[美] Matt Pietrek 著
米东 王森 等译

美国IDG“奥秘”丛书

Windows 95 系统编程 奥秘

Windows 95 System Programming SECRETS

了解内部：
包括Windows 95系统和存储结构、进程、线程、Win16模式和任务等。

掌握底层：
如User和GDI子系统、KERNEL32、PE和COFF OBJ格式。

达到新高度：
通过研究Win32平台的内部行为和数据结构，可以使你在Windows开发方面达到一个新高度。

IDG BOOKS™
PHEI
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Windows 95系统编程奥秘

用Matt Pietrek的特别程序探索Windows 95.

包含Spy实用程序和有价值的源代码

打开Windows 95黑盒子，深入到Windows 95内部：

本书不仅能指导“我如何写好一个Windows 95应用程序”，而且提示了核心技术资料，通过展望Win32 (Windows NT, Win32s和Windows 95) 前景，你可了解到Windows 95是如何适应Microsoft Win32策略这个大文章的，还可进一步逐层了解这一强有力的桌面操作系统，通过彻底地熟悉Windows 95，你可获得真正的优质产品及编制更先进健壮的应用程序。

解开Windows 95编程之秘密

- 发现Windows 95新特点
- 深入了解运行模式、进程和线程的数据结构
- 获取16位和32位数据结构的真实细节
- 解剖Windows 95存储结构
- 明确KRNL386.EXE、KERNEL32.DLL和VWIN32.VXD之间的关系
- 细述Windows 95可执行格式
- 获得开发Windows 95的专门知识

ISBN 7-5053-3278-3

9 787505 332782 >

系统要求
80386或更高，Windows 95

读者范围
中级～高级编程人员

IDG BOOKS™

PHEI

ISBN7-5053-3278-3/TP・1226

定价：85.00元

美国 IDG“奥秘”丛书

Windows 95 系统编程奥秘

[美]Matt Pietrek 著

米东 王森 等译

电子工业出版社

内 容 简 介

本书属于美国 IDG 环球出版公司的“奥秘”丛书之一。

本书详细揭示了 Windows 95 系统的核心技术，解剖了 Windows 95 的存储结构、进程、线程、模块、任务以及有关函数。

本书共有十章，第一章展望了 Windows 95 的未来，与 Windows NT 等工具软件进行了综合比较。第二章论述了 Windows 95 的新特点，与 Windows 3.1 进行了详细的比较。第三章详细论述了 Windows 95 中的模块、进程和线程。第四章和第五章分别揭示了 Windows 95 中 USER 和 GDI 两个子系统，以及 Windows 95 的内存管理。第六章详细分析了 Windows 95 的三个核心部件(VWIN32. VXD、KERNEL32. DLL 和 KRNL386. EXE)。第七章论述了 WIN16 模块和任务。第八章考察了 Windows 95 中可移植的执行模块和通用目标文件格式。第九章教读者自身去探索 Windows 95 奥秘的方法。第十章作者介绍了自己编写的一个 WIN32 API 侦探程序，读者可在此基础上进行扩展。

Windows 95 System Programming SECRETS　　by Matt Pietrek

美国 IDG “奥秘” 丛书

Windows 95系统编程奥秘

米东　王森　等译

责任编辑：郭庆春

*

电子工业出版社出版（北京市万寿路）

电子工业出版社总发行　各地新华书店经售

北京市顺义县天竺颖华印刷厂印刷

*

开本：787 × 1092 毫米 1/16　印张：38.25　字数：976 千字

1996 年 8 月第一版　1996 年 8 月第一次印刷

印数：5000 册　　定价：85.00 元

ISBN 7-5053　3278-3/TP · 1226

著作权合同登记号　图字：01-95-940 号

IDG Books 奥秘系列丛书的优势

奥秘系列丛书的宗旨很简单：给专家作者设一个论坛，把他或她的经验传授给读者。奥秘系列丛书的作者（而不是出版公司）直接组织、精选和处理该课题的材料。作者们通过文章的反馈、培训班、e-mail 交换、参加用户组织及咨询工作与最终用户保持密切联系。由于作者们了解计算机日常使用的现状，我们的奥秘系列丛书具有一个战略优势：奥秘系列丛书的作者们与读者们是紧密结合的。

奥秘系列丛书的作者们具有以最有效的方式处理某一课题的经验，并且我们知道您——亲爱的读者，会从与作者一对一的关系中受益。我们的调查表明，读者们之所以要购买计算机书籍是想就某一产品听取专家们的意见，读者们想从作者的经验中受益，而奥秘系列丛中自始至终都有作者的声音。有些人将阅读该丛书比喻成在休息时间与作者聊天，并得到作者充分的关注。

此外，在奥秘系列丛书中，作者还免费提供或推荐一些有用的软件。随书所附的软件是经过精心挑选的，与书中的内容、主题或步骤有密切的关联。我们相信您会从所附的软件中获益匪浅。

无论您是从头到尾一段段地阅读，还是一次只读一个专题，您都会从本书中发现您所需要的内容。作为一位计算机用户，您应该获得全面解答的资源。IDG 书籍出版公司荣幸地以《Windows 95 系统编程奥秘》为您提供这一资源。

译 者 序

Windows 工具软件可以说是目前最优秀的软件之一，Windows 95 又是 Microsoft 公司最新推出的功能极丰富的产品，市场需求前景是非常可观的，在相当一段时间内必定是广大计算机软件工作者所青睐的产品，同行们若对 Windows 95 想有一个较深入地了解，请参阅本书。

要求本书的读者对 Windows 的使用比较熟悉，至少对 Windows 3.1 有比较强的编程能力，同时对 C 语言有一定的编程经验，那么你读起这本书来会更有情趣。

正如作者比喻的，Windows 95 是一个扑朔迷离的洞穴，则此书是为那些洞穴探索者所准备的。本书不是教你如何去编写 Windows 95 程序，而是带你走进其内部，详细考察 Windows 95 的本质，认清其本来面目，以便更好地使用 Windows 95 开展有关工作，但愿本书会给你带来较大帮助。

参加本书翻译的有：米东（第一章、第二章、第三章（前 11 节）），郝杰成（第四章），崔新军（前言及序部分，第三章（后 8 节）、第五章），朱金钧（第六章），齐剑峰（第七章），卢凌云（第八章、第九章），王森、董永乐（第十章、附录 A），米东、王森作了全书统校。

由于译者水平有限，错译之处在所难免，敬请读者指正。

1996 年 4 月 于石家庄

序

《Windows 95 系统编程奥秘》是 Matt Pietrek 有关如何真正了解 Microsoft Windows 的第三项主要工作。Matt 从事 Windows 方面的工作已有多年。1988 年从 Santa Cruz 大学毕业后，他就开始逐渐显露了他的这方面的技术才华，尽管当时他获得的是物理学学位而且只拿到了二门计算机学科方面的学分。在进入 Borland 的技术保障部之后，他很快获得了客户的高度评价。

之后，他在 Borland 的 R&D 室取得了更大的成绩。他编写了 TDUMP 和 WinSpector，甚至在 OS/2 Turbo Debugger 上也卓有成效。其后，由于 Borland 的多次裁员，他离开了 Borland 而最后来到了他自己的 Nu-Mega 公司。今天，他已经成为了 Bounds-Checker 系列产品的主要设计者。

我第一次与 Matt 相遇是在 1991 年春天召开的软件开发会议(Software Development conference)上，那时我们所倡导的 Windows 尚未引起关注。Charles Petzold 和我是 Windows 与 OS/2 辩论会的小组成员，当时我们已是四面楚歌，公众都已认定未来必是属于 PC OS 的了。

然而，事实却如 Matt、Charles 和我所预料的——今天，Windows 已经远远超出纯技术的王国而成为大众文化的一个部分。在 Windows 95 展示周里，其盛况超过了“侏罗纪公园”首映的场面。但事实上，在您为此欢欣鼓舞之际，尚有许多问题等待解决。完成最后一个从 Windows 3.x 向 Windows 95 的升级尚需多年的努力，只有到那时我们才算真正地进入了 32 位世界。

如果将 Windows 比做一个扑朔迷离的洞穴，那么这本书就是为那些洞穴探索者所准备的，而且要比 Win32 API 更合适。Matt 是进入 Windows 95 洞底的第一人。(事实上，按本书的工作性质讲，可称做“Windows 洞穴探险”。)许多其它“当今的”Windows 95 书籍曾承诺向您展示所有的黑盒子，但实际上一、二年前就已不再“黑”了。为了抢到先机，那些书本的作者往往会不合时宜地走在了前面，而对 Chicago 的探险也不会超过 1994 年 5 月份公诸于世的内容。甚至，有些工作还在被那些过时了的信息和错误的假设搞得头绪皆无。

与此相反，Matt 则深入洞察了 Chicago 的一切——包括 Windows 95 的零售版本——因而给您带来了真正时新的信息。还在等什么？亲爱的读者！还不戴上您的安全盔，点起手中的火把，开始您的洞穴探险！

Eric J. Maffei
Microsoft 系统杂志主编
1995 年 9 月于纽约

目　录

引　言

* *

近来，Microsoft 一直在问，“今天你又想到哪儿去?”现在，公司已不再介意推出 Windows 95 是我们达到目的的一种手段的观念了。我们作为编程者需要知道的是，Windows 95 是不是真正地如我们所预期的那样有效。几乎所有的人都会同意，Windows NT 如凯迪拉克一样漂亮(如果你更愿意的话，你也可以称它是奔驰)，因为它有着近乎完美的设计和安装。但问题是，Windows 95 会不会成为雪弗莱或其它跑车呢？唯一的验证办法是打开车盖自己去看个究竟。这也正是我写此书的目的。只有在考察了像 Windows 95 之类的操作系统的本质之后，我们才有可能自己认清它是否有排气管和火花塞，以及它的安全性，舒适性如何。

你可能不理解，像我这样的编程人员，为什么一直会对 Windows 95 之类的操作系统的根本细节情有独钟。如果把更多的精力投入像 OLE、MFC、或最新的图象或是多媒体 API 之类的新科技上去，不会更好吗？当然，对一些编程者来讲，只要了解得够用就行了，但也有一些编程者则更喜欢刨根问底，把代码层搞得透透彻彻，也或许我们并不想把自己的代码运行于不大了解的其它代码之上。无论什么原因，《Windows 95 系统编程奥秘》一书是为这些人所准备的。知识就是力量，你对 Windows 95 这样的系统掌握得越多，你对它的控制就会越强。

《Windows 95 系统编程奥秘》并不想透视 Windows 95 设计与执行的各个方面，我选择了自己深感兴趣的内容。希望本书能为您自己的 Windows 95 编程工作中提供一些有益的参考。

关于您和读者的假定

从全书的背景考虑，这里对读者作一定的知识假定。当然，确切地讲，本书的读者应当具有较强的 Windows 编程能力，至少为 Windows 3.x 编过程。本书并不是一本有关“如何为 Windows 95 编写程序”之类的书籍，我想这方面的书已彼彼皆是了。

因此，《Windows 95 系统编程奥秘》一书的出发点是认定您已知道了在 Windows 3.1 和 Windows 95 中的编程方法，而且您现在正想继续走下一步：想了解为什么 Windows 95 会如其所设计的那样工作。

通过了解 Windows 95 内部的理论黑盒子的工作方式，您就可以按照 Windows 95 的工作方式来达到自己的目的而不会陷于盲目。这样，您的程序当中出现了故障时(这当然是我们应尽力避免的)，如果您已经了解了 Windows 95 的工作原理，那么调试过程就会非常迅速。为什么会这样？因为如果您掌握着 Windows 95 的一举一动，您通常可以在调试过程中尽早地明确程序是在何处中断的。

本书的示例是以 C 语言的形式写出的，并混入了一些汇编语言。我所提供的各种 Windows 95 伪码也是以 C 语言为基础的。因此，为了有效地掌握本书内容，您应当了解有关 C/C++的知识。如果您在其它编译语言，如 Borland Pascal/Delphi 上有一定的编程经验，则也可以。

伪码

由于本书的目的是首先要展示 Windows 95 的工作机理，我需为 DLL 系统中的各种函数提供伪码，这类伪码通常与可编译的 C 代码相似。但为了不破坏 C 语言的语法规则，我单独给出了伪码。这里的伪码是以 Windows 95 的调试版本为基础的，它提供了许多有用的诊断字符和其它信息，可以使我们更容易精确察看到 Windows 所作的一切。如果您运行的不是 Windows 95 的调试版本，应当改变过来。在有错误出现时，Windows 95 的调试 DLL 会给出非常有用的信息。如果您不用调试版本，而想用零售版本，那么您的调试程序的内容与本书的伪码会有一定的区别。

样本程序

《Windows 95 系统编程奥秘》有几个程序可供您考察 Windows 95，它们均附于本书的磁盘上（包括.EXE 程序和源程序）。由于这些源代码要一次占去 30 多页，除了第十章的 APISPY32 程序外，在书中正文基本没给出。第十章的重点是建立一个 Win32 API 的侦探程序，因而在论述有关概念时需对源代码进行深入的考察。

如果您读过《Microsoft 系统杂志》或《PC 杂志》，您在这些杂志上会发现与本书相类似的某些程序。事实上，书中的一些章节在上述杂志上已经讨论过。但是，如果您已经读过这些杂志，在看本书时也不要跳过相应的章节，因为书中的程序已较以前大有改进。而且，由于当时篇幅所限，有些内容并未涉及。

例如，第八章的 PEDUMP 程序在功能方面已比当时发表于《Microsoft 系统杂志》上的程序大了一倍。与此类似，在《Microsoft 系统杂志》上出现过的 APISPY32 程序也比 Windows 95 中的情况差得远。本书的 APISPY32 程序是以船载 Windows 95 形式工作的，NT 3.51 中的.EXE 程序也有类似的情况。

第 一 章

展望 WINDOWS 95

* *

Win32 综述

我写这本书的时候，Microsoft 正在狂热地制作 Windows 95 的拷贝。

另一方面，Windows NT 已经使用两年了，并且在许多人的脑海里已经形成了某种印象，其印象是即慢又笨(Windows NT 3.5 在第一次发布的 NT 基础上有重大改进。然而，仍有许多初始的毛病，我很希望进一步开发 Windows NT)。与 Windows NT 3.1 同时发布的 Windows 3.1 的 Win32 库函数，被广泛地认为不值得付出多大精力去用它开展工作。

一直到 Windows 95 发布以后，才看到了 32 位 Windows 编程的光明前途。如果你想占领 Microsoft 地盘并且继续使用最新技术，不管你是否喜欢，你必须用 Win32 编程，Microsoft 正在把它的全部成果置入 Win32 系统中。尽管将来的 Microsoft 操作系统会继续支持 16 位 Windows 3.x 应用程序，但 16 位程序不能利用许多新特性的优点。Win32 则是未来，关键是“你的编程注意力在哪儿?”

这本书的主要任务集中在 Windows 95 的结构和执行方面，该操作系统刚进入 Win32 这块竞技场地，尽管 Windows NT 和 Win32 系统已经运行一些时间了，直到 Windows 95 出现前，许多人在 Win32 编程方面并未给予多大关注。Microsoft 公司针对 Win32 的应用程序接口(API)策略和其操作系统的可测性，我们已经接触三年了，决不能把 Windows 95 看成是一个没有历史的新名字。此外，尽管 Windows 95 目前很响，但正在设计 Microsoft 未来操作系统的是 NT 小组，Microsoft 计划将来把 Windows NT 和 Windows 95 合并，其结果主要是基于 Windows NT 技术，少量地基于 Windows 95。所以，在研究 Windows 95 的技术细节之前，我在本章简述一下 Microsoft Win32 策略的意义，展示一下 Windows 95 如何安装(适合)到图形中，相信我，本书的其余部分均为 Windows 95 的资料及其执行情况。但是，第一章是很重要的，从中可了解 Windows 95 置入 Win32 编程和 Win32 API 这个大文章中的情况。

无疑，Microsoft 不喜欢我的这些说法，因为一直认为“只有一个 Win32 API，写出的程序可在我们系统上运行。”尽管这似乎是一个好思想，实际上是不行的。

进行这些讨论的最好方法是，要么先定义一下 Win32，要么用 Win32 定义一系列的操作系统函数(一种 API)，这些函数称为 Win32 API，当 Microsoft 第一次

介绍 Windows NT 时，许多程序员混淆了 Win32 和 Windows NT 两者的不同，Windows NT 仅仅是 Win32 API 的一个工具。然而，自从它第一次被宣布为 Win32 工具以来，一些程序员很难区分操作系统(Windows NT)和它的 API(Win32)，关于 Win32，Microsoft 的主要目的之一是提供简单的 Win32 API 入口，Win32 API 函数非常类似于 Windows 3.x API(如视窗管理和显示输出)。如果 Microsoft 限制了 Win32 API 在 Windows NT 上的用法，那么 Windows 95 则有很大不同，Microsoft 公司为几种操作系统提供了 Win32 API，每一种操作系统被允许适应一种专门环境。对于高档计算机而言，Windows NT 是优选的工具软件，对于受存储空间限制仍然运行 Windows 3.1 的低档 386 微机，在 Windows 95 发布之前，Win32 系统库函数仍是优选的，Microsoft 的着眼点是，用户可以用 Win32 API 编写程序，其可执行程序可运行在任一 Win32 工具软件上。

从理论上讲，每个操作系统中的 Win32 API，都可以掩盖不同硬件或低级操作系统设计的缺陷，这与 Microsoft 公司的“Scaleable Architecture”计划有关，这个计划是在 1992 年 2 月第一次 Win32 开发人员讨论会上倡仪的，Win32 这个名字意味着从 Windows 3.x API 转换成 Win32 API。最关键的优点是 32 位编码，定义 Win32 API 时，Microsoft 还概述了新 32 位可执行文件格式，其格式类似 PE (Portable Executable)格式，是由 UNIX 系统目标文件格式 V(COFF)派生来的。Win32 API 和 PE 格式相匹配，所有的 Win32 操作系统都使用 PE 格式作为主要的可执行格式，由于对所有 Win32 操作系统使用了同样的可执行格式，Microsoft 希望 Win32 程序可运行在 Win32 工具软件上。当然，目前来讲可移植性已经不是很关键了，当可执行格式可移植时，你仍不能把为 DEC Alpha 编译的程序运行在一个 Intel CPU 的计算机上去。

在 Windows NT 面世不久，Microsoft 宣布了 Win32 API 的另一个工具 Win32s，其作用是集结 Microsoft 提供的 DLL 和虚拟设备驱动程序(VxD)，被加载到装有 Windows 3.1 的机器上，以便可运行 Win32 程序。不幸的是 Windows NT 的某些特性在 Windows 3.1 的结构下是不适应的。因此产生了 Win32s 的概念，Win32s 库函数提供了某些(不是全部)Windows NT 和 Windows 95 具有的 API 功能，实际上，Win32s 后面的 s 代表子集(subset)，Win32s 的主要不足是不支持现代操作系统的许多特性，如线程和分址空间，线程是高档操作系统的一个特性，它允许一次可执行多个程序。线程的传统用法是一个线程处理打印，另一个线程继续响应用户的输入。Win32s 也易被 Windows 3.X 的某些限制所制约(详看本章“Win32s 工具软件”一节)。

像 Win32s 一样，Windows 95 仅仅提供了如 Windows NT 定义的 Win32 API 的一个子集，Microsoft 原来以子集 Win32c 命名的(尾部的 c 代表 Chicago，这是 Windows 95 原来的名字)。Win32c API 子集包括 Win32s 中的所有功能，并且在 NT API 集合上追加了一个有效数。Windows 95 成功希望是很大的，因为尽管他的 API 是 Windows NT 的子集，但 Windows 95 含有大量程序设计人员在先进的操作系统中发现的特性，例如线程和分址空间(这两个特性 Win32s 是没有的)，一般讲程序设计人员喜欢分址空间，因为这一特性可防止其它程序的冗余数据的影

响，更重要的或许是防受操作系统本身的影响。Windows 95 需要的内存比 Windows NT 少，这使得他更适合于通常桌式 PC。

和 Windows NT 不同，Windows 95 未考虑与其它处理器的可移植性，这是因为 Intel 市场大的足以使 Microsoft 实现两方面相对独立的 Win32 工具软件，但并不适合于任何给出的平台，Windows 95 产生的 Win32 工具软件可适合于 Intel 80386。如果 Microsoft 没有一个版本适合于 Intel 平台，它将失去有关操作系统的领地（如 OS/2）。实际上，许多人认为 OS/2 和 Windows 95 非常类似，并且 Windows 95 是一个“OS/2 杀手”。

前一段时间，Microsoft 开辟了“Win32c”这一术语，看起来似乎是突出于 Windows NT 和 Windows 95 两者，以致于使程序人员有些混淆。之后，Microsoft 开始强调 Win32 API，并且强调为 Win32 API 书写的程序可以运行在所有的 Win32 工具软件上。然而，现实是程序编制人员仍然把 Windows 95 工具软件看成是 Windows NT（Win32）API 的子集，Microsoft 关心的是程序人员可能拖延使用 Win32 API，因为他们不知道这个子集的目的。后来，Microsoft 适图在产品上进一步迫使“只有一个 Win32 API”思想形式，支持 Windows NT 和 Windows 95 两者使用 Microsoft Win32。

当然，适图进一步超过各种 API 子集是一派谎言，子集不同执行的任务也不同。例如，开发人员发现获得 Microsoft Win32 Logo 程序是很困难的，因为 Windows NT 和 Windows 95 Win32 工具软件之间的不同，使得他满足 Microsoft 公司关于其产品支持两个工具软件的要求几乎是不可能的。再例，依赖多线程执行任务的程序不能运行在 Win32s 上，因为 Win32s 不支持多线程。为了使编写的程序有效，程序人员必须关注 Win32 子集和了解下面的操作系统。

Win32 操作系统的地位

为了阐明 Win32 平台的结构，我提出了一种 audio-system 比拟（见图 1-1），很好地描述了各种平台之间的相互关系。为了便于讨论，假设高频配套磁盘不存在，而盒式磁带是录制音乐的最好形式。

在我的比拟法中，Win32 程序就像盒式磁带，16 位 Windows 3.X 程序就像旧的 8 轨磁带。同样，Win32 操作系统就像一个盒式磁带机，可以录放盒式磁带，Windows 3.X 就像 8 轨磁带机。

假设这样一个剧目，如果你是一个音乐爱好者，并喜欢上成的立体声合成效果，你要买一盒高质量的盒式磁带，也就是说你要买 Windows NT。换句话说，如果你只有 8 轨磁带机，但你又想去放盒式磁带，你该怎么办呢？你或许为 8 轨磁带机去买一个适配器可使你去听盒式磁带。在 Windows 编程中，Win32s 等价于盒式磁带-8 轨磁带机的适配器，你把 Win32 程序插入 Win32s 适配器，然后放入 Windows 3.1（8 轨磁带机）。在使用这些磁带适配器时，你能作的事情是有限的。例如，你可用 8 轨适配器去录制一盘盒式磁带，而且，这声音的质量不会像盒式录放机的效果那样好。同样，Win32s 能干的事也是有限的，即不支持所有的 Win32 API，也没有

线程的那些特性。

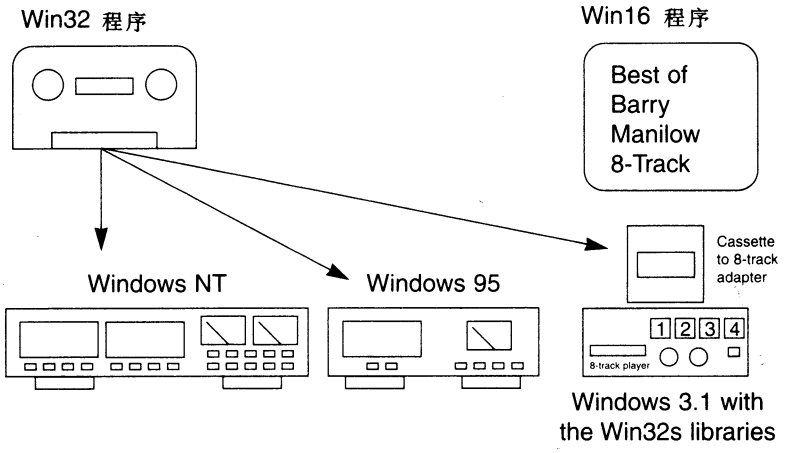


图 1-1　此方法描述了当前 Win32 平台的结构及相互间的关系

Windows 95 放那儿合适呢？Windows 95 等价于基本模式的具有 8 轨磁带机某些特性的盒式磁带机，Windows 95 有很多新的 32 位码，而且吸收了 Windows 3.1 类似视窗管理特性的码库，Windows 95 可以作 Windows NT 能作的大部分事情，但缺乏某些高层特性，比如不支持双字节字符集。从正面讲 Windows 95 是一个比较便宜的产品，换句话说 Windows 95 与 NT 相比，占据内存空间少且运行速度快。

暂时把这个比拟存放在你脑子里，让我们转回来看看微软公司的 Win32 工具软件相互之间的关系。

Windows NT 工具软件

Windows NT 的主要目的是可靠并可移植到其它平台上，编写平台的代码大部分是用 C 或 C++ 而不是用汇编语言，即便你正在看好 Windows 95、Win32s、Windows 3.2 或者是 DOS，这里强调 NT 仍是一较理想的平台，换句话说，可移植性和可靠性就是其代价。可使用 Windows NT 的机器最少是具有 16MB 内存的 486，即便是在这种机器上，NT 也不像运行 OS/2 或 Windows 95 那样快。（然而，Windows NT 3.5 版本比 NT 3.1 更有效，而 NT 3.1 是许多程序人员第一基本印象）。

NT 比较可靠的主要原因之一是他的保安子系统，在保安子系统中，执行 API 函数的操作系统码运行在不同的地址空间和进程中，而不是调用程序。Windows NT 中最重要的子系统是 Win32 子系统。Win32 子系统是他特有的进程，具有 USER 和 GDI 码，且大部分置于 DLL，叫 WINSERV.DLL。当你的程序调用一个 API 函数（例如 TextOut），你不必直接调 TextOut 码，实际上 NT 的 GDI32.DLL 把你

的参数复制到一个你的进程和 Win32 子系统进程均可达到的内存区，然后，你的线程向 Win32 子系统发出信号，当 Win32 子系统收到这个需要执行任务的信号时，将处理这些任务（比如把文字串显示到屏幕上），然后通知调用线程，此功能已执行完毕。这些子系统的运作模式也应用到 NT 支持的其它操作系统上，如 OS/2 1.x 和 POSIX。

保安子系统的优点是，地址空间较好的受到了保护，以防内存溢出和其它在应用程序码中出现的缺陷。操作系统（如 Windows 3.x 和 Windows 95）中没有这种子系统模式，操作系统码和数据被安排进所有进程的地址空间，使得有缺陷的程序有可能改写和破坏操作系统。该子系统的缺点是增加了运行时间。每一次调用理论上都要产生一个进程变换，同时内存就要产生变化，时钟周期消耗也很昂贵，每一次调用估计要耗费 2000-3000 时钟周期，由于这个原因，NT 开发人员允许大量的使用程序，以致不需要进程切换。此外，GDI 每一次调用均是匹配的，不需要进行进程切换。

所有可靠性的改进，对 Win32 应用程序是很重要的。但是，运行于 NT 下的 16 位应用程序是什么呢？16 位 Windows 程序运行在协同多任务模式下，并且期望能够访问属于其它任务的内存。NT 通过在分离的进程 WOW（Windows ON Windows）中运行 16 位应用程序保持该任务在可达范围之内。16 位 Windows 应用程序运行的单个“WOW box”，称为多线程 DOS box。

这个 WOW box 是一个 Windows 3.1 类“Sandbox”，在 16 位应用程序内部可以做任何事情，他们的活动不会损坏这个 box 之外的任何事情。

WOW 子系统与 Win32 子系统码通讯，执行显示输出，允许 16 位和 32 位 Windows 程序在同一屏幕上相互作用。Windows NT 3.5 采用了这一特性，在自己的 WOW box 中运行 Win16 应用程序，在扩展内存运行 Win16 的几种程序之间增加稳定性。

我一直介绍 WOW 子系统，因为他是一种很重要的结构，不同于 Windows NT 和 Windows 95。（Windows 95 运行 Win16 应用程序在同一地址空间，象 Win32 应用程序一样），NT 中的 WOW 子系统唯一的不足是，16 位 Windows 应用程序相对于 Windows 3.1 而言在同样的机器上运行比较慢。

Win32s 工具软件

与 Windows NT 形成对照，Win32s 是另一个动摇 DOS 和 Windows 大厦的，Win32s 本身不是操作系统，实际上是为 Windows 3.1 扩展的一组库函数，同样 Windows 3.x 自身也不是真正的操作系统，实际上还是要依赖于 DOS 的操作。

一些概念（如存储图文件），如果特有的一点功能不在 Windows 3.1，等价于 Win32 功能不在 Win32 子集，Win32s 码的好处是，从你的 32 位程序降到能执行同一任务的 16 位 Windows 3.1 码比 Thunks 不需要作多少工作。Thunks 是编程相对词，其意思是用橡皮糖、绳子和紧箍线把东西拼凑在一起，Thunks 在 Win32s 和 Windows 95 中是处理 16 位和 32 位码之间传输转换的。

Win32s 有很多的局限性，第一个也是最重要的一个就是不支持多线程，关于这一点 Nuff 说过。第二个不足是，对所有 Win16 和 Win32 程序都是单一地址空间。因为 Windows 95 和 NT 对 Win32 程序有单独的地址空间，这样就把 Win32s 归到了“尽量不用”类。Win32 程序运行在 Win32s 期间可以访问其它 Win32 程序的内存区，以及 16 位程序的内存区，这样内存区很可能被破坏。第三是 Win32s 缺乏 per-process DLL 数据。在 NT 和 Windows 95 中，DLL 的数据区是 per-process 基础。这就意味着，你可以放心地在 DLL 中使用全局变量，不用担心其它的进程调用 DLL 和重写这个变量的值。因为所有的 DLL 用户共享同样的数据区，你可能会遇到麻烦。很典型，你的程序及其 DLL 运行在 NT 或 Windows 95 中很好，但在 Win32s/Windows 3.1 中则不好，这又是忘掉 Win32s 的另一个原因，现在 Windows 95 问世了。

Win32 的其它问题(Windows NT 和 Windows 95 不存在)与进程调度和信息系统有关，在 Windows NT 和 Windows 95 中，线程被切换。此外，Windows NT 和 Windows 95 给每一线程以自己的信息队列，并且还要给适当的队列和输入系统线程分配空间和键盘信息，这两个设计因素允许一个线程无反应并花销不影响其它程序太多的时间。相反，Win32s 与 Windows 3.1 合成多任务模式展开竞争，为了执行一个任务，必须让出 CPU，这个工作通过调用一个函数来实现，如 GetMessage 或 PeekMessage。如果一个任务没有收回它的信息就让出 CPU，则用户就不能打开或使用另一个程序。

最后会怎样的，Win32s 有一个很不好的名声，如果你感到我认为 Win32s 有麻烦，这就对了。非常幸运，此时有了 Windows 95。

Windows 95 工具软件

Windows 95 Win32 工具软件最好的方面可以这样说，他严格地运行了 Win32s。另一方面体现了 Win32s 中 Windows NT 工具软件的最好特性，Windows 95 包括许多高级操作系统特色，实际上，视窗 95 存储标记与装有 Win32s 的 Windows 3.1 类似，使得 Windows 95 完全替代了 Win32s。

在底层，Windows 95 与 Windows 3.1 和 Win32s 有着很大的类似，与 Windows NT 则次之。与 Windows 3.1 一样，Windows 95 的最底层是由虚拟机器管理器(VMM)和相配的 VxD 构成的 ring 0 系统代码。理论上讲，运行在 CPU ring 0 级的编码是最稳定的。所以，与运行在 CPU ring 3 级的应用层编码相比，可以更多地访问硬件和操作系统数据。还有，像在 Windows 3.1 中一样，为运行 Windows 程序设置了一个虚拟机器，同时为每一 DOS Session 启动期间设一单独的虚拟机器。在 Windows 程序使用的系统虚拟机器中，你将会发现类似的 ring 3 系统 DLL (USER、KERNEL、GDI)以及等效的 32 位系统(KERNEL32、USER32、GDI32)。

Windows 95 像 Win32s 一样，大量的代码是在 16 位系统 DLL 中执行的，并且通过形实转换程序把 32 位程序转换成 16 位代码，几乎所有的视窗和信息系统码都驻留在 16 位 USER.EXE 中，试图把 USER.EXE 中大规模的复杂的视窗系统

代码转换成 32 位代码将导致大幅度增加容量，并且与当前 16 位程序不相容，这些问题 Microsoft 都没接受，因为用当前软硬件环境反向相容不值得去讨论。所以，Windows 95 中的视窗和信息系统基本上是 Windows 3.1 代码的改进版本，这些改进主要是允许 16 位和 32 位对接，以及增加了执行 Windows 95 支持的 Win32 函数所需的功能度。

Windows 95 中的 32 位 GDI API 的执行程序，置于 16 位 GDI.EXE 当前码和 GDI32.DLL 新码之间，只要有可能或合理，Windows 95 GDI32 函数就转换成可执行的 16 位 GDI 码，Microsoft 比较注重 KERNEL API，已经指出 32 位 KERNEL32.DLL 不能转换成 16 位 KRNL386.EXE，然而 Andrew Schulman 证明，在《Unauthorized Windows 95》中，KERNEL32 实际上能调用 KRNL386.EXE。本书后面的章节（特别是第三章）还要详细涉及这个题目。

我在前面描述了有关 Win32s 的一些问题，如无线程支持、单地址空间、缺乏 per-process 的 DLL 数据以及合成多任务。最重要的是 Windows 95 改进了这些问题，也就是说，他工作起来像 NT，Windows 95 32 位程序可以有线程（尽管不能执行 16 位任务）和 DLL 数据，然而某些棱角已被切掉，例如 Windows 95 中的每一个进程均有自己的地址空间，但是所有的装载系统 DLL 对一个 Windows 95 进程都是显而易见的。并不仅仅是该进程已经装载的 DLL 本身。此外，针对正在运行的 Win32 进程，所有 Win16 任务的存储和某些 DOS 存储量低于 1MB 是很显然的，换句话说，部分 DOS、Win16 程序和当前 Win32 进程全部混合于同一地址空间，这与 Windos NT 不同。正如第五章所表明的，Windows 95 中内存出错仍然是可能的。但是，一个 32 位程序破坏内存的可能性要比在 Win32s 下少的多。

关于 Windows 95 的热点之一是，在 16 位程序出现时不很平稳（not-very-smooth）的多任务状态，Windows 95 确实能执行有优先权的多任务，但是一个坏的 16 位程序可以引起其它线程在进入 16 位 DLL（如 USER.EXE 和 GDI.EXE）时跳走，这个问题 16 位系统 DLL 是不允许的，也就是说，在作某些事情的过程中不希望被断掉。由于许多 Win32 API 函数转换成系统 DLL，在不适宜的瞬间保护线程改变的一些方法是必要的，在 Windows 95 早期设计阶段，许多解决办法就被反复讨论过。

最终决定的保护办法就是 Win16Mutex，Win16Mutex 是一个相互禁止信号机，对那些需要进入 16 位系统 DLL（如 USER.EXE 和 GDI.EXE）的信号进行调解，Win16Mutex 的意思就是在一个时间内只允许一个线程执行 16 位系统码。尽管自身不会出问题，但为了防止其它问题，即是执行 16 位应用程序，Win16Mutex 也是固有的。不幸的是，不能通过调用 GetMessage 或 PeekMessage 适时放弃的 16 位程序，将阻止执行 32 位应用程序的用户接口线程。

Win16Mutex 有两层隐含，第一：如果一个系统不能运行任何有缺陷的 16 位程序，Win16Mutex 决不是困境源（source of trouble），请把你的应用程序转换成 32 位，越快越好，（正如在《Unauthorized Windows 95》中指出的，Windows 95 系统没有完全游离于 Win16 任务之外，因为系统本身就应用了一或二个 16 位程序。然而，这些系统程序可以很好地控制其它程序，并且释放 Win16Mutex 也理想）；第

二：如果你正在作 time-critical 工作，你希望把你的应用程序切入多线程（一个用户接口线程和一个或多个工作者线程）中，Win16Mutex 不会对那些没有转换成（像 USER 或 GDI）16 位码的线程有影响，即使整个用户接口与还未放弃的有缺陷的 16 位码程序有关，这些线程会继续执行。最可靠的方法是，在等待 Win16Mutex 时，通过调用 USER 和 GDI 函数中止你的 Win32 应用程序，由于设计先进，在你的 time-sensitive 线程中可以不调用这些模块。

Microsoft 之外的 Win32 工具

前三部分主要讨论的是 Microsoft 提供的 Win32 平台。然而，Win32 API 有效的定义和特性，使得其它公司完全可以使用它。例如，大部分人都知道 OS/2 Warp，IBM 也看到了 Microsoft 所有 Win32 操作系统的优点，尽管 Win32 API 直接与本公司的 OS/2 API 竞争，IBM 近期的很多 OS/2 版本，已经支持 Win32 API 子集。写本书的时候，OS/2 支持的 Win32 子集是 Win32s。毫无疑问，IBM 将来会完全支持 Windows 95 子集。

我最关心的是 Win32 与 DOS 的关系，尽管我主要运行的既不是 Windows 95 也不是 Windows NT，我频繁使用的仍是 DOS 和 Windows 3.1。当我作这些事情时，我就感到没有使用 Win32 API 来编写可利用的程序而恼火。幸运的是，当在 Win32 支持环境中进行操作时，不需要废弃你的工具，Phar Lap 和 Borland 都使 DOS 进行了扩展，完全可以支持 Win32 API，并允许控制模式程序在 DOS 或 Windows 3.1 下运行。如果你使用各种图形或视窗系统函数，DOS 扩展不一定满足需要，但是你更需要的是控制模式程序。

Phar Lap 的 DOS 扩展叫 TNT，Borland 的 DOS 扩展是跟随 Borland DOS Power Pack 而来的，使用这些扩展，你可以使用 Printf 和 fread 这些函数写 C/C++ 程序，而不要关心你的程序是在 Windows NT、Windows 95 下运行，还是在 DOS 下运行。

使用 Phar Lap 或 Borland DOS 扩展非常简单，你可以继续使用当前的 Win32 编译器而不要作任何改变，DOS 扩展可提供在 Win32 下运行的特殊程序，如果你在 Windows 95 或 NT 下运行 EXE，操作系统就会忽略这些程序，如果运行来自 DOS 的这些程序，则这些程序装载有 DOS 扩展，并装入 DOS 下提供给 Win32 API 子集的代码。

Microsoft 本身在第一次发布的 32 位 Visual C++ 版本中使用了 TNT DOS 扩展，由于为 Win32s 开发的程编人员没有 NT 机器，Microsoft 没有使 Windows NT 满足运行编译器（CL.EXE）和连接器（LINK.EXE）的需要。通过使用 TNT 扩展，NT 上开发的 Win32 控制模式应用程序在 Microsoft 工具运行起来，像为 Win32s 开发的 DOS 扩展应用程序一样没有任何问题。Borland 的 command-line 工具也是 Win32 应用程序，并且到目前仍在使用 Borland Power Pack DOS 扩展。

未来开发程序的考虑

如果你决定你的下一步计划是在 Windows 95 和 Windows NT 上运作，那么你的选择是很关键的，如果使你的程序能在 Windows 95 下正确工作，那么不用改变即可在 Windows NT 上运行。换句话说，Windows 95 不像 Windows NT 那么健全。花费在 Windows 95 重新启动的时间要比 Windows NT 上多(至少我是这样)。这个观点导致这样一个论点，NT 是 Win32 理想的开发平台。

不管是在 Windows NT 上还是在 Windows 95 上开发，这些选择似乎是与个人有关，有些人厌恶 Windows 3.1 风格而喜欢 Windows 95。相反，我喜欢 Windows NT，在我的所有工作中，在两年的周期内，使用 Windows NT 出问题仅有一二次(这不是因为 Windows 95 不稳定，关键是 NT 的设计更适合于硬件系统)。但是，尽管 Windows NT 稳定，我仍然在 Windows 95 上开发了许多程序，因为有些工具(如 SoftIce/W)只适用于 Windows 95 而不适用于 Windows NT。简言之，对这些问题没有很好的答案，作为 Win32 开发平台 Windows NT 和 Windows 95 两者各有特长。

Win32 的未来

本书出版的时候，Microsoft 正在致力于 NT 的下一个主要版本，代码名叫 Cairo。Cairo 将使用 Win32 API 并且被认为是面向对象的，Cairo 也展示了 Microsoft Post-Windows 95 关于用户接口的设计，因为 Cairo 将是一个 NT 代码库的修订版，它的平台独立性可通过增加代码容量和降低执行速度为代价来实现，在 Cairo 到来之前，Microsoft 或许增加有效性的赌注一直下在平衡机器执行时间和可利用存储空间上面。

Windows 95 不是 Microsoft 32 位 Intel-specific 操作系统的最后产品，Windows 95 设计或许只能有几年的生命力。如果硬件价格和性能有助于运行 Cairo，Microsoft 将会中止开发并行的两种 Win32 操作系统。换句话说，如果大多数用户硬件不支持运行 Cairo 代码库，Microsoft 肯定继续开发 Intel-specific Win32 平台，以保持他们的那份市场。

小结

这里结束我对 Windows 95 与其它 Win32 工具之间关系的旋风般解释，第二章的主题是关于 Windows 95 的，特别是论述了与 Windows 3.1 相比 Windows 95 中的新颖之处，本书的其它章节将深入到具体的内容，开展广泛研究。

第　二　章

什么是 Windows 95 的新特征

* *

近两年来，人们一直在思考着这样一个问题，Windows 95 到底是什么？一些人描述 Windows 95 像 NT Lite 一样——但是，Windows 95 不仅仅是 NT 的“light”版本。还有一些人描述 Windows 95 类似 Win32s——尽管在这两种操作系统中有极为类似的地方，这些描述也并不正确，Windows 95 比 Win32s 重要的多。

本章从程序和结构方面概述了 Windows 95，因为大多数使用 Windows 95 的用户先前都使用过 Windows 3.1，所以，我要用 Windows 3.1 作为各种比较的基础。

我描述的一些结构几乎均与 Windows 应用程序有关，讨论的主题将落在 KERNEL、USER 和 GDI 这三个紧密相关的函数上，这里给出的 Windows 95 决不是完整的，有许多题目——如 OLE 2.0，即插即用，MAPI(Mail API)，和 TAPI (Telephony API)——超出了本书的范围。

在本章中，我描述的 Windows 95 特征和结构，技术上采用的是 Win32 特征和概念，而不是 Windows 95 的特殊细节，这些特征 NT 中已经存在。然而，对许多程序编制人员而言，真正揭露 Win32 程序设计的，Windows 95 是第一次。我可以说，在许多地方用“Win32”或“NT 和 Windows 95”称呼 Windows 95，从技术上讲可能更准确。

正如我看到的，Windows 95 有两个主要特征：

- 提供一种具有 Windows NT 所有吸引人的特性(线程、分址空间、虚存等等)的 Win32 API 工具，没有类似保密和单码支持的 space-eating 特性。
- 在同样的一个具有 4MB 内存的机器上，运行 MS-DOS 和 16 位 Windows 应用程序，比运行 Windows 3.1 不错甚至更好。

第一个特征是指运行 Windows NT 所需要的处理能力和内存并不是每一台计算机都具备的，尽管 NT 是一个十足的“无害”操作系统，其资源需求超出了平均 4MB 内存的桌面 PC 机，Windows 95 带给用户一个具有 NT 性能的子集，可使用户不需要有 NT 的硬件，也不需要像 NT 或 UNIX 的高可靠的操作系统。由于有数千万台非 NT 能力的机器，Microsoft 把 NT 的性能移植到了 Win32 工具上，该工具软件可运行在桌式 PC 机上。而 Win32 API 层非常类似于 NT 和 Windows 95 之间的内容，Windows 95 工具与 80386 类的 Intel CPU 密切相关。这就使得 Win32 适应绝大多数机器，也值得耗资去生产两个操作系统。

第二个特征描述的也很明显，请注意 Microsoft 从不要求你在一个具有 4MB Windows 95 的机器上顺利地执行两个很大的程序。实际上，Windows 95 的目标是这样的：在具有 4MB 或更多存储空间的机器上装载相当的程序，Windows 95 不会比 Windows3.1 运行的差。我认为，在 4MB 机器上运行应用程序比在 Windows 3.1 上运行的更好是很少见的。然而，希望在 Windows 95 上运行的和在 Windows 3.1 上运行的一样好似乎是合理的（请记住，几乎每一个人都认为，8MB 是 Windows 3.1 所需的下限，所以运行的“一样好”不等同于“运行得好”）。由于 Windows 95 不能舍弃 Windows 3.1 的特征，很显然，Win32 相对于 Windows 3.1 要占用较多的空间。因此，设计 Windows 95 是一个折衷方案。

我把本章分成四个主要部分：

- Windows 95 和 Windows 3.1 如何一致
- Windows 95 如何改进了 Windows 3.1 的特性
- Windows 95 新特征介绍
- Windows 95 的“一点不足的小秘密”

本章只讨论 Windows 95 的改进综合情况，详细深入地讨论在以后章节进行。

与 Windows 3.1 的相似点

Microsoft 想使人们确信 Windows 95 是一个新操作系统。然而，你不应该相信别人告诉你的每一件事情。如果你作一点小变化（下面我还要说），你可以据理力争 Windows 95 系统实际上正在运行 Windows 3.1，因为 Windows 95 基本上是 DOS 和 Windows 3.1 代码库的改进。实际上，Windows 95 有许多新特征，我将在这里乃至全书中给以叙述。然而，为了真正地了解 Windows 95，我们把有关宣传放在一边，先来看看 Windows 95 的基本情况。

我刚才宣称 Windows 95 是在 DOS 和 Windows 3.1 的基础上改进的，现在停止阐述我本人的观点。作为第一个实验，让我们看一下开机时会发生什么（假设你已经安装了 Windows 95），在重新启动你的机器之前，我们作一个小变化，在你的启动驱动器的根目录有一个叫作 MSDOS.SYS 的隐含系统文件，如果你键入 dir/AH 命令，你会看到：

```
C:\> dir /ah MSDOS.SYS
 Volume in drive C is MS-DOS_5
 Volume Serial Number is 1CDE-9CF5
 Directory of C:\
MSDOS    SYS         1,641  07-17-95  9:40p MSDOS.SYS
         1 file(s)          1,641 bytes
         0 dir(s)      71,696,384 bytes free
```

如果此刻你使用的是 PC 系统并不为怪，然而在 Windows 95 中，这个文件已经有所变化，实际上是一个 ASCII 文本文件，让我们改变一下他的属性，使得他可被文本编辑器访问：

```
C:\> ATTRIB -r -h -s MSDOS.SYS
RHSA_ -> ___A_  C:\MSDOS.SYS
```

把 MSDOS. SYS 停在编辑器中将显示如下内容：

```
[Paths]
WinDir=C:\WINDOWS
WinBootDir=C:\WINDOWS
HostWinBootDrv=C
[Options]
BootMulti=1
BootGUI=1
Network=0
;
;The following lines are required for compatibility with other programs.
;Do not remove them. (MSDOS.SYS needs to be >1024 bytes.)
;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxa
... rest of file omitted...
```

你的文件或许稍有不同，但是你知道这点即可，现在我们在[options]部分加一行（“Logo＝0”）：

```
[Options]
Logo=0
BootMulti=1
```

下一步把文件保存起来，此时，你可能想把其属性改变回去，方法同前（＋r＋h＋s）。现在重新启动，假设你在安装 Windows 95 之后仍有 CONFIG. SYS 或 AUTOEXEC. BAT 文件，你将会看到，在 Windows 95 用户界面到来之前将执行那些文件的内容，在启动序列中的标识语，Windows 95 通常是显示出来的，而这些是不需显示的。很显然，在启动计算机的过程中，那些标识语主要是用来描述那些技术细节信息，而这类信息又很容易使末端用户混淆，眼不见心不烦，你说是吧。我们可以用“用户友好”这句话，对 Windows 95 来说大部分界面是不合适的。

很显然，DOS 仍要卷入这里的叙述，为了检查这个问题，我删掉 CONFIG. SYS 和 AUTOEXEC. BAT 文件并且重新启动，我们仅仅看到类似 DOS 的一些现象，在没有 CONFIG. SYS 或 AUTOEXEC. BAT 启动之后，我运行了 MEM/DEBUG 命令，看看在低于 1MB 的内存中是什么，输出如下：

```
Conventional Memory Detail:

 Segment            Total          Name          Type
 -------       ---------------  -----------   -----------------------
  00000          1,024    (1K)                Interrupt Vector
  00040            256    (0K)                ROM Communication Area
  00050            512    (1K)                DOS Communication Area
  00070          1,344    (1K)  IO            System Data
                                   CON        System Device Driver
                                   AUX        System Device Driver
                                   PRN        System Device Driver
                                   CLOCK$     System Device Driver
                                   A: - D:    System Device Driver
```

```
                                        COM1        System Device Driver
                                        LPT1        System Device Driver
                                        LPT2        System Device Driver
                                        LPT3        System Device Driver
                                        CONFIG$     System Device Driver
                                        COM2        System Device Driver
                                        COM3        System Device Driver
                                        COM4        System Device Driver
000C4                5,072     (5K)     MSDOS       System Data
00201               11,584    (11K)     IO          System Data
                     1,152     (1K)        XMSXXXX0 Installed Device=HIMEM
                     2,848     (3K)        IFS$HLP$ Installed Device=IFSHLP
                       688     (1K)        SETVERXX Installed Device=SETVER
                       544     (1K)                 Sector buffer
                       400     (0K)                 Block device tables
                     1,488     (1K)                 FILES=30
                       256     (0K)                 FCBS=4
                       512     (1K)                 BUFFERS=24
                       448     (0K)                 LASTDRIVE=E
                     3,072     (3K)                 STACKS=9,256
004D5                   80     (0K)     MSDOS       System Program
004DA                  192     (0K)     WIN         Environment
004E6                3,312     (3K)     WIN         Program
005B5                   32     (0K)     vmm32       Data
005B7                   16     (0K)     MSDOS       - Free -
005B8                1,152     (1K)     vmm32       Program
00600                  208     (0K)     COMMAND     Data
0060D                5,728     (6K)     COMMAND     Program
00773                1,312     (1K)     COMMAND     Environment
007C5                  240     (0K)     MEM         Environment
007D4               90,400    (88K)     MEM         Program
01DE6              532,896   (520K)     MSDOS       - Free -
```

图 2-1　MEM/DEBUG 命令显示的 DOS 段落(假定 DOS 在 Windows95 中执行)

如果 Windows 95 真正脱离了 DOS,我们不会看到任何 DOS 痕迹。在图 2-1 中,请注意下面的两行:

```
000C4                5,072     (5K)     MSDOS       System Data
00201               11,584    (11K)     IO          System Data
```

其中有 5K 被标注为 MSDOS,11K 具有 IO 名,当我们运行 DOS 和 Windows 3.1 而不是合成 Windows 95 时,这些或许是与 IO.SYS 的相关之处吧。让人们查看一下另一个在根目录中的内容:

```
C:\> dir /AH IO.SYS
 Volume in drive C is MS-DOS_5
 Volume Serial Number is 1CDE-9CF5
 Directory of C:\
IO       SYS       223,148  07-11-95  9:50a IO.SYS
         1 file(s)        223,148 bytes
         0 dir(s)      71,688,192 bytes free
```

实际上，IO. SYS 是一个大文件，尽管将近 220K，当装入我的系统时，只有 11K，这 11K 并不是很大的存储量，而且驻留在每个 Windows 95 系统中的 DOS 类代码也有很好的保护。

下面是图 2-1 中另外两个有意思的内容：

```
004DA              192    (0K)  WIN          Environment
004E6            3,312    (3K)  WIN          Program
```

这两行看起来确实是把一个叫 WIN 的程序装入了内存。等一会儿，在 DOS 提示符下，我们启动 Windows 3.1 而不是键入 WIN，让我们看看 WIN. COM 是否仍在 Windows 95 附近。

```
C:\>dir c:\windows\win.com
 Volume in drive C is MS-DOS_5
 Volume Serial Number is 1CDE-9CF5
 Directory of C:\WINDOWS
WIN      COM        22,487  03-14-95  6:44p WIN.COM
         1 file(s)         22,487 bytes
         0 dir(s)      68,542,464 bytes free
```

确实，WIN. COM 似乎仍在 Windows 95 中，再看看 WIN 程序进入内存之后，下一步会如何呢？Windows 95 中的 WIN. COM 与 Windows 3.1 中起的作用一样，也就是说，WIN. COM 脱离了实(或 V8086)模式进入了 Windows 环境保护模块。

我们在 DOS 区作最后一个实验，适合这样一条理论：在 CONFIG. SYS 文件中，改变 DOS 命令处理器(COMMAND. COM)为其它形式，恰好我有一个 COMMAND. COM 兼容的 4DOS，可替换有上述特性而不是 CMMAND. COM 的 JP-Software，现换成 4DOS(假设你有拷贝)，在 CONFIG. SYS 文件中加入一行：

```
SHELL=C:\4DOS.COM
```

当我作完这些，重新启动机器后，发现我面前是 4DOS 提示符，而不是 Windows 95 环境，似乎我的 4DOS. COM 版本不知道在 AUTOEXEC. BAT 文件处理完后去执行 WIN. COM，与 Windows 95 配套的 COMMAND. COM 则执行，这正是另一个天衣无缝的合成，这个直接进入 Windows 95 的透明启动，与在你的 AUTOEXEC. BAT 文件的最后一行加入如下一条的结果完全等价：

```
WIN
```

因为我们是在 4DOS 提示符下，我们可查一下正在运行的 DOS 版本：

```
C:\>ver
4DOS 5.0   DOS 7.00
```

怎么会出现 DOS 7 呢？这并不奇怪，DOS 的前一个版本是 6.X，对不对？如果你启动 Windows 95 COMMAND.COM，问同一个问题，你会得到如下的响应：

```
Microsoft(R) Windows 95
   (C)Copyright Microsoft Corp 1981-1995.
C:\>ver
Windows 95. [Version 4.00.950]
```

这就奇怪了，任何地方没有涉及到 DOS。实际上，Microsoft 并不希望非技术终端用户知道 DOS 混合于 Windows 95 之中。

我可以继续展示其它例子来表明 Windows 95 中存在 DOS 类代码，然而，Unauthorized Windows 95 在很多的细节上覆盖了这个主题，如果你在这方面有兴趣，可查看一下《Unauthorized Windows 95》。

现在看看 Windows 95 启动之后发生了什么，如果你在 WINICE（这是为 Windows 3.1 编写的）版本下装入 Windows 95，你将进入 WINICE，你自己都会相信你看到的确实是 Windows 3.1。例如，Windows 95（像 Windows 3.x）仍以 VxD 为基础，许多 VxD 系列仍存在于 Windows 95，如 VMM、VPICD、VTD、VDMAD、V86MMGR 等等（还有许多新的 VxD，以后再讨论），此外，你可以通过 SYSTEM.INI 继续装入你自己的 VxD，（然而，Microsoft 宁可让你通过注册去加入 VxD，某些事情后面还要讨论），执行 WINICE 中的一个 MOD 命令，也将会使你返回到 Windows 3.1 时期：

```
:mod
hMod PEHeader      Module Name       .EXE File Name
0117               KERNEL            C:\WINDOWS\SYSTEM\KRNL386.EXE
01C7               SYSTEM            C:\WINDOWS\SYSTEM\system.drv
01BF               KEYBOARD          C:\WINDOWS\SYSTEM\keyboard.drv
01CF               MOUSE             C:\WINDOWS\SYSTEM\mouse.drv
01E7               DISPLAY           C:\WINDOWS\SYSTEM\atim32.drv
036F               DIBENG            C:\WINDOWS\SYSTEM\DIBENG.DLL
023F               SOUND             C:\WINDOWS\SYSTEM\mmsound.drv
02EF               COMM              C:\WINDOWS\SYSTEM\comm.drv
042F               GDI               C:\WINDOWS\SYSTEM\gdi.exe
17FF               FONTS             C:\WINDOWS\fonts\vgasys.fon
1807               FIXFONTS          C:\WINDOWS\fonts\vgafix.fon
17F7               OEMFONTS          C:\WINDOWS\fonts\vgaoem.fon
17CF               USER              C:\WINDOWS\SYSTEM\user.exe
```

Windows 3.1 中所有的 DLL，在 Windows 95 中起着同样的作用。同样，一个 WINICE HEAP 命令将展示给你未加任何改变的 16 位内容。再有，我还可继续举例说明 Windows 95 的大部分功能与 Windows 3.1 所作的事情完全一样。事实上，毫无疑问地是 Windows 95 在 Windows3.1 的基础上已经改进了，是的，在某些方面是戏剧性的，如果你熟悉 Windows3.1，那么对你了解 Windows 95 是一个很好的开端。第七章详细叙述了 Windows 95 的 Win16 内容与 Windows 3.1 如何类似又不完全一样。

我阐明一下在本节我曾说过的一些事情，我认为 Microsoft 改进 Windows 95 是正确的选择，尽管 Windows 95 与 Windows 3.1 不百分之百的兼容，但比 Windows NT 或 OS/2 Warp 更好一些。对 Windows 95 而言，从一个全新的代码库开始并保持其兼容性必定是一件可怕的事情，同样，一个全新的代码库会存在某些不足，并且不能运行在普通 PC 这个大市场上的操作系统是没有前途的。你不得不称赞 Microsoft 面对了广大终端用户这一现实。

如果你是一个操作系统爱好人员，并且已转向 Windows 95，你可以为他作一些事情。运行一下 Windows NT、OS/2 或 UNIX，当你真正需要运行的程序在这些平台上不工作时，请不要抱怨，真诚地说，我对 Windows 95 结构的抱怨是惭愧的。当然我仍作为一个系统使用 Windows NT，我的观点是，Windows 95 和 Windows NT 两个都是有效的操作系统平台，你必需选择更适合于你的那个。

我认为今后几年内，Windows 95 和 Windows NT 都是非常重要的。正因为如此，我很注重这两个平台，那么为什么本书只涉及 Windows 95 呢？因为我认为在近期针对 Windows 95 的编程市场比 Windows NT 大。

在 Windows 3.1 上的改进

即便你不关心 Windows 95 的新特征，为了改进 Windows 3.1，将其升级为 Windows 95 是值得的。本节我将泛泛地给出改进的内容，详细内容后面章节再讨论。

DOS 几乎不存在了

尽管你没有注意到，在 DOS/Windows 3.1 结合的基础上，Windows 95 最大的改进就是把 DOS 替换成了 VxD，在 Windows 3.x 中，虚拟机器管理器(VMM)作为 DOS 扩展，当一个程序调用 DOS 去执行某些任务时(如从一个文件中读取)，那么 INT 21h 首先弹出到 ring 0 WIN386，然后中断靠近 Windows 下部运行的 16 位 DOS。在 Windows 95 中，一旦 VMM32.VXD 启动运行，几乎所有调用的 DOS 功能，均在 VMM32 中用全新的 32 位码进行处理。

把 DOS 放入 VxD 的最大的市场效益之一是，I/O 文件可用 32 位 ring 0 码进行处理。我说 DOS 时，所指的改进范围不仅仅是 DOS 程序。称为_Lread 的 Windows 程序执行 I/O 文件，最终结束是以同样的 VxD 代码形式，而 DOS 程序是通过调用 INT 21h 来完成。

为了向后兼容老的硬件设备，Windows 95 响应一定的临界中断到 Windows-95 下的实模式 DOS 码(实际是 V86 模式)。例如，当 VMM32 获知 DOS 设备驱动程序要被使用时，可以老的响应中断行为到 16 位 DOS 虚拟机器上，让设备驱动程序执行其任务。其它中断，如 DOS Get Time 功能，总是以实模式(V86 模式)DOS

码响应。需要记住的重要事情是，DOS 的大多数功能已经移入 Windows 本身的 32 位码，少量功能，Microsoft 使得 Windows 95 完全摆脱了实模式 DOS 码。尽管这些是操作系统热心人员呼吁的事，他的获取会降低与当前软件的兼容性，如果你希望这样，运行 Windows NT。

窗式系统

对一些程编人员来说，Windows 95 所提供的最大宽慰是 32 位视窗和图形部分的引进，在 Windows 3.0 乃至更早一些，所有的视窗和相关数据结构都塞进了 USER DGROUP，且最大限度是 64K。在 Windows 3.1 中，一些视窗系统的数据被移出放入另一个 64K 段中，但这仅仅是缓和了一下。在 Windows 95 中，USER 中的视窗码用了两个 32 位堆场来存储像 HWND 数据结构的项。作为结果，你不必再限制视窗的上限或在一个 Listbox 中只能有 8160 个输入。

虽然 Windows 95 视窗系统使用 32 位数据，你不能与 32 位码混淆，Windows95 中的所有视窗（即便是 32 位应用程序产生的视窗）也是被 16 位 USER.EXE 管理的。相反，Windows NT 小组是用新 Win32 码编写的 USER。对于 Windows 95 小组用 16 位 USER 码进行改进的决定，在程序编制人员中间是有争论的。然而有两个重要的特征，在第一章中曾讨论过，在下边的两段中概述了我的主要观点。

用 16 位 USER.EXE 保留视窗系统码的第一个原因是容量问题。有两个视窗系统码的拷贝，一个是 16 位而另一个是 32 位，要将几十万个字节的内容追加到 Windows 95 内存，在一个 4MB 的系统上运行是不能接受的，记住一点，Windows-95 不仅仅是为具有中高档硬件系统的开发者们的，Windows 95 需要运行在所有的旧式 386 系统上（当然公司最近把其内存升级为 4MB），你可以考虑一下，为什么 Windows 95 不把视窗系统码置入 32 位 DLL 并且调用他？

第二个原因是：16 位 USER.EXE 不容易移植，重要的部分是用汇编语编写的。此外，USER.EXE 是传统代码，无疑含有应用程序依赖于正常风格的特性。一个人完全依靠 USER 的工作模式是靠不住的，如果 USER 的代码完全转入 32 位码，目前有关应用程序将不能使用。

除了容量约束之外，Windows 95 百分之九十九点四四向后兼容，NT 中的视窗系统是 32 位，并且尽可能与 16 位处理器兼容。还有，Microsoft 并不要求百分之百的与 Win 16 兼容，Windows 95 保持着向后兼容的高标准。这就决定了用 16 位码来保持视窗系统。

讨论了上层原理后，让我们看一下视窗系统是如何适应 32 位应用程序的，我已经说过，USER 使用两个不同的 32 位堆场，但这仅仅是一部分，Windows 95 的 USER 在某些地方实际上使用的是一个 16 位和 32 位堆场的综合体。在 Windows 3.1 中，16 位 USER.EXE 内部还有一个具有局域堆场的 16 位 DGROUP 段，该局域堆场是有关基本量，视窗类和特征方面的项。然而，明显的不是视窗（更准确地说是 WMD 结构），USER 必须产生一个专门的 32 位堆场来保持视窗，对不对？我

说是对的，但这个故事还未结束。

如果你密切注意一下分配给 USER 中 DGROUP 的选择器，你会发现他的长度极限不是接近 64K，而是远远超过 64K。在 Windows 95，USER DGROUP 选择器的极限是 2MB＋128K，Windows 95 中 32 位视窗堆场在底层实际上已经包含了 USER DGROUP 段。考虑到这些细节，USER 使用的各类数据结构用一个选择器均可访问停留在正常 DGROUP 局域堆场中与这些数据项有关的 USER 码，在 Windows 3.1 中执行任务时可继续使用 16 位偏移，只有 32 位堆场中处理数据的代码（如 WND 数据结构）需要被改变去使用 32 位偏移，请记住，这些 32 位基址与 USER 数据结构的起始有关，不是 32 位线性地址。

除了新 32 位堆场存储视窗之外，Windows 95 的 16 位 USER 还有另外一个 32 位堆场来存储菜单和字符串，和 32 位视窗堆场不一样，在菜单堆场底部没有 16 位局域堆场。顺便说一下，中断相关菜单项进入一个分项堆场对 Windows 95 而言不是新鲜事物，Windows 3.1 有一个独立的菜单堆场，尽管仅仅是 16 位的。第四章中将详细叙述 Windows 95 中的 32 位堆场。

Windows 95 对一个 32 位堆的移位结果涉及到视窗柄（HWND），在 Windows 3.X 中，一个 HWND 是 USER 内 DGROUP 的一个局域堆柄块，因为 WND 结构存储在 LMEM FIXD 块中，局域柄不比偏移复杂。所以，把 USER DGROUP 选择器与一个 HWND 结合起来，一个程序可有一个直接指向 WND 结构的远距离指示器。Windows95 中不再使用这些，Windows 95 HWND 是些比较小的值（如 0x80、0x84 和 0x8C），这些值不是偏移量，实际上是 32 位视窗堆柄块，从内部讲，USER 能把这些句柄的一部分转换成 32 位偏移并且再转换回来。第四章叙述如何把 HWND 从 16 位 HWND 形式转换成一个 32 位指示器，并且再转换回来。

应用程序只了解自身部分，USER 已经改变了这种情况，他包含视窗类列表，在 Windows 3.1 中，所有的视窗类都存在一个链接表中，你可用 TOOL HELP ClassFirst 和 ClassNext 功能从该表获取类名及其模块。在 Windows 95 中，ClassFirst 和 ClassNext 仍在被使用，但是他们返回的信息仅仅是 USER 启动时记录的标准系统类，由应用程序记录的类被保存在专用存储器表中，其中一部分保存在 USER 的 16 位 DGROUP 内。但是，TOOLHELP.DLL 不知道任何内容。第四章还要详细叙述。

转换成信息系统

Windows 95 中，Microsoft 为每一进程提供了输入信息队列，实际上对每一个线程而言都有独立的信息队列。但重要的是，他不再是一个由系统内部所有任务共享的单一系统输入队列。为什么单一输入队列不好呢？简单地说就是迫使所有任务都从一个口获取用户输入，使他们易受损害。当一个任务被激活时，他有一个用户输入系统块，在该任务结束前，其它任务不能从中获取输入信息。

Windows 95（像 Windows NT 一样）抛弃了这种陈旧的模式，允许立即把信息输送到有关任务的输入队列。不幸的是，有争论的 Win16Mutex 使得 Windows 95

在某种场合下继续保持了 Windows 3.1 的特征(如通过调用 GetMessage 或 PeekMessage 不能用适当的方法放弃 16 位任务)。Win32 则没有这些问题,在信息处理过程中不会影响其它的进程。

Windows 95 向程序传送输入信息的方法是一种扩展的 Windows 3.1 模式,原始鼠标信息和键盘信息通过中断处理被传送到一个单一的系统队列,在 Windows 3.1 中,所有的程序都从这个单一队列中读取输入,并且当某一程序从该队列读取信息的时候拒绝其它程序使用。在 Windows 95 中(主要注重线程),当输入信息进来之后,Raw Input Thread(RIT)不监视这个队列,并为适当的线程传送到一个独立的输入队列。这样,既是一个程序没有被放弃,其它程序可以继续获取输入信息,当然仍然遗留着 Win16Mutex 和 16 位程序的问题,独立输入系统线程的益处主要适应于 32 位有优先权的列表程序。

尽管每一个线程有一个独立的输入队列,Win32 不准一个进程去改变另一个进程正在使用的数值和状态。在 Windows 3.1 中,USER 保留了许多作为全局值的视窗系统状态,其中有一单一的全局变量叫 HWndFocus,任何需要该变量的任务都可调用 SetFocus,这样就可以取走另一个应用程序的使用权(并且使得 HWndFocus 变量改变),对视窗占领和其它视窗系统状态而言也一样。在 Win32 模式下则不能,Windows 95 中每一个线程(并不仅仅是每一个进程)有其自身的视窗系统状态变量集合,当你调用类似 SetFocus 的 API 函数时,你激活的是当前线程状态而不是单一全局状态,存在每个队列上的状态有视窗捕获(Capture window)、视窗压缩(focus window)、视窗激活(active window)和光标。第四章详细叙述每一个队列视窗系统状态。

除了在每个线程上存储视窗状态外,Windows 95 USER 通常不允许一个线程去改变另一个线程的视窗状态。例如,如果你调用 SetFocus,让不同的线程队列拥有一个 HWND,你将收到一个警告信息,操作不会成功。通过 HWND 到 SetFocus,USER 可获得自己的视窗队列,与当前队列比较,USER 可以告诉你内部线程是否企图改变,从出现在其它 USER.EXE 版本中的消息进行判断,内部队列视窗活动也是不允许的。

当某一消息队列传递一个消息到 Windows 95 的一个视窗时,这个消息不会立即出现在目标视窗的队列中,实际上这个消息系统保存着一张消息表,只有当他们的出现可能影响 USER 的结果时才把他们分配给相应的队列。例如,一个任务无论何时进入 16 位进度表,Windows 95 首先把保存的消息分配给适当的线程信息队列,如果进度表不执行这条消息,那么下次也不会选择他运行。同样,调用 GetQueueStatus 迫使 USER 清洗掉这个临时消息列表。作为应用程序编制人员不要担心改变,正如 Windows 3.1 所作的那样 USER 承担了这些责任。

16 位和 32 位进程接口

Windows 95 视窗系统工具引起兴趣的一个方面,是 16 位和 32 位应用程序的视窗接口,尽管一个 32 位程序视窗产品是用 32 位码编写的,但 16 位应用程序不

知道或不关心这个，这些程序希望像在 Windows 3.x 中那样活动。现在，考虑视窗子集等一些事情，一个 16 位程序得到的图像为 32 位程序的视窗保持着一个 HWND，然后，这个 16 位程序通过保存原始的 WNDPROC 地址并置入一个新的 16 位 WNDPROC 地址将 32 位程序的视窗进行再细分，如果在 32 位视窗 WND 结构中 Windows 95 已经存储了一个 32 位线性地址，进展将是缓慢的，为了防止这样的问题，Windows 95 使得所有视窗特性似乎像 16 位视窗一样。

USER 作的大量额外工作是其另外一个方面，Win 16 单独控制信息的信息值(message number)以 WM_USER 开始并逐渐增加。此外，这些信息值的一部分与其它控制的信息值有重叠。例如，在 Win 16 中 BM_GETSTATE 信息被定义为 WM_USER＋2，这与 EM_SETRECT、LB_INSERTSTRING 和 CB_SETEDITSEL 一样，在这种情况下，除非你知道正在作用的控制类型是什么，否则信息值本身无任何意义。或许是为了使某些事情更一致，Win32 工具对一定的控制重新分配了信息值，以使低于 WM_USER 并不与其它单独的信息值重叠，重新分配的信息值如下：

Message group	Use	Win32 starting message
EM_	Edit controls	0x00B0
SBM_	Scroll bars	0x00E0
BM_	Buttons	0x00F0
CB_	Combo boxes	0x0140
STM_	Static controls	0x0170
LB_	List boxes	0x0180

如果在 Win16 和 Win32 程序中相同的信息有不同值的话，16 位和 32 位视窗如何才能进行通讯呢？USER 将会把这个信息转换成目标视窗适应的信息值，16 位程序之间的信息传递则不必如此进行。

然而，16 位和 32 位视窗联合工作的复杂性，似乎不仅仅是停留在采样信息传送上，许多信息是通过 WPARAM 和 LPARAM 参数进行传递的，通常一个 Win6 信息的 LPARAM 包括一个指向临时建立的数据或缓存的指示器，当一个 16 位程序通过 LPARAM 中 16:16 指针指向一个 32 位程序发送一条信息时会出现什么情况呢？重复一下，USER 转换需要产生 32 位视窗过程中代码可使用的信息，在这个例子中，转换器把 16:16 指针转换成了等价于 32 位线性地址。反之，32 位进程向 16 位视窗发送一条信息，一个 32 位线性地址必须转换成 16:16 指针，这种形势下，USER 为这一特定目的保持一种选择器：即改变选择器基本地址匹配于 32 位线性地址。这个选择器的限定值设为 0xFFFF 字节。

除了这些以外，Windows 95 视窗系统在 16 位和 32 位应用程序交换中间还要作参数变换。信息中的 WPARAM 参数需要进行变换，在 Win32 中，WPARAM 参数是 32 位，而 Win16 是 16 位，在一般情况下，当把一个 16 位信息转换成 32 位视窗可利用的信息时，USER 在 32 位 WPARAM 的字节置入一个 0，反之则从高字节去掉，本章不再继续讨论这个问题。

Win16Mutex

尽管 Windows 95 的线程调度表是 pre-emptive，但每个线程都影响他，一个 Win 32 进程可以产生许多线程，这些线程不调用 GetMessage 或其它与用户输入有关的函数。将 π 值(3.14159265...)计算到 50 位有效数就是一个线程的例子。这些不起用户接口作用的 32 位线程仅仅受 VMM32 中 32 位线程调度表的指配，既是由于用户接口线程拥挤使的内容满满的不能动弹，线程调度表会继续在这些线程之间进行 pre-emptively 切换，不幸的是，16 位任务不能引起更多的线程。所以，不会介入 pre-emptive 的多任务中去。

我刚谈到了用户接口线程的拥挤，那么我谈论的是什么呢？这些线程不被 pre-emptively 切换吗？回答这些问题将导致 Win 16Mutex 彻底完蛋，基于这一点，Windows 95 是一个旧 16 位和新 32 位码混合而成的事实在你心中会有很深印象。导致 Win16Mutex 产生问题原因是 16 位 USER 和 GDI 码不是用 pre-emptive 的多任务编写的，这些码假设不被任何理由中断，在某些确定的地方也会切换到其它任务上，还有一些全局变量贯穿于 USER 和 GDI 码，如果 Wimdows 95 忽略了这个问题，在 USE 或 GDI 调用期间就可能被切换掉。

针对 pre-emptive 多任务 16 位码没有准备好的问题不在于视窗码的限制，有几千种第三方 DLL(third-party DLL)也没考虑到用 pre-emption 编写，即是 Microsoft 用魔术来解决 USER 和 GDI 码，那些其它的 DLL 在关键时刻仍然会使系统受到线程切换的干扰。

解决 pre-emptive 切换问题的方法之一就是使 USE 和 GDI 中的所有有易损区与保护他们的同步机一致，关键是在 USER 上扩展同步机码将扩大其存储空间，因此是一个不可取的方案。在 NT 中没有这样严峻，视窗和图形系统有临界段(Critical Section)保护，所以可再次进入。

Microsoft 解决 pre-emptive 线程问题的办法就是 Win16Mutex，这是一个互斥信号器，当线程切换时可覆盖所有可能发生问题的 16 位系统区，Win16Mutex 覆盖所有 16 位码，因为大部分 32 位视窗和图形系统当调用 16 位拷贝时将被执行。即是执行相关用户接口或其它转换 16 位代码时，Win32 线程同样受影响，当 Win32 不作这些事情时，则不拥有 Win16Mutex 并继续保持 pre-emptively 调度表。

无论一个线程何时以 16 位码执行，均拥有 Win16Mutex，Win16Mutex 将阻止其它线程进入 16 位 USER 和 GDI 码，把当前线程移交给其它线程时，16 位线程通过调用释放函数(如 GetMessage)释放 Win16Mutex，其它线程获取 Win16Mutex 并继续执行。需要记住的重要事情是，16 位线程整个执行期间一直拥有 Win16-Mutex，不仅仅是刚调入操作系统那一刻。

而 Win32 线程拥有 Win16Mutex 只是在调入一定的操作系统函数时有，结果是，不放弃 Win16Mutex 的一个 16 位任务将阻止其它线程去获得 Win16Mutex，其它线程不管是属于 16 位还是 32 位进程，将一直被挂起直到 Win16Mutex 被释

放，一个未放弃的 16 位程序可以锁住其它程序(不管 16 还是 32 位)的执行。

既不处理信息又不释放的一种应用程序是 Windows 中的问题之一，Windows 95 的新特性就是有 Pre-emptive 多任务，然而 Win16Mutex 对任何必经通过旧 16 位码(USER. EXE)执行的代码而言，起着一个瓶颈的作用，反过来一个很次的(有编写错误的)16 位任务也能影响 32 位程序。

令人非常不快的是，未放弃的 16 位程序可以使用户输入系统停止，Win16Mutex 对于一个只包含 32 位程序的系统几乎没有一点问题，对于一个 32 位程序的线程也需要 Win16Mutex，但仅仅是在需要转换成 16 位码(如 USER 或 GDI)去执行用户接口时才要，USER 和 GDI 将迅速执行，然后释放掉 Win16Mutex，一般情况下，在有效时间周期内，没有 32 位线程永久保持 Win16Mutex 的，如果你担心 Win16Mutex 影响你的 Win32 程序，你可以生成一些不需向下调用 16 位码的线程。

Win16Mutex 在系统上的作用描述起来是很简单的，如果有 16 位应用程序运行，Windows 95 中多任务用户接口继续起着与 Windows 3.1 类似的作用，如果没有 16 位程序运行，则像 NT 一样，用户接口是平滑的，这里的含义是很明显的，即编写新的 32 位程序或尽可能将 16 位应用程序转向 Win32，对 16 位码要说“No”，对平滑多任务要说“Hello”。

Windosw 95 GDI

Windows 95 图形系统(GDI)是 Windows 3.1 16 位 GDI 和 32 位 GDI32. DLL 的新图形功能的混合品。一般讲，Windows 3.1 中的 GDI 函数已经保留在 Windows 95 的 GDI. EXE 中，像 Bezier 这些新函数等也加到了 GDI. EXE，像 True Type 和打印子系统等放到了 GDI32. DLL 中。像 USER. EXE 一样，Windows 95 GDI 也有直接使用 32 位堆的 16 位码，这些 32 位堆是用来存放区(region)和源(font)的。也像 USER. EXE 另一点，32 位 GDI 堆的前 64K 包含有 16 位 GDI DGROUP，GDI 客体不同于区(region)仍保留在 16 位 GDI DGROUP 内。

Windows 3. X 的众所周知的限制之一是，可使用的系统宽度极限是 5 屏 Device Contexts(DC)，如果一个应用程序使用于这 5 个 DC，其它应用程序则不能进行显示输出，系统经常显得不稳定，Windows 95 中这个限制已经有改进。

因为大部分 Windows 95 GDI 以 16 位码保留着，Windows 95 GDI 座标系统仍然被限制为 16 位。

Windows 95 保留 16 位码的另一个方面与设备驱动程序有关，当 GDI 在屏幕上或其它设备上显示输出时，GDI 要调进一个 16 位设备驱动程序 DLL，尽管 Windows 95 希望可移值的(PE)驱动程序，但 GDI 必须保持向下兼容(目前的显示和打印驱动程序是 16 位的)，许多使用 32 位指令的高性能驱动程序，还保留了 16 位新的可执行(NE)格式 DLL。

系统资源清除

Windows 95 将每一个 16 位任务作为一个单独进程来执行，原因之一是资源清除问题，原来的 Windows 版本没有标志出 USER 客体，当一个任务结束时，USER 没有任何固有资源的概念，所以在任务完成之后也不清除，重复执行一个程序系统则会溢出，结果是后面的程序不能运行。这个问题是 Windows 在一定市场范围还未被接受的主要原因。Windows 95 向前迈了一大步，可为那些进程分配资源，这些进程可以重新定位，当一个进程结束时，Windows 95 可重复其资源并清除中断的进程。

对这个改进资源使用模式，有一个意想不到的结果，在 Windows 3.1 中，一个任务可重定位一个资源并可处理给另一任务去使用，既是重定位的任务退出了，第二个任务可继续使用这个资源，新 Windows 95 在这种情况下则会适得其反，为了保持向下兼容，Windows 95 在清除结束的进程所占资源时，首先检查是那一种进程，如果是 16 位任务也未标志与 Windows 4.0 兼容，那么只要有 16 位应用程序运行，则不会删除资源。

如果 Windows 95 在清除资源方面作的再好一些，对自由系统资源(Free System Resources)有何影响呢？ Windows 3.1 中可以在四个堆中查看系统资源空间，其中有三个 16 位 USER 堆和一个 16 位 GDI 堆，自由空间比率最低的是 USER 报告的自由系统资源。由于 Windows 95 有 32 位堆，则计算需要改变，大部分情况下 32 位堆并未改变自由系统资源，因为这些堆有着比 16 位 USER 或 GDI DGROUP 更高的自由空间比率。然而，把 16 位 USER 和 GDI DGROUP 中空间消耗客体移出后，Windows 95 使得可利用资源减少了。第四章将详细讨论 Windows 95 中如何计算自由系统资源。

把内存消耗压缩在 1MB 以下

最后，我重点放在“insufficient memory to load this program(无足够内存装载这个程序)”这条信息上，好消息是 Microsoft 编码人员已经解决了“below 1MB”的问题，Windows 3.X 中 DLL 的 FIXED 段和 GlobalPageLock()的段是在堆的底部结束的，这就意味着低于 1MB，这些段落可以占用 1MB 以下的全部内存，这样可防止 Windows 启动其它任务，在 1995 年 5 月份的“Microsoft Systems Journal”刊物上，我详细论述了这个问题。Windows 95 中 FIXED 和 GlobalPageLock()段仍然在内存底部，但是在 1MB 以下，如继续增长，则应用程序不能运行，因在 1MB 以下无足够内存。

新产品特性

到目前为止，我们一直在讨论 Windows 95 即与 Windows 3.1 相似，又在

Windows 3.1 特性的基础上有改进。现在来检查一下 Windows 95 的全新特性，当然许多地方与 Windows NT 类似，然而对大部分程序编制人员和终端用户来讲了解这方面的情况还是第一次。

Windows 95 Win32 工具

以一个程序编制人员的观点，关于 Windows 95 最大的新闻就是 Win32 API，Microsoft 希望用一种可移植的(Portable)方法编 Win32 API 应用程序，理论上讲，不加改动可运行在不同的操作系统上(如 Windows NT)，只要支持 Win32API 和具备同样的 CPU 即可。也可通过简单地编译运行在其它 CPU 上，只要运行的是支持 Win32 API 的操作系统。如何使 Win32 API 适应于不同的操作系统则是今后几年的主要讨论内容。

当我第一次听说 Windows 95 支持 Win32 时，脑子中最大的问题就是“是像 NT 那样执行还是像 Win32s?”，使用其工作两年以后，我认为 Windows 95 严格地讲像 Win32s。Windows 95 的 32 位 DLL 可转换成相应的 16 位 DLL，大部调用给 Win32 视窗和信息的 API 函数都转换成 16 位 USER. EXE。同样，许多调用给 Win32 的图形函数则转换成 16 位 GDI. EXE。相反，Windows NT 完全是 32 位 USER 和 GDI 模块，运行在 NT 下的 16 位应用程序以及调用都要转换成 32 位 USER32 和 GDI32。

Windows 95 在执行过程中比 NT 更接近于 Win32s，但 Windows 95 的兴盛期要远远超过 Win32s，Win32s 的执行受到了 Windows 3.1 代码的约束，Win32s 的开发者们没有改变 Windows 3.1，因为 Windows 3.1 已安装在几百万台计算机上，仅仅升级为 Win32 支持的新的 Windows 版本不是一种选择。这样，相对 Windows 95 或 Windows NT 而讲，Win32s 受到了一些限制。

而另一方面，Windows 95 的开发者们则大刀阔斧地改变和修订了基本的功能，为的是更好地执行 Win32 API，在 Windows 3.1 代码基础上开始，ring 0 组成部分(VMM32. VXD 中的 VMM 和 VxD)和 ring 3 组成部分(KRNL386，USER 和 GDI)已经扩展修改为支持 Win32 系统 DLL(如 KERNEL32. DLL，USER32. DLL 和 GDI32. DLL)。实质上，Windows 95 有许多 NT 的特性，但使用的工具更接近于 Win32s。对广大用户来说，Windows 95 在速度、内存利用、特性和系统稳定性之间提供了最好的替换性。

仅仅因为 Windows 95 中仍有 16 位成分，不能意味着没注意新增加的 32 位内容，第六章将有更多的这方面的内容。

Windows 95 Win32 系统 DLL

Windows 95 的 Win32 API 是以 16 位和 32 位 DLL 混合形式执行操作的，表 2-1 列出了一些 Win32 API DLL 并说明如何被执行，不管什么原因，Microsoft 都试图使用已有的 16 位码。

这有两个益处，首先是 16 位码平均比等效的 32 位码要小，第二是 16 位 Windows 3.x 码已在全世界运行并被测试，那么重新用 32 位码编写一个像 USER 的 DLL 版本需要作大量的工作乃至推迟 Windows 95 的发布，用 16 位 USER.EXE 执行 Windows 系统是成熟的并且容易掌握，如果用 32 位码对 Windows 系统重新编码，就必经重新对 16 位版本的所有细节进行编码。

NT 开发人员选择 32 位码编写 USER，其代价是失去了与当前 16 位应用程序的兼容性，NT 的设计准则是允许这样的，Windows 95 则不允许，向下兼容是其基本原则。

表 2-1　Windows 95 32 位 DLL 选择的工具

DLL 名	用途	如何执行
KERNEL32.DLL	Win32/Windows 95 核心服务	一些在 VxD 中使用一些用于 KERNEL386.EXE
USER32.DLL	视窗管理器函数	大部分转换成 16 位 USER.EXE 有些在 USER32.DLL 中执行
GDI32.DLL	图形函数	大部分转换成 16 位 GDI.EXE
ADVAPI32.DLL	视窗记录	在 VMM.VXD 中使用
OLE32.DLL	OLE 2.0 bBase DLL	全部为 32 位码
COMDLG32.DLL	视窗对话	主要是 32 位，部分需要转换
SHELL32.DLL	shell(32 位)库	大部分是 32 位码，部分转换成 32 位
LZ32.DLL	LZA 文件分解	转换成 16 位码
VERSION.DLL	版本特征库	转换成 16 位码
WINMM.DLL	多媒体函数	16 位和 32 位混合使用

Windows 95 中 ring 0 的组成

现在我们深入到系统 DLL 的下面一级，看一看 Windows 95 中 ring 0 的组成，主要是 VMM 和 VxD，在 Windows 3.x 中这些成分全部并入 WIN386.EXE 文件，在 Windows 95 中这些成分仍然并在一起，但是名字是 VMM32.VXD，表 2-2 和 2-3 给出与 WIN386.EXE 相比 VMM32.VXD 中的标准 VxD。

表 2-2　Windows 95 的 VMM32.VXD 文件中的新 VxD

VXD 名	作用
CONFIGMG	结构管理器(即插即用)
DYNAPAGE	页管理器
IFSMGR	可安装文件系统管理器
IOS	I/O 管理程序

PERF	结构/状态信息
SHELL	SHEEL 支持
SPOOLER	局域假脱机
VCACHE	磁盘高速缓存器
VCDFSD	CD 文件系统
VCOMM	COMM 设备驱动器
VCOND	控制台设备
VDD	显示设备
VDEF	(不知道)
VFAT	文件分配表帮助
VFBACKUP	针对备份装置
VFLATD	FLAT 内存设备
VMM	虚存管理器
VMOUSE	鼠标设备
VPD	打印机设备
VSHARE	文件共享支持
VTDAPI	虚拟计时设备 API
VWIN32	WIN32 设备
VXDLDR	VxD 装载器

表 2-3　从 Windows 95 的 VMM32. VXD 文件中去掉的 VxD

VXD 名	作用
BLOCKDEV	块设备
CDPSCSI	SCSI CD 设备
PAGEFILE	页文件设备
QEMMFIX	固定对 QEMM
VDDVGE	VGA 显示设备
VFD	软驱设备
VNETBIOS	Netbios 设备
WDCTRL	Western Digital fastdisk
WIN386	取代 VMM
WSHELL	旧 SHELL 设备

其中最有趣的是 VWIN32 设备，实际上 VWIN32 不是真正的设备，而是 16 位 KRNL386. EXE 和 32 位 KERNEL32. DLL 中的 ring 0 码，用于执行一定的低级(low-level)操作。Windows NT 中最接近于 VWIN32 的是 NTDLL. DLL。其中包含许多低级操作系统的内容。

VWIN32. VXD 和 VMM. VXD 两个提供了可调用的 ring 3 函数称 Win32 VxD 服务，许多 KERNEL32 的操作是依靠 Win32 VxD 服务调用给 VWIN32 和 VMM 的，第六章要详细叙述 VWIN32 和 Win32 VxD 服务。

Windows 95 开发人员使内存消耗降低的一种方法是通过 VxD 结构的优点，Windows 95 支持动态装载 VxD，而 Windows 3. x 则必须在 Windows 启动时装载，且在整个期间保留在系统中，Windows 95 可在需要时随时装卸，就象一个程序需要打印时装入打印机驱动程序一样。新的 VxD 结构也支持可分页的 VxD，不使用的 VxD 部分可以被分离，一旦需要时再装入内存。

Windows 95 下的 Win32 程序不允许在代码中安装中断处理，也不能使用中断与其它码通讯，使用中断的大部分代码是为了与硬件设备通讯，Microsoft 建议你写一个 VxD 去执行这个中断码，你的程序可通过 DeviceIoControl 函数与 VxD 通讯，如果你需要调用一中断函数(如 INT 21h 或 INT 31h)，VWIN32 VxD 可提供给你子程序。

进程管理

Win16 中一个正在运行的程序被称为一个任务(task)，Windows 16 位 KERNEL 把每一个 Win16 任务的信息保持在一个叫任务数据库(TDB)的段内，任务数据库的选择器被认为是一个 HTASK，通过他可获知正在执行任务的 API。

Windows 95 中针对 32 位程序作的所有改进如何呢？对起步者，称一个运行的程序为一个进程而不是一个任务，每一个进程运行在自己的地址空间内，在“Windows 95 地址空间”一节中我还要详细叙述。现在我们来看看进程，他们可以看到自己的内存和操作系统，而看不到其它的进程或其它进程的空间，使进程相互之间保持分离的基本原因是防止有问题的进程影响其它进程。

在 Win32 程序中，给 WinMain 的 hPrevInstance 参数总是 0，不管其它程序是否在运行，一般情况下，一个进程自认为在系统中只有该程序在运行。当然，如果你确实需要与另一个进程通讯(或是去操作另一个进程)也是很容易的，这在编写你的代码之前就要考虑到。

每一个 Windows 95 进程在系统中被分配一个单一值。这个值称为进程 ID，一个程序可以通过 GetCurrentProcessId 函数获取自己的进程 ID，这个进程 ID 非常近似于一个 Win16 HTASK，NT 中的进程 ID 不分配给系统数据结构，因为典型的进程 ID 值是数字的(像 74、77、88 等)，Windows 95 中的进程 ID 值比较高且是随机的，然而在第三章你会看到，一个进程 ID 可以通过转换获取一个指示器，该指示器指向 KERNEL32. DLL 用于跟踪进程的进程数据库结构。

当 Windows 95 进程工作时，你不用使用进程 ID，实际上，大部分相关进程 API 函数期望一个 HANDLE 参数，通常称作 hProcess，hProcess 与某些事情(如 Win16 任务数据库)没有直接的关联，与进程 ID 不一样，可有多重独特的 hProcess 值，但都属于同一进程。

KERNEL32 对象句柄

句柄渗透着 Win32 API，一个句柄就是当需作某件事情时，从操作系统返回给 API 函数的一个万能值(Magic Value)，理论上讲一句柄值对应用程序是无意义的，只有操作系统知道如何去解释他，(几乎所有 Win16 程序的句柄值可被解释为选择器值或近指针)。

当用 KERNEL32 API 工作时，大部分句柄属于调用 KERNEL32 的柄簇，KERNEL32 柄有专门属性，比如可传递给象 WaitForSingleObject 这样的函数，KERNEL32 对象柄包括进程柄、线程柄、文件柄、Mutex 柄等等，第三章描述了各种 KERNEL32 句柄类型。

一个 KERNEL32 句柄只有在进程自身内部有效，企图将一个进程柄用于另一个进程是没有意义的，尽管句柄在理论上是透明的，对一应用程序而言，将一句柄转换成有用的对象指针是可能的，第三章 有如何将 KERNEL32 句柄转换成一个有用指针的叙述。

Windows 95 中最基本的进程函数是 CreateProcess，这是模拟 Win16 WinExec 和 LoadModule 函数，且这两个函数仍存在于 Windows 95，但其内部有些改变，如果你需要查询或操作后来的进程，你会愿意使用 CreateProcess，因可反馈给你一个 hProcess HANDLE。

因为 WinExec 和 LoadModule 没有 hProcess 和 HANDLE 的概念，不能返回 hProcess，实际上这两个函数调用 CreateProcess 以后，立即关闭了 CreateProces 返回的 hProcess，这样作的目的是防止为那联系紧密且无必要的进程分配系统资源。请记住关闭一个处理并不意味着结束这个进程，相反你还可通过特殊处理到该进程进行访问，当进程结束和所有的处理被关闭时，操作系统仔细地清除相关进程资源。

除了产生一进程获取一个 hProcess 外，另一个方法是用有效的进程 ID 去调用 OpenProcess，用 hProcess 你可以作一些基本的进程查询和操作。在进程控制的范围，一个程序可以用 TerminateProcess 终止另一个进程，用 SetPriorityClass 影响另一个进程的执行优先权。

学习一下 Windows mirror KERNEL 客体(如 16 位和 32 位两者中的有关任务和模块)是很有趣的，在进程的任务区，每一个 Win32 进程有一个 16 位任务数据库(TDB)，并把 TDB 连接到 TDB 链上，如果你用 TOOLHELP 流览这个任务表，你会看到除了这个 16 位任务以外，对每一个正在运行着的 Win32 程序也有一个 TDB，TDB 有 8 个字节的文件名，你可重新调用。

除了 TDB 以外，对 16 位或 32 位进程而言，对 Windows 中的所有 TDB(包括 Win32 进程的 TDB)还有一个 PSP，和 Windows 3.x 不一样，Windows 95 TDB 中的 PSP 没有必要跟着 TDB 立即进入内存，在 TDB 和 PSP 之间的 100h 字节内是存放当前目录区，这个区可有效地保存 Windows 95 支持的足够大的长文件名和

路径名目录，Windows 3.x 中当前目录存放在 TDB 内一个只有 65 字长的区域内，第七章将再详细地介绍。

线程管理

线程是 Windows 95 的新特性，一个线程就是一个执行程序的事例，线程允许一个程序同时在多于一个以上的地方被执行，这有些像多 CPU，每一个 CPU 执行程序的一部分。在单处理器系统中(Windows 95 只能支持单处理器系统)，只有同时处理时才出现线程，Windows 95 在系统中所有的线程之间切换 CPU，称为时间片(timeslicing)，因为硬件内部的计时器是以有规律的时间间隔通知操作系统的，而操作系统可以选择一不同的线程。顺便说一下，尽管 16 位程序作为一个线程出现在系统线程表中，但只有 Win32 应用程序在线程中能产生附加的线程。

一个线程被剔除有两个原因，一个原因是本线程需要另一个线程先执行，此时当前线程则把 CPU 给另一个线程。另一个原因是当一个线程执行了足够长的时间，需要把时间给另一个线程。Windows 95 线程调度使用的是这样一种算法，既把大部分时间给那些最急需的线程，CPU 在时间间隔期用硬件时钟中断操作系统，内部计时器中断处理，调度决定是否有另一个线程需要运行，如果要运行，则切换到另一个线程上。Windows 95 中的时间片是 20 毫秒，也就是说一秒钟内理论上可在 50 个线程之间进行切换。

每一个线程被分配给一个进程，当操作系统产生一新进程时，也要设置一个初始线程，一个进程中的所有线程共享该进程的资源(下面我要用“资源”一词来表示操作系统提供的内容，不指其它)，进程资源包括内存文本、文件柄和当前目录。

一般来讲，进程不交换也不使用其它进程的资源，然而一个进程中的多线程可能在进程资源的使用上发生冲突，这样资源共享可能是一个混合物。例如，你的程序有一组改变了几个全局变量的代码序列，如果一个线程正好在这个序列中间被切换掉，那么下一个线程将作用这些全局变量，而且与状态不一致。成功地执行多线程程序要求你标记出一个进程中所有的资源，这些资源需要由同步机进行监视，保证他们不被不适时宜的线程侵害，临界段(CriticalSection)和其它线程同步机在下面要进行讨论。

尽管线程共享进程资源，每一个线程还有一定的资源提供给自身，最重要的是栈。不，每一个线程本身没有 SS 寄存器和栈段，实际上，每一个线程在本身所在进程的地址空间内部有一个地址空间区，每一线程被分配的栈区隐含值是 1MB，这个容量要么置于可执行文件的.DBF 文件栈中，要么在调用 CreateThread 产生线程时指定一个非零栈区，Windows 95 对每一个线程栈不使用 MB，而是用“guard page(保护页)”，这在本章“结构异常处理”一节中详细讨论。

另一个虚拟全线程资源是线程寄存器集合，不管何时切换掉一个线程，在终止时操作系统则把线程寄存器的值复制保存，GetThreadContext API 允许你取回和改变一个线程寄存器，正常程序不需要作这些，读取和变更寄存器是调试人员的重要内容。

在操作系统内部，每一个线程有一个单值叫做线程 ID，象进程 ID 一样，Windows 95 线程 ID 有相对高的值但不是 32 位线性地址，然而大部分线程 API 函数不用线程 ID 工作，这些函数期待一个 HANDLE，通常称作 hThread，这个 hThread 只有在线程本身所处的进程内部才有意义，可有多个 hThread 值但都是同一线程的。

如果你开始注意在线程和进程之间的类似问题那就太好了，请记住：进程和线程 ID 在系统内是单一值，没有两个线程或进程可以有同一 ID 值，其句柄是不同的，每一个进程和线程可由 hProcess 或 hTread 句柄参照，这些句柄要么是不同进程或不同进程的线程，要么是自身的进程或线程。

进程与线程同步

对那些熟悉 DOS 或 Win16 程编的程序编制人员而言，Win32 程编中的进程与线程同步是一新东西。同步的意思是一个程序保证在不适时宜地被切换掉时不要出问题，虽然 Win16 有多任务，但没有真正的同步基础，因为这些多任务是协作多任务，一个 Win16 程序除非自愿放弃控制，否则不会被切换掉，自愿放弃控制是通过调用 API 函数（如 GetMessage 和 PeekMessage），如果一个程序调用了 GetMessage 或 PeekMessage，意思是说“现在我处在可中断状态”。

Win32 程序没有这样的协作多任务，他们必须作好随时被 CPU 切换掉的准备，一个真正的 Win32 程序不会耗尽 CPU 时间等待某些事件的发生，Win32 API 有四个主要的同步对象：

- Events　　事 件
- Semaphores　　信号器
- Mutexes　　互 斥
- Critical Sections　　临界段

除 Critical Sections 外，其余是系统全局对象并且与不同进程及相同进程中的线程一起工作，这样同步机也可以用于分离进程的同步活动（同一进程内部的线程除外）。

事件（Events）

这是同步对象的第一种类型，正如其名字的含义，在这个中心周围是一些发生在另一个进程或线程中的特殊活动。当你希望你的线程暂时挂起，等到所需要的活动发生时再恢复，此时你可以使用一个 Event，线程被挂起时不会浪费任何 CPU 周期。

程序可用 CreateEvent 或 OpenEvent 对事件对象获得一个句柄，然后该程序再调用 WaitForSingleObject，选定事件柄和暂停周期，那么线程就被挂起，一直到其它线程给出事件有关信号才被再次激活，其它线程指调用 SetEvent 或 PulseEvent 产生所需活动的线程，在事件获得这个信号后，被挂起的线程即被唤醒并继续执行。

例如，当一个线程要使用另一个线程的排序结果时，你或许希望去使用一个事件。比较糟的方法是执行这个排序线程并在结束时设置全局变量标志，另一个线程循环检查这个标志是否已设置，这将浪费许多 CPU 时间。用事件(Event)做同样的事情则很简单，排序线程在结束时产生一个事件(Event)，其他线程调用 WaitForSingleObject，这就使得线程被挂起以致不浪费任何 CPU 周期，当排序线程完成排序时，调用 SetEvent 唤醒另一个线程继续执行，有效地利用了 CPU。

除了 WaitForSingleObject 外，还有 WaitForMultipleObject 允许一个线程被挂起，在调用 WaitForMultipleObject 时，所挂起的线程要等所有事件(Event)信号发生后才能恢复，更有甚者，使用 MsgWaitForMultipleObject 挂起的线程，一直到要么满足 Event 条件要么有一个等待视窗信息时才能恢复，其它挂起的函数一直等到挂起的条件被满足或 I/O 操作已经完成时才能恢复，无疑这里有灵活性。

信号器(Semaphores)

这是同步对象的第二种类型，当你需限制访问特殊资源或限制一段代码到某些线程时，Semaphore 非常有用，这好比学校的全体人员通过的一个大厅通道，在给定的时间内只允许少量的学生在大厅通道内，如果你要去什么地方，并且目前所有通道都在用着，你必须等待。在 Win32 程编中，获得一个 Semaphore 就好象得到大厅通道的一次控制。

为了利用 Semaphore，一个线程调用 CreateSemaphore 去获得一个 HANDLE 给 Semaphore，该调用包括同时有多少线程使用资源或代码，如果这个 Semaphore 仅在一个进程内使用着，其它线程可通过全局变量去获得，如果其它线程在另一个进程中，可调用 OpenSemaphore 去获得一个可利用的 HANDLE，当一个线程需要访问共享资源时，要把资源传递给 WaitForSingleObject，如果这个 Semaphore 没有被等待的所有线程请求，等待功能将简单处理 Semaphore 的使用数且线程继续执行，换句话说，如果 Semaphore 已经超出最大值，调用等待功能的线程将被挂起，一个线程的含义就是使用一个 Semaphore 来执行，并处理给 ReleaseSemaphor。

互斥(Mutexes)

这是同步对象的第三种类型，Mutex(互斥)是“mutual exclusion”的缩略语，一个程序或一组程序希望一次只有一个线程去访问一个资源或一段代码时可使用一次互斥，如果一个线程正在使用这个资源，另一个线程则被排斥在同一资源之外。互斥的用法非常类似于信号器，产生、打开和释放信号器函数都有与互斥类似的内容，当一个线程有互斥要求时，可调用 WaitForXXX 系列中的函数。

临界段(Critical sections)

这是 Win32 同步对象的第四种类型，和上述三种不一样，他只能被同一进程中的线程使用，临界段是为了防止多线程同时执行同一段代码，相对其它同步机而言，临界段相对简单和易用，一个临界段可以被认为是仅在单一进程中有效的轻量级互斥，为了使用临界段，一个程序要么分配要么声明一个 CRITICAL_SECTION

类型的全局变量，在临界段首次使用之前，其场地需要通过调用 InitializeCriticalSection 进行初始化，之后调用 EnterCriticalSection 将一线程调入临界段，调用 LeaweCriticalSection 则是告诉操作系统另一个线程可以进入临界段了。

正如我所说的，临界段使用起来很简单，在 Windows 95 中，当没有其它线程时，如果一个线程调用 EnterCriticalSection，只需在 CRITICAL_SECTION 结构中调整和设置一些场地即可，只有已经存在临界段的另一个线程把 EnterCriticalSection 调入 VWIN32 VxD 时才能使该线程挂起。

WaitForXXX 函数

至此我已经概述了线程和进程同步的四种基本方法，我想谈论一下同步线程的其它方法，除了事件、信号器和互斥外，WaitForXXX 系列函数可接受几种其它的句柄，把一个进程 HANDLE 传到一个 WaitForXXX 函数则会引起线程挂起，如果这个进程已经终止，Wait 函数立即返回。同样把一个线程 HANDLE 传到 WaitForXXX，线程也将被挂起。

WaitForXXX 函数可以挂起的另一个 HANDLE 是这个文件的变更，文件的变更可以去限定一个给定的目录及有选择的子目录。WaitForXXX 函数的另外一个 HANDLE 是一个针对输入装置的 HANDLE 文件，一旦有未经使用的输入进入输入缓存，Wait 函数则返回并告诉线程继续执行。

模块管理

在进程和线程之后，我将告诉你 KERNEL 的概念是一模块，模块是代码、数据及可执行文件或 DLL 资源的内存形式，每一进程有.EXE 文件模块，由进程使用的每一个 DLL 也是一种独立的模块，如果两个或多个进程使用同一个 DLL 则共享同一 DLL 模块。同样，如果一个进程的两个复本正在运行，则两个复本共享同一个 EXE 模块。

Win16 中，一个新的可执行(NE)格式的可执行文件中的代码和数据产生一个任务，Win16 把这个可执行文件头复本保留在一个模块数据库的段中，该段的选择器称为一个 HMODULE，每一个 Win16 DLL 也有一模块数据库，因为 Win16 可执行程序和 DLL 共享同一个文件格式，win16 程序把 HMODULE 传给 API 函数，以便知道和你有关的可执行文件和 DLL 文件。

Windows 95 从一个可移动执行(PE)文件产生一个 32 位进程，PE 格式是 UNIX COFF(Command Object File Format)的改进形式，第八章将详细论述 PE 格式，这里不再讨论。

Windows 95 中最近似 Win16 模块数据库的是一个程序或 DLL PE 文件头(header portion)，每一个 EXE 或 DLL 的文件头存于内存，因为 Windows 95 要用内存映射文件装入程序代码和数据，在本章“内存映射文件”一节中再详细讨论，现在把内存映射文件看为内存的一块地方，在这里操作系统已读入了一个文件的一部分(或全体)。

一个 Windows 95 HMODULE 值不会比线性内存地址多(针对内存映射文件),一个 HMODULE 和少量计算给定后,你可把 HMODULE 转换成 PE 文件头的指针,利用指针一个程序可查找内存模块中的代码、数据和资源地址。

Win16 中 HMODULE 和 HINSTACE 不同,Win16 HINSTACE 是一个任务或 DLL DGROUP 段的选择器值,且频繁地使用于两个不同任务之间。在 32 位 Windows 95 进程中,HMODULE 和 HINSTACE 是同一个东西——内存模块的基本地址。

如果 Windows 95 具有 Win32 进程和 Win16 任务,则把模块信息存储在 16 位和 32 位两者的各自一边,每一个 32 位进程模块对应一个 16 位 NE 模块数据库,然而 16 位描述这些模块是最小的,并不是所有的 16 位 HMODULE 段的区域都是满的,我把这种最小的 HMODULE 叫做"伪 HMODULE",伪 HMODULE 并不出现在 16 位模块的链接表中,如果你想用 TOOLHELP 流览模块表,伪 HMODULE 不会给出任务显示,第七章给出进一步的叙述。

Windows 95 地址空间

Windows 95 和 NT 之间主要结构区别在于运行在同一虚拟机器和地址空间中的 16 位和 32 位应用程序。为了增加系统稳定性,NT 在一个叫做 Windows on Windows(WOW)的独立的虚拟机上运行 16 位视窗应用程序,NT 工具的趋势是将 32 位进程与 16 位进程、地址空间分开执行,进行 32 位与 16 位之间的转换是很艰苦的,许多有用的 16 位视窗 DLL 不会立即应用于 32 位形式。

从一个 16 位应用程序的观点看,地址空间是不变的,所有 16 位应用程序都从一通用局域描述表(LDT)中使用 ring 3 16 位选择器,这些程序通过选择器值继续与其它 16 位程序一道访问和共享内存。假设一个 16 位任务有一个指向其它任务内存的有效选择器,这个 16 位任务总能看到另一个 16 位任务的内存,一页内存可以被 VMM 进行标志,但接触这部分内存将会以透明方式返回,尽管 Microsoft 建议你在任务之间共享内存时用 GMEM_SHARE 分配内存,Windows 3.x 程序忽略了这个建议,Windows 95 下的 16 位程序可继续这样去作。

为 32 位进程描述的地址空间有很大不同,在 Windows NT 中,32 位 Windows 95 进程的独立内存是在 CPU 的页映射表中,当调度切换到另一个 32 位进程时,第一个进程的独立内存对任何其它进程均不可再访问,这样做的结果使得一个任务有可能去搞乱另一个任务的内存,不管是有意的还是无意的。

由于 Win16 任务是在共享内存区分配代码和数据,任何时间内,当前的 32 位 Windows 95 进程均可看到 16 位程序正在使用的所有内存。当然一个 32 位进程不能看到其它 32 位进程的内存,一次只能分配一个进程内存。16 位码可以看到所有共享系统内存,以及当前 Win32 进程的内存。

保护进程相互不干扰一般讲是好办法,但是有时你确实需要内存共享,进程之间共享内存的基本方法是用内存映射文件,NT 和 Windows 95 在可视文件映像方面的结构不同,NT 的内存映射文件仅供进程访问(调用 CreatFileMapping 和

MapViewOfFile)。此外,文件的内存区是以不同进程中不同虚拟地址为基础的。一个 Windows 95 程序一旦产生内存映射文件,其内存区所有程序均可访问。这样,一个 Windows 95 内存映射文件的虚拟地址在所有进程中是一样的,这无疑简化了 Windows 95 中的虚存管理代码,在"内存映射文件"一节中我还要谈更多关于这方面的内容。

初始的 Windows 95 应用程序省去了选择器的作用,Windows 95 在启动时用同样的代码和数据选择器初始化所有的 32 位程序,应用程序本身从不改变段寄存器(Windows 95 系统的 DLL 在转换成 16 位码时临时改变段寄存器)。例如,当我运行 Win32 应用程序时,每一个程序一个具有 0x013F 值的 ring 3 LDT 码选择器,该选择器基址为 0 且上限为 0xFFFFFFFF(4GB),所有 Windows 95 应用程序的数据选择器通常在 DE、ES 和 SS 寄存器中,这是一个低消费的选择器,其极限值低于 1MB。

直到现在,低消耗选择器几乎未变,其中原因是为了移植,低消耗选择器的限制是一个程序选择器可使用的最低偏置,最高可使用偏置是选择器可寻址的内存底部。Windows 95 的数据段选择器是一个基址为 0 的 32 位 LDT 选择器,这就意味着其有效地址在小于 1MB 和 4MB 之间。

所有 32 位进程使用同一选择器经常使熟悉 16 位程序的程编人员混淆,如何在两个不同的程序中使用同一个代码选择器呢? Windows 95 使用 CPU 的页分配特性把物理 RAM 映射成线性地址,每个进程有自己的一组页分配表,无论何时切换任务,均改变 CPU 的页分配表去响应新进程。这样,尽管两个程序有同一个选择器,在线性地址中有着完全不同的代码。因此,不知道属于那一进程的自身地址是没用的。

Windows 95 内存管理

大部分人认为,从表面上看 Windows 95 的 32 位内存管理结构与 NT 的非常类似。其实不然,KERNEL32 依赖于 VMM32.VXD 提供的内容去执行 Win32 内存管理 API,对于 16 位码,KRNL386 也是直接调进 VMM32 中的 VWIN32 VxD。Windows 3.1 的 KRNL386 完成许多同样的服务是通过使用 WIN386 的 DPMI 函数实现的。

对大部分整天都在进行编程的工作人员来说,最大的新闻就是 Win32 和 Windows 95 中没有更多的段,在转向 32 位程编过程中,你可不考虑远近指针,也可以把 GlobalLock、LocalLock 和任何与内存模块有关的东西忘掉,Windows 95 的 32 位程序中的每件事情都是小模块。当然,如果你希望用内存管理器去执行一项任务,Win32 API 和 Windows 95 有一套全新的函数将使你高兴。

Windows 95 中最低级的内存操作是由 VirtualXXX 函数提供的,该函数在第五章有详细描述。VirtualAlloc 可让你分配一具有 4K 粒度的大块地址空间。尽管有很大不同,Windows 3.1 程编中最接近于 VirtualAlloc(虚存分配)的是 GlobalAlloc(全局分配),两者都是分配大块内存的,你或许希望使用的并不是这些。

在用 VirtualAlloc 分配地址空间的同时，你可使用 MEM_COMMIT 标志有选择地把地址空间与物理 RAM 结合起来，为什么不让内存立即返馈一个地址空间分配呢？内存太少是主要原因。比如，你的程序需要大量内存，但事先又不知道有多少，你应该 VirtualAlloc 一大块地址空间。大的使你不再需要多的内存，当你的程序使用的地址域越来越大时，可通过再一次调用 VirtualAlloc 调整内存。更详细内容请看“结构异常处理”一节。顺便提一下，“仅需要时调整”算法是 Windows 95 执行大程序的精确方法。

比较高级的 Windows 95 内存管理是以堆函数形式出现的(第五章有叙述)，当 Windows 95 产生一新 32 位进程时，产生一个堆并隐含在地址空间内，该 32 位堆大体上与一个 16 位 Windows 局域堆相当，因为每个进程有一个堆。然而，这个 32 位堆实际上不限于 64K！Windows 95 支持多堆。所以，当你对一个堆内存块进行分配、解除或其它堆内存块操作时，需要把一个句柄传递结堆函数，一个程序用 GetProcesHeap API 到隐含进程堆抽取该句柄。

和 VirtualXXX 函数不一样，来自 Win32 堆函数的分配有非常小的粒度，每次仅 4 字节，在 HeapAllo()返回地址之间从 DWORD 函数得到。

当 Win32 进程进展时，仍有向下兼容问题，数以千计的 16 位程序使用 GlobalAlloc 和 LocalAlloc，移植起来容易吗？是的！Microsoft 在 Win32 API 中保留了这个重要的全局和局域堆函数，因此在转换成 32 位程序时不需要改变，然而意味着 API 及其工具有变化。首先也是最重要的，全局和局域堆函数基本上是一样的，你可以用 GlobalAlloc 去分配一个内存块，用 LocalFree 去释放该块。第二，全局和局域堆函数是用 32 位进程堆执行的，因为这个需调用

```
HeapAlloc( GetProcessHeap(),    // Heap Handle
           0,                   // Flags
           0x100 );             // bytes requested
```

如果你已经调用，应该返回同一指针。

```
Loc>alAlloc( LMEM_FIXED, 0x100 );
```

成功地执行完时，HeapAlloc 总是返回一个可用的指针，因此所有块分配等同于 LMEM_FIXED，Fixed 堆块有时导致分片。正如在 Win16 中，用一可移动的块柄去调用 Locallock 得到一个可用的指针，可移动块柄总有位 1 设置，因此其块柄值总是以 2、6、0xA 或 0xE 结束。还有，如果你想把一个可移动的块柄处理成象指针到指针一样，你能不依赖他去获得用块柄分配的当前块地址。在这里可以看到 Microsoft 强调的向下兼容性，因同样规划应用到 32 位堆。

局域堆函数和新 Win32 HeapXXX 函数的分配功能非常类似，HeapAlloc 和 HeapFree 替换了 LocalAlloc 和 LocalFree。同样，HeapReAlloc 和 HeapSize 接管了 LocalReAlloc 和 LocalSize，HeapCreate 与 GlobalAlloc 大体相当，Win16 没有与 HeapDesfroy 相当的。在 Win16 中，如果你用你产生的堆开展工作，你必须用 Glob-

alFree 放弃堆段，第五章有更详细的描述。

内存映射文件

Win32 和 Windows 95 最平淡的特性之一是内存映射文件，Win16 没有等同的内存映射文件，且 Windows 95 下的 16 位任务不能使用他们，内存映射文件在 Windows 95 中有三种用法，第一个也是最明显的用法，是用指针可从一磁盘文件中容易的读写数据，把磁盘文件的一段（或全部）映射到虚拟地址空间的内存范围，当你在这个地址空间区内向一地址进行读写时，操作系统在磁盘文件内读写同样的字节。

第二种用法有些像不同的 Win32 进程共享内存，一个进程可以为一个 NULL 文件设置一个文件映像，目的是保存一块地址空间但并不分配给实际的磁盘文件，然后其它的进程可打开自己的文件映像视区，连接这个映像地址区域的物理内存对其它进程或许是可视的，希望与第一进程共享内存的进程只不过是要求进入同一文件映射区，没有磁盘文件需要这样。

第三种用法是针对模块装载，当 Windows 95 的 32 位装载器需要装载一个可执行文件或 DLL 时，使用内存映射文件在进程地址空间分配可执行文件区，因为内存映射文件其它进程也可视，Windows 95 共享一个 EXE 或 DLL 的代码、数据及两个或更多进程的资源，相对说要容易和有效。使用存于 PE 文件中的值，Windows 装载器把各种可执行文件段分配到内存中特定的起始地址，第八章有更详细的叙述。

结构异常处理

最有用但还未被了解的 Win32 和 Windows 95 结构组成之一是结构异常处理，Windows 3.1 之前针对一个程序处理中断的 Windows API 没有规范的结构，Windows 3.1 引进了 TOOLHELP.DLL，这倒是往前走了一大步，可调用结构只有一点点进展。TOOLHELP.DLL 截取了一小部分但是有用的中断如断点中断（INT3）和 GP 故障（不含 13h），当一个异常发生时，TOOLHELP 的中断处理器开始控制，该处理器设置一牢固的栈结构并调用处理器函数。

TOOLHELP.DLL 允许许多灵活性，还为问题留下了许多空间，具有中断收回的每一个任务可以看到来自 TOOLHELP 的所有中断和异常，收回函数可通知 TOOLHELP 是否调用其它中断收回函数，这样，一个任务可以阻止另一个任务去看可能是相对独立的中断，如果这个中断收回处理器有问题，可能引起成堆的 GP 故障和其它系统破坏行为。对于 32 位进程，Windows 95 替换了这种方法，用一种更好的方法让进程去处理意外。

不是调试人员，为什么要处理异常呢？例子之一是，一个进程进行的操作可能引起 GP 故障或者是分母是零的除法，如果进程知道如何克服这种情况，则不会被操作系统中止。另一个例子是一个进程使用剩余内存，如果一个程序需要使用大量

的内存，但事先又不能精确地知道到底有多少，使用 VirtualAlloc 函数，程序可获得足够大的虚拟地址空间，当进程访问一页内存但未被物理 RAM 及时补上时，CPU 产生一个页故障，使用异常处理一个 Win32 进程可处理页故障，并且告诉操作系统在原故障指令处重新启动。

从技术上讲结构异常处理被置于操作系统并独立于任何特殊语言，然而结构异常在操作系统一级是非常复杂的，实际上在我写本书时，我并不知道关于这个主题的任何可利用的规范文件，因此，许多程编人员在编写编译器及运行时间库时主要面对的是结构异常处理，更详细内容请看第三章。

当一个进程不能处理异常时，则被传递给操作系统故障处理器，其结果是终止这个程序并进行资源清理。Windows 95 执行这个问题时使用的是独立的线程，即当一个线程突然发生意外时，线程的结构可能处于不稳定状态，通过在一个独立的线程中用一已知的正确结构对其进行清洗，Windows 95 开发人员希望能切断大量硬件系统的介入。

记录(registry)

在 Windows NT 之前，系统和程序都把其固有信息存放在 INI 文件中，Windows 95 使用一个记录把大量信息移入一个中心位置则是一个很大的进步。

在 Windows 95，那些要放入 Windows 3.1 的.INI 文件中的信息实际上被存入了这个记录，该记录是一个集合信息数据库(在图 2-2 中，Windows 95 REGEDIT 程序描述了这个集合)，最上层有一预定义“关键字”结点的小集合，每一个关键字结点下边有一些子关键字，在集合的任何一点，一个子关键字有一个或多个值，或者再附加一些子关键字，还有一个为追加、删除、修改和查询这个记录的 API 扩展集合。

Windows 95 顶层关键字有如下 6 个预定义：

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS
- HKEY_CURRENT_CONFIG
- HKEY_DYN_DATA

特别感兴趣的是关键字 HKEY_DYN_DATA，沿该关键字向下追踪几个结点有大量有用的信息，如子关键字 HKEY_DYN_DATA\PerfStats\StartStat 可使你得到一具有名字 KERNEL\CPUUsage 的值，同一关键字的另一个值是 VFAT\ReadsSec。

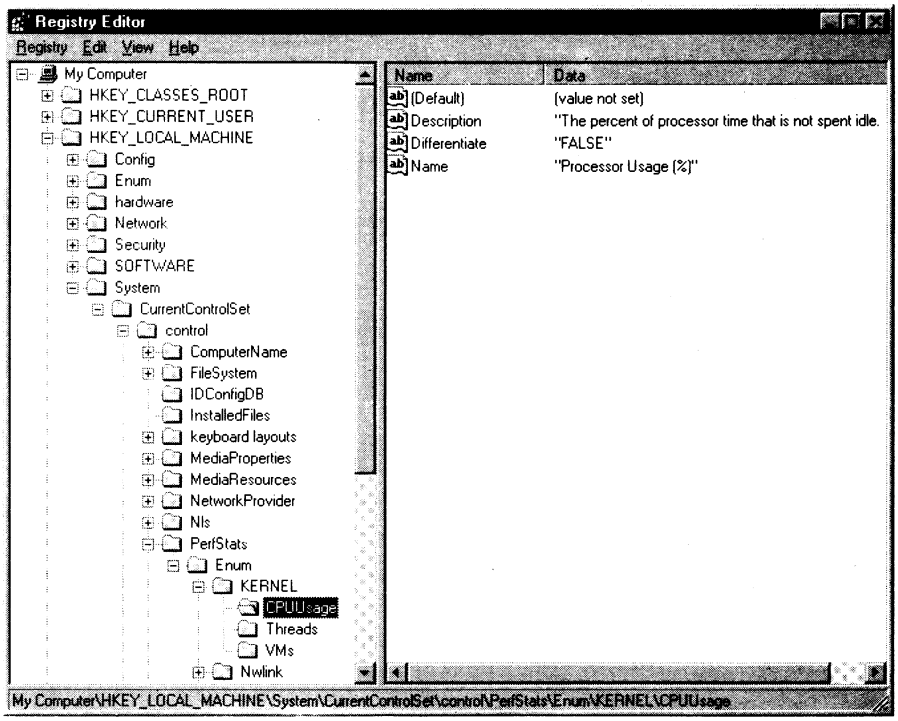


图 2-2　Windows 95 REGEDIT 程序显示的记录集合

这个记录实际上是在 VMM. VXD 中执行的，把该记录码置入已经装入的第一个 VxD，则该记录的信息可被 VxD 访问和使用，你可通过查看 Windows 95 DDK 的 VMM. H 文件看到这些，在该文件中，你还会看到其它 VxD 可使用的 VxD 服务：

```
// Registry APIs for VxDs
/*MACROS*/
VMM_Service (_RegOpenKey)
VMM_Service (_RegCloseKey)
VMM_Service (_RegCreateKey)
VMM_Service (_RegDeleteKey)
VMM_Service (_RegEnumKey)
VMM_Service (_RegQueryValue)
VMM_Service (_RegSetValue)
VMM_Service (_RegDeleteValue)
VMM_Service (_RegEnumValue)
VMM_Service (_RegQueryValueEx)
VMM_Service (_RegSetValueEx)
```

在 Win32 API 层，这个记录函数是在 ADVAPI32. DLL 中执行的，在 Windows 95 中该文件相对较小，深入内部你就会看到所有记录函数只不过是调用 VMM 记录函数的外壳。当然，因为 ADVAPI32. DLL 是 ring 3 码，不能直接调用 VMM 函

数，实际使用的是 Win32 VxD 服务，这也是 KERNEL32 为其它目的而使用的，这些 VxD 服务第六章进一步叙述。

给用户附加的内容

Windows 95 视窗系统新在那里？对初学者，Windows 95 给出了大量的新扩展的视窗风格，这些新风格包括：

种类	目的
WS_EX_MDICHILD	产生一个 MDI 子视窗
WS_EX_TOOLWINDOW	针对 TOOLBAR 视窗
WS_EX_CLIENTEDGE	视窗边缘
WS_EX_RIGHT	视窗文本被调整在右侧
WS_EX_LEFTSCROLLBAR	滚动棒在左侧

对许多开发人员增加的新东西是一个新控制集合，控制类型如下：

控制类型	目的
Animate	显示. AVI 文件
DragListBoxes	画表间的表框项
Header	Header 棒
HotKey	热键控制
ImageList	列表图像
ListView	列表视图
Progress	进度测量
Property Sheets	编辑特性
RichEdit	精美格式文本
StatusWindow	视窗状态
TabControl	列表对话
ToolBar	位图键
Tooltips	帮助
TrackBar	列跟踪
TreeView	树视图
UpDown	上下箭头增加/减少

和标准控制不一样，这些新控制不是运行在 USER. EXE，而是运行在 COMCTL32. DLL 和 COMMCTRL. DLL，结果是这些控制仅适用于 32 位进程。

系统信息和调试

Windows 95 执行的 Win32 调试 API 比 Win16 的规范的多，Windows 3.1 或 Windows 95 下的 16 位调试器使用 TOOLHELP 去设置断点和返回信息，通过查看这些断点信息流，调试器可知道正在干什么。然而，调试器返回信息需要把那些属于其它进程或本身不感兴趣的事件过滤掉，此外当调试器碰到一个断点或发生意外时，调试器的意外处理程序需要循环分类直到重新执行，一句话，16 位调试器比较凌乱。

Windows 95 调试 API 以 WaitForDebugEvent 函数为中心，在产生或连接一个进程之后，调试器调用 WaitForDebugEvent，把指针指向 DEBUG_EVENT 结构，WaitForDebugEvent 可返回的事件如表 2-4 所示

表 2-4　调试 WaitForDebugEvent 可返回的事件

调试事件	说　明
EXCEPTION_DEBUG_EVENT	告诉调试器断点、访问干扰及其他异常
CREAT_THREAD_DEBUG_EVENT and EXIT_THREAD_DEBUG_EVENT	保持对调试线程的跟踪
LOAD_DLL_DEBUG_EVENT and UNLOAD_DLL_DEBUG_EVENT	调试器为 DLL 装卸符号表
OUTPUT_DEBUG_STRING_EVENT	可使你看见输出调试的字符串信息
CREAT_ PROCESS_ DEBUG_ EVENT and EXIT_PROCESS_DEBUG_EVENT	告诉调试器正在调试的程序已经生成了另一个进程或者已经中断
RIP_EVENT	即便产生信息也不显示

使每一个调试事件发生联系的是，包含有关事件详细信息的结构，调试器可使用这些内容对运行着的 DLL 符号表进行装卸。OUTPUT_DEBUG_STRING_EVENT 对调试器开发人员更感兴趣，在 Win32 下这是查看输出调试字符串信息最好的方法，如果用其它方法，你必须使用 WaitForDebugEvent 函数在调试器下运行你的程序。在 Win16 中任何程序均可看到输出调试字符串信息，这就是 Win16DBWIN 程序所作的一切。

无论何时 WaitForDebugEvent 给调试器返馈一个事件，子进程的所有作用被冻结，调试器不必担心子进程的所有线程挂起，不管什么时候需要用这个事件去调用 ContinueDebugEvent 函数，都将使该线程重新运行。Win32 调试器的核心是循环调用 WaitForDebugEvent 和 ContinueDebugEvent 直到调试器接收到一个 EXIT_PROCESS_DEBUG_EVENT 为止。

除了了解有关事件外，还需一种进入调试寄存器和内存的方法，ReadProcessMemory 和 WriteProcessMemory 函数可以解决这个问题(请看第五章)。同样，Get-

ThreadContext 和 SetThreadContext 可以读写调试过程中特殊线程的寄存器。

除了提供中断和系统事件外，Windows 3.1 的 TOOLHELP.DLL 也提供一种反复进入各种系统数据结构的便利方法，Windows 95 这些数据结构（如模块、任务和堆）已经换成 32 位程序，Microsoft 执行的是 32 位 TOOLHELP，在 TLHELP32.H 中定义了 TOOLHELP32 函数，列表如下：

CreatToolhelp32Snapshot
Heap32ListFirst
Heap32ListNext
Heap32First
Heap32Next
Toolhelp32ReadProcessMemory
Process32First
Process32Next
Thread32First
Thread32Next
Module32First
Module32Next

这些 API 函数类似 Win16 TOOLHELP.DLL 函数，但确实又不一样，因此，如果你的 16 位码使用这些函数，还要做一点工作。Win16 TOOLHELP.DLL 与 KRNL386 是分离的，TOOLHELP32 函数是在 KERNEL32.DLL 中执行的。

在 Windows 95 中执行这些函数有一个问题，比如在浏览线程表过程中使用，则该线程可能被切换掉，在该线程返回控制之前，线程表可能换了，为了防止类似问题发生，TOOLHELP32 函数有一个“快照”概念，当你要浏览一个表（如进程表），首先调用 CreatToolhelp32Snapshot 生成一个快照，存入关于系统状态完整信息组成的一个缓存内，然后由 TOOLHELP32 再提取相关信息。

TOOLHELP32 函数明显的遗漏（与 Win16TOOLHELP.DLL 相比），是对于浏览视窗、获取系统堆使用信息和执行另一进程栈的跟踪等函数，然而在 Windows 95 中有其它一些方法可以作这些事情，在 1995 年 9 月《Microsoft Systems Journal》刊登了我的一篇文章，详细叙述了关于新 TOOLHELP32 函数的有关情况。

关于 Windows 95 的一点“设计垃圾”秘密

在结束本章之前，我想抛出一些 Microsoft 或许在任何时候都不会公布的设计缺陷和令人为难的信息。

现在我要谈及的许多许多问题在本章或在其它书中或在某些文章中的某些方面已经讨论过了，在这里我给出了以下内容。

- 实模式 DOS 码残存于其中；

- 共享内存地址空间几乎完全没有保护，Win16 和 Win32 应用程序可损伤过于灵敏的系统数据区；
- Win16Mutex 与质量差的 16 位任务一起可影响整个系统多任务；
- 尽管与主张不同，KERNEL32 确实调用 KRNL386。

再一次讨论这些题目，我希望把注意力放在 Windows 95 中其它有兴趣的问题上——那些到目前为止一直未引起极大注意的问题，下边列出的是本节的每一个主题的简况：

- 新 anti-hacking 码试图阻止你访问未标注的 KERNEL32 函数；
- Windows NT 和 Windows 95 小组之间缺乏协作和沟通，其结果是在 NT 和 Windows 95 两者中几乎没有多少 Win32 函数；
- 自由系统资源计算的变化似乎有较多的 USER 和 GDI 堆空间(尽管不是这样)；
- 16 位码中新增加的代码已经完成(尽管 Microsoft 公开声明执行的是 32 位码)。

Anti-hacking 码

《Unauthorized Windows 95》扩展使用了 KERNEL32. DLL 未标注的函数，这些函数虽然没有文件头，但确实出现在 KERNEL32. DLL 的引入库中，调用这些函数像提供一个原型和与 KERNEL32. LIB 链接一样简单。

在 Windows 95 后来的设计中——在《Unthorized Windows 95》发布之后——这些函数从 KERNEL32. DLL 的引入库中消失了，同时这些函数名字从 KERNEL32. DLL 的输出名字中消失了，但是，这些函数仍然在输出，不同的是仅仅依次输出。

现在看来这些事情对周围的工作有一些麻烦，如你要调用 GetProcAddress 要直接以函数次序(HIWORD 中是 0，LOWORD 是次序)进入和返回，正常情况下应该工作，然而在某些点 Microsoft 给 GetProcAddress 增加了一些代码，为的是看看是否以函数的次序在调用，如果是这样并且 HMODULE 转到的 GetProcAddress 是 KERNEL32. DLL 的 GetProcAddress，则 GetProcAddress 是不成功的调用，在 KERNEL32. DLL 的版本中调试器给出一条诊断信息："GetProcAddress：kernel32 by id not supported。"

现在让我们来这样考虑一下，由于未注明函数不是按名字输出的，你不能把一个 KERNEL32 函数名传给 GetProcAddress 去获得其输入点，同时 GetProcAddress 明确的拒绝你传给一个次序值，对这些事情负有责任的 Microsoft 确实不希望人们去调用这些 KERNEL32 函数，很显然调用这些函数的好办法就是有万能 KERNEL32 引入库(Mirosoft 没有提供给 Win32 SDK)。

绝不要担心，在本书的后面你会看到，我充分利用了 KERNEL32 未注明的函数，使用 V C++工具生成一个引入库，其中含有这些函数，附录 A 包含有这些函数和引入库信息。

置入 Windows 95 的另一个 anti-hacking 码的例子是 Obsfucator 标志，在早期的 Windows 95 版本中，GetCurrentProcessId 和 GetCurrentTheadId 把指针返回给

相关的进程和线程数据库结构，这些将在第三章给以描述，在《Unauthorized Windows 95》发布之后不久，这些函数返回的明显不是指针的值，调查结果返回的是顺序指针值，但是 XOR 内似乎是一随机值，这个随机值是每一次系统启动时利用系统时钟计算的一个随机值，有趣的是在调试 KERNEL32 时将该随机值起名为“Obsfucator”。注意 KERNEL32 编码人员误把“Obfuscator”拼写为“Obsfucator”，这是 KERNEL32 一开始为拼写检查带来的麻烦。

像 GetProcAddress 码一样，对 GetCurrentProcessId 和 GetCurrentThreadId 中的 XOR 没有任何理由，他不同于防止人们获取系统数据结构。当 Microsoft 试图隐瞒这件事情时，对那些真正需要这些函数的用户也未声明何时通报，在第三章描述了在运行时间计算 Obsfucator 值的技术，所以你可以访问线程和进程数据库结构。

Win32 API 隐患

Microsoft 希望你对 Win32 API 非常满意，但当时 NT 和 Windows 95 小组内部并没有进行沟通，其结果是使可用于 NT 和 Windows 95 两者的许多 Win32 函数遭受了不必要的损失，下面的三个“示例”可证明我的观点。

示例 1：关于新 Toolhelp32 函数。我已经听到了 NT 管理部门许多次声明，即：绝不执行他们。如果你再注意一下 TOOLHELP32 函数，就会发现函数量很少，而且主要是进程和线程算出函数，这些信息可以从 Windows NT 纪录内提取，就像 PVIEW 从 Win32 SDK 给出的一些简单演示一样，我的问题是：Windows 95 小组为什么不简单地执行 NT 提供的同样的记录信息，使得 PVIEW 也能工作？NT 小组为什么不在该记录函数的上部写一层并把 Toolhelp32 函数放在 Windows NT 上？如果双方都希望这样作，就应该结合一种 Win32 API 方法去执行系统信息算出。当我正要完成本书时，我从 NT 小组那听到了许多说法，TOOLHELP32 函数不久会出现在 Windows NT 的版本上。

示例 2：关于堆函数。有几个 Windows 95 不执行的 Win32 堆函数（虽然完成他们用不了多少时间）。主要例子是 Windows NT 的 HeapWalk 函数，该函数在 Windows 95 中是不执行的，如果再看看 TLHELP32. H，会同样发现两个函数：Heap32First 和 Heap32Next。这不同于简单地执行一个 Win32 API，Windows 95 编码人员完成了两个全新的函数，而 NT 小组说他们不打算支持这些函数，真是精神错乱！

示例 3：关于 HeapLock 函数。在 Windows NT 中，这些函数可简单地查询 Win32 堆中规定的内容，正如在第五章将要看到的，Windows 95 有一个功能完全相同的函数，然而 KERNEL32 开发组不输出该函数，那么 HeapLock 函数在 Windows 95 中不执行，最可能的原因是某些人不原意改变其名字并从 KERNEL32. DLL 中输出。

需要指出的是，Microsoft 试图使人们确信 Win32 API 是标准的，可 Microsoft 的两个小组仅仅是按照各自的意愿在进行，这只能损伤 Microsoft 的利益，我已经

归纳了我的部分 WINBUG 报告并且发了许多电子邮件(e-mail),现在让市场决定,看看假想标准的 Win32 API 会发生什么。

自由系统资源的粗制滥造

在启动 Windows 95 之后,如果立即停止 Windows 95 测试程序(Explorer),然后到 HELP/ABOUT Windows 95 对话框,你会看到一个相当高的自由系统资源值;典型值是 95%,这要比你在 Windows 3.1 下看到的值高的多,Windows 95 偶然获得了 16 位 USER 和 GDI 堆计算的全部空余内存了吗? 不! 实际上许多新内容加到了 USER 的 DGROUP 段,如果有区别的话,Windows 95 中自由系统资源已经下降或者是一样的。

正如我在第四章中要描述的,在 Windows 95 连续启动期间,测试程序使视窗去计算 Windows 3.1-like 的自由系统资源值,以后所有调用 GetFreeSystemResources 都以这些初始值偏置。这样,当测试程序报告可利用系统资源有 95%时,则意味着测试程序和其他程序已经进入内存之后资源的 95%。这种变化只不过是让非技术人员认为 Windows 95 比 Windows 3.1 好罢了。

Win16 仍然有生命力

尽管 Microsoft 正在极力推动每一个人员转向 Win32,但是支持 Win32 API 的大部分基础是 16 位码,这没有什么秘密,不值得再提了,然而 Microsoft 对加到 16 位 DLL 上的新 API 函数一直没有任何说法,在许多情况下,这些函数是等效于 Win32 API 的 16 位码,我正在讨论像 CreateDirectory 和 GetPrivateProfileSection 这样的函数,在某些情况下,这些函数被悄悄地加到 16 位 WINDOWS.H 中,其他情况下,从 16 位 DLL 中输出这些函数,但是在有关的.H 文件中没有样本。

如果 Microsoft 不公布这些增加的内容,那么谁来使用他们呢? 如果每个人都使用 Win32 码,为什么 Microsoft 还要增加新的 Win16 API 呢? 看起来 Microsoft 知道 Win16 在 Windows 95 之后还有相当长的生命力,又告诉人们可继续使用 Win16,我个人同意用户们应尽可能的把注意力放在 Win32 编程上,但是强迫他们面向 Win32 编程似乎不是好办法。

小结

确切的说 Windows 95 是他自己的操作系统,Windows 95 的大部分代码是从 Windows 3.1 的基础上导出的,Windows 95 的 16 位码已经进行了改造,目的是剔除许多 16 位限制以及处理 Win32 多进程的要求,Windows 95 既不是 Win32,Windows 95 有线程和多地址空间,结构比 Win32 多,Windows 95 也不是 NT Lite,Windows 95 的代码执行上是优化的,并且在 Intel X86 CPU 内存上消耗最小,NT 的注意力是可移植性和耐久性,虽然 Windows 95 和 NT 的结构在某些关键地方不

一样，二者在 Microsoft 的操作系统战略上同等重要——并且在未来几年仍然是重要的。

第 三 章

模块、进程和线程

* *

大多数人都有自己的爱好，我自己最喜欢数据结构。事实上，我已经收集了三个组成 ring 3 Windows 95 核心的数据结构。这三个紧密联系的结构是模块、进程和线程，当它们被作为一个整体的时候，与它们没有接触上的 Windows API 功能很难被发现，不相信吗？看一下 ShowScrollBar 函数，第一个参数是 HWND 每个 HWND，对应着一个消息队列，而且你还会看到，在 Windows 95 里，每个消息队列对应着一个线程。因此，ShowScrollBar 代码的某些地方将需要线程数据结构里的信息。

在本章我们将研究模块、进程和线程这三个核心数据结构，同时还经常会遇到一些辅助数据结构。例如，每个进程包含一个指向进程表的指针（很象 DOS Program Segment Prefix[PSP]里的进程表）。当查看进程表时我们会遇到 KERNEL 32 对象。同样，当查看线程时，要想忽视线程信息块（Thread Information Block-TIB）的存在是很困难的。TIB 在结构化特殊处理上是必不可少的。

本章充满了信息：除了对三个关键数据结构的描述外，我还加入了各种 Win32 函数的伪码，这些函数直接与数据结构有关，他将使你很方便地了解到这些数据结构的运转，例如了解 KERNEL 32 是怎样处理线程同步这样的重要问题。本章的最后我还进行了关于 WIN32WLK 程序的讨论。WIN32WLK 是我编的一个帮助我学习这些关键数据结构的程序，它可以使你轻松的浏览系统内所有的进程、线程和模块，且可方便的检查个体数据区。另外，线程数据库包含一个指向自己的进程的指针。

如果你是一个 Windows 3. x 的程序编制员，你可能已经很熟悉模块和任务的概念了。在 Win32 里，任务的概念被分成两部分，即进程和线程，更有甚者，Win16 和 Win32 的模块和任务进程概念表面看来完全一样，事实上它们完全不同，Win32 的模块数据库没有与 Win16 的模块数据库相同的地方，Win16 的任务结构也没有一点看起来与 Win32 的线程或进程数据结构相同。

在 Windows 95 设计中有一个有趣的地方，这在 Windows NT 里没有发现，它就是 Win16/Win32（是 16 或 32 位）信息的“镜像映射”，Windows 95 中每个启动程序表现为一个 Win16 任务和一个 Win32 进程，你可以通过 Win16 TOOLHELP. DLL 浏览任务表并且在表内看到 Win32 程序，同样你可以通过 Win32 TOOLHELP32 函数浏览进程表并且在表内看到 Win16 程序，除了任务（或进程）映射，

Windows 95 还含有 Win16 模块的每个已装载的 EXE 或 DLL 的信息，而不管是 16 位还是 32 位的。不幸的是，Win16 TOOLHELP. DLL 不能“看到”Windows 95 为 Win32 模块设置的 Win16 模块数据库。还好，第七章的 SHOW16. EXE 程序可以看到它。本章和 Win32WLK 侧重于 Win32 方面，而第七章和 SHOW16. EXE 从 Win16 方面做了描述。

在深入论述模块、进程和线程的细节以前，我必须指出，这个方法没有得到 Microsoft 的承认，Microsoft 不想让你把这些数据结构的信息和你自己的代码联系起来。它们对待有关模块、进程和线程的申请的一贯作法，是利用 TLHELP32. H 中的 TOOLHELP32 API。

TOOLHELP32 函数提供有限的模块内的信息区段访问，这些信息在模块、线程和进程数据结构里，这些数据结构被 Microsoft 认为是“安全的”以便让你使用。有必要说一下的是这些访问是只读访问。象很经常的事，所有 Microsoft 认为是“安全的”东西远远达不到象我这样一个系统级程序编辑人员的需要。例如，TOOLHELP32 没有提供列举进程柄表的方法，如果你需要细节，你不得不直接进入并得到信息，就象 Win32LWLK 程序做的一样。但是，如果你完全能用 TOOLHELP32 替代直接获取数据，你最好还是用 TOOLHELP32。

Win32 模块

和 Win16 一样，Win32 模块描述的是数据、代码和已经被 Win32 装载器装入的 EXE 或 DLL 资源。因此，存储器里的每个模块直接和磁盘上的某个文件相联系，EXE 和 DLL 本身不是一种模块。当然，装载器把信息从文件读到存储器并且依此信息建立模块。一个 Win32 Portable Executable(PE)文件的好处是将它们装入存储器的过程比较简单。装载器通过用存储器变址文件将 PE 文件的被选中区域映射到线性存储器来建立模块。(重点：不论人们信不信，装载器不是简单地把所有 PE 文件映射到一个存储器上)。操作系统把所有的有关装载模块的顶层信息保存在一个我称为“模块数据库”的结构里，第八章详细的描述了 PE 头标和模块数据库。

当提到装载模块时，一般用 HMODULES(handles to modules)，在 Win16 里，HMODULES 是包含 16 位模块数据库段的全局堆柄，(第七章详细介绍 Win16 模块)，在 Win32 里，没有段(至少没有那众所周知的程序)，所以需要其它方法来装载模块，Microsoft 的方案是使 HMODULES 做为存储器的初始线性地址，在这里 Win32 装载器内存变换 PE 文件地址。例如，许多 EXE 程序被 Win32 装载器装在 0x400000(4MB)地址，所以它们的 HMODULES 是 0x400000。是的，这意味着当不同的 EXE 同时运行时可有同样的 HMODULES，这不成问题，因为 Windows 95 和 NT 针对每个进程都有分地址空间。Win32 HMODULES 仅在已安装了模块的进程文本中有效(第五章将详细介绍进程文本)。

模块数据库的位置离装载 EXE 或 DLL 的内存的起始位置很近，它含有装入内存的文件代码和数据段信息，模块代码和数据比编译器把你的程序生成的代码

和数据要多。模块中其余的数据区是引入表(import)、引出表(export)和目录资源，引入表(通常是数据段)通知装载器不应只装载模块需要的 DLL 等文件，还应装载一些单个函数，引出表正好相反，并告诉操作系统模块输出函数的地址(可能是名字)。资源段含有一个类似目录分级结构，系统用它来快速查找特定的资源。不管是控制台模块应用还是其它什么，象操作系统要求的版本一样，模块数据库含有查找这些表的信息。

戴上我们的眼罩，点上乙炔焊枪，让我们切开模块数据库，看看 Microsoft 一直试图向我们隐瞒的东西。奇怪，奇怪！模块数据库的格式一目了然，就在我们鼻子底下。

在 Win32 里，模块数据库不过是 EXE 或 DLL 的 PE 头标，看一下 WINNT.H，你会看到 IMAGE_NT_HEADERS 结构，它是由一个 DWORD 和两个底层结构组成的。IMAGE_NT_HEADERS 结构里的信息是 Windows 95 用来在内部寻找已装入的 EXE 或 DLL 文件的代码数据和资源。

虽然我能以好多篇幅讲 IMAGE_NT_HEADERS 结构的细节，但我不会这样干，为什么?因为 IMAGE_NT_HEADERS 结构和 PE 文件的细节还得用来写它们自己那几章呢。(如果你看了目录并决定跳过有关 PE 格式的第八章，你该进一步认为，这里我没简单的描述该章有关内容是因为我喜欢剖析文件格式)。

Win32 要求每个进程有它自己的模块表，如果一个进程没有和 DLL 连接或没有通过 LoadLibrary 安装 DLL，甚至当另一个进程已经装入了时它还是不能在内存中看到 DLL 模块。这与 Win16 很不相同，在 Win16 里装入了的模块可以被所有任务看到，甚至这些任务和 DLL 无任何联系。虽然每个 Win32 进程有自己的模块表，对保密和耐用来说是个好主意，但它对节约共享代码和资源空间的愿望来说是不现实的。总之，如果你有 3 个 WINHELP 运行样品，WINHELP 码不应该装入三次，对吗?

KERNEL32 面临着严峻的选择，从应用前景看，每个进程应有自己的模块表。从 KERNEL32 的观点看，通过维持一个单一全局模块表来共享代码和数据是比较容易的(就象 Win16 做的一样)，无论何时，一个进程启动或一个新的 DLL 请求装入，KERNEL32 都能迅速的检查对应的单一全局表以确定 EXE 或 DLL 文件是否已被装入了。如果已装入了，那么 KERNEL32 能轻松地执行模块的参考计数。如果没有，KERNEL32 就需要在内存中设置一个新模块。

有两种数据结构提供 KERNEL32 用来在显示每个进程模块表的同时维持一个全局模块表的办法，一个是 IMTE(Internal Module Table Entry)，KERNEL32 码用来把模块表处理为全局表，另一个是 MODREF，KERNEL32 码用来使每个进程有自己的模块表。MODREF 将在稍后的“MODREF 结构”一节中讲到。

IMTE(Internal Module Table Entries [?])

