labs(-017);
long t=labs(-0x5f);
printf(PL,x);      /* 0          */
printf(PL,z);      /* 79364191   */
}

```

```

printf(PL,w);      /* 79364111 */
printf(PL,(long)z); /* 95      */
printf(PL,(long)w); /* 15      */
printf(PX,(long)z); /* 5f      */
printf(PO,(long)w); /* 17      */
printf(PL,u);      /* 8       */
printf(PL,v);      /* 15      */
printf(PL,t);      /* 95      */
}

```

—3 double —Cdecl fabs (double x);

求双精度数 x 的绝对值,结果是双精度值。注意 printf 函数对其结果的处理。适用于 UNIX 系统。

C>TYPE MATH3.C

```

#include "math.h"
main(){
double x=fabs(-0.5);
double y=fabs(-HUGE-VAL);
int z=fabs(-0x5f);
int w=fabs(-017);
double u=fabs(-8);
double v=fabs(-017);
double t=fabs(-0x5f);
printf("%f\n",x); /* 0.500000 */
printf("%f\n",y); /* 1.797693134862315710000000000000000000000000e+308 */
printf("%d\n",z); /* 95 */
printf("%f\n",z); /* 0.000000 */
printf("%f\n",w); /* 0.000000 */
printf("%f\n",u); /* 8.000000 */
printf("%f\n",v); /* 15.000000 */
printf("%f\n",t); /* 95.000000 */
printf("%f\n",(double)z); /* 95.000000 */
printf("%f\n",(double)w); /* 15.000000 */
}

```

—4 #define cabs(z) (hypot ((z).x, (z).y))

宏,求复数 $x+iy$ 的模 (或绝对值),它等于 $\sqrt{x^2+y^2}$ 。 z 的定义是 struct complex z;

结构 complex 在 math.h 中定义为

```

struct complex
{double x, y;};

```

返回参见 hypot() 函数。适用于 UNIX 系统。

C>TYPE MATH4.C

```

#include "math.h"
main(){

```

```

struct complex z;
double c;
int t=-MATH-H;
z.x=sin(-4.1);
z.y=3.5;
c=cabs(z);
printf("%f\n",c); /* 3.594381 */
printf("%f\n",hypot(z.x,z.y)); /* 3.594381 */
printf("-MATH-H=%d\n",t); /* -MATH-H=1 顺便列出该常量 */
}

```

—5 div_t —Cdecl div (int numer, int denom);

div 将整数 numer(被除数) 除于整数 denom(除数), 返回的商和余数是 div_t 类型结构的成员值。数据类型 div_t 在 stdlib.h 中定义为

```

typedef struct {
    int quot; /* 除得的商 (quotient) */
    int rem; /* 除得的余数 (remainder) */
}div_t;

```

C>TYPE MATH5.C

```

#include "stdlib.h"
main() {
    div_t x;
    x=div(10,3);
    printf("10 除于3 的商是%d,余数为%d\n",x.quot,x.rem);
}/* 程序输出:10 除于3 的商是3,余数为1 */

—6 ldiv_t —Cdecl ldiv (long numer, long denom);

```

ldiv 将长整数 numer(被除数) 除于另一个长整数 denom(除数), 返回的商和余数是 ldiv_t 类型结构的成员值。数据类型 ldiv_t 在 stdlib.h 中定义为

```

typedef struct {
    long quot; /* 商 (长整型) */
    long rem; /* 余数 (长整型) */
}ldiv_t;

```

C>TYPE MATH6.C

```

#include "stdlib.h"
main() {
    ldiv_t x;
    x=ldiv(100000L,30000L);
    printf("100000 除于30000 的商是%d,余数为%d\n",x.quot,x.rem);
}/* 程序输出:100000 除于30000 的商是3,余数为10000 */

```

—7 double —Cdecl sin (double x);

返回 x 正弦,返回值域 $[-1,1]$ 。角度 x 的单位用弧度。适用于 UNIX 系统。

—8 double —Cdecl cos (double x);

返回 x 余弦,返回值域 $[-1,1]$ 。角度 x 的单位用弧度。函数错误处理程序可以通过 matherr() 修改。适用于 UNIX 系统。

```
#include "math.h"
```

```
double x=M-PI,y=M-PI-2,z=M-PI-4*7;
```

```
printf("x=%f x1=%f\n",x,x1); /* x=3.141593 x1=-1.000000 */
```

```
printf("y=%f y1=%f\n",y,y1); /* y=1.570796 y1=0.000000 */
```

```
printf("z=%f z1=%f\n",z,z1); /* z=5.497787 z1=0.707107 */
```

3

```
—9 double —Cdecl tan (double x);
```

```
C>TYPE MATH8.C
```

```
#include "math.h"
```

```
main(){
```

```
double u=1e-32,w=M-PI-2/2,x=M-PI-2-0.01,y=M-PI-2;
```

```
double u1=tan(u),w1=tan(w),x1=tan(x),y1=tan(y);
```

```
printf("u= %.33f u1= %.33f\n",u,u1);
```

```
printf("w= %f w1= %.10f\n",w,w1);
```

```
printf("x=%f x1=%f\n",x,x1);
```

```
printf("y= %f y1= %f\n",y,y1);
```

}

```
/* 程序输出:
```

[illegible]

$y_1 \approx 0.0000000000000000000000000000010$

w=0.785398 w1=1.0000000000

x=1.560796 x1=99.9966666444

y=1.570796 y1=16324552277619072.000000

* /

```
—10 double —Cdecl asin(double x);
```

获得 x 的反正弦值（弧度）。 x 的定义域为 $[-1, 1]$ ，值域为 $[-M-\pi/2, M-\pi/2]$ 。

```
—11 double —Cdecl acos(double x);
```

获得 x 的反余弦值 (弧度)。 x 的定义域为 $[-1, 1]$, 即 $-1 \leq x \leq 1$ 。超出此范围时,

它将调用函数 `matherr`，印出定义域错误 (DOMAIN error)。

```
/* #include "stdio.h" */    /* 此句可不要 */
```

```
#include "math.h"
```

```
main() {
```

```
double x1=.001,x2=1.001;
```

```
double y1=acos(x1),y2;    /* 程序输出:          */
```

```
y2=acos(x2);          /* acos| DOMAIN error          */
```

```
perior("MATH ERROR="); /* MATH ERROR=: Math argument */
```

```
printf("%f\n", y2);          /* +NAN */
```

```
perror("PRINT ERROR="); /* PRINT ERROR=: Math argument */
```

```
printf("%f\n",y1);      /* 1.569796      */
}
```

—12 double —Cdecl atan (double x);

返回 x 的反正切值 (弧度), 其值在 $-M-\pi/2$ 到 $M-\pi/2$ 之间, 这里 $M-\pi$ 是圆周率。

—13 double —Cdecl atan2 (double y, double x);

返回 y/x 的反正切值 (弧度), 即使角度接近 $M-\pi$ 或 $-M-\pi$ (即 x 接近于 0), 也能产生正确结果。返回值在 $-M-\pi/2$ 到 $M-\pi/2$ 之间。

C>TYPE MATH10.C

```
#include "math.h"
```

```
main(){
```

```
double x=1e-99,y=1,z;
```

```
z=atan(y/x);
```

```
printf("%f\n",y/x); /* 9.9999999999999996700000000000000000000000000e+98 */
```

```
printf("%f\n",z); /* 1.570796 */
```

```
}
```

—14 double —Cdecl sinh (double x);

计算双曲正弦。 $\sinh(x) = (\exp(x) - \exp(-x))/2$ 。当 $m \times m \geq 710.475$ 时结果溢出, 返回带适当符号的 HUGE-VAL 值。当 x 太小时便返回 x 值。错误处理程序可通过 `matherr()` 来修改。适用于 UNIX 系统。

C>TYPE MATH11.C

```
#include "math.h"
```

```
main(){
```

```
double u=1e-19,w=8.88,x=-1,y=7.1e2,z=7.11e+2;
```

```
double u1=sinh(u),w1=sinh(w),x1=sinh(x),y1=sinh(y),z1=sinh(z);
```

```
printf("u= %.20f u1= %.20f\n",u,u1);
```

```
printf("w= %.2f w1= %.10f\n",w,w1);
```

```
printf("x= %.2f x1= %.10f\n",x,x1);
```

```
printf("y= %.2f y1= %f\n",y,y1);
```

```
printf("z= %.2f z1= %f\n",z,z1);
```

```
}
```

/* 程序输出:

sinh: OVERFLOW error

u=0.00000000000000000001 u1=0.00000000000000000001

w=8.88 w1=3593.3952983284

x=-1.00 x1=-1.1752011936

y=710.00 y1=1.11699738308085546000000000000000000000000e+308

z=711.00 z1=1.79769313486231571000000000000000000000000e+308

*/

—15 double —Cdecl cosh (double x);

计算双曲余弦。 $\cosh(x) = (\exp(x) + \exp(-x))/2$ 。当 $m \times m \geq 710.475$ 时结果溢出, 返回带适当符号的 HUGE-VAL 值。适用于 UNIX 系统。

C>TYPE MATH12.C

```
#include "math.h"
main(){
double w=7.1e+2,x=7.11e+2,y=-1,z=0;
double w1=cosh(w),x1=cosh(x),y1=cosh(y),z1=cosh(z);
printf("w= %.1f w1= %.6f\n",w,w1);
printf("x= %.1f x1= %.6f\n",x,x1);
printf("y= %.1f y1= %.6f\n",y,y1);
printf("z= %.1f z1= %.6f\n",z,z1);
}
```

程序输出:

```
cosh: OVERFLOW error
w=710.0 w1=1.116997383080855460000000000000000000000000e+308 /* 不算溢出 */
x=711.0 x1=1.797693134862315710000000000000000000000000e+308 /* 溢出 */
y=-1.0 y1=1.543081
z=0.0 z1=1.000000
```

—16 double —Cdecl tanh (double x);

计算双曲正切。 $\tanh(x) = (\exp(x) - \exp(-x)) / (\exp(x) + \exp(-x))$

注意:1. 当丢失有效位时,使用本公式可能接近 0 ;

2. $\tanh(-x) = -\tanh(x)$;
3. 如果 $0 \leq x < 2^{-33}$, 则函数返回 x ;
4. 如果 $2^{-33} \leq x < 0.17325$ 时将用
 $y = \exp(2x) - 1$
 $\tanh(x) = y / (2 + y)$
5. 当 $32 > x > 0.17325$ 时用
 $y = \exp(x)$
 $\tanh(x) = (y - 1/y) / (y + 1/y)$
6. 当 $x \geq 32$ 时函数返回 1 .

适用于 UNIX 系统。

C>TYPE MATH13.C

```
#include "math.h"
main(){
double u=1e-32,w=0.17324,x=.17325,y=32,z=HUGE-VAL;
double u1=tanh(u),w1=tanh(w),x1=tanh(x),y1=tanh(y),z1=tanh(z);
printf("u= %.33f u1= %.33f\n",u,u1);
printf("w= %.10f w1= %.10f\n",w,w1);
printf("x= %.10f x1= %.10f\n",x,x1);
printf("y= %.10f y1= %.10f\n",y,y1);
printf("z= %.10f z1= %.10f\n",z,z1);
}
/* 程序输出:
u=0.0000000000000000000000000000000000000000000
u1=0.000000000000000000000000000000000000000000
w=0.173240 w1=0.1715274573
x=0.173250 x1=0.1715371630
y=32.000000 y1=1.000000
```

```
z=1.79769313486231571000000000000000000000e+308 z1=1.000000
*/
```

—17 double —Cdecl log (double x);

计算自然对数函数 $\ln(x)$ 的值, 它的底为 e , 是欧拉常数 ($e=M-E$)。

当 x 小于或等于 0 时出错, 此时置 `errno` 为 33(EDOM), 函数返回 HUGE—VAL。出错时自动调用 —matherr 函数处理。适用于 UNIX 系统。

C>TYPE MATH14.C

```
#include "math.h"
#define PR printf("%f\n", y);
main(){
double x=4.2, y;
y=log(x);PR; /* 1.435085 */
y=log(0x45);PR; /* 4.234107 */
y=log(0236);PR; /* 5.062595 */
y=log(-HUGE-VAL); /* log, DOMAIN error */
PR; /* -NAN */
}
```

—18 double —Cdecl log10 (double x);

计算常用对数函数 $\lg(x)$ 的值, 它的底是 10。当 x 小于或等于 0 时出错, 置 `errno` 为 33(EDOM), 函数返回负 HUGE—VAL, 并自动调用 —matherr 函数处理。适用于 UNIX 系统。

C>TYPE MATH15.C

```
#include "math.h"
#define PR printf("%f\n", y);
main(){
double x=4.2, y;
y=log10(x);PR; /* 0.6232492904 */
y=log10(0x45);PR; /* 1.8388490907 */
y=log10(0236);PR; /* 2.1986570870 */
y=log10(HUGE-VAL);
PR; /* 308.2547155599 */
}
```

—19 double —Cdecl exp (double x);

它是以 e 为底、指数为 x 的指数函数。其中 e 为欧拉常数 ($e=M-E$)。

当正确的值溢出时, 它返回 HUGE—VAL。若最后值很大, 它将置 `errno` 为 34(ERANGE)。它适用于 UNIX 系统。

C>TYPE MATH16.C

```
#include "math.h"
main(){
double w=709.78, x=7.098e+2, y=-1, z=0;
double w1=exp(w), x1=exp(x), y1=exp(y), z1=exp(z);
printf("w=%f w1=%f\n", w, w1);
printf("x=%f x1=%f\n", x, x1);
}
```



```
printf("y= %.1f y1= %f\n",y,y1);
printf("z= %.1f z1= %f\n",z,z1);
}
```

程序输出结果:

```
exp: OVERFLOW error
w=709.78 w1=1.792822794394515540000000000000000000000000e+308 /* 未溢出 */
x=709.80 x1=1.797693134862315710000000000000000000000000e+308 /* 溢出 */
y=-1.0 y1=0.367879
z=0.0 z1=1.000000
```

—20 double —Cdecl pow (double x, double y);

计算指数函数 x^y (底为 x , 指数为 y) 的值。当正确值溢出时, 它返回 HUGE-VAL。如果值太大, 置 `errno` 为 34(ERANGE)。当 x 小于或等于 0 而 y 不是整数, 或者 x 和 y 均为 0 时, 它返回负 HUGE-VAL, 并置 `errno` 为 33(EDOM)。它适用于 UNIX 系统。

C>TYPE MATH17.C

```
#include "math.h"
#define POWXY z=pow(x,y)
#define PR printf("x= %.1f y= %.1f z= %.3f\n",x,y,z)
#define POW POWXY;PR
main(){
double x=2,y=3,z;
POW; /* x=2.0 y=3.0 z=8.000 */
x=0;POW; /* x=0.0 y=3.0 z=0.000 */
x=-2.1;POW; /* x=-2.1 y=3.0 z=-9.261 */
x=2,y=0;POW; /* x=2.0 y=0.0 z=1.000 */
y=-2.1;POW; /* x=2.0 y=-2.1 z=0.233 */
x=HUGE-VAL,y=-1;POW;
/* x=1.797693134862315710000000000000000000000000e+308 y=-1.0 z=0.000 */
y=0;POW;
/* x=1.797693134862315710000000000000000000000000e+308 y=0.0 z=1.000 */
y=1;POW; /* exp: OVERFLOW error */
/* x=1.797693134862315710000000000000000000000000e+308 y=1.0
z=1.797693134862315710000000000000000000000000e+308 */
x=-1.2,y=HUGE-VAL;POW; /* log: DOMAIN error */
/* Floating point error: Domain */
}
```

—21 double —Cdecl pow10 (int p);

计算指数函数 10^y (底为 10, 指数为 y) 的值。其余同 `pow()`。

C>TYPE MATH18.C

```
#include "math.h"
#define PO po=pow10
#define PR printf(" %.10f\n",po)
main(){ /* 注意,函数参数规定为整型,下取双精度,由函数原型检查转换 */
double p1=46481,p2=0,p3=2,p4=2.5,p5=32767,p6=1000;
double po;
```

```

PO(p1),PR; /* 0.0000000000          参数值太大时 */
PO(p2),PR; /* 1.0000000000          */
PO(p3),PR; /* 100.0000000000        */
PO(p4),PR; /* 100.0000000000        */
PO(p5),PR; /* +INF                  */
PO(p6),PR; /* Floating point error: Overflow 无法打印 */
}

```

—22 double —Cdecl frexp (double x, int *exponent);

它将一个双精度数 x 分解成尾数 f 和指数 n, x、f 和 n 满足等式

$$x = f * 2^n$$

即 x 等于 f 与 2 的 n 次方的积。这里 $f < 1$, 它是函数返回的值; 指针 exponent 所指变量存储整数 n 的值。frexp 是 fraction(小数)和 exponent(指数)的助记符。适用于 UNIX 系统。

```

C>TYPE MATH19.C
#include "math.h"
main(){
double value=-4.2;
double x;
int y,z,*exponent=&z;
printf("value=%f\n",value);          /* value=-4.200000 */
x=frexp(value,&y);
printf("x=%f\n",x);                  /* x=-0.525000    */
printf("y=%d\n",y);                  /* y=3            */
printf("%f\n",x*pow(2,y));           /* -4.200000      */
x=frexp(value,exponent);
printf("x=%f\n",x);                  /* x=-0.525000    */
printf("z=%d\n",*exponent);          /* z=3            */
printf("%f\n",x*pow(2,z));           /* -4.200000      */
x=frexp(HUGE_VAL,exponent);
printf("x=%f\n",x);                  /* x=1.000000     */
printf("z=%d\n",z);                  /* z=1024         */
printf("%f\n",x*pow(2,z));           /* exp: OVERFLOW error */
/* 1.7976931348623155100000000000000000000000000e+308 */
}

```

—23 double —Cdecl ldexp (double x, int exponent);

计算 $x * \text{pow}(2, \text{exponent})$ 的值, 即结果为 $x * 2^{\text{exponent}}$ 。

出错时自动调用 —matherr 函数处理。ldexp 可理解为 limited(有限的)exponent(幂)的助记符。适用于 UNIX 系统。

```

C>TYPE MATH20.C
#include "math.h"
main(){ double value,x=-4.2;
int exponent=2;
value=ldexp(x,exponent);
printf("value=%f\n",value);          /* value=-16.800000 */
}

```


C>TYPE MATH23.C

```
#include "math.h"
main()
{
double x=4.4,y=4.6,z=4.6;
double x1=ceil(x),y1=ceil(y),z1=ceil(z);
printf("%.1f %.1f %.1f\n",x1,y1,z1); /* 5.0 5.0 5.0 */
x1=ceil(-4.4);y1=ceil(-4.5);z1=ceil(-4.6);
printf("%.1f %.1f %.1f\n",x1,y1,z1); /* -4.0 -4.0 -4.0 */
x1=ceil(-0);y1=ceil(0);z1=ceil(+0);
printf("%.1f %.1f %.1f\n",x1,y1,z1); /* 0.0 0.0 0.0 */
}
```

—27 double —Cdecl floor (double x);

floor 是【下舍入】(原意是底层的意思)。返回小于或等于x的最大整数。注意,结果为双精度值。适用于UNIX系统。

C>TYPE MATH24.C

```
#include "math.h"
main(){
double x1=floor(-4.4),x2=floor(-4.5),x3=floor(-4.6);
double y1=floor(4.4),y2=floor(4.5),y3=floor(4.6);
double z1=floor(x1),z2=floor(0),z3=floor(0x5f);
printf("%.2f %.2f %.2f\n",x1,x2,x3); /* -5.00 -5.00 -5.00 */
printf("%.2f %.2f %.2f\n",y1,y2,y3); /* 4.00 4.00 4.00 */
printf("%.2f %.2f %.2f\n",z1,z2,z3); /* -5.00 0.00 95.00 */
}
```

—28 double —Cdecl round(double value,int n); <自定义.h>

将浮点数 value 按四舍五入法则舍入到 n 位小数。

C>TYPE MATH25.C

```
double round(double value,int n)
{ long m;
double v=1.0;
if(value == 0.0)return 0.0;
while(n-->0) v *= 10;
m=(long)(value * v+(value>0? 0.5:-0.5));
return ((double)(m/v));
}
main()
{
double x=123.456789,y,z;
y=round(x,2);
x=round(x-0.002,2);
printf("%f %f\n",y,x);
}
```

—29 double —Cdecl fmod (double x, double y);

计算 x 对 y 的模即 X/Y 的余数 f , 其中 f 满足等式 $x=i*y+f$

这里 i 是商的整数部分, $0 \leq f < y$. f 可以看作商的小数部分与除数的积, 如果小数部分有意义的话。

显然, 求模运算 (参见运算符 `%` 的说明) 只是它的特殊情况, 那里是求两个整数相除的余数。注意被除数和除数为特殊值的情况。

C>TYPE MATH26.C

```
#include "math.h"
main(){
    double H=HUGE-VAL; /* 最大值, 无穷大 */
    double x1=fmod(-4.1,0), x2=fmod(4.5,0), x3=fmod(H,0); /* 除数为 0 */
    double y1=fmod(0,H), y2=fmod(-1.19,H), y3=fmod(5,H); /* 除数为无穷大 */
    double z1=fmod(0,H), z2=fmod(0,-1), z3=fmod(0,1.0000001); /* 被除数为 0 */
    double w1=fmod(H,0), w2=fmod(H,-0.54), w3=fmod(H,1.8); /* 被除数为无穷大 */
    double u1=fmod(-4.1,-2), u2=fmod(4.5,-2), u3=fmod(-4.5,2); /* 正常情况 */
    printf("%f\n", fmod(H,H)); /* 0.000000 */
    printf("%.2f %.2f %.2f\n", x1,x2,x3); /* 0.00 0.00 0.00 */
    printf("%.2f %.2f %.2f\n", y1,y2,y3); /* 0.00 -1.19 5.00 */
    printf("%.2f %.2f %.2f\n", z1,z2,z3); /* 0.00 0.00 0.00 */
    printf("%.2f %.2f %.2f\n", w1,w2,w3); /* 0.00 0.33 0.36 */ /* 溢出 */
    printf("%.2f %.2f %.2f\n", u1,u2,u3); /* -0.1 0.50 -0.50 */
}
```

—30 double —Cdecl modf(double x, double *ipart);

它将 x 分成整数部分和小数部分: 整数部分存在 `ipart` 所指的变量中, 返回小数部分。

C>TYPE MATH27.C

```
#include "math.h"
main(){
    double H=HUGE-VAL;
    double x1,y1,z1,z2,u1,u2,u3,w1,w2,w3;
    double xx1,yy1,zz1,zz2,uu1,uu2,uu3;
    double *ipart0=&xx1, *ipart1=&yy1, *ipart2=&zz1; /* 指向具体的变量 */
    double *ipart3=&zz2, *ipart4=&uu1, *ipart5=&uu2, *ipart6=&uu3;
    x1=modf(0,ipart0);
    y1=modf(H,ipart1);
    z1=modf(-4.1,ipart2), z2=modf(1.0000000001,ipart3);
    w1=fmod(-44.1,-2.2), w2=fmod(44.5,-2.2), w3=fmod(-44.5,2.2);
    /* fmod 为求模 (即余数) */
    u1=modf(-44+w1,ipart4), u2=modf(44+w2,ipart5), u3=modf(w3,ipart6);
    printf("%.3f %.3f\n", x1,xx1); /* 0.000 0.000 */
    printf("%.3f %.3f\n", y1,yy1);
    /* 0.000 1.7976931348623157100000000000000000000000000e+308 */
    printf("%.3f %.3f\n", z1,zz1); /* -0.10 -4.000 */
    printf("%.3f %.3f\n", z2,*ipart3); /* 0.000 1.000 */
    printf("w1=%.3f\n", w1); /* w1=-0.10 */
    printf("%.3f %.3f\n", u1,uu1); /* -0.10 -44.000 */
}
```

```

printf("w2=%.3f\n",w2);           /* w2=0.500      */
printf(" %.3f %.3f\n",u2,*ipart5); /* 0.500 44.000 */
printf("w3=%.3f\n",w3);           /* w3=-0.500     */
printf(" %.3f %.3f\n",u3,uu3);    /* -0.500 -0.000 */
}

```

—31 double cdecl poly(double x, int degree, double coeffs []);
 为方便叙述,幕次 degree 用 n 代替,系数 coeffs 用 c[i] 表示,其中
 i=n, n-1, ..., 1, 0

则 poly (即 polynome) 函数是求 n 次多项式

$$c[n] * x^n + c[n-1] * x^{(n-1)} + \dots + c[1] * x + c[0]$$

的值。值太大时函数自动调用 —matherr() 处理。适用于 UNIX 系统。

```

C>TYPE MATH28.C
#include "math.h"
main(){
double c[]={1,2,3,4}; /* 定义了 c[0]~c[3] 的值 */
double x=2,fx;
int n=4;
while(n--){
    fx=poly(x,n,c);
    printf("n=%d fx=%f\n",n,fx);
}
n=4;
do
{
    fx=poly(x,n,c);
    printf("n=%d fx=%f\n",n,fx);
}while(n--);
}
/* 程序输出结果为
n=3 fx=49.000000
n=2 fx=17.000000
n=1 fx=5.000000
n=0 fx=1.000000
n=4 fx=81.000000 /* 这是不正确的值,因未定义 c[4] 的值! */
n=3 fx=49.000000
n=2 fx=17.000000
n=1 fx=5.000000
n=0 fx=1.000000
*/

```

—32 #define max(a,b) (((a)>(b))?(a):(b))

—33 #define min(a,b) (((a)<(b))?(a):(b))

max() 是获 a 与 b 中最大值的宏; min() 是获 a 与 b 中最小值的宏。a 和 b 可以是表达

式。

—34 #define sign(s) ((int)(s)>0? 1:((int)(s)<0? -1:0)) <自定义.h>
当表达式 s 的值大于 0 时其值为 1, 小于 0 时为 -1, 等于 0 时为 0。

C>TYPE MATH29.C

```
#define sign(s) ((int)(s)>0? 1:((int)(s)<0? -1:0))
main()
{
    float x=-100.3,y=0,z=105.5;
    printf("x=%d,y=%d,z=%d\n",sign(x),sign(y),sign(z));
} /* 程序输出: x=-1,y=0,z=1 */
/* math.02z */
```

—35 double —Cdecl atof(const char *s);

它相当于 strtod(const char *s, NULL);, 将字符串 s 转换为双精度值。它把串中的数全当十进制数处理。它能识别:

[制表符或空格串][+/-]数字串[.数字串][e/E[+/-]整数]

遇不能识别的字符时转换终止, 返回已转换的字符。如字符串不能转换为对应类型的数, 则返回 0。

它适用于 UNIX 系统。

C>TYPE MATH30.C

```
#include "stdlib.h"
main() /* 程序中专门列出制表符使用情况。*/
{
    double d[20]={0};
    char a1[]="+1231.1981e-123"; /* 溢出或不能转换均当 0 处理 */
    char a2[]="5028E9";
    char a3[]="\a-2010.952";
    char a4[]="\b88.952";
    char a5[]="\f+1231.1981e-1"; /* 制表符: \f 换页 */
    char a6[]="\n502.85E2"; /* 制表符: \n 换行 */
    char a7[]="\r-2010.952"; /* 制表符: \r 回车 */
    char a8[]="\t034e-1"; /* 制表符: \t 跳格(横向) */
    char a9[]="\v+1231.1981e-1"; /* 制表符: \v 跳行(纵向) */
    char a10[]="\5028E99";
    char a11[]="\'-2010.952";
    char a12[]="\88.952";
    char a13[]="\? +1231.1981e-1";
    char a14[]="\50"; /* Error: a14[]="\500" Numeric constant too large */
    char a15[]="\050";
    char a16[]="\x-2010.952";
    char a17[]="\x-2010";
    char a18[]="\8\t34e-1"; /* 制表符不在头部, 无效 */
    char a19[]="\t\t34e-1"; /* 头部两个连续制表符起都合法 */
    d[0]=atof(a1); printf("%f\n", d[0]); /* 0.000000 */
    d[1]=atof(a2); printf("%f\n", d[1]); /* 5028000000000.000000 */
}
```

```

d[2]=atof(a3);printf("%f\n",d[2]); /*      0.000000 */
d[3]=atof(a4);printf("%f\n",d[3]); /*      0.000000 */
d[4]=atof(a5);printf("%f\n",d[4]); /*     123.119810 */
d[5]=atof(a6);printf("%f\n",d[5]); /*    50285.000000 */
d[6]=atof(a7);printf("%f\n",d[6]); /*   -2010.952000 */
d[7]=atof(a8);printf("%f\n",d[7]); /*      3.400000 */
d[8]=atof(a9);printf("%f\n",d[8]); /*     123.119810 */
d[9]=atof(a10);printf("%f\n",d[9]); /*      0.000000 */
d[10]=atof(a11);printf("%f\n",d[10]); /*      0.000000 */
d[11]=atof(a12);printf("%f\n",d[11]); /*      0.000000 */
d[12]=atof(a13);printf("%f\n",d[12]); /*      0.000000 */
d[13]=atof(a14);printf("%f\n",d[13]); /*      0.000000 */
d[14]=atof(a15);printf("%f\n",d[14]); /*      0.000000 */
d[15]=atof(a16);printf("%f\n",d[15]); /*      0.000000 */
d[16]=atof(a17);printf("%f\n",d[16]); /*      0.000000 */
d[17]=atof(a18);printf("%f\n",d[17]); /*      8.000000 */
d[18]=atof(a19);printf("%f\n",d[18]); /*      3.400000 */
}

```

—36 double —Cdecl strtod (const char *s, char **endptr);

将字符串 s 转换为双精度值。

s 应是一个可解释为双精度值的字符串序列；

[ws][sn][ddd][.][ddd][fmt[sn]ddd]

这里

方括号表示其内容是可选的，方括号本身不是字符串组成部分。

ws 制表符或空白字符组成的字符串

sn 数符 + 或 - 号

ddd 数字 0 ~ 9 组成的序列

fmt 字母 e 或 E

. 小数点

注意，它们的排列顺序不能颠倒，例如字符串 "+\t23" 将被转换为 0！

strtod 在碰到不能解释为合适精度值的第一个字符时便停止读字符串，此时如果 endptr 不是空指针 (NULL)，则 strtod 便将 endptr 指向停止扫描的字符串（不能解释的第一个字符和它以后字符构成的串），即 endptr 等于不能解释字符所在内存中地址。

有关制表符使用见上述 atof() 函数。

C>TYPE MATH31.C

#include "stdlib.h"

main()

{

double d[8]={0};

static char a0[2]=""; /* 数组无 static 时下面的指针数组 p 不能初始化 */

char a1[]="+1231.1981e-1";

char a2[]="502.85E2";

char a3[]="-2010.952";

char a4[]="0x34"; /* 十六进制数 */

char a5[]="+12A31.1981e-1"; /* 有不能解释的字符 A */


```

char a6[]="50E2.85E2",          /* 有两个 E          */
char a7[]="-2010.9*52",          /* 夹有一个 *          */
char a8[]="0x34e-1",
char *p[2]={&a0[0],&a0[1]},      /* p 为指针数组 */
char **s=p,                      /* s 为指向指针的指针,为便于理解,故让它指向 p */
p[0]=a1,                          /* 指针 p[0] 指向数组 a1 */
d[0]=strtod(a1,s),printf("%f\n",d[0]); /* 123.119810 */
p[0]=a2,
d[1]=strtod(a2,s),printf("%f\n",d[1]); /* 50285.000000 */
p[0]=a3,
d[2]=strtod(a3,s),printf("%f\n",d[2]); /* -2010.952000 */
p[0]=a4,
d[3]=strtod(a4,s),printf("%f\n",d[3]); /* 0.000000 */
p[0]=a5,
d[4]=strtod(a5,s),printf("%f\n",d[4]); /* 12.000000 */
p[0]=a6,
d[5]=strtod(a6,s),printf("%f\n",d[5]); /* -5000.000000 */
p[0]=a7,
d[6]=strtod(a7,s),printf("%f\n",d[6]); /* -2010.900000 */
p[0]=a8,
d[7]=strtod(a8,s),printf("%f\n",d[7]); /* 0.000000 */
}

```

用 F7 热键调试时逐步可得下列调试表达式的值。

```

p,s,{ "", "" }
* * s, '\x0'
* s, s, ""
* s, p, DS:01D4 a0
s, p, DS: FFE2 /* 如将 p 数组说明成 static, 则其变成 s, p, DS: FFE2 p */
D[0], 8f, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
p, s, { "+1231.1981e-1", "" }
* * s, '+'
* s, s, "+1231.1981e-1"
* s, p, DS: FF8C
s, p, DS: FFE2
D[0], 8f, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
p, s, { "", "" }
* * s, '\x0'
* s, s, ""
* s, p, DS: FF99
s, p, DS: FFE2
D[0], 8f, 123.11981, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
p, s, { "502.85e2", "" }
* * s, '5'
* s, s, "502.85E2"
* s, p, DS: FF9A
s, p, DS: FF9A

```

```

D[0],8f,123.11981, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
p,s:{"", ""}
* * s: '\x0'
* s,s: ""
* s,p:DS:FFA2
s,p:DS:FFE2
D[0],8f,123.11981, 50285.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
p,s:{"-2010.952", ""}
* * s: '-'
* s,s: "-2010.952"
* s,p:DS:FFA4
s,p:DS:FFE2
D[0],8f,123.11981, 50285.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
p,s:{"", ""}
* * s: '\x0'
* s,s: ""
* s,p:DS:FFAD
s,p:DS:FFAD
D[0],8f,123.11981, 50285.0, -2010.952, 0.0, 0.0, 0.0, 0.0, 0.0
p,s:{"0x34", ""}
* * s: '0'
* s,s: "0x34"
* s,p:DS:FFAE
s,p:DS:FFE2
D[0],8f,123.11981, 50285.0, -2010.952, 0.0, 0.0, 0.0, 0.0, 0.0
p,s:{"x34", ""}
* * s: 'x'
* s,s: "x34"
* s,p:DS:FFAF
s,p:DS:FFE2
D[0],8f,123.11981, 50285.0, -2010.952, 0.0, 0.0, 0.0, 0.0, 0.0
p,s:{"+12A31.1981e-1", ""}
* * s: '+'
* s,s: "+12A31.1981e-1"
* s,p:DS:FFB4
s,p:DS:FFE2
D[0],8f,123.11981, 50285.0, -2010.952, 0.0, 0.0, 0.0, 0.0, 0.0
p,s:{"A31.1981e-1", ""}
* * s: 'A'
* s,s: "A31.1981e-1"
* s,p:DS:FFB7
s,p:DS:FFE2
D[0],8f,123.11981, 50285.0, -2010.952, 0.0, 12.0, 0.0, 0.0, 0.0
p,s:{"50E2.85E2", ""}
* * s: '5'
* s,s: "50E2.85E2"

```

```

*s,p,DS,FFC4
s,p,DS,FFE2
D[0],8f,123.11981, 50285.0, -2010.952, 0.0, 12.0, 0.0, 0.0, 0.0
p,s,{".85E2",""}
**s:','
*s,s:".85E2"
*s,p,DS,FFC8
s,p,DS,FFE2
D[0],8f,123.11981, 50285.0, -2010.952, 0.0, 12.0, 5000.0, 0.0, 0.0
p,s,{"-2010.9*52",""}
**s:'-'
*s,s:"-2010.9*52"
*s,p,DS,FFCE
s,p,DS,FFE2
D[0],8f,123.11981, 50285.0, -2010.952, 0.0, 12.0, 5000.0, 0.0,, 0.0
p,s,{"*52",""}
**s: '*'
*s,s:"*52"
*s,p,DS,FFD5
s,p,DS,FFE2
D[0],8f,123.11981, 50285.0, -2010.952, 0.0, 12.0, 5000.0, -2010.9, 0.0
p,s,{"0x34e-1",""}
**s:'0'
*s,s:"0x34e-1"
*s,p,DS,FFDA
s,p,DS,FFE2
D[0],8f,123.11981, 50285.0, -2010.952, 0.0, 12.0, 5000.0, -2010.9, 0.0
p,s,{"x34e-1",""}
**s:'x'
*s,s:"x34e-1"
*s,p,DS,FFDB
s,p,DS,FFE2
D[0],8f,123.11981, 50285.0, -2010.952, 0.0, 12.0, 5000.0, -2010.9, 0.0

```

从上面我们可以清楚地看到指针数组、指向指针的指针和 endptr 的真实含意。

```
—37 int —Cdecl atoi (const char *s);
```

或

```
#if —STDC—
```

```
#define atoi(s) ((int)atol(s))
```

```
#endif
```

函数将字符串 s 转换为一个整型数,它相当于定义为

```
atoi(const char *s)
```

```
{ return (int)atof(s); }
```

对小数部分是舍去而不是四舍五入,这和 BASIC 中的 INT() 函数有点相同。

```
C>TYPE MATH32.C
```

```
#include "stdlib.h"
```

```

main()
{
char *s="123.556e8";
char *t="[]-012.4A";
int x;
double y;
long z;
x=atoi(s);
y=atof(s);
z=atol(t);
printf("x=%d,y=%lf,z=%ld\n",x,y,z);
} /* 程序输出:x=123,y=12355600000.000000,z=-12 */

```

—38 long —Cdecl atol (const char *s);

它把串 s 转换为长整型数。s 可以是

[制表符或空格串][+/-][数字串]

如遇不能识别的字符则转换停止,返回已转换的字符。如果 s 不能转换为对应类型的数,则结果为 0。

适用于 UNIX 系统。

—39 long —Cdecl strtol (const char *s, char **endptr, int radix);

将字符串 s 转换为长整型值, s 可以是表示某种数制的串,可用参数 radix 说明串 s 的【数制的基数】,转换后的数一定是十进制数。

s 应是一个可解释为长整型值的字符串序列:

[ws][sn][0][x][ddd]

这里

方括号表示其内内容是可任选的,方括号本身不是字符串组成部分。

ws 制表符或空白字符组成的字符串

sn 数符 + 或 - 号

0 数 0

x 英文字母 x(小写)或 X(大写)

ddd 数字 0 ~ 9 组成的序列

strtol 在碰到不能解释为合适整型值的第一个字符(例如,小数点)时便停止读字符串,此时如果 endptr 不是空指针(NULL),则 strtol 便将 endptr 指向停止扫描的字符串(从不能解释字符和它以后字符构成的串),即 endptr 记录不能解释字符所在内存中的地址。

radix 数 0 或正整数 2 ~ 36。当其值为 2 ~ 36 时,它被解释成串 s 按 radix 数制转换成十进制数,即 radix 表示数进制的基数。例如,其值为 2,则表示把串 s 看成二进制数转换成十进制数;其值为 16 时,把串当成十六进制数转为十进制数;当其值为 10 时,[ddd] 中头部连续的数字 0 将被忽略。

当 radix 为数 0 时,这时串的基数由串的

[0][x][ddd]

部分隐含决定。对八进制数应是

0[ddd]

或者说,第一个字符应是 0,而第二个字符应为 1~7(它也是转换后的数字的组成部分);对十六进制数应是

0x[ddd]

即第一个字符应为 0,第二个字符应为英文字母 x 或 X。其余情况便当十进制数处理。

当 radix 实际表示数制 n(2~36) 时,则 strtol 对串 s 中的 0~@@ 的字符能解释,而对其余字符被认为是不能解释成适当值的字符。@@ 与进制 n 的对应关系如下所示:

n =	2	3	4	5	6	7	8	9	10	11	12	13	14	15	6		
@@ =	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
n =	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	
@@ =	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	
n =	33	34	35	36													
@@ =	W	X	Y	Z	/* 注意:英文字母大小写是一样的,即效果相同 */												

当 radix 的值为 1,或小于 0,或大于 36 时均无效。任意一个无效的 radix 值都将使结果返回数字 0,且 endptr 指向跟在串 s 后边的另一字符串的起始地址。

```
C>TYPE MATH33.C
#include "stdlib.h"
main()
{
    int n=10;
    long d[8]={0};
    static char a0[2]="";
    char a1[]="+123";
    char a2[]="+051";
    char a3[]="-0x1F";
    char a4[]="-12A3";
    char a5[]="-081";
    char a6[]="+0X1G";
    char a7[]="0";
    char a8[]="";
    char *p[2]={&a0[0],&a0[1]};
    char **s=p;
    p[0]=a1;
    d[0]=strtol(a1,s,n);printf("%ld\n",d[0]); /* 123 */
    p[0]=a2;
    d[1]=strtol(a2,s,n);printf("%ld\n",d[1]); /* 51 */
    p[0]=a3;
    d[2]=strtol(a3,s,n);printf("%ld\n",d[2]); /* 0 */
    p[0]=a4;
    d[3]=strtol(a4,s,n);printf("%ld\n",d[3]); /* -12 */
    p[0]=a5;
    d[4]=strtol(a5,s,n);printf("%ld\n",d[4]); /* -81 */
    p[0]=a6;
    d[5]=strtol(a6,s,n);printf("%ld\n",d[5]); /* 0 */
    p[0]=a7;
    d[6]=strtol(a7,s,n);printf("%ld\n",d[6]); /* 0 */
    p[0]=a8;
    d[7]=strtol(a8,s,n);printf("%ld\n",d[7]); /* 0 */
}
```

```

d[6]=strtol(a7,s,n);printf("%ld\n",d[6]); /* 0 */
p[0]=a8;
d[7]=strtol(a8,s,n);printf("%ld\n",d[7]); /* 0 */
}

```

当更换程序中数制基数 n 值时,有

n	0	2	8	10	16	-1	1	37
d[0]	123	1	83	123	291	0	0	0
d[1]	41	0	41	51	81	0	0	0
d[2]	-31	0	0	0	-31	0	0	0
d[3]	-12	-1	-10	-12	-4771	0	0	0
d[4]	0	0	0	-81	-129	0	0	0
d[5]	1	0	0	0	1	0	0	0
d[6]	0	0	0	0	0	0	0	0
d[7]	0	0	0	0	0	0	0	0

—40 unsigned long —Cdecl strtoul (const char *s,char **endptr,int radix);

将字符串 s 转换为无符号的长整数。这里,对 s 有严格的限制:

[空格符][+][0][x][ddd]

在有效数字串之前只允许空格存在,其它无意义的字符会导致结果为不正确的值(通常为0)。有效字符串之后允许其它字符存在,这可以通过 $endptr$ 观察。 $endptr$ 是一个指向指针的指针。如果它指向 s ,则当有效数字串转换后, $endptr$ 指向剩余的子串(即 s 去了前面有效数字串后的串)。 $radix$ 是数制的基数,有效值为 $2 \sim 36$ 之间的整数。当已知待转换的数字串是 8、10 或 16 进制数时,可用 0 给 $radix$ 赋值, Turbo C 会自动分析出基数值。当 $radix > 10$ 时,则字母 A ~ Z 是构成数字串的有效字符。

C>TYPE MATH34.C

```
#include "stdlib.h"
```

```
#include "stdio.h"
```

```
#define PNEXT printf("指向下一个字符=%s\n",endptr)
```

```
main()
```

```
{
```

```
unsigned long x;
```

```
char *str=" 0x07HfZYX"; /* 0x7FFF,0X7Hf,0X07FFF 等都可以 */
/* 串开头允许有空格,但不允许是无意义的字符 */
```

```
char *endptr;
```

```
endptr=str; /* 让 endptr 先指向 str,以观察变化 */
```

```
PNEXT;
```

```
x=strtoul(str,&endptr,16); /* 注意,串中为 16 进制数,所以你只能将 */
/* 数制基数写成 16,否则不会得到正确值 */
/* 如果串中是 C 语言的 8、10 或 16 进制 */
/* 数时,也可以将 radix 处写成 0,程序 */
/* 会自动判断而得到正确值。所以这里 16 */
```

```
PNEXT; /* 可写成0 */
```

```
printf("转换后的长整数=%ld\n",x); /* 输出时可用十进制数表示 */
```

```

}
/* 程序输出:指向下一个字符=0x07fffZYX
           指向下一个字符=ZYX
           转换后的长整数=32767

```

如果你在程序中增加和修改语句:

```

#include "errno.h"
printf("errno=%d\n",errno);
char *str=" 0x07fff12345678901234ZYX";
则程序输出:
指向下一个字符= 0x07fff12345678901234ZYX
errno=34
指向下一个字符=678901234ZYX
转换后的长整数=-1      */

```

```

-41 char *—Cdecl ecvt (double value, int ndig, int *dec,int *sign);

```

ecvt 把一个双精度数 value 转换为一个有 ndig 个数字并以空字符 ('\0') 为终止符的字符串,返回值是一个指向该字符串的指针(静态指针,它所指单元的内容在每次调用本函数会被重写)。

ndig 说明 value 如何被圆整,表示返回串(不妨设为 yyyyy)中数字个数(其值范围为 1 ~ 18。ndig<1 时按 1 处理,大于 18 时按 18 处理)。

返回的字符为 ASCII 码形式,它用数字 0 来右对齐以填满所要求的位数(ndig)。

如果小数点位于串最左端(有效位),则计算指数(或阶)值,串中无小数点出现。如果该值还为 0,则阶为 0,串用全 0 填充。

如果 value 值为无穷大(或 NAN),则指数为 MAXSHORT,串全为 9。

整型指针 dec 也指向指数变量。指数值由 *dec 指出(有±之分,数 0 被认为有整数位 1)。

value 与其它几个变量的关系是

```

value=(( *sign) > 0 ? -1:1) * 0. yyyyyE(*dec)

```

value 的符号(即数符)由 *sign 指出,当 value 为正数时它为 0,负数为 1;数 0 被认为是正数。最低数位是四舍五入。

从程序 MATH35.C 中可以看到,当 ndig 为负值或 value 很大时结果均有可能不正确。

它适用于 UNIX 系统。

```

C>TYPE MATH35.C

```

```

#include "float.h"
#include "stdlib.h"
#define PMP ptr=ecvt(a1,n,m,p),printf("%s %d %d\n",ptr,*m,*p)
main()
{
double a1=89.9e23;
char *ptr;
int k,n;
int *m=(int *)malloc(sizeof(int)),*p=(int *)malloc(sizeof(int));
for(k=0;k<2;k++)
{ if (k==0)n=8;
else n=-8; /* 当 n=8 时输出 当 n=-8 时输出 */

```



```

1500000000 2 0
1500000000 32767 0
17976931348623157100000000000000000000 309 0
17976931348623157100000000000000000000 309 0
899000000000000000160000000 25 1
0 1 0
0 1 0
0 1 1
0 1 0
0 1 1
90 2 0
90 2 0
31 2 0
15 2 0
15 32767 0
17976931348623157100000000000000000000 309 0

```

从中也可以看到,当 `ndig` 为负值或 `value` 很大时结果均有可能不正确。

`fcvt` 本身返回一个字符指针,它是静态数据指针,它所指内存单元的内容在每次调用 `fcvt` 时被重写。

它适用于 UNIX 系统。

—43 `char * Cdecl fcvt (double value, int ndec, char * buf);`

它将值 `value` 转换为以空字符为终结符的 ASCII 码字符串,结果放在 `buf` 中。输出格式如果能输出 F 格式,则优先输出 F 格式:

`xxxxx.yyyyy` 或 `-xxxxx.yyyyy`

它接近于 `value` 原样(但不会出现 `xxxxx.` 或 `-xxxxx.` 即小数点在最右边的形式),实际位数受到 `ndec` 约束。这种 F 格式和 `fcvt` 的 F 格式内容上和表示的数含义是不同的。只是为方便叙述,仍称它为 F 格式。

否则输出格式有点像 Fortran 中的 E 格式输出:

`x.yyyyyE ± dd` 或 `-x.yyyyyE ± dd`

原 `value` 中数符 `+` 是不会输出的。

具体一点讲,在不考虑 `value` 数符(`+` 或 `-`)及小数点时设它有 `m` 位,则可分三种情况判断它以何种格式输出。

(1) `value` 的整数部分不为 0 时,则

当 $m \leq ndec + 1$ 时,印出 F 格式。当有 `ndec+1` 位时,它仅供参考,可能有舍入;否则以 E 格式印出,其中串 `yyyyy` 含字符数为 `ndec-1`,两个 `dd` 按实际换算。

(2) `value` 的整数部分为 0,即纯小数,此时

当小数点后连续尾随的 0 的个数超出 3 个时,如 `-0.00005`,则按 E 格式输出;否则按 F 格式输出。

小数部分尾部无效 0 一律被丢掉,不被输出。

如果值无穷,则输出可能为 `± 9E+999`。

尽管输出位数可多于 16 位,但超过部分对许多机器已不精确,故一般 `ndec` 应小于 18。

注意:`gcvt` 返回结果和 `buf` 中的内容是一样的。使用指针时但是最好用动态地址分配函

数给 buf 分配足够的存储空间。

它适用于 UNIX 系统。

```
C>TYPE MATH36.C
#include "alloc.h"
#include "stdlib.h"
#define PTR ptr=gcvt(a[j],n[k],b);\
    printf("%lg %s %s",a[j],ptr,b)

main()
{
double a[]={123.456789,1234567.89560,0,+1234567.0,2341.23,-321,-422.0,
    .534567,-.6e8,075,0x80,12345.88,0.4,0.5,0.04,0.05,0.004,
    0.005,0.0004,0.0005,0.00004,0.00005};
char *b=(char *)calloc(20,1); /* 也可用char b[20]; */
static char *ptr;
int j,k,n[]={4,1};
for(k=0;k<2;k++)
{
    for(j=0;j<22;j++)
    {
        PTR,printf("    ");
        PTR,printf("\n");
    }
    printf("\n");
}

/* 程序输出 (同使用缺省精度的 printf() 比较):
123.457 123.5 123.5 123.457 123.5 123.5
1234570 1.235e+06 1.235e+06 1234570 1.235e+06 1.235e+06
0 0 0 0 0 0
1234570 1.235e+06 1.235e+06 1234570 1.235e+06 1.235e+06
2341.23 2341 2341 2341.23 2341 2341
-321 -321 -321 -321 -321 -321
-422 -422 -422 -422 -422 -422
0.534567 0.5346 0.5346 0.534567 0.5346 0.5346
-6e+07 -6e+07 -6e+07 -6e+07 -6e+07 -6e+07
61 61 61 61 61 61
128 128 128 128 128 128
12345.9 12350 12350 12345.9 12350 12350
0.4 0.4 0.4 0.4 0.4 0.4
0.5 0.5 0.5 0.5 0.5 0.5
0.04 0.04 0.04 0.04 0.04 0.04
0.05 0.05 0.05 0.05 0.05 0.05
0.004 0.004 0.004 0.004 0.004 0.004
0.005 0.005 0.005 0.005 0.005 0.005
0.0004 0.0004 0.0004 0.0004 0.0004 0.0004
0.0005 0.0005 0.0005 0.0005 0.0005 0.0005
```

```

4e-05    4e-05    4e-05    4e-05    4e-05    4e-05
5e-05    5e-05    5e-05    5e-05    5e-05    5e-05
123.457  1e+02  1e+02    123.457  1e+02  1e+02
1234570  1e+06  1e+06    1234570  1e+06  1e+06
0         0         0         0         0         0
1234570  1e+06  1e+06    1234570  1e+06  1e+06
2341.23  2e+03  2e+03    2341.23  2e+03  2e+03
-321     -3e+02 -3e+02    -321     -3e+02 -3e+02
-422     -4e+02 -4e+02    -422     -4e+02 -4e+02
0.534567 0.5     0.5     0.534567 0.5     0.5
-6e+07   -6e+07 -6e+07    -6e+07   -6e+07 -6e+07
61        60        60        61        60        60
128       1e+02  1e+02    128       1e+02  1e+02
12345.9   1e+04  1e+04    12345.9   1e+04  1e+04
0.4       0.4     0.4     0.4       0.4     0.4
0.5       0.5     0.5     0.5       0.5     0.5
0.04      0.04    0.04    0.04      0.04    0.04
0.05      0.05    0.05    0.05      0.05    0.05
0.004     0.004   0.004   0.004     0.004   0.004
0.005     0.005   0.005   0.005     0.005   0.005
0.0004    0.0004  0.0004  0.0004    0.0004  0.0004
0.0005    0.0005  0.0005  0.0005    0.0005  0.0005
4e-05     4e-05   4e-05   4e-05     4e-05   4e-05
5e-05     5e-05   5e-05   5e-05     5e-05   5e-05
*/

```

—44 char * —Cdecl itoa(int value, char * string, int radix);

把值 value 转换为以空字符 ('\0') 终结的串, 并把结果放在 string 所指串中。radix 指明被转换的 value 的数进制的基数值, 它必须在 2 ~ 36 之间。

如果 value 为负数, radix=10, 则 string 所指串首字符为减号。

注意: string 所指空间应足够大, 以便能容纳所返回的所有字符 (包括 '\0'), 通常应分配 17 个字节, 它是 itoa() 能返回的最大字节数。

函数返回指向 string 的指针。

—45 char * —Cdecl ltoa(long value, char * string, int radix);

函数把长整数 value 转换成与其相当的字符串, string 指向该字符串。string 所指的数组变量应有足够的长度 (最大为 33 个字节) 来存放结果, 否则可能出错。radix 为数制的基数, 可选 2 ~ 36。

C>TYPE MATH37.C

```
#include "stdlib.h"
```

```
#include "stdio.h"
```

```
main()
```

```
{
```

```
    unsigned long x;
```

```
    char s[33], *str=s;
```

```
    printf("%s\n", ltoa(0x45678, str, 10)); /* 284280 */
```

```
    printf("%s\n", str); /* 284280 */
```

```

printf("%s\n",s);          /* 284280 */
ltoa(0x10000, str, 10);
printf("%s\n",s);          /* 65536 */
ltoa(0x7fffffff, str, 10);
printf("%s\n",s);          /* 2147483647 */
ltoa(0x7FFFFFFF, str, 2);
printf("%s\n",s);          /* 11111111111111111111111111111111 */
ltoa(0x7fffffff, str, 36);
printf("%s\n",s);          /* zik0zj */
ltoa(0x8FFFFFFF, str, 10);
printf("%s\n",s);          /* -1879048193 */
ltoa(0x5fffffff, str, 10);
printf("%s\n",s);          /* -1 */
}

```

— 46 char * — Cdecl ultoa(unsigned long value, char * string, int radix);

将无符号长整数转换成一个字符串,串最多为 33 个字符。当 value 为负数, radix=10 时, string 所指串的首字符不会是减号,也不进行溢出检查。

C>TYPE MATH38.C

```
#include "stdlib.h"
```

```
#include "alloc.h"
```

```
main()
```

```
{
```

```
long int x=-1;
```

```
char *s=(char *)calloc(34,1);
```

```
char v[34];
```

```
s=ultoa(x,v,10);
```

```
printf("%s=%s",s);
```

```
printf("%s\n",v);
```

```
} /* 程序输出一些乱七八糟的东西 */
```

```

—47 double —Cdecl —matherr(—mexcep why, char *fun, double *arg1p,
double *arg2p, double retval);

```

浮点运算处理函数。它是一个 TC 内部函数,不能直接为源程序使用。当数学库函数调用时出现错误时库函数便自动调用它。TC 内部调用它时首先检查源程序中是否有用户编写的 matherr() 函数。如果有,便转而调用并执行 matherr 函数;如果没有 matherr 函数,或者虽有而它返回 0 值,则调用 fprintf 函数打印出错信息,对外部变量 errno 赋值,最后返回 e.retval。e.retval 可以由用户在 matherr 函数中修改。如果 matherr 返回非 0 值,则本函数只返回 e.retval,其它事没有进行。为帮助理解,在下面 MATH39.C 程序中给出了其定义。尽管给出了定义,但它可执行而不能用 F7 热键跟踪,因此在其间加上了原函数没有的 printf 语句。值得指出,若把原函数的语句

```
if(matherr(&e)==0)
```

改成

```
if(matherr(&e)!=0)
```

则程序按后一语句执行。这使我们看到,在源程序中的定义的函数优先于 TC 同名的库函数!

函数中的参数类型 —mexcep(可理解为 math exception 助记符) 是在 math.h 中定义的一个枚举类型 :

```
typedef enum
{
DOMAIN = 1, /* 参数定义域错误,如 log (-1) */
SING, /* 参数异常,如 pow (0,-2) */
OVERFLOW, /* 上溢错误,函数结果大于 MAXDOUBLE,如 exp (1000) */
.....
```

```

{
    case DOMAIN:
        printf("argument DOMAIN error ! \n");
        if(strcmp(a->name,"log")==0)
        {
            a->retval=log(-(a->arg1));          /* 修改 retval 缺省返回值 */
            printf("LOG| DOMAIN Error\n");
            return(1);
        }
        return(1);
    case SING:
        printf("argument SINGularity ! \n");
        return 1;
    case OVERFLOW:
        printf("OVERFLOW range error ! \n");
        return 1;
    case UNDERFLOW:
        a->retval=0;                          /* 修改 retval 缺省返回值 */
        printf("UNDERFLOW range error! flush underflow to 0 \n");
        return 1;
    case TLOSS:
        printf("Total loss of precision, but ignore the problem\n");
        return 1;
    default:
        printf("CARE ,all other errors are fatal! \n");
        return(0);                          /* 不能处理,函数返回 0 */
}
}

#include "stdio.h"
#include "errno.h"
char * whyS[]=
{
    "DOMAIN",
    "SING",
    "OVERFLOW",
    "UNDERFLOW",
    "TLOSS",
    "PLOSS"
};

double __matherr(__mexcep why,          /* 等效函数 */
                char * fun,
                double * arg1p,
                double * arg2p,
                double retval
                )
{

```

```

struct exception e;
printf("\nwhy=%d funname=%s\n",why,fun);
printf("arg1p=%f addr=%p\narg2p=%f addr=%p\n", *arg1p,arg1p, *arg2p,arg2p);
printf("retval=%f\n",retval);
e.type=why;
e.name=fun;
e.arg1=(NULL==arg1p)? 0: *arg1p;
e.arg2=(NULL==arg2p)? 0: *arg2p; /* 以第一个参数为判断依据 */
e.retval=retval;
if(matherr(&e) != 0) /* 原函数语句为 if(matherr(&e) == 0),即 */
/* 如果源程序中无 matherr(),或 matherr() 返回 0 */
{
    fprintf(stderr," %s: %s error\n",fun,whyS[why-1]);
    errno=((OVERFLOW==why)|| (UNDERFLOW==why))? ERANGE:EDOM;
    printf("errno=%d\n",errno); /* #define ERANGE 34 结果太大 */
} /* #define EDOM 33 定义域错 */
return e.retval;
}
main()
{
    int i=-1,ki;
    double j=0.1,kf;
    printf("\nHUGE-VAL=%f\n",HUGE-VAL); /* #define HUGE-VAL -huge-dble */
    i=log(i);
    printf("i=-1 test end! \n");
    ki=log(0.0); /* 执行到此句,程序中断运行 */
    printf("ERROR: log(0.0)\n");
    i=log(1e-23);
    printf("Error! i=0\n");
    ki=log(j);
    printf("ki=%d\n",ki);
    kf=log(j);
    printf("kf=%f\n",kf);
}

```

将 `—matherr` 中语句 `if(matherr(&e)==0);` 改成 `if(matherr(&e)! =0);` 后运行结果为

```

HUGE-VAL=-1.797693134862315710000000000000000000000000e+308 /* 常数 */
why=1 funname=log
arg1p=-1.000000 addr=FFD4
arg2p=1.700707510824434990000000000000000000000000e+166 addr=0000
retval=-NAN
Execution: funname=log Errortype=1
arg1=-1.000000
arg2=1.700707510824434990000000000000000000000000e+166
retval=-NAN
argument DOMAIN error !
LOG| DOMAIN Error

```

```

log| DOMAIN error /* 如 --matherr 中 if(matherr(&e)==0); 则此句无 */
errno=33
i=-1 test endl
why-2 funname=log
arg1p=0.000000 addr=FFCA
arg2p=1.70070751082443499000000000000000000000000e+165 addr=0000
retval=-1.7976931348623157100000000000000000000000e+308
Execution: funname=log Errortype=2
arg1=0.000000
arg2=1.70070751082443499000000000000000000000000e+166
retval=-1.7976931348623157100000000000000000000000e+308
argument SINGularity !
log| SING error /* 如 --matherr 中 if(matherr(&e)==0); 则此句无 */
errno=33
Floating point error: Domain.

```

如将语句 `ki=log(0.0);` 去掉,忽略其它输出结果,有

```

ERROR: log(0.0)
Error! i=0
ki=-2
kf=-2.302585

```

```

--49 void --Cdecl srand (unsigned seed);
--50 #define randomize() srand((unsigned)time(NULL))

```

初始化随机数发生器,它用来建立由 `rand()` 函数所产生的伪随机序列数的第一个数。利用它,可使多个子程序用不同的伪随机序列。`seed` 常常取成为时间的函数:如

```

seed=(unsigned int)time(&variable)/2; /* &variable 为定义的某变量地址 */

```

它的取值范围为 0~32767,缺省时,seed=1,函数返回 346。为方便起见,stdlib.h 中定义了一个依赖时间的随机数发生器即宏 `randomize()`:

```

#define randomize() srand((unsigned)time(NULL))

```

程序员直接可用它产生伪随机序列。

```

--51 int --Cdecl rand(void);

```

从由随机数发生器产生的伪序列中任取一个数。在 `stdlib.h` 中定义了 `rand()` 可得的最大随机数是 32767,即

```

#define RAND--MAX 0x7FFF
C>TYPE MATH40.C
#include "stdlib.h"
#include "stdio.h"
#define PRAND for (k=0;k<5;k++)printf("%d\n",rand())
main()
{
int k,seed;
PRAND;
srand(seed=10);

```



```

PRAND;
seed=1;
srand(seed);
PRAND;
srand(10);
PRAND;
}

```

```

/* 程序输出:第一次 第二次 第三次 第四次
          346    3463    346    3463
          130    30957   130    30957
          10982   10345   10982   10345
          1090    10368   1090    10368
          11656   8444    11656   8444  */

```

—52 #define random(num) (rand() % (num))

定义的宏返回一个 0 ~ num-1 之间的一个随机数, num 为正整数。% 为两者相除 后的求余数运算。

```

C>TYPE MATH41.C
#include "stdlib.h"
#include "stdio.h"
#include "graphics.h"
#include "time.h"
#include "conio.h"
#define X1 260
#define Y1 140
#define X2 320
#define Y2 180
int colours;
main()          /* 在画出的计算机屏幕上从上往下动画显示 JiaLing!,
                  并随机改变显示颜色,直到按任一键显示才会终止 */
{
    int driver=DETECT,mode,color,k;
    initgraph(&driver,&mode,"");          /* 初始化图形系统 */
    if(graphresult() != grOk)
    {
        printf("图形初始化有问题:%s\n",grapherrormsg(graphresult()));
        exit(1);
    }
    colours=getmaxcolor();          /* colours 是最大可用颜色数 */
    setcolor(WHITE);                /* 画显示器框 */
    setfillstyle(1,DARKGRAY);
    bar3d(X1-20,Y1-20,X2+56,Y2+70,0,3);
    setfillstyle(CLOSE-DOT-FILL,BLUE);
    setfillstyle(SOLID-FILL,RED);    /* 画显示屏指示灯 */
    bar(X1+4,Y1+97,X1+20,Y1+102);
    setcolor(BLUE);                /* 画红、绿、兰按钮 */
}

```

```

circle(X2+28,Y2+60,4);
setcolor(GREEN);
circle(X2+16,Y2+60,4);
setcolor(MAGENTA);
circle(X2+4,Y2+60,4);
setcolor(WHITE);          /* 画机箱          */
setfillstyle(SOLID-FILL,DARKGRAY);    /* 设置填充颜色    */
bar3d(X1-60,Y1+120,X1+154,Y1+170,0,2); /* 画主框          */
bar3d(X1+20,Y1+126,X1+120,Y1+164,0,2); /* 画驱动器框      */
line(X1+20,Y1+145,X1+120,Y1+145);    /* 将驱动器框一分为二 */
setfillstyle(SOLID-FILL,GREEN);
bar(X1+26,Y1+130,X1+34,Y1+132);      /* 画上驱动器指示灯 */
bar(X1+26,Y1+150,X1+34,Y1+152);      /* 画下驱动器指示灯 */
setfillstyle(WIDE-DOT-FILL,RED);
bar(X1-24,Y1+128,X1-44,Y1+148);      /* 画左边部分      */
settextstyle(DEFAULT-FONT,HORIZ-DIR,1);
while(! kbhit())                    /* 按任一键退出循环 */
{
    color=random(colors-1)+1;        /* 产生随机颜色数 */
    if(color != BLACK) setcolor(color); /* 设置当前画线颜色 */
    for(k=0;k<8;k++)
    {
        setfillstyle(1,BLACK);      /* 选单色 BLACK (缺省背景色) 填充 */
        bar3d(X1-12,Y1-12,X2+48,Y2+52,0,1); /* 在显示器框中画显示器屏幕 */
        outtextxy(X1,Y1+k*10,"JiaLing!"); /* 你不能用这样的语句:
                                           setcolor(BLACK);
                                           outtextxy(X1,Y1+k*10,"");
                                           即用改变前景和输出空格符的办法
                                           来消去当前位置上的字符串! */
        delay(200);                  /* 延时,控制动画速度 */
    }
}
closegraph();                        /* 关闭图形系统 */
}

```

26.4 例

C>TYPE MATH42.C

```

#include "stdio.h"
#include "string.h"
#include "math.h"
char * zh();
#define YIWE 11 /* 亿位 */
main()          /* 将数字金额转换为大写金额程序 */
{              /* 根据杨晓加的程序改写并注释 */
    char * str;

```

```

double y=123456789.145;      /* 最大数为几亿,超过则不允许转换 */
str=zh(y);
printf("%.3lf== %s\n",y,str);
}
char * zh(double x)
{
int i,n,bz;
double xx=100
*x,yy;
char * c=je,je[14],temp[13];
char f1[10][3]={"零","壹","贰","叁","肆","伍","陆","柒","捌","玖"};
char f2[11][3]={"亿","仟","佰","拾","万","仟","佰","拾","元","角","分"};
/* 整数转换成字符,乘100是取到分,分后面第一位四舍五入 */
yy=floor(1000*x)-(floor(xx)*10);
xx=yy>4? ceil(xx),floor(xx); /* 如 0.145 将转换成 15 */
sprintf(je,"%.0lf",xx);
n=strlen(je);
if(n>YIWE)
{
printf("对不起,数太大(%.3lf),整数位数(%d)太多(最大允许%d位),\不能处理!\n",x,n-2,YI-
WE-2);
exit(1);
}
c=je=(char *)calloc(80,1);
bz=1; /* 正常标志 */
for(i=0;i<n;i++) /* 对 n 位数字逐位处理 */
{
strcpy(temp,&je[i]); /* temp 中存放从 &je[i] 开始的数字串 */
if(atoi(temp) == 0) /* 把数字字符串变成整数,为 0 表示位全为 '0' */
{
/* 从某位后数值为 0 时设标记 bz=2 */
bz=2;
break; /* 跳出 i 循环 */
}
if(je[i] != '0') /* 判断当前位 */
{
if(bz == 0)
strcat(c=je,f1[0]); /* 写上“零” */
strcat(c=je,f1[je[i]-'0']); /* 写上“壹”~“玖” */
/* 可用 Ctrl-F4 键操作,用类似 '9'-'0' 那样的估算表达式观察 */
bz=1;
strcat(c=je,f2[YIWE-n+i]); /* 写上“亿”~“分” */
}
else /* je[i]='0' 时 */
{
/* 判断第 8 位即拾万位且后三位都不为 0 时写“万”字 */
if((n-i==7 && (je[i-1] != '0' || je[i-2] != '0' || je[i-3] != '0'))
strcat(c=je,"万");
}
}
}

```

```

        if(n-i == 3)          /* 判断写“元”字 */
            strcat(c-je,"元");
        bz=0;                  /* 写“零”标志 */
    }
} /* i 循环结束 */
if(bz == 2) /* 特殊情况处理 */
{
    if(n-i >= 7 && n-i < 10)    /* 拾万位后全 0 时 */
        strcat(c-je,"万");
    if(n-i >= 3)                /* 拾元位后全 0 时 */
        strcat(c-je,"元");
    strcat(c-je,"正");          /* 分位为 0 时 */
}
return (c-je);                 /* 返回指向大写金额字符串的指针 */
}
/* 程序输出:123456789.145=壹亿贰仟叁佰肆拾伍万陆仟柒佰捌拾玖元壹角伍分 */

```

第二十七章 80x87 数学协处理器

27.1 概述

80x87 是为与 80x86 主微处理器并行操作而设计的。Turbo C 2.0 对应 8087/80287, 所以此处内容仅供参考 (特别 /*... */ 中内容是浮点处理, 并不一定是协处理器的)。

8087、80287 和 80387 分别产生于 1979 年、1982 年和 1985 年。使用协处理器可提高数值运算的速度, 而当系统没有装协处理器时, 80x86 CPU 则采用仿真程序来解释执行。

只有当机器上装有 80x87 时你才能用 80x87 函数, 并选择相应的 8087/80287 项进行编译源程序。

一 浮点堆栈

80x87 使用了 80 位内部结构, 遵循 IEEE 浮点格式。80287 是 40 引脚, 80387 是 68 引脚。

协处理器是面向堆栈的处理器, 数字可以用装载命令压入堆栈而用弹出命令从堆栈弹出。堆栈由 8 个寄存器 (R0 ~ R7, 或称堆栈单元) 组成, 每个寄存器长度为 80 位, 正好能存放一个【暂时实数】(或临时实数)。暂时实数可以不经转换直接存放到内存中去, 或从内存中取扩协处理器。协处理器的所有运算操作, 都是对栈中寄存器的内容进行的。在计算中所用的常数及中间结果也常放在栈中。

环境寄存器中用 3 位指向堆栈顶 (ST 或 ST(0))。

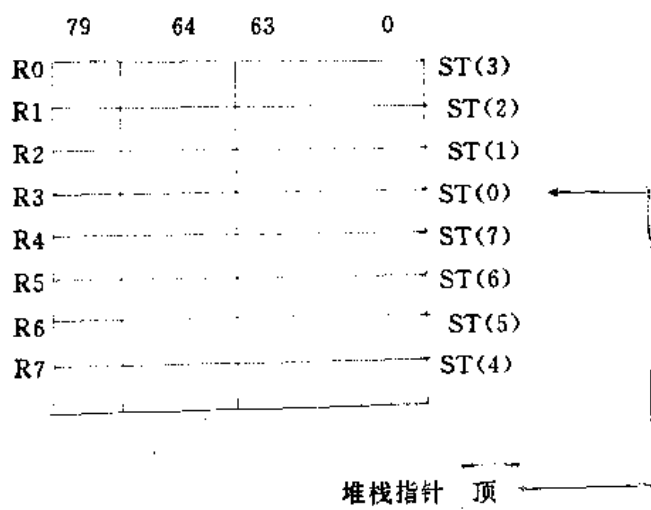


图 27-1

由分成若干个场的
8 个 80 位单元组
成的 80x87 堆栈

二 状态字 (Status Word)

它反映了协处理器的全部状态。

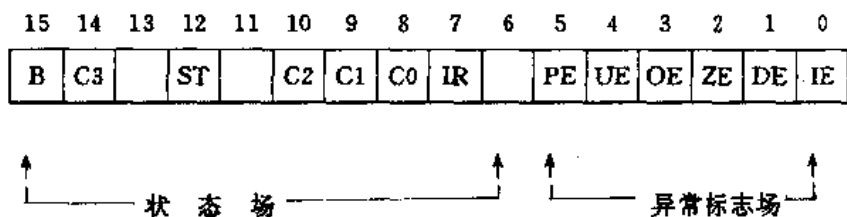


图 27-2

说明: /* ... */ 中的内容是 float.h 中定义的符号常数,下同。

位15 (B) 表示是否正在执行指令 (=1 时)或空闲 (=0 时)

位13,12 和11 该 3 位场是一个指针,它总是指向当前栈顶寄存器。或者说,它表示 (ST)

8 个堆栈单元中的哪一个单元现在是堆栈顶:

000 单元 0 是堆栈顶

001 单元 1 是堆栈顶

...

111 单元 7 是堆栈顶

入栈 (或称压入) 操作把 ST 减 1 后再把数值装入新的栈顶寄存器; 出栈 (或称弹出) 操作是先取出当前栈顶寄存器中的值,然后将 ST 加 1。注意,在 ST = 000 时,入栈操作将使 ST = 111,而 ST = 111 时,出栈操作将使 ST = 000。

位14,10~8

(C3 C2 C1 C0) 条件码,获得有关当前栈顶的附加信息,其结果形成某种条件分支

位7 (IR) 要求 80x86 中断的位。除 ST 和条件码外,当某事故发生时对应标志 位置为 1,如控制字中相应事故屏蔽位为 0,则中断申请位为 1。

位5 (PE) /* #define SW-INEXACT 0x0020 */

精度异常标志。如果结果必须四舍五入以便能够用浮点格式表示时,该标志置位

位4 (UE) /* #define SW-UNDERFLOW 0x0010 */

下溢出异常标志。当计算结果太小而不用非正常的方法就不能以指定的 目的操作数的浮点格式存储时,该标志置位

位3 (OE) /* #define SW-OVERFLOW 0x0008 */

上溢出异常标志。对计算结果太大时置位。

位2 (ZE) /* #define SW-ZERODIVIDE 0x0004 */

除数为 0 异常标志位

位1 (DE) /* #define SW-DENORMAL 0x0002 */

非正常 (不可规格化) 操作数标志位

位0 (IE) /* #define SW-INVALID 0x0001 */

非法操作异常标志位

三 控制字 (Contrl Word)

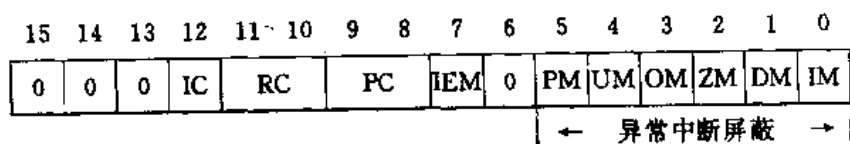


图 27-3

说明: 它是一个 unsigned int 数,各位功能是:

位15~13 Intel 公司保留,一般置为 0

位12 (IC) /* #define MCW-IC 0x1000 */

无穷大控制

```

/* #define IC—AFFINE      0x1000 */
确定无穷大算术运算类型:如 IC = 1,使用远交型。正无穷大和 负无穷大有不
同的编码表示 ( 称异值处理 )
/* #define IC—PROJECTIVE 0x0000 */
确定无穷大算术运算类型:如 IC = 0,使用投影型。没有正负无 穷大之分。初
始化时 IC = 0 ( 正无穷大和负无穷大作同值处理 )
位10,11 (RC) /* #define MCW—RC      0x0c00 */
四种舍入方向控制 ( 见指令介绍 )
/* #define RC—CHOP      0x0c00 */
载尾,趋向零舍入 (chop)
/* #define RC—UP        0x0800 */
向上 ( + ) 舍入 ( up,趋向正无穷大 )
/* #define RC—DOWN      0x0400 */
向下 ( - ) 舍入 ( down,趋向负无穷大 )
/* #define RC—NEAR      0x0000 */
无偏向的舍入到最近的 ( near,指+或-方向 ) 数,若一样接近,则选择偶数
值
位9,8 (PC) /* #define MCW—PC      0x0300 */
精度控制
/* #define FLT—MANT—DIG 24 */
/* #define PC—24      0x0000 */
精度控制:短实数 ( 24 位结果 )
/* 保留 ,0x0100 */
/* #define DBL—MANT—DIG 53 */
/* #define PC—53      0x0200 */
精度控制:长实数 ( 53 位结果 )
/* #define LDBL—MANT—DIG 64 */
/* #define PC—64      0x0300 */
精度控制:长实数 ( 64 位结果 )
/* #define MCW—EM      0x003f */
设置控制字为 0x03ff,以清除异常标志和忙中断,并使所有堆栈单元变为空
中断允许屏蔽。Intel 公司保留,一般为 0
位7 (IEM) Intel 公司保留,一般为 0
位6 Intel 公司保留,一般为 0
位5 (PM) /* #define EM—INEXACT 0x0020 */
精度屏蔽
位4 (UM) /* #define EM—UNDERFLOW 0x0010 */
下溢屏蔽
位3 (OM) /* #define EM—OVERFLOW 0x0008 */
上溢屏蔽
位2 (ZM) /* #define EM—ZERODIVIDE 0x0004 */
除数为 0 屏蔽
位1 (DM) /* #define EM—DENORMAL 0x0002 */
非规格化操作数屏蔽
位0 (IM) /* #define EM—INVALID 0x0001 */
无效的操作屏蔽
/* #define CW—DEFAULT (RC—NEAR+PC—64+IC—AFFINE+EM—UNDER-

```

FLOW+EM-INEXACT) */ 初始化控制字,即缺省控制字值,它是一些条件组合。

四 特征字 (Tag Word)

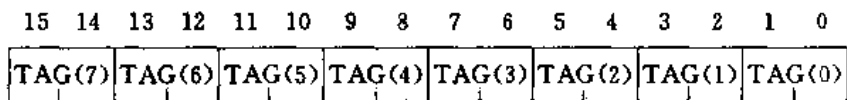


图 27-4

它描述了 8 个堆栈单元内容的各种特征,每 2 位对应一个堆栈单元,16 位对应 8 个堆栈单元。每两位特征值的含义是:

- 00 = 单元内容为有效值 (规格化的或非规格化的)
- 01 = 单元内容为零
- 10 = 单元内容为特殊值 (不是一个数字,或者为无穷大,非规格化的)
- 11 = 单元为空

27.2 数据类型

一 二进制整数

1. 字整数 $-32768 \leq X \leq 32767$ (位数:16,十进制有效位:4)
2. 短整数 $-2000000000 \leq X \leq 2000000000$ (位数:32,十进制有效位:9)
3. 长整数 $-9000000000000000000 \leq X \leq 9000000000000000000$
(位数:64,十进制有效位:19)

二 组合的十进制数

$-999999999999999999 \leq X \leq 999999999999999999$
(位数:80,十进制有效位:18)

三 短、长和暂存实数

1. 短实数 $0, 1.2 \times 10$ 的负 38 次方 $\leq X \leq 3.4 \times 10$ 的 38 次方
(位数:32,十进制有效位:6)

```
/* #define FLT-DIG      6          */
有效位数为 6
/* #define FLT-EPSILON  1.19209290E-07F */
/* #define FLT-MAX      -huge-flt      */
/* #define FLT-MIN      1.17549435E-38F */
/* #define FLT-MAX-10-EXP +38 */
/* #define FLT-MIN-10-EXP -37 */
/* #define FLT-MAX-EXP   +128 */
基值指数:127 (0x7f)
/* #define FLT-MIN-EXP   -125 */
```

2. 长实数 $0, 2.3 \times 10$ 的负 308 次方 $\leq X \leq 1.7 \times 10$ 的 308 次方
(位数:64,十进制有效位:15)

```
/* #define DBL-DIG      15          */
```


有效位数为 15

```
/* #define DBL-EPSILON 2.2204460492503131E-16 */
/* #define DBL-MAX      -huge-dble */
/* #define DBL-MIN      2.2250738585072014E-308 */
/* #define DBL-MAX-10-EXP +308 */
/* #define DBL-MIN-10-EXP -307 */
/* #define DBL-MAX-EXP   +1024 */
```

基值指数:1023 (0x3ff)

```
/* #define DBL-MIN-EXP   -1021 */
```

3. 暂存实数(临时实数) 0, 3.4×10 的负 4932 次方 $\leq X \leq 1.1 \times 10$ 的 4932 次方
(位数:80,十进制有效位:19)

```
/* #define LDBL-DIG      19 */
有效位数为 19
/* #define LDBL-EPSILON 1.084202172485504E-19 */
/* #define LDBL-MAX      -huge-ldble */
/* #define LDBL-MIN      -tiny-ldble */
/* #define LDBL-MAX-10-EXP +4932 */
/* #define LDBL-MIN-10-EXP -4931 */
/* #define LDBL-MAX-EXP   +16384 */
```

基值指数:16383 (0x3fff)

```
/* #define LDBL-MIN-EXP   -16381 */
```

四 特殊值

非正常数、非正规数、无穷大数、正或负的非数值数据、+和一无穷大表示及符号零

27.3 80x87 指令简要说明

ST 规定是 80x87 的 TOS(堆栈顶)。ST(n) 表示栈中从 ST 算起的第 n ($0 \leq n \leq 7$) 个寄存器,或者说,它表示 ST 下面的第 n 个寄存器,所以 ST(0) 即 ST。注意:根据 ST 的值决定现在栈顶寄存器是哪一个。对实数加法指令

FADD ST,ST(2)

若 ST = 011B,表示 R3 是栈顶,则指令就表示 R3 和 R5 两寄存器的内容相加。若执行该指令时 ST = 6,则表示 R6 与 R0 的内容相加。TOP 指 80x87 状态字中的堆栈指针。

注意:每一个指令助记符总是用字母 F(fast 之意)开始的,每当 CPU 遇到 F 开头的助记符时,就把控制转移到协处理器去执行。

注:如果在使用浮点运算和子程序中使用嵌入汇编(当 TCC.EXE 使用 -f 选项时),则不能使用序号前带有 * 的助记符。

· 把数值加载到寄存器堆栈

FBLD、FILD、FLD

· 将堆栈最高位寄存器中的数据写入到存储器或寄存器堆栈的另外寄存器

FBSTP、FIST、FISTP、FST、FSTP

· 算术运算

加法 FADD、FADDP、FIADD

减法 FISUB、FSUB、FSUBP

乘法 FIMUL、FMUL、FMULP

除法 FDIV、FIDIV、FIDIVP

· 函数

基本函数 FABS、FCHS、FPREM、FRNDINT、FSCALE、FSQRT、FXTRACT

超越函数 F2XM1、FPATAN、FPTAN、FYL2X、FYL2XP1

· 常数

FLDT、FLDL2E、FLDL2T、FLDLG2、FLDLN2、FLDPI、FLDZ

· 其它

1. F2XM1

计算 2 的 X 次方减 1 的值, X 的值从 ST 中获得, 其结果替换了原有的 ST 的值。

X 值必须是 $0 \leq X \leq 5$ 。

2. FABS

取出 80x87 堆栈顶单元的内容, 然后用其绝对值替换原内容。

3. FADD

(1) FADD 目的操作数, 源操作数

允许堆栈中任何寄存器作为两个操作数之一, 第二个操作数存在堆栈顶。指令使两数相加后存入目的操作数中。

(2) FADD 源操作数

源操作数可以是存在存储器中的实数, 隐含目的数是堆栈顶寄存器中的内容。

(3) FADD

把堆栈顶 ST 的内容加上堆栈中第二单元 ST(1) 的内容返回到新的堆栈顶。

这三个指令是完成实数相加。

4. FADDP 目的操作数, 源操作数

源操作数是当前堆栈顶, 目的数来自堆栈中其它任何一个寄存器。两数相加的和数返回到目的操作数。堆栈被弹出, 即实数相加并弹出。

5. FBLD 源操作数

假定源操作数是有效的 BCD 数 0 ~ 9, 则指令把组合的 BCD 数转换为暂存实数格式, 并把结果压入 80x87 堆栈, 即装入组合的十进制 (BCD) 数。

6. FBSTP 目的操作数

把堆栈顶保存的数值转换为组合十进制 (BCD) 整数格式, 将结果存储在目的操作数中, 并将 80x87 堆栈弹出。

7. FCHS

取堆栈顶的数值并改变它的符号。

8. FCLEX

清除所有异常标志、中断请求标志和状态字中的忙标志。在 80287 和 80387 中还清除错误状态标志。

9. FCOM

(1) FCOM 源操作数

实数比较。将堆栈顶当前内容与源操作数 (堆栈中寄存器或长、短实数的寄存器操作数) 比较。80x87 状态字中的条件码根据下列比较结果而变化:

	C3	C2	C0 (对 C1 无影响)
ST > 源操作数	0	0	0
ST = 源操作数	1	0	0
ST < 源操作数	0	0	1
ST 与源操作数不可比较	1	1	1

(2)FCOM

源操作数隐含为 ST(1)。

10. FCOMP

它和 FCOM 类似,只是最后弹出 80x87 堆栈。

11. FCOMPP

它和 FCOM 类似,比较实数并弹出两次。

12. FCOS

计算一个角度的余弦。角度取自 ST,而且没有范围限制。计算完成时,余弦值替换堆栈顶中的 0。

* 13. FDECSTP

状态字中的 TOP 减 1,不改变特征字和寄存器。

14. FDISI

通过设置控制字中的中断屏蔽位而防止 80x87 发出中断请求,即禁止中断。

15. FDIV

(1)FDIV 目的操作数,源操作数

源操作数除目的操作数,商存到目的操作数。

(2)FDIV 源操作数

目的操作数隐含为 ST(1)。

(3)FDIV

目的操作数隐含为 ST(1),源操作数为 ST。

16. FDIVP 目的操作数,源操作数

目的操作数为 ST(n),源操作数为 ST。取出两数后把 TOS(堆栈顶)弹出,再将源除目的数,商返回到 ST(n)寄存器。

17. FDIVR

(1)FDIVR 目的操作数,源操作数

和 FDIVP 不同的是将目的数除源操作数。

(2)FDIVR 源操作数

目的操作数隐含为 ST(1)。

(3)FDIVR

目的操作数隐含为 ST(1),源操作数为 ST。

18. FDIVRP 目的操作数,源操作数

目的操作数总是 ST(n),源操作数总是 ST 寄存器,ST(n)除 ST,商存入目的操作数,并将 80x87 堆栈弹出。

19. FENI

清除控制字中的中断屏蔽位,允许微处理器产生中断请求。

* 20. FFREE 目的操作数

改变目的寄存器的 TAG(特征字)为空,不改变寄存器的内容而释放目的寄存器。

21. FIADD 源操作数

目的操作数总是 ST。源和目的相加后存入目的寄存器。

22. FICOM 源操作数

整数比较。将源操作数先转换为暂存实数并与 ST 中值比较。

80x87 状态字中的条件码根据下列比较结果而变化：

	C3	C2	C0 (对 C1 无影响)
ST > 源操作数	0	0	0
ST = 源操作数	1	0	0
ST < 源操作数	0	0	1
ST 与源操作数不可比较	1	1	1

23. FICOMP 源操作数

与 FICOM 相同,不过在比较后弹出 TOS。

24. FIDIV 源操作数

整数除法。目的操作数是 ST,源除目的后商存入目的操作数。

25. FIDIVR 源操作数

与 FIDIV 相似,只是以目的除源后商存入目的操作数。

26. FILD 源操作数

把源操作数从二进制整数格式改为暂存实数格式后压入 ST。

27. FIMUL 源操作数

整数乘法。源乘 ST 中数后积存入 ST。

* 28. FINCSTP

TOP 加 1,不改变特征字和寄存器。

29. FINIT

初始化协处理器:设置控制字为 03FFH,清除异常标志和忙中断,并使所有浮点堆栈单元变空。实际这同硬件复位。把它放在开头保证用户程序不受前面程序的寄存器值的影响。把它放在结尾保证用户程序的寄存器值不影响后面的程序。

30. FIST 目的操作数

整数存储。取 ST,根据控制字的 RC 场对其进行四舍五入,结果返回目的操作数。

31. FISTP 目的操作数

和 FIST 类似,只是最后将弹出 TOS。

32. FISUB 源操作数

整数减法。ST 减源操作数,差返回 ST。

33. FISUBR 源操作数

颠倒的整数减法。源操作数减 ST,差返回 ST。

34. FLD 源操作数

装入实数。把 TOP 减 1,并把源操作数复制到 ST 中。

35. FLD1

用暂存实数 64 位精度和 19 个十进制精度把常数 +1.0 压入 80x87 堆栈。

36. FLDCW 源操作数

装入控制字。用源操作数替换控制字。

37. FLDENV 源操作数

由源操作数定义的存储区装入 80x87 的环境。

38. FLDDL2E

以暂存实数 64 位精度和 19 个十进制精度把以 2 为底 e 的对数值压入 80x87 浮点堆栈顶部。

39. FLDDL2T

以暂存实数 64 位精度和 19 个十进制精度把以 2 为底 10 的对数值压入 80x87 浮点堆栈顶部。

40. FLDLG2

以暂存实数 64 位精度和 19 个十进制精度把以 10 为底 2 的对数值压入 80x87 浮点堆栈顶部。

41. FLDLN2

以暂存实数 64 位精度和 19 个十进制精度把以 e 为底 2 的对数值压入 80x87 浮点堆栈顶部。

42. FLDPI

以暂存实数 64 位精度和 19 个十进制精度把常数 $PI(3.1415\dots)$ 压入 80x87 浮点堆栈顶部。

43. FLDZ

以暂存实数 64 位精度和 19 个十进制精度把常数 $+0.0$ 压入 80x87 浮点堆栈顶部。

44. FMUL

(1)FMUL 目的操作数,源操作数

实数乘法。两者相乘的积乘入目的操作数。

(2)FMUL 源操作

目的操作数隐含为 ST。

(3)FMUL

目的操作数 ST,源操作数 ST(1)。

45. FMULP 目的操作数,源操作

与 FMUL 类似,只是目的操作数是 ST(n),存储并弹出。

46. FNCLEX

清除所有异常标志、忙标志和状态字中的中断请求标志。

47. FNDISI

设置控制字中的中断屏蔽位,从而禁止微处理器发出中断请求。

48. FNENI

清除控制字中的中断屏蔽位,从而允许微处理器产生中断请求。

49. FNINIT

同 FINIT,复位协处理器。

50. FNOP

空操作。

51. FNSAVE 目的操作数

把完整的 80x87 环境,包括寄存器堆栈保存在由目的操作数指定的存储器单元中。80x87 接着进行初始化。

52. FNSTENV 目的操作数

把当前环境、异常指针、特征字、控制字和状态字存储在由目的操作数指定的存储器单元中。

53. FNSTSW 目的操作数

把状态字的内容存入目的操作数指定的存储器单元中。

54. FPATAN

从当前的 ST 中弹出 X 值 (此弹出后, ST(1) 变成 ST), 接着第二次又弹出堆栈, 取出这时的 ST 作为 Y 值, 然后将反正切值 $\text{ARCTAN}(Y/X)$ 存入 ST。X 和 Y 的正确值为正数, Y 必须小于 X。

55. FPREM

取 ST 当前内容, 并用下一个单元 ST(1) 的内容除, 然后将部分余数替换 ST。

56. FPTAN

计算 ST 的正切值。ST 中的值相当于一个角度 ($0 \leq \text{角度} < \pi/4$), 结果得到比值 $\tan(\text{角度}) = Y/X$ 。完成计算时, Y 替换堆栈中的角度, 并压入 X, 使之成为新的堆栈顶。指令执行后, X 保留在 ST, Y 保留在 ST(1)。超出范围的值并无标志。该值可用于 SIN、COS 等计算。

57. FRNDINT

将 ST 内容四舍五入成整数:

控制字的 RC = 00 舍入成最接近的整数。如果数值完全是中点, 就选择偶数值。

RC = 01 舍去

RC = 10 进入

RC = 11 向 0 舍入

58. FRSTOR 源操作数

以源操作数作为存储器区的基准, 取出 94 个字节以便恢复 80x87 的状态。为了兼容, 这些数据应当是曾用 FSAVE 或 FNSAVE 保存的。

59. FSAVE 目的操作数

同 FNSAVE。

60. FSCALE

定标换算, 即得 $ST * (2 \text{ 的 } ST(1) \text{ 次方})$ 。

61. FSETPM

设置 80x87 保护方式, 直到下一次硬件复位。

62. FSIN

将堆栈顶中的内容作为角度, 并把求得的其正弦值替换堆栈顶中的角度。

63. FSINCOS

将堆栈顶中的内容作为角度, 并把求得的其正弦值放在 ST(1) 中, 余弦值放在 ST 中。ST(1) 的先前值在 ST(2) 中。

64. FSQRT

把 ST 的平方根替换 ST 的值。

65. FST 目的操作数

将 ST 中的值保存在由目的操作数指定的寄存器单元中。

66. FSTCW 目的操作数

将当前控制字保存在由目的操作数指定的寄存器单元中。

67. FSTENV 目的操作数

将当前环境, 包括状态、控制、特征字和异常指针保存在由目的操作数指定的寄存器单元中。

68. FSTP 目的操作数

将 ST 中的值保存在由目的操作数指定的寄存器单元中,然后将堆栈弹出。

69. FSTSW 目的操作数

将当前状态字保存在由目的操作数指定的寄存器单元中。

70. FSUB

(1)FSUB 目的操作数,源操作数

目的减源的差存入目的中。

(2)FSUB 源操作数

目的操作数隐含为 ST。

(3)FSUB

目的操作数为 ST,源操作数为 ST(1)。

71. FSUBP

与 FSUB 类同,只是结果浮点堆栈被弹出。

72. FSUBR 目的操作数,源操作数

与 FSUB 类同,只是颠倒的实数减法,即差等于源减目的。

73. FSUBRP 目的操作数,源操作数

与 FSUBP 类同,只是颠倒的实数减法,即差等于源减目的。

74. FTST

ST 与 +0.0 比较,改变状态字条件码:

		C3	C2	C0 (对 C1 无影响)
ST > 0	ST 是正数	0	0	0
ST = 0	ST 是 0	1	0	0
ST < 0	ST 是负数	0	0	1
ST 与 +0.0 不可比较,即NAN		1	1	1

75. FWAIT

80286 等待 80x87 完成操作,当操作完后 80286 才执行下一条指令。

76. FXAM

检查 ST 的内容,并根据下列结果设置状态字中的条件码:

C0	C1	C2	C3	结果
0	0	0	0	非规格化正数
0	0	0	1	正的非数值数据
0	0	1	0	非规格化负数
0	0	1	1	负的非数值数据
0	1	0	0	正的规格化数
0	1	0	1	正无穷大
0	1	1	0	负规格化数
0	1	1	1	负无穷大
1	0	0	0	+0
1	0	0	1	空
1	0	1	0	-0
1	0	1	1	空
1	1	0	0	不可规格化正数
1	1	0	1	空
1	1	1	0	不可规格化负数

77. FXCH

(1)FXCH 目的操作数

目的操作数与 ST 内容交换。

(2)FXCH

ST 与 ST(1) 的内容交换。

78. FXTRACT

抽取指数和有效数字。将 ST 中的数字变成以实数表示的有效数字和指数,用指数替换原始的 TOS,然后把有效数字压入 ST。

79. FYL2X

从 ST 中取出 X(必须是正数),从 ST(1)中取出 Y,指令接着弹出堆栈并将 $Y * (以 2 为底、X 的对数)$ 存入到 ST 中。

80. FYL2XP1

从 ST 中取出 X,从 ST(1)中取出 Y,指令接着弹出堆栈并将 $Y * (以 2 为底、X+1 的对数)$ 存入到 ST 中。X 的绝对值必须在 0 到 2 的平方根除以 2 之间。当求一个非常接近于 1 的对数时使用这条指令。

27.4 80x87 函数

为了使用 80x87 函数,你的机器上应装有 80x87。使用 80x87 函数的程序可能不能在没装 80x87 的机器上运行,这是要注意的。为此,你可以使用 biosequi() 函数返回值的位 1 来检查机器上是否装有 80x87,然后用程序分支来适应不同情况。

```
—1 unsigned int —Cdecl —clear87(void);
```

清除浮点状态字,该字是 8087/80287 状态字和由 8087/80287 异常处理程序检测到的其它条件的组合。它清除异常标志、中断请求和忙标志,返回原状态字低 9 位的内容。

```
C>TYPE TC80871.C
```

```
#include "float.h"
```

```
main()
```

```
{
```

```
int status;
```

```
double x=1.5,y=.8,z;
```

```
status=—clear87();
```

```
printf("status=%x\n",status);
```

```
z=x*y;
```

```
printf("z=%f\n",z);
```

```
status=—clear87();
```

```
printf("status=%x\n",status);
```

```
}
```

```
/* 在装有80387的机上,且在集成环境下,选
```

```
Options/Compiler/Code generation/Floating point/8087/80287
```

```
或
```

```
Options/Compiler/Code generation/Floating point/Emulation
```

```
则程序输出:status=0
```



```
z=1.200000
```

```
status=20
```

注意:1. 如选

Options/Compiler/Code generation/Floating point/None

编译时则出现连接错误:变量 ——clear87、FIWRQQ 和 FIDRQQ 符号未定义。

2. 选 80287 编译时执行码为 13928 字节,而选 Emulation 编译时为 23835 字节。

3. 在没有装 80x87 的机上,只能选 Emulation,而选 8087/80287 编译将得 错误的结果,例如,上述程序输出

```
status=3f
```

```
z=0.000400000200005000000000000000000000000000e+4933
```

```
status=3f
```

而只有在没有浮点运算,也没有用到涉及 80x87 函数时才可选 None。

*/

—2 void —Cdecl —fpreset(void);

它对浮点数学包重新初始化。在执行 exec...、spaw...、system 或 longjmp、signal 等函数后,由于子进程中可能使用了浮点操作运算,而这样就有可能改变父进程中的浮点状态(在 DOS 3.X 之前的版本中,如果使用了 8087/80287 则会发生这种情况)。因此,为了恢复父进程原浮点状态,就必须使用本函数复位浮点状态。

本函数并不返回值。

注意:如果使用了 8087/80287,在计算一个浮点表达式时不得调用 system、exec... 或 spaw... 等函数。

```
C>TYPE TC80872.C
```

```
#include "float.h"
```

```
#include "stdio.h"
```

```
main()
```

```
{
```

```
double x=.2,y; /* 不能定义为 int x=.2,y; 那样编译后执行时将出现: */
```

```
y=x*9; /* Printf:floating point formats not linked 错误 */
```

```
printf("y=%f\n",y); /* 从而得不到正确结果,虽然你包括了 float.h 文件 */
```

```
}
```

```
C>TYPE TC80873.C
```

```
#include "float.h"
```

```
#include "process.h"
```

```
#include "stdio.h"
```

```
#define PS status=—clear87();\
```

```
printf("status=%0x\n",status)
```

```
main()
```

```
{
```

```
int status,k;
```

```
double x=1.5,y=.8,z;
```

```
for(k=1;k<3;k++)
```

```
{
```

```
z=x*y*k;
```

```
printf("z=%f\n",z);
```

```

spawnl(P-WAIT,"e.exe",NULL);
/* system("dir/w w.c"); */
/* _fpreset(); */
}
PS,
}
/* 在 DOS 5.0、使用了 80387 环境下,程序中无 _fpreset(); 语句时输出:
z=1.200000
y=1.800000
z=2.400000
y=1.800000
status=20
程序中有 _fpreset(); 语句时输出:
z=1.200000
y=1.800000
z=2.400000
y=1.800000
status=0
*/

```

—3 unsigned int —Cdecl —status87(void);

返回当前的浮点状态,它是 8087/80287 状态字和其它由 8087/80287 例外情况处理程序检测到的条件组合(实际返回当前状态字的低 7 位值,即当前状态字值和 0x3f 相与运算的结果,由位值反映出来)。

```

C>TYPE TC80874.C
#include "float.h"
#include "stdio.h"
#define PS status=_status87();\
printf("status=%x\n",status)
main()
{
int status;
double x=1.5,y=.8,z;
PS,
z=x*y;
printf("z=%f\n",z);
PS,
}
/* 程序输出,status=0
z=1.200000
status=20
*/

```

—4 unsigned int —Cdecl —control87(unsigned int new,unsigned int mask);

函数返回浮点控制字的状态。

本函数包括两个过程:改变当前浮点控制字,返回当前控制字被改变之前的值,即原当前控制字的值。mask 是屏蔽字,可分两种情况,一是当 mask 非零时,控制字根据 mask 与 new

的对应位值来改变当前控制字的值;而当 mask 为零时,当前控制字不会被改变。其改变过程参见下面的程序的输出说明。

```
C>TYPE TC80875.C
#include "float.h"
#include "stdio.h"
#include "math.h"
#define PC status=--status87();\
printf("% #x\n",status)
#define PS printf("% #x\n",con)
#define PCS PC;con=--control87(0xe905,0x814f);\
PS;PC;con=--control87(0xe9f5,0xffff);\
PS;PC;con=--control87(0xe9f5,0x0);\
PS;PC

main()
{
int status,con;
double x=3.1416*3,z=18;
PC;
z=sqrt(-x);
printf("z=%f\n",z);
PCS;
z=x/0.9;
printf("z=%f\n",z);
PCS;
}
/* 程序输出:
0
sqrt,DOMAIN error 负数不能开方
z=0.000000 值得怀疑的值
0
0x1330 返回缺省控制字 CW—DEFAULT
0
0x1335 返回上次设置的控制字,数据是
原控制字 = 0x1330=0001 0011 0011 0011
new = 0xe905=1110 1001 0000 0101
mask = 0x814f=1000 0001 0100 1111
新控制字 = 0x1335=0001 0011 0011 0101
生成过程是,先看 mask 位 7,如为 1,则将new 的位 7 的值代替原控制字的位 7 的值;如 mask 位 7 为 0,则原控制字的位 7 的值不变。然后用同样的方法处理位 6~0。要注意的一点是,由于控制字的位 15、14、13 和位 7、位 6 为 Intel 公司保留,且始终自动处理为 0 值,因此上述处理对这几位无效。亦即不管怎样处理,这几位总为 0。
0
0x935 返回上次设置的控制字
0
z=10.472000
```

```

0x20
0x935
0x20
0x935
0x20
0x935
0x20
*/

```

27.5 其它一些说明

-- 外部变量和符号常量

在 float.h 中定义了一些外部变量：

```

extern float  —Cdecl —huge—flt;
extern double —Cdecl —huge—dble;
extern long double —Cdecl —huge—ldble;
extern long double, —Cdecl —tiny—ldble;

```

另外,还定义了一些符号常量

```

#define FLT—RADIX      2  数进制的基数为 2
#define FLT—ROUNDS     1  浮点舍入
#define FLT—GUARD      1  浮点保护位
#define FLT—NORMALIZE  1  浮点规格化数

```

C>TYPE TC80876.C

```

#include "float.h"
#include "stdio.h"
main()
{
double x=.2,y;
y=x*9;
printf("y=%f\n",y);
printf("%f\n",—huge—flt);
printf("%f\n",—huge—dble);
printf("%e\n",—huge—ldble);
printf("%e\n",—tiny—ldble);
printf("%f\n",FLT—RADIX);
printf("%f\n",FLT—ROUNDS);
! printf("%f\n",FLT—GUARD);
printf("%f\n",FLT—NORMALIZE);
printf("%d\n",FLT—RADIX);      /* 这是正确的输出格式 */
printf("%d\n",FLT—ROUNDS);
printf("%d\n",FLT—GUARD);
printf("%d\n",FLT—NORMALIZE);
}

```

```

/* 程序输出: y=1.800000
340282346638528860000000000000000000000.0
1.79769313486231571000000000000000000000e+308
-NAN          负的非数值数据
-0.00000e+00
-0.000000    注意如用 %f 格式输出,结果就错误
0.000000
0.000000
0.000000
2          如改用 %e 格式输出,就有 2.35344e-185
1          2.35344e-185
1          2.35344e-185
1          2.35344e-185
*/

```

浮点错误 (FPE, Floating point error) SIGFPE 类型 (对整型和浮点例外情形) raise() 函数在 signal.h 中有定义, 下面这些符号常量可用于条件判断。

```

#define FPE—INTOVFLOW 126 /* 在溢出时 80x86 中断 */
#define FPE—INTDIV0 127 /* 80x86 整数除 0 */
#define FPE—INVALID 129 /* 80x87 的无效操作 */
#define FPE—ZERODIVIDE 31 /* 80x87 数除 0 */
#define FPE—OVERFLOW 132 /* 80x87 运算溢出 */
#define FPE—UNDERFLOW 133 /* 80x87 运算下溢 */
#define FPE—INEXACT 134 /* 80x87 精度丢失 */
#define FPE—EXPLICITGEN 140 /* 当用 raise(SIGFPE) 说明时 */

```

SIGSEGV 错误类型

```

#define SEGV—BOUND 10 /* 边界违反 */
#define SEGV—EXPLICITGEN 11 /* 当用 raise(SIGSEGV) 说明时 */

```

SIGILL 错误类型

```

#define ILL—EXECUTION 20 /* 非法操作说明 */
#define ILL—EXPLICITGEN 21 /* 当用 raise(SIGILL) 说明时 */

```

二 几点说明

1. 使用浮点数

在集成环境下, Turbo C 能根据源程序中是否使用了浮点数而作出正确的选择选择项和库。当然, 这时你不能将 Floating point 开关选择为 None。但在用 TCC 独立编译时可输入 (假定源程序文件名为 my.c):

```
tcc -mX my
```

则 Turbo C 自动产生 my.obj 文件和执行文件 my.exe。这里 X 表示选择的存储模式的第一个字母。如 -ms 表示选择 small 存储模式。如果要产生 80x87 芯片的代码, 可用

```
tcc -f87 -mX my
```

如果读者要将多个 .OBJ 文件连接起来以产生代码, 则必须指定依赖于存储模式的正确

的数学库和仿真库 EMU.LIB。除非 TURBOC.CFG 文件含有其它浮点开关（-f 或 -f87），否则 -f 仿真选择项的缺省值为 ON。如果使用缺省的浮点仿真，TLINK 将按如下方式调用：

```
tlink c0X my,my,my,emu.lib mathX.lib cX.lib
```

其中字母 X 仍表示存储模式的首字母。注意：要严格按照规定书写，例如如漏了第三个逗号而改用空格，则可能毁了原有的正确的 emu.lib 库！另外，除了 tlink 命令行必须在一行中给出外，库的排列顺序也很重要！如果使用与 80x87 相关的库 fp87.lib，TLINK 将按如下方式调用：

```
tlink c0X my,my,my,fp87.lib mathX.lib cX.lib
```

2. 不使用浮点数

在集成环境下，如源程序中并未涉及浮点数，则可置 Floating point 为 None（选 None 后按三次 ESC 键或按回车键返回到主菜单）。用 TCC 时应使用 -f- 选择项，如

```
tcc -f- -mX my
```

由于这样将不连接任何浮点数、数学库子程序，从而加快连接速度。在使用 TLINK 时如使用了 /c 选择项，用户必须自己连接目标代码，但不必指定数学库。

3. 80x87 环境变量的设置

如果用 80x87 仿真子程序连接程序（即选择 Floating point 为 Emulation 或在 TCC 命令行中使用 -f 选择项），程序运行时自动检测模块 c0X.OBJ 将自动检测系统是否有 80x87 芯片存在。如有则使用它，否则运行时使用仿真子程序。

事实上，在运行程序启动时，启动模块会先去查看 87 环境变量，因此，通过重新设置 87 环境变量可以忽略缺省的自动检测 80x87 存在。

如在 DOS 提示符下，为设置 87 环境变量为 N 可键入

```
C>SET 87=N
```

则启动模块 c0X.OBJ 不使用 80x87，或者说即使有也不会使用它。而

```
C>SET 87=Y
```

则使用 80x87，但要注意，如果程序运行时没有 80x87 存在，将出现致命性错误，或者说程序将不能运行。

如想解除 87 环境变量的定义，可在 DOS 提示符下键入

```
C>SET 87=
```

在键入等号后直接回车。此时启动代码在检测 80x87 后作出相应的运行调整。

4. 80x87 和寄存器变量

进行浮点运算时如使用寄存器变量时应注意：

在 80x87 仿真方式下，不支持寄存器变量（warparound）。

如在嵌入汇编中使用了浮点数，则在使用寄存器时要特别小心。因为 Turbo C 调用函数前要清除 80x87 寄存器，除非确信有足够的空闲寄存器，否则在使用协处理器的函数前必须对 80x87 寄存器进行保护。

第二十八章 日期和时间函数

28.1 概述

1. 时区和地方时

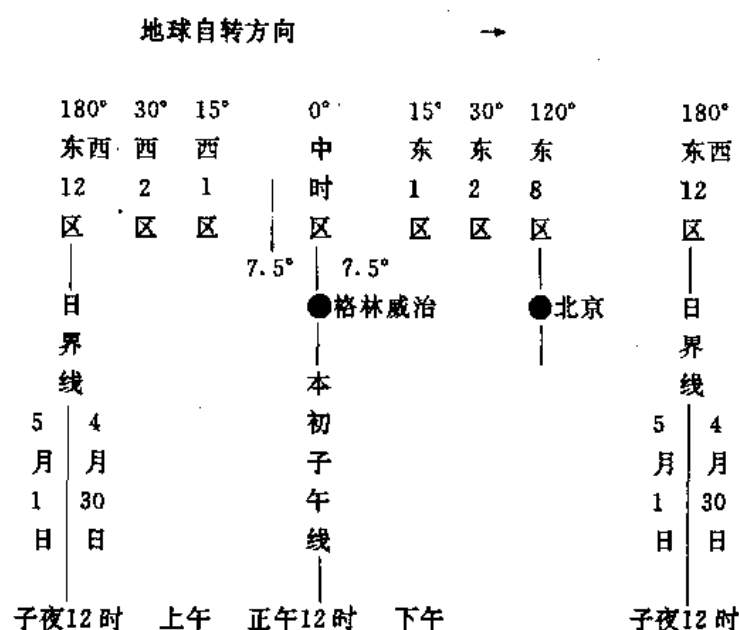


图 28-1 二十四个小时划分示意图

规定每隔经度 15° (地球 1 小时转过经度 15°) 划分一个【时区】，全球按经度划分为 24 个时区。相邻两个时区的区时相差 1 个小时。过日界线 (国际日期变更线) 要改变日期。

在地球不同经度处的时刻称【地方时】。北京处在东八区， 120° 是东八区的中央经线，因此“北京时间”是采用东经 120° 的地方时，即东八区的区时。

“格林威治 (中央) 时间” (GMT, Greenwich Mean Time) 是采用的中时区 (经度为 0°) 的地方时。

2. 【DOS 日期与时间】

DOS 日期计数是从 1979 年 12 月 31 日的午夜开始算起的日期。例如，1980 年 1 月 1 日算 1 日。DOS 有效日期范围是 1980 年 1 月 1 日到 2099 年 12 月 31 日。DOS 日期的一般用年、月、日表示，其具体的格式与分隔符同国家有关 (参见函数 country())。DOS 的时间用小时、分、秒表示，具体的格式与分隔符也同国家相关。

3. 【UNIX 时间】

它用秒作单位，时间计数是从 (GMT) 1970 年 1 月 1 日 0 时 0 分 0 秒算至当前已逝去的秒数。

28.2 库函数

一 规定

- 返回与国家有关的日期、时间或货币等信息 —1 country()

二 系统日期与时间

- 取系统当前 DOS 日期并填入 date 类结构中 —2 getdate()
- 取系统当前 DOS 时间并填入 time 类结构中 —3 gettime()
- 利用 date 类结构设置 DOS 日期 (会改变 CMOS 中的值) —4 setdate()
- 利用 time 类结构设置 DOS 时间 (会改变 CMOS 中的值) —5 settime()
- 利用 UNIX 时间设置系统日期与时间 —6 stime()

三 UNIX 时间

- 取当前 UNIX 时间 —7 time()
- 取当前 UNIX 时间并放到 timeb 类结构中 —8 ftime()

四 日期及时间的转换

- 将 DOS 日期与时间转换成 UNIX 时间 —9 dostounix()
- 将 UNIX 时间转换成 DOS 日期与时间 —10 unixtodos()
- 把 tm 类结构中的时间信息转换成字符串 —11 asctime()
- 把 UNIX 时间转换成字符串 —12 ctime()
- 把 UNIX 时间分解并存入 tm 类结构中 —13 gmtime()
- 把 UNIX 时间换成地方时并分解存入 tm 类结构中 —14 localtime()

五 计算时间

- 计算两个 UNIX 时间差 —15 difftime()
- 计算程序开始执行到函数调用时逝去的时钟滴答数 —16 clock()

六 延时

- 毫秒延时 —17 delay()
- 秒延时 —18 sleep()

七 磁盘文件的日期与时间

- 取文件的 DOS 日期与时间 —19 getftime()
- 修改文件的 DOS 日期与时间 —20 setftime()

八 其它

- 与 UNIX 兼容函数 —21 tzset()
- 读取或设置 BIOS 实时时钟值 —22 boistime()

—1 struct country * —Cdecl country (int xcode, struct country * cp);

返回包括与国家代码 `xcode`(country code) 有关的日期、时间和货币情况的指针 `cp`。函数相当于 DOS 中断 INT 21H 功能调用 38H, 入口参数是, `AH=38H`, `AL=` 国家 (00H 为当前国家, 01H ~ FEH 设置 `xcode` 小于 255 的国家, FFH 设置 `xcode` 大于或等于 255 的国家), `BX=xcode`, `DS:DX → cp` (返回信息缓冲区)。只适用于 MS-DOS 3.0 或更高版本。结构类型 `country` 在 DOS.H 中定义为:

```
struct country {
int co—date;      日期格式 (date format):
                    0 表示美国 (USA) 的 (月—日—年)
                    1 为欧洲 (Europe) 的 (日—月—年)
                    2 为日本 (Japan) 的 (年—月—日)

char co—curr[5];   货币符号 (currency symbol)

char co—thsep[2];  千位分隔符 (thousands separators)

char co—dese[2];   小数分隔符 (decimal separators)

char co—dtsep[2];  日期分隔符 (date separators)

char co—tmsep[2];  时间分隔符 (time separators)

char co—currstyle; 货币类型 (currency style):
                    0 表示货币符号在值之前, 且符号和数间无空格
                    1 表示货币符号在值之后, 且符号和数间无空格
                    2 表示货币符号在值之前, 且符号和数间有一个空格
                    3 表示货币符号在值之后, 且符号和数间有一个空格

char co—digits;    货币中有效数位 (significant digits in currency)

char co—time;      时间格式 (time format) (=0 是 12 小时时钟, =1 为 24 小时时钟)

long co—case;      情况变换 (case map function), 大小写对照例程地址 (对远调用, AL = 要被转换
                    成大写的字符 (它大于或等于 80H))

char co—dasep[2];  数据分隔符 (data separators)

char co—fill[10];  进位填充数 (filler)
};
```

尽管你在获取信息后可重新对其赋值, 但这不会修改系统设置的值 (即 CMOS 中的值)。

C>TYPE DATE1.C

```
#include "stdio.h"
#include "dos.h"
main()
{
int i;
struct country cr;
for(i=0;i<3;i++)
{
country(i,&cr);
printf("date format=%d\n",cr.co—date);
printf("currency symbol=%s\n",cr.co—curr);
printf("thousands separators=%s\n",cr.co—thsep);
printf("decimal separators=%s\n",cr.co—dese);
printf("date separators=%s\n",cr.co—dtsep);
printf("time separators=%s\n",cr.co—tmsep);
printf("currency style=%c\n",cr.co—currstyle);
```

```

printf("significant digits in currency=%c\n",cr.co—digits);
printf("time format=%c\n",cr.co—time);
printf("case map function=0x%x\n",cr.co—case);
printf("data separators=%s\n",cr.co—dasep);
printf("filler=%s\n\n",cr.co—fill);
}
}
/* 三种输出都是:
date format=0
currency=$
thousands separators=,
decimal separators=.
date separators=-
time separators=:
currency style=(3D)    括号内数据是不可打印码,下同
significant digits in currency=(02)
time format=(3D)
case map function=0xcf5
data separators=,
filler=(30 5B E0 45 0E E8 90 FF CB 90 02)
*/

```

—2 void —Cdecl getdate (struct date * datep);

函数将系统当前日期填入由 datep 所指的 date 型结构中。它相当于 DOS 中断 INT 21H 的功能调用 2AH,入口参数是, AH=2AH。返回值在 CX(年)、DH(月)和 DL(日)中。它只适用于 MS-DOS,结构类型 date 在 dos.h 中定义为:

```

struct date {
    int   da—year;      /* 年 1980~2099 */
    char  da—day;       /* 日 1~31    */
    char  da—mon;       /* 月 1~12   */
};

```

C>TYPE DATE2.C

```

#include "dos.h"
main(){
    struct date d1,*today;
    getdate(&d1);
    printf("缺省日期:%d年%d月%d日\n",d1.da—year,d1.da—mon,d1.da—day);
    today—>da—year=1993;
    today—>da—day=8;
    today—>da—mon=10;
    setdate(today);
    getdate(&d1);
    printf("重置日期:%d年%d月%d日\n",d1.da—year,d1.da—mon,d1.da—day);
}
/* 输出:缺省日期:1993年6月6日
      重置日期:1993年10月8日

```

—3 void —Cdecl gettime (struct time *timep);

把系统当前时间填入由 timep 所指的 time 型结构中。它是靠调用 DOS 中断 INT 21H 的功能 2CH 实现的。入口参数是, AH=2CH。返回时 CH= 小时, CL= 分, DH= 秒, DL= 1/100 秒。注意,对大多数系统,系统时钟的分辨率约为 5/100 秒,所以返回的时间通常增量不为 1,某些系统可能总是返回 00H。它只适用于 MS-DOS。结构 time 在 DOS.H 中定义为:

```
struct time {
    unsigned char  ti—min; /* 分钟      (Minutes),0~59 */
    unsigned char  ti—hour; /* 小时      (Hours),0~23  */
    unsigned char  ti—hund; /* 百分之一秒 (Hundredths of seconds) */
    unsigned char  ti—sec; /* 秒        (Seconds),0~59  */
};
```

C>TYPE DATE3.C

```
#include "dos.h"
```

```
#define GP gettime(t1);\
```

```
printf("时间:%02d 小时%02d 分%02d. %02d 秒\n",\
    t1->ti—hour,t1->ti—min,t1->ti—sec,t1->ti—hund);
```

```
main(){
```

```
struct time *t1,now;
```

```
printf("缺省"),GP;
```

```
now.ti—hour=22; /* 注意,不能用像 now.ti—hour=t1->ti—hour+1, 这样 */
```

```
now.ti—min=3; /* 的语句,也不能用 now.ti—hour=before+1; 其中尽 */
```

```
now.ti—sec=59; /* 管变量 before 假定是早已定义且赋值的变量。 */
```

```
now.ti—hund=0;
```

```
settime(&now);
```

```
printf("重置"),GP;
```

```
}
```

```
/* 例如输出: 缺省时间:21 小时35 分48.84 秒
```

```
重置时间:22 小时03 分58.96 秒
```

```
稍有误差。
```

```
*/
```

—4 void —Cdecl setdate (struct date *datep);

设置 MS-DOS 日期,它是靠调用 MS-DOS 中断 INT 21H 的功能 2BH 实现的。入口参数是, AH=2BH, CX= 年 (1980 ~ 2099), DH= 月, DL= 日。返回 AL=00H 时表示设置成功; AL=FFH 表示无效日期,系统日期不变。注意,该日期设置会改变系统 CMOS 中的缺省值。只适用于 MS-DOS。

—5 void —Cdecl settime (struct time *timep);

设置系统为新的时间 (timep 所指 time 型结构中的时间)。它实际是 DOS 功能调用 2DH。入口参数是, AH=2DH, CH= 小时, CL= 分, DH= 秒, DL=1/100 秒。返回 AL=00H 时表示设置成功; AL=FFH 表示无效时间,系统时间不变。注意:该时间设置会改变系统 CMOS 中的缺省值。只适用于 MS-DOS。

—6 int —Cdecl stime(time—t *tp);

利用 UNIX 时间格式 (指针 tp 所指值) 设置系统日期和时间。类型 time—t 在 time.h

中定义为

```
typedef long time_t;

C>TYPE DATE4.C
#include "time.h"
#include "dos.h"
int stime(time_t *tp)          /* 等效函数 */
{
    struct date d;
    struct time t;
    unixtodos(*tp,&d,&t); /* 将 UNIX 时间转换为 DOS 日期和时间 */
    setdate(&d);          /* 设日期 */
    settime(&t);           /* 设时间 */
    return(0);            /* 返回 0 */
}

main()
{
    int tr;
    struct date d;
    struct time t;
    long s=740350851;
    tr=stime(&s);
    printf("tr=%d\n",tr);
    getdate(&d);
    gettime(&t);
    printf("%d 年 %u 月 %u 日 %u:%u:%u. %u\n",d.da-year,d.da-mon,d.da-day,
        t.ti-hour,t.ti-min,t.ti-sec,t.ti-hund);
}

/* 输出:
    tr=0
    1993 年 6 月 17 日 17:0:50.98
*/
```

—7 time_t —Cdecl time(time_t *timer);

取得当前的 UNIX 时间,并把它存在 timer 所指的地址中。适用于 UNIX 系统。

C>TYPE DATE5.C

```
/* #include <time.h>
#include <dos.h>
long time(long *dt)      等效函数
{
    struct date d;
    struct time t;
    long x;
    getdate(&d);
    gettime(&t);
    x=dostounix(&d,&t);
    if(dt) *dt=x;
    return (x);
}
```

```

    }
    /*
#include "time.h"
main()
{
time_t st,lt;
lt=time(&st);
printf("st=%ld lt=%ld\n",st,lt);
}
/*
st=740350851 lt=740350851
*/

```

—8 void —Cdecl ftime(struct timeb *);

函数把当前时间保存到指针所指的 timeb 型结构中。该结构内含从 1970 年 1 月 1 日以来的时间秒值和分离的毫秒值,同时也含地方时区和夏令时标志。结构 timeb 定义在 sys/timeb.h 中:

```

struct timeb{
    long time;        /* GMT 的秒数 */
    short millitm;    /* 秒的小数部分,milliseconds, 毫秒=千分之一秒 */
    short timezone;   /* 地方时间和 GMT 时间差 */
    short dstflag;    /* 如值为 0 表示昼间节省时间无效 */
};

```

C>TYPE DATE6.C

```

#include "stdio.h"
#include "sys/timeb.h"
#define PP ftime(&t);\
    printf("time=%ld millitm=%u timezone=%u dstflag=%u\n",\
        t.time,t.millitm,t.timezone,t.dstflag)

main()
{
    struct timeb t;
    PP;
    delay(5008);
    PP;
} /* 输出,从中可见在毫秒级有误差

```

```

    time=740408362 millitm=700 timezone=300 dstflag=1

```

```

    time=740408367 millitm=750 timezone=300 dstflag=1 */

```

—9 long —Cdecl dostounix (struct date *d,struct time *t);

将 DOS 日期和时间转换成 UNIX 时间。当前日期由 getdate() 获得并填入由 d 所指 date 类结构中;时间由 gettime() 获得并填入由 t 所指 time 类结构中。

只适用于 MS — DOS。

—10 void —Cdecl unixtodos (long time,struct date *d,struct time *t);

将第一个参数 time 给定的 UNIX 时间转换成 DOS 日期和时间。函数返回时把日期和时间分别填入由 d 和 t 所指的 date 与 time 类结构中。它只适用于 MS-DOS。

C>TYPE DATE7.C

```
#include "dos.h"
main()
{
    struct date d,a;
    struct time t,i;
    long udt;
    getdate(&d);          /* 这两句必须要有,否则输出像 -2041784836 */
    gettime(&t);
    printf("DOS date=%dyear %umonth %uday\n",d.da--year,d.da--mon,d.da--day);
    printf("DOS time=%uhour %uminute %usecond. %uhundredth\n",t.ti--hour,
            t.ti--min,t.ti--sec,t.ti--hund);

    udt=dostounix(&d,&t);
    printf("UNIX=%ld\n",udt);    /* 输出:740281934 */
    unixtodos(udt,&a,&i);
    printf("DOS date=%dyear %umonth %uday\n",a.da--year,a.da--mon,a.da--day);
    printf("DOS time=%uhour %uminute %usecond. %uhundredth\n",i.ti--hour,
            i.ti--min,i.ti--sec,i.ti--hund);
}
/* 程序输出:
    DOS date=1993year 6month 16day
    DOS time=21hour 52minute 14second. 10hundredth
    UNIX=740281934    UNIX 时间格式
    DOS date=1993year 6month 16day
    DOS time=21hour 52minute 14second. 0hundredth 转换时有些误差
*/
```

—11 char *—Cdecl asctime(const struct tm *tblock);

将 tblock 所指 tm 类结构中的星期、日期和时间转换为一个 ASCII 码字符串,该串有固定格式:

星期 月 日 时:分:秒 年

函数返回一个指向该字符串的指针。结构类型 tm 在time.h 中定义为:

```
struct tm {
    int tm--sec;          /* 秒 */
    int tm--min;          /* 分 */
    int tm--hour;         /* 小时 (0~23) */
    int tm--mday;         /* 一月中第几天 (1~31) */
    int tm--mon;          /* 月 (0~11),实际月 = tm--mon+1。
                           它从 0 开始,可以暗示已“过了几个月” */
    int tm--year;         /* 年,实际年 = tm--year+1900 */
    int tm--wday;         /* 星期 (Week, 0~6, 星期日为 0) */
    int tm--yday;         /* 从元旦至今为止已过天数 (0~365) */
    int tm--isdst;        /* 非 0 表示夏令时间(DST, daylight saving time) */
};
```

该结构是静态的,因此在每次调用时都被重写。函数字符串中的星期日~六相当数组

```
static const char *const Weekday[7]={
    "Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
```

中的元素,一月~十二月也相当于数组

```
static const char *const Months[12]={
    "Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"};
```

中的元素。

适用于 UNIX 系统。

```
C>TYPE DATE3.C
```

```
#include "stdio.h"
#include "time.h"
#include "dos.h"
struct tm t;
char *pt;
gett()
{
    /* 注意:下句中对字符串的续行符(\)必须有 */
    printf("sec=%d min=%d hour=%d mday=%d mon=%d year=%d wday= \
%d yday=%d isdst=%d\n",t.tm-sec,t.tm-min,t.tm-hour,t.tm-mday,
        t.tm-mon,t.tm-year,t.tm-wday,t.tm-yday,t.tm-isdst);
    printf("pt=%s\n",pt);
}
main()
{
    time_t st;
    time(&st);
    t = * localtime(&st);
    gett();
    pt=asctime(&t); /* 利用结构返回字符串 */
    gett();
    delay(5000); /* 延迟 5 秒钟 */
    time(&st); /* 重取日期和时间;如果此句没有,则输出结果不会发生变化 */
    t = * localtime(&st);
    pt=asctime(&t);
    gett();
}
/* 注意:时间结构中的内容和指针所指内容的区别
sec=35 min=28 hour=9 mday=17 mon=5 year=93 wday= 4 yday=167 isdst=1
(35 秒 28 分 9 时 17 日 6 月 1993 年 星期4 过了167 天 夏令标志是1)
pt=(null)
sec=35 min=28 hour=9 mday=17 mon=5 year=93 wday= 4 yday=167 isdst=1
pt=Thu Jun 17 09:28:35 1993
(星期四 6 月 17 日 9 时:28 分:35 秒 1993 年)
sec=40 min=28 hour=9 mday=17 mon=5 year=93 wday= 4 yday=167 isdst=1
pt=Thu Jun 17 09:28:40 1993
*/
```

—12 char *—Cdecl ctime(const time_t *time);

把 time 所指的 UNIX 时间转换成一个字符串,该字符串具有固定格式:

星期 月 日 时:分:秒 年

返回指向该串的指针。该串所指区域是静态的,因次每次函数调用时它将被重写。适用于 UNIX 系统。

C>TYPE DATE9.C

```
#include "stdio.h"
#include "time.h"
#include "dos.h"
main()
{
    struct date d;
    struct time tt;
    char *pt;
    time_t st;
    struct tm t;
    time(&st);
    t = *localtime(&st);
    printf("sec=%d min=%d hour=%d mday=%d mon=%d year=%d wday= \
%d yday=%d isdst=%d\n", t.tm_sec, t.tm_min, t.tm_hour, t.tm_mday,
        t.tm_mon, t.tm_year, t.tm_wday, t.tm_yday, t.tm_isdst);
    getdate(&d);          /* 取日期 */
    gettime(&tt);          /* 取时间 */
    st = dostounix(&d, &tt); /* 把 DOS 日期和时间转换成 UNIX 时间 */
    pt = ctime(&st);        /* 直接利用前面取得的日期和时间 */
    printf("pt=%s\n", pt);
}
/* 从计算结果可知,这里 DOS 时间和由 localtime() 计算所得的地方时是一致的
sec=45 min=20 hour=16 mday=17 mon=5 year=93 wday= 4 yday=167 isdst=1
pt=Thu Jun 17 16:20:45 1993
*/
```

—13 struct tm *—Cdecl gmtime(const time_t *timer);

分解一个 timer 所指的 UNIX 时间,并把结果存入 tm 类结构中。返回指向该结构的指针。适用于 UNIX 系统。

C>TYPE DATE10.C

```
#include "time.h"
main()
{
    struct tm *t;
    long ct=740281934L;
    t=gmtime(&ct);
    printf("%d %d %d %d %d %d %d %d %d\n", t->tm_sec, t->tm_min, t->tm_hour,
        t->tm_mday, t->tm_mon, t->tm_year, t->tm_wday, t->tm_yday, t->tm_isdst);
}
```



```
/* 输出:14 52 1 17 5 93 4 167 0 */
```

```
—14 struct tm * —Cdecl localtime(const time_t * timer);
```

它根据时区和夏令时间修正 timer 所指的 UNIX 时间,使该时间变成地方时并存入 tm 类结构中。函数返回指向 tm 类结构的指针。

适用于 UNIX 系统。

```
—15 double —Cdecl difftime(time_t time2,time_t time1);
```

计算两个 UNIX 时刻的差,即计算 time2—time1 的值。返回结果为双精度值。

适用于 UNIX 系统。

```
C>TYPE DATE11.C
```

```
#include "stdio.h"
```

```
#include "time.h"
```

```
#include "stdlib.h"
```

```
main()
```

```
{
```

```
time_t t20=740281934L,t10=730221945L,t1,t2;
```

```
double t;
```

```
t2=max(t10,t20);
```

```
t1=min(t10,t20);
```

```
t=difftime(t2,t1);
```

```
printf("difference between 2 times=%f\n",t);
```

```
}/ * 输出: difference between 2 times=10059989.000000 */
```

```
—16 clock_t —Cdecl clock(void);
```

本函数返回从程序开始执行到函数调用处所化时钟滴答数,把该值除于常量 CLK—TCK 就转化为秒值。CLK—TCK 在 time.h 中定义为

```
#define CLK—TCK 18.2
```

BIOS 的计时程序在每次时钟中断(硬件中断 INT 8H)后给时钟计数器加 1,计数器的值在午夜时置为 0,或者被程序通过中断 INT 1AH 设为正确值。时钟中断每秒钟大约发生 18.2 次(精确值为 1193180/65536 次)。因此用户能计算出午夜以来过了多长时间,即将当前时钟计数器的值乘上相应频率值(即每次相当于 0.054925 秒)。

```
C>TYPE DATE12.C
```

```
#include "time.h"
```

```
#define PP printf("%d %ld\n",daylight,timezone)
```

```
/* 在time.h 中定义:
```

```
#if 1 —STDC
```

```
extern int —Cdecl daylight;全局变量,值为 1 时取夏令时间,为 0 时取标准时间。
```

```
extern long —Cdecl timezone;全局变量,时间范围,它包含 GMT 和地方标准时间之间的差,单位为秒(在 EST,即 Eastern Standard Time,美国东部标准时间,时区为西五区,它比 GMT 晚 5 个小时,即 5 * 60 * 60 = 18000);当有非 0 值时,当且仅当提供了美国标准夏令时转换。
```

```
#endif */
```

```
main()
```

```
{
```

```
long clk—t;
```

```

PP;
tzset();          /* 在 DOS 中它没做什么事 */
PP;
delay(5000);      /* 延迟 5 秒钟          */
clk-t=clock();
printf("clk-tck-number=%ld seconds=%ld\n",clk-t,(long)(clk-t/CLK-TCK));
PP;
}
/* 输出,
    1 18000    缺省值
    1 18000
    clk-tck-number=91 seconds=5
    1 18000
*/

```

—17 void —Cdecl delay(unsigned milliseconds);

程序调用后,当前程序的执行被暂停,暂停时间为 milliseconds(ms, 毫秒, 1 毫秒 等于千分之一秒),暂停时间到后程序自动恢复执行,执行从本函数调用语句的下一语句开始。注意:实际精确的暂停时间依赖于不同的操作环境,同时因程序已暂停执行,故此期间即使你按 Ctrl-Break 键也无效。

只适用于 MS-DOS。

—18 void —Cdecl sleep(unsigned seconds);

调用函数时当前程序暂停执行,暂停时间为 seconds(秒),间断时间能精确到百分之一秒,或依赖于 MS-DOS 时钟。它既能用于 MS-DOS,也能用于 UNIX 系统。

C>TYPE DATE13.C

```

#include "dos.h"
void sleep(unsigned seconds)          /* 等效函数 */
{
    /* 注意 second 与 seconds 区别,为了能区分 */
    struct time t; /* 它们,选标识符有效个数应不小于 7 个 */
    register int second,hundred;      /* 利用了寄存器变量 */
    gettimeofday(&t);                  /* 取时间 */
    hundred=(t.ti-hund>90)? 90:t.ti-hund;
    while(seconds--)                   /* 控制秒数 */
    {
        second=t.ti-sec;
        do gettimeofday(&t);
            while(second==t.ti-sec);    /* 秒值相等时 */
        }
        do gettimeofday(&t);
            while(hundred>t.ti-hund);
    }
}
main()
{
    sound(440);
    sleep(2);
    nosound();
}

```

}

—19 int gettimeofday(int handle, struct ftime *ftimep);

取磁盘文件的 DOS 日期和时间。它对早已存在且被打开的磁盘文件, 获得文件句柄 handle, 并将文件写盘时的日期和时间存入 ftimep 所指的 ftime 型结构中。ftime 结构在 io.h 中有定义(这里使用了位域, 参见《位运算与位域》一章):

```
struct ftime{
    unsigned ft—tsec : 5; /* 位0~4, 两秒间隔 */
    unsigned ft—min : 6; /* 位5~10, 分钟 */
    unsigned ft—hour : 5; /* 位11~15, 小时 */
    unsigned ft—day : 5; /* 位0~4, 日 */
    unsigned ft—month : 4; /* 位5~8, 月 */
    unsigned ft—year : 7; /* 位9~15, 从1980 起算的年数 */
};
```

磁盘文件的写盘日期和时间记录在文件目录项的第 24 ~ 25 字节(日期)和 22 ~ 23 字节(时间)中。注意: 数值均用二进制表示, 低位在前, 高位在后(如图 28-2 所示)。

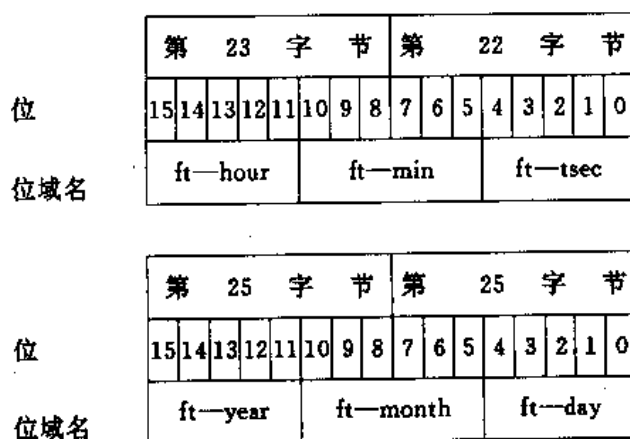


图 28-2

它相当于 DOS 中断 INT 21H 的功能调用 57H 子功能 0。入口参数是, AH=0x57, AL=0, BX=handle, DX 中返回日期, CX 中返回时间。调用成功时函数返回 0, 否则返回 -1, 并置全局变量 errno 为 1(EINVAL, 无效功能号)或 6(EBADF, 无效文件号)。

它只适用于 MS-DOS。

C>TYPE DATE14.C

```
#include "io.h"
```

```
#include "fcntl.h"
```

```
main()
```

```
{
```

```
int handle, —year;
```

```
struct ftime t;
```

```
if((handle=open("178.c", O—RDWR))== -1)exit(1);
```

```
getftime(handle, &t);
```

```
—year=t.ft—year+1980; /* ft—year 是从 1980 年开始起算时间 */
```

```
printf("ft—tsec=%u ft—min=%u ft—hour=%u ft—day=%u ft—month=%u ft—year=%u\n",
```

```

    \n", t.ft-tsec * 2, t.ft-min, t.ft-hour, t.ft-day, t.ft-month, t.ft-year);
printf("-year=%d\n", -year);
}
/* 输出:
    ft-tsec=10 ft-min=30 ft-hour=9 ft-day=18 ft-month=6 ft-year=13
    -year=1993
*/

-20 int setftime(int handle, struct ftime *ftimep);

```

修改已打开的磁盘文件（句柄为 handle）的 DOS 日期和时间。它相当于 DOS 中断 INT 21H 的功能调用 57H 的子功能 1。入口参数是，AH=0x57, AL=1, BX=handle, DX 中是设置的新日期，CX 中是设置的新时间。调用成功时函数返回 0，否则返回 -1，并置全局变量 errno 为 1 (EINVAL, 无效功能号) 或 6 (EBADF, 无效文件号)。它只适用于 MS-DOS。

```

C>TYPE DATE15.C
#include "io.h"
#include "fcntl.h"
#define VIEW getftime(handle, &t); \
    -year=t.ft-year+1980; \
printf("ft-tsec=%u ft-min=%u ft-hour=%u ft-day=%u ft-month=%u ft-year=%u\n", \
    \n", t.ft-tsec, t.ft-min, t.ft-hour, t.ft-day, t.ft-month, t.ft-year); \
printf("-year=%d\n", -year)
main()
{
    int handle, -year;
    struct ftime t;
    if((handle=open("178.c", O_RDWR))==-1)exit(1);
    VIEW;
    t.ft-year=12; /* 只修改文件日期和时间, 不会影响系统日期和时间 */
    setftime(handle, &t);
    VIEW;
}
/* 输出:
    ft-tsec=1 ft-min=19 ft-hour=10 ft-day=18 ft-month=7 ft-year=13
    -year=1993
    ft-tsec=1 ft-min=19 ft-hour=10 ft-day=18 ft-month=7 ft-year=12
    -year=1992
*/

```

```

-21 void _Cdecl tzset(void);

```

它是为与 UNIX 兼容而提供的。它根据环境变量 TZ 情况重置 daylight、timezone 等一些变量的值（没有找到 TZ 串或伪造的则设置缺省值，否则分析 TZ 串并置相应值）。

在 DOS 中，它被调用时不做任何事情。

```

C>TYPE DATE16.C
#include "stdio.h"
#include "time.h"

```

```

main()
{
    static char —DfltZone[4],—DfltLight[4];      /* 函数将跟这些数组相关 */
    char *const tzname[2]={ &—DfltZone[0],&—DfltLight[0]};
    tzset();
    fprintf(stdout,"tzname=%s %s\n",tzname[0],tzname[1]);
} /* 在MS-DOS 5.0 下输出 :tzname= */

—22 long —Cdecl biostime(int cmd,long newtime);

```

这是一个和实时钟相关的函数,它调用 BIOS 的中断 INT 1AH 的功能 00H 和 01H,用来读取 (cmd=00H) 或设置 (cmd=01H) RAM 中保持的 BIOS 实时钟 (又称 BIOS 计时器)。cmd 是它的功能号, newtime 是要设置的新时间 (相当于 CX,DX 寄存器中值,子夜以来秒摆次数)。时钟每秒大约摆动 18.2 次,将函数返回值的单位是秒摆次数,除于 18.2 便可得到十进制的秒数。注意,如果连续两次通过子夜而没有读时间 (例如,系统虽然接通,但是处于闲置状态),则将损失一天。

因为它只改变 RAM 中的时钟值,而没有改变 CMOS 时钟值,因此当设置新 RAM 中时钟后如热启动机器时, RAM 中时钟值又自动取当前 CMOS 中时钟值为值。

注意:INT 1AH 的其它功能是和 CMOS 时钟数据相关,除非你对 INT 1AH 的那些功能调用很了解,否则不要轻易将 cmd 写成 0 或 1 以外的值。

本函数只适用于 IBM PC 及其兼容机。

```

C>TYPE DATE17.C
#include "bios.h"
main()
{
    long t;
    printf("原时钟值=%ld 次\n",biostime(0,0));
    printf("设新时钟值=%ld 次\n",biostime(1,2500));
    printf("现时钟值=%ld 次\n",biostime(0,0));
}
/* 程序输出: 原时钟值=2761 次
              设新时钟值=2500 次
              现时钟值=2500 次      */

```

第二十九章 目录函数

Turbo C 中有一个专门和目录相关的标头文件 `dir.h`，利用它提供的库函数可以方便地对磁盘文件进行目录管理，如建立、更改或删除目录等。

29.1 分类

一 驱动器

- 取当前驱动器号 —1 `getdisk()`
- 设置当前驱动器 —2 `setdisk()`

二 取目录

- 取指定驱动器的当前目录 —3 `getcurdir()`
- 将指定目录变成当前目录 —4 `chdir()`
- 得到当前工作目录的完整路径名 —5 `getcwd()`

三 子目录

- 创建子目录 —6 `mkdir()`
- 删除子目录 —7 `rmdir()`

四 搜索目录

- 搜索指定文件或目录的完整路径名 —8 `searchpath()`
- 从磁盘目录中搜索指定文件并将文件信息存入 `ffblk` 类结构中 —9 `findfirst()`
- 搜索与 `findfirst()` 所取信息（在 `ffblk` 类结构中）相匹配的后续文件 —10 `findnext()`

五 分解或聚合文件名

- 聚合文件名 —11 `fnmerge()`
- 分解文件名 —12 `fnsplit()`

六 构造文件

- 构造一个不和当前驱动器的磁盘上文件同名的样板文件 —13 `mktemp()`

29.2 库函数

—1 `int Cdecl getdisk(void);`

获取当前缺省磁盘驱动器号;返回整数 0 表示驱动器 A, 1 表示驱动器 B, ...。它相当于 DOS 功能调用 19H。入口参数是, AH=0x19, 返回值在 AL 中。

只适用于 MS-DOS。

—2 int —Cdecl setdisk(int drive);

设置当前磁盘驱动器号为 drive(0=A, 1=B, ...), 函数返回可用驱动器总数(即系统允许设置的软盘和硬盘驱动器总数。如果系统仅有一个软盘驱动器, 则被认为是两个, 以便和认为系统有两个逻辑驱动器 A 和 B 的想法一致。它也包括一个用 VDISK.SYS 等设置的“虚拟盘”)。它相当于调用 DOS 中断 INT 21H 的功能 0EH。入口参数是, AH=0xe, AL=drive, 返回结果在 AL 中。

只适用于 MS-DOS。 C>TYPE DIR1.C

```
#include "dir.h"
main()
{
    int num;
    num=setdisk(0); /* 设置 A 驱动器 */
    printf("num=%d\n", num); /* 输出: num=5 */
}
/* 程序在结束时去读 A 驱动器, 并显示 A 提示符, 例如: A> */
```

—3 int —Cdecl getcurdir(int drive, char * directory);

取指定驱动器号 drive 的当前目录名放到 directory 所指的串中。drive 为 0 表示缺省驱动器, 为 1 表示 A 驱动器, 为 2 表示 B 驱动器等等。directory 指向一个最大长度为 MAXDIR(在 dir.h 中该符号常量被定义为 66)的存储区, 用于存放目录名。目录名总是相对于根目录而言, 以空字符 '\0' 结束, 它不包括驱动器号、冒号和一开头的反斜杠。如要包括这三部分内容, 应由用户程序预处理。调用成功返回 0, 否则返回 -1。

本功能相当于 DOS 中断 INT 21H 的功能 47H。入口参数: AH=47H, DL=drive(驱动器号), DS:SI=directory(指针指向 64 字节的内存区域)。

只适用于 MS-DOS。

C>TYPE DIR2.C

```
#include "stdio.h"
#include "dir.h"
char * cur—DIR(char * path)
{int i; /* 为包括驱动器名、冒号和反斜杠而预处理 */
    strcpy(path, "X:\\"); /* 字符 X 是暂用的, 以表示未定驱动器名 */
    path[0]='A'+getdisk(); /* 对第一个元素允许赋值, 由 getdisk 得 */
    /* 当前驱动器号。加 A 后得当前驱动器名 */
    i=getcurdir(0, path+3); /* 从 path 第 3 个元素开始装入目录名 */
    if(i != 0){printf("error i=%d\n", i); exit(1);}
    return(path); /* 注意, 此处无 return(path) 效果一样 */
} /* 因 path 指向数组 curdir */

main()
{
    char curdir[MAXDIR];
    cur—DIR(curdir);
```

```
printf("当前目录是:%s\n",curdir);
}
```

/* 例如:可能输出:当前目录是:B:\TC2\TC22 */

—4 int —Cdecl chdir(const char * path);

将指定的目录 path 变为当前工作目录。path 中可以带驱动器名和开始的反斜杠,如 chdir("a:\\tc");、chdir("A:\\tc");、chdir("a:/tc");等。path 必须是已存在的目录。当函数调用成功时返回 0,否则返回 -1,并置变量 errno=2(ENOENT,没有这样的文件或路径。注意,有些 DOS 可能返回出错码 3),而原有的当前路径保持不变。

如果新的目录名包括驱动器字母,则默认驱动器不变,仅改变该驱动器的当前目录。改变当前目录的同时也改变了 FCB 文件调用工作的目录。

它相当于 DOS 中断 INT 21H 的功能 3BH。入口参数是, AH=0x3B, DS:DX=path。适用于 UNIX 系统。

DOS 的 CHDIR 命令除了指定当前目录外,还可以显示当前目录(如用 C>CD 操作)。

C>TYPE DIR3.C

```
#include "dir.h"
#include "errno.h"
main()
{
    const char * path="F:/tc";
    int num;
    num=chdir(path);
    if(num != 0)printf("errno=%d ",errno);
    printf("num=%d\n",num);
}
```

/* 当 path 不存在时输出:errno=2 num=-1. */

—5 char * —Cdecl getcwd(char * buf, int buflen);

得到当前工作目录 cwd(current working directory)的完整路径名,最长为 buflen 个字节,并把它存在缓冲区 buf 中。如果该完整路径名的长度(包括结尾的空字符 '\0')超过 buflen,则出现超出范围的错误(ERANGE)。所谓完整路径名,是指该路径包含当前缺省驱动器名、冒号、开始的反斜杠及随后的路径部分,完整路径名仍是相对于根目录而言的。为保证不出错,最好将 buflen 取成 MAXDIR。

如果 buf 为 NULL,可用 malloc() 函数分配一个 buflen 个字节长的缓冲区。如果分配失败,则出现存储区不够的错误(ENOMEM)。以后如果需要释放这个占用的内存缓冲区,可用 free(buf) 进行。

函数调用成功,返回指针 buf,否则返回 NULL,并置 errno 为 15(ENODEV),或 8(ENOMEM),或 34(ERANGE)。只适用于 MS-DOS。

C>TYPE DIR4.C

```
#include "stdio.h"
#include "dir.h"
main()
{
    char * ptr,curdir[MAXDIR];
    ptr=getcwd(curdir,MAXDIR);
```



```
printf("当前目录是:%s,就是:%s\n",curdir,ptr);
}
```

/* 例如,执行

```
B:\TC2\TC22>B:DIR4
```

输出:当前目录是:B:\TC2\TC22,就是:B:\TC2\TC22

*/

```
—6 int —Cdecl mkdir(const char *path);
```

在当前文件目录项末尾建立一个名为 path(ASCII 码字符串)的子目录。当指定目录被建立, mkdir 返回 0, 否则返回 -1。出错时, errno=3(路径未找到)或 5(拒绝存取,例如 path 早已存在)。

字母 mk 可理解为 mark(标记),也可理解为 make(建立)。

它相当于 DOS 中断 INT 21H 的功能 39H,入口参数为, AH=39H, DS:DX= 文件路径名指针 path。

此函数相当于 DOS 的 MKDIR 命令。

```
C>TYPE DIR5.C
```

```
#include "dir.h"
```

```
#include "dos.h"
```

```
main(int argc,char *argv[])
```

```
{
```

```
char *path2="B:/TC2", *path3="b:\\tc3";
```

```
/* char *path1="B:\\TC1";
```

```
union REGS r;
```

```
struct SREGS s;
```

```
r.h.ah=0x39;
```

```
s.ds=FP—SEG(path);
```

```
r.x.dx=FP—OFF(path);
```

```
int86x(0x21,&r,&s); 程序执行结果是在 B 盘上产生了目录 tc1
```

```
*/
```

```
if(argc==2)mkdir(argv[1]);
```

```
mkdir(path2);
```

```
mkdir(path3);
```

```
}
```

```
/* 键入 dir1 b:tc4 后,B 盘上将建立三个子目录:TC2、TC3 和 TC4 */
```

```
—7 int —Cdecl rmdir(const char *path);
```

它是 mkdir() 的反函数,用于删除已有的子目录 path。删除子目录 path 时,path 必须满足三个条件,一是该子目录中除包括 '.' 和 '..' 项外别无其它文件(或称子目录为“空”);其次,它不能是根目录;最后,它也不能是当前目录。删除成功时返回 0,出错时返回 -1。它相当于 DOS 中断 INT 21H 的功能 3AH。入口参数: AH=3AH, DS:DX= 文件路径名指针 path。

此函数相当于 DOS 的 RMDIR 命令。

```
C>TYPE DIR6.C
```

```
#include "dir.h"
```

```
#include "errno.h"
```

```

main()                                /* 当指定的目录未找到或无效时 errno=3 */
{
    /* 当指定目录里尚有文件即非空时,或者 */
    rmdir("b:\\tc2");                /* 指定目录为当前目录,或为根目录时, */
    printf("errno=%d\n",errno);        /* 都将打印 errno=5 */
}

```

—8 char * —Cdecl searchpath(const char * file);

对一个指定的文件或子目录 file 搜索其所在的路径。其搜索过程是,先检查当前驱动器的当前目录,如果没有找到则取环境变量 PATH (参见 getenv() 函数),按它指定的目录再顺序搜索,直到找到该文件或子目录为止。如果搜索成功,函数返回一个完整的路径名(即含驱动器名、开始的反斜杠等等),否则返回 NULL。返回的字符串可用于 open 或 exec... 等函数,以便存取文件。

注意:由于函数返回的字符串自动存于一个静态缓冲区中,所以本次对它调用的结果将冲了前次获得的结果,或者说任何后续调用都将破坏前一次调用的内容。

只适用于 MS-DOS。

C>TYPE DIR7.C

```

#include "stdio.h"
#include "dir.h"
#include "errno.h"
#define PP(PTR) printf("errno=%d ",errno);\
    printf("#PTR"="%p\n",PTR);\
    printf("内容:%s\n",PTR)

main()
{
    char * file="*.l", * ptr, * ptr1;
    ptr=searchpath(file);
    PP(ptr);                                /* 不允许使用通配符星号 */
    ptr=searchpath("B:\\TC2\\tc22\\*.exe");
    PP(ptr);
    ptr=searchpath("B:\\TC2\\tc22\\DIR?.exe"); /* 不允许使用通配符问号 */
    PP(ptr);
    ptr=searchpath("B:\\TC2\\tc22");          /* 允许查子目录 */
    PP(ptr);
    ptr=searchpath("B:/TC2/TC22\\DIR3.EXE"); /* 允许查文件 */
    PP(ptr);
    ptr1=searchpath("B:/TC2/TC22\\DIR3.EXE"); /* ptr 和 ptr1 有同一地址 */
    PP(ptr1);
}

/* 如果 B 驱动器中有文件 B:\TC2\TC22\DIR3.EXE 存在,则输出:
errno=0 ptr=0000
内容:(null)
errno=0 ptr=0000
内容:(null)
errno=0 ptr=0000
内容:(null)

```

```

errno=0 ptr=05DF      全局变量 errno 不能作为判别条件
内容:B:\TC2\TC22
errno=0 ptr=05DF
内容:B:/TC2/TC22\DIR3.EXE
errno=0 ptr=05DF
内容:B:/TC2/TC22\DIR3.EXE
*/

```

—9 int —Cdecl findfirst(const char *path, struct fblk *fblk, int attrib);

函数通过 DOS 的 INT 21 中断功能 4EH 从磁盘目录中搜索指定的文件 path。4EH 的入口参数是, AH=0x4E; DS:DX=path, CX=attrib, 如果找到与路径名(path)及指定的属性(attrib)相匹配的文件, 则将信息填入磁盘传输区(DTA, disk transfer address)中。DTA 是内存中的一个缓冲区。当 DOS 启动一个文件时, 它把程序段前缀(PSP)偏移为 80H k FFH 的 128 字节当作 DTA (参见《磁盘文件的结构》一章)。不过, DAT 也可由 INT 21H 中断的功能 1AH 重新设置到内存中任何位置(入口参数是, AH=1AH, DS:DX=DTA 的首地址), 当然, 应保证有足够的空间, 否则会出错(参见函数 getdta() 和 setdta(), 如果需要, 在每次调用 findfirst() 或 findnext() 函数后, 可用 getdta() 或 setdta() 保存或恢复)。注意, 对流式文件管理可不需 DTA。只适用于 MS-DOS。

与 DTA 相当的缓冲区是结构 fblk(调用函数后设 DTA 为 fblk 地址), 结构 fblk 在 dir.h 中被定义为:

```

struct fblk {
    char    ff-reserved[21]; /* 为 DOS 保留, 用于查找下一个文件 用户不应对它操作。对 DOS
                                3.2、DOS 3.3 和 DOS 5.0 版本有
                                偏移量 大小    说    明
                                00H    字节    驱动器号 (0= 缺省, 1=A)
                                01H    11 字节 搜索文件名样板
                                0CH    字节    搜索属性
                                0DH    字     目录中登记项计数
                                0Fh    字     亲目录开始处的编号
                                11H    4 字节 保留          */
    char    ff-attrib;       /* 文件的属性          */
    unsigned ff-time;        /* 文件的时间          */
    unsigned ff-fdate;       /* 文件的日期          */
    long     ff-fsize;        /* 文件的长度, 低字在前, 高字在后 */
    char     ff-name[13];     /* 文件基本名 + 扩展名 */
};

```

参见定义的结构类型 dosSearchInfo;

```

typedef struct {
    char    drive;
    char    pathtern[13];
    char    reserved[?];
    char    attrib;
    short   time;
    short   date;
    long    size;
}

```

```

char nameZ[13];
}dosSearchInfo;

```

函数将 DTA 的地址设置为 fblk 的地址。如果找到一个文件,结构 fblk 将用文件目录信息来填入。

文件属性 (attrib) 在 dos.h 中定义有符号常量:

```

#define FA_RDONLY    0X01    /* 只读属性 */
#define FA_HIDDEN    0x02    /* 隐藏文件 */
#define FA_SYSTEM    0x04    /* 系统文件 */
#define FA_LABEL     0x08    /* 卷 标 */
#define FA_DIRECT    0x10    /* 子 目 录 */
#define FA_ARCH      0x20    /* 档 案 */

```

attrib 可以是它们的适当组合,例如,attrib=FA_RDONLY | FA_ARCH。

如果搜索成功返回 0,否则返回 -1,并置全局变量 errno 为 2(ENOENT)或 18(ENMFILE)。

注意:当文件名 path 中含通配符时,由于指定属性 attrib 的位 0 和位 5 有可能被忽略,即属性为归档或只读的也将被找出来,结果寻得的匹配目录或文件有可能多于所求的。为确定起见,在找出后需将 ff-attrib 与 attrib 再作一次严格比较。

```

C>TYPE DIR8.C
#include "stdio.h"
#include "dir.h"
#include "errno.h"
#define PP printf("done=%d fk 地址=%p\n\n",done,&fk)
main()
{
    struct fblk fk;
    int done;
    done=findfirst("b:\\tc2\\tc22\\*. *",&fk,0); /* 函数不能用于搜索子目录 */
    PP;      /* 如 findfirst("b:\\tc2\\tc22",&fk,0) 将调用出错 */
    while(!done)
    {
        printf("属性=%d\n",fk.ff-attr);
        printf("日期=%d\n",fk.ff-fdate);
        printf("时间=%d\n",fk.ff-ftime);
        printf("文件名=%s\n",fk.ff-name);
        printf("文件长度=%ld\n",fk.ff-fsize);
        printf("BY DOS=%p\n",fk.ff-reserved);
        done=findnext(&fk);
        PP;
    }
}

/* 如果 B 驱动器的目录 B:\TC2\TC22\ 中有文件
DIR3.EXE 10054 07-28-93 9.17p
P.C664 07-29-93 4.41p
则输出:

```

```

done=0   fk 地址=FFB6
属性=32
日期=6908
时间=-21967           可以化为更明显的时间,参见后面叙述
文件名=DIR3.EXE
文件长度=10054
BY DOS=FFB6
done=0   fk 地址=FFB6
属性=32
日期=6909
时间=-31451
文件名=P.C
文件长度=664
BY DOS=FFB6
done=-1   fk 地址=FFB6
*/

```

```

-10 int --Cdecl findnext(struct fblk * fblk);

```

随 findfirst() 函数之后取得与 findfirst() 中给定的 path 相匹配的后续文件。

它相当于 DOS 中断 INT 21H 的功能 1AH、4FH。1AH 用于建立磁盘 I/O 操作和目录搜索的缓冲区地址,入口参数是, AH=1AH, DX=fblk; 4FH 的入口参数是, AH=0x4F。但在此前,应先调用 INT 21 中断的功能 4EH,以便 DTA 指向 4EH 调用后的区域。

```

-11 void --Cdecl fnmerge (char * path, const char * drive, const char * dir,
                        const char * name, const char * ext);

```

建立一个形式上像如下表达式中的完整的文件名:

```

path=drive\dir\name.ext

```

参数的可取的最大值(最多字符数,内中都包括了空字符终结符 '\0')参见 dir.h 中定义的符号常量:

```

#define MAXPATH    80   /* 完整文件名 path 的最大长度          */
#define MAXDRIVE   3    /* 驱动器 drive,包括冒号(,)          */
#define MAXDIR     66   /* 子目录 dir,包括开始和结尾的反斜杠(\) */
#define MAXFILE    9    /* 基本文件名 name                  */
#define MAXEXT     5    /* 文件扩展名 ext,包括开始的圆点(.)    */

```

它们的最小值为 NULL (即 0)。当为 NULL 时,表示相应成份被分析,但不存储。

只适用于 MS-DOS。

```

-12 int --Cdecl fnsplit (const char * path, char * drive, char * dir,
                        char * name, char * ext);

```

它是 fnmerge() 的反函数,即将一个完整文件名 path 分成 drive、dir、name 和 ext。不同的是 fnsplit() 返回的一个整数是一个标志,各位在 dir.h 中有定义:

```

#define WILDCARDS0x01 /* 位0,当值为1时表示有通配符 * 或 ? */
#define EXTENSION0x02 /* 位1,当值为1时表示有扩展名          */
#define FILENAME 0x04 /* 位2,当值为1时表示有基本文件名      */
#define DIRECTORY0x08 /* 位3,当值为1时表示有子目录名        */

```

```
#define DRIVE 0x10 /* 位4,当值为 1 时表示有驱动器名 */
```

只适用于 MS-DOS。

注意:当目录名为“.”或“..”时,调用本函数后获得的文件名 name 为空。

C>TYPE DIR9.C

```
#include "stdio.h"
#include "dir.h"
char drive[MAXDRIVE],dir[MAXDIR],file[MAXFILE],ext[MAXEXT];
main()
{
    char s[MAXPATH],t[MAXPATH];
    int flag;
    for(;;)
    {
        printf("\n 请输入一个完整文件名,若击Ctrl-C则退出!\n");
        printf(">");          /* 输入一个提示符 */
        if(!gets(s))break;    /* 输入字符串 */
        printf("您键入了:%s\n",s);
        flag=fnsplit(s,drive,dir,file,ext);
        printf("标志=0x%x\n",flag);
        printf("各部分为:%s %s %s %s\n",drive,dir,file,ext);
        fnmerge(t,drive,dir,file,ext);
        printf("重新合并为:%s\n",t);
        if(strcmp(t,s)==0)printf("两者相同\n");
        else printf("前后不一致!\n");
    }
}

/* 例如输出:
    请输入一个完整文件名,若击Ctrl-C则退出!
    >C:\\TC\\MY?. *
    您键入了:C:\\TC\\MY?. *
    标志=0x1f
    各部分为:C:  \\TC\\MY?. *
    重新合并为:C:\\TC\\MY?. *
    两者相同
    请输入一个完整文件名,若击Ctrl-C则退出!
    >
    */
```

—13 char * —Cdecl mktemp(char * template);

template 是“模型、样板”的意思,函数是供程序员通过一个样板文件名 template 来构造一个不和当前驱动器的磁盘上文件同名的文件名(不妨称它为 file)。template 文件名有这样的特点:它的最后 6 个字符一定是大写字母 X。file 与 template 的不同仅仅在于这最后 6 个字符(即 XX.XXX,后 3 个为文件扩展名),或者说,除此之外的字符它们是完全一样的。如果构造成功,函数返回非空的 template 的地址,并且 template 中的内容已是 file;否则返回 NULL,或者说 file 将不存在。template 的后 6 个字母的替换是从 AA.AAA 开始的,其每一

个 X 字母可能被 A ~ Z 中任一字母依次替换,是否被替换取决于磁盘上是否有同名文件存在, file 始终不会和磁盘上的已有文件名同名。

```
C>TYPE DIR10.C
#include "dir.h"
#include "errno.h"
#define PPprintf("errno=%d ",errno);\
    printf("addr=%p\n",addr);\
    printf("内容:%s\n",addr)

main()
{
    char *template="b;QsssDIRXXXXXX", *addr;
    addr=mkttemp("b;dirxxxxxx"); /* 6 个小写字母 x */
    PP;
    addr=mkttemp("b;dirXXXXX"); /* 5 个大写字母 X */
    PP;
    addr=mkttemp("b;dirXXXXXX"); /* 6 个大写字母 X */
    PP;
    addr=mkttemp(template); /* 从输出可见未对文件名检查 */
    PP;
}

/* 当驱动器上已有文件 DIRAA.AAA 时程序输出:
errno=0 addr=0000
内容:(null)
errno=0 addr=0000
内容:(null)          全局变量 errno 不能作为判别条件!
errno=2 addr=01F7
内容;b;dirAA.AAB
errno=2 addr=0194
内容;b;QsssDIRAA.AAA
```

注意:驱动器的磁盘上不会出现文件 b;dirAA.AAB 或 b;QsssDIRAA.AAA */

29.3 一个全盘搜索文件程序

在目录操作中最主要的是搜索文件。下列程序则能在全盘找出任意文件,而不管它的文件属性如何。它能显示文件全名、属性、长度、书写日期和时间。

当一个文件搜索到后就可进行其它操作(代替程序DIR11.C中的dirproceeds()函数),如

1. 删除文件

首先用户应确认是否要删除;当要删除时使用 remove() 或 unlink() 函数;最后检查删除是否成功(成功时返回 0)。文件删除后,磁盘目录中该文件登记项的第 0 个字节改为 E5H,而文件在磁盘上其它内容未变(这些内容也可称为无用的“垃圾”)。这样的文件必要时可恢复。如果要清除垃圾,可在删除前先用 open() 打开该文件后用 filelength() 函数求出文件长度,然后对整个文件用 write() 写入清 0 数据(可使用像 memset(缓冲区, '\0', 文件长度)那样的语句)。但这样处理后的文件不能再被恢复。

删除目录也可用这两个函数,但先将该目录内文件除 "." 和 ".." (它们属性为目录)外

的文件全删除,然后返回上一级目录删除该目录。

对只读、隐含或系统文件等在删除前要先将它们改变属性。

2. 改文件属性

首先用户应确认是否要改属性;当要改时使用函数 `chmod()`,如将文件改成可读写用 `chmod(要改文件名,S-IREAD|S-IWRITE)`;

最后检查改写是否成功(成功时返回 0)。

3. 文件换名

首先用户应确认是否要换名;当要换名时使用 `rename()` 函数;最后检查换名是否成功(成功时返回 0)。

利用 `rename()` 函数将文件换名也可以实现文件移动(move),即将某子目录下的文件搬到另一个子目录下,而在原子目录里该文件已不再存在。

4. 对非指定文件操作

指定对全盘所有文件寻找,找到后判别是否为指定文件。若是,则取出处理。

