



如何开发 Windows 95 应用程序

[美] Kevin J. Goodman 著

郭 勇 译

学习如何将现有的程序转换到 Windows 95 环境
给出有关 32 位程序设计的高效指南
学习如何充分利用网络支持
学习用多个应用程序进行内存管理



清华大学出版社



如何开发 Windows 95 应用程序

[美] Kevin J. Goodman 著

郭 勇 沈 龙
赵 军 赵 华 译
陈 隆 审校

清华大学出版社

(京)新登字 158 号

如何开发 Windows 95 应用程序

Building Windows 95 Applications

Original English Language Edition Copyright © 1995 by M&T Books.

Published by arrangement with the original publisher, M&T Books, U. S. A.

本书中文版由 Far East Books 授权清华大学出版社出版。

中华人民共和国国家版权局著作权合同登记章 图字: 01-96-0386 号

版权所有,翻印必究。

本书封面贴有清华大学出版社激光防伪标签,无标签者不得销售。

图书在版编目(CIP)数据

如何开发 Windows 95 应用程序/(美)Goodman,K. J. 著;郭勇等译. —北京:清华大学出版社,1996. 5

书名原文:Building Windows 95 Applications

ISBN 7-302-02261-5

I. 如… II. ①G… ②郭… III. 软件开发 IV. TP311.52

中国版本图书馆 CIP 数据核字(96)第 12887 号

出版者:清华大学出版社(北京清华大学校内,邮编 100084)

印刷者:清华大学印刷厂印刷

发行者:新华书店总店北京科技发行所

开 本:787×1092 1/16 印张:25.75 字数:640 千字

版 次:1996 年 10 月第 1 版 1996 年 10 月第 1 次印刷

书 号:ISBN 7-302-02261-5/TP·1105

印 数:0001—8000

定 价:38.00 元 附光盘书定价:63.00 元

目 录

第一章 32 位 Windows 介绍	(1)
1.1 32 位计算的优点	(1)
1.1.1 线性编程模式	(1)
1.1.2 更大的数据和可用地址空间	(2)
1.1.3 不需要复杂的编译器支持	(3)
1.1.4 优化为 32 位模式的处理器	(4)
1.2 了解目标平台	(5)
1.2.1 判断基本类型的长度	(7)
1.2.2 Intel 386/486 寄存器集合	(8)
1.2.3 MIPS R400 和 R4400 处理器	(13)
1.2.4 DEC Alpha AXP	(20)
1.3 小结	(21)
1.4 如何阅读 MAP 文件	(21)
第二章 移植到 Win32	(25)
2.1 两个代码基:一个用于 Win16,一个用于 Win32	(25)
2.1.1 处理 makefile 的问题	(26)
2.1.2 数据类型长度的改变	(27)
2.1.3 窗口消息的改变	(31)
2.1.4 API 调用的改变	(33)
2.1.5 分段体系结构问题	(34)
2.1.6 直接硬件访问	(34)
2.1.7 WIN.INI 和 SYSTEM.INI 的直接访问	(35)
2.1.8 无文档说明的函数和内部变量的使用	(35)
2.1.9 虚拟设备驱动程序(VXD)	(36)
2.1.10 对 hPrevInstance 的依赖	(36)
2.1.11 输入状态的改变	(37)
2.1.12 对 DOS 的依赖	(37)
2.1.13 移植汇编语言	(40)
2.2 一个代码基:与 Win16 和 Win32 兼容	(41)
2.2.1 C 语言的方法	(41)
2.2.2 得到移植宏	(45)
2.2.3 移植用于控件的宏	(45)