正如图 3－1 所示，全局 KERNEL32 模块表只不过是一列指向 IMTE 的指针数组，在稍后要讲的伪码里，我将这列指针数组称为 pModuleTableArray。储存指

针数组的内存块是由 KERNEL32 堆分配的，该堆是一个规则的 HeapAlloc 型堆（就和我在第五章讲的一样），对应于向内存装载或卸载新模块，KERNEL32 通过 HeapReAlloc 函数扩大或缩小保存 pModuleTableArray 的内存块，当 KERNEL32 生成新的 IMTE 时，它在 pModuleTableArray 里寻找自由单元，如果找到，它就把指向 IMTE 的指针放置在里面。我们看 MODREF 时，数组元素的索引将变的很重要。pModuleTableArray 里的第一个数组元素（数组索引 0）和 KERNEL32. DLL 模块有关。

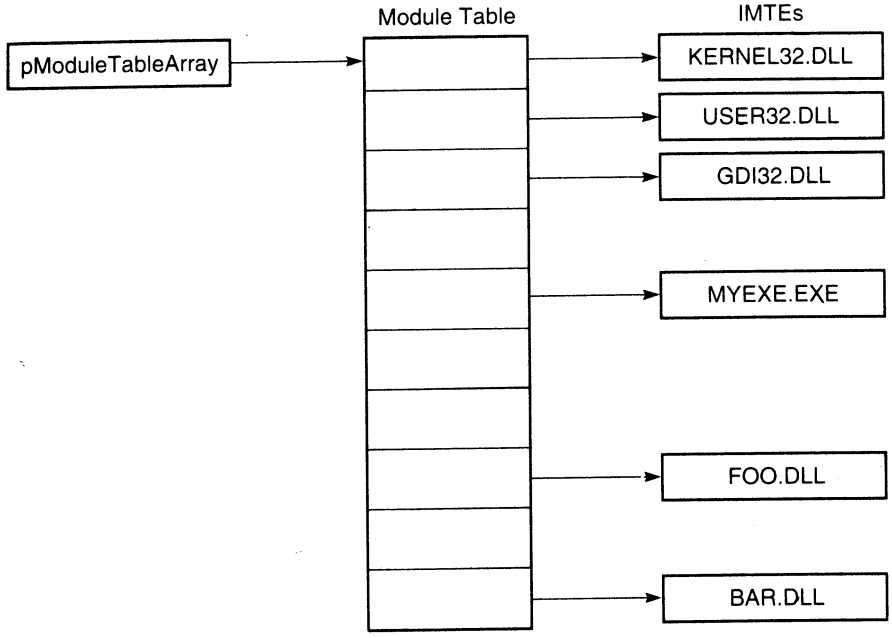


图 3-1　全局模块表是 IMTE 的指针数组

再强调一下，pModuleTableArray 里的每个非零元素代表着系统内一个已装载了的 EXE 或 DLL 文件，每个非零元素都是指向 IMTE（或 PIMTE，我在伪码中用到它）的指针。每个实模块数据库的格式都是文本化的（恰好是 IMAGE_NT_HEADERS 结构）然而 IMTE 的格式却不是（至少现在不是）。

IMTE 结构

Win32WLK 的 MODULE32. H 文件包含有一个 IMTE 结构的 C 型定义，每个 IMTE 有如下区段。

00h　　DWORD　　unl

此段保存了某种标记。

04h　　PIMAGE_NT_HEADERS　　pNTHdr

此指针指向内存中的一个 IMAGE_NT_HEADERS 结构。然而，它指向的结构只不过是显现在内存中模块的基地址上的 IMAGE_NT_HEADERS 结构的付本。此字段所指结构的内存是从 KERNEL32 堆分配来的，所以能被所有的进程“看到”，相比之下，安装在模块基址附近的初级 IMAGE_NT_HEADERS 可能低于 2GB，所以它只能访问安装了此模块的进程，通过付本能访问所有文本的 IMAGE_NT_HEADERS，KERNEL32 不用访问 ring 0 来打开内存文本就可以轻松的为任何装载的模块设置信息。

08h　　DWORD　　un2

这个 DWORD 的意思还不清楚，它通常被置为－1。

0Ch　　PSTR　　pszFileName

pszFileName 区含有指向源模块的 EXE 或 DLL 文件名的指针。例如，字符串 C:\WINDOWS\SYSTEM\KERNEL32. DLL 由 GetModuleFileName 函数反馈。GetModuleHandle 函数将该串作为参量与搜索的串进行比较，内存保存这个从 KERNEL32 堆中产生的文件名串。

10h　　PSTR　　pszModName

此 PSTR 指向一模块名串，在 Win32 里，模块名就是不包括路径信息的 EXE 或 DLL 文件名，例如，C:\WINDOWS\CALC. EXE 程序的模块名装入内存后就是 CALC. EXE，GetModuleHandle 函数也将此串与有关串进行比较，这个 pszModName PSTR 确确实实地指向了 pszFileName 字符串的内部，例如，前一个例子，它将指向 CALC. EXE。

14h　　WORD　　cbFileName

此 WORD 是自 0ch 区以后的 pszFileName 里的字符数量，在 GetModuleHandle 里被用来快速查看 pszFileName 串是否可以和输入串匹配。

16h　　WORD　　cbModName

此 WORD 是自偏移量 10ch 区后的 pszModName 里的字符数。也是被 GetModuleHandle 用来快速查看 pszModName 串是否和输入串匹配。

18h　　DWORD　　un3

此 DWORD 的意思我不清楚。

1ch　　DWORD　　cSections

此字段记录模块包含的区段(. text 和. idata 等等)数量，这个数值也可通过偏

移量 04h 区指向的 IMAG_NT_HEADERS 结构抽取。

20h　　DWORD　　un5

这个字段的意思我不清楚。它通常是 0，但在一个例子(COMCTL32. DLL)里，含有指向 KERNEL32 堆一个程序块的指针。

24　　DWORD　　baseAddress/ModuleHandle

baseAddress DWORD 含有装载了模块的位置的基址。在 Win32 里，模块的基址与它的 HMODULE 和 HINSTANCE 一样，所以这个字段可以理解为模块的 HMODULE 或 HINSTANCE。对 EXE 文件来说，基址总是 0x40000，对系统 DLL 文件来说，其用内存区的基址在 2GB 之上，第八章详细介绍了基址和如何从基址设置模块数据库。

28h　　WORD　　hModule16

此 WORD 含有一个线性地址指向 Win16NE 模块数据库的选择器(第七章介绍了 NE 模块数据库的格式)，在 Win32 应用程序中，NE 模块数据库含有从内存什么地方可找到 Win32 模块里资源的重要信息，这很有必要，因为资源操作码是在 Win16 KRNL386 和 USER. EXE 里，应该注意，hModule16 选择器不是通过 Win16 GlobalAlloc 函数分配的，所以此选择器不会像 Win16 全局内存句柄那样出现，因为这些和其它原因，Win16 TOOLHELP 不能看到对应每个 Win32 模块的 NE 模块。

2Ah　　WORD　　cUsage

这个字段含有模块的参考计数。例如，如果有 3 个拷贝在运行，那么 CALC. EXE 模块数据库将含有数值 3。

如果在 Win32 里有 GetModuleUsage 函数，肯定要报告这个字段的数值。不管怎样，下面这段话是 Win32 SDK 文件编制者不得不说的：

GetModuleUsage 函数已经过时了，它是被用来简化 16 位基于视窗应用程序入口的，每个基于 Win32 的应用程序都在自己的地址空间运行。

你相信谁？是文件编制者还是 KERNEL32 的实际运作。

2Ch　　DWORD　　un7

这个 DWORD 的意思还是不懂，然而，它明显含有一个指向 KERNEL32 堆块的有效指针。

30h　　PSTR　　pszFileName2

此 PSTR(包括以下 3 个字段)还是个秘密，它们看起来执行同样的功能，像偏移量 0ch 到 16h 做的那样。此字段(pszFileName2)指向不同的拷贝，这些拷贝是相互联系的 EXE 或 DLL 文件的完整路径的拷贝，pszFileName(偏移量 0ch)和 psz-

FileName2 所指向的串总是一样的。

34h　　WORD　　cbFileName2

这个字段含有 pszFileName2 所指串的长度。它应该总是和 cbFileName(偏移量 14h)的数值相同。

36h　　DWORD　　pszModName2

此字段指向 pszFileName2 串的一部分模块名(即基文件名)。此字段相当于 pszModName 字段(偏移量 10h)

3AH　　WORD　　cbModName2

这个字段含有 pszModName2 所指串的长度,它应该总是和 cbModName(偏移量 16h)的数值相等。

IMTE 实际上有两个指针分别指向模块的文件名和模块名,这很奇怪。我不敢肯定这么做的目的。还好,在模块方面有点好消息:在 Win16 里,EXE 和 DLL 的模块名在常驻名字表里位于第一条,并且被设置在连接程序的.DEF 文件内,在这儿也有问题,因为 Win16 装载器在决定一个模块是否已被装载时是把模块名等同于基文件名的。模块名和文件名不同的 EXE 或 DLL 文件能将 Win16 装载器弄糟并引起奇怪的问题,象模块名空间冲突,即两个或更多的毫无联系的 DLL 文件有同样的模块名。例如,如果你在不同的目录里有两个同名 DLL 文件,那么 Win16 装载器仅装载其中一个,装载另一个 DLL 的企图只会使装载器增加第一个模块的参考计数。实在糟透了!!! Win16 不能分辨不同目录里的同名 DLL,这会在可怜的意想不到的用户终端上引起莫名其妙的破坏。

幸运的是,在 Win32 里模块名的问题已不复存在了。传送给 GetModuleHandle 的 Win16 模块名是与 EXE 或 DLL 文件名相同的。因此,在 B 程序能从\BAZ 目录里装载它的 FOO.DLL 文件同时,A 程序也能从\BAR 目录里装载 FOO.DLL 文件。

当一个程序想同时用两个名字相同但内容不同的 DLL 时,一个 Microsoft 从没提起的情况出现了。例如,A 程序隐式连接着 FOO.DLL,并且装载器发现 FOO.DLL 在\BAR 目录里,后来此程序在 C:\BAZ\FOO.DLL 上作 LoadLibrary。是装载 C:\BAZ\FOO.DLL 呢,还是增加 C:\BAZ\FOO.DLL 的参考计数呢?Microsoft 的编辑者没有说。然而,我曾经和 Windows 95 的编码员讨论过,他宣称在内存里装载了两个截然不同的 FOO.DLL 拷贝。在浏览 SoftIce/W 的模块表时我亲眼看到了。

MODREF 结构

到现在为止你已经知道 KERNEL32 是怎样含有全局模块数组的了,我们可以将其余的问题归纳到一起。先前我讲了每一进程有自己的模块表并且不为其它

进程装载的模块所认同，将每个进程模块表与全局模块表连接起来的结构是MODREF 结构。每个进程模块表就是 MODREF 结构的连接表。为每个服务的MODREF 表含有为进程的 EXE 服务的 MODREF，就象为进程使用的 Win32 DLL 服务的 MODREF 一样，存储每个 MODREF 的内存都来源于 KERNEL32 堆，此堆位于 2GB 之上的共享区。因此，虽然 MODREF 强调模块表就是进程表，但 MODREF 表自己确实是全程路径性质的。Win32WLK 程序能浏览每个进程的模块表证明了这一点。

MODREF 表头被保留在进程数据库里(下面将讨论)每个 MODREF 结构都含有进入 pModuleTableArray 表的索引，图 3-2 描述了 MODREF 和 IMTE 的关系。

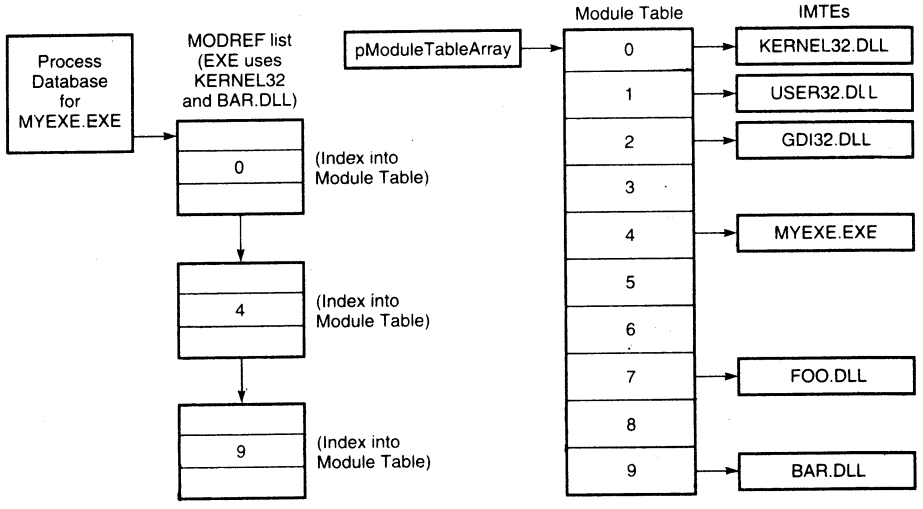


图 3-2　单一进程 MODREF 和全局 IMTE 表

来源于 WIN32WLK 的 MODULE32.H 文件包含一个为已知 MODREF 结构段服务的 C 结构定义。已知的字段如下：

00h　　PMODREF　　pNextModRef

此指针指向 MODREF 的当前进程表里的下一个 MODREF 结构。这个字段的 NULL 指针指明此表的末尾。列出一个进程所知模块的表就象从进程数据库里取得头 MODREF 节点然后在表里浏览一遍一样简单。此书附带磁盘内的 Win32WLK 程序给出了一个做此项工作的例子。

10h　　WORD　　mteIndex

此 WORD 是指向 IMTE 的全局指针数组(此指针数组在伪码一节中等同于 pModuleTableArray)的零基址索引。

18h　　PVOID　　ppdb

该指针为 PPROCESS_DATABASE(指向 PROCESS_DATABASE 结构),从 MODREF“向后连接”自身 MODREF 的进程,后面我们再去看 PROCESS_DATABASE。

因为 Windows 95 不得不让每个进程看起来都有各自的模块表,所以和 GetModuleHandle 一样,相关 API 没有立即着手全局模块表(pModuleTableArray)。相反,他们仅仅与那些参照入口进程 MODREF 表的全局模块表起作用。例如,GetProcAddress 函数只能查看当前进程的 MODREF 表的模块,甚至当一个模块已经被另一个进程装入后,GetProcAddress 也甭想把函数置入,除非该模块也在当前进程的 MODREF 表内。

Module-Related API 函数

现在你已经知道,在保留进程模块表的同时,KERNEL32 是怎样管理已装入模块的全局表的。

GetProcAddress 和 IGetProcAddress

GetProcAddress 在 Win32 程序里是个关键函数,因为通过它你可以随时取得装在均衡器上的 DLL 文件(相对于隐式连接的 DLL 文件)。给出模块标识符(HMODULE)和函数标识符(或者它的名字或输出次序),GetProcAddress 回送该函数的入口地址。为完成此项工作,GetProcAddress 必须首先在内存中设置规定的模块数据库,然后浏览输出操作表以找到地址。

实际的 GetProcAddress 码只不过是一个有效参数层,这证明 IpszProc 参数也是字符串或一个重要序号。通过查看 IpszProc 参数的高位 WORD 可识别两类不同的函数区分符,如果高位 WORD 是 0。那么低位 WORD 就是输出序数且不能执行进一步的操作。如果高位 WORD 不是 0,那么 IpszProc 就被作为 PSTR 并且扫描字符串以寻找 NULL 终止符。如果 PSTR 坏了,在扫描中就会出现异常,结构化异常处理器捕获此异常并将 0(失败)回送到调用程序(在后面的线程一节中我们将讨论结构化异常处理),如果执行通过了检查,那么控制器就跳到 IGetProcAddress 子程序,在那里驻留着 GetProcAddress 码的精髓。

GetProcAddress 的伪码

```
// Parameters:
//      HMODULE  hModule
//      LPCSTR  lpszProc

    Set up structured exception handling frame.
```

```
if ( lpszProc > 0x10000 )   // Values < 0x10000 contain ordinals in the
{                           // low WORD, so they aren't valid LPSTRs.

    AL = 0
    EDI = lpszProc          // Touch all the bytes in the lpszProc routine
    REPNE SCASB             // up to a NULL. If it faults, the exception
}                           // handler will catch it and return FALSE.

Remove structured exception handling frame.

goto IGetProcAddress
```

IGetProcAddress 负责在高层寻找引出函数操作的步骤，IGetProcAddress 先使线程同步以确保当前线程不会被突然中断。接着，调用 MRFromHLib 取回指向 MODREF 的指针。MRFromHLib 是 KERNEL32 内部的子程序，它通过扫描进程 MODREF 表来寻找带有 HMODULE 匹配的模块，此模块已被传送给 MRFromHLib。然后 IGetProcAddress 用 MODREF 结构里的模块目录索引来检查相关模块的 IMTE。

在 IGetProcAddress 的第二阶段，KERNEL32 检查有效操作地址。因为 IGetProcAddress 即可作为输出序数被传递也可作为一串指针被传递。它可决定那种形式并调用合适的低级子程序来检查操作，如果传送的是输出序数，那么 IGetProcAddress 调用 x_FindAddressFromExportOrdinal 子程序；如果传送的是一串指针，就调用 x_FindAddressFromExportName。另外，如果低级函数没能发现指定的操作，IGetProcAddress 就发出错误指示并回送 0。

在 Windows 95 的 β3 函数之前，IGetProcAddress 没有设置任何检查模块功能的特殊值。在 β3(也叫“Window Preview Program”释放)里，IGetProcAddress 获得了一小片实在令人讨厌的代码。

特别是只有在寻找 KERNEL32.DLL 函数时，IGetProcAddress 不允许通过此函数的引出序数来得到它的地址。Microsoft 为什么做这样一件可怕的事呢？在 KERNEL32.DLL 里有许多只能通过序数引出的非文本化的函数(可在附录 A 里看到一些它们的名字)，由于这些函数的名字不在 KERNEL32.DLL 里，所以他们不会在 KERNEL32 引入库里。因此，应用程序不能直接调用这些想来是 Microsoft 专门保留的函数。在《Unauthorized Windows 95》一书里，Schulman 写了一些调用非文本化的 KERNEL32 函数的程序——在 Windows 95 的后期设计中，这些程序取消了。是 Microsoft 故意的吗？你自己决定吧。自 β3 以后，调用非文本化 KERNEL32 函数的直接途径就不起作用了。然而，除此之外有很多不错的程序，你可以用 GetProcAddress 获得某个函数的地址并通过回送指针调用它。如果你知道非文本化函数的引出序数，你就可以设置，对吗？注意！IGetProcAddress 块里可怕的代码段企图通过非法 GetProcAddress 使用非文本化 KERNEL32 函数。因此，连 Schulman 也要试着用 GetProcAddress 来修理那破坏了的程序，事情变复杂了……

我个人认为这个 IGetProcAddress 很幼稚，任何 Windows 95 系统的编辑人员

都可写出他自己对 GetProcAddress 的看法，在第八章里给出关于 PE 模块的信息。我采用的一个可选择的途径是利用带有 Visual C++ LIB.EXE 的.DEF 文件建立一个带有非文本化函数的 KERNEL32 引入库。后面讲的 WIN32WLK 程序用了此引入库，附录 A 介绍了我的 Windows 95 非文本化 KERNEL32 功能引入库。

让我们回到关于 IGetProcAddress 内合法代码的讨论。在成功的找到指定函数的地址之后，你要考虑处理一下 IGetProcAddress。因为一些偶然因素使得 Windows 95 中一个进程被装入以进行调试时，会使系统 DLL 文件首先审查装载器建立在衬页上的特殊代码，这些代码的目的是防止应用调试程序进入 ring 3 系统 DLL 文件。为了隐式连接的功能，装载器暗地处理每件事。然而，调用 GetProcAddress 程序通常会避开这段代码。因此，GetProcAddress 检查程序是否在调试；如果 IGetProcAddress 要回送的地址超过 2GB，IGetProcAddress 会检查对应的地址并回送此地址。

IGetProcAddress 的最后一位是检查特定函数是否找到。如果没有，它会回送“ERROR_PROC_NOT_FOUND”错误信息。最后，IGetProcAddress 脱离临界段到函数的开始。

IGetProcAddress 伪码

```
// Parameters:
//      HMODULE  hModule
//      LPCSTR  lpszProc
// Locals:
//      PTHREAD_DATABASE    ptdb
//      FARPROC             pfnProc // Return value
//      PMODREF             pModRef
//      PIMTE               pimte

    pfnProc = 0;        // Initial return value

    // Synchronization stuff
    _EnterSysLevel( ppCurrentProcessId->crst );
    // Get a pointer to the MODREF that represents the module
    // specified by the hModule param. MRFromHLib() just scans
    // through the MODREF list, looking for a MODULE whose HMODULE
    // matches the HMODULE passed in.

    pModRef = MRFromHLib( hModule );

    if ( !pModRef ) // If the MODREF wasn't found, bail out.
    {
        InternalSetLastError( ERROR_INVALID_HANDLE );

        _DebugOut( SLE_MINORERROR, "GetProcAddress: %x not a Module handle",
                   hModule );

        if ( x_LoaderDiagnosticsLevel > 2 )
            dprintf("On ..\peldr.c Failure Path line %d\n", linenumber);
```

```
    goto done;
}

// Get a pointer to the IMTE for the specified module by looking
// it up in the pModuleTableArray.
pimte = pModuleTableArray[ pModRef->mteIndex ];

if ( lpszProc < 0x10000 )   // Looking for a specified export ordinal.
{
    if ( hModule == hModuleKERNEL32 )
    {
        InternalSetLastError( ERROR_NOT_SUPPORTED );
        _DebugOut( "GetProcAddress: kernel32 by id not supported",
                   SLE_MINORERROR );

        if ( x_LoaderDiagnosticsLevel > 2 )
            dprintf( "On ..\peldr.c Failure Path line %d\n", line num );

        goto done;
    }

    // Scan through the module database, looking for the function
    // with the specified export ordinal.
    pfnProc = x_FindAddressFromExportOrdinal( pimte->pNTHdr, lpszProc );

    if ( !pfnProc ) // Function not found? Spit out an error message.
    {
        pModRef = MRFromHLib( hModule, lpszProc )
        _DebugOut(  SLE_MINORERROR,
                    "GetProcAddress(%s, %d) not found"
                    pModuleTableArray[pModRef->mteIndex]->pszModName,
                    lpszProc );
    }
}
else        // Looking for a specified function name.
{
    // Scan through the module database, looking for the function
    // with the specified name.
    pfnProc = x_FindAddressFromExportName( pimte->pNTHdr, 0, lpszProc );

    if ( !pfnProc ) // Function not found? Spit out an error message.
    {
        pModRef = MRFromHLib( hModule, lpszProc )

        _DebugOut(  SLE_MINORERROR,
                    "GetProcAddress(%s, %s) not found"
                    pModuleTableArray[pModRef->mteIndex]->pszModName,
                    lpszProc );
    }
}

// If the function is in a shared, system DLL (i.e., it's above 2GB),
// *AND* if the process is being debugged, change the returned
// function address to point to the bizarre pre-API stubs that
```

```
    // KERNEL32 sets up. These stubs sit between the call to the
    // API and the actual API code.

    if ( (pfnProc >= 0x80000000) && (pfnProc != &DebugBreak) )
    {
        if ( ptdb->pProcess2->WaitEventList
             && !ppCurrentTDBX->MustCompleteCount )
        {
            pfnProc = DEBCreateDIT( ppCurrentTDBX->TopOfStack, pfnProc )
        }
    }

    // If the function is going to return a failure, set the GetLastError code.
    if ( pfnProc == 0 )
        InternalSetLastError( ERROR_PROC_NOT FOUND );

done:
    // Undo the synchronization stuff.
    LeaveSysLevel( ppCurrentProcessId->crst );

    return ESI;
```

x_FindAddressFromExportOrdinal

x_FindAddressFromExportOrdinal（我定义的函数，不是 Microsoft 的）是 KERNEL32 的核心程序。不仅仅因为它来自 GetProcAddress 还因为它是在安装隐式装载的 DLL 的功能时，被 PE 装载器调用。一般说来，这个程序是一个检查 KERNEL32.DLL 里输出函数地址的过渡程序。

x_FindAddressFromExportOrdinal 函数主要依赖于 IMAGE_NT_HEADER 内的信息和 PE 文件的.edata 段。此 PE 文件是装入内存以建立模块用的。（我再一次强调这就是为什么有关 PE 文件的第八章如此重要，甚至在你不用 PE 文件工作的情况下也一样）。

虽然在 x_FindAddressFromExportOrdinal 内有相当规模的代码，但此函数相当简单。在模块的引出表（.edata 段）里，你会发现一个 RVA（相关有效地址）数组，它为模块的输出功能服务。此数组被称为“引出地址表”。数组的第一部分含有引出序数 1 的 RVA，第二部分含有输出序数 2 的地址，等等。x_FindAddressFromExportOrdinal 不得不做的唯一一件事是索引数组以获得 RVA，然后加上模块的装载地址将 RVA 变成可用线性地址。以上情况有两点需要说明：

第一点（不明显）是 x_FindAddressFromExportOrdinal 需要解释序数基值。在 PE 文件里，最短位数的引出序数被做为基值，这可以使引出地址表比通常小。例如，我们从序数 100 到 109 看一个 DLL 引出函数，一般情况下，在引出地址表内应有 110 项，但只用了最后 10 项，为了节约空间，连接程序将序数基值设为 100，即设置了一个只有 10 项的引出地址表。检查一个引出函数时，x_FindAddressFromExportOrdinal 不得不记着序数基值以得到真正数组索引。

另一点是 x_FindAddressFromExportOrdinal 不得不处理传送函数，第八章详细介绍了传送函数。现在，知道传送函数是一种 DLL 内引出函数的别名就足够了，例如，在 Windows NT 内，KERNEL32. DLL 内的 HeapAlloc 函数被传送给 NTDLL. DLL 内的 RtlAllocateHeap，引出地址表为传送函数保留的地址总是在. edata 段内。此地址不是引出函数的地址。相反，此地址指向象 NTDLL. RTLAllocateHeap 这样的字符串。如果 x_FindAddressFromExportOrdinal 看到这样的事情发生，它就将此字符串分为模块名和函数名，并且调用有这些数值的 GetProcAddress。会造成 GetProcAddress 的递归。

x_FindAddressFromExportOrdinal 伪码

```
// Parameters:
//      PIMAGE_NT_HEADERS pNTHdr
//      DWORD             ordinal
// Locals:
//      char    szForwardedModule[ MAX_PATH ]   // 0x260
//      PIMAGE_EXPORT_DIRECTORY pExpDir;
//      PDWORD                  pFunctionArray;
//      DWORD                   imageBase;
//      DWORD                   retAddr;
//      DWORD                   exportDirSize

    // Get the size of the export table out of the NT header.
    exportDirSize =
        pNTHdr->OptionalHeader.
                DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size

    // If no functions are exported, bail out immediately.
    if ( exportDirSize == 0 )
    {
        InternalSetLastError( ERROR_MOD_NOT_FOUND );
        if ( x_LoaderDiagnosticsLevel > 2 )
        {
            dprintf("On ..\peldr.c Failure Path line %d\n", line number );

        }

        return 0;
    }

    // Get the address where the module is loaded in memory.
    imageBase = pNTHdr->OptionalHeader.ImageBase;

    // Get a pointer to the export table.
    pExpDir = pNTHdr->OptionalHeader.
                DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress
              + imageBase;
    // Get a pointer to the array of exported function addresses.
    pFunctionArray = imageBase + pExpDir->AddressOfFunctions
```

```
    // If the ordinal requested is greater than the number of exported
    // functions, bail out. Make sure to take the ordinal base into account.
    if ( pExpDir->NumberOfFunctions <= (ordinal - pExpDir->Base) )
        return 0;

    // Read RVA of the exported entry out of the array (again, taking
    // the ordinal base into account).
    retAddr = pFunctionArray[ ordinal - pExpDir->base ];

    // Bias the RVA extracted from the table by the image base to convert the
    // RVA into a usable linear address.
    if ( retAddr )
        retAddr += imageBase;

    // See if the found address is within the export directory. If so,
    // it's a forwarded DLL, and the address is a pointer to the name
    // of the function that it's forwarded to.
    //
    // If the address isn't within the export directory, we're done.  Return
    // the found address to the caller.
    if ( (retAddr < pExpDir) || (retAddr >= (pExpDir + exportDirSize) )
    {
        PSTR pszForwardedFunctionName
        HMODULE hForwardedMod;

        Copy the DLL name pointed at by retAddr into the szForwardedModule
        local variable, stopping when a '.' is reached. Point
        pszForwardedFunctionName at the character after the '.'

        hForwardedMod = IGetModuleHandleA( szForwardedModule )
        if ( !hForwardedMod )
        {
            _DebugOut( SLE_MINORERROR, "Unable to find forwarded DLL %s",
                       szForwardedModule );
            retAddr = 0;
            goto done;
        }

        // Call GetProcAddress to get the real address of the forwarded
        // function in the DLL that contains it. Yes, this does make
        // GetProcAddress recursive if it's a forwarded function.
        retAddr = IGetProcAddress( hForwardedMod, pszForwardedFunctionName );
        if ( !retAddr ) // Oops! Didn't find the forwarded function.
        {
            _DebugOut( SLE_MINORERROR, "Unable to find forwarded export %s.%s",
                       szForwardedModule, pszForwardedFunctionName);
        }
    }

done:
    return retAddr;
```

x_FindAddressFromExportName

此函数是 X_FindAddressFromExportOrdinal 的姐妹函数。两者之间的主要区别是 x_FindAddressFromExportName 以一个函数名开始，而不是以它的引入序数开始。两者程序的第一部分是一样的，因为二者都需要设置指向内存中不同位置的同样的指针。

x_FindAddressFromExportName 实质是通过引出名数组寻找带有 lpszProc 参数的匹配。如果此函数找到一匹配串，就用 AddressOfNameOrdinal 数组将此串数组索引转化为引出地址表目录。在这一点上，x_FindAddressFromExportName 能轻易的检查一遍引入函数的 RVA 并将它回送给调用程序，然而这会使它略去 X_FindAddressFromExportOrdinal 函数的特殊情况码(即处理序数基值和调试程序码)。因此，程序将它找到的引出序数传送给 x_FindAddressFromExportOrdinal 函数使之工作。x_ FindAddressFromExportOrdinal 回送的内容和 x_ FindAddressFromExportName 回送的一样。

为使所有这些简单明了，函数的地址即可通过名字检索也可由序数值来检索。实际上，地址最终总是用引出序数定位的。当你传送一名字给 GetProcAddress 或输出一个函数名字时，KERNEL32 只插入一特殊步骤将字符串名转化为它的引出序数。

x_FindAddressFromExportName 的伪码

```
// Parameters:
//      PIMAGE_NT_HEADERS   pNTHdr
//      DWORD               hintNameOrdinal
//      PSTR                lpszProc
// Locals:
//      PIMAGE_EXPORT_DIRECTORY pExpDir;
//      DWORD                   imageBase;
//      PDWORD                  pNamesArray;
//      PWORD                   pNameOrdinalsArray;
//      DWORD                   cbProcName
//      DWORD                   numNamesMinus1
//      DWORD                   nameOrdinal
//      DWORD                   curTestingNameOrdinal

    if ( hintNameOrdinal != some number )   // ???
    {
        CheckDll();
    }

    // If no functions are exported, bail out immediately
    if ( 0 == pNTHdr->OptionalHeader.
            DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size )
    {
        if ( x_LoaderDiagnosticsLevel > 2 )
        {
```

```
            dprintf("On ..\peldr.c Failure Path line %d\n", line number);
        }

error_return:
        InternalSetLastError( ERROR_MOD_NOT_FOUND );
        if ( x_LoaderDiagnosticsLevel > 2 )
        {
            dprintf("On ..\peldr.c Failure Path line %d\n", line number);
        }

        return 0;
    }

    // Get the address where the module is loaded in memory.
    imageBase = pNTHdr->OptionalHeader.ImageBase;

    // Get a pointer to the export table.
    pExpDir = pNTHdr->OptionalHeader.
                DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress
              + imageBase;

    // Get a pointer to the array of PSTRs for the exported function names.
    pNamesArray = imageBase + pExpDir->AddressOfNames;

    // Get a pointer to the array that correlates names array indices
    // to indices in the export address table.
    pNameOrdinals = imageBase + pExpDir->AddressOfNameOrdinals;

    // If no names were exported, bail out.
    if ( pExpDir->NumberOfNames == 0 )
    {
        if ( x_LoaderDiagnosticsLevel > 2 )
        {
            dprintf("On ..\peldr.c Failure Path line %d\n", line number);
        }

        return 0;
    }

    // Calculate how many names are exported.
    numNamesMinus1 = pExpDir->NumberOfNames - 1;

    curTestingNameOrdinal = 0;

    cbProcname = strlen( lpszProc )

    // It appears that the function can be passed a "hint" ordinal
    // that may or may not be the ordinal of the actual function
    // we're looking for. Check to see if the name of the function that
    // corresponds to the hint ordinal is the same string as was passed
    // in the lpszProc parameter. If so, we know the ordinal, and we
    // can skip the linear search through all the function names that comes
    // later.
    if ( numNamesMinus1 >= hintNameOrdinal )
```

```
    {
        // Uses CompareStringA() with SystemDefaultLangID as the LCID to
        // see if the strings match.
        if (!GlorifiedStringCompare(imageBase + pNamesArray[hintNameOrdinal]))
        {
            ordinal = hintNameOrdinal;
            goto FoundOrdinal
        }
    }

    if ( numNamesMinus1 < 0 )
        goto error_return;

    // Scan through the array of function names PSTRs, looking for a
    // string that matches the passed-in lpszProc parameter.

    A nasty little piece of code iterates through the entries in the
    "AddressOfNames" array. Each entry is compared (REP CMPSB) with the
    lpszProc string.

    if a match is found
    {
        set nameOrdinal to the index of the matching string in the
        AddressOfNames array

        goto to FoundOrdinal
    }

    if a match isn't found
        goto error_return:

FoundOrdinal:

    return x_FindAddressFromExportOrdinal(
                pNTHdr, pNameOrdinalsArray[nameOrdinal] + pExpDir->Base );
```

GetModuleFileName 和 IGetModuleFileName

GetModuleFileName 函数将一个 HMODULE 做为输入，并将完整的路径回送给用来建立模块的 EXE 或 DLL。GetModuleFileNameA(译者注：原文可能有误，末尾的 A 不应该有)码本身非常小，只是一参数变量存根。在验明 lpszPath 参数合法以后，GetModuleFileName 就跳到 IGetModuleFileName。

如果 IGetModuleFileName 不必将自己与 ANSI 和 OEM 文件名联系起来，它会更加简单，KERNEL32 里的 SetFileApisToANSI 和 SetFileApisToOEM 函数，使调用程序明确文件名应该用 ANSI 字符还是用 OEM 字符。在内部，Windows 95 将所有文件名以 ANSI 形式储存，在需要时将它们转化为 OEM 字符或将 OEM 字符转化回来。

IGetModuleFileName 函数的内容被做此转化的代码掩盖着。除了文件名问题之外，IGetModuleFileName 的核心相当简单，所有要做的就是将完整的文件从

IMTE 里拷贝到输出缓冲区。然而，由于每个进程都有自己的模块表，IGetModuleFileName 不能简单地搜索 pModuleTableArray 寻找有关模块。IGetModuleFileName 用 MRFromHLib 函数代替以寻找模块的 MODREF（我早在 GetProcAddress 的讨论中就清楚的讲述了 MRFromHLib 函数）有了所决定模块的 MODREF，IGetModuleFileName 就用此 MODREF 的 mteIndex 字段来索引 pPModuleTableArray 并取得 IMTE 指针。一但有了 IMTE 指针，剩下的事情只是将 IMTE 的 pszFileName 字段所指的字符串拷贝进来传送给 GetModuleFileName 的缓冲区内。

GetModuleFileNameA 伪码

```
// Parameters:
//      HMODULE  hinstModule
//      LPTSTR  lpszPath
//      DWORD  cchPath

    Set up structured exception handling frame

    *lpszPath += 0;      // Harmlessly write to lpszPath. If a fault occurs,
                         // the exception handler will catch us and return
                         // failure.

    Remove structured exception handling frame

    goto IGetModuleFileNameA
```

IGetModuleFileNameA 伪码

```
// Parameters:
//      HMODULE  hinstModule
//      LPTSTR  lpszPath
//      DWORD  cchPath
// Locals:
//      DWORD   fOem
//      DWORD   retValue
//      PMODREF pModRef

    retValue = 0;

    EnterSysLevel( ppCurrentProcessId->crst );

    // Deal with OEM stuff (if SetFileApisToOEM is somehow involved).
    fOem = x_AreFileApisOEM();

    if ( fOem )
    {
        // Calls k32CharToOemA and some other things.
        SomeFunction( lpszPath, 1 );
    }
```

```
if ( cchPath )      // Null out the return path string.
    *lpszPath = 0;

if ( hInstModule == 0 )     // The HMODULE was 0. We want the EXE's name.
{
    pModRef = ppCurrentProcessId->pExeMODREF
}
else                // We were passed a specific HMODULE to look for.
{
    // Scan through the process's MODREF list, looking for a module
    // with an HMODULE that matches the hInstModule parameter.
    pModRef = MRFromHLib( hInstModule );
}

if ( pModRef == 0 ) // Oops! Didn't find the module.
{
    InternalSetLastError( ERROR_INVALID_PARAMETER );

    if ( x_LoaderDiagnosticsLevel > 2 )
        dprintf("On ..\peldr.c Failure Path line %d\n", line number);
}
else                // We found the module.
{
    PIMTE pimte;

    // Get a pointer to the IMTE by looking it up in the global module
    // table array.
    pimte = pModuleTableArray[ pModRef->mteIndex ];

    if ( cchPath )  // Are we supposed to write anything out?
    {
        retValue = pimte->cbFileName;
        if ( retValue >= cchPath )
            retValue = cchPath - 1;

        // Copy the path name to the output buffer.
        memmove( lpszPath, pimte->pszFileName, retValue )
        lpszPath[ retValue ] = 0;       // Null terminate it.
    }
}

if ( fOem ) // If fOEM'ing, convert the output buffer to OEM.
{
    ppCurrentProcessId->flags &= ~fOKToSetThreadOem;   // Turn off flag.

    if ( cchPath )
    {
        // Also calls k32CharToOemA and some other things.
        SomeOtherFunction( lpszPath, 1 );
    }
}

LeaveSysLevel( ppCurrentProcessId->crst );

return retValue;
```

GetModuleHandle 和 IGetModuleHandle

GetModuleHandle 函数与 GetModuleFileName 函数的操作相反。给定了模块名，此函数回送模块的 HMODULE(或基地址，如果你乐意)。不幸的是，Microsoft 说明书中关于模块名由什么组成的说法有些含糊。然而，下面的伪码会将此问题完全弄清楚，实际上模块名即可以是一个基文件名也可以是到 EXE 或 DLL 文件的完整的路径名。另外，如果文件的扩展名是 DLL，模块名可以忽略它。因此，下面是所有 C:\WINDOWS\SYSTEM\USER32.DLL 的有效模块名。

USER32
USER32.DLL
C:\WINDOWS\SYSTEM\USER32
C:\WINDOWS\SYSTEM\USER32.DLL

真正的 GetModuleHandle 码非常短；它只是使 lpszModule 参数合法化以肯定它是有效串指针。如果它是，GetModuleHandle 就跳到 IGetModuleHandle，和 IGetModuleFileName 一样，IGetModuleHandle 的核心被执行 ANSI 到 OEM 转换的代码掩盖着。代码的核心部分首先用大写字母拼写模块名以便代码能在以后做更快更灵敏的对比。接着，IGetModuleHandle 检查文件名是否有扩展名(例如 EXE 或 DLL)如果没有，代码就加上.DLL 扩展名。

剩下的核心码包括两个帮助函数的调用：即 x_GetMODREFFromFileName 和 x_ GetHMODULEFromMODREF。首先，x_ GetMODREFFromFileName 扫描 MODREF 表找到一个与其匹配的文件名，并回送指针到 MODREF。接着，x_GetHMODULEFFromMDREF 接收 PMODREF 并回送它的相关 HMODULE。这些帮助函数在以下两段里做了描述。

GetModuleHandleA 伪码

```
// Parameters:
//      LPCTSTR  lpszModule;

    Set up structured exception handling frame

    if ( lpszModule )                       // Read each byte of the name to
        REPNE SCASB till a zero is found    // make sure it's valid. The
                                            // exception handler will catch
                                            // us if something's wrong.

    Remove structured exception handling frame

    goto IGetModuleHandleA
```

IGetModuleHandleA 伪码

```
// Parameters:
//      LPCTSTR  lpszModule;
// Locals:
//      DWORD    myLocal
//      BOOL     fOem
//      DWORD    retValue
//      char     szBuffer[260]
//      PMODREF pModRef

    pszFileExtension = 0;

    fOem = x_AreFileApisOEM();

    if ( fOem )
     {
         // Calls k32OemToCharA and some other things.
         lpszProc = SomeFunction( lpszModule, 0 );
     }

     if ( lpszModule == 0 )    // Asking for the EXE.
     {
         retValue = x_GetHModuleFromMODREF( ppCurrentProcessId->pExeMODREF );
     }
     else    // Caller specified a module name.
     {
         strcpy( szBuffer, lpszModule );

         x_UppercasePathName( szBuffer, &pszFileExtension );

         if ( pszFileExtension == 0 )    // If no extension found, tack
         {                               // on ".DLL".
             strcat( szBuffer, ".DLL" )
         }
         else
         {
             if ( *pszFileExtension == 0 )   // Strip off a trailing '.' if
                 *(pszFileExtension-1) = 0;  // present.
         }

         pModRef = x_GetMODREFFromFilename( szBuffer );

         retValue = x_GetHMODULEFromMODREF( pModRef );

         if ( retValue == 0 )
             InternalSetLastError( ERROR_MOD_NOT_FOUND );
     }

     if ( fOem )
     {
         ppCurrentProcessId->flags &= ~fOKToSetThreadOem;   // Turn off flag.

         // Also calls k32CharToOemA and some other things.
         SomeOtherFunction( lpszPath, 0 );
     }

     return retValue
```

x_GetMODREFFromFilename

x_GetMODREFFromFilename 函数(我自己起的名字)扫描进程相连接的 MDREF 表,将每个模块的文件名与传送的 pszModName 参数进行比较。如果匹配,x_GetMODREFFromFilename 就回送一个 PMODREF。否则,回送 NULL。

我发现 x_GetMODREFFromFilename 不是对一两对,而是 4 对字符串的比较,每对字符串是输入的字符串加 MODREF 的文件名。

首先,x_GetMODREFFromFilename 将输入的字符串与 MODREF 的基文件名比较(例如,与 KERNEL32.DLL)。如果失败,x_GetMODREFFromFilename 就将输入的字符串与 MODREF 指向的完整路径比较。如果又失败,此函数将再追加两次比较:第三次与基文件名的次拷贝比较,第四次与储存在 MODREF 内的完整路径名的次拷贝比较。如果这些比较中有成功的,此函数将回送一个指针到匹配的 MODREF。

为加快比较速度,x_GetMODREFFromFileName 首先计算输入字符串的长度。由于 MODREF 结构指向的字符串的长度也储存在 MODREF 内,所以 x_GetMODREFFromFileName 首先将输入的字符串长度与 MODREF 字符串长度进行比较。如果它们不匹配,此函数就没必要再为那特殊的 MODREF 字符串进行字符串比较了。

x_GetMODREFFromFileName 伪码

```
// Parameters:
//      PSTR    lpszModName
//      PMODREF pModRef;
//      PIMTE   pimte;
//      DWORD   nameLen;

    nameLen = strlen( lpszModName );

    pModRef = ppCurrentProcessId->MODREFlist;

    if ( !pModRef )
        return 0;

    while ( pModRef )
    {
        pimte = pModuleTableArray[ pModRef->mteIndex ]

    if ( nameLen == pimte->cbModName )
    {
        if ( 0 == strcmp(lpszModName, pimte->pszModName) )
            break;      // Found it!!!
    }

    if ( nameLen == pimte->cbFileName )
    {
```

```
            if ( 0 == strcmp(lpszModName, pimte->pszFileName) )
                break;      // Found it!!!
        }

        if ( nameLen == pimte->cbModName2 )
        {
            if ( 0 == strcmp(lpszModName, pimte->pszModName2) )
                break;      // Found it!!!
        }

        if ( nameLen == pimte->cbFileName2 )
        {
            if ( 0 == strcmp(lpszModName, pimte->pszFileName2) )
                break;      // Found it!!!
        }

        // We didn't find it in any of the above comparisons. Try
        // the next module in the list.
        pModRef = pModRef->pNextModRef;
    }

    // When we get here, we've either found a PMODREF with the right name,
    // or pModRef == 0;

    return pModRef;
```

x_GetHModuleFromMODREF

此函数将 PMODREF 作为输入参数，并回送相应模块的 HMODULE(或基地址)，做此事的工作量很小，此函数从传送给它的 MODREF 结构内得到一个指向模块数据库(一个 IMAGE_NT_HEADERS 结构)的指针。有一个 IMAGE_NT_HEADERS 内部的字段是模块的基装载地址，就我们目前所知，此模块与 HMODULE 一样。

x_GetHMODULEFFromMODREF 伪码

```
// Parameters:
//      PMODREF     pModRef
// Locals:
//      PIMAGE_NT_HEADERS pNTHdr
//      PIMTE       pimte;

    if ( pModRef == 0 )
        return 0;

    pimte = pModuleTableArray[ pModRef->mteIndex ];
```

```
pNTHdrs = pimte->pNTHdr

return pNTHdr->ImageBase;       // The load address (image base) is
                                // the same as the HMODULE.
```

KERNEL32(简称 K32)对象

我喜欢直接讨论有关进程和线程的问题,但在这里我得首先将对象的概念解释清楚,尽管 KERNEL32 从其堆中分配给内存的任何事情都可称为"一个对象",但在这儿我有特殊定义。

K32 对象来自 KERNEL32 堆的关键系统数据结构,K32 对象的类型颇多,并且首部相同,决定某些事情是否 K32 对象的一种办法是问一下"应用程序对这个对象有句柄吗?"例如应用程序可以有文件柄或事件柄,那么文件或事件就是 K32 对象。另外,我从没见过应用代码对 MODREF 或 IMTE 有句柄,因此 MODREF 和 IMTE 不是 K32 对象。

每个 K32 对象都有如下格式的头标:

00h　　DWORD

对象的类型,该值决定如何解释结构的某些部分。

04h　　DWORD

对象的参考值,该值决定其它代码可访问对象多少次。例如,当你调用 GetFileInformationByHandle()时,正在访问的文件对象的参考值上升一位,在函数返回之前,则减少文件对象的参考值。

至此,你大概急于想知道 K32 对象有些什么类型。下面就是

K32OBJ_SEMAPHORE(0x1)
K32OBJ_EVENT(0x2)
K32OBJ_MUTEX(0x3)
K32OBJ_CRITICAL_SECTION(0x4)
K32OBJ_PROCESS(0x5)
K32OBJ_THREAD(0x6)
K32OBJ_FILE(0x7)
K32OBJ_CHANGE(0x8;参看 FindFirstChangeNotification)
K32OBJ_CONSOLE(0x9)
K32OBJ_SCREEN_BUFFER(0xA)
K32OBJ_MEM_MAPPED_FILE(0xB;参看 CreateFileMapping)

K32OBJ_SERIAL(0xC)
K32OBJ_DEVICE_IOCTL(0xD;参看 DeviceIoControl)
K32OBJ_PIPE(0xE)
K32OBJ_MAILSLOT(0xF)
K32OBJ_TOOLHELP_SNAPSHOT(0x10;参看 CreateToolhelp32Snapshot)
K32OBJ_SOCKET(0x11)

在这章的剩余部分,我们主要集中在进程和线程对象(ID 5 和 6)一个进程数据库就是一个 K32_PROCESS 对象,一个线程数据库也是一个 K32_THREAD 对象。和你将在"什么是进程柄? 什么是进程 ID?"一节看到的一样,一个进程柄表只不过是一个指向上面所示的不同类型 K32 对象的指针数组。通过 KERNEL32 和 VWIN32.VXD,代码检查假设对象的第一个 DWORD 以确定真的在处理它所认为类型的对象。

如果你熟悉 Win16 的内核,你可能注意到与 Win16 的任务和模块不一样,8 位 Win32 对象没有任何储存链接表指针的字段。在 Win16 内,一但你在表内发现第一个任务或模块,你就可以轻松的走完全表。在 Windows 95 内,KERNEL32 有它自己的保存 K32 对象表的代码段(LSTMGR.C)。

Windows 95 进程

现在,是引出进程的一般定义的时候了,进程是一个所有制单元,进程拥有内存(实际上拥有内存文本)。进程拥有应用程序用来读写文件的文件柄。进程拥有线程(我将在本章后面给出线程的完整定义)。进程拥有已装入进程内存文本中的一个 DLL 模块表。我可以继续讲,但我认为你已掌握大意了。

注意进程不代表执行(线程代表码的执行),且进程不是 EXE 文件,在被装入之前,磁盘上的 EXE 文件只是一个程序。就在它被装入内存的同时,Windows 95 建立一个进程。另外,每个进程与一个磁盘文件有关(KERNEL32.DLL 进程例外,WIN32WLK 一节介绍)。

当 Windows 95 建立一个新的进程时,同时也建立一个新的内存文本以便执行进程的线程。另外,Windows 95 还为进程建立一个初始执行线程。如果需要,进程能建立附加线程,系统还建立一个文件柄表,进程可在文件柄表内保存一个开式柄表,最后,并且对下面几段的讨论很重要,Windows 95 建立一个进程数据库描述相应的进程。

进程数据库是一个含有大量有关进程信息的 K32 对象(我们将在"Windows 95 进程数据库(PDB)"一节看到)。进程数据库内存在 KERNEL32 堆中进行分配,所以所有进程数据库对所有任务都是可视的(假设它们知道到哪里去看;那是一个棘手问题,我将在 WIN32WLK 资源内给出怎样去做)。

进程数据库主要包括一个线程表、一个装载模块表、隐式进程堆的堆柄,一个指向进程柄表的指针和一个指向用来运行进程的内存文本的指针(第五章)。但这

些只是一部分，还有很多很多。实际上，还有一个内存映象文件表、一个指向当前进程的指针、一个有效线程局域存储器表和一个指向设备块的指针，如果你现在就要，只需寄 $49.95 给 KERNEL32.DLL 以 1 Microsoft Way……

什么是进程柄？
什么是进程 ID？

在继续进行之前，我想弄清楚进程柄和进程 ID 的关系。两个同样夸大的 Win32 程序—— GetCurrentProcess 和 GetCurrentProcessId——把相当数量的程序员搞糊涂了。两个函数之间的区别实际上很明显。

进程柄实际上和文件柄一样。它是一个不带有任何有效指针值的“不透明”数值，事实上，系统把 K32 对象柄（例如进程或文件柄）做为进程柄表的索引。通过索引进程柄表数组回送的数值事实上是指向 K32 对象的指针。然而，如果没有给应用程序直接访问柄表的权力，进程柄是没有用的。

记住由于每个应用程序有它自己的柄表，很有可能不同的进程在它们特有的进程文本内有相同的进程表，例如，一般每个进程自己有一个进程柄，且此柄的数值是 1。要讲明的一点是，进程柄不能区分不同的进程。又例如，如果一个应用程序为它自己的进程又打开一个进程柄，那么就有两个不同的柄数值，它们都标记着同一个进程。

进程柄不适合用来鉴别哪一个进程正在使用。它的进一步证明将在 GetCurrentProcess 码中描述。

GetCurrentProcess 伪码

```
// Normally this function does nothing. It appears to be there
// for the benefit of the KERNEL32 developers.
x_LogSomeKernelFunction( function number for GetCurrentProcess );

return 0x7FFFFFFF;
```

对了！忽略了对记录函数的调用，GetCurrentProcess 只是执行一下回送一个固定数值（0x7FFFFFFF）的函数。不管什么进程调用 GetCurrentProcess，它总是反馈 0x7FFFFFFF 值。此数值是一个“万能”值，KERNEL32 将它译为“使用当前进程”，在这个过程中，KERNEL32 期望有个进程柄，查寻数值 Ox7FFFFFFF 并且取代当前进程。还需要更多的证据证明进程柄除非在它们自己的文本中使用才有效吗？我想不必了。

现在让我们看看进程 ID，与《Unauthorized Windows 95》中讲的一样，Windows 95 的早期版本通过 β1 将进程数据库的地址做为进程 ID，由于进程数据库储存在所有进程都可访问的共用内存区，所以进程数据库的地址在整个系统内保证是单一的。《Unauthorized Windows 95》使用扩展的 GetCurrentProcessId 函数来获

得一个指向当前进程数据库的指针，然后从此数据库抽取关键字段。不幸的是，Microsoft KERNEL32 的编码员破坏了此特殊部分，我们可以在很多目前的 GetCurrentProcessId 版本中看到它们。

GetCurrentProcessId 伪码

```
x_LogSomeKernelFunction( function number for GetCurrentProcessId );

return PDBToPid( ppCurrentProcess );
```

同样，记录函数 GetCurrentProcessId 只是传送一个全局变量(ppCurrent-Process)给 PDBToPID 函数，让我们仔细看看，因为这对理解本章的剩余部分极其重要，ppCurrentProcess 全局变量是一个指向当前进程数据库的指针的指针。引入符号 C，这意味着 * * ppCurrentProcess 指向当前进程数据库。

你不得不曲折的通过此指针两次的原因是第六章里讲的一件令人着迷的事。目前，只需记住 ppCurrentProcess 指针是一个 KERNEL32. DLL 内的全局变量，它可以让 KERNEL32 发现当前进程的进程数据库(为简单起见，我在伪码中用到 ppCurrentProcess 变量时，假设它就是指向进程数据库的指针，而不是指向指针的指针)。

因此，如果 KERNEL32 有一个指向当前进程数据库的指针，且此数据库使用很方便，GetCurrentProcessId 为什么不回送该指针呢？为了寻找答案，让我们看看 PidToPDB(译者注：与给出的伪码名不一致，原文是这样)函数：

PDBToPid 伪码

```
// Parameters:
//      PROCESS_DATABASE * ppdb

    if ( ObsfucatorDWORD == FALSE )
    {
        _DebugOut( "PDBToPid() Called too early! Obsfucator not yet"
                   " initialized!" );
        return 0;
    }

    if ( ppdb & 1 )
    {
        _DebugOut( "PDBToPid: This PDB looks like a PID (0%lxh) Do a"
                   " stack trace BEFORE reporting as bug." );
    }

    // Here's the key! XOR the obsfucator DWORD with the process database
    // pointer to make the PID value.

    return ppdb ^ ObsfucatorDWORD;
```

那是真的吗？是的，“Obsfucator”一词直接来自 Microsoft 的二进制（是的，“Obsfucator”拼写错了；应该是“Obfuscator”），除了通过检查来确认有效进程数据库指针已被传送以外，PDBToPID 必须要做的唯一一件事情就是将当前进程数据库指针与 ObsfucatorDWORD 执行 XOR，很明显这是 Microsoft 方面企图阻止研究者追究系统数据结构的实质。然而，象我将在这章末尾的 Win32WLK 码里所示的一样，这只不过是一个微小的，暂时的障碍（注：考虑二进制 XOR 的传递特性）。

顺便提一下，如果你迫切想知道 ObsfucatorDWORD 数值是从哪儿来的，一但你知道它是在系统每次启动时被算出来的，你会感到惊讶。为使事情复杂，不仅进程数据库而且连线程数据库也是被此 ObsfucatorDWORD“守卫”着的。后面我将向你展示 GetCurrentThreadID 函数与 GetCurrentProcessID 函数是多么不可思议地相似。

总之，进程柄很象文件柄，它是不透明的，且在定义它的进程文本之外没有意义。另外，进程 ID 在所有进程中都是一个值，它实质上是指向进程数据库结构的指针，虽然 Microsoft 已采取步骤来“Obsfucate”此事实（他们选择的单词，不是我）。此章末尾的 Win32WLK 程序揭示了将进程 ID 转变为可用指针的万能转化公式。

如果你曾经看过 TOOLHELP32 Process32First 和 Process32Next 函数，你可能会注意到 PROCESSENTRY32 结构内的 th32ProcessID 字段。这些和 GetCarrentProcessId 回送的数值有什么联系吗？还好，答案是肯定的。Win32WLK 程序利用这些东西让 TOOLHELP32 处理一些反复通过系统的进程和线程的错乱工作。

Windows 95 进程数据库（PDB）

在 Windows 95 内，每个进程数据库都是一块从 KERNRL32 共用内存堆中分配出来的内存。KERNEL32 经常用缩写 PDB 代替冗长的词组“process database”。不幸的是，在 Win16 内，PDB 和所有程序都有的 DOS PSP 是同义词，这不就乱了吗？是的！我将在本章的 KERNEL32 范围内使用 PDB。每个 PDB 被考虑为一个 KERNEL32 对象，这被结构的第一个 DWORD 内的数值（K32OBJ_PROCESS）证明了。来自 WIN32WLK 程序的 PROCDB.H 文件给出了一个 PDB 结构的 C 型观点。让我们仔细看看这些字段。

00h　　DWORD　　Type

此 DWORD 包含数值 5，某个进程的 KERNEL32 对象类型。

04h　　DWORD　　cReference

此 DWORD 是进程的参考计数，这是当前进程结构数量（例如，他们有一个进程的开式句柄）。

08h　　DWORD　　un1

此 DWORD 的含义还不知道，它可能是一个 KERNEL32 对象头标的标准部分，它通常为 0。

0Ch　　DWORD　　pSomeEvent

此 DWORD 是指向一个事件对象(K32OBJ_EVENT)的指针，事件对象被传送给象 WaitForSingleObject 这样的函数。当你将一个进程柄传送给一个 WaitForSingleEvent 函数系列时，看起来此事件确实是被拜访的对象。

10h　　DWORD　　TerminationStatus

此 DWORD 是通过调用 GetExitCodeProcess 回送的数值，进程出口代码是从主程序或 WinMain 函数回送的数值。换句话说，是进程调用 ExitProcess 或 TerminateProcess 时给定的。当进程仍在运行时，它的出口代码是 0x103(STILL_ACTIVE)。

14h　　DWORD　　un2

还不知道此 DWORD 的意思，它通常为 0。

18h　　DWORD　　DefaultHeap

此 DWORD 含有缺省进程堆的地址。GetProcessHeap 为当前进程回送此数值。

1Ch　　DWORD　　MemoryContext

此 DWORD 是指向进程的内存文本的指针。内存文本含有页面目录映射，这对提供一个 4GB 地址空间内有自己区域的进程很有必要。第五章详细介绍了内存文本。

20　　DWORD　　flags

这些标志在下表中均有描述。

标志名	位值	说　　明
fDebugSingle	0x00000001	如果进程正在被调试则设置
fCreateProcessEvent	0x00000002	在调试进程启动之后设置
fExitProcessEvent	0x00000004	可能在调试进程结束时设置
fWin16Process	0x00000008	一个 16 位程序
fDosProcess	0x00000010	一个 DOS 程序

fConsoleProcess	0x00000020	一个控制台(文本)Win32 程序
fFileApisAreOem	0x00000040	参看 SetFileApisToOEM()
fNukeProcess	0x00000080	
fServiceProcess	0x00000100	例如 MSGSRV32. EXE
fLoginScriptHack	0x00000800	可能是 Novell 网络注册进程
fSendDLLNotifications	0x00200000	
fDebugEventPending	0x00400000	例如调试中停止
fNearlyTerminating	0x00800000	
fFaulted	0x08000000	
fTerminating	0x10000000	
fTerminated	0x20000000	
flnitError	0x40000000	
fSingaled	0x80000000	

24h　　DWORD　　pPSP

此 DWORD 保存着为进程建立的 DOS PSP 的线性地址，这个字段 Win16 和 Win32 都设置了，此字段内的线性地址通常低于 1MB(真正形式的 DOS 码所能达到的最多地址)，参看 28h 字段。

28h　　WORD　　PSPSelector

此 WORD 是一个指向进程服务的 DOS PSP 的选择器，Win16 和 Win32 应用程序都有 DOS PSP，参看 24h 字段。

2Ah　　WORD　　MTEIndex

此 WORD 含有一个进入全局模块表(PModuleTableArray)的索引，进入模块表索引访问的 IMTE 是此模块的 IMTE，先前讨论过 IMTE 和全局模块表。

2Ch　　WORD　　cThreads

此字段是属于进程中线程的数量。

2Eh　　WORD　　cNotTermThreads

此字段保存进程还没有终止的线程数量，在至今所有已知的例子中，这个 WORD 总是和 2Ch 字段中的 WORD 有同样的值。

30h　　WORD　　un3

此 WORD 的意思不知道，它总是为 0。

32h　　WORD　　cRing0Threads

此 WORD 保存 VMM. VXD 管理的 ring 0 线程的数量，对一般应用程序来说，这个数值和 2Ch(cThreads)字段的一样。然而，在特殊的 KERNEL32. DLL 进程情况下，此字段比 cThreads 字段多 1 位。

34h　　HANDLE　　HeapHandle

此 DWORD 保存着一个堆表的堆柄，(还可能有别的什么)此字段看起来总是含有 KERNEL32 共用堆柄。

38h　　HTAST　　W16TDB

此 DWORD 保存着与这个进程有关的 Win16 Task Database(TDB)选择器，Win16 和 Win32 应用程序都有 TDB 选择器且都保留着有效任务数据库。

3Ch　　DWORD　　MemMapFiles

这是一个指向进程使用的内存变址文件表中堆码的指针，每个内存变址文件都对应着表内一个节点，每个节点的格式是：

DWORD　　内存变址区的基地址
DWORD　　指向下一节点的指针，或 0

40h　　PENVIRONMENT_DATABASE　　pEDB

此 DWORD 是指向环境数据库的指针，环境数据库含有当前目录、环境、进程命令行、“标准”句柄等等，在后面的“环境数据库”一节我将介绍环境的格式。

44h　　PHANDLE_TABLE　　pHandleTable

此字段是指向一个进程柄表的指针，所有的句柄(文件柄、进程柄、事件柄等等)都得进入句柄表。DOS/Win16 等效于 Win32(柄)表的是 DOS 系统文件柄(SFT)。然而，DOS SFT 适用于整个系统，而 Win32 进程柄表只适用于它自己的进程。在“进程柄表”一节介绍了 Win32 柄表的格式。

48h　　PPROCESS_DATABASE　　ParentPDB

此 DWORD 是一个指向 PROCESS_DATABASE 的指针。此 PROCESS_DATASE 是为建立进程的进程服务的。

4Ch　　PMODREF　　MODREFlist

此字段指向进程模块表头，这是前面“MODREF 结构”一节所讲的 MODREF 的链接表。

50h　　DWORD　　ThreadList

这是一个指向属于进程的线程表的指针，此表是 Listmgr.c 型的(此型表的确切格式我还不知道)。

54h　　DWORD　　DebuggeeCB

此 DWORD 看来象个调试文本块，当进程被调试时，此字段指向一块 2GB 上部的内存，此块内存包含一个指向调试的进程数据库的指针。

58h　　DWORD　　LocalHeapFreeHead

这个 DWORD 指向进程隐式堆内的自由表头，第五章介绍了进程堆的格式和自由表。

5Ch　　DWORD　　InitialRing0ID

此 DWORD 的意思我不懂，它总是为 0。

60h　　CRITICAL_SECTION　　crst

此字段是一个被不同的 API 函数使用的 CRITICAL_SECTION，这些函数是为同一进程内的同步线程服务的。

78h　　DWORD　　un4[3]

这 3 个 DWORD 总是被置为 0，它们的意思我现在还不知道。

84h　　DWORD　　pConsole

如果此进程使用这个操作台(即，如果是一个文本方式进程)，那么此 DWORD 就指向用来输出的操作台对象(K32OBJ_CONSOLE)。

88h　　DWORD　　tlsInUseBits1

这 32 位代表着 32 个最低 TLS(Thread Local Storage)标志(index)的状态。如果某位被设置了，那么 TLS 标志就被使用了，每个相继的 TLS 标志由依次增大的位值表示，例如：

```
TLS index:0 0x00000001
TLS index:1 0x00000002
TLS index:2 0x00000004
```

后面的“Thread Local Storage”一节详细介绍了有关内容。

8Ch　　DWORD　　tlsInUseBits2

此 DWORD 代表自 32 到 63 位的 TLS 标志的状态。

90h　　DWORD　　ProcessDWORD

此字段的含意目前还不明白，虽然有一个尚未证明的 API(GetProcessDword)抽取它的数值。

94h　　PPROCESS_DATABASE　　ProcessGroup

此字段为 0 或指向进程群中的主进程，进程群是同类进程的汇集。一但此群被破坏，它内部所有的进程便都破坏了，一般来说，每个进程被认为在它自己的群中，且此字段指向进程本身的 PDB(一个循环标记)。如果一个进程正在进行调试，它就属于调试程序的进程群。

98h　　DWORD　　pExeMODREF

此字段指向 EXE 的 MODREF(模块表入口)。先前曾讲过 MODREF，特别是 EXE 的 MODREF 是表头 MODREF，所以此字段与 4Ch 字段相匹配，除非进程已通过 LoadLibrary 或 LoadModule 装载了附加的 DLL。

9Ch　　DWORD　　TopExcFilter

此 DWORD 为进程保存着“Top Exception Filter”，如果没有其它异常处理器来处理异常情况，可调用其进行处理。其值通过 SetUnhandledExceptionFilter 函数来设置，后面将讨论结构异常处理。

A0H　　DWORD　　BasePriority

这个 DWORD 为进程保存着进度次序。Windows 95 支持 32 个优先级，分成 4 个级别，Windows 95 支持下面的优先级别：

Idle	4
Normal	8
High	13
Realtime	18

在每个级别内，缺省优先级的上下级的优先权不同，后面详细介绍。

A4h　　DWORD　　HeapOwnList

此字段指向进程堆链接表头，每个进程有一独立堆；通过调用 GetProcessHeap 可检索堆柄。然而，进程可以通过调用 HeapCreate 建立附加堆。此堆一旦建立就被放入进程的堆链接表。第五章详细介绍了这方面的问题。

A8h　　DWORD　　HeapHandleBlockList

进程堆内的可移动内存块被嵌入堆的可移动柄表管理着，此字段是一个指向

缺省进程堆内可移动柄表头的指针，第五章详细介绍了可移动柄表。

ACh　　DWORD　　pSomeHeapPtr

此字段的确切含义还不知道，通常为 0，否则，它就是一个指向缺省进程堆内移动柄表块的指针。

B0h　　DWORD　　pConsoleProvider

此字段也是 0，或一个指向 KERNEL32 控制台对象的指针。它好象在控制台方式 Win32 进程时总是 0，而在 WINOLDAP 进程时不是 0。WINOLDAP 是管理视窗内 DOS 程序的视窗进程。

B4h　　WORD　　EnvironSelector

此 WORD 保存一个指向进程环境的选择器，此选择器的基地址和环境数据库内 pszEnvironment 字段里的线性地址一样。

B6h　　WORD　　ErrorMode

此字段含有 SetErrorMode 函数设置的数值，KERNEL32 内的 SetErrorMode 继承了 KERNEL386 的 SetErrorMode，所以此字段仅仅反映进程的 Win16 错误模块数值，错误模块数值有：

SEM_FAILCRITICALERRORS
SEM_NOALIGNMENTFAULTEXCEPT
SEM_NOGPFAULTERRORBOX
SEM_NOOPENFILEERRORBOX

B8h　　DWORD　　pevtLoadFininished

此 DWORD 指向一个 KERNEL32 事件对象(K32OBJ EVENT)。

BDh　　WORD　　UTState

还不知道此字段的含义，但从名字上来看，它可能和 Universal Thunks 有关系。通常被置为 0。

在所有这些进程数据库字段中，需要注意有关 DOS 字段的数量。一个 PSP 选择器和一个线性地址同时为 DOS PSP(通常低于 1MB)服务。看看视窗将 INT 21h 变成 Virtual 86 型 DOS 码这种情况的次数，就不十分奇怪了。(参考《Unauthorized Windows 95》第八章，可以彻底证明将 INT 21h 发送给 DOS 不十分奇怪)。Windows NT 进程数据库含有所有进程的 PSP 信息是不大可能的。DOS 看来是不会到此结束的，至少在从 Windows 1.x 码发展来的平台上还没有结束。现在我们已经看了什么是进程数据库，让我们看一些有关进程函数伪码。

GetExitCodeProcess 和 IGetExitCodeProcess

GetExitCodeProcess 检索进程的终端状态，该进程是由 hProcess 句柄描述的。主函数可有效地确认并传送作为第二个参数的指针，实码是 IGetExitCodeProcess，在一些标准线程同步和录入与许多相关线程函数相适的代码之后，代码就取得 hProcess 参数并检查指向 PROCESS_DATABSE 的关键指针。hProcess 是一个句柄，这意味着索引进程柄表和抽取进程指针。x_ConvertHandleToK32Object 句柄的另加工任务，与进程数据库使用次数成正比。

利用 PPROCESS_DATABASE 指针，函数可抽取 TerminationStatus 字段的数值并存入调用程序的缓冲区，为了简单，IGetExitCodeProcess 减少了进程对象的使用次数，并脱离“必须完整”状态。

GetExitCodeProcess 伪码

```
// Parameters
//      HANDLE  hProcess;
//      LPDWORD lpdwExitCode;

    Set up structured exception handling frame

    if ( lpdwExitCode )         // It a non-null pointer was passed, verify
        EAX = *lpdwExitCode;    // that the DWORD it points to can be written.

    Remove structured exception handling frame

    goto IGetExitCodeProcess;
```

IGetExitCodeProcess 伪码

```
// Parameters
//      HANDLE  hProcess;
//      LPDWORD lpdwExitCode;
// Locals:
//      PPROCESS_DATABASE ppdb;
//      BOOL    retValue;

    retValue = TRUE;        // Assume successful return.

    x_EnterMustComplete();  // Prevent us from being interrupted.
                            // Increments ptdbx->MustCompleteCount.
     x_LogSomeKernelFunction( function number for GetExitCodeProcess );

    // Get a pointer to the PROCESS_DATABASE struct
    ppdb = x_ConvertHandleToK32Object( hProcess, 0x80000010, 0 );

    if ( ppdb )
    {
```

```
    // Save away exit status.
    *lpdwExitCode = ppdb->TerminationStatus;
    x_UnuseObjectWrapper( ppdb );   // Decrement usage count.
}
else....                // Opps! No process database.
{
    retValue = FALSE;
}

// Call the API logging function again (???).
x_LogSomeKernelFunction( function number for GetExitCodeProcess );

LeaveMustComplete();    // Decrements ptdbx->MustCompleteCount.

return retValue;
```

SetUnhandledExceptionFilter

SetUnhandledExceptionFilter 设置 KERNEL32 的 UnhandledExceptionFilter 调用的函数地址，但只是在没有其它异常筛选程序可供选择来处理异常的情况下使用，此功能储存进程数据库内 TopExcFilter 字段的当前值，然后用传送进来的参数值替换此数值。此功能回送 TopExcFilter 的先前值。

SetUnhandledExceptionFilter 伪码

```
// Parameters:
//      LPTOP_LEVEL_EXCEPTION_FILTER  lpTopLevelExceptionFilter
// Locals:
//      LPTOP_LEVEL_EXCEPTION_FILTER prevValue;

    // Save old value.
    prevValue = ppCurrentProcess->TopExcFilter;

    // Stuff in new value.
    ppCurrentProcess->TopExcFilter = lpTopLevelExceptionFilter;

    return prevValue;       // Return old value.
```

OpenProcess

OpenProcess 记录一个进程 ID 并回送一个与该进程有关的句柄，然后此句柄可以被传送给象 ReadProcessMemory 和 VirtualQueryEx 这样的函数。当你把此函数和 TOOLHELP32 的能力连接起来以得到系统内任何进程的进程 ID 时，你得到的是一个强有力的联合。有点奇怪的是 Windows 95 允许你打开进程柄而不允许打开线程柄。也许 Microsoft 认为允许打开线程柄引起的损害太大了。

OpenProcess 首先将 ID 参数转化为一个 PPROCESS_DATABASE。由于将进

程 ID 转化为进程指针的算法和将线程 ID 转化为线程指针的算法完全一样，所以 OpenProcess 检查以确保它有 PPROCESS_DATABASE 指针。

OpenProcess 接下来保存标志参数，以保证它有唯一合法且(或)需要的标志。最后，OpenProcess 调用一个内部函数以在当前进程柄表内开一小口，并将 PPROCESS_DATABASE 指针放置在此口内。

OpenProcess 伪码

```
// Parameters:
//  DWORD   fdwAccess;
//  BOOL    fInherit;
//  DWORD   IDProcess;
// Locals:
//  PPROCESS_DATABASE ppdb;
//  DWORD   flags;

    x_LogSomeKernelFunction( function number for OpenProcess );

    // Convert the process ID to a PPROCESS_DATABASE.
    ppdb = PidToPDB( IDProcess )

    if ( !ppdb )
        return 0;

    if ( ppdb->Type != K32OBJ_PROCESS ) // Make sure thread ID not passed.
    {
        InternalSetLastError( ERROR_INVALID_PARAMETER );
        return 0;
    }
    flags = fdAccess & 0x001FFFBF;  // Turn off all non-allowed flags.
                                    // Flags like PROCESS_QUERY_INFORMATION
                                    // and PROCESS_VM_WRITE are allowed.
    if ( fInherit )
        flags |= 0x80000000;

    flags |= PROCESS_DUP_HANDLE;    // Always pass. PROCESS_DUP_HANDLE

    // Allocates a new slot in the handle table of the current process.
    // The slot contains the ppdb pointer.
    return x_OpenHandle( ppCurrentProcess, ppdb, flags );
```

SetFileApisToOEM

此函数改变相关文件 KERNEL32 函数解释文件名的方法。KERNEL32 用 ANSI 字符串表示文件名，通过调用 SetFileApisToOEM，程序可用 OEM 字符串代替它。做为一个实例，可以看看前面讲过的 GetModuleFileName 和 GetModuleHandle 函数。

实际上，此函数不能再简单了。它占有一个指向当前进程的进程数据库指针

并打开了标志段的 fFileApisAreOem 标志。

SetFileApisToOEM 伪码

```
x_LogKernelFunction( function number for SetFileApisToOEM )
ppCurrentProcess->flags |= fFileApisAreOem;
```

环境数据库(The Environment Database)

进程数据库内的偏移量 40h 是一个指向重要数据结构的指针，此数据结构也含有进程相关信息。在 KERNEL32 内部使用时，此指针的名字是 pEDB，我将它解释为“Pointer to Envioment Database”和 PROCESS_DATABASE 结构一样，我已经给出了 PROCDB. H 文件内 ENVIRONMENT_DATABASE 的结构。现在让我们看看这些字段：

00h　PSTR　pszEnviroment

此字段指向进程环境。此环境是典型 DOS 环境(字符串＝数值，在很多项之间有分号，象字符串＝数值；字符串＝数值)。进程环境存在于每个进程数据区的内存块中，并通常恰好在装载 EXE 模块的地址之上。

04h　DWORD　un1

不懂此 DWORD 的含义，它总是为 0。

08h　PSTR　pszCmdLine

此字段指向传送给 CreateProcess 以启动此进程的命令行。在很多情况下，命令行就是进程的 EXE 的完整文件名，但某些情况下，它是指向空串(\0)的指针。

0Ch　PSTR　pszCurrDirectory

这是一个指向进程当前目录的指针。

10h　LPSTARTUPINFOA　pStartupInfo

此指针指向进程的 STARTUPINFOA 结构，此结构定义在 WINBASE. H 内。STARTUPINFOA 结构被传送给 CreatProcess 以说明进程的视窗尺寸、标题、标准的文件柄等等。此字段指向 STARTUPINFOA 结构的一个付本。

14h　HANDLE　hStdIn

这是进程用来表示标准输入设备的文件柄，如果不被使用(例如，如果这是一个 GUI 应用程序)，那么句柄值就是-1。

18h　HANDLE　hStdOut

这是进程用来表示标准输出设备的文件柄。如果不被使用(例如,如果这是一个 GUI 应用程序),句柄值就是-1。

1Ch　HANDLE　hStdErr

这是进程用来表示标准错误设备的文件柄,如果不被使用(例如,如果这是一个 GUI 应用程序),句柄值就是-1。

20h　DWORD　un2

不懂它的意思,它总是为 1。

24h　DWORD　InheritConsole

此字段大概是指明进程是否从它的源进程那里继承了控制台(相对于获得它自己的控制台)。参考 CreateProcess 函数的 CREATE_NEW_CONSOLE 标志。就我的观察,此字段总为 0。

28h　DWORD　BreakType

此字段很有可能是指明应该处理什么样的控制台事件的(CTRL+C 等等)。在我运行的程序内,它通常为 0,偶尔也会被置为 0xA。

2Ch　DWORD　BreakSem

此字段一般为 0,但如果调用 SetConsoleCtrlHandler,此 DWORD 就指向一个 KERNEL32 信号对象(K32OBJ_SEMAPHORE)。

30h　DWORD　BreakEvent

此字段一般为 0,但如果调用 SetConsoleCtrlHandler,此 DWORD 就指向一个 EVENT 对象(K32OBJ_EVENT)。

34h　DWORD　BreakThreadID

此字段一般为 0。然而,当调用 SetConsoleCtrlHandler 时,此 DWORD 指向安装了处理器的线程的线程对象(K32OBJ_THREAD)。

38h　DWORD　BreakHandlers

此字段一般为 0,但当调用 SetConsoleCtrlHandler 时,此 DWORD 指向一个从 KERNEL32 共用堆分配来的数据结构,此数据结构是一个控制台操纵处理器表。

现在让我们再看一些进程函数的伪码,这次和我们刚看过的 ENVIRONMENT_DATABASE 有关。

GetCommandLineA

在 GetCommandLineA 码里真的没什么好解释的，此函数回送储存在环境数据库内的命令行指针。

GetCommandLineA 伪码

```
return ppCurrentProcess->pEDB.pszCmdLine
```

GetEnvironmentStrings

关于 GetEnvironmentStrings 也没什么好说的，和 GetCommandLineA 一样，它只是回送环境数据库内的有关指针。然而，实际的执行和 SDK 文件描述的是两码事情，SDK 文件说：

当 GetEnvironmentStrings 被调用时，它给一环境字符串配置内存，当此环境字符串不再需要时，应该调用 FreeEnvironmentStrings。

虽然这对 Windows NT 来说还可以，但它决不适合 Windows 95。

GetEnvironmentStrings 伪码

```
return ppCurrentProcess->pEDB.pszEnvironment
```

FreeEnvironmentStringsA

此函数有点意思，由于 GetEnvironmentStrings 事实上并没有配置任何内存，所以 FreeEnvironmentStringsA 就没有事情可干了。然而，为了活动活动，此函数检查输入参数串，看看它是否与指向来自环境数据库的指针相匹配。如果不匹配，FreeEnvironmentStringsA 就将 LastError 值赋于 ERROR_INVALID_PARAMETER。

FreeEnvironmentStringsA 伪码

```
// Parameters:
// LPSTR  lpszEnvironmentBlock;

    x_LogSomeKernelFunction( function number for FreeEnvironmentStringsA );
```

```
    if ( ppCurrentProcess->pEDB.pszEnvironment != lpszEnvironmentBlock )
    {
        InternalSetLastError( ERROR_INVALID_PARAMETER );
        return FALSE;
    }

    return TRUE;
```

GetStdHandle

和它的写法一样，它就是执行此功能的。给定设备 ID 后，此函数就检索环境数据库的有关文件柄。如果传送了假设备 ID，此操作就失败并设置最后错误码。

GetStdHandle 伪码

```
// Parameters:
//      DWORD   fdwDevice
// Locals:
//      PENVIRONMENT_DATABASE pEDB;

    pEDB = ppCurrentProcess->pEDB;

    if ( fdwDevice == STD_INPUT_HANDLE )
        return pEDB->hStdIn;
    else if ( fdwDevice == STD_OUTPUT_HANDLE )
        return pEDB->hStdOut;
    else if ( fdwDevice == STD_ERROR_HANDLE )
        return pEDB->hStdErr;

    InternalSetLastError( ERROR_INVALID_FUNCTION );

    return 0xFFFFFFFF;
```

SetStdHandle

此函数比 GetStdHandle 有趣，首先验证句柄是有效的 KERNEL32 对象句柄。怎样验证呢？通过调用 x_ConvertHandleToK32Object，如果是有效句柄，它就回送一个指向有关 KERNEL32 对象指针。SetStdHandle 从来不用 K32 对象指针，仅需做一个有关 NULL 值的简单测试。在验证了 hHandle 参数之后，其余的代码就把 hHandle 存入环境数据库结构内的合适字段。

SetStdHandle 伪码

```
// Parameters:
//      DWORD  IDStdHandle
//      HANDLE  hHandle
// Locals:
//      PVOID               pK32Object;
//      PENVIRONMENT_DATABASE pEDB;

    if ( hHandle == STD_INPUT_HANDLE )
    {
        pK32Object =
            x_ConvertHandleToK32Object( hHandle, 0x00002140, 0x00000020 );
    }
    else if ((hHandle == STD_OUTPUT_HANDLE) || (hHandle == STD_ERROR_HANDLE))
    {
        pK32Object =
            x_ConvertHandleToK32Object( hHandle, 0x00002140, 0x00000110 );
    }
    else
    {
        InternalSetLastError( ERROR_INVALID_FUNCTION );
        return FALSE;
    }

    if ( pK32Object )
    {
        pEDB = ppCurrentProcess->pEDB;

        if ( IDStdHandle == STD_INPUT_HANDLE )
            pEDB->hStdIn = hHandle;
        else if ( IDStdHandle == STD_OUTPUT_HANDLE )
            pEDB->hStdOut = hHandle;
        else
            pEDB->hStdErr = hHandle;
    }

    return TRUE;
```

进程柄表

PROCESS_DATABASE 内的偏置 44h 是一个指向相应进程的句柄表指针。在这里我用了“柄”一词，通过进程柄表可涉及到有关内容。除了文件柄外，Windows 95 还为其它系统对象设立了句柄。进程、线程、事件和互斥信号就是几个例子，事实上，在前面“KERNEL32”对象部分列出的每个 KERNEL32 对象都有句柄。

从理论上讲，句柄值应该是“含糊的”，即句柄值不能告诉你任何有关的东西。例如，告诉你句柄值 5，你分不清它是文件柄还是互斥信号柄。然而，一旦你明白了 Windows 95 内的进程柄表，你就能轻易地把句柄值和它所涉及的东西联系起来。

Windows 95 进程柄表非常简单，此表的第一个 DWORD 是进入当前表的句

柄表的最大值。进程启动时的缺省值是 0x30(48)个句柄，这并不意味着进程只限于 48 个开句柄，当进程打开的句柄比将要放进当前柄表的句柄还多时，KERNEL32.DLL 就重新配置内存句柄块以便增大句柄表。增加量在 0x10 位以内。例如，在扩展了最初的 0x30 个句柄输入后，重新配置的句柄表有 0x40 个输入句柄的数量好象没有明显的上限，我写了一个小程序以循环打开文件柄，在此之前它配置的句柄超过了 255 个(旧 DOS 上限)。

紧接第一个 DWORD 的是一个 8 位结构数组，每个结构含有两个 DWORD：

```
DWORD       flags(标志)
DWORD       pK32Object
```

第二个字段(pK32Object)是指向 17 个 KERNEL32 对象之一的指针，这 17 个对象我在前面的“KERNEL32 对象”部分讲过。第一个 DWORD 是对对象的访问控制标志，标志的含意依赖于此条目指向什么类型的对象。例如，如果指向进程对象(K32OBJ_PROCESS)，标志就是来自 WINNT.H(PROCESS_TERMNATE，PROCESS_VM_READ 等等)的 PROCESS_xxx 标志。

现在，你可能在怀疑句柄值代表什么。如果你认为句柄值是进入进程柄表的索引，那么你对了，一旦你明白了这一点，就能轻易地将值和它所指的 KERNEL32 对象联系起来。没用的句柄表条目的两个 DWORD 都置 0，当配置一个新句柄时，KERNEL32 用表中第一个空位的索引。虽然浏览进程柄表不是必须的，但 WIN32WLK 程序提供了此功能。用 Win32Wlk 时，注意 KERNEL32 使用的句柄数量和类型。

线程(THREAD)

目前已经讲了模块和进程，讲完了线程就可以结束我们关于基本 KERNEL32 数据结构的行程了。虽然进程主要表示象文件表、地址空间等等的所有权，线程却代表模块码的执行。看见它们之间的联系了吧？很难孤立一个或不掺杂其它东西。例如，先前在进程的描述中，我不得不提到线程和同步对象。

概括地说，当其它部分等待一些外部事件发生时，线程是保存程序运行的捷径。通过把进程执行的不同任务分成线程，可能会消除类似于反复定时询问的情况。在等待事件(象敲键盘)发生时，反复定时询问会浪费大量的 CPU 时间。

在任何给定的时间，线程处于三个基本状态之一。线程运行时处于第一个状态，线程的寄存器保留在 CPU 寄存器内，当一个线程运行时，系统内所有其它的线程都处于停止状态。第二个状态是“准备运行”状态。此时，线程没有理由不运行，除非其它线程正在使用 CPU，在适当的时候，准备运行的线程将得到 CPU 的控制权。

第三个状态是锁定状态。当线程被锁定时，是在等待某事发生。调度程序一直等到事情发生后才允许线程运行。锁定线程的东西称为同步对象，Windows 95 的同步对象有临界段(critical section)、互斥(mutexe)、事件(event)和信号

(semaphore)。

在此书中，我没有像介绍进程、线程、模块一样介绍同步对象。有很多好书，象Jeffrey Richter 的《Advanced Windows》详细讲述了同步对象的使用；如果有兴趣可以参考它们。在本书中，你不得不承认同步对象的存在并同意我所讲的。

最初，每个进程开始于一个线程。如果进程允许，可以建立附加线程以便CPU 同时执行进程码的不同部分。这方面的典型例子是字处理器，当字处理器要打印时，程序引入另一个处理所有打印事务的线程，并允许原来的线程继续与用户保持接触，所以在打印时用户可以继续工作。

当然，如果你熟悉 CPU 的基本结构，你知道只有一个 CPU 的机器根本不可能同时在多个配置内执行。多个线程同时运行的假象是通过 VMM 调度程序实现的，它使用硬件计时器和复合规则集以实现不同线程之间的快速转换。

Microsoft 宣称 Windows 95 给调度使用了 20 毫秒时间片。也就是说，在排除其它因素(如线程优先级)情况下，每个线程运行 20 毫秒时系统进行线程切换。关于线程调度和 VMM 调度此书讲的并不深入，和同步对象一样，这是其它书的事。

和进程一样，每个线程代表着 KERNEL32. DLL 内部一个从共用 KERNEL-32 堆分配来的内存块，此内存块保存所有 KERNEL32 需要为线程保留的信息(实际上，在块内保存的是一些指向块外信息的指针)，此内存块在本书内称为线程数据库(TDB)(注意有时 Microsoft 用 TDB 表示任务数据库)。和进程数据库一样，线程数据库也是一个 KERNEL32 对象，它的第一个 DWORD 含有值 6，标志着它是一个 K32OBJ_THREAD 对象。

如果你是一个曾经深入接触 DDK 或使用过 WDEB386 和 SoftIce/W 的高级程序员，你可能遇到过另一个称为 THCB(线程控制块)的相关线程数据结构，THCB 是线程的 ring 0 表示。在 Windows 95 中，线程被分开的 ring 0 和 ring 3 数据结构表示。ring 0 部分，例如 VMM. VXD，主要通过线程控制块与线程发生联系。ring 3 部分，例如 KERNEL32. DLL，则主要利用线程数据库，我将在下面讲到“线程数据库”，本章着重讲 ring 3 线程的行为和结构，对 ring 0 部分就不讲了。

虽然进程是拥有实际东西的主要 K32，线程也是拥有(或与之发生关系)某些项目的。首先，线程有一个寄存器，当线程正在执行时，它的寄存器储存在 CPU 的寄存器内。即，线程的 EIP 值是 EIP 寄存器内的值。当线程处于停止状态时，其寄存器需要储存在内存的其它地方。因此，每个线程有一个指向内存缓冲区的指针，在那儿可以储存中止线程的寄存器。和每个线程都有密切联系的另外一件东西就是进程，同一进程内部的所有线程对进程拥有的东西都可以访问。例如，进程有一个内存文本和一个专用地址空间，进程内所有的线程都在同一个地址空间内运行，进程还有一个涉及文件、事件、控制台和内存映射文件等的句柄表，进程内所有的线程共用同一个句柄值。例如，如果句柄值 3 代表一个内存映射文件，那么进程内任何一个线程都可以用句柄值 3 代表那个内存映射文件。

线程还有其它许多东西。每个线程有自己的堆栈区，自己的视窗消息队列，自己的 Thread Local Storage 值集和自己的结构异常处理链。另外，线程还获取或释放它在执行中使用的各种同步对象的所有权。不久我们会看到线程数据库，那时

线程所有的东西就一目了然了。

什么是线程柄？
什么是线程 ID？

先前我曾讲过进程柄与进程 ID 的区别，这对线程柄和线程 ID 同样适用，只需把“进程”换为“线程”就行。GetThreadHandle 函数回送一个凡是能用真正线程柄的地方都能用的稳定值(Microsoft 称为“伪柄”)。

GetCurrentThread 伪码

```
x_LogSomeKernelFunction( function number for GetCurrentThread );

return 0xFFFFFFFE;
```

类似于 GetCurrentProcessId，GetCurrentThreadId 回送一个指向当前线程数据库的指针，除非 KERNEL32 编码员故意弄乱回送数值：

GetCurrentThreadId 伪码

```
return TDBToTid( ppCurrentThread );
```

KERNEL32 是怎样把线程回送数值弄乱的呢？让我们看看：

TDBToTid 伪码

```
// Parameters:
//      THREAD_DATABASE * ptdb

    if ( ObsfucatorDWORD == FALSE )
    {
        _DebugOut( "TDBToTid() Called too early! Obsfucator not yet"
                   " initialized!" );
        return 0;
    }

    if ( ptdb & 1 )
    {
        _DebugOut( "TDBToTid: This TDB looks like a TID (0%lxh) Do a"
                   " stack trace BEFORE reporting as bug." );
    }

    // Here's the key! XOR the obsfucator DWORD with the thread database
    // pointer to make the TID value.

    return ptdb ^ ObsfucatorDWORD;
```

KERNEL32 用一个 ObsfucatorDWORD 将进程和线程数据库“换算”成 ID，一旦你明白了 ObsfucatorDWORD 值是什么。你就可以用它将进程或线程 ID 转换成有用的指针。

线程数据库

线程数据库是一个从 KERNEL32 共用堆分配来的 KERNEL32 对象(K32OBJ_THREAD 类型)。和进程数据库一样，线程数据库并不是直接连接成链形表的样子，来源于 WIN32WLK 的 THREADB. H 文件对线程数据库有一个 C 型结构定义，下面是线程数据库的格式：

00h　DWORD　Type

此 DWORD 含有数字 6，代表线程的 KERNEL32 对象类型。

04h　DWORD　cReference

此 DWORD 含有线程的参考计数，这是当前某些正在使用线程结构的数量。

08h　PPROCESS_DAIABASE　pProcess

这是一个指向线程所属进程的指针。

0Ch　DWORD　pSomeEvent

此 DWORD 是一个指向事件对象(K32OBJ_EVENT)的指针，事件对象被传送给象 WaitForSingleObject 这样的函数，当把一个线程柄传送给一个 WaitForSingleEvent 函数时，看起来此事件确实被访问了。

10h　DWORD　pvExcept

此 DWORD 是指向结构化异常处理链头的指针(后面将讲到结构化异常处理)注意，此字段还标志着任务数据库内的一个 TIB(线程信息块)结构的开始，TIB 结构在后面也要讲到。

14h　DWORD　TopOfStack

此 DWORD 保存线程所占堆栈区的最大(顶端)地址，为每个线程保留的典型地址空间是 1MB。

18h　DWORD　StackLow

这个 DWORD 保存线程所用堆栈区最低页地址。

1Ch　WORD　W16TDB

此 WORD 为 Win16 任务数据库保存 Win16 全局内存句柄(实际是个选择

器),就象第七章讲的,每个进程(Win16 或 Win32)有一个 16 位任务数据库段和一个 Win32 进程数据库。

1Eh　　WORD　　StackSelector16

Win32 码需要和一个 16 位堆栈匹配才能转换为 16 位码,此 WORD 保存一个选择器,它是 KERNEL32 在转换为 16 位码时做为 16 位堆栈选择器的。

20h　　DWORD　　SelmanList

这是一个指向线程的 SelmanList 的指针(Selman 是"Selector Manager"的缩写),KERNEL32 的 Selman 分量看来管理着选择器表,线程可分配此选择器不同的用途(例如实现 16 位和 32 位码之间的转换)。

24h　　DWORD　　UserPoinfer

此 DWORD 的确切含义还没弄明白,不管怎样,TIB 结构的文件编制者说此字段适用于应用程序。记住,TIB 结构属于线程数据库结构。

28h　　PTIB　　pTIB

此字段指向线程的线程信息块(TIB),在 Windows 95 中,TIB 属于线程数据库,所以此指针指向线程数据库中的另一字段(确切的说,是偏移量 10h 的 pvExcept 字段)。

2Ch　　WORD　　TIBFlags

此 WORD 含有这个 TIB 的标志,这些标志如下:

标志名	位值	说　明
TIBF_WIN32	0x0001	该线程来自一个 Win32 应用程序
TIBF_TRAP	Ox0002	异常处理分类

2Eh　　WORD　　Win16MutexCount

此字段和 Win16Mutex(又称 Win16Lock)有某些联系,一般此字段对 Win32 线程是-1,对 Win16 线程是 0。

30h　　DWORD　　DebugContext

如果和此线程有关的进程正在被调试,此字段就指向一个调试文本结构,此结构的格式还不清楚,但看起来它内部有调试进程的寄存器数值,如果进程没有被调试,此 DWORD 就是 0。

34h　　PDWORD　　pCurrentPriority

这个字段指向一个含有此线程当前优先级的 DWORD，此 DWORD 位于地址 OxC0000000 之上，此地址处于 VxD 地区。

38h　　DWORD　　MessageQueue

此 DWORD 的低 WORD 为线程的消息队列保存着一个 Win16 全局堆柄，消息队列表示视窗信息怎样在系统内移动，这将在第四章讲到。此字段与偏移量 1Ch 的 W16TDB 字段有密切关系。

3Ch　　PDWORD　　pTLSArray

此指针指向线程的 TLS 数组，此数组的各项条目被 TlsSetValue 函数簇使用。

40h　　PPRocess_DATABASE　　pProcess2

此 DWORD 含有一个指针，它指向与此线程有联系的进程，它和线程数据库内偏移量 08h 区的指针一模一样。

44h　　DWORD　　Flags

此 DWORD 存有线程的各种标志，下面是一些已知的：

标志名	位值	说　明
fCreateThreadEvent	0x00000001	线程被调试时设置
fCancelExceptionAbort	0x00000002	
fOnTempStack	0x00000004	
fGrowableStack	0x00000008	
fDelaySingleStep	0x00000010	
fOpenExeAslmmovableFile	0x00000020	
fCreateSuspended	0x00000040	CreateProcess 时设置
fStackOverflow	0x00000080	
fNestedCleanAPCs	0x00000100	APC 为异步过程调用
fWasOemNowAnsi	0x00000200	ANSI/OEM 文件函数
fOKToSetThreadOem	0x00000400	ANSI/OEM 文件函数

48h　　DWORD　　TerminationStatus

这是一个应该通过调用 GetExitCodeThread 回送的值。线程输出码是从线程执行开始的地方回送的数值。换句话说，它在线程调用 ExitThreda 或 TerminateThread 时就能被确定下来。线程处于运行状态时，它的输出码是 0x103 (STILL_ACTIVE)。

4Ch　WORD　TIBSelector

这个 WORD 是一个非常重要的字段，它含有一个注明当前线程的 TIB(线程信息块)的选择器，此 TIB 含有极重要的信息，象线程的异常处理器链头等。Windows 95 在线程之间进行切换时，将修改 FS 寄存器值，这样可以使当前线程利用 FS 寄存器指向的内存，不间断的检查有关自己的信息。

4Eh　WORD　EmulatorSelector

此 WORD 可能是指向一块内存的选择器，这块内存有线程当前 80387 仿真状态的信息，此数据区可能包括一个 FSAVE 类结构，机器使用一个协处理器时，此字段总是 0。

50h　DWORD　cHandles

不懂此 DWORD 的意思，它总是为 0。

54h　DWORD　WaitNodeList

如果线程正在等待一个或更多事件，此字段就指向储存于 VxD 的一个事件节点链接表，每个节点保存一个指向事件对象的指针，和一个指向正在处理事件的线程的指针。

58h　DWORD　un4

此 DWORD 的意思还没弄明白，它不是 0 就是 2。

5Ch　DWORD　Ring0Thread

此 DWORD 为线程保存一个指向 ring 0 线程控制块(THCB)的指针。

60　PTDBX　pTDBX

此字段指向一个 TDBX 结构，TDBX 结构是线程的 VWIN32. VXD 的再现。

64h　DWORD　StackBase

对 Win32 线程来说，此 DWORD 保存线程栈使用的最小地址，从最大栈地址中减去此数值，你就可以计算出栈占有多少地址空间了，对 Win32 线程来说，此字段是 0。

68h　DWORD　TerminationStack

此字段含有线程最初使用的 ESP 值，对 Win32 线程来说，此值和 TopOfStack 值(偏移量 14h)一样，对 Win16 线程，此字段保存一个恰位于共用 KERNEL32 堆之下的地址。

6Ch　DWORD6　EmulatorData

此字段大概是线程的 80387 仿真数据的 32 位线性地址，如果是这样，那么此字段和 EmulatorSelector 字段(偏移量 4Eh)有关。

70h　DWORD　GetLastErrorCode

此 DWORD 保存 GetLastError 为当前线程回送的数值，此数值可通过调用 SetLastError 设置。

74h　DWORD　DebuggerCB

如果一个线程起调试程序线程的作用(即：如果它调用WaitForDebugEvent)，此字段就含有一个指向调试程序使用的信息块的指针，此字段的信息包括指向调试程序的进程数据库、线程数据库的指针和指向被调试的线程数据库的指针。

78h　DWORD　DebuggerThread

如果此线程正被调试，此字段就含有一个 non_NULL 值，此数值的含义还不清楚，因为它短得不能做为有效指针。

7ch　PCONTEXT　ThreadContext

此指针指向一个 Intel CONTEXT 结构，此结构在线程停止时为它保存寄存器值，用 GetThreadContext 和 SetThreadContext 函数可对此结构进行读写。

80h　DWORD　Except16List

此 DWORD 的确切含义还没弄清楚，我测得它总为 0。

88h　DWORD　NegStackBase

如果你将此字段的值加到 StackBase 字段(偏移量 64h)上，你会得到 FFEF9000，别问我为什么。

8Ch　DWORD　CurrentSS

此 DWORD 为从 32 位码转换到 16 位码保存了一个 16 位栈选择器，此字段和与它相似的 StackSelector16 字段(偏移量 1Eh)有关，两个字段在使用中的区别目前还不知道。

90h　DWORD　SSTable

此字段是指向一个内存块的指针，此内存块含有转换为 16 位码时使用的 16 位栈的信息。

94h　DWORD　ThunkSS16

此 DWORD 含有转换用的另一个选择器值，在一些线程中，它和 StackSelector16 字段（偏移量 1Eh）的数值相匹配，而在其它线程中，它等于 CurrentSS 字段（偏移量 8Ch）的值。

98h　DWORD　TLSArray[64]

TLSArray 字段是一个有 64 个 DWORD 的数组，每个 DWORD 保存一个为给定的 TLS ID 回送的 TLSGetValue 数值。例如，数组中的第一个 DWORD 保存回送的 TLSGetValue(0)值，第二个保存 TLSGetValue(1)的值，等等。后面会讲到 TLS。

198h　DWORD　DeltaPriority

此 DWORD 保存线程各类优先级和拥有此线程的进程的优先级，此字段的典型值有：

THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_HIGHEST	1
THREAD_PRIORITY_ABOVE_NORMAL	2

19Ch　DWORD　un5[7]

此 DWORD 看来总是 0，含义不知道。

1B8h　DWORD　pCreateData16

如果非 0，此字段就指向一个有两个 32 位指针的结构，这两个指针是：

00h pProcessInfo 是一个 PPROCESS_INFORMATION
04h pStartupInfo 是一个 PSTARTUPINFO

然而，在我的测试中，pCreateData16 指针总是等于 0。

1BCh　DWORD　APISuspendCount

此字段的值每当调用 SuspendThread 时增加而当调用 ResumeThread 时减少。

1C0h　DWORD　un6

此 DWORD 的意思还不清楚。

1C4h　DWORD　WOWChain

此字段大概和 Windows 95 支持的 WOW(Windows on Windows)有关,Windows NT 通过 WOW 可以在 16 位应用程序的保留地址空间内运行它们,这防止了它们对 32 位应用程序潜在的破坏,在测试中此字段总是为 0。

1C8h　WORD　wSSBig

此字段含有一个做为栈段使用的 32 位选择器,此字段在测试中总是为 0。

1CAh　WORD　un7

不知道此 WORD 的意思,它可能就是一个用来保持以后字段 DWORD 均衡的填充数。

1CCh　DWORD　lp16SwitchRec

不知道这个 DWORD 的含义,它很可能与 Win16 转换程序有关。

1D0h　DWORD　un8[5]

这 5 个 DWORD 总是为 0,确切含义不知道。

1E4h　DWORD　pSomeCritSect1;

此字段指向一个关键段对象(K32OBJ_CRITICAL_SECTION)每个进程的关键段对象不同,这样做的目地还不清楚,此字段总是和 pSomeCritSect2(下面要讲)有同样的值。

1E8h　DWORD　pWin16Mutex;

此指针指向 KRNL386.EXE 内的 Win16Mutex。

1ECh　DWORD　pWin32Mutex;

此指针指向 KERNEL32.DLL 内的 Krn32Mutex。

1F0h　DWORD　pSomeCritSect2;

此字段指向一个关键段对象(K32OBJ_CRITICAL_SECTION)。

1F4h　DWORD　un9

此 DWORD 总是为 0,不知道确切含义。

1F8h　DWORD　ripString

你可能认为此字段是一个将在 FatalAppExit RIP 中用到的字符串的 PSTR,然而,在大多情况下,此字段为 0,就算不是 0,它也不指向字符串。

200h　DWORD　LastTlsSetValueEIP[64]

这个有 64 个 DWORD 的数组和线程数据库内偏移量 98h 的 TLS 数组类似，此数组中的每个 DWORD 代表一个 TLS 索引值，且每个 DWORD 都含有设置对应 TLS 索引值的 EIP，EIP 值来自于 TlsSetValue 设置的栈结构。

关于线程数据库，最后要注意的一点是：还有很多途径可以得到指向线程数据库的指针，除了先前讲的 XOR 方法，每个 Win16 任务数据库也含有一个指向线程数据库的指针，在 Win16 任务数据库的偏移量 54h 是任务（或进程）的第一个线程的线程数据库的线性地址。

线程信息块（TIB）

在线程库内，某些区段对于程序的运行相当有用。事实上，由于它们如此重要，Win32 设计者使他们在任何时候均可被立即访问而无需在线程库中查找。这些区段是由一个被称为线程信息库（KERNEL32 称其为 TIB）的结构所拥有的。Windows 95 TIB 的区段占据了线程库中的偏置 10h 到 3Ch。

那么应用代码是如何访问 TIB 的？如果你看过许多编译 Win32 代码的汇编语言输出，你可能注意到 FS 段位寄存器只使用了一点点。那么是不是 Win32 不准备从编程画面移走区段？尽管答案是肯定的，但在 Win32 形式下（Windows NT，Windows 95，和 Windows Win32s），设计者还是将 FS 寄存器指向了当前线程的线程信息块。事实证明，Win32 并不是第一个这样做的。OS/2 2.0 在 Win32 问世之前就已经这样做了。正如你所怀疑的，当 Windows 95 切换线程时，进度表需要更新寄存器以便包括指向新线程的 TIB 的选择器。

FS 寄存器和 TIB 的主要用途是增加结构异常处理链的入口（本章的后面我将谈到）。结构异常处理链的首部处于 TIB 中的偏置 0，因而当你查看使用 FS:[0] 的汇编代码时，你会知道它正进行与结构异常处理相关的动作。

Windows 95 TIB 中的其它两个广泛使用的区段是 pvQueue 和 pvTLSArray 区段（偏置分别为 28h 和 2Ch）。pvQueue 区段包括对于当前线程的信息队列的队柄。该区段由 USER.EXE 的视窗系统代码频繁使用，因为在 Windows 95 中，像窗口聚焦之类的事件是以线程为单位存放的。pvTLSArray 区段指向了线程库中的线程局部存储数组。编译商利用其与可执行文件中的.tls 部分共同提供透明的以线程为单位的全局变量。

尽管 TIB 结构的设计可由线程库的结构推断出来，但仅具有结论性。根据 WIN32WLK 源，可以在 TIB.H 中找到有关 C 结构的定义。对于首项所作的正式的 Microsoft 定义是在 Windows NT 3.5 DDK 的 NTDDK.H 文件中的（注意区段必需与 OS/2 2.0 兼容）。这显然是早期 NT 的遗留内容，当时 Microsoft 还正试图更多地考虑 OS/2 方面的问题。（如果对这方面的问题感兴趣，请看 Z. Pascal Zachary 的一书《Showstopper》）

Windows 95 中 TIB 区段具有下述形式：

00h	DWORD	pvExcept
04h	DWORD	TopOfStack
08h	DWORD	StackLow
0Ch	WORD	W16TDB
0Eh	WORD	StackSelector16
10h	DWORD	SelmanList
14h	DWORD	UserPointer
18h	PTIB	pTIB
1Ch	WORD	TIBFlags
1Eh	WORD	Win16MutexCount
20h	DWORD	DebugContext
24h	PDWORD	pCurrentPriority
28h	DWORD	MessageQueue
2Ch	PDWORD	pTLSArray

对于每个区段的描述，将 10h 加到偏置上并查看本章前面“线程库”一节的偏置。注意，这些区段只有一部分适用于所有的 Win32 平台。

线程优先

Windows 95 虚拟设备管理器(VMM)的核心规程并未涉及进程方面的内容，其重点是规划具有最高优先级的线程，而无需考虑它们是处于何种进程。从另一角度来讲，进程并不真正具有优先级。但是，对于这些线程规划服务的最后一个用户而言(即，应用编程员)，认为进程具有优先级是一种很有用的抽象方式。SetPriority 和 GetPriority 函数在进程/线程优先级的两个影象之间充当了翻译的角色。

任何时刻，无需等待的具有最高优先级的线程是那些将要运行的线程。为了保证系统运行顺利并防止问题的发生，系统在运行中改变着线程的优先级。例如，当一个 I/O 操作正等待完成时，某一线程的优先级可能是临时给定的。如果进入线程规程的内部，就会更清楚地了解这一点。因此，我准备在另一本书中(或在有关杂志上)详细介绍线程优先方面的问题。

Windows 95 VMM 规程中，具有 32 个明显的优先级(levels)，这 32 个优先级又分成了 4 个组，称为优先类(classes)。每个优先类与某个特定的优先级相关联，此优先级是该优先类的线程的缺省优先级。在优先类内，线程可以从低于缺省级的两个级别至高于缺省级的两个级别内变化。(有上些特殊情况，如 THREAD_PRIORITY_LEVEL，基中线程的优先可以超出其整个的优先类。)除非有特殊说明，当操作系统生成一个进程时，新进程被给定为 NORMANL_PRIORITY_CLASS。

4 种优先类，其缺省优先值以及其优先值的范围如下：

优先级	缺省值	优先级的范围
IDLE_PRIORITY_CLASS	4	2-6
NORMAL_PRIORITY_CLASS	9 或 7(前台为 9;其它为 7)	6-10
HIGH_PRIORITY_CLASS	13	11-15
REALTIME_PRIORITY_CLASS	24	16-31

优先级为 1 是一种特殊情况。通常具有 IDLE_PRIORITY_CLASS，NORMAL_PRIORITY_CLASS，或 HIGH_PRIORITY_CLASS 的线程可以通过 SetPriorityClass 函数将优先级设为 1。

作为 Windows 95 优先级的一个附加注解，Windows 95 规程中的 32 个级别在数值上与 WINBASE.H 中的优先类的数值并不对应。例如，NORMAL_PRIORITY_CLASS 在 WINBASE.H 中为 0x20，Windows 95 的 KERNEL32.DLL 将这些值映射成了适合于 Windows 95 线程规程的优先值。

GetThreadPriority(获得线程优先级函数)

GetThreadPriority 是一个简单的函数。给定某个线程句柄(可以是任意进程的任意线程句柄)，代码将句柄转换成指向该线程的进程库的指针。假定句柄转换顺利，GetThreadPriority 则返回线程库中 DeltaPriority 区段(偏置 198h)的数值。所有的这类代码是由 EnterSysLevel 和 LeaveSysLevel 所环绕的，以防止诸如切入不适当线程之类的问题发生。

GetThreadPriority 伪码

```
// Parameters:
//      HANDLE  hThread;
// Locals:
//      PTHREAD_DATABASE ptdb;
//      DWORD   retValue;

    x_LogSomeKernelFunction( function number for

    _EnterSysLevel( pKrn32Mutex );

    retValue = 0x7FFFFFFF;

    ptdb = x_ConvertHandleToK32Object( hThread, 0x20, 0 );

    if ( ptdb )
        retValue = ptdb->DeltaPriority;

    _LeaveSysLevel( pKrn32Mutex );
```

SetThreadPriority(设置线程优先级函数)

SetThreadPriority 代码分成 4 个部分。首先,函数将线程句柄变成一个线程库指针。然后,SetThreadPriority 确认所通过的新的优先级,看其是否将输入优先参数转换成由 Windows 95 规程所使用的 32 个线程优先级的其中之一。

最后,SetThreadPriority 调用 VWIN32. VXD,通知新优先级的 ring 0 分量。KERNEL 32 调入 ring0 的机制是由 VxDCall 函数(例如,VxDCall0)提供的。ring3 分量通过 VxDCall 激活 Win32 VxD 服务。此时,VWIN32. VXD 提供一个 ring3 可调用服务来设置线索的优先级。Win32 VxD 服务是 Windows 95 的新内容,它在 ring0-ring3 转换中起关键的作用。事实上,新的 Windows 95 Win32 VxD 服务是相当重要的,第六章将专门讨论。因为 Win 32 VxD 服务在本书后面要全面讨论,这里有关 VxDCall 的机理就不详谈了。

SetThreadPriority 伪码

```
// Parameters:
//      HANDLE  hThread
//      int     nPriority;
// Locals:
//      PTHREAD_DATABASE ptdb;
//      DWORD   retValue;

    x_LogSomeKernelFunction( function number for SetThreadPriority );

    _EnterSysLevel( pKrn32Mutex );

    ptdb = x_ConvertHandleToK32Object( hThread, 0x20, 0 );
    if ( ptdb )
    {

        if (    (nPriority < THREAD_BASE_PRIORITY_MIN)
            &&  (nPriority > THREAD_BASE_PRIORITY_MAX) )
        {
            if (    (nPriority != THREAD_BASE_PRIORITY_LOWRT)
                &&  (nPriority != THREAD_BASE_PRIORITY_IDLE) )
            {
                InternalSetLastError( ERROR_INVALID_PRIORITY );
                goto error;
            }
        }

        ptdb->DeltaPriority = nPriority;

        if ( ptdb->Ring0Thread )
        {
            DWORD newAbsPriority = CalculateNewPriority(ptdb, ptdb->pProcess2);
```

```
            // Call into VWIN32 to do the real work.
            // Set_Thread_Win32_Pri == 0x002A0021
            VxDCall0(Set_Thread_Win32_Pri, ptdb->Ring0Thread, newAbsPriority);
        }

        retValue = TRUE;
    }
    else
    {
error:
        retValue = FALSE;
    }

    _LeaveSysLevel( pKrn32Mutex );

    return retValue;
```

CalculateNewPriority(计算新的优先级函数)

CalculateNewPriority 封装了 Windows 95 规程中有关线程优先的所有规划。给定某进程及线程,该函数可以计算出线程应具有的优先级(1-31 范围之内)。根据进程库,函数为线程(正常,空置,高,实时)抽取了优先类。它将线程的优先增量加到此基础优先类上,优先增量通常为+-2。在将优先类的优先值加到线程优先增量上之后,代码将明确新的优先级应处于期望值之内。应注意,实时优先线程在此是经特殊处理的;因为实时优先级的范围大于其它优先类的范围。

CalculateNewPriority 伪码

```
// Parameters:
//  PTHREAD_DATABASE ptdb;
//  PPROCESS_DATABASE ppdb;
// Locals:
//      DWORD baseProcPri
//      DWORD sum
//      DWORD upperLimit, lowerLimit

    baseProcPri = ppdb->BasePriority;

    if ((baseProcPri != 4) &&
        (baseProcPri != 8) &&
        (baseProcPri != 13) &&
        (baseProcPri != 24))
    {
        x_Assertion2( "..\priority.c" );
    }
```

```
sum = ptdb->DeltaPriority + ppdb->BasePriority;

if ( ppdb->BasePriority == 24 ) // Real time class thread?
{
    upperLimit = 31
    lowerLimit = 16
}
else                            // Other priority class.
{
    upperLimit = 15
    lowerLimit = 1
}

if ( upperLimit >= sum )
    upperLimit = sum

if ( lowerLimit <= upperLimit )
    return upperLimit;
else
    return lowerLimit;
```

SetPriorityClass(设置优先类函数)

SetPriorityClass 函数使调用程序改变了某一进程所有线程的优先类。该函数首先将其 hProcess 参数变成一个 PPROCESS_DATABASE 指针。利用该指针，函数确定出进程的当前优先类。如果其与新的优先类相同，函数则跳出。

如果新优先类与前一级别不同，SetPriorityClass 则将新优先类的缺省值接入进程库的 BasePriority 区段。前面我曾提到，有关进程优先类的注记只不过是一种表达方式而已，因为 VMM 规程本身仅考虑了线程，而与进程无关。为了沟通优先级的两个景象，SetPriorityClass 环入进程中的每个线程并调入 VWIN32. VXD，以便为每个线程设置新的优先级。

有一点要注意，线程可以拥有与缺省类的优先级略有不同的优先级。该差别保留在线程库(后面要谈到)的"DeltaPriority"区段中。当计算线程的新优先值时，SetPriorityClass 需要考虑每个线程的优先增量，此工作由 CalculateNewPriority 函数完成。

SetPriorityClass 伪码

```
// Parameters:
//      HANDLE  hProcess
//      DWORD  fdwPriority
// Locals:
//      BOOL    retValue
//      PPROCESS_DATABASE ppdb;
//      PTHREAD_DATABASE ptdb;
//      DWORD   newPriority
//      PK32OBJECTLISTENTRY pK32Object;
```

```
    x_LogSomeKernelFunction( function number for SetPriorityClass );

    _EnterSysLevel( pKrn32Mutex );

    ppdb = x_ConvertHandleToK32Object( hProcess, 0x10, 0 );

    if ( ppdb )
    {
        retValue = TRUE;
        if ( fdwPriority == NORMAL_PRIORITY_CLASS )
            goto SetNormal;

        if ( fdwPriority == IDLE_PRIORITY_CLASS )
            goto SetIdle;

        if ( fdwPriority == REALTIME_PRIORITY_CLASS )
            goto SetHigh;

        if ( fdwPriority == HIGH_PRIORITY_CLASS )
            goto SetRealTime;

        // None of the allowable priorities was specified, so bomb out.
        retValue = FALSE;
        InternalSetLastError( ERROR_INVALID_PRIORITY );
        goto done;

SetNormal:
        if ( ppdb->BasePriority == 8 )  // No change from previous state?
            goto done;
        ppdb->BasePriority = 8;
        goto SetIt;

SetIdle:
        if ( ppdb->BasePriority == 4 )  // No change from previous state?
            goto done;
        ppdb->BasePriority = 4;
        goto SetIt;

SetHigh:
        if ( ppdb->BasePriority == 13 ) // No change from previous state?
            goto done;
        ppdb->BasePriority = 13;
        goto SetIt;

SetRealTime:
        if ( ppdb->BasePriority == 24 ) // No change from previous state?
            goto done;
        ppdb->BasePriority = 24;

SetIt:
```

```
        // Start looping through all the threads for this process.
        pK32Object = x_GetNextObjectInList( ppdb->ThreadList, 0 );

        while ( pK32Object )
        {
            ptdb = pK32Object->pObject;
            if ( ptdb->Ring0Thread )
            {
                // Calculate the new priority, taking into account the
                // process's base priority and the thread's relative priority.
                newPriority = CalculateNewPriority( ptdb, ppdb );

                // Call into VWIN32 to do the Dirty Deed (Done Dirt Cheap).
                // VxDCall ID == 0x002A0021
                VxDCall0(Set_Thread_Win32_Pri, ptdb->Ring0Thread, newPriority);
            }

            pK32Object = x_GetNextObjectInList( ppdb->ThreadList, 1 );
        }
    }
    else
    {
        retValue = FALSE;
    }

done:

    _LeaveSysLevel( pKrn32Mutex );

    return retValue;
```

GetPriorityClass(获得优先类函数)

GetPriorityClass 函数返回了特定进程的优先类。将 hProcess 参数变成一个 PPROCESS_DATABASE 之后，函数从进程库检索出优先类，其优先级别应处于 1-31 之内，且与 WINBASE. H 中的 xxx_PRIORITY_CLASS #define 不同。因此，GetPriorityClass 将 VMM 规程中的优先级转换成了相应的 xxx_PRIORITY_CLASS 标志。

GetPriorityClass 伪码

```
// Parameters:
//  HANDLE  hProcess
// Locals:
//  DWORD   retValue;

    x_LogSomeKernelFunction( function number for GetPriorityClass );
    retValue = 0;

    _EnterSysLevel( pKrn32Mutex );
```

```
    ppdb = x_ConvertHandleToK32Object( hProcess, 0x10, 0 );

    if ( ppdb )
    {
        if ( ppdb->BasePriority == 4 )
            retValue = IDLE_PRIORITY_CLASS;
        else if ( ppdb->BasePriority == 8 )
            retValue = NORMAL_PRIORITY_CLASS;
        else if ( ppdb->BasePriority == 13 )
            retValue = HIGH_PRIORITY_CLASS;
        else if ( ppdb->BasePriority == 24)
            retValue = REALTIME_PRIORITY_CLASS;
    }

    _LeaveSysLevel( pKrn32Mutex );

    return retValue;
```

线程执行控制

Win32 API 提供了一套小型的 API 用于修改和查询其它线程的执行状态。低等级的线程可以读或写另一个线程的寄存器(假定第一个线程具有一个有效的句柄,用于处理其它线程)。对于更宽级别的线程,Win32 函数可以让你冻结,解冻其它线索的执行状态。下面就来讨论这类线程的控制函数。

GetThreadContext 和 IGetThreadContext

GetThreadContext 可以使一个线程获得另外一个线程的寄存器值的备份。对于任何给定时刻,线程或者是正执行的,或者是被暂停挂起的。当线程处于暂停状态时,其寄存器值保留在了所谓的"线程文本"的数据结构当中。GetThreadContext 可以读某个暂停线程的线程文本结构中的数值。

GetThreadContext 实际上仅仅是一个参数有效层,它检验所通入的指针指向的内存应具有足够的尺寸容纳一个 CONTEXT 结构。如果检测成功,代码则跳入内部的 IGetThreadContext 例程。

IGetThreadContext 是一个盘旋例程。它首先将 hThread 参数转换成一个线程库指针。然后,调用 x_ThreadContext_CopyRegs 将寄存器输入设置拷贝到线程的 ring3 CONTEXT 结构。(x_ThreadContext_CopyRegs 将在下一节讨论。)除此之外,IGetThreadContext 同样调入 VWIN32. VXD 以得到寄存器的 ring0 形式。为什么寄存器的 ring0 及 ring3 形式都存在的原因尚不完全清楚。

在填写完 CONTEXT 输入结构之后,GetThreadContext 验证 CS 和标志寄存器应包括有效的数值。此时,有效是指 CS 寄存器被设置到了用于执行 ring3 代码

的选择器上。标志寄存器的测试很简单，只不过是明确 V86 模块标志应处于关闭状态。

GetThreadContext 伪码

```
// Parameters:
//      HANDLE,     hThread
/       LPCONTEXT  lpContext

    Set up structured exception handling frame

    Touch the first and last bytes that lpContext point to.
    If a fault occurs, it's considered a bad pointer, and the exception
    handler returns FALSE;

    Remove structured exception handling frame

    goto IGetThreadContext;
```

IGetThreadContext 伪码

```
// Parameters:
//      HANDLE      hThread
//      LPCONTEXT  lpContext
// Locals:
//      PTHREAD_DATABASE ptdb;
//      BOOL    retValue
//      DWORD   errCode;
  retValue = TRUE;

  x_CheckNotSysLevel_Win16_Krn32_mutexes();

  x_LogSomeKernelFunction( function number for GetThreadContext );

  _EnterSysLevel( pKrn32Mutex );

  ptdb = x_ConvertHandleToK32Object( hThread, 0x20, 0 );

  if ( !ptdb )
  {
      retValue = FALSE;
  }
  else    // Found a valid process database.
  {
      // Is there a valid ThreadContext field in the thread database?
      if ( ptdb->ThreadContext )
      {
          x_ThreadContext_CopyRegs( lpContext->ContextFlags,
                                    pdtb->ThreadContext, lpContext );
      }
```

```
    else    // ThreadContext is 0 in the thread database.
    {
        if ( ptdb->DebugContext && ptdb->DebugContext.SomeField )
        {
            // Are floating point or debug regs specified?
            if ( lpContext->ContextFlags
                & (CONTEXT_FLOATING_POINT | CONTEXT_DEBUG_REGISTERS) )
            {
                ptdb->DebugContext.ThreadContext.ContextFlags
                    = (CONTEXT_FLOATING_POINT | CONTEXT_DEBUG_REGISTERS);

                // Call VWIN32 to do the copying.
                // _VWIN32_Get_Thread_Context == 0x002A0014
                retValue = VxDCall0( _VWIN32_Get_Thread_Context,
                                     ptdb->Ring0Thread,
                                     &ptdb->DebugContext.ThreadContext );

                if ( retValue == 0 )
                    goto error;
            }

            x_ThreadContext_CopyRegs( lpContext->ContextFlags,
                                      &ptdb->DebugContext.ThreadContext,
                                      lpContext );
        }
        else    // ptdb->DebugContext.SomeField == 0
        {
            if ( lpContext->ContextFlags == 0xFFFFFFFF )
            {
                x_Assertion2( line number, "..\deb.c" );
            }

            // Call VWIN32 to do the copying.  _VWIN32_Get_Thread_Context
            // == 0x002A0014
            retValue = VxDCall0( _VWIN32_Get_Thread_Context,
                                 ptdb->Ring0Thread, lpContext );
        }
    }
}

if ( retValue == FALSE )
    goto error;

if ( lpContext->CONTEXT_CONTROL & 1 )   // Were CONTEXT_CONTROL regs
{                                       // requested?

    // Make sure the right CS is in the context buffer.
    if ( lpContext->SegCs != Ring3_flatCS )
        x_Assertion2( line number, "..\deb.c" );

    // Make sure the VM (V86 mode) flag isn't set in the EFlags field.
    if ( lpContext->EFlags & 0x20000 )
        x_Assertion2( line number, "..\deb.c" );
}
```

```
    errCode = ERROR_SUCCESS;        // = 0

error:
    if ( retValue == FALSE )
        retValue = GetLastError();

    SomeOutputFunction( "GetThreadContext ptdb %08x lpContext %08x eip %08x "
                        "esp %08x ebp %08x erc %d\n",
                        ptdb, lpContext, lpContext->Eip, lpContext->Esp,
                        lpContext->Ebp, errCode );

    _LeaveSysLevel( pKrn32Mutex );

    return retValue;
```

x_ThreadContext_CopyRegs

x_ThreadContext_CopyRegs 例程(我起的名称)将所选择的寄存器从某个源 CONTEXT 记录拷贝到了目标 CONTEXT 的相应区段中去。CONTEXT 记录是为 WINNT.H 的 x86 芯片所定义的，且首区段是一个 DWORD 标志，表明哪一个寄存器应当在 CONTEXT 之内拷贝。

x86 芯片的寄存器分组设立如下：

CONTEXT_DEBUG_REGISTERS	调试寄存器 0-3,6,7
CONTEXT_FLOATING_POINT	数学协处理器状态
CONTEXT_SEGMENTS	DS,ES,FS,GS
CONTEXT_INTEGER	EAX,EBX,ECX,EDX,ESI,EDI
CONTEXT_CONTROL	SS:ESP,CS:EIP,EFLAS,EBP

x_ThreadContext_CopyRegs 函数是单向向前的，它严格地检查了源文本中每一个标志，而且在设置时会将相应的寄存器值拷贝到目标文本中去。该代码并无特别之处。

x_ThreadContext_CopyRegs 伪码

```
// Parameters:
//      DWORD       flags
//      PCONTEXT    pSrcCtx;
//      PCONTEXT    pDestCtx;

    if ( flags & 0x00000010 )       // CONTEXT_DEBUG_REGISTERS
    {
        // Copy the debug registers over.
```

```
        pDestCtx->Dr0 = pSrcCtx->Dr0;
        pDestCtx->Dr1 = pSrcCtx->Dr1;
        pDestCtx->Dr2 = pSrcCtx->Dr2;
        pDestCtx->Dr3 = pSrcCtx->Dr3;
        pDestCtx->Dr6 = pSrcCtx->Dr6;
        pDestCtx->Dr7 = pSrcCtx->Dr7;
    }

    if ( flags & 0x00000008 )       // CONTEXT_FLOATING_POINT
    {
        // Copy the FLOATING_SAVE_AREA. See WINNT.H for the
        // layout of a FLOATING_SAVE_AREA struct.
        memcpy( &pDestCtx->FloatSave, &pSrcCtx->FloatSave,
                sizeof(FLOATING_SAVE_AREA) );
    }

    if ( flags & 0x00000004 )       // CONTEXT_SEGMENTS
    {
        pDestCtx->SegGs = pSrcCtx->SegGs;   // Copy the non-control-related
        pDestCtx->SegFs = pSrcCtx->SegFs;   // segments over.
        pDestCtx->SegEs = pSrcCtx->SegEs;
        pDestCtx->SegDs = pSrcCtx->SegDs;
    }

    if ( flags & 0x00000002 )       // CONTEXT_INTEGER
    {
        pDestCtx->Edi = pSrcCtx->Edi;
        pDestCtx->Esi = pSrcCtx->Esi;
        pDestCtx->Edx = pSrcCtx->Edx;
        pDestCtx->Ecx = pSrcCtx->Ecx;
        pDestCtx->Ebx = pSrcCtx->Ebx;
        pDestCtx->Eax = pSrcCtx->Eax;
    }

    if ( flags & 0x00000001 )       // CONTEXT_CONTROL
    {
        pDestCtx->Ebp = pSrcCtx->Ebp;
        pDestCtx->Eip = pSrcCtx->Eip;
        pDestCtx->SegCs = pSrcCtx->SegCs;
        pDestCtx->EFlags = pSrcCtx->EFlags;
        pDestCtx->Esp = pSrcCtx->Esp;
        pDestCtx->SegSs = pSrcCtx->SegSs;
    }
```

SetThreadContext 和 ISetThreadContext

SetThreadContext 可以使一个线程改变另一个线程的寄存器的值，对 GetThreadContext 函数起到了补充作用。取决于你将哪一个标志置入了输入文本结构，SetThreadContext 将某区段拷贝到了 Windows 95 用于保留暂停线程的寄存器的真正线程文本中。

真正的 SetThreadContext 是一个参数有效层，它验证通过的指针所指向的内

存应具有足够的尺寸容纳一个 CONTEXT 结构。如果成功，代码则跳至内部的 ISetThreadContext 例程。

ISetThreadContext（与 IGetThreadContext 类似）是盘旋式的。它首先将输入的 hProcess 转换成一个线程库指针。然后，根据外部条件，它生成对 x_ThreadContext_CopyRegs 函数的调用（前面已经介绍）。同时，生成了对 VWIN32.VXD 的调用。这里的真正原因也不完全清楚。

SetThreadContext 伪码

```
// Parameters:
//      HANDLE      hThread
//      LPCONTEXT   lpContext

    Set up structured exception handling frame

    Touch the first and last bytes that lpContext point to.
    If a fault occurs, it's considered a bad pointer, and the exception
    handler returns FALSE;

    Remove structured exception handling frame

    goto ISetThreadContext;
```

ISetThreadContext 伪码

```
// Parameters:
//      HANDLE      hThread
//      LPCONTEXT   lpContext
// Locals:
//      PTHREAD_DATABASE ptdb;
//      BOOL    retValue
//      DWORD   errCode;
//      PCONTEXT pDestCtx;

    retValue = TRUE;

    x_CheckNotSysLevel_Win16_Krn32_mutexes();

    x_LogSomeKernelFunction( function number for GetThreadContext );

    _EnterSysLevel( pKrn32Mutex );

    ptdb = x_ConvertHandleToK32Object( hThread, 0x20, 0 );

    if ( !ptdb )
    {
        retValue = FALSE;
    }
```

```
    else    // Found a valid thread database.
    {
        if ( ptdb->Flags & 0x20000000 )
        {
            retValue = 0;
            InternalSetLastError( ERROR_INVALID_PARAMETER );
            goto doneCopying;
        }

        if ( ptdb->ThreadContext )
        {
            x_ThreadContext_CopyRegs( lpContext->ContextFlags,
                                      lpContext, pdtb->ThreadContext );
        }
        else
        {
            if ( ptdb->DebugContext && ptdb->DebugContext.SomeField )
            {
                x_ThreadContext_CopyRegs(lpContext->ContextFlags,
                                         lpContext,
                                         &ptdb->DebugContext.ThreadContext);

                if ( !(ptdb->DebugContext.ThreadContext.ContextFlags
                     & (CONTEXT_FLOATING_POINT | CONTEXT_DEBUG_REGISTERS)) )
                    goto doneCopying;

                pDestCtx = &ptdb->DebugContext.ThreadContext
            }
            else
            {
                pDestCtx = lpContext;

                if ( lpContext->ContextFlags == 0xFFFFFFFF )
                    x_Assertion2( line number, "..\deb.c" );
            }

            // Call VWIN32 to do the copying.  _VWIN32_Set_Thread_Context
            // == 0x002A0015
            retValue = VxDCall0( _VWIN32_Set_Thread_Context,
                                 ptdb->Ring0Thread, pDestCtx );
        }
    }

doneCopying:

    errCode = ERROR_SUCCESS;        // = 0
      if ( retValue == FALSE )
          errCode = GetLastError();

      SomeOutputFunction( "SetThreadContext ptdb %08x lpContext %08x eip %08x "
                          "esp %08x ebp %08x erc %d\n",
                          ptdb, lpContext, lpContext->Eip, lpContext->Esp,
                          lpContext->Ebp, errCode );

      _LeaveSysLevel( pKrn32Mutex );

      return retValue;
```

SuspendThread 和 VWIN32 SuspendThread

SuspendThread 判断特定线程的暂停计数。如果暂停计数不为 0，则 ring0 规程不允许该线程执行。SuspendThread 代码其实是所谓的 VWIN32 SuspendThread 内部例程的外壳。VWIN32 SuspendThread 函数期待一个线程库指针。因而在调用它之前，SuspendThread 首先将 hTread 转换成一个线程库指针。如果 VWIN32 SuspendThread 成功，SuspendThread 同样递增线程库偏置 1BCh 中的暂停计数。

然而，更重要的是要记住当决定哪一个线程是可以运行的时候，线程的暂停计数并未使用，VWIN32 而是将真正的暂停计数保留在了 TDBX 结构中。如第六章所述，VWIN32 SuspendThread 例程本身是更低一级例程的外壳。此时，VWIN32 SuspendThread 建立起事务并根据非资源的 VxDCall 函数调入 VWIN32. VXD。通入此事例的 Win32 VxD 服务的 ID 是 0x002A001A

SuspendThread 伪码

```
// Parameters:
//      HANDLE  hThread
// Locals:
//      PTHREAD_DATABASE ptdb;
//      DWORD   retValue

    retValue = 0xFFFFFFFF;
    _EnterSysLevel( pKrn32Mutex );

    ptdb = x_ConvertHandleToK32Object( hThread, 0x20, 0 );

    if ( ptdb )
    {
        if ( !ptdb->Flags & fCreateSuspended )
        {
            retValue = VWIN32_SuspendThread( ptdb );

            if ( retValue != 0xFFFFFFFF )
                ptdb->APISuspendCount++;
        }
    }

done:
    _LeaveSysLevel( pKrn32Mutex );

    return retValue;
```

VWIN32_SuspendThread 伪码

```
// Parameters:
//      PTHREAD_DATABASE ptdb;
// Locals:
//      DWORD   retValue

    retValue = 0xFFFFFFFF;

    // Make sure _EnterSysLevel was called earlier.
    _ConfirmSysLevel( pKrn32Mutex );

    if ( !(ptdb->Flags & 0x10000000) && ptdb->Ring0Thread )
    {
        // Call VWIN32 to suspend it. "SuspendThread" == 0x002A001A.
        retValue = VxDCall ( SuspendThread, ptdb );
    }

    if ( retValue = 0xFFFFFFFF )
        SetLastError( ERROR_NO_MORE_ITEMS );    // ???

    return retValue;
```

ResumeThread

ResumeThread 是 SuspendThread 函数补充，它递减了某个特定线程的暂停计数(包括线程的 TDBX 结构中的 ring0 暂停计数和线程库中的 1BCh 偏置)。当暂停计数为 0 时，ring0 规程将认为线程符合执行条件。

ResumeThread 首先将通入的 hTread 转换成一个线程库指针。然后，检查暂停计数的 ring3 形式，确定其为非 0。(如果为 0，ResumeThread 则毫无用途。)接下来，函数调用另一个 VWIN32. VXD 服务来递减暂停计数。在该事例中，Win32 服务 ID 是 0x002A001B(大于用于生成暂停计数的 0x002A001A)。如果 VWIN32 可以成功志递减暂停计数的 ring0 形式，ResumeThread 同样也可以存于线程库的 ring3 形式。

ResumeThread 伪码

```
// Parameters:
//      HANDLE  hThread
// Locals:
//      PTHREAD_DATABASE ptdb;
//      DWORD   retValue

    retValue = 0xFFFFFFFF;

    _EnterSysLevel( pKrn32Mutex );
```

```
ptdb = x_ConvertHandleToK32Object( hThread, 0x20, 0 );

if ( ptdb && ptdb->APISuspendCount )
{
    if ( ptdb->Flags & fCreateSuspended )
    {
        ptdb->APISuspendCount--;
    }
    else
    {
        // Call VWIN32 to wake up the thread.  VWIN32_ResumeThread
        // is identical to VWIN32_SuspendThread, except that it
        // calls VWIN32 service 0x002A001B instead of 0x002A001A.
        retValue = VWIN32_ResumeThread( ptdb );

         if ( retValue != 0xFFFFFFFF )
             ptdb->APISuspendCount--;
     }
 }

 _LeaveSysLevel( pKrn32Mutex );

 return retValue;
```

结构异常处理

结构异常处理(通常称 SEH),在诸如 OS/2,Windows NT,和 Windows 95 之类的现代操作系统中是一个极易引起误解的内容。绝大多数的有关 SEH 的书籍和文章是以编译程序级来讨论的。编译程序使用了如__try,__except,catch,throw,之类的关键词,将一个相当好的外壳置于一个相当混乱的操作系统的周围。到目前为至,我尚未看到有关 SEH 是如何在操作系统级上实现的好的介绍。因此我在此准备认真讨论一下这个问题。

关于如何在应用代码中使用 SEH 方面的知识已有多方面的介绍,这里我不做进一步讨论。第二章简单介绍了如何在 C/C++代码中使用__try 和__except。除此之外,我们认定你已经熟悉 SEH 方面的基本知识。否则,建议你读一下 Brian Meyer 的《Mastering Windows NT Programming》一书或 Jeffery Richer 的《Advanced Windows》一书。

无论何时出现异常(如缺页中断),CPU 都会立刻将控制切换到 ring0 异常处理程序,其地址储存在中断描述符表内。然后,此 ring0 处理程序来决定对该异常应采用什么样的动作。如果此异常是系统所预期的并知道如何处理的(例如,当需要更多的栈空间时出现的缺页中断),ring0 处理程序将在任何必要时工作并重新开始指令。实际上,这些异常 ring3 应用程序和 DLL 系统是看不到的,这里不作介绍。

我们在此所关心的是，当第一线系统异常处理程序所不知道的异常出现时，将会发生什么情况。老式的操作系统(如 Windows 3.0)的典型响应是中断造成异常的进程。这就是当一个故障应用程序造成系统在 Windows 3.0 中出现 UAE 对话框或在 Windows 3.1 中出现 GPF 对话框时所看到的内容。

当采用所谓的“中断造成任何异常失效事务”的观点时，并不是十分灵活。更好的办法是通知应用程序(或其它应用程序)失效的发生并让其决定应采取什么措施。如果你用过 16 位 TOOLHELP InterruptRegister 函数，你可能有了这方面的体会。(但每个安装进程只有一个处理程序。)当选择的异常组或中断组其中之一出现时，TOOLHELP 回调已安装的 Callback 函数。并利用其返回值确定应如何处置异常。例如，像 WinSpector 或 Dr. Watson 利用异常 callback 函数记入有关异常处理时设备状态的信息之类的程序。之后，他们通知 TOOLHELP 将异常传入下一个处理程序。假定没有 TOOLHELP callback 函数重新起动失效指令，缺省操作系统异常处理程序则被激活。此缺省处理程序中止了失效任务。

尽管 TOOLHELP callback 设计向前迈了一大步(与无任何控制的状况相比较)，但仍不太完善。当诸如 OS/2，NT 之类的 32 位 PC 操作系统面世时，他们引入了更灵活的处理异常的方法。这种新型的设计就是所谓的“结构异常处理”，该方法在处理多线程及满足 C++语言的 catch 和 throw 异常机制的要求上具有更加成熟的方法。在 C++异常中，代码本身可以生成一个与 CPU 异常无关的异常。例如，如果 C++的 new 操作符失败，它将抛出一个带有指示内存溢出代码的异常。32 位操作系统的结构异常处理是相当灵活的。利用同一代码，既可以处理硬件异常，又可以满足处理语言“异常”的需要。

在具体讨论之前，我想重新强调一下这里所介绍的结构异常处理是如何在 OS 级上实现的。下面的内容听似与你在 C/C++中所了解的内容可能“完全不同”。

对于具有结构异常处理的系统，每个线程拥有其各自的专用异常处理程序链接列表。当异常出现时，操作系统行至已安装的异常处理程序列表并调用处理程序。直到处理函数返回一个指示处理异常的代码，此进程方停止。这是第一步(寻找想处理异常的处理程序)，如果不存在想要处理异常的用户安装处理程序，位于此链末端的系统处理程序将中止此进程。这种情况很简单，不做进一步介绍。

一旦某个程序决定处理异常，下一步就是从头部开始重新行至异常处理程序列表。非资源的 RtlUnwind 函数完成了此项工作，而且被决定要处理异常的异常处理程序所调用。当异常处理程序由 RtlUnwind 激活时，系统将一个不同的标志通过此处理程序函数。该标志通知处理程序函数线程的栈正被展开。展开栈可以控制存放异常处理程序安装时的程序状态。系统为每个安装了的并不处理函数的处理程序提供了一个净化的机会，而不是仅仅在__except 块中再执行。由于给定了这样的机会，像为以 C++目标为基础的栈来调用解除程序之类重要的工作就可以按照有序的方式完成了。

有关上述讨论至此为止。那么，Windows 95 使用的真正的结构和接口到底是什么呢？正如我在前面 TIB 中所介绍的那样，FS:[0]处的指针总是指向为当前线程安装的异常处理程序列表的首部的。异常处理程序列表是一个链接的 EXCEP-

TIONREGISTERATIONRECORD(异常寄存器记录)结构的列表。(此长名称来自于 OS/2 2.0 BSEXCPT.H 文件。出于某种考虑,Microsoft 似乎掩示了 OS 级结构异常处理的细节。)EXCEPTIONREGISTERATIONRECORD 的设计如下:

```
DWORD prev_structure    // A pointer to the previously installed
                        // EXCEPTIONREGISTRATIONRECORD.
DWORD ExceptionHandler  // Address of the exception handler function.
```

异常处理程序列表的末端由 prev-structure 区段中的-1 来指明。

在通常使用中,程序代码在栈上生成所需的每一个 EXCEPTIONREGISTERATIONRECORD 结构。在 C/C++ 代码中,每个 EXCEPTIONREGISTERATIONRECORD 一个__try/__except 顺序。当__try 块加入时,编译程序在栈上建立一个新的 EXCEPTIONREGISTERATIONRECORD 并将其置于链的首部。在此块存在之后,编译程序将 FS:[0]指向链中的下一个 EXCEPTIONREGISTERATIONRECORD。图 3-3 是这类链接记录:

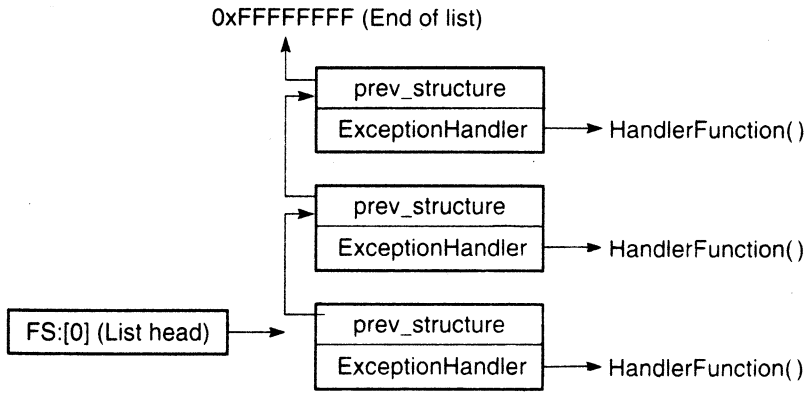


图 3-3　操作系统级的结构异常处理链

要记住上面给出的 8 字节结构仅是操作系统的最小需要。编译程序可以在栈上生成更大的结构并将 EXCEPTIONREGISTERATIONRECORD 置于结构的起点。编译程序从该结构得到的附加区段提供了足够的文本信息,可以使一个单独的异常处理程序函数用于所有的__try 块。你将看到 Microsoft 和 Borland 的编译程序都使用了具有所需的 EXCEPTIONREGISTERATIONRECORD 结构的异常处理结构。

至于异常处理程序函数,又是怎么一回事呢? Microsoft 似乎又一次掩盖这方面的真实信息,但至少我们在 Win32 的头文件中可以看出异常处理程序的样子。在 EXCPT.H 文件中,你会看到:

```
EXCEPTION_DISPOSITION __cdecl _except_handler (
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext
    );
```

当你第一次看到时，其形式好象很复杂。EXCEPTION_DISPOSITION 返回的数值仅是一个枚举数，它告知系统处理程序函数如何选择处理异常：

```
typedef enum _EXCEPTION_DISPOSITION {
    ExceptionContinueExecution,
    ExceptionContinueSearch,
    ExceptionNestedException,
    ExceptionCollidedUnwind
} EXCEPTION_DISPOSITION;
```

最后两个枚举数很少碰到。ExceptionContinueExecution 返回代码通知系统异常处理程序函数已经处理了异常并想让执行继续。ExceptionContinueSearch 函数返回的代码则通知系统处理程序函数不想处理异常，而且系统应继续行至 EXCEPTIONREGISTERATIONRECORD 列表，以寻找一个可返回 ExceptionContinueExecution 的处理程序。

重写_except_handler 的形式使其更便于访问，可以得到：

```
int _except_handler (
            PEXCEPTION_RECORD ExceptionRecord,
            PVOID EstablisherFrame,
            PCONTEXT ContextRecord,
            PVOID DispatcherContext );
```

译成英文，其意思是某个异常处理程序函数采用了四个指向有关异常信息和异常出现时设备状态信息的指针。函数返回了一个整数来通知系统该处理程序是否处理了异常。EXCEPTIONRECORD 结构包括异常代码(处于其它信息中间)，并作为 WINNT.H 文件中的资源。CONTEXT 结构包括了异常出现时所设置的寄存器，也在 WINNT.H 中进行了描述。EstablisherFrame 参数包括一个指向联合 EXCEPTIONREGISTERATIONRECORD 所建立的栈的指针，而 DispatcherContext 参数在此似乎未被使用。

早些时候，我曾提到处理程序函数在一个典型的异常现场被调用了二次。第一次调用是系统用于寻找能处理异常的处理程序。第二次调用则是系统要从该异常展开，而且异常程序被认定可完成任何必要的清除工作(如为以栈为基础的目标来激活解除程序)。那么，异常处理程序在这二种事务中是如何区分的？ExceptionRecord 结构(由第一参数所指)包括了一个 ExceptionFlags 区段。如果 EH_UNWINDING(0x2)或 EH_EXIT_UNWIND(0x4)标志均是确切的，则异常处理程序将被激活来检查处理程序是否想处理此异常。如果此位之一已被设置，处理程序

将正被激活来由此异常展开。

尽管我们刚才所述尚不足以使你写出自己的 OS 级异常处理代码，但足够使我们对 SEH 的工作方式有了一个大体的了解。为此我编写了 SHOWSEH 程序作为验证，此程序在附盘上。SHOWSEH 利用了__try 块来建立一个具有多入口的异常处理链。一旦建立，该程序将行至 SEH 列表并输出有关每个结点的信息。

图 3-4 是运行 SHOWSEH 的输出，这里有几点要加以说明。首先，要注意处于下一个记录列的地址总是递增的。这些地址处于 SHOWSEHss 的栈区之中，它们同编译程序一道将每个 EXCEPTIONREGISTERATIONRECORD 置于栈上。最初的四个入口是根据 SHOWSEH.C 代码中的__try 块直接生成的。其次，要注意处于下一个记录列的头四个地址是如何通过一个常值与各种 SHOWSEH 函数中的 ESP 值相关联的。(输出的最后显示了处于每个 SHOWSEH.C 中的函数的内部 ESP 值。)

```
offset of __except_handler3: 00401468

next rec  handler
========  ========
0063FD90  00401468
0063FDC0  00401468
0063FDF0  00401468
0063FE30  00401468
0063FF68  00401468
FFFFFFFF  BFFC2D18

in    c(), ESP = 0063FD48
in    b(), ESP = 0063FD78
in    a(), ESP = 0063FDA8
in main(), ESP = 0063FDD8
```

图 3-4　SHOWSEH.EXE 输出

最后一点，要注意处理程序列中的头 5 个地址是相同的。该地址处于 SHOWSEH 代码区之内，表明编译程序生成的代码对所有的__try 块均使用了相同的异常处理程序函数。而且，输出中的头 4 个入口来自于 SHOWSEH.C 中的__try 块。第 5 个异常处理程序(但要具有相同的处理程序地址)是由运行库代码在调用 main()函数之前所安装的。这些处理程序地址对于__except_handler3 而言，均可视为 C++中的一个运行库函数。最后的异常处理程序是由系统缺省的，安装于 KERNEL32.DLL 中。

结构异常处理及参数有效化

Windows 95 中结构异常处理的最重要用途之一是作为一种快速，便捷的方法将进入 API 函数的参数有效化。其基本思想是认为参数是正确的，然后对其运行一系列的明确的检测。但在进行此确切检测之前，代码首先将一个新结构异常处

理帧加至异常处理程序列表的首部。如果参数是有效的，在此的确切检测应不出现任何意外，而且异常处理程序被顺便移走。此执行进程只需很短时间。

如果在确切检测代码的执行中参数证明为不正确，则生成一个 CPU 异常，它是由新安装的异常处理程序所处理的。异常处理程序通知操作系统在造成 API 函数返回失败值(如 FALSE)的地方恢复正常的线程执行。让我们来看一看 KERNEL32 API 的一些伪码此时的情况。在本书的其它 API 中，我介绍了 SEH 在参数有效化中所起的作用。对于 GetCurrentDirectory 函数，我给出了详细的伪码，用于表明操作系统对 SEH 的使用情况。

除了表明 SEH 概念之外，GetCurrentDirectoryA 伪码还详细表示了一个典型的参数有效层存根是怎样一回事。具有参数有效层存根的函数所拥有的代码分为二部分。位于输出函数的地址的代码仅是用于测试参数的有效性的一小段代码。如果参数正确，存根代码则跳到位于模块其它地址的实码之上。在 Windows 3.1 中，某个具有有效参数的函数的代码的真正形式与带有 I 前置形式的输出函数的名称是一致的。例如，本章早些时候所介绍的，GetProcAddress 代码分成了两部分：

```
GetProcAddress stub
    Validate the procedure name string parameter
    JMP IGetProcAddress

IGetProcAddress
    Meat of the code that looks up a function address
```

为了保持连贯性，我沿用了 Windows 3.1 的方式以 I 来预设某个函数代码的内部形式。

GetCurrentDirectoryA

GetCurrentDirectoryA 是一个典型的参数有效层存根。它首先在栈上生成了一个新的 EXCEPTIONREGISTERATIONRECORD(利用了 PUSH 指令)。该结构的 prev_structure 部分通过推入 FS:[0](异常链的当前首部)而被置于栈上。置入 EXCEPTIONREGISTERATIONRECORD 的异常处理程序函数是一个由我命名的例程，x_invalid_param_2_params。具有两个参数和具有参数有效存根的所有函数均使用了 x_invalid_param_2_params，作为其升效阶段的异常处理程序。

将EXCEPTIONREGISTERATIONRECORD置于栈上之后，代码将一个指向该结构的指针移入FS:[0]，这样就将新的EXCEPTIONREGISTERATIONRECORD置于了异常处理链的首部(使其成为异常发生时的第一个被调用项)。通过这种函数的异常处理程序，代码就可以安全地接触和考察所通过的 lpsCurDir 指针而无需考虑其有效与否。此时所谓参数的有效化是指检验由 lpsCurDir 和参数长度描述的整个内存块是可写的。

如果检验当中出现了异常，x_invalid_param_2_params 函数将获得控制。(x_invalid_param_2_params 函数将在下一节介绍。)如果参数合适，执行将按预期路

径继续。GetCurrentDirectory 代码的倒数第二位移走了异常处理程序帧。一个 POP FS:[0]存放了指向异常处理程序列表首部的各种指针，而一个 ADD ESP,4 则移走了异常处理程序地址。

假定一切进行顺利(即，无任何异常出现)，GetCurrentDirectory 的最后一项事务是跳至 IGetCurrentDirectory 代码。你可能感到有点奇怪，KERNEL32 表面上并未使用存放在环境库中的当前目录指针，而是指向了 Win16 任务库并利用 VWIN32 的 Int 21h 调度机制来调用 INT 21h，函数 19h(GetCurrentDirectory)，和函数 7147h(GetCurrentDirectory 的长文件名形式)。

GetCurrentDirectory 伪码

```
// Parameters:
//      DWORD  cchCurDir
//      LPTSTR  lpszCurDir

    // Set up structured exception handling frame. We do this by creating
    // the exception record on the stack. An exception record looks like this:
    //
    // prev_structure;      // Pointer to previous record.
    // ExceptionHandler     // Address to call on an exception.
    //

    push    offset x_invalid_param_2_params     // Offset of handler.
    push    FS:[0]                              // Head of list in FS:0.
    mov     FS:[0], ESP                         // Point FS:0 at record
                                                // we just built.

    if ( lpszCurDir && cchCurDir )  // If both params are non-null...
    {
        LSPTR lpszEndPtr = lpszCurrDir + cchCurDir + 1;
        LPSTR lpszTemp;

        *lpszEndPtr += 0;       // Harmlessly write the last byte in the
                                // buffer. If a fault occurs, the exception
                                // handler will be invoked.

        if ( lpszEndPtr != lpszCurDir )
        {
            lpszTemp = lpszCurDir;

            // Go through each page between the start and end of the buffer
            // and touch it. If a fault occurs, the exception handler will
            // be invoked.
            while ( lpszTemp < lpszEndPtr )
            {
                  *lpszTemp +=0;          // Harmlessly write to the page.
                  lpszTemp += 0x1000;     // Advance pointer to next page.
              }
        }
    }
```

```
// If we got here, everything went well. Clear off the exception
// record from the stack and restore the previous head pointer.

pop     FS:[0]      // Put the previous head of the list into FS:0.
add     esp,4       // Throw away the exception handler address we pushed.

goto    IGetCurrentDirectoryA
```

x_invalid_param_handler

x_invalid_param_handler 函数处于典型失效参数异常结束的位置上，它并不是被直接调用的。KERNEL32 拥有 10 个存根系列，其中每个存根将一个不同的参数字节计数通至了 x_invalid_param_handler。拥有 10 个不同存根的原因是 x_invalid_param_handler 函数需要从栈将所有的形参移至具有失效参数的函数上。

在 x_invalid_param_handler 内部，代码起着三方面的作用。首先，函数输出了“所通过的无效参数:XXXXXXXXX”信息，它显示于调试终端上。其次，函数调用 RtlUnwind 来清除任何由参数有效存根安装的异常处理程序之后所安装的异常处理程序。第三，也是最重要的一步，函数调用我已命名的 ReturnFailureCode 函数。ReturnFailureCode 为原先的函数(如 GetCurrentDirectory)计算了正确的失败返回值，然后跳至原先函数的退出端口。

如果对上述进程产生曲解，其结果是向某个 Win32 函数通入一个无效参数的程序将会直接认为该函数失败了。程序对于内部所运行的 SEH 是一无所知的。另外，如果你正运行调试版的 Windows 95，会出现一个无效参数诊断。

x_invalid_parm_X_param 伪码

```
x_invalid_param_1_param proc
    x_invalid_param_handler( 0x04 );

x_invalid_param_2_params proc
// parameters:
//  struct _EXCEPTION_RECORD *ExceptionRecord,
//  void *EstablisherFrame,
//  struct _CONTEXT *ContextRecord,
//  void *DispatcherContext

    x_invalid_param_handler( 0x08 );

x_invalid_param_3_params proc
    x_invalid_param_handler( 0x0C );

x_invalid_param_4_params proc
    x_invalid_param_handler( 0x10 );
```

```
x_invalid_param_5_params proc
    x_invalid_param_handler( 0x14 );

x_invalid_param_6_params proc
    x_invalid_param_handler( 0x18 );

x_invalid_param_7_params proc
    x_invalid_param_handler( 0x1C );

x_invalid_param_8_params proc
    x_invalid_param_handler( 0x20 );

x_invalid_param_9_params proc
    x_invalid_param_handler( 0x24 );

x_invalid_param_special proc
    x_invalid_param_handler( 0x80000000 );
```

x_invalid_param_handler 伪码

```
// Parameters:
//  DWORD   cbParams
//  DWORD   caller_retAddr
//  struct _EXCEPTION_RECORD *ExceptionRecord,
//  void *EstablisherFrame,
//  struct _CONTEXT *ContextRecord,
//  void *DispatcherContext
// Locals:
//      DWORD   faultEBX
//      DWORD   faultEBP
//      DWORD   faultESI
//      DWORD   faultEDI
//      DWORD   fSomeFlag

    // If the unwinding flags aren't set (TRUE the first time through),
    // then handle the exception...
    if ( 0 == (pExcRec->ExceptionFlags & (EH_UNWINDING | EH_EXIT_UNWIND) ))
    {
        if ( cbParams == 0 )
        {
            fSomeFlag = -1
            if ( !(pEstablisherFrame->8 &0x100) )
                fSomeFlag = 0;
        }
        else
            fSomeFlag = 0;

        dprintf( "Invalid parameter passed to:\n" );

        // Send the EIP out via the debugger INT 41h interface.
        x_INT41_DS_printf( "%pLNS", pContext->Eip );
        dprintf( " (%04x:%08x)\n", pContext->SegCS, pContext->EIP );
```

```
        push    EBX, ESI, EDI   // Preserve across the RTLUnwind call.

        RtlUnwind( pEstablisherFrame, FFC00BAD, 0, 0 );

        pop     EDI, ESI, EBX

        SetLastEror( ERROR_INVALID_PARAMETER );

        // Restores EBX, ESI, EDI, ESP, and returns to original code.
        return ReturnFailureCode(   cbParams,
                                    fSomeFlag,
                                    pEstablisher,
                                    faultEBX
                                    faultESI
                                    faultEDI
                                    faultEBP );
    }

    return XCPT_CONTINUE_SEARCH;
```