```
C>TYPE DIR11.C
#include "stdio.h"
#include "dos.h"
#include "dir.h"
#include "conio.h"
#include "string.h"
#include "sys\stat.h"
#include "stdlib.h"
#include "time.h"
#define MAX-DIR-NUM 500
#define FAD FA-DIREC |FA-LABEL
#define FAF FA-RDONLY |FA-HIDDEN |FA-SYSTEM |FA-ARCH
#define FADF FAD|FAF
char *week(int y,int m,int d);
char *dateT(int d);
char *timeT(int d);
char file--0[80];
static char path[80],path1[MAX-DIR-NUM][80],s[80],display-name[80];
int sum=0,pageline=0,lab;
struct fblk fp;
main(int argc,char *argv[]) /* 全盘搜索文件程序 */
{
    /* 根据陈凤国的程序改写并注释 */
    register int done1,done2;
    char safe;
    char path--0[MAXDIR];
    char drive[3],subdir[66],file[9],ext[5];
    int level=0,n=0;
    int drive--0=getdisk(); /* 保存当前驱动器号 */
    if(argc<3 || argc>4)help();
    /* 选不同 lab 值便可显示不同属性的文件 */
    if( !strcmp(argv[1],"-DF")|| !strcmp(argv[1],"-FD"))lab=FADF;
    else if( !strcmp(argv[1],"-F"))lab=FAF;
```



```

else if( !strcmp(argv[1],"-D"))lab=FAD;
else help();
if( argc==4 && strcmp(argv[3],"/p"))help();
clrscr();
printf("找到时是否要显示(Y/N)?");
safe=toupper(getche());
printf("\n");
getcwd(path-0,MAXDIR);          /* 保存当前目录 */
path[0]='A'+getdisk();
fnsplit(argv[2],drive,subdir,file,ext);
if(argc==3)
{
done1=findfirst(argv[2],&fp,lab);
if(!done1)
{
strcpy(file-0,drive);
strcat(file-0,subdir);
strcat(file-0,fp.ff--name);
if(safe=='Y')dirprocess();
else printf("%s 被找到!\n",file-0);
}
else printf("%s 未寻找到!\n",argv[2]);
return;
}
if(argv[2][1]!=':')
{
path[0]=argv[2][0];
strcat(file,ext);
strcpy(argv[2],file);
}
strcat(path,".");
do                                /* DO 循环开始,全盘查找 */
{
if(kbhit())                      /* 按任一键结束查找 */
{
getch();
exit(1);
}
strcat(path,"\\");
strcpy(path1[n],path);
strcpy(s,path);
strcat(path,argv[2]);
done2=findfirst(path,&fp,lab);    /* 搜索 */
while(!done2)                  /* done2 循环开始,处理当前目录中的文件 */
{
strcpy(display-name,s);

```

```

    strcat(display—name,fp. ff—name);
if(safe == 'Y')
{
    strcpy(file—0,display—name);
    dirprocess();
}
else
    printf("%s 已找到\n\n",display—name);
    pause();
done2=findnext(&fp);
} /* done2 循环结束 */
    strcpy(path,path1[n]);
    strcat(path," * . * ");
    done1=findfirst(path,&fp,FA—DIREC);
while(!done1) /* done1 循环开始,记忆各级目录位置 */
{ /* 搜索非"."和".."的目录名 */
if(stricmp(fp. ff—name,".") && stricmp(fp. ff—name,".."))
    if(fp. ff—attrib == FA—DIREC)
    {
        strcpy(path1[n+1],path1[n]); /* 当前目录后退 */
        strcat(path1[n++],fp. ff—name); /* 登记找到目录 */
    }
    done1=findnext(&fp);
} /* done1 循环结束 */
    strcpy(path,path1[level++]); /* 获得下次循环的当前目录 */
}while(level<=n); /* DO 循环结束 */

```

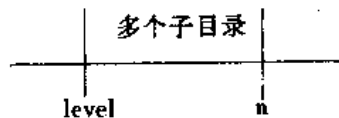


图 29—1

```

setdisk(drive—0); /* 恢复原来驱动器 */
chdir(path—0); /* 恢复原来路径 */
}
dirprocess() /* 处理显示 */
{
    pause();
    printf("\n 名: %s, 属性: ",file—0);
    if(fp. ff—attrib & FA—RDONLY)printf("只读+");
    if(fp. ff—attrib & FA—HIDDEN)printf("隐含+");
    if(fp. ff—attrib & FA—SYSTEM)printf("系统+");
    if(fp. ff—attrib & FA—LABEL )printf("卷标+");
    if(fp. ff—attrib & FA—DIREC )printf("目录+");
    if(fp. ff—attrib & FA—ARCH )printf("归档+");
    printf("\b,");
    if(fp. ff—attrib == FA—DIREC || fp. ff—attrib == FA—LABEL)printf("\n");
    else printf(" 长度: %ld 字节\n", fp. ff—fsize);
    printf(" 即 %s ",dateT(fp. ff—fdate));
}

```

```

        printf("即%s",timeT(fp.ff-ftime));
    }
char *dateT(int d) /* 将磁盘目录中记录的日期 d 转换为实际日期 */
{
    int i;
    char str[10], *p="{ "};
    char *s[]={"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
    unsigned weekday,year,month,day;
    year=(d>>9&0x7f)+1980; /* 或d=(year-1980)*512+month*32+day */
    month=d>>5&0xf;
    day=d&0x1f;
    p=week(year,month,day);
    for(i=0;i<7;i++)
        if(!strcmp(s[i],p)) {weekday=i;break;}
    printf(" %d 年%d 月%d 日 星期%d",year,month,day,weekday);
    strcat(p," ");

    if(month<10)strcat(p,"0");
    strcat(p,itoa(month,str,10));
    strcat(p,"-");
    if(day<10)strcat(p,"0");
    strcat(p,itoa(day,str,10));
    strcat(p,"-");
    strcat(p,itoa(year,str,10));
    return (p);}

char *week(int y,int m,int d) /* 求星期几 */
{
    /* 0 = 星期日 Sunday */
    /* 1 = 星期一 Monday */
    /* 2 = 星期二 Tuesday */
    /* 3 = 星期三 Wednesday */
    /* 4 = 星期四 Thursday */
    /* 5 = 星期五 Friday */
    /* 6 = 星期六 Saturday */
    struct date d1;
    struct time t1;
    long td;
    char *p=(char *)malloc(25);
    d1.da-year=y;
    d1.da-mon=m;
    d1.da-day=d;
    t1.ti-min=t1.ti-hour=t1.ti-hund=t1.ti-sec=0;
    td=dostounix(&d1,&t1);
    p=ctime(&td);
    p[3]='\0';
    return (p);
}

char *timeT(int d) /* 将磁盘目录中记录的时间 d 转换为实际时间 */
{
    char p[20],str[10];
    unsigned hours,minutes,second;
    hours=d>>11&0x1f; /* 或 d=hours*2048+minutes*32+second/2 */
    minutes=d>>5&0x3f;

```

```

second=(d & 0x1f) * 2;
if( hours<12 || (hours==12 && minutes==0 && second == 0))
    strcpy(p,itoa(hours,str,10));
else strcpy(p,itoa(hours-12,str,10));
strcat(p,":");
if(minutes<10)strcat(p,"0");
strcat(p,itoa(minutes,str,10));
strcat(p,":");
if(second<10)strcat(p,"0");
strcat(p,itoa(second,str,10));
if( hours<12 || (hours==12 && minutes==0 && second == 0))
    strcat(p,"a");
else strcat(p,"p");
printf(" %d 时%d 分%d 秒",hours,minutes,second);
return (p);
}

help()                                /* 帮助 */
{printf(" * * * * 命令行输入方法 * * * *\n");
 printf(" 执行文件名 -DF|-F|-D 文件名 [/p]\n");
 printf(" -DF 表示查各种属性的文件\n");
 printf(" -F 表示只寻找文件\n");
 printf(" -D 表示只寻找目录与卷标\n");
 printf(" /p 表示全盘寻找,缺省时在指定目录(不进子目录)中寻找\n");
 printf(" 文件名允许用通配符 * 或 ?\n");
 exit(1);
}

pause()                                /* 满屏处理 */
{
sum++;                                /* 找到文件数加 1 */
pageline+=2;                          /* 显示行数加 2 */
if(pageline>21)
{
printf("\n 按任一键继续...");
getch();
printf("\n");
pageline=0;
}
}

```

29.4 DOS 5.0 的 dir 命令

在 DOS 提示符下键入 C>DIR/? 得到下述有关 DOS 的 dir 命令的帮助信息:

```

Displays a list of files and subdirectories in a directory.
DIR [drive:][path][filename] [/P] [/W] [/A[:attributes]]
    [/O[:sortorder]] [/S] [/B] [/L]

```

[drive;][path][filename]

Specifies drive, directory, and/or files to list.

/P Pauses after each screenful of information.

/W Uses wide list format.

/A Displays files with specified attributes.

attributes D Directories R Read-only files

H Hidden files

A Files ready for archiving

S System files

- Prefix meaning "not"

/O List by files in sorted order.

sortorder N By name (alphabetic) S By size (smallest first)

E By extension (alphabetic) D By date & time (earliest first)

G Group directories first - Prefix to reverse order

/S Displays files in specified directory and all subdirectories.

/B Uses bare format (no heading information or summary).

/L Uses lowercase.

Switches may be preset in the DIRCMD environment variable. Override

preset switches by prefixing any switch with - (hyphen) -- for example, /-W.

1. 语法:

```
DIR[drive;][path][filename][/p ][/w][/a[[:]attributes]]  
[ /o[[:]sortorder] ][/s][/b][/l]
```

不使用开关的缺省命令即 `dir filename` 相当于

`DIR filename /ad-d`

2. 任选项使用方法 (任选项中说明开关的字母大小写不分, 开关位置任意)

(1) [drive;][path]filename]

文件的驱动器名、路径名和文件基本名与扩展名。

(2) /p

每次显示一整屏幕子目录或文件后暂停, 按任一键继续显示。

(3) /w

横向显示目录或文件, 每行显示 5 个子目录或文件。

(4) /a[[:]attributes]

使用本开关可只显示你指定属性的子目录或文件。不用本开关时显示除系统和隐含文件外的子目录或文件。如果你只用 /a 而不指定属性, 则可显示包括隐含和系统文件在内的全部文件。冒号可用也可不用。开关参数可组合使用, 参数之间不用空格隔开 (如用空格隔开则应加上 /a。如 /abs, 也可写成 /ah /as)。当开关书写错误时, 将出现

Invalid switch

的提示。开关参数 attributes 的值可为:

h 隐含文件

-h 非隐含文件

s 系统文件

-s 非系统文件

d 子目录

-d 非子目录, 即只是文件

a 用于归档文件 (backup)

- a 非归档即不能改变的文件（如子目录、系统文件及某些大库文件等）
- r 只读文件
- r 非只读文件

对同一种类的开关同时在命令行上出现时，最后的起作用。如 C>DIR/A;-HH 将显示隐含文件，即H起作用，-H被忽略。

(5) /o[[:]sortorder]

按某种目录排序显示子目录或文件。没有使用开关时，显示按子目录或文件在磁盘目录项里的实际位置的先后次序进行。如果只使用 /p 而没用 [[:]sortorder] 参数，则先按字母顺序显示子目录，然后按字母顺序显示文件。冒号可用也可不用。

开关参数 sortorder 的值可为（也允许组合使用）：

- n 按子目录或文件名的基本名的字母排列顺序（从A到Z）显示
- n 按子目录或文件名的基本名的反字母排列顺序（从Z到A）显示
- e 按子目录或文件名的扩展名的字母排列顺序（从A到Z）显示
- e 按子目录或文件名的扩展名的反字母排列顺序（从Z到A）显示
- d 按从小到大的日期和时间排列显示
- d 按从大到小的日期和时间排列显示
- s 按由小到大的文件长度排列显示
- s 按由大到小的文件长度排列显示
- g 按先子目录后文件的排列显示
- g 按先文件后子目录的排列显示

注意：当同时指定几个排序开关时，先指定的先起作用，后指定的是再对已排结果起作用，即相当于多重排序。如用 C>DIR /O,e-s z *. *
有

				第一次排序	第二次排序
Z4		9841 09-12-93	7,17a	先无扩展名	长度最大
ZMH		4710 10-27-93	8,10a		长度次之
Z3		4001 05-14-93	10,43a		长度最小
ZMH93	<DIR>	10-31-93	3,39p	再子目录	
ZMULU	93	3851 11-08-93	1,33p	有扩展名	
Z1	I	9818 08-25-93	3,09p		长度最大
Z6	I	8259 11-11-93	1,41p		长度次之
Z2	I	2306 05-14-93	4,17p		长度最小

第一次按扩展名排序，第二次再对第一次排序结果按文件长度排序。值得指出的是，当你在命令上多次使用同种开关时，如 C>DIR /O:n z *. c /o:-n 则只有前一种起作用（/O:n），后一种被忽略。

(6) /s

当没有指定路径时，列出指定文件（通常带有通配符 * 或 >）所在的全部子目录名和文件名等全部信息，并最终指出文件总数和共占字节数。当指定路径时，只列出指定路径中的文件信息。

(7) /b

和开关 /s 类似，不同的是每行只是列出子目录名或文件基本名和扩展名，而其它信息不列出。使用本开关后，将忽略 /w 开关。

(8) /l (l 为字母L的小写字母)

使用本开关后,子目录或文件名按小写字母显示(不用此开关则按大写字母显示),

3. 使用环境变量dircmd

可以用DOS的set命令设置环境变量dircmd。例如,当你设置 SET DIRCMD=/A/O:N/P 之后,则用

```
C>DIR *.C
```

时相当于执行 C>DIR /A/O:N/P *.C

注意:命令行上的开关优先于环境变量中设置的开关。例如,当你按上述设置环境变量后,如用 C>DIR/W *.C 列目录时仍以横向列目录(每行5个),即忽略了环境变量中设置的开关/P。

可像取消其它环境变量一样,用

```
SET DIRCMD=
```

便可将它设置的值取消。

29.5 功能强于DOS 5.0内部命令dir的CDIR

从上述可见,DOS 5.0的DIR命令比之于DOS 3.X在功能上有所增强,主要表现在:

1. 可访问各种属性(包括隐含、系统)文件目录;
2. 能对搜索到的文件目录进行多重排序;
3. 累计列出文件的总长度;
4. 可访问当前目录下的各子目录里的文件目录。

对DOS 3.X的用户,一定也希望拥有这种功能,程序CDIR则能满足他们的要求(参见程序CDIR.C中函数help())。该程序可在Turbo C集成环境(Small存储模式)下编译连接后生成执行文件CDIR.EXE。它比DOS 5.0的DIR命令在功能上增加了允许全盘查找和访问指定日期范围内的文件(分小于、等于和不等三种情况)。程序将不常用的卷标显示作为单独的选项处理。容易看出,用户只要稍将程序修改,还可列出指定长度范围内的文件。对能直接显示汉字的系统,还可将显示内容改用汉字显示。

本程序未考虑对环境变量的处理(如要处理,可用Turbo C的getenv()函数)。此外,对输入文件属性是严格的。例如,对一个只读、隐含、系统及归档文件,应同时指出这四种属性,即/arhsa,而不能只指出其中部分属性。最后,文件名是一个单独选项,后面不能紧跟开关"/"(这种限制旨在增加命令行的清晰性)。

注意:当在执行程序时如最后出现“Null pointer assignment”,表明程序被分配了一些不可靠的内存,应酌情减小一些数组尺寸。

本程序要占用较多内存和堆栈(因为使用了递归),为处理更多文件可以考虑采用暂时文件、程序覆盖等技术,如果不需排序,也可使用查得后立即显示等方法。

最后要指出的是,本程序由于并不直接访问磁盘扇区,所以只处理标准目录,而对将目录自行更改或加密的无效。

从本程序读者也可看到命令行上一些开关的处理方法。

```
C>CDIR.C
```

```
#include "stdio.h"
```

```
#include "dos.h"
```

```

#include "dir.h"
#include "conio.h"
#include "string.h"
#include "sys\stat.h"
#include "stdlib.h"
#include "ctype.h"
#include "time.h"
#define PP 0x1
#define WW 0x2
#define SMALL 0x0
#define CAPS 0x1
#define ONOT 0x01
#define OYES 0x02
#define ONO 0x0
#define NUM      5 /* 待排序关键字个数 */
#define MAXELE    350 /* 与排序段相关的临时变量个数 */
#define MAXDIRNUM 200 /* 子目录数 */
#define PSTOP if(j9==23)
                { printf("press any key to continue ... \n"); \
                  getch(); j9=1; \
                  printf("( Continuing %s ) \n",cur0); \
                }

int o[NUM]; /* 是否已排序标志 */
char sor[NUM],sor1[NUM]; /* 按关键字顺序确定排序先后 */
/* 记录排序参数 对-x用小写字母表示 */
int ns; /* 记录下次分段排序对数 */
int mmns[2 * MAXELE]; /* 每两个元素记录一个排序段范围 */
int fol[MAXELE],dow[MAXELE]; /* fol记录前次排序后上下元素相同情况,
dow则记录本次排序情况,为下次排序作准备 */
int nn; /* 中间变量,记录相同元素个数 */
int attr,flag;
int sl=CAPS,sn,sc,sd,ss,sg,h=0;
int stop; /* 纵:0=不停,1=停;横:2=不停,3=停 */
int os; /* os=0或者/o 开关后无参数,表示未用/o 排序开关 */
int s-s; /* 1=对子目录中全部子目录查找,2=全盘查找标志 */
int vf; /* 1=显示卷标与磁盘剩余空间 */
int fn,fn1; /* 找到的文件数、总文件数 */
struct fblk fp;
long attrib[MAXELE];
unsigned ftime[MAXELE];
unsigned long fdate[MAXELE];
long fsize[MAXELE];
char name1[MAXELE][9];
char ext1[MAXELE][5]; /* 用来存储待排序的目录或文件 */
int sb; /* 1=有/B 标志 */
int st; /* st0=1要时间比较,只能一次有效 */

```



```

int stf; /* & = % 时间比较标志 */
long unsigned st1,st2,fz,fzn; /* 两个时间值,文件长度 */
unsigned m[12]; /* 记录时间值 */
char filename[80],path1[MAXDIRNUM][80];
char *dateT(int d), *timeT(int d), *week(int y,int m,int d);
int ff,j9;
main(int argc,char *argv[]) /* 功能强于DOS 5.0内部命令DIR的CDIR */
{
    register int done1,done2;
    char cur0[80],path[80],path-0[MAXDIR];
    char drive[3],subdir[66],file[9],ext[5];
    int level=0,n=0,j,j0,fh,drive-0,je,fg,fk;
    struct dfree de;
    unsigned long freesize,frees;
    struct date d;
    struct time tm;
    char *temp=(char *)calloc(15,1);
    sn=se=sd=ss=sg=ONO;
    getcwd(path-0,MAXDIR); /* 保存当前目录 */
    drive-0=getdisk(); /* 保存当前驱动器号 */
    getdate(&d); /* 取日期与时间作缺省值 */
    gettime(&tm);
    m[2]=m[8]=d.da-year-1980; /* 左边时间为0,右边为当前日期 */
    m[0]=m[6]=d.da-mon;
    m[1]=m[7]=d.da-day;
    m[9]=tm.ti-hour;
    m[10]=tm.ti-min;
    m[11]=tm.ti-sec;
    for (j=0;j<argc;j++) /* 分析命令行 */
        check(j,argv[j]);
    if(filename[1]==' ')
        { getdfree( toupper(filename[0])-'A'+1,&de); }
    else getdfree(0,&de);
    freesize=(unsigned long)(de.df-bsec*de.df-sclus)*
        (unsigned long)de.df-avail;
    if(vf) /* 显示卷标及磁盘剩余字节数 */
    {
        path[0]='\0';
        fnsplit(filename,drive,subdir,file,ext);
        strcat(path,"vol ");
        strcat(path,drive);
        system(path);
        path[0]='\0';
        strcat(path,drive);
        strcat(path,subdir);
        if(!strlen(path))strcpy(path,path-0);
    }
}

```

```

printf(" Directory of %s\n",path);
printf("\t\t%lu bytes free\n\n press Q=exit else other=Continue...\n",freesize);
path[0]=toupper(getch());
if(path[0]=='Q')exit(1);
path[0]='\0';
}
if(filename[1]==',' && filename[2]=='\0') /* 键入像c: 命令处理 */
    if(toupper(filename[0]) == path-0[0]) flag=0;
    else strcat(filename,"\\*.*");
if(!attr){attr=FA-DIREC | FA-ARCH; /* 缺省属性 */
    ff=1;}
if(!flag) /* 没有给出具体文件名时使用缺省值:当前目录里所有文件 */
{
    strcpy(filename,path-0);
    strcat(filename,"\\*.*");
}
if(filename[1]!=':') /* 没有给出路径名时 */
{
    if(filename[0]=='.') /* 文件名为.xxx */
    { strcpy(path1[0],"*");
      strcat(path1[0],filename);
      strcpy(filename,path1[0]);
    }
    strcpy(path1[0],path-0); /* 获缺省目录 */
    strcat(path1[0],"\\");
    strcat(path1[0],filename);
    strcpy(filename,path1[0]);
    memset(path1[0],'\0',80);
}
else if(flag && s-s==2)s-s=1; /* 指定了路径名时忽略全盘查找 */
fnsplit(filename,drive,subdir,file,ext); /* 此处文件名为全名 */
n=strlen(subdir);
switch(s-s)
{
    case 0:
    case 1: /* 在目录(包括子目录)中寻找 */
        strcpy(path,drive);
        if(n>0)subdir[n-1]='\0';
        strcat(path,subdir);
        break;
    case 2: /* 全盘查找 */
        path[0]=filename[0];
        path[1]='\0';
        strcat(path,".");
        break;
}
}

```

```

n=0; strcat(file,ext);
strcpy(filename,file); /* 文件名=基本名+扩展名 */
if(strchr(filename,'*')==NULL && strchr(filename,'?')==NULL)fh=0;
else fh=1;
printf("Search ... \n");
do /* DO 循环开始,查找开始 */
{
if(kbhit()) /* 按任一键结束查找 */
{
getch();
exit(1);
}
fn=0; fz=0;
strcat(path,"\\");
strcpy(path1[n],path);
strcpy(cur0,path);
strcat(path,filename);
done2=findfirst(path,&fp,attr); /* 搜索 */
while(!done2) /* done2循环开始,处理当前目录中的文件 */
{
fg=0;
if(!ff)
{
if(fp.ff-attr != attr && fh)fg=1;
if(attr != (FA-HIDDEN | FA-SYSTEM | FA-RDONLY))
{
if(attr==FA-ARCH && (fp.ff-attr & FA-RDONLY))fg=0;
if(attr & FA-DIREC && (fp.ff-attr & FA-DIREC))fg=0;
if(attr & FA-HIDDEN && (fp.ff-attr & FA-HIDDEN))fg=0;
if(attr & FA-RDONLY && (fp.ff-attr & FA-RDONLY))fg=0;
if(attr & FA-SYSTEM && (fp.ff-attr & FA-SYSTEM))fg=0;
}
}
if(!fg)
{
if(fn==MAXELE){printf("Too many files!\n");
return ;}
attrib[fn]=fp.ff-attr;
fz += fp.ff-fsize; /* 累计文件、目录数与字节和 */
ftime[fn]=fp.ff-ftime;
ldate[fn]=fp.ff-ldate * 86400 + fp.ff-ftime;
fsize[fn]=fp.ff-fsize;
fnsplit(fp.ff-name,drive,subdir,file,ext);
if(fp.ff-name[0]!='.')strcpy(name1[fn],fp.ff-name);
else strcpy(name1[fn],file);
if(ext[0]!='.')

```

```

        for(je=1;je<4;je++)ext1[fn][je-1]=ext[je];
        ext1[fn][3]='\0';
        fn++;
    }
done2=findnext(&fp);
} /* done2循环结束 */
strcpy(path,path1[n]);
strcat(path,"*.*");
done1=findfirst(path,&fp,FA-DIREC);
if(s--s)
{
    while(!done1) /* done1循环开始,记忆各级目录位置 */
    {
        /* 搜索非"."和".."的目录名 */
        if(stricmp(fp.ff-name, ".") && stricmp(fp.ff-name, ".."))
        if(fp.ff-attr==FA-DIREC)
        {
            strcpy(path1[n+1],path1[n]); /* 当前目录后边 */
            strcat(path1[n+1],fp.ff-name); /* 登记找到目录 */
        }
        done1=findnext(&fp);
    } /* done1循环结束 */
    strcpy(path,path1[level++]); /* 获得下次循环的当前目录 */
}
/* 显示预处理 */
if(fn)
{
    printf("Directory of %s\n",cur0); j9++;
    PSTOP;printf("\n");j9++;PSTOP;
    for(j=0;j<fn;j++)
    {
        if(attrb[j]==FA-DIREC) attrb[j]=FA-DIREC; /* 为目录处理 */
        else attrb[j]=FA-ARCH; /* 提供方便 */
        if(os){
            strcpy(sor1,sor); /* 考虑对多个子目录 */
            manysort(sor1,fn); /* 排序 */
        }
        fk=fn;
        for(j=0,j0=1;j<fn;j++,j0++) /* 显示开始 */
        {
            je=0;free=fdate[j];
            if(st1+st2!=0)
            {
                switch(stf)
                {
                    case 0:
                        if(free<st2 && free>st1)je=0;

```

```

        else je=1;
        break;
    case 1:
        if((free != st1))je=1; break;
    case 2:
        if((free<st1 || free>st2))je=0;
        else je=1;
    }
}
if(je)
{
    j0--,n--, fz -= fsize[j]; fk--; continue;
}
temp[0]='\0';
strcat(temp,name1[j]);
if(strlen(ext1[j]))
{
    strcat(temp,"."); strcat(temp,ext1[j]);
}
if(sl==SMALL)strlwr(temp);
if(sb)
{
    if(temp[0]!='.')
        printf("%-15s\n",temp);j9++;
    if(stop & PP)
    {
        if(j9==23){printf("press any key to continue ... \n");
            if(j==fn-1)printf("( Continuing %s )\n",cur0);
            getch();j9=0;}
    }
}
else if(stop & PP || !stop){
    if(stop & 'W')
    {
        if(temp[0]!='.'){printf("[%s] ",temp);
            if(temp[1]!='.')printf(" ");}
        else printf("%-15s",temp);
        if(j0>4)
            {printf("\n");j0=0;j9++;}
    }
    else
    {
        if(sl==SMALL){strlwr(name1[j]);strlwr(ext1[j]);}
        printf("%-10s",name1[j]);
        if(attrb[j] & FA--DIREC) printf(" [DIR]");
        else printf("%-6s",ext1[j]);
        j9++;
        if(! (attrb[j] & FA--DIREC))printf("%-10lu",fsize[j]);
        else printf(" ");
        printf("%-16s",dateT(((date[j]-(time[j])/86400)));
        printf("%-12s\n",timeT(time[j]));
    }
}

```

```

        if(j9==23 && stop)
        { printf("press any key to continue ... \n");
          getch();
          if(j==fn-1)printf("( Continuing %s )\n",cur0);
          j9=0;
        }
      }
    else
    {
      if(temp[0]==' '){printf("[%s]          ",temp);
                       if(temp[1]!=' '){printf(" ");}}
      else printf(" %-15s",temp);
      if(j0>4) {printf("\n");j0=0;j9++; }
    }
  }
  if(st1+st2 !=0)fn=fk;
  printf("\n");PSTOP;
  if(fn)printf("\t%d file(s)   %lu bytes\n",fn,fz);j9++;
  PSTOP;
  fnn +=fn; fzn +=fz;
}
for(j=0;j<NUM;j++)o[j]=0; /* 恢复要排序标志 */
}while(level<=n && s==s); /* DO 循环结束.只有当s==0才循环 */
if(fnn)
{
  if(fnn > fn)
  {
    PSTOP;
    printf("Total files listed:");j9++;
    PSTOP;
    printf("\n\t%d file(s) %lu bytes\n",fnn,fzn);j9++;
  }
  PSTOP;
  printf("\t\t%lu bytes free\n",freesize);
}
else printf("File not found !\n");
setdisk(drive-0); /* 恢复原来驱动器 */
chdir(path-0); /* 恢复原来路径 */
}
check(int jj,char *s) /* 检查命令行 */
{
  if(*s=='/')commparser(s);
  else {
    if (jj)
    {
      if(flag==1)

```

```

        {printf("too many parameters -- %s\n", s);
        exit(1);
        }
    /* 取文件名, 命令行上只允许出现一次文件名 */
    else {
        strcpy(filename, s);
        flag=1;
    }
}
else {
    while (*s++)
    {
        if (*s != '/') {if (*s == '\0') break;}
        else commparser(s);
    }
}
}

char *dateT(int d) /* 将磁盘目录中记录的日期d 转换为实际日期 */
{
    int i;
    char str[10], *p="{ " ;
    char *s[]={"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
    unsigned weekday, year, month, day;
    year=(d>>9 & 0xf)+1980; /* 或d=(year-1980)*512+month*32+day */
    month=d>>5 & 0xf;
    day=d & 0x1f;
    p=week(year, month, day);
    for(i=0; i<7; i++) /* 此处编译时出现警告, 没关系 */
        if(!strcmp(s[i], p)) {weekday=i; break;} /* 必要时可用weekday 印出星期 */
    strcat(p, " ");
    if(month<10)strcat(p, "0");
    strcat(p, itoa(month, str, 10));
    strcat(p, "-");
    if(day<10)strcat(p, "0");
    strcat(p, itoa(day, str, 10));
    strcat(p, "-");
    strcat(p, itoa(year, str, 10));
    return (p);
}

char *week(int y, int m, int d) /* 求星期几 */
{
    struct date d1; /* 0 = 星期日 Sunday */
    struct time t1; /* 1 = 星期一 Monday */
    long td; /* 2 = 星期二 Tuesday */
    /* 3 = 星期三 Wednesday */
    /* 4 = 星期四 Thursday */
    /* 5 = 星期五 Friday */
    char *p=(char *)malloc(25);
    d1.da-year=y;

```

```

d1.da--mon=m; /* 6 = 星期六 Saturday */
d1.da--day=d;
t1.ti--min=t1.ti--hour=t1.ti--hund=t1.ti--sec=0;
td=dostounix(&d1,&t1);
p=ctime(&td);
p[3]='\0';
return (p);
}

char *timeT(int d) /* 将磁盘目录中记录的时间d 转换为实际时间 */
{
char p[20],str[10];
unsigned hours,minutes,second;
hours=d>>11&0x1f; /* 或d=hours*2048+minutes*32+second/2 */
minutes=d>>5&0x3f;
second=(d&0x1f)*2;
if( hours<12 || (hours==12&& minutes==0&& second==0))
strcpy(p,itoa(hours,str,10));
else strcpy(p,itoa(hours-12,str,10));
strcat(p,".");
if(minutes<10)strcat(p,"0");
strcat(p,itoa(minutes,str,10));
strcat(p,".");
if(second<10)strcat(p,"0");
strcat(p,itoa(second,str,10));
if( hours<12 || (hours==12&& minutes==0&& second==0))
strcat(p,"a");
else strcat(p,"p");
return (p);
}

help() /* 帮助 */
{
char c;
clrscr();
printf(" * * * * * 命令行输入方法 * * * * *\n\n");
printf(" CDIR [drive:][path][filename] [/P] [/W] [/A[:]att\n\n");
printf(" ributes]]\n");
printf(" p[/O[:]sortorder] [/S[:]folower] [/B] [/L][/?][H]\n");
printf(" [/T[:]comparator]][/V][F]\n\n");
printf("drive 驱动器名,path 路径名,filename 文件名(可含通配符)\n");
printf("/P 显示满屏时暂停,/W 每行列5个文件(基本名+扩展名)\n");
printf("/A 用于指定待显示文件的属性(如带前缀“-”表示“非”)\n");
printf("attributes: D 目录,R 只读,H 隐含,A 归档,S 系统\n");
printf("/O 用于指定按某种顺序显示文件(如带前缀“-”表示“非”)\n");
printf("sortorder: N 按文件基本名,E 按文件扩展名\n");
printf(" S 按文件大小, D 按文件日期与时间\n");
printf(" C 先目录后文件\n");

```



```

printf("/S 显示指定目录里的文件(无/S 时只在当前目录里查找)\n");
printf("follower: G 当前目录中(包括它子目录里)的文件\n");
printf("          B 全盘查找\n");
printf("/B 每行显示一个文件名(只有基本名+扩展名,无文件时间、长度等)\n");
printf("/L 用小写字母显示文件名(无/L 时用大写字母)\n");
printf("/T 用于列出指定日期与时间范围内的文件(无/T 时不比较)\n");
printf("comparator: 月-日-年,时-分-秒 & 或=或% 月-日-年,时-分-秒\n");
printf("/V 或 /F 显示卷标与磁盘剩余空间(无/V 则不显示)\n");
printf("/? 或 /H 显示本帮助。按 Q 键退出,按回车键继续\n");
c=toupper(getch());
if(c=='Q')exit(1);
}

commparser(char *s) /* 分析命令行 */
{
    char u[2], t[128], *t=tt;
    int a,b;
    unsigned long temp;
    strcpy(tt,s);
    t=strupr(t);
    t++;
    if(*t=='\0')return;
    do {
        switch(*t)
        {
            case '?':
            case 'H':
                help();
            case '/':break;
            case 'V':
            case 'F':
                vf=1;break;
            case 'S':
                t++;
                if(*t=='/')break;
                do {
                    if(*t=='\0')break;
                    switch(*t)
                    {
                        case 'G':s--s=1;break;
                        case 'B':s--s=2;break;
                        default:errs(s);
                    }
                }while(*t++!='');
                t--;break;
            case 'A':
                t++;

```

```

if(*t == '/')break;
do {
    if(*t == '\0')break;
    if(*t != '-')
    {
        switch(*t)
        {
            case 'H':
                attr |= FA-HIDDEN;break;
            case 'S':
                attr |= FA-SYSTEM;break;
            case 'D':
                attr |= FA-DIREC;break;
            case 'A':
                attr |= FA-ARCH;break;
            case 'R':
                attr |= FA-RDONLY;break;
            case ',':break;
            default;errs(s);
        }
    }
} else
{ t++;
    if(*t == '\0')break;
    switch(*t)
    {
        case 'H':
            attr &= ~FA-HIDDEN;break;
        case 'S':
            attr &= ~FA-SYSTEM;break;
        case 'D':
            attr &= ~FA-DIREC;break;
        case 'A':
            attr &= ~FA-ARCH;break;
        case 'R':
            attr &= ~FA-RDONLY;break;
        case ',':break;
        default;errs(s);
    }
}
}while(*++t != '/');
t--;break;
case 'P':
    stop |= PP;break;
case 'W':
    stop |= WW;break;

```

```

case 'B',
    sb=1, break;
case 'L',
    sl=SMALL, break;
case 'O',
    t++,
    if(*t == '/')break;
    do {
        if(*t == '\0')break;
        if(*t != '-')
        {
            os=1;
            switch(*t)          /* 大写记录 n */
            {
                case 'N',
                    if(sn == ONO){sn = OYES; sor[h++]='N';}break;
                case 'E',
                    if(se == ONO){se = OYES; sor[h++]='E';}break;
                case 'D',
                    if(sd == ONO){sd = OYES; sor[h++]='D';}break;
                case 'S',
                    if(ss == ONO){ss = OYES; sor[h++]='S';}break;
                case 'G',
                    if(sg == ONO){sg = OYES; sor[h++]='G';}break;
                case ', ', break;
                default, {os=0; errs(s);}
            }
        }
    }
else
    {
        t++;
        if(*t == '\0')break;
        os=1;
        switch(*t)
        {
            case 'N',          /* 小写记录 -n */
                if(sn == ONO){sn = ONOT; sor[h++]='n';}break;
            case 'E',
                if(se == ONO){se = ONOT; sor[h++]='e';}break;
            case 'D',
                if(sd == ONO){sd = ONOT; sor[h++]='d';}break;
            case 'S',
                if(ss == ONO){ss = ONOT; sor[h++]='s';}break;
            case 'G',
                if(sg == ONO){sg = ONOT; sor[h++]='g';}break;
            case ', ', break;
            default, {os=0; errs(s);}
        }
    }

```

```

    }
}while( * ++t != '/');
t--;break;
case 'T':
    q=b=0; stt= !stt;t++;
    if( * t == '/')break;
    do {
        if( * t == '\0')break;
        switch( * t)
        {
            case ':':break;
            case '-':if(stt)q++;break;
            case ',':if(stt)q+=3;break;
            case '&':if(stt){q=6;stf=0;}break;
            case '=':if(stt){
                stf=1;m[8]=m[2];m[6]=m[0];m[7]=m[1];
                m[9]=m[3];m[10]=m[4];m[11]=m[5];
            }break;
            case '%':if(stt){q=6;stf=2;}break;
        default:
            if( !isdigit( * t))errs(s); /* 非数字则错 */
            if(stt)
            { u[b++] = * t;
              t++; /* 允许数字后缺省 */
              if(b==2 || * t=='-' || * t=='\0' || * t=='+' ||
                * t=='&' || * t=='=' || * t=='%')
              {
                  if(u[0]=='0'){u[0]=u[1];u[1]='\0';}
                  b=atoi(u);
                  switch(q)
                  {case 0:
                      case 6:
                          if(b<13 && b>0)m[q]=b;
                          break;

                      case 1:
                      case 7:
                          if(b<32 && b>0)m[q]=b;
                          break;

                      case 2:
                      case 8: if(b>79)m[q]=b+1900-1980;
                          break;

                      case 3:

                      case 9:
                          if(b<25)m[q]=b;
                          break;

                      case 4:

```

```

        case 10:
        case 5:
        case 11:
            if(b<61)m[q]=b;
            break;
        }
        b=0;u[0]=u[1]='\0';
    }
    t--;
}

}
}while(*++t != '/' && stf !=1);
st1=(m[2]*512+m[0]*32+m[1])*86400+m[3]*2048+m[4]*32+m[5]/2;
st2=(m[8]*512+m[6]*32+m[7])*86400+m[9]*2048+m[10]*32+m[11]/2;
if(st1==st2)stf=1;
else if(st1>st2) /* 保证st1<st2 */
    { temp=st1; st1=st2; st2=temp; }
t--;break;
default;errs(s);
}
} while(*++t != '\0');
}
errs(char *tt) /* 错误处理 */
{
printf("Invalid switch - /%s\n",++tt);
exit(1);
}
manysort(char *sor,int n) /* 排序 */
{
/* directions=1,小写字母表示正向排序, */
int u,c=0,directions;
ns=1;
mmns[1]=n-1;
for(c=0;c<strlen(sor);c++) /* 多重排序 */
{
if(sor[c]=='\0')return;
directions=! (islower(sor[c]));
sor[c]=tolower(sor[c]); /* 一律变成小写字母处理 */
switch(sor[c])
{
case 'n':
if(!ns)break; /* 没有一个相同则不再排序 */
for(u=0;u<ns;u++)
qs(name1,mmns[u*2],mmns[u*2+1],directions);
o[0]=1;comps(name1,0,n);
break;
case 'e':

```

```

        if(!ns)break;
        for(u=0;u<ns;u++)
            qs(extl,mmns[u*2],mmns[u*2+1],directions);
        o[1]=1;comps(extl,0,n);
        break;
    case 'g':
        if(!ns)break;
        for(u=0;u<ns;u++) qn(attrib,mmns[u*2],mmns[u*2+1],directions);
        o[2]=1;compn(attrib,0,n); break;
    case 'd':
        if(!ns)break;
        for(u=0;u<ns;u++) qn(fdate,mmns[u*2],mmns[u*2+1],directions);
        o[3]=1; compn(fdate,0,n); break;
    case 's':
        if(!ns)break;
        for(u=0;u<ns;u++) qn(fsize,mmns[u*2],mmns[u*2+1],directions);
        o[4]=1; compn(fsize,0,n); break;
    }
} /* 多重排序结束 */
}
qn(unsigned long *numeral,int left,int right,int direction)
{
    /* 对数字使用Quicksort 排序 */
    int i,j;
    unsigned long x,y;
    i=left,j=right;
    x=numeral[(left+right)/2];
    do {
        if(direction)
        {
            /* 逆向排序 */
            while(numeral[i]>x && i<right) i++;
            while(x>numeral[j] && j>left) j--;
        }
        else
        {
            /* 正向排序 */
            while(numeral[i]<x && i<right) i++;
            while(x<numeral[j] && j>left) j--;
        }
    } while(i<=j);
    if(i<=j) {
        swasn(i,j);
        i++;j--;
    }
} while(i<=j);
if(left<j)qn(numeral,left,j,direction); /* 递归 */
if(i<right)qn(numeral,i,right,direction); /* 递归 */
}
qs(char literal[][9],int left,int right,int direction)

```

```

}          /* 对正文使用Quicksort 排序 ,sflag=1区分大小写 */
int i,j;
char *x=(char *)malloc(9);
i=left;j=right;
strcpy(x,literal[(left+right)/2]);
do {
    if(direction)
        /* 逆向排序 */
        while(strcmp(literal[i],x)>0 && i<right)i++;
        while(strcmp(literal[j],x)<0 && j>left)j--;
    }
    else
        /* 正向排序 */
        while(strcmp(literal[i],x)<0 && i<right)i++;
        while(strcmp(literal[j],x)>0 && j>left)j--;
    }
if(i<=j) {
    swasn(i,j);
    i++;j--;
}
}while(i<=j);
if(left<j)qs(literal,left,j,direction);
if(i<right)qs(literal,i,right,direction);
}
swasn(int i1,int j1) /* 交换元素 */
{
long x;
char *y=malloc(9);
if(!o[0]) { strcpy(y,name1[i1]); strcpy(name1[i1],name1[j1]);
strcpy(name1[j1],y); }
if(!o[1]) { strcpy(y,ext1[i1]); strcpy(ext1[i1],ext1[j1]);
strcpy(ext1[j1],y); }
if(!o[2]) { x=attrib[i1]; attrib[i1]=attrib[j1]; attrib[j1]=x; }
if(!o[3]) { x=fdate[i1]; fdate[i1]=fdate[j1]; fdate[j1]=x; }
if(!o[4]) { x=fsize[i1]; fsize[i1]=fsize[j1]; fsize[j1]=x;
x=ftime[i1]; ftime[i1]=ftime[j1]; ftime[j1]=x; }
}
comps(char *s[],int e,int k) /* 对正文分析排序后元素 */
{
/* sflag=1 区分大小写 */
int yy,ii,jj,d1;
char *ss=(char *)malloc(9);
memset(mmins,'\0',2*MAXELE);
strcpy(ss,s[e]);
nn=ii-1; ns=jj=d1=0;
for (yy=e+1;yy<k;yy++)
{

```

```

if(!strcmp(ss,s[yy]) && fol[yy]==fol[yy-1]) /* 相同 */
{ if(!jj)++ns;
  mmns[nn]=yy;
  jj=1;
  dow[yy]=d1;
}
else
{ jj=0;
  if(++ii==2){ns++;mmns[nn]=yy-1;ii=1;
}
  mmns[++nn]=yy;
  nn++;
  dow[yy]=++d1;
}
strcpy(ss,s[yy]);
if(yy==k-1 && ++ii==2)mmns[nn]=yy;
}
for(yy=0;yy<k;yy++)fol[yy]=dow[yy];
}
compn(long *ut,int e,int k) /* 对数字分析排序后元素 */
{
int yy,ii,jj,d1;
long t1=nt[e];
nt++;memset(mmns,0,2*MAXELE); /* 必须初始化 */
nn=ii=1; ns=jj=d1=0;
for (yy=e+1;yy<k;yy++)
{
if(t1==*nt && fol[yy]==fol[yy-1]) /* 相同 */
{ if(!jj)++ns;
  mmns[nn]=yy;jj=1;
  dow[yy]=d1;
}
else
{ jj=0;
  if(++ii==2){ns++;mmns[nn]=yy-1;ii=1;}
  mmns[++nn]=yy;
  nn++;
  dow[yy]=++d1;
}
t1=*nt++;
if(yy==k-1 && ++ii==2)mmns[nn]=yy;
}
for(yy=0;yy<k;yy++)fol[yy]=dow[yy];
}

```


第三十章 文件管理

30.1 缓冲型文件系统和非缓冲型文件系统

Turbo C 没有用于输入 / 输出 (I/O) 的关键字, 但它在标头文件 `stdio.h` 中说明了一组完整的遵守 ANSI 标准的 I/O 库函数。这些库函数构成了所谓的缓冲型文件系统。为了兼顾旧的 UNIX 标准, 它又在 `io.h` 中说明了一组库函数, 它们构成所谓的非缓冲型文件系统。非缓冲型文件系统现在很少使用。为了方便大量的控制台 (通常指键盘和屏幕) 操作, Turbo C 又将缓冲型文件系统中一些专用的子系统 (函数) 的说明放在 `conio.h` 中, 这些函数用于标准 I/O。要使用这些 I/O 库函数, 只要在源程序中包含它们原型所在的标头文件即可。涉及三个标头文件中除包含一类库函数原型说明外, 还有一些预定义类型和常数等。出于叙述完整性考虑, 本章还收入了标头文件 `dos.h` 和 `stat.h` 中部分库函数。

1. `io.h`

非缓冲型文件 I/O 类, 该类函数又称低级 I/O 函数。非缓冲型文件系统 (unbuffered file system) 又称非格式 (unformatted) 文件系统或 UNIX 型 (UNIX-like) 文件系统, 它仅仅是由 UNIX 标准定义的, ANSI 标准没有定义它们。虽然非缓冲型文件系统应用越来越少了, 但 Turbo C 支持它 (不过 Turbo C 侧重于 ANSI 标准的 I/O 系统即缓冲型 I/O)。

2. `stdio.h`

缓冲型文件 I/O 类。缓冲型文件系统 (buffered file system) 又称格式 (formatted) 文件系统或高级 (high-level) 文件系统, 也是 ANSI 标准定义的标准 I/O 系统。

3. `conio.h`

控制台 I/O 类, 它由缓冲型文件系统中一些更一般性的函数以特殊的形式所组成的。如果没有重定向, 对 MS-DOS 控制台是指键盘、屏幕等。

为便于比较, 下面叙述时将它们混合起来说明, 并在序号前冠以一个字母加以区别。

i	<code>io.h</code>	c	<code>conio.h</code>	t	<code>stat.h</code>	g	<code>signal.h</code>
s	<code>stdio.h</code>	d	<code>dos.h</code>	b	<code>bios.h</code>	p	<code>process.h</code>

30.2 C 语言的 FILE 结构剖析

缓冲型文件系统可支持多种不同的设备, 如键盘、屏幕、驱动器等等。为了缓冲, Turbo C 使用了【流, stream】。

流是一个指向文件控制结构 FILE 的指针所管理的磁盘文件或设备 (也称为设备文件)。这就是说, 每一个流都有一个指向 FILE 的指针。下面我们还可以看到, FILE 实际是管理内存中一个数据 I/O 的缓冲区。另一方面, 流必须与一个文件相联。图 30-1 给出了流与文件、输入或输出 (用 \leftarrow 或 \rightarrow 表示) 之间的相互关系示意图。

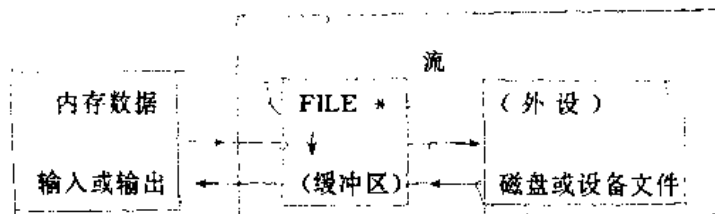


图 30-1

流必须在对其 I/O 之前先行用 `fopen()`、`fdopen()` 或 `freopen()` 等函数打开与流相联的文件。打开流时文件就与 `FILE` 联系在一起,并且系统自动调用 `malloc()` 给流的缓冲区分配 `BUFSIZ`(512 字节)的存储区。

流是 Turbo C 读写数据时一种简便的工具,能使 I/O 操作灵活方便。由于流的引入,与它相关的文件便不再专指磁盘文件,还可以是物理外设(如键盘、打印机和显示器等)。现在, Turbo C 的所有文件不一定具有相同的功能了!例如,一个磁盘文件可以允许随机存取,而一个指向 `FILE` 型结构的指针所管理的外设就不行了。事实上,与流相关的磁盘文件也是通过 `FILE` 型文件控制结构相联的。因此,可以说一个流就是一个指向结构类型为 `FILE` 的文件。结构 `FILE` 在标头文件 `stdio.h` 中定义为

```
typedef struct {
    short      level; /* fill/empty level of buffer */
    unsigned   flags; /* File status flags */
    char       fd;     /* File descriptor */
    unsigned char hold; /* Ungetc char if no buffer */
    short      bsize; /* Buffer size */
    unsigned char *buffer /* Data transfer buffer */
    unsigned char *curp; /* Current active pointer */
    unsigned   istemp; /* Temporary file indicator */
    short      token; /* Used for validity checking */
    }FILE; /* This is the FILE object */
```

目前不少有关 C 语言的工具书都没有对此作更详细的介绍,致使用户有时不免感到困惑。换言之,欲要对流有很好的了解,就必须对 `FILE` 结构的成员和使用方法有更多的了解。

注意:不同版本的 C 语言可能对 `FILE` 的定义不一样。

一 FILE 的使用方法

在用户源程序中,除了首先应有包含文件语句 `#include "stdio.h"` 外,常用像

```
FILE *stream; /* 定义一个指向 FILE 的指针 stream */
stream=fopen("MY.TXT","r"); /* 指针与文件相联,打开了一个流 */
```

语句。这里 `fopen` 是一类与 `FILE` 相关的函数(凡 `stdio.h` 中函数形参前有 `FILE` 定义者均与 `FILE` 相关)。当然,根据结构类型的含意,你可以用像

```
FILE file;
file *stream;
stream=fopen("MY.TXT","r");
```

这样的语句,但对 `FILE` 而言,这种拷贝是不允许的。同时也不允许重新用

```
typedef struct { ..... }FILE;
```

来建立 `FILE`。从下面的程序中还可以看到,指向 `FILE` 的指针和 DOS 文件句柄是有联系的。

但是两者有着不同的概念。另外,在对程序跟踪调试时,也不能用像

`FILE ,r`

这样错误的调试表达式,而只能对指向它的指针使用调试表达式。

二 对 FILE 的成员说明

1. level

它是记录已打开流的缓冲区填入数据情况的变量,流在其被执行前,必须先用像 `fopen()`、`fdopen()` 或 `freopen()` 那样的函数打开。流一经打开,便自动分配一缓冲区,且 `level=0`。例如,当用像 `fgetc()` 或 `fgets()` 那样的函数从与流相联的文件中取得数据进入流时, `level>=0`。预读入缓冲区的数据个数为 `bsize1(<=bsize)`, `bsize` 缺省为 512。也可用 `setbuf()`、`setvbuf()` 重新指定新的缓冲区, `level<=bsize1`。每读入一个字符,其值减 1(对文本方式要少减换行符个数);与此同时, `FILE` 的 `curp` 将发生变化,字符总是读入 `curp` 所指的存储单元内。当用像 `fputc()`、`fputs()` 那样的函数向与流相联的文件输出数据时, `level<=0`。其处理过程与读入数据类同(如可为 `-1-bsize`,或将当前 `level` 的值减 1 等)。

在与另一些与 `FILE` 相关的函数被执行时,也可能引起 `level` 发生变化。如 `fclose()`、`fflush()` 与 `fseek()` 等。

2. flags

这是记录文件状态标志,可以从 `stdio.h` 中定义的和它相关的符号常量中得知其用处:

```
#define _F_RDWR 0x0003 /* 读/写标志 */
#define _F_READ 0x0001 /* 只读文件 */
#define _F_WRITE 0x0002 /* 只写文件 */
#define _F_BUF 0x0004 /* Malloc (分配主存函数) 的缓冲数据 */
#define _F_LBUF 0x0008 /* 行缓冲文件 */
#define _F_ERR 0x0010 /* 错误标识 */
#define _F_EOF 0x0020 /* EOF (文件结束) 标识 */
#define _F_BIN 0x0040 /* 二进制文件标识 */
#define _F_IN 0x0080 /* 数据输入 */
#define _F_OUT 0x0100 /* 数据输出 */
#define _F_TERM 0x0200 /* 文件是终端设备 */
```

从常量值的分布可以看出,它们各自占用了 `flags` 两字节的某一特定位置,因此它们可以进行或操作(|),即组合使用。`fflush()`、`fgetc()`、`fclose()`、`fseek()` 等许多函数调用时都可能改变它的值。

3. fd

与流相联的文件标识符,相当于 DOS【文件句柄】(即给文件分配的一个通道号)。虽然 DOS 文件句柄常为一整数,而 `fd` 被定义为 `char`,但这不会发生问题,因为对 Turbo C 来说,整型和字符可自动转换。`fd` 被定义为 `char`,也说明它最多只能打开 128 个文件(ASCII 码值 0~127,但 0~4 已为外设占用,故用户只能占用 5~127)。每当打开一个新文件,一般该文件对应的 `fd` 上增 1(初始时为 5;对应流被关闭后它被置于空格符),这一过程是自动进行的。要打开更多的文件,宜用别的方法。

在 `io.h` 中定义了一个和文件句柄相关的宏

```
#define HANDLE_MAX 20U
```

表示最大文件句柄数为 20。同样,在 `stdio.h` 中也定义了

```
#define OPEN—MAX 20
```

```
#define SYS—OPEN 20
```

MS-DOS 的最大限制是用户同时只能打开最多 20 个文件句柄。当 MS-DOS 装入一个程序后,它把文件句柄放入程序段前缀 PSP 中,而 PSP 最多只能容纳 20 个文件句柄。注意,对 DOS 3.3 及以上版本,由于使用了 DOS 中断 INT 21H 的功能 67H 而取消了这个限制。

4. hold

在流缓冲区为空 (level=0) 时,可以由 ungetc() 函数回退一个字符到输入流中,hold 记录回退字符。如程序 FILE0.C 所示。

```
C>TYPE FILE0.C
#define "stdio.h"
#define P ch=fgetc(fp);printf("ch=%c\n",ch)
main()
{
FILE *fp;
static char ch,c='B'; /* 以下为调试表达式的值 */
fp=fopen("e:x.c","r"); /* fp[0]:{0,5,'x5','x0',512,"", "", 0,612} */
P; /* fp[0]:{0,133,'x5','x0',512,"A","", 0,612} */
ungetc(c,fp); /* fp[0]:{1,133,'x5','B', 512,"A","B",0,612} */
P; /* fp[0]:{0,133,'x5','B', 512,"A","", 0,612} */
fclose(fp); /* fp[0]:{0, 0,' ', 'B', 0,"A","", 0,612} */
}/* 假定文件 e:x.c 中只有一个字符 A,则程序输出
    ch=A
    ch=B
*/
```

5. bsize

缓冲区大小,缺省为 512 字节,可用 setbuf() 或 setvbuf() 使流与用户指定的缓冲区相联,从而改变流缓冲区 bsize 的大小。

6. buffer

它的值是流缓冲区的首址,即是一个指向流缓冲区的指针。当缓冲区变动时,其它值也将随之变动。

7. curp

缓冲区当前激活的指针。读写流数据时一般是读写 curp 所指的字符,读写时 curp 也随之发生变化。

8. istemp

临时文件标识符,当临时文件产生时,其值为临时文件名某位上的值。产生的临时文件名将不同于当前目录里任何文件名(参见程序 FILE1.C)。

```
C>TYPE FILE1.C
```

```
#include "stdio.h"
```

```
#include "io.h"
```

```
main()
```

```
{
```

```
FILE *fp;
```

```
char *sptr;
```

```

sptr=tmpnam("e:test.c"); /* 产生临时文件 TMP1. $$$ */
printf("%s\n",sptr);
fp=tmpfile(); /* 产生临时文件 TMP2. $$$ */
fputs("C",fp); /* fp->istemp,x:0x2 */
fclose(fp); /* 关闭文件后,TMP1. $$$ 和 TMP2. $$$ 被删除 */
} /* fp->istemp=0, 用 exit() 也要删除临时文件 */
/* 但 abort() 不会删除临时文件 */

```

9. token

常用于文件有效性检查。一些与 FILE 相关的库函数在执行时经常先检查

`fp->token == (short)fp;`

是否为真。若两者不等,则出错返回。由此可知 `fp->token` 中存放指针 `fp` 的值。

读者可以从程序 FILE2.C 中获得更多对 FILE 的理解,还可以理解当前文件位置指示器 (或称文件指针) 的含义。

在调试该程序时你可以用像 `count`、`fp[0]`、`x`、`fp[0]`、`ch`、`fch`、`c`、`fp->buffer` 和 `fp->curp` 这样一些调试表达式进行观察。

```

C>TYPE FILE2.C
#include "stdio.h"
#include "io.h"
extern int _stdinStarted;
static void pascal near FlushOutStreams(void)
{
register FILE *fp;
register int Ndx;
for(Ndx=OPEN--MAX,fp=_streams,Ndx--,fp++)
    if((fp->flags & (_F--TERM | _F--OUT)) == (_F--TERM | _F--OUT))fflush(fp);
}
static int near pascal _fill(FILE *fp) /* 填充预读缓冲区 */
{
long fch,ch;
if(fp->flags & _F--TERM)FlushOutStreams();
if((fp->level=read(fp->fd,(fp->curp=fp->buffer),fp->bsize))>0)
{
/* 装满缓冲区 */
fch=ftell(fp);
ch=tell(fp->fd); /* 调试时可观察 fch 与 ch 的异同 */
fp->flags &= ~_F--EOF;
return 0;
}
else if (0==fp->level)
{ fp->flags=(fp->flags & ~(_F--IN | _F--OUT)) | _F--EOF; }
else
{
fp->level=0;
fp->flags |= _F--ERR;
}
return -1;
}

```

```

}
int fgetc(register FILE *fp) /* 从与流相联的文件中读取一个字符 */
{
    /* 或者简单地说,从流中读取字符 */
    unsigned char c;
    getAgain:
    if(!--(fp->level)>=0)
    return((unsigned char)(++fp->curp)[-1]); /* 流的当前指针指向所取字符 */
    if(++(fp->level)<0 || fp->flags & (-F-OUT | -F-ERR))
    {
        fp->flags |= -F-ERR;
        return EOF; /* 是写方式,出错返回 */
    }
    restartStdin:
    fp->flags |= -F-IN;
    if(fp->bsize != 0)
    {
        /* 必须 level=0 */
        if(!fill(fp)) /* 有缓冲区时 */
        {
            return EOF;
            goto getAgain;
        }
    }
    else
    {
        (! !stdinStarted && ((short)fp == (short)stdin));
        if(! isatty(fp->fd)) /* 检查是否为设备 */
        {
            fp->flags &= ~-F-TERM;
            setvbuf(fp, NULL, (fp->flags & -F-TERM) ? -IOLBF : -IOFBF, BUFSIZ);
            goto restartStdin;
        }
        do
        {
            if((fp->flags & -F-TERM) FlushOutStreams());
            if(1 != read(fp->fd, &c, 1)) /* 读一个字符到变量 c 中 */
            {
                if(1 != eof(fp->fd)) fp->flags |= -F-ERR;
                else fp->flags = (fp->flags & ~(-F-IN | -F-OUT)) | -F-EOF;
                return EOF;
            }
        } while ('\\r' == c && ! (fp->flags & -F-BIN)); /* 对文本方式处理 */
        fp->flags &= ~-F-EOF;
        return ((unsigned char)c);
    }
}
main()
{
    FILE *fp;

```

```

int i, count = 0;
char c = 0;
long fch, ch;
fp = fopen("e.y.c", "t");
fch = ftell(fp);
for(i = 0; i < 494; i++)
{
    count++;
    /* 起初可用热键 F7 调试,以便进入各函数 */
    c = fgetc(fp);
    ch = tell(fp -> fd);
    fch = ftell(fp);
    printf("%c", c);
}
for(i = 0; i < 200; i++) /* 在此句可用 Ctrl-F8 键设一断点,并用 Ctrl-F9 执行 */
{
    count++;
    /* 然后用 F7 单步调试,必要时为加快可用 F8 调试 */
    c = fgetc(fp);
    ch = tell(fp -> fd);
    fch = ftell(fp);
    printf("%c", c);
}
fclose(fp);
}

```

调试前如先进行编译,可能会出现这样的警告:

'ch'(或'fch') is assigned a value which is never used

可不管它! 在调试中可用断点、F7 或 F8 综合进行调试。通过调试表达式的值观察程序的执行过程,特别是流的内容被填充和 FILE 成员的变化情况,从中可以看出使用流的一些优点。即便于检查非法读写,从而免受危害。同时还可以看到,当出错时返回值 -1 的好处,即不会输出有意义的可打印字符。此外,比较程序中两语句

```

read(fp -> fd, (fp -> curp = fp -> buffer), fp -> bsize);
read(fp -> fd, &c, 1);

```

它们都是从与流相联的文件中读取字符,并随之改变当前文件位置指示器的值。注意,对文本方式,read() 在读入时删除回车符。在调试中还可以看到 ch、fch 的变化(值不一定相同),明了函数 tell() 和 ftell() 所得结果是不同的。前者是与流相联的文件位置指示器的当前值,后者是 ch 减去遇到的换行符的个数的差。而且 fp->level 也要减去换行符个数;对二进制方式,则没有这种减的过程。

30.3 文本流与二进制流

有两种类型的流:文本流(text stream)和二进制流(binary stream)。

文本流用于普通的 DOS 文本文件。假设文本文件有若干行,一般存在磁盘上的 DOS 文本文件的行尾有 ASCII 码回车和换行符(CR/LF,即 0DH 和 0AH),用于和下一行的分隔。用文本流将磁盘文件输入内存时,Turbo C 将两个字符 CR/LF 转换为一个字符 LF(换行符);而在输出时,则将 LF 又换为 CR/LF 对。换句话说,读写字符是变化的。

二进制流就不存在这种CR/LF转换,即读写的字符个数不会改变。二进制流输出时有可能(由工具程序决定)在磁盘文件末尾增加一些空字节,致使文件最后占满一个扇区。

对磁盘上的一个文件 Turbo C 自然不知道原先是用文本方式即通过文本流记录的还是用二进制方式(流)记录的?因此,由于这种转换的不慎,对于一些字处理软件生成的文本在转换时可能会多出现一个 0DH,这是要注意的。必要时可编一程序专门滤去这个多余的 0DH。另外,当你某些文件不能确定是用哪一种方式记录时,在只读时可用二进制方式打开较好。如对数据库文件 *.DBF 就是这样。因为它的文件中可能会出现如 1AH 那样的字节,否则用文本方式处理时很可能出错(如发现某些数据未被读入等等)。

若没有指定用哪种方式打开流时则以全局变量 `_fmode` 指定的缺省方式打开,而 `_fmode` 本身的缺省方式为文本方式。

30.4 标准 I/O 预定义流 (Standard I/O predefined streams)

在 `stdio.h` 中定义了 5 个预定义流的宏:

```
extern FILE _Cdecl _streams[]; /* 结构数组 */
#define stdin (&_streams[0]) /* 标准输入设备,一般指键盘 */
#define stdout (&_streams[1]) /* 标准输出设备,一般指屏幕 */
#define stderr (&_streams[2]) /* 出错信息输出设备,一般指屏幕 */
#define stderr (&_streams[3]) /* 辅助设备,异步通信控制器 */
#define stderr (&_streams[4]) /* 打印机 */
```

注意:定义的 5 个宏名相当于关键字,它们在 Turbo C 中有固定的含义,用户不要改变它们。如果操作系统未重定向, `stdin` 一般从控制台读, `stdout` 和 `stderr` 向控制台写。

对 5 个预定义流一开始并没有分配缓冲区,这可以在包含 `stdio.h` 的文件调试开始时(自然是非重定向)用调试表达式看出。例如有

```
_streams[0],x: {0x0,0x209,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x30C} /* stdin */
_streams[1],x: {0x0,0x20A,0x1,0x0,0x0,0x0,0x0,0x0,0x0,0x31C} /* stdout */
_streams[2],x: {0x0,0x202,0x2,0x0,0x0,0x0,0x0,0x0,0x0,0x32C} /* stderr */
_streams[3],x: {0x0,0x243,0x3,0x0,0x0,0x0,0x0,0x0,0x0,0x33C} /* stderr */
_streams[4],x: {0x0,0x242,0x4,0x0,0x0,0x0,0x0,0x0,0x0,0x34C} /* stderr */
```

容易从 `FILE` 的成员 `flags` 看出,只有 `stdin` 采用了行缓冲方式,其它均不缓冲(`flags` 中不含 `_F—BUF` 或 `_F—LBUF`)。它们都是设备文件(`flags` 含 `_F—TERM`)。一般情况下不要人为地去改变这种缺省状态,而当调用某些函数时,例如调用 `printf()` 时便可看到 `stdout` 的缓冲区发生变化。

除非重定向,否则 `stdin` 和 `stdout` 将连接到控制台。若未重定向, `stdin` 带有行缓冲,而 `stdout` 则不带缓冲。其它三个预定义流(`stderr`、`stderr` 和 `stderr`)也不带缓冲。

向这些设备文件上读写的数据可以重定向到其它文件,本书中在适当的地方将给出对它们重定向的例程。

在一个程序开始执行时, `stdin`、`stdout` 和 `stderr` 将被自动打开(对有些操作系统另外两个 `stderr` 和 `stderr` 也是打开的)。它们和计算机系统的标准输入输出设备相关。在程序结束时它们将自动关闭。换言之,对这 5 个预定义流的打开或关闭用户不必关心。

30.5 文件控制块 FCB

FCB 在 PSP 中的位置参见《程序头前缀 PSP》一章,与其对应的结构 fcb、xfcb 参见《磁盘文件的结构》一章。

一 FCB 格式

偏移量	长度	说明
(以下由 DOS 填满,不应该改变)		
-7H	1 个字节	若为 FFH 则为扩充 FCB
-6H	5 个字节	保留(通常为 0)
-1H	1 个字节	对扩充 FCB 是文件属性
(以下可由程序员填满)		
00H	1 个字节	驱动器编号(0 = 缺省,1 = A, ...)
01H	8 个字节	文件基本名,左对齐,空格在后
09H	3 个字节	文件扩展名
0CH	1 个字	当前块编号。与文件开头相对应的现行块计数,用 0 来起动(用打开功能调用设置的 0)。一块由 128 个记录组成,每个记录的大小指定在逻辑记录大小区域中
0EH	1 个字	逻辑记录长度。打开文件时系统缺省设置为 128 字节。若该值不为 128,则用户在打开文件后在接着的读写之前应将它设置为正确值,因为 DOS 要用它来为所有磁盘读写确定正确的位置。
10H	双字	文件长度(低字节在前,单位:字节)
14H	1 个字	最后一次修改文件后写入的 DOS 日期
16H	1 个字	最后一次修改文件后写入的 DOS 时间
18H	8 个字节	保留
20H	1 个字节	当前块内的记录
21H	双字	随机存取记录号,指向当前选择的记录。从文件的开始(0)计算。这 4 个字节不能用打开文件系统调用来初始化,必须在对文件随机读/写之前设置它。若记录长度大于 64 字节,则 4 个字节都用,否则只用前 3 个字节。

二 利用 FCB 的 DOS 功能调用(现在已很少用)

—1 功能 0FH

用途:使用 FCB 打开文件

入口:AH=0FH(AH 是装入功能号,下同)

DS:DX→指向未打开的 FCB 的指针

返回:AL=00H 文件存在且被打开

FFH 文件不存在或打开失败

—2 功能 10H

用途:使用 FCB 关闭文件

入口:DS:DX→打开的 FCB

返回:AL=00H 关闭成功

FFH 关闭失败

—3 功能 11H

用途:用 FCB 查找第一个相符的文件名

入口:DS:DX→未打开的 FCB

返回:AL=00H 成功

FFH 失败

[DTA] = 第一个相符文件用的未打开 FCB

—4 功能 12H

用途:用 FCB 查找下一个相符文件名

入口:DS:DX→未打开的 FCB

返回:同功能11H

—5 功能 13H

用途:用 FCB 删除文件

入口:DS:DX→未打开的 FCB

返回:AL=00H 成功

FFH 失败

—6 功能 14H

用途:从标明已打开的 FCB 的文件顺序读一个记录 (通常为 128 字节) 到磁盘传输区 (DTA)

入口:DS:DX→打开的 FCB

返回:AL=00H 成功

01H 文件结尾 (无数据)

02H DTA 太小

03H 文件结尾,读出部分记录 (不足部分填 0)

—7 功能 15H

用途:从磁盘传输区写入一个记录 (通常为 128 字节) 到预定的用 FCB 打开的文件

入口:DS:DX→打开的 FCB

[DAT] = 要写的记录

返回:AL=00H 成功

01H 磁盘满

02H DTA 段重叠

—8 功能 16H

用途:用 FCB 建立或截断文件

入口:DS:DX→未打开的 FCB

返回:AL=00H 成功

FFH 失败

—9 功能 17H

用途:用 FCB 给文件换名

入口:DS:DX→修改过的 FCB。修改方法:老的文件名放在标准位置,新的文件名放在自偏移量 11H 开始的 11 个字节内。

返回:AL=00H 成功

FFH 失败

—10 功能 21H

用途:用 FCB 从文件中随机读指定记录

入口:DS:DX→打开的 FCB

返回:AL=00H 成功

01H 文件结尾,已无数据可读

02H DTA 太小,无数据读

03H 文件结尾,读部分数据

[DTA] = 读出的记录

说明:记录是从由 FCB 的随机记录和记录大小字段规定的当前文件位置读出的。读出记录后文件位置不改变。当部分读出时,其余部分由 0 填充

—11 功能 22H

用途:用 FCB 写随机记录到文件中

入口:DS:DX→打开的 FCB

[DTA] = 要写的记录

返回:AL=00H 成功

01H 磁盘空间满

02H DTA 太小,未写

说明:记录按 FCB 规定的随机记录和记录大小字段写入当前文件位置,写入后不改变当前文件位置。如果记录被定位于文件结尾之后,则将文件扩充但插入的数据未进行初始化。如果记录小于扇区空间,则先放于 DOS 磁盘缓冲区,等待以后写入。

30.6 库函数及设备驱动程序

一 文件名

· 换文件名

s—1 rename()

· 分析文件名

d—2 parsfnm()

· 聚合文件名 (参见《目录函数》一章)

fnmerge()

· 分解文件名

fnsplit()

二 文件日期与时间 (参见《日期和时间函数》一章)

· 取文件日期与时间

i—getftime()

· 修改文件日期与时间

i—setftime()

三 文件属性与访问方式

· 查询或改变文件访问权限或属性,只适用于 MS-DOS i—3 —chmod()

· 查询或改变文件访问权限或属性

i—4 chmod()

· 改变已打开文件访问方式 (二进制或文本方式)

i—5 setmode()

四 文件长度

· 得到文件长度

i—6 filelength()

· 改变文件大小

i—7 chsize()

五 查找文件

- 检查路径或文件是否存在、读写允许 i—8 access()

六 创建文件

- 建立新文件同时删了磁盘上同名文件,只适用于MS-DOS i—9 --creat()
- 建立新文件同时删了磁盘上同名文件 i—10 creat()
- 建立文件,但若磁盘上已有同名文件则创建失败 i—11 creatnew()
- 建立一个和磁盘上所有文件不同名的暂时文件,文件属性由用户指定 i—12 creattemp()
- 在磁盘上建立一个扩展名为 . \$ \$ \$ 的暂时文件 s—13 tmpnam()
- 产生并同时打开一个与流相联的暂时文件 s—14 tmpfile()

七 打开文件

- 打开文件供读写,只适用于 MS-DOS i—15 --open()
- 打开文件供读写 i—16 open()
- 和早期版本兼容的打开文件的宏 i—17 sopen()
- 打开一个与流相联的文件 s—18 fopen()
- 打开与文件句柄相关联的文件 s—19 fdopen()
- 替换已打开流 s—20 freopen()

八 设备检测

- 检测 I/O 设备信息 i—21 ioctl()
- 检测打开文件是否是设备文件 i—22 isatty()
- 附注:设备驱动程序

九 流缓冲区

- 将已打开流与用户指定缓冲区相联,缺省方式 s—23 setbuf()
- 将已打开流与用户指定缓冲区相联,非缺省方式 s—24 setvbuf()

十 文件句柄

- 复制一个新文件句柄,新句柄原来未和任何文件相联, i—25 dup()
新老句柄与同一文件相联
- 复制一个新文件句柄,与此同时关闭新句柄原来相联 i—26 dup2()
的文件,新老句柄与同一文件相联
- 通过流取文件句柄 s—27 fileno()

十一 关闭文件

- 关闭文件,只适用于 MS-DOS i—28 --close()
- 关闭文件 i—29 close()
- 关闭流 i—30 fclose()
- 关闭所有已打开流 s—31 fcloseall()

- . 刷新打开流 s—32 fflush()
- . 刷新所有打开流 s—33 fflush()

十二 删除文件

- . 删除一个文件 i,s—34 unlink()
s—35 remove()

十三 读操作

- . 从控制台(键盘)读一个字符,不回显(在屏幕上不显示) c—36 getch()
- . 从控制台(键盘)读一个字符,回显 c—37 getche()
- . 从预定义流 stdin 中读一个字符 s—38 getchar()
s—39 fgetchar()
- . 读当前流指针所指字符 s—40 getc()
s—41 —getc()
s—42 fgetc()
- . 从流中读一个整数 s—43 getw()
- . 从控制台读一个字符串 c—44 cgets()
- . 从预定义流 stdin 中读字符串 s—45 gets()
- . 从流中读字符串 s—46 fgets()
- . 利用格式串读字符串(参见三十一章) s ... scanf()
- . 读若干字节到缓冲区,只适用于 MS-DOS i—47 —read()
- . 读若干字节到缓冲区 i—48 read()
- . 从流中读 n 项数据 s—49 fread()

十四 写操作

- . 字符输出到控制台 c—50 putchar()
- . 向预定义流 stdout 中写一个字符 s—51 putchar()
s—52 fputc()
- . 写一个字符到流中 s—53 putc()
s—54 —putc()
s—55 fputc()
- . 写一个字(整数)到流中 s—56 putw()
- . 写字符串到控制台 c—57 cputs()
- . 把一个字符串送到流中,串尾加换行符 s—58 puts()
- . 把一个字符串送到流中,串尾不加换行符 s—59 fputs()
- . 利用格式串输出字符串(参见三十一章) s ... printf()
- . 写若干字节到文件中,只适用于 MS-DOS i—60 —write()
- . 写若干字节到文件中 i—61 write()
- . 将若干项数据写到流中 s—62 fwrite()
- . 回退一字符到流缓冲区 c—63 ungetch()
s—64 ungetc()

十五 文件读写定位

- | | |
|-------------------|----------------|
| · 检测到达文件尾 | i—65 eof() |
| · 检测流式文件结束标志 | s—66 feof() |
| · 报告文件指针当前位置 | i—67 tell() |
| · 报告流式文件当前指针位置 | s—68 ftell() |
| · 将文件指针移到指定位置 | i—69 lseek() |
| · 重定位流指针 | s—70 fseek() |
| · 定位流指针在文件开头 (重绕) | s—71 rewind() |
| · 保存流当前读写指针位置 | s—72 fgetpos() |
| · 重恢复流原指针位置 | s—73 fsetpos() |

十六 按键检测 (参见《键盘与鼠标》一章)

- | | |
|--------------|-------------|
| · 检查当前按下键 | i kbhit() |
| · BIOS 键操作检测 | b bioskey() |

十七 错误检测和处理

(perror()、—strerror() 和 strerror() (参见《DOS 错误处理函数》一章) signal() 和 raise() (参见《对 ANSI 定义的信号对应动作的重定义》一章)

- | | |
|--------------------|-----------------|
| · 检查流出错标志 | s—74 ferror() |
| · 清流状态标志和出错标志 | s—75 clearerr() |
| · 印出用户定义的系统出错信息字符串 | s perror() |
| · 返回指向用户定义的错误信息的指针 | s —strerror() |
| · 返回和错误号对应的错误信息 | s strerror() |
| · 指定信号动作 | g signal() |
| · 给执行程序发信号并执行相应动作 | g raise() |

十八 执行子程序 (参见《过程控制函数》一章)

- | | |
|-----------------|---------------|
| · 执行子程序后不再执行父程序 | p exec...() |
| · 执行子程序后继续执行父程序 | p` spawn...() |

十九 程序长驻内存 (参见《中断函数》一章)

- | | |
|-------------|----------|
| · 将当前程序长驻内存 | d keep() |
|-------------|----------|

二十 口令或共享锁

- | | |
|------------|----------------|
| · 设置文件共享锁 | i—76 lock() |
| · 解除共享锁 | i—77 unlock() |
| · 输入一个通行口令 | c—78 getpass() |

二十二 文件状态

- | | |
|------------------------|--------------|
| · 保存文件或目录信息到一个结构中 | t—79 stat() |
| · 保存与文件句柄相联的文件信息到一个结构中 | t—80 lstat() |

二十一 其它

· 重新设置文件读/写权限屏蔽

i-81 umask()

说明:

函数 clreol()、clrscr()、delline()、gettext()、gettextinfo()、gotoxy()、highvideo()、insline()、lowvideo()、movetext()、normvideo()、puttext()、textattr()、textbackground()、textcolor()、textmode()、wherex()、wherey()、window() 及变量 directvideo (参见《视频函数》一章)。

—1 int —Cdecl rename(const char *oldname, const char *newname);

函数把老文件名 oldname 换为新文件名 newname。要注意的是,如果文件名中包含 驱动器名,则两者的驱动器名应完全一样;另外,文件名中不允许出现 DOS 通配符 * 或?。

由于文件名中的路径名可以不同,因此它可以将一个目录中的文件按同名文件移到另一个目录中。注意:在操作系统下使用 DOS 的 rename(或 ren) 命令是不能实现文件移动的。

例如 C>REN SS.C C:\TC\SS.C

将出现

Invalid parameter

的错误提示。虽然可以先将某目录下的文件拷贝到另一目录下,接着将原目录下的文件删除,从而实现文件移动。但这样做将产生所谓的“磁盘文件碎片(fragmentation)”。这是因为文件的物理存储位置已改变,原来文件存储处成了“碎片”;二是由于 DOS 拷贝时并不检查在另一个子目录里是否已有同名文件存在,因此,稍有不慎,用户的重要文件可能被该操作破坏。而使用 rename() 则可避免这两种不利情况。

对 DOS 而言,由于子目录也被当作一个文件名对待,根据文件名唯一的原则,可以用本函数直接将子目录换名,而这要靠 DOS 的 ren 命令是做不到的。

对卷标能显示,但不能靠它换名。卷标在根目录内,一个盘只有一个卷标。可用绝对读函数 absread() 读出,然后用 abswrite() 写回,从而达到修改目的。

本函数相当于 DOS 功能调用 56H,入口参数是, AH=56H, DX=oldname, DI=newname。

如果调用成功返回 0,否则返回 -1,并置 errno=2(ENOENT)、或 5(EACCES)、或 17(ENOTSAM,不是同一设备)。

它只适用于 MS-DOS。

```
C>TYPE STD1.C
#include "stdio.h"
main()
{
    int mov;
    mov=rename("a:\\tc\\ss.c","a:\\tc\\zmh\\ss.c");
    if(mov)printf("文件移动成功! \n");
    else printf("文件未移动! \n");
}
```

```
C>TYPE STD2.C
#include "stdio.h"
```

```

main()
{
int k;
k=rename("*.i","*.t");
printf("%d\n",k); /* 结果输出 -1 */
k=rename("c:\\w.i","a:\\w.i");
printf("%d\n",k); /* 结果输出 -1 */
}

```

—2 char * —Cdecl parsfnm(const char * cmdline, struct fcb * fcb, int opt);

分析 cmdline 所指的字符串,一般为 DOS 命令行 (command line),以找到一个文件 名。该文件名包括驱动器名、文件基本名和扩展名,但不包括目录路径名。找到的文件名的 内容放入 fcb(参见《磁盘文件的结构》一章中函数 randbrd()) 中。opt 参数是 DOS 分 析系统调用中 AL 说明的值:

位0: 其值为 1 时,对 cmdline 中引导分隔符

.,:;=+ \ < > ! / " [] TAB(制表符)与空格符

忽略不管,否则当其值为 0 时就不将这些分隔符从文件名中去掉,一遇到文件 分隔符,则所有分析停止(即不再往下分析了,只将前面的分析输出),注意,文件名的结束符除这些分隔符外还可以是任何控制符(如响铃、换行、回车等等),当遇到文件名结束符时,文件名分析便停止。

位1: 当值为 1 时,只有当 cmdline 指定了驱动器,才在结果 fcb 中设立驱动器字节;如果没有包含驱动器号,则使用在 fcb 中的驱动器号。否则该字节值或 0(当没有指定驱动器号时),或者是无意义的。

位2: 当值为 1 时,只有当 cmdline 含有文件名时,才修改 fcb 中的文件名,否则 使用 fcb 中的文件名。当值为 0 时,如没有指定文件名,则 fcb 中文件名被 置成 8 个空格符。

位3: 当值为 1 时,只有当 cmdline 含有文件扩展名时,才修改 fcb 中的文件扩展 名,否则使用 fcb 中原有扩展名。当值为 0 时,如没有指定扩展名,则以空格 符填充。

位4~位7:未用。一般为 0。

如果找到了文件名,就建立相应的未打开的 fcb;如果没有提供驱动器标识符,就采 用缺省驱动器;如果没有提供扩展名,就全采用空格符;如果在文件基本名中出现通配符 *,那么它将文件基本名的所有字符全 置成?。对文件扩展名也有类似的结论。

如果分析一个文件名成功,函数返回一个指向紧跟在文件名后的字节指针;如果在分析 过程中出错,返回 0(即 NULL)。

它相当于 DOS 功能调用 29H,入口参数是, AH=0x29, SI=cmdline, DI=fcb, AL=opt。返回时 AL =00H 表示分析成功,未遇通配符

01H 表示分析成功,但遇到了通配符

FFH 表示碰到无效驱动器说明符

DS:SI 指向文件名后的第一个未分析的字符, ES:DI 指向格式化 fcb 的第一个字节。如果提供无效的文件 名, ES:DI+1 将是空白符。

它只适用于 MS-DOS。

C>TYPE STD3.C

#include "dos.h"

main(int argc, char * argv[]) /* 利用设置不同命令行参数可观察各种结果 */

{

int opt=0xf;

char * ptr,—al;

struct fcb f;


```

if(argc<2)exit(1);
ptr=parstrnm(argv[1],&f,opt);
--al=-AL;
printf("--AL=%d\n",--al);
printf("ptr=%p\n",ptr);
printf("fcb--drive=%d\n",f.fcb--drive);
printf("fcb--name[8]=%s\n",f.fcb--name);
printf("fcb--ext[3]=%s\n",f.fcb--ext);
printf("fcb--curblk=%u\n",f.fcb--curblk);
printf("fcb--reclsize=%u\n",f.fcb--reclsize);
printf("fcb--filsize=%ld\n",f.fcb--filsize);
printf("fcb--date=%u\n",f.fcb--date);
printf("fcb--resv[10]=%s\n",f.fcb--resv);
printf("fcb--currec=%c\n",f.fcb--currec);
printf("fcb--random=%ld\n",f.fcb--random);
}

```

/* 键入 C> 本程序名 C:L124.C 后输出:

```

--AL=-5
ptr=FFFB
fcb--drive=3
fcb--name[8]=L124    C
fcb--ext[3]=C
fcb--curblk=0
fcb--reclsize=0
fcb--filsize=-3538897
fcb--date=1303
fcb--resv[10]=x      x 为不可打印字符
fcb--currec=x
fcb--random=1543837184

```

这里 f 是一个【未打开的 FCB】,因此仅包括一个驱动器说明符和可包含通配符 (* 或 ?)的文件名。对打开的 FCB,则全部字段被填满 */

```
--3 int --Cdecl --chmod (const char *path,int func,... /* int attr */);
```

省略号 (...) 表示必要时可以增加一个变量 attr,它是重新设置 path 的文件属性变量。重置文件属性实际是对该文件在磁盘目录区的目录中有关文件属性字节的修改。文件属性字节各位的意义在 dos.h 中有说明:

```

#define FA_RDONLY    0x01    /* 位0,只读 */
#define FA_HIDDEN    0x02    /* 位1,隐含 */
#define FA_SYSTEM    0x04    /* 位2,系统 */
#define FA_LABEL     0x08    /* 位3,卷标 */
#define FA_DIRC     0x10    /* 位4,目录 */
#define FA_ARCH     0x20    /* 位5,档案 */
                        /* 位6、位7 未定义 */

```

显然,文件属性字节的各位都可能被置值 1,例如,一个系统文件可能是系统、隐含且只读时就是这样。另外,由于位 6 和位 7 未用来定义,所以在某些情况下也可作它用。

当 func=0 时,函数返回指定文件 path 的属性,这时函数可以不包含变量 attr。将函数返回值和 dos.h 中规定的文件属性符号常量“与”的结果便可知道文件属性。

当 func=1 时,函数应包括变量 attr,函数调用后,文件 path 的属性变成 attr。注意:它不能改变卷标或目录属性!所以,当 func=1 时,path 一定要是真正的文件名,而不是目录名或卷标名。如 path 是目录名,则用诸如:

```
—chmod(path,1,FA—ARCH);
```

等语句会产生不可预料的结果(改目录名可用 rename())。换言之,当 func=1 时 attr 只能用

```
FA—RDONLY  
FA—HIDDEN  
FA—SYSTEM  
FA—ARCH
```

这些符号常量。

如果调用成功,函数返回 path 的当前属性字,否则返回 -1,并置 errno=2(ENOENT)或5(EACCES)。

本函数相当于 DOS 功能调用 43H。func 相当其子功能号。入口参数是,AH=0x43;AL=func;DS:DX=path;CX=attr;返回时 CX=属性(对 func=0),出错时 CF=1,AX 返回出错码。

它只适用于 MS-DOS 2.0 或以上版本。

```
C>TYPE STD4.C  
#include "io.h"  
#include "errno.h"  
#include "dos.h"  
#define PCHMOD(X) printf("%s 的属性是:",X);\n\n    if(a != -1)\n    {\n    if( (a & FA—RDONLY) != 0 )printf("只读\\n");\n    else if( (a & FA—HIDDEN) != 0 )printf("隐含\\n");\n    else if( (a & FA—SYSTEM) != 0 )printf("系统\\n");\n    else if( (a & FA—LABEL) != 0 )printf("卷标\\n");\n    else if( (a & FA—DIREC) != 0 )printf("目录\\n");\n    else if( (a & FA—ARCH) != 0 )printf("档案\\n");\n    else printf("未知\\n");\n    }\n\nmain()\n{\n    int a;\n    char s[]="E.c";\n    char *p="c:\\tc";\n    a=—chmod("c:\\tc",0);\n    PCHMOD(p);\n    a=—chmod(s,0);\n    PCHMOD(s);\n    a=—chmod(s,1,FA—RDONLY);
```

```

PCHMOD(s);
a=—chmod(s,1,FA—ARCH);
PCHMOD(s);
}
/* 程序输出,c:\tc的属性是:目录
   E.c的属性是:档案
   E.c的属性是:只读
   E.c的属性是:档案 */

```

—4 int —Cdecl chmod (const char * path,int amode);

根据 amode 的值设置文件 path 的存取（访问）权限，与此同时它将改变 path 的文件属性。在 sys\stat.h 中定义 amode 可取值：

amode 值	stat.h 中符号常量值	存取权限
S—IREAD	0x0100	path 允许读
S—IWRITE	0x0080	path 允许写
S—IWRITE S—IREAD	0x0180	path 允许读和写

在 DOS 中，写允许隐含了读允许！当调用成功返回 0，否则返回 -1，并置 errno=2 (ENOENT) 或 5(EACCES)。path 不能是目录名。它适用于 UNIX 系统。

```

C>TYPE STD5.C
#include "io.h"
#include "sys\stat.h"
#include "dos.h"
int chmod(const char * path,int amode) /* 等效函数 */
{int attr; /* 函数实际只修改文件属性字节的位 0 的值 */
attr=—chmod(path,0); /* 取原文件属性 */
if(attr==—1)return (—1);
attr &= ~FA—RDONLY; /* 将 attr 位 0 置为 0 */
if( (amode & S—IWRITE) == 0) /* 如 amode 为不含有 S—IWRITE */
attr |= FA—RDONLY; /* attr 将被置成只读属性 */
attr = —chmod(path,1,attr); /* 重置文件属性 */
if(attr == —1) return (—1);
return 0;
}
main()
{ char * ptr="q.c";
if ( ! chmod(ptr,S—IREAD | S—IWRITE))printf("文件q.c 设置成可读写\n");
else printf("设置未成功\n");
}

```

—5 int —Cdecl setmode (int handle,int amode);

函数用来改变已打开文件的存取方式，handle 是已打开文件的文件句柄；在 MS-DOS 下，amode 是通过全局变量 _fmode 来设置的新的存取标志，只能是 O—TEXT 或 O—BINARY 中的一个。

如调用成功返回新存取标志，否则返回 -1 表示出错，并置 errno=19(EINVAL)。它适用于 UNIX 系统。

C>TYPE STD6.C

```
#include "io.h"
#include "fcntl.h"
main()
/* 假定 qq.c 内容为: 0x61 0xd 0xa 0x62 0x1a */
/* 假定 q1.c 文件也存在 */
{
    int handle,h,get;
    if( (handle=open("qq.c",O-TEXT))<0)exit(1); /* 如用 -open 则效果不同 */
    printf("handle=%d\n",handle);
    h=dup(handle); printf("-fmode=%#x,%d\n",-fmode,-fmode);
    get=setmode(h,O-BINARY);
    printf("h=%d, get=%#x,%d\n",h,get,get);
    -fmode=O-BINARY;
    get=setmode(h,-fmode);
    printf("get=%#x,%x\n",get,get);
    h=open("q1.c",-fmode); /* 可看看文件句柄增加情况 */
    printf("h=%d,-fmode=%#x,%d\n",h,-fmode,-fmode);
}
/* 程序输出: handle=5
-fmode=0x4000,16384
h=6, get=0x4000, 16384
get=0x8000,8000
h=7,-fmode=0x8000,-32768 */
-6 long -Cdecl filelength(int handle);
```

得到文件长度的字节数。若调用成功,返回一个与文件句柄 handle 相联的文件长度(注意,是长整型值)。出错时返回 -1L,并置 errno=6(EBADF)。

C>TYPE STD7.C

```
#include "io.h"
main()
/* 假定 qq.c 内容为 ABCDEF */
{ int handle;
    if( (handle=open("qq.c",1,1))<0)exit(1);
    printf("%ld\n",filelength(handle)); /* 6 */
    printf("%ld\n",filelength(15)); /* -1 */
    printf("%#lx\n",filelength(15)); /* 0xFFFFFFFF */
}
```

-7 int -Cdecl chsize (int handle,long size);

本函数改变同文件句柄 handle 相联的文件的大小(单位为字节数)。它可以根据 size 的值同文件原来的大小进行比较后决定截断(当 size 小于原文件长度)或扩展(当 size 大于原文件长度)文件。如果是扩展文件则在文件尾部添加空字符('\0');如果是截断文件,则新文件结束标志之后的所有数据将被丢弃。对文件打开模式必须允许写。调用成功时函数返回 0,否则返回 -1,并置 errno=5(EACCES)或 6(EBADF)。

它只适用于 MS-DOS。

C>TYPE STD8.C

```
#include "io.h"
#include "stdio.h"
```

```

#include "fcntl.h"
/* 注意：此处必须以二进制、可写方式打开文件才能得正确结果 */
#define PLEN handle=open("q.c",O_RDWR|O_BINARY); /* 打开文本文件 */
if(handle== -1) {printf("因文件打开失败而退出\n");exit(1);}
fp=fopen(handle,"r"); /* 定义流 */
len=filelength(handle); /* 取文件长度 */
printf("文件长度=%ld\n",len);
for(k=0;k<len;k++) /* 输出文件内容 */
{ c=fgetc(fp);
if(c != '\0') putchar(c);
else putchar('=');
}printf("\n")

```

```

main()
{
int handle; /* 文件句柄 */
long len; /* 文件长度 */
long k;
FILE *fp;
char c;
printf("原");
PLEN;
printf("增加4个字节后");
chsize(handle,len+4); /* 扩展文件 */
close(handle); /* 必须关闭文件 */
PLEN;
printf("减少8个字节后");
chsize(handle,len-8); /* 截断文件 */
close(handle);
PLEN;
}

```

/* 程序输出 说 明

原文件长度=55 此行末尾有回车换行符未输出

```
#include "stdlib.h" 21
main(int argc,char *argv[]){}32
增加4个字节后文件长度=59
#include "stdlib.h" 21
main(int argc,char *argv[]){}32==== 等号为扩展字符'\0'
减少8个字节后文件长度=51
#include "stdlib.h" 21
main(int argc,char *argv[]){}
/*
```

—8 int —Cdecl access (const char * path,int amode);

函数根据 amode 的值和 path 本身的文件属性查看指定的文件或路径（路径中不能有通配符 * 或 ?）path 存在与可读写性。

amode 值	功 能
0	检查 path 是否存在。只要文件存在就允许读。
1	检查能否执行 (未用)
2	检查 path 是否可进行写访问。允许写当然也允许读。当磁盘文件贴有写保护时就不允许写
4	检查是否允许读
6	检查是否允许读写

当允许做指定的访问时函数返回 0, 否则返回 -1。出错时置 `errno=2(ENOENT, 文件或路径未找到)` 或 `5(EACCES, 拒绝访问)`。

如果 path 指向一子目录, amode 应选 0, 即只确定其是否存在。

函数不是 ANSI 标准定义的, 它适用于 UNIX 系统。

C>TYPE STD9.C

```
#include "io.h"
#include "errno.h"
#define FA_RDONLY 0X01 /* 文件属性为:只读。在 dos.h 中有定义,下同 */
#define FA_HIDDEN 0X02 /* 文件属性为:隐含 */
#define FA_DIRC 0X10 /* 文件属性为:目录 */
int access(const char *fname,int amode) /* 等效函数 */
{
int attrib=chmod(fname,0); /* 返回文件属性,当 fname 不存在时返回0xFFFF */
if(attrib==-1)return (-1);
if( (amode & 0x0002) == 0 || (attrib & FA_RDONLY) == 0)return 0;
errno=EACCES; /* 对隐含且只读文件或不存在的 fname 返回-1 */
return -1;
}
main()
{
int a;
char s[]="w.c";
printf("目录c:\tc");
if((a=access("c:\tc",0)) == 0)printf("存在,");
else printf("不存在,");
printf("a=%d\n",a);
a=0;
if(a==access(s,4))printf("文件w.c 存在且可读\n");
else printf("指定文件不存在\n");
}
/* 假定指定目录和文件均存在,则程序输出:
    目录c:\tc 存在,a=0
    文件w.c 存在且可读 */
```

—9 int —Cdecl —creat (const char *path,int attribute);

根据文件属性字节 attribute 创建文件 path, 则 path 的长度为 0。若原磁盘上已有 path, 则该文件若没有写保护就会被抹去, 即该文件的原有内容不再存在。

创建后的文件 path 的属性只能有下述四种:

attribute 取值	path 文件属性
FA_RDONLY	FA_ARCH FA_RDONLY

FA—HIDDEN	FA—ARCH FA—HIDDEN
FA—SYSTEM	FA—ARCH FA—SYSTEM
FA—ARCH	FA—ARCH

这很容易调用 `—chmod(path,0)`；验证。对属性为 FA—ARCH 的文件，文件指针被置为文件开头，打开后可读可写。对已经存在的文件使用本函数时，如已有文件的属性为只读，则调用本函数会失败，原文件不变。要想改变它，应考虑先用 `—chmod(path,1,FA—ARCH)` 将文件改成可读写，然后再创建；对其它已有文件用本函数创建时，原文件的长度将变为 0，属性不变。

用本函数创建的文件由全程变量 `—fmode` 决定文件按 O—TEXT 或 O—BINARY 方式（在 `fcntl.h` 中有定义）传送。可以在程序中对 `—fmode` 先行赋值，以控制传送方式。注意，对已有文件 `path` 创建时，`—creat()` 总是以 O—BINARY 方式打开已有文件，然后对它修正的。如果创建文件成功，文件移动指示器（或文件指针）被置于文件开头。

如调用成功返回文件句柄（文件句柄与流或文件移动指示符均无关），否则返回 -1，并置 `errno=2(ENOENT)`、`4(EMFIFE)` 或 `5(EACCES)`。

它实际是 DOS 功能调用 3CH，入口参数是，AH=3CH,CX=attribute,DS:DX=path。它只适用于 MS—DOS。

```
C>TYPE STD10.C
#include "io.h"
#include "stdio.h"
#include "dos.h"
main()
{ int ct;
  FILE *fp;
  ct=—creat("qq.c",FA—ARCH);
  fp=fopen(ct,"w");
  if(fp != NULL)
  {
    fputc('B',fp); /* 向 qq.c 输入一个字符 B */
    fclose(fp); /* 此句不可少 */
    close(ct); /* 生成长度为 1 的文件 qq.c */
  }
}

C>TYPE STD11.C
#include "io.h"
#include "dos.h"
#include "sys\stat.h"
#include "fcntl.h"
main()
{
  int ct;
  printf("前,—fmode=%#x\n",—fmode); /* 输出全程量 —fmode 的缺省值 */
  ct=—creat("qq",FA—DIREC);
  printf("后,—fmode=%#x\n",—fmode);
  if(ct==—1)
```

```

{
printf("ct=%d,定义目录qq失败\n",ct);
ct=-creat("qq.c",FA-RDONLY);
if(ct != 0)
{
printf("定义文件qq.c成功\n");
ct=unlink("qq.c");
if(ct == 0){printf("删除文件qq.c成功! \n");exit(0);}
else {printf("删除文件qq.c未成功\n");
ct=-creat("qq.c",FA-ARCH);
if(ct != 0)
{printf("改写qq.c文件属性成功! \n");
ct=unlink("qq.c");
if(ct == 0)printf("再次试删除qq.c成功! \n");
else { printf("又失败\n");
chmod("qq.c",S-IWRITE);
ct=unlink("qq.c");
if(ct == 0)printf("删除成功\n");
else printf("删除未成功\n");
}
}
}
}
}
else printf("定义目录qq成功\n");
} /* 在 DOS 提示符下键入 C>DEL QQ.C 后显示 Access denied 表示删除失败 */
/* 程序输出:前:-fmode=0x4000 缺省方式为 O-TEXT 而不是 O-BINARY
后:-fmode=0x4000
ct=-1,定义目录qq失败
定义文件qq.c成功
删除文件qq.c未成功
改写qq.c文件属性成功!
又失败
删除成功
*/

```

—10 int —Cdecl creat (const char * path,int amode);

它的功能基本上同 —creat(),它只检查存取模式字 amode 的一位,该位为 UNIX 的用户写允许位。如果该位为 1,则文件可写;为 0 时文件为只读文件。所有其它 MS-DOS 的属性位都置为 0。amode 只在新建文件时对用户写允许位起作用,对已有文件被忽略。amode 可取在 sys\stat.h 中定义的符号常量:

```

#define S-IREAD    0x0100    /* 只读 */
#define S-IWRITE   0x0080    /* 可写 */

```

也可用 S-IREAD | S-IWRITE,不过对DOS可写也就隐含了可读,因此它不是必须的。

如果文件已存在并且写属性被置位,则函数截断文件长度为0,但文件属性不变;若文件属

性为只读,则函数调用失败,而文件不变。

它适用于 NUIX 系统。

—11 int —Cdecl creatnew(const char *path,int mode);

按属性 mode 创建一个新文件 path。它只在一处与 —creat() 不同,如果 path 早已存在,则函数调用便失败,而原文件 path 不变(长度也不会截为 0)。因此,有时也可用它来检查某文件是否早已存在。

调用成功,返回文件句柄,否则返回 -1。

它实际是 DOS 功能调用 5BH,入口参数是, AH=5BH, CX=mode, DS:DX=path。

它只适用于 MS-DOS 3.0 及以上版本。

C>TYPE STD12.C

```
#include "io.h"
#include "dos.h"
#include "sys\stat.h"
#include "fcntl.h"
main()
{ int ct;
  char *path="qq.c";
  char c;
  ct=creatnew(path,FA-ARCH);
  if(ct== -1)
  {printf("文件%s 早已存在,要重建? (Y/N):\n",path);
   c=getch();
   if(c=='y'|c=='Y')
   {
    _chmod(path,1,FA-ARCH);
    ct=creat("qq.c",S-IWRITE);
    if(ct != 0)
    { printf("文件%s 重建成功,文件句柄是%d,",path,ct);
      ct=_chmod("qq.c",0);
      printf("属性是%#x\n",ct);
      return;
    }
   }
   else{ printf("文件%s 未重建\n",path);return;}
  }
  printf("文件%s 建立成功\n",path);
}
```

—12 int —Cdecl creattemp(char *path,int amode);

它用来建立一个具有唯一的文件名字而后在程序运行结束时又被删除的暂时文件,文件名放在 path 所指的串中。path 应有足够的长度(至少容纳 13 个字符),暂时文件的属性由 amode 设置(同 —creat() 中的 attribute)。

调用成功,返回文件句柄,否则返回 -1。

它实际 DOS 功能调用 5AH,入口参数是, AH=5AH, CX=amode, DS:DX=path。

它只适用于 MS-DOS 3.0 及以上版本。

```

C>TYPE STD13.C
#include "io.h"
#include "dos.h"
main()
{ int ct;
  char drive[3],dir[66],name[9],ext[5]; /* 它们必须定义成这样 */
                                     /* 不能定义成指针形式:char *drive,... */
  char path[13];                      /* 至少定义为 13,一般为 80 */
  ct=createmp(path,FA-ARCH);
  if(ct== -1) {printf("不能创建暂时文件\n");exit(1);}
  else printf("%s\n",path);
  fnsplit(path,&drive,&dir,&name,&ext); /* 拆分文件名 */
  printf("drive=%s, dir=%s, name=%s, ext=%s\n",drive,dir,name,ext);
}
/* 程序输出:\AOAPCDFO
   drive=, dir=\\, name=AOAPCDFO, ext= */
—13 char *—Cdecl tmpnam(char *s);

```

它自动在磁盘上建立一个【暂时（或临时）文件名】，供执行程序在执行中间存放中间数据。当程序运行结束时如果对暂时文件未操作，则该文件自动从磁盘上消失；否则便留在磁盘上。暂时文件只建立在当前目录下，且该文件名不同磁盘上已有文件同名，即是唯一的。假定函数返回串 t，则 t 便是一个刚建立的暂时文件名。t 一般的形式为 TMPnn. \$\$\$，其中 nn 表示 1 到 2 位数字。注意：当 s 为 NULL 时应该用其返回的字符串 t 作暂时文件名，否则也可用 s 作暂时文件名，因为 t 和 s 指向同一内存区。当 t 或 s 重新被指向一暂时文件时，它们跟原指的暂时文件便无关。

特别要注意的是，如 s 或 t 作指针定义，它们应有足够的确定的存储缓冲区，其大小由

```
#define L—tmpnam 13
```

定义。例如，要用指针 s 作临时文件名，则最好定义

```
char *s="123456789012";
```

否则有可能出错。最大可生成的暂时文件数虽然被定义为

```
#define TMP—MAX 0xFFFF
```

但实际上一是很少在一个程序中用许多暂时文件，二是同时被打开的文件数也受到制约。参见定义

```
#define OPEN—MAX 20
```

```
#define SYS—OPEN 20
```

注意：它跟 dir.h 中的 mktemp() 的不同。

```

C>TYPE STD14.C
#include "stdio.h"
#include "string.h"
#include "stdlib.h"
main()

```

```

{
FILE *fp;
char *s1=0,t1[L—tmpnam];
char *s2,t2[L—tmpnam];
char *s3="c",t3[L—tmpnam];
char *s4="QQ. C",t4[L—tmpnam];
char *s5="QQQCqqcc",t5[L—tmpnam];
char *s6="c;\tc\QQQQQ",t6[L—tmpnam]="ABCDEFGHUKLM";
strcpy(t1,tmpnam(s1));
strcpy(t2,tmpnam(s2));
strcpy(t3,tmpnam(s3));
fp=fopen(t3,"w+");
putc('H',fp);
strcpy(t4,tmpnam(s4));
strcpy(t5,tmpnam(s5));
strcpy(t6,tmpnam(s6));
}

```

/* 假定磁盘上有文件 TMP1. \$\$\$,而没有 TMP2. \$\$\$ ~TMP7. \$\$\$,在输入相应调试表达式后,除在磁盘上有暂时文件 TMP1. \$\$\$ 外,尚有

```

s1: NULL
t1: TMP2. $$$
s2: "TMP3. $$$"
t2: "TMP3. $$$"
s3: "TMTMP5. TMP6. $$$"
t3: "TMP4. $$$"
s4: "TMP5. TMP6. $$$"
t4: "TMP5. $$$"
s5: "TMP6. $$$"
t5: "TMP6. $$$"

```

特别有

```

t6: "TMP7. $$$"
s6: "TMP7. $$$"
s6,s: "TMP7. $$$"
s6,p: DS:01B2

```

注意:这时你按 F4 键,发现这些调试表达式显示的内容形式不同了,跟踪的亮条消失。例如有

```

t6: "Fr \x21 \x1 \x1"
s6: DS:8931
s6,s: unknown
s6,p: DS:8931

```

如你又按 F4 键后又返回原状态。当然,光标必须在调试行上,否则会出现错误提示

No code generated for this line. Press ESC */

—14 FILE *—Cdecl tmpfile(void);

该函数先产生一个暂时文件,用户可以不知道它究竟有什么文件名。接着它又调用 fopen (暂时文件名,"w+b"); 打开与此暂时文件相联的二进制读/写流。如调用成功,它返回指

向该流的指针,否则返回 NULL。用户对该流的操作就是对暂时文件的操作。

如果在包含它的程序运行时将流关闭,或者用了 exit() 等退出运行,则磁盘上的暂时文件将被自动消去;而用 abort() 中途退出时磁盘上暂时文件将保留。

```
C>TYPE STD15.C
#include "stdio.h"
#include "stdlib.h"
main()
{
FILE *fp, *fp1;
char s[4], *str="uvwxyz",
char c,ch[10];
fp=tmpfile();
setvbuf(fp,s,-IOFBF,2); /* 设置与暂时文件相联的流缓冲区 */
fputc('K',fp);          /* 向流写入字符 */
fputs(str,fp);
fputc('H',fp);
rewind(fp);             /* 此句不能少 */
c=fgetc(fp);            /* 从流取出字符 */
fgets(ch,6,fp);
fp1=fopen("QQ.C","w+b");
fputc(c,fp1);           /* 向流写入字符 */
fputs(ch,fp1);
/* 如用语句 fclose(fp); 最后磁盘上的暂时文件会自动消失 */
fclose(fp1);
/* 如
加上语句 abort(); 可以看到磁盘上产生的暂时文件 */
exit(0); /* 不管有没有这一句,最终磁盘上不会有暂时文件出现 */
}
/* C>TYPE QQ.C
Kuvwxyz */
```

—15 int —Cdecl —open(const char *path,int oflags);

它打开已存在文件 path() (即使 path 的文件属性为隐含或系统的,也可用它打开),对 DOS 2.X 版本, oflags 只限于 fcntl.h 中定义的三个符号常量:

```
#define O_RDONLY 1 /* 即 0x0001,只读 */
#define O_WRONLY 2 /* 0x0010,只写 */
#define O_RDWR 4 /* 0x0100,读写 */
```

在 DOS 3.X 下还可以用以下值 (定义在 fcntl.h 中,它们对 DOS 2.X 是不适用的): #define O_NOINHERIT 0x80 /* 文件为当前进程私用,不由于进程继承 */

```
#define O_DENYALL 0x10 /* 只允许当前程序读写,拒绝其它程序读写文件 */
#define O_DENYWRITE 0x20 /* 只允许当前程序写,拒绝其它程序写打开文件 */
#define O_DENYREAD 0x30 /* 只允许当前程序读,拒绝其它程序读打开文件 */
#define O_DENYNONE 0x40 /* 允许其它程序共享打开的文件 */
```

这些符号常量是对存取和共享模式而言的,O_NOINHERIT 是 oflags 的位 7,它说明【继承方式】;其它为位 6、位 5 和位 4 的值,它们说明【共享方式】,当这三位值为 0x00 时为兼容模式;

这可以参见 share.h 中定义的符号常量：

```
#define SH_COMPAT 0x0000 /* 兼容模式 */
#define SH_DENYRW 0x0010 /* 拒绝读写 */
#define SH_DENYWR 0x0020 /* 拒绝写 */
#define SH_DENYRD 0x0030 /* 拒绝读 */
#define SH_DENYNONE 0x0040 /* 允许读写接收 */
#define SH_DENYNO SH_DENYNONE /* 用一个宏名定义另一个 */
```

位3 未用，常为 0；位2、位1 和位0 的组合说明【存取方式】：

```
000    只读
001    只写
010    读写
```

文件被打开后，指明文件当前读写位置的文件读写指针指向文件的头部。从亲进程所继承的文件句柄同时也继承了共享和存取访问限制。

对文件共享而言，一个文件被打开后还有可能用此函数进行第二次打开，不过 oflags 只能取某一个 O-DENYxxxx 值，而且文件不应是锁上的。第二次如打开失败，有可能返回错误码，也可能产生中断 INT 24H。

同时能打开的最大文件数由系统配置文件 CONFIG.SYS 设置。

如打开成功，函数返回文件句柄，否则返回 -1，并置 errno=2(ENOENT)、或 4(EMFILE)、或 5(EACCES) 或 12(EINVAL)。

它使用了 DOS 功能调用 3DH，入口参数是 AH=3DH，AL=oflags，DS:DX=path，CL=0f0H（文件属性屏蔽，仅服务器调用。如果文件所具有的属性不是所指定的只读或档案，则拒绝打开）。

它只适用于 MS-DOS。

```
C>TYPE STD16.C
#include "io.h"
#include "fcntl.h"
main()
{ int handle,k; /* 假定 qq.c 内容为 ABCDEF */
  if( (handle=-open("qq.c",O-DENYREAD))<0)exit(1);
  printf("%d\n",handle);
}
/* 在未装网络的DOS 5.0 环境下运行，输出：5 */
```

```
—16 int —Cdecl open(const char *path,int access,... /* unsigned mode */);
```

它是基于 -open() 基础上打开 path 文件的函数。省略号表示【存取权限】mode，其可取值为 sys/stat.h 中的 S-IWRITE、S-IREAD 或 S-IWRITE | S-IREAD。只有当 access 取 O-CREAT 时才用它。access 由允许读写标志和存取标志进行“或”操作得到：

【允许读写标志】是 fcntl.h 中定义的符号常量 O-RDONLY（只读）、O-WRONLY（只写）和 O-RDWR（读写）；【存取标志】在 fcntl.h 中定义为

```
#define O_CREAT 0x0100 /* 如果 path 早已存在，则它无用，否则创建 path 且用 mode 设置 path 文
                           件属性位，如同 chmod() 函数那样 */
#define O_TRUNC 0x0200 /* 将已存在文件 path 的长度截为 0，但属性不变 */
```

```
#define O—EXCL 0x0400 /* 它只和 O—CREAT 一起使用,如 path 早已存在,返回 错误 并禁止打开
path */
```

这三个常量在文件打开后就可以在执行程序中作它用。

```
#define O—APPEND 0x0800 /* path 被打开后内容不变,只是文件一打开文件指针便指 向文件 path
的末尾,写入内容便接在文件尾 */
```

```
#define O—TEXT 0x4000 /* 显式表示 path 以文本方式打开。注意,此种方式在读 写文件时要进行回
车/换行符 (CR/LF) 的自动转换 */
```

```
#define O—BINARY 0x8000 /* 显式表示以二进制方式打开文件,不进行 CR/LF 转换 */
```

注意:如果没有显式地给出 O—TEXT 或 O—BINARY,则 path 按全局变量 —fmode 设置的值传送文件;如果用户没有对 —fmode 设置值,则它以缺省方式 (一般为 O—TEXT) 传送。

它适用于 UNIX 系统。返回等输出情况同 —open()。

说明:fcntl.h 中还有几个符号常量的含义是:

```
#define O—CHANGED 0x1000 /* 这两个常量主要用在寄存器传送输入输出 #define O—DE-
VICE 0x2000 (RTL\io) 中,用户一般不直接使用它们 */
```

```
#define —O—RUNFLAGS 0x0700 /* 执行标志,用户一般可不用 */
```

```
#define —O—EOF 0x0200 /* 判别文本文件遇到 1AH 码时的标志 */
```

C>TYPE STD17.C

```
#include "io.h"
```

```
#include "fcntl.h"
```

```
main()
```

```
{ int handle; /* 假定 qq.c 内容为 ABCDEF */
```

```
if( (handle=open("qq.c",O—RDWR|O—EXCL|O—CREAT))<0)
```

```
{ printf("文件qq.c 早已存在,但不能打开它作其它用处\n");
```

```
exit(1);
```

```
}
```

```
printf("文件句柄=%d\n",handle);
```

```
}
```

```
/* 程序输出:文件qq.c 早已存在,但不能打开它作其它用处 */
```

```
—17 #define sopen(path,access,shflag,mode) open(path,(access)
|(shflag),mode)
```

它定义了一个和早期版本及其它编译程序兼容的打开文件的宏。其中,shflag 是一指明文件共享方式的类型标志,参见 share.h 中的常数定义 SH—xxxxx。

```
—18 FILE *—Cdecl fopen(const char *path,const char *mode);
```

打开由 path 指定的文件。函数返回一个和 path 相联的流指针,流同时被打开。这里 mode 是决定文件打开的方式。它包括两层意思,一是指出文件是用来读还是用来写,二是打开的流数据是作为一个文本文件还是作为一个二进制数据来进行处理。它相当于 open 函数中的第二个参数 (access),即包含读 / 写标志和存取标志。mode 是一个字符串 (可称“打开方式串”),它最多由三个字符构成。构成串的字符一般在 rwatb+ 这六个字符中选择,而且其第一个字符必须是 rwa 中的一个字母。第二字符可为 tb+ 中的一个字符。第三个字符也可从 tb+ 中选一个,但它不能和第二个字符相同。必要时,第二、第三个字符也可以一个也不要。各字符的含义是 (注意:凡字母均为小写字母):

r 将文件打开用于只读

- w 如果 path 已存在,则先将它长度改为 0,否则建立新文件 path。随后将文件 打开,允许接着的操作向文件写数据
- a 如果 path 已存在,则在打开后允许随后的操作将数据写到它的末尾;否则,建立 新文件 path,并允许以后写入数据
- + 表示可读可写
- t 流数据作为文本文件处理
- b 流数据作为二进制数据处理

例如,rt,r+t,rt+,wb,a+b 等等。

若 mode 中没有给出 b 或 t,则由全局变量 `__fmode` 决定流数据的打开方式:
`__fmode` 为 `O-TEXT` 则以文本文件方式,为 `O-BINARY` 则以二进制方式。除非你已确信一个文件是完全的文本文件,或者说在其正文中至少不含有像 `1AH` 这样有可能作为文件结束处理的字符,才能以文本方式打开,否则最好以二进制方式打开。

注意:当文件按可读 / 写方式 (当打开字符串中含有加号时,象 `r+`、`rb+`、`w+`、`a+` 等) 打开时,流上既可以输入,也可以输出。但是,要想在输入后直接输出 (或在输出后直接输入),必须在调用语句之前先调用 `fseek(stream,0L,SEEK-SET)`; 或 `rewind(stream)`; 使流当前指针指向流缓冲区头部,否则可能出错。总而言之,要小心应用。

若函数调用成功,返回新打开的流,否则返回 `NULL`。

它适用于 UNIX 系统。

```
C>TYPE STD18.C
#include "stdio.h"
static FILE *pascal near getfp(void) /* 获取文件指针,局限于本模块 */
{
    register FILE *fp;
    fp=__streams; /* 初始指向预定义流 */
    while(fp->fd >= 0 && fp++ < __streams+OPEN-MAX)
        continue;
    if((fp->fd >= 0))return NULL;
    else return fp;
}
void __xfclose(void) /* 退出时调用,以关闭所有打开的流 */
{
    register FILE *fp; /* __streams 在 stdio.h 中定义为数组: */
    register int k; /* extern FILE __Cdecl __streams[]; */
    for(k=OPEN-MAX-5,fp=__streams+5; --k; fp++)
        if(fp->flags & __F-RDWR)fclose(fp); /* 前 5 个预处理流未关闭 */
}
#include "fcntl.h"
#include "sys/stat.h"
#include "io.h"
static unsigned pascal near CheckOpenMode(register const char *type,
    unsigned *oflagsP,unsigned *modeP) /* 检查文件打开模式 */
{
    extern void (*__exitfopen)(); /* 函数指针 */
    extern void __xfclose();
    unsigned oflags=0;
```

```

unsigned mode=0;
unsigned flags=0;
char c;
if((c=*type++)=='r') /* 如果打开方式串首字母用了r */
{
    oflags = O_RDONLY;
    flags = --F_READ; /* 可读 */
}
else if(c=='w') /* 打开方式串首字母为 w 时 */
{
    oflags=O_CREAT | O_WRONLY | O_TRUNC; /* 将存在文件长度截为 0,属性不变 */
    /* 文件不存在则建立它 */
    mode =S-IWRITE;
    flags =--F-WRIT; /* 可写 */
}
else if(c=='a') /* 打开方式串首字母为 a 时 */
{
    oflags=O_WRONLY | O_CREAT | O_APPEND; /* 如果文件不存在,则建立它 */
    mode =S-IWRITE;
    flags =--F-WRIT; /* 可写 */
}
else return 0; /* 打开方式串首字母为其它字符时为非法 */
/* 对打开方式串第 2、第 3 个字符程序不检查 */
c=*type++; /* 检查打开方式串第 2 个字符 */
if(c=='+' || (*type=='+' && (c=='t' || c=='b')))
{ /* 第 2 个字符为 +,或第 2 个字母为 t 或 b 而第 3 个字符为 + 时 */
    if(c=='+') c=*type; /* 如第 2 个字母为 + 时,c 取第 3 个字符 */
    oflags=(oflags & ~(O_WRONLY | O_RDONLY)) | O_RDWR;
    mode =S-IREAD | S-IWRITE;
    flags =--F-READ | --F-WRIT; /* 可读可写 */
}
if(c=='t') oflags |= O_TEXT; /* 文本方式,c 是 (第 2 个字符,或当第 2 个为 + 时则为第 3 个
                                字符) 小写字母 t 时 */
else if(c=='b') /* c 为 b 时 */
{
    oflags |=O-BINARY;
    flags |=--F-BIN; /* 二进制方式 */
}
else /* c 非 t 或 b 时 */
{
    if((oflags |=(--fmode & (O_TEXT | O-BINARY))) & O-BINARY)
        flags |=--F-BIN; /* 二进制方式 */
}
--exitfopen=--xfclose;
* oflagsP=oflags; /* open() 读写与存取标志 */
* modeP=mode; /* open() 读写允许与存取权限 */

```



```

return flags;          /* fp->flags          */
}
static FILE *pascal _openfp(FILE *fp,const char *path,const char *mode) {
    /* 具体的打开文件模块 */
    unsigned oflag,type;
    if(((fp->flags==CheckOpenMode(mode,&oflag,&type)) == 0) ||
        ((fp->fd<0) && (fp->fd=open(path,oflag,type))<0)) /* 调用 open() */
    {
        fp->fd=-1;          /* 三种导致文件打开失败的处理方法 */
        fp->flags=0;
        return NULL;
    }
    if(isatty(fp->fd))fp->flags |= _F_TERM;          /* 打开设备文件时 */
    if(setvbuf(fp,NULL,(fp->flags & _F_TERM)? _IOLBF:_IOFBF,BUFSIZ))
    {
        fclose(fp);          /* 系统自动分配流缓冲区失败时的处理 */
        return NULL;
    }
    else
    {
        fp->istemp=0;          /* 流缓冲区分配成功后的进一步处理 */
        return fp;
    }
}
FILE *fopen(const char *path,const char *mode) /* 等效函数 */
{
    register FILE *fp;
    if((fp=getfp()) == NULL)return NULL; /* 取文件句柄失败时 */
    else return _openfp(fp,path,mode);
}
main() /* 实际调用 fopen() 时源程序里不用书写上述等效函数 */
{
    /* 等效函数供调试时用来观察流的变化情况 */
    FILE *fp,*fptr;
    fp=fopen("qq.c","r");
    printf("%p,%x\n",fp,fp->fd);
    fptr=fopen("qw.c","w");
    printf("%p,%x\n",fptr,fptr->fd);
    fclose(fp);
    fp=fopen("qe.c","r");
    if(fp != NULL)printf("%p,%x\n",fp,fp->fd);
    else printf("第一次打开文件失败!\n");
    if((fptr=fopen("qr.c","r")) != NULL)printf("%p,%x\n",fptr,fptr->fd);
    else printf("第二次打开文件失败!\n");
    fclose(fptr);
    fp=fopen("QE.C","w");
    if(fp != NULL)printf("%p,%x\n",fp,fp->fd);
}

```

```

else printf("第三次打开文件QE.C 失败! \n");
if((--chmod("qe.c",0) & S-IWRITE) != S-IWRITE)printf("文件QE.C 为只读文件\n");
if((fptr=fopen("qr.c","w")) != NULL)
{
printf("%p,%x\n",fptr,fptr->fd); /* 向 qr.c 中写入“打开文件成功!” */
fprintf(fptr,"打开文件成功! \n");
}
else printf("第四次打开文件失败! \n");
}
/* 如果已有文件 QE.C 为只读文件,程序输出:032E,0x5
                                033E,0x6
                                032E,0x5
                                034E,0x7
                                第三次打开文件QE.C 失败!
                                文件QE.C 为只读文件
                                032E,0x5          */

```

C>TYPE STD19.C

```

#include "stdio.h"
#include "fcntl.h"
main() /* 本程序说明对新建的流不能在送字符后又取出的原因 */
{
FILE *fp;
char y;
char *path="QW.C";
--fmode=O-TEXT;
fp=fopen(path,"w"); /* 为读写不能用这种打开方式! 打开方式串中应含加号 */
fputc('H',fp); /* 送字母 H 到流中 */
rewind(fp); /* 缺少此句也会出错 */
y=fgetc(fp); /* 用调试表达式容易看出 fp->flags 有出错标志 -F-ERR */
printf("%c\n",y); /* 不能得到正确值 */
}

```

C>TYPE STD20.C

```

#include "stdio.h"
#include "fcntl.h"
#include "string.h"
main() /* 本程序说明对新建的可读/写流在输入后又直接输出的情况 */
{
FILE *fp;
static char y[100];
char *path="QW.C";
int n=8;
--fmode=O-TEXT;
fp=fopen(path,"rt+"); /* 只对这种可读可写文件(打开方式串中含 */
/* 加号)才行。对其它打开方式是不行的! */
fputs("ABCDEFGH",fp); /* 送字符串到流中 */
/* 或用 fwrite("ABCDEFGH",sizeof(char),strlen("ABCDEFGH"),fp); */

```

```

fseek(fp,0L,SEEK-SET); /* 指向文件头部 */
/* 或用 rewind(fp); 使文件指针指向文件头部 */
/* 用fseek(fp,0L,SEEK-END),或fsseek(fp,0L,SEEK-CUR)因到文尾故不读入数据 */
n=fread(y,sizeof(char),strlen("ABCDEFGH"),fp);
printf("n=%d\n",n);
if(n == strlen("ABCDEFGH"))printf("%s\n",y);
else printf("未读进数据\n");
}
/* 程序输出:n=8
      ABCDEFGH      */

```

C>TYPE STD21.C

```

#include "stdio.h"
#include "fcntl.h"
#define PS fscanf(fp,"%s",str);\
      printf("%s\n",str)
main() /* 本程序用于对流格式化输入输出时的情况 */
{
FILE *fp;
char y;
char str[1024];
char *path="QW.C";
—fmode=O—TEXT;
fp=fopen(path,"w");
fprintf(fp,"QW.C, 向文件输入本字符串\n");
/* 注意,"字"和"符"之间有一个空格符 */
fclose(fp); /* 如流未先关闭然后再打开操作则出错。跟打开方式无关 */
fp=fopen(path,"r");
fscanf(fp,"%c",&y);
printf("%c\n",y);
PS;
PS; /* 可以看到串输出时遇空格符便终止 */
PS;
}
/* 程序输出:Q
      W.C,
      向文件输入本字
      符串      */

```

—19 FILE * —Cdecl fdopen(int handle,char * type);

它打开一个流,假定为 stream,则流 stream 与早先用 creat()、open()、dup() 或 dup2() 等打开文件时得到的文件句柄 handle 相联。换言之,stream 与 handle 相关的文件相联,所以 stream->fd=handle。type 是打开模式,它必须与早先打开的文件模式相匹配。

本函数不是由 ANSI 标准所定义的,handle 可以是通过调用 UNIX 型 I/O 函数所获得,因此利用本函数对移植性有利。

如调用成功,返回新打开流指针,否则返回 NULL。

C>TYPE STD22.C

```

#include "stdio.h"
#include "fcntl.h"
main()          /* 本程序说明对新建的流不能在送字符后又取出的原因 */
{
    /* 由于两者打开模式不一致而导致出错 */
    FILE *fp;
    int handle;
    char *path="QW.C",y;
    handle=open(path,O_CREAT|O_WRONLY); /* 原打开方式只能写,而不能读 */
    /* 如将上句改成 handle=open(path,O_CREAT|O_RDWR); 就正确了 */
    fp=fdopen(handle,"w+");
    fputc('H',fp); /* 送字母 H 到流中 */
    rewind(fp);
    y=fgetc(fp); /* 用调试表达式容易看出 fp->flags 有出错标志 -F-ERR */
    printf("%c\n",y); /* 不能得到正确值 */
}

```

—20 FILE *—Cdecl freopen(const char *path,const char *mode,FILE *stream);

用指定文件名 path 与已打开的流 stream 相联,原与流相联的文件便不再与 stream 相联,因为在同一时刻只能有一个文件与一个流相联。相联的过程是这样实现的,先关闭 stream,然后利用指向原流的指针 stream(虽然关闭了流,但其指针 stream 仍存在)、文件名 path 和打开文件方式 mode 打开文件 path。mode 的含义同 fopen() 中的 mode。

注意:不管文件打开成功与否,原与 stream 相联的流都会被关闭,因为是先关闭原与 stream 相联的文件后才去打开名为 path 的文件的。另外,在使用本函数前,名为 path 的文件不能是已打开的,即不允许将两个同时打开的流相联。

这样,现在通过流 stream 来实现的操作等于是对与 stream 相联的文件 path 的操作。跟函数 fopen() 不同的是,fpopen() 是通过返回来获得流的指针,而本函数无需靠返回指针,而是直接用原流的指针名即可操作了!因为两个指针有同样的值。换句话说,多少有的“挂羊头卖狗肉”的味道。

函数常用于改变联在预定义流 stdin、stdout 或 stderr 上的文件。

函数调用成功返回流指针,否则返回 NULL。

适用于 UNIX 系统。

```

C>TYPE STD23.C
#include "stdio.h"
#include "fcntl.h"
main()
{
    FILE *fp;
    int handle;
    static char str[100];
    char *path="QW.C",*stream; /* 当然也可定义:FILE *stream; */
    fp=fopen(path,"w");
    fclose(fp);
    stream=(char *)&-streams[1];
    printf("本字符串输出到屏幕\n");
    fprintf(stdout,"本串也通过stdout 输出到屏幕\n");
}

```

```

/* 预定义流 stdout 缺省是与屏幕相联 */
fp=fopen(path,"rt+",stdout); /* 现在 stdout 改向与path 相联 */
fprintf(stdout,"本串将改向输出到文件QW.C 中\n");
rewind(stdout);
fread(str,1,28,stdout); /* 从流 stdout 中读出字符 */
freopen("CON","rt+",stdout); /* 注意,stdout 的缺省名字是:CON */
/* 此句是重新使输出指向屏幕 */
printf("%s\n",str); /* 没有上句仍输出到文件 QW.C 中 */
printf("%p %p\n",fp,stream); /* 检查是否借用流指针。如相等便是借用 */
}
/* 程序输出:本字符串输出到屏幕
   本串也通过stdout 输出到屏幕
   本串将改向输出到文件QW.C 中
0276 0276

```

假定本程序已编译成执行文件 STD.EXE,如果在 DOS 下用重定向操作

C>STD>Q9.OUT

则用

C>TYPE Q9.OUT

本字符串输出到屏幕

本串也通过stdout 输出到屏幕

结果发现后面的字符串:

本串将改向输出到文件QW.C 中

0276 0276

未被写入Q9.OUT 中

*/

C>TYPE STD24.C

```

#include "stdio.h"
main()
{
FILE *fp, *stream, *fp1;
static char str[100];
char *path="QW.C";
int k;
fp=fopen(path,"w");
fclose(fp);
stream=&—streams[0];
fread(str,1,2,stdin); /* 假定从键盘输入 qwer 后回车 */
fprintf(stdout,"%s\n",str);
fp=fopen(path,"rt+",stdout);
fprintf(stdout,"本串将改向输出到文件QW.C 中");
fclose(fp); /* 必须关闭,因为同一文件名不能同时与两个流相联 */
fp1=fopen(path,"rt+",stdin);
rewind(stdin);
fread(str,1,28,stdin); /* 从流 stdin 中读出字符 */

```

```

fp=fopen("CON","rt+",stdout); /* 重新打开预定义流 stdout */
printf("=====%s\n",str);
printf("%p %p\n",fp1,stream);
fclose(fp);
fp1=fopen("CON","rt+",stdin);
for(k=0;k<5;k++)str[k]='\0'; /* 对 MS-DOS 键盘和屏幕称控制台,共用 */
/* 同一个名字 CON,因此必须清理 */
fread(str,1,4,stdin); /* 假定从键盘输入 ASDFGHJKL 后回车 */
fclose(fp1);
freopen("CON","rt+",stdout);
printf("%s\n",str);
}
/* 程序输出:qw
=====本串将改向输出到文件QW.C 中
0256 0256
ASDF */

```

—21 int —Cdecl ioctl (int handle,int func, ...);

它是一个取得 handle(文件句柄或驱动器号)的设备文件通道的信息。实际上,对 MS-DOS 它是 DOS 功能调用 44H,func 是它的子功能号。必要时它还可以有另外两个参数,即省略号所能描述的参数是 void *argdx 或 int argcx。入口参数是,AH=44H,AL=func,BX=handle,CX=argcx,DX=argdx。这是一个和操作系统十分密切相关的函数,对不同的 MS-DOS 或 UNIX 版本,它们可能有不同在结果,因此在使用它时你最好进行必要的功能验证,以免出错。

DOS 功能调用 44H 的功能简述如下(都要调用寄存器 AH=44H,AL=func):

1. 子功能 0H:取指定文件或设备的特性

调用寄存器:BX=handle(文件句柄)

返回DX=【设备信息字】

当其位7 为 1 时表示字符设备,这时当以下位值为 1 时表示的意义是

位0 标准输入设备

位1 标准输出设备

位2 空设备(NUL)

位3 时钟设备

位4 快速控制台输出(参见中断 INT 29H 的功能)

位5 以二进制方式操作(对 Ctrl-Z 不作检查),反之以 ASCII 方式并检查

位6 输入遇 EOF(文件尾),即文件结束符在输入上出现

位7 为1

DH 的值对应于设备驱动程序属性字的高字节

位11 驱动程序支持 OPEN/CLOSE 调用

位13 支持连续输出、遇忙停止的功能

位14 设备驱动程序支持 IOCTL 功能调用

当位7 为 0 时表示磁盘文件

位0~位5 驱动器号(0 表示 A 驱动器、1 表示 B 驱动器等)

位6 文件未写入

位7 为 0

位8 对 DOS 4+, 写文件时如无磁盘空间则产生 INT 24H

位11 介质不可移动

位14 关闭时不设置文件日期/时间 (DOS 3+)

位15 文件为远程 (DOS 3+)

2. 子功能 1H: 设置设备信息

调用寄存器: BX=handle (必须是设备文件句柄, 0~4)

DX=设备信息字 & 0x00ff (即 DH=0)

返回设备信息字

3. 子功能 2H: 从指定设备 (handle) 的控制通道中读 argcx 个字节到 argdx 所指的地址中, 数据格式为驱动程序所指定。设备驱动程序必须支持 IOCTL 调用。

调用寄存器: BX=handle (必须是设备文件句柄, 0~4)

CX=argcx (要读的字节数)

DS:DX=argdx (缓冲区)

返回实际读出字节数

4. 子功能 3H: 从 argdx 所指的地址中写 argcx 个字节写到指定设备通道中, 数据格式为驱动程序所指定。设备驱动程序必须支持 IOCTL 调用。

调用寄存器: BX=handle (必须是设备文件句柄, 0~4)

CX=argcx (要写的字节数)

DS:DX=argdx (缓冲区)

返回实际写入字节数

5. 子功能 4H: 从指定驱动器 (handle) 的控制通道中读 argcx 个字节到 argdx 所指的地址中, 数据格式为驱动程序所指定。设备驱动程序必须支持 IOCTL 调用。

调用寄存器: BL=handle (驱动器号, 00 = 默认, 01 = A 驱动器等)

CX=argcx (要读的字节数)

DS:DX=argdx (缓冲区)

返回实际读出字节数

6. 子功能 5H: 从 argdx 所指的地址中写 argcx 个字节到指定驱动器控制的通道中, 数据格式为驱动程序所指定。设备驱动程序必须支持 IOCTL 调用。

调用寄存器: BL = handle (驱动器号, 00 = 默认, 01 = A 驱动器等)

CX=argcx (要写的字节数)

DS:DX=argdx (缓冲区)

返回实际写入字节数

7. 子功能 6H: 得到文件的输入状态

调用寄存器: BX=handle

返回: 如 AL = 00 表示未准备好 (对设备文件) 或 EOF (对磁盘文件)

如 AL = FF 表示已准备好

8. 子功能 7H: 得到文件的输出状态

调用寄存器: BX=handle

返回: 如 AL=00 表示未准备好

如 AL=FF 表示已准备好

9. 子功能 8H: 检查块设备是否可移动, 如软驱动器是可移动的, 硬盘是不可移动的。

调用寄存器:BL=handle (驱动器号,00 = 默认,01 = A 驱动器等)

返回:如 AX=0000 则是可移动的

如 AX=0001 则是不可移动的

10. 子功能 BH:规定在打开或锁定文件过程中如果共享失败时的重试次数

调用寄存器:CX=argcx (重试间隔,默认值为 1)

DX=argdx (重试次数,默认值为 3)

在任何情况下,如果检测到一个错误,函数返回 -1,并置 errno=19(EINVAL)、6(EBADF) 或 13(EINVAL)。

对 MS-DOS 一般要用 DOS 3.2 以上版本。

C>TYPE STD25.C

```
#include "io.h"
#define P printf("%X\n",stat)
main()
{ int handle,stat; /* 假定 qq.c 内容为 ABCDEF */
  if( (handle=open("qq.c",1,1))<0)exit(1);
  stat=ioctl(0,8,0,0); /* 假定默认驱动器是硬盘 C */
  P;
  stat=ioctl(1,8); /* 测试 A 驱动器 */
  P;
  stat=ioctl(0,0);
  P;
  stat=ioctl(handle,0);
  printf("handle=%d, ",handle); P;
}
/* 程序输出:1
0
80D3
handle=5, 42 */
```

附注:设备驱动程序

一个使应用程序的高级需要与硬件接口的低级需要相配合的软件接口程序称【设备驱动程序】。安装设备驱动程序可以用 CONFIG.SYS 文件,通常将像

DEVICE= 设备驱动程序名 参数

这样的语句放到 CONFIG.SYS 文件中。例如要将鼠标驱动程序在机器启动时安装到内存中,可以用

DEVICE=MOUSE.SYS

另一个名为 ANSI.SYS 的设备驱动程序是一个用来取代标准控制台设备(CON)以提高显示性能和改变键盘功能的字符设备驱动程序。MS-DOS 启动时执行 CONFIG.SYS 的过程如图 30-2 所示。

在汇编设备驱动程序时,其文件开头不允许使用 ORG 100H,因为它不能使用程序段前缀(PSP),设备驱动程序可以装在内存的任何地方。不同设备驱动程序的长度可以不同。

为了记忆所有的设备驱动程序,DOS 使用了一个链接表来管理。

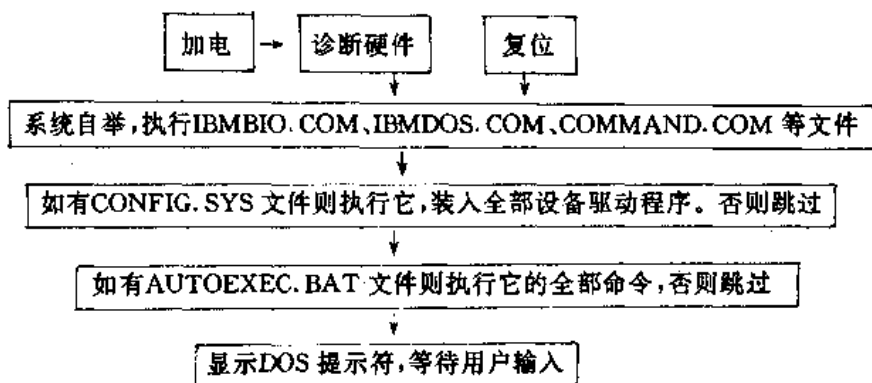


图 30-2 MS-DOS 启动时执行任务示意图

设备驱动程序一般由设备标题 (device header)、策略程序 (strategy routine) 和 中断程序 (interrupt routine) 组成 (图 30-3)。

1. 设备标题

它由设备驱动程序开头的 18 个字节构成。这 18 个字节又分成 5 部分。

(1) 下一个设备标题字段 (4 个字节)

它是指向下一个设备标题的指针 (偏移值在前, 段址在后), 如无下一个设备驱动程序, 则置为 -1 (即 FFFFH)。当有多个设备驱动程序时, 可用它建立一个地址链接表。当 DOS 装入下一个驱动程序时就将其驱动程序的起始地址放在此处, 这样, DOS 就可沿着地址链找到一个特定的驱动程序。最后装入的驱动程序的设备标题段值为 -1。

(2) 设备属性 (2 个字节)

【字符设备】是以一个接一个的字符方式进行输入和输出的。标准字符设备名如 CON (控制台, 它使用键盘输入并用显示屏幕输出), AUX 或 COM1 (辅助通信口, 通过它可连接绘图机、鼠标或调制解调器), PRN 或 LPT1 (并行打印机口), 时钟设备 CLOCK\$ 等。

【块设备】指软、硬磁盘或 RAM 盘 (也称【虚拟盘】) 等大容量存储单元, 每次传输数据是以块为单位进行的, 通常一块包含一个磁盘数据扇区 (512 字节)。块设备不特别命名, 而是通过设备字母 (A、B、C 等) 映射而成, 称【设备号】。

【CLOCK\$ 设备】定义和执行功能都像其它字符设备一样, 能置位、复位或取数等。当发生对这个设备进行读写时, 准确地传送 6 个字节, 前两个字节是一个字, 它表示从 1980 年 1 月 1 日以后的天数计算; 第三个字节是分针; 第四个字节是小时; 第五个字节是百分之一秒; 第六个字节秒。读 CLOCK\$ 设备能得到日期和时间, 对它写数能置入日期和时间。

位15	1=字符设备	0=块设备
位14	1=允许 IOCTL	同左
	0=不支持 IOCTL (参见 DOS 中断 INT 44H 和库函数 ioctl())	
位13	1=(DOS 3+)支持连续输出, 直到遇忙为止	1=IBM 块格式 (影响 BIOS 参数数据块)
位12	保留	同左
位11	1=支持 OPEN/CLOSE 调用	1=支持 OPEN/CLOSE 调用
位10	保留	同左
位9	保留	1=不允许直接 I/O
位8	保留	
位7	1=(DOS 5.0)支持 IOCTL 校验调用	同左
位6	1=(DOS 3.2+)支持 IOCTL 调用	同左

位 5	保留	同左
位 4	设备特殊 (支持 INT 29H 快速控制台输出)	保留
位 3	1=时钟设备 CLOCK\$	保留
位 2	1=NUL 设备 (NUL 是 DOS 用于测试的空设备)	保留
位 1	1=设备为标准输出设备	驱动程序支持 32 位扇区寻址
位 0	1=设备为标准输入设备	保留

(3) 设备策略指针 (2 个字节)

这是一个进入该设备驱动程序到策略程序的偏移地址,策略程序必须与该设备的标题在同一段内。在 MS-DOS 2.0 下,策略程序只是排队设备请求和返回到 DOS;在 DOS 高版本中,它能够帮助实行优先排队多个任务或分时状态那样的操作。

(4) 设备中断指针 (2 个字节)

这是一个进入该设备驱动程序到中断程序的偏移地址,中断程序与设备标题在同一段内。当从策略程序接收控制返回时, DOS 调用该驱动程序的中断程序,这个程序为该驱动程序提供所有的性能,即执行该设备驱动程序操作的代码。

(5) 设备名字段 (8 个字节)

对一个字符设备,它包含该设备名;对块设备,它的第一个字节为由该驱动程序支持的设备号的个数,剩余的 7 个字节未用。

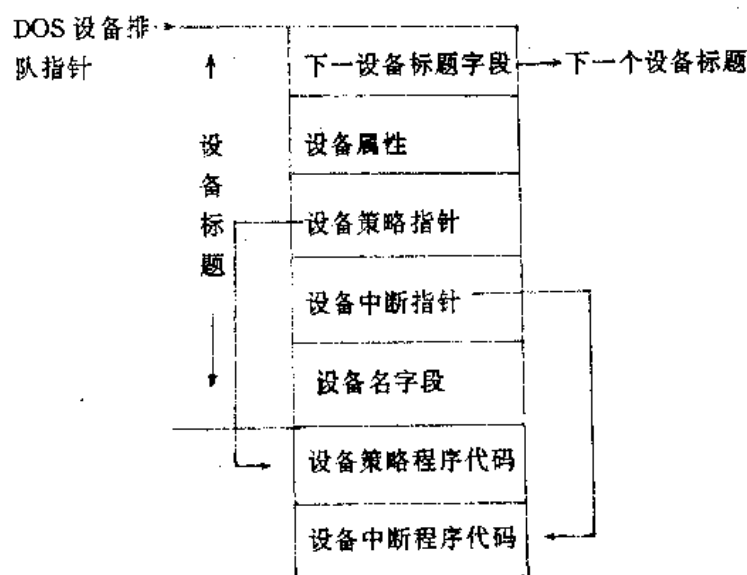


图 30-3 设备驱动程序的结构示意图

在 dos.h 中结构 devhdr 与设备驱动程序标题对应 (但该结构没有相应的库函数):

```
struct devhdr {
    long        dh-next;    /* 下一个设备驱动程序的地址 */
    short       dh-attr;    /* 驱动程序属性 */
    unsigned short dh-strat; /* 驱动程序设备策略例程入口 */
    unsigned short dh-inter; /* 驱动程序中断入口点 */
    char        dh-name[8]; /* 驱动设备名 */
};
```

—22 int —Cdecl isatty(int handle);

它确定和文件句柄 handle 的文件是不是字符设备文件,即可能是终端、控制台、打印机或串行口等。如果是,返回一个非 0 整数,否则返回 0。

C>TYPE STD26.C

```
#include "io.h"
main()
{ int handle,k; /* 假定 qq.c 内容为 ABCDEF */
  if( (handle=open("qq.c",1,1))<0)exit(1);
  for(k=0;k<5;k++) /* 输出缺省设备文件句柄 */
    printf("handle=%d, %d\n",k,isatty(k));
  printf("handle=%d, %d\n",handle,isatty(handle));
}
/* 程序输出,handle=0, 1
      handle=1, 1
      handle=2, 1
      handle=3, 1
      handle=4, 1
      handle=5, 0 */
```

—23 void —Cdecl setbuf(FILE *stream,char *buf);

像 setvbuf() 一样,在流 stream 被打开后可以用程序员指定的缓冲区 buf 作缓冲。跟 setvbuf() 不同的是,它规定 buf 的大小一定为 BUFSIZ,该符号常量在 stdio.h 中定义为

```
#define BUFSIZ 512
```

另外,当 buf=NULL 时不缓冲,否则缓冲。该函数相当于执行语句

```
setvbuf(stream,buf,(buf != NULL)? _IOFBF:_IONBF,BUFSIZ);
```

由此可见,当你重定义 BUFSIZ 后也能改变缓冲区大小,但对磁盘文件来说,512 是个很好的数字,你不应当轻易改变它。

注意:它不返回值。

—24 int —Cdecl setvbuf(FILE *stream,char *buf,int type,size_t size);

把已打开流 stream 与程序员指定的缓冲区 buf 相联系起来,这时指向流的缓冲区头部的指针 stream->buffer 将指向 buf 所指的存储区域(即它们有同一值)。

type 是文件缓冲的类型,它只能取下列三个值之一(定义在 stdio.h 中):

```
#define _IOFBF 0 /* 文件全部缓冲。当缓冲区为空时,下一次输入操作试图填满整个缓冲区;对于输出,在写数据到文件之前,先填满整个缓冲区 */
```

```
#define _IOLBF 1 /* 文件行缓冲。当缓冲区为空时,下一次输入操作试图填满整个缓冲区;对于输出,当换行符('\n')写到文件上时,缓冲区被清除 */
```

```
#define _IONBF 2 /* 文件不缓冲。参数buf和size被忽略,每一输入操作直接从文件读出,而每一输出操作将直接把数据写到文件中 */
```

函数调用成功返回 0,否则返回非 0 值。

C>TYPE STD27.C

```
#include "stdio.h"
#include "stdlib.h"
#define PBUF printf("c=%c,in->buffer=%p\n",c,in->buffer)
extern void (*_exitbuf)();
```

```

extern void --xfflush();
int --stdinStarted=0; /* 它们将影响 fgetc() 和 fputc() 那样的函数 */
int --stdoutStarted=0;
int setvbuf(register FILE *fp, char *buf, int type, size_t size)

{
    /* 等效函数 */
    if((fp->token != (short)fp || type >= IONBF || size > 0x7FFF)
        return (EOF); /* 检查参数的有效性 */
    if(! --stdoutStarted && ((short)fp == (short)stdout))
        --stdoutStarted = 1;
    else if(! --stdinStarted && ((short)fp == (short)stdin))
        --stdinStarted = 1;
    if((fp->level)fseek(fp, 0L, SEEK_CUR); /* 确保改变缓冲区时不丢失原来 */
        /* 连在文件上的缓冲区中的字符 */
    if((fp->flags & --F-BUF)free(fp->buffer); /* 释放先前分配的缓冲区 */
    fp->flags &= ~(--F-BUF | --F-LBUF);
    fp->bsize = 0;
    fp->curp = fp->buffer = &fp->hold;
    if((type != --IONBF && size > 0) /* 如果要缓冲 */
        { --exitbuf = --xfflush;
            if(buf == NULL) /* 如果 buf 写成 NULL 则调用 malloc 分配缓冲区 */
                { if((buf = malloc(size)) != NULL) /* 如分配缓冲区成功 */
                    fp->flags |= --F-BUF; /* 设置标志 */
                    else return (EOF);
                }
            fp->buffer = fp->curp = buf;
            fp->bsize = size;
            if((type == --IOLBF)fp->flags |= --F-LBUF;
        }
    return (0);
}

main() /* 可用 F7 单步调试观察缓冲区 */
{
    char data[600], *sp = data;
    char c = 'M';
    FILE *in, *out;
    in = fopen("qq.c", "r"); /* 假定 qq.c 内容为 ABCDEFGH */
    PBUF;
    c = fgetc(in);
    PBUF;
    setvbuf(in, sp, --IOLBF, 514);
    PBUF;
    c = fgetc(in);
    PBUF;
}
/* 程序输出: c=M, in->buffer=055A

```

```

c=A,in->buffer=055A
c=A,in->buffer=FD88
c=B,in->buffer=FD88      */

```

下面对 Turbo C 与流相联的缓冲区作一些说明。

FILE 中除了有流的当前指针 curp 外,还有指向流数据缓冲区头部的指针 buffer 和描述缓冲区大小,bsize,以及表明流缓冲区是空或满的标志 level 和当 level=0(缓冲区为空)时记录回退字符的变量 hold。要想了解为什么流能使读写数据(I/O)灵活、高效地进行,就应当了解与 FILE 众多成员都相关的缓冲区的情况,以及如何正确使用 setvbuf() 和 setbuf() 这两个重置与流相联缓冲区的函数。

下面主要对磁盘文件进行讨论,文中涉及的符号常量均在标头文件 stdio.h 中有宏定义。

Turbo C 的库函数 setvbuf() 或 setbuf() 允许程序员对打开的流重新指定新的流缓冲区,或者说现在的流缓冲区是程序员自己指定的,而不是流刚打开时系统自动分配的。为区别于原缓冲区,可以称重置的缓冲区是与流相联的另一种缓冲区。对同一个流只有一个缓冲区与之相联。重置流缓冲区主要出于两个目的,一是为了设置新的缓冲方式,二是为了改变缓冲区大小。文件可以被看成字节流。一般磁盘文件在流打开后缺省地设文件全部缓冲(—IOF-BF),而用这两个函数还可选择行缓冲方式(—IOLBF)或不缓冲(—IONBF);缺省打开的流缓冲区的大小是 BUFSIZ(512 字节),而用函数 setvbuf() 可使缓冲区大小在 1~65535 之间变化。

由于 setbuf() 可以认为是 setvbuf() 的特殊情形,因此,下面我们只讨论 setvbuf()。

1. 因为在流被打开时,系统调用 malloc() 分配缓冲区,设置全缓冲方式,并使 flags 中包含 —F—BUF。这就是 stdio.h 中对 —F—BUF 的注释

Malloc'ed Buffer data

的真实含意。首次调用 setvbuf() 时先前系统自动分配的缓冲区将被释放。当用 fclose() 关闭一个流时,只释放先前那些使 (flags & —F—BUF) = —F—BUF 的与流相联的缓冲区,因为这些缓冲区是系统用 malloc() 分配的。

2. 以后可多次对同一个流使用 setvbuf() 重置与流相联缓冲区,但只有使用诸如

```
setvbuf(stream, NULL, byte, size);
```

这样的语句时函数才先释放以前用 malloc() 分配的与流相联的缓冲区,然后重新用 malloc() 分配新的与流相联的缓冲区。否则,当 buf 非空 (NULL) 时就不会有这种操作过程。参见程序 BUF1.C: C>TYPE BUF1.C

```

#include "stdio.h"
#include "alloc.h"
#define PBUF c=fgetc(in);printf("c=%c, in->buffer=%p, ",c,in->buffer);\
    if(in->flags & —F—BUF)printf("标志位上有—F—BUF, ");\
    else printf("无!");k=coreleft();printf("k=%d\n",k)
main() /* 此程序用于观察 —F—BUF 的存在和与流相联缓冲区的释放情况 */
{
    FILE *in;
    char sp[BUFSIZ],c;
    char *ptr;
    int k;
    in=fopen("qq.c","r"); /* 假定 QQ.C 内容为 ABCDEFGH */

```

```

PBUF,
ptr=(char *)malloc(1024);
setvbuf(in,ptr,-IOFBF,BUFSIZ);
PBUF,
setvbuf(in,NULL,-IOFBF,BUFSIZ*4);
PBUF,
setvbuf(in,ptr,-IOFBF,BUFSIZ);
PBUF,
setvbuf(in,sp,-IOFBF,BUFSIZ);
PBUF,
}
/* 程序输出: c=A, in->buffer=0612, 标志位上有一F-BUF, k=0xf49a
c=B, in->buffer=081A, 无! k=0xf092
c=C, in->buffer=0C22, 标志位上有一F-BUF, k=0xe88a
c=D, in->buffer=081A, 无! k=0xf092
c=E, in->buffer=FDDE, 无! k=0xf092 */

```

3. 缓冲意指数据 I/O 先与缓冲区打交道, 然后累积在缓冲区的数据块再与文件打交道。这样零碎的数据 I/O 由于是在内存区进行, 速度就很快。而用块读写文件就避免了频繁读写磁盘。指向 FILE 的指针则对流进行有条不紊的管理。不缓冲意指读写时不用缓冲区, 读写直接与文件打交道。不缓冲也称关闭缓冲区。缓冲区被关闭时先前与文件相联的流仍是打开的, 也不会引起任何问题。因为那时 FILE 的部分成员已置为

```

stream->flags &= ~(F-BUF | F-LBUF); /* 是否缓冲的标志位被置 0 */
stream->bsize=0; /* 缓冲区尺寸为 0 */
stream->curp=stream->buffer=&stream->hold; /* 当前指针位置 */

```

这就是说, 当需要时, 你还可以重置与此流相联的缓冲区, 并改变缓冲方式。参见程序 BUF2.

C: C>TYPE BUF2.C

```

#include "stdio.h"
#define PCHAR ch=fgetc(fp); -prbuf() /* 取字符并输出 */
FILE *fp;
int n=8;
char str[BUFSIZ],ch='Y';
void -prbuf()
{
printf("n=%d, ",n);
printf("fp->flags=%#x,bsize=%d,curp=%p, ",fp->flags,fp->bsize,fp->curp);
printf("&fp->hold=%p,fp->hold=%c,ch=%c\n",&fp->hold,fp->hold,ch);
}
main()
{
fp=fopen("qq.c","r"); /* 假定 QQ.C 的内容为 ABCDEFGH */
-prbuf();
ungetc('T',fp); /* 回退字符 T */
n=setvbuf(fp,str,2,256); /* 选 -IONBF, 不缓冲 */
-prbuf();
PCHAR;

```

```

n=setvbuf(fp,str,0,256);    /* 选 —IOFBF,全缓冲 */
—prbuf();
PCHAR,
ungetc('S',fp);            /* 回退字符 S */
PCHAR,
PCHAR,
}
/* 程序输出,
n=8, fp->flags=0x5,bsize=512,curp=0756,&fp->hold=02A3,fp->hold= ,ch=Y
n=0, fp->flags=0x1,bsize=0,curp=02A3,&fp->hold=02A3,fp->hold=T,ch=Y
n=0, fp->flags=0x81,bsize=0,curp=02A3,&fp->hold=02A3,fp->hold=T,ch=A
n=0, fp->flags=0x81,bsize=256,curp=0484,&fp->hold=02A3,fp->hold=T,ch=A
n=0, fp->flags=0x81,bsize=256,curp=0485,&fp->hold=02A3,fp->hold=T,ch=B
n=0, fp->flags=0x81,bsize=256,curp=0485,&fp->hold=02A3,fp->hold=T,ch=S
n=0, fp->flags=0x81,bsize=256,curp=0486,&fp->hold=02A3,fp->hold=T,ch=C
*/

```

4. 当 buf 和流相联后程序员不应该再在程序中对它任意赋值,或在内存中移动缓冲区,而要把它看作 FILE 的一个特殊的“成员”! 否则搞得不好会出错。对 FILE 的成员赋值应当十分小心,必须注意各成员之间十分严格的相互依赖关系,初接触 Turbo C 的程序员尤其要注意。调用 setvbuf() 后 buffer 便和 buf 有相同值,即它们指向同一存储区。当 buf 非空时应有确定的存储区,并且该存储区大小不应小于 size。所以,通常把 buf 定义为字符数组,其大小为等于 size 的一个具体的数值。如果 buf 定义为指针,则再好定义时用 malloc() 给它分配一个确定的存储区,以免意外。参见程序 BUF3.C 和 BUF4.C: C>TYPE BUF3.C

```

#include "stdio.h"
#include "string.h"
#define PBUF printf("c=%c,in->buffer=%p,sp=%p,"c,in->buffer,sp): \
printf("in->curp=%p,in->level=%d\n",in->curp,in->level)

main()
{
char *sp=(char *)malloc(514);
char r[]="xyz";
char c='M';
FILE *in, *out;
int k;
in=fopen("qq.c","r");    /* 假定 QQ.C 内容为 ABCDEFGH */
c=fgetc(in);
PBUF;
k=setvbuf(in,sp,—IOLEBF,514);
PBUF;    /* 注意输出 in->level=0 及 in->curp 值的变化 */
strcpy(sp,r);    /* 局部变量 sp 赋新值 */
c=fgetc(in);    /* 通过调试表达式可观察 sp 所指内容的变化 */
PBUF;    /* sp 重新赋值后输出还算正确 */
strcpy(sp,r);    /* sp 又赋新值 */
c=fgetc(in);
PBUF;    /* sp 再次重新赋值后输出便不是文件中字符 */
}

```

```

k = fclose(in);          /* 尚能关闭文件,不会引起死机          */
printf("k=%d\n",k);
}
/* 程序输出:c=A,in->buffer=07EC,sp=05E4,in->curp=07ED,in->level=8
   c=A,in->buffer=05E4,sp=05E4,in->curp=05E4,in->level=0
   c=B,in->buffer=05E4,sp=05E4,in->curp=05E5,in->level=7
   c=y,in->buffer=05E4,sp=05E4,in->curp=05E6,in->level=6
   k=0          */

```

C>TYPE BUF4.C

```

#include "stdio.h"
#include "string.h"
#define PCHAR ch=fgetc(fp), --prbuf(fp,str,ch)
void --prbuf(FILE *fp,char *s,char c)
{
int k;
printf("fp=%p,c=%c\n",fp,c);
for(k=0;k<512;k++)
strcat(s,"U");          /* 对与流相联缓冲区再分配      */
}
main()
{
FILE *fp;
char str[BUFSIZ],ch='Y';
fp=fopen("qq.c","r");    /* 假定 QQ.C 的内容为 ABCDEFGH */
setvbuf(fp,str,--IOFBF,256);
PCHAR;
PCHAR;
fclose(fp);              /* 不能关闭文件,可能会死机 */
}

```

5. 当你在源程序中重定义 BUFSIZ 后也能改变缓冲区大小,但对磁盘文件来说,512 是一个很好的数字,你不应当轻易改变它。而且在使用 setbuf() 时应确保缓冲区的大小为 BUFSIZ,否则有时会产生不可预料的效果。参见程序 BUF5.C:

C>TYPE BUF5.C

```

#include "stdio.h"
main()          /* 本程序说明 setbuf 中缓冲区定义不 */
{              /* 是 BUFSIZ 时将引起不可预料的结果 */
FILE *in;
char sp[2]={'q','w'},sq[2]={'e','r'},c;
in=fopen("qq.c","r");    /* 假定 QQ.C 内容为 ABCDEFGH */
setbuf(in,sp);
c=fgetc(in);
printf("%c %c\n",c,sq[0]); /* 输出的 sq[0] 内容将不对! */
}

```

/* 在用 F7 单步追踪时,程序输出 A C 后进到用户屏幕,追踪便无法进行下去,须按 Ctrl-Break 才能返回集成环境 */

—25 int —Cdecl dup (int handle);

复制一个文件句柄,该文件句柄和 handle 的值不同。它们的共同点是,指向同一个打开的文件(或设备)。因此,新老句柄中的一个文件指针的移动都将同时移动另一个句柄的指针;相同的存取模式(读、写、读/写)。

若调用成功,它返回新文件句柄,否则返回 -1。在出现错误时,errno=4(EMFILE)或 errno=6(EBADF)。

对 MS-DOS,它实际是 DOS 功能调用 45H,入口参数是,AH=45H,BX=handle。
它适用于所有的 UNIX 系统。

C>TYPE STD28.C

```
#include "io.h"
#include "dos.h"
#include "sys\stat.h"
#include "fcntl.h"
main()
{ int handle—old,handle—new;
  handle—old=open("qq.c",O—RDONLY,S—IREAD);
  if(handle—old == -1)exit(1);          /* 若调用失败则终止程序 */
  handle—new=dup(handle—old);
  printf("%d %d\n",handle—old,handle—new);
  lseek(handle—new,1L,SEEK—CUR);      /* 从当前文件指针位置开始移动指针 */
  printf("%ld %ld\n",tell(handle—new),tell(handle—old));
  lseek(handle—old,1L,SEEK—CUR);      /* tell() 报告文件指针当前位置 */
  printf("%ld %ld\n",tell(handle—new),tell(handle—old));
}
/* 程序输出:5 6
           1 1
           2 2          */
```

—26 int —Cdecl dup2 (int oldhandle,int newhandle);

迫使文件句柄 newhandle 成为指定文件句柄 oldhandle 的复制句柄。当调用本函数时,原来与 newhandle 相联的文件将被关闭,newhandle 指向 oldhandle 的文件。
newhandle 本身的值不变。

对 MS-DOS,它实际是 DOS 功能调用 46H,入口参数是,AH=46H,BX=oldhandle,CX=newhandle。如调用成功返回 0,否则返回 -1。在出现错误时,errno=4(EMFILE)或 errno=6(EBADF)。

适用于除系统 ■ 外的一些 UNIX 系统。

C>TYPE STD29.C

```
#include "io.h"
#include "dos.h"
#include "sys\stat.h"
#include "fcntl.h"
#define PON printf("%d %d\n",handle—old,handle—new)
main()
{ int handle—old,handle—new;
  static char s—old[10],s—new[10];
```

```

handle—old=open("qq.c",O—RDONLY,S—I—READ); /* 假定 qq.c 内容为 ABCDEF */
handle—new=open("q1.c",1,0x0100);          /* 假定 q1.c 内容为 abcdef */
if(handle—old == -1 || handle—new == -1)exit(1);
PON;
lseek(handle—new,1L,SEEK—CUR); /* 将文件指针从文件开始处后移 1 */
read(handle—new,s—new,1);          /* 读出一个字符 */
printf("%s\n",s—new);
dup2(handle—old,handle—new);
PON;
lseek(handle—old,1L,SEEK—CUR);
read(handle—old,s—old,1);          /* 函数调用后文件指针自动后移 1 */
lseek(handle—new,1L,SEEK—CUR—1); /* 将文件指针退回 1 */
read(handle—new,s—new,1);
printf("%s %s\n",s—old,s—new);
}
/* 程序输出:5 6
           b
           5 6
           B B */

```

—27 #define fileno(f) ((f)—>fd)

它被定义为通过流 (f) 取得与该流相联文件的文件句柄 (f)—>fd 的宏。如果流 f 有多个句柄。则它返回该流首次被打开时所赋的文件句柄。

它不是 ANSI 标准定义的。适用于 UNIX 系统。

```

C>TYPE STD30.C
#include "stdio.h"
main()
{
char fd;
FILE *ptr;
ptr=fopen("QQ.C","r");          /* 假定文件 QQ.C 存在 */
if(ptr == NULL)
{printf("打开文件失败\n");exit(1);}
fd=fileno(ptr);
printf("与流ptr 相联文件句柄=%#x\n",fd);
}
/* 程序输出:与流 ptr 相联文件句柄 = 0x5 */

```

—28 int —Cdecl —close (int handle);

关闭文件句柄为 handle 的文件:清所有缓冲区、更新文件目录 (置文件时间和日期为当前时间和日期),并使 handle 无效。在关闭文件时它不会自动在文件末尾加上 Ctrl-Z 字符 (即 1AH)。如果想在文件尾加上这个字符,必须使用 write()、putc()等函数先将该字符写入文件,然后将文件关闭。

handle 可由 —creat()、creat()、creatnew()、creattemp()、dup()、dup2()、—open() 或 open() 等函数调用后得到,它是一个非负整数。

调用成功时返回 0,否则返回 -1,并置 errno=6(EBADF)。失败时通常是指定的 han-

dle 有错误。

它实际进行 DOS 功能调用 3EH, 入口参数是, AH=3EH, BX=handle。它只适用于 MS-DOS。

—29 int —Cdecl close (int handle);

它的功能同 —close(), 只是它也适用于 UNIX 系统。

—30 int —Cdecl fclose(FILE * stream);

FILE 结构的成员是由 Turbo C 在程序运行时自动设置的, 一般情况下用户在自编程中不要对它们轻易设置特定值! 否则是很危险的。本函数关闭流 stream, 从而关闭与流相联的文件(即相当执行了语句 close(stream->fd);)。在文件打开成功时, 系统自动分配缓冲区(相当于 stream->flags 的位 2 置位, 即为 —F—BUF)。当调用本函数后, 该缓冲区即被释放(相当于执行 free(stream->buffer); 语句)。与此同时, 虽然与流 stream 相联的缓冲区在流关闭时一般将被清除, 但是, 由 setbuf() 或 setvbuf() 设置的与 stream 相关的缓冲区不会被自动释放。在关闭文件时还将关闭暂时文件。最后有 stream->flags=0; stream->bsize=0; stream->level=0; stream->istemp=0; stream->fd=-1。

注意: 文件不用时在程序结束前最好用本语句关闭文件, 否则有可能造成数据丢失, 即有些数据并未真正写入文件。

当调用成功时, 它返回 0, 否则返回 EOF。

适用于 UNIX 系统。

C>TYPE STD31.C