2.2.4	简易方案:使用应用程序框架.....	(46)
2.2.5	把C代码基升级到可接受的层次	(46)
2.2.6	把C程序作为C++程序编译	(49)
2.2.7	引入MFC对象	(50)
2.2.8	MFC 2.x和MFC 3.0的比较	(50)
2.3	关于MFC	(51)
2.4	UNICODE支持	(51)
2.4.1	MFC和多线程	(51)
2.4.2	MFC的优点	(51)
2.5	访问移植实验室	(52)
2.5.1	DEC(数字设备公司)的Alpha处理器移植实验室	(53)
第三章	形式替换	(54)
3.1	编译型形式替换程序	(57)
3.1.1	为形式替换程序编译器建立命令文件	(58)
3.1.2	编译命令文件生成(.ASM)汇编文件	(59)
3.2	编译型形式替换程序是如何运行的	(69)
3.3	调试	(70)
3.4	一般的形式替换程序	(70)
3.5	万能形式替换程序	(72)
3.5.1	登录形式替换程序	(73)
3.5.2	脱离形式替换程序	(74)
3.5.3	指针转换	(74)
3.6	范例程序:ExitWindows32s	(76)
3.7	Win16的开发	(90)
3.7.1	与Win16应用程序并存	(90)
3.7.2	NotifyRegister和Win32s	(91)
3.8	小结	(92)
第四章	进程的生存期	(93)
4.1	对CreateProcess的深入考察	(93)
4.1.1	可以被子进程继承的属性	(94)
4.1.2	不能被继承的属性	(95)
4.1.3	创建优先级标志和进程类型	(96)
4.2	线程	(98)
4.2.1	何时用附加线程	(99)
4.2.2	线程创建	(100)
4.2.3	挂起线程运行	(101)
4.2.4	设置线程的优先级	(102)

4.2.5	使用线程局部存储	(103)
4.2.6	终止一个线程	(105)
4.2.7	什么时候线程是不必要的	(105)
4.3	重叠的 I/O	(106)
4.4	结构和进程创建 API	(107)
4.4.1	环境变量	(107)
4.4.2	PROCESS_INFORMATION(进程信息)结构	(107)
4.4.3	停止 GUI 进程	(107)
4.4.4	同步	(108)
4.5	Win16 和 Win32 的区别	(108)
4.6	同步对象	(109)
4.6.1	关键段对象	(109)
4.6.2	事件对象	(110)
4.6.3	互斥对象	(111)
4.6.4	信号量对象	(112)
4.7	同步处理和 GDI 对象	(123)
第五章	Win32 的内存管理	(124)
5.1	Windows NT 与 Windows 95 的差异	(124)
5.2	Windows 95 和 NT 的虚拟内存管理	(124)
5.3	内存 API	(126)
5.3.1	虚拟 API	(126)
5.3.2	VirtualFree	(128)
5.3.3	保护页	(128)
5.4	稀疏内存管理策略	(130)
5.4.1	VirtualQuery	(130)
5.4.2	VirtualLock	(131)
5.5	Win32 中的全局以及局部分配	(131)
5.5.1	用标准 C 语言库函数操纵内存	(132)
5.6	Win32 新引入的堆 API	(133)
5.6.1	用于分配更多内存的堆分配函数	(134)
5.6.2	用于取消一个堆的堆分配函数	(135)
5.6.3	MFC C++ 的 new 和 delete 函数	(135)
5.6.4	跨进程内存管理	(136)
5.6.5	共享内存 DLL	(140)
5.6.6	用于传输只读数据的 WM_COPYDATA	(141)
5.6.7	ReadProcessMemory 和 WriteProcessMemory 函数	(142)
5.7	范例程序:读写内存	(145)
5.8	Windows 95 的可应用性	(169)

5.9 小结	(170)
第六章 登录	(171)
6.1 登录的必要性	(171)
6.2 登录的基本结构	(173)
6.2.1 键和值项.....	(173)
6.2.2 预定义句柄.....	(175)
6.3 确保 Windows NT 启动的控制设置	(177)
6.4 INI 文件映射	(178)
6.5 设置环境变量	(181)
6.6 如何建立应用程序数据结构	(181)
6.7 登录程序设计	(183)
6.7.1 查询登录.....	(185)
6.7.2 列举键和数值.....	(187)
6.7.3 写入登录.....	(187)
6.7.4 加密登录项.....	(188)
6.7.5 存储和从磁盘文件中恢复登录.....	(188)
6.7.6 连接一台远程机器.....	(189)
6.7.7 下载 SDK	(189)
6.7.8 性能数据.....	(190)
6.7.9 获取计数器数据.....	(195)
6.8 范例程序:REGISTRAR	(196)
6.9 小结	(200)
6.10 从登录中提取的典型内容.....	(200)
第七章 远程过程调用	(204)
7.1 什么是 RPC	(204)
7.1.1 Win32 RPC 与 OSF/DCE	(205)
7.1.2 RPC 和数据转换	(205)
7.2 开发 RPC 应用程序.....	(206)
7.2.1 使用 Microsoft 界面定义语言定义一个界面	(206)
7.2.2 GUID 结构	(207)
7.2.3 将类型属性与定义的类型关联.....	(209)
7.2.4 RPC 客户	(211)
7.2.5 客户 API 联编函数	(213)
7.2.6 RPC 服务器	(217)
7.3 范例程序:分布式 make	(220)
7.4 小结	(222)
7.5 命名约定	(222)