线程局部存储器(TLS)

线程局部存储器(TLS)是 Win32 一个很好的特征,它使得多线程编程更加简便。利用线程局部存储器,一个程序可以拥有以线程为单元的全局变量。即,某个进程中的所有线程可以拥有似在全局内起作用的数据,但实际上只限于特定的线程。例如,你可能有一个多线程程序,其中每个线程只对应写入一个单独文件(因而要使用单独文件柄)。此时,储存在 TLS 中每个线程均使用的文件柄是一件很方便的事。当某个线程的代码需要知道所要使用的句柄时,它只需从 TLS 中检索出数值即可。有一点要注意,文件写入线程用来检索文件柄代码的代码在任何时候都是确切的。但是,由 TLS 机构所返回的文件柄对于每个线程又是不同的。这也就是说,变量是供全局使用的,但以每个线程为单元。

当然,你可以利用一个联系线程 ID 与文件柄的链接列表,以及单结点线程单元模拟此时的 TLS。当一个线程需要知道要利用哪一个文件柄去写入时,它应可以在此链接列表中查找文件柄。返过来,你也可以在线程栈的真实局部变量中存放每个线程的文件柄。但这样你就不得不从一个函数到另一个函数,一次又一次地绕过文件柄。这样做当然费事,TLS 利用一个简单的 alloc/set/get/free API 就解决了这一难题。

尽管 TLS 很方便,但也有其使用范围。在 Windows NT 和 Windows 95 中,对于每个线程,有 64 个 DWORD 槽可以使用。这就是说,一个进程可以以线程为单元存放 64 个单独的 DWORD。为了保留每个线程的一个槽,需要有一个调用 TlsAlloc 的程序。每个对 TlsAlloc 的调用返回了由每个线程使用的单一索引值。此索引通常存放在一个真实的全局变量之内。当一个线程想将某个值存放于已配置槽的其中之一时,它将调用 TlsetValue,通入 TLS 索引值以及要存入槽的数值。其后,当线程需要使用已放弃了的线程单元值时,代码要再一次调用 TlsGetValue 并通

入特定某特殊 TLS 槽的索引。最后，当程序利用某个 TLS 槽完成时，将 TLS 索引通至 TlsFree。这样做，当然使得此槽未能被所有线程利用，因为在某个进程中一个 TLS 线程是由所有线程使用的。

尽管 TLS 肯定可以存储诸如文件柄之类的单一数值，其更经常的用途是存储指向线程单元数据的指针。在许多情况下，一个线程的程序需要储存一组变量，而且每个是以线程为单元的。此时，许多程序员所要做的是将线程单元的变量置入某个 C 型结构，然后在 TLS 槽中保留一个指向该结构的指针。当新线程生成时，程序为该结构 mallocs 一部分内存并将指针储存于为其所保留的 TLS 槽中。当线程中断时，代码释放出已配置的块。

第十章的 APISPY32 程序较好地展示了上述的编程机理。APISPY32. DLL 需要从生成中断的函数保留一块返回地址的栈。（此时，我使用栈这一术语以便与经典计算机科学相符；也就是说，我利用它来表示某个结构数组和某个栈指针。）由于被监测的程序可能拥有多个线程，APISPY32. DLL 需要为每个线程保持单独的返回地址栈。如果某个进程中的每个线程拥有 64 个槽用于储存每个线程数据，那么这些槽是从何处而来的呢？正如本章前面所述，每个线程库包括一个 64DWORD 数组，供 TLS 函数使用。当你利用一个 TLS 函数设置或检索某个值时，实际上你正读或写入当前线程的线程库。要注意的是，TlsGetValue 和 TlsSetValue 函数隐含地操作了当前线程的 TLS 数据。没有资源化的办法可让一个线程访问另一个线程的 TLS 数据。现在让我们进一步考察一下 TLS 函数的执行情况。

TlsAlloc

由于 TLS 可为每个线程提供 64 个槽，因而有必要寻找某种办法，以跟踪哪一个 TLS 槽正被使用。KERNEL32 利用两个 DWORD（总共为 64 位）来达到此目的。这两个 DWORDS 可以看成是一个 64 位数组。如果给定位处于开状态，则与此位相关的 TLS 索引处于使用状态。

64 位 TLS 槽数组储存在进程库中（而不是线程库）。记住，当你分配某个 TLS 槽时，该槽在进程的所有线程中利用槽的索引而被访问。64 位 TLS 槽数组处于进程库的偏置 0x88 和 0x8C 的 DWORD 中。尽管下述给出的 TlsAlloc 伪码看似复杂，但实际上并非如此。此代码所进行的所有工作只不过是查看 64 位数组中的位，并找出值为 0 的位。当 TlsAlloc 找到位时，它将此位打开并返回位在数组中的值。因此，如果 64 位数组中第 5 个位是 0，TlsAlloc 则将此位打开并返回索引为 4 的 TLS 值（TLS 索引是以 0 基准的）。

TlsAlloc 伪码

```
// Locals:
//      DWORD   i;
//      PDWORD  pTlsInUseBits;
//      DWORD   newFlag;
```

```
x_LogSomeKernelFunction( function number for TlsAlloc );

i = 0;

_EnterSysLevel( x_TlsMutex );

pTlsInUseBits = &ppCurrentProcess->tlsInUseBits1;

// Position pTlsInUseBits so that it points at the first of the two
// tls bit DWORDs that has a free bit available.
while ( *pTlsInUseBits == 0xFFFFFFFF && (i < 2) )
{
    i++;
    pTlsInUseBits++;        // Point at next DWORD of tlsInUseBits.
}

if ( i < 2 )    // If a free bit-slot was found, i is 0 or 1.
{
    i *= 32;    // 'i' starts at either 0 or 1, so the end result is
                // either 0 or 32. There are 32 "inUse" bits in each
                // of the TlsInUseBits DWORDs.

    newFlag = 1;

    if ( *pTlsInUseBits & newFlag )
    {
        // Blast through the bits in this DWORD until we find one that's
        // 0 (available). Keep incrementing 'i' so that when we're
        // done, it's a TLS index.
        do
        {
            i++;
            newFlags << 1;
        } while ( *pTlsInUseBits & newFlag )
    }
    *pTlsInUseBits |= newFlag;  // Turn on the newly allocated bit to
                                // indicate that the corresponding TLS
                                // index is in use.
}
else    // No free bits were found.
{
    // If we get here, all the TLS indices were in use. Return -1 and
    // set the last error code.

    i = TLS_OUT_OF_INDEXES; // 0xFFFFFFFF

    InternalSetLastError( ERROR_NO_MORE_ITEMS );
}

_LeaveSysLevel( x_TlsMutex );

return i;
```

TlsSetValue

利用 TlsSetValue，线程可以将某个值存入另一个预先配置了的 TLS 槽。正如你所期待的，这里的两个参数是要写入的 TLS 槽以及被写入的数值。代码首先检验所通入的 TLS 索引应低于最大的 TLS 索引值(64)。在 Windows 95 的 β 构造中，TlsSetValue 函数检验了所通入的 TLS 索引应是预先配置好了的。从 β3 开始，TlsSetValue 仅进行最小的索引检查。如果通入的 TLS 索引低于 64，TlsSetValue 则从当前线程库将想储存的值装入 64DWORD 数组中的适当元素内。

另外(也是鲜为人知的一点)，TlsSetValue 还更新了与 TLS 数据数组平行的第二个 64DWORD 数组。该数据包括 TlsSetValue 被最后一次由当前线程调用时的 EIP 值。无疑，该 EIP 数组是用于调试目的的。但 Microsoft 似乎并未为程序提供一种办法来访问该信息。

TlsSetValue 伪码

```
// Parameters:
//      DWORD   dwTlsIndex;
//      LPVOID  lpvTlsValue;
// Locals:
//      PTHREAD_DATABASE    ptdb

    // The thread database starts 0x10 bytes before the TIB pointed to by
    // the FS register. Make a pointer to the thread database.
    ptdb = FS:[ptibSelf] - 0x10;

    if ( dwTlsIndex < TLS_MINIMUM_AVAILABLE (64) )
    {
        ptdb->TLSArray[ dwTlsIndex ] = lpvTlsValue;

        // Grab return EIP off the stack and store in the other TLS
        // array that runs parallel to the main array.
        ptdb->LastTlsSetValueEIP[ dwTlsIndex ] = [EBP+04];

        return TRUE;
    }
    else    // The TLS index passed in was >= TLS_MINIMUM_AVAILABLE.
    {
        ptdb->GetLastErrorCode = ERROR_INVALID_PARAMETER ;

        return 0;
    }
```

TlsGetValue

TlsGetValue 基本上可以看成是 TlsSetValue 函数的一个镜面影象，明显的差别是它检索而不是储存了某个值。象 TlsSetValue 一样，TlsGetValue 函数首先是检验所通入的 TLS 索引是否有效。如果有效，函数则利用此 TLS 索引值检索当前线

程的线索库的 64DWORD 数组。TlsGetValue 将此时的情况返回。

TlsGetValue 伪码

```
// Parameters:
//      DWORD   dwTlsIndex;
// Locals:
//      PTHREAD_DATABASE    ptdb
// The thread database starts 0x10 bytes before the TIB pointed to by
// the FS register. Make a pointer to the thread database.
ptdb = FS:[ptibSelf] - 0x10;

if( dwTlsIndex < TLS_MINIMUM_AVAILABLE (64) )
{
    // Set last error value to 0.
    ptdb->GetLastErrorCode =  ERROR_SUCCESS;

    return ptdb->TLSArray[ dwTlsIndex ];
}
else    // The TLS index passed in was >= TLS_MINIMUM_AVAILABLE.
{
    ptdb->GetLastErrorCode = ERROR_INVALID_PARAMETER ;
    return 0;
}
```

TlsFree

TlsFree 函数解除了前面 TlsAlloc 和 TlsSetValue 的调用，它利用所通入的 TLS 索引并检验其应是预先配置好了的。如果符合要求，TlsFree 则关闭处于 64TLS 槽位数组中的相应位。为了防止程序使用存于前一配置 TLS 槽中具有潜在无效的数值，TlsFree 将 0 值存进刚被释放的 TLS 槽。其结果是，如果某个特殊的 TLS 索引之后被重新配置，所有使用那条索引的线程在他们调用 TlsSetValue 之前将确保返回一个值。

TlsFree 伪码

```
// Parameters:
//      DWORD   dwTlsIndex;
// Locals:
//      DWORD   retValue
//      PDWORD  pTlsInUseBits;
//      PTHREAD_DATABASE ptdb;
//      PK32OBJECTLISTENTRY pK32Object;

    x_LogSomeKernelFunction( function number for TlsFree );
```

```
    _EnterSysLevel( pKrn32Mutex );

    _EnterSysLevel( x_TlsMutex );

    point pTlsInUseBits to either ppCurrentProcess->tlsInUseBits1 or
    ppCurrentProcess->tlsInUseBits2 as appropriate.

    if ( dwTlsIndex < TLS_MINIMUM_AVAILABLE (64) )
    {
        DWORD turnOffFlag;

        // Create a DWORD with the appropriate flag set that represents the
        // TLS index to be freed.
        turnOffFlag = 1 << ( dwTlsIndex & 0x1F );

        // If that bit is already turned off in the process database's
        // tlsInUseBits field, the TLS index isn't allocated. This is
        // a bad thing, so go report an error.
        if ( 0 == turnOffFlag & *pTlsInUseBits )
            goto error;

        // Turn off the correct bit in the tlsInUseBits field of the process
        // database.
        *pTlsInUseBits = ~turnOffFlag;

        // Now walk through each of the threads of the process, putting the
        // value 0 into the DWORD assigned to the TLS index we're freeing.

        pK32Object = x_GetNextObjectInList(ppCurrentProcess->ThreadList, 0);

        while ( pK32Object )
        {
            ptdb = pK32Object->pObject;

            ptdb->TLSArray[dwTlsIndex] = 0;
            ptdb->AnotherTLSArray[dwTlsIndex] = 0;

            pK32Object=x_GetNextObjectInList(ppCurrentProcess->ThreadList,1);
        }

        retValue = 1;
    }
    else
    {
error:
        retValue = 0;
        InternalSetLastError( ERROR_INVALID_PARAMETER );
    }
    done:
        _LeaveSysLevel( x_TlsMutex );

        _LeaveSysLevel( pKrn32Mutex );

        return retValue;
```

其它线程函数

本节所介绍的函数不归属前述的任何一类函数，但在展示线程库的 KERNEL32. DLL 用途上同样是相当重要的。

GetLastError

利用 GetLastError，应用程序可以确定为什么某个特殊系统的调用会失败。当某个 Windows 95 函数失败时，它可以有选择地在当前线程中设置最后的出错代码以指示为什么函数会失败。这些出错在 WINERROR. H 文件中。从某种程度上讲，GetLastError 函数类似于 C 运行库中的错误变量。

除了系统函数之外，应用程序也可自行调用 SetLastError。从理想角度来讲，这些出错代码应处于系统定义的出错值的范围之外。

GetLastError 的执行很简单。在检测存在某条要查询的当前线程之后，代码从当前线程的线程库返回 GetLastError 区段（偏置 70h）。如果不存在当前线程，函数则返回在 KERNEL32 全局中所出现的一切，如下列伪码所示（全局变量没有给定名称）：

GetLastError 伪码

```
if ( ppCurrentThread )
    return ppCurrentThread->GetLastErrorCode
else
    return x_LastErrorIfNoCurrentThread;    // A global variable.
```

SetLastError

SetLastError 的执行也很简单。如果存在当前线程，函数则返回线程库中 GetLastError 区段的数值。

SetLastError 伪码

```
// Locals:
//      DWORD   fdwError

    if ( ppCurrentThread )
        ppCurrentThread->GetLastErrorCode = fdwError;
```

GetExitCodeThread 和 IGetExitCodeThread

GetExitCodeThread 函数返回了由 hThread 参数特定的线程的当前出口状

态。线程的出口状态保留在了线程库的 TerminationStatus 区段(偏移量 48h)。在正常的执行当中,出口状态是 0x103(STILL_ACTIVE)。

GetExitCodeThread 函数是一个参数有效的外壳。在检验了某个要存放出口状态的有效的指针通过之后,跳至 IGetExitCodeThread。

IGetExitCodeThread 首先进入一个“绝对完整”区间。其后,将 hThread 参数转换成一个线程库指针。在检索出线程库中 TerminationStatus 区段的数值之后,代码将离开“绝对完整”状态。

GetExitCodeThread 伪码

```
// Parameters:
//      HANDLE  hThread;
//      LPDWORD lpdwExitCode;

    Set up structured exception handling frame

    *lpdwExitCode += 0;     // Verify that lpdwExitCode can be written to.

    Remove structured exception handling frame

    goto IGetExitCodeThread;
```

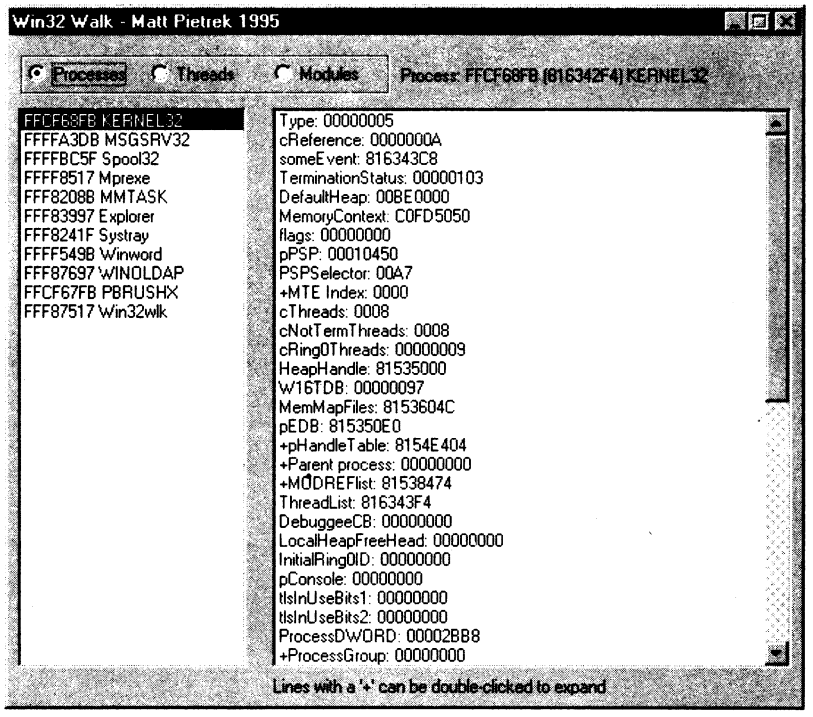
IGetExitCodeThread 伪码

```
// Parameters:
//      HANDLE  hThread;
//      LPDWORD lpdwExitCode;
// Locals:
//      PTHREAD_DATABASE ptdb;
//      BOOL    retValue;

    retValue = TRUE;        // Assume successful return.

    x_EnterMustComplete();

    x_LogSomeKernelFunction( function number for GetExitCodeThread );

    ptdb = x_ConvertHandleToK32Object( hThread, 0x80000020, 0 );

    if ( ptdb )
    {
        *lpdwExitCode = ptdb->Status;

        x_UnuseObjectWrapper( ptdb );
    }
    else
    {
        retValue = FALSE;
    }

    LeaveMustComplete();

    return retValue;
```

Win32Wlk 程序

为了清晰地展示我在这一章里所介绍的各种数据结构的概念，我编写了 Win32Wlk 程序。除此之外，该程序还可用于考察 KERNEL32 数据结构。

如图 3-5 所示（分成三个屏幕栏），Win32Wlk 是一个 GUIWin32 程序。Win32Wlk 所显示的三个基本的数据结构是进程库，线程库，和 IMTE（模块）。另外，Win32Wlk 还显示了三个附加的数据结构：进程柄表，进程模块列表，和线程信息块（TIB）。



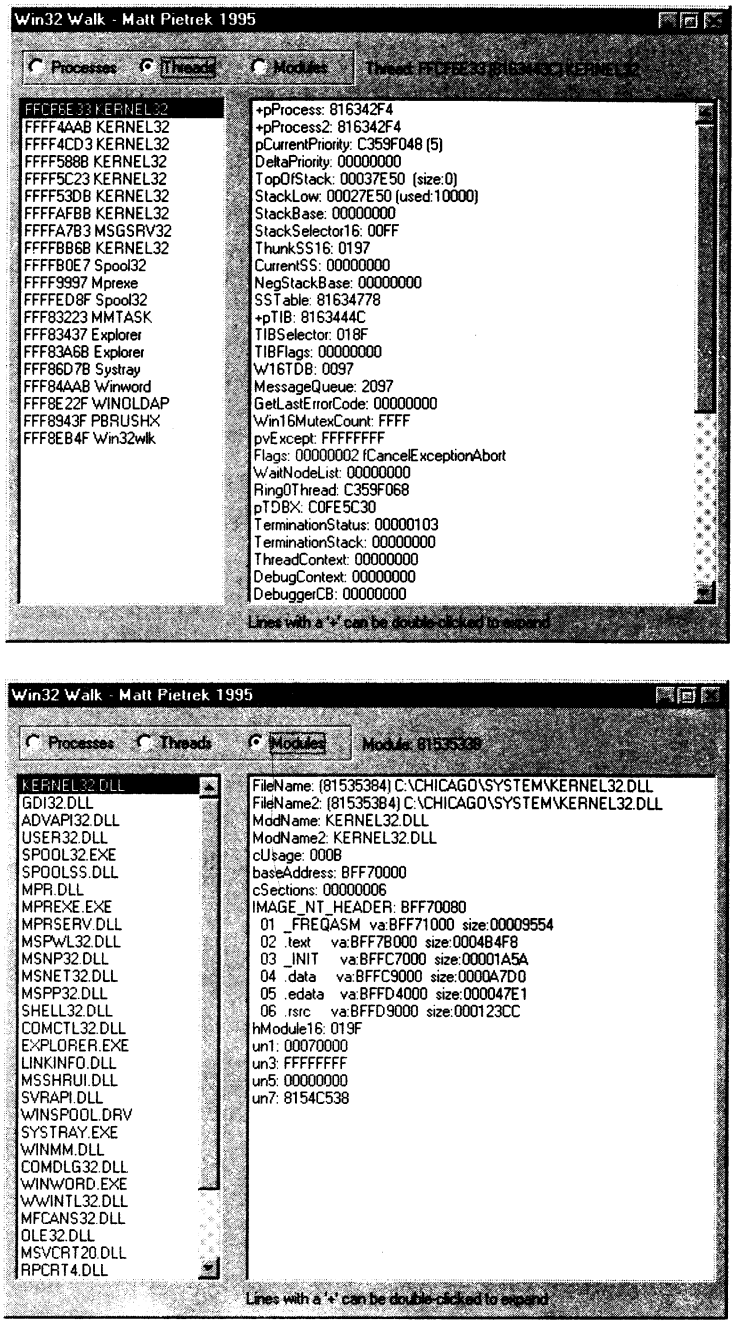


图 3-5　Win32Wlk 程序由三个基本的数据结构组成：进程列表，线程列表，和模块列表

Win32Wlk 是在两个输出栏之间切换的。左边的输出栏所显示的或者是进程，或者是线程，或者是模块的一个当前列表。你可以通过激活窗口左上部的对应区

(进程,线程,或模块)在这三个列表之间切换。要记住,左列表栏中的输出在新的进程、线程或模块进入或离开系统时,并不动态地更改。如果你想更改此时的输出,只需按一下相应的键就可以了。

右输出栏为一详细窗。在某一时刻,它显示的是某个进程库、线程库、IMTE、进程柄表、进程 MODREF 列表或 TIB 的区段。该详细窗的内容以两种方式变更。首先,当你激活左侧输出栏的某一行时,详细窗变更所要显示的选择进程,线程,或模块的所有有价值的区段。另外一种方式是热链接法(hot links)。在详细窗的某些输出中,你会发现某些行的前面带有"+"符号,表明该行可以使用热链接法。例如,在显示进程细节时,pHandleTable 行实际上是+pHandleTable。如果利用"+"健双击这类行,详细窗则输出有关该项的细节。对于+pHandleTable 行,详细窗则变成了此时进程的进程柄表的输出。要记住,在详细窗内显示的区段与其在联合数据结构中出现的顺序并不一致,我所显示的某些数据结构中的每一区段也同样如此。在设计详细窗时,我试着利用相关性将信息分类。不太引起重视的意义含糊的区段列在了输出栏的底部。对于某个特殊区段的意义不明确而且总是显示为 0 的情况,我没有显示其值,而且声明该区段一旦为非零,我会立即知道。如果你正在使用 Win32Wlk 并遇到了这样的声明(这种声明一般是中止程序),你可能想跟踪哪一个区段提出了声明并想从声明列表中将其移走。

Win32Wlk 的内部解析

有关 Win32Wlk 程序中的技术难点以及实现的细节是有待进一步考察的。当我开始编制 Win32Wlk 时,我打算行至所有的数据结构而无需任何 TOOLHELP32 函数,而 Win32Wlk 基本上也达到了这一目标,但有两个例外,KERNEL32 中的进程和线程列表是很难在 Windows 95 中行走的。这主要是因为这类结构并不像 Win16KRNL386 使用时那样仅作为一些简单链接的列表。好在 TOOLHELP32 第 32 个进程 ID 和第 32 个线程 ID 的数值正好是由 GetCurrentProcessId 和 GetCurrentThreadId 所返回的 PID 和 TID 值。

因此我就无需使用 Process32First、Process32Next、Thread32First 和 Thread32Next 函数去得到一系列进程和线程的 ID。Win32Wlk 进而可以利用数据结构去寻找其它信息。

寻找 KERNEL32 在何处保存了指向 IMTE(即系统模块列表)的指针是另一个难题。指向 IMTE 指针数组的指针存放在 KERNEL32.DLL 当中(即我前面所提到的 pModuleTableArray)。关键是寻找 pModuleTableArray。对 KERNEL32.DLL 进行一定的搜索之后,我发现了非资源的 GDIReally 函数(KERNEL32 出口 #32)。该函数被调用之后,ECX 寄存器包含了 pModuleTableArray 指针,这是一个极为讨厌的编码方式。在做这类事之前,你应当了解一些有关严重警告之类的标准信息。当然,有些事还不得不做。尽管尚有其它方式可以找到 pModuleTableArray,但优越性不大。因此,Microsoft 没有制作一个返回系统模块所有列表的 TOOLHELP32 API 也并非一件太糟的事,至少避免了上述之类事件的发

生。

Win32Wlk 中其它一些棘手的事是如何将进程和线程 ID 转换成进程和线程指针。前面已经介绍了进程和线程 ID 是如何通过与 KERNEL32 Obsfucator DWORD 进行 XOR 运算之后而成为 KERNEL32 目标指针的。二元 XOR 的特点之一是其互换性。即，如果：

(A XOR B) == C

则

(A XOR C) == B

也就是说，如果你知道了 XOR 操作中的任两个值，就可以计算出第三个值。现在将 KERNEL32 的值代入上述方程：

(PTHREAD_DATABASE XOR Obsfucator) == ThreadId

对于 Win32Wlk，最初只有一个值（由 GetCurrentThreadId 所返回的 ThreadId）。我们尚不知道 Obsfucaotr DWORD 的值，而且不能使用硬编码值 (hard-coded value)，因为每次引导时此硬编码值是不同的。但如果我们能得到一个指向 THREAD_DATABASE 的指针，则可将方程改写为：

(ThreadId XOR PTHREAD_DATABASE) == Obsfucator

实践证明，有几种方法可以得到指向当前进程的 THREAD_DATABASE 的指针。我选择了最方便的方法来得到当前 TIB 的地址并减去 0x10。当前 TIB 总是由 FS 段位寄存器所指。你会在 TIB 的偏置 18h 上找到 TIB 的线性地址，其算法大体是：

```
pTIB = FS:[18h]
pThreadDatabase=pTib - 0x10
```

这一点可以在 WIN32WLK.C 的 InitUnobsfucator 函数中的一小段在线编译程序中完成。一旦你得到了 Unobsfucator 值，则进程和线程的问题就会迎刃而解。进程和线程的 ID 均是利用 KERNEL32 Obsfucator DWORD 计算出的。Win32Wlk Unobsfucator DWORD 在检索进程和线程库指针中的运行是等效的。

最后一点有关 Win32Wlk 的说明是，Win32Wlk 是值得进一步考察其在故障修复的 Windows 95 升级方面的运行情况的。Win32Wlk 可以很容易地寻找并解码这些数据结构，而 Windows 95 则可很容易地断开 Win32Wlk。例如，他们可以对 PID 和 TID 数值进行第二次 XOR 运算，从而轻微地扭转了 GDIReallyCares，因而在函数返回时指向 IMTE 指针数组的指针就不在 ECX 之内了。

如果 Microsoft 不愿意这样做，还有一种办法可以找到必要的信息。Microsoft 应当意识到，外人是不介意利用此类信息编制出更优秀的产品的。对于

Win32Wlk，其代码并不试图改变任何系统属性，只不过是为编程人员提供一种理解系统的更好的方法。在编程中所遇到的“黑盒子”对于编写“Hello World”程序很有用，但遗憾的是它对游戏程序之外的程序没有用途。

如果 Windows 95 编制人员真的考虑安全性问题并想使系统数据结构不易接触，他们应将 Windows 95 设计得更像 Windows NT。如果精明的程序员想了解这类信息，还是有办法的。为什么不在操作系统之上再增加一层“垃圾”来防止自己的信息被偷视呢？Windows 95 毕竟不是 Windows NT，并没有打算这么做，其实也没有必要。

小结

模块、进程、线程构成了 Windows 95 的结构核心。本章考察了其数据结构及各种 KERNEL32 函数对他们的应用。第八章有关文件格式一节将进一步探讨模块方面的内容。但是，几乎在后面的所有章节中你都会得到有关模块、进程或线程方面的知识。因此，了解这些内容是打开 Windows 95 宝藏之门的金钥匙。

第 四 章

USER 和 GDI 子系统

* *

正如本章标题所示，下面将对 Windows 95 USER 和 GDI 模块的特点进行探讨和描述。USER 模块包含了视窗子系统中全部的消息传递和管理代码，GDI 是视窗图形系统的核心。USER 和 GDI 两个模块相结合才能在显示屏上产生一个窗口。因此，可以想像得出，仅仅是对 USER 和 GDI 最上一层的描述，也可以写两本书了，这就是我为什么不讨论 USER 和 GDI 如何完成他们绝妙任务的原因，而是集中讨论 Windows 95 USER 和 GDI 是如何从 16 位的 Windows 3.1 及 Windows NT USER 和 GDI 演变而来的。

在本书中我甚至将不讨论具有 Windows 95 重要特征的新的 USER 和 GDI 的泛函性（如新的普通控制）。我甚至认为如写一本名为《WndProc Internals》的书将会受到欢迎的（开句玩笑）。在那本书中（纯粹是假设了），将全部列出 Window 过程的伪码（如按钮窗口，工具条窗口等等）。这其中与我们讨论关系最大的是桌面窗口过程，稍后我将给出其伪码。

现在你可能对我将讨论些什么和不讨论些什么应该是比较清楚的了。为了满足 Win32 API 的需要，Windows 3.1 USER 和 GDI 基本码不得不重新编写，使得这两个模块变化比较大。如果你对 Windows 3.1 工作思路比较清楚的话，这一章将帮助你转到 Window 95 新的工作方式上来。正如你所希望的，本章将分两部分，USER 和 GDI。

由于 USER 改变比较大，加之若掌握了 Windows 95 USER 的变化部分再去学习 GDI 就不会感到困难了，因此，USER 的内容可能会多一些。

Windows 95 USER 模块

在我写这本书的整个过程中，一直努力把 USER 的变化分清类型。然而，USER 系统的变化却不能简单地被归纳为一或二个特殊的类型。信息系统代码的绝大部分驻留在 16 位的 USER.EXE 中，然而整个 16 位模块中，还有 32 位的代码。16 位的 USER.EXE 的一些部分和 Windows 3.1 简直是一样的，其它部分和 Windows 3.1 差别就比较大了。

Windows 95 USER 模块也包含了 32 位的与 Win32 EXE 和 DLL 接口的 USER32.DLL。你可能已听说过 USER32.DLL 仅是转换成 16 位的 USER.EXE 文件

之一。USER32 中的大多数函数需转换成 16 位的代码，也有不需转换的，稍后我们将看到几个例子。

试图用几个简要的方框来说明 16 位的 USER. EXE 和 32 位的 USER32. DLL 设计和实现看来是不可能的。但我可以说，Windows 编程者在解决向下兼容性和由 Windows NT 要求的 Win32 规范这对矛盾上做的是最好的。

在许多情况下，向后兼容和遵守 Win32 API 的规范是有矛盾的。这就导致了不可避免地在设计上做些妥协，使得人们不会感到特别满意。但从总的来看，我认为 Windows 95 USER 的编程者在处理兼容性和 Win32 API 规范的矛盾上，是做的相当不错的，许多其他编程者也想这样做，但都不如 Windows 95 USER 做的出色。

为了对 Windows 95 USER 模块有个了解，先来看一下在 Windows 95 Win32 中 USER 的有关部分。Windows NT USER 是 32 位的，主要是实现 Win32 API。能做到向下兼容当然是好的，但不是绝对基本的。Windows NT 中 16 位 USER. EXE 模块是通过转换为 NT USER32. DLL 中真正的 USER 码来实现的。

Windows 95 的另一方面，也是最易被忽略的，是 Win32s 尽力提供 Win32 API，而同时并不减少 16 位 Windows 3. 1 USER. EXE 的驻留。对于 Win32s，是不许改变 16 位 USER. EXE 的。早在 Win32s 最初版本发行前一年，Win32s 编码者就不得不把大部分基本码进行固化了。

那么 Windows 95 USER 系统偏重哪一点呢？Win32 的支持者（包括我自己）喜欢把 Windows 95 看成是与 Windows NT 相同的思路，这并不是偶然的。Windows 95 是作为 Windows 3. 1 的新版本推向市场的，那么是不应该牺牲兼容性的。因为有太多的程序依赖于 16 位 USER. EXE 的特性和内部结构（在这一点上，Microsoft 评论是“瞧，我们告诉过你不要用未注明的软件！”）。

除了与现有程序不兼容外，Microsoft 在 16 位 USER. EXE 中保留了 USER 函数的核心部分，特别地是代码的尺寸。一般来讲，在 32 位编码中，对于许多指令会占用较大的内存空间。（客观地讲，这一点已引起了热烈的争论，我们可以举出很多例子，在单个指令时用 32 位指令占的字节会更少）。然而，总的来看，Microsoft 编程者认为若 USER 函数全部用 32 位码重新编写，其程序会增加 40%，结果 Windows 95 在 4MB 机器上与 Windows 3. 1 运行得一样糟（Microsoft 市场部人员希望我说“和其一样好”），重新用 USER 为 Win32 码（如 Windows NT 一样）并不是偶然的。所以，一个真正的 Win32 USER 子系统是不适用的，Windows 95 编程者们在下面的工作做得是最好的。他们是用 Windows 3. 1 USER. EXE 码开始的（与 Win32 编程者所不同），并且允许修改。Windows 95 的设计是针对 80386 以上微机的，对 USER. EXE 做了重大修改。在整个 16 位 USER. EXE 码段，都散布着 32 位指令。（这就是为什么你会在 Windows 95 USER. EXE 码段中发现许多超长操作码（66h））。

USER. EXE 大部分 16 位编码用的是 32 位的数据偏移量，使得程序看起来紧凑些。USER 的许多程序是用 C 语言写的。你可能知道，PC 机上的编译器在编译时用的是存储模式。一般 16 位 C 编译器，如 Borland C++产生 16 位码指令，用的

16 位偏移量的码段和附加数据段。即使是 16 位编译器也能产生 32 位指令，但段中寻址却不能超出 64K。

32 位编译器用的是扩展的存储模式，在这种模式下，PC C 编译器忽略了段的存在。其所产生的指令也不在依赖于码段选择器、数据段选择器或栈段选择器(即 CS、DS 和 SS 寄存器)。Windows 95 USER.EXE 看起来像是 16 位指令和扩展存储模式的结合，即 USER.EXE 码驻留在 16 位段中，该码又明显依赖于段寄存器。另一方面 USER 码中又包含了一个段中超过 64K 地址的指令。

考查下面从 USER.EXE 中摘录的一小段代码：

```
1ACA:   MOV     AX,SEG 0021:0000
1ACD:   MOV     ES,AX
1ACF:   MOV     EAX,ES:[062E]
1AD4:   CMP     WORD PTR ES:[EAX+46],BX
1AD9:   JNE     1ADC
1ADB:   RET
```

从上述指令的大小(如首条 3 字节指令)表明这是 16 位码，前两条指令在 USER DGROUP 偏移 062Eh 处用全局变量设置了段寄存器，但是第四条指令却把 EAX 寄存器作为计算地址的一部分了。在实际操作中，EAX 的数值会大于 128K。我还从来没有见过这种编译器，在产生 16 位码同时用的是 32 位的数据偏移量，这使我怀疑 Windows 95 USER 编程者们是否应用了 Microsoft 语言部新开发的特殊编译器(最新消息：在我写完这章后，不愿透露来源的消息证明这种编译器在 Microsoft 是存在的)。

尽管 16 位 USER.EXE 的许多改变是为了增加容量(因为运行时超出堆的空间一直是 Windows 3.1 未能解决的问题)，但许多是为了满足 Win32 API 的需要。(另外，Windows 95 编程者不得不尽力赶上 NT 编程人员)。例如把一个线程输入状态与另一线程联系起来的 Win32 AttachThreadInput 函数，没有 Win16 相似部分。这甚至在早期 16 位版本 Windows 也算不了什么，然 Windows 95 16 位 USER.EXE 却完全包含了完成 AttachThreadInput 的指令码。USER.EXE 是一合适的 DLL，并不输出 AttachThreadInput，而 USER.DLL 却支持它。如果你仔细研究一下，就会发现 USER.DLL 中的 AttachThreadInput 码只不过是转换的 USER.EXE。USER32.DLL 因提供了 Win32 API 而受到赞扬，其实 Cinderella 16 位 USER.EXE 也有类似的功能。

16 位 USER.EXE 和它的 Win32 相似的另一个例子是在资源上。你将在第八章看到存储在 Win32 Portable Executable(PE)文件中的资源与 16 位 New Executable(NE)文件中的格式是完全不同的。然而如第七章所示，Windows 95 为 32 位模块创建的 16 位 NE 模块数据库包含了一个指向存储器 Win32 中资源的指针。这就是为什么 16 位 USER.EXE 即支持老的 16 位 NE 格式资源，又支持新的 Win32 PE 格式资源的原因。USER32.DLL 中的与资源相关的函数就成为转换 USER.EXE 的角色了。

USER32 转换例子

现在讨论转换，这正是解释 Windows 95 转换如何工作的好时机。Windows 95 有两种码，16 位和 32 位，因此要想真正掌握 Windows 95 的结构，了解转换是不可避免的。现在来看一个典型的函数 SetFocus，USER 32 用它来转换成 16 位的 USER.EXE。SetFocus 带有一个参数，当 16 位码用此参数（一个 HWND）时，它的数值不需任何转换。（Windows NT 是不同的，但这是将来某本书的内容）。

SetFocus 函数

SetFocus 函数与其它 USER32 中转换为 USER.EXE 函数是相似的。在 USER32 debug 版本中，其码是通过调用一个记录函数开始的。如果 USER32 数据区域某一个地方被设置了特殊标志，该函数向 debug 口输出一串“[F] SetFocus”。USER32 SetFocus 主要部分是把一个基本 16:16 跳转表地址的变址装入 CL 寄存器。SetFocus 变址是 0x7E，这意味着跳转表中第 0x7E 入口是一个指向 16 位版本 SetFocus 的 16:16 指针。

把 0x7E 装入 CL 后，SetFocus 用 JMP 转到我命名的 ThunkToUSER16_One_Param 的一个小例行程序。这个小的例行程序是 USER32 例行程序的普通入口，USER32 是带有一个参数转换成 16 位 USER.EXE 的例行程序。ThunkToUSER16_One_Param 程序所做的工作是把调用函数的参数和转换变址压入栈，然后调用另一个我命名的 Common Thunk（下面将讨论到）例行程序。

SetFocus (32→16)伪码

```
LogWin16ThunkFunction1( "[F] SetFocus" );

CL = 0x7E   // Thunking index for SetFocus.

goto ThunkToUSER16_One_Param
```

ThunkToUSER16_One_Param 伪码

```
// Parameters:
//  DWORD   param1
//  DWORD   thunkIndex  // Actually in CL register.

    return CommonThunk( param1, thunkIndex );
```

Common Thunk 代码是很简单的，用 C 语言去写反而变得复杂了。由于某些未知的原因，这个例行程序驻留在 USER32 数据区。或许这个程序是在启动时建立的。但在任何情况下，其例行操作都是极其简单的。首先，它得到转换索引（例如

SetFocus 函数的 0x7E),将其作为进入 16:16 指针表数组的索引。例如程序从这个数组中检索出合适 16:16 地址,然后把它放到 EDX 寄存器。最后 Common Thunk JMP 到 KERNEL32.DLL(下面要讨论到)中的 QT_Thunk 例行程序。

Common Thunk 代码

```
// This code actually resides in USER32's data area.

XOR     ECX,ECX                 ;; 0 out ECX.
MOV     CL,[EBP-04]             ;; Grab the thunk index (pushed by
                                ;; ThunkToUSER16_One_Param).

MOV     EDX,[8014E264+4*ECX]    ;; Index into the array of 16:16 pointers
                                ;; into the 16-bit DLLs. Put the appropriate
                                ;; 16:16 pointer (e.g., SetFocus) into EDX.

MOV     EAX,offset KERNEL32!QT_Thunk    ;; Jump to the QT_Thunk routine
JMP     EAX                             ;; in KERNEL32.DLL.
```

QT_Thunk 函数

QT_Thunk 函数是从 KERNEL32.DLL 输出的。QT_Thunk 是个一般函数,是被需要转换为 Win16 码的 Win32 码调用的。换句话说,它的用途并不局限于 KERNEL32 和 USER32。事实上,如果你看一下从 Win32 SDK 转换编译程序输出的汇编程序(THUNK.EXE),你将看到它引用了 QT_Thunk 例行程序。

QT_Thunk 用汇编语言编写非常清楚,并在占用空间和运行速度上进行了优化。这部分我简要地对给出的函数原始汇编程序讨论一下。然而,除了非常短的一组汇编语言外,代码是不允许插入的。因此在下面的伪码中,你将看到的即有 C 语言又有汇编语言,我意在尽最大努力表达一个相当复杂的例行程序。你若要想了解其运行,最好的办法是在 SoftIce/W(或其他调试程序)中 QT_Thunk 设置断点,并分步执行它。我保证用不了多长时间你就会碰到断点的。

从运行流程来了解例行程序(如 Microsoft 工作人员所说的),QT_Thunk 的工作是相当简单的:取出 EDX 中的 16:16 地址,然后把控制转到该地址。当然,事情并不像这样说的简单,还有其他一些事情需要注意。对于初学者,把完成 16 位码后要返回的地址存起来是非常有益的。同样把选择器从扩展 32 位栈选择器转向 16 位栈选择器也是一个好想法。

对这个例行程序再进一步研究,QT_Thunk 可被分成五个步骤。首先在 debug 下程序调用一个记录调用的例程(假定记录标志设置正确,这往往是容易出错的)。这部分代码也检验 Thread Information Block(TIB)选择器和 FS 寄存器是否一致。如果不同,例程给出警告(在 debug 下是这样的)。

QT_Thunk 第二步是把转换的最终地址压入栈。(我们在第五章还将讨论)。这一步也处理返回地址的保留和 32 位寄存器变量。在 16 位码完成后,控制返回地

点的 32 位返回地址被存到不被干扰的栈和某个区域里。被保存的寄存器变量是 ESI、EDI 和 EBX，这些是 Win32 编译时所需的普通寄存器变量。

第三步 QT_Thunk 得到 Win16Mutex。现在我们都知道，任何时候 32 位码转换 16 位码时，操作系统都需获得 Win16Mutex。Win16Mutex 只不过是一个刚好驻留在 KRNL386.EXE 数据段的循环运行互斥(mutex)标志。通过强制所有 Win32 码转换成 16 位而获得 Win16Mutex，Windows 95 能够保证在同一时间只有一个线程通过 Win16 系统 DLL(其他 16 位 DLL 也是一样)。

这就是 Microsoft 如何避免 16 位系统 DLL 不用多线程编写而产生问题的想法。全部 Win16Mutex 对象都引起了激烈争论，仅仅是这个问题就可以轻而易举地写上一章。我将在“信息系统改变”一节再谈一下，这里只简单地说明 QT_Thunk 例程是 Windows 95 获得 Win16Mutex 地方之一。

第四步 QT_Thunk 从 Win32 码用的扩展 32 栈转向 Win16 码用的 16:16 栈。因为 Win32 线程有典型的 1MB 栈。在转换过程中，ESP 可以在 1MB 内任何地方，你可以认为转向 16:16 栈可能是个技巧。在线程建立和设置其基地址过程中，仅仅是分配 16 位栈选择器是不够的，而是在转换为 16 位过程中，QT_Thunk 例程可能调整执行 16 位码时线程使用选择器的基地址。16 位选择器被设置成指向在转换之前 ESP 使用的同一普通线性地址区域。

栈选择器处理完后，QT_Thunk 计算出适当的 16 位 SS:SP，把计算出的数值装出 SS 和 SP 寄存器。

第五步，也是 QT_Thunk 最后一步，是把控制转向预先设置的转换目标地址。如我在第二步所指出的，依据 QT_Thunk 入口，16:16 目标地址被存入 EDX，随后被压入栈。通过 RETF 技术 QT_Thunk 跳转到 16:16 地址。但在转移控制之前，QT_Thunk 码把所有不需要的段寄存器清零(DS,ES,FS 和 GS)。QT_Thunk 并不向 16:16 目标函数提供带有比较好的、丰富的 32 选择器的 DS 寄存器，若需要的话，16:16 函数可将建立段寄存器。

QT_Thunk 伪码

```
// On entry, EDX contains the 16:16 address to transfer control to.

//
// Phase 1: logging and sanity checking
//
if ( bit 0 not set in FS:[TIBFlags] )
    goto someplace else;            // Not interested in that here.

PUSHAD          // Save all the registers.

SomeTraceLoggingFunction( "LS", EDX, 0 );   // EDX is 16:16 target.

// Make sure that the FS register agrees with the TIB register stored
// in the current thread database.

if ( (ppCurrentThread->TIBSelector != FS)
      && (ppCurrentThread != SomeKERNEL32Variable) )
```

```
{
    _DebugOut( SLE_MINORERROR,
               "32=>16 thunk:  thread=%lx, fs=%x, should be %x\n\r",
               ppCurrentThreadId, FS, ppCurrentThread->TibSelector );

}

POPAD           // Restore all the registers.
//
// Phase 2: saving away the return address and register variable registers
//
POP     DWORD PTR [EBP-24]      // Grab return address off the stack
                                // and store it away for later use.

PUSH    DWORD PTR [someVariable]    // ???
PUSH    EDX     // Push 16:16 address on the stack. The RETF
                // at the end will effectively JMP to it.

MOV     DWORD PTR [EBP-04],EBX  // Save away the common
MOV     DWORD PTR [EBP-08],ESI  // compiler register variables.
MOV     DWORD PTR [EBP-0C],EDI

//
// Phase 3: Acquiring the Win16Mutex
//
PUSHAD, PUSHFD                  // Save all registers.
_CheckSysLevel( pWin16Mutex )
POPFD, POPAD                    // Restore all registers.

FS:[Win16MutexCount]++;
if ( FS:[Win16MutexCount] == 0 )
    GrabMutex( pWin16Mutex );

PUSHAD, PUSHFD                  // Save all registers.
_CheckSysLevel( pWin16Mutex )
POPFD, POPAD                    // Restore all registers.

//
// Phase 4: Saving off the old SS:ESP and switching to the 16:16 stack
//
Calculate the 16:16 stack ptr. Set EBX for the SUB EBP,EBX instruction below.

MOV     DX,WORD PTR [EDI->currentSS]    // Load DX with 16-bit SS.

MOV     DI,SS       // Save away the flat SS value into DI.
                    // (The callee is expected to preserve it.)

MOV     SS,DX       // Load SS:(E)SP with the 16-bit stack ptr.
MOV     ESP,ESI

SUB     EBP,EBX     // Adjust EBP for the thunk.
MOV     SI,FS       // Save away FS (TIB ptr) register into SI.
                    // (The callee is expected to preserve it.)
 //
 // Phase 5: Jumping to the 16:16-bit code
```

```
//
GS = FS = ES = DS = 0;  // Zero out the segment registers.

RETF                    // Effectively does a JMP 16:16 to the address
                        // passed in the EDX register.
```

16 位码完成任务以后，QT_Thunk 转向 32 位码。完成上述步骤还涉及到其它部分。尽管这里我也可以把它们列出来，但我认为没有太大必要。这里要注意在整个转换 16 位码例子中，没有任何扩展 32 指针参数需要转换成 16:16 地址。这种转换码是比较复杂的，这里我们就不涉及了。

32 位堆

USER 子系统变化最大的可能是增加了 32 位堆。你可能发觉任何 Win32 程序都可以访问 32 位堆和使用通过 Win32 HeapXXX API(如 HeapAlloc，HeapFree 等等)提供的堆服务。但你可能不了解 16 位 USER. EXE 和 16 位 GDI. EXE 也应用 32 位堆存储一定的项。16 位 USER. EXE 和 GDI. EXE 实际转换 32 位 KERNEL32. DLL 是从特殊 32 位堆分配存储单元的，这个特殊的堆是为 16 位 USER 和 GDI 模块特殊建立的。尽管这些特殊堆是为 USER 和 GDI 服务的，但它们与 Win32 程序 GetProcessHeap 堆的格式是一样的。例如你可以用第五章的 WALKHEAP 程序浏览 UESR 或者 GDI 32 位堆(尽管你首先需将它们定位，这一点后面我们再讨论)。

为什么 32 位堆带来这些问题呢?在 Windows 95 以前版本的 Windows 中，USER 和 GDI 使用的内存分配是标准的 LocalAlloc 形式的，其最大是 64K。不用说，这将对这么多窗口和图形系统模块在任何给定时间进行保存是个相当大的限制。通过把这些大的目标模块移进 32 位堆，使 Windows 95 在系统容量上有了重大改进。每个这些特殊创建的堆是 2MB，所以在容量上一时还不是问题。

实际上 USER. EXE 使用了两个单独的 32 位堆。一个存放 WND 结构。在系统中每个窗口有一个 WND 结构。(本章稍后我们将看到 WND 结构)。另一个 USER 32 位堆存放菜单。GDI. EXE 仅有一个 32 位堆用来存放字体和区。对 USER 的 WND 和 MENU 字形和区相对比较大，所以把它们移出 16 位堆是正确的。

如果说 Windows 95 在 16 位部分增加了 32 位堆是个较大的新闻的话，那么这些堆的定位就更有意思了。你看当在 32 位堆中访问数据时，USER 和 GDI 却不用项的扩展模式线性地址，相反 USER 和 GDI 继续使用它们访问常规 128K DGROUP 的同一 DS 选择器。这是怎么回事呢? 32 位 WND 堆和 32 位 GDI 堆正好超过 16 位 DGROUP 区从 128K 开始。这听起来还有些不清楚的话，请看图 4-1。

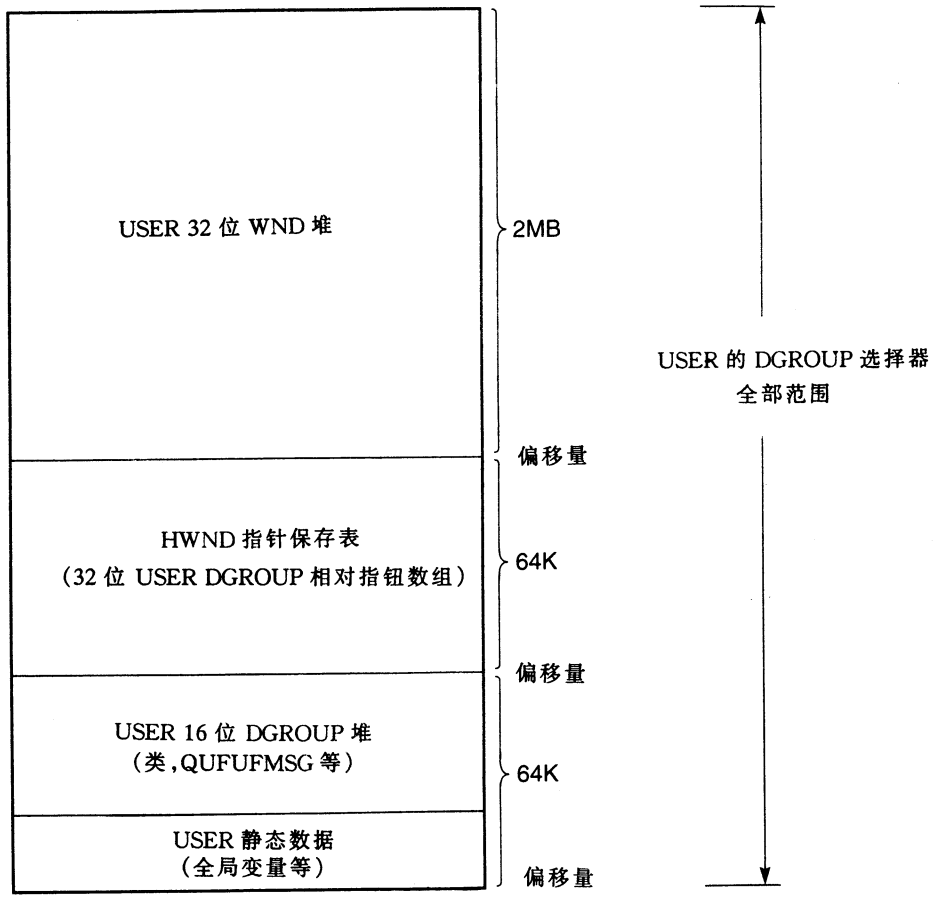


图 4-1　USER.EXE 的 16 位和 32 位堆框图

如前面我提到的,在 32 位堆中,USER 和 GDI 并不用指向项的 32 位扩展指针,而是存储基于 USER 或 GDI DGROUP 选择器基地址的偏移量,当然这些偏移量是 32 位的。例如,USER 的 16 位(128K)DGROUP 区最大是 64K,32 位 WND 堆从 16 位 DGROUP 区端部 128K 开始,这意味着你将在 Windows 95 中发现最低 WND 结构偏移量是 0x20000。在实际使用时(正如你在第五章看到的)Win32 堆的前 2 段被保留,典型 WND 结构偏移量应是 0x20924。因为这个偏移量不是扩展线性地址,所以它只有在选择器(即 USER 或 GDI 的 DGROUP)被确定后才有意义。当然,你若知道 USER 和 GDI DGROUP 段线性地址,你可以把该地址的数值与 32 位堆中目标偏移量相加或与访问数据目标的扩展线性地址相加。这章稍后介绍的 SHOWWND 程序正好做这项工作。

下面来说明通在常 DGROUP 上面,32 位 WND 堆真正开始于 128K,它是一个真正标准的 Win32 风格的堆。为了证明,我们将应用 SoftIce/W。首先,需要找出 USER 和 DGROUP 的基地址。为了完成这个工作,必须找到 DGROUP 句柄/选择器。正如第七章所指出的,对于模块 DGROUP 可以从 16 位模块数据库提取。

对 USER 使用 SoftIce/W MOD 命令产生下面结果：

```
:mod user
hMod PEHeader      Module Name      EXE File Name
17CF               USER              C:\WINDOWS\SYSTEM\user.exe
1857 0147:81537DB8 USER32            C:\WINDOWS\SYSTEM\USER32.DLL
```

现在我们知道 USER 模块句柄是 17CF，在模块数据库中偏移为 8 的地方是一个指向 DGROUP 段的 10 字节段记录的指针，把它显示出来：

```
:dw 17cf:8
17CF:00000008 0180 10D9 C341 0021 157C 0000 1F42 0015     ....A.!.|...B...
....
```

好，在 17CF:180 处是 USER 的 DGROUP 10 字节的段记录，段记录的最后一个字被指定为那个段的句柄，把这个段记录显示出就得到：

```
:dw 17cf:180
17CF:00000180 4042 0B02 0177 157C 16C6 0005 800C 000F     B@..w.|.........
```

现在我们知道 USER 的 DGROUP 的句柄是 16C6(与此相对应的选择器是 16C7)，让我们用 SoftIce/W LDT 命令得到这个选择器的线性地址(注意，段的极限超过了 64K)：

```
:ldt 16c6
16C7  Data16    Base=81D09000  Lim=0021FFFF  DPL=3  P   RW
```

由此知道 USER 的 DGROUP 的线性地址为 0x81D09000，把它与 0x20000 相加就得到了 USER32 窗口堆的开始地址。让我们试一下，给 SoftIce/W“Heap32”命令输入地址：

```
:heap 32 81d29000
Heap: 81D29000  Max Size: 2048K  Committed: 16K  Segments: 1
Address   Size      EIP       TID   Owner
81D290E0  00000088  BFFA0A27  0001  hpWalk+082D
81D29178  00000058  BFF71AA6  0001  IGetLocalTime+0942
81D291E0  00000058  BFF71AA6  0001  IGetLocalTime+0942
81D29248  0000005C  BFF71AA6  0001  IGetLocalTime+0942
81D292B4  00000058  BFF71AA6  0004  IGetLocalTime+0942
81D2931C  00000058  BFF71AA6  0007  IGetLocalTime+0942
81D29384  00000060  BFF71AA6  000A  IGetLocalTime+0942
81D293F4  0000005C  BFF71AA6  000A  IGetLocalTime+0942
81D29460  00000058  BFF71AA6  000A  IGetLocalTime+0942
81D294C8  0000005C  BFF71AA6  000A  IGetLocalTime+0942
81D29534  0000005C  BFF71AA6  000A  IGetLocalTime+0942
81D295A0  0000005C  BFF71AA6  000A  IGetLocalTime+0942
81D2960C  0000005C  BFF71AA6  000A  IGetLocalTime+0942
81D29678  00000058  BFF71AA6  000A  IGetLocalTime+0942
... rest of output omitted...
```

正如你看到的，SoftIce/W 对于我们输入的地址没有拒绝。事实上，它打印输出的结果看起来相当有逻辑性。特别地要注意整个块大约为 0x58 字节，后面就会知道，0x58 是 WND 结构的最小尺寸。通过使用窗口附加字(参见 WNDCLASS 结构的 cbWndExtra 域部分，它被用来寄存一个类)，这块可能比这个尺寸略大。通过上述说明，看起来确实是在 USER 的 DGROUP 段开始的上部驻留着一个 Win32 堆。

在这一点上，你可能会有疑问，为什么 32 堆开始于越过 USER 或者 GDI DGROUP 段尾的 128K(你过去可能就有，是吧?)，为什么堆不正好在 16 位 128K DGROUP 域尾开始呢?简单地说，答案就是句柄!尽管使用 USER 的 DGROUP 32 位偏移量 WND 结构可以被访问，但这种麻烦的向下兼容性意味着 HWND 必须是 16 位的。

在 Windows 3. x 或者更早的版本中，HWND 只不过是一个进入 USER DGROUP 段的偏移量。明显地当 WND 结构进入 USER 的混合的 16/32 位 DGROUP 时，它不能工作。为了能把一个 16 位的数值(如一个 HWND)映射成一个 32 位偏移量，USER 和 GDI 使用在它们的 16 位 DGROUP 和 32 位堆之间的 64K 区域，作为句柄表。特别地，一个句柄值(如一个 HWND)仅是一个进入句柄表的偏移量。如图 4—2 所示，通过句柄表给出的偏移量，你就能找到实际数据的 32 位偏移量(相对于合适的 DGROUP)。

为了说明句柄表，再一次运行 SoftIce/W。选出桌面 HWND，通过句柄表向上看。SoftIce/W WND 命令提供了窗口列表的分层显示，在下面输出中，窗口 HWND 是 0x80：

```
:hwnd
Window Handle   hQueue  SZ  QOwner    Class Name        Window Procedure
0080(0)         1437    32  MSGSRV32  #32769            17B7:571C
 00B4(1)        1A4F    32  EXPLORER  Shell_TrayWnd     1457:0140
  00B8(2)       1A4F    32  EXPLORER  Button            1457:01AE
  00BC(2)       1A4F    32  EXPLORER  TrayNotifyWnd     1457:01C4
... rest of windows omitted...
```

如果前面讲的是正确的话，现在我们应当把 HWND 数值加到 0x10000，用它在 USER 的 DGROUP 发现一个双字，就是 WND 结构地址。0x1000＋0x0080＝0x10080，所以在 16C7:10080 显示存储器：

```
:dd 16c7:10080
16C7:00010080 0002:0178 0002:01E0 0002:0248 0002:02B4     x.......H.......
```

忽略“:”冒号将导致 SoftIce/W 死机(它试图把其数值显示成 16:16 指针)，WND 结构偏移量是在 0x20178 的地方。因为 USER 的 DGROUP 的线性地址是 0x81d09000，那么 WND 结构的线性地址为 81D29178。回头再看前面 SoftIce/W 给出的 32 位用户堆，你就会看到 0x81D29178 确实是堆中的一个块地址。这次好

像前面讲的都给验证了。

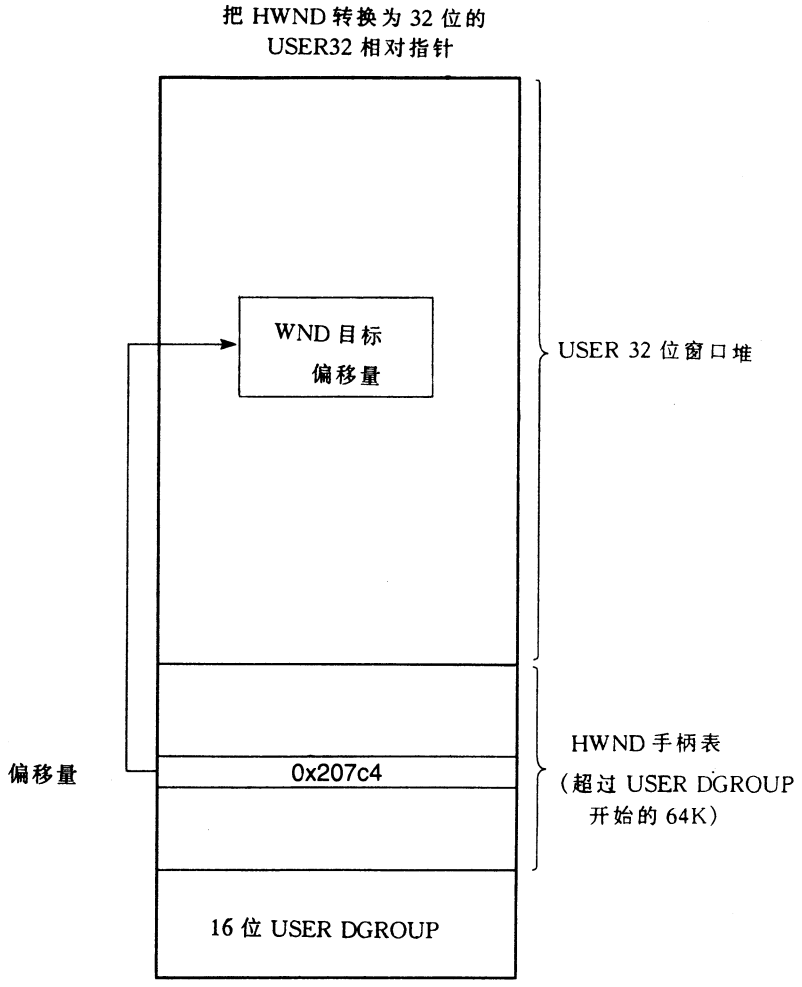


图 4-2　由句柄表给出的偏移量中找到的实际数据的 32 位偏移量

从 GDI 的 32 位堆中可以看出，GDI 目标码的结构与其句柄表的结构基本相同。例如，一个被保存在 32 位堆并被 HRGN 结构引用的区，你可以对 HRGN 应用一系列相同的步骤，找到这个区结构的实际线性地址。

如果句柄表区是 64K 并每个句柄表是真正的 4 字节 DWORD 指针的话，句柄最大的数值应是 16384(65,536/4==16,384)。Microsoft 声称可以有多达 32,767 个窗口和 32,767 个菜单，但我却不知道这数字从何而来。然而，还未说明的是在你设法创建 1.6 万(或 3.2 万)个窗口时，最有可能会受到其它系统的限制。

前面我提到过 USER 也有一个 32 位的菜单堆。菜单堆的域和句柄表区与 USER 窗口堆在运算上是相同的(当然，地址是不同的)，唯一不同的是 64K16 位 DGROUP 在句柄表的下面。你可能认为 Microsoft 把菜单分开分别放入各自的 32 位堆中比较好，但这并没有你想像的改变那么大。在 Windows3.1 中，菜单已经被

分开放入各自的 16 位堆中了。在 Windows 95 中唯一的改变是增加了菜单堆的尺寸。顺便提一下，菜单堆的基址选择器可以通过稍后讨论的 UserSeeUserDo 函数得到。

如果 USER 和 GDI 使用的 32 位堆在功能上与 Win32 应用的堆一样的话，对 Win32 堆操作的 KERNEL 32 函数也同样可以被 USER 和 GDI 堆使用，情况确实是这样的。当 USER 为一个 WND 结构分配内存时，KERNEL32. DLL 中实现 HeapAlloc 函数的码通过一个转换被调用。然而，USER 和 GDI 不能直接转换成 KERNEL32。更确切地说，KRNL386. EXE 提供了一套未说明的函数负责调用 KERNEL32 堆码。KRNL386 函数如下：

KRNL386. 209—Local32Alloc
KRNL386. 210—Local32ReAlloc
KRNL386. 211—Local32Free
KRNL386. 213—Local32Translate(把句柄转换成 16:16 地址)
KRNL386. 214—Local32FreeQuickly

尽管这些函数名都以 Local32 开始，它们真正调用的是相似的 Heapxxx 函数(例如，Local32Alloc 调用 HeapAlloc)。第五章将说明 Win32 局部堆函数仅是没有多少内容的 Win32 HeapXXX 函数的包装程序。列表中要特别注意的是函数 214。这个函数似乎要创造标明一个自由块的网络效应，并不实际转换成 KERNEL32。然而，一些关键的事情，如把块加到自由列表上，该例程并没有做。

神秘的 GetFreeSystemResources 释放

讨论完 32 位堆后，我们可以有足够的基础来讨论增加的神秘 FreeSystemResources 释放。我讲神秘，尽管找不出任何理由，但 Windows 95 中的平均 FreeSystemResource 数看起来增加很大。因为对大多数非编程者来说，他们仅在 USER 和 GDI 中注意到了自由系统资源，所以我先来讨论它。如果自由系统资源增加了，那它必定是好的，对吗？不要急于下结论！

自由系统资源的数量对于在各种系统堆中剩余的存储容量来说，确实是一个有意义的词，特别地被用百分比说明的时候。在 Windows 3.1 中，自由系统资源只有最小的几个百分率。这个百分率是在 USER DGROUP 堆，USER 菜单堆，USER 串堆(似乎在 Windows 95 中消失了或者说不重要了)，及 GDI DGROUP 堆中自由空间的大小。除了这些堆，带有最小百分率的堆自由空间变成了自由系统资源。

在 Windows 95 中，FreeSystemResources 的计算开始路径是相同的，但在结束时有个不能预料的转折。简单地说，在 Windows 95 中 FreeSystemResources 开始好像是下述五个堆中最低的百分率：

1) USER 16 位 DGROUP 堆

2) 32 位窗口堆
3) 32 位菜单堆
4) 16 位 GDI 堆
5) 32 位 GDI 堆

因为三个 32 位堆都是 2MB 的，它们的自由百分数通常达到了可笑的高值，如 99%。因此，无论出于何种目的，都不应把它计入自由系统资源。只剩下两个 16 位 USER 和 GDI DGROUP 堆了。若哪一个有较小的百分率就意味着系统资源的百分率。因为还有一些 USER 和 GDI DGROUP 的项不好计入，因此他们不应当有接近 96%的自由空间(启动 Window 95 时，你可以在 Explorer About 框中看到典型的数值)。

在这方面，我建议你做个小实验。引导 Windows 95 然后立即启动 CALC，或者 Explorer，或者一些系统带来的标准应用程序。选择 Help|About 得到 About 对话框，它将显示自由系统资源的数值。在 Windows 95 下，典型的如 96%。这听起来有点高的话，你是对的。如你将稍后看到的 GetFreeSystemResources 伪码，USER 和 GDI 在它们堆中任何地方都没有接近 96%的自由空间。

那这究竟是怎么回事呢？简单地说，Window 95 是“一本需认真钻研的书”书，Windows 95 自由系统资源是一个相对的数字，而不能简单地报告堆中最低自由百分率。毫不怀疑，你将发问“相对于什么呢”？

Windows 95 报告的自由系统资源是与系统被调用后还有多少是自由空间有关。特殊地，在系统被调用及 Explorer 完成自己任务后，Windows 95 快速地计算一下真正的自由百分率。随后，当你询问系统自由系统资源时，它报告一个与原始快速计算有关的自由百分率。

让我们看一个例子。假定 Windows 95 调入和运行，真正的自由系统资源值是 75%。也就是说，你已经运行了一些应用程序，现在每个堆中只有 50%自由空间了。Windows 95 将报告系统资源是 66%(50/75)而不是真正的 50%，如果这不是有意的话，我真不知道为什么。也许 Microsoft 感到应让他的用户相信 Windows 95 已真正解决了自由系统资源问题。诚然，Windows 95 用它的 32 位堆使问题改进了不少，但并没有像说的那样大。

为避免被 Microsoft 所责怪，我这里对 Microsoft 为什么改变自由系统资源计算方式，唯一的解释是：有一个被充分定义最大量的系统资源可用的存储器。启动和创建窗口(如桌面及窗口形式)都消耗一部分内存，这部内存没有办法回收，为什么还报告呢？从终端用户角度来看，新的自由系统资源数值可以更精确，如果他有 50%自由资源，那么他就是用了大约 50%的可用容量。终端用户不知道(可能也不关心)系统本身占去了多少自由系统资源。

GetFreeSystemResources 函数

即然我们基本了解了新的自由系统资源计算方法，下面来看 Windows 95 提供此数值的细节。GetFreeSystemResources 是在 16 位 USER.EXE 中实现的(必要

时，SHELL32. DLL 转换并得到在系统效用 About 框内的数值）。该函数本身仅是我在第三章描述的标准参数验证层存根。在对输入的变化检查正确后，GetFreeSystemResources JMP 到 IGetFreeSystemResources 码。

IGetFreeSystemResources 有三个不同的代码部分。第一部分由 USER 和 GDI 部分的自由百分率数值组成。USER 自由百分率是 USER 16 位 DGROUP，32 位窗口堆和 32 位菜单堆的最低自由百分率。GDI 自由百分率是通过调用被 GDIFreeResources 调用的 16 位 GDI. EXE 函数得到的。在这部分码的后部，函数有两个自由资源数值，一个是 USER，另一个是 GDI。

IGetFreeSystemResources 的第二部分是在考虑 USER 和 GDI 堆空间在启动时被系统占去多少后进行调整。这部分码的关键是两个 USER. EXE 全局变量；我给这两个变量的名字是 base_USER_FSR_percentage 和 base_GDI_FSR_percentage。在 USER. EXE 数据段中，这两个变量初始值为 0。当 IGetFreeSystemResources 被调用时，如果这两个变量为 0 的话那么函数对以前计算出的 USER 和 GDI 百分率自由数值不做任何调整。然而，若这两个全局变量不是 0，那么在 Windows 95 启动后，他们包话了 USER 和 GDI 堆中的自由百分率，IGetFreeSystemResources 用启动时的这些数值除以当前 USER 和 GDI 自由百分率数值，得到相对百分率。

当我第一次发现这些全局变量时，第一个问题就是“谁设置的”？你认为是 Explorer？（即使你在屏幕上没看见 Explorer 窗口，Explorer 仍然是一个正运行的进程。现在我提醒你，Explorer 并没有直接进入 USER 的 DGROUP 段设置 base_USER_FSR_percentage 和 base_ GDI_ FSR_ percentage 变量。简单地说，是让 USER. EXE 自己做这件事情。这是怎么做的呢？从某种意义上讲当 Explorer 确定 USER. EXE 被完全建立后，USER. EXE 向桌面窗口过程送一个带有 0x400 处 MSG 数字（WM_USER）窗口消息。正如你稍后看到的，WM-USER 消息的桌面 WNDPROC 句柄建立了这些全局变量，这转来转去不太好理解。如果在桌面 WM_USER 消息被送以前你有个过程或一个调用 GetFreeSystemResources DLL，你将得到一个和该消息被传送后完全不同的数值。

IGetFreeSystemResources 的第三部分是调用输入的参数。如果你查询 USER 或者 GDI 自由资源（GFSR_USERRESOURCES 或者 GFSR_GDIRESOURCES），代码返回一个先前计算的适当数值。如果你询问 GFSR_ SYSTEMRESOURCES，函数将返回较小的 USER 和 GDI 百分率。

GetFreeSystemResources 伪码

```
// Parameters:
// UINT    fuSysResource

    // Is the input parameter within range?
    if ( (fuSysResource < 0) || (fuSysResource > 2) )
    {
```

```
    // Calls LogParamError.
    HandleParamError( ERR_BAD_VALUE );
}

// JMP to the real code.
return IGetFreeSystemResources( fuSysResource );
```

IGetFreeSystemResources 伪码

```
// Parameters:
//  UINT    fuSysResource
//  WORD    gdiResourcePercentage, userResourcePercentage

    //
    // Phase 1:  Getting USER and GDI's percentage free
    //
    if ( UserTraceFlags & 0x200 )
        _DebugOutput( DBF_USER, "GetFreeSystemResources" );

    userResourcePercentage =
        GetPercentFree16BitHeap(hInstanceWin);  // Get 16-bit DGROUP % free.

    // Call GDI and let it do its heap free calculations.
    gdiResourcePercentage = GDIFreeResources( 0 );
    // Take the lesser of the USER's DGROUP and the 32-bit menu heap.
    // (Gee, I wonder which one it will be???)
    if ( GetPercentFree32BitHeap(hMenuHeap) < userResourcePercentage )
        userResourcePercentage = GetPercentFree32BitHeap(hMenuHeap);

    // Now take the lesser value of the previous calculation and the
    // percentage free in the 32-bit window heap.
    if ( GetPercentFree32BitHeap(hWindowHeap) < userResourcePercentage )
        userResourcePercentage = GetPercentFree32BitHeap( hWindowHeap );

    //
    //  Phase 2: Cooking the books
    //

    // Adjust the percentages so that they're relative to the percent
    // free after booting. This might be an attempt to make Windows 95 look
    // like it has more free system resources than Windows 3.1.
    if ( base_USER_FSR_percentage )
    {
        userResourcePercentage = MulDiv( userResourcePercentage, 0x100,
                                         base_USER_FSR_percentage );

        gdiResourcePercentage  = MulDiv( gdiResourcePercentage, 0x100,
                                         base_GDI_FSR_percentage );
    }

    if ( userResourcePercentage > 99 )
        userResourcePercentage = 99;
```

```
if ( gdiResourcePercentage > 99 )
    gdiResourcePercentage = 99;

//
// Phase 3
//
switch ( fuSysResources )
{
    case GFSR_SYSTEMRESOURCES:
        return min( userResourcePercentage, gdiResourcePercentage );

    case GFSR_GDIRESOURCES:
        return gdiResourcePercentage;

    case GFSR_USERRESOURCES:
        return userResourcePercentage;

    default: return fuSysResources;
}
```

GetPercentFree16BitHeap 和 GetPercentFree32BitHeap 函数

GetPercentFree16BitHeap 和 GetPercentFree32BitHeap 函数是两个被 IGetFreeSystemResources 使用的帮助例程。两个函数均需要一个说明堆性质的参数。GetPercentFree16BitHeap 函数使用了未注明的 GetHeapSpace 函数，这个函数将在第五章讨论。它认为自由空间(用 K 表示)与全部空间(也用 K 表示)的比率就是自由百分率。

GetPercentFree32BitHeap 有点复杂。它用的基本代码与 Windows 95 16 位 TOOLHELP 函数输出作为 Local32Info 函数是相同的。对于讨论的堆，该代码返回 dwMemCommitted、dwTotalFree 和 dwMemReserved 域。dwMemCommitted 和 dwMemReserved 域看起来总是相同的，dwTotalFree 值也总是正好在那。在从 dwMemCommitted 域减去 dwTotalFree 域，函数用 dwMemReserved 域去除该结果。因为这些数据基本上是相等的，GetPercentFree32BitHeap 函数返回的典型数值是 98%或 99%。

GetPercentFree16BitHeap 伪码

```
// Parameters:
//  HGLOBAL hHeap
// Locals:
//  DWORD   freeK, totalK
//  DWORD   myDWORD

    myDWORD = GetHeapSpaces( hHeap );   // See Undocumented Windows,
                                        // Chapter 5.
    freeK = LOWORD(myDWORD) / 1024;

    totalK = HIWORD(myDWORD) / 1024;

    return (freeK * 100) / totalK
```

GetPercentFree32BitHeap 伪码

```
// Parameters:
//  HGLOBAL hHeap
// Locals:
//  LOCAL32INFO local32Info;
//  WORD percentUsed;
// Call the same function that TOOLHELP.DLL's Local32Info uses.
local32Info.dwSize = sizeof( LOCAL32INFO );
if ( KRNL386_Local32Info( &local32Info, hHeap ) == 0 )
    return 0;

if ( local32Info.dwMemReserved == 0 )    // Some problem here officer???
    return 0;

percentUsed =
    CalculatePercentage(
        100 * (local32Info.dwMemCommited - local32Info.dwTotalFree),
        local32Info.dwMemReserved );

// percentUsed is typically some ridiculously low value, like 1%. Thus
// this function usually returns 99% free for 32-bit heaps.

return 100 - percentUsed;
```

从 32 位代码获得自由系统资源:不用转换编译器的转换

不管你是否相信,Windows 95 没有为 32 位应用程序提供一个简易的方法使其从 32 位程序获得自由系统资源(FSR)数值。即使 Windows 95 在 About 框中显示 FSR 时,32 应用程序也是从 SHELL32. DLL 中的 32 位到 16 位转 换中获得 FSR 数值的。如果你正在编写一个 32 位程序和要调用现存的 16 位系统函数(如 GetFreeSystemResources),你可能要花上几个小时(或几天)来学习 Windows 95 转换编译器,然后再写一对转换 DLLs. Ugh。这可能是个好办法。

如我在“SetFocus 函数”部分讨论的,USER32. DLL 一直是转换为 USER. EXE 的,没有分开的 16 位和 32 位 DLL 转换。相反 32 位 SetFocus 码却调用了我先前在“QT_Thunk 例程”部分描述的 QT_Thunk 函数。你也可以在你的程序中调用这样的例程,但这比使用标准的 Win32 API 函数稍微复杂一点,需要点技巧。它是一个未注明的函数(尽管你会看到 THUNK. EXE 转换编译器表明引用了它),并且在你调用它时,还得使用点汇编语言。

在你的码中调用 QT_Thunk 时需要你做两件事情:首先,你必需把 16:16 地址送入 EDX 寄存器。第二,你必须保证你调用的 QT_Thunk 码有一个 EBP 栈,并且有你不用的 0x3C 字节的空间,这里为 QT_Thunk 在 EBP 寄存器指定的下方要为调用 16 码建立环绕式栈框。

为了说明在你的程序中调用 QT_Thunk，我写了个 FSR32 程序，这是个用 QT_Thunk 获得自由 USER 和 GDI 系统资源的程序。下面 FSR32 是源文件代码 FSR32.C，且比较短。使用下面的 Visual C++命令对其编译：

```
cl fsr32.c k32lib.lib thunk32.lib
```

你也可以使用本书源程序盘上的 BUILFSR BAT 文件。

```
//==================================
// FSR32 - Matt Pietrek 1995
// FILE: FSR32.C
//==================================
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <stdio.h>
#pragma hdrstop

typedef int (CALLBACK *GFSR_PROC)(int);

// Steal some #define's from the 16-bit WINDOWS.H.
#define GFSR_GDIRESOURCES       0x0001
#define GFSR_USERRESOURCES      0x0002

// Prototype some undocumented KERNEL32 functions.
HINSTANCE WINAPI LoadLibrary16( PSTR );
void WINAPI FreeLibrary16( HINSTANCE );
FARPROC WINAPI GetProcAddress16( HINSTANCE, PSTR );
void __cdecl QT_Thunk(void);

GFSR_PROC pfnFreeSystemResources = 0;   // We don't want these as locals
in
HINSTANCE hInstUser16;                  // main(), since QT_THUNK could
WORD user_fsr, gdi_fsr;                 // trash them...

int main()
{
    char buffer[0x40];

    buffer[0] = 0;  // Make sure to use the local variable so that the
                    // compiler sets up an EBP frame.

hInstUser16 = LoadLibrary16("USER.EXE");
if ( hInstUser16 < (HINSTANCE)32 )
{
    printf( "LoadLibrary16() failed!\n" );
    return 1;
}

FreeLibrary16( hInstUser16 );   // Decrement the reference count.

pfnFreeSystemResources =
    (GFSR_PROC) GetProcAddress16(hInstUser16,
```

```
"GetFreeSystemResources");
        if ( !pfnFreeSystemResources )
    {
        printf( "GetProcAddress16() failed!\n" );
        return 1;
    }

    __asm {
        push    GFSR_USERRESOURCES
        mov     edx, [pfnFreeSystemResources]
        call    QT_Thunk
        mov     [user_fsr], ax

        push    GFSR_GDIRESOURCES
        mov     edx, [pfnFreeSystemResources]
        call    QT_Thunk
        mov     [gdi_fsr], ax
    }

    printf( "USER FSR: %u%%  GDI FSR: %u%%\n", user_fsr, gdi_fsr );

    return 0;
}
```

The output from FSR32.C looks like this:

```
C:\NEWBOOK\USERGDI>FSR32.EXE
USER FSR: 90%  GDI FSR: 90%
```

FSR32.C 码中有两个事情需要讨论。一是 FSR32.C 是怎样从 32 位码中获得 16 位 GetFreeSystemResources 函数地址的？FSR32.C 使用了三个未注明的函数(Loadlibrary16,Freelibrary16,GetProcAddress16)与 16 位系统 DLL 一起工作。附录 A 给出了 KERNEL32 中的全部未注明的函数列表。为了成功地把 FSR32 与这些函数联接起来，需要你在第 3 章看到的 K32LIB.LIB 库(该库在附录中 A 讨论)。

为了保证栈中有足够的空间使 QT_Thunk 发挥作用，FSR32.C 声明了一个没任何作用的 0x40 个字符局部数组，使得 QT_Thunk 码能够随意使用这部分存储器。FSR32.C 把重要的变量都声明为局部的，以免被 QT_Thunk 破坏(学到这些我是费了好大劲!)。

FSR32.C 实际调用 QT_Thunk 的是汇编语言码，FSR32.C 在用 C 去调用 QT_thunk 的理由是由于在调用前 EDX 需要建立调用的 16:16 地址。理论上讲在正常调用 QT_Thunk 前可以用内部汇编把地址装入 EDX，然而在 CALL 指令执行前，你还要考虑使编译器不破坏 EDX 寄存器。

最后要注意的是这些并不需要很高的技巧，如给 16 位码传送一个指针等。Win32 API 函数(它转换成 16 位码并传送指针给 16 位 DLL)中有为建立别名选择器而精精心编写的代码。如果你要应用点技巧，建议使用转换编辑器。上述例子只传送一个参数，并且 16 位码使用时并示需要任何转换。参数需要转换的例子包

括指针和窗口信息数值。简而言之，在你决定避开或直接使用转换编译器之前要仔细考虑。

窗口系统的混合 16/32 位特性

前面我说过 WND 结构存放在 32 位堆中，因此它的相对于 USER DGROUP 的偏移量大于 64K。我还说过，HWND 被限于是 16 位数值，所以在 16 位 DGROUP 和窗口堆之前的区域被用于转换 HWND 为 WND 结构的指针句柄表。

在这一点上，不考虑 16 位和 32 位码/数的混合，强调窗口系统的双态性是特别重要的。首先要清楚的是在整个系统中用的是 16 位 HWND 数值，不管运行的是 Win16 还是 Win32 应用程序都没关系。被传送的 HWND 是 16 位的，并且偏移量被放到窗口堆句柄表中。再清楚说一下：一个 HWND 就是一个 HWND，就是它本身。不论你是在 Win16 或是在 Win32 码中：HWND 均是 16 位数值，并不像在 Windows 3.1 中那样简单地把偏移量放入 USER DGROUP 中。

既然你知道 HWND 在任何地方都是 16 位的，我可以告诉你在 USER.EXE 内部通常把这些 HWND 转换成 32 位指针。这些指针是与 USER 的 DGROUP 相联的指针，不是扩展的 32 位指针。WND 结构本身正是 USER 使用这些特殊 32 位指针的最好例子。一个 WND 结构前四部分分别是父窗口、本窗口、子窗口和属窗口。USER 把 32 位指针(非 16 位 HWND 数值)放到适当的父、子、本和属窗口中。这很可能是运算上的原因，因为 USER 在窗口层需要把 HWND 转换为指针。稍后我将回到窗口层。

当然对 WND 结构 USER 内部可能使用 32 位指针，但它与外面的界面仍然是 16 位的 HWND。所以必须有一个快速和简易的方法从 16 位 HWND 转换为 32 位指针，这方法确实有。稍后我们在看几个窗口函数伪码时你就会看到了。当试图同时支持在同一系统 Win16 和 Win32 应用程序时，难解的问题是窗口的过程是不同的。当不带 typedef 名字时，Win16 应用程序有个如下过程：

```
WndProc16(  unsigned short hWnd,
            unsigned short wMsg,
            unsigned short wParam,
            unsigned long  lParam );
```

另一方面，Win32 应用程序有个 WNDPROC 如下：

```
WndProc32(  unsigned long hWnd,
            unsigned long wMsg,
            unsigned long wParam,
            unsigned long lParam );
```

最大的问题是：当 Win32 应用程序在 16 位程序中对一个窗口完成 SendMes-

sage 时会发生什么？很显然，将发生某种问题。除非参数被重新排列或截短。同样，如果一个 16 位应用程序给 Win32 应用程序送一个消息，大多数被压入栈的参数被加宽(hWnd、wMsg 和 wParam 参数)。因为不能希望应用程序处理这些细节，所以 USER.EXE 来完成这项工作。

另一个相联系的问题是窗口子类程。窗口程序已把其它应用程序的窗口归为子类程。子类程的基本想法是一个程序使用 GetWindowLong(GWL_WNDPROC)为窗口搜索当前的 WNDPROC 回收地址并存储该数值。下一步，程序用 SetWindowLong(GWL_WNDPROC)改变窗口 WNDPROC 地址到应用子类程过程。现在这里有个问题：32 位应用程序创建的窗口 WNDPROC 是个 32 位线性地址。如果一个 16 位应用程序改变 32 位窗口 WNDPROC 地址为 16:16 地址，无疑将处于一个尴尬的境地。调用 WNDPROC 的 32 位码需要一个扩展的 32 线性地址，被定为段的 16:16 地址作为扩展的 32 位线性地址肯定不能工作。

为了防止这些明显的问题，USER.EXE 为每一个被创建时带有一个 32 位 WNDPROC 的窗口建立一个小代码存根。这个存根是 16 位代码，它为 Win32 应用程序使用作为自己 WNDPROC 的真正的 WNDPROC 包含了 32 位线性地址。这里有个 Explorer 的 tray 窗口的存根例子：

```
:u 1457:156
1457:00000156   PUSH    0040180D    ; A 32-bit WNDPROC address.
1457:0000015C   PUSH    00030000
1457:00000162   JMP     0127:7555
```

此段的下面是：

```
:u 1457:140
1457:00000140   PUSH    00401DFA    ; A 32-bit WNDPROC address.
1457:00000146   PUSH    00030000
1457:0000014C   JMP     0127:7555
```

正如你想像的，地址 0127:7555 是某种转换例程(在 KRNL386.EXE 中)，它为 Win16 WNDPROC 调用把参数转换成 Win32 WNDPROC 所希望的参数形式，然后在转换时调用被说明的地址。至于这些转变驻留的段，是 USER.EXE 从全局堆中分配的，码段替换入口选择器(0x1457)也被建立。

上述这些意味着什么呢？如果你研究任何窗口 WND 结构，你会总是发现一个为 WNDPROC 给定的一个 16:16 地址。然而，若你查看在那个 16:16 地址的存储器内容，你就会知道这是否是原来的 Win16 WNDPROC 还是转换的 Win32 WNDPROC。当然对 GetWindowLong(GWL_WNDPROC)函数还有：依赖于是被 Win16 程序还是 Win32 程序调用，以给出相应的地址。

信息系统的变化

Windows 95 USER 子系统改变比较大的地方之一(与以前版本相比)就是窗

口的信息处理。我称记录、传送、处理信息的这部分码为信息系统。关于 Windows 95 信息系统最好的消息是它取消了对 Win32 应用程序的同步特性。在 16 位 Windows 中，在某一时刻只能执行一个任务。当调用信息 API 时，该任务不得不放弃控制。典型的是尽管 SendMessage 也产生任务停滞，但通过主循环调用 GetMessage 或者 PeekMessage 时，也引起任务停滞。

这种方式的问题是任务被不规则地停滞（就是压入消息）阻碍了其他任务的实施，这就把输入系统挂起来了。只要 Win16 任务不调用 GetMessage 或 PeekMessage，就不能执行任何任务。只有花费长时间完成处理的任务，才让出系统无用的部分。

在做 Windows NT 时，编程人员重新做了 USER，停滞和运行才不受一个任务是否调用 GetMessage 或 PeekMessage 的影响。Win32 程序可以在处理信息时对其他过程没有什么影响。即然 Windows NT 做了此项改进，Windows 95 当然没有理由不采用了。

当然，如果 16 位应用程序可以继续在 Windows 95 正确运行的话，这些信息系统对 Win16 应用程序好似没有改变。太多的 Win16 应用程序依赖合作多任务模式，此时应用程序直到自己准备好以后才停滞。因此，只有 Win32 才可以以自己的步骤处理信息（或就根本不处理）时而对系统其他部分没有影响。

Windows 95 建立这种双重模式（Win16 应用合作多任务，而 Win32 应用程序事先占用多任务）的方法之一是通过 Win16Mutex 线程排队。在任何给定的时间，Windows 95 线程排队器均排好了已准备运行具有最高优先权的线程。没有使线程运行准备好的事情之一是它等待获得一个互斥信号（如 Win16 Mutex）。

无论何时当 Win16 任务正在执行时，它就占有 Win16Mutex（实际上说得更准确些，当任何 16 位码执行时，是 Win16Mutex 被占有）。当一个 Win16 任务运行时，有关需要调用 GetMessage 或 PeekMessage 任务的所有旧规则仍然有效，目的是保证 16 位任务能够运行。然而仅因为一个 Win16 任务控制 Win16Mutex 并不意味着线程排队器向其打开。当一个 Win32 线程通过正常 32 位码运行的时候，并需要占有 Win16Mutex。因此即使一个 Win16 任务没有及时传出消息，至少 32 位线程可以继续运行。其他 16 位应用程序当然不行了。

这种设置有个问题。因为信息系统代码是在 16 位 USER.EXE 中的，一个使用信息处理循环的 32 位应用程序，在它处理 USER.EXE 之前，需要获得 Win16-Mutex。所以这种事先占有多任务是不完全的。如果一个线程正在进行计算或其他不需转换 16 位 DLL 如 USER.EXE 的工作时，对于不传出消息的错误应用程序，这个线程是不受干扰的。然而，如果一个 Win32 线程需要调用 USER，GDI 或其他 16 位程序时，它需要获得 Win16Mutex，该线程被卡住直到 Win16Mutex 可调用为止。因此一个有缺陷 16 位应用程序可以卡住其他运行的应用程序（假定那些应用程序是使用信息系统或者相关函数的）。

这里我们所看到的是带有一系列 Achilles 后果的事先占有多任务系统。这后果就是不能及时传送消息的 Win16 应用程序。尽管你不能一起去掉 Win16Mutex，但你可以从你的系统中尽可能多地消除 Win16 任务。通过缩短得到 Win16Mutex

的时间，你也可以减少缺陷程序将把较入系统挂起的概率。

Microsoft 声称在 Windows 95 中所做的设计改进之一是增加了一个“粗输入线程”（即 RIT）。在所有 Microsoft 表示进入系统消息的图中，显示出中断处理器是把消息存入中央系统队列的。

然后，一个独立的线程（RIT）继续监视该线程，搜索消息和向相应的线程消息队列分配消息。（下面部分就来讨论消息队列的细节）。

尽管从理论上讲，RIT 听起来不错（假定 Windows NT 也使用此方法），但我还不能证实 Windows 95 RIT 的存在。我在 KERNEL32. DLL 中发现了一个调用 DispatchRITInput 函数。然而，在这个例程中设置断点并检查当前例程时，遇到断点却表明，它不是被一个单线程调用的。更确切地说，当 DispatchRITInput 被调用时，许多应用程序成为当前线程。最后，在 16 位 USER. EXE 中 Dispatch RITInput 转换为 DispatchInput 例程。我试着在那里设置断点，尽管断点常常消失，但它仍然是被其它线程调用的。我试着在 USER 中对其它内部信息系统函数做相同实验，但未能发现仅被单系统线程调用的特殊例程。最后我不得不停下来，向 Windows 95 开发者询问，得到的答复是：

确实是有个真正的 RTI，但如果我们可以在某一随机线程上可以处理一般情况的话，我们应用 RTI 是为了速度/效率，而不是预定 RIT。这就是你看到的为什么 DispatchInput 被调用的原因。我们把事情推至 RIT 是作为最后的一个解释。

《Unauthorized Windows 95》和 Win16Mutex 问题

在 Unauthorized Windows 95 552 页上，Schulman 引用了我在 PC Magazine 和 Microsoft Systems Journal 上的几篇文章来说明有关 Win16Mutex 是不正确的。他引用的几句如下：

……把你的应用程序移向 32 位，越快越好。如果你的系统中没有任何 16 位程序运行的话，Win16Mutex 是不会出问题的……

然后他说：“一个 Windows 95 系统（至少Windows95β-1）总是有两个运行的 Win16 任务，TIMER 和 MSGSRV32”。

从 Microsoft Systems Journal 中 Schulman 又引用了几句：

USER 和 GDI 码将很快执行和释放 Win16Mutex。在任何重要的时间段内，不存在保留 Win16Mutex 不释放的 32 位线程。

然后他给出了一小段 Win32 程序（W16LOCK），此程序获得 Win16Mutex 后只要不想释放就一直占有它。

下面两点值得探讨。第一点（即一个 Windows 95 系统总是至少有两个 Win16 任务）有点变化，最新的进展是 Windows 95 真的只用一个 16 位任务 MSGSRV32。在你的系统中可能有一个 16 位 MSGSRV32 任务，但它不是被要求的，我去掉它后没产生什么不良后果。一个重要的任务是允许你在 DOS 提示下启动的。为了查明 MSGSRV32 是否真的是个问题，我使用 SoftIce/W 在 KRNL386. EXE 中 CurTDB 变量设置了一个断点。该断点只在 MSGSRV32 HTASK 被写时才被检查。只要 MS-

GSRV32(系统中长位任务)变成激活任务后,就会遇到断点。大多数情况下,我能够把 MSGSRV32 变成当前任务的唯一方法是在提示命令下启动应用程序。在其它时间,MSGSRV32 能成为当前任务是非常偶然的。

查阅 MSGSRV32 码,没有发现 MSGSRV32 想拖延和不及时处理信息的地方。我看到的其停止工作时是 MSGSRV32 通过 WinExec 启动了另外一个程序。

整个 WinExec 调用执行期间,Win16Mutex 一直被占用。这里关健的问题是你不能从 Windows 95 中去掉全部的 16 位应用程序。另一方面,MSGSRV32 看起来好像是不用获得 Win16Mutex 和占有它。

至于第二点(即 W16LOCK 程序从一个程序中获得和占有 Win16Mutex),我的感觉 W16LOCK 是个特殊情况。它给出了 Windows 95 允许对系统函数和同步目标进行访问的特例。然而 W16LOCK 清楚地表明是从 Win32 程序中获得和占有 Win16Mutex 的。这不是 Win32 应用程序偶然这样做的(如果他们转换成 16 位 DLL,那是另外一回事了)。我承认 Win16Mutex 可能引起麻烦,但另一方面,如果你取消非系统 Win16 应用程序和不想破坏系统,你可能永远注意不到 Win16Mutex 的影响。换句话说,请对 Win16Mutex 保持警惕。

线程消息队列

在 Windows 95 中,每一线程可以拥有自己的消息队列。简单地说,消息队列就是一种数据结构,它控制着哪个消息被 GetMessage 调用或者被 PeekMessage 搜索。在 Windows 3.1 和以前版本中,每一个 Win16 任务都有自己的消息队列。消息队列是在程序建立后很快创建的。在 Windows 95 中,每个线程拥有自己的消息队列,这个队列只在线程实际需要时才首次创建。因为在 Windows 95 中每个 Win16 任务有一个联合线程,所以每个 Win16 任务继续拥有一个单消息队列。

让我们仔细研究一下消息队列,因为他们是贯穿整个 USER 子系统主要数据结构之一。当一个线程调用 GetMessage 或者 PeekMessage 时,它要寻找当前线程队列内的消息。当前线程的说明是隐含在 GetMessage 和 PeekMessage 代码内的。你不能够从另一线程队列请求消息。消息队列也被用来作为向其他程序传送消息的一部分。从 USER 的角度,SendMessage 的调用是从一个消息队列到另一个消息队列(尽管源和目的是同一个)。

这里我不打算涉及 GetMessage,PeekMessage 或者 SendMessage 的所有细节,我在 Windows Internals 已深入地讨论过。虽然从 Windows 3.1 到 Windows 95 有些变化,我感到再重复一遍大多数相同的内容益处不大。相反,是把精力集中在从 Windows 3.1 到 Windows 95 的改变上。

消息队列的格式

对于初学者,先来看一下消息队列的格式是什么样的。每个消息队列被存储在

USER.EXE 从 16 位全局堆中分配的一个段里。每个线程数据库(第三章)和任务数据库(第七章)包含了相关联消息队列选择器。一个消息队列已知域是在 SHOWWND 程序中的 MSGQUEUE.H 文件里。这些域的详细情况如下(注意:每一入口的前三项是偏移量,类型和名字):

00h　　WORD　　nextQueue

上述 WORD 包含了表中的下一个队列。全部消息队列被存在一个链表中,此域中用 0 表示结束。

02h　　WORD　　hTask

这个 WORD 保存和此队列相关联的 HTASK。如我在第七章所示,即使 Win32 过程也有一个与他们相关联的 16 位任务数据库。

04h　　WORD　　headMsg

此处 WORD 保存一个指向 QUEUEMSG 链表头的近指针(QUEUEMSG 在下面讨论)。

06h　　WORD　　tailMsg

此处 WORD 保存一个指向 QUEUEMSG 链表尾的近指针(相对于 USERD 的 DGROUP)。

08h　　WORD　　lastMsg

WORD 保存指向一个通过调用到 GetMessage 或者 PeekMessage 被恢复的 QUEUEMSG 的一个近指针。确切地说,此时消息没有被确定。

0Ah　　WORD　　cMsgs

此处 WORD 是等待队列中的消息号码。(即由偏移量 04h 指向列表中 QUEUEMSG 结构中的号码)。

0Dh　　BYTE　　sig[3]

对于 Win32 应用线程队列,这三个字节保存代表"MJT"的 ASCII 码。(可能代表 Jon Thomason,Microsoft 编程人员)。对于绝大多数 Win 应用程序线程,这三个字节为 0。

10h　　WORD　　npPerQueue

这个 WORD 是一个指向 PERQUEUEDATA 结构的近指针(相对于 USER 的 DGROUP)。这个结构保存着每个线程激活的,焦点的,捕获的窗口。这些概念我将在"队列系统窗口"部分描述。

16h WORD npProcess

这个 WORD 是一个指向 QUEUEPROCESSDATA 结构的近指针。如果一个过程有多线程和多队列，所有队列中的域将指向同一 QUEUEPROCESSDATA 结构。QUEUEPROCESSDATA 结构包含有诸如与此队列相关联的过程 ID 等信息，这将稍后讨论。

24h DWORD messageTime

这个 DWORD 保存通过调用 GetMessageTime 得到的数值就是消息被传递的时间。当每个消息被 GetMessage/PeekMessage 恢复时，这个值是从 QUEUEMSG 结构拷贝来的。

28h DWORD messagePos

这个 DWORD 保存通过调用 GetMessagePos 得到的数值。(此时的光标 X,Y 座标)。当每个消息被 GetMessage/PeekMessage 恢复时，这个数值是从 QUEUEMSG 结构拷贝来的。

2Eh WORD lastMsg2

这个域保存一个指向最后被恢复的 QUEUEMSG 结构的近指针(相对于 USER 的 DGROUP)。

30h DWORD extraInfo

这个 DWORD 保存通过调用 GetMessageExtraInfo 得到的数值。当每个消息被 GetMessage/PeekMessage 恢复时，这个数值是从 QUEUEMSG 结构拷贝来的。

3Ch DWORD threadId

这是个与此队列相关的线程的线程 ID，线程 ID 和线程数据库的关系已在第三章讨论了。

42h WORD expWinVer

这是个应用程序设想的 Windows 版本，通常用 0x300，0x30A，0x400 来代表 Windows 3.0，3.1，4.0。这个数值在启动时的程序标题部分。在一定的情况下，它被 USER 用来决定信息应当怎样处理和哪个信息被传送。它允许 USER 与多版本 Windows 兼容。

48h WORD ChangeBits

该 WORD 由最近一次调用 GetQueueStatus 后代表发生的各种消息事件类型的 QS_XXX 标志组成。

下面是 WINUSER.H 中 QS_XXX 标志：

```
QS_KEY              0x0001
QS_MOUSEMOVE        0x0002
QS_MOUSEBUTTON      0x0004
QS_POSTMESSAGE      0x0008
QS_TIMER            0x0010
QS_PAINT            0x0020
QS_SENDMESSAGE      0x0040
QS_HOTKEY           0x0080
```

这个 WORD 在 GetQueueStatus 返回 DWORD 的低字节中。第七章窗回内部结构中有有关 QS_XXX 标志及其意义的更详细介绍。

4AH　　WORD　　WakeBits

这个数值是由代表队列中各种消息类型的标志组成。QS_XXX 标志在先前描述域描述。此处 WORD 在 GetQueueStatus 返回 DWORD 的高字节中。

4Ch　　WORD　　WakeMask

如果一个线程被卡住，等待 GetMessage 或 PeekMessage 中的调用信息，这个字 WORD 保存它等待信息类型的 QS_ XXX 标志。特别是应用程序在调用 GetMessage 时可能被卡住，所以这个域将保存 QS_ALLINPUT，这是个所有 QS_XXX 标志的集合。

50h　　WORD　　hQueueSend

如果这个线程处理的是另一个线程送来的消息，这个 WORD 保存传送线程队列的句柄。

56h　　WORD　　sig2

这个 WORD 保存着 0x5148，是 HQ 的 ASCII 代码（可能代表 Handle Queue?）。每个消息队列是与一个特殊线程相联系的。反过来每个线程又与一个过程相联系。因此，消息队列与一个过程是多对一的联系。

一个过程中所有队列之间一般信息组成的信息系统被放在一个我称之为 QUEUEPROCESSDATA 的结构中。QUEUEPROCESSDATA 结构被存储在从 16 位 USER 堆中分配的一个块中。指向 QUEUEPROCESSDATA 结构的指针被放在消息队列中偏移量是 0x16 的地方。在 Windows 95 16 位 TOOLHELP.H 文件中，这个数据结构是用 LT_ USER_ PROCESS(0x1D)标志符来标记的。(只有 USER.EXE debug 版本用一个类型标示符来标记块的)。

你可以从为本章编写的 SHOWWND 程序中的 NSGQUEUE.H 文件中，找到 QUEUEPROCESSDATA 结构中的这个已知域。这个域的详细情况如下。(注意每一入口开始的前三项分别为偏移量，类型和名称)：

00h WORD npNext

该域是指向系统中下一个 QUEUEPROCESSDATA 的近指针(相对于 USER 的 DGROUP)。

02h WORD un2

该域指向什么是不知道的。然而在 Windows debug 版本中,给定的是 LT_USER_SUBSYSTEM 的类型。

04h WORD flags

某种 WORD 标志,意义不详。

08h DWORD processId

此处 DWORD 保存与该队列相联的过程 ID

0Eh WORD hQueue

WORD 是一个 hQueue 数值。具体的不太清楚,可能是个指向过程中为线程创建的队列的后指针。

QUEUEMSG 结构

在 Windows 3.1 和更早版本中,消息队列实际包含有传给它的信息。在队列结构尾部一大块区域基本上只是 MSG 结构数组。在队列结构开始双 WORD 是头和尾指针。因为信息是被存入数组中的,因此在任何给定的时间内,队列中有一个最大可存的信息号码。出错的话,此数值是 8 个信息,但它通过调用 SetMessageQueue 用新信息计算法可以向上增加。

Windows 95 完全改变了一个队列消息存储的方式。在 Windows 95 消息队列中,有一个指向结构列表头的近指针,每个消息有一个结构。我称这些结构为 QUEUEMSG。QUEUEMSG 结构是从 16 位 USER DGROUP 分配的。这是令人吃惊的,因为 USER 的 QUEUEMSG 有大量的输入输出,所以在那设置信息结构看来是事与愿违的。偶尔 Windows 95 16 位 TOOLHELP.H 通过 LT_ USER_QMSG(0x1A)引用这些结构。

如果发现一个队列消息不在存入队列尾部的数组中是很难理解的话,考虑下面的 Windows 95 的 SetMessageQueue 函数的代码:

```
SETMESSAGEQUEUE proc
C0ED:   XOR     AX,AX
C0EF:   INC     AX
C0F0:   RETF    3702
```

对于不读汇编程序的人来说,函数简单地返回 1。这就是因为在一个队列中不

存在 MSG 结构数组的缘故。相反，而 Windows 3.1 SetMessageQueue 计算出新队列将是多大(考虑消息号码)然后为队列分配一个新的全局堆块。

QUEUEMSG 结构格式，在 SHOWWND 程序 MSGQUEUE.H 文件中，是用 C 风格形式给出的。QUEUEMSG 域如下(每一入口前三项为偏移量，类型和名字)：

00h　　WORD　　HWND

这个 WORD 是消息将被传送到的窗口句柄(HWND)。

02h　　WORD　　msg

这个 WORD 是消息号。只有消息号的底 16 位被存储。在 Win16 中，这不是问题，因为消息只有 16 位。但对 Win32 应用程序，一个消息应是 DWORD 数值，所以 Win32 程序消息数值顶部 WORD 被丢失了。

04h　　WORD　　wParamLow

对于 Win16 应用程序，这个域是 WPARAM 数值。对于 Win32 程序，该域是 WPARAM 数值低 WORD。

06h　　DWORD　　lParam

该域含有消息的 LPARAM。

0Ah　　DWORD　　messageTime

此处 DWORD 是消息被排进队列的时间。根据 SDK 文件，消息时间是系统启动后的毫秒数。该域数值最终是由 GetMessageTime 函数返回的。作为检索该消息的一部分，GetMessage 和/或者 PeekMessage 拷贝该数值到消息队列的偏移量为 24h 的地方，这里是 GetMessageTime 开始检索的地方。

0Eh　　DWORD　　messagePos

此处 DWORD 是信息形成时光标的 X，Y 座标。该域值最终由 GetMessagePos 函数返回。作为检索该信息的一部分，GetMessage 和/或 PeekMessage 拷贝它到消息队列偏移量为 28h 的地方，这里是 GetMessagePos 开始检索它的地方。

12h　　WORD　　wParamHigh

对 Win32 应用程序，这个 WORD 保存 WPARAM 的高 WORD，对 Win16 程序，这个 WORD 被忽略。

14h　　DWORD　　extraInfo

这个 DWORD 含有有时与一个消息相关联的特殊信息。该域数值最终是由 GetMessageExtraInfo 函数返回的。作为检索消息的一部分，GetMessage 和/或

PeekMessage 把它拷贝到消息队列偏移量为 30h 的地方，GetMessageExtraInfo 正是在这个地方开始检索的。

18h	WORD	nextQueueMsg

这是一个指向表中下一个 QUEUEMSG 结构的近接针(相对于 USER 的 DGROUP)。该域为"0"，表明这是列表尾部。

单一队列系统窗口

Windows NT 采纳的设计概念之一是一个进程不应当对另一个进程产生不利的影响(允许除外)，Windows 3.1 或更早的版本却不遵循这一原则，特别是当其处于窗口系统状态时。在任何时候，Windows 3.1 和以前版本仅有一个激活(active)窗口，一个焦点(focus)窗口和一个捕获(capture)窗口。通过调用 SetFocus，任何应用程序可以从另一个程序中隔离出焦点窗口。同样调用 SetActiveWindow 可以在活动窗口任务下改变窗口。

Windows NT 通过给每个应用程序的激活和捕获的 HWNP 的拷贝来解决这个问题的(实际上这样说过于简单了，但现在也足够了)。把这些系统状态窗口应用程序放在一起，Win32 程序不必考虑其他程序对自己的影响(不利或有利的)。由于有分解的信息系统，每个应用程序系统状态窗口的想法比较好，所以 Windows 95 也采用了(这种情况也被 Win32 API 规定了，但没关系)。

每队消息是被存在从 USER 的 DGROUP 段分配的结构里。(我想 Windows 95 可能把某些东西移出 USER 的 DGROUP，也没增加新的东西)。在消息队列偏移量为 0x10 的地方可以找到指向每队数据域的指针。我称每队消息结构为 PERQUEUEDATA。Windows 95 16 位 TOOLHELP.H 引用此结构为 LT_USER_VWININFO(类型 ID=0x1B)。

偶然地在最后 Windows 95 beta(M8)期间，3 月 27 日的 InfoWorld 以"Win95 β 下了个蛋"为题，刊登了一篇文章，随后由此引起的争论最终在 PERQUEUEDATA 结构结束。在最后 Windows 95 β 时，每个 PERQUEUEDATA 在长度上是几百个字节。结果，启动大量线程时，很快占完了 USER 的 64K DGROUP。随后，Microsoft 重新组织 PERQUEUEDATA 结构，显著地减少了其占用空间，结果争论才结束。

PERQUEUEDATA 的 C 风格结构定义是在 SHOWWND 例子程序中 MSGQUEUE.H 文件中给出的，详细说明如下(注意每个入口前三项分别为偏移量，类型和名字)：

00h	WORD	npNext

这是个指向系统中下一个 PERQUEUEDATA 结构的近指针(相对于 USER 的 DGROUP)，显然 PERQUEUEDATA 被存在一个列表里。

06h　　WORD　　npQMsg

这是个指向一个 QUEUEMSG 结构的近指针(相对于 USER 的 DGROUP)。(QUEUEMSG 在前面已有描述)。

14h　　WORD　　somehQueue1

这个 WORD 是个消息队列句柄。目前其确切含义不详。

16h　　WORD　　somehQueue2

这个 WORD 是另一个消息队列句柄,其意义目前不详。

18h　　DWORD　　hWndCapture

这个 DWORD 保存一个指向该队列的当前捕获窗口的 32 位指针(相对于 USER 的 DGROUP)。

1Ch　　DWORD　　hWndFocus

这个 DWORD 保存一个指向该队列当前焦点窗口的 32 位指针(相对于 USER 的 DGROUP)。

20h　　DWORD　　hWndActive

这个 DWORD 保存着一个 32 位指向该队列当前激活窗口的近指针(相对于 USER 的 DGROUP)。

Windows 95 中(H)WND 结构的改变

在 Windows 95 中 WND 结构可以说是用得最广的系统数据结构。对应于系统中的每个窗口(可见的和不可见的),都有一个相对应的 WND 结构。在 Windows 3.1 中,HWND 是一个指向 USER 的 DGROUP 中 WND 结构的近指针。如前面我所说的,Windows 95 HWND 是指向 WND 结构 32 位 USER32 相对指针数组的偏移量。

由于每个 WND 结构包含有一个指向本身的父窗口、同属窗口及它的首要子窗口的指针,所以你很容易看到窗口被存放在三级结构中。图 4-3 给出了三级结构图,并简要地描述了每个"级"。WND 三级结构的根是桌面窗口。桌面窗口下面第一级是 WS_OVERLAPPED 或 WS_ POPUP。大多数开发者认为它是"顶级"或"主要"窗口。再下面是 WS_CHILD。图中典型的子窗口是一个控制窗口(如一个按钮)。由于窗口分级,你可以从桌面窗口开始,数出系统中所有窗口,就像 SHOWWND.C 程序显示的那样(请看 SHOWWND 程序部分)。

尽管这个事实一般不太了解,然而窗口之一命令是由它们的分级结构中的相对位置决定的。在给定一组同属窗口中,列表中的第一个窗口在命令中是最高级

的，表中第二个窗口次之，依次类推。例如，所有顶级窗口(WS_OVERLAPPED 和 WS_POPUP)之间是同属的，都是桌面窗口的子窗口。桌面窗口的第一个子窗口(即 WS_OVERLAPPED 或 WS_POPUP 窗口)在 Z 命令中是最高级的。

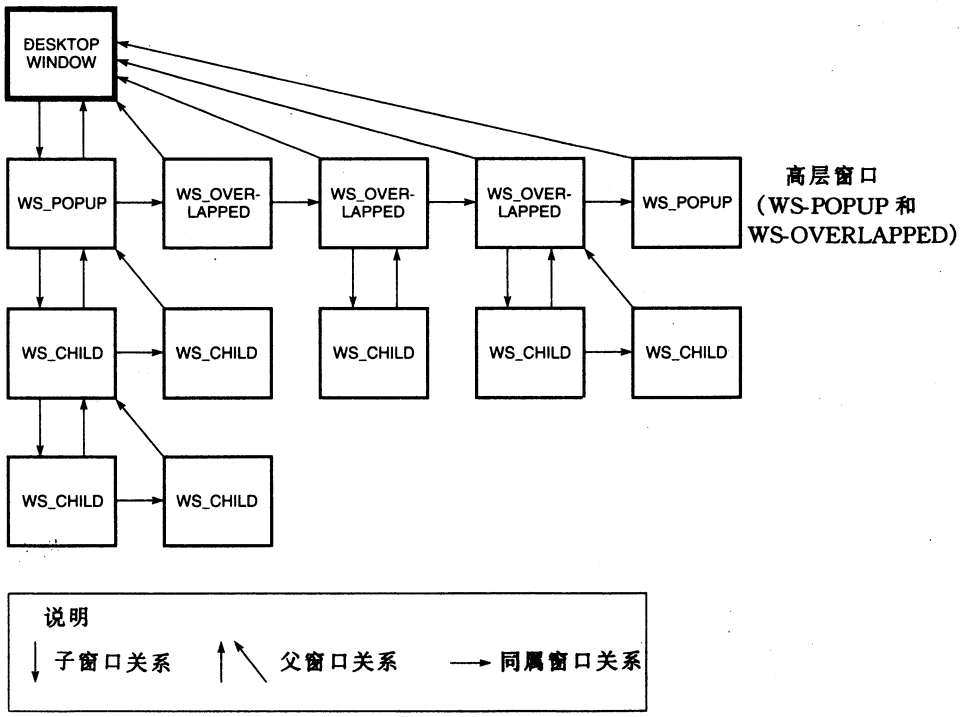


图 4-3　从 WND 结构自由层可以看出是从桌面窗口开始的及系统中的所有其他窗口

WND 结构详述

尽管在 Windows 95 中 WND 结构有些变化(相对于 Windows 3.1)，但其改变并不很大。对于大多数部分，域的顺序没有改变(尽管某些域的大小有些变化)。在 Windows 95 中的 WND 结构增加了新的域。主要的新域是 WORD，它保存窗口的 16 位 HWND 数值。这个 WORD 是容易被 16 位 HWND 或 32 位 USER 的 DGROUP 相对指针引用的。

SHOWWND 中 HWND32.H 文件对 Windows 95 WND 结构有个 C 风格的说明。详细如下(注意每项开始的前三项分别为偏移量，类型和名字)：

```
00h        struct_ WND32 *          hWndNext
```

此处 DWORD 是一个指向同属窗口的 32 位指针(相对于 USER 的 DGROUP)。同属窗口是在同级，即共有同一父窗口。你可以通过调用带有 GW_HWNDNEXT 参数的 GetWindow 得到同属窗口的 16 位 HWND。

04h　　struct_WND32 *　　hWndChild

这个 DWORD 是一个指向该窗口的第一个子窗口的 32 位指针(相对于 USER 的 DGROUP)。可以通过调用带有 GW_CHILD 参数的 GetWindow 得到第一个子窗口的 16 位 HWND。通过调用 GetWindow(GW_HWNDNEXT),可以得到每个后续窗口的子窗口。

08h　　struct_WND32 *　　hWndParent

这里的 DWORD 是一个指向该窗口的父窗口的 32 位指针(相对于 USER 的 DGROUP)。通过调用 GetParent 你可以得到父窗口的 16 位 HWND。唯一没有父窗口的是桌面窗口。

0Ch　　struct_WND32 *　　hWndOwner

此处的 DWORD 是一个指向归属窗口的 32 位指针(相对于 USER 的 DGROUP)。归属窗口是一个接收通告消息的窗口(例如 BN_CLICKED 信息)。对于 WS_OVERLAPPED 和 WS_POPUP 窗口,它们的归属窗口和父窗口常常是相同的,但并不是一回事。对于 WS_CHILD 窗口,父窗口总是作为归属窗口即它接收通告消息)。通过调用 GetWindow(GW_ OWNER),你可以得到窗口的归属窗口 16 位 HWND。

10h　　RECTS　　rectWindow

这是一个定义窗口(包括非用户区域)边界的 16 位 RECT 结构(四小段)。

18h　　RECTS　　rectClient

这是个定义窗口的用户域边界的 16 位 RECT 结构。用户域也是窗口的一部分,从 BeginPaint 或者 GetDC 得到的设置范围,应用程序被允许调入。

20h　　WORD　　hQueue

这个域包含有为特殊窗口处理消息队列的 16 位全局堆句柄。这表明在 Win32 中,窗口被限定为单队的,因此也是单线程的,所以有一个 Win32 GetWindowThreadProcessId 函数。

22h　　WORD　　hrgnUpdate

如果窗口某部分需要刷新,该域保存描述需要刷新部分的 HRGN。刷新部分是 GDI 数据结构,并且被存储在 32 位堆中。

24h　　WORD　　wndClass

这个 WORD 是一个指向 USER_DGROUP_WNDCLASS 结构的近指针(相对于 USER 的 DGROUP)。一个 USER_DGROUP_WNDCLASS 结构只不过是

USER 需要经常访问的 window-class-related 信息的最好单位。不太经常访问的类信息被存储在 32 位堆的另一结构中。我们将在“Windows 95 窗口类”部分看到 USER_DGROUP_WNDCLASS 和其他结构的格式。总的来说，WND 结构的这个域给出了窗口类的类型。

26h WORD hInstance

大多数情况下，这个 WORD 是创建窗口的应用程序所需的 16 位 hInstance (DGROUP)。然而，编辑控制需要非常大的缓冲区(达到 64K)，该域含有编辑控制 WNDPROC 所用的 DS 数值。在调用一个窗口过程之前，USER 把该域装入 AX 寄存器。在某些被输出的 Win16 函数程序中，代码所用的 DS 值在 AX 中。正常情况下，程序要用的 DS 是 DGROUP 段的 DS，在编辑控制有大量文本时，需要单独的段。

28h WNDPROC lpfnWndProc

这个 DWORD 是与窗口有关的窗口过程地址，看起来总是远 16:16 地址。如果窗口声明窗口过程是在 Win32 中，该域是一个指向 16:16 转变成 Win32 窗口过程的指针。

2Ch DWORD dwFlags

这个 DWORD 是窗口内部状态的特殊标志。每位的意义没有注明。

30h DWORD dwStyleFlags

这个 DWORD 保存 16 位 WINDOWS. H 和 32 位 WINUSER. H 给出的 WS_XXX 风格的标志。

34h DWORD dwExStyleFlags

这个 DWORD 是 16 位 WINDOW. H 和 32 位 WINUSER. H 给出的 WS_EX_XXX 扩展风格的标志。Windows 95 增加了几个新的扩展风格的标志，这将在“其他窗口系统改变(或缺少)”部分详述。

38h DWORD moreFlages

这是个被用作标志的域，其意义不清。

3Ch HANDLE ctrlID(hMenu)

对于顶级窗口(WS_OVERLAPPED 或 WS_POPUP)，该域为窗口保留 hMenu。它的数值是由 GetMenu 函数给出的。对于 WS_CHILD 窗口，该域保留的是控制 ID。在框图中，控制部分你最熟悉的可能是控制 ID 了。如果该窗口是个 WS_CHILD 窗口，该域的数值可以通过调用 GetDlgCtrlId 得到。

40h　　DWORD　　Some32BitHandle

对窗口文本，这个 DWORD 是个 32 位句柄，与 HWND 相似，但其所使用的堆即不是窗口堆也不是菜单堆。

42h　　WORD　　scrollBar

该处 WORD 保留的是与窗口滚动条属性有关的信息。

44h　　WORD　　properties

这个 WORD 是一个特性列表中第一个窗口特性的句柄。对于一个窗口，特性是真正的最小单位，允许你把它和命名的 16 位数值结合在一起。详细内容请看 SDK 说明中 GetProp 和 SetProp 函数部分。

46h　　WORD　　hWnd16

这个 WORD 是 WND 结构中关键域之一。它为窗口保存的是 16 位的 HWND 数值。当 USER 码有个指向一个 WND 结构的 32 指针时，它以这个域的内容，返回一个真正的 16 位 HWND。这就允许 USER 内部传送指向 WND 结构的 32 位指针。而不用再传送相应的 16 位 HWND。只要需要，任何时候都可以在 WND 结构中查找 HWND。

48h　　struct_WND32*　　lastActive

这个 DWORD 保存的是指向与此窗口相联的最后激活弹出窗口的 32 位指针（相对于 USER 的 DGROUP）。GetLastActivePopup 函数从这个数值可得到指向 WND 结构的指针，然后返回 16 位 HWND，这个 16 位 HWND 被存储在结构中偏移量为 46h 的地方。

4Ch　　HANDLE　　hMenuSystem

这个域是个与该窗口相联的系统菜单的句柄。详细情况请看 SDK 说明中的 GetSystemMenu 函数。

56h　　WORD　　classAtom

这个 WORD 保存的是与类名字相联的元。它可以是一个一般的元（即＞0xC000），或者是预先定义窗口类的类型：

```
0x8000  (PopupMenu)
0x8001  (Desktop)
0x8002  (Dialog)
0x8003  (WinSwitch)     // The ALT-TAB window.
0x8004  (IconTitle)     // In Win 3.X, the title window below an icon.
0x002A  ???             // The class associated with MMTASK.TSK.
```

这个域(偏移量为 56h)通常与由 wndClass 指针(WND 类的偏移量 24h)指向的结构中偏移量为 2 的域是一样的。

58h	DWORD	alternatePID
5Ch	DWORD	alternateTID

这两个域从表面上看不是进程 ID 或者线程 ID。然而,GetWindowThreadProcessId 码却表明这些域能够存放一个 PID 和一个 TID。

对于 Windows 和 WND 结构最后需要注意的是,当我告诉人们创建一个窗口并不需要占用 USER 的 64k DGROUP 内存时,人们往往会感到惊奇。如果你正在从现存的类创建一个窗口,你所需的分配给的数据就是 WND 结构本身。因为 WND 结构是来自一个 32 位堆,因此创建窗口时,绝对对 USER 的 16 位 DGROUP 没有影响。为了证明这一点,我编写了一个程序,能够创建几千个窗口,并在整个创建过程中许多点对 USER 的 DGROUP 自由空间进行检查。在所有情况下,在这整个进程中,USER 的 DGROUP 中的存储没有被分配。

其他窗口系统的变化(或缺少)

对某些用户,对 Windows 95 最大的失望之一就是标准窗口中最大 64K 文本的限制并未消失。根据 Windows 95 设计目标,这没有什么惊讶的,窗口处理和显示文件的代码是 16 位的。大量的 16 位 USER 不得不被转换成突破 64K 限的 32 位代码。这里面就有个大小和兼容性问题,不知为什么 Windows 95 编程人员没有解决此问题。另一方面,Windows NT 却有一个全 32 位的 USER 和 GDI,并没有此限制。这种 64K 窗口文本的限制,是大多数终端用户能够看出的 NT 和 Windows 95 差别之一。

一个确实的事实是 Windows 95 定义了许多新风格的窗口给 Win32 应用程序(它使 Windows 95 外观看起来更简洁)。WINUSER.H 中新风格如下:

```
#define WS_EX_MDICHILD          0x00000040L
#define WS_EX_TOOLWINDOW        0x00000080L
#define WS_EX_WINDOWEDGE        0x00000100L
#define WS_EX_CLIENTEDGE        0x00000200L
#define WS_EX_CONTEXTHELP       0x00000400L
#define WS_EX_RIGHT             0x00001000L
#define WS_EX_LEFT              0x00000000L    (The default in Win 3.1)
#define WS_EX_RTLREADING        0x00002000L
#define WS_EX_LTRREADING        0x00000000L    (The default in Win 3.1)
#define WS_EX_LEFTSCROLLBAR     0x00004000L
#define WS_EX_RIGHTSCROLLBAR    0x00000000L    (The default in Win 3.1)
#define WS_EX_CONTROLPARENT     0x00010000L
#define WS_EX_STATICEDGE        0x00020000L
#define WS_EX_APPWINDOW         0x00040000L
```

这里我引用SDK说明来解释这些新风格的功能，并不是烦扰你。然而，这里确实有些事情很有趣，可能不会立即显示出来。如果深入研究16位WINDOWS.H，不会找到任何被列出的这些新的WS_EX_XXX风格。这些新的WS_EX_XXX风格似乎只在32位WINUSER.H文件里。现在我强调一下，几乎所有的USER子系统函数(包括窗口系统)都是在16位USER.EXE中实现的。有些事情需要记住，如果16位USER.EXE实现窗口系统，那么16位USER.EXE必须实现这些风格——假定只为32位的应用程序的。为什么16位应用程序不能使用同一新的WS_EX_XXX风格？结果是没有合理的理由。事实上，在某些非正式试验中，结果我发现某些16位Windows 95实用程使用了这些新的被扩展的风格。

Windows 95 窗口类的改变

在探讨Windows 95窗口类管理的变化之前，先来简要讨论一下窗口类。窗口类是被用于创建窗口属性的集合。这些属性包括一些项，如窗口过程回调地址，窗口风格位，辅助存储窗口所需的附加数据字节的号码等等。理论上讲USER可以不使用类，然而在每次创建窗口时，你都得对这些类进行说明。特别是在窗口中程序创建许多按钮时，更是如此。

窗口类是作为窗口特殊例子被建立时的模板的。在窗口建立以后，被拷贝到WND结构的某些类的属性可以修改，这样的典型例子便是窗口过程地址。开始时所有从同一类创建的所有窗口均有同一窗口过程地址。进一步程序可以用SetWindowLong来改变一个特殊窗口的窗口进程，这就是子类。

当窗口启动时，它建立了含有12个标准类的小集合：

Button	ListBox
ComboBox	MDIClient
ComboLBox	PopupMenu
Desktop	ScrollBar
Dialog	Static
Edit	WinSwitch

这些类中的绝大多数会被各种应用程序反复用到，所以把它们定义为普通系统类。调用RegisterClass可以建立附加的应用类。不管你使用的是标准的系统类还是你自己建立的类，你都必须向CreateWindow或者CreatWindowEx函数传送一个类标识符(典型的是它的名字)。

在Windows 3.1或更早的版本中，USER把所有系统中登记的窗口类放在列表里，甚至还有TOOLHELP函数(ClassFirst和ClassNext)来枚举所有的登记类。在Windows 95中，所有登记类的列表没有了，但仍有你可以通过ClassFirst和ClassNext得到信息的小部分类，这部分仅是标准的系统类(例如按钮，表框等)。在USER启动阶段登记的是标准系统类(参见第一章有关部分)。

如 Windows 3.x 和更早版本一样，Windows 95 USER 仍然从 16 位堆中给类分配空间(每个新类都占有一小部分可用的系统资源)。在 USER debug 版本中，给一个类结构分配的存储器事先由 LT_USER_CLASS(1)标记。

作为设计 Win32 释放进程原则的一部分就是尽可能的减少对其他进程的依赖，现在每个 Windows 95 32 位进程都拥有自己的类表。这个自己的类表包括由系统 DLL 登记的类，如 COMCTL32.DLL。每次当一个新进程调用 COMCTL32.DLL 时，大致有 12 个新类被登记。这些类是由 COMCTL32.DLL 提供的应用程序私有的拷贝。如果你认为这些所有的应用程序拥有的类在 USER 的 64K 堆中，可以很快地吸取空间，你是正确的。如果给出应用程序私有类表的话，可以枚举出他们，不幸的是，16 位和 32 位 TOOLHELP API 都没有为浏览进程私有类表提供方法。直到现在我能够找到私有类表的唯一方法是对系统中所有窗口进行枚举和在 WND 结构中搜索类指针。类指针是在每个 WND 结构偏移量为 24h 的地方。使用 SHOWWND 程序，你可以手动找到进程类表(即你先找到类表的头，然后再往向下移动)。

Windows 95 窗口类结构 C 风格定义是由 SHOWWND 程序中 WNDCLASS.H 文件给出的(参见下一部分)。这个结构我定义为 USER_DGROUP_WNDCLASS。

这个结构包含一个 16 位 USER 经常访问的最小域。窗口类域不太经常使用的部分被放在一个 32 位堆的单独结构中。USER_DGROUP_WNDCLASS 域如下：(注意入口前三项分别为偏移量、类型和名字)：

```
00h     DWORD                   lpIntWndClass
```

这是个进入窗口堆的远(16:16)指针。它指向下面要介绍的 INTWNDCLASS 结构。INTWNDCLASS 基本上是 USER 暂时不需要访问的类信息。

```
04h     WORD                    hcNext
```

这里 WORD 里是一个指向下个类的近指针(在 USER 的 DGROUP 中)。这下个类可以是个系登记的类，或应用程序私有类。

```
06h     ATOM                    classNameAtom
```

这个 DWORD 保存的是描述类名的元。它可以是正常元(如>0xC000)或是个标准类元(0x8000，0x8001 等等)。在辅助盘上 SHOWWND.C 源文件中的 GetClassNameFromAtom 函数表明怎样把这些元译成类名。

```
08h     DWORD                   style
```

这个 DWORD 为类保存 CS_XXX 风格(如 CS_VREDRAW)。这个域是从 Win3.1 的 WORD 加宽到 DWORD。

把 USER_DGROUP_WNDCLASS 结构中的域加起来，你会发现每个类在 USER 的 DGROUP 中占有 0x0C 个字节。在 Windows 3.1 中，关于 WNDCLASS

所有的信息都被存储在 USER 的 DGROUP 中。为了释放增加的 USER 的 DGROUP 存储器，Microsoft 把窗口类的大多数域放入了一个单独的 32 位堆。堆中 USER_DGROUP_WNDCLASS 第一个域含有个远指针，该指针指向我称之为一个 INTWNDCLASS(INTernal WNDCLASS)结构。INTWNDCLASS 与你传送给 RegisterClass 函数的 WNDCLASS 结构相似，但不相同。INTWNDCLASS 的格式也是在 WNDCLASS.H 头文件给出的，详细情况如下：

00h　　WORD　　cClsWnds

此域里是该类当前窗口的序号。

04h　　DWORD　　lpfnWndProc

这个 WORD 为该类窗口保存的是窗口过程地址。可以使用 SetClassLong 来改变此域。

08h　　WORD　　cbClsExtra

此处 WORD 保存的是在 INTWNDCLASS 结构尾部分配给的附加字节的号码。应用程序可以使用这些字节来存储一些特殊数据。这些特殊字节是被 SetClassWord/Long 和 GetClassWord/Long 访问的。

0Ah　　WORD　　hModule

这个 WORD 保存的是登记类的 16 位 HMODULE。注意它与 SDK 说明不同，SDK 引用它是作为 HINSTANCE。USER 用 16 位 HMODULE 开始是因为 USER 在正常情况下，用第七章描述的 16 位 GetExePtr 例程把 HINSTANCE 转换成 HMODULE。

0Ch　　WORD　　hIcon

这是与类窗口相联的图标。

1Eh　　WORD　　hCursor

当鼠标在此类窗口上时，这是被使用的光标。

10h　　WORD　　hBrBackground

当刷新窗口背景时，这是个刷子。

12h　　DWORD　　lpszMenuName

这是此类窗口使用的菜单名字。此域通常为 0，偶尔也含有一个 16:16 有效指针。

16h　　DWORD　　hIconSm

这是一个小的图标，是与该类窗口相关的。如果它非零，在 Explorer 任务在

时，可用来代表窗口。

18h	WORD	cbWndExtra

此处的 WORD 保存的是附加字节的号码，此附加字节是在此类创建的窗口尾部分配给的。应用程序可以使用这些字节存储每个窗口的数据。这些附加字节可由 SetWindowWord/Long 和 GetWindowWord/Long 访问。

SHOWWND 程序

我写 SHOWWND 程序是为了说明绝大多数数据结构。SHOWWND 的核心是窗口层次。你可以变换系统的每个窗口来显示窗口各种域的详细情况。如果某个域与另外一个重要的数据结构相关，你也可以转到那个结构上去。用这种方法，辅助盘上的 SHOWWND 程序显示出了 WND 结构、窗口类和消息队列。因为所有这三类结构包含有链向（指向）相同结构类型的链（指针），你可以轻松地浏览窗口表、类表和消息队列表。

为了证明我前面描述的数据结构的真实性，SHOWWND 尽可能少地使用 USER 函数，如果许可的话，SHOWWND 直接访问数据结构。如 SHOWWND 可以调用 EnumWindows 和 EnumChildWindows 来显示窗口层次。但这并不证明 WND 结构与我叙述的一样。当然仅仅围绕数据结构并不是程序实践的好方法，应尽可能避免。然而为了显示的目的，这是证明我叙述是正确的唯一方法。

正如本书中的其它程序一样，SHOWWND 有二个表框，如图 4-4，左边的表框是系统中当前窗口嵌套层的显示。在任何时候，你可以通过刷新按钮刷新列表。右边表框是个详细面板，当你在左边面板选择一个窗口后，右边表框来显示被选的窗口 WND 结构的内容。

如果你仔细检看一直右边表框，有几行前面有＋号（加号）。＋表示此行可以激活详细幕显示其详细资料。从 WND 幕上，你可以转向另一个 WND，窗口类或窗口队列。从类幕上，你可以在表中沿 hcNext 指针到下一个类。消息队列幕的工作方式与此相似。

SHOWWND.C 代码的绝大多数是相当易懂的，这里就不列出了。然而这里需要指出的是 SHOWWND 是个 Win32 程序。如你所知道的，Win32 程序可以事先被其他线程清除，结果 SHOWWND 可以留在窗口层中，其他线程可以进入和改变窗口层。尽管这种情况可能是很少见的，但确实可能会发生。

为了阻止这种情况的发生，SHOWWND 在整个窗口层次中要调用和占有 Win16Mutex。SHOWWND.C 是通过调用三个未注明的函数来完成的：GetpWin16Lock、EnterSyslevel 和 LeaveSyslevel。GetpWin16Lock 函数把 Win16Mutex（实际上在 KRNL386.EXE 内）的地址放入一个 DWORD 内。通过传送此地址到 EnterSysLevel，程序可以获得 Win16Mutex 和释放 mutex 及 LeaveSyslevel。这种技术有点与《Unauthorized Windows 95》的 W16LOCK 程序相似。其不同之处，W16LOCK 是使用这些函数来表明系统可以被 Win32 应用程序锁死，而

SHOWWND 使用它们作为句柄线程同步释放。

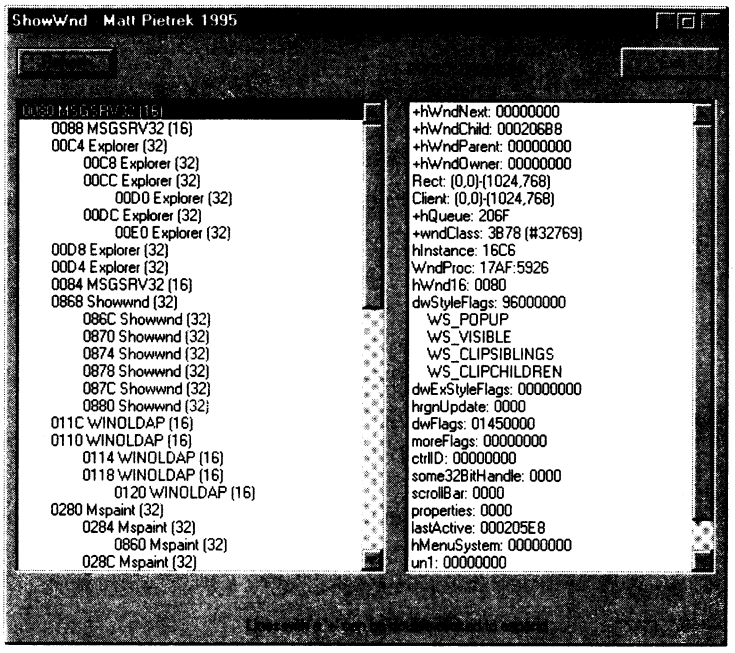


图 4-4　SHOWWND 程序有两个对话框，一个显示当前窗口的层次结构，一个显示每个窗口的内容

选择 16 位 USER. EXE 函数的伪码

现在我们已研究过了一些关键 16 位 USER 数据结构和 USER 使用的新的 Win32 堆，现在我们来看 USER. EXE 中一些函数的伪码。下面部分都是实用的，因为我要让你看到这些数据结构和概念是怎样被应用的。

USER. EXE 的 IsWindow 和 IsWindow16 函数

IsWindow 函数带有一个 16 位 HWND 作为参数并检查其是否是真正有效的 HWND。在 debug USER. EXE 中，IsWindow 码仅是记录码，如果一个确定 USER 跟踪模式标志(未说明)起作用的话，它向 debug 口发出函数的名字。在 IsWindow 转为 IsWindow16 码后，才真正校验 HWND。

IsWindow16 快速地排除掉无效的 HWND 值。如前所述，在 Windows 95 中，HWND 总是四位的，所以 IsWindow16 返回的 FALSE 是它的最低两位。若该数小于 0x80，Is Window 16 立即清掉它，为什么是 0x80?因为句柄表域前 0x80 字节(在 USER 的 DGROUP 基地址+0x10000)被用来存储与 32 位窗口堆相关的其他信息。句柄表域第一个可用的窗口指针是在偏移量为 0x80 的地方，这个地方看起来似乎总被桌面窗口占用。(这是正常的，因为桌面窗口是第一个被创建的窗口)。

若 HWND 数值太大，IsWindow16 也要清掉它。在句柄表域偏移量为 0x70 的

地方是一个 DWORD，它里面存放的是使用的最大句柄表偏移量。如被传送到 IsWindow 的 HWND 值大于此数，是一个无效的 HWND，所以 IsWindow16 返回 FALSE。

IsWindow16 最后部分是使用 16 位 HWND 数值去查找指向 WND 结构的 32 位指针。（记住，指向 WND 结构的 32 位指针是相对于 USER 的 DGROUP 的，它不是扩展的 32 位指针）。IsWindow16 对 HWND 非引用指针做两次检查，首先指针必须大于 0x10000。（IsWindow USER32 版本中，测试是 0x20000，这可能更精确）。第二，HWND 非引用指针必须为非零。如果此二条件均被满足，IsWindow16 返回 TRUE，表明 16 位 HWND 是有效的。

IsWindow 伪码

```
// In 16-bit USER.EXE
// Parameters:
//      HWND    hWnd    // The 16-bit version.

    Push DS

    Load DS with USER's DGROUP

    Grab UserTraceFlags WORD from USER's DGROUP

    restore USER's DGROUP

    if ( UserTraceFlags & 0x2000 )
        _DebugOutput( DBF_USER, "IsWindow" );

    // Execution falls through to IsWindow16...

IsWindow16 proc
// Parameters:
//      HWND    hWnd    // The 16-bit version.
// Locals:
//      PWND32  pWnd32  // 32-bit USER DGROUP relative pointer to HWND32.
//      PVOID   USER_dgroup_base    // Base address of USER's DGROUP.

    // Pops return address and HWND off the stack, then pushes them
    // back on. Supposedly saves space on stack frames.

    if ( hWnd & 3 )     // HWND16s must be a multiple of 4.
        return 0;

    if ( hWnd < 0x80 )  // HWND16s are always >= 0x80.
        return 0;
    // At offset 0x10070 in the USER DGROUP seg is a DWORD with the
    // maximum HWND value.
    if ( hWnd > *(PDWORD)(USER_dgroup_base + 0x10070) )
        return 0;
    // Use the HWND as an offset into the handle table area at
```

```
// offset 0x10000 in USER's DGROUP. Grab the pointer stored there.
pWnd32 = *(PDWORD) (USER_dgroup_base + 0x10000 + hWnd);

if ( pWnd32 <= 0x10000 )    // All HWND structs are above 0x10000.
    return 0;               // Actually, they're above 0x20000, but...

if ( pWnd32 )               // if the HWND ptr table contains a nonzero
    return TRUE;            // entry, we'll say it's a valid HWND.
```

GetCapture、GetFocus 和 GetActiveWindow 函数

如我前面提到的，Windows 95 的捕获、焦点、激活窗口被存储在单一队列底部。不像 Windows 3.x，GetCapture，GetFocus 和 GetActiveWindow 不仅仅是与 USER 的 DGROUP 段相关联数值的收集器。另一方面，在 PERQUEUEDATA 结构（前面讨论到的）中三个 HWND（实际上是 USER 32 位指针）是依次存储的。这样搜索三个 HWND 的代码有部分可以是相同的。

这三个函数每个都向一个寄存器（被称为伪码中的 PerQueueOffset）装入所希望窗口的指针（在 PERQUEUEDATA 结构中）的偏移量。函数然后跳转到公共点（伪码中被称为 Get_XXX_common）。公共代码首先调用 KRNL386 得到指向当前线程消息队列的指针。队列内是个指向 PERQUEUEDATA 结构的指针。代码把该指针与 GetCapture、GetFocus 或 GetActiveWindow 函数设定的偏移量相加，得到的是个指向所希望的窗口的指针，公共码其它部分所做的工作是进到 WND 结构和在偏移量为 46h 的地方提取 16 位 HWND 的数值，是通过调用 HWnd32ToHWnd16 实现的。

GetCapture、GetFocus 和 GetActiveWindow 伪码

```
// In 16-bit USER.EXE
// Locals:
//  PMSGQUEUE   pQueue;
//  WORD        perQueueOffset
//  BOOL        flag
//  PWND32      pWnd

GetCapture proc
    perQueueOffset = 0x0018 // Offset of the capture WND in the PERQUEUEDATA.
    flag = FALSE
    goto Get_XXX_common

GetFocus proc
    perQueueOffset = 0x001C // Offset of the focus WND in the PERQUEUEDATA.
    flag = FALSE
    goto Get_XXX_common
GetActiveWindow proc
```

```
    perQueueOffset = 0x0020 // Offset of the active WND in the PERQUEUEDATA.
    flag = TRUE

Get_XXX_common:

    pQueue = GetCurrentThreadQueue();   // KERNEL.625
    if ( !pQueue )
        INT 3;      // Oops! No queue. Break into the debugger.

    if ( pQueue->npPerQueue == 0 )
        INT 3;      // Oops! No per-queue data. Break into the debugger.

    // Using the perQueueOffset value (in the BX register), index into the
    // per-queue area and extract a USER relative pointer to the desired WND.
    pWnd = *(PWND32 *)(pQueue->npPerQueue + perQueueOffset);

    if ( !pWnd && flag )    // If pWnd is 0, but "flag" is set (which
    {                       // only happens for GetActiveWindow)...

        // Try a second approach to getting the active window. If
        // the conditions are right, try calling GetForegroundWindow.
        // npCurrentPerQueueData is a USER.EXE global variable.
        if ( pQueue->npPerQueue == npCurrentPerQueueData )
            return GetForegroundWindow();
    }

    // Convert from the 32-bit HWND form to the 16-bit form, and return it.
    return HWnd32ToHWnd16( pWnd );
```

GetWindowThreadProcessId 和 IGetWindowThreadProcessId 函数

GetWindowThreadProcessID 函数是面向 Win32 API 的。(Windows 3.x 中与其最相似的是 GetWindowTask)。尽管 GetWindowThreadProcessId 函数是由 32 位 USER32. DLL 输出的,但它还是在 16 位 USER. EXE 中实现的。GetWindowThreadProcessId 函数基本上仅是个参数校验层。真正的工作是在 IGetWindowThreadProcessId 函数里。然而,在调用 GetWindowThreadProcessId 函数之前,代码首先把 16 位 HWND 转换成 USER32 相关的 32 位指针,并把它传送过去。

IGetWindowThreadProcessId 不得不从两个不同的地方提取进程 ID 和线程 ID。与窗口相关联的线程 ID 被存储在线程消息队列里。因为队列是每线程(不是每进程),进程 ID 设在消息队列里,被放在前面我讨论过的 QUEUEPROCESSDATA 结构里。IGetWindowThreadProcessId 使用消息队列得到一个指向 QUEUEPROCESSDATA 数据的指针,从该结构提取进程 ID。

IGetWindowThreadProcessId 代码似乎有点奇怪。如果在 QUEUEPROCESSDATA 结构中设置一些标志,显然窗口的线程 ID 和进程 ID 应该被存在 WND 结构自己的后部,但我却从来没有发现这个情况。

GetWindowThreadProcessID 伪码

```
// In USER.EXE (believe it or not)
// Parameters:
//      HWND    hWnd            // 16-bit version
//      LPDWORD lpdwProcessId   // Pointer at which to store the process ID.
// Locals:
//      PWND32  pWnd32;

    pWnd32 = HWnd16toHWnd32( hWnd ); // Convert the 16-bit HWND value into
                                     // the 32-bit pointer version.

    // Verify that a valid pointer to at least 4 bytes was passed.
    VerifyPtr( lpdwProcessId, sizeof(DWORD) )

    return IGetWindowThreadProcessId( pWnd32, lpdwProcessId );
```

IGetWindowThreadProcessId 伪码

```
// Parameters:
//      PWND32  pWnd;
//      LPDWORD lpdwProcessId
// Locals
//      LPMSGQUEUE lpMsgQueue;
//      DWORD   threadId;

    lpMsgQueue = MAKELP( pWnd->hQueue, 0 ); // Get a pointer to the window's
                                            // message queue.
    if ( UserTraceFlags & 0x00042000 )
        _DebugOutput( DBF_USER, "GetWindowThreadProcessId" );

    if ( lpMsgQueue->npProcess->flags & 2 ) // This is rarely the case.
    {
        processId = pWnd->alternatePID; // Grab the PID/TID from the WND
        threadId = pWnd->alternateTID;  // struct.
    }
    else    // Execution most often comes through here.
    {
        processId = lpMsgQueue->npProcess->processId;
        threadId = lpMsgQueue->threadId;
    }

    if ( SELECTOROF(lpdwProcessId) )
        *lpdwProcessId = processId;

    return threadId;
```

DesktopWndProc 函数

当决定本书要有哪些函数值得讨论时，我一下子就想到了 DesktopWndProc 函数。这有两个原因，一是此函数相对比较简单，我要显示的是工作系统提供的窗

口进程；二是 DesktopWndProc 里包含有前面我提到的能够释放系统的自由资源。

首先要注意的是 DesktopWndProc 是准 32 位的 WNDPROC，即 hWnd 和 msg 域是 16 位的，但 WPARAM 却是 32 位的（像 Win32 程序中的 WNDPROC）。另一个值得注意的事情是函数立即把 16 位 HWND 转换成一个 USER32 相关的 32 位指针。它用这个 32 位指针对 WND 结构进行访问。在此方面，所有其它标准系统的 WNDPROC 工作是一样的（即它们用 USER 的 DGROUP 相关的 32 位指针）。

DesktopWndProc 码的核心是个开关语句。（Windows 95 编程人员仍然没有使用 MFC 开关窗口）。DesktopWndProc 处理的窗口信息如下：

- WM_USER 信息：WM_USER 信息是 DesktopWndProc 处理的不确定信息。当桌面第一次收到此信息时（只是第一次），它调用 GetFreeSystemResources 得到在 USER 和 GDI 堆中的自由百分率。然而调用 GetFreeSystemResources 使它们返回与自由百分率相关的数值。谁传送的 WM_USER 信息到桌面？是在 Explorer 完成初始化后，Explorer 本身处理的。WM_USER 信息的想法显然是要建立个基线系统资源用法，与随后调用的 GetFreeSystemResources 可以进行比较。这听起来是很有道理的，与 Windows 3.1 相比是个很大的变化。若 Microsoft 向初学者介绍这种变化，将是个好想法。看起来典型系统的自由系统资源号码增加了，其实变化并不很大。
- WM_ERASEBKGND 信息：这个信息擦除背景和验证指定的矩形。除此之外，没有什么其他的。
- WM_CANCELMODE 信息：如果不存在系统模式窗口，此处理转向错误处理。
- WM_NCCREATE 信息：此信息似乎主要被用于作检查。代码检查要确认无其它类桌面窗口。它也验证桌面窗口有没有父窗口。
- WM_LBUTTONDBLCLK 信息：此信息处理程序把被处理的信息改变为带有 SC_TASKLIST 的 WM_SYSCOMMAND 信息，作为 WPARAM 的高 WORD。在 Windows 3.1 中，在桌面上连续按两下鼠标，能够激活任务管理器。在 Windows 95 中，当 DefWindowProc 接收到 SC_TASKLIST 命令时，它调用一个壳，激活 Explorer 启动菜单。
- WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED 信息：这两个函数调用 USER 中某个函数（你可能已猜到了）与调色板做某种事情。任何通过 DesktopWndProc 的信息和没有被上述处理程序处理的，转向开关语句，调用 DesktopWndProc。（仅 DesktopWndProc 的内部工作情况就可能写一本书了）。

DesktopWndProc 伪码

```
// In 16-bit USER.EXE
// Parameters:
//      HWND    hWnd
//      UINT    msg
//      WPARAM  wParam      // 32 bits, not 16.
//      LPARAM  lParam
// Locals:
```

```
//      PWND32  pWnd32      // 32-bit pointer, relative to USER DGROUP.

    pWnd32 = HWnd16ToHWnd32( hWnd )

    if ( UserTraceFlags & 0x4 )
        _DebugOutput( DBF_USER, "DesktopWndProc" );

    switch ( msg )
    {
        case WM_ERASEBKGND:
            // Erase the desktop. The function calls:
            // FILLRECT, GETCLIPBOX, GETDCORG, GETTEXTEXTENT, LOADSTRING
            // LSTRCATN, LSTRLEN, OFFSETRECT, SETBKMODE, SETBRUSHORG,
            // SETTEXTCOLOR, SETVIEWPORTORG, and TEXTOUT.
            SomeFunction( wParam );     // wParam == HDC to paint with.

            ValidateRect( pWnd32->hWnd16, 0 );
            return 1;

        case WM_CANCELMODE:
            if ( HWndSysModal == SomeUserGlobalVar )
                return 0;
            break;

        case WM_NCCREATE:
            // This is the first message through the WND proc.
            if ( pWnd32->wndClass->cClsWnds != 1 )
            {
                _DebugOutput( DBF_FATAL | DBF_USER, "USER: Assertion failed" );
            }

            pWnd32->classAtom = DesktopClassAtom;   // USER global variable.

            if ( 0 == DefWindowProc32( pWnd32->hWnd16, msg, wParam, lParam ) )
                return 0;

            // The desktop window better not have a parent!!!
            if ( 0 == pWnd32->hWndParent )
                return 1;

            _DebugOutput( DBF_FATAL | DBF_USER, "USER: Assertion failed" );
            return 1;

        case WM_LBUTTONDBLCLK:
            msg = WM_SYSCOMMAND;
            HIWORD( wParam ) = SC_TASKLIST
            break;

        case WM_QUERYNEWPALETTE:
        case WM_PALETTECHANGED:
            if ( wParam == hWnd )   // wParam == HWND that changed the palette.
                SomeFunction();     // Same basic actions as 3.1, including
                                    // calling RedrawWindow().
            return 0;

        case WM_USER:   // 0x0400 (sent by Explorer)
```

```
        if ( base_USER_FSR_percentage == 0 )
        {
            base_GDI_FSR_percentage
                = GetFreeSystemResources( GSFR_GDIRESOURCES );

            base_USER_FSR_percentage
                = GetFreeSystemResources( GSFR_USERRESOURCES );
        }

        return 0;
    }

    return DefWindowProc32( pWnd32->hWnd16, msg, wParam, lParam );
```