```
#include "stdio.h"
#define P printf("level=%d #x, %d #x\n", in->level, out->level); \
printf("flags=%d #x, %d #x\n", in->flags, out->flags); \
printf("fd=%d #x, %d #x\n", in->fd, out->fd); \
printf("hold=%d #x, %d #x\n", in->hold, out->hold); \
printf("bsize=%d #x, %d #x\n", in->bsize, out->bsize); \
printf("buffer=%p, %p\n", in->buffer, out->buffer); \
printf("curp=%p, %p\n", in->curp, out->curp); \
printf("istemp=%d #x, %d #x\n", in->istemp, out->istemp); \
printf("token=%d #x, %d #x\n", in->token, out->token)

main()
{
    FILE *in, *out;
    char c;
    int k=1;
    in=fopen("qq.c", "rb"); /* 假定文件 QQ.C 存在, 且以二进制形式打开 */
    out=fopen("qw.c", "wb");
    P;
    while( ! feof(in) )
    {
        c=getc(in);
        if(k==1)
            {P; /* 此句写成 P, 即非复合语句效果会不一样 */
            if(ferror(in))clearerr(in);
            else {
```

```

        putc(c,out);
        if(k==1)
            {P;k++;}
        if( ferror(out) )clearerr(out);
    }
}
fcloseall();
P;
}
/* 程序输出,level=0,0
    flags=0x45,0x46
    fd=0x5,0x6
    hold=0,0
    bsize=0x200,0x200
    buffer=0712,091A
    curp=0712,091A
    istemp=0,0
    token=0x45c,0x46c
    level=0x33,0
        fd=0x5,0x6
    hold=0,0
    bsize=0x200,0x200
    buffer=0712,091A
    curp=0713,091A
    istemp=0,0
    token=0x45c,0x46c
    level=0x33,0xfe00
    flags=0xc5,0x146
    fd=0x5,0x6
    hold=0,0
    bsize=0x200,0x200
    buffer=0712,091A
    curp=0713,091B
    istemp=0,0
    token=0x45c,0x46c
    level=0,0
    flags=0,0
    fd=0xffff,0xffff
    hold=0,0
    bsize=0,0
    buffer=0712,091A
    curp=0712,091A
    istemp=0,0
    token=0x45c,0x46c    */

```

—31 int —Cdecl fcloseall(void);

它关闭除 5 个预定义流外的所有已打开的流。它相当于对所有非预定义流分别调用了 函

数 `fclose()`。调用成功返回已关闭流数。

```
—32 int —Cdecl fflush(FILE *stream);
```

函数刷新一个打开的流 (flushes a stream), 简称【刷新流】或【整理流】。这个整理流的过程包含这样几个内容: 对行缓冲文件或有回退字符的 (回退字符存在 `stream->hold` 中) 的输入流, 它将 `stream->level=0`, 这就是说, 流现在为“空”。当有回退字符存在时, 还将忽略回退字符, 即将原指向存储回退字符变量 `stream->hold` 的地址的流的当前指针 `stream->curp` 改为指向流缓冲区, 即使 `stream->curp=stream->buffer`; 而对输出流便将缓冲区中的内容写入与流关联的文件, 并自动调整流当前指针, 即 `stream->curp=stream->stream->buffer`, 并改写流是否为空的标志 `stream->level`。值得指出的是, 缓冲区中的原来存储的内容并未改变, 流本身仍是打开的, 并未关闭。

由此可见, 整理流实际是调整缓冲区的当前指针和缓冲区空或满的标志, 换言之是整理流的缓冲区。

除了用 `fflush()` 或 `flushall()` 来整理流外, 在下列情形流将被自动整理:

1. 缓冲区满; 2. 流被关闭; 3. 程序正常结束。

在一个流打开时都要指出流是用于读入数据还是用于输出数据。为方便起见, 可以以流的缓冲区为基准来区分【输入流】或【输出流】。当用像 `fgetc()`、`fgetchar()`、`fgets()` 等函数从流中取得字符或字符串时, 该流便称为输入流。执行这些函数时文件数据将按数据块形式先行进入流缓冲区, 填满缓冲区, 然后再被逐一读走。由于缓冲区是在内存中, 且有固定的长度, 因此不但存取方便, 而且存取速度就可加快; 而用像 `fputc()`、`fputchar()`、`fputs()` 等函数是将取得的数据逐一填充流缓冲区, 一般填满后按数据块再写入与流相联的文件, 这样的流便是输出流。输入流又称【读流】, 一般打开文件时是用于读或可读/写的; 相应的输出流亦可称【写流】, 打开的文件一般用于写或可读/写的。

由于一个文件还可以在打开时是可读可写的, 这时与之相联的流能否又读又写? 一般来说, 应尽可能避免这样做。对同一个流不能任意地混合读、写操作, 如果实在需要, 必须在读、写之间调用 `fflush()`、`flushall()`、`fseek()` (因为执行该函数时要调用 `fflush()` 或 `rewind()` 等函数整理缓冲区。为了具有最大的可移植性, 即使用 `setbuf()` 或 `setvbuf()` 设置成不缓冲, 也要调用这些函数刷新它 (注意, 不缓冲不等于缓冲区 `stream->buffer` 不存在, 它只是忽略 `buffer`, 而将写到流中的字符立即输出到磁盘文件或设备中而已。实质上这时 `stream->flags |= _IONBF`)。

当然你也不要 在一个程序中随意使用本函数, 否则会像下面例程中得不到正确的结论的 (这里指 `QW.C` 不和 `QQ.C` 一样)。

函数调用成功返回 0, 否则返回 -1。

```
C>TYPE STD32.C
#include "stdio.h"
#include "io.h"
static char buf[512];
char *ptr=buf;
/* 可用 fp[0] 或 fp[0], x 这样的调试表达式在用 F7 调试时观察 fp 所指成员的情况。在缓冲区 buf 与
   流 in 相联后还有:
       fp->buffer, p, DS:038E &—ScanTodVector */
/*           ptr, p, DS:038E &—ScanTodVector */
int fflush(register FILE *fp) /* 等效函数 */
{
```

```

register count;
if( fp->token != (short)fp )return EOF;          /* 检查 fp 的合法性 */
/* EOF 在 stdio.h 中定义为文件指示器终止符, #define EOF (-1) */
/* 如果 打开的是输入流, 则首先对输入流进行处理 */
if( fp->level >= 0 ) /* 当文件刚打开时 fp->level=0, 如果文件按只读 */
/* 或可读/写打开后从流中读数据时 fp->level>=0 */
{
    if(fp->flags & _F_LBUF || fp->curp == &fp->hold)
        /* fp->flags & _F_LBUF 非 0 时表示行缓冲; fp->curp
        ==&fp->hold 表示缓冲区有回退字符 */
        {
            fp->level=0;          /* 表示流现在为空 */
            if( fp->curp == &fp->hold)
                fp->curp=fp->buffer; /* 读取函数一般读流当前指针所指字符,
                现其改指表示流中回退字符被忽略 */
        }
        return 0;                /* 调用成功 */
    }
/* 如果打开的是一个输出流, 下面则将缓冲区中内容写入与流相联的文件 */
count=fp->bsize+1+fp->level;
/* 计算要写到与 fp->fd 相关文件中的字节数 */
fp->level -= count;
fp->curp=fp->buffer;            /* 使流指针指向缓冲区 */
if( (write(fp->fd, fp->curp, count) != count) && /* 如果写失败 */
((fp->flags & _F_TERM) == 0) ) /* 而且又不是一个终端设备文件 */
{
    fp->flags |= _F_ERR;          /* 错误指示器标志置位 */
    return EOF;
}
/* 调用失败 */
return 0;                        /* 调用成功 */
}

main() /* 这是一个仅供观察的程序, 输出文件 QW.C 的内容将不等于输入文件 */
{
    FILE *in, *out;
    char ch;
    in=fopen("qq.c", "r"); /* 假定文本文件 QQ.C 的内容为 "WAB\nCC\x1A" */
    out=fopen("qw.c", "w");
    ch=fgetc(stdin);          /* 标准输入设备 stdin (键盘) 是行缓冲文件 */
    /* 你可以从键盘输入像 abcdef 字符后回车 */
    fputc(ch, out);
    fflush(stdin);
    fflush(out);
    ungetc('H', stdin);       /* 回退一个字符 H 到键盘缓冲区 */
    fflush(stdin);
    ch=fgetc(in);              /* in 为非行缓冲文件 */
    fputc(ch, out);
}

```

```

fflush(in);
fflush(out);
setvbuf(in,buf,-IOLBF,2); /* 将 buf 与 in 相联,文件行缓冲,缓冲区为2 */
ch=fgetc(in); /* 此句执行后将有 2 个字符进入 in->buffer */
fputc(ch,out); fflush(in); /* 通过调试表达式可发现 in->buffer 指向 buf */
fflush(out);
in->level=0; /* 如将此句加上或去掉看看程序执行时结果有什么不同 */
/* 在一般情况下不应在程序中对 FILE 成员赋值! */
ungetc('E',in); /* 观察和上面 ungetc('H',stdin); 的不同处 */
fflush(in);
}

```

—33 int —Cdecl fflushall(void); 对所有打开的读、写流实施 fflush() 操作。当调用成功, 返回一个表示打开输入与输出流总数的整数。这里要注意两点,一是返回值包括缺省打开的 5 个设备文件,二是由 于在 stdio.h 中有

```

#ifdef __STDC__
int —Cdecl fflushall(void);
#endif

```

所以在集成环境下如选用

Options/Compiler/Source/ANSI keywords only On

编译,因这时——STDC——=1,所以将忽略这些函数原型。这表明此函数不是由 ANSI 标准定义的,因此在移植性方面须注意。

```

C>TYPE STD33.C
#include "stdio.h"
main()
{
    FILE *in,*out;
    int k;
    in=fopen("qq.c","r"); /* 假定 QQ.C、QW.C 都存在 */
    out=fopen("qw.c","w");
    printf("in-handle=%d,out-handle=%d\n",in->fd,out->fd);
    k=fflushall();
    printf("被刷新流个数=%d\n",k);
}
/* 程序输出:in-handle=5,out-handle=6
被刷新流个数=7 */

```

—34 int —Cdecl unlink(const char *path);

删掉由 path 所指定的文件,只读文件不能通过它删除。文件名 path 可以包含任意 MS-DOS 驱动器名、目录路径名、文件基本名和扩展名等,但不允许有通配符 * 或 ? 出现。

要删掉只读文件,应先调用 —chmod() 或 chmod() 改变此文件的属性为非只读。

它相当于 DOS 功能调用 41H,入口参数是,AH=0x41,DS:DX=path。

调用成功时返回 0,否则返回 -1,并置 errno=2(ENOENT,无此文件或目录)或 5(EACCES,无此权限)。

适用于 UNIX 系统。

```
C>TYPE STD34.C
#include "dos.h"
#include "sys\stat.h"
main()
{
    int ch,un;
    char *ptr="c:\\tc\\j124.c";      /* 注意此句中反斜杠的写法 */
    ch=chmod(ptr,S-IREAD);           /* 将文件置为只读文件 */
    un=unlink(ptr);
    printf("对只读文件,ch=%d un=%d\n",ch,un);
    ch=chmod(ptr,S-IREAD|S-IWRITE); /* 将文件置为可读写文件 */
    un=unlink(ptr);
    printf("对可读写文件,ch=%d un=%d\n",ch,un);
}
/* 输出:对只读文件,ch=0 un=-1
        对可读写文件,ch=0 un=0 */
```

—35 #define remove(path) unlink(path)

像 unlink() 一样,它也删掉由 path 所指定的文件,但它是一个宏。它定义在 STDIO. H 中。

关于结构 dosSearchInfo 的一点说明

结构 dosSearchInfo 在 DOS. H 中定义,但并未为任何库函数引用,它事实上相当于 标头文件 DIR. H 中定义的结构 fblk (参见《目录函数》一章)。

```
typedef struct {
    char drive;          /* 前 21 个字节用户不能改变 */
    char pattern[13];    /* Microsoft 保留 */
    char reserved[?];
    char attrib;
    short time;
    short date;
    long size;
    char nameZ[13]; /* 用 DOS 功能 4E、4F 搜索目录得到的 asciiz 文件名 */
}dosSearchInfo;
```

C>TYPE STD35.C

```
#include "dos.h" /* 键入 C>RE-DOS ... 就可找到当前目录的第一个文件 */
#include "dir.h" /* 键入 C>RE-DOS L5.C 就在当前目录中可找到文件 L5.C */
/* 如果设置了路径,则可列出该路径内的第一个文件 */
main(int argc,char *argv[])
{
    dosSearchInfo d;
    struct fblk f;
    int k;
    findfirst(argv[1],&d,FA-ARCH); /* 搜索第一个文件 */
    /* Warning: Suspicious pointer conversion */
```



```

/* 在编译时会出现这种警告,可不管它 */
for(k=0;k<13;k++)
{
    printf("%c",d.nameZ[k]);    /* 印出文件名 */
}
printf("\n");
}

```

/* 上述程序和下述程序等价,可见 findfirst() 调用时用到了 DOS 功能 4EH。当然,在此之前它还调用了 DOS 功能调用 1AH 来设置磁盘传输地址为 fblk,这是因为当 DOS 启动一个用户程序时,它自动设置了一个磁盘传输区 DTA,DTA 在程序段前缀 PSP 偏移为 80H ~ FFH 的 128 字节内,并与同时打开的文件记录长度一致。但当你改变了记录长度,则需相应地改变 DAT 的长度,或者说不能再使用 DOS 原先设定的 DTA 了。新的 DTA 由 DOS 功能调用 1AH 设置,它可在内存中任何位置,用 DS:DX 指向 DTA 首地址。

传统的文件管理功能在调用时还使用 DTA 来存放从文件读出或准备写入的文件记录的信息,因此,DTA 的长度不得小于文件记录的长度。

C>TYPE STD36.C

```

#include "dos.h"
#include "dir.h"
main(int argc,char *argv[])
{
    dosSearchInfo d;
    struct fblk f;
    int k;
    findfirst(argv[1],&f,FA-ARCH);
    d.drive=f.ff-reserved[0];
    for(k=1;k<14;k++) d.pattern[k-1]=f.ff-reserved[k];
    d.attrib=f.ff-attr;
    d.time=f.ff-ftime;
    d.date=f.ff-fdate;
    d.size=f.ff-fsize;
    for(k=0;k<13;k++)
        {d.nameZ[k]=f.ff-name[k];
         printf("%c",d.nameZ[k]);
        }
    printf("\n");
}
*/

```

C>TYPE STD37.C

```

#include "dos.h"
#include "dir.h"
#include "ctype.h"
#define PP printf("文件长度=%ld \n 文件名为:",d.size),\
    for(k=0;k<13;k++)
    {if( isgraph(d.nameZ[k]))
        printf("%c",d.nameZ[k]);
    }

```

```

    }
    printf("\n")
main(int argc, char * argv[])
{
dosSearchInfo d;
struct fcb f;
int k;
if(argc<2){printf("xx yy zz\n");exit(1);}
findfirst(argv[1],&d,FA-ARCH);
PP;
findnext(&d);
PP;
}
/* 键入C>RE-DOS *. * 后输出:文件长度=22036
                               文件名为:README\
                               文件长度=4200
                               文件名为:README.COM\

*/

C>TYPE STD38.C
#include "stdio.h"
#include "dos.h"
#include "string.h"
#include "conio.h"
#include "ctype.h"
void display(int,int);
unsigned char buf[53000];          /* 缓冲区 */
main(int argc, char * argv[])      /* 对磁盘搜索指定串的程序 */
{                                  /* 根据刘民的程序改写并注释 */
    /* 在集成环境下可用Options/Arguments 命令输入命令行参数,例如 */
    /* 为搜索 A 盘上的串key 可键入: a: s key */

int Onesec,disk,length,clen,c2;
unsigned register i,j;
long Totalsec,start,c1;
union REGS regs;
unsigned char *p,*sp,comp[80];
clrscr();
if(argc==4)
{
printf("磁盘搜索开始\n");
disk=toupper(argv[1][0])-'A'; /* 先将盘符小写变成大写,disk 得驱动器号 */
regs.h.ah=0x1c;                /* DOS 功能调用 1CH 得到指定驱动器信息 */
regs.h.dl=disk+1;              /* 驱动器号,0 = 缺省,1 = A, 2 = B,... */
intdos(&regs,&regs);           /* 返回的全是 16 进制值 */
Totalsec=regs.x.dx*regs.h.al; /* dx = 簇的总编号,al = 每簇扇区数 */
                               /* 得到待读写的总扇区数 Totalsec */
Onesec=51200/regs.x.cx;        /* cx = 每扇区字节数 */

```

```

/* 得到每一次搜索的扇区数 Onesec */
p=argv[3]; /* 指向待搜索字节 */
if(toupper(argv[2][0])=='S') /* 如果是字符串 */
{
    strcpy(comp,argv[3]); /* 将待搜索串拷入 comp,最多 80 个字符 */
    clen=strlen(comp); /* 求出待搜索串长度 */
}
else
/* 将待搜索串每两位翻译成一个字符,放入 comp[i]。可用下述调试表达式帮助
64,c 与 48,c 求出 ASCII 码 64 和 48 对应的字符
'a'>64 与 '0'>64 看结果是 1 或 0 */
for(i=0;i<strlen(argv[3])/2;clen=++i)
/* 同时处理偶数个输入,奇数时最后一个被忽略 */
comp[i]=(toupper(*p)-48-(*p>64)*7)*16+toupper(*(++p))-48-(*p>64)*7;
for(start=0;Totalsec>0;start+=Onesec,Totalsec-=Onesec)
{
    length=Totalsec>Onesec? Onesec:Totalsec; /* 得到要读扇区个数,16 进制数 */
    absread(disk,length,start,buf); /* start 是起始逻辑扇区号 */
    printf("搜索扇区: %8ld~%8ld\n",start,start+length-1);
    p=buf; /* 指向缓冲区头部,现缓冲区中装的是从磁盘读入的内容 */
    for(i=0;i<length*regs.x.cx;i++,p++) /* length*regs.x.cx 为读入字节数 */
    {
        sp=p; /* sp 为中间指针,p 未动 */
        for(j=0;j<clen;j++) /* clen 为待搜索串字节数 */
        {
            if(*(sp++)!=comp[j])break; /* 每次从缓冲区取出一个字节与搜索串比较 */
            if(j==clen-1) /* 全部相等时 */
            {
                c1=start+i/regs.x.cx;
                c2=i%regs.x.cx;
                printf("\n 已找到,起始扇区: %8x 偏移: %8x\n",c1,c2);
                display(i,length*regs.x.cx);
                printf("按任一键继续进行搜索\n");
                getch();
            }
        }
    }
    printf("\n 搜索结束\n");
}
else
{
    printf("\n a 正确的命令行应为:\n"); /* 响铃提示命令行未输对 */
    printf("\t 本执行程序名 盘符 S 待搜索ASCII 码串\n 或\n");
    printf("\t 本执行程序名 盘符 H 待搜索16 进制串\n");
    printf("ASCII 码串不要用西文双引号括起,16 进制串应有偶数个字符\n");
}

```

```

    printf(" 压任一键结束程序运行,请重新输入正确的命令行! \n");
    getch();
}
}

void display(int sta,int num)    /* 相当于用 debug 的 D 命令显示 */
{
    int i,j,k=sta/256*256;
    char str[4];
    num -=sta;
    for(i=0;i<16;i++)
    {
        for(j=0;j<16;j++,num--)    /* 左边显示 16 进制值 */
        {
            printf("%02X ",buf[i*16+j+k]);
            if(j==7)printf(" ");
        }
        printf(" ");
        for(j=0;j<16;j++)    /* 右边显示对应的 ASCII 码值 */
        {
            if(isprint(buf[i*16+j]))printf("%c",buf[i*16+j+k]);
            else printf(".");
        }
        printf("\n");
    }
}

/* std=io.04z */
—36 int —Cdecl getch(void);
    函数等待从标准输入设备（一般是键盘）读入一个字符,但此字符不会在屏幕上显示。
    （参见《键盘与鼠标》一章。
—37 int —Cdecl getche(void);
    函数等待从标准输入设备（一般是键盘）读入一个字符,但此字符会在屏幕上显示。（参
    见《键盘与鼠标》一章）。只适用于 MS-DOS。
—38 #define getchar() getc(stdin)
    它直接从预定义流 stdin 中取一个字符的宏。适用于 UNIX 系统。
—39 int —Cdecl fgetchar(void);
    直接从预定义流 stdin 中取一个字符。它相当于语句
    { return fgetc(stdin); }
    它不是 ANSI 标准定义的。适用于 UNIX 系统。
—40 #define getc(f) \
    (((--((f)->level))>=0) ? (unsigned char) (++(f)->curp)[-1];-fgetc(f))

    它是一个作用和 fgetc() 完全一样的宏,使用它是为了程序的兼容性。
    f 加括号表示它可以是更复杂的表达式,例如可以是一个结构的成员等等。

```

C>TYPE STD39.C

```

main()
{
char *ptr="ABCD";
ptr++;
printf("%c\n",ptr[-1]);    /* 输出字母 A */
printf("%c\n",ptr[0]);    /*      B */
printf("%c\n",ptr[1]);    /*      C */
}

```

C>TYPE STD40.C

```

#include "stdio.h"
main()
{
FILE *fp;
char ch;
fp=fopen("qq.c","ab");    /* 假定 QQ.C 的内容为 ABCDEFGH */
ch=fgetc(fp);
if(ch == EOF)
{
printf("取得字符的十进制数=%d\n",ch);
if( (fp->flags & _F_ERR) == _F_ERR)
printf("fp->flags 中含有出错标记 _F_ERR\n");
fclose(fp); /* 如不先关闭而对同一个文件又打开,则要多占一个文件句柄 */
fp=fopen("QQ.C","rb");    /* 文件打开时,流缓冲区中并没有装入数据 */
}
ch=_fgetc(fp);printf("它是一个不确定字符,可能十六进制值为:%#x\n",ch);
ch=fgetc(fp);printf("%c, ",ch);
ch=fgetc(fp);printf("%c, ",ch);
ch=_fgetc(fp);printf("%c, ",ch);
ch=_fgetc(fp);printf("%c, ",ch);
ch=getc(fp);printf("%c\n",ch);    /* 可用 CPP.EXE 观察其扩展情况 */
}
/* 程序输出:取得字符的十进制数=-1
             fp->flags 中含有出错标记 _F_ERR
             它是一个不确定字符,可能十六进制值为:0xff8a
             A, B, C, D, E          */

```

C>TYPE STD41.C

```

#include "stdio.h"
main()
{
FILE *fp;
char ch;
int k;
fp=fopen("QQ.C","r+t");    /* 假定原 QQ.C 的内容为 ABCDEFGH */
for(k=0;k<6;k++)printf("%c",fgetc(fp));
printf("\n");
fseek(fp,0L,SEEK-SET);

```

```
fputc('H',fp); /* 用 TYPE 命令可以看到 QQ.C 的内容变为 HBCDEFGH */
rewind(fp); /* 在输出后直接输入时此句不能少 */
ch=getc(fp);printf("%c",ch);
ch=getc(fp);printf("%c\n",ch);
}
/* 程序输出,ABCDEF
HB */
```

—41 int —Cdecl —fgetc(FILE *stream);

它是一个为宏 getc() 服务的函数,功能同 fgetc(),但不能用在文件刚打开时,否则出错。一般不要直接用它! 它的原码为

```
int —fgetc(register FILE *stream)
{
    —stream—>level; /* 仅此一点和 fgetc() 不同 */
    return(fgetc(stream));
}
```

—42 int —Cdecl fgetc(FILE *stream);

返回流 stream 中当前流指针所指的字符。注意,返回值已自动转换为无符号扩展的(即 unsigned char)整型值。在遇到文件结束或出错时,返回 EOF 即 -1。

本函数定义参见程序 FILE2.C。

适用于 UNIX 系统。

C>TYPE STD42.C

```
#include "stdio.h"
main() /* 以二进制方式拷贝文件 */
{ FILE *fp1,*fp2;
    char c;
    fp1=fopen("qq.c","rb");
    fp2=fopen("QW,.C","wb");
    while (!feof(fp1)) /* feof() 根据 flags 判别读文件是否结束 */
    { c=fgetc(fp1); /* fgetc() 在读到文件结束时才返回 EOF */
        if (c != EOF)fputc(c,fp2); /* 此句不能去掉 if (c != EOF), 否则
            QW.C 尾将比 qq.c 多一个字节 0xFF */
    }
    fclose(fp1);
    fclose(fp2);
}
```

—43 int —Cdecl getw(FILE *stream);

它返回流 stream 中下一个整数(两个字节)。在遇到文件结束标志(—F—EOF)或出错标志(—F—ERR)时,它返回 EOF。注意,因为 EOF 是它返回的合法值,因此应该用 feof() 或 ferror() 检测是文件结束或出错。在程序 STD62.C 中有该函数的定义。

注意:使用它时在文件中应无特殊的对齐字符存在。适用于 UNIX 系统。

—44 void * —Cdecl cgets(char *str);

从控制台输入字符串和它的字符个数(字符串长度)到 str 指定位置中。

调用前用户应用 `str[0]` 指定要读入的最多字符数。`cgets()` 读字符直到遇到 CR/LF 或已读了最大字符数为止。如果它读到 CR/LF 则在存储前用 `'\0'` 代替之。

如调用成功, `str[1]` 为读入实际字符数(回车换行符不包括在内),它一般小于或等于 `str[0]-1`。实际读入串的起始地址是 `&str[2]`。由于 Turbo C 不会检查数组下标,因此用数组定义 `str` 时应使下标等于要读的最多字符数再加上 2,方可避免意外。它适用于 UNIX 系统。

```
C>TYPE STD43.C
#include <stdio.h>
#include <conio.h>
main(){
char buffer[8];    /* 定义的数组长度 = buffer[0]+1          */
char *ptr;         /* 定义指针 ptr          */
buffer[0]=6;       /* 指定要读的字符串最大长度(包括CR/LF)到 0 元素中 */
ptr=cgets(buffer); /* 当输入字符超过它时系统响铃警告,并不接受超过字符 */
                  /* 故真正能接收的字符为 5 个,加 CR/LF 共 6 个          */
printf("\n cgets got %d chars, \"%s\"\n",buffer[1],ptr);
printf("buffer[0]=%p buffer[1]=%p\n",&buffer[0],&buffer[1]);
printf("ptr=%p,buffer[2]=%p\n",ptr,&buffer[2]);
    /* 从键盘输入一个字符 A 后,屏幕输出:
        cgets got 1 chars,"A"          buffer[1] 中为返回的实际接收
                                         的字符个数,它不包括 CR/LF 符 */
    /* buffer[0]=FFDE buffer[1]=FFDF
        ptr=FFE0 buffer[2]=FFE0      说明 ptr 指向 buffer[2] */
    buffer[0]=6;
    ptr=cgets(buffer);
}
```

键入字符 A 回车后的调试表达式与值:

```
ptr,p          DS:FFE0
ptr            "A"          /* 可见 ptr 指向 buffer[2] */
buffer,MH      06 01 41 00 FA 01 85 6D /* 定义几个,显示几个 */
buffer         "\x6\x1A"      /* 调试式不能列出数组地址 */
```

未输入任何字符而直接按回车键后的调试表达式与值:

```
ptr,p          DS:FFE0
ptr            ""
buffer,MH      06 00 00 00 FA 01 84 6D /* CR/LF 用空字符来存储 */
buffer         "\x6"          /* 实际存储 0 个字符未列出 */
```

—45 char *—Cdecl gets (char *s);

从标准预定义流 `stdin` 中读入一个字符串到 `s` 中,并用空字符(`'\0'`)代替输入中的换行符(`'\n'`)。当从键盘输入时一般按回车键便认为输入结束。

由于可用 `freopen()` 使 `stdin` 重定向,故 `stdin` 必要时也可和磁盘文件相联,虽然大多数时候不需要这样做。

调用成功时串 `s` 中装入输入的字符,否则在出错或文件结束时返回 `NULL`。

适用于 UNIX 系统。

C>TYPE STD44.C

```
#include "stdio.h"
#include "string.h"
char * gets(char * s) /* 等效函数 */
{
    register int ch;
    register char * p=s;
    while( (ch=getc(stdin)) != EOF && ch != '\n') /* 遇换行符'\n'终止输入 */
        *p++=ch; /* 如按 Ctrl-Z (即0x1a) 后回车A 便相当于键入EOF */
    if( ch == EOF && p == s)return NULL;
    *p=0; /* 在串尾写上空字符'\0' */
    return ( (ferror(stdin)) ? NULL:s);
} /* 可用调试表达式 —streams[0].flags & 0x0010 观察,非 0 知有错误 */

main()
{
    char buf[80];
    char * ptr="指针";
    strcpy(buf,"将本串拷入数组中,结果指针指向非空串");
    printf("ptr=%p, 请用键盘输入若干字母后回车\n",ptr);
    ptr=gets(buf);
    printf("ptr=%p, 你刚刚输的字母是:%s\n",ptr,buf);
    fputc('H',stdin);
    ptr=gets(buf);
    if(ptr == NULL)printf("ptr=NULL\n");
}

/* 假定从键盘输入字母 as,程序输出:ptr=0194, 请用键盘输入若干字母后回车
ptr=FF92, 你刚刚输的字母是:as
ptr=NULL */
```

—46 char * —Cdecl fgetc(char * s,int n,FILE * stream);

从流 stream 中读字符到字符串 s 中,具体过程是,当读了 n-1 个字符或遇到换行符时,读便停止。它保留换行字符,并在串的最后最后一个字符后面自动加上空字符('\0')。

调用成功时串 s 中装入读入的字符,否则在出错或文件结束时返回 NULL。

适用于 UNIX 系统。

C>TYPE STD45.C

```
#include "stdio.h"
#include "string.h"
main()
{
    FILE * fp;
    char buf[80]="串";
    char * ptr="指针";
    fp=fopen("qq.c","r"); /* 假定 QQ.C 的内容是:ABC 0x0d 0x0a
                                abcdefgh 0x0d 0x0a 0x1a
                                文件长度是 16 个字节 */

    ptr=fgets(buf,10,fp);
```



```

printf("ptr=%p, 读入的字符是:%s 串长度是:%d\n",ptr,buf,strlen(buf));
ptr=fgets(buf,6,fp);          /* 自动换行输出表示 buf 中有换行符\n */
printf("接着读入的字符是:%s 串长度是:%d\n",buf,strlen(buf));
}
/* 程序输出:ptr=FF92, 读入的字符是,ABC
           串长度是:4
           接着读入的字符是,abcde 串长度是:5      */

```

—47 int —Cdecl —read (int handle,void *buf,unsigned len);

对与文件句柄 handle 相联的文件中,从文件指针当前所指位置开始读出 len 个字节到由 buf 所指的缓冲区中。这里, len 可为 1 ~ 65534。读入文件可以是磁盘文件,也可以是设备文件。如果从控制台(CON,例如键盘)读入,遇到文件中的第一个回车符(CR)便停止读。在读出 len 个字节后,文件指针自动增量 len,即指针后移。若调用本函数之前文件指针已指向文尾,则返回 0,否则返回实际读入缓冲区的字节数(如果文件以文本方式打开,所读字节数不会包括回车符和 Ctrl-Z 即 1AH 码)。当出错时,返回 -1,并置 errno 为 5 (EACCES) 或 6 (EBADF)。

显然,为容纳每一次读入的字符数,缓冲区 buf 一般定义为足够大的数组。

它实际调用 MS-DOS 的功能 3FH,入口参数是,AH=3FH,BX=handle,CX=len,DS:DX=buf。

它只适用于 MS-DOS。

```

C>TYPE STD46.C
#include "io.h"
#include "fcntl.h"
main()                                /* 假定 qq.c 内容为: AB */
{ static char buf[2];                /* C */
  int handle,len,length;
  if( (handle==open("qq.c",O_RDWR|O_TEXT))<0)exit(1);
  while(! eof(handle))
  {
    length=tell(handle);
    len=read(handle,buf,1);
    printf("%d %d % #x=%c\n",length,len,buf[0],buf[0]);
    /* 或 printf("%d %d %s\n",length,len,buf); */
    buf[0]='\0';
  }
  length=tell(handle);
  len=read(handle,buf,1);
  printf("%d %d % #x=%c\n",length,len,buf[0],buf[0]);
}
/* 程序输出:0 1 0x41=A
           1 1 0x42=B
           2 1 0xd=
           3 1 0xa=
           4 1 0x43=C
           5 1 0xd=

```

6 1 0xa=

7 0 0= */

—48 int —Cdecl read (int handle,void * buf,unsigned len);

它是基于 —read() 基础上的函数,参数含义及功能和 —read() 大致相同,唯一不同的是,当 handle 是 open() 用 O—TEXT 打开而得到的文本文件的文件句柄时,本函数要忽略遇到的回车符 (0xd, 即转义符 '\r'), 即不把它读入 buf 中。另外,在这种情况下,当它读到 Ctrl—Z 字符即 1AH 码时,即认为已读到文件结尾。此种特性有时是不能忽略的,例如,当用本函数读写一些二进制文件 (像 DBASE 的 *.DBF 文件) 时就应注意,否则会出错的。

对本函数当用 —open() 打开文件或用 O—BINARY 方式打开文件时这种特性将不存在。

适用于 UNIX 系统。

C>TYPE STD47.C

```
#include "io.h"
```

```
#include "fcntl.h"
```

```
main()                        /* 假定 qq.c 内容为:0x61 0xd 0xa 0x62 0x1a */
```

```
{ static char buf[2];
```

```
  int handle,len,length;
```

```
  if( (handle=—open("qq.c",O—RDWR|O—TEXT))<0)exit(1);
```

```
  while(! eof(handle))
```

```
  {
```

```
    length=tell(handle);
```

```
    len=read(handle,buf,1);
```

```
    printf(" %d %d % #x=%c\n",length,len,buf[0],buf[0]);
```

```
    buf[0]='\0';
```

```
  }
```

```
  length=tell(handle);
```

```
  len=—read(handle,buf,1);
```

```
  printf(" %d %d % #x=%c\n",length,len,buf[0],buf[0]);
```

```
}
```

```
/* 程序输出:0 1 0x61=a
```

```
  1 1 0xd=                回车
```

```
  2 1 0xa=                换行
```

```
  3 1 0x62=b
```

```
  * 假定 q1.c 文件也存在 */
```

```
如将if( (handle=—open("qq.c",O—RDWR|O—TEXT))<0)exit(1);改成
```

```
  if( (handle=open("qq.c",O—RDWR|O—TEXT))<0)exit(1);
```

```
则输出: 0 1 0x61=a
```

```
  1 1 0xa=
```

```
  3 1 0x62=b
```

```
  4 0 0=
```

```
回车换行符被忽略。
```

```
  */
```

—49 size—t —Cdecl fread(void * ptr,size—t size,size—t n,FILE * stream);

从一个指定输入流 stream 中读取 n 项数据（每项数据的长度为 size 字节）到 ptr 所指的存储块中。参数 ptr 是指向任意对象的指针。

在调用成功时返回实际读入的项数，在遇到文件结束或出错时，返回一个短（short）计数值（有可能为 0）。

适用于所有 UNIX 系统。

C>TYPE STD48.C

```
#include "stdio.h"
#include "stdlib.h"
#include "io.h"
#ifdef SS=1 /* 在调试时你可选 Options/Compiler/Defines 输入宏名 ss=1 或 SS = 或 SS 时则下列
           程序被调试，而当空时则不被调试。因此 你可以看看这些函数的正确性。注意，你不要
           用 #ifdef SS 这样的语句！那样是不起作用的 */
static unsigned pascal near _fgetn(register void * ptr, size_t n,
                                   FILE * stream)
{
    register int Byte, Temp;
    while(n) /* 当 n 大于 0 时 */
    {
        n++; /* n 先增 1 后不妨称新 n */
        Temp = min(n, stream->bsize); /* 取新 n 与流缓冲区的尺寸两者中最小者 */
        if((stream->flags & _F_BIN) && stream->bsize && (n > stream->bsize)
            && (stream->level == 0))
            /* 对流 stream 要同时满足的条件：
              状态标志中包含 _F_BIN（即二进制流）；
              缓冲区尺寸大于 0 而小于新 n；
              缓冲区为空 */
        {
            n--; /* 恢复原 n */
            Temp = 0; /* 置初值，以便累加 */
            while(n >= stream->bsize)
            {
                Temp += stream->bsize; /* 取 n 中缓冲区尺寸的整数倍 */
                n -= stream->bsize; /* 最后 n 为小于缓冲区尺寸 */
            }
            Byte = _read(stream->fd, ptr, Temp); /* 表示读 Temp 个字节到 ptr 所指的缓冲区中，如读成功
            返回 Byte 为读入字节数；否则为 0 或 -1 */
            (char *)ptr += Byte; /* 获新指针值。因 ptr 定义类型为 void，因此有 (char *)
            */
            if(Byte != Temp) /* 表示读失败时的处理 */
            {
                n += (Temp - Byte); /* 由于 Byte=0 或 -1 故 n>0 */
                break;
            }
        }
    }
    else
    {
```

```

while(--n && --Temp && (Byte =getc(stream)) != EOF)
    * ((char *)ptr)++ = Byte; /* 在指针所在位置置字符后指针移动 */
if(Byte == EOF)break; /* 从流中读字符失败后中止 while(n) 循环,此时必有 n>0 (因
先判断 n 后执行 Byte=getc(stream)) */
}
}
return n; /* 如调用成功返回 0,否则返回非 0 值 */
}
size_t fread(void * ptr,size_t size,size_t n,FILE * stream)
{
register size_t n1;
unsigned long temp;
if(!size)return 0; /* 如果长度为 0 则返回 0 */
if( (temp=(unsigned long)size * (unsigned long)n) < 0x10000L)
    /* temp 为读入字节总数,0x10000L=65536 */
return ( (unsigned)temp - fgetn(ptr,(unsigned)temp,stream))/size;
/* 返回实际读入项数,不是读入的总字节数 注意,当 fgetn() 返
回非 0 值时将有一部分字节不被读入 */
else
{
    n1=n+1;
    while (--n1 && fgetn(ptr,size,stream) == 0)
        (char *)ptr += size; /* 当用 huge 存储模式编译时此句可改成
ptr=(char huge *)ptr)+size; */
    return (n-n1);
}
}
#endif
main()
{
FILE * stream;
char s[2];
char ptr[10];
int item;
stream=fopen("QQ.C","r+"); /* 假定 QQ.C 的内容为:
AB 0x0d 0x0a CDE 0x0d 0x0a 0x1a */
setvbuf(stream,s,-IOFBF,2);
item=fread(ptr,sizeof(char),8,stream);
ptr[item*sizeof(char)]=0; /* 因 fread() 对 ptr 不会自动在尾部加'\0' */
printf("item=%d, ptr=%s\n",item,ptr);
stream=fopen("QQ.C","rb+"); /* 按二进制打开 */
setvbuf(stream,s,-IOFBF,2);
item=fread(ptr,sizeof(char),8,stream);
ptr[item*sizeof(char)]=0;
printf("item=%d, ptr=%s\n",item,ptr);
}

```

```
/* 程序输出:item=7, ptr=AB
```

```
CDE
```

```
item=8, ptr=AB
```

```
CDE
```

如将 fread() 及 ptr[]=0 语句中的 sizeof(char) 全改成 sizeof(char)*2, 则输出

```
item=3, ptr=AB
```

```
CDE
```

```
item=5, ptr=AB
```

```
CDE
```

注意:二进制输出时在串后将增加一个 0x0d,在 DOS 下用 type 列出时看不出 */

```
—50 int —Cdecl putchar(int c);
```

将单个字符 c 送至屏幕窗口内,字符使用当前颜色和显示属性,返回字符 c 的 ASCII 码值。相当于 DOS 中断 INT 21H 的功能 2H,入口参数是 AH=2H,DL=c。它将检查 Ctrl-C 和 Ctrl-Break 字符,若发现则执行 INT 23H 中断。对 DOS 2.0 以上版本标准设备可以重定向。

```
—51 #define putchar(c) putc((c),stdout)
```

宏,即将字符输出到预定义流 stdout 中。如调用成功,返回 c,否则返回 EOF。如输出未重定向,则它将单个字符输出到屏幕,但不加 '\n'。它相当于

```
fprintf("%c",c);
```

语句。编译后生成的 .EXE 文件比用 printf() 的要小得多,执行速度也快。所以,除非必要,能用 putc()、putchar() 的地方不要用 printf() 代替。

适用于 UNIX 系统。

```
—52 int —Cdecl fputc(int c);
```

它相当于 fputc(c,stdout);。和 putchar() 不同的是,它是一个函数,而不是一个宏。它不是 ANSI 标准规定的函数。

如调用成功返回 c,否则返回 EOF。适用于 UNIX 系统。

```
C>TYPE STD49.C
```

```
#include "stdio.h"
```

```
main()
```

```
{
```

```
FILE *fp;
```

```
fp=fopen("QQ.C","r+",stdout); /* 假设 QQ.C 的原内容为 ABC */
```

```
if(fp == NULL)
```

```
{printf("用指定文件名QQ.C 代替预定义流stdout 失败\n");exit(1);}
```

```
putc('S');
```

```
fputc('T');
```

```
}
```

```
/* 程序执行后,屏幕并无任何输出,而有
```

```
C>TYPE QQ.C
```

```
STC
```

```
即原AB 两个字母被置换成ST */
```

```
—53 #define putc(c,f)((++((f)->level)<0)? \
```

```
(unsigned char)((++(f)->curp)[-1]=(c));--fputc((c),
f))
```

它是一个宏,功能和 fputc() 一样,将字符 c 送到输出流 f 中。

调用成功返回 c,否则返回 EOF。适用于 UNIX 系统。

C>TYPE STD50.C

```
#include "stdio.h"
```

```
#include <string.h>
```

```
#include "alloc.h"
```

```
main()
```

```
{
```

```
struct pp{
```

```
FILE *fp;
```

```
char c[80];
```

```
char ch;
```

```
}pt;
```

```
struct pp *ptr=malloc(sizeof(pt));
```

```
strcpy(ptr->c,"QQ.C");
```

```
ptr->ch='R';
```

```
ptr->fp=fopen(ptr->c,"wb+");
```

```
putc(ptr->ch,ptr->fp);
```

```
((++(ptr->fp->level)<0)?(unsigned char)((++ptr->fp->curp)[-1]=(ptr->ch));
--fputc((ptr->ch),ptr->fp));
```

```
}
```

/* 程序执行后有

C>TYPE QQ.C

```
RR
```

```
*/
```

```
--54 int --Cdecl --fputc(char c,FILE *stream);
```

它相当于执行这样两个过程:

```
--stream->level; /* 和 fputc() 不同之处 */
```

```
return (fputc(c,stream));
```

因此,当缓冲区为空即 stream->level=0(如文件刚打开、调用了 rewind() 或 lseek()) 时使用它是很危险的,这时它可能先将缓冲区中的已有内容(不管它是什么)装入文件,然后再将字符 c 放入缓冲区。

该函数主要是为宏 putc() 服务的,一般不要直接用它。

调用成功返回 c,否则返回 EOF。

C>TYPE STD51.C

```
#include "stdio.h"
```

```
#include "ctype.h"
```

```
PUTC(char ch,FILE *fp)
```

```
{
```

```
char c;
```

```
c=--fputc(ch,fp); /* 将产生一个 514 个字节长度的文件 QQ.C,而且只有字母 X 被写入(文件最后
```

一个字符)。如将此句 `fputc()` 改成函数 `fputc()` 则产生文件长度为 3, 三个字符均被写入文件。这说明前者是错误的, 后者是正确的。 */

```
if(c == ch && isprint(c))printf("%c", c); /* 你将发现输出还是正确的 */
else printf("%x", c);
}
main()
{
FILE *fp;
char ch;
fp=fopen("qq.c", "w+");
PUTC('X', fp);
PUTC('Y', fp);
PUTC('\0', fp);
}
```

—55 int —Cdecl fputc(int c, FILE *stream);

送一个字符到流 stream 中。调用成功返回输出字符, 否则返回 EOF。

C>TYPE STD52.C

```
#include "stdio.h"
#include "ctype.h"
#define DEF(C) err=ok=fputc(C,fp)
#define PEO printf("%d", fp->fd); \
    if(isprint(err=ok)) printf("%c", err=ok); \
    else printf("%x", err=ok)
main() /* 利用本程序可观察行缓冲的结果 */
{
FILE *fp;
int err=ok;
char s[BUFSIZ];
fp=fopen("c:\\tc\\qq.c", "r"); /* 只读 */
DEF('G'); /* 肯定要失败 */
PEO;
fp=fopen("c:\\tc\\qq.c", "r+"); /* 可读/写 */
DEF('H');
PEO;
fp=fopen("c:\\tc\\qq.c", "w"); /* 只写 */
DEF('L');
PEO;
fclose(fp);
fp=fopen("c:\\tc\\qq.c", "w+"); /* 只写/读 */
setvbuf(fp, s, _IOLBF, 6); /* 设置行缓冲 */
DEF('M');
DEF('N');
PEO;
DEF('\n' /* 换行符0xa */);
DEF('\r' /* 回车符0xd */);
DEF('P');
```

```
PEO;
```

```
}
```

```
/* 程序输出: 5, 0xffff; 6, H: 7, L: 7, N: 7, P;
```

下面列出断点跟踪方法:

将光标移到 DEF('\r' /* 回车符0xd */); 行上,按 Ctrl-F8 键,该行出现红亮条,表明该行已设置为断点行。按 Ctrl-F9 后又按 Ctrl-F2 终止断点调试,然后按 Alt-F 后再按 O 键后暂时退出集成环境。在 DOS 状态下用 debug.exe 分两种情况来观察文件 QQ.C 的内容:

```
用fp=fopen("c:\\tc\\qq.c","w+");    用fp=fopen("c:\\tc\\qq.c","wb+");
```

```
C>DEBUG QQ.C
```

```
-D
```

```
XXXX:0100 4D 4E 0D 0A
```

```
    即 M  N
```

```
C>DEBUG QQ.C
```

```
-D
```

```
XXXX:0100 4D 4E 0A 00
```

```
    M  N
```

将光标移到 DEF('\r' /* 回车符0xd */);行上,按 Ctrl-F8 键,该行出现红亮条消失,表明该行已不再为断点行。将 DEF('P');行设置为新断点行,再按上述过程 进行:

```
C>DEBUG QQ.C
```

```
-D
```

```
XXXX:0100 4D 4E 0D 0A 0D
```

```
    即 M  N
```

```
C>DEBUG QQ.C
```

```
-D
```

```
XXXX:0100 4D 4E 0A 0D
```

```
    M  N
```

最后将整个程序执行完,有

```
C>DEBUG QQ.C
```

```
-D
```

```
XXXX:0100 4D 4E 0D 0A 0D 50
```

```
    即 M  N      P
```

```
C>DEBUG QQ.C
```

```
-D
```

```
XXXX:0100 4D 4E 0A 0D 50
```

```
    M  N      P
```

```
*/
```

```
—56 int —Cdecl putw(int w,FILE * stream);
```

把一个字 w(2 个字节) 送到流 stream 中。它不是 ASCII 标准定义的。

调用成功返回整数,否则返回 EOF。适用于 UNIX 系统。

```
C>TYPE STD53.C
```

```
#include "stdio.h"
```

```
#define PW    /* 取消此句便可在调试时不进入 putw() 函数 */
```

```
#ifdef PW
```

```
int putw(int w,FILE * stream)    /* 等效函数 */
```

```
{
```

```
if(putc( *((unsigned char *) &(w)),stream) != EOF) /* 向流输入第一个字符 */
```

```
if(putc( *((unsigned char *) &(w) + 1), stream) != EOF) /* 输第二个字符 */
```

```
    return (w);                                /* 调用成功 */
```

```
    return (EOF);                             /* 调用失败 */
```

```
}
```

```
#endif
```

```
main()
```

```
/* 本程序用于向流中输入字的情况 */
```

```
{
```

```
FILE * stream;
```

```
struct {
```



```

char ch;
int w;
}pw;
int t1,t2,t3,t4,t5;
pw.w='S';
stream=fopen("QQ.C","w+b"); /* 建立二进制文件 QQ.C */
t1=putw(pw.w,stream);
pw.w='TU';
t2=putw(pw.w,stream);
t3=putw('\r\n',stream);
t4=putw('U'+1,stream);
t5=putw('W\xla',stream);
printf("%d;%#06x,%d;%#x,%d;%#x,%d;%#06x,%d;%#x\n",
        t1,t1,t2,t2,t3,t3,t4,t4,t5,t5);
}
/* 程序输出:83;0x0053,21844;0x5554,2573;0xa0d,86;0x0056,6743;0x1a57

```

C>DEBUG QQ.C

-D

XXXX;0100 53 00 54 55 0D 0A 56 00 - 57 1A

即 S T U V W

C>TYPE QQ.C

S TU

V W

可以用以下调试表达式观察流中情况:

&stream->curp[-10],p

stream->curp[-10],10mx

stream->curp[-10],10mc 从它中可见字母S TU V W 等等

stream[0],p

stream[0],x

stream[0]

从中可以看出,输出到流中字符存储在指针 stream->buffer 到 stream->curp 之间所指的位置上。第一个字符存在缓冲区头部,以后是第二个字符,等等 */

-57 int -Cdecl cputs (const char *str);

将指定字符串 str 输出到控制台 (一般也指屏幕),返回最后一个送到屏幕上的字符。字符串输出后光标不会自动换行。

C>TYPE STD54.C

#include "conio.h"

main(){

char a[]="abcd";

char *s=a;

cputs(a);

cputs(s);

getch();

}

程序输出:

abcdabcd

可见 cputs 未加换行符。F7 热键追踪结束时显示调试表达式

cputs CS:0E3C

—58 int —Cdecl puts(const char *s);

它也将以空字符 ('\0') 为终结的字符串 s 输出到预定义流 stdout 中, 但和 fputs() 不同的是, 它要在串尾加上换行符 ('\n')。适用于 UNIX 系统。

```
C>TYPE STD55.C
#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"
#define PRETU if(isprint(retu))printf("%c", retu); \
               else printf("%#06x", retu)

void pr(char r)
{
    if(isprint(r))printf("%c", r);
    else printf("%#06x", r);
}

main()
{
    FILE *fp;
    char *s1="ACBD", s2[]="efg", *s3="##", s4[]={"* * *"};
    char retu, r1, r2;
    fp=fopen("qq.c", "wb+");
    retu=fputs(s1, fp);PRETU;
    retu=fputs(s2, fp);PRETU;
    retu=fputs("==", fp);PRETU;
    /* 如加 exit(0); 程序执行后生成 QQ.C 的内容为:ACBDefg== */
    /* 如加 abort(); 则程序执行后生成的 QQ.C 中什么也没有, 长度为 0 */
    fclose(fp); /* 此句必须有, 否则流中数据可能会丢失, 即不会输入 QQ.C 中 */
    printf("\n");
    fp=fopen("qq.c", "ab", stdout);
    r1=puts(s3);
    r2=puts(s4);
    fclose(fp);
    freopen("con", "r+", stdout);
    pr(r1);
    pr(r2);
    printf("\n");
}

/* 程序输出: D, g, =,
               0x000a, 0x000a,

C>DEBUG QQ.C
-D
XXXX,0100 41 43 42 44 65 66 67 3D - 3D 23 23 0A 2A 2A 2A 0A
```

即 A C B D e f g = = # # * * * * /

—59 int —Cdecl fputs(const char *s, FILE *stream);

它将以空字符 ('\0') 为终结的字符串 s 输出到指定流 stream 中, 但串尾的空字符 并不进入流, 它也不会 在串尾自动加上换行符 ('\n' 或 0xa)。

如调用成功返回最后写字符, 否则返回 EOF。适用于 UNIX 系统。

—60 int —Cdecl —write(int handle, void *buf, unsigned len);

将缓冲区 buf 中的 len 个字节的内容写到具有文件句柄 handle 的文件中去。文件 可以是磁盘文件或设备文件。它是 —read() 函数的反函数。

对于磁盘文件, 写总是从文件指针当前所指的位置开始的 (用户可用 lseek 移动读写指针); 对于设备文件, 字节将直接传送到设备中。

如果实际写入的字节数少于 len, 就表示可能有错误, 例如磁盘空间满, 已经不能再 写入内容了。当 len=0 时, 则无数据写入。

对于用 O—APPEND 存取标志打开的文件, 由于文件打开后文件指针已指向文件末尾, 因此写入的数据将从已打开文件尾部开始添加。

对于文本文件, 当函数发现一个换行符 (LF, 即转义符 '\n', 0xA) 时不会将它变 成回车换行符 (即变成 '\r' 与 '\n' 或 0xD 与 0xA) 输出。函数输出时总是按二进制 方式进行的, 即缓冲区中是什么就输出什么。

函数返回所写字节数。出错时返回 -1, 并置 errno=5(EACCES) 或 6(EBADF)。

它实际 DOS 功能调用 40H, 入口参数是, AH=40H, BX=handle, CX=len, DS:DX=buf。

它只适用于 MS-DOS。

C>TYPE STD56.C

```
#include "stdio.h"
#include "io.h"
#include "fcntl.h"
main()
{
    static char s[]="AB\nC\x1A";          /* 显式给出文件结束标志 1AH */
    /* 如此句写成 static char s[]="AB\nC"; 则文尾不产生 1AH 码 */
    int fp, flag;
    if( (fp=open("qq.c", O—WRONLY|O—APPEND))<0)exit(1);
    flag=—write(fp, s, 10);
    printf("flag=%d\n", flag);
    close(fp);
}
```

/* 在 DOS 提示符下键入

C>COPY CON QQ.C

W Z

则在当前目录里生成文件 QQ.C, 它只有一个字母 W, 也无结尾码 1AH。接着 执行本程序, 有

flag=10

再用

C>DEBUG QQ.C

```

-D
2CF0:0100 57 41 42 0A 43 1A .....
    即 W A B C
*/

```

```

-61 int _Cdecl write(int handle,void *buf,unsigned len);

```

就像 `_read()` 与 `read()` 一样,本函数和 `_write()` 的区别在于,对文本文件,当 `write` 发现一个换行符 (LF) 时,它会将此换行符变成回车换行符对输出,即在换行符前自动增加一个回车符。注意,与 `handle` 相联的文件必须是用 `creat()` 创建或 `open()` 打开的文本文件。如将 `_write()` 中说明的例程里的 `_write` 语句改成 `write`,则有

```

C>DEBUG QQ.C
-D
2CF0:0100 57 41 42 0D 0A 43 1A .....
    即 W A B C

```

当用它写一个文本文件时,写到文件中的字节数不会多于所要求的字节数。它适用于 UNIX 系统。

```

-62 size_t _Cdecl fwrite(const void *ptr,size_t size,
                        size_t,n,FILE *stream);

```

有 n 个数据项,每个数据项均有 `size` 个字节,调用本函数可将从指针 `ptr` 所指位置算起的 $n * size$ 个字节的数据写到流 `stream` 中。在函数原型中,参数 `ptr` 被定义为指向任意对象的指针,因此在程序中程序员自己可以选定具体的类型。值得注意的是,从 `ptr` 所指位置算起应有的 $n * size$ 个字节数应不大于 `strlen(ptr)` 即字符串长度,而串的长度应小于 65536; `size` 再好为 `sizeof(*ptr)`,以便项所占字节数和串中数据类型协调。流自然应该是可写或可写/读的。如果要在文件最后明确写上结束符 `0x1a`,则应先定义类似

```

int t3[]={0x1a};

```

那样的语句,然后在关闭流前使用语句

```

fwrite(t3,1,1,stream);

```

如调用成功返回写入流的实际项数。在遇到文件结束或出错时返回一小于 n 的值 (也可能是数字 0)。

它适用于所有 UNIX 系统。

```

C>TYPE STD57.C
#include "stdio.h"
main(int argc,char *argv[])
{
FILE *fp;
char *ptr="AB",*s="cdefGHIJKLMNOP";
int t1[]={1,22,0x33,4,5,6,7,8,9};
int t2[]={9,88,077,4,5,6};
int item1,item2,item3,item4;
if(argc != 2)
{printf("命令行参数应有且只能有2个参数!\n");exit(1);}
/* printf("%s\n",argv[1]);需要观察时增加此行 */

```

```

if( strcmp(argv[1],"w+") == 0 || strcmp(argv[1],"wb+") == 0);
/* 注意上句中末尾的分号不能少,尽管看起来它是一个并没做任何事的语句 */
/* 也可用stricmp()或strcmipi()取代strcmp(),就可不区分串中字符的大小写 */
else {printf("第2 参数应为w+或wb+ \n");exit(1);}
fp=fopen("qq.c",argv[1]);
item1=fwrite(ptr,1,5,fp);          /* 每项为一个字节 */
item2=fwrite(s,2,3,fp);            /* 每项为两个字节 */
item3=fwrite(t1,2,4,fp);           /* 每项为两个字节 */
item4=fwrite(t2,1,5,fp);           /* 每项为一个字节 */
printf("%d, %d, %d, %d\n",item1,item2,item3,item4);
}

```

/* 在集成环境里,选 Options/Arguments 项并输入 wb+ (或 "wb+") ,或 w+ ("w+") 命令行参数
后再按 Ctrl-F9 执行程序 (假定本程序经编译连接后 生成的执行文件名为 Q.EXE,也可以在命令
行上键入

C>Q wb+

或

C>Q w+

则起同样作用), 程序输出: 5, 3, 4, 5。用 DEBUG.EXE 观察,有

c>debug qq.c

-d

xxxx:0100 41 42 00 63 64 63 64 65 - 66 47 48 01 00 16 00 33

即 A B \0 c d c d e f G H 数1 数 22

xxxx:0110 00 04 00 09 00 58 00 3F -

数0x33 数4 数9 数88 数077 的低位

从中可以看出,由于项数和指定串 ptr 的长度不协调,导致输出到流中内容混乱,这是要注意的。

*/

-63 int -Cdecl ungetch(int ch);

将字符 ch 回退到键盘缓冲区,使其成为下一个所读字符。当下一次用 getch() 或其它控制台输入函数调用时,将返回 ch。如果调用成功返回 ch,否则返回 EOF (例如,调用一次本函数后,如未将回退字符 ch 读出,又继续多次调用本函数则会失败)。它只适用于 MS-DOS。

-64 int -Cdecl ungetc(int c, FILE * stream);

把一个字符回退到输入流缓冲区中。该字符在下一次用 fgetc()、getc() 或 fread() 时被读出。在所有情况下,都只能回退一个字符。在未将回退字符读出的情况下,如连续多次调用它可能会产生不可预料的结果。fseek() 清除所有被退回字符。

调用成功时总是返回回退字符,否则返回 EOF。适用于 UNIX 系统。

#define ungetc(c,f) ungetc((c),f)

该宏功能同 ungetc(),它是一个传统使用的函数,定义它是为了兼容。

C>TYPE STD58.C

#include "stdio.h"

#undef ungetc /* 不使用 stdin.h 中定义的传统宏 ungetc(c,f) */

int ungetc(int c, FILE * stream)

```

{
if(c != EOF)
    if( ++(stream->level) > 1)          /* 缓冲区非空时不接收回退字符 */
        return (unsigned char) (* --(stream->curp) = c);
    /* 流当前指针先减 1, 然后在流当前指针所指位置置回退字符, 因此流状态标志 stream->level 要增 1 了, 因为流缓冲区中增加了一个回退字符. 注意变量 stream->hold 中并未置回退字符. 当从流中取字符时因为总是从流当前指针所指字符开始读取的, 因而回退字符被读取 */
else if (stream->level == 1) /* 只有当缓冲区为空时才接收回退字符 */
    {
        /* 即原 stream->level=0 时才行 */
        stream->curp = &stream->hold; /* 流当前指针指向 hold 变量 */
        return(unsigned char) (stream->hold = c); /* hold 被赋予字符 */
    }
    else stream->level--; /* 缓冲区虽空, 但不是输入方式 */
return (EOF);
}

#include "ctype.h"
#define PC c=getc(stream); if(isprint(c)) printf("%c", c); \
        else printf("%#x", c)

main()          /* 可用 stream[0], x 调试表达式在用 F7 单步调试时观察 */
{
FILE *stream;
char c;
long pos;
stream=fopen("qq.c", "rb"); /* 假定 QQ.C 的内容为 ABCabc */
PC;
ungetc(0x44, stream); /* 回退字母 D */
ungetc('W', stream); /* 一般不要一次连续回退多个字符, 因为当流当前指针值小于 stream->buffer 指针值时, 有时是很危险的 */

ungetc('s', stream);
PC;
PC;
fflush(stream);
PC;
PC;
ungetc('t', stream);
fflush(stream);
PC;
PC;
fgetpos(stream, &pos); /* 保留当前文件指针位置 */
stream->level=0; /* 强行使流为空, 一般你不要这样做, 很容易出问题的 */
ungetc('X', stream);
PC;
ungetc('Y', stream);
fflush(stream);
PC;

```

```

PC,
PC,
printf("\n 重新开始");
fsetpos(stream,&pos);          /* 调试时看看有没有恢复原文件指针位置? */
fseek(stream,0L,SEEK-SET);
ungetc('j',stream);
ungetc('k',stream);
PC,
PC,
PC,
}
/* 程序输出,A,s,W,D,B,t,C,X,0xffff,0xffff,0xffff,
   重新开始k,j,0xffff,          */

```

C>TYPEP STD59.C

```

#include "stdio.h"
#include "ctype.h"
#define PCH ch=fgetc(stdin);if(isprint(ch))printf("ch=%c\n",ch); \
    else if (ch == 0xa)printf("ch='\n'\n"); \
    else printf("#%#x\n",ch)
#define FOO c=getchar(); \
    printf("c=%c\n",c); \
    PCH; \
    ungetc(c,stdin); \
    PCH
/* 单步调试时光标不会进入这些宏定义区域 */

main()
{
    char c,ch;
    printf("本程序用于观察回退函数ungetc()产生的结果\n");
    printf("请不输任何字母而直接按回车键,然后再按一次回车键:");
    FOO;
    printf("只输一个字母后再按回车键:");
    FOO;
    printf("输入多个不同的字母后再按回车键:");
    FOO;
}
/* 程序输出,本程序用于观察回退函数ungetc()产生的结果
   请不输任何字母而直接按回车键,然后再按一次回车键:
   c=
   ch='\n'
   ch='\n'
   只输一个字母后再按回车键:A
   c=A
   ch='\n'
   ch=A
   输入多个不同的字母后再按回车键,XYZ

```

```

c=X
ch=Y
ch=X
*/

```

—65 int —Cdecl eof(int handle);

它检测与文件句柄 handle 相联的文件是否到了文件末尾。如当前位置是文件末尾，则返回 1。返回 0 表示还未到达文件尾或者 handle 是设备文件句柄。返回 -1 则表示出错，并置 errno=6(EBADF)。

```

C>TYPE STD60.C
#include "io.h"
#include "dos.h"
#include "sys\stat.h"
#include "fcntl.h"
#define PEND end==eof(handle);printf("%d\t",end)
main()
{ int handle,end; /* 假定 qq.c 内容为 ABCDEF */
  if( (handle=open("qq.c",O_RDONLY,S_IREAD))<0)exit(1);
  PEND;
  lseek(handle,2L,SEEK_SET);
  PEND;
  lseek(handle,1L,SEEK_END); /* 文件指针已指向文件末尾 */
  PEND;
  printf("\n");
  handle=1; /* handle 与 stdout 标准输出文件相联 */
  PEND;
  handle=45; /* 假定 CONFIG.SYS 中设置最大文件数为 20 */
  PEND;
  handle=-10; /* 错误的文件句柄 */
  PEND;
}
/* 程序输出,0 0 1
0 1 -1 */

```

—66 #define feof(f) ((f)->flags & _F_EOF)

是检查流 (f) 的 flags 位上是否有文件结束标志 _F_EOF 置位的宏。由于 _F_EOF 置位是在调用其它函数时才发生，因此，使用本宏时应注意到这一点（参见下面例程）。

对文件读写操作时将检测文件结束标志是否置位。如果置位，它返回非 0 值，而且置位一直保持到文件被关闭或调用 rewind() 函数后才被清 0。每一次对文件进行输入操作都复位文件结束标志。在程序 STD60.C 调试时可以用调试表达式 fp->flags & 0x0020 观察 feof(fp) 的变动情况。

适用于 UNIX 系统。

```

C>TYPE STD61.C
#include "stdio.h"
main()
{
FILE *fp;

```



```

int inter,k;
char c;          /* 假定 QQ.C 的内容为 0x41 0x42 0x43 0x0d 0x0a 0x1a */
fp=fopen("qq.c","rb");
for(k=0;k<6;k++)
if( (c=fgetc(fp)) != EOF)printf("%#x, ",c);
else printf("[%#x]",c);
printf("\n");
fseek(fp,0L,SEEK-SET);
for(k=0;k<4;k++)
{
inter=getw(fp);
printf("%#x\n",inter);
}
}

```

/* 在集成环境下选 Options/Compiler/Code generation/Alignment/Byte (或word)

程序输出: 0x41, 0x42, 0x43, 0xd, 0xa, 0x1a,

0x4241

0xd43

0x1a0a

0xffff /* 0xffff 即 -1 */

如将语句 fp=fopen("qq.c","rb"); 改成 fp=fopen("qq.c","r");, 程序输出:

0x41, 0x42, 0x43, 0xa, [0xffff][0xffff]

0x4241

0xa43

0xffff

0xffff */

C>TYPE STD62.C

#include "stdio.h"

int getw(FILE *fp) /* 等效函数 */

{

int ch,res;

if((ch=getc(fp)) != EOF) /* 检查取得的第一个字符 ch */

if((res = getc(fp)) == EOF) /* 检查取得的第二个字符 res */

ch=EOF;

else (* ((char *)&(ch)+1)) = res; /* 将 ch 高位装 res */

/* 可用调试表达式观察,例如,

ch & 0xFF, c: 'A'

* ((char *)&ch+1): 'B'

(char *)&ch+1, p: DS:FFD5

&ch+1, DS:FFD6

&ch, DS:FFD4

ch, x: 0x4241

从中可见指针加法的例子 */

return (ch);

}

main()

```

{
FILE *fp;
int inter; /* 假定 QQ.C 的内容为 0x41 0x42 0x43 0x0d 0x0a 0x1a */
fp=fopen("qq.c","rb");
while( ! feof(fp)) /* 不难用调试表达式 fp->flags & 0x0020 进行单步跟踪 */
{
/* (当然不能用 fp->flags & _F_EOF) 发现,只有调用了 */
inter=getw(fp); /* getw() 时才使 fp->flags 发生变化,因而比期望的多 */
printf(" %x\n",inter); /* 了一次,这是在用 feof() 时要注意的。 */
}
}
/* 用 fp=fopen("qq.c","r"); 时输出 用 fp=fopen("qq.c","rb"); 时输出
0x4241 0x4241
0xa43 0xd43
0xffff 0x1a0a
0xffff */

```

—67 long —Cdecl tell(int handle);

它返回与文件句柄 handle 相联的文件指针的当前位置,即从文件头部到当前位置的字节数。它相当于语句

lseek(handle,0L,SEEK—CUR);

—68 long —Cdecl ftell(FILE *stream);

在调用成功时返回文件当前指针的位置,即指向与流 stream 相联文件从头开始算起的第几个字节处,为一个长整数(注意:它不是流当前指针的位置。因文件长度可能大于 64K,所以返回长整型)。当在文件头部时,它返回 0。调用失败时返回 -1L。如果所指定的流不能随机搜索,例如控制台,则返回值没有意义。它适用于所有的 UNIX 系统。

C>TYPE STD63.C

```

#include "stdio.h"
#include "io.h"
static int pascal near Displacement (FILE *fp) /* 决定新 fp->level */
{
register level;
int disp;
register char *pch;
disp=level=fp->level;
if(fp->flags & _F_BIN)return disp; /* 对二进制文件 */
pch=fp->curp;
while(level--)
if('\n' == *pch++) disp++; /* 处理换行符 */
return disp;
}
long ftell(register FILE *stream) /* 等效函数 */
{
long n;
if( fflush(stream) ) return -1L;
n=lseek(stream->fd,0L,SEEK—CUR); /* 移位文件读写指针到文件指针当前位置 */

```

```
return( stream->level > 0 ) ? n - Displacement(stream) : n;
}
```

—69 long —Cdecl lseek(int handle, long offset, int fromwhere);

它把与文件句柄 handle 相联的文件读写指针移到由 fromwhere 所指的文件地址加上偏移量 offset(单位是字节。注意,它是长整型数,当它是具体数字时别忘了在数字后面加上字母 l 或 L)。它是一个计算文件指针新位置的函数,fromwhere 是移动起点。在 io.h 中定义了三个符号常量用于说明 fromwhere:

```
#define SEEK_CUR    1    /* 文件指针在当前位置 */
#define SEEK_END    2    /* 文件指针在文件末尾 */
#define SEEK_SET    0    /* 文件指针在文件开头 */
```

它相当于 DOS 功能调用 42H,入口参数是,AH=42H,AL=SEEK_CUR || SEEK_END || SEEK_SET, BX=handle,DX=offset,CX=offset+2。

返回从文件开始处算起的文件指针新位置的偏移量。在出错时,它返回 -1L,并置 errno=6(EBADF),或 19(EINVAL)。如果 handle 是设备文件,则由于它们无查找能力。因此函数返回值便没有定义。

适用于 UNIX 系统。

—70 int —Cdecl fseek(FILE *stream, long offset, int whence);

重新定位与已打开流 stream 相联的文件指针,指针的新位置被定在距离 whence 为 offset 个字节的地方。whence 可取值在 stdio.h 定义为

```
#define SEEK_CUR    1    /* 当前文件指针所在的位置 */
#define SEEK_END    2    /* 文件结束位置 */
#define SEEK_SET    0    /* 文件开始位置 */
```

它将忽略由 ungetc() 回退的任何字符。它清除文件结束标志(—F—EOF)。调用成功(或者说文件指针移动成功时)返回 0,否则返回 -1。

由于调用 fseek() 时它将先行整理流,因此以后的操作可以是输入,也可以是输出。

注意:offset 的类型是 long(因文件长度可能超过 64K),因此在用数字表示 offset 时别忘了在数字后加上字母 L,否则可能会出错。

它适用于所有的 UNIX 系统。

```
C>TYPE STD64.C
#include "stdio.h"
#include "io.h"
static int pascal near Displacement (FILE *fp) /* 决定新 fp->level */
{
    register level;
    int disp;
    register char *pch;
    disp=level=fp->level;
    if(fp->flags & _F_BIN)return disp; /* 对二进制文件 */
    pch=fp->curp;
    while(level--)
        if('\n' == *pch++) disp++; /* 处理换行符 */
    return disp;
}
```

```

}
int fseek(register FILE *fp, long offset, int whence) /* 等效函数 */
{
    if(fflush(fp))return EOF; /* 参见 fflush() */
    if(whence == SEEK-CUR && fp->level > 0) /* 调用条件: 只对输入流 */
        offset -= Displacement(fp);
    fp->flags &= ~(F-OUT | F-IN | F-EOF); /* 将三位值清 0 */
    fp->level = 0; /* 流为空 */
    fp->curp = fp->buffer; /* 缓冲区当前指针指向缓冲区头部 */
    return (lseek(fp->fd, offset, whence) == -1L)? EOF: 0;
} /* 调用成功返回 0, 否则返回 -1 */
main() /* 可用调试表达式 level, in[0] 和 fp[0] 观察调试结果 */
{ /* in[0] 和 fp[0] 可合并成一个 fp[0], 不过要将流名 in 全改成 fp */
    FILE *in;
    long filesize, curpos, curpos1, curpos2, curpos3, curpos4;
    in = fopen("qq.c", "r"); /* 假定 qq.c 内容为二行: ABCDEFGH 和 abcd */
    curpos = ftell(in); /* ftell() 报告当前文件指针位置 */
    fseek(in, 0L, SEEK-END);
    filesize = ftell(in);
    fseek(in, curpos, SEEK-SET);
    curpos1 = ftell(in);
    fseek(in, curpos + 1L, SEEK-SET);
    curpos2 = ftell(in);
    fseek(in, curpos + 1L, SEEK-CUR);
    curpos3 = ftell(in);
    fprintf(stdout, "无任何回退字符时: %c", fgetc(in));
    fseek(in, 0L, SEEK-CUR); /* 注意: 不要将此句理解成未做任何事情 */
    ungetc('T', in); /* 回退一个字符 */
    fseek(in, curpos + 1L, SEEK-CUR);
    curpos4 = ftell(in);
    fprintf(stdout, "有回退字符T时: %c", fgetc(in));
    printf("\n 文件长度 = %ld\n", filesize);
    printf(" %ld, %ld, %ld, %ld, %ld\n", curpos, curpos1, curpos2, curpos3, curpos4);
}
/* 有 fseek(in, 0L, SEEK-CUR); 语句时程序输出 (实际跟有无 ungetc('T', in); 无关):
    无任何回退字符时: C, 有回退字符T 时: E
    文件长度 = 14
    0, 0, 1, 2, 4
    无fseek(in, 0L, SEEK-CUR); 语句时程序输出 (实际跟有无 ungetc('T', in); 无关):
    无任何回退字符时: C, 有回退字符T 时: D
    文件长度 = 14
    0, 0, 1, 2, 3 */

```

—71 void —Cdecl rewind (FILE * stream);

将流 stream 的当前指针定在与流相联文件的开始位置。该函数功能有时也称为【重绕】它

适用于所有 UNIX 系统。它的源码是

```
#include <stdio.h>
void rewind(FILE *stream)
{
    if(fseek(stream, 0L, seek-set) == 0) /* 如果调用 fseek() 成功 */
        stream->flags &= ~FERR; /* 清除错误标志 */
    /* 如果调用失败,则什么也不做 */
}
```

—72 int —Cdecl fgetpos(FILE *stream, fpos_t *pos);

保存与流 stream 相联的文件的当前文件读写指针的位置,它将离开文件头部的字节数存在指针 pos 所指的变量中。类型 fpos_t 在 stdio.h 中定义为

```
typedef long fpos_t;
```

函数调用成功返回 0,否则返回非 0 值。它实际上相当于调用语句

```
{return (( *pos=ftell(stream)) == -1) ? -1: 0; }
```

—73 int —Cdecl fsetpos(FILE *stream, const fpos_t *pos);

重新将与流 stream 相联的文件指针定位到先前刚刚调用 fgetpos() 时的位置。该位置由指针 pos 指出。它相当于调用语句

```
{return fseek(stream, *pos, SEEK-SET);}
```

显然,函数具有调用 fseek() 的结果,如清除 stream->flags 中的文件结束标志 (FEOF),并忽略对文件已有的回退字符操作(调用 ungetc()) 产生的结果。在调用本函数后,随后对文件的操作可以是输入或输出。

函数调用成功返回 0,否则返回非 0 值。

C>TYPE STD65.C

```
#include "stdio.h"
#define PLEN len=ftell(fp),printf("len=%ld\n",len)
#define PLEN-C printf("len=%ld, ",len); c=fgetc(fp);\
len=ftell(fp),printf("c=%c, newlen=%ld\n",c,len)

main()
{
    FILE *fp;
    long len;
    fpos_t pos;
    char c;
    char buf[2];
    fp=fopen("qq.c","r"); /* 假定 QQ.C 的内容为 ABCDEFGH */
    setvbuf(fp,buf,-IOFBF,2);
    PLEN;
    fseek(fp,2L,SEEK-SET); /* 将当前文件指针移过 2 个字节 */
    /* 原指向字母 A,现指向字母 C */

    PLEN;
    fgetpos(fp,&pos); /* 保存当前文件指针位置 */
    PLEN-C;
```

```

PLEN-C;
PLEN-C;
fsetpos(fp,&pos);          /* 使文件指针指向原保存位置 */
PLEN-C;
}
/* 程序输出: len=0
           len=2
           len=2, c=C, newlen=3
           len=3, c=D, newlen=4
           len=4, c=E, newlen=5
           len=5, c=C, newlen=3      */

```

—74 #define ferror(f) ((f) -> flags & _F_ERR)

是检查流 (f) 的 flags 位上是否有出错标志 _F_ERR 置位的宏。如检测到有错误, 它返回非0值。它一旦被置位, 它将保持不变, 除非调用了 clearerr()、rewind() 或关闭了流时才改变。它适用于 UNIX 系统。

—75 void _Cdecl clearerr(FILE *stream);

将与流 stream 相关的文件状态标志 flags 的位 5 (文件结束标志, 当文件指针指向文尾时该位为 _F_EOF) 和位 4 (出错标志, 当出错时该位为 _F_ERR) 复位, 即使这两位恢复为 0 值。flags 是结构 FILE 的成员, 它和符号常量 _F_ERR 和 _F_EOF 在 stdio.h 中有定义。

适用于 UNIX 系统。

C>TYPE STD66.C

```

#include "stdio.h"
#define PFP if(fp != NULL)printf("fp=%p,",fp);\
    else printf("fp=NULL,")\
    printf("fp->flags=%x\n",fp->flags);\
    if((fp->flags & _F_ERR) != 0)printf("错误标志位置位\n");\
    else printf("NO SET _F_ERR ! \n");\
    if((fp->flags & _F_EOF) != 0)printf("文件结束标志置位\n");\
    else printf("NO SET _F_EOF ! \n")

main()
(
FILE *fp;
static char s[10];
fp=fopen("qw.c","r"); /* 假定文件 qw.c 并不存在 */
fread(s,1,1,fp);
PFP;
fp=fopen("qq.c","r"); /* 假定文件 qq.c 存在 */
PFP;
fseek(fp,1L,SEEK-END);
PFP;
fread(s,1,1,fp);
PFP;
printf("读入字符=%c\n",s[0]);
fp=fopen("qw.c","r");

```

```

fread(s,1,1,fp);
PFP;
clearerr(fp);
PFP;
}
/* 程序输出,fp=NULL,fp->flags=0x90
    错误标志位置位
    NO SET --F--EOF !
    fp=0456,fp->flags=0x5
    NO SET --F--ERR !
    NO SET --F--EOF !
    fp=0456,fp->flags=0x5
    NO SET --F--ERR !
    NO SET --F--EOF !
    fp=0456,fp->flags=0x25
    NO SET --F--ERR !
    文件结束标志位置位
    读入字符=
    fp=NULL,fp->flags=0x90
    错误标志位置位
    NO SET --F--EOF !
    fp=NULL,fp->flags=0x80
    NO SET --F--ERR !
    NO SET --F--EOF !
    Null pointer assignment */

```

清除这两个标志有时是十分重要的,参见程序 STD67.C。

C>TYPE STD67.C

```

#include "stdio.h"
main()
{
    FILE *input, *output;
    char c;
    input=fopen("qq.c","rb"); /* 假定文件 QQ.C 存在,且以二进制形式打开 */
    output=fopen("qw.c","wb");
    while( ! feof(input) )
    {
        c=getc(input); /* 读字符时发现错误则标志位置位 */
        if(ferror(input))clearerr(input); /* 如不清除错误标志,该错误标志 */
        else { /* 一直存在,读写将无法进行下去 */
            /* 因为读写字符前要检查这些标志 */
            putc(c,output);
            if( ferror(output) )clearerr(output);
        }
    }
    fcloseall();
}

```

—76 int —Cdecl lock (int handle, long offset, long length);

设置文件共享锁,即对文件锁定一个区域,称【锁定区】,对该区域完全不允许其它 进程访问。一个试图向锁定区进行读写的程序将反复试三次操作,如都失败了,则出错。为避免出错,在文件关闭之前必须用 unlock() 解除锁定区。程序在执行完之前也应释放 所有的锁定区。

只能使用 DOS 3.0 以上版本且装有 SHARE 或网络的机器上才能调用本函数。

它相当于 DOS 功能调用 5CH 的子功能 0,入口参数是, AH=5CH, AL=0, BX=handle (文件句柄), CX;DX=offset(文件内区域内的开始偏移量), SI;DI=length(按字节计算的区域长度)。

如调用成功,返回 0,否则返回 -1。

C>TYPE STD68.C

```
#include "io.h"
main()
{int handle,k;                                /* 假定 qq.c 内容为 ABCDEF */
  if( (handle=open("qq.c",1,1))<0)exit(1);
  k=lock(handle,0,128);
                                /* 将与文件句柄 handle 相联的文件的前 128 个字节锁定 */
  printf("%d %d\n",handle,k);
}
/* 在未装网络的 DOS 5.0 环境下运行,输出:5 -1 */
```

—77 int —Cdecl unlock (int handle, long offset, long length);

它是 lock() 的反函数,解除指定的锁定区。注意:该区域应与 lock() 指定的锁定区严格对应。

只能使用 DOS 3.0 以上版本且装有 SHARE 或网络的机器上才能调用本函数。

它相当于 DOS 功能调用 5CH 的子功能 1,入口参数是, AH=5CH, AL=1, BX=handle (文件句柄), CX;DX=offset(文件区域内的开始偏移量), SI;DI=length(按字节计算的区域长度)。

如调用成功,返回 0,否则返回 -1。

—78 char * —Cdecl getpass (const char * prompt);

该函数允许用户从按照 prompt 提示符提示的信息,从控制台(一般是键盘输入)读入一个口令。操作者键入的口令不会在屏幕上显示出来。函数返回一个指针,该指针便指向用户键入的口令所在的字符串,用户最多能键入 8 个字符,字符串最后以空字符('\0')结束。返回指针是一个静态字符串指针,因此每次调用本函数时它将被重写,亦即 原有的内容将不再存在。每次调用函数时,键盘缓冲区将被刷新。

该函数常用于控制操作者的使用应用系统的权限。适用于 UNIX 系统。

C>TYPE STD69.C

```
#include "conio.h"
#include "string.h"
main()
{
  char *ptr,pass[80]="请输入一口令\n";
  static char pas[9]="ZYXW";                                /* 预置口令 */
  int n=3;
  while(n--)
```



```

{
    clrscr(); /* 清屏幕 */
    printf("口令最多8个字符,若不到8个字符,回车确认\n");
    printf("若口令对便往下执行程序,否则终止.\n");
    ptr=getpass(pass); /* 接收口令 */
    printf("你刚刚输入的口令是:%s\n",ptr);
    if( strcmp(ptr,pas) != 0)
    {
        printf("口令不对! \n");
        delay(1500); /* 延迟 */
        if(n==0)
        {
            printf("给你三次机会,可惜口令都不对! \n");
            exit(1); /* 终止程序执行 */
        }
    }
    else
    {
        printf("继续执行程序\n");
        break;
    }
}
}

```

—79 int —Cdecl stat (char *path, struct stat *statbuf);

它将文件或目录 path 的信息存到 statbuf 所指的 stat 类结构中。

如果调用成功,返回 0,否则返回 -1,并置 errno=2(ENOENT)。

结构 stat 在标头文件 stat.h 中定义为:

```

struct stat
{
    short st_dev; /* 文件或文件句柄所在的磁盘驱动器号,0=A,1=B 等 */
    short st_ino; /* 在 DOS 下无意义,一般为 0 */
    short st_mode; /* 用位值给出文件打开方式等信息,见下面符号常量说明 */
    short st_nlink; /* 一般为 1 */
    int st_uid; /* 在 DOS 下无意义,一般为 0 */
    int st_gid; /* 在 DOS 下无意义,一般为 0 */
    short st_rdev; /* 同 st_dev */
    long st_size; /* 文件长度,字节数 */
    long st_atime; /* 文件最近修改的时间 */
    long st_mtime; /* 同 st_atime */
    long st_ctime; /* 同 st_atime */
};

```

和 st_mode 相关的符号常量是:

```

#define S_IFMT 0xF000 /* 文件类型掩码 (file type mask) */
#define S_IFDIR 0x4000 /* 目录 (directory) */

```

```

#define S—IFIFO 0x1000 /* 先进先出指定 (FIFO special) */
#define S—IFCHR 0x2000 /* 专用字符 (character special) */
#define S—IFBLK 0x3000 /* 块指定 (block special) */
#define S—IFREG 0x8000 /* 或 0x0000, 常规文件 (regular) */
#define S—IWRITE 0x0080 /* 用户可写 (owner may write) */
#define S—IEXEC 0x0040 /* 用户可执行 (owner may execute)
                        目录搜索 <directory search> */

```

—80 int —Cdecl fstat (int handle, struct stat * statbuf);

它将与文件句柄 handle 的相联的文件信息存到 statbuf 所指的 stat 类结构中。
如果调用成功,返回 0,否则返回 -1,并置 errno=6(EBADF)。

C>TYPE STD70.C

```

#include "sys\stat.h"
#include "fcntl.h"
#include "stdio.h"

#define PRT printf("st—dev=%d\n",s.st—dev);\
            printf("st—ino=%d\n",s.st—ino);\
            printf("st—mode=%#x\n",s.st—mode);\
            printf("st—nlink=%d\n",s.st—nlink);\
            printf("st—uid=%d\n",s.st—uid);\
            printf("st—gid=%d\n",s.st—gid);\
            printf("st—rdev=%d\n",s.st—rdev);\
            printf("st—size=%ld\n",s.st—size);\
            printf("st—atime=%ld\n",s.st—atime);\
            printf("st—mtime=%ld\n",s.st—mtime);\
            printf("st—ctime=%ld\n",s.st—ctime)

main()
{
    struct stat s;
    int k,handle;
    k=stat("c:\\tc",&s); /* 目录 */
    if(k== -1)exit(1);
    PRT;
    k=stat("CON",&s); /* 外设 */
    if(k==0)PRT;
    handle=open("m80.c",O—RDONLY);/* 对文件也可用 stat(),则不必打开文件 */
    k=fstat(handle,&s); /* 文件 */
    if(k==0)PRT;
}

/* 程序输出后经整理为:

```

目录	外设	文件
st—dev=2	st—dev=-1	st—dev=2
st—ino=0	st—ino=0	st—ino=0
st—mode=0x4140	st—mode=0x2180	st—mode=0x8100
st—nlink=1	st—nlink=1	st—nlink=1

st—uid=0	st—uid=0	st—uid=0
st—gid=0	st—gid=0	st—gid=0
st—rdev=2	st—rdev=-1	st—rdev=2
st—size=0	st—size=112	st—size=793
st—atime=732917260	st—atime=0	st—atime=754951362
st—mtime=732917260	st—mtime=0	st—mtime=754951362
st—ctime=732917260	st—ctime=0	st—ctime=754951362

*/

—81 unsigned —Cdecl umask (unsigned cmask);

重新设置由 open 和 read 相关文件的读 / 写权限屏蔽。

此函数原型虽然在 io.h 中出现,但 Turbo C 2.0 对此未让用户使用它。

关于宏—IO—H 的一点说明(对其它宏也可仿此理解):

如果在程序头部定义了 —IO—H ,则 Turbo C 将不检查函数原型,而 io.h 中涉及的函数仍可用。

C>TYPE STD71.C

```
#define —IO—H
#include "stdio.h"
#include "io.h"
#include "fcntl.h"
main()
{
    static char s[]="AB\nC\x1A";
    int fp,flag;
    if( (fp=open("qq.c",O—WRONLY|O—APPEND))<0)exit(1);
    flag=—write(fp,s,10);
    printf("flag=%d\n",flag);
    close(fp);
}
```

第三十一章 格式输入与输出函数

31.1 格式输出函数

- 1 int —Cdecl printf(const char *format, ...);
- 2 int —Cdecl vprintf(const char *format, va—list arglist);
- 3 int —Cdecl fprintf(FILE *stream, const char *format, ...);
- 4 int —Cdecl vsprintf(FILE *stream, const char *format, va—list arglist);
- 5 int —Cdecl sprintf(char *buffer, const char *format, ...);
- 6 int —Cdecl vsprintf(char *buffer, const char *format, va—list arglist);
- 7 int —Cdecl cprintf(const char *format, ...);

它们被称为【格式化输出函数】，主要是因为它们都含有格式化串 format 的原因。它们以 printf() 为基础，分别加上前缀。各前缀的含义是：

- f 表示与流 stream 相关
- s 表示与缓冲区 buffer 相关
- v 表示从 va—arg 数组的 va—list arglist 中接受可变参数（参见《接收自变量个数可变的宏》一章）
- c 表示控制台函数。

31.1.1 参数 format 的书写规则

format 实际是一个字符串，它包含两类字符，一是单纯的字符（plain characters），这些字符只是简单地被拷贝到输出流中，换句话说，它们将被原样输出；另一类字符则按一定的规则组成【格式化字符串】（或称【转换规范，conversion specification】），一个格式化串对应地说明一个变量或表达式的值按何种规范化格式输出。格式化串可以作为一个整体（即当作一个单纯字符那样）任意插入单纯字符串中。format 中允许多个格式化串（一个没有也行）存在。

注意：对串中转义符（像 \n 等）应按规定书写。

格式化串规定由下列几部分组成

% [flags] [width] [.prec] [F|N|h|l] [type]

方括号内的内容表示是任选的。各可选项的摆放顺序绝不能颠倒。以下逐一详细说明之。

1. %

每一格式字符串必须以百分号开始，随后才是各选项；

2. flags

称【标志字符串】，用于说明输出结果的对齐方式、正负数头部是否需要加 + 或 - 号、及对类型字符 type（见下面说明）的影响。

它可以由加号（+）、减号（-）、井号（#）、空格符或数字 0 组成，这几种字符可按任意顺序和组合形式出现。当然，每种字符最多只能在子串中出现一次。

(1) 如果加号在标志字符串子串中出现，则当类型 type 为 x、X、c、s 或 p 外时，输出开

头总有加号或减号出现。因此加号可称为数值符号标志。

```
C>TYPE STDFOR1.C
#include "stdio.h"
#define PCH(CH)printf("%+d, ",CH);\
                /* printf("%+f, ",CH);\ 对整数不能用此类语句 */\
                printf("%+c, ",CH);\
                printf("%+x\n",CH)
                /* 注意:对宏中注释语句行后也应加续行符,不要忘记 */

main()
{
    int x=-14;
    double y=3.14;
    char c1='a';
    char c2='B\n';
    char c3=0xff22;
    char c4=0x22ff;
    PCH(x);
    PCH(c1);
    PCH(c2);
    PCH(c3);
    PCH(c4);
    printf("%+f\n",y);
}
/* 程序输出: -14,    , fff2
              +97, a, 61
              +66, B, 42
              +34, ", 22
              -1,    , ffff
              +3.140000          */
```

(2) 当子串中给出宽度指示符 width(见下面解释) 时,如果又给出了减号标志字符,则结果左对齐,即输出字符从指定宽度的左边第一个位置开始,当输出字符个数小于宽度时,右边不足部分以空格符填充;否则右对齐,左边不足指定宽度部分填充空格或 0。因此减号被称为对齐标志。

```
C>TYPE STDFOR2.C
#include "stdio.h"
#define PCH(CH)printf("=%4d=",CH); \
                printf("%04d=",CH); \
                printf("%-4d=",CH); \
                printf("%-04d=\n",CH); \
                printf("[% * d=",k,CH); \
                printf("%- * d=",k,CH); \
                printf("[%0 * d=",k,CH); \
                printf("%-0 * d=\n",k,CH)
                /* 宽度指示符 * 只对整数有效,变量 k 指出具体宽度 */

main()
```

```

{
int    k=3;
int    x=-13545;
double y=3.14;
char   c1='a';
char   c2='B\n';
char   c3=0xff22;
char   c4=0x22ff;
PCH(x);
PCH(c1);
PCH(c2);
PCH(c3);
PCH(c4);
printf("=%8f=",y);
printf("%08f=",y);
printf("%-8f=",y);
printf("%-08f=\n",y);
printf("=123456789 12345678=123456789 123456789\n");
printf("=%18f=",y);
printf("%018f=\n",y);
printf("=%-18f=",y);
printf("%-018f=\n",y);
}

```

```

/* 程序输出:=-13545=-13545=-13545=-13545=
          [-13545=-13545]=[-13545=-13545]
          = 97=0097=97 =97 =
          [ 97=97 ]=[097=97 ]
          = 66=0066=66 =66 =
          [ 66=66 ]=[066=66 ]
          = 34=0034=34 =34 =
          [ 34=34 ]=[034=34 ]
          = -1=-001=-1 =-1 =
          [ -1=-1 ]=[-01=-1 ]
          =3.140000=3.140000=3.140000=3.140000=
          =123456789 12345678=123456789 123456789
          = 3.140000=00000000003.140000=
          =3.140000 =3.140000 = */

```

(3) 如果标志字符选用空格符,则当输出值为非负值时,则以空格符代替数值开头的加号;而对负数则以减号开头。注意,如果在子串中同时给出加号和空格标志字符,则加号优先,即空格被忽略。因此,空格可称为真实数符标志。

C>TYPE STDFOR3.C

```

#include "stdio.h"
#define PS(X)printf("x=%"X"d=",x);\
               printf("y=%"X"d=",y);\
               printf("z=%"X"d=",z);\
               printf("x1=%"X"f=",x1);\

```

```

        printf("y1=%"X"f=",y1),\
        printf("z1=%"X"f=",z1),\
main()      /* 只管不要续行,但在最后加上续行符时下面至少要有一空行 */
{
    int x=88,y=-888,z=0;
    float x1=0.0,y1=-2.3,z1=0.99;
    PS(" ");
    printf("\n");
    PS(" +");
}
/* 程序输出:  x= 88=y=-888=z= 0=x1= 0.000000=y1=-2.300000=z1= 0.990000=
              x=+88=y=-888=z=+0=x1=+0.000000=y1=-2.300000=z1=+0.990000=
              =
*/

```

(4) 如果使用了井号标志字符,则要看其后的类型字符(type)是什么才能决定产生什么影响。它主要决定在非0数字前面添0x(0X)或小数尾部是否添0,因此可称为添0(0x)标志。

<1> 当类型字符为c、s、d、i或u时,不产生任何影响;

<2> 当为x(或大写字母X)时,在输出的非0值开头有0x(或0X)出现,这对标明输出数值是十六进制数非常有用。

```

C>TYPE STDFOR4.C
#include "stdio.h"
#define PRESULT printf("调用结果=%d\n",r)
main()
{
    int r=100;
    int x=0xff15;
    double y=3.14;
    int s=0x7f15;
    r=printf("=%d.",x);PRESULT;
    r=printf("=%x.",x);PRESULT;
    r=printf("=%#8d.",x);PRESULT;
    r=printf("=%#8x.",x);PRESULT;
    r=printf("=%#08d.",x);PRESULT;
    r=printf("=%#08x.",x);PRESULT;
    r=printf("=%d.",s);PRESULT;
    r=printf("=%x.",s);PRESULT;
    r=printf("=%#8d.",s);PRESULT;
    r=printf("=%#8x.",s);PRESULT;
    r=printf("=%#08d.",s);PRESULT;
    r=printf("=%#08x.",s);PRESULT;
    r=printf("=%#lf.",y);PRESULT;
    r=printf("=%#x.",y);PRESULT; /* 错误的输出格式,但 Turbo C 不作检查 */
    r=printf("=%p.",y,x);PRESULT; /* 错误的输出格式,但 Turbo C 不作检查 */
}

```

```

/* 输出结果:=-235,调用结果=6
   =ff15,调用结果=6
   = -235,调用结果=10
   = 0xff15,调用结果=10
   =-0000235,调用结果=10
   =0x00ff15,调用结果=10
   =32533,调用结果=7
   =7f15,调用结果=6
   = 32533,调用结果=10
   = 0x7f15,调用结果=10
   =00032533,调用结果=10
   =0x007f15,调用结果=10
   =3.140000,调用结果=10
   =0x851f,调用结果=8
   =851F,调用结果=6 */

```

<3> 当为 e、E 或 f 时跟它们前面没有 # 是一样的。在缺省情况下数字中有小数点出现。

<4> 当为 g 或 G 时,不删除小数部分尾部的 0(当它们前面没有 # 号时则要删除的)。

C>TYPE STDFOR5.C

```

#include "stdio.h"
#define PXY(S)      printf("=%e=",S);\
                    printf("=%E=",S);\
                    printf("=%f=",S);\
                    printf("=%g=",S);\
                    printf("=%G=\n",S);
#define PJIN(S)     printf("=%#e=",S);\
                    printf("=%#E=",S);\
                    printf("=%#f=",S);\
                    printf("=%#g=",S);\
                    printf("=%#G=\n",S);

main()
{
double x=1,y=0.58,u=0,v=-5.6732;
PXY(x);
PXY(y);
PXY(u);
PXY(v);
PJIN(x);   PJIN(y);   PJIN(u);   PJIN(v);
}

```

```

/* 程序输出: =1.00000e+00=1.00000E+00=1.000000=1=1=
   =5.80000e-01=5.80000E-01=0.580000=0.58=0.58=
   =0.00000e+00=0.00000E+00=0.000000=0=0=
   =-5.67320e+00=-5.67320E+00=-5.673200=-5.6732=-5.6732=
   =1.00000e+00=1.00000E+00=1.000000=1.00000=1.00000=
   =5.80000e-01=5.80000E-01=0.580000=0.580000=0.580000=
   =0.00000e+00=0.00000E+00=0.000000=0.00000=0.00000=

```


$-5.67320e+00 = -5.67320E+00 = -5.673200 = -5.67320 = -5.67320 =$

(5) 如果标志字符为数字 0, 而同时又指定宽度指示符时, 非 0 值的左边不足部分将全部用数字 0 填满, 因此可称为左满 0 标志符。

3. width

这是输出值占据的最小宽度 (或者说输出的字符个数), 因而称【宽度指示符】。说它占用最小宽度是指当实际输出值的字符数大于这个宽度时则以实际字符数输出, 因而在任何情况下都不必担心会出现输出值被截断的情况。

缺省时是没有指定宽度, 因而一般情况下按实际输出值的长度输出。指定输出宽度有时是有用的, 例如, 对多行输出需要上下对齐而成表格状时是就可用它, 以便输出数据看起来清楚明了。

宽度指示符有两种形式:

(1) n

n 是一个具体的正整数。如果输出值少于 n 个字符, 则根据对齐规则对不足部分填于适当的字符 (空格或 0)。

(2) *

* 是一个表明具体的宽度将由 format 串后面的参数表中的某个参数给出, 该参数一定紧挨在它要说明的变量前面 (或左边)。例如, 下列语句中 variable—width 是变量 variable 的宽度参数:

```
printf("数值=%*d", variable—width, variable);
```

4. .prec

它是表明数值输出精度的指示符, 总以小数点开头, 随后可能跟数字 0、数值 n 或字符星号 *。这三个字符只能有其中一个出现在精度指示符中。

在格式子串中, .prec 也可以不出现, 这时就根据字符类型 (type) 使用缺省精度:

字符类型	缺省精度值 (包括小数点在内的位数)
d i o u x X	1
e E	6 (有效位数)
f	7
g G	6 (有效位数)
s	从第一个字符开始输出, 直到空字符 ('\\0') 为止
c	无影响

C>TYPE STDFOR6.C

```
#include "stdio.h"
main()
{
    char *s=" ABCD";      /* 首字符为空格符 */
    char ch='H';
    int k=2;
    float x=3.1415925;
    printf("%f\\n", x);    /* 输出 3.141593 */
    printf("%e\\n", x);    /*      3.14159e+00 */
    printf("%E\\n", x);    /*      3.14159E+00 */
    printf("%g\\n", x);    /*      3.14159 */
    printf("%G\\n", x);    /*      3.14159 */
}
```

```

printf("%f\n",x); /* 3.1 */
printf("%le\n",x); /* 3e+00 */
printf("%lE\n",x); /* 3E+00 */
printf("%lg\n",x); /* 3 */
printf("%lG\n",x); /* 3 */
printf("%Of\n",x); /* 3 */
printf("%Oe\n",x); /* 3e+00 */
printf("%OE\n",x); /* 3E+00 */
printf("%Og\n",x); /* 3e+00 */
printf("%OG\n",x); /* 3E+00 */
printf("=%c=\n",ch); /* =H= */
printf("=%2c=\n",ch); /* =H= */
printf("=%.*c=\n",k,ch); /* =H= */
printf("=%s=\n",s); /* =ABCD= */
printf("=%0s=\n",s); /* == */
printf("=%3s=\n",s); /* =AB= */
printf("=%.*s=\n",k,s); /* =A= */
printf("=%.*s=\n",2,s); /* =ABCD= */
printf("=%.*s=\n",4,"ab"); /* =ab= */
printf("%0d\n",1); /* 输出数字 1 */
printf("%ld\n",1); /* 输出数字 1 */
printf("%2d\n",1); /* 输出数字 01 */
printf("%0g\n",1); /* 这三句或者不能执行,出现浮点格式没有连接的提示 */
printf("%lg\n",1); /* Printf;floating point formats not linked */
printf("%2g\n",1); /* Abnormal program termination */
} /* 或者会输出错误的结果,这是要特别注意的 */

```

(1). 0 (小数点后跟数字零)

对于字符类型为 `d i o u x X`, 精度自动选为缺省值; 而对 `e f g` 则无小数点出现, 不输出小数部分; 对 `s` 则没有字符输出。

(2). n (小数点后跟数值 n)

它用正整数 `n` 指明输出精度, 精度和字符类型相关:

字符类型	精度
<code>d i o u x X</code>	至少有 <code>n</code> 个字符输出。当实际输出字符个数少于 <code>n</code> 时, 输出左边 填 0, 否则按实际字符输出。
<code>e E</code>	指明小数点后的 <code>n</code> 个字符要输出。最后一位数字被舍入。
<code>g G</code>	最多输出 <code>n</code> 个有效数字。
<code>s</code>	当输出字符大于或等于 <code>n</code> 时输出 <code>n</code> 个字符, 超出部分被截断或舍入; 否则按实际字符数输出。

C>TYPE STDFOR7.C

```
#include "stdio.h"
```

```
main()
```

```
{
```

```
int x1=21034,x2=-567,x3=1,x4=-8;
```

```
unsigned y1=056,y2=56;
```

```
unsigned z1=0xFE,z2=0567; /* 注释中为输出值 */
```

```

double x5=3.141525;
printf("=%%.2d=\n",x1); /* =21034= */
printf("=%%.2i=\n",x2); /* =-567= */
printf("=%%.4d=\n",x3); /* =0001= */
printf("=%%.4i=\n",x4); /* =-008= */
printf("=%%.4o=\n",y1); /* =0056= */
printf("=%%.4u=\n",y2); /* =0056= */
printf("=%%.2x=\n",z1); /* =fe= */
printf("=%%.2X=\n",z2); /* =177= */
printf("=%-.2d=\n",x1); /* =21034= */
printf("=%-.2i=\n",x2); /* =-567= */
printf("=%-.4d=\n",x3); /* =0001= */
printf("=%-.4i=\n",x4); /* =-008= */
printf("=%-.4o=\n",y1); /* =0056= */
printf("=%-.4u=\n",y2); /* =0056= */
printf("=%-.2x=\n",z1); /* =fe= */
printf("=%-.2X=\n",z2); /* =177= */
printf("=%+.2d=\n",x1); /* =+21034= */
printf("=%+.2i=\n",x2); /* =-567= */
printf("=%+.4d=\n",x3); /* =+001= */
printf("=%+.4i=\n",x4); /* =-008= */
printf("=%+.4o=\n",y1); /* =0056= */
printf("=%+.4u=\n",y2); /* =+056= */
printf("=%+.2x=\n",z1); /* =fe= */
printf("=%+.2X=\n",z2); /* =177= */
printf("=%%.2f=\n",x5); /* =3.14= */
}

```

(3). * (小数点后跟星号)

类似于宽度指示符中用的星号,真正的精度在参数表中用一个参数指出。注意:它们的位置不能搞错。

C>TYPE STDFOR8.C

```

#include "stdio.h"
main()
{
char *s="ABCDEFGH";
int k1=10,k2=4; /* 精度指示符应在宽度指示符之后 */
double x=3.141525;
printf("=%*.*s=\n",k1,k2,s); /* 输出 = ABCD= */
printf("=%-*.*s=\n",k1,k2,s); /* =ABCD = */
printf("=%*.*s=\n",k2,k1,s); /* =ABCDEFGH= */
printf("=%-*.*s=\n",k2,k1,s); /* =ABCDEFGH= */
printf("=%.*f=\n",2,x); /* =3.14= */
}

```

5. F|N|h|l

它是长度修饰符,针对具体的输出字符类型只能选其中的一个(当一个不选时使用缺省

值)。

F 指明对应的输出参数按远 (far) 指针形式输出,输出形式为 xxxx:xxxx(段:偏移量)。

N 按近 (near) 指针形式输出,即以 xxxx(偏移量)输出。

注意:F 或 N 常用在 %p 的 % 的后面,它们应和对应的指针类型相适应。例如,当存储模式不同时指针的类型也会改变,这时应选用相应的 F 或 N。另外,在调试表达式中也可以使用它们。例如,对远指针 u 可用 u,Fp 表达式,调试时它可能为 u,Fp: 88AE(DS):FFDC

h 对参数类型为 dio u x X 可用之,即它只适用于短整型数。对 c s p n 字符类型不产生影响。

l 注意:它不是大写字母 L。它适用于 dio u x X e E f g G 等字符类型。特别值得指出的是,对长整型数再好使用它,否则有时会出问题的。

C>TYPE STDFOR9.C

```
#include "stdio.h"
main()
{long x=345678902;
printf("%ld\n",x); /* 正确的输出 345678902 */
printf("%Ld\n",x); /* 错误的输出 --23498 */
}
```

C>TYPE STDFOR10.C

```
#include "stdio.h"
main()
{
char *ptr="S123456789";
int x=800,y=x;
int *pint;
int far *u=&y;
pint=&x; /* pint 指向整型量 x */
printf("%p\n",ptr);
printf("%Fp\n",ptr);
printf("this is test\n",pint);
printf("\n 已写过的字符个数即x=%d\n",x);
printf("%dTHIS IS TEST\n\n",y,pint); /* 这是允许的 */
printf("%d\n",x);
/* 一般 x 和 pint 不要同时出现在 printf() 中 */
printf("123456%n=%d\n",pint,x); /* 此句执行完后 x 才能取到新值 */
printf("%d\n",x);
printf("%d=12345678\n",x,pint); /* 这是不正确的写法 */
printf("%d\n",x);
printf("12345678%nWXYZ\n",&x); /* %n 后面的字符将不被统计在内 */
printf("%p,%d\n",pint,x);
printf("%Fp\n",pint);
printf("www%Fn\n",u);
printf("%Fp,%d\n",u,y);
printf("%Np\n",u);
}
```

```

/* 程序输出:0194
056D;0194
this is test
已写过的字符个数即x=12
800THIS IS TEST
15
123456=15
6
6=12345678
6
12345678WXYZ
FFDA.8
056D;FFDA
www
796F;FFDC,3
FFDC
*/

```

6. type

字符类型有以下几种:

字符类型字母对应的参数类型输出格式。按参数类型选择相应字母,例如对 unsigned 类型应选 u。

数值类

d	整数	带数符号的十进制整数
i	整数	带数符号的十进制整数
o	整数	无数符号的八进制整数
u	整数	无数符号的十进制整数
x	整数	无数符号的十六进制整数 (0~9,a,b,c,d,e,f)
X	整数	无数符号的十六进制整数 (0~9,A,B,C,D,E,F)
f	浮点数	格式为 [-]dddd.dddd 的带数符号的数值
e	浮点数	格式为 [-]d.ddde[+/-]ddd 的带符号数值
g	浮点数	由给定值和精度确定的 e 或 f 格式,尾部的无意义的零或小数点一般情况下不输出 (在必要时可以输出)
E	浮点数	格式为 [-]d.ddddE[+/-]ddd 的带符号数值,和 e 格式,唯一不同之处只是将 e 换成 E
G	浮点数	同格式 g 相似,只是在用 e 格式时将表示指数的 e 用 E 表示

字符类

c	字符	单个字符 注意,对它也可按数值方式输出,而得其 ASCII 码值
s	字符串	输出整个字符串,或者当指定精度时输出指定个数的字符 (包括一个不输出也行)
%	不要参数	输出 % 字符。因为格式子串必须以 % 当头,所以你要输出 % 时必须用 %% 形式。如 printf("%d%%\n",x), 才能输出像 98% 这样的百分数

指针类

n	指向整数的指针	它把当前格式子串中它左边已输出的字符数存入指针所指的整型变量中,你必须用另一个 printf() 语句 输出那个整型变量的值才能知道已输出字符数。注意,它右边的字符不被统计在内。
---	---------	---

```

C>TYPE STDFOR11.C
#include "stdio.h"
main()
{
    int x=10,y=200,z=-1;
    int *px=&x, *py=&y, *pz=&z;
    printf("a23%nB567890%n%f%n",px,py,-0.4,pz);
    printf("\nx=%d, y=%d, z=%d\n",x,y,z);
}
/* 程序输出:a23B567890-0.400000
           x=3, y=10, z=19 */

```

p 指针 对远指针:段:偏移量 (xxxx:xxxx)
 对近指针:只输出偏移量 (xxxx)

31.1.2 ... (可变参数表)

这三个连续的小数点表示多个参数(参见《接收自变量个数可变的宏》一章),这些参数可以是常量、变量或表达式。每一个参数必须有一个格式子串和它对应;格式子串在格式化串中排列的顺序应和参数在参数表中排列的顺序相对应(如果有宽度指示符 * 或精度指示符 . * 出现时还应适当增加参数);而且格式子串的内容必须和参数的类别相适应。如果格式化串中的一个格式子串不能和参数表中的参数相匹配,结果就是不可预料的,很可能得到错误的结果,或者不能输出等等。

反之,参数与格式子串不匹配,结果也很难预料。不过,当参数个数多于格式子串时多余的参数将被忽略。

总而言之,书写时应严格检查这种对应关系。当出现错误时,最好将有多格式子串的 format 分解成多个只有一个格式子串的 format(为此可能要增加一些语句),这样便于查错。当确信无误后再将它们合并书写。

```

C>TYPE STDFOR12.C
#include "stdio.h"
#include "float.h"
#include "math.h"
main()
{
    char x=1;
    int k1,k2;
    k1=printf("x=%p\n",x);          /* 指针类与整数不匹配 */
    printf("k1=%d\n",k1);
    printf("k1=%d\n",k1,9+x);        /* 多余参数被忽略 */
    k2=printf("x=%d\n",x*9-1);
    printf("k2=%d\n",k2);
    k1=printf("%f\n",DBL-EPSILON-2);
    printf("%d\n",k1);
    k2=printf("%f\n",sqrt(x));
    printf("%d\n",k2);
}
/* 程序输出:x=0001

```

```

k1=7
k1=7
x=8
k2=4
-2.000000
10
1.000000
9
*/

```

31.1.3 库函数

各函数调用成功时返回输出字节数,失败时返回 EOF。

它们适用于 UNIX 系统。

—1 int —Cdecl printf(const char *format, ...);

它把输出送到预定义流 stdout 中。注意:它对可变参数输出顺序是从右到左。

C>TYPE STDFOR13.C

```
#include "stdio.h"
```

```
main()
```

```
{
```

```
int y=10;
```

```
printf("f=%d,y=%d\n",f(y),y++); /* 将它改成 printf("f=%d,y=%d\n",f(y++),y);
```

```
程序输出:f=100,y=10 */
```

```
}
```

```
f(int t)
```

```
{
```

```
return(t*t);
```

```
}
```

/* 在单步调试时利用调试表达式 y 和 t 就可先输出 y,后输出函数 f(y),即变量组是从右往左进行输出的,因而最后有 f=121, y=10 */

—2 int —Cdecl vprintf(const char *format,va—list arglist);

它同 printf() 相似,只是从 va—arg 数组 va—list arglist 中接受参数。

C>TYPE STDFOR14.C

```
#include "stdio.h"
```

```
#include "stdarg.h"
```

```
main()
```

```
{
```

```
char *s="串一";
```

```
char t[]="ABCDEF";
```

```
int y=88; /* 在一般情况下自定义函数名不要和库函数同名,最好在不区分大小写字母时也没有一个字符相同。 */
```

```
PRINTF("a0123=%s=%s=%d\n",s,t,y); /* PRINTF 的形式和 printf 一样 */
```

```
PRINTF("b45=%s=%s=%d\n","串一",t,y);
```

```
PRINTF("c6=%s=%s=%d\n","串一","ABCDEF",y);
```

```
}
```

PRINTF(char *format1) /* 这个函数参数表中只有一个参数 format1,而是靠 ptr 将其后跟的可变

参数 (s,t,y) 引入到函数中来。注意 ptr 和 format1 之间特殊的关系!
并不是所有的函数都可以这样做的 */

```
{
char *u;
va—list ptr;          /* 相当于 void *ptr      */
u=format1;
va—start(ptr,format1); /* ptr 指向可变字符数组 (s,t,y) 的头部,此句必须有 */
vprintf(format1,ptr);
va—end(ptr); /* 只是表示 format1 调用结束,以增加源程序可读性,别无它用 */
format1 +=4;      /* 指针加法 */
va—start(ptr,format1); /* ptr 指向 format1 当前所指位置 */
vprintf(format1,ptr);
va—end(ptr);
    /* 如在此处增加 format1=u; 语句 (强行使 format1 指向原位), 看看结果有什么不同
    */
va—start(ptr,format1);
printf("原始=%p,%p,%#x,%c\n",ptr,format1,*(char *)ptr,*format1);
vprintf(format1,ptr);
printf("中间=%p,%p,%#x,%c\n",ptr,format1,*(char *)ptr,*format1);
vprintf(format1,ptr); /* 在执行前一 vprintf() 后 format1 自动返回指向原位置 */
printf("最后=%p,%p,%#x,%c\n"***\n",ptr,format1,*(char *)ptr,*format1);
}
/* 程序输出(注意:最后一部分输出显然有错误了):
a0123=串一=ABCDEF=88
3=串一=ABCDEF=88
原始=FFD0,01A4,0xff9b,3
3=串一=ABCDEF=88
中间=FFD0,01A4,0xff9b,3
3=串一=ABCDEF=88
最后=FFD0,01A4,0xff9b,3
***
b45=串一=ABCDEF=88
串一=ABCDEF=88
原始=FFD0,01B4,0xffbe,%
串一=ABCDEF=88
中间=FFD0,01B4,0xffbe,%
串一=ABCDEF=88
最后=FFD0,01B4,0xffbe,%
***
c6=串一=ABCDEF=88
s=串一=469
原始=FFD0,01C7,0xffd0,s
s=串一=469
中间=FFD0,01C7,0xffd0,s
s=串一=469
最后=FFD0,01C7,0xffd0,s
```


* * *

为弄清 ptr 和 format1 的情况,可以用调试表达式观察,例如一开始有

```
(char *)s,p; DS:019B
(char *)t,p; DS:FFDA
y,x: 0x58
* (char *)ptr,6m; 9B 01 DA FF 58 00
      相当于      s      t      x
format1,p; DS:01A0
format1; *a0123=%s=%s=%d\n"          */
```

—3 int —Cdecl fprintf(FILE *stream,const char *format,...);

它和 printf() 不同之处在于,不是把输出送到特定的流 stdout 中,而是把输出送到任一指定的流 stream 中。

C>TYPE STDFOR15.C

```
#include "stdio.h"
main()
{
FILE *fp;
if((fp=fopen("QQ.C","w"))==NULL)
{
fprintf(stdout,"文件建立失败! \n"); /* 输出到屏幕(stdout) */
exit(1);
}
else fprintf(fp,"文件创建成功! \n"); /* 输出到文件 QQ.C 中 */
}
/* C>TYPE QQ.C
文件创建成功! */
```

—4 int —Cdecl vfprintf(FILE *stream,const char *format,va—list arglist);

它同 fprintf() 相似,只是从 va—arg 数组 va—list arglist 中接受参数。

—5 int —Cdecl sprintf(char *buffer,const char *format,...);

它同 printf() 相似,只是将输出送到用户指定的缓冲区 buffer 中。注意:当写入串结束时,它会自动将空字符('0')添加串的末尾,但是在写入中间是不会添加空字符的,除非人为地加入。这很容易用调试表达式观察出来。

C>TYPE STDFOR16.C

```
#include "stdio.h"
main()
{
char a[100]="ABCDEFGHJKLMNOPQRSTUVWXYZ"; /* 数组应定义得足够大 */
char *s=(char *)calloc(100,1); /* 用 calloc() 清除内存垃圾,提高可靠性 */
int w=10;
long x=0x4567;
struct {
char *t;
double y;
}uv={"ABCD",3.14};
```

```

sprintf(s,"%d,%ld\t",w,x);      /* 像 /t 等转义符也将被写入缓冲区中 */
sprintf(a,"%s=%010.4lf",uv.t,uv.y);
printf("%s",s);
printf("%s\n",a);
}

```

/* 程序输出:10,17767 ABCD=00003.1400

可输入像 a[0],20c 或 s[0],20c 那样的调试表达式观察 */

—6 int —Cdecl vsprintf(char *buffer,const char *format,va—list arglist);

它同 sprintf() 相似,只是从 va—arg 数组 va—list arglist 中接受参数。

—7 int —Cdecl cprintf(const char *format,...);

写一格式串到屏幕文本窗口内。返回输出字节数(参见《视频函数》一章)。注意:对该函数 '\n' 不能被理解成回车 / 换行,即光标移到下一行开头。这只是换行。要实现回车 / 换行必须用 '\n\r',如 cprintf("\n\r");。

31.2 格式输入函数

```

—8 int —Cdecl scanf(const char *format,...);
—9 int —Cdecl fscanf(const char *format,...);
—10 int —Cdecl vscanf(const char *format,va—list arglist);
—11 int —Cdecl fscanf(FILE *stream,const char *format,...);
—12 int —Cdecl vfscanf(FILE *stream,const char *format,va—list arglist);
—13 int —Cdecl sscanf(const char *buffer,const char *format,...);
—14 int —Cdecl vsscanf(const char *buffer,const char *format,
va—list arglist);

```

这些函数统称为【格式输入函数】。它们一般包括三个部分:指定流(stdin 或 stream)或缓冲区(buffer)作为数据输入源;格式串 format 用于控制函数扫描、转换和存储输入字段的格式;读入并经转换后的数据存入可变参数表(... 或 va—listarglist) 相应的地址单元中。某种程度上可以说它是格式输出函数的反函数。函数名的前缀含义同格式输出函数。

尽管它们和格式输出函数有许多相同之处,但是它们依然是一类较难掌握的函数,它不像格式输出函数那样好理解,因此宜多实践。

31.2.1 参数 format 的书写规则

format 和格式输出函数中的 format 类似,但有许多不同(注意:尽管在集成环境下按 F1 键你可能看到 scanf() 和 printf() 的 format 几乎用的是同一个说明,只有一处例外)。通常它有三种串组成:格式串、空白串和非空白串。每一种串可以没有,也可以出现一次以上。格式串前面(左边)可以出现其余两种串。函数调用时从左到右依次按串进行扫描(或称读入)数据、将读入数据进行转换和将转换后的数据存入地址参数表中与字段对应的地址单元中去这样三个过程。格式串前的空白串或非空白串通常作为一个格式串与前一格式串之间的分隔符。格式串说明输入函数按何种格式读入及将数据转换的类型。format 中允许有多个格式串存在。如果一个格式串在地址参数表中应该有与其相对应的地址参数存在,则它是唯一的,否则多余的无格式串对应的地址参数将被忽略;反之,当应该有对应的地址参数的格式串在地址参数表中却并无对应的地址参数存在时,那是十分危险的。并且,格式串说明的数据格式和地

址参数说明的类型不相匹配时,例如,语句

```
int x;          /* x 为整型 */
scanf("%f",&x); /* 格式说明为浮点型 */
```

也是不允许的。否则,结果是不可预测的,很可能是灾难性的(例如在单步调试时调试被意外地结束,或死机等)。最后,format 中串与 va-list arglist 中的地址参数的排列顺序的对应性也是很重要的。

(1)空白串

它包括空格符、制表符(\t)或换行符(\n),常用于分隔两个格式串。这些字符也被称为空白字符,因为它们都不会在屏幕上有任何显示。如果一个格式串前面有空白串,则在键入数据时应严格地依顺序连续键入数据。对空白串如果键入了非空白字符,那是危险的,有可能使结果意外。格式输入函数将读入空白串字符,但是不将它们存储。实际上,地址参数表中没有也不应该有地址参数和它们对应。

两格式串中间允许有空白串,这时空白串相当于格式串之间的分隔符。当输入多个数据时,使用分隔符可以使输入明确,不易发生错误和产生误解。显然,对分隔符要求函数应能读入(或者说“理解”)但不将它们存储,而格式输入函数正是这样做的。为避免引起混乱和给操作者增加不必要的记忆负担,对同一程序你再好规定用同一种分隔符分隔格式串。

注意:当一个空白串用于分隔符时,假定空白串中只有一个空白字符,但你在输入时可以输入一个以上的空白字符,不会对结果产生错误。

对格式串 %c,它说明输入一个字符,其中包括空白字符。为了跳过空白字符,应用格式串 %s 等。

```
C>TYPE STDFOR17.C
#include "stdio.h"
#include "ctype.h"
#define PC1C2printf("c1=%c,",c1);\
    if (isalpha(c2))printf("c2=%c\n",c2);\
    else printf("c2=\\n\\n") /* 输出换行'\n' */

main() /* 可用调试表达式 —streams[0] 观察输入流 stdin 的当前的 stdin->level 和指针 stdin->
curp 所指字符的变化 */
{
    char c1,c2;
    printf("键入一个字母A后按回车键");
    scanf("%c",&c1); /* 执行此句后 stdin->level=1 */
    scanf("%c",&c2); /* 执行此句后 stdin->level=0 */
    PC1C2;
    printf("键入一个字母A后按回车键");
    scanf("%c",&c1); /* 执行此句后 stdin->level=1 */
    fflush(stdin); /* 执行此句后 stdin->level=0 */
    printf("再键入一个字母B后按回车键");
    scanf("%c",&c2); /* 执行此句后 stdin->level=1 */
    PC1C2;
}

C>TYPE STDFOR18.C
```

```

#include "stdio.h"
#include "ctype.h"
#define PC1C2printf("c1=%c,"c1);\
    if (isalpha(c2))printf("c2=%c\n",c2);\
    else printf("c2=\\n\\n") /* 输出换行 '\n' */

main()
{
    /* 如本程序中无 fflush(stdin); 语句,则键入调试表达式
    —streams[0] 后,显示
    —streams[0]; No type information
    即不能用它来观察输入流 stdin 的情况 */
    char c1,c2; /* 当用类型字符 %s 来输入单个字符时,空白字符被忽略 但空白字符后的非空白字
    符将为随后的格式串读取。还可以用键入
    A B
    AA B
    A B C
    三种方式测试。它们均会使 c1=A, C2=B,而对第三种,字母 C 还将自动赋给下
    一地址参数单元 */
    fflush(stdin); /* 注意,加上这一句就有 —streams[0] 的信息了,否则会出现
    —streams[0]; No type information 信息 */
    printf("键A 后按回车键");
    scanf("%s",&c1);
    printf("键B 再按回车键");
    scanf("%s",&c2);
    PC1C2; /* 输出 c1=A,c2=B */
    printf("先键c 后按回车键");
    scanf("%hs",&c1);
    printf("后键d 再按回车键");
    scanf("%hs",&c2);
    PC1C2; /* 输出 c1=c,c2=d */
    printf("键E 回车");
    scanf("%ls",&c1);
    printf("后键F 回车");
    scanf("%ls",&c2);
    PC1C2; /* 输出 c1=E,c2=F */
}

```

C>TYPE STDFOR19.C

```

#include "stdio.h"
main()
{
    int y;
    char a[10],c[10];
    y=scanf("%s\t\n%s",a,c); /* 格式串中有 '\t' 和 '\n',注意它们的写法 */
    printf("%d=%s=%s\n",y,a,c);
}

```

/* 当从键盘上输入了一个字母 W 后,接着又按了空格键或制表键 (Tab),最后键入 X1Y2Z3 六个字符后,程序输出:2=W=X1Y2Z3 */

C>TYPE STDFOR20.C

```
#include "stdio.h"
main()
{
    int x,y,u,v;
    printf("输入45ST88后回车,接着键入5 6后回车\n");
    printf("或输入45ST88 后回车,接着键入5 6后回车\n");
    printf("或输入45ST88 5 6后回车\n");
    scanf("%dST%d",&x,&y); /* 必须输入像 12ST98 那样的字段,这里串 ST 虽然是 format 的一部分,
                                但不是格式串的一部分.注意这种 严格的匹配关系.函数将扫描 ST
                                但不存储它.这里 format 的末尾有一个空格符,一般是不应该写的.如
                                果写了,则可以不输入 */
    scanf("%d %d",&u,&v); /* 此句被自动执行 */
    printf("%d,%d,%d,%d\n",x,y,u,v);
} /* 程序输出:45,88,5,6 */
```

(2)非空白串

它指除了百分号(%)以外所有可打印的 ASCII 码字符(即 isprint(字符)为非0时,字符的 ASCII 码值为 0x20 ~ 0x7e)。它们一般不能作为两个格式串间分隔符,但只有一种例外,即它处于两个数值格式串(如 %d%d)之间时有可能起分隔符的作用,因为这时非空白串虽被读取但可能被当作不可转换字符处理。即使在这种情况下,如果你在键入数据时又没有将相关的非空白串键入,则很危险,scanf()将终止。总而言之,一般不要将非空白串当分隔符使用。

C>TYPE STDFOR21.C

```
#include "stdio.h"
main()
{
    int y,x1,x2;
    char a[10],c[10];
    printf("请键入20t100后回车\n"); /* 一般应用这样的提示输入较好 */
    y=scanf("%st%s",&x1,&x2); /* 这是错误的,x1中为"20",x2中为"t1",y=1,主要是对数字不
                                能用%s */
    printf("%d %d %d\n",y,x1,x2);
    printf("请键入99tn-1005后回车\n");
    y=scanf("%dtn%d",&x1,&x2); /* 正确接收,x1=99,x2=-1005,y=2 */
                                /* 当然,多个数字常用逗号来分隔较好 */
    printf("%d,%d,%d\n",y,x1,x2);
    printf("请键入WWWWWtnXXXX后回车\n");
    y=scanf("%stn%s",a,c); /* 没有正确接收,因为不能区分 */
    printf("%d=%s=%s\n",y,a,c); /* 输出:1=WWWWWtnXXXX=X */
}
```

(3)格式串

格式串主要用于说明数据读入并转换的格式,一个格式串的一般形式是

%[*][width][F/N][h/l] type

format 中可以有多格式串存在,两个格式串中间允许用空白串或非空白串分隔。每一格式串一般和 format 后面紧跟的地址参数表中的一个地址参数相对应(但对 %* 格式串例外,它不允许有地址参数和它相对应)。

同一格式串内各选项之间自然也应匹配,不应出现矛盾。

1. %

format 必须以百分号 % 开头,随后的选项必须严格按顺序排列。

2. *

它是一个修饰符,也称【赋值抑制字符】,表示它所在的格式串要求你严格按规范输入一个值,但是在可变参数表中该格式串并无相对应的地址参数存在,因而输入的值不被存储,或者说被忽略。这里实际包括两层意思,一是在输入值时必须按格式串的定义输入相应类别的值。例如,规定输数值的不能输字符串等,否则会出问题的;其次,当在地址参数表中考虑了有它对应的一个地址参数,那是不应该的,而初学者往往会这样做。这是千万要注意的,因为那样也往往影响正确地接收数据。

使用赋值抑制字符的一个重要用处是,在输入多个数据时把它所在的格式串当作输入数据的分隔符。这样,操作者就可以突破必须用空白串或非空白串来作分隔符的限制,从而可以人为地自由地设置想要用的分隔符,而且这种分隔符既有规范性又有灵活性。某种意义上可以说,它相当于起到“分隔符变量”的作用。

它一般不能单独使用,如

```
scanf("%*c");
```

很少见,通常要和其它格式串连用。

使用它应很小心。在它之后跟格式串或新的赋值抑制字符时应验证其可靠性。

```
C>TYPE STDFOR22.C
```

```
#include "stdio.h"
```

```
main() /* %*d 类型 */
```

```
{
```

```
int t=4,y;
```

```
char m[10]="ABCD",m1[10]="ABC";
```

```
y=scanf("%*d",&t); /* y=0, 字段被扫描但不给地址参数 &t 赋值,因为和 &t 对应的格式子串并不存在 */
```

```
y=scanf("%s%*d",m); /* y=1, 如输入 aa 44, 只有 m 被赋值 "aa", 44 被忽略 */
```

```
y=scanf("%*d%s",m); /* y=1, 如输入 555 WWW 则 555 被忽略, m 被赋 "WWW" */
```

```
y=scanf("%d%*d%s",&t,m); /* y=2, 如输入 22 33 dd 则 33 被忽略, t=22, 而对 m 有 "dd" */
```

```
y=scanf("%s%*d%d",m,&t); /* y=2, 如输入 \ 22 44, 则 22 被忽略, m 为 "\ " 而 t=44 */
```

```
y=scanf("%s%*d%s",m,m1); /* y=2, 如输入 q -484 w, 则 m 中为 "q", m1 中为 "w", 而数字 -484 被忽略 */
```

```
y=scanf("%d%*d%s%s",&t,m,m1); /* y=3, 如输入 55 77 KKKK LLL, 则 t=55, m 为 "KKKK", m1 为 "LLL", 数字 77 被忽略 */
```

```
printf("y=%d, t=%d, s=%s, s1=%s\n",y,t,m,m1);
```

```
}/ * 可用下列调试表达式观察:
```

```
m1[0],10c;
```

```
m[0],10c;
```

```

    *(&t+1),cm;
    t,mc;
    &m1[0],p: DS,FFD8
    &m[0],p: DS,FFCE
    &t, DS,FFCC
    y:
    */

C>TYPE STDFOR23.C
#include "stdio.h"
main() /* %s 类型 */
{
    int t=4,y;
    char m[10]="ABCD",m1[10]="ABC";
    y=scanf("%s",m); /* y=0, 字段被扫描但不给下一地址参数赋值
                       因 m 并没有对应的格式子串存在 */
    y=scanf("%s%s",m,m1); /* y=2, 如输入 XX YY ZZ 则 m 中有串 "XX",
                           m1 中有串 "ZZ",而 YY 被忽略 */
    y=scanf("%s%s",m); /* y=1 如输入 cccc dddd 则 m 被赋值 dddd
                       而 cccc 被忽略 */
    y=scanf("%s%s%d",m,&t); /* y=2, 如输入 ss tt 245 则 m 中有串 "ss",
                           t=245, 而 tt 被忽略 */
    y=scanf("%d%s",&t,m); /* y=2, 如输入 22 / cccc 则 t=22,m 中为
                           cccc, 而 / 被忽略 */

    printf("y=%d, t=%d, s=%s, s1=%s\n",y,t,m,m1);
}/* 可用下列调试表达式观察:
    m1[0],10c;
    m[0],10c;
    &m1[0],p: DS,FFD8
    &m[0],p: DS,FFCE
    t;
    y;
    */

```

3. width

宽度指示符,它指出格式串最多可读入的字符数,是一个十进制正整数(比如常量 n)。

如果对指定宽度为 n 的格式串输入的字符个数小于 n,则输入的所有字符被读入,如能被转换则当地址参数所说明的变量类型和格式串中类型字符一致时转换后的数据被存储。

如果对指定宽度为 n 的格式串输入的字符个数大于 n,则输入的 n 个字符被读入,如能转换并存储则存储。剩余的字符将为下一个格式串处理时读入。

当接收数值时如要用宽度指示符限制接收字符,效果不好。如确要使用,应小心从事。

```

C>TYPE STDFOR24.C
#include "stdio.h"
main() /* 用调试表达式 wx,*pint,pint[0],6d,pint[0],6m,x,n 观察结果 */
{ /* 通过调试可以发现,当键入数不正确时,将影响下一个 scanf() 接收 */
    int n;
    int x,wx,*pint=&x; /* 从 wx 可见键入的数是当数值处理 */

```

```

wx=1;
printf("键入数-2244\n");
n=scanf("%3d",&x);          /* 注意,是 %d */
wx=x+5;
wx=1;
printf("键入数-2\n");
n=scanf("%3d",&x);          /* 此句被自动执行且 x=44,即无需键入而自动跳过 */
                                /* 数字 44 是上一 scanf() 调用后未读完的字符 */
wx=x+5;                        /* 但还不至于影响下面语句的执行 */
                                /*
    wx=1;
    printf("键入数-2.\n");    /* -2 后跟一个小数点 */
    n=scanf("%3d",&x);        /* 如执行此语句则会影响下面语句执行 */
    wx=x+5; */
wx=1;
printf("键入数-6后加两个空格\n");
n=scanf("%3d",&x);
wx=x+5;
wx=1;
printf("键入数+67\n");
n=scanf("%3d",&x);
wx=x+5;
    /* wx=1;
    printf("键入数2L\n");
    n=scanf("%3d",&x);        /* 此句也不能被正确接收 */
    wx=x+5;
    wx=1;
    printf("键入数2u\n");;    /* 此句也不能被正确接收 */
    n=scanf("%3d",&x);
    wx=x+5; */
wx=1;
printf("键入数012\n");
n=scanf("%3d",&x);    /* 结果 x=12,而不是八进制数 x=\012 */
wx=x+5;
    /* wx=1;
    printf("键入数\\11\n");
    n=scanf("%3d",&x);        /* 此句也不能被正确接收 */
    wx=x+5;
    wx=1;
    printf("键入数\\x55\n");    /* 此句也不能被正确接收 */
    n=scanf("%3d",&x);
    wx=x+5; */
printf("n=%d,wx=%d,pint=%p\n",n,wx,pint);
}

```

C>TYPE STDFOR25.C

```
#include "stdio.h"
```

```
main()/* 用调试表达式 s 或 t 去跟踪观察键入字符超过指定宽度时的处理 */
```



```

{
char s[2]="",t[3]="";
printf("键入字母A\n");
scanf("%2s",s);          /* s="A" t="" */
printf("键入字母BC\n");
scanf("%2s",s);          /* s="BC" t="" */
printf("键入DEFGHI\n");
scanf("%2s",s);          /* s="DE" t="" */
printf("键入jkl\n");
scanf("%2s",t);          自动执行,s="DEFG" t="FG" */
printf("键入88\n");
scanf("%2s",s);          /* 自动执行,s="HI" t="" */
printf("键入MNOP\n");
scanf("%2s %s",s,t);     /* s="MNOP" t="OP" */
}

```

C>TYPE STDFOR26.C

```

#include "stdio.h"
main()
{
char str[20],s[4];
printf("输入AS后回车\n");
scanf("%0s",s);          /* 宽度指示符为0是允许的 */
printf("s=%s",s);
scanf("%2s",str);
printf("str=%s\n",str);
} /* 程序输出:s= str=AS */

```

4. F|N|h|i

(1)F 或N

指针的缺省类型和存储模式相关,在小数据模式下(Tiny,Small,Medium)指针缺省值为near,值为16位,用数据寄存器的内容为段地址,所以一般只使用偏移量来表示;而在大数据模式下(Compact,Large,Huge),指针缺省值为far,值为32位,包括段地址和偏移量。对far或near型指针应用F或N说明,否则会出错的。但当指针缺省类型为far时可省写F,缺省类型为near时可省写N。总而言之,格式串中使用F或N时一定要和地址参数表中的指针类型完全一致。

注意:N不能在Huge存储模式中使用。

C>TYPE STDFOR27.C

```

#include "stdio.h"
#include "stdarg.h"
#include "alloc.h"
char far *str="串一";    /* 为了便于观察,让它为全局变量 */
char *ptr;              /* 不能使用 char *ptr=(char *)malloc(2); 这会产生 Illegal
                           initialization 错误 */

char t[]="ABCDEF";
int y=88;

```

```

main()
{
ptr=(char *)malloc(2);
printf("%%4Fs%%6s%%d%%2s\n");          /* 写出输入提示符 */
VSCAN("%%4Fs%%6s%%d%%2s\n",str,t,&y,ptr); /* 在 Small 模式下,远指针应加 F,否则会出错;而
对 ptr 因为缺省类型是 near,若加 F 便出错 */
printf("str=%s\n",str);
printf("t=%s\n",t);
printf("y=%d\n",y);
printf("ptr=%s\n",ptr);
}
VSCAN(char *format1)
{
va—list ptr;
va—start(ptr,format1); /* ptr 指向可变字符数组 (str,t,&y,ptr) 的头部 */
vscanf(format1,ptr);
va—end(ptr);
}
/* &y,p,&t[0],p ,str,p ,format1,p ,format1 ,y,t,str 等调试表达式观察。注意:作为结束字符,对最后
一个格式串的第三个字符不应是空白 字符 */

```

(2) b|l (即 H 或 L 的小写字母)

如果地址参数的类型为 short,则应在其对应的格式串中加上 h,如 %hd;如果为 long 或 double,则应加 l。

```

C>TYPE STDFOR28.C
#include "stdio.h"
main()
{
double x=3.14,*px=&x;
short y=2,*ps=&y;
printf("输入浮点数,整数\n");
scanf("%lf%*c%hd",&x,&y); /* 对 %lf 中的字母 l 不能少 */
printf("%lf,%hd\n",x,y);
}

```

5. type

类型字符,只用一个字母标记。字母大小写是有区别的,一般大写字母表示长整数或双精度数。

类型字符	可输入数	地址参数(指针)类型
	对数值	
d	十进制数	整型 (int * arg)
D	十进制数	长整型 (long * arg)
o	八进制数	整型 (int * arg)
O	八进制数	长整型 (long * arg)
i	八、十、十六进制数	整型 (int * arg)
I	八、十、十六进制数	长整型 (long * arg)

u	无符号十进制数	无符号整型 (unsigned int * arg)
U	无符号十进制数	无符号长整型 (unsigned long * arg)
x	十六进制数	整型 (int * arg)
X	十六进制数	长整型 (long * arg)
e	浮点数	浮点型 (float * arg)
E	浮点数	双精度型 (double * arg)
f	浮点数	浮点型 (float * arg)
g	浮点数	浮点型 (float * arg)
G	浮点数	双精度型 (double * arg)
	对字符	
c	字符	单个字符
		若指定宽度指示符,则字符数组
s	字符串	字符数组
[搜索范围]	字符串	字符数组 (下面单独说明)
	对指针	
n	无	指向整型的指针 (int * arg),成功读入的 (到 %c 为止) 字符数被存到此指针中
p	xxxx,xxxx	指向对象的远指针 (far *)
	xxxx	指向对象的近指针 (near *)
		xxxx 为十六进制数

注意:使用 scanf("%s",chname); 来接收字符串时最好用像 char chname[256]; 那样的数组来定义 chname,而不是用指针 char *chname。这是因为数组在说明后内存中即为字符串留出了真正的空间,而指针在说明后,如未用动态分配函数为它分配空间,那末内存并未给它预留空间。因此,在接收字符串时指针就不一定能指向自己的空间,很可能侵占别的内存空间,这是很危险的(参见《指针》一章)。

C>TYPE STDFOR29.C

```
#include "stdio.h"
main()          /* 本程序说明使用 %s 时应注意事项 */
{
    char ch[4],c='u';
    printf("输入ABCD1F\n");
    scanf("%5s",ch);    /* 定义接收字符串的数组应足够大,至少比串的长度大 1 否则可能会出意外。
                        */
    printf("ch=%s, c=0x%x\n",ch,c);
} /* 程序输出:ch=ABCD1, c=0x0
    超过宽度的字母 F 未被存储,串尾必有终止符 '\0',这是函数自动加上的 */
```

C>TYPE STDFOR30.C

```
#include <stdio.h>
main()
{
    char *ptr[]={ "ABCD", "efg" };
    char *p1="QWXYZ", *p2="qwxzabc";
    char * *str=p1;
    printf("ptr[0]=%s\n",ptr[0]);
    printf("输入%p\n",str);
}
```

```

scanf("%p",ptr);
printf("ptr[0]=%s\n",ptr[0]);
}
/* ptr[0]=ABCD
   输入01A1
   ptr[0]=QWXYZ      */
C>TYPE STDFOR31.C
#include <stdio.h>
main()      /* 本程序说明 %n 格式 */
{
int x,y;
char ch[5];
printf("%%d%%5c\n");
printf("输入234ABCDE\n");
scanf("%d%%5c%n",&x,ch,&y);
ch[5]='\0';
printf("x=%d,ch=%s,y=%d\n",x,ch,y);
}
/* %d%%5c
   输入234ABCDE
   x=234,ch=ABCDE,y=8      */

```

(1) 有关 [搜索范围] 的详细说明

严格地说,它是不存在的“类型字符”,因为它实际是一个字符串。当对一个读入的字符串,希望只选其中某些字符(即搜索字符)时就可不用类型字符s,而可改用这种由方括号对括起来的特殊的类型字符来对字符串进行转换。当它和类型字符s同时出现在一个格式串中时,它只能在s的前面出现。

搜索字符的集合是所有ASCII码字符(0x0~0x7f)。值得指出的是,空白字符等都可能为可转换字符,只要它们被包括在搜索范围内。若要将转义符写入搜索范围内,仍按转义符的书写规定进行。如

[ABCD\tEF]

中\t即是一个横向制表符,而不是表示字符\和t。

同一字符可以在搜索范围内出现一次以上,但多余的重复自然是不必要的。

scanf()函数读入相应的输入字符时,如果遇到一个搜索范围外的第一个字符时,搜索便停止。由于像空白字符等都可以是搜索范围内的字符,因此当这些字符在搜索范围内时,遇上它们搜索将不会停止。

搜索范围是一个字符串,由该串内的字符决定的可转换的那些字符称为搜索范围内的字符。它分两种情况:

<1> 当该串中的第一个字符不是^时

串中(显式或隐含)指出的所有字符均被认为是可转换的字符,否则是不可转换字符。例如,显式说明

[abcd]

和隐式说明

[a-d]

均表示字母 a ~ d 是可转换字母,除此之外的字符是不可转换字符。

<2> 当该串中的第一个字符是 ^ 时

串中 (显式或隐含) 指出的所有字符均被认为是不可转换的字符,否则是可转换字符,即它正好和上一种情况相反。例如,显式说明

[^abcd]

和隐式说明

[^a-d]

均表示字母 a ~ d 是不可转换字母,除此之外的字符均是可转换字符。

需要将串中多个字符进行隐式说明时可用连字符 - (即减号)。使用连字符的规则是,连字符前后必须有一个字符;连字符前面 (左边) 的字符的 ASCII 码值必须小于其后面 (右边) 的字符的 ASCII 码值。例如,

[0-9-a-z]

表示可转换字符是数字 0 ~ 9、减号和小写字母 a ~ z。一个搜索范围内可以出现多个显式或隐式说明。

```
C>TYPE STDFOR32.C
```

```
#include "stdio.h"
```

```
main()
```

```
{
```

```
char ch[20],cg[20];
```

```
printf("输入ACD1A\n");
```

```
scanf("%5[A-C]",ch);
```

```
printf("ch=%s\n",ch);
```

```
scanf("%5[^A-C]",cg);
```

```
printf("cg=%s\n",cg);
```

```
}
```

```
/* 输入ACD1A
```

```
ch=AC
```

```
cg=D1 */
```

```
C>TYPE STDFOR33.C
```

```
#include "stdio.h"
```

```
main()
```

```
{
```

```
char ch[20],cg[20];
```

```
printf("输入AB@9CD1A\n");
```

```
scanf("%5[9-A-C]",ch); /* 字母 B 将被转换,但尽量不要这样写,以免混淆 */
```

```
printf("ch=%s\n",ch);
```

```
scanf("%5[^A-C]",cg);
```

```
printf("cg=%s\n",cg);
```

```
}
```

```
/* 输入AB@9CD1A
```

```

    ch=AB@9C
    cg=D1                */
C>TYPE STDFOR34.C
#include "stdio.h"
main()                  /* 本程序说明使用 %s 时可在 %s 前加 [ 搜索范围 ] */
{                        /* 但在 %s 后加 [ 搜索范围 ] 则 [ 搜索范围 ] 便无效 */
    char ch[6];
    char c1='h',c='u';
    FILE *fp;
    fflush(stdin);
    printf("输入ABCDEF 后回车\n");
    scanf("%*c%5[A-CW-Z]s",ch);
    printf("ch=%s, c=0x%x\n",ch,c);
} /* 可用 streams[0],ch,c,c1 等调试表达式观察。程序输出:ch=BC,c=0x75 */

```

(2) 浮点转换 (%e、%E、%g 和 %G) 中,输入的浮点数的格式为

[+/-]ddddddddd[.]dddd[E|e][+|-]ddd

其中方括号内为任选项,ddd 代表十进制、八进制或十六进制数字。

(3) %d、%i、%o、%x、%D、%I、%O、%X、%c 和 %n 在允许使用指向字符、整数转换中,都允许使用无符号字符或整数。

(4) 输入字段

例如,在用键盘输入时,一次可能会输入多个字符,输入字符进入流缓冲区 (stdin) 内。这些字符可能和 format 对应,也可能并不和它对应,例如,输入字符个数可能多于要求输入数。格式输入函数将扫描流缓冲区,从缓冲区读入、转换和存储字符。【输入字段】可以由下列三种字符中的任一种字符构成:

- a. 下一个空白字符前的所有字符,但不包括该空白字符;
- b. 对当前格式串不能转换的第一个字符前的所有字符;
- c. 如果指定宽度指示符 n,则前 n 个字符。

(5) 格式输入函数遇到下列情况时停止对当前格式串的读入、转换和存储处理,转去处理下一输入字段:

- a. 在格式串百分号后出现赋值抑制符 (*),当前输入字段被读入但不被存储;
- b. 设宽度指示符 width 为 n,则当读了 n 个字符后;
- c. 在当前格式下遇到了一个不能转换的字符;
- d. 输入字段中的某一字符并没有在搜索范围内出现;未读字符被当作后一输入字段的首字符,或当作后来格式输入函数读操作的首字符。

(6) 在下列情况下,格式输入函数终止操作。

- a. 输入字段中的下一个字符与格式串中的相应空白字符相冲突;
- b. (文件) 输入字段中下一个字符为 EOF;
- c. 格式字符已处理完。

可变参数表参见格式输出函数。不过有一点要指明的是,这里参数是指向所使用的变量的指针。

31.2.2 函数返回值

格式输入函数返回成功地进行了扫描、转换和存储的输入字段个数,但对被扫描而未能存

储的输入字段不计算在内。

对函数 `sscanf()` 和 `vsscanf()` 从指定字符串末尾 (遇 '\0') 读入数据时, 函数返回 EOF; 同样, 对其余几个函数试图在文件末尾读入时也将返回 EOF。

如果没有一个字段被存储, 函数返回 0。

31.2.3 函数说明

—8 int —Cdecl `scanf(const char *format, ...)`;

它是一个常用的输入函数, 它的源流是 `stdin`, 通常指从键盘读入数据。读入数据存入可变地址参数指定的内存单元中。

—9 int —Cdecl `cscanf(const char *format, ...)`;

直接从控制台执行格式化输入, 将从控制台读入的数据存在格式串指定的地址参数 `&arg1, ..., &argn` 给定的地址中。如果函数试图在文件末尾读, 将返回 EOF。如果没有字段被存储, 返回 0。

—10 int —Cdecl `vscanf(const char *format, va-list arglist)`;

它的源流也是 `stdin`, 但读入数据存入由 `va-list arglist` 指定的地址参数指定的单元中。

—11 int —Cdecl `fscanf(FILE *stream, const char *format, ...)`;

跟 `scanf()` 不同的是, 它将用户打开的流 `stream` 作为源流, 而不是用 `stdin` 作源流。

C>TYPE STDFOR35.C

```
#include "stdio.h"
main()
{
    char string[80];
    double x;
    FILE *fp;
    fp=fopen("qq.c", "r"); /* 假定 QQ.C 中的内容为ABCDE1.23e-05
                                ABCDE 1.23e-05 */
    fscanf(fp, "%5s%lf", string, &x);
    printf("%s=%lf\n", string, x);
    fscanf(fp, "%s%g", string, &x);
    printf("%s=%g\n", string, x);
} /* 程序输出:ABCDE=0.000012
           ABCDE=1.23e-05 */
```

—12 int —Cdecl `vfscanf(FILE *stream, const char *format, va-list arglist)`;

跟 `vscanf()` 不同的是用 `stream` 代替 `stdin`。

—13 int —Cdecl `sscanf(const char *buffer, const char *format, ...)`;

跟 `scanf()` 不同的是用用户指定的缓冲区 `buffer` 作源流, 而不是用 `stdin`。

C>TYPE STDFOR36.C

```
#include "stdio.h"
main()
{
    char str[20];
    int k;
    sscanf("ABCD 12 34 5", "%s%d", str, &k);
```

```
printf("str=%s, k=%d\n",str,k);  
/* 程序输出, str=ABCD, k=12 */  
—14 int —Cdecl vsscanf (const char * buffer, const char * format,  
va—list arglist);  
跟 vscanf() 不同的是用缓冲区 buffer 代替了 stdin。
```


第三十二章 过程控制函数

32.1 进程管理函数

一 程序或进程终止

- 只是终止调用进程,不关闭文件与清缓冲区等 —1 —exit()
- 终止调用进程,关闭文件、清缓冲区和调用注册出口函数等 —2 exit()
- 立即调用 —exit() 异常中止程序运行,并印出非正常中止信息 —3 abort()
- 测试一个条件,如果它非 0,调用 abort() 中止程序运行 —4 assert()

二 注册终止函数

- 注册一个终止函数(程序结束前才运行已注册的终止函数,当注册有多个函数时,先注册的后执行) —5 atexit()

三 运行子程序(子进程覆盖父进程)

- 子进程执行后父进程不再被执行 —6 exec...()
- 子进程执行完后又继续执行父进程 —7 spawn... (P—OVERLAY,...)
—7 spawn... (P—WAIT,...)

四 直接执行一个 DOS 命令

- 相当于在 DOS 提示符下执行一个命令 —8 system()

五 非局部转移

- 保存当前任务状态的寄存器值供 longjmp() 调用 —9 setjmp()
- 执行非局部转移,程序在上一次调用 setjmp() 处开始继续执行 —10 longjmp()

—1 void —Cdecl —exit(int status);

函数终止调用进程,提供调用进程的出口状态 status。一般情况下, status 为 0 表示正常出口,非 0 值表示有错误发生。它与 exit() 不同之处是,它在终止程序时不关闭文件,不清除输出缓冲区,也不调用由 atexit() 注册的出口函数。

函数不返回值。适用于 UNIX 系统。

—2 void —Cdecl exit(int status);

本函数也终止调用进程。在退出调用函数之前,将所有文件关闭,缓冲区内正等待输出的内容被写完,所有已登记的“出口函数”(由 atexit() 注册的终止函数)被调用。

函数不返回值。适用于 UNIX 系统。 将父程序 PRO1.C 和子程序 PRO2.C 分别编译并键入 C>PRO1 PRO2 end 便可观察 exit() 产生的效果。每执行一次后,你将发现 MS

—DOS 的 F3 等键的重现功能无效。

```
C>TYPE PRO1.C
#include "string.h"
#include "stdio.h"
#include "process.h"
main(int argc, char * argv[])
{
    printf("父程序开始...\n");
    if(argc<3)
        {printf("父程序参数太少! \n");exit(1);}
    spawnl(P-WAIT,argv[1],NULL);
    printf("父程序继续...\n");
    if( strcmp(argv[2],"end")==0)exit(0);
    printf("父程序结束! \n");
}
/*
```

```
C>TYPE PRO2.C
#include "string.h"
#include "stdio.h"
#include "process.h"
main()
{
    int ch;
    printf("子程序...输出一个数(0~9):\n");
    ch=getche();
    if(ch != '0') printf("继续父程序...\n");
    else{
        printf("子程序结束! \n");exit(1);
    }
    printf(".....\n");
}
*/
```

—3 void —Cdecl abort(void);

异常终止一个进程。本函数被调用时,程序将自动调用函数 —exit() 而立即终止程序执行,并印出非正常终止执行信息“ Abnormal program termination ”到预定义的流 stderr(一般指屏幕)中。

适用于 UNIX 系统。

—4 #if ! defined(NDEBUG)

```
define assert(p)if(! (p)) {fprintf(stderr,\
    "Assertion failed: %s, file %s, line %d\n",\
    #p, —FILE—, —LINE—);abort();}

# else
# defined assert(p)
# endif
```

assert(int test); 是一个宏,它测试一个条件test,如果它非 0,则调用abort()函数中止

程序执行,并输出

Assertion failed: 条件,被终止的源文件名,终止行号

Abnormal program termination

这里 #p 是将条件原样印出;源文件名由标准的五个预定义宏之一——FILE——指出;另一个预定义宏——LINE——则指出在源程序哪一行上终止。stderr 是在 STDIO.H 中预定义的标准 I/O 流,将错误信息输出至屏幕。至于“Abnormal program termination”是由函数 abort() 调用产生的,表明程序非正常终止。

如果在语句 #include "assert.h" 之前写有一个 #define NDEBUG 语句(或称一条指令)则源程序中的 assert() 不起任何作用,或者说只起一个注释作用。当然,必要时你将前面的 #define NDEBUG 指令取消,它立刻起作用。要注意的是, #defined NDEBUG 指令一定要在 #include "assert.h" 语句之前使用,否则该指令不起作用。

不难看出,在调试源程序时在某些地方插入 assert() 是有用的。

```
C>TYPE PRO3.C
#include "assert.h"
#include "stdio.h"
struct ZP {
    int key;
    int value;
};
void add—ZP(struct ZP * ptr)
{
    assert (ptr != NULL);
    printf("如果程序不终止,则此行被打印\n");
}
main()
{
    struct ZP g;
    g.key=1;
    printf("开始...\n");
    add—ZP(&g);
    printf("继续...\n");
    add—ZP(NULL);
    printf("结束!\n");
}
/* 程序输出:
开始...
如果程序不终止,则此行被打印
继续...
```

Assertion failed: ptr != 0, file l130.c, line 9

Abnormal program termination

如用工具软件:C>CPP PRO3.C 则生成 PRO3.I,PRO3.I 中有

```
pro3.c 7: void add—ZP(struct ZP * ptr)
pro3.c 8: {
pro3.c 9: if(! (ptr != 0)){fprintf((&—streams[2]), "
```

```

        Assertion failed: %s, file %s, line %d\n",
        ptr 1 = 0", "pro3.c", 9); abort(); } ;
pro3.c 10: printf("如果程序不终止,则此行被打印\n");
pro3.c 11: }

```

可帮助你明白宏 `assert()` 扩展后的结果。*/

```
—5 int —Cdecl atexit(atexit—t func);
```

它【注册, registration, 或称登记】由 `func` 所指的【终止函数】(或【退出(`exit`)函数】)名。`atexit—t` 在 `stdlib.h` 中定义为

```
typedef void —Cdecl (* atexit—t)(void);
```

即是一个不接受任何参数,不返回任何值的函数指针。

终止函数是在程序执行结束前被调用执行。当有多个终止函数注册时,按先注册后执行(即后进先出)的顺序执行。每次对 `atexit()` 调用只能注册一个终止函数。虽然在一个程序中可以注册多于 32 个的终止函数,但只有前 32 个有效,或者说 Turbo C 最多允许在一个程序中注册 32 个函数。

如果调用成功,返回 0,否则返回非 0 值,表明没有足够空间注册函数。

C>TYPE PRO4.C

```

#include "stdlib.h" /* exit() 和 _exit() 在此标头文件中也有定义 */
#include "stdio.h"
void exit1()
{printf("执行终止函数 一1\n");}
void exit2()
{printf("执行终止函数 一2\n");}
main()
{
    int ch;
    atexit(exit1);
    atexit(exit2);
    printf("请输入一个数(0~9):\n");
    ch=getche();
    printf("\n 试验开始...\n");
    if(ch != '0')exit(0);/* 如这两句没有,程序执行结束前执行注册函数 */
    else _exit(0);
    /* 注册函数一般应放在源程序开头,贯彻先注册后应用的原则,否则可能无效 */
    /* atexit(exit1);
       atexit(exit2); */
    printf("程序执行结束!\n");
}
/* 输出:请输入一个数(0~9);
      8
      试验开始...
      执行终止函数 一2
      执行终止函数 一1
或,
      请输入一个数(0~9);

```

试验开始...

*/

【进程, process】是并发程序出现后出现的一个重要概念。它是按照由进程说明中定义的模式来启动一组顺序操作的动作的执行,被启动的进程的执行可以和启动程序并发执行。这里也可简单地把 DOS 执行一个程序称为一进程。

词头 exec 可理解为英文单词 execute,意为“执行、作完”。

```
—61 int —Cdecl execl(char *path, char *arg0, ...);
—62 int —Cdecl execlp(char *path, char *arg0, ...);
—63 int —Cdecl execlp(char *path, char *arg0, ...);
—64 int —Cdecl execlpe(char *path, char *arg0, ...);
—65 int —Cdecl execv(char *path, char *argv[]);
—66 int —Cdecl execve(char *path, char *argv[], char **env);
—67 int —Cdecl execvp(char *path, char *argv[]);
—68 int —Cdecl execlpe(char *path, char *argv[], char **env);
```

exec... 函数族是用来加载并运行子进程的函数集合。

当 exec... 调用成功时,子进程便覆盖父进程,必须有足够的内存空间用于加载和执行子进程。一个子进程可以调用另一个子进程,这种调用的嵌套层数只受内存容量和程序大小的限制。2.01-2 以前的 NOVELL 网系统软件不支持使用 exec...() 的 DOS 调用。

C>TYPE PRO5.C

```
main()
{
    printf("执行子进程...\n");
    printf("子进程结束\n");
}
```

C>TYPE PRO6.C

```
#include "process.h"
main()
{
    printf("父进程开始执行...\n");
    execl("c:\\tc\\PRO5.EXE", "");          /* 传送空参数 ("") */
    printf("父进程结束\n");                  /* 由于覆盖,此句将不被执行 */
}
/* 输出: 父进程开始执行...
        执行子进程...
        子进程结束 */
*/
```

path 是被调用子进程的文件名。若文件没有扩展名或句点,将搜索给定的文件名,如果没有找到,在文件名后加上 .EXE 扩展名后再进行搜索;若文件名最后有句点但其后并无字母时按无扩展名文件处理;若文件有扩展名,则按实际文件名搜索。搜索是用 MS-DOS 标准算法进行的。

这里要注意的是,当子进程名无效时,含该子进程名的 exec?() 函数不被执行,程序自动

执行语句的下一语句;其次,当有多个文件基本名相同而扩展名不同时你必须指明扩展名;最后,你不能将原为 .COM 的文件直接用 MS-DOS 的换名命令换成像 .EXE 那样的文件名,反之亦然。否则会产生不可预料的结果。

1. 关于对 EXEC 库函数中执行子程序名的规定

在实际使用中,被执行的子程序其内容可为 .COM 或 .EXE 两种格式,但是有时它们的文件扩展名和内容有可能并不是一一对应的,换句话说,它们有可能用别的扩展名。例如, DOS 3.30 中的 BACKUP.COM 和 RESTORE.COM 这两个文件的真实内容是 .EXE 格式的。若以这种名不符实的文件名为 EXEC 的执行子程序名将会发生什么情况?下面我们查找字符串实用程序 GREP.COM 说明之。

```
C>COPY GREP.COM GG.COM
C>TYPE MY.C
#include "process.h"
main()
{
    execl("gg.com", "", "main", "my.c", ""); /* 注意 argv[0] 不要漏写 */
}
```

在集成环境缺省条件下编译连接后生成 MY.EXE,然后执行该程序。

```
C>MY
File MY.C:
main()
    execl("gg.com", "", "main", "my.c", "");
```

如将 MY.C 中 GG.COM 改成 GG.Cxx,且用 DOS 换名命令 rename 将文件名 GG.COM 同时改成 gg.Cxx,这里 xx 是文件名的任意两个合法字符,则也会得到相同结果。

但是,如果将 GG.COM 换成非 .Cxx 扩展名的文件名,则发现不能正确执行。特别注意,将它改成文件名带句点而无附加名时也不能正确执行。无扩展名时同样不能正确执行。

同样可以验证,任何实际内容为 .EXE 形式的文件如改用扩展名 .Cxx,则也不能正确执行,而改成其它形式的扩展名则能执行。

这是 Turbo C 设置的一种限制扩展名使用的方法,以免误用这两种占用内存不同的文件造成不可预料的后果。

这一控制是在运行库文件中进行的。例如,对 CS.LIB 可用 debug 按下法进行修改,就可取消这种限制。

```
C>DEBUG CS.LIB
-U A36C
XXXX:A36C      NOP
XXXX:A36D      NOP
XXXX:A36E      NOP
-U A396
XXXX:A396      JNZ A3D4
-W
-Q
```

· 建议你在万不得已时才这样做,否则改用换扩展名的方法解决较好。

2. 函数名中后缀的含义

加在函数名中字符 `exec` 后面的后缀 `p`、`l`、`v` 和 `e` 表示隐含有某种功能:

- `p` 表示将在由 DOS 环境变量 `PATH` 指定的目录中搜索指定的子进程名; 在没有 `p` 后缀时, 只在根目录和当前目录下搜索。因此, 它是描述路径的后缀。
- `l` 表示参数中指针 `arg0, ...` 等按独立参数传送。一般说来, 当知道要传送的参数个数时才使用 `l` 后缀。
- `v` 当参数中含有指针数组 `argv[]` 时才使用它。一般说来, 当要传送的参数可变时才用后缀 `v`。
- `e` 当参数中含有用于改变子进程的环境参数 `env` 时才用它。如果不用后缀 `e`, 则子进程继承父进程的环境。因此它是描述环境的后缀。关于环境的概念 (参见《程序结构和主函数》一章)。

这四个后缀允许按规定的方式组合使用, 其中 `v` 和 `l` 只能用两者中的一个, 而 `e` 和 `p` 是任选的。

`exec...` 最少也必须传送一个参数 (`arg0` 或 `argv[0]`) 给子进程, 可以传送多个独立的参数 `arg0`、`arg1`、...、`argn`, `NULL`。`NULL` 一定是最后一个, 它在 `STDIO.H` 中有定义, 表示参数表的结束。Turbo C 约定第 0 个参数即参数 `arg0` 或 `argv[0]` 只是参数 `path` 的一个拷贝, 因此, 即使对它赋予不同的值也不会产生错误。在 MS-DOS 3.x 版本中 `path` 还可在子进程中使用。但是, 在早期的 DOS 版本中, 子进程是不能使用所传送的第 0 个参数的值的, 或者说是子进程是不能使用 `path` 的。

```
C>TYPE PRO7.C
#include "stdio.h"
main(int argc, char * argv[])
{
    int k;
    printf("执行子进程...\n");
    for(k=0; k<argc; k++)
        printf("argv[%d]: %s\n", k, argv[k]);
    printf("子进程结束\n");
}

C>TYPE PRO8.C
#include "process.h"
#include "stdio.h"
main()
{
    int status;
    printf("父进程开始执行...\n");
    status=execl("c:\\tc\\PRO7.EXE", "arg0", "arg1", "arg2", NULL);
    printf("如执行子进程发生错误, 打印: %d\n", status);
}

/* 输出: 父进程开始执行...
      执行子进程...
      argv[0]: C:\TC\PRO7.EXE
      argv[1]: arg1
      argv[2]: arg2
```

子进程结束

*/

应当指出,第 0 个参数写成 NULL 和 "" 是不同的,这对某些程序是需要的。例如,对于 MS-DOS 的程序 MEM.EXE 你不能给很多参数,如将

```
execl("c:\\dos\\mem.EXE",NULL);
```

写成

```
execl("c:\\dos\\mem.EXE","arg0","arg1",NULL);
```

将产生“太多参数出现的错误 (Too many parameters)”,因为子进程 mem.exe 对此是不能接收的。

```
C>TYPE PRO9.C
```

```
#include "stdio.h"
```

```
#include "process.h"
```

```
main()
```

```
{
```

```
printf("子剩余空间...\n");
```

```
execl("c:\\dos\\mem.EXE",NULL);
```

```
}
```

```
C>TYPE PRO10.C
```

```
#include "process.h"
```

```
#include "stdio.h"
```

```
main()
```

```
{
```

```
static int a[1000];
```

```
printf("父进程开始执行...\n");
```

```
execl("c:\\dos\\mem.EXE",NULL);/* mem.exe 为 DOS 提供的求剩余空间的程序 */
```

```
printf("如执行子进程发生错误,打印本语句\n");
```

```
}
```

```
/* 程序输出:
```

```
父进程开始执行...
```

```
655360 bytes total conventional memory
```

```
655360 bytes available to MS-DOS
```

```
164224 largest executable program size
```

```
3407872 bytes total contiguous extended memory
```

```
3407872 bytes available contiguous extended memory
```

若将语句 execl("c:\\dos\\mem.EXE",NULL); 改成

```
execl("c:\\tc\\PRO9.EXE",NULL);
```

则有:

```
父进程开始执行...
```

```
子剩余空间...
```

```
655360 bytes total conventional memory
```

```
655360 bytes available to MS-DOS
```

```
164208 largest executable program size
```

```
3407872 bytes total contiguous extended memory
```


*/

注意:参数 `arg0`、`arg1`、...、`argn` (或 `argv[0]`、`argv[1]`、...、`argv[n]`) 的字符串长度之和不应大于 128 字节。终结的 `NULL` 的字符长度不必考虑。

当使用 `e` 后缀时,通过参数 `env` (即 `environment`) 传送一个新的 DOS 环境设置表。该参数是一个字符串指针数组,每个数组元素 (指针) 指向的字符串必须具有以下形式:

环境变量名 = 值

数组的最后一个指针必须是 `NULL`。如果它的第一个指针是 `NULL`,则子进程继承父进程的环境设置,或者说两者具有相同的环境。

```
C>TYPE PRO11.C
#include "stdio.h"
#include "process.h"
#include "stdlib.h"
main(int argc,char *argv[])
{
    char *env;
    env=getenv("PATH");    /* 从环境中取与环境变量名 PATH 相对应的入口项 */
    printf("子进程环境=%s\n",env);
    printf("argc=%d argv[0]:%s argv[1]:",argc,argv[0]);
    printf(argv[1]);        /* 未给出输出格式而直接输出 */
    printf("\nargv[2]:%d argv[3]:%s\n",atoi(argv[2]),argv[3]);
}
```

```
C>TYPE PRO12.C
#include "process.h"
#include "stdio.h"
#include "stdlib.h"
main()
{
    char *path;
    char *arg[]={ "default","OK!","100" };
    char *e[]={ "PATH=C:\\TC;C:\\ACAD",NULL };
    char **env=e;
    path=getenv("PATH");
    printf("父进程开始执行,环境是:%s\n",path);
    execve("c:\\tc\\PRO11.EXE",arg,env);
    printf("如执行子进程发生错误,打印本语句\n");
}
/* 输出:父进程开始执行,环境是:C:\\DTP;C:\\DOS;C:\\XPC\\SYS
子进程环境=C:\\TC;C:\\ACAD
argc=4 argv[0]:C:\\TC\\PRO11.EXE argv[1]:OK!
argv[2]:100 argv[3]:PATH=C:\\TC;C:\\ACAD
```

注意:当子进程结束后,环境依然为 `C:\\DTP;C:\\DOS;C:\\XPC\\SYS`,这可在 MS-DOS 命令行上

打入命令

```
C>PATH
```

检查核实。

*/

当一个 exec... 函数被执行时,原已打开的文件在子进程中仍然打开。

如果调用成功,它们不返回任何值;但当调用失败时,返回 -1,并置全局量 errno 为

2 ENOENT (路径或文件名没找到)

4 EMFILE (打开的文件太多)

5 EACCES (无此权限)

8 ENOMEM (无足够内存空间)

20 E2BIG (参数表太长)

21 ENOEXEC (Exec 格式错误)

与 exec... 函数族形式稍有不同的函数族 spawn... 在功能上包含了 exec... 函数族的功能,且作了功能扩充(参考 DOS 中断 INT 21H 的功能 4BH 的 EXEC 装入/执行例程)。两者在返回结果上稍有差异。注意,对后缀等说明同 exec...。

英文单词 spawn 有“大量生产、产卵、引起”之意。

```
—71 int —Cdecl spawnl(int mode, char * path, char * arg0,...);
—72 int —Cdecl spawnle(int mode, char * path, char * arg0,...);
—73 int —Cdecl spawnlp(int mode, char * path, char * arg0,...);
—74 int —Cdecl spawnlpe(int mode, char * path, char * arg0,...);
—75 int —Cdecl spawnv(int mode, char * path, char * argv[]);
—76 int —Cdecl spawnve(int mode, char * path, char * argv[], char * * env);
—77 int —Cdecl spawnvp(int mode, char * path, char * argv[]);
—78 int —Cdecl spawnvpe(int mode, char * path, char * argv[], char * * env);
```

不难看出,spawn... 比 exec... 多了一个参数 mode,它在 process.h 中定义为三种情况:

```
#define P—WAIT 0 /* 执行子进程时父进程暂时“挂起”,当子进程执行 */
/* 结束,从执行子进程下一语句开始继续执行父进程 */
#define P—NOWAIT 1 /* 父进程和子进程同时执行,但 Turbo C 目前尚未提供 */
#define P—OVERLAY 2 /* 子进程执行时覆盖了父进程原来的存储位置,子进 */
/* 程执行后父进程不再被执行,这和 exec... 调用相同 */
```

在执行成功时,返回值为子进程退出状态(如值为 0 表示子进程正常退出)。如返回 -1,表明调用失败,像 exec... 那样置 errno 为适当值。

把上面的 PRO12.C 的最后两句用 spawn... 来表示,结果依模式 mode 的值不同而有所不同。2.01—2 以前的 NOVELL 网系统软件不支持使用 spawn...() 的 DOS 调用。

```
C>TYPE PRO13.C
#include "process.h"
#include "stdio.h"
#include "stdlib.h"
main()
{
char * path;
char * arg[]={"default","OK!","100"};
char * env[]={"PATH=C:\\TC;C:\\ACAD",NULL}; /* 此句也可稍作修改 */
```

```

/* char **env=e; 上句修改后将此句取消 */
path=getenv("PATH");
printf("父进程开始执行,环境是:%s\n",path);
spawnve(P-WAIT,"c:\\tc\\PRO11.EXE",arg,env);
printf("执行子进程过后,便打印本语句.\n");
spawnve(P-WAIT,"c:\\tc\\PRO11.EXE",arg,env); /* 再调用一次 */
printf("父进程结束!\n");
}
/* 输出:父进程开始执行,环境是:C:\DTP\C:\DOS\C:\LXPC\SYS
子进程环境=C:\TC\C\ACAD
argc=4 argv[0]:C:\TC\PRO11.EXE argv[1]:OK!
argv[2]:100 argv[3]:PATH=C:\TC\C\ACAD
执行子进程过后,便打印本语句。
子进程环境=C:\TC\C\ACAD
argc=4 argv[0]:C:\TC\PRO11.EXE argv[1]:OK!
argv[2]:100 argv[3]:PATH=C:\TC\C\ACAD
父进程结束!