第八章 Windows 95 和 Win32 的异常	(223)
8.1 32 位 Windows 中的异常	(225)
8.1.1 内核中的未处理异常	(226)
8.1.2 异常处理程序和过滤程序	(226)
8.1.3 展开	(228)
8.1.4 不要吞没异常	(232)
8.2 软件产生的异常	(233)
8.2.1 终断处理程序	(234)
8.3 异常处理优先级	(237)
8.4 对异常的调试	(238)
8.5 结构化异常处理和 Win32s	(239)
8.6 结论	(240)
第九章 编写控制台应用程序	(244)
9.1 控制台应用程序不支持的功能	(246)
9.2 控制台应用程序中的挂接	(246)
9.3 定时器和控制台应用程序	(247)
9.4 控制台与其它窗口的交互	(247)
9.4.1 子类与控制台应用程序	(247)
9.5 控制台应用程序的功能	(247)
9.5.1 控制台句柄	(248)
9.6 对控制台输入和输出的 C 运行时库支持	(252)
9.6.1 GetLastError 与 errno	(253)
9.7 在控制台应用程序中使用 GUI 功能	(256)
9.8 控制台窗口与图形应用程序的组合	(258)
9.8.1 使输出定向于控制台窗口	(259)
9.8.2 接收控制台输入	(261)
9.9 重定向标准输入/输出	(262)
9.10 检测事件	(269)
9.10.1 控制台事件与信号	(271)
9.11 使用定时器	(271)
9.12 断开进程	(272)
9.13 从控制台应用程序中打印	(273)
9.13.1 在运行时确定控制台程序	(277)
9.14 小结	(280)
第十章 32 位操作系统——发展和特点	(281)
10.1 不要两次犯同样的错误	(287)

10.2	与 Win16 的兼容性	(287)
10.3	基于 32 位模式	(288)
10.4	多线程进程结构	(289)
10.5	独立的地址空间	(289)
10.6	稳定性	(289)
10.7	对不同体系结构的支持	(291)
10.7.1	SGI 公司的 MIPS R4x00	(291)
10.7.2	DEC 公司的 Alpha AXP	(291)
10.7.3	对称的多处理机制	(291)
10.8	安全性	(292)
10.9	Win32 中没有的功能	(293)
10.10	Win32、Windows NT、Windows 95 的对比	(294)
第十一章	Windows 95 Shell	(300)
11.1	Shell 游戏	(300)
11.1.1	Shell 扩展是如何工作的	(302)
11.2	使用 MFC 创建 Shell 扩展 DLL	(305)
11.2.1	使用 APPWIZARD 创建 _USRDLL	(305)
11.2.2	初始化 Shell 扩展	(307)
11.2.3	图标处理程序	(307)
11.2.4	上下文菜单处理程序	(310)
11.2.5	QueryContextMenu	(310)
11.2.6	InvokeCommand	(313)
11.2.7	IContextMenu	(316)
11.2.8	ICOPYHOOK 接口	(316)
11.2.9	登录 Shell 扩展	(317)
11.3	调试 Shell 扩展	(318)
11.3.1	如何建立或升级一个 Shell 扩展	(319)
11.4	小结	(319)
11.5	Explorer	(319)
第十二章	Windows 和 Win32 的安全性	(320)
12.1	用户识别	(320)
12.2	安全性标识符	(321)
12.3	访问令牌	(322)
12.4	可施加安全保护的對象	(323)
12.5	控制访问许可	(324)
12.6	范例程序:SecureView	(329)
12.7	启用优先权	(350)