USER 32 并不仅仅转换成 USER. EXE

整个这一章，我要强调的是 Windows 95 USER 子系统的真正工作是由 16 位 USER. EXE 来完成的。确实 USER 32 大部分要转换成 USER. EXE。然而，认为 USER 32 仅由转换组成那是错误的。查看 USER 32 列表，很容易发现 Microsoft 是花了些功夫的来决定哪些例程被经常调用而且不用转换成 USER. EXE 就能实现。在几种情况(如下面的)下，Windows 95 编程人员决定不用转换换取速度时必须保证在 USER32. DLL 中使用比较少的额外代码。下面我讨论的函数不是一个完备的列表，而仅是个窗口系统函数代表例子。

USER32. DLL 中的 IsWindow 函数

IsWindow USER32 版本仅比 USER. EXE 稍微复杂点。因为 USER32 的 IsWindow 函数可以被一个当前没有保持 Win16Mutex 的 Win32 线程调用，此函数使用两个帮助函数(GrabWin16Mutex 和 ReleaseWinMutex)把调用例程的核保护起来。整个 USER32. DLL 都用到 GetWndPtr32 函数(下面将看到)。如果 GetWndPtr32 返回 0，IsWindow 返回 FALSE，表明传来的 HWND 无效。否则当 GetWndPtr32 返回任何非零值时 IsWindow 返回 TRUE。

IsWindow 伪码

```
// in USER32.DLL
// Parameters:
//  HWND    hWnd        // The 16-bit version.
// Locals:
//  BOOL    retValue;

    GrabWin16Mutex();

    retValue = GetWndPtr32( hWnd );     // Pass 16-bit HWND version.

    ReleaseWin16Mutex();
```

GrabWin16Mutex 伪码

```
EnterSysLevel( pWin16Mutex );   // Call KERNEL.97 to acquire the Win16.
```

GrabWin16Mutex 伪码

```
LeaveSysLevel( pWin16Mutex );   // Call KERNEL.98 to release the Win16
                                // mutex semaphore.
```

USER32 中的 GetWndPtr32 函数

GetWndPtr32 函数是个 USER32 一般目的的例程。给定一个 16 位 HWND，它返回一个 USER32 相关的指向 WND 结构的 32 位指针。就其校验 16 位 HWND 和查找 WND 结构而言，GetWndPtr32 函数与 16 位 USER. EXE 中 IsWindow 函数相同。真正的区别是在函数的尾部：16 位 IsWindow 返回 TRUE 或者 FALSE，而 GetWndPtr32 返回一个指向 WND 的 USER32 DGOUP 相关的指针。

GetWndPtr32 伪码

```
// Parameters:
//      HWND    hWnd        // The 16-bit version.
// Locals:
//      DWORD   retValue;

    ConfirmSysLevel( pWin16Mutex );     // Make sure we already have acquired
                                        // the Win16Mutex.

    if ( !hWnd )                        // Filter out the 0 HWND case.
        return 0;

    if ( hWnd & 3 )                     // HWNDs are always multiples of 4.
        return 0;

    if ( hWnd < 0x80 )                  // The lowest HWND value is 80.
        return 0;

    // At offset 0x10070 in the USER DGROUP seg is a DWORD with the
    // maximum HWND value.
    if ( hWnd > *(PDWORD)(USER_dgroup_base + 0x10070) )
        return 0;

    // Dereference the DWORD at 0x10000 + the HWND value to get a pointer.
    retValue = *(PDWORD) (USER_dgroup_base + 0x10000 + hWnd);

    if ( retValue < 0x20000 )   // The HWND(32) heap starts 0x20000 bytes
        return 0;               // into USER's DGROUP. Note the different
                                // comparison than the one IsWindow16 uses.

    // Return a flat PTR to the WND32 structure. The value in the HWND
    // table is a USER DGROUP 32-bit relative offset.
    return (PWND32) (retPtr + UserDgroupBase);
```

USER32. DLL 中的 GetCapture,GetFocus 和 GetActiveWindow 函数

前面我给出的 GetCapture,Getfocus 和 GetActiveWindow 函数的伪码是在 USER. EXE 中实现的。(参见"选择 16 位 USER. EXE 函数"伪码部分)。USER32 中 32 位版本核心部分与其差不多,但有几点是不相同的。第一点是 USER32 版本所有获得和释放 Win16Mutex 是围绕对 USER 数据结构访问进行的。而 16 位的却完全没有必要做这个,通过它们已是 16 位码和通过定义能获得 Win16Mutex。第二个不同是 USER32 版本中没有错误检查。在 PERQUEUEDATA 结构中 16 位版本的这些函数开始之前,要检查和确认存在一个当前队列。

GetCapture,GetFocus,GetActiveWindow 伪码

```
// Locals:
//      DWORD   perQueueOffset;

GetActiveWindow proc
    perQueueOffset = 0x20;  // Offset of the active WND in the PERQUEUEDATA.
    goto GetWndXXX_common

GetCapture proc
    perQueueOffset = 0x18;  // Offset of the capture WND in the PERQUEUEDATA.
    goto GetWndXXX_common

GetFocus proc
    perQueueOffset = 0x1C   // Offset of the focus WND in the PERQUEUEDATA.
    // Fall though...
GetWndXXX_common:
// Locals:
//  PMSGQUEUE pQueue;
//  PWND32 pWnd;

    pQueue = GetCurrentQueuePtr();

    GrabWin16Mutex();

    if ( pQueue->npPerQueue == 0 )
        goto SuckHWND16_release_Win16Mutex; // Oops! No per-queue data.

    // Extract the USER DGROUP relative 32-bit PWND32 pointer out of the
    // per-queue data structure.
    pWnd = *(PWND32 *) (UserDgroupBase + pQueue->npPerQueue + perQueueOffset );

    goto SuckHWND16OutOfUserDGROUP;

SuckHWND16OutOfUserDGROUP:

    // Execution arrives here with a pointer to actual WND32 struct (in EAX).

    if ( pWnd )
```

```
    {
        pWnd = (WORD)( UserDgroupBase + pWnd->hWnd16 );
        // pWnd is now really a 16-bit HWND, not a pointer.
    }

SuckHWND16_release_Win16Mutex:

    ReleaseWin16Mutex();
    return pWnd;    // Either 0, or a 16-bit HWND.
```

USER32. DLL 中的 GetMessagePos, GetMessageTime 和 GetMessageExtraInfo 函数

USER32 中的 GetMessagePos, GetMessageTime 和 GetMessageExtraInfo 函数基本上与 USER. EXE 中的 16 位的差不多。因为这三个函数每次仅从当前线程消息队列中提取单个变量，在跳转到公用空间之前，它们都是小存根开始的，该存根把所希望的偏移量装入寄存器。在公用空间，代码得到指向当前线程队列的指针，并提取相关的 DWORD。有趣的是这三个函数并不像 USER GetCapture, GetFocus 和 GetActiveWindow 函数那样要得到 Win16Mutex。

GetMessagePos, GetMessageTime, GetMessageExtraInfo 伪码

```
// In USER32.DLL
// Locals:
//  DWORD   infoOffset
GetMessagePos proc
    infoOffset = 0x28;
    goto GetMsgXXX_common

GetMessageTime proc
    infoOffset = 0x24;
    goto GetMsgXXX_common

GetMessageExtraInfo proc
    infoOffset = 0x30;
    // Fall through...

GetMsgXXX_common:
// Locals:
//  PMSGQUEUE pQueue;

    // Note that this code doesn't grab the Win16Mutex like the GetWndXXX
    // functions do.

    pQueue = GetCurrentQueuePtr();

    // Add the infoOffset to the base address of the queue, and return
    // the DWORD stored therein.
    return *(PDWORD)( pQueue + infoOffset );
```

USER32. DLL 中的 SendMessage 函数

你对我给出的 USER32 的 SendMessage 例程伪码会感到有点惊奇。毕竟 SendMessage 是在所有 USER 子系统中最复杂例程之一，所以它必应转换成 16 位的 USER. EXE，是吗？在许多情况下，这个假设是正确的。然而，SendMessage 是用得比较多的例程，如果条件满足的话，它不用转换成 16 位的代码。这里讨论的是其主要功能的改进。

USER32 SendMessage 通过获得 Win16Mutex 启动，然后代码转入一长串测试看是否不用真正的 SendMessage(在 USER. EXE 中)这个特殊的消息能否安全地被传送，取消偿试和强迫转换成 USER. EXE 的条件如下：

- HWND 为 0。
- 目的窗口队列与当前线程队列不相同。
- USER. EXE 的 DGROUP 某些变量是非零。

如果被传送的消息使 SendMessage 通过强迫转换测试，SendMessage 开始设置调用目的 WNDPROC。特殊情况，SendMessage 需要它调用的 WNDPROC 地址。

如前面我提到的，WND 结构本身不存储指向 WNDPROC 实际 32 位扩展指针。相反，如果一个 WND 结构是个 32 位窗口，WND 结构中的 WNDPROC 地址指向 16 位码的存根，该存根最终把控制转向 32 位域部分。代码存根实际上是 32 位的 WNDPROC。SendMessage 代码知道这些特殊的存根，从其读出 32 位 WNDPROC 地址。最后，在 JMP 到目标 WNDPROC 之前，SendMessage 释放 Win16Mutex。

SendMessage 伪码

```
// 32-bit version in USER32.DLL
// Parameters:
//  HWND    hWnd
//  UINT    uMsg
//  WPARAM  wParam
//  LPARAM  lParam
// Locals:
//  PWND32  pWnd
//  PMSGQUEUE pQueue;
//  LPVOID  lpvMsgProcThunk // A 16:16 pointer.
//  WNDPROC wndProc32

    GrabWin16Mutex();

    pWnd = GetWndPtr32( hWnd );
    if ( !pWnd )                        // No HWND... gotta thunk.
        goto ThunkToSendMessage16;
```

```
    if ( !pWnd->flags & 0x02000000 )    // Some flag ain't set... gotta thunk.
        goto ThunkToSendMessage16


    if ( pCurrentTIB->pvQueue != pWnd->hQueue ) // Sending a message to a
        goto ThunkToSendMessage16               // different queue. Gotta
                                                // thunk.

    if ( SomeVariableInUserDgroup != 0 )        // USER's in some funky state.
        goto ThunkToSendMessage16               // Gotta thunk.
    if ( SomeOtherVariableInUserDgroup != 0 )
        goto ThunkToSendMessage16

    // Get a flat pointer to the message queue.
    // MapSL takes a selector and an offset, and returns a linear address.
    pQueue = MapSL( pCurrentTIB->pvQueue, 0 );

    if ( pQueue->(0x6A+0xA) != 0 )      // ??? Gotta thunk.
        goto ThunkToSendMessage16;

    if ( pQueue->(0x6A+0x1A) != 0 )     // ??? Gotta thunk.
        goto ThunkToSendMessage16;

    // Get a pointer to the thunk code that USER.EXE created for this
    // window. Index 2 bytes into the USER message thunk, and grab the
    // linear address of the window procedure.
    lpvMsgProcThunk = pWnd->lpfnWndProc;
    wndProc32 = *(LPWORD)(lpvMsgProcThunk+2)

    ReleaseWin16Mutex();    // Don't need this no more.

    // If all went well, jump to the 32-bit window procedure.
    // We've successfully avoided the intertask SendMessage contortions,
    // and have also avoided thunking down to 16-bit USER.EXE.
    goto wndProc32;

ThunkToSendMessage16:       // Well, it looks like we gotta thunk down to
                            // USER.EXE.
    ReleaseWin16Mutex();

    pop return address into EAX
    pop hWnd into ECX

    push    0
    push    hWnd            // in ECX
    push    0
    push    returnAddress   // in EAX

    goto    common thunking code
```

USER32. DLL 中的 GetDlgItem 函数

GetDlgItem 函数是另一个 Win32 中常用到函数，特别是与对话有关的代码部分。给定一个 HWND 和一个子控制 ID，函数需要返回子控制的 HWND。如果你能回想起前面讨论的 WND，你就能明白 GetDlgItem 只是浏览 WND 层的一部分，寻找带有正确控制 ID 的窗口。

USER GetDlgItem 函数通过得到 Win16Mutex 开始的（当我们浏览时，不必改变 WND 层）。对话控制是简单的对话窗口的子窗口。因此，GetDlgItem 所做的全部工作只是为指定的对话列出子窗口表和把它发现的每个窗口控制 ID 与 idControl 输入参数进行比较。这就是 GetDlgItem 所做的事情。当它找到带有控制 ID 域的某个 WND 结构与输入参数相匹配时，代码查表并返回在 WND 结构内某地方发现的 16 位 HWND 数值。当然，在返回之前，释放 Win16Mutex。

GetDlgItem 伪码

```
// 32-bit version in USER32.DLL
// Parameters:
//  HWND    hwndDlg
//  int     idControl
// Locals:
//  PWND32  pWnd

    GrabWin16Mutex();

    pWnd = GetWndPtr32( hwndCtl );  // Get a flat pointer to the WND struct.

    if ( pWnd )
    {
        pWnd = pWnd->hWndChild; // Start at the first child window.

        while ( pWnd )  // While there are child windows...
        {
            pWnd += UserDgroupBase; // convert USER DGROUP relative pointer
                                    // to a flat pointer.

            // Is the control ID of this window what we're looking for?
            if ( idControl == pWnd->ctrlID )
            {
                 pWnd = pWnd->hWnd16;
                 break;
             }

             pWnd = pWnd->hWndNext;  // Advance to next child window.
        }
    }

    ReleaseWin16Mutex();

    return pWnd;    // This is always either 0 or a 16-bit HWND value.
```

USER32. DLL 中的 GetDlgCtrlID 函数

GetDlgCtrlID 函数是 GetDlgItem 函数的补充函数。给定一个 16 位 HWND 值，它仅仅需要返回存储在对应窗口 WND 结构中的控制 ID。与 GetDlgItem 配合，当保存 Win16Mutex 时，代码可完成所有工作了。

GetDlgCtrlID 函数也并不很简单。它传送 16 位 HWND 输入参数给 GetWndPtr32 然而得到一个 USER32 相关的 32 位 WND 指针。假定它得到一个非零指针，函数从 WND 结构中相应地址搜索控制 ID 值，然后返回。（不要忘掉释放 Win16Mutex）。

GetDlgCtrlID 伪码

```
// 32-bit version in USER32.DLL
// Parameters:
//  HWND    hwndCtl
// Locals:
//  PWND32  pWnd
//  DWORD   retValue;

    GrabWin16Mutex();

    pWnd = GetWndPtr32( hwndCtl );  // Get a flat pointer to the WND struct.

    if ( !pWnd )
        retValue = 0;
    else
        retValue = pWnd->ctrlID;    // Grab the ctrlID field out of the WND.

    ReleaseWin16Mutex();

    return retValue;
```

Windows 95 中单一码支持

不论你是否相信，Windows 95 有一部分实用的单一码支持。如果你不相信，检查下面名为 WIN95UNI. C 小程序：

```
#define UNICODE
#include <windows.h>

int main()
{
    MessageBox( 0,
                TEXT("Yes! Really!"),
                TEXT("Unicode in Windows 95?"),
                MB_ICONQUESTION );
    return 0;
}
```

编译后，产生单一码程序。用第八章 PEDUMP 打印出 EXE 文件可以检验它：

```
Imports Table:
  USER32.dll
  Hint/Name Table: 00006084
  TimeDateStamp:   00000000
  ForwarderChain:  00000000
  First thunk RVA: 000060D4
  Ordn  Name
   395  MessageBoxW

  KERNEL32.dll
  Hint/Name Table: 0000603C
  TimeDateStamp:   00000000
  ..... rest omitted....
```

这确实存在一个 MessageBox 单一码的调用。当运行此程序时，会发生什么呢？请检查图 4-5。

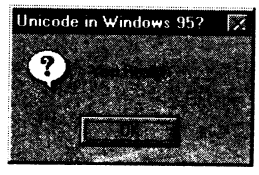


图 4-5 WIN95UNI 程序证明 Windows 95 支持单一码

Windows 95 假定不支持单一码，但如你看到的，在 WIN95UNT 程序中确有某种形式的单一码支持。下面是 Windows 95 单一码 MessageBoxW 启用调用链：

```
MessageBoxW
    MessageBoxExW
        WideCharToMultiByte // Convert the 2nd parameter to ASCII.
        WideCharToMultiByte // Convert the 3rd parameter to ASCII.
        MessageBoxExA       // Invoke the ASCII MessageBoxEx.
```

为什么 Windows 95 找支持单一码这个麻烦呢？为了 Windows 95 教学语言的需要，这些程序不能支持所有功能。程序所能做的事情之一就是放弃 MessageBox，然后说“对不起，我不能运行”。通过提供 MessageBoxW 函数，Windows 95 把被编译的程序中不能工作(不能正确工作)部分去除掉了。

UserSeeUserDo 函数(USER. EXE)

若不讨论 UserSeeUserDo 函数，就不能够完成本章 USER 子系统的讨论。在 Windows 3. 1 中，该函数是被作为未标明的 USER 变量和函数的后入口而引入的。在 Windows 95 中，能通过该入口的东西的增加了。对于 USER 体系认为是关键事物的，应当检查 UserSeeUserDo 提供了什么处理方法是很重要的。

UserSeeUserDo 是在 16 位 USER. EXE 中实现的，带有四个输入参数。第一个

参数指明 UserSeeUserDo 应当做什么，或者它应该返回什么变量值。后面三个参数的意义取决于第一个参数的需要。

前三个子函数允许调用者从 USER 的 16 位 DGROUP 堆中分配、释放，或者压缩存储单元。下五个子函数返回重要的 USER 全局变量值：菜单堆句柄，系统类表头，USER 的 DGROUP 句柄，设备入口链头（参见第六章）和指向桌面窗口的指针。这最后变量不是桌面窗口的 16 位 HWND，而是指向桌面窗口的 WND 结构的 USER32 相关的 32 位指针。

UserSeeUserDo 提供的最后两个子函数是从 USER 使用的新 32 位堆中分配和释放存储单元。子函数 10 分配存储单元，而子函数 11 释放。如果 UserSeeUserDo 第二个参数非零，代码从 32 位菜单堆中分配存储单元。否则，它从 32 位窗口堆中分配存储单元。

UserSeeUserDo 伪码

```
// Parameters:
//  WORD    wReqType
//  WORD    param1, param2, param3

    if ( UserTraceFlags & 0x1000 )
        _DebugOutput( DBF_USER, "UserSeeUserDo" );

    switch ( wReqType )
    {
        case 1:
            // Call LocalAlloc, using USER's DGROUP.
            return UserLocalAlloc(LT_USER_USERSEEUSERDOALLOC, param1, param3);

        case 2:
            // Call LocalFree, using USER's DGROUP.
            return UserLocalFree( param1 );

        case 3:
            // Call LocalCompact, using USER's DGROUP.
            return LocalCompact( param3 );

        case 4:
            return hMenuHeap;   // Handle to the 32-bit menu heap.

        case 5:
            return PClsList;    // Near pointer to first class in list of
                                // system classes registered by USER.EXE.
        case 6:
            return DS;          // USER's DGROUP.

        case 8:
            return PDCEFirst;   // Head of DCE (Device Context Entry) list.
                                // See "DCE" in Chapter 5 of Undocumented
                                // Windows.
```

```
        case 9:
            return HWndDesktop; // The USER-DGROUP-relative 32-bit version.

        case 10:
            // Allocate memory from either the 32-bit menu or window heaps.
            if ( param1 )
            {
                return Local32Alloc( MenuHeapHandleTableBase, param3, 0, 0, 0);
            }
            else
            {
                return Local32Alloc(WindowHeapHandleTableBase, param3, 0,0,0);
            }

        case 11:
            // Free memory from either the 32-bit menu or window heaps.
            if ( param1 )
            {
                return Local32Free( MenuHeapHandleTableBase, param3, 0 );
            }
            else
            {
                return Local32Free(WindowHeapHandleTableBase, param3, 0 );
            }

        case 7:
        default:
            return -1;
    }
```

Windows 95 GDI 模块

在我讨论完 Windows 95 USER 里面一些新东西后，再来讨论 Windows 95 GDI，对于喜爱 GDI 的人来说，有点不太合适。这不是说 GDI 不重要。在图形方面，Windows 95 中确实有很多新的和令人激动的事情。下面以我的经验来讨论图形和 GDI。

如果把 Windows 95 GDI. EXE 和 GDI32. DLL 看成一体的话，这两个子系统与 USER 子系统是平行的。GDI 和 USER 管理的均是从他们堆中分配的目标。USER 目标主要是窗口、菜单、类，GDI 是笔、刷子和位表等。在 Windows 3. 1 中，USER 和 GDI 受到各自的 64K 堆的限制(尽管 USER. EXE 打破了菜单 64K 堆的限制)。在 Windows 95 中，USER 和 GDI 也仍然依赖从 DGROUP 堆中分配的数据结构。同时，USER 和 GDI 可以访问带有 2MB 存储器的 Win32，Win32 可以存放较大的数据项。USER DGROUP，句柄表域，32 位窗口堆格式转换成 GDI 等同部分，可用图 4-6 表示。

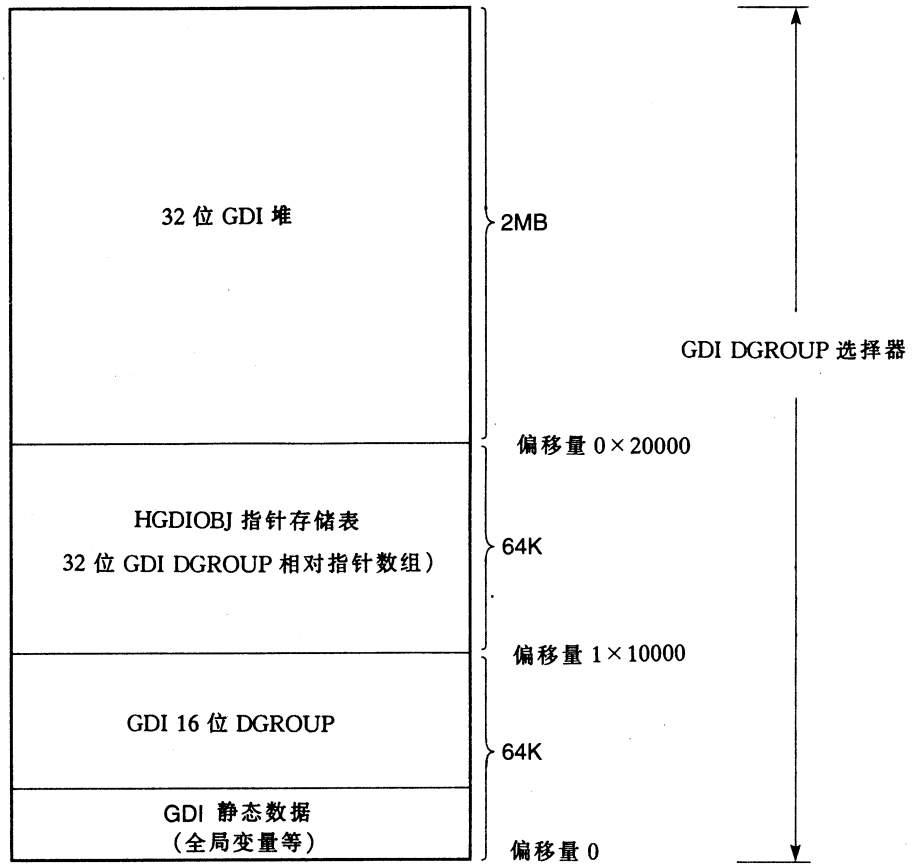


图 4-6　Windows 95 中 GDI 和 USER 结构基本上是平行的，这里你可以看到 DGROUP 的格式与 GDI 的句柄表域是相似的

正如你访问带有句柄(HWND 和 HMENU)的 USER 的目标一样，你也可以用句柄访问 GDI 目标(HPEN，HBRUSH 等等)。前面我讨论过 16HWND 怎样被使用在一个阵表中查出指向 WND 结构 32 位指针的实际偏移量。对于存放在 32 位 GDI 堆中 GDI 目标，从 16 位句柄到 32 位指针转换与 USER 的 HWND 一样的。如果一个句柄引用的 GDI 目标是从 GDI 的 DGROUP 分配的，那么该句柄是个一般 16 位堆局部句柄，对 GDI 的 16 位 DGROUP 的偏移量引用也是很容易的。

从上述观点看，在操作上 USER 和 GDI 是平行的，至少是在对数据结构操作上是这样的。如果你真正掌握了 USER 代码、句柄和指针是怎样工作的话，那么你就能比较容易地了解 GDI 代码的工作了。

Windows 95 GDI 感到应该把什么项移入它的 32 位堆呢？依据 Win32SDK 中 HEAPWALK 程序，存在着浮动的字体和区域。也存在着 HEAPWALK 不能确认的附加目标。

另一个 USER 和 GDI 平行的地方是转换。大多数 USER 子系统码是在 16 位

USER. EXE 中实现的，USER32 主要是(不全部是)服务于转换的。GDI 情况与此相似，但不完全相同。大部分 GDI 仍然是在 16 位 GDI. EXE 中实现的，然而，Microsoft 增加了许多新的 GDI 相关的特征，这些特征不得不用 Win32 支持。部分新码是在 16 位 GDI. EXE 中的。然 Microsoft 声称 GDI 一些新功能是在 GDI32. DLL 中实现的和 GDI. EXE 转换成 GDI32. DLL。Microsoft 指的域是 GDI32 中的 TrueType 光栅、脱机和打印子系统及 DIB 驱动。我还没有证明其真实性，然而通过查看 GDI32. DLL 中运行的程序，好像在 GDI32 中有小部分代码与简单地转换到 GDI. EXE 没关系。

对 16 位 GDI. EXE 特别值得注意的是 16 位模块中 32 位码。在第七章，我在 16 位 New Executable(NE)文件里(该文件通知 Windows 95 装入器为段设置一个 32 位码的选择器)讨论了一点，即当 CPU 把选择器装入它的 CS 寄存器时，它把代码译成 32 位的而不是大多数 Win16 应用程序和 DLL 使用的 16 位码。16 位 GDI. EXE 使用了四个这样 32 位段。尽管在这些 32 位段中没有被输出函数，我还是从 GDI. EXE 中检查了调用这些 32 位段的码，结论如下：

GDI. EXE Segment 0x20： Bezier
GDI. EXE Segment 0x23： 路径，扩展文件(EMF)支持
GDI. EXE Segment 0x24： ??? (不清楚)
GDI. EXE Segment 0x26： “engine font”串在此段里

Microsoft 在讨论 16 位和 32 位 Windows 95 程序关系时讲 Beziers，路径和扩展文件是在 16 位 GDI. EXE 中的。这与我的发现是一致的。

GDI 目标

对 GDI 专家来说，关键的事情是要掌握 GDI 目标。GDI 子系统处理十几种不同的目标类型。绝大多数它们有自己专用的句柄名称(这你已经熟悉了)。例如，设备上下文(DC)是 GDI 目标一种类型，你可传送一个 HDC(指向 DC 句柄)给各种 GDI 函数。同样，一个笔也是个 GDI 目标，你可通过 HPEN(指向 PEN 句柄)访问特定的笔。接受任何特别 GDI 目标类型的函数采用的是 HGDIOBJ 参数。更特殊的 GDI 目标可以认为 HGDIOBJ 是一个基本类，如 HDC、HBRUSH 等。你可以在 TOOLHELP. H 检查 LT_ GDI_xxx # defines 发现 Windows 3. 1 中的 GDI 目标表。然而不幸的是，这些 # defines 并没有在 Windows 95 中对新 GDI 目标类型进行更新。

你可以通知 GDI 试图用制式方式处理自己的目标，因为它有像 SelectObject 和 DeleteObject 这样不必被告知它们正被传送的函数。GDI 查阅目标以决定它是何种类型的，采取相应的动作。那么 GDI 是如何知道传送给它的特别目标类型的呢？每个 GDI 是用标准头标开始的，头标中包括一个 WORD 来指明其类型。Windows 95 GDI 目标表(包括相应的标号值)如下：

```
PEN             0x4F47 (1)
BRUSH           0x4F48 (2)
FONT            0x4F49 (3)
PAL             0x4F4A (4)
BITMAP          0x4F4B (5)
REGION          0x4F4C (6)
DC              0x4F4D (7)
IC              0x4F4E (8)

// Beyond this point, the markers get a bit sketchy, but here's my
// best guess...

METADC          0x4F4F
METAFILE        0x4F50
ENHMETADC       0x4F51
ENHMETAFILE     0x4F52
```

GDI.EXE 中的 IsGDIObject 函数

如果输入的 GDI 目标句柄(HGDIOBJ)是无效时,IsGDIObject 函数返回 FALSE。有趣的是,其说明中讲如果 IsGDIObjecct 返回 TRUE 时,其输入句柄也可能不是真正的 GDI 目标句柄。然而,函数的定义是要决定一个 HGDIOBJ 是否有效。说明没有告诉如果输入 HGDIOBJ 参数是有效的,返回的数值代表输入的是什么类型。这对应用程序如 Bounds-Checker/W 需要验明句柄像 HDC、HBRUSH 等非常方便。

如我前面提到的,GDI 把一部分目标存储在 16 位 DGROUP 堆中而其他(字和区)在 32 位堆中。IsGDIObject 需要做的第一件事情就是找出它应当在哪个地方去读出目标类型 WORD(如 0x4F47)。幸运的是,这并不困难。从 16 位 DGROUP 堆中分配的 GDI 目标被分配带有 LMEM_MOVEABLE 属性。简而言之,16 位 LMEM_MOVEABLE 句柄总是以 2,6,0xA 或 0xE 结尾的。你可能回想起本章的前部分,32 位 USER 或 GDI 堆的句柄总是 4 的倍数。

知道了这两种目标类型的关键区别,IsGDIObject 仅仅需要检查倒数第二位就行了。如果设置句柄是以 2,6,0xA,或 0xE 结尾的,那么目标是从 GDI 的 16 位 DGROUP 分配的。如果倒数第二位为 0,句柄值以 0,4,8 或 0xC 结尾,那么目标是在 32 位 GDI 堆中分配的。在这两种情况下,IsGDIObject 计算出可以找到目标的地址和指向目标的指针。使用该指针,IsGDIObject 提取块类型 WORD。

有了块类型 WORD 后,IsGDIObject 屏掉在别处有定义而在此无定义的两个位,得到的结果应该在 0x4F47 和 0xF52 之间。如果不是,表明这是个无效的 GDI 目标,IsGDIObject 返回 0。如果块类型 WORD 是在此范围的,IsGDIObject 从这个值中减去 0x4F46,得到一个基数,这就是 IsGDIObject 返回的数字。

IsGDIObject 伪码

```
// In 16-bit GDI.EXE
// Parameters:
//  HGDIOBJ hObj
// Locals:
//  PGDIOBJ pObj;
//  WORD    retValue;   // The doc says a BOOL, but it's really an obj type.

    // Note that the doc says that this function can return TRUE without
    // it really being a GDI object.

    if ( hObj == 0 )    // Check for the bonehead case.
        return 0;

    if ( (hObj & 2) == 0 )  // Object handles in 32-bit heap are
    {                       // multiples of 4.

        // Use the handle as an offset into the GDI object table that
        // starts 0x10000 from GDI's DGROUP. The DWORD there is a PGDIOBJ.
        // Actually dereferences through ES. ES points to GDI's DGROUP.
        pObj = *(PGDIOBJ)( 0x10000 + hObj );
    }
    else    // Object handles that end in 2, 6, A, or E are GDI 16-bit
    {       // heap local handles.

        // Since the hObj is a moveable handle, it's a pointer to a 16-bit
        // local heap handle table entry. The WORD at offset 2 in a
        // handle table entry is 0xFF if the block is free. Check for
        // this case, and bail out if so.
        if ( *(NPWORD)(hObj+2) == 0xFF )
            return 0;

        // If we get here, it's (theoretically) an in-use handle.
        // Dereference the first WORD of the handle table entry to get
        // a near pointer to the GDI object within the 16-bit GDI heap.
        pObj = *(NPWORD)(hObj);

        // If LMEM_DISCARDED (???) flag set in handle flags, then
        // the pObj is really a 32-bit heap handle. Go dereference it
        // in the table starting 64K into GDI s DGROUP.
        if ( *(NPWORD)(hObj+2) & 0x40 )
            pObj = *(PGDIOBJ)( 0x10000 + pObj );
    }

    retValue = pObj->ilObjType
    retValue &= 0x5FFF;         // Mask off the 0x8000 and 0x2000 bits.

    retValue -= 0x4F46          // Make the object type value 1-based.
    if ( retValue <= 0 )
        return 0;

    if ( retValue > 13 )        // Is the object type out of range?
        return 0;               // Yes? Sorry, you lose. Do not pass Go.

    return retValue;    // Return value indicates the object type.
```

GDI32. DLL 中的 GetObjectType 函数

Win32 API 中没有 IsGDIObject 函数。然而 Win32 API 更前进了一步，提供了一个返回传送给它的 HGDIOBJ 句柄类型的函数。在检验 HGDIOBJ 的类型方面，GetObjectType 比 IsGDIObject 稍微复杂点。

GetObjectType 首先检查 HGDIOBJ 句柄，看它是否真是一个选择器。总的文件目标句柄对其本身中的数据是一个实际选择器。如果 HGDIOBJ 看起来像个选择器的值，GetObjectType 转到该段，如果找到了它认为是正确的域的话，它返回值 OBJ_METAFILE。

完成这个不太认真的工作后，GetObjectType 下面所做的工作与 GDI. EXE 中 IsGDIObject 所做的工作相似。如果句柄以 2,6,0xA 或 0xE 结尾，GetObjectType 假定是 USER 16 位 DGROUP 中目标的 16 位局部堆句柄。这种情况下，GetObjectType 调用 Win16Mutex 以阻止潜在的改变 USER 堆的状态或者正在被检查的目标的线程发生。如果 HGDIOBJ 不以 2,6,0xA 或者 0xE 结尾，GetObjectType 认定目标是 32 位堆中的字或区。在这两种情况下，代码创建一个指向 GDI 目标的 32 位指针。

应用这个指向目标的指针，GetObjectType 提取目标类型 WORD 和进行与 IsGDIObject 相似的屏位和减法过程。GetObjectType 然后检查目标类型以确认其是否在允许的范围内，如果不是的话返回 0。如果目标类型为 6(一个 HDC)，GetObjectType 进一步探查目标数据看它是否可能是个扩展的总的文件 DC 或存储器 DC。如果是的话，函数从 WINGDI. H 中返回相应的 OBJ_XXX 数值。

GetObjectType 最后一步是转换 16 位目标类型值(如 TOOLHELP. H 中 LT_GDI_XXX 值)为其相应的 32 位 OBJ_XXX。因为一些奇怪的理由，OBJ_XXX 值与目标本身存储的目标类型值的映射不是一对一的。(这可能是由于 OBJ_XXX 值最初是由 Windows NT GDI 人员定义的，并不是基于 Windows 3. 1 GDI. EXE 的)。在任何情况下，目标类型需要从 GDI. EXE 使用的值转换成 WINGDI. H 定义的 OBJ_XXX 值。这种转换是通过查表完成的。GetObjectType 的最后部分是释放 Win16Mutex，当然如果是它先前占用的话。

GetObjectType 伪码

```
// in GDI32.DLL
// Parameters:
//  HGDIOBJ hObj;
// Locals:
//  BYTE fHaveWin16Mutex
//  DWORD retValue;
//  PGDIOBJ pObj;

    fHaveWin16Mutex = FALSE;    // We'll only grab the Win16Mutex if we
                                // absolutely have to.
```

```
    Set up a structured exception handling frame in case all this monkey
    business goes bad on us.

    if ( LAR (load access rights) succeeds on hObj )
    {
        if ( access rights indicate a non-system, ring 3 descriptor )
        {
            WORD MetaFileType;

            Use hObj as a selector, and grab the first WORD of the
            segment it points to. Call this value MetaFileType.

            if ( MetaFileType < 1 )
            {
                Grab the WORD at offset 2 in the segment.
                if ( this WORD == OBJ_METAFILE )
                {
                Grab the WORD at offset 4 in the segment.
                if ( (this WORD == 0x100)  || (this WORD == 0x300) )
                {
                    retValue = OBJ_METAFILE
                    goto done;
                }
            }
        }
    }
}

// Figure out where the object resides (in GDI.EXE's DGROUP? or in
// the 32-bit GDI heap?).

if ( hObj & 2 ) // Object handles that end in 2, 6, A, or E are GDI
{               // 16-bit heap local handles.
    EnterSysLevel( pWin16Mutex );
    fHaveWin16Mutex = TRUE;

    pObj = ConvertHGDIOBJToPtr32( hObj );
}
else    // Object handles in a 32-bit heap are multiples of 4.
{
    // Index into the handle table and grab out the GDIOBJ pointer.
    pObj = *(PGDIOBJ) ( hGDIHeapHandleTableBase + hObj );

    // The GDIOBJ pointer is relative to GDI's DGROUP, so go add the
    // offset of GDI's DGROUP to make it a flat pointer.
    pObj += GDIDGroupBase;
}

retValue = pObj->ilObjType; // Get the object type WORD.
retValue  &= 0x5FFF;        // Mask off the 0x8000 and 0x2000 bits.

retValue -= 0x4F47          // Make the value 0 based (so that we
                            // can do an array-based translation later).
```

```
    if ( retValue >= 12 )   // Out of range? You lose. Do not pass Go.
    {
        SetLastError( ERROR_INVALID_HANDLE );
        retValue = 0;
        goto done;
    }

    // If the object is a DC, it could be one of several different subtypes.
    // Peek inside the DC structure and see if we can figure out what it is.
    if ( retValue == 6 )        // 6 == DC
    {
        if ( pObj[102] != 0 )   // Is WORD at offset 102 in DC != 0 ?
        {                       // Yes? Then it's an enhanced metafile.
            retValue = OBJ_ENHMETADC;
            goto done;
        }

        if ( pObj[0xE] & 1 )    // Is bit 1 in the BYTE at offset 0xE turned
        {                       // on? If so, it's a memory DC.
            retValue = OBJ_MEMDC;
            goto done;
        }
    }

    // Convert the 16-bit object type stored in the object into its
    // equivalent OBJ_xxx value as given in WINGDI.H.
    retValue = ObjectTypeConversionArray[ retValue ]

    // The array conversions are as follows:

    Win16 (TOOLHELP.H)      Win32 (WINUSER.H)
    ------------------      -----------------
    LT_GDI_PEN(1)           OBJ_PEN
    LT_GDI_BRUSH(2)         OBJ_BRUSH
    LT_GDI_FONT(3)          OBJ_FONT
    LT_GDI_PALETTE(4)       OBJ_PAL
    LT_GDI_BITMAP(5)        OBJ_BITMAP
    LT_GDI_RGN(6)           OBJ_REGION
    LT_GDI_DC(7)            OBJ_DC
    LT_GDI_DISABLED_DC(8)   OBJ_DC
    LT_GDI_METADC(9)        OBJ_DC
    LT_GDI_METAFILE(10)     0
    ??? (11)                OBJ_METADC
    ??? (12)                OBJ_ENHMETAFILE

done:
    if ( fHaveWin16Mutex )              // If we grabbed the Win16Mutex
        LeaveSysLevel( pWin16Mutex );   // earlier, release it now.

    remove structured exception handling frame

    return retValue;
```

Win16 应用程序可用的新 Win32 GDI 函数

关于 GDI 最后一点，我好奇地想看究竟有多少新 Win32 API GDI 函数为 16 位码的(这与想知道有多少 Win32 新的 GDI 函数是在 16 位 GDI 中实现的一样，是很自然的)。为了查明是否任一假定的单一 Win32 GDI 函数可被 16 位码调用，我只好打印出从 Windows 95 GDI.EXE 中输出的结果，并把它与从 Windows 3.1 GDI.EXE 中输出的进行比较。滤除未说明的函数外，只剩下了被放在 Win32 说明中的 GDI 函数，仍然是可被 Win16 码调用的。使用“未注明窗口”中的极好的 EXEUTIL 程序来做 GDI.EXE 两个版本的比较是相当容易的。命令：

```
EXEUTIL -diff C:\WIN31\SYSTEM\GDI.EXE C:\WINDOWS\SYSTEM\GDI.EXE
```

输出的结果清楚地显示出了两个版本 GDI 差别。(只有三个未说明的函数从 Windows 95 GDI.EXE 中被移出)。在 Windows 95 GDI.EXE 中增加了许多新的 16 位 GDI 函数。我滤掉了所有的未说明的函数和其他 Win32 API 中没有相应的输出函数。经过重新排列，把 16 位 GDI 函数列在表 4-1 中。

这些被输出的函数假定在被 Win16 码调用时，是没问题的。

表 4-1 从 Win16 码可被调用的新 GDI 函数

函数类型	函数名称
打印(都转换成 GDI32.DLL)	ABORTPRINTER, CLOSEPRINTER, ENDDOCPRINTER, ENDPAGEPRINTER, OPENPRINTERA, STARTDOCPRINTERA, STARTPAGEPRINTER, WRITEPRINTER
设备-独立的位映象(在 GDI.EXE 中实现)	CREATEDIBSECTION, GETDIBCOLORTABLE, SETDIBCOLORTABLE
扩展文件 (在 GDI.EXE 中实现)	CLOSEENHMETAFILE, COPYENHMETAFILE, CREATEENHMETAFILE, DELETEENHMETAFILE, GDICOMMENT, GETENHMETAFILE, GETENHMETAFILEBITS, GETENHMETAFILEDESCRIPTION, GETENHMETAFILEHEADER, GETENHMETAFILEPALETTEENTRIES, PLAYENHMETAFILERECORD, SETENHMETAFILEBITS, SETMETARGN
划线(在 GDI.EXE 中实现)	GETARCDIRECTION, POLYBEZIER, POLYBEZIERTO, SETARCDIRECTION
路径(在 GDI.EXE 中实现)	ABORTPATH, BEGINPATH, COLSEFIGURE, ENDPATH, FILLPATH, FLATTENPATH, GETMITERLIMIT, GETPATH, PATHTOREGION, SELECTCLIPPATH, SETMITERLIMIT, STROKEANDFILLPATH, STROKEPATH, WIDENPATH

混合的(在 GDI.EXE 中实现)	CREATEHALFTONEPALETTE,ENUMFONTFAMILIESEX, EXTCREATEPEN,EXTCREATEREGION, EXTSELECTCLIPRGN,GETCHARACTERPLACEMENT, GETFONTLANGUAGEINFO,GETREGIONDATA

小结

整个这一章,我讨论的都是 Windows 95 USER 和 GDI 模块的奇怪的混合特性,它们都有相当明显的源于 Windows 3.1 的继承性,而又有许多 32 位码。这样可使编程人员利用有 16 和 32 位程序的优点作许多改进。另外,把使用频繁的数据结构(如 WND)移出 16 位堆使 Windows 95 从 Windows 3.1 提高了一大步,尽管 Windows 95 USER 和 GDI 模块并不是任何地方如它的 Windows NT 一样完善,但对 Windows 95 所做的改进,是应当受到人们欢迎的。

第 五 章

内存管理

* *

正当程序员准备熟悉 Windows 3.x 中内存管理的特点和本质之际，Microsoft 推出了 Win32 API（应用程序接口），这给压得喘不过气来的程序员又提出了新的挑战。

从理论上讲，对于 Win32 的三种形式：NT，Windows 95，和 Win32s，它们的内存管理应该是相类似的。但你了解了 Microsoft 在这一领域的历史之后，你会发现 Windows 95 的内存管理与 NT 和 Win32 有着明显的差异。情况确实如此！我将在这一章里剖析 Win32 内存管理的 Windows 95 执行过程。另外，这里所介绍的一般概念也适用于 NT 和 Win32s。

本章将内存管理的内容分成两个部分。第一部分论述了进程地址空间（process address space）、内存文本（memory contexts）和分页属性（paging behaviors）（如写时拷贝）等内容。之后，我们将论述内存管理方面的另一部分内容，即操作系统所提供的用于实现内存配置和管理的 API。

本章没有介绍有关 16 位或 DOS 虚拟设备内存管理（virtual machine memory management）方面的内容。除少数例外，这里所涉及的内容是基于 32 位的。如果你对 Windows 95 的 16 位内存管理感兴趣，请参阅《Windows Internals》一书第二章的有关内容。Windows 95 的 16 位内存管理与 Windows 3.1 的内容几乎完全一样（错误修复 bug fixes 除外）。

基于页面的 Windows 95 内存管理

要想真正了解 Windows 95 内存结构，就必须首先了解 Intel 80386 CPU 的内存分页技术。尽管内存分页技术远早于 80386，但我们所感兴趣的是 Windows 95 如何在 80386 上实现分页的，因而我将在 80386 这一特定环节上详细论述一下。如果你对分页代码已相当了解，则可跳过此节。

内存分页

分页的根本目的是为操作系统提供一种方法，使其与 CPU 共同为程序制造一种假象，使程序认为配置于计算机上的可用内存要比实际内存多。当一个程序读

或写一个字节的内存时，它可以或无需去访问一个字节的物理 RAM。如果一个程序接触到了一个不直接映射此物理 RAM 的地址，则 CPU 会通知操作系统。然后，操作系统会采取必要的步骤将物理内存与程序要使用的地址相联系。

如果运行程序的总内存大于计算机的配置内存，则操作系统需从其它正在使用内存的程序中移走一个 RAM 块。这种处理方式具有潜在的危险，因此在 RAM 块被重新使用之前，Windows 95 在“别处”保留了 RAM 块的原始内容。这里的“别处”指的是计算机的硬件驱动器。对于任何给定时刻，操作系统和运行程序所使用的全部内存均存于 RAM 或硬件驱动器。（这里所介绍的是简化情况，但已经足够了。）虚拟内存（virtual memory）是一个常用术语，它是指一种在次级内存设备，如硬件驱动器上模拟内存使用分页和空间的方法。Windows 95 虚拟设备管理器（VMM 32. VXD 中的 VMM 模块）的主要工作之一是为应用程序提供干扰率最低的虚拟内存。

使许多人感到不解的是分页要影响到 CPU 的内存寻址。如果没有分页，某个程序通知 CPU 使用的地址将和离开计算机存储总线（memory bus）的地址是一样的。例如，对于一个实际模式的程序，你可以方便地根据某个段（segment）来计算物理地址：通过将此段值乘以 16 并加上偏移基址就可获得偏置组合（offset combination）。如果将分页动作激活，则程序所用的内存地址可能与 CPU 向存储总线发送的地址是不同的。分页引入了对所有地址的间接级（a level of indirection）（实际上是两个级）。当某个程序通过地址对 CPU 进行访问时，CPU 将利用 32 位的地址来查看应送入设备总线的物理 RAM 地址。CPU 用来翻译地址的表是由操作系统控制的。将地址译码表置于操作系统控制之下使得操作系统可以通知程序使用任何处于 4GB 区域的 32 位地址，即使对某个给定地址来说并不存在物理地址。

分页（paging）一项之所以实现是因为 CPU 并不为每个以字节-字节（byte-by-byte）为基础的地址提供这种间接级。而且，内存地址的转换会影响 4K 的内存块。例如，如果你利用分页将一个物理 RAM 地址 0x1000 赋值给一个程序地址 0x400000，则 RAM 地址 0x1001 将出现在地址为 0x400001 的程序中，而 RAM 地址 0x1FFF 将出现在地址为 0x400FFF 的程序中。然而，下一个程序地址（即 0x401000）是新的 4K 页面的开始，因此物理地址 0x2000 无需被映射到程序地址 0x401000。程序地址 0x401000 可能被映射到某个不同的物理 RAM 地址中去，如 0x6000，或者根本无需映射。至于那一页需要 RAM 的映射是由操作系统的分页代码（operating system's paging code）来决定的。

除了允许操作系统提供虚拟内存之外，CPU 对分页的支持还允许操作系统在安排内存目标方面具有极大的伸缩性。这里的目标（objects）是指操作系统代码、程序代码、程序数据段、和内存映射文件（memory mapped files）。操作系统所使用的内存设计被称为地址空间设计（address space layout）。我将在 Windows 95 中简单介绍地址空间。

分页的优点是操作系统可以将操作系统目标布于整个 CPU 地址区域（对于 Intel 386 系列的 CPU 而言，是 40 亿的字节区域）。CPU 理论上可访问的整个内存地址区被称为地址空间（address space）。CPU 因其分页动作的激活而要转换的地

址称为线性地址(linear address),这样它就与由 CPU 转换后的地址区别开来。这些是实际的地址,它们要离开 CPU 总线而进入物理 RAM。这些地址称为物理地址(physical addresses)。在绝大多数情况下要记住,程序和 API(应用程序接口)的调用所处理的是线性地址而非物理地址。

在分页支持的前提下,操作系统可以为特殊的项目赋予各种地址空间区段,同时为那些需要成长或需要追加的项目留有空间。例如,当某个程序开始运行时,Windows 95 通过缺省来为该程序栈保留 1MB 的 CPU 地址空间。这并不表明 Windows 95 要将 1MB 的物理 RAM 映射到内存地址栈区内,而是表明该栈区的最小尺寸是 1MB。Windows 95 只是将物理内存映射到程序使用的 4KB 区域内。

分页使得操作系统有能力毫不费力地保留大量的未被使用的内存地址(带 RAM 的)。这就好比你为某个音乐会保留了 12 个位子,尽管并不知道你有多少朋友要来。如果只来了三个,那你只需付三个人的钱就可以了。

对于特定的时间,CPU 的 4GB 地址空间中的 4KB 区段(页面)处于下述四种可能的状态:

状态 1:可使用的(Available)。此页内存没有为任何人保留,因而理论上是可分配的。企图向该内存读或写将导致缺页中断异常(异常 14(0Eh)),有关缺页中断我将谈到。

状态 2:保留的(Reserved)。该页面属于已被查询了的内存区的一部分。然而,物理 RAM 并不现时地映射到这一地址,也没有任何的硬件驱动器被用来保留其内容的备份。任何对该内存的读或写的访问均导致缺页中断异常(异常 14(0Eh))。注意,即使是操作系统给某页面的拥有者一个机会将页面状态改变为可提交并存在的状态(状态 3)(Commited and Present),情况也会如此。

状态 3:被提交并存在的(Committed and Present)。该地址区已由某人所配置,而且某个程序正用其来存放信息。CPU 的分页机构已将某 4KB 的物理 RAM 块映射到该页地址。对该页的读或写均将使此物理 RAM 被映射到所要读或写的页面上去。被提交并存在状态的一个替代形式是所谓的页面锁定(pagelocked)。一个锁定的页面是被提交了的、存在的、并确保不被调出的。在页面未被锁定之前,总存在与某个锁定页面相联的物理 RAM。

状态 4:被提交但不存在的(Committed and not-present)。这种状态类似于前一状态(状态 3)。程序已经分配了内存并正在利用该内存来存放信息。其差别是操作系统已经确定出被映射到页面上的 RAM 在其它地址更急需使用。因此,CPU 将此部分内存内容复制到硬盘驱动器中并将该页标记为"不存在"。

像状态 1 和状态 2 一样,如果页面中的某个内存地址被访问则将导致缺页中断。不同的是,当一个程序访问该内存时,操作系统将明确地控制着缺页中断异常并将一个 4KB 的物理 RAM 块重新映射到该页。接下来,操作系统从硬盘读取该页的原始内容,最后重新运行失效页面的指令。结果是该程序并不知道发生过缺页中断。这种对硬盘上 RAM 使用空间的透明模拟其实就是虚拟内存的实质。

Windows 95 提供了可以分配内存页面并可将其改变为上述属性的应用级的 API。它们是 VirtualXXX(VirtualAlloc,VirtualFree 等)函数,这些将在本章后面讨论。

内存分页与选择器

如果你为 Windows 3.x 编过程,你可能想了解内存分页是如何与选择器相协调的。以 16 位保护模式运行于 Intel CPU 上的程序必须始终使用选择器访问 CPU 地址空间内的一个存储单元。每一个 Win16 位的程序代码段均与一个选择器相连,因为该程序的数据段和它所分配的任何内存块都是全局的堆函数(例如,GlobalAlloc)。因此,我们不可能在使用 16 位 Windows 编程时不使用选择器。

与每个选择器相关的最重要信息是它指向内存的何处(即它的基本地址)。在 386 上,选择器的基本地址处于 0 到 4GB—1 之间。另外,选择器从理论上可以指向 CPU 的所有地址范围。然而,选择器的基本地址是被定义成为线性地址的,而不是物理地址。因此,CPU 的分页机构是在选择器之下运行的。对于 Windows 3.1 和 Windows 95,16 位代码均不考虑对分页和虚拟内存的支持,反而认定有大量的内存可以获得。16 位全局堆管理代码保留了大量来自于 ring 0 操作系统分量的存储单元,并将这类存储单元分成更小的存储单元以使程序可以通过选择器来访问他们。选择器的基本地址不必从 4KB 的页边缘开始,同时给定内存段的每页也不必以物理方式存在。

如上所述,选择器/段(selector/segment)管理代码并不需深入涉及分页级。这样就使得下面的分页系统代码可以提供虚拟内存并设定在要访问时此内存是存在的。"Windows Internal"(Windows 内核)一书的第二章介绍了 16 位 Windows 3.1 的选择器/段管理代码。此部分内容在 Windows 95 中变动不大。

如果你是在保护模式下运行的,则可以避免使用选择器。它们将对内存进行绝对的访问。对于 Windows 95 来讲,重要的是它至少要在 386 CPU 上运行,而 386 的关键特征之一是你可以使用分布于 CPU 地址空间的所有 4GB 区域。如果你将这些选择器装入了 CS 和 DS 寄存器,则可以有效地忘掉这些段的存在。程序可以是内存中带有 32 位偏移量的地址。在本节中,32 位偏移与线性地址相同。使用具有 0 基址和 4GB 边界选择器的模式被称为水平存储模型(flat memory model),以区别于 16 位编程当中出现的小型、中型、压缩型和大型模型。然而要记住,尽管对于 Win32 程序来说,水平式程序似乎不再使用段,实际上 CPU 仍在防护情况下使用着段。如果你想混用 16 位代码与 32 位代码,则要尤其记住这一点,因为 16 位代码不能隐藏段的真实性。

对于允许程序可以接触 CPU 的任意地址的开放式段,你可能想了解操作系统是如何来保护内部数据结构和其它应用代码不该涉入的内存区域的。对于 16 位编程系统来讲,这并不困难,因为选择器定义了特定的起始地址和终止地址,以便程序接触。从理论上讲,操作系统不应该提供任何应用程序不该访问的基址选择器。然而,Windows 3.1 和 Windows 95 则允许你生成并使用所需的选择器,这种作法

的优点将在后面谈到。

如果一个 Win32 程序使用了水平标志段，那么操作系统是如何严格限制该程序对不该接触的区域进行访问呢？这时，操作系统为其页面设置了适当的属性，但不是以区段界限为基础。例如，一个程序不应该盲目地写入或破坏代码区。因此，操作系统就为该代码区设置了只读属性。程序可以读这些页面，但如果试图写则导致缺页中断。否则，带有无用指针的程序则可能写入未被分配的内存页面上去。

操作系统把不被任何人拥有的页面设定为不存在状态。试图进入这类地址的操作也会导致缺页中断。另外，操作系统可将一系列页面设定为超级属性。具有超级属性的页面只能被拥有优先权的代码访问(即操作系统和 VxD 的某些区域)。一般用户水平的程序对于超级页面的访问也将导致缺页中断。你将发现，即使没有段，Windows 95 也可以利用页面管理来有效地保护敏感区域的内存。唯一的不足是，在最低水平上的内存配置是 4KB 页面的形式而不是 16 位段中的单字节形式。

Windows 95 的地址空间以及 Win32 进程

对于 Windows 95 之前的 Windows 版本，所有的运行程序均在同一地址空间内运行。即，任何程序可以很方便地读另一程序的内存，而且一个程序也可以修改另一程序的内存，因而就有可能引入带有错误的程序内容。例如，一个 16 位 Windows 程序(即使在 Windows 95 上)可以拥有 16 位用户组(16-bit USERS's DGROUP)的选择器并写入随机的垃圾内容，从而导致 Windows 系统的运行失败。

Windows 95 首次推出了每一个进程在各自的地址空间上运行的方式(至少是每个 Win32 进程)。这里所谓的自己的地址空间是指，每一个程序可以看到它自己所拥有的内存。其它进程所使用的内存不能被物理地访问。尤其是，Windows 95 内存管理器使用了 CPU 的基于页面的内存管理能力以确保只有被当前进程所拥有的内存才可以在 CPU 4GB 的地址空间内得到映射。其它进程所使用的物理 RAM 在当前进程的页面表里是不能简单地显示出来的，其最大的优点是垃圾程序从理论上只能自己堆积，而不会影响其它程序。每个程序都有自己的领地，如果其内部出了问题，则只能自己倒霉。

其实你不必为 Windows 的这些改进而过于高兴，这种将各种程序隔离的方法并不新颖。UNIX 操作系统已经这样作了很多年了。Windows NT 也同样将每个 Win32 进程置于各自的地址空间。因此，该是 Microsoft 向大众推出的桌面操作系统采用这种最基本的方法的时候了。(Win32s，这个 Win32 家族中被忘掉的成员，对于每个进程没有采用地址空间分开的办法。)

尽管将所有程序的内存分开是相当重要的，但一定范围的内存是需被所有的过程共享的。即，所有进程的线性地址空间中某些页面应当映射到同一物理的 RAM 页面。为何如此？每个进程所使用的 DLL 系统就是一个极好的例证。例如，每个进程以最小限度来申请使用 KERNEL32. DLL，而为每个运行进程均装入一个新的 KERNEL32. DLL 拷贝则是相当浪费的。因此，KERNEL32. DLL(以及其它 DLL 系统如 USER32)将驻留于共享内存。当操作系统因需运行新的进程而切换 CPU 的页面表时，它将离开页面表映射(page table mapping)而单独到达共享内

存。后面我将介绍其它有关共享内存的例子和需要。

由于 Windows 95 将不同进程的内存分开，因而任何有关 Windows 95 在 4GB 地址空间设计的讨论必须引入内存文本的概念(memory contexts)。一个内存文本实际上就是一列 RAM 页以及当给定进程激活时所要映射到线性地址上的内容。换一种说法，内存文本也就是操作系统提供给进程的 CPU 线性地址的景象。

每个进程有其各自的内存文本。当 Windows 95 的进度表将一个程序挂起而运行另一个程序时，它同时要将原来进程的内存文本切换到新进程的内存文本上。由于内存文本是以进程为单位进行保存的，它们有时指的是一个进程文本(process context)。内存文本同时也称为地址文本(address context)。无论叫什么，重要的是要知道内存地址本身并无实际意义，除非你已将内存文本置于其中了。

对于最高级水平，用于 Win32 进程的 Windows 95 内存设计是很简单的。在 4GB 的地址范围内，Windows 95 为应用内存保留了底部的 2GB 地址(从 0h 到 7FFFFFFFH)。顶部的 2GB 地址(从 80000000h 到 FFFFFFFFh)是供操作系统使用的。地址空间的这二个半区内又有几个分区，图 5-1 说明了 4GB 地址空间内各类分区的划分情况。如果你有 Windows 95 DDK，你或许想读一下以“Arenas”(场)为标题的“页面映射和地址空间”一节的内容。

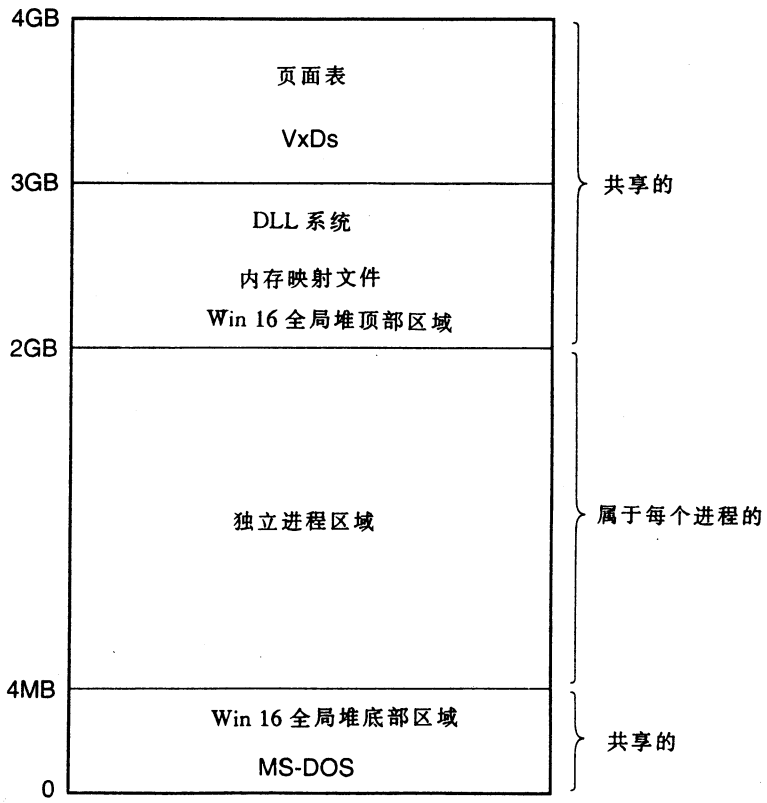


图 5-1　Windows 95 的线性地址空间

最初的 4MB 地址空间被系统虚拟设备(VM)中的所有进程所共享。这个区域

的一部分是低于 1MB 的内存，它包括被安装成为了 Windows 95 引导进程一个部分的 MS-DOS 内存映象。同时，低于 1MB 部分也是 16 位全局堆(16-bit global heap)的较低段。正如我在《Windows Internals》一书所述，Windwos 3.1 中所有的 16 位堆段具有一个或着是低于 1MB 或者是高于 2GB 的线性地址。带有 GMEM_FIXED(全局内存固定)属性的 16 位堆配置(heap allocations)是根据全局堆中的最低级可用地址实现的。因此配置了的块通常以一个低于 1MB 的线性地址结束。在最初 4MB 的地址空间内，你将发现大量 16 位 DLL 系统的内存，因为其中许多(如 KRNL386) 需要固定的和页面锁定的内存，这一点很重要，下面我还要谈到。

4GB 地址空间的下一个区域是从 4MB 到 2GB。这是每一个 Win32 进程使用的独立进程地址空间(per-process address space)。每个 Win32 拥有其各自的代码、数据、和要映射到近 2GB 区域的源。当你切换内存文本时，其效果是将一套不同的页面映象应用于此内存区域。除了程序员设定的特殊情况外，由另一进程映射到该区域内使用的物理 RAM 页面是不能被其它进程所访问的。除了可执行的代码和数据，该区域同样包括由进程使用的具有特殊用途的 DLL 代码和数据。在此区域内，你同样会发现每个线程(thread)的应用堆和栈(application's heap and stack)。

单独应用区的 Win32 程序的缺省安装地址正好处于底部(4MB)。除非你真正了解了分页机理，否则这一点并不好撑握。多个程序是如何装入同一内存地址的？答案是他们共享同一线性地址，但不是同一物理地址。一般来讲，一个进程的线性地址不会映射到 RAM 中的同一物理地址中去的。由于分页，每一进程可认定它自己拥有整个 4MB 到 2GB 的地址。该进程不能看到其它进程的内存，当然其它进程也看不到该进程的内存。分页的魅力在于它使得其它进程物理地消失了。

当 Windows 95 确信在一个程序的多个拷贝之间共享同一 RAM 页是安全的时候，上述将 4MB 到 2GB 内每一进程分开的规则是可以不遵守的。程序代码就是一个典型的例子，因为一个程序通常无需修改其代码。如果你在运行一个程序的多个拷贝，则 Windows 95 通过将含有程序代码的 RAM 映射到所有进程事例(instances of the process)的地址空间来保留有用的 RAM 区。

从最完善的操作系统角度来看，如果每个 16 位进程都能置于其各自的地址空间则是最理想的，32 位进程也同样如此。然而，大量的 16 位程序取决于查看其它程序内存能力的大小。为了与现存的 16 位代码兼容，Windows 95 不得不允许 16 位程序拥有比 Win32 进程更大的对另一个进程的访问权。Windows NT 3.5 曾引入在各自地址空间内运行每个 Win 16 进程的能力，但这将用掉更多的内存并使问题更加复杂。Windwos 95 的设计者们认为这样作不值得。

从第一次看到 Windows 95 就一直吸引我的问题是，16 位任务(tasks)是如何共享地址空间而仍然作为独立进程运行的。结果表明，16 位任务使用的内存总是来自于低于 4MB 和高于 2GB 的共享内存区域的。

现在让我们来看看 4GB 地址空间的上半部分，从图 5-1 你可以看出它分成了二个区域。2GB 到 3GB 的内存区由所有的进程所共享，而且它还可以被 ring 3(用户水平)操作系统代码所触及。在这一区域的最低部地址中，你将发现 16 位全局堆

的剩余部分。在全局堆的上面是内存映射文件(memory mapped files)。这点很有趣,值得深入了解。

如果内存映射文件处于所有进程的共享内存内,似乎任何进程均可看到该内存。即使该进程并不是明确地生成了影象。情况的确如此。对于 Windows 95,使用一个内存映射文件的操作会令该文件对所有的进程进行访问。从这个角度上讲,Windows 95 与 Windows NT 是不同的。Windows NT 使用的是更加成熟的分页模式,因而内存映射文件只能在向该文件打开影象的进程的内存文本中看到。

2GB 到 3GB 区域的最顶部是 32 位 DLL 系统(KERNEL 32,USER 32 等等)。为了在运行至 ring 3 的 DLL 系统之前尽可能多地为内存映射文件释放空间,Windows 95 从 3GB 的下行内存中将 DLL 加到系统上。下面的内容是从 SoftIce/WMOD 命令中摘录的,它清楚地说明了上述方法。

```
:mod
hMod Base     PEHeader      Module Name    EXE File Name
019F BFF70000 0147:BFF70080 KERNEL32
                                           C:\WINDOWS\SYSTEM\KERNEL32.DLL
01A7 BFF20000 0147:81525AF4 GDI32          C:\WINDOWS\SYSTEM\GDI32.DLL
186F BFEF0000 0147:81525E98 ADVAPI32
                                           C:\WINDOWS\SYSTEM\ADVAPI32.DLL
1827 BFC00000 0147:815270F0 USER32         C:\WINDOWS\SYSTEM\USER32.DLL
```

第二列的数据是模块的安装地址。KERNEL 32. DLL 是第一个要装入 32 位 DLL 系统的,而且在保证所有的内容处于 2GB 到 3GB 的范围内应尽可能靠近 3GB 的位置(即地址 BFF70000h)。内存的下一个较低部分是 GDI32. DLL,它处于尽可能靠近 KERNEL32. DLL 的 BFF20000 位置。这些安装地址似乎要计算成 DLL 安装的形式,其实不然。Microsoft 有这样一个程序(来自 Win32 SDK 的 REBASE. EXE),它可以确定每个 DLL 需要多少地址空间,然后算出安装地址以使 DLL 系统尽可能近地靠在一起。在编译和连接完 DLL 系统之后,Windows 95 设计规程将修改 DLL 以使它们具有完善的由 REBASE. EXE 来计算的安装地址。其结果是 DLL 系统尽快地实现了安装而且无需 Windows 95 安装程序的重新定位。

Windwos 95 地址空间的最后一部分是从 3GB 到 4GB(C0000000h 至 FFFFFFFFh)。此最后 GB(吉字节)属于 Windows 95 ring 0 系统分量的使用区(即 VxD)。下面是从 SoftIce/W VXD 命令中压缩输出的内容,可以说明上述问题。

```
:vxd
VxD Name  Address   Length  Seg  ID    DDB       Control   PM  V86  VxD  Win32
VMM       C0001000  00FDC0  0001 0001  C000E990  C00024F8  Y   Y    402  41
WINICE    C001A9C8  04D0FC  0001 0202  C0042418  C001A9CD  Y   Y    2    0
NWLINK    C0067AC4  007C78  0001 0487  C006D1F4  C006C538  N   N    7    0
VNetSup   C006F73C  00121C  0001 0480  C0070814  C006F798  Y   Y    7    0
CONFIGMG  C0070958  0003F8  0001 0033  C0070CF4  C0070958  Y   Y    91   0
VSHARE    C0070D50  001864  0001 0483  C0071130  C007100B  Y   Y    1    0
VWIN32    C00725B4  00277C  0001 002A  C0073DA0  C00725B4  Y   N    29   79
VFBACKUP  C0074D30  0004D0  0001 0036  C0075174  C0074D34  Y   Y    6    0
```

VCOMM	C0075200	000434	0001	002B	C00754F4	C0075200	Y	Y	35	27
COMBUFF	C0075634	000264	0001	0000	C00757D8	C0075634	N	N	0	0
IFSMgr	C0075898	007140	0001	0040	C007A964	C0075964	N	N	117	0
IOS	C007C9D8	002264	0001	0010	C007E8DC	C007C9D8	Y	Y	17	0
SPOOLER	C007EC3C	000140	0001	002C	C007ED20	C007EC3C	N	N	17	0
VFAT	C007ED7C	00A410	0001	0486	C0089064	C0086FD8	N	N	0	0
VCACHE	C008918C	0016A0	0001	048B	C0089A2C	C00893FA	Y	Y	25	0
VCOND	C008A82C	0000A0	0001	0038	C008A870	C008A82C	Y	Y	2	53
VCDFSD	C008A8CC	00019C	0001	0041	C008A938	C008A8CC	N	N	4	0
VXDLDR	C008AA68	0000F0	0001	0027	C008AAF8	C008AA68	Y	Y	18	0
VDEF	C008AD68	0004EC	0001	0000	C008B204	C008AFB8	N	N	0	0
VPICD	C008B254	002314	0001	0003	C008CCCC	C008B690	Y	Y	25	0
VTD	C008DD9C	000570	0001	0005	C008E238	C008DED1	Y	Y	11	0
REBOOT	C008E744	0002F0	0001	0009	C008E9AC	C008E744	Y	N	4	2
VDMAD	C008EA34	002164	0001	0004	C009083C	C008EBF4	N	N	34	0
VSD	C0091364	000220	0001	000B	C0091524	C0091364	N	N	4	0
V86MMGR	C0091584	001334	0001	0006	C0092730	C0091B13	Y	N	25	0
PAGESWAP	C00928B8	0000D8	0001	0007	C0092938	C00928B8	N	N	10	0
DOSMGR	C0092990	000324	0001	0015	C0092B34	C0092990	N	Y	19	0
VMPOLL	C0094350	00018C	0001	0018	C0094470	C0094350	N	N	4	0
SHELL	C0094630	000C24	0001	0017	C0094FE0	C0094CD2	Y	Y	28	0
PARITY	C00953DC	000118	0001	0008	C009549C	C00953DC	N	N	0	0
BIOSXLAT	C00954F4	00009C	0001	0013	C009553C	C00954F4	N	N	0	0
VMCPD	C0095590	0006A0	0001	0011	C0095BC8	C0095590	Y	N	9	0
VTDAPI	C0095C30	0002F0	0001	0442	C0095EB8	C0095E71	Y	N	0	0
PERF	C0096190	000140	0001	0048	C0096274	C0096190	N	N	5	0
VREDIR	C00962D0	005A50	0001	0481	C009B188	C0099E8C	N	N	17	0
NDIS	C0098D20	0071BC	0001	0028	C00A0190	C009C6E7	Y	Y	96	0
VNETBIOS	C00A2EDC	001B78	0001	0014	C00A4818	C00A376F	N	N	8	2
EBIOS	C00A4A54	000078	0001	0012	C00A4A7C	C00A4A54	N	N	2	0
PAGEFILE	C00A4ACC	0000F8	0001	0021	C00A4B6C	C00A4ACC	Y	N	10	0
VCD	C00A4BC4	000430	0001	000E	C00A4F2C	C00A4BD8	Y	N	13	0
VPD	C00A4FF4	000A30	0001	000F	C00A5860	C00A4FF4	Y	N	0	0
INT13	C00A5A24	0009F8	0001	0020	C00A62D8	C00A5A5A	N	N	5	0
VKD	C00A641C	001540	0001	000D	C00A75D8	C00A641C	Y	N	21	0
VDD	C00A795C	000FD8	0001	000A	C00A7F08	C00A795C	Y	Y	23	0
VFLATD	C00A8934	000330	0001	011F	C00A8BDC	C00A8934	Y	N	2	0
VMOUSE	C00A8C64	0008B4	0001	000C	C00A92A0	C00A8C64	Y	Y	12	0
MSMINI	C00A9518	00056C	0001	0000	C00A998C	C00A9518	N	N	0	0
----------	Dynamically Loaded VxDs	----------								
LPTENUM	C0FD7DB8	0005C8	0001	0000	C0FD8328	C0FD7DB8	N	N	0	0
SERENUM	C0FD616C	00007C	0001	0000	C0FD6194	C0FD616C	N	N	0	0
ESDI_506	C102AA7C	001388	0001	008D	C102BD94	C102AA7C	N	N	0	0
HSFLOP	C1029CEC	000808	0001	0000	C102A4A4	C1029CFE	N	N	0	0
VSERVER	C1013A10	014C98	0001	0032	C10268E8	C101C8D0	Y	N	4	0
NETBEUI	C1007CB0	007E18	0001	0031	C100E9FC	C1007CB0	N	N	0	0
SPAP	C1002880	001D74	0001	0000	C1002B4C	C1002880	Y	Y	0	0
PPPMAC	C0FE8840	01961C	0001	0499	C0FE9100	C0FE8928	Y	Y	10	0
voltrack	C0FE4088	0005C8	0001	0090	C0FE45C4	C0FE4088	N	N	0	0
DiskTSD	C0FD85F8	0002B0	0001	0000	C0FD8850	C0FD85F8	N	N	0	0
SB16	C0FD8920	0092BC	0001	32A5	C0FE10A4	C0FE1794	Y	Y	0	0
VJOYD	C0FD6240	0016EC	0001	0449	C0FD7510	C0FD7560	Y	N	2	0
MMDEVLDR	C0FD5088	000090	0001	044A	C0FD50A0	C0FD50F0	Y	Y	6	0
ATI	C0FD4E28	0001AC	0001	0000	C0FD4F54	C0FD4E28	N	N	0	0
ISAPNP	C0FD51AC	00007C	0001	003C	C0FD51CC	C0FD51AC	N	N	0	0

从 SoftIce/W VxD 命令中的全部输出超过了 360 行。灵机一动，我粗略合计了一下 VxD 分量所占用数据块的大小，其将近 1MB。减掉 SoftIce/W 所用去的内存之后，VxD 分量的大小只约为 1KB。尽管这类内存中有些是可以分页的，但操作系统代码最好是隐于 ring 0 区而不要被绝大多数程序员所接近。

你可能认为 Windows 95 会使用分页属性以防高于 0x0000000 地址的 VxD 内存区被窃视或被 ring 3 系统代码所干扰。其实不然。KERNEL 32 的许多地方均将 ring 0 分量中的指针设为变量。同样，VxD 代码中的多处也将指针指向 KERNEL 32 变量，或是指向更糟的 KRNL 386 变量。这里最糟的肇事者可能是 VWIN32.VXD，在第六章中你会看到，它甚至要为此输出两个 Win 32 VxD 服务函数。一项服务由指针通过 VWIN32 向下指向 ring 3，另一项服务是接收在 KERNEL 32 和 KRNL 386 中的 ring 3 地址。

共享内存

在 Win16 中，所有程序和 DLL 的内存均可被其它程序和 DLL 访问。(Win16 对于所有的进程均使用同一局部描述符表(local descriptor table)。因此，各进程之间很容易共享内存：你可以简单地安排两个或多个程序使用同一选择器。尽管 Microsoft 一再声明，但在配置内存时使用 GMEM_SHARE 属性并非 Win16 的需要。

现在来讨论 Windows 95's Win32 的内存管理问题，它将一个 Win 32 进程的全部内存与其它进程分开，除非你采用特殊步骤来实现内存共享。但是，这些步骤并不象定义 GMEM_SHARE 那样简单。将 GMEM_SHARE 设定为 GlobalAlloc 并不需占用由多个内存文本所共享的内存。(这是 Microsoft 的典型情况。GMEM_SHARE 对于 Win16 或 Win32 中的共享内存毫无影响。在 16 位的情况下，这并不必要因为所有内容都 是共享的，而在 32 位时，这点没有考虑。)

你或许听过一些 Win32 的权威人士说，在 Windows 95(或此时的 NT)中共享内存的唯一途径是利用内存映射文件。尽管你可与内存映射文件一同共享内存，但这绝非唯一的解决办法。如果你要共享的只是同一程序的几个事例之间的一小块内存，使用内存映射文件则有点小题大做。本书当中，我重点论述了如何在应用程序之间来共享可读/可写数据。但别忘记，4GB 地址空间的所有上半部是为系统使用保留的，而且可以被所有的进程所观察和共享。

对于最低级水平，内存文本之间的内存共享只不过是在多个进程的页面表映象(page table mappings)中将 RAM 页也包括进去。所共享的内存页面可以映射到每个进程的同一线性地址中去，或者是映射到不同的线性地址中。

在 Windows 95 中，通过内存映射文件所共享的内存总是处于每个进程的同一线性地址。(本章后面要谈到的 PHYS 程序说明了这一点。)然而，如果在你自己的代码中作这种假设是不安全的。原因之一是 Windows NT 并不确保内存映射文件在每个文本中都具有相同的地址。由于利用内存映射文件实现内存共享在 Win 32 编程方面的书中多有涉及，这里我不做详细讨论。

共享内存的最简单方法并没有在有关 Win32 内存管理的内容上提到。尤其是

在连接时，将程序数据段部分设定为 SHARED(共享)属性可以使你很容易地在 EXE 文件的多个拷贝之间或者是在 DLL 的多个用户之间实现内存共享。给定 DLL 数据块 SHARED 属性将使其功能与 Win16 DLL 的数据段的功能相同。好在 Windows 95 提供了共享程序数据段的功能，并同时使独立进程拥有其它数据。你可以在 EXE 或 DLL 中生成多个数据段，将你想共享的所有数据置于一个段(section)并将此段设定为 SHARED 属性，数据的剩余部分将进入另外的属性缺省(非共享)的段。我所介绍 HYS 程序明确地说明了共享内存与非共享内存之间的差别。

通常，Microsoft 编译器将初始化了的数据置于某个以.data 为扩展名的可执行段，并将该段的 IMAGE_SCN_MEM_SHARED 属性去掉，这样使用该数据的每个进程均要生成一个新拷贝。为了共享内存，你需让编译器生成新的段，该段的名称由你给定(尽管 EXE 区段表中要使用前 8 个字符。)例如：

```
#pragma data_seg("SHAREDAT")
```

在 #pragma 之后，你可以声明想要共享的任何变量。但变量应当初始化，否则，编译器将会把它们放入非初始化的数据段中。你可能不准备共享非初始化的数据段，因此应当把该初始化的数据进行初始化。

在完成变量声明之后，如果你想返回来将这些变量放入非初始化的数据段中，则在共享变量声明的结尾加入下述语句：

```
#pragma data_seg()
```

一旦你已将所需的数据区声明为共享，最后一步是将你的设想告知连接程序。有二种方法可以实现上述步骤。通常的方法是将该段及其属性放入扩展名为.DEF 的文件中，例如：

```
SECTIONS
    SHAREDAT READ WRITE SHARED
```

另外一个方法是在连接程序的命令行上设定下述属性：

```
LINK /SECTION:SHAREDAT,RWS <other linker options and files>
```

在该例中，RWS 是“读，写和共享”(Read，Write，and Shared)之意。

关于共享 DLL 数据段，我想提出所谓“顾客需知”(buyer beware)的警告。如果你用另一地址代码或数据标号来初始化数据，当 DLL 在两个或多个进程中装入不同的线性地址时会出现一些有趣的现象。例如，在一个共享的数据块中来考察这样一个看似无害的数据声明：

```
int i;
int * AddressOf_i = &i;
```

问题是 AddressOf_i 在 DLL 装入之前是不知道的。因此，DLL 含有一个预记录(fixup record)来通知安装程序在 AddressOf_i 变量中临时接入正确的数值。在

DLL 首次安装时是没有问题的。现在，来考虑当另一进程装入 DLL 时会出现什么情况，要记住 DLL 在第二次处理时是不能装入同一线性地址的。因为 AddressOf_i 变量已在第一个进程中使用(记住它是共享的)，所以安装程序在第二次处理中是无法得到 AddressOf_i 的正确数值的。当我在自己的代码中碰到这个问题时，我利用指针绕过了它。在我每个进程的数据变量中，我将一个指针指向共享区。由于指针处于每个进程区，安装程序总是可以通过调整指针来使每个进程运行正常。

除了数据共享的明显特点之外，Windows 95 还共享内存的其它区域。我已经提到，上述内存均是一个 2GB 的线性地址并可被 Windows 进程所共享。如果你在运行一个 EXE 文件的多个备份，或者是多个进程中使用同一 DLL，那么为每个代码用户都装入所有的代码段是不值得的。尽管这些代码段不具有 IMAGE_SCN_MEM_SHARED 属性，Windwos 95 只安装了该代码的一个备份，并通过 CPU 页面表将此代码映射到了所有代码用户的内存文本中去。

这种多个进程之间代码段的共享的例外是发生在某个 DLL 无法将同一基本地址装入每个进程的时候。例如，假定 FOO.DLL 由两个不同的进程所使用。当进程 A 装入了 DLL 时，它将内存置于线性地址 X。进程 B 可能使用了某个不同的 DLL 组(但要包括 FOO.DLL)。当进程 B 安装时，在安装程序装入 FOO.DLL 之前其它的 DLL 可能已被赋值到线性地址 X。因为地址 X 无法被进程 B 的内存文本所使用，FOO.DLL 只能装入其它地址。如果出现了这类情况，你通常可以通过将此 DLL 重新定位于某个未曾使用的基本地址上来解决。

Windows 95 的“写时拷贝”(Copy on Write)

在了解了 Windwos 95 的进程之间代码共享之后，现在要关注的是调试程序是如何控制这种操作的。为什么这是一个命令发布(issue)？调试程序通过在代码中写入断点指令(INT 3 中断，opcode(选择代码)0xCC)来实现断点设置。如果调试程序写入断点指令的代码页是由两个进程共享，将存在潜在的问题。调试程序只能调试此两个进程中的其中之一，因此它无法看到另一个进程击入的中断指令。当操作系统接收到一个进程的 INT 3 中断并决定此进程不被调试时，它将因为无法控制的异常的出现而中止此进程。如果 Windwos 95 的内存管理代码准备以上一节所述的方式工作时，你实际上是无法通过由多个进程同时使用的 DLL 来实现调试的，至少并不是使所有其它进程立即中止。当然，你也不可能在一个程序的备份运行时来调试另一个备份。

像 UNIX 之类先进的操作系统是通过“写时拷贝”(copy on write)这样的操作来解决这一问题的。在带有写时拷贝的系统中(如 Windwos NT)，内存管理器利用 CPU 分页在任何可能的地方实现内存的共享，而且只有在必要时才复制 RAM 中的一页内存。

下面通过例子来加以说明。假定一个程序的两个备份正共享相同的代码页(此代码页具有只读属性)。进程之一正被调试，而且用户通知调试程序在代码中设置了断点。当调试程序试图写出断点指令时，它将导致一个缺页中断(此页为只

读页)。当操作系统看到缺页中断,它首先就会判定某个调试程序正想读取内存,而且此请求是合法的。然而,操作系统不仅仅允许对共享代码页的写入,而是生成了一个受到影响页的备份,并迫使调试者的页面表使用此备份。一旦此页面被拷贝并映射,操作系统就会允许写命令通过。此写操作只影响拷贝页而对原页面无影响。

写时拷贝不仅仅限于共享代码。在 Windows NT 中,可写数据页是以只读属性开始的。当程序写入这些页面的其中之一时,CPU 就会生成一个缺页中断。然后操作系统就会将此类页面标记为读/写属性。为何会出现这样的麻烦?当 EXE 或 DLL 的第二个备份装入之后,内存管理器就可以共享仍只具有只读属性的数据页了。如果之后这些数据页被写入,则写时拷贝会在任何必要的地方介入并为每个进程提供必要的独立 RAM 页。

写时拷贝的优点是内存可以尽可能高效地共享。系统只在必要时才生成某个共享页的新备份。但是,写时拷贝需要一个规划成熟的内存区和页面管理表。表面上看,由于 Windows 95 不是直接通过分页来支持写时拷贝,这似乎不够十分成熟,这一点也曾引起 Windows 95 早期使用者的不满。但 Microsoft 目前正推出既能在 Windows NT 上又能在 Windows 95 上运行的全部 Win 32 程序。如果我们在总体设计时忽视了写时拷贝方面的问题则会带来极大不便。

为了保护 Windows 95,我们不能盲目地向共享内存中写入内容。由于需要使用调试程序,Windwos 95 提供了一个所谓的伪写时拷贝方案(pesudo copy on write scheme。在此方案中,WriteProcessMemory(写进程内存)函数代替了某个共享页上的缺页中断。根据此 WriteProcessMemory 函数,操作系统来确定你想写入的地址区是否处于共享内存的范围内。如果是这样,操作系统会为原页面生成新的备份,并将新的页面映射到当前进程的同一线性地址中去,然后进行写操作。PHYS 程序就是这方面的例子。

尽管 WriteProcessMemory 函数足以使调试程序通过绝大多数 DLL 来实现调试,但它在高于 2GB 之上的共享区内是无法工作的。(也可称为功能不全。)因为像 KERNEL 32 这样的 DLL 系统是位于 Windows 95 2GB 地址之上的,通常的应用调试程序是不能像在 Windows NT 中那样介入 DLL 系统的。但是我们仍可偿试在 Windows 95 中生成自己有用的应用调试程序并介入某个操作系统的调用。Visual C_{++} 调试程序和 Turbo 调试程序均跨过了这类调用,即使你的板面没有装配并通知调试程序替代这种调用。如果你想进入 Windows 95 的系统代码,则需要一个不依赖于 WriteProcessMemory 函数的调试程序,例如像 SoftIce/W 或 WDEB386 之类系统级的调试程序。

PHYS 程序

为了验证我所讲述的所有 Windows 95 内存管理的细节,我编制了 PHYS 程序。PHYS 没有想象的用户接口,但它有效地展示了内存设计、共享内存、以及 Windows 95 所支持的伪写时拷贝。

PHYS 的含义很简单。它寻找并显示内存中的各种线性地址(例如,一个代码段或内存映射文件)。当只有一个 PHYS 拷贝运行时,它可以粗略但有效地验证 Windows 95 进程内存的设计。当然,PHYS 程序的功能不仅仅如此。除了显示内存目标的线性地址外,它还可以显示映射到线性地址的物理 RAM 地址和页面的保护属性。通过运行两个或多个 PHYS 的备份,你可以看到哪些内存区域被多个进程所共享。另外,PHYS 还演示了如何对内存代码页的写入以及地址之前与之后的情况,从而证明了 WriteProcessMemory 函数成功地实现了写时拷贝。

PHYS 的全部内容在附带的磁盘上。列表 5-1 是其主要的运行例程(workhorse routine)。ShowPhysicalPages 函数将各种内存目标的线性地址和物理地址计算并显示出来,而且每行一个地址内容。然而,PHYS 并不试图显示每一个地址空间中的内存目标,而是显示了便于表明某个进程的内存设计的内容。

PHYS. EXE 程序中的 ShowPhysicalPages 函数

```
void ShowPhysicalPages(void)
{
    DWORD linearAddr;
    MEMORY_BASIC_INFORMATION mbi;

    //
    // Get the address of a 16-bit DLL that's below 1MB (KRNL386's DGROUP).
    //
    linearAddr = Get_KRNL386_DGROUP_LinearAddress();
    printf( "KRNL386 DGROUP      - Linear:%08X  Physical:%08X  %s\n",
            linearAddr,
            GetPhysicalAddrFromLinear(linearAddr),
            GetPageAttributesAsString(linearAddr) );

    //
    // Get the starting address of the code area. We'll pass VirtualQuery
    // the address of a routine within the code area.
    //
    VirtualQuery( ShowPhysicalPages, &mbi, sizeof(mbi) );
    linearAddr = (DWORD)mbi.BaseAddress;
    printf( "First code page     - Linear:%08X  Physical:%08X  %s\n",
            linearAddr,
            GetPhysicalAddrFromLinear(linearAddr),
            GetPageAttributesAsString(linearAddr) );

    //
    // Get the starting address of the data area. We'll pass VirtualQuery
    // the address of a global variable within the data area.
    //
    VirtualQuery( &callgate1, &mbi, sizeof(mbi) );
    linearAddr = (DWORD)mbi.BaseAddress;
    printf( "First data page     - Linear:%08X  Physical:%08X  %s\n",
            linearAddr,
            GetPhysicalAddrFromLinear(linearAddr),
            GetPageAttributesAsString(linearAddr) );
```

```
//
// Get the address of a data section with the SHARED attribute.
//
MySharedSectionVariable = 1;    // Touch it to force it present.
linearAddr = (DWORD)&MySharedSectionVariable;
printf( "Shared section      - Linear:%08X  Physical:%08X  %s\n",
        linearAddr,
        GetPhysicalAddrFromLinear(linearAddr),
        GetPageAttributesAsString(linearAddr) );

//
// Get the address of a resource within the module.
//
linearAddr = (DWORD)
        FindResource(GetModuleHandle(0), MAKEINTATOM(1), RT_STRING);
printf( "Resources           - Linear:%08X  Physical:%08X  %s\n",
        linearAddr,
        GetPhysicalAddrFromLinear(linearAddr),
        GetPageAttributesAsString(linearAddr) );

//
// Get the starting address of the process heap area.
//
linearAddr = (DWORD)GetProcessHeap();
printf( "Process Heap        - Linear:%08X  Physical:%08X  %s\n",
        linearAddr,
        GetPhysicalAddrFromLinear(linearAddr),
        GetPageAttributesAsString(linearAddr) );
//
// Get the starting address of the process environment area.
//
VirtualQuery( GetEnvironmentStrings(), &mbi, sizeof(mbi) );
linearAddr = (DWORD)mbi.BaseAddress;
printf( "Environment area    - Linear:%08X  Physical:%08X  %s\n",
        linearAddr,
        GetPhysicalAddrFromLinear(linearAddr),
        GetPageAttributesAsString(linearAddr) );

//
// Get the starting address of the stack area. We'll pass
// the address of a stack variable to VirtualQuery.
//
VirtualQuery( &linearAddr, &mbi, sizeof(mbi) );
linearAddr = (DWORD)mbi.BaseAddress;
printf( "Current Stack page  - Linear:%08X  Physical:%08X  %s\n",
        linearAddr,
        GetPhysicalAddrFromLinear(linearAddr),
        GetPageAttributesAsString(linearAddr) );

//
// Show the address of a memory mapped file.
//
linearAddr = (DWORD)PMemMapFileRegion;
printf( "Memory Mapped file  - Linear:%08X  Physical:%08X  %s\n",
```

```
            linearAddr,
            GetPhysicalAddrFromLinear(linearAddr),
            GetPageAttributesAsString(linearAddr) );

    //
    // Show the address of a routine in KERNEL32.DLL.
    //
    linearAddr = (DWORD)
        GetProcAddress( GetModuleHandle("KERNEL32.DLL"), "VirtualQuery" );
    printf( "KERNEL32.DLL        - Linear:%08X  Physical:%08X  %s\n",
            linearAddr,
            GetPhysicalAddrFromLinear(linearAddr),
            GetPageAttributesAsString(linearAddr) );
}
```

PHYS 所显示的内存目标是 16 位 DLL 例程、内存映射文件、和 32 位的 DLL 例程。另外，此例程也显示了 PHYS.EXE 堆地址、代码、共享数据段、源、和栈区。我选择的 KRNL 386 的 DGROUP 表明，Win16 DLL 事实上是被映射到了某个 Win32 进程的地址空间内。(如果不是这样，进程转换是很困难的。)通过显示内存映射文件和 KERNEL 32 例程的地址，证实了它们是处于 2GB 至 3GB 的 ring 3 共享区内的。

图 5-2 是 PHYS 两个备份运行结果的输出。为了显示进程之间的内存共享情况并得到有意义的结果，使用下述正确的顺序是很重要的。运行 PHYS 的第一个例程。当暂停于"Press any key ..."时，跳过该程序，开始运行 PHYS 的第二个备份。这样就确保了第二个例程与第一个例程同时运行。最后，切换到第一个例程并按任一键从而得到其输出的其余部分。

现在让我们来看看第一个例程的第一部分地址区。这些地址以线性地址的方式分类。考察物理地址与线性地址的关系。能不能找出结果？不要太浪费气力，因为什么都没有。Windows 95 保留了一系列可用的 RAM 页面，因此不要试图将物理地址与任何特殊的线性地址相关联。

地址栏的第一项是 KRNL 386 DGROUP。接下来的四项是 PHYS.EXE 可执行中的内存区段。早些时候，我曾在 Windows 95 中提到，32 位进程的缺省安装地址是 4MB(0x400000)。如果你将 PHYS.EXE 的标题(header)由第八章的 PEDUMP 来转储，你会发现该代码区是起始于 0x1000 的某个相对虚拟地址(称为 RVA)。将 0x1000 加到 4MB 上就是 0x401000，该地址在 PHYS 的输出地址中。你可以再深入一步，从而得到数据区、共享数据区、以及源区的 RVA，并验证只要将这些 RVA 加到 4MB 上就可以得到 PHYS 信息表中相同的线性地址。

PHYS 分类输出的下一项内容是缺省的进程堆函数。对于地址 0x410000，它离 PHYS.EXE 模块中的代码和数据段所使用的最后一个线性地址并不远。这好象是 KERNEL 32 以所谓的底部-顶部形式(bottom-up fashion)分配了线性内存。Windows 95 中的初始进程堆的缺省大小是 1MB+4K。这就使得下一个在地址空间中可以得到的线性地址大约为 0x511000。Windows 95 是以 64K 为单元来开始一个新的虚拟内存配置的，因而下一个可得地址实际上是 0x520000。多么奇怪，这

正好是进程环境区开始的地址！这样看来，所谓的底部-顶部配置理论好象并不一定可行。

```
//
// First instance output:
//

***** FIRST INSTANCE *****
KRNL386 DGROUP      - Linear:00036F60  Physical:00245F60  Read/Write USER
First code page     - Linear:00401000  Physical:00BBE000  ReadOnly USER
First data page     - Linear:00408000  Physical:006E2000  Read/Write USER
Shared section      - Linear:0040B000  Physical:0041D000  Read/Write USER
Resources           - Linear:0040D088  Physical:00B3F088  ReadOnly USER
Process Heap        - Linear:00410000  Physical:0082A000  Read/Write USER
Environment area    - Linear:00520000  Physical:00A2E000  Read/Write USER
Current Stack page  - Linear:0063F000  Physical:00ADD000  Read/Write USER
Memory Mapped file  - Linear:8233A000  Physical:0099D000  Read/Write USER
KERNEL32.DLL        - Linear:BFFAF09C  Physical:004F609C  ReadOnly USER
Press any key...

Now modifying the code page
KRNL386 DGROUP      - Linear:00036F60  Physical:00245F60  Read/Write USER
First code page     - Linear:00401000  Physical:00CA1000  Read/Write USER
First data page     - Linear:00408000  Physical:006E2000  Read/Write USER
Shared section      - Linear:0040B000  Physical:0041D000  Read/Write USER
Resources           - Linear:0040D088  Physical:00805088  ReadOnly USER
Process Heap        - Linear:00410000  Physical:0082A000  Read/Write USER
Environment area    - Linear:00520000  Physical:00A2E000  Read/Write USER
Current Stack page  - Linear:0063F000  Physical:00ADD000  Read/Write USER
Memory Mapped file  - Linear:8233A000  Physical:0099D000  Read/Write USER
KERNEL32.DLL        - Linear:BFFAF09C  Physical:004F609C  ReadOnly USER

//
// Second instance output:
//

***** SECONDARY INSTANCE *****
KRNL386 DGROUP      - Linear:00036F60  Physical:00245F60  Read/Write USER
First code page     - Linear:00401000  Physical:00BBE000  ReadOnly USER
First data page     - Linear:00408000  Physical:002FF000  Read/Write USER
Shared section      - Linear:0040B000  Physical:0041D000  Read/Write USER
Resources           - Linear:0040D088  Physical:00B3F088  ReadOnly USER
Process Heap        - Linear:00410000  Physical:00704000  Read/Write USER
Environment area    - Linear:00520000  Physical:00809000  Read/Write USER
Current Stack page  - Linear:0063F000  Physical:00B95000  Read/Write USER
Memory Mapped file  - Linear:8233A000  Physical:0099D000  Read/Write USER
KERNEL32.DLL        - Linear:BFFAF09C  Physical:004F609C  ReadOnly USER
Press any key...
```

图 5-2　同时运行两个 PHYS.EXE 事例时的联合输出

绝大多数环境并不包括 64KB 字符串，但规则总归是规则，因此下一个可得地

址区间应当是该环境开始之后的 64KB，即 0x530000。根据 PHYS 的输出内容，我们可以看到程序当前栈页是从 0x63F000 开始的。起初，这点好象露出了所谓底部顶部理论的破绽。然而，再仔细分析一下，底部-顶部配置方案在这里应当是可行的。记住，一个栈是从某个高级地址向某个低级地址发展的，因此我们要从该栈的顶部减去栈区的长度才可得到该栈区的起始长度。如果当前程序的栈页位置处于 0x63F000，而且我们尚未使用太多的栈空间，则该栈区的结束位置应当是在 0x640000。PHYS.EXE 的缺省程序栈大小是 1MB，因而从 0x640000 中减去 1MB 则得到了 0x540000，这比我的底部-顶部理论所建议的 0x530000 高 64KB。但是，我在此栈区内的某个地址上调用了 VirtualQuery，因此由 VirtualQuery 返回的 AllocationBase 数值为 0x530000。这好象是当安装程序为此程序栈计算所需的大小时，其恰好是以 64KB 来成数的。因此，是某个大小为 1MB+64KB 的区域范围（而不是恰好 1MB）完成了配置。根据我能看到的结果，这种所谓的底部-顶部分配理论是可行的。

将有关项目与程序的数据区直接相关联之后，PHYS 显示了其生成的一个内存映射文件。该内存在偏移量 0x8233A000 处的基本地址是以大于 32MB 的形式进入 2GB 至 3GB 之间的共享 ring 3 区的。因为此 2GB 至 3GB 区是被所有的进程所映射的，任何程序均可观察（并有可能破坏）内存映射文件，即使是对那些进程无法观察到的内存，也会如此。这是 Windows 95 中存在的坏指针重写的一个潜在危险根源。Windows NT 的内存管理器则更加成熟一些，它不允许这种不良地址的存在。

PHYS 输出中的剩余部分是 KERNEL 32.DLL 中 VirtualQuery 例程的地址（0xBFFAF09C），该地址非常接近共享 2GB 至 3GB 区段的尾部。地址为何会这样高？Windows 95 设置了 DLL 系统的基本地址，因而它们就尽可能靠近和高一些。其目的是在 2GB 至 3GB 区域内为内存映射文件留有尽可能多的自由空间。这一点，你自己可以通过考察某些 DLL 系统，如 KERNEL 32.DLL、USER 32.DLL、和 GDI 32.DLL 等来验证。

利用 PHYS 来检测共享内存

为了看到 Windows 95 的进程之间哪些内存区域的内存是共享的，我们来运行 PHYS 的两个备份并对比其输出内容，如图 5-2 所示。让我们将 PHYS 第一个例程的第一部分地址与第二个例程的地址相比较。在这两类地址中，具有相同的物理地址的内存块由这两个例程共享。简单起见，我在图 5-3 中把这些项目分为共享和非共享二部分。

处于共享内存的	处于非共享内存的
KRLA386 DGROUP	第一个数据页
第一个代码页	进程堆

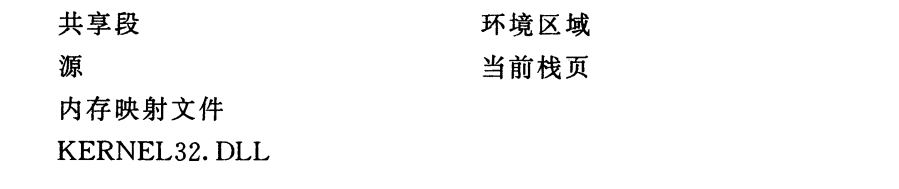


图 5-3　两个同时运行的 32 位进程之间的共享和非共享内存

共享部分并无特别之处。KERNEL 368 的 DGROUP 和 KERNEL 32. DLL 都是 DLL 系统的一部分，你当然希望是共享的。PHYS. EXE 代码和源也是共享的，这表明 Windows 95 正试图充分利用内存。PHYS 还明确地生成了两个剩余的共享区，(共享区段和内存映射文件)，以便与其它例程共享内存。所有这些项目都是读/写程序数据。如果 Windows 95 企图共享这些内存区域，则运行 PHYS 的多个例程时将迅速导致系统故障。

用 PHYS 来测试写时拷贝(copy on write)

PHYS 的最后一项验证是由 WriteProcessMemory(写进程内存函数)提供的伪写时拷贝。让我们来看首代码页的这样三行(压缩如下)：

```
***** FIRST INSTANCE *****
First code page     - Linear:00401000  Physical:00BBE000  ReadOnly USER
...
Now modifying the code page
First code page     - Linear:00401000  Physical:00CA1000  Read/Write USER

***** SECONDARY INSTANCE *****
First code page     - Linear:00401000  Physical:00BBE000  ReadOnly USER
...
```

为了便于输出，当 PHYS 的两份备份运行时有必要记住事件的顺序。第一和第三地址行来自两个不同的进程，而且在代码页被写入之前已经出现。这两个进程中代码页的物理地址是 0x00BBE000，这也证明该页是由两个事例所共享的。中间的地址行是在第一个事例通过 WriteProcessMemory 写入代码页之后输出的。注意到它现在的物理地址的不同了吗？这表明 WriteProcessMemory 已将下面的物理 RAM 页变成了一个不同的内存页。尽管这里并没有显示，但第一个代码页的物理地址保留在了第二个事例的 0x00BBE000 地址上。

PHYS 程序中的“好素材”(适用于高水平读者)

PHYS 程序内部的内容是低级的系统代码，这些代码 Mircosoft 是不大想让你了解的。在一个设计很好的操作系统中，程序不应该访问物理内存与线性地址之间的映射部分。通常，一个程序不用去确定这些映射，但这个能力是 PHYS 程序功

能的心脏。由于 Windows 95 不提供一种支持的方式来获得页面的映射，PHYS 不得不回避操作系统。这种回避的一部分是运行 ring 0 代码（此为 CPU 最高优先级）。除非精确地由操作系统控制，应用程序通常只运行于 ring 3（次优先级），而无需升级至 ring 0。因为 PHYS 运行的 ring 0 代码并未经操作系统认可，我需要写一个常规操作，以便使 ring 3 的 Win32 应用程序可以调用 ring 0 代码。你可简单地改变该 PHY 的 S+ring 0 相关代码并将其卸入你自己的代码中去。

为了将线性地址映射到物理地址，需要将 GetPhysicalAddrFromLinear（从线性地址获得物理地址）函数参与页面表。（参与（party）一词是 Microsoft 常用术语，用于表示做不想做的事情。）页面表是一个很复杂的题目，我将在下一节“内存文本”中来重点讨论这一内容。如果你并不知道页面表是什么，现在就只把它们设想为描述由程序使用的物理地址与线性地址之间映射的数据结构。页面表由操作系统保留而由 CPU 来使用。参照可靠的 CPU 手册，你会发现页面目录是由 CR3 寄存器指向的。然而，不幸的是，检索 CR3 寄存器数值的指令是优先化了的。试图从 ring 3 调用它将导致 CPU 产生一个常规的保护失效（0Dh 除外）。当 Windows 95 看到该失效显示，它会分析该指令并知道其是一个优先指令。Windows 95 并不会中断应用进程，只是不再检索 CR3 寄存器的数值就重新返回应用进程。

这意味着什么？Windows 95 正防止来自应用程序的对页面表的直接进攻。当然，我可以写一个 VxD 来获取 CR3 值，但我不喜欢太多的 VxD 漂入我的系统。另外，即使能得到 CR3 的值，还有很大的问题没有解决。CR3 寄存器告知了页面表的物理地址。但还没有好的方法来将某个物理地址转换成 PHYS 能用的线性地址。由于在使用物理地址时没有关闭分页动作，对于 CR3 值我几乎无能为力。

下一步是看看 Windows 95 是否将页面表映射到了 ring 3 代码可以看到的线性地址中去了，事实表明如此。页面表入口的全部 4MB 区域总是映射到线性地址 0xFF800000（线性内存结束之前的 8MB）的。由于所知信息甚少，这里做了一定的简化。我直接生成了一个指向页面表的指针并开始读你想要的信息。由于 ring 3 Win 32 程序使用的是 32 位线性地址，你甚至可以从任意的 Win32 程序中读取页面表，是不是这样呢？还不会这么快！尽管 Windows 95 的代码器似乎使得页面表非常便于非法指针的重写，实际上这些表并不会轻易被破坏。页面目录和每个页面表入口均保持了一个字节位（用户/监督位）用于表明是否允许该页面在优先级或 ring 0 级上访问代码。用来映射 4MB 页面表区域的页面目录入口是具有用户/监督位的。这意味着由页面表使用的所有的 4MB 内存区域对于 ring 3 代码是闭界的（off-limit）。

因为 Windows 95 页面表对于 ring 3 应用程序是闭界（off-limit），我们需要运行 ring 0 的代码来访问页面表。在我 1993 年五月发表于“Microsoft Systems Journal”（Microsoft 系统杂志）上的关于环形优先级（ring privilege levels）的文章中，曾编写出 RING0.EXE。在 Windows 的内存管理中，RING0 是利用一些实现 Windows 内存管理的穴（holes）从 ring 3 的 Windows 程序来调用位于 ring 0 的 16 位代码的。RING0 如何工作的关键是采用了 CPU 调用门，它为优先级较低的代码提供了一种使用较高优先级代码的方法（例如，从 ring 3 到 ring 0）。因为 Windows 提供

的调用门并不刚好符合请求，RING0 要进入 LDT 并自己产生一个调用门。为此，RING0 使用了相同的 INT 2Fh 子函数，此子函数同样是由 KRNL386 调用来得到一个指向 LDT 的选择器的。（确实如此，即使是在 Windows 95 中！）

在 RING0 出现之后，Alex Schmidt 发表了一篇很好的文章，它将 RING0 推广可以调用 32 位的 ring 0 代码。Alex 提供了一种方法，可以动态安装 VxD 并使用了这些调用门技巧。（所幸的是，Windows 95 现在可以支持动态安装 VxD 而无需使用这些技巧。）当我看到 PHYS 程序需要调用 ring 0 代码时，我趁机更新了 Win32 程序中的可用的原始 RING0 代码。另外，这也意味着生成的是 32 位调用门而非 16 位调用门，其结果见于本书附带磁盘提供的 PHYS 程序中。

这种从某个 Win32 应用程序调用 ring 0 代码的方法只不过带有一点点技巧而已。图 5-4 所示的 GetPhysicalAddressFromLinaer 中的代码就是一个很好的例子。首先，你需要通过调用 GetRing0Callgate 来生成一个调用门选择器。该函数正好是某个 Windows 95 转换成 16 位代码的前部未端部分。在 GetRing0Callgate 的 16 位部分中，该代码生成了以后要在 32 位区域用到的 32 位调用门。GetRing0-Callgate 中存在两个参数，第一个是你想运行 ring 0 的 32 位线性代码地址，第二个是 DWORD 参量数目，它经过栈到达运行于 ring 0 的代码。

一旦你有了调用门选择器，下一步是将其存入某个 6 字节的远指针。是 6 个字节吗？是的。对于 32 位模式，某个远程调用是通过某个与 32 位偏移量联合的 16 位选择器来实现的。由于偏移量为 32 位，选择器肯定属于 32 位区段，这很像 Win32 程序使用的水平选择器。而我们目前需要的是生成一个使用调用门选择器的远程调用，通过它来使 CPU 切换到 ring 0 上。在图 5-4 中，代码将调用门选择器存入了某个 6 字节数组的高 WORD 中（3 WORDs）。指针的偏移量部分并不重要，因为 CPU 忽略了它而是从调用门描述符的偏移量中装入了 EIP。在指针生成之后，代码利用在线编译器通过一个向前的指针来完成调用（因为 C 编译器仅为 32 位的近程调用）。我将调用门的调用用 cli 和 sti 括起来以防对 ring 0 代码的干扰。这样，一旦我们处于 ring 0 代码就可以避免切换到某个安全栈的问题发生。

```
DWORD GetPhysicalAddrFromLinear(DWORD linear)
{
    if ( !callgate1 )
        callgate1 = GetRing0Callgate( (DWORD)_GetPhysicalAddrFromLinear, 1
);

    if ( callgate1 )
    {
        WORD myFwordPtr[3];

        myFwordPtr[2] = callgate1;
        __asm   push    [linear]
        __asm   cli
        __asm   call    fword ptr [myFwordPtr]
        __asm   sti
```

图 5-4　PHYS.EXE 调用 32 位的调用门(callgate)的显示

由于对从 Win32 程序进入 ring 0 的误解，使我认为有必要在编译器中写上 ring 0 PAGETABLE. ASM 代码。首先，16:32 的 ring 0 代码的远程调用使得 CPU 将 8 个字节置于栈上，而不是一般的 4 个。因此，在建立 EBP 框架之后，第一个参数是在 EBP+0Ch 而不是在 EBP+08。更重要的是，当代码返回 ring 3 时，它需要完成一个 16:32 的 RETF 而不是一个 32 位的近程返回。与 16:32 的远程调用一样，RETF 编译器并不知道 16:32 的 RETF 是如何产生的。

总结一下从 Win32 应用程序调用 ring 0 代码的方式，首先是写出此 ring 0 代码(最有可能是写在编译器中)，然后将我们刚才提到的要点加以考虑。下一步是在你的程序中，调用 GetRing0Callgate 并将其通入 ring 0 例程的名称和形参的数目。然后，生成一个带有调用门的 16:32 远程指针，并通过指针进行调用。最后，当你不再需要调用 ring 0 例程时，通过调用 FreeRing0Callgate 来删除调用门。这种作法并不太巧妙，但比在操作系统中完成更合适些。

内存文本(先进的内容)

尽管内存文本是一个很好的讨论内容，但也要考虑到实际的需要。Windows 95 需要保持数据结构以便在某个给定的进程中跟踪哪一页 RAM 应当映射到线性地址上去。为了了解 Windows 95 的内存文本，你需要了解低级的 CPU 分页机理。我在此简要回顾一下 80386 的分页系统，它忽略了一些相当先进的细节。如果你对分页问题感兴趣，可以参照 Intel 手册或其它有关 386 结构方面的书。

CPU 的 80386 系列利用两个查表等级将线性地址译成了送入地址总线的物理地址。第一个查看的表是页面目录，它是 4KB 并可视为具有 1024 个 DWORD 的数组。页面目录数组中的每个 DWORD 包括另外一个被称为页面表(page table)的 4KB 块的物理地址。像页面目录一样，页面表是一个 1024 的 DWORD 数组。页面表数组中的每个 DWORD 包括一个 4KB 内存块的物理地址。

为了使用页面目录和页面表，CPU 将一个 32 位的线性地址分成三个部分，如图 5-5 所示。CPU 将该地址顶部的十个位用作进入页面目录的索引。接下来较低的十个位是进入 4KB 页面表的索引。哪一个页面表来完成这些位索引呢？这是由 CPU 在前一步发现的页面目录所指向的页面表来完成的。该页面表的地址是一个以 4KB 为边界的物理地址。计算的最后部分是线性地址底部的 12 个位，它们作为进入由页面表指向内存的偏移量。

从更方便的角度讲，地址顶部的 10 个位索引成为一个含有指向其它数组的 1024 个指针的数组。该地址接下来的 10 个位则索引成为用于得到物理地址的次级数组。线性地址最下面的 12 个位被加到该物理地址上去从而得到最后的物理地址。

CPU 是如何知道从何处发现页面目录的？页面目录是由 CR3 寄存器指向的，80386 存在一种特殊的寄存器。提供内存文本的一种有力方法是为每个进程直接生成一个页面目录和 1024 个联合页面表，这样就使得 CR3 寄存器在需要时可以指向每个进程的页面目录。

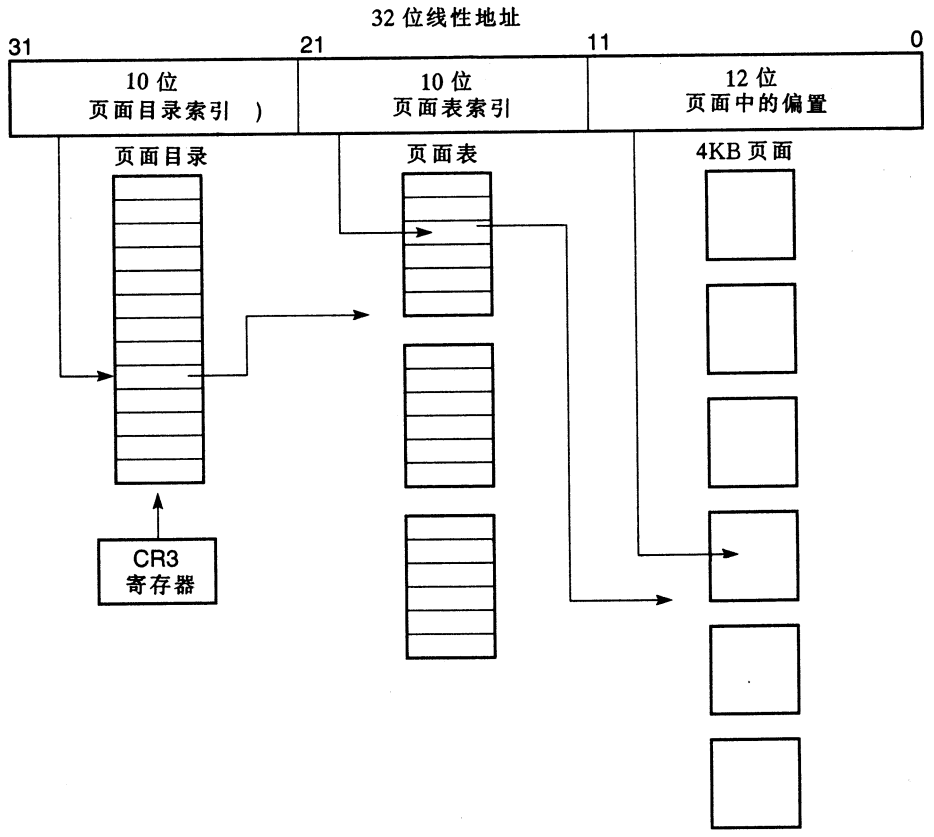


图 5-5　CPU 是如何将某个线性地址转换为物理地址的

这样带来的问题是，映射整个 4GB 地址空间将需要 1024 个页面表，每个表的大小为 4KB。这将在每个进程中占掉 4MB 的内存，显然不能有效地利用 RAM。因此，为了改变内存文本，Windows 95 建立了一个独立的 4MB 的内存区，并在页面目录之内修改输入来迅速改变页面的映射。

你大可不必耽心 4MB 用于分页所花的内存是否会过多。就页面目录级来讲，操作系统仍能通知 CPU 某个 4KB 页面表在内存中并不存在，因而不必将一个 4KB 的物理内存块映射到页面表上去。Windows 95 不使用任何靠近 4MB 的内存区域来管理分页。Windows 95 的页面目录和页面表被映射到某个 4MB 的内存区域，该区域来自于 32 位地址空间端部的 8MB 内存区域。从另一个角度来讲，它们并不使用该地址空间的最后 4MB，但用了前面的 4MB。该内存区域开始于 FF800000h，并可视于 SoftIce/W。页面目录本身存放在该 4MB 区域的某个页面内。

通过 SoftIce/W CR 命令卸出 CR3 寄存器后，你可以很容易地发现页面目录的线性地址。在我的机器上，CR3 包含了 6EE000h。CR3 寄存器包括一个物理地址，因而如果想查看它的话，你需要寻找与其相关联的线性地址，这通过 SoftIce/

W PHYS 命令可以很容易地完成。PHYS 命令可以搜寻页面表，以便发现与物理地址相应的所有线性地址。命令 PHYS 6EE000 生成了两个线性地址，其中第二个地址为 FFBFE000h，它处于为页面表保留的 4MB 的内存区域内。

假定可以在 SoftIce/W 中找到页面目录，则我们应该能证明或反证我所提出的在页面目录中通过设置某个硬件写断点来实现文本切换的观点。如果断点没有关闭，则文本的切换有可能以另一种方式实现。否则，通过操纵页面表来完成文本切换将是一个很强的指示标志。同时，写定位还应当提供一条响应文本切换的线程。

在 SoftIce/W 中，通过运行下列简单的实例便证实了页面目录正被写入某个正常的基本地址。为了能看清这一点，从写动作生成的位置后退几个指令，如下面的 SoftIce/W 代码窗输出所示：

```
_ContextSwitch
0028:C0004856  MOV      EAX,[C001084C]
0028:C000485B  MOV      EDX,[ESP+04]
0028:C000485F  CMP      EAX,EDX
0028:C0004861  JZ       C0004893
0028:C0004863  PUSH     ESI
0028:C0004864  PUSH     EDI
0028:C0004865  MOV      EDI,FFBFE000
0028:C000486A  MOV      ECX,[EDX+04]
0028:C000486D  MOV      ESI,[EDX]
0028:C000486F  REPZ MOVSD
0028:C0004871  MOV      ECX,[EAX+04]
0028:C0004874  SUB      ECX,[EDX+04]
0028:C0004877  JBE      C0004880
0028:C0004879  MOV      EAX,[C00107E0]
0028:C000487E  REPZ STOSD
0028:C0004880  XCHG     EDX,[C001084C]
0028:C0004886  MOV      EAX,EDX
0028:C0004888  MOV      ECX,[C0010CDC]
0028:C000488E  MOV      CR3,ECX
0028:C0004891  POP      EDI
0028:C0004892  POP      ESI
0028:C0004893  RET
```

_ContextSwitch 例程的核心是 REPZ_MOVSD 和 REPZ STOSD 指令。通向 REPZ MOVSD 的三个 MOV 指令正在建立从一个位置向另一个位置拷贝某个内存区域的进程。目标地址为 FFBFE000h 的事实(这一点在页面目录的早些时候曾提到)提供了这样的一个暗示，它表明该例程正清除一套新的进入页面目录的页面表映射。而且，它所备份的每个 DWORD 对应于 1024 个可能的页面表的其中之一。

同样很有趣的是，所移动的 DWORD 数目并不是一个硬编码数，该代码每次是以 DWORD(页面表映射)的数目装入 ECX 的。第二个 REPZ STOSD 的效果并不明显。DWORD 刚刚被拷贝的数目与前一次_ContextSwitch 所调用的 DWORD 数目正进行比较。如果刚刚拷贝的 DWORD 的数目低于前一次，则对前一次内存

文本存在着多余的页面表入口，而且不允许新的文本查看。因而在必要时，REPZ STOSD 将利用一个标志为不存在的页面表来清除这些保留的页面目录入口。

SoftIce/W 有助于将标签_ContextSwitch 置于代码览的顶部。结果表明，_ContextSwitch 例程是 VMM VxD 中的 VMM 服务之一。其地址出现在由某个 VMM 设备描述符块所指向的 VMM 服务表中。那么，SoftIce/W 是从何处获得这样的名称的？让我们来看一看来自 Windows 95 DDK 的 VMM.INC 文件。每个以 VMM_服务开始的行都是由 VMM VxD 提供的服务例程。靠近列表览的末端，你会发现例程_ContextSwitch。同样令我们感兴趣的是，VMM. INC 中_ContextSwitch 的附近是_PageModify 和_PageModifyPermissions 函数。

在 VMM 中找到_ContextSwitch 例程之后，我们可以看到 Windows 95 必需为每个内存文本保留一套页面映射和页面数目。如果幸运的话，我们可以通过 SoftIce/W 命令来证实这一点：

```
:addr
Handle    PGTPTR    Tables  Min Addr  Max Addr  Mutex     Owner
C0FE5D04  C103C6F8  0004    00400000  7FFFF000  C0FD83B4  KERNEL32
C103C9B0  C103E274  0200    00400000  7FFFF000  C103C9E4  MSGSRV32
C1040854  C10416D0  0200    00400000  7FFFF000  C1040898  Explorer
C1045808  C1046190  0200    00400000  7FFFF000  C104584C  Winword
C10483C4  C10402F4  0002    00400000  7FFFF000  C104A220  HEAPWALK
C1048BEC  C0FE38E4  0002    00400000  7FFFF000  C1048C20  WINMINE
C1048850  C104921C  0002    00400000  7FFFF000  C1048884  FREECELL
C1040304  C1042534  0200    00400000  7FFFF000  C10406BC  Systray
C1041398  C104031C  0002    00400000  7FFFF000  C10413CC  MMTASK
C103EA78  C103EF2C  0200    00400000  7FFFF000  C103EAAC  Mprexe
C103CE70  C103D344  0200    00400000  7FFFF000  C103CEB4  Spool32
C10CD00C  C10CD024  0002    00400000  7FFFF000  C10CD050
```

在此输出表中，程序 FREECELL，WINMINE，MMTASK 和 HEAPWALK 均为 16 位程序。有趣的是，即便 Win16 程序总可以看到另外一个程序，Windows 95 也会把它们看成独立的进程和内存文本。然而，这只不过是学术性的，因为 Win16 中的代码和数据段总是被安装在共享内存区域内的(0-4MB 和其上的 2GB)。因此，Win16 程序总可以看到其它程序，即使从技术上讲，它们具有不同的地址文本。

所有的 ADDR 览中剩余的进程或者是 32 位的或者是未知的。标志为“表”(Tables)的列具有误导性，因为它是用于填充内存文本的页面目录入口的数目。每个页面目录映射了 1024 个页面表，而其中每个页面表又映射了一个 4K 区域。因此，每个页面目录入口对应于 4MB 的线性地址空间。要注意 16 位程序只使用了两个页面入口，这是因为 16 位程序在 Win32 每个进程数据区中(0x00400000-0x7FFFFFFF)并不需要内存。Win32 进程，从另一个角度来讲，需要对应于该整个区域的独立页面映射，即使是绝大多数页面已标有不存在的标记。

每个内存文本的句柄(handles)看起来很像线性地址。让我们从某个句柄值确定的地址卸出内存。为此，我任意选择了第一个文本(KERNEL 32 的 C0FE5D04 句柄)：

```
:dd c0fe5d04
0030:C0FE5D04 C103C6F8  00000004  C0FD4D1C  C103C9B0        .         .M......
```

这样，我们就很容易地将第一和第二个 DWORD 与 SoftIce/W ADDR 的输出相匹配。第一个 DWORD(C103C6F8)是 ADDR 命令为 PGTPTR(页面表指针)数值所要报告的内容。第二个 DWORD(00000004)与表纵向数值相匹配。如果你回来重新研究 ContextSwitch 代码，你会发现_ContextSwitch 正期待一个指向数据结构的指针，此时其格式如下：要求指针指向所要拷贝的页面目录入口，并由要拷贝的入口数目所跟随。

在我们卸出内存文本句柄时所发现的第四个 DWORD 同样可以很容易地在 ADDR 输出中找到。它正好是 ADDR 览中下一个文本的文本句柄。(通过进一步考察，可以证实文本保留了在一个连结了的览中。)那么第三个 DWORD (C0FD4D1C)又如何呢？它似乎应当是个指针，为此让我们也把它卸出：

```
:dd c0fd4d1C
0030:C0FD4D1C 00000400  0007FFFF  C0E0E310  C0E0E31C        ................
```

非常有趣！如果你用 0x1000(为一页面尺寸)来乘以第一和第二个 DWORD，你会得到 ADDR 命令为内存文本所报告的最大和最小地址的数值。这好像我们已经找到了 Windows 95 文本管理的核心。

如果你想深入发掘 Windows 95 内存文本，DDK 是必不可少的。不像 SDK 文本那样，DDK 并不试图对程序员隐埋很多。DDK 声明，内存文本是由 VMM. VXD 中的_ContextCreate 所生成的并由_ContextDestroy 来破坏掉。通过编写 VxD 代码，你可以精确地生成、切换至、并破坏你自己的内存文本。当然，将某些东西挂钩联系以便 Windows 95 的其它部分知道你在做什么是需要费一点气力的。

其它一些较有用的 VMM 函数是_CopyPageTable 和_PageAttach。利用_CopyPageTable，你无需像我在 PHYS 程序中所作的那样进入页面表中去，就可以为某个内存文本得到逻辑-物理映射(logical-to-physical mappings)。_PageAttach 函数用于将一个文本的内存映射到另一个具有相同线性地址的文本中去。这正是 Windows 95 所采用的在某个进程的多个备份之间共享代码和数据的机理。

Windows 95 内存 API

Windows 95 的内存管理函数是以分层形式建立的。对于每一个等级(而非底层)，函数取决于更低一层的函数。我认为 Windows 95 的内存管理系统含有 4 级代码。对于最低一级，虚拟设备管理器(VMM)提供的函数用于分配庞大的内存区域并在这些区域内管理页面。应用程序并不直接调用这些 API，而是由 KERNEL 32. DLL 以更高一级内存 API 函数的名义来使用 VMM 内存函数。

接上来的一层是由 KERNEL 32 提供的 VirtualXXX 函数：VirtualAlloc，VirtualFree，和 VirtualProtect 函数。这些函数是依据较低级的 VMM 函数来执行的。

VirtualXXX 函数为应用程序提供了在页粒水平上管理大范围内存空间的能力。

再向上移动一级，我们就可以看到 KERNEL 32 堆函数，HeapXXX。此 HeapXXX 函数包括 HeapAlloc，HeapFree，和 HeapCreate。它们大体上相当于 C 运行库的内存函数（如 malloc，free 等）。事实上，在 Windows NT SDK 运行库函数 DLL 中，malloc 就是一个 HeapAlloc 函数的外壳（wrapper）。内存管理函数的最高一层包括 LocalXXX 和 GlobalXXX 函数。不像在 Win16 程序中，LocalXXX 和 GlobalXXX 实际上是恒定的。例如，GlobalAlloc 和 LocalAlloc 是同一函数；KERNEL32 在其代码中利用同一地址输出这两个函数。在 Win32 中保留 GlobalAlloc 和 LocalAlloc 是没有太多道理的。内存函数不再像 Win16 GlobalAlloc 那样利用选择器来工作，内存也同样不再像 Win16 LocalAlloc 那样，分配于应用数据段之外。GlobalXXX 和 LocalXXX 之所以存在于 Win32 的主要原因是其将现存的 Win16 应用程序转换为 Win32 形式。本章的其余部分主要是深入讨论 Windows 95 Win32 的内存管理 API，分为 4 个层次。除了 VMM VxD 中最低级的函数外，我将为每个内存管理函数给出伪码。在某些情况下，一个 Win32 函数可能在 Windows 95 中并不能执行，或者只是映射到另一个函数中去了，这一点我也将谈到。

VMM 函数

Windows 95 中最低级的内存管理代码是在 VMM. VXD 中。在 VMM 中是 VxD 函数，它用于保留、提交、下发、和释放具有线性地址空间的页面。VMM 同时还包括 VxD 函数，用于询问页面状态、管理内存文本、和安装失效的处理器，它还为 VxD 的使用提供堆函数。表 5-1 的 DDK 描述，说明了 VMM 内存管理相联函数的主要内容。

表 5-1　VMM 内存管理函数的 DDK 描述

VMM 函数名称	目的
_PageReserve	不用分配任何物理空间，就可在当前文本中保留某个范围的线性地址
_PageFree	释放特定的内存块
_PageCommit	将物理页面提交给某一区域的线性地址
_PageDecommit	从某个特定的线性地址区域下发物理空间
_PageAttach	在当前内存文本中将某一线性页面区域映射到与某特定文本（源文本）映射时相同的物理空间
_PageFlush	通过调用适当的分页函数将某一区域的被提交页面写入后备文件。这项服务并不将页面标记为不存在的状态。
_PageModifyPermissions	在特定范围内修改页面的允许属性
_PageQuery	恢复有关某虚拟页面范围的信息。该信息与 VirtualQuery 返回的格式一致。
_PagerRegister	将一个新的页面调度程序通知系统

_PagerQuery	恢复有关注册页面调度程序的信息
_ContextCreate	生成一个新的内存文本。Windows 95 任务与计划部分利用这一服务来为新的 Win 32 应用程序生成单独的线性地址空间
_ContextDestroy	破坏掉由_ContextCreate 服务生成的内存文本
_ContextSwitch	改变当前的内存文本。当前内存文本决定私有空间中的页面映射
_GetCurrentContext	确定当前内存文本
_HeapAllocate	从系统堆分配一个内存块
_HeapReAllocate	在系统堆中再分配或再初始化某个内存块
_HeapFree	在系统堆中释放现存的内存块

如果你熟悉 VxD，你可能在想，这个关于 VMM 内存相关函数的表是不错的，但对于 ring 3 应用代码又该如何呢？毕竟，常规的 ring 3 程序不能只调用任何偶尔出现的 VxD 函数。表 5-1 是个很好的说明：每个函数可由 ring 3 应用程序调用，但非直接地。

事实表明，Windows 95 的代码器感到这套函数对于 KERNEL32.DLL 来说是很重要的。为此，他们为每个函数提供了 Win32 VxD 服务。Win32 VxD 服务是 Windows 95 的一个新的机制，它允许使用一个 C 型的调用转换（无需寄存器）将 ring 3 应用代码调入 VxD。它们不与 Windows NT 服务相关联，是真正的专用进程。

第六章将更详细地介绍 Windows 95 Win32 VxD 的服务。在此，了解到每个由 VxD（如 VMM）提供的 Win32 VxD 是由某个特定数目来确定的就已足够了。高 WORD 是 VxD 设备的 ID，而低 WORD 则是进入设备 Win32 VxD 服务的索引。为了说明表 5-1 的 VMM 函数，图 5-6 给出了 Win32 VxD 服务 ID。第六章介绍了 Win32 的 VxD 服务并有一个更详细的 ID 览。

0x00010000 _PageReserve
0x00010001 _PageCommit
0x00010002 _PageDecommit
0x00010003 _PageRegister
0x00010004 _PagerQuery
0x00010005 _HeapAllocate
0x00010006 _ContextCreate
0x00010007 _ContextDestory
0x00010008 _PageAttach
0x00010009 _PageFlush
0x0001000A _PageFree
0x0001000B _ContextSwitch

```
0x0001000C _HeapReAllocate
0x0001000D _PageModifyPermissions
0x0001000E _PageQuery
0x0001000F _GetCurrentContext
0x00010010 _HeapFree
```

图 5-6　调用 ring 0 VMM 函数时的 VMM Win32 VxD 服务

为了通过某个 Win32 服务来调用这些 VMM 函数的其中之一，KERNEL 32 直接将形参推入栈，并跟有 Win32 VxD 服务数。它然后调用 VxDCall 函数(参照 Unauthorized Windows 95 中的 VxDCall0)。例如，VMM. VXD 中的_PageReserve 函数具有如下形式：

```
ULONG EXTERNAL _PageReserve(ULONG page, ULONG npages, ULONG flags);
```

下面的 KERNEL 32 安装程序代码表明了_PageReserve 是如何从 ring 3 被调用的。

```
BFFA00A6:   PUSH    10              ;; PR_STATIC from VMM.INC

BFFA00A8:   MOV     EAX,DWORD PTR [EBP-000000F4]
BFFA00AE:   ADD     EAX,00000FFF
BFFA00B3:   SHR     EAX,0C          ;; Round up to 4K boundary
BFFA00B6:   PUSH    EAX

BFFA00B7:   PUSH    80000400        ;; PR_PRIVATE from VMM.INC

BFFA00BC:   PUSH    00010000        ;; VWIN32 call 00010000 = _PageReserve

BFFA00C1:   CALL    VxDCall0
```

正如我在高级内存管理 API 中所作的那样，我没有对这些 VMM 函数提供码。应用程序并不直接调用它们，而是将其视为 ring 3 内存管理函数所依赖的基本建筑块。因为一些读者没有用于介绍这些函数 Windows 95 DDK，在此我将它们列了出来，这也为我下面介绍 VxD 函数提供了便利。

WIN 32 虚拟函数

在 Win32 API 的最低级内存管理中，你会发现虚拟函数(如 VirtualAlloc 和 VirtualProtect)。虚拟函数是用来大块地分配和管理内存的。在 Windows 95 中，虚拟函数的粒度为 4KB，使其不适合于替代 C/C++运行库中的 malloc 和 new 函数。就绝大部分而言，虚拟函数是 VMM 函数之上的一个薄层。在我给出虚拟函数的伪码时你将会经常看到这点。

Win16 中最接近虚拟函数的等效函数是全局的堆函数(例如，GlobalAlloc)。

Win16 的全局堆函数和 Win32 的虚拟函数均允许你配置很大的你在管理并想使用的内存区域。不像全局堆函数那样，虚拟函数并不使用选择器来参照内存，而是在 4KB 的区域内处理但并不使用选择器。与此同时，Win16 全局堆函数还允许你以 20h 小字节的方式来配置内存空间。

VirtualAlloc

VirtualAlloc 实际上包括了几个函数。对于任何给定时间，VMM 内存管理器将每页线性内存认为或者是自由的、保留的、或者是提交了的。VirtualAlloc 函数使你可以在一个方向上(从自由向提交)改变某一区域页面的状态。另外，它还可以将保留页面提前改变为提交状态。

最后一次的状态改变——从保留变为提交——对于提供稀疏内存和栈是尤其有效的。在本节中，某个程序首次利用 VirtualAlloc 来保留了一块足够大的内存区域来满足程序的需要。程序其后建立了一个构造好的异常处理程序，以便在驻留的内存区域内发现缺页中断。这些缺页中断一旦产生，程序就会第二次调用 VirtualAlloc。这一次，VirtualAlloc 调用将失效的页面从保留状态改变成提交状态。这样，一个程序就可以“分配”大量的内存空间而无需物理 RAM 在分配时生成备份，而是只有在内存页面结束时物理 RAM 才产生映射。

通常，VirtualAlloc 是由操作系统和程序用来在应用地址空间内分配内存的(即，低于 2GB)。然而，VirtualAlloc 有一个非资源标志(0x8000000)，该标志允许 VirtualAlloc 占用 2GB 之上的内存。2GB 以上的内存是由所有的应用程序所共享的，因此这是一种非资源的方法在进程之间共享内存。你可以通过资源式的内存映射文件来完成上述工作。事实上，简单的验证表明，内存映射文件使用的地址范围是等效于 VirtualAlloc 以 0x8000000 标志所返回的内容的。

在保留内存时，Win32 的 VirtualAlloc 下舍入最接近的 64K 的边界。事实上，由 VirtualAlloc 分配的内存块总是好象排成列的。然而，VirtualAlloc 代码并不做这种舍入，舍入是出现在 VirtualAlloc 使用的_PageReserve 函数中的。

VirtualAlloc 以检查所需的内存区域是否太大而开始。在该文本中过大的内存区域意味着从 2GB 到 4MB，这是为每个应用内存保留的线性地址的大小。VirtualAlloc 然后计算需要分布于内存区域上的页面的数目。在确定所需页面数目之后，VirtualAlloc 将起始地址下舍入到最近的 4KB，而结束的地址则上舍入下一个 4KB。这样，如果你查询了覆盖某页的最后字节和下一页的首字节的某个二字节区域(2-byte region)，VirtualAlloc 将试图保留此二页。

下一步，VirtualAlloc 要处理在 fdwProtect 参数中通过的各种标志。首先，代码寻找非资源的 0x8000000 标志，该标志通知代码在 2GB 之上的共享区域内配置内存。VirtualAlloc 忽略了 MEM_TOP_DOWN 标志，如果通过该标志则将其置为关闭状态。之后，函数检测你是否只通过了 MEM_COMMIT 或 MEM_RESERVED 标志。这两个标志之外的任何位均会导致一个调试方案警告。最后，代码通知 mmPAGEToPC 函数，该函数为一帮助函数(下一节将谈到)，用于将 fwdProtect 参数

标志转换为 VMM PageReserve 所使用的标志。

对于此时的代码，其函数分成两个部分。如果用户不考虑内存保留在哪个地址则其中一部分将运行。另一部分将处理诸如用户在何处设定一个特殊的地址以便保留或提交之类的情况。在上述任何情况中，如果内存已被保留，VirtualAlloc 将调用 Win32 服务 00010000，它是 VMM_ PageReserve 函数的一个外壳(wrappwer)。在保留内存之后(如果必要)，如果调用程序已设定了 MEM_COMMIT 标志，VirtualAlloc 将调用 Win32 服务 00010001，它是 VMM PageReserve 函数的一个外壳。如果调用程序设定了一个特殊地址提交内存，VirtualAlloc 则要检查该地址是否在 0xC0000000 之下，而此 0xC0000000 则是 VxD 区域的起点。

就此代码的整体而言，VirtualAlloc 仔细地检查从_PageReserve 和_PageCommit 返回的数值。如果出现问题，代码则发出一个调试诊断信号，然后从某个出口中断运行。该出口只在失败情况下运行，并将前面保留的页面释放出来。

VirtualAlloc 伪码

```
// Parameters:
//  LPVOID  lpvAddress
//  DWORD   cbSize
//  DWORD   fdwAllocationType
//  DWORD   fdwProtect
// Locals:
//  DWORD   address, startPage
//  DWORD   sizeInPages;
//  DWORD   pcFlags;                // Returned from mmPAGEToPC
//  BOOL    fReserve;

    if ( cbSize > 0x7FC00000 )  // 2GB - 4MB
    {
        _DebugOut( "VirtualAlloc: dwSize too big\n\r",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_NOT_ENOUGH_MEMORY );
        return 0;
    }

    address = lpvAddress;

    // Calculate how many pages will be spanned by this memory request.
    sizeInPages = lpvAddress & 0x00000FFF;
    sizeInPages += cbSize
    sizeInPages += 0x00000FFF;
    sizeInPages = sizeInPages >> 12;

    startPage = PR_PRIVATE; // 0x80000400h from VMM.INC This value can
                            // be either an actual page number or a PR_ equate.

    if ( fdwAllocationType & 0x8000000 )    // Undocumented shared mem flag.
    {
```

```
        startPage = PR_SHARED;              // 0x80060000 in VMM.INC.
        fdwAllocationType &= ~0x8000000;    // Don't need this flag anymore.
    }

    fdwAllocationType &= ~MEM_TOP_DOWN;     // Ignore the MEM_TOP_DOWN flag.

    // You can specify MEM_COMMIT and/or MEM_RESERVE, but no other flags
    // (the undocumented one above notwithstanding).
    if (    (fdwAllocationType != MEM_COMMIT)
         && (fdwAllocationType != MEM_RESERVE)
         && (fdwAllocationType != (MEM_RESERVE | MEM_COMMIT)) )
    {
        _DebugOut( "VirtualAlloc: bad flAllocationType\n\r",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_INVALID_PARAMETER );
        return 0;
    }

    // Convert the fdwProtect flags into the PC_ flag values used by
    // VMM.VXD. Pseudocode follows this function.
    pcFlags = mmPAGEToPC(fdwProtect);
    if ( pcFlags == -1 )    // Something wrong?
        return 0;

    if ( lpvAddress == 0 )  // Don't care where the memory is allocated.
    {
        // Reserve the memory block. startPage should be either
        // PR_PRIVATE or PR_SHARED.
        lpvAddress = VxDCall( _PageReserve, startPage, sizeInPages, pcFlags );

        if ( lpvAddress == -1 )
        {
            _DebugOut( "VirtualAlloc: reserve failed\n",
                       SLE_WARNING + FStopOnRing3MemoryError );
            InternalSetLastError( ERROR_NOT_ENOUGH_MEMORY );
            return 0;
        }

        // If caller is just reserving, we're finished.
        if ( !(fdwAllocationType & MEM_COMMIT) )
            return lpvAddress;

        // Caller has specified MEM_COMMIT.
        if ( VxDCall(_PageCommit,lpvAddress>>12, sizeInPages, 1, 0, pcFlags))
            return lpvAddress;      // Success!

        // Oops. Something went wrong. Tell the user, then fall through
        // to the code to free the pages.
        _DebugOut( "VirtualAlloc: commit failed\n",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_NOT_ENOUGH_MEMORY );
    }
    else    // Caller specified a particular address to allocate/commit at.
    {
        if ( address > 0xBFFFFFFF )
        {
```

```
    _DebugOut( "VirtualAlloc: bad base address\n\r",
               SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_ADDRESS );
    return 0;
}

fReserve = fdwAllocationType & MEM_RESERVE;
if ( fReserve )
{
    // Call VMM _PageReserve to allocate the memory. Note that
    // the caller-specified lpvAddress is rounded down to the
            // nearest 4KB page. Note that it's not down to 64KB like
            // the doc says. However, _PageReserve still rounds it down.
            lpvAddress=VxDCall(_PageReserve,address>>12, sizeInPages,pcFlags);
            if ( lpvAddress == -1 )
            {
                _DebugOut( "VirtualAlloc: reserve failed\n",
                           SLE_WARNING + FStopOnRing3MemoryError );
                InternalSetLastError( ERROR_NOT_ENOUGH_MEMORY );
                return 0;
            }

            // Hmmm...It turns out that KERNEL32 will complain if you
            // didn't specify an address aligned on a 64KB boundary!
            if ( lpvAddress != (address & 0xFFFF0000) )
                _DebugOut("VirtualAlloc: reserve in wrong place 1\n\r",
                           SLE_ERROR);
        }

        if ( !(fdwAllocationType & MEM_COMMIT) )
            return lpvAddress;

        lpvAddress &= 0xFFFFF000;

        if ( VxDCall(_PageCommit,lpvAddress>>12, sizeInPages, 1, 0, pcFlags) )
            return lpvAddress;
        else
        {
            _DebugOut( "VirtualAlloc: commit failed\n",
                       SLE_WARNING + FStopOnRing3MemoryError );
            InternalSetLastError( ERROR_NOT_ENOUGH_MEMORY );
            if ( !fReserve )
                return 0;
        }
    }

    // Unreserve the memory allocated earlier.
    VxDCall( _PageFree, lpvAddress & 0xFFFF0000, 0 );

return_0:
    lpvAddress = 0;

return_lpvAddress:
    return lpvAddress;
```

mmPAGEToPC

mmPAGEToPC 是由 VirtualAlloc，VirtualProtectEx 以及扩展函数 VirtualProtect 使用的。该函数将 PAGE_标志从 WINNNT.H(如 PAGE_READONLY)转换成等效的 PC_标志。PC_标志(页面提交)是在 VMM.INC 中定义的，并由 VMM 的_PageCommit 函数使用。

由 Windows 95 使用的标志之一表明某个特殊的页面是一个警戒页(guard page)，操作系统需要在栈的底部提交附加的内存以允许该栈可以向下发展。然而，从表面上看，你并不能利用 VirtualAlloc 申请某个警戒页面，因为 mmPageToPC 已将 PAGE_GUARD 位过滤掉。该函数同时还通过关闭方法忽略了 PAGE_NOCACHE 标志。对于除 PAGE_NOACCESS 外的所有情况，被转换的标志均包括 PC_USER 位，该位表明页面可由 ring 3 代码(用户水平)访问。如果该页面是可写入的，则 PC_WRITEABLE 标志被返回。换一个角度讲，除 PAGE_NOACCESS 以外，所有 PAGE_标志均映射到 PC_USER 或 PC_USER|PC_WRITEABLE。任何位，而不是对应于 PAGE_的标志，才造成了 mmPageToPC 在调试方案中的响应并造成 VirtualAlloc 或 VirtualProtect(Ex)的调用失效。

mmPAGEToPC 伪码

```
// Parameters:
//  DWORD   PAGE_flags;
// Locals:
//  DWORD   retValue;

    if ( PAGE_flags & PAGE_GUARD )
    {
        _DebugOut( "mmPAGEToPC: PAGE_GUARD flag not supported\n"
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_CALL_NOT_IMPLEMENTED );

        return -1;
    }

    PAGE_flags &= ~PAGE_NOCACHE;        // Turn off the PAGE_NOCACHE flag.
    if ( PAGE_flags == PAGE_NOACCESS )
        return 0;
    if ( PAGE_flags == PAGE_READONLY )
        return PC_USER;

    if ( PAGE_flags == PAGE_READWRITE )
        return PC_USER | PC_WRITEABLE;

    if ( PAGE_flags == PAGE_EXECUTE )
        return PC_USER;
```

```
    if ( PAGE_flags == PAGE_EXECUTE_READ )
        return PC_USER;

    if ( PAGE_flags == PAGE_EXECUTE_READWRITE )
        return PC_USER | PC_WRITEABLE;

    if ( PAGE_flags == PAGE_EXECUTE_WRITECOPY )
        return PC_USER;

    _DebugOut( "mmPAGEToPC: extra fdwProtect flags\n",
               SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_PARAMETER );
    return -1;
```

VirtualFree

VirtualFree 实施了 VirtualAlloc 的镜面影象功能。它可以将页的状态从保留状态改变为提交状态，从提交状态改变为保留状态，或从保留状态改变为释放状态。VirtualAlloc 的第一部分来检查是否通过了有效的地址和尺寸参数。地址必需低于 3GB，而尺寸必需小于 2GB-4MB 的数值(即专用应用程序区的尺寸)。

你可以将 MEM_RELEASE 或 MEM_DECOMMIT 标志置于 VirtualFree 函数，但只能用其中之一。MEM_RELEASE 使 VirtualFree 调用 VMM 的_PageFree 函数来下发或释放所有页面区域。对于这种模式，你必需以 0 作为通过尺寸，它使_PageFree 释放了原来整个由 VirtualAlloc 配置的页面块。MEM_DECOMMIT 的通过使得 VirtualFree 调用了 VMM 的_PageDecommit 函数来下发特定的页面块。

VirtualFree 的伪码

```
// Parameters:
//  LPVOID  lpvAddress
//  DWORD   cbSize
//  DWORD   fdwFreeType
// Locals:
//  DWORD   decommitPageSize

    // Is range to free bigger than 2GB-4MB? Fail if so.
    if ( cbSize > 0x7FC00000 )
    {
        _DebugOut( "VirtualFree: dwSize too big\n\r",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_INVALID_ADDRESS );
        return 0;
    }

    // Are pages in VxD land? If so, something's wrong.
    if ( lpvAddress > 0xBFFFFFFF )
    {
```

95DDK

95ddk\INC32

文件 VMM.H

```
#define UNDEFINED_DEVICE_ID 0x00000
#define VMM_DEVICE_ID       0x00001 /* Used for dynalink table */
#define DEBUG_DEVICE_ID     0x00002
#define VPICD_DEVICE_ID     0x00003
#define VDMAD_DEVICE_ID     0x00004
#define VTD_DEVICE_ID       0x00005
#define V86MMGR_DEVICE_ID   0x00006
#define PAGESWAP_DEVICE_ID  0x00007
#define PARITY_DEVICE_ID    0x00008
#define REBOOT_DEVICE_ID    0x00009
#define VDD_DEVICE_ID       0x0000A
#define VSD_DEVICE_ID       0x0000B
#define VMD_DEVICE_ID       0x0000C
#define VKD_DEVICE_ID       0x0000D
#define VCD_DEVICE_ID       0x0000E
#define VPD_DEVICE_ID       0x0000F
#define BLOCKDEV_DEVICE_ID  0x00010
#define VMCPD_DEVICE_ID     0x00011
#define EBIOS_DEVICE_ID     0x00012
#define BIOSXLAT_DEVICE_ID  0x00013
#define VNETBIOS_DEVICE_ID  0x00014
#define DOSMGR_DEVICE_ID    0x00015
#define WINLOAD_DEVICE_ID   0x00016
#define SHELL_DEVICE_ID     0x00017
#define VMPOLL_DEVICE_ID    0x00018
#define VPROD_DEVICE_ID     0x00019
#define DOSNET_DEVICE_ID    0x0001A
#define VFD_DEVICE_ID       0x0001B
#define VDD2_DEVICE_ID      0x0001C /* Secondary display adapter */
#define WINDEBUG_DEVICE_ID  0x0001D
#define TSRLOAD_DEVICE_ID   0x0001E /* TSR instance utility ID */
#define BIOSHOOK_DEVICE_ID  0x0001F /* Bios interrupt hooker VxD */
#define INT13_DEVICE_ID     0x00020
#define PAGEFILE_DEVICE_ID  0x00021 /* Paging File device */
#define SCSI_DEVICE_ID      0x00022 /* SCSI device */
#define MCA_POS_DEVICE_ID   0x00023 /* MCA_POS device */
#define SCSIFD_DEVICE_ID    0x00024 /* SCSI FastDisk device */
#define VPEND_DEVICE_ID     0x00025 /* Pen device */
#define APM_DEVICE_ID       0x00026 /* Power Management device */
#define VPOWERD_DEVICE_ID   APM_DEVICE_ID   /* We overload APM since we replace it */
#define VXDLDR_DEVICE_ID    0x00027 /* VxD Loader device */
#define NDIS_DEVICE_ID      0x00028 /* NDIS wrapper */
#define BIOS_EXT_DEVICE_ID   0x00029 /* Fix Broken BIOS device */
#define VWIN32_DEVICE_ID        0x0002A /* for new WIN32-VxD */
#define VCOMM_DEVICE_ID         0x0002B /* New COMM device driver */
#define SPOOLER_DEVICE_ID       0x0002C /* Local Spooler */
#define WIN32S_DEVICE_ID    0x0002D /* Win32S on Win 3.1 driver */
#define DEBUGCMD_DEVICE_ID      0x0002E /* Debug command extensions */
/* #define RESERVED_DEVICE_ID   0x0002F /* Not currently in use */
/* #define ATI_HELPER_DEVICE_ID    0x00030 /* grabbed by ATI */

/* 31-32 USED BY WFW NET COMPONENTS     */
/* #define VNB_DEVICE_ID           0x00031 /* Netbeui of snowball */
/* #define SERVER_DEVICE_ID        0x00032 /* Server of snowball */

#define CONFIGMG_DEVICE_ID  0x00033 /* Configuration manager (Plug&Play) */
#define DWCFGMG_DEVICE_ID   0x00034 /* Configuration manager for win31 and DOS */
#define SCSIPORT_DEVICE_ID  0x00035 /* Dragon miniport loader/driver */
#define VFBACKUP_DEVICE_ID  0x00036 /* allows backup apps to work with NEC */
#define ENABLE_DEVICE_ID    0x00037 /* for access VxD */
#define VCOND_DEVICE_ID     0x00038 /* Virtual Console Device - check vcond.inc */
/* 39 used by WFW VFat Helper device */

/* 3A used by WFW E-FAX */
/* #define EFAX_DEVICE_ID   0x0003A /* EFAX VxD ID      */

/* 3B used by MS-DOS 6.1 for the DblSpace VxD which has APIs */
/* #define DSVXD_DEVICE_ID  0x0003B /* Dbl Space VxD ID */

#define ISAPNP_DEVICE_ID    0x0003C /* ISA P&P Enumerator */
#define BIOS_DEVICE_ID      0x0003D /* BIOS P&P Enumerator */
/* #define WINSOCK_DEVICE_ID       0x0003E  /* WinSockets */
/* #define WSIPX_DEVICE_ID     0x0003F  /* WinSockets for IPX */

#define IFSMgr_Device_ID    0x00040 /* Installable File System Manager */
#define VCDFSD_DEVICE_ID    0x00041 /* Static CDFS ID */
#define MRCI2_DEVICE_ID     0x00042 /* DrvSpace compression engine */
#define PCI_DEVICE_ID       0x00043 /* PCI P&P Enumerator */
#define PELOADER_DEVICE_ID  0x00044 /* PE Image Loader */
#define EISA_DEVICE_ID      0x00045 /* EISA P&P Enumerator */
#define DRAGCLI_DEVICE_ID   0x00046 /* Dragon network client */
#define DRAGSRV_DEVICE_ID   0x00047 /* Dragon network server */
#define PERF_DEVICE_ID      0x00048 /* Config/stat info */

#define AWREDIR_DEVICE_ID   0x00049 /* AtWork Network FSD */

/*
 *   Far East DOS support VxD ID
 */

#define ETEN_Device_ID      0x00060 /* ETEN DOS (Taiwan) driver */
#define CHBIOS_Device_ID    0x00061 /* CHBIOS DOS (Korean) driver */
#define VMSGD_Device_ID    0x00062 /* DBCS Message Mode driver */
#define VPPID_Device_ID     0x00063 /* PC-98 System Control PPI */
#define VIME_Device_ID      0x00064 /* Virtual DOS IME */
#define VHBIOSD_Device_ID   0x00065 /* HBIOS (Korean) for HWin31 driver */
```

```
        _DebugOut( "VirtualFree: bad base address\n\r",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_INVALID_ADDRESS );
        return 0;
    }

    if ( fdwFreeType == MEM_RELEASE )
    {
        if ( cbSize != 0 )
        {
            _DebugOut( "VirtualFree: dwSize must be 0 for MEM_RELEASE\n\r",
                       SLE_WARNING + FStopOnRing3MemoryError );
            InternalSetLastError( ERROR_INVALID_PARAMETER );
            return 0;
        }

        // Unreserve the range of memory.
        return VxDCall( _PageFree, lpvAddress, 0 );
    }

    if ( fdwFreeType == MEM_DECOMMIT )
    {
        if ( cbSize == 0 )
        {
            _DebugOut( "VirtualFree: dwSize == 0 not allowed with
MEM_DECOMMIT\n\r",
                       SLE_WARNING + FStopOnRing3MemoryError );
            return 1;
        }

        // Calculate how many pages will be affected.
        decommitPageSize = lpvAddress & 0x00000FFF;
        decommitPageSize += cbSize;
        decommitPageSize += 0x00000FFF;
        decommitPageSize = decommitPageSize >> 12;

        return VxDCall( _PageDecommit, lpvAddress >> 12, decommitPageSize, 0);
    }

    _DebugOut( "VirtualFree: bad dwFreeType\n\r",
               SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_PARAMETER );
    return 0;
```