*/

```

—8 int —Cdecl system(const char *command);

发出一个 MS-DOS 命令 command,即相当于在 DOS 提示符下键入 COMMAND 命令一样。system() 通过 MS-DOS 的 COMMAND.COM 文件执行该命令。由于环境变量可通过 COMSPEC 寻找 COMMAND.COM 文件,所以 COMMAND.COM 可以不在当前目录中。函数返回给定命令完成后 COMMAND.COM 的出口状态。2.01-2 以前的 NOVELL 网系统软件不支持使用 system() 的 DOS 调用。

批处理文件 *.BAT 也可用 system() 来写,结果生成 *.EXE 文件。

```

C>TYPE PRO14.C
#include "stdlib.h"
main()
{
system("dir/w *.c>q1.out"); /* 利用 DOS 的重定向功能将列 */
}                             /* 出目录装入到文件 q1.out 中 */

C>TYPE PRO15.C
#include "process.h"
main() /* 将 *.BAT 文件中语句用 system() 来写 */
{
system("echo off");
system("ws");
}

```

—9 int —Cdecl setjmp(jmp_buf jmpb);

它用缓冲区 jmpb 保存诸如寄存器值这样上下文相关的信息(程序任务状态),以便函数 longjmp() 使用。jmpb 为一结构类型,在 setjmp.h 中定义为:

```

typedef struct {
unsigned j-sp; /* 栈指针 */
unsigned j-ss; /* 栈段指针 */
unsigned j-flag; /* 标志寄存器 */

```

```

unsigned j—cs;      /* 代码段指针 */
unsigned j—ip;      /* 指令指针 */
unsigned j—bp;      /* 基址指针,辅助指针,用于取栈中参数 */
unsigned j—di;      /* 目的索引 */
unsigned j—es;      /* 附加段指针 */
unsigned j—si;      /* 源索引 */
unsigned j—ds;      /* 数据段指针 */
} jmp—buf[1];      /* 1 是数字 */

```

当执行 longjmp() 时, longjmp() 的第一个参数就是 setjmp() 的缓冲区。

初次调用时它返回 0。

—10 void —Cdecl longjmp(jmp—buf jmpb, int retval);

执行非局部转移。它使程序在上一次调用 setjmp() 处开始继续执行。注意: 程序中有 setjmp() 而无 longjmp() 是可以的, 反过来则不行! 而且, longjmp() 中第一个参数名 jmpb 应和 setjmp() 的参数名相同, 否则会产生不可预料的结果。而且, jmpb 必须在调用 longjmp() 之前先由 setjmp() 设置好。

longjmp() 是根据 retval 重新设置堆栈来操作的。retval 重置 setjmp() 的返回值 (初次 setjmp() 一定返回 0), 注意: 该值不能为 0! 程序根据这个值可以确定它是从哪里转跳来的。

longjmp() 不返回 0, 如果 0 传给 retval, 它将返回 1。

longjmp() 最常用的方法是当一个灾难性的错误发生时, 从嵌套很深的子程序中返回。

C>TYPE PRO16.C

```

#include "stdio.h"
#include "setjmp.h"
#define PDASK printf("j—sp=0x%x\n", jbuf[1].j—sp); \
               printf("j—ss=0x%x\n", jbuf[1].j—ss); \
               printf("j—flag=0x%x\n", jbuf[1].j—flag); \
               printf("j—cs=0x%x\n", jbuf[1].j—cs); \
               printf("j—ip=0x%x\n", jbuf[1].j—ip); \
               printf("j—bp=0x%x\n", jbuf[1].j—bp); \
               printf("j—di=0x%x\n", jbuf[1].j—di); \
               printf("j—es=0x%x\n", jbuf[1].j—es); \
               printf("j—si=0x%x\n", jbuf[1].j—si); \
               printf("j—ds=0x%x\n", jbuf[1].j—ds)

int value;
jmp—buf jbuf;
main()
{
    int j=0, k=0;
    printf("j=%d k=%d\n", j, k);
    value=setjmp(jbuf); /* 调用 sub() 时恢复原先任务, 从此句开始往下执行 */
    PDASK;
    k++;
    sub—1();
    printf("j=%d k=%d value=%d\n", j, k, value);
    k++;
    if(value != 0){

```

```

        printf("longjmp()有值,%d\n",value);
        exit(value);    /* 如不退出,将出现无穷循环打印 */
    }
    printf("即将调用函数sub...\n");
    k++;
    sub=2();
    sub();
    printf("程序结束!\n");/* 此句由于执行 sub() 时有 exit() 函数调用而不执行 */
}
sub()
{
    longjmp(jbuf,100); /* 或即 longjmp(jbuf,0x64); */
}
sub=1()
{printf("函数一\n");}
sub=2()
{printf("函数二\n");}

```

/* 输出: 在 Small 存储模式下

```

j=0 k=0
j-sp=0x0
j-ss=0x0
j-flag=0x0
j-cs=0x0
j-ip=0x0
j-bp=0x0
j-di=0x0
j-es=0x0
j-si=0x0
j-ds=0x0

```

函数一

```
j=0 k=1 value=0
```

即将调用函数sub...

函数二

```
j-sp=0x64 j-sp=0x64
```

```

j-ss=0x0
j-flag=0x0
j-cs=0x0
j-ip=0x0
j-bp=0x0
j-di=0x0
j-es=0x0
j-si=0x0
j-ds=0x0

```

函数一

```
j=0 k=1 value=100
```

longjmp()有值:100

在 Huge 存储模式下

```

j=0 k=0
j-sp=0x0
j-ss=0x3d6a
j-flag=0x6425
j-cs=0x6b20
j-ip=0x253d
j-bp=0xa64
j-di=0x6a00
j-es=0x735f
j-si=0x3d70
j-ds=0x7830

```

函数一

```
j=0 k=1 value=0
```

即将调用函数sub...

函数二

←retval 值

```

j-ss=0x3d6a
j-flag=0x6425
j-cs=0x6b20
j-ip=0x253d
j-bp=0xa64
j-di=0x6a00
j-es=0x735f
j-si=0x3d70
j-ds=0x7830

```

函数一

```
j=0 k=1 value=100
```

longjmp()有值:100

*/

将上述程序 PRO16.C 等稍作修改后得 PRO17.C ,从中你能更好地领略其含义。

```
C>TYPE PRO17.C
#include "stdio.h"
#include "setjmp.h"
#define PDASK printf("j-sp=0x%x\n",jbuf[1].j-sp)
int value;
jmp—buf jbuf;
main()
{
    int j=0,k=0;
    printf("j=%d k=%d\n",j,k);
    PDASK;
    k++;
    sub—1();
    printf("j=%d k=%d value=%d\n",j,k,value);
    k++;
    if(value != 0){
        printf("longjmp()有值:%d\n",value);
        exit(value);
    }
    printf("即将调用函数sub...\n");
    k++;
    sub—2();
    sub();
    printf("程序结束!\n");
}
sub()
{
    longjmp(jbuf,100);
}
sub—1()
{
    printf("前:");PDASK;
    value=setjmp(jbuf);
    printf("后:");PDASK;
    printf("函数—1\n");
}
sub—2()
{printf("函数—2\n");}
/* 输出:
    j=0 k=0
    j-sp=0x0
    前:;j-sp=0x0
    后:;j-sp=0x0
    函数—1
```

```

j=0 k=1 value=0
即将调用函数sub...
函数—2
后: j-sp=0x64
函数—1
程序结束!
*/

```

32.2 TC.EXE 文件结构剖析

利用下列程序可以印出二进制文件 L5.EXE 的全部字节。

```

C>TYPE PRO18.C
#include "stdio.h"
#include "io.h"
main()
{
FILE *fp;
unsigned char c;
long fsize;
int k;
fp=fopen("l5.exe","rb");
fsize=filelength(fp->fd);
printf("\n");
for(k=0;k<fsize;k++)
{
c=fgetc(fp);
printf("%x ",c);
}
}

```

.EXE 文件是 DOS 中的二进制代码的可执行文件,它可使用多个段,这些段可以实现再定位。某些 .EXE 文件可以通过 DOS 的 EXE2BIN 文件的执行变成 .COM 文件。它的结构参见图 32-1。下面以 TC.EXE 为例说明之。

先在 DOS 提示符下进行下述操作:

```

C>DIR TC.EXE
看到文件长度为 290249 字节,或 0x46DC9 字节。
C>COPY TC.EXE MY
C>DEBUG MY          /* 未重定位时的情况 */
-D
1A26:0100  4D 5A C9 01 37 02 F4 0C — 40 03 95 05 FF FF 9A 47
1A26:0110  64 19 00 00 00 00 00 00 — 22 00 00 00 01 00 FB 20
1A26:0120  72 6A 01 00 00 00 E8 00 — 00 00 A3 00 77 3B 9D 00
-D34E0
1A26:34E0  1C 37 60 3E 1C 37 99 3E — 1C 37 E2 44 1C 37 FA 44
1A26:34F0  1C 37 00 00 00 00 00 00 — 00 00 00 00 00 00 00 00

```

以上列出了文件标头的一些数据,下面说明之(除非特别说明,否则数字均为 16 进制值)。

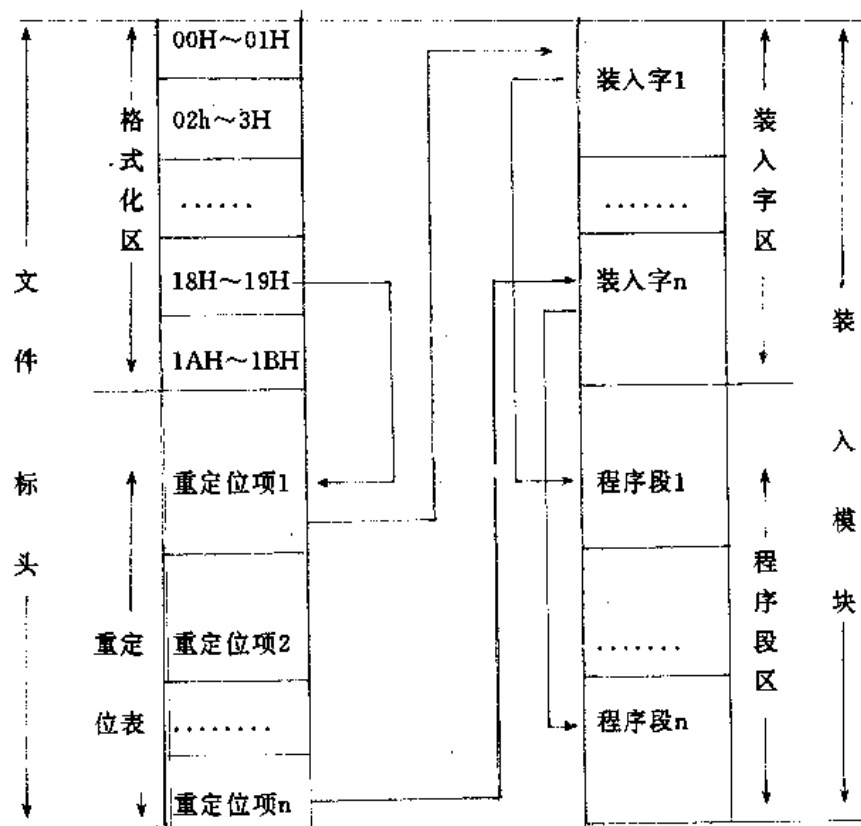


图 32-1 .EXE 文件的结构示意图

— 格式化区

格式化区共有 28 个字节,每两个字节为一项(注意:低位在前),也称偏移量。连字符后为相应的值(低字节在前)。

—1 00H 01H — 4D 5A

它是 LINK 程序的签字,也称 .EXE 特征字(4D 5A 即“MZ”),表明这个文件是一个有效的 .EXE 文件。如无此标识,运行结果将是不可预料的。用 debug 将其装入后,也不会对它执行重定位操作。

注意:对有些 .EXE 文件特征字可能是“ZM”,即 5A 4D。

—2 02H 03H — C9 01

$0x46DC9 \% 0x200 = 0x1C9$,即被装载模块映象的容量除以 $0x200 (=512)$ 后的余数,或即最后扇区中文件实际使用的字节数 ($\leq x200$)。

—3 04H 05H — 37 02

$0x46DC9 / 0x200 + 1 = 0x237$,包括文件标头在内的文件共占用页数(每页 $0x200$ 字节),即等于文件长度除以 512 的商加 1。

—4 06H 07H — F4 0C

重定位表中重定位项的项数。

—5 08H 09H — 40 03

文件标头所占节数。因 1 节 = 16 字节, 所以文件标头共占 $0x340 * 0x10 = 0x3400$ 字节。用于确定被装载模块在文件中的起始位置。

—6 0AH 0BH — 95 05

从装入程序末尾往前计算所需最小内存字节数 (以节为单位) 为 $0x595$ 节。对有些程序也可 0。

—7 0CH 0DH — FF FF

从装入程序末尾往前计算所需最大内存字节数 (以节为单位) 为 $FFFFH$ 节 (已是最大值)。

0AH ~ 0DH 主要是在 DOS 分配内存时对被装入程序进行检查, 若模块装载后, 其上方至 DOS 命令处理程序的暂存区之间的空间不满足要求, 则装载会失败, 显示内存不足信息。若 0AH ~ 0DH 均为零, 则程序装入可分配内存的尽可能高处。模块一般装在可用内存最低端。

—8 0EH 0FH — 9A 47

SS=479A, 装入模块堆栈段的段的偏移值, 由此可求得堆栈段 SP 的相对段值 (相对于模块起点, 连接程序 LINK 定其为 0:0)。

—9 10H 11H — 64 19

SP=1964, 装入模块取得控制时, 这个值被送到 SP 寄存器中。这是相对于 SS 的堆栈指针偏移量。

—10 12H 13H — 00 00

字校验和 (在可执行部分的所有字的总和的补码), 0000 表示忽略溢出文件中所有字的负校验和。

—11 14H 15H — 00 00

IP=0000, 它在模块取得控制时这个值被送到 IP 寄存器中。它是相对于 CS 的指令指针偏移量。

—12 16H 17H — 00 00

CS=0000, 装入模块代码段的段的偏移值, 由此可求得代码段 CS 的相对段值 (相对模块起点 0:0)。

—13 18H 19H — 22 00

第一个重定位项的位移 (相对于文件起点的段偏移量)。

—14 1AH 1BH — 00 00

覆盖号 (通常 0000 = 主程序)。

—15 1CH 1DH — 01 00

Borland TLINK.EXE 一般总用这个值。

—16 1EH — FB

特征字。

—17 1FH — 20

TLINK.EXE 的版本号。主版本号在高 4 位, 副版本号在低 4 位。

—18 20H 21H — 72 6A

TLINK.EXE 文件版本 V2.0 为 72 6A, 对 V3.0 为 6A 72。

二 重定位表

它紧跟在格式化区之后,由若干重定位项组成。重定位项数由格式化区的 06H 07H 决定,它等于程序段区中的程序段的段数。每个重定位项含 4 个字节,前两个字节为偏移值,后两个为段值。TC.EXE 的重定位表共占 $0 \times \text{CF}4 * 4 = 0 \times \text{33D}0$ 字节。重定位项与装入字区的装入字对应,它指出其对应的装入字位置。

TC.EXE 最后一个实际重定位项的值是 371C:44FA,而它后面还有 4 个未用项是供程序运行时覆盖用的。

三 装入模块

它由装入字区和文件段区构成。装入模块位于文件标头之后,它是程序本身的运行部分(或程序正文),但其起始地址必须在 0×200 字节的边界处,换句话说,装入字区一般并不紧挨着重定位表。

四 装入字区

装入字区由若干装入字构成,它主要用于程序的重定位。装入字的内容决定了程序段运行时的段值。装入字是与重定位项相对应的,这个字在与其对应的程序段得到控制权之前要进行修改,修改的过程称为【重定位】或【再定位】。因此,若改变装入字的内容,便可实现对程序模块的重定位。

五 程序段区

程序段区包括若干程序段,它们是执行代码。

六 .EXE 文件的装入

对 TC.EXE 先操作:

```
C>COPY TC.EXE MY.EXE
```

```
C>DEBUG MY.EXE
```

```
-R
```

```
AX=0000 BX=0004 CX=6BC9 DX=0000 SP=1964 BP=0000 SI=0000 DI=0000
```

```
DS=1A3F ES=1A3F SS=61E9 CS=1A4F IP=0000 NV UP EI PL NZ NA PO NC
```

```
1A4F:0000 BAC655MOV DX,55C6
```

```
-
```

1. 在可分配的内存最低处建立一个 PSP。DS 和 ES 中含有程序的 PSP 段值。
2. 将文件标头的格式化区读入内存;
3. 根据 04H 05H 和 08H 09H 之差算出装入模块长度。这个长度再根据 02H 03H 的值适当进行减少调整;
4. 再根据所设置的装入程序开关的高/低,决定装入模块的【开始段】,又称【起始段, STARTING SEGMENT】。起始段可由系统装入时确定,一般是可变的;当然也可以由用户在编程时确定。

这里对 TC.EXE 加 0×10 便是装入的起始段:

```
1A3F+10=1A4F
```

5. 把装入模块读入从起始段开始的内存中,其容量为文件长度减去文件标头长度;

6. 将文件标头的重定位表读入工作区;

7. 将重定位表项的段值与起始段值之和作为段值、把重定位表项的偏移值作为偏移,根据这个新地址(段:偏移)找到这个重定位项对应的装入字;然后将装入字值加上起始段值,这个和数作为新的装入字代替原有的装入字。对所有的重定位项和装入字都进行此操作。

TC.EXE 的第一个重定位项是 0000:0001,作为指针(相对于模块起点)指向模块中的字 77 3B(未重定位时),装载(重定位)后变成 C6 55:

$$1A4F + 3B77 = 55C6$$

即加上起始段值 1A4F。

8. 把格式化区 0EH 0FH 中的段值加上起始段值作为 SS 寄存器的值;

9. 把格式化区 10H 11H 中的值作为 SP 寄存器的值;

10. 把 DS 和 ES 寄存器的值置成 PSP 的段值;

11. 把格式化区 12H 13H 中的段值作为 IP 寄存器的值;

12. 把格式化区 14H 15H 中的值加上起始段值作为 CS 寄存器的值。

这些过程完成后,装入模块就得到了控制权。

七 说明

1. .EXE 转换成 .COM 文件

可以用 MS-DOS 的外部命令将 .EXE 文件转换成 .COM 文件,但是并不是所有的 .EXE 文件都可以转换成 .COM 文件。它必须具备以下条件:

(1) 必须是由连接程序 LINK 产生的有效的 .EXE 文件;

(2) 文件驻留在内存的部分,即实际代码和数据所占字节数必须小于 64K;

(3) 不能有堆栈(STACK)段。

对 Turbo C 而言,一般不能用 EXE2BIN 将 .EXE 文件转换成 .COM 文件(不能转换时会出现 File cannot be converted 的提示)。而只能在微(Tiny)存储模式下编译生成的 .EXE 文件可以用 DOS 的 EXE2BIN.EXE 程序转换成 *.COM 文件。

2. .EXE 文件格式的变化

随着 MS-DOS 版本的升级,例如,为了和 MS-DOS 4.0 和 Microsoft Windows 相适应,新的 .EXE 文件格式已作了修改。

第三十三章 键盘与鼠标

33.1 键盘

键盘是微机常用的数据输入设备。程序员应当了解怎样在软件中处理键盘输入,尤其要弄清 TC 为 IBM PC 提供的处理键盘输入的库函数。

键盘由一组排列成矩阵方式的按键开关组成。以 IBM PC/XT 键盘为例,它由 83 个键(AT 机有 84 个键,多一个键是 Sys Req 键)。每个键在键盘上都有一个【位置码】,即位置编号。因此 83 键的位置码为 1~83,或十六进制数 0x1~0x53。该键盘内部采用 Intel 8048 单片微处理器控制的,具有自测试功能(确定有无键按下,同时按下的键数及按键的位置等)。它用 5 芯电缆与主机相连(针脚 1~5 的信号分别为时钟、数据、备用、地和 +5V 电源),输出按下或释放(即松开)键的扫描位置码,简称【单键扫描码】。一次只按一个键(称按单键)时,扫描码等于键位置码,即 0x1~0x53(见表 33-1)。

表 33-1 83 键键位置码(或单键扫描码)

键 名	0 进制	16 进制	键 名	10 进制	16 进制
F1~F10	59~68	3b~44	Alt	56	38
~`	41	29	Spacebar	57	39
! 1~)0	2~11	2~B	Caps Lock	58	3A
=	12	C	ESC	1	1
+ =	13	D	Num Lock	69	45
\ /	43	2B	Scroll Lock	70	46
Backspace	14	E	Sys Req	84	54
Tab	15	F	7Home	71	47
Q~P	16~25	10~19	8↑	72	48
[26	1A	9PgUp	73	49
]	27	1B	PrtSc *	55	37
Ctrl	29	1D	4Left arrow	75	4B
A~L	30~38	1E~26	5	76	4C
;	39	27	6Right arrow	77	4D
"'	40	28	-(小键盘)	74	4A
Enter	28	1C	1End	79	4F
Leftshift	42	2A	2↓	80	50
Z~M	44~50	2C~32	3PgDn	81	51
<.	51	33	0Ins	82	52
>.	52	34	Del	83	53
? /	53	35	+(小键盘)	78	4E
Rightshift	54	36			

一次按了一个以上键（称按组合键，即同时按住几个键，或者按住一个键始终不放，接着又依次按若干键）时，扫描码就可能大于 0x53（但最大是 0x84）。当一个键按着 0.5 秒后仍不松开，将自动重复输出该键扫描码，重复频率为每秒 10 次。

本章将简要介绍几个和键盘相关的库函数。

- . 使用 BIOS 的中断 INT 16H 执行各种键盘操作 —1 bioskey()
- . 检查键盘缓冲区 —2 kbhit()
- . 从键盘缓冲区读一字符并回显 —3 getch()
- . 从键盘缓冲区读一字符但不回显 —4 getche()
- . 向键盘缓冲区回退一个字符 —5 ungetch()

下面讨论的环境为西文 DOS 状态，至于 CCDOS 状态请参考有关 CCDOS 对键操作的说明。

33.1.1 键的分类

为便于叙述，按键用途和在键盘上的位置将其分为几类。

1. 打字机键

键盘中间部分的一些键与标准英文打字机键盘上的键类似，有

1)、2@、3#、4\$、5%、6^、7&、8*、9(、0)、—、=+、\|、
Q、W、E、R、T、Y、U、I、O、P、[{、}]、
A、S、D、F、G、H、J、K、L、;、'、
Z、X、C、V、B、N、M、,<、.>、/?、
Space bar(空格键)

图 33-1

这些键表示的符号可以在打印机上打出。

2. 控制键

Esc	即 Escape 键，非字符键，产生一字节 ASCII 码，也可由 Ctrl-[键操作产生
Ctrl	即 Control 键
Alt	字符交换键
Back—S	即 Backspace 键（有些键盘上标为 ←），退格删除，非字符键，产生一字节 ASCII 码，也可由 Ctrl-H 键操作产生
L—Shift	即键盘左边的 Shift 键
Caps Lock	字母大小写键，简称为 Caps
Tab	非字符键，产生一字节 ASCII 码，也可由 Ctrl-I 键操作产生
R—Shift	即键盘右边的 Shift 键
Num Lock	数字锁定键，简称为 Num
Enter	回车键，非字符键，产生一字节 ASCII 码，也可由 Ctrl-M 键操作产生
* PrtSc	打印屏幕
Scroll Lock	屏幕滚动锁定

3. 功能键

F1、F2、F3、F4、F5、F6、F7、F8、F9、F10

4. 副键盘键 (小键盘键)

				Sys Req
	/	*	-	
7Home	8 ↑	9PgUp	+	
4←	5	6→		
1End	2 ↓	3PgDn		
0Ins (即Insert 键)		. Del		

图 33-2

副键盘中间的标识有数字 5 的键,当 Num 键处于光标方式而不是数字方式时,它不产生编码,或者说按它后不起任何作用。

AT 机上第 84 个键 Sys Req 称为【系统请求键】,是为多用户系统进入主系统目录提供一种方法。按下该键时,AX 中出现 8500H 码,并执行了 INT 15H。释放该键时,AX 中出现 8501H 码,并再次执行 INT 15H。AT 机的 BIOS 没有为 INT 15H 中的功能 84H 和 85H 提供代码,只有简单的返回。但系统软件能代替中断向量 15H。系统软件指的是 Sys Req 例程。它首先读取 AL (如果按着 Sys Req 键,AL=0;释放 Sys Req 键,AL=1)。用户可以利用 INT 15H 中断及 Sys Req 键作一些别的事情。这只要让 Sys Req 例程重新设置中断向量。若在 AH 中发现与 84H 和 85H 不同的功能号,则控制转到指定的 INT 15H 例程。

5. 双态键

Scroll Lock、Caps、Num、Ins 键都有两种状态,假定按一下为状态 A (状态 B 的相反状态),再按一下则为状态 B,再按又回到状态 A,如此等等。

6. 换档键

L—Shift、R—Shift、Ctrl、Shift

7. 组合键

Ctrl—Breakspace、Ctrl—Alt—Del (热启动)、Ctrl—Alt—←、Ctrl—Alt—→等等。

Ctrl—H、Ctrl—I、Ctrl—M、Ctrl—[{

非法组合键不产生编码。

组合键的优先级是 Alt、Ctrl、Shift。

33.1.2 接通码和释放码

键盘以串行方式向主机发送扫描码。8048 微处理器将扫描码存储在一个可编程外围输入输出接口芯片 8255A—5 的端口 A 中 (端口地址为 60H)。8255A—5 芯片装在主机板上,以下简称 8255。

每当按下一个键,8048 便向 8255 的端口 A 送去一个一字节扫描码,这时最高位即第 7 位为 0,低 7 位对应于键的位置码。例如,键 Z 的位置码是 2CH。当按下 Z 键时,8048 送给 8255 芯片端口 A 的扫描码是 00101100B,我们称这种扫描码为【接通码 (Make code)】。对每个键都规定了一个唯一的 8 位接通扫描码,当按键时就送出这个扫描码。送出接通码的速率称为【拍发速率】。当将键松开 (或称释放) 时,8048 又向端口 A 送去一个扫描码,我们称这种扫描码为【释放码 (Break code)】或【断开码】。在释放按键时,除 AT 机外,释放码也是一字节,一般将最高位变成 1,而低 7 位不变。例如,上面 Z 键释放时,送给端口 A 的扫描码为 10101100B,即 ACH。

对 AT 机,接通码和其它机器是一样的,但释放码则不同。释放码为两个字节,是在扫描

码前面加上一个 F0H 字节,即第一个字节总是 F0H,第二个字节是通常所说的扫描码,而扫描码的最高位即第 7 位总是 0。

由此可见,每个键入字符在 8255 芯片上要登记两次。每次登记后,8255 的端口 B (地址为 61H) 将向 8048 一个“应答”信号(端口 B 的位 7 为 0 时,端口 A 放键盘接收的 8 位扫描码)。端口 A 第一次登记扫描码时,应答信号将位扫描码 7 变成 1;第二次登记时,把位 7 变成 0。端口 B 的位 6 控制着时钟信号,它总是 1,否则将锁闭键盘(参见本章 33.1.6 节程序 K5.C)。

CPU 在响应键盘中断时,通过 8255 的端口 A 读取键盘扫描码。在扫描码存入端口 A 的同时,键盘中断 INT 9H 被调用,CPU 立即停止它正在进行的工作,转去执行一个分析扫描码的例程,即 INT 9H 中断处理程序。例程主要处理步骤有:

1. 读扫描码;
2. 设置组合键和双态键;
3. 把扫描码变换成字符码;
4. 在键盘缓冲区中设置字符码(但对双态键或换档键,除 Ins 键外,无字符码送入缓冲区,因而对它们按单键时屏幕无反应。);
5. 对特殊功能键(如 Ctrl-Break, Ctrl-Alt-Del 等)进行解释。只有中断 INT9H 能立即响应按键事件,而中断 INT 16H 通常发生在读键盘输入(从键盘缓冲区中取字符)的时候。

33.1.3 换档键 / 双态键的状态字节

内存 0040:0017 和 0040:0018 两个字节的位表示了换档键和双态键的状态。

1. 0040:0017 字节(称【移位标志】)

表 33-2

位	键操作	当位值为 1 时的含义
7	Ins	On 处于插入状态,Insert 键被按下 (对 101/102 键盘任一 Insert 键)
6	Caps	On 处于大写字母状态,Caps 键被按下 AT 键盘大写状态指示灯亮
5	Num	On 处于数字状态,Num 键被按下 AT 键盘数字指示灯亮
4	Scroll Lock	On 处于屏幕锁住状态,Scroll lock 键被按下 AT 键盘屏幕指示灯亮
3	Alt-Shift	两键同时按下
2	Ctrl-Shift	两键同时按下
1	L-Shift	左边的 Shift 键被按下
0	R-Shift	右边的 Shift 键被按下

2. 0040:0018 字节(扩充移位标志,只用于有改进键盘的 AT 或 PS/2 机)

表 33-3

位	键操作	当位值为 1 时的含义
7	Ins	此键按下
6	Caps	此键按下
5	Num	此键按下
4	Scroll Lock	此键按下
3		按下 Ctrl+Num 后 (或暂停, Pause 被锁) 等待用户按其它键。如用于打印时暂停等, 位本身无意义
2		未使用 (或按下了 Sys Req 键)
1		未使用 (或按下了左边的 Alt 键)
0		未使用 (或按下了左边的 Ctrl 键)

键盘中断在分析键入字符之前, 先要检查上述的状态位, 而不管键盘缓冲区中读到什么信息。用户程序如改变上述有效位中的一个, 等于实际敲入对应键。可以通过程序 K1.C 来理解状态字节中的值的变化。

```

C>TYPE K1.C                                /* 测试状态字节程序 */
#include "stdio.h"
#include "bios.h"
#include "ctype.h"

/* 定义判断 0040:0017 字节位意义的字符常量 */
#define Ins      0x80
#define Caps     0x40
#define NumLock  0x20
#define ScrollLock 0x10
#define Alt      0x08
#define Ctrl     0x04
#define LeftShift 0x02
#define RightShift 0x01
main()
{
    char *a[]={"Ins","Caps","NumLock","ScrollLock","Alt",
               "Ctrl","LeftShift","RightShift",""},
    int key,count;
    unsigned char kb417,kb418; /* 与 0040:0017、0040:0018 两字节相关变量 */
    for(count=0;count<8;count++)
    {
        if(strcmp(a[count],"")==0)break;
    }

```



```

fprintf(stderr, "press %s key, Please! \n", a[count]);
bioskey(0); /* 因函数 bioskey(2) 只判别状态字节, 故用 bioskey(0) 来 */
/* 接收键盘输入字符。此处不能用语句 while(bioskey(1) == 0); */
/* 代替, 因为函数 bioskey(1) 判断一旦有键输入, 便退出循 */
/* 环, 并且键入值还保留。故 for 循环实际将连续对同一键 */
/* 值进行, 这是我们不希望的。 */
kb417 = peekb(0x40, 0x17); /* peekb() 取得段 0040H、偏移量 0017H 中的值 */
/* 上句也可写成 kb417 = peekb(0, 0x417); */
printf("+++++ kb417 = %u : 0x%x\n", kb417, kb417);
kb418 = peekb(0x40, 0x18); /* 取得 0040:0018 中的值 */
printf("+++++ kb418 = %u : 0x%x\n", kb418, kb418);
key = bioskey(2); /* 取得 0040:0017 中的值 */
printf("bioskey(2) = %d : 0x%x\n", key, key);
kb417 = peekb(0x40, 0x17);
printf("==== kb417 = %u : 0x%x\n", kb417, kb417);
kb418 = peekb(0x40, 0x18);
printf("==== kb418 = %u : 0x%x\n", kb418, kb418);
if(key){ /* 连续判断有那几个键按下 */
    if(key & Ins) printf("Ins\n");
    if(key & Caps) printf("Caps\n");
    if(key & NumLock) printf("NumLock\n");
    if(key & ScrollLock) printf("ScrollLock\n");
    if(key & Alt) printf("Alt\n");
    if(key & Ctrl) printf("Ctrl\n");
    if(key & LeftShift) printf("LeftShift\n");
    if(key & RightShift) printf("RightShift\n");
}
}
}

```

程序经编译后退出集成环境, 然后执行。例如按提示连续按相应键便有下面结果 (注意: 由于像 Caps 等键是非字符键, 因此你按了 Caps 键后松开, 然后按一字符键)。

```

+++++ kb417 = 128 : 0x80
+++++ kb418 = 128 : 0x80
bioskey(2) = 128 : 0x80
==== kb417 = 128 : 0x80
==== kb418 = 128 : 0x80
Ins
+++++ kb417 = 192 : 0xc0
+++++ kb418 = 0 : 0x0
bioskey(2) = 192 : 0xc0
==== kb417 = 192 : 0xc0
==== kb418 = 0 : 0x0
Ins
Caps
+++++ kb417 = 224 : 0xe0
+++++ kb418 = 0 : 0x0

```

bioskey(2)= 224 ; 0xe0
===== kb417= 224 ; 0xe0
===== kb418= 0 ; 0x0

Ins

Caps

NumLock

++++ kb417= 240 ; 0xf0
++++ kb418= 0 ; 0x0
bioskey(2)= 240 ; 0xf0
===== kb417= 240 ; 0xf0
===== kb418= 0 ; 0x0

Ins

Caps

NumLock

ScrollLock

++++ kb417= 248 ; 0xf8
++++ kb418= 0 ; 0x0
bioskey(2)= 248 ; 0xf8
===== kb417= 248 ; 0xf8
===== kb418= 0 ; 0x0

Ins

Caps

NumLock

ScrollLock

Alt

++++ kb417= 244 ; 0xf4
++++ kb418= 0 ; 0x0
bioskey(2)= 244 ; 0xf4
===== kb417= 244 ; 0xf4
===== kb418= 0 ; 0x0

Ins

Caps

NumLock

ScrollLock

Ctrl

++++ kb417= 242 ; 0xf2
++++ kb418= 0 ; 0x0
bioskey(2)= 242 ; 0xf2
===== kb417= 242 ; 0xf2
===== kb418= 0 ; 0x0

Ins

Caps

NumLock

ScrollLock

LeftShift

++++ kb417= 241 ; 0xf1

```

+++++ kb418= 0 : 0x0
bioskey(2)= 241 : 0xf1
===== kb417= 241 : 0xf1
===== kb418= 0 : 0x0
Ins
Caps
NumLock
ScrollLock
RightShift

```

从上可见, bioskey(2) 相当于 0040:0017 字节中的值; 其次, 除 Ins 键外, 其余 0040:0018 字节中的值为 0。这是按键操作引起的。如果我们按这样的方法操作: 按住 Caps 键的同时又按一个字符键, 便可看到 0040:0018 中的值将不为 0 了, 而且, 有时可以发现在执行函数 bioskey(2) 前后 0040:0018 中的内容不同 (与按键相关)。下面便是这样操作的一些例子。

```

+++++ kb417= 144 : 0x90
+++++ kb418= 128 : 0x80
bioskey(2)= 144 : 0x90
===== kb417= 144 : 0x90
===== kb418= 128 : 0x80
Ins
ScrollLock
+++++ kb417= 16 : 0x10
+++++ kb418= 128 : 0x80
bioskey(2)= 16 : 0x10
===== kb417= 16 : 0x10
===== kb418= 128 : 0x80
ScrollLock
+++++ kb417= 80 : 0x50
+++++ kb418= 64 : 0x40
bioskey(2)= 80 : 0x50
===== kb417= 80 : 0x50
===== kb418= 64 : 0x40
Caps
ScrollLock
+++++ kb417= 112 : 0x70
+++++ kb418= 32 : 0x20
bioskey(2)= 112 : 0x70
===== kb417= 112 : 0x70
===== kb418= 32 : 0x20
Caps
NumLock
ScrollLock
+++++ kb417= 32 : 0x20
+++++ kb418= 0 : 0x0
bioskey(2)= 32 : 0x20

```

```
===== kb417 = 32 : 0x20
===== kb418 = 0 : 0x0
```

在分析扫描码时,除换档键和双态键外,所有释放码都被舍弃。对于换档键和双态键,释放码改变状态字节。在处理接通码时,状态字节也要改变。即使在第一字节为 ASCII 码时,也要参考状态字节,例如区分同一键表示大小写字母等。

键盘例程一次只识别一个输入字符,并区别字符 ASCII 码字符或扩充码。

33.1.4 库函数 bioskey()

在标头文件 bios.h 中有直接键盘操作函数 bioskey() 的原型:

```
—1 int —Cdecl bioskey(int cmd);
```

函数使用 BIOS 的中断 INT 16H 执行各种键盘操作。cmd 是 INT 16H 的功能号(00H ~ 02H)。cmd 可理解为 combine (组合功能)或 command (命令)。

cmd=0 它等待从键盘上键入一个值,如果已键入了一个键,则它返回该键值。AH=扫描码,AL=ASCII 码字符。

- 1 它查询是否按下了一个键,有一个键按下,它将留在键盘缓冲区中。返回一个非 0 值 (否则返回 0)。如再调用 bioskey(0),则此时不必再按键,原先存在缓冲区中的键值将立即被取出。
- 2 它返回的低 8 位 (在 AL 中) 的每一位表示一个移位键是否被按下,即
 - 位 0=1 右边的 Shift 键被按下
 - 位 1=1 左边的 Shift 键被按下
 - 位 2=1 Ctrl 键被按下 (对 101/102 键盘任一 Ctrl 键)
 - 位 3=1 Alt 键被按下 (对 101/102 键盘任一 Alt 键)
 - 位 4=1 Scroll lock 键被按下
 - 位 5=1 Num 键被按下
 - 位 6=1 Caps 键被按下
 - 位 7=1 Insert 键被按下 (对 101/102 键盘任一 Insert 键)

注意:本函数是针对 83/84 键盘而言的,其它和它不相容的扩充键击都将被清除,因此获扫描码可能不实。要想不丢弃扩充键击,可调用 INT 16H 的功能 10H、11H 和 12H,但本函数不涉及这几个功能。

本函数只适用于 IBM PC 及其兼容机上。

```
C>TYPE K2.C
#include "bios.h"
main()
{
    int j,key,mod;
    while(bioskey(1)==0)printf("请按键结束无限循环...\n");
    /* 当有键按下时 bioskey(1) 返回非 0 值,且键值留在键盘缓冲区中 */
    mod=bioskey(2); /* 返回 BIOS 支持的各个移位键标志 */
    printf("印出移位键状态位:\n");
    for(j=0;j<8;j++) /* 低 8 位 */
        printf("%d=%#01x\t",j,mod>>j & 0x1);
    key=bioskey(0); /* 取得键盘缓冲区中值 */
```

```

printf("\n 你刚刚按的键值:\n");
printf("扫描码AH=%xH ASCII 字符AL=%xH\n",key>>8 & 0x00ff,key & 0x00ff); }
/* 当按了一个 A 键时程序输出:
请按键结束无限循环...
.....
请按键结束无限循环...
印出移位键状态位:
0=0 1=0 2=0 3=0 4=0 5=0 6=0 7=0
你刚刚按的键值:
扫描码AH=1eH ASCII 字符AL=61H
当按了一个组合键(左)SHIFT-A 时程序输出:
请按键结束无限循环...
.....
请按键结束无限循环...
印出移位键状态位:
0=0 1=0x1 2=0 3=0 4=0 5=0 6=0 7=0
你刚刚按的键值:
扫描码AH=1eH ASCII 字符AL=41H
*/

```

33.1.5 键盘缓冲区

键盘缓冲区是指内存中由 32 字节的数据区和 4 字节的指针区构成的存储区(图 33-3)。

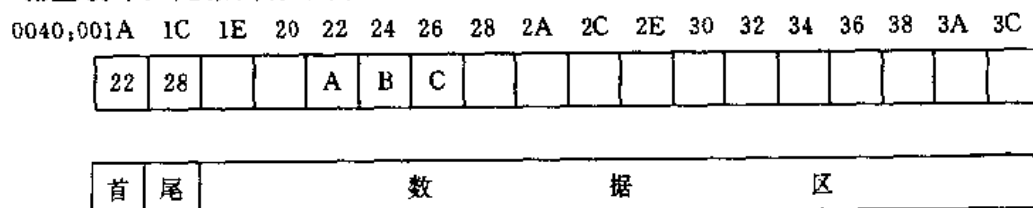


图 33-3 键盘缓冲区构造图

数据区内存占有固定空间,起始地址是 0040:001EH,终止地址是 0040:003DH。键入一个字符要占两个字节。对于一字节的 ASCII 码字符,其 ASCII 码值存于第一个字节即低 8 位中,扫描码存于第二个字节即高 8 位中;对于扩充码,第一字节存 ASCII 码的 0(即 0x0,而不是数字 0),第二字节存扩充码。数据区是一个首尾衔接的闭环队列,是先进先出(FIFO)。

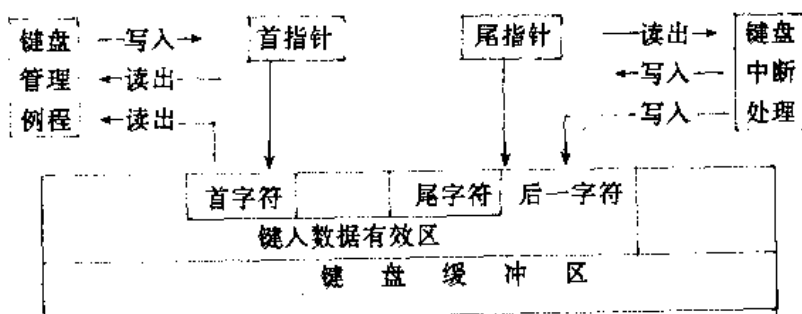


图 33-4 键盘缓冲区管理图

数据区可以连续存放 15 个字符和最后一个回车符。第 16 个字符位置是虚设的,因为它放

回车符时就不再能放通常的 ASCII 码字符（打字机字符）。当缓冲区已满时，后输入的字符将依次复盖从 0040:001E 开始存放的字符，换言之，原先这些字节内的字符会丢失。

指针区占内存固定的 4 个字节，首指针是 0040:001AH ~ 0040:001BH，尾指针是 0040:001C ~ 0040:001DH。首指针指向缓冲区有效数据区的第一个字符，尾指针指向有效区最后一个字符的后一字符。首尾指针各自指向存放字符的第一个字节（见图 33-4）。

1. 从缓冲区读字符的过程

(1) 取出首指针；(2) 如首指针等于尾指针，表明缓冲区空，等待键盘字符的到来；否则，(3) 将首指针所指单元内的值读出；(4) 将首指针值加 2，然后存入 0040:001AH ~ 0040:001BH 内。

2. 向缓冲区写字符过程

(1) 取出尾指针；(2) 将尾指针值加 2 后与首指针值比较，如相等，表明缓冲区满，此时随后键入的字符丢失，系统发出警告（蜂鸣声）。否则，(3) 向尾指针原值所指单元写入字符；(4) 将尾指针加 2 后的值写入 0040:001CH ~ 0040:001DH 中。

在一般情况下，写入缓冲区的字符很快被读出，所以首尾指针值不断在变动。大多数情况出现首尾指针值相等，即缓冲区为空，允许写入字符。但是，如果写入比读出快，就可能出现缓冲区空间不够。实际证明，DOS 设置的这种缓冲区能应付最熟练的操作员的击键输入，即不会使写入与读出发生问题。

缓冲区数据区是循环队列，即当指针值超过 0040:003CH 时，指针值自动调整为 0040:001EH。当首指针为 0040:001EH 时，尾指针值最大为 0040:003CH，超过时系统将响铃告警。从写字符过程可以看出，此时不能写入第 16 个字符，所以说，缓冲区最多只能存放 15 个字符，第 16 个存放字符的单元是虚设的。

程序 K3.C 是一个利用标头文件 dos.h 中宏 peek 检查键盘缓冲区的例子。从程序输出结果可以看到，当首尾指针值相等时，表示缓冲区空。但这并不是说，数据区各存储单元内没有任何东西。

```
C>TYPE K3.C
```

```
#include "conio.h"
#include "dos.h"                                /* 打印缓冲区内容 */
#define P1A-3C printf("%x %x",peek(0x40,0x1a),peek(0x40,0x1c));\
printf("%x%x",peek(0x40,0x1e),peek(0x40,0x20));\
printf("%x%x",peek(0x40,0x22),peek(0x40,0x24));\
printf("%x%x",peek(0x40,0x26),peek(0x40,0x28));\
printf("%x %x",peek(0x40,0x2a),peek(0x40,0x2c));\
printf("%x %x",peek(0x40,0x2e),peek(0x40,0x30));\
printf("%x %x",peek(0x40,0x32),peek(0x40,0x34));\
printf("%x%x",peek(0x40,0x36),peek(0x40,0x38));\
printf("%x %x\n",peek(0x40,0x3a),peek(0x40,0x3c));

main(){
int ch;
getch();                                        /* 按键开始测试缓冲区情况 */
if(kbhit()!=0)printf("First !=0\n"); /* kbhit() 函数测试是否有键按下 */
else printf("First ==0\n");           /* 等于 0 表示缓冲区空，没击键 */
P1A-3C;                                       /* 第一次打印缓冲区内容 */
delay(5000); /* 在延迟 500 毫秒时间内你迅速连续按键，键值进入缓冲区，*/
```

```

/* 但当你按第 16 个键时,系统将响铃报警。如在延迟时间内 */
/* 内你没有按任何键,到时间程序自动结束运行 */
if(kbhit() != 0)printf("Second != 0\n"); /* 不等于 0 表示至少有一键按下 */
else printf("Second == 0\n");
P1A←3C; /* 再次检查缓冲区情况 */
}

```

/* 在延迟时间内无任何键按下时程序运行后的结果

```

First == 0          头尾指针值相等表明执行 getch() 后缓冲区被清除为空
26  26  342e  2e63  1c0d  1e61  e08  e08  e08  e08
08  e08  e08  e08  1970  433  343e  266c
Second == 0
26  26  342e  2e63  1c0d  1e61  e08  e08  e08  e08
e08  e08  e08  e08  1970  433  343e  266c

```

在延迟时间内连续按下了 qwertyuiopasdfg 十五个键,再按键听到响铃声

```

First == 0
2a  2a  343e  326d  342e  2e63  1c0d  1e61  1579
1970 1265 3920  266c  342e  2e63  1c0d  1970  433
Second != 0
2a  28  1e61  1f73  2064  2166  2267  1e61  1071
1177 1265 1372  1474  1579  1675  1769  186f  1970

```

C>qwertyuiopasdfg

*/

如果在程序中将 int ch; 语句改成 int ch,i=15; ,并在语句 delay(5000); 后增加语句

```
while(i--)>getch();
```

后重新编译运行程序,可以发现当键入到铃响后 getch(); 语句并不要求你输入一键,返回 DOS 时 DOS 提示符下也不回显你先前打入的字符,并且缓冲区首尾指针相等。之后,再把 i 初值改成 16 后编译运行,这回你发现在延迟后要键入一键才会结束程序运行,并且首尾指针最后相等,即缓冲区变空。程序 K4.C 使你看到首尾指针值与缓冲区数据区内字符对应情况。

C>TYPE K4.C

```

#include "dos.h"
#include "stdlib.h"
#define KEY bioskey(0)
#define P(B) peek(0,B) /* 定义取 2 字节的宏 */
#define PX printf("Head=%x Tail=%x\n",k41A,k41C); \
printf("1E=%x %x %x %x %x %x %x %x\n",k41E,k420,k422,k424,\
k426,k428,k42A,k42C);printf("2E=%x %x %x %x %x %x %x %x\n",\
k42E,k430,k432,k434,k436,k438,k43A,k43C) /* 打印缓冲区单元内容 */
#define PEEK k41A=P(0x41A);k41C=P(0x41C);k41E=P(0x41E);k420=P(0x420);\
k422=P(0x422);k424=P(0x424);k426=P(0x426);k428=P(0x428);\
k42A=P(0x42A);k42C=P(0x42C);k42E=P(0x42E);k430=P(0x430);\
k432=P(0x432);k434=P(0x434);k436=P(0x436);k438=P(0x438);\
k43A=P(0x43A);k43C=P(0x43C);PX

```

main(){

```

unsigned k41A,k41C,k41E,k420,k422,k424,k426,k428,k42A,
        k42C,k42E,k430,k432,k434,k436,k438,k43A,k43C;
int i,n=0,m;
PEEK;
for(i;n<4;n++)
{
    i=random(3)+1;printf("i=%d ",i); /* random 获得 0~2 的随机整数 */
    for(m=1;m<=i;m++)
        (KEY;while(bioskey(1)==0);
    }
    PEEK;
}
}

```

程序编译后运行,根据 i 值按键,可能输出

```

Head=32 Tail=32 /* 缓冲区最初情况 */
1E=e08 e08 e08 e08 e08 e08 e08 e08
2E=3d00 1c0d 1265 1265 2166 2166 2267 2267
i=2 Head=36 Tail=38 /* 按了 a、b、c 键后 */
1E=e08 e08 e08 e08 e08 e08 e08 e08
2E=3d00 1c0d 1e61 3062 2e63 2166 2267 2267
i=2 Head=3a Tail=3c /* 按了 d、e 键后 */
1E=e08 e08 e08 e08 e08 e08 e08 e08
2E=3d00 1c0d 1e61 3062 2e63 2064 1265 2267
i=3 Head=20 Tail=22 /* 按了 f、g、h 键后 */
1E=2267 2368 e08 e08 e08 e08 e08 e08
2E=3d00 1c0d 1e61 3062 2e63 2064 1265 2166
i=2 Head=24 Tail=26 /* 按了 i、j 键后 */
1E=2267 2368 1769 246a e08 e08 e08 e08
2E=3d00 1c0d 1e61 3062 2e63 2064 1265 2166

```

33.1.6 键码测试程序

从程序 K5.C 可以了解每个键的两种扫描码,即接通码和释放码。它们在屏幕左上角两行上显示。逐次按键并松开便可看到。该程序涉及如何改写键盘中断 INT 9H、访问 8255 端口及怎样编制常驻内存程序等问题。

```

C>TYPE K5.C /* 测试键扫描码程序 */
#include "dos.h" /* 根据马文骞的程序改写并注释 */
#define tsr-size 400 /* tsr-size 告诉 DOS 常驻部分需要占用 */
/* 内存容量,单位为节,一节等于16字节 */
#define ITOACAT(X) itoa(invalue,ascii-scan-code,X);\
    strcat(string1,ascii-scan-code);
/* 将值按 X 进制转换为字符串,最后放到 string1 中 */
#define CAT strcat(string1," ") /* string1 尾部加上逗号 */
void interrupt (*doskbint)(void); /* 定义指向函数的指针 */
void interrupt (*int0x81)(),interrupt(*int0x82)();
/* 注意,上句不能写成 void interrupt (*int0x81)(),(*int0x82)(); */

```



```

void interrupt mykbint(); /* 定义中断函数 */
unsigned mess,mess, kb—ss, kb—stack, invalue, m;
static union REGS reg;
static struct SREGS segm;
char ascii—scan—code[8], string1[17], string0[17];
int vedosegm=0xb800;
/* 对 CGA 和 EGA 显示器,显示 RAM 的起始地址 B800:0000 */
int flag=0; /* 生成接通码或释放码的标志 */
main(int argc,char * argv[]) /* 命令行允许有参数 */
{int i;
int0x81=getvect(0x81); /* 对标准 DOS,中断 0x81 和 0x82 都未使用 */
int0x82=getvect(0x82); /* 取得中断向量(或中断处理函数)的入口地址 */
doskbint=getvect(0x09); /* doskbint 获 DOS 的 INT 9H 中断向量入口 */
if(argc>1)
{ /* 键入 C>K5□stop 则停止查扫描码 */
if ((strcmp(argv[1],"stop")==0||strcmp(argv[1],"STOP")==0) &&
int0x81!=0 && int0x82!=0)
{
disable(); /* 屏蔽中断。但从外设来的不可屏蔽中断仍允许 */
setvect(0x09,int0x82); /* 恢复 DOS 原中断 INT 9H */
setvect(0x82,0);
enable(); /* 开放中断,允许对任何设备的中断发生 */
return(0);
} /* 键入 C>K5□restar 则恢复查扫描码 */
if ((strcmp(argv[1],"restart")==0||strcmp(argv[1],"RESTART")==0) &&
int0x81!=0 && int0x82==0)
{
disable();setvect(0x82,doskbint);
setvect(0x09,int0x81); /* 中断 81H 始终指向中断函数 mykbint */
enable();return(0); /* 修改中断 INT 9H */
}
else return(0);
}
if(int0x82!=0) /* 表明已常驻,不允许重入,即它已被使用 */
{ /* 就不允许再一次用 C>K5 命令激活 */
sound(1200);sleep(1); /* 扬声器鸣叫,程序暂停执行一会儿 */
nosound();return(0); /* 关闭扬声器后退出 */
}
segread(&segm); /* 读段寄存器当前值 */
kb—ss=—SS; /* 在调用函数前后,数值不变的寄存器仅有一CS、 */
/* 一BP、一SI 和 一DI。因编译器在产生机器码时 */
disable(); /* 要用到其它寄存器。所以对关键值应先保留, */
/* 后恢复 */
setvect(0x82,doskbint); /* 原中断 INT 9H 暂存,保留 */
setvect(0x09,mykbint); /* 修改中断 INT 9H */
setvect(0x81,mykbint); /* 保留函数入口 */

```

```

enable();
kb—stack=(tsr—size—(segm.ds—segm.cs))*16—300;
reg.x.ax=0x3100; /* DOS 功能调用 31H */
reg.x.dx=tsr—size; /* 常驻部分需要占多少内存,一般估算。将 .EXE 文件 */
/* 所占字节数除以 16 后再乘以 2。精确数字通过实验 */
intdos(&reg,&reg); /* 确定。结束并驻留内存。由于不直接调用 mykbint 函 */
/* 数,所以用 F7 单步调试时不能访问 mykbint 中的变量 */
void interrupt mykbint() /* 中断函数 */
{
int color=3;
disable();mesp=—SP;mess=—SS;—SP=kb—stack;—SS=kb—ss;enable();
invalue=inportb(0x60); /* 读 8255 芯片 A 口,取扫描码 */
strcpy(string1,""); /* 清串 string1 */
ITOACAT(10),CAT; /* 十进制 */
ITOACAT(16),CAT; /* 十六进制 */
ITOACAT(2); /* 二进制 */
if(flag==0)strcpy(string0,string1); /* 取接通码放入 string0 */
else{
for(m=0;m<320;m++)pokeb(vedosegm,m*2,0x7000+' '); /* 清屏幕左上角 */
for(m=0;m<320;m++)pokeb(vedosegm,m*2+1,0x7000+color); /* 颜色 */
for(m=0;m<strlen(string0);m++)
pokeb(vedosegm,160+m*2,0x7000+string0[m]); /* 显示 */
for(m=0;m<strlen(string1);m++)
pokeb(vedosegm,320+m*2,0x7000+string1[m]);delay(10);
}
flag=1—flag;
disable();—SP=mesp;—SS=mess;enable();
(*doskbint)(); /* 执行原 DOS 的 INT 9H 中断。如无此句,运行程序将死机 */
} /* 码按 10、16 和 2 进制三种方式在屏幕左上角显示出来。下为例子 */

/*      键名          接通码          释放码
Ctrl      29,1d,11101    156,9c,10011100
Left Shift 42,2a,101010    170,aa,10101010
Right Shift 54,36,110110    182,b6,10110110
Alt        56,38,111000    184,b8,10111000
Caps       58,3a,111010    186,ba,10111010
Num        69,45,1000101    197,c5,11000101
Scroll Lock 70,46,1000110    198,c6,11000110
Sys Req    84,54,1010100    212,d4,11010100
*/

```

K6.C 是测试键 ASCII 码和扩充码的程序,编译后生成 K6.EXE,用命令行 C>K6>M.C 向指定文件 M.C 输出。注意,程序不能测试 Ctrl、←(Left arrow)、→(Right arrow)、Alt、Caps、Num、Scroll Lock 和 Sys Req 等键的扫描码,或者说程序对这些键不产生码。

C>TYPE K6.C /* 测试第一字节(或称【普通字节】)和第二字节【扩充字节】内容 */

```
#include "stdio.h"
```

```

#include "bios.h"
#include "ctype.h"
#define MAX1 300
main()
{
    /* 测试键明细表。最后用空字符""作终结标志 */
    char *a[]={"aA","bB","cC","dD","eE","fF","gG","hH","iI","jJ","kK","lL",
        "mM","nN","oO","pP","qQ","rR","sS","tT","uU","vV","wW","xX",
        "yY","zZ","[{","\\","|","~","1","2@","3#","4$","5%","6^",
        "7&","8*","9(",")","_","=","<",">","/?",";","'\",'",
        "L-arrow4","R-arrow6","U-arrow8","D-arrow2","Home?","End1",
        "PgUp9","PgDn3","Ins0","Del.", " * PrtSc","Num-","Num+",
        "Esc","Back-S","Tab","Enter","Space",
        "F1","F2","F3","F4","F5","F6","F7","F8","F9","F10",""};
    char *b[]={"","Alt-","Ctrl-","Shift-"};
    int key[MAX1],lo,hi;
    int count=0,n;
    printf(" Key\tSingle\t\tAlt-Single\tCtrl-Single\tShift-Single\n\n");
    for(count=0;count<MAX1;count++)
    {
        if(strcmp(a[count],"")==0)break; /* 如果是空字符,程序终止运行 */
        /* 对字符比较,不能用这样的语句: if(a[count]=="")break; */
        fprintf(stdout,"%s\t",a[count]);
        for(n=0;n<4;n++)
        {
            fprintf(stderr,"\npress %s%s key, Please! \t\t",b[n],a[count]);
            while(bioskey(1)==0); /* 直到有按键才退出循环 */
            /* 键值为下一次 bioskey(0) 调用。 */
            key[count]=bioskey(0); /* 返回按键值。如低 8 位非 0, 则低 8 */
            /* 位为 ASCII 码,高 8 位为扫描码;如低 8 位为 0,高 8 位为扩充码。 */
            lo=key[count]& 0xFF; /* 低 8 位 */
            hi=(key[count]>>8)& 0xFF; /* 高 8 位 */
            fprintf(stderr,"%d,%d:%x,%x\t",lo,hi,lo,hi);
            fprintf(stdout,"%d,%d:%x,%x\t",lo,hi,lo,hi);
            /* scan-ascii=(lo==0)? hi+256;lo;
            strcpy(str0,"");strcpy(str1,"");
            itoa(lo,str0,2);itoa(hi,str1,2); /* 化成二进制数显示
            fprintf(stderr,"%x,%x,%d,%d:%s,%s\t",
                lo,hi,lo,hi,str0,str1);
            fprintf(stdout,"%x,%x,%d,%d:%s,%s\t",
                lo,hi,lo,hi,str0,str1); */
        }
        fprintf(stdout,"\n");
    }
}
/* C>K6>M.C /* 利用 DOS 的重定向命令将原向屏幕输出送入 M.C */
C>TYPE M.C /* 有两个标识符的键名,按 Singe,Shift-Singe 排列 */

```

/* 为节省排列, Shift—Single 对应小写字母, 直接 */
 /* 按小写字母键, 而不是去按 Shift—字母键。*/
 /* Shift 栏相当于大写字母 (键操作 Caps—字母键) */

表 33—4 (印出格式: 低 8 位, 高 8 位)

Key	Single	Alt—Single	Ctrl—Single	Shift—Single
Aa	65,30;41,1e	0,30;0,1e	1,30;1,1e	97,30;61,1e
Bb	66,48;42,30	0,48;0,30	2,48;2,30	98,48;62,30
Cc	67,46;43,2e	0,46;0,2e	3,46;3,2e	99,46;63,2e
Dd	68,32;44,20	0,32;0,20	4,32;4,20	100,32;64,20
Ee	69,18;45,12	0,18;0,12	5,18;5,12	101,18;65,12
Ff	70,33;46,21	0,33;0,21	6,33;6,21	102,33;66,21
Gg	71,34;47,22	0,34;0,22	7,34;7,22	103,34;67,22
Hh	72,35;48,23	0,35;0,23	8,35;8,23	104,35;68,23
Ii	73,23;49,17	0,23;0,17	9,23;9,17	105,23;69,17
Jj	74,36;4a,24	0,36;0,24	10,36;a,24	106,36;6a,24
Kk	75,37;4b,25	0,37;0,25	11,37;b,25	107,37;6b,25
Ll	76,38;4c,26	0,38;0,26	12,38;c,26	108,38;6c,26
Mn	77,50;4d,32	0,50;0,32	13,50;d,32	109,50;6d,32
Nn	78,49;4e,31	0,49;0,31	14,49;e,31	110,49;6e,31
Oo	79,24;4f,18	0,24;0,18	15,24;f,18	111,24;6f,18
Pp	80,25;50,19	0,25;0,19	16,25;10,19	112,25;70,19
Qq	81,16;51,10	0,16;0,10	17,16;11,10	113,16;71,10
Rr	82,19;52,13	0,19;0,13	18,19;12,13	114,19;72,13
Ss	83,31;53,1f	0,31;0,1f	19,31;13,1f	115,31;73,1f
Tt	84,20;54,14	0,20;0,14	20,20;14,14	116,20;74,14
Uu	85,22;55,16	0,22;0,16	21,22;15,16	117,22;75,16
Vv	86,47;56,2f	0,47;0,2f	22,47;16,2f	118,47;76,2f
Ww	87,17;57,11	0,17;0,11	23,17;17,11	119,17;77,11
Xx	88,45;58,2d	0,45;0,2d	24,45;18,2d	120,45;78,2d
Yy	89,21;59,15	0,21;0,15	25,21;19,15	121,21;79,15
Zz	90,44;5a,2c	0,44;0,2c	26,44;1a,2c	122,44;7a,2c
[{	91,26;5b,1a		27,26;1b,1a	123,26;7b,1a
\	92,43;5c,2b		28,43;1c,2b	124,43;7c,2b
]}	93,27;5d,1b		29,27;1a,1b	125,27;7d,1b
~	96,41;60,29			126,41;7e,29
11	49,2;31,2	0,120;0,78		33,2;21,2
2@	50,3;32,3	0,121;0,79	0,3;0,3	64,3;40,3
3#	51,4;33,4	0,122;0,7a		35,4;23,4
4\$	52,5;34,5	0,123;0,7b		36,5;24,5
5%	53,6;35,6	0,124;0,7c		37,6;25,6
6^	54,7;36,7	0,125;0,7d	30,7;1e,7	94,7;5e,7
7&	55,8;37,8	0,126;0,7e		38,8;26,8
8*	56,9;38,9	0,127;0,7f		42,9;2a,9
9(57,10;39,a	0,128;0,80		40,10;28,a
0)	48,11;30,b	0,129;0,81		41,11;29,b
=	45,12;2d,c	0,130;0,82	31,12;1f,c	95,12;5f,c

= +	61,13,3d,d	0,131;0,83		43,13;2b,d
, <	44,51;2c,33			60,51;3c,33
. >	46,52;2e,34			62,52;3e,34
/ ?	47,53;2f,35			63,53;3f,35
! :	59,39;3b,2			58,39;3a,27
"	39,40;27,28			34,40;22,28
L—arrow4	0,75;0,4b	4,0,4,0	0,115;0,73	52,75;34,4b
R—arrow6	0,77;0,4d	6,0,6,0	0,116;0,74	54,77;36,4d
U—arrow8	0,72;0,48	8,0,8,0	56,72;38,48	56,72;38,48
D—arrow2	0,80;0,50	2,0,2,0	50,80;32,50	50,80;32,50
Home7	0,71;0,47	7,0,7,0	0,119;0,77	55,71;37,47
End1	0,79;0,4f	1,0,1,0	0,117;0,75	49,79;31,4f
PgUp9	0,73;0,49	9,0,9,0	0,132;0,84	57,73;39,49
PgDn3	0,81;0,51	3,0,3,0	0,118;0,76	51,81;33,51
Ins0	0,82;0,52			48,82;30,52
Del.	0,83;0,53			46,83;2e,53
* PrtSc	42,55;2a,37			0,114;0,72
Num—	45,74;2d,4a			45,74;2d,4a
Num+	43,78;2b,4e			43,78;2b,4e
Esc	27,1;1b,1			27,1;1b,127,1;1b,1
Back—S	8,14;8,e			127,14;7f,e 8,14;8,e
Tab	9,15;9,f			0,15;0,f
Enter	13,28;d,1c			13,28;d,1c
Space	32,57;20,39	32,57;20,39	32,57;20,39	32,57;20,39
F1	0,59;0,3b	0,104;0,68	0,94;0,5e	0,84;0,54
F2	0,60;0,3c	0,105;0,69	0,95;0,5f	0,85;0,55
F3	0,61;0,3d	0,106;0,6a	0,96;0,60	0,86;0,56
F4	0,62;0,3e	0,107;0,6b	0,97;0,61	0,87;0,57
F5	0,63;0,3f	0,108;0,6c	0,98;0,62	0,88;0,58
F6	0,64;0,40	0,109;0,6d	0,99;0,63	0,89;0,59
F7	0,65;0,41	0,110;0,6e	0,100;0,64	0,90;0,5a
F8	0,66;0,42	0,111;0,6f	0,101;0,65	0,91;0,5b
F9	0,67;0,43	0,112;0,70	0,102;0,66	0,92;0,5c
F10	0,68;0,44	0,113;0,71	0,103;0,67	0,93;0,5d
F11	0,133;0,85	0,139;0,8b	0,137;0,89	0,135;87
F12	0,134;0,86	0,140;0,8c	0,138;0,8a	0,136;88
* /				

利用此程序并按提示击相应键后,就可获输出。用 TC 读入 M.C 并稍加编辑后便可得到一张详细的键码表。此表对程序设计或程序运行时估量键操作都是十分重要的。

对表 33—4 说明如下:

1. 码按“10 进制第一字节, 第二字节;16 进制第一字节, 第二字节”方式输出;
2. 当第一字节为 0 时,第二字节称为【扩充码】(也有人称它为【扩展码】)。
3. 当第一字节不为 0 时,第一字节中的值为键的 ASCII 码,第二字节中的值为键的位置码。例如,键盘上两个加号的码为

43,13;2b,d(按 Shift+= 键)和 43,78;2b,4e(小键盘上的加号键)它们的 ASCII 码

相同,但位置码不同。

4. 对有些键盘上功能键 F11 和 F12 的扩充码为

F11	0,133;0,85	0,139;0,8b	0,137;0,89	0,135;0,87
F12	0,134;0,86	0,140;0,8c	0,138;0,8a	0,136;0,88

但有些DOS 并不一定支持;

5. 为从码看键,你可以对这些值排序后生成另外的表;

6. 用上述程序还可测试:按 Alt—小键盘上的数字(Num 键在 On 状态),第一字节可得到 ASCII 码,第二字节为 0。其实,在使用 TC 编辑器时,例如你按了 Alt—6—5 键(按数字时一直按住 Alt 键,键松开后屏幕上显示字符 A。在 DOS 或刚进入 CCDOS 时你也可以这样做。事实上,第一字节能接收的 ASCII 值为

数字值 % 256

即 1~255,不包括 ASCII 码 0,即按 Alt—0Ins 键不会产生作用,也不会产生 ASCII 码 0。

5. 注意:有些应用程序为提高速度,或争夺【链头位置】(例如,都是通过修改中断 INT 9H 的几个不同的常驻程序装进内存后,它们的中断 INT 9H 处理程序将按照装入的先后次序连成一条链。链头是最后装入的常驻程序,链尾是由 ROM BIOS 提供的原始的中断 INT 9H 处理程序。链头位置是最有利的位置),可能绕过 BIOS 或 DOS 直接从键盘缓冲区中取字符,因此某些键可能丧失预定的功能。

33.1.7 程序中定义键值的方法

对键值常用的方法是

1. 用 `c=getch()`; 或 `c=getche()`; 取一个字符;
2. 用 `c=bioskey(0)`; 同时取扩充码和字符,如程序 K7.C 所示。有三种情况:
 - (1)同时取高位和低位判断;
 - (2)只取低位判断字符;
 - (3)只取高位判断。

实际上常用宏定义键值,直观明了。例如对 ESC 键可用两种定义

```
#define KEY—ESC    0x011b
#define U—arrow    0x4800
#define RETURNKEY  0x1c0d
#define ESC        27
```

以及

```
#define F1          315
#define F1KEY       (256+59)
#define ASCSPACE    0x20
#define BS          8
```

等等。

```
C>TYPE K7.C
#include "stdio.h"
#include "bios.h"
```

```

typedef struct keyvalue{
    int key;
    int lowbit;
    int highbit;
};

main()
{
    struct keyvalue u;
    getkey(&u);
}

int getkey(struct keyvalue *v)
{
    v->key=bioskey(0);
    v->lowbit=v->key & 0x00FF;
    if(v->lowbit == 0)
        v->highbit=((v->key & 0xFF00) >> 8)+256;
    else v->highbit=(v->key & 0xFF00) >> 8;
}

/* 按 F1 键后得调试表达式的值:
    v[0],rx: { key:0x3B00, lowbit:0x0, highbit:0x13B }
    v[0],r : { key:15104, lowbit:0, highbit:315 }
    或
    u,r : { key:15104, lowbit:0, highbit:315 }
    u: { 15104, 0, 315 }

    按 A 键后得调试表达式的值:
    v[0],rx: { key:0x1E41, lowbit:0x41, highbit:0x1E }
    v[0],r : { key:7745, lowbit:65,highbit:30 }
    或
    u,r : { key:7745, lowbit:65,highbit:30 }
    u: { 7745, 65, 30 } */

```

33. 1. 8 键盘中断

一 中断 INT 16H 的功能

INT 16H 的调用方式是:

对汇编	对 Turbo C
MOV AH,功能号	union REGS r;
INT 16H	r.h.ah=功能号;
	int86(0x16,&r,&r);

1. 功能 0H

取键击值。如键盘缓冲区内有字符或有键击,就把字符放入 AL 中。返回: AH=扫描码, AL=ASCII 码字符。如果 AL(即字节的低八位)非 0,则字符为 ASCII 码字符(ASCII 码 <128);如果 AL 为 0,则 AH(高八位)为扩充码(或称扩展码)。

biokey(0) 即相当此功能。

2. 功能 1H

检查键盘缓冲区中是否为空。如果缓冲区是空时,则置标志寄存器 flags(见 dos.h 中结构 WORDREGS) 的位 6(零标志,用 ZF 表示) 为 1; 否则为 0,表示有字符等待,这时缓冲区最前面的字符被放在 AX 中,字符本身并没有出缓冲区(首指针未变)。

函数 biokey(1) 相当此功能。

```
C>TYPE K8.C
#include "dos.h"
main()
{
    union REGS r;
    int ch;
    while(2)
    {r.h.ah=1;                /* 执行 INT 16H 子功能 1 */
     int86(0x16,&r,&r);
     if(r.x.flags & 0x40)      /* flags=1 继续循环 */
        {int86(0x28,&r,&r);
         continue;
        }
     r.h.ah=0;                /* flags=0 表示已按下一键 */
     int86(0x16,&r,&r);         /* 上次按键还保留(在 AX 中),不再 */
     if(r.h.al==0)ch=r.h.ah|128; /* 等待按键,立即执行下面的语句 */
     else ch=r.h.al;          /* AL=0 表示扩充码,同时进行处理 */
     printf("ch=0x%x %c\n",ch,ch); /* 注意,输出扩充码和实际的差异 */
     break;                  /* 结束循环 */
    }
}
```

ROM BIOS 的中断 INT 28H 是由 DOS 内部使用的,当 DOS 处于安全状态或“空”状态时, DOS 就调用 INT 28H。在操作系统的运行过程中,某些程序片断要求绝对不允许打断(例如,在进行大量数据文件输入输出时就不允许打断,否则很可能丢失数据),这样的程序片断称为【关键片断】。在关键片断内部, DOS 绝不会去调用中断 INT 28H。利用这个特性,常驻内存程序就可把握时机,在允许打断 DOS 的时刻安全地使用 DOS 调用(对常驻内存程序,最好是在打入热键并 DOS 又处在安全状态时才激活它)。

```
C>TYPE K9.C
#include "dos.h"
main(){
    int x,y;
    x=biokey(1);              /* 单步跟踪时并不要求按键 */
    y=biokey(0);              /* 要求按键,例如按了 A 键 */
    printf("%d %x\n",x,y);    /* 0 1e41 */
}
/* —FLAGS>>6,x: 0xA      逐次调试表达式
   —FLAGS,x: 0x292
   —AX,x: 0x0
```



```

--FLAGS>>6,x: 0xA
--FLAGS,x: 0x292
--AX,x: 0x1E41
--FLAGS>>6,x: 0xA
--FLAGS,x: 0x292
--AX,x: 0x7
*/

C>TYPE K10.C
#include "dos.h"
main(){
int x,y;
while((x=bioskey(1))!=0); /* 单步跟踪时,按一键,例如 A 键退出循环 */
y=bioskey(0); /* 假定按了 M 键 */
printf("%x %x\n",x,y); /* 1e41 324d */
}
/* --FLAGS>>6,x: 0x8 逐次调试表达式
--FLAGS,x: 0x206
--AX,x: 0x1E41
--FLAGS>>6,x: 0xA
--FLAGS,x: 0x292
--AX,x: 0x324D
--FLAGS>>6,x: 0xA
--FLAGS,x: 0x292
--AX,x: 0xA
*/

```

比较程序 K9.C 和 K10.C 便可知道,当不断执行 bioskey(1) 语句中如按了一个键,该键值将进入缓冲区,并把它放在 AX 寄存器中,且 AH 和 AL 中内容和执行 bioskey(0) 一样。

3. 功能 2H

检查状态字节 0040:0017,返回 AL = 移位状态 (参见表 33-2)。函数 bioskey(2) 相当此功能。

二 和键盘相关的中断 INT 21H 的功能

现将和键盘有关的几个功能简述于下,供参考。

对于功能 01H、06H、07H、08H、0AH 和 0BH 与等待按键、屏幕回显、对 Ctrl-Break、Ctrl-C 检查之间的关系见表 33-5。

表 33-5 INT 21H 几个功能的比较

功能	等待	回显	Ctrl-Break	Ctrl-C
01H	Yes	Yes	Yes	Yes
06H	No	No	No	No
07H	Yes	No	No	Yes
08H	Yes	No	Yes	Yes
0AH	Yes	Yes	Yes	Yes
0BH	No		Yes	Yes

Ctrl-Break 与 Ctrl-C 是不同的,前者是键盘中断处理例程的一部分,后者是由 DOS 功能调用本身所执行的。

1. 功能 01H

当键盘缓冲区为空时,它等待键入字符,并把接收到的字符送到屏幕上光标所指示的地方(对 DOS 2.0 以上版本, I/O 设备可重定向)。当接到 Ctrl-Break 时,转去执行 Ctrl-Break 例程(执行 INT 23H)。否则,如果接收到的是 ASCII 码,就放到 AL 中;否则(第一字节为 0),再次执行中断 INT 21H,使第二字节送 AL,等等。

```
MOV AH,1      ; 取第一字节
INT 21H       ; 设中断功能号
CMP AL,0      ; 等待键入字符
JE EXTENDED-CODE AL = 0 时表明是扩充码,转向 EXTENDED-CODE
                ; 处理扩充码的 EXTENDED-CODE 例程
INT 21H       ; 再次中断,等待第二个字节送 AL
CMP AL,77     ; 检查是否为 R-arrow 键
                ; (以下略)
```

注意:对功能 06H、07H 和 08H,扩充码均要两次功能调用。

2. 功能 06H

调用本中断时如将 DL=FFH,缓冲区被检查,如没有字符,零标志 ZF=1 且马上返回;如果缓冲区有字符表明键盘先前有输入,则处理字符(AL 中返回键入字符)并清零标志(ZF=0)。不处理 Ctrl-Break 或 Ctrl-C。如果 AL=00H,则用户按了一个具有扩充键编码的键,将由该功能的第二次调用返回扩充码。调用中断时如将 DL 中置除 FFH 外字符,则存于 DL 中的字符在当前光标位置上显示。

3. 功能 07H

当缓冲区空时等待键入字符。当一个字符到达缓冲区时,不把字符送屏幕。返回时 AL 中为键入字符。不处理 Ctrl-Break 或 Ctrl-C 字符。

4. 功能 08H

基本同功能 7,但它处理 Ctrl-Break 或 Ctrl-C 字符(若发现则执行中断 INT23H)。

5. 功能 0AH

该功能允许向用户设置的输入缓冲区(不是键盘缓冲区)连续输入多至 254 个字符,直到按下回车键为止(允许空串,即未键入任何字符便按了回车键)。输入的字符被送向屏幕。字符是一个接一个在屏幕显示,当某个字符到达屏幕底部右下角时,屏幕向上滚动一行。允许用退格键或光标左移动键在屏幕上及内存中删除不显示的字符;使用 Tab 键不会产生扩充码。如输入的字符个数超出了输入缓冲区能存放的最大字符数(第 1 字节值-3),则超出部分丢失,系统蜂鸣告警。

```
STRING DB 53 DUP(?) ; 指定输入缓冲区,空 53 个字节,最多放 50 个字符
                ; 第 1 字节存放输入缓冲区能保存的最大字节数(如
                ; 它为 0 则调用不读入任何输入字符就马上返回)
                ; 分配给串的字节数 50+1,50 指 50 个字符,1 指回车符
                ; (ASCII 码 13),即接收到的最后一字符,它一定是回车符。
                ; 第 2 个字节存放接收到的字符数(不包括回车符)。
```

	;	从第 3 字节开始,是接收的字符
LEA DX,STRING	;	设内存地址。DS:DX 给出了串在内存中地址
MOV BX,DX	;	BX 中放串地址
MOV AL,51	;	AL 中放第 1 字节值,不能小于 1 (50 个字符加 1
	;	个回车符)
MOV [BX],AL	;	将值放入第 1 字节
MOV AH,0AH	;	设中断号
INT 21H	;	接收字符
MOV AH,[BX]+1	;	返回实际读入字符数(不包括回车符,放在AH 中)

6. 功能 0BH

检查标准输入状态。如果输入的字符是可用的,或者说缓冲区中有字符,返回时 AL=FFH,否则 AL=00H。它还检查 Ctrl-Break 或 Ctrl-C,如有则执行中断 INT 23H。

7. 功能 0CH

在 AL 中设置功能号 01H、06H、07H、08H 或 0AH,中断执行时先清除键盘缓冲区,然后执行这些功能。它迫使系统等待,直到打入字符为止。

MOV AH,0CH	;	选 DOS INT 21H 功能 0CH
MOV AL,1	;	选功能号
INT 21H	;	清键盘缓冲区,等待输入字符
	;	Turbo C 的 bdos(0x0C,0x00,0x02);只清键盘缓冲区

8. 功能 3FH

这是一个通用的输入功能,大都用于磁盘操作,但也可以用于键盘输入。使用它要求预先定义一个寄存器,用来存放操作系统调用的 I/O 设备编号(对流而言,它是 stdin),键盘的编号是 0(设备文件句柄),它放在 BX 中。DS:DX 放键入字符串的分配地址, CX 中放字符串长度,然后调用中断。读入的字符串会在屏幕上显示。

MOV AH,3FH	;	置中断功能号
MOV BX,0	;	置设备号,这里是键盘
LEA DX,STRING-BUFFER	;	DS:DX 指示串缓存地址
MOV CX,100	;	设串长度,包括两个终止符(0xD 和 0xA)
INT 21H	;	等待输入

当接收到回车符时,字符串输入结束,DOS 自动在键入的回车符(0xD)后自动加上一个换行符(0xA)。因此,若设 CX=100,则字符缓存区真正能接收并存储的字符为 98 个,最后两个字符为回车和换行符。输入字符串的长度(小于或等于 CX 中的值)将放在 AX 中返回。

```
C>TYPE K11.C
#include "dos.h"
main(){
static char a[80];
char *p=a;
union REGS r;
r.h.ah=0x3f;
r.x.bx=0; /* 如下句写成 r.x.dx=p; 将发生Non-portable pointer assignment */
/* 警告,可强行抑止,即不管它,也可得同样结果 */
}
```

```

r.x.dx=FP-OFF(p); /* 获指针 p 地址偏移量 */
r.x.cx=5; /* 最多只能键入 3 个字符,超出时结果不可预料 */
int86(0x21,&r,&r);

/* —AX: 5 调试表达式
a: "ABC\r\n" 输入 ABC 后 */
printf("%s\n",p); /* 输出 ABC 后换 2 行 */
}

```

三 库函数 kbhit()

TC 的库函数 kbhit() 便相当于中断 INT 21H 的功能 0BH。它的原型在标头文件 conio.h 中。

```
—2 int —Cdecl kbhit (void);
```

它并不等待键击,它只是检查在执行此函数前有否按过键,如果按了能产生 ASCII 码 的键(像光按 Caps 键,函数 getch() 对此不予理睬),函数返回非 0 整数。键值可为 getch() 或 getche() 函数读取。如无键击,返回 0。程序 K12.C 和 K13.C 很好地说明了这一点。

```

C>TYPE K12.C
#include "dos.h"
main()
{
int i;
while(2) /* 击除Ctrl、Alt、Caps、Num、*/
/* Scroll Lock 键之外的键,循环终止 */
{
if (kbhit()!=0) /* 本句用 if (bioskey(1)!=0); 效果一样 */
{
printf("exit\n");exit(1);
}
printf("process\n");
}
}

```

```

C>TYPE K13.C
#include "dos.h"
#define AB a=peek(0x40,0x1a);b=peek(0x40,0x1c); /* 取缓冲区首尾指针 */ \
printf("a=%x b=%x\t",a,b)
#define SA poke(0x40,0x1a,0x28);poke(0x40,0x1c,0x28) /* 置首尾指针相同 */ \
#define SB poke(0x40,0x1a,0x1e);poke(0x40,0x1c,0x28) /* 置首尾指针不同 */ \
#define KBHIT AB; /* 取指针值输出 */ \
c0=kbhit(); /* 测试缓冲区 */ \
printf("kbhit=0x%x\t",c0); /* 如空,返回 0,否则返回非 0 整数 */ \
if(c0>31 && c0<127)printf("=%c",c0); /* 输出返回值的 ASCII 码 */ \
printf("\n");
#define S4 poke(0x40,0x1e,0x41); /* 强行填缓冲区,001E 填字母 A */ \
poke(0x40,0x20,0x42); /* 0020 填字母 B */ \
poke(0x40,0x22,0x43); /* 0022 填字母 C */ \

```

```

                poke(0x40,0x24,0xd)    /*          0024 填回车符  */
main(){
int c0,c;
unsigned int a,b;
SA;
KBHIT;
getch();
SB;
S4;
KBHIT;
c=getch();
printf("c=%x %c\n",c,c);
}

```

假定 C 盘当前目录里除有 K13.EXE 外,还有一个编译好的文件 ABC.EXE,其源文件为

```

C>TYPE ABC.C
main(){
printf("TEST OK ! \n");
}

```

可分三种情况执行程序：

1. 在命令行键入

```
C>K13
```

后再键入一个普通键,例如键 M,则有

```

a=28 b=28 kbhit=0x0
a=1e b=28 kbhit=0xffff
c=41 A
C>BC
Baf command or file name

```

程序在结束后自动去执行名为 BC.EXE 的程序,显示表明目录中无此程序。

2. 在命令行键入 C>K13 后再键入一个特殊键,例如功能键 F10,则有

```

a=28 b=28 kbhit=0x0
a=1e b=28 kbhit=0xffff
c=44 D
C>ABC
TEST OK !
C>A

```

程序在结束后自动去执行名为 ABC.EXE 的程序,并获成功。结束 ABC 的执行后返回 DOS,并在提示符后显示字母 A。

3. 在命令行键入 C>K13 后再键入按 Ctrl-Break 键(或 Ctrl-@ 键),则有

```

a=28 b=28 kbhit=0x0
a=1e b=28 kbhit=0xffff
a= ^ C

```

```
C>ABC
TEST OK ;
C>
```

这表明检查到 Ctrl-Break 后,终止现行程序,转去执行缓冲区指明的程序。

如果把语句

```
poke(0x40,0x24,0xd) /* 0024 填回车符 */
```

改成

```
poke(0x40,0x24,0x44) /* 0024 填字母D */
```

缓冲区中没有回车符,则程序 ABC.EXE 便不会被自动执行(但在 DOS 提示符后显示出文件名)。

如果用户在程序开始运行时使用屏幕、打印机或异步通讯接口,并且键入了 Ctrl-Break,那么系统将立即转去执行 ROM BIOS 的中断 INT 23H。

四 中断 INT 23H 和键盘相关的功能

它在按键 Ctrl-Break 后被调用,它控制断开例程的出口地址。如果控制断开子程序保存断开前的所有寄存器,则可用中断指令 IRET 返回主程序并继续执行;如果程序使用远程返回,进位标志便被用来决定是否要撤消程序执行。若进位标志置位,则撤消,否则继续执行(如果 IRET 返回也同样)。如果控制中断了缓冲区的 I/O 功能调用 9H、10H 或 CH,那么就输出一个回车换行。如果用 IRET 来继续执行,I/O 将从该行开始继续执行。当中断发生时,所有寄存器置成原来对 DOS 进行功能调用时的值。只要对控制断开子程序去做什么没有限制,也可以进行 DOS 功能调用及使用 IRET,但所有寄存器的值不能改变。

如果一个程序建立了一个新段并装入了第二个程序,第二个程序段改变了 Ctrl-Break 地址,则第二个程序结束后返回到第一个程序,Ctrl-Break 地址也恢复到执行第二个程序之前的值(这可从第二个程序段前缀内恢复)。

DOS 命令中的 BREAK 命令使用时用

BREAK OFF

则只对屏幕、打印机或异步通讯(包括键盘)进行检测 Ctrl-Break,否则设成

BREAK ON

时对所有功能每次调用时都要检测 Ctrl-Break(这样可能使程序执行速度变慢)。可以将这个命令放到 AUTOEXEC.BAT 或 CONFIG.SYS 文件中去。

TC 在 dos.h 标头文件中有两个函数就是用来设置 BREAK 状态的(参见《中断和中断函数》一章):

```
int —Cdecl getcbrk(void);
int —Cdecl setcbrk(int cbrkvalue);
```

当检测到 BREAK OFF 时,getcbrk() 返回 0,否则返回 1;setcbrk() 用来设置 BREAK 状态,当 cbrkvalue 的值为 0 时,setcbrk() 置 BREAK OFF,为 1 时置 BREAK ON 状态。

```
C>TYPE K14.C
#include "dos.h"
```

```

int get—set—cbrok(int int—n,int cbrkvalue) /* 输 int—n=0 为 getcbrk */
{
    /* 输 int—n=1 为 setcbrk */
    —AX=0x3300+int—n;
    if (int—n!=0)—DL=cbrkvalue; /* int—n=1 时置值 */
    geninterrupt(0x21); /* 发生中断 */
    return (—DL); /* 返回寄存器 DL 中值 */
}

main()
{ int c1,c2=0,n;
  printf("input INT 33—? \n"); /* 输入中断 33 的 AL 值 */
  c1=getch(); /* getch 获得一字符 */
  printf("%d\t",c1);
  if(c1=='1'){c1=1;
    printf("cbrkvalue=? \n");
    c2=getch();
    if(c2=='1')c2=1;
    else c2=0;
    printf("c2=%d\t",c2);
  }
  else c1=0;
  n=get—set—cbrok(c1,c2);
  if(n==0)printf("n=0; BREAK OFF\n"); /* 判断 BREAK 状态 */
  else printf("n=1; BREAK ON\n");
}

```

五 有关键盘的其它一些信息

内存中

0040:0019 字节用于按 Alt—小键盘数字键时暂存 ASCII 码

0040:0071 表示 Break 键状态

0040:0072 ~ 0040:0073 复位标志

0040:0080 ~ 0040:0081 键盘缓冲区数据区头指针,一般为 1eH

0040:0082 ~ 0040:0083 键盘缓冲区数据区尾后一指针,一般为 3eH

0040:0096 键盘方式状态和类型标志

0040:0097 键盘 LED 灯标志。例如,位 0 表示 Scroll Lock、位 1 表示 Num、位 2 表示

Caps 状态

注意:对不同的操作系统和键盘它们可能不同。

C>TYPE K15.C

```

#include "dos.h"
#define PK(X) peek(0x40,X)
#define PB(Y) peekb(0x40,Y)
int return—value=0;
int control—break(void)
{int c471=PB(0x71);
  printf("Control break TEST 1 \n");
  printf("c471=%x\n",c471);
}

```

```

return (return—value);
}
main()
{unsigned char b419=PB(0x19),b471=PB(0x71),b496=PB(0x96),b497=PB(0x97);
int b472473=PK(0x72),b480481=PK(0x80),b482483=PK(0x82);
char ch,n=0;
printf("Input return—value:");
scanf("%d\n",&return—value);
while((ch=getche())!=26)
{
    ctrlbrk(control—break);
    if(n==0){setcbrk(1);n=1;}
    else {setcbrk(0);n=0;}
    printf("ch=%c\n",ch);
    printf("b419=%x b471=%x",b419,b471);
    printf(" b496=%x b497=%x\n",b496,b497);
    printf("b472473=%x ",b472473);
    printf("b480481=%x b482483=%x\n",b480481,b482483);
}
}

Input return—value:0 /* 输入数字 0 */
a
ach=a
b419=0 b471=0 b496=0 b497=10
b472473=1200 b480481=1e b482483=3e /* 不按 Ctrl—Break 键便执行语句 */

/* ctrlbrk(control—break); 的下一语句,一按 Ctrl—Break 键 */
^ C /* (或 Ctrl—@ 键)便立即转去执行其指 */
Control break TEST ! /* 出的 control—break 函数(子程序) */
c471=0 /* 然后程序终止执行 */
Input return—value;1 /* 重新开始执行,输入数字 1 */
a
ach=a
b419=0 b471=0 b496=0 b497=10
b472473=1200 b480481=1e b482483=3e
^ C
Control break TEST !
c471=0 /* 执行完 control—break 后接着执行 */
ch= /* main 函数的程序 */
b419=0 b471=0 b496=0 b497=10
b472473=1200 b480481=1e b482483=3e
Ach=A
b419=0 b471=0 b496=0 b497=14
b472473=1200 b480481=1e b482483=3e
/* 按 Ctrl—Z 退出运行 */

```

注意: Ctrl—Break 不支持重定向,即如用 C>K15>M.C 当按 Ctrl—Break 后,文件

M. C 中没有任何东西。比较上述 `ctrlbrk()` 函数与程序 K12. C 中 `ctrl-brk()` 函数的异同。

```
C>TYPE K16.C
#include "dos.h"
void interrupt mykbint()
{printf("TEST OK! \n");}
void ctrl-brk(void interrupt (* fptr)())
{ setvect(0x23,fptr);}
main(){
printf("+++++\n");
getch();
ctrl-brk(mykbint);
printf("=====\n");
}
/* ++++++
   = ^C          等待输入时按了 Ctrl-Break 键
   =====
*/
```

33.1.9 应用

例1 将文本文件在屏幕上显示

```
C>TYPE K17.C
#include "stdio.h"
main(){
int c;
while((c=getch())!=EOF+27) /* 只有按 Ctrl-Z 键才能退出循环,结束运行 */
putchar(c);                /* 定义的宏,将字符输出到屏幕 */
}
```

—3 int —Cdecl getch (void);

函数 `getch()` 是等待从标准输入设备读入一个字符,返回时该字符放在 AL 中, AH 为 0,但此字符不会在屏幕显示,系统也不对字符作是何种字符的检查,或者说,它就是中断 INT 21H 的功能 7。在程序中,常将此函数作为“暂停,按任一键继续运行程序”用。

由于它是无参函数,同时它又返回整型,所以在调用它时也可不包含标头文件 `conio.h`。

在标头文件 `stdio.h` 中定义了一个常量 `EOF`,其值等于 `-1`,它被用来作为文本文件的结束标志。但从前面键扫描码看出,没有按键会产生 `-1` 的(但在其它 C 语言中的函数 `getch()`,可能将 `Ctrl-Z` 处理成 `-1`);其次,文本文件的结束标志是 ASCII 码 26(1AH)。因此,对于 TC 而言,使用像

```
while((c=getch())!=EOF)
```

这样的语句是有问题的。这是因为 `-1=0xFFFF`,为两个字节,而 `c` 是 `char` 型,只一个字节,所以不管键入什么值,`c` 是不可能等于 `-1` 的。可用 `EOF+27=26` 来判别,否则由于不断循环而死机。

上述程序可单独运行,也可用 DOS 的重定向功能即输入命令行 `C>K17<K17.C` 将源文件 K17.C 的内容在屏幕输出。

虽然 `getch()` 的原型在 `conio.h` 中, 它从键盘缓冲区取一字符并回显。

在 `conio.h` 中还定义了另一个键盘输入函数

—4 int —Cdecl `getche(void)`;

其功能和 `getch()` 基本相同, 唯一不同之处是取得的字符将在屏幕上显示出来 (称【回显】)。

例2 使用四个键作热键

下列程序定义 4 个热键是这样操作的, 例如, 将程序编译后打入命令行 `C>K18 □ALL`, 然后一只手先按住 `Ins` 键和右 `Shift` 键, 另一只手按住 `Caps` 键后再按 `A` 键。这四键组成一个乒乓开关, 假定奇数次按封锁了整个键盘, 这意味着此时你按任意键都不起作用, 包括按热启动键 (`Ctrl-Alt-Del`); 再将此开关启动一次, 则恢复到以前未封锁状态。这种封锁键盘的功能有时对用户是有用的。例如利用这种原理来防止非法拷贝 (软件一运行便被封锁键盘)。

程序另外两个功能是按 `Ins-R-Shift-Caps-B` 键来封锁热启动; `Ins-R-Shift-Caps-C` 来封锁 `Ctrl-C` 开关。

本程序结尾对 8255 的端口 A 和端口 B 的处理特别重要。注意: 程序中不宜加入像 `printf()` 那样的函数, 否则会影响事件的处理。

```
C>TYPE K18.C
#include "dos.h"
#define tsr—size 400
void interrupt (* doskbint)();
void interrupt (* int0x80)(), interrupt (* int0x7f)();
void interrupt mykbint();
unsigned mesp, mess, kb—ss, kb—stack, invalue;
#define IF—STR(M,N,P) if(strcmp(argv[i], #M)==0 || strcmp(argv[i], #N)==0)\
    {P=1; continue;}
#define IF—GOTO—KB(X,Y) if((invalue==X) && \
    (((kb417 & kbmask1)^ kbmask1)==0) && (((kb418 & kbmask2)^ kbmask2)==0))\
    {Y=1-Y; flag=1; goto kb;}
unsigned letter—A=30; /* A 键码 */
unsigned letter—B=48; /* B 键码 */
unsigned letter—C=46; /* C 键码 */
unsigned letter—Del=83; /* Del 键码 */
unsigned kbmask1=0x1; /* 判别右 Shift 键是否为 On */
unsigned kbmask2=0xc0; /* 判断 Ins, Caps 是否为 On */
unsigned kbmask3=0x4; /* 对 Ctrl */
unsigned kbmask4=0xc; /* 对 Ctrl 和 Alt */
static union REGS reg;
static struct SREGS segm;
int flag=0, kb417, kb418, aa=0, bb=0, cc=0; /* 根据马文骞的程序改写并注释 */
main(int argc, char * argv[]) /* 注意: 未对命令行参数作检查 */
{int i;
 int0x7f=getvect(0x7f); int0x80=getvect(0x80);
 doskbint=getvect(0x09);
 for(i=1; i<=argc-1; i++)
```

```

{
IF—STR("all", "ALL", aa);
IF—STR("boot", "BOOT", bb);
IF—STR("ctr—c", "CTR—C", cc);
if ((strcmp(argv[i], "stop") == 0 || strcmp(argv[i], "STOP") == 0) &&
    int0x7f! = 0 && int0x80! = 0)
    {disable(); setvect(0x09, int0x80);
    setvect(0x80, 0); enable(); return(0);
    }
if ((strcmp(argv[i], "restart") == 0 || strcmp(argv[i], "RESTART") == 0) &&
    int0x7f! = 0 && int0x80 == 0)
    {disable(); setvect(0x80, doskbint);
    setvect(0x09, int0x7f); enable(); return(0);
    }
}
if(int0x7f! = 0){sound(1200);sleep(1);nosound();return(0);}
segread(&segm); kb—ss = —SS;
disable();
setvect(0x80, doskbint);
setvect(0x09, mykbint);
setvect(0x7f, mykbint);
enable();
kb—stack = (tsr—size - (segm.ds - segm.cs)) * 16 - 300;
reg.x.ax = 0x3100; reg.x.dx = tsr—size;
intdos(&reg, &reg);
}

void interrupt mykbint() /* 中断函数 */
{
disable(); mesp = —SP; mess = —SS; —SP = kb—stack; —SS = kb—ss; enable();
flag = 0; inval = inportb(0x60); /* 从端口 A 读取码值 */
kb417 = peekb(0, 0x417); kb418 = peekb(0, 0x418); /* 取状态字节值 */
IF—GOTO—KB(letter—A, aa)
IF—GOTO—KB(letter—B, bb)
IF—GOTO—KB(letter—C, cc)
if((inval == letter—C) && (((kb417 & kbmask3) ^ kbmask3) == 0))
    {flag = cc; goto kb;}
if((inval == letter—Del) && (((kb417 & kbmask4) ^ kbmask4) == 0))
    {flag = bb; goto kb;}
flag = aa;
kb; /* 向键盘回送应答信号方法 */
disable(); —SP = mesp; —SS = mess; enable();
if(flag! = 0){
kb418 = inportb(0x61); /* 取端口 B 当前值保存 */
outportb(0x61, 0x80); /* 向端口 B 送原中断值 */
outportb(0x61, kb418); /* 本句和下句对硬件中断必须 */
outportb(0x20, 0x20); /* 否则不能保证中断优先级 */
}
}

```

```

return;
}
(* doskbint)();          /* 执行 INT 9H 中断的正常处理 */
}

```

例3 用 C 语言实现音乐的后台演奏

C>TYPE K19.C

```

#include "stdio.h"
#include "dos.h"
#define L1 1000
#define L2 L1/2
#define L4 L1/4
int HZ[4][7]={ {131,147,165,175,196,220,247}, {262,294,
330,349,392,440,494}, {523,587,659,698,784,880,988},
{1047,1174,1319,1397,1568,1760,1976}};
int *s;
int buf[100]={11,L1,12,L1,13,L1,14,L1,15,L1,16,L1,17,
L1,21,L1,22,L1,23,L1,24,L1,25,L1,26,L1,27,L1,31,L1,0};
void interrupt new—int9(),interrupt(*old—int9)();
/* 注意:上句不能写成 void interrupt new—int9(),(*old—int9)(); */
main()
{
play(buf);          /* 自定义演奏音乐函数 */
while(*s && ! bioskey(0)); /* 按任一键结束程序执行 */
nosound();          /* 关闭 PC 扬声器库函数 */
setvect(0x1c,old—int9); /* 恢复中断 0x1c 的原有值 */
}
play(int *ms)
{
s=ms;          /* 取缓冲区 buf 中的值 */
old—int9=getvect(0x1c); /* 保存中断 0x1c 当前 4 字节值 */
setvect(0x1c,new—int9); /* 设置中断 0x1c 新值 */
}          /* DOS 功能调用 */
void interrupt new—int9()
{
static int count=0,tt=0;
count++;
if(*s!=0)
{if(count>=tt)
(sound(HZ[*s/10][*s%10])/* 以频率 HZ[][] 打开扬声器,扬声器 */
s++;          /* 发声,频率以赫兹为单位 */
tt=*s*18.2/1000;s++;count=0;
}
}
else nosound();
(*old—int9)();
}

```

/* 单步跟踪,中途按键的调试表达式值

s, buf+6

* s; 14 */

例4 回退字符到键盘缓冲区

—5 int —Cdecl ungetch(int ch);

该函数将字符 ch 回退到键盘缓冲区,使其成为下一个待读字符。当下一次用 getch() 或其它控制台输入函数调用时,如果调用成功,将返回 ch, 否则返回 EOF。它只适用于 MS-DOS。

C>TYPE K20.C

```
#include "conio.h"
```

```
kbtest()
```

```
{
```

```
char c,ch,ch2;
```

```
/* char ch1; */
```

```
if(kbhit()) /* 如不按任一键,下面不被执行 */
```

```
{
```

```
c=getch();
```

```
ch2=kbhit();
```

```
printf("c1=%c ch2=%x\n",c,ch2);
```

```
ch=ungetch('MN'); /* 双字符只有前一个被接收 */
```

```
/* ch1=ungetch('P'); */ /* 两个连续的 ungetch() */
```

```
c=getch();
```

```
printf("c2=%c ch=%c\n",c,ch);
```

```
/* printf("ch1=%x\n",ch1); */
```

```
}
```

```
}
```

```
main(){
```

```
printf("=====");
```

```
delay(3000); /* 延迟时间模拟程序正在进行其它工作 */
```

```
kbtest();
```

```
printf("-----\n");
```

```
}
```

按键 Q 和 W 后程序输出:

```
/* =====c1=Q ch2=ffff
```

```
C2=M ch=M
```

```
-----
```

```
*/
```

```
if(kbhit())ch1=getch();
```

和语句

```
ch2=getch();
```

的区别在于,前者并不像后者要等待按键,而是在执行此语句前如果已按了一键,则此键入字符将赋给变量 ch1, 否则此 ch1=getch(); 语句将不被执行。

从 K20.C 我们可以得到几点结论：

1. kbbitt() 和 ungetch() 都不等待按键,它们直接检查缓冲区或对缓冲区作用；
2. 如将时间延迟可以看作程序正在进行某一工作,则此时按键值被存入缓冲区；
3. 如将注释中语句使用,则可发现 ch1=fiff,即值为 -1(相当于 stdio.h 中常量 EOF),这表明后一个 ungetch() 操作失败,而前一个 ungetch() 依然有效；
4. 执行 c=getch(); 语句后,缓冲区为空,ungetch() 返回字符到缓冲区头部。

33.2 鼠标

近年来,鼠标器(简称鼠标)几乎和键盘一样是一种重要的输入设备。鼠标允许你通过弹出菜单来快速选择命令,从而可以直观而迅速地运行程序。或者,也可以用它自由地手动绘图。

鼠标兴起的一个原因还在于它使用几乎是“标准”的 INT 33H 中断接口。本节将通过一个具体的程序介绍中断 INT 33H 的常用的几个功能。

一些软件直接支持鼠标,象 PCTOOLS,联想或金山系统的一些软件,只要你在运行这些软件前装入了鼠标驱动程序,进入后便可直接使用鼠标。但是有些软件则不支持。这时若要使用鼠标便要自己编写程序。

33.2.1 鼠标安装

常见的鼠标有 Microsoft 鼠标(两个按钮)和 Logitech 鼠标(三个按钮)。有些鼠标上有一个开关,控制按钮数。要使用 IBM 鼠标,一般要有 256KB 以上内存,一个 RS-232 串行接口,鼠标驱动程序 MOUSE.COM 和达到 EGA 或 VGA 分辨率的显示适配器。在进入 DOS 后要先运行 MOUSE.COM(鼠标常驻内存程序),即把鼠标驱动程序装入内存(也可以在 AUTOEXEC.BAT 增加一行

MOUSE

以便启动系统时自动将该文件装入内存);或者对 Microsoft 鼠标也可以在 CONFIG.SYS 文件中增加一行

DEVICE=MOUSE.SYS

当然,当前目录中应当有 MOUSE.SYS 文件。

下面给出光电式三键鼠标(A4 TECH)程序(MOUSE.COM)的启动语法,即指出如何使用命令行参数。在 DOS 提示符下键入(方括号内为任选项)

C>MOUSE [[/0-4][/Sp,i/X/P/B/B3]] [/D/T/Q/U/R0/Rn] [/H/?] [OFF]

这里选项

- /1-4 是设备口编号。指定编号后对 RS-232 端口使用串行鼠标时将自动配置。
- /3 对 COM3; (地址 03E8,IRQ4)
- /4 对 COM3; (地址 02E8,IRQ3)
- /Sp,i 使用 16 进制 I/O 口 p 和 10 进制 IRQ 数 i
- /X 改变鼠标方式为 3 按钮 PC 方式(只适于软开关鼠标)
- /P 在标示设备口(Pointing Device Port)用 PS2 和兼容鼠标时

■/B3	在总线鼠标口 (Bus Mouse Port) 用总线鼠标时
■/D	双精度动态分辨率 (X2)
■/T	三倍动态分辨率 (X3)
■/Q	四倍动态分辨率 (X4)
■/U	五倍动态分辨率 (X5)
■/R0	关闭鼠标动态分辨率 (X0)
■/Rn	设置鼠标动态分辨率倍数 (n=1~10)
■/0	从内存中移去鼠标驱动程序并释放内存
■/OFF	从内存中移去鼠标驱动程序并释放内存
■/H	获帮助信息
■/?	获帮助信息

当鼠标在移动、按钮按下或松开时,会不断产生 INT 33H 中断,鼠标驱动程序处理该中断,然后返回。鼠标在移动时,其光标并非直接显示在屏幕上,而是在一个虚拟屏幕上,以像素为单位移动,再映射到显示屏上。显示模式不同,屏幕上的像素个数不同,鼠标通过其驱动程序自动维护的一个计数器再逐点或隔点映射到实际屏幕上。

33.2.2 使用鼠标的演示程序

程序 MOUSE1.C 和 MOUSE2.C 在 TC 集成环境里在小模式下编译后运行,运行的环境是 DOS 5.0,联想 9.0 超级汉卡, SVGA 卡。鼠标为光电式三键鼠标。

程序 MOUSE1.C 没有用到 TC 的 graphics.h,编译时也不用 graphics.lib,从而不能直接使用 TC 的图形库函数。

程序 MOUSE2.C 则用了 TC 的 graphics.h。

注意:在汉字环境下编译调试时,尽量使用下拉式菜单,而少用或不用热键,以避免键操作的冲突。

```
C>TYPE MOUSE1.C
#include "dos.h"
#include "stdio.h"
#include "bios.h"
#include "conio.h"
#include "stdlib.h"

#define None 0    /* 未按鼠标按钮 */
#define Left 1   /* 按左按钮 */
#define Right 2  /* 按右按钮 */
#define Both 3   /* 同时按住左右按钮 */
#define True 1   /* 真 */
#define False 0  /* 假 */
#define linech 18 /* 一个字符垂直方向的像素值 */
#define colch 8   /* 一个字符水平方向的像素值 */

void mode();
void hzprint();
void let—menu();
void resetmouse();
void showorhidemscurs();
```

```

void getkeystatus();
void setscurpos();
void setgraphicscursor();
void settextrcursor();
void setmaxminlinepos();
void setmaxmincolpos();
void querybtndnup();
void hz();
void setmkstatus();
unsigned int anymkpressed();
unsigned int leftmkpressed();
unsigned int rightpressed();
unsigned int bothmkpressed();
typedef struct { /* 移动鼠标时鼠标驱动程序会自动在屏幕上显示一个图形光标, */
    /* 下述光标均指这种光标。可以在程序中重新设置它的形状 */
    int mak[2][16]; /* 存放光标形状数据的数组 */
    int horzh; /* 光标的水平方向的热点 */
    int vertb; /* 光标的垂直方向的热点 */
    }mkcurstype; /* 所谓热点 (hot spot) 可以和大家熟悉的 */
    /* 热键联系起来。热键对键盘而言, 热点对鼠 */
    /* 标而言。热点用于确定鼠标的坐标, 其值 */
    /* 一般在 -16 到 16 之间。当其值落入某 */
    /* 一值域时, 便可按程序设定的内容激活相 */
    /* 关菜单项 */

int maktext[2][16]={
    {0x3fff,0x1fff,0x0fff,0x07ff,0x03ff,0x01ff,0x00ff,0x00ff,
    0x00ff,0x00ff,0x01ff,0x10ff,0x30ff,0xf8ff,0xf8ff,0xfcff},
    {0x0000,0x4000,0x6000,0x7000,0x7800,0x7c00,0x7e00,0x7f00,
    0x7f00,0x7f00,0x7c00,0x4600,0x4600,0x0300,0x0100,0x0000}};

struct cmenu{
    int startx,starty,endx,endy; /* 菜单的起始和结束坐标 */
    char ** menu; /* 菜单的内容 */
    int count,len; /* 菜单的项数, 单项长度 */
    int H—V—flag; /* 菜单的排列标志。0 = 主菜单 */
    }hzmnu[5]; /* 非0 = 非主菜单 */

char * main—menu[]={
    " 光标形状 "," 前景颜色 "," 背景颜色 "," 其它 "
    };

char * formenu[]={
    " 黑 色 "," 兰 色 "," 绿 色 "," 青 色 ",
    " 红 色 "," 紫 色 "," 棕 色 "," 白 色 ",
    " 灰 色 "," 淡蓝色 "," 淡绿色 "," 淡青色 ",
    " 淡红色 "," 淡紫色 "," 黄 色 "," 亮白色 "
    };

char * backmenu[]={
    " 黑 色 "," 兰 色 "," 绿 色 "," 青 色 ",

```



```

        " 红    色 ", " 紫    色 ", " 棕    色 ", " 灰    色 "
    },
    char *othermenu[] = {"返    回", "结    束"};
    char *intrmenu[] = {"箭    头", "手    形", "十    字    形"};
    /* 在图形方式下,光标由屏幕掩码、光标掩码和热点构成。它们连续存放,  */
    /* 屏幕掩码在前,光标掩码随后,热点在最后。屏幕掩码将与屏幕上已有  */
    /* 象素进行与 (AND) 操作,光标掩码与屏幕上已有象素进行与或 (XOR)  */
    /* 操作,某种意义上说,光标掩码决定光标形状,而屏幕掩码决定其是否  */
    /* 在屏幕上显示出来。对于图形方式,掩码为位图块 (bit-mapped block)。 */
    /* 每个字包含了一行象素信息;它从顶行开始。光标大小一般有两种,8*16  */
    /* 或 16*16,前者每个象素为 2 位,后者为 1 位。                        */
    mkcurstype handcurs = {
        /* 前 16 个为屏幕掩码,后 16 个为光标掩码,最后为热点 */
        0xe1ff, 0xe1ff, 0xe1ff, 0xe1ff, 0xe1ff, 0xe000, 0xe000, 0xe000,
        0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
        0x1e00, 0x1200, 0x1200, 0x1200, 0x1200, 0x13ff, 0x1249, 0x1249,
        0x1249, 0x9001, 0x9001, 0x9001, 0xa711, 0x8001, 0x8001, 0xffff,
        0xffff, 0x0000};
        /* 0xf3ff, 0xe1ff, 0xe1ff, 0xe1ff, 0xe1ff, 0xe049, 0xe000, 0x8000,
        0x0000, 0x0000, 0x07fc, 0x07f8, 0x9ff9, 0x8ff1, 0xc003, 0xe007,
        0x0c00, 0x1200, 0x1200, 0x1200, 0x1200, 0x13b6, 0x1249, 0x7249,
        0x9249, 0x9001, 0x9001, 0x8001, 0x4002, 0x4002, 0x2004, 0x1ff8,
        0x0004, 0x0004}; */
    mkcurstype standshapecurs = {
        0x3fff, 0x1fff, 0x0fff, 0x07ff, 0x03ff, 0x01ff, 0x00ff, 0x007f,
        0x003f, 0x001f, 0x01ff, 0x10ff, 0x30ff, 0xf87f, 0xf87f, 0xfc3f,
        0x0000, 0x4000, 0x6000, 0x7000, 0x7800, 0x7c00, 0x7e00, 0x7f00,
        0x7f80, 0x7fc0, 0x7c00, 0x4600, 0x4600, 0x0300, 0x0180, 0x00c0,
        0xffff, 0xffff};
        /* 0x1fff, 0x0fff, 0x07ff, 0x03ff, 0x01ff, 0x00ff, 0x007f, 0x003f,
        0x001f, 0x003f, 0x01ff, 0x01ff, 0xe0ff, 0xf0ff, 0xf8ff, 0xf8ff,
        0x0000, 0x4000, 0x6000, 0x7000, 0x7800, 0x7c00, 0x7e00, 0x7f00,
        0x7f80, 0x7c00, 0x4c00, 0x0600, 0x0600, 0x0300, 0x0300, 0x0000,
        0x0001, 0x0001}; */ /* 注释内为另一种 */
    mkcurstype reccrosscurs = {
        0xfc3f, 0xfc3f, 0xfc3f, 0x0000, 0x0000, 0x0000, 0xfc3f, 0xfc3f,
        0xfc3f, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff, 0xffff,
        0x0000, 0x0180, 0x0180, 0x0180, 0xffff, 0x0180, 0x0180, 0x0180,
        0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
        0xffff, 0xffff};
        /* 0xf01f, 0xe00f, 0xc007, 0x8003, 0x0441, 0x0c61, 0x0381, 0x0381,
        0x0381, 0x0c61, 0x0441, 0x8003, 0xc007, 0xe00f, 0xf01f, 0xffff,
        0x0000, 0x07c0, 0x0920, 0x1110, 0x2108, 0x4004, 0x4004, 0x783c,
        0x4004, 0x4004, 0x2108, 0x1110, 0x0920, 0x07c0, 0x0000, 0x0000,
        0x0007, 0x0007}; */
    enum mkey {none, left, right, both};

```

```

unsigned int mouseX,mouseY,msvisible,mspresent,mskeynum;
enum mkey mousekey;
int msdelay=10,bkcolor,fcolor;
unsigned int mkip;
int vediomaxx,vediomaxy,vediominx=0,vediominxy=0;
int mbtnstatus,mopcount,mxpos,mypos;
main()
{
int menuselect,menu2select,done=1;
mkip=0,bkcolor=3,fcolor=15;
let—menu(0, main—menu,4,8,1);          /* 给菜单变量预先赋值 */
let—menu(1, intrmenu,3,8,2);
let—menu(2, formenu,16,23,2);
let—menu(3, backmenu,8,38,2);
let—menu(4, othermenu,2,53,2);
mode(18);          /* 分辨率 640×480 */
vediomaxx=1024;    /* 设置鼠标活动范围 */
vediomaxy=768;
directvideo=0;
hzprint(10,11,"根据邓永祥的程序改写并注释",fcolor);
hzprint(10,12,"按键盘上任一键后演示开始!",fcolor);
getch();          /* 在使用鼠标的同时也可用键盘,因它们激活不同的中断 */
resetmouse();
setmaxminlinepos(vediomaxy,vediominxy);
setmaxmincolpos(vediomaxx,vediominx);
do{
clr(0,1,17,79,bkcolor);    /* 清屏幕 1~17行 */
dispaly—menu(0);          /* 显示主菜单 */
menuselect=selectmenu(0);  /* 鼠标点主菜单 */
clr(2,1,24,79,bkcolor);    /* 清屏幕 2~24行 */
switch(menuselect){        /* 进入子菜单 */
case 0;dispaly—menu(1);    /* 显示子菜单 1 */
menu2select=selectmenu(1); /* 鼠标点子菜单 1 */
clr(23,1,24,70,bkcolor);  /* 清 23 和 24 行 */
switch(menu2select){      /* 进入下一级子菜单 */
case 0;setgraphicscursor(standshapecurs);
hzprint(10,24,"光标现在已经改变为箭头形状!",fcolor);break;
case 1;setgraphicscursor(handcurs);
hzprint(10,24,"光标现在已经改变为手形状!",fcolor);break;
case 2;setgraphicscursor(reccrosscurs);
hzprint(10,24,"光标现在已经改变为十字形状!",fcolor);break;
case -1;hzprint(10,24,"光标现在没有改变形状!",fcolor);break;
}break;
case 1;dispaly—menu(2);
menu2select=selectmenu(2);
clr(23,1,24,70,bkcolor);

```

```

        if(menu2select>=0){
            fcolor=menu2select;
            hzprint(10,24,"前景颜色已经改变,现在的前景是:",fcolor);
        }
        else
            hzprint(10,24,"前景颜色没有改变!,现在的前景色是:",fcolor);
        hzprint(45,24,formenu[fcolor],fcolor);
        break;
    case 2:dispalymenu(3);
        menu2select=selectmenu(3);
        clr(23,1,24,70,bkcolor);
        if(menu2select>=0){
            bkcolor=menu2select;
            setcolor(bkcolor);
            hzprint(10,24,"背景颜色已经改变!,现在的背景色是:",fcolor);
        }
        else
            hzprint(10,24,"背景颜色没有改变!,现在的背景色是:",fcolor);
        hzprint(45,24,backmenu[bkcolor],fcolor);
        break;
    case 3:dispalymenu(4);
        menu2select=selectmenu(4);
        switch(menu2select)
        {
            case 0:done=1;break;
            case 1:done=0;break; /* 退出本程序 */
        }break;
    }
}
if(done!=0)
{ hz(mbtnstatus,mopcount,mxpos,mypos,fcolor); /* 显示光标位置 */
  delay(1000); /* 延迟,以便观察 */
}
}while(done);
clr(0,0,24,79,bkcolor); /* 清 0~24 行 */
hzprint(14,20,"按键盘上任一键,演示结束!",fcolor);
getch();
mode(3); /* 返回文本方式.对金山汉卡可用 mode(1); */
}
void mode(int mode—code)
{ /* 设置显示方式 */
    —AH=0;
    —AL=mode—code;
    geninterrupt(0x10); /* 参见中断 INT 10H 的有关说明 */
}
clr(int x1, int y1, int x2, int y2, int at) /* 清屏幕指定区域 */
{

```

```

--AH=0x06;
--DH=x2;
--DL=y2;
--BX=at;
--AL=0;
--CH=x1;
--CL=y1;
geninterrupt(0x10);
;
set---color(int colorpp) /* 设置屏幕背景 */
{
--BH=0x00;
--BL=colorpp;
--AH=0x0b;
geninterrupt(0x10);
}
void hzprint(int x,int y, unsigned char *str, int atr)
{ /* 在指定位置显示彩色字符串 */
while(*str){
--DH=y-1;
--DL=x++;
--AH=0x02;
--BH=0x00;
geninterrupt(0x10); /* 定位 */
--AH=0x09;
--BL=atr;
--BH=0;
--AL=*str++;
--CH=0;
--CL=1;
geninterrupt(0x10);} /* 显示 */
}
void let---menu(int num, char *menu[],int count,int x,int y)
{ /* 菜单变量预赋值 */
int endx,endy;
hzmenu[num].len=strlen(menu[0]); /* 长度 */
if(num==0){
endy=y;endx=x+(hzmenu[num].len+2)*count;
hzmenu[num].H---V---flag=0; /* 主次菜单标志 */
}
else {endy=count+y;endx=hzmenu[num].len+x;
hzmenu[num].H---V---flag=1;}
hzmenu[num].startx=x,hzmenu[num].endx=endx; /* 起始和终止坐标 */
hzmenu[num].starty=y,hzmenu[num].endy=endy;
hzmenu[num].count=count; /* 菜单项数 */
hzmenu[num].menu=(char ** )menu; /* 菜单内容 */
}

```

```

}
dispaly--menu(int num)      /* 显示菜单 */
{
    register int i,x,y;
    x=hzmenu[num].startx;
    y=hzmenu[num].starty;
    if(hzmenu[num].H—V—flag){
        for(i=0;i<hzmenu[num].count;i++,y++){
            hzprint(x,y,hzmenu[num].menu[i],fcolor);
        }
    }
    else{
        for(i=0;i<hzmenu[num].count;i++,x=x+2+hzmenu[num].len)
            hzprint(x,y,hzmenu[num].menu[i],fcolor);
    }
}

selectmenu(int num)        /* 根据鼠标位置确定选择的菜单项 */
{
    register int i;
    showorhidemscurs();     /* 显示或者隐藏光标 */
    while(!anymkpressed());
    while(anymkpressed());  /* 没有压键则等待下一次选择,以保持显示状态 */
    getkeystatus();
    showorhidemscurs();
    if(hzmenu[num].H—V—flag) /* 从非主菜单中确定点中的菜单 */
    {
        /* 先决定列位置 */
        if(mouseX>=(hzmenu[num].startx-1)*8
            && mouseX<(hzmenu[num].endx)*8)
            for(i=0;i<hzmenu[num].count;i++){
                /* 决定行的位置 */
                if(mouseY>=(hzmenu[num].starty+i-1)*linech
                    && mouseY<(hzmenu[num].starty+i)*linech)
                    return i; /* 返回选中的菜单项数 */
            }
    }
    else /* 从主菜单中确定点中的菜单 */
    {
        if(mouseY>=(hzmenu[num].starty-1)*linech
            && mouseY<(hzmenu[num].starty)*linech)
            for(i=0;i<hzmenu[num].count;i++){
                if(mouseX>=(hzmenu[num].startx+(hzmenu[num].len+2)*i)*8
                    && mouseX<(hzmenu[num].startx+hzmenu[num].len+(hzmenu[num].len+2)*i)*8)
                    return i;
            }
    }
    return -1;
}

```

```

void setmkstatus(unsigned int mst)
{
    switch(mst){
        case 0: mousekey=None; break;    /* 没有按钮 */
        case 1: mousekey=Left; break;    /* 按了左按钮 */
        case 2: mousekey=Right; break;    /* 按了右按钮 */
        case 3: mousekey=Both; break;    /* 同时按了左右两按钮 */
    }
}

void resetmouse()
{
    union REGS mr;
    struct SREGS s;
    mr.x.ax=0;    /* 检查是否安装了鼠标或鼠标驱动程序 */
    int86x(0x33,&mr,&mr,&s);
    if(mr.x.ax>0)    /* mr.x.ax=FFFFH 表示硬件/驱动程序已安装 */
        /* mr.x.ax=0000H 表示硬件/驱动程序未安装 */
        /* 如在此处(函数内)设置断点,则可以得到类似下述调试表达式的值:
        mr.rx: {x: {ax: 0xFFFF, bx: 0x3, cx: 0x2, dx: 0x803, si: 0x73f2, di: 0x98BF,
            cflag: 0x0, flags: 0x7212}, h: {al: 0xFF, ah: 0xFF, bl: 0x3, bh: 0x0,
            cl: 0x2, ch: 0x0, dl: 0x3, dh: 0x8}}
        */
        mskeysnum=mr.x.bx;    /* 鼠标按钮数 */
        /* mr.x.bx=FFFFH 两个按钮 */
        /* mr.x.bx=0000H 不是两个按钮 */
        /* mr.x.bx=0003H 是 Mouse Sysytems/Logitech 鼠标 */
        /* 缺省的鼠标器参数 值(仅供参考)
        光标位置 屏幕中心
        内部光标位置 -1
        图形光标 箭头
        文本光标 反相
        中断屏蔽码 0
        光笔模拟 允许
        水平 Mickey 象素比 8:8
        垂直 Mickey 象素比 16:8
        高速移动下限 每秒 64 Mickey
        最小水平位置 0
        最大水平位置 虚拟屏幕 X 值 -1
        最小垂直位置 0
        最大垂直位置 虚拟屏幕 Y 值 -1
        CRT 页号 0 */
        /* Mickey 数【米基数,物理点】是鼠标器的移动单位,等于 1/8 mm,大约为 */
        /* 1/2000英寸,其计数范围为: -32768~32767. 正的水平计数表示向右移动, */
        /* 负数为向左; 正的垂直计数表示向下移动,负数表示向上移动. 初始化时, */
        /* 改变两个物理点对应于改变一个鼠标屏幕点. 确定移动的物理点数可用中断 */
        /* INT 13H 的功能 0BH. 入口参数 AX=0BH. 返回: CX = 自上次调用中断 */
        /* INT 13H 功能以来水平移动 X 点数; DX = 垂直移动的 Y 点. */

```

```

else{
    mskeynum=0;
    hzprint(10,9,"系统没有安装鼠标或鼠标驱动程序!",fcolor+1);
    exit(0);
}

msvisible=False;

}

void showorhidemscurs() /* 显示或者隐藏光标 */
{
    /* 执行本函数后,原先显示的变成隐藏,隐藏的变成显示 */
    union REGS mr,outr; /* 在屏幕上显示字符前,一般要先熄灭光标,待字符 */
    struct SREGS s; /* 显示后再点亮 */
    if(msvisible){
        --AX=2; /* 使鼠标的光标不可见,或称熄灭光标 */
        msvisible=False; /* 初始化后光标一般在熄灭状态 */
    }
    else{
        --AX=1; /* 使鼠标的光标可见,或称点亮光标 */
        msvisible=True; /* 注意,为避免在屏幕上留下“鼠标器 */
        /* 斑点”,在任何改变视频显示的行动 */
        /* 之前应该总是使光标隐藏起来 */
    }
}

geninterrupt(0x33);
}

void getkeystatus() /* 获得鼠标按键状态和鼠标位置 */
{
    union REGS mr,outr;
    struct SREGS s;
    mr.x.ax=3; /* 返回位置和按钮状态 */
    int86x(0x33,&mr,&outr,&s);
    setmkstatus(outr.x.bx); /* outr.x.bx = 按钮状态 */
    /* 0000H = 左按钮 */
    /* 0001H = 右按钮 */
    /* 0002H = 中间按钮 */
    mouseX=outr.x.cx; /* 虚拟屏上光标的横座标(列) */
    mouseY=outr.x.dx; /* 虚拟屏上光标的纵座标(行) */
    querybtndnup(5,outr.x.bx);
}

void setmscursor(unsigned int x,unsigned int y) /* 设置虚拟屏上光标位置 */
{
    union REGS mr;
    struct SREGS s;
    mr.x.ax=4; /* 设置光标到新位置,或称定位鼠标光标 */
    mr.x.cx=x; /* 新列(X)坐标 */
    mr.x.dx=y; /* 新行(Y)坐标 */
    int86x(0x33,&mr,&mr,&s); /* 行列被截断为象格大小的整数倍(取低端) */
    mouseX=x; /* 记下坐标位置 */
}

```

```

mouseY=y;
}
void setgraphicscursor(mkcursortype mask)
{
    /* 定义图形光标的方法 */
    union REGS mr;
    struct SREGS mseg; /* 如显示方式为文本方式,则不能改变光标形状 */
    mr.x.ax=9; /* 改变光标形状 */
    mr.x.bx=mask.horz; /* 位映象中热点的列(X)坐标(-16~16) */
    mr.x.cx=mask.vert; /* 热点的行(Y)坐标(-16到16) */
    /* ES:DX= 位映象,指向屏幕掩码和光标掩码的指针 */
    /* 前16个字为屏幕掩码,后16个字为光标掩码 */
    /* 每一个字都定义了一行16个像素,低位为最右边 */
    mr.x.dx=FP-OFF(&mask); /* 偏移量 */
    mseg.es=FP-SEG(&mask); /* 段值 */
    int86x(0x33,&mr,&mr,&mseg);
}
#define software 0
#define hardware 1
void settextrcursor(int ctype,int s--line, int e--line)
{
    union REGS mr;
    struct SREGS mseg;
    mr.x.ax=0x0a; /* 定义文本光标的方法 */
    mr.x.bx=ctype?hardware:software; /* software=0, hardware=1 */
    if(mr.x.bx==1)
    {
        /* 选择硬件光标,即通过硬件寄存器编程实现的光标 */
        mr.x.cx=s--line; /* 设置硬件光标开始线 s--line=6 */
        mr.x.dx=e--line; /* 设置硬件光标结束线 e--line=7 */
    }
    else
    {
        /* 选择属性光标,即通过软件实现的光标 */
        mr.x.cx=FP-OFF(&maktext[0]); /* 屏幕掩码 */
        mr.x.dx=FP-OFF(&maktext[1]); /* 光标掩码 */
    }
    int86x(0x33,&mr,&mr,&mseg);
} /* 在文本方式下,鼠标光标将修改硬件光标的属性并移动硬件光标。属性 */
/* 光标由屏幕掩码和光标掩码定义。屏幕掩码保持字符的原来属性,光标 */
/* 掩码选择改变属性。这两者都是一个字长的值,屏幕掩码的低位字节应 */
/* fFH; 光标掩码的低位字节应为 0H。逻辑上,软件光标位置处的属性和 */
/* 字符字节应先与屏幕掩码进行与(AND)操作,然后与光标掩码进行异 */
/* 或(XOR)操作。硬件光标设置结果与使用中断 INT 10H 设置结果相同。 */
unsigned int anymkpressed()
{
    getkeystatus();
    mkp=mousekey!=none;
    return(mkp);
}

```



```

}
unsigned int leftmkpressed()
{
    getkeystatus();
    mkp=mousekey==Left;
    return(mkp);
}
unsigned int rightmkpressed()
{
    getkeystatus();
    mkp=mousekey==Right;
    return(mkp);
}
unsigned int bothmkpressed()
{
    getkeystatus();
    mkp=mousekey==both;
    return(mkp);
}
void setmaxminlinepos(int max,int min)
{
    —AX=7; /* 设置光标在水平方向上的活动范围 */
    —CX=min; /* 最小列号 */
    —DX=max; /* 最大列号 */
    geninterrupt(0x33);
}
void setmaxmincolpos(int max,int min)
{
    —AX=8; /* 设置光标在垂直方向上的活动范围 */
    —CX=min; /* 最小行号 */
    —DX=max; /* 最大行号 */
    geninterrupt(0x33);
}
void querybtndnup(int dn—or—up,int button)
{
    union REGS mrl,outr1;
    struct SREGS sl;
    mrl.x.ax=dn—or—up; /* 5 或6 */ /* 功能 5 为返回按钮按下数据 */
    /* 功能 6 为返回按钮释放数据 */
    mrl.x.bx=button; /* 设置按钮状态 */
    /* 0000H = 左按钮 */
    /* 0001H = 右按钮 */
    /* 0002H = 中间按钮 */
    int86x(0x33,&mrl,&outr1,&sl);
    mbtnstatus=outr1.x.ax; /* 返回:位0 = 1 表示按下左按钮 */
    /* 位1 = 1 表示按下右按钮 */
}

```

```

/*      位2 = 1表示按下中间按钮 */
mopcount=outr1.x.bx; /* 从最后一次调用起指定按钮按下次数 */
mxpos=outr1.x.cx; /* 指定按钮最后按下的列 */
mypos=outr1.x.dx; /* 指定按钮最后按下的行 */
}

void hz(int mbtnstatus1,int mopcount1,int mxpos1,int mypos1,int attr)
{
static unsigned char btnstatus[2],opcount[2],xpos[2],ypos[2];
itoa(mbtnstatus1,btnstatus,10);
hzprint(14,11,btnstatus,attr);
itoa(mopcount1,opcount,10);
hzprint(14,12,opcount,attr);
itoa(mxpos1,xpos,10);
hzprint(14,13,xpos,attr);
itoa(mypos1,ypos,10);
hzprint(14,14,ypos,attr);
}

```

判断鼠标器按钮是否按下及鼠标器是否移动,也可以通过中断 INT 13H 的功能 0CH 实现。利用该功能可以设置一个被鼠标驱动程序认识的特殊处理程序。处理功能可对应于按下按钮、释放按钮、光标位置变化或这些事件的组合。该功能使用方法为:

```

void interrupt mousehandler() /* 自定义事件处理子程序(中断函数) */
{ ..... }

void task(int mask)
{
union REGS mr; /* 规定对每个鼠标器事件要执行的动作 */
struct SREGS mseg;
mr.x.ax=0xC; /* 功能号 */
mr.x.cx=mask; /* 屏蔽码(或称【调用掩码】) */
mr.x.dx=FP—OFF(mousehandler); /* ES:DX=鼠标器服务程序的地址 */
mseg.es=FP—SEG(mousehandler); /* 中断处理子程序执行完后继续 */
int86x(0x33,&mr,&mr,&mseg); /* 执行现行程序 */
} /* 无返回 */

```

屏蔽码 mask 是一个整数,它用于确定在那些条件下可以引起鼠标器硬件中断。它的每一位值为 1 时对应一个可以引起硬件中断的条件(仅供参考):

位15~7	保留,未用,通常为 0
位6	若中间按钮放开则调用
位5	若中间按钮按下则调用
位4	若右边按钮放开则调用
位3	若右边按钮按下则调用
位2	若左边按钮放开则调用
位1	若左边按钮按下则调用
位0	若鼠标器移动则调用

例如,屏蔽码可为 0x2B。确定当前鼠标驱动程序版本和存在的鼠标器类型可用中断 INT 33H 的功能 24H。入口参数是 AX=24H。返回

AX=FFFFH 出错
 AX=非FFFFH BH = 主版本号
 BL = 副版本号
 CH = 类型 (1= 总线, 2= 串行, 3= Inport, 4= PS/2, 5= HP)
 CL = 中断 (0=PS/2, 2=IRQ2, 3=IRQ3, ..., 7=IRQ7)

33.2.3 鼠标的图形光标设计

图形光标由屏幕掩码和光标掩码组成, (参见图 33-1 和图 33-2)。

高位	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	低位
FD							X										ff
F8						X	X	X									ff
F0					X	X	X	X	X								7f
E0				X	X	X	X	X	X	X							3f
C0			X	X	X	X	X	X	X	X	X						1f
80		X	X	X	X	X	X	X	X	X	X	X					0f
00	X	X	X	X	X	X	X	X	X	X	X	X	X				07
00	X	X	X	X	X	X	X	X	X	X	X	X	X	X			03
F0					X	X	X	X	X	X							3f
F8						X	X	X	X	X	X						1f
FC							X	X	X	X	X	X					0f
FE								X	X	X	X	X	X				07
FF									X	X	X	X	X	X			03
FF										X	X	X	X	X	X		81
FF											X	X	X	X	X	X	c0
FF											X	X	X	X	X	X	c0

图 33-1 屏幕掩码 (简称掩码)

高位	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	低位
FD							X										ff
FA						X		X									ff
F7					X				X								7f
EF				X						X							bf
DF			X								X						df
BF		X										X					ef
7F	X												X				f7
0F	X	X	X	X						X	X	X	X	X			83
F7					X					X							bf
FB						X					X						df
FD							X					X					ef
FE								X					X			7b	f7 FF
FF										X					X		bd
FF											X					X	de
FF											X	X	X	X	X	X	cf

图 33-2 光标掩码 (亦称图形码)

如采用这种掩码（打×的小框为0，不打的为1。光标掩码为屏幕掩码的边框）：

```
mkcurstype handcurs={
    /* 前16个为屏幕掩码，后16个为光标掩码，最后为热点 */
    0xfdff,0xf8ff,0xf07f,0xe03f,0xc01f,0x800f,0x0007,0x0003,.....}
```

这样的鼠标在不同颜色的背景中均能看见。

33.2.4 用鼠标画图

通过程序 MOUSE2.C 使你知道如何使用鼠标画图的一般性原理，以及怎样在自己的程序中使用鼠标作为输入设备。

```
C>TYPE MOUSE2.C
#include "conio.h"
#include "ctype.h"
#include "dos.h"
#include "graphics.h"
#define HOMEKEY 71
#define LEFTKEY 75
#define RIGHTKEY 77
#define PGUPKEY 72
#define PGDNKEY 80
#define txt(liner,str) gotoxy(1,liner);cprintf(str);
int msinit(int Xlo,int Xhi,int Ylo,int Yhi);
int msread(int *pX,int *pY,int *pbuttons);
int Xpixel(int x),Ypixel(int y),xhpg(int X),yhpg(int Y);
void cursor(int X,int Y);
void getline(int *pxstart,int *pystart);
void drline(int x1,int y1,int x2,int y2);
void endprogram(void);
void newposition(int *px,int *py,int *pX,int *pY,int *pbut);
void mode(int mode—code);
void hzprint(int x,int y,unsigned char *str,int atr);
union REGS regs;
int y200,xmin=0,xmax,ymin=0,ymax,Smin,Smax,Tmin,Tmax;
int mouse—flag=0,u—max=640,v—max=480;      /* 老坐标系下坐标极值 */
long x—max200,y—max200;
main()                                          /* 利用鼠标任意画图演示程序 */
{
    int buttons,xm,ym,X,Y,x,y;
    int x—max=1000,y—max=7; /* 为建立一种新坐标系而用，Y轴将向上，坐标原点在屏幕左下角。这两个数跟具体显示器类型有关。如果选择得不合适则光标达不到屏幕上某些位置。*/

    char str[100];
    int driver,mode;
    setgraphmode(3);                          /* 置文本方式 */
    clrscr();                                  /* 清屏幕 */
```

```

txt(1,"strike any key ,start..."),getch(); /* 击任一键开始 */
driver=VGA;
mode=VGAHI;
initgraph(&driver,&mode,""); /* 初始化图形方式 */
/* mode(18); 注意,在用 initgraph() 后不允许再用此语句 */
setfillstyle(SOLID-FILL,BLUE); /* 置兰色背景 */
bar(0,0,640,480); /* 全屏幕 */
hzprint(15,15,"用任一按钮移动鼠标",2); /* 绿色显示串 */
/* outtextxy(15,15,"用任一按钮移动鼠标"); 用这种形式在某些汉字操 outtextxy(15,35,"
键Q 退出"); 作系统下不能显示汉字 */
hzprint(15,16,"键 Q 退出",4); /* 红色显示串 */
x—max200=200 * x—max;
y—max200=y200=200 * y—max;
Smin=4;
Smax=x—max—4;
Tmin=2;
Tmax=y—max—12;
xmax=xhpg(Smax);
ymax=yhpg(Tmax);
setwritemode(XOR-PUT); /* 设置写象点方式,异或操作 */
if( msinit(0, (int)x—max200,0, (int)y—max200)==0)
{
    hzprint(15,18,"鼠标或鼠标驱动程序未装",15); /* 白色显示串 */
    delay(5000); /* 延迟,以便观察 */
    /* cprintf("鼠标或鼠标驱动程序未装"); 此句在某些汉字操作系统下不能显示 */
    endprogram();
}
while(msread (&xm,&ym,&buttons) >= 0);
x=xm;
y=ym;
cursor(x,y);
for(;;) /* 不断循环,只有当按 Q 键或 Ctrl-C 时才退出并结束程序 */
{
    newposition(&x,&y,&X,&Y,&buttons);
    if(buttons)getline(&x,&y); /* 为简明起见,鼠标按钮压住时程序只画一点 */
}
}
void cursor(int x,int y)
{
    int Xm4,Ym2,Ym1,Xp4,Yp2,Yp1,X,Y;
    X=Xpixel(x);
    Y=Ypixel(y);
    Xm4=X-4;

```

```

Xp4=X+4;          /* 用于标示画图位置的光标形状 */
Ym2=Y-2;          /* (Xm4,Ym2) (Xp4,Ym2) */
Ym1=Y-1;          /* ●—————● */
Yp1=Y+1;          /* ●—————● */
Yp2=Y+2;          /* (Xm4,Ym1) (Xp4,Ym1) */
line(Xm4,Ym2,Xp4,Ym2); /* (Xm4,Yp1) (Xp4,Yp1) */
line(Xm4,Yp2,Xp4,Yp2); /* ●—————● */
line(Xm4,Ym1,Xm4,Yp1); /* ●—————● */
line(Xp4,Ym1,Xp4,Yp1); /* (Xm4,Yp2) (Xp4,Yp2) */
}

```

```

void getline(int *pxstart,int *pystart) /* 画一线,本程序简化为画一点 */

```

```

{
int x,y,xstart,ystart,buttons,x0,y0,X,Y;
x=xstart=*pxstart;
X=Xpixel(x);
y=ystart=*pystart;
Y=Ypixel(y);
do
{
x0=x;
y0=y;
newposition(&x,&y,&X,&Y,&buttons);
if(x!=x0 || y!=y0)
{
/* 注意,鼠标不要移动太快 */
drlne(xstart,ystart,x0,y0); /* 先抹去旧的线段 */
drlne(xstart,ystart,x,y); /* 再画出新的线段 */
}
}while(buttons); /* 当鼠标按钮一直压着时进行循环 */
*pxstart=x; /* 利用指针传值 */
*pystart=y;
}

```

```

void drlne(int x1,int y1,int x2,int y2)

```

```

{
int aux,X1,Y1,X2,Y2;
X1=Xpixel(x1);
Y1=Ypixel(y1);
X2=Xpixel(x2);
Y2=Ypixel(y2);
if(X1==X2 && Y1==Y2)return; /* 如果两点重合,可以不画象点 */
switch(mouse—flag)
{
/* 画一象点 */
case 1: putpixel(X2,Y2,YELLOW);break; /* 按左按钮画黄色 */
case 2: putpixel(X2,Y2,GREEN);break; /* 按右按钮画绿色 */
case 4: putpixel(X2,Y2,RED);break; /* 按中间按钮画红色 */
default:putpixel(X2,Y2,0); /* 未按按钮 */
}
}

```

```

}
void endprogram(void) /* 结束程序 */
{
    restorecrtmode(); /* 恢复图形初始化前模式 */
    exit(0);
}
int msinit(int Xlo,int Xhi,int Ylo,int Yhi) /* 鼠标初始化 */
{
    int retcode;
    regs.x.ax=0; /* 测试鼠标及类型 */
    int86(0x33,&regs,&regs);
    retcode=regs.x.ax;
    if(retcode==0)return 0; /* 未装鼠标或鼠标驱动程序 */
    regs.x.ax=7; /* 定义水平光标范围 */
    regs.x.cx=Xlo;
    regs.x.dx=Xhi;
    int86(0x33,&regs,&regs); /* 定义垂直光标范围 */
    regs.x.ax=8;
    regs.x.cx=Ylo;
    regs.x.dx=Yhi;
    int86(0x33,&regs,&regs);
    return retcode;
}
int msread(int *px,int *py,int *pbuttons)
{
    /* 从本函数可看到鼠标和键盘能同时发挥作用 */
    static int x0=-1000,y0=-1,buto=-1; /* 注意静态变量的特殊性 */
    int xnew,ynew;
    do {
        if(kbhit())return getch(); /* 如果在等待时发现有一个键被按下,退出循环,返回键值 */
        /* 返回按钮状态 */
        regs.x.ax=3;
        int86(0x33,&regs,&regs);
        xnew=regs.x.cx; /* 列 */
        ynew=regs.x.dx; /* 行 */
        *pbuttons=regs.x.bx; /* 按钮状态 */
        /* 只有当鼠标位置发生变化时才退出循环,返回鼠标状态 */
    }while( xnew == x0 && ynew == y0 && *pbuttons == buto);
    /* 记录鼠标状态 */
    *px=xnew;
    *py=y200-ynew;
    mouse--flag=*pbuttons;
    return -1; /* 如鼠标起作用返回 -1 */
}
void newposition(int *px,int *py,int *pX,int *pY,int *pbut)
{
    int ch,x0=*px,y0=*py,x,y;
    static int xm,ym,xcorr=0,ycorr=0;

```

```

ch=tolower( msread(&xm,&ym,pbut) );    /* 将返回值转为小写字符 */
if(ch>=0)                                /* 按鼠标时返回值为 -1 */
{
    if(ch==0)                            /* 按了控制键 */
    {
        ch=getch();                      /* 取扩充码 */
        switch(ch)
        {
            /* 利用击键盘键只移动光标,不画象点 */
            case PGUPKEY ,ycorr++,break;    /* 按了 ↑ 键 */
            case LEFTKEY ,xcorr--,break;    /* 按了 ← 键 */
            case RIGHTKEY ,xcorr++,break;    /* 按了 → 键 */
            case PGDNKEY ,ycorr--,break;    /* 按了 ↓ 键 */
            case HOMEKEY :xcorr=ycorr=0;break; /* 按了 Home 键 */
        }
    }
}
if(ch=='q')endprogram();                /* 如按了 Q 键结束程序运行 */
x=xm+xcorr;
y=ym+ycorr;
if(x<xmin)x=xmin;                       /* 极值处理,使不超界 */
if(x>xmax)x=xmax;
if(y<ymin)y=ymin;
if(y>ymax)y=ymax;
if(x!=x0 || y!=y0)
{
    cursor(x0,y0); /* 先将老位置上的光标清除.注意,此句是必不可少 */
    cursor(x,y);   /* 然后在新位置画出一光标,这相当于看到光标移动 */
}
*px=x;
*py=y;
*pX=Xpixel(x);
*pY=Ypixel(y);
}
int xhpg(int X) /* 计算老坐标系下的象点在新坐标系下坐标 */
{
    return (int) (X * x--max200/u--max);
}
int yhpg(int Y)
{
    return (int) ((v--max-Y) * y--max200/v--max);
}
int Xpixel(int x) /* 计算新坐标系下的象点在老坐标系下坐标 */
{
    return (int) ((long)u--max * x/x--max200);
}
int Ypixel(int y)

```



```

{
return v--max-(int)((long)v--max*y/y--max200);
}
void mode(int mode--code) /* 利用中断 INT 10H 设置显示模式。本程序未用 */
{
--AH=0;
--AL=mode--code;
geninterrupt(0x10);
}
void hzprint(int x,int y,unsigned char *str,int atr)
{
/* 利用 BIOS 一级兼容性,调用中断 INT 10H 显示字符串,这样 */
while(*str) /* 可解决汉字操作系统改 INT 10H 引起的不能显示汉字的问题 */
{
--DH=y-1;
--DL=x++;
--AH=0x02;
--BH=0x00;
geninterrupt(0x10);
--AH=0x09;
--BL=atr;
--BH=0;
--AL=*str++;
--CH=0;
--CL=1;
geninterrupt(0x10);
}
}

```

第三十四章 打印机

34.1 概述

(一) 打印机的种类

1. 点阵(或矩阵)式打印机

它的打印头由若干钢针(目前大多为9针和24针)组成,由钢针击打色带,从而在纸上由点拼成字符。这种打印机既可打印文本,也可打印图象,是目前用得最多的打印机。

2. 喷墨式打印机

它的打印头上装有多個喷咀(目前有9~32个),打印时将墨水从喷咀喷到纸上。

3. 激光打印机

它由激光头产生极细的光束,在感光鼓上形成静电潜象,然后通过墨粉粘附而显影在纸上。

其它的打印机还有热敏打印机、液晶快门式打印机等等。下面主要涉及点阵打印机。

从通讯方式来分也可分为串行打印机和并行打印机。象某些激光打印机就是串行打印机。在使用串行打印之前要先定义软件接口,即向指定串行口送规定信号。例如,当HP2686A激光打印机与IBM PC/XT机硬件接口联接好后,在DOS下执行下列批命令文件就可定义软件接口(参见串行通讯部分)。

```
C>TYPE AUTO.BAT
MODE COM1:9600,N,8,1,P /* 波特率 9600, 不奇偶校验, 数字位为 8, 1 停止位 */
/* P 指异步通讯连接器用于串行接口打印机 */
MODE LPT1:=COM1 /* 这两条命令次序不能颠倒 */
```

在BIOS数据区中依次分配两个字节存放各串行口的地址,存放单元参见表34-1。

表 34-1

COM1	COM2	COM3
0040:0000~0040:0001 口地址 0x3f8	0040:0002~0040:0003 0x2f8	0040:0004~0040:0005 0x0(未装)

在集成环境下,直接利用调试表达式

```
* ((int far *)0x00400000),x
```

可立即查得指定地址单元中的口地址。

使用打印机前一般都需要装入相应的打印驱动程序。通常一些汉字操作系统都配有打印驱动程序,如金山系统的PRT24.COM、联想系统的LX-PRN.EXE和2.13系统的PR-

TA.COM 等等。注意：在某些汉字系统下对某些打印机也可以不要打印驱动程序。

(二)并行口地址

DOS 操作系统能处理三个并行设备 (LPT1、LPT2 和 LPT3)，并行打印机一般和并行口相接。每个并行设备在与 CPU 相联时，主机必须有一个相应的适配器。对并行设备的控制是对其对应的适配器操作实现的。对适配器的操作是通过对三个 I/O 寄存器的读写实现的。每个 I/O 寄存器都有自己的口地址：

表 34-2

适配器	数据输出寄存器	状态寄存器	控制寄存器
AT 机上 LPT1	0x378	0x379	0x37A
AT 机上 LPT2	0x278	0x279	0x27A

数据输出寄存器的口地址又称为适配器的基地址。在 BIOS 数据区中依次分配两个字节存放每一个适配器的基地址。当某个并行口上没有安装适配器时，系统将该口基地址初始化为 0。

表 34-3

LPT1	LPT2	LPT3
0040,0008~0040,0009	0040,000A~0040,000B	0040,000C~0040,000D
基地址 0x378	0x278	0x0(未装)

1. 数据寄存器

它的长度为一个字节，需要打印的数据必须送给它。

2. 状态寄存器

它的长度为一字节，各位意义如下：

位7 0=打印机忙	1=不忙
位6 0=收到字符的确信信号	1=正常 (CPU 发来的打印数据字节已被接收，此时控制寄存器的位 4 如为 1，则向 CPU 发肯定信号 (INT 7H) 通知 CPU 可以发送下一个打印数据字节)
位5 0=打印机有纸	1=无纸
位4 0=打印机未联上(称“脱机”)	1=打印机联机 (又称“选中”)
位3 0=打印机出错	1=正常
位2 未用	
位1 未用	
位0 未用	

在打印中间它经常向 CPU 报告打印机有否出错，以便 CPU 根据它的报告来正确地发送数据。

3. 控制寄存器

它的长度为一字节，各位意义如下：

位7	未用	
位6	未用	
位5	未用	
位4	0=禁止打印机中断	1=允许中断
位3	0=使打印机脱机	1=正常设置
位2	0=初始化打印口	1=正常设置
位1	0=正常设置	1=自动回车换行
位0	0=正常设置	1=字节数据输出

它的功能包括初始化适配器并控制数据的输出。它也可以将并行口设为中断方式,即当该口就绪时向 CPU 发出中断请求信号。这种方式可以使 CPU 在发送一个数据后能转去处理别的事情,而不是单纯的等待。

34.2 控制打印机函数

—1 测试机器上安装的打印机数

测试已装打印机数就是测试已装并行口数。

程序调用 BIOS 中断 11H 取设备表。注意:设备标志也可以通过访问内存地址单元 0040:0010 而得到。

```
C>TYPE PRINT1.C
main()
{
printf("当前机器中安装的并行口个数是:%d\n",p—install());
}
#include "dos.h"
p—install(void)          /* 安装的打印机数          */
{
    geninterrupt(0x11);    /* 调用 BIOS 中断 11 (取设备表) */
    return ((—AX & 0xc000) >>14); /* 检查位 14,15          */
    /* 上句也可改成:
        return ((peek(0x40,0x0010) & 0xc000) >>14); */
}
/* 程序输出:当前机器中安装的并行口个数是:1 */
```

—2 测试缺省的第一个并行口基地址

编程时要访问某个并行口首先要检查其占用的基地址。

```
C>TYPE PRINT2.C
main()
{
printf("缺省并行口 (Parallel Port)地址是:%#x\n",p—base());
}
#include <dos.h>          /* 此句可要可不要 */
p—base()                  /* LPT1 的基地址 */
{
    return (peek(0x0040,0x0008)); /* 存放在 0040:0008~0040:0009 中 */
}
```

/* 程序输出,缺省并行口(Parallel Port)地址是:0x378 */

—3 将送往缺省并行口即第一个并行口(LPT1)的数据在另一个并行口输出

BIOS 的屏幕打印子程序总是将输出送到 LPT1,利用本程序便可将它改向输出到其它并行口,或即在其它打印机上打出。注意:第一次调用函数假定 LPT1 和 LPT2 交换,第二次调用便又将它们交换回来。

DOS 功能调用 5H 是向标准打印机设备输出一个字符,而标准打印机设备通常指第一个并行口。在 DOS 2.0 以上版本下可被重定向。如果打印机“忙”,则该功能将等待。

C>TYPE PRINT3.C

```
main()
{
    int p=no;
    printf("输入端口数:2 或3\n");
    scanf("%d",&p-no);
    if(p-no < 2 || p-no > 3)exit(1);
    p=swap(p-no);
    printf("已将LPT1 改成LPT%d\n",p-no);
}

p=swap(int number)    /* 交换存放基地址单元中的基地址 */
{
    int far *p=lpt1=(int far *)0x00400008;
    int far *p=lpt;
    int temp;
    p=lpt=(int far *) (0x00400008+(number-1)*2);
    temp=*p-lpt1;
    *p-lpt1=*p-lpt;
    *p-lpt=temp;
}
```

—4 设置延迟打印出错信息显示时间

当打印机并未联机时送打印字符,便会显示出错信息

Not ready error device PRN

Abort Retry Ignore?

该信息不是一出错便马上显示出来,而是延迟一定时间后才显示,这就是延迟时间。延迟时间使用户在敲入打印命令后有适当的时间去准备打印机。缺省的延迟时间是 20 秒。

C>TYPE PRINT4.C

```
void p-timeout(int port,unsigned char seconds)
{
    /* 直接修改内存超时计数表的值 */
    if(port<1 || port>3)exit(1);    /* port 只允许为 1,2 和 3 */
    if(seconds<0 || seconds>255)exit(2); /* seconds 的取值范围 0~255 */
    *((char far *) (0x00400077+port)) = ++seconds;
    /* 注意:seconds 的值要先加上 1,这是因为 BIOS 对它采用的是将它
    减 1 的值。因此,如 seconds = 255, (unsigned char)(255+1)=0,
    随后 BIOS 对它进行减 1 操作,而 (unsigned char)(0-1) = 255,
    因而复原或者说只有这样才能和程序员设想的一致 */
}
```

```

}
#include "stdio.h"
main()
{
int x,t;

/* 打印 BIOS 数据区保留的超时计数器表 */
printf("LPT1 缺省的延时时间是:%d 秒\n",peekb(0x40,0x0078));
printf("LPT2 缺省的延时时间是:%d 秒\n",peekb(0x40,0x0079));
printf("LPT3 缺省的延时时间是:%d 秒\n",peekb(0x40,0x007a));
/* 在编程时你也可用估算表达式
* ((char far *)0x00400078),d
直接得出这类值,而无须调试得到 */
printf("选输LPT1、LPT2 或LPT3\n");
scanf("%* 3s%d",&x); /* 严格输入,只接收一位整数 */
printf("延迟秒数(0~255)\n");
scanf("%u",&t);
p=timeout(x,(unsigned char)t);
}

```

—5 查打印机状态

一般利用 BIOS 的中断 INT 17H 功能 2 来获得状态字节处理后的值。可以直接访问 打印机状态寄存器,但一般不要直接用之。容易从下面程序中看出, BIOS 中断 17H 的功能 2 将对它进行处理,例如屏蔽了三个无用位等等。常用的是测试打印机是否联机等。

C>TYPE PRINT5.C

```

#include "dos.h"
main()
{
int port;
int p=no=0; /* 打印机号。LPT1 为 0,LPT2 为 1,LPT3 为 2 */
unsigned char bit=4; /* 检测打印机状态字节的位 4,bit=0~7 */
printf("%* #x\n",p=status(p,no,bit));
port=inportb(0x379); /* 直接读状态寄存器中的值 */
printf("port=%* #x,%* #x\n",port,(port & (0x1<<bit)));
} /* 程序输出,状态字节=0x10, 0x10
port=0x4f,0 */
int p=status(int p=no, unsigned char bit)
{
unsigned char a;
—AH=2;
—DX=p=no;
geninterrupt(0x17); /* BIOS 中断 17H 的功能 2,取打印机状态 */
a=—AH;
printf("状态字节=%* #x",a);
return (a & (0x1<<bit));
}

```

—6 初始化打印口

初始化当前与 LPT1、LPT2 或 LPT3 相连的打印机端口,其目的是清打印机缓冲区,使所有通过软件送给打印机的控制码无效,恢复打印口的最初参数。一些打印机上有复位按钮,就是起这个作用。例如:对 AR3240 打印机,如在按住“联机”按钮后又按“装饰体”按钮,则起到“清缓存/复位”的作用(前三秒钟清打印机缓冲区,随后打印机响三下表示打印机复位)。

注意:常有这样的情况,当程序执行已经结束,但程序送往打印机缓冲区中的数据可能还未打印完,此时你不能让新执行的程序一开始初始化打印口,那样会把缓冲区中的内容冲掉,从而不能被打印出来。

打印机初始化和初始化打印口是不同的。前者是在打印机上电时自动进行的,它根据打印机上设定的 DIP(双列直插式组件)开关自动设置初始参数。这些参数一般是和打印机本身相关的性能,如打印方式、右边距、页首边限、控制码兼容、缓冲量、字间距、行距等等,这显然和初始化打印口涉及的参数是不同的。

要想改变打印机初始参数值,应在上电前重新设置打印机上的开关,因打印机只在开机时才读这些开关值。

```
C>TYPE PRINT6.C
#include "dos.h"
void p—init();
main()
{
    int port;
    int p—no=0;          /* 打印机号. LPT1 为 0,LPT2 为 1,LPT3 为 2 */
    p—init(p—no);
}
void p—init(int p—no)
{
    unsigned char a;
    —AH=1;
    —DX=p—no;
    geninterrupt(0x17); /* BIOS 中断 17H 的功能 1,初始化打印机 */
    a=—AH;
    printf("状态字节初始值=%#x\n",a);
}
/* 在没有连接打印机时,可能输出:状态字节初始值=0x18 或 0x10 */
—? int —Cdecl biosprint(int cmd,int abyte,int port);
它调用 BIOS 的中断 INT 17H 对打印机进行操作。
```

表 34-4 INT 17H 子功能

参数或寄存器	向打印机输出一个字符	初始化打印机端口	返回打印机状态
子功能号cmd 或AH	00H	01H	02H
字节abyte 或AL	输出一个字符(0~255)	参数值被忽略	
端口port 或DX	0~2(0 对应LPT1,1 对应LPT2,2 对应LPT3)		

函数返回值的 8 位（在 AH 寄存器中）是打印机端口状态（它们可能以组合形式出现），当位值为 1 时分别表示：

- 位 0 超时错误（有些机器未用，有的用位 1 作判别）
- 位 1~2 未用
- 位 3 I/O 错
- 位 4 联机（选择）
- 位 5 打印纸用完
- 位 6 确认打印机接收到字符的信号为正常
- 位 7 打印机不忙

注意：初始化打印机端口和打印机初始化是两个不同的概念。前者是 CPU 对打印机适配器上的三个 I/O 寄存器（数据输出寄存器、状态寄存器和控制寄存器）的操作，后者则是打印机内部的事。返回的端口状态实际是状态寄存器中的值。

本函数只适用于 IBM PC 及其兼容机。

注意：有些 CCDOS 增加了一些中断功能。如方正汉卡增加

入口：AH=80H（安装打印驱动程序，如为 81H 则卸去打印驱动程序）

返回：AX=FF16H 安装了 16 点阵驱动程序

=FF24H 安装了 24 点阵驱动程序

C>TYPE PRINT7.C