12.8	范例应用程序:ExitWindows	(352)
12.9	安全性如何影响应用程序	(361)
12.9.1	独立工作站	(361)
12.9.2	单域和多域网络	(362)
12.10	模仿	(362)
12.11	安全性和屏幕保护程序	(363)
12.12	NT 的安全体系结构:总结	(363)
第十三章	图形设备接口	(365)
13.1	GDI 的客户/服务器的本质概述	(365)
13.1.1	对 GDI API 函数的修改	(367)
13.1.2	被删除的 GDI 函数	(367)
13.1.3	对 GDI 的改进	(367)
13.1.4	对 path 的改进	(372)
13.1.5	对位图的改进	(378)
13.1.6	新的 BITBLT 函数	(380)
13.1.7	照相铜版	(382)
13.1.8	增强型元文件	(382)
13.1.9	编辑元文件	(385)
13.1.10	修改调色板	(386)
13.1.11	改进 GDI 绘图性能	(386)
13.1.12	GDI 批量是特定于线程的	(388)
13.2	小结	(389)
第十四章	Win32s	(390)
14.1	Win32s 的工作方式	(390)
14.1.1	WIN32S.EXE:创建任务数据库	(391)
14.1.2	在 16 位和 32 位模块间转译的形式替换程序	(392)
14.1.3	构成 Win32s 系统的文件	(392)
14.2	Win32s 何时适合于应用程序	(395)
14.2.1	用 Win16 替代 Win32s	(395)
14.2.2	用 Win32 替代 Win32s	(395)
14.2.3	与 Win16 应用程序共存	(395)
14.2.4	NotifyRegister 和 WIN32s	(397)
14.2.5	用于 Win32 的 NotifyRegister 的另一个选择	(398)
14.3	调试 Win32s 应用程序	(398)
14.3.1	暂时不要丢开 INT 21H	(402)
14.4	小结	(402)

第一章 32 位 Windows 介绍

对一些人来说,Windows 95 的介入恰恰是在广大的程序员(和无数的用户)对 Windows 3.1 日渐得心应手的时候。但是,对于软件产业来说,创新就意味着要在用户尚未意识到他们需要它之前开发出能够满足这种需求的新产品。如果读者是众多的 Windows 3.1 的开发人员之一,那么一定知道 Windows 95 的推出已经有些迟了。如果不是一名 Windows 3.1 的开发人员,那也许是因为已经意识到了 Windows 3.1 的一些缺陷。不论怎样,本章将详细介绍 16 位计算的缺陷以及 32 位计算是如何克服这些缺陷的,从而向读者说明 32 位操作系统并非仅仅是计算机科学领域的一个尝试。本章还将概要介绍 32 位操作系统在编程上与以往的操作系统的细微差别,并将对 Windows 95 和 Windows NT 这两种操作系统做专门介绍。

由于编写能在 32 位操作系统上运行的代码只是整体工作中的一小部分,本章同时将对 Intel 和 RISC 两种平台做一些解释,以便读者在程序出现问题的时候能够对其进行调试。

1.1 32 位计算的优点

32 位处理器及操作系统较之 16 位处理器有多方面的优越性,显而易见的优点可以归纳为以下四种:

- 线性编程模式取代了分段编程模式;
- 更大的数据和可用地址空间;
- 不再必需复杂的编译器支持;
- 处理器被优化为 32 位模式。

我们将逐一介绍这些优点。

1.1.1 线性编程模式

32 位处理器的一个主要优点是它采用了线性编程模式。与线性编程模式关联的内存模式有时被称作平面模式(Flat model)。平面模式与分段模式中的微模式(Tiny model)类似。平面模式对 Intel 系列处理器来说并不陌生,它只是由 16 位模式转换到 32 位模式所获得的一种益处。Windows 3.1 在编写设备驱动程序时提供了平面模式。为了访问平面模式中的一个内存位置,只需使用偏移量即可。因为寄存器是 32 位的,所以平面模式下的内存寻址范围是 4GB。在平面模式中,段寄存器对于开发人员不再那么重要了。我们常可以听到,在 Intel 系统的平面模式中,SS、DS 和 CS 寄存器总是相等的(即 $SS=DS=CS$)。无论对于何种使用目的,这一点均成立,因为三个段寄存器都指向某一个相同的物理地址空间。不过,在调试过程中,会发现 SS、DS 和 ES 寄存器的内容总是相同的,而 CS 寄存器却总有一个不同的值。基于 RISC 的机器只支持 32 位模式,有些可以支持 4 位模式。