VirtualQueryEx

VirtualQueryEx 可能是 Windows 95 中最好的函数之一。它提供了有关某个特定地址的内存形式的大量信息。例如，在某个进程的地址空间中给定任意地址，VirtualQueryEx 会通知你哪一个 EXE 或 DLL 拥有该内存。VirtualQueryEx 处于 Windows NT PWALK 程序的中心，显示了对于给定进程的内存设计的映射。

VirtualQueryEx 最初并没有准备用在 Windows 95 Win32 的子集中。它的引入，给系统级编程工具如调试程序的设计者带来了冲击。但幸运的是，Windows 95 的设计者注意到了这一点，并将 VirtualQueryEx 包括进 Windows 95 的 API 中，这或许可能是因为部分地考虑到了大家的要求。

VirtualQueryEx 以有关某个特定地址的信息填充一个 MEMORY_BASIC_INFORMATION 结构。该结构形式如下：

```
PVOID BaseAddress;
PVOID AllocationBase;
DWORD AllocationProtect;
DWORD RegionSize;
DWORD State;
DWORD Protect;
DWORD Type;
```

该结构区域在 Win32 资源中详细介绍。但是，这里要对其中一个区加以进一步说明。AllocationBase 区听起来相当空洞，但它常常是最重要的一个区。从技术上讲，它包括了由 VirtualQueryEx 配置的原始内存区域的基本地址。更重要的是，当指向 VirtualQueryEx 的参数 lpvAddress 落入 EXE 或 DLL 模块的任何地方，AllocationBase 均是该 EXE 或 DLL 的基本地址。即，AllocationBase 与 EXE 或 DLL 的 HMODULE/HINSTANCE 是相同的。来自 NT SDK 的 PWALK 程序利用这一个位在某一进程的地址空间内行走并以它们拥有的 EXE 或 DLL 的名称来标记各种区域。调试程序可以利用该功能来计算出哪一个 EXE 或 DLL 与某个失效的地址相关联。

从本质上讲，VirtualQueryEx 就是对 VWIN32. VXD 的 Win32 服务 40h (VxDCall 0002A0040)的一个调用。该服务则反过来调用 VMM_PageQuery 函数。在 DDK 中，_PageQuery 被描述成一个指向 MEMORY_BASIC_INFORMATION 的参数。或许是为了防止某个不适当的线程开关在 MEMORY_BASIC_INFORMATION 结构中返回不恒定的数值，VirtualQueryEx 在入口处抓住了 Krn32Mutex 并在出口处释放此 mutex。这项工作是通过非资源的 KERNEL32_EnterSysLevel 和_LeaveSysLevel 函数来完成的。

VWIN32 服务 43h 填充了 MEMORY_BASIC_INFORMATION 结构，它不仅仅是一个_PageQuery 调用的外壳。这里我尚无法讲清它的具体工作过程。但是，事实表明，该外壳需要知道正在查询的进程中当前线程的 ring 0 栈的地址。因此，在调用 VWIN32 服务之前，VirtualQueryEx 利用 hProcess 参数来得到一个指向进程结构的指针(详见第六章)。由此开始，VirtualQueryEx 析取进程的当前线程的线程库来传给 Win32 服务。有趣的是，在 VWIN32 服务 43h 的几个步骤中，我还没有发现代码不调用_PageQuery 的情况。

VirtualQueryEx 伪码

```
// Parameters:
//  HANDLE  hProcess;
//  LPCVOID lpvAddress; // Address of region.
//  PMEMORY_BASIC_INFORMATION  pmbiBuffer;  // Address of information buffer.
//  DWORD   cbLength;   // Size of buffer.
// Locals:
//  DWORD   pProcess;   // Pointer to process structure.
//  DWORD   ptdb;       // Per-thread database.
//  DWORD   retValue;
   // Function that emits function names and parameters to the KERNEL
   // debugger if a KERNEL32 global variable is TRUE (off by default).
   x_LogKernelFunction( number indicating the VirtualQueryEx function );

   _EnterSysLevel( Krn32Mutex );

   retValue = 0;

   pProcess = x_GetObject( hProcess, 0x80000010, 0 );

   if ( pProcess )
   {
       if ( ppCurrentProcessId == pProcess )
           ptdb = ppCurrentThreadId;
       else
           ptdb = SomeFunction( pProcess->threadList, 0 );

       if ( ptdb && (lpvAddress < 0xC0000000) )
       {
           // Call into the VWIN32 VxD to do the real work.
           // VWIN32 ultimately calls the VMM _PageQuery function.
           retValue = VxDCall( 0x002A0040, ptdb->ring0_hThread,
                               lpvAddress, pmbiBufer, cbLength );
       }

       x_UnuseObjectSafeWrapper( pProcess );
   }

   _LeaveSysLevel( Krn32Mutex );

   return retValue;
```

VirtualQuery 和 IVirtualQuery

VirtualQuery 函数是 VirtualQueryEx 函数的一个特例。VirtualQuery 恢复了当前进程文本中有关特定地址的信息，而 VirtualQueryEx 则对任何进程均起作用。

VirtualQuery 代码几乎不带有数值，它只是一个参数有效层(parameter validation layer)。VirtualQuery 的代码仅仅检测进入缓冲区的指针是否可以足够大地容纳 MEMORY_BASIC_INFORMATION。假定测试是成功的，VirtualQuery 则跳

到 IVirtualQuery 代码的开始位置。在跳到某个进行实际操作的公共进程之前，参数的 VirtualQuery 有效性确认对 DLL 系统中(如后面要谈到的 VirtualProtect)的许多函数都是很典型的。

除了调试方案中的一些登录代码，IVirtualQuery 只不过是以当前进程的伪句柄作为第一个参数来调用 VirtualQueryEx 的。注意到 Windows 95 中，IVirtualQuery 是调用 VirtualQueryEx 的。而在 Win32s 中，VirtualQueryEx 只是对 VirtualQuery 的一个调用。其中的关键区别是，在 Win32s 中所有的进程共享同一地址空间，因而 VirtualQuery 应当与 VirtualQueryEx 等效。

VirtualQuery 的伪码

```
// Parameters:
//  LPCVOID lpvAddress; // Address of region.
//  PMEMORY_BASIC_INFORMATION  pmbiBuffer;  // Address of information buffer.
//  DWORD   cbLength;   // Size of buffer.

    Set up structured exception handler frame

    // Make sure that the beginning and end of the MEMORY_BASIC_INFORMATION
    // structure is acccessible.
    *(PBYTE)pmbiBuffer += 0;
    *(PBYTE)(pmbiBuffer+0x1B) += 0;

    Remove structured exception handler frame

    goto IVirtualQuery;
```

IVirtualQuery 的伪码

```
// Parameters:
//  LPCVOID lpvAddress; // Address of region.
//  PMEMORY_BASIC_INFORMATION  pmbiBuffer;  // Address of information buffer.
//  DWORD   cbLength;   // Size of buffer.

    // Function that emits function names and parameters to the KERNEL
    // debugger if a KERNEL32 global variable is TRUE (off by default).
    x_LogKernelFunction( number indicating the VirtualQuery function );

    // Let VirtualQueryEx do the real work. 0x7FFFFFFF is the process
    // pseudohandle that GetCurrentProcess() would return.
    return VirtualQueryEx( 0x7FFFFFFF, lpvAddress, pmbiBuffer, cbLength );
```

VirtualProtectEx

VirtualProtectEx 改变了页面区的某个被提交页面的访问保护。它对于任何具有进程句柄的进程均生效。VirtualProtectEx 与 VirtualAlloc 之间的关键区别是，

VirtualProtectEx 认定你已经提交了你正在改变访问属性的页面。而 VirtualAlloc 则允许你配置、提交、并设定页面的访问权。

VirtualProtectEx 的代码是向前递增的。正如我所介绍的其它虚拟函数一样，它以查错开始。代码确认要修改的地址是低于 2GB-4MB 的，而且起始地址低于 0xC0000000。VirtualProtectEx 的中心是对 VWIN32 服务 0x3F 的调用。该服务最终要调用 VMM. VXD 的_PageModifyPermission 服务。由于是在 VirtualProtectEx 中，再加上某些原因，VWIN32 的调用期待一个指向特定进程当前线程的 ring 0 栈的指针。这里存在大量的代码用于确定该 ring 0 栈是否与我们在 VirtualProtectEx 发现的栈一致。还由于利用了 VirtualQueryEx，VirtualProtectEx 在 VWIN32 调用当中抓住并控制了 Krn32Mutex。

当调用成功时，VWIN32 服务的 0x3F 调用返回了被改变页面的前一状态。然而，该状态是根据 VMM 的 PC_flags 标志来给定的，而非调用程序所期待 VirtualProtectEx 进行快速转换的 PAGE_style 标志。最后，如果调用程序设定了一个指向旧页面属性空间的指针，则代码将 PAGE_flags 拷贝到那个位置中去。

VirtualProtectEx 伪码

```
// Parameters:
//  HANDLE  hProcess;
//  LPVOID  lpvAddress;     // Address of region of committed pages.
//  DWORD   cbSize;         // Size of the region.
//  DWORD   fdwNewProtect;  // Desired access protection.
//  PDWORD  pfdwOldProtect; // Address of variable to get old protection.
// Locals:
//  DWORD   pcFlags;        // Returned from mmPAGEToPC.
//  DWORD   pProcess, ptdb;
//  DWORD   oldProtectFlags

    // Function that emits function names and parameters to the KERNEL
    // debugger if a KERNEL32 global variable is TRUE (off by default).
    x_LogKernelFunction( number indicating the VirtualProtectEx function );
    if ( cbSize > 0x7FC00000 )
    {
        _DebugOut( "VirtualProtect: dwSize too big\n\r",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_INVALID_ADDRESS );
        return 0;
    }

    if ( lpvAddress > 0xBFFFFFFF )
    {
        _DebugOut( "VirtualProtect: bad base address\n\r",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_INVALID_ADDRESS );
        return 0;
    }
```

```
pcFlags = mmPAGEtoPC( fdwNewProtect );
if ( pcFlags == -1 )                    // Were invalid flags passed?
    return 0;

_EnterSysLevel( Krn32Mutex );

pProcess = x_GetObject( hProcess, 0x80000010, 0 );
if ( !pProcess )
{
    _LeaveSysLevel( Krn32Mutex );
    _DebugOut( "VirtualProtectEx: Invalid process handle\n",
               SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_PARAMETER );
    return 0;
}

if ( pProcess == ppCurrentProcessId )
    ptdb = ppCurrentThreadId;
else
    ptdb = SomeFunction( pProcess->threadList, 0 );

if ( ptdb && (lpvAddress < 0xC0000000) )
{
    // Call into the VWIN32 VxD to do the real work. The VWIN32
    // service calls VMM's _PageModifyPermissions.
    oldProtectFlags = VxDCall( 0x002A003F, ptdb->ring0_hThread,
                               lpvAddress, cbSize, 0, pcFlags );
}
else
{
    oldProtectFlags = some uninitialized local variable;  // ???
}

x_UnuseObjectSafeWrapper( pProcess );

_LeaveSysLevel( Krn32Mutex );

if ( oldProtectFlags == -1 )
{
    _LeaveSysLevel( Krn32Mutex );
    _DebugOut( "VirtualProtect: ModifyPagePermission failed\n",
               SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_PARAMETER );
    return 0;
}

// This section is sort of a quick-and-dirty "PCPAGETomm". It converts
// the PC_ flags returned by the VWIN32 service into MEM_ flags.
if ( oldProtectFlags & PC_USER )    // PC_USER flag set
{
    if ( oldProtectFlags & PC_WRITEABLE )
        oldProtectFlags = PAGE_READWRITE;
    else
        oldProtectFlags = PAGE_READONLY;
```

```
}
else                                    // PC_USER flag not set
    oldProtectFlags = PAGE_NOACCESS;

// If the caller specified a pointer to a DWORD as the last param,
// fill it in with the old flag's value.
if ( pfdwOldProtect )
   *pfdwOldProtect = oldProtectFlags;
```

VirtualProtect 和 IVirtualProtect

VirtualProtect 是 VirtualProtectEx 的简化版本，只能运行于当前进程。VirtualProtect 代码仅仅是带有 IVirtualProtect 真实代码的有效层。VirtualProtect 中实施的唯一有效性确认(与 VirtualProtectEx 的检查相应)是确定 pfdwOldProtect 指针是否是一个有效的 DWORD 指针或 0。

IVirtualProtect 代码是对 VirtualProtectEx 调用的一个外壳。它所传递的 hProcess(h 进程)是一个代表当前进程(0x7FFFFFFF)的伪句柄。在调试方案中，IVirtualProtect 同样调用某个函数，该函数要登录对于调试终端的某个 API 调用。

VirtualProtect 的伪码

```
// Parameters:
//  LPVOID  lpvAddress;     // Address of region of committed pages.
//  DWORD   cbSize;         // Size of the region.
//  DWORD   fdwNewProtect;  // Desired access protection.
//  PDWORD  pfdwOldProtect; // Address of variable to get old protection.

   Set up structured exception handler frame

   // If nonzero, verify that the pointer to DWORD where the previous
   // protection flags will be stored is valid.
   if ( pfdwOldProcect )
      EAX = *pfdwOldProtect;

   Remove structured exception handler frame

   goto IVirtualProtect;
```

IVirtualProtect 的伪码

```
// Parameters:
//  LPVOID  lpvAddress;     // Address of region of committed pages.
//  DWORD   cbSize;         // Size of the region.
//  DWORD   fdwNewProtect;  // Desired access protection.
//  PDWORD  pfdwOldProtect; // Address of variable to get old protection.

   // Function that emits function names and parameters to the KERNEL
   // debugger if a KERNEL32 global variable is TRUE (off by default).
```

```
x_LogKernelFunction( number indicating the VirtualProtect function );

// Let VirtualProtectEx do the real work. 0x7FFFFFFF is the value that
// GetCurrentProcess() would return.
return VirtualProtectEx( 0x7FFFFFFF, lpvAddress, cbSize, fdwNewProtect,
                         pfdwOldProtect );
```

VirtualLock 和 VirtualUnlock

VirtualLock 和 VirtualUnlock 函数并不由 Windows 95 提供。在支持这些函数的 Win32 平台中，它们允许一个进程将一系列的页面锁定。系统来保证物理 RAM 将总是设定于这些页面。这一点，在你无法提供一个缺页中断(如实时设备驱动器)时将会很有用。

在 Windows 95 中，VirtualLock 和 VirtualUnlock 都跳到了 CommonUnimpStub 代码。CommonUnimpStub 是所有的非执行 Win32 API 指定通过的代码的一个窄区。CommonUnimpStub 的效果是双倍的。首先，在调试方案中，KERNEL32 向调试终端发出一个诊断。例如：

```
*** Unimplemented Win32 API: VirtualLock
```

其次，CommonUnimpStub 将适量的参数从栈中清除。对于 VirtualLock/Unlock，它是 8 字节的。因为 CommonUnimpStub 处理着带有各种参数的 API，要马上移开的字节数需要给到 CommonUnimpStub。这是通过置于 CL 寄存器中的一个数值来完成的，该数值是编码的位字段，而非要移开的字节数。

VirtualLock 的伪码

```
EAX = "VirtualLock"
CL = 12
JMP CommonUnimpStub
```

VirtualUnlock 的伪码

```
EAX = "VirtualUnlock"
CL = 12
JMP CommonUnimpStub
```

Win32 堆函数(HEAP FUNCTIONS)

在 Win32 中，Microsoft 最终还是将相当好的堆管理代码放入了操作系统。DOS 内存配置表生成的块通常过大，而且作为一个堆函数使用也太慢。在 Win16 中，GlobalAlloc 函数有一个最小的配置尺寸，为 20h 字节，而且过快地用掉了 8192

个选择器。Win16 的 LocalHeap 函数在某种程度上讲，更适合于小型的配置，对于 64KB 的堆配置是受限的。另外，函数中也不支持泄漏跟踪或内存溢出检测。

Win32 堆函数远优于 Microsoft 先前的操作系统的配置方案。对于零售版的 Windows 95，每个块只为 4 字节，你可以生成最大理论尺寸为 2GB-4MB 的堆。另外，Windows 95 的 Win32 堆为各种尺寸的块保存 4 个分开的自由列表，以防止多余的内存碎片。另一个优点是其堆只在调试版本中出现。对于这种模式，每个配置的块均以附加信息作为标志，从而便于你很容易地发现溢出，内存外漏，并知道谁分配了内存。在后面 Windows 95 的调试版本中，在线使用块方面的内容更多地介绍了这些附加信息是如何使用的。但不幸的是，实施堆块溢出检查的唯一办法是通过使用一个暗物(obscure)，即 Windows 95 的唯有函数——HeapSetFlags。据我所知，在写此书时，这个函数尚未出现在 Microsoft 的任何资料中过，但它必然会引起重视。在 1995 年 9 月出版的 Microsoft System's Journal 中曾介绍过该函数。(在这本书里，由于我找到该函数太迟而未能包括进来)。

除了这些巧妙之处外，Windows 95 允许应用程序在同一进程中支持多个堆。这样，就使我们很方便地在一个堆中编排所有的内存配置。(这常常是避免堆碎片的一个好办法。)由于 Windows 95 支持了多个堆，你必需总要将某个堆柄置于 Win32 的堆函数中去。此堆柄来验明哪一个堆可以让你使用，它实际上只不过是某特定堆的起始线性地址。

另外一个优点是，如果你愿意的话，Windows 95 的堆可以超出初始保留尺寸而生成。此时，KERNEL 32 将分配线性地址空间的附加块并将此块与堆相连。我称这些附加的内存块为子堆(sunheap)。图 5-7 显示了含有多个堆的复杂的建立过程，而且其中某些堆使用了子堆。

Win32 堆函数的列表包括 HeapAlloc、HeapFree、HeapReAlloc。你可能会认为，正是这些基本的函数为那些需要提供 malloc、realloc、free、new、delete 函数的编译商创造了一个好的机会。事实并非如此。Borland 和 Microsoft 在他们的运行库中均忽略了此 Win32 堆函数。一个值得注意的例外是 Win32 的 SDK 运行库(CRTDLL. DLL)。CRTDLL. DLL 中的 malloc 和 free 函数均使用了 HeapAlloc 和 HeapFree 函数。Windows NT 和 Windows 95 中均有不同版本的 CRTDLL。

更正：在此书将要发行之际，我发现 Visual C++4. 0 在其 C/C++运行堆中使用了 Win32 堆函数。

在 Windows 95 的 Win32 堆服务之上的一层，你将发现 GlobalAlloc 和 LocalAlloc 函数。GlobalAlloc 和 LocalAlloc 是根据 HeapAlloc 系列函数配置的，尽管 LocalAlloc 并不仅仅是 HeapAlloc 的一个外壳。原因是 Win16 的程序员对 LocalAlloc 化的块作了某些不适的处理，使得 LocalAlloc 的 Windows 95 Win32 版本需要保留某些落后的兼容性。这点在后面的"The Win32 Local and Global HeapFunction"中要详细谈到。Windows95's Win32 堆函数之下的一层，代码直接使用了 VMM 提供的内存管理 Win32 VxD 服务。然而，这与提到的虚拟函数的功能并无两样。因此，我将 Win32 堆函数看做一个处于 Win32 虚拟内存函数之上的层。有趣的是，为 VxD 提供堆功能的 VMM HeapXXX 函数与 KERNEL32 ring 3 进程使用的堆结构的格式是相同的。

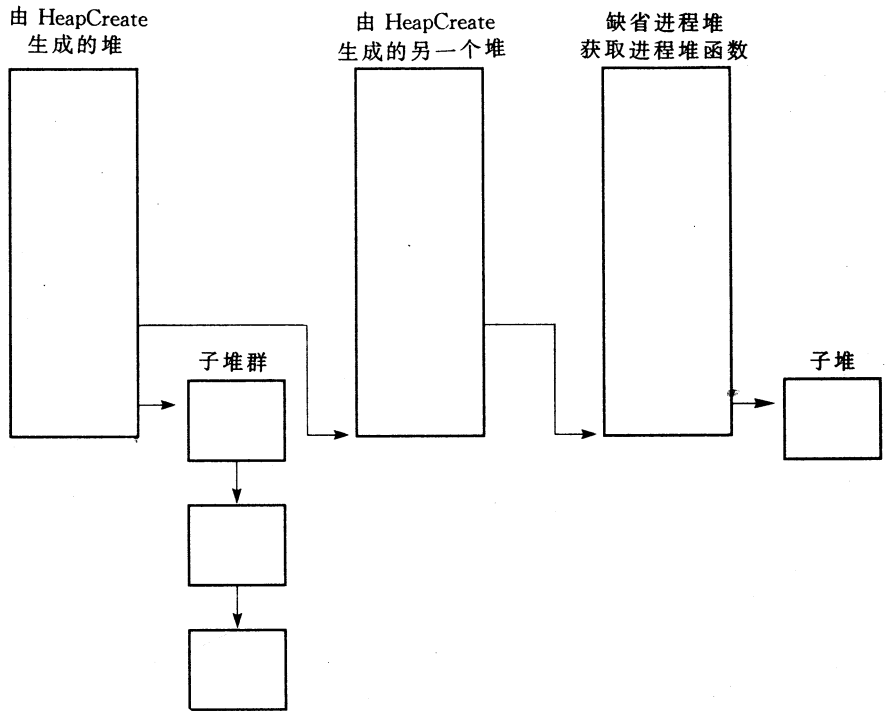


图 5-7　带有多个 Win32 堆的进程

Win32 堆首(heap head)与堆场(heap arenas)

所有 Windows 95 堆的分量均是由内存区域生成的，而该内存区域是由 VMM_PageReserve WIN32 VxD 服务所保留的，并分成两块。堆区的开始部为一个首部(header)，它包含了堆管理方面的信息(我们马上要涉及)，如自由列表，堆尺寸以及堆生成标志。紧接下来是堆内存块。每个堆块以某个场结构(arena structure)开始，该场结构含有下面块的信息。每个堆块的起始部分与前一块的末端相连。这些块扩展到了配置堆区的末端，尽管实际上并不是每个页面都要提交。图 5-8 是一个典型的堆设计。

如上所述，每个堆块，无论是自由的还是使用的，均以某个标准的场结构开始。场的格式在 Windows 95 的调试版本与零售版本之间是不同的。另外，如果某个块成为了自由块，将会出现附加的区域。这样就带来了场的 4 种不同形式：个别自由，个别在线使用，调试自由和调试在线。但第一个区是所有场所共有的。

每个堆场以某个含有该块尺寸的 DWORD 开始。该尺寸包括由场本身所占据的空间。但是，你并不能简单地将某个场中的第一个 DWORD 看成块尺寸。为什么呢？因为该场的第一个 DWORD 中的某些位已被与块尺寸无关的项目所使用。该 DWORD 的高字节总是 0xA0。0xA0 的意义尚不大清楚。我相它是某种位型，允许 KERNEL32 来告知某个场是否已被写过。因为所有堆块的尺寸总是一个 4 字节

的倍数，也就出现了其它并不表示块尺寸的位。这样就留下底部的两个位(值为 1 和 2)可以用作标志，其意义是：

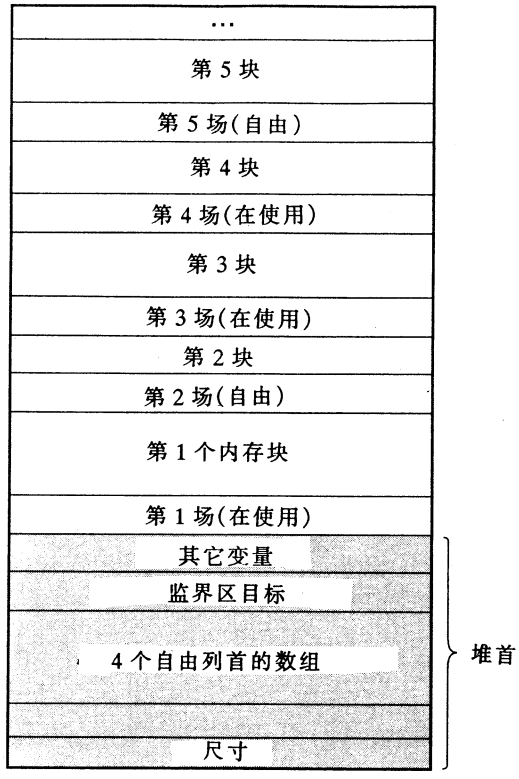


图 5-8　典型的 Win32 堆

1 — 表明该块为自由块。其位值为 0 时表明该块已配置。

2 — 表明该块前面的块是自由的。该位应该只能在已配置的块中结束设置。当当前块释放后，它可以与前面的自由块合并。如果该位未被设置，则前面的块不是自由的，因而没有必要将块合并。

将所有的位综合考虑，就很容易算出某个堆块的大小。简单地讲，只需以数值 0x5FFFFFFC 对场的第一个 DWORD 进行按位 AND 运算即可。这样就将 DWORD 中与尺寸无关的位关闭掉。更简单的办法是在 C 的符号中，利用～0xA0000003 对场首 DWORD 进行一逻辑 AND。为了通过调用程序来计算出该块中的多少内存可供使用，只需简单地将块的尺寸用场的尺寸替代即可。

零售版 Windows 95 的使用块

该块具有 4 种场型的最简单形式：

```
DWORD   size            // OR'ed with 0xA0000000 or 0xA0000002.
```

另外，这里所谓的场型不过是 DWORD 的初始尺寸/标志(size/flags)。

零售版 Windows 95 的自由块

自由块与使用块的起点是相同的，但它们分别加到了前一个和下一个区域。

```
DWORD   size            // OR'ed with 0xA0000001.
DWORD   prev            // Pointer to the previous heap arena.
DWORD   next            // Pointer to the next heap arena.
```

Windows 95 调试版本的使用块

某个使用的堆场起始形式与零售版相同，但其加到了附加区域。

```
DWORD   size            // OR'ed with 0xA0000000 or 0xA0000002.
DWORD   allocating EIP  // The EIP value that called HeapAlloc/HeapReAlloc.
WORD    thread number   // The thread number (not ID) that allocated the block.
WORD    signature       // 0x4842 == "BH"
DWORD   checksum        // A checksum of the previous three DWORDSs.
```

附加区域的目的是跟踪内存溢出以及堆破坏故障。配置了的 EIP 区域内存放了该块所分配的程序地址。它可用于探测在何处分配了在某种程度上并不自由的代码块。线程标号段(thread number field)的使用目的是类似的，但它用于明确哪一个线程分配了该标号段。注意该标号段与某个 ID 线程(即 GetCurrentTreadId 所返回的值)并不一样，它是进入当前线程列的一个索引。你可以通过 SoftIce/W THREAD 命令来观看线程标号。对于某个使用的块而言，标志 WORD 应总是 0x4842，否则，该场可能已被破坏。

场的最后区段提供了一个更强大的堆破坏克星。在该场中，区段包含了前面三个 DWORD 的检验和(checksum)。在本章后面所介绍的 CheckHeapBlock 中，将讨论其算法。尽管检验和通常是提供的，但调试 KERNEL32 并不自动实现该检验和——你必需通知它这样作。这种特点称为“妄想的堆破坏检查”(paranoid heap corruption checking)，并由 HeapSetFlags 函数来将其标为开或闭的状态。

```
hpWalk: bad busy block checksum trashed addr between 560014 and 560020
heap handle=460000
```

Windows 95 调试版本的自由块

Windows 95 的自由块场(free block arenas)是自由零售场(free retail arena)和使用调试场(in-use debug arena)的混合。像自由零售场一样，这里存在前一个和下一个区段。根据调试的使用场，这里有线程标号、标志、和检验和段。标志段变动轻微(从 0x4842 到 0x4846)，正如检验和的算法所述。另外还有多于使用版本的一个 DWORD，因而当 KERNEL32 对该场进行检验和运算时，它使用了最初的 4 个(而不是 3 个)DWORD。

```
DWORD   size            // OR'ed with 0xA0000000 or 0xA0000002.
DWORD   prev            // Pointer to the previous heap arena.
WORD    thread number   // The thread number (0xFEFE for free blocks).
WORD    signature       // 0x4846 == "FH"
DWORD   next            // Pointer to the next heap arena.
DWORD   checksum        // A checksum of the previous four DWORDS.
```

Windows 95 堆首(heap header)

每个堆的开始部分都是一个堆首结构。一个堆柄,如你从 GetProcessHeap 中返回的结果,只不过是一个指向堆首的指针。HeapCreate 函数(除了堆的保留内存)的最初工作是初始化该结构。对于 Windows 95 的零售版本和调试版本,堆首结构的尺寸是在变化的(但格式变化不大)。紧接着堆首结构的是第一个堆块场,已在上一节介绍过。

Windows 95 零售版本的堆首

00h　　WORD　　dwSize

指的是为该堆所保留的整个内存尺寸。为每个进程所生成的缺省的进程堆在该段具有 1MB+4KB 的大小。

04h　　DWORD　　nextBlock

如果以 dwMaximum 的尺寸参数为 0 来调用 HeapCreate,则该堆可以在为前一个 dwSize 段设定的配置尺寸之外生成。这时,如果调用程序提交的块对于当前的堆区来讲过大,KERNEL32 则保留多余的内存区域并建立子堆(subheaps)。子堆使用了堆场块,但没有一个完整的堆首结构。为了跟踪这些子堆,KERNEL32 将它们存于某个联结列表。该列表的首部存于初始堆结构的区段中(偏移量 4)。指向下一个保留区域的指针保留于每个子堆的偏移量 4(offset 4)。当该堆被破坏时,KERNEL32 则运行至子堆列表并将它们的页面释放回系统。

08h　　FREE_LIST_HEADER_RETAIL　　freeListArray[4]

将碎段最小化并加快寻找自由块,每个堆首保存有 4 个自由列表,它们分别是块尺寸小于 0x20 字节、小于 0x80 字节、小于 200h 字节和小于 0xFFFFFFFF 的自由列表。当寻找某个新的内存块时,KERNEL32 在最合适的自由列的起点开始寻找。例如,在寻找一个尺寸为 0x18 的块时,KERNEL32 首先寻找 0x20 字节和其下的列表。而在寻找某个 0x100h 字节的块时,它首先是寻找 0x200h 字节的自由列表。

4 个自由列表是以简单的数组结构形式来表示的。每个结构具有如下格式:

DWORD　该列的 maxBlockSize(最大块尺寸)。包括 0x20、0x80、0x200 或 0xFFFFFFFF

free arena　该场实际上是一个规则的零售自由场,除了块尺寸给定为 0 字节(在移走 0xA0000001 字节之后)。该场的前一个指针指向第一个自由场。对于该场而言,块尺寸为 0,因而搜寻算法可以很简单,但该场尚未配置。

48h　PVOID　nextHeap

零售版本 Windows 95 堆中的偏移量 48h 是一个指向下一个堆的指针,而该堆是由当前进程的 HeapCreate 生成的。注意,此下一个堆与偏移量 4 给定的下一个子堆是不同的。该区段中由一个非零指针指向的区域是一个成熟的堆(full-fledged heap)。该区段为零,除非进程调用了 HeapCreate。

4Ch　HCRITICAL_SECTION　hCriticalSection

该区段通过控制临界区域的句柄来协调对该堆的访问,而该句柄是由堆函数使用的。注意,该区段并不是 CRITICAL_SECTION 的本身(参看下一个区段),而是一个指向由 KERNEL32 使用的内部数据结构的指针。句柄值看似总与下面介绍的区段中的偏移量 0Ch 处的 DWORD 相匹配。

50h　CRITICAL_SECTION　criticalSection

堆首的该部分包括一个 CRITICAL_SECTION 结构(在 WINBASE.H 中定义)。当进入需要同步访问的代码时,KERNEL32 经过一个指向该区域的指针到达 EnterCriticalSection。在程序的起始阶段,该区段的结构数目通过调用 InitialCriticalSection 进行初始化。如果你无需同步(如你仅有一个线程),你可以通过指向 HeapAlloc,或 HeapCreate,或者是二者的 HEAP_NO_SERALIZE 标志,跳过这一步。

68h　DWORD　unknown1[2]

表明该区段是未知的。

70h　BYTE　flags

该 BYTE 包括可以通过 HeapCreate 的 HEAP_flags。

```
HEAP_NO_SERIALIZE
HEAP_GROWABLE
HEAP_GENERATE_EXCEPTIONS
HEAP_ZERO_MEMORY
HEAP_REALLOC_IN_PLACE_ONLY
HEAP_TAIL_CHECKING_ENABLED
HEAP_FREE_CHECKING_ENABLED
HEAP_DISABLE_COALESCE_ON_FREE
```

Windows 95 的有关资料只介绍了 HEAP_NO_SERIALIZE 和 HEAP_GENERATE_EXCEPTIONS。

71h　　BYTE　　unknown2

表明该字节是未知的。它可能是被保留了，以防需要附加的 HEAP_flags。

72h　　WORD　　signature

该 WORD 包括用于明确某个堆的标志。对于有用的 Windows 95 堆，它包括 0x4948("HI")。

Windows 95 调试版的堆首(Heap Header)

调试版的 Win32 堆首与零售版相当接近。但是，其嵌入的自由场结构更大一些，并且有几个附加区段。下面是调试堆首的设计。

00h　　DWORD　　dwSize

参看前面的说明。

04h　　DWORD　　nextBlock

参看前面的说明。

08h　　FREE_LIST_HEADER_DEBUG　　freeListArray[4]

除了自由场部分为一调试场而非零售场外，4 个自由列首的数组与零售版相同。其结构格式如下：

DWORD　　该列的最大块尺寸(maxBlockSize)。包括 0x20、0x80、0x200 或 0xFFFFFFFF

free arena　　该场是一个常规的调试自由场，除非块尺寸给定为 0 字节(在移走 0xA0000001 标志之后)

68h　　PVOID　　nextHeap

参看前面的说明。

6Ch　　HCRITICAL_SECTION　　hCriticalSection

参看前面的说明。

70h　　CRITICAL_SECTION　　criticalSection

参看前面的说明。

88h　　DWORD　　unknown1[14]

参看前面的说明。

C0h　　DWORD　　creating EIP

该 DWORD 控制着 EIP 例程，而该例程是来调用内部的 HPInit 函数来初始化堆的。它表面上总是置于 HeapCreate 调用 HPInit 的位置。

C4h　　DWORD　　checksum

该区段控制了堆首(尺寸段)的第一个 DWORD 与 0x17761965 进行 XOR 运算的结构。这样就有可能帮助 KERNEL32 检查在堆首的重写。

C8h　　WORD　　creating thread number

给出生成堆的线程的数目(而非线程 ID)。此内容参看前一节中在介绍调试块场时对线程数目的介绍。

CAh　　WORD　　unknown2

该 WORD 是不使用的。

CCh　　BYTE　　flags(HEAP_xxx FLAGS)

参看前面的说明。

CDh　　BYTE　　unkown3

参看前面的说明。

CEh　　WORD　　signature(0x4948)

参看前面的说明。

行走堆(WALKHEAP)程序

为了有效地展示 Windows 95 Win32 堆首和堆场，我写了 WALKHEAP 程序。WALKHEAP 的源代码在附带的磁盘上。WALKHEAP 包括两个程序文件：WALKHEAP.C，它包括了行走和显示 Win32 堆的代码，和 WALKHEAP.H，它包括了对于堆首和场结构的定义。WALKHEAP 程序需要在调试版的 Windows 95 上运行。磁盘上一个类似的程序(WALKHEAP2.EXE)可以在零售版的 Win32 堆上运行。当然，我也可以通过使用 32 位的 TOOLHELP32 函数在堆上运行，但无太大必要。TOOLHELP32 函数将某些有趣的细节隐埋了。

当不带任何命令行参数运行时，WALKHEAP 会行走并显示所有的堆。更有趣的是，WALKHEAP 首先利用缺省堆产生一系列配置和删除，并生成第二个堆。

WALKHEAP 可以通过使用堆首的 Next Heap 区段对所有进程的 Win32 堆累接。

如果你知道了你想行走的特定堆的地址(必需可由 WALKHEAP 访问),你可以在 WALKHEAP 的命令行上通过堆址(也叫句柄)。其数目必需以 16 位字节定义,并不带有任何 0x's 或 h's。例如,我可以这样行走至 USER 的 32 位窗堆:

WALKHEAP 81CEC000(地址可能与你的设备有所不同)

图 5-9 显示了不带任何参数运行 WALKHEAP 时的输出。块(Block)列向的数目是该块的线性地址。注意,堆中显示的最初 4 个块是自由列首(free list headers)且尺寸为 0。同时也要注意到,你可以通过跟随前一指针并以 4 个块(自由列首)之一为起点,行走至自由列表(free list)。

```
Heap at 00B60000
size:                                   00100000
next block:                             00000000
Free lists:
  Head:00B6000C  size: 20
  Head:00B60024  size: 80
  Head:00B6003C  size: 200
  Head:00B60054  size: FFFFFFFF
Next heap:                              00410000
CritSection:                            1066F7C6
Creating EIP:                           BFF8BAE0
checksum:                               17661965
Creating Thread:                        0040
Flags:                                  05
                                        HEAP_NO_SERIALIZE
                                        HEAP_GENERATE_EXCEPTIONS
Signature:                              4948
Heap Blocks
Block     Stat  Size      Checksum  Thrd
--------  ----  --------  --------  ----
00B6000C  free  00000000  FF3009F6  1066  prev:00B60024  next:00B600D0
00B60024  free  00000000  FF30692B  707F  prev:00B6003C  next:00B6000C
00B6003C  free  00000000  FF30F214  EB00  prev:00B60054  next:00B60024
00B60054  free  00000000  FF438FBB  66F7  prev:00C5F014  next:00B6003C

00B600D0  free  000FEF34  FF4CF8B6  FEFE  prev:00B6000C  next:00C5F014
00C5F004  used  00000010  FF341977  0000  EIP: 00000000
00C5F014  free  00000FDC  FF30E8C2  FEFE  prev:00B600D0  next:00B60054


Heap at 00410000
size:                                   00101000
next block:                             00760000
Free lists:
  Head:0041000C  size: 20
  Head:00410024  size: 80
  Head:0041003C  size: 200
  Head:00410054  size: FFFFFFFF
```

```
Next heap:                              00000000
CritSection:                            8153C074
Creating EIP:                           BFF8BAE0
checksum:                               17660965
Creating Thread:                        0040
Flags:                                  40
                                        HEAP_FREE_CHECKING_ENABLED
Signature:                              4948

Heap Blocks
Block     Stat  Size      Checksum  Thrd
--------  ----  --------  --------  ----
0041000C  free  00000000  FF201A5C  0000  prev:0051002C  next:00410314
00410024  free  00000000  FF3019DC  0000  prev:005100D8  next:00510060
0041003C  free  00000000  FF301A8C  0000  prev:005102A4  next:0051014C
00410054  free  00000000  FF30156C  0000  prev:00510850  next:00510458

004100D0  used  00000244  FF740EBC  0040  EIP: 004015DD
00410314  free  000FFCF0  FFD01B4E  FEFE  prev:0041000C  next:00AE0028
00510004  used  00000010  FF341977  0000  EIP: 00000000
00510014  used  00000018  FF740D41  0040  EIP: 0040147C
0051002C  free  00000018  FF20E7EE  FEFE  prev:00510060  next:0041000C
00510044  used  0000001C  FF740DA5  0040  EIP: 0040149E
00510060  free  00000020  FF20E7B2  FEFE  prev:00410024  next:0051002C
00510080  used  00000024  FF740DC3  0040  EIP: 004014C0
005100A4  used  00000034  FF740DC0  0040  EIP: 004014D1
005100D8  free  00000038  FF20E6CA  FEFE  prev:0051014C  next:00410024
00510110  used  0000003C  FF740DE8  0040  EIP: 004014F3
0051014C  free  00000040  FF20E73E  FEFE  prev:0041003C  next:005100D8
0051018C  used  00000044  FF740C76  0040  EIP: 00401515
005101D0  used  000000D4  FF740CD8  0040  EIP: 00401529
005102A4  free  000000D8  FF20E326  FEFE  prev:00510458  next:0041003C
0051037C  used  000000DC  FF740CAA  0040  EIP: 00401551
00510458  free  000000E0  FF20E58A  FEFE  prev:00410054  next:005102A4
00510538  used  000000E4  FF740CBA  0040  EIP: 00401579
0051061C  used  00000234  FF740E9C  0040  EIP: 0040158D
00510850  free  00000238  FF20E932  FEFE  prev:00510CC4  next:00410054
00510A88  used  0000023C  FF740EAE  0040  EIP: 004015B5
00510CC4  free  0000032C  FFD41CF2  FEFE  prev:00B5F014  next:00510850
```

图 5-9　WALKHEAP 程序运行输出

现在，我们已经看到了 Win32 堆首和块场的设计，下面该讨论一下伪码了。在这个问题上，我们将看到 KERNEL32 是如何生成、管理并破坏堆的。该伪码是为调试版的 Windows 95 而设计的。而对于零售版来讲，它并没有如此多的调试和检测代码，因而其效果更好。

GetProcessHeap（获得进程堆函数）

使用 Win32 堆函数时首先碰到的是堆柄。当应用程序生成时，绝大多数程序利用了由 KERNEL32 生成的缺省进程堆。你可以通过调用 GetProcessHeap 来恢

复某个堆柄。GetProcessHeap 函数很简单。该函数恢复了一个指向当前进程库的 KERNEL32 全局变量(见第六章的详细介绍)。某个进程库的内部是该进程的缺省 Win32 堆的句柄(即起始地址)。

GetProcessHeap 伪码

```
return  ppCurrentProcessId->lpProcessHeap;
```

HeapAlloc(堆配置函数)和 IHeapAlloc

HeapAlloc,正如其名称所内涵的那样,你可以利用它从某特定堆中分配一个内存块。HeapAlloc 代码是 KERNEL32.DLL 中有效层的一部分。配置该块的真正工作是由 IHeapAlloc 和 HPAlloc(后面要介绍)来完成的。HeapAlloc 所作的唯一有效性检验是检验 hHeap 句柄指向的内存区是否可以足够大地包容一个堆首。尽管此代码只能验证附加区段,包括符号区段和检验和区段,HeapAlloc 却忽略了这些区段。假定 hHeap 句柄通过了检验(不是非常精确),则函数将跳到 IHeapAlloc。

HeapAlloc 伪码

```
// Parameters:
//    HANDLE    hHeap
//    DWORD     dwFlags
//    DWORD     dwBytes

    Set up structured exception handler frame

    // Make sure that the hHeap is valid. A heap handle is just a
    // pointer to the beginning of the heap area.
    AL = *(PBYTE)hHeap;
    AL = *(PBYTE)(hHeap + 0xCF);

    Remove structured exception handler frame

    goto IHeapAlloc;
```

IHeapAlloc 只不过是一个 HPAlloc 函数的外壳(即,"真正的"HeapAlloc)。在调用 HPAlloc 之前,IHeapAlloc 对 dwFlags 参量进行乘法运算,而所剩下来的标志则只有 HEAP_ZERO_MEMORY 标志和 HEAP_GENERATE_EXCEPTIONS 标志了。HEAP_ZERO_MEMORY 标志(如果能保留下来的话)结束时要比开始高出 3 个位。

IHeapAlloc 伪码

```
// Parameters:
//    HANDLE    hHeap
//    DWORD     dwFlags
//    DWORD     dwBytes
// Locals:
//    DWORD     modifiedFlags;
    // Apparently some apps need a little extra room...
    if ( 0x00400000 bit set in TDB AppCompatibility flags )
        if ( hHeap == ppCurrentProcessId->DefaultHeap )
            dwBytes += 0x10;

    modifiedFlags = dwFlags;
    modifiedFlags &= HEAP_ZERO_MEMORY;
    dwFlags &= HEAP_GENERATE_EXCEPTIONS;
    modifiedFlags << 3;
    modifiedFlags |= dwFlags;

    return HPAlloc( hHeap, dwBytes, modifiedFlags );
```

HPAlloc

HPAlloc 是真正的 HeapAlloc 函数。其代码以检查所需的块尺寸是否过大而开始。这样，过大意味着是 0x0FFFFF98 个字节(大概是 256MB)。接下来，HPAlloc 调用 hpTakeSem，它使堆首的临界区处于查询状态。这样，直到最初的线程从 HPAlloc 返回，进程中的其它线程才能开始运行。对于调试版本，hpTakeSem 同样可以有选择地验证堆是否被破坏。另外，hpTakeSem 可以行走至堆并验证场的检验和以及堆符号表(0x4948)是否仍然存在。你应将这一特征与 HeapSetFlags 函数紧紧连在一起，该函数我在“Win32 堆函数”一节的开始已经提到，由于时间关系这本书中没有包括进来。

HPAlloc 接下来处理所需块的尺寸参量并将其与 4 相乘(同样是在考虑了所需场尺寸之后)。最小块尺寸是 0x18 字节，对于最后一个用户来讲，在减去场之后应转换成 8 字节。有了块尺寸，HPAlloc 就可以确定它应当寻找 4 个以尺寸为基础的自由列表中的哪一个了。在找到正确的列表之后，HPAlloc 就经过该列表(使用自由块中的前一个指针)来寻找具有足够尺寸的第一个块。

为此，假定 HPAlloc 找到了具有足够尺寸的某自由块。其后，HPAlloc 调用 hpCarve(在下面给出的伪码中可以看到)。hpCarve 函数检查一个块，看其是否刚好足够大或它是否需分成两块。如果该块需分成两块，hpCarve 则控制着所有诸如生成新场，建立前一个和下一个指针之类的工作。hpCarve 生成的块之一刚好足够大地满足了 HPAlloc 的需要，而其它块则是所剩下来的并进入了自由列表。

在 hpCarve 返回之后，HPAlloc 则开始在新的块中对场区段进行初始化。这是一系列简单的赋值语句，除了获得 HPAlloc 调用程序的 EIP 的调用和对该场的前

三个区段进行检验和运算的调用之外。最后，HPAlloc 释放堆的临界区并在该场之后返回一个指向首字节的指针。

现在让我们回来看看当 HPAlloc 在自由列中没有找到自由块时会发生什么。如果该堆是允许成长的(即，当堆生成时 0 被设定为 dwMaximumSize 参数)，则 HPAlloc 需要生成一个子堆。如前所述，一个子堆是从包括附加堆块的原始堆分离出来的一个内存区域。KERNEL32 通过将这些子堆保留在一个联合列表中而实现对它们的跟踪。如果某个子堆需要生成，KERNEL32 则确定其初始尺寸(典型的为 4MB)，并调用 VMM 来保留一定范围的页面。下一步，HPAlloc 调用 HPlnit 来初始化新子堆的堆首。有关 HPInit，在后面介绍 HeapCreate 函数时要详细谈到。最后，HPAlloc 跳回代码的起点来寻找自由列表。可以推测，这时就可以找到一个具有足够尺寸的块了。

HPAlloc 伪码

```
// Parameters:
//    HANDLE    hHeap       // ebp+0x08
//    DWORD     dwBytes     // ebp+0x0C
//    DWORD     dwFlags     // ebp+0x10
// Locals:
//      DWORD   newSubHeap
//      DWORD   temp;
//      HEAP_ARENA * pArena
//      DWORD   carvedSize;
//      DWORD   commitSizeBytes, commitSizePages;
//      FREE_LIST_HEADER *pFreeList;

    if ( dwBytes > 0x0FFFFF98 )
    {
        _DebugOut( "HPAlloc: request too big\n\r",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_NOT_ENOUGH_MEMORY );
        return 0;
    }
    // Grab the heap semaphore. This allows only one thread at a time to
    // be in the heap code so that the heap doesn't get corrupted.
    // In the debug version, with paranoid heap checking enabled, this is
    // where the heap would be walked and checked for corruption.
    if ( !hpTakeSem(hHeap, 0, dwFlags) )
        return 0;

    temp = dwBytes + 13;    // Round up to the next multiple of 4.
    temp &= 0xFFFFFFFC
    if ( temp <= 0x18 )     // Minimum allocation size is 0x18 bytes
        dwBytes = 0x18      // (or 8 bytes after subtracting the arena).

HPAlloc_find_free_block:

    // Figure out which of the four free lists should be searched
    // (based on the size of the requested block).
    pFreeList = hHeap->freeListArray;          // Point at first free list.
    while ( dwBytes > pFreeList->dwMaxBlockSize )
```

```
    pFreeList++;        // Advance to next free list.

// Walk the free list looking for a block that's big enough. If one
// is found, jump to HPAlloc_split_block. Otherwise, fall through.

// Are there entries in the free list?
if ( pFreeList->arena.prev != &hHeap.freeListArray[0].freeArena )
{
    pArena = pFreeList->arena.prev; // Start at head of free list

    // Scan through the list, looking for a block that's big enough.
    // When we find one, go split it.
    while ( pArena != &hHeap.freeListArray[0].freeArena )
    {
        if ( (pArena->size & 0x0FFFFFFC) < dwBytes )
            goto HPAlloc_split_block;

        // Not big enough. Go on to next (previous) block in free list.
        pArena = pArena->prev;
    }
}

// If we get here, there's not enough room to satisfy the request.
// If the HEAP_FREE_CHECKING_ENABLED flag wasn't specified when the
// heap was created, display a message and then bail out. The
// HEAP_FREE_CHECKING_ENABLED flag is set by specifying 0 as the
// dwMaximumSize param to HeapCreate.
if ( ! (hHeap->flags & HEAP_FREE_CHECKING_ENABLED) )
{
    _DebugOut( "HPAlloc: not enough room on heap\n",
                SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_NOT_ENOUGH_MEMORY );
    goto HPAlloc_error;
}

// If we get here, there wasn't enough room to satisfy the heap, but
// HEAP_FREE_CHECKING_ENABLED was specified (the dwMaximumSize param
// was 0). Therefore, KERNEL32 can try to extend the heap by
// allocating more virtual memory. The normal size of these new
// subheaps is 4MB.
if ( dwBytes <= 0x400000 )
    commitSizeBytes = 0x400000;

commitSizePages = commitSizeBytes >> 12;    // Convert bytes to pages.

// Reserve the memory for the new subheap. Check the hHeap value
// to see if it should be in app private memory or in shared memory.
newSubHeap = VxDCall( _PageReserve,
                    hHeap > 0x80000000 ? PR_SHARED : PR_PRIVATE,
                    commitSizePages, PR_STATIC );

if ( newSubHeap == -1 )             // Oops! The reserve failed.
{
    _DebugOut( "HPAlloc: reserve failed\n",
                SLE_WARNING + FStopOnRing3MemoryError );
```

```
        InternalSetLastError( ERROR_NOT_ENOUGH_MEMORY );
        goto HeapAlloc_error
    }

    // Go initialize the new subheap. If the init fails, free the memory.
    if ( !HPInit(hHeap, newSubHeap, commitSizeBytes, hHeap->flags & 0x0100) )
    {
        VxDCall( _PageFree, newSubHeap, 0x10 );
        goto HeapAlloc_error;
    }

    // Insert the newly allocated subHeap in the linked list of subHeaps.
    newSubHeap->next = hHeap->next;
    hHeap->next = newSubHeap;

    // Go back and start the search again.
    goto HPAlloc_find_free_block;

HPAlloc_split_block:
    // If we get here, we've found a free block that's either just big
    // enough or too big. If necessary, hpCarve splits the block into
    // two blocks, one of which is just big enough for the allocation.
    dwBytes = hpCarve( hHeap, pArena, dwBytes, dwFlags );
    if ( dwBytes == 0 )
        goto HPAlloc_error;

    // Start filling in the fields of the new block's arena.
    pArena->size = carvedSize | 0xA0000000;
    pArena->signature = "BH";                    // "BH" = 0x4842
    pArena->calling_EIP = x_GetCallingEIP();

    if ( ppCurrentThreadId )
        pArena->threadID = ppCurrentThreadId->processID->CurrentThreadOrdinal;
    else
        pArena->threadID = 0;

    pArena->checksum = ChecksumHeapBlock(pArena, 3);    // Checksum the block.

    x_hpReleaseSem( hHeap, dwFlags );   // Release the heap semaphore.

    return pArena+0x10; // Return first address following the arena struct.


HPAlloc_error:
    // If we get here, something went wrong.

    // Release the heap semaphore.
    x_hpReleaseSem( hHeap, dwFlags );

    // If the HEAP_GENERATE_EXCEPTIONS flag is set in the heap header
    // or dwFlags param, make a STATUS_NO_MEMORY exception.
    if ( (hHeap->flags | dwFlags) & HEAP_GENERATE_EXCEPTIONS )
        x_RaiseException( STATUS_NO_MEMORY, 0, 1, &dwBytes );

    return 0;
```

hpCarve(堆分隔函数)

hpCarve 从 HPAlloc 函数中取走了一个自由块并将其分成两个部分。两个合成块的第一个块必需具有调用程序(HPAlloc)所需的尺寸。hpCarve 以某种有效性检验代码开始,这种代码确保了该块不会小于所需的尺寸。第二个测试是确保所要分隔的块未被使用。

hpCarve 代码的主体是单向向前并很容易跟随的,其主要目的是建立一个新的自由块场并确信所有的前面指针和下面指针已经建立,详见其伪码。

比一般的 hpCarve 代码更有趣的是内存提交代码(memory committing code)。正如我们在前面 HPAlloc 代码所看到的那样(而且你还会在 HeapCreate 函数中看到),Win32 堆是稀疏的。即,处于堆界限中的所有内存是被保留了的但未被提交。如果我们拥有 1MB 的堆并对其提交了 1MB 的物理内存,但这 1MB 的内存并不需要,这种作法无疑是很浪费的。然而,当一个进程触及到了某个保留了的但并未提交的页面时,将会导致一个缺页中断。因此,堆函数需要明确要提交的所有的供在线块使用的页面。

当一个块分成两个部分时,hpCarve 需要提交所有横跨第一个块的页面。另外,hpCarve 为第二个块("剩余项")生成(并写入)了一个新的场,因而页面也同样要提交。提交工作是由 hpCommit 函数来完成的。hpCommit 确定了内存页面的状态(status),而且必要时,可以调用 VMM_PageCommit Win32 服务。如果你认为 Windows 95 是利用结构化的异常处理在堆中提交必要的页面的话,其实是错误的!(至少,在调试版的 Windows 95 中是这样。)

在剩余项的块场建立之后,hpCarve 利用一个常量初始化了分隔块的第一部分。(这就是为什么 hpCarve 没有设置其场区段的原因。)如果 HEAP_ZERO_MEMORY 通过了 HeapAlloc,hpCarve 则将该块置于 0 场。否则,hpCarve 将该块置于 0xCC's,即断点选择代码(breakpoint opcode)。hpCarve 的调用程序负责在被分隔块的起点生成场结构。

hpCarve 伪码

```
// Parameters:
//      HANDLE hHeap        // ebp+08
//      HEAP_ARENA * pArena // ebp+0c
//      DWORD   dwBytes     // ebp+10
//      DWORD   dwFlags     // ebp+14
// Locals:
//      DWORD   myLocal
//      DWORD   currBlockSize
//      DWORD   startCommitPage, endCommitPage, pagesToCommit
//      HEAP_ARENA *pNextArena

    // Get the size of the block that's about to be split. Mask
    // the 0xA0000003 bits to get the actual size.
    currBlockSize = pArena->size & 0x0FFFFFFC;
```

```
if ( dwBytes > currBlockSize )
{
    _DebugOut( "hpCarve: carving out too big a block\n", SLE_ERROR );
}

if ( 0 == (pArena->size & 1) )      // Check the "block in use" flag.
{
    _DebugOut( "hpCarve: target not free\n", SLE_ERROR );
}

endCommitPage1 = ((DWORD)pArena + currBlockSize - 4) >> 12;

startCommitPage = (pArena + 0x1013) >> 12;

// At this point, the code checks to see if the block being carved
// is the same size (or only slightly bigger) than the requested block
// size. If so, it doesn't make sense to make two separate blocks.
// The "if" portion of the following code handles the case where the
// block being split is large enough to warrant making two blocks.
// The first of the resulting two blocks will be the block of size
// dwBytes. The remaining memory will go into a new free block.

if ( (dwBytes + 0x18) <= currBlockSize )
{
    endCommitPage2 = (pArena + dwBytes + 0x13) >> 12;

    if ( endCommitPage2 == endCommitPage1 )
        endCommitPage2--;

    pagesToCommit = endCommitPage2 - startCommitPage + 1

    // hpCommit ultimately calls VMM's _PageCommit Win32 service
    // to commit the page.
    if ( !hpCommit(startCommitPage, pagesToCommit, hHeap->flags) )
        return 0;

    // Set up the new arenas.
    pArena->prev->next = pArena->next;
    pArena->next->prev = pArena->prev;
    pArena->prev->freeBlockChecksum = ChecksumHeapBlock( pArena->prev, 4 );
    pArena->next->freeBlockChecksum = ChecksumHeapBlock( pArena->next, 4 );

    // Make a new free block starting "dwBytes" into the block we're
    // splitting. hpFreeSub is the same routine used by HeapFree.
    hpFreeSub( hHeap, pArena + dwBytes, currBlockSize - dwBytes, 0 );
}
else    // The block isn't large enough to warrant making two blocks.
{
    // hpCommit ultimately calls VMM's _PageCommit Win32 service
    // to commit the page.
    if ( !hpCommit(startCommitPage, endCommitPage1-startCommitPage,
                   hHeap->flags))
        return 0;
```

```
    pArena->prev->next = pArena->next;
    pArena->next->prev = pArena->prev;
    pArena->prev->freeBlockChecksum = ChecksumHeapBlock( pArena->prev, 4 );
    pArena->next->freeBlockChecksum = ChecksumHeapBlock( pArena->next, 4 );

    // The next arena is for an in-use block. (If it weren't in use,
    // it would have been coalesced with this block.)
    pNextArena = pArena + currBlockSize;
    HIBYTE(pNextArena->size) &= 0xFD;
    pNextArena->checksum = x_ChecsumBlock( pNextArena, 3 );
}

if ( dwFlags & 0x40  )                  // 0x40 == HEAP_ZERO_MEMORY << 3
    memset( pArena, 0, dwBytes );       // Zero fill the block.
else
    memset( pArena, 0xCC, dwBytes );    // Fill the blocks with 0xCC's.

return dwBytes;
```

ChecksumHeapBlock(堆块检验和函数)

ChecksumHeapBlock 是我们要考察的最后一个有关 HeapAlloc-related(堆配置相关函数)的例程。ChecksumHeapBlock 只用于调试版的 Windows 95。它采用了一个指向场的起点并指向所使用的 DWORD 数目的指针。ChecksumHeapBlock 被通知处理某在线块的三个 DWORD 以及某自由块的 4 个 DWORD。以某个为 0 的初始值开始,ChecksumHeapBlock 将每个成功的 DWORD 与某个检验和 DWORD 进行 XOR 运算。最后,ChecksumHeapBlock 利用 0x17751965 对检验和 DWORD 进行 XOR 运算并返回结果。

ChecksumHeapBlock 伪码

```
// Parameters:
//      DWORD   count   // Number of contiguous DWORDs to checksum.
//      PDWORD  pBlock  // Starting address to checksum.
// Locals:
//      DWORD   accumulator, i;

    accumulator = 0;

    for ( i=0; i < count; i++ )
        accumulator ^= pBlock[i];    // XOR the accumulator with the next
                                     // DWORD in the block;

    accumulator ^= 0x17761965;       // 1776 == U.S. Independence?
                                     // 1965 == year of birth of an MS programmer?
    return accumulator;
```

HeapSize 和 IHeapSize 函数(堆尺寸函数)

HeapSize 采用了一个指向前一配置块的指针并返回了该块的尺寸(不计入场)。

HeapSize 代码就是一个参量有效层(parameter-validation layer),它在 JMPing 到达 IHeapSize 之前使通过的参量有效化。IHeapSize 代码以从 lpMem 指针减去 0x10 开始,进而得到指向块场的指针——这或许正是我们所期待的! 接下来,IHeapSize 抓住堆的临界区域以防将无效的结果赋予某个不切时的线程开关。IHeapSize 的核心只不过是获取场的尺寸区段,AND 掉 0xA0000003 位,然后减去 0x10。减去 0x10 考虑了场的尺寸,因此返回的数值是调用程序所用的内存和。最后,IHeapSize 放弃了堆的临界区域并返回了块尺寸(减去了场)。

HeapSize 伪码

```
// Parameters:
//    HANDLE    hHeap
//    DWORD     dwFlags
//    DWORD     lpMem

   Set up structured exception handler frame
   // Make sure that the hHeap is valid. A heap handle is just a
   // pointer to the beginning of the heap area.
   AL = *(PBYTE)hHeap;
   AL = *(PBYTE)(hHeap + 0xCF);

   // Verify that the lpMem parameter points to valid memory.
   AL = *(LPBYTE)(lpMem+0x7)
   AL = *(LPBYTE)(lpMem-0x10);

   Remove structured exception handler frame

   goto IHeapSize;
```

IHeapSize 伪码

```
// Parameters:
//    HANDLE    hHeap
//    DWORD     dwFlags
//    LPCVOID   lpMem
// Locals:
//      HEAP_ARENA * pArena
//      DWORD size;

   pArena = lpMem - 0x10;
```

```
// Grab the heap semaphore. This allows only one thread at a time to
// be in the heap code so that the heap doesn't get corrupted.
if ( hpTakeSem(hHeap, lpMem, dwFlags) )
    return 0;

// Get the size field from the arena, get rid of the 0xA0000000 flags,
// and subtract 0x10 (to subtract out the size of the arena).
size = (pArena->size & 0x0FFFFFFC) - 0x10;

x_hpReleaseSem( hHeap, dwFlags );

return size;
```

HeapFree 和 IHeapFree(堆释放函数)

HeapFree 是另一个实际上只是一个参量有效性存根的函数。HeapFree 检查 hHeap 句柄指向的有效内存是否足够大地容纳一个堆首。代码同时测试 lpMem 指针是否指向了一个有效的堆块指针。应当有 0x10 字节的场先于此 lpMem 参量，而且此 lpMem 块应至少是 8 字节长(不计入场)。HeapFree 因而验证了内存是否可在 lpMem 之前被 0x10 字节访问并在 lpMem 之后被 0x7 字节所访问。这些测试之后，HeapFree 跳到某个称为 x_HeapFree 的特殊例程(下面要谈到)。

HeapFree 伪码

```
// Parameters:
//    HANDLE    hHeap
//    DWORD     dwFlags
//    LPVOID    lpMem

    Set up structured exception handler frame.

    // Make sure that the hHeap is valid. A heap handle is just a
    // pointer to the beginning of the heap area.
    AL = *(PBYTE)hHeap;
    AL = *(PBYTE)(hHeap + 0xCF);

    // Verify that the lpMem parameter points to valid memory.
    AL = *(LPBYTE)(lpMem+0x7)
    AL = *(LPBYTE)(lpMem-0x10);

    Remove structured exception handler frame.

    goto x_HeapFree;
```

x_HeapFree 例程位于 HeapFree 有效代码与 IHeapFree 函数之间，用于释放块。似乎出于某种原因，不是每个块都可以直接通向 IHeapFree 的。由某一例程释放的块似需特殊的处理。x_HeapFree 的任务就是确定谁调用了它。如果存不在来自特殊地址的调用，x_HeapFree 将跳到 IHeapFree 代码。(几乎全部如此。)如果

x_HeapFree 被某个特殊的例程所调用，它将调用一个好像要弄乱块场的函数。在该函数返回之后，x_HeapFree 跳到 IHeapFree。

x_HeapFree 伪码

```
// Locals:
//      DWORD   returnAddress;

    returnAddress = *(LPWORD)ESP;

    if ( (returnAddress & 0x00000FFF) != some number )
        goto IHeapFree;

    if ( !someFunction( ) )
        goto IHeapFree;

    Munge the return address on the stack so that control returns to
    to x_HeapFreeRet when IHeapFree returns

    goto IHeapFree

x_HeapFree_ret:
```

IHeapFree 具有两个功能。首先，如果先于正被立即释放块的块本身是自由的，IHeapFree 则会将这类块合并。IHeapFree 是如何知道此先前块是否自由呢？如果此先前块是自由的话，块场尺寸段中的位 1(值为 2)是处于开状态的。那么，IHeapFree 是如何知道怎样寻找此先前块的？结果表明，先前块内存区的最后一个 DWORD 是一个指向此先前块场指针的。因此，IHeapFree 刚好从正被释放的场中减去了 4。这样得到的地址是指向堆中的前一块的指针的。IHeapFree 通过为前一块调用 hpFreeSub 函数将此加以合并，并通知 hpFreeSub 该块的长度是两个联合块的尺寸。

IHeapFree 的第二个任务是激活 hpFreeSub。hpFreeSub 是 HPAlloc 函数的合作者。hpFreeSub 实施着将某个块释放回堆的实际工作，下面要介绍。在这项工作进行之际，IHeapFree 控制着堆的信号量，它抢占了入口并在调用 hpFreeSub 之后解脱。

IHeapFree 伪码

```
// Parameters:
//    HANDLE    hHeap
//    DWORD     dwFlags
//    LPVOID    lpMem
// Locals:
//      HEAP_ARENA * pArena
//      HEAP_ARENA * pPrevArena
//      DWORD   blockSize;
```

```
pArena = lpMem - 0x10;

// Grab the heap semaphore. This allows only one thread at a time to
// be in the heap code so that the heap doesn't get corrupted.
if ( !hpTakeSem(hHeap, pArena, dwFlags) )
    return 0;

blockSize = pArena->size & 0x0FFFFFFC;

// Is previous arena free? If so, we'll be coalescing this block
// with the previous block. This is going to affect the arenas
// of the previous block's previous/next blocks, so recalculate
// the checksums.
if ( pArena->size & 2 )
{
    pPrevArena = *(PDWORD)(pArena-4);

    blockSize += (pPrevArena->size & 0x0FFFFFFC);

    pPrevArena->prev->next = pPrevArena->next
    pPrevArena->next->prev = pPrevArena->prev

    pPrevArena->prev.freeBlockChecksum
            = ChecksumHeapBlock(pPrevArena->prev, 4);
    pPrevArena->next.freeBlockChecksum
            = ChecksumHeapBlock(pPrevArena->next, 4);

    pPrevArena = pArena;
}

// Call hpFreeSub to do the real work.
hpFreeSub( hHeap, pArena, blockSize, 0x200 );

// Give up the heap critical section.
x_hpReleaseSem( hHeap, dwFlags );

return 1;
```

hpFreeSub

hpFreeSub 函数将一个块释放回堆。所要释放场的地址和一个长度通过了该函数。该函数可以从几个地方被调用，包括 IHeapFree 和 hpCarve。hpFreeSub 的近期用法是有趣的，因为正被释放的块是某个已经自由了的块的一部分。

hpFreeSub 的伪码相当长，因此我不准备详细介绍。从高一点的角度来讲，hpFreeSub 包括两个明显的部分。hpFreeSub 的第一个部分考虑了下放(decommitting)内存的问题，如果必要的话。当一个程序将一大块内存释放回操作系统时，它并不想保留所有交付了的内存。因此，hpFreeSub 来决定所提交的页面并不会弄乱堆中的其它块。如果存在着符合标准的块，hpFreeSub 则调用 VMM 的_PageDecommit(页面下放)Win32 服务来释放该块。当 hpFreeSub 由 hpCarve 调用时会出

现例外。这时，hpFreeSub 并不下放任何页面，而是查看受影响的页面是否仍处于保留状态。

hpFreeSub 的第二个部分考虑了对场更新的问题。首先，在此自由块之后的块必需是一个正在使用的块；否则，它应该是正被释放块的一部分。hpFreeSub 因而在下一个场的尺寸段内将位 1(值为 2)置于开的状态，从而通知下一个场前一个场是一个自由块。接下来，hpFreeSub 决定正被释放的块应进入哪一个基于尺寸的自由列表。在找到合适的自由列之后，hpFreeSub 流览该列表，寻找恰当的点来插入此新释放块。(自由列表根据尺寸分类保留。)最后，hpFreeSub 为此新释放块写入一个新的场，其中包括填写前一个和下一个区段以及计算检验和。

hpFreeSub 伪码

```
// Parameters:
//      HANDLE  hHeap
//      HEAP_ARENA * pArena
//      DWORD   size            // Size to make the free block.
//      DWORD   flags
// Locals:
//      HEAP_ARENA * pNextArena // Arena that immediately follows pArena.
//      HEAP_ARENA * pFreeListArena // Pointer to first arena in the free list.
//      DWORD nextArenaSize;
//      DWORD *myLocal
//      DWORD bytesToblast;
//      PSTR pszError
//      DWORD startDecommitPage, endDecommitPage1, endDecommitPage2;
//      FREE_LIST_HEADER *pFreeList;

    if ( size < 0x18 )
        _DebugOut( "hpFreeSub: bad param\n", SLE_ERROR );

    endDecommitPage1 = 0x00100000;

    pNextArena = &pArena + size;    // Get a pointer to the next arena.

    if ( pNextArena->size & 1 )
    {
        nextArenaSize = pNextArena->size & 0x0FFFFFFC;

        pNextArena->prev->next = pNextArena->next

        pNextArena->next->prev = pNextArena->prev

        pNextArena->prev->freeChecksumBlock
                = ChecksumHeapBlock( pNextArena->prev, 4 );

        pNextArena->next->freeChecksumBlock
                = ChecksumHeapBlock( pNextArena->next, 4 );
```

```
        endDecommitpage1 = (pNextArena + 0x1013) >> 12;

        pNextArena = pArena + size + nextArenaSize;
    }

    // Figure out how many bytes there are from the start of the arena
    // to the end of the containing page. Then round down to the size
    // of the block to free (if less).
    bytesToBlast = 0x1000 - (&pArena & 0x00000FFF);
    if ( bytesToBlast >= size )
        bytesToBlast = size;

    // Fill in the block to be freed with 0xFE's.
    memset( &pArena, 0xFE, bytesToBlast );

    if ( flags & 0x200 )    // True if called from IHeapFree; not true if
    {                       // called from hpCarve.

        startDecommitPage = (&pArena + 0x1013) >> 12;
        endDecommitPage2 = (&pNextArena - 4) >> 12;

        if ( endDecommitPage2 < endDecommitPage1 )
            goto hpFreeSub_modify_arenas;
        if ( VxDCall( _PageDecommit, startDecommitPage,
                    endDecommitPage2 - startDecommitPage, 0x20000000) )
            goto hpFreeSub_modify_arenas;

        pszError = "hpFreeSub: PageDecommit failed\n"
        goto hpFreeSub_error
    }
    else    // This code is hit when hpCarve is the caller.
    {
        MEMORY_BASIC_INFORMATION mbi;

        startDecommitPage = (&pArena + 0x1013) >> 12;
        endDecommitPage2 = &pNextArena >> 12;

        if ( endDecommitPage2 < startDecommitPage )
            goto hpFreeSub_modify_arenas;

        VxDCall( _PageQuery, startDecommitPage << 12,
                &mbi, sizeof(mbi) )

        // Check that the structure was filled in with values indicating
        // that the range of pages is all reserved.
        if ( (mbi.state == MEM_RESERVE) &&
             ((endDecommitPage2>> 12) <= someStruct[3]) )
               goto hpFreeSub_modify_arenas

        pszError = "hpFreeSub: range not all reserved\n"
    }

hpFreeSub_error:
    _DebugOutput( pszError, SLE_ERROR );
```

```
hpFreeSub_modify_arenas:
    *myLocal = pArena;

    // The next block must be an in-use block; otherwise, it would
    // been coalesced already. Turn on the "Previous block is free"
    // bit and redo its checksum.
    pNextArena->size |= 2;
    pNextArena->checksum = ChecksumHeapBlock( pNextArena, 3 );

    // Find the appropriate free list to insert this block into. The free
    // lists are kept as an array of FREE_LIST_HEADER structures starting
    // at offset 8 in the heap structure.
    pFreeList = hHeap->freeListArray;
    while ( size > pFreeList->dwMaxBlockSize )
        pFreeList++;        // Advance to next free list
    // We found the right free list. Now go insert it into the list in
    // size-sorted order.
    pFreeListArena = &pFreeList->arena;

    // Figure out where in the free list this block should go. The blocks
    // are kept in size-sorted order.
    while ( size > (pFreeListArena->prev.size & 0x0FFFFFFC) )
        pFreeListArena = pFreeListArena->prev;

    pArena->prev = pFreeListArena->prev;
    pFreeListArena->prev->next = pArena;
    pArena->next = pFreeListArena;
    pFreeListArena->prev = pArena;

    pArena->prev->freeBlockChecksum = ChecksumHeapBlock( pArena->prev, 4 );

    pFreeListArena->freeBlockChecksum = ChecksumHeapBlock( pFreeListArena, 4 );

    pArena->signature = "FH";   // FH = 0x4846
    pArena->freeBlockChecksum = ChecksumHeapBlock( pArena, 4 );
    pArena->size = size | 0xA0000001;
```

HeapReAlloc 和 IHeapReAlloc(堆再配置函数)

HeapReAlloc 函数用来在某个 Win32 堆中再配置一个存在的块。HeapReAlloc 代码也同样是一个参量有效层存根。HeapReAlloc 进行的测试与 HeapFree 实施的验证是一致的。hHeap 参量必需指向一个长度为 0xD0 字节的内存块。lpMem 参数则必需在该指针之前为 0x10 字节,而在此指针之后为 0x7 字节。如果测试通过,HeapReAlloc 则跳到 IHeapReAlloc。

IHeapReAlloc 是一个位代码。在它调用 HPReAlloc 函数之前,此代码重新将 dwFlags 参量安排至 HPReAlloc 的优先级。(为什么原来的通过 HeapReAlloc 的 HEAP_xxx 标志并不足够好这一问题对我来说仍是一个迷……)使其通过 HeapReAlloc 标志区的仅有标志为:

HEAP_GENERATE_EXCEPTIONS
HEAP_NO_SERIALIZE
HEAP_ZERO_MEMORY
HEAP_REALLOC_IN_PLACE_ONLY

HeapReAlloc 伪码

```
// Parameters:
//      HANDLE  hHeap
//      DWORD   dwFlags
//      LPVOID  lpMem
//      DWORD   dwBytes

    Set up structured exception handler frame

    // Make sure that the hHeap is valid. A heap handle is just a
    // pointer to the beginning of the heap area.
    AL = *(PBYTE)hHeap;
    AL = *(PBYTE)(hHeap + 0xCF);

    // Verify that the lpMem parameter points to valid memory.
    AL = *(LPBYTE)(lpMem+0x7)
    AL = *(LPBYTE)(lpMem-0x10);

    Remove structured exception handler frame

    goto IHeapReAlloc;
```

IHeapReAlloc 伪码

```
// Parameters:
//      HANDLE  hHeap
//      DWORD   dwFlags
//      LPVOID  lpMem
//      DWORD   dwBytes
// Locals:
//      DWORD   modifiedFlags

    modifiedFlags = some contorted mess of calculations with dwFlags.

    HEAP_GENERATE_EXCEPTIONS and HEAP_NO_SERIALIZE are passed through
    unscathed.

    The HEAP_ZERO_MEMORY bit is shifted left by 3.

    If the HEAP_REALLOC_IN_PLACE_ONLY bit is off, bit 1 (value 2) is turned
    on.

    return HPRealloc( hHeap, lpMem, dwBytes, modifiedFlags );
```

HPReAlloc 函数

HPReAlloc 包括了 HeapReAlloc 函数的核心。HPReAlloc 中的代码也相当长，但并不难算出。其伪码包括所有的真实细节。进一步讲，HeapReAlloc 有 4 个方面的情况要面对：

新块尺寸小于原块尺寸。

新块尺寸基本上等于原块尺寸。

新块尺寸大于原块尺寸。堆中的下一个块是自由的并可与原块合并来生成一个尺寸足够大的块。

新块尺寸大于原块尺寸，而且下一块并不自由。或者，下一块虽为自由块，但并不足够大地可以与原块合并来满足配置的需要。

对于第一种情况(新块尺寸小于原块尺寸)，HPReAlloc 利用 hpFreeSub 函数将原块分成两个部分。第一部分为新块，而第二部分为自由块。

对于第二种情况(即新块尺寸基本上等于原块尺寸)，HPReAlloc 则直接单独地离开了存在块。其分界点(threshold)大约为 8 字节(如果你计入了场则为 0x18 字节)。

对于第三种情况(即下一个块自由并足够大地满足合并需要)，HPReAlloc 计算出了它所需的下一个块的多少。其后，代码利用 hpCarve 将下一个自由块分成两块，其中的第一块可以足够大地与原块合并来满足新的所需尺寸，自由块的剩余部分则变成了一个更小的自由块。

最后一种情况是当其它情况失败时，HPReAlloc 试图利用 HPAlloc 分配一个满足所需尺寸的新块。如果配置成功，HPReAlloc 则将原块的内容拷贝到新配置的块中。之后，HPReAlloc 调用 HeapFree 的内部版本来释放原块内存。

HPReAlloc 伪码

```
// Parameters:
//      HANDLE  hHeap
//      LPVOID  lpMem
//      DWORD   dwBytes
//      DWORD   dwFlags
// Locals:
//      DWORD   newSize;
//      HEAP_ARENA *pArena, *pNextArena
//      DWORD   nextBlockSize;
//      LPVOID  lpMem2;
//      DWORD   originalBlockSize;
//      PVOID   prevFreeArena
```

```
newSize = dwBytes;

// Make sure the new size isn't too big.
if ( newSize > 0x0FFFFF98 )
{
    _DebugOut( "HPReAlloc: request too big\n\r",
        SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_NOT_ENOUGH_MEMORY );
    goto HPReAlloc_failure;
}

// Point at the arena of the block to be reallocated.
pArena = lpMem - 0x10;

// Prevent any other threads from coming through here and
// screwing up the heap.
if ( !hpTakeSem( hHeap, pArena, dwFlags ) )
    goto HPReAlloc_failure;

// Round up the requested size by 0x10 bytes (for the arena), and then
// make sure it's a multiple of 4.
if ( (newSize + 0x13) & 0xFFFFFFFC < 0x18 )
    newSize = 0x18;

originalBlockSize = pArena->size & 0x0FFFFFFC;

// Is the new block size + 0x18 less than the original size? If so,
// we can simply shorten the existing block.

if ( (newSize + 0x18) <= originalBlockSize )
{
    // Shorten the existing block by having hpFreeSub make a new arena
    // right past where the realloc'ed block will end.
    hpFreeSub(hHeap, pArena+newSize, originalBlockSize - newSize, 0x200);

    // Update the arena's size field to contain the new size. Leave
    // the high BYTE and bottom 2 bits of the size the way they were.
    // Yes, this is somewhat of a brain twister at first.
    pArena->size = (pArena->size & 0xF0000003) | newSize;

    pArena->checksum = ChecksumHeapBlock( pArena, 3 );
    goto HPReAlloc_success;
}

// If the new block size is only marginally smaller than the original
// size, just leave the block alone.
if ( originalBlockSize >= newSize )
    goto HPReAlloc_success;

// If we get here, the block is being reallocated to a size bigger than
// it was originally.
```

```
pNextArena = pArena + originalBlockSize;    // Get pointer to next arena.
nextBlockSize = pNextArena->size;           // Get size of next block.

// If the next arena is free, we can combine part of it with the
// existing block. Whatever's left over will remain a free block.
if ( nextBlockSize & 1 )        // Is next arena free?
{
    if ( (nextBlockSize & 0x0FFFFFFC) >= (newSize - originalBlockSize) )
    {
        DWORD extraNeeded = newSize-originalBlockSize;

        // Carve out a block big enough to tack onto the existing
        // block. The remainder becomes a new free block.
        if ( !hpCarve(hHeap, extraNeeded, pNextArena, , dwFlags))
            goto HPReAlloc_failure;

        pArena->size = (pArena->size & 0xF0000003) | extraNeeded;
        pArena->checksum = ChecksumHeapBlock( pArena, 3 );
        goto HPReAlloc_success;
    }
}

// If HEAP_REALLOC_IN_PLACE_ONLY wasn't specified, we can alloc a
// new block somewhere else, then copy the original block's contents
// over. Normally, HEAP_REALLOC_IN_PLACE_ONLY isn't specified.
if ( dwFlags & 2 )
{
    // Save some fields of the original arena, because we'll need to
    // copy them into the new block's arena.

    WORD threadID = pArena->threadID;
    prevFreeArena = pArena->prev

    if ( dwFlags & 0x20 )   // This doesn't seem to happen normally.
    {
        HeapFree_special( hHeap, HEAP_NO_SERIALIZE, lpMem );
        lpMem = HPAlloc( hHeap, newSize, dwFlags | HEAP_NO_SERIALIZE );
        if ( lpMem )
            goto HaveNewBlock;

        _DebugPrintf( "HPReAlloc: HPAlloc failed 1\n" );
        goto HPReAlloc_failure;
    }

    // Allocate a new block of the desired size from the heap.
    lpMem2 = HPAlloc( hHeap, newSize, dwFlags | HEAP_NO_SERIALIZE );
    if ( !lpMem2 )
    {
        _DebugPrintf( "HPReAlloc: HPAlloc failed 2\n" );
        goto HPReAlloc_failure;
    }

    // Copy the contents of the original block to the new block.
    // We subtract 0x10 because we don't need to copy the arena.
```

```
        memcpy( lpMem2, lpMem, originalBlockSize - 0x10 );

        // Free the original block.
        lpMem = HeapFree_special( hHeap, HEAP_NO_SERIALIZE, lpMem );

HaveNewBlock:
        // Fill in the arena header of the new block.
        pArena = lpMem - 0x10;

        lpMem->prev = prevFreeArena;
        lpMem->threadID = threadID;
        pArena->checksum = ChecksumHeapBlock( pArena, 3 );
        goto HPReAlloc_success;
    }

    // If we get here, HEAP_REALLOC_IN_PLACE_ONLY was specified, and there
    // wasn't enough memory. Display a warning to the debug terminal.
    _DebugOut( "HPReAlloc: fixed block\n",
               SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_LOCKED );

    // Fall through to failure.

HPReAlloc_failure:
    x_hpReleaseSem( hHeap, dwFlags );   // OK, safe for other threads now.
    return 0;

HPReAlloc_success:
    x_hpReleaseSem( hHeap, dwFlags );   // OK, safe for other threads now.
    return lpMem;
```

HeapCreate 函数(堆生成函数)

HeapCreate 函数是所有 Win32 堆的起点。每个 Win32 程序在应用开始之前都拥有一个生成的堆。另外,程序调用 HeapCreate 来生成堆,与缺省的程序堆分开。除了由程序使用之外,KERNEL32 本身也调用了 HeapCreate 以在共享内存中生成堆。它利用这些堆来存放系统的数据结构,如线程和进程例程结构。尽管未被资源化,应用程序仍能利用此功能,在调用 HeapCreate 时,通过在 fOptions 标志中设定 0x4000000 位值,来生成一个共享堆。

生成一个 Win32 的进程包括两部分。HeapCreate 掌握着此堆的保留内存的高级细节并将此堆连结到了进程堆的列表。堆生成的其它部分是初始化堆首的所有区段。为此,HeapCreate 调用了 HPInit 函数,前面已经介绍过这一函数。

HeapCreate 以检查并修改输入尺寸参量(如果必要)开始。首先,它查看 dwInitialSize 参量是否小于最大的尺寸参量。然后,HeapCreate 将此 dwMaximumSize 参数舍到最近的 4KB 页面边界中。dwInitialSize 等于 0 的情况需要特殊处理。此时,堆可根据需要成长。如果 HeapAlloc 函数无法在当前堆中找到足够自由的内存,它可以保留另外一大块内存并在该块内建立一个子堆。HeapCreate 代码检查

dwMaximumSize 是否被置为 0，如果是，在 fOptions 参数内设置 0x40 位(或许是 HEAP_FREE_CHECKING_ENABLED)。初始参量测试的最后一个位是观察 0x04000000 位是否已设置，从而表明此堆应位于 2GB 之上的共享内存中。

在 HeapCreate 已经决定其应当生成之后，它要调用 VMM 的_PageReverse (页面保留函数)Win32 服务来保留足够的线性地址空间以容纳此堆。假定页面保留已经完成，HeapCreate 则会调用 HPInit 来初始化堆首区段。在 HPInit 返回之后，HeapCreate 则查看其是否正在生成 KERNEL32 共享堆并采取必要的动作。HeapCreate 代码的最后一部分是为该进程将新生成的堆加到堆的列表上去。对于为某个进程而生成的第一个堆的情况，新堆放在了被保留于进程库的堆列表的首部(见第六章)。如果新堆并非首次生成的堆，HeapCreate 则将此新堆置于列表开始部分。

HeapCreate 伪码

```
// Parameters:
//      DWORD   fOptions
//      DWORD   dwInitialSize
//      DWORD   dwMaximumSize;
// Locals:
//      HANDLE  hHeap, hHeap2;
//      DWORD   retValue
//      DWORD   fShared

    retValue = 0;

    // If a nonzero maximum size was specified, make sure it's bigger
    // than the initial size.
    if ( dwMaximumSize && (dwMaximumSize < dwInitialSize) )
    {
        _DebugOut( "HeapCreate: dwInitialSize > dwMaximumSize\n",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_INVALID_PARAMETER );
        return 0;
    }

    // Round dwMaximumSize up to the nearest page boundary.
    dwMaximumSize += 0xFFF;
    dwMaximumSize &= 0xFFFFF000;

    // Specifying dwMaximumSize == 0 means that the heap is "growable."
    if ( dwMaximumSize == 0 );
    {
        fOptions |= HEAP_FREE_CHECKING_ENABLED;
        dwInitialSize &= 0xFFFFF000;
        dwMaximumSize = dwInitialSize + 0x100000;
    }

    fShared = fOptions & 0x04000000;    // Check for undocumented shared flag.
```

```
// Reserve the memory for the heap.
retValue = hHeap = VxDCall0(
                    _PageReserve,
                    fShared ? PR_SHARED : PR_PRIVATE,
                    dwMaximumSize >> 12,
                    ((fOptions & 0x80) >> 4) | PR_STATIC );
if ( retValue == -1 )   // Did allocation succeed?
{
    _DebugOut( "HeapCreate: reserve failed\n"
                SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_PARAMETER );

    return 0;
}

// Turn off all the flags that we don't care about.
fOptions &= ( HEAP_FREE_CHECKING_ENABLED | HEAP_GENERATE_EXCEPTIONS |
              HEAP_NO_SERIALIZE );

// Initialize the data fields of the heap header.
retValue = HPInit( hHeap, hHeap, dwMaximumSize, fOptions );

if ( retValue == 0 )    // Did the initialization fail?
{
    // Unreserve the memory we just reserved.
    VxDCall0( _PageFree, hHeap, 0x10 );

    return retValue;
}

// If it's the KERNEL32 heap, make the critical section effective in
// all processes.
if ( fShared && HKernelHeap )
    MakeCriticalSectionGlobal( hHeap + 0x70 );

if ( 0 == ppCurrentProcessId )  // If no current process, we're finished.
{
    _DebugOut( "HeapCreate: private heap created too early",
                SLE_ERROR );
}

// Insert the new heap at the head of the process's heap list.
hHeap->nextHeap = ppCurrentProcessId->HeapOwnList;
ppCurrentProcessId->HeapOwnList = hHeap;

return retValue;
```

HPInit(堆初始化函数)

HPInit 例程考虑了对新堆区段的初始化。HPInit 实施操作的堆可以是一个主体堆,也可以是一个离开主堆的子堆。对于后一种情况,堆首相当小。

在检查完初始边界之后,HPInit 调用 hpCommit 提交该堆的第一个页面。为什

么呢？因为堆首是在堆的第一个页面的起点写入的。在此提交之前，整个堆区域处于保留状态（而非提交状态）。HPInit 之后开始填写堆首区段。当某个正常的堆初始化时，HPInit 需要填写大量的区段，包括堆尺寸、符号 WORD、以及配置线程 ID。如果 HEAP_NO_SERIALIZE 标志未被特定（一般是不被特定的），HPInit 则调用 InitializeCriticalSection（初始化临界区）函数，通过堆首中 CRITICAL_SECTION 目标的地址。HPInit 在该点同时建立自由列首（free list headers）数组。

如果 HPInit 正初始化某个新子堆，则此初始化太小。此时，堆首仅包括两个 DWORD：堆区域的尺寸和一个指向下一子堆的指针。

在初始化完堆首区段之后，HPInit 从堆的尾端为一个 0 长度的 4KB 块生成一个场。因为最后一页最初同样处于未提交状态，HPInit 则要在写入之前调用 hpCommit 函数来提交此最后页面。由此，我们可以推论每个新堆至少占用了 8KB 要提交的物理内存：4KB 给第一页，4KB 给最后一页。

从堆区域的尾端生成长度为 0 的 4KB 标志场之后，HPInit 形成了一个大型的自由块。该自由块分布于堆首尾端与 0 长度标志场之间。为此，HPInit 利用了 hpFreeSub 函数（前面已经介绍过）。通过查看前面给出的 WALKHEAP 的输出中第一个堆，你可以看到 Win32 堆中块的初始设计。

HPInit 伪码

```
// Parameters:
//      HANDLE hHeap
//      PVOID  pHeapRegion
//      DWORD  size
//      DWORD  flags
// Locals:
//      HEAP_HEADER pHeap;
//      DWORD startPage, lastPage;
//      HEAP_ARENA * pLastArena;
//      HEAP_ARENA * pArena, * pArena2;
//      FREE_LIST_HEADER * pFreeListEntry, pFreeListArrayEnd;
//      PVOID pFirstArena;  // Pointer to first byte after HEAP_HEADER.
//      PDWORD pFreeListSize;
// Statics:
//      DWORD freeListSizes[4] = { 0x20, 0x80, 0x200, 0xFFFFFFFF };

    // Make sure the heap base and size are page-aligned.
    if ( (pHeapRegion & 0x00000FFF) || ( size == 0 ) || (size & 0x00000FFF) )
    {
        DebugOut( "HPInit: invalid parameter\n", SLE_ERROR );
    }

    pHeap = pHeapRegion;

    startPage = pHeapRegion >> 12;
```

```
// Commit the first page of the heap. We'll be writing a header there.
if ( !hpCommit(startPage, 1, flags) )
    return 0;

if ( !(flags & 0x100) )      // True if called from HeapCreate.
{                            // Not true if called from HeapAlloc.
    pHeap->nextHeap = 0;
    pHeap->nextBlock = 0;

    pFirstArena = pHeap + sizeof(HEAP_HEADER);

    pHeap->signature = 0x4948;  // 0x4948 = "HI"
    pHeap->flags = flags;
    pHeap->size = size;
    pHeap->checksum = ChecksumHeapBlock( pHeap, 1 );
    pHeap->allocating_EIP = x_GetCallingEIP():

    if ( ppCurrentThreadId )
        pHeap->creating_thread_ordinal
            = ppCurrentThreadId->processID->CurrentThreadOrdinal;
    else
        pHeap->creating_thread_ordinal = 0;

    if ( !(flags & HEAP_NO_SERIALIZE) ) // TRUE if serialization needed.
    {
        if ( HKernelHeap )  // KERNEL heap has already been initialized.
         {                   // This is typically the case.
             InitializeCriticalSection( &pHeap->criticalSection );
             pHeap->pCriticalSection = a field in pHeap->criticalSection;
         }
         else    // We're creating the KERNEL heap (the first heap).
         {
             pHeap->pCriticalSection = &pHeap->criticalSection
             some critical section init function(&pHeap->criticalSection);
         }
     }

     pFreeListArrayEnd = &pHeap->freeListArray
                       + (4 * sizeof(FREE_LIST_HEADER));
     pFreeListEntry = pHeap->freeListArray;

     pFreeListSize = freeListSizes;  // Point to array of free list sizes.

     // Build the array of free lists.
     while ( pListFreeEntry < pFreeListArrayEnd )
     {
         pFreeListEntry->dwMaxBlockSize = *pFreeListSize;
         pFreeListEntry->arena.size = 0xA0000001;
         pFreeListEntry->arena.signature = 0x4846;       // "FH"
         pFreeListEntry->arena.prev = previous free list entry;
         pFreeListEntry->arena.next = next free list entry;
         pFreeListEntry->freeBlockChecksum
             = ChecksumHeapBlock( &pFreeListEntry->arena, 4 );
         pFreeListEntry++;   // Point at next entry in free list array.
         pFreeListSize++;    // Point at next free list block size.
```

```
    }

    // Hook up the first and last free list arenas (the array of four
    // arenas near the beginning of the heap that point to four separate
    // free lists).

    // Point at arena in the first FREE_LIST_HEADER structure.
    pArena = &pHeap->freeListArray[0]->arena;

    // Point at arena in the last FREE_LIST_HEADER structure.
    pArena2 = &pHeap->freeListArray[3]->arena;

    pArena->next = pArena2;
    pArena2->prev = pArena;
    pArena->freeBlockChecksum = ChecksumHeapBlock( &pArena, 4);
    pArena->freeBlockChecksum = ChecksumHeapBlock( &pArena2, 4);

}
else    // TRUE if called from HeapAlloc. We're creating a subheap.
{
    pFirstArena = 8;
}

//
// At this point we're going to write the final arena at the
// end of the last page of this heap region.
//

pHeap->size = size;

pLastArena = pHeap + size - 0x10;
lastPage = pLastArena >> 12;

if ( size > 0x1000 )
{
    if ( !hpCommit( lastPage, 1, flags) )
    {
        // Decommit the starting page (we couldn't commit the last page).
        VxDCall0( _PageDecommit, startPage, 1, 0x20000000 );
        pHeap = 0;
        return 0;
    }
}

// Make the last block in the heap a zero-length in-use block.
pLastArena->size = 0xA0000000;
pLastArena->signature = 0x4842; // "BH"
pLastArena->checksum = ChecksumHeapBlock( pLastArena, 3 );

// Make an in-use block of length 0x10 bytes at the end of the heap.
if ( !(flags & 0x0400) && (size > 0x1000) )     // Comes through here in
{                                               // the typical case.
    size -= pFirstArena;
    size -= 0xFFC;
```

```
    pArena = pHeapRegion + size + pFirstArena;

    pArena->size = 0xA0000010;
    pArena->signature = 0x4842; // 0x4842 = "BH"

    pArena->checksum = ChecksumHeapBlock( pArena, 3);

    // Call hpFreeSub on this block.
    hpFreeSub( hHeap, pArena + 0x10, 0xFDC, 0);
}
else
{
        size -= pFirstArena;
        size -= 0x10;
    }

    // Make one huge free block out of the the region between the heap
    // header and the end of the heap.
    hpFreeSub( hHeap, pHeapRegion + pFirstArena, size, 0);

    if ( FParanoidHeapChecking )       // Verify the heap?
        hpWalk( pHeap );

    return pHeapRegion;
```

HeapDestory 和 IHeapDestory(堆破坏函数)

HeapDestory 也只不过是一个参量有效层存根。破坏某个 Win32 堆是由 IHeapDestory 来完成的。HeapDestory 所进行的唯一有效性工作是标准(假的)的 hHeap 有效化:此堆柄指向了一个长度至少为 0xD0 字节的内存区间吗?

HeapDestory 伪码

```
// Parameters:
//      HANDLE hHeap

    Set up structured exception handler frame

    // Make sure that the hHeap is valid. A heap handle is just a
    // pointer to the beginning of the heap area.
    AL = *(PBYTE)hHeap;
    AL = *(PBYTE)(hHeap + 0xCF);

    Remove structured exception handler frame

    goto IHeapDestroy;
```

与你所设想的或许正好相反,破坏一个 Win32 堆并不象将堆页面释放回操作系统那样简单。存在两点使问题复杂化。首先,所有生成的不具有 HEAP_NO_SE-

RIALIZE 属性的堆是由某个临界区目标所拥有的。IHeapDestory 查看该堆是否拥有这样一个目标并将适当地释放。

IHeapDestory 中其它复杂之处是堆的关联列表。如果 IHeapDestory 直接释放了堆页面，则此进程的堆的关联列将被破坏。通过行经堆列并适当更新该列，IHeapDestory 控制着这一点。

在链(chain)更新之后，IHeapDestory 调用 VMM_PageFree Win32 服务将堆页释放。对_PageFree 服务的一次调用可能不足以释放所有的堆页。为什么呢？如果堆用户已经进行了许多配置，或者是很大的配置，HeapAlloc 或许生成了附加的子堆并将这些子堆加到了子堆列表(堆首中的偏移量 4)上去了。因此，IHeapFree 采用循环来释放基本堆以及任何子堆块。

有关 HeapDestory 的最后一点说明是，当存在某个程序时，HeapDestory 不能由系统调用。因而，当程序地址空间移走时，所有的堆内存将被释放出来。

IHeapDestory 伪码

```
// Parameters:
//      HANDLE hHeap
// Locals:
//      DWORD  nextSubHeap;
//      DWORD   retValue;
//      HEAP_HEADER_DEBUG pHeap;
//      HANDLE currentHeap;

    EnterMustComplete();    // Prevent us from being interrupted.

    // Grab the heap semaphore. This allows only one thread at a time to
    // be in the heap code so that the heap doesn't get corrupted.
    retValue = hpTakeSem( hHeap, 0, 0);
    if ( !retValue )
    {
        LeaveMustComplete();
        return 0;
    }

    pHeap = hHeap;

    x_hpReleaseSem( hHeap, 0 );

    if ( !(hHeap->flags & HEAP_NO_SERIALIZE) )
    {
        if ( hHeap == HKernelHeap )
        {
            DestroyCrst( pHeap->pCriticalSection );
            goto elsewhere
        }
```

```
    else          //Not the KERNAL32 heap.
    {
        if ( (pHeap->pCriticalSection->Type & 0x7FFFFFFF) != 4 )
            _assert( line number, "..\lmem.c" );

        if ( (pHeap->pCriticalSection->Type & 0x7FFFFFFF) == 4 )
            some critsect deletion function( pHeap->pCriticalSection );
    }
}

if ( ppCurrentProcessId == 0 )
    goto HeapDestroy_free_it;

if ( hHeap == HKernelHeap )         // Is this the KERNEL heap?
    goto HeapDestroy_free_it;

if ( ppCurrentProcessId == HKernelProcess ) // Is this the KERNEL process?
    goto HeapDestroy_free_it;

if ( hHeap > 0x80000000 )       // Is it a shared heap?
    goto HeapDestroy_free_it;

if ( 0 == ppCurrentProcessId->HeapOwnList )     // No heaps in this
    goto HeapDestroy_not_in_list;               // process? Oops!

//
// We have to walk through the list of heaps for this process. After
// we free the heap region, we need to update the linked list of heaps.
//

// Start at the first heap.
currentHeap = ppCurrentProcessId->HeapOwnList;

if ( currentHeap == hHeap ) // Are we destroying the default (main) heap?
{
    // Yes!
    ppCurrentProcessId->HeapOwnList = currentHeap->nextHeap;
    goto HeapDestroy_free_it;
}
if ( !currentHeap->nextHeap )       // Hmmm...There are no other heaps.
    goto HeapDestroy_not_in_list;   // How can we free it? Complain!

if ( ppCurrentProcessId->HeapOwnList->nextHeap == hHeap )
{
    currentHeap->nextHeap = pHeap->nextHeap;
    goto HeapDestroy_free_it;
}

do
{
    if ( currentHeap->nextHeap == hHeap )
    {
        currentHeap->nextHeap = pHeap->nextHeap;
        goto HeapDestroy_free_it;
```

```
        }

        currentHeap = currentHeap->nextHeap;
    } while ( currentHeap->nextHeap->nextHeap );

HeapDestroy_not_in_list:

    _DebugOut( "HeapDestroy: Heap not on list", SLE_ERROR );

HeapDestroy_free_it:

    nextSubHeap = hHeap->nextBlock; // Determine whether there's another
                                    // subheap block chained onto this one.

    // Free the range of memory.
    VxDCall0( _PageFree, hHeap, 0x10 );

    if ( nextSubHeap )      // If there is another block, loop back and
    {                       // unreserve it as well.
        hHeap = nextSubHeap;
        goto HeapDestroy_free_it;
    }

    LeaveMustComplete();    // We can now be interrupted. A lot of good
                            // it'll do though!

    return retValue;    // Value returned from hpTakeSem.
```

HeapValidate(堆有效化函数)

HeapValidate 是一个 Windows NT 函数,它流览 Win32 并检查其一致性。当你认为在 hpTakeSem 中存在使堆升效的代码时,不要为 Windows 95 API 没有此函数而抱歉。

请参照 VirtualLock 的有关 CommonUnimpStub 如何工作的详细介绍。

HeapValidate 伪码

```
EAX = "HeapValidate"
CL = F3
JMP CommonUnimpStub
```

HeapCompact(堆压缩函数)

HeapCompact 也为一 Windows NT 函数,它试图将自由块合并并下放 Win32 堆中的未使用页面。从表面上看,Windows 95 作的是一些一般管理方面的事务,因而可能此函数不大需要。

请参照 VirtualLock 的有关 CommonUnimpStub 如何工作的详细介绍。

HeapCompact 伪码

```
EAX = "HeapCompact"
CL = 2
JMP CommonUnimpStub
```

GetProcessHeaps(获取进程堆函数)

GetProcessHeaps 为一 Windows NT 函数,它返回了某个进程的堆柄数组。奇怪的是,它并不在 Windos 95 的 API 中,尽管看似应当提供出来。事实上,函数 TOOLHELP32 Heap32ListFirst 和 Heap32ListNext(为 Win32 工具函数)将给出这方面的信息。

请参照 VirtualLock 的有关 CommonUnimpStub 如何工作的详细介绍。

GetProcessHeaps 伪码

```
EAX = "GetProcessHeaps"
CL = 2
JMP CommonUnimpStub
```

HeapLock(堆锁定函数)

HeapLock 为一 Windows NT 函数,它掌握着当前线程的 Win32 堆临界区目标。令人费解的是,Windows 95 的 API 也没有考虑这个函数。HPAlloc 所使用的 hpTakeSem 函数似乎刚好完成了你期待 HeapLock 函数要作的工作。

请参照 VirtualLock 的有关 CommonUnimpStub 如何工作的详细介绍。

HeapLock 伪码

```
EAX = "HeapLock"
CL = 1
JMP CommonUnimpStub
```

HeapUnlock(堆解锁函数)

HeapUnlock 为一 Windows NT 函数,它释放了某个 Win32 堆的临界区目标。像 HeapLock 一样,Windows 95 的 API 也没考虑此函数。(同行,你能说"正好可以得到?")

请参照 VirtualLock 的有关 CommonUnimpStub 如何工作的详细介绍。

HeapUnlock 伪码

```
EsAX = "HeapUnlock"
CL = 1
JMP CommonUnimpStub
```

HeapWalk(堆行走函数)

HeapWalk 为一 Windows NT 函数,它累接 Win32 堆的所有块。此 API 是一个"很好的"事例,它说明了 Win32 API 的变化。Windows 95 编码器在定义 Windows 95 API 时忽略了 HeapAlloc。但是,在决定从 Windows 95 API 的子设备中删去 HeapWalk 之后,他们又追加了 TOOLHELP32 Heap32First 和 Heap32Next 函数。

参照 VirtualLock 的有关 CommonUnimpStub 如何工作的详细介绍。

HeapWalk 伪码

```
EAX = "HeapWalk"
CL = 1
JMP CommonUnimpStub
```

Win32 局部和全局堆函数 (Local and Global Heap Functions)

Win32 局部和全局堆函数是 Win16 时代的滞留品——因而在 Windows 95 中是不必要的。局部堆是在 Win16 中生成的,因此应用程序和 DLL 应可以在无需改变选择器的情况下到达这些堆数据。与此类似,由于找不到办法来配置大的内存区域而无需处理选择器,使得 Win16 中还要有全局堆。Windows 95 之下的 Win32 程序则没有这类限制。所以从理论上讲,Win32 API 应当不再需要全局和局部堆了。

众所周知,Win32 API 为了考虑到兼容性的限制而采取了一些补偿。由于有太多的 Win16 程序使用了全局和局部堆函数,从 Win32 API 中移去它们会给 Win32 进程加重负担。因此,Microsoft 保留了这类函数并在堆函数的 Win32 版本与 Win16 版本之间保留了相同等级的语议(semantics)。

总体上讲,Windows 95 局部和全局堆函数是明确的。即,GolbalAlloc 和 LocalAlloc 函数都是输出了的,但位于 KERNEL32. DLL 中的同一地址。同样,GlobalFree 和 LocalFree 也是相同的函数。下面,在函数的伪码中,我将指出所有的不同点。在检查 Windows 95 的执行情况时,我发现通常的 Global/Local 函数是由局部

名称(Local name)指明的。此处我也遵守这样的常规。

可以广泛利用 Win32 局部堆函数的一种代码是 Windows 95 的 Win16 部分。Windows 95 的 USER(用户)和 GDI(通用数据库接口)仍处于 16 位代码段,但在许多情况下,使用了 32 位指针,诸如 HWNDs、menus、和 GDI 目标。这些目标保留在 Win32 的局部堆中,这些局部堆可以立即驻留于内存中的 USER 和 GDI DGROUP 区段之上。第四章详细介绍了其总体设计。从内存管理的角度来讲,重要之处在于 KRNL386 输出了那些通过调进 KERNEL32 来使用 Win32 局部堆函数的 16 位函数。例如,非资源化的 K209 函数(KRNL386 输出了 209)转换到了 KERNEL32 的 LocalAlloc 函数。16 位的 USER 和 GDI 调用 K209 来为窗、设备文本等配置内存。与此类似,一个类似的函数(K211)调用了 KERNEL32 的 LocalFree 函数。

Win32 局部堆

Windows 95 中的局部堆要比 Win16 中的更简单一些。Win32 局部堆函数使用的是我前面已经介绍过的基本的 Win32 堆代码。这大大简化了局部堆函数中的代码。例如,利用 LMEM_FIXED 标志调用 LocalAlloc 实质上与调用 HeapAlloc 函数是完全一样的。这样,LocalAlloc 和 HeapAlloc 都可以调用 KERNEL32 HPAlloc 函数了。

Win32 局部堆函数的另一个区域在局部堆的绝对数目(sheer number of local heaps)方面也比基于 Win16 的更简单。在 Win16 中,可执行程序有其自己的局部堆,它使用的每个 DLL 也是同样(明显的例外是诸如字符文件的 DLL)。通过缺省,Windows 95 进程只有一个 Win32 局部堆。通过 Win32 局部堆的 API 函数完成的配置来自缺省的进程堆(前面已经介绍)。如果目的不是为了 LMEM_MOVEABL 块,你可以这样简单地给出 LocalAlloc:

```
HLOCAL WINAPI LocalAlloc(UINT uFlags, UINT cbBytes)
{
    return (HLOCAL) HeapAlloc( GetProcessHeap(), 0, cbBytes );
}
```

在伪码中你会马上看到,带有 LMEM_FIXED 标志的 LocalAlloc 并不比假想的工具更加复杂多少。LMEM_MOVEABLE 块的追加使 Win32 局部堆更加复杂。你可能会问,"那么为什么要提供 LMEM_MOVEABLE 呢? 为何不将那个标志忽略而进行与 LMEM_FIXED 相同的工作呢?"以 LMEM_MOVEABLE 配置的内存是不能在它所配置的堆内移动的。而且,KERNEL32 不能仅只抓住 LMEM_MOVEABLE 标志。许多应用程序(apps)(包括 Windows 本身)都利用了这样的事实,即 LMEM_MOVEABLE 句柄是一个真正的指向内存块的指针:

```
pMemoryBlock = *(void *)_LMEM_MOVEABLE_handle;
```

通过将句柄看成一个指针并再定位它，这些应用程序可以获得一个指向联合内存块的指针而无需调用 LocalLock。尽管这并非一个好的编程方法，但它一旦广泛采用，你会认同的。

Win32 局部堆函数与其 Win16 系列具有相同的语议规则。如果再定位一个 LMEM_MOVEABLE 句柄，你将得到一个指向联合内存块的指针。关键区别是，在 Win16 中它是一个 2 字节的近指针，而在 Win32 中则是一个 4 字节的近指针。为了保证与 Win16 的兼容性，Win32 局部堆函数使用了并无新意的句柄表(handle tables)。正如我在 Windows Internals 的第二章所述，Win16 局部堆函数也同样使用了句柄表，尽管格式不同。

每个 Win32 局部堆柄表保留了直到 8 个局部句柄的信息。当一个程序一次使用多于 8 个的局部句柄时，LocalAlloc 就会为另外 8 个句柄配置附加的局部句柄表。句柄表是从它们参照的内存块的相同堆中配置的。这些表被保留在一个联合的列表中以便于寻找某个自由柄表入口。指向柄表列首部的指针保留在进程库中。句柄表的形式如下：

```
struct HANDLE_TABLE          // Size == 0x48 bytes
{
    WORD    signature;           // "LA" (0x414C)

    WORD    cHandleTables;       // Number of previously allocated
                                 // handle tables - 1.
    DWORD   pPrevHandleTable;    // Pointer to previous handle table.

    LOCAL_HANDLE_TABLE_ENTRY     handleEntries[8];
};
```

每个 LOCAL_HANDLE_TABLE_ENTRY 形成如下：

```
struct LOCAL_HANDLE_TABLE_ENTRY
{
    WORD    signature;  // "BS" (0x5342) if an in-use entry.
                        // "FS" (0x5346) for free entries.
    union
    {
        PVOID   pBlock;     // If in-use: pointer to data block.
        PVOID   pNextFree;  // If free: Points to next free
                            // LOCAL_HANDLE_TABLE_ENTRY.
    } x;

                            // These two fields are valid for in-use blocks.
    BYTE    flags;          // 0x02 == discardable
    BYTE    cLock;          // Lock count of the block.
}
```

当你利用 LMEM_MOVEABLE(或 GMEM_MOVEABLE)分配内存时，LocalAlloc 需要在句柄表列中寻找一个可用的 LOCAL_HANDLE_TABLE_ENTRY 槽(slot)。在找到自由入口之后，它将按所需尺寸配置一个块并将块地址置入

LOCAL_HANDLE_TABLE_ENTRY 的 pBlock 区段。LocalAlloc 所返回的“句柄”是 LOCAL_HANDLE_TABLE_ENTRY 的 pBlock 区段的地址。

假定 LMEM_FIXED 块只是一个指向内存的指针而 LMEM_MOVEABLE 则不是，你可能想知道 KERNEL32 是如何认知你正在使用的句柄的形式的。例如，你可以将 LMEM_FIXED 或 LMEM_MOVEABLE 句柄通过 LocalFree。KERNEL32 是如何区分它们的？实际上是很容易的。结束于 0、4、8、或 0xC 的局部堆柄是固定的块。句柄之间的区别是由设计给定的。由 HPAlloc 返回的所有内存块具有结束于 0、4、8、或 0xC 的地址。为了使运动块总是结束于 2、6、0xA、或 0xE，Microsoft 将 pBlock 指针的两个字节置入 LOCAL_HANDLE_TABLE_ENTRY 结构。出于偶然，Win16 局部堆柄表在这方面有着类似的设计形式。

LocalAlloc 和 ILocalAlloc(局部配置函数)

LocalAlloc 代码看上去并不多。它是对 KERNEL32 内部函数的一个调用，并由一个对 ILocalAlloc 的跳入跟随。从字面上看，HouseCleanLogicallyDeadHandles(逻辑清除无效句柄)似乎也作着同样的工作。但是，我在实际中还从未看到过它可以这样，因而“逻辑无效句柄”的意义并不明确。

LocalAlloc 伪码

```
HouseCleanLogicallyDeadHandles();
goto ILocalAlloc;
```

ILocalAlloc 是从查看进程库中的缺省进程堆的地址开始的。其后，它获取进程堆的堆信号量(heap semaphore)，从而允许代码将 HEAP_NO_SERIALIZE 标志经过 ILocalAlloc 后面使用的低级函数。这样，ILocalAlloc 分成两个路径，一路是对于 LMEM_MOVEABLE 块的，另一路是对于 LMEM_FIXED 块的。

在配置了一个 LMEM_MOVEABLE 块之后，ILocalAlloc 在进程库中查找自由柄列的首部。如果该自由柄列是空着的，ILocalAlloc 将利用 HPAlloc 为某个新的柄表分配内存并将此新表初始化。无论采用何种方式，ILocalAlloc 最终将得到一个自由柄表的入口。利用此入口，ILocalAlloc 将区段填写从而表明它是一个在使用的句柄。

在填写完几乎所有的柄表入口之后，ILocalAlloc 则调用 HPAlloc 获取一块大小由 LocalAlloc 给定的内存块。ILocalAlloc 将 4 个字节加到此配置尺寸上去，以便它可以根据自己的需要来使用配置的第一个 DWORD。ILocalAlloc 将可能把什么置入此首 DWORD 呢？只不过是一个指向柄表入口的指针而已。由此，局部堆函数可以将某个指针转换成一个可移动的内存块而返回其句柄。由于配置块的首 DWORD 是由局部堆函数使用的，当将块指针存放到柄表槽(handle table slot)时，ILocalAlloc 将把 4 加到 块地址上去。

ILocalAlloc 可使用的其它代码路径是由 LMEM_FIXED 句柄使用的。这时，

代码调用 HPAlloc 来得到内存块。块地址是 ILocalAlloc 作为句柄时所返回的结果。采用另外一种方式，局部堆中 LMEM_FIXED 的句柄与其地址是一样的。进一步讲，它与 Win16 局部堆函数是一样的。

ILocalAlloc 伪码

```
// Parameters:
//      UINT uFlags;
//      UINT uBytes;
// Locals:
//      HANDLE hHeap;
//      DWORD retHandle;
//      LOCAL_HANDLE_TABLE_ENTRY *pFreeHandle, *pHandleEntry;
//      LOCAL_HANDLE_TABLE * pHandleTable;
//      PVOID pBlock;

    // Get the default process heap from the process database.
    hHeap = ppCurrentProcessId->lpProcessHeap;

    uFlags &= 0xFFFF8FFF;   // Turn off LMEM_INVALID_HANDLE bit if set.

    // Acquire the heap semaphore so that we're not interrupted.
    x_WaitForSemaphore( hHeap->pCriticalSection );

    if ( uFlags & 0xFFFF808D )  // Check for any invalid or undefined flags,
    {                           // e.g., LMEM_INVALID_HANDLE or LMEM_MODIFY.
        _DebugOut( "LocalAlloc: invalid flags\n",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_INVALID_PARAMETER );
        goto return_0;
    }

    if ( uFlags & LMEM_MOVEABLE )
    {
        // pNextFreeHandle is at offset 0x58 in Process Database.
        pFreeHandle = ppCurrentProcessId->pNextFreeHandle;
        if ( pFreeHandle )
            goto have_handle_table

        // Hmmm...There's no available LOCAL_HANDLE_TABLE_ENTRYs.
        // Go create a new handle table.
        pHandleTable = HPAlloc( hHeap, 0x48, HEAP_NO_SERIALIZE );
        if ( !pHandleTable )
            goto return_0;

        // Initialize the new handle table.
        pHandleTable->signature = "LA"; // "LA" = 0x414C
        // KERNEL32 keeps a linked list of LOCAL_HANDLE_TABLEs. Insert
        // the new table at the head of the list.

        if ( ppCurrentProcessId->pHandleTableHead )
```

```
        {
            pHandleTable->cHandleTables =
                    ppCurrentProcessId->pHandleTableHead->cHandleTables+1;
        }
        else
            pHandleTable->cHandleTables = 0;

        // Point to first entry in the array of LOCAL_HANDLE_TABLE_ENTRYs,
        // then initialize the 8 elements of the LOCAL_HANDLE_TABLE_ENTRY
        // array.
        pHandleEntry = pHandleTable + sizeof(LOCAL_HANDLE_TABLE);
        pFreeHandle = pHandleEntry;
        while ( pHandleTable2 < end of handle table )
        {
            pHandleEntry->signature = "FS"
            pHandleEntry->pNextFree = pHandleEntry + 8;
            pHandleEntry += sizeof( LOCAL_HANDLE_TABLE_ENTRY );
        }

        // Add the new handle table to the head of the list of handle
        // tables. The pointer to the list head is kept in the process
        // database.
        pHandleTable->pPrevHandleTable=ppCurrentProcessId->pHandleTableHead;
        ppCurrentProcessId->pHandleTableHead = pHandleTable;

have_handle_table:
        if ( pFreeHandle->signature != "FS" )
            _DebugOut( "LocalAlloc: bad handle free list 2\n", 1 );

        // Remove this handle entry from the list of free entries.
        ppCurrentProcessId->pNextFreeHandle = pFreeHandle->pNextFree;

        // Modify the handle entry to describe the new block.
        pFreeHandle->cLock = 0;
        pFreeHandle->signature = "BS";
        pFreeHandle->flags = 0;

        if ( (uFlags & LMEM_DISCARDABLE) == LMEM_DISCARDABLE )
            pFreeHandle->flags |= 2;

        if ( uBytes == 0 )
            goto moveable_0_bytes;
        if ( uBytes > 0xFFFFF98 )
        {
            _DebugOut( "LocalAlloc: requested size too big\n",
                        SLE_WARNING + FStopOnRing3MemoryError );
            InternalSetLastError( ERROR_NOT_ENOUGH_MEMORY );

            goto moveable_alloc_error;
        }

        // Call HeapAlloc to allocate the memory block of the requested size.
        // Add an extra 4 bytes, because the back pointer to the handle
        // table entry needs to be stored in the first 4 bytes.
```

```
        pBlock = HPAlloc( hHeap, uBytes+4, flags & HEAP_NO_SERIALIZE );
        if ( !pBlock )
            goto moveable_alloc_error;

        // Store the pointer to the data area in the handle table entry.
        pFreeHandle->pBlock = &pBlock + 4;

        // Store a pointer to the handle table entry in the first 4 bytes
        // of the allocated block.
        *(PDWORD)pBlock = pFreeHandle;

        retHandle = &pFreeHandle->pBlock;
        goto moveable_alloc_done

moveable_alloc_0_bytes:
    pFreeHandle->pBlock = 0;

moveable_alloc_done:

        if ( (retHandle & 2) == 0 )
            _DebugOut( "LocalAlloc: handle value w/o LH_HANDLEBIT set\n", 1);

        goto return_retHandle;
    }

    // This code allocates LMEM_FIXED blocks.

    // Call HeapAlloc to allocate the memory block of the requested size.
    pBlock = HPAlloc( hHeap, uBytes, flags & HEAP_NO_SERIALIZE );

    if ( pBlock )
    {
        // Verify that HeapAlloc returned a pointer that's a multiple of 4.
        // (LMEM_FIXED blocks must be a multiple of 4.
        if ( pBlock & 2 )
            _DebugOut("LocalAlloc: pointer value w/ LH_HANDLEBIT set\n", 1);
         retHandle = pBlock;
         goto return_retHandle;
     }

 moveable_alloc_error:

     // Put the LOCAL_HANDLE_TABLE_ENTRY that we acquired earlier back
     // into the free list of LOCAL_HANDLE_TABLE_ENTRYs.
     pFreeHandle->pNextFree = ppCurrentProcessId->pNextFreeHandle;
     ppCurrentProcessId->pNextFreeHandle = pFreeHandle;
     pFreeHandle = "FS"; // (0x5346)

 return_0:
     retHandle = 0;

 return_retHandle:
     InternalLeaveCriticalSection( hHeap->pCriticalSection );
     return retHandle;
```

LocalLock 和 ILocalLock(局部锁定函数)

在 Win16 中,LocalLock 函数有两个用途:一是防止某个块被移动,二是返回与句柄相关的内存地址。在 Win32 中,LocalLock 则主要是一个句柄升效函数,尽管它也返回了相关块的地址。在 Win32 中,局部堆块是不四处移动的,因此无需锁定堆。同时由于你无法真正得到可移动的内存,因而没有理由在起始位置分配 LMEM_MOVEABLE 块。而且,KERNEL32 也进行了锁定计算。

实际的 LocalLock 函数是有效层的一部分。它检验所通过的 hLock 在从 0x10 到达 7 字节之前以及在该指针之后,是可以得到的。任何句柄——LMEM_FIXED 或 LMEM_MOVEABLE——均应满足此准则。假定测试成功,LocalLock 则跳到 ILocalLock。

如果通过 ILocalLock 句柄的是一个 LMEM_MOVEABLE 句柄,则函数要从该句柄中减去二个字节,从而得到了指向块的 LOCAL_HANDLE_TABLE_ENTRY(局部柄表入口)的指针。利用这个指针,ILocalLock 验证了特征(signature)(BS)并恢复了当前的锁计数(the current lock count)(为一个字节)。如果锁计数是 0xFE,ILocalLock 则拒绝锁计数的任何增量。否则,函数则增加 LOCAL_HANDLE_TABLE_ENTRY 结构中的锁计数并返回指向联合内存的指针。

如果给定 ILocalLock 的指针是 LMEM_FIXED,将没有任何锁计数为其保留。而且,ILocalLock 趁机检验了句柄。如果块已经由 HPAlloc 分配,则此时的句柄应该是相同的。因此,在句柄/地址(handle/address)之前应当存在一个 HPAlloc 型的大小为 0x10 字节的场。LocalAlloc 从该场得到了尺寸段(size field)并检查是否为在使用块设置了合适的位。LocalAlloc 为某个有用的 LMEM_FIXED 块所返回的地址与所通过的句柄是一样的。

LocalLock 伪码

```
// Parameters:
//      HLOCAL  hLocal

    Set up a structured exception handler frame

    AL = *(PBYTE)(hLocal + 7 );     // If the pointer is bogus, these will
    AL = *(PBYTE)(hLocal - 0x10 );  // fault, and the exception handler
                                    // returns a failure value to the caller.
    Remove structured exception handler frame

    goto ILocalLock
```

ILocalLock 伪码

```
// Parameters:
//      HLOCAL  hLocal
// Locals:
```

```
//      HANDLE hHeap;
//      PSTR pszError
//      BYTE lockCount;
//      HEAP_ARENA pHeapArena;
//      LOCAL_HANDLE_TABLE_ENTRY *pHandleEntry
//      DWORD retValue;

    // Get the default process heap from the process database.
    hHeap = ppCurrentProcessId->lpProcessHeap;

    // Acquire the heap semaphore so that we're not interrupted.
    x_WaitForSemaphore( hHeap->pCriticalSection );

    // Verify that the local handle is even with the range of valid handles.
    if ( !x_IsHandleInRange(hHeap, hLocal) )
    {
        pszError = "LocalLock: hMem out of range\n";
        goto error;
    }

    if ( hLocal & 2 )        //A moveable block.
    {
        // The handle points 2 bytes into the LOCAL_HANDLE_TABLE_ENTRY
        // struct. Subtract 2 bytes to get a pointer to the
        // LOCAL_HANDLE_TABLE_ENTRY.
        pHandleEntry = hLocal - 2;

        if ( pHandleEntry->signature != "BS" )  // "BS" = 0x5342
        {
            pszError = "LocalLock: invalid hMem, bad signature\n";
            goto error;
        }

        lockCount = pHandleEntry->cLock;

        // Make sure the lock count isn't going to overflow.
        if ( lockCount == 0xFE )
        {
            _DebugPrintf("LocalLock: lock count overflow, handle "
                         "cannot be unlocked\n");
        }

        if ( lockCount != 0xFF )    // If lockCount != 0xFF, bump it up.
        {
            lockCount++;
            pHandleEntry->cLock = lockCount;
        }

        // Return the address of the associated data block.
        retValue = pHandleEntry->pBlock
        goto return_retValue;
    }
    else    // A fixed block.
    {
        // The hLocal parameter is just the pointer to the data.
        // Back up to the HEAP_ARENA structure.
```

```
        pHeapArena = hLocal - 0x10;

        // Are the bits indicating an in-use block set in the
        // HEAP_ARENA size field?
        if ( (pHeapArena->size & 0xF0000003) != 0xA0000001 )
        {
            pszError = LocalLock: hMem is pointer to free block\n;
            goto error;
        }
        retValue = hLocal;      // Just return the handle parameter, because
                                // it points directly to the block's memory.
        goto return_retValue;
    }

error:
    _DebugOut( pszError, SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_HANDLE );
    retValue = 0;

return_retValue:
    InternalLeaveCriticalSection( hHeap->pCriticalSection );
    return retValue;
```

LocalUnlock(局部解锁函数)

LocalUnlock 函数是有效层的一部分。它检验所通过的 hLocal 在 0x10 字节到 7 字节之前以及在此指针之后是有效的。任何句柄——LMEM_FIXED 或 LMEM_MOVEABLE——均应符合此准则。假定测试是成功的,LocalUnlock 则跳到了 ILocalUnlock。

ILocalUnlock 代码是 ILocalLock 代码的一个再现,但正好相反。如果句柄参数是一个 LMEM_FIXED 句柄,ILocalUnlock 则不用作任何事情。它甚至不用象 LocalLock 那样将句柄升效。如果句柄是一个可能的 LMEM_MOVEABLE 句柄,ILocalUnlock 则检验柄表入口中的特征字节(signature byte)以确信它为一个有效句柄。如果是这样,ILocalUnlock 则检验块的锁计数,看其是否对减量安全。如果安全,LocalUnlock 则递减锁计数器并返回一个 BOOL(布尔符号)以表明此块是否依然锁定。

LocalUnlock 伪码

```
// Parameters:
//      HLOCAL  hLocal

    Set up a structured exception handler frame

    AL = *(PBYTE)(hLocal + 7 );     // If the pointer is bogus, these will
    AL = *(PBYTE)(hLocal - 0x10 );  // fault, and the exception handler
                                    // returns a failure value to the caller.
    Remove structured exception handler frame

    goto ILocalUnlock
```

ILocalUnlock 伪码

```
// Parameters:
//      HLOCAL  hLocal
// Locals:
//      HANDLE hHeap;
//      PSTR pszError
//      BYTE lockCount;
//      LOCAL_HANDLE_TABLE_ENTRY *pHandleEntry
//      DWORD retValue;

    retValue = 0;       // FALSE: the block isn't locked.

    // Get the default process heap from the process database.
    hHeap = ppCurrentProcessId->lpProcessHeap;

    // Acquire the heap semaphore so that we're not interrupted.
    x_WaitForSemaphore( hHeap->pCriticalSection );

    // Verify that the local handle is even with the range of valid handles.
    if ( !x_IsHandleInRange(hHeap, hLocal) )
    {
        pszError = "LocalUnlock: hMem out of range\n";
        goto error;
    }

    if ( (hMem & 2) == 0 )      // If it's a FIXED block, there's nothing to do.
        goto return_retValue;

    if ( pHandleEntry->signature != "BS" )  // "BS" = 0x5342
    {
        pszError = "LocalUnlock: invalid hMem, bad signature\n";
        goto error;
    }

    // The handle points two bytes into the LOCAL_HANDLE_TABLE_ENTRY struct.
    pHandleEntry = hLocal - 2;

    // A lock count of 0xFF seems to be some sort of error condition.
    if ( pHandleEntry->cLock == 0xFF )
        goto return retValue;

    // Make sure the lock count won't underflow.
    if ( lockCount == 0 )
    {
        _DebugOut( "LocalUnlock: not locked" );
        goto return_retValue;
    }

    // Decrement the lock count in the handle table entry.
    pHandleEntry->cLock--;


    if ( pHandleEntry->cLock )
        retValue = 1;       // Return TRUE (the block is still locked).
```

```
    goto return_retValue;

error:
    _DebugOut( pszError, SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_HANDLE );
    EDI = 0;

return_retValue:
    InternalLeaveCriticalSection( hHeap->pCriticalSection );
    return retValue;
```

LocalFree 和 ILocalFree(局部释放函数)

Win32 局部释放函数 LocalFree 是相当奇特的。在它为释放一个被局部配置了的句柄(LocalAlloc'ed handle)得到真正代码之前,它首先要为某个特殊的句柄进行检查。从某种程度上讲,KERNEL32 和 KRNL386 是共同来生成并使用句柄组的(handle groups)。这些句柄组到底是什么对我来讲确实是个谜,因为我还尚未发现过它们。无论如何,句柄组是某个 Win16 任务数据库(Task database)、已局部配置了的 Win32 句柄(a Win32 LocalAlloc'ed handle)、和某个句柄组(handle group)之间的一个所谓三方联合体。当 LocalFree 探知此特殊局部句柄已被释放时,它将调用 GlobalNukeGroup 函数来摆脱此句柄组。柄组列(handle group list)是由 KRNL386 来维护的,因而 GlobalNukeGroup 以调用至 KRNL386 而结束。这一点也驳斥了 Microsoft 有关 KERNEL32 不能切换到 KRNL386 的观点。绝大部分时间是用于调用 LocalAlloc,而不是为了某个柄表的句柄(a handle table handle)。在这种情况下,LocalFree 变成了一个对 ILocalFree 的调用。

LocalFree 程序

```
// Parameters:
//      HLOCAL hMem

    _CheckSysLevel( x_Another_Win16_mutext );

    CheckHGHeap();      // Check Handle Group Heap. Thunks down to KRNL386.

    _EnterSysLevel( x_Another_Win16_mutext );

    if ( *someGlobal )      // *someGlobal points to a Handle Group selector.
    {
        // This is a loop that iterates through a list. This list
        // associates a Win16 TDB with a Win32 LocalAlloc handle and a
        // "handle group" (whatever that is). The node is considered found
        // if the TDB and local handle match the current thread's TDB
        // and the handle passed to this function.
        while ( not at end of list )
        {
```

```
            if ( the node being searched for is found )
            {
                _LeaveSysLevel( x_Another_Win16_mutext )
                GlobalNukeGroup( EBX );
                HouseCleanLogicallyDeadHandles();
                return hMem;
            }
            go to next node in list
        }
    }

    _LeaveSysLevel( x_Another_Win16_mutext )

    CheckHGHeap();      // Check Handle Group Heap yet again.

    return ILocalFree( hMem );
```

ILocalFree 代码位于某个被局部配置了的句柄所释放的地方。对于绝大部分其它的 Win32 局部堆函数而言，用于处理 LMEM_FIXED 块的代码是比较简单的；实际上，它是对于某个现行堆函数的调用。对于 LocalFree 这种情况，代码则仅仅调用 IHeapFree。

释放 ILocalFree 中的某个 LMEM_MOVEABLE 块是相当复杂的。在验证了某个有效局部堆柄通过之后，ILocalFree 要检查块的锁计数器。如果计数不为零，ILocalFree 则抗议该块仍被锁定。接下来，ILocalFree 将通过 IHeapFree 函数，把和句柄参量相关的句柄释放回堆。最后，ILocalFree 将柄表入口置于可得到的柄表入口列的头部。

有趣的是，当其所有的 8 个入口都未使用时，ILocalFree 并不试图将某个柄表删除。也就是说，返回到空表中并不是一个好的再循环。为了确保自己并没忽略某些细节，我修改了一份 WALKMHEAP 程序的备份在某行生成了 50 个 LocalAllocs 并释放了此 50 个句柄。输出结果表明，所有的柄表是保留在内存当中的。作为一个追加的项目(实际上并不是!)，堆具有一个良好的、规则的碎段形式。唯一的安慰是柄表将用于未来的可移动内存。

ILocalFree 伪码

```
// Parameters:
//      HLOCAL hMem
// Locals:
//      HANDLE hHeap;
//      DWORD retValue;
//      LOCAL_HANDLE_TABLE_ENTRY *pHandleEntry;

    Set up structured exception handler frame
```

```
// Get the default process heap from the process database.
hHeap = ppCurrentProcessId->lpProcessHeap;

// Acquire the heap semaphore so that we're not interrupted.
x_WaitForSemaphore( hHeap->pCriticalSection );

retValue = hMem;

if ( hMem & 2 )     // A moveable block (bit 1 set)?
{
    if ( !x_IsHandleInRange(hHeap, hMem) )
    {
        _DebugOut( "LocalFree: hMem out of range\n",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_INVALID_HANDLE );
        goto return_retValue;
    }

    // Back up two bytes to point at the handle table entry.
    pHandleEntry = hMem - 2;
if ( pHandleEntry->signature != "BS" )  // 0x5342
{
    _DebugOut( "LocalFree: invalid hMem, bad signature\n",
               SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_HANDLE );
    goto return_retValue;
}

// If the handle is still locked, complain.
if ( pHandleEntry->cLock )
{
    _DebugOut( "LocalFree: invalid handle\n",
               SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_HANDLE );
}

// If the memory block hasn't been discarded, free it with IHeapFree.
// Note that the code subtracts 4 from the pBlock field to get
// the original value returned by HeapAlloc.
if ( pHandleEntry->pBlock )
    if ( IHeapFree(hHeap, HEAP_NO_SERIALIZE, &pHandleEntry->pBlock-4))
    {
        retValue = pHandleEntry;
        goto return_retValue;
    }

// Insert the handle being freed at the head of the free handle list.
pHandleEntry->pNextFree = ppCurrentProcessId->pNextFreeHandle;
ppCurrentProcessId->pNextFreeHandle = pHandleEntry;

// Set the handle table entry's signature back to the free version.
pHandleEntry->signature = "FS"; // 0x5346
```

```
        retValue = 0;
    }
    else    // A fixed block.
    {
        if ( IHeapFree(hHeap, HEAP_NO_SERIALIZE, hMem) )
            retValue = hMem
        else
            retValue = 0;


return_retValue:
    InternalLeaveCriticalSection( hHeap->pCriticalSection );

    Remove structured exception handler frame

    return retValue;
```

LocalReAlloc 和 ILocalReAlloc(局部再配置函数)

LocalReAlloc 函数是有效层的一部分。它检验所通过的 hLocal 在 0x10 字节到 7 字节之前以及在此指针之后是有效的。任何句柄——LMEM_FIXED 或 LMEM_MOVEABLE——均应符合此准则。假定测试是成功的,LocalReAlloc 则跳到了 ILocalReAlloc。

LocalReAlloc 伪码

```
// Parameters:
//      HLOCAL  hLocal
//      UINT uBytes;
//      UINT uFlags;

    Set up a structured exception handler frame

    AL = *(PBYTE)(hLocal + 7 );     // If the pointer is bogus, these will
    AL = *(PBYTE)(hLocal - 0x10 );  // fault, and the exception handler
                                    // returns a failure value to the caller.
    Remove structured exception handler frame

    goto ILocalReAlloc
```

ILocalReAlloc 是 KERNEL32 中最复杂、最长的堆函数。与其它局部堆函数一样,代码分成了 LMEM_FIXED 块区和 LMEM_MOVEABLE 块区。LMEM_FIXED 代码相当简单,并包括对 HPReAlloc 的调用,它也是 LocalReAlloc 的后续函数(underlying function)。尽管在此之前,ILocalReAlloc 需要检查调用程序是否为某一 LMEM_FIXED 块修改了标志。这是一个双非切换(no-no)。

LMEM_MOVEABLE 的 ILocalReAlloc 代码开始于查看调用程序是否直接想修改标志。如果是这样,代码则修改句柄的 LOCAL_HANDLE_TABLE_ENTRY

(局部柄表入口)中的标志并跳出去。接下来,代码查看它是否被尺寸参量为0的程序所调用。如果如此,调用程序则想要废除此块。ILocalReAlloc以块柄通过IHeapFree来实现编译。尽管在此之前ILocalReAlloc检查了块是否是锁定的,并在适当时候会提出拒绝。

如果块参量为非零,调用程序则去查询一个新块的配置。如果句柄的当前内存块为零,则块已经提前废除。此时,函数直接调用HPAlloc来得到所需尺寸的块。如果内存块已经与该句柄联合,ILocalReAlloc则将内存块地址通过HPAlloc,以使其完成再分配块的繁索工作。

ILocalReAlloc伪码

```
// Parameters:
//      HLOCAL  hMem
//      UINT uBytes;
//      UINT uFlags;
// Locals:
//      DWORD fDiscardable;
//      HANDLE hHeap;
//      HANDLE hNewHandle;
//      LOCAL_HANDLE_TABLE_ENTRY * pHandleEntry;
//      PVOID pBlock;

    uFlags &= 0xFFFFDFFF;       // Turn off 0x00002000 bit, which has no
                                // meaning.

    HouseCleanLogicallyDeadHandles();   // ???

    // Get the default process heap from the process database.
    hHeap = ppCurrentProcessId->lpProcessHeap;

    // Acquire the heap semaphore so that we're not interrupted.
    x_WaitForSemaphore( hHeap->pCriticalSection );

    if ( uFlags & 0xFFFFD02D )          // Test for any flags that aren't
        goto LocalRealloc_invalid_flags // defined, or which shouldn't be
                                        // used (e.g., LMEM_INVALID_HANDLE).

    fDiscardable = uFlags & LMEM_DISCARDABLE;

    if ( (uFlags & LMEM_DISCARDABLE) && !(uFlags & LMEM_MODIFY) )
        goto LocalRealloc_invalid_flags;

    if ( hMem & 2 )     // If an LMEM_MOVEABLE block.
    {
        if ( !x_IsHandleInRange(hHeap, hMem) )
        {
            _DebugOut( "LocalReAlloc: hMem out of range\n",
                       SLE_WARNING + FStopOnRing3MemoryError );
            InternalSetLastError( ERROR_INVALID_HANDLE );
            goto LocalRealloc_error;
        }
```

```
// Point to the HANDLE_TABLE_ENTRY for this handle.
pHandleEntry = hMem - 2;

if ( pHandleEntry->signature != "BS" )
{
    _DebugOut( "LocalReAlloc: invalid hMem, bad signature\n",
                SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_HANDLE );
    goto LocalRealloc_error;
}

pBlock = pHandleEntry->pBlock;      // Get pointer to the data area.

if ( uFlags & LMEM_MODIFY )
{
    pHandleEntry->flags |= fDiscardable ? 2 : 0
    goto done;
}

if ( uBytes == 0 )  // Setting size to 0 is the same as discarding
{                   // the block.
    if ( pHandleEntry->cLock )
    {
        _DebugOut( "LocalReAlloc: discard of locked block\n",
                    SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_INVALID_HANDLE );
        goto LocalRealloc_error
    }

    if ( pBlock == 0 )  // If no data area is associated with this
        goto done;      // handle, there's nothing else to do.

    // There is a data area associated with this handle. Go
    // free it.
    if ( IHeapFree( hHeap, HEAP_NO_SERIALIZE, pBlock - 4 ) )
        goto LocalRealloc_error;

    // Set the pointer to the data area to NULL, because we just
    // released the memory.
    pHandleEntry->pBlock = 0;
    goto done;
}

// If we get here, we're not setting the size to NULL. This
// means that we'll need to HeapAlloc or HeapReAlloc a new block.

uBytes += 4;    // Add space for back-pointer to HANDLE_TABLE_ENTRY.

    if ( pBlock == 0 )  // If there's no data area associated with this
    {                   // handle, we can just HeapAlloc a new area.
        if ( uBytes == 0 )
            goto new_moveable_handle
```

```
            hNewHandle = HPAlloc( hHeap, uBytes, uFlags & HEAP_NO_SERIALIZE );
            if ( !hNewHandle )
                goto LocalRealloc_error

            // Set the first DWORD of the HeapAlloc'ed area to be a pointer
            // to our HANDLE_TABLE_ENTRY struct.
            *(PDWORD)hNewHandle = pHandleEntry;
            goto new_moveable_handle;
        }

        // If we get here, there's already a data area associated with
        // this handle. Therefore, we'll use HeapReAlloc to get the new block.

        if ( pHandleEntry->cLock )
            uFlags |= HEAP_GROWABLE;

        hNewHandle = HPReAlloc( hHeap, hMem, uBytes,
                                uFlags | HEAP_NO_SERIALIZE );
        if ( hNewHandle )
        {
new_moveable_handle:
            // Set the pointer to the data area to be 4 bytes into the
            // block returned by HeapReAlloc/HeapAlloc. (The first DWORD
            // of this block is a pointer to our HANDLE_TABLE_ENTRY struct.)
            pHandleEntry->pBlock = hNewHandle+4;
            goto done;
        }
        else    // Oops! Something is wrong. Return 0.
        {
            hMem = 0;
            goto done;
        }
    }
    else    // An LMEM_FIXED block.
    {
        if ( uFlags & LMEM_MODIFY )
        {
            _DebugOut( "LocalReAlloc: can't use LMEM_MODIFY on fixed block\n",
                       SLE_WARNING + FStopOnRing3MemoryError );
            InternalSetLastError( ERROR_INVALID_PARAMETER );
            goto LocalRealloc_error;
        }
            // There's always memory associated with an LMEM_FIXED handle, so
            // we can just call HeapReAlloc without all the contortions
            // that an LMEM_MOVEABLE block needs to go through.
            hMem = HPReAlloc( hHeap, hMem, uBytes, uFlags & HEAP_NO_SERIALIZE );
            goto done;
        }

    LocalRealloc_invalid_flags:

        _DebugOut( "LocalReAlloc: invalid flags\n",
                   SLE_WARNING + FStopOnRing3MemoryError );
        InternalSetLastError( ERROR_INVALID_PARAMETER );
```

```
LocalRealloc_error:

    hMem = 0;

done:
    InternalLeaveCriticalSection( hHeap->pCriticalSection );
    return hMem;
```

LocalHandle 和 ILocalHandle(局部句柄函数)

LocalHandle 是有效层的一部分。它检验所通过的 hLocal 在 0x10 字节到 7 字节之前以及在此指针之后是有效的。任何句柄——LMEM_FIXED 或 LMEM_MOVEABLE——均应符合此准则。假定测试是成功的,LocalHandle 则跳到了 ILocalHandle。

LocalHandle 函数利用了某个内存块的地址并返回了与此块关联的局部堆柄。这项工作对于一个 LMEM_FIXED 块而言是很简单的,因为块地址与句柄是相同的。但是,ILocalHandle 至少应当足够有效地可以检验地址是否真的是某个 HPAlloc 了的块(已经配置的块)的地址。

其它 ILocalHandle 应当满足的条件是 LMEM_MOVEABLE 句柄。这一点更带有些技巧性。ILocalHandle 的伪码表明,对于 LMEM_MOVEABLE 块,ILocalHandle 将 4 个字节加到了配置尺寸上。在配置的头 4 个字节中,ILocalHandle 塞入了一个指向局部柄表入口的指针。这 4 个字节正是在 ILocalHandle 函数中产生动作的。ILocalHandle 仅仅从所通过的指针中减去 4,然后读入那一点的 DWORD。DWORD 应当是一个指向柄表入口的指针。ILocalHandle 检查此指针是否真正指向了柄表入口。如果是,ILocalHandle 则将柄表入口地址加 2 后返回。正如我们前面已经看到的,此时的柄表入口是一个指向内存块的指针。

LocalHandle 伪码

```
// Parameters:
//      PVOID   pMem

    Set up a structured exception handler frame

    AL = *(PBYTE)(hLocal + 7 );     // If the pointer is bogus, these will
    AL = *(PBYTE)(hLocal - 0x10 );  // fault, and the exception handler
                                    // returns a failure value to the caller.
    Remove structured exception handler frame

    goto ILocalHandle
```

ILocalHandle 伪码

```
// Parameters:
//      PVOID   pMem
// Locals:
//      HANDLE hHeap;
//      HLOCAL hLocal
//      LOCAL_HANDLE_TABLE_ENTRY * pHandleEntry;
//      DWORD pLocalArena;
//      PSTR pszError;

    // Get the default process heap from the process database.
    hHeap = ppCurrentProcessId->lpProcessHeap;

    // Acquire the heap semaphore so that we're not interrupted.
    x_WaitForSemaphore( hHeap->pCriticalSection );

    // Verify that the local handle is even with the range of valid handles.
    if ( !x_IsHandleInRange(hHeap, pMem) )
    {
        pszError = "LocalHandle: pMem out of range\n";
        goto error;
    }

    // If the block is MOVEABLE, then 4 bytes before the block is a
    // pointer to the handle table entry. This pointer is sandwiched
    // between the HPAlloc arena and the block's data.
    pHandleEntry = *(PDWORD)(pMem-4);

    if ( x_IsHandleInRange(hHeap, pHandleEntry) )
    {
        // It's an LMEM_MOVEABLE handle. Verify the signature
        if ( pHandleEntry->signature == "BS" )  // "BS" = 0x5342
        {
            hLocal = pHandleEntry+2;
            goto return_hLocal
        }
        // Hmmm...it's not an LMEM_MOVEABLE handle. Fall through to
        // see if it's LMEM_FIXED.
    }
    else    // An LMEM_FIXED handle.
    {
        pLocalArena = pMem - 0x10;
        if ( (pLocalArena->size & 0xF0000001) == 0xA0000000 )
        {
            hLocal = pMem;
            goto return_hLocal;
        }
    }

    // If we get here, it's not a valid MOVEABLE or FIXED block.
    pszError = "LocalHandle: address not a heap block\n";

error:
    _DebugOut( pszError, SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_HANDLE );
```

```
    hLocal = 0;

return_hLocal:
    InternalLeaveCriticalSection( hHeap->pCriticalSection );
    return hLocal;
```

LocalSize 和 ILocalSize(局部尺寸函数)

LocalSize 是有效层的一部分。它检验所通过的 hLocal 在 0x10 字节到 7 字节之前以及在此指针之后是有效的。任何句柄——LMEM_FIXED 或 LMEM_MOVEABLE——均应符合此准则。假定测试是成功的,LocalSize 则跳到了 ILocalSize。

LocalSize 返回了与所通过的局部句柄相关的内存块的尺寸。确定尺寸的实际工作是由面向代码尾部的 HeapSize(堆尺寸函数)完成的。如果局部句柄是 LMEM_FIXED,LocalSize 则几乎直接地跳到 HeapSize 调用的位置。

如果句柄是 LMEM_MOVEABLE,在调用 HeapSize 之前,LocalSize 则需首先将句柄转换成指向内存块的指针。如果是这样,LocalSize 则首先检查局部句柄参量是否为一有效的局部句柄。如果是,LocalSize 则从 LOCAL_HANDLE_TABLE_ENTRY 结构中抓住指向内存块的指针。

LocalSize 中代码的最后一个位只应用于 LMEM_MOVEABLE 句柄。正如我在 ILocalAlloc 代码所示,LMEM_MOVEABLE 内存块比所需尺寸大 4 个字节。这 4 个字节用于容纳返回柄表入口的指针。为了使 LocalSize 所报告的内容与 LocalAlloc 所配置的内容一致,对于 LMEM_MOVEABLE 块,LocalSize 从 HeapSize 所返回的数值中减去了 4。

LocalSize 伪码

```
// Parameters:
//      HLOCAL  hLocal

    Set up a structured exception handler frame

    AL = *(PBYTE)(hLocal + 7 );     // If the pointer is bogus, these will
    AL = *(PBYTE)(hLocal - 0x10 );  // fault, and the exception handler
                                    // returns a failure value to the caller.
    Remove structured exception handler frame

    goto ILocalSize
```

ILocalSize 伪码

```
// Parameters:
//      HLOCAL  hLocal
// Locals:
//      HANDLE hHeap;
//      DWORD size;
//      PSTR pszError;
//      LOCAL_HANDLE_TABLE_ENTRY * pHandleEntry;

    // Get the default process heap from the process database.
    hHeap = ppCurrentProcessId->lpProcessHeap;
    // Acquire the heap semaphore so that we're not interrupted.
    x_WaitForSemaphore( hHeap->pCriticalSection );

    if ( hLocal & 2 )       // A moveable handle.
    {
        if ( !x_IsHandleInRange(hHeap, hLocal) )
        {
            pszError = "LocalSize: hMem out of range\n";
            goto error;
        }

        // The handle points 2 bytes into the LOCAL_HANDLE_TABLE_ENTRY
        // struct. Subtract 2 bytes to get a pointer to the
        // LOCAL_HANDLE_TABLE_ENTRY.
        pHandleEntry = hLocal - 2;

        if ( pHandleEntry->signature != "BS" )
        {
            pszError = "LocalSize: invalid hMem, bad signature\n";
            goto error;
        }

        hLocal = pHandleEntry->pBlock
        if ( !hLocal )
        {
            size = 0;
            goto return_size;
        }
    }

    size = IHeapSize( hHeap, HEAP_NO_SERIALIZE, hLocal );

    if ( hLocal is a MOVEABLE block )
        size -= 4;
    goto return_size;

error:
    _DebugOut( pszError, SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_HANDLE );
    size = 0;

return_size;
    InternalLeaveCriticalSection( hHeap->pCriticalSection );

    return size;
```

LocalFlags(局部标志函数)

LocalFlags 返回了低级 BYTE 中的局部堆块的锁计数器和次最低级 BYTE 中的块标志。LocalFlags 代码以检查句柄的有效性开始。接下来,代码分成二路。如果代码是 LMEM_FIXED 句柄(最低点结束在 0、4、0x8、或 0xC,)函数则返回 0(标志=LMEM_FIXED 时,计数=0)。但是,LocalFlags 的确检查了句柄是否指向某个 HPAlloc 了的块。如果不是这样,LocalFlags 将返回 LMEM_INVALID_HANDLE(无效句柄)。

LocalFlags 所面临的其它情况是 LMEM_MOVEABLE 句柄。此时,函数从句柄中减去 2,从而生成了一个指向 LOCAL_HANDLE_TABLE_ENTRY 的指针。函数从此结构中减去 clock、flags、和 pBlock 区段。锁计数进入了未被修改的返回值。然而,标志段并不包括 LMEM_XXX 型标志。因此,LocalFlags 需要根据标志和 pBlock 区段的信息,与返回的 LMEM_XXX 标志合并。如果 pBlock 为 0,则表明该块已经废除。(这应当只发生在 LocalReAlloc 以 0 尺寸调用时。)对于带有 LMEM_FIXED 的情况,如果所通过的局部句柄看起来不正确,LocalAlloc 则返回 LMEM_INVALID_HANDLE。

LocalFlags 伪码

```
// Parameters:
//      HLOCAL  hMem
// Locals:
//      HANDLE hHeap;
//      DWORD flags;
//      PSTR pszError;
//      WORD retValue;
//      LOCAL_HANDLE_TABLE_ENTRY * pHandleEntry;
//      HEAP_ARENA * pArena;

    Set up structured exception handler frame

    retValue = LMEM_INVALID_HANDLE;

    // Get the default process heap from the process database.
    hHeap = ppCurrentProcessId->lpProcessHeap;

    // Acquire the heap semaphore so that we're not interrupted.
    x_WaitForSemaphore( hHeap->pCriticalSection );

    if ( !x_IsHandleInRange(hHeap, hMem) )
    {
        pszError = "LocalFlags: hMem out of range\n";
        goto error;
    }
```

```
if ( hMem & 2)     // A moveable block.
{
    // Back up two bytes to point at the LOCAL_HANDLE_TABLE_ENTRY.
    pHandleEntry = hMem - 2;

    // Look for signature at start of handle table entry.
    if ( pHandleEntry->signature != "BS" )
    {
        pszError = "LocalFlags: invalid hMem, bad signature\n";
        goto error;
    }

    retValue = pHandleEntry->cLock;

    if ( pHandleEntry->pBlock == 0 )    // Is address of real data 0?
        HIBYTE(retValue) |= LMEM_DISCARDED;

    // If the discardable (2) bit is set in the handle table entry flags,
    // turn on the LMEM_DISGARDABLE bits in the return value.
    if ( pHandleEntry->flags & 2 )
        HIBYTE(flags) |= LMEM_DISCARDABLE;

    goto return_flags;
}
else    // A fixed block.
{
    // The hMem points to a HPAlloc block, so there should be an HPAlloc
    // style arena 0x10 bytes earlier.
    pArena = hMem - 0x10;

    // Check the arena's size field to make sure it's consistent with
    // an in-use block. If hMem is a bogus pointer, this will
    // fault, but the structured exception handler will catch it.
    if ( (pArena->size & 0xF0000001) == 0xA0000000 )
    {
        retValue = 0;
        goto return_flags
    }

    pszError = "LocalFlags: invalid hMem\n";
        // Fall through to error code.
    }

error:
    _DebugOut( pszError, SLE_WARNING + FStopOnRing3MemoryError );
    InternalSetLastError( ERROR_INVALID_HANDLE );

return_flags:

    InternalLeaveCriticalSection( hHeap->pCriticalSection );

    Remove structured exception handler frame

    return retValue;
```

LocalShrink(局部压缩函数)

在 Win32 中,LocalShrink 对于堆本身无任何影响,因为 Win32 堆块是不能移动的。但是,对于 Win16 的局部压缩函数(LocalShrink)来讲,函数返回了堆的大小。因此,为了考虑到兼容性,Win32 的 LocalShrink 返回了缺省进程堆的大小。LocalShrink 对于 Win32 的应用程序来讲,可能有些用途。由于一些特殊的原因,WIN32 API 在获取缺省进程堆的大小方面好象不具备好的、可资源化的方法。Windows 95 利用 LocalShrink 函数解决了这一问题。

LocalShrink 伪码

```
// Parameters:
//      HLOCAL hMem          // Neither of the two parameters is used.
//      UINT cbNewSize
// Locals:
//      HANDLE hHeap;

    // Get the default process heap from the process database.
    hHeap = ppCurrentProcessId->lpProcessHeap;

    return hHeap->size;      // Size field is first DWORD in heap region.
```

LocalCompact(局部压缩函数)

象 LocalShrink 一样,Win32 的 LocalCompact 之所以存在的原因也是考虑到了与 Win16 的兼容性。由于 Win32 的堆块是不移动的,因而堆也就无需压缩。

LocalCompact 伪码

```
    return 0;        // Easy enough?
```

Win32 全局堆函数(Global Heap Functions)

Windows 95 中的全局堆函数是很少存在的。在绝大多数情况下,它们或者是直接跳到其对应的局部堆中,或者是在 GlobalAlloc(全局配置时),共享相同的入口位置。接受 HGLOBAL 参量的绝大多数函数生成一个记号(token)来试图检查是否通过了一个有效的 HGLOBAL。这项测试并不像局部堆函数所进行的测试那样严格。任何由 GlobalAlloc 或 LocalAlloc 配置的块最终是来自 HPAlloc 的。因此,在该块的 0x10 字节之前和 7 字节之后,应当总存在有效的内存块。

由于全局堆函数相当小，因此其伪码本身就足以说明问题，而无需分别描述每一个函数。

GlobalAlloc

GlobalAlloc 与 LocalAlloc 共享同一入口点。

GlobalLock

GlobalLock 伪码

```
Set up a structured exception handler frame

    AL = *(PBYTE)(hGlobal + 7 );    // If the pointer is bogus, these will
    AL = *(PBYTE)(hGlobal - 0x10 ); // fault, and the exception handler
                                    // returns a failure value to the caller.
    Remove structured exception handler frame

    goto GlobalWire;                // JMPs to ILocalLock.
```

GlobalUnlock

GlobalUnlock 伪码

```
same tests as GlobalLock
    goto GlobalUnwire;              // JMPs to ILocalUnlock.
```

GlobalFree

GlobalFree 伪码

```
same tests as GlobalLock
    goto LocalFree;
```

GlobalReAlloc

GlobalReAlloc 伪码

```
// Parameters:
//      HGLOBAL hGlobal
    same tests as GlobalLock
         goto IGlobalReAlloc;       // JMPs to ILocalReAlloc.
```

GlobalSize

GlobalSize 伪码

```
same tests as GlobalLock
    goto IGlobalSize;      B    // JMPs to ILocalSize.
```

GlobalHandle

GlobalHandle 伪码

```
same tests as GlobalLock
   goto IGlobalHandle;            // JMPs to ILocalHandle.
```

GlobalFlags and IGlobalFlags

GlobalFlags 伪码

```
same tests as GlobalLock
   goto IGlobalFlags;
```

IGlobalFlags 伪码

```
// Parameters:
//      HGLOBAL hMem

    // Pass through to LocalFlags, and then turn off any bits in the
    // high BYTE of the low WORD that aren't valid GMEM_xxx flags.
    return LocalFlags( hMem ) & 0xFFFFF1FF
```

GlobalWire

GlobalWire 伪码

```
goto ILocalLock;
```

GlobalUnWire

GlobalUnWire 伪码

```
goto ILocalUnlock;
```

GlobalFix

GlobalFix 伪码

```
// Parameters:
//      HGLOBAL hMem

    if ( hMem != 0xFFFFFFFF )
        return GlobalLock( hMem );  // GlobalLock ultimately calls ILocalLock.
```

GlobalUnFix

GlobalUnfix 伪码

```
// Parameters:
    //      HGLOBAL hMem

        if ( hMem != 0xFFFFFFFF )
            return GlobalUnlock( hMem );    // GlobalUnlock ultimately
                                            // calls ILocalUnlock.
```

GlobalCompact

```
        goto LocalCompact;
```

杂函数(Miscellaneous Functions)

本章所要介绍的最后几个函数不属于前面所介绍的任何一类,但也同样相当重要。我并未想包括每一个可能的内存函数,而是选择了一些有趣的函数。(本章已相当长,因而不准备加入附加的例程.)