```
#include "bios.h"
main()
{
    int k;
    char d[]="WAIT", *ptr=d;
    for(k=0;k<8;k++)
        printf("k=%d %#01x\t",k,biosprint(2,1000,0)>>k & 0x1);
    printf("\n");
    while(*ptr) biosprint(0,*ptr++,0); /* 向打印机输出字符串 WAIT */
    /* 注意，上句不能写成：while(*d) biosprint(0,*d++,0); */
    /* 由此可见数组和指针在使用上的区别 */
}
/* 程序输出（在并行口 LPT1 存在，但又未装打印机时）：
k=0 0 k=1 0 k=2 0 k=3 0 k=4 0x1 k=5 0x1 k=6 0 k=7 0 */
```

—8 检查当前使用的 MS-DOS 版本，并检查内存中有否装入 PRINT.COM 或 PRINT.EXE 程序，该程序运行时将常驻内存。

C>TYPE PRINT8.C

```
#include "dos.h"
main()
{
    printf("当前使用的版本是：%#06x，即%d.%d 版\n",_version,_osmajor,_osminor);
    if(_osmajor<3) /* _osmajor 是 MS-DOS 主版本号 */
        (printf("非DOS 3.X 及以上版本\n"),return(1));
    _AH=1;
    _AL=0;
}
```



```

geninterrupt(0x2F);
switch(—AL)
{
case 0:
    printf("未安装print.com 或print.exe 程序,但可以安装\n");
    return (2);
case 1:
    printf("未安装print.com 或print.exe 程序,也不能安装它\n");
    return (3);
case 0xff:
    printf("已安装print.com 或print.exe 程序\n");
    return (4);
default:
    printf("不能判定\n");
    return (—AL);
}
}

/* 例如:当没有装入 print.exe 时程序输出:
    当前使用的版本是:0x0005,即5.0 版
    未安装print.com 或print.exe 程序,但可以安装 */

```

34.3 DOS 5.0 的脱机打印程序 PRINT.EXE

一 语法

```

print [/d:device] [/b,size] [/u,ticks1] [/m,ticks2] [/s,ticks3]
    [/q,qsize] [/t] [[drive:][path]filename[...]] [/c] [/p]

```

二 说明

1. device

指定的打印设备名。对并行口为 LPT1、LPT2 和 LPT3；对串行口是 COM1、COM2、COM3 和 COM4。缺省值是 prn。prn 和 LPT1 都指向同一并行口。开关“/d:”是必写在打印设备名前的。下述带反斜杠的开关都是这样。

2. size

设置内部缓冲区的字节数。数据在送往打印机前可暂存在该缓冲区中。size 最小并且也是缺省值为 512 字节,最大为 16384 字节。增加此值可提高打印速度,但将占用可用有效内存。

3. ticks1

print 等待打印机的最大时钟滴答数(时钟滴答数为每秒出现约 18 次),如果在等待时间内打印机未准备好,则作业(如指定文件等)将不被打印。该值范围为 1~255,缺省值为 1。

4. ticks2

规定 print 在打印机上打印一个字符所需的最大时钟滴答数。如果一个字符打印太慢,

MS-DOS 将显示出错信息。该值范围是 1 ~ 255,缺省值是 2。

5. ticks3

指定用 print 后台打印时 (background printing) 最大时钟滴答数。提高此值可加快打印速度,但同时将放慢其它程序的执行速度。该值范围是 1 ~ 255,缺省值是 8。

打印子程序可以使用打印中断。当打印机准备好时就中断 CPU,中断程序迅速发送字符给打印机,然后 CPU 马上转去做别的事 (因为输出一个字符对 CPU 只需很短的时间,而打印机打印一个字符则要较长时间)。这种方法称为【脱机打印】或【后台打印】。

6. qsize

指定打印队列 (print queue) 中允许的最大文件数,范围是 4 ~ 32,缺省值是 10。

7. /t

从打印队列中消除所有文件。

8. drive:

指定文件的驱动器名。

9. path

指定文件的路径名 (目录和子目录名)。

10. filename

指定文件的基本名和扩展名。

11. ...

可以在一个 DOS 命令行上指定多个文件名。

12. /c

它跟在文件名后,表示将该文件从打印队列中消除。

13. /p

它跟在文件名后,表示将该文件增加到打印队列中。

三 注意

1. 每个打印队列最多允许包含 64 个字符。

2. 开机后,开关 /d、/b、/u、/m、/s 和 /q 只能用一次,要使它再次使用有效必须重新启动操作系统 (冷、热启动或复位)。

3. 每个任选项间用空格符隔开。

```
C>TYPE PRINT9.C
```

```
#include "string.h"
#include "dos.h"
#define NOerror 0          /* 没有错误 */
main(int argc,char *argv[]) /* 使用本程序前 print.exe 必须装入内存 */
{                          /* 适用于 DOS 3.X 及以上版本 */
    int errorcode;
    char k;
    static struct{
        char level;        /* 级别 (必须为 00h) */
        char far *file;    /* 递交文件名,不能使用 DOS 通配符 */
    }submitpack={'\0'}; /* DS:DX 要递交包的格式,这是规定格式 */
    char *filename="qq.c";
    static char buffer[65];
```

```

if(argc != 2)exit(1);
if (strcmp(argv[1],"/c")==0)      /* 从队列中消除一个文件 */
{
    /* 文件名中允许用通配符 * 或 ? */
    k=2;
    strcpy(buffer,filename);      /* 为什么这样做？一是如果文件名中包含星号 DOS 则把它扩充为一串
                                   ?,直接写进缓冲区。因此需要更大的缓冲区以避免存储冲突。二是寄
                                   存器 DS 必须包含 filename 的段地址。但在大的存储器中,每个文件的
                                   数据是分段存放的,并且 DS 包含了当前文件数据的段地址。因此把
                                   filename 拷贝到局部缓冲区而不是改变 DS 的内容。 */
    --DX=(unsigned int )buffer;
}
else if (strcmp(argv[1],"/p")==0) /* 将文件加入要打印的文件队列 */
{
    k=1;
    submitpack.file=filename;
    --DX=(unsigned int)&submitpack; /* 将递交包的段,偏移地址放在 DS:DX 中 */
}
else if (strcmp(argv[1],"/t")==0) /* 停止打印当前文件,并消除全部文件 */
{
    k=3;
}
else exit(1);
--AH=1;
--AL=k;
geninterrupt(0x2f);
errorcode=--AX; /* 如成功,可能返回一个很大的随机数 */
if(errorcode<16)return(errorcode); /* 但对将文件加入到队列,AL=01H */
else return (NOerror);
}

```

34.4 设置打印参数

打印参数实际是一些控制打印机动作的控制码,它由 ASCII 字符集前 32 个代码开头,后跟一些连续的 ASCII 码组成。对一种打印机,每一个控制码都严格规定其组成代码顺序和个数。打印机收到控制码后会自动分析,知道它不是数据而是命令。前 32 个代码常用于通信及打印机或其它设备的操作中,它们的助记符和用处如表 34-5 所示(注意:有些代码功能几乎是不可变的,但有些随设备而变)。

表 34-5

16 进制码	助记符	用 处
\0x0	NUL	间隔符(字符本身无意义,常用于延迟)
\0x1	SOH(Start of Heading)	启动标题。开始传输数据块或新文件
\0x2	STX(Start of Text)	启动文本。表明标题后的文本开始
\0x3	ETX(End of Text)	文本结束。可标识检查数据的错误开始

\0x4	EOT(End of Transmission)	传输结束。信号断开,有时也可标识文件结束
\0x5	ENQ(Enquiry)	查询。询问远程站传来的状态信息
\0x6	ASK(Acknowledge)	应答。确认站之间通信成功
\0x7	BEL(Bell)	告警。喇叭发声,信号设备需要维护
\0x8	BS(Backspace)	退格
\0x9	HT(Horizontal Tab)	实行横向(或称水平)制表
\0xA	LF(Line Feed)	跳行(进一行)
\0xB	VT(Vortical Tab)	实行纵向(或称垂直)制表
\0xC	FF(Form Feed)	跳页(进一页)
\0xD	CR(Carriage Return)	回车
\0xE	SO(Shift Out)	移出。用于改变字符集
\0xF	SI(Shift In)	移入。用于改变字符集
\0x10	DLE(Data Link Escape)	数据链换码,指明后续字符的意思(有点象ESC)
\0x11	DC1(Device Control 1)	设备控制符1,用作远程通信的XON信号 对某些打印机用它作联机信号
\0x12	DC2(Device Control 2)	设备控制符2,一般用于触发信号。
\0x13	DC3(Device Control 3)	设备控制符3,用作远程通信的XOFF信号。对某些打印机作为脱机信号
\0x14	DC4(Device Control 4)	设备控制符4,一般用于触发信号
\0x15	NAK(Negative Acknowledge)	拒绝应答信号,信号传输失败
\0x16	SYN(Synchronous Idle)	同步空位,用于数据块间的同步传输
\0x17	TB(End of Transmission Block)	传输块结束标志,ETX的另一标记
\0x18	CAN(Cancel)	消除,常用于信号传输错误的消去。对某些打印机用于清除行缓冲区
\0x19	EM(End of Medium)	介质末端,数据源信号实际结束
\0x1A	SUB(Substitute)	替代。转换那些无效的或不可显示字符
\0x1B	ESC(Escape)	换码。标明其后跟的是控制码序列
\0x1C	FS(File Separator)	文件分隔符,标识文件间逻辑界限。在打印机上,也常把它当作ESC那样,后跟控制码序列
\0x1D	GS(Group Separator)	数据群分隔符
\0x1E	RS(Record Separator)	数据记录分隔符
\0x1F	US(Unit Separator)	数据单元分隔符
\0x7F	DEL(Delete)	删除字符

打印参数有些可通过打印机面板上的按钮设置,但更多的是通过软件设置。在大多数情况下,打印参数一经设定便一直起作用,直到用下一个命令将它修改后为止。当然,在打印机出错或断电后又重新通电后原设的打印参数便要重新设置了。

缺省的打印参数是由打印驱动程序预先设置处理的。所谓打印驱动程序是将打印子程序

所产生的指令转变为特定打印机所认可的协议。打印驱动程序将自动分析处理程序送来的控制码和数据,从而使字符或图象在打印机上输出。

早期的打印驱动程序在启动后便常驻内存,现在对某些汉字系统则是需打印时才装入,打印完后便将它从内存中消去,以便留出更多的可用内存给其它程序。

不同的打印机的控制码有可能是不同的(参考各打印机使用说明书),所以这也是不同的打印机有不同的驱动程序的原因之一。因为在程序执行时预定义流 `stdprn` 将打开,所以可以使用函数 `fprintf()` 将控制码或数据送往打印机。

```
C>TYPE PRINT10.C
#include "stdio.h"
main()      /* 将控制码按字符型发送给 AR3240 点阵打印机 */
{
    char *initprinter="\x1b\x40";      /* 打印机复位控制码 */
    char *underlinestart="\x1c\x2d\x1"; /* 产生下画线控制码 */
    char *underlineend="\x1c\x2d\x0";  /* 取消下画线控制码 */
    fprintf(stdprn,"%s%s",initprinter,underlinestart); /* 送控制码 */
    fprintf(stdprn,"打印有下划线的汉字\n");          /* 送数据 */
    fprintf(stdprn,"%s",underlineend);
}
```

注意:对行缓冲打印机只有在送出 '\n' 后才会有输出。

还有一种方法是使用 `%c` 格式而不用 `%s` 格式,如

```
fprintf(stdprn,"%c%c%c",27,106,1); /* AR3240 逆向跳一行 */
```

相当于执行控制码 `ESC j n`,语句中用的是十进制值。

发送控制码也可以靠连续调用 `bdos(5,dx,0)`;实现,这里 `dx` 是控制码中各码的整数值。

当然也可以用 `biosprint()` 函数发送。例如,对打印机初始化可用

```
biosprint(0,0x1b,0);
biosprint(0,0x40,0);
```

这两个语句实现。

由于预定义流 `stdprn` 占用文件句柄为 4,所以有时也可考虑用

```
write(4,"Hello\n",6);
```

向打印机输送6个字符。

对一个指针所指的字符串

```
char *string;
```

也许可用

```
while(*string)fputc(*string++,stdprn);
```

这样的循环输出语句输出整个字符串。

懂得向打印机发送控制码的方法后就不难在程序中直接设置控制打印机动作的程序段了。

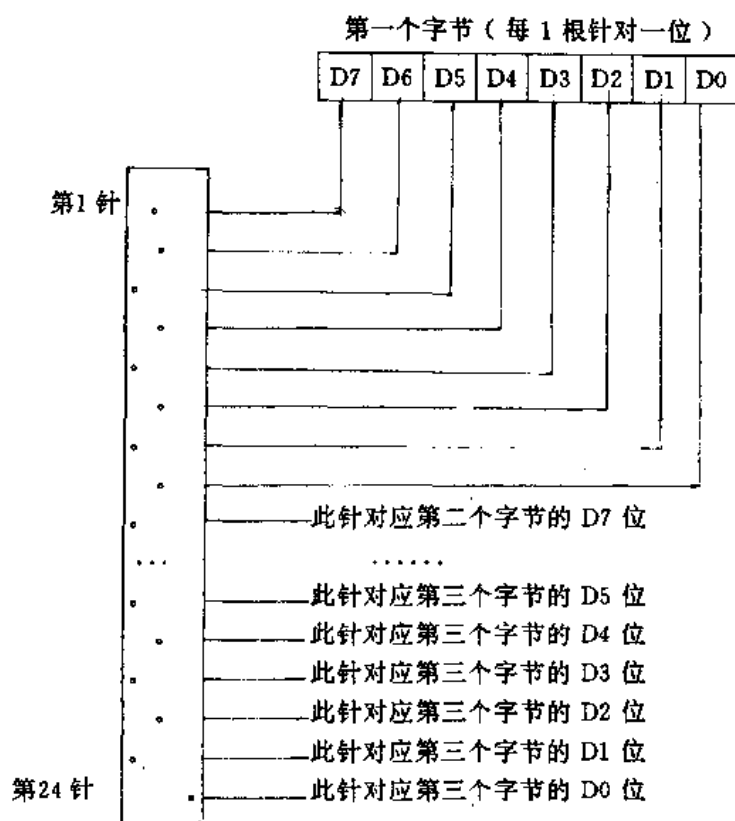


图 34-2 24 针图象排列

二 选择图象模式的命令

要想打印图象就必须给打印机发送图象打印命令。AR3240 有一个选择图象打印的命令是

ESC * m n1 n2 d11 d12 d13 d21 d22 d23 dk1 dk2 dk3

其中:

表 34-6

m 值 针数 每英寸点数 (DPI) $n1+n2 * 256$ 最大值 (一行上的列数)

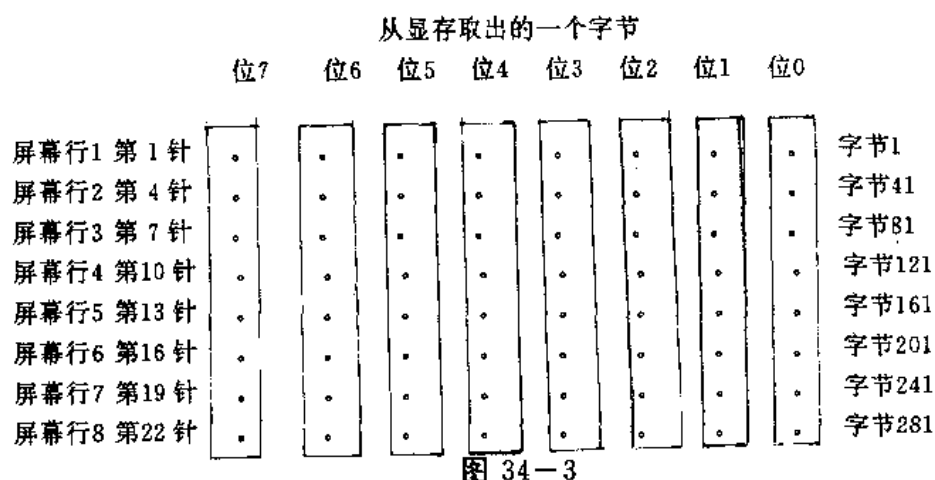
0	8	60	816
1	8	120	1632
2	8	120	1632
3	8	240	3264
4	8	80	1088
6	8	90	1224
32	24	60	816
33	24	120	1632
38	24	90	1224
39	24	180	2448
40	24	360	4896

m 称为所选图象模式。 $n=n1+n2 * 256$ 是组成图象的列数。 dij 为组成图象数据, 每个

为一字节。对 8 针图象排列,每一列是一个图象数据,即一个字节。如把组成一列的图象数据的字节数称为一项数据,则对 8 针排列应有 n 项数据;而对 24 针图象排列,每一列是由三个图象数据组成,因此它需要 $3n$ 个图象数据。在送参数时除了要注意书写顺序外,还特别要注意这种严格的一一对应关系(不能多也不能少),否则会产生错误的结果。

三 屏幕显示象点和打印列的关系

在图形方式下,从显示象点的内存中读出一个字节,该字节对应屏幕一行上的 8 个象点,或者说每个象点占一位。另一方面,打印针是纵向排列的,换句话说一根针对应屏幕一行上的一象点,对 8 针图象排列,将对应连续的 8 行;对 24 针图象排列,则对应屏幕上连续的 24 行。实际上打印屏幕图象的操作是先从显存中取出字节,然后再通过打印针打出的。由此不难理解它们之间的对应关系。图 34-3 (8 针图象模式)为打印屏幕左上角象点的示意图(取出 8 个字节、打印 64 个象点)。



从图 34-3 可以看出,一行象点在显存中要占 40 个字节($320/8=40$)。为打印此 64 点应一次先取出 8 个字节,放到一个缓冲区中,然后先取每个字节的头一个位,打印一列的 8 个象点;然后再取出每个字节的第二位,再打印一列,如此等等。通过对字节进行移位和与操作便可得到字节的位值。位值为 1 便打印,否则不打印。

因为打印机的纵横比(每英寸的水平点数与垂直点数之比)为 1:1,而显示的纵横比可能为 5:6,所以屏幕拷贝的图形一般要变形。为此可能要调整纵横比,例如调整打印机行距等。

四 屏幕拷贝举例

1. 720x348 单色显示器

屏幕缓冲区首址为 0xB0000000,一行 720 列占 90 个字节($720/8=90$)。象点阵分 4 页存放,每页大小为 2000H。第一页放第一行、第五行,...;第二页放第二行、第六行,...;第三页放第三行、第七行,...;第四页放第四行、第八行,...;如此等等。对 348 行每次取出 8 行,因 348 除以 8 等于 43 余 4,故要取 44 次。

```
C>TYPE PRINT11.C
#include "dos.h"
#include "bios.h"
#include "mem.h"
```



```

void interrupt (* oldint5)(void);
char ch—mask[8]={0x80,0x40,0x20,0x10,0x08,0x04,0x02,0x01};
/* 一个字节的各位值,用于测试某位是否要打印 */
void word—8(int ln,char mask)
{
    int wn,lk,col;
    char far * pageptr; /* 指向显示页某字节的指针 */
    char ch;
    char cc=0; /* 置初值,下面求出各位值 */
    pageptr=(char far *)farmalloc(sizeof(char)); /* 动态分配一个字节 */
    lk=ln; /* 准备指向同页的另一字节 */
    if(ln<43*180)col=8; /* 开始全是一次打印 8 行 */
    else col=4; /* 最后只有 348%8=4 行 */
    for(wn=0;wn<col;wn++) /* 对垂直方向的 8 行的各行同一列上的象点处理 */
    {
        pageptr=MK—FP(0xb000,ln); /* 指针在 4 页显存中来回移动 */
        memcpy(&ch,(char *)pageptr,1); /* 拷贝一字节显存到变量 ch 中 */
        if((ch & mask) == mask)cc +=mask; /* 如果指定位非 0,则写该位 */
        if(wn==3)ln=lk+90; /* 取同一页的另一行 */
        else ln +=0x2000; /* 翻页,每页为 8192 字节 */
    }
    biosprint(0,cc,0); /* 打印一个字节 */
}
void p—time—8line(int byte)
{
    int i,j;
    for(i=0;i<90;i++,byte++) /* 一行象点需用 90 个字节存储 */
    {
        for(j=0;j<8;j++) /* 每一字节对应 8 个象点 */
            word—8(byte,ch—mask[j]); /* 每次处理一列上 8 个象点 */
    }
}
void interrupt graph—p(void) /* 屏幕打印程序主要部分 */
{
    int n;
    int byteno=0; /* 字节数 */
    biosprint(0,0x1b,0); /* 在 LPT1 上输出字符 ESC=0x1b, 以及 @=0x40 */
    biosprint(0,0x40,0); /* ESC+@ 是初始化打印机,打印机缓冲区中数据被清除 */
    biosprint(0,0x1b,0); /* 此三个语句送 ESC+A+8,置打印行距为 8/72 英寸 */
    biosprint(0,0x41,0);
    biosprint(0,0x8,0);
    for(n=0;n<44;n++) /* 对屏幕象点作 44 次扫描,即打印 44 次才能将屏幕打完 */
    {
        biosprint(0,0x1b,0); /* 此四句送 ESC+L+0xd0+0x02,即置双倍密度位映象方式 */
        biosprint(0,0x4c,0); /* 0xd0+0x02*256=720 点,即每行位映象数据为 720 列 */
        biosprint(0,0xd0,0);
    }
}

```

```

biosprint(0,0x02,0);
p--time--8line(byteno); /* 打印一次打出屏幕上 8 行 */
byteno +=180; /* 因取一次垂直 8 行,每一页要取二行,每行占 90 个字节 */
}
biosprint(0,0x1b,0); /* 置行距为正常行距:六分之一英寸 */
biosprint(0,0x41,0);
biosprint(0,0xc,0);
}
void change--int5(void interrupt (*fun)())
{
    oldint5=getvect(5); /* 取 BIOS 中断 INT 5H,屏幕打印 */
    setvect(5,fun); /* 用新中断 fun 代替原 INT 5H */
}
/* void reset--int5(void interrupt (*f1)())
{
    setvect(5,oldint5); /* 可恢复原 BIOS 中断 INT 5H
}
*/
main() /* 在 EPSON 系列九针打印机上对单显图形拷贝示例 */
{
    /* 根据王迎的程序改写并注释 */
    /* 画出图形等语句 */
    change--int5(graph--p);
    /* 此后按 PrtSc 或 Print Screen 键后由 INT 9H 中断处理例程启动
       (也可被应用程序直接激发),打印屏幕图形。 */
}

```

2. CGA 显示器

CGA 在显示方式 4 时分辨率为 320x200,即横向 320 列,水平 200 行。每个字节存有 4 个象点的信息,每个象点占字节的两位,因此可表示 4 种颜色。一行上为 320 个象点。显存的起始地址是 B800:0000。所有偶数行 (0,2,4,...,198) 的象点从 B800:0000 开始连续存放,而所有奇数行 (1,3,...,199) 的象点从 BA00:0000 开始连续存放。它们在内存中各占 8K 字节。在每一个字节内,各象点按它们在屏幕上从左到右的顺序存放的,即最左边的一个象点占据位 7 和位 6,最右边的一个象点占据位 1 和位 0。因此,只要表示一个象点的两位不是 00,则该象点将被打印。

```

C>TYPE PRINT12.C
#include <dos.h>
#include <stdio.h>
#include <graphics.h>
char statuse(),put--out();
int printscr(),printscr8(),printscr24();
void mode(),palette();
main() /* 本程序演示对 CGA 显示器、用 AR3240 打印机进行屏幕硬拷贝 */
{
    /* 内中将屏幕图象与 AR3240 的两种图象打印方式作了对照说明 */
    int graphdrive=DETECT,graphmode;
    initgraph(&graphdrive,&graphmode,"");
    mode(4); /* 模式 4 为 CGA 图形方式,320X200,25X40 */
}

```

```

palette(0);          /* 置调色板 0 */
setfillstyle(SOLID—FILL,MAGENTA); /* 设置填充模式和颜色 */
bar3d(100,10,200,100,5,1);      /* 画三维条形图 */
setfillstyle(HATCH—FILL,RED);
bar(30,30,80,80);      /* 画二维条形图 */
/* 对 8 针打印, 下句改成 printscr(8); 不打印为 printscr(0) */
printscr(24);
getch();
mode(3);              /* 模式 3, 文本方式 */
}
void palette(int pnum) /* 设置调色板 */
{
    union REGS r;
    r.h.ah=11;
    r.h.bh=0;
    r.h.bl=pnum;
    int86(0x10,&r,&r);
}
void mode(int mode—code) /* 调用 BIOS 设置显示模式 */
{
    union REGS r;
    r.h.ah=0;
    r.h.al=mode—code;
    int86(0x10,&r,&r);
}
printscr(int pointnum) /* 分析所选打印图象方式, 8 针或 24 针 */
{
    switch(pointnum)
    {
        case 8:
            printscr8(pointnum);
            break;
        case 24:
            printscr24(pointnum);
            break;
        default:
            break;
    }
}
printscr8(int reverse) /* 用 AR—3240 打印机的 8 针打印 */
{
    int y;
    static unsigned char out—buff[324],byte[8],bytes;
    int i,j,k,m,mask=0xc0;
    unsigned char xor;
    /* 指向 CGA 显示 RAM 的起始地址 B800:0000H */

```

```

char far * ptr=(char far *)0xb8000000;
if(reverse)
{
    /* 置行距:<ESC>An=chr$(27);chr$(65);chr$(7)=7/60 英寸 */
    fprintf(stdprn,"\x1b\x41\x7");
    for(y=0;y<25;y++)
    {
        if(kbhit()) /* 打印中按任一键将中断打印 */
        {fputs("\x1b\x41\x9",stdprn);
         getch();
         break;
        }
        for(k=0;k<80;k++)
        {
            byte[7]=*ptr;          byte[6]=*(ptr+8192);
            byte[5]=*(ptr+80);      byte[4]=*(ptr+8272);
            byte[3]=*(ptr+160);     byte[2]=*(ptr+8352);
            byte[1]=*(ptr+240);     byte[0]=*(ptr+8432);
            for(j=0;j<4;j++)
            {
                bytes=0;
                m=0x01;
                for(i=0;i<8;i++)
                { xor=byte[i] & 0x80;
                 if(xor==0x80 || (byte[i] & mask) >0)bytes|=m;
                 m<<=1;
                 byte[i]<<=2;
                }
                out—buff[j+k*4+5]=bytes;
            }
            ptr++;
        }
        ptr+=240;
    }
    /* 以下赋值使你可以和打印机说明书上列出图案比较
    out—buff[5]=1;   out—buff[6]=3;   out—buff[7]=7;   out—buff[8]=15;
    out—buff[9]=31;  out—buff[10]=63;  out—buff[11]=127; out—buff[12]=255;
    out—buff[13]=255; out—buff[14]=127; out—buff[15]=63;  out—buff[16]=31;
    out—buff[17]=15;  out—buff[18]=7;   out—buff[19]=3;   out—buff[20]=1;
    */
    /* 设置图象模式:<ESC>*m n1 n2
    <ESC>=0x1b; * =\0x2a; m=6 (CRT I 模式);
    组成图象列数=325=69+1x256; n1=69=0x45; n2=1=0x1 */
    fprintf(stdprn,"\x1b\x2a\x06\x45\x01");
    for(i=0;i<325;i++) /* 打印一行 */
        put—out(out—buff[i]);
    put—out('\r');
}

```

```

        put—out('\n');
    }
    fputs("\x0c",stdprn);
    fflush(stdprn);
    /* 进纸换页 */
    /* 清除打印机缓冲区内容 */
}
printscr24(int reverse)
{
    int y;
    static unsigned char out—buff[2][329],byte[2][8],bytes;
    int i,j,k,n,m,mask=0xc0;
    unsigned char xor;
    char far *ptr=(char far *)0xb8000000;
    if(reverse)
    {
        fprintf(stdprn,"\x1b\x41\x7");
        for(n=0;n<3;n++)
        for(i=0;i<9;i++)
            out—buff[n][i]=0;
        for(y=0;y<9;y++)
        {
            if(kbhit())
            {fputs("\x1b\x41\x18",stdprn);
             getch();
             break;
            }
            for(k=0;k<80;k++)
            {
                byte[0][7]=*ptr;        byte[0][6]=*(ptr+8192);
                byte[0][5]=*(ptr+80);    byte[0][4]=*(ptr+8272);
                byte[0][3]=*(ptr+160);   byte[0][2]=*(ptr+8352);
                byte[0][1]=*(ptr+240);   byte[0][0]=*(ptr+8432);
                if(y==8)
                {
                    byte[1][7]=0;    byte[1][6]=0;
                    byte[1][5]=0;    byte[1][4]=0;
                    byte[1][3]=0;    byte[1][2]=0;
                    byte[1][1]=0;    byte[1][0]=0;
                    byte[2][7]=0;    byte[2][6]=0;
                    byte[2][5]=0;    byte[2][4]=0;
                    byte[2][3]=0;    byte[2][2]=0;
                    byte[2][1]=0;    byte[2][0]=0;
                }
            }
            else
            {
                byte[1][7]=*(ptr+320);  byte[1][6]=*(ptr+8512);

```

```

        byte[1][5]=*(ptr+400); byte[1][4]=*(ptr+8592);
        byte[1][3]=*(ptr+480); byte[1][2]=*(ptr+8672);
        byte[1][1]=*(ptr+560); byte[1][0]=*(ptr+8752);
        byte[2][7]=*(ptr+640); byte[2][6]=*(ptr+8832);
        byte[2][5]=*(ptr+720); byte[2][4]=*(ptr+8912);
        byte[2][3]=*(ptr+800); byte[2][2]=*(ptr+8992);
        byte[2][1]=*(ptr+880); byte[2][0]=*(ptr+9072);
    }
    ptr++;
    for(n=0;n<3;n++)
    {
        for(j=0;j<4;j++)
        {
            bytes=0;
            m=0x01;
            for(i=0;i<8;i++)
            {
                xor=byte[n][i]&0x80;
                if(xor==0x80 || (byte[n][i]&mask)>0)bytes|=m;
                m<<=1;
                byte[n][i]<<=2;
            }
            out_buff[n][j+k*4+9]=bytes;
        }
    }
}

/* out_buff[0][0]=1; out_buff[1][0]=1; out_buff[2][0]=1;
   out_buff[0][1]=3; out_buff[1][1]=3; out_buff[2][1]=3;
   out_buff[0][2]=7; out_buff[1][2]=7; out_buff[2][2]=7;
   out_buff[0][4]=15; out_buff[1][4]=15; out_buff[2][4]=15;
   out_buff[0][5]=31; out_buff[1][5]=31; out_buff[2][5]=31;
   out_buff[0][6]=63; out_buff[1][6]=63; out_buff[2][6]=63;
   out_buff[0][7]=127; out_buff[1][7]=127; out_buff[2][7]=127;
   out_buff[0][8]=255; out_buff[1][8]=255; out_buff[2][8]=255;
   out_buff[0][9]=255; out_buff[1][9]=255; out_buff[2][9]=255;
   out_buff[0][10]=127; out_buff[1][10]=127; out_buff[2][10]=127;
   out_buff[0][11]=63; out_buff[1][11]=63; out_buff[2][11]=63;
   out_buff[0][12]=31; out_buff[1][12]=31; out_buff[2][12]=31;
   out_buff[0][13]=15; out_buff[1][13]=15; out_buff[2][13]=15;
   out_buff[0][14]=7; out_buff[1][14]=7; out_buff[2][14]=7;
   out_buff[0][15]=3; out_buff[1][15]=3; out_buff[2][15]=3;
   out_buff[0][16]=1; out_buff[1][16]=1; out_buff[2][16]=1;
   */

/* 0x9 为横向制表符 */
fprintf(stdprn,"\x9\x9\x1b\x2a\x21\x49\x01");
for(i=0;i<329;i++)

```

```

        {
            put—out(out—buff[0][i]);
            put—out(out—buff[1][i]);
            put—out(out—buff[2][i]);
        }

        put—out('\r');
        put—out('\n');
        ptr+=880;
    }
    fputs("\x0c",stdprn);
    fflush(stdprn);
}
}
char status()                /* 检查打印机忙状态 */
{
    union REGS regs;
    regs.h.ah=2;
    regs.x.dx=0;
    int86(0x17,&regs,&regs);
    return(regs.h.ah&0x80);
}
char put—out(char character)    /* 打印一个字符 */
{
    union REGS regs;
    while(! status());        /* 只检查打印机是否有空 */
    regs.h.ah=0;
    regs.h.al=character;
    regs.x.dx=0;
    int86(0x17,&regs,&regs);
    return(regs.h.ah);
}

```

在这个程序里我们又用了 stdprn ,但又用了 put—out() ,显然可以将通过 stdprn 输出的字符调用 put—out() 输出。因为 stdprn 是作为一个预定义流打开的,缺省的打开方式是文本方式,而不是二进制方式。虽然可以用 freopen() 在二进制方式下重新打开这种标准流,但发送图形字符的最快最好的方法还是直接使用 BIOS ,而 put—out() 就是这样。特别在一些不兼容的汉字系统下调用 BIOS 更有利。

3. VGA 显示器

可用取屏幕象点颜色直接获得打印图象字节值。例如对 640x480 高分辨图形方式时,象点颜色码非 0 时处理为 1,否则便为 0,以此构成打印字节。先在屏幕列方向取一列、行方向取 24 个像素,对它们处理后可得 3 个打印字节;接着又往右取一列,按此法处理,等等。等全部列(640)处理完后打印机进一行,行距应等于 24 个打印针点间的距离。

```

C>TYPE PRINT13.C
#include <stdio.h>
#include <graphics.h>
putbit(unsigned char c,int pp,int bit0) /* 处理象点 */

```

```

{
    if(pp==0) c &= 0xFE;          /* 将位 0 置成 0      */
    else c |= 0x1;                /* 将位 0 置成 1      */
    if(bit0 != 7) c = c << 1;     /* 依次处理位 7 到 1 */
    return c;
}

p=screen()                       /* 每次取 24 行从左往右打印 */
{
    /* 从上往下每次取 24 行 */
    int times;
    int y,x,bit,p;
    unsigned char byte[3][640];
    fprintf(stdprn,"%s\n","\\0x1b\\0x4a\\0x8");
    /* ESC+J+n 为置 n/120 英寸行距 */
    for(times=0;times<20;times++) /* 打印整个屏幕需要 480/(8*3)=20 次 */
    {
        for(y=0;y<640;y++)
            for(x=0;x<3;x++) byte[x][y]=0; /* 初始化 */
        for(y=0;y<640;y++) /* 将屏幕上各象点值存入数组 */
            for(x=0;x<3;x++)
                for(bit=0;bit<8;bit++)
                {
                    p=getpixel(y,times*24+x*8+bit);
                    /* p 可能为 0 或非 0, 非 0 不一定为 1 */
                    if(p != 0) p=1; /* 得象点颜色值 0 或 1 */
                    byte[x][y]=putbit(byte[x][y],p,bit);
                } /* byte[x][y] 为垂直方向一字节值 */
        fprintf(stdprn,"%c%c%c%c",27,71,2,128); /* ESC+G+n1+n2 位映象单向打印 */
        for(y=0;y<640;y++) /* 从屏幕的左边打到右边 */
            for(x=0;x<3;x++) /* 每次打 24 象素行,即 3 个字节值 */
                biosprint(0,byte[x][y],0);
        fprintf(stdprn,"%s\n","\\0xd\\0xa"); /* 以当前行距进行 */
    }
}

main() /* 直接从屏幕上取象点,并在 M1724 打印机上打印屏幕图象 */
{
    int driver=VGA, mode=VGAHI; /* VGA 高分辨率 640x480 */
    char *path="c:\\tc";
    initgraph(&driver,&mode,path);
    circle(150,150,60); /* 画一个圆 */
    printf("ABCDEFGH\n"); /* 屏幕左上角印出 (如要在西文下打汉字,应将汉字从字库中取出后再调用 putpixel 逐点显示汉字) */
    fflush(stdprn);
    p=screen(); /* 在打印机上打出图形 */
}

```

从上可见,对图象打印应当了解的有:

(1) 显示缓冲区,即显存情况;

(2) 打印针的排列方法和图象打印命令；

(3) 怎样获取象点并把它组成一个打印点阵；在此基础上，只要

<1> 改变打印点阵结构

<2> 动用不同的打印针数

<3> 改变垂直走纸的距离

<4> 改变打印针与象点的对应关系（例如一象点可以对应多根打印针）就有可能对同一屏幕图形得到不同输出方向或比例的打印图。

最后应了解如何调用 BIOS 而不用 stdprn，以及当象点数不能被整除时的处理。

注意：有的汉字操作系统自配有屏幕图象打印程序，并可常驻内存。

第三十五章 视频函数

35.1 Turbo C 涉及的显示卡

连接主机和【显示器】（或【监视器】）的【显示卡】（或【图形适配器】、【图形卡】）包括一个 9（或 15）针接头，以及一块可以插在主机板插槽内的的板子。板上的集成电路可以产生控制屏幕的图形信号。

Turbo C 涉及的几种图形卡种类参见表 35-1。

表 35-1 Turbo C 图形适配器与图形驱动程序

图形适配器	驱动程序文件 (* .BGI)	驱动程序号 graphics—drivers
CGA	CGA.BGI	1=CGA
MCGA	CGA.BGI	2=MCGA
EGA	EGAVGA.BGI	3=EGA
EGA64	EGAVGA.BGI	4=EGA64
EGAMONO	EGAVGA.BGI	5=EGAMONO
IBM8514	IBM8514.BGI	6=IBM8514
HERULES	HERC.BGI	7=HERCMONO
AT&T400	ATT.BGI	8=ATT400
VGA	EGAVGA.BGI	9=VGA
PC3270	PC3270.BGI	10=PC3270

说明：使用 TVGA 卡时如果用 9 到 15 针的转接电缆将 D 型 9 针监视器插头连接到 D 型 15 针插座上，应符合表 35-2 要求（仅供参考，视频模拟信号，黑色电平为 0V，全亮电平为 +0.7V）。

表 35-2 9 到 15 针转换表

9 针插座信号	插针号	15 针插座信号	插针号
红	1	红	1
绿	2	绿	2
兰	3	兰	3
水平同步	4	水平同步	13
垂直同步	5	垂直同步	14
红地	6	返回红地	6
绿地	7	返回绿地	7

兰地	8	返回兰地	8
同步地	9	数字地	10
		地	5

注:对单色显示器管脚 12 接地,对彩色悬空。

图形显示器和显示卡的种类很多,它们甚至互不兼容。下面列出一些常见的显示卡如 CGA、EGA 和 VGA 的主要功能,供读者参考。

1. MDA 卡

单色显示卡(Monochrome Display Adapter)。

2. HGC 卡

它是单色显示卡(Hercules Graphics Card),它可以在同一个显示器上显示图形和 清晰的文字。

3. CGA 卡

它又称彩色图形卡(Color Graphics Card),它可以显示以点绘制的图形和 ASCII 码字符。

4. MCGA 卡

又称多彩色图形阵列(Multi-Color Graphics Array)。MCGA 与 CGA 不同之处是它有一个数字模拟转换器(【DAC】),因此可从 262144 个调色板(palette)中同时显示 256 种颜色。除了能模拟彩色显示外,它还能模拟单色显示。对单色显示,它可以显示 64 种灰度。

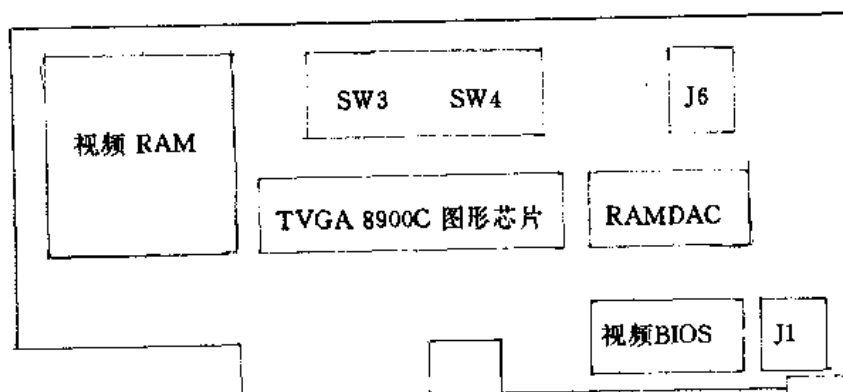
5. EGA 卡

又称加强型图形卡(Enhanced Graphics Adapter)。EGA 比 CGA 分辨率高,可显示比 CGA 更清晰的图象和 16 种颜色,并能模拟单色显示卡 MDA(它可以在单色显示卡上显示以点绘制的图形)。

6. VGA 卡

又称视频图形阵列(Video Graphics Array),它用单一的集成片代替了 EGA 的几个片子。对 EGA 上能运行的程序不经修改便能在 VGA 上运行。VGA 的分辨率又比 EGA 高(如文本方式为 720×400 ,图形方式为 640×480)。注意:它也有一个 DAC。

IBM8514 是 IBM PS/2 的监视器,它的显示模式基本上同 VGA。



注:1. 跳线 J1 控制中断请求 IRQ9,缺省是不控制,使 LAN 网络卡在系统上运行 不会发生中断冲突。2. 跳线 J6 用于显示器检测(非标准彩色或非标准单色)的控制。3. SW3 用于选择扫描频率。4. SW4 用于允许/禁止 BIOS 自动检测。

图 35-1TVGA 图形显示卡主要元器件示意图

显示卡上有一块用于存储映射屏幕图象的数据缓冲区,称为【视频 RAM, VRAM】)或显示存储区,它是视频缓冲区的物理定位处。依据固有的时钟及扫描格式扫描显示卡上的 VRAM,从而实现在 CRT 上信息显示。显示存储器由动态显示存储器件构成。对 TVGA 卡由多片(2片、4片或8片)快速页模式 DRAM 构成。各种卡上 VRAM 容量不等,如

MDA: 4K

CGA: 16K

EGA: 64K 或 256K

VGA: 256K、512K 或 1024K

测定显示卡类型可用 INT 11H 中断大体判断(参见本章 initgraph() 函数)。

注:有些显示器上有设置显示方式的开关;对某些操作环境,也许要使用视频驱动程序(参见一些显示器说明书)。有些显示卡上还有视频 ROM。像 MDA、CGA 和 HGC 卡的视频 ROM 里含有字符定义表,它不在 CPU 的地址空间内,只有硬件的字符发生器可以读它们。所以在文本方式下,这些显示卡的显示的字符的形状不能由程序控制。例如,对 COMPAQ II PORTABLE(便携式),要用 400 线模式前,要在 DOS 提示符下键入 SET SEL = CGA。

35.2 显示卡的体系结构

不同显示卡的基本结构可能不相同,EGA/VGA 卡控制参见图 35—2。下面我们将简要介绍与它相关的一些存储器的功能。除了在对寄存器编程时要用到这些存储器外,读者凭此可以增加对视频系统整体性的认识。

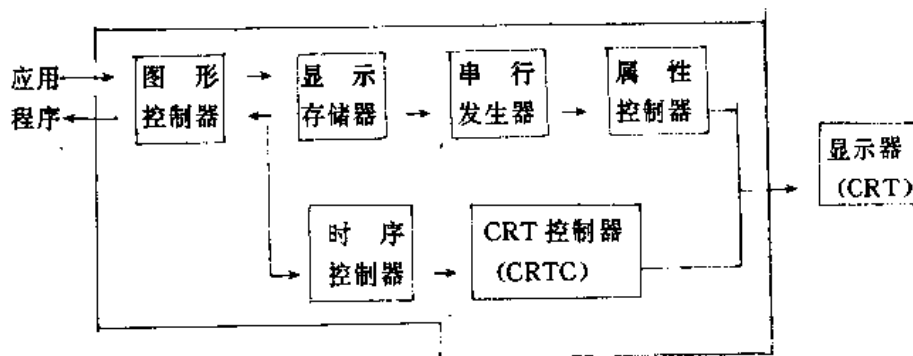


图 35—2 EGA/VGA 显示卡控制示意图

一 CRT 控制器

程序员要想设置屏幕模式、产生并控制光标、定义颜色等就要了解视频系统。CGA 和 EGA 等视频系统都是以 Motorola 6845 CRTC(阴极射线管控制器)芯片为核心。VGA 的 CRTC 与 EGA 的不完全相同。6845 的基本任务是翻译 ASCII 码数,并从适配器 ROM(有时是 RAM)中取出对应字符的数据,对数据解码以得出属性或颜色,据此调整屏幕显示。此外还生成并控制光标。注意:EGA 将上述部分功能交给其它芯片处理。

6845 内部寄存器有 19 个,其中一个作内部地址寄存器,称【索引寄存器】(简称 AR),余下的 18 个寄存器是【数据寄存器】(简称为 R0 ~ R17)。它们都是 8 位寄存器。前 10 个寄存器(R0 ~ R9)用于确定水平及垂直方向显示参数,它由 BIOS 自动对其调整,程序员不应轻易对其访问,否则会产生意想不到的问题。

18 个数据寄存器通过同一个端口地址访问,在单色卡上该口为 3B5H,在彩色卡上为 3D5H,EGA 则两种地址都可用。由于 6845 不具有直接访问 R0 ~ R17 的地址端口,因此在访问这些寄存器之前,需将这些寄存器的编号(0 ~ 17)先送入 AR 索引寄存器(这表示选通了某编号的寄存器)。对单色卡其端口地址为 3B4H(彩色卡为 3D4H)。然后,将需写入值送入端口 3B5H(或 3D5H)。

(一) 地址寄存器和数据寄存器的端口地址

要访问数据寄存器,必须先将数据寄存器的编号通过访问地址寄存器的端口送入地址寄存器,然后通过数据寄存器的端口将数据送入数据寄存器。

表 35-3 地址寄存器和数据寄存器的端口

寄存器名	MDA	HGC	CGA	MCGA	EGA/VGA
地址寄存器端口	3B4H	3B4H	3D4H	D4H	B4H(单色)或3D4H(彩色)
数据寄存器端口	3B5H	3B5H	3D5H	D5H	B5H(单色)或3D5H(彩色)

(二) 数据寄存器的内容

1. MDA、CGA 和 HGC 的数据寄存器的内容

表 35-4-1

寄存器 编 号	内 容	读/写	文本方式初值		图形方	
			单色	彩色, 40×25	80×25	式初值
R0	水平总字符数	只写	61H	38H	71H	38H
R1	水平显示字符数	只写	50H	28H	50H	28H
R2	水平同步位置	只写	52H	2DH	5AH	2DH
R3	水平同步脉冲宽度	只写	0FH	0AH	0AH	0AH
R4	垂直总字符数	只写	19H	1FH	1FH	7FH
R5	垂直总调整	只写	06H	06H	06H	06H
R6	垂直显示字符数	只写	19H	19H	19H	64H
R7	垂直同步位置	只写	19H	1CH	1CH	70H
R8	隔行或逐行扫描方式	只写	02H	02H	02H	02H
R9	最大扫描线	只写	0DH	07H	07H	01H
R10	光标起始光栅线	只写	0BH	06H	06H	06H
R11	光标结束光栅线	只写	0CH	07H	07H	07H
R12	(页)起始地址高位	只写	00H	00H	00H	00H
R13	(页)起始地址低位	只写	00H	00H	00H	00H
R14	光标位置高位	读写	00H			
R15	光标位置低位	读写	00H			
R16	光笔高位	只读	00H			
R17	光笔低位	只读	00H			

2. EGA 和 MCGA 数据寄存器的内容

表 35-4-2

寄存器编号	MCGA 内 容	读/写	EGA 内 容	读/写
R0	水平总字符数	读/写	水平总字符数	只写
R1	水平显示字符数	读/写	水平显示允许结束	只写
R2	水平同步位置	读/写	开始水平扫描消隐	只写
R3	水平同步脉冲宽度	读/写	结束水平扫描消隐	只写
R4	垂直总字符数	读/写	开始水平回扫	只写
R5	垂直总调整	读/写	结束水平回扫	只写
R6	垂直显示字符数	读/写	垂直总调整	只写
R7	垂直同步位置	读/写	溢出存放	只写
R8	(保留)	预置行扫描	只写	
R9	每字符扫描线	读/写	最大的扫描线地址	只写
R10	光标起始光栅线	读/写	光标起始光栅线	只写
R11	光标结束光栅线	读/写	光标结束光栅线	只写
R12	起始地址高位	读/写	起始地址高位	读/写
R13	起始地址低位	读/写	起始地址低位	读/写
R14	光标位置高位	读/写	光标位置高位	读/写
R15	光标位置低位	读/写	光标位置低位	读/写
* R16	模式控制	读/写	垂直回扫开始	只写
			光笔高位	只读
* R17	中断控制	读/写		
* R18	字符发生器,同步极性	读/写	垂直回扫结束	只写
		光笔低位		只读
* R19	字符发生器指针	读/写	垂直显示允许结束	只写
R20	字符发生器计数	读/写	偏移量(逻辑线宽)	只写
R21			下划线置位	只写
* R22			开始垂直扫描消隐	只写
* R23			结束垂直扫描消隐	只写
* R24			模式控制	只写
R25			线比较	只写

注: * 表示修改可能导至危险的寄存器。

详细说明:

(1) R10: 设置光标起始(光栅)线

为了少占 I/O 口地址,一般访问寄存器采用多重编址方式,即第一步先取端口地址,然后对这个端口地址送要选择的寄存器编号,第二步取第二个端口并送数据。

```
MOV DX,3B4H    ;对单色卡选端口 3B4H (对彩色卡选 3D4H 等)
MOV AL,10      ;选寄存器 10,即编号 10 的寄存器
OUT DX,AL      ;送入端口 3B4H (或 3D4H 等) 中的地址寄存器,即送请求信号
MOV AL,0       ;起始线为第 0 线
INC DX         ;转端口 3B5H (或 3D5H 等) 访问数据寄存器
OUT DX,AL      ;起始线已送入寄存器
```

注意:寄存器为 8 位,设置起始线值时其第 5 位控制光标闪烁周期,第 6 位为 0 光标闪烁,为 1 不闪烁。

在 TC 中可用函数 outportb() 经指定端口送值,例如上述汇编程序段相当于语句

```
outportb(0x3b4,0xA);
outportb(0x3b5,0x0);
```

这种直接访问寄存器的方法也称为【寄存器编程】。由于中文操作系统常将视频中断 INT 10H 修改,所以当调用 BIOS 中断发生困难时用寄存器编程则能较好地解决问题。

原则上讲,这种方法可以用来初始化数据寄存器,问题是要求数据正确。例如用

```
static unsigned char TextTable[]={0x61,0x50,0x52,0x0f,0x19,0x06,
    0x19,0x19,0x02,0x0d,0x0b,0x0c,0x0,0x0,0x0,0x0,0x0,0x0};
register int i;
for(i=0;i<18;i++)
{
    outportb(0x3b4,i);
    outportb(0x3b5,TextTable[i]);
}
```

可以将单色卡数据寄存器初始化成文本模式。

(2) R11: 设置光标终止线

```
outportb(0x3b4,0xB);
outportb(0x3b5,0x7);    /* 终止线为第 7 线 */
```

(3) R12: 处理分页。页面起始地址的高位(只用低 6 位)。

(4) R13: 处理分页。页面起始地址的低位。R12 和 R13 是只写寄存器,它们决定在垂直消隐区域以后,作为刷新地址的第一个地址,即决定屏幕左上角的字符窗口所对应的存储单元的地址,故称【页顶寄存器】。修改其内容,可以实现按字符、行或页对画面卷动。

(5) R14: 存储光标位置的高位(只用低 6 位)。

(6) R15: 存储光标位置的低位。R14 和 R15 称【光标地址寄存器】或光标寄存器,它们是可读可写寄存器。CPU 读取其内容,可以取得下一个字符存放在 VRAM(视频 RAM)中的地址。当现行刷新地址和其内容相同时,6845 按 R10 和 R11 的规定输出光标信号。

(7) R16: 告知光笔位置,它是光笔的高位(只用低 6 位)。

(8) R17: 告知光笔位置,它是光笔的低位,R16 和 R17 是只读寄存器。

除 R14 ~ R17 寄存器外,其余的寄存器是只能写入而不能读出的。

EGA 有 6 个附加寄存器用于外观技术方面,例如 20 号寄存器决定一行字符在何处加下划线。

(9)MCGA 的前 16 个寄存器与 CGA 大都相似,只是它的 R9、R10 和 R11 中所存的值是两条扫描线,而 CGA 只是一条扫描线。它的 ASCII 码字符点阵具有 16 条扫描线的高度。

(10) 因 EGA 能显示 256 条以上的扫描线,所以 CRT 存放扫描线数目的寄存器要 9 个位,而不是 8 个位。这些寄存器的高位部分就存在 R7 寄存器中。

(11)VGA 的输入输出同 EGA,但 VGA 寄存器有更多的位。它没有提供使用光笔的功能。

二 外部寄存器

(一) 混合输出寄存器 (口地址 3C2H)

这个寄存器对 EGA 是只写的,对 VGA 可从口地址 3CCH 读回。不过使用它应小心,不要轻易用。

位7 垂直同步极性

位6 水平同步极性

位5 奇/偶页位

位4 禁止视频

位3 时钟选择 1

位2 时钟选择 0

位1 允许显示 RAM

位0 I/O 地址选择 (=0 选单色,=1 选彩色)

(二) 输入状态 (Status) 寄存器 1 (或对 EGA 为特征控制寄存器)

该寄存器告知显示状态信息。内中常有 2 个位表示 CRT 所产生的水平及垂直时序信号的当前状态 (这些位称状态位,不同卡的状态位有可能不同,参见表 35-5)。

表 35-5

显示卡	端口地址	位7	位3	位2	位1	位0
MDA	3BAH	①		②		
HGC	3BAH	③	①	④	②	
CGA	3DAH	③	⑤	④	③	
EGA	3BAH 或 3DAH	③	⑤	④	⑥	
VGA	3BAH 或 3DAH	③		⑥		
MCGA	3DAH		③		⑥	

说明:① 视频驱动

② 1 = 水平同步

③ 0 = 垂直同步

④ 1 = 光笔触发

⑤ 1 = 光笔开关关闭

⑥ 0 = 可以显示

(三) 模式控制器 (Mode control register)

它控制字符发生器及显示字符的速率。实际它决定显示方式（参见表 35—10），又称方式控制器（非 EGA/VGA 卡）。

1. 端口地址

表 35—6

MDA/HGC	CGA	MCGA (有 3 个)
3B8H	3D8H	3D8H, 3D4H/3D5H, 3DDH

2. MDA 模式控制器的位值

位0: 1 (永远为1)

位1、2、4、6、7: 0 (永远为0)

位3: 1 (视频启动, 能使图象更新), 0 (视频熄灭)

位5: 1 (字符可闪烁), 0 (不能闪烁)

3. CGA 和 MCGA 模式控制器 (端口 3D8H) 的位值

图 35—3 是与 3D8H 相关的模式控制器各位的功能。

7	6	5	4	3	2	1	0	
未用	1	0	1	1	0	0	0	40×25 黑白字符显示
		1	0	1	0	0	0	40×25 彩色字符显示
		1	0	1	1	0	1	80×25 黑白字符显示
		1	0	1	0	0	1	80×25 彩色字符显示
		×	0	1	1	1	0	320×200 黑白图形显示
		×	0	1	0	1	0	320×200 彩色图形显示
		×	1	1	1	1	0	640×200 黑白图形显示

图 35—3 模式控制寄存器

位0: 1 (80 字符数字模式即 80×25), 0 (40 字符数字模式即 40×25)

位1: 1 (320 列图形模式), 0 (其它模式, 如文本模式)

位2: 0 (对 CGA 可以有彩色; 对 MCGA 从 DAC 寄存器位 1 得到前景颜色)

1 (对 CGA 关闭彩色; 对 MCGA 从调色板寄存器 (端口 3D9H) 的 DAC 的位0~3 得前景)

位3: 0 (视频关闭, 屏幕为空), 1 (视频可见)

位4: 1 (640×200 列图形方式), 0 (其它方式)

位5: 在文本方式下, 1 (允许闪烁), 0 (不闪烁)

位6~位7: 0 (未用)

注意: 对 MCGA 的模式控制寄存器是可读可写的, 而 CGA 是只允许读的。

4. MCGA 端口为 3D4H/3D5H 的模式控制器 (Memory Controller) 的位值

位0: 1 (选择 320×200 的 256 种颜色模式), 0 (其它模式)

位1: 1 (选择 640×200 双色模式), 0 (其它模式)

位2: (保留)

位3: 1 (对显示方式计算水平调准参数), 0 (参数由寄存器 0~3 决定)

位4: 1 (总为1)

位5: (保留)

位6: 翻转垂直显示寄存器的位 8

位7: 1 (写保护寄存器 0~7), 0 (允许修改寄存器 0~7)

5. HGC 的模式控制器的位值

位0: (未用)

位1: 1 (选择 720×348 图形方式), 0 (80×25 文本方式)

位2: 0 (未用)

位3: 1 (视频可见), 0 (视频不可见)

位4: 0 (未用)

位5: 1 (可闪烁), 0 (不闪烁)

位6: 0 (未用)

位7: 图形方式, 1 (从 B800:0000 开始显示页号 1)

0 (从 B000:0000 开始显示页号 0)

6. HRC 的组合开关 (端口 3BFH) 寄存器 (亦称【x 模式寄存器】) 的位值

位0: 1 (图形方式), 0 (非图形方式)

位1: 1 (可用 B800:0000 开始的 32K 的视频缓冲区), 0 (不能用)

位2~位7: (未用)

(四) CGA 彩色选择寄存器 (端口 3D9H)

端口 3B9H (或 3D9H) 与屏幕颜色相关。图 35-4 是与 3D9H 相关的图形方式显示时彩色选择寄存器各位功能。该寄存器是只写的。

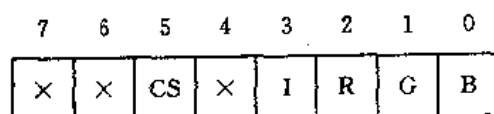


图 35-4 图形显示时彩色选择寄存器

低 4 位决定背景的 16 种彩色。对中分辨显示 320 (水平) × 200 (垂直) 点, VRAM 的一字节对应屏幕上 4 个点, 每两位构成一个点 (图 35-4)。每个点可以为 4 种彩色中的一种。彩色可由位 5 和两位位值决定。当位 5 为 0 时, 选择绿、红、黄, 否则选青、绛红、白颜色 (详见图 35-6)。

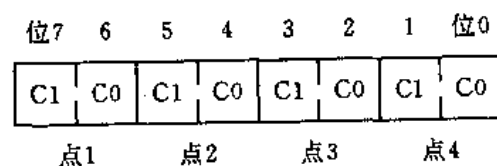


图 35-5 中分辨显示时的象点

C1	C0	位5 = 0	位5 = 1
0	0	16 种彩色中的一种 (由彩色选择寄存器低 4 位决定)	
0	1	绿	青
1	0	红	绛红
1	1	黄	白

图 35-6 中分辨显示时的象点颜色

位 4、6 和 7 未用。图 35-7 是文本方式时彩色选择寄存器情况。

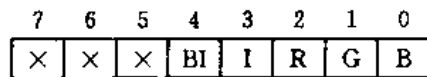


图 35-7 文本显示时彩色选择寄存器

低 4 位选择屏幕边沿的彩色 (16 种), 位 4 是决定字符背景的亮度位。

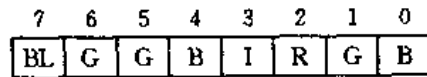


图 35-8 文本显示时字符属性的控制

对黑白字符, 位 0 ~ 2 控制前景; 位 4 ~ 6 控制背景; 位 3 为 0 字符为正常亮度, 为 1 为加强亮度; 位 7 为 0 表示字符不闪烁, 否则闪烁。

对彩色字符, 除将低 4 位 (0 ~ 3) 作为控制前景 (所以有 16 种色) 外, 其余位的作用和黑白是一样的。

当在 EGA 和 VGA 中建立了一种显示方式时, 就可以控制它们的时序控制器、图形控制器和属性控制器。注意, 对 EGA 上的控制器是只能写而不能读, 但对 VGA 控制器是可读可写的。

三 时序 (Sequencer) 控制器

它产生视频 RAM 所需的内部时序。有一个地址寄存器 (端口为 3C4H), 和 5 个数据寄存器 (端口为 3C5H)。访问它们的方法同访问 CRTC。

表 35-6

寄存器编号	说	明
* R0	复位 (Reset), 在复位时 (位 0 和位 1 值为 0) CPU 不能对显示存储器存取	
* R1	时钟脉冲方式 (Clocking Mode), BIOS 在模式选择时将它初始化, 软件通常不必修改它	
R2	位图屏蔽 (Map mask), 对彩色页面写允许寄存器, 位 0~位 3 对应 4 个页面, 值为 1 时表示写允许	
R3	字符变换选择 (Character Map Select), 又称字符发生器选择寄存器	
* R4	存储方式 (Memory Mode), BIOS 进行模式选择时对它初始化, 没有必要通过软件修改它	

注: * 表示修改将导致危险的寄存器

四 图形控制器 (Graphics Controller)

它主要处理视频缓冲区到属性控制器之间的数据传送, 它可对处理器读写数据进行逻辑操作。它的地址寄存器端口为 3CEH, 9 个数据寄存器的端口为 3CFH。

表 35-7

数据寄存器	说	明
R0	设置/复位 (Set/Reset), 填写数据寄存器, 用于图形控制器中的数据的预置或复位, 必须在写方式 0 时进行。位 0~位 3 分别对应于 4 个页面, 当位值为 0 时是复位, 为 1 时是预置	
R1	允许设置/复位 (Enable Set/Reset), 位 0~位 3 置 1 表示对应哪个页面允许填颜色, 为 0 时则不允许	
R2	颜色比较 (Color Compare)。用一个显示存储周期将四个页面的数据与参考颜色比较, 并报告对每个象素位置是否颜色匹配, 而不是一次读出四个页面正在给出的颜色是否存在	

- R3 数据循环 / 功能选择 (Data Rotate/Function Select)。写数据移位及对写入数据进行逻辑运算(在写方式 0 或 2 时)。例如对 VGA 有
- ```

outp(0x3ce,3); /* 选择 R3 */
outp(0x3cf,op); /* 送数据,确定逻辑功能:
 op=0x0 写入数据不变
 0x8 写入数据与处理器锁存器“与”
 0x10 写入数据与处理器锁存器“或”
 0x18 写入数据与处理器锁存器“异或” */
circle(x,y,r); /* 画一个圆 */
outp(0x3ce,3); /* 圆画出后重设为数据不变方式 */
outp(0x3cf,0);

```
- \* R4 读位图 (即页面) 选择 (Read Map Select), 决定哪一页允许读出
- R5 图形方式 (Graphics Mode), 定义当前图形控制器的操作方式位 0~位 100=写方式 0 01=写方式 1 10=写方式 2 11=写方式 3
- 位2 保留
- 位3 0=读方式 0 1=读方式 1
- 位4 0=CPU 按页面顺序读数据  
1=CPU 奇地址线访问页面 0 和 2, 偶地址线访问页面 1 和 3
- 位5 0=串行输出至属性控制器  
1=串行输出, 奇 (或偶) 数位由奇 (或偶) 数页面 给出, 仅用于显示方式 4 和 5
- 位6 1=256 种彩色方式
- 位7 保留
- \* R6 多功能 (Miscellaneous), 不允许修改 \* R7 颜色不考虑 (Color Don't Care), 在颜色比较周期屏蔽某些页面, 使它不测试, 仅用于图形控制读方式 1。位 0~位 3 对应 4 个页面, 当位值为 0 时不考虑
- \* R8 位屏蔽 (Bit Mask), 在读出一修改一写入周期屏蔽某些位使其不作修改
- 注: \* 表示修改将导致至危险的寄存器

## 五 属性控制器 (Attribute Controller)

它决定被显示的文本或图形的颜色, 提供一个 16 色调色板给 EGA 和 VGA。它的 1 个地址寄存器和 21 个数据寄存器共同使用一个输入输出, 端口地址为 3C0H。写到该端口的值是存在地址寄存器还是数据寄存器中, 要看属性控制器内部的触发器 (flip-flop) 状态而定。要设定触发器状态, 由 CRT 状态寄存器对 I/O 口 (对单色模式, 端口地址为 3BAH, 对彩色是 3DAH) 做读操作来初始化。初始化之后地址 3C0H 在第一个写周期将直接作为属性控制器的序号寄存器。例如, 对 VGA 可将属性控制数据寄存器的编号先写到属性索引寄存器 (端口 3C0H, 可读写) 中, 然后再向该端口写入属性控制数据寄存器的值。可从端口 3C1H 中读出属性控制数据寄存器的值。

### 1. 属性控制索引寄存器

它用于选择属性控制数据寄存器。

位 7~位 6 保留

位 5 调色板地址源

0=CPU 可访问调色板寄存器, 调色板可改变, 此时显示器全空白

1= 属性控制器可访问调色板寄存器

位4~位0 属性控制数据寄存器编号 (0~19)

## 2. 属性控制数据寄存器

表 35-8

| 数据寄存器                                                                                                         | 功   | 能   |
|---------------------------------------------------------------------------------------------------------------|-----|-----|
| R0~R15 调色板 (Palette)                                                                                          |     |     |
| 位7~位6 保留                                                                                                      |     |     |
|                                                                                                               | 对彩色 | 对单色 |
| 位5 第二显示卡红色 P5                                                                                                 | 0   | 0   |
| 位4 第二显示卡绿色 P4                                                                                                 | 增强  | 增强  |
| 位3 第二显示卡蓝色 P3                                                                                                 | 0   | 视频  |
| 位2 第一显示卡红色 P2                                                                                                 | 红   | 0   |
| 位1 第一显示卡绿色 P1                                                                                                 | 绿   | 0   |
| 位0 第一显示卡蓝色 P0                                                                                                 | 蓝   | 0   |
| R16 属性模式控制 (Attribute Mode Control)                                                                           |     |     |
| 位7 0= 调色板寄存器位5 和位4 由调色板寄存器驱动<br>1= 由彩色选择寄存器位1 和位0 驱动                                                          |     |     |
| 位6 像素宽度位。0= 除显示方式 13H 外的所有方式<br>1= 方式 13H, VGA256 色                                                           |     |     |
| 位5 水平移动兼容性 (仅 VGA 用), 0= 行比较无效果<br>1= 行比较成功                                                                   |     |     |
| 位4 0 (保留)                                                                                                     |     |     |
| 位3 0= 允许背景增强 (文本方式) 或彩色 (图形方式)<br>1= 允许字符闪烁                                                                   |     |     |
| 位2 在 9 点点阵模式中 0= 第 9 点匹配背景 1= 第 9 点重复第 8 点<br>位1 0= 属性字节用彩色属性, 1= 用单色属性<br>位0 0= 属性控制器以文本方式处理数据, 1= 以图形方式处理数据 |     |     |
| R17 过扫描 (Overscan), 定义屏幕边框颜色 (文本方式)                                                                           |     |     |
| R18 彩色页面允许 (Color Plane Enable), 位0~3 对应页面 0~3, 当位值为 0 时不允许从输入状态寄存器中读取数据, 为 1 时允许                             |     |     |
| R19 水平像素平扫 (Horizontal pixel panning)                                                                         |     |     |
| R20 彩色选择 (只适用于 VGA)                                                                                           |     |     |
| 位7~位4 保留, 一般为 0                                                                                               |     |     |
| 位3、位2 作属性控制器到 VGA DAC 的 8 位视频数据的高位, 256 种颜色模式例外                                                               |     |     |
| 位1、位0 作调色板寄存器的输出 (见 R16)                                                                                      |     |     |

## 六 视频数模转换器 DAC (Video Digital to Analog Converter)

DAC 是 VGA 所特有的寄存器, 它能将彩色数字信号转换成模拟信号以驱动 VGA 的模拟显示器。VGA DAC 实际上包括 3 个 DAC (红、绿、蓝各一个), 它们分别控制红、绿、蓝三原色的亮度值 (RGB 值)。每一个 DAC 占 6 位 (3 个 DAC 构成一个 18 位颜色寄存器), 其亮度值范围为 0~63, 所以 3 个 DAC 总共可以配置 262144 (64 的 3 次方) 种颜色。DAC 必须从一个 8 位写地址寄存器 (口地址 3C8H) 中取得颜色寄存器的编号 (256 个颜色寄存器的编号为 0~255, 构成一个查色表, Color Look-up Table, 参见图 35-9)。这意味着

DAC 一次只能从查色表中转换 256(2 的 8 次方) 种颜色。

涉及软件编程的 VGA DAC 的寄存器有,读写状态寄存器(口地址 3C7H),读地址寄存器(口地址 3C7H)、写地址寄存器(3C8H)和数据寄存器(3C9H)。例如,当向 3C7H 端口写入一个颜色寄存器号(0~15 个调色板号可以和 16 个颜色寄存器号对应,对应关系依据显示卡。例如,对 VGA 卡在 VGAHI 模式下,图形初始化后 16 个调色板寄存器和 256 个颜色寄存器分别被置为系统缺省值。颜色寄存器号 0~6、20、56~63 和调色板号 0~15 对应。它有点和 graphics.h 中枚举 EGA—COLORS 相似。EGA 每个调色板寄存器中一位表示 6 根 EGA 输出线之一。属性控制器的 16 个调色板寄存器决定当前可的 16 种颜色。增强型彩色显示器可达到 64 种不同的颜色。对 VGA,在文本方式下,当系统的闪烁位被禁止时,背景色有 16 种;当闪烁位有效时背景色只有 8 种,0~7 调色板号对应颜色寄存器号为 0~5,20,7。),从 3C9H 就可以读出 3 个 6 位值即颜色号的 RGB 值(程序 DAC.C 给出了寄存器编程方法)。

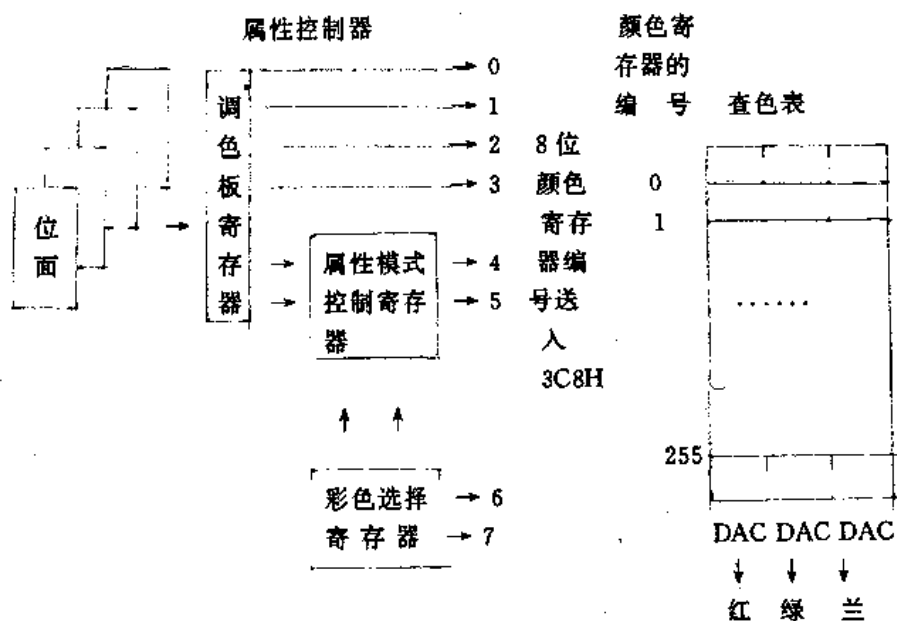


图 35-9 VGA 视频 RAM 读出象素与查色表关系示意

写颜色按读颜色相反的顺序进行。因为 VGA DAC 中的 256 个颜色寄存器不具有独立的 I/O 地址,所以要向某个颜色寄存器写入时,可把该颜色寄存器的编号先送到端口 3C8H 中,然后连续 3 次将颜色数据(依次为红、绿、兰)送入端口 3C9H。

设置或读取 DAC 的 RGB 值也可以调用 BIOS 中断 INT 10H 的子功能 10H、12H、15H 和 17H 等实现。它们不要求用户对图形硬件有更多的了解,安全方便,通常情况下宜用它编程。

```
C>TYPE DAC.C
#include <dos.h>
#include <stdio.h>
main() /* 检查颜色号对应的 RGB */
{
 union REGS r;
 int colurnum=5; /* 调色板寄存器号 */
 int DAC—Red,DAC—Green,DAC—Blue;
```

```

output(0x3c7,0); /* 位0 等于 0 表示进行读操作,等于 1 表示进行写操作 */
output(0x3c7,colornum);
DAC—Red=inportb(0x3c9);
DAC—Green=inportb(0x3c9);
DAC—Blue=inportb(0x3c9);
printf("颜色号=%d 红=%d 绿=%d 兰=%d\n",
 colornum,DAC—Red,DAC—Green,DAC—Blue);
}/* 程序输出:颜色号=5 红=42 绿=0 兰=42 */

```

### 35.3 视频缓冲区 (Video Buffer)

它是指在内存中有一块专门用来存放屏幕显示数据的 RAM。这一 RAM 块中的数据用户也可以象对内存中其它 RAM 那样读出 (如屏幕硬拷贝或将数据转储) 或写入 (要改变屏幕显示内容时)。

视频系统的显示电路会连续不断地读取视频缓冲区中的数据 (这些数据包括显示在屏幕上每一位置的颜色、亮度等), 并将数据按一定的规则在屏幕上显示出来。屏幕大约每秒 50 ~ 70 次的速度不断更新, 由于人眼的视觉滞留的原因, 所以当视频缓冲区中数据不变时, 人们感到屏幕上的数据几乎不变; 而当视频缓冲区中数据变更时, 屏幕显示会立即跟着变更。

对视频 RAM 的访问实际是对内存中这一数据区的访问 (DRAM 是它们的物理定位)。系统通过定时扫描这些数据而对屏幕进行刷新。对不同的视频系统, 其扫描方式、屏幕模式、可用存储容量、缓冲区的大小及在内存中的起始和终止 (段) 位置也不相同。

#### 1. MDA 的视频缓冲区

在 B000:0000 到 B000:FFFF 之间。这个缓冲区也可以在由 B100:0000、B200:0000、...、B700:0000 开始的 4K RAM 上。注意, MDA 的视频缓冲区是不能被关闭的 (disable)。

#### 2. HGC 的视频缓冲区

它占用了由 B000:0000 开始的 64K RAM。当它的组合开关寄存器的位 1 的值为 0 (缺省值) 时, 视频缓冲区只用到前 32K RAM (B000:0000 ~ B000:7FFF); 若要用后半段缓冲区 (B800:0000 开始, 与 CGA 的视频缓冲区重合), 则应将该位 1 置成 1。

#### 3. CGA 的视频缓冲区

它在 B800:0000 到 B800:3FFF 之间。它可以对映在 BC00:0000 到 BC00:3FFF 之间 (但很少用)。若与 MDA 一起使用, 其缓冲区地址不变。注意, 不同的 CGA 卡可能略有不同。

#### 4. EGA 的视频缓冲区

它根据图形控制器的 R6 的位 2 和位 3 的值对应到 4 种地址。这 2 位的值依设置的显示方式而自动确定。

表 35-9

| 位 3 的值 | 位 2 的值 | 视频缓冲区地址范围           |
|--------|--------|---------------------|
| 0      | 0      | A000:0000~B000:0000 |
| 0      | 1      | A000:0000~A000:FFFF |
| 1      | 0      | B000:0000~B000:7FFF |
| 1      | 1      | B800:0000~B800:7FFF |

当 EGA 的杂项输出 (Miscellaneous Output) 寄存器 (端口为 3C2H) 的位 1 为 0 时, 整个视频缓冲区就可设在 CPU 内存可寻址范围之外。

#### 5. MCGA 的视频缓冲区

它的 64K 缓冲区占用了从 A000:0000 到 A000:FFFF 的范围。但是其后半的 32K 视频 RAM (从 A800:0000 开始) 对映到 CGA 的视频 RAM 范围 (B800:0000 ~ B800:7FFF)。可由 INT 10H 中断的子功能

AH=12H BL=32H

来设置 CPU 是否能读写视频 RAM (参见 INT 10H 中断)。

#### 6. VGA 的视频缓冲区

它的定址控制基本与 EGA 相同, 但它还有一个 EGA 没有的视频子系统可用寄存器 (Video Subsystem Enable), 端口为 3C3H, 它控制视频缓冲区与 I/O 口定址。

INT 10H 的子功能 12H 提供了与 MCGA 相同的介面。

注意: VGA 的种类较多, 应注意了解它们的相同与不同之处。

### 35.4 屏幕显示方式

【显示方式】又称【显示模式】或【视频模式】, 它决定屏幕在文本模式下横向一行显示的列数 (80 列或 40 列)、在图形模式下显示的分辨率, 以及显示的类型 (彩色或黑白)。表 35-10 列出 IBM PC 的屏幕显示模式, 开机时是由 BIOS 决定的。对一个具体的显示器你可以用 BIOS 中断 INT 10H 的功能 0H 选择设置并测试其显示效果。

表 35-10

| 模式 | 类型 | 分辨率     | 列×行   | 字符   | 颜色数   | 页数 | 显示器类型    | 缓冲区首址 |
|----|----|---------|-------|------|-------|----|----------|-------|
| 0  | 文本 | 320×200 | 40×25 | 8×8  | 16 灰  | 8  | CGA      | B800H |
|    |    |         |       | 8×14 | 16 灰  | 8  | EGA      | B800H |
|    |    |         |       | 8×16 | 16    | 8  | MCGA     | B800H |
|    |    |         |       | 8×16 | 16    | 8  | VGA      | B800H |
| 1  | 文本 | 350×320 | 40×25 | 8×14 | 16/64 | 8  | CGA      | B800H |
|    |    |         |       | 8×14 | 16    | 8  | EGA      | B800H |
|    |    |         |       | 8×16 | 16    | 8  | MCGA     | B800H |
|    |    |         |       | 8×16 | 16    | 8  | VGA      | B800H |
| 2  | 文本 | 640×200 | 80×25 | 8×8  | 16 灰  | 8  | CGA      | B800H |
|    |    |         |       | 8×14 | 16 灰  | 8  | EGA      | B800H |
|    |    |         |       | 8×16 | 16    | 8  | MCGA     | B800H |
|    |    |         |       | 8×16 | 16    | 8  | VGA      | B800H |
| 3  | 文本 | 640×350 | 80×25 | 8×14 | 16/64 | 8  | CGA      | B800H |
|    |    |         |       | 8×14 | 16    | 8  | EGA      | B800H |
|    |    |         |       | 8×16 | 16    | 8  | MCGA     | B800H |
|    |    |         |       | 8×16 | 16    | 8  | VGA      | B800H |
| 4  | 图形 | 320×200 | 40×25 | 8×8  | 4     | 1  | CGA/EGA  | B800H |
|    |    |         |       |      |       |    | MCGA/VGA |       |
| 5  | 图形 | 320×200 | 40×25 | 8×8  | 4     | 1  | CGA/EGA  | B800H |
| 6  | 图形 | 640×200 | 80×25 | 8×14 | 2     | 1  | CGA/EGA  | B800H |



|    |                         |          |        |       |                  |                    |              |       |  |          |  |
|----|-------------------------|----------|--------|-------|------------------|--------------------|--------------|-------|--|----------|--|
|    |                         |          |        |       |                  |                    |              |       |  | MCGA/VGA |  |
| 7  | 文本                      | 720×350  | 80×25  | 9×14  | 1                | 8                  | MDA/EGA      | B000H |  |          |  |
|    | 图本                      | 3720×400 | 80×25  | 9×16  | 1                | 8                  | VGA          | B000H |  |          |  |
|    | 图形                      | 640×200  | 80×25  | 8×8   | 2                |                    | VGA          | B800H |  |          |  |
| 8  | 图形                      | 200×160  | 20×25  | 8×8   | 16               | 1                  | PCjr         |       |  |          |  |
| 9  | 图形                      | 320×200  | 40×25  | 8×8   | 16               | 1                  | PCjr         |       |  |          |  |
| A  | 图形                      | 640×200  | 80×25  | 8×8   | 4                | 1                  | PCjr         |       |  |          |  |
| B  | 保留(EGABIOS 功能 11H 会用到)  |          |        |       |                  |                    |              |       |  |          |  |
| C  | 保留(EGA BIOS 功能 11H 会用到) |          |        |       |                  |                    |              |       |  |          |  |
| D  | 图形                      | 320×200  | 40×25  | 8×8   | 16               | 2(64K)             | EGA/VGA      | A000H |  |          |  |
|    |                         |          |        |       |                  | 4(128K)            |              |       |  |          |  |
|    |                         |          |        |       |                  | 8(256K)            |              |       |  |          |  |
| E  | 图形                      | 640×200  | 80×25  | 8×8   | 16               | 1(64K)             | EGA/VGA      | A000H |  |          |  |
|    |                         |          |        |       |                  | 2(128K)            |              |       |  |          |  |
|    |                         |          |        |       |                  | 4(256K)            |              |       |  |          |  |
| F  | 图形                      | 640×350  | 80×25  | 8×14  | 4                | 1(64K)             | EGA/VGA/MCGA | A000H |  |          |  |
|    |                         |          |        |       |                  | 2(128K)            |              |       |  |          |  |
| 10 | 图形                      | 640×350  | 80×25  | 8×14x | 4/64(64K)1(128K) | 16/64(128K)2(256K) | EGA/VGA      | A000H |  |          |  |
| 11 | 图形                      | 640×480  | 80×30  | 8×16  | 2/256K           | 1                  | VGA/MCGA     | A000H |  |          |  |
| 12 | 图形                      | 640×480  | 80×30  | 8×16  | 16/256K          | 1                  | VGA          | A000H |  |          |  |
| 13 | 图形                      | 320×200  | 40×25  | 8×8   | 256/256K         | 1                  | VGA/MCGA     | A000H |  |          |  |
| 50 | 文本                      | 640×480  | 80×30  | 8×16  | 16               |                    | VGA          | B800H |  |          |  |
| 51 | 文本                      | 640×473  | 80×43  | 8×11  | 16               |                    | VGA          | B800H |  |          |  |
| 52 | 文本                      | 640×480  | 80×60  | 8×8   | 16               |                    | VGA          | B800H |  |          |  |
| 53 | 文本                      | 1056×350 | 132×25 | 8×14  | 16               |                    | VGA          | B800H |  |          |  |
| 54 | 文本                      | 1056×480 | 132×30 | 8×16  | 16               |                    | VGA          | B800H |  |          |  |
| 55 | 文本                      | 1056×473 | 132×43 | 8×11  | 16               |                    | VGA          | B800H |  |          |  |
| 56 | 文本                      | 1056×480 | 132×60 | 8×8   | 16               |                    | VGA          | B800H |  |          |  |
| 57 | 文本                      | 1188×350 | 132×25 | 9×14  | 16               |                    | VGA          | B800H |  |          |  |
| 58 | 文本                      | 1188×480 | 132×30 | 9×16  | 16               |                    | VGA          | B800H |  |          |  |
| 59 | 文本                      | 1188×473 | 132×43 | 9×11  | 16               |                    | VGA          | B800H |  |          |  |
| 5A | 文本                      | 1188×480 | 132×60 | 9×8   | 16               |                    | VGA          | B800H |  |          |  |

说明:1. 更详细的说明参考有关显示器和显示卡的用户手册。

2. 在图形方式下不显示光标。

显示器按显示的颜色可分为彩色或单色两种。数十种显示方式又可分为文本方式和图形方式两大类。在【文本方式】下,显示是按 ASCII 码字符为单位进行的,显示可以是 256 个 ASCII 码字符中的任何一个。组成字符的矩阵点数有多种。显示字符之间、上行与下行之间都留有不会显示的空点。在【图形方式】下,显示是按像素为单位进行的,显示的分辨率就是横向和纵向象点的个数。

在文本方式下对彩色显示器可以显示不同颜色的字符。在图形方式下,象点也可以有不同的颜色。

占用的显示内存方面,文本方式比图形方式要少得多,显示速度也快得多。

## 35.5 文本方式

### 35.5.1 屏幕的绝对坐标和窗口的相对坐标

可用坐标 (col,row) 表示屏幕上一个显示字符的位置,这里 col 是列数, row 是行数,屏幕左上角的位置是 (1,1),即 col=row=1。对 80×25 屏幕,col 的可选范围是 1~80,row 是 1~25。这就是【屏幕的绝对坐标】。

可用公式

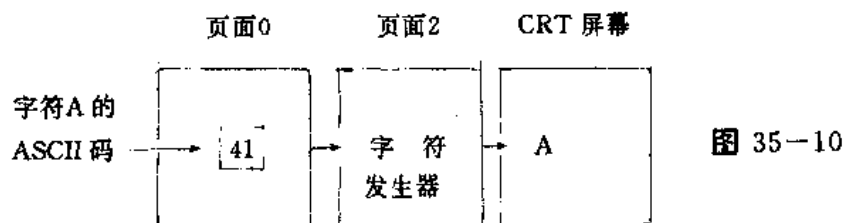
$$\text{偏移量} = (\text{row} - 1) * \text{width} + (\text{col} - 1) * 2$$

计算屏幕上某一个字符 (col,row) 在视频缓冲区中的相对地址, width 是每行可显示字符数。

可以在屏幕上用 window() 函数定义一个矩形区域,这就是【窗口】。窗口一经定义后,以后所有的坐标都是相对于窗口的,而不是相对于屏幕的。或者说,在窗口内的坐标 (col,row) 都是相对于窗口最左上角的字符位置而言的。假定窗口宽为 x,高为 y,则 col 不能超过 x, row 不能超过 y。这就是【相对坐标】。在文本方式,缺省的窗口(如果在程序中你没有定义窗口)是屏幕最大的窗口。定义窗口时用的坐标是屏幕的绝对坐标,而定义窗口后的操作是对窗口进行的。注意,在文本方式下,不要用图形方式下使用的函数,否则可能会有问题的。不过,反过来可以在图形方式下使用某些文本方式下使用的函数,但执行速度慢得多。

### 35.5.2 文本方式下的数据格式

在文本方式下,一个字符占用视频缓冲区中 2 个字节 (16 位) 表示,视频硬件的字符发生器再把这 2 字节数据转化成点阵并显示在屏幕上。对 MDA 和 CGA,字符发生器被装在 ROM 中;对 EGA 和 VGA 将字符发生器数据装入视频 RAM 的页面 2 中,如图 35-9 所示(页面参见本章 35.6 节)。



这 2 个字节的分工是,低位字节用字符的 ASCII 码表示,高位则用来表示字符的显示属性(或简称【属性】)。属性是指显示字符的颜色、亮度和闪烁等。每一个字符都可通过其高位字节值的变化而有不同的属性。

### 35.5.3 字符属性

1. MDA (HGC 模拟它,故两者无多大的区别)

(1)前景与背景属性(前景和背景可互换)

表 35-11

| 属性字节值 | 背景  | 前景  | 说明     |
|-------|-----|-----|--------|
| ?     | 0   | 7   | 前景正常亮度 |
| 0FH   | 0   | 0FH | 前景高亮度  |
| 70H   | 70H | 0   | 背景正常亮度 |
| F0H   | F0H | 0   | 背景高亮度  |

如果属性字节的位 7 等于 1, 且模式控制寄存器 (端口为 3B8H) 的位 5 也等于 1, 则字符闪烁。如果模式控制寄存器的位 5 等于 0, 则属性字节的位 7 代表背景的灰度, 而不代表闪烁。

(2) 下划线 (背景: 黑色)

表 35-12

| 前 景  | 前景属性 |
|------|------|
| 正常亮度 | 01   |
| 高亮度  | 09   |

注意: 对有些机器, 象 Compag, 由于其使用底线的属性 (前景属性) 用于显示灰度, 故无法再得到底线属性了。

## 2. CGA

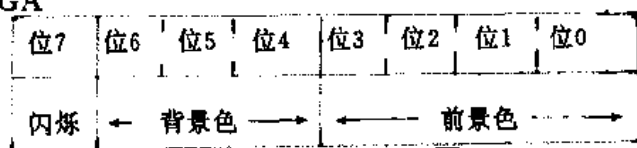


图 35-11

第 7 位: 为 1 时字符闪烁, 为 0 时不闪烁。为使字符闪烁, 只要将前景色加上 128 即可, 而这相当于把属性加上常量 BLINK (在枚举 COLORS 中)。

## 3. EGA

在 16 色文本方式下, EGA 采用和 CGA 相同的属性格式, 只是 4 位前景与背景值并不直接表示特定的颜色, 而是将这 4 个位值与属性控制器的寄存器 R18 做逻辑运算, 并用结果指定属性控制器的 16 个调色板寄存器 (R0 ~ R15) 中的一个, 再由这个指定的调色板寄存器的前 6 位产生 6 个 RGB 信号来驱动显示器。

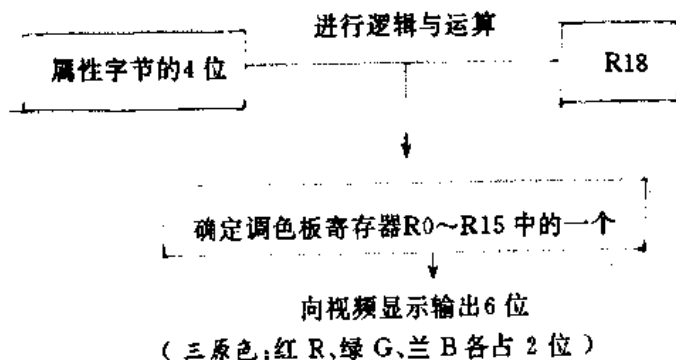


图 35-12

输出 6 位的情况参见图 35-13~15, 其中

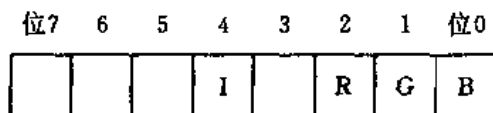
R、G、B——高亮度时三原色红、绿、兰

r、g、b——低亮度时三原色红、绿、兰

I——亮度控制

V——单色视频

① 350 行彩色时选中的调色板寄存器 (适用于 EGA)



对应 EGA 的 9 针连接头的针号                      6                      3      4      5

图 35-13

② 350 行单色时选中的调色板寄存器 (与 MDA 兼容)

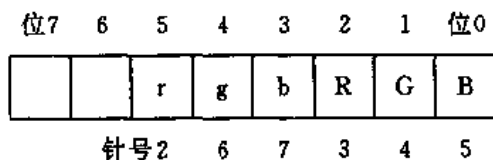


图 35-14

③ 单色视频时选中的调色板寄存器

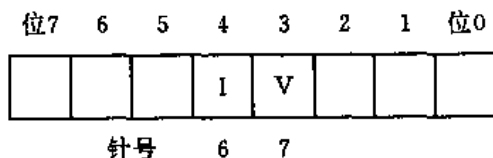


图 35-15

对 EGA 你可以先对每个字符选择前景与背景属性值, 再利用更改调色板的值来改变整个屏幕同一属性值的颜色。

EGA 的属性控制器的 R16 寄存器的位 3 是控制闪烁位。当其值为 1 时, 只有确定背景色的位 4~位 6 被用来指定最多 8 个调色板, 所以闪烁时背景只有 8 种颜色; 而当其值为 0 时, 虽然不闪烁, 但 16 个调色板都可被属性背景值指定。在选用文本方式时, BIOS 就会将调色板的值设定为与 CGA 相同的颜色。16 个调色板的后 8 个只是比前 8 个调色板颜色增亮。

在单色文本方式下, EGA 会模拟 MDA 的单色显示。视频 BIOS 将调色板的初值设为和 MDA 相同颜色属性值, 其位 3 决定象素是否显示, 位 4 决定是否要增亮。底线则在前景属性值为 1 或 9 时产生, 而不管调色板本身的值是多少。注意, 在 16 色文本方式下, 虽然前景为 1 或 9, 但只要背景是 0 或 8, 则将无法看到加底线的效果。

表 35-13

单色文本方式下EGA 的调色板寄存器值 属 性

|     |      |
|-----|------|
| 0   | 黑色   |
| 8   | 正常亮度 |
| 10H | 暗淡色  |
| 18H | 高亮度  |

## 4. MCGA

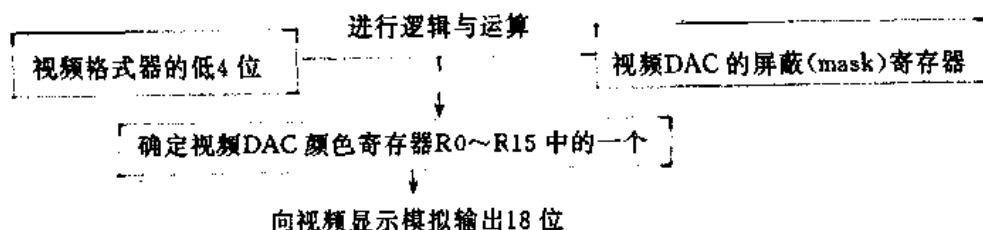


图 35-16 MCGA 颜色的产生 (正常情况下屏蔽寄存器的值为 0FFH, 端口为 3C6H)

对 DAC 的 256 个颜色寄存器, 在文本方式下, 只有前 16 个可选用 (其余的要在  $320 \times 200$  的 256 色图形方式下才能用)。当 MCGA 显示彩色时, BIOS 会将 DAC 前 16 个颜色寄存器的初值设为 CGA 颜色。每个颜色寄存器有 18 位, 红、绿、兰三原色各占用 6 位 (参见图 35-16)。

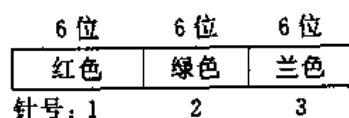


图 35-17

MCGA 的单色显示并不模拟 MDA 属性表示, 而是将 16 个 DAC 颜色寄存器分为 4 组:  $R0 \sim R3$ 、 $R4 \sim R7$ 、 $R8 \sim R11$ 、 $R12 \sim R15$ 。每组有 4 个不同的灰度, 且每一组灰度都比前一组高。设置方法使用 INT 10H 的子功能 10H。

## 5. VGA

一般情况下, VGA 模拟 EGA 处理属性, 但 VGA 除了 16 个属性控制器的调色板寄存器外, 还有一个 DAC。每一个调色板寄存器的值可选择 256 个 DAC 颜色寄存器中的一个, 这个颜色寄存器的值可以决定所要显示的颜色, 参见图 35-18。根据 R16 的值决定 VGA 颜色参见表 35-14。

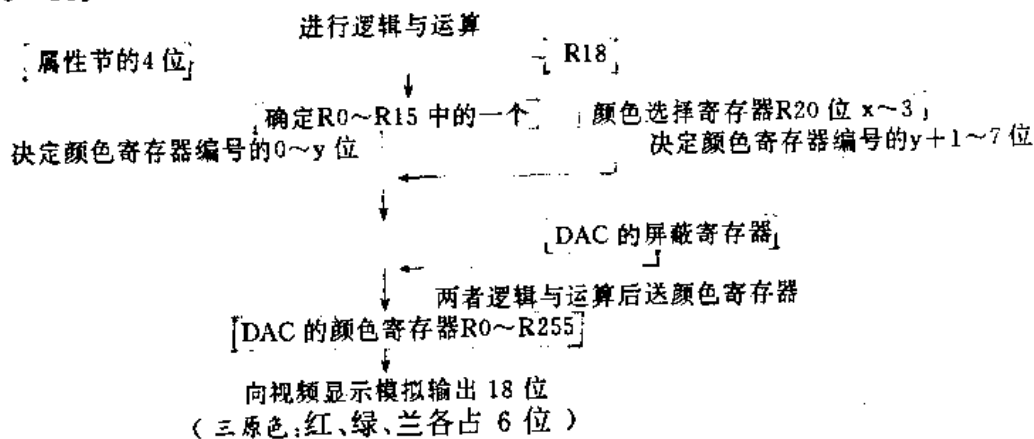


图 35-18

表 35 -14

属性控制器的模式寄存器 R16 的值 x y DAC 颜色寄存器分组数 每组个数

|   |   |   |    |    |
|---|---|---|----|----|
| 0 | 2 | 5 | 4  | 6  |
| 1 | 0 | 3 | 16 | 16 |

## 35.5.4 视频页

屏幕只是显示视频缓冲区的一部分,也就是说,屏幕是视频缓冲区的一个显示窗。

在  $80 \times 25$  的文本方式下,可以使缓冲区中  $80 \times 25 \times 2 = 4000$  个字节的内容在屏幕上显示出来。每一屏数据称为一页。一般视频缓冲区的字节按一定规则可以分成几页,而在同一时刻,只有一页数据可以显示,该页称为【显示页】或【可见页】。BIOS 视频数据区的 0040:0062 处有一个字节变量(值域为  $0 \sim 7$ )标示着当前显示页的页号。

## 1. CGA

CGA 有 16K 视频缓冲区,ROM BIOS 将 CGA 的视频缓冲区分页及每页起始地址为:

| 页号 | 每页 $80 \times 25$ 字符 | 每页 $40 \times 25$ 字符 |
|----|----------------------|----------------------|
| 0  | B800:0000            | B800:0000            |
| 1  | B800:1000            |                      |
| 2  | B800:2000            |                      |
| 3  | B800:3000            |                      |
| 4  |                      |                      |
| 5  |                      |                      |
| 6  |                      |                      |
| 7  |                      |                      |

怎样选定显示页?利用 INT 10H 功能 5 (AH=5)。

怎样显示缓冲区内的任何区域?只要将显示区域的起始地址的偏移量存入 CRTC 的寄存器 R12 和 R13,同时将 0040:004E 中的值改变。

## 2. EGA、MCGA 和 VGA

对文本方式,它们基本上与 CGA 类似,不同点在于对 R12 和 R13 寄存器不只是写入,而且可以读出检查。

## 35.5.5 光标形状

## 1. MCGA

在 MCGA 的 CRTC 中,将存放在光标起始光栅线和结束光栅线的值放大一倍,所以将光标的扫描线数放大了一倍,或者说将文本方式下的光标大小乘上了 2 倍。

在文本方式下,一个字形点阵的高为 16 点,因此扫描线是  $0 \sim 15$ 。如将起始光栅线设为 6,结束线设为 7,实际上扫描线是  $12 \sim 14$ 。存放光标起始光栅线和结束光栅线值的寄存器

只有低 4 位有效,即实际值是被限制在 0 ~ 7。如果超过 7,又会从 0 开始计算相对位置。

## 2. EGA 和 VGA

在 EGA 上,结束光栅线寄存器的值要比实际光标最低点的值大 1。如果结束线值比起始线值小,则结束线相当于从起始线到最大线,另加从 0 线到结束线 -1 的线。当结束线等于起始线时,光标就只有一行的高度。此时,结束线值应小于整个字型高度,否则将显示整个字型点阵,而不管起始线值如何。

VGA 的结束线寄存器就代表光标的最低行,从而不必象 EGA 那样要减 1;而且显示的光标一定是整个方块,而不会象 EGA 那样当结束线值小于起始线值时,光标会分成两块(光标中间部分不显示)。

对 EGA 和 VGA,结束线寄存器的位 5 或位 6 不为 0 时,则光标显示位置在 CRTC 的光标位置寄存器(R14 和 R15)所指位置 1 ~ 3 个字的地方。但在缺省情况下,这两位为 0。

## 3. 隐形光标

即将光标变成不可见。在 MDA、CGA 和 VGA 上,将 20H 放入起始光栅线寄存器内,就会产生隐形光标。在 EGA 上则将起始线值设为比字形点阵数还大,而把结束线值设为 0 即可。

### 35.5.6 文本方式下使用的库函数

TC 的文本模式函数适用于文本模式下工作。至于具体系统中可用的模式,取决于它的视频适配器的类型和所用的显示器。可以通过调用 `textmode()` 函数来决定当前的文本模式。

为方便叙述,我们将由屏幕绝对坐标构成的矩形区域简称为【矩域】,以便和窗口区分。显然,最大的矩域就是整个屏幕,矩域可以包括窗口,也可以交叉或重合。矩域的左上角坐标也是 (1,1)。

#### 一 定义显示模式和窗口

- 将屏幕设置为一个新文本模式 —1 `textmode()`
- 在屏幕上定义一个文本窗口 —2 `window()`

#### 二 属性控制

##### (一)颜色

- 设置新的字符背景颜色 —3 `textbackground()`
- 设置新的字符前景颜色 —4 `textcolor()`
- 设置字符属性(字符前景和背景颜色) —5 `textattr()`

##### (二)亮度

- 它将字符前景置成低亮度 —6 `normvideo()`
- 它将字符置成正常亮度 —7 `lowvideo()`
- 它将字符前景置成高亮度 —8 `highvideo()`

#### 三 状态查询

- 返回当前光标的 x(列)坐标 —9 `wherex()`
- 返回当前光标的 y(行)坐标 —10 `wherey()`
- 返回当前窗口的信息 —11 `gettextinfo()`

#### 四 光标及正文管理

- . 清除当前窗口内全部字符,光标移到窗口左上角 (1,1) —12 clrscr()
- . 将当前窗口内的光标移到指定坐标 (x,y) 处 —13 gotoxy()
- . 清除从光标位置到行末 (或光标右边) 的所有字符 —14 clrcol()
- . 在当前窗口内删除光标所在行 —15 delline()
- . 在当前窗口的光标处插入一空行 —16 insline()
- . 将屏幕上已定义的一个窗口位移到一个新位置 —17 movetext()

#### 五 正文块移动

- . 将由定义的矩域内的正文内容存入指定的内存区域内 —18 gettext()
- . 将指定的内存中的数据写到指定的矩域内显示 —19 puttext()

#### 六 文本读写(参见《文件管理》一章)

- . 从控制台得到字符并在屏幕上显示出来 —20 getche()
- . 将单个字符 c 送至屏幕窗口内 —21 putchar()
- . 将一字符串输出到屏幕窗口内 —22 cputs()
- . 它将格式化的输出送至屏幕窗口内 —23 printf()

—1 void —Cdecl textmode(int newmode);

将屏幕设置为新一个文本模式 newmode,该值可从由枚举 text—modes(在 conio.h 中)定义的常量选择(它受限于系统配置的显示器和适配器)。注意:使用符号常量必须包括标头文件 conio.h。

```
enum text—modes {LASTMODE=-1, /* 恢复上一次选择的文本方式。它只在 */
 /* 从图形方式返回到文本方式时才有用 */
 BW40=0, 黑白, 40 列 */
 C40, /* 即 C40=1, 彩色, 40 列 */
 BW80, /* 即 BW80=2, 黑白, 80 列 /
 C80, /* 即 C80=3, 彩色, 80 列 */
 MONO=7 /* 单色, 80 列 */
};
```

本函数将按指定模式将屏幕初始化成全屏文本窗口,并清除窗口中所有图形和正文。一般屏幕输出程序将缺省地设置一全屏的文本窗口,不用定义模式就可立即读、写及管理正文。当前窗口缺省置为全屏时,当前文本属性被调用函数 normvideo() 后的正常属性。程序输出也可以限制在屏幕某个区域(窗口)内,窗口可用 window() 函数建立。

—2 void —Cdecl window (int left, int top, int right, int bottom);

函数在屏幕上定义一个文本窗口。其中:

left 和 top 是窗口左上角在屏幕上的绝对坐标,缺省值为 left=1, top=1。  
right 和 bottom 是窗口右下角在屏幕上的绝对坐标,全屏时缺省值为  
对 80 列屏幕: right=80, bottom=25;  
对 40 列屏幕: right=40, bottom=25。

left 和 right 是列坐标,最小最大值为 1 和 40(或 80); top 和 bottom 是行坐标,最小最大值



为 1 和 25。left 应小于 right，top 应小于 bottom。否则，坐标无效。当坐标无效时，对函数调用无效，即不起作用（或称被忽略）。一旦该函数调用成功，所有定位坐标都是相对于刚刚定义的窗口的。

只适用于 IBM PC 及兼容机。

C>TYPE SCR1.C

```
#include <conio.h>
#define P(X) gettextinfo(&initial—info);\
/* 当前文本信息填入结构 initial—info 中 */
printf("X=%d winleft=%d\n",X,initial—info.winleft);\ /* 窗口 left */
printf("wintop=%d\n",initial—info.wintop);\ /* 窗口 top */
printf("winright=%d\n",initial—info.winright);\ /* 窗口 right */
printf("winbottom=%d\n",initial—info.winbottom);\ /* 窗口 bottom */
printf("attribute=%d\n",initial—info.attribute);\ /* 当前字符属性 */
printf("normattr=%d\n",initial—info.normattr);\ /* 屏幕字符属性 */
printf("currmode=%d\n",initial—info.currmode);\ /* 文本模式 */
/* 指 LASTMODE=-1,BW40=0,C40=1,BW80=2,C80=3,MONO=7 之一 */
printf("screenheight=%d\n",initial—info.screenheight);\
/* 初始化后屏幕总行数,或称屏幕高度 */
printf("screenwidth=%d\n",initial—info.screenwidth);\
/* 初始化后屏幕总列数,或称屏幕宽度 */
printf("curx=%d\n",initial—info.curx);\ /* 窗口内光标所在列 */
printf("cury=%d\n",initial—info.cury) /* 窗口内光标所在行 */
struct text—info initial—info; /* 结构 text—info 定义在 conio.h 中 */
main() /* 打印的信息为该结构的域中的信息 */
{ /* 其域都定义为 unsigned char 型 */
P(1);
window(2,3,60,20);
P(2);
textmode(initial—info.currmode);
P(3);
}
/* 程序输出整理后有,
X=1 winleft=1 X=2 winleft=2 X=3 winleft=1
wintop=1 wintop=3 wintop=1
winright=80 winright=60 winright=80
winbottom=25 winbottom=20 winbottom=25
attribute=7 attribute=7 attribute=7
normattr=7 normattr=7 normattr=7
currmode=3 currmode=3 currmode=3
screenheight=25 screenheight=25 screenheight=25
screenwidth=80 screenwidth=80 screenwidth=80
curx=1 curx=1 curx=2
cury=3 cury=1 cury=3 */
```

—3 void —Cdecl textbackground(int newcolor);

设置新的字符背景颜色。newcolor 只能从 conio.h 中定义的枚举

```
enum COLORS { BLACK,BLUE,GREEN,CYAN,RED,MAGENTA,BROWN,
 LIGHTGRAY,DARKGRAY,LIGHTBLUE,LIGHTGREEN,LIGHTCYAN,
 LIGHTRED,LIGHTMAGENTA,YELLOW,WHITE};
```

中取值 0 ~ 6 (BLACK ~ BROWN)。注意: newcolor 直接相当于字符属性字节中的位 6 ~ 位 4 的值, 故不需再移位。刚刚设置的新背景色在函数调用成功后起作用, 但它不会对当前已显示的字符的背景颜色产生影响。只影响 cprintf() 输出显示的字符。

只适用于 IBM PC 及与 BIOS 兼容的系统。

—4 void —Cdecl textcolor(int newcolor);

设置新的字符前景颜色, newcolor 可取值枚举 COLORS 中的所有常量 (0 ~ 15)。它只影响函数调用后所输出的字符颜色, 对已在屏幕上显示的字符不产生任何影响, 即不改变已显示字符的颜色。

注意: 有些显示器不能处理枚举 COLORS 后 8 种亮度颜色, 此时显示相当于“深度颜色”显示。不能显示彩色的视频系统可能将这些数字看成是一种颜色的灰度、特殊模式或特殊属性 (如下划线、黑体和斜体等)。总之, 对不同的视频系统要进行一点测试比较稳当。只适用于 IBM PC 及与 BIOS 兼容的系统。

—5 void —Cdecl textattr(int newattr);

设置字符属性函数, 利用此函数一次就可设置字符前景和背景颜色。注意: 使用枚举 COLORS 中的符号常量时应在源程序中包括标头文件 conio.h。

本函数只能对 cprintf() 函数起作用 (显示字符), 而对象 printf() 函数不起作用。值得指出的是, 本函数只影响 cprintf() 输出的字符颜色, 而屏幕上其它字符的颜色不会改变。正是利用这种特性, 有可能在一般菜单中比如对串

File

使第一个大写字母显示一种颜色, 而其余小写字母用另一种颜色。

只适用于 IBM PC 及与 BIOS 兼容系统。

函数参数 newattr 便是字符属性字节值。当背景值为 0 ~ 7 时, 可用将其左移 4 位的方法直接指定。例如带有常量值 WHITE + (RED << 4) 表达式非常易阅读, 一眼就可看出 WHITE 是字符前景, RED 是字符背景。显然, 利用这种方法设置背景比较方便, 免去了人为的对字节值的换算。

```
C>TYPE SCR2.C
#include "conio.h"
#define HEADERCOLOR WHITE+(RED<<4)
#define NOLIGHT WHITE+RED
#define HIGHT WHITE+(BLUE<<4)+BLINK
main(){
char *a="Please";
char *b="OK!";
char *c="hai";
char *d="1992.12";
textattr(HEADERCOLOR),cprintf(a);
textattr(NOLIGHT),cprintf(b,c);
textattr(HIGHT),cprintf(d);
textattr(YELLOW+(GREEN<<4)),cprintf("\n\n");
```

```
printf("\n%s%s%s%s\n",a,b,c,d);
}
```

—6 void —Cdecl lowvideo(void);

它将字符前景置成低亮度,相当于将控制字符属性字节中决定前景的最高位(位3)置成0,而对背景、闪烁等均置0。只适用于IBM PC及兼容机。

—7 void —Cdecl normvideo(void);

它将字符前景置成正常亮度,相当于将控制字符属性字节中决定前景的最高位(位3)置成0,而对背景、闪烁等不变。只适用于IBM PC及兼容机。

—8 void —Cdecl highvideo(void);

它将字符前景置成高亮度,相当于将控制字符属性字节中决定前景的最高位(位3)置成1,而对背景、闪烁等不变。只适用于IBM PC及兼容机。

单步调试程序SCR3.C便可以看到这几个函数引起的效果。

C>TYPE SCR3.C

```
#include "conio.h"
#define T(X,Y) textcolor(X),textbackground(Y)
#define CP(C,CH) putch(C),cprintf("#CH)
main(){
clrscr();
window(10,10,20,18);
T(WHITE+BLINK,RED);
cprintf("+- */");
T(MAGENTA,BLUE),cprintf("high"); /* 前景为低亮度 */
highvideo();
CP('A',a); /* 印出字符为高亮度 */
T(LIGHTCYAN+BLINK,WHITE),cprintf("norm"); /* 前景为高亮度,闪烁 */
normvideo();
CP('B',bb); /* 印出字符为正常亮度 */
T(LIGHTGREEN+BLINK,BROWN),cprintf("low"); /* 前景为高亮度,闪烁 */
lowvideo();
CP('C',ccc); /* 印出字符为低亮度 */
}
```

状态查询函数主要是查询当前文本窗口的信息,包括当前视频模式、在绝对屏幕坐标中窗口的位置、窗口大小、当前的前景及背景、光标的当前位置。

要求知道窗口内全部信息可用函数gettextinfo()。只想了解当前光标的位置,可用函数wherex()和wherey()。

—9 int —Cdecl wherex(void);

返回当前窗口下光标的x(列)坐标。如光标在当前窗口内,返回值范围为1~80(或40)。当光标不在当前窗口内,它返回相对于当前窗口左上角顶点的坐标,所以可能出现负值。只适用于IBM PC及兼容机。

—10 int —Cdecl wherey(void);

返回当前窗口下光标的y(行)坐标。如光标在当前窗口内,返回值范围为1~25。当光标不在当前窗口内,它返回相对于当前窗口左上角顶点的坐标,所以可能出现负值。只适用于IBM PC及兼容机。

—11 void —Cdecl gettextinfo(struct text—info \*r);

将有关当前窗口的信息填入 text—info 型结构指针 r 所指的区域中。函数本身不返回值。只适用于 IBM PC 及兼容机。结构 text—info 在 conio.h 中定义为

```
struct text—info {
 unsigned char winleft; /* 窗口左上角列坐标 */
 unsigned char wintop; /* 窗口左上角行坐标 */
 unsigned char winright; /* 窗口右下角列坐标 */
 unsigned char winbottom; /* 窗口右下角行坐标 */
 unsigned char attribute; /* 当前字符属性 */
 unsigned char normattr; /* 屏幕字符属性 */
 unsigned char currmode; /* 文本模式 */
 unsigned char screenheight; /* 屏幕高 */
 unsigned char screenwidth; /* 屏幕宽 */
 unsigned char curx; /* 光标所在列 */
 unsigned char cury; /* 光标所在行 */
};
```

程序 SCR4.C 是一个非常有趣的程序。当你在集成环境下单步调试时很容易弄清这样一些事实：

1. textmode(3) 只管设置文本模式 (16 色, 80 × 25), 而对屏幕置成象调用 normvideo() 函数后的正常属性；

2. normvideo() 只管将闪烁变成不闪烁, 即使字符显示正常亮度。对字符的前景和背景未管；

3. clrscr() 将定义窗口内字符全部消去, 从而原窗口内的字符将不再存在。另一方面, 如果在它前面有设置前景或背景的语句, 则窗口立即使设置的颜色起作用。这有点象 cprintf() 函数, 但 cprintf() 只使要显字符的窗口部分变色, 而 clrscr() 使整个窗口变色；

4. 单步调试到 delay(5000); 语句时, 屏幕自动转向用户屏幕, 这时尽管你并没有按象 Alt+F5 那样的键。如果你只按 F7(或 F8) 键, 而延迟时未按其它键, 则 (if(kbhit())) 后的语句不被执行。反之, 将执行。而使屏幕恢复到正常的状态：黑底白字而不闪烁。

5. 多次连续单步调试, 你可以看到两种不同颜色的用户屏幕。结束调试时光标处的字符属性被保留 (但闪烁特性不保留), 作为下次调试时的 normattr() 属性；

6. 在 DOS 状态, 使用 CLS 命令总能使屏幕恢复到正常字符属性状态：黑底白字而不闪烁。

C>TYPE SCR4.C

```
#include "conio.h"
#define P1(X1,Y1,X2,Y2,X) window(X1,Y1,X2,Y2),textattr(X)
#define P2(Z) gettextinfo(&a);\
 gotoxy(1,Z);printf("a.normattr=0x%x",a.normattr);\
 gotoxy(1,Z+1);printf("a.attribute=0x%x",a.attribute)
#define P3(Y1,Y) gotoxy(1,Y1);cprintf("#Y");printf("#Y");\
 gotoxy(1,Y1+1);cprintf("norm=0x%x",a.normattr);\
 gotoxy(1,Y1+2);cprintf("attr=0x%x",a.attribute)

main(){
 struct text—info a;
```

```

int x,y;
textmode(3);
P2(1);P3(3,OK! OK! OK! OK! OK!);
window(1,1,80,25);
clrscr();
P2(1);P3(3,+++++);
P1(20,5,50,15,WHITE+(RED<<4)+BLINK);
P2(1);P3(3,-----);
textattr(GREEN+(LIGHTGRAY<<4)+BLINK);
P2(6);
P3(8,* * * * *);
P1(30,16,60,25,YELLOW+(LIGHTGRAY<<4)+BLINK);
normvideo();
P2(1);
P3(3,/////);
P1(40,1,60,15,YELLOW+(LIGHTGRAY<<4)+BLINK);
clrscr();
P2(1);
P3(3,?????);
delay(5000);
getch();
if(kbhit())
{
 P1(60,16,80,25,LIGHTGRAY+(BLACK<<4));
 clrscr();
 P2(1);
 P3(3,?????);
}
}

```

程序 SCR5.C 对上述结论作了进一步说明。

```

C>TYPE SCR5.C
#include "conio.h"
#define P1 gettextinfo(&a);x=wherex();y=wherey();\
gotoxy(1,2);cprintf("a.normattr=0x%x",a.normattr);\
gotoxy(1,3);cprintf("a.attribute=0x%x",a.attribute);\
gotoxy(1,4);cprintf("a.curx=%u",a.curx);\
gotoxy(1,5);cprintf("a.cury=%u",a.cury);\
gotoxy(1,6);cprintf("x=%d y=%d",x,y);
/* 可以看到 a.curx 和 wherex() 值相等,a.cury 和 wherey() 值相等 */
main()
{
 struct text—info a;
 int x,y;
 clrscr();
 textattr(WHITE+(RED<<4)+BLINK);
 P1; /* 输出为红底白字,闪烁 */
}

```

```

textmode(3); /* 重设置模式后,若不考虑最高位,a.attribute=a.normattr, */
P1; /* 即前景和背景为正常状态,字符不再闪烁 */
window(20,10,50,20);
textattr(GREEN+(WHITE<<4)+BLINK); /* 背景选了后 8 种之一 */
gotoxy(18,4); /* 结果虽用了 BLINK, */
P1; /* 但输出字符不闪烁 */
a.attribute=a.normattr;
window(4,18,30,25);/* 由于 a.attribute=a.normattr 在 gettextinfo() 之前 */
clrscr(); /* 所以此句不起作用,因为 gettextinfo() 是取得视频 */
gotoxy(5,5); /* 缓冲区的值,而前赋值语句并不改变缓冲区的值。而 */
P1; /* 当执行 gettextinfo() 时 a.attribute 将被重新赋值。 */
window(40,18,70,25);
textattr(LIGHTGREEN+(LIGHTGRAY<<4)+BLINK);
clrscr(); /* 如在集成环境下连续多次单步调试,则当重新开始执行 */
gotoxy(5,5); /* 程序时,a.normattr 值被改变。而在 DOS 状态下可不变。 */
P1;
}

```

—12 void —Cdecl clrscr(void);

清除当前文本窗口内全部字符,并将光标移到窗口左上角,坐标位置为 (1,1)。只适用于 IBM PC 及兼容机。

—13 void —Cdecl gotoxy(int x, int y);

将当前窗口内的光标移到指定坐标 (x,y) (指当前窗口的列和行) 处。如果坐标超出窗口坐标范围,则 TC 不能实现,因而将它忽略 (即调用此函数未起作用)。当前光标位置可用函数 wherex() 和 wherey() 查询。只适用于 IBM PC 及兼容机。

—14 void —Cdecl clreol(void);

清除从光标位置到行末 (或光标右边) 的所有字符,光标本身位置未动。

—15 void —Cdecl delline(void);

在当前活动窗口内删除光标所在行 (窗口外的内容不受影响),即该行的全部字符不复存在。并把光标所在行的下面各行都上移一行。只适用于 IBM PC 及兼容机。

—16 void —Cdecl insline(void);

在当前窗口的光标处插入一空行 (窗口外的内容不受影响),原光标下面各行都向下移一行,最底下一行滚出窗口底部。只适用于 IBM PC 及兼容机。

—17 int —Cdecl movetext(int left,int top, int right, int bottom,  
int destleft, int desttop);

将屏幕上由 left、top、right 和 bottom 定义的矩域位移到屏幕上一个新位置,新位置左上角在屏幕上的坐标是 (destleft,destop) (即这六个坐标值是屏幕的绝对坐标)。例如,对彩色文本屏幕为 25 × 80 时,其有效坐标为 (1,1) ~ (80,25)。如果坐标给出正确,移动成功,返回值 1; 否则只要有一个坐标错误 (例如越界) 便返回 0。还应指出,移动亦包括字符颜色,即不只搬移指定窗口内字符的 ASCII 码值,搬移到新坐标处的字符颜色由其在原屏幕处的颜色决定。

只适用于 IBM PC 及与 BIOS 兼容系统。

程序 SCR6.C 是说明上述函数的一个有趣的综合演示程序。在彩色文本方式,你可以看到程序一步步产生的效果,包括显示字符越出窗口时自动换行到窗口的下行开始处等。

```

C>TYPE SCR6.C
#include "conio.h"
#define UP 2
#define DOWN 3
#define LEFT 0
#define RIGHT 1
#define DE delay(2500) /* 延迟时间,以便观察 */
#define PP(X) cprintf(#X),DE /* 向窗口输出,超出边界自动换行 */
#define GG gotoxy(inx,iny) /* 窗口内光标移到新位置 */
void scroll(int, int, int, int, int, int, int);
int _r;
main(){
 textmode(3); /* 置 C80 彩色文本模式 */
 clrscr(); /* 清屏 */
 window(1, 1, 80, 25); /* 窗口为全屏幕 */
 scroll(UP,0,1,1,30,1,WHITE+(RED<<4)),PP(444444444444444444444444AABCCDD);
 scroll(DOWN,4,1,6,30,9,GREEN+(MAGENTA<<4)),PP(333333333333333333333333);
 gotoxy(10,1),clrscr(),DE;
 /* 光标移到窗口内第 1 行第 10 列并删去光标右边字符 */
 scroll(UP,1,1,1,8,5,RED+(WHITE<<4)),PP(222222222222222222222222);
 scroll(LEFT,2,1,10,30,15,GREEN+(RED<<4)),PP(000000000000000000000000);
 gotoxy(2,2),DE; /* 光标移到窗口内第 2 行第 2 列后删去第 2 行 */
 scroll(RIGHT,3,1,17,30,23,(BLUE+RED<<4)+BLINK);
 PP(1111111111111111111111111111);
 gotoxy(1,2),insline(); /* 光标移到窗口内第 2 行第 1 列后在光标处增加一行 */
 window(1,1,80,25); /* 将当前窗口开成全屏幕 */
 printf("\n-r=%d demonstration --- end !",_r); /* 演示结束 */
}

void scroll(int direction, int lines, int x1, int y1, int x2, int y2,
 int attrib)
{
 int inx=x1+3,iny=y1+1;
 if (lines == 0)
 {
 window(x1, y1, x2, y2+1),PP(ABCDEFGHJKLMNOPQRSTUVWXYZabcdefghij),DE;
 _r=movetext(1, 1, 40, 2, 32, 2),DE; /* 观察 movetext() 返回值 */
 }
 /* 及窗口移动情况 */
 else switch(direction)
 {
 case UP :
 movetext(x1, y1 + lines, x2, y2, x1, y1),DE;
 window(x1, y2 - lines + 1, x2, y2),GG;
 break;
 case DOWN :
 movetext(x1, y1, x2, y2 - lines, x1, y1 + lines),DE;
 window(x1, y1, x2, y1 + lines - 1),GG;
 break;
 case LEFT :

```

```

 movetext(x1 + lines, y1, x2, y2, x1, y1);DE;
 window(x2 - lines + 1, y1, x2, y2);GG;
 break;
case RIGHT:
 movetext(x1, y1, x2 - lines, y2, x1 + lines, y1);DE;
 window(x1, y1, x1 + lines - 1, y2);GG;
 break;
}
textattr(attrib);
}

```

—18 int —Cdecl gettext(int left, int top, int right, int bottom,  
void \*destin);

将由 left、top、right 和 bottom(屏幕绝对坐标)定义的矩域内的内容存入由指针 destin 指定的内存区域内。矩域中的内容在内存中存放的方式是,从左到右、从上到下顺序读入。屏幕上的每个字符要占两个字节的内存单元,第一个字节用于存放字符的 ASCII 码,第二字节存储字符的显示属性(颜色、闪烁等)。一个  $W \times H$ (列数 $\times$ 行数)的矩域需用  $2 * W * H$  字节。

如果操作成功,函数返回值 1,否则返回 0。只适用于 IBM PC 及与 BIOS 兼容的系统。

—19 int —Cdecl puttext(int left, int top, intright, int bottom,  
void \*source);

本函数是 gettext() 函数的反函数,它将由 source 指针指定的内存中的数据写到由 left、top、right 和 bottom(屏幕绝对坐标)定义的矩域内,并被显示。其余说明同 gettext() 函数。

程序 SCR7.C 是一个说明怎样使用这两个函数的演示程序。注意:使用这两个函数最容易出错的是将范围搞错(超出屏幕最大许可范围!),结果产生不可预料的结果。为简单起见,程序中 puttext() 只是将原窗口还原,实际上它可以使它送到屏幕上任何位置。例如,你把程序中涉及这两个函数的语句改成

```

x=gettext(1,1,80,25,ch—buf);
y=puttext(76,24,80,25,ch—buf);

```

也是可行的,只是越出屏幕的部分将不可见而已。只适用于 IBM PC 及与 BIOS 兼容系统。

```

C>TYPE SCR7.C
#include "conio.h"
#define A(XY) —x[XY]=wherex();—y[XY]=wherey() /* 查询当前光标位置 */
main(){
 int x,y,j;
 int —x[15],—y[15];
 /* 记录 0~14 个光标位置.注意,不能定义成 —x[14],—y[14] */
 static ch—buf[2*40*25]; /* 设置内存缓冲区 */
 textmode(3); /* 设置文本模式 3: 彩色 80×25 */
 window(1,1,80,25); /* 窗口为全屏 */
 A(0); /* 当前光标的坐标: 1,1 */
 clrscr(); /* 清当前文本窗口 */
}

```



```

A(1); /* 1,1 */
textattr(WHITE+(RED<<4)+BLINK); /* 设置字符前景和背景 */
cprintf("==+==*/=="), /* 必须用 cprintf() 才能使颜色设置起作用 */
A(2); /* 11,1 */
window(1, 1, 4, 4); /* 设置新窗口 */
A(3); /* 1,1 */
clrscr();
A(4); /* 1,1 */
printf("AABCCDDEEFFGGHHIIJJKK\n111222333\
444555666777888999000\n====="); /* 注意含有数字的字符串的续行方法: 斜杠紧挨 */
/* 着其前面的数字,下一行必须从第一列开始,即前面无空格 */
/* 注意,字符串续行时必须在行尾有续行符(\). */

```

```

A(5); /* 7,3 */
x=gettext(1,1,40,25,ch-buf); /* 变量 x 记下函数返回状态 */
printf("%x\n",x);
A(6); /* 1,4 */
clrscr();
A(7); /* 1,1 */
window(10,8,50,20);
A(8); /* 1,1 */
textattr(GREEN+(LIGHTGRAY<<4));cprintf(" * ");
gotoxy(2,2);
A(9); /* 2,2 */
printf("aaabbbccddddd\n");

```

/\* 函数 wherex() 和 wherey() 计算坐标位置的方法是:

wherex() 返回 x (当前列) 坐标 = 最近用 window() 设置的窗口的 第一列的屏幕绝对 x 坐标 - 光标现在的实际屏幕绝对 x 坐标 + 1 wherey() 返回 y (当前行) 坐标 = 最近用 window() 设置的窗口的 第一行的屏幕绝对 y 坐标 - 光标现在的实际屏幕绝对 y 坐标 + 1 因而出现负值! \*/

```

A(10); /* -8,3 */
y=puttext(1,1,40,25,ch-buf); /* 变量 y 记下函数返回状态 */
A(11); /* -8,3 */
textattr(WHITE+(LIGHTGRAY<<4));
printf("\n\n");
A(12); /* -8,5 */
cprintf(" ");
A(13); /* -1,5 */
printf("\nx=%d y=%d\n",x,y);
A(14); /* -8,7 */
for(J=0;J<15;J++)printf("J=%d(%d,%d) ",J,-x[J],-y[J]);
printf("\n");
}

```

/\* printf() 函数可能会使字符越出已定义的窗口。在单步调试时可以使用调试表达式: -x[0],15 和 -y[0],15 观察光标位置并随时将观察结果记录到源程序中。为节省时间,每有 Source modified, rebuild (Y/N) 显示时,可按 N 键响应。看输出屏幕按 Alt-F5 键 \*/

-20 int -Cdecl getche(void);

从控制台（通常指键盘）得一字符，并在屏幕上显示出来。只适用于 MS-DOS。（参见《键盘与鼠标》一章）。

—21 int —Cdecl putchar(int c);

将单个字符 c 送至屏幕窗口内。字符使用当前颜色和显示属性。返回字符 c。（参见《文件管理》一章）。

—22 int —Cdecl cputs(const char \*str);

将一字符串（可带颜色属性）输出到屏幕窗口内。该函数的输出不能改变方向，并且它会自动防止字符输出到当前窗口的外面。调用成功，返回最后一个送到屏幕上的字符，否则返回 EOF。

```
C>TYPE SCR8.C
#include "conio.h"
main(){
 int i,len;
 char ch;
 char *s="Press A key,Please!";
 len=strlen(s); /* 获串长度 */
 cputs(s); /* 显示串 Press A key,Please! */
 ch=getche(); /* 等待按键,ch 接收键值 */
 printf("\nch=%c\n",ch);
 if(ch=='A')
 for(i=0;i<len;i++,s++)putch(*s); /* 逐个显示串字符 */
 putch('\r\n'); /* 显示光标到下行头部 */
}
/* 如果在提示时键入字母 A 则显示
Press A key,Please!A
ch=A
Press A key,Please!
否则击其它键会显示
Press A key,Please!D
ch=D */
```

—23 int —Cdecl cprintf(const char \*format, ...);

它将格式化的输出送至屏幕窗口内（对字符串可带颜色属性）。返回输出字节数（参见《格式输入与输出函数》一章）。

由于它只输出到屏幕，故在 DOS 状态下输出时使用重定向（如输出到指定文件中）无效。凡指出输出只送至屏幕的都有此特性。

```
C>TYPE SCR9.C
#include "conio.h"
#define HEADERCOLOR WHITE+(RED<<4)
#define NOLIGHT WHITE+RED
#define HIGHT WHITE+(BLUE<<4)+BLINK
main(){
 int i,num[5];
 char *a[]={"%d=Please","%d=OK!","%d=hai","%d=1992.12"};
```

```

for(i=0;i<4;i++)
{
 switch(i)
 {
 /* 改变显示字符颜色 */
 case 0;textattr(HEADERCOLOR);break;
 case 1;textattr(NOLIGHT);break;
 case 2;textattr(HIGHT);break;
 default;
 textattr(YELLOW+(GREEN<<4));break;
 }
 num[i]=cprintf(a[i],i);
}
cprintf(""); /* 此句如改成 cprintf("\n"); 输出效果略有不同 */
printf("\n%s%s%s%s\n",a[0],a[1],a[2],a[3]);
for(i=0;i<4;i++)printf("%d=%d ",i,num[i]);
printf("\n");
}
/* 输出: 0=Please1=OK!2=hai3=1992.12 cprintf 输出,每串有一种颜色
 %d=Please%d=OK!%d=hai%d=1992.12 printf 输出,一种颜色
 0=8 1=5 2=5 3=9 cprintf() 返回值输出
*/

```

## 35.6 图形方式

在图形方式下,可以控制屏幕上显示的每一点,因此也称为 APA (All Points Addressable) 方式。

在图形方式下,也可在屏幕上定义一个矩形区域,称为【视口】。当程序输出图形时,视口就是实际的屏幕。视口可通过 graphics.h 中的 setviewport() 函数定义。

【视口坐标】是指视口内的坐标,其原点也在左上角,坐标值为 (0,0)。

在通常情况下图形光标是不可见的,它的当前位置(用坐标表示)简称【CP】(Cursor Position)。

### 35.6.1 像素和字节的关系

显示系统中常用一个字节的一组(一个或一个以上)位来表示屏幕上的像素(点),每个像素点的颜色都直接或间接由这组位值决定,每个字节可以表示一个或多个像素。每一象点由几位决定,这和视频 RAM 的结构相关。

#### 1. CGA

##### (1)象点所用位数

|            |              |              |
|------------|--------------|--------------|
|            | 640×200 双色方式 | 320×200 四色方式 |
| 表示一个象点所用位数 | 1 位          | 2 位          |

##### (2)扫描线的分布情况

CGA 图形方式采用隔场扫描方式,它将 16K 视频缓冲区分成二个区,屏幕显示的扫描线共 200 条,依次是扫描线 0,1,2,...,其中 100 条偶数扫描线(0,2,4,...)对应视频缓冲区从 B800:0000 开始的数据,而奇数扫描线(1,3,5,...)则对应从 B800:2000 开始的数据。

## 2. HGC

在  $720 \times 348$  图形方式下,每一象点用一个二进制位表示,故只能表示亮与不亮两种“颜色”。它的 348 条长度为  $(720/8=)90$  个字节的扫描线,是由四块视频缓冲区的分区中的数据构成(四场扫描方式):

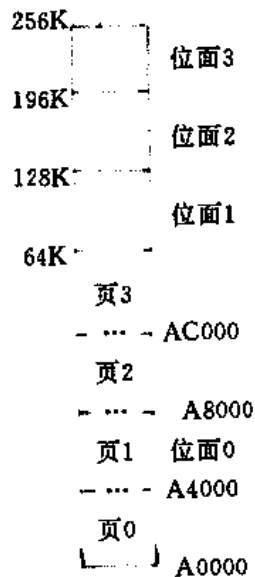
| 分区号 | 分区起始地址    | 涉及扫描线      |
|-----|-----------|------------|
| 0   | B000,0000 | 0,4,8,...  |
| 1   | B000,2000 | 1,5,9,...  |
| 2   | B000,4000 | 2,6,10,... |
| 3   | B000,6000 | 3,7,11,... |

每一分区包括  $87(=348/4)$  条扫描线。可见它的扫描线也是象 CGA 那样交叉访问的。

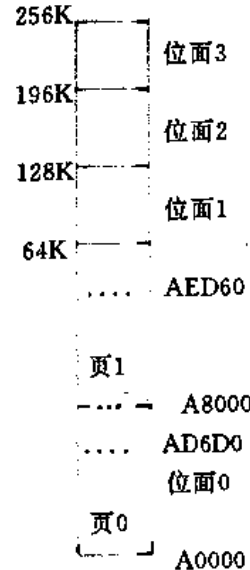
## 3. EGA

EGA 和 VGA 的图形方式采用连续扫描方式。例如,在显示方式 10H 或 12H 下,其象点与缓冲区的映象关系为,随着存储单元地址的递增,屏幕上象点从左到右、自上向下与之相对应,缓冲区中每一位代表一个象点。

EGA 将 256K 视频缓冲区分割成四块平行的 64K 区域,每一个区称为一个【位面】或【位面】(bit planes),或【页面】。每个页面的组成方式与彩色卡中单色高分辨模式相同:当一字节数据被送入视频缓冲区的某一地址时,每位都与屏幕上一个象点对应,这些象点的排列成一个水平线段,位 7 对应于最左象点。



(a) EGA 和 VGA 640X200 16色  
图形卡为每个位面提供16384  
字节,实际每个位面只用  
16000 字节



(b) EGA 和 VGA 640X350 16色,  
每个位面要28000字节  
4个位面要 112000 字节



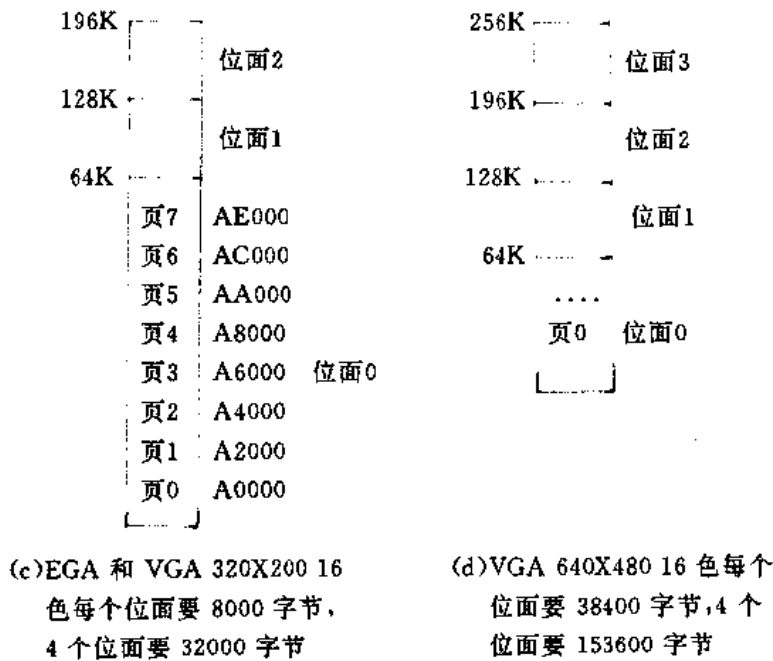


图 35-19 显示内存映象 (Memory map)

在图形方式下,每一象点对应这四块区域中相同字节偏移量和相同位偏移量的数据。EGA 的时序控制寄存器和图形控制寄存器可分别或同时读取这 4 个位面的数据。如图 35-19 所示,在 16 色彩色图形方式下,各位面一字节值形成屏幕上 8 个象点的过程(由四个位面生成一个四位二进制数,其各位从高到低依次是位面 3 的位、位面 2 的位、位面 1 的位和位面 0 的位)。

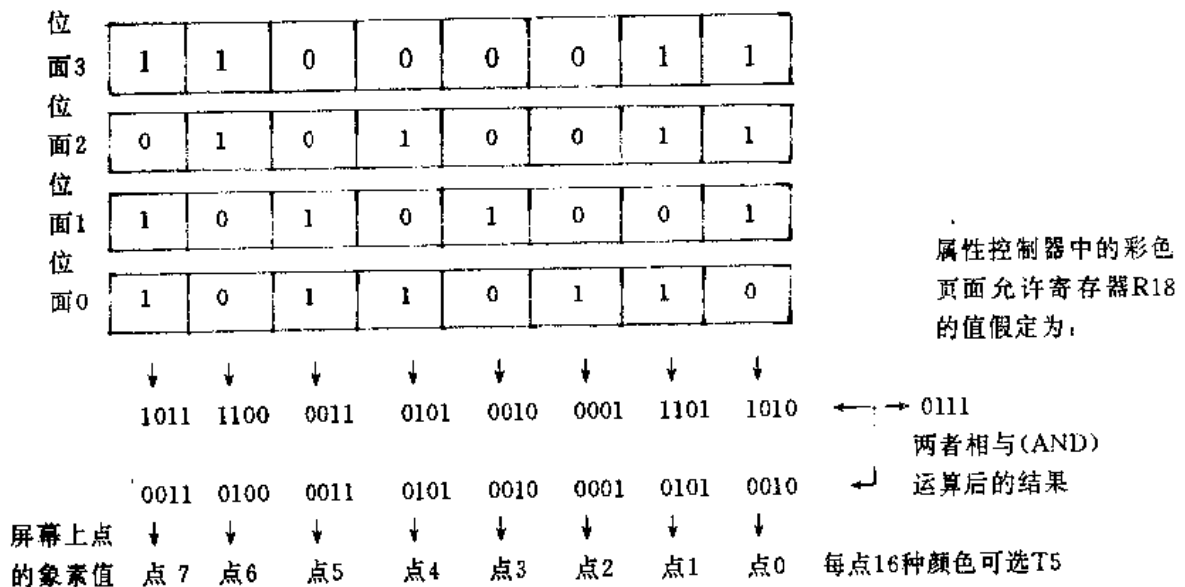


图 35-20 EGA 图形方式下像素对应关系

在 EGA 的 350 行图形方式下,占用了 64K 视频 RAM,在选择奇数点位置时,会使用奇数位面(位面 3 和 1)的位;使用偶数位置点时,会用偶数位面(位面 2 和 0)的位。但不管哪一种情况,色平面选择寄存器(R18)只有位 2 和位 0 参与“与”运算。

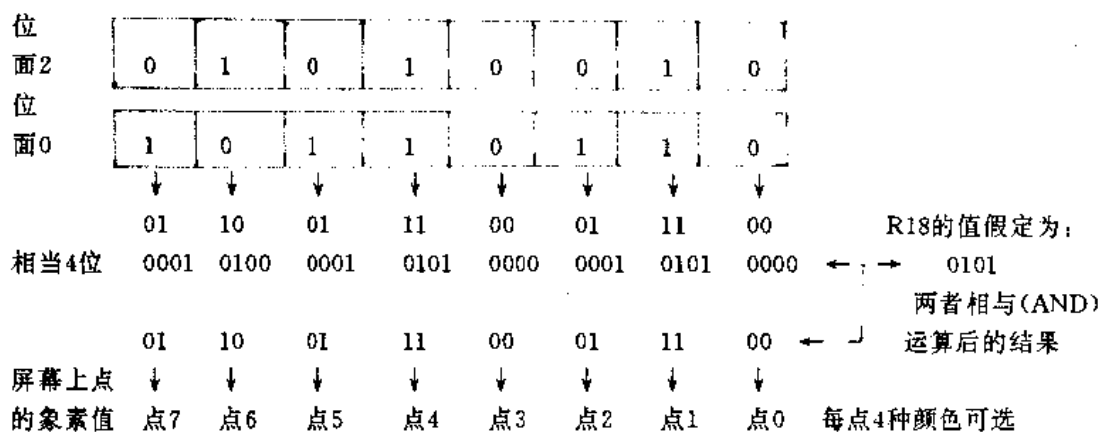


图 35-21 在 350 图形方式下偶数点位置存放

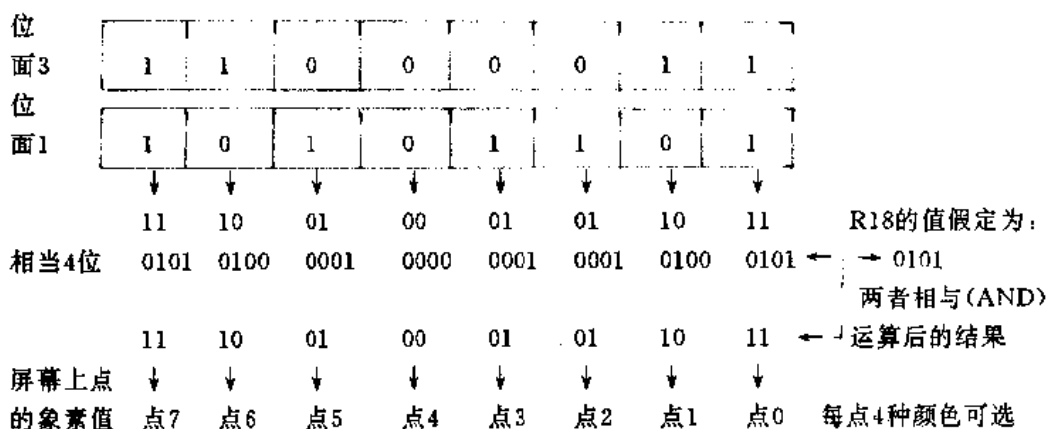


图 35-22 在 350 图形方式下奇数点位置存放

EGA 的象点位置不象 CGA 那样与视频缓冲区中摆放的数据是奇偶关系（或象 HGC 是 4 倍的关系），而是和文本方式下相同，象点位置与视频缓冲区中的位置都有相同的次序。假定要在程序中直接读写视频缓冲区中的象素，就必须知道这种位平面和视频缓冲区中数据的这种关系。

#### 4. EGA 和 VGA 读写方式的确定

图形控制器的图形方式寄存器（R5）的位包含了两部分值：位 3 表示读方式。例如，该位置 0 表示读方式 0，为 1 表示读方式 1；位 1 和位 0 则表示写方式号（0～3）。

在时序控制器中的位面屏蔽寄存器（R2）在 16 色图形方式下，这个寄存器的位 3～位 0 通常设成 1，此时允许把数据传到视频缓冲区位面上。它的每一位对应一个位面，如果某位为 0，则数据不会写到对应位面上。注意，很少使用这一寄存器。

#### 一 读方式

##### （一）读方式 0

直接将某个位面中的一字节值读入 CPU 中。选择哪个位面由图形控制器的读位图选择寄存器 R4 的低 2 位（位 1 和位 0）决定。每次只能读一个位面。

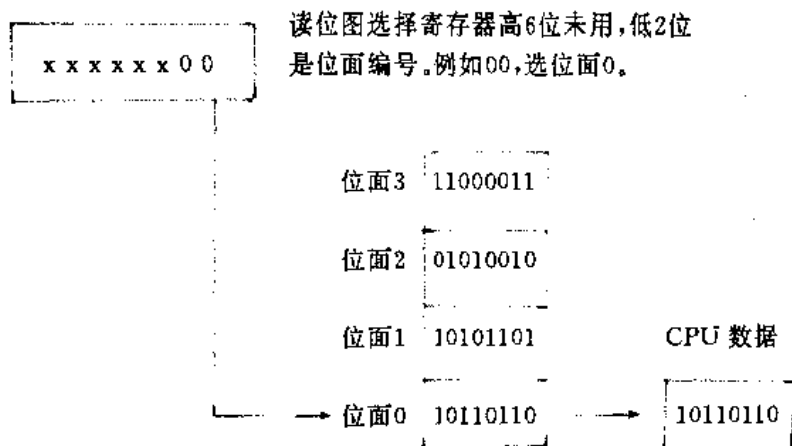


图 35-23 EGA 和 VGA 的读方式 0

## (二) 读方式 1

CPU 的读动作使位面值存入锁存寄存器,经过处理后送 CPU。

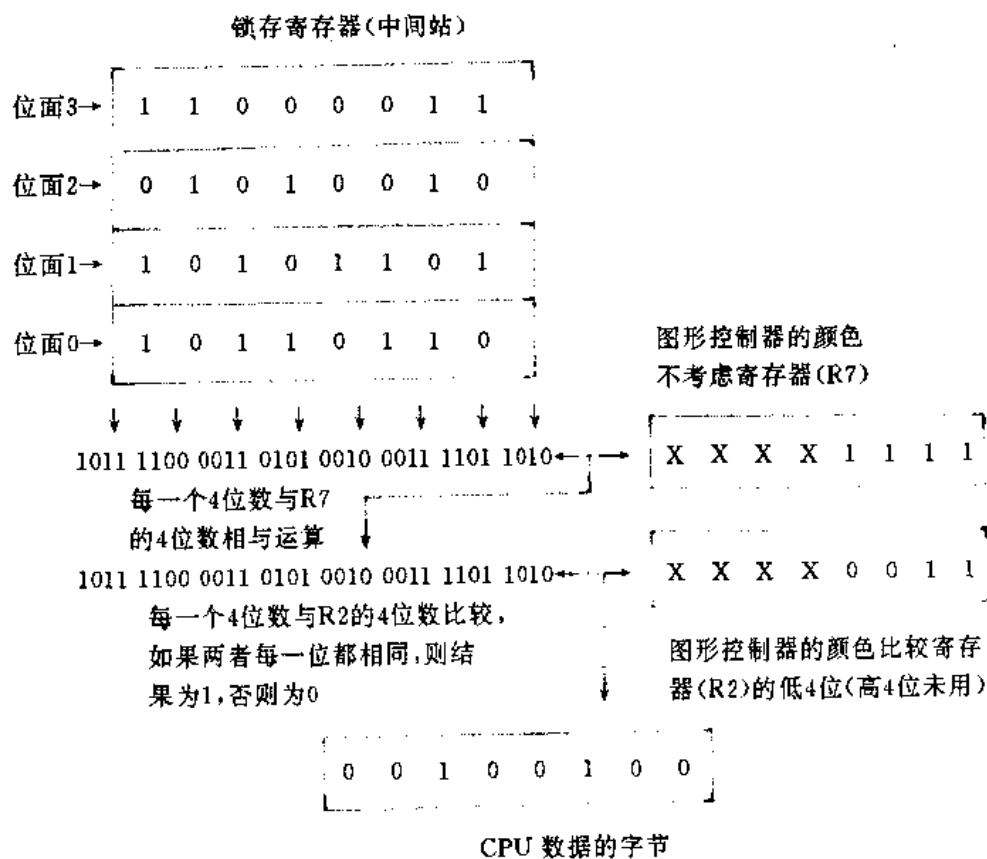


图 35-24 EGA 和 VGA 的读方式 1

## 二 写入方式

写入方式主要用于设置像素值。当接收到 CPU 发来的一字节数据,三种写模式都能改变四个位面中字节。CPU 数据的作用由几个寄存器的设置状态决定。

### (一) 写入方式 0

它比 INT 10H 的功能 CII 在速度和灵活性方面更优越一些,允许数据转置和位操作,它可以逐点控制每一个像素。分两种情况:

(1) 当允许设置 / 复位寄存器值为非 0 时

锁存寄存器(中间站)

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |

“允许设置/复位寄存器”的值,它控制“设置/复位寄存器”参与运算位

XXXX 1111

↓ (低4位参与运算)

1011 1100 0011 0101 0010 0011 1101 1010 ← XXXX 0111 设置/复位寄存器值

先根据位屏蔽寄存器的值决定哪几位参与运算

0000 1111 ← 位屏蔽寄存器的值(允许两者低4位参与运算,高4位不变)

再根据数据循环/功能选择寄存器的位4位3的值决定两者间运算法则

XXX00XXX ← 数据循环/功能选择寄存器的值(位4=0,位3=0)

↓ (2位含意:00→代换,01→与运算,10→或运算,11→异或运算)

1011 1100 0011 0101 0111 0111 0111 0111 ← 前4个二进制数未变,后4个全被代换成设置/恢复寄存器中的值

位面3 1100 0000

位面2 0101 1111 比较锁存器和最后结果,可以看出,由于位屏蔽寄存器值

位面1 1010 1111 为00001111,故每个位面字节的高4位不变,低4位要变。

位面0 1011 1111

图 35-25 EGA 和 VGA 写入方式 0 (允许设置 / 复位寄存器值为非 0 时)

(2) 当允许设置/复位寄存器值为 0 时

锁存寄存器(中间站)

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |

“允许设置/复位寄存器”的值,它控制“设置/复位寄存器”参与运算位

0000 0000 ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ (低4位参与运算)

1011 1100 0011 0101 0010 0011 1101 1010 ← 0001 0111 CPU 数据字节值

先根据位屏蔽寄存器的值决定哪几位参与运算

0000 1111 ← 位屏蔽寄存器的值(允许两者低4位参与运算,高4位不变)

再根据数据循环/功能选择寄存器的位0~位2的值决定将CPU中的字节数据在和锁存器中值合并前向右循环移动几位(这里由于低3位为0,而没。有向右移动)。而位4和位3的值决定两者间运算法则。

XXX00XXX ← 数据循环/功能选择寄存器的值(位4=0,位3=0)

↓ (2位含意:00→代换,01→与运算,10→或运算,11→异或运算)

1011 1100 0011 0101 0111 0111 0111 0111 ← 前4个二进制数未变,后4个全被代换成设置/恢复寄存器中的值



位面3 1100 0000  
 位面2 0101 1111 比较锁存器和最后结果,可以看出,这里用CPU 字节值  
 位面1 1010 1111 起作用,而设置/复位寄存器值未考虑。  
 位面0 1011 1111

图 35—26 EGA 和 VGA 写入方式 0( 允许设置 / 复位寄存器值为 0 时 )

## (二) 写入方式 1

在这个方式,先执行一个 CPU 的读取操作,用以设定锁存器的值,然后锁存器中的值被直接写入视频缓冲区的位面中。可用于从存储器的一块区域往另一块区域快速复制一块数据。

## (三) 写入方式 2

它使用一种特殊的颜色填充屏幕的某一区域。

锁存寄存器(中间站)

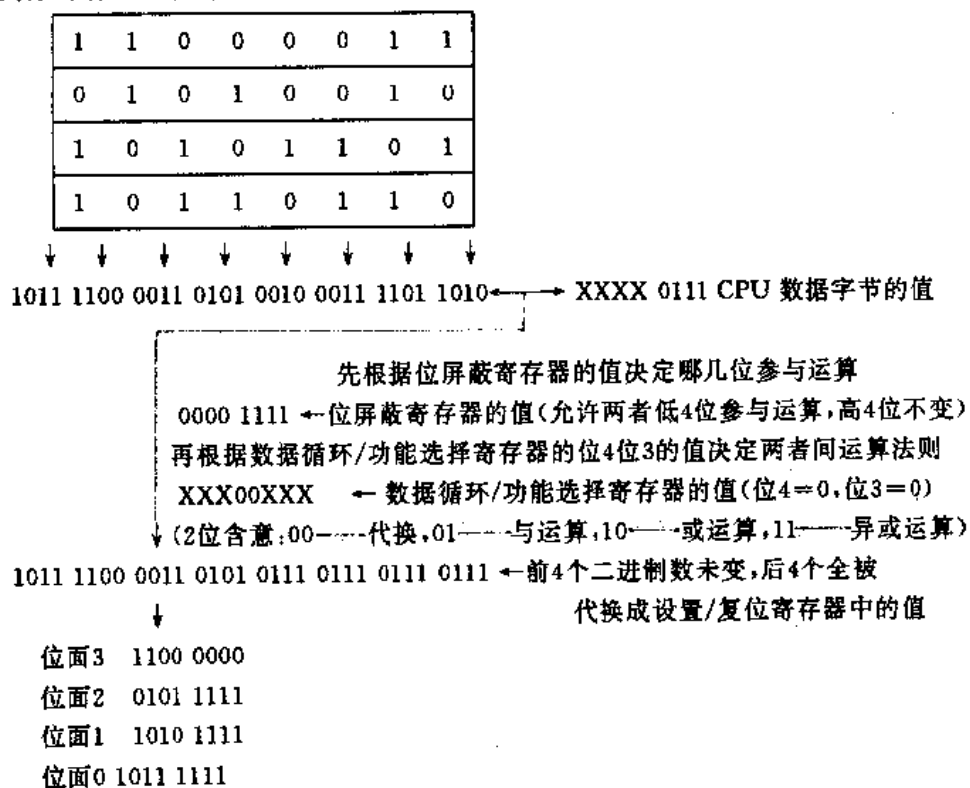


图 35—27 EGA 和 VGA 写入方式 2

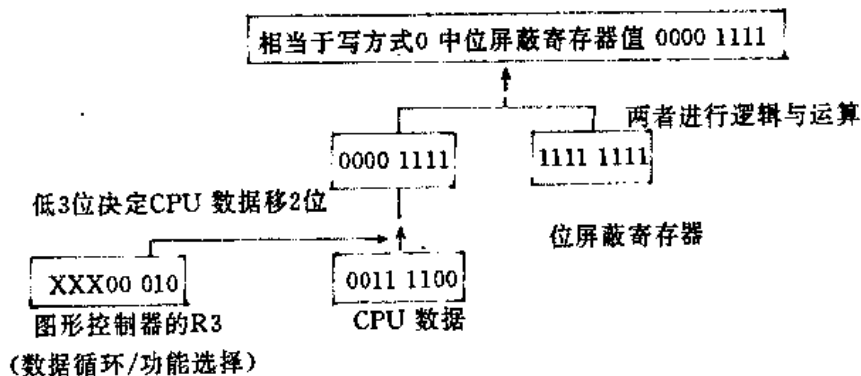


图 35—28 EGA 和 VGA 写入方式 3

#### (四) 写入方式 3

这个方式只有 VGA 才有, 它和写入方式 0 相似, 不同之处在于位屏蔽寄存器的功能由图 35-19 说明的结果代替。这种方式允许操作单个像素。

#### 35.6.2 变量 directvideo

在 conio.h 中定义了一个高性能的输出变量:

```
extern int _Cdecl directvideo;
```

该变量可以控制程序的控制台输出; 或是直接将输出送到视频缓冲区, 此时该变量的值为 1; 或是通过 BIOS 中断来输出, 此时应将该变量置值为 0。该变量的缺省值为 1。

一般情况下, 将控制输出直接送视频输出速度较快, 但要求计算机必须与 IBM PC100% 兼容, 即视频硬件必须与 IBM 相一致。在有些机上, 置变量为 0 并再调用 BIOS 中断常能很好地处理汉字输出问题。将变量置成 0 时, 可以在任何与 IBM BIOS 一级兼容的机器上运行, 不过这时的输出较慢。

#### 35.6.3 使用图形函数的注意事项

TC 提供了 70 多个图形库函数, 在使用这些函数时注意到如下几点是有好处的。

1. 若在集成环境里 (使用 TC.EXE) 调试、编译或执行含有图形函数的程序, 应将开关 Options/Linker/Graphics library

置成 On, 以便当程序连接时连接器 Linker 自动地将目标代码与 TC 的图形库 graphics.lib 相连接。

2. 若使用 TCC.EXE 编译连接, 则必须在命令行上加上 graphics.lib。例如程序 MY.C 中用到图形函数, 则 TCC 的命令行为 C>TCC MY GRAPHICS.LIB

注意: 图形库 graphics.lib 对各种内存模式都适用, 因为它将每个函数都定义为远调用 (far) 函数, 所用的指针都是远调用指针。为了使函数能在各种内存模式下运行, 使用图形库时应在源程序中包含标头文件 graphics.h。

用户也可以用 BIOS 中断 INT 10H 定义自己的图形函数, 这时可不包括 graphics.h。

#### 35.6.4 系统控制

1. 图形驱动程序名 (graphdriver name)

Turbo C 提供了 ATT.BGI、CGA.BGI、EGAVGA.BGI、HERC.BGI、IBM8514.BGI 和 PC3270.BGI 等图形驱动程序, 把程序基本名 (即不带扩展名) 称为【驱动程序名】。

图形驱动程序号由枚举 graphics-drivers 决定。(驱动程序和驱动程序号之间的关系参见表 35-1。)

图形【驱动程序号】在 graphics.h 中定义为 enum graphics-drivers {

```
DETECT, /* 即 DETECT=0, 让系统自动检测出所需驱动程序并用最高分辨率 */
CGA, /* 即 CGA=1, */
MCGA, EGA, EGA64, EGAMONO, IBM8514,
HERCMONO, ATT400, VGA, PC3270,
CURRENT_DRIVER = -1 /* 其它驱动程序 */
```

```
};
```

## 2. 【驱动程序的图形模式名】与【图形模式号】

Turbo C 用驱动程序选择显示方式。驱动程序的图形模式在 graphics.h 中定义为

```
enum graphics—modes {
```

```
/* 图形模式名 图形模式号 列×行 调色板号 页数 */
CGAC0 = 0, /* 320x200 调色板 0; 1页 */
CGAC1 = 1, /* 320x200 调色板 1; 1页 */
CGAC2 = 2, /* 320x200 调色板 2; 1页 */
CGAC3 = 3, /* 320x200 调色板 3; 1页 */
CGAHI = 4, /* 640x200 1页 */
MCGAC0 = 0, /* 320x200 调色板 0; 1页 */
MCGAC1 = 1, /* 320x200 调色板 1; 1页 */
MCGAC2 = 2, /* 320x200 调色板 2; 1页 */
MCGAC3 = 3, /* 320x200 调色板 3; 1页 */
MCGAMED = 4, /* 640x200 1页 */
MCGAHI = 5, /* 640x480 1页 */
EGALO = 0, /* 640x200 16 color 4页 */
EGAHI = 1, /* 640x350 16 color 2页 */
EGA64LO = 0, /* 640x200 16 color 1页 */
EGA64HI = 1, /* 640x350 4 color 1页 */
EGAMONOH = 0, /* 640x350 (64K 卡), 1页
 (256K 卡) 4页 */
HERCMONOH = 0, /* 720x348 2页 */
ATT400C0 = 0, /* 320x200 调色板 0; 1页 */
ATT400C1 = 1, /* 320x200 调色板 1; 1页 */
ATT400C2 = 2, /* 320x200 调色板 2; 1页 */
ATT400C3 = 3, /* 320x200 调色板 3; 1页 */
ATT400MED = 4, /* 640x200 1页 */
ATT400HI = 5, /* 640x400 1页 */
VGALO = 0, /* 640x200 16种颜色 4页 */
VGAMED = 1, /* 640x350 16种颜色 2页 */
VGAHI = 2, /* 640x480 16种颜色 1页 */
PC3270HI = 0, /* 720x350 1页 */
IBM8514LO = 0, /* 640x480 256种颜色 */
IBM8514HI = 1, /* 1024x768 256种颜色
```

```
};
```

## 3. 函数

- |                                 |                     |
|---------------------------------|---------------------|
| · 初始化图形系统,并将硬件置于图形方式            | —1 initgraph()      |
| · 关闭图形系统                        | —2 closegraph()     |
| · 恢复到由 initgraph() 检测到的原视频的文本模式 | —3 restorecrtmode() |
| · 将所有图形重设置为它们的缺省值               | —4 graphdefaults()  |
| · 获得当前使用的图形驱动程序名                | —5 getdrivername()  |
| · 获得当前图形驱动程序的图形模式名、分辨率          | —6 getmodename()    |
| · 获得当前驱动程序下可用的最大图形模式号           | —7 getmaxmode()     |
| · 得到当前图形模式号                     | —8 getgraphmode()   |

- 取得一个给定图形驱动程序的有效图形模式号范围 —9 getmoderange()
- 选择一个不同于 initgraph() 所设置的缺省图形方式 —10 setgraphmode()
- 检测硬件以决定所用的图形驱动程序号和图形模式号 —11 detectgraph()
- 返回当前驱动程序下屏幕坐标的最大 x (列) 值 —12 getmaxx()
- 返回当前驱动程序下屏幕坐标的最大 y (行) 值 —13 getmaxy()
- 图形光标的当前位置 (CP) 的 x 坐标 —14 getx()
- 图形光标的当前位置 (CP) 的 y 坐标 —15 gety()
- 调用图形库 graphics.lib 中例程来为图形驱动程序、字体或内部缓冲区分配的内存 —16 —graphgetmem()
- 释放 —graphgetmem() 为图形函数所分配的内存 —17 —graphfreemem()
- 改变内部图形缓冲区的大小 —18 setgraphbufsize()
- 安装新的图形驱动程序 —19 installuserdriver()
- 安装新的字体文件 —20 installuserfont()
- 注册驱动程序 —21 registerbgidriver()
- 注册字体 —22 registerfarbgidriver()
- 注册字体 —23 registerbgifont()
- 注册字体 —24 registerfarbgifont()

```
—1 void far —Cdecl initgraph(int far * graphdriver, int far * graphmode,
 char far * pathtodriver);
```

初始化图形系统 (装入正确的 \*.BGI 文件, 初始化各个内部变量和默认条件, 清屏), 并将硬件置于图形模式。

在指定路径 pathtodriver 中自动寻找图形驱动程序 graphdriver (即磁盘上的 \*.BGI 文件)。如路径 pathtodriver 为空, 则在当前目录中寻找。initgraph() 将图形的当前位置、调色板、颜色、视区等复位为缺省值。如果设置成功, 便清除整个屏幕, 装载图形驱动程序, 并使 graphresult() 返回 0, 否则返回错误码。设置的缺省图形模式为该图形驱动器的最高分辨率。设置成功与否, 可用函数 graphresult() 检测。

graphdriver 是存放图形驱动程序号变量的地址; graphmode 是存放驱动程序的图形模式号变量的地址。

如果发生了错误, \*graphdriver 和 graphresult() 都将返回同样的错误代码:

| 错误代码 | 意义           |
|------|--------------|
| -2   | 不能检测图形卡      |
| -3   | 不能找到驱动程序文件   |
| -4   | 无效驱动程序       |
| -5   | 无足够内存来加载驱动程序 |

注意: 如果你当前用的是 CGA 卡, 而用象程序 SCR10.C 那样进行图形初始化, 结果屏幕一定非你所需! 注意, 强制使用不存在的硬件, 后果往往不可预料。因此, 编制适于多种适配器的通用程序时, 应考虑自动判别硬件并作出相应处理。如有错误, 变量 i 便是错误返回码。例如, 当 E 盘上无任何图形驱动程序时, 程序返回 -3。

```
C>TYPE SCR10.C
#include "graphics.h"
main(){
```

```

int i;
 /* int far graphdriver=CGA,graphmode=1; 整型变量不能用 far
 Error: Conflicting type modifiers in function main */
int graphdriver=VGA,graphmode=1;
char far *pathtodriver="E:\\";
initgraph(&graphdriver,&graphmode,pathtodriver);
i=graphresult();
printf("%d\\n",i);
}

```

初始化图形系统有三种方法:

1. 使用 detectgraph() 函数。
2. 使用语句