使这些段寄存器不再重要的原因是,在用户模式下运行时,应用程序无法改变这些寄存

器的值(在 NT 中,有两类进程:无优先权的用户模式和有优先权的内核模式。本书的实例均是用户模式进程)。虽然 Windows 95 不是真正的客户服务器类型的操作系统,但用户模式这个术语同样可以适用。CS、DS、ES 和 SS 的值是由内核(它在内核模式下运行)在创建用户模式进程时决定的。这就是 Windows 95 和 NT 分割地址空间的方法。同样的偏移量在进程 A 和进程 B 中可以得到不同的物理内存位置。作为一名开发人员,可以完全忘记段寄存器。这一点更适用于 MIPS、Power PC 和 Alpha 处理器,因为它们根本没有段寄存器!

1.1.2 更大的数据和可用地址空间

平面模式有另一个优点——它不需要编译器支持内存模式。多数 16 位编译器一般支持四种或五种不同的内存模式:

- 微模式:在这种模式下,代码和静态数据(包括堆栈)都必须可以存放在一个 64KB 的段里。多数保护模式程序不能支持这种模式。因此,许多编译器不对这种模式提供支持。如果一个编译器支持微模式,那么所得程序为 COM 程序(由 .COM 扩展名得名)。
- 小模式:小模式程序有一个代码段和一个独立的数据段,它们均必须小于 64KB。
- 中模式:中模式程序可以包括多个代码段(它们中的每一个都必须小于 64KB)和一个数据段。
- 紧凑模式:与中模式相反,紧凑模式的程序只有一个代码段而有多数据段。
- 大模式:大模式程序可有多个代码段和多个数据段。
- 巨模式:巨模式程序基本与大模式相同,但是它有创建大于 64KB 数据对象(如数组)的能力。

在 Win32 中,这些内存模式不复存在。不过,在 Win32 中仍存在多种模式编程的痕迹,其中之一就是 FAR 和 NEAR 关键字(FAR 和 NEAR 实际上是类型定义关键字。ANSI 标准需要 `_near` 和 `_far`)。FAR 和 NEAR 可以帮助用户越过内存模式的障碍。例如,在中模式程序中,可以通过使用关键字 FAR 来访问缺省数据段以外的数据。关键字 FAR 指示编译器通过选择器和偏移量来引用一个变量而不是只使用 16 位偏移量。

在大模式、紧凑模式和巨模式中,数据的缺省类型是 FAR。为了访问大模式、紧凑模式和巨模式中缺省数据段中的数据,16 位的程序员需使用 NEAR 关键字。同样的道理也适用于代码段:微模式、小模式和紧凑模式中的缺省类型是 NEAR;而中模式、大模式和巨模式中的缺省类型是 FAR。为了具有兼容性,Win32 仍支持 FAR 和 NEAR,不过它们已没有任何意义了。所以,如果你看到别人的代码中包含 FAR 指针和 FAR 函数,不要惊慌。该段代码的编写者可能是在设法与 Win16 保持源代码级兼容;或者,他可能是一位尚未改变分段操作系统编程习惯的老的 Win16 程序员。

Windows 95 和 NT 中所有类型的缺省值均为 NEAR(与微模式中相同),因为在引用对象和调用函数时只需使用一个偏移量。

消除了 16 位计算中 64KB 的障碍是 32 位系统的另一个显著优势。由于每个进程中代码和数据的存储空间为 2GB,NT 和 Windows 95 可以更高效率地支持大于 64KB 的数组等数据对象,为大于 64KB 的图形和代码等分配空间。在巨模式程序中,16 位的编译器也可以

支持这些方式。不过,速度要慢得多,下面一节将讲述其原因。

1.1.3 不需要复杂的编译器支持

在 Microsoft Systems Journal 1992 年 7 月刊中的“The Case for 32 Bits”一文中,Charles Petzold 说明了为什么一个程序的 16 位版本在运行速度上几乎比同一程序的 32 位版本慢了五倍的原因。执行速度缓慢的原因是,16 位编译器需要做许多额外的工作才能执行 32 位的操作。在 32 位模式下,编译器(或用户)不需要产生额外的代码去管理分散在多个段中的对象。请看下面这段代码范例,它执行的操作是将两个数求和,然后调用一个虚构的用于求平均值的名为 avg 的函数:

```
void avg(int *, int *, int *);  
int a,b,c;  
void main(void)
```

```
c=a+b;  
avg(&a,&b,&c);
```