WriteProcessMemory 和 ReadProcessMemory (写进程内存和读进程内存函数)

WriteProcessMemory 和 ReadProcessMemory 函数可用于读和修改另外一个函数的内存。为了利用这些函数,你需要拥有一个另外进程的句柄,而对 Win32 API 来讲,得到这样的句柄并不容易。WriteProcessMemory 和 ReadProcessMemory 是 Win32 调试程序的两个关键函数。调试程序处于需要读和写入另一进程内存(调试者要预先设定好)的小型应用程序中。

在上述条件下，WriteProcessMemory 和 ReadProcessMemory 是类似的。因此，我这里只给出一类伪码，即 WriteProcessMemory 的伪码。唯一明显的差别是，WriteProcessMemory 调用的是 VWIN32 服务 0x002A0017，而 ReadProcessMemory 则调用 0x002A0016 服务。

WriteProcessMemory 以某种合并代码(synchronization code)开始。它要确信其并未容纳 Win16Mutex 或 Krn32Mutex。然后，代码进入“绝对完整”(must-complete)区域，该区域表明是不能关闭的。WriteProcessMemory 随后要确定源地址必需处于应用专用场(application private arena)中，该场正是 VMM 资源所调用的位于 4MB 之上以及 2GB 之下的区域。

WriteProcessMemory 所进行的下一步是获取指向与源地址进程相关联的进程结构的指针。代码利用进程结构来寻找源地址进程的线程列。出于某些原因，复制内存的 VWIN32 服务也想得到目标进程中当前线程的 ring 0 栈地址。一旦 WriteProcessMemory 需要调用 VWIN32，它就必需查询 Krn32Mutex 并调用 VWIN32 的 0x002A0017 服务。在 VWIN32 利用内存文本(memory context)完成其工作之后，WriteProcessMemory 则释放 Krn32Mutex 并通过调用 LeaveMust-Complete(离开绝对完整函数)退出所谓的绝对完整状态。如果在上述步骤中出现错误，WriteProcessMemory 会调用 SetLastError(设置最后错误)来通知调用程序错误信息。

WriteProcessMemory 伪码

```
// Parameters:
//  HANDLE  hProcess;       // Handle of the process whose memory is read.
//  LPCVOID lpBaseAddress;  // Address to start writing to.
//  LPVOID  lpBuffer;       // Address of buffer with data to write.
//  DWORD   cbRead;         // Number of bytes to write.
//  LPDWORD lpNumberOfBytesWritten;     // Actual number of bytes written.
// Locals:
//  DWORD   pProcess;
//  DWORD   ptdb;
//  DWORD   lastError;

    // Make sure we don't already have the Krn32Mutex or Win16Mutex.
    x_CheckNotSysLevel_Krn32_Win16_mutex();

    // Function that emits function names and parameters to the KERNEL
    // debugger if a KERNEL32 global variable is TRUE (off by default).
    x_LogKernelFunction( number indicating the WriteProcessMemory function );

    EnterMustComplete();

    if ( lpNumberOfBytesWritten )
        *lpNumberOfBytesWritten = 0;
```

```
    if ( lpBuffer < 0x00400000 )
        goto set_invalidParam_lasterror_with_bp

    if ( lpBuffer < 0xC0000000 )
        goto set_invalidParam_lasterror_with_bp

    pProcess = x_GetObject( hProcess, 0x80000010, 0 );

    if ( !pProcess )
    {
        lastError = 1;
        goto emit_trace_info;
    }

    if ( some flag set in a certain pProcess field )
    {
        lastError = ERROR_PROCESS_ABORTED;
        goto set_last_error;
    }

    myLocal1 = x_SomeListFunction(pProcess->threadList, 0);

    if ( myLocal1 )
    {
        do
        {
            ptdb = *(PDWORD)(myLocal1+8);
            if ( ptdb->ring0_hThread )
                break;
        } while ( myLocal1 = x_SomeListFunction( pProcess->threadList, 1) )
    }
    else
        ptdb = some unitialized local variable?

    if ( !myLocal1 )
    {
        InternalSetLastError( ERROR_PROCESS_ABORTED );
        goto done;
    }

    _EnterSysLevel( Krn32Mutex );

    // Call the Win32 VxD service in VWIN32.VXD to copy the memory.
    lastError = VxDCall( 0x002A0017, ptdb->ring0_hThread, lpBaseAddress,
                         lpBuffer, cbRead, lpNumberOfBytesWritten );
    if ( !lastError )
        InternalSetLastError( lastError );

    _LeaveSysLevel( Krn32Mutex );

done:
    x_UnuseObjectSafeWrapper( pProcess );
    goto emit_trace_info;

set_invalidParam_lasterror_with_bp:
    INT 3
```

```
    InternalSetLastError( ERROR_INVALID_PARAMETER );

emit_trace_info:

    x_SomeLoggingFunction( "WriteProcessMemory ptdb %08x Src %08x (%02x) "
                           "Dst %08x cb %d erc %d\n",
                           ptdb, lpBuffer, *(PWORD)lpBuffer,
                           lpBaseAddress, cbRead, lpNumberOfBytesWritten );

    LeaveMustComplete();

    return !lastError
```

GlobalMemoryStatus 和 IGlobalMemoryStatus（全局内存状态函数）

GlobalMemoryStatus 函数是很便于查看设备内存状态的。函数填写了具有诸如多少个物理 RAM 页正在使用，交换文件的尺寸之类信息的某个 MEMORYSTATUS(内存状态)结构。从许多方面讲，该函数是 Windows 3.1 MemManInfo(内存信息)例程的 Win32 等效函数。

真正的 GlobalMemoryStatus 代码是一个参数有效层存根(parameter validation layer stub)。它唯一的测试是确定通过该函数的指针指向的内存应足可以容纳一个 MEMORYSTATUS 结构。尽管资源中曾强调过，在调用 GlobalMemoryStatus 之前，你不必初始化 MEMORYSTATUS 结构的 dw4Length 区段。

GlobalMemoryStatus 伪码

```
// Parameters:
//  LPMEMORYSTATUS lpmstMemStat

    Set up structured exception handler frame

    // Make sure that the beginning and end of the MEMORYSTATUS
    // structure is accessible.
    *(PBYTE)lpmstMemStat += 0;
    *(PBYTE)(lpmstMemStat+0x1F) += 0;

    Remove structured exception handler frame

    goto IGlobalMemoryStatus;
```

IGlobalMemoryStatus 所作的工作只不过是利用 DemandInfoStruc 结构中的压缩信息(abreviated version of the information)来填写某个 MEMORYSTATUS 结构。该结构是通过调用 VMM.VXD 中的_GetDemandPageInfo VxDs 函数被填写的。考虑到可能不具备 DDK 资源的情况，这里给出了 DemandInfoStruc 的形式。

DemandInfoStruc 结构

```
DWORD   DILin_Total_Count   ; Pages in linear address space.
DWORD   DIPhys_Count        ; Specifies the total number of physical pages
                            ; managed by the memory manager.
DWORD   DIFree_Count        ; Specifies the number of pages currently in the
                            ; free pool.
DWORD   DIUnlock_Count      ; Specifies the number of pages that are currently
                            ; unlocked. Free pages are always unlocked.
DWORD   DILinear_Base_Addr  ; Always zero.
DWORD   DILin_Total_Free    ; Total number of free virtual pages in the
                            ; current memory context. This value includes only
                            ; pages in the private arena.
DWORD   DIPage_Faults       ; Total page faults.
DWORD   DIPage_Ins          ; Calls to pagers to page in.
DWORD   DIPage_Outs         ; Calls to pagers to page out.
DWORD   DIPage_Discards     ; Calls to pagers to discard.
DWORD   DIInstance_Faults   ; Instance page faults.
DWORD   DIPagingFileMax     ; Current maximum size of the swap file, in pages.
                            ; Zero if swapping is turned off.
DWORD   DIPagingFileInUse   ; Number of swap file pages currently in use. This
                            ; is the number of pages by which physical memory
                            ; is overcommitted. Zero if swapping is disabled
                            ; or if physical memory is available for all
                            ; swappable pages.
DWORD   DICommit_Count      ; Total committed pages.
DWORD   DIReserved[2]       ; Reserved; do not use.
DemandInfoStruc ends
```

无疑，将存在许多写成的程序，它们处于屏幕的角部并告知用户什么是“内存安装”(memory load)。那么到底什么才是真正的“内存安装”呢？在伪码中，你可以看到被提交的页面数与由 Windows 95 内存管理器所管理的物理页面数相除的结果为 50 次。例如，一个具有 8MB RAM 和 11MB 提交页面的系统应当具有一个 68 的内存安装(最大为 100)：

$$(11 \times 50)/8 == 68.75$$

是的，你可以拥有比实际 RAM 更多的被提交页面。提交某个页面并不意味着 RAM 将总是与其关联。除非你将此内存页面锁定，否则 Windows 95 对于此页面的使用是自由的。

IGlobalMemoryStatus 伪码

```
// Parameters:
//  LPMEMORYSTATUS lpmstMemStat
// Locals:
//  DemandInfoStruc dis;
//  DWORD   memLoad;

    Set up structured exception handler frame
```

```
// Call the VMM Win32 VxD service to fill the struct
VxDCall( _GetDemandPageInfo, &dis, 0 );

memLoad = (dis.DICommit_Count * 50) / dis.DIPhys_Count
if ( memLoad < 100 )
    lpmstMemStat->dwMemoryLoad = memLoad;
else
    lpmstMemStat->dwMemoryLoad = 100;

lpmstMemStat->dwTotalPhys = dis.DIPhys_Count * 4096;
lpmstMemStat->dwAvailPhys = dis.DIFree_Count * 4096;

lpmstMemStat->dwTotalPageFile = dis.DIPagingFileMax * 4096;

lpmstMemStat->dwAvailPageFile = 4096 *
    (dis.DIPagingFileMax - dis.DIPagingFileInUse)

lpmstMemStat->dwTotalVirtual = 0x7FC00000;  // Size of app private data
                                            // area (2GB - 4MB).

lpmstMemStat->dwAvailVirtual = dis.DILin_Total_Free * 4096;

lpmstMemStat->dwLength = sizeof( MEMORYSTATUS )

Remove structured exception handler frame
```

GetThreadSelectorEntry 和 IGetThreadSelectorEntry（获取线程选择器入口函数）

在我考察 GetThreadSelectorEntry 时，使我感到意外的是，它也包括在 Win32 的 API 中。GetThreadSelectorEntry 其实与线程是毫无关联的。实际上，hThread 参数的有效性是被检查了的，但从未被使用过。GetThreadSelectorEntry 只允许你对系统的 VM 的局部描述符号表进行只读访问。该表正好包含了 Win32 应用程序的浮点代码和数据段，它也是 Win16 应用程序获得其代码和数据段的地方，同样也包括 GlobalAlloc 了的句柄。该函数在任何系统的工具包中均是一个有价值的工具！

假定你已将一个有效的选择器通入了 GetThreadSelectorEntry，你则可得到一个 8 字节的结构，且该结构与某个 LDT 描述符是相同的。每个描述符中的信息是基本地址和长度。因为 Win32 应用程序具有一个可以到达任何地方的水平指针（flat pointer），它们可以利用该函数将一个 16:16 地址转换为一个水平的 32 位地址，以便 Win32 应用程序可向其读和写。你甚至可以构造你自己的 Win16 GetSelectorBase（获取选择器基地址）和 GetSelectorLimit（获取选择器界限）函数的 Win32 版本。

对于 GetSelectorLimit 函数而言，在《Unauthorized Windows 95》一书的 449 页，有一个代码用于获取选择器的基本地址。该代码使用了 VWIN32 VxD 服务来激活 DPMI 子函数 6。该 DPMI 子函数返回了特定选择器的基本地址。由于此函数

技术性强，GetThreadSelectorEntry 应该同时运行并使代码简化。好在 GetThreadSelectorEntry 是一个资源性函数，在所有可能的情况下，它应当比非资源化的函数更优先使用。

GetThreadSelectorEntry 代码的主要价值是 LDTAlias 和 LDTPtr 变量。它们都是 KERNEL32.DLL 中的全局变量，LDTPtr 包含了系统 VM 的 LDT 的线性地址。LDTAlias 是一个可对选择器表的内存进行读和写访问的选择器值。这一点与 KRNL386 所使用的用于在全局堆函数内部破坏选择器表的 LDT 化名选择器是一样的。（见《Windows Internals》一书的第二章）。

GetThreadSelectorEntry 程序

```
// Parameters:
//      HANDLE  hThread;
//      DWORD  dwSelector;
//      LPLDT_ENTRY  lpSelectorEntry;

    Set up structured exception handling frame

    Touch the first and last bytes that lpSelectorEntry points to.
    If a fault occurs, it's considered a bad pointer, and the exception
    handler returns FALSE;

    Remove structured exception handling frame

    goto IGetThreadSelectorEntry;

IGetThreadSelectorEntry proc
// Parameters:
//      HANDLE  hThread;
//      DWORD  dwSelector;
//      LPLDT_ENTRY  lpSelectorEntry;
// Locals:
//      PTHREAD_DATABASE ptdb;
//      BOOL    retValue;
//      LPLDT_ENTRY pLDTAliasDesc;
//      LPLDT_ENTRY pDesiredDesc;


    retValue = TRUE;

    x_CheckNotSysLevel_Win16_Krn32_mutexes();

    x_LogSomeKernelFunction( function number for GetThreadSelectorEntry );
     _EnterSysLevel( Win16Mutex );

     _EnterSysLevel( Krn32Mutex );

     ptdb = x_ConvertHandleToK32Object( hThread, 0x20, 0 );
     if ( ptdb )              // The hThread is okay.
     {
```

```
        if ( dwSelector & 0x4 ) // Check if it's a GDT selector. Bail if so.
        {
            InternalSetLastError( ERROR_INVALID_PARAMETER );
            goto error;
        }

        pDesiredDesc = dwSelector & 0x0000FFF8;     // Get offset in LDT.

        // Get a ptr to LDT alias selector's descriptor in the LDT.
        pLDTDesc = LDTPtr + (LDTAlias & 0x0000FFF8);

        // Check if the selector asked for is outside the upper limit
        // of in-use selectors in the LDT.
        if ( pDesiredDesc > pLDTDesc->limit )
        {
            InternalSetLastError( ERROR_INVALID_PARAMETER );
            goto error;
        }

        pDesiredDesc += LDTPtr;     // Make it point into the LDT now.

        // Copy the LDT descriptor into lpSelectorEntry.
        memcpy( lpSelectorEntry, pDesiredDesc, sizeof(LDT_ENTRY) );
    }
    else
    {
error:
        retValue = FALSE;
    }

    SomeOutputFunction( "GetThreadSelectorEntry sel %04x erc %d\n",
                        dwSelector, (retValue ? 0 : GetLastError()) );

    _LeaveSysLevel( Krn32Mutex );

    _LeaveSysLevel( Win16Mutex );

    return retValue;
```

C/C++编译器的 malloc 和 new 函数

在许多情况下，C/C++程序员忽略了所有的操作系统内存管理函数并利用C运行库实现其内存管理，尤其是 malloc 和 free 函数。但如果使用了 C++，情况会怎样呢？在我所知道的所有 PC 编译器中，新的运算符是直接映射到 malloc 的，并且 delete(删除函数)则映射到 free 函数。问题是，这些函数是如何根据下面的 OS 功能来运行的？

本章，我已经介绍了堆函数(如 HeapAlloc，和 HeapFree 函数)在功能方面是如何与 malloc 和 free 函数相接近的。这是否表明 C 运行库中的 malloc 和 free 函数就是 HeapAlloc，和 HeapFree 函数的外壳呢？直到可视 C++4.0(Visual C++4.0)之

前，答案一直是不，只有一个例外：Microsoft 的 C 运行库的 CRTDLL. DLL 版本。在 CRTDLL. DLL 中，malloc 和 new 函数直接调用了 HeapAlloc，而 free 和 delete 则调用了 HeapFree。CRTDLL. DLL 被许多标准的 Windows NT 和 Windows 95 EXE 以及 DLL 所使用。这是很好的办法，因为它避免了 Microsoft 为每个 EXE 和 DLL 留有一个 C 运行库的备份。

但不幸的是，C 编译器的出售商并没有很好地合作来使每个人使用安装于他们自己的操作系统上的 CRTDLL. DLL。因此，我们仍需为每个运行程序生成 C 运行库的备份或将 C 运行库装载于各自的程序。因为这种状况不可能很快改变，所以知道在这些运行库内部所发生的一切是很有必要的。

这里，我不准备像介绍操作系统函数那样详细地介绍 C 运行库的 malloc 和 new 函数。反之，我准备提供总体思路，以供你选择来满足自己的内存管理代码。

据我所知，Borland 和 Microsoft 以类似的方式提供了他们的运行库堆(run-time library heaps)。事实上，除了堆的尺寸之外，状况与 Windows 3. x 时代的变化并不太大。每个运行文件或 DLL 拥有其各自的堆。带有三个 DLL 的程序将以具有 4 个独立堆作为结束(一个用于 EXE，其余用于 DLL)。在给定 DLL 中生成的配置将来自于其 DLL 的堆。这一点与 Win32 HeapAlloc 函数相反——无论它从哪里来——总是从应用程序的堆中分配内存(假定你总是进入了一个缺省进程的堆柄)。

由 C 编译器的 RTLs 所提供的堆使用了其各自的数据结构和内存管理代码，而不是使用高级的操作系统函数如 HeapAlloc。这样，在同一程序中，就很难混合并匹配 HeapAlloc 了的和 malloc 了的内存(Nu-Mega 的程序员同行在这方面就深有体会)。

通过深入考察 C/C++代码，我们可以看到 malloc 是如何映射后续的 OS 函数的。我已经仔细研究了各级的 Borland C++4. 5 RTL，你无需在这方面再作工作。下面给出的调用栈说明了 malloc 是如何在 Windows 95 的顶部实施的。

```
malloc (HEAP.C)
    _getmem (GETMEM.C)
        _virt_reserve (VIRTMEM.C)
            VirtualAlloc( NULL, size, MEM_RESERVE, PAGE_NOACCESS )
```

C 运行库利用 VirtualAlloc 从 OS 分配了大块的内存，这实质上正是 HeapAlloc 所要作的工作。配置区的页面最初是保留的(下放了的)，而且在使用之前，必需以_virt_commit 形式提交。(_virt_commit 就是 VirtualCommit 的一个外壳。)这种提交页面的方法听起来是否有些熟悉？应当是。这种方法与 Windows 95 将其内存提交给其堆的方法是一样的。如果你尚感新鲜，可以反过头来读一下 hpCarve 和 hpCommit 一节的内容。

运行库不准备为每个 malloc 的调用去调用 VirtualAlloc。他们需要建立并保持内部数据结构以跟踪配置了什么样的块，同时为快速配置保留了自由列表。那么堆块的形式如何呢？HEAP. C 可以说明：

```
/*----------------------------------------------------------------------
 * Knuth's "boundary tag" algorithm is used to manage the heap.
 * Each block in the heap has tag words before and after it, which
 * contain the size of the block:
 *  SIZE
 *  block ...
 *  SIZE
 * The size is stored as a long word, and includes the 8 bytes of
 * overhead that the boundary tags consume.  Blocks are allocated
 * on LONG word boundaries, so the size is always even.  When the
 * block is allocated, bit 0 of the size is set to 1.  When a block is
 * freed, it is merged with adjacent free blocks, and bit 0 of the
 * size is set to 0.
 *
 * When a block is on the free list, the first two LONG words of the block
 * contain double links.  These links are not used when the block is
 * allocated, but space needs to be reserved for them.  Thus, the minimum
 * block size (not counting the tags) is 8 bytes.
```

Windows 95 的零售版本以类似的方式管理着堆块(但并不完全相同)。但是,一个 HeapAlloc 的总量只是 4 字节(大小),而 C++运行库的每 一个块就使用了 8 字节。同时也要注意,Borland C++和 HeapAlloc 函数在使用内存方面的共性是将一个自由块指向了另一个自由块。

使用由运行库提供的堆时,要注意到它们的使用寿命。当一个 DLL 从内存卸下并接收到 DLL_PROCESS_DETACH 注记时,运行库将调用 VirtualFree 来释放堆的内存。如果其它 DLL 拥有指向该内存块的指针,则该指针将立即失效。如果另外的 DLL 在后来也卸下,并在其 DLL_PROCESS_DETACH 处理中使用了这些指针中的其中之一,将会出现很难调试的程序故障。

因此,要回答我前面提出的问题,malloc 实际上是由 Windwos 95 HeapAlloc 函数所实现的一种编译器,但至少有二点不同。首先,每个 EXE 和 DLL 拥有其各自的由运行库提供的堆,而所有的 HeapAlloc 配置是来自于由操作系统建立的缺省进程堆的。其次,运行库堆的寿命比缺省进程堆的寿命要短。对于一定的顺序关联的操作而言,可能出现诸如使用运行库堆之类的问题。这并不是说你应当避免 new 和 malloc 函数,只不过是提醒你注意他们是什么而且存在哪些潜在不利因素。

小结

本章(尽管是本书的最长一章)只是仅仅触及了 Windows 95 内存管理方方面面的一些皮毛而已。我们已经考察了 CPU 的内存分页,每个进程的独立地址空间,以及 Windows 95 在所有进程之间共享的内存区域。对于 Win32 API 级,我们看到 VirtualXXX 函数是如何在页面级上管理页面的,同时也看到 HeapXXX 函数是如何以相当高的粒度水平来提供内存管理的。来自 Win16 API(即,GlobalXXX 和 LocalXXX 函数)的所谓的过时的堆函数只不过是 HeapXXX 函数之上的一个薄层而已。下一章,我们将会看到 ring 3 KERNEL32.DLL 是如何与 ring 0 虚拟设备管理器通迅来获得作为堆函数建造基础的基本建筑块服务的。

第　六　章

Windows 95 的三个核心部件

* *

一天，几个同事和我聚在办公室里，讨论 Windows95 的核心结构。和通常一样，话题转到不同的 Windows95 部件如何得到其它部件的内部情况。(这通常被认为是一件坏事，一件应尽可能避开的事。)就在这天，一个同事提出疑问，"为什么 Microsoft 非要使用分离的 16 位和 32 位内核呢，还有一个类似内核的 VxD，为什么不把它们放在一个文件中使用呢?"

在这一章，我们就来讨论这个问题。本章题目所说的部件是指下面这三个部件：VWIN32. VXD，KERNEL32. DLL 和 KRNL386. EXE。注意：本章包含了一些相当高级的资料，但对于理解本章内容和继续学习本书其它内容，它们不是必需的。

Windows 应用程序员可能立即会意识到 KRNL386. EXE 是 16 位核心部件，KERNEL32. DLL 是它的 32 位等价物。这些动态连接库(DLL)负责提供任何 Win16 或 Win32 应用所需的核心函数集(例如，LoadLibrary，_lread 等)。这些 DLL 中的(大多数)函数在同 SDK 或编译器一起提供的标准系统头文件中说明。对于 16 位程序，文件 WINDOWS. H 提供 KRNL386. EXE 中函数的原型说明。对于 Win32 程序，则由 WINBASE. H 和 WINCON. H 给出 KERNEL32. DLL 中大多数函数的说明。

遗憾的是，上面提到的第三个部件(VWIN32. VXD)在微软的文档或头文件中很少提及。就我所知，VWIN32. VXD 仅有的文档是 Windows95 DDK 中的文件 VWIN32. H。我们很需要 VWIN32. H 这样的文档，特别是当你知道 VWIN32. VXD 是高层两个最重要的虚拟设备驱动程序(另一个是 Virtual Machine Manager，也可以称作 VMM)之一时。VWIN32. VXD 提供了操作系统最基本层次上的关键功能——由 16 位的 KRNL386 和 32 位的 KERNEL32 都使用的功能。在为这本书所作的研究中，我多次发现，任何考证 KRNL386 和 KERNEL32 的努力都会很快把你领进 VWIN32. VXD 的王国。

看到微软公司关于 VWIN32. VXD 的文档如此缺乏，我试图在这一章补充这方面的资料。首先，概述 VWIN32. VXD 和它的界面，然后说明这三个部件是怎样相互联系和通信的。请看下面的说明：

- KRNL386. EXE 调用 VWIN32. VXD
- KRNL386. EXE 调用 KERNEL32. DLL

- KERNEL32. DLL 调用 KRNL386. EXE
- KERNEL32. DLL 调用 VWIN32. VXD
- VWIN32. VXD 和 KERNEL32. DLL 交换信息
- VWIN32. VXD 和 KRNL386. EXE 交换信息

在这些情况中，人们特别感兴趣的是，在什么地方，KERNEL32 调用了 KRNL386. EXE。微软声称不存在这样的调用，但《Unauthorized Windows95》却证实微软的声明是错误的。在这一章，我提供一个内容丰富的列表，说明 KERNEL32 的哪些函数调用了 KRNL386. EXE。

《Unauthorized Windows 95》涉及的另一个问题是 Win32 VxD 服务。这些服务提供了一个简便的方法，使 Win32 程序员以标准的 C 风格的调用规则来调用 VxD。Win32 VxD 是 Windows 95 结构的主要部分。例如，所有的文件 I/O 最终都转换成了对 Win32 VxD 的调用。（这看起来似乎有些陌生，特定的 Win32 VxD 最终调用 VMM. VXD 的中断 Exec_PM_Int，中断号为 21h。（听起来很熟悉吗？DOS 还不会消亡，是吗？）

遗憾的是，微软公司没有正式公开 Win32 VxD。因为《Unauthorized Windows 95》也仅提供了为数不多的几页说明，所以，在这一章，我将更深入的讨论它们。通常情况下，获悉和探究未公开接口的最好办法是写一些这样的工具程序。因此，这一章包含了一个监视对 Win32 VxD 的调用的侦探程序（W32SVSPY）。我必须越过不少的障碍——其中一些可能要被微软公司抛弃——来使 W32SVSPY 正常工作。在本章末，我将解释 W32SVSPY 怎样完成它神奇的功能。其中涉及的一些技术可能会在你的系统级编程中显示出其方便的优点。

VxD 剖析

因为在后面紧接着要展开讲述 VWIN32. VXD，所以现在即使顺便熟悉一下 VxD 的基本知识也会有所帮助。为了那些没有 VxD 经验的读者，需要顺序地对 VxD 作一快速回顾。如果你曾经写过 VxD 并且对它们还有印象，可以跳过这一节。

顾名思义，VxD 就是虚拟设备驱动程序，也就是说，当多个程序使用特定的硬件设备时，它可以为每个程序虚拟一个设备。不过，没有任何材料说明 VxD 必须和硬件设备相联结。实际上，VxD 只是运行在处理器 ring 0 上的 DLL。由于 VxD 运行在最高优先级上，所以没有它不能完成的事情。然而为此付出的代价是，VxD 非常的难写，并且不能象常规的 ring 3 DLL 那样容易地调用。

我不想说明如何写 VxD，也不想涉及对 VxD 作者来说很有用的各种各样的精采的技巧和技术。有一些书，如《Unauthorized Windows 95》和 Dave Thielen 的《Writing Windows Device Drivers》，更深地讲述了这些内容。我只对 VxD 作足够的解释，使我能顺利地讲解 VWIN32. VXD。

被装入内存时，VxD 有一个唯一的 16 位的设备号（设备 ID，是一个唯一的识别码）。VxD 中设备 ID 为 1 的是 VMM. VXD，而虚拟键盘设备（VKD）使用的设备

ID 为 0Dh。VWIN32. VXD 的 ID(本章要着重讲的)是 2Ah。通过在 Windows 95 的 DDK 中的头文件 VMM. INC 或 VMM. H 中查找形式为 xxx_DEVICE_ID 的宏定义,可以得到关于标准的预定义 VxD 及其 ID 的相当完整的列表。要注意,低于 512 的 VxD ID 被微软公司保留。其它编写 VxD 的公司应从微软申请 VxD ID。

从其它 VxD 中调用 VxD 函数

就象 ring 3 系统 DLL 以一个标准方式供 EXE 和 DLL 使用其函数一样,VxD 中的函数也可以被其它的 VxD 按某种方式调用。当生成 VxD 时,所有可以从外部调用的函数都列在一个数组里。每个这样的函数称作一个服务(service)。当一个 VxD 调用另一个 VxD 中的功能时,并不使用服务的名字,而是使用函数在数组中的索引号。下面从文件 VMM. INC 中摘录一段说明,作为例子:

```
Begin_Service_Table VMM, VMM
VMM_Service  Get_VMM_Version, LOCAL
VMM_Service  Get_Cur_VM_Handle
VMM_Service  Test_Cur_VM_Handle
VMM_Service  Get_Sys_VM_Handle
```

一个 VxD 对函数 Get_VMM_Version 的调用将调用 VMM 的 0 号服务。对函数 Get_Cur_VM_Handle 的调用实际上调用 VMM 的 1 号服务。函数 Test_Cur_VM_Handle 就是 VMM 的 2 号服务,依此类推。

从一个 VxD 调用另一个 VxD 中服务的实现机制是非常有趣的。同 ring 3 系统 DLL 不一样,VxD 装入程序并不把包含服务函数地址的 CALL 指令加入到发出调用的 VxD 的代码中。实际上,建立 VxD 时,根本没有加入任何 CALL 指令!在应有 CALL 指令的地方,对 VxD 服务函数的调用是下面的格式:

```
INT 20h
DD  device_and_service_number   ;; A different value for each VxD service
```

INT 20h 后面的双字(DWORD)不是一个数值,它的高字包含了设备号(前面讲过),低字包含了设备中的服务号。回到前面的例子,对 Test_Cur_VM_Handle (即 VMM 的 2 号服务)的调用将象下面这样:

```
    INT 20h
    DD  00010002h   ;; 0001=VMM device ID, 0002=service # for
Test_Cur_VM_Handle
```

请看下一个例子,GetSystemTime 服务是 VWIN32. VXD 中的第三个服务。因此,当建立 VxD 时,对这个服务函数的调用表现为如下形式:

```
INT 20h
DD  002A0002h   ;; 002A=VWIN32 device ID, 0002=service # for
                ;; GetSystemTime (service numbers start at 0)
```

当 ring 0 服务 INT 20h 被调用时，它检查跟随在中断指令后面的 DWORD，并根据设备号和服务号搜索目标的地址。显然，它的速度较慢，不过不用担心。在代码中给出的 INT 20h 指令被调用一次以后，INT 20h 处理程序就把实际的 CALL 指令补充到代码中。它工作得很好，因为 INT 20H 指令加上一个 DWORD 是 6 字节，刚好是一个 32 位近程间接调用所需的大小（也就是，call DWORD PTR [xxxxxxxx]）。从一个现象可以观察到这一点，就是 VxD 装入程序在装入一个 VxD 时，不会把其中所有引入函数（imported functions）转换成 CALL 指令，而是在确实用到一段代码时，才把它补全。

当服务号低字的最高位被置 1 时（0x8000），就会产生一个类似的，也是通过 INT 20h 动态补全代码的系统。这种情况下，代码被用跳转（JMP）指令补全，而不是用 CALL 指令。例如，下面的程序将跳转到函数 Test_Cur_VM_Handle，而不是调用它：

```
INT 20h
DD  00018002h
```

从 Win16（保护模式）代码调用 VxD 函数

如果 VxD 的函数只能被另外的 VxD 调用，Windows 就会令人生厌。因为 VxD 可以在任何地方，做任何事情，可以被通常的 ring 3 应用代码调用是它应有的基本特性。使 ring 3 代码可以调用 ring 0 代码的能力，使得应用可以完成那些它们自己不能完成的任务。这段时间有个时髦的思想，就是当有些功能不能用通常的应用代码完成时，就写一个 VxD，然后从应用中调用它。

有些人（包括我自己）认为这种策略应只在迫不得已时使用。任何人写出的 VxD 都有可能破坏操作系统（不管是有意还是无意）。我个人认为，应尽量避免写 VxD。系统中，功能强大而又毫无限制的代码越少越好。我很害怕有那么一天，由于缺乏经验的程序员认为编写 VxD 是完成某些任务的唯一方法，而使那些虚无的 VxD 把我的硬件设备搞得乱七八糟。

我的意见姑且不说，从 DOS 或 Win16 程序调用 VxD 多少有些别扭，不过倒也不是很困难。一个 VxD 可以提供一组函数，它们有的可以在虚拟 8086 模式（V86 模式，即实模式）下调用，有的可以在 ring 3 保护模式下调用，有的在两种情况下均可调用。VxD 中的入口点对 V86 模式和 ring 3 保护模式的程序来说是不同的，如果你愿意，也可以为它们设置相同的入口地址。

在 V86 或 16 位保护模式下调用 VxD 时，应用程序首先要申请一个可以执行长调用（far CALL）的地址空间。这个地址空间可以通过 AX 寄存器值为 1684h 的 INT 2Fh 中断调用获得。为了识别申请的是哪一个 VxD 的入口点，还要把 BX 设置成我前面提到的 16 位 VxD 号。从该 INT 指令返回时，寄存器 ES:DI 中包含一个 16:16 位的远指针，调用它就可以把控制传送给在 ring 0 运行的 VxD。

让我们通过从 KRNL386 中摘录的一段代码，看看 KRNL386 是如何得到 VWIN32. VXD 的入口点的(并且查看 VWIN32. VXD 的版本号)。

```
XOR     DI,DI                   ; Zero out ES:DI in case the operation fails.
MOV     ES,DI
MOV     AX,1684                 ; INT 2Fh, AX = 1684h -> Get Device Entry Point
MOV     BX,002A                 ; 002Ah = Device ID for VWIN32.VXD
INT     2F
MOV     AX,ES                   ; ES:DI should now contain the entry point.
OR      AX,AX                   ; Is the segment part of the return address 0?
JE      failure                 ; Yes? Go to the failure case code.

MOV     AH,00                   ; VWIN32 service 0 = VWIN32_Get_Version

PUSH    DS                              ; Save away the current DS on the stack.
MOV     DS,WORD PTR CS:[0002]           ; Load DS with KRNL386's DGROUP selector.

MOV     WORD PTR [lpfnVWIN32],DI        ; Save away the entry point (in ES:DI).
MOV     WORD PTR [lpfnVWIN32+2],ES
CALL    FAR [lpfnVWIN32]                ; Call the entry point with AH = 0.
```

也许有对 Intel 体系 CPU 的保护模式比较熟悉的读者要怀疑这一切是怎样完成的。ring 3 代码是不能调用 ring 0 代码的，CPU 中有专门的保护机制来避免这种情况的发生。(全面地讨论 ring 等级和保护超出了本章的范围。)除非使用特殊的组织方式，ring 3 代码试图调用(也就是装入)ring 0 代码会产生一般保护错。INTEL 体系支持一种很少用到的称为调用门(call gates)的机制，它允许 ring 3 代码以一种严格控制的方式调用 ring 0 代码。没有比这更精采的工作了。

前面给出的代码段中，AX＝1684h 的 INT 2Fh 中断调用返回一个地址，如果你用象 SoftIce/W 或 WDEB386 这样的系统调试器反汇编该地址处的内容，将看到下面这样的结果：

```
:u 3B:03d0
003B:000003D0   INT     30  ; #0028:C025DB52  VWIN32(04)+0742
003B:000003D2   INT     30  ; #0028:C0002BC9  VMM(01)+1BC9
003B:000003D4   INT     30  ; #0028:C022F713  VMM(0D)+0713
```

嗯……奇怪！INT 2Fh 调用返回的入口点指向一条 INT 30h 指令。发生什么事情了？原来 Windows 使用 INT 30h 强制 CPU 从 ring 3 转到 ring 0。任何中断或异常都将导致 CPU 把控制传给合适的 ring 0 处理程序，这些处理程序的地址存储在中断描述表(Interrupt Descriptor Table，IDT)中。Windows 95 的 INT 30h 处理程序通过中断时 CS:IP 的内容搜索要接手控制的 ring 0 代码的地址。列表中分号(;)后面的内容就是处理每个特定的 INT 30h 的程序的地址。(SoftIce/W 知道怎样找到和解释 INT 30h 处理程序使用的调度表(dispatch table)，所以它能够给出处理程序的地址。)关于 VWIN32. VXD 的 INT 30h 处理程序的地址位于 VWIN32. VXD 本身中，这并不奇怪。如果更进一步，反汇编分号后面的地址(这是赋给 VWIN32 的入口点 INT 30h 的 ring 0 代码的地址)处的内容，将得到下面的结果：

```
:u 28:c025db52

0028:C025DB52   MOVZX   EAX,BYTE PTR [EBP+1D]   ; Get AH value at INT 30h.

0028:C025DB56   CMP     EAX,+15                 ; There are 16 VWIN32 PM
0028:C025DB59   JA      C025DB62                ; services. Is it within
                                                ; range? If not, go to
                                                ; the error-reporting code.

0028:C025DB5B   JMP     [C03229A4+4*EAX]        ; Call through the service
                                                ; JMP table to the appropriate
                                                ; service entry point.

0028:C025DB62   PUSH    C03229FC    ; string ptr -> "VWIN32_PMAPI_Proc: "
                                    ; "invalid function number\r\n"

0028:C025DB67   INT     20 VXDCall _Debug_Out_Service   ; Emit error diagnostic.
```

第一条指令需要作一些解释。当 VxD V86/PM API 例程被调用时，应用程序不是把参数压入堆栈中。通常的原因是 ring 0 VxD 代码和 ring 3 应用程序使用的是不同的堆栈。(当 CPU 在保护等级之间切换时，同样切换了堆栈寄存器，使它指向新等级代码使用的指定堆栈。)

因为 ring 3 代码传给 VxD 函数的参数不能压入堆栈，所以约定在调用 INT 30h 前，把传给 VxD 函数的参数放入寄存器中。当 INT 30h 处理程序调用 ring 0 处理程序时，传给它一个结构指针，该结构中含有 INT 30h 被调用时的寄存器值。这个指针是一个展开的 32 位指针，保存在 EBP 寄存器中。EBP 指针指向的结构称作客户寄存器结构(Client Register Structure，见 VMM. INC 中的 Client_Reg_Struc)。能被 V86 或 16 位 ring 3 保护模式程序调用的 API 的 VxD 能够通过 EBP 中的客户寄存器结构指针读写 ring 3 寄存器的值。

第一条指令(你刚才看到的那段代码中)把执行 INT 30h 时 AH 寄存器的值装入 EAX。因为从 V86 或 ring 3 16 位保护模式程序调用 VxD 时有个约定：把函数识别号放在寄存器中。如果函数的识别号在合法范围里，处理代码就用跳转表把控制传递给 VWIN32. VXD 中相应的函数。如果函数识别号非法，就打印一条错误信息。

Win32 代码调用 VxD 函数

我刚才讲述的 VxD 的两个接口在 Windows 3. 0 中就有了。在 Windows 95 中，并没有对它们作根本的改变。不过，Windows 95 的确加入了另一个调用 VxD 的接口。因为 Windows 95 不仅支持 DOS 和 Win16 应用程序，也支持 Win32 应用程序，所以微软公司提供 Win32 代码调用 VxD 的方法就不足为奇了。这使得所有 VxD 接口的数目增加到四个(ring 0 VxD 服务，V86 模式程序的接口，ring 3 16 位保护模式代码的接口，和下面我要讲的新接口)。

因为这个新的 VxD 接口只能被 ring 3 Win32 代码使用，所以该接口称作 Win32 VxD 服务。不要把 Win32 VxD 服务这个概念和通常的 VxD 服务(指 VxD 函数可以被其它 VxD 调用)相混淆。也不要把 Win32 VxD 服务和你在 Windows NT 中看到的 Win32 服务(Win32 Service)相混淆。Windows NT 的服务更象忙碌的处理程序，与 Win32 VxD 服务毫无关系。

唉，由于一些不可理解的原因(至少我是这样)，微软公司决定隐藏 Win32 VxD 服务接口。这也许是劝阻人们，不要写和 Windows NT 不兼容的代码，因为 Windows NT 不支持 VxD。作为替代，微软公司希望你使用设备输入输出控制(DeviceIoControl)Win32 API，它在 Windows NT 和 Windows 95 之间是半兼容的。麻烦在于，使用设备输入输出控制接口更慢。实际上，在 Windows 95 中，设备输入输出控制最终也是调用了 Win32 VxD 服务。

因为 Windows 95 比 Windows 的早期版本更多的使用 C 语言来书写程序，所以 Win32 VxD 服务接口理所当然地是 C 调用方式。也就是说，Win32 VxD 服务函数可以很容易地被用 C 写的 ring 3 代码调用。Win32 VxD 服务通过堆栈传递参数，类似于通常的函数调用。与其它 VxD 相比，这是一个显著的进步。因为其它 VxD 由于需要用寄存器传递参数，因此一般用汇编语言调用。

在 SoftIce/W 中观察 VxD 接口

SoftIce/W for Windows 95 能使用这一节讲过的所有各种各样的 VxD 接口，包括新的 Win32 VxD 服务。你可以用 VxD 命令后跟指定的 VxD 名字，看到其接口。例如，命令“VXD REBOOT”产生下面的输出：

```
:VXD REBOOT
VxD Name   Address    Lenth  Seg  ID   DDB      Control   PM  V86  VxD  Win32
REBOOT     C00910CC 0002F0 0001 0009 C0091334 C00910CC  Y   N    4    2
REBOOT     C0201F94 0002E9 0002
REBOOT     C037E9A0 00010C 0003
REBOOT     C02269D4 0000EE 0004
REBOOT     C0233B44 00009C 0005
REBOOT     C02373BC 00004B 0006
Total Memory:     3K
Init Order=24000000  Reference Data=0  Version 4.00
PM API=C02269D4 (3B:3EC)  V86 API=0 (0:0)
4 VxD Services
  0000  C00912D5
  0001  C00912DA
  0002  C00912E2
  0003  C009123B
```

```
2 Win32 Services ———
  0000   C0226A04   Parms=02
  0001   C0226A19   Parms=02
```

前两行给我们提供了丰富的信息。我们知道了(通过"PM"下面的"Y")REBOOT 设备为 ring 3 16 位保护模式程序提供了调用接口。还知道了(通过"V86"下面的"N")设备 REBOOT 没有为 V86 模式的代码提供调用接口。继续下去看到,设备 REBOOT 有四个常规 VxD 服务(可以被其它 VxD 调用),还有两个 Win32 VxD 服务。

现在看列表的结尾,注意最后三行,它们给出了 REBOOT 设备提供的 Win32 VxD 服务信息。有两个 Win32 VxD 服务,其细节给出在最后两行。这两行分别包含了一个服务入口地址,就象服务期望的双字(DWORD)参数。从这些信息(或通过研究 VMM.INC 的相关内容),可以推出,每个 Win32 服务有一个和它联系的 8 字节(两个双字)结构:

```
DWORD   pfnService ;      // The address of the service function.
DWORD   cParams ;         // The number of DWORD parameters.
```

我把这个结构称作服务表入口(service table entry)。当一个 VxD 激活时,它必须向系统注册它的 Win32 VxD 服务。这通过调用 VMM.VXD 中的_Register_Win32_Service 函数完成。传给_Register_Win32_Service 函数的参数中,有一个是指向 Win32 VxD 服务表的指针。这个指针存放在 VxD 的设备描述块(DDB,Device Descriptor Block)中,SoftIce/W 就是从这个描述块中得到了前面的显示结果。

调用 Win32 VxD 服务和调用任何其它可用的 VxD 接口都不一样。它不是申请中断,也不是调用函数指针,而是从调用一个未公开的函数开始,该函数在 KERNEL32 中,名为 VxDCall。调用函数 VxDCall 之前,调用代码把传给 Win32 VxD 服务的所有参数都压入堆栈。在调用 VxDCall 之前,最后压入堆栈的数值是一个双字,类似于常规 VxD 服务的格式。也就是,高字包含要使用的 VxD 的识别号,低字包含一个从 0 开始的函数索引号。这个函数索引号是 Win32 VxD 服务表的索引,而不是常规 ring 0 VxD 服务表的索引。

举个例子你就明白了。下边的代码调用 VWIN32.VXD 中的 VWIN32_sleep 函数。VWIN32_sleep 是 VWIN32.VXD 提供的第 10 个 Win32 VxD 服务,所以其函数索引号(识别号)为 9(Win32 VxD 的服务函数索引号从 0 开始)。

```
PUSH    DWORD PTR [EBP+08]  // Push a parameter.
PUSH    002A0009            // 002A = VWIN32, 0009 = VWIN32_sleep
CALL    VxDCall
```

VxDCall 函数是一个标准调用方式函数(stdcall function)(也就是说,参数是从右向左压入堆栈,并且由被调用者清栈)。前面的代码如果用 C 语言编写,就是下面这样:

```
VxDCall( 0x002A0009, parameter );
```

如果列出所有从 KERNEL32.DLL 中提取的内容,你会发现前 8 个入口点(输出顺序号为 1 到 8)都是同一个地址。这个地址处就是 VxDCall 函数。为什么对同一个函数使用了 8 个分离的入口点呢? 简单地说就是:从内部看,这些入口点称作 VxDCall@0,VxDCall@4,VxDCall@8,直到 VxDCall@28。MS C 编译器把标准调用格式的函数名拆开重组,成为包含一个@字符,后跟函数所用参数的字节数的形式。因为不同的 Win32 VxD 服务带有不同个数的参数,对 VxDCall 函数的调用,最终有的被编译器翻译成 VxDCall@4,有的被翻译成 VxDCall@16,用略有区别的的名字提供多个入口点,那么无论一个对 VxDCall 函数的调用需要多少参数,连接器都可以确定它应是对哪一个 VxDCall 函数的调用。在本章中,把所有这些入口点统称为 VxDCall 函数(VxDCall funtion)(如果你读过《Unauthorized Windows 95》一书,那么要注意在那本书中,VxDCall 函数指的是 VxDCall0)。

下边把这部分内容作一小结。调用 Win32 VxD 服务的 Win32 代码首先把参数压入堆栈,然后把双字的服务识别号(Service ID)压入栈。这个双字标记出被调用的 VxD 和 VxD 中的 Win32 VxD 服务函数。最后,代码调用 KERNEL32.DLL 中的 VxDCall 函数。当 Win32 VxD 服务返回时,栈中所有参数已被清除,程序在紧随在调用 VxDCall 的指令后面的指令恢复执行。

好了,这是从外部看,是怎样调用 Win32 VxD 服务的。下边我们进入 Win32 VxD 服务实际实现的细节。我们从分析 VxDCall 函数的代码开始:

```
VxDCall:

MOV     EAX,DWORD PTR [ESP+04]  ; Get service code (e.g., 0x002A0010) into EAX.

POP     DWORD PTR [ESP]         ; Move the return address up on stack so
                                ; that the call below returns directly to
                                ; the caller.

CALL    FWORD PTR CS:[BFFC9004] ; 16:32 CALL to INT 30 instruction that
                                ; transfers control to ring 0.
```

前两条指令的作用就是把双字的 VxD 服务号从堆栈弹出,并把它放入 EAX。对 VxDCall 函数的 32 位近调用语句在发出调用时,把返回的 EIP 值也压入栈中,当双字的 VxD 服务号被弹出后,用于返回的 EIP 值就上升到 Win32 VxD 服务号占据过的位置。第三条指令是对 INT 30h 的 32 位长调用指令。我们前面已见过 INT 30h,那时它们是 V86 模式和 16 位保护模式程序调用 VxD 的方法。然而,这里的 INT 30h 不是通常的 INT 30h 指令。

```
u 3B:000003DE:   // The 16:32 pointer found at BFFC9004

003B:000003DE   INT     30  ; #0028:C02301E4  VMM(0D)+11E4
```

这个 INT 30h 指令是 VxDCall 函数用来把控制传递给 ring 0 代码的，它使控制跳转到 VMM.VXD 内的某个地方。让我们看一看在 VMM.VXD 中相应地址处发现的一些伪码：

```
//-------------------------------------
// Entry point for all Win32 VxD Services (in VMM.VXD).
//-------------------------------------
// Parameters:
//      Client_Reg_Struct   * pClientRegs
// Locals:
//      PVOID   pRing3StackFrame // ESP at time of INT 30 call that got us here.
//      DWORD   service_DWORD;
//      WORD    vxd_id;         // HIWORD of the service_DWORD.
//      WORD    service_index;  // LOWORD of the service_DWORD.
//      DWORD   cParams;        // # of parameters for this service.
//      PROC    pfnService;     // The address of the service entry point.

   DS = pClientRegs->Client_SS;

   pRing3StackFrame = pClientRegs->Client_ESP;

   // pRing3StackFrame now points to following on the ring 3 stack:
   //
   //  Args pushed for VxDCall()       <- pRing3StackFrame + C
   //  Return Address for VxDCall()    <- pRing3StackFrame + 8
   //  CALL FWORD PTR CS value         <- pRing3StackFrame + 4
   //  CALL FWORD PTR EIP value        <- pRing3StackFrame + 0

   access_rights = LAR pClientRegs->Client_SS;

   if ( !(access_rights & BIG_BIT) )   // If "big" bit not set, use just
   {                                   // the low WORD of pRing3StackFrame.
        pRing3StackFrame = LOWORD(pRing3StackFrame);
    }
// Fill in the client registers with the CS:EIP that ring 3 execution
// should resume at. The CS value on the ring 3 stack comes from the
// CALL FWORD PTR [xxxxxxxx] to the INT 30h. The EIP is the return
// address from the call to VxDCall0. (Yes, this is goofy.)

pClientRegs->Client_EIP = pRing3StackFrame->EIP;
pClientRegs->Client_CS - pRing3StackFrame->CS;

// Advance pRing3StackFrame to the location in the ring 3 stack where
// the VxDCall parameters are located.
pRing3StackFrame += 0xC;

// Get the service DWORD param to VxDCall (e.g., 0x002A0014).
service_DWORD = pClientRegs->Client_EAX;
```

```
vxd_id = service_DWORD >> 0x10; // Which VxD is it? (Look in the high word.)

if ( vxd_id < 0x40 )    // 0x40 is the last of the "standard" VxDs.
{
    // Does this particular VxD even have a Win32 VxD service table?
    if ( ppServiceTable[ vxd_id ] == 0 )
        goto error;

    // If we get here, this VxD supports Win32 VxD services. Is the
    // service index within the range of services provided?
    service_index = LOWORD( service_DWORD );

    if ( ppServiceTable[ vxd_id ].cServices <= service_index )
        goto error;

    service_index++;    // Bias the index up by 1, since the first entry
                        // in a service table holds the # of services.

    // Index into the Win32 service table and grab out the number of
    // DWORD params for this service, as well as the entry point address
    // of this service.

    cParams = ppServiceTable[ vxd_id ].cParams;
    pfnService = ppServiceTable[ vxd_id ].pfnService;

    // Now we start some stack contortions. The parameters pushed on the
    // ring 3 stack prior to the VxDCall now need to be copied to the
    // ring 0 stack.

    POP     EAX     // Remove return address from stack and save it
                    // away in EAX. (This is an address in VMM.)

    ESP -= cParam * 4;      // Make space on the stack for the arguments.
    EDI = ESP;              // Point destination register to the space
                            // we made on the stack for the arguments.

    PUSH    EBX     // Push current VM Handle.
    PUSH    EBP     // Push pointer to client regs struct.
    PUSh    EAX     // Push return address (saved away earlier).

    // If this service takes 1 or more parameters, copy them to the
    // ring 0 stack location we just made.

    if ( cParams )
        REP MOVSD   // ECX = cParams, ESI = pRing3StackFrame, EDI=

    // At this point, the stack looks like this:
    //
    // Args copied by REP MOVSD             <- ESP+0Ch
    // Current VM handle                    <- ESP+08h
    // Client reg struct pointer            <- ESP+04h
    // Return address from this PM API call <- ESP+00h
```

```
    DS = SS     // Ain't the flat model great?

    // Set the ring 3 ESP upon return to point just past the parameters
    // pushed on the stack by the call to VxDCall().

    pClientRegs->Client_ESP = pRing3StackFrame + (cParams * 4)

    goto    pfnService  // Jump to the service entry point.
}
```

VMM.VXD 中的 VxDCall 处理程序很复杂。但是，如果作足够的分析，这些代码可以分解为若干个特定的任务：

1. 从 EBP 指向的客户寄存器结构中读入重要的寄存器值。这些值包括 ring 3 EAX（里面含有服务号）和 ring 3 ESP（它指向压入参数的 ring 3 堆栈）。
2. 修改客户寄存器结构中的 CS 和 EIP 寄存器值，使得当 ring 0 代码返回时，控制从调用 KERNEL32 VxD 的指令后面继续。同样地，修改 ring 0 ESP 寄存器的值，从而弹出调用发生前压栈的参数。
3. 得到 Win32 VxD 服务号的双字并把它的各部分分离（16 位 VxD 号和 VxD 中的 16 位服务号）。检查 VxD 设备号，看它是不是标准的系统 VxD，指定的 VxD 中是否提供 Win32 VxD 服务。如果以上通过，就继续检查，确认 16 位服务号在 VxD 提供的函数号范围内。
4. 把压到 ring 3 堆栈中的参数拷贝到 ring 0 堆栈顶部。
5. 查找指定的 Win32 VxD 服务的入口点并跳转过去。因为该函数有可能需要访问当前的 VM 句柄或客户寄存器结构，所以跳转前，要先把这些值压入堆栈。

当 Win32 VxD 服务函数结束并返回时，控制传递回 VMM.VXD，处理下面的工作：把 CPU 置回 ring 3 状态，并把客户寄存器结构中的值分别装入各寄存器。

从哪儿可以找到 Win32 VxD 服务？

前面提到过，微软公司没有正式公布 Win32 VxD 服务，所以 DDK 没有给出提供 Win32 VxD 服务的 VxD 的现成列表。根据用 SoftIce/W VXD 命令观察的结果，我确认下面的 VxD 提供 Win32 VxD 服务（可能还有其它的）：

VxD	设备号	服务数	服务的功能
VMM	0001h	41	虚拟机器管理
REBOOT	0009h	2	重启动设备
VNETBIOS	0014h	2	虚拟 NETBIOS 设备
VWIN32	002Ah	79	虚拟 Win32 设备
VCOMM	002Bh	27	虚拟 COMM 设备
VCOND	0038h	53	虚拟控制台设备

就象我在本书其它地方说过的，KERNEL32. DLL 很频繁地使用 VWIN32、VMM、VCOND 和 VCOMM。在一些情况下，可外部引用的 KERNEL32. DLL 只不过是 Win32 VxD 服务的外壳。对于 ADVAPI32. DLL 更是如此。由 VMM. VXD 提供的 Win32 VxD 服务中包括注册函数，它与 Win32 API 的注册函数相似。ADVAPI32. DLL 的注册函数是一个调用 VMM Win32 VxD 服务的很简单的外壳。

VMM 提供的 Win32 VxD 服务

本章主要讨论 Win32 VxD 服务和 VWIN32. VXD。然而，对 VMM. VXD 提供的 Win32 VxD 服务的识别号，我至少要把它们列出来，否则，未免有些失职。

00010000h	PageReserve	00010014h	RegDeleteKey
00010001h	PageCommit	00010015h	RegSetValue
00010002h	PageDecommit	00010016h	RegDeleteValue
00010003h	PagerRegister	00010017h	RegQueryValue
00010004h	PagerQuery	00010018h	RegEnumKey
00010005h	HeapAllocate	00010019h	RegEnumValue
00010006h	ContextCreate	0001001Ah	RegQueryValueEx
00010007h	ContextDestroy	0001001Bh	RegSetValueEx
00010008h	PageAttach	0001001Ch	RegFlushKey
00010009h	PageFlush	0001001Eh	GetDemandPageInfo
0001000Ah	PageFree	0001001Fh	BlockOnID
0001000Bh	ContextSwitch	00010020h	SignalID
0001000Ch	HeapReAllocate	00010021h	RegLoadKey
0001000Dh	PageModifyPermissions	00010022h	RegUnloadKey
0001000Eh	PageQuery	00010023h	RegSaveKey
0001000Fh	GetCurrentContext	00010024h	RegRemapPreDefKey
00010010h	HeapFree	00010025h	PageChangePager
00010011h	RegOpenKey	00010026h	RegQueryMultipleValues
00010012h	RegCreateKey	00010027h	RegReplaceKey
00010013h	RegCloseKey		

我没有给出每个函数参数的信息，原因很简单：我自己也不知道。要完成为本书所作的探索，只要知道 VMM Win32 VxD 服务的名字就足够了。不必疑惑，随着时间的推移，有关这些服务的参数将会弄清。通过分析与服务通信的 Win32 函数的文档，可以推导出有关注册服务的参数。

在许多其它情况下，Win32 VxD 服务总是和在 VMM. VXD 的文档中说明的某个特定的 ring 0 VxD 服务通信。例如，上面列出的名为_PagerQuery 的 Win32 VxD 服务总是和在 VMM. HLP 中描述的_PagerQuery 服务通信。请读者自己把

这些问题弄清。

对 VMM.VXD 提供的 Win32 VxD 服务可以作下面的分类：

分类	Win32 VxD 服务
基于页的内存管理	_GetDemandPageInfo,_PageAttach,_PageCommit, _PageDecommit, _PageFlush,_PageFree,_ PageModifyPermissions,_PageQuery, _PageReserve
虚拟内存页支持	_PageChangePager,_PagerRegister,_PagerQuery
ring 0 堆管理	_HeapAllocate,_HeapFree, _HeapReAllocate
内存文本管理	_ContextCreate,_ ContextDestroy, _ContextSwitch,_ GetCurrentContext
注册函数	_RegCloseKey,_RegCreateKey,_ RegDeleteKey,_RegDeleteValue, _RegEnumKey,_ RegEnumValue, RegFlushKey,_RegLoadKey, _RegOpenKey,_RegQueryMultipleValues,_RegQueryValue, _RegQueryValueEx, _RegRemapPreDefKey,_RegReplaceKey, _RegSaveKey,_RegSetValue,_RegSetValueEx,_RegUnloadKey
同步控制	_BlockOnID, _SignalID

在第三章和第五章已经看到，KERNEL32 毫无疑问使用了基于页的内存管理和内存文本管理的服务。KERNEL32.DLL 的其它未讲述的领域使用了其它种类的服务。但注册函数是一个例外。它们由 ADVAPI32.DLL 使用(通过 KERNEL-32.DLL 中的 VxDCall 函数)。

应用程序调用 Win32 VxD 服务

据我所知，微软公司仅在 Windows 95 系统 DLL 的代码中调用了 Win32 VxD 服务。然而，没有任何理由说明一般应用程序不能调用 Win32 VxD 服务，为证明这一点，我写了图 6-1 所示的 WIN95MEM 程序(随书带的磁盘上有完整的源程序)。

```
//=============================================================
// WIN95MEM - Matt Pietrek 1995
// FILE: WIN95MEM.C
// Demonstrates calling a Wir32 VxD service from application code
//=============================================================
#irclude <windows.h>
#irclude "win95mem.h"
#include "d:\chicddk\inc32\vmm.h"

DWORD WINAPI VxDCall2( DWORD service_number, DWORD, DWORD );
```

```
void Handle_WM_TIMER(HWND hWndDlg, WPARAM wParam, LPARAM lParam)
{
    struct DemandInfoStruc dis;
    char szBuffer[256];

    // Demonstrate calling a Win32 VxD service (in this case, the
    // _GetDemandPageInfo service).
    VxDCall2( 0x0001001E, (DWORD)&dis, 0 );

    wsprintf(szBuffer, "Comm: %uK", dis.DICommit_Count * 4);
    SetDlgItemText( hWndDlg, IDC_TEXT_commited, szBuffer );

    wsprintf(szBuffer, "Phys: %uK", dis.DIPhys_Count * 4);
    SetDlgItemText( hWndDlg, IDC_TEXT_physical, szBuffer );

    wsprintf(szBuffer, "%u%%",
              (dis.DICommit_Count * 100) / dis.DIPhys_Count);
    SetDlgItemText( hWndDlg, IDC_TEXT_percentage, szBuffer );
}

BOOL CALLBACK Win95MemDlgProc(HWND hWndDlg, UINT msg,
                              WPARAM wParam, LPARAM lParam)
{
    switch ( msg )
    {
        case WM_INITDIALOG:
            SetTimer( hWndDlg, 0, 1000, 0 ); return TRUE;
        case WM_TIMER:
            Handle_WM_TIMER(hWndDlg, wParam, lParam); return TRUE;
        case WM_CLOSE:
            KillTimer(hWndDlg, 0);
            EndDialog(hWndDlg, 0);
            return FALSE;
    }

    return FALSE;
}
```

WIN95MEM 使用了 VMM.VXD 提供的 Win32 VxD 服务_GetDemandPageInfo。这个服务也不过是一个同名的 ring 0 VxD 服务的外壳。象我在第五章说明的，名为 GlobalMemoryStatus 的 Win32 函数调用了服务_GetDemandPageInfo，并简单地将_GetDemandPageInfo 给出的数据选择一部分返回。既然我们能够直接使用与 GlobalMemoryStatus 所用相同的资源得到全部系统信息，为什么还要通过它去获取被滤过的系统信息呢？

_GetDemandPageInfo 服务的参数是一个指向 DemandInfoStruc 结构的指针。该服务将把结果填写在结构的字段中。各字段如下：

DWORD	DILin_Total_Count
DWORD	DIPhys_Count
DWORD	DIFree_Count

```
DWORD    DIUnlock_Count
DWORD    DILinear_Base_Addr
DWORD    DILin_Total_Free
DWORD    DIPage_Faults
DWORD    DIPage_Ins
DWORD    DIPage_Outs
DWORD    DIPage_Discards
DWORD    DIInstance_Faults
DWORD    DIPagingFileMax
DWORD    DIPagingFileInUse
DWORD    DICommit_Count
DWORD    DIReserved
```

我不打算把这里列出的所有字段都讲清。如果你感兴趣，可以看 Windows 95 DDK 的 VMM 文档中对_GetDemandPageInfo 的说明。就 WIN95MEM 程序来说，我们对两个字段感兴趣：DICommit_Count 和 DIPhys_Count。DICommit_Count 是 VMM 内存管理器已分配(交付)的内存页面总数。注意，已交付的页面不需要映射到物理的 RAM。这更象为将来使用而保留页面。DIPhys_Count 字段中存放虚拟内存管理器(virtual memory manager)控制下的 RAM 的页面数。这是 Windows 95 保护模式部分启动后，所有可用内存的数量。它不计入在 DOS 装入 Windows 95 时，通过 DPMI 为驻留程序(TSR)和设备驱动程序分配的内存。

因为 Windows 95 支持虚拟内存，所以已交付的内存数量常常超过了 Windows 95 虚拟内存管理器控制下的物理内存数量。WIN95MEM 程序在一个对话框中同时显示出已交付的内存和物理内存的数量(以 kb 为单位)。这两个数据以及它们的百分比率每一秒钟更新一次。图 6-2 显示出 WIN95MEM 程序运行时的情况。(当然，这个程序的界面并不吸引人，但用完就可以毫不吝惜的把它扔掉，对吗?)

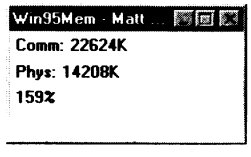


图 6-2　运行 WIN95MEM 程序，演示如何在应用程序中调用 VxDCall

对 VxDCall 函数的调用是 WIN95MEM 代码的重要部分。由于 VxDCall 是未公开的函数，所以 WIN95MEM.C 文件中给出了名为 VxDCall2(这个 2 表示有两个参数)的原型。因为当加入 Win32 VxD 服务号双字(0x0001001E)时，实际上成为三个参数，所以编译器生成一个对名为 VxDCall2@12 的外部函数的调用(末尾有@12 是由于这个函数原型说明为一个标准调用[WINAPI]函数，参数长度为 12

字节)。我建立的 K32LIB.LIB 文件是为调用 KERNEL32 函数从而引出一个名为 VxDCall2@12 的函数,使连接器能顺利工作。附录 A 描述了 K32LIB.LIB 文件的内容。

实际对 VxDCall 函数的调用是下面这样:

```
VxDCall2( 0x0001001E, (DWORD)&dis, 0 );
```

第一个参数是双字服务号(有 VMM 设备号和_GetDemandPageInfo 函数的识别号组成)。第二个参数是一个指针,指向堆栈中一个 DemandInfoStruc 结构类型的局部变量。最后一个参数的含义还不知道。因为 GlobalMemoryStatus 调用该服务时传递了一个 0,所以我也传递了一个 0。就是这些。

分析 VWIN32.VXD

现在,我们已经讲过了 VxD 的接口,下面集中讲述 VWIN32.VXD。在 Windows 95 中,这是一个新 VxD(即在 Windows 3.1 及以前的版本中没有)。VWIN32.VXD 的 16 位 VxD 号是 0x002A。VWIN32.VXD 和 VMM.VXD 所承担的任务有何区别还不清楚(至少在微软公司之外是这样)。然而,可以对 VWIN32.VXD 作如下概括:它包含了影响进程和线程的调度与同步的 Win32 VxD 服务。我认为 VMM.VXD 和 VWIN32.VXD 是一起完成 Windows 95 启动和运行需要的 ring 0 工作的 VxD 服务。

VWIN32.VXD 不提供 V86 模式 API。然而,它提供 ring 0 VxD 服务、16 位保护模式服务和 Win32 VxD 服务。在 Windows 95 所有的 VxD 中,VWIN32 中的 Win32 VxD 服务数目最多。如果不是 VWIN32.VXD 提供了这么多 Win32 VxD 服务,或这些服务对 KERNEL32 的运行不是至关重要,我就不会费这力气写前面几节关于 Win32 VxD 服务的内容。好了,既然如此,现在就让我们进入 VWIN32.VXD 的实质内容。

VWIN32.VXD 的 ring 0 VxD 服务 API

要讲的 VWIN32.VXD 的第一个接口是它提供给其它 VxD 的服务。幸运的是,微软公司在 Windows 95 DDK 的 VWIN32.H 和 VWIN32.INC 中保留了这些服务的一个列表。为方便那些不常用计算机的读者,我在图 6-3 中给出该列表。好在 SoftIce/W VxD 命令能访问这些服务,所以很容易地得到了这些服务的名字及其函数号的列表。

```
:vxd vwin32

VxD Name  Address   Length  Seg   ID    DDB       Control   PM  V86  VxD  Win32
VWIN32    C0075654  0026FC  0001  002A  C0076DE0  C0075654  Y   N     29     79

... some output omitted for brevity

29 VxD Services
  0000  C00756BF  VWIN32_Get_Version
  0001  C0075776  VWIN32_DIOCCompletionRoutine
  0002  C007678F  _VWIN32_QueueUserApc
  0003  C0261002  _VWIN32_Get_Thread_Context
  0004  C0261293  _VWIN32_Set_Thread_Context
  0005  C025E4F8  _VWIN32_CopyMem
  0006  C025E568  _VWIN32_Npx_Exception
  0007  C0262D16  _VWIN32_Emulate_Npx
  0008  C0076AA8  _VWIN32_CheckDelayedNpxTrap
  0009  C0260971  VWIN32_EnterCrstR0
  000A  C0260A9C  VWIN32_LeaveCrstR0
  000B  C025EC78  _VWIN32_FaultPopup
  000C  C007662B  VWIN32_GetContextHandle
  000D  C007575F  VWIN32_GetCurrentProcessHandle
  000E  C02620B6  _VWIN32_SetWin32Event
  000F  C02620F7  _VWIN32_PulseWin32Event
  0010  C026213F  _VWIN32_ResetWin32Event
  0011  C0262229  _VWIN32_WaitSingleObject
  0012  C0262175  _VWIN32_WaitMultipleObjects
  0013  C0260D2C  _VWIN32_CreateRing0THread
  0014  C0076A78  _VWIN32_CloseVxDHandle
  0015  C0078D51  VWIN32_ActiveTimeBiasSet
  0016  C00756D4  VWIN32_GetCurrentDirectory
  0017  C0075802  VWIN32_BlueScreenPopup
  0018  C025E972  VWIN32_TerminateApp
  0019  C007683F  _VWIN32_QueueKernelAPC
  001A  C025EB9D  VWIN32_SysErrorBox
  001B  C00757C0  _VWIN32_IsClientWin32
  001C  C007572E  VWIN32_IFSRIPWhenLev2Taken
```

图 6-3　VWIN32 的 ring 0 VxD 服务

输出的内容中，第一列是在 VWIN32.VXD 中的服务号，中间一列是服务函数的地址。右边一列是服务的名字。这同在 VWIN32.INC 中给出的一样。

ring 0 VWIN32 服务的列表是个大杂烩。然而，线程同步类的函数却有鲜明的标志。函数名使我们对函数有个一知半解，并提示我们，VWIN32.VXD 中有关于各进程（VWIN32_ GetCurrentProcessHandle，VWIN32_ GetCurrentDirectory 和 VWIN32_TerminateApp 服务）的详尽的信息。这很有意思，因为 VMM.VXD 居然只是能访问这些线程，并不提供关于进程管理的功能。它把进程的管理留给了 VWIN32.VXD。

VWIN32. VXD 的 16 位保护模式 API

微软公司没有公布 Windows 95 DDK 中 VWIN32. VXD 的 16 位保护模式 API 函数。有关这些函数的列表最早出现在 1993 年 8 月的《Microsoft System Journal》上我的文章："Stepping up to 32 Bits:Chicago's Process, Thread, and Memory Management"。写这篇文章时,那些函数还包含在 VWIN32. INC 中。现在把这些函数列出在下面:

VWIN32_GET_VER	AH = 0
VWIN32_THREAD_SWITCH	AH = 1
VWIN32_DPMI_FAULT	AH = 2
VWIN32_MMGR_FUNCTIONS	AH = 3
子函数:	
VWIN32_MMGR_RESERVE	AH = 3, AL = 0
VWIN32_MMGR_COMMIT	AH = 3, AL = 1
VWIN32_MMGR_DECOMMIT	AH = 3, AL = 2
VWIN32_MMGR_PAGEFREE	AH = 3, AL = 3
VWIN32_EVENT_CREATE	AH = 4
VWIN32_EVENT_DESTROY	AH = 5
VWIN32_EVENT_WAIT	AH = 6
VWIN32_EVENT_SET	AH = 7
VWIN32_PDB_INFO	AH = 8
VWIN32_THREAD_BOOST_PRI	AH = 9
VWIN32_WAIT_CRST	AH = 10
VWIN32_WAKE_CRST	AH = 11
VWIN32_SET_FAULT_INFO	AH = 12
VWIN32_EXIT_TIME	AH = 13
VWIN32_BOOST_THREAD_GROUP	AH = 14
VWIN32_BOOST_THREAD_STATIC	AH = 15
VWIN32_WAKE_IDLE_SYS	AH = 16
VWIN32_MAKE_IDLE_SYS	AH = 17
VWIN32_DELIVER_PENDING_KERNEL_APCS	AH = 18

那么,什么可以调用这些函数呢? 只有 KRNL386。这些函数就是 Windows 95 的 ring 3 16 位程序和 ring 0 VWIN32 部件的接口。这个 API 中的大多数函数可以归到下面的类中:线程调度,线程同步,内存管理,错误处理。

在前面的列表中,特别需要注意的一个函数是 VWIN32_MAKE_IDLE_

SYS。当没有任务可以调度时，KRNL386.EXE 中的 ring 3 Win16 调度程序就会申请调用这个函数。（如果你有《Windows Internals》一书，可参看其中的 Reshedule 函数）当 16 位 KRNL386 调度程序进入自身空循环时，KRNL386 调用 VWIN32_MAKE_IDLE_SYS 函数。除非在 16 位应用中有活动发生，控制才会返回 KRNL386。

顺便说一下，在前面表中的 VWIN32_EXIT_TIME 函数属于错误处理类。如果看过《Undocumented Windows》，你可能会记得一个叫做 Bunny_351 的函数。在 Windows 3.1 和 Windows 95 中，当 Windows 关闭时，就会调用 Bunny_351。它只有一个目的，就是改变不缺省处理的异常的处理程序地址。在 Windows 95 中，Bunny_351 只不过是 VWIN32.VXD 中 VWIN32_EXIT_TIME 函数调用的一个外壳。

VWIN32.VXD 的 Win32 VxD 服务 API

在前面 VWIN32.VXD 的 API 中，微软公司公布了所有可访问的名字和识别号。但他们没有公布 Win32 VxD 服务的有关信息。我花了一段时间，力图构成一个有关这类接口的已知服务入口点的列表。首先我要声明，下面的列表并不完整，而且有些函数名只是猜测，这些都基于对 KERNEL32 和 VWIN32 代码的观察。

服务识别号	名字（有些是猜测的）
0x002A0000	GetVersion
0x002A0001	Stuff VWIN32 code pointers into caller-supplied buffer
0x002A0002	GetSystemTime
0x002A0003	Stuff code pointers from KERNEL32 into VWIN32's Data area
0x002A0004	Block on some semaphore
0x002A0005	Calls Signal_Semaphore_No_Switch on some semaphore
0x002A0006	Calls VMM Create_Semaphore, and stuffs into global var
0x002A0007	Calls VMM Destroy_Semaphore on semaphore created by 0x002A0006
0x002A0008	VWIN32_CreateThread(including allocating TDBX)
0x002A0009	VWIN32_sleep
0x002A000A	WakeThread
0x002A000B	TerminateThread
0x002A000C	Some sort of initialization function
0x002A000D	_VWIN32_QueueUserApc
0x002A000E	VWIN32_Initialize
0x002A000F	_VWIN32_QueueKernelApc
0x002A0010	VWIN32_Int21Dispatch

0x002A0011	Calls IFSMgr_Win32DupHandle
0x002A0012	VWIN32_BlockThreadSetBit
0x002A0013	Adjust_Thread_Exec_Priority
0x002A0014	_VWIN32_Get_Thread_Context
0x002A0015	_VWIN32_Set_Thread_Context
0x002A0016	Read process memory (used by ReadProcessMemory)
0x002A0017	Write process memory (used by WriteProcessMemory)
0x002A0018	Calls VMCPD_Get_CR0_State
0x002A0019	Calls VMCPD_Set_CR0_State
0x002A001A	SuspendThread
0x002A001B	ResumeThread
0x002A001C	未知
0x002A001D	WaitCrst
0x002A001E	WakeCrst
0x002A001F	Something to do with loading/unloading VxDs
0x002A0020	VMCPD_Get_Version
0x002A0021	Set_Thread_Win32_Pri
0x002A0022	Calls Boost_With_Decay
0x002A0023	Calls Set_Inversion_Pri
0x002A0024	Calls Release_Inversion_Pri_ID
0x002A0025	Calls Release_Inversion_Pri
0x002A0026	Calls Attach_Thread_To_Group
0x002A0027	Calls Set_Thread_Static_Boost
0x002A0028	Calls Set_Group_Static_Boost
0x002A0029	VWIN32_Int31Dispatch
0x002A002A	VWIN32_Int41Dispatch
0x002A002B	VWIN32_BlockForTermination
0x002A002C	TerminationHandler2
0x002A002D	未知
0x002A002E	dwBlockSingleWnod(WaitForSingleObject)
0x002A002F	dwBlockMultipleWnod(WaitForMultipleObjects)
0x002A0030	VWIN32_SetEvent
0x002A0031	Something to do with delivering APCs
0x002A0032	未知
0x002A0033	InitUserAPCList
0x002A0034	未知
0x002A0035	Calls VMM Signal_Semaphore_No_Switch
0x002A0036	Calls System_Control(KERNEL32_INITIALIZED)

0x002A0037	VWIN32_CommonFaultPopup
0x002A0038	VWIN32_ForceCrsts
0x002A0039	未知
0x002A003A	VWIN32_FreezeAllThreads
0x002A003B	VWIN32_UnFreezeAllThreads
0x002A003C	Calls IFSMgr_Ring0_FileIO
0x002A003D	Calls Get_ Initial_ Thread_ Handle, Attach_ Thread_ To_ Group, andBoost_ Thread_With_VM
0x002A003E	VWIN32_ActiveTimeBiasSet
0x002A003F	ModifyPagePermission (used by VirtualQueryEx)
0x002A0040	Used by VirtualQueryEx
0x002A0041	ForceLeaveCrst
0x002A0042	ForceEnterCrst
0x002A0043	Calls VMCPD_Set_Thread_Excpt_Type
0x002A0044	VTD_Get_Real_Time
0x002A0045	Calls System_Control(SET_DEVICE_FOCUS)
0x002A0046	Calls VWIN32_UnFreezeThread
0x002A0047	Calls VMM_Replace_Global_Environment
0x002A0048	Calls System_Control(KERNEL32_SHUTDOWN)
0x002A0049	未知
0x002A004A	VW32_AddSysCrst
0x002A004B	VW32_SetTimeOut
0x002A004C	VW32_Cancel_Time_Out
0x002A004D	未知
0x002A004E	Something to do with setting and reflecting hotkeys

其中：

Crst 是 Critical Section

APC 是 Asynchronous Procedure Call

VMCPD 是 the Virtual Math Coprocessor Device

IFSMgr 是 the Installable File System Manager

System_Control 是向各 VxD 播放消息的 VMM. VXD ring 0 服务

看了这张表你就知道，VWIN32 提供了大量的 Win32 VxD 服务——这么多，实际上，单为它们，我就可以写一本书。当然是另一本，不是这本书。在这本书中，我想集中讲述前面表格中所列服务的其中三个。

0x002A0010 — VWIN32_Int21Dispatch (the DOS interrupt)

0x002A0029 — VWIN32_Int31Dispatch (the DPMI interrupt)

0x002A002A — VWIN32_Int41Dispatch (the debugger notification interrupt)

微软公司声明 Win32 代码不能象 Win16 代码那样可以申请中断。然而，这并不意味着不需要使用中断。当 Win32 代码需要申请 INT 21h，31h，41h 中断时，VWIN32. VXD 中的 Win32 VxD 服务可以很好的完成工作。KERNEL32. DLL 中到处使用这些中断调度函数。让我们分析一下 VWIN32. VXD 中的 VWIN32_Int31Dispatch 服务函数的代码，看它是怎样工作的：

VWIN32_Int31Dispatch 的伪码：

```
// Parameters:
//      Client_Reg_Struct   * pClientRegs
//      DWORD   ring3_EAX
//      DWORD   ring3_ECX

    _Debug_Flags_Service( DFS_TEST_BLOCK );

    EAX = pClientRegs->Client_EAX = ring3_EAX
    ECX = pClientRegs->Client_EAX = ring3_ECX

    Exec_PM_Int( EAX = 0x31 );

    if ( carry set )
        _Debug_Out_Service( "VW32_Int31Dispatch: Exec_PM_Int Failed!\r\n" );
```

VWIN32 中的中断调度服务不过如此。调度中断的 Win32 VxD 服务只是 ring 0 Exec_PM_Int 服务调用的外壳。有关 Windows 95 有许多有些夸张的说法，其中心在于“DOS 已成为过去”这个概念。因为几乎所有以前调用 DOS 功能完成的工作都放在 VxD 中了，这些中断调度 Win32 VxD 服务不会被那么频繁地使用，对吗？好吧，下面是 KERNEL32. DLL 中 FindClose 函数的伪码，分析它并作出你自己的判断。

FindClose 的伪码：

```
// Parameters:
//  HANDLE  hFile;

    x_LogSomeKernelFunction( function number for FindClose );

    if ( hFile == HFILE_ERROR )
        goto error;     // Calls SetLastError(ERROR_INVALID_HANDLE), then
                        // returns FALSE to caller.

    EAX = 71A1h         // 71A1h == Long Filename FindClose code
    EBX = hFile;
    INT_21H_DISPATCH();

    if ( carry flag set )
        goto error;     // Calls SetLastError(ERROR_INVALID_HANDLE), then
                        // returns FALSE to caller.
    return TRUE;
```

INT_21h_DISPATCH 的伪码：

```
return VxDCall( 0x002A0010, EAX, ECX );
```

实际情况是，KERNEL32 中有数十个对 VWIN32_Int21Dispatch 服务的调用。通过在 KERNEL32.DLL 中的搜索可以看出，KERNEL32.DLL 可以执行如下的 INT 21h(DOS)调用。

DOS 子函数	功能
0E00	设置隐含驱动器
1900	获得当前驱动器
2A00	获得系统日期
2B00	设置系统日期
2C00	获得系统时间
2D00	设置系统时间
3600	获得硬盘剩余空间
3D00	打开已存在文——只读
3D02	打开已存在文件——读/写
3E00	关闭文件
3F00	读文件
4000	写文件
4200	设置当前文件指针——相对于文件开头
4201	设置当前文件指针——相对于文件指针当前位置
4202	设置当前文件指针——相对于文件结尾
4400	IOCTL——获得驱动器信息
4401	IOCTL——设置驱动器信息
4408	IOCTL——检查是否是可移动块设备
4409	IOCTL——检查是否是远程块设备
440D	IOCTL—— 一般块设备请求
4B00	运行可执行程序
4D00	获得返回代码
5000	设置当前 PSP
5700	获得文件日期/时间
5701	设置文件日期/时间
5704	设置文件扩展属性
5705	未知
5706	未知
5707	未知

5900	获得扩展错误信息
5C00	锁定文件区
5C01	解锁文件区
5E00	网络功能
5F32	未知
5F33	未知
5F34	未知
5F35	未知
5F36	未知
5F37	未知
5F38	未知
5F3B	未知
5F3C	未知
5F4D	未知
5F4F	未知
5F52	未知
6800	提交文件
7139	LFN 创建目录
713A	LFN 删除目录
713B	LFN 改变目录
7141	LFN 删除文件
7143	LFN 获得/设置文件属性
7147	LFN 获得当前目录
714E	LFN 查找第一个文件
714F	LFN 查找下一个文件
7156	LFN 改变文件名
7160	LFN 获得规范的文件名
716C	LFN 扩展的打开/创建
71A0	LFN 获得卷信息
71A1	LFN 查找关闭
71A3	未知
71A4	未知
71A5	未知
71A6	LFN 通过句柄获得文件信息
71A7	LFN 把文件时间转换成 DOS 时间
B400	未知
EA00	未知

哇！KERNEL32.DLL 中做了这么多 INT 21h 调用。虽然有人认为 DOS 已被 Windows 95 消灭了，从这似乎可以看出 INT 21h 依然在我们身边。唯一的改变就是，现在 INT 21h 是被 KERNEL32.DLL 调用，而不是由我们自己的代码调用。

为什么微软公司要克服重重困难来完成这些 INT 21h 调用？他们不能直接调用底层的操作系统功能，从而绕过这已有 15 年历史的 INT 21h 接口吗？当然可以。但是，如果微软公司这么做，那么与 INT 21h 调用紧密相关的设备驱动程序和 VxD 就崩溃了，因为它们将看不到所期望的基本操作系统功能。再者，微软公司的立场是用略差的代码换来和原有的应用程序及设备驱动程序的兼容。

现在回到我们前面关于 VWIN32.VXD 调度某个中断的讨论，你可能对 INT 21h(DOS)和 INT 31h(DPMI 中断)已经熟悉了，而 INT 41h 你可能还没有听说过。INT 41h 是这样一个中断，它被操作系统 KERNEL 用来把一些重要的事件通知给系统级的调试器(WDEB386，SoftIce/W)。例如，KERNEL32.DLL 调用下面的 INT 41h 子函数(它在 DDK 的 DEBUGSYS.INC 中有说明)：

```
DS_LoadSeg_32   equ  0150h ; Define a 32-bit segment for Windows 32.
DS_FreeSeg_32   equ  0152h ; Notify the debugger that a segment has been freed.
DS_Printf       equ  0073h ; Formatted output standard "C" printf syntax.
DS_Out_Str      equ  0002h ; Function to display a NUL terminated string.
DS_IntRings     equ  0020h ; Tell debugger which INT 1's & 3's to grab.
DS_CheckFault   equ  007Fh ; Checks if the debugger wants control on the fault.
DS_CondBP       equ  F001h ; Conditional breakpoint.
```

VWIN32 TDBX

在第三章，我讲了由 KERNEL32 维护的进程数据库和线程数据库。明白了 VWIN32 怎样如此密切地与进程、线程相联系，就不会惊讶，VWIN32 还有自己的数据结构，用来保持对进程和线程的跟踪。这个数据结构称作 TDBX，在第三章已经引用过。

系统中的每一个线程都有一个 TDBX。指向 TDBX 的指针放在线程控制块(THCB，the Thread Control Block)中一个线程局部存储(TLS，Thread Local Storage)槽中。THCB 是 VMM 线程控制器用来跟踪它所建立的线程的局部数据结构。其它 VxD 可以从 THCB 中为它们的每个线程的存储申请一个槽位。它们通过 VMM 的_AllocateThreadDataSlot 函数完成这一功能，该函数返回 THCB 中一个偏移量，在那里可以保存一个指向线程数据的指针。VWIN32 TDBX 结构是在 KERNEL32 调用 Win32 VxD 服务 0x002A0008(VWIN32_CreateThread)时分配的。指向 TDBC 结构的指针保存在每个 VMM 线程控制块中为它保留的双字槽位中。

干脆点，让我们就来看一看 VWIN32 TDBX 结构的内容。和本书中讲的其它结构不同，TDBX 的各字段的含义只能根据它的名字猜测。

00h　　DWORD　　ptdb

一个指针，指向和该线程关联的 ring 3 PROCESS_DATABASE 结构。这个结

构的格式在第三章“Windows 95 进程数据库”一节讲过。

04h　DWORD　ppdb

一个指针，指向与本线程关联的 ring 3 THREAD_DATABASE 结构。该结构的格式在第三章中给出。

08h　DWORD　ContextHandle

一个指针，指向和本线程相关联的内存文本结构。该结构在第五章讲述。

0Ch　DWORD　un1

未知。

10H　DWORD　TimeOutHandle

未知。

14h　DWORD　WakeParam

未知。

18h　DWORD　BlockHandle

未知。

1Ch　DWORD　BlockState

未知。

20h　DWORD　SuspendCount

为这个线程调用 Win32 SuspendThread 函数的次数。

24h　DWORD　SuspendHandle

未知。

28h　DWORD　MustCompleteCount

当该值非 0 时，这个线程不能被中断。在第三章和第五章讲的 EnterMustComplete 和 LeaveMustComplete 函数分别增加和减少这个值。

2Ch　DWORD　WaitExFlags

这个线程的状态标志，下面是已知含义的值：

0x00000001　WAITEXBIT
0x00000002　WAITACKBIT

0x00000004	SUSPEND_APC_PENDING
0x00000008	SUSPEND_TERMINATED
0x00000010	BLOCKED_FOR_TERMINATION
0x00000020	EMULATE_NPX
0x00000040	WIN32_NPX
0x00000080	EXTENDED_HANDLES
0x00000100	FROZEN
0x00000200	DONT_FREEZE
0x00000400	DONT_UNFREEZE
0x00000800	DONT_TRACE
0x00001000	STOP_TRACING
0x00002000	WAITING_FOR_CRST_SAFE
0x00004000	CRST_SAFE
0x00040000	BLOCK_TERMINATE_APC

30h DWORD SyncWaitCount

未知。

34h DWORD QueuedSyncFuncs

未知。

38h DWORD UserAPCList

(APC 是 Asynchronous Procedure Call 的缩写,意思是同步过程调用)。

40h DWORD pPMPSPSelector

一个指针,指向保护模式的 PSP 选择器。

44H DWORD BlockedOnID

未知。

48h DWORD un2[7]

未知。

64h DWORD TraceRefData

未知。

68h DWORD TraceCallBack

未知。

6Ch　DWORD　TraceEventHandle

未知。

70h　WORD　TraceOutLastCS

未知。

72h　WORD　K16TDB

和本线程关联的 Win16 任务数据库选择器。

74h　WORD　K16PDB

和本线程关联的 Win16 程序段前缀(PSP)选择器。

76h　WORD　DosPDBSeg

和线程的进程相关联的 PSP 的实模式段值。

78h　WORD　ExceptionCount

未知。

前两个字段最吸引人。它们把指向 ring 3 KERNEL32. DLL 使用的进程和线程数据结构的指针提供给 ring 0 VWIN32. VXD。(回想一下第三章学过的内容，KERNEL32 在 ring 3 THREAD_DATABASE 结构中为每一个线程保存一个 TDBX 指针。把它们两两放到一起，你就会发现，KERNEL32. DLL 的 THREAD_DATABASE 和 VWIN32. VXD 的 TDBX 中指针是相互指向的。)

一些其它的 TDBX 字段也可以作类似的分析。在偏移量 8 处是一个指针，指向该线程的内存文本(实际上，是线程所属进程的)。同样，象我们在第三章和第五章看到的，当进程绝对不允许中断时，MustCompleteCount 字段对最低层次的线程同步来讲，就变得至关重要。

最后，TDBX 结构包含一个 Win16 任务数据库的选择器，每个进程都可以得到它(不管进程是 16 位还是 32 位)。最后两个字段是保护模式和实模式指针，指向线程拥有的进程的程序段前缀。显然，ring 0 VWIN32. VXD 不仅需要知道 Win16 的 KRNL386 的数据结构，也需要知道 DOS 的数据结构。关键的一点是 Windows 95 的所有三个核心部件(16 位 KRNL386. EXE，ring 3 KERNEL. DLL 和 ring 0 VWIN32. VXD)之间能相互访问。下边我们讨论这些部件的内部联系。

Windows 95 的三个核心部件怎样通信

在 VWIN32. VXD 中经过了漫长而又曲折的历程后，我们终于有足够的基础来讨论 Windows 95 的三个部件是怎样相互通信的。这三个 Windows 95 的核心部件通信和相互作用的程度令人惊讶，至少我是如此。为什么是这样？就我的(我承

认不切实际)看法而言,我认为一个操作系统的核心是一个独立的实体,不依赖其它任何事物。它是建立其它任何事物的基础,不应当依靠外部的部件。换句话说,我倾向于一个核心可以被看成一个黑盒子,理解这个黑盒子的工作不需要知道黑盒子以外部件的任何事情。然而,就象我在这本书中讲过的,这三个核心黑盒子并不是真的这样黑。下边我要作的就是说明每个核心是怎样得到外部知识和与其它核心模块交互作用的。

VWIN32 的 KRNL386 的知识

VWIN32.VXD 了解 KRNL386 及其数据的第一个证明出现在 TDBX 结构的尾部,在那里你可以找到 TDBX 所关联的进程的 Win16 任务数据库(TDB)选择器。然而,一个更生动的关于 VWIN32 的 KRNL386 知识的例子来自 VWIN32.VXD 中一个例程的伪码,我把该例程称为 ThreadSwitchCallback。

ThreadSwitchCallback 被 VMM.VXD 调用。每当线程调度器切换到一个新线程时,就要调用这个例程。函数 ThreadSwitchCallback 是抢占式多线程的 Win32 和非抢占式的 Win16 KRNL386.EXE 的交汇点,同时也是多任务的 Win32 和单任务的 MS-DOS 的交汇点。它扮演这样的角色,使 Windows 95 就其一部分看起来是十足的多任务系统,而就其另一部分看来,又是象 DOS 和 Windows 3.1 那样的系统。

ThreadSwitchCallback 的伪码:

```
// Parameters:
//  THCB    *pCurrentTHCB, *pOldTHCB;   // Pointer to Thread Control Blocks.
// Locals:
//  PTDBX   pNewTDBX, pOldTDBX;         // Pointers to TDBX structures.

    // On entry, EAX is the old THCB and EDI is the current THCB
    // (THCB = thread control block).

    pNewTDBX = pCurrentTHCB->TDBX_pointer;
    if ( !pNewTDBX )
        return;

    pOldTDBX = pOldTHCB->TDBX_pointer;
    if ( !pOldTDBX )
        return;

    // Make sure the parameter that points to the old thread database
    // matches what VWIN32.VXD has saved away in a global variable. (cur_ptdb)
    if ( pOldTDBX->ptdb != cur_ptdb )
        _Debug_Out_Service( "VWin32: invalid current Win32 thread\r\n" );

    cur_ptdb = pNewTDBX->ptdb;  // Update VWIN32 current thread global var.
    cur_ppdb = pNewTDBX->ppdb;  // Update VWIN32 current process global var.
    CurTDBX = pNewTDBX;         // Update VWIN32 current TDBX global var.

    // This line bashes the CurTDB global variable in KRNL386.EXE.
    *pWin16CurTDB = pNewTDBX->K16TDB;
```

```
// If the new thread differs from the old thread, update the PSP
// segment down in DOS, and save away the old PSP segment.
if ( prevTDBX != pNewTDBX )
{
    // Save away the current PSP segment for the previous thread.
    prevTDBX->DosPDBSeg
            = Get_Set_Real_DOS_PSP( ECX=0, EBX = Get_Sys_VM_Handle() );

    // Set the current PSP segment for the new thread. This should
    // bash a variable down in DOS.
    Get_Set_Real_DOS_PSP( AX = pNewTDBX->DosPDBSeg, ECX=1,
                          EBX = Get_Sys_VM_Handle() );

    prevTDBX = pNewTDBX;        // prevTDBX is a VWIN32 global variable.
}

// Switch the memory address context.
if ( pNewTDBX->ContextHandle )
{
    CurContext = pNewTDBX->ContextHandle;   // Update VWIN32 global var.

    _ContextSwitch( pNewTDBX->ContextHandle );
}
```

完成一些初步的完整性检查后，ThreadSwitchCallback 更新 VWIN32 用来保存当前 ring 3 进程和线程库的全局变量指针。同时，它还更新 VWIN32 用来指向当前 TDBX 结构的全局变量指针。然后，它做了一些其它事情，当我第一次用 SoftIce/W 看到这些事情时，着实吃了一惊。这是我从未看到过的，ring 0 VWIN32.VXD 把 KRNL386.EXE 中的全局变量 CurTDB 消灭了。

从开始，一直到 Windows 95 出现，CurTDB 一直是一个神圣的变量。在 Windows 3.x 中，改变 CurTDB 的值的唯一途径是调用核心调度例程（也就是函数 Reschedule）。Windows 3.x 中非常有序的协同式多任务世界和 Windows 95 中的抢占式多任务发生了冲突，而抢占式多任务取得了胜利。这是一个极其病态的世界，KRNL386 中基本的东西，当前任务的全局变量居然被有些程序员都知道的另外的部件消灭了。

ThreadSwitchCallback 要完成的其它工作还包括与多任务有关的清除工作。被移出的线程的 PSP 段保存在它的 TDBX 结构中。然后，取得正调入的线程的 PSP 段，并用它设置系统 VM。这样的 PSP 切换对 Windows 95 来说并不是新东西。Windows 3.1 尽管运行 ring 3 KRNL386 代码，但也可以做类似的事情。ThreadSwitchCallback 要做的最后一件工作是把当前内存文本切换为刚调入线程的内存文本。在第五章讲过，切换内存文本就使得每个进程能拥有自己的私有地址空间。在 Windows 95 中，进程的私有地址空间是 4MB 和 2GB 以上的线性地址空间。

VWIN32 的 KERNEL32. DLL 知识

VWIN32 了解 KERNEL32 的主要证明就是，在 VWIN32 使用的 TDBX 结构中，存在指向 THREAD_DATABASE 和 PROCESS_DATABASE 的指针。VWIN32 还有指向由 KERNEL32 维护的指向当前进程和线程的全局指针变量。除进程和线程外，在 KERNEL32 的启动阶段，VWIN32 还能得到 KERNEL32. DLL 中各例程的指针的列表。KERNEL32. DLL 通过把函数的地址作为参数传递给 Win32 VxD 中识别号为 0x002A0003 的服务，向 VWIN32 提供这些信息。

KERNEL32. DLL 的 VWIN32 知识

KERNEL32 了解 VWIN32 的最有力的证明就是 KERNEL32 调用了 VWin32 提供的 Win32VxD 服务这个事实。这在本书中，尤其是在本章的前面几节，已多次说明过。对这个问题，没有更多需要说明的。

除了 Win32 VxD 服务，KERNEL32 和 VWIN32. DLL 之间其它合作出现在当一个特定的 Win32 VxD 服务被调用时，VWIN32. VXD 提交一个 VWIN32. VXD 中的函数地址的列表。这个调用就是 Win32 VxD 服务 0x002A0001，它被 KERNEL32. DLL 中的 FInitPager 函数调用。对此，一个可能的解释是，在执行页面错误处理时，KERNEL32. DLL 将调用这些 VWIN32. VXD 中的地址。正象《Unauthorized Windows 95》中所讲的，VMM 的页面错误处理调用了（在 ring 0）KERNEL32. DLL 的代码。也可以认为在 KERNEL32. DLL 的页面初始化期间，VWIN32 将把自己各例程的地址传递给它。

KERNEL32. DLL 的 KRNL386. EXE 知识（或者说，微软公司没告诉你的东西）

根据微软公司的 Windows 95 技术资料，32 位 KERNEL32 在任何功能上都不依赖 KRNL386。（可以与 USER 和 GDI 部件作个比较，在 USER 和 GDI 中，允许 32 位部件使用 Win16 部件。）《Unauthorized Windows 95》彻底打破了微软公司关于 KERNEL32. DLL 不调用 16 位的 KRNL386. EXE 的声明。然而，目前还不清楚 KERNEL32 调用 KRNL386 的程度。

那本书对 Windows 95 中 KERNEL32 和 KRNL386. EXE 之间的相互调用已作了很好的说明，我不想再重复这些内容。不过，我要提供一些新的东西——被 KERNEL32. DLL 调用的 KRNL386. EXE 的函数的列表，不知道你是否感兴趣。这个列表特别与微软公司的一个声明有关，他们说调用任何 KERNEL32 函数都不会导致调用线程为申请名为 Win16Mutex 的服务而等待。而你在下表可以看到，有一组 KERNEL32 的函数就阻塞在 Win16Mutex 上，且其中的函数无一不重要。

请考虑 profile 类的函数(例如,GetPrivateProfileSection),不要认为这些函数不会带来什么问题。典型情况下,它们访问硬盘后能很快返回。但是,如果它们要查找的文件在 CD-ROM 驱动器上,并且驱动器上并没有盘片时会怎么样呢? CD-ROM 驱动程序要等待许多秒才能超时退出,并且在这一段时间里,Win16Mutext 被这个调用任务占有。(这个现象我在 Windows 95β 中确实遇到了。)

下边按序对下表作一解释。在左列的名字是 KERNEL32 内部使用的名字。如果一个特定的 KERNEL32 内部标志被置位,KERNEL32 就向调试终点处提供这个串。这些函数中有些是以普通的被外部引用的 KERNEL32 函数出现。另一些是未公开的函数或被外部引用的 KERNEL32 函数的变种。还不知道普通的应用是否能调用它们。例如,LoadLibrary16 调用 KRNL386 中的 16 位 LoadLibrary。LoadLibrary16 被外部从 KERNEL32. DLL 引用时,其函数号为 35,但它和普通的 KERNEL32. DLL的LoadLibrary函数不同。同样的,WritePrivateProfileSection32A 函数也不同于普通的 KERNEL32 函数 WritePrivateProfileSectionA 函数。其名字是严格匹配的。

下表右边一列函数名是 KERNEL32 的 thunk(形实替换程序)直接调用的一组 KRNL386 的函数。这一列没有函数名的地方,该 thunk 不直接调用引出函数。

KERNEL32. DLL 的内部名字	KRNL386. EXE 的引出名字
TerminateZombie	
DiagOutput16	DiagOutput
DispatchRITInput	
GetFastQueue	KERNEL. 625
SetVolumeLable16	
PK16FNF	
CommConfigThk	
InitAtomNameA	
DeleteAtom	
FindAtomA	
AddAtomA	
GlobalLock16	GlobalLock
IsDriveCDRom	
ExecConsoleAgent	
ThkOpenFile	OpenFile
GetErrorMode	
SetErrorMode16	
GetSystemDirectoryA	GetSystemDirectory
GetWindowsDirectoryA	GetWindowsDirectory

GlobalGetAtomNameA	
GlobalDeleteAtom	
GlobalFindAtomA	
GlobalAddAtomA	
GetPrivateProfileSectionNames32A	GetPrivateProfileSectionNames
WritePrivateProfileStruct32A	WritePrivateProfileStruct
GetPrivateProfileStruct32A	GetPrivateProfileStruct
WriteProfileSectionA	WriteProfileSection
GetProfileSectionA	GetProfileSection
WritePrivateProfileSection32A	WritePrivateProfileSection
GetPrivateProfileSection32A	GetPrivateProfileSection
WritePrivateProfileString32A	WritePrivateProfileString
GetPrivateProfileString32A	
WriteProfileStringA	WriteProfileString
GetProfileStringA	
GlobalHandle16	GlobalHandle
GlobalSize16	GlobalSize
GlobalFlags16	GlobalFlags
GlobalUnlock16	GlobalUnlock
GlobalFree16	GlobalFree
GlobalReAlloc16	GlobalRealloc
GlobalAlloc16	GlobalAlloc
WinExecEnv	
PrivateGetModuleFileName	GetModuleFileName
GetProductName	GetProductName
GetWinFlags	GetWinFlags
GetModuleName16	
GetTaskName16	
SetTaskName16	
ThkDeleteTask	
ThkCreateTask	
ThkInitWin32Task	
FreeSelector16	
ThunkInitLSWorker16	
GetProcAddress16	
FreeLibrary16	FreeLibrary
LoadLibrary16	LoadLibrary
GlobalUnWire16	GlobalUnWire

GlobalWire16	GlobalWire
GlobalUnfix16	GlobalUnfix
GlobalFix16	GlobalFix
GlobalNukeGroup	
CheckHGHeap	
SegCommonThunkDetach32	
SegCommonThunkAttach32	
GrowMBABlock	
FakeThunkTheTemplateHandle	
TCD_UnregisterPDB32	
TCD_Enum	
WOWGlobalLockSize16	
WOWGlobalUnlockFree16	
WOWGlobalAllocLock16	
WOWGlobalUnlock16	GlobalUnlock
WOWGlobalLock16	GlobalLock
WOWGlobalFree16	GlobalFree
WOWGlobalAlloc16	
WOWDirectedYield16	DirectedYield
WOWYield16	Yield
Yield16	
FreeLibrary16ByName	
SSChk	
UTThunkLSHelper	
UTUnregisterInt	
UTRegisterInt	
UTProcessExit	
FreeCB	

我注意到，许多 Windows 95 程序员急于知道在调用 KRNL386 过程中，哪些 KERNEL32 函数会被阻塞在 Win16Mutex 处。微软公司的没有一个函数会被阻塞的说法根本就靠不住。通过研究下面的列表，你可以很容易地得到等待 Win16Mutex 时会被阻塞的 KERNEL32 函数并把它们归并为几类。

函数分类	函数名
Atom 操作函数	AddAtomA，DeleteAtom，FindAtomA，GetAtomNameA，GlobalAddAtomA，GlobalDeleteAtom，GlobalFindAtomA，GlobalGetAtomNameA，InitAtomTable

目录操作函数	GetSystemDirectoryA，GetWindowsDirectoryA，
WIN. INI 文件操作函数	GetProfileSectionA，GetProfileStringA，WriteProfileSectionA，WriteProfileStringA

KERNEL32 也知道 KRNL386. EXE 的专用全局变量。我们早已看过 VWIN32. VXD 怎样与 KRNL386. EXE 的全局变量(比如 CurTDB)合作。对于 KERNEL32，最明显的例子是它使用了 KRNL386 的全局变量 Win16Mutex。Win16Mutex 实际上不过是一个 CRITICAL_SECTION 结构，保存在 KRNL386. EXE 的 DGROUP 段中。KERNEL32 怎样得到 Win16Mutex 的地址呢？在装入 KERNEL32. DLL 后，KRNL386. EXE 把 Win16Mutex 的地址作为一个初始化调用的一部分传递给 KERNEL32. DLL。

KRNL386 的 KERNEL32. DLL 知识

KERNEL32 有一个关于它调用的 KRNL386. EXE 中函数的长长的列表，反过来也是一样。下边的表格列出了 KRNL386 从 KERNEL32. DLL 中调用的函数。

函数分类	函数名
进程管理函数	CreateProcessFromWinExec，IFatalAppExit，NukeProcess，RegisterServiceProcess，ThunkExitProcess，ThunkMapProcessHandle，ThunkCreateProcessWin16，WinExecWait
线程管理函数	IsThreadId，ThunkCreateThread16，ThunkTerminateThread
模块管理函数	GetModuleHandle32，GetNEPEBuddyFromFileName32，LoadLibraryEx32W，ThunkFreeLibrary32（释放 Win32 DLL），ThunkGetHModK16FromHModK32(从 Win32 模块句柄得到 Win16 模块句柄)，ThunkGetModuleFileName，ThunkLoadLibrary32(装入一个 Win32 DLL)
目录管理函数	GetCurrentDirectory(存放在每个进程的 KERNEL32 进程数据库中)，SetCurrentDirectory32，ThunkGetCurrentDirectory
文件 I/O 函数	FindClose，FindFirstFile，FindNextFile(Win32 Findxxx 函数的 16 位版本)，FileTimeToDosDateTime，OpenFileEx16And32，ThunkCloseDosHandles，ThunkCloseW32Handle
32 位堆函数	LocalAlloc32NG，ThunkLocal32Alloc，ThunkLocal32Free，ThunkLocal32Init，ThunkLocal32ReAlloc，ThunkLocal32SizeThkHlp，ThunkLocal32Translate，ThunkLocal32ValidHandle
同步函数	ThunkCreateW32Event，ThunkResetW32Event，ThunkSetW32Event，WaitForMultipleObjects，WaitForSingleObject
错误处理函数	CreateFaultThread，FaultRestore，FaultSave

WINOLDAP 支持函数	WOAAbort, WOACreateConsole, WOADestroyConsole, WOAFullScreen, WOAGimmeTitle, WOASpawnConApp, WOATerminateProcess
资源清理函数	FreeInitResources32, HGCleanupDepartingHTask, NotifyDetachFromWin16, ThunkDeallocOrphanedCrsts,
其它函数	CallProc32WFixHelper, CallProc32WHelper, FlatCommonThunkConnect16, FullLoRes, GetProcessDword, GetVersionEx, InitK32AfterSysDllsLoaded, ISetErrorModeEx, InvalidateNLSCache, LateBindWin32ThunkPartner, SetProcessDword, SmashEnvironment, ThunkConvertToGlobalHandle, ThunkGetProcAddress32, ThunkTheTemplateHandle, VirtualFree

也许你想知道前边表中的名字是从哪里来的，它们都嵌入在 KERNEL32.DLL 中。通过研究对 KERNEL32.DLL 内容的列表，我注意到被 KRNL386.EXE 引用的函数的代码有一致的模式。作为模式的一部分，它们都含有一个指向函数名的指针。写一个编辑器宏从 KERNEL32.DLL 中找到所有有这种代码模式的位置，并把函数名拷贝到一个文件中，是一件很简单的事情。

KRNL386 的 VWIN32 知识

KRNL386.EXE 的 VWIN32.VXD 知识具体表现在它对 VWIN32 PM API 服务的调用，这些服务我在前面题为“VWIN32.VXD 16 位保护模式 API”一节讲过了。

Win32 VxD 服务侦探程序(W32SVSPY)

如果不给出一个程序，使你能在我讲述的范围内仔细探究，这一章就显得不够完整。我写了一个 W32SVSPY 程序来侦探 Win32 VxD 服务调用。从某种意义上说，它就象第十章中的 API 侦探程序——只是没那么完整。在写 W32SVSPY 中，也有两个技术难题，我用很有意思的方式解决了它们。所以，我还将用点时间讲一讲从 W32SVSPY 学到的技巧。

随书所附的磁盘上有完整的源程序。我不打算讲述 W32SVSPY 的所有内部工作，因为它们相当复杂，并且没有太多人感兴趣。但是，W32SVSPY 的输出应当是大家所感兴趣的。

图 6-4 给出了 W32SVSPY 的初始窗口。这是个大列表框，窗口用来存放侦探活动期的输出。要开始观察 Win32 VxD 的服务调用情况，点击 Start 按钮，这时登记马上开始，它将一直持续到你点击 Stop 按钮，或 W32SVSPY 的缓冲区被填满。(我设置的可以存储的调用的数目为 16K。你可以改变这个值，并重新编译 W32SVSPY 源程序。)Save...按钮使你可以将监视活动的结果输出到一个文本文件中。

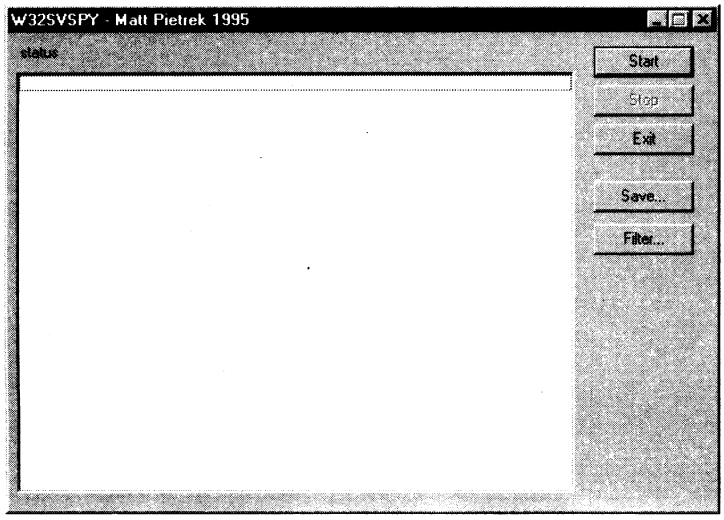


图 6-4　W32SVSPY 的初始屏幕

点击 Filter...按钮，会打开图 6-5 所示的对话框。这个对话框使你可以从侦探的结果中滤掉指定的 Win32 VxD 服务——后边将会看到。在 Win32 VxD 服务中有许多杂乱的东西，把它们滤掉的功能很有用。该过滤对话框有两个列表框。从左边的列表框中选择一个 VxD，就会用这个 VxD 中的 Win32 VxD 服务更新右边的列表框。现在，前面带＋(加号)的服务是使能的(即显示出来)。在右边的服务行上，用鼠标双击，可以使该行的状态在＋(使能)和(不使能)之间切换。缺省情况下，使能所有的服务。这里提供的 W32SVSPY 只识别 VMM 和 VWIN32 的服务。它登记所有的 Win32 VxD 服务调用，但只能给出那些它认识的 Win32 VxD 服务的名字。如果你想加入其它 VxD 中的 Win32 VxD 服务的知识，可以在 W32-SRVDB.C 文件中增加。

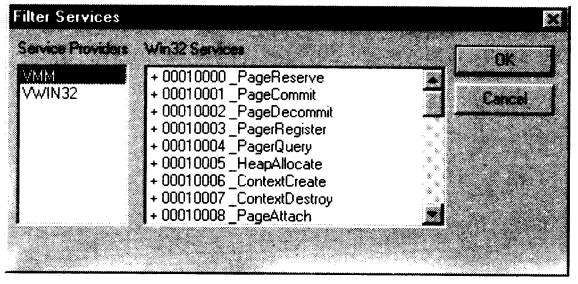


图 6-5　W32SVSPY 的过滤对话框

你在对话框中执行的任何过滤，都可以保留起来，方法是向 W32SVSPY 申请把要滤掉的服务识别号(二进制形式)输出到一个文件中。如果你想关闭过滤(也

就是你想看到所有的东西),就在启动 W32SVSPY 之前删除.FLT 文件,也可以返回到 Filter...对话框,重新使能所有的函数。

W32SVSPY 输出的每一行的格式如下:

```
<CurrentTaskName>  <Service Name>(<parameter 1 value>)
```

例如,有一行内容是下面这样:

```
Explorer VWIN32_SetEvent(8154A230)
```

本例中,当前进程(也就是产生这个调用的进程)是 Explore。被调用的服务是 VWIN32_SetEvent。第一个参数的值(在括号中的值)是 8154A230h。W32SVSPY 没有为每个 Win32 VxD 服务显示所有参数,因为那样将使登记进程变得非常复杂。如果你想要这个特征,就作为练习,给它加上。

如果 Win32 VxD 服务调用是 VWIN32_Int21Dispatch,W32SVSPY 就将第一个参数的值转换成描述所引用 DOS 函数的字符串。这时,DOS 函数名字符串将出现在服务名(VWIN32_Int21Dispatch)后面,第一个参数前面。例如:

```
Calc     VWIN32_Int21Dispatch LFN File Time To DOS Time(000071A7)
```

有时候,你会看到下面这样的行:

```
FFFF56F3 VWIN32_Int41Dispatch(00000150)
```

这时,行中的第一个串是进程的识别号。这是由于 W32SVSPY 没能从 Windows 95 为每个进程建立的 Win16 任务库(TDB)中取出该进程的名字。这通常只发生在该应用正在启动,KERNEL32.DLL 尚未把应用名加到任务库中的时候。一般指出识别号所指进程并不困难——只要搜索所有的入口,直到它们停止。服务调用列表中下一个其名字从未出现过的任务可能就是你要找的进程。

W32SVSPY 工作的一个样本

为演示 W32SVSPY 能够做什么,从工作台上的快捷按钮启动一个应用,看看对于 Win32 VxD 有什么情况发生。为启动应用,选择 CALC.EXE 并在工作台上为它建一个快捷按钮。然后启动 W32SVSPY 并点击 Start 按钮。然后马上双击 CALC 快捷按钮。最后,点击 W32SVSPY 中的 Stop 按钮。运行结果将出现在图 6-6 所示的 W32SVSPY 的主列表中。在这里,你可以在列表框中浏览这些服务,也可以用 Save...按钮把结果存到一个文件中。

我完全按上面叙述的步骤运行了一遍,得到输出结果,然后把它们精简,剔除大量的杂乱信息和重复内容,并加了一点注释。图 6-7 就是最后的结果:

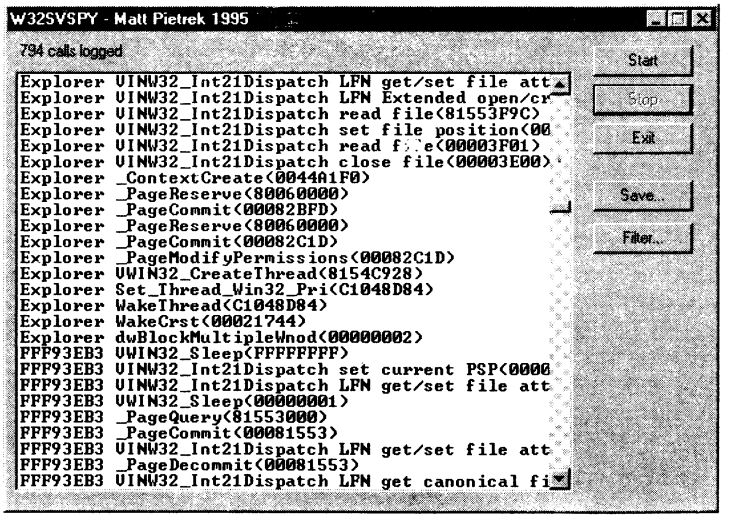


图 6-6　在 W32SVSPY 下运行 CALC

```
... Many preliminary calls to DOS and the registry by the Explorer omitted

// Create the memory context for the new process.
Explorer _ContextCreate(004463D8)

Explorer _PageReserve(80060000)
Explorer _PageCommit(000828B4)
Explorer _PageReserve(80060000)
Explorer _PageCommit(000828D4)
Explorer _PageModifyPermissions(000828D4)

// Create the initial thread for the new process.
Explorer VWIN32_CreateThread(8154915C)

// Set the priority of the initial thread of the new process.
Explorer Set_Thread_Win32_Pri(C10464D8)

// Wake up the initial thread of the new process. This should cause
// the new memory context to be switched to.
Explorer WakeThread(C10464D8)

Explorer WakeCrst(&Win16Mutex)
Explorer dwBlockMultipleWnod(00000002)

// The first act of the new process (take a nap!).
FFFF56F3 VWIN32_Sleep(FFFFFFFF)

// Do file I/O (presumably with the EXE file of the process).
FFFF56F3 VINW32_Int21Dispatch set current PSP(000050F7)
FFFF56F3 VINW32_Int21Dispatch LFN get/set file attributes(00007143)
FFFF56F3 VWIN32_Sleep(00000001)
```

```
FFFF56F3 VINW32_Int21Dispatch LFN get/set file attributes(00007143)
FFFF56F3 VINW32_Int21Dispatch LFN get canonical filename(00007160)
FFFF56F3 VINW32_Int21Dispatch set current PSP(000050B7)
FFFF56F3 VINW32_Int21Dispatch LFN Extended open/create(0000716C)
FFFF56F3 VINW32_Int21Dispatch read file(828D3F60)
FFFF56F3 VINW32_Int21Dispatch set file position(00004200)
FFFF56F3 VINW32_Int21Dispatch read file(00003F01)
FFFF56F3 VINW32_Int21Dispatch set file position(00004200)
FFFF56F3 VINW32_Int21Dispatch read file(00003F01)
FFFF56F3 VINW32_Int21Dispatch LFN get canonical filename(00007160)

// Reserve and commit the memory to be used by the process.
FFFF56F3 _PageReserve(00000400)
FFFF56F3 _PageCommit(00000400)      < - repeat similar calls 11 times
FFFF56F3 _PageReserve(80000400)
FFFF56F3 _PageCommit(00000420)
FFFF56F3 _PageReserve(80000400)
FFFF56F3 _PageCommit(00000440)
FFFF56F3 _PageFree(00440000)
FFFF56F3 _PageFree(00420000)

FFFF56F3 VINW32_Int21Dispatch set current PSP(0000502B)

// Tell the system debugger (WINICE) about the EXE's 6 sections.
FFFF56F3 VWIN32_Int41Dispatch(00000150)     < - repeat this sequence 6 times

FFFF56F3 _RegQueryValueEx(C1126A54)

// Reserve and commit the memory range where SHELL32.DLL is mapped in.
// Since some other process has already loaded SHELL32.DLL, certain
// SHELL32.DLL pages can be shared with this process. Share the pages
// by calling PageAttach.
FFFF56F3 _PageReserve(0007FE10)
FFFF56F3 _PageCommit(0007FE10)
FFFF56F3 _PageAttach(0007FE11)
FFFF56F3 _PageAttach(0007FE70)
FFFF56F3 _PageAttach(0007FE71)
FFFF56F3 _PageAttach(0007FE74)
FFFF56F3 _PageCommit(0007FE76)
FFFF56F3 _PageCommit(0007FE77)
FFFF56F3 _PageAttach(0007FE78)
FFFF56F3 _PageAttach(0007FEC8)

// Tell the system debugger (WINICE) about all the DLL sections.
FFFF56F3 VWIN32_Int41Dispatch(00000150)     < - repeat this sequence 33 times

// Checks if the debugger wants control during a fault.
FFFF56F3 VWIN32_Int41Dispatch(0000007F)

FFFF56F3 _VWIN32_Get_Thread_Context(00000000)
FFFF56F3 _VWIN32_Set_Thread_Context(00000000)
... Omitted rest of output
```

图 6-7　启动 CALC.EXE，精简后的 W32SVSPY 输出

通过研究图 6-7 中的输出，你可以看到一个新进程的诞生。一个很重要的事是注意在事件左边显示的进程名仅仅是当前进程。调用的 Win32 VxD 服务绝大多数来自 KERNEL32. DLL，是由当前进程发出的(这里是 Explore)。

输出从 Explore 进程(在 KERNEL32. DLL 进程中运行)建立新的内存文本开始。然而，内存文本不是由一个进程来建立的。有更多的事情要做。看输出的下边几行，注意为新进程建立一个初始化线程，同时也要注意线程优先级的设置。KERNEL32 的下一个动作是对新线程调用 WakeThread，它使得新线程成为当前运行的线程，注意两行后的内容。(这是进程名为 FFFF56F3 的第一行)

当新进程被唤醒后，它开始执行文件 I/O。这是在设定新进程在哪里检查它的. EXE 文件并把它装入内存。把 EXE 装入内存自然需要进程使用基于页面的内存管理 Win32 VxD 服务在进程的内存文本中保留和交付内存。在这个阶段系统确认它能和一个属于另一个进程的已装入自己私有地址的 DLL 共享特定的内存页。在这里，这个 DLL 是 SHELL32. DLL。当系统确认某一页可以共享(一般是代码页)时，就使用 VMM_PageAttach 服务开始和该进程(可能不止一个)共享该内存页。

编写 W32SVSPY 的技术挑战

当我刚开始构思编写 W32SVSPY 时，映入头脑中的第一个问题就是 KERNEL32 VxDCall 函数被系统中所有应用的内存文本调用。象我前边讲过的，Win32 VxD 服务是通过调用 VxDCall 函数完成的。所以我也必须把实际处理间接服务调用的代码放到内存中一个所有进程都能访问的地方(换句话说，就是我必须把代码放在共享内存中)。

另一个关于被所有内存文本申请的 Win32 VxD 访问调用的难题是在调用发生时，我只能登记它，而不能使用文件 I/O 操作。不能这样做的原因是文件句柄只在打开它们的内存文本中才是合法的。每次打开一个文件句柄时，将调用一个 Win32 VxD 服务，这是不可更改的。KERNEL32 使用 Win32 VxD 服务完成文件 I/O，所以这样很明显就导致可重入问题。(不是担心为每个调用打开和关闭文件句柄会引起系统停顿。)从而，解决方案就是使用所有进程都能访问的共享内存。W32SVSPY 把关于每个 Win32 VxD 服务调用的信息存进内存缓冲区，在安全时才把它们从缓冲区取出显示。

第三个难题是截获 Win32 VxD 服务。前面，我讲过 VxDCall 函数就象下面这段代码：

```
VxDCall:
MOV     EAX,DWORD PTR [ESP+04]
POP     DWORD PTR [ESP]
CALL    FWORD PTR CS:[BFFC8004]
```

截获对 Win32 VxD 服务的调用应当很简单，只要把我的代码和截获例程的地址放到内存地址 BFFC8004h 处，和原来的地址链接上即可。实际上，这一点都不困难。难就难在找到 VxDCall 的地址，这样才能取得指向 INT 30h 指令的指针的

偏移量。

为什么我不简单地对 VxDCall 调用 GetProcAddress，获得它的地址呢？因为这个函数是未公开的，不能用名字从外部调用。我也不能在调用 GetProcAddress 时使用 VxDCall 的序号。第三章说明了 GetProcAddress 怎样禁止应用程序用序号访问 KERNEL32.DLL 函数。微软公司为什么要这样做？毫无疑问是力求阻止人们写类似 W32SVSPY 这样的应用程序。当然你是知道的，微软公司的努力失败了，W32SVSPY 能够突破这粗糙的防御。

把 W32SVSPY 放进共享内存中

使 W32SVSPY 取得运行的第一部分工作是把它的代码放进所有进程可以共享的内存。在第三章中，我讲过完成这一工作的方法是把代码放进 DLL 中。因此，除了 EXE 文件，W32SVSPY 还有一个 DLL 部件(W32SPDLL.DLL)。但是，这样简单地把侦探代码放进一个 DLL 还不够。这个 DLL 还要能被装进内存中可以被所有进程共享的位置。在 Windows 95 中，这意味着 2GB 到 3GB 之间的内存范围。这是象 KERNEL32.DLL 和 USER32.DLL 这样的系统 DLL 所在的地方。我们需要用某种方法使 Windows 95 把 W32SPDLL.DLL 装入这块共享内存区。

现在，你的第一个倾向也许是告诉连接器把 W32SPDLL.DLL 的基址定在 2GB 到 3GB 之间的地址上。尽管你可以使连接器以你给出的任意地址作为 W32-SPDLL.DLL 的基址，但这还不够。我把 W32SPDLL.DLL 基址定于 2GB 以上的地址，然后装入它，但第一次努力失败了。嗯，这个 DLL 装好了。问题出现了，操作系统重定位了 DLL，使它定位于应用程序的私有地址空间。显然，Windows 95 装入程序不想把我的 DLL 装入为系统 DLL 保留的共享区域和共享内存。

通过研究微软提供的可以被装入程序装入 2GB 以上内存的 DLL，一个普遍的模式提醒了我。凡能被 Windows 95 装入程序成功地装入 2GB 以上共享内存的 DLL，它的可写数据段都被标志成共享的。现在，已经很明显了，如果装入程序把 DLL 装入到 2GB 以上的共享内存，Windows 95 自然就不能在数据段中提供单进程数据。是 DLL 中的单进程数据段引起的问题。通过下面的两步工作，我最终把 W32SPDLL.DLL 装入到 2GB 以上的内存。

- 使连接器把 DLL 基址定在 2GB 以上的一个地址。我选择了 KERNEL32.DLL 装入的地址，因为我知道 Windows 95 装入程序将为基址重叠的 DLL 重新定位。
- 使 DLL 的.data，.bss，和.idata 段成为共享的。通过在 W32SVSPY 的制作文件(makefile)中使用/section：开关完成。

上面的各项要求都用连接器响应文件中的以下几行完成：

```
-BASE:0xBFF70000
/section:.data,RWS
/section:.idata,RWS
/section:.bss,RWS
```

(注意：RWS 意思是可读(readable)，可写(writeable)，共享(shared))

查找 VxDCall 的地址

前面，我提到 Windows 95 的 KERNEL32. DLL 极力阻止应用程序获得 KERNEL32. DLL 中象 VxDCall 这样的未公开的函数。GetProcAddress 对这些函数根本就不起作用。然而，如果你隐含地连接一个函数，就能很容易地找到它的地址。在 C 或 C++ 中，你可以使用后面不跟括号的函数名。现在，就有办法隐含地连接这些未公开的函数。不过说回来，微软公司为什么不愿引出它们呢？

在附录 A 中，我给出了怎样为从 KERNEL32 引出的 100 个左右的未公开函数建立一个引入库。建立这个引入库，这样当你引入库中一个函数时，可以通过它的序号。在 W32SPDLL.C 中，我必须给出所期望函数的原型：

```
__declspec(dllimport) int WINAPI VxDCall0(void);
```

然后，我需要使用不带括号的函数名取得它的地址。

```
pfnVxDCall0 = VxDCall0;
```

一旦我知道了 KERNEL32. DLL 中 VxDCall0 的地址，那么进入函数的代码和取得保存执行 INT 30h 指令的 16:32 指针的地址就不是什么难事了。

```
ppfnOriginalVxDCall = (PBYTE)*(PDWORD)((DWORD)pfnVxDCall0 + 0xA);
```

这行看起来似乎有些不可理解，事实上事情并不那么糟。它只是取出 VxDCall 函数代码中 0xA 字节处的双字，然后把它转换成一个指针。

是的，依据 VxDCall 函数中一个固定的偏移量来找到一个指针相当令人讨厌。但当你编写象 W32SVSPY 这样的低层次的系统分析工具时，这往往是必不可少的。微软公司可以很容易地使 W32SVSPY 失效，他只要重新组织 VxDCall 的代码，使原来为 0xA 的偏移量改变。但是，没有理由使他们因为存心破坏 W32SVSPY 而将 VxDCall 弄乱。我们将可以看到 Windows 95 未来版本中会发生什么事，该多有意思。

小结

在这一章，我给出了大量未公开的函数和相当多的技术资料。不必力求读完本章就记住所有这些信息。我要告诉你的最重要的一点是，Windows 95 有可以调用的三个分离的部分(KRNL386. EXE，KERNEL32. DLL，和 VWIN32. VXD)。这三个核心的每一个都有关于另外两个核心的详细的知识，它们之间有非常广泛的相互作用。

如果想只理解其中一个部件，而不理解另外两个部件，这是非常困难的。你会发现你将反复地再读本章内容，以找到那些第一次阅读时没记住的地方。我希望我已经使你懂得 Windows 95 中三个核心部件之间相互作用的范围，并使你能够自己做更多的探索。

第七章

Win16 模块和任务

* *

这本书是讲 Windows 95 的 32 位体系结构的，中间插入一章关于 16 位 KERNEL 数据结构的内容，似乎显得有些古怪。然而，后边你会看到，在 Windows 95 的整个体系结构中，无论对于 16 位还是 32 位应用程序，这些数据结构都扮演着重要的角色。

在这一章，我们将在 KRNL386 中的 16 位模块和任务中作一次旅行。如果你熟悉 Windows 3.1 中的模块和任务，那么最先你会看到，它们在 Windows 95 中并没有改变。既然已经有最新、最好的 Win32 构件可以从外部调用，为什么还要为这些陈旧的 16 位的概念耗费心神？如果你研究的再深入一些，就会发现，16 位的 KRNL386.EXE、32 位的 KERNEL32.DLL 和 VWIN32 VxD 相互知道对方的内部情况，其操作是相互联系的。因此，研究 16 位模块和任务对学习 Windows 95 的体系结构很重要。

我们从一个模块开始，这个模块被 Windows 95 的 16 位部分用来跟踪装入系统中的 EXE 和 DLL。它看起来有些陌生，Windows 95 不仅为 16 位的 EXE 和 DLL，也为 32 位的 EXE 和 DLL 生成 16 位模块。我先讲述这个 16 位模块，然后讲述一些很有用的 16 位 KRNL386 函数的伪码，以演示实际中这些模块的用法。

下一步，我将讲述 16 位任务的问题和 KRNL386 用来维护这些任务的数据结构。（任务是从模块生成的，自然是先讲模块，再讲任务。）如果你对 Windows 95 为 32 位的 EXE 和 DLL 生成 16 位模块还不感到奇怪，你至少也要惊讶 Windows 95 居然为每个 Win32 进程维持着一个 16 位的任务。讲述完任务的特征后，我将给出一些操作和使用任务信息的 KRNL386 函数的伪码。

在本章末“SHOW16 程序”一节中，讨论了一个 16 位 SHOW16.EXE 程序，我写这个程序的目的是使你能容易地浏览系统中的模块和任务。虽然我本来可以使用 TOOLHELP 来获得需要的信息，但最后还是选用了从模块和任务中直接取得数据的方法。这样做还可以证明这些模块和任务不是什么只有微软公司的程序员才能接触的神秘事物。用 SHOW16 浏览到的现象可能会使你惊讶不已。

在深入讨论这些模块和任务的细节之前，还需要解释一点。在整个本章中，我常把全局内存句柄（global memory handles）和 CPU 选择器（CPU selector）作为同样含义的术语使用。在 Windows 3.1 和 Windows 95 中，16 位固定全局堆句柄是一个 ring 3 选择器的值。通过打开句柄值的最低位，一个可移动句柄可以容易地转化

成一个这样的选择器。这其实就是 GlobalLock 所完成的一切。这一点在这里交代清楚，以后在其它内容中，对选择器和全局内存句柄之间微小的差异不再作说明。在本章中，把它们作为同样的概念就可以了。

为什么 32 位的模块和进程要有 16 位的表示方法？

你可能惊讶于 Microsoft 为什么要不厌其烦地把 32 位的程序和 DLL 以对应的老式 16 位程序和 DLL 表示。答案很简单。和 Windows NT 不同，Windows 95 并不限制 16 位应用程序只能使用自己对应的虚拟机，将其从 32 位世界里分离出来。正相反，16 位和 32 位程序共存于同一虚拟机内，甚至在一定程度上共享地址空间（详细内容请参看第五章）。而且，被所有 Windows 95 应用程序使用的大部分代码都包括 16 位 DLL 中（例如，USER，GDI，COMMDLG，甚至还有 KRNL386）。

16 位模块

在 Windows 95 的 16 位世界中，模块（module）是一种数据结构，KRNL386 用它来表示代码、数据和装入内存中的 EXE 和 DLL 资源。归为 DLL 类的文件有不同的扩展名，如. DRV 和. FON。每个模块都与系统中一个磁盘文件相关联。

在这一节，我讲述 Windows 95 的一些 16 位模块，它们均出自 Windows 3.1 的 16 位模块。系统中装入的任何 EXE 和 DLL，不管它是基于 16 位还是 32 位的，都有一个 16 位的模块。然而，32 位 EXE 和 DLL 同时也在 32 位领域中被 KERNEL32. DLL 表示成 32 位模块，弄清这一点很重要。一般来说，16 位表示的模块被 16 位的系统 DLL 使用，而 32 位的模块被 32 位系统 DLL 使用（例如，KERNEL32）。关于 32 位模块更多的信息，请参看第三章的内容。

所有 16 位模块的数据都保存在调用 GlobalAlloc 从 16 位全局堆分配的一个段中。这个保存模块信息的段称作模块数据库（module database）。包含模块数据库的全局堆块的句柄称为模块句柄（module handle），更熟悉的名字是 HMODULE。这就是象 GetModuleHandle 这样的函数所指的句柄。

在 Windows 3.1 中，所有的模块都是在 LoadModule 例程中建立的。对 LoadLibrary 或 WinExec API 函数的调用，实际上最终都调用了 LoadModule。在 Windows 95 中，用于 16 位 EXE 和 DLL 的模块仍在 KRNL386 的 LoadModule 函数中建立，而代表 32 位 EXE 和 DLL 的 16 位模块由 KERNEL32. DLL 建立。用于基于 32 位的新可执行（NEW Executable）模块的选择器不在全局堆的句柄列表中，所以找到这些 HMODULE 需要一些技巧。（后面将看到）

16 位模块数据库格式基于 Windows 和 OS/2 1. x 版本开始时使用的 16 位可执行程序格式。这个文件格式称作新可执行格式（NE，即 the New Executable format）。在本章后面的内容中，我把 Windows 95 中的 16 位模块称作 NE 模块，以区别于 32 位模块（它们基于兼容可执行格式，英文为 the Portable Executable format，

即 PE 模块)。本章不讲 NE 文件的格式,因为在微软公司的文档和其它地方有足够多的资料讲解这个问题。下边几节讲模块数据库的格式,它通过读 NE 文件而建立。如果你把 NE 文件格式和 NE 模块数据库格式作一比较,就会发现,尽管它们非常地相似,但还是有一些非常重要的差别。

在每个 NE 文件开始都有一个 0x40 字节长的数据结构,称作 NE 头。这个结构之所以被赋予这样一个名字,是由于它的第一个字(WORD)包含的值为 0x454E,作为 ASCII 码显示就是 NE(New Executable 的缩写)。在 LoadModule 的代码中,KRNL386 从可执行文件的开始读出 NE 头,存进它建立的模块数据库的开头。NE 文件的许多域被识别为对应于 NE 模块在内存中的偏移量。然而,KRNL386 把许多域处理为只对磁盘文件的 NE 头有含义,把它们用于其它目的。

模块数据库中,跟在 0x40 字节的 NE 头后面的是段表(the Segment Table)。段表是一个结构数组,包含着关于每个模块的代码段和数据段的极其重要的信息(如大小、代码、数据等)。在段表后是资源表(the resource table),它包含了在相应的 NE 文件中可以找到的所有资源,不过它不是这些资源本身。确切地说,资源表是这样一类表,其内容告诉 KRNL386 在 NE 文件的什么地方可以找到实际的资源数据。

在段和资源信息后面,是模块数据库的输入输出信息。调用另一个 EXE 或 DLL 中的函数称作引入该函数(importing the function)。例如,USER. EXE 调用了 KRNL386. EXE 中的函数,所以 USER. EXE 引入 KRNL386. EXE 及其函数。那么反过来便称作引出函数(exporting a function)。引出函数的意思是使函数可以被其它 EXE 和 DLL 调用。我刚说的例子中,KRNL386 引出它的函数,USER. EXE 引入它们。

在位和字节的层次上来看,引出一个函数就意味着把它的地址放进一个称作入口表(entry table)的表中。当你装入一个 NE 文件,而它从另一个模块中引入函数时,Windows 装入程序(也就是 LoadModule 函数)就在目标模块的入口表中查找引出函数的地址。装入程序怎么能知道使用入口表的哪一个槽呢?当你引出一个函数时,连接器赋给它一个顺序号,这个顺序号可以看成模块入口表的索引。在要引入函数的 EXE 或 DLL 中,存储有要引入的函数在模块入口表中的顺序号。

也可以不用入口表顺序号,而用名字引用一个函数。这就要用到驻留和非驻留名表。这两个表将函数名和模块中引出函数地址相关联。模块数据库段中包含了整个的驻留名表,但只为非驻留名表提供一个文件偏移量(这就是驻留和非驻留的区别)。

在 Windows 3. x 下,KRNL386 把模块数据库保存在一个链表中,当新模块被 LoadModule 建立后,它们就被加入到该链表尾部。表的开头是 KRNL386,这是第一个被装入系统的模块。你自己就可以很容易地浏览这个模块列表(就象 SHOW16 程序所做的),或者也可以用 TOOLHELP 的 ModuleFirst 和 ModuleNext 函数完成浏览。(从内部看,TOOLHELP 和 SHOW16 做同样的事,只不过它是由微软公司批准的,而我们的 SHOW16 自己浏览系统数据结构没有被正式批准。)

在 Windows 95 下,KRNL386 用同样的方式维护用于 16 位 EXE 和 DLL 的

NE 模块。然而，表示 32 位 PE 文件的 16 位模块数据库不加入到该链表中，它们只是挂在全局堆中，与 16 位模块没有联系，相互之间也没有联系。使用 16 位代码，我没有找到任何更好的方式可以列举 32 位 NE 模块。然而，用一种很原始、很低层的方法可以完成这一工作，SHOW16 将演示这一过程。

在深入分析其实际格式之前，让我们先从高层快速浏览一遍 16 位模块数据库的部件：如图 7-1 所示，0x40 字节的 NE 头位于模块数据库的开头。其后是段表和资源表。尾部是关于引入模块（引入名表(imported names table)和模块引用表(Module reference table)）和引出函数（入口表(the entry table)和驻留名表(resident names table)）的表。

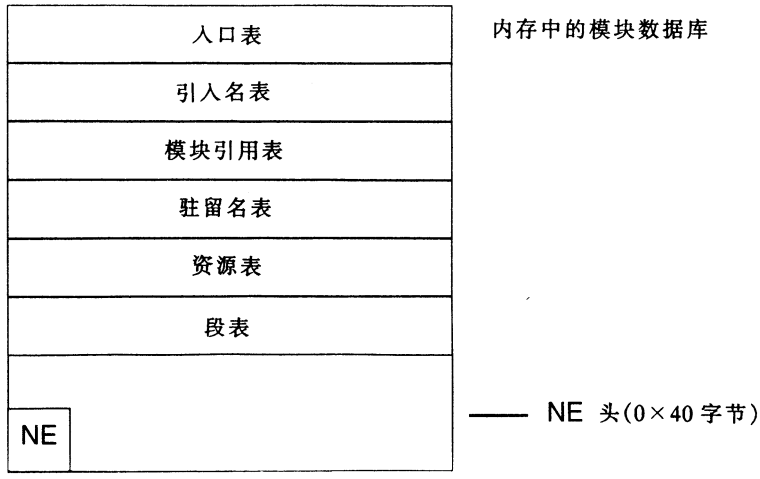


图 7-1　16 位模块数据库的各个部件

NE 头

如果你只想快速浏览一下模块数据库的各域，可以看 SHOW16 样本程序的 HMODULE.H 头文件（在随书所附的磁盘上）。这里，我想详细介绍 0x40 字节的 NE 头的各域。对每个域有段描述，每段描述的第一行给出它在模块中的偏移量，类型（如 WORD 或 DWORD）和一个简短描述。

00h　WORD　　识别码

这个 WORD 通常包含值 0x454E，表示 ASCII 字符 NE（即 New Executable）。在可执行文件开头放一个识别码是微软和 IBM 的操作系统的传统格式。其他用于可执行文件的识别码有 PE（即 the Win32 Portable Executable format），LE（Linear Executables，用于 Windows 3.x 和 Windows 95 的 VxD），还有 LX（也是 Linear Executable，但用于 32 位 OS/2 2.x 程序）。

02h　　WORD　　模块引用计数器

这个 WORD 表示使用该模块的其他模块的数目。例如，一个 DLL 被三个程序使用，则它的这个域值为 3。如果你通过 LoadLibrary 函数装入了一个 DLL，那么在这个 DLL 的模块数据库中本域值为 1。以后每次对这个 DLL 调用 LoadLibrary 或 LoadModule 都将为本域值加 1，每次调用 FreeLibrary 则为本域值减 1。但是，确定本域值的规则并不总是这么简单。例如，如果一个程序使用了两个 DLL（假设分别称为 A 和 B），这两个 DLL 都引用了第三个 DLL（C），那么模块 C 的引用计数器值为 1，而不是 2。我于 1994 年 5 月在《Microsoft Systems Journal》上发表的文章《Q&A column》（在 MSDN CD-ROM 上边有）中讲述了 IncExeUsage 和 DecExeUsage，这两个 KRNL386 的核心函数用于为模块引用计数器增值或减值，包括含有循环引用 DLL 的这样困难的地方。在为 PE 文件建立的 NE 模块中，模块引用计数器初始化为 1，而且似乎从来不变。

当模块的引用计数器值降到 0 时，KRNL386 就释放掉模块的段和资源，如果该模块是一个 DLL 模块，还需调用 WEP 例程（如果有 WEP 例程的话），最后，用 GlobalFree 释放模块数据库段。

04h　　WORD　　指向入口表的近指针

这个域是一个近指针（相对于 HMODULE 段），指向模块入口表。这个入口表包含了函数地址、它的引入序号和一些标志。有关入口表的详细内容请参看“入口表”一节。为 Win32 可执行文件建立的 NE 模块不包含入口表，因为 PE 模块不支持入口表约定的 16 位远地址。

06h　　HMODULE　　下一个模块数据库

这个大小与 WORD 相同的域中存放的是 NE 模块链表中下一个模块的 HMODULE。KERNEL 模块（KRNL386.EXE）总是在表的开头。有两种方式可以得到 KERNEL 的 HMODULE：可以调用 GetModuleHandle（KERNEL），也可以对任何模块调用 GetModuleHandle，函数返回时，KERNEL 的 HMODULE 就放在 DX 寄存器中。新模块被装入时，它们被追加到链表末尾。表中最后一个模块本域值为 0。Win32 文件的 NE 模块不在模块链表中，在这个位置上，值均为 0。

08h　　WORD　　指向 DGROUP 段入口的近指针

这个域是个近指针（相对于 HMODULE 段），指向模块的 DGROUP 段的段表入口。段表的格式在下一节讨论，题目为“段表”。DGROUP 段是数据段，缺省情况下，模块所有的专有数据都放在这里。DGROUP 段常常还包含一个局部堆，在 EXE 模块中为程序的栈。为 Win32 EXE 和 DLL 建立的 16 位模块中本域值为 0。

0Ah　　WORD　　近指针，指向修改过的 OFSTRUCT（它含有文件名）

这个域是个近指针(相对于 HMODULE 段),指向一个数据结构,该结构与 Win16 WINDOWS.H 文件中给出的 OFSTRUCT 非常相似。在 Windows 95 DDK 中,16 位 WINDOWS.H 文件把这个结构称为 OFSTRUCTEX。

```
typedef struct tagOFSTRUCTEX
{
    WORD cBytes;            // The length of the struct, in bytes.
    BYTE fFixedDisk;        // TRUE if nonremoveable media.
    UINT nErrCode;          // DOS error code if OpenFile failed.
    DWORD fileDateTime;     // Date/Time of file in MS-DOS format.
    char szPathName[260];   // The path to the file.
} OFSTRUCTEX;
```

这个结构和常规的 OFSTRUCT 之间的主要区别是 cByte 域是一个 WORD,而不是一个 BYTE。为什么要这样?因为 Windows 95 支持长文件名(长达 260 个字符)。因此,结构中的一个 BYTE 不能包含结构的整个长度。另外一个区别是,结构尾部的数组(包含路径名的域)是 260 字节,而不是 OFSTRUCT 中的 128 个字节。

0Ch　　WORD　　模块状态标志

这个 WORD 包含一些位域标志,表示该模块在内存中的状态信息。这些标志的含义有别于磁盘上对应的 NE 文件使用的标志。已知的 Windows 95 NE 模块的标志如下:

标志名和位值	含义
MODFLAGS_DLL 0x8000	对于 16 位 NE 模块,这个标志说明该模块是一个 DLL,而不是 EXE。对表示 Win32 模块的 NE 模块数据库,这个标志总是被置位。
MODFLAGS_CALL_WEP 0x4000	这个标志只对 DLL 有效,说明当 DLL 未装载时,要调用 DLL 的 WEP 例程。除了任务模块和 Win32 模块,这个标志总是被置位。
MODFLAGS_ SELF_ LOADING 0x0800	这个标志说明该模块使用自装入机制。在这种机制下,模块提供自己的段装入程序,LoadModule 调用它把模块的段装入内存。微软极力阻止人们使用自装入程序,且很少公布它们的用法。然而,在过去,一些微软的应用程序(象早期的 Word for Windows 和 Microsoft Fortran)使用了自装入的特征。SLR 系统中的 Optlink 5.x(现在由 Symantec 所有)可以生成使用自装入机制压缩可执行文件长度的 EXE。当 OPTLINK 连接器把段表数据写到 NE 文件中时,它压缩信息;当该模块被调入内存时,内部自装载代码把段数据解压缩并把它装入到内存中。

MODFLAGS_APPTYPE 0x0300 (0x0100|0x0200)	这两个位是 OS/2 1.x 时代的遗物。在 OS/2 中，程序的用户界面可以是下面三个可能类型中的一个。值为 0x0300 说明程序使用操作系统的 GUI windowing API。值为 0x0200 说明应用程序是一个控制台(文本模式)应用，但其屏幕输出只能是可以在 GUI 窗口中虚拟显示的文本模式输出(一个这样的例子是在 Windows 状态下运行 MS-DOS 提示)。位值为 0x0100 表示应用程序直接操作显示缓存，这时它必须运行在全屏幕模式下。常规的 Windows NE 模块该标志值总是 0x0300，表示它们使用 GUI API。在 Windows 95 中，Win32 文件的 NE 模块不设置该值，这意味着域值为 0，根据 NE 的规定，这个值未定义。
MODFLAGS_IMPLICIT_LOAD 0x0040	这个标志说明该模块现在内存中，因为其他模块对它有隐含的连接。任务模块没有这个标志，通过 LoadLibrary 装入的 DLL 也没有这个标志。然而，如果一个 DLL 通过 LoadLibrary 装入时，隐含地装入了其他一些 DLL，则那些 DLL 的模块数据库中的本标志被置位。
MODFLAGS_WIN32 0x0010	这个 Windows 95 的新标志说明这个 NE 模块表示一个 Win32 PE 文件。
MODFLAGS_AUTODATA 0x0002	这个标志告诉 16 位装入程序，每个模块对每一个用户应有一个独立的 DGROUP 实例。这个标志是针对 EXE 而言的，对 EXE 程序的每个运行实例都需要有一个自己的 DGROUP 段
MODFLAGS_SINGLEDATA 0x0001	这个标志说明本模块的所有用户共用一个 DGROUP，只在 DLL 模块中本标志才置位，因为对 16 位 DLL 来讲，不管被那个任务调用，它都使用同一个 DGROUP。如果 MODFLAGS_AUTODATA 和 MODFLAGS_SINGLEDATA 都没有被置位，那么该模块没有 DGROUP 段或局部堆。有趣的是，SYSTEM 模块(在 KERNEL 后直接装入)就属于这一类。Win32 文件的 NE 模块的模块数据库标志字值总是 0x8010，它被解释成 MODFLAGS_DLL 和 MODFLAGS_WIN32。

0Eh　　WORD　　DGROUP 段的索引

这个域包含模块的 DGROUP 段的段表的索引，索引号从 1 开始。这个域有些多余，因为在模块数据库偏移量为 08h 处的近指针提供了相同的信息(显然是不同的格式)。在 Win32 NE 模块中，这个域的值总是 0。

10h　　WORD　　初始局部堆大小

这个 WORD 是模块的 DGROUP 段的初始内存数目，是 Windows 装入程序时应当为其局部堆保留的内存。以后局部堆可以变大，如果局部堆变大了，这个域的值并不更新(这个域的值看来没打算修改，因为启动程序的另一个实例时需要用到局部堆大小这个数据)。许多标准 16 位系统 DLL 的初始堆大小为 0。有趣的是，在 Windows 95 中，有两个系统 DLL 的初始堆大小与 Windows 3.1 中同样的 DLL 相比反而缩小了。这可能是为尽可能缩小 Windows 95 占有内存所作努力的一部

分。在 Win32 NE 模块中，本域的值为 0。

12h　WORD　栈大小

这个 WORD 中的值是装入程序在模块的 DGROUP 段中为栈保留的空间的大小。因为 DLL 代码使用调用它的应用程序的栈，所以栈大小只对 EXE 模块起作用。在 Windows 3.x 和 Windows 95 中，16 位应用程序的栈最小为 5K。如果 EXE 文件中本域的值小于 5K，装入程序生成 NE 模块时自动把它的值增加到 5K。在 Win32 NE 模块中，本域值总为 0。

14h　FARPROC　模块入口点

模块表结构的这个成员包含了模块的入口点。对 EXE 模块，入口点就是程序开始运行的地方。在用 C 或 C++编译的 EXE 中，入口点就是编译器的运行库启动代码开始的地方。C/C++启动代码最终调用 WinMain 过程。在 DLL 模块中，入口点是运行库代码开始的地方，它最终调用 LibMain 过程。EXE 模块的入口点不能为 0；DLL 模块的入口点可以为 0，这时装入程序实际上不执行任何调用。

本域中存储的地址是一个逻辑地址。逻辑地址都是 16:16 地址，但段部分并不是真正的选择器值。这个段值只是 NE 头后面的段表的索引。因此，如果模块入口点是模块中第三个段的 0x017C 字节处，则入口点值为 0003:017C。装入程序调用模块的入口点时，要用逻辑段值作为索引从 NE 头后面的段数组中取出实际段值。

在 Win32 NE 模块中，入口点总是 0。原因很简单，因为 32 位代码使用 32 位偏移量，而不使用 16:16 远指针。

18h　DWORD　初始栈指针值

这个域的值是入口点被调用时，EXE 模块的 SS:SP 中的初始值。和前边那个域相似，这个地址的逻辑段部分也是一个逻辑地址，而不是物理的“选择器：偏移量”形式的值。这个地址的逻辑段部分总是和偏移量 0Eh 处的 WORD 中给出的 DGROUP 的索引一样。对于 DLL 和所有的 Win32 NE 模块，本域值总是 0。

1Ch　WORD　模块中段的数目

这个 WORD 中存放模块中段（不管是代码段还是数据段）的数目。模块表中 NE 头后面是一个段表入口的数组，每个入口为 10 字节。该数组中入口的数目由这个域的值给出。有时模块段数目可能是 0；最典型的例子是字体模块（Font module）。典型的字体模块只包含资源，没有段。对于基于 PE 文件的 NE 模块，本域值总是 0。

1Eh　WORD　引入的模块的数目

这个域包含了该模块隐含连接的模块数。例如，如果一个 EXE 调用了以下模块中的函数：KERNEL，USER，GDI，则它的本域值为 3。这个域用来和模块引用表

(见域 28h)相联系。模块引用表中的入口的数目对应本域中值。(我刚讲的例子中,有三个入口,对应本域值为 3。)Win32 NE 模块的本域值总是 0。

20h　WORD　非驻留名表的大小

这个域中包含的值是 NE 文件中非驻留名表的大小。非驻留名表(域 26h 指向驻留名表)和一个带引出顺序号的引出符号(通常是函数名)相联系。要访问非驻留名表,KERNEL 需找到 NE 文件中起始处的偏移量(见域 2Ch),然后读出与本域值相等的字节数(更详细的信息请参看"驻留名表和非驻留名表"一节)。在 Win32 NE 文件中,本域值总是 0。

22h　WORD　指向段表的近指针

这个域是一个近指针(相对于 HMODULE 段),指向模块的段表。段表是一个 10 字节结构的数组,对模块管理的每个代码段和数据段有一个元素。详细信息见下一节。)因为段表总是紧跟在 0x40 字节的 NE 头后面,所以一般 NE 模块中本域值总是 0x40。对 Win32 NE 模块,本域值总是 0x4C——但这没什么含义,因为 Win32 文件不使用 16 位段或段表。

24h　WORD　指向资源表的近指针

这个域是个近指针(相对于 HMODULE 段),指向模块的资源表。资源表是一种实际资源数据的"目录表",这些资源数据存放在可执行文件的其他地方。(资源表的格式在"资源表"一节讲述)有趣的是,16 位和 32 位 NE 模块都频繁地使用本域。这意味着使用资源的 16 位代码常常同时处理 16 位 NE 文件和 32 位 NE 文件中的资源。

26h　WORD　指向驻留名表的近指针

这个域是一个近指针(相对于 HMODULE 段),指向模块的驻留名表。驻留名表用来把从一个模块中引出的函数名和引出顺序号对应起来。驻留名表的格式和非驻留名表相同。这两个名表的关键区别在于驻留名表总在内存中(在 HMODULE 段内),而非驻留名表每次需要时都从磁盘上装载。这两种名表的格式在"驻留与非驻留名表"一节中讲述。

所有的 NE 模块,无论是从 NE 文件生成的还是从 PE 文件生成的,都有一个驻留名表,原因是每个模块必须有一个名字(例如,KERNEL,USER,TOOLHELP,等等)。在驻留名表中,模块名总是第一个入口。因此,当 KERNEL 建立它的最小的 NE 模块数据库时,总要包含一个只有一个入口的驻留名表,该入口就是这个模块自身的名字(即 KERNEL32)。

28h　WORD　指向模块引用表的近指针

本域值是个近指针(相对于 HMODULE 段),指向模块引用表。模块引用表是被这个模块使用的(就是被引入的)所有模块的列表。

这个列表只不过是一个 HMODULE 数组。在当前可执行程序的重定位信息中，你可以发现，每个引入的函数的重定位信息包含一个本模块表的索引。例如，一个程序调用了 GDI.EXE 中的 SetPixel。SetPixel 从 GDI.EXE 引出的顺序号为 31。在当前程序的模块表中，GDI 是第四个模块，因此，这时对 SetPixel 调用的重定位信息包含两个值 4 和 31(4 是 GDI 的 HMODULE 的模块引用表索引值，31 是 SetPixel 在 GDI 中的引出顺序号)。

2Ah　　WORD　　指向引入名表的近指针

本域是一个近指针(相对于 HMODULE 段)，指向模块的引入名表(imported names table)。引入名表是一系列 PASCAL 风格的字符串，它们是被本模块引入的所有 DLL 的模块名。引入名表也可以包含函数名，这样的函数用名字而不是用顺序号引入。建立模块数据库时，KRNL386 根据引入名表中的模块名搜索和装载被本模块引用的其他模块。KRNL386 每查到或装载一个引入模块，就把被装入模块的 HMODULE 存入模块引用表(域 28h)。一旦建立了一个模块，Windows 其实就不再使用引入名表。在 Win32 NE 模块中，这个域是个非零值，但该值没有意义，因为在这些模块数据库中没有引入名表。

2Ch　　DWORD　　非驻留名表的文件偏移量

本域中值是非驻留名表在 NE 文件中的偏移量(按字节计)。这个值和域 20h 一起使用，在需要时把非驻留名表装入内存。在 Win32 NE 模块中，这个域总是 0。更多的信息见域 20h 和本章后面就是的“入口表”一节的内容。

30h　　DWORD　　入口表中可移动入口的数目

随着实模式 Windows 的消亡，本域其实已经废弃了。入口表中有些入口含有的地址随实模式下段的移动而改变，这样的入口的数目就是本域的值。在保护模式的 Windows 中，选择器和描述器隐藏了段在内存中的移动，所以可移动入口就不再有用了。入口表中可移动入口和固定入口在“入口表”一节讲述。

32h　　WORD　　调整移位

在 NE 模块中，实际段数据和段资源的文件偏移量不是以 DWORD 偏移量(你所期望的格式)存储的。段和资源在 NE 文件中的位置是按“扇区”计算的。NE 模块中的扇区不等于磁盘扇区。它总是 2 的幂(2,4,8,16,32,依此类推)。要确定一个给定 NE 模块的扇区大小，可以用本域中给出的值作为 2 的指数求出结果。另外一种方法是，设初值为 1，本域中值为几，就将该值左移几次，得到结果。

对 NE 模块，本域的一个典型值是 9。(1<<9)==2^9==512，说明扇区大小为 512 字节。如果一个段在文件中偏移量为 1536 处，则它的位置用扇区给出就是 3。本域另一个常见值是 4，(1<<4)=16 字节。当连接一个 NE 文件时，可以为它设置扇区大小。对为 PE 文件生成的 NE 模块，这个调整值总为 1。一般来说，应使用尽可能小的调整值连接文件。如果你使用了比所需大的调整值，就会浪费磁

盘文件的空间。因为这时连接器必须加入额外空间，以保证每个段和资源开始处的文件偏移量是扇区大小的整数倍。段和资源的扇区偏移量是用 WORD 存储的，所以使用 16 字节扇区的最大文件长度为 1M(65535 扇区乘以 16 字节/扇区＝＝1MB)。如果使用 512 字节调整值(这是大多数连接器的缺省值)，最大文件长度为 32MB。

34h　WORD　未知

在 Windows 3.1 中，如果模块中包含 TrueType 字体，则本域值为 2。在 Windows95 中，本域可能不用了，其值总是 0。

36h　BYTE　所需操作系统

这个 BYTE 包含的值，表示本模块运行所需的操作系统。已知值如下：

0　未知(尽管 Windows 1.0 文件使用了这个值)
1　OS/2
2　Windows
3　欧洲 DOS4(DOS 的一个多任务版本，未在美国发行)
4　Windows/386(只在 Windows 2.x 时代存在)

一般来说，除非你在 16 位 OS/2 下工作，在 NE 文件中很少遇到除 2 以外的值。在 Win32 NE 模块中，本域值总是 0。

37h　BYTE　其他模块标志

这个 BYTE 包含一些 Windows 1.x 之后加入到模块数据库中格式的一些标志。(否则，它们可能会出现在偏移量 0Ch 处 WORD 里的标志中，或 0Ch 域不得不扩展成 DWORD。)

在 Windows 3.x 中，本域值为 0x02 和 0x04 说明本模块是为 Windows 2.x 写的，但经检查，可以在 Windows 3.x 下正常运行。因为 Windows 95 不支持运行任何 2.x 或更早的应用程序，所以这些标志值实际上被废弃了。

标志值为 0x08 说明本 NE 文件有一个快速装入区。快速装入区是一组段和资源，它们已经被集中在一块连续的区域中，Windows 装入程序可以用一次读操作把这些段装入内存，而不必为每个段、每个资源去单独定位和读写文件。其目的是在初始装入模块时节省时间。

在 Windows 95 中，一些 16 位模块里出现了一个新的标志值(0x10)。如果这个标志被置位，KRNL386 就不再查找和调用 16 位 DLL 中的 DLLENTRYPOINT 函数。Win32 NE 模块的本域值总是 0。Windows 95 下的 Win16 DLL 的新 DLLENTRYPOINT 功能在微软公司关于形实转换程序编译器(THUNK.EXE)的文档中有说明。

38h　WORD　指向引入名表的近指针

这个域总是指向驻留名表，且总和域 2Ah 的值相同。

3Ah　WORD　指向驻留名表的近指针

这个域总是指向驻留名表，且总是与域 2Ah 和 38h 的值相同。这一规则的例外是对第一个模块：KERNEL。这个例外不会引起问题，因为 KRNL386 是由一段与 Windows 3.1 中的常规 Windows 装入程序不同的代码装载的，且这个装载程序在 Windows 95 中仍在使用。

3Ch　WORD　未知

这个域值的含义尚不清楚。但它似乎总是 0x10 的整数倍。一个经常出现的异常情况是，在 16 位 NE 模块列表的每个子模块中该值升高。Win32 NE 模块中该域值总是 0。

3Eh　WORD　期望的 Windows 版本

这个域中是使用本模块所需的 Windows 的最低版本。常见值为 Windows 3.0 (0x0300)，Windows 3.1 (0x0310)，Windows 95 (0x0400)，和 Windows NT3.5 (0x0350)。这个 WORD 的高字是 Windows 的主版本号，低字节是小版本号(即小数部分)。显示版本号用的正确的 printf()格式串为：

%u.%02u

随书所附磁盘上的 SHOW16 程序可以作演示。

Windows 95 中的模块数据库新域

下边三个域在 Windows 95 中是新加的，它们只存在于基于 PE 文件的 NE 模块中。(也就是，NE 头中的标志域 0x0010 位被置位)。如果模块数据库是从 NE 文件生成的，则这些域不存在，在相应的位置上是段表的第一个入口。

40h　DWORD base　相关联的 PE 文件的地址

这个 DWORD 是相对于 Windows 95 的 32 位部分装入 PE 文件的内存位置的虚拟地址(展开的 32 位指针)。从 32 位程序可以看到，其值与 32 位 PE 文件的 HMODULE 的值相同。

44h　DWORD　相关联的 PE 文件的基地址

这个域总是和上一个 DWORD(40h)的值相同。

48h　DWORD　在内存映射的PE文件中的资源部分的基地址

这个 DWORD 中包含与这个 16 位 HMODULE 相关的 PE 文件的资源部分的 32 位线性地址。在后面“资源表”一节中，将看到 Windows 95 的 16 位部件中有 32 位 PE 文件中资源的信息。

段表

在模块数据库中，紧跟在 0x40 字节的 NE 头后面的是段表(Win32 文件的 NE 模块没有段表)。段表(the Segment Table)是一个结构数组，每个结构变量表述一个代码段或数据段的特征。每个结构的前八个字节和 NE 文件的段表相同，额外的 WORD 用来存放 16 位 Windows 装载程序分配给那个段的选择器。这一点非常重要：KRNL386 一旦把一个 NE 文件中的段装入内存，它总能把该段和一个用来访问它的选择器相关联反之亦然。

每个段表入口的格式如下：

00h　WORD　在 NE 文件中的扇区偏移数

这个 WORD 中包含一个 NE 文件中的位置，在该位置处可以找到该段的原始数据。这不是一个以字节计的偏移量，这个偏移量的单位是扇区。各个文件的扇区大小不一样，它可以根据 NE 头中偏移量为 32h 处的调整移位值计算得到。扇区大小的典型值为 16 字节和 512 字节。如果本域值为 0，那么这个段是尚未初始化的数据段，且在 NE 文件中该段没有原始数据。

02h　WORD　文件中段的长度

这个 WORD 中是 NE 文件中该段数据的长度。注意，这不是 KRNL386 分配用来装入该段的内存块的大小。要知道段在内存中的大小，需看域 06h 中的值。为什么数据段在 NE 文件中的长度和它们在内存中的长度有可能不一样？最一般的原因是有些数据段的尾部含有未初始化的数据。例如，假设你有 3K 的实际数据，但还需要一个 4K 的未初始化数据块(比如一个数组)，那么，在段表中本段的入口处，本域值为 3K，06h 处的分配长度域值为 7K。

04h　WORD　状态标志

这个 WORD 包含着关于段的信息的标志。下表列出的标志的含义和 NE 文件的说明给出的相匹配。然而，如果你检查内存中模块数据库的标志，就会发现 KRNL386 打开了一些在 NE 文件中没有说明的位。已知的标志如下：

标志和位值	含义
DATA 0x0001	本段是一个数据段，如果本标志没有置位，则是代码段
ITERATED 0x0008	段中包含重复数据

MOVEABLE 0x0010	段在线性内存中是可移动的。 如果这个标志未被置位，则该段是固定的。Windows 装载程序一般关闭 EXE 文件中的该位，因为 EXE 文件极少需要固定内存
DRELOAD 0x0040	此段应在模块装入后被装入，而不是当第一次被访问时被装入。
RELOC 0x0100	段中包含内存中紧跟原始段数据后面的重定位信息
DISCARDABLE 0x1000	该段是可丢弃的。如果内存供应紧张，KRNL386 可以把该段的描述标志为未提供，把它占用的 RAM 提供给其他任务、程序使用
32BIT 0x2000	该段是 32 位代码段，当装载程序为该段分配选择器时，将把描述中的"big"位置位，这样该段将被解释为 32 位代码

06h　　WORD　　分配大小

这是当 KRNL386 把该段装入内存时应为它分配的内存块的大小。它可能比 NE 文件中该段的原始数据的数量大。更多的信息请参看域 02h 的说明。

08h　　WORD　　全局内存句柄

这是 KRNL386 为把该段数据装入内存而为它分配的内存块的全局堆内存句柄。如果句柄以 06h 或 0Eh 结尾(例如，0476h，047Eh)，则该段是一个可移动段。反过来，如果句柄以 07h 或 0Fh 结尾，则该段为固定段。

段表中入口的顺序非常重要，因为它是逻辑地址的基础。当象连接器、调试器这样的程序需要使用在模块的段中的地址时，需要通过逻辑地址来完成这一工作，而不是通过实际的选择器和偏移量。因为 Windows 用来存放模块的段的选择器对每次装载来说都不一样，所以它们不能使用实际的选择器值。与使用选择器值不同，逻辑地址使用段表的以 1 开始的索引号来描述指的是哪个段。段表数组中的第一个段是逻辑段 1，第二个段是逻辑段 2，依此类推。如果你观察连接器产生的.MAP 文件中函数的地址，就能看到逻辑地址。如下面这样：

```
0001:5F46       _free
0001:5F5C       __GetSubAllocClientData
0002:0030       _errno
0002:0032       __protected
```

一个模块中最多可以有 253 个段。这是因为入口表(后面专门有一节讲它)把引出函数的地址存储为逻辑地址，而且只用一个字节存储逻辑段号。逻辑段号 0、0FEh、0FFh 已被 Windows 装载程序留作他用，所以一个 NE 模块中段的最大数目是 253，而不是 256。

资源表(The Resource Table)

除了段的信息,每个模块数据库也包含了所有在可执行文件范围内的资源(如图标、点阵图等)的位置和属性。和一些程序员想象的不同,资源并不计入模块段表中的段,你可以拥有超过 255 个的资源。

通常情况下,资源表在模块数据库中紧跟在段表后面。与段表不同,资源表不是数组。它是自由表格式,要找到一个资源,必须作相当多的运算。模块数据库中资源的格式和相关的 NE 文件中的资源表非常相似。

资源表的第一个 WORD 是调整移位次数(扇区大小),用来计算相关的 NE 文件中的资源的偏移量。这个扇区大小的含义和在前面"NE 头"一节中域 32h 描述的 NE 扇区大小的含义相同。这个 WORD 中的扇区大小应当与前一节域 32h 指示的相匹配,否则,就是 NE 文件出错了。

在第一个字后面是一系列变长度的小节。每节存放一条关于某个特定类型的资源的信息。例如,USER.EXE 有以下小节:光标、图标、点阵图、菜单、对话框、字符串表和版本信息。每节中是一个数据结构的数组,每个数据结构对于一个特定的资源实例。例如,如果你有一个含有五个图标的 NE 文件,则就有一个关于图标的节,里面含有五个结构。

在内存中,每个这样的节都紧跟在前面一个资源的后面。因此,为找到一个资源的特定实例,需要计算出每个节的长度,这取决于其中包含的特定资源的实例的数目。随书所附的磁盘上有个 SHOW16 程序,如果对这些概念有些混淆,可以用该程序演示有关观察资源表的例子。每类资源的节(图标、点阵图等)都以下面给出的结构开头(参看 HMODULE.H 中 C 风格结构的定义):

00h	WORD	资源识别号

这是该资源的识别号。如果高位(0x8000)被置位,它就是预定义的资源。若屏蔽掉高位,资源的类型就由下面的值给出:

1 —— 光标(Cursor)
2 —— 点阵图(Bitmap)
3 —— 图标(Icon)
4 —— 菜单(Menu)
5 —— 对话框(Dialog)
6 —— 字符串表(String table)
7 —— 字体目录(Font directory)
8 —— 字体(Font)
9 —— 加速键(Accelerator)
10 —— RC 资源(RC data(user-defined data))
11 —— 错误表(Error table)

12 —— 光标组(Group cursor)
13 —— 未知
14 —— 图标组(Group icon)
15 —— 名表(Name table(went away in Windows 3.1))
16 —— 版本信息(Version info)

在 NE 文件格式说明中有对各种资源类型的完整描述。

如果资源识别号的高位没有置位,该资源就是用户自定义的命名资源。这时,识别号是一个偏移量(相对于 NE 模块中资源表的开头),指向资源类型名。这个名字是一个 PASCAL 风格的字符串。

02h　WORD　　这种类型资源的个数

这个 WORD 中是特定类型资源的实例数目。根据这个域的值就可以确定属于这种资源类型的信息段的大小。关于每个资源的信息就紧跟在资源表后面。

04h　DWORD　　资源处理函数

这个域是处理这种资源的函数的指针。处理函数负责在需要资源时,把它们装入内存。因为竞争对手能够作这样低层次的资源管理是微软公司所不希望的,所以关于资源处理函数的文档非常少。从关于 SetResourceHandler 和 LoadProc 的 SDK 文档多少可以得到点微软提供的信息。

特定资源类型的资源处理函数对每个模块来说可能不同,这取决于其 SetResourceHandler 函数。可是这个函数的文档中说它需要一个 HINSTANCE,而你所能得到的是一个 HMODULE,就把这个 HMODULE 传给它吧。这还有另外一个例子,关于微软怎样把 16 位 HINSTANCE 和 HMODULE 的含义弄混淆了。本章后面将更详细地讨论这个问题。

紧跟在每个资源类型头后面的是一个结构数组。对这种资源类型的每个实例有一个结构;每个结构长度为 12 字节。数组元素的个数在资源类型头偏移量为 02h 处的 WORD 中。每个数组元素都是下面的格式(见 HMODULE.H 中 C 风格的结构定义):

00h　WORD　　在 NE 文件中的偏移量

对基于 NE 文件的模块来说,这个域是这类资源的特定实例在 NE 文件中的偏移量,单位为扇区,而不是字节(关于 NE 文件扇区的详细信息在偏移量 32h 处的描述)。对基于 PE 文件的模块,这个域是一个 DWORD 的偏移量(相对于资源部分的开头)。该 DWORD 中的值是 PE 文件中 IMAGE_RESOURCE_DATA_ENTRY 结构的偏移量(相对于.rsrc 部分)。在 IMAGE_RESOURCE_DATA_ENTRY 中可以找到 PE 图像中原始资源数据的位置和长度。PE 文件个数的细节请参看第八章内容。如果 Windows 95 的 16 位部件要使用这个信息,它必须知道 PE 文件的.rsrc 部分位于内存的什么地方。怎样才能得到.rsrc 部分的基地址呢?很简

单，看看基于 PE 文件的 NE 模块的 NE 头中域 48h 的就明白了。

02h　　WORD　　长度

对基于 NE 文件的模块，这个 WORD 是资源的长度，以扇区为单位。对基于 PE 文件的模块，这个域是资源数据的实际长度，以字节为单位。这个域的值和 PE 文件的.rsrc 部分给出的 IMAGE_RESOURCE_DATA_ENTRY.Size 的值相等。

04h　　WORD　　状态标志

关于这个资源的标志。一般来说，这些标志和段标志（见本章前部“段表”一节的域 04h）相同。但是，KRNL386 还打开了其他的一些含义未知的位。已知的标志如下：

标志名和位值	含义
LOADED 0x0004	本资源现在已在内存中
MOVEABLE 0x0010	该段在线性内存中是可移动的，如果这个标志未置位，则该段是固定的
READONLY 0x0020	这个资源不能在内存中修改
PRELOAD 0x0040	装载模块时，也要同时装载这个段，而不是先装载模块，需要访问该段时才装入它
DISCARDABLE 0x1000	该段是可丢弃的。如果内存供应紧张，KRNL386 可以把该段的描述器标志为未提供，而把它所占用的 RAM 重新分配给其他模块或任务等使用

06h　　WORD　　识别号

这个字是资源编译器给予资源的识别号。如果高位（0x8000）被置位，说明这个资源是通过整数型的识别号引用；否则，该资源就是一个有名资源。这时，这个识别号是资源名的偏移量（相对于整个资源表开头）。其名字是 Pascal 风格的字符串。有名资源的典型例子是对话框。例如，在一个.RC 文件中有这样一句：

```
Show16Dlg DIALOG 8, 18, 360, 280
```

这个对话框的识别号就是 Pascal 风格的字符串 Show16Dlg 相对于资源表开头的偏移量。

08h　　WORD　　句柄

如果该资源已装入内存，这个 WORD 就是指向资源数据的全局内存句柄。如果资源未被装入，本域为 0。这个域和资源状态标志字（04h）的 LOADED 标志相关。如果 LOADED 标志未置位，本域为 0。

0Ah　　WORD　　使用计数器

这个字存放资源被引用的次数。本域的值通过 LockResource 函数增值，通过 FreeResource 函数减值。

入口表

NE 模块的入口表是它的引出函数被他模块使用的途径。在实模式 Windows 时代，入口表也充当可移动段中所有远函数的中心转换程序的角色。这里，我忽略掉入口表的那方面的功能，而只讲述它对引出函数的专有功能。

与段表和资源表不同，模块数据库中的入口表只不过是 NE 文件中入口表的等价转换(a passing resemblance)。NE 文件中的入口表常被优化以节省空间，而内存中的入口表则常被优化以加快扫描速度。象资源表一样，从最外层看，入口表由长度可变的一块数据构成，经过它需要作跳跃式计算(on-the-fly caculating)。

因为模块的函数的引出顺序号不一定非要是连续的，也不一定从 1 开始，所以入口表由一系列“包”(bundle)组成，每个包描述一个连续的引出顺序号范围。在入口表中查找一个指定函数的过程就成了扫描这些包，直到找到包含所找引出顺序号的包的过程。每个包都以下面描述的头作为开始：

00h　　WORD　　本包中第一个引出顺序号－1

这个 WORD 包含的值是包中第一个入口的引出顺序号减 1，例如，一个入口包中引出序号为 3 到 14，那么这个 WORD 中的值为 2。

02h　　WORD　　包中最后一个引出顺序号

这个域包含的值是包中描述的最后一个引出顺序号。例如，入口包中引出顺序号范围为 3 到 14，那么这个域中包含的值为 14。从这个域的值中减去偏移量 00h 处 WORD 中的值，结果就是后面跟随的函数入口点数据结构数组中的元素数目(入口序号为 3 到 14，包含 3 和 14)。

紧跟在包头后面的是一个数据结构的数组，每个结构对应一个引出函数。每个引出函数由对应结构中的信息来描述。结构是下面这样的：

00h　　BYTE　　段类型

如果本域值为 0FFh，这个段就是可移动的，并且如果程序在实模式下运行就需要一个特殊的转换程序。在保护模式下运行不需要转换程序，因为当数据结构在线性空间中移动时，代码段的选择器值不变。

如果本域值为 0FEh，说明这个入口是个特殊入口。这种类型的入口没有实际(远)地址。偏移量域(the offset field)被作为一种与本入口相连的代码中的全局变量使用。这类入口仅知的例子是 KRNL386 的一些输出值：__AHSHIFT，__0000H，等。对这些特殊入口的完整的列表和描述参见 Undocumented Windows。如

果本域值既不是 0FFh，也不是 0FEh，那它包含的就是引出函数地址的逻辑段号。这时，它的值应和结构中偏移量为 02h 的 BYTE 相同。

01h　　BYTE　　状态标志

本入口的状态标志。已知标志如下：

标志	含义
0x01	函数是引出函数，除了在需运行于实模式下的程序中，这个标志应总置位。如果未置位，说明该函数需要一个不能被其他模块引出使用的实模式转换程序。
0x02	本函数对所有的调用者使用同一个数据段。这应只发生在 DLL 模块中。缺省状态下，本标志对 EXE 模块是关闭的，对 DLL 模块是打开的。然而，你可以在. DEF 文件中对应的 EXPORTS 行上用 NODATA 语句把 DLL 引出函数的这个标志强行关闭。

02h　　BYTE　　逻辑段号

这个字节存放引出函数地址的逻辑段号部分。用这个段号作为 NE 模块段表的索引就可以确定实际选择器的值。

03h　　WORD　　偏移量

这个 WORD 是引出函数在上一个域(02h)给出的段中的起始偏移量。

要查找一给出的引出函数的地址(象 KRNL386 所做的那样)，就要扫描各包的包头，查找包含引出入口的包。找到后，就可以确定引出函数的数组索引，它相对于包中第一个数组入口。例如，使用这个常用例子，一个包中入口为 3 到 14，引出函数 7 的地址可以在包的第五个数组元素中得到。

在前面所有关于入口表的内容中，都没有提到函数名的问题。然而，GetProcAddress 函数使你可以在另一个模块中按指定的函数名查找引出函数的地址。这样，就必须有一种方法可以把函数名和它的引出顺序号联系起来，这就引出了下一节的问题：

驻留名表和非驻留名表

驻留名表和非驻留名表是把 NE 模块中的函数名和引出顺序号相联系的方式。这些表格式相同。表中每个入口都是下面这样：

偏移量	解释
01 BYTE	跟随在后面的引出名的长度
?? char	引出符号(函数)的名字，非 NULL 结尾的字符串
?? WORD	这个符号的引出顺序号

例如，GDI 中的 SetPixel 函数引出顺序号为 31，则在 GDI 的非驻留名表中存在下面的数据(8 表示“SetPixel”字符串长度，31 是引出顺序号)：

```
8, 'S', 'E', 'T', 'P', 'I', 'X', 'E', 'L', 31
```

驻留名表和非驻留名表的第一个入口有特殊的含义。二者中第一个入口的引出顺序号总为 0。在驻留名表中，第一个入口是本模块的模块名(例如：KERNEL，GDI，TOOLHELP，等等)。这个串和用于生成 NE 文件的.DEF 文件中 NAME 或 LIBRARY 行给出的字符串完全相同。

在非驻留名表中，第一个入口是描述域。这个字符串是该模块所完成功能的简述。连接器把用来生成 NE 文件的.DEF 文件中 DESCRIPTION 行的内容拷贝过来，作为这个字符串的内容。如果未提供 DESCRIPTION 行，连接器就缺省用 EXE 或 DLL 的名字作为该字符串的内容。Windows 95 的 KRNL386，USER，GDI 模块中典型的描述串为：

```
KRNL386: 'Microsoft Windows Kernel Interface Version 4.00'
USER:    'Microsoft Windows User Interface'
GDI:     'Microsoft Windows Graphics Device Interface'
```

为什么要用两个名表呢？唯一的理由是节省空间。大多数程序和 DLL 通过顺序号而不是通过名字来引用函数。因此，在引出函数的 DLL 中，没有必要在不需要这些名字时，还把它们整个地保留在内存中的模块数据库里。这些名字应该放在非驻留名表中，只在需要时(比如在执行 GetProcAddress 调用时)才装载它们。那些需要快速查找的名字，或在你不想执行磁盘操作的地方需要的名字，应当放进驻留名表。

可以用.DEF 文件控制把一个引出函数放进哪个名表中。如果你引出一个函数并在.DEF 文件中明确给出它的顺序号，则最终它被放入非驻留名表。但是，如果你没有在.DEF 文件中指定引出顺序号，连接器就会把它放进驻留名表。一般来说，如果你有一个含很多引出函数的 DLL，就应当用 TDUMP 或 EXEHDR 这样的程序把这些函数汇集输出到一个文件中，然后观察你的引出函数名在哪个表中。除非你有充分的理由把函数放在驻留名表中，一般总应做些工作，以确保使用的是非驻留名表。那样，你就不会让你的 DLL 函数的名字不知不觉地“吃掉”宝贵的内存。

HMODULE 和 HINSTANCE

在 Win16 编程中最容易混淆的事情就是模块句柄(HMODULE)和实例句柄(HINSTANCE)的区别。我刚讲过，HMODULE 表示一个已装入内存的 EXE 或 DLL。下一节我要讲，HINSTANCE 只是一个正在运行的任务或 DLL 的缺省数据段的全局堆句柄。从概念上说，HMODULE 和 HINSTANCE 差别很大。HMOD-

ULE 可以告诉你关于已装入的可执行程序的丰富的信息(诸如它的资源装载在什么地方等)。而 HINSTANCE，除告诉你段中的数据外，不能提供任何其他信息。

之所以会混淆 HMODULE 和 HINSTANCE，是因为许多 Win16 API 函数把许多实际需要 HMODULE 的参数定义成了 HINSTANCE 参数。例如，DialogBox 函数，它的第一个参数是 HINSTANCE。但是，可以仔细考虑一下，建立一个对话框都需要什么。很明显，DialogBox 函数需要知道在什么地方可以找到描述对话框的资源。资源保存在 EXE 或 DLL 中，所以毫无疑问 DialogBox 需要包含对话框资源的 NE 文件的 HMODULE。把一个 HINSTANCE 传递给 DialogBox 实际上毫无意义，因为数据段的全局堆句柄(HINSTANCE)对 DialogBox 函数获得对话资源毫无帮助。但是，也许你已知道，你完全可以把一个 HINSTANCE 传给函数 DialogBox，而且它成功地执行了。因此，暗中一定还有什么操作执行了。

未公开的 GetExePtr 函数提供了一个关键信息，使我们能理解 DialogBox(和其他 API 函数)怎样使用 HINSTANCE 值完成功能：

```
HMODULE GetExePtr( HANDLE );
```

GetExePtr 是一个神奇的函数，可以在其权限范围内作所有的事情以返回和传给它的句柄相关联的 HMODULE。如果你传给它一个 HINSTANCE 句柄，GetExePtr 就扫描所有的 DLL 和任务，查找哪个含有一个和传进的句柄相匹配的 HINSTANCE。如果找到了，就把该 DLL 或 EXE 的 HMODULE 返回。如果向 GetExePtr 传递一个 HMODULE，它就立即返回相同的 HMODULE。如果你单步运行进入 DialogBox 函数，就会看到其代码先调用 GetExePtr，然后，用它返回的 HMODULE 定位对话资源。因此，不管你传递给 DialogBox 的句柄是 HMODULE 还是 HINSTANCE，只要整个句柄合法，它就能完成所期望的功能。对于许多公开的带有 HINSTANCE 参数的其他 API 函数，有与此相同的情况。

知道了 HINSTANCE/HMODULE 参数分别是作什么用的，你就可以回答那些经常遇到的问题，如“我想在我的 EXE 中弹出一个对话框，但对话框资源在我的 DLL 中，我应把哪一个 HINSTANCE 传递给 DialogBox 呢?”答案是，可以传递任何一个 HINSTANCE 或 HMODULE 或包含该资源的任何 NE 文件。如果微软的文档对 API 函数的参数的用途介绍得更清楚些，就不会造成这么多混乱。

你也许想知道为什么大量的 Windows API 函数被说明为接收一个 HINSTANCE 参数，而内部又马上把 HINSTANCE 转化成 HMODULE。我想最合适的理由就是在你的程序中，HINSTANCE 比 HMODULE 更容易得到。一般情况下，程序或 DLL 并不知道它的 HMODULE，必须调用 GetModuleHandle 得到它。反过来，EXE 和 DLL 在启动时，都得到了传进来的 HINSTANCE。通过重取 SS 寄存器值，你可以很容易地重取主程序的 HINSTANCE。运行 DLL 代码时也是这样。当 EXE 启动时，它的 SS 寄存器被设置成和 DS 寄存器一样的值。尽管在 EXE 代码和 DLL 代码之间切换时，DS 寄存器的值会改变，但 SS 寄存器仍保持原值——也就是 EXE 的 DS 寄存器的值。

在 Win32 程序中(带有 Win32 异常)，这种 HINSTANCE 和 HMODULE 的含

义的模糊性变得更彻底。在 Win32 中，HMODULE 和 HINSTANCE 成为同样的东西；确切地说，HMODULE 和 HINSTANCE 二者都是内存中装载 EXE 或 DLL 的地方的基地址。

与模块相关的函数

前面已经讲述了 16 位模块数据库，现在让我们看一些存取或操纵模块数据库中信息的函数。在这一章中，我适当地选择了一组提供了伪码的函数。还有一些被选用的函数（如 LoadModule）没有提供伪码，因为它们实在太复杂了，由于篇幅所限，这里不再赘述。

GetModuleHandle 函数

当你研究有关模块数据库的函数时，GetModuleHandle 是最适合第一个研究的函数。那是因为它可以演示一些最重要的模块概念，而又不需要太多的伪码。GetModuleHandle 被说明为接收内存中一个模块的名字，并返回该模块数据库段的全局堆句柄（就是它的 HMODULE）。但是，文档没有说清模块名的定义。它是指实际的模块名（在模块驻留名表的第一个入口处），还是指模块的文件的名字？而且，通过后面的伪码可以看到，文档还遗漏了 GetModuleHandle 的行为中的一些其他好东西。

GetModuleHandle 的代码开始一段是作参数合法性检查的。代码检查每一个参数，确保它是合法的字符串指针。如果有非法参数，就返回到调用者，其调试版本还要置 0x700A 错误码（ERR_BAD_STRING_PTR）。如果字符串参数检查成功，代码就跳转到 IGetModuleHandle 代码（字符串参数仍保留在栈中）。

IGetModuleHandle 的第一段代码有点奇怪。它检查要搜索的模块名指针的高字（HIWORD）值是不是 0。如果是 0，则跳过所有将执行的正常代码，它把字符串指针的偏移量传给 GetExePtr，然后返回 GetExePtr 找到的信息（GetExePtr 将在下一节讨论，它返回和给定的全局句柄相联系的 HMODULE）。微软没有说明可以用 0 作为传递给 GetModuleHandle 的参数的高字，把该参数传给它。你几乎可以把任何与模块相联系的全局句柄（如 HINSTANCE）传递给 GetModuleHandle，而取回对应的 HMODULE。记住，GetModuleHandle 的字符串参数的高字要传 0，低字传你的句柄。整个句柄可以是 HINSTANCE，也可以是模块中的代码段或数据段，或是任何 GetExePtr 函数能处理的其他句柄。

IGetModuleHandle 的主要部分是搜索模块列表，查找模块数据库中和传给 GetModuleHandle 的字符串参数相同的名字。它按下面的顺序检查五种可能性：

- **可能性 1**：GetModuleHandle 得到的模块名和一些模块的驻留名表中第一个入口严格匹配。遍历系统的每个模块数据库作名字比较的函数是 FindExeInfo。FindExeInfo 的伪码在 IGetModuleHandle 的伪码后面，它很简单，无需注释。
- **可能性 2**：GetMudoleHandle 得到的模块名和一些模块的驻留名表中第一个入口

匹配，但大小写不同。检查这种情况和检查第一种可能性完全相同，只是 IGetModuleHandle 首先把字符串参数转换成大写形式，然后申请 FindExeInfo。

- **可能性 3**：代码传来的是文件名。这里有两种子状态：一个基本文件名（例如，KRNL386. EXE），或一个完整路径名（例如，C:\WINDOWS\SYSTEM\KRNL386. EXE）。IGetModuleHandle 在调用 FindExeFile 之前，处理两种情况，只留下基本文件名部分（如 KRNL386. EXE）。FindExeFile 和 FindExeInfo 非常相似，但是，它是把模块的文件名而不是模块名跟传进的字符串相比较。

IGetModuleHandle 的最后一段隐藏了另外两个未公开的秘密。在 Windows 3. 1 中，有一个叫做 TIMER. DRV 的 DLL；在 Windows 95 中，这个 DLL 消失了。可能一些应用通过调用 GetModuleHandle（TIMER）来检查它是否存在，GetModuleHandle 检查字符串 TIMER，如果找到就返回 1。看来微软是想以此保持这些应用的使用。当然，试图使用这个模块句柄的应用很不走运——但不管怎样它们能工作，对吗？第二个未公开的 IGetModuleHandle 的特征在前面提到过。任何对 GetModuleHandle 的调用，如果通过了参数合法性检查，其代码都将把模块列表（KERNEL）的头返回在 DX 寄存器中。

关于 GetModuleHandle 的最后要注意的一点是，不要试图用它得到 Win32 模块的 16 位模块数据库。这些模块没有插入到 HMODULE 列表中。本章后面的 SHOW16 程序给出一种粗苯而有效的办法找到这些 HMODULE。

GetModuleHandle 的伪码：

```
// Parameters:
//      LPSTR   lpszModName

    Verify that lpszModName is either a valid string pointer, or has
    a 0 in its HIWORD(). If not, RIP in the debug KERNEL with a code
    of 700A (ERR_BAD_STRING_PTR).
    goto IGetModuleHandle
```

IGetModuleHandle 的伪码：

```
// Parameters:
//      LPSTR   lpszModName
// Locals:
//      char    szBuffer[130];
//      WORD    len;
//      LPSTR   lpszBaseFilename;

    if ( HIWORD(lpszModName) == 0 )
        goto global_handle_in_LOWORD;

    // First let's assume that the user passed in a real module name (such as what
    // you'd put in the NAME or LIBRARY line in a .DEF file).
```

```
// Copy the string into a local buffer, but make the first byte
// be the length of the copied string (that is, make it a PASCAL string).
// 0 as the last parameter means copy the source exactly.
// Returns the length of the copied string.
len = CopyName( lpszModName, szBuffer, 0 );

// Scan through the list of modules in the system, looking for
// one with a module name that exactly matches the string passed
// to FarFindExeInfo. If a match is found, return the HMODULE in AX.
// The len parameter lets the function quickly eliminate modules with
// names of different lengths than the input module.
// This particular call is looking for the module name exactly as it
// was passed to GetModuleHandle.
AX = FarFindExeInfo( szBuffer+1, len );
if ( AX )
    goto return_AX;

// Do like the first CopyName call above, but this time the last
// parameter is -1, meaning uppercase the destination string.
len = CopyName( lpszModName, szBuffer, -1 );

// Do like the previous call to FarFindExeInfo, but this time we're
// searching for the uppercased version of the module name passed
// to GetModuleHandle.
AX = FarFindExeInfo( szBuffer+1, len );
if ( AX )
    goto return_AX;

// If we get here, we didn't find a real module name, so let's try
// looking for modules that have a filename matching what was
// passed to GetModuleHandle.

// NResGetPureName scans backward from the end of the string param
// until it finds a :, a \\, a /, or the start of the string. It
// returns a pointer to the next character. Essentially, this
// function returns a pointer to the base filename portion of a
// complete path. This allows you to pass names like
// C::\\WINDOWS\\SYSTEM\\KRNL386.EXE to GetModuleFileName

lpszBaseFilename = NResGetPureName( &szBuffer+1 );

// This function is essentially like FarFindExeInfo (above), but
// instead of comparing module names in the resident names table,
// it compares the base filenames.
AX = FindExeFile( lpszBaseFilename );
if ( AX )
    goto return_AX;
// If we get here, we didn't find a matching real module name or a
// matching filename. Do one last check to see if the string passed
// to GetModuleFileName was TIMER. In Windows 3.1, there was a
// TIMER.DRV, but that DLL doesn't exist in Windows 95. Perhaps this
// special-case code is to keep applications that look for the TIMER
// module from failing.
if ( 0 == strcmp(szBuffer+1, "TIMER") )
```

```
    {
        AX = 1;
        goto return_AX;
    }

global_handle_in_LOWORD:
    AX = GetExePtr( LOWORD(lpszModName) );

return_AX:          // Return whatever value is in the AX register.

    DX = hExeHead;  // Also return the head of module list in DX.
                    // Seems to always be KERNEL (KRNL386.EXE).
```

FindExeInfo(被 FarFindExeInfo 用同样的参数调用)的伪码：

```
// Parameters:
//      LPSTR   lpszSearchName;
//      WORD    len;
//  Locals:
//      LPMODULE lpModule;
//      LPBYTE  lpResNames;

    if ( !hExeHead  )
        return 0;

lpModule=MAKELP(hExeHead,0);
    while ( lpModule ) // Iterate through the list of modules.
    {
        // Get a pointer to the current module's name (the first entry in
        // the resident names table). The module name is prefixed by
        // a length byte.
        lpResNames = MAKELP(SELECTOROF(lpModule), lpModule->ne_resNamesTab);

        // If the length of the current module's name is the same as the
        // module name we're searching for, compare the two strings.  If
        // they match, we found the right module, so return its global
        // memory handle (its HMODULE). If the two strings differ in
        // length, don't bother to compare the strings.
        if ( *lpResNames == len )
          {
              if ( 0 == strcmp(lpResNames+1, lpszSearchName) )
                  return SELECTOROF(lpModule);
          }

          // A match was not found. Try the next module in the module list.
          lpModule = MAKELP( lpModule->ne_npNextExe, 0 );
      }
      return 0;
```

GetExePtr 函数

GetExePtr 可以说是 Windows 95 的未公开的 16 位函数中最有用的。它的接

口简单且固定不变，真不可思议，为什么微软要把这个精采的函数隐藏起来。研究 GetExePtr 是分析模块、任务、实例及全局内存句柄的内部联系的好方法。重要的是，观察 GetExePtr 等效于简要分析 16 位 KERNEL 的数据结构。

GetExePtr 的工作是根据一个输入的全局内存堆句柄，查找与该句柄相关联的 HMODULE。典型情况下，GetExePtr 被 KRNL386 内部使用，把 HINSTANCE 转换成 HMODULE。几乎你看到的每一个带 HINSTANCE 参数的函数，内部都调用了 GetExePtr 以获得 HMODULE。然而，GetExePtr 并不仅限于使用实例句柄。输入的句柄甚至可以是几乎任何类型的全局堆内存句柄。除了 HINSTANCE，GetExePtr 也接收 HTASK 句柄并返回生成该任务的 HMODULE。同样的，你也可以传送属于内存中模块的代码或数据选择器，GetExePtr 也能返回 HMODULE。你甚至可以传送一个由 GlobalAlloc 分配的句柄。GetExePtr 将返回和拥有这块内存的任务相关联的 HMODULE。一句话，GetExePtr 是一个能竭尽全力的函数，决不会轻易完不成任务。

虽然我还没有正式讲述任务和任务数据库(TDB)，但在 GetExePtr 的代码中还是把它们明显地标出来。直到本章下一节才讲任务，所以我要略作调整，讲一下 GetExePtr 伪码中的任务。

GetExePtr 代码开始是把传进来的句柄转化成一个选择器。其目的一般是保证句柄的最低位是打开的。下一步，GetExePtr 检查最好的情况：传进来的参数是一个 HMODULE。这个检查就是在段的第一个 WORD 中查找 NE 的标志码。如果检查成功，GetExePtr 的工作就完成了，它简单地返回该 HMODULE。如果传进来的句柄不是 HMODULE，GetExePtr 就需要做更多的搜索。第一件事是检查传进的参数是不是一个正在运行的任务的 HINSTANCE。函数的代码遍历任务列表，搜索其 HINSTANCE 和传进参数匹配的任务。如果 GetExePtr 找到了匹配的 HINSTANCE，就将用来生成该任务的 HMODULE 返回。

如果传进来的句柄既不是 HMODULE，也不是任务的 HINSTANCE，GetExePtr 就把传进的句柄传给另一个支持函数，该函数可以在系统的数据结构中做更多的搜索。在伪码中，我把这个函数称为 GetExePtrHelper。GetExePtrHelper 首先检查输入的句柄是不是一个带 CPU 的 LAR 指令的合法的 ring 3 选择器。如果不是，GetExePtrHelper 返回 0。

假设传进来的句柄通过了合法性检查，GetExePtrHelper 的下一个动作就是查找该句柄的拥有者。大多数全局堆块的拥有者是 HMODULE 或 PDB 段(PDB 与 DOS 的 PSP 非常相似)。EXE 或 DLL 文件中的代码段就是被 HMODULE 拥有的内存块的典型例子。通过 GlobalAlloc 分配的无 GMEM_SHARE 标志的内存块被当前任务的 PDB 拥有。得到传进句柄的拥有者后，GetExePtrHelper 检查该拥有者是不是 HMODULE。如果是，GetExePtrHelper 任务就完成了，返回拥有传进句柄的 HMODULE。

如果拥有者不是 HMODULE，GetExePtrHelper 下一步就通过在任务数据库(在后面“任务数据库[TDB]”一节讲述)中查找 TD 标志串以确定输入的句柄是不是 HTASK。如果传进句柄不是 HTASK，则传进句柄的拥有者可能是一个活动任

务的 PDB 段。为检查这种可能性，GetExePtrHelper 扫描任务表，取得每个任务的 PDB 选择器，把它与传进的句柄相比较。如果有匹配的 PDB 段，它所在任务的相关 HMODULE 就被 GetExePtrHelper 返回。

通过某种方式，GetExePtrHelper 把控制返回给 GetExePtr。如果 GetExePtrHelper 找到了一个 HMODULE，GetExePtr 就把那个 HMODULE 返回给它的调用者。否则，GetExePtrHelper 返回 0，这样，GetExePtr 就知道传进来的句柄是假的（不存在）。这时在 KRNL386 的调试版本中，代码将中断并显示下面的信息：

```
wn K16 GetExePtr(#ax) invalid parameter
```

其中 #ax 用传给 GetExePtr 的参数替换。

GetExePtr 的实现与 Windows 3.1 中相比有所变化。在 Windows 3.1 中，如果传给 GetExePtr 一个 HTASK 参数，它就会阻塞（尽管看上去有许多合理的事可做）。在 Windows 95 中，如果传给 GetExePtr 一个 HTASK，它将工作的很好。我在《Windows Internal》上对 Windows 3.1 版本的抱怨也许是这个改变的一部分原因。

GetExePtr 的伪码：

```
// Parameters:
//      HANDLE handle;
// Locals:
//      LPMODULE lpModule;
//      LPTDB lpTDB;    // Far pointer to Task Database.
//      WORD temp;

    if ( !(handle & 1) )    // If a MOVEABLE handle (bit 0 off), convert
    {                       // to a selector.
        handle = MYLOCK( handle );  // MYLOCK is similar to GlobalLock.
        if ( !handle )
            goto invalid_param;
    }

    // Try the obvious first: Were we passed an HMODULE?
    lpModule = MAKELP( handle, 0 );
    if ( lpModule->ne_signature == 'NE' )
    {
        AX = handle;
        goto return_AX;
    }

    // Okay. It's not a module. Perhaps it's the HINSTANCE of a task.
    // Or perhaps it's an HTASK. Walk through the list of tasks, checking
    // for this.

    lpTDB = MAKELP( HeadTDB, 0 );

    while ( lpTDB ) // While not at the end of the task list...
    {
```

```
        // Does this TDB match the handle passed in?
        if ( SELECTOROF(lpTDB) == handle )
            goto call_GetExePtrHelper   // Why not just return the HMODULE
                                        // here, rather than calling
                                        // GetExePtrHelper???
        // Does the HINSTANCE of this task match the handle passed in?
        if ( handle == lpTDB->TDB_HInstance )
        {
            AX = lpTDB->TDB_HMODULE;    // Yes! Return the HMODULE stored
            goto return_AX;             // in this task's TDB.
        }
        else
            lpTDB = MAKELP( lpTDB->TDB_next, 0 );   // Go on to next task.
    }

call_GetExePtrHelper:

    // Bring out the big guns by checking the PDBs in the task list in addition
to looking for the owning HMODULE in the Burgermaster arenas.
    // GetExePtrHelper returns an HMODULE, or 0.
    temp = GetExePtrHelper( handle );
    if ( temp )
        return temp;

    // Hmmm.... We still didn't find anything. Complain in the debug KERNEL.

    AX = handle
    _KRDEBUGTEST( "wn K16 GetExePtr(#ax) invalid parameter" );

    _AX = 0;    // Return 0 to the caller.

return_AX:
    CX = AX     // Return value both in AX and CX (good for JCXZ tests).
```

GetExePtrHelper 的伪码：

```
// Parameters:
//      WORD    handle;
// Locals:
//      LPMODULE lpModule;
//      LPTDB lpTDB;
//      WORD owner;

    LAR handle  // LAR instruction -> Load Access Rights (of selector).

    if ( LAR instruction returns failure code ) // Not a valid selector?
        return 0;

    if ( present bit not set in access rights )
    {
        owner = low 16-bits of handle's limit in the LDT
// In a not-present segment under Windows, the low 16 bits of
// the offset in the segment's descriptor hold the HMODULE
```

```
    // that owns that segment (if it's code/data segment belonging
    // to the module).
}
else
{
    owner = GetOwner( handle );  // Retrieve the owner out of the
    if ( !owner )                // appropriate arena in the
        return 0;                // Burgermaster segment.
}

// See if the owner of the block is an HMODULE. If so, return it.
lpModule = MAKELP( owner, 0 );
if ( lpModule->ne_signature == 'NE' )
    return SELECTOROF( lpModule );

// The owner wasn't an HMODULE. Is the handle parameter an HTASK?
// If so, return it.
LSL handle  // Get size of handle's segment.
if ( size of segment > 0xFB )
{
    lpTDB = MAKELP( handle, 0 );

    if ( lpTDB->TDB_sig == 'TD' )
        return lpTDB->TDB_HMODULE;
}

// Global memory blocks allocated without GMEM_SHARE are owned by
// the PDB of the task that allocated the memory. Walk the list
// of tasks looking for a task whose PDB matches the handle's owner.
// If a match is found, return the HMODULE associated with that task.

if ( HeadTDB == 0 ) // If no tasks, there is nothing more we can do to
    return 0;       // try to find additional modules.

lpTDB = MAKELP( HeadTDB, 0 )

while ( lpTDB )
{
    if ( owner == lpTDB->TDB_PSP )
        return lpTDB->TDB_HMODULE;

    lpTDB = MAKELP( lpTDB->TDB_next, 0 );
}
return 0;
```

GetProcAddress 函数

把 GetProcAddress 函数包含进这个有关 16 位模块的函数表有几个原因。第一个是，这个函数提供了一个很好的例子，让我们理解模块的入口表和它的驻留/非驻留名表是怎样联系的。第二个，研究 GetProcAddress 有助于你理解 Windows 的装入程序怎样解决对其他模块的预备。第三个，你可以用 GetProcAddress 探察 Windows 动态连接库的机理，这是 Windows 的最强大的特征。

GetProcAddress(就象许多其他 Windows API 函数)开头是一小块检查入口参数合法性的代码。在 GetProcAddress 中,合法性检查代码要确保你在 HINSTANCE 参数上传进的是一个合法的选择器(或 0 或-1)。对第二个参数(函数要查找的名字),合法性检查代码要检查你是否传进一个合法的 LPSTR 或 MAKEINTATDM 类型的字符串:高字(HIWORD)中为 0,且低字(LOWORD)中不为 0。GetProcAddress 认为你知道所查函数的引出顺序号,并把顺序号放进 LPSTR 参数的低字中。如果这两个参数的任何一个检查失败,GetProcAddress 立即返回调用者。例外情况是,当你运行调试版本的 KRNL386 时,你将在显示的提示信息中看到错误号 0x6002 或 0x700A。如果两个参数都正确,GetProcAddress 跳转至 IGetProAddress 代码,这是其代码的实质性部分。

GetProcAddress 做的第一事是用足可信赖的 GetExePtr 函数把你传给它的 HINSTANCE 转化成 HMODULE。前边讲过,GetExePtr 可以将任何句柄转化成 HMODULE,所以你实际并不限于只能把 HINSTANCE 传递给 GetProcAddress。任何和模块数据库相关联的全局内存句柄都可以。一旦 IGetProcAddress 得到了 HMODULE,它将在里边查找指定函数的 HMODULE,并将检查以确认 HMODULE 是属于一个 DLL 的。如果该 HMODULE 不是 DLL 的,IGetProcAddress 就使这次调用失败。如果运行运行调试版本的 KRNL386,发生这种现象时,会得到如下信息:

```
Can not GetProcAddress a task.
```

为什么要这样?Windows 程序设计人员加入这个检查来阻止程序对 EXE 文件的函数调用 GetProcAddress。EXE 文件中的函数只能在 EXE 程序的任务内存文本中调用,且栈寄存器要设置为程序的 DGROUP。KRNL386 通过使应用程序难以获得 EXE 文件中的引出函数的地址,可以避免因程序员调用 EXE 文件中的函数而使程序运行在错误的任务内存文本和错误的栈中。这个规定的例外情况是,如果你给 GetProcAddress 的 HINSTANCE 参数传一个 0,这时,IGetProcAddress 使用生成本任务的 HINSTANCE。用另一句话说就是,你可以对你的 EXE 和 DLL 使用 GetProcAddress,但不能对其他 EXE 使用。

IGetProcAddress 知道了要搜索的 HMODULE,下一步就是计算出给定函数的入口表顺序号。如果你传进的 LPSTR 参数高字为 0,则引出顺序号在低字里,这时 IGetProcAddress 就马上跳转到负责在模块的入口表中查找引出顺序号的代码处。但更多的情况下,传进的参数是一个 ASCII 字符串。因此,IGetProcAddress 必须把这个字符串转化成在目标模块中对应的引出顺序号。

把 LPSTR 参数转化成一个引出顺序号是 FarFindOrdinal 函数的工作。我没有提供 FarFindOrdinal 的伪码,因为想象它做了些什么并不困难。FarFindOrdinal 调用 FindOrdinal,它只是简单地扫描驻留和非驻留名表,把每个字符串和 GetProcAddress 的传入字符串作比较。如果有匹配的字符串,引出顺序号就是驻留或非驻留名表中随在匹配字符串后面的 WORD。GetProcAddress 的另一个未公开的用途是通过 FindOrdinal 把象 #97 这样的字符串转换成顺序号。这时,FindOrdinal 只是

简单的去掉前面的#，然后把串转化成它的二进制值(这里是 97)。

以上情况说明，有三种不同的方式调用 GetProcAddress，能的到相同的结果。例如，假如你想找到 GetMessage 的地址(在 USER.EXE 中，引出顺序号为 108)。下边三行中的任何一行都可以完成这个功能：

```
GetProcAddress( GetModuleHandle("USER"), "GetMessage" );
GetProcAddress( GetModuleHandle("USER"), MAKELONG(108, 0) );
GetProcAddress( GetModuleHandle("USER"), "#108" );
```

不管用哪种方式，IGetProcAddress 得到 HMODULE 和函数在 HMODULE 里边的顺序号后，它就扫描该模块的入口表，查找属于那个引出顺序号的入口。引出入口中含有计算所找函数在内存中地址所需的信息。扫描入口表和取出函数地址是 FarEntProcAddress 函数的工作，它只是 EntProcAddress 函数的一个外壳。

紧跟在 IGetProcAddress 伪码后面的是 EntProcAddress 的伪码。EntProcAddress 扫描入口表，这个数据结构在前面“入口表”一节讲过了。为便于快速刷新内存，每个入口表包是一组有连续引出顺序号的引出函数的入口表的记录。在每个包中，EntProcAddress 检查要搜索的引出顺序号是否包含在该包的记录数组中。当 EntProcAddress 找到正确的包后，就产生一个指针，指向包中所要的入口表记录。这个记录就可以用来计算引出函数在内存中的实际地址了。

如果你翻回“入口表”一节，看看入口表记录的格式，你将看到每个入口都包含了对应函数所在段号和段中的偏移量，但段号不是实际的选择器值，而是一个逻辑段号。因此，EntProcAddress 需要把逻辑段号转化成 Windows 装入程序赋给内存中该段的选择器。

EntProcAddress 怎样把逻辑段号转化成一个选择器?很简单，每个模块数据库都包含一个段表数组(在前面“段表”一节讲过)，把从入口表得到的逻辑段号作为段表入口数组的索引，我们所找函数的选择器值可以很容易地从相应段表入口最后一个 WORD 中取出。剩下的任务就是 EntProcAddress 把选择器值和偏移量合并成一个远指针，EntProcAddress 把这个远指针返回给 FarEntProcAddress，FarEntProcAddress 又把这个远地址返回给 IGetProcAddress，IGetProcAddress 最终把这个地址返回给 GetProcAddress 的调用者。

GetProcAddress 的伪码：

```
// Parameters:
//      HINSTANCE   hinst;
//      LPSTR       lpszProcName

   Validate the hinst parameter. The following rules apply:
       - If hinst is 0, it's okay.
       - If LDT bit (bit 2) is not set in selector, it's bad.
       - If hinst is -1, it's okay.
       - If LAR hints fails, it's bad.
```

```
    If any of these tests fail, RIP in the debug KERNEL with code 6022
    (ERR_BAD_GLOBAL_HANDLE).

    Validate the lpszProcName parameter. The following rules apply:
        - If lpszProcName is NULL, it's bad.
        - If HIWORD(lpszProcName) is 0, it's okay (unless LOWORD is also 0).
        - If lpszProcName is an invalid pointer, it's bad.
        - If lpszProcName is > 0x100 bytes long, it's bad.
    If any of these tests fail, RIP in the debug KERNEL with code 700A
    (ERR_BAD_STRING_PTR).
    goto IGetProcAddress
```

IGetProcAddress 过程的伪码：

```
// Parameters:
//      HINSTANCE   hinst;
//      LPSTR       lpszProcName
// Locals:
//      char        szBuffer[130];
//      WORD        hModule;
//      WORD        exportOrdinal;
//      LPMODULE    lpModule;

    if ( hinst )
    {
        hModule = GetExePtr(hinst)
        if ( !hModule )
            return 0;

        lpModule = MAKELP(hModule, 0);
        if ( lpModule->ne_flags & MODFLAGS_DLL )
            goto have_HMODULE;

        FarKernelError( "Can not GetProcAddress a task." );
        return 0;
    }
    else    // hinst parameter was 0.
    {
        hModule = CurTDB->TDB_HMODULE;
    }

have_HMODULE:
    if ( HIWORD(lpszProcName) == 0 )
    {
        exportOrdinal = LOWORD( lpszProcName );
        goto have_export_ordinal;
    }

    CopyName( lpszProcName, szBuffer, 0 );

    exportOrdinal = FarFindOrdinal( hModule, szBuffer, -1, 0 );
    if ( exportOrdinal )
        goto have_export_ordinal;
```

```
    CopyName( lpszProcName, szBuffer, 0 );

    exportOrdinal = FarFindOrdinal( hModule, szBuffer, -1, 0 );
    if ( !exportOrdinal )
        return 0;

    // RIP in the debug KERNEL with code 0x5000.
    _KRDEBUGTEST("wn K16 GetProcAddress(@ES:BX) case-sensitive new for Win4");

have_export_ordinal:

    return FarEntProcAddress( hModule, exportOrdinal );
```

EntProcAddress 过程的伪码：

```
// Parameters:
//      HMODULE     hModule
//      WORD        exportOrdinal
//      WORD        fComplain
// Locals:
//      LPENTRY_BUNDLE_HEADER lpBundle;     // See HMODULE.H.
//      LPENTRY lpEntry;                    // See HMODULE.H.
//      LPMODULE lpModule;
//      LPSEGMENT_RECORD    lpSeg;

    lpModule = MAKELP( hModule, 0 );    // Make pointer to module database.

    // Check for invalid export ordinals.
    if ( (exportOrdinal == 0) || (exportOrdinal == 0x8000) )
        goto invalid_ordinal;

    exportOrdinal--;    // Ordinals in entry table are zero-based, so adjust.
// Make a pointer to the module's entry table.
lpBundle = MAKELP( hModule, lpModule->ne_npEntryTable );

// Walk through the list of bundles. Look for the bundle whose  starting
// and ending ordinals encompass the exportedOrdinal that was passed.
while ( lpBundle->firstEntry < exportOrdinal )
{
    if ( lpBundle->lastEntry > exportOrdinal )
    {
        // Each bundle is immediately followed by an array of ENTRY
        // structures.
        lpEntry = address of the appropriate slot in the array
                    of ENTRY structures following the bundle header.

        goto have_entry_pointer;
    }

    // Go on to the next bundle.
    lpBundle = MAKELP( hModule, lpBundle->nextBundle );
}
```

```
invalid_ordinal:

    // Something went wrong...
    if ( !fComplain )
    {
        // RIP in the debug KERNEL with code 0x5004.
        BX = exportOrdinal
        _KRDEBUGTEST("wn K16 Invalid ordinal reference (##BX) to %ES1");
    }

    return 0;

have_entry_pointer:
    // At this point, we've found the correct entry in the entry table.
    // Now we have to decode the entry information to an address that we
    // can pass back to the caller.

    // If this entry is from segment 0xFE, it's one of the special
    // entries (for example, __F000H). Return the entry's offset.
    if ( lpEntry->segType == 0xFE )
        return MAKELP( 0xFFFF, lpEntry->offset );

    // There are two types of entries: MOVEABLE or FIXED.
    // FIXED entries have segment numbers between 1 and 253.
    // MOVEABLE entries are indicated by a segment number of
    // 0xFF. Take special action if it's a FIXED entry.
    if ( lpEntry->segType != 0xFF )
    {
        // The entry is in a FIXED segment. Make sure that segment is
        // loaded in memory.
        if ( !LoadSegment(hModule, lpEntry->segNumber, -1, -1) )
            goto invalid_ordinal;
    }

    // Point at the appropriate segment structure in the segment table.
    // We need to do this in order to look up the handle/selector assigned
    // to the segment by the Windows loader.
    lpSeg = lpModule->ne_segtab[lpEntry->segNumber-1];
    if ( lpSeg->handle == 0 )   // Make sure there's a handle for this segment.
        return 0;

    // Combine the segment and the offset to create the entry point address.
    return MAKELP( lpSeg->handle & 1, lpEntry->offset );
```

16 位任务

如果把 Windows 的模块看成表示部件的无生命的躯体，那任务就可以认为是赋予那个躯体生命的火花。不过在讲述任务如何完成它神奇的工作之前，我需要解释两个关于术语的问题。任务有时用来指程序，在 16 位 Windows 中，正确的术语是任务(TASK)，不是程序。在 Win32 部分，术语进程(Process)替换了任务这个词，尽管从概念上讲 16 位任务和 32 位的进程是指同一事物。在这一节，我们来

分析 Windows 95 的 16 位部分的任务。

在 Windows 95 中，任务表示两种东西。首先，任务表示代码的执行。在 Windows 3.1 或更早时，任务是调度的基本单位。在某一段时间中，只有一个任务在运行。第二点，任务表示一种所有权，每个任务有自己的一组文件句柄，它建立的窗口，它分配的内存，等等。在本节后面，我还将讨论这几点。

每次 Windows 95 启动一个程序，KRNL386 就建立一个新任务。如果你启动两个 CALC.EXE 的拷贝，Windows 95 就向 KRNL386 的任务表中加入两个任务。甚至对 Win32 进程，Windows 95 也要建立一个 16 位任务表示。16 位的部件可能会高兴，因为这是用老的 16 位代码认识的格式来表示 Win32 进程的存在。

一个任务存在的一般证明是称为任务数据库(Task Database，简写为 TDB)的数据结构。TDB 包含了一个程序的特定实例专有的信息。单独的 TDB 域在下一节描述，这一节把精力集中在更一般的有关 TDB 的概念上。

TDB 是一组域，在从 16 位全局堆分配的段中。该段的全局堆句柄称为 HTASK。知道了 HTASK 只是一个选择器，你就可以直接读取 TDB 的域。在这方面，任务数据库和它的 HTASK 和模块数据库及其 HMODULE 非常相似。GetCurrentTask 返回一个 HTASK，并且你可以把 HTASK 传递给象 PostAppMessage 和 EnumTaskWindows 这样的函数。

从某些方面来讲，Windows 的任务和 DOS 的程序相似。在 DOS 中，每个正在运行的程序都有一个程序段前缀(Program Segment Prefix，简称为 PSP)，其中包含了文件句柄和一些附加信息，如程序的命令行。因为 Windows 最初是 DOS 的扩展，所以 Windows 的任务在它的 HTASK 段中总是保留着一个 DOS 的 PSP。在 Windows 95 之前的 Windows 版本中，当执行实模式 DOS 操作，如文件 I/O 时，Windows 实际使用了任务的 TDB 中的 PSP 区域。在 Windows 的术语中，TDB 中的 PSP 区域叫做 PDB(即 Process Database)；注意不要把它和 Win32 进程数据库(Win32 Process Database)相混淆。返回当前任务的 PSP 的 Windows 函数称作 GetCurrentPDB，而不是 GetCurrentPSP。除此之外可以说，在 Windows 中，PSP 和 PDB 是同样的东西。这里要强调一点，Windows TDB 中包含的是混合物，有原来实模式 DOS 的东西，也有新的只在 16 位保护模式环境中才有意义的东西。在 Windows 95 中，又向该混合物加入了东西，在 TDB 中包含一个指向 Win32 线程数据库的指针，这样 Windows 95 的任务数据库实际上成了 DOS、Win16 和 Win32 信息的总汇。

就象维持生命必须要有躯体一样，没有模块，任务也不能存在。当你第一次启动一个程序时，Windows 95 建立一个 16 位模块数据库，然后为这个新任务建立一个任务数据库。如果这时你启动这个程序的新实例(原来的实例仍然在运行)，Windows 95 将建立又一个任务数据库，但是并不建立新的模块数据库。两个任务都联接于同一个模块数据库(HMODULE)。模块表示象代码和资源这样的东西，它们对程序的正在运行的多个实例来说是公共的。任务表示对程序的多个实例来说不一样的信息。如堆栈段不同，当前工作目录相同。

姑且忘掉 32 位进程和线程，在任何给定时间，Windows 95 正在运行一个且只

有一个任务。除当前正在运行的线程，所有其他线程都阻塞了，并且直到正在运行的任务放弃对 CPU 的控制后，才会有机会重新运行。这就是协同式多任务方式。每个任务都是需要运行多长时间，就运行多长时间，然后才放弃 CPU 控制权使另一个任务运行。

任务怎样放弃控制权呢？一般来说，任务通过使用象 GetMessage，PeekMessage，SentMessage，和 WaitMessage 这样的函数放弃控制。如果这些函数确认该任务不需要继续运行了（例如，没有等待处理的消息），它们就调用 16 位调度程序，如果 16 位调度程序发现另一个任务有工作要做，它就挂起第一个任务，并切换到有工作要做的任务。大多数情况下，对程序员隐藏了放弃控制权的需求，因为象 GetMessage 这样的函数透明地管理着协同多任务方式。

Windows 95 用类似于 16 位模块数据库列表的方式跟踪任务列表。在每个 TDB 中有一个 WORD 域包含了下一个任务的选择器。任务链表和前面讲的模块列表不同，它不是静态的。任务的顺序在不断地变化，以便于 16 位程序调度操作。有趣的是，对为表示 16 位领域中的 Win32 进程而建立的 TDB 来说，其顺序并没有因 16 位调度程序的执行而改变。相反，看上去，为 Win32 进程建立的 TDB 似乎在链表头上扎下了根，除非该任务/进程退出了，它不会移动。详细情况请参看后面“任务数据库（TDB）”一节中偏移量 8 处域的解释。

微软认可的观察任务表的方法是使用 TOOLHELP 的 TaskFirst 和 TaskNext 函数。如果你想直接观察链表（象 SHOW16 程序那样），可以使用调用 GetCurrentTask 函数后 DX 寄存器中的值，它是任务表的头。也可以通过读出 KRNL386 输出的符号 THHOOK 后面 0xE 字节处 WORD 的值，这是链表中的第一个 TDB（也就是，为 THHOOK 调用 GetProcAddress，在符号返回地址的偏移量部分加上 0xE，然后读出那个位置的 WORD）。THHOOK 周围的内存还包含了其他几个有用的 KRNL386 全局变量：

THHOOK+0　　hGlobalHeap

这是一个句柄（选择器-1），指向保存有关 16 位全局堆信息的 Burgermaster 数据结构。更多信息见第二章。调用未公开的 KRNL386 函数 GlobalMasterHandle 后，这个值也返回在 AX 寄存器中。

THHOOK+2　　pGlobalHeap

这是指向 Burgermaster 段的选择器，实际上和 hGlobalHeap 域所指的是同一东西。调用 GlobalMasterHandle 后，该值就在 DX 寄存器中。

THHOOK+4　　hExeHeap

这个 WORD 存放 16 位模块表中第一个模块的 HMODULE。第一个模块总是 KERNEL（KRNL386. EXE）。这个值也可在成功调用 GetModuleHandle 后的 DX 寄存器中找到。更多信息见模块数据库中偏移量 06h 处的解释（在本章“NE 头”一节）。

THHOOK+8　　topPDB

KRNL386.EXE 的 PSP(啊,是 PDB)段的选择器。这是作为实模式 DOS 可执行程序装入 KRNL386 时的 PSP。这个值被 GetCurrentPDB 返回在 DX 寄存器中。

THHOOK+0Ah　headPDB

PDB 链表中第一个 PDB 的 PDB/PSP 选择器。

THHOOK+0Eh　HeadTDB

TDB 链表中第一个 TDB,这个值由 GetCurrentTask 返回。

THHOOK+10h　CurTDB

当前正运行的任务的 TDB。这几乎总是链表中的最后一个任务。该值由 GetCurrentTask 返回在 AX 寄存器中。

THHOOK+12h　LoadTDB

除了有新任务正在建立过程中,这个域总为 0。它非 0 时,就包含一个 TDB 选择器的值,新任务要用这个值。

THHOOK+16h　SelTableLen

这个 WORD 是 Burgermaster 段中选择器表(一个 DWORD 数组)的长度。详细描述见第二章。

THHOOK+18h　SelTableStart

这个 DWORD 是 Burgermaster 段中选择器表(一个 DWORD 数组)的起始偏移量。详细描述请见第二章。

与模块数据库不同,Windows 95 很好地保持着所有的任务数据库,不管是 16 位还是 32 位应用的,都在任务链表中。所以你可以用 16 位 TOOLHELP.DLL 的 TaskFirst 和 TaskNext 函数观察这个所有正运行程序的链表,而不用管它们是 16 位的还是 32 位的。同样,与 16 位模块不同,所有的任务有一个对应的 Windows 95 的 32 位部分的表示。确切地说,每个 Windows 95 任务数据库都含有一个 32 位展开的指针,指向 Win32 线程数据库。甚至对 16 位程序也是如此。归纳起来就是:每个程序(无论它是 16 位 NE 程序还是 32 位 PE 程序)都有一个 16 位任务数据库和一个 32 位线程数据库(同样有一个对应的 32 位进程)。

一些关于任务的错误认识

任务有时比较难理解,所以程序员有一些关于任务的错误认识也不足为奇。这一节讲述并澄清两个经常遇到的错误认识。

一个最普遍的错误认识是“每个任务有一个窗口”。尽管屏幕上的窗口是任务存在的最显而易见的证明，但任务和窗口实际上毫不相干，不要把它们混为一谈。任务只表示运行过程，除此之外，别无他义。任务是否显示窗口完全取决于你。当然可以很容易地建立一个任务而不建立任何窗口，使它完成有用的工作。当你分析 Windows 95 中的“任务链表”时，这一点很重要，一定要记住。那个链表显示顶层窗口，与 KRNL386 维护的真正的任务链表完全不是一回事。如果你运行随书所附磁盘上的 SHOW16 程序，就可以看到在输出窗口列表中未显示的任务。

程序员们另一个普遍的错误认识是“DLL 有类似于任务的特性”。这些程序员说了这样一些话，“我想让我的 DLL 建立一个能被它的所有客户程序使用的窗口”，再如，“我的 DLL 将打开一个文件句柄，使用 DLL 的不同程序都可以使用该句柄”。这些话说明程序员有这样的错误认识，就是 DLL 拥有文件句柄和窗口。DLL 只不过是一些被任务使用的附加代码。DLL 的代码所在文件与 EXE 的完全不同。因此它与系统资源的拥有权根本扯不上关系。

建立窗口或打开文件句柄的 DLL 只是代表调用它的任务来做这些事。而这个 DLL 自己无权拥有这些东西。因此，当你在一个 DLL 中调用 CreateWindows 建立一个窗口时，是当前正在运行的任务拥有这个的窗口，而不是 DLL。如果该任务结束了，即使 DLL 还保持在内存中，那个窗口也不存在了。同样地，如果一个 DLL 打开一个文件句柄，这个句柄属于当前任务。如果另一个任务现在也调用 DLL，并且试图使用为第一个任务打开的文件句柄，就会产生一个错误——或更糟糕的情况——使用了错误的文件句柄。为什么呢？因为文件句柄只对于第一个正在运行的任务（打开文件的任务）是合法的。

任务数据库（TDB）

前边几节概况地讲了一些关于任务的一般性概念。在这一节，讲述有关任务数据库（TDB）的细节。Windows 的 TDB 中的每一个域都列在这里并给予解释；如果你只想快速浏览一遍 TDB 的各域，可以看附盘上 SHOW16 代码中的 TDB. H 头文件。和前边讲述模块数据库时一样，说明的第一行仍是三个条目：域在任务数据库中的偏移量，域的类型（如 WORD 或 DWORD）和一段简短的描述。

00h　　WORD　　下一个 TDB

这个 WORD 是 Win16 任务链表中下一个任务的 HTASK。链表头由 KRNL386 的全局变量 HeadTDB 给出（它的值在调用 GetCurrentTask 后的 DX 寄存器中）。本域为 0 说明这是链表尾。

02h　　DWORD　　任务的 SS:SP

这个 DWORD 是任务挂起在 16 位调度程序中时，它的 SS:SP 值。在这个地址的一个固定偏移量处，有当该任务被调用时要恢复到 CPU 寄存器中的各寄存器值。实际上，TOOLHELP 中的 TaskSwitch 和 TaskSetCSIP 就依靠这些寄存器值来

完成它们神奇的功能。本域对当前正在运行的任务毫无意义，因为这个任务没有被阻塞在 16 位调度程序中。

06h　　WORD　　事件的数目

这个 WORD 中是等待任务处理的事件的数目。通常在 Windows 编程中不会遇到事件的问题。当一个事件发生时，常常是表示一个需要任务处理的等待窗口消息。例如，你向一个应用发了一条消息，该消息被写入那个任务的消息队列(message queue)，而且该任务的事件计数器增值。但是，事件和 Windows 消息含义不同，任务可以有不对应等待窗口消息的等待事件。事件是 16 位任务调度程序用来决定是否唤醒一个任务使它运行的尺度。调度程序只启动事件数目非 0 的任务。

08h　　BYTE　　优先级

这个 BYTE 中是任务的相对调度优先级。但是这个域并不被任何 Windows 应用使用，应用都在相同的优先级上结束运行。从理论上讲，这个域的值范围在-32 到 15 之间，由 KRNL386 中的未公开的 SetPriority 函数设置。KRNL386 将任务按优先级排序，优先级低的任务最先检测等待事件。但是，调整你的任务的优先级不会给你带来任何好处。因为调度程序只根据是否有等待事件来调度任务。你可以把你的任务的优先级给成-32，但如果没有等待事件，它仍不会被调度。

09h　　BYTE

这个域没有用。

0Ah，0Eh，10h，12h　　WORD　　未用域

TDB 的这些域只在 OS/2 1.x 和 Windows 共享代码的时代被 OS/2 1.x 的程序用作线程信息。在 Windows 3.x 和 Windows 95 中，这些域没有用，总被置为 0。

0Ch　　WORD　　这个 TDB

这个 WORD 存放这个 TDB 的 TDB(就是自己引用自己)。

14h　　WORD　　浮点控制字

在 Windows 3.x 中，这个 WORD 存放任务切换出去时的浮点控制字。浮点控制字包含 80x87 数学协处理器的状态标志，通过 CPU 指令 FLDCW 和 FSTCW 保存和恢复。在 Windows 95 中，这个域没有用。这可能是因为 Windows 95 任务切换也涉及 Win32 线程切换，浮点控制字可能在 ring 0 线程切换时保存和恢复。

16h　　WORD　　任务状态标志

这个 WORD 中有如下的位域标志：

标志和位值	解释
TDBF_WIN32 0x0010h	如果置位，说明任务是一个 Win32 程序，在为运行在 Win32 下的 Win32 应用建立的 TDB 中，这一位也置位。
TDBF_NEWTASK 0x0008	当建立一个 Win16 任务时，被置位，任务首次进入 16 位调度程序(Reschedule 函数)时被清除。
TDBF_WINOLDAP 0x0001h	这个任务是 WINOA386.MOD(模块名:WINOLDAP)。WINOLDAP 任务在 Windows 95 下用来在 DOS 程序自己的虚拟机上运行它们。WINOLADAP 有点象 DOS 程序的一种外壳。在任务表中，你看到的是名字 WINOLDAP，而不是 DOS 程序的名字。

18h　　WORD　　错误模式

这个域包含一组位域，可由用户设置 Windows 95 对发生在任务中的特定错误的反应。这些状态可以用 SetErrorMode API 函数设置。概况的标志如下：

标志名和位值	解释
SEM_FAILCRITICALERRORS 0x0001	从遇到提示错(由“Abort,Retry,Ignore?”错误信息说明的错误)的 DOS 函数调用中悄悄地返回一个失败。如果这个标志没有置位，Windows 95 将弹出一个对话框询问如何进行
SEM_NOGPFAULTERRORBOX 0x0002	发生 GP 错时，不显示通常的 GP 错对话框，在 Windows 3.x 中，这个标志一般由调试程序在想结束被调试程序时使用。调试器设置被调试程序的这个标志，然后修改被调试程序，使它运行时产生一个 GP 错，从而 Windows 中止这个应用(不显示 GP 错对话框)。可参看 TOOLHELP 的 TerminateApp 文档，因为 TerminateApp 可以选择修改这个标志。
SEM_NOOPENFILEERRORBOX 0x8000	当找不到文件时，不显示对话框。这个标志常用于你想使失败的 LoadLibrary 调用悄悄地失败退出而不显示文件未找到的对话框。

1Ah　　WORD　　期望的 Windows 版本

这个字中是运行本程序所需 Windows 的最低版本号。这个域是生成这个任务的可执行文件的模块数据库偏移量 0x3E 处期望 Windows 版本号的复制品。详细情况见“NE 头”一节中 06h 入口的内容。

1Ch　　WORD　　这个任务的 HINSTANCE

这个字中是这个任务的 HINSTANCE。HINSTANCE 只不过是任务的缺省数据段 DGROUP 的全局堆句柄。这个 HINSTANCE 值作为第一个参数传给 Win-

Main 函数。HINSTANCE/GROUP 段和任务的堆栈段也一样。任务的每个模块都有自己的 HINSTANCE 值,它常用来区分正在运行的程序(尽管 TDB 也同样可以作这个工作)。对 Win32 任务来说,TDB 中的 HINSTANCE 和 HMODULE 域(偏移量 1Eh,下一个就要讲)相同。

1Eh	WORD	这个任务的模块句柄

这个字中是用来建立任务的 EXE 被装入内存后的 HMODULE。这个句柄可以传给 GetModuleFileName 以获得与该任务相关联的 EXE 文件的名字。

20h	WORD	消息队列

这个域中是任务消息队列的选择器。消息队列是传给任务的窗口的消息所在的地方。与 Windows 早期版本不同,Windows 95 没有限制每个消息队列中最多消息数。第四章对此有详细讲述。

22h	WORD	父任务的 TDB

这个 WORD 是运行本任务的 WinExec 的任务的 TDB 选择器。例如,如果你正在调试一个程序,那么父任务就是调试器任务 TDB。典型情况下,如果你从 Explorer 启动一个程序,则该程序的父任务就是 EXPLORER. EXE,如果你从 DOS 命令行启动一个程序,父任务就是 MSGSRV32. EXE。对 Win32 应用,父任务 TDB 总是 0。

24h	WORD	应用信号灯

在 Windows 3. 1 中,这个域的值影响任务的应用信号过程的工作,但确切含义还不清楚。在 Windows 95 中,应用信号过程地址(偏移量 26h)好象没有用。

26h	DWORD	Windows 3. 1 应用信号过程

在 Windows 3. x 中,本域是一个指针,指向应用的信号过程。应用信号过程是在 ctrl-break 按下时,用来回调一个程序的。信号过程通过调用未公开的 SetSigHandler 函数设置。在 Windows 95 中,没有 SetSigHandler 了,本域值总为 0。

2Ah	DWORD	USER 信号过程

这个域中是指向 USER 信号过程的指针。USER 信号过程在装载或卸载 DLL 时被调用。这给了 USER 一个机会,清理留在内存中的系统资源。在卸载回调时,USER 也调用 GDI 的 SignalProc 函数,从而给 GDI 一个机会清理(或作上标志以备将来清理)任何未释放的 GDI 资源。

调用未公开的函数 SetTaskSignalProc(KERNEL. 386)可以修改 TDB 中的信号处理程序。该函数原型如下:

```
FARPROC SetTaskSignalProc( HTASK hTask, FARPROC lpfnNewSignalProc );
```

返回值是原来的信号过程的地址。

USER 信号过程回调函数如下：

```
void FAR PASCAL UserSignalProc(
                        HMODULE hModule,    // Module under consideration.
                        WORD actionCode,    // See actionCode values, below.
                        WORD unknown,
                        HISNTANCE hInstance,
                        WORD hQueue);

    actionCode values:
    0x0040  DLL Load
    0x0080  DLL Unload
    0x0100  ??? (task exit?)
```

在 Windows 3.1 中，TOOLHELP.DLL 用自己的处理程序替换了 USER 信号过程。在 TOOLHELP 的处理程序中，TOOLHELP 为正在运行的任务卸掉所有已安装的中断或标志处理程序。在 Windows 95 中，TOOLHELP 不再使用信号过程，而是使用新的 DLLENTRYPOINT 机制，这一机制在 Windows 95 转换程序编译器文档中说明。

2Ch　　DWORD　　全局标志回调过程(GlobalNotify callback)

这个 DWORD 中是一个指针，指向任务的全局标志回调过程。KRNL386 在丢弃可丢弃的全局堆时调用这一过程。这个回调函数既可以允许 KRNL386 丢弃块，也可以阻止它丢弃块，这取决于回调函数的返回值。生成新任务时，这个域被初始化成 0。

30h　　DWORD[7]　　任务中断处理程序(中断 0,2,4,6,7,3Eh,75h)

Windows 95 中，对大多数中断都有一个可被所有任务使用的全局句柄。但是，Windows 95 允许任务为特定中断装入自己的处理程序(提供 INT 21h 的功能 25h)。当发生一个这样的中断时，Windows 95 在当前任务的 TDB 中查找该中断处理程序并调用它。在 TDB 的这个 7 元素 DWORD 数组中，每个 DWORD 存放一个特定中断处理号的中断处理程序入口。基于单任务处理的中断如下：

0 —— 除 0 错
2 —— NMI
4 —— INTO
6 —— 非法操作码
7 —— 协处理器不可用
3Eh —— 80x87 仿真
75h —— 80x87 错

建立任务时，在它的 TDB 中有系统提供的缺省处理过程。CALC.EXE 是一个

修改中断向量的很好的例子。附盘上的 SHOW16 程序使你可以清楚地看到任务安装的中断处理程序。

4Eh DWORD 兼容标志

这个标志是在 Windows 3.1 中引入的。Windows 3.1 中，在许多方面和 Windows 的早期版本有所不同，但许多应用依赖于早期 Windows 的一些行为特征，所以在 Windows 3.1 中运行这些程序时，必须告诉 Windows 3.1 使用和早期版本相同的特征，以使这些应用能正常运行，这个标志就是这个作用。当 Windows 看到运行的是需要早期 Windows 特征的应用时，就检查这些标志，并依此为根据对 Windows 系统作适当调整。从 WIN.INI 文件中[Compatibility]一节的内容可以看到需要这些兼容性调整的程序的模块名。令人奇怪的是，许多这样的应用是微软公司自己的程序。《Undocumented Windows》一书第五章中有一个位域及其在 Windows 3.1 中含义的列表。在 Windows 95 中又加进一些标志。你可以用未公开的 GetAppCompatFlags 函数取出一个特定任务的标志。

```
DWORD FAR PASCAL GetAppCompatFlags(HTASK hTask);
```

52h WORD TIB 选择器

这是 Win32 线程代码访问 TIB(线程信息块，thread information block)的 FS 寄存器值。所有的任务(甚至 16 位任务)都为自己维护着 Win32 进程的线程。用来访问任务的 Win32 线程信息的指针和选择器的拷贝保存在每个任务的 TDB 段中。

线程信息块包含有单线程信息，包括下面各域：

```
00h DWORD   pvExcept        // Head of exception record list.
04h DWORD   pvStackUserTop  // Top of thread's stack.
08h DWORD   pvStackUserBase // Base of thread's stack.
30h DWORD   pvTLSArray      // Pointer to Thread Local Storage array.
```

TIB 结构在线程数据库中从 0x10 字节处开始。在 TDB 的下一个域(偏移量 54h)中给出了线程数据库的展开的 32 位指针。详细描述见第三章有关内容。

54h DWORD 任务的线程数据库(THREAD_DATABASE)的线性地址

这个 DWORD 存放与此任务相关的 ring 3 线程数据库的展开的 32 位线性地址。线程数据库包含了线程信息块(见域 52h)，它从线程信息块前 0x10 个字节处开始。有关线程数据库的详细情况见第三章内容。

58h WORD DGROUP 任务句柄

对基于 16 位的任务，这个 WORD 是 DGROUP 段的全局堆句柄。在观察

KRNL386 调试版本中的错误信息时，这个域可能用来在 16/32 位执行程序转换时获得任务的原子表段(也就是它的 DGROUP)的句柄。对于基于 Win32 的任务，这个域总是 0。

5Ah　BYTE[6]　未用

这 6 个字节在 Windows 95 中未用。

60h　WORD　任务的 PDB

这个域是任务的 PDB(即 PSP)段的选择器。PDB/PSP 中包含着任务的文件句柄表，它的命令行，和其他在大量 DOS 编程书上公开的各种各样的域。在 Win16 任务中，每个任务的 PDB 就存储在 HTASK 选择器可访问内存的尾部。特别是，PDB 的基址总是比 HTASK 选择器大 0x210 字节。对 Win32 任务，PDB 总是在 1MB 以下的线性地址中，而 TDB 的地址常常高于 2GB。

GetCurrentPDB 函数返回当前 TDB 的本域值。

62h　DWORD　DOS 磁盘传送区

这个 DWORD 指向 MS-DOS 磁盘传送区(DTA)。关于 DTA 的详细内容请参看 DOS 编程书籍。对 Win16 任务，PDB 段中 DTA 的初始值是 80h 字节(也就是，本域的选择器部分与偏移量 60h 处的 WORD 值匹配)。所有的 Win32 任务共享同一个 DTA 值。

66h　BYTE　当前驱动器

这个 BYTE 包含任务当前目录的驱动器部分。本域值用 0x80 作了调整，所以要获得驱动器号码，须从这个 BYTE 的值中减去 0x80(0x80＝驱动器 A，0x81＝驱动器 B，依此类推)。在 Windows 3.x 中，当前目录的路径部分就存在本域后面。但在 Windows 95 中，路径移到偏移量 0x100 处。详细情况见偏移量 0x100 处域的说明。

67h　char[65]　未用

在 Windows 3.x 中，这个数组包含任务的当前目录的路径部分，路径的最大长度限制在 65 个字符。对于 Windows 95 中的长文件名，这个长度太小了，无法放下可能的最长路径；现在当前路径存储在 TDB 的偏移量 0x100 处。

A8h　WORD　初始任务合法性检查

在 Windows 3.x 中，本域中初始值是任务启动时 AX 中应包含的值。但是，现在未发现任何启动代码检查这个标志，所以实际上它已成为未用的域。

AAh　WORD　要调度的下一个任务(DirectedYield)

如果为非 0 值，则本域中包含调用调度程序时，Win16 调度程序要唤醒的

HTASK 值。如果你调用 DirectedYield 指定一个任务为下一个要运行的，则它的 TDB 的本域值为非 0 值，否则，这个域的值总为 0。DirectedYield 把 HTASK 参数存进这个域，然后调用 Win16 调度程序(Reschedule)。在 Reschedule 函数的开始部分，它检查本域值，如果非 0，就绕过对下一个要调度任务的搜索，直接调用这个任务。Reschedule 将本域值置为 0，所以很少看到本域为非 0 值。

ACh DWORD 要初始化的 DLL 列表的地址，形式为选择器:偏移量

在应用启动时，这个 DWORD 中包含一个指针，指向一个 DLL 模块句柄的一个数组，该数组以一个 0 元素结尾。所有这些 DLL 都是第一次被装入内存，所以需调用它们每一个的 LibMain 入口点。启动这个任务时，被它引用的 DLL 如果已经在内存中，则这个 DLL 的 HMODULE 不在这个数组中。InitTask 函数扫描这个 HMODULE 数组，调用它们每一个的 LibMain 入口点，然后，释放 HMODULE 数组占用的内存并将此 DWORD 设置为 0。注意:本域中的长指针，其选择器和偏移量与普通长指针位置相反，选择器在低 WORD 中，偏移量在高 WORD 中。

B0h WORD 这个 TDB 的代码段的别名

Windows 95 开始要在 TDB 中为自己建立 MakeProcInstance 转换程序(见域 BAh)。因为 CPU 不能用数据选择器运行代码(TDB 是数据选择器)，所以 KRNL-386 建立一个选择器别名并存入本域，这是个代码选择器。这个选择器的基地址和长度与 TDB 选择器相同，唯一的区别是别名是作为代码选择器而不是作为数据选择器建立的。你建立的前七个 MakeProInstance 转换程序的基址，其选择器部分与本域值相同。

B2h WORD 有附加转换程序段的选择器

如果要建立的 MakeProInstance 转换程序多于七个，KRNL386 就另外分配一个代码段容纳另外七个转换程序。这个段和 TDB 的 B0h 到 F1h 域格式相同。甚至需要更多转换程序时，还可以再分配附加的段，新附加段挂在比它先分配的附加段的尾部，这样形成一个链表，这个域就充当链表后向指针的角色。

B4h WORD PT 识别码(5450h)

这个域包含值 5450h，用 ASCII 字符表示就是“PT”。“PT”可能是 Procedure Thunks 或 ProcInstance Thunk 的词头缩写形式。

B6h WORD 未用

这个域未用，被置为 0。

B8h WORD 下一个可用的转换程序槽的偏移量加 6

把本域值减 6，就得到 TDB 中建立下一个 MakeProcInstance 转换程序处的偏

移量。每建立一个转换程序，它的值增大 8。

BAh　BYTE[38h]　　MakeProcInstance 转换程序区

这个地方存放七个 MakeProcInstance 转换程序。每个转换程序 8 字节长，有如下格式：

```
MOV AX, hInstance       ; hInstance == parameter 2 to MakeProcInstance
JMP FAR PTR lpfnProc    ; lpfnProc == parameter 1 to MakeProcInstance
```

F2h　char[8]　　任务的模块名

本域中是任务的模块名。这个名字是从和任务一起建立的模块数据库(HMODULE)中简单地拷贝来的。如果模块名足够 8 个字符，就没有结尾的 0 了。

FAh　TD　　识别码

这个 WORD 包含值 0x4454，用 ASCII 字符表示就是 TD(Task Database 的缩写)。IsTask 函数和其他 KRNL386 例程使用这个识别码来保证它们使用的是合法的任务数据库。

FCh　DWORD　　未用

这个 DWORD 未用，被置为 0。

100h　char[110h]　　任务的当前目录

因为 Windows 95 支持长文件名，所以任务的当前工作目录的长度有可能大于偏移量 67h 处为它保留的空间。因此，当前目录(去掉驱动器部分)存放在这个字符数组中。

210h　char[110h]　　任务的 PDB/PSP(只对 Win16 任务而言)

对于基于 Win16 的任务，这个区域存放任务的 PDB/PSP。在 TDB 中偏移量 60h 处的指针也指向这个区域。有点奇怪的是这个域大小为 110h 字节，而 Windows 95/DOS 7 之前，PSP 总是只有 100h 字节长。

关于任务的函数

我们已经知道了 Windows 95 的 16 位 TDB 是什么样的，现在介绍一些访问和管理 TDB 结构的函数。我所选择的函数都是最简单的函数。因为象核心 Windows 调度程序(Reschedule 函数)这样的函数，只要一个就可以占一个章节。

GetCurrentTask()函数

GetCurrentTask 是有关任务的函数中最基本的一个。这个函数已公开的返回

值放在 AX 寄存器中，任务链表头放在 DX 寄存器中。当前任务和任务链表头都保存在 KRNL386 的全局变量中。因为 KRNL386 的数据段是固定的且页面被锁定，所以这个函数要取得的两个变量总是在物理内存中。因此，在中断处理程序中调用本函数是完全安全的。这否认了微软公司的一个严肃警告，他们说在中断处理中可以安全调用的唯一函数是 PostMessage。你相信谁？请看下面的证据，然后作出你自己的判断。

因为 GetCurrentTask 太简单了，所以用几条汇编指令来演示这个函数比用 C 语言伪码还要简洁明了。

GetCurrentTask 的伪码：

```
PUSH    DS                      ; Save caller's DS.
MOV     DS,WORD PTR CS:[MYCSDS] ; MyCSDS is a global var kept in the
                                ; code segment that holds the selector
                                ; of KRNL386's data segment (segment 4).
MOV     AX,[CurTDB]             ; Load documented return value into AX.
MOV     DX,WORD PTR [HeadTDB]   ; Undocumented head of task list.

POP     DS                      ; Restore caller's DS.
RETF
```

IsTask()函数

IsTask 可以用来方便地校验你得到的任务句柄是不是合法的。它很不严格，仅仅是检查传进的段的偏移量 0xFA 处是不是值 0x4454。(实际上，你可以很容易地构造一个段，使它可以通过这个检查，但却不是合法的 HTASK 段。)

注意在 IsTask 中有这样一个有趣的现象：从表面看，IsTask 没有检查传进来的句柄以确认它是一个合法的全局内存句柄，你可能会想到传进一个假选择器值会引起一个 GP 错，使 Windows 中止你的应用。在这种情况下，确实发生了 GP 错，但 KRNL386 却有处理这种可能性的准备。

为处理可能发生 GP 错的地方的代码序列，KRNL386 有一个可能发生 GP 错的地址范围的表。与每个地址范围相关联的是一个安全恢复地址(safe recovery address)。如果 KRNL386 GP 错处理程序发现在一个这样的地址范围中发生了 GP 错，它就把控制传到恢复地址。对于 IsTask，恢复地址处的代码简单地把 AX 寄存器置为 0(就是 FALSE)，然后返回调用 IsTask 的代码。看上去这种机制有点象 Win32 结构化异常处理，尽管存在许多重要差别，从本质上说，它本来就是。关于结构化异常处理的 KRNL386 版本的更多信息请参看《Undocumented Windows》中关于__GP 和 HasGpHandler 入口的内容。

IsTask 的伪码：

```
// Parameters:
//      HTASK hTask
// Locals:
//      TDB far * lpTDB     // Pointer to TDB structure.

    if ( hTask == 0 )
        return FALSE;

    lpTDB = MAKELP( hTask, 0 );

    BX = *(LPWORD)MAKELP( hTask, 0x202 );   // ??? Offset 0x202 in the TDB
                                            // is near the end of the current
                                            // directory area.

    if ( lpTDB->TDB_sig == 0x4454 )         // Look for the TD signature.
        return TRUE;                        // (0x4454)
    else
        return FALSE;
```

GetTaskQueue()函数

GetTaskQueue 是一个未公开的函数，它返回与传进的 HTASK 参数相关的消息队列句柄。如果 HTASK 参数为 0，它就返回当前任务的消息队列句柄。第四章讲述了消息队列的详细内容。

GetTaskQueue 是一个很有用的函数，可以确定一个任务是否还能接收查看消息(接收处理的消息或传送消息都需要消息队列)。应用的消息队列直到它在启动代码中调用了 InitApp 函数后才建立。对 InitApp 的调用则直到指定装入的 DLL 的 LibMain 被调用后才发生，所以一个任务生存期的重要部分在没有消息队列时就可能过去了。专门用于 Windows 的调试器需要知道正在调试的程序是否有消息队列，当被调试程序停止后，这一信息对不同的调试器有所不同，这取决于调试器怎样处理被调试进程及其出错信息。

GetTaskQueue 函数不对其输入参数作严格的合法性检查。如果你传进一个不是合法选择器的非 0 值，就会接收到一个 KRNL386 内部的 GP 错。

GetTaskQueue 的伪码：

```
// Parameters:
//      HTASK hTask
// Locals:
//      TDB far * lpTDB     // Pointer to TDB structure.

    lpTDB = GetATaskSomehow( hTask );   // See following pseudocode.

    if ( lpTDB->TDB_Queue )
        return lpTDB->TDB_Queue;    // Return message queue.
    else
        return -1;                  // Windows 3.1 didn't do this, and
                                    // returned whatever was in the TDB.
```

GetTaskSomehow 的伪码：

```
// Parameters:
//  HTASK   hTask

    if ( hTask )        // If any nonzero hTask passed in, return it;
        return hTask;   // otherwise, return the current task.
    else
        return CurTDB;
```

MakeProcInstance()函数

尽管在新编译器中，好多地方已不需要 MakeProcInstance，但当你在你的 EXE 代码而非 DLL 代码中需要回调函数时，还是常常用到 MakeProcInstance。例如，你也许想使用 TOOLHELP 中的 NotifyRegister 或 InterruptRegister 回调你的 EXE 代码。如果你使用__loadds 函数修改器，就会最终把程序限制为单个实体。这时 MakeProcInstance 转换程序就成为拯救点。

MakeProcInstance 转换程序的工作很简单：把 AX 寄存器设置为回调函数要使用的 DS 寄存器值，然后跳转到指定地址。函数的起始代码应当取出 AX 寄存器值，把它放入 DS 寄存器。

MakeProcInstance 首先用参数合法性检查代码来确保传进的是一个合法的目标地址和 HINSTANCE。ValidateHInstance 和 ValidateCodePtr(本节后面有说明)的伪码列出了 MakeProcInstance 认为合法的参数：对于 ValidateHInstance 是 HANDLE，对于 ValidateCodePtr 是 FARPROC。如果这两个参数中任何一个非法，MakeProcInstance 都不会建立转换程序，并且返回。如果运行 KRNL386 的调试版，MakeProcInstance 显示一个错误代码，告诉你编程中的错误。

对参数作过合法性检查之后，MakeProcInstance 就跳转到 IMakeProcInstance 代码，它是实际上建立转换程序的地方。IMakeProcInstance 在提交建立的转换程序之前，先做一些对它自身的检查。如果 HINSTANCE 参数和发出调用的代码的 DS 寄存器值不同，就给出一个信息："MakeProcInstance only for current Instance"，意思是你不能为你拥有的 EXE 模块之外的 EXE 模块建立转换程序(就是说，在调用 MakeProcInstance 之前，你必须小心把 DS 修改成正确的值)。

另一个重要的检查是看你是不是为 DLL 中的函数申请转换程序。DLL 不需要 MakeProcInstance 转换程序，因为他们能使用固定编码(Hard-coded)DS 值的输出函数重入代码。例如：

```
MOV AX,17C7h
MOV DS,AX
```

如果你真的把 DLL 中一个例程地址传给 MakeProcInstance，它就把你传进来的地址原封不动地返回。

IMakeProcInstance 的下一个主要部分是确定在哪里建立新转换程序。如果到现在你建立的转换程序不足七个，新转换程序就会建立在当前任务的 TDB 段中的区域。否则，IMakeProcInstance 在它建立用来存储附加转换程序的附加段中搜索。如果在指向段中没有打开的槽，IMakeProcInstance 就分配另一个段(用 GlobalAlloc)，初始化该段，然后把它加入到转换程序段链表中。

一旦 IMakeProcInstance 知道了在哪里建立新转换程序，实际建立工作出奇地简单。MakeProcInstance 的剩余部分是这样的：

```
MOV AX, hInstance
JMP FAR PTR lpfnProc
```

建立转换程序不过是生成完整的指令的过程。字节 0 和 3 常用常数值(操作码 0xB8 和 0xEA)填充，偏移量 1 处的 WORD 设置成 MakeProcInstance 的 hinstance 参数的值，偏移量 4 处的 DWORD 被设置成 lpProc 参数值。

MakeProcInstance 的伪码：

```
// Parameters:
//      FARPROC     lpProc
//      HINSTANCE   hinst

    ValidateCodePtr( lpfnProc );    // If either of these functions fail,
    ValidateHInstance( hinst );     // the function returns without JMP'ing
    goto IMakeProcInstance          // to IMakeProcInstance.
```

ValidateHInstance()的伪码：

```
// Parameters (in AX):
//      HANDLE handle

    if ( handle == 0 )
        return;

    // Make sure the LDT bit is on. Win16 code only deals with LDT
    // selectors, and not with GDT selectors.
    if ( (handle & 0x0004) == 0 )
        RIP in the debug KERNEL (code 6022 - ERR_BAD_GLOBAL_HANDLE)

    if ( handle == -1 ) // Apparently -1 is allowed.
        return;
        LAR handle                  // Get access rights WORD.
        if ( LAR instruction fails )
            RIP in the debug KERNEL (code 6022 - ERR_BAD_GLOBAL_HANDLE)

    return
```

ValidateCodePtr()的伪码：

```
// Parameters ( in CX:AX ):
//      FARPROC lpfn;
// Locals:
//      WORD    opcode

    LAR SELECTOROF( lpfn )                  // Get access rights WORD.
    if ( LAR instruction fails )
        RIP in the debug KERNEL (code 7088)

    if ( Code bit (0x0008) not set in access rights )
        RIP in the debug KERNEL (code 7088)

    AL = *(LPBYTE)lpfn  // Test to see if the memory can be read. If it
                        // GP faults, the KERNEL __GP handler will catch it.

    opcode = *(LPWORD)(lpfn+2); // Grab the opcode bytes 2 bytes into the PROC.

    // Verify that the code pointer passed to us has an export prologue
    // in it. 0x581E == PUSH DS / POP AX,  0xD88C == MOV AX,DS.
    if ( (opcode != 0x581E) && (opcode != 0xD88C) )
        RIP in the debug KERNEL (code 7088);

    return;
```

IMakeProcInstance()的伪码：

```
typedef struct
{
    BYTE    mov_ax_opcode
    WORD    hinstValue;
    BYTE    jmp_far_opcode;
    DWORD   lpfn;
} MAKEPROCINSTANCE_THUNK;
// Parameters:
//      FARPROC     lpProc
//      HINSTANCE   hinst
// Locals:
//      LPMODULE    lpModule;
//      MAKEPROCINSTANCE_THUNK far * lpThunk;
//      WORD newThunkSegment;       // If additional thunk slots are needed.

    if ( hInstance )
    {
        if ( HIWORD(GlobalHandle(hinst)) != Calling application's DS. )
            _KRDebugTest("fatl K16 %dx2 MakeProcInstance only for"
                         " current instance.");
    }

    // Get the owner of the hinst segment, which should be an HMODULE,
    // and make a far pointer out of it.
    lpModule = MAKELP( FarGetOwner(hinst), 0 );
```

```
// Check if the owning segment is a valid HMODULE by looking for
// the NE signature. If HMODULE isn't valid, something is seriously wrong,
// so pop into a debugger with an INT 3.
if ( 'NE' != lpModule->ne_signature )
    INT 3

// If the owning module is a DLL, just return the FARPROC passed in.
// MakeProcInstance thunks aren't necessary for DLLs.
if ( lpModule->ne_flags & MODFLAGS_DLL )
    return lpProc;

if ( spaces left in TDB for thunk )
{
    lpThunk = MAKELP( TDB, TDB->TDB_next_MPI_thunk );
    goto InsertThunk
}

if ( space in the add-on thunk segment (offset B2h in TDB) )
{
    lpThunk = MAKELP( segment & offset of next free slot in
                      add-on segment );
    goto InsertThunk
}

// Allocate memory for a new thunk segment (0x40 bytes in size).
newThunkSegment = GlobalAlloc( GMEM_ZEROINIT, 0x40 );
if ( newThunkSegment == 0 )
    goto ReturnFailure;
    Use AllocSelector and PrestoChangoSelector to make a new code segment
    alias for the thunk segment.

    if ( AllocSelector fails )
        goto ReturnFailure;

    Initialize fields of new thunk segment to be the same format as offsets
    B0H through F1h of the Task Database. Link this new segment into
    the linked list of thunk segments. The head of this list is the
    WORD at offset 0xB2 in the current TDB.

    lpThunk = first slot in newly created thunk segment

    goto InsertThunk;

ReturnFailure:

    _KRDEBUGTEST( "err K16 MakeProcInstance failed. Did you check return"
                  " values?" );
    return 0;

InsertThunk:

    Update the nextThunk field to point at the next available slot in
    whatever segment we're putting the new thunk into.
```

```
lpThunk->mov_ax_opcode = 0xB8;
lpThunk->hinstValue = hinst;
lpThunk->jmp_far_opcode = 0xEA;
lpThunk->lpfn = lpProc;

// Return a far pointer that's a callable code address.
return MAKELP( code alias selector, OFFSETOF(lpThunk) );
```

TaskFindHandle 函数

因为许多程序员有这样的印象，TOOLHELP 函数有些神秘，所以在这一章我要讲一讲 TOOLHELP 的 TaskFindHandle 函数。就象你从伪码可以看到的，TaskFindHandle 使你可以方便地访问任务数据库中选定域。TaskFindHandle 的一个缺点是，即使你只需要知道一个特定的值，它也会把整组信息提供给你。如果你的代码对时间要求较严格，每秒要调用好多次，你就应当绕过 TaskFindHandle，直接从任务数据库中读出信息。也许有人会说你降低了兼容性，但在这个阶段，TaskFindHandle 收集的 TDB 域不会改变，否则将有太多的应用会崩溃。

就象几乎所有的 TOOLHELP 函数那样，TaskFindHandle 首先检查传进的参数，以确保它们是合法的。也就是传进来的是指向 TASKENTRY 结构的合法指针，且第一个域(dwSize)初始化为 TASKENTRY 的大小。执行完这些检查后，TaskFindHandle 调用一个内部例程，由它真正完成把 TDB 中的信息拷贝进 TASKENTRY 结构。在伪码中，我调用了这个函数，CopyTaskInformation。

为保证传进来的参数是合法的 HTASK，CopyTaskInformation 所作的唯一检查是在 TDB 偏移量为 0xFA 处的 WORD 中查找识别码 TD。你可以很容易地构造一个假段通过这个很不严格的检查。而 IsTask API 函数的防御体系一点不比它强。假设通过了 TD 识别码检查，CopyTaskInformation 的主要功能就是把 TDB 段中的域拷贝到 TASKENTRY 结构中。在例程接近尾部的地方，又把任务的栈段中栈顶、栈底和最小值拷贝到 TASKRNTRY 结构中。

CopyTaskInformation 的代码与 TOOLHELP 的 Windows 3.1 版本相比有两个改变，都与 32 位任务有关。第一个改变是由于为 Win32 进程建立的伪任务没有 HINSTANCE 段。对这些任务，TOOLHELP 用任务的 TDB 段填充 TASKENTRY.hInst 域。第二个改变与栈边界域(wStackTop 等)有关。Win32 进程的 TDB 的 SS:SP 正常情况到达的域均是 0。因此，CopyTaskInformation 不必为 Win32 任务填充 wStackTop，wStackMininum 和 wStackBottom 域。

TaskFindHandle 的伪码：

```
// Parameters:
//  TASKENTRY far * lpTask
//  HTASK           hTask

    // Verify that TOOLHELP has been initialized, that a nonzero LPTASKENTRY
    // has been passed, and that the dwSize field of the TASKENTRY struct
    // has been filled in.
    if (   (ToolhelpInitialized == FALSE)
        || ( lpTask == NULL )
        || (lpTask->dwSize != sizeof(TASKENTRY)) )
    {
        return FALSE;
    }

    // Internal function that fills in the TASKENTRY struct.
    CopyTaskInformation( lpTask, hTask );
```

CopyTaskInformation 的伪码：

```
// Parameters:
//  TASKENTRY far * lpTask
//  HTASK           hTask
// Locals:
//  LPTDB           lpTDB;

    hTask |= 1;     // If a MOVEABLE handle was passed, convert to a selector.

    Make sure the segment referenced by the hTask segment is at least
    0x204 bytes long. If not, return FALSE.

    lpTDB = MAKELP( hTask, 0 );     // Make a pointer to the TDB segment.

    if ( lpTDB->tdb_sig != 0x4454 )     // Verify TD signature is present.
        return FALSE;

    // Start filling in fields in the TASKENTRY struct, copying the data
    // from the TDB segment.

    lpTask->hNext = lpTDB->TDB_next;            // Next task.
    lpTask->hTask = hTask;                      // Current task.
    lpTask->hTaskParent = lpTDB->TDB_Parent;    // Parent task.

    lpTask->wSS = lpTDB->TDB_taskSS;            // Task's SS:SP.
    lpTask->wSP = lpTDB->TDB_taskSP;

    lpTask->wcEvents = lpTDB->TDB_nEvents;      // Number of waiting events.
    lpTask->hQueue = lpTDB->TDB_Queue;          // Message queue handle.

    lpTask->wPSPOffset = lpTDB->TDB_PSP;        // PSP/PDB of task.

    if ( lpTDB->TDB_flags & TDB_FLAGS_WIN32 )
        lpTask->hInst = hTask;  // Win32 programs don't have real HINST's.
```

```
else
    lpTask->hInst = lpTDB->TDB_HInstance;   // HINSTANCE of task.

lpTask->hModule = lpTDB->TDB_HMODULE;       // HMODULE of task.

// Copy the module name from the TDB over into the TASKENTRY struct.
memcpy( &lpTask->szModule, lpTDB->TDB_ModName, 8 )
lpTask->szModule[8] = 0;    // Null-terminate the string.

// If it's a Win32 program, don't bother to try and retrieve the
// stack bounds values listed below. Just return TRUE.
if ( lpTDB->TDB_flags & TDB_FLAGS_WIN32 )
    return TRUE;

if ( VERR lpTDB->wSS fails )    // Make sure the task's stack segment
    return TRUE;                // is accessible.

// Copy the stack boundary fields:
lpTtask->wStackTop = WORD at offset 0x0A in lpTask->wSS segment;
lpTtask->wStackMinimum = WORD at offset 0x0C in lpTask->wSS segment;
lpTtask->wStackBottom = WORD at offset 0x0E in lpTask->wSS segment;

return TRUE;
```

SHOW16 程序

我写了 SHOW16 程序来更好地解释本章中讲的概念。(SHOW16 的源码在附盘上)这个程序显示了任务表、模块表,还有关于当前选择的任务或模块的细节。另外,你可以双击细节窗口中的某行以访问关于该行的更深层的信息。

SHOW16 是一个 Windows 95 专用应用,几乎不能在其他 Win16 环境如 Windows 3.1,NT,或 OS/2 2.x 中正常运行。SHOW16 的目标是显示尽可能多的关于 Windows 95 的任务和模块的情况,而不是保持兼容性。

第一次启动 SHOW16 时,它看起来就象图 7-2 所示屏幕。左侧的列表框显示任务表(启动时缺省情况)或模块表。点击这个列表框左上角的两个单选按钮可以在这两个列表框之间切换。每次点击有关单选按钮,相应的列表就显示在下面的框中,所以这是强制刷新这两个链表的很方便的方法。

右边的列表框(细节窗口)显示左边列表框选中条目的细节信息。这些详细信息是从任务数据库或模块数据库提取的,没有通过 TOOLHELP 函数。细节窗口中前面缀有+号的条目可以被双击以修改细节窗口。如果那是一个 TDB 或 HMODULE 行,细节窗口就改变为显示你双击的 TDB 或 HMODULE 的细节。否则,细节窗口就改为显示你双击的行的更详细的报告信息。

在图 7-2 中看到的任务的一些条目需要注意。在左边列表框的任务表中,后面

跟有(Win32)的任务名是 32 位进程。在右边的列表框中,第二行显示与任务关联的 HMODULE,开头缀有一个+号。HMODULE 值传给 GetModuleFileName 以取得关联的 EXE 或 DLL 的路径,已经显示出来。如果你双击这一行,细节窗口将变成显示关于这一 HMODULE 的模块数据库细节视图(module database details view)。(模块数据库细节视图在本节稍后部分介绍。)在任务细节窗口中另一个你可以双击的行是父任务。这将使细节视图改变为父任务的细节视图。

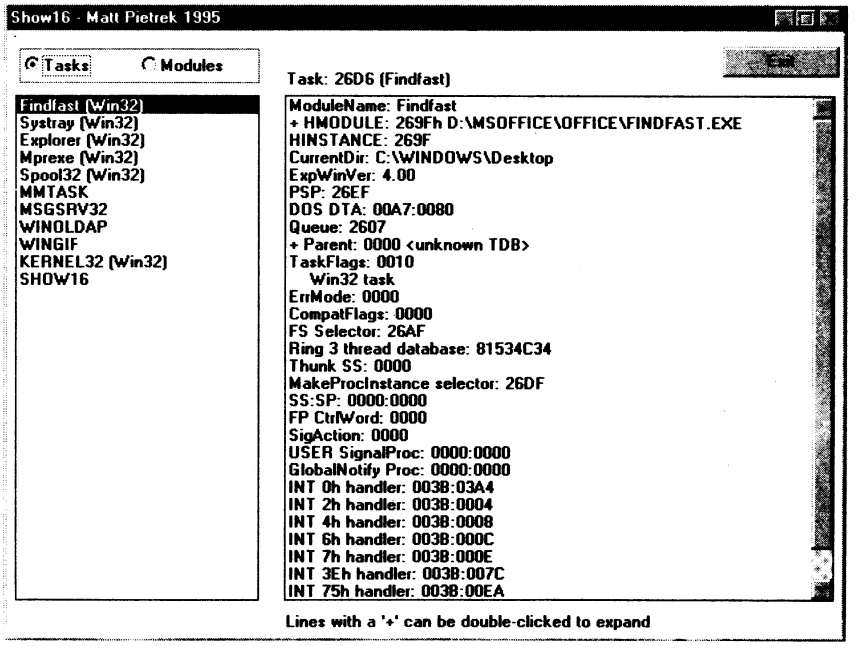


图 7-2　SHOW16 打开的屏幕包含两个列表框,提供了关于当前选择的任务或模块的信息。左边的列表框使你可以观察任务表或模块表其中之一,右边的列表框显示在那个列表中某条目的细节

任务细节窗口显示了任务数据库中任何有用的域。TDB 中没有任何用的域没有显示。另外,在前面讲述 TDB 时未用的域作了特殊处理。在显示任务细节的代码尾部是一系列语句。每个断言行作一检查,确保某一未用域被置成 0。如果这些断言中任何一个失败,就会提示该域可能被我未发现的东西使用了。

图 7-3 展示出 SHOW16 给出的其他主要信息。这是模块列表,选择了模块按钮后就得到它。左边的列表框中,第一组模块是来自 16 位 EXE 和 DLL 的一般模块数据库。根据前边讲的链表在模块数据库中游历,就能找到这些模块。在模块表尾部是为 Win32 EXE 和 DLL 建立的假模块数据库(可能需要你向下卷动窗口才能看到它们)。这些模块的模块名后缀有串"(Win32)"。因为这些模块不在一般的模块数据库列表中,所以 SHOW16 用一种粗苯而有效的方法找到它们。在 SHOW16.C 中 UpdateModuleList 函数的后部,其代码检查每一个可能的 ring 3 LDT 选择器,搜索是模块数据库的段。对找到的每一个模块数据库,检查其偏移量

0xC 处的 MODFLAGS_WIN32 标志，如果这个标志被置位，就把这个模块加到窗口的列表后面。

图 7-3 展示的模块细节窗口中有许多有趣的东西。请看“imported modules”一行。它下面的缩进行是这个模块隐含调用的 DLL。双击其中一个缩进行，就可以使模块细节窗口显示出被选模块的细节。在这个视图上面也有许多更精确的细节窗口。特别是，你可以把细节视图切换到下面的模块数据库条目：

- 段表(Segment table)
- 入口表(Entry table)
- 资源(Resources)
- 驻留名(Resident names)
- 非驻留名(Nonresident names)

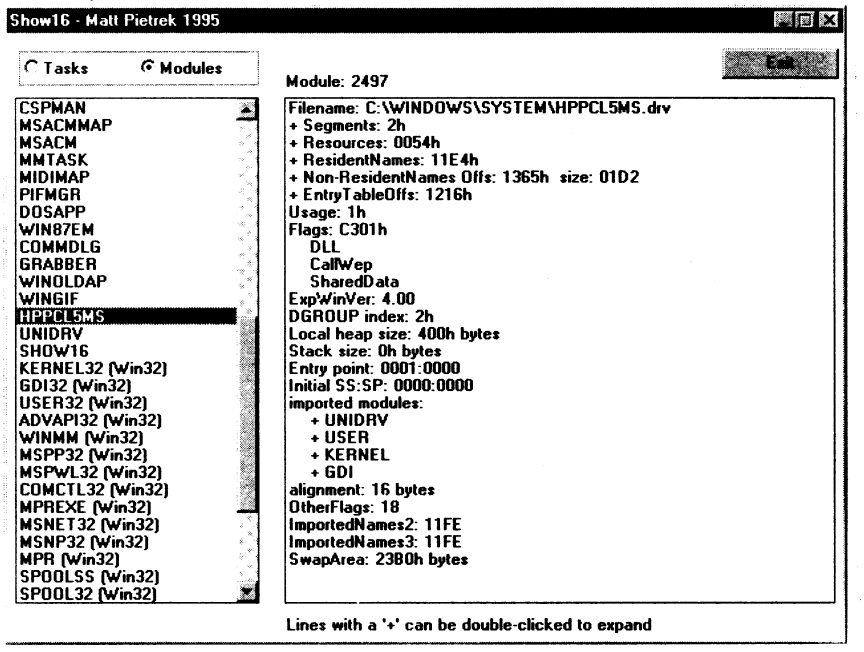


图 7-3　左边列表框中的模块列表展示一般模块数据库和假模块数据库，右边的列表框展示当前模块的细节

图 7-4 中展示一个典型的段表。对模块段表中的每个段，细节视图展示段的顺序号(逻辑地址的段部分)，全局堆句柄，类型(无论是代码段还是数据段)和大小。对此仍不满足的程序员可以修改 SHOW16 的源码，使它在这些行被双击时，给出十六进制的转化窗口。顺便说一句，字体模块没有段表，所以双击它的 segment，entrytable，resident names 或 nonresident names table 行时细节窗口不变。

图 7-5 展示一个资源视图细节窗口。可以看到，这个窗口中信息的格式与模块数据库中资源表展开后的样子相似。每一节开头是后面所跟资源的类型(如，版本信息，图标等)。紧随资源类型后的是一系列缩进行，每个缩进行表示一个资源实

例(也就是,一个点位图,一个光标等)。每个缩进行提供了关于那个资源的如下信息:资源在文件中的偏移量(单位是 Win16 模块的扇区),资源的大小(单位是 Win16 模块的扇区),资源的 ASCII 名或识别码(ID),资源的全局堆句柄(如果已装入内存的话)。SHOW16 的扩展版本可以让你双击其中每一行,从而看到资源的实际形式。

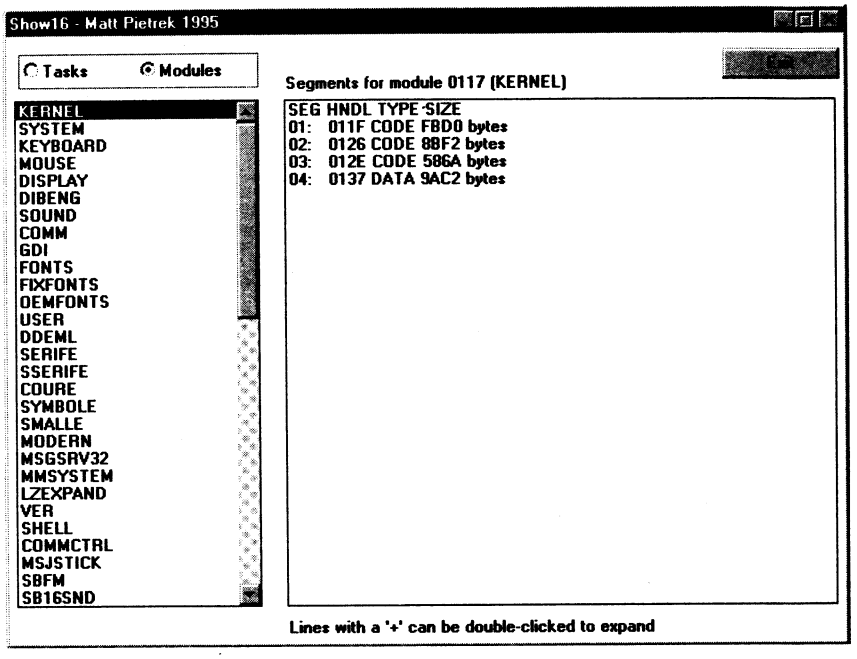


图 7-4　段窗口表的细节窗口展示每个段的顺序号、全局堆句柄、类型和大小

图 7-6 展示出驻留名表细节视图。这个视图和非驻留名的细节视图相同。在 SHOW16 代码中,这两种视图的主要区别是,非驻留名必须从磁盘文件读入,而驻留名可以直接从模块数据库中取出。驻留/ 非驻留名细节视图的每一行格式都是以引出函数或变量的引出顺序号开头,后跟其名字。驻留名视图的第一行引出顺序号为 0,这是模块名(例如,USER)。非驻留名视图的第一行引出顺序号为 0,这是对模块的描述(例如,Microsoft Windows User Interface)。

图 7-7 这是最后一个视图,这是入口表。每一行是模块入口表的一个槽,均以入口的引出顺序号和逻辑地址开始。行的其余部分包含入口的状态标志。每个入口都是可移动的或固定的,一般是引出的。显示出入口的函数名确实是好事,但这需要花费相当的时间来完成,因为绝大多数模块名常需从非驻留名表中找到,这需要申请磁盘读操作。另外,没有办法根据给出的驻留/ 非驻留名表中的引出顺序号很快地找到其名字。显示入口名的时间开销对小的 DLL 还不算啥,但对象 USER 这样的大型 DLL(它有几百个入口),你就要等待相当长一段时间。

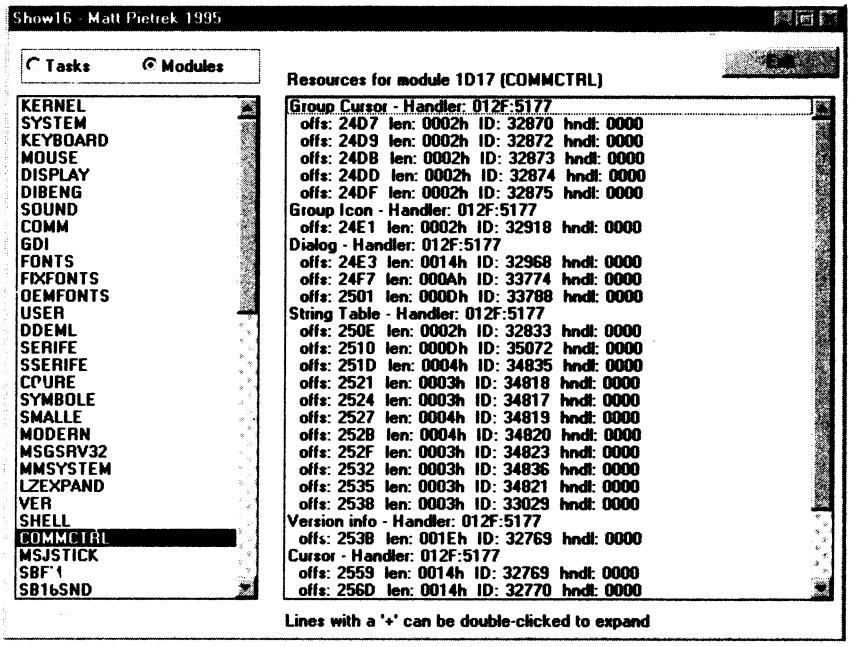


图 7-5 资源视图细节窗口提供的信息，其格式与模块数据库的资源表展开后相似

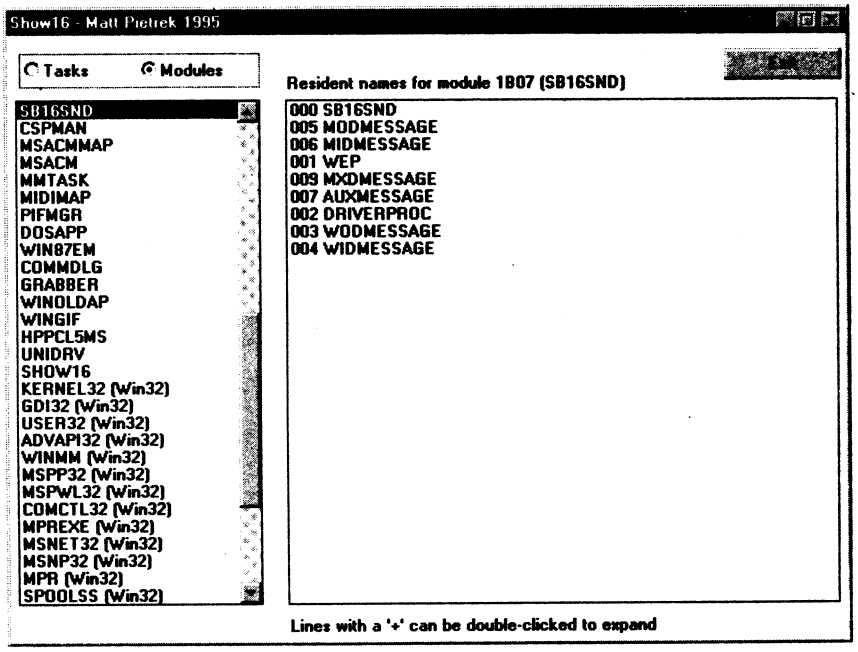


图 7-6 驻留名细节视图，这是引出函数及其顺序号，它的格式与非驻留名细节视图相同

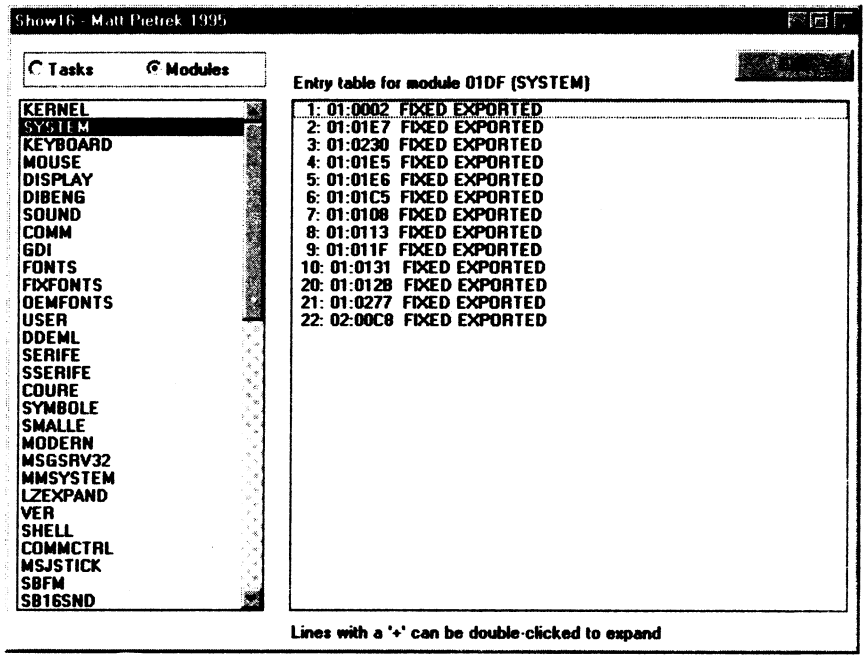


图 7-7　入口表视图，展示入口的引出顺序号、逻辑地址和状态标志

小结

尽管 Windows 95 作为 32 位操作系统而上市，但它仍有大量依赖于 16 位代码的部分。另外，直到绝大多数应用开发成为基于 32 位的之前，Windows 95 仍将主要运行 16 位程序。因此，理解 Windows 95 的 16 位部件是怎样工作的是很有用的。在这一章，我讲了两个关键的 16 位数据结构(模块和任务数据库)。我在一些地方展示了这些 16 位数据结构是怎样和它们的 32 位等价物(在第三章讲过)并行运行。尽管我没有覆盖 Windows 95 对 16 位模块和任务的支持的所有方面，但我提供的信息对大多数探密者来说都足够了。

第　八　章

可移植的执行模块和 COFF OBJ 格式

* *

一个操作系统可执行文件的格式，在很多方面上讲，是该操作系统内部采用格式和行为的反映。尽管研究一个可执行文件格式的输入输出并不是大多数程序员先要做的首要工作，但大量关于操作系统的知识可以在研究过程中看到。动态连接、装载操作和内存管理正好是通过研究可执行(文件)格式能导出的操作系统特征的三个例子。

在本章中，我将给出可移植的执行(Portable Executable，缩写为 PE)文件格式的一个真实考察，该格式是 Microsoft 为它所有 Win32 操作系统(Windows NT，Windows 95 和 Win32 系列)而设计的。

你或许会问为什么我要在本书中阐述 PE 格式，因为在 Microsoft Developer Network CD-ROM 上已有该格式的几种描述。我在这里描述 PE 格式可执行文件的主要原因，是因为在 PE 文件中用到的格式也是 Windows 95 自己内部的关键数据结构。例如，Windows 95 把一个 PE 文件的头标节映射到内存中，并用它来表示一个已装入的模块。要想理解 Windows 核心是如何工作的，你必须先弄明白 PE 格式。

我在本书中讨论 PE 文件的另一个原因是，像几乎所有 Microsoft 规范一样，Microsoft 的 PE 文献假定你只与可执行文件格式打交道。使用 Microsoft 规范的简本将便于理解。在本章中我的目标是丰富该文献内容，并把它与你每天所经历的事情相联系。以此方式我已经用了许多方法来展示 PE 格式与操作系统执行之间的相互影响。

在可预见的将来一段时期中，PE 格式在所有 Microsoft 操作系统中扮演一个关键角色。即使你正用 Visual C_{++} 在 Windows3.1 下编程，你仍是在使用 PE 文件(Visual C_{++} 的 32 位 DOS 扩展成分使用这个格式)。如果你要在 Windows 95 下进行几乎任何一种的低级系统编程，则 PE 文件的工作知识是必须具备的。

在讨论 PE 格式过程中，我不会在诸如二进制位这样的细节问题上花费精力和篇幅。而是将把主要精力花在表述 PE 文件格式中的概念上，并把它们作为你 Win32 编程的部分与你每天涉及的事物相联系。例如，线程局部变量(thread local variables)曾使我迷惑不解，直到我了解了它在可执行文件中是如何实现的时，才豁然开朗。因为许多 Win32 程序员是从 Win16 背景转过来的，因此我将把 PE 文件格式的组成结构与它们在 16 位文件格式中的相联系。

同时 Microsoft 也引入了一个不同的可执行格式，它还引入了它自身编译器和汇编器所产生的新目标模块和库格式。(这个新的 LIB 文件格式基本上就是一些 OBJ 连同一个索引集成在一起的，因此当我在这里和别处谈及 OBJ 文件时，我既引用 COFF OBJ，也用 LIB 文件。)这些新 OBJ 和 LIB 文件与 PE 格式共享了许多概念。直到最近，也还没有关于 Microsoft OBJ 和 LIB 文件公开的有效信息。甚至写到此时时也如此，该信息还是空白。因此，还是值得叙述 OBJ 和 LIB 文件格式的。

Windows NT(第一个 Win32 操作系统)继承了 VAX VMS 和 UNIX 的一些性质是很正常的。很多关键的 NT 开发者在转向 Microsoft 之前，一直为这些平台设计和编码。当设计 NT 时，他们很自然地采用以前编写并已测试过的工具，以使编程时间最短。这些工具产生和使用的可执行目标模块，被称为 COFF(Common Object File Format)。

COFF 相对旧的(就计算机产生的时期而言)自然特征在如下事实中体现：文件中的某些域以八进制格式说明。COFF 格式本身有一个好的起点，但要满足现代的操作系统，如 Windows NT 或 Windows 95，还必须加以扩展。这个扩展的结果就是 PE (即 Portable Executable)格式。被称之为可移植的(Portable)，是因为所有在各种平台(Intel386，MIPS，Alpha，Power PC，以及其它等)上实现的 NT，都采用了这种相同的可执行格式。当然，诸如像 CPU 指令的二进制译码一类，还是各不相同。你不能把 MIPS 编译过的 PE 可执行模块放在 Intel 系统上运行。然而，重要的是对一个新的 CPU，要实现这一点，不必再重写操作系统装载程序和程序设计工具了。

Microsof 委员会要使 Windows NT 崛起并快速运转的力量，从它抛弃现有的 Microsoft 32 位工具和文件格式的事实中可明显看出。在 NT 出现之前，为 Windows 3. x 编写的实际设备驱动程序，采用了不同的 32 位文件设计(LE 格式)长度。为了保持 Windows 的“如果它未被损坏，就不要维修它”的性质，Windows 95 既用 PE 格式也用 LE 格式。这就允许 Microsoft 在很大程度上可使用已有的 Windows 3. x 代码。

尽管有理由期望一个全新的操作系统(即 Windows NT)具有一个完全不同的可执行格式，但在形成目标模块(OBJ 和 LIB)格式时，还是与期望的不相同。在 Visual C++ 32 位的 1. 0 版本之前，所有 Microsoft 编译器都采用了 Intel 的 OMF(目标模块格式)说明。在 Win32 下的 Microsoft 编译器产生的是 COFF 格式的 OBJ 文件。一些 Microsoft 的竞争对手，如 Borland，选择了前述的 COFF 格式的 OBJ，并且保留了 Intel 的 OMF 格式。结果是，使用多个编译器来产生 OBJ 和 LIB 的这些公司，必须回头为不同的编译器提供单独的产品版本(如果他们还未干的话)。

喜欢探究 Microsoft 行动计谋的人，也许会看到改变 OBJ 格式的决策，正是 Microsoft 试图隐瞒对手的一个明证。为了声明在 OBJ 级上与 Microsoft 真正向下兼容，其它公司必须将他们所有的 32 位工具转换为 COFF 的 OBJ 和 LIB 格式。简言之，这种 OBJ 和 LIB 文件格式可以看成是 Microsoft 抛弃已有标准以求更好的又一例子。

PE 格式与一些 COFF 格式 OBJ 的结构定义一起在 WINNT.H 头文件中被说明。(在本章后面我将使用 WINNT.H 中的域名。)大约位于 WINNT.H 中间的是标题为"Image Format"的一节。该文件的这一节在进入较新的 PE 信息之前,以旧的 DOS MZ 格式和 NE 格式头标的一些小"精品"作为开始。WINNT.H 提供了 PE 文件所使用的原始数据结构的定义,但只包含了解释这些结构和标志含义的最起码的提示。PE 格式头文件的作者(某个 MichaelJ. o'Leary),一定是一个采用长而便于描述的名称并使用较深层结构和宏的忠实信徒。在用 WINNT.H 编码时,像下面这个表达式是不常见的:

```
pNTHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_DEBUG].VirtualAddress;
```

除了阅读关于 PE 文件由什么组成的外,你也可转储一些 PE 文件,以使你更好地理解这里提出的概念。如果你使用为 Win32 开发的 Microsoft 工具,则 Visual C++和 Win32 SDK 所带的 DUMPBIN 程序能以人工可读的形式来解剖和输出 PE 文件和 COFF OBJ/LIB 文件。DUMPBIN 甚至有一个选项来重组文件中被分离的代码段。按照 Microsoft 的版权要求,你是不允许重组它的产品的。但非常有趣的是,它又提供了使得重组它的程序和 DLL 工作非常容易的一个工具。假如重组 EXE 和 OBJ 的功能没有什么用处,那 Microsoft 为什么又将该特征含到 DUMPBIN 中呢?这听起来一定与"像我们说的哪样做,但不能做我们所做的事"如出一辙。

Borland 用户可以使用 TDUMP 来阅读 PE 文件,但是 TDUMP 不认 COFF 风格的 OBJ 文件。这并不是一个大问题,因为 Borland 编译器并不是一开始就产生 COFF 格式的 OBJ 文件。我也编写了一个 PE 和 COFF OBJ/LIB 文件的转储程序,我自认为它能提供比 DUMPBIN 更易理解的输出。尽管它不具有重组功能,但它与 DUMPBIN 的其他功能等价,并且增加了一些新特征更使得它是值得被考虑的。PEDUMP 的源代码在随本书所带的磁盘上,因此我不在这里将它们全部列出。在描述一些概念时,我将提供从 PEDUMP 输出的样本以例示这些概念。

PEDUMP 程序

PEDUMP 程序是用于转储 PE 文件和 COFF OBJ/LIB 格式文件的命令行工具。它使用了 Win32 的控制台功能以减少实现用户接口的工作量。PEDUMP 的语法如下所示:

```
PEDUMP [switches] filename
```

其开关可通过不带参地运行 PEDUMP 来看到。PEDUMP 用到如下开关:

```
/A      include everything in dump (essentially, enable all the switches)
/H      include a hex dump of each section at the end of the dump
/I      include Import Address Table thunk addresses
/L      include line number information (both PE and COFF OBJ files)
/R      show base relocations (PE files only)
/S      show symbol table (both PE and COFF OBJ files)
```

缺省时，所有开关都无效。这种情况下，你需要的大多数信息都有效，但这时你不能建立一个大数量的输出。

PEDUMP 把它的输出送到标准的输出文件（例如屏幕）上，因此它的输出可以通过在命令行中用一个>（大于号）来重定向到一个文件中。

PEDUMP 的源码也一起被包含在该盘中。PEDUMP 是用 Microsoft Visual C++ 2.0 编译器创建的，同时我也用 Borland C 4.x 将它编译过。

基本的 Win32 和 PE 概念

在进行 PE 文件设计的讨论之前，我需要先提及一些渗透于它的设计中的基本概念。为了便于讨论，我将用术语模块（module）来指已被装入内存的一个可执行文件或 DLL 的代码、数据和资源。除了程序中直接用到的代码和数据外，一个模块还由 Windows 使用的支撑数据组成，Windows 用这些支撑数据来决定代码和数据将被放在内存的何处。在 Win16 下，该支撑数据结构在模块数据库中（被一个 HMODULE 访问的段中）。在 Win32 下，该信息保存在 PE 头标中（的 IMAGE_NT_HEADERS 结构），在后面不久我将对 PE 头标进行详细解释。

对于了解 PE 文件的最重要事情是：磁盘上的可执行文件和 Windows 装载它后的模块所体现的东西非常类似。那是因为 Windows 装载器不需要做太多工作就可由磁盘文件建立一个进程。该装载器做此事颇为容易，并使用 Win32 内存映射文件来把 PE 文件的适当片段装入到一个程序地址空间中。用一个建筑上的类比，一个 PE 文件就像一座活动房屋；它有相对少的片，且每一片可以用较少的工作来移动到位，并且在一座活动房屋中进行完整的水电连接也很容易。同样，把一个 PE 文件连到 DLL 之类中也是一个简单的事情。

应用到 DLL 的装载也是同样的容易。一旦一个 EXE 或. DLL 模块被装载，Windows 能有效地把它像任何其他内存映射文件一样对待。这与 16 位 Windows 中情形形成鲜明对照。16 位 NE 文件装载器读入文件一些部分，并建立独立的数据结构来表示内存中的模块。当一个代码或数据段需要被装载时，装载器不得不从全局堆中分配一个新段，并在可执行文件中寻找生数据存储的位置，然后定位到该位置，读入这些数据，并把它用到相应的地方上。此外，每个 16 位模块要负责记住它正在使用的所有选择器，以及该段是否已被放弃等等。

然而，对于 Win32，模块为代码、数据、资源、引入表、引出表和其他等所用的所有内存是在一个连续的线性地址空间范围内的。在这种情形下，你需要知道的所有事情就是装载器把可执行文件映射到内存中的地址。你然后可很容易地通过后面的指针来寻找该模块的所有不同片段，而这些指针是作为映象的一部分被存储的。

在我们正式开始之前，你应该了解的另一概念是相对虚拟地址（Relative Virtual Address），或简写为 RVA。PE 文件中的许多域用 RVA 来描述。一个 RVA 简单地讲是一些项相对于文件映射到内存的位置的偏移量。例如，设 Windows 装载器把一个 PE 文件映射到从虚拟地址空间的地址 0x400000 开始的内存中。如果在

映象中的某个表开始于地址 0x401464，则该表的 RVA 是 0x1464：

（虚拟地址 0x401464）－（基地址 0x400000）＝ RVA 0x1464

要把一个 RVA 转换为一个指向内存的可用指针，简单地把 RVA 加上该模块被装入的基地址就行了。术语基地址(Base address)是应记住的的另一重要概念。一个基地址描述了内存映射 EXE 或 DLL 的起始地址。为了方便起见，Windows NT 和 Windows 95 用一个模块的基地址作为该模块的实例句柄(HINSTANCE)。在 Win32 中，称一个模块的基地址为一个 HINSTANCE 有点容易使人混淆，因为术语实例句柄(instance handle)来源于 16 位的 Windows。在 Win16 中，一个应用程序的每个拷贝获取它自己单独的数据段(和一个相联系的全局句柄)以与该应用程序的其它拷贝相区别；因此才有该术语：实例句柄。

在 Win32 中，应用程序拷贝不必再相互区分，因为它们不共享相同的地址空间。然而，术语 HINSTANCE 被延袭下来以使在 Win16 与 Win32 之间至少保持了表面上的连续性。对于 Win32 重要的是，你能为你过程所用的任何 DLL 调用 GetModule Handle()函数，并且得到一个指针，你能用该指针来访问模块的部件。这里所说的部件是指模块的引入和引出函数、重定位、它的代码和数据段以及其他等等。

在探讨 PE 文件和 COFF OBJ 时，另一个熟知的概念是节(section)。一个 PE 文件或 COFF OBJ 文件中的一节，大致等价于 16 位 NE 文件中的一个段或资源。节可包含代码也可含数据。一些节包含代码或你程序中声明和直接使用的数据，然而其他数据节是被连接器和库管理程序创建的，并含有某些与操作系统紧密相关的信息。在某些对 PE 格式的 Microsoft 描述中，节也被称之为目标(object)。尽管节这个术语很可能有冲突的含义，然而我仍把代码和数据区称之为节。我将在本章中后面部分更全面地讨论节的内容。至于现在，对于你重要的是要知道节是什么。

在进入 PE 文件细节之前，先看图 8-1，该图显示了一个 PE 文件的结构概况。我将对各部分逐一解释。但把它们放在一起来看看是有好处的。

PE 头标

我们的 PE 格式旅行的第一站就是 PE 头标。像所有其他 Microsoft 可执行文件格式一样，PE 文件在一个已知(或容易找到的)位置上，有一系列域来定义该文件其余部分看起来像什么。PE 头标包含了至关重要的一些信息，诸如代码和数据区的位置和大小、该文件要用什么操作系统，以及初始的堆栈大小。

和其他 Microsoft 的可执行格式相似，PE 头标也不是在文件的最前面。典型的 PE 文件的开头近百个字节被 DOS 存根所占据。该存根是一段最小化的 DOS 程序，它打印出“该程序不能在 DOS 模式下运行”之类的信息。它的目的是：如果你在一个不支持 Win32 的环境中运行一个 Win32 程序时，你将得到一个有益的错误提示。当 Win32 装载器内存映射一个 PE 文件时，该文件映射的第一个字节对应此