```

int driver=DETECT,mode; /* 指定自动检测 */
initgraph(&driver,&mode,"");

```

它自动调用 detectgraph() 函数进行初始化。

3. 使用语句

```

int driver=CGA,mode=CGAHI; /* 指定驱动程序和模式 */
initgraph(&driver,&mode,"");

```

或

```

int driver=1,mode=4; /* 指定驱动程序和模式 */
initgraph(&driver,&mode,"");

```

注意:使用本法时指定的图形模式必须存在,否则不可预料。

```

C>TYPE SCR11.C
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#define PE(X) errorcode=graphresult();printf("n=%d %d\\n",X,errorcode);\\
printf("press any key, continue...\\n");getch()

main()
{
int g—driver=DETECT,g—mode,errorcode,textfont; /* 请求系统自动测试 */
PE(1);
textfont=installuserfont("SANS.CHR");
PE(2);
if(errorcode!=grOk)
{
printf("Error installing SANS.CHR font\\n");
exit(1);
}
initgraph(&g—driver,&g—mode,"");
PE(3);
if(errorcode!=grOk)
{

```



```
printf("0040:0010—0040:0011=0x%x\n",i0040—10—11);
i0040—10=peekb(0x40,0x10);
printf("0040:0010=0x%x\n",i0040—10);
}
```

/\* 程序输出:

AX=0x4263

0040:0010—0040:0011=0x4263 \*/

—2 void far —Cdecl closegraph(void);

关闭图形系统,屏幕上图形消失。释放所有图形系统分配的存储区(自动调用函数 —graphfreemem() 释放如驱动程序、字体和内部缓冲区等占用的存储区),然后将屏幕恢复为调用 initgraph() 之前的模式。

将驱动程序占用的释放也称为“将驱动程序从内存中卸下”。

关闭图形系统后,可用 restorecrtmode() 恢复初始的视频模式。

切忌在没有设置图形方式时使用它。

—3 void far —Cdecl restorecrtmode(void);

本函数使屏幕恢复到由 initgraph() 检测到的原视频的文本模式(text—mode)。它可以和函数 setgraphmode() 连用,以便屏幕可以在文本和图形模式之间互相转换。

—4 void far —Cdecl graphdefaults(void);

将所有图形重设置为它们的缺省值。即视区为整个屏幕;光标的当前位置(CP)为(0,0);调色板颜色、背景颜色、画线颜色、图形填充类型和模式、文本字体和对齐方式均为缺省值(参见用 TCINST.EXE 设置 TC 缺省参数)。

—5 char \* far —Cdecl getdrivername( void );

返回一个指向当前使用的图形驱动程序名(字符串)的远指针,也可以用它来测试硬件适配器。

—6 char \* far —Cdecl getmodename( int mode—number );

它返回由 initgraph() 或 setgraphmode() 所设置的当前图形模式。程序在使用本函数之前应先成功地调用 initgraph() 函数。图形模式参见枚举 graphics—mode。

```
C>TYPE SCR13.C
#include <graphics.h>
#include <stdio.h>
main()
{
int graphdriver=DETECT,graphmode;
char * driver, * mode;
initgraph(&graphdriver,&graphmode,"");
driver=getdrivername();
mode=getmodename(graphmode);
outtext("Using driver:");
outtext(driver);
outtext(" Using graphmode:"); /* 在串中转义符不起其原来作用 */
outtext(mode); /* 象串尾加 \n 时,显示并不换行 */
outtextxy(1,20,"TEST OK!"); /* 注意:这里坐标是指 x:0~639,y:0~199 */
getch();
closegraph();
```

```

}
/* 输出: Using driver:CGAUsing graphmode: 640x200 CGA
 TEST OK!
*/

```

—7 int far —Cdecl getmaxmode(void);  
 返回当前驱动程序下可用的最大显示模式号。

```

C>TYPE SCR14.C
#include "graphics.h"
main()
{
 int GraphMode,GraphDriver = DETECT,m1,m2;
 initgraph(&GraphDriver, &GraphMode, "");
 setgraphmode(1);
 m1=getmaxmode();
 m2=setgraphmode();
 printf("%s %d %d\n",getdrivename(),m1,m2); /* CGA 41 */
 getch();
 closegraph();
}

```

—8 int far —Cdecl getgraphmode(void);  
 返回由 initgraph() 或 setgraphmode() 设置的当前图形模式。如果给了setgraphmode() 一个对当前图形驱动无效的模式,本函数返回的是原先有效的模式。  
 使用本函数之前必须先成功地调用 initgraph() 函数。

```

C>TYPE SCR15.C
#include "graphics.h"
#include "stdio.h"
#include "conio.h"
#define X1 500
#define X2 5000
#define DE(X) delay(X)
#define TEXTMODE gettextinfo(&info);\
 printf("currmode=%u\n",info.currmode)
main()
{
 struct text—info info;
 int errorcode;
 int graphdriver;
 int graphmode=4; /* 初始化选模式 4 */
 graphdriver=1; /* 对 CGA 显示卡 */
 printf("test graphdriver: %d\n",getgraphmode());
 TEXTMODE; /* 输出:test graphdriver: -1 */
 DE(X1);
 initgraph(&graphdriver,&graphmode,"c:\\"); /* 初始化 */
 DE(X1);
 errorcode=graphresult();
}

```



```

if(errorcode!=grOk)
{
 /* 如用语句 graphdriver=12; */
 printf("graphics error!"); /* 则本程序段将被执行 */
 closegraph(); /* 因为枚举 graphics-drivers */
 exit(1); /* 中没有这样的图形驱动器 */
};
printf("Old graphmode=%d\n",graphmode);
DE(X2); /* 在延迟时间内,你可按任一键或不按,便有两种结果 */
if(kbhit()){getch();setgraphmode(8);}
else setgraphmode(3);
printf("New graphmode=%d\n",getgraphmode());
outtextxy(50,50,"BGI TEXT!");
printf("Press any key, continue ... ");
getch();
setgraphmode(0);
printf("mode=0: %d\n",getgraphmode());
DE(X2);
restorecrtmode(); /* 从图形模式切换到文本方式 */
TEXTMODE;
printf("restorecrtmode=%d\n",getgraphmode());
printf("Hit any key, continue ... ");
getch();
setgraphmode(3); /* 重新从文本模式切换到图形模式 */
printf("OK! ... Press key for End!");
getch();
closegraph();
}
/* 部分输出结果:
test graphdriver: -1
currmode=3执行initgraph()前测出text-mode=3
Old graphmode=4
New graphmode=3
Press any key, continue ... mode=0: 0
currmode=3 执行 restorecrtmode() 后测出的 text-mode
restorecrtmode=0 可见恢复到执行 initgraph() 之前的形式
Hit any key, continue ...
*/

```

```

—9 void far _Cdecl getmoderange(int graphdriver, int far *lomode,
 int far *himode);

```

取得一个给定图形驱动程序 graphdriver 的有效图形模式范围: \*lomode 中返回最低图形模式值; \*himode 中返回最高图形模式值。如参数 graphdriver 是一个无效值,则后两个参数均返回 -1。

```

C>TYPE SCR16.C
#include "graphics.h"
main(){

```

```

int lo,hi;
getmoderange(VGA,&lo,&hi);
printf("VGA: lomode=%d himode=%d\n",lo,hi);
getmoderange(14,&lo,&hi);
printf("NOT: lomode=%d himode=%d\n",lo,hi);
}
/* 程序输出:
VGA: lomode=0 himode=2 对 VGA:0, 1, 2
NOT: lomode=-1 himode=-1 对不存在的驱动程序输出
*/

```

—10 void far \_Cdecl setgraphmode(int mode);

选择一个不同于 initgraph() 所设置的缺省图形模式。mode 必须是一个对当前图形适配器是合法的图形模式（参见枚举 graphics—modes），否则，函数调用将不起作用，即图形模式仍为 initgraph() 设置的模式，graphresult() 返回 -10。

本函数调用有效时，它将清整个图形屏幕，将所有图形设置为缺省值（CP、调色板、颜色和视区等）。常用它和 restorecrtmode() 进行文本和图形模式之间的转换。使用本函数之前必须先成功地调用 initgraph() 函数。

—11 void far \_Cdecl detectgraph(int far \* graphdriver, int far \* graphmode);

通过检测硬件来确定当前系统所用的图形驱动程序号和图形模式号。graphdriver 指向存放图形驱动程序号的变量；graphmode 指向存放检测到的最大图形模式号的变量。函数本身不返回值，只是将检测到的两个变量的地址值给指针。如检测成功，调用 graphresult() 函数返回 0，否则该函数返回 -2。BIOS 数据区 0040:0049 中保存当前图形模式号。

其一般使用方法是：

```

1. int driver, mode; /* 定义两个整型变量 */
 detectgraph(&driver, &mode); /* &driver, &mode 为变量地址 */

```

函数调用后两个变量立即获得枚举 graphics—drivers 和 graphics—modes 中相应值。

```

2. int driver, mode;
 detectgraph(&driver, &mode);
 initgraph(&driver, &mode, "c:\\tc"); /* 检测后初始化 */

```

C>TYPE SCR17.C

```

#include "conio.h"
#include "graphics.h"
#define SCR scr4049=peekb(0x40,0x49);scr404a=peekb(0x40,0x4a); \
/* BIOS 在 0040:0049 中保存当前模式号,0040:001A 中保存文本模式下的列数 */
/* 注意:象scr4049=peekb(0x40,0x49);也可用语句
 scr4049=* (char far *)0x00400049; 代替,结果一样 */
printf("n=%d 0040:0049=%u 0040:004A=%u\n",n,scr4049,scr404a)
#define PS(X) n=X;textmode(n);SCR
/* 将屏幕按模式值 n 值置成文本模式 */
#define PKEY printf("press any key,continue\n");getch()
main(){
int n;
unsigned char scr4049,scr404a;
int g—driver,g—mode,g—result;

```

```

int g—mode—change;
PS(C80);
PKEY;
PS(C40);
PKEY;
detectgraph(&g—driver,&g—mode);
printf("g—diver=%d g—mode=%d\n",g—driver,g—mode);
if(g—driver<0)exit(1);
SCR;
PKEY;
initgraph(&g—driver,&g—mode,""); /* 初始化图形系统 */
g—result=graphresult();
/* graphresult 返回最后一次不成功的图形
 操作的错误代码,如无错误,返回 0 */
if(g—result<0)
{printf("error!\n");exit(1);}
printf("g—result=%d\n",g—result);
delay(5000); /* 延迟 5000 毫秒 */
g—mode—change=g—mode-1;
if(g—mode—change<0)g—mode—change=0;
setgraphmode(g—mode—change); /* 可设置任意一个合法的图形模式 */
printf("g—result=%d\n",g—result);
SCR;
PKEY;
PS(LASTMODE);
}
/* 在 CGA 上输出的部分内容
n=3 0040:0049=3 0040:004a=80
n=1 0040:0049=1 0040:004a=40
g—driver=1 g—mode=4
n=1 0040:0049=1 0040:004a=40
g—result=0 No error
n=1 0040:0049=6 0040:004a=80
n=-1 0040:0049=1 0040:004a=40 */
-12 int far —Cdecl getmaxx(void);
返回当前驱动程序下屏幕坐标的最大 x(列)值。
-13 int far —Cdecl getmaxy(void);
返回当前驱动程序下屏幕坐标的最大 y(行)值。
C>TYPE SCR18.C
#include "graphics.h"
#define PCP printf("%d %d\n",getx(),gety())
main(){
int GraphMode,GraphDriver = DETECT,mx,my;
initgraph(&GraphDriver, &GraphMode, "");
PCP; /* 0 0 */

```

```

setgraphmode(1); /* 设显示模式 1 320 × 200 */
PCP; /* 0 0 */
setviewport(1,2,250,180,1);
PCP; /* 0 0 */
mx=getmaxx();
my=getmaxy();
printf(" %s %d %d\n",getdrivername(),mx,my); /* CGA 639 199 */
PCP; /* 0 0 */
moveto(100,110);
PCP; /* 100 110 */
outtext("CP");
PCP; /* 116 110 */
getch();
closegraph();
}

```

程序 SCR18.C 说明 getmaxx() 和 getmaxy() 与设置的显示模式、设置的视口无关。

—14 int far —Cdecl getx(void);

返回当前图形模式下,图形光标的当前位置 (Cursor Position, 简称【CP】) 的 x 坐标。坐标是相对于视口的。

注意:在图形方式下 CP 处并无实际的光标显示。因此常用本函数和 gety() 查询其实际位置。

—15 int far —Cdecl gety(void);

返回当前图形光标位置 (CP) 的 y 坐标。

—16 void far \* far —Cdecl —graphgetmem(unsigned size);

通常,该函数可以不在用户源程序中出现,即不出现函数调用。但当用户程序执行时,TC 调用图形库 graphics.lib 中例程来为图形驱动程序、字体或内部缓冲区 (缺省值为 4096 个字节) 分配的内存 (执行到相关位置才分配)。这种缺省的分配是调用函数 malloc() 进行的。用户可以定义自己的 —graphgetmem() 函数,但必须考虑到用 —graphgetmem() 释放,否则分配的空间可能不会被自动释放。用户可以在自己定义的这个函数中调用函数 farmalloc(), 而不是调用 malloc()。注意:当用户定义了自己的 —graphgetmem() 函数后,在 main() 主函数或其它函数中也可以不出现调用该函数的语句,但它仍将在调用需要分配内存的图形函数时自动被执行。

你可以对程序 SCR19.C 用 F7 进行单步跟踪,便不难得上述结论。注意单步调试不能进入函数 —graphgetmem() 和 —graphfreemem() 内。

当用户自定义上面两个函数时,一是注意定义的函数名字必须使用这种名字,而不能用别的函数名;其次,如果用了自己的定义的函数后,在编译时,由于 graphics.lib 图形库中已有这两个函数,有时可能出现 duplicate symbols 警告信息,可不管它。

—17 void far —Cdecl —graphfreemem(void far \* ptr, unsigned size);

通常,该函数可以不在用户源程序中出现,即不出现函数调用。但当用户程序执行时,TC 调用图形库 graphics.lib 中例程来释放 —graphgetmem() 为图形函数所分配的内存 (它缺省分配的缓冲区内内存为 4096 个字节)。这种缺省的释放是调用函数 free() 进行的。用户可以定义自己的 —graphfreemem() 函数,但必须考虑到 —graphgetmem() 的定义内容,否则不正确的定义可能会带来意想不到的结果。用户可以在自己定义的这个函数中调用函数 farfree

()，而不是调用 free()。注意：当用户定义了自己的—graphfreemem() 函数后，在 main() 主函数或其它函数中也可以不出现调用该函数的语句，但它仍将在执行 closegraph() 函数即关闭图形系统时被执行。

```
C>TYPE SCR19.C
#include "graphics.h"
#include "stdio.h"
#include "conio.h"
#include "alloc.h"
main()
{
 int errorcode;
 int graphdriver;
 int graphmode=0;
 graphdriver=CGA; /* 图形驱动程序选 CGA */
 initgraph(&graphdriver,&graphmode,"c:\\");
 /* 在 C 当前目录中要有 CGA.BGI 图形驱动程序 */
 errorcode=graphresult();
 if(errorcode!=grOk) /* 检查图形初始化是否成功 */
 {
 printf("graphics error,%s\n",grapherrormsg(errorcode));
 closegraph();
 exit(1);
 };
 /* 设置字体为哥特体,方向从 */
 settextstyle(GOTHIC_FONT,HORIZ_DIR,4); /* 左到右,字符大小因子为 4 */
 /* 在当前目录中应有哥特字体文件 GOTH.CHR */
 outtextxy(100,100,"BGI TEST"); /* 在视区 (100,100) 处显示字母 */
 outtext("BGI TEXT!"); /* 在 CP 处输出字符 BGI TEXT! */
 getch(); /* 等待按键 */
 closegraph(); /* 关闭图形系统 */
}

/* 注意：—graphgetmem 和 —graphfreemem 是两个专用名 */
void far *far —graphgetmem(unsigned size) /* 自定义分配内存函数 */
{
 /* 参数 size 不由用户确定,而由系统根据调用图形函数自动确定 */
 printf("—graphgetmem called[size=%d]—hit any key",size);
 getch();printf("\n"); /* 屏幕输出 size 值,供用户观察 */
 return(farmalloc(size)); /* 如果分配失败,返回NULL */
}
/* 注意此分配语句一定要有。 */
void far —graphfreemem(void far *ptr,unsigned size)/* 自定义释放函数 */
{
 /* 参数 ptr,size 均不用用户指定,由系统自行确定 */
 printf("—graphfreemem called[size=%d]—Press any key",size);
 getch();printf("\n");
 farfree(ptr); /* 释放远堆中以前 —graphgetmem() 分配的块 */
}
/* 注意,此释放语句必须有! 其它语句可随意 */
/* 单步调试时部分输出:
分配时:
—graphgetmem called[size=6253]—hit any key
```

```

—graphgetmem called[size=4096]—hit any key
—graphgetmem called[size=8576]—hit any key
释放时:
—graphfreemem called[size=4096]—Press any key
—graphfreemem called[size=6253]—Press any key
—graphfreemem called[size=8576]—Press any key
调试中用 Compile/Get info 可查看有效空间:

```

Available memory: 272K

你不妨将释放函数去掉再调试,看看这

一信息有什么变化 \*/

```

—18 unsigned far —Cdecl setgraphbufsize(unsigned bufsize);

```

该函数能改变内部图形缓冲区的大小。一些图形函数(如 floodfill()) 要使用 initgraph() 调用时所分配的存储缓冲区,该缓冲区在调用 closegraph() 时被释放。由 —graphgetmem() 所分配的这个缓冲区的缺省大小为 4096 个字节。

有时为节省存储空间希望缓冲区小一些,或者象调用 floodfill() 出错时要增大缓冲区时就可使用本函数。setgraphbufsize() 通知 initgraph() 在调用时为内部图形缓冲区应分配多少存储空间。

从程序 SCR20.C 可以看出,使用本函数时应注意两个问题,一是它和 initgraph() 同时出现时,本函数应在 initgraph() 函数之前;其次,首次分配的是缺省值 4096,而不是指定的 size 值。

注意:程序中多次使用本函数只是示例,实际使用中也许不要那么多。

它返回内部缓冲区原来的大小。

```

C>TYPE SCR20.C
#include "graphics.h"
#include "alloc.h"
#include <stdio.h>
#include <conio.h>
#include <process.h>
main(){
int cbsize,x;
int g—driver,g—mode,g—error;
x=setgraphbufsize(10);
printf(" %d free= %d\n",x,coreleft());
cbsize=setgraphbufsize(1000);
printf(" %d free= %d\n",cbsize,coreleft());
detectgraph(&g—driver,&g—mode);
if(g—driver<0)
{printf("No graphics hardware detected!\n");
exit(1);
}
printf("Detected graphics # %d,mode # %d\n",g—driver,g—mode);
getch();
if(g—mode==CGAHI)
{printf("g—mode=CGAHI\n");g—mode=CGAC0;}

```

```

cbsize=setgraphbufsize(10);
printf(" %d free=%d\n",cbsize,coreleft());
initgraph(&g—driver,&g—mode,""); /* 如将上两句放到本句后面,则出错 */
g—error=graphresult();
if(g—error<0)
{
 printf("initgraph error : %s\n",grapherrormsg(g—error));
 exit(1);
}
bar(0,0,getmaxx()/2,getmaxy()); /* 显示一个矩形 */
getch();
closegraph();
x=setgraphbufsize(3000);
printf(" %d free=%d\n",x,coreleft());
cbsize=setgraphbufsize(2000);
printf(" %d free=%d\n",cbsize,coreleft());
}
/* 部分输出结果.单步调试时按 F7 或 F8 键,则 getch() 语句跳过。
4096 free=-3348
4096 free=-3868
Detected graphics #1,mode #4
g—mode=CGAHI
1000 free=-3868
1000 free=-3868
3000 free=-3868
*/

—19 int far —Cdecl installuserdriver(char far * name,
 int huge (* detect)(void));

```

TC 磁盘上只有 ATT. BGI、CGA. BGI、EGAVGA. BGI、HERC. BGI、IBM8514. BGI、PC3270. BGI 等 6 个图形（设备）驱动程序,除此之外,用户可以用本函数安装新的图形驱动程序。

这里 \* name 是驱动程序名 (\*. BGI), detect 是一指向检测初始化参数的函数（它没有参数）并返回一个整数的指针。

有两种使用厂商提供的驱动程序的方法。一种是厂商只提供驱动程序,如 NEWGA. BGI,但没有提供自动检测图形初始化参数的函数,比如叫 detectNEWGA();。另一种则两者都提供。那末,对前者用户程序中可用

```

int graphdriver, graphmode;
graphdriver=installuserdriver("NEWGA. BGI",NULL); /* 检测函数为 NULL,即空 */
graphmode=0; /* 用户设定 */
initgraph(&graphdriver,&graphmode,""); /* 设 NEWGA. BGI 在当前目录中 */

```

对后者用

```

int graphdriver, graphmode;
graphdriver=installuserdriver("NEWGA. BGI",detectNEWGA());

```

```
initgraph(&graphdriver,&graphmode,"");
```

其中 detectNEWGA() 的返回值赋给变量 graphmode。

调用成功时返回驱动程序安装号,否则 graphresult() 可能返回

- 3 无此文件
- 4 无效驱动器名
- 5 内部驱动程序表满
- 11 图形错误

```
-20 int far _Cdecl installuserfont(char far *name);
```

TC 磁盘上只有 GOTH.CHR、LITT.CHR、SANS.CHR、TRIP.CHR 四个字体文件,要使用新的字体文件,例如 TEXT.CHR,便可用本函数进行(\*name 就是 TEXT.CHR)。注意:生成的新字体文件一定要有 \*.CHR 的形式。如你直接将 GOTH.CHR 改成 TEXT.CHR,然后用 SCR21.C 来执行,会产生错误代码 -14。但如用 DEBUG.COM 文件稍对它修改(如程序尾所指出的那样),便可进行了(注意:对其它三个文件修改的字节位置可能与此不同,但都是改文件名)。

本函数返回一个安装标志(ID)号,程序中是 textfont 变量值。程序中给出使用新字体的方法(调用函数 setttextstyle())。

如果安装成功,返回新字体安装号 11。否则 graphresult() 函数返回下列值之一:

- 8 未找到字体文件
- 9 装载字体时内存不够
- 11 图形错误
- 12 图形I/O 错误
- 13 无效的字体文件
- 14 无效的字体号

注意:执行程序 SCR21.C 时,当前目录里应有测试出来的图形驱动程序(例如CGA.BGI)和字体文件(TEXT.CHR),否则出错。

```
C>TYPE SCR21.C
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#define PE(X) errorcode=graphresult();printf("n=%d %d\n",X,errorcode);\
printf("press any key, continue...\n");getch()
#define ERR(Y) if(errorcode!=grOk){printf("#Y": %d\n",errorcode);\
exit(1);}

main()
{
int g=driver=DETECT,g=mode,errorcode,textfont;
PE(1);
textfont=installuserfont("TEXT.CHR");/* 对老字体也起作用,不过不必安装 */
printf("textfont=%d\n",textfont);
PE(2);
ERR(Error:installing TEXT font);
initgraph(&g=driver,&g=mode,"");
```



```

PE(3);
ERR(graphics error1);
settextstyle(textfont,0,4); /* 安装新字体或老字体语句 */
 /* 安装新字体等效语句: textfont 一定是11, settextstyle(11,0,4); */
/* textfont 对 TRIP.CHR,LITT.CHR,SANS.CHR 和 GOTH.CHR 分别为 1,2,3,4 */
PE(4);
ERR(graphics error2);
cleardevice();
outtext("Instable font support...\0");
getch();
closegraph();
) /* 使用 DEBUG.COM 文件将 GOTH.CHR 拷贝出一个 TEXT.CHR 文件的方法为:
 c>debug goth.chr
 -d100
 -e015b 47.54 4F.45 54.58 48.54
 -n text.chr
 -w
 -q
 */

-21 int -Cdecl registerbgidriver(void (* driver)(void));
-22 int far -Cdecl registerfarbgidriver(void far * driver);
-23 int -Cdecl registerbgifont(void (* font)(void));
-24 int far -Cdecl registerfarbgifont(void far * font);

```

前两个函数用法大致是一样的。用于注册驱动程序。后两个用于注册字体。在执行程序 SCR21.C 时,当前目录中应有驱动程序 CGA.BGI 和要用的字体文件 TEXT.CHR,那么能否不要它们呢?从根本上说,是不行的。但是可以预先将它们装入用户的执行程序中(\*.EXE),以后便可直接执行用户程序,而不用担心目录中是否有驱动程序或字体文件了。这一过程便可用本函数及相应的预处理实现。

将驱动程序 (\*.BGI) 和字体文件 (\*.CHR) 连接到用户执行文件前的预处理

(1) 如果 TC 磁盘上没有指定驱动程序的目标文件 (\*.OBJ),则用 TC 磁盘上的 BGIOBJ.EXE 文件生成:

C>BGIOBJ □ CGA 最后便可看到有 CGA.OBJ 文件生成。

(2) 如果无字体的目标文件,则可仿照生成驱动目标文件的方法生成:

C>BGIOBJ □ GOTH /\* 注意不要用 BGIOBJ □ GOTH.CHR 命令行 \*/ 便有 GOTH.OBJ 生成。

(3) 在用户源程序里必须有注册函数 regisrebgidriver() 与 regiserbgifont()。如果没有它们,即使你已将驱动程序或字体文件连接到执行程序中,也无任何作用。换句话说,执行时将出现错误 (grFileNotFound = -3 (参见枚举 graphics—errors))。此时,除非驱动文件或字体文件在当前目录中,才能避免这种错误出现。这就是把这些函数作用称为“注册已连接进用户执行文件中的图形驱动程序或字体文件代码”的真实含义。如果调用成功,registerbgidriver() 返回一个内部驱动程序号,registerbgifont() 返回注册的字体号。结果把驱动程序或字体代码连接到用户执行程序中。

(4) 在集成环境下,先用编辑器生成一个规划文件 BGICHR.PRJ,内容为

```
C>TYPE SCR22.prj
SCR22.C
CGA.OBJ
GOTH.OBJ
```

这几个文件应在当前目录下。

(5) 在集成环境下,选 Project/Project name 项并输入规划文件名 SCR22.PRJ,然后按 F9 热键进行编译连接生成 SCR22.EXE 用户可执行文件。现在你可以直接用 SCR22.EXE 了,而且可以不管当前目录中是否有驱动程序或字体文件了。

上述过程也可以用 TCC.EXE 完成。

注意:graphics.h 中以下几个函数不能为用户调用,它专用于图形驱动程序和字体连接。

```
void —Cdecl CGA—driver(void);
void —Cdecl EGAVGA—driver(void);
void —Cdecl IBM8514—driver(void);
void —Cdecl Herc—driver(void);
void —Cdecl ATT—driver(void);
void —Cdecl PC3270—driver(void);
void —Cdecl triplex—font(void);
void —Cdecl small—font(void);
void —Cdecl sansserif—font(void);
void —Cdecl gothic—font(void);
```

在用 detectgraph() 决定了使用那种图形驱动程序之后,initgraph() 将检查所需的驱动程序是否注册。若已注册,initgraph() 将直接从内存中使用驱动程序(驱动程序已和用户执行文件同时装入内存)。否则,initgraph() 将为此驱动程序分配内存,并从磁盘上读入相应的驱动程序(\*.BGI)。

```
C>TYPE SCR22.C
#include <graphics.h>
#include <stdio.h>
#include <conio.h>
#define PE(X) errorcode=graphresult();printf("n=%d %d\n",X,errorcode);\
printf("press any key, continue...\n");getch()
#define ERR(Y) if(errorcode!=grOk){printf("# Y": %d\n",errorcode);\
exit(1);}

main()
{
 int j,k;
 int g—driver=DETECT,g—mode,errorcode,txtfont;
 if((j=registerbgidriver(CGA—driver))<0)exit(1); /* 注册驱动程序 */
 if((k=registerbgifont(gothic—font))<0)exit(1); /* 注册字体 */
 /* 或 if((j=registerfarbgidriver(CGA—driver))<0)exit(1);
 if((k=registerfarbgifont(gothic—font))<0)exit(1); */
 printf("registerbgidriver=%d registerbgifont=%d\n",j,k);
 /* 输出 registerbgidriver=0 registerbgifont=4 */
```

```

PE(1);
PE(2);
ERR(Error,installing GOTH font);
initgraph(&g—driver,&g—mode,""); /* 注意,使用注册函数必须在本语句前 */
PE(3);
ERR(graphics error1);
settextstyle(4,0,4); /* 字体不为4时,则未连入并注册的字体文件应在 */
PE(4); /* 当前目录中.否则会出错 */
ERR(graphics error2);
cleardevice();
outtext("Instable font support...\0");
getch();
closegraph();
}

```

### 35.6.5 屏幕及视口管理

#### 一 屏幕管理

- . 清图形屏幕 (活动页) —1 cleardevice()
- . 使指定页成为图形活动页 —2 setactivepage()
- . 使指定页成显示页 —3 setvisualpage()

#### 二 视口管理

- . 只清除当前视区内的图形 —4 clearviewport()
- . 获取当前视口的信息 —5 getviewsettings()
- . 定义一个新的视口 —6 setviewport()

#### 三 图象管理

- . 将屏幕上一个矩域的位象图存到主存储区中 —7 getimage()
- . 将以前用 getimage() 保存的图象重新送回屏幕 —8 putimage()
- . 保存指定矩域所需要用的字节数 —9 imagesize()

#### 四 象素管理

- . 取得指定视口内一点 (x,y) 处的象素值 —10 getpixel()
  - . 在视口内 (x,y) 处画一象点 —11 putpixel()
- 1 void far —Cdecl cleardevice(void);

清除整个图形屏幕的内容 (活动页), 并将图形当前位置 CP (注意: 图形屏幕初始化后并无光标显示, 但若将整个屏幕当作一个视口, 则这个视口的左顶左上角为 CP) 移到原点 (0, 0)。

注意: closegraph() 也清除屏幕, 并关闭整个图形系统。

—2 void far —Cdecl setactivepage(int page);

本函数使 page 成为图形【活动页】, 亦即是说, 以后的图形输出都将输出到这个页。

—3 void far —Cdecl setvisualpage(int page);

本函数使 page 页成【显示页】。

C>TYPE SCR23.C

```
#include <graphics.h>
#include <stdio.h>
main()
{
 struct viewporttype view, *w;
 int graphdriver=DETECT,graphmode;
 initgraph(&graphdriver,&graphmode,"");
 setvisualpage(0); /* 对 CGA 只有一个有效图形页,如将页号写成 1 */
 setactivepage(0); /* 则下面的 printf 语句只能打出一个 K 字母 */
 printf("llll\nkkkkkkkkkkkkkkkkkk");
 outtextxy(40,40,"pppppp"); /* 对 EGA 如将页号填成 2,则看不到输出 */
 getviewsettings(&view);
 printf("left=%d top=%d right=%d bottom=%d clip=%d\n",
 view.left,view.top,view.right,view.bottom,view.clip);
 w=&view;
 getviewsettings(w); /* 使用指针输出 */
 printf("left=%d top=%d right=%d bottom=%d clip=%d\n",
 w->left,w->top,w->right,w->bottom,w->clip);
 getch();
 closegraph();
}
/* 输出结果:
 lll
 kkkkkkkkkkkkkkkkkleft=0 top=0 right=639 bottom=199 clip=1
 left=0 top=0 right=639 bottom=199 clip=1
 pppppp
*/
```

—4 void far —Cdecl clearviewport(void);

只清除当前视区内的图形,并将 CP 移到原点 (0,0)。

单步调试程序 SCR24.C 可以看到几种清图形函数之间的区别。

C>TYPE SCR24.C

```
#include <graphics.h>
#include <stdio.h>
#define OUT(X1,Y1,X2,Y2) setviewport(X1,Y1,X2,Y2,1);\
 outtext("Using driver,");bar3d(10,10,30,30,8,1)
/* 设置视口,输出文字及图形 */

main()
{
 int graphdriver=DETECT,graphmode;
 initgraph(&graphdriver,&graphmode,"");
 OUT(90,90,300,130);
 OUT(9,9,230,70);
 clearviewport(); /* 清当前视口 */
```

```

OUT(90,90,300,130);
OUT(9,9,230,70);
cleardevice(); /* 清屏幕 */
getch();
printf("TEST end");
closegraph(); /* 屏幕上 TEST end 被清除 */
}

```

由于各种视频系统（单色卡除外）都有较多的存储器，因此可以将它分成多个视频缓冲区，每一个缓冲区至少对应一个整屏幕的数据。我们把这样的每一个缓冲区称为一【页】。送入显示器的数据可以来自视频存储器的不同页。页数的划分由视频缓冲区的读写存储量决定。视频缓冲区的第一页的页号为 0，以后页号逐次增 1。

注意：在文本方式下一般有八页，每页为 2K 或 8K 字节。这里讲的是在图形方式下，其页数跟图形模式相关（参见枚举 graphics—modes）。对显示模式 4 ~ 10，在高分辨率四彩色模式和中分辨率 16 彩色模式的一页为 32K，而其余模式为 16K。对其它显示模式，页占字节一般较大，例如显示模式 15 和 16，每页至少要 128K。

单色卡上的存储器不能满足分多页的需要，因此可将主存（即用于存储用户数据的内存区）划出一部分作为视频缓冲区。通过快速交换视频缓冲区及单色卡上的存储器缓冲区（位于 B000:0000）的全部数据即可分页。主存内这块缓冲区称为【伪页】。

使用分页时应注意，每一时刻可显示页（即该页的数据全在屏幕上显示）只有一页，缺省的显示是 0 页。其次，按当前能否读 / 写页内数据，又可将页分为【活动页】和【不活动页】。活动页可以是显示页，也可以不是显示页，但输出的数据可以写入该页。

BIOS 数据区内有一个字节 0040:0062 中存储着当前显示页的页号（值的范围为 ~ 7）。

在 TC 中，用函数 setactivepage() 设置活动页，用函数 setvisualpage() 定义显示页。活动页原来不是显示页的，被 setvisualpage() 设置后立即显示，与此同时，原来的显示页变为不显示页。

对不同显示器和适配器，可用页数是不尽相同的。在使用这两个函数时，应正确填入页号。不正确的页号可能产生不可预料的结果。

EGA 和 VGA 有多个有效页，程序可以将图形先输出到一个不显示的活动页内，然后调用显示页的函数 setvisualpage() 来快速显示原来关闭的图象，该技术在动画中特别有用。

—5 void far cdecl getviewsettings(struct viewporttype far \*viewport);

获取当前视口的信息，并将信息填入指针 viewport 所指的结构中。结构 viewporttype 在标头文件 graphics.h 定义为

```

struct viewporttype {
 int left,top,right,bottom; /* 视口坐标 */
 int clip;
};

```

其中 clip 的含义是，如果 clip 非 0，则所有图形在当前视口边缘区被剪切掉；为 0 时则不剪切。运行程序 SCR25.C 后，对其含意你便有明确的理解。

```

C>TYPE SCR25.C
#include <graphics.h>
#include <stdio.h>
main()

```

```

{
struct viewporttype view, * w;
int graphdriver=CGA, graphmode=1; /* 注意,对确定的驱动程序和模式应 */
int j; /* 同时给出,否则会出错 */
initgraph(&graphdriver, &graphmode, "");
setviewport(1,1,20,20,1); /* 剪切后只输出两个 pp */
for(j=0; j<5; j++) outtextxy(1,1+j*8, "pppppppppppppppppppp");
getch();
setviewport(1,1,20,20,0); /* 未剪切可越出视口显示多个 p */
for(j=0; j<5; j++) outtextxy(1,1+j*8, "pppppppppppppppppppp");
getch();
closegraph();
}

```

—6 void far —Cdecl setviewport(int left, int top, int right, int bottom,  
int clip);

本函数为图形输出定义一个新的视口。视口的左上角坐标为 (left, top), 右下角为 (right, bottom), 它们都是屏幕的绝对坐标 (对 C80 屏幕 right 最大 79, 行 bottom 最大 24)。其最大值由屏幕的显示模式的分辨率决定。调用本函数后, CP 便是视口的左上角 (0,0)。

如果输入的数据无效 (例如坐标超出范围), 则用 graphresult() 函数检查时返回 -1, 当前视口保持不变, 即本函数调用无效。

—7 void far —Cdecl getimage(int left, int top, int right, int bottom,  
void far \* bitmap);

本函数将屏幕上一个矩域的位象图 (由矩域内实际占用的象素构成) 存到主存储区中。定义矩域的四个坐标 (left、top、right 和 bottom) 为屏幕的绝对坐标。bitmap 为指向主存储区中存放图象的区域的指针。该区域的前两个字节用于存放矩形的宽 (right-left), 随后的两个字节存放矩形的高 (bottom-top), 其余部分存放图象本身, 最后还加上两个字节存放 ASCII 码 0, 作为结束标志。存取图象不能太大, 其最大占用字节由 imagesize() 决定 ( $\leq 64K-1$ )。

—8 void far —Cdecl putimage(int left, int top, void far \* bitmap, int op);

将以前用 getimage() 保存的图象重新送回屏幕。图象的左上角位于 (left, top)。bitmap 指向由 getimage() 保存的源图象区域。参数 op 是一个组合算子, 用于控制源图象返回到屏幕上的颜色。换言之, 图象重新写到屏幕上时, 既可以完全是源图象颜色, 也可以是别的颜色。op 在 graphics.h 的枚举 putimage—ops 中定义。

```

enum putimage—ops{
COPY—PUT, /* 将源图象拷贝到屏幕, 屏幕上原图象不再存在 */
XOR—PUT, /* 将源图象与屏幕上已有图象进行“异或”运算(实际是表示象素的字节的二进制位
之间的“异或”运算, 参见《位运算与位域》一章)。这种图象运算的好处是两次“异
或”运算恢复原象。因而在动画中常用 */
OR—PUT, /* 源图象与屏幕已有图象进行“或”运算 */
AND—PUT, /* 源图象与屏幕已有图象进行“与”运算 */
NOT—PUT, /* 拷贝源图象“非” */
};

```

—9 unsigned far —Cdecl imagesize(int left, int top, int right, int bottom);

本函数用于保存指定矩域的所需的字节数,这些字节数包括存储图象字节数和保存矩域高和宽的字节及最后两个结尾字节(ASCII 码 0)。它主要用来为 getimage() 函数确定所需的存储块,因此,一般这两个函数的坐标应一致,或者 getimage() 定义的矩形尺寸应小于 imagesize() 定义的尺寸。否则,由于 TC 在调用 getimage() 时并不对内存范围作检查,有可能产生意想不到的结果。如果所选图象的大小大于或等于 64K-1 字节,函数返回 0xFFFF(即 -1)。

存取屏幕图形例参见程序 SCR26.C 和 SCR27.C。

```
C>TYPE SCR26.C
#include "stdio.h"
#include "graphics.h"
#include "fcntl.h"
#include "stdlib.h"
main() /* 本程序说明如何存取屏幕图象到磁盘文件 */
{
 int d=VGA,m=VGAHI;
 void *ptr;
 size_t size;
 int x1=30,y1=40,x2=70,y2=80;
 int handle;
 initgraph(&d,&m,""); /* 初始化图形屏幕 */
 setcolor(RED);
 rectangle(x1,y1,x2,y2); /* 画一红色矩形 */
 size=imagesize(x1,y1,x2,y2); /* 计算保存位图象所需字节数 */
 if(size==0xFFFF)exit(2); /* 如超过 64K-1 字节,则为 0xFFFF */
 ptr=malloc(size); /* 分配空间 */
 getimage(x1,y1,x2,y2,ptr); /* 将位图象存到主存储区 */
 handle=open("a",O_RDWR|O_BINARY); /* 打开文件 */
 if(handle==-1)exit(1);
 write(handle,ptr,size); /* 将位图象保存到磁盘文件 */
 close(handle);
 free(ptr);
 delay(1000); /* 延时供观察 */
 clearviewport(); /* 清视口 */
 delay(500);
 handle=open("a",O_RDONLY);
 if(handle==-1)exit(1);
 size=filelength(handle); /* 获得文件长度 */
 ptr=malloc(size);
 read(handle,ptr,size); /* 从磁盘文件读出图象数据 */
 putimage(x1,y1,ptr,COPY_PUT); /* 将原保存的图象拷贝到屏幕 */
 close(handle);
 free(ptr);
 delay(1000);
 restorecrtmode(); /* 恢复原视屏模式 */
}
```

}

/\* 如下程序所示,也可以用流来管理文件,但有些不同。内中变量 size1 的应用 是必须的,即读写应是 512 字节的整数倍,否则会有问题的。

C>TYPE SCR27.C

```
#include "stdio.h"
#include "graphics.h"
#include "fcntl.h"
#include "stdlib.h"
main()
{
 int d=VGA,m=VGAHI,
 void *ptr;
 size_t size,size1;
 int x1=30,y1=40,x2=70,y2=80;
 int handle;
 FILE *fp;
 initgraph(&d,&m,"");
 setcolor(RED);
 rectangle(x1,y1,x2,y2);
 size=imagesize(x1,y1,x2,y2);
 if(size==0xFFFF)exit(2);
 size1=0;
 if(size%512 !=0)
 size1=512;
 size=size/512 * 512+size1;
 ptr=malloc(size);
 getimage(x1,y1,x2,y2,ptr);
 fp=fopen("a","wb");
 if(fp==NULL)exit(1);
 fwrite(ptr,1,size,fp);
 fclose(fp);
 free(ptr);
 delay(1000);
 clearviewport();
 delay(500);
 fp=fopen("a","rb");
 if(fp==NULL)exit(1);
 handle=fileno(fp);
 size=filelength(handle);
 ptr=malloc(size);
 fread(ptr,1,size,fp);
 putimage(x1,y1,ptr,COPY_PUT);
 close(handle);
 free(ptr);
 delay(1000);
 restorecrtmode();
}
```



```
} */
```

—10 unsigned far —Cdecl getpixel(int x, int y);

取得指定视口内一点 (x,y) 处的像素值。坐标指视口坐标。像素值与图形颜色是密切相关的,但它们对不同的视频系统有不同的关系。枚举 CGA—COLORS 中等号右边的数值便是像素值。从中可见像素值还跟调色板相关。

```
C>TYPE SCR28.C
#include <graphics.h>
main()
{
int graphdriver=CGA,graphmode=0; /* 用 CGA 驱动程序 */
int i,color1,color2,color3;
int r[]={0,0,10,0,10,10,0,10,0,0}; /* 画正方形的数据 */
initgraph(&graphdriver,&graphmode,"");
setbkcolor(1); /* 设置背景 */
setcolor(2); /* 设置当前的绘画色 */
fillpoly(4,r); /* 画一正方形 */
color1=getpixel(10,10); /* 取得正方形右下角点的像素值 */
setviewport(28,10,62,20,1); /* 设置新视口。屏幕绝对坐标 */
fillpoly(4,r); /* 在新视口内又画一正方形 */
color2=getpixel(10,10);
outtext("pp");
putpixel(10,10,1); /* 对新视口右下角点画另一色 */
outtext("m");
color3=getpixel(10,10);
getch();
printf("%d %d %d\n",color1,color2,color3);
getch(); /* 输出像素值比较 */
closegraph();
}
```

—11 void far —Cdecl putpixel(int x, int y, int color);

本函数按指定的像素值 color 在视口内 (x,y) 处画一点。

程序 SCR29.C 使我们看到在 CGA 屏幕上除了原有颜色的图形颜色不变外,在从左上角到右下角出现一根由四色点组成的直线。当坐标值无效时, getpixel() 取得值 0。

```
C>TYPE SCR29.C
#include <graphics.h>
main()
{
int graphdriver=CGA,graphmode=0; /* 选 CGA 驱动程序 */
int i,color,max,bkcolor;
initgraph(&graphdriver,&graphmode,"");
max=getmaxcolor()+1; /* 求颜色号数 */
setbkcolor(0); /* 设置背景色 */
printf("\tmaxcolor=%d\n",max);
filfillipse(10,10,8,6); /* 画椭圆 */
}
```

```

bkcolor=getbkcolor(); /* 取背景色 */
for(i=1;i<200;i++){
color=getpixel(i,i); /* 画对角线 */
if(color%max==bkcolor)putpixel(i,i,(color+i)%max);
else putpixel(i,i,color%max);
}
getch();
color=10;setbkcolor(1);bkcolor=getbkcolor();
color=getpixel(850,1); /* 检查坐标无效时情况 */
printf("bkcolor=%d color=%d\n",bkcolor,color);
getch();
closegraph();
}

```

### 35.6.6 颜色控制

#### 1. 像素与调色板

图形屏幕可以看作是由一个像素矩阵组成的。每个像素就是图形屏幕上的一点。说显示器为  $640 \times 200$ ，是说它在水平方向（一行）有 640 个像素，垂直方向（一列）有 200 个像素，因此整个屏幕可显示 128000 ( $640 \times 200$ ) 个点。每个像素可以有颜色，像素也有自己的值，但这个值并不直接和显示的颜色对应，而是对被称为调色板（palette）颜色表的一个索引。在枚举 `graphics—modes` 中可以看出 CGA、MCGA 和 ATT400 都有四个调色板（0 ~ 3）。每个调色板的项与像素值具有约定的对应关系。函数 `putpixel()` 中的参数 `color` 便是指像素值。

在 `graphics.h` 中枚举 `CGA—COLORS` 列出了 CGA 调色板与像素值及对应颜色的关系。像素值为 0 时对应颜色便是背景色。

```

enum CGA—COLORS { /* 等号左边为像素颜色，右边为像素值 */
 CGA—LIGHTGREEN = 1, /* 调色板 0 颜色表 */
 CGA—LIGHTRED = 2,
 CGA—YELLOW = 3,
 CGA—LIGHTCYAN = 1, /* 调色板 1 颜色表 */
 CGA—LIGHTMAGENTA = 2,
 CGA—WHITE = 3,
 CGA—GREEN = 1, /* 调色板 2 颜色表 */
 CGA—RED = 2,
 CGA—BROWN = 3,
 CGA—CYAN = 1, /* 调色板 3 颜色表 */
 CGA—MAGENTA = 2,
 CGA—LIGHTGRAY = 3
};

```

四个调色板的项与给定的像素值对应，保存着像素的真实颜色。这种间接的结构有许多用途。尽管硬件可能能够做到显示许多种颜色，但任何时候只能显示这些颜色的某个子集。同时可显示的颜色数同调色板中的项数（【称为调色板的尺寸】）相等。例如，枚举 `EGA—COLORS` 列出了 EGA 的像素值与颜色的对应关系。

```

enum EGA—COLORS {
 EGA—BLACK = 0, /* 暗色 */

```

|                  |   |     |          |
|------------------|---|-----|----------|
| EGA-BLUE         | = | 1,  |          |
| EGA-GREEN        | = | 2,  |          |
| EGA-CYAN         | = | 3,  |          |
| EGA-RED          | = | 4,  |          |
| EGA-MAGENTA      | = | 5,  |          |
| EGA-BROWN        | = | 20, |          |
| EGA-LIGHTGRAY    | = | 7,  |          |
| EGA-DARKGRAY     | = | 56, | /* 亮色 */ |
| EGA-LIGHTBLUE    | = | 57, |          |
| EGA-LIGHTGREEN   | = | 58, |          |
| EGA-LIGHTCYAN    | = | 59, |          |
| EGA-LIGHTRED     | = | 60, |          |
| EGA-LIGHTMAGENTA | = | 61, |          |
| EGA-YELLOW       | = | 62, |          |
| EGA-WHITE        | = | 63  |          |

};

同一时刻只有 16 种颜色显示,但事实上硬件可以显示的颜色多达 64 种。EGA 调色板的大小 (size) 为 16,该尺寸决定了象素值的范围,即 0 ~ (size-1)。函数 getmaxcolor() 将最大的有效象素值 (size-1) 返回给当前的图形驱动程序及显示模式。

对 VGA 有些情况完全同 EGA,但也有许多不同。

从上可见,通过改变调色板而不改变象素值也能改变显示的色彩。调色板和象素值的组合决定当前象素的显示颜色。因此,象素值是调色板的一个索引值,或即等于调色板寄存器号。

## 2. 库函数

### 一 取颜色信息

- . 获得当前图形屏幕的背景颜色 —1 getbkcolor()
- . 获得当前画图颜色 —2 getcolor()
- . 获得当前驱动程序的调色板结构 —3 getdefaultpalette()
- . 获得当前图形驱动程序和显示模式下最大的有效颜色值 —4 getmaxcolor()
- . 获得当前调色板的尺寸和颜色信息 —5 getpalette()
- . 获得当前图形驱动程序和显示模式允许的调色板入口数 —6 getpalettesize()

### 二 设置颜色

- . 一次将所有的入口对应的颜色全改变成各自指定的值 —7 setallpalette()
- . 改变调色板的各入口项对应的颜色 —8 setpalette()
- . 设置图形屏幕的背景颜色 —9 setbkcolor()
- . 设置当前的画图颜色 —10 setcolor()
- . 定义 IBM-8514 图形卡的颜色 —11 setrgbpalette()

—1 int far —Cdecl getbkcolor(void);

本函数返回当前图形屏幕的背景颜色。返回的整数值是 graphics.h 中的枚举 COLORS 中的值。例如, BLACK 颜色为 0, BLUE 为 1 等等。

```
enum COLORS {
BLACK, /* dark colors */
BLUE,
GREEN,
CYAN,
RED,
MAGENTA,
BROWN,
LIGHTGRAY,
DARKGRAY, /* light colors */
LIGHTBLUE,
LIGHTGREEN,
LIGHTCYAN,
LIGHTRED,
LIGHTMAGENTA,
YELLOW,
WHITE
};
```

—2 int far —Cdecl getcolor(void);

\* 返回当前画线颜色。画线颜色是指在画线或其它图形时象素被设置的值。例如,对 CGA 调色板有 4 种颜色:背景色、淡绿色、淡红色和黄色。如 getcolor() 返回 1,则表示当前画线颜色为淡绿色。

—3 struct palettetype \* far —Cdecl getdefaultpalette( void );

本函数返回一远指针,在调用 initgraph() 函数初始化图形系统后,它指向当前图形驱动程序缺省的调色板结构。结构 palettetype 在 graphics.h 中定义为:

```
#define MAXCOLORS 15
struct palettetype {
 unsigned char size; /* 给出当前图形驱动程序和当前 */
 /* 前模式下调色板的颜色数目 */
 signed char colors[MAXCOLORS+1];
}; /* colors 是一个具有 size 字节的数组,每个元素(占一字节)中 */
 /* 存放对应于调色板入口的实际的原始颜色编号。具体为
```

表 35-15

| CGA                |              |       | EGA               |           |                   |
|--------------------|--------------|-------|-------------------|-----------|-------------------|
| 入口编号<br>(colornum) | colors<br>元素 | 常量名   | 原始颜色编号<br>(color) | 常量名       | 原始颜色编号<br>(color) |
| 0                  | colors[0]    | BLACK | 0                 | EGA—BLACK | 0                 |
| 1                  | colors[1]    | BLUE  | 1                 | EGA—BLUE  | 1                 |
| 2                  | colors[2]    | GREEN | 2                 | EGA—GREEN | 2                 |
| 3                  | colors[3]    | CYAN  | 3                 | EGA—CYAN  | 3                 |
| 4                  | colors[4]    | RED   | 4                 | EGA—RED   | 4                 |

|    |            |              |    |                         |    |
|----|------------|--------------|----|-------------------------|----|
| 5  | colors[5]  | MAGENTA      | 5  | EGA—MAGENTA             | 5  |
| 6  | colors[6]  | BROWN        | 6  | EGA—BROWN               | 20 |
| 7  | colors[7]  | LIGHTGRAY    | 7  | EGA—LIGHTGRAY           | 7  |
| 8  | colors[8]  | DARKGRAY     | 8  | EGA—DARKGRAY            | 56 |
| 9  | colors[9]  | LIGHTBLUE    | 9  | EGA—LIGHTBLUE           | 57 |
| 10 | colors[10] | LIGHTGREEN   | 10 | EGA—LIGHTGREEN          | 58 |
| 11 | colors[11] | LIGHTCYAN    | 11 | EGA—LIGHTCYAN           | 59 |
| 12 | colors[12] | LIGHTRED     | 12 | EGA—LIGHTRED            | 60 |
| 13 | colors[13] | LIGHTMAGENTA | 13 | EGA — LIGHTMA-<br>GENTA | 61 |
| 14 | colors[14] | YELLOW       | 14 | EGA—YELLOW              | 62 |
| 15 | colors[15] | WHITE        | 15 | EGA—WHITE               | 63 |

注意：有效颜色依赖于当前图形驱动程序和当前的显示模式。调色板上的变化可以立即从屏幕上看到，屏幕上所有出现该颜色的象素均改为新颜色。程序 SCR30.C 使你理解这一点 \*/

```

C>TYPE SCR30.C
#include <graphics.h>
#include "stdlib.h"
#define KB if(palette->size>1){
 do
 if(graphdriver==CGA)setpalette(0,RED);
 /* 对 CGA,setpalette() 只能改变调色板的第一个入口项,即背景色 */
 else {j0=random(palette->size);j1=random(palette->size);
 printf("j0=%d j1=%d\n",j0,j1);setpalette(j0,j1);}
 while (!kbhit());/* 在 EGA 下,看到屏幕不断改变颜色 */
 getch(); /* 连续显示中按任一键显示结束 */
 };
#define PP getpalette(&p);palette=&p; \
 printf("Palette->size=%d p.size=%d\n",palette->size,p.size);\
 for(j=0;j<p.size;j++)printf("j=%d %d\n",j,p.colors[j])
#define KP setallpalette(palette);circle(120,120,10);getch(); \
 KB,PP;getch()
 /* 如将 setallpalette(palette); 放到 KB; 或PP; 之后效果会不同 */
main()
{
 int graphdriver=DETECT,graphmode,j,j0,j1;
 struct palettetype p,*palette;
 initgraph(&graphdriver,&graphmode,"");
 palette=getdefaultpalette(); /* 取得初始化时调色板结构 */
 printf("palette adress=%Fp\n",palette);/* 输出指针值: 0BC1:014E */
 PP; /* 输出结构 palettetype 中域的值.见枚举 CGA—COLORS 或 EGA—COLORS */
 /* 对 CGA,palette->size=0; 对 EGA 等,palette->size=16 */
 circle(100,100,10);
 printf("drawing color=%d\n",getcolor());/* 输出前画图颜色号 =1 */
 getch();

```

```

KP;
setcolor(1); /* CGA 设置当前画图色 */
circle(120,120,10);
getch();
palette->size=4; /* 只对 EGA,VGA 等起作用 */
KP;
closegraph();
}

```

在使用程序 SCR30.C 时应注意到两个问题,一是用单步调试时用户屏幕颜色可能非实际颜色,它可能会受到 TC 的影响。因此,最终应脱离 TC 单独执行观察;其次,用与不用宏定义 KB 对用 F7 单步跟踪的效果是不一样的。使用了宏定义, F7 跟踪将不能跟踪宏定义中的语句,因此,必要时你可不用宏。

—4 int far —Cdecl getmaxcolor(void);

返回当前图形驱动程序和显示模式下最大的有效颜色值(等于调色板尺寸-1)。

—5 void far —Cdecl getpalette(struct palettetype far \*palette);

本函数将有关当前调色板的尺寸和颜色信息填入由 palette 指针所指的 palettetype 型的结构中。

—6 int far —Cdecl getpalettesize(void);

返回当前图形驱动程序和显示模式允许的调色板入口数。

—7 void far —Cdecl setallpalette(struct palettetype far \*palette);

如果事先改变了指针 palette 所指的结构的域 palette->colors[i] 中的值,并且改变了多个 colors 的多个元素,则调用本函数一次便可将所有的入口对应的颜色全改变成各自指定的值。

如果 colors 的某一个元素的值是 -1,则那一个入口项的调色板的颜色将不改变。

—8 void far —Cdecl setpalette(int colornum, int color);

改变一个调色板颜色。函数中 colornum 是调色板入口编号, color 是与 colornum 对应的编号颜色改变后的颜色。如果 size 是当前调色板的入口项数,则 colornum 的范围为 0 ~ size-1。在 CGA 中,由于显示模式预定义了调色板,例如模式 CGAC0 定义了 0 号调色板,模式 CGAC1 定义了 1 号调色板等等,所以只能调用 setpalette() 改变调色板 0 号入口,即用

```
setpalette(0,RED); /* 将背景设成红色 */
```

那样的语句改变背景色,对调色板的其它入口不起作用。对 EGA 等,可用本函数分别改变调色板的各入口项对应的颜色。例如, setpalette(0,5); 将当前调色板中的第一种颜色(背景色)改为实际颜色数 5。注意,有效颜色数依赖于当前图形驱动程序和当前图形模式。调色板的变化可以在屏幕上看到。每当一个调色板颜色改变时,屏幕上所有出现该颜色的象素均为新颜色。

当无效值作为函数参数时, graphresult() 函数返回 -11,且当前调色板不变。

—9 void far —Cdecl setbkcolor(int color);

本函数将背景设计成参数 color 所指定的颜色。color 参见枚举 COLORS 的内容。如想把背景设成兰色,可用 setbkcolor(1); 或 setbkcolor(BLUE); 都可以。参数 color 指出了当前调色板的入口项,可通过改变调色板的第一个入口点来改变背景色。

本函数对 CGA 和 EGA 是不同的。在 EGA 上, setbkcolor() 将存在第 color 项中的有效颜色值拷贝到第 0 项(原为 BLACK)。

—10 void far —Cdecl setcolor(int color);

函数将当前的画线颜色置为 color, 其取值范围为 0 ~ getmaxcolor()。例如, 对 CGA 显示器, setcolor(3) 选黄色为画线颜色。

—11 void far —Cdecl setrgbpalette(int colorm, int red, int green, int blue);

定义 IBM-8514 图形卡的颜色。这里 colorm 是待加载的调色板入口, 范围为 0 ~ 255。其它三个参数 red(红)、green(绿)和 blue(蓝)用来定义分量的颜色。这些值中, 只有低字节的 6 个有效位被装入调色板。

在高分辨率 (640 × 200) 下, CGA 只显示两色: 黑色的背景和有颜色的前景。像素值可为 1 或 0, 它由彩色选择寄存器低 4 位决定的 16 种颜色之一。值得指出, 这里 CGA 把其前景色作为硬件认为的背景色来处理的, 因此你应该用 setbkcolor() 函数来设置前景, 而不是用 setcolor() 函数, 该函数不能起改变前景的作用 (当然也不能改变背景)。前景色可以是枚举 COLORS 中的 16 种颜色, CGA 显示的像素值为 1。注意, 每一种颜色均起作用! 或者说, 不可能出现前景和背景相同的颜色, 即不会出现图象不能显示的情况 (注意, 对有些 CGA 显示器可能有例外)。

类似这种情况的还有 MCGAMED、MCGAHI、ATT400MED 和 ATT400HI 等。

```
C>TYPE SCR31.C
#include <graphics.h>
#include "stdlib.h"
#define PR(X) p=getpixel(X,X);printf("pixel=%d\n",p); \
printf("drawing color=%d\n",getcolor()); \
printf("getbkcolor=%d\n",getbkcolor());getch()
/* 获取圆心处像素值,打印画图色和背景色 */

main()
{
int graphdriver=CGA,graphmode=CGAHI; /* 在 CGA 高分辨率情况下 */
int p,i;
struct palettetype *palette;
initgraph(&graphdriver,&graphmode,"");
printf("getpalettesize=%d\n",getpalettesize());
fillellipse(100,100,10,10); /* 画一个填充颜色的圆 */
PR(100);
for(i=0;i<17;i++)
{
/* 当 i=16 时获得和 i=15 时同样的结果 */
setbkcolor(i);
fillellipse(120,120,10,10);
PR(120); /* 单步调试按 F7 键为偶数次时, getch() 才让屏幕转为用户屏幕 */
}
closegraph();
}
```

### 35.6.7 图形方式下的正文输出

在 TC 提供的 BGIDEMO.C 中, 用户能够看到如何使用图形模式下的正文输出函数的例

子。TC 提供的 5 种字体中,由于缺省的位图字体是由图形系统决定的,因此每个位图字体都由一个像素矩阵定义;其余 4 种字体则在 TC 磁盘上的字体文件(\*.CHR)中,而且字体的每个字符是由一系列矢量(或向量)来定义的,所以又称为【矢量字体】。几种矢量字体都可以在一定范围内进行放大,但是它们之间是有区别的。放大一个位图字体时,字体矩阵将与一个纵横放大率相乘,当这个放大率较大时,字体的密度就变疏了。所以 TC 不允许对位图字体放大。对于小一点的字体,位图字体可以满足要求的,但对较大字体,使用矢量字体则较好。因为矢量字体是用矢量来定义的,当字体放大时,它仍能保持很好的分辨率和质量。此外,有时对美观性有要求时,矢量字体有时也能帮忙。

- . 得到当前图形驱动程序和显示模式下的正文字体、方向—1 gettextsettings()  
大小和图形当前位置(CP)在水平方向(horiz)与垂直方向上的对齐方法的信息
- . 设置文本字体、字符串显示方向和放大因子 —2 settextstyle()
- . 设置矢量字体的大小 —3 setusercharsize()
- . 获得以像素为单位的字符串高度 —4 textheight()
- . 获得以像素为单位的字符串的宽度 —5 textwidth()
- . 在当前视区中使用当前字体、当前字符串显示方向、—6 outtext()  
当前放大因子和当前字符对齐方法来显示串
- . 在给定位置(x,y)输出串 —7 outtextxy()
- . 设置输出的字符串与 CP 对准方式 —8 settextjustify()
- . 注册字体文件 —9 registerbfont()  
—10 registerfarbfont()

```
—1 void far —Cdecl gettextsettings(struct textsettingstype
 far *texttypeinfo);
```

将当前图形驱动程序和显示模式下的正文字体(font)、方向(direction)、大小(charsize)、和图形当前位置(CP)在水平方向(horiz)与垂直方向(vert)上的对齐方法的信息填入由 texttypeinfo 指针所指的 textsettingstype 型的结构中。graphics.h 中定义了结构 textsettingstype:

```
struct textsettingstype {
 int font;
 int direction;
 int charsize;
 int horiz;
 int vert;
};
```

枚举 font—names 定义了几种(西文)字体(font)可选值。

```
enum font—names {
 DEFAULT—FONT = 0, /* 8x8 位图字体 */
 TRIPLEX—FONT = 1, /* 三重矢量(Stroked)字体 */
 SMALL—FONT = 2, /* 小号矢量字体 */
 SANS—SERIF—FONT = 3, /* 无衬线(光滑不修饰)矢量字体 */
 GOTHIC—FONT = 4 /* 哥特矢量字体 */
};
```



};

符号常量 `HORIZ-DIR` 和 `VERT-DIR` 定义了两种字体显示方向 (direction):

```
#define HORIZ-DIR 0 /* 水平文本是从左到右 */
#define VERT-DIR 1 /* 垂直文本是自底向上 */
```

用 `charsize` 因子可以将每个字符的大小放大。如果它非 0, 则能影响这 5 种字体, 否则只影响矢量字体。

(1) 当 `charsize=1` 时, `outtext()` 和 `outtextxy()` 将把  $8 \times 8$  位图字体的字符在屏幕上显示为  $8 \times 8$  像素矩阵。

(2) 当 `charsize=2` 时, `outtext()` 和 `outtextxy()` 将把  $8 \times 8$  位图字体的字符在屏幕上显示为  $16 \times 16$  像素矩阵。

(3) 一般地, 当 `charsize=k` 时, 且  $k$  为  $1 \sim 10$  之间的整数, 则上述两函数将使字体显示为  $8k \times 8k$  的像素矩阵。

(4) 当 `charsize=0` 时, `outtext()` 和 `outtextxy()` 用缺省的字符放大因子 (等于 4) 或者用 `setusercharsize()` 给出的用户自定义字符大小来放大矢量字体。

枚举 `text-just` 定义了 `horiz` 和 `vert` 可选值, 它们决定当前显示的字符串和图形当前位置 (CP) 的对准方法。

```
enum text-just {
 LEFT-TEXT = 0, /* 用于 horiz 选择 */
 CENTER-TEXT = 1,
 RIGHT-TEXT = 2,
 BOTTOM-TEXT = 0, /* 用于 vert 选择 */
 /* CENTER-TEXT = 1, 可用 1, 前面已有定义 */
 TOP-TEXT = 2
};
```

程序 `SCR32.C` 能帮助你理解参数 `horiz` 和 `vert` 选取的符号常量的意思。它在视口中先画出一个矩形, 然后显示的字符以这个矩形的左上角为基准, 观察字符串和矩形的相对位置。屏幕右上角显示当前的字体、方向、放大因子、水平和垂直方向对齐模式值。

**LEFT-TEXT:** 在水平方向字符串左边第一个字符的最左边和 CP (图形当前位置) 对准;

**CENTER-TEXT:** 对水平方向, 在字符串的长度为串长一半处和 CP 对准;  
对垂直方向, 在字符串的高度一半处和 CP 对准;

**RIGHT-TEXT:** 在水平方向字符串最后一个字符的最右边和 CP 对准;

**BOTTOM-TEXT:** 在垂直方向, 字符串的底端和 CP 对准;

**TOP-TEXT:** 在垂直方向, 字符串的顶端和 CP 对准。

C>TYPE SCR32.C

```
#include "graphics.h"
#define CL getch(), cleardevice(), setviewport(60, 60, 230, 130, 0)
 /* 注意: setviewport() 中的剪切因子被定为 0, 即不剪切视口外的图形 */
 /* 若将剪切因子改为 1, 则运行程序后你会发现: 缺省的位图字体在输出 */
 /* 时并不受到剪切, 也即字体可在视口外显示, 而另一字体即哥特矢量 */
 /* 字体将受到剪切, 视口外的字体将不可见 */
#define TEXT(FONT, DIRECTION, CHARSIZE, HORIZ, VERT) rectangle(0, 0, 139, 99); \
```

```

/* 矩形坐标是对当前视口的相对坐标,outtextxy() 的坐标也是这样 */
settextstyle(FONT,DIRECTION,CHARSIZE);\
settextjustify(HORIZ,VERT);outtextxy(0,0,"Text")
main()
{
int oldx,oldy;
int graphdriver=CGA,graphmode=CGAHI;
int f,font[]={DEFAULT-FONT,GOTHIC-FONT};
int d,dire[]={HORIZ-DIR,VERT-DIR};
int c,chars[]={0,1};
int h,horiz[]={LEFT-TEXT,CENTER-TEXT,RIGHT-TEXT};
int v,vert[]={BOTTOM-TEXT,CENTER-TEXT,TOP-TEXT};
struct textsettingstype oldtext;
initgraph(&graphdriver,&graphmode,"");
gettextsettings(&oldtext); /* 可用调试表达式 oldtext.r 观察结构缺省值 */
setviewport(60,60,230,130,0); /* 设视口,坐标以屏幕分辨率为准的绝对坐标 */
for(f=0;f<2;f++) /* 缺省(8×8 位图)字体和哥特矢量字体 */
{
for(d=0;d<2;d++) /* 两种字体方向 */
{
for(c=0;c<2;c++) /* 放大因子 */
{
for(h=0;h<3;h++) /* 水平对准方法 */
{
for(v=0;v<3;v++) /* 垂直对准方法 */
{
TEXT(font[f],dire[d],chars[c],horiz[h],vert[v]);
gotoxy(70,1);
/* 文本方式下用的函数,指向屏幕右上角,屏幕 C80 绝对坐标 */
printf("%d %d %d %d %d\n",font[f],dire[d],chars[c],horiz[h],vert[v]);
CL; /* 清屏幕,重设相同视口,按键继续 */
}
}
}
}
}
}
closegraph(); /* 关闭图形系统 */
}

```

用程序 SCR33.C 可输出当前字体的缺省情况。

```

C>TYPE SCR33.C
#include "graphics.h"
#define PP printf
main()
{
int graphdriver=CGA,graphmode=0;
struct textsettingstype initext;

```

```

initgraph(&graphdriver,&graphmode,"");
gettextsettings(&initext);
PP(*font=%d direction=%d\n",initext.font,initext.direction);
PP(*charsize=%d\n",initext.charsize);
PP(*horiz=%d vert=%d\n",initext.horiz,initext.vert);
getch();
closegraph();
}/* 在 CGA 上,缺省的输出
 font=0 direction=0 8×8 位图字体方向从左到右
 charsize=1 8×8 像素矩阵
 horiz=0 vert=2 水平方向上与 CP 左对齐,垂直方向与 CP 顶对齐
*/

```

—2 void far —Cdecl settextstyle(int font, int direction, int charsize);

对当前视口设置文本字体 (font)、字符串显示方向 (direction) 和放大因子 (charsize)。本函数被调用后,将影响 outtext() 和 outtextxy() 函数的输出结果。

一般情况下,本函数通过为字体分配内存来装载一个字体文件,然后从磁盘上读入合适的 \*.CHR 字体文件。如果发生了错误,graphresult() 将返回到下列值之一:

- 8 未找到字体文件
- 9 没有足够的内存装入字体文件
- 11 图形错误
- 12 图形I/O 错误
- 13 无效字体文件
- 14 无效字体号

—3 void far —Cdecl setusercharsize(int multx, int divx,  
int multy, int divy);

本函数由用户控制矢量字体的大小。在调用本函数前一定要先调用 settextstyle(), 并使该函数的参数 charsize=0, 否则调用本函数无效。为此, graphics.h 中专门定义了一个符号常量:

```
#define USER—CHAR—SIZE 0
```

就是在这种情况下发生时用于 settextstyle() 函数的。

四个参数用于指定字体宽度的比例因子 (multx/divx) 和高度的比例因子 (multy/divy), 实际字体大小等于字体的缺省值乘以比例因子。

用单步调试程序 SCR34.C 可以使你对 setusercharsize() 函数有更多的了解。从中可以发现,字符串在宽度方向所占的像素并不一定和串中的字符数成比例,对矢量字体放大时应程序进行足够的调试,以确切把握实际放大情况。另外,函数在程序中起连续性的,即其设置的值虽然对位图字体是无效的,但是对随后的语句依然会起作用的。

```

C>TYPE SCR34.C
#include "graphics.h"
#define CPXY tw=textwidth(title);th=textheight(title);\
 cpx=getx();cpy=gety()

main()
{

```

```

int graphdriver=CGA,graphmode=0;
char * title="TEXT int a BOX";
int cpx=1,cpy=1;
int tw=1,th=1;
initgraph(&graphdriver,&graphmode,"");
settextjustify(CENTER—TEXT,CENTER—TEXT);
setusercharsize(1,1,1,1);
outtextxy(10,5,title);
CPXY; /* 单步调试中可用 cpx,cpy,tw,th 调试表达式监视 */
setusercharsize(1,1,1,1);
CPXY;
outtextxy(10,5,title); /* 本语句显示结果与前一显示语句相同 */
CPXY;
clearviewport(); /* 可在此句用 Break/watch/Toggle breakpoint Ctrl—F8 */
/* 设置断点.当不需调试前面语句时用 Ctrl—F9一次执 */
/* 行到本句,再用 F7 或 F8 键单步调试 */
setusercharsize(1,1,1,1);
outtextxy(10,5,title);
CPXY; /* 如将下句参数改成 1,1,1,1 看看有怎样结果 */
setusercharsize(4,1,8,1); /* 因这句不同,两次显示的字符将会不重合 */
CPXY; /* 但不会影响位图字体的大小 */
outtextxy(10,5,title);
CPXY; /* cpx=0 cpy=0 tw=112 th=8 */
rectangle(0,0,200,100); /* 画出一矩形框 */
settextstyle(TRIPLEX—FONT,HORIZ—DIR,USER—CHAR—SIZE);
/* 前面的 setusercharsize() 会影响本语句,即不再选取指定字体的缺省值 */
/* 究竟当前串的高和宽应实际测定.如取消上句,下面的显示会有问题 */
CPXY; /* cpx=0 cpy=0 tw=964(=4x241) th=224(=8x28) */
setusercharsize(200,textwidth(title),100,textheight(title));
CPXY;
outtextxy(100,50,title);
getch();
closegraph();
printf("textwidth=%d height=%d CPx=%d y=%d\n",tw,th,cpx,cpy);
}

```

—4 int far —Cdecl textheight(char far \*textstring);

返回以象素为单位的字符串高度,该高度实际是字符串中任一字符的高度。它只与字体种类及放大因子有关。特别地,当串中只有一个字符(对字母大小写均可)时可检查一个字符所占的象素值。

—5 int far —Cdecl textwidth(char far \*textstring);

它除了和字体种类、放大因子有关外,还与字符串长度相关。它返回以象素为单位的字符串 \*textstring 的宽度。对一个字符测得的宽度,包括了字符间的间隔。

textheight() 和 textwidth() 函数对于调整两行之间的距离大小,计算视区的高度和宽度,确定一个标题尺寸放在图形和方框中的位置是极有用的。这样既省去了对字符显示位置的具体估算,又可以在选择了不同字体时甚至也可不修改源代码,因为各图形文本之间仍可保持

正确的相对位置,而不会发生问题。

从程序 SCR35.C 输出可以看出字体放大尺寸与 charsize 参数的关系。

```
C>TYPE SCR35.C
#include "graphics.h"
char * Fonts[] = {"DefaultFont", "TriplexFont", "SmallFont",
 "SansSerifFont", "GothicFont"};
char * TextDirect[] = {"HorizDir", "VertDir"};
char * HorizJust[] = {"LeftText", "CenterText", "RightText"};
char * VertJust[] = {"BottomText", "CenterText", "TopText"};
void changetextstyle(int font, int direction, int charsize)
{
 settxtstyle(font, direction, charsize);
 if(graphresult() != grOk)
 {
 closegraph();
 printf(" Graphics System Error: %s\n"),exit(1);
 }
}

void TextDemo(void)
{
 int font,direction,charsize,horiz,vert;
 struct viewporttype vp;
 getviewsettings(&vp);printf("%d %d %d %d %d\n",vp.left,\
 vp.top,vp.right,vp.bottom,vp.clip);getch();clearviewport();
 /* for(font=0; font<2; ++font) /* 容易用此段程序证明,字体高与 */
 {
 /* 宽只同字体和 charsize 相关 */
 for(direction=0;direction<2;direction++)
 {
 for(charsize =0; charsize<5; charsize++)
 {
 for(horiz=0;horiz<3;horiz++)
 {
 for(vert=0;vert<3;vert++)
 {
 getviewsettings(&vp);
 changetextstyle(font, direction, charsize);
 settxtjustify(horiz,vert);
 gotoxy(1,3);
 printf("%s(%d)\t%s(%d)\t%d\t%s(%d)\t%s(%d)\t%d\t%d\n",
 Fonts[font],font,TextDirect[direction],direction,
 charsize,HORIZJust[horiz],horiz,VERTJust[vert],vert,
 textwidth("M"),textheight("m"));
 }
 }
 }
 }
 }
}
```

```

for(font=0 , font<5 ; ++font)
{
 for(charsize =0; charsize<12; charsize++)
 {
 changetextstyle(font, HORIZ—DIR, charsize);
 printf(" %s(%d)%d :%d x %d\n",Fonts[font],font,charsize,
 textwidth("M"),textheight("m")); /* 检测一个字符 */
 } /* 用textwidth(),textheight()确定字符的实际大小 */
} /* 输出字符的宽×高 */
}

main(){
int GraphMode=CGAHI,GraphDriver =CGA,ErrorCode;
initgraph(&GraphDriver, &GraphMode, "");
ErrorCode = graphresult();
if(ErrorCode != grOk){
printf(" Graphics System Error: %s\n", grapherrormsg(ErrorCode));
exit(1);
}
TextDemo();
getch();
closegraph();
}

/* 输出结果与显示模式无关,与字符串显示方向也无关。
0 0 639 199 1 屏幕视口大小
DefaultFont(0) 0 :8 x 8 缺省位图字体
DefaultFont(0) 1 :8 x 8
DefaultFont(0) 2 :16 x 16
DefaultFont(0) 3 :24 x 24
DefaultFont(0) 4 :32 x 32
DefaultFont(0) 5 :40 x 40
DefaultFont(0) 6 :48 x 48
DefaultFont(0) 7 :56 x 56
DefaultFont(0) 8 :64 x 64
DefaultFont(0) 9 :72 x 72
DefaultFont(0) 10 :80 x 80
DefaultFont(0) 11 :88 x 88 charsize 超过 10 时起作用
TriplexFont(1) 0 :24 x 28 缺省三重矢量字体
TriplexFont(1) 1 :14 x 16
TriplexFont(1) 2 :16 x 18
TriplexFont(1) 3 :18 x 21
TriplexFont(1) 4 :24 x 28
TriplexFont(1) 5 :32 x 37
TriplexFont(1) 6 :40 x 46
TriplexFont(1) 7 :48 x 56
TriplexFont(1) 8 :60 x 70
TriplexFont(1) 9 :72 x 84

```

```

TriplexFont(1) 10,96 x 112
TriplexFont(1) 11,96 x 112 charsize 超过 10 时不起作用
SmallFont(2) 0,6 x 8 缺省小号矢量字体
SmallFont(2) 1,3 x 4
SmallFont(2) 2,4 x 5
SmallFont(2) 3,4 x 6
SmallFont(2) 4,6 x 8
SmallFont(2) 5,8 x 10
SmallFont(2) 6,10 x 13
SmallFont(2) 7,12 x 16
SmallFont(2) 8,15 x 20
SmallFont(2) 9,18 x 24
SmallFont(2) 10,24 x 32
SmallFont(2) 11,24 x 32 charsize 超过 10 时不起作用
SansSerifFont(3) 0,21 x 28 缺省无衬线矢量字体
SansSerifFont(3) 1,12 x 16
SansSerifFont(3) 2,14 x 18
SansSerifFont(3) 3,15 x 21
SansSerifFont(3) 4,21 x 28
SansSerifFont(3) 5,28 x 37
SansSerifFont(3) 6,35 x 46
SansSerifFont(3) 7,42 x 56
SansSerifFont(3) 8,52 x 70
SansSerifFont(3) 9,63 x 84
SansSerifFont(3) 10,84 x 112
SansSerifFont(3) 11,84 x 112 charsize 超过 10 时不起作用
GothicFont(4) 0,27 x 28 缺省哥特矢量字体
GothicFont(4) 1,16 x 16
GothicFont(4) 2,18 x 18
GothicFont(4) 3,20 x 21
GothicFont(4) 4,27 x 28
GothicFont(4) 5,36 x 37
GothicFont(4) 6,45 x 46
GothicFont(4) 7,54 x 56
GothicFont(4) 8,67 x 70
GothicFont(4) 9,81 x 84
GothicFont(4) 10,108 x 112
GothicFont(4) 11,108 x 112 charsize 超过 10 时不起作用
*/

```

—6 void far —Cdecl outtext(char far \*textstring);

在当前视区中使用当前字体、当前字符串显示方向、当前放大因子和当前字符对齐方法来显示串 textstring。函数在 CP 处开始输出串。如果文本显示的方向是 HORIZ—DIR, 且水平对齐方法是 LEFT—TEXT, 则当串输出后, CP 的 x 坐标将增大 (长度等于 textwidth(\*textsting)); 否则 CP 保持不变。将程序 SCR35.C 中 outtextxy() 改成 outtext() 后编译运行可以看出它将受到当前设置的对齐方法的影响。为了在使用几种字体时保持代码的兼容性, 可

用 `textwidth()` 和 `textheight()` 决定字符串的尺寸大小。

—7 void far —Cdecl outtextxy(int x, int y, char far \*textstring);

本函数和 `outtext()` 不同之处是在给定位置 (x,y) 输出串 textstring。

—8 void far —Cdecl setttextjustify(int horiz, int vert);

本函数调用后,随后的 `outtext()` 和 `outtextxy()` 输出的字符串将按指定的方式(水平方向由 horiz 参数决定,垂直方向由参数 vert 决定)与 CP 对准。确省的对齐方法是以 (LEFT—TEXT, TOP—TEXT) 与 CP 对齐,即字符串左边第一个字符的左顶上的象素与 CP 重合。

—9 int —Cdecl registerbgifont(void (\*font)(void));

—10 int far —Cdecl registerfarbgifont(void far \*font);

可用的 5 种字体中,缺省的位图字体 (DEFAULT—FONT) 存在于图形系统中,所以在运行程序中使用该字体时并不需要另外装入字体文件 (\*.CHR); 而对其余 4 种矢量字体则不同,使用时要么将和它对应的字体文件装入内存,因此在运行用户程序时该字体文件应在当前目录中(对 TRIPLEX—FONT 应有 TRIP.CHR, 对 SMALL—FONT 应有 LITT.CHR, 对 SANS—FONT 应有 SANS.CHR, 对 GOTHIC—FONT 应有 GOTH.CHR); 或者你也可以将字体文件预先装入你的执行文件中,而执行文件中应有注册字体文件的 `registerbgifont()` 或 `registerfarbgifont()` 函数调用。连接注册方法参见前面的叙述 (35.6.4 节)。

### 35.6.8 绘图与填充

运用 TC 的绘图函数,可以画出彩色的线段、弧、扇形、椭圆、圆、矩形、二维及三维的条形(直方)图、多边形,再用这些基本图形可以组合成更复杂(规则或不规则)的图形;可以用 11 种预定义的填充模式或自定义的填充模式及填充颜色来填充任何有界区域;也可以控制画线的类型(或称【线型】)、宽度和颜色;还可以控制图形的当前位置(简称【CP】, Current position)的定位。

#### 1. 角度

当画图涉及角度时,总是这样规定的:角的两边(始边和终边)都和笛卡儿坐标系 X 轴正向重合(或者说时钟的时针指向 3 点)时角度为 0 度;当角的始边和 X 轴正向重合,而另一边即终边和 Y 轴正向重合(或者说时钟的时针指向 12 点)时,便说角度为 90 度。如此等等。角度的度量单位为十进制的度。角的始边逆时针转向终边时角度为正,反之为负。角度可以用负值,但须将它转为正值计算画弧,方法是,负角值加上  $n * 360$ ,  $n=1,2,3,\dots$ ,  $n$  的值为计算式第一次出现正值时的值。例如,角  $-390$  度相当于角  $2 * 360 - 390 = 330$  度,此时  $n=2$ 。程序 SCR1.C 可以帮助你理解这一点。

使用画图函数有可能失败,可用 `graphresult()` 或 `grapherrormsg()` 函数检查。最常见的错误是扫描填充时内存不够。

```
C>TYPE SCR35.C
#include <graphics.h>
main()
{
 int g=driver,g=mode;
 detectgraph(&g=driver,&g=mode);
 initgraph(&g=driver,&g=mode,"");
 setgraphmode(2);
```



```

setaspectratio(1000,1000);
arc(150,150,0,-90,40);
arc(150,150,0,-390,30);
setcolor(1);
arc(150,150,0,90,40);
setcolor(2);
arc(150,150,270,360,40);
pieslice(100,100,0,134,20);
setfillstyle(LINE—FILL,LIGHTBLUE);
pieslice(100,100,135,225,20);
setfillstyle(INTERLEAVE—FILL,WHITE);
pieslice(102,102,225,360,20);
setviewport(10,10,300,180,1);
setfillstyle(WIDE—DOT—FILL,RED);
sector(80,40,0,-270,40,20);
setcolor(1);
setfillstyle(LTBKSLASH—FILL,RED);
sector(80,40,0,90,40,20);
getch();
closegraph();
)

```

## 2. 绘图库函数

注意:画线颜色和填充颜色、线的形式和线的模式(两者统称线的种类)、线的宽度以及图形填充模式、图形卡的显示的纵横比率等在这里是很重要的概念。

### 一 图形卡的纵横比

- 获得当前图形卡的纵横比率 —1 getaspectratio()
- 设置新的图形纵横比率 —2 setaspectratio()

### 二 坐标

- 获得最近画过的弧(或部分椭圆)后的坐标 —3 getarccoords()

### 三 线的种类和画线模式

- 获得当前线段的种类和宽度 —4 getlinesettings()
- 设置画线种类和宽度 —5 setlinestyle()
- 设置当前画线的输出模式(覆盖已有线或者异或运算) —6 setwritemode()

### 四 填充模式和填充颜色

- 获得当前使用的填充模式和填充颜色 —7 getfillsettings()
- 获得当前用户自定义填充模式 —8 getfillpattern()
- 设置填充模式和填充颜色 —9 setfillstyle()
- 设置自定义填充模式和填充颜色 —10 setfillpattern()

## 五 移动象点

- . 将 CP 移到指定点 (x,y) —11 moveto()
- . 将 CP(x0,y0) 移动一相对距离 (dx,dy) —12 moverel()

## 六 画直线段

- . 在指定两点间画一直线 —13 line()
- . 从 CP 位置 (x0,y0) 到指定点 (x,y) 间画一直线 —14 lineto()
- . 从 CP 位置 (x0,y0) 到与 CP 有相对距离的  
点 (x0+dx,y0+dy) 间画一直线 —15 linerel()

## 七 画圆弧、圆、椭圆、矩形和多边形

- . 画一条圆弧 —16 arc()
- . 画一个圆 —17 circle()
- . 画一个椭圆 —18 ellipse()
- . 画一个矩形 —19 rectangle()
- . 画一个多边形 —20 drawpoly()

## 八 画图形并填充

- . 画二维条形图 —21 bar()
- . 画三维条形图 —22 bar3d()
- . 画一椭圆并填充 —23 fillellipse()
- . 画扇形并填充 (馅饼图) —24 pieslice()
- . 画椭圆扇区并填充 —25 sector()
- . 画一多边形并填充 —26 fillpoly()

## 九 填充图形有界区域

- . 填充一块封闭区域 —27 floodfill()

—1 void far —Cdecl getaspectratio(int far \*xasp, int far \*yasp);

返回当前显示模式下的纵横比,该纵横比用来保证象 arc()、circle() 和 pieslice() 函数画出的圆不变形,即接近实际的圆。

\*yasp 是 y 方向的纵横比,一般值为 10000。\*xasp 为 x 方向纵横比,对 VGA 卡,由于其像素是“方形”,因而 \*xasp = \*yasp = 10000; 而对其它卡,一般 \*xasp < \*yasp, 例如,对 CGA 卡有 \*xasp = 4167; 对 EGA 卡, \*xasp = 7750。对具体的显示卡你可用本函数查询它们的值。对于同一显示卡的不同显示模式,虽然 \*xasp 一样,但显示效果并不一样,这时你可用 setaspectratio() 函数重新设置新的纵横比,以达到显示图形(象圆)基本不变形。

应当注意到 getaspectratio() 和 setaspectratio() 的参数类型的不同。

—2 void far —Cdecl setaspectratio(int xasp, int yasp);

设置图形新的纵横比。x(列)方向是 xasp, y(行)方向是 yasp。

—3 void far —Cdecl getarccoords(struct arccoordstype far \*arccoords);

函数获取最后一次调用 arc()、pieslice() 或 sector() 函数后的坐标,坐标填入 arccoords 所指 arccoordstype 类结构中。arccoordstype 结构在标头文件 graphics.h 中定义为

```

struct arccoordstype {
 int x, y; /* (x,y) 为 arc 等所在圆的圆心坐标 */
 int xstart, ystart, xend, yend; /* 弧的起点和终点坐标 */
}; /* 如果要在 arc 函数所画圆弧末尾接着画一条直线,可用结构中信息 */

```

—4 void far \_\_cdecl getlinesettings(struct linesettingstype far \*lineinfo);

将当前画线的线型、模式和宽度的信息存到由 lineinfo 所指的 linesettingstype 类结构中。  
结构 linesettingstype 在 graphics.h 中定义为

```

struct linesettingstype {
 int linestyle; /* 线形 */
 unsigned upattern; /* 线模式 */
 int thickness; /* 宽度 */
};

```

(1) 线形由 graphics.h 中的枚举 line—styles 预定义:

```

enum line—styles {
 SOLID—LINE = 0, 实线
 DOTTED—LINE = 1, 点划线
 CENTER—LINE = 2, 中心线
 DASHED—LINE = 3, 破折线
 USERBIT—LINE = 4, 用户自定义的线形
};

```

(2) 当线形取 USERBIT—LINE 或数字 4 时,选线模式参数 upattern 取的值便是线的实际形状。该参数值为 16 位,例如取 0xFFFF 时,每一位上均为 1,值为 1 的位对应的象素就用当前颜色画出,故当 upattern 取值 0xFFFF 时,画出的线相当于实线,而 upattern 取值 0X3333 时便相当于破折线。取不同的值可得不同的线形。

(3) 宽度

线宽只有两种形式,参数值可由枚举 line—widths 决定

```

enum line—widths {
 NORM—WIDTH = 1, 正常宽度,相当 1 个象素宽
 THICK—WIDTH = 3, 粗线宽度,相当 3 个象素宽
};

```

—5 void far \_\_cdecl setlinestyle(int linestyle, unsigned upattern,  
int thickness);

设置当前画线形式、线模式和画线宽度。三个参数的意义同 getlinestyle()。注意,只有当线形取 4 或 USERBIT—LINE 时所取的 upattern 值才起作用,否则该参数将被忽略,即不管你对它取什么值均不起作用。如果函数的参数无效,则 graphresult() 返回 -1,且保持当前线型不变。

—6 void far \_\_cdecl setwritemode(int mode);

根据设定的当前画线颜色及屏幕已有颜色决定最后向屏幕输出的线颜色,这就是设置【画线的输出模式】。模式只用两种: mode=0 和 mode=1。当模式为 0 时,新画的线将覆盖了屏幕上原有的线,这实际也是一般没有用本函数指明线输出时的缺省情况;当模式为 1 时,新线象素点与旧线象素点之间先进行“异或 (XOR)”运算后再往屏幕输出。

对象素的异或运算,可能将已有线消去;对同一线又进行一次异或运算,便可将它恢复。

这种特性对于动画设计是有用的。

```
C>TYPE SCR37.C
#include <graphics.h>
main()
{
 int g=driver,g=mode;
 detectgraph(&g=driver,&g=mode);
 initgraph(&g=driver,&g=mode,"");
 setgraphmode(1);
 setcolor(1);
 line(8,8,10,10);
 line(40,40,60,60);
 line(10,10,20,20);
 setwritemode(1);
 setcolor(1);
 line(10,10,30,30);
 setwritemode(0);
 setcolor(2);
 line(15,15,40,40);
 setfillstyle(2,1);
 fillellipse(88,38,10,10);
 setwritemode(1);
 setcolor(1);
 setfillstyle(3,2);
 fillellipse(88,38,20,20);
 setwritemode(0);
 setcolor(2);
 setfillstyle(4,3);
 fillellipse(88,38,30,30);
 getch();
 closegraph();
}
```

—7 void far —Cdecl getfillsettings(struct fillsettingstype far \* fillinfo);

函数将有关当前填充模式和填充颜色的信息填入由 fillinfo 所指的 fillsettings 型的结构中去。该类型结构在 graphics.h 中定义为：

```
struct fillsettingstype {
 int pattern; 当前填充模式
 int color; 当前填充颜色
};
```

如果返回的 pattern=12，表明使用了用户自定义的填充模式，否则参数 pattern 是用了枚举 fill-patterns 中的预定义模式。枚举 fill-patterns 在 graphics.h 中定义为：

```
enum fill-patterns {
 EMPTY-FILL, /* 值=0, 用背景色填充 */
 SOLID-FILL, /* 值=1, 用单色实线填充 */
};
```

```

LINE—FILL, /* 值=2, 用 --- 线填充 */
LTSLASH—FILL, /* 值=3, 用细的 /// 线填充 fill */
SLASH—FILL, /* 值=4, 用粗的 /// 线填充 */
BKSLASH—FILL, /* 值=5, 用粗的 \\\ 线填充 */
LTBKSLASH—FILL, /* 值=6, 用 \\\ 线填充 */
HATCH—FILL, /* 值=7, 用淡阴影线填充 */
XHATCH—FILL, /* 值=8, 用浓交叉阴影线填充 */
INTERLEAVE—FILL, /* 值=9, 用隔行的间隔线填充 */
WIDE—DOT—FILL, /* 值=10, 用稀疏空白点填充 */
CLOSE—DOT—FILL, /* 值=11, 用密集空白点填充 */
USER—FILL /* 值=12, 表示用户用了自定义填充模式 */
);

```

—8 void far \_\_cdecl getfillpattern(char far \* pattern);

把用户最近自定义的填充模式拷贝到由 pattern 所指的 8 字节存储区内。通常可定义一个 8 字节字符数组, 如定义

```
char c[8]={0xAA,0x55,0xAA,0x66,0x55,0x66,0xAA,0x55};
```

后让指针指向它。数组各元素便装着各字节值。每个字节与该模式下的 8 个像素相对应, 只要那一位上置 1, 则对应的像素便被显示出来。

—9 void far \_\_cdecl setfillstyle(int pattern, int color);

本函数用于设置当前填充模式 (pattern) 和填充颜色 (color)。但是应注意两点, 一是如果你想使用用户自定义的填充模式, 则你必须使用函数 setfillpattern() 设置, 而不是用本函数并将 pattern 取成 12 (或 USER—FILL); 其次, 除模式选用常量 EMPTY—FILL 时填充用背景色外, 其余的预定义填充模式均用选定的 color 颜色填充 (当然, 应注意对 CGA 选显模式为 CGAHI 时, 实际是背景起作用, 所以这里设置的 color 无效)。

如果函数使用了无效参数, graphresult() 返回 -11, 当前填充模式和填充颜色不变。

—10 void far \_\_cdecl setfillpattern(char far \* upattern, int color);

设置用户自定义的 8 × 8 填充模式和填充颜色。upattern 是一指向连续 8 字节的指针, 每个字节与该模式下的 8 个像素相对应, 只要那一位上置 1, 则对应的像素便被画出来。

本函数和 setfillstyle() 的不同之处只在设置填充模式上不同, 其余都是一样的。

—11 void far \_\_cdecl moveto(int x, int y);

将当前位置 (CP) 移到当前视区的点 (x,y) 处。

—12 void far \_\_cdecl moverel(int dx, int dy);

将当前位置 (CP) 移动一相对距离 (dx,dy)。设 CP 坐标为 (x0,y0), 则调用 moverel() 后 CP 移到 (x0+dx,y0+dy)。

—13 void far \_\_cdecl line(int x1, int y1, int x2, int y2);

用当前颜色、当前线型和宽度在指定两点 (x1,y1)、(x2,y2) 之间画一条直线。注意, 直线虽然画出, 但 CP 不会因此而改变其画线前的值。

—14 void far \_\_cdecl lineto(int x, int y);

用当前颜色、当前线型和宽度在指定两点 CP、(x,y) 之间画一条直线。线画出后, CP 便移到 (x,y) 处。

—15 void far \_\_cdecl linerel(int dx, int dy);

用当前颜色、当前线型和宽度在指定两点 CP(x0,y0)、(x0+dx,y0+dy) 之间画一条直

线。线画出后, CP 便移到  $(x_0+dx, y_0+dy)$  处。 $(dx, dy)$  可称为【CP 的增量】, 或称为与 CP 的相对距离。

—16 void far —Cdecl arc(int x, int y, int stangle, int endangle,  
int radius);

画圆弧函数, 这里圆弧的圆心坐标是  $(x, y)$ , 起始角是 stangle, 终止角是 endangle, 圆弧半径是 radius, 圆弧的颜色是当前颜色。

函数 arc( $x, y, 0, 360, radius$ ) 将画出一个整圆。

圆弧画出后, CP 值不会改变画圆弧前的原有值。

—17 void far —Cdecl circle(int x, int y, int radius);

画一圆,  $(x, y)$  为圆心, radius 为半径。圆的颜色是当前颜色。

圆画出后, CP 值不会改变画圆前的原有值。

—18 void far —Cdecl ellipse(int x, int y, int stangle, int endangle,  
int xradius, int yradius);

以  $(x, y)$  为中心、从起始角 stangle 逆时针转到终止角 endangle 画一(部分)椭圆, 椭圆的长短轴分别为 xradius 和 yradius。如 stangle=0, endangle=360, 则画出一个完整的椭圆。

画线颜色为当前颜色。椭圆画出后, CP 值不会改变画椭圆前的原有值。

—19 void far —Cdecl rectangle(int left, int top, int right, int bottom);

用当前线型、宽度和颜色画一矩形, 矩形左上角为 left(列)与 top(行), 右下角为 right(列)与 bottom(行)。当坐标无效时函数被忽略。

矩形画出后, CP 值不会改变画矩形前的原有值。

—20 void far —Cdecl drawpoly(int numpoints, int far \* polypoints);

用当前画线类型和颜色画一多边形, 其中 numpoints 为边数, polypoints 为指向一个整数序列, 该整数序列为多边形各顶点的坐标值  $(x, y)$ , 所以共有  $2 * numpoints$  个整数。这个整数序列常放在一个一维数组中。

画出的 numpoints 个边的多边形不一定是一个封闭的多边形(因此, 确切地说, 画出的只是可以构成一个多边形的边, 而不一定是多边形)。若要画一边数为 numpoints 的封闭的多边形, 则要  $2 * (numpoints + 1)$  个整数, 而且最后一个顶点的坐标要和第一个一样。

多边形画出后, CP 值不会改变画多边形前的原有值。

—21 void far —Cdecl bar(int left, int top, int right, int bottom);

画一长方形条形图(左顶点 left, top; 右下角 right, bottom), 并用当前填充模式和当前填充颜色将条形图填充。注意: 本函数并不画出条形图的轮廓线, 因此它并不等于通常所说的直方图(它有轮廓线)。要画直方图可用函数 bar3d(), 并使其参数 depth 等于 0。画边框可用 rectangle() 函数。

条形图画出后, CP 值不会改变画条形图前的原有值。

—22 void far —Cdecl bar3d(int left, int top, int right, int bottom,  
int depth, int topflag);

画出一个三维的有轮廓线的条形图(左顶点 left, top; 右下角 right, bottom), 并用当前颜色和当前填充模式填充其前面的矩形, 而其三维顶和右侧面均未被填充(要填充可以使用函数 floodfill())。其轮廓线用当前的线型和当前的颜色画出。图形画出后, CP 值不会改变画图前的原有值。

这种三维条形图实际是一个立方体图, 如果把屏幕最前面的矩形当作二维条形图(直方

图), 则向屏幕里边的厚度(实际按斜视画出)就可称为【条形深度】, 其值可用参数 depth 给出(单位: 像素。一个常用深度是取  $(right-left)/4$ ); 立方体的俯视图(从顶上往下看)便可称为【三维顶】。三维顶实际是一个菱形。要不要画出三维顶可由参数 topflag 决定。如其值非 0, 则放一个三维顶, 否则(为 0)便不放。当需要几个三维条形图重迭摆放时, 有时要求底下的三维图不放三维顶就变成可能。当然, 各三维图可用不同的填充色来区分。程序 SCR38.C 可以说明这一点。

```
C>TYPE SCR38.C
#include "graphics.h"
main()
int GraphMode=CGAHI, GraphDriver =CGA, ErrorCode;
int x,y,x4,y8;
initgraph(&GraphDriver, &GraphMode, "");
setgraphmode(2);
x=getmaxx(), y=getmaxy(); x4=x/4; y8=y/8;
printf("%d %d: %d %d\n", x,y, getx(), gety());
bar3d(x4,y8,x4+40,y8+40,10,1);
printf("%d %d\n", getx(), gety()); /* 比较画条形图前后的 CP 位置 */
setcolor(1);
bar3d(x4,y8+40,x4+40,y8+80,10,0);
setfillstyle(XHATCH-FILL,1);
bar3d(1,30,41,70,6,1);
setfillstyle(CLOSE-DOT-FILL,3);
bar3d(1,70,41,110,10,1); /* 有三维顶 */
setfillstyle(HATCH-FILL,2);
bar3d(50,130,90,120,8,0); /* 无三维顶 */
setfillstyle(WIDE-DOT-FILL,15);
bar3d(50,170,91,199,8,1);
setfillstyle(SLASH-FILL,3);
bar(120,120,130,140); /* 画二维条形图 */
setfillstyle(CLOSE-DOT-FILL,2);
bar(220,120,230,140);
closegraph();
}
```

—23 void far —Cdecl fillellipse( int x, int y, int xradius, int yradius );

以 (x,y) 为中心, xradius 与 yradius 为水平和垂直的半轴画一个完整的填充椭圆。椭圆用当前颜色和填充方式填充, 边线用当前线型和颜色。填充椭圆画出后, CP 值不会改变画填充椭圆前的原有值。

—24 void far —Cdecl pieslice(int x, int y, int stangle, int endangle,  
int radius);

用当前的画线颜色画出以 (x,y) 为中心、从起始角 stangle 转到终止角 endangle、radius 为半径的扇形外廓, 并用当前定义的填充模式和填充颜色填充扇形内部。画出的图形也称【馅饼图】。如果在填充扇形时发生错误, graphresult() 返回 -6。

扇形画出后, CP 值不会改变画扇形前的原有值。

—25 void far —Cdecl sector( int X, int Y, int StAngle, int EndAngle,

int XRadius, int YRadius );

画出以 (x,y) 为中心、StAngle 为起始角、EndAngle 为终止角、长短轴分别为 XRadius 和 YRadius 的椭圆扇区,并用当前定义的填充模式和填充颜色填充椭圆扇区内部。当前的填充模式和颜色可由函数 setfillstyle() 或 setfillpattern() 设置。当长短轴相等时,函数相当于 pieslice()。

若在画线或填充时出现错误,函数 graphresult() 返回 -6。

椭圆扇区画出后,CP 值不会改变画椭圆扇区前的原有值。

—26 void far —Cdecl fillpoly(int numpoints, int far \*polypoints);

用当前线型和颜色画一多边形外廓,然后用当前填充模式和填充颜色填充此多边形。如果在填充时发生错误,graphresult() 返回 -6。

多边形画出后,CP 值不会改变画多边形前的原有值。

—27 void far —Cdecl floodfill(int x, int y, int border);

函数用来填充一块封闭的区域。设 (x,y) 是该封闭区域内的一点,该区域的原有画线颜色(注意:决不是该区域已有的填充颜色!)是 border,则使用本函数后该区域便为当前填充模式和颜色填充。注意:如果你指定 (x,y) 所在区域的颜色 border 时和原有颜色不同,则可能整个视口(包括视口内所有封闭区域)为当前填充模式和颜色填充。当然,该域内原有的填充模式和填充颜色随之消失;或者你指定了一个封闭区域外的点 (x,y)(确切地说,该点不在任何封闭区域内),则所有封闭区域外的地方为当前填充模式和颜色所填充,而原有任一封闭区域内已有填充模式和颜色均不受影响。

若在填充过程中出现错误,graphresult() 返回 -7。如果可能,尽量用 fillpoly() 代替本函数。注意,本函数不能用于 IBM 8514 显示卡。

要填充圆、椭圆或多边形,尽量使用它们的画线且填充的函数。

因为对 CGA 的显示模式 CGAHI,背景和前景设定有特定的方法,因此上述结论稍要修整。程序 SCR39.C 在 CGA 的 CGAHI 显示模式下可以填充三维条形图的两个侧面,但对 EGA 而言,则应将个别语句修改一下,才能得到满意结果。

```
C>TYPE SCR39.C
#include <graphics.h>
main()
{
 int graphdriver=DETECT,graphmode,x1,y1,x2,y2,size;
 initgraph(&graphdriver,&graphmode,"");
 setviewport(1,10,300,150,1); /* 可以发现,在视区外的区域将不被填充 */
 setpalette(graphmode,1);
 size=getpalettesize();
 setcolor(GREEN); /* 决定前景 */
 setfillstyle(INTERLEAVE-FILL,LIGHTMAGENTA); /* 决定当前填充模式和颜色 */
 bar3d(10,10,100,100,10,1); /* 只用当前模式和颜色填充 */
 /* 条形图的矩形,三维顶和右侧面未能用色填充 */
 setfillstyle(XHATCH-FILL,LIGHTGREEN);
 x1=getx(),y1=gety(); /* 单步调试时可用它查询CP位置 */
 floodfill(102,50,WHITE); /* 对 EGA 此句应为 floodfill(102,50,GREEN); */
 x2=getx(),y2=gety();
 setfillstyle(LINE-FILL,LIGHTGREEN);
}
```



```

floodfill(150,98,GREEN); /*注意:指定点应落在希望的封闭区域内 */
getch();
closegraph();
printf("size=%d x1=%d y1=%d ; x2=%d y2=%d\n",size,x1,y1,x2,y2);
}

```

TC 在磁盘用演示程序 BGIDEMO.C 对画图库函数使用作了说明。程序 SCR40.C 则更进一步作了详细说明。对不同显示卡你可以稍作修改,或许能得到更好的结果。

```

C>TYPE SCR40.C
#include <stdio.h>
#include <graphics.h>
#include <conio.h>
main()
{
static char patterns[][8] = {
 { 0xAA, 0x55, 0xBB, 0x66, 0xCC, 0x77, 0xDD, 0x88 },
 { 0x33, 0x33, 0xCC, 0xCC, 0x33, 0x33, 0xCC, 0xCC },
 },
int g—driver,g—mode;
struct arccoordstype arcinfo;
struct linesettingstype lineinfo;
struct fillsettingstype fillinfo;
static char u[8];
int xasp,yasp,oldx,oldy;
long xlong;
int t1[]={180,10,220,50,200,80},n;
int t2[]={180,20,200,50,190,80,180,20};
int t3[]={5,160,45,160,45,180,5,180};
detectgraph(&g—driver,&g—mode);
initgraph(&g—driver,&g—mode,"");
printf("g—driver=%d g—mode=%d\n",g—driver,g—mode);
/* g—driver=1 g—mode=4 */

setgraphmode(2);
arc(150,150,0,89,50);
getarccoords(&arcinfo);
/* {x:150, y:150, xstart:200, ystart:150, xend:150, yend:131 } */
line(arcinfo.xstart,arcinfo.ystart,arcinfo.xend,arcinfo.yend);
moveto(260,30);
linereel(10,20);
lineto(250,70);
moverel(10,20);
getlinesettings(&lineinfo);
/* {linestyle:0, upattern:0, thickness:1 } */
setlinestyle(DOTTED—LINE,1,THICK—WIDTH);
lineto(300,70);
circle(150,150,10);

```

```

ellipse(150,150,0,270,70,25);
ellipse(150,150,0,360,80,15);
setfillstyle(CLOSE-DOT-FILL,LIGHTRED);
setbkcolor(3);
setcolor(RED);
pieslice(100,100,0,134,49);
setfillstyle(SOLID-FILL,LIGHTBLUE);
pieslice(100,100,135,225,49);
setfillstyle(INTERLEAVE-FILL,WHITE);
pieslice(105,105,225,360,49);
getaspectratio(&xasp,&yasp); /* xasp=4167 yasp=10000 */
oldx=xasp;oldy=yasp;
xlong=(100L*(long)yasp)/(long)xasp; /* xlong=239 */
rectangle(0,0,(int)xlong,100);
setfillstyle(8,MAGENTA); /* 8=XHATCH-FILL */
bar3d(100,10,140,60,5,1);
setfillstyle(HATCH-FILL,RED);
bar(30,30,70,50);
setlinestyle(USERBIT-LINE,0xFFFF,THICK-WIDTH);
n=2;
drawpoly(sizeof(t1)/(n*sizeof(int)),t1);
setlinestyle(USERBIT-LINE,0xFFFF,2); /* 2=NORM-WIDTH */
drawpoly(sizeof(t2)/(n*sizeof(int)),t2);
setfillstyle(LINE-FILL,RED);
filledipse(280,150,40,40);
n=10;
if(xasp<yasp)
 do
 { xasp+=2000;
 setaspectratio(xasp,yasp);
 circle(280,150,n);
 n+=8;
 }while(xasp<yasp);
setfillstyle(WIDE-DOT-FILL,RED);
sector(280,40,0,90,40,20);
setaspectratio(cldx,oldy);
delay(5000);
setfillstyle(LTBKSLASH-FILL,RED);
sector(280,40,0,90,40,20);
setfillstyle(EMPTY-FILL,GREEN);
fillpoly(4,t3);
getfillsettings(&fillinfo); /* { pattern,0, color:2 } */
setfillpattern(&patterns[0][0],3);
bar(10,120,40,150);
getfillpattern(u); /* u[0],9m: AA 55 BB 66 CC 77 DD 88 00 */
setfillpattern(&patterns[1][0],2);

```

```

bar(45,120,60,150);
getch();
closegraph();
}

```

### 3. 状态查询函数一览

不论在文本方式状态,或者在图形方式状态,有时需要了解当前屏幕情况时便可用状态函数查询。汇总如下:

#### 文本方式

|                |               |
|----------------|---------------|
| · 返回窗口信息       | gettextinfo() |
| · 返回光标所在处 x 坐标 | wherex()      |
| · 返回光标所在处 y 坐标 | wherey()      |

#### 图形方式

|                                  |                   |
|----------------------------------|-------------------|
| · 返回上次调用 arc() 或 ellipse() 的坐标信息 | getarccoords()    |
| · 返回屏幕图形的纵横比                     | getaspectratio()  |
| · 返回当前的背景色                       | getbkcolor()      |
| · 返回当前的绘图色                       | getcolor()        |
| · 返回当前图形驱动程序名                    | getdrivername()   |
| · 返回用户定义的填充模式                    | getfillpattern()  |
| · 返回当前填充模式及颜色                    | getfillsettings() |
| · 返回当前图形模式                       | getgraphmode()    |
| · 返回当前线段格式、模式和粗细                 | getlinesettings() |
| · 返回当前最高的有效像素值                   | getmaxcolor()     |
| · 返回最大的模式给当前的驱动程序                | getmaxmode()      |
| · 返回当前 x 方向的分辨率                  | getmaxx()         |
| · 返回当前 y 方向的分辨率                  | getmaxy()         |
| · 返回一个给定驱动程序的模式名                 | getmodename()     |
| · 返回一个给定驱动程序的模式范围                | getmoderange()    |
| · 返回当前的调色板及其尺寸                   | getpalette()      |
| · 返回 (x,y) 处的像素颜色                | getpixel()        |
| · 返回当前的正文字体、方向、尺寸及对齐方式           | gettextsettings() |
| · 返回当前视口信息                       | getviewsettings() |
| · 返回 CP 的 x 坐标                   | getx()            |
| · 返回 CP 的 y 坐标                   | gety()            |

程序 SCR41.C 列出了 TC 的状态查询函数 (但未包括两个查错函数 grapherrormsg() 和 graphresult()), 这些函数名的前三个字母均为 get。其中一些并没有参数,只返回一个要查的信息;另一些则要利用 graphics.h 中定义的结构类型定义一个指向结构的指针,查得的信息便填入所指结构的域中,而函数本身并不返回任何值。

程序 SCR41.C 列出了图形方式下各查询函数的使用方法,注解中列出的结果为单步调试时用调试表达式得到。

```

C>TYPE SCR41.C
#include <graphics.h>

```

```

main()
{
int g=driver,g=mode,g=modelo,g=modehi;
int xasp,yasp;
int bk,co;
char *dname,*mname,pa;
int gm,mac,mam,maX,maY;
struct arccoordstype ainfo;
struct fillsettingstype finfo;
struct linesettingstype linfo;
struct palettetype pinfo;
struct textsettingstype tinfo;
struct viewporttype vinfo;
unsigned pixel;
int x,y;
detectgraph(&g=driver,&g=mode);
initgraph(&g=driver,&g=mode,"");
getarccoords(&ainfo); /* 返回最近画圆弧后的坐标信息 */
/* ainfo.r: {x:0, y:0, xstart:0, ystart:0, xend:0, yend:0} */
getaspectratio(&xasp,&yasp); /* 返回图形屏幕的纵横比率 */
/* xasp=4167 yasp=10000 */
bk=getbkcolor(); /* bk=0 返回当前的背景色 */
co=getcolor(); /* co=1 返回当前的画线色 */
dname=getdrivername(); /* dname="CGA" 返回当前驱动程序名 */
getfillpattern(&pa); /* pa,r: ' ' 返回当前用户定义的填充模式 */
getfillsettings(&finfo); /* 返回当前填充模式和颜色 */
/* finfo.r: { pattern:1, color:1 } */
gm=getgraphmode(); /* gm=4 返回当前显示的图形模式 */
getlinesettings(&linfo); /* 返回当前画线的种类及宽度 */
/* linfo.r: {linestyle:0, upattern:0, thickness:1} */
mac=getmaxcolor(); /* mac=1 返回当前最高像素值 */
mam=getmaxmode(); /* mam=4 返回当前驱动程序的最大显示模式 */
maX=getmaxx(); /* maX=639 返回当前图形驱动程序和模式下最大的 x (列)值 */
maY=getmaxy(); /* maY=199 返回当前图形驱动程序和模式下最大的 y (行)值 */
mname=getmodename(1); /* 返回当前驱动程序模式名 */
/* mname="320 x 200 CGA C1" */
getmoderange(g=driver,&g=modelo,&g=modehi); /* 返回驱动程序模式范围 */
/* g=driver=1 g=modelo=0 g=modehi=4 */
getpalette(&pinfo); /* 返回当前调色板尺寸及颜色 */
/* pinfo.r: {size: '\x10', colors:""} */
pixel=getpixel(1,1); /* pixel=0 返回指定像素点的颜色 */
gettextsettings(&tinfo); /* 返回当前文本的字体、方向、放大因子和对齐方式 */
/* tinfo.r: {font:0, direction:0, charsize:1, horiz:0, vert:2} */
getviewsettings(&vinfo); /* 返回当前视口的信息 */
/* vinfo.r: {left:0, top:0, right:639, bottom:199, clip:1} */
x=getx(); /* x=0 返回图形当前位置 (CP) 的 x (列)坐标 */

```

```

y=gety(); /* y=0 返回图形当前位置 (CP) 的 y (行)坐标 */
getch();
closegraph();
}

```

### 35.6.9 图形方式下的错误处理

- 返回最近一次出错图形操作的错误代码      —1 graphresult()
  - 返回同一个对应于错误代码的错误信息串      —2 grapherrormsg()
- 1 int far —Cdecl graphresult(void);
- 返回最近一次出错图形操作的错误代码 (0 ~ -18, 参见枚举 graphics—errors)。

```

enum graphics—errors {
 grOk = 0, /* 无错误 */
 grNoInitGraph = -1,
 grNotDetected = -2,
 grFileNotFound = -3,
 grInvalidDriver = -4,
 grNoLoadMem = -5,
 grNoScanMem = -6,
 grNoFloodMem = -7,
 grFontNotFound = -8,
 grNoFontMem = -9,
 grInvalidMode = -10,
 grError = -11, /* generic error */
 grIOError = -12,
 grInvalidFont = -13,
 grInvalidFontNum = -14,
 grInvalidVersion = -18
};

```

下列函数调用可能产生错误码:

|                   |                |                     |                     |
|-------------------|----------------|---------------------|---------------------|
| bar()             | bar3d()        | clearviewport()     | closegraph()        |
| detectgraph()     | drawpoly()     | fillpoly()          | floodfill()         |
| getgraphmode()    | imagesize()    | initgraph()         | installuserdriver() |
| installuserfont() | pieslice()     | registerbgidriver() | registerbgifont()   |
| setallpalette()   | setfillstyle() | setgraphbufsize()   | setgraphmode()      |
| setlinestyle()    | setpalette()   | settextjustify()    | settextstyle()      |

—2 char \* far —Cdecl grapherrormsg(int errorcode);

它返回同 errorcode 相联系的字符串指针,而 errorcode 是由 graphresult() 得到的。如果将字符串打印出来,就可以从字符串更好理解发生错误的原因。

```

C>TYPE SCR42.C
#include "graphics.h"
#define ER(X) printf("%d %s\n",X,grapherrormsg(X))
main()
{
 int errorcode,code;
 errorcode=graphresult();
}

```

```

printf("code string\n\n");
ER(errorcode); /* 自动判断错误靠 graphresult() 函数 */
ER(-1); /* 人为控制错误输出只要指定错误码 */
for(code=2; code<19;code++)
ER(-code);
}
/* 实际测得的输出错误字符串 (可跟枚举 graphics—errors 比较)
code string
0 No error 无错误
-1 (BGI) graphics not installed *.BGI 文件未安装 (未找到)
-2 Graphics hardware not detected 图形硬件不能测试
-3 Device driver file not found () 设备驱动文件未找到
-4 Invalid device driver file () 无效的设备驱动文件
-5 Not enough memory to load driver 没有足够的内存来装载驱动程序
-6 Out of memory in scan fill 扫描填充图形时内存不够
-7 Out of memory in flood fill 填充图形时内存不够
-8 Font file not found () 字体文件 (*.CHR) 未找到
-9 Not enough memory to load font 装载字体文件时内存不够
-10 Invalid graphics mode for selected driver 对选定的驱动程序指定的图形显示模式无效

-11 Graphics error 图形错误
-12 Graphics I/O error 图形 I/O (输入/输出) 错误
-13 Invalid font file () 无效的字体文件
-14 Invalid font number 无效的字体号
-15 Graphics error (65521) 图形错误 (65521)
-16 Invalid Printer Initialize 无效的打印机初始化
-17 Printer Module Not Linked 打印机模块不能连接
-18 Invalid File Version Number 无效的文件版本
*/

```

C>TYPE SCR43.C

```

#include "graphics.h"
main(){
int GraphMode,GraphDriver = VGA,ErrorCode1,ErrorCode2;
initgraph(&GraphDriver, &GraphMode, "");
ErrorCode1 = graphresult();
ErrorCode2 = graphresult(); /* 连续检测 */
printf("Graphics System Error: %s\n",grapherrormsg(ErrorCode1));
/* printf(" Graphics System Error: %s %s\n",
grapherrormsg(ErrorCode1), grapherrormsg(ErrorCode2));
如用此语句,而图形卡与程序执行时装入的驱动程序 EGAVGA.BGI 不同,则输出:
Divide error
*/
getch();
closegraph();
}

```

/\* 如果当前目录中有驱动文件 EGAVGA.BGI (注意:不是 VGA.BGI),但当前图形卡实际是

CGA 卡,则在集成环境调试时程序输出:

Graphics System Error:Graphics error (139)

而当退出集成环境执行程序时则没有这种提示输出。 \* /

C>TYPE SCR44.C

```
#include "graphics.h"
main(){
int GraphMode,GraphDriver = DETECT,ErrorCode1,ErrorCode2;
initgraph(&GraphDriver, &GraphMode, "");
settextstyle(1,1,2); /* 设置字体 */
ErrorCode1 = graphresult();
ErrorCode2 = graphresult(); /* 连续检测 */
printf("Graphics System Error:%s\n",grapherrormsg(ErrorCode1));
/* printf(" Graphics System Error: %s\n %s\n ",
grapherrormsg(ErrorCode1), grapherrormsg(ErrorCode2));
用此语句,而当前目录中无指定字体文件时输出:
Graphics System Error:Font file not found (TRIP.CHR)
Font file not found (TRIP.CHR)连续输出和前次一样的内容
*/
getch();
closegraph();
}
/* 如当前目录中无检测出的驱动程序 (*.BGI),则输出:
BGI Error: Graphics not initialized (use 'initgraph')
如当前目录中无检测出的字体文件 (*.CHR),则输出:
Graphics System Error:Font file not found (TRIP.CHR)
*/
```

很容易从程序 SCR43.C 和 SCR44.C 看出图形函数报告的形式和不同。当且仅当图形函数报告一个错误时,错误返回码才发生变化。由于函数被调用时只检测最近一次错误,如无错误,内部错误返回代码复位为 0,因此前次检测到的结果将被丢弃。为此,想要知道一个图形函数调用后可能出现的错误,常常先用 graphresult() 函数检测,并将其返回值存入一个临时变量中,然后再对这个变量处理;或调用 printf()、grapherrormsg() 和 exit() 函数向屏幕输出后退出执行程序,或者在源程序中作另一些处理,如调用别的函数等。

使用伪变量 —AH 和 —AL 及产生软中断的函数 geninterrupt() 也可得到或设置屏幕显示模式。程序 SCR45.C 为得到模式,程序 SCR46.C 为设置模式。

C>TYPE SCR45.C

```
#include "dos.h"
struct mode{ int videomode,columns,videopage;}; /* 视频模式、列数、页数 */
main(){
struct mode mtr; * modeptr;
unsigned char hold;
modeptr=&mtr;
—AH=0xf; /* BIOS 中断 10H 的子功能 15H 找出当前图形模式 */
geninterrupt(0x10);
hold=—AH; /* 注意:这句是必要的! 如果你将此句换成 modeptr->columns=—AH; */
```

```

/* 则将出错! 从此可见使用伪变量的危险性。事实上,你只要使用监 */
/* 视表达式 —AH,X,—AL,X 和 —BH,X,则在用 F7 单步调试时清楚地看 */
/* 到在执行 getinterrupt(0x10); 和其后语句 modeptr->columns= */
/* —AH 前后,—AH,—AL 中的值发生了变化,这是不允许的。这是因 */
/* 为在执行赋值语句时要使用寄存器 AX 的缘故。 */
modeptr->videomode=-AL; /* AL 中返回模式 */
modeptr->videopage=-BH; /* BH 中返回页数 */
modeptr->columns=hold; /* AH 中返回列数 */
printf("videomode=%d columns=%d videopage=%d\n",
 modeptr->videomode,modeptr->columns,modeptr->videopage);
}
/* 程序输出 videomode=3 columns=80 videopage=0 */
C>TYPE SCR46.C /* 设置上述模式,采用调用函数方式 */
#include "dos.h"
struct mode{ int videomode,columns,videopage;};
scrsetmode(struct mode *modeptr) /* 设置模式和页函数 */
{
 -AH=0x0; /* BIOS 中断 10H 的子功能 0 是设置屏幕模式 */
 -AL=modeptr->videomode; /* AL 中为模式号 */
 geninterrupt(0x10); /* 执行中断 10H */
 -AH=0x5; /* BIOS 中断 10H 的子功能 5H 是设置页 */
 -AL=modeptr->videopage; /* AL 中放激活页号 */
 geninterrupt(0x10);
}
main(){ /* 调用函数方法 */
 struct mode newmode;
 newmode.videomode=3; /* 模式 3 */
 newmode.videopage=0; /* 0 页 */
 scrsetmode(&newmode); /* 调用函数 */
}

```

## 35.7 BIOS 中断 INT 10H 的功能

虽然可以对显卡寄存器编程,但在很多情况下可用 BIOS 例程来编程。利用 BIOS 例程编程比用 DOS 功能调用速度要快,并且还能利用 BIOS 一级的兼容性来解决汉字显示问题。

IBM PC 的 ROM BIOS 中含有管理视频的例程,这些例程可以通过中断 INT 10H 方式调用。选择不同的视频例程的方法是,先在寄存器 AH 中存入适当的功能号,并将要传递给例程的参数按规定存入指定的其它 80x86 寄存器中,最后执行 INT 10H。例程执行后的返回值也是存在指定的 80x86 寄存器中。一般说来,例程不会检查参数值的正确性。因此,在填参数值时务必正确,以免出现不可预料的结果。

TC 调用 INT 10H 的方法主要有两种,一种使用标头文件 dos.h 中的三个结构

```

struct WORDREGS {unsigned int ax,bx,cx,dx,si,di,cflag,flags;};
struct BYTEREGS {unsigned char al,ah,bl,bh,cl,ch,dl,dh;};
struct REGS {

```



```

struct WORDREGS x;
struct BYTEREGS b;
};

```

和一些中断调用函数

int —Cdecl int86(int intno, union REGS \*inregs, union REGS \*outregs);等;另一种是使用伪变量 —AX,—BX,... 等给参数赋值,然后调用 dos.h 中定义的宏

```
#define geninterrupt(i) —int—(i)
```

实现。仅管使用伪变量能提高运行速度,但使用伪变量有时是危险的,因此建议你常采用第一种方法。下面列出中断 INT 10H 的一些主要功能,注意,对有些汉字操作系统(汉卡),可能会修改 INT 10H 的部分功能。

- |                                |            |
|--------------------------------|------------|
| · 设置显示模式(或称显示方式)               | —0 功能 0H   |
| · 设置文本方式下光标大小                  | —1 功能 1H   |
| · 设置光标位置                       | —2 功能 2H   |
| · 获取光标位置和大小信息                  | —3 功能 3H   |
| · 读光笔位置                        | —4 功能 4H   |
| · 选择视频可见页(活动显示页)               | —5 功能 5H   |
| · 规定垂直向上滚动行数                   | —6 功能 6H   |
| · 规定垂直向下滚动行数                   | —7 功能 7H   |
| · 获取光标所在字符及显示属性                | —8 功能 8H   |
| · 在光标处写入要显示的字符及属性              | —9 功能 9H   |
| · 在光标处显示指定字符                   | —10 功能 0AH |
| · 设置屏幕边界颜色,或选择四色调色板            | —11 功能 0BH |
| · 设置像素值                        | —12 功能 0CH |
| · 获取像素值                        | —13 功能 0DH |
| · 以电传打字机方式显示字符                 | —14 功能 0EH |
| · 获取当前 BIOS 的显示方式              | —15 功能 0FH |
| · 设置调色板寄存器,控制边界颜色、灰度及闪烁属性      | —16 功能 10H |
| · 装载 ASCII 码字符集/(字符发生器介面)      | —17 功能 11H |
| · 视频系统状态(第二组)                  | —18 功能 12H |
| · 显示字符串                        | —19 功能 13H |
| · 装入用户规定的液晶(LCD)显示字体           | —20 功能 14H |
| · 视频显示的配合方式                    | —21 功能 1AH |
| · 视频 BIOS 功能及状态                | —22 功能 1BH |
| · 储存或还原视频 ROM BIOS 的状态(仅用于VGA) | —23 功能 1CH |

—0 功能 0H: 设置显示方式(或称显示模式,参见表 35—10)

(1)【入口参数】(即设置进入 BIOS 的 INT 10H 例程的参数值)

AH=0 功能号

AL=1 BIOS 的显示方式(mode)

(2)返回值:如果 AL 的高位置位,IBM 标准模式不清屏(只用于 EGA 或更高档显示器)

### (3) 视频数据区 (BIOS 工作区) 变动部分

0040:0049 (简称为 49)、4A、4C、4E、50、60、62、63、65、66、84、85、87、88

### (4) 说明

本功能在设置指定显示方式的同时,自动选择内定的调色板;对 MCGA 和 VGA 设置视频数据区中的 Flags 标志;

对 MCGA、EGA 和 VGA,显示方式 0 和 1 相同,方式 2 和 3 相同,方式 4 和 5 相同。对它们如将 AL 内指定的显示方式的最高位 (第 7 位) 置成 1,则选定此新显示方式时,BIOS 不会清除视频缓冲区内的内容。

对有些机型可能采用专用显示方式。

C>TYPE INT10-1.C

```
#include "dos.h"
#define PE(X) printf(" 0x%x",peek(0x40,X))
#define PEB(Y) printf(" 0x%x\n",peekb(0x40,Y))
#define PEND printf("\n")
#define PE4(Z) PE(Z),PE(Z+2),PE(Z+4),PE(Z+6);PEND
/* 语句行后注释中为测得 CGA 卡的值,未给出值均为 0x0 */
main()
{
 int x10,x49,x4A,x4C,x4E,x50,x60,x62,x63,x65,x66,x84,x85,x87,x88;
 PEB(0x10); /* 0x63 显示方式 */
 PEB(0x49); /* 0x3 */
 PE(0x4A);PEND; /* 0x50 */
 PE(0x4C);PEND; /* 0x1000 */
 PE(0x4E);PEND; /* 0x0 视频开始页 */
 PE(0x50),PE(0x52),PE(0x54),PE(0x56),PE(0x58);
 PE(0x5A),PE(0x5C),PE(0x5E);PEND;
 /* 0x1800 0x0 0x0 0x0 0x0 0x0 0x0 0x0 光标位置 */
 PE(0x60);PEND; /* 0x607 文本方式下光标大小 */
 PEB(0x62); /* 0x0 当前显示页 */
 PE(0x63);PEND; /* 0x3d4 CRTC 地址寄存器的端口地址 */
 PEB(0x65); /* 0x29 模式控制寄存器的值 */
 PEB(0x66); /* 0x30 选择 4 色调色板 */
 PEB(0x84);
 PE(0x85);PEND;
 PEB(0x87);
 PEB(0x88);
 PEB(0x89);
 PEB(0x8A);
 PE4(0xA8);
 PE4(0xAC);
 PE4(0xB0);
 PE4(0xB4);
 PE4(0xB8);
 PE4(0xBC);
 PE4(0xC0);
 getch();
}
```

```

}
/* 对 EGA 卡测得的值
0x61
0x3
0x50
0x1000
0x0
0x1800 0x0 0x0 0x0 0x0 0x0 0x0 0x0
0x607
0x0
0x3d4
0x29
0x30
0x18
0xe
0x60
0xffff9
0x0
0x0
0x8a 0xc000 0x0 0x0
0x0 0x0 0x27 0x0
0x27 0x0 0x0 0x0
0x0 0x0 0x0 0x0
0x0 0x0 0x0 0x0
0x0 0x0 0x0 0x0
0x0 0x0 0x0 0x0
*/

```

#### (5) 定义函数方法

##### 第一种方法

```

#include "dos.h"
void Setvideomode(int mode)
{ union REGS r;
 r.h.ah=0;
 r.h.al=mode;
 int86(0x10, &r, &r);
}

```

##### 第二种方法 (用伪变量)

```

#include "dos.h"
void Setvideomode(int mode)
{
 —AH=0;
 —AL=mode; /* 选显示方式 mode */
 geninterrupt(0x10);
}

```

—1 功能 1H: 设置文本方式下光标大小

##### (1) 入口参数

AH=1 功能号

CH=0 光标最上面一条线 (startline)

CH 的位 7 应等于 0, 位 6、位 5 确定光标闪动 (00= 正常, 01= 不可见,

10= 飘忽, 11= 慢)

CL=7 光标最下面一条线 (endline)

(2) 返回值: 无

(3) 视频数据区变动字节:0040:0060。注意:0040:0085和0040:0086给出了当前显示卡对应的字符高度,可用它作为基础设置光标。字符高度=peek(0x40,0x85)-1。

#### (4) 说明

它实际设置光标起始线 (Cursor start) 和光标终止线 (Cursor end) 两个寄存器的值,并使文本光标出现在指定的位置上。对显示方式 0~3,缺省的起始线是 6,终止线是 7,光标闪烁(闪烁是由硬件决定的)。起始线和终止线的合法值取决于所用适配器的类型。将文本光标关闭的方法之一,是设置不合法的起始线或终止线,例如取 CH=20H。但这种方法有时并不可靠。另一种方法是将光标设置到一个不可显示的位置上,例如 (x,y)=(0,25)。因为有效行是 (0,24),第 26 行不存在。

本功能只能用于文本方式,对图形方式不适用。

#### (5) 定义函数

C>TYPE INT10-2.C

```
#include "dos.h"
```

```
void Setstyle(int startline, int stopline)
```

```
{
```

```
static union REGS r;
```

```
r.x.ax=0x0100; /* AH=1 AL=0 */
```

```
r.x.bx=0; /* 视频页 0 */
```

```
r.h.ch=startline; /* 起始线 */
```

```
r.h.cl=endline; /* 终止线 */
```

```
int86(16,&r,&r); /* 即 int86(0x10,&r,&r) */
```

```
}
```

### —2 功能 2H: 设置光标位置

#### (1) 入口参数

AH=2 功能号

BH=1 视频当前显示页号 (page) 对文本方式 0 和 1 取值范围 0~7  
2 和 3 取值范围 0~3  
对图形方式 0

DH=5 光标指定位置,行号 (row), 00H 是顶部

DL=10 光标指定位置,列号 (col), 00H 是左边

#### (2) 返回值: 无

#### (3) 视频数据区变动部分: 0040:0050

#### (4) 说明

(DL,DH) 值的范围,对 80×25 显示是 (0,0)~(79,24); 对 40×25 显示是 (0,0)~(39,24) 超出此范围将产生不可预料的结果。

#### (5) 定义函数

C>TYPE INT10-3.C

```
#include "dos.h"
```

```
void Setcp(int row, int col, int page)
```

```
{
```

```
union REGS r;
```

```
r.h.ah=2;
```

```
r.h.bh=page;
```

```

r.h.dh=row;
r.h.dl=col;
int86(0x10,&r,&r);
}

```

### —3 功能 3H: 获取光标位置和大小信息

#### (1) 入口参数

AH=3 功能号

BH=0 视频页号

#### (2) 返回值

CH= 光标起始线 (\* startline)

CL= 光标终止线 (\* endl ine)

DH= 光标所在行 (\* row)

DL= 光标所在列 (\* col)

#### (3) 视频数据变动部分: 无

#### (4) 说明

CH,CL 值只在文本方式下才有意义。

CGA 在  $80 \times 25$  文本方式中,有 4 个显示页;  $40 \times 25$  文本方式有 8 个显示页。每页均有单独的光标,每个光标都可以用这一功能查看,而不管该页是否实际显示在屏幕上(即是否为可见页)。页间转换可用 INT 10H 的功能 5 实现。EGA 和 VGA 有类似情况。

#### (5) 定义函数

C>TYPE INT10—4.C

```
#include "dos.h"
```

```
void Getcp(int *startline, int *endl ine, int *row, int *col, int page)
```

```
{
```

```
union REGS r;
```

```
r.x.ax=3;
```

```
r.x.bx=page;
```

```
int86(0x10,&r,&r);
```

```
*startline=r.h.ch;
```

```
*endl ine=r.h.cl;
```

```
*row=r.h.dh;
```

```
*col=r.h.dl;
```

```
}
```

### —4 功能 4H: 读光笔位置

#### (1) 入口参数

AH=4 功能号

#### (2) 返回值

AH=1 表示传回了光笔位置; 为 0 表示没有传回

BX= 像素的 x 坐标(列坐标)

CH= 像素的 y 坐标(CGA 和 EGA 方式 4、5 和 6)

CL= 像素的 y 坐标(EGA 的非 4、5 和 6 的图形方式)

DH= 光标所在字符的行

DL= 光标所在字符的列

(3) 视频数据区变动部分：无

(4) 说明

1. MCGA 和 VGA 不能使用光笔。

2. 虽然光笔返回的 (x,y) 坐标范围可以是当前图形屏幕的整个范围,但是坐标分布是不均匀的。y 坐标总是 2 的倍数,而 x 坐标或是 4 的倍数 (320 × 200 图形方式),或是 8 的倍数 (640 × 200 图形方式)。

3. 使用光笔宜用较亮的背景色。

—5 功能 5H: 选择视频可见页 (活动显示页)

(1) 入口参数

AH=5 功能号

AL= 选择的可见页号

(2) 返回值：无

(3) 视频数据区变动部分：0040:004E、0040:0062

当前页面的大小 (size) 即所占字节数存在 0040:004C 和 0040:004D 中。

C>TYPE INT10—5.C

```
page—size()
{
 char far *p=0;
 long size;
 size=p[0x44C]+256*p[0x44D];
 printf("页面大小=%ld 字节\n",size);
}
```

(4) 说明

1. 页间转换不影响它们各自的内容。

2. 对 CGA 在图形方式下只有一页 (0 页),因此本功能无用处。对不同的显示方式有不同的页存在 (参见 graphics.h 中的枚举 graphics—modes)。

3. 由于 BIOS 例程不检查 RAM 是否有足够的空间够指定的视频页用,因此,若指定页超出了实际的视频缓冲范围,则不产生任何图象。

(5) 定义函数

C>TYPE INT10—6.C

```
#include "dos.h"
void Setvisualpage(int page)
{
 union REGS r;
 r.h.ah=5;
 r.h.al=page;
 int86(0x10,&r,&r);
}
```

—6 功能 6H: 将滚动窗内的显示数据垂直向上滚动规定行数

(1) 入口参数

AH=6 功能号

AL=2 垂直向上滚动行数 (uprow),如它等于 0 则清窗口

BH=0xd7 写窗口底部空行用的字符属性或像素值 (attr)

CH=1 向上滚动窗口左上角所在行 (top)

CL=26 左上角所在列 (left)

DH=10 右下角所在行 (bottom)

DL=30 右下角所在列 (right)

(2) 返回值：无

(3) 视频数据区变动部分：无

(4) 说明

1. (CH,CL,DH,DL) 组成了一个滚动范围,或称为【滚动窗】,其坐标值是指在当前显示方式下屏幕的绝对坐标。例如,对 CGA 文本方式  $80 \times 25$  行,坐标范围为: (0,0) ~ (79,24),对 BIOS 显示方式  $40 \times 25$  行,坐标范围为: (0,0) ~ (39,24)。

2. 本功能只对滚动窗内的显示的数据产生影响,或者说滚动窗外的显示的数据在执行本功能后显示没有任何变化。而滚动窗底下从 bottom 行往上的显示数据行从 left 到 right 指定列范围内的显示的数据将全部上移 uprow 行,滚动窗顶上的 uprow(指定的滚动)行的从 left 到 right 指定列范围内的显示数据将消失,滚动窗底下部分用空白填补(背景色)。

3. BH 所存的属性为上滚后滚动窗底下部分留下空白部分的属性。在文本方式,就是显示字符的显示属性;在图形方式,对  $640 \times 200$  双色显示,显示属性字节的每一位等于一个像素值,而对  $320 \times 200$  四色显示,显示属性字节每两位代表一个像素值。在 EGA、MCGA 和 VGA 的其它图形方式中,BH 值决定了填补空白的像素值。

4. 在 EGA、MCGA 和 VGA 的  $320 \times 200$  的四色方式中,无论目前显示的是那一页,本功能只滚动第 1 页即 0 页。若 AL 值为 0,则 BIOS 会把整个滚动窗变为空白;当滚动窗为整个屏幕且 AL=0 时,整个屏幕的显示数据被清除。

(5) 定义函数

```
C>TYPE INT10-7.C
#include "dos.h" /* 在文本方式下 */
void Scrollup(int uprow, int attr, int left,
 int top, int right, int bottom)
{
 union REGS r;
 r.h.ah=6;
 r.h.al=uprow;
 r.h.bh=attr;
 r.h.ch=left;
 r.h.cl=top;
 r.h.dh=right;
 r.h.dl=bottom;
 int86(0x10,&r,&r);
}
main(){
 int i,j;
 window(0,0,79,24); /* 指定整个屏幕 */
 clrscr(); /* 清屏幕 */
 for(i=0;i<25;i++)
 for(j=0;j<80;j++)
```

```

 cputs("O"); /* 屏幕显示整屏字母 O */
window(10,1,60,20); /* 指定新窗口 */
clrscr(); /* 清当前指定窗口 */
cputs("abcdefghijklmnopqrstuvwxyz"); /* 输出 */
gotoxy(1,2);
cputs(" g12345678901234567890g");
gotoxy(1,3);
cputs("aAbBcCdDeEfFgGhHiIjKlLmMnNoOpPqQrRsStTuUvVwWxXyYzZ012345");
gotoxy(1,4);
cputs(" g1A2B3C4D5E6F7G8H9I0J1K2L3M4N5O6P7Q8R9S0T1U2V3Wg");
gotoxy(1,9);
cputs("Joooooooooooooooooooooooooooo");
gotoxy(1,10);
cputs("hHHH");
gotoxy(1,11);
cputs("fFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF");
gotoxy(1,12);
cputs("Tttttttttttttttttttttttttttttttt");
getch();
Scrollup(2,0xd7,1,26,10,30); /* 滚动窗底下有粉红色空白区域 */
getch();
}

```

C>TYPE INT10—8.C

```

#include "graphics.h"
#include "dos.h" /* 在图形方式下 */
void Scrollup(int uprow, int attr, int left,
 int top, int right, int bottom)
{
 union REGS r;
 r.h.ah=6;
 r.h.al=uprow;
 r.h.bh=attr;
 r.h.ch=left;
 r.h.cl=top;
 r.h.dh=right;
 r.h.dl=bottom;
 int86(0x10,&r,&r);
}
main(){
 int i,j;
 int driver=CGA,mode=CGAC2;
 initgraph(&driver,&mode,"");
 cleardevice();
 setviewport(0,0,319,199,1);
 for(i=0;i<200;i+=8)
 {moveto(0,i);

```





```

union REGS r;
r.h.ah=UpDown ? 6:7; /* 当 UpDown 值大于 0 时为上滚,小于 0 时为下滚 */
r.h.al=uprow;
r.h.bh=attr;
r.h.ch=left;
r.h.cl=top;
r.h.dh=right;
r.h.dl=bottom;
int86(0x10,&r,&r);
}

```

—8 功能 8H: 获取光标所在处的文本字符及显示属性

(1) 入口参数

AH=8 功能号

BH=0 可见页号

(2) 返回值:

AH= 显示属性 (仅用于文本方式)

AL= 字符的 ASCII 码

(3) 视频数据区变动部分: 无

(4) 说明

1. 在图形方式, PC/XT 和 PC/AT 机的 BIOS 用的是存在 ROM 的 F000:FA6E 处的表。EGA、MCGA 和 VGA 使用 INT 43H 中断向量所指定的图形字符定义表。在 CGA 相兼容的图形方式 4、5 和 6 中, 若 ASCII 码为 80H ~ 0FFH, 则 BIOS 会用 INT 1FH 中断向量指定字符定义表。

判断图形方式中的字符为那一个 ASCII 码, BIOS 视非 0 的像素为前景像素。BIOS 拿来比较的是前景 (非 0) 和背景 (0) 像素的模式, 若和字符定义表中某个字模相同, 则为该字符; 如果没有找到这样的字符, AL=0。

2. 在 EGA、MCGA 和 VGA 的  $320 \times 200$  四色显示方式下, 只能处理 0 页。

3. 对于单色显示, 前景为 1 背景为 0 时出现下划线。

(5) 定义函数

```

C>TYPE INT10-11.C
#include "dos.h" /* 文本方式下 */
#include "conio.h"
#define BSET—row—col(Y,X) —AH=2; /* 功能 2 是置当前光标位置 */ \
—BH=0; \
—DH=Y; /* 坐标为屏幕绝对坐标 */ \
—DL=X; \
geninterrupt(0x10)
#define CPRINT cprintf("\nattribute=%d ascocode=%d\n", \
a.attribute,a.ascocode);

struct asc—att{
unsigned char attribute;
unsigned char ascocode;
};

Getasc—attr(int page, struct asc—att *AA)

```

```

{
 /* 得到光标处的字符和属性 */
 union REGS r;
 r.h.ah=8;
 r.h.bh=page;
 int86(0x10,&r,&r);
 AA->attribute=r.h.ah;
 AA->ascode=r.h.al;
}

main(){
 static struct asc—att a;
 clrscr();
 textattr(WHITE+(GREEN<<4));
 —AH=0xF;
 geninterrupt(0x10);
 BSET—row—col(3,10);
 cputs("A");
 BSET—row—col(3,10);
 Getasc—attr(0, &a);
 CPRINT;
 BSET—row—col(22,10);
 printf("A");
 BSET—row—col(22,10);
 Getasc—attr(0, &a);
 CPRINT;
 printf("Demo—end !");
}

```

C>TYPE INT10—12.C

```

#include "graphics.h"
#include "dos.h"
#include "conio.h"
#define BSET—row—col(Y,X) —AH=2;\
 —BH=0;\
 —DH=Y;\
 —DL=X;\
 geninterrupt(0x10)
#define PRINT printf("\nattribute=%d ascode=%d\n",\
 a.attribute,a.ascode);

struct asc—att{
 unsigned char attribute;
 unsigned char ascode;
};

Getasc—attr(int page, struct asc—att * AA)
{
 union REGS r;
 r.h.ah=8;
 r.h.bh=page;

```

```

int86(0x10,&r,&r);
AA->attribute=r.h.ah;
AA->asccode=r.h.al;
}
main(){
static struct asc=att a;
int driver=CGA,mode=CGAC2;
initgraph(&driver,&mode,"");
cleardevice();
—AH=0xF; /* 未给出 AL 值,取缺省显示方式 */
geninterrupt(0x10); /* 完成设置方式的一系列过程 */
BSET—row—col(3,10);
outtext("A"); /* 字母 A 输出到图形当前位置 (CP), */
BSET—row—col(3,10); /* 而不在指定坐标处 */
Getasc—attr(0, &a);
PRINT; /* 输出的属性值无用 */
BSET—row—col(22,10);
printf("A"); /* 字母 A 输出到指定位置 */
BSET—row—col(22,10);
Getasc—attr(0, &a);
PRINT;
printf("Demo—end !");
closegraph();
}

```

—9 功能 9H: 在当前光标处写入要显示的字符及属性

(1) 入口参数

AH=9 功能号

AL=65 写入字符的 ASCII 码或字符 (\*string)

BH=0 视频页号 (page) 或对 320 × 200 的 256 色方式是背景像素值

BL=0x17 写入字符属性 (attr, 文本方式下) 或前景像素值 (图形方式下)

如果位 7 置位, 字符被异或到屏幕上

CX= 写同一字符的次数 (num), 有时称为重复次数或【复制次数】

(2) 返回值: 无

(3) 视频数据变动部分: 无

(4) 说明

1. 复制次数仅对当前页产生效果, 且 CX 值至少为 1。在文本方式, 复制的字符超出屏幕一行长度时, 便自动在下一行接着显示; 对图形方式, 则不允许超出该行剩余的未显示字符的长度。否则将产生不可预料的后果。

2. 在文本方式和 EGA 特有的图形方式, BH 只能是视频页号值; 在图形方式中, 每一个字符所占的大小和字符点阵大小相同, BL 为前景像素值。在 320 × 200 四色方式, 对 EGA、MCGA 和 VGA 的 BIOS 均不得用 BH 的值, 但对 320 × 200 的 256 色图形方式中, BH 为字符背景像素值。在其它所有图形方式中, BH 均代表页号。在非 320 × 200 的 256 色图形方式中, 如果 BL 的最高位即位 7 为 1 时, 则写入字符场和屏幕当前显示字符相异或 (XOR)。同一字符的两次异或运算可以将该字符“擦去”。

3. 字符写入后光标仍在原有的位置上,即未动。要将光标移到指定位置,可用功能2。
4. 为写一个字符而又不想改变其属性可用 INT 10H 的功能 0AH。
5. 如果 AL=32,即写入空字符,那么可以将光标所在字符和它右边的全部字符全部擦除了。这可以调用功能 0FH,从返回的 AH 寄存器内的值知道一行的列数;调用功能 3求得当前光标位置。两者的差便是要删除光标右边字符的重复次数。

#### (5) 定义函数

```
C>TYPE INT10-13.C
#include "dos.h"
#include "conio.h"
#define BSET—row—col(Y,X) —AH=2;\
 —BH=0;\
 —DH=Y;\
 —DL=X;\
 geninterrupt(0x10)

Setchar—attr(int page, char * string, int attr, int num)
{
 /* 写字符及属性 */
 union REGS r;
 r.h.ah=9;
 r.h.al= * string; /* 指定一个字符 */
 r.h.bh=page; /* 页号 */
 r.h.bl=attr; /* 属性 */
 r.x.cx=num; /* 重复次数 */
 int86(0x10,&r,&r);
}

main()
{
 char * s="AB\nCD"; /* 特殊字符 \n 也将显示 */
 clrscr();
 BSET—row—col(3,10);
 Setchar—attr(0,s++,0xd7,4); /* 闪烁显示 4 个字母 A */
 BSET—row—col(5,10);
 Setchar—attr(0,s++,0x2c,80); /* 共显示 80 个字母 B,分 2 行显示 */
 printf("\n\nDemo—end !"); /* 注意这里用 2 个 \n 才能实现换行 */
 BSET—row—col(7,40);
 Setchar—attr(0,s,0x75,8); /* 连续显示 8 个特殊字符 */
}
```

—10 功能 0AH: 在光标当前位置写指定字符

#### (1) 入口参数

AH=0xa 功能号

AL=65 写入字符的 ASCII 码或字符

BH=0 视频页号或对 320 × 200 的 256 色方式为背景像素值

BL=0x7 前景像素值 (仅用于图形方式)

CX=2 重复次数 (2) 返回值: 无

#### (3) 视频数据区变动部分: 无

#### (4) 说明

—11 功能 0BH: 设置屏幕边界 (Overscan) 颜色

|        |     |
|--------|-----|
| AH=0xb | 功能号 |
|--------|-----|

BL=7 边界颜色或背景色 (0~15) BL=0~1 调色板标识符

(3) 视频数据变动部分：0040:0066

本功能主要用于 320 × 200 四色方式和 CGA 的文本方式。当然,其它方式也能用。

对 CGA 和 MCGA, BIOS 会将 BL 的低 5 位值存入颜色选择寄存器 (端口地址是 H)。在  $320 \times 200$  四色方式中, BL 的低 4 位决定了背景色和边界色; 在  $640 \times 200$  和  $640 \times 480$  双色方式中, 低 4 位决定了前景像素值。在 CGA 上这低 4 位也决定了文本方式的边界颜色。BL 的低 5 位中的最左边一位即位 4 决定 CGA 或 MCGA 在图形方式下字为正常亮度或高亮度。当 BL 的位 4 为 1 时, EGA 和 VGA 的为保持兼容性, BIOS 会模拟结果。

在 EGA 和 VGA 的其它显示方式中, BH 不能为 0, 若要对它们设置属性控制器, 应使用 INT 10H 的功能 10H。

对非 320 × 200 四色图形方式,使用它要小心(最好不用它)。

BL=01H 背景,靛、品红和白

C>TYPE INT10-14.C

```
void Setpalette(int —bh, int color—palette)
```

```
/* 大于 0 取 1, 否则取 0 */
```

1

```
/* 双色方式下改变背景色 */
```

```

printf("AAAAA\n");
getch();
Setpalette(0,5);
printf("BBBBBBBB\n");
getch();
closegraph();
Setpalette(0,5); /* 文本方式下改变边界颜色 */
printf("cccccccccccccccc\n");
getch();
}

```

## —12 功能 0CH: 设置像素值

### (1) 入口参数

|        |                                |             |
|--------|--------------------------------|-------------|
| AH=0xc | 功能号                            |             |
| AL=1   | 像素值 (pixel), 如位 7 置位, 该值异或到屏幕上 |             |
|        | 合法像素值: 显示方式 4H~5H              | 像素值 0~3     |
|        | 6H 或 FH                        | 0~1         |
|        | DH, EH 或 10H                   | 0~15        |
|        | BH=0                           | 视频页号 (page) |

CX=200 X(列) 坐标 (col)

DX=100 Y(行) 坐标 (row)

### (2) 返回值: 无

### (3) 视频数据区变动部分: 无

### (4) 说明

坐标值的范围取决于当前选择的图形方式。

在非 320 × 200 的 256 色方式中, 若 AL 的位 7 为 1, 则将 AL 的值与屏幕指定坐标处的像素值相异或 (XOR) 后作为新的像素值, 否则直接取代。

使用 EGA、MCGA 和 VGA 时, BH 为当前选择的页号, 但如果当前显示模式只支持一个页面时 (如在 320 × 200 的四色方式) 选择的页号可忽略。

若目前的 EGA 有 64K, 但要设置 350 行图形方式中的像素值, 则必须考虑存储器位图 (memory map) 和位平面 (bit plane) 的关系。在这种情况下, 应只用奇数位指定像素值, 因此可能的值是: 0, 1, 4 和 5, 或二进制数 0000, 0001, 0100 和 0101。

### (5) 定义函数

C>TYPE INT10—15.C

```
#include "graphics.h"
```

```
#include "dos.h"
```

```
void Printpoint(int pixel, int page, int col, int row)
```

```
{ /* 在屏幕上画出一个点 */
```

```
union REGS r;
```

```
r.h.ah=0xc;
```

```
r.h.al=pixel;
```

```
r.h.bh=page;
```

```
r.x.cx=col;
```

```
r.x.dx=row;
```

```
int86(0x10, &r, &r);
```

```

}
main(){
int driver=CGA,mode=CGAC0,
int i;
initgraph(&driver,&mode,"");
cleardevice();
setbkcolor(1);
setcolor(RED); /* 画出一条直线 */
for(i=0;i<30;i++)Printpoint(1,0,120+i,100);
for(i=0;i<22;i++) /* 进行异或运算后,已画线的中间部分被擦了 */
Printpoint(getcolor()^0x40,0,124+i,100);
getch();
closegraph();
textattr(WHITE+(RED<<4));
for(i=0;i<30;i++) /* 对文本方式不起作用,屏幕上无任何点显示 */
Printpoint(1,0,200+i,100);
getch();
}

```

### —13 功能 0DH: 获取象素值

#### (1) 入口参数

AH=0xd 功能号  
BH=0 视频页号 (page)  
CX=200 X(列)坐标 (col)  
DX=100 Y(行)坐标 (row)

#### (2) 返回值

AL=1 象素值 (point—pixel)

#### (3) 视频数据区变动部分: 无

#### (4) 说明

如当前显示模式只支持一个页面,则 BH 可忽略。

本功能实际和 getcolor() 函数功能大体相同。

#### (5) 定义函数

C>TYPE INT10—16.C

```

#include "graphics.h"
#include "dos.h"
#include "stdlib.h"
void Printpoint(int pixel, int page, int col, int row)
{
union REGS r;
r.h.ah=0xc;
r.h.al= pixel;
r.h.bh=page;
r.x.cx=col;
r.x.dx=row;
int86(0x10,&r,&r);
}

```



```

)
Get—point—pixel(int page, int col, int row)
{
 /* 取得一点的象素 */
 union REGS r;
 r.h.ah=0xd;
 r.h.bh=page;
 r.x.cx=col;
 r.x.dx=row;
 int86(0x10,&r,&r);
 return (r.h.al);
}

main(){
 int driver=CGA,mode=CGAC0;
 int i,point—pixel;
 static char string[10],*s; /* 注意:不要将 string 定义为指针 */
 initgraph(&driver,&mode,"");
 cleardevice();
 setbkcolor(1);
 setcolor(RED);
 for(i=0;i<30;i++)Printpoint(1,0,120+i,100);
 point—pixel=Get—point—pixel(0,124,100);
 outtext("Get point pixel: ");
 s=itoa(point—pixel,string,10); /* 将 point—pixel 转为字符串 */
 outtext(string); /* 实际 s 所指内容和 string[] 相同 */
 outtextxy(getx()-8,gety()+9,s); /* 在上行对应位置上输出象素值 */
 getch();
 closegraph();
}

```

—14 功能 0EH: 以电传打字机方式显示字符

(1) 入口参数

AH=0xe 功能号

AL=7 要显示字符的 ASCII 码或字符

BH=0 视频页号 (page)

BL=1 前景象素值 (只用于图形方式)

(2) 返回值: 无

(3) 视频数据区变动部分: 0040:0050

(4) 说明

1. 当字符的 ASCII 码为 0x7 ('\a', 响铃)、0x8 ('\b', 退一格)、0xa ('\n', 换行)、0xd ('\r', 回车) 这样几个特殊字符时, 屏幕对此并无对应字符显示, 而是控制光标移动或产生相应的动作。INT 10H 的功能 0AH 对这几个特殊字符不产生动作, 而是显示一些不易识别的字符。

2. 对文本方式, 字符可以写入任何合法的当前显示页, 而不管当前显示的是哪一页。

3. 本功能兼有自动换行和自动滚动屏幕功能 (电传打字机就是这样的)。这就是说, 如光标已在行尾, 再显示字符后它会自动移到下一行的行首; 如果光标在屏幕最后一行的末

尾,则再显示字符时屏幕将向上滚动一行,光标位于新的空白行的行首。整个空白行的字符显示属性和上一行最末一个字符的显示属性一样。

4. 因为本功能实际只写字符而不管显示属性,因此显示位置上的字符显示属性应在调用本功能前便先设置。

5. 在图形方式中字符是以点阵大小的长方形写入视频缓冲区中的,字符的像素由 BL 值决定,背景像素一律为 0。本功能会将 BL 值直接传给 INT 10H 的功能 0AH 处理,所以若 BL 的最高位等于 1 时, BIOS 会将 BL 值与当前像素进行异或运算后写入视频缓冲区(结果在屏幕上显示出来)。

#### (5) 定义函数

C>TYPE INT10—17.C

```
#include "graphics.h"
#include "dos.h"
void Writechar(char ch,int page, int pixel)
{
 union REGS r;
 r.h.ah=0xe;
 r.h.al=ch;
 r.h.bh=page;
 r.h.bl=pixel;
 int86(0x10,&r,&r);
}
main(){
 int driver=CGA,mode=CGAC0;
 Writechar('\a',0,0); /* 响铃 */
 Writechar(0x9,0,0); /* 字符 \t,但并不产生横向跳格, */
 textcolor(GREEN); /* 而是显示一个不常见的字符 */
 Writechar(65,0,1); /* 对文本方式,指定像素不起作用 */
 Writechar(66,0,2);
 Writechar(67,0,3);
 initgraph(&driver,&mode,"");
 cleardevice();
 setbkcolor(1);
 setcolor(RED);
 Writechar(65,0,3); /* 显示三个不同颜色的字母 A */
 Writechar(66,0,2);
 Writechar(67,0,1);
 getch();
 closegraph();
}
```

—15 功能 0FH: 获取当前 BIOS 的显示方式

#### (1) 入口参数

AH=0xf 功能号

#### (2) 返回值

AH= 每行显示的字符数

AL= 当前显示模式

BH= 视频可见页号

(3) 视频数据区变动部分：无

(4) 说明

设置显示方式用 INT 10H 的功能 0。如果设置显示方式时位 7 置位(“不闪动”),返回显示方式的位 7 也置位。

(5) 定义函数

C>TYPE INT10—18.C

```
#include "graphics.h"
```

```
#include "dos.h"
```

```
#include "conio.h"
```

```
struct videomode{
```

```
 int columns;
```

```
 int videomode;
```

```
 int page;
```

```
};
```

```
Getmode(struct videomode * V)
```

```
{ /* 获取当前 BIOS 显示模式 */
```

```
 union REGS r;
```

```
 r.h.ah=0xf;
```

```
 int86(0x10,&r,&r);
```

```
 V->columns=r.h.ah;
```

```
 V->videomode=r.h.al;
```

```
 V->page=r.h.bh;
```

```
}
```

```
main(){
```

```
 struct text—info T;
```

```
 struct videomode M;
```

```
 int driver=DETECT,mode;
```

```
 int M—col,M—video,M—page;
```

```
 Getmode(&M);
```

```
 M—col=M.columns;
```

```
 M—video=M.videomode;
```

```
 M—page=M.page;
```

```
 initgraph(&driver,&mode,"C:\\\\TC");
```

```
 gettextinfo(&T);
```

```
 printf("TEXT MODE:\nright window coordinate=%d\n",T.winright);
```

```
 printf("currmode=%d\n\n",T.currmode);
```

```
 printf("columns=%d videomode=%d page=%d\n\n",M—col,M—video,M—page);
```

```
 printf("GRAPH MODE:\nmaxgraphmode=%d\n",getmaxmode());
```

```
 printf("current graphmode=%d\n",getgraphmode());
```

```
 printf("graphdrivername=%s\n",getdrivername());
```

```
 printf("graphmodename=%s\n",getmodename(driver));
```

```
 Getmode(&M);
```

```
 printf("columns=%d videomode=%d page=%d\n",M.columns,
```

```

 M. videomode, M. page);
getch();
closegraph();
}
/* 从程序输出可以分清模式号的不同含意,可以避免混淆。
TEXT MODE, /* 在文本方式下 */
right window coordinate=80 /* 全屏幕作窗口时最右边列的值 */
currmode=3 /* 当前 BIOS 显示方式 */
columns=80 videomode=3 page=0 /* 当前 BIOS 显示方式是3,80×25彩色文本 */
GRAPH MODE, /* 在图形方式下 */
maxgraphmode=4
current graphmode=4 /* 当前的图形方式为4: 640×200 双色 */
graphdrivername=CGA /* 方式 4 是枚举 graphics—modes 中的值 */
graphmodename=320 X 200 CGA C1
columns=80 videomode=6 page=0 /* 当前的BIOS 显示方式为6: 640×200 双色 */
*/

```

—16 功能 10H: 设置调色板寄存器,控制边界颜色、灰度及闪烁属性等

本功能可以根据 AL 的值分成 16 种子功能,对 VGA 这些功能都能用;对 EGA 只能用子功能 0~3;对 MCGA 可用子功能是:0, 3, 10H, 12H, 15H, 17H, 18H, 19H 和 1BH。对其它卡不能用。所有子功能在文本及图形方式下均有效。其入口参数中必定有

AH=0x10 功能号

AL=10H 子功能号

视频数据变动部分是:0040:0065, 0040:0066。未指明返回值的,均表明无返回值。

0— 子功能 0: 更新指定的调色板寄存器

(1) 入口参数

BH= 颜色值

BL= 调色板号码

(2) 说明: 注意: BIOS 并不检查 BL 值的正确性。

1— 子功能 1: 设置边界颜色寄存器

(1) 入口参数

BH= 颜色值

2— 子功能 2: 更新所有 16 个调色板寄存器及边界颜色值寄存器

(1) 入口参数

ES:DX=17 个字节的表的地址(段: 偏移量)。前 16 个字节为调色板寄存器的颜色,最后一个字节为边界色。

(2) 说明

BIOS 会把表中的值写入到属性控制器对应的寄存器中。

3— 子功能 3: 选择背景亮度或闪烁属性

BL=0 (背景增亮但不闪烁); 为 1 时闪烁

4— 子功能 4: BIOS 保留,未用

5— 子功能 5: BIOS 保留,未用

6— 子功能 6: BIOS 保留,未用

7— 子功能 7: 读取指定的调色板寄存器中的值

(1) 入口参数

BL= 调色板寄存器号码

(2) 返回值: 在 BH 中

8— 子功能 8: 读取边界颜色寄存器中的值

返回值: 在 BH 中

9— 子功能 9: 读取所有 16 个调色板寄存器和边界寄存器中的值

(1) 入口参数

ES:DX= 存 17 个字节的表的地址

(2) 返回值: 在 ES:DX 中, 其中 0 ~ 15 字节为调色板值, 16 字节为边界值。

10— 子功能 10H: 更新指定的视频 DAC(模拟数字转换器)颜色寄存器

(1) 入口参数

BX= 颜色寄存器号(0~255)

CH= 绿色值(0 ~ 63)

CL= 兰色值(0 ~ 63)

DH= 红色值(0 ~ 63)

(2) 说明

对 MCGA 和 VGA, BIOS 将红、绿、兰值存入 DAC 的颜色寄存器中, 颜色值只有最低 6 个位有效。

(3) 函数

C>TYPE INT10--19.C

```
#include "dos.h"
```

```
main()
```

```
{
```

```
union REGS rg;
```

```
int p[] = { 0x0, 0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F,
 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x14, 0x7,
 0x0};
```