在按小模式程序编译时,用 Microsoft C 8.00 产生的 16 位汇编语言如下:

```
_main    PROC NEAR  
    mov  ax,WORD PTR _b  
    add  ax,WORD PTR _a  
    mov  WORD PTR _c,ax  
    push OFFSET DGROUP:_c  
    push OFFSET DGROUP:_b  
    push OFFSET DGROUP:_a  
    call _avg  
    add  sp,6  
    ret  
  
_main    ENDP
```

然而,如按 Large 模式编译,16 位的汇编语言代码膨胀成了这样:

```
_main    PROC FAR  
; Line 4  
; Line 5  
    mov  es,WORD PTR $T106  
    mov  ax,WORD PTR es:_b  
    mov  es,WORD PTR $T107  
    add  ax,WORD PTR es:_a  
    mov  es,WORD PTR $T108  
    mov  WORD PTR es:_c,ax  
; Line 6  
    push es
```

```

push OFFSET _c
push SEG _b
push OFFSET _b
push SEG _a
push OFFSET _a
call FAR PTR _avg
add sp,12      ;000cH
; Line 7
ret

```

如果从来不看汇编语言的输出,那么也许永远也不会知道这个程序的大模式版本竟比小模式版本多了约 15%。因为变量 a,b,c 分别定位在不同的数据段中,所以 16 位编译器必须产生附加代码来操纵这些段。这个道理对于 avg 函数调用也适用。就编译器所知,avg 是一个独立的代码段,因此编译器就必须为此做些调整。如果实际上是我们编译并链接了这个简单的程序,那么我们也可以看出这两个程序在磁盘上所占空间的不同,如清单 1-1 所示。在 Win32 中由于只有一种模式,因此编译器不需要做额外的支持。

清单 1-1 16 位程序由于内存模式不同而占用不同的空间

TEST	EXE	2875 9-29-94 9:44a
TESTFAR	EXE	3257 9-29-94 9:44a
		2 file(s) 6132 bytes

1.1.4 优化为 32 位模式的处理器

OS/2 最初的带有挖苦性的口号是“比 Windows 更好的窗口系统”。IBM 设计这条营销标语是为了将操作系统购买者的视线从 Windows 3.1 上吸引走。这也许误导了某些开发人员,因为对现实更好的描绘是,32 位的解决办法比 16 位的好。换句话说,不要劝说开发人员在 32 位操作系统去开发 16 位的应用程序,干脆劝说他们去直接开发 32 位的应用程序。

同样的方案也适用于 Windows 95 和 Windows NT。它们所支持的处理器在 32 位模式下运行比在 16 位模式下运行的速度快。不仅如此,在诸如 MIPS R4x00 和 Alpha AXP 等的处理器上,相应的 32 位应用程序比起 16 位的应用程序有令人难以置信的巨大优势,因为处理器必须通过一种仿真模式才能运行 16 位的软件。如果注重的是程序的性能,那么一个 32 位版本的 Windows 应用程序在 NT 上的运行速度将比 16 位版本更快。

这一点在 16 位版本需要使用大模式时尤为正确。特别使我感到不安的一件事便是看到有人在不必要的情况下,在 16 位 Windows 上使用大模式。很多时候,一个缺乏经验的程序员是意识不到这样使用大模式所带来的损害的,毕竟其 C 源代码与其它模式看起来没有什么不同。如果静态数据不到 64KB,那么在 Windows 3.1 上使用大模式是不必要的。在 NT 中,开发人员则可以不必为此多费心思。

不幸的是,对 Windows 3.1 的程序来说,有些情况下大模式的使用是不可避免的。在 Windows 3.1 中驻留在 DLL 中的所有 Microsoft Foundation Class(MFC)类必须用大模式编译,这又是由于 16 位系统的复杂编程模式造成的,因为它规定在 DLL 中堆栈段与数据段不

等(即 $SS! = DS$)。

即使开发人员设法避免在 Windows 3.1 中使用大模式, FAR 调用和 FAR 数据仍然会使性能降低, 选择器加载的开销尤其大。当在其它段中调用例程或者在多个段中操纵数据时, 发生选择器加载。可由下表比较一下 16 位 NEAR 调用和 16 位 FAR 调用在 Intel x86 系统上所用时间的不同。