```
/* int k; */
```

```
rg.h.ah = 0x10;
```

```
rg.h.al = 0;
```

```
rg.h.bh = 6;
```

```
rg.h.bl = 7;
```

```
int86(0x10, &rg, &rg);
```

```
cputs("palette new color=6=yellow");
```

```
getch();
```

```
printf("palette registr number=7\n");
```

```
getch();
```

```
rg.h.ah = 0x10;
```

```
rg.h.al = 1;
```

```
rg.h.bh = 1;
```

```
int86(0x10, &rg, &rg);
```

```
cputs("color value for overscan=1=blue");
```

```
getch();
```

```

printf("palette registr number=11H"),
getch();
—ES=FP—SEG(p);
/* for(k=0;k<17;k++){ */
rg.h.ah=0x10;
rg.h.al=2;
rg.x.dx=FP—OFF(p);
int86(0x10,&rg,&rg);
/* } */
cputs("AA palette");
getch();
printf("BB palette registr");
getch();
}

```

对 EGA 在集成环境下调试可以看到颜色的变化。

11— 子功能 12H: 更新一组视频 DAC 颜色寄存器

(1) 入口参数

BX= 第一个开始更新的寄存器, 范围为 0 ~ 0FFH

CX= 要连续更新的寄存器个数

ES,DX= 红、绿、兰值表的地址

(2) 说明

由于 BIOS 不检查错误, 因此若 BX 与 CX 的和大于 256 时, 则大于的数相当于减去 256 后的值 (这种现象可称为回卷)。

表中每一项 (寄存器) 占三个字节, 三个字节依次为红、绿、兰, 每一个字节只有最低 6 位有效。适用于 VGA/MCGA 系统, 下同。

12— 子功能 13H: 设置视频 DAC 彩色页面

(1) 入口参数

BL=0 设置页面模式

BH=00H 选 4 组 64 色

01H 选 16 组 16 色

或

BL=1 选页

BH= 页号 (00H ~ 03H) 或 (00H ~ 0FH)

13— 子功能 15H: 读取指定视频 DAC 颜色寄存器值

(1) 入口参数

BX= 颜色寄存器的号码

(2) 返回值

CH= 绿

CL= 兰

DH= 红

14— 子功能 17H: 读取一组视频 DAC 颜色寄存器值

(1) 入口参数

BX= 第一个颜色寄存器的号码

CX= 读取寄存器个数

ES:DX= 红、绿、兰表的地址

(2) 返回值

ES:DX 表中的字节  $0 \sim 3 * n - 1$  (这里  $n$  是指定的 CX 值) 为指定寄存器中的红、绿、兰值。

15— 子功能 18H: 更新视频 DAC 的 Mask 寄存器的值

(1) 入口参数

BL= 新 mask(掩模) 值

16— 子功能 19H: 读取视频 DAC 的 Mask 寄存器的值

(1) 返回值

BL=DAC 的 mask 值

17— 子功能 1AH: 取视频 DAC 彩色页面状态

(1) 返回值

BL=0 选页面模式

BH=00H 选 4 组 64 色

01H 选 16 组 16 色

或

BL=1 选页方式

BH= 页号

18— 子功能 1BH: 设置一组颜色寄存器的亮度比例组合 (求灰度和)

(1) 入口参数

BX= 第一个颜色寄存器

CX= 要转换的颜色寄存器数

(2) 说明

根据 DAC 寄存器的当前彩色转换成等效的灰度级别

19— 子功能 F1H: 取 DAC 类型

返回值

BL=00H 正常 VGADAC

01H Sierra SC1148x H:Color DAC

否则为其它 HiColor DAC

—17 功能 11H: 装载 ASCII 码字符集 / (字符发生器介面)

入口参数中一定有

AH=11H

AL=子功能号

子功能 01H ~ 04H 是文本方式字符发生器功能, 修改文本显示器的字体和大小。它们设定一个模式, 使视频环境完全复位, 但不清视频缓冲区; 子功能 20H ~ 29H 是设定图形方式下文本字符的字体和显示大小。适用于装有 EGA、VGA 或 MCGA 系统。

对 MCGA, 子功能 0、1、2、4 和 10H、11H、12H、14H 是一样的, 它们都将文本字符装入视频 RAM 中; 但对 EGA 和 VGA 就有一些区别。不同之处在于前者只装入文本方式字符定义表 (供字符发生器用), 而后者除此外 BIOS 还要设置 CRTC 的值, 以便能处理字符点阵的新高度。

对 MCGA 还要注意两点：一是在执行这 4 种功能中任何一种之后要置  $AL=3$  后，才能调用功能  $11H$ 。其次，MCGA 的 CRTC 只能显示高度为  $2 \sim 16$  中的偶数条线高的字符，所以 BH 必取这一范围内的偶数值。同时，BIOS 会将取 14 条线的值自动补成 16 条线。

视频数据区变动部分为：

0040:004C、0040:0060、0040:0084、0040:0085。

0—子功能 0(或  $10H$ )：装用户定义的字符定义表

(1) 入口参数

BH= 点数(每个字符占多少字节)

BL= RAM 字符发生器中的那一个表

CX= 表中的字符个数

DX= 第一个字符的 ASCII 码

ES:BP= 表的地址

1—功能 1(或  $11H$ )：ROM BIOS  $8 \times 14$  的字符

BL=RAM 字符发生器中的那一个表

2—子功能 2(或  $12H$ )：ROM BIOS  $8 \times 8$  的字符

BL=RAM 字符发生器中的那一个表

3—子功能 3：选择显示用的字符发生器的那一个表

(1) 入口参数

BL= 对 EGA 和 VGA 是 Character Map Select 寄存器的值

对 MCGA 是 RAM 字符发生器中表的号码

(2) 说明

使用 EGA 或 MCGA 时，若字符属性字节的位 3 为 0，则 BL 的位 0 和位 1 代表要用四个由 256 个字符组成的字符定义表中的一个；若其值为 1，则 BL 的位 2 和位 3 决定用哪一个表。

使用 VGA 时，若字符属性字节的位 3 为 0，则位 0～位 3 代表要用 8 个字符定义表中的哪一个；若其值为 1，则位 2、位 3 和位 5 代表要用哪一个表。

4—子功能 4(或  $14H$ )：ROM BIOS  $8 \times 16$  的字符

(1) 入口参数

BL=RAM 字符发生器中的那一个表

对 EGA 不能用；对 MCGA 由于 BIOS 不含  $8 \times 14$  字符定义表，故它和  $AL=1$  是一样的。

5—子功能  $20H$ ：装入用户定义  $8 \times 8$  图形字符(相当于中断 INT  $1FH$ )

(1) 入口参数

ES:BP= 用户字符定义表的地址

(2) 说明

将 ES:BP 中的地址值写入存在  $0000:007C$  的中断向量  $1FH$ 。向量指向 ASCII 码为  $80H \sim FFH$  的  $8 \times 8$  字符表；CGA 在  $320 \times 200$  四色及  $640 \times 200$  双色图形方式中都会用到这张表。

6—子功能  $21H$ ：用户图形字符定义表

(1) 入口参数

BL= 0 (见 DL)



- 1 屏幕共 14 行字符
- 2 屏幕共 25 行字符
- 3 屏幕共 43 行字符
- CX= 每个字符占多少个字节
- DL= 屏幕上有多少字符行 (当 BL=0 时)
- ES:BP= 用户字符定义表的地址

(2) 说明

本子功能和 22H、23H、24H 子功能类似于子功能 0、1、2、4，BIOS 会更新 INT 43H 的向量值及变更视频数据区中 0040:0084 和 0040:0085 的值。在装入图形方式的字符定义表时，不会重置 CRTC 的参数值。

7— 子功能 22H: ROM 8 × 14 图形字符

(1) 入口参数

DL=00 由用户用 DL 规定行数

- 01 14 行
- 02 25 行
- 03 43 行

8— 子功能 23H: ROM 8 × 16 图形字符

(1) 入口参数

同子功能 22H

9— 子功能 24H: 装入 8 × 16 图形字符 (VGA/MCGA)

(1) 入口参数

同子功能 22H

10— 子功能 29H: 装入 8 × 16 图形字符 (Compaq Systempro)

(1) 入口参数

同子功能 22H

11— 子功能 30H: 获取目前字符发生器的有关字体信息

(1) 入口参数

BH=0 INT 1FH 向量值

- 1 INT 43H 向量值
- 2 ROM 8 × 14 字符表定义地址
- 3 ROM 8 × 8 字符表定义地址
- 4 ROM 8 × 8 字符表定义第二部分的地址
- 5 第二 ROM 9 × 14 字符表定义表地址
- 6 ROM 8 × 16 字符表定义表地址 (VGA, MCGA)
- 7 第二 ROM 9 × 16 字符表定义表地址 (VGA)

(2) 返回值

- CX= 字符点阵高度 (每字符占用字节数)
- DL= 显示行数减 1
- ES:BP= 字符定义表地址

(3) 说明

由 BH 值可知是哪一个表

注意:对 EGA 而令 BH=6 或 7,或对 MCGA 令 BH=5 或 7,则 ES:BP 值无效。

—18 功能 12H: 视频系统状态 (第二组)

它主要根据 BL 的值区分子功能。注意: 下面的入口参数中一定有

AH=12H

BL= 子功能号

0—子功能 10H: 获取视频系统状态

(1) 返回值

BH=0 (BIOS 显示方式为彩色) 或 1 (为单色)

BL=n 这里 n 可为 0 ~ 3, 分别表示 (EGA、VGA、MCGA) 的视频 RAM 的大小  
为  $n * 64K$  (字节)

CH= 特征字节, 位 0 ~ 位 3 来自 INFO—3 的位 4 ~ 位 7 (EGA 特性接头的输入)

CL= 组合开关值, 位 0 ~ 位 3 来自 INFO—3 的位 0 ~ 位 3

(2) 说明

返回的状态数据来自视频显示数据区的 INFO 和 INFO—3, 它是由 BIOS 初始化程序设置的。

1—子功能 20H: 选择另一个屏幕打印程序

(1) 入口参数

BL=20H (选替换屏幕打印例程)

(2) 说明

使用 EGA、MCGA 和 VGA 时, BIOS 会令 0000:0014 的中断向量 INT 5H 指向视频 ROM BIOS 内的另一个屏幕打印程序。装置视频卡 BIOS 的 PtrScr 例程来替默认的 ROM BIOSPtrScr 例程。该例程和 ROM BIOS 例程的区别在于前者根据视频显示 RAM 决定打印多少行文字, 而后者则一律印 25 行。注意, 有些显示卡是关断屏幕打印而不是改进它。

2—子功能 30H: 选择文本方式的扫描线数

(1) 入口参数

AL=0 200 条扫描线 (垂直分辨率)

1 350 条扫描线

2 400 条扫描线

(2) 返回值

正常返回值为 AL=12H, 否则 AL=0 (当 INFO—3 的位 3 为 1 时, 表示 VGA 无作用)

(3) 说明

使用 VGA 时, BIOS 会更新 INFO—3 (0040:0088) 的位 0 ~ 位 3、Flags (在 0040:0089) 字符的位 4 ~ 位 7。选择文本方式时, 在调用本功能后再调用 INT 10H 的功能 0 设置显示方式, 便可随意设置线数。

3—子功能 31H: 选择默认调色板装入

(1) 入口参数

AL=0 (装入默认调色板) 或 1 (不装)

(2) 返回值: 正常返回为 AL=12H

(3) 说明

使用 MCGA 或 VGA 时, BIOS 例程会更新 0040:0089 处的 Flags 子节的位 3 值, 以便

使用 INT 10H 的功能 0 设置显示方式时知道是否应装入内定的 ROM BIOS 的调色板。

4—子功能 32H: CPU 可否读视频 RAM 及 I/O 缓冲区的标志

(1) 入口参数

AL=0 (CPU 可读视频 RAM 及 I/O 缓冲区) 或 1 (不可读)

(2) 返回值: 正常返回 AL=12H

(3) 说明

EGA 没有本功能,但可更改其端口为 3C2H 的寄存器的位 1,就可以控制 CPU 能否读写 EGA 的视频 RAM。

5—子功能 33H: 使用亮度比例组合标志 (控制灰度求和的可用性)

(1) 入口参数

AL=0 (使用亮度比例组合标志,即允许灰度求和) 或 1 (不使用)

(2) 返回值: 正常返回 AL=12H

(3) 说明

对 MCGA 或 VGA, BIOS 会改变 0040:0089 处的 Flags 字节,以表示 INT 10H 功能 0 和功能 10H 更新视频 DAC 寄存器时,是否应计算红、绿、兰的平均值,以产生适当的亮度。

6—子功能 34H: 执行光标模拟标志 (光标仿真)

(1) 入口参数

AL=0 (允许字母数字光标仿真) 或 1 (不执行)

(2) 返回值: 正常返回 AL=12H

(3) 说明

使用 VGA 时, BIOS 会更新 INFO0 (0040:0087) 的位 0,表示是否执行光标模拟。BIOS 上电自测期间通过运行一个存储器容量检测程序确定显示存储器的尺寸,并把该值存在 0040:0087 地址开始的单元中,通过 BOIS 功能 IBH 可读出显示器容量。

7—子功能 35H: 视频显示的转换 (VGA/MCGA)

(1) 入口参数

AL=0 设置系统初始状态,使扩充显示卡不起作用

1 设置系统初始状态,使用主机板上的显示卡,设置 80 × 25 文本方式

2 使目前的视频显示系统不起作用

3 使不起作用的视频显示系统起作用

80H 设系统板视频活动标志

ES:DX=128 个字节保存区的地址 (如 AL=0、2、3)

(2) 返回值: 正常返回 AL=12H

(3) 说明

先执行 AL=0 或 1,然后再执行 AL=2 或 3。

8—子功能 36H: 视频刷新控制

(1) 入口参数

AL=0 (开视频,允许视频刷新) 或 1 (关视频,禁止视频刷新,显示变为空白)

(2) 返回值: 正常返回 AL=12H

(3) 说明

不执行视频更新时 (AL=1) 时会使频繁读写视频缓冲区的程序执行快些。

—19 功能 13H: 显示字符串

(1) 入口参数

AH=13H

AL 是写模式 AL 位 0=1 更新光标位置,即显示字符后移动光标

AL 位 1=1 串由字符和属性交替组成

BH= 视频页号

BL= 字符属性,如果显示串只含属性

CX= 字符串长度,即串中字符数

DH= 开始显示行 (row)

DL= 开始显示列 (column)

ES;BP= 字符串起始地址,即指向要写的串

(2) 返回值: 无

(3) 视频数据区变动部分: 0040;0050

(4) 说明

本功能将指定字符串写入视频缓冲区。响铃、退格、换行、回车等字符均看作控制字符,而不在屏幕上显示。当显示字符超过一行或一屏时,会自动换行或滚动屏幕。

在 320 × 200 四色方式中, BH 只能用 0 页; 在非 320 × 200 的 256 色图形方式下, 若将 BL 的属性值的位 7 设为 1, 则 BIOS 会以异或 (XOR) 的方式将字符串写入视频缓冲区内。

—20 功能 14H: 装入用户规定的液晶 (LCD) 显示字体

AH=14H

AL= 子功能号

0—子功能 0H: 改变字体

(1) 入口参数

BH= 每字符需要用字节数 (08H 或 10H)

BL=00H 装主字体 (块 0)

01H 装替换字体 (块 1)

CX= 要存储的字符数

DX=RAM 字体区的字符偏移值

ES;DI= 指向字体

(2) 返回: 无

1—子功能 1H: 装入系统默认的液晶字体

(1) 入口参数

BL=00H 装主字体 (块 0)

01H 装替换字体 (块 1)

(2) 返回: 无

2—子功能 2H: 设液晶高亮度属性的映射

(1) 入口参数

BL=00H 忽略高亮度属性

01H 映射高亮度为下划线

02H 映射高亮度为反显示

03H 映射高亮度为选择的替换字体

B0H 半亮度 (Compaq)

B1H 翻转有效高亮度位的解释 (Compaq)

—21 功能 1AH: 视频显示的配合方式

它的子功能以 AL 值区分。在入口参数中必有

AH=1AH

AL= 子功能号

视频数据区变动部分: 0040:008A

视频数据区内的 DCC(0040:008A) 保存了目前的视频显示的配合方式, 其值为 ROM BIOS 显示组合码表中的索引。组合码表中项为一对对数据, 代表哪两种视频系统可以混合使用。视频系统组合代码如下:

|     |                                         |
|-----|-----------------------------------------|
| F1  | BIOS 不认得的显示器                            |
| 0H  | 没有显示器                                   |
| 1H  | MDA 及单色显示器                              |
| 2H  | CGA 彩色显示器                               |
| 3H  | (保留)                                    |
| 4H  | EGA 彩色显示器                               |
| 5H  | EGA 单色显示器                               |
| 6H  | Professional Graphics Controller(PGA 卡) |
| 7H  | VGA 模拟单色显示器                             |
| 8H  | VGA 模拟彩色显示器                             |
| 9H  | (保留)                                    |
| 0AH | MCGA 数值式彩色显示器                           |
| 0BH | MCGA 模拟单色显示器                            |
| 0CH | MCGA 模拟彩色显示器                            |

0—子功能 0H: 获取视频显示配合方式

(1) 入口参数

BL= 使用的显示器代码

BH= 替换显示器代码(尚未使用的)

(2) 说明

使用 MCGA 或 VGA 时, 视频 BIOS 例程以 DCC 为索引, 把显示组合码表中的对应项写到 BH 和 BL 中。如果计算机内有两个视频系统, 则一个必须是单色, 另一个为彩色; BIOS 会根据 EQUIP—FLAG(0040:0010) 的位 4 和位 5 找出目前在用那一个视频系统。

(3) 返回值

AL=1AH 如果功能被支持

BH= 替换显示代码

BL= 活动显示代码

1—子功能 1H: 设置视频显示配合方式码

同子功能 0H

—22 功能 1BH: 视频 BIOS 功能及状态

(1) 入口参数

AH=1BH 功能号

BX=0 设计方式(必为 0)

ES:DI= 64 个字节的的状态信息缓冲区地址

## (2) 返回值

ES:DI= 功能及各种动态显示状态数据

AL=1BH 如果功能被支持

## (3) 说明

1. ES:DI 的 64 个字节的部分内容为：

表 35-16 ES:DI 的 64 个字节内容

| 偏移量     | 内 容 说 明                                                                                                                                                             |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0~3     | 存放固定数据表（或称静态功能表）的 32 位地址                                                                                                                                            |
| 4       | 正在起作用的显示方式                                                                                                                                                          |
| 5~6     | 列数                                                                                                                                                                  |
| 7~8     | 视频缓冲区显示部分的长度（单位：字节）                                                                                                                                                 |
| 9~AH    | 视频缓冲区左上角位置（起始地址）                                                                                                                                                    |
| 0BH~1AH | 8 个视频页（0~7）中光标的位置（每页为两个字节，一个字节为行，另一为列）                                                                                                                              |
| 1BH     | 光标终止线                                                                                                                                                               |
| 1CH     | 光标起始线                                                                                                                                                               |
| 1DH     | 目前使用的视频页                                                                                                                                                            |
| 1EH~1FH | CRTC 地址寄存器对应的输入 / 输出缓冲区（CRTC 口地址）                                                                                                                                   |
| 20H     | CRT 模式设置值（端口为 3x8H 寄存器的目前值）                                                                                                                                         |
| 21H     | CRT 调色板（端口为 3x9H 寄存器的目前值）                                                                                                                                           |
| 22H     | 目前每屏显示的行数                                                                                                                                                           |
| 23H~24H | 字符点阵的高度（每字符用字节数）                                                                                                                                                    |
| 25H     | 正在使用中的显示组合代码                                                                                                                                                        |
| 26H     | 没有使用的显示组合代码                                                                                                                                                         |
| 27H~28H | 当前模式支持的显示颜色数（单色为 0）                                                                                                                                                 |
| 29H     | 当前模式支持的可使用的视频页数                                                                                                                                                     |
| 2AH     | 值为 0、1、2、3 分别表示扫描线为 200 条、350 条、400 条、480 条                                                                                                                         |
| 2BH     | 基本字符                                                                                                                                                                |
| 2CH     | 二级字符块                                                                                                                                                               |
| 2DH     | 杂项，各种状态数据。当它的相应位为 0 时则没有所述功能<br>位 0=1 除 MCGA 为 0 外，其余均为 1<br>位 1=1 执行亮度比例组合<br>位 2=1 接了单色显示器<br>位 3=1 内定为不装入固定的调色板<br>位 4=1 执行光标模拟<br>位 5=1 使用闪烁属性<br>位 6~位 7 保留，未用 |
| 2EH~30H | 保留，未用                                                                                                                                                               |
| 31H     | 视频 RAM 大小。0、1、2、3 分别表示 64K、128K、192K、256K                                                                                                                           |
| 32H     | 保存区状态<br>位 0=1 同时用两组文本字符定义表（只用于 VGA）<br>位 1=1 使用了动态保护区<br>位 2=1 用户文本字符定义表在使用中<br>位 3=1 用户图形字符定义表在使用中                                                                |

位4=1 用户调色板在使用中  
 位5=1 显示组合代码在使用中  
 33H~3FH 保留,未用

## 2. 固定数据表的内容

表 35-17 固定数据表

| 偏移量 | 内 容 说 明                                                                                                                                                                             |
|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0   | 如位x=1,则表示可用的显示方式是x(字节能表示方式0~7)                                                                                                                                                      |
| 1   | 如位x=1,则表示可用的显示方式是8+x(字节能表示方式8~15)                                                                                                                                                   |
| 2   | 如位x=1,则表示可用的显示方式是16+x(字节能表示方式16~19)                                                                                                                                                 |
| 3~6 | 保留,未用                                                                                                                                                                               |
| 7   | 文本方式中扫描线数目。<br>位0=1表示200条、位1=1表示350条、位2=1表示400条                                                                                                                                     |
| 8   | 最多可显示几组文本字符定义表(或称字符块总数)                                                                                                                                                             |
| 9   | RAM字符发生器中有多少字符定义表(活动字符块最大数)                                                                                                                                                         |
| 0AH | 各种视频 BIOS 功能<br>位0=1 INT 10H 功能0设置的显示方式(但对MCGA永为0)<br>位1=1 执行亮度比例组合<br>位2=1 装入字符定义表<br>位3=1 装入内定调色板<br>位4=1 执行光标模拟<br>位5=1 64色调色板<br>位6=1 装入视频 DAC 的值<br>位7=1 以属性控制器上的颜色选择寄存器控制 DAC |
| 0BH | 位0=1 可用光笔功能<br>位1=1 储存/还原视频状态(INT 10H 功能1CH)<br>位2=1 闪烁/背景强度<br>位3=1 显示组合码<br>位4~位7 保留,未用<br>0CH~0DH 保留,未用                                                                          |
| 0EH | 保存区功能<br>位0 一组以上字符<br>位1 动态保存区<br>位2 用户文本字符定义表<br>位3 用户图形字符定义表<br>位4 用户调色板<br>位5 扩充的显示组合码<br>位6~位7 保留,未用                                                                            |
| 0FH | 保留,未用                                                                                                                                                                               |

—23 功能 1CH: 储存或还原视频 ROM BIOS 的状态(仅用于 VGA)  
 子功能按 AL 值划分。入口参数中一定有

AH=1CH

AL= 子功能号

执行本功能时, BIOS 可能改变目前视频状态。如果不想在调用本功能后改变视频状态, 可在执行本功能后应立即再执行子功能 2。

0—子功能 0: 获取储存或还原缓冲区大小

(1) 入口参数

CX= 要了解的状态: 位 0=1 了解视频硬件状态

位 1=1 了解视频 BIOS 数据区

位 2=1 了解彩色寄存器和 DAC 状态

位 3~位 7 保留, 未用

(2) 返回值

AL=1CH (入口参数中 CX 的最低三位中至少有一位为 1 时) 表示功能被支持

BX= 状态所需缓冲区大小 (以 64 个字节为单位)

1—子功能 1: 储存指定视频状态

(1) 入口参数

CX= 状态 (同子功能 0)

ES, BX= 缓冲区的地址

2—子功能 2: 还原指定状态

(1) 入口参数

CX= 状态 (同子功能 0)

ES, BX= 缓冲区的地址

## 35.8 图形驱动程序和字体转换工具 BGI OBJ. EXE

在启动图形系统时, 一般要调用 `initgraph()` 函数, 它将装入图形驱动程序, 并将系统置为图形模式; 如果用到 `settextstyle()` 函数, 则还要使用矢量字体程序。

TC 系统盘上有一些图形驱动程序 (程序对应于相应的显示适配器);

| 程序名         | 字节数  | registerbgidriver() 函数中 driver 的值 |                    |
|-------------|------|-----------------------------------|--------------------|
|             |      | 未用 /F 选项时                         | 用了 /F 选项时          |
| ATT.BGI     | 6269 | ATT—driver                        | ATT—driver—far     |
| CGA.BGI     | 6253 | CGA—driver                        | CGA—driver—far     |
| EGAVGA.BGI  | 5363 | EGAVGA—driver                     | EGAVGA—driver—far  |
| HERC.BGI    | 6125 | Herc—driver                       | Herc—driver—far    |
| IBM8514.BGI | 6665 | IBM8514—driver                    | IBM8514—driver—far |
| PC3270.BGI  | 6029 | PC3270—driver                     | PC3270—driver—far  |

和字体程序 (TC 提供五种字体, 缺省的 8x8 位图字体建立在图形系统中, 故不必专门装入);

| 程序名      | 字节数  | registerbgifont(*font) 函数中 *font 值 |                    |
|----------|------|------------------------------------|--------------------|
|          |      | 未用 /F 选项时                          | 用了 /F 选项时          |
| GOTH.CHR | 8560 | gothic—font                        | gothic—font—far    |
| LITT.CHR | 2138 | small—font                         | small—font—far     |
| SANS.CHR | 5438 | sansserif—font                     | sansserif—font—far |
| TRIP.CHR | 7241 | triplex—font                       | triplex—font—far   |



通常它们并不装入到用户编制的执行程序中,而是在用到的时候,去访问这些程序。我们把这种过程称为【动态驱动程序和字体装入法】。这些程序一般应在执行程序的当前目录中(或用 `initgraph()` 函数指定其所在目录),用户执行程序运行时将访问它们(因而要读盘)。如果访问不到,则执行程序将中止运行,并显示像

`initgraph error : Device driver file not found (CGA. BGI)`

这样的错误(在初始化图形系统时未找到图形驱动程序 `CGA. BGI`)。

对于使用频繁而内存空间又允许时,可以使用 `BGI OBJ` 将它们预先直接装入到用户执行程序中,免去了反复访问磁盘和专门指定路径的过程。注意,由于将它们装入执行程序,所以执行程序长度将增加。

### 一 语法

在 DOS 提示符下键入 `C>BGI OBJ`

屏幕显示

```
BGI to OBJ Converter Version 2.0 Copyright (c) 1987,1988 Borland International
Usage: BGI OBJ [/F] <source> <destination[. OBJ]> <public name>
(用法) <segment-name> <segment-class>
The <source> parameter is required, the rest is optional.
/F selects 'far' version (please read the documentation before using /F).
```

Examples(例子)

```
 BGI OBJ GOTH
 BGI OBJ /F CGA
 BGI OBJ HERC. BGI HERCDRV -HERC-fdriver HERC-TEXT
```

注意:除 `<source>` 为必选项外,其余为任选项。由于各选项之间是用空格符分隔的,因此如果要指定一个任选项,则其前面的所有选项应列出!

■ `/F` selects 'far' version (please read the documentation before using `/F`).

按照缺省规定, `BGI OBJ. EXE` 建立的目标文件均使用同一个段,段名为 `-TEXT`。而用了选项 `/F` 或 `-F`,表示 `BGI OBJ. EXE` 使用一个不是 `-TEXT`(缺省)的段名。之所以这样做是考虑到执行程序装入多个这样的目标文件时如都用同一个段名(缺省值 `-TEXT`),那么这个段就会负担太重!使用了 `/F` 选项就可使多个目标文件装入不同的段中。

值得注意的是,这对于微型模式(tiny)可能有些问题,因为各段(`CS, DS, SS, ES`)均用相同的地址值(即 `CS=DS=SS=ES`),为此必须去掉一些或所有的驱动程序和字体文件,而改用动态驱动程序和字体装入法。具体应用见下面例。

■ The `<source>` parameter is required, the rest is optional.

`<source>` 是必选的(其余部分是任选的),它是待转换的图形驱动程序名(`*. BGI`)或字体程序文件名(`*. CHR`)。它们如果是上述文件,则键入时可省去扩展名;否则应用全名。

■ `<destination[. OBJ]>`

它是要产生的目标文件名,缺省为 `<source>. OBJ`。如果使用了 `/F` 选项,则缺省名为 `<source>F. OBJ`。

例如

C>BGIOBJ -F CGA

回车后屏幕显示

```
BGI to OBJ Converter Version 2.0 Copyright (c) 1987,1988 Borland International
6253 bytes from 'CGA.BGI' converted into 'CGAF.OBJ',
public name = '-CGA-driver-far', segment name = 'CGA-TEXT'.
```

则产生一个 CGAF.OBJ 文件,其 public name 用于填写 registerbgidriver() 函数中的 \* driver,而段名为 CGA-TEXT,而不是缺省名 -TEXT。同样, C>BGIOBJ /F SANS 显示

```
BGI to OBJ Converter Version 2.0 Copyright (c) 1987,1988 Borland International
5438 bytes from 'SANS.CHR' converted into 'SANSF.OBJ',
public name = '-sansserif-font-far', segment name = 'SANS-TEXT'.
```

生成一个 SANSF.OBJ 文件。而用

C>BGIOBJ CGA

则显示

```
BGI to OBJ Converter Version 2.0 Copyright (c) 1987,1988 Borland International
6253 bytes from 'CGA.BGI' converted into 'CGA.OBJ',
public name = '-CGA-driver'.
```

从这里看到 public name 的内容不同了,缺省段名 -TEXT 未显出!

■ <public name>

它是指用户执行程序调用 registerbgidriver() 函数和 registerbgifont() 函数时要用到的公共名字(\* driver 或 \* font 的值)。该公共名将告诉编译程序、连接程序连接目标代码。程序员可以使用自己定义的公共名,但由此必须要在你的执行源程序中加上语句

```
void public-name(void); /* 未使用 /F 选项时 */
```

或

```
extern int far public-name[]; /* 使用了 /F 选项时 */
```

注意:这里 public-name 即是用 BGIOBJ.EXE 转换时显示的 public name。当然,你也可像缺省的 public name 已在 graphics.h 标头文件中登记那样,将你的 public-name 登记到 graphics.h 中去(参见 graphics.h 中最后一段的内容)。这样就不必写入源文件了。

不管你是否用了 /F 选项,为了对装入的图形驱动程序或字体程序进行合法性检查,即对其进行【注册】,或者说,在源程序中要使用函数

```
registerbgidriver()和registerbgifont() /* 如未用 /F 选项,也未改变缺省段名 */
```

或用

```
registerfarbgidriver()和registerfarbgifont() /* 如用 /F 选项或改了缺省段名 */
```

对其检查,如代码有效则把它们登记到执行程序的内部表中。

在源程序中可用下述类似的语句进行检查:

```
if (registerbgidriver(CGA-driver)<0) exit(1);
/* 如果测试到一个内部驱动器号小于 0,出错!非正常退出。*/
```

或

```
if (registerbgifont(gothic-font)!=GOTHIC-FONT) exit(1);
/* 如果测试到一注册字体号跟要求的矢量字体号不同,出错! */
```

#### ■ <segment-name>

它是指一个任选的段名,缺省值是 —TEXT。如你用了 /F 选项,并指定了段名 segname,则产生段名 segname—TEXT; 如用了 /F 而未指定段名,则产生段 <source>—TEXT。

#### ■ <segment-class>

它是指一个任选的段类,缺省值为 —CODE。

### 二 将驱动程序和字体连入用户执行程序的方法

有三种方法:

#### 法1

因为驱动程序与字体实际要在图形方式下工作,因此可把它们通过 BGIOBJ 生成的目标文件 (.OBJ) 用 TLIB.EXE 连入库 GRAPHICS.LIB 中。然后在用户源文件中加入语句

```
#include <graphics.h>
```

例如,现把 CGA 图形驱动程序、哥特字体和三重字体转换为目标模块后连入到程序 MY.C 中。

第一步 用 BGIOBJ.EXE 把 TC 盘上三个二进制文件 CGA.BGI、GOTH.CHR 和 TRIP.CHR 转换为目标模块 CGA.OBJ、GOTH.OBJ 和 TRIP.OBJ:

```
C>bgiobj cga
C>BGIOBJ GOTH
C>BGIOBJ TRIP
```

第二步 用 TLIB.EXE 将这三个模块加入到库 GRAPHICS.LIB 中:

```
C>TLIB GRAPHICS +cga +goth +trip
```

注意:一般 GRAPHICS.LIB 中未包含任何驱动程序和字体模块。如果在用这种方法加入时出现类似

```
Warning: 'CGA' already in LIB, not changed!
```

表明 CGA 模块已装入库中,不能再装入! 要把它从库中消去可用

```
C>TLIB GRAPHICS -cga -goth -trip
```

如果库中无指定模块,则显示

```
Warning: 'CGA' not found in library
```

第三步 把这些文件登记到程序 MY.C 中。

```
C>TYPE MY.C
#include <graphics.h> /* 头文件中申报 CGA-driver, */
 /* gothic-font 和 triplex-font */
.....
if (registerbgidriver(CGA-driver)<0) exit(1); /* 注册并检查错误 */
if (registerbgifont(gothic-font)<0) exit(1);
if (registerbgifont(triplex-font)<0) exit(1);
.....
```

```
initgraph(...);
```

```
.....
```

#### 法2.

在上例中不将目标文件 CGA.OBJ、GOTH.OBJ 和 TRIP.OBJ 装入库 GRAPHICS.LIB，而是在集成环境中将它们写入一个规划文件（Project file）中，该文件中也包括 MY.C（内容同上）。然后再在集成环境里编译连接。

#### 法3

如不用集成环境，也可用 TCC.EXE 命令行或 TLINK.EXE 命令行的办法：

```
C>TCC MY GRAPHICS.LIB cga.obj goth.obj trip.obj
```

### 三 可能出现的错误或提示

1. .OBJ '%s' : unable to open file

.OBJ: 不能打开文件 s。

2 Abnormal program termination

非正常程序终结。

3 Arg list too big

参数表太大。

4 Attempted to remove current directory

试图消去当前目录。

5 Bad file number

错误的文件号。

6 Divide error

分解错误。

7 Enter BGIOBJ with no parameters to receive help.

为取得帮助应在输入 BGIOBJ 后不要输任何参数。

8 Error 0

错误 0。

9 Error reading input file : %s

错误地读输入文件 s。

10 Error writing output file : %s

错误地写输出文件 s。

11 Exec format error

执行格式错误。

12 F—far '%s' : unable to open file

不能打开文件。

13 Input file is too large.

输入文件太大。

14 Invalid access code

无效的接收码。

15 Invalid argument

无效参数。

16 Invalid data

无效数据。

17 Invalid environment

无效环境。

18 Invalid format

无效格式。

19 Invalid function number

无效功能号。

20 Invalid memory block address

无效存储块地址。

21 Maximum size = 65535 bytes.

最大尺寸为 65535 字节 (例如段长超过 64K 不允许)。

22 Memory arena trashed

存储区域要除去垃圾。如果未初始化初值的指针发生在某一函数处,而该函数尚未使用到的堆栈空间曾经被其它程序使用过,则这些空间内的数据可能依旧存在,它们并未清 0,因而可能对别的程序有害。因此,称这些空间内的数据为【垃圾, trashed】。

23 No more files

无更多文件。

24 No such device

无此设备。

25 No such file or directory

无此文件或目录。

26 Not enough memory

内存不够。

27 Not enough memory to run BGIOBJ.

无足够内存执行 BGIOBJ。

28 Not same device

无相同设备。

29 Path not found

路径未找到。

30 Permission denied

拒绝存取。

31 Please correct, or specify full source filename and public name.

请重指定完整源文件名和公共名。

32 Too many open files

文件打开太多。

33 Unknown source filename '%s'.

不认识的源文件。

34 out of disk space

磁盘空间不够。

35 print scanf, floating point formats not linked

浮点格式不能连接。

## 35.9 图形演示程序 BGIDEMO.C 注释

/\*

这是 Turbo C 提供给用户理解图形库函数的表演程序,它列举了 56 个图形函数的用法(另外还涉及了其它 14 个函数)。它在程序设计方面也很值得读者借鉴。为了帮助读者理解,下面对其给出详细注释。

使用命令行编译:

```
tcc bgidemo graphics. lib
```

也可以在集成环境中编译。但当注释内容太多时它们有区别:使用命令行编译没有问题,然而在集成环境中虽然可以进行文本编辑,但在编译时将出现

Out of memory

即发现内存不够用,从而无法编译下去。

\*/

```
#ifdef ——TINY—— /* 此三句相当于编译说明,本程序不能使用 tiny 存储模式 */
```

```
#error BGIDEMO will not run in the tiny model.
```

```
#endif /* 如在集成环境下选 Options/Compiler/Model/Tiny 或用命令行
```

```
tcc -mt bgidemo graphics. lib
```

编译,将出现错误

Error directive: BGIDEMO will not run in the tiny model

\*\*\* 1 errors in Compile \*\*\*

从而不能得到执行文件 \*/

```
#include <dos.h> /* 函数涉及的包含文件 */
```

```
#include <math.h>
```

```
#include <conio.h>
```

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

```
#include <stdlib.h>
```

```
#include <stdarg.h>
```

```
#include <graphics.h>
```

```
#define ESC 0x1b /* 定义 ESC 键 */
```

```
#define TRUE 1 /* 定义常数“真” */
```

```
#define FALSE 0 /* 定义常数“假” */
```

```
#define PI 3.14159 /* 定义圆周率 PI */
```

```
#define ON 1 /* 定义“要裁剪” */
```

```
#define OFF 0 /* 定义“不裁剪” */
```

/\* 说明字体 \*/

```
char * Fonts[] = { "DefaultFont" /* 缺省字体 */,
 "TriplexFont" /* 三重矢量字体 */,
 "SmallFont" /* 小号矢量字体 */,
 "SansSerifFont" /* 光滑无修饰字体 */,
 "GothicFont" /* 哥特矢量字体 */
```

```
};
```

/\* 说明线型 \*/

```

char *LineStyle[] = { "SolidLn", /* 实线 */ "DottedLn", /* 点线 */
 "CenterLn", /* 中心线 */ "DashedLn", /* 破折线 */
 "UserBitLn" /* 用户定义线 */ };

/* 说明填充模式 */
char *FillStyles[] = { "EmptyFill", /* 背景 */ "SolidFill", /* 实心 */
 "LineFill", /* 粗1 */ "LtSlashFill", /* 细 */
 "SlashFill", /* 粗/ */ "BkSlashFill", /* 更粗\ */
 "LtBkSlashFill", /* 粗\ */ "HatchFill", /* 正方格 */
 "XHatchFill", /* 斜方格 */ "InterleaveFill", /* 间隔线 */
 "WideDotFill", /* 稀疏点 */ "CloseDotFill" /* 密集点 */
 };

/* 说明文本显示方向 */
char *TextDirect[] = { "HorizDir" /* 从左到右 */ , "VertDir" /* 从上到下 */ };
/* 说明在水平方向文本与当前光标对齐方式 */
char *HorizJust[] = { "LeftText", /* 水平左对准 */
 "CenterText", /* 中心对准 */
 "RightText" /* 水平右对准 */ };

/* 说明在垂直方向文本与当前光标对齐方式 */
char *VertJust[] = { "BottomText", /* 底端 */
 "CenterText", /* 中心 */
 "TopText" /* 顶端 */ };

struct PTS { int x, y; }; /* 处理“顶点”的结构.顶点坐标值 */
/* 也可用 graphics.h 中的结构
struct pointtype { int x,y; } */

int GraphDriver; /* 图形适配器号:0,1~10,-1,见枚举 graphics-drivers */
int GraphMode; /* 图形方式值,见枚举 graphics-modes */
double AspectRatio; /* 屏幕像素的纵横因子比 */
int MaxX, MaxY; /* 屏幕最大分辨率(列,行) */
int MaxColors; /* 最大颜色号 */
int ErrorCode; /* 报告任何图形错误码 */
struct palettetype palette; /* 用于读调色板信息 */
void Initialize(void); /* 用到的函数原型 */
void ReportStatus(void);
void TextDump(void);
void Bar3DDemo(void);
void RandomBars(void);
void TextDemo(void);
void ColorDemo(void);
void ArcDemo(void);
void CircleDemo(void);
void PieDemo(void);
void BarDemo(void);
void LineRelDemo(void);
void PutPixelDemo(void);
void PutImageDemo(void);
void LineToDemo(void);

```

```

void LineStyleDemo(void);
void CRTModeDemo(void);
void UserLineStyleDemo(void);
void FillStyleDemo(void);
void FillPatternDemo(void);
void PaletteDemo(void);
void PolyDemo(void);
void SayGoodbye(void);
void Pause(void);
void MainWindow(char * header);
void StatusLine(char * msg);
void DrawBorder(void);
void changetextstyle(int font, int direction, int charsize);
int gprintf(int * xloc, int * yloc, char * fmt, ...);
int main() /* 程序执行总是从 main() 函数开始 */
{
 Initialize(); /* 初始化图形系统 */
 ReportStatus(); /* 报告初始化结果 */
 ColorDemo(); /* 显示颜色号对应的颜色 */
 if(GraphDriver==EGA || GraphDriver==EGALO || GraphDriver==VGA)
 PaletteDemo(); /* 显示调色板随机变化时的图形,只对 EGA 和 VGA 进行 */
 PutPixelDemo(); /* 写随机象点并将它们收起 */
 PutImageDemo(); /* 读象和写象表演 */
 Bar3DDemo(); /* 三维条形图 */
 BarDemo(); /* 直方图 */
 RandomBars(); /* 随机条形图 */
 ArcDemo(); /* 随机弓形图 */
 CircleDemo(); /* 随机圆 */
 PieDemo(); /* 馅饼图 */
 LineRelDemo(); /* 镶嵌图 */
 LineToDemo(); /* 定界圆上画多根弦 */
 LineStyleDemo(); /* 所有可用的标准线型 */
 UserLineStyleDemo(); /* 用户自定义线型 */
 TextDump(); /* 所有字符的各种字体 */
 TextDemo(); /* 对不同字体、不同放大因子时的文本 */
 CRTModeDemo(); /* 改变显示模式后的结果 */
 FillStyleDemo(); /* 所有标准填充模式 */
 FillPatternDemo(); /* 用户自定义填充模式 */
 PolyDemo(); /* 随机画和填充多边形 */
 SayGoodbye(); /* 关闭图形屏幕后结束演示 */
 closegraph(); /* 让系统返回到文本方式 */
 return(0); /* 主函数也可以返回一个值 */
}

void Initialize(void) /* 初始化图形系统,并报告出错信息 */
{
 int xasp, yasp; /* 用于存放图形纵横比 */

```



```

GraphDriver = DETECT; /* 自动测试机器上装的图形适配器类型,
 获得枚举 graphics-drivers 中对应值 */
initgraph(&GraphDriver, &GraphMode, ""); /* 初始化图形系统 */
ErrorCode = graphresult(); /* 读初始化结果 */
if(ErrorCode != grOk){ /* 当初始化有错误时执行下一语句 */
 printf(" Graphics System Error: %s\n", grapherrormsg(ErrorCode));
 exit(1); /* 打印出出错原因(简短英文说明) */
}

getpalette(&palette); /* 将调色板信息装入结构 palette 中 */
MaxColors = getmaxcolor() + 1; /* 最大可用颜色号 */
MaxX = getmaxx(); /* 最大 x 坐标,列数 */
MaxY = getmaxy(); /* 最大 y 坐标,行数 */
getaspectratio(&xasp, &yasp); /* 将当前图形模式的纵横比存入变量 */
AspectRatio = (double)xasp / (double)yasp; /* 得到纵横比 */
}

/* 在自动检测与初始化后报告当前系统配置情况 */
void ReportStatus(void)
{
 /* 查询中要用的一些参数 */
 struct viewporttype viewinfo; /* 视区信息 */
 struct linesettingstype lineinfo; /* 当前线型 */
 struct fillsettingstype fillinfo; /* 图形填充信息 */
 struct textsettingstype textinfo; /* 文本信息 */
 struct pallettetype palette; /* 调色板 */
 char *driver, *mode; /* 指向图形驱动程序名 (*.BGI) 与模式名的字符串 */
 int x, y;

 getviewsettings(&viewinfo); /* 取相应信息装入对应结构中 */
 getlinesettings(&lineinfo);
 getfillsettings(&fillinfo);
 gettextsettings(&textinfo);
 getpalette(&palette);
 x = 10; y = 4; /* 初始坐标值:(列=10,行=4) */
 MainWindow("Status report after InitGraph"); /* 显示标题 */
 setttextjustify(LEFT-TEXT, TOP-TEXT);
 /* 文本在水平方向以左对齐,垂直方向以顶端对齐 */
 driver = getdrivename(); /* 获取图形驱动程序名,如 EGAVGA,即EGAVGA.BGI */
 mode = getmodename(GraphMode); /* 如 640x480 VGA,即模式名 */
 /* 格式符 %-20s 中负号表示字符左对齐,20 表示最多印出 20 个字符 */
 gprintf(&x, &y, "Graphics device : %-20s (%d)", driver, GraphDriver);
 gprintf(&x, &y, "Graphics mode : %-20s (%d)", mode, GraphMode);
 gprintf(&x, &y, "Screen resolution: (0, 0, %d, %d)", getmaxx(), getmaxy());
 /* 屏幕分辨率 0,0,最大列,最大行 */
 gprintf(&x, &y, "Current view port : (%d, %d, %d, %d)", viewinfo.left,
 viewinfo.top, viewinfo.right, viewinfo.bottom); /* 当前视口大小 */
 gprintf(&x, &y, "Clipping : %s", viewinfo.clip ? "ON" : "OFF");
 /* 视口是否会剪切.ON 表示输出时会剪切,此时 viewinfo.clip=1 */
 gprintf(&x, &y, "Current position: (%d, %d)", getx(), gety());
}

```

```

 /* 当前光标位置.注意:一般在图形方式下光标是不可见的 */
gprintf(&x, &y, "Colors available : %d", MaxColors); /* 可变最大颜色数 */
gprintf(&x, &y, "Current color : %d", getcolor()); /* 当前用的绘图色 */
gprintf(&x, &y, "Line style : %s", LineStyles[lineinfo.linestyle]);
 /* 线型 */
gprintf(&x, &y, "Line thickness : %d", lineinfo.thickness);
 /* 线宽 */
gprintf(&x, &y, "Current fill style : %s", FillStyles[fillinfo.pattern]);
 /* 当前填充模式 */
gprintf(&x, &y, "Current fill color : %d", fillinfo.color);
 /* 当前填充颜色号 */
gprintf(&x, &y, "Current font : %s", Fonts[textinfo.font]);
 /* 当前字体 */
gprintf(&x, &y, "Text direction : %s", TextDirect[textinfo.direction]);
 /* 文本显示方向 */
gprintf(&x, &y, "Character size : %d", textinfo.charsize);
 /* 字符放大因子 */
gprintf(&x, &y, "Horizontal justify : %s", HorizJust[textinfo.horiz]);
 /* 水平对齐方式 */
gprintf(&x, &y, "Vertical justify : %s", VertJust[textinfo.vert]);
 /* 垂直对齐方式 */
Pause(); /* 暂停,等待用户读屏幕内容,根据屏幕底下显示的状态行操作 */
}

void TextDump() /* 所有字符的各种字体的显示 */
{
 static int CGASizes[] = { 1, 3, 7, 3, 3 };
 static int NormSizes[] = { 1, 4, 7, 4, 4 };
 char buffer[80]; /* 缓冲区 */
 int font, ch, wwidth, lwidth, size;
 struct viewporttype vp;
 for(font=0; font<5; ++font) /* 对各种不同的字体号:0~4 */
 { /* 将图幅标题送缓冲区 */
 sprintf(buffer, "%s Character Set", Fonts[font]);
 MainWindow(buffer); /* 显示图幅标题 */
 getviewsettings(&vp); /* 获得当前视口信息 */
 settextrjust(LEFT-TEXT, TOP-TEXT); /* 文本对准:左,顶 */
 moveto(2, 3); /* 将图形光标从当前位置移到第 2 列、第 3 行 */
 buffer[1] = '\0'; /* 置终止符,为用 outtext() 作准备 */
 wwidth = vp.right - vp.left; /* 确定当前视口宽度 */
 lwidth = textwidth("H"); /* 得到正常字符宽度 */
 if(font == DEFAULT-FONT) /* 缺省字体,即 font=0 时 */
 { /* 置字体、文本水平对齐、放大因子为 1 */
 changetextstyle(font, HORIZ-DIR, 1);
 ch = 0; /* 从 ASCII 码 0 开始 */
 while(ch < 256) /* ASCII 码 0~255,即对每一个可能的字符 */
 { /* 注意:每一字符在屏幕上显示是没有问题的 */

```

```

 buffer[0] = ch; /* 将字符放到缓冲区首字节中 */
 outtext(buffer); /* 将字符送屏幕输出 */
 if((getx() + lwidth) > wwidth) /* 判断列是否超出视口 */
 moveto(2, gety() + textheight("H") + 3); /* 超出时从下一行显示 */
 ++ch; /* 转向下一个字符 */
}
}
else(/* 对非缺省字体显示 */
 size = (MaxY < 200) ? CGASizes[font] : NormSizes[font];
 /* 由最大可显示行数 MaxY 确定放大因子,对 CGA 是 199 */
 changetextstyle(font, HORIZ—DIR, size);
 /* 放大后字符显示为 size * 8xsize * 8 */
 ch = '!'; /* 从第一个可在打印机上打印的字符开始 */
 while(ch < 127) /* 对每一个可打印字符处理 */
 {
 buffer[0] = ch; /* 将要显示字符放入缓冲区内 */
 outtext(buffer); /* 将字符送屏幕显示 */
 if((lwidth+getx()) > wwidth) /* 列坐标是否还在视口内? */
 moveto(2, gety()+textheight("H")+3); /* 超出时换行 */
 ++ch; /* 取下一个字符 */
 }
}
Pause(); /* 暂停显示,直到操作者应答 */
} /* 结束字体显示循环 */
}

void Bar3DDemo(void) /* 显示三维条形图 */
{
 static int barheight[] = {1,3,5,4,3,2,1,5,4,2,3}; /* 与条形高度相关 */
 struct viewporttype vp;
 int xstep, ystep;
 int i, j, h, color, bheight;
 char buffer[10]; /* 缓冲区 */
 MainWindow("Bar 3-D / Rectangle Demonstration");
 h = 3 * textheight("H"); /* h 等于 3 倍字高 */
 getviewsettings(&vp);
 setttextjustify(CENTER—TEXT, TOP—TEXT); /* 水平中心对准,垂直从上往下 */
 changetextstyle(TRIPLEX—FONT, HORIZ—DIR, 4); /* 三重矢量字体,水平,4 倍 */
 outtextxy(MaxX/2, 6, "These are 3-D Bars"); /* 第 6 行输出字符串 */
 changetextstyle(DEFAULT—FONT, HORIZ—DIR, 1); /* 缺省字体、水平摆放、放大1 */
 setviewport(vp.left+50, vp.top+40, vp.right-50, vp.bottom-10, 1);
 getviewsettings(&vp);
 line(h, h, h, vp.bottom-vp.top-h); /* 画左边竖线 */
 line(h, (vp.bottom-vp.top)-h, (vp.right-vp.left)-h, (vp.bottom-vp.top)-h);
 /* 在屏幕底下画横线 */
 xstep = ((vp.right-vp.left) - (2 * h)) / 10; /* 列方向步长 */
 ystep = ((vp.bottom-vp.top) - (2 * h)) / 5; /* 行方向步长 */
}

```

```

j = (vp.bottom - vp.top) - h; /* 实际可用高度 */
settextjustify(CENTER-TEXT, CENTER-TEXT); /* 水平和垂直方向均中心对准 */
for(i=0 ; i<6 ; ++i) /* 从屏幕底下往上在已画竖线上画短横线和填纵坐标 */
{
 line(h/2, j, h, j); /* 画一水平短横线 */
 itoa(i, buffer, 10); /* 把整数 i 转成十进制字符串放入 buffer 中 */
 outtextxy(0, j, buffer); /* 将 i 值输出到竖线左边 */
 j -= ystep; /* 改变行值 */
}
j = h;
settextjustify(CENTER-TEXT, TOP-TEXT); /* 水平中心对准,垂直方向以顶端对准 */
for(i=0 ; i<11 ; ++i) /* 从左到右画短竖线、横坐标,并画出三维条形图 */
{
 color = random(MaxColors); /* 取随机颜色 */
 setfillstyle(i+1, color); /* 设置填充模式和颜色 */
 line(j, (vp.bottom - vp.top) - h, j, (vp.bottom - vp.top - 3) - (h/2));
 /* 画一根短竖线 */
 itoa(i, buffer, 10); /* 将 i 转为字符 */
 outtextxy(j, (vp.bottom - vp.top) - (h/2), buffer); /* 在短竖线底下印出 i 值 */
 if(i != 10)
 {
 bheight = (vp.bottom - vp.top) - h - 1; /* 计算三维条形图底高 */
 bar3d(j, (vp.bottom - vp.top - h) - (barheight[i] * ystep),
 j+xstep, bheight, 15, 1);
 }
 /* 画三维条形图,深度为 15 个象素,最后一个参数值为 1 表示要放一个三维顶 */
 j += xstep; /* 计算横向移动量 */
}
Pause();
}

void RandomBars(void) /* 显示随机条形图 */
{
 int color;
 MainWindow("Random Bars");
 StatusLine("Esc aborts or press a key..."); /* 在屏幕底行显示操作信息 */
 while(!kbhit()) /* 当用户没有按任一键时显示随机条形图 */
 {
 color = random(MaxColors-1)+1; /* 取随机颜色 */
 setcolor(color); /* 设置当前绘图色 */
 setfillstyle(random(11)+1, color); /* 设置随机填充模式与颜色 */
 bar3d(random(getmaxx()), random(getmaxy()),
 random(getmaxx()), random(getmaxy()), 0, OFF);
 /* 以当前光标坐标取随机数,画无三维顶的三维条形图,其实就是二维图形 */
 }
 Pause();
}

```

```
void TextDemo(void) /* 对不同字体、不同放大因子的文本显示 */
```

```
{
 int charsize[] = { 1, 3, 7, 3, 4 }; /* 字符放大因子 */
 int font, size;
 int h, x, y, i;
 struct viewporttype vp;
 char buffer[80]; /* 缓冲区 */
 for(font=0; font<5; ++font) /* 每次取四种字体中的一种 */
 {
 sprintf(buffer, "%s Demonstration", Fonts[font]);
 MainWindow(buffer);
 getviewsettings(&vp);
 changetextstyle(font, VERT-DIR, charsize[font]);
 /* 设置字体、垂直方向摆放、放大因子 */
 setttextjustify(CENTER-TEXT, BOTTOM-TEXT);
 /* 水平中心对准,垂直方向从底往上 */
 outtextxy(2 * textwidth("M"), vp.bottom - 2 * textheight("M"), "Vertical");
 /* 输出串 Vertical */
 changetextstyle(font, HORIZ-DIR, charsize[font]); /* 改成水平方向 */
 setttextjustify(LEFT-TEXT, TOP-TEXT); /* 水平从左到右,垂直从上往下 */
 outtextxy(2 * textwidth("M"), 2, "Horizontal"); /* 输出 Horizontal */
 setttextjustify(CENTER-TEXT, CENTER-TEXT); /* 水平或垂直均以中心对准 */
 x = (vp.right - vp.left) / 2; /* 决定列坐标 */
 y = textheight("H");
 for(i=1; i<5; ++i) /* 对同种字体的各种放大因子 size 的处理 */
 {
 size = (font == SMALL-FONT) ? i+3, i; /* SMALL-FONT=2, 小号矢量字体 */
 changetextstyle(font, HORIZ-DIR, size); /* 文本水平摆放 */
 h = textheight("H");
 y += h; /* 决定行坐标 */
 sprintf(buffer, "Size %d", size); /* 将字符串 Size 和 size 的值送入缓冲区,而 size 无须进行转换 */
 outtextxy(x, y, buffer); /* 将缓冲区中内容输出到屏幕 */
 }
 if(font != DEFAULT-FONT) /* 如果是缺省字体 */
 {
 /* 显示用户怎样说明字体放大因子 */
 y += h / 2; /* 行坐标下移 */
 setttextjustify(CENTER-TEXT, TOP-TEXT); /* 水平中心对准,垂直以顶为准 */
 setusercharsize(5, 6, 3, 2); /* 新文本宽度 = 5/6 * 缺省宽度
 高度 = 3/2 * 缺省高度 */
 changetextstyle(font, HORIZ-DIR, USER-CHAR-SIZE); /* 注意此句写法 */
 outtextxy((vp.right - vp.left) / 2, y, "User Defined Size"); /* 输出串 */
 }
 Pause();
 }
}
```

/\* 显示能在当前屏幕上显示的 15 种颜色, 每种颜色用一个有颜色的方块表示方块底下中间位置显示颜色号 \*/

void ColorDemo(void)

```
{
 struct viewporttype vp;
 int color, height, width;
 int x, y, i, j;
 char cnum[5];
 MainWindow("Color Demonstration");
 color = 1;
 getviewsettings(&vp);
 width = 2 * ((vp.right+1) / 16); /* 得每一种颜色显示方块的宽度 */
 height = 2 * ((vp.bottom-10) / 10); /* 方块高度 */
 x = width / 2;
 y = height / 2; /* 取二分之一方块高度 */
 for(j=0; j<3; ++j){ /* 行循环, 共 3 行 */
 for(i=0; i<5; ++i){ /* 列循环, 每行有 5 个方块 */
 setfillstyle(SOLID_FILL, color); /* 设置实心填充模式 */
 setcolor(color); /* 边框颜色和填充颜色一样 */
 bar(x, y, x+width, y+height); /* 将矩形内部用当前模式和颜色填充 */
 rectangle(x, y, x+width, y+height); /* 画矩形轮廓 */
 if(color == BLACK){ /* 如果颜色是黑色 */
 setcolor(WHITE); /* 置当前绘图色为白色 */
 rectangle(x, y, x+width, y+height); /* 用白色画矩形轮廓 */
 }
 itoa(color, cnum, 10); /* 把颜色号 color 转换成字符串 cnum */
 outtextxy(x+(width/2), y+height+4, cnum); /* 显示颜色号 */
 color = ++color % MaxColors; /* 取下一个颜色号 */
 x += (width / 2) * 3; /* 移动列基点 */
 }
 y += (height / 2) * 3; /* 移动行基点 */
 x = width / 2; /* 恢复列基点 */
 }
 Pause();
}
```

void ArcDemo(void) /\* 显示随机弓形图 \*/

```
{
 int mradius; /* 允许的最大半径 */
 int eangle; /* 随机的圆弧终止角 */
 struct arccoordstype ai; /* 用于读圆弧坐标信息 */
 MainWindow("Arc Demonstration");
 StatusLine("ESC Aborts - Press a Key to stop"); /* 底行显示的状态行 */
 /* 压 ESC 键终止 - 压其它键停止随机显示 */
 mradius = MaxY / 10; /* 决定最大半径 */
 while(!kbhit()){ /* 反复显示, 直到按一键才停止 */
 setcolor(random(MaxColors - 1) + 1); /* 随机选择颜色 */

```

```

eangle = random(358) + 1; /* 选择一个终止角 */
arc(random(MaxX), random(MaxY), random(eangle), eangle, mradius);
/* 画一条圆弧 */
getarccoords(&ai); /* 将刚刚调用 arc() 后的坐标信息填到 ai 中 */
line(ai.x, ai.y, ai.xstart, ai.ystart); /* 从起点到圆心画直线 */
line(ai.x, ai.y, ai.xend, ai.yend); /* 从终点到圆心画直线 */
} /* 当没有按键时又开始新一轮的循环 */
Pause();
}

void CircleDemo(void) /* 显示随机圆 */
{
 int mradius; /* 允许的最大半径 */
 MainWindow("Circle Demonstration");
 StatusLine("ESC Aborts -- Press a Key to stop"); /* 底行显示的状态行 */
 mradius = MaxY / 10; /* 决定的最大半径 */
 while(!kbhit()) { /* 反复循环,直到按一键后停止 */
 setcolor(random(MaxColors - 1) + 1); /* 选择随机颜色 */
 circle(random(MaxX), random(MaxY), random(mradius)); /* 画随机圆 */
 }
 Pause();
}

#define adjasp(y) ((int)(AspectRatio * (double)(y))) /* 调整纵向比率 */
#define torad(d) ((double)(d) * PI) / 180.0 /* 将角度 d 变成弧度 */
void PieDemo(void) /* 显示馅饼图 */
{
 struct viewporttype vp;
 int xcen, ycen, radius, lradius;
 int x, y;
 double radians, piesize;
 MainWindow("Pie Chart Demonstration");
 getviewsettings(&vp);
 xcen = (vp.right - vp.left) / 2; /* 水平方向馅饼中心坐标 */
 ycen = (vp.bottom - vp.top) / 2 + 20; /* 垂直方向馅饼中心坐标 */
 radius = (vp.bottom - vp.top) / 3; /* 半径取屏幕高的三分之一 */
 piesize = (vp.bottom - vp.top) / 4.0; /* 最适当的馅饼高度比 */
 while((AspectRatio * radius) < piesize) ++radius;
 /* 得实际显示用半径不超过 piesize 的半径,但其本身未考虑纵横比 */
 lradius = radius + (radius / 5); /* 用于存放象标记 20% 位置的半径 */
 changetextstyle(TRIPLEX-FONT, HORIZ-DIR, 4);
 /* 三重矢量字体、水平方向、放大四倍 */
 settextjustify(CENTER-TEXT, TOP-TEXT); /* 水平中心对准,垂直以顶对准 */
 outtextxy(MaxX/2, 6, "This is a Pie Chart"); /* 输出串 */
 changetextstyle(TRIPLEX-FONT, HORIZ-DIR, 1); /* 改放大因子 */
 settextjustify(CENTER-TEXT, TOP-TEXT);
 setfillstyle(SOLID-FILL, RED); /* 实心填充,红色 */
 pieslice(xcen+10, ycen-adjasp(10), 0, 90, radius); /* 画 90 度馅饼 */
}

```

```

radians = torad(45); /* 得 45 度角弧度值 */
x = xcenter + (int)(cos(radians) * (double)radius);
y = ycenter - (int)(sin(radians) * (double)radius * AspectRatio);
settextjustify(LEFT—TEXT, BOTTOM—TEXT); /* 水平从左到右,垂直从底往上 */
outtextxy(x, y, "25 %"); /* 标记 25% */
setfillstyle(WIDE—DOT—FILL, GREEN); /* 用稀疏点填充,绿色 */
pieslice(xcenter, ycenter, 90, 135, radius); /* 画 90 度到 135 度馅饼 */
radians = torad(113); /* 得 113 度的弧度值 */
x = xcenter + (int)(cos(radians) * (double)radius);
y = ycenter - (int)(sin(radians) * (double)radius * AspectRatio);
settextjustify(RIGHT—TEXT, BOTTOM—TEXT); /* 水平以右为准,垂直以顶为准 */
outtextxy(x, y, "12.5 %"); /* 标记 12.5% */
setfillstyle(INTERLEAVE—FILL, YELLOW); /* 用间隔线填充,黄色 */
settextjustify(RIGHT—TEXT, CENTER—TEXT); /* 水平以右为准,垂直中心对准 */
pieslice(xcenter—10, ycenter, 135, 225, radius); /* 画 135 度到 225 度馅饼 */
radians = torad(180); /* 180 度弧度值 */
x = xcenter + (int)(cos(radians) * (double)radius);
y = ycenter - (int)(sin(radians) * (double)radius * AspectRatio);
settextjustify(RIGHT—TEXT, CENTER—TEXT);
outtextxy(x, y, "25 %"); /* 标记 25% */
setfillstyle(HATCH—FILL, BLUE); /* 用正方格填充,蓝色 */
pieslice(xcenter, ycenter, 225, 360, radius); /* 画 225 度到 360 度馅饼 */
radians = torad(293); /* 293 度弧度值 */
x = xcenter + (int)(cos(radians) * (double)radius);
y = ycenter - (int)(sin(radians) * (double)radius * AspectRatio);
settextjustify(LEFT—TEXT, TOP—TEXT); /* 水平左对准,垂直以顶对准 */
outtextxy(x, y, "37.5 %"); /* 标记 37.5% */
Pause();
}

void BarDemo(void) /* 画直方图 */
{
 int barheight[] = { 1, 3, 5, 2, 4 };
 int styles[] = { 1, 3, 10, 5, 9, 1 };
 int xstep, ystep;
 int sheight, swidth;
 int i, j, h;
 struct viewporttype vp;
 char buffer[40];
 MainWindow("Bar / Rectangle demonstration");
 h = 3 * textheight("H"); /* 取三倍字符高 */
 getviewsettings(&vp); /* 获当前视口信息 */
 settextjustify(CENTER—TEXT, TOP—TEXT); /* 水平中心对准,垂直以顶为准 */
 changetextstyle(TRI—LEX—FONT, HORIZ—DIR, 4);
 /* 三重矢量字体、水平方向、放大 4 倍 */
 outtextxy(MaxX / 2, 6, "These are 2—D Bars"); /* 显示串 */
 changetextstyle(DEFAULT—FONT, HORIZ—DIR, 1); /* 改用缺省字体,放大为 1 */
}

```



```

setviewport(vp.left+50, vp.top+30, vp.right-50, vp.bottom-10, 1);
/* 重新设置视口 */
getviewsettings(&vp); /* 获得当前视口信息 */
sheight = vp.bottom - vp.top; /* 高 */
swidth = vp.right - vp.left; /* 宽 */
line(h, h, h, sheight-h); /* 画竖线,纵坐标轴 */
line(h, sheight-h, sheight-h, sheight-h); /* 画横线,横坐标轴 */
ystep = (sheight - (2 * h)) / 5; /* 纵向间隔 */
xstep = (swidth - (2 * h)) / 5; /* 横向间隔 */
j = sheight - h;
settextjustify(CENTER-TEXT, CENTER-TEXT); /* 水平与垂直均中心对准 */
for(i=0; i<6; ++i) /* 在纵坐标轴上画6根短横线,并在短线左边印出数字 */
{
 line(h/2, j, h, j); /* 画短横线 */
 itoa(i, buffer, 10); /* 将 i 转为字符,存于 buffer 中 */
 outtextxy(0, j, buffer); /* 在横线左边印出 i 值 */
 j -= ystep; /* 移向下一个横线 */
}
j = h;
settextjustify(CENTER-TEXT, TOP-TEXT); /* 水平中心对准,垂直以顶为准 */
for(i=0; i<6; ++i) /* 画竖线,标横坐标 */
{
 setfillstyle(styles[i], random(MaxColors)); /* 取填充模式,取随机颜色 */
 line(j, sheight - h, j, sheight - 3 - (h/2)); /* 画竖线 */
 itoa(i, buffer, 10);
 outtextxy(j, sheight - (h/2), buffer); /* 在横坐标轴下印出数字 */
 if(i != 5){
 bar(j, (sheight-h)-(barheight[i] * ystep), j+xstep, sheight-h-1);
 /* 画条形图 */
 rectangle(j, (sheight-h)-(barheight[i] * ystep), j+xstep, sheight-h);
 /* 画条形图的边框 */
 }
 j += xstep; /* 转向下一个图形 */
}
Pause();
}

void LineRelDemo(void)
{
 /* 利用 moverel() 和 linerel() 函数画出一个镶嵌物体 */
 struct viewporttype vp;
 int h, w, dx, dy, cx, cy;
 struct PTS outs[7]; /* PTS 是一个表示顶点 (x,y) 的结构 */
 MainWindow("MoveRel / LineRel Demonstration"); /* 显示图幅标题 */
 StatusLine("Press any key to continue, ESC to Abort"); /* 显示状态行 */
 getviewsettings(&vp); /* 获得当前视口信息 */
 cx = (vp.right - vp.left) / 2; /* 屏幕坐标中心,横坐标 */
 cy = (vp.bottom - vp.top) / 2; /* 纵坐标 */
}

```

```

h = (vp.bottom - vp.top) / 8; /* 纵向间隔 */
w = (vp.right - vp.left) / 9; /* 横向间隔 */
dx = 2 * w; /* 横向增量 */
dy = 2 * h; /* 纵向增量 */
setcolor(BLACK); /* 设置当前绘图色为黑色 */
setfillstyle(SOLID_FILL, BLUE); /* 用实心填充,蓝色 */
bar(0, 0, vp.right-vp.left, vp.bottom-vp.top); /* 画底框,以便在上画图 */
outs[0].x = cx - dx; /* 计算各点(0~6)的纵横坐标 */
outs[0].y = cy - dy;
outs[1].x = cx - (dx-w);
outs[1].y = cy - (dy+h);
outs[2].x = cx + dx;
outs[2].y = cy - (dy+h);
outs[3].x = cx + dx;
outs[3].y = cy + dy;
outs[4].x = cx + (dx-w);
outs[4].y = cy + (dy+h);
outs[5].x = cx - dx;
outs[5].y = cy + (dy+h);
outs[6].x = cx - dx;
outs[6].y = cy - dy; /* 6点和0点重合 */
setfillstyle(SOLID_FILL, WHITE); /* 实心,白色 */
fillpoly(7, (int far *)outs); /* 填充六边形 */
outs[0].x = cx - (w/2); /* 计算各点坐标,0点与4点重合 */
outs[0].y = cy + h;
outs[1].x = cx + (w/2);
outs[1].y = cy + h;
outs[2].x = cx + (w/2);
outs[2].y = cy - h;
outs[3].x = cx - (w/2);
outs[3].y = cy - h;
outs[4].x = cx - (w/2);
outs[4].y = cy + h;
setfillstyle(SOLID_FILL, BLUE); /* 实心,蓝色 */
fillpoly(5, (int far *)outs); /* 画四边形,结果屏幕中间有一小块蓝色 */
/* 用moverel()和linerel()画出一个镶嵌物体 */
moveto(cx-dx, cy-dy); /* 将当前光标移到新坐标处 */
linerel(w, -h); /* 从当前光标到与它有相对距离处画一直线,即从
 (cx-dx,cy-dy)到(cx-dx+w,cy-dy-h)画直线 */
linerel(3*w, 0);
linerel(0, 5*h);
linerel(-w, h);
linerel(-3*w, 0);
linerel(0, -5*h);
moverel(w, -h); /* 将当前位置移动一段距离.假定原位置为(x,y),则新位置为(x+w,y-h)
*/

```

```

 linerel(0, 5 * h);
 linerel(w+(w/2), 0);
 linerel(0, -3 * h);
 linerel(w/2, -h);
 linerel(0, 5 * h);
 moverel(0, -5 * h);
 linerel(-(w+(w/2)), 0);
 linerel(0, 3 * h);
 linerel(-w/2, h);
 moverel(w/2, -h);
 linerel(w, 0);
 moverel(0, -2 * h);
 linerel(-w, 0);
 Pause();
}

void PutPixelDemo(void) /* 先写随机象点,而后将它们收起,即在屏幕上看不到它们 */
{
 int seed = 1958;
 int i, x, y, h, w, color;
 struct viewporttype vp;
 MainWindow("PutPixel / GetPixel Demonstration");
 getviewsettings(&vp);
 h = vp.bottom - vp.top;
 w = vp.right - vp.left;
 srand(seed); /* 初始化随机数发生器,起点为 seed */
 for(i=0; i<5000; ++i){ /* 将 5000 个象点放到屏幕上 */
 x = 1 + random(w - 1); /* 产生随机位置 */
 y = 1 + random(h - 1);
 color = random(MaxColors); /* 产生随机颜色 */
 putpixel(x, y, color); /* 在指定坐标处画一象点 */
 }
 srand(seed); /* 产生同一新起点 */
 for(i=0; i<5000; ++i){ /* 使 5000 象点消失 */
 x = 1 + random(w - 1); /* 产生随机位置 */
 y = 1 + random(h - 1);
 color = getpixel(x, y); /* 读指定坐标处象点的颜色 */
 if(color == random(MaxColors)) /* 保持随机同步 */
 putpixel(x, y, 0); /* 写象点为背景色,从而无可见象点显示 */
 }
 Pause();
}

void PutImageDemo(void) /* 读象和写象表演 */
{
 static int r = 20;
 static int StartX = 100;
 static int StartY = 50;

```

```

struct viewporttype vp;
int PauseTime, x, y, ulx, uly, lrx, lry, size, i, width, height, step;
void *Saucer; /* 注意,它定义为 void */
MainWindow("GetImage / PutImage Demonstration");
getviewsettings(&vp);

/* 画茶盘 (Saucer) */
setfillstyle(SOLID—FILL, getmaxcolor()); /* 用实心填充,取最大号颜色 */
fillellipse(StartX, StartY, r, (r/3)+2); /* 画椭圆并填充 */
ellipse(StartX, StartY-4, 190, 357, r, r/3); /* 画椭圆弧 */
line(StartX+7, StartY-6, StartX+10, StartY-12); /* 画线 */
circle(StartX+10, StartY-12, 2); /* 画圆 */
line(StartX-7, StartY-6, StartX-10, StartY-12); /* 画线 */
circle(StartX-10, StartY-12, 2); /* 画圆 */
/* 读茶盘象 */

ulx = StartX-(r+1);
uly = StartY-14;
lrx = StartX+(r+1);
lry = StartY+(r/3)+3;
width = lrx - ulx + 1;
height = lry - uly + 1;
size = imagesize(ulx, uly, lrx, lry); /* 计算保存位图象需要的字节数 */
Saucer = malloc(size); /* 分配动态内存 */
getimage(ulx, uly, lrx, lry, Saucer); /* 将指定区域的位图象存到 Saucer 中 */
putimage(ulx, uly, Saucer, XOR—PUT); /* 将屏幕上已有象点与早先存在 Saucer 中的象点进行
 "异或"运算,图象左上角为 (ulx,uly) */
/* "异或"运算,0+0=0, 0+1=1, 1+0=1, 1+1=0, 1 表示见象点,0 表示不见象点 */
for (i=0; i<1000; ++i) /* 绘 1000 个星点 (stars) */
 putpixel(random(MaxX), random(MaxY), random(MaxColors-1)+1);
/* 在指定 (随机) 位置用指定颜色画一象点 */

x = MaxX / 2;
y = MaxY / 2;
PauseTime = 70; /* 茶盘不动时间 */
while (!kbhit()) { /* 茶盘将来回移动,按一键后终止循环 */
 putimage(x, y, Saucer, XOR—PUT); /* 画茶盘 */
 delay(PauseTime); /* 延迟一段时间 */
 putimage(x, y, Saucer, XOR—PUT); /* 擦去茶盘 */
 /* 从中可见"异或"运算的具体用处 */
 step = random(2 * r); /* 移动茶盘量 */
 if ((step/2) % 2 != 0)
 step = -1 * step;
 x = x + step;
 step = random(r);
 if ((step/2) % 2 != 0)
 step = -1 * step;
 y = y + step;
 if (vp.left + x + width - 1 > vp.right) /* 防止 x 越界 */

```

```

 x = vp.right - vp.left - width + 1;
else
 if (x < 0) x = 0;
if (vp.top + y + height - 1 > vp.bottom) /* 防止 y 越界 */
 y = vp.bottom - vp.top - height + 1;
else
 if (y < 0) y = 0;
}
free(Saucer); /* 释放动态内存 */
Pause();
}

#define MAXPTS 15 /* 定界圆上最多顶点数 */
void LineToDemo(void) /* 在一个定界圆上画多根弦 */
{
 struct viewporttype vp;
 struct PTS points[MAXPTS]; /* PTS 是结构, 有两个成员: (x,y), 即顶点坐标 */
 int i, j, h, w, xcenter, ycenter;
 int radius, angle, step;
 double rads;
 MainWindow("MoveTo / LineTo Demonstration");
 getviewsettings(&vp);
 h = vp.bottom - vp.top;
 w = vp.right - vp.left;
 xcenter = w / 2; /* 确定圆心坐标 */
 ycenter = h / 2;
 radius = (h - 30) / (AspectRatio * 2);
 step = 360 / MAXPTS; /* 确定增量 */
 angle = 0; /* 开始角度为 0 */
 for(i=0; i<MAXPTS; ++i) /* 确定圆上 MAXPTS 个相交点坐标 */
 {
 rads = (double)angle * PI / 180.0; /* 将角度转为弧度 */
 /* 确定交点坐标 */
 points[i].x = xcenter + (int)(cos(rads) * radius);
 points[i].y = ycenter - (int)(sin(rads) * radius * AspectRatio);
 angle += step; /* 角度增值, 指向下一点 */
 }
 circle(xcenter, ycenter, radius); /* 画一定界圆 */
 for(i=0; i<MAXPTS; ++i) /* 对圆画 MAXPTS 根弦 (cord) */
 {
 for(j=i; j<MAXPTS; ++j) /* 对每一个保留的交点 */
 {
 moveto(points[i].x, points[i].y); /* 移向弦起点 */
 lineto(points[j].x, points[j].y); /* 画一根弦 */
 }
 }
 Pause();
}

```

```

}

void LineStyleDemo(void)/* 显示所有可用的标准线型 */
{
 int style, step;
 int x, y, w;
 struct viewporttype vp;
 char buffer[40];
 MainWindow("Pre-defined line styles");
 getviewsettings(&vp);
 w = vp.right - vp.left;
 x = 35;
 y = 10;
 step = w / 11;
 setttextjustify(LEFT-TEXT, TOP-TEXT); /* 水平左对准,垂直顶对准 */
 outtextxy(x, y, "Normal Width"); /* 输出串"标准宽度" */
 setttextjustify(CENTER-TEXT, TOP-TEXT); /* 改水平为中心对准 */
 for(style=0 ; style<4 ; ++style) /* 四种字体 */
 {
 setlinestyle(style, 0, NORM-WIDTH);
 line(x, y+20, x, vp.bottom-40); /* 画线 */
 itoa(style, buffer, 10); /* 将线型变为字符 */
 outtextxy(x, vp.bottom-30, buffer); /* 输出线型 */
 x += step; /* 横向移动一个位置 */
 }
 x += 2 * step;
 setttextjustify(LEFT-TEXT, TOP-TEXT); /* 水平左对准,垂直顶对准 */
 outtextxy(x, y, "Thick Width"); /* 输出串"粗宽度" */
 setttextjustify(CENTER-TEXT, TOP-TEXT); /* 水平以中心对准,垂直以顶为准 */
 for(style=0 ; style<4 ; ++style) /* 四种宽线 */
 {
 setlinestyle(style, 0, THICK-WIDTH);
 line(x, y+20, x, vp.bottom-40);
 itoa(style, buffer, 10);
 outtextxy(x, vp.bottom-30, buffer);
 x += step;
 }
 setttextjustify(LEFT-TEXT, TOP-TEXT); /* 水平左对准,垂直以顶对准 */
 Pause();
}

void CRTModeDemo(void) /* 在当前屏幕上演示改变显示模式后的结果 */
{
 struct viewporttype vp;
 int mode;
 MainWindow("SetGraphMode / RestoreCRTMode demo");
 getviewsettings(&vp);
 mode = getgraphmode(); /* 返回当前的图形模式 */
}

```

```

settextjustify(CENTER—TEXT, CENTER—TEXT); /* 水平和垂直方向均中心对准 */
outtextxy((vp.right—vp.left)/2, (vp.bottom—vp.top)/2,
"Now you are in graphics mode..."); /* 印出“现在是图形模式” */
StatusLine("Press any key for text mode...");
getch(); /* 印出“压任一键进入文本模式” */
restorecrtmode(); /* 恢复 initgraph() 前的屏幕模式 */
printf("Now you are in text mode.\n\n"); /* 打印“现在是文本模式” */
printf("Press any key to go back to graphics...");
getch(); /* 印出“按任一键返回图形模式” */
setgraphmode(mode); /* 选择图形模式,清屏,恢复缺省值 */
MainWindow("SetGraphMode / RestoreCRTMode demo"); /* 置图幅标题 */
settextjustify(CENTER—TEXT, CENTER—TEXT);
outtextxy((vp.right—vp.left)/2, (vp.bottom—vp.top)/2,
"Back in Graphics Mode..."); /* 印出“在图形模式” */
Pause();
}

void UserLineStyleDemo(void) /* 显示用户自定义的线型 */
{
 int x, y, i, h, flag;
 unsigned int style;
 struct viewporttype vp;
 MainWindow("User defined line styles");
 getviewsettings(&vp);
 h = vp.bottom — vp.top;
 x = 4;
 y = 10;
 style = 0;
 i = 0;
 settextjustify(CENTER—TEXT, TOP—TEXT); /* 水平中心对准,垂直以顶为准 */
 flag = TRUE; /* 当 flag 为这种值时设置位 */
 while(x < vp.right—2) /* 画横穿屏幕的线 */
 {
 if(flag) /* 如果 flag 为真,设置位 */
 style = style | (1 << i); /* 设置 style 的第 i 位为 1 */
 else /* 如果 flag 为假,清位值 */
 style = style & 1(0x8000 >> i); /* 将 style 第 i 位清 0 */
 setlinestyle(USERBIT—LINE, style, NORM—WIDTH); /* 用户定义线型 */
 line(x, y, x, h—y); /* 画新的线 */
 x += 5; /* 移动线X位置 */
 i = ++i % 16; /* 进到下一个位图式样,i 超过 15 则循环 */
 if(style == 0xffff) /* 所有的位都要设置? */
 flag = FALSE; /* 设清除这些位标记 */
 i = 0; /* 重新开始,注意 x 值已变,故屏幕上从左到右出现重复图形 */
 }
 else{ /* 位还没有设置 */
 if(style == 0) /* 所有的位都被清了? */

```

```

 flag = TRUE; /* 设置位要重新设置的标记 */
 }
}
settextjustify(LEFT—TEXT, TOP—TEXT); /* 水平左对准,垂直以顶对准 */
Pause();
}

void FillStyleDemo(void) /* 显示所有标准填充模式 */
{
 int h, w, style;
 int i, j, x, y;
 struct viewporttype vp;
 char buffer[40];
 MainWindow("Pre—defined Fill Styles");
 getviewsettings(&vp);
 w = 2 * ((vp.right + 1) / 13);
 h = 2 * ((vp.bottom - 10) / 10);
 x = w / 2;
 y = h / 2; /* 预留 1/2 空白页边 */
 style = 0; /* 初始填充模式 */
 for(j=0; j<3; ++j) /* 显示矩形分成三行 */
 {
 for(i=0; i<4; ++i) /* 每行有四个矩形 */
 {
 setfillstyle(style, MaxColors-1); /* 设置所有填充模式时用白色 */
 bar(x, y, x+w, y+h); /* 画实际的矩形 */
 rectangle(x, y, x+w, y+h); /* 画矩形轮廓线 */
 itoa(style, buffer, 10); /* 将 style 转为字符 */
 outtextxy(x+(w / 2), y+h+4, buffer); /* 显示 style */
 ++style; /* 转向下一个填充模式 */
 x += (w / 2) * 3; /* 转向本行上下一个矩形 */
 }
 x = w / 2; /* 重新设置一行的第一个矩形的横向位置 */
 y += (h / 2) * 3; /* 进到下一行位置 */
 }
 settextjustify(LEFT—TEXT, TOP—TEXT); /* 水平以左为准,垂直以顶为准 */
 Pause();
}

void FillPatternDemo(void) /* 使用用户定义填充模式 */
{
 /* 未注释处参见 void FillStyleDemo(void) 中说明 */
 int style;
 int h, w;
 int x, y, i, j;
 char buffer[40];
 struct viewporttype vp;
 static char patterns[][8] = { /* 用户定义的填充模式 */
 { 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55, 0xAA, 0x55 },

```



```

 { 0x33, 0x33, 0xCC, 0xCC, 0x33, 0x33, 0xCC, 0xCC },
 { 0xF0, 0xF0, 0xF0, 0xF0, 0x0F, 0x0F, 0x0F, 0x0F },
 { 0x00, 0x10, 0x28, 0x44, 0x28, 0x10, 0x00, 0x00 },
 { 0x00, 0x70, 0x20, 0x27, 0x24, 0x24, 0x07, 0x00 },
 { 0x00, 0x00, 0x00, 0x18, 0x18, 0x00, 0x00, 0x00 },
 { 0x00, 0x00, 0x3C, 0x3C, 0x3C, 0x3C, 0x00, 0x00 },
 { 0x00, 0x7E, 0x7E, 0x7E, 0x7E, 0x7E, 0x7E, 0x00 },
 { 0x00, 0x00, 0x22, 0x08, 0x00, 0x22, 0x1C, 0x00 },
 { 0xFF, 0x7E, 0x3C, 0x18, 0x18, 0x3C, 0x7E, 0xFF },
 { 0x00, 0x10, 0x10, 0x7C, 0x10, 0x10, 0x00, 0x00 },
 { 0x00, 0x42, 0x24, 0x18, 0x18, 0x24, 0x42, 0x00 }
},
MainWindow("User Defined Fill Styles");
getviewsettings(&vp);
w = 2 * ((vp.right + 1) / 13);
h = 2 * ((vp.bottom - 10) / 10);
x = w / 2;
y = h / 2;
style = 0;
for(j=0; j<3; ++j){
 for(i=0; i<4; ++i){
 setfillpattern(&patterns[style][0], MaxColors-1); /* 注意此句 */
 bar(x, y, x+w, y+h);
 rectangle(x, y, x+w, y+h);
 itoa(style, buffer, 10);
 outtextxy(x+(w/2), y+h+4, buffer);
 ++style;
 x += (w/2) * 3;
 }
 x = w / 2;
 y += (h/2) * 3;
}
settextjustify(LEFT-TEXT, TOP-TEXT);
Pause();
}
void PaletteDemo(void) /* 显示调色板随机变化时的图形 */
{
 int i, j, x, y, color;
 struct viewporttype vp;
 int height, width;
 MainWindow("Palette Demonstration");
 StatusLine("Press any key to continue, ESC to Abort");
 getviewsettings(&vp);
 width = (vp.right - vp.left) / 15; /* 矩形宽 */
 height = (vp.bottom - vp.top) / 10; /* 矩形高 */
 x = y = 0; /* 开始在左上角 */

```

```

color = 1; /* 第一个多边形颜色 */
for(j=0 ; j<10 ; ++j){ /* 10 行矩形图 */
 for(i=0 ; i<15 ; ++i){ /* 每行 15 个矩形 */
 setfillstyle(SOLID—FILL, color++); /* 实心填充 */
 bar(x, y, x+width, y+height); /* 画条形 */
 x += width + 1; /* 置同行上下一个的x */
 color = 1 + (color % (MaxColors - 2)); /* 设新颜色 */
 }
 x = 0; /* 转向第一个 */
 y += height + 1; /* 转向下一行 */
}
while(!kbhit()){ /* 直到按键后结束循环 */
 setpalette(1+random(MaxColors - 2), random(65)); /* 改变调色板颜色 */
} /* 调色板一变,所有的矩形颜色也改变 */
setallpalette(&palette); /* 改变所有调色板 */
Pause();
}

#define MaxPts 6 /* 多边形最多顶点数 */
void PolyDemo(void) /* 随机画和填充多边形 */
{
 struct PTS poly[MaxPts]; /* 处理数据点的空间 */
 int color; /* 当前绘图颜色 */
 int i;
 MainWindow("DrawPoly / FillPoly Demonstration");
 StatusLine("ESC Aborts — Press a Key to stop");
 while(!kbhit()) /* 未按键前一直循环 */
 {
 color = 1 + random(MaxColors-1); /* 得所有随机颜色 */
 setfillstyle(random(10), color); /* 得随机填充模式 */
 setcolor(color); /* 设置当前绘图色 */
 for(i=0 ; i<(MaxPts-1) ; i++) /* 确定随机多边形顶点坐标 */
 {
 poly[i].x = random(MaxX); /* 点 x 坐标 */
 poly[i].y = random(MaxY); /* 点 y 坐标 */
 }
 poly[i].x = poly[0].x; /* 最后一点等于第一点 */
 poly[i].y = poly[1].y;
 fillpoly(MaxPts, (int far *)poly); /* 画实际多边形并填充 */
 }
 Pause();
}

void SayGoodbye(void) /* 在结束表演时的显示 */
{
 struct viewporttype viewinfo;
 int h, w;
 MainWindow("== Finale ==");

```

```

getviewsettings(&viewinfo);
changetextstyle(TRIPLEX—FONT, HORIZ—DIR, 4); /* 三重矢量字体,放大 4 倍 */
settextjustify(CENTER—TEXT, CENTER—TEXT);
h = viewinfo.bottom - viewinfo.top;
w = viewinfo.right - viewinfo.left;
outtextxy(w/2, h/2, "That's all, folks!"); /* 印出“这就是所有的演示” */
StatusLine("Press any key to EXIT"); /* 按键终止程序执行 */
getch(); /* 等待按键 */
cleardevice(); /* 清图形屏幕 */
}

/* 暂停,直到用户按了一个键,如按 ESC 键,程序中中止运行,否则清屏后简单返回 */
void Pause(void)
{
 static char msg[] = "Esc aborts or press a key...";
 int c; /* 按 ESC 键终止显示,按其它键显示继续 */
 StatusLine(msg); /* 将字符串 msg 放到屏幕最底下一行显示 */
 c = getch(); /* 从键盘读一字符 */
 if(ESC == c){ /* 如果你键 ESC 键,表示想离开图形系统 */
 closegraph(); /* 关闭图形库,返回最初的文本方式 */
 exit(1); /* 返回操作系统 */
 }
 if(0 == c){ /* 如果你刚刚按的是非 ASCII 键 (如控制键) */
 c = getch(); /* 则允许你再从键盘键入一个字符 */
 }
 cleardevice(); /* 清屏幕 */
}

/* 建立主显示窗 */
void MainWindow(char * header) /* header 指向图幅标题 */
{
 int height;
 cleardevice(); /* 清屏 */
 setcolor(MaxColors - 1); /* 设置当前显示颜色为白色 */
 setviewport(0, 0, MaxX, MaxY, 1); /* 将整个屏幕作视口 */
 /* 第 5 个参数 1 表示视口外图形将被剪切了 */
 height = textheight("H"); /* 以象素形式返回一个字符的高 */
 changetextstyle(DEFAULT—FONT, HORIZ—DIR, 1); /* 缺省字体、方向和比例因子 */
 settextjustify(CENTER—TEXT, TOP—TEXT);
 /* 水平以当前光标中心对准,垂直以顶端为准 */
 outtextxy(MaxX/2, 2, header); /* 屏幕上第2行以横向中心为准打印出图幅标题 */
 setviewport(0, height+4, MaxX, MaxY-(height+4), 1); /* 改变视口大小 */
 DrawBorder(); /* 用实线画当前视口的边框 */
 setviewport(1, height+5, MaxX-1, MaxY-(height+5), 1); /* 再改变视口 */
}

/* 在屏幕底行上输出状态行字符串 */
void StatusLine(char * msg)
{

```

```

int height;
setviewport(0, 0, MaxX, MaxY, 1); /* 打开整个屏幕 */
setcolor(MaxColors - 1); /* 设置当前颜色为白色 */
changetextstyle(DEFAULT-FONT, HORIZ-DIR, 1); /* 缺省字体, 字体从左到右 */
settextjustify(CENTER-TEXT, TOP-TEXT); /* 中心对准, 从上往下 */
setlinestyle(SOLID-LINE, 0, NORM-WIDTH); /* 实线, 正常宽度 */
setfillstyle(EMPTY-FILL, 0); /* 背景色填充 */
height = textheight("H"); /* 决定当前字符高 */
bar(0, MaxY-(height+4), MaxX, MaxY); /* 画条形图 */
rectangle(0, MaxY-(height+4), MaxX, MaxY); /* 画矩形 */
outtextxy(MaxX/2, MaxY-(height+2), msg); /* 输出字符串 msg */
setviewport(1, height+5, MaxX-1, MaxY-(height+5), 1); /* 设置当前视区 */
}
/* 用实线画当前视口的边框 */
void DrawBorder(void)
{
 struct viewporttype vp;
 setcolor(MaxColors - 1); /* 设置当前颜色为白色 */
 setlinestyle(SOLID-LINE, 0, NORM-WIDTH); /* 设实线, 正常宽度 */
 /* 第2个参数只有当第1个参数为USERBIT-LINE即用户自定义时才起作用 */
 getviewsettings(&vp); /* 取当前视口信息 */
 rectangle(0, 0, vp.right-vp.left, vp.bottom-vp.top); /* 画矩形边框 */
}
/* 本函数仅与 Turbo C 库函数多了一个检测错误过程。当装入一个字体文件
 (GOTH.CHR、LITT.CHR、SANS.CHR 和 TRIP.CHR) 时可能出现 */
void changetextstyle(int font, int direction, int charsize)
{
 int ErrorCode;
 graphresult(); /* 调用后最后一次不成功错误码复位为 0, 或者说被清除 */
 settextstyle(font, direction, charsize); /* 设置文本字体、方向和放大因子 */
 ErrorCode = graphresult(); /* 检查设置文本时有否产生错误 */
 if(ErrorCode != grOk) /* 如果有错误发生 */
 {
 closegraph(); /* 先关闭图形系统 */
 printf("Graphics System Error: %s\n", grapherrormsg(ErrorCode));
 exit(1); /* 当没有错误发生时此信息自然不会打印出来 */
 }
}
/* 自定义象 printf() 那样的屏幕打印函数, 将输出按图形方式送屏幕指定坐标处, 然后使行指针指向下一行 */
int gprintf(int *xloc, int *yloc, char *fmt, ...)
{
 va-list argptr; /* 参数列表指针 */
 char str[140]; /* 缓冲区 */
 int cnt; /* 记录 gprintf() 输出字节数 */
 va-start(argptr, format); /* format 也可以用其它标识符代替 */

```

```

cnt = vsprintf(str, fmt, argptr); /* 将格式串送入缓冲区 str 中,并返回送入字节数 */
outtextxy(*xloc, *yloc, str); /* 按图形方式将串送屏幕 */
* yloc += textheight("H") + 2; /* 将表示行的指针当前所指变量值增值,结果指针下一次指向
 下一行,2 是行间隔,这是利用指针传值 */
va—end(argptr); /* 表示 va— 功能结束 */
return(cnt); /* 返回转换输出字节数 */
}

```

## 35.10 在西文操作系统下直接显示汉字

表 35—18 键入的区位码、显示的汉字和磁盘上存的码对照表

| 键入的区位码      | 显示的汉字 | 磁盘文件上存的码(占两个字节)  | 区、位内码 |   |
|-------------|-------|------------------|-------|---|
| 1601(16区1位) | 啊     | B0 A1(低位在前,高位在后) | 15    | 0 |
| 1602(16区2位) | 阿     | B0 A2(低位在前,高位在后) | 15    | 1 |

从磁盘上先读入“啊”字时实际读入两字节,先读入低字节  $ch[ci]=B0$ ,后读入高字节  $ch[ci+1]=A1$ 。它们分别减去  $0xa1$  后再略去最高位(位 7)便是区内码和位内码。最后按公式  
 汉字库中该字对应的记录号 =  $\{ch[ci]-0xa1\} * 94 + ch[ci+1]-0xa1$

在  $16 * 16$  点阵字库(例如 UC DOS 2.0 的  $16$  点阵字库 `cclib.dat`、2.13 系统的 `hbk16`)中寻找该汉字。注意,由于不同的  $16$  点阵字库可能有些不同,因此公式对有的  $16$  点阵汉字库将要修正。用试探法这是不难做到的。例如,对 `cclibj.dot` 是

汉字库中该字对应的记录号 =  $\{ch[ci]-0xa1-6\} * 94 + ch[ci+1]-0xa1$

事实上,这些公式对  $24 * 24$  点阵也可用,因为区位码并不和点阵数相关。只是  $16$  点阵在字库中一个汉字占  $32$  个字节,而  $24$  点阵则占  $72$  字节。并且这些字节描述汉字的方法有些不同而已。

```

C>TYPE HZ1.C
#include "stdio.h"
#include "ctype.h"
#include "graphics.h"
main() /* 在西文下直接显示汉字的方法 */
{ /* 根据傅叔平的程序改写并注释 */
void SC();
int driver,mode;
unsigned char c[]="阿伟伪大"; /* 待显示字符串 */
driver=DETECT; /* 用最前面的汉字容易帮你找正汉字在字库中位置 */
initgraph(&driver,&mode,""); /* 对不同显示器应有相应的*.BGI文件 */
cleardevice(); /* 清图形屏幕 */
SC(50,50,c,12,YELLOW); /* 显示字符串 */
getch(); /* 等待击任一键 */
restorecrtmode(); /* 将屏幕恢复为 initgraph 设置前情形 */
}
void SC(int x,int y,unsigned char *ch,int s,int color)

```

```

{ /* x,y 是汉字串显示的起始位置, ch 指向要显示的汉字串, 其中可以有 ASCII 码字符, s 是汉字
 之间间隔, 以像素为单位, color 是颜色. */
void Re();
unsigned char buf[32];
unsigned int up;
int i, ci;
char ascstr[2];
setcolor(color);
ci=0;
while(ci<strlen(ch))
{
 if(ischn(ch[ci])) /* 判断是否是汉字. 是, 转去处理汉字 */
 {
 /* ch[ci] */
 Re(ch[ci], ch[ci+1], buf); /* 读汉字点阵入缓冲区 */
 for(i=0; i<32; i+=2)
 {
 up=(buf[i]<<8)|buf[i+1]; /* 确定线模式 up */
 /* 构成一个汉字的 32 个字节分布情况是:
 [
 buf[0] buf[1] y i=0
 buf[2] buf[3] y+1 i=2

 buf[30] buf[31] y+15 i=30
]
 */
 setlinestyle(USERBIT+LINE, up, NORM+WIDTH); /* 正常宽为一个点 */
 line(x+15, y+i/2, x, y+i/2); /* 划线方法从上往下进行, 每次从左往右画 2 个字节 */
 }
 x=x+16+s; /* 移动横向(列方向)位置 */
 ci+=2;
 continue;
 }
 if(isprint(ch[ci])) /* 处理非汉字, 但字高同汉字 */
 {
 ascstr[0]=ch[ci], ascstr[1]='\0';
 settextstyle(SMALL+FONT, HORIZ+DIR, 7); /* 必须有 LITT.CHR 文件 */
 /* 采用 SMALL+FONT 小号矢量字体, 字符尺寸为 7, 字符大小与汉字相当 */
 outtextxy(x, y-2, ascstr); /* 在视区中显示一字符串 */
 /* textwidth("H") 返回一个西文字符 (H) 在横向所占的像素个数 */
 x=x+textwidth("H")+s; /* 移动横向(列方向)位置 */
 ci+=strlen(ascstr);
 }
}
}

/* 从汉字字库中读取所需汉字的点阵数据 */

```

图 35-29

汉字是 16×16 点阵, 即在图形方式下, 一个汉字占 16 行, 而在一行上有 16 个像素位

```

void Re(unsigned char low,unsigned char high,unsigned char * buf)
{
 /* high 是高位, low 是低位, buf 是缓冲区 */
 FILE * fp, * fopen();
 long p; /* 打开指定目录下的汉字库文件 cclib, 可用其字汉字库, 如 CCDOS 4.0
 的 CCLIB, 王码系统的 CCLIBJ, DOT 等, 但应注意库结构。 */
 if((fp=fopen("c:\\ccdosm2\\cclib","rb"))==NULL)
 {
 /* 如字库文件不存在, 则退出 */
 restorecrtmode();
 printf("no open\n");
 exit(1);
 }
 p=low-0xa1;
 if(p>=15)p-=6; /* 有些字库可能不要此项比较, 如 UCdos 2.0 的 cclib.dat 汉字
 库 (16x16) */
 p=p*94+high-0xa1-188; /* 188 是根据 cc dos 2.1m 的 cclib 修正值, 对其它字库可能不
 要此数值 */
 fseek(fp,(long)p*32,SEEK-SET); /* 总是从文件头部开始查找 */
 fread(buf,sizeof(unsigned char),32,fp); /* 将 32 字节读入缓冲区 */
 fclose(fp);
}

int ischn(int c) /* 判断是否为汉字字符 */
{
 return (c>=0xa1 && c<=0xfe);
}

```

以下为对 24 \* 24 点阵字库的处理。一个汉字的 24 \* 24 点阵和一个字节排列方式为

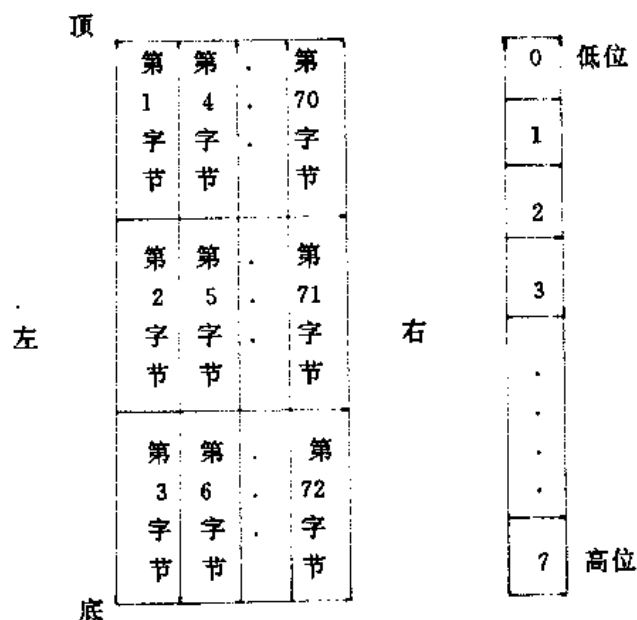


图 35-30

即横向 24 个字节, 纵向为 3 个字节。

```

C>TYPE HZ2.C
#include "graphics.h"

```

```

#include "stdio.h"
#include "fcntl.h"
#include "ctype.h"
#define GETBIT by[i1*3+i2]>>(7-i3)&0x1 /* 取一位,看是否为0 */
#define DISTANCE 8
int handle;
main() /* 在西文下显示“欢OK迎” */
{
int gdriver=VGA,gmode=VGAHI;
initgraph(&gdriver,&gmode,""); /* 当前目录 */
setbkcolor(BLUE); /* 设置背景色为蓝色 */
cleardevice(); /* 清图形屏幕 */
handle=open("c:\\clibs.dot",O_RDONLY|O_BINARY); /* 打开 24x24 字库文件 */
if(handle==-1){printf("失败");exit(1);}
puthz24(200,200,DISTANCE,LIGHTRED,"欢OK迎"); /* 调用显示汉字函数 */
getch();
closegraph();
}
int puthz24(int x,int y,int z,int color,char *string)
{
unsigned int i,c1,c2,f=0;
int i1,i2,i3,rec;
long len;
char by[72];
FILE *fp;
char t[2];
fp=fopen("a","w+b"); /* 打开文件,文件用于存储字库中某些字的点阵 */
while((i=*string++)!=0)
{
if(i>0xa1) /* 如果是汉字,有 i>0xa1,这里 0xa1=161 */
{
if(f==0)
{
c1=(i-0xa1)&0x7f; /* 得到汉字区内码 */
f=1;
}
else
{
c2=(i-0xa1)&0x7f; /* 得到汉字位内码 */
f=0;
rec=c1*94+c2; /* 得到汉字在字库中的记录号 */
len=rec*72; /* 每次取 72 字节 */
lseek(handle,len,SEEK-SET); /* 定位到字库头部 */
read(handle,by,72); /* 读出 72 个字节 */
fwrite(by,1,72,fp); /* 将读出字节记录在文件中 */
for(i1=0;i1<24;i1++) /* 一个汉字在水平方向的 24 点 */

```



```

 for(i2=0;i2<=2;i2++) /* 一个汉字在垂直方向三个字节 */
 for(i3=0;i3<8;i3++) /* 每个字节的 8 位 */
 if(GETBIT) /* 判断某位是否为 1 */
 putpixel(x+i1,y+i2*8+i3,color);
 x=x+24+z; /* 确定下一个汉字显示的位置 */
 }
}
else { /* 显示西文字符 */
 if(isprint(i)) /* 如果将西文用汉字表示,则可不要这一步 */
 {
 t[0]=i;
 t[1]='\0';
 settextstyle(SMALL-FONT,HORIZ-DIR,7); /* 选用适当的西文字体 */
 outtextxy(x,200,t); /* 以便使显示的西文和汉字匹配 */
 x +=textwidth("H")+8;
 }
}
}
return x;
}

```

明白了其显示原理后就不难将汉字进行放大、缩小、旋转、倾斜、空心或隐形处理。在处理中可以使用象 `bar()`、`ellipse()`、`fillellipse()`、`setfillstyle()` 等函数。

对一个只用少数汉字显示的程序,也可以先将字库中相关汉字的记录取出,然后将它装入程序或专门文件中再处理,这样就可摆脱字库。

当然,必要时你也可直接自构汉字点阵而不用字库。

## 第三十六章 发声

### 36.1 计算机发声原理和相关库函数

计算机内部装有一个扬声器（喇叭）。发声主要由可编程外围接口芯片 8255 端口 B 位 1 的信号和定时器芯片 8253(AT 机为 8254) 的通道 2 输出信号决定（参见《中断和中断函数》一章）。

利用 8255 可关闭扬声器,对 Turbo C 可直接调用函数 nosound() 实现。

—1 void —Cdecl nosound(void)

```
{
 int old,new;
 old=inp(0x61); /* 读取端口 B 中的数据 */
 new=old & 0xfc; /* 把位 1 和位 0 置成 0 */
 outp(0x61,new); /* 把新值送回端口 B,关扬声器 */
}
```

在音响电路中,8253 的通道 2(端口地址是 42H)的输出端与扬声器相连,可用程序设定通道 2 输出波形的频率和延续时间,以控制扬声器的音调和发声的长短。

利用 8253 和 8255 使扬声器发声,对 Turbo C 可直接调用库函数 sound() 实现。sound 以指定频率 frequency(单位:赫兹)打开扬声器。它只适用于 IBM PC 及兼容机。

—2 void —Cdecl sound(unsigned frequency)

```
{
 unsigned long clk=1193180; /* 1193180 是系统时钟的速率 */
 int tc,old,new;
 if(frequency<1)frequency=65535;
 tc=clk/frequency; /* 根据已知声音的频率,求送8253 通道 2 的时间常数 */
 old=inp(0x61); /* 对 8255 芯片操作,开扬声器 */
 new=old | 0x03;
 outp(0x61,new);
 outp(0x43,0xb6); /* 往 8253 方式控制寄存器(端口 43H)送 B6H */
 outp(0x42,tc & 0x00ff); /* 把时间常数送入通道 2 */
 outp(0x42,(tc>>8) & 0x00ff);
}
```

### 36.2 乐曲构成原理

#### 1. 音乐中使用的音的频率

频率大致在 27Hz ~ 4100Hz( 赫兹 ) 范围内。在实际使用时,只用其中不到 100 个 频率的音。目前,大钢琴也只有 88 个高低不同的【乐音】( 每个乐音对应钢琴上的一个 键 )。

乐音分成若干组,称为【音阶】。每个音阶中包括 7 个【基本音】和 5 个【半音】( 即 12 个音 )。所以,一般只有  $(7+5) \times 7 = 84$  个有效音符( 后 7 指只有 7 个音阶 )。

简谱中的基本音是 1、2、3、4、5、6 和 7, 可用【音符】C、D、E、F、G、A 和 B 表示。音符是记录音高低的记号。每个音符对应一个频率。从中音 C 起的八音度的频率( 音阶为 4, 内中未列出半音的值,读者可自行计算出来 ):

表36-1

| 音 阶   | C     | D     | E     | F     | G     | A     | B     |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 3(低音) | 262   | 294   | 330   | 349   | 392   | 440   | 494   |
| 4(中音) | 523.3 | 587.3 | 659.3 | 698.3 | 784.0 | 880.0 | 987.7 |
| 5(高音) | 1047  | 1175  | 1319  | 1397  | 1568  | 1760  | 1976  |

高一个八度( 高一音阶 ) 的频率大约是这些值的一倍,高两个八度的频率再增大一倍;相应地,低一个八度的频率约为这些值的一半,如此等等。

每个八度( 音程 ) 都是从 C 调到 B 调。音阶可分成 0 ~ 6。音阶 3 是以中音 C 调开始的。一些 BASIC 语言以音阶 4 作为缺省的八度音。

在乐曲中用一个或两个圆点加在基本音的上面( 或下面 ) 表示升高( 或降低 ) 一个或两个音阶。例如,

.
   
2
   
及
   
2
   
.

在 BASIC 中是用 > 或 < 表示升高或降低一个音阶的,也可以用字母 O 后面跟数字 n ( n = 0~6 ) 表示演奏的音符所属的音阶。

将基本音升高半音后叫【升音】。升音可用音符修饰符 # ( 或用 + ) 加在音符前面。例如, #C 表示将 C 升高半音。有 5 个升音: #C、#D、#F、#G 和 #A; 将基本音降低半音叫【降音】。降音可用音符修饰符 b ( 或用 - ) 加在音符前面。例如, bD 表示将 D 降半音。共有 5 个降音: bD、bE、bG、bA 和 bB。

从半音角度看,实际只有 5 个半音,因为显然有 bD = #C、bE = #D、bG = #F、bA = #G 和 bB = #A。

基本音对应钢琴上的白键,半音对应钢琴上的黑键。

## 2. 音符演奏时间

每首乐曲都规定了演唱速度,用每分钟的节拍数来表示( BASIC 中用 T 后跟数字 n 表示, n 为每分钟节拍数 )。例如,速度为每分钟 60 拍时,则 1 拍所占用的时间是 1 秒。一般乐曲中,最常用的是每个四分音占 1 拍,即演奏 1 秒时间。因而八分音演奏 0.5 秒( 半拍 ), 依此类推其它音的演奏时间。

在 BASIC 中,用 T 后面加一个数字 n ( n = 32 ~ 255, 缺省值为 120 ) 表示每分钟内 演奏

的四分音符的个数。 $n$  值越大,演奏速度越快。

如在基本音,例如在四分音符后面每加一短横线,则延长一个四分音符的时间;加两个横线延长两个四分音符的演奏时间。例如,3 — 及 3 — — 等。

在基本音右下角加一小圆点叫【符点】,表示延长前面音符时间的一半,如 1.6。如加两个圆点表示以前面已延长后的时间为基准(为原时间的 1.5 倍),再延长它的一半时间;如在基本音下面每加一根短划线表示缩短基本音时间的一半,像 1 等。

在 BASIC 中用 L1、L2、L4、L8、...、L64 表示演奏全音、半音、四分音、八分音、...、六十四分音的演奏持续时间分别为 1、1/2、1/4、...、1/64。L 后跟数字  $n$  称为【音符长度】,而  $1/n$  为演奏时间。 $n$  越大,演奏时间越短。

P 后面跟数字  $n$  表示休止符,即停  $n$  拍。P 后面也可以跟符点,其含义同上。

### 36.3 演奏音乐例程

C>TYPE MUSIC.C

```
#ifndef —PLAY— /* 如将以下内容放在某一头文件中,则可根据是否 */
#define —PLAY— /* 定义了 —PLAY— 来决定是否使用本演奏音乐的函数 */

#include "dos.h"
#include "math.h"
#include "ctype.h"
#include "string.h"
#include "stdio.h"
#include "stdlib.h"

#define PLAY—MAX 255 /* 演奏速度:最大分音符个数 */
#define PLAY—MIN 1 /* 最小分音符个数 */
#define PLAY—O—MAX 6 /* 乐谱最大音阶 */
#define PLAY—O—MIN 0 /* 乐谱最小音阶 */
#define NUM—MAX 4 /* T,L,P,O 后跟最大数字位数 */
#define YF—1 4.0 /* 四分音符 */

char *num="00000000";
char play—yf[]="XX"; /* 记录演奏的音符 */
float yf—low=32.7; /* 音符最低频率 */
float yf—freq; /* 频率 */
float yf—x=1.05946; /* 相邻两个(半)音符的频率值之比,约等于 2 开 12 次方。
例如: #C÷C≈1.05946,这里 #C 比 C 高半个音 */
float play—fd=1.5; /* 在音符后每加一个圆点(.),可使音符演奏时间延长一半,1.5 就是描述它的
比例系数。例如,"A4"演奏为 1/4 拍,则"A4."演奏 1.5×1/4 拍,而"A4.."
则演奏的音长为 1.5×(1.5×1/4) 拍等等 */

float play—l=4.0; /* 四分音符 */
float play—p=4.0;
float play—ll=4.0;
float play—pp=4.0;
```

```

int play—o=3; /* 缺省音阶 */
int play—t=120; /* 演奏速度,每分钟演奏 120 个四分音符 */
int play—mode=0; /* 当遇 ^ (表示连音符) 时为 1, 遇 x (结束连音演奏) 时为 0, x 出现在最
 后一个连音符的前面 */
int play—err=0; /* 检查用错误标志。为 1 表示出错 */

float get—freq(int yj, char *yf) /* 求频率函数 */
{
 /* yj=play—o 为音阶初值, *yf 指向音乐串 */
 int i,m,n;
 char y,x; /* y 记录 yf[0], x 记录 yf[1] */
 if(yj<0 || yj>12) return 0;
 i=yj+12;
 y=toupper(yf[0]); /* 转成大写 */
 x=yf[1]; /* 可在此行设一断点,用 Ctrl—F9 操作,并观察 yf 调试表达式值 */
 if(y<'A' || y>'G') return 0; /* 对无效音符处理 */
 n=y-'A'-2; /* 例如,用 Ctrl—F4 输入调试表达式 'B'-'A'-2 得 -1 */
 if(n<0) n += 7;
 n *= 2; /* 得音串中的当前实际 (半) 音阶值 */
 if(x == '#' || x == '+')
 {
 /* 程序处理的升音符是放在音符后面,跟乐谱中不同 */
 if(y=='E' || y=='B') return 0; /* 处理不允许的升音 */
 n++;
 }
 if(x == '-') /* 程序不允许用乐谱中的 b 作降音修饰符 */
 {
 if(y=='C' || y=='F') return 0; /* 处理不允许的降音符 */
 n--;
 }
 if(n>4) n--;
 m=i+n; /* pow() 的指数值,它的底是 yf—x ≈ 1.05946 */
 yf—freq=yf—low * pow(yf—x,m); /* 使用公式求得实际频率 */
 return(yf—freq);
}

float play—time(float n) /* 求演奏时音调持续时间 */
{
 float t=0;
 t=60 * YF—1/(n * play—t);
 return t;
}

void play—music(int yj,char *yf,float time) /* 演奏处理 */
{
 if(yj<0) yj=0;
 if(yj>PLAY—O—MAX) yj=PLAY—O—MAX;
 yf—freq=get—freq(yj,yf);

```

```

if(yf—freq==0 && yf[0] != 'P')
{
 printf("\n串 \"%s\" 不对! \x7\n",yf); /* 响铃告警 */
 nosound();
 exit(-1);
}
if(yf—freq==0)nosound(); /* 频率为 0 时停止蜂鸣 */
sound(yf—freq); /* 演奏 */
delay(time * 1000); /* 音调持续时间,此值应适当 */
if(play—mode==0)nosound(); /* 遇连音结束关闭扬声器 */
}

```

```

reterr(int i) /* 错误处理 */
{
 play—yf[0]=0,play—yf[1]=0; /* 清 0 */
 play—err=1; /* 置错误标志 */
 return i; /* 返回 i */
}

```

```

play—decimal(char *aso,int type,int j) /* 处理多个符点 */
{
 while(1)
 {
 if(aso[i]=='.')
 {
 j++;
 if(type == 'P') play—p=play—p/play—fd;
 else play—l=play—l/play—fd;
 }
 else
 {
 j--; break; /* 退出循环 */
 }
 }
 return j;
}

```

```

play—num(char *aso,int i,int len) /* 处理音符后数字 */
{
 char type;
 int m=0,n=0,max=PLAY—MAX,min=PLAY—MIN;
 type=aso[i];num="";
 switch(type)
 {
 case 'C': /* 音符,基本音 1 */
 case 'D':

```

```

case 'E';
case 'F';
case 'G';
case 'A';
case 'B', play-yf[0]=type; /* 基本音 7 */
 play-yf[1]=0;
 switch(aso[i+1]) /* 半音规定写在音符后面,这跟乐谱不同 */
 {
 case '#'; /* 升高半音 */
 case '+';
 case '-'; /* 降低半音 */
 i++; play-yf[1]=aso[i]; return i;
 case '.': i++; i= play-decimal(aso,type,i);
 return i;
 default: return i;
 }
case 'O', max=PLAY-O-MAX,min=PLAY-O-MIN;break; /* 音阶 */
case 'P', play-yf[0]=type; /* 休止符 */
 play-yf[1]=0;
 break;
case 'L', /* 音符演奏时间 */
case 'T', break; /* 拍子设定,音乐演奏速度 */
case 'X', play-mode=0; return i; /* X 结束连音演奏,它应出现在最后一个连音符前面 */
case '^', play-mode=1; return i; /* ^ 后面的音符进行连音演奏 */
case '<', play-o=play-o-1; return i; /* 降低一个音阶 */
case '>', play-o=play-o+1; return i; /* 升高一个音阶 */
default: return i; /* 可能是小节线 | 或空格符等,它们不被演奏 */
}
while(1)
{
 if(i>len || n>=NUM-MAX) reterr(i);
 if(isdigit(aso[i+1])) i = 0
 {
 i++;
 num[n++] = aso[i];
 }
 else break;
}
if(n>0)
{
 num[n]=0;
 play-err=0;
 m=atoi(num); /* 将字符串转为数字 */
 if(m<min || m>max) reterr(i); /* 检查音阶 m */
 switch(type)
 {

```

```

 case 'T':play--t=m;return i;
 case 'O':play--o=m;return i;
 case 'L':play--l=m;play--ll=m;return i;
 case 'P':play--p=m;play--pp=m;i++;
 }
}

if(type != 'P') reterr(i);
i++;
i=play--decimal(aso,type,i);
return i;
}

play(char * aso)
{
 int i,len,times,next;
 char type;
 aso=strupr(aso); /* 将串中字符全转为大写字母 */
 len=strlen(aso); /* 求出串长 */
 if(len==0) return 0;
 for(i=0;i<len;i++) /* 处理串每一个字符 */
 { if(kbhit() != 0)
 { getch();exit(0);} /* 正在演奏时如按任一键便结束演奏 */
 play--err=0;
 type=aso[i];
 next=i; /* next 记录 i 原值 */
 play--yf[0]=0;
 play--yf[1]=0;
 play--l=play--ll; /* 每次从上次选的音阶开始 */
 play--p=play--pp;
 i=play--num(aso,i,len);
 if(play--err != 0) /* 如果发生了错误 */
 {
 nosound(); /* 关闭扬声器 */
 if(i<len-1) i++;
 printf("\n 乐串(\"");
 for(times=next;times<i;times++) /* 打印出错子串 */
 printf(" %c", (char)aso[times]);
 printf("\")是错误的! \007\n"); /* 响铃告警 */
 exit(-1);
 }
 }
 if(play--yf[0] != 0)
 {
 times=play--l;
 if(type == 'P')times=play--p; /* 停顿时间 */
 play--music(play--o,play--yf,play--time(times));
 }
}

```



```

 } /* 一个乐曲串演奏结束 */
 nosound();
 return 0;
}
#endif

```

/\* 歌曲《潇洒走一回》内容（注音未标）：

```

6 7 1 2 3 | 3 2 1 7 | 6 7 1 7 6 | 6 - - - | 6 7 1 2 3 | 3 6 3 3 2 2 | 1 1 1 2 1 3 | 3 - - - |
天地悠悠过 客匆匆 潮起又潮落 恩恩怨怨生 死白 头 几人能 看透

6 6 6 6 3 3 | 5 5 5 5 3 2 | 2 1 1 1 5 3 | 3 - - - | 6 6 6 6 7 6 | 5 5 5 5 3 2 | 1 1 7 6 5 |
红尘呀滚 滚 痴痴呀情深 聚 散总有时 留一半清醒 留一半醉至少 梦里有你追

6 - - - | 2 2 2 3 5 3 5 | 6 - - - | 1 1 1 1 7 6 5 | 6 5 5 - - | 2 2 2 3 5 4 3 | 2 2 3 5 3 3 |
随 我拿青春赌明 天 你用真情换此 生 岁月不知人 间 多少的忧 伤

2 2 2 3 5 1 7 | 6 - - - |
何不潇洒走一 回

```

音符对照：1→c、2→d、3→e、4→f、5→g、6→a 和 7→b

```

/*
main() /* 根据耿兴华的程序改写并注释 */
{
 int k;
 char *music[]={"T200 O2 \
L4ab>cL8d^e xL4edc<b ab>c<L8b^a xL1a \
L4ab>cL8d^e xL4ea^L8eexdd cc^L8cxd c^e xL1e \
L8aL4aL8aa^e xL4e L8gL4gL8gL4gL8^exd dxcL4ccL8g^e xL1e \
L8aL4aL8aL4a^L8bxa gL4gL8gL4gL8ed L4cc<L8baL5g L1a \
>L8dddeL4g^L8exg <L1a >>L8ccccL4<bL8^axg ^agL2xg. \
L8dddeL8^gxfl4e L8dL4dL8e^gxel4e L8dd^dxel4g^L8>cx<b aL1a ",
 "T100 O2 \
L4ab>cL18d^e xL4edc<b ab>c<L8b^a xL1a \
L4ab>cL8d^e xL4ea^L8eexdd cc^L8cxd c^e xL1e \
L8aL4aL8aa^e xL4e L8gL4gL8gL4gL8^exd dxcL4ccL8g^e xL1e\
L8aL4aL8aL4a^L8bxa gL4gL8gL4gL8ed L4cc<L8baL5g L1a \
>L8dddeL4g^L8exg <L1a >>L8ccccL4<bL8^axg ^agL2xg. \
L8dddeL8^gxfl4e L8dL4dL8e^gxel4e L8dd^dxel4g^L8>cx<b \
^L8baxL2a. ||"};
 for(k=0;k<2;k++)
 play(music[k]);
}

```

## 第三十七章 搜索与排序函数

一 从指定表中搜索出指定值的库函数有：

- 对已知表中数据快速排序 —1 qsort()
- 从已排序的表中搜索出指定值（二分法搜索） —2 bsearch()
- 表中数据未排序，线性搜索表中指定值。若未  
表中未找到指定值，则指定值将添加到表中 —3 lsearch()
- 表中数据未排序，线性搜索表中指定值。若  
在表中未找到指定值，但指定值不添加到表中 —4 lfind()

二 自定义函数

- 对数组 Quicksort 排序 —5 qs()
- 6 qn()
- 多重排序 —7 manysort()

```
—1 void —Cdecl qsort(void *base,size—t nelem,size—t width,
int —Cdecl (*fcmp)(/* const void *,const void * */));
```

函数使用快速排序例程对一个待排序表进行排序。base 指向待排序表的起始位置（第 0 个元素），nelem 是表中元素的个数，width 是以字节为单位的表中元素的长度，对比较函数 fcmp() 有较严格的限制（参见例程 SEARCH1.C）。

函数使用的是“中树遍历”的算法，据说是由 C. A. R. Hoare 开发的。所谓中树遍历是指对一棵树，你总是往左走，一直走到端结点（或目标）为止；当遇到端结点后，则退回上一层，向右走，然后再往左走，直到遇到端结点为止。重复这个过程，直到找到目标或检查完树的最后一个结点为止。

经排序后的表，数组以升序排列，即最小地址存最小数。

它不返回值。适用于 UNIX 系统。

C>TYPE SEARCH1.C

```
#include "stdlib.h"
#include "stdio.h"
#define P(X) printf("#X=");for(k=0;k<4;k++)printf("%d ",colors[k]);\
printf("\n")
fcmp(const void *c1,const void *c2) /* 规定接收参数形式 */
{
 if(* (int *)c1>* (int *)c2)return 1; /* 规定返回值的分类 */
 else if(* (int *)c1<* (int *)c2)return -1;
 return 0;
/* 这三个语句对 int,char 类型可用;return (* (int *)c1-*(int *)c2);
语句代替,但是对 float,double 等可不能这样做,它们的结果是不一样的 */
}
```

```

main() /* 对整数 (int 类型) 数字表快速排序 */
{
 static int colors[4]={421,9898,1,2};
 int k,ncolors=4;
 P(1);
 qsort(colors,ncolors,sizeof(int),fcmp);
 P(2);
}
/* 程序输出:1=421 9898 1 2
 2=1 2 421 9898
但当用 F7 进行调试时,可以发现光标不会跟踪进入函数 fcmp()!
*/

```

—2 void \*Cdecl bsearch(const void \*key,const void \*base,size\_t nelem, size\_t width,int —Cdecl (\*fcmp)(/\* const void \*,const void \* \*/));  
任意一组数据可以构成一个【搜索表】,例如定义的数组

```
int table[]={123,146,512,600,700,810};
```

是一个搜索表。函数用于从搜索表中找出指定的一个数字,并且在搜索前搜索表已经进行过从小到大的排序。

size\_t 在 stdlib.h 中定义为 typedef unsigned size\_t;

key(可称关键字)指向要寻找的数字,base 指向搜索表的基址(第 0 个元素),nelem 是搜索表中元素的个数(整数),width 是搜索表中每一个元素的一个项所占的字节数。确切地说,width 可能是 sizeof(int)、sizeof(char) 或 sizeof(double) 等等,而不是指表中元素所占字节数,尽管有时两者的值是相同的。fcmp 是返回整数的函数指针,注释中的说明表示它可以接受两个指针。搜索时函数反复调用由 fcmp 传递的子程序。该子程序是一个比较子程序,它接受的两个参数,第一个指向 key,第二个指向从搜索表中取出的待比较的元素。如果两者相等,返回 0。

如果找到指定元素,返回与搜索关键字相匹配的第一个表项地址,否则返回 0。

在算法上它采用了【对分搜索】(【或二分法搜索】),即先将搜索表一分为二,变成左右两部分。先将 key 减去左边部分的最右边的数,如大于 0,由于搜索表中的数字是由小到大排列的,因而说明左边部分无 key,转去右边部分按二分法比较;否则,再将左边部分一分为二用上法比较,直至当差为 0 时表明 key 被找到,否则说明 key 不在搜索表中。

容易看出,本函数不适用于搜索表元素为字符串的情况。

它适用于 UNIX 系统。

C>TYPE SEARCH2.C

```

#include "stdio.h"
#include "stdlib.h"
#define NELEMS(arr) (sizeof(arr)/sizeof(arr[0])) /* 求出搜索表中元素个数 */
int table[]={1,3,4,5,19}; /* 定义搜索表,已排序 */
{
 printf("%d %d %d\n",*p1,*p2,*p1-*p2);
 return (*p1-*p2); /* 必定返回形式 */
}

```

```

int lookup(int key)
{
 int *item;
 item=(int *)bsearch(&key,table,NELEMS(table),sizeof(int),fcmpe);
 printf("指针=%p, %x\n",item,*item);
 return (item != NULL); /* 指针 item 非空时返回 1 */
}

main()
{
 int x;
 while(1) /* 括号中为数字 1 */
 {
 printf("输入表中一个数(输0退出)");
 scanf("%d",&x); /* 不能将这两句并成一句: */
 /* scanf("输入表中一个数(输0退出)%d",&x); */
 if(x==0)exit(0);
 printf("数字%d%s\n",x,lookup(x)?" 被找到,OK!":" 没有找到");
 }
}
/* 程序输出:输入表中一个数(输0退出)1
1 4 -3
1 3 -2
1 1 0
指针=0194, 1
数字1 被找到,OK!
输入表中一个数(输0退出)2
2 4 -2
2 3 -1
2 1 1
指针=0000, 0
数字2 没有找到
输入表中一个数(输0退出)19
19 4 15
19 19 0
指针=019C, 13
数字19 被找到,OK!
输入表中一个数(输0退出)0
*/

```

如果搜索表中的元素为浮点数,应注意语句的正确应用。

C>TYPE SEARCH3.C

```

#include "stdio.h"
#include "stdlib.h" #define NELEMS(arr) (sizeof(arr)/sizeof(arr[0]))
double table[]={1.1,3.2,4,5,19}; /* 表中元素为浮点数 */
int fcmpe(double *p1,double *p2) /* 注意函数类型一定为 int */

```

```

{
printf("%f %f %f\n", *p1, *p2, *p1 - *p2);
return (int)(*p1 - *p2);
}

int lookup(double key)
{
int *item;
item=(int *)bsearch(&key,table,NELEMS(table),sizeof(double),fcmp);
printf("指针=%p, %x\n",item,*item);
return (item != NULL);
}

main()
{
double x;
while(1)
{
printf("输入表中一个数(输0退出)");
scanf("%lf",&x); /* 不能将这句写成:scanf("%f",&x); */
if(x==0.0)exit(0);
printf("数字%f%s\n",x,lookup(x)? " 被找到,OK!" : " 没有找到");
}
}

```

—3 void \*Cdecl lsearch(const void \*key,const void \*base,size\_t \*num,  
size\_t width,int —Cdecl (\*fcmp)(/\* const void \*,const void \* \*/));

本函数的参数含义同 bsearch(), 但 bsearch() 中的 nelem 现在变成 \*num。函数用于对搜索表进行线性搜索, 即搜索时从表中的第一个元素开始, 依次进行比较, 直至将 key 与最后一个元素比较。比较中如发现有一个相等, 比较便结束。因此, 在调用本函数前无须先对搜索表进行排序。

函数指针 fcmp 可以是 string.h 中提供的比较函数 strcmp() 或 stricmp()。  
如果找到指定元素, 返回与搜索关键字相匹配的第一个表项地址, 否则返回 0。  
如果 key 不在搜索表中, 函数将把它添加到搜索表中。它适用于 UNIX 系统。

#### C>TYPE SEARCH4.C

```

#include "stdlib.h"
#include "stdio.h"
#include "string.h"

int ncolors=3;
cmp(const void *c1,const void *c2)
{
char *c,*ch;
c=c1,ch=c2;
printf("%s %s\n",c,ch);
if(strlen(c) != strlen(ch)){printf("111\n");return 1;}
while(*c != '\0')

```

```

if(*c++ | = *ch++){printf("222\n");return 1;}
printf("0000\n");return 0;
}

int addelem(char *color)
{
 char *colors[4]={"red","blue","green",""}, /* 指针数组 */
 int n,oldn;
 oldn=ncolors=strlen(colors[i]);
 lsearch(color,colors[i],&ncolors,sizeof(char),cmp);
 /* 编译时会出现 Suspicious pointer conversion 的警告,可抑制它 */
 /* 你也可用原型在 string.h 中的函数 strcmp 代替 cmp,可得同样结果 */
 for(n=0;n<4;n++)printf("%s",colors[n]);
 printf("\n");
 return(ncolors==oldn);
}

main() /* 从指定表中线性搜索指定字符串 */
{
 static char s[20];
 int oldc=ncolors;
 printf("输入一个颜色");
 gets(s);
 printf("\n");
 for(i=0;i<oldc;i++)
 (printf("开始...");
 if(addelem(s))
 {
 printf("颜色%s 早在颜色表中,共有%d 种颜色\n",s,oldc);
 return 0;
 }
 if(i | = (oldc-1))ncolors=oldc;
 }
 printf("指定颜色%s 不在颜色表中!\n",s);
}

/* 可输入red 等颜色名,现在假定输入了两个字母QW(非表中颜色)后程序输出:
 输入一个颜色
 开始...QW red
 111
 QW ed 每次向后移动 size-t width 个字节
 222
 QW d
 111 如果由 key 所指项不在表中,函数将它添加到表中
 redQblue,blue,green,, 字母 Q 占据了原 '\0' 位置,故印出 redQblue, 而
 开始...QW blue 字母 W 丢失,因这里 width=1. 用监视表达式也有
 111 相同结果,下同
 QW lue

```

```

111
QW ue
222
QW e
111
redQblueQgreen,blueQgreen,green,,
开始... QW green
111
QW reen
111
QW een
111
QW en
222
QW n
111
redQblueQgreenQ,blueQgreenQ,greenQ,,
指定颜色QW 不在颜色表中!
*/

```

```

—4 void * Cdecl lfind(const void * key,const void * base,size_t * num,
size_t width,int —Cdecl (* fcmp)(/* const void *,const void */));

```

lsearch() 实现的功能也可靠本函数调用得到。但是,它们之间有区别:当 key 不在表中时,lsearch() 将把它添加到表中,而本函数则不这样做。因此,在使用本函数时应将 addelem() 函数略作修改,变成如下形式:

```

int addelem(char * color)
{
char * colors[4]={"red","blue","green",""};
int n,* lf;
ncolors=strlen(colors[i]);
lf=(int *)lfind(color,colors[i],&ncolors,sizeof(char),cmp);
for(n=0;n<4;n++)printf("%s,",colors[n]);
printf("\n");
return(lf!=NULL);
}

```

下列子程序也许能帮助你理解它们之间的区别。

```

static void * near pascal —lsearch(const void * key,register void * base, size_t * nelem,size_t
width,int cdecl (* fcmp)(const void *,const void *),int flag)
{
register int Wrk;
for(Wrk=* nelem;Wrk>0;Wrk--)
{
if(((* fcmp)(key,base))==0)return (base); /* 如果相等 */
((char *)base) += width; /* 字符指针 base 移动 */
}
if(flag) /* flag 不是 0 时 */

```

```

 {
 (*nelem)++; /* nelem 将增加 */
 movmem(key,base,width); /* 把 width 字节的块从 key 拷贝到 base */
 }
 else base=NULL; /* flag 是 0 时 */
 return(base);
}

void *lsearch(const void *key,void *base,size_t *nelem,
 size_t width,int cdecl (*fcmp)(const void *,const void *))
{
 return(--lsearch(key,base,nelem,width,fcmp,1)); /* flag=1 */
}

void *lfind(const void *key,const void *base,size_t *nelem,
 size_t width,int cdecl (*fcmp)(const void *,const void *))
{
 return(--lsearch(key,base,nelem,width,fcmp,0)); /* flag=0 */
}

```

使用 `lfind()` 或 `lsearch()` 也可以对数值表进行搜索,例如,对 `STDLIB1.C` 中的 `lookup` 函数作如下修改,便可得同样结果:

```

int lookup(int key)
{
 int *item;
 int y=NELEMS(table);
 item=(int *)lfind(&key,table,&y,sizeof(int),fcmp);
 printf("指针=%p, %x\n",item,*item);
 return (item != NULL); /* 指针 item 非空时返回 1 */
}

```

在 `ctype.h` 中定义了一个宏 `isalpha(c)`,它是根据 `c` 的位值判断其是否为字母的。这里可用下列程序检查输入字符是否为字母:

```

C>TYPE SEARCH5.C
#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"
char table[26]="abcdefghijklmnopqrstuvwxyz"; /* 搜索表 */
main() /* 检查输入字符是否在搜索表中 */
{
 char c;
 char *p;
 int num=26;
 int comp();
 do{
 printf("请输入一个字母:");
 scanf("%c",&c); /* 此句不能写成 scanf("%c",&c), 因为每输入一
 /* 个字符后要回车,%c 便是用来接收回车符的。如 */
 }while(!isalpha(c));
}

```



```

/* 它去掉,则按的回车符将被当成输入的第二个字符 */
/* 而回车符显然不是字母,因此按回车后循环便结束 */
/* 百分号后面的星号表示当前输入字段被扫描但不存 */
/* 储。为理解这一点,你可增加一个变量 a,例如, */
/* static int c,a; scanf("%c%c",&c,&a); 然后用 */
/* 调试表达式 c,4m 和a,4m 进行观察,便可明白其意 */
c=tolower(c); /* 把字符转换成小写字母,因此输入时可不管大小写 */
p=(char *)lfind(&c,table,&num,1,comp);
if(p)printf("是一个字母\n");
else printf("非字母\n");
}
while(p); /* 如输入一个字母,则循环继续, */
/* 输入一个非字母后自动退出循环 */

comp(const void *ch,const void *s)
{
return (*(char *)ch-*(char *)s); /* 不允许写成:return (*ch-*s); */
} /* 这是 void * 类型指针对此的限制 */

```

Quicksort 排序法是由 C. A. R. Hoare 发明的,是一种好理解排序又快的方法。它的基本原理如图 37-1 所示,先找出一个分界值元素,按划分原则把  $n$  个元素分成两部分;然后对划分出来的两部分又分别重复这个过程(在程序中采用递归实现),直到每个部分无元素要划分为止。

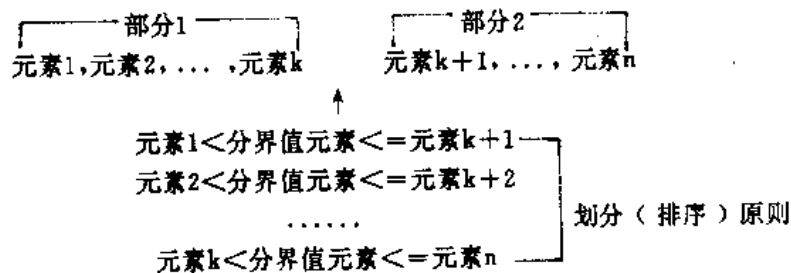


图 37-1

—5 qs(char \*literal[],int left,int right);  
 对字符数组 literal 的 left 到 right 的元素 Quicksort 排序。  
 —6 qn(unsigned long \*numeral,int left,int right);  
 对数字数组 numeral 的 left 到 right 的元素 Quicksort 排序。

```

C>TYPE SEARCH6.C
#include "string.h"
#define ITEM—NUM 5
main()
{
char *s[]={"fde","fd","a","asd","fd"}; /* 原始字符数组 */
long t[]={5,8,3,10,8}; /* 原始数字数组 */
int n=ITEM—NUM; /* 元素个数 */
qs(s,0,n-1); /* 对字符数组排序 */
printf("s=");

```

```

for(n=0;n<ITEM-NUM;n++)printf("%s",s[n]);
printf("\b \nt=");
qn(t,0,n-1); /* 对数字数组排序 */
for(n=0;n<ITEM-NUM;n++)printf("%lu",t[n]);
printf("\b \n");
}
qn(unsigned long *numeral,int left,int right)
{ /* 数值按由小到大输出 */
int i,j;
unsigned long x,y;
i=left;j=right;
x=numeral[(left+right)/2]; /* 取分界值元素为中间值 */
do { /* 如将下两句中 <x 与 x< 的小于符号改成大于,则结果由大到小输出 */
while(numeral[i]<x && i<right) i++; /* i 向右方向增大 */
while(x<numeral[j] && j>left) j--; /* j 向左方向减小 */
if(i<=j) {
y=numeral[i]; /* 交换 */
numeral[i]=numeral[j];
numeral[j]=y;
i++;j--;
}
}while(i<=j);
if(left<j)qn(numeral,left,j); /* 采用递归 */
if(i<right)qn(numeral,i,right);
}

qs(char *literal[],int left,int right) /* 执行过程基本上同 qn() */
{ /* 字符按 ASCII 值由小到大输出 */
int i,j;
char *x,*y;
i=left;j=right;
x=literal[(left+right)/2];
do { /* 如将下两句中 <0 与 >0 的小于或大于符号反向,则结果由大到小输出 */
while(strcmp(literal[i],x)<0 && i<right)i++;
while(strcmp(literal[j],x)>0 && j>left)j--;
if(i<=j) {
y=literal[i];
literal[i]=literal[j];
literal[j]=y;
i++;j--;
}
}while(i<=j);
if(left<j)qs(literal,left,j);
if(i<right)qs(literal,i,right);
} /* 程序输出:s=a,asd,fd,fd,fde t=3,5,8,8,10 */

—7 mansort(char *sor,char *s[],char *v[],long t[],long r[],int n,

```

```
int directions,int sflags);
```

多重排序用于将多组数据逐级分类,先排序的数组制约后排序的数组。多重排序应用范围很广,例如在电子表格软件中就用它(可参见作者编写的《高级汉字自动制表软件OFFICE使用技巧》一书的第二章的“置换操作”部分),因而它也受到C爱好者的青睐。这里给出一个多重排序程序 SORT.C,它最多可指定4个排序关键字(显然,从该程序可见关键字个数可随意增减)。多重排序结果分析如图 37-2 所示。

|     | ① | ② | ③ | ④ | ← 数组排序顺序 |
|-----|---|---|---|---|----------|
| 列 → | t | s | v | r | ← 排序用关键字 |
| 行 ↓ | 1 | A | d | 8 |          |
|     | 1 | A | e | 2 |          |
|     | 2 | B | f | 4 |          |
|     | 3 | C | f | 3 |          |
|     | 3 | C | f | 7 |          |
|     | 3 | C | f | 9 |          |
|     | 3 | 3 | h | 6 |          |
|     | 4 | D | f | 1 |          |
|     | 4 | D | f | 5 |          |

图 37-2

可在集成环境下调试该程序,通过观察调试表达式的值理解其采用的算法。假定还确定某些规约,如当关键字为数字时,遇正文字符按 0 计,或者对正文关键字,遇数字按空格计,则对一系列上元素有字符、数字混杂时,程序稍经修改也能处理。

```
C>TYPE SORT.C
```

```
#include "string.h"
#define ITEM- NUM 9 /* 每个关键字对应的元素个数(行) */
#define NUM 4 /* 待排序关键字个数 (列) */
#define MAXELE 500 /* 与排序段相关的临时变量个数 */
#define PSVTR printf("t=");
for(z=0;z<ITEM- NUM;z++)printf("%lu," ,t[z]);\
printf("\b \ns=");
for(z=0;z<ITEM- NUM;z++)printf("%s," ,s[z]);\
printf("\b \nv=");
for(z=0;z<ITEM- NUM;z++)printf("%s," ,v[z]);\
printf("\b \nr=");
for(z=0;z<ITEM- NUM;z++)printf("%lu," ,r[z]);\
printf("\b \n")
/* 待排序串可以有多个不同字符,串与串之间字符个数也可以不等,但多个数组的元素个数应
相等。下面采用了一组最简单的数据。*/
char *s[]={ "D", "A", "C", "B", "D", "C", "C", "A", "C"};
char *v[]={ "f", "e", "f", "f", "f", "h", "f", "d", "f"};
long t[]={ 4, 1, 3, 2, 4, 3, 3, 1, 3};
long r[]={ 1, 2, 3, 4, 5, 6, 7, 8, 9};
int o[NUM]; /* 是否已排过序标志 0→s,1→v,2→t,3→r */
char sor[NUM]="tsvr"; /* 指定排序关键字的先后顺序 */
int ns; /* 记录下次分段排序对数 */
int mmns[MAXELE]; /* 每两个数组元素记录一个排序段范围(起始、终止) */
```

```

int fol[MAXELE],dow[MAXELE]; /* fol 记录前次排序后上下元素相同情况,dow 则记录本次排序
 情况,为下次排序作准备 */
int nn; /* 中间变量,记录相同元素个数 */
main()
{
 int z;
 printf("未排序前:\n"); PSVTR;
 manysort(sor,ITEM-NUM,0,1);
 printf("排序后为:\n"); PSVTR;
}

manysort(char *sor, int n, int directions, int sflags)
{
 /* directions=1 表示正向排序,sflags=1 表示要区分大小写字母 */
 int u, c=0;
 ns=1;
 mmns[1]=n-1;
 for(c=0; c<strlen(sor); c++) /* 多重排序循环开始 */
 {
 if(sor[c]=='\0')return;
 switch(sor[c])
 {
 case 's':
 if(! ns)break; /* 如无相同元素则不再排序 */
 for(u=0; u<ns; u++) /* 分段排序 */
 qs(s, mmns[u*2], mmns[u*2+1], directions, sflags);
 o[0]=1; comps(s, 0, n, 1); break; /* 作标志,分析新数组 */
 case 'v':
 if(! ns)break;
 for(u=0; u<ns; u++)
 qs(v, mmns[u*2], mmns[u*2+1], directions, sflags);
 o[1]=1; comps(v, 0, n, 1); break;
 case 't':
 if(! ns)break;
 for(u=0; u<ns; u++)
 qn(t, mmns[u*2], mmns[u*2+1], directions);
 o[2]=1; compn(t, 0, n); break;
 case 'r':
 if(! ns)break;
 for(u=0; u<ns; u++)
 qn(r, mmns[u*2], mmns[u*2+1], directions);
 o[3]=1; compn(r, 0, n); break;
 }
 }
 /* 多重排序结束 */
}

qn(unsigned long *numeral, int left, int right, int direction)
{
 /* 对数字使用 Quicksort 排序 */
 int i, j;

```

```

unsigned long x, y;
i=left, j=right, /* left 与 right 为指定排序元素的范围 */
x=numeral[(left+right)/2];
do {
 if(direction)
 {
 /* 逆向排序 */
 while(numeral[i]>x && i<right) i++;
 while(x>numeral[j] && j>left) j--;
 }
 else
 {
 /* 正向排序 */
 while(numeral[i]<x && i<right) i++;
 while(x<numeral[j] && j>left) j--;
 }
 if(i<=j) {
 swasn(i,j);
 i++; j--;
 }
} while(i<=j);
if(left<j) qn(numeral, left, j, direction); /* 递归 */
if(i<right) qn(numeral, i, right, direction); /* 递归 */
}
qs(char * literal[], int left, int right, int direction, int sflag) { /* 对正文使用 Quicksort
排序 ,sflag=1 区分字母大小写 */
 int i, j;
 char * x;
 i=left, j=right;
 x=literal[(left+right)/2];
 do {
 if(direction)
 {
 /* 逆向排序 */
 if(sflag)
 {
 /* 区分大小写 */
 while(strcmp(literal[i], x) >0 && i<right) i++;
 while(strcmp(literal[j], x) <0 && j>left) j--;
 }
 else
 {
 /* 忽略大小写 */
 while(stricmp(literal[i], x) >0 && i<right) i++;
 while(stricmp(literal[j], x) <0 && j>left) j--;
 }
 }
 else
 {
 /* 正向排序 */
 if(sflag)
 {
 /* 区分大小写 */

```

```

 while(strcmp(literal[i], x) <0 && i<right)i++;
 while(strcmp(literal[j], x) >0 && j>left)j--;
 }
 else
 {
 /* 忽略大小写 */
 while(stricmp(literal[i], x) <0 && i<right)i++;
 while(stricmp(literal[j], x) >0 && j>left)j--;
 }
}
if(i<=j) {
 swasn(i,j); /* 交换元素 */
 i++;j--;
}
} while(i<=j);
if(left<j) qs(literal, left, j, direction, sflag);
if(i<right) qs(literal, i, right, direction, sflag);
}
swasn(int il, int jl) /* 根据排序与否决定是否要交换元素 */
{
 long x;
 char *y;
 if(! o[0]) { y=s[i1]; s[i1]=s[j1]; s[j1]=y; }
 if(! o[1]) { y=v[i1]; v[i1]=v[j1]; v[j1]=y; }
 if(! o[2]) { x=t[i1]; t[i1]=t[j1]; t[j1]=x; }
 if(! o[3]) { x=r[i1]; r[i1]=r[j1]; r[j1]=x; }
}
comps(char *s[], int e, int k, int sflag) /* 对正文分析排序后元素 */
{
 /* sflag=1 区分大小写 */
 int xx, yy, ii, jj, dl;
 char *ss=(char *) malloc(128);
 memset(mmns, '\0', 500);
 strcpy(ss, s[e]);
 nn=ii=1; xx=ns=jj=dl=0;
 for (yy=e+1; yy<k; yy++) /* 每次分析从指定元素开始 */
 {
 if(sflag) xx=strcmp(ss, s[yy]);
 else xx=stricmp(ss, s[yy]);
 if(! xx && fol[yy]==fol[yy-1]) /* 约束条件 */
 {
 if(! jj) ++ns;
 mmns[nn]=yy;
 jj=1;
 dow[yy]=dl;
 }
 else
 {
 jj=0;
 if(++ii==2) {ns++; mmns[nn]=yy-1; ii=1;

```

```

 }
 mmns[++nn] = yy;
 nn++;
 dow[yy] = ++d1;
}
strcpy(ss, s[yy]);
if(yy == k-1 && ++ii == 2) mmns[nn] = yy;
}
for(yy=0; yy<k; yy++) fol[yy] = dow[yy];
}
comprn(long *nt, int e, int k) /* 对数字分析排序后元素 */
{
 int yy, ii, jj, d1;
 long t1 = nt[e];
 nt++; memset(mmns, 0, 500); /* 必须初始化 */
 nn = ii = 1; ns = jj = d1 = 0;
 for (yy=e+1; yy<k; yy++)
 {
 if(t1 == *nt && fol[yy] == fol[yy-1])
 {
 if(!jj) ++ns;
 mmns[nn] = yy; jj = 1;
 dow[yy] = d1;
 }
 else
 {
 jj = 0;
 if(++ii == 2) { ns++; mmns[nn] = yy-1; ii = 1; }
 mmns[++nn] = yy;
 nn++;
 dow[yy] = ++d1;
 }
 t1 = *nt++;
 if(yy == k-1 && ++ii == 2) mmns[nn] = yy;
 }
 for(yy=0; yy<k; yy++) fol[yy] = dow[yy];
} /* 程序输出:
 未排序前:
 t=4,1,3,2,4,3,3,1,3
 s=D,A,C,B,D,C,C,A,C
 v=f,e,f,f,f,h,f,d,f
 r=1,2,3,4,5,6,7,8,9
 排序后为:
 t=1,1,2,3,3,3,3,4,4
 s=A,A,B,C,C,C,C,D,D
 v=d,e,f,f,f,f,h,f,f
 r=8,2,4,3,7,9,6,1,5 */

```

## 第三十八章 对 ANSI 定义信号对应的动作重定义

### 38.1 库函数

—1 void (\* —Cdecl signal(int sig, void (\* func)(/\* int \*/)))(int);

该函数用于定义信号 sig 所对应的动作,动作由函数名 func 指定。信号是由 raise() 函数或发生异常情况时产生的。当信号产生时对应的函数 func 便被执行。缺省的动作函数名为

```
#define SIG_DFL ((void (* —Cdecl)(int))0) /* 缺省动作,终止程序执行 */
#define SIG_IGN ((void (* —Cdecl)(int))1) /* 忽略动作即该信号不会产 */
#define SIG_ERR ((void (* —Cdecl)(int))-1) /* 返回错误码 */
```

对 OS/2 机有:

```
#define SIG_SGE ((void (* —Cdecl)(int))3) /* 信号接通错误 */
#define SIG_ACK ((void (* —Cdecl)(int))4) /* 信号应答 */
```

定义的这些宏有助于你理解 signal() 的原型,引用它们时可用像

signal(SIGFPE, SIG\_DFL);

即可。用户可以定义自己的动作函数名。

信号 sig 规定为:

```
#define SIGABRT 22 /* 退出 */
#define SIGFPE 8 /* 浮点陷阱 */
#define SIGILL 4 /* 非法指令 */
#define SIGINT 2 /* 中断 */
#define SIGSEGV 11 /* 存储存取违反 */
#define SIGTERM 15 /* 终止 */
```

对 OS/2 机有:

```
#define SIGBREAK 21 /* OS/2 Ctrl-Brk 信号 */
#define SIGUSR1 16 /* OS/2 处理标志 A */
#define SIGUSR2 17 /* OS/2 处理标志 B */
#define SIGUSR3 20 /* OS/2 处理标志 C */
```

用户自己定义的信号值可能无效,或者说此时仅管用了 raise(自己定义的信号)激发,但相应的动作函数不起作用。

应当指出, signal.h 中定义的该函数的原型和帮助文件中提供的是不同的,在那里

```
typedef void (* sigfun)(int subcode);
int signal(int sig, sigfun fname);
```

这是 Turbo C 定义的函数,而 signal.h 中定义的是 ANSI 中定义的形式,两者稍有不同。如果将它们写在源程序中,则源程序不应包括 signal.h,或者虽包括但在源程序一开始便加上定



## 义语句

```
#define ——SIGNAL—H
```

不然在编译时会出现重复定义错误。

```
C>TYPE SIGNAL1.C
```

```
#include "stdio.h"
#include "signal.h"
main()
{
printf("first\n");
signal(SIGABRT,SIG—DFL);
printf("second\n");
raise(SIGABRT);
printf("End. OK! \n"); /* 此句未被执行 */
}
/* 程序输出: first
 second
```

注意:在 DOS 下执行此程序时不支持重定向输出 \*/

```
C>TYPE SIGNAL2.C
```

```
#define ——SIGNAL—H /* 此句应在 #include "signal.h" 句之前 */
#include "stdio.h"
#include "signal.h"
```

```
typedef void (* sigfun)(int subcode);
```

/\* 定义接受整形参数的函数指针 sigfun \*/

```
int signal(int sig,sigfun fname);
```

```
void —sigs—1(int kk)
```

```
{
printf("kk= %d\n",kk);
}
```

```
void —sigs—2(int yy)
```

```
{
printf("yy * yy=\n",yy * yy);
}
```

```
main()
```

```
{
```

```
int x=88;
```

```
printf("ss\n");
```

```
x=signal(22,—sigs—1);
```

```
printf("x= %d\n",x);
```

```
signal(23,—sigs—2); /* 不能将此句写成 signal(22,—sigs—2);,那样会使同一 信号对应两个动作函数,虽然编译能通过,但由于结果无所适从,程序执行后便不能得到正确的结果 */
```

```
raise(22);
```

```
}
```

```
/* 程序输出:ss
```

```

x=0
kk=22 */

```

```

—2 int —Cdecl raise(int sig);

```

该函数允许程序在执行过程中给自己发一个信号,以便处理机执行该信号所规定的动作。因此,在它之前应先调用 signal() 定义信号所对应的动作函数。

C>TYPE SIGNAL3.C

```

#include "math.h"
#include "stdio.h"
#include "setjmp.h"
#include "conio.h"
#include "signal.h"
#include "errno.h"
#define PP printf("
#define PL gotoxy(1,9);PP;gotoxy(1,11)
jmp—buf jumper; /* 它是一个记录当前任务的结构,记录有段寄存器 */
 /* cs,ds,es,ss, 寄存器变量 si,di, 指针 sp,ip,bp */
 /* 和标志 flag */
void catcher() /* 用户自定义的动作函数 */
{
gotoxy(1,8);
printf("输入数导致浮点溢出,重输! \n");
errno=EDOM; /* 不加这一句,那末只要执行一次本函数,由于自定义 matherr */
longjmp(jumper,2); /* 的特点,errno 始终等于ERANGE,便不能区分其它错误了 */
} /* 当然,如用缺省的 matherr 则不会这样,它能自动设置 errno */
int matherr(struct exception *e) /* 用户自定义的数学错误处理子程序 */
{
fprintf(stderr,"type=%d,"e->type);
if(errno==ERANGE)
{
/* 设置浮点陷阱信号 SIGFPE 对应动作函数 catcher */
signal(SIGFPE,catcher);
raise(SIGFPE); /* 向正在执行过程发信号 SIGFPE,执行其对应动作 catcher() */
}
gotoxy(1,8);
printf("输入数超出函数定义域,重输! \n");
longjmp(jumper,1);
}
int s=1; /* 用于观察 longjmp() 从何处开始执行 */
main()
{
int flag;
float x,y,f;
clrscr();
printf("试验开始... %d,"s++);
flag=setjmp(jumper); /* 容易看出 longjmp() 从此开始执行 */
printf("flag=%d,s=%d",flag,s++);

```

```

fflush(stdin);
if(flag==0)printf(" 正常执行\n");
else printf(" 执行longjmp()后重新输入\n");
gotoxy(1,4);
printf("输入指数函数pow(x,y)的底x\n");
if(flag!=0)PP,gotoxy(1,5); /* 清空行并移动光标 */
scanf("%f",&x);
printf("输入指数y\n");
if(flag!=0)PP,gotoxy(1,7);
scanf("%f",&y);
f=pow(x,y); /* 计算 x 的 y 次方 */
if(flag!=0)PP,printf("\n");PP,printf("\n");PP;
gotoxy(1,9);
printf("有效结果为:%e, s=%d\n",f,s);
}
/* 可输入 2e345,2e345 或 -2,8.7 这样一些不正确的数对以观察其输入结果 */

typedef int sig_atomic_t; /* 自动存在类型变量 (对 ANSI) */

```

## 38.2 关系 signal() 和 gsignal() 函数的转换

注意:Turbo C 2.0 的库和 SIGNAL.H 文件不再支持函数 signal() 和 gsignal()。这两个函数曾在 Turbo C 1.5 版中使用。它们是从 UNIX 系统中借鉴过来的,ANSI 标准不支持它们,目前的 UNIX 系统 V 也不支持它们。

使用下面定义的两个宏可以实现它们同等功能。注意:这里的 SIG\_IGN 和 SIG\_DFL 两个常量已与 Turbo C 1.5 版中不同。全局表的入口 sigTable[0] 是宏 signal 的一个临时变量,允许该宏进行值交换后,还能返回原先的值。

```

int (*sigTable[16]) = /* 定义全局函数指针数组 */
{
 SIG_IGN, SIG_IGN, SIG_IGN, SIG_IGN,
 SIG_IGN, SIG_IGN, SIG_IGN, SIG_IGN,
 SIG_IGN, SIG_IGN, SIG_IGN, SIG_IGN,
 SIG_IGN, SIG_IGN, SIG_IGN, SIG_IGN,
};
#define signal(num, action) \
(\
 ((num) < 1) || ((num) > 15) ? SIG_DFL : \
 (\
 (sigTable[0] = sigTable[(num)]), /* 作用列表 */ \
 sigTable[(num)] = (action), /* 最后一个表达式 */ \
 sigTable[0] /* 是返回值 */ \
) \
) \

#define gsignal(num) \

```

```

(\
(((num) < 1) || ((num) > 15)) ? 0 : \
(\
 (—sigTable[(num)] == SIG—IGN) ? 1 : \
 (\
 (—sigTable[(num)] == SIG—DFL) ? 0 : (*—sigTable[(num)]()) \
) \
) \
) \
) \

```

## 第三十九章 如何用 C 语言访问扩页内存

扩页内存规范 EMS(Expanded Memory Specification) 是 PC 处理器寻址范围之外的物理存储器。通常,将扩页内存板(expanded-memory board)加插到 PC 适配器槽上,用上位存储器中的一段地址作为扩页内存的页面帧,需要时再把扩页内存的一部分映射到页面帧中。与扩展内存规范 XMS(Extended Memory Specification)不同,扩展存储器只能由 80286 和 80386 处理器使用,而扩页存储器可供各种处理器使用。不过,对扩页存储器只能将数据存放在其中,而不能存放程序代码。它可以用作 Turbo C 的编辑缓冲区。

8086/8088 处理器能对 1024K 的常规内存编址,而对 MS-DOS 只有 640K 对操作系统和用户程序有效,高于 640K 的内存保留给适配卡与系统 BIOS。另一方面,应用程序会碰到两个问题:静态数据被限制在 64KB 以内,这是 DOS 采用段式管理导致的;另一个问题是对数据量很大的程序,内存中动态数据堆已不能满足要求。XMS 或 EMS 便是用来解决这类问题的。

图 39-1 ~ 图 39-4 说明了 DOS 三种存储器(系统存储器、扩展存储器和扩页存储器)的分布情况。

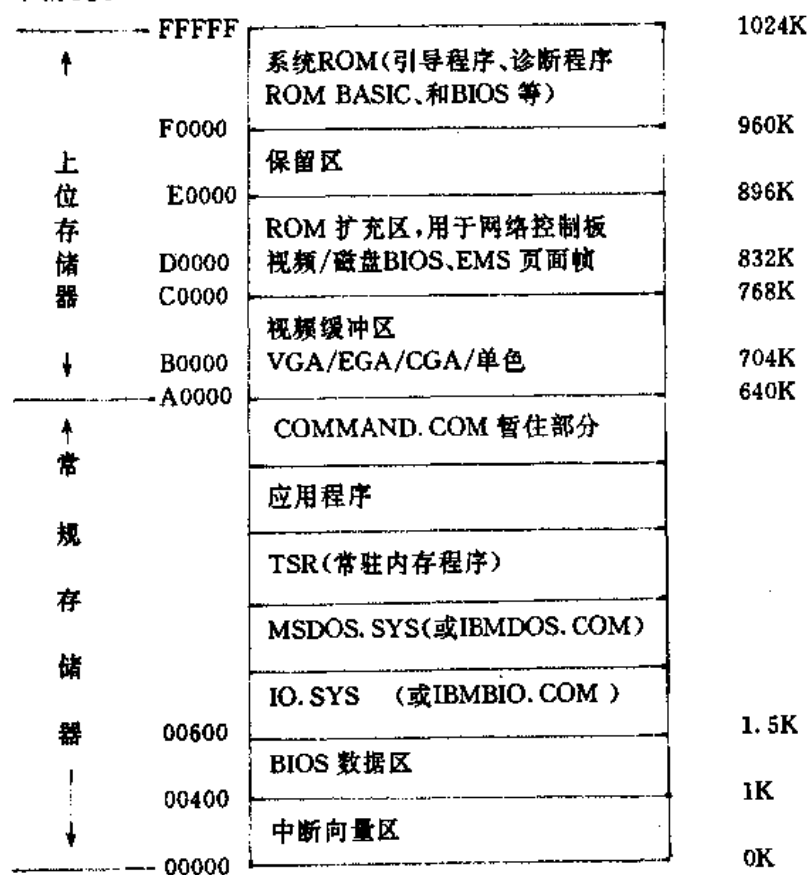


图 39-1 PC 机 DOS 环境使用的系统存储器

注:本节部分内容摘编于周斌、吴立德的《如何在 C 语言应用程序中访问扩充内存》和汤玮的《PC 机存储器结构及使用》。

常规存储器 (Conventional Memory) 也称低端内存、基本内存、基本 RAM、640k 或自由内存等;上位存储器 (Upper Memory) 也称高端内存、BIOS 内存、基本 ROM、384K 或适配器内存等。

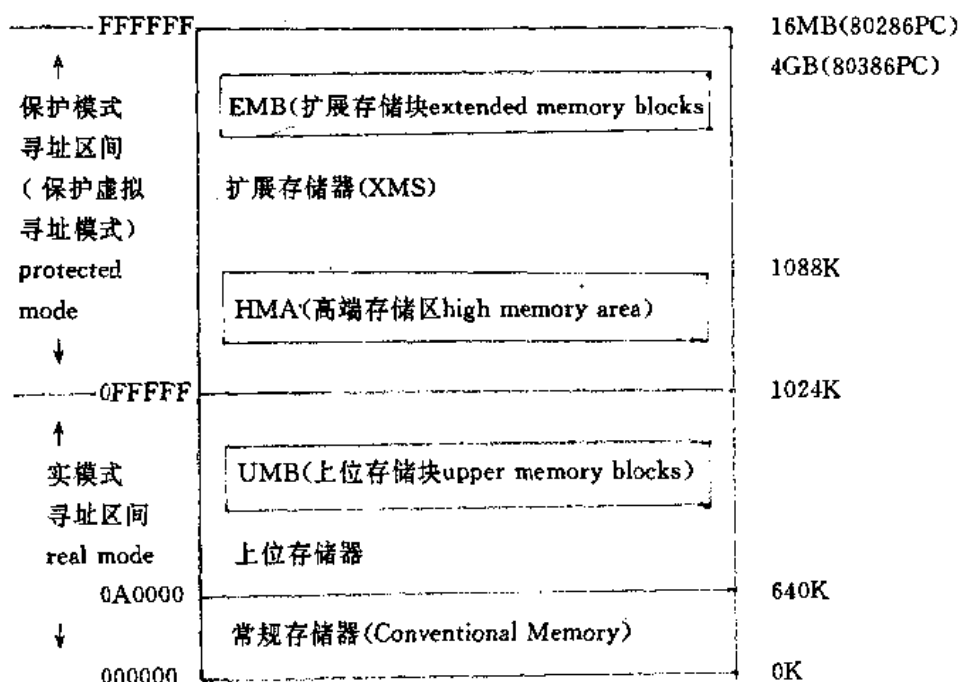


图 39-2 扩展存储器 XMS

HMA 是扩展存储器中紧挨 DOS 1MB 边界的第一个 64KB 区段 (0FFFFFFH ~ 10FFFFFFH) 只能在 80286 或 80386 机上使用。它必须作为一个单独的存储块处理,即不能分割共享,只能调入单独的程序。所以, HMA 一般尽量用来存放接近 64KB 的程序。

UMB 是上位存储器中一些未用的存储地址,在实模式下访问。而 EMB (从 10FFFFH 以上的扩展存储器中分配) 只能在保护模式下访问。如 MS DOS 5.0 用 HIMEM.SYS 管理系统或应用程序对扩展内存的访问 (在此之前应用 RAMDRIVE.SYS 创建 RAM 盘)。各种驱动程序之间可能互不兼容。

保护模式寻址和实模式寻址的根本区别在于两者的寻址地址位数不同。实模式寻址空间为 1MB,故它用了 20 位地址;而保护模式的寻址能力为 16MB 以上,它需要 24 位地址。

在实模式下,段寄存器只要用 16 位,它的值左移 4 位便可得到一个 20 位的物理地址,直接送地址寄存器去驱动地址总线。另一方面,其段偏移量也为 16 位,故每个内存段最大为 64KB。因此程序中可以设置不同的段寄存器值和偏移量,访问任意的 1MB 的内存单元。此外,段又是以节 (16B) 为单位,四个内存段 (代码段 CS、数据段 DS、堆栈段 SS 和附加段 ES) 也可以相互重迭。通过改变段地址 (只要把段地址加上 010000H) 便可访问任何内存单元。因此,在实模式下,程序不仅可以修改其自身,还可以修改其它程序或系统数据。所以,实地址是没有任何保护的。

在保护模式下,段基地址不再存放在 16 位的段寄存器中,而是由原来的段寄存器指向一个包含 24 位基地址的描述符,根据这 24 位基地址由硬件自动形成一个 24 位的物理地址:

物理地址 = 24 位基地址 + 偏移量

对应每个内存段都有一个存放段基址的段描述符,它占 8 个字节。第 0~1 两个字节指出寻址范围;第 2~4 字节存放 24 位基地址;第 3 字节存放高 8 位,第 2 字节 存低 8 位;第 5 字节是权限,指出段是否被访问过,数据段是向高位或低位增长,数据段 能否写入,代码段 能否读,是系统控制段还是代码或数据段,段特权属性和扫描内容是否有效等;第 6~7 字节保留。

保护模式规定寄存器不可修改,代码段不可再写入,数据段和代码段不可重迭存放。这些与实模式是完全不同的,这也是为什么在保护模式下不能运行在实模式下编写的程序的原因之一。

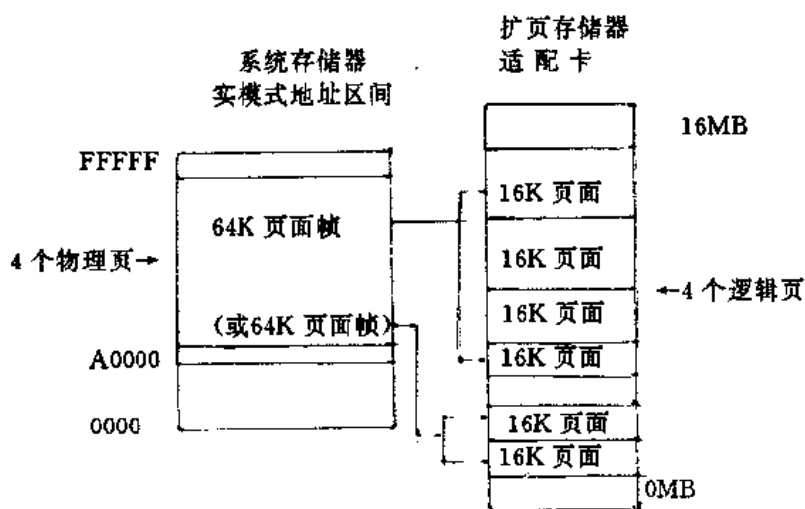


图 39-3 在 8086 或 80286 PC 上使用扩页存储器示意图

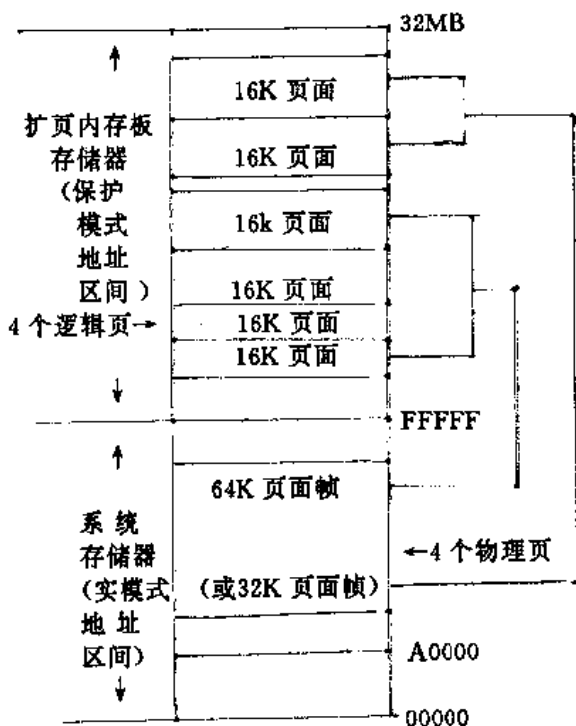


图 39-4 在 80386 PC 上使用扩页存储器示意图

在 80386PC 机上使用扩页存储器需要安装扩展内存板和内存管理程序 (如 MS DOS 5.0 的 EMM386.EXE), 用来模拟扩页存储器, 致使那些使用扩页存储器的程序能运行。

在控制 EMS 方面,须用扩页内存管理程序 EMM(它是 Lotus/Intel/Microsoft 的 Expanded Memory Manager),它独立于硬件,即它使用户使用扩页内存时不用考虑硬件特性。EMM 已有三个标准:EMS 3.2、增强 EMS 3.2 和 EMS 4.0。它们规定了硬件(内存适配板)和软件(内存驱动程序)的条件,可以在 PC 正常寻址范围之外访问多达 32KB 的扩页内存。一般 EMM 是一个设备驱动程序,也是内存驻留程序。为使用这个程序必须在 CONFIG.SYS 文件中有象

DEVICE=C:\EMS40.SYS n

那样的语句。n 表示要求扩页的内存容量,以字节为单位。

说明:EMS 4.0 版的页框可大于 64K,也可把扩页内存映射到内存区域低于 640K 的部分。此外,不同设备也许对此会有不同。

对不同系统可能有不同的要求。例如对 MOS DOS 5.0,它使 DOS 在扩展内存中模拟扩页内存:

(1) 当要使用扩页内存时用 device=c:\dos\emm386.exe 1024-ram

(2) 当不用扩页内存时用 device=c:\dos\emm386.exe noems

对 SUPPER 286 机,除了在 CONFIG.SYS 中加入 device=ems.sys 外,还应事先在启动机器时按 Del 键选 ADVANCED NEAT CHIPSET REGISTER SETUP,再移动光标到 6BH,bit4 处,对

EMS Enable Bit

1= EMS Enable

2= EMS Disable

选1,使能用EMS;再移动光标到6FH, bit5, 6, 7 处,然后选择 EMS 容量大小(Set EMSMemory Size),移到 6DH, bit4, 5, 6, 7 处选择 EMS 基地址(Expanded Memory Base Address),移到 6EH, bit6, 7 处选择 EMS 页 0 的位置(EMS page 0 position),移到 6DH, bit0, 1, 2, 3 处选择 EMS 页面寄存器的 I/O 地址(EMS Reg I/O Base Address),由此完成对 EMS 的设置。

对 Compag 机可能要用 device=CEMM.EXE, 或 device=CEMM.EXE ram, 或 device=CEMM.EXE noems。

所有 EMM 服务通过中断 INT 67H 的功能访问的,因此,如果没有装入 EMM,则调用中断 INT 67 可能会产生不可预料的结果。注意,有些汉字系统修改了中断INT 67H的功能,致使它在汉字系统下不能很好使用。

从 640K ~ 1M 内存中可分配一个 64K 用作映射区域,称【页框】(page-frame, 物理页面框)。所有涉及页框中内存地址的程序指令都被扩页内存板的硬件所修改,使它们的访问对象指向与页框相对应的内存区域。应用程序只能通过页框获得信息。

事实上,页框可分为四个 16K, 每个 16K 称为一个【物理页面】(Physical page): 分别记为物理 0 页、1 页、2 页和 3 页。它们在高端内存中占用连续的地址,因此,既可把 64K 作为整体访问,也可按页访问。

相对地,也可把扩页内存按页(每页也是 16K)划分,每页称为一个【逻辑页面】,被分配的逻辑页面以 0 开始顺序编号。4 个连续的逻辑页面可称为一个【逻辑段】。逻辑页面由【页柄】(handle, 也可称为句柄,它相当于文件句柄)管理。一个页柄可管理多个逻辑页面(大于 4 个,小于 256 个),但要访问页柄管理的逻辑页面,就必须先把它们映射到物理页面。因



为物理页面为 4 个,所以每次最多只能将一个逻辑段映射到 页框的 4 个物理页。

```

/* 以下为一个用 Turbo C 编写的程序,从这个程序可进一步理解扩页存储器 */
/* 在 Turbo C 2.0 中编译应选择 Compact、Large 或 Huge 三种模式之一进行 */
/* 这三个模式都是大数据模式。注意:至少要有 640K 的扩页内存 */
#include "conio.h"
#include "dos.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
static union REGS r;
void ExmemError(int AH—Value)
{
 /* 每次调用 INT 67H 中断,都要检查是否成功地调用了 EMM */
 printf("\n EMS error: %xH",AH—Value);exit(1); /* AH—Value 为错误码 */
}
/*

```

表 39—1 EMS 错误代码表

| 错误码                                        | 含 义                           |
|--------------------------------------------|-------------------------------|
| 00H                                        | 调用成功                          |
| 80H                                        | EMM 软件有错误                     |
| 81H                                        | EMS 硬件故障                      |
| 82H                                        | 内存管理程序忙                       |
| 83H                                        | 无效句柄                          |
| 84H                                        | 无定义的功能请求                      |
| 85H                                        | 无 EMM 空闲句柄                    |
| 86H                                        | 保留和恢复映射状态时出错                  |
| 87H                                        | 请求分配的逻辑页数多于可用物理页数,结果无页可分配     |
| 88H                                        | 同 87H                         |
| 89H                                        | 对小于 4.0 版本,企图分配 0 页。页已用完,不能分配 |
| 8AH                                        | 逻辑页超出了分配给程序的范围                |
| 8BH                                        | 非法的物理页号 (应为 0~3)              |
| 8CH                                        | 保留区域已无空间用于存放                  |
| 8DH                                        | 保留映射状态失败,保留区中已经包含着与该指针有关的内容   |
| 8EH                                        | 恢复映射状态失败,保留区中无给定指针的内容 (原先未保留) |
| 8FH                                        | 子功能参数没有定义                     |
| */                                         |                               |
| unsigned long —02—SegAddrOfPageFrame(void) |                               |
| {                                          | /* 返回 EMM 指定的页框的段地址 */        |
| r.h.ah=0x41;                               |                               |

```

int86(0x67,&r,&r);
if(r.h.ah != 0x00)ExmemError(r.h.ah); /* AH= 错误代码 */
else return(r.x.bx); /* BX= 页框段地址 */
}

char huge * —021—PhysAddrOfPageFrame(void)
{
 /* 把页框的段地址转为其起始单元的实际地址 */
 unsigned long int Ptr;
 Ptr=—02—SegAddrOfPageFrame()* 0x10000;
 return ((char huge *)Ptr); /* 返回规格化指针 */
}

void —04—GetHandleAndAllocMem(unsigned int Num,unsigned * Handle)
{
 /* 向 EMM 申请标号为 Handle 的页柄,并为该页柄分配 Num 个逻辑页面的扩页内存 */
 /*
 r.h.ah=0x43;
 r.x.bx=Num; /* 要分配的逻辑页面数 */
 int86(0x67,&r,&r);
 if(r.h.ah != 0x00)ExmemError(r.h.ah);
 * Handle=r.x.dx;
 */
}

void —05—PageMapping(unsigned int PhysicalPageNum,
 unsigned int LogicPageNum,
 unsigned int Handle)
{
 /* 把 Handle 中的 LogiPageNum 映射到页框中第 PhysicalPageNum 物理页上 */
 r.h.ah=0x44;
 r.h.al=PhysicalPageNum; /* 物理页面号 (0~3) */
 r.x.bx=LogicPageNum; /* 逻辑页面数 */
 r.x.dx=Handle; /* 页柄 */
 int86(0x67,&r,&r);
 if(r.h.ah != 0x00)ExmemError(r.h.ah);
}

void —051—LogicSegMapping(unsigned int LogicSegNum,unsigned int Handle)
{
 /* 把 4 个逻辑页分别映射到 4 个物理页上 */
 unsigned int Num;
 for(Num=0;Num<=3;Num++)—05—PageMapping(Num,LogicSegNum* 4+Num,Handle);
}

void —06—ReleaseHandle(unsigned int Handle)
{
 /* 释放标号为 Handle 的页柄,并把原来属于它管理的所有逻辑页面作为空闲的
 扩页内存归还给 EMM */
 r.h.ah=0x45;
 r.x.dx=Handle;
 int86(0x67,&r,&r);
 if(r.h.ah != 0x00)ExmemError(r.h.ah);
}

unsigned int —13—PageNumOfHandle(unsigned int Handle)
{
 /* 返回页柄 Handle 拥有的页面数 */

```

```

r.h.ah=0x4c;
r.x.dx=Handle;
int86(0x67,&r,&r);
if(r.h.ah != 0x00)ExmemError(r.h.ah);
return(r.x.bx); /* BX=逻辑页面数 */
}
typedef struct proj{
 char ProjCode[8];
 char Charger[18];
 unsigned int Sum;
 unsigned int Members;
 unsigned int During;
}PROJECT; /* 类型变量共要占 32 字节 */
char huge * AddrMap(unsigned long Index,
 unsigned int LenOfPtr,
 unsigned int Handle)
{
 /* 此函数为寻址函数,把索引为 Index 的元素所在之逻辑页面映射到物理 0 页上,并取得该
 元素在页框的实际地址 */
 unsigned int ActiveLogicPage,offset;
 char huge * ptr;
 ActiveLogicPage=(Index * LenOfPtr)/(16 * 1024);
 /* 计算索引为 Index 的元素所在的逻辑页 */
 —05—PageMapping(0, ActiveLogicPage, Handle);
 /* 把该逻辑页映射到物理 0 页上 */
 offset=(Index * LenOfPtr)%(16 * 1024);
 /* 计算该元素在逻辑页内的位移 */
 ptr=—021—PhysAddrOfPageFrame() + offset;
 /* ptr 指向该元素在页框内的实际地址 */
 return(ptr); /* 返回指针值 */
}
—00—Installed(void) /* 这是 Lotus/Intel/Microsoft EMS 中推荐的一种测 */
{
 /* 试 EMS 是否安装的方法。如未安装返回 0,否则非 0 */
 static char * name="EMM"; /* EMM 是该驱动程序的设备名的前三个字母 */
 char far * ptr;
 int k;
 r.h.ah=0x35;
 r.h.al=0x67;
 int86(0x21,&r,&r); /* 获得中断 INT 67H 的当前段值,即驱动程序所在处 */
 ptr=MK—FP(—ES,0x0a); /* 对设备驱动程序,段偏移0xA 开始处是设备名字段 */
 for(k=0;k<3;k++)
 if(* name++ != * ptr++)return (0);
 return (1);
}
—00—Version(void) /* 如已确信装了 EMS,可用它检查程序版本号 */
{
 r.h.ah=0x46;

```

```

int86(0x67,&r,&r);
if(r.h.ah != 0x00)ExmemError(r.h.ah);
return (r.h.al); /* 返回 EMM 版本号,高(低)4位=主(副)版本号 */
}
—00—PageAvail(void) /* 确定有多少可用页可供分配 */
{
 /* 注意:如返回 0 值表示所有 EMS 已被分配 */
 r.h.ah=0x42;
 int86(0x67,&r,&r);
 if(r.h.ah != 0x00)ExmemError(r.h.ah);
 return (r.x.bx); /* 返回当前可用来分配的扩页内存的 16K 字节逻辑页数 */
}

/* 以上是 EMM 几个基本功能的实现函数。其它与 INT 67 有关的几个功 */
/* 能为:r.h.ah=0x40 若 r.h.ah 返回 0,表明内存扩页卡已安装 */
/* r.h.ah=0x47;r.x.dx=Handle; 存储(或保留)当前页映射寄存器值 */
/* 当一个暂驻程序正在使用扩页内存时被突然中断,控制转向另一个中断 */
/* 处理程序。如该程序也要访问扩页内存,那就要保留原暂驻程序运行时 */
/* 的映射状态,以便能正常返回。存储前一定要保证已将逻辑页映射到物 */
/* 理页(功能 0x44) */
/* r.h.ah=0x48;r.x.dx=Handle; 恢复被保留的页映射寄存器值,即物理 */
/* 页面与逻辑页面之间联系的内容 */
/* 注意:在释放扩页内存前,一定要先恢复被保留的页映射寄存器值, */
/* 如果先前已将页寄存器的值存储过。出错时 r.h.ah 中一般返回错误码 */

main()
{
 PROJECT huge *ptr;
 unsigned int HandleArr;
 unsigned int NumOfPages,LogicSegCount,count,MaxIndexOfSeg;
 unsigned long Index;
 FILE *FileName;
 NumOfPages=40; /* 要申请的页数 */
 —04—GetHandleAndAllocMem(NumOfPages,&HandleArr);
 /* 向 EMM 申请一个句柄 HandleArr,并为它分配 NumOfPages 个逻辑页。 */
 /* 注意:一个程序中只有将原先分配的所有页释放后才能进行二次分配 */
 for(LogicSegCount=0; LogicSegCount<NumOfPages/4; LogicSegCount++)
 {
 /* 循环是对数组初始化 */
 —051—LogicSegMapping(LogicSegCount, HandleArr);
 /* 每次把一个逻辑段映射到 4 个物理页 */
 ptr=(PROJECT *)—021—PhysAddrOfPageFrame(); /* 取页框首地址 */
 MaxIndexOfSeg=1024/sizeof(PROJECT)*64;
 /* 每个逻辑段(或者说页框)所能存放的元素个数 */
 for(count=0;count<MaxIndexOfSeg;count++)
 {
 strcpy(ptr->ProjCode,"");
 strcpy(ptr->Charger,"");
 ptr->Sum=0;
 ptr->Members=0;
 }
 }
}

```

```

 ptr->During=0;
 ptr++; /* 以地址指针为数组索引,指向下一个元素 */
 }
}

Index=10240; /* 以下为对数组元素 10240 的随机访问 */
ptr=(PROJECT *)AddrMap(Index, sizeof(PROJECT), HandleArr);
 /* 把索引为 Index 的元素所在页映射到物理 0 页 */
 /* 上,并取得该元素在页框的实际地址,然后赋值 */
strcpy(ptr->ProjCode,"NO052493");
strcpy(ptr->Charger,"ZhuQi");
ptr->Sum=800;
ptr->Members=20;
ptr->During=2;
FileName=fopen("PROJ.DAT","wt"); /* 把页柄号放在文件 PROJ.DAT 中 */
fprintf(FileName,"%u\n",HandleArr);
fclose(FileName); /* 注意:关闭文件但没有释放页柄,故页柄中的 */
 /* 数据在内存中并未丢失,必要时可再访问它 */

/* 从 PROJ.DAT 中恢复页柄后重新访问内存中数据:用本 main 取代前 main */
main()
{
 PROJECT *ptr;
 unsigned int HandleArr;
 unsigned long Index,MaxIndex;
 unsigned int TotalPages;
 FILE *FileName;
 FileName=fopen("PROJ.DAT","rt");
 fscanf(FileName,"%u\n",&HandleArr);
 fclose(FileName);

 /* 计算 HandleArr 拥有逻辑页面数 */
 TotalPages=-13-PageNumOfHandle(HandleArr);
 /* 计算页柄容纳的数组元素个数 */
 MaxIndex=1024/sizeof(PROJECT)*16*TotalPages;
 Index=0; /* 以下在数组中顺序寻找编号为 NO052493 的元素 */
 do {
 ptr=(PROJECT *)AddrMap(Index,sizeof(PROJECT),HandleArr);
 Index++;
 }while ((strcmp(ptr->ProjCode,"NO052493")!=0) && (Index<MaxIndex));
 if (Index<MaxIndex)printf("%ld\n",Index);
 else printf("Not found ! \n");
 _06-ReleaseHandle(HandleArr); /* 释放页柄,其占用的所有内存单元也释放 */
 }
 /* 一旦把某些内存分配给页柄,便不能再将 */
 /* 它分配给另外的页柄;当用完该内存后, */
 /* 应及时将它释放,以便供它用 */

```

## 第四十章 命令行编译器TCC.EXE

通常可以利用集成环境完成源程序编辑、编译和连接,最后直接生成可执行程序。但是对像 C 源程序里嵌有汇编语句时或者你想用 Turbo C 去生成汇编语言代码时,就不能使用集成环境,而必须使用 TCC.EXE 命令行编译器。使用 TCC 是在 DOS 提示符下键入一行命令实现的。它的大多数命令选择项都在集成环境 (TC.EXE) 的 Options 菜单下有对应的选择项。为方便查询,下面将列出它们之间的对应表。为简略起见,TC 选项采用从主菜单开始的首字母表示法,只有最后项才列出其内容。例如 O/C/S/ANSI keywords only(On) 表示

Options/Compiler/Source/ANSI keywords only 选On

编译出错信息参见《错误、警告及提示信息》。由于 TCC 编译器生成的执行文件由于没有包括调试信息,所以通常比在集成环境下生成的执行文件要小一些。

在 DOS 提示符下键入如下命令

C>TCC

屏幕显示 TCC.EXE 的帮助信息:

Turbo C Version 2.0 Copyright (c) 1987, 1988 Borland International

Syntax is: TCC [ options ] file[s] \* = default; -x- = turn switch x off

|       |                           |        |                             |
|-------|---------------------------|--------|-----------------------------|
| -l    | 80186/286 Instructions    | -A     | Disable non-ANSI extensions |
| -B    | Compile via assembly      | -C     | Allow nested comments       |
| -Dxxx | Define macro              | -Exxx  | Alternate assembler name    |
| -G    | Generate for speed        | -lxxx  | Include files directory     |
| -K    | Default char is unsigned  | -Lxxx  | Libraries directory         |
| -M    | Generate link map         | -N     | Check stack overflow        |
| -O    | Optimize jumps            | -S     | Produce assembly output     |
| -Uxxx | Undefine macro            | -Z     | Optimize register usage     |
| -a    | Generate word alignment   | -c     | Compile only                |
| -d    | Merge duplicate strings   | -exxx  | Executable file name        |
| -f    | * Floating point emulator | -f87   | 8087 floating point         |
| -gN   | Stop after N warnings     | -iN    | Maximum identifier length N |
| -jN   | Stop after N errors       | -k     | Standard stack frame        |
| -lx   | Pass option x to linker   | -mc    | Compact Model               |
| -mh   | Huge Model                | -ml    | Large Model                 |
| -mm   | Medium Model              | -ms    | * Small Model               |
| -mt   | Tiny Model                | -nxxx  | Output file directory       |
| -oxxx | Object file name          | -p     | Pascal calls                |
| -r    | * Register variables      | -u     | * Underscores on externs    |
| -v    | Source level debugging    | -w     | Enable all warnings         |
| -wxxx | Enable warning xxx        | -w-xxx | Disable warning xxx         |
| -y    | Produce line number info  | -zxxx  | Set segment names           |

注意:在 TCC.EXE 的所在目录里应有 TLINK.EXE 文件,因为 TCC.EXE 在运行时会自动

地装入它并运行。

## 40.1 TCC 命令行书写语法 (Syntax) 规则

TCC [option... ] [file [file... ]

— options ( 命令选项 ) 的填写

. 所有选择项均以连字符 ( — , 即减号 ) 开头;

. —x— = turn switch x off 表示选择项 x 被关掉;

—x = turn switch x on 表示选择项 x 被打开, 它覆盖了在配置文件 ( \*.CFG ) 中已经设置的同名选择项 x。

. \* = default; 英文说明前带星号者表示缺省选择。

. 选择项可分为八类:

1. 存储模式类: 指明编译模式 ( 六种中的一种 )。

—mc —mh —ml —mm —ms —mt

2. 宏定义类: 定义宏为缺省值、数值或字符串, 或解除原宏定义。

—Dxxx —Uxxx

3. 代码生成类: 控制代码生成方式, 如浮点处理、调用约定、字符类型或 CPU 指令。

—l —l— —K —K— —N —a —a— —d —f —f— —f87 —k —p —u —u— —v —y

4. 源代码选择类: 了解源代码特征, 如实现专用 ( 非 ANSI ) 的关键字、嵌套注释或标识符长度。

—G —O —Z —r —r—

5. 优化选择类: 指明目标码的优化方式。

—A —C —iN 6. 错误报告类: 选择警告类型等。

—gN —jN —w —w— —wxxx —w—xxx

7. 段名控制类: 重置命令段等。

—zAxxx —zCxxx —zPxxx —zBxxx —zDxxx —zGxxx —zRxxx —zSxxx —zTxxx  
—zX

8. 编译控制类: 使编译程序生成汇编代码, 而不是目标模块。

—S

81. 编译含有直接插入汇编语句的源代码;

—B

82. 选择一汇编程序;

—Exxx

83. 只编译不连接。

—c

. 编译控制选择项

—B —Exxx —S —c —oxxx

. 连接选择项

—M —exxx —lx

. 环境选择项

—lxxx —Lxxx —nxxx

表 40-1

命令选项      英文说明      集成环境中对应的菜单选择 (从主菜单开始)

- -1 80186/286 Instructions =O/C/Code generation(80186/80286)  
产生扩充的 80186 指令,当在非保护模式 (如 IBM PC AT 的 MS-DOS 3.0)下运行生成的 80286 程序时,也可使用本选择项。
- 1- 8088/8086 Instructions =O/C/Code generation(8088/8086)  
产生 8088/8086 指令。
- -A Disable non-ANSI extensions =O/C/Source/ANSI Keywords only(On) 建立与 ANSI 标准兼容的代码。此时下列 TC 使用的非 ANSI 关键字 (扩充关键字): near, far, huge, cdecl, asm, pascal, interrupt, -cs, -ds, -es, -ss, -AX, -BX, -SI 等将被认为是一般的标识符,而不能当作关键字对待。
- -B Compile via assembly =无  
编译和调用汇编程序 (TASM.EXE) 处理 C 源程序中嵌入的汇编代码,在集成环境 (TC.EXE) 下本选择无效。
- -C Allow nested comments =O/C/Source/Nested comments(On)  
允许源程序中有嵌套注释。
- -Dx xx Define macro =O/C/Defines  
包括两种:一是 -Dxxx,把标识符 xxx 定义为单个空格符;另一是 -Dxxx=YY,把 xxx 定义为字符串 YY,YY 一般可由英文字母与数字等组成,内中不应有空格符。
- -Ex xx Alternate assembler name =无  
代替汇编程序 (TASM.EXE) 的新汇编程序名。如果指定了 -B 选项,TC.EXE 将寻找合适的汇编程序,缺省的汇编程序名为 TASM.EXE。现用 -E 选项,可用别的汇编程序进行编译。句法是  
TCC -EMASM (汇编程序名) -B 用户源程序名
- -G Generate for speed =O/C/O/Optimize for(speed)  
编译首先按时间优化。
- -Ix xx Include files directory =O/D/Include directories  
指定搜索目录名,该目录中含有包含文件。
- -K Default char is unsigned =O/C/C/Default char type(Unsigned)  
说明编译程序处理字符说明为 Unsigned。
- K- Default char is signed =O/C/C/Default char type(signed)  
说明编译程序处理字符说明为 signed。
- -Lx xx Libraries directory =O/D/Library directories  
指定搜索目录名,该目录中含有库文件。
- -M Generate link map =O/L/Map file(On)  
使连接程序产生一个完全连接的映射文件 (.MAP)。缺省时映射文件中不产生序号。
- -N Check stack overflow =O/C/C/Test stack overflow(On)  
在每个函数的入口,产生【堆栈溢出逻辑】:一旦发现堆栈溢出就产生溢出信号。这将使程序变大,运行速度变慢。但在调试程序时非常有用,因为堆栈溢出是很难发现的。堆栈溢出时显示 stack overflow,程序以出口码 -1 退出。
- -O Optimize jumps =O/C/O/Jump optimization(On)  
通过消去多余的转移进行优化,并重新组织循环和 switch 语句。
- -S Produce assembly output =无  
编译指定的 C 源文件并产生汇编语言输出文件 (.ASM),但不进行汇编。它在处理有嵌入汇编的源程序时可以分析嵌入式代码转换后的形式。



■ -Ux xx Undefine macro =无

解除宏原有的定义。

■ -Z Optimize register usage =O/C/O/Register optimization(On)

通过记住寄存器内容并尽可能地使用它们来消去多余的存取操作。注意:使用本选项要小心,因为编译程序无法知道寄存器值是否在某一时刻已被某一指针间接地改变过。

■ -a Generate word alignment =O/C/C/Alignment(Word)

确保整数按机器字(word)边界对齐,多余字节被插入到一个结构中以确定字段对齐的安全性。**【局部变量】**和**【全局变量】**也适当对齐,char和unsigned char型的字符变量和字段可放在任何地址处,即不能偶对齐,而其它变量和地址必定放在偶地址处。字对齐可提高存取数据速度。

-a- Generate byte alignment =O/C/C/Alignment(Byte)

按字节对齐。

■ -c Compile only =C/Compile to OBJ

只编译扩展名为.C或汇编扩展名为.ASM类文件(汇编需TASM.EXE支持),但不执行连接命令。

■ -d Merge duplicate strings =O/C/C/Merge duplicate strings(On)

当两个字符串匹配时将它们合并,以产生较小程序。

■ -ex xx Executable file name =无

在不带扩展名的文件名xxx后加上扩展名.EXE形成可执行文件。注意,文件名与字母e之间应无空格。本选项缺省时,用**【文件名表】**(由命令行上多个源文件或目标文件构成)中的第一个文件名作为这里的xxx。例如,

C>TCC -eMY MYPRO.C

把MYPRO.C编译成MY.EXE。

■ -f \* Floating point emulator =O/C/C/Floating point(Emulation)

如果计算机内装有协处理器,就调用它进行浮点运算;否则进行仿真调用,当然速度要慢一些。

-f- \* Floating point none =O/C/C/Floating point(None)

指明程序不进行浮点运算,连接时也不用浮点库文件。

■ -f87 8087 floating point =O/C/C/Floating point(8087/80287)

用内部8087指令而不是通过8087仿真库函数来进行浮点操作。用此编译的执行程序在运行时机器内应装有8087/80287浮点协处理器。

■ -gN Stop after N warnings =O/C/E/Warnings;stop after (N)

当产生了N个警告信息后停止编译。

■ -iN Maximum identifier length N =O/C/S/Identifier length (N)

使编译程序只识别每一个标识符最前面的N个字符。如N=8(UNIX等系统中规定),则两个标识符SsTtUuVvA与SsTtUuVvB被认为是一样的,因为它们前8个字符相同。TC缺省值为N=32,即最多可以用32个字符来标识变量名、函数名或宏名等。

■ -jN Stop after N errors =O/C/E/Errors;stop after (N)

当产生了N个错误信息后停止编译。

■ -k Standard stack frame =O/C/C/Standard stack frame(On)

产生标准的堆栈结构。在调试程序时用它跟踪被调用子程序的堆栈时很有用。

■ -lx Pass option x to linker =无

传送命令行选项x给连接程序(TLINK.EXE),以便连接时发挥作用。选项-lt相当于告诉TLINK.EXE选用选项/t,即连接后生成COM文件。

C>TYPE F.C

```
#include "alloc.h"
```

```
int (*q)[7][6];
```

```
int i,j,k;
```

```

main()
{
q=(int (*)[7][6])malloc(8*7*6*sizeof(int));
for(i=0;i<8;i++) /* 这是一种较特殊的动态分配方法,多维数组初始化 */
 for(j=0;j<7;j++)
 for(k=0;k<6;k++)
 q[i][j][k]=i*j*k;
for(i=0;i<8;i++)
 for(j=0;j<7;j++)
 {
 for(k=0;k<6;k++)
 {
 printf("%d %d %d %d",i,j,k,q[i][j][k]);
 printf("\n");
 }
 }
}

```

/\* 在集成环境下选用 Tiny 模式编译连接后生成的执行文件 F.EXE 长度为 9887 字节,如在 DOS 提示符下用 C>TCC -mt -lt f.c

便可直接产生长度为 6200 字节的 COM 文件 F.COM。注意:这里选用微模式 -mt 是必须的,否则会产生

cannot generate COM file , stack segment present

而不能生成 COM 文件。另外,执行 TCC.EXE 文件也需要相当的内存,如内存不够,也可能出现

cannot execute C:\TC\TCC.EXE

即不能执行该文件的提示,在某些中西文兼容的系统中使用时是要注意的。如果在程序 F.C 中增加语句

```

#include "math.h"
double x;
x=sqrt(4);
printf("%f\n",x);

```

则不能生成 COM 文件。利用 TCC.EXE 时会产生

cannot generate COM file , segment - relocatable items present

即执行文件中有需要重定位的段而不能产生 COM 文件。如用 DOS 的 EXE2BIN 也会产生

File cannot be converted

的提示,结果也不会有 COM 文件产生。因此,当程序中有浮点运算函数时将不能产生 COM 文件。 \*/

- -mc Compact Model =O/C/Model(Compact)  
紧凑存储模式。
- -mh Huge Model =O/C/Model(Huge)  
巨型存储模式。
- -ml Large Model =O/C/Model(Large)  
大型存储模式。

- **-mm** Medium Model =O/C/Model(Medium)  
中型存储模式。
- **-ms** \* Small Model =O/C/Model(Small)  
小型存储模式。
- **-mt** Tiny Model =O/C/Model(Tiny)  
微型存储模式。此模式产生的代码几乎与小型相同,只是在连接时使用 COT.  
OBJ 产生微型执行程序。
- **-nxxx** Output file directory =O/D/Output directory  
指定输出文件目录名,该目录中含有目标文件、执行文件或映射文件等。
- **-ox** xx Object file name =无  
给源文件指出编译后的目标文件名 (.OBJ)。
- **-p** Pascal calls =O/C/C/Calling convention(Pascal)  
使编译程序产生的所有子程序调用和函数调用均使用 pascal 参数传递序列,这  
使函数调用既快又短。函数调用时,传递参数的个数和类型必须正确,不能像一  
般 C 语言那样允许函数参数个数可变。可用 cdecl 语句来覆盖此选择项,把函 数  
调用明确地说明成 C 参数传递系列。
- **-r** \* Register variables =O/C/O/Use register variables(On)  
允许使用寄存器变量。
- r-** =O/C/O/Use register variables(Off)  
禁止使用寄存器变量。此时编译程序将不使用寄存器变量,也不保存 SI 或 DI 寄  
存器值,这时不能有内含寄存器变量调用的代码。
- **-u** \* Underscores on externs =O/C/C/Generate underbars(On)  
使用本选项后,若定义了一个标识符,TC 便把它保存到目标模块内,并自动在 标  
识符首字符前面加上一个下划线 ( \_ )。对 pascal 型标识符 ( 用 pascal 关 键字来  
修饰 ),TC 将它们大写且不加下划线。
- u-** =O/C/C/Generate underbars(Off)  
不加下划线。这时如使用 TC 的标准库函数会遇到麻烦! 除非你重建这些库。因  
此,一般不要用此选项。
- **-v** Source level debugging =O/C/C/OBJ debug information(On)  
使用本选择项时,编译程序在生成目标代码过程中加入了调试信息,这样可以 使  
用集成环境调试器进行源程序级调试,也可以用 Turbo Debugger 调试器进 行调  
试。
- **-w** Enable all warnings =O/C/E/Display warning(On)  
允许产生全部警告信息。
- w-** Disable all warnings =O/C/E/Display warning(Off)  
不产生任何警告信息。
- **-wx** xx Enable warning xxx =O/C/E/Portability warnings 等(On)  
允许至少产生由 xxx 指出的警告信息,其中 xxx 由三个字母组成,如 xxx=sus  
时表示输出“值得怀疑的指针转换”。
- **-w-x** xx Disable warning xxx =O/C/E/Portability warnings 等(Off)  
不产生由 xxx 指定的警告信息。
- **-y** Produce line number info =O/C/C/Line numbers(On)

在目标文件中放入了供符号调试用的行号信息。这会增大目标文件,但不会影响执行文件的大小和运行速度。

#### ■ -zx xx Set segment names

用于改变段名。具体地说,可分为下列 10 种情况。

- zAxx x  
改变代码段的类(class)名,缺省类名为 -CODE。
- zCxx x  
改变代码段的段名。对 Tiny (微)、Small (小)、Compact (紧凑)三种模式,缺省段名为 -TEXT; 对其它三种模式,缺省段名为 xxx-TEXT, 这里 xxx 为源文件名。
- zPxxx  
使产生带代码组的输出文件的代码段的段名为 xxx。本选项不能在 Tiny (微)模式下使用。
- zBxx x  
改变未初始化数据段的类名为 xxx。缺省类名为 -BSS。
- zDxx x  
改变未初始化数据段的段名为 xxx。缺省段名为 -BSS。注意,在 Huge (巨型)模式中不生成未初始化段。
- zGxx x  
改变未初始化数据段的组名为 xxx。缺省组名为 -DGROUP。注意,在 Huge (巨型)模式中无数据段组。
- zRxx x  
置已初始化数据段的段名为 xxx。缺省时除了在 Huge (巨型)模式中段名为 xxx-DATA 外,其它模式中段名均为 -DATA。
- zSxx x  
改变已初始化数据段的组名为 xxx。缺省时除了在 Huge (巨型)模式中因无数据组故本选项不起作用外,其余模式中数据组名为 -DGROUP。
- zTxx x  
置已初始化数据段的段名为 xxx。缺省时段名为 -DATA。
- zX \*  
字母 X 表示使用缺省值。例如,-zA \* 表示代码段的类名为缺省类名 -CODE。

## 二 任选项的分类

可以分为两大类:

1. -I 或 -L, 暂称 IL 类。
2. 除 -I 和 -L 外的所有选项, 暂称非 IL 类。

## 三 任选项执行规则

1. 对 IL 类选项,左边的选项优先于右边的选项。或者说,当同一选项在命令行上多次出现时,最左边的起作用,其它的被忽略。
2. 对非 IL 类选项,右边的选项优于左边的选项。

#### 四 file (文件名) 的填写

文件名可以是:

|              |                              |
|--------------|------------------------------|
| filename.asm | 它用 TASM 汇编成 .OBJ 文件          |
| filename.obj | 连接时用到的目标文件                   |
| filename.lib | 连接时用到的库文件                    |
| filename     | 编译 filename.C (扩展名为 .C 可以不写) |
| filename.c   | 编译 filename.C                |
| filename.xyz | 编译 filename.XYZ              |

注意: TCC.EXE 会自动地装入它所需要的标准库, 如 Cx.lib 等, 所以文件名可以不包括它们。它不会自动搜索图形库 graphics.lib。

#### 五 可执行文件名的产生

可执行文件是第一个文件基本名加上扩展名 .EXE 构成。如果想另定义一个可执行文件名, 则在命令行上 TCC 和文件名之间加上选项 -e 和执行文件名 (-e 和执行文件名之间无空格)。

### 40.2 命令行配置文件 TURBOC.CFG

除了在命令行上设置命令行选项外, 还可以用 TURBOC.CFG 配置文件设置一系列任选项。凡是在命令行上能出现的任选项均可以在该文件中出现。

如果已将一些常用的选项设置到该文件中, 那么就不必在 TCC 命令行上再键入这些任选项了; 反过来, 如不想使用该文件中某一选项, 那么你可以在 TCC 命令行上再设置此选项, 就可对该选项进行抑制。换句话说, 命令行上设置的选项是最终起作用的选项, 它优于 TURBOC.CFG 中的选项。

在运行 TCC.EXE 时, 它首先在当前目录里查找 TURBOC.CFG; 如果未找到, 则到 TCC.EXE 所在的目录中去找。注意, 虽然 TURBOC.CFG 也可以用别的名保存, 但 TCC.EXE 只认 TURBOC.CFG, 其它的名字不予理睬。

可用字处理器 (例如 TC 编辑器或行编辑器 EDLIN.EXE 等) 创建 TURBOC.CFG。可以将任一选项列在同一行上 (相邻两个选项之间用空格隔开), 也可以列在不同行上。这样, 在用 TCC.EXE 编译程序时, 不但可以在命令行上使用选项, 还会使用 TURBOC.CFG 中的选项。虽然对一些常用的选项可放在 TURBOC.CFG 中。

注意: 不要将它与 TC 缺省配置文件 TCCONFIG.TC 混淆。

### 40.3 配置文件转换实用程序 TCCONFIG.EXE

该文件可以把命令行配置文件 TURBOC.CFG 与配置文件 \*.TC (如 TCCONFIG.TC) 互相转换, 其一般形式为

C>TCCONFIG 源文件 目标文件

例 1 将 TURBOC.CFG 转换为 MY.TC

C>TCCONFIG TURBOC.CFG MY.TC

例2 将 TURBOC.CFG 转换为 TCCONFIG.TC

```
C>TCCONFIG TURBOC.CFG
```

例3 将 MY.TC 转换为 TURBOC.CFG

```
C>TCCONFIG MY.TC
```

创建 TCCONFIG.TC 时, TCCONFIG.EXE 对于命令行配置文件没有说明的条目将使用缺省值。反过来,它只将 TCCONFIG.TC 中与缺省值不同的选项包括到 TURBOC.CFG 中。TCCONFIG.EXE 做完转换工作后将返回到 DOS 提示符下。

## 40.4 应用举例

例1

输入下面的命令行

```
C>TCC -a -f -O -C -Z -eMYEXE oldfile1 oldfile2 nextfile.c
```

TCC 将对 oldfile1.c、oldfile2.c 和 nextfile.c 三个文件进行编译,先产生 MYEXE.OBJ 目标文件,最后连接生成 MYEXE.EXE 可执行文件。在此过程中,采取字对齐(-a)、浮点仿真(-f)、允许跳转优化(-O)和注释嵌套(-C),并通过寄存器优化处理(-Z)。

注意:如要生成可执行文件,当前目录中应有 TLINK.EXE 文件;对命令行上有 .ASM 文件或 .C 文件中插入有汇编语句时,当前目录里应有 TASM.EXE 文件(或用 -E 选项指定的其它汇编编译程序,如 MASM.EXE 等)。

例2

```
C>TCC -IB:\include -LB:\lib -eMY start.c S2.obj end
```

包含文目录是 B:\include,库所在的目录是 B:\LIB,生成的执行文件是 MY.EXE,被编译的文件是 start.c 和 end.c,被连接的文件是 start.obj、S2.obj 和 end.obj。

例3

```
C>TCC -mm -C -K S1 S2.C y.asm mylib.lib
```

包含文件和库文件目录用缺省值,使用中存储模式,允许源程序中有嵌套注释,使用的 char 无符号,编译源文件 S1.C 和 S2.C,汇编 Y.ASM,连接库文件名 mylib.lib,生成可执行文件名为 S1.EXE。

## 40.5 TCC.EXE 与 TASM.EXE (Turbo assembly) 的关系

TCC 可以利用 TASM 的开关 /D—mdl—

其中 mdl 是六种模式:TINY、SMALL、MEDIUM、COMPACT、LARGE 和 HUGE。还可  
用开关

/mx

它告诉 TASM 区分大小写标识符。

## 第四十一章 独立连接程序 TLINK.EXE

TC 的命令行版本使用一个独立的连接程序 TLINK.EXE, 又称【Turbo C 连接器】, 它将目标模块与库文件连接起来生成可执行文件。它解决了访问多个目标模块时涉及的函数调用、全程变量和存储地址等问题。

集成环境 (TC.EXE) 中有一个内部连接程序, 因此它不用 TLINK.EXE。或者说, 与之相当的功能是由 TC.EXE 自动完成的, 无须程序员考虑。使用命令行编译器 (TCC.EXE) 编译程序时, 编译器会自动装入 TLINK.EXE, 并用它连接编译成可执行文件。对此程序员可能感觉不到它的存在。然而 TLINK.EXE 也可单独使用。

### 41.1 使用 TLINK.EXE 的一般语法

在 DOS 提示符下键入 (不带任何参数) C>TLINK 后回车, 屏幕显示

Syntax: TLINK objfiles, exefile, mapfile, libfiles

@xxxx indicates use response file 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 = lower case significant in symbols

/3 = enable 32-bit processing

/v = include full symbolic debug information

/e = ignore Extended Dictionary

/t = create COM file

这是 TLINK.EXE 连接程序时的一般格式 (下面还将介绍其用于连接 Turbo C 程序的方法, 两者之间稍有不同), 下面解释这些参数和选择项 (为便于区别原文和附加的汉字说明, 下面将用头部标有■的英文行表示原文), 它告诉你怎样使用 TLINK。

■Syntax: TLINK objfiles, exefile, mapfile, libfiles

语法: TLINK □ 目标文件 (.OBJ), 执行文件 (.EXE), 映射文件 (.MAP), 库文件 (.LIB)

1. 此语法规则用于连接非 TC 的相关程序。
2. TLINK 后面由四部分组成, 它们的顺序不能颠倒。
3. 扩展名省略时, TLINK 自动加上相应的缺省扩展名 (.OBJ, .EXE, .MAP, .LIB)。
4. 文件名可以带路径名。
5. 逗号是命令行的一部分, 用于区分各类型文件。
6. 目标文件 (objfiles) 为想要连接的全部 .OBJ 文件, 即可以有多个, 文件之间用空格

符 (□) 隔开。第一个目标文件必须是和存储模式相关的初始化模块名。

7. 执行文件名 (exefile) 未给出时, TLINK 生成的执行文件名除扩展名为 .EXE 不同外, 其余和第一个 OBJ 文件相同。

8. MAP 文件名 (mapfile) 未给出时, TLINK 生成的 MAP 文件名除扩展名为 .MAP 外, 其余同执行文件名。

9. 库文件 (libfiles) 为用到的各种库, 可以没有。如果使用了库文件, 则必须用空格符分隔多个库。

■ @xxxx indicates use response file xxxx

@xxxx 表示一个响应文件名。【响应文件】是一种文本文件, 它包括命令行中 TLINK 后的各选择项及文件名等。响应文件可以有连续几行, 只要在行尾加上一个加号 (+) 就可续行。各连接文件 (objfiles、exefile、mapfile 和 libfiles) 也可以不在同一行上。如果在续行处有逗号, 则在书写的下一行上应将分隔用的逗号去掉, 即不写。

下面用例说明怎样使用响应文件。

不用响应文件时

```
C>TLINK /c mainline wd ln tx,fin,mfin,lib\comm lib\support
```

连接时, 区分字母大小写 (/c), 待连接文件为 mainline.obj、wd.obj、ln.obj 和 tx.obj, 可执行程序名为 fin.exe, 连接的库文件为子目录 lib 里的 comm.lib 和 support.lib。为使用响应文件, 先写一个响应文件 MYRES:

```
C>TYPE MYRES
/c mainline wd +
ln tx, fin +
mfin +
lib\comm lib\support
```

现在只要打入命令行 C>TLINK @MYRES 就可完成连接。当然, 也可将一个响应文件分成几个响应文件来写, 或者将命令行的一部分用响应文件。如上例, 先编写两个响应文件:

```
C>TYPE MYRES1
mainline +
wd +
ln tx

C>TYPE MYRES2
lib\comm +
lib\support
```

则可打入以下命令完成连接:

```
C>TLINK /c @MYRES1, fin, mfin, @MYRES2
```

以下选项 (Options) 可以放在执行文件名 TLINK 之后, 有多个选项时它们之间位置可任意。

■ /m        = map file with publics

用此选项, 生成一个更完备的 MAP 文件, 它是可执行文件的映射, 所以称映射文件。缺省映射文件只包括程序中段的表、程序起始地址、连接期间生成的警告或错误消息, 以及公共变量 (或称公用符号表, 将按地址递增次序排序)。MAP 文件在程序调试时有时很有参考



价值。

■ /x = no map file at all

用此选项后,不生成任何 MAP 文件,即使在命令行上指定了 MAP 文件名时也如此。否则总要生成 MAP 文件。

■ /i = initialize all segments

初始化全部段。

■ /l = include source line numbers

将源程序行号放在可执行文件中,以便调试。为了使用它,在创建 \*.OBJ 文件时,即在使用命令行编译程序 TCC.EXE 时必须加上选择项 -y(表明在目标文件中放入了供符号调试用的行号信息)。

■ /s = detailed map of segments

生成的 MAP 文件有详细的分段映像,即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、类名、段名、组名、模块和 ACBP 信息等。它包括 /m 选项的内容。

■ /n = no default libraries

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,本选择项是必须的,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。

■ /d = warn if duplicate symbols in libraries

在库中发现重复符号时显示警告信息,列出在(多个)库中所有的重复符号,即使在程序中并没有用到这些符号时也这样。

■ /c = lower case significant in symbols

连接时对 PUBLIC(公共的)和 EXTERN(外部的)标识符区分大小写

■ /3 = enable 32-bit processing

允许 32 位处理。当目标模块含有 80386 处理器的 32 位代码时,应选用本项。使用本项时内存需求增加,且会减慢连接速度。

■ /v = include full symbolic debug information

包含全部符号调试信息。

■ /e = ignore Extended Dictionary

忽略全部扩展字典集,从而忽略了带有扩展字典集的库中的调试信息,程序连接时也少用内存。

■ /t = create COM file

连接后建立缺省扩展名为 .COM 的可执行文件,它无须使用 DOS 的 EXE2BIN.EXE 进行转换。仅对微存储模式(Tiny)使用。.COM 文件长度不能超过 64KB,它没有段的驻留内容,也没有定义堆栈段。注意:Turbo C 的微存储模式虽然和 COM 格式兼容,但是当程序中使用了浮点子程序(函数)时将不能被转换为 COM 文件。

## 41.2 连接 Turbo C 程序的方法

使用 TLINK 连接 TC 相关程序生成可执行文件时,必须包含存储模式所使用的一个初始化模块和适当标准库文件。

语法: TLINK [C0x] [( .OBJ), ( .EXE), [ .MAP], ( .LIB)] [emu.LIB 或 fp87.LIB mathx] [Cx]

1. C0x 是指初始化模块名 C0x.OBJ。Turbo C 共有六个初始化模块名: C0t.OBJ(微型

模式)、C0s.OBJ(小模式)、C0c.OBJ(紧凑模式)、C0m.OBJ(中模式)、C0l.OBJ(大模式)或c0h.OBJ(巨型模式);

2. 初始化模块名 C0x 必须作为第一目标出现!它安排程序各段的次序。如果不是第一个,程序段就不能被正确地存放到相应的存储区中,结果连接自然会有问题;如果没有连入正确的初始化模块,将产生类似标识符没找到或栈没有创建那样的错误。

3. 如果程序中用到浮点运算,就必须在命令行中包含一个浮点库(emu.lib 或 fp87.lib)和一个数学库(mathx.lib)。

4. emu.LIB 或 fp87.LIB 是浮点库,是任选项,它们与存储模式无关。

如果想包含浮点仿真程序以便程序在有无 8087/80287 协处理器的机器上都能运行,就必须使用 EMU.LIB。

如果程序在带有 8087/80287 协处理机上运行,可用 FP87.LIB,结果将产生一个较小而运行速度较快的执行程序。

如果程序中没有浮点运算,最好不包含这两个库中的任何一个库,以便减少连接时间。5. mathx 是指与存储模式库相关的数学库名: mathS.LIB(微型模式)、mathS.LIB(小模式)、mathC.LIB(紧凑模式)、mathM.LIB(中模式)、mathL.LIB(大模式)或 mathH.LIB(巨型模式)。是任选项。只有在用了浮点库时才选用。

6. Cx 是指与存储模式库相关的【运行时刻库】名: CS.LIB(微型模式)、CS.LIB(小模式)、CC.LIB(紧凑模式)、CM.LIB(中模式)、CL.LIB(大模式)或CH.LIB(巨型模式),是必选项。

7. 对应位置上的扩展名可缺省使用,即可以不写。

### 41.3 TCC.EXE 要使用 TLINK.EXE

TCC.EXE 可以作为 TLINK.EXE 的“前导程序”,它会正确地启动文件、库及调用 TLINK.EXE 以便进行正确的连接(参见 TCC.EXE 程序说明),此时 TCC.EXE 当前目录中应有 TLINK.EXE 文件。

注意:此时 TCC 命令行的文件扩展名应显式地给出。如给出命令行

```
C>TCC -mm MAINFILE.obj sub1.obj mylib.lib
```

TCC.EXE 将启动文件 C0M.OBJ、EMU.LIB、MATHM.LIB 和 CM.LIB 来调用 TLINK.EXE。TLINK.EXE 将这些文件和 MAINFILE.OBJ、SUB1.OBJ、MYLIB.LIB 连接在一起。

### 41.4 例子

1. TLINK□objfile1□objfile1,exefile,mapfile,libfile1□libfile2.lib
2. TLINK□objfile1□objfile1,,,libfile
3. TLINK□/c□@objfiles,exefile,mapfile,@listlibs
4. tcc□-mx□mainfile.obj□sub1.obj□mylib.lib
5. TLINK□/x□/l□c0s□MMYFILE,,,CS

## 41.5 混合模式的连接

如果编译的两个模块先前使用的模式不同（因其内容决定），现在又想将它们连接在一起，怎么办？

（一）首先考虑可能出现的麻烦，如

1. 小模式模块中函数将用near 调用指令调用大模式模块中函数，而不是用far 指令；
2. 小模式中函数希望传递和接受 near 指针，而大模式则希望用 far 指针。

（二）解决方法

使用函数原型。例如，有源程序 MYMAIN.

```
C>TYPE MYMAIN.C
#include <stdio.h>
#include "myputs.h"
main(){
 char near *mystr;
 mystr="Hello,word\n";
 myputs(mystr);
}
```

中新增了一个头文件 MYPUTS.H,它只有一个函数原型构成：

```
C>TYPE MYPUTS.H
void far myputs(char far *s);
```

这样，即使在小模式下，也能生成适当的调用代码，因为当编译程序从 MYPUTS.H 中已知道 myputs 将是期望的一个有 far 指针 far 函数。当然大模式下编译更好。

在库子程序连接时，最好的办法就是使用某一大存储模式库，并将所有的函数和指针说明成 far。例如：你可以将头文件 STDIO.H 拷贝一个备份，取名为 FSTDIO.H，然后编辑其内的函数原型，使函数和指针都有 far。譬如，将

```
int _Cdecl printf (const char *format, ...);
```

改成

```
int far _Cdecl printf (const char far *format, ...);
```

现在你将源程中包含 STDIO.H 改成 FSTDIO.H，便可以编译了，因为不但函数调用变成了 far 调用，而且指针传递也将使用 far。

对这种混合方式用 TCC.EXE 编译程序（用 TLINK 连接）时要指定一大模式库，如 CL.LIB。

混合方式比较复杂，但能实现。使用不当，错误难找，尽量避免。

在集成环境下，使用 \*.PRJ 文件对多个模块进行编译连接时也必须注意防止混合模式的发生，即是说，所有模块 (\*.OBJ) 应用同一种存储模式生成，否则会产生意想不到的效果。为此，当你编译前改变存储模式时，要检查先前的模块是否是用同一种存储模式生成的，必要时应当在新的存储模式下重新进行编译。或者，一种简单的办法是在 DOS 环境下使用 DEL 命令将相关的 OBJ 文件全删除掉，然后就可用 TC 对全部源文件用同一存储模式进行编译。

## 41.6 可能产生的错误信息

1. Not enough memory  
无足够内存。
2. 32-bit record encountered in module s  
在模式 s 中 32 位记录冲突。
3. Invalid group definition  
无效的 GROUP 定义。
4. invalid entry point offset  
无效的入口偏移量。
5. invalid initial entry point address  
无效的初始化入口地址。
6. Cannot generate COM file  
不能建立 .COM 文件。
7. relocation table full  
重定位表满。
8. base fixup offset overflow  
基本的装设偏移溢出。
9. Fixup overflow in module  
在模式中装设溢出。
10. Undefined symbol in module  
在模式中不能定义符号。
11. no stack  
连接时无堆栈。这通常在用微存储模式 (Tiny) 编译连接时发生。
12. program exceeds 64K  
程序超出 64K。
13. segment/group exceeds 64K  
SEGMENT/GROUP 超出 64K。
14. invalid initial stack offset  
无效的堆栈偏移量。
15. Not enough memory to link  
连接用内存不够。

## 第四十二章 独立的管理开发程序 MAKE.EXE

TC 软件包中有一个 MAKE.EXE 程序,它是提供给程序员来管理开发程序的,它不在集成环境里,使用它必须单独执行 MAKE.EXE(在不混淆时简称 MAKE)。概括地说,它具有下列功能:

1. 当大型、复杂程序(往往有许多文件构成)中部分文件变动时,用它可只对这部分变动程序进行重新编译和连接。由于避免了对全部程序进行编译和连接,因而为用户节省了时间。

2. 能备份文件。

3. 能从不同子目录里提取文件。

4. 当程序使用的数据文件被修改后能自动运行程序。

MAKE.EXE 主要是针对(嵌入)汇编的,因为在通常情况下利用集成环境就能实现类似的功能,用户一般不必考虑与之相关的操作细节。

### 42.1 文件间的依赖关系

假定下列 C 源文件中用到的包含文件为

| C 源文件      | 包含文件                                       |
|------------|--------------------------------------------|
| STARLIB.C  | 未用                                         |
| GSPARSE.C  | stardefs.h                                 |
| GSCOMP.C   | stardefs.h, starlib.h                      |
| GETSTARS.C | stardefs.h, starlib.h, gsparse.h, gscomp.h |

为产生 GETSTARS.EXE 执行文件,使用操作

```
C>TCC -c -mm -f starlib
C>TCC -c -mm -f gsparse
C>TCC -c -mm -f gscomp
C>TCC -c -mm -f getstars
C>TLINK lib\COm starlib gsparse gscomp getstars,getstars,getstars,lib\emu,lib\mathm lib\cm
```

显然,修改其中一个头文件就可能要对与该头文件相关的 C 源文件重新编译一遍。自然,最后还要重新进行连接。这就是文件间的依赖关系。

为使用 MAKE.EXE,现在根据上面的依赖关系和编译连接命令可以建立一个 MAKE 文件,并给它取名为 MAKEFILE。

```
C>TYPE MAKEFILE
getstars.exe : getstars.obj gscomp.obj gsparse.obj starlib.obj
tlink lib\COm starlib gsparse gscomp getstars, \
 getstars,getstars,lib\emu lib\mathm lib\cm
getstars.obj : getstars.c stardefs.h starlib.h gscomp.h gsparse.h
tcc -c -mm -f getstars.c
```

```

gscomp.obj : gscomp.c stardefs.h starlib.h
 tcc -c -mm -f gscomp.c
gsparse.obj : gsparse.c stardefs.h
 tcc -c -mm -f gsparse.c
starlib.obj : starlib.c
 tcc -c -mm -f starlib.c

```

MAKE.EXE 将按下列原则解释执行它们：

getstars.exe 依赖于它后面的 4 个 .obj 文件，只要这 4 个 .obj 中有一个发生了变化，就重新编译，并通过 TLINK.EXE 重新进行连接。同样，getstars.obj 依赖于它后面的 5 个（一个 C 源文件，4 个 .OBJ 文件），只要这 5 个文件中有一个发生变化，则利用 TCC.EXE 重新编译。如此等等。每一次，当你只对其中任一个文件改变后，只要用 C>MAKE 操作，MAKE.EXE 启动并自动寻找 MAKEFILE 文件（当你没有专门指定 MAKE 文件时，MAKE.EXE 自动寻找这个缺省文件）。检查（主要看文件修改日期和时间有否变化）后自行作出相应的反映（要否重新编译连接）。显然，这就大大减轻了程序员的负担，而且准确可靠。

## 42.2 MAKE 文件

事实上上面的例子给出了常用的 MAKE 文件生成过程和形式，下面我们将指出其更一般性的生成过程和形式。

执行 MAKE 需要使用【MAK 文件】，它是一种专为 MAKE 服务的文件，有定义、命令和指令，形式上有点像一个程序文件，但其构成有严格的规定，包括以下 5 个部分：

- 注释
- 显式规则
- 隐式规则
- 宏定义
- 指令：包含文件、条件执行、错误检测及解除宏定义等。

它是一个 ASCII 码文件，可以用 WORDSTAR 等字处理软件编辑，也可以用集成环境里的编辑器编辑。

### 42.2.1 注释

#### 1. 作用

用于说明 MAKE 文件的功能。

#### 2. 书写规则

(1) 注释以井号（#）开头，后跟注释用正文串。该正文串在 MAKE 执行时被忽略，即不起作用；

(2) 注释可以在一行中任一位置（列）开始；

(3) 当注释在一行内写不下而要写到下一行上时，不能在行尾用反斜杠（\）作续行符，而必须在续行上的注释串前面也加上井号（#）。反斜杠如在井号后面，则它被认为是注释内容的一部分；

(4) 不能在显式规则的行之间插入注释；

#### 3. 例子

为明确起见,下面这个 MAKE 文件中将注释正文串用中文表示。

```
C>TYPE GETSTARS.MAK
#GETSTARS.EXE 的MAKE 文件
#对整个规划 (project) 进行维护
getstars.exe; getsars.obj□gscomp.obj□gsparse.obj□starlib.obj
#不能在下一行末书写注释,因为该行还有续行。
tlink□lib□c0m□starlib□gsparse□gscomp□getstars,getstars,\
 getstats.lib\emu□lib\mathm□lib\cm
#合法注释
#不能在下面两显式规则行之间书写注释
getstars.obj,getstars.c□stardefs.h□starlib.h□gscomp.h□gsparse.h
 tcc□-c□-mm□-f□getstars.c□#you can put a comment here
#MAKE 文件结束
```

#### 42.2.2 显式规则

##### 1. 格式

```
target□[target...]:[source...]
 □[command]
 □[command]

```

说明:

(1) target( 对像文件 )

是需要更新的文件,可以有多个。文件可以含有驱动器名与路径名,但不能含有通配符 (\* 和 ?)。【对像文件】可以是通常的目标文件 (\*.OBJ)、库文件 (\*.LIB) 或执行文件 (\*.EXE) 等。一个显式规则里的对象文件也可能是另一个显式规则里的源文件。

(2) source( 源文件 )

是对像文件的【依赖文件】,可以是一个,也可以有多个。所谓依赖文件,就是那些能帮助建立对像文件的文件,如 C 源文件 (\*.C) 与头文件 (\*.H) 等。源文件可以含有驱动器名与路径名,但不能含有通配符 (\* 和 ?)。

(3) command( 命令 )

是任何有效的 MS-DOS 命令(包括命令行参数),包括调用 \*.BAT 文件、\*.EXE 和 \*.COM 等执行文件等。命令主要用来建立或更新对象文件。

##### 2. 书写规则

(1) 规则必须从一行的第 1 列开始书写;

(2) 命令 (command) 前面至少要有一个空格符。命令行的续行的左边开始列上至少要有一个空格符;否则,第一列为非空白符的行有可能被看作上面命令的一部分。

(3) 如规则在一行内写不完,可以在行尾加反斜杠 (\) 表示续行,余下的内容可写到下一行上;

(4) 在 MAKE 文件中,一个对象文件在显式规则的左边只能出现一次;

(5) 源文件和命令都是任选的,它们可以含有 target □ [target...] 后跟冒号的显式规则。

(6) 无任何字符的空行允许存在,因为 MAKE 将忽略它。

### 3. 例子

下面的中文注释不是文件的一部分。

```
C>TYPE MYPROG.MAK
```

```
myprog.obj:myprog.c # 第一个显式规则,myprog.obj 依赖于myprog.c
□□tcc□-c□myprog.c # myprog.obj 在执行TCC.EXE 后建立
prog2.obj:prog2.c□include\stdio.h # 第二个显式规则,prog2.obj 依赖于文件
□tcc□-c□-K□prog2.c # prog2.c 和stdio.h(在子目录include 中)
 # prog2.obj 用TCC 建立
prog.exe:myprog.c□prog2.c□include\stdio.h # 第三个显式规则,prog.exe 依赖
□□□tcc□-c□myprog.c # 于myprog.c,prog2.c 和stdio.h
□□□tcc□-c□-K□prog2.c # 只要三个依赖文件中有一个改变
□□□□tlink□lib\c0s□myprog□prog2,prog,,lib\cs # 随后的三行命令将被执行
```

如果把第三个显式规则改成

```
prog.exe:myprog.obj□prog2.obj
□tlink□lib\c0s□myprog□prog2,prog,,lib\cs
```

则好得多,因为现在当三个依赖文件中有一个改变时,不必对前两个显式命令都执行,而只需执行其中的一个便可以了。事实上,这与 MAKE 规定的执行过程有关。

#### 4. MAKE 对显式规则的处理

(1) MAKE 遇到一个显式规则,首先检查其中的源文件是否是其它地方的对象文件。如果是,则先去执行那个地方的规则;

(2) 一旦所有的源文件被建立和更新后,MAKE 便检查是否有对象文件存在。如果没有,则用给出的顺序调用命令。如果存在,则把对象文件上次修改的时间和源文件修改的时间相比较;如果源文件比对象文件修改的更晚,则执行规则中的命令。

(3) MAKE 忽略空行;

(4) 显式规则中跟不跟命令行的区别;

跟命令行:对象文件只取决于列出的依赖文件;

不跟命令行:对象文件除取决于列出的依赖文件外,还依赖于和对象文件匹配的隐含规则的文件。

### 42.2.3 隐含规则及部分 DOS 命令

隐含规则用于简化显式规则。

#### 1. 格式

```
.source—extension.target—extension:
{command}
{command}
.....
```

说明:

(1) .source—extension k 源文件扩展名,如 .C

(2) .target—extension k 对象文件扩展名,如 .OBJ 所以显式规则中当文件基本名一样而只是文件扩展名不同时,如

```
any.obj:any.c
```



可以写成

.c.obj;

(3) command k 类似于显式规则中的命令,但已不纯粹是 MS-DOS 命令,每个命令行由可选择的前缀表和命令体组成。

[prefix...]□command-body

prefix...是前缀表,command-body 是命令体。

#### <1> 前缀表

命令中的前缀规定限制 MAKE 的命令的处理方式。前缀基本形式有两种,可以组合使用。

##### —1 前缀是一个字符 @

前缀 @ 将阻止 MAKE 执行当前命令前显示该命令。即使在命令行中不给出 -s 选择项,命令也不会被显示(参见本章42.3节)。前缀只对其所在的命令行起作用。

##### —2 前缀是一个连字符(-)后跟数字,例如,-4。

使用这个前缀影响 MAKE 如何处理【退出码】。退出码是 MAKE 执行中间退出时返回的代码。

情况1:如果有连字符且指定了数字,只有当退出码大于给出的数字时 MAKE 才终止;

情况2:如果有连字符但无数字,则 MAKE 不检查退出码,即不管退出码如何,MAKE 继续执行;

情况3:如果没有前缀,MAKE 要检查退出码,只要退出码非零,MAKE 将终止并删除当前对象文件。

#### <2> 命令体

它是 MS-DOS 的 COMMAND.COM 的一个内部命令,但不支持重定向和管道两种命令。【重定向】指重定向命令流的输入及输出(Turbo C 也可以实现重定向,(参见《文件管理》一章中 freopen() 函数),主要有三种:

##### —1 改向输出:在命令中使用一个大于符号(>)。

例如:

键入命令

```
C>DIR>D:MYFILE
```

则列出的 C 盘上的文件目录不是在屏幕上出现,而是被装入 D 盘上的文件 MYFILE 内。

##### —2 追加输出:使用两个大于符号(>>)。

例如

```
C>DIR>>D:MYFILE
```

把输出的目录追加到文件 D:MYFILE 的末尾。

##### —3 改向输入:使用一个小于符号(<)。

例如

```
C>SORT<MYFILE.OLD>MYFILE.NEW
```

排序命令 SORT 的输入来自一个文件 MYFILE.OLD,排序的结果存入文件 MYFILE.NEW。

【管道】就是将一命令的输出当作另一命令的输入。管道是使用垂直的断开线符号 (|) 来表示。例如

C>DIR|SORT

表示将全部输出目录作为 SORT 命令的输入。

注意:有些版本的 MS-DOS 无此两种功能。

MAKE 通过调用 command.com 副本执行命令体中的命令,可执行的 DOS 内部命令 (它们是 command.com 文件的一部分,因此在用 dir 命令列目录时将看不到这些命令名。在启动 MS-DOS 时,内部命令被装进内存,故当在 DOS 提示符下键入内部命令时,MS-DOS 便立即执行它们。凡是扩展名为 .com、.exe 和 .bat 的文件均可看作是 DOS 的外部命令。在 DOS 提示符下键入外部命令时不必键入文件的扩展名。)有

#### —1— BREAK

格式: BREAK ON 或 BREAK OFF

功能:设置 Ctrl-C 检查。BREAK OFF 仅在 (屏幕、键盘或打印机) 读写操作时检查是否按了 Ctrl-C 键。如按了,则终止程序执行;而 BREAK ON 则将功能扩大到诸如磁盘操作等范围。如键入 BREAK 并按回车后可以看到当前 BREAK 处于何种状态,是 ON 或是 OFF。

注意:有些用户程序中可设置自己的响应任意时刻发出的 Ctrl-C。

#### —2— CD 或 CHDIR

格式:CHDIR [路径]

功能:CD\TC 操作是将当前目录置为子目录 TC; CD.. 操作是将目录移到其双亲目录; CD 是显示当前目录名。参见《目录函数》一章。

#### —3— CLS

清除屏幕。

#### —4— COPY

格式: COPY [a/或/b] [源文件] [a/或/b] [+源文件 [a/或/b] [+...]] [目标文件 [a/或/b]] [/v]

功能:将源文件拷贝为目标文件

说明: /a 或 /b 是拷贝开关,它们能否起作用和它们在命令上的位置相关。当一个开关设置后如果没有遇到另一个开关,则该开关一直起作用。

##### a. 拷贝开关

/a 或 /A

当它在源文件名后出现时,被拷贝文件 (源文件) 被处理为一个 ASCII 码文件,拷贝中一旦遇到【文件结尾标志字符】(ASCII 的十进制数是 26 或十六进制数 1AH) 便结束拷贝,即该字符将不被拷贝,文件的其余部分也不再被复制;当它在目标文件后出现时,拷贝结束时自动将 1AH 添到文件尾部,所以拷贝后生成的文件 (目标文件) 最后一个字符一定是文件结束字符 (1AH)。

/b 或 /B

当它在源文件名后出现时被拷贝文件被处理为一个二进制文件,不管这个文件中是否有文件结尾标志字符 (甚至有多),文件的所有字符均被拷贝 (包括 1AH) 字符。当它在目标文件名后出现时,文件结尾也不会加上文件结尾标志字符 1AH。

例如：有一个可执行文件 MY.EXE（二进制文件），内部无一个 1AH（甚至在文件结尾），字节数假定为 755。分别按

```
C>COPY MY.EXE /A MYA 用开关 /A
C>COPY MY.EXE /B MYB 用开关 /B
C>COPY MY.EXE MY 未用开关
```

拷贝后列表，它们的文件长度分别为（字节数）

```
MY.EXE 755
MYA 756
MYB 755
MY 755
```

为什么 MYA 比 MYB（或 MY）多一个字节？用 DEBUG.COM 等工具软件对其分析，可知其尾部增加了字符 1AH。现用 DEBUG 的 E 命令将 MY.EXE 的前两个字节改成 41H 和 1AH，然后用 W 命令存盘，再用上述方法拷贝并列表，各文件长度为

```
MY.EXE 755
MYA 2
MYB 755
MY 755
```

注意：由重定向操作产生的文件在结尾处是无 1AH 的，因此仅管你用 DOS 的 TYPE 命令能正确显示其内容，但如你用 WORDSTAR 字处理软件编辑，在文件结尾处将出现一些意外符号，编辑时如未将它们删除而直接按 F10 键则可能出现死机，无法进行编辑下去。解决的方法之一是将光标逐渐移到该行上，然后用 Ctrl-Y 命令将该行删除，或者也可在进入 WORDSTAR 前用

```
C>COPY 源文件 /A 目标文件
```

的方法，然后对目标文件编辑。

/v 或 /V

MS-DOS 将验证记录在目标盘上的扇区，如果不能写，便显示错误信息。使用此开关将使拷贝速度变慢。

b. 附加文件用符号（+）

加号（+）是将多个文件附加到一起，变成一个文件，文尾有 1AH。例如，

```
C>COPY A:MY1+B:MY2 C:MY 将 B:MY2 附加到 A:MY1 后面，生成新文件 MY
```

或

```
C>COPY A:MY1+B:MY2 将 B:MY2 附加到 A:MY1 后面，生成新文件 MY1
```

c. 组合文件

```
C>COPY *.TXT MY.DOC
```

该命令将所有 \*.txt 文件组合进一个文件 MY.DOC 中。

```
C>COPY *.TXT MY.txt
```

该命令将所有 \*.txt 文件（但中间不能有 my.txt 文件）组合进一个文件 MY.txt 中。  
d. DOS 5.0 还有一个更改文件日期和时间的拷贝方式

C>COPY MY.C+,,

则将 my.c 文件的日期和时间改为当前日期和时间。

对拷贝的其它说明（如不能拷贝子目录及文件长度为 0 的文件，也不允许把一个文件复制到自身等）参见有关 MS-DOS 手册。

#### —5— CTTY

格式:CTTY[设备名]

功能:允许改变发布命令的装置(控制台,一般指键盘和屏幕),仅对 MS-DOS 的程序起作用。

说明:有效的设备名为 LPT1(或 PRN)、LPT2、LPT3;CON、AUX、COM1、COM2、COM3 和 COM4 等。

例

CTTY[AUX] 把所有 I/O 从当前设备(控制台)移到 AUX 端口(另一个终端)。

CTTY[CON] 把 I/O 移回到控制台。

#### —6— DATE

格式:DATE[mm-dd-yy]

功能:查询系统日期或更改它

说明:

mmk 指月(1~12);ddk 指日(1~31);mmk 指年(80~99 相当于1980~1999 年,而输00~79 相当于2000~2079 年)。

键入DATE 后回车,屏幕显示系统日期。如不想改变,再按回车键;否则输入新日期后再回车。

#### —7— DEL 或 ERASE

格式:DEL[文件名]

功能:删除指定文件,文件可以带路径名,也可用通配符 \* 与 ?。

#### —8— ECHO

格式:ECHO ON 或 ECHO OFF

功能:使显示可见或不可见。

#### —9— MD 或MKDIR

格式:MD[当前目录下的目录名]

功能:建立一个新的子目录。(参见《目录函数》一章。)

#### —10— PATH

格式:PATH[路径1][;路径2...]

功能:设置命令搜索路径。

例如:PATH \USER\MY1;B:\USER\MY2。(参见《程序结构和主函数》一章。)

MAKE 利用 MS-DOS 查找其它命令(如找文件)的算法:

先在当前目录中查找。找不到,便按 PATH 指定的路径查找。

#### —11— PROMPT

格式:prompt[指定字符...]

功能:改变 MS-DOS 的提示符形式。(参见《程序结构和主函数》一章)。

—12— REM

格式:REM 要注释的内容

功能:注释。

—13— REN 或 RENAME

格式:REN□老文件名□新文件名

功能:更改文件名。

—14— SET

格式:SET□[串=[串]]

功能:在命令中设置环境串。(参见《程序结构和主函数》一章。)

—15— TIME

格式:TIME□[小时:分钟][,秒][百分之一秒]

功能:显示及设置时间。

—16— TYPE

格式:TYPE□文本文件名

功能:在屏幕上显示文本文件的内容。

—17— VER

格式:VER

功能:显示 DOS 版本号。

—18— VERIFY

格式:VERIFY□ON 接通

VERIFY□OFF 关闭

VERIFY 查看当前校验开关。

功能:在写盘时接通或关闭校验。

—19— VOL

格式:VOL A:显示A 盘

VOL 显示缺省盘

功能:显示指定驱动器中盘的卷标或卷 ID (若它存在)。

注意:尚有以下内部命令它未提到:

—1— dir

列出磁盘上目录或文件,(参见《目录函数》一章。)

—2— exit

退出命令处理器,返回到先前的命令层。

—3— for

对一系列文件执行一条命令(循环)。其格式为:

在命令行上用 for %变量名 in (set) do 命令 [命令参数]

在批处理文件中用 for %%变量名 in (set) do 命令 [命令参数]

例1

C>for %f in (\*.asm) do MASM %f

它相当于执行形如

C>MASM my.asm

C>MASM you.asm

的命令,其中 my.asm 和 you.asm 是当前目录里仅有的两个 .asm 文件,即 \*.asm。%f in (\*.asm) 告知 DOS 用这两个文件替代 MASM 后面的变量 %f。

例2

C>FOR %f in (\*.doc \*.txt) do type %f>prn:

它将当前目录里扩展名为 .doc 和 .txt 的所有文件输出到打印机。这也说明 set 可以是一个文件名或命令行的任意部分。

—4— goto 标号

它从指标号开始处理命令。标号用冒号定义,如

:end

注意:对 Turbo C 而言,标号是像 end:。

—5— if

条件满足时执行命令。

—6— pause [提示信息]

在批处理文件 (\*.bat) 处理期间暂停并显示提示信息。

—7— rmdir

删除子目录。

—8— shift

它后面没有其它参数。在批处理文件中用它改变可替换参数的位置。批处理文件中一般可处理十个替换参数(%0 ~ %9),如多于十个,可用本命令。例如在命令行上如给的参数多于十个,则在执行一次本命令移动后,原在第十个参数(%9)后面的那个参数被移到第十个参数的位置。

#### 42.2.4 宏

当 MAKE 遇到一个宏调用时,它就用定义该宏的宏体取代宏名,这就是宏扩展(参见《预处理指令与编译控制行》一章)。

##### 1. 宏的定义

使用宏定义的目的是为了简写MAKE文件中许多重复的符号,有时也是为了达到“一改百改”的目的,但由此可能使阅读增加了困难。

格式:宏名=宏体

说明:

(1) 宏名由字母 A ~ Z, a ~ z 或 0 ~ 9 构成的正文串组成,且应以字母开头。字母大小写是有区别的。

(2) 宏名和等号之间允许有空格符;

(3) 宏名中不允许再出现宏(调用);

(4) 新定义的宏代替在此之前已定义的宏;

(5) 宏体也是正文串,其间允许出现另外的宏调用(这就是【宏中宏】)。嵌在宏体中的宏只有在宏被调用时才会被扩展。

(6) 宏是先定义后应用。

##### 2. 宏的调用

宏调用可以出现在宏体中;也可以出现在显式或隐式规则中;在命令和指令中也可以调用。在执行 MAKE 文件时宏将被展开。在 !if 或 !elif 指令中宏调用时,如果当前没有定义,则扩展成值 0(False)。格式中的圆括号是必须的,即使宏名只有一个字符时也如此。

格式1:\$(宏名)k 扩展时宏名被宏体代替。

格式2:\$d(宏名)k 宏定义测试宏(MAKE 内部宏)。如宏名已定义,宏扩展为 1,否则为 0,这与宏体无关。

例1

MAKE 文件开始部分为

```
! if ! $d(TC) # 如果TC 在命令行上没有定义
TC=C:\TC\INCLUDE\ # 定义缺省路径
! endif
```

现在用(参见本章42.3节)

```
C>MAKE -DTC=C:\TC\SYS\
```

这时TC就被定义为C:\TC\SYS\  
而

```
C>MAKE
```

即未在命令行上使用参数时,则 TC 便被定义为 C:\TC\INCLUDE\

格式3,\$\*k 基本文件名宏(MAKE 内部宏),该宏无宏体。它用于显式或隐式规则,展开成不含扩展名的文件名。

例2

对显式规则:

```
getstars.exe: getstars.obj gscomp.obj gsparse.obj \
starlib.obj
tlink starlib gsparse gscomp getstars, getstars, getstars, \
lib\emu lib\mathm lib\io
```

可改写成

```
getstars.exe: getstars.obj gscomp.obj gsparse.obj \
starlib.obj
tlink starlib gsparse gscomp $*, $*, $*, \
lib\emu lib\mathm lib\io
```

当执行本规则命令时,宏 \$\* 就用目录文件(不带扩展名)getstars 所替代。

对隐含规则可定义

```
.asm.obj:
tasm /t $*
```

格式4:\$<k 完整文件名宏(MAKE 内部宏)。在显式规则中,它展开成目标文件名(包括扩展名);在隐式规则中,\$<是文件名加源扩展名。

例3

```
starlib.obj, starlib.asm
```

```
copy $< \oldobjs
tasm /t s*
```

它先将 starlib.obj 拷贝到 \oldobjs 目录下,然后用 tasm 编译 starlib.asm 文件。

例4

```
.obj.asm:
tasm /t $ *.asm
```

可写成

```
tasm /t $<
```

格式5: \$: — 文件名路径宏 (MAKE 内部宏)。展开成没有具体文件名的路径名。

格式6: \$. — 文件基本名和扩展名宏 (MAKE 内部宏)。展开成不含路径的文件名,即只包含文件基本名和扩展名。

格式7: \$& — 文件名宏 (MAKE 内部宏)。展开成的文件无路径名和扩展名。

#### 42.2.5 指令

指令必以感叹号 (!) 开头。指令有 7 种形式: ! include、! if、! else、! elif、! endif、! error 和 ! undef。可以使用指令包括其它的 MAKE 文件,使规则和命令 条件执行,输出错误信息,解除宏定义等。

—1— ! include

包含指令。当 MAKE 遇到此指令时,它就打开指定的包含文件,并读入其内容,使其变成当前 MAKE 文件的一部分。

例

```
! include "filename" 或 ! include <filename>
```

—2— ! if                如果

—3— ! else             否则

—4— ! elif             否则如果(相当于 else if)

—5— ! endif            结束

这 4 个指令构成【条件组句】,它至少由一个 ! if 指令开头, ! endif 指令结尾。

例1 如果表达式成立(值为1),那么执行[MAKE 文件行 ...]

```
! if 表达式 #条件行开始
[MAKE 文件行...]
! endif #条件行结束
```

例2 如果表达式成立,那么执行[MAKE 文件行1...],否则执行[MAKE 文件行2...]

```
! if 表达式 #条件行开始
[MAKE 文件行1...]
! else
[MAKE 文件行2...]
! endif #条件行结束
```

例3 如果表达式 1 成立,那么执行 [MAKE 文件行 1...], 否则如果表达式 2 成立,那么执行 [MAKE 文件行 2...]



```

! if 表达式1 # 条件行开始
[MAKE 文件行1...]
! elif 表达式2
[MAKE 文件行2...]
! endif # 条件行结束

```

对条件组句的规定：

(1) MAKE 文件行可以包括：

- 宏定义 (macro definition)
- 显式规则 (explicit rule)
- 隐含规则 (implicit rule)
- 包含文件指令 (include directive)
- 条件组句 (if group)
- 错误指令 (error directive)
- 解除宏定义指令 (undef directive)

(2) 有一个 ! if 指令,必须有一个与之配对的 ! endif ;反之亦然。

(3) ! elif 指令可以出现在 ! if 和 ! else 指令之间。组中可出现 ! else 指令。

(4) 宏和其它指令可以以任意数目出现在不同的条件指令之间。

(5) 完整的规则和命令不能用条件指令分隔开。

(6) 条件指令可以嵌套 (即条件指令后又有条件指令等) 任意深度。

(7) 表达式的构成：

· 常量：十进制数、八进制数 (如 0677) 和十六进制数 (如 0X23AF)

· 操作符：单目 (—、~、!) ; 双目 (+、—、\*、/、%、>>、<<、&、|、^、&&、>、<、>=、<=、==、!=) ; 三目 (? : )。它们的含意和使用规则同 C 语言。

· 在表达式中可以调用宏 \$d() ,但宏展开后必须符合语法规则。

· 表达式求值结果为 32 位带符号的整数。

(8) 任何规则、命令或指令必须在单个文件中结束,即不能骑跨在两个文件内。

—6— ! error

执行到本指令时,执行被终止,随后 MAKE 输出错误信息。

例

```

! if ! $d(MDL) # 如果没有定义了 MDL, $d(MDL)=0, 则! $d(MDL)=1
! error MDL not defined # 没有定义MDL 时本句将被执行,! error 后的内容被原样输出
! endif # 条件结束行

```

输出信息为：

Fatal makefile 5,error directive: MDL not defined

—7— ! undef

解除先前定义的宏。

例

```
! undef MDL
```

则原定义的 MDL (宏名) 被取消定义,即不再起作用。如果宏名原先没有被定义,则本指令不起作用。

## 42.3 使用 MAKE 的方法

在 DOS 提示符下键入 C>MAKE -? 屏幕显示使用 MAKE 的帮助信息:

MAKE Version 2.0 Copyright (c) 1987, 1988 Borland International

Syntax: MAKE [options ...] target[s]

|                 |                                                   |
|-----------------|---------------------------------------------------|
| -Dsymbol        | defines symbol                                    |
| -Dsymbol=string | defines symbol to string                          |
| -Idirectory     | names an include directory                        |
| -ffilename      | uses filename as the makefile                     |
| -a              | performs auto-dependency checks for include files |
| -n              | prints commands but does not do them              |
| -s              | silent, does not print commands before doing them |
| -? or -h        | prints this message                               |

语法: MAKE [options...]target[s]

说明:

1. options 选择项由连字符(-)开头,其后有时紧跟一些字符,内中字母大小写是有区别的。多个选择项之间应用空格隔开,选择项之间的排列次序是任意的。

■ -Dsymbol defines symbol

即 -D 符号,定义单个字符组成的有名标识符。

■ -Dsymbol=string defines symbol to string

即 -D 符号=字符串,定义符号为等号后面的字符串。串中不能含有空格或制表符。

■ -Idirectory names an include directory

即 -I 目录名,指定一个包含文件目录。MAKE 先在当前目录中寻找指定文件,如未找到会再到包含文件目录中寻找。

■ -ffilename uses filename as the makefile

即 -f 文件名,指定一个 MAKE 文件名,例如 MYFILE. \$\$\$。如果 MYFILE. \$\$\$ 不存在,或者它没有给出扩展名 .MAK, MAKE 将去查找缺省文件 MYFILE. MAK。

■ -a performs auto-dependency checks for include files

对包含文件进行自相关性检查。

■ -n prints commands but does not do them

输出命令,但不执行。这常用于调试 MAKE 文件。

■ -s silent, does not print commands before doing them

执行命令前不把该命令显示出来,即保持沉默(silent)。否则,不用此选项时会将命令显示。

■ -? or -h prints this message

输出帮助了解 MAKE 语法的信息。

2. target 是目标文件名。多个目标文件名之间用空格隔开。如

C>MAKE -Iinclude -DMDL=compact -fss.mak

MAKE 按顺序执行目标,重新编译必须编译的部分。如果命令行中不含目标名,MAKE 使用第一个显式规则中的目标。如果命令行中有多个目标,需要时它们都会被执行。

3. 用 Ctrl-Break 或 Ctrl-C 会中止当前执行的命令及 MAKE 的继续执行。

## 42.4 BUILTINS.MAK 文件的使用

Turbo C 盘上并没有这个文件的内容。当使用 MAKE 时,常常会发现有许多宏和规则(通常是隐含规则)被不断使用。已有了处理它们的三种方法。第一种是把它们放到创建的每一个 MAKE 文件中;第二种方法是把它们放在一个文件中(类似于 C 的标头文件 \*.h),并在每个创建的 MAKE 文件中使用 `! include` 指令包含这个文件;第三种方法便是把它们放到一个叫 BUILTINS.MAK 文件中。

每当运行 MAKE.EXE 时,它首先去查找 BUILTINS.MAK 文件。如果找到了,MAKE.EXE 就在处理 MAKE 文件之前把该文件读入并解释执行。如果没有找到,则直接处理指定的 MAKE 文件或缺省的名为 MAKEFILE 的文件。

该文件中可使用任何规则(通常是隐含规则)和宏,可被计算机的所有 MAKE 文件使用。该文件并不一定要求存在。如存在,一般放在 MAKE.EXE 文件所在的目录下。

## 42.5 MAKE 错误信息

有星号(\*)标志的为致命错误,此时执行被立即停止。

1. Bad filename format in include statement

在包含语句中书写文件名格式不对。例如,无西文双引号或尖括号对。

2. Bad undef statement syntax

`! undef` 语句错误。注意:它只能有一个宏名。

3. Character constant too long

字符常量太长。

4. Command argument too long

命令行上参数太多。

5. Command too long

命令行上字符超出了 127 个(MS-DOS 的规定)。

6. Command syntax error

出现命令语法错误。如

- . MAKE 文件规则的第一行的开头有空白字符;
- . 隐含规则未由 .EXT.EXT;那样的格式构成;
- . 在冒号(:)前面显式规则没有含对象文件名;
- . 在等号(=)前宏定义没有给出宏名。

7. Division by zero

表达式中用 0 做除数不允许。

- \* 8. [s] does not exist — don't know to make it

s 不存在,不知道怎样对它操作。

- \* 9. Error directive [s]

s 是错误的指令。

10. expression syntax error in ! if statement

在 `! if` 指令中出现表达式语法错误,如圆括号不配对,有额外的操作符或常量等。

11. File name too long  
在 ! include 指令中文件名太长 ( 加路径等应少于 78 个字符 )。
12. If statement too long  
! if 语句太长。
13. Illegal character in constant expression  
在常量表达式中出现非法字符。如单词拼错。
14. Illegal octal digit  
非法八进制数。例如含有数字 8 或 9。
- \* 15. Incorrect command line argument :[s]  
不正确的命令行参数 s。
16. Macro expansion too long  
宏扩展后字符数太多 ( 超过 4096 个 )。如果宏是递归地扩展自己,常会发生这种错误。一个宏不能对自身进行合法扩展。
17. Misplaced elif statement  
! elif 指令前面没有匹配的 ! if 指令。
18. Misplaced else statement  
! else 指令前面没有匹配的 ! if 指令。
19. Misplaced endif statement  
! endif 指令前面没有匹配的 ! if 指令。
20. No file name ending  
在 ! include 指令中文件名后缺 " 或 >。
- \* 21. Not enough memory  
无足够内存。
22. Redefinition of target [s]  
对象文件 s 重定义。例如对象文件在多个显式规则的左边出现。
23. Rule line too long  
规则行字符数太多。
- \* 24. Untable to execute command  
不能执行命令。例如没有找到命令,或命令拼写错误等。
25. Untable to open include file [s]  
不能打开包含文件 s。包含文件不允许包含它自身。
26. Untable to open makefile [s]  
不能打开 MAKE 文件 s。
27. Unexpected end of file in conditional started at line [s]  
条件指令在 s 行开始后直到文件结束都未遇见 ! endif 指令。
28. Unknown preprocessor statement  
不是合适的语句。例如在行首遇到感叹号 (!),但后面跟的字符不是指令的组成部分。

## 第四十三章 库管理程序 TLIB.EXE

库是一个单独的整体,它搜集了一些目标模块(.OBJ 文件)。库可以加速连接器(TLINK.EXE)对多个目标模块的连接,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。当在程序中连入库时,连接器就浏览库并自动地选出当前程序需要的那些模块,再加以连接。由于有了库,你就可以将那些成功的源程序(例如 MY.C)先经过编译成 MY.OBJ 文件(也可在集成环境中选 Compile to OBJ 命令),然后用 TLIB 将 MY.OBJ 文件装入某个库内。以后,当要用到 MY.C 时,当前源程序中可调用 MY.C 中的函数,但可不再包含 MY.C 源程序本身,而只要在 .PRJ 文件里包含 MY.OBJ 即可。由此可见,TLIB 是用来管理库的工具。管理包括建立用户自己的库,修改用户库、TC 库及其它程序员建立的库。其内容包括(括号内为基本操作):

1. 用一组目标模块建立新库(TLIB 新库名 + 目标块名);
2. 将目标模块或其它库加到一个已有库中(TLIB 已有库名 + 目标块名);
3. 从原有库中删除目标模块(TLIB 原有库名 - 目标块名);
4. 替换库中的目标模块(TLIB 原库名 - + 新目标块名);
5. 从原有库中抽取指定目标模块(TLIB 原库名 \* 抽出目标块名);
6. 列出一个库的内容(TLIB 库名,CON、或 TLIB 库名,PRN、或 TLIB 库名、列表文件名)。

### 43.1 语法

在 DOS 命令下键入 C>TLIB 后回车便显示

Syntax: TLIB libname [/C] [/E] commands, listfile

libname library file pathname

commands sequence of operations to be performed (optional)

listfile file name for listing file (optional)

A command is of the form: <symbol>modulename, where <symbol> is:

- + add modulename to the library
- remove modulename from the library
- \* extract modulename without removing it
- + or + - replace modulename in library
- \* or \* - extract modulename and remove it
- /C case-sensitive library
- /E create extended dictionary

Use @filepath to continue from file "filepath".

Use '&' at end of a line to continue onto the next line.

下面对此作出相应解释,引用的原文将标以符号■。

■Syntax: TLIB libname [/C] [/E] commands, listfile

语法: TLIB 库文件名 [/C] [/E] TLIB 命令,列表文件名

方括号内的内容表示是任选的。其中各项的内容解释如下：

■ **libname**      library file pathname

库文件名。TLIB 命令要求必须给出一个库名。库名不允许使用 DOS 匹配符 \* 或 ?。如果文件名未带扩展名,则 TLIB 自认为是 .LIB。建议文件扩展名都用 .LIB,因为为了识别库文件,像 TCC.EXE 和 TC.EXE 对项目管理时都要求这种库名的扩展名为 .LIB。

注意:如果你已给出了库名,而该库文件事实上在磁盘上并不存在,那么假定你现在在库名后面还加上一个动作符号

□+

则 TLIB 将创建这个库文件。

■ **commands sequence of operations to be performed (optional)**

【TLIB 的命令表】,它描述了 TLIB 程序需要完成的动作,由一系列操作组成,每个操作都由一个或两个动作符号和文件名或模块名组成。在动作符号、文件名和模块名之间可用空格,但不能在两个动作符号之间或一个名中间出现空格。

一个命令行中可以包含多个操作,它们只受到命令行总长度的限制(不能超出 127 个字符)。

各操作之间的排列顺序是无关紧要的,TLIB 始终按下列特定的顺序来执行操作:

- 优先执行所有抽取操作;
- 然后执行全部删除操作;
- 最后执行所有添加操作。
- 在替换一个模块时,先删除后添加。

如果只想查查库中的内容,则不要给出任何操作命令。

■ **listfile file name for listing file (optional)**

库内容的列表文件名。如未给出它,则不产生列表文件。其缺省扩展名是 .LST。若其名为 CON,则列表内容显示于屏幕;若为 PRN,则在打印机上打出。

注意:写文件名时其前面要有一个逗号。

■ **A command is of the form: <symbol>modulename, where <symbol> is:**

动作符号共有 5 种:

■ **+**      add modulename to the library

添加动作符号。TLIB 在库中添加指定的文件。如果文件本身也是一个库(.LIB),则该操作将由指定库中的所有模块添加到目标文件中。如果目标库中已存在一个要添加的模块,则 TLIB 显示相应信息并停止执行该操作。

■ **-**      remove modulename from the library

删除动作符号。TLIB 从库中删除指定的模块,如模块不存在,则显示相应出错信息。删除操作仅需要模块名(允许带驱动器名和扩展名)。注意:对模块名仅管可以包括驱动器名、路径、扩展名等,但 TLIB 不予理会。或者说,模块应和 TLIB 在一个目录里。

■ **\***      extract modulename without removing it

【抽取动作】符号。将库中相应模块拷贝到文件中的方式建立给定名的文件。如果该模块不存在,TLIB 给出信息并不建立文件;如果指定的文件已存在,则被覆盖。

■ **-+ or +-**      replace modulename in library

-+ 或 +-,是替换操作符号。用指定模块的新拷贝替换库中与指定模块同名的模块。操作过程一般是先删除后添加。注意:指定模块假定原名为 MY.OBJ,那么一是库中应有

MY.OBJ ;二是新模块名必须还是 MY.OBJ ,而且它是原 MY.OBJ 更新的结果。只有具备这两点才是正确的。例如,你用另一个模块 MYNEW.OBJ 经 DOS 操作换名后为 MY.OBJ ,那么想替换是不会成功的。

■ - \* or \* - extract modulename and remove it

- \* 或 \* - ,抽取并删除操作。先将指定模块拷贝到相应的文件中,然后从库中删除该文件。其操作过程是先抽取后删除。

■ /C case-sensitive library

它是一个区分模块内符号大小写的标志。使用它则区分,否则不区分。一般 TLIB 在查看符号时不区分大小写。如符号 lookup 和符号 LOOKUP 被认为是重复符号。因为 C 语言区别大小写,所以在想用大小写来区别新旧模块中的符号时,应该用 /C 选项。然而,由于你对别的库中是否采用大小写区分不清楚时,最好在用 TLINK.EXE 时也不用 /C 选项。本任选项不常用。

注意:当往库中添加一模块时,在库中所有的符号必须是不同的。因此,想往库中添加一个会引起重复符号的模块,TLIB 会显示相应的出错信息并不添加这个模块。

■ /E create extended dictionary

使用它则创建【扩展字典集】,即把扩展字典加到库文件中去,以便 TLINK.EXE 对这些库能快速连接。扩展字典以十分紧凑的形式包含了标准库文件字典中所没有的信息。

■ Use @filepath to continue from file "filepath".

当有大量的操作或要重复某些操作时,使用命令行将遇到困难,因为一行的可容字符数是有限制的(最多 128 个)。解决的方法是使用响应文件(或称【应答文件】)。

使用响应文件名时,其前面应加上前缀符号 @。

■ Use '&' at end of a line to continue onto the next line.

【响应文件】是一个文本文件,它可以包含所有或部分 TLIB 命令。响应文件的构成:

1. 一行以上的文本可组成一个响应文件,当需续行时可在本行末使用符号 &,表明下行为续行(注意:它和 TLINK.EXE 中提到的响应文件的续行方法不同,那里用 + 号续行);
2. 在响应文件中不必放入所有 TLIB 命令,未放入的命令可放在 DOS 命令行上;
3. 一个 DOS 命令行上可以使用多个响应文件。

## 43.2 例

例1 用 X1.OBJ、YY.OBJ 和 ZZZ.OBJ 建立一个库 MY.LIB,列表文件名为 LIST-MY.LST

TLIB MY + X1 + YY + ZZZ, LISTMY

例2 在 MY.LIB 库中拷贝替换模块 X1.OBJ,增加 AA.OBJ,并同时删去 ZZZ.OBJ

TLIB MY - X1 + AA - ZZZ

例3 从 MY.LIB 中抽取模块 X1 并到打印机

TLIB MY \* X1, PRN

例4 用响应文件建立一个新库。先用 WORDSTAR 或 EDLIN.COM 等软件编辑一个文本文件 ALPHA.RSP,内容为

```
+a.obj +b.obj +c.obj &
+d.obj
```

它就是一个响应文件。然后用

```
TLIB NEW @ALPHA.RSP
```

就生成一个库 NEW.LIB。

例 5 修改存储模式库,显示彩色汉字

像 2.13 汉字操作系统,把原 MS-DOS 的中断 INT 10H 改为中断 INT 78H,而将原中断 INT 10H 换成了自己的中断 INT 10H,所以常常要用 int86() 函数调用中断 INT 78H 才能设置中文显示方式下的全屏幕彩色背景。然而,也可以采用修改存储模式库的方法解决。

下面给出一种用 TLIB 修改小模式库 CS.LIB 的具体方法。

1. 将 C 盘上的 CS.LIB 拷贝到虚拟盘 E(或其它盘上也可以)。

```
C>COPY C:\TC\LIB\CS.LIB E:
```

2. 将 TLIB.EXE 文件拷贝到 E 盘上。

```
C>COPY C:\TC\TLB.EXE E:
```

3. 将 CS.LIB 中的模块 CRTINIT 从 CS.LIB 中抽出,并将它从 CS.LIB 中删除。操作完成后可以看到盘上有 CS.BAK、CS.LIB 和 CRTINIT.OBJ。

```
C>TLIB CS - *CRTINIT
```

4. 将 DEBUG.COM(或 DEBUG.EXE)拷贝到 E 盘。

```
E>COPY C:\DEBUG.COM
```

5. 使用 DEBUG 修改 CRTINIT.OBJ。

```
E>DEBUG CRTINIT.OBJ
```

```
-D 180
```

```
XXXX:0180S.....- /* TC 在调用 CONIO.H 中的函数时 */
```

```
XXXX:0190 -video..... /* TC 自动连接 -c0crtinit 函数 */
```

```
XXXX:01A0 -crtinitl... /* 该函数初始化 VIDEO 结构,通 */
```

```
XXXX:01B0 ...-directvideo. /* 过调用 -crtinit 函数完成初 */
```

```
XXXX:01C0 ..N.....-c0crt /* 始化工作。*/
```

```
XXXX:01D0 inito..5.....-
```

```
XXXX:01E0 Videoint@..4....
```

```
XXXX:01F0-turboCrt..
```

```
-E 282 /* 修改2个字节,将原指令 MOV AL,03 */
```

```
XXXX:0282 B0.90 03.90 /* 改成两条空指令:NOP NOP。实际 */
```

```
/* 修改显示模式 */
```

```
-W /* 存盘 */
```

```
-Q /* 退出 */
```

6. 把 CRTINIT 模块加到 CS.LIB 文件中去。

```
E>TLIB CS +CRTINIT
```

整个过程未在 TC 目录中进行,是为了避免意外操作引起麻烦。现在你可以将 TC 目录



上原有的 CS.LIB 更换为另一个文件名,然后将 E 盘上修改后的 CS.LIB 按原名拷入 TC 目录即可。

此后,你在小存储模式下编译的程序,如果用到 CONIO.H 中的函数,如设置前景或背景函数 textattr(),输出函数 cputs()、cprintf(),输入函数 cscanf()等,便可显示彩色汉字。此法可在配有 EGA/VGA 卡的西山汉字系统下(或方正汉卡)上使用,效果尚可以。在 CGA 卡上设置的背景为双重亮度。

对其它几种存储模式也可按同样的方法修改,此时 -E 282 应相应为

|            |        |
|------------|--------|
| 对 CM.LIB 为 | -E 288 |
| CL.LIB     | -E 28A |
| CH.LIB     | -E 28A |
| CC.LIB     | -E 284 |

### 43.3 注意事项

1. 使用 TLIB 修改已有库时,将自动产生该库的一个后备文件(文件基本名同库文件基本名,扩展名为 .BAK)。这个文件可删除,也可不删除,作为参考用。
2. 在 TLIB 操作中,文件名或 .OBJ 模块名均不允许用 DOS 的通配符(\* 或 ?),即必须用具体的名字(允许带路径名)。
3. 当用 TLIB 往指定文件中添加一个目标模块文件(简称【模块】)时,进入库内的文件一般只保留文件基本名,而不含驱动器名、路径或扩展名。
4. 对库中的模块直接改名是不可能的,而只能通过间接操作进行:先从库中抽取它,继而把它删除,最后把用新名创建的文件再添加到库中去。

### 43.4 可能出现的错误或警告

1. [s] already in LIB, not changea!  
s 早在 LIB 中,不能改变!
2. [s] not found in library  
在库中没有找到 s。
3. Arg list too big  
参数表太大。
4. Attempted to remove current directory  
企图取消当前目录。
5. Bad file number  
错误文件数。
6. Bad header in input LIB  
在输入 LIB 中,头部错误。
7. Could not write output  
不能写输出。
8. Exec format error  
执行格式错误。

9. File already exists  
文件早已存在。
10. Gross—device link  
交叉设备连接。
11. Invalid access code  
无效存取码。
12. Invalid argument  
无效参数。
13. Invalid data  
无效数据。
14. Invalid function number  
无效功能号。
15. Invalid memory block address  
无效的存储块地址。
16. Invalid environment  
无效环境。
17. Invalid format  
无效格式。
18. Math argument  
数字参数。
19. Memory arenatrashed  
内存垃圾。
20. Memory full  
内存满。
21. No more files  
没有更多文件。
22. No such device  
没有这样的设备。
23. No such file or directory  
没有这样的文件或目录。
24. Not enough memory  
无足够内存。
25. Not same device  
无同样设备。
26. Outout device is full  
输出盘满。
27. Path not found  
路径没有找到。
28. Permission denied  
存取被拒绝。
29. print scanf: floating point formats not linked  
打印 scanf 时发现浮点格式错而不能连接。

- 30. Result too large  
结果太大。
- 31. Too many open files  
打开文件太多。
- 32. Unknown error  
不知道的误差。
- 33. bad OMF record type 0x%x  
错误的 OMF 记录类型。
- 34. encountered record length %u  
记录长度冲突。
- 35. exceeds available buffer  
大于有效缓冲区。

## 第四十四章 目标模块交叉引用工具 OBJXREF.COM 程序

为了了解指定目录下的目标文件 (\*.OBJ, 模块) 或库文件 (\*.LIB) 中的信息, 可以使用 TC 附带的程序 OBJXREF.COM。它主要有两个作用, 一是报告文件中公用名的定义和对它们的引用; 另一个作用是报告目标模块定义的段的长度。

在 DOS 提示符下键入 C>OBJXREF 屏幕便显示如下有关 OBJXREF.COM 的帮助信息:

```
OBJXREF Version 2.0 Copyright (c) 1988 Borland International
Syntax: objxref [options] fileset [fileset ...]

Report format options:
 /RR * public refs and defs /RP public definitions
 /RM publics by module /RX externals by module
 /RS module sizes by segment /RC module sizes by class
 /RU unreferenced publics /RV verbose - all report formats
 (* = default)

Control options:
 /I ignore case /F include full libraries
 /V verbose output /Z show zero length segments
 /Nn selective report by name /On output file

Response file options:
 @n input from free format ASCII file (filename.ext)
 /Ln input from a linker response file (filename.ext)
 /Pn input from a TC project file (filename[.PRJ])
 /Dn search directories (separate with ',')

fileset (file names and extensions may contain wild cards):
 List of OBJs and Libs (default extensions is .OBJ)
```

下面用 OBJZMH.C 和 OBJZMH1.C 程序 (假定它们在集成环境下编译后均生成相应的 .OBJ 文件) 说明其使用方法。

```
C>TYPE OBJZMH.C
#include "stdio.h"
main()
{
 int k=1;
 printf("k=%d\n",k);
}
```

```
C>TYPE OBJZMH1.C
#include "math.h"
main()
{
 double k=100.8;
 printf("k=%f\n",k);
}
```

### 44.1 语法

在 DOS 提示符下键入

C>OBJXREF [Options] [fileset] [fileset...]

Options 可选项包括两方面的内容：报告格式选择项和控制选择项。每一选项间可用空格符隔开，它们之间的排列顺序可以任意的。

#### 一 报告格式选择项 (Report format options)

它分 8 种格式 (斜杠后面的字母大小写都可以)：

##### 1. /RC

只输出类内模块大小 (module sizes by class)

C>OBJXREF /RC /OQ1. OUT OBJZMH OBJZMH1 MATHS. LIB

C>TYPE Q1. OUT (为节省篇幅已略去不必要的空行,下同)

WARNING: symbol —main defined in OBJZMH duplicated in OBJZMH1

WARNING: Unresolved symbol FIDRQQ in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH

MODULES SIZES BY CLASS

CODE

|             |         |
|-------------|---------|
| 20 (00014h) | OBJZMH  |
| 52 (00034h) | OBJZMH1 |
| 72 (00048h) | total   |

DATA

|             |         |
|-------------|---------|
| 6 (00006h)  | OBJZMH  |
| 6 (00006h)  | OBJZMH1 |
| 12 (0000Ch) | total   |

Symbols = 18

Modules = 3

Segments = 2

Classes = 2

##### 2. /RM

只输出模块内公用名 (publics by module)

C>OBJXREF /RM /OQ2. OUT OBJZMH OBJZMH1 MATHS. LIB

C>TYPE Q2. OUT

WARNING: symbol —main defined in OBJZMH duplicated in OBJZMH1

WARNING: Unresolved symbol FIDRQQ in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH

PUBLIC SYMBOL DEFINITIONS BY MODULE NAME

MODULE: —undefined— defines the following symbols

FIDRQQ

—printf

—printf

MODULE: OBJZMH defines the following symbols

—main

MODULE: OBJZMH1 defines the following symbols

Symbols = 18  
Modules = 3

### 3. /RP

只输出公用名和模块定义情况 (public definitions)

```
C>OBJXREF /RP /OQ3. OUT OBJZMH OBJZMH1 MATHS. LIB
C>TYPE Q3. OUT
```

WARNING: symbol —main defined in OBJZMH duplicated in OBJZMH1

WARNING: Unresolved symbol FIDRQQ in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH

PUBLIC SYMBOL DEFINITIONS BY SYMBOL NAME

| SYMBOL  | DEFINED IN  |
|---------|-------------|
| FIDRQQ  | —undefined— |
| —main   | OBJZMH      |
| —printf | —undefined— |

Symbols = 18

Modules = 3

### 4. /RR (缺省格式)

只输出公共引用和定义 (refs and defs)

```
C>OBJXREF /RR /OQ4. OUT OBJZMH OBJZMH1 MATHS. LIB
C>TYPE Q4. OUT
```

WARNING: symbol —main defined in OBJZMH duplicated in OBJZMH1

WARNING: Unresolved symbol FIDRQQ in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH

PUBLIC SYMBOL DEFINITION AND REFERENCES BY SYMBOL NAME

FIDRQQ (—undefined—)

OBJZMH1

—printf (—undefined—)

OBJZMH

OBJZMH1

Symbols = 18

Modules = 3

### 5. /RS

只输出段内模块大小 (module sizes segment)

```
C>OBJXREF /RS /OQ5. OUT OBJZMH OBJZMH1 MATHS. LIB
C>TYPE Q5. OUT
```

WARNING: symbol —main defined in OBJZMH duplicated in OBJZMH1

WARNING: Unresolved symbol FIDRQQ in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH

MODULES SIZES BY SEGMENT

—DATA

|             |         |
|-------------|---------|
| 6 (00006h)  | OBJZMH  |
| 6 (00006h)  | OBJZMH1 |
| 12 (0000Ch) | total   |

—TEXT

|             |         |
|-------------|---------|
| 20 (00014h) | OBJZMH  |
| 52 (00034h) | OBJZMH1 |
| 72 (00048h) | total   |

Symbols = 18

Modules = 3

Segments = 2

Classes = 2

6. /RU

只输出未被引用的公共部分 (unreferenced publics)

C>OBJXREF /RU /OQ6. OUT OBJZMH OBJZMH1 MATHS. LIB

C>TYPE Q6. OUT

WARNING: symbol —main defined in OBJZMH duplicated in OBJZMH1

WARNING: Unresolved symbol FIDRQQ in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH

UNREFERENCED PUBLIC SYMBOLS BY MODULE NAME

MODULE: OBJZMH defines the following unreferenced symbols

—main

Symbols = 18

Modules = 3

7. /RV

输出所有报告格式 (verbose — all report formats)

C>OBJXREF /RV /OQ7. OUT OBJZMH OBJZMH1 MATHS. LIB

C>TYPE Q7. OUT

WARNING: symbol —main defined in OBJZMH duplicated in OBJZMH1

WARNING: Unresolved symbol FIDRQQ in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH

PUBLIC SYMBOL DEFINITIONS BY SYMBOL NAME

| SYMBOL | DEFINED IN |
|--------|------------|
|--------|------------|

|        |             |
|--------|-------------|
| FIDRQQ | —undefined— |
|--------|-------------|

|       |        |
|-------|--------|
| —main | OBJZMH |
|-------|--------|

|         |             |
|---------|-------------|
| —printf | —undefined— |
|---------|-------------|

PUBLIC SYMBOL DEFINITIONS BY MODULE NAME

MODULE: —undefined— defines the following symbols

FIDRQQ

—printf

—printf

MODULE: OBJZMH defines the following symbols

```

 --main
MODULE: OBJZMH1 defines the following symbols
EXTERNAL SYMBOL REFERENCES BY MODULE NAME
MODULE: --undefined-- references the following symbols
MODULE: OBJZMH references the following symbols
 --printf
MODULE: OBJZMH1 references the following symbols
 FIDRQQ
 --printf
PUBLIC SYMBOL DEFINITION AND REFERENCES BY SYMBOL NAME
FIDRQQ (--undefined--)
 OBJZMH1
--printf (--undefined--)
 OBJZMH
 OBJZMH1

```

#### MODULES SIZES BY SEGMENT

```

--DATA
 6 (00006h) OBJZMH
 6 (00006h) OBJZMH1
 12 (0000Ch) total

--TEXT
 20 (00014h) OBJZMH
 52 (00034h) OBJZMH1
 72 (00048h) total

```

#### MODULES SIZES BY CLASS

```

CODE
 20 (00014h) OBJZMH
 52 (00034h) OBJZMH1
 72 (00048h) total

DATA
 6 (00006h) OBJZMH
 6 (00006h) OBJZMH1
 12 (0000Ch) total

```

#### UNREFERENCED PUBLIC SYMBOLS BY MODULE NAME

```

MODULE: OBJZMH defines the following unreferenced symbols

```

```

 --main
Symbols = 18
Modules = 3
Segments = 2
Classes = 2

```

8. /RX

只输出模块外部引用部分 (externals by module)

```
C>OBJXREF /RX /OQ8.OUT OBJZMH OBJZMH1 MATHS.LIB
```

```
C>TYPE Q8.OUT
```

```
WARNING: symbol --main defined in OBJZMH duplicated in OBJZMH1
```



```

WARNING: Unresolved symbol FIDRQQ in module OBJZMH1
WARNING: Unresolved symbol —printf in module OBJZMH1
WARNING: Unresolved symbol —printf in module OBJZMH
EXTERNAL SYMBOL REFERENCES BY MODULE NAME
MODULE: —undefined— references the following symbols
MODULE: OBJZMH references the following symbols
 —printf
MODULE: OBJZMH1 references the following symbols
 FIDRQQ
 —printf
Symbols = 18
Modules = 3

```

注意: 选用格式中的字母大小写是不重要的, 下同。

## 二 控制选择项 (Control options)

它有 5 种格式 (斜杠后面的字母大小写都可以):

### 1. /F

包含整个库文件 (include full libraries) 中所有目标模块, 即使它们并不含有正被 OBJXREF 处理的目标模块所引用的公有名。这一功能对于了解 Turbo C 的库有帮助。

```
C>OBJXREF /F /RC /OQ9.OUT OBJZMH OBJZMH1 MATHS.LIB
```

```
C>TYPE Q9.OUT
```

```

WARNING: symbol —main defined in OBJZMH duplicated in OBJZMH1
WARNING: Unresolved symbol FIDRQQ in module —MATHERR (MATHS)
WARNING: Unresolved symbol FIWRQQ in module —MATHERR (MATHS)
WARNING: Unresolved symbol —errno in module —MATHERR (MATHS)
WARNING: Unresolved symbol —fprintf in module —MATHERR (MATHS)
WARNING: Unresolved symbol FIDRQQ in module MATHERR (MATHS)
WARNING: Unresolved symbol FIWRQQ in module MATHERR (MATHS)
WARNING: Unresolved symbol —streams in module —MATHERR (MATHS)
WARNING: Unresolved symbol FIDRQQ in module POW10 (MATHS)
WARNING: Unresolved symbol FIWRQQ in module XCVT (MATHS)
WARNING: Unresolved symbol FIDRQQ in module XCVT (MATHS)
WARNING: Unresolved symbol FIERQQ in module XCVT (MATHS)
WARNING: Unresolved symbol FIDRQQ in module TANH (MATHS)
WARNING: Unresolved symbol —errno in module LDTRUNC (MATHS)
WARNING: Unresolved symbol FIWRQQ in module LDTRUNC (MATHS)
WARNING: Unresolved symbol FIDRQQ in module LDTRUNC (MATHS)
WARNING: Unresolved symbol FIDRQQ in module SCANTOD (MATHS)
WARNING: Unresolved symbol —ctype in module SCANTOD (MATHS)
WARNING: Unresolved symbol FIWRQQ in module SCANTOD (MATHS)
WARNING: Unresolved symbol FIDRQQ in module SCANTOD (MATHS)
WARNING: Unresolved symbol —8087 in module TAN (MATHS)
WARNING: Unresolved symbol FIDRQQ in module TAN (MATHS)
WARNING: Unresolved symbol FIDRQQ in module TAN (MATHS)
WARNING: Unresolved symbol —scantod in module STRTOD (MATHS)

```

WARNING: Unresolved symbol FIDRQQ in module STRTOD (MATHS)  
 WARNING: Unresolved symbol —errno in module STRTOD (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module STRTOD (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module STRTOD (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module STAT87 (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module STAT87 (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module SQRT (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module SQRT (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module SINH (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module SINH (MATHS)  
 WARNING: Unresolved symbol —8087 in module SIN (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module SIN (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module SIN (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module LOG (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module LOG (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module LOG (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module EXP (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module EXP (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module POW (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module POW (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module POW (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module POLY (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module POLY (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module MODF (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module MODF (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module LOG10 (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module LOG10 (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module LOG10 (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module LDEXP (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module LDEXP (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module HYPOT (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module HYPOT (MATHS)  
 WARNING: Unresolved symbol —REALCVT in module GCVT (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module FTOL (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module FTOL (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module FREXP (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module FREXP (MATHS)  
 WARNING: Unresolved symbol —EMURESET in module FPRESET (MATHS)  
 WARNING: Unresolved symbol —exit in module FPERR (MATHS)  
 WARNING: Unresolved symbol —fprintf in module FPERR (MATHS)  
 WARNING: Unresolved symbol —streams in module FPERR (MATHS)  
 WARNING: Unresolved symbol —SignalPtr in module FPERR (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module FPEXCEP (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module FPEXCEP (MATHS)  
 WARNING: Unresolved symbol FIERQQ in module FPEXCEP (MATHS)  
 WARNING: Unresolved symbol emws—control in module FPEXCEP (MATHS)

WARNING: Unresolved symbol emwa—status in module FPEXCEP (MATHS)  
 WARNING: Unresolved symbol ——8087 in module FPEXCEP (MATHS)  
 WARNING: Unresolved symbol —exit in module FPEXCEP (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module FMOD (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module FMOD (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module FLOOR (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module FLOOR (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module FABS (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module FABS (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module CTRL87 (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module COSH (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module COSH (MATHS)  
 WARNING: Unresolved symbol ——8087 in module COS (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module COS (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module COS (MATHS)  
 WARNING: Unresolved symbol ——8087 in module CLEAR87 (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module CLEAR87 (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module CEIL (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module CEIL (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module ATAN2 (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module ATAN2 (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module ATAN (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module ACOSASIN (MATHS)  
 WARNING: Unresolved symbol FIWRQQ in module ACOSASIN (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module ACOSASIN (MATHS)  
 WARNING: Unresolved symbol FIDRQQ in module OBJZMH1  
 WARNING: Unresolved symbol —printf in module OBJZMH1  
 WARNING: Unresolved symbol —printf in module OBJZMH

#### MODULES SIZES BY CLASS

##### BSS

|             |                                |
|-------------|--------------------------------|
| 42 (0002Ah) | EFCVT (MATHS), uninitialized   |
| 2 (00002h)  | FLAGS87 (MATHS), uninitialized |
| 44 (0002Ch) | total                          |

##### CODE

|             |                  |
|-------------|------------------|
| 298(0012Ah) | ACOSASIN (MATHS) |
| 13 (0000Dh) | ATAN (MATHS)     |
| 171(000ABh) | ATAN2 (MATHS)    |
| 16 (00010h) | ATOF (MATHS)     |
| 48 (00030h) | CEIL (MATHS)     |
| 42 (0002Ah) | CLEAR87 (MATHS)  |
| 76 (0004Ch) | COS (MATHS)      |
| 113(00071h) | COSH (MATHS)     |
| 46 (0002Eh) | CTRL87 (MATHS)   |
| 93 (0005Dh) | EFCVT (MATHS)    |
| 110(0006Eh) | EXP (MATHS)      |
| 15 (0000Fh) | FABS (MATHS)     |

|               |                  |
|---------------|------------------|
| 48 (00030h)   | FLOOR (MATHS)    |
| 77 (0004Dh)   | FMOD (MATHS)     |
| 125 (0007Dh)  | FPERR (MATHS)    |
| 605 (0025Dh)  | FPEXCEP (MATHS)  |
| 5 (00005h)    | FPRESET (MATHS)  |
| 54 (00036h)   | FREXP (MATHS)    |
| 43 (0002Bh)   | FTOL (MATHS)     |
| 31 (0001Fh)   | GCVT (MATHS)     |
| 90 (0005Ah)   | HYPOT (MATHS)    |
| 124 (0007Ch)  | LDEXP (MATHS)    |
| 155 (0009Bh)  | LDTRUNC (MATHS)  |
| 149 (00095h)  | LOG (MATHS)      |
| 149 (00095h)  | LOG10 (MATHS)    |
| 40 (00028h)   | MATHERR (MATHS)  |
| 73 (00049h)   | MODF (MATHS)     |
| 20 (00014h)   | OBJZMH           |
| 52 (00034h)   | OBJZMH1          |
| 113 (00071h)  | POLY (MATHS)     |
| 403 (00193h)  | POW (MATHS)      |
| 200 (000C8h)  | POW10 (MATHS)    |
| 422 (001A6h)  | REALCVT (MATHS)  |
| 865 (00361h)  | SCANTOD (MATHS)  |
| 76 (0004Ch)   | SIN (MATHS)      |
| 161 (000A1h)  | SINH (MATHS)     |
| 62 (0003Eh)   | SQRT (MATHS)     |
| 22 (00016h)   | STAT87 (MATHS)   |
| 171 (000ABh)  | STRTOD (MATHS)   |
| 79 (0004Fh)   | TAN (MATHS)      |
| 119 (00077h)  | TANH (MATHS)     |
| 451 (001C3h)  | XCVT (MATHS)     |
| 160 (000A0h)  | —MATHERR (MATHS) |
| 6185 (01829h) | total            |

# DATA

|              |                  |
|--------------|------------------|
| 18 (00012h)  | ACOSASIN (MATHS) |
| 24 (00018h)  | ATAN2 (MATHS)    |
| 12 (0000Ch)  | COS (MATHS)      |
| 7 (00007h)   | COSH (MATHS)     |
| 12 (0000Ch)  | EXP (MATHS)      |
| 124 (0007Ch) | FPERR (MATHS)    |
| 129 (00081h) | FPEXCEP (MATHS)  |
| 2 (00002h)   | FREXP (MATHS)    |
| 36 (00024h)  | HUGEVAL (MATHS)  |
| 6 (00006h)   | HYPOT (MATHS)    |
| 14 (0000Eh)  | LDEXP (MATHS)    |
| 12 (0000Ch)  | LOG (MATHS)      |
| 14 (0000Eh)  | LOG10 (MATHS)    |

|             |                  |
|-------------|------------------|
| 8 (00008h)  | MATHERR (MATHS)  |
| 6 (00006h)  | OBJZMH           |
| 6 (00006h)  | OBJZMH1          |
| 5 (00005h)  | POLY (MATHS)     |
| 4 (00004h)  | POW (MATHS)      |
| 128(00080h) | POW10 (MATHS)    |
| 2 (00002h)  | REALCVT (MATHS)  |
| 26 (0001Ah) | SCANTOD (MATHS)  |
| 6 (00006h)  | SCANTOD (MATHS)  |
| 12 (0000Ch) | SIN (MATHS)      |
| 5 (00005h)  | SINH (MATHS)     |
| 13 (0000Dh) | SQRT (MATHS)     |
| 12 (0000Ch) | TAN (MATHS)      |
| 77 (0004Dh) | —MATHERR (MATHS) |
| 720(002D0h) | total            |

Symbols = 70

Modules = 46

Segments = 5

Classes = 3

## 2. /I

对公用名中忽略大小写的区别 (ignore case)。

## 3. /N 名

按名选择输出 (selective report by name)。它可用于报告指定的模块、段、类和公用名情况。注意：字母 N 后无空格。下例为只报告 DATA 类的情况。

```
C>OBJXREF OBJZMH OBJZMH1 MATHS.LIB /RV /NDATA /OQ10.OUT
```

```
C>TYPE Q10.OUT
```

WARNING: symbol —main defined in OBJZMH duplicated in OBJZMH1

WARNING: Unresolved symbol FIDRQQ in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH1

WARNING: Unresolved symbol —printf in module OBJZMH

PUBLIC SYMBOL DEFINITIONS BY SYMBOL NAME

SYMBOL DEFINED IN

PUBLIC SYMBOL DEFINITIONS BY MODULE NAME

EXTERNAL SYMBOL REFERENCES BY MODULE NAME

PUBLIC SYMBOL DEFINITION AND REFERENCES BY SYMBOL NAME

MODULES SIZES BY SEGMENT

MODULES SIZES BY CLASS

DATA

|            |         |
|------------|---------|
| 6 (00006h) | OBJZMH  |
| 6 (00006h) | OBJZMH1 |
| 12(0000Ch) | total   |

UNREFERENCED PUBLIC SYMBOLS BY MODULE NAME

Symbols = 18

Modules = 3

Segments = 2

Classes = 2

#### 4. /V

完整输出 (verbose output), 即列出读入的文件名, 显示整个公用名、模块名、段名和类等。

```
C>OBJXREF /V /RS /OQ11.OUT OBJZMH OBJZMH1 MATHS.LIB
```

```
C>TYPE Q11.OUT
```

```
Loading OBJZMH.OBJ
```

```
Loading OBJZMH1.OBJ
```

```
WARNING: symbol --main defined in OBJZMH duplicated in OBJZMH1
```

```
Loading MATHS.LIB
```

```
WARNING: Unresolved symbol FIDRQQ in module OBJZMH1
```

```
WARNING: Unresolved symbol --printf in module OBJZMH1
```

```
WARNING: Unresolved symbol --printf in module OBJZMH
```

```
MODULES SIZES BY SEGMENT
```

```
--DATA
```

|             |         |
|-------------|---------|
| 6 (00006h)  | OBJZMH  |
| 6 (00006h)  | OBJZMH1 |
| 12 (0000Ch) | total   |

```
--TEXT
```

|             |         |
|-------------|---------|
| 20 (00014h) | OBJZMH  |
| 52 (00034h) | OBJZMH1 |
| 72 (00048h) | total   |

```
Symbols = 18
```

```
Modules = 3
```

```
Segments = 2
```

```
Classes = 2
```

#### 5. /Z

显示长度为 0 段 (show zero length segments)。目标模块可以定义没有任何空间的段, 列出这些长度为 0 的段定义通常使模块长度报告更难以使用, 但如果要删除一个段的所有定义, 这是有价值的。

```
C>OBJXREF /Z /RS /OQ12.OUT OBJZMH OBJZMH1 MATHS.LIB
```

```
C>TYPE Q12.OUT
```

```
WARNING: symbol --main defined in OBJZMH duplicated in OBJZMH1
```

```
WARNING: Unresolved symbol FIDRQQ in module OBJZMH1
```

```
WARNING: Unresolved symbol --printf in module OBJZMH1
```

```
WARNING: Unresolved symbol --printf in module OBJZMH
```

```
MODULES SIZES BY SEGMENT
```

```
--BSS
```

|            |                        |
|------------|------------------------|
| 0 (00000h) | OBJZMH, uninitialized  |
| 0 (00000h) | OBJZMH1, uninitialized |
| 0 (00000h) | total                  |

```
--DATA
```

|            |        |
|------------|--------|
| 6 (00006h) | OBJZMH |
|------------|--------|

```

 6 (00006h) OBJZMH1
 12 (0000Ch) total
-TEXT
 20 (00014h) OBJZMH
 52 (00034h) OBJZMH1
 72 (00048h) total
Symbols = 18
Modules = 3
Segments = 3
Classes = 3

```

#### 6. 输出文件名 (report by name)

格式为 /O 文件名

文件名指输出文件名 (output file)。如未指定,输出结果到屏幕。注意,字母O后面无空格。

## 44.2 响应文件选择项 (Response file Options)

由于 DOS 的命令行上最多只能输入 128 个字符,因此当选择项和输入文件名的字符总数超出 127 时就应该用响应文件处理。

响应文件是一个文本文件,可以用集成编辑器编辑。

在命令行上可以指定多个响应文件,它们在命令行上的次序可在 .OBJ 或 .LIB 的前边或后边。

OBJXREF 能处理 4 种特殊格式的文件或目录。

#### 1. @ 响应文件名

响应文件也称【自由格式的 ASCII 码文件】,文件可带路径名和扩展名 (input from free format ASCII file(filename.ext))。例如:假定 MYFILE.REF 是一个自由格式的 ASCII 码文件:

```

C>TYPE MYFILE.REF
objzmlh /* 未写扩展名就是隐含 .OBJ */
OBJZMH1.OBJ
MATHS.LIB
C>OBJXREF /RC /OQ13.OUT @MYFILE.REF

```

则 Q13.OUT 的内容和 Q1.OUT 相同。注意:对于自由格式的 ASCII 码文件中的每一个单词,除有带像 .OBJ 或 .LIB 的外,均认为是一个目标文件 (\*.OBJ)。因此对标准的规划文件 (\*.PRJ) 应用格式 /P 处理,否则其括号里的包含文件等也会当目标文件处理(见下面叙述)。

#### 2. /D 目录名

搜索文件目录 (Search directories),有多个要搜索的目录时,多个目录间要用分号 (;) 隔开 (separate with ';')。假定文件 MYFILE.REF 中文件 MATHS.LIB 在目录 C:\TC\LIB 中,而 OBJZMH.OBJ 和 OBJZMH1.OBJ 在目录 E:\ 中,则可用格式

```
/DC:\TC\LIB;E:\
```

那么 OBJXREF 将先后在这两个目录中寻找。否则,未用 /D 选项时 OBJXREF 只在当前目录

中寻找,像例中情况便找不到,因为两个文件不在同一个目录下。注意:在书写各字符时不要用空格符。

注意:仅管只有一个文件在当前目录里,但只要你用了 /D 选项,则对在当前目录里的这个文件也应在 /D 后指出其所在目录。

### 3. /L 文件名

直接借用 TLINK.EXE 使用的连接器响应文件 (input from a linker response file (file-name.ext))。

```
C>TYPE MYFILE.ARF /* TLINK.EXE 用的响应文件 */
OBJZMH1
objzmh1.exe
objzmh1.map /* 在集成环境下编译时选用 O/L/Mapfile Detailed 生成 */
MATHS.LIB /* 连接响应文件应包括正确连接需要的所有.OBJ 和.LIB 文件 */
/* 本示例中没有这样做,以便让读者可看看其结果 */
C>OBJXREF /RV /LMYFILE.ARF /OQ14.OUT
C>Q14.OUT
WARNING: Unresolved symbol FIDRQQ in module OBJZMH1
WARNING: Unresolved symbol --printf in module OBJZMH1
PUBLIC SYMBOL DEFINITIONS BY SYMBOL NAME
SYMBOL DEFINED IN
FIDRQQ --undefined--
--main OBJZMH1
--printf --undefined--
PUBLIC SYMBOL DEFINITIONS BY MODULE NAME
MODULE: --undefined-- defines the following symbols
 FIDRQQ
 --printf
MODULE: OBJZMH1 defines the following symbols
 --main
EXTERNAL SYMBOL REFERENCES BY MODULE NAME
MODULE: --undefined-- references the following symbols
MODULE: OBJZMH1 references the following symbols
 FIDRQQ
 --printf
PUBLIC SYMBOL DEFINITION AND REFERENCES BY SYMBOL NAME
FIDRQQ (--undefined--)
 OBJZMH1
--printf (--undefined--)
 OBJZMH1
MODULES SIZES BY SEGMENT
--DATA
 6 (00006h) OBJZMH1
 6 (00006h) total
--TEXT
 52 (00034h) OBJZMH1
 52 (00034h) total
```



## MODULES SIZES BY CLASS

### CODE

```
52 (00034h) OBJZMH1
52 (00034h) total
```

### DATA

```
6 (00006h) OBJZMH1
6 (00006h) total
```

## UNREFERENCED PUBLIC SYMBOLS BY MODULE NAME

MODULE: OBJZMH1 defines the following unreferenced symbols

—main

Symbols = 18

Modules = 2

Segments = 2

Classes = 2

### 4. /P 规划文件名 (\*.PRJ)

该格式专用于标准规划文件 (input from a TC project file(filename.ext))。因为对规划文件处理一次就可对多个目标文件或库文件处理,有此专用格式就比较方便对现成的规划文件直接分析。注意:凡规划文件涉及的目标文件、库文件及头文件等都必须指定的目录内。

```
C>TYPE OBJZMH2.C C>TYPE OBJZMH3.C C>TYPE MYFILE.PRJ
```

```
main() #include "math.h" objznh2
{ obj() OBJZMH3
obj(): { MATHS.LIB
} double k=100.8;
 printf("k=%f\n",k);
 }
```

```
C>OBJXREF /RC /OQ15.OUT /PMYFILE.PRJ
```

```
C>TYPE Q15.OUT
```

WARNING: Unresolved symbol FIDRQQ in module OBJZMH3

WARNING: Unresolved symbol —printf in module OBJZMH3

## MODULES SIZES BY CLASS

### CODE

```
8 (00008h) OBJZMH2
52 (00034h) OBJZMH3
60 (0003Ch) total
```

### DATA

```
6 (00006h) OBJZMH3
6 (00006h) total
```

Symbols = 19

Modules = 3

Segments = 2

Classes = 2

## 44.3 输入文件名 (fileset)

它是指一组待处理的文件名 (fileset (file names and extensions may contain wild cards)),

多个文件名之间用空格或制表符隔开。文件名也可以用 DOS 的通配符 (\* 或 ?)。文件类型只限于目标文件 (\*.OBJ) 和库文件 (\*.LIB) 两种,如未加扩展名则自动认为是 .OBJ (List of OBJs and Libs(default extentions is .OBJ))。文件也可带路径名。此项也允许与响应文件选项同时使用,输出结果为两者处理后的内容。

OBJXREF 报告一般被输出到屏幕,也可以用 DOS 重定向符把报告送到打印机 (用 > LPT1;) 或一个磁盘文件 (用 > 磁盘文件名)。

## 44.4 OBJXREF 处理过程

主要包括 4 个方面:

1. 读 (Reading modules...)
2. 连接 (Linking modules...)
3. 对输出排序 (Sorting reports...)。从前面叙述的实例可见,这有利于阅读。
4. 写输出 (Writing reports...)

## 44.5 可能出现的警告或错误

(方括号中内容视具体情况由 OBJXREF 自动填写)

1. WARNING: Symbol [s1] defined in [s2] duplicated in [s3]  
在 s2 中定义的符号 s1 和 s3 中的相同,OBJXREF 将忽略 s3 中的定义。
2. WARNING: No files matching [s]  
无文件匹配 s。在命令行上或响应文件中命名为 s 的文件不存在或打开失败,OBJXREF 将继续处理下一个文件。
3. WARNING: Invalid file specification [s]  
无效的文件 s,OBJXREF 继续处理下一个文件。
4. WARNING: File not found [s]  
没有找到文件 s。
5. WARNING: Unable to open input file [s]  
不能打开输入文件 s (s 可能不存在),OBJXREF 继续处理下一个文件。
6. WARNING: Unable to open output file [s]  
不能打开输出文件 s。
7. WARNING: Unresolved symbol [s1] in module [s2]  
在模块 s2 中引用的符号 s1 没有在指定的 .OBJ 或 .LIB 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。
8. WARNING: Unknown option - [s]  
不认识的选择项 s,OBJXREF 将忽略此选择项。
9. Modules defines more than [s] names  
模块定义超过 s 个名。
10. Out of memory  
OBJXREF 在 RAM 内执行交叉引用时发现内存不够。

# 附录

表 0-1 库函数与宏

| 序  | 所在 | 库函数或宏名        | 原型或定义的标头文件            |
|----|----|---------------|-----------------------|
| 1  | 32 | abort         | process. h, stdlib. h |
| 2  | 26 | _____abs_____ |                       |
| 3  | 26 | abs           | math. h, stdlib. h    |
| 4  | 20 | absread       | dos. h                |
| 5  | 20 | abswrite      | dos. h                |
| 6  | 30 | access        | io. h                 |
| 7  | 26 | acos          | math. h               |
| 8  | 25 | allocmem      | dos. h                |
| 9  | 35 | arc           | graphics. h           |
| 10 | 28 | asctime       | time. h               |
| 11 | 26 | asin          | math. h               |
| 12 | 32 | assert        | assert. h             |
| 13 | 26 | atan          | math. h               |
| 14 | 26 | atan2         | math. h               |
| 15 | 32 | atexit        | stdlib. h             |
| 16 | 26 | atof          | math. h, stdlib. h    |
| 17 | 26 | atoi          | stdlib. h             |
| 18 | 26 | atol          | stdlib. h             |
| 19 | 35 | bar           | graphics. h           |
| 20 | 35 | bar3d         | graphics. h           |
| 21 | 22 | bdos          | dos. h                |
| 22 | 22 | bdosptr       | dos. h                |
| 23 | 23 | biosecom      | bios. h               |
| 24 | 20 | biosdisk      | bios. h               |
| 25 | 22 | biosequip     | bios. h               |
| 26 | 33 | bioskey       | bios. h               |
| 27 | 22 | biosmemory    | bios. h               |
| 28 | 34 | boisprint     | bios. h               |
| 29 | 28 | biostime      | bios. h               |
| 30 | 25 | brk           | alloc. h              |
| 31 | 37 | bsearch       | stdlib. h             |
| 32 | 26 | cabs          | math. h               |
| 33 | 25 | calloc        | alloc. h, stdlib. h   |
| 34 | 26 | ceil          | math. h               |
| 35 | 30 | cgets         | conio. h              |
| 36 | 29 | chdir         | dir. h                |
| 37 | 30 | __chmod       | io. h                 |
| 38 | 30 | chmod         | io. h                 |
| 39 | 30 | chsize        | io. h                 |
| 40 | 35 | circle        | graphics. h           |
| 41 | 27 | __clear87     | float. h              |

102 135  
 0.44 0.38 + 3.38  
 428 626

|    |    |               |             |
|----|----|---------------|-------------|
| 42 | 35 | cleardevice   | graphics. h |
| 43 | 30 | clearerr      | stdio. h    |
| 44 | 35 | clearviewport | graphics. h |
| 45 | 22 | __cli__       | dos. h      |
| 46 | 30 | __close       | io. h       |
| 47 | 30 | close         | io. h       |
| 48 | 35 | closegraph    | graphics. h |
| 49 | 28 | clock         | time. h     |
| 50 | 35 | clreol        | conio. h    |
| 51 | 35 | clrscr        | conio. h    |
| 52 | 27 | __control87   | float. h    |
| 53 | 25 | coreleft      | alloc. h    |
| 54 | 26 | cos           | math. h     |
| 55 | 26 | cosh          | math. h     |
| 56 | 28 | country       | dos. h      |
| 57 | 31 | cprintf       | conio. h    |
| 58 | 30 | cputs         | conio. h    |
| 59 | 30 | __creat       | io. h       |
| 60 | 30 | creat         | io. h       |
| 61 | 30 | creatnew      | io. h       |
| 62 | 30 | creattemp     | io. h       |
| 63 | 31 | cscanf        | conio. h    |
| 64 | 28 | ctime         | time. h     |
| 65 | 22 | ctrlbrk       | dos. h      |
| 66 | 28 | delay         | dos. h      |
| 67 | 35 | delline       | conio. h    |
| 68 | 35 | detectgraph   | graphics. h |
| 69 | 28 | difftime      | time. h     |
| 70 | 22 | disable       | dos. h      |
| 71 | 26 | div           | stdlib. h   |
| 72 | 18 | dosexterr     | dos. h      |
| 73 | 28 | dosounix      | dos. h      |
| 74 | 35 | drawpoly      | graphics. h |
| 75 | 30 | dup           | io. h       |
| 76 | 30 | dup2          | io. h       |
| 77 | 26 | ecvt          | stdlib. h   |
| 78 | 35 | ellipse       | graphics. h |
| 79 | 22 | __emit        | dos. h      |
| 80 | 22 | enable        | dos. h      |
| 81 | 30 | eof           | io. h       |
| 82 | 32 | execl         | process. h  |
| 83 | 32 | execle        | process. h  |
| 84 | 32 | execlp        | process. h  |
| 85 | 32 | execlpe       | process. h  |
| 86 | 32 | execv         | process. h  |

|     |    |             |                       |
|-----|----|-------------|-----------------------|
| 87  | 32 | execve      | process. h            |
| 88  | 32 | execvp      | process. h            |
| 89  | 32 | execvpe     | process. h            |
| 90  | 32 | __exit      | process. h, stdlib. h |
| 91  | 32 | exit        | process. h, stdlib. h |
| 92  | 26 | exp         | math. h               |
| 93  | 26 | fabs        | math. h               |
| 94  | 25 | farcalloc   | alloc. h              |
| 95  | 25 | farcoreleft | alloc. h              |
| 96  | 25 | farfree     | alloc. h              |
| 97  | 25 | farmalloc   | alloc. h              |
| 98  | 25 | farrealloc  | alloc. h              |
| 99  | 30 | fclose      | stdio. h              |
| 100 | 30 | fcloseall   | stdio. h              |
| 101 | 26 | fcvt        | stdlib. h             |
| 102 | 30 | fdopen      | stdio. h              |
| 103 | 30 | feof        | stdio. h              |
| 104 | 30 | ferror      | stdio. h              |
| 105 | 30 | fflush      | stdio. h              |
| 106 | 30 | __fgetc     | stdio. h              |
| 107 | 30 | fgetc       | stdio. h              |
| 108 | 30 | fgetchar    | stdio. h              |
| 109 | 30 | fgetpos     | stdio. h              |
| 110 | 30 | fgets       | stdio. h              |
| 111 | 30 | filelength  | io. h                 |
| 112 | 30 | fileno      | stdio. h              |
| 113 | 35 | fillellipse | graphics. h           |
| 114 | 35 | fillpoly    | graphics. h           |
| 115 | 29 | findfirst   | dir. h                |
| 116 | 29 | findnext    | dir. h                |
| 117 | 35 | floodfill   | graphics. h           |
| 118 | 26 | floor       | math. h               |
| 119 | 30 | flushall    | stdio. h              |
| 120 | 26 | fmod        | math. h               |
| 121 | 29 | fnmerge     | dir. h                |
| 122 | 29 | fnsplit     | dir. h                |
| 123 | 30 | fopen       | stdio. h              |
| 124 | 27 | __fpreset   | float. h              |
| 125 | 8  | FP_OFF      | dos. h                |
| 126 | 8  | FP_SEG      | dos. h                |
| 127 | 31 | fprintf     | stdio. h              |
| 128 | 30 | __fputc     | stdio. h              |
| 129 | 30 | fputc       | stdio. h              |
| 130 | 30 | fputchar    | stdio. h              |
| 131 | 30 | fputs       | stdio. h              |

|     |    |                   |                    |
|-----|----|-------------------|--------------------|
| 132 | 30 | fread             | stdio. h           |
| 133 | 25 | free              | alloc. h, stdio. h |
| 134 | 25 | freemem           | dos. h             |
| 135 | 30 | freopen           | stdio. h           |
| 136 | 26 | frexp             | math. h            |
| 137 | 31 | fscanf            | stdio. h           |
| 138 | 30 | fseek             | stdio. h           |
| 139 | 30 | fsetpos           | stdio. h           |
| 140 | 30 | fstat             | stat. h            |
| 141 | 30 | ftell             | stdio. h           |
| 142 | 28 | ftime             | timeb. h           |
| 143 | 30 | fwrite            | stdio. h           |
| 144 | 26 | gcvt              | stdlib. h          |
| 145 | 22 | geninterrupt      | dos. h             |
| 146 | 35 | getarccoords      | graphics. h        |
| 147 | 35 | getaspectratio    | graphics. h        |
| 148 | 35 | getbkcolor        | graphics. h        |
| 149 | 30 | getc              | stdio. h           |
| 150 | 22 | getcbk            | dos. h             |
| 151 | 33 | getch             | conio. h           |
| 152 | 30 | getchar           | stdio. h           |
| 153 | 33 | getche            | conio. h           |
| 154 | 35 | getcolor          | graphics. h        |
| 155 | 29 | getcurdir         | dir. h             |
| 156 | 29 | getcwd            | dir. h             |
| 157 | 28 | getdate           | dos. h             |
| 158 | 35 | getdefaultpalette | graphics. h        |
| 159 | 20 | getdfree          | dos. h             |
| 160 | 29 | getdisk           | dir. h             |
| 161 | 35 | getdrivername     | graphics. h        |
| 162 | 20 | getdta            | dos. h             |
| 163 | 14 | getenv            | stdlib. h          |
| 164 | 20 | getfat            | dos. h             |
| 165 | 20 | getfard           | dos. h             |
| 166 | 35 | getfillpattern    | graphics. h        |
| 167 | 35 | getfillsettings   | graphics. h        |
| 168 | 28 | getftime          | io. h              |
| 169 | 35 | getgraphmode      | graphics. h        |
| 170 | 35 | getimage          | graphics. h        |
| 171 | 35 | getlineettings    | graphics. h        |
| 172 | 35 | getmaxcolor       | graphics. h        |
| 173 | 35 | getmaxmode        | graphics. h        |
| 174 | 35 | getmaxx           | graphics. h        |
| 175 | 35 | getmaxy           | graphics. h        |
| 176 | 35 | getmodename       | graphics. h        |

|     |    |                   |             |
|-----|----|-------------------|-------------|
| 177 | 35 | getmoderange      | graphics. h |
| 178 | 35 | getpalette        | graphics. h |
| 179 | 35 | getpalettesize    | graphics. h |
| 180 | 30 | getpass           | conio. h    |
| 181 | 35 | getpixel          | graphics. h |
| 182 | 21 | getpsp            | dos. h      |
| 183 | 30 | gets              | stdio. h    |
| 184 | 22 | getswitchar       | dos. h      |
| 185 | 35 | gettext           | conio. h    |
| 186 | 35 | gettextinfo       | conio. h    |
| 187 | 35 | gettextsettings   | graphics. h |
| 188 | 28 | gettime           | dos. h      |
| 189 | 22 | getvect           | dos. h      |
| 190 | 22 | getverify         | dos. h      |
| 191 | 35 | getviewsettings   | graphics. h |
| 192 | 30 | getw              | stdio. h    |
| 193 | 35 | getx              | graphics. h |
| 194 | 35 | gety              | graphics. h |
| 195 | 28 | gmtime            | time. h     |
| 196 | 35 | gotoxy            | conio. h    |
| 197 | 35 | graphdefaults     | graphics. h |
| 198 | 35 | grapherrormsg     | graphics. h |
| 199 | 35 | __graphfreemem    | graphics. h |
| 200 | 35 | __graphgetmem     | graphics. h |
| 201 | 35 | graphresult       | graphics. h |
| 202 | 18 | harderr           | dos. h      |
| 203 | 18 | hardresume        | dos. h      |
| 204 | 18 | hardretn          | dos. h      |
| 205 | 35 | highvideo         | conio. h    |
| 206 | 26 | hypot             | math. h     |
| 207 | 35 | imagesize         | graphics. h |
| 208 | 35 | initgraph         | graphics. h |
| 209 | 22 | inp               | dos. h      |
| 210 | 22 | inport            | dos. h      |
| 211 | 22 | __inportb__       | dos. h      |
| 212 | 22 | inportb           | dos. h      |
| 213 | 35 | inline            | conio. h    |
| 214 | 35 | installuserdriver | graphics. h |
| 215 | 35 | installuserfont   | graphics. h |
| 216 | 22 | __int__           | dos. h      |
| 217 | 22 | int86             | dos. h      |
| 218 | 22 | int86x            | dos. h      |
| 219 | 22 | intdos            | dos. h      |
| 220 | 22 | intdosx           | dos. h      |
| 221 | 22 | intr              | dos. h      |

|     |    |           |                     |
|-----|----|-----------|---------------------|
| 222 | 30 | ioctl     | io. h               |
| 223 | 7  | isalnum   | ctype. h            |
| 224 | 7  | isalpha   | ctype. h            |
| 225 | 7  | isascii   | ctype. h            |
| 226 | 30 | isatty    | io. h               |
| 227 | 7  | isctrl    | ctype. h            |
| 228 | 7  | isdigit   | ctype. h            |
| 229 | 7  | isgraph   | ctype. h            |
| 230 | 7  | islower   | ctype. h            |
| 231 | 7  | isprint   | ctype. h            |
| 232 | 7  | ispunct   | ctype. h            |
| 233 | 7  | isspace   | ctype. h            |
| 234 | 7  | isupper   | ctype. h            |
| 235 | 7  | isxdigit  | ctype. h            |
| 236 | 26 | itoa      | stdlib. h           |
| 237 | 33 | kbhit     | conio. h            |
| 238 | 22 | keep      | dos. h              |
| 239 | 26 | labs      | math. h, stdlib. h  |
| 240 | 26 | ldexp     | math. h             |
| 241 | 26 | ldiv      | stdlib. h           |
| 242 | 37 | lfind     | stdlib. h           |
| 243 | 35 | line      | graphics. h         |
| 244 | 35 | linerel   | graphics. h         |
| 245 | 35 | lineto    | graphics. h         |
| 246 | 28 | localtime | time. h             |
| 247 | 30 | lock      | io. h               |
| 248 | 26 | log       | math. h             |
| 249 | 26 | log10     | math. h             |
| 250 | 32 | longjmp   | setjmp. h           |
| 251 | 35 | lowvideo  | conio. h            |
| 252 | 10 | __lrotl   | stdlib. h           |
| 253 | 10 | __lrotr   | stdlib. h           |
| 254 | 37 | lsearch   | stdlib. h           |
| 255 | 30 | lseek     | io. h               |
| 256 | 26 | ltoa      | stdlib. h           |
| 257 | 25 | malloc    | alloc. h, stdlib. h |
| 258 | 26 | __matherr | math. h             |
| 259 | 26 | matherr   | math. h             |
| 260 | 26 | max       | stdlib. h           |
| 261 | 24 | memccpy   | mem. h, string. h   |
| 262 | 24 | memchr    | mem. h, string. h   |
| 263 | 24 | memcmp    | mem. h, string. h   |
| 264 | 24 | memcpy    | mem. h, string. h   |
| 265 | 24 | memicmp   | mem. h, string. h   |
| 266 | 24 | memmove   | mem. h, string. h   |



|     |    |              |                   |
|-----|----|--------------|-------------------|
| 267 | 24 | memset       | mem. h, string. h |
| 268 | 26 | min          | stdlib. h         |
| 269 | 29 | mkdir        | dir. h            |
| 270 | 8  | MK_FP        | dos. h            |
| 271 | 29 | mktemp       | dir. h            |
| 272 | 26 | modf         | math. h           |
| 273 | 24 | movedata     | mem. h, string. h |
| 274 | 35 | moverel      | graphics. h       |
| 275 | 35 | movetext     | conio. h          |
| 276 | 35 | moveto       | graphics. h       |
| 277 | 24 | movmem       | mem. h            |
| 278 | 35 | normvideo    | conio. h          |
| 279 | 36 | nosound      | dos. h            |
| 280 | 30 | __open       | io. h             |
| 281 | 30 | open         | io. h             |
| 282 | 22 | outp         | dos. h            |
| 283 | 22 | outport      | dos. h            |
| 284 | 22 | __outportb__ | dos. h            |
| 285 | 22 | outportb     | dos. h            |
| 286 | 35 | outtext      | graphics. h       |
| 287 | 35 | outtextxy    | graphics. h       |
| 288 | 30 | parsfnm      | dos. h            |
| 289 | 22 | peek         | dos. h            |
| 290 | 22 | peekb        | dos. h            |
| 291 | 18 | perror       | stdio. h          |
| 292 | 35 | pieslice     | graphics. h       |
| 293 | 22 | poke         | dos. h            |
| 294 | 22 | pokeb        | dos. h            |
| 295 | 26 | poly         | math. h           |
| 296 | 26 | pow          | math. h           |
| 297 | 26 | pow10        | math. h           |
| 298 | 31 | printf       | stdio. h          |
| 299 | 30 | putc         | stdio. h          |
| 300 | 30 | putch        | conio. h          |
| 301 | 30 | putchar      | stdio. h          |
| 302 | 14 | putenv       | stdlib. h         |
| 303 | 35 | putimage     | graphics. h       |
| 304 | 35 | putpixel     | graphics. h       |
| 305 | 30 | puts         | stdio. h          |
| 306 | 35 | puttext      | conio. h          |
| 307 | 30 | putw         | stdio. h          |
| 308 | 37 | qsort        | stdlib. h         |
| 309 | 38 | raise        | signal. h         |
| 310 | 26 | rand         | stdlib. h         |
| 311 | 20 | randbrd      | dos. h            |

|     |    |                      |                     |
|-----|----|----------------------|---------------------|
| 312 | 20 | randbwr              | dos. h              |
| 313 | 26 | random               | stdlib. h           |
| 314 | 26 | randomize            | stdlib. h           |
| 315 | 30 | __read               | io. h               |
| 316 | 30 | read                 | io. h               |
| 317 | 25 | realloc              | alloc. h, stdlib. h |
| 318 | 35 | rectangle            | graphics. h         |
| 319 | 35 | registerbgidriver    | graphics. h         |
| 320 | 35 | registerbgifont      | graphics. h         |
| 321 | 35 | registerfarbgidriver | graphics. h         |
| 322 | 35 | registerfarbgifont   | graphics. h         |
| 323 | 30 | remove               | stdio. h            |
| 324 | 30 | rename               | stdio. h            |
| 325 | 35 | restorecrtmode       | graphics. h         |
| 326 | 30 | rewind               | stdio. h            |
| 327 | 29 | rmdir                | dir. h              |
| 328 | 10 | __rotl               | stdlib. h           |
| 329 | 10 | __rotr               | stdlib. h           |
| 330 | 25 | sbrk                 | alloc. h            |
| 331 | 31 | scanf                | stdio. h            |
| 332 | 29 | searchpath           | dir. h              |
| 333 | 35 | sector               | graphics. h         |
| 334 | 22 | segreadd             | dos. h              |
| 335 | 35 | setactivepage        | graphics. h         |
| 336 | 35 | setallpalette        | graphics. h         |
| 337 | 35 | setaspectratio       | graphics. h         |
| 338 | 35 | setbkcolor           | graphics. h         |
| 339 | 25 | setblock             | dos. h              |
| 340 | 30 | setbuf               | stdio. h            |
| 341 | 22 | setcbrk              | dos. h              |
| 342 | 35 | setcolor             | graphics. h         |
| 343 | 28 | setdate              | dos. h              |
| 344 | 29 | setdisk              | dir. h              |
| 345 | 20 | setdta               | dos. h              |
| 346 | 35 | setfillpattern       | graphics. h         |
| 347 | 35 | setfillstyle         | graphics. h         |
| 348 | 28 | setftime             | io. h               |
| 349 | 35 | setgraphbufsize      | graphics. h         |
| 350 | 35 | setgraphmode         | graphics. h         |
| 351 | 32 | setjmp               | setjmp. h           |
| 352 | 35 | setlinestyle         | graphics. h         |
| 353 | 24 | setmem               | mem. h              |
| 354 | 30 | setmode              | io. h               |
| 355 | 35 | setpalette           | graphics. h         |
| 356 | 35 | setrgbpalette        | graphics. h         |

|     |    |                 |             |
|-----|----|-----------------|-------------|
| 357 | 35 | settextjustify  | graphics. h |
| 358 | 22 | setswitchchar   | dos. h      |
| 359 | 35 | settextstyle    | graphics. h |
| 360 | 28 | settime         | dos. h      |
| 361 | 35 | setusercharsize | graphics. h |
| 362 | 30 | setvbuf         | stdio. h    |
| 363 | 22 | setvect         | dos. h      |
| 364 | 22 | setverify       | dos. h      |
| 365 | 35 | setviewport     | graphics. h |
| 366 | 35 | setvisualpage   | graphics. h |
| 367 | 35 | setwritemode    | graphics. h |
| 368 | 38 | signal          | signal. h   |
| 369 | 26 | sin             | math. h     |
| 370 | 26 | sinh            | math. h     |
| 371 | 28 | sleep           | dos. h      |
| 372 | 30 | sopen           | io. h       |
| 373 | 36 | sound           | dos. h      |
| 374 | 32 | spawnl          | process. h  |
| 375 | 32 | spawnle         | process. h  |
| 376 | 32 | spawnlp         | process. h  |
| 377 | 32 | spawnlpe        | process. h  |
| 378 | 32 | spawnv          | process. h  |
| 379 | 32 | spawnve         | process. h  |
| 380 | 32 | spawnvp         | process. h  |
| 381 | 32 | spawnvpe        | process. h  |
| 382 | 31 | sprintf         | stdio. h    |
| 383 | 26 | sqrt            | math. h     |
| 384 | 26 | srand           | stdlib. h   |
| 385 | 31 | sscanf          | stdio. h    |
| 386 | 30 | stat            | stat. h     |
| 387 | 27 | __status87      | float. h    |
| 388 | 22 | __sti__         | dos. h      |
| 389 | 28 | stime           | time. h     |
| 390 | 7  | strcpy          | string. h   |
| 391 | 7  | strcat          | string. h   |
| 392 | 7  | strchr          | string. h   |
| 393 | 7  | strcmp          | string. h   |
| 394 | 7  | strcmpi         | string. h   |
| 395 | 7  | strncpy         | string. h   |
| 396 | 7  | strncpy         | string. h   |
| 397 | 7  | strdup          | string. h   |
| 398 | 18 | __strerror      | string. h   |
| 399 | 18 | strerror        | string. h   |
| 400 | 7  | strcmp          | string. h   |
| 401 | 7  | strlen          | string. h   |

|     |    |                |                         |
|-----|----|----------------|-------------------------|
| 402 | 7  | strlwr         | string. h               |
| 403 | 7  | strncat        | string. h               |
| 404 | 7  | strncmp        | string. h               |
| 405 | 7  | strncpy        | string. h               |
| 406 | 7  | strnicmp       | string. h               |
| 407 | 7  | strncmpi       | string. h               |
| 408 | 7  | strnset        | string. h               |
| 409 | 7  | strpbrk        | string. h               |
| 410 | 7  | strrchr        | string. h               |
| 411 | 7  | strrev         | string. h               |
| 412 | 7  | strset         | string. h               |
| 413 | 7  | strspn         | string. h               |
| 414 | 7  | strstr         | string. h               |
| 415 | 26 | strtod         | stdlib. h               |
| 416 | 26 | strtol         | stdlib. h               |
| 417 | 7  | strtok         | string. h               |
| 418 | 26 | strtoul        | stdlib. h               |
| 419 | 7  | strupr         | string. h               |
| 420 | 7  | swab           | stdlib. h               |
| 421 | 32 | system         | process. h, stdlib. h   |
| 422 | 26 | tan            | math. h                 |
| 423 | 26 | tanh           | math. h                 |
| 424 | 30 | tell           | io. h                   |
| 425 | 35 | textattr       | conio. h                |
| 426 | 35 | textbackground | conio. h                |
| 427 | 35 | textcolor      | conio. h                |
| 428 | 35 | textheight     | graphics. h             |
| 429 | 35 | textmode       | conio. h                |
| 430 | 35 | textwidth      | graphics. h             |
| 431 | 28 | time           | time. h                 |
| 432 | 30 | tmpfile        | stdio. h                |
| 433 | 30 | tmpnam         | stdio. h                |
| 434 | 7  | toascii        | ctype. h                |
| 435 | 7  | _tolower       | ctype. h                |
| 436 | 7  | tolower        | ctype. h                |
| 437 | 7  | _toupper       | ctype. h                |
| 438 | 7  | toupper        | ctype. h                |
| 439 | 28 | tzset          | time. h                 |
| 440 | 26 | ultoa          | stdlib. h               |
| 441 | 30 | umask          | io. h                   |
| 442 | 30 | ungetc         | stdio. h                |
| 443 | 33 | ungetch        | conio. h                |
| 444 | 28 | unixtodos      | dos. h                  |
| 445 | 30 | unlink         | dos. h, io. h, stdio. h |
| 446 | 30 | unlock         | io. h                   |

|     |    |          |          |
|-----|----|----------|----------|
| 447 | 12 | va_arg   | stdarg.h |
| 448 | 12 | va_end   | stdarg.h |
| 449 | 12 | va_start | stdarg.h |
| 450 | 31 | vfprintf | stdio.h  |
| 451 | 31 | vfscanf  | stdio.h  |
| 452 | 31 | vprintf  | stdio.h  |
| 453 | 31 | vscanf   | stdio.h  |
| 454 | 31 | vsprintf | stdio.h  |
| 455 | 31 | vsscanf  | stdio.h  |
| 456 | 35 | wherex   | conio.h  |
| 457 | 35 | wherey   | conio.h  |
| 458 | 35 | window   | conio.h  |
| 459 | 30 | _write   | io.h     |
| 460 | 30 | write    | io.h     |

表 0-2 结构与联合

| 序  | 所在章 | 结构或联合            | 定义的标头文件     |
|----|-----|------------------|-------------|
| 1  | 35  | arccoordstype    | graphics. h |
| 2  | 22  | BYTEREGS         | dos. h      |
| 3  | 26  | complex          | math. h     |
| 4  | 28  | country          | dos. h      |
| 5  | 28  | date             | dos. h      |
| 6  | 30  | devhdr           | dos. h      |
| 7  | 20  | dfree            | dos. h      |
| 8  | 26  | div _t           | stdlib. h   |
| 9  | 18  | DOSERROR         | dos. h      |
| 10 | 29  | dosSearchInfo    | dos. h      |
| 11 | 26  | exception        | math. h     |
| 12 | 20  | fatinfo          | dos. h      |
| 13 | 20  | lcb              | dos. h      |
| 14 | 29  | ffblk            | dir. h      |
| 15 | 30  | FILE             | stdio. h    |
| 16 | 35  | fillsettingstype | graphics. h |
| 17 | 28  | ftime            | io. h       |
| 18 | 32  | jmp _buf[1]      | setjmp. h   |
| 19 | 26  | ldiv _t          | stdlib. h   |
| 20 | 35  | linesettingstype | graphics. h |
| 21 | 35  | palettetype      | graphics. h |
| 22 | 35  | pointtype        | graphics. h |
| 23 | 22  | REGPACK          | dos. h      |
| 24 | 30  | stat             | stat. h     |
| 25 | 22  | SREGS            | dos. h      |
| 26 | 35  | text _info       | conio. h    |
| 27 | 35  | textsettingstype | graphics. h |
| 28 | 28  | time             | dos. h      |
| 29 | 28  | timeb            | timeb. h    |
| 30 | 28  | tm               | time. h     |
| 31 | 22  | WORDREGS         | dos. h      |
| 32 | 35  | viewporttype     | graphics. h |
| 33 | 20  | xfcb             | dos. h      |
| 34 | 22  | REGS             | dos. h      |

表 0-3 枚

| 序  | 所在页 | 定义的标头文件     |
|----|-----|-------------|
| 1  | 35  | graphics. h |
| 2  | 35  | graphics. h |
| 3  | 35  | graphics. h |
| 4  | 35  | graphics. h |
| 5  | 35  | graphics. h |
| 6  | 35  | graphics. h |
| 7  | 35  | graphics. h |
| 8  | 35  | graphics. h |
| 9  | 35  | graphics. h |
| 10 | 35  | graphics. h |
| 11 | 26  | math. h     |
| 12 | 35  | graphics. h |
| 13 | 35  | graphics. h |
| 14 | 35  | graphics. h |

## 参 考 资 料



- [1] 林学焦等, TURBO C 2.0, 用户手册, 中科院希望高级电脑技术公司, 1990
- [2] 叶欣, TURBO C 2.0 参考手册, 中科院希望高级电脑技术公司, 1990
- [3] 徐金梧等, Turbo C 使用大全 (V1.5~2.0), 北京科海培训中心, 1989
- [4] 萧黎等, Turbo C 2.0 运行库函数源程序与参考大全, 中科院希望高级电脑技术公司, 1990
- [5] 东阳生等, Turbo C 2.0 高级系统程序设计技术, 北京希望电脑公司, 1991
- [6] 翟彬, 微机高级 C 语言调试技巧, 北京希望电脑公司, 1991
- [7] 谭浩强, C 程序设计, 清华大学出版社, 1992
- [8] 张福炎等, 微型计算机 IBM PC 的原理与应用, 南京大学出版社, 1986
- [9] 尹彦之等, C 语言的常用算法和子程序, 北京科海培训中心, 1989
- [10] 李沐荪, Turbo C 常驻内存实用程序及窗口式软件编程技术, 北京科海培训中心, 1989
- [11] 叶砚霜等, 用 C 语言开发 PC Tools 原理与实例, 北京希望电脑公司, 1991
- [12] 明智, 中、高分辨率接口板 EGA/VGA 应用开发指南, 北京科海培训中心, 1990
- [13] 舒青, 微型计算机高级图形程序设计技巧与实例, 中科院希望高级电脑技术公司, 1990
- [14] 用 C 语言开发图形软件, 中科院希望高级电脑技术公司, 1988
- [15] 李春葆, 用 Turbo C 开发三维图形软件, 北京希望电脑公司, 1991
- [16] 朱传乃等, 80286 微机系统分析, 北京中科院计算所第十四研究室, 1988
- [17] Robert Jourdain, IBM PC XT/AT 机高级程序员编程指南, 中科院希望高级电脑技术公司, 1988
- [18] 张志辉等, Turbo Debugger 2.0 调试工具使用手册, 中科院希望高级电脑技术公司, 1990
- [19] 汪亚文等, Turbo Assmbler 汇编大全, 中科院希望高级有电脑技术公司, 1990
- [20] Microsoft Macro 5.0 宏汇编程序, 北京希望高级电脑公司, 1991
- [21] PC 中断调用大全, 北京科海培训中心, 1993
- [22] 计算机世界报, 1991 年、1992 年及 1993 年的合订本

