调用	时间
NEAR 调用	7 个时钟周期
对内存中段的 FAR 调用	28 个时钟周期

这在 Windows 3.1 中是非常普遍的, 因为多数程序使用中模式。为了避免在 Windows 3.1 上使用不必要的选择器调入, 开发人员有时人工优化代码, 以便把那些经常互相调用的函数放在同一个段中, 形成一个准 NEAR 调用。

在 Windows 3.1 的标准模式中, 程序员还要确保各段不要有太大的责任。不像 386 增强模式那样可以将内存分页到在 4KB 的簇中的一个交换设备上去, 标准模式只能将整个段交换出去。于是产生了 Windows 3.1 的第三方案, 它统一了代码段的大小。它的思想是: 如果所有段是同样大小的, 那么标准模式中的 Memory Manager(内存管理器)则可以随意地控制各段的交换, 而无需寻找一个足够大的空间来放置该段。

64KB 的障碍不仅限制了代码段和数据段, 算术运算同样受到了 16 位模式的限制。变成 32 位模式以后, 整数的算术运算性能将得到提高, 因为编译器不再需要使用临时的 16 位寄存器来仿真 32 位模式了。

不过, 在 Win32 中不再有这些麻烦。在 Win32 中, 如果是在编写全新的代码, 那么可以完全忘记 NEAR 和 FAR 以及 64KB 的约束。如果是在从 16 位 Windows 移植代码, 那么那些 NEAR 和 FAR 也无碍于事。

值得一提的是, 大模式有一点优越性。因为大模式程序中的指针缺省值是 32 位的, 所以将大模式代码移植到 32 位模式的工作量便不像移植中模式程序那么大(因为必须保证所有指针都从 16 位变到了 32 位)。

1.2 了解目标平台

许多 16 位计算固有的缺陷程序员是看不出来的。如果不花点时间来看看 16 位编译器的编译结果, 那么也许会开发出一段执行效果很不理想的程序, 而又不知其中的原因。这一问题在 Win32 中同样存在。因为虽然 Windows 95 和 Windows NT 使用 32 位模式, 并且 32 位模式可以解决 16 位计算的一些问题, 但这并不能作为可以忽视其底层结构的借口。尤其是现在每一种新的开发工具的问世, 似乎都使开发人员和他们正在使用的机器离得愈来愈远了。有了诸如 Windbg 或 Visual C++ 2.0(见图 1-1 和图 1-2)等程序, 可以很容易地与所使用的机器相互独立开来。

这一点当在一个图形环境(如 Windows)编程时尤为正确。可以设想这样一种方案, 一个比较聪明但又不熟悉 Windows 95 的人学习了“Power Objects”(一个假想的编程环境, 它可

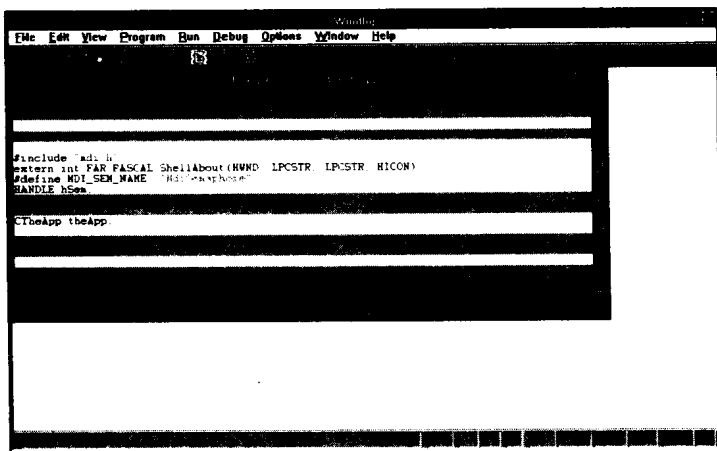


图 1-1 Windbg, 一种 GUI 源代码级调试程序

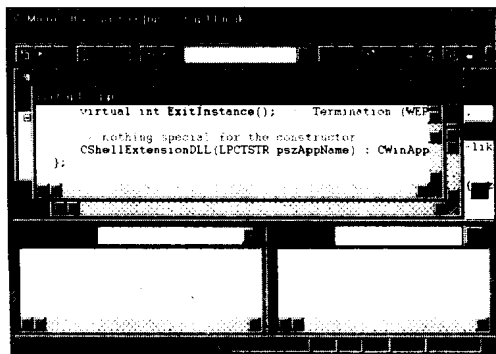


图 1-2 Visual C++ 2.0≠GENERIC

以为开发人员生成代码)之后就不再学习其它任何关于该图形环境的知识了,那么几天之后,这个人就可以交付能够产生一些可爱的简单图形的程序原型。不幸的是,几周之后,这个人所能交付的仍旧是这些程序原型。为什么呢?这当然不是 Power Objects 的问题。在一个有经验的 DOS/Windows 程序员手中,Power Objects 可以产生的远不止这些。但是一种高级工具只能为用户服务到这种程度。当碰到这种工具的限制时,除非懂得它是如何工作的,否则便不能更好地利用它。换句话说,在一个初学者手中,每项工具都有其不可超越的界限。

许多成功的 Windows 3.1 的开发人员都会告诉你,他们成功的部分原因是他们对基础硬件了如指掌。对于 Win32 的开发人员来说,这一点同样适用。

不过,请不要误解我的意思。我不是在抨击“Power”之类的语言。事实上,本书中的实例是用 Visual C++ 创建的。然而,这有很大的不同。一旦懂得了 Win32 API 并且熟悉了 Windows 95 和 NT 的基础体系结构,就可以对存在问题的地方进行调试或者可以超越 Visual C++ 的极限。

如果不了解编译器对代码做了哪些工作,那么便不可能写出高效率的程序。如果不知道机器希望看到怎样的程序,那就只能听由编译器摆布了。观察编译器对源代码做了哪些工作

有两种办法,一种是检查 MAP 文件(参阅本章 1.4“如何阅读 MAP 文件”),另一种办法是学会看编译器生成的汇编语言。

对了——就是汇编语言。人们已经讨论了这么多关于 NT 本身是如何几乎全部用 C/C++ 写成的,你可能因此已不再想使用汇编器了(Windows 95 是用 C 语言和汇编语言写的)。因为大多数编译器可以产生并读懂汇编语言(利用 `_inline` 关键字),把编译器丢在一边是一个好主意,不过可不要丢弃汇编语言知识。我可以理解,忘掉汇编技能对开发人员有很大的诱惑。除非是在 Win32 环境下编程,否则你的代码对其它平台没有可移植性。而这种可移植性又是通过写 C/C++ 代码获得的,而不是汇编语言。有了 Visual C++ 中可作源代码级调试的调试程序,更增加了忘记汇编语言的诱惑力。但我几乎从来不在源代码级调试程序。因为如果是我写了源代码,那么我应该知道它的功能。我想知道的是,在编译器看来,我的源代码的功能是怎样的。所以,最好的协调办法是用 C/C++ 编写源代码,同时懂得基础硬件的汇编语言。

Windows NT 使编程人员得到的程序是那些源代码与 MIPS, Intel, Power PC 和 Alpha 处理器平台(或者其它可用的平台)有兼容性的程序。在我写本书的时候,还没有可用的交叉编译器。这也就是说,如果想提供一个 Alpha 版本的软件,那么就必须先有 Alpha 处理器,并准备好在 Alpha 处理器上进行调试。如果想提供一个 MIPS 版本的软件,就必须最终在一个有 MIPS 处理器的机器上调试。如果想提供一个 Intel 版本的软件——那么我想你一定知道应如何去做。

1.2.1 判断基本类型的长度

在任何一个想做开发工作的操作系统上,了解基本数据类型的长度都是很重要的,因为这些基本数据类型的长度是独立实现的。表 1-1 列出了 Win32 中的基本类型及其长度。

表 1-1 Win32 平台上的基本类型长度

类型	长度
char(字符型)	8 位(一个字节)
short(短字)	16 位
int(整型)	32 位
long(长字)	32 位
float(浮点型)	32 位(IEEE)
double(双字)	64 位(IEEE)
ULONG;	32 位
USHORT;	16 位
UCHAR;	8 位
DWORD;	32 位
BOOL;	32 位