DOS 存根的第一字节。你启动的每个 Win32 程序的同时，也顺带装入了一个 DOS 程序。（在 Win16 中，DOS 存根不被装入内存中）。

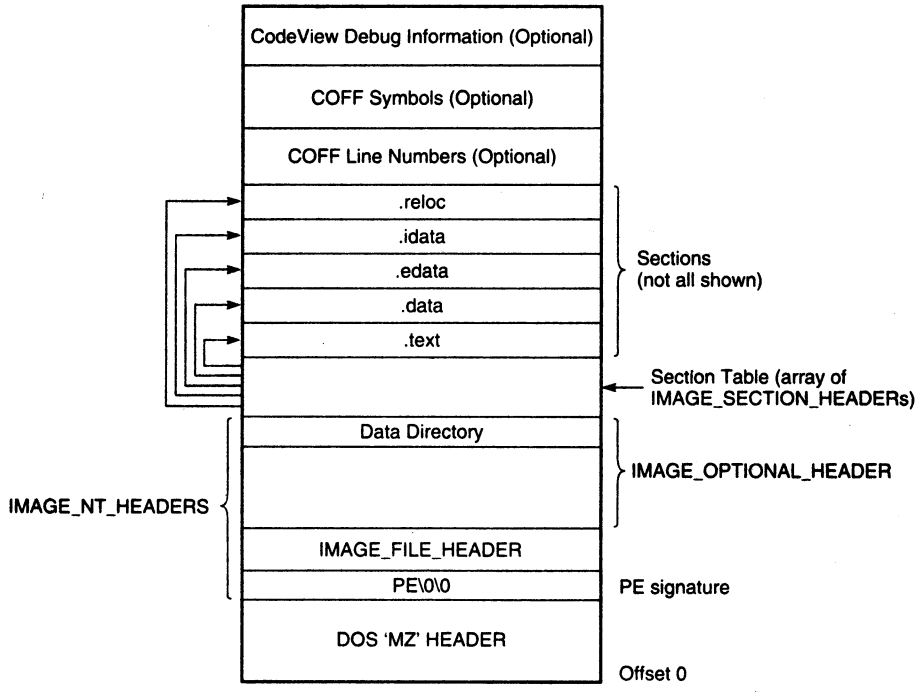


图 8-1　一个 PE 文件的结构概况

像在其他 Microsoft 可执行文件格式中一样，你可通过查它的起始偏移量来寻找真正的头标，该偏移量存于 DOS 头标中。WINNT.H 文件包含一个 DOS 存根头标的结构定义，该头标使你可很容易地查 PE 头标在何处开始。这个 e_lfanew 域是相对实际 PE 头标的相对偏移量（或 RVA）。要得到内存中一个指向 PE 头标的指针，只须将该域的值与映象的基相加：

```
// Ignoring typecasts and pointer conversion issues for clarity...
pNTHeader = dosHeader + dosHeader->e_lfanew;
```

一旦你得到一个指向主 PE 头标的指针，就可干一些真正有趣的事情了。主 PE 头标是一个 IMAGE_NT_HEADERS 类型的结构，该类型在 WINNT.H 中定义。在内存中 Windows 95 把 IMAGE_NT_HEADERS 结构作为它内存中的模块数据库。在 Windows 95 中，每个被装入的 EXE 或 DLL 都用一个 IMAGE_NT_HEADERS 结构来说明。该结构由一个 DWORD 的两个子结构组成，如下所示：

```
DWORD Signature;
IMAGE_FILE_HEADER FileHeader;
IMAGE_OPTIONAL_HEADER OptionalHeader;
```

视为 ASCII 文本文件的记号域是 PE\0\0(PE 后跟两个 0)。如果在 DOS 头标中的 e_lfanew 域在该位置上指向一个 NE 记号而不是 PE 记号,你将使用一个 Win16 NE 文件。另外,在该记号域上的一个 LE 指示一个虚拟设备驱动器(VxD)文件。在这里的一个 LX 记号则是 Windows 95 强劲竞争对手 OS/2 的文件标记。

在 PE 头标中紧随 PE 的 WORD 记号的是一个 IMAGE_FILE_HEADER 类型的结构。这个结构的域只包含了关于文件最基本的信息。该结构以不可修改的原始 COFF 形式出现。除了作为 PE 头标的一部分,它也可以出现在用 Microsoft Win32 的编译器产生的 COFF OBJ 文件的最开始处。该 IMAGE_FILE_HEADER 的域如下:

WORD　　Machine

该文件运行所要求的 CPU。有如下的 CPU ID 定义:

Intel	I386	0x14C
Intel	i860	0x14D
MIPS	R3000	0x162
MIPS	R4000	0x166
DEC	Alpha AXP	0x184
Power	PC	0x1F0 (小结尾格式)
Motorola	68000	0x286
PA RISC		0x290 (精确结构)

WORD　　NumberOfSections

在 EXE 或 OBJ 中的节数。

DWORD　　TimeDateStamp

连接器(或编译成 OBJ 的编译器)生成该文件的时间。该域值是指从 1969 年 12 月 31 日下午 4 点正开始至文件生成时之间的秒数。

DWORD　　PointerToSymbolTable

文件的 COFF 符号表的偏移量。该域只用在 OBJ 文件和带有 COFF 调试信息的 PE 文件中。PE 文件支持多种调试(debug)格式,因此调试程序应该访问数据字典中的 IMAGE_DIRECTORY_ENTRY_DEBUG 项。(数据字典在后面定义。)

DWORD　　NumberOfSymbols

在 COFF 符号表中的符号数目。参见前一个域。

WORD　　SizeOfOptionalHeader

紧跟该结构之后的一个可选头标的大小。在可执行文件中,它是紧随该结构

的 IMAGE_FILE_HEADER 结构的大小。在 OBJ 中，Microsoft 说该域可被假定总为 0。然而，在转储出 KERNEL32. LIB 引入库过程中，有一个该域值不为 0 的 OBJ，因此该说法并不完全可靠。

WORD Characteristics

关于文件的信息化标记。一些重要的域在此处描述(其它域在 WINNT. H 中定义)：

0x0001 该文件中没有重定位。
0x0002 文件是一个可执行的映象(即不是一个 OBJ 或 LIB)。
0x2000 文件是一个动态链接库，不是一个程序。

PE 头标的第三部分是一个 IMAGE_OPTIONAL_HEADER 类型结构。对于 PE 文件，这部分一定是不选的。除了标准的 IMAGE_FILE_HEADER 外，COFF 格式还允许单独定义一个附加信息结构。PE 设计者们觉得 IMAGE_OPTIONAL_HEADER 中的这个域就是除了 IMAGE_FILE_HEADER 中的基本信息之外的至关重要的信息。

IMAGE_OPTIONAL_HEADER 的所有域对你来讲，都不是非知道不可的。更重要的域是 ImageBase 和 Subsystem 域。如果你想的话，你可快速浏览或跳过对下面的这些域的描述。

WORD Magic

标识映象文件状态的一个 WORD 记号。值定义如下：

0x0107 一个 ROM 映象
0x010B 一个普通的可执行映象(大多数文件含此值)

BYTE MajorLinkerVersion
BYTE MinorLinkerVersion

生成该文件的连接器版本号。该数字以十进制形式显示，而不是十六进制。一个典型的连接器版本号是 2. 23。

DWORD SizeOfCode

所有代码段组合聚集在一起的尺寸大小。通常大多数文件只有一个代码段，因此典型情况下该域与. text 节的尺寸大小相匹配。

DWORD SizeOfInitializedData

这大概是由初始化的数据(不包括代码段)组成的所有节的总尺寸。然而，它

似乎并不与文件中初始化的数据节的尺寸大小保持一致。

DWORD　　SizeOfUninitializedData

装载器在虚拟地址空间中对它们分配空间的那些节的尺寸，但是这并不在磁盘文件中占据任何空间。这些节在程序启动时不必有一个指定值，因此有术语 uninitialized data(未初始化的数据)。未初始化的数据通常被归入称为.bss 的一节中。

DWORD　　AddressOfEntryPoint

映象开始执行位置的地址。这是一个 RVA，并且通常在.text 节中可被找到。该域对 EXE 和 DLL 都有效。

DWORD　　BaseOfcode

文件代码节开始处的 RVA。典型情况下代码节在 PE 头标之后，并在数据节之前进入内存。在 Microsoft 生成的 EXE 文件中，该 RVA 通常是 0x1000。Borland 的 TLINK32 在该域的典型值是 0x10000，因为它缺省地在 64K 范围内对齐对象，而不是像 Microsoft 连接器那样用 4K。

DWORD　　BaseOfData

文件的数据节开始处的 RVA。典型情况下数据节最后进入内存，排在 PE 头标和代码节后面。

DWORD　　ImageBase

当连接器创建一个可执行文件时，它假设该文件将被内存映射到内存中的一个指定位置上。该位置的地址就存于此域中。假设一个装载地址允许连接器进行优化，如果该文件真的被装载器内存映射到那个地址上，在代码运行前它不须做任何修补。在对基址重定向的讨论中，我还将对此谈论更多的东西。在 NT 3.1 可执行文件中，缺省的映象基址是 0x10000。对于 DLL 文件，缺省值是 0x400000。在 windows 95 中，地址 0x10000 不能用来装载 32 位 EXE 文件，因为该地址在被所有过程共享的线性地址范围内。因此，在 Windows NT 3.5 中，Microsoft 为了 Win32 可执行文件而把缺省基地址变为 0x400000。连接时基地址被赋为 0x1000 的旧程序在 windows 95 下，要花更长一点的时间来装载，因为装载器需要使用基址重定向。在后面我将会详细描述基址重定向。

DWORD　　SectionAlignment

每个节被映射到内存中时，被保证开始于是本域值倍数的虚地址处。因为分页的原因，最小节对齐值是 0x10000，该值是 Microsoft 连接器的缺省值。Borland C_{++}的 TLINK 的缺省值是 0x100000(64KB)。

DWORD　　FileAlignment

在 PE 文件中，由每个节组成的生数据被保证开始于是本域值倍数的位置处。缺省值是 0x200 字节，或许可确信节总是开始于一个磁盘扇区的起始处（一个扇区长度也是 0x200 字节）。该域等价于 NE 文件的段/资源对齐尺寸。不像 NE 文件，PE 文件一般不会有数百个节，因此因对齐文件节而浪费的空间通常非常少。

WORD　　MajorOperatingSystemVersion
WORD　　MinorOperatingSystemVersion

使用该可执行文件所要求的操作系统最小版本。该域含义有点模棱两可，因为 subsystem 域（后面的一些域）也体现类似的目的。在大多数 Win32 文件中该域为版本 1.0。

WORD　　MajorImageVersion
WORD　　MinorImageVersion

一个用户自定义域。该域允许你具有一个 EXE 或一个 DLL 的不同版本。你可用连接器的/VERSION 开关来置该域的值，例如：

```
LINK/VERSION:2.0 myobj.obj
```

WORD　　MajorSubsystemVersion
WORD　　MinorSubsystemVersion

运行该可执行文件所要求的最小子系统版本。该域的一个典型值是 4.0（意为 Windows4.0，即 Windows 95）。

DWORD　　Reserved1

似乎总为 0。

DWORD　　SizeOfImage

这似乎是装载器不得不关心的映象部分的总尺寸。它是从映象基地址开始直到最后一节的尾端这个范围的长度。最后一节的尾端是被调整为最接近节对齐值的倍数的。

DWORD　　SizeOfHeaders

PE 头标和节（对象）表的尺寸。这些节的生数据直接跟在所有头标部分之后。

DWORD　　CheckSum

大概是该文件的一个 CRC 校验和。对于其他 Microsoft 可执行格式，该域通常被忽略，并置为 0。然而，所有驱动器 DLL、在 boot 时装入的 DLL 和服务器 DLL 都

必须有一个有效的校验和。该校验和的算法可以在 IMAGEHLP. DLL 中找到。IMAGEHLP. DLL 的源码在 Win32 SDK 中提供。

WORD　　Subsystem

该可执行文件为它用户接口而使用的子系统类型。WINNT. H 定义了如下值：

NATIVE = 1	不需要子系统(例如,一个设备驱动器)
WINDOWS_GUI = 2	在 Windows GUI 子系统中运行
WINDOWS_GUI = 3	在 Windows 字符子系统中运行(一个控制台应用程序)
OS2_GUI = 5	在 OS/2 字符子系统中运行(只对 OS/2 1. x 的应用程序)
POSIX_CUI = 7	在 Posix 字符子系统中运行

WORD　　DllCharacteristics (在 NT 3. 5 中标为 obsolete)

指示什么情况下一个 DLL 的初始化函数(例如 DllMain())要被调用的标志集合。该值看起来总被置为 0,然而操作系统仍为 4 个事件调用了 DLL 被始化函数。被定义的值如下：

1 — 当 DLL 第一次被装入一个进程的地址空间时调用
2 — 当一个线程终止时调用
4 — 当一个线程启动时调用
8 — 当 DLL 退出时调用

DWORD　　SizeOfStackReserve

为初始线程栈保留的虚拟内存量。然而这些内存不是都要交付的(见后一个域)。该域缺省为 0x100000(1MB)。如果你对 CreateThread()指定一个 0 作为栈的大小,结果线程仍是得到一个与缺省值相同的栈。

DWORD　　SizeOfStackCommit

为初始线程栈首先交付的内存量。在 Microsoft 连接器中,该域缺省值是 0x1000 字节(1 页),而 TLINK 缺省为 0x2000 字节(2 页)。

DWORD　　SizeOfHeapReserve

为初始进程堆保留的虚拟内存量。该堆句柄可通过调用 GetProcessHeap()来获得。这些内存也不是都要交付的(见下一个域)。

DWORD　　SizeOfHeapCommit

在进程堆中初始交付的内存量。连接器在该域的缺省值是 0x1000 字节。

DWORD　　LoaderFlags（在 NT 3.5 中标记为 obsolete）

根据 WINNT.H，它们似乎是与调试支持有关的域。我还未见到过任何一个文件在此域具有一个有效值，也不清楚怎样使连接器对它们置位。域值定义如下：

1—在开始该进程前请求一个断点指令？
2—进程装入后在该进程上调用一个调试程序？

DWORD　　NumberOfRvaAndSizes

在 DataDiretory 数组中项的数目（见后面域的描述）。目前的工具总把该域的值置为 16。

IMAGE_DATA_DIRECTORY DataDirectory [IMAGE_NUMBEROF_DIRECTORY_ENTRIES]

一个 IMAGE_DATA_DIRECTORY 结构数组。数组中前面的元素包含了该可执行文件重要部分的起始 RVA 和尺寸。数组尾端的元素目前还未用到。数组的第一个元素总是引出函数表（如果有的话）的地址和尺寸。第二个数组项是引入函数表的地址和尺寸，如此等等。对于一个完整的数组项的定义列表，请见 WINNT.H 中的 IMAGE_DIRECTORY_ENTRY_xxx #define's。

该数组的目的是允许装载器可迅速地找到一个映象的特定节（例如引入函数表），而不必遍历映象的每一个节并逐一比较它们的名字。

数组的大多数项描述了一个完整的节的数据。然而 IMAGE_DIRECTORY_ENTRY_DEBUG 元素只含了.rdata 节中一小部分字节。在本章的“.rdata 节”部分中有对此的更多的信息。

节表

节表位于 PE 头标和映象节的生数据中间。节表包含了关于映象中的每节的信息。映象的节是以它们的地址而不是其字母来排序的。

在此处是值得弄清楚一个节到底是什么的时候了。在一个 NE 文件中，程序的代码和数据是以分别的段（segment）存于文件中的。NE 头标部分是一个结构数组，该数组与程序用到的每一个段相对应。数组中的每个结构含了一个段的信息。所存储的信息包括段的类型（代码或数据）、它的尺寸以及它在文件中其他地方的位置。在一个 PE 文件中，节表类似于 NE 文件中的段表。

然而与 NE 文件的段表又不同，一个 PE 节表并不为每个代码或数据块保存

一个选择器的值。取而代之的是，节表的每一项存储一个地址，该地址是文件的生数据被影射入内存所在位置的地址。尽管节类似于 32 位段，但它们确实不是单独的段。实际上，一个节简单地对应一个进程的虚拟地址空间中的一片内存区域。

PE 文件不同于 NE 文件的另一个方面，体现在它们是如何管理支撑数据上的，你的应用程序不使用这些支撑数据，但操作系统要用。可执行模块用到的 DLL 列表和安置表的位置是支撑数据的两个例子。在一个 NE 文件中，资源不被认为是段。尽管它们有赋给它们的选择器，但是关于资源的信息并不存于 NE 头标的段表中。取而代之的是，资源被归到靠近 NE 头标尾端的一个单独的表中，有关引入和引出函数的信息也不在它自己的段中，而是被划到 NE 头标的范围中。

用 PE 文件就不同了。任何被认为是相关的代码和数据被存储在一个节中。因此，关于引入函数的信息存储在它自己的节中，它被作为模块引出的函数表。对重定位数据也是如此。任何可能被程序或操作系统需要的代码或数据同样也是获得它们自己的节。

我还将讨论一下一些特殊的节，但是我需要先描述操作系统管理这些节所用的数据。在内存中紧跟在 PE 头标之后的是一个 IMAGE_SECTION_HEADER 数组。这个数组中的元素个数在 PE 头标中(的 IMAGE_NT_HEADER. FileHeader. NumberOfSection 域)给出。PEDUMP 程序可输出节表和所有节的域和属性。图 8-2 显示了对于一个典型的 EXE 文件 PEDUMP 输出的节表。图 8-3 显示了一个 OBJ 文件的节表输出。

```
01 .text   VirtSize: 00005AFA VirtAddr:  00001000
  raw data offs:   00000400  raw data size: 00005C00
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00009220  line #'s:      0000020C
  characteristics: 60000020
    CODE  MEM_EXECUTE  MEM_READ

02 .bss      VirtSize: 00001438  VirtAddr:  00007000
  raw data offs:   00000000  raw data size: 00001600
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:      00000000
  characteristics: C0000080
    UNINITIALIZED_DATA  MEM_READ  MEM_WRITE

03 .rdata    VirtSize: 0000015C  VirtAddr:  00009000
  raw data offs:   00006000  raw data size: 00000200
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:      00000000
  characteristics: 40000040
    INITIALIZED_DATA  MEM_READ

04 .data     VirtSize: 0000239C  VirtAddr:  0000A000
  raw data offs:   00006200  raw data size: 00002400
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:      00000000
  characteristics: C0000040
    INITIALIZED_DATA  MEM_READ  MEM_WRITE
```

```
05 .idata    VirtSize: 0000033E  VirtAddr:  0000D000
  raw data offs:   00008600  raw data size: 00000400
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:      00000000
  characteristics: C0000040
    INITIALIZED_DATA  MEM_READ  MEM_WRITE

06 .reloc    VirtSize: 000006CE  VirtAddr:  0000E000
  raw data offs:   00008A00  raw data size: 00000800
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:      00000000
  characteristics: 42000040

    INITIALIZED_DATA  MEM_DISCARDABLE  MEM_READ
```

图 8-2　由一个 EXE 文件得到的典型节表

```
01 .drectve  PhysAddr: 00000000  VirtAddr:  00000000
  raw data offs:   000000DC  raw data size: 00000026
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:      00000000
  characteristics: 00100A00
    LNK_INFO  LNK_REMOVE

02 .debug$S  PhysAddr: 00000026  VirtAddr:  00000000
  raw data offs:   00000102  raw data size: 000016D0
  relocation offs: 000017D2  relocations:   00000032
  line # offs:     00000000  line #'s:      00000000
  characteristics: 42100048
    INITIALIZED_DATA  MEM_DISCARDABLE  MEM_READ

03 .data     PhysAddr: 000016F6  VirtAddr:  00000000
  raw data offs:   000019C6  raw data size: 00000D87
  relocation offs: 0000274D  relocations:   00000045
  line # offs:     00000000  line #'s:      00000000
  characteristics: C0400040
    INITIALIZED_DATA  MEM_READ  MEM_WRITE

04 .text     PhysAddr: 0000247D  VirtAddr:  00000000
  raw data offs:   000029FF  raw data size: 000010DA
  relocation offs: 00003AD9  relocations:   000000E9
  line # offs:     000043F3  line #'s:      000000D9
  characteristics: 60500020
    CODE  MEM_EXECUTE  MEM_READ

05 .debug$T  PhysAddr: 00003557  VirtAddr:  00000000
  raw data offs:   00004909  raw data size: 00000030
  relocation offs: 00000000  relocations:   00000000
  line # offs:     00000000  line #'s:      00000000
  characteristics: 42100048

    INITIALIZED_DATA  MEM_DISCARDABLE  MEM_READ
```

图 8-3　由一个 OBJ 文件得到的典型节表

每个 IMAGE_SECTION_HEADER 是关于 EXE 或 OBJ 文件中一节的信息的一个完整数据库，它具有如下格式：

```
BYTE        Name[IMAGE_SIZEOF_SHORT_NAME]
```

这是给本节命名的一个 8 字节长的 ANSI 名。多数节名以一个小数点作为开始(例如：.text)，尽管某些 PE 资料要你相信这一点，但小数点并不是必须要求有的。你可以在汇编语言中用段指示来命名自己的节，也可以在 Microsoft C/C++编译器中用#pragma data_seg 和#pragma code_seg 来命名。(Borland C++用户应该用#pragma codeseg。)重要的是注意如果节名占满了 8 个字节，则就没有 NULL 终止字节。(Borland C++ 4.0x 的 TDUMP 忽略这个事实，并会强加无用的字节到某些 PE 的 EXE 文件上。)如果你爱用 printf()，你可使用"%.8s"，以避免拷贝名字串到另一缓冲区时误用空白符终止了它。

```
union{
DWORD PhysicalAddress
DWORD VirtualSize
}Misc;
```

根据是出现在 EXE 文件中还是在 OBJ 文件中，该域具有不同的含义。在一个 EXE 中，它保存代码或数据节的虚拟尺寸。这是调整到最接近文件对齐值倍数的尺寸。本结构中后面的 SizeOfRawData 域保存这个对齐值。有趣的是 Borland 的 TLINK32 颠倒了本域和 SizeOfRawData 域的含义，并且看起来仍是正确的连接器。对于 OBJ 文件，该域指示本节的物理地址。第一节在地址 0 上开始。要找到其下一节的物理地址，只需将 SizeOfRawData 值与当前节的此物理地址相加即可。

```
DWORD       VirtualAddress
```

在 EXE 文件中，该域存的是一个 RVA，装载器用该 RVA 将本节映射到相应地方。要计算一个给定节在内存中真正的起始地址，应将映象的基地址与本节存于该域的 VirtualAddress 值相加。用 Microsoft 工具时，第一节的本域值缺省为一个 0x1000 的 RVA。在 OBJ 文件中，该域无意义并被置为 0。

```
DWORD       SizeOfRawData
```

在 EXE 文件中，该域含本节被对齐到文件对齐尺寸后的尺寸。例如，假设一个文件对齐尺寸为 0x200。如果 VirtualSize 域指示本节长度是 0x35A 字节，则本域值为 0x400，表示本节是 0x400 字节长。在 OBJ 文件中，本域存的是编译器或汇编器确定的本节的精确长度。换句话说，对于 OBJ 文件，本节等价于 EXE 文件中的 VirtualSize 域。

```
DWORD       PointerToRawData
```

这是基于文件的偏移量，用它可以找到本节的生数据所在位置。如果你自己内

存映射一个 PE 或 COFF 文件(而不是让操作系统装载它),则该域比 VirtualAddress 更为重要。那是因为在这种情形下,你将有整个文件的一个完全线性映射,因此你将在该偏移处找到本节的数据,而不是用 VirtualAddress 域指定的 RVA。

DWORD PointerToRelocations

在 OBJ 文件中,这是一个该节重定位信息基于文件的偏移量。OBJ 每节的重定位信息直接跟在该节数据之后。在 EXE 文件中,这个域(和后一个域)无意义,并总被置为 0。当连接器创建 EXE 时,它已解决了大多数的地址分配和安排问题,只有基地址重定位和引入函数才在装载时解决。有关基地址和引入函数的信息存储在基地址和引入函数节中。因此,对一个 EXE,不需要在节的生数据之后还要有每节重定位的数据。

DWORD PointerToLinenumbers

行号表的基于文件的偏移量。一个行号表把源文件行号和一个地址对应起来,在该地址上可找到给定行产生的代码。像 CodeView 格式之类的现行调试格式,行号信息被作为调试信息的一部分来存储。然而在 COFF 调试格式中,行号信息与符号的名称/类型信息有概念上的区别。通常只有代码节(例如. text 或 CODE)才有行号。在 EXE 文件中,行号在本节生数据后被收集,直到文件的尾端。在 OBJ 文件中,一个节的行号表位于生的节数据和该节的重定位表之后。我将在本章的"COFF 调试信息"一节中讨论行号表的格式。

WORD NumberOfRelocations

本节重定向表中重定向的数目(PointerTorRelocations 域已在前面列出)。该域只用于 OBJ 文件。

WORD NumberOfLinenumbers

本节行号表中行号的数目。(PointerToLinenumbers 域已在前面列出)。

DWORD Characteristics

多数程序员将此称为标记(flag),而 COFF/PE 格式称之为特征(characteristics)。该域是指示节属性(代码/数据,可读,可写,等等)的标记的一个集合。对所有可能的节属性的列表请见 WINNT. H 中的 IMAGE_SCN_XXX_XXX #defines。一些较重要的标记在表 8-1 中列出:

表 8-1 COFF 节标记

标记	用途
0x00000020	这一节含代码。置该标记时通常和可执行标记(0x80000000)相"与"。
0x00000040	这一节含初始化过的数据。几乎除了可执行和. bss 节外的所有节都要置这个标记。

0x00000080	这一节含未初始化的数据(例如.bss 节)。
0x00000200	这一节含注释或一些其他类型的信息。这种节的一个典型用途是.drectve 节,该节是编译器产生的,它含连接器要用的命令。
0x02000000	这一节可被放弃,因为一旦它被装入后进程就不再需要它了。最常用的可被放弃节是基址重定位节(.reloc)。
0x10000000	这一节可共享。当它和一个 DLL 一起被使用时,该节中的所有数据在所有用该 DLL 的进程中被共享。对数据节该缺省值是不可共享,意为使用一个 DLL 的各个进程获得它们各自的该节数据的拷贝。从技术角度上看,一个可共享节告诉内存管理器为该节设置页映射,以便所有用一个 DLL 的进程都访问内存中相同的物理页。要使一个节可共享,则可在连接时使用 SHARED 属性。例如: LINK/SECTION:MYDATA,RWS… 告诉连接器被称为 MYDATA 的那一节应是可读、可写和可共享的。在缺省情况下,Borlamd C++的 DLL 数据段具有共享属性。
0x20000000	这一节是可执行的。当含代码标记(0x00000020)被设置时通常该标记也被设置。
0x40000000	这一节是可读的。在 EXE 文件中对所有节该标记几乎都被设置。
0x80000000	这一节是可写的。如果在一个 EXE 文件的节中该标记未被设置,则装载器将把其内存映射页标记为只读的或只执行的。具有该属性的典型节是.data 和.bss。

注意从为每个节存储的信息中所省略的东西是令人感兴趣的。首先,注意这里没有任何 PRELOAD 属性的表示。NE 文件格式要你在模块装入时,应立即给被装载的段指定一个 PRELOAD 属性。OS/2 2.0 LX 格式有类似的东西,允许你对被先装载的最多可达 8 页进行指定属性。另一方面,PE 格式却没有这个属性。基于此,我们不得不假定,Microsoft 对他们的 Win32 所实现的请求式页面调度的执行表现是自信而有把握的。

PE 格式还省去了一个中间的页查询表。在 OS/2 LX 格式中对应 IMAGE_SECTION_HEADER 的等价体不能直接指示在文件中什么地方可以找到一个节所用的代码或数据。因而一个 OS/2 LX 文件包含了一个页查询表,该表记录了在一个节中指定范围的数页在文件中的位置和属性。PE 格式不必要要这个表,它保证一个节的数据在文件中一定被连续存储。比较这两种格式,LX 方法允许有更大的自由度,但 PE 风格使用起来明显更容易和简单。因为我为这两种格式都编写了文件转储程序和反汇编程序,因此我可以保证确实是如此的。

PE 格式对旧的 NE 的格式所做的另一个受欢迎的改变是:它把各项的位置以一个简单的 DWORD 偏移量来存储。在 NE 格式中,几乎所有东西的位置都作为一个扇区值加以存储。要找到真正的文件偏移量,你必须首先在 NE 头标中查到对齐单位尺寸,并将它转换为一个扇区尺寸(典型地为 16 或 512 字节),然后再与指定的扇区偏移量相乘,以获得一个实际的文件偏移量。如果某个东西恰好不是以一个扇区偏移量来存储的话,它或许是以一个相对 NE 头标的偏移量来存储的。因为 NE 的头标不在文件的开头,你必须在你的代码中求得 NE 头标的偏移量。形成对照的是,PE 文件通过使用相对于文件被内存映射到的位置的简单偏移量,来表示

各项的位置。综上所述，PE 格式与 NE、LX 或 LE 格式相比，其使用要简单得多(假设你会使用内存映射文件)。

经常遇到的节

现在你已经有了节是什么和它们是如何被定位的概图，你还可以了解在 EXE 和 OBJ 文件中常用节的更多东西。尽管后面所列出节还不太完整，但它包括了你每天要遇到的节(即使你没有意识到它)。这些节以它们的重要性和它们被碰到的频率为序来进行描述。

.text 节

编译器或汇编器产生的所有通用码都有.text 节。因为 PE 文件运行于 32 位模式并且不受 16 位段限制，因此没有理由把来自于分离文件的代码分成分离的节。实际上连接器把来自于各个 OBJ 文件的所有.text 节连入 EXE 文件中一个大的.text 节。如果你使用 Borland C_{++}，则编译器把它的代码归到一个名为 CODE 的段。因此用 Borland C_{++}产生的 PE 文件有一个名为 CODE 的节，而不是.text 节。详细内容请见本章的“Borland C_{++}和.icode 节”一节的内容。

发现在.text 节中，除了我用编译器创建的和从运行时间库中用到的外，还有另外附加的代码时，我感到很惊奇。在 PE 文件中，当你调用另一个模块中的函数(例如 USER32.DLL 的 GetMessage())时，编译器产生的 CALL 指令并不是把控制直接传给 DLL 中的该函数。取而代之的是，该调用指令把控制传给也是在该.text 节中的一个 JMP DWORD PTR[XXXXXXXX]指令。该 JMP 指令跳转到以一个 DWORD 存于.idata 节中的一个地址上。这个.idata 节的 DWORD 含该操作系统函数入口点的真正地址，如图 8-4 所示。

在对这个问题沉思一阵后，我开始明白对 DLL 的调用为什么要用这种方法来实现。通过一个位置把所有对一个给定的 DLL 函数的调用进行归结后，装载器就没有必要对每个调用 DLL 的指令进行拚凑了。PE 装载器必须要做的只是把目标函数的正确地址放入.idata 节中的该 DWORD 中就行了。没有任何的 CALL 指令需要拚凑。这与 NE 文件有显著的不同，后者中每个段含有一个用在该段上的安置表。如果该段调了某个 DLL 函数 20 次，则装载器必须将该函数的地址拷贝到该段中 20 次。在 PE 方法下，你不能用一个 DLL 函数的真正地址来初始化一个变量。例如，你会以为如下句子：

```
FARPROC pfnGetMessage = GetMessage;
```

将会把 GetMessage 的地址放入变量 pfnGetMessage 中。在 Win16 中，这确实如此，但在 Win32 中则行不通。在 Win32 中，变量 pfnGetMessage 结果存的是在.text 节中转换了的 JMP DWORD PTR[XXXXXXXX]的地址。如果你通过函数指针来调用，结果会像你期望那样出现。然而如果你要在 GetMessage()的开始处读

这些字节，则就不那么幸运了(除非你自己做一些附加的工作来跟随.idata 的“指针”。)在“PE 文件引入”这一节中我将会再回到这个题目上来。

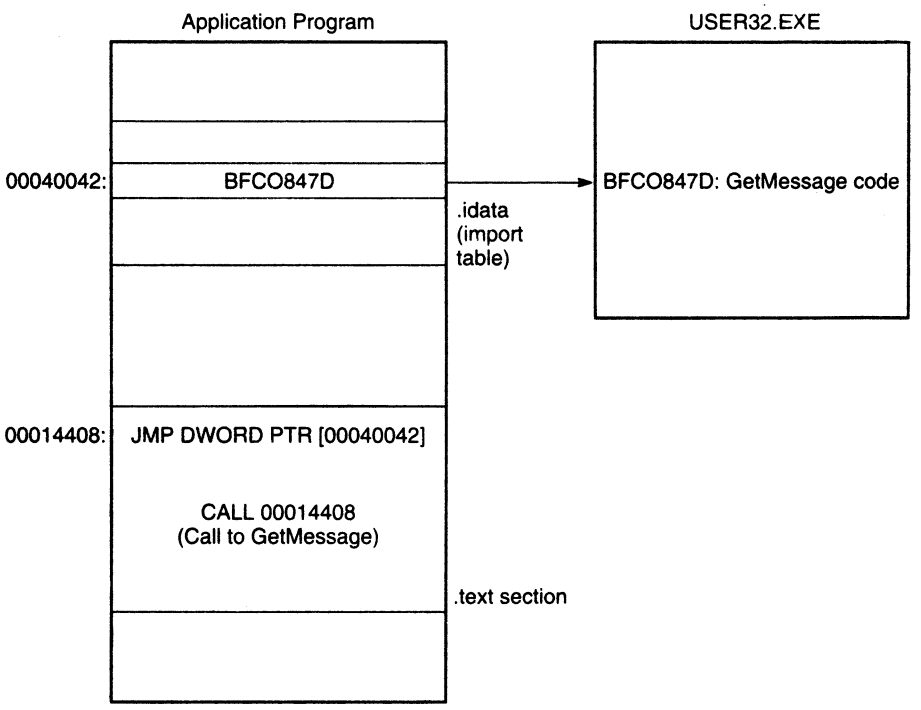


图 8-4　一个调用了引入函数的 PE 文件

在我写完本章的最初版本后，Visual C++2.0 发布了；该版本对调用引入函数引入了一个新的“孪生子”。如果你看一个 Visual C++2.0 的系统头文件(例如 WINBASE.H)，你会看到与 Visual C++1.0 头文件有所区别。在 Visual C++2.0 中，系统 DLL 中的操作系统函数原型包含一个__declspec(dllimport)，并把它作为它们的定义的一部分。在调用引入函数时，__declspec(dllimport)产生有一个很有用结果。当你调用一个用__declspec(dllimport)原型化的引入函数时，编译器并不在模块中别的地方产生一个 JMP DWORD PTR[XXXXXXXX]指令的调用，而是产生函数调用 CALL DWORD PTR[XXXXXXXX]，这里的[XXXXXXXX]地址是在.idata 节中的，它和以前用到的 JMP DWORD PTR[XXXXXXXX]是同一个地址。据我所知，目前为止，包括 Borland C++在内的另一些 C++还不具有该特征。

Borland 的 CODE 和.icode 节

Borland C++编译器和连接器没用 COFF 格式的 OBJ 文件，它采用的是 Intel OMF 格式的 32 位版本。尽管 Borland 能让编译器产生名为.text 的段，但它选择的缺省段名是 CODE。为了在 PE 文件中确定一个节名，Borland 连接器(TLINK32.

EXE)从 OBJ 文件中取出段名,并把它截为 8 个字符(如果必需的话)。因此,用 Borland C++时,PE 文件将有一个 CODE 节而不是.text 节。

节名的差别是一件小事,更重要的差别在于 Borland 的 PE 文件是怎样连到其他模块上的。正如我先前在.text 描述中提及的那样,对 OBJ 的所有调用,要执行一个 JMP DWORD PTR[XXXXXXXX]转换。在 Microsoft 系统下,这个转换是从一个引入库的.text 节到 EXE 文件。当你连接外部 DLL 时,库管理器创建引入库(并且转换)。作为一个结果,连接器并不是必定"知道"它自己产生这些转换。引入库正好是连接进 PE 文件中多出的代码和数据。

处理引入函数的 Borland 系统与 Microsoft 不同,简单地讲,它是对 16 位 NE 文件处理方法的一个扩展。Borland 连接器使用的引入库正好是函数名和它们所在的 DLL 的一个表。TLINK32 因此负责决定哪些安置是对外部 DLL 的,并负责为它产生一个恰当的 JMP DWORD PTR[XXXXXXXX]转换。在 Borland C++4.0 中,TL1ND32 把它建立的转换存于名为.icode 的节中。在 Borland C++4.02 中,TLINK32 被改成为把所有 JMP DWORD PTR[XXXXXXXX]转换并放入到 CODE 节中。

.data 节

正像.text 是代码的缺省节一样,.data 节就是初始化了的数据所在的地方。初始化了的数据由全局变量和静态变量组成,它们在编译时被初始化。它还包括字符串文字(例如,在一个 C/C++程序中的字符串"Hello World")。连接器把来自于 OBJ 和 LIB 文件的所有.data 节组合为 EXE 中的一个.data 节。局部变量是被定位在一个线程栈上的,并且在.data 或.bss 节中不占空间。

DATA 节

Borland C++对它缺省数据节采用名称 DATA。这等价于 Microsoft 编译器所用的.data 节(请见前一节".data 节"内容)。

.bss 节

.bss 节是未初始化的静态和全局变量存储的地方。连接器把来自于 OBJ 和 LIB 文件的所有.bss 节组合为 EXE 中的一个.bss 节。在节表中,为.bss 节所用的 RawDataOffset 域被置为 0,表示这一节在文件中未占任何空间。TLINK32 不产生.bss 节,代之的是它扩展 DATA 节的虚拟尺寸以说明未被始化的数据。

.CRT 节

.CRT 节是另一个初始化了的数据节,它被 Microsoft C/C++运行时间库(因

而称为.CRT)所使用。该节中的数据被用于这样一些事情:例如在 Main 或 WinMain 被执行之前调用静态 C++类的构造函数。

.rsrc 节

.rsrc 节包含了本模块所用的资源。在 NT 出现后的较早时期,16 位 RC.EXE 产生的.RES 文件输出格式不能被 Microsoft 连接器所识别。CVTRES 程序把这些.RES 文件转换成 COFF 格式的 OBJ,并把数据放入 OBJ 内部的.rsrc 节中。连接器然后才能把资源 OBJ 当做另一个 OBJ 连接进来,这意味着连接器并不必"知道"关于资源的任何特殊的东西。Microsoft 较新的连接器似乎能够直接处理.RES 文件。在本章稍后的"PE 文件资源"一节中,我将介绍资源文件节的格式。

.idata 节

.idata 节包含模块从其他 DLL 引入的函数(和数据)的信息。该节等价于 NE 文件的一个模块访问表。不同的关键点在于:PE 文件引入的每个函数特别地要在这一节中列出。要在一个 NE 文件中找等价的信息,你不得不深入到每段所用的生数据尾端处的重定位。在本章稍后的"PE 文件引入"一节中我将详细介绍该引入表的格式。

.edata 节

.edata 是被其他模块使用的 PE 文件引出的函数和数据的一个列表。NE 文件与此等价的是项表、驻留名字表和非驻留名字表这三个表的结合。不像在 Win16 中那样,几乎没有理由从一个 EXE 文件中输出任何东西,因此通常你只能在 DLL 文件中看到.edata 节。例外的是 Borland C++产生的 EXE 文件,它似乎总是引出一个函数(__GetExceptDLLinfo),该函数为运行时间库内部使用。

引出表的格式将在本章稍后的"PE 文件引出"一节中讨论。当使用 Microsoft 工具时,.edata 节中的数据通过.EXP 文件到 PE 文件中。另一方面,连接器本身不产生这些信息,而是依靠库管理器(LIB32)来扫描 OBJ 文件,并创建.EXP 文件,然后连接器把该.EXP 文件加到模块列表中以便连接。那些麻烦的.EXP 文件确实正是具有一个不同扩展名的 OBJ 文件。通过用/S(显示符号表)选项来运行 PEDUMP 程序(在本章后面提供),你可以看到从一个.EXP 中引出的函数。

.reloc 节

.reloc 节容纳了一个基址重定位(base relocation)的表。基址重定位是对指令或初始化过的变量值的一个调整;如果装载器不能把 EXE 或 DLL 文件装到连接器假定它应该放置的地址上时,该文件需要做这个调整。如果装载器能把映象装到

连接器预先确定的基地址上，则装载器将忽略该节中的重定位信息。

如果你想要进行一个选择，并且希望装载器能够总把映象装到假定的基地址上，你可使用/FIXED 选项来告诉连接器除去这个信息。尽管这样可在可执行文件中节省空间，但它可能使该可执行文件不能在别的 Win32 平台上运行。例如。让我们设你建立了一个 NT 下的 EXE 文件，并把该 EXE 基址定到 0x10000 处。如果你告诉连接器除去重定位信息，则该 EXE 将不能在 Windows 95 下运行，因为在 Windows 95 中，地址 0x10000 不是有效的(在 Windows 95 中最小的装载地址是 0x400000，即 4MB)。

注意被编译器产生的 JMP 和 CALL 指令用的是相对于其指令的偏移量，而不是在 32 位段中的实际偏移量。假如映象需要装载到与连接器所指定的基地址不同的位置上，这些指令也不需改变，因为它们用的是相对地址。结果需要重定位的并没有像你想像的那么多，通常只有使用了对某些数据的 32 位偏移量的指令才需要重定位。例如，假如你有如下的全局变量声明：

```
int i;
int *ptr = &i;
```

如果连接器给映象指定的基址是 0x10000，则变量 i 的地址结果是含像 0x12004 之类的值。在用来存指针 ptr 的内存处，连接器将写值 0x12004，因为那是变量 i 的地址。如果装载器(不管什么原因)决定在 0x70000 为基地址的地方装载该文件，i 的地址则将为 0x72004。然而，这样该预先初始化过的 ptr 变量的值就不正确了，因为 i 现在在内存中比原来高了 0x60000 字节。

这正是重定位信息发挥作用的地方。.reloc 节实际上是记录了映象中一些位置的一张表，在这些位置上，连接器所假定的装载地址和实际装载地址之间的差别需要考虑。在“PE 文件基址重定位”一节中我将谈更多关于重定位的内容。

.tls 节

当你用了编译器指示“__declspec(thread)”，则你定义的数据既不进入.data 节也不进入.bss 节，而是它的一个拷贝结果进入.tls 节。.tls 节的名字来源于术语“线程局部存储”(threak local storage)，并且它与 TlsAlloc()函数族相联系。

为了简单总结一下线程局部存储，考虑一下在一个线程基体上有全局变量的情形。即，每个线程可以有它自己的静态数据值集，而使用这些数据的代码不用考虑哪个线程正在执行。考虑一个具有工作在同一任务上，并且因而通过相同的代码来执行的几个线程程序，如果你声明一个线程局部存储变量，例如：

```
__declspec (thread) int i = 0;        // This is a global variable declaration.
```

则每个线程将透明地具有自己的变量 i 的拷贝。

通过使用 TlsAlloc、TlsSetValue 和 TlsGetValue 函数，在运行时可以明确地申请和使用线程局部存储。(第三章详细叙述了 TlsXXX 函数。)大多数情况下，在程

序中用__declspec(thread)来声明数据，比用一个 TlsAlloc()类的函数来在一个线程基体上分配内存并把一个指针存到内存中更容易一些。

关于.tls 节和__declspec(thread)变量，有一点要注意。在 NT 和 Windows 95 中，如果一个 DLL 不是动态地被 LoadLibrary()装载，则线程局部存储机制就不能用在该 DLL 中。在一个 EXE 或被隐含装入的 DLL 中，所有的都会运行良好。如果你不能对 DLL 隐含地连接，但又需要一单线程数据，你就不得不回头用 TlsAlloc()和 TlsGetValue()来动态地分配内存。注意到在运行时，实际的单线程内存块不是存于.tls 节中的这一点，是很重要的。这样，当切换线程时，内存管理器不改变被映射到模块的.tls 节的物理内存页。而.tls 节只不过存的是被用来初始化实际的单线程数据块的数据而已。单线程数据区的初始化是操作系统与编译器运行时间库两者的合作结果，当然，这还需要附加的数据——TLS 信息表——它存在.rdata 节中。

.rdata 节

.rdata 节至少可用在四件事上。第一，在被 Microsoft Link 产生的文件中，.rdata 节包含了调试信息表(OBJ 文件没有调试信息表)。在 TLINK32 的 EXE 文件中，调试信息表被存于名为.debug 的节中。调试信息表是一个 IMAGE_DEBUG_DIRECTORY 结构数组，该结构保存了存于该文件中各种调试信息的类型、尺寸和位置。三种主要的调试信息类型是：CodeView、COFF 和 FPO。图 8-5 显示了 PEDUMP 对一个典型调试信息表的输出：

```
Type            Size     Address  FilePtr  Charactr TimeData Version
--------------- -------- -------- -------- -------- -------- --------
COFF            000065C5 00000000 00009200 00000000 2CF8CF3D 0.00
(unknown)       00000114 00000000 0000F7C8 00000000 2CF8CF3D 0.00
FPO             000004B0 00000000 0000F8DC 00000000 2CF8CF3D 0.00
CODEVIEW        0000B0B4 00000000 0000FD8C 00000000 2CF8CF3D 0.00
```

图 8-5　一个典型的调试信息

调试信息表不一定能在.rdata 的开头处找到。要找到调试信息表的开始处，你必须使用一个 RVA，该 RVA 在数据信息表(数据信息表位于文件的 PE 头标部分的尾部)的第七项(IMAGE_DIRECTORY_ENTRY_DEBUG)中能找到。要决定一个 Microsoft Link 的调试信息表中的项目数，只须用一个 IMAGE_DEBUG_DIRECTORY 结构的尺寸来除调试信息表的尺寸(该尺寸可在调试信息表项的 size 域中找到)。与之对照，TLINK32 在 size 域中放的是一个实际的调试信息表的计数，而不是总的字节长度。PEDUMP 样本程序对这两种情形都能操纵。

.rdata 节第二个有用的方面是描述字符串。如果你在程序的.DEF 文件中指定了一个 DESCRIPTION 项，则这个指定的描述字符串将出现在.rdata 节中。在 NE 格式中，描述字符串总是非驻留名称表的第一项，描述字符串是用来描述文件

的一个有用的文本字符串。不幸的是，我还未找到一种简单的办法来找到它。我看见过有的 PE 文件在调试信息表之前有描述字符串，而另一些文件则在调试信息表之后。我现在还没有任何固定的方法来找到描述字符串(甚至不能肯定是否有办法)。

.rdata 节的第三种用途是用在 OLE 编程中用到的 GUID 上的。UUID.LIB 引入库包含一个 16 字节 GUID 集，后者用于诸如接口 ID 之类的东西上。这些 GUID 结果在 EXE 或 DLL 的.rdata 节中。

我认为.rdata 节的最后一个用途是作为存放 TLS(Thread Local Storage，即线程局部存储)信息表的地方。TLS 信息表是一个特殊数据结构，编译器运行时间库用它来为程序代码中声明的变量透明地提供线程局部存储。TLS 信息表的格式可在标为 Portable Executable File Format 的 MSDN(Microsoft Developer Network) CD-ROM 上找到。TLS 信息主要令人感兴趣的是指向被用于初始化线程局部存储块的数据拷贝的指针。TLS 信息表所用的一个 RVA，可以在 PE 头标数据信息表的 IMAGE_DIRECTORY_ENTRY_TLS 项中找到。用于 TLS 块初始化的实际数据存在.tls 节中。

.debug $S 和.debug $T 节

.debug $S 和.debug $T 节只出现在 COFF 的 OBJ 文件中，它们包含了 CodeView 符号和类型信息。这两个节名来源于以前的 Microsoft 编译器为此目的所用的段名($ $SYMBOLS 和 $ $TYPES)。.debug $T 节的唯一目的就是保存.PDB 文件所在的路径名，.PDB 文件为本工程中所有 OBJ 文件保存了 CodeView 类型信息。连接器用.PDB 文件来为生成的 EXE 文件建立 CodeView 信息的某些部分。

.drective 节

该节只出现在 OBJ 文件中，它包含了给连接器用的命令的文本表述。例如，在我用 Microsoft Visual C++编译器编译的任何 OBJ 文件中，如下的字符串会出现在.drectve 节中：

```
-defaultlib:LIBC -defaultlib:OLDNAMES.
```

当你在你的代码中使用了__declspec(export)，编译器会简单地把该命令行等价的东西放到.drectve 节中(例如，export:MyFunction)。

名字含 $ 的节(只对 OBJ/LIB 文件)

在 OBJ 文件中，名字含 $ 的节会被连接器特殊对待。对名字中 $ 字符前面的所有字符都相同的所有节，连接器把它们结合成一个节，而结果的节名是 $ 前面的

所有字符。因此，如果连接器遇到两个节：.idata $2 和.idata $6，它将把它们结合成一个名为.idata 的节。

节被结合的顺序由 $ 之后的字符来确定。连接器以符号序来排序，因此.idata $2 排在.idata $6 前面，而.data $A 排在.data $B 的前面。

这个 $ 约定有什么用处呢？最普遍的使用是引入库，它使用.idata(引入)节的各个部分。颇令人感兴趣的是：连接器不是一开始就非生成.idata 节不可的，最终的.idata 节主要是由 OBJ 和 LIB 文件中的节建立的，而连接器对待这些节跟对待其他要被连接的节完全一样。

各式各样的节

在用 PEDUMP 来试验的过程中，我不时地遇到了其他一些节。举一个例子，Windows 95 的 GDI32.DLL 含有一个名为_GPFIX 的数据节，该节看起来是与 GP 失败控制有关的东西。

从中得到的收获有两点。第一，不要感到被限于只能使用编译器或汇编器所提供的标准节。如果你需要一个独立的节，就毫不犹豫地使用它。在 Microsoft C/C++ 编绎器中，可使用 #pragma code-seg 和 #pragma data-seg。Borland 用户可以使用 #pragma codeseg 和 #pragma dataseg。在汇编语言中，可用一个与标准节不相同的名字来建立一个 32 位段。TLINK32 结合同一个类的代码段，因此你必须要么给每一个代码段赋一个唯一的类名，要么就关闭代码段结合。由上面讨论得到的另一点是：不常用的节名经常能够使人对一个特殊的 PE 文件的目的和实现有更深的了解。

PE 文件引入

在前面，我叙述了函数是如何对外部 DLL 文件调用的，它并不是直接调 DLL 的，代之的是，CALL 指令转向该可执行文件中的.text 节(或者.icode 节，如果你用的是 Borland C++ 4.0 的话)中其他地方上的一个 JMP DWORD PTR [XXXXXXXX]。作为一种选择，如果在 Visual C++ 中用了__declspec(dllimport)，则函数调用变成为一个“CALL DWORD PTR[XXXXXXXX]”。在这两种情况下，JMP 或 CALL 要查的地址存于.idata 节中。JMP 或 CALL 指令把控制传给该地址，该地址是所要求的目的地址。如果你仍不明白这一点，请回到前面去看图 8-4。

在被装入内存之前，PE 文件的.idata 节包含了这样一些信息：这些信息对于装载器确定目标函数的地址并把它们拼入可执行的映象中，是必不可少的。在.idata 节被装入后，它包含了一个指针，该指针指向 EXE/DLL 引入的函数。注意在本节我所讨论的所有数组和结构都被包含在.idata 节中。

.idata 节(或者是 import table，即引入表)用一个 IMAGE_IMPORT_DESCRIPTOR 的数组作为开始。对于 PE 文件隐含连接的每个 DLL，都有一个 IMAGE_IMPORT_DESCRIPTOR。对于该数组，没有任何计数来指示该数组中结构

的数目，数组的最后一个元素是通过所有域都被填入 NULL 的一个 IMAGE_IMPORT_DESCRIPTOR 来表示的。一个 IMAGE_IMPORT_DESCRIPTOR 的格式如下：

DWORD　　Characteristics/OriginalFirstThunk

该域是相对于一个 DWORD 数组的一个偏移量(RVA)。每一个这样的 DWORD 实际上是一个 IMAGE_THUNK_DATA 联合。每个 IMAGE_THUNK_DATA DWORD 对应一个被该 EXE/DLL 引入的函数。在本节中后面我将描述一下 IMAGE_THUNK_DATA DWORD 的格式。如果你运行 BIND 实用程序，则这个 DWORDS 数组被单独留下，而在另一方面，FirstThunk DWORD 数组(过一会就描述它)被修改了。

DWORD　　TimeDateStamp

该时间/日期印记指示文件是什么时间建立的。该域一般为 0。然而，Microsoft BIND 实用程序将用该 IMAGE_IMPORT_DESCRIPTOR 访问的 DLL 的时间/日期印记来修改本域的值。

DWORD　　ForwarderChain

该域与传递相关联，包括一个 DLL 把对它的一个函数的访问传递给另一个 DLL。例如，在 Windows 中，KERNEL32. DLL 把它的一些引出函数传递给 NTDLL. DLL，一个应用程序或许认为它调用了 KERNEL32. DLL 中的一个函数，但实际上它的调用进入到了 NTDLL. DLL 中。该域把一个索引含到 FirstThunk 数组(过一会要描述该数组)中，被该域索引过的函数将被传递给另一个 DLL。不幸的是，一个函数是怎样被传递的格式，在 Microsoft 文献中没有任何描述。关于传递的更多信息，请见本章中后面的“引出传递”一节。

DWORD　　Name

这是相对一个用 null 作为结束符的 ASCII 字符串的一个 RVA，该字符串含的是该引入 DLL 文件的名字(例如：KERNEL32. DLL 或者 USER32. DLL)。

PIMAGE_THUNK_DATA FirstThunk;

该域是相对一个 PIMAGE_THUNK_DATA DWORD 数组的一个偏移量(RVA)。大多数情况下，该 DWORD 被解释成指向一个 IMAGE_IMPORT_BY_NAME 结构的一个指针。然而，也有可能用顺序值引入一个函数。

一个 IMAGE_IMPORT_DESCRIPTOR 的重要部分是引入 DLL 的名字和两个 IMAGE_THUNK_DATA DWORD 数组。每个 IMAGE_THUNK_DATA DWORD 对应一个引入函数。在 EXE 文件中，这两个数组(被 Characteristics 和 FirstThunk 域所指向)并行地运行，并且都是在尾端处以一个 NULL 指针项作为终止。

为什么会有两个并行的指向 IMAGE_THUNK_DATA 结构的指针数组呢？第一个数组(被 Characteristics 指向的哪一个)被单独留下，并且绝不会被修改，它有时也被称作提示名称表(hint-name table)。第二个数组(被 IMAGE_IMPORT_DESCRIPTOR 中的 FirstThunk 域所指向)被 PE 装载器改写，装载器逐一检查每个 IMAGE_THUNK_DATA，并找到它要访问的函数的地址，然后用该引入函数的地址来改写 IMAGE_THUNK_DATA DWORD 的值。

在前面我提起过，对 DLL 函数的 CALL 调用要通过一个“JMP DWORD PTR [XXXXXXXX]”转换，该转换的[XXXXXXXX]部分要根据 FirstThunk 数组中的某一项而定。因为被装载器实际地改写了的这个 IMAGE_THUNK_DATA 数组，保存了所有引入函数的地址，因此它被称为“引入地址表”(Import Address Table)。图 8-6 显示了这两个数组。

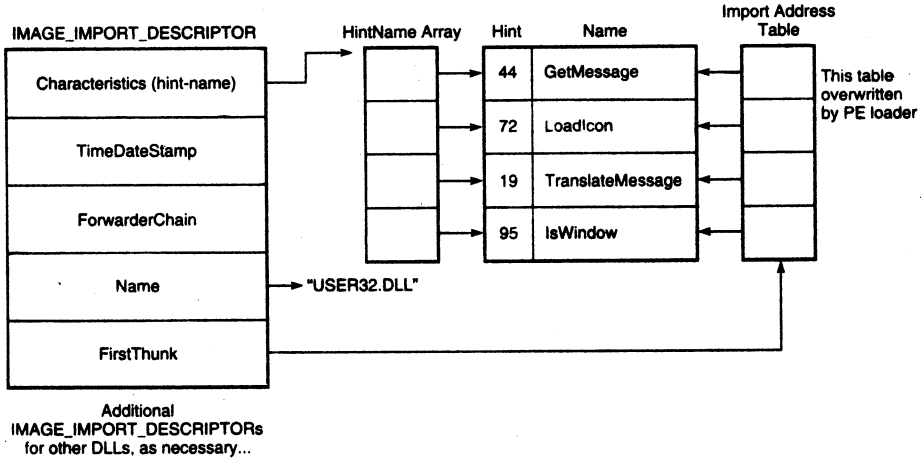


图 8-6　PE 文件怎样引入函数

对 Borland 用户，对这个描述稍微要绕点弯。被 TLINK32 生成的 PE 文件省掉了一个数组。在这样的可执行文件中，在 IMAGE_IMPORT_DESCRIPTOR(即提示名字数组)中的 Characteristics 域是 0(Win32 装载器明显不需要这个数组)。因此，只有被 FirstThunk 域指向的数组(即引入地址表)才被保证在所有 PE 文件中都是存在的。

除了在编写 PEDUMP 时我偶然碰到的一个有趣的问题外，到这里故事该结束了。在无休止的追求优化过程中，Microsoft 把 IMAGE_THUNK_DATA 数组“优化”进 Windows NT 系统 DLL(例如 KERNEL32. DLL)中。在这个优化下，IMAGE_THUNK_DATA DWORD 转而已经包含了引入函数的地址。换句话说，装载器不再需要查询函数的地址并用函数的地址来改写该转换数组了。该数组甚至在被装入之前，已经含的是引入函数的地址了。(来自 Win32 SDK 的 BIND 实用程序执行这个优化。)不幸的是，对于期望该数组含有相对于 IMAGE_THUNK_DATA 的 RVA 的 PE 转储程序而言，这就产生了一个问题。你或许会想，“但你为什

么不只用提示名字表数组呢?",除了提示名字表在 Borland 的文件中不存在外,这是一种理想的情形。PEDUMP 程序能控制这两种情形,但代码有点混乱是可以理解的。

因为该引入地址表通常是在一个可写的节中,因此可相对比较容易地截取一个 EXE 或 DLL 文件对另一个 DLL 调用。你可简单地把恰当的引入地址表项指向希望截取的函数,这不需要修改调用者和被调用者中的任何代码。这个功能是非常有用的。事实上在第十章中,我就用该技巧来建立了一个 Win32 API 的侦探程序。

注意在 Microsoft 产生的 PE 文件中,这个重要的表并不是全由连接器来合成的,而所有对调用另一个 DLL 中函数必不可少的片段驻留在一个引入库中。当你连接一个 DLL 时,库管理器(LIB. EXE)扫描正被连接的 OBJ 文件,并创建一个引入库。该引入库与被 16 位 NT 文件连接器所使用的引入库有所不同,32 位的 LIB 程序所产生的这个引入库有一个. text 节和几个. idata $ 节。该引入库中的. text 节含有我早先提到过的 JMP DWORD PTR[XXXXXXXX]替换,该替换有一个名字存于 OBJ 文件的符号表中,该符号名字和正被 DLL 引出的函数名相同(例如:_DispatchMessage@4)。

在引入库中的一个. idata 节包含了替换要反查的 DWORD。另一个. idata 节有一个空间用于"提示序数",而引入函数名紧跟其后。这两个域构成一个 IMAGE_IMPORT_BY_NAME 结构。当你稍后连接一个用了该引入库的 PE 文件时,该引入库的节将会被加到连接器需要处理的 OBJ 文件的节所组成的列表中。因为在该引入库中的替换具有和被引入的函数相同的名字,连接器认为这个替换真正就是引入函数,并且把对引入函数的调用安置到替换点上。在引入库中的替换基本上可以看成是引入函数。

除了提供一个引入函数替换的代码部分外,引入库提供了 PE 文件的. idata 节(或称为引入表)的部分东西。这些部分来自于库管理器放入引入库中的各个. idata 节。简言之,连接器并不真正知道引入函数与出现在另一个 OBJ 文件中的函数之间的差别,它只不过是遵循为建立和组合节而预先设置好的规则,并且每件事都自然而然地到了位。

IMAGE_THUNK_DATA DWORD

正如我早些时候谈到的,每个 IMAGE_THUNK_DATA DWORD 对应一个引入函数。该 DWORD 的解释根据该文件是否已被装入内存和该函数是否已通过名字或序数来引入了(通过名字更常用一些)而变化。

当一个函数是通过其序数值引入的时(少见的情形),EXE 文件的 IMAGE_THUNK_DATA DWORD 中置最高一个二进位为 1(0x80000000)。例如,考虑 GDI32. DLL 数组中一个具有值为 0x80000112 的 IMAGE_THUNK_DATA,该 IMAGE_THUNK_DATA 是引入来自于 GDI32. DLL 中的第 112 个引出函数。用序数来引入的问题是:Microsoft 不能在 Windows NT、Windows 95 和 Win32 之间使 Win32 API 函数的引出序数保持不变。

如果一个函数用名字来引入，则它的 IMAGE_THUNK_DATA DWORD 含一个 RVA，该 RVA 是 IMAGE_IMPORT_BY_NAME 结构所要用到的。一个 IMAGE_IMPORT_BY_NAME 结构非常简单，看起来像如下：

```
WORD        Hint
```

猜测是引入函数所用的引出序数之类的一个值。不像用 NF 文件那样，这个值不是非得正确不可的，装载器使用它来作为一个建议的开始值，以用于该引出函数的二分查找。

```
BYTE[?]
```

具有该引入函数名字的一个 ASCIIZ 字符串。IMAGE_THUNK_DATA DWORD 的最终解释是在 PE 文件被 Win32 装载器装入之后的。Win32 装载器使用 IMAGE_THUNK_DATA DOWRD 中的原始信息来查阅引入函数（不管是用名字还是用序数引入的）的地址。装载器然后用引入函数的地址再改写该 IMAGE_THUNK_DATA DWORD。

把 IMAGE_IMPORT_DESCRIPTOR 和 IMAGE_THUNK_DATA 并在一起

你已经看到了 IMAGE_IMPORT_DESCRIPTOR 结构和 IMAGE_THUNK_DATA 结构了，现在很容易就可构造关于一个 EXE 或 DLL 使用的所有引入函数的报表。简单地在该 IMAGE_IMPORT_DESCRIPTOR 数组（它的每一元素对应一个引入的 DLL）上反复进行如下操作就可达到目的：对每个 IMAGE_IMPORT_DESCRIPTOR，找出 IMAGE_THUNK_DATA DWRD 数组的位置并适当地解释它们。图 8-7 显示了对这个操作的 PEDUMP 输出。（无名字的函数是用序数来引入的。）

```
Imports Table:
  USER32.dll
  Hint/Name Table: 0001F50C
  TimeDateStamp:   2EB9CE9B
  ForwarderChain:  FFFFFFFF
  First thunk RVA: 0001FC24
  Ordn  Name
   268  GetScrollInfo
   133  DispatchMessageA
   333  IsRectEmpty
   431  SendMessageCallbackA
   255  GetMessagePos
    //  Rest of table omitted...
```

```
GDI32.dll
Hint/Name Table: 0001F178
TimeDateStamp:   2EB9CE9B
ForwarderChain:  FFFFFFFF
First thunk RVA: 0001F890
Ordn  Name
  31  CreateCompatibleDC
 309  SetTextColor
 276  SetBkColor
  99  ExtTextOutA
   9  BitBlt
   //  Rest of table omitted...

MPR.dll
Hint/Name Table: 0001F2F0
TimeDateStamp:   2EAF4824
ForwarderChain:  FFFFFFFF
First thunk RVA: 0001FA08
Ordn  Name
  26
  35
  34
  33
  55
   //  Rest of table omitted...

KERNEL32.dll
Hint/Name Table: 0001F1CC
TimeDateStamp:   2EB9DA61
ForwarderChain:  FFFFFFFF
First thunk RVA: 0001F8E4
Ordn  Name
 636  SetEvent
 348  GetTimeFormatA
 375  GlobalGetAtomNameA
 301  GetProcAddress
 572  RtlZeroMemory
   //  Rest of table omitted...

COMCTL32.dll
Hint/Name Table: 0001F0DC
TimeDateStamp:   2EAD4AE5
ForwarderChain:  FFFFFFFF
First thunk RVA: 0001F7F4
Ordn  Name
 152
  21  ImageList_Draw
 354
 352
  28  ImageList_GetIconSize
   //  Rest of table omitted...

ADVAPI32.dll
Hint/Name Table: 0001F0A0
TimeDateStamp:   2EA8A148
```

```
ForwarderChain:  FFFFFFFF
First thunk RVA: 0001F7B8
Ordn  Name
 149  RegQueryValueA
 119  RegCloseKey
 142  RegOpenKeyExA
 131  RegEnumKeyExA
 126  RegDeleteKeyA
  //  Rest of table omitted...
```

图 8-7　来自于 EXE 文件(EXPLORER.EXE)的一个典型引入表

PE 文件引出

引入一个函数的反过程就是引出一个函数给 EXE 或其他 DLL 文件使用。PE 文件在.edata 节中存储它引出函数的信息。一般情况下，由 Microsoft LINK 程序产生的 PE EXE 文件不引出任何东西，因此它们没有.edata 节。另一方面，TLINK32 的 EXE 文件通常输出一个符号，因此它们有一个.edata 节。大多数 DLL 文件要引出函数并且有一个.edata 节。一个.edata 节(即引出表)主要成分是：函数名称表、项目指示地址和引出序数值。在 NE 文件中，引出表的等价物就是项目表、驻留名称表和非驻留名称表。在 NE 文件中，这些表被当作 NE 头标的一部分来存储，而不是存在段或资源中。

位于.edata 节的开头处是一个 IMAGE_EXPORT_DIRECTORY 结构。紧随该结构的是由一个 IMAGE_EXPORT_DIRECTORY 结构中的域指向的数据。一个 IMAGE_EXPORT_DIRECTORY 看起来像这样：

```
DWORD       Characteristics
```

该域似乎没有被用到，并且总是被置为 0。

```
DWORD       TimeDateStamp
```

该时间/日期印记指示该文件建立的时间。

```
WORD        MajorVersion
WORD        MinorVersion
```

该域看起来也没有用，并且被置为 0。

```
DWORD       Name
```

具有该 DLL 名的一个 ASCIIZ 字符串(例如：MYDLL.DLL)的 RVA。

```
DWORD       Base
```

被本模块引出的函数的起始引出序号。例如，如果文件用序数值 10，11 和 12 来引出其函数，

则本域的值是 10。

DWORD NumberOfFunctions

AddressOfFunctions 数组中元素个数。该值也是被本模块引出的函数个数。这个值通常和 NumberOfNames 域(见下一个描述)的相同,但它们也可以不同。

DWORD NumberOfNames

在 AddressOfNames 数组中的元素个数。这个值包含了用名字来引出的函数个数,它通常(但不总是)和引出函数的总数相匹配。

PDWORD * AddressOfFunctions

该域是一个 RVA,并且指向一个函数地址数组。该函数地址是本模块中每个引出函数的 RVA。

PDWORD * AddressOfNames

该域是一个 RVA,并且指向一个字符串指针数组。该串含的是从这个模块中通过名字来引出的函数的名字。

PDWORD * AddressOfNameOrdinals

该域是一个 RVA,并且指向一个 WORD 数组。这些 WORD 基本上是从本模块中所有通过名字来引出的函数的引出序数。然而不要忘记加上在 Base 域中给出的起始引出序号。

引出表的设计有点奇怪。正如我早先提到的,引出一个函数所需要的是一个地址和一个引出序数。如果你用名字来引出函数,则应存在一个函数名。你应该想到 PE 格式的设计者会把这三个项放入一个结构中,并且再具有这些结构的一个数组。否则的话你不得不在三个分离的数组中查询各个部分。

被 IMAGE_EXPORT_DIRECTORY 指向的数组的最重要部分,是由 AddressOfFunctions 域所指向的数组。它是一个 DWORD 数组,每个 DWORD 含一个引入函数的地址(RVA)。每个引出函数的引出序数对应于数组中它的位置。例如(假定序数起始值为 1),具有引出序数为 1 的函数的地址将在该数组的第一个元素中,引出序数是 2 的函数的地址存在该数组的第二个元素中,依此类推。

关于 AddressOfFunctions 数组,有两件重要的事情要记住。第一,引出序数需要把一个 IMAGE_EXPORT_DIRECTORY 中的 Base 域的值作为基准值。如果 Base 域值为 10,则 AddressOfFunctions 数组中的第一个 DWORD 对应引出序数 10,第二项对应引出序数 11,并且依次类推。另一件要牢记的事情是引出序数可能有空白。让我们假定你明确地引出一个 DLL 中的两个函数,用序数值 1 和 3。即使你只引出两个函数,但 AddressOfFunctions 数组不得不含三个元素。在该数组中任何不与一个引出函数相对应的项,其值都为 0。

当 Win32 装载器安排一个对由序数值引入的函数的调用指令时,它只有极少

的工作需要做。装载器只简单地把函数的序数值当作一个进入目标模块的 AddressOfFunctions 数组的索引。当然，装载器必须要考虑到最低的引出序数可能不是 1，并且必须适当地调整索引。

相比这下，Win32 EXE 和 DLL 更经常用名字而不是序数来引入函数。这正是另外两个数组要发挥作用的地方，这两个数组由一个 IMAGE_EXPORT_DIRECTORY 结构中的值指向。AddressOfNames 和 AddressOfNamesOrdinals 数组是为了让装载器更快地找到与给定函数名相对应的引出序数。这两个数组都包含相同数目的元素（该数目由一个 IMAGE_EXPORT_DIRECTORY 的 NumberOfNames 域给出）。该 AddressOfFunctions 数组是一个索引值的数组，该索引将用在 AddressOfFunctions 数组中。

让我们看看装载器是如何解决一个对用名字来引入的函数的调用的。首先，装载器将查寻在 AddressOfNames 中的值指向的字符串。让我们假定在第三个元素中寻找时，找到了该字符串。其次，装载器将使用它已找到的索引值来查找 AddressOfOrdinals 数组中相应元素（在这种情况下是第三个元素）。这个数组只不过是一个 WORD 集，其中的每个 WORD 实际是用在 AddressOfFunctions 数组中的一个索引。最后一步是取出 AddressOfOrdinals 数组中的这个值，并把它作为进入 AddressOfFunctions 数组的一个索引。

在 C 代码中，寻找一个用名字来引入的函数的地址，看起来像下面的那样：

```
WORD nameIndex = FindIndexOfString( AddressOfNames, "GetMessageA" );
WORD functionIndex = AddressOfNameOrdinals[ nameIndex ];
DWORD functionAddress = AddressOfFunctions[ functionIndex - OrdinalBase ];
```

图 8-8　显示了引出节的格式和它的三个数组。

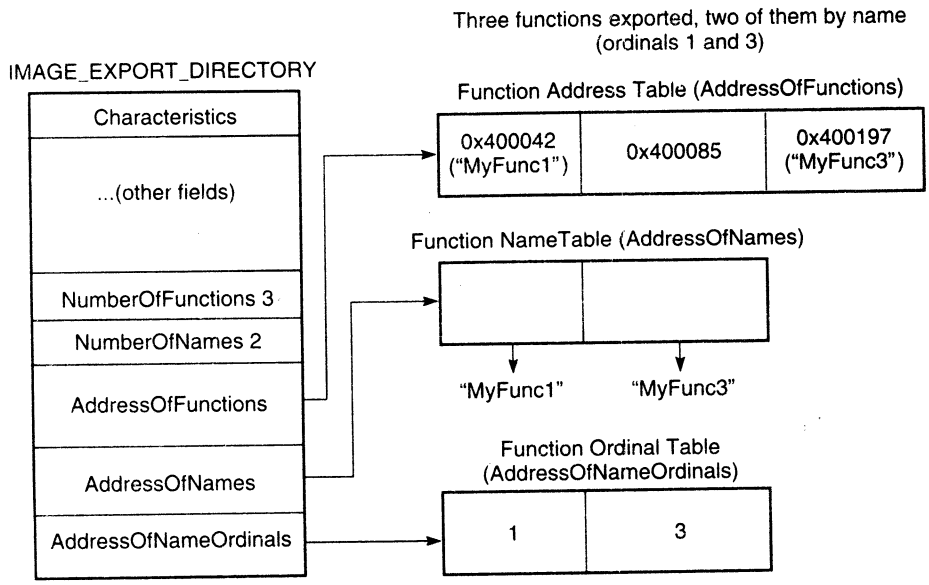


图 8-8　来自于 EXE 文件的一个典型引出表

图 8-9 显示了对 KERNEL32.DLL 引出节的 PEDUMP 输出。

```
Name:            KERNEL32.dll
Characteristics: 00000000
TimeDateStamp:   2C4857D3
Version:         0.00
Ordinal base:    00000001
# of functions:  0000021F
# of Names:      0000021F

Entry Pt  Ordn  Name
00005090     1  AddAtomA
00005100     2  AddAtomW
00025540     3  AddConsoleAliasA
00025500     4  AddConsoleAliasW
00026AC0     5  AllocConsole
00001000     6  BackupRead
00001E90     7  BackupSeek
00002100     8  BackupWrite
0002520C     9  BaseAttachCompleteThunk
00024C50    10  BasepDebugDump
// Rest of table omitted...
```

图 8-9 对 KERNEL32.DLL 引出节的 PEDUMP 输出

如果你偶然地转储从系统 DLL(例如 KERNEL32.DLL 和 USR32.DLL)引出的函数,你会发现在很多情形下,两个函数的名字只是最后一个字符不相同,例如为 CreateWindowExA 和 CreateWindowExW。这正体现了单一码支持是如何"透明地"被贯彻的。用 A 结束的函数是 ASCII(或 ANSI)兼容函数;以 W 结束的是该函数的单一码版本。在你的代码中,你不要明确地指定哪个函数要调用,而代之在 WINDOWS.H 中用预处理 #ifdef 来选择合适的函数。下面的从 NT WINDOWS.H 中的一段摘录是这样做的一个例子:

```
#ifdef UNICODE
#define DefWindowProc  DefWindowProcW
#else
#define DefWindowProc  DefWindowProcA
#endif // !UNICODE
```

引出传递

有时候一个 DLL 要引出一个函数,但函数的实际代码又存于另一个 DLL 中。在这种情形下,一个 DLL 可以把一个函数传递给另一个 DLL。当 Win32 装载器遇到对一个被传递函数的调用时,它将通过该函数去调用包含实际代码的 DLL 中的函数。

举一个例子可使这变得清楚一些。考虑下面 PEDUMP 对 Windows NT 3.5 的 KERNEL32.DLL 的输出的一段摘录：

```
00043FC3   335  HeapAlloc (forwarder -> NTDLL.RtlAllocateHeap)
00044005   339  HeapFree (forwarder -> NTDLL.RtlFreeHeap)
0004402C   341  HeapReAlloc (forwarder -> NTDLL.RtlReAllocateHeap)
0004404D   342  HeapSize (forwarder -> NTDLL.RtlSizeHeap)
0004466F   442  RtlFillMemory (forwarder -> NTDLL.RtlFillMemory)
00044691   443  RtlMoveMemory (forwarder -> NTDLL.RtlMoveMemory)
000446AF   444  RtlUnwind (forwarder -> NTDLL.RtlUnwind)
000446CD   445  RtlZeroMemory (forwarder -> NTDLL.RtlZeroMemory)
```

在这个输出中，每个函数都被传递给 NTDLL 中的一个函数。因此，调用了 HeapAlloc 的程序，事实上是调用 NTDLL.DLL 文件中的 RtlAllocateHeap。与此相似，对 HeapFree 的调用，事实上是对 NTDLL 文件的 RtlHeapFree 函数的调用。

如果一个函数被传递，那又怎样来表示呢？一个函数被传递的唯一指示是：该函数的地址被放入引出表(即.edata 节)中。这种情形下，被这样调用的函数地址真是一个 RVA，该 RVA 相对于包含被传递的 DLL 和函数名的一个串。例如，在先前的输出中，HeapAlloc 的 RVA 是 0x43FC3。KERNEL32.DLL 中的偏移量 0x43FC3 是在.edata 节里面。在 KERNEL32.DLL 文件中偏移量是 0x43FC3 处的是字符串 NTDLL.RtlAllocateHeap。PEDUMP 程序中的 DumpExportsSection 函数显示了传递的函数可如何来识别。

尽管引出传递看起来像是一个真正漂亮的特征，Microsoft 还是没有描述你怎样才能把传递用于你自己的 DLL 中。还有，直到今天，我也才只看到一个 DLL(在刚才提及的 Windows NT 的 KERNEL32.DLL)用到了传递。尽管我在 Windows 95 中还没有见着任何用到传递的 DLL，但 Windows 95 装载器是支持这个功能的，正如我在第三章中显示的那样。

PE 文件资源

与 NE 文件相比，在 PE 文件中寻找资源要更复杂一些。单个资源(例如一个菜单)的格式与 NE 文件相比，没有特别大的改变，但你需要遍历一个复杂的层次结构才能找到它们。

资源目录结构很像硬盘的目录。它有一个主目录(根目录)，主目录含有子目录，子目录还可有它自己的子目录。在这些子目录中你可以找到文件。这些文件类似于生资源数据，后者含诸如一个 IMAGE_EXPORT_DIRECTORY 类型的结构。该结构格式如下：

DWORD　　Characteristics

理论上讲，这个域应该保存该资源的标记，但看起来它总是为 0。

DWORD　　TimeDateStamp

这个时间/日期印记描述该资源创建时间。

WORD MajorVersion
WORD MinorVersion

理论上讲，这些域保存该资源的版本号。但这些域似乎总被置为 0。

WORD NumberOfNameEntries

使用名字并且跟在本结构之后的数组元素(稍后描述)的数目。要了解更多信息请见对 DirectoryEntries 域的描述。

WORD NumbeOfIdEntries

使用整数 ID 并且跟在结构和任何有命名的项之后的数组元素的数目。更多的信息请见后面的对 DirectoryEntries 域的描述。

IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[]

该域形式上并不是 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构的部分，而是紧跟其后的一个 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构数组。该数组中元素个数是 NumberOfNamedEntries 和 NumberOfIdEntries 域之和。含有名称标识符(而不是整数 ID)的目录项元素位于数组的最前面。

一个目录项既可指向一个子目录(即另一个 IMAGE_RESOURCE_DIRECTORY_ENTRY)，也可指向一个 IMAGE_RESOURCE_DIRECTORY_ENTRY，后者描述了在文件中什么地方可以找到资源的生数据。一般情况下，在你到达一个给定资源的 IMAGE_RESOURCE_DATA_ENTRY 之前，至少有三个目录层。最顶层目录(只有一个)总位于资源节(.rsrc)的开始处。最顶层目录的子目录对应于文件中能找到的资源的各种类型。例如，如果一个 PE 文件包括对话、字符串表和菜单，则这三个子目录将会是一个对话目录、一个字符串表目录和一个菜单目录。每一个这样的“类型”子目录将轮流具有“ID”子目录。对一种给定资源类型的一个实例，将有一个 ID 子目录。在上面的例子中，如果有 4 个对话框，则对话目录将有 4 个 ID 子目录。每个 ID 子目录将有一个字符串名称(例如：MyDialog)或整数 ID，这整数 ID 是用来标识.RC 文件中资源的。图 8-10 以一个更易理解的可视形式显示了资源目录层次结构。

图 8-11 显示了 PEDUMP 对 Windows NT 3.5 的 CLOCK.EXE 中的资源的输出。看看犬牙交叉状结构的第二层，你可以看到其中有图标、菜单、对话、字符串表、组图标和版本资源。在第三层上，有两个图标(具有标识 1 和 2)，两个菜单(具有名字 CLOCK 和 GENERICMENU)，两个对话(一个被命名为 ABOUTBOX，另一个具有整数 ID 0x64)，以及其它等等。在第 4 层，图标 1 的数据是：位于 RVA 0x9754 处并且是 0x130 字节长，类似，CLOCK 菜单的数据是位于偏移量 0x952C 处并且占据 0xEA 字节。

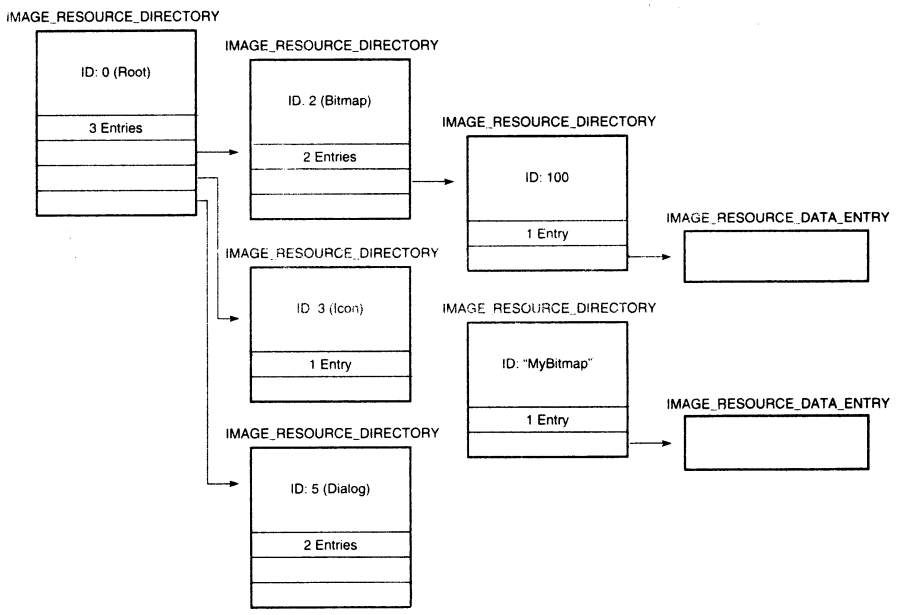


图 8-10　一个典型的 PE 文件资源层次结构

每个资源目录项是一个 IMAGE_RESOURCE_DIRECTORY_ENTRY 类型的结构。每个 IMAGE_RESOURCE_DIRECTORY_ENTRY 具有如下格式：

DWORD　　Name

该域包含的既可是一个整数 ID，也可是指向含一个字符串名字的结构的指针。如果高位(0x80000000)是 0，该域被解释为一个整数 ID。如果高位非零，则低的 31 个二进位是相对于一个 IMAGE_RESOURCE_DIR_STRING_U 结构的偏移量(相对资源节的开始处)。这个结构含一个 WORD 字符计数，后跟一个具有资源名称的单一码字符串。是的，即使是为非单一码 Win32 而设计的 PE 文件，也在这里使用单一码。要把该单一码字符串转换为一个 ANSI 字符串，请见 WideCharToMultiByte()函数。

DWORD　　OffsetToData

该域既可是相对于另一个资源目录的一个偏移量，也可是指向关于一个特定资源实例的信息的一个指针。如果高位(0x80000000)被置为 1，该目录项对应一个子目录，低 31 个二进位是一个相对于另一个 IMAGE_RESOURCE_DIRECTORY 结构的偏移量(相对于资源的开始处)。如果高位置为 0，则低 31 位是一个相对于一个 IMAGE_RESOURCE_DATA_ENTRY 结构的偏移量(相对于该资源节)。IMAGE_RESOURCE_DATA_ENTRY 结构包含了资源的生数据的位置、它的尺寸和它的代码页。

```
Resources
ResDir (0) Named:00 ID:06 TimeDate:2E601E3C Vers:0.00 Char:0
    ResDir (ICON) Named:00 ID:02 TimeDate:2E601E3C Vers:0.00 Char:0
        ResDir (1) Named:00 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
            ID: 00000409  DataEntryOffs: 000001E0
            Offset: 09754  Size: 00130  CodePage: 0
        ResDir (2) Named:00 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
            ID: 00000409  DataEntryOffs: 000001F0
            Offset: 09884  Size: 002E8  CodePage: 0
    ResDir (MENU) Named:02 ID:00 TimeDate:2E601E3C Vers:0.00 Char:0
        ResDir (CLOCK) Named:00 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
            ID: 00000409  DataEntryOffs: 00000200
            Offset: 0952C  Size: 000EA  CodePage: 0
        ResDir (GENERICMENU) Named:00 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
            ID: 00000409  DataEntryOffs: 00000210
            Offset: 09618  Size: 0003A  CodePage: 0
    ResDir (DIALOG) Named:01 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
        ResDir (ABOUTBOX) Named:00 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
            ID: 00000409  DataEntryOffs: 00000220
            Offset: 09654  Size: 000FE  CodePage: 0
        ResDir (64) Named:00 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
            ID: 00000409  DataEntryOffs: 00000230
            Offset: 092C0  Size: 0026A  CodePage: 0
    ResDir (STRING) Named:00 ID:02 TimeDate:2E601E3C Vers:0.00 Char:0
        ResDir (1) Named:00 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
            ID: 00000409  DataEntryOffs: 00000240
            Offset: 09EA8  Size: 000F2  CodePage: 0
        ResDir (2) Named:00 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
            ID: 00000409  DataEntryOffs: 00000250
            Offset: 09F9C  Size: 00046  CodePage: 0
    ResDir (GROUP_ICON) Named:01 ID:00 TimeDate:2E601E3C Vers:0.00 Char:0
        ResDir (CCKK) Named:00 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
            ID: 00000409  DataEntryOffs: 00000260
            Offset: 09B6C  Size: 00022  CodePage: 0
    ResDir (VERSION) Named:00 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
        ResDir (1) Named:00 ID:01 TimeDate:2E601E3C Vers:0.00 Char:0
            ID: 00000409  DataEntryOffs: 00000270
            Offset: 09B90  Size: 00318  CodePage: 0
```

图 8-11　CLOCK.EXE 的资源层次结构

要更深入探讨资源格式，我就必须要讨论单一资源类型(对话、菜单等等)的格式。谈论这些题目可以很容易地占满一章的篇幅。如果你感兴趣的话，可以阅读 Win32 SDK 带的 RESFMT.TXT 文件，它有对所有资源类型格式的详细描述。PEDUMP 程序显示了资源的层次结构，但不能分解单个资源实例。

PE 文件基址重定位

当连接器创建一个 EXE 文件时，它对文件将会被映射到内存的什么地方作一个假定，并且然后把这些假定的代码和数据项的地址放入可执行文件中。如果该可执行文件结果被装到虚拟地址空间中别的地方上，则连接器放入进映象中的

这些地址就不正确了。存于.reloc 节中的信息允许 PE 装载器较正被装载模块中的这些地址。如果装载器能够把文件装到连接器假定的基地址上的话，则.reloc 节数据就不需要了，并且会被忽略。在.reloc 节中的项被称为基址重定位，是因为这些项的使用要依赖被装映象的基地址。

与 NE 文件格式中的重定位不同，PE 文件基址重定位非常简单。它们不用访问外部 DLL，甚至不用该模块中的其他节。基址重定位工作量减至到与对一个该映象中位置列表的操作相对应的程度。

这里举一个例子，以显示基址重定位是如何工作的：让我们设一个可执行文件在连接时假定的基地址是 0x400000，在该映象内部偏移量为 0x2134 处是一个指针，该指针含的值是 0x404002。然后你装载此文件，但装载器决定需要把它映射到起始物理地址是 0x600000 的地方。连接器假定的装载基地址和实际装载地址之间的差称为基差。在这种情形下，该基差是 0x200000(0x600000 -0x400000)。因为整个映象在内存中比原来都高了 0x200000 字节，因此这个串也如此(现在在地址为 0x604002 的地方处)。指向该串的指针现在也不正确，它需要把这个基差(0x200000)加上去，以使它重新成为正确的值。

为了让 Windows 装载器做这个调整，可执行文件包含了一个对该指针所驻留的内存位置(该映象中的 0x2134 偏移处)的一个基址重定位。为了解决一个基址重定位，装载器把基差值加到位于该基址重定位地址上的原始值上。在本例中，装载器将把 0x200000 加给该原始指针值(0x404002)，并且把结果(0x604002)存回到指针的存储单元中。因为该串真的是在 0x604002 处，所有的都再一次正确了。图 8-12 显示了这个过程。

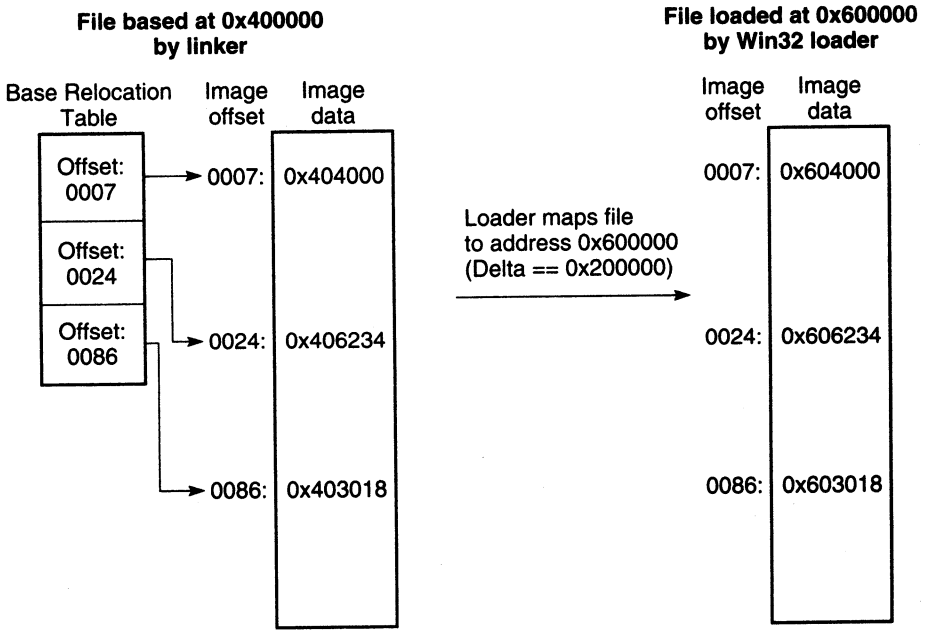


图 8-12　PE 文件基址重定位

基址重定位数据的形成有点特别。重定位信息被打包进一系列连续可变长度的块中。每一块描述映象中一个 4K 页的重定位，并用一个 IMAGE_BASE_RELOCATION 作为开始，该结构着起来像这样：

DWORD　　VirtualAddress

该域含本重定位块的起始 RVA。其后的每个重定位的偏移量被加上该值以形成实际的 RVA，这个实际 RVA 指示的地方需要运用重地位。

DWORD　　SizeOfBlock

该结构的尺寸加上其后的所有 WORD 重定位数目。为了确定该块中重定位的数目，应从该域的值中减去一个 IMAGE_BASE_RELOCATION 的尺寸(8 字节)，并且然后除以 2(一个 WORD 的尺寸)。例如，如果该域值为 44，则有 18 个重定位：

```
( 44 - sizeof(IMAGE_BASE_RELOCATION) ) / sizeof(WORD) = 18
```

WORD　　TypeOffset

这不是一个单一的 WORD，而是一个 WORD 数组，该数组的数通过前一个 DWORD 描述中的公式来计算。每 WORD 的低 12 位是一个重定位偏移量，它需要与来自该重定位块头标的 VirtualAddress 域的值相加。每个 WORD 的高 4 位是一个重定位类型。对于在 Intel CPU 上运行的 PE 文件，你只能见到两种重定位：

- 0(IMAGE_REL_BASED_ABSOLUTE)：这个重定位是无含义的，并且只被用作一个存储场所，以把重定位信息调整至一个 DWORD 倍数的尺寸。
- 3(IMAGE_REL_BASED_HIGHLOW)：重定位意味着把基差值的高 16 位和低 16 位与计算后的 RVA 所指向的 DWORD 相加。

在 WINNT.H 中还定义了其他的重定位，其中大多数是用在非 i386 结构上的。

图 8-13 描绘了一些用 PEDUMP 显示出的基址重定位。注意图中显示的 RVA 值已经用 IMAGE_BASE_RELOCATION 域中的 VirtualAddress 偏置过了。

COFF 符号表

如果你只对操作系统用到的 PE 文件的实际部分感兴趣，则你可跳过本节和下一节(“COFF 调试信息”)。在“PE 文件和 COFF OBJ 文件之间的差别”一节中你可再重新继续阅读。

在 Microsoft 编译器产生的任何 COFF 格式的 OBJ 文件中，你都将会找到一个符号表。不像 CodeView 信息，该符号表不是一个多余的、只在当带调试信息连接可执行文件时才用到的东西。事实上，这个符号表保存了关于被本模块访问的所有公共和外部符号的信息。被编译器产生的安置信息将访问该符号表中的特定

项。COFF 符号表的格式非常简单 - 事实上，它是如此的简单，以至于它使 Microsoft/Intel OMF 格式体制（用了它的 LNAME、PUBDEF 和 EXTDEF）相形见拙。

```
Virtual Address: 00001000  size: 0000012C
  00001032 HIGHLOW
  0000106D HIGHLOW
  000010AF HIGHLOW
  000010C5 HIGHLOW
  // Rest of chunk omitted...
Virtual Address: 00002000  size: 0000009C
  000020A6 HIGHLOW
  00002110 HIGHLOW
  00002136 HIGHLOW
  00002156 HIGHLOW
  // Rest of chunk omitted...
Virtual Address: 00003000  size: 00000114
  0000300A HIGHLOW
  0000301E HIGHLOW
  0000303B HIGHLOW
  0000306A HIGHLOW
  // Rest of relocations omitted...
```

图 8-13　来自于一个 EXE 文件的基址重定位

如果你没用调试信息允许选项来编译，你将只能获得具有最小数目符号的 OBJ 符号表。

如果你打开调试信息选项（用/Zi），编译器将加一些关于模块中每个函数的开始、长度和结束的附加信息。如果你然后用/DEBUGTYPE:COFF 或/DEBUGTYPE:BOTH 来连接，则连接器将输出一个 COFF 风格的符号表到结果的 EXE 文件中。

既然已经有更完整的 CodeView 信息，为什么你还要 COFF 信息呢？这是因为如果你打算使用 NT 系统调试器（NTSD）或者 NT Kernel 调试器（KD），则 COFF 是唯一能被使用的信息。另外，如果你的 PE 程序与 Windows NT 相冲突，DRWTSN32 可用该信息来产生有用的符号化的转储。

对于 EXE 和 OBJ 文件，你都可以通过查看 IMAGE_FILE_HEADER（如果你有些忘了该结构的话，请参见本章中“PE 头标”一节的内容）来找出 COFF 符号表的位置和尺寸。该符号表在结构上是非常的简单，它由一个 IMAGE_SYMBOL 结构数组组成，数组的元素个数由 IMAGE_FILE_HEADER 结构的 NumberOfSymbols 域给出。图 8-14 显示了由 PEDUMP 程序产生的符号样本输出。

```
Symbol Table - 433 entries  (* = auxiliary symbol)
Indx Name                 Value    Section    cAux  Type    Storage
---- -------------------- -------- ---------- ----- ------- --------
0000 .file                0000005B sect:DEBUG aux:1 type:00 st:FILE
     * EXEDUMP.c
0002 .debug$S             0001B457 sect:7     aux:1 type:00 st:STATIC
     * Section: 0000  Len: 017C8  Relocs: 002C  LineNums: 0000
0004 .data                0000B040 sect:4     aux:1 type:00 st:STATIC
     * Section: 0000  Len: 006CA  Relocs: 0020  LineNums: 0000
0006 _SzRelocTypes        0000B1E0 sect:4     aux:0 type:00 st:EXTERNAL
0007 _SzResourceTypes     0000B148 sect:4     aux:0 type:00 st:EXTERNAL
0008 _SzDebugFormats      0000B088 sect:4     aux:0 type:00 st:EXTERNAL
0009 _PCOFFDebugInfo      0000B040 sect:4     aux:0 type:00 st:EXTERNAL
000A .text                000026A0 sect:1     aux:1 type:00 st:STATIC
     * Section: 0000  Len: 00CE0  Relocs: 00A3  LineNums: 00D0
000C _DumpDebugDirectory  000026A0 sect:1     aux:1 type:20 st:EXTERNAL
     * tag: 000E  size: 01A4  Line #'s: 00009220  next fn: 0013
000E .bf                  00000000 sect:4     aux:1 type:00 st:FUNCTION
0010 .lf                  0000001A sect:4     aux:0 type:00 st:FUNCTION
0011 .ef                  000001A4 sect:4     aux:1 type:00 st:FUNCTION
0013 _GetResourceTypeName 00002844 sect:1     aux:1 type:20 st:EXTERNAL
     * tag: 0015  size: 004A  Line #'s: 000092BC  next fn: 001A
0015 .bf                  000001A4 sect:4     aux:1 type:00 st:FUNCTION
0017 .lf                  00000006 sect:4     aux:0 type:00 st:FUNCTION
0018 .ef                  000001EE sect:4     aux:1 type:00 st:FUNCTION
// Rest of symbols omitted...
```

图 8-14　一个典型的 COFF 符号表

每个 IMAGE_SYMBOL 结构具有下面的格式：

```
typedef struct _IMAGE_SYMBOL {
    union {
        BYTE    ShortName[8];
        struct {
            DWORD   Short;     // If 0, use LongName.
            DWORD   Long;      // Offset into string table.
        } Name;
        PBYTE   LongName[2];
    } N;
    DWORD   Value;
    SHORT   SectionNumber;
    WORD    Type;
    BYTE    StorageClass;
    BYTE    NumberOfAuxSymbols;
} IMAGE_SYMBOL;
typedef IMAGE_SYMBOL UNALIGNED *PIMAGE_SYMBOL;
```

现在让我们详细地检查每一个域：

union N（符号名联合）

该符号名字根据其长度可用两个方法来表示。如果符号名小于或等于 8 个字符，则该联合的 ShortName 成员含该 ASCIIZ 符号名。如果符号名正好是 8 个字符长时要特别小心，这时串没有 null 作结束。如果 Name. Short 域非零，则你必须使用该联合的 ShortName 成员。

表示一个符号名的第二种方式出现在当 Name. Short 域为 0 时。在这种情形中，Name. Long 域是进入字符串表的一个字节偏移量。该字符串表实际上就是内存中一个字符串跟着另个字符串的 ASCIIZ 字符串数组。该表在内存中直接跟在符号表后面。要找到该字符串的起始地址，用符号数目乘上一个 IMAGE_SYMBOL 的尺寸即可。该字符串表的长度，由该表中偏移量为 0 处的一个 DWORD 来给出其字节数。

DWORD　　Value

该域含的是与符号相联系的值。对于一般和数据符号（即，函数和全局变量），Value 域含的是该符号访问项的 RVA。对于一些符号，该值有不同的解释。表 8-2 简单地列出了特殊符号的 Value 域的一些含义。

表 8-2　COFF 符号表中特殊符号

符号名	用途
. file	下一个. file 符号的符号表索引。你可用该索引来快速遍历 EXE 中所有文件的列表。
. data	一个数据区的起始 RVA。该区由前面. file 符号所给出的源文件来定义。
. text	一个代码区的起始 RVA。该区由前面. file 符号所给出的源文件来定义。
. lf	一个函数的行号表中的项数。该函数由前面定义该函数的符号指定。

SHORT　　SectionNumber

SectionNumber 域含该符号所属节的节号。例如，对全局变量的符号，典型地具有. data 节的节号。除了在 PE 文件中的标准节外，还有三个另外的特殊节值被定义：

- 0(IMAGE_SYM_UNDEFINED)：该符号未定义。这个节号用于 OBJ 文件中，以表示符号是在本模块的外部，例如：外部函数和外部全局变量。
- -1(IMAGE_SYM_ABSOLUTE)：该符号是一个绝对值，并且不与任何给定的节相联系。例子有局部和寄存器变量。
- -2(IMAGE_SYM_DEBUG)：该符号只被调试器使用，并且对程序是不可见的。给出一个源文件名字的. file 符号就是这个符号节的例子。

WORD　　Type

该符号的类型。WINNT. H 文件定义了一个丰富的符号类型（int，struct，

enum，以及其他等等）集。（完整的清单请见 IMAGE_SYM_TYPE_xxx #defines。）不幸的是，Microsoft 工具似乎并不产生所有的符号类型，对于所有全局变量和函数都分别是 NULL 类型和函数返回为 NULL 的类型。

BYTE　　StorageClass

符号的存储类。与符号类型一起，WINNT.H 文件定义了一个丰富的存储类型（automatic，static，register，lable，以及其他等等）集。（完整的清单请见 IMAGE_SYM_CLASS_xxx #defines。）同样，Microsoft 工具也只产生少量的存储类型。所有全局变量和函数都是外部存储类。好像还没有任何方法可为局部变量、寄存器变量以及其他等等获取符号。

BYTE　　NumberOfAuxSymbols

符号表并不完完全全是一个 IMAGE_SYMBOL 结构数组。如果一个符号在它的 BYTE NumberOfAux Symbols 纪录中有一个非零值，则该符号后跟相同数量的 IMAGE_AUX_SYMBOL 结构。例如，一个.file 符号后跟的 IMAGE_AUX_SYMBOL 结构的数目，和它用此来容纳一个源文件的完整路径名的结构数目相同。

幸运的是，一个 IMAGE_AUX_SYMBOL 的尺寸和一个 IMAGE_SYMBOL 相等，因此你仍然可以把符号表作为一个 IMAGE_SYMBOL 数组对待。记住一个符号索引应被作为一个数组索引对待，即使一些元素或许是辅助记录。要计算下一个正规符号的索引，你需要加入该符号用到的辅助结构的数目。例如，设一个符号具有索引为 1，如果它使用了 3 个辅助符号，则下一个正规符号索引将是 4。

一个 IMAGE_AUX_SYMBOL 是域的一个混合联合。要确定使用哪个联合成员，你需要知道与辅助符号相联系的正规符号的类型。尽管我还没有完全想出在每种情况中应使用哪一个辅助联合域，但我还是能够想到两种：

- 存储类是 IMAGE_SYM_CLASS_FILE 的符号，使用 IMAGE_AUX_SYMBOL 结构中的 File 联合成员。
- 存储类是 IMAGE_SYM_CLASS_STATIC 符号，使用 IMAGE_AUX_SYMBOL 结构中的 Section 成员。

我所知道的怎样来解释辅助符号的全部知识，都包含在 COMMON.C 源文件中的 DumpAuxSymbols() 例程中，该源文件是来自于 PEDUMP 的。如果你自己想到了更多的方法，可自由地加到该例程中。

如果你检查在符号节中的信息，你将会看到符号不是被随机排序的，它们是以它们所来自的目标模块（或者源文件）来分组的。在 COFF 符号表中的第一个记录是一个.file 记录。一个.file 记录的值是对下一个.file 记录的符号表的一个索引，通过跟随这个.file 记录值链，你可以在 EXE 中的每个目标模块上进行迭代。直接跟在每个.file 记录的是与源文件相联系的其他记录。例如，在一个源文件中定义的所有公共符号（全局变量和函数）跟在表示该源文件的.file 记录后面。对于一个正规的源模块，符号记录的“层次”看起来像这样：

```
Source File record                          // Name of the source file.
    Data Section record (e.g., ".data")     // Data declared in file.
        GlobalVariable1 record              // Information about variable.
        GlobalVariable2 record
        // Rest of global variable records
    Code Section record (e.g., ".text")     // Code declared in file.
        Function1 record                    // Information about function.
            .BF record                      // Function begin info.
            .LF record                      // Function length info.
            .EF record                      // Function end info.
        Function2 record
            .BF record
            .LF record
            .EF record
        // Rest of function records
```

COFF 调试信息

对于 PC 程序员，术语调试信息(debug information)包含了符号和行号两方面信息。在 COFF 格式中，符号和行号记录是存在文件不同的分离区域中的。(在 Borland 或 CodeView 符号表格式中，行号和符号信息来自于文件的同一部分。)我先讨论 COFF 符号表部分，因为它在 OBJ 和 EXE 文件中都可出现。还有，在学习 PE 格式过程的开始时，你已碰见了 IMAGE_FILE_HEADER 中的 PointerToSymbolTable 域。因为这些原因，我把符号表作为一个单独的项加以描述。

在一个 EXE 文件中的完整 COFF 符号表由三个部分组成：头标、行号信息和符号表。它们在内存中不必一定是连续存储的，但 Microsoft 连接器在文件中是把它们连续放在一起的。一个完整的 COFF 符号表看起来像这样：

IMAGE_COFF_SYMBOLS_HEADER 结构
行号表
符号表(先前已讨论过)

该 IMAGE_COFF_SYMBOLS_HEADER 结构的目的是使调试者能够快速地定位在他们需要知道的重要信息上，该结构含有指向行号和符号表的指针，也含有一些可在文件中其他地方找到的信息。

要找到该 IMAGE_COFF_SYMBOLS_HEADER 结构，则要在文件的.rdata 节中的 IMAGE_DEBUG_DIRECTORY 结构数组中查看。具有一个 Type 域值为 1 的 IMAGE_DEBUG_DIRECTORY 含一个指向 COFF 符号表的指针。快速重述这个过程为：数据目录(在 PE 头标的尾端)含一个相对于一个 IMAGE_DEBUG_DIRECTORY 数组的 RVA，对文件中出现的每一种调试信息都有一个 IMAGE_DEBUG_DIRCTORY 与之相对应。如果这些 IMAGE_DEBUG_DIRECTORY 中

的一个应用于 COFF 风格的调试信息，则它包含一个相对于一个 IMAGE_COFF_SYMBOLS_HEADER 结构的 RVA，而 IMAGE_COFF_SYMBOLS_HEADER 结构包含了指向 COFF 符号表和行号信息的指针。IMAGE_COFF_SYMBOLS_HEADER 结构具有下面的格式：

```
typedef struct _IMAGE_COFF_SYMBOLS_HEADER {
    DWORD   NumberOfSymbols;
    DWORD   LvaToFirstSymbol;
    DWORD   NumberOfLinenumbers;
    DWORD   LvaToFirstLinenumber;
    DWORD   RvaToFirstByteOfCode;
    DWORD   RvaToLastByteOfCode;
    DWORD   RvaToFirstByteOfData;
    DWORD   RvaToLastByteOfData;
} IMAGE_COFF_SYMBOLS_HEADER, *PIMAGE_COFF_SYMBOLS_HEADER;
```

让我们详细看看 IMAGE_COFF_SYMBOLS_HEADER 的各个域：

DWORD　　NumberOfsymbols

COFF 符号表中符号的数目。该域含有与 IMAGE_FILE_HEADER. NumberOfSymbols 域相同的值，这点在本章中前面的“PE 头标”一节中已讨论过。

DWORD　　LvaToFirstSymbol

相对于该结构开始处的对 COFF 符号表的字节偏移量。将该值和 IMAGE_COFF_SYMBOLS_HEADER 的 RVA 相加会得到与 IMAGE_FILE_HEADER. PointerToSymbolTable 域相同的结果。

DWORD　　NumberOfLinenumbers

行号表中的项数(见图 8-15)。

DWORD　　LvaToFirstLinenumber

相对于该结构开始处的对 COFF 行号表的字节偏移量。

DWORD　　RvaToFirstByteOfCode

映象中可执行代码第一个字节的 RVA。该域通常和. text 节的 RVA 相同。通过扫描可执行文件的节表可以找到该值。

DWORD　　RvaToLastByteOfCode

映象中可执行代码最后一个字节的 RVA。假定你只有一个代码节(. text)的话，这个域将等于该节的 RVA 加上它的生数据尺寸。这个值也可通过扫描节表来找到。

```
Line Numbers
SymIndex: C (_DumpDebugDirectory)
 Addr: 016A9  Line: 0008
 Addr: 016B5  Line: 0009
 Addr: 016BF  Line: 000A
 Addr: 016C4  Line: 000E
 // Rest of line number for function omitted...
SymIndex: 13 (_GetResourceTypeName)
 Addr: 0184A  Line: 0001
 Addr: 01854  Line: 0002
 Addr: 0186F  Line: 0003
 Addr: 01874  Line: 0004
 // Rest of line number for function omitted...
SymIndex: 1A (_GetResourceNameFromId)
 Addr: 01897  Line: 0004
 Addr: 018A1  Line: 0006
 Addr: 018B6  Line: 0007
 Addr: 018BB  Line: 000A
 // Rest of line numbers omitted...
```

图 8-15　一个 EXE 文件中典型的 COFF 行号信息

DWORD　　RvaToFirstByteOfData

映象中数据的第一个字节的 RVA。这个域通常和. bss 节的 RVA 相同。

DWORD　　RvaToLastByteOfData

映象中程序可存取数据的最后一个字节的 RVA。被这两个 FirstByteOfData 和 LastByteOfData 域包含的区域可以跨越几个节(例如,. bss,. rdata,和. data)。

COFF 行号表

被 IMAGE_COFF_SYMBOLS_HEADER 结构指向的 COFF 行号表非常简单:它只不过是一个 IMAGE_LINENUMBER 结构数组。每个结构把一行源代码与在可执行映象中它的 RVA 相对应。图 8-15 显示了通过 PEDUMP 显示出的一个样本行号表。一个 IMAGE_LINENUMBER 的格式有两个域:一个联合和一个 WORD:

```
union{
DWORD SymbolTableIndex
DWORD VirtualAddress
} Type
```

如果后面的 Linenumber 域非零，则该域应该被作为一行代码对应的一个 RVA 对待。如果 Linenumber 域为 0，则该域含一个进入符号表的索引，被该索引对应的符号记录标识一个函数，而那个函数相应的所有行号记录跟在该特殊记录之后。通过查看 PEDUMP 的输出，你可看到这个行号表由一个符号表索引记录组成，它后跟一些正规的行号记录，而这些行号记录又后跟另一些符号表索引记录，依此类推。

WORD　　Linenumber

为一个行号，是相对于函数开始处的一个相对值。该域不是文件中的一个真正行号，要把该域转换为文件中的一个可用行号，应查找符号表中相应函数的起始行号。这个函数是最近的、该域为 0 的行号记录所对应的函数。如果不清楚的话，请见图 8-15 中的 PEDUMP 输出。

如果对一个给定代码节，你只想存取其行号，则你可只查看来自该节表的行号项相关范围。一个节的 IMAGE_SECTION_HEADER 包含了一个文件偏移量和该表中行号的一个计数。COFF 格式的 OBJ 还在我刚才已描述过的格式中包含了行号信息。因为在一个 OBJ 文件中不存在 IMAGE_COFF_SYMBOLS_HEADER 结构，你需要通过 IMAGE_SECTION_HEADER 结构来找到行号记录。

PE 文件和 COFF OBJ 文件之间的差别

在前面讨论的许多点上，我已经注意到 COFF OBJ 文件和 PE 文件中的许多结构和表是相同的。COFF OBJ 和 PE 文件在位于或接近于其开始处的地方，都有一个 IMAGE_FILE_HEADER。该头标后跟一个节表，节表包含了关于文件中所有节的信息。这两种格式还共享相同的行号和符号表格式，尽管 PE 文件也可有附加的非 COFF 格式的符号表。这两种格式之间的共同之处可以在 PEDUMP 源代码中看到，在此程序中最大的文件就是 COMMON.C，该源文件包含 了所有的既可被 PE 转储程序又可被 OBJ 转储程序部分使用的例程。

这两种文件格式之间相似处并不是偶然的，这样设计的目的是为了使连接器的任务尽可能的简单。理论上讲，由一个单 OBJ 创建一个 EXE 文件应该只需插入一些表并修改映象内的一对文件偏移量就行了。照这种思想，你可认为一个 COFF 文件是一个萌芽期的 PE 文件。只有一些东西是省略或不同的，因此我在这里把不同点列出来：

- COFF OBJ 文件直接用一个 IMAGE_FILE_HEADER 作为开始。在头标之前没有 DOS 存根，在 IMAGE_FILE_HEADER 之前也没有 PE 标记。
- OBJ 文件没有 IMAGE_OPTIONAL_HEADER。在一个 PE 文件中，该结构直接跟在 IMAGE_FILE_HEADER 后面。有趣的是，COFF LIB 内部中一些 OBJ 有 IMAGE_OPTIONAL_HEADER。
- OBJ 文件没有基址重定位。与此对照的是，它们有规则的基于符号的安置。我没有谈及 COFF OBJ 文件的重定位格式，是因为它们颇为含混。节表项中的 Point-

ToRelocations 和 NumberOfRelocations 域指向每节的重定位，该重定位是一个 IMAGE_RELOCATION 结构数组，该结构在 WINNT.H 中定义。如果你使用合适的开关，PEDUMP 程序能显示出 OBJ 文件的重定位。

- OBJ 文件中的 CodeView 信息被存于两节(.debug$S 和.debug$T)中。当连接器处理 OBJ 文件时，它并不把这些节放入 PE 文件中，代之的是它收集所有这些节并且建立一个单一的符号表。该符号表存在文件的尾部，它不是一个正式的节(即，在 PE 的节表中并没有为它而设的项)。

COFF LIB 文件

一旦理解了 COFF OBJ 文件，使用 COFF LIB 文件就不是太困难的了。COFF LIB 文件基本上就是一个 COFF OBJ 文件集，再加之一些初始节，这些初始节使你能够快速地查找在该库中所需要的 OBJ 文件的位置。有关 COFF LIB 格式的资料作为档案参考了 LIB 文件，为了保持连续性，我在这里先讨论一下 LIB 文件。

所有 LIB 文件开始都是一个相同的 8 字节标记。该标记在 WINNT.H 中定义：

```
#define IMAGE_ARCHIVE_START                    "!<arch>\n"
```

文件余下的部分是一系列可变长的记录，而每条记录以一个 IMAGE_ARCHIVE_MEMBER_HEADER 结构作为开始：

```
typedef struct _IMAGE_ARCHIVE_MEMBER_HEADER {
    BYTE     Name[16];
    BYTE     Date[12];
    BYTE     UserID[6];
    BYTE     GroupID[6];
    BYTE     Mode[8];
    BYTE     Size[10];
    BYTE     EndHeader[2];
} IMAGE_ARCHIVE_MEMBER_HEADER, *PIMAGE_ARCHIVE_MEMBER_HEADER;
```

每个 IMAGE_ARCHIVE_MEMBER_HEADER 相对应的是该库中的一个 OBJ 文件或者特殊的记录集中的一条记录。这些特殊记录位于库的开始处，是为了使连接器能够快速查找文件中后面的 OBJ 文件。档案文件成员所用的生数据直接跟在 IMAGE_ARCHIVE_HEADER 之后。对于大多数文件成员记录，其生数据正好是和包含的 OBJ 文件相同的文件。事实上，当转储出 LIB 文件时，PEDUMP 程序调用同一个 OBJ 转储例程，PEDUMP 用该例程来处理 OBJ 文件。图 8-16 显示了 LIB 文件的格式。

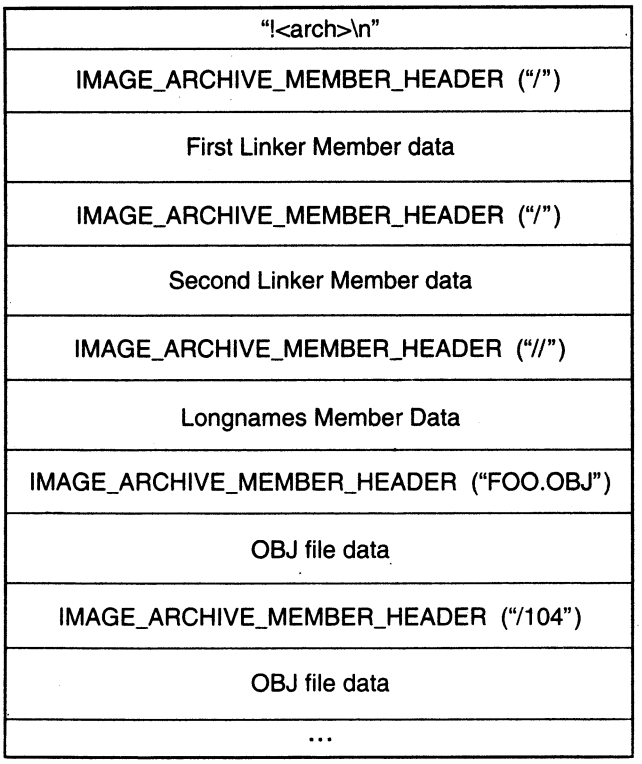


图 8-16　COFF 格式的 LIB 文件的结构

让我们看一看 IMAGE_ARCHIVE_HEADER 的各个域：

BYTE　　Name[16]

档案文件成员名字。如果一个"/"号出现在一个 ASCII 串后面(例如：FOO. OBJ/)，则在"/"号前面的串就是该成员名。如果该名字以一个"/"开始并且后跟一个十进制数(例如：/104)，则该数值是该 LIB 文件的 Longname 成员中档案文件成员的偏移量。在前面这个例子中该成员名将在 Longname 区域的 104 字节处开始。

也有特殊的名字来标识该特殊的档案文件成员：

```
#define IMAGE_ARCHIVE_LINKER_MEMBER          "/               "
#define IMAGE_ARCHIVE_LONGNAMES_MEMBER       "//              "
```

对于一个引入库中的 OBJ 文件，该域是包含了引入函数的 DLL 的名字。

BYTE　　Date[12]

该成员的创建日期和时间。该数值以 ASCII 十进制形式存储。

BYTE　　UserID[6]

用户 ID 的 ASCII 十进制表示。该域似乎总是一个 NULL 串。

BYTE　　GroupID[6]

组 ID 的 ASCII 十进制表示。该域似乎总是一个 NULL 串。

BYTE　　Mode[8]

该文件模式的 ASCⅡ十进制表示。该域似乎总是为 0。

BYTE　　Size[10]

跟在后面的成员数据的尺寸，用 ASCII 十进制形式表示。其数据的格式依其类型(由先前描述过的 Name 域指定)而定。

BYTE　　EndHeader[2]

ASCII 串“\n”。

Linker 成员

每个 LIB 文件都有两个 Linker 成员节，其作用是作为一个文件其余部分所用的内容表。

两个成员的名字都后带“/”，并且根据其出现在文件中的顺序来区别。第一个 Linker 成员是第一个具有名字后带“/”的档案文件成员，而第二个 Linker 成员是第二个这样的档案文件成员。

两个 Linker 成员基本上都是 LIB 文件中公共符号的列表，并连带相对于包含这些公共符号的 OBJ 成员的文件偏移量。这两个 Linker 成员具有不同的格式。为什么同一个信息会有两个拷贝呢？第一个 Linker 成员是以 OBJ 在 LIB 文件中出现的顺序为序来存储信息的，这导致非优化的查询。第二个 Linker 成员以字母顺序来存储它的信息，因此它对连接器更为有用。根据 Microsoft 文献说明，连接器忽略第一个 Linker 成员，并且总是使用第二个 Linker 成员。

第一个 Linker 成员具有如下格式：

DWORD　　NumberOfSymbols

该库中公共符号的数目。该数目是大结尾(big-endian)格式(在不同于 i386 的机器上表现 COFF 的继承性)。PEDUMP 的 LIBDUMP.C 文件中的 ConvertBigEndian 函数可操纵开关将大结尾格式转换为 i386 所用的小结尾(little-endian)格式。

DWORD　　Offset[NumberOfSymbols]

这是相对于其他档案文件成员的文件偏移量组成的一个数组。该偏移量是用

的大结尾格式。这些成员的每一个都是一个 OBJ 类型成员。该数组的每个元素相对应于紧随其后的 ASC Ⅱ 串列表中等价的、已排序过的符号名。

BYTE StringTable[?]

这是内存中不间断的一系列 C 风格下的串。

基本上讲，Offset 数组中的每个元素都对应一个公共符号，该符号的名字在 StringTable 区域中出现。例如，Offset 数组第三个元素和 StringTable 区域中的第三个串相联系。PEDUMP 的输出使这更清晰：

```
First Linker Member:
  Symbols:          00000006
  MbrOffs   Name
  --------  ----
  00000180  _DumpCAP@0
  00000180  _StartCAP@0
  00000180  _StopCAP@0
  ...
```

第二个 Linker 成员的格式更为复杂，因为为了符号快速查找，则必须对数组增加内容。第二个 Linker 成员的格式如下：

DWORD NumberOfMembers

该 DWORD 含的是后面出现在该文件中的 OBJ 档案文件成员的数目。

DWORD Offsets[NumberOfSymbols]

这是一个相对于其他档案文件成员的文件偏移量组成的数组。不像第一个 Linker 成员，这些偏移量采用机器自然格式(即 i386 机器用的小结尾格式)。

DWORD NumberOfSymbols

在 StringTable 数组中公共符号的数目(因此也是库中公共符号的数目)。该域还包含了紧随其后的 Indices 数组元素个数。

WORD Indices[NumberOfSymbols]

该数组保存进入 Offsets 数组的索引，该索引从 1 开始。该数组和 StringTable 数组中的串并行运行。

BYTE StringTable[NumberOfSymbols]

这是内存中不间断的一系列 C 风格下的串。

要找到与用第二个 Linker 成员来指定的符号相对应的 OBJ 文件，连接器首先查询 StringTable 数组，并且计算数组中该串的相对索引。然后，连接器用该索引查找出 Indices 数组的一个 WORD。最后连接器把这个 Indices 数组中的 WORD 值减去 1，并且把此结果作为进入 Offset 数组的一个索引。被找出的 Offsets 数组的

DWORD 就是包含该公共符号的 OBJ 文件的文件偏移量。PEDUMP 的 LIBDUMP.C 中的 DumpSecondLinkerMember 函数显示了这个处理过程。

Longnames 成员

在 Longnames 档案文件成员节中的数据，简单地是一个 C 风格串集合，这些串是一个接一个的。如果一个串长度大于 16 字节的话，它就被放入 Longnames 节中，因为 IMAGE_ARCHIVE_MEMBER_HEADER 结构中的 Name 域保留长度是 16 字节。在这种情形中，该 Name 域含了一个“/”符号，并且后跟 Longnames 节中该串的偏移量的 ASCII 十进制表示。

小结

随着 Win32 的问世，Microsoft 对 OBJ 和可执行文件格式做了一个延伸性的改变。这个改变允许 Microsoft 能够在使先前所做的工作适应另外操作系统的过程中节省时间。这些被修改的文件格式的主要目的是为了增强在不同平台上的兼容性。COFF OBJ 格式是先于 Win32 之前出现的，而 PE 格式是对 COFF 格式的一个扩展，并且是为 Win32 平台而设计的。

OBJ 和可执行文件两者的有用部分，都是用一个 IMAGE_FILE_HEADER 结构作为开始的。跟随该结构（和可能的一个附加可选的结构）的是一个节表。节表包含了文件中所有节的位置和属性。一个节是一些代码和数据的集合，这些代码和数据在逻辑上应该是在一起的。。为了使快速查找信息比较容易，PE 文件包含了一个数据目录，该目录指向文件中有用的位置（例如，文件的引出表位置）。除了头标、节表和生的节数据外，COFF OBJ 文件和 PE 文件还可包含有关符号名称和行号的信息，这些信息是位于所有头标和节数据之后，存储在文件尾端的。

第 九 章

读者自身探密

* *

与本书的其它部分不同，本章并不是着眼于 Windows 95 的结构和工作原理，而是要描述独立编码所需的许多基本要素中的一部分。有句谚语叫做“授之以鱼不如授之以渔”。本书其它各章是“给你一条鱼”，而这一章却是要“教给你捕鱼的方法”。本章就是要教你自己如何去探明 Windows 的奥密。

当然，你在这儿将要学到的方法，也可以用于其它情况，例如应用于设备驱动程序或最终用户应用程序。

在理想状态下，你可以在文档资料中找到所有你所需要的操作系统信息，它允许你将操作系统作为一个“黑匣子”来看待，你不必要理解这种操作系统的内部机制和数据结构，因为对文档所描述的接口的理解和应用，对于你的编程、建库和写设备驱动程序已经足够了。

在缺乏完整资料的条件下，操作系统的源代码可以代替资料信息。尽管你不得不参考别人所编写的代码，但只要充分挖掘操作系统源代码，几乎对于有关操作系统的所有问题都能得到解决。事实上，Eric S. Raymond 的《The New Hacker's Dictionary》中已含有 UTSL(即 Use the Source Luke)的条目。

在 UNIX 中，存取操作系统源代码相当普通。但不幸的是，对我们这些 Windows 的编程者，源代码是无法得到的。你可以在 DDK 中找到一些实际的 Windows 95 代码。但就大多数情况而言，多数编程问题所涉及的既不是 SDK 提供的也非 DDK 所提供的代码。虽然，Microsoft 在这些领域的资料在近几年有了显著的改进，但在 Windows SDK 文档资料中仍存在许多漏洞，这些漏洞只能用真正的源代码来弥补。

当你工作涉及操作系统时，你遇到的问题不仅仅是原始资料的不足和源代码的不易获取。你的应用程序有可能要和另一个应用程序相互作用，而你对这个程序的准确行为却一无所知。典型的情况就是程序员被迫花大量时间去记录 Microsoft Excel 所发出的 DDE 信息，以及以何顺序发送。在 Windows 3.1 下的另一个例子是一个程序运行时，发生其它程序不能运行的情况，它的问题在于：一些程序在运行时占满了 1M 地址以下的所有内存。

在这章里，我们讨论 Windows 程序员用到的如下“探密”方法：

- 文件转储实用程序
- API 和消息侦探(spy)程序(如第十章中的 APISPY32)

·反汇编

在本章各节中，我将描述常用而有效的工具，并给出如何使用这些工具来找出有用信息的例子。最后一节是关于反汇编技术的，其内容特别详细，这是因为反汇编被认为是一种“黑色魔术”，在一般书中很少涉及。在一个调试器中阅读和单步执行汇编代码的许多技巧只能了解普通的编译器代码生成格式。有一些涉及实际经验的方法，但我将在稍后一些时间才对此作详尽的讨论。

本章中描述的大部分探密概念，对于初级或中级程序员是很有用的。但本章的最后一部分，对于谨慎的探密者来说，集中了警告和技巧。这毕竟是我通过艰苦的摸索得到的经验和教训，如果你感兴趣的话，你将从我得知不易的知识中受益非浅。

尽管 Windows 95 是一个 Win32 操作系统，但其核心部分仍用 16 位代码和 16 位 NE 文件格式，因此，本章叙述 Win16 的探密方法就十分必要了。尽管一些探密技术从 Win16 过渡到 Win32，但它们之间仍有显著区别，因此，我也将描述 Win32 下的工具和技术。

探密概览

了解一段代码细节的最简易和最可行的方法是用文件显示程序，如 Borland 的 TDUMP，Microsoft 的 EXEHDR 和 DUMPBIN，或第八章中的 PEDUMP。这些程序可以告诉你诸如一个程序所用的 DLL 和 API 的函数，但不能提供给你有关内部算法和数据结构的信息。可以这样认为：对于探密来讲，文件转储好象执行侦察任务时你是透过你自己的窗户向外看房屋的全景一样，这相对容易，但你可能得不到你所需的所有信息。

为了对一个程序的内部作更为复杂的探密，你可以用一个“侦探”程序。像 Microsoft Windows SDK 的 SPY 和 Borland C++的 WinSight 一类的程序，可显示出一个程序发送和接收的窗口消息。最近，如 Nu-Mega 的 BoundsChecker 产品和 Periscope 的 WinScope 等程序增加了可以观察你的程序对操作系统 API 函数进行调用的功能。第十章提供了你可使用的一个可扩展的 Win32 API 侦探程序。用这些信息加上少量的工作，你几乎可以看到任何一个“精巧的”代码段是如何执行的。我将在本章稍后给出一个用侦探程序进行探密的例子。再回到我们的“房屋中的侦查”这个类比上来，侦探程序可以被想象成为对进出房子的邮件或电话的拦截。

最后，当你需要知道一个程序或 DLL 的内部算法和数据结构时，你可以用反汇编来达到目的。尽管用一个好的调试器你可以作有限的反汇编工作，但你或许更想用一个基于文件的反汇编器，如 Eclectic Software 的 Win2Asm 或 V-Communications 的 Sourcer。增加你自己的注释和格式到列表文件的功能，使得基于文件的反汇编程序在进行真正的反汇编时要优于你所忠爱的调试器。进一步推出我们的“在房屋中侦察”的类比，反汇编就如同破门而入并对屋内各部分进行检查。我将在“用反汇编来探密”一节中描述反汇编工具和技术。

用文件转储工具探密

探密一个程序的第一步是转储文件内容，这是了解你处理的文件类型和文件可能被用作什么的快速而容易的方法。表 9-1 列出了一些著明的、明确为分解一个文件内容而编写的工具的功能。

表 9-1　常用的文件分解工具功能

文件类型	DUMPIN	DUMPEXE	EXEHDR	TDUMP
MZ 文件(DOS)		×	×	×
NE 文件(WIN16)		×	×	×
PE 文件(WIN32)	×	×		×
调试信息	×	×		×
反汇编	×			
OBJ	×			×
资源	×	×		×

DUMPBIN 来自于 Microsoft Win32 SDK 和 Visual C++

DUMPEXE 来自于 Symantec C++

EXEHDR 来自于 Microsoft Visual C++

TDUMP 来自于 Borland C++

如果你用 Borland C++进行开发，在 BIN 目录中试用 TDUMP.EXE 文件。如果你用 16 位的 Microsoft C/C++进行开发，EXEHDR 可能是你应选择的工具。如果你用 Visual C++或 Win32 SDK ，在 BIN 目录下的 DUMPBIN 更适合可移植的执行格式(PE)和由 Microsoft 32 位编译器产生的 COFF 格式的目标文件。正如 9-1 表所示，没有一个程序可以完成所有的事，所以至少持有两个程序是个好主意！TDUMP 和 DUMPBIN 两个结合起来可以应付大多数情况。

你从文件转储程序中得到的最有用的信息通常是程序或 DLL 引入的 DLL 和函数的名字。当你在某点上被难住时，通常只要知道一个程序用了某个函数就足以让你继续。例如，在 Window 3.0 中，没有任何资料说明怎样改变桌面壁纸(desktop wallpaper)，但控制面板(Control Panel)应用程序可以改变壁纸，这就说明该功能就在 Windows 中的某个地方。通过在 Window3.0 的控制面板程序上运行 TDUMP 或 EXEHDR，你可以看到程序用到了资料未说明的 SetDeskWallPaper 函数。(在 Window3.1 中，有资料说明的 SystemParametersInfo API 接管了这个功能)。

寻找一个 16 位 NE(New Executable)格式程序或 DLL 所用的函数，是一个分为两步的过程。NE 文件不包含来自其它 DLL 的引入函数的简要列表。因此，这个过程的第一步就是找到可执行段的安置数据。如果你使用 EXEHDR，你就还得用/VERBOSE 开关去获得安置信息，下面这个输出结果显示了在 Window3.1 CALC.EXE 中出现的典型安置序列的 TDUMP 输出：

```
...
PTR     0AD9h   GDI.91
PTR     0121h   GDI.93
PTR     00EAh   USER.89
PTR     0223h   USER.90
PTR     04ADh   USER.91
PTR     1DCAh   USER.92
...
```

这个输出中的重要信息是每行最后的模块名和引入序号，在这种情况下，程序引入 6 个函数，两个来自 GDI，四个来自 USER。像 GDI. 91 一类的函数名本身并不能实际使用，所以该过程的第二步是把模块名(GDI)和序数(91)转化为真正的函数名。

当一个 EXE 文件或 DLL 引出函数时，函数名和其它相关的引出序数值被存储在可执行文件中，同时也存储在一个引入库中。因为对于 Microsoft 用户没有一个简便的方法来转储 16 位引入库的内容，因此我将演示如何直接从 DLL 获得函数名。先让我们回到第二步：我们必须指出 GDI. 91 是什么，也就意味着要转储出 GDI. EXE 以便看看引出序号是 91 的引出函数是什么。下面的输出显示了一个非驻留名字表(Non-Resident Name Table)的一部分，它是通过在 GDI. EXE 上运行 TDUMP 得到的。如果你用 EXEHDR，你将在标为“Exports”的节中找到类似的信息：

```
Non-Resident Name Table     offset: 0C41h
    Module Description: 'Microsoft Windows Graphics Device Interface'
    Name: GETWINDOWEXTEX                Entry:   474
    ....
    Name: GETTEXTEXTENT                 Entry:    91
```

看一下 GDI. EXE 的 TDUMP 输出，请注意对应于 GDI. 91 的函数 GetTextExtent。把上面两个放在一起来看，很显然，这是 CACL. EXE 调用 GetTextExtent (GDI. 91)。其它引入函数名可以通过重复以上描述的这个二步过程来获得。不过要记住，这种方法并不能发现在运行时通过调用 GetProcAddress 来与一个程序相连的函数。在这种特定情况下，你得求助于反汇编以发现这些调用。

寻找 32 位 PE 文件引入的函数要简单得多。在一个 Win32 程序上运行 DUMPBIN 或 PEDUMP，可以显示出一个 PE 文件包含了一个它引入的所有函数的列表。(在转储中该表看起来比较简单，然而在文件内部，引入节颇为复杂。引出节和 PE 格式的详细描述请见第八章。)引入函数表甚至也是被模块存储的。以下是通过在 Windows NT 3. 5 USER32. DLL 上运行 DUMPBIN 得到的引入的片断：

```
ntdll.dll
Hint/Name Table: 0002F31C
TimeDateStamp:   2E67E68D
ForwarderChain:  FFFFFFFF
First thunk RVA: 0002F050
```

```
Ordn  Name
  78  NtCreateSection
 226  NtUnmapViewOfSection
 503  RtlUnwind
 901  strrchr
 890  sscanf
 ...  rest of functions omitted

KERNEL32.dll
Hint/Name Table: 0002F3CC
TimeDateStamp:   2E67E68D
ForwarderChain:  FFFFFFFF
First thunk RVA: 0002F100
Ordn  Name
 119  FindClose
 150  GetAtomNameW
 378  LocalReAlloc
 368  LoadLibraryW
 236  GetModuleFileNameW
 ...  rest of functions omitted
```

在这个输出中，包含了函数名的每一行的第一个数字是线索序号。Win32 操作系统通过名字来引出函数，但线索号可以加速这个过程，它给装载器提供线索，说明应该在引出了函数的 DLL 中的那个位置开始进行寻找 API 名字的二分查找。

通过看 USER32. DLL 的引入表，我们可以看到它调用 KERNEL32 函数，如 GetAtomNameW 和 LocalReAlloc。有趣的是，当 NT USER32. DLL 有调用 ASCII 或调用单一码 API 的选择时，它调用的是单一码版本(GetAtomNameW，LoadLibraryW 等等)，这与 Microsoft 的声明是一致的，即 Windows NT 在内部用单一码。

这个输出也表明了 NT 的 USER32. DLL 使用了 NTDLL. DLL 的许多函数，而 NTDLL. DLL 是一个其函数完全没有资料说明的 DLL！有趣的是，在 NT 中，KERNEL32. DLL 在很大程度上依赖 NTDLL. DLL 的函数。与之对照的是，在 Windows 95 中，NTDLL. DLL 仍然存在，但 KERNEL32. DLL 看起来没用它任何一点东西了。事实上，Windows 95 NTDLL. DLL 与 NT 版本反了一个个，它很大程度上依赖 KERNEL32 的函数。

尽管知道 EXE 或 DLL 用的 API 是什么很有用，但其相反一面也同等地重要。文件转储程序可以显示出一个 DLL 为了别的程序和 DLL 的使用而引出的 API。引出函数受限于 DLL 的目的和功能。对一个没有资料说明的函数，有时一个名字就可提供足够的信息去猜测函数的参数，而其它情况下，你就需要结合引出 API 函数名字，用反汇编技术来探明应如何调用一个无说明的 DLL 函数。

下面的输出显示了从一个 16 位 NE 文件 SPELL. DLL 引出函数的 TDUMP 输出结果，这个 DLL 是 Windows 2. 0 下的 Microsoft Word 所带的，但它的 API 没有资料说明。

一些函数名字本身就给出了该 DLL 是干什么的明显线索，同时也给出了一个应该如何调用这个函数的主意。例如，函数 SpellVer()可能不带任何参数，而且可能在 AX 或 DX:AX 寄存器中返回一个版本值。编写一个小程序来验证这个推断

是非常容易的，只要在 SPELL.DLL 上作一个 LoadLibrary，调用 GetProcAddress 以获得 SpellVer()函数的地址，然后调用该函数。

```
Non-Resident Name Table     offset: 02A8h
    Module Description: 'Word for Windows v. 2.0 Spell Checker DLL'
    Name: SPELLOPENUDR                  Entry:     8
    Name: SPELLGETSIZEUDR               Entry:    13
    Name: SPELLADDUDR                   Entry:     9
    Name: SPELLOPTIONS                  Entry:     3
    Name: SPELLDELUDR                   Entry:    11
    Name: SPELLTERMINATE                Entry:     5
    Name: SPELLADDCHANGEUDR             Entry:    10
    Name: SPELLINIT                     Entry:     2
    Name: SPELLVER                      Entry:     1
    Name: SPELLCLOSEMDR                 Entry:    15
    Name: SPELLCHECK                    Entry:     4
    Name: SPELLVERIFYMDR                Entry:     6
    Name: SPELLOPENMDR                  Entry:     7
    Name: SPELLCLOSEUDR                 Entry:    16
    Name: SPELLCLEARUDR                 Entry:    12
    Name: SPELLGETLISTUDR               Entry:    14
```

在写一个小程序来进行验证时，我发现在这种方式下调用时，这个函数总返回零。求助于反汇编，我发现 SpellVer 事实上对 WORD(LPWORD)采用了三个远指针。由此得到的教训就是：尽管文件转储是探密最方便的方法，但它不能在所有情况下给出完全合适的信息。

再回头看看从 SPELL.DLL 引出的其它函数，注意其中有 SpellInit()、SpellCheck()和 SpellTerminate 函数。因此这个 DLL 可能期望是要被初始化、被调用来检查一些文件，然后关闭。我们所不知道的是这些 API 需要何种参数。这是再一次要用反汇编的地方了。

如果你想了解一个产品不同版本之间所产生的新而令人振奋的改变，比较两个相对应的 DLL 的引出是个好方法。表 9-2 列出了在 Windows3.1 中的 KRNL-386.EXE 和 Windows 95 中的 KRNL386.EXE 所引出函数的差别。为了获取这个信息，我用 EXEHDR 把 Windows3.1 中的 KRNL386.EXE 和 Windows 95 中的 KRNL386.EXE 的引出分别转储到分离的文件中。接着，我在各个文件中将函数按字母排序。最后，我运行 DIFF 程序来显示区别。

如表 9-2 所示，一些过时的函数被删除了，而增加了一批令人感兴趣的新函数。一些函数有资料说明(例如 GetPrivateProfileStruct)，但大多数没有(例如 Piglet_361 和 GetVDMPoint32W)。值得注意的是，该表遗漏了大量新引出的 KRNL386 函数，这些函数只用序号引出，而且它们在 KRNL386 的驻留或非驻留名字表中也没有名字。

在表 9-2 中，注意一些函数名被相对应的诸如 K209 和 K210 之类的所掩盖。如果我们知道这些函数的名字，那肯定就能比较容易地猜出它们的目的。正如猜想的那样，一些新 Kxxx 函数是用于从 16 位 NE 程序中在 Win32 堆上分配和释放内存，用到了 Kxxx 函数的一个很好例子就是 USER.EXE，它把 WND 结构存储到

USER DGROUP 的上面部分(超出 64K 限制),第五章包含了有关这些函数的更多信息。

表 9-2　Windows 95 KRNL386 引出函数与 Windows 3.1 KRNL386 引出函数对照:哪些是新增的,哪些是删除的

	KRNL386 引出函数
Windows 95 新增的	CALLPROC32W,CREATEDIRECTORY,DELETEFILE, FINDCLOSE,FINDFIRSTFILE, FINDNEXTFILE,FREELIBRARY32W, GETCURRENTDIRECTORY,GETDISKFREESPACE, GETFILEATTRIBUTES,GETLASTERROR, GETMODULENAME,GETPRIVATEPROFILESECTION, GETPRIVATEPROFILESECTIONNAME, GETPRIVATEPROFILESTRUCT,GETPROCADDRESS32W, GETPRODUCTNAME,GETPROFILESECTION, GETPROFILESECTIONNAMES,GETVDMPOINTER32W, GETVERSIONEX,GLOBALSMARTPAGELOCK, GLOBALSMARTPAGEUNLOCK,INVALIDATENLSCACHE, ISBADFLATREADWRITEPTR,K208,K209, K210,K211,K213,K214,K215,K228,K229,K237, LOADLIBRARYEX32W,LSTRCATN,OPENFILEEX, PIGLET_361,REGCLOSEKEY,REGCREATEKEY, REGDELETEKEY,REGDELETEVALUE,REGENUMKEY, REGENUMVALUE,REGFLUSHKEY, REGISTERSERVICEPROCESS,REGLOADKEY, REGOPENKEY,REGQUERYVALUE, REGQUERYVALUEEX,REGSAVEKEY,REGSETVALUE, REGSETVALUEEX, REGUNLOADKEY, REMOVEDIRECTORY, SETCURRENTDIRECTORY,SETFILEATTRIBUTES, SETLASTERROR,WRITEPRIVATEPROFILESECTION, WRITEPRIVATEPROFILESTRUCT, WRITEPROFILESECTION,_CALLPROCEX32W
Windows 95 删除的	DIAGOUTPUT,DIAGQUERY,DOSIGNAL,EMSCOPY, GETFREEMEMINFO,GETTASKQUEUEDS, GETTASKQUEUEES,GETWINOLDAPHOOKS,INITTASK1, K327,K329,K403,K404,REGISTERWINOLDAPHOOK, RESERVED1,RESERVED2,RESERVED3, RESERVED4,RESERVED5,SETSIGHANDLER, SETTASKQUEUE,SETTASKSIGNALPROC, WINOLDAPCALL

在 Win32 方面,下面是一个在 KERNEL32.DLL 的 Win95 版本上运行 DUMPBIN 的输出片段:

```
1    0   AddAtomA (00040475)
2    1   AddAtomW (000134aa)
3    2   AddConsoleAliasA (00014a6a)
4    3   AddConsoleAliasW (00014ab1)
5    4   AllocConsole (0001c4f2)
6    5   AllocLSCallback (00029d84)
7    6   AllocMappedBuffer (0003ea55)
8    7   AllocSLCallback (00029db7)
9    8   BackupRead (0001490d)
A    9   BackupSeek (00014733)
B    A   BackupWrite (00014928)
```

在这个输出中有两个重要的事情值得注意。首先，几个引出 API 有两种形式例如 AddAtomA 和 AddAtomW，AddAtomA 是 AddAtom 的使用 ASCII 串的版本，而 AddAtomW 是使用单一码串的等价函数。（在 Windows 95 和 Win32 中，大多数函数的单一码版本只是简单地弹出它们的参数并返回，这是因为 Win32 平台不支持单一码。）

值得注意的第二件事是在输出的每行最后是一个数字，这个数字是函数在模块中的相对虚拟地址(RVA)。这太棒了！这个引出节包含了足够的信息去把符号名字连接到代码地址上。如同后面你将看到的那样，有了名字就增加了通过几个重要的顺序来探密的容易程度。

下面的输出显示了 Windows NT 3.5 NTDLL.DLL 中的引出函数的 DUMPBIN 显示：

```
ordinal hint   name
    ...
     13   12   DbgBreakPoint (0000aa58)
     14   13   DbgPrint (0000aa5e)
     15   14   DbgPrompt (0000aaa2)
    ...
     24   23   LdrGetProcedureAddress (000082ff)
     25   24   LdrInitializeThunk (00001108)
    ...
     39   38   NtAllocateVirtualMemory (00001198)
     3A   39   NtCancelIoFile (000011a8)
    ...
     49   48   NtCreateMutant (00001298)
     4A   49   NtCreateNamedPipeFile (000012a8)
     4B   4A   NtCreatePagingFile (000012b8)
     4C   4B   NtCreatePort (000012c8)
     4D   4C   NtCreateProcess (000012d8)
     4E   4D   NtCreateProfile (000012e8)
     4F   4E   NtCreateSection (000012f8)
     50   4F   NtCreateSemaphore (00001308)
    ...
     A1   A0   NtQuerySystemInformation (00001800)
    ...
    19E  19D   RtlLocalTimeToSystemTime (00019b3c)
    19F  19E   RtlLockHeap (00011178)
```

```
1A0  19F   RtlLogStackBackTrace  (0001b120)
1A1  1A0   RtlLookupElementGenericTable  (0001a104
1A2  1A1   RtlLookupSymbolByAddress  (0001bcdf)
1A3  1A2   RtlLookupSymbolByName  (0001bb8b)
...
```

因为具有诸如 DbgPrint()、NtGreatProcess()和 NtQuerySysteminformation()一类的函数，所以 NTDLL.DLL 有许多令人感兴趣的功能。在 Windows NT 中，利用 DUMPBIN 可了解许多无资料说明的 API 是真正做创建进程、管理内存等工作的。对大多数 Windows NT 中的 API 函数而言，KERNEL32.DLL 只不过是在 NTDLL.DLL 的真正代码之上的很薄一层而已。你可能会想："这很好，但我可能不能亲自用 NTDLL.DLL。"错了！如果你在一些诸如 WPERF.EXE 的 NT 程序上运行 DUMPBIN 或 PEDUMP，你可以看到它们调用了无资料说明的 NTDLL.DLL，例如 NtQuerySystemInformation。

你可以通过检查文件所包含的一些文本串来获得对该文件更进一步的了解。最有用的文本串之一是描述域。连接器把所有你在 DEF 文件的 DESCRIPTION 行上给出的任何东西放入到可执行文件的描述域中。在 16 位 NE 文件中，描述串是非驻留名字表的第一项。以下的输出显示了 Windows 95 \WINDOWS 目录下文件的一些典型描述串。

```
RUMOR.EXE:    Party Line
WINBUG10.DLL: DLL for LZ compression functions for WINBUG
DEFRAG.EXE:   Disk Defragmenter (Optimizer)
MCIOLE.DLL:   OLE handler DLL for MCI objects
SCANDSKW.EXE: ScanDisk for Windows
CARDS.DLL:    Card Display Technology
WINPOPUP.EXE: Microsoft Windows Message Popup Application
MORICONS.DLL: MS-DOS Application Icons For Windows 3.1
CHARMAP.EXE:  Utility for easily selecting special characters.
PROGMAN.EXE:  Windows Program Manager 3.1
RUNDLL.EXE:   Turn a DLL into an App
WINFILE.EXE:  Windows File System 3.1
DIALER.EXE:   Microsoft Windows Telephony Dialer
```

在 32 位 PE 文件中，连接器把描述串放在.rdata 节的某个地方。不幸的是，对描述串的放置似乎没有任何固定的格式，如果你想看这些字符串，你最好对.rdata 节作一下原始的 16 进制转储，并查一下嵌入的.ASCⅡ字符串。另外由于 Microsoft Win32 工具通常不需要.DEF 文件，因此你会发现许多文件没有描述字符串。

通过转储一个 EXE 或 DLL 获得有用字符串的另一个地方是资源节。在 Win16 和 Win32 编程中，你既可以通过序号也可以用名字来指定资源。有时对话框有有趣的名字或对话框之外的隐藏控制。字符性资源一般包含着你通常未见过的有用东西。例如，在 Microsoft 游戏 TAIPEI.EXE 中，如果你胜了，程序将奖励你一条谚语。如果你想看到所有可能的谚语，你要么能够完全驾驭这个游戏，要么你能像我转储出字符表那样进行"欺骗"以达到目的。

要找到文件中的资源，是有许多方法的。像 Borland 的 Resource Workshop 一

样的程序可以让你交互地阅读和编辑任何文件中的资源。如果你喜欢使用命令行，Eclectic Software 的反汇编器(Win2Asm)有一个功能，可在可执行文件二进制码的资源中进行阅读，它产生一个适当的.RC 文件，如果你需要的话，可以把该.RC 文件返回给资源编译器。

在用文件转储程序时，最好的情况是你遇到的文件包含了调试信息。调试信息包含了有关程序的各种有用信息。现代编译器的调试信息包括你所用的所有变量和函数名、资源文件名、结构定义的格式、类的层次结构以及其他的许多东西。简言之，对分析文件而言，调试信息几乎和源代码一样的好。

Borland 的 TDUMP 能够转储出 Borland 的两种调试信息(16 和 32 位)，另外还可转储 Microsoft C7 调试信息。Microsoft 用户还可以用 CVDUMP 把 CodeView 信息分解成为可读的文本。除了 CodeView 信息，Microsoft 的 32 位编译器还产生另一种称为 COFF 的调试信息，(第八章中的 DUMPBIN.EXE 和 PEDUMP.EXE 可以分解 COFF 调试信息)。最后，来自于 Nu-Mega SoftIce/W 的 DBG2MAP 可以把 Borland 和 Microsoft 32 位调试格式创建成人工可读格式的.MAP 文件。

.SYM 文件是另一种对探密有益的调试信息形式。尽管相对来说.SYM 文件有点古老和原始，但如果你偶尔用一下，它还是有帮助的。Microsoft 把用于调试一些二进制文件的.SYM 文件作为 Windows 95 SDK 的一部分。另外，对大多数人想要检测的系统 DLL，还不存在好的、有用的.SYM 文件。

调试信息(.SYM 除外)首先明白地告诉你可执行文件是用哪个公司的连接器产生的(通过查看编译器运行时间库放入程序数据区的版权字，你也可以发现这个信息。)然而，更重要的是，你可以得知可执行文件所有的函数名和变量名。除了函数名和变量名外，调试信息也包含这些符号的地址。如果需要求诸于反汇编，有了符号名，会成倍加速你的成功。

除了符号名外，调试信息可能还包含变量类型、函数的参数表以及结构和类的格式。简而言之，调试信息包含了几乎所有你不想让对手知道的关于程序的一切信息。有一次，当我告诉一位程序员在他的代码中的一个 GP 错误及所在行数时，他震住了！我是从公告板上获取该程序的，并没有源代码，该程序所带的调试信息就足以使我指出这个问题及所在的源代码行号。你的对手就未必这么好了！这就是为什么对你很重要的是：不要把调试信息带入你的产品。许多公司，包括 Microsoft、Borland 和 Delrina 过去在这点上做得不够，你可以通过在 Windows 3.1 的 SOUNDDREC.EXE 上运行 TDUMP 或 CVDUMP 来看到这一点。

即使你在 EXE 和 DLL 中不留调试信息，通过转储文件内容和分析结果，仍可了解文件的许多东西。在 1993 年 7 月的《Microsoft Systems Journal》上，我发表了一个名叫 EXESIZE 的实用程序，它可以扫描 16 位 NE 文件，找出由于低收率或懒散的编程而浪费的空间。EXESIZE 可以确定文件对齐尺寸是否应该小一些、是否产生了低效的实模式代码以及你在文件中是否留有调试信息等。在某些情形下 EXESIZE 还发现了浪费超过 100K 的文件。经过一段时间后，我发现大多数情况下，浪费了很多空间的文件是由懒散或无知的程序员编制的。相反地，如果一个可执行文件通过了所有的 EXESIZE 检测，那它可能是由一个十分谨慎并且重视细

节的教授编写的。

当我们着眼于 EXE 和 DLL 的转储时，没有注意在其它相关文件中可以发现的极有价值的信息，尤其是 OBJ 和 LIB 文件中包含有相当多的有关一个给定模块（或模块集）的信息。Borland 的 TDUMP 可以分解 Intel OMF OBJ 文件以显示公共或外部符号以及段名等等，Symantec C++包含了 OBJ2ASM 实用程序，它可符号化反汇编出包含在 Intel OMF OBJ 文件中的代码。Microsoft 的 DUMPBIN 和我的 PEDUMP 程序都可以完成通用的 COFF OBJ 和 LIB 文件转储，DUMPBIN 甚至还可以反汇编 COFF OBJ/LIB 文件。

用侦探工具探密

尽管文件转储可能是丰富且十分有趣的。但它不会告诉你在实际问题中有关代码的全部内容。使你可以侦探程序和操作系统相互作用的工具通常更适于这项任务。最有名的 Windows 侦探工具是消息侦探程序：Microsoft 的 SPY 和 Borland 的 WINSIGHT。消息侦探程序可以显示窗口所接收的消息，以及程序是如何响应这些消息的。

尽管这些消息可能已很有用了，但程序员通常需要更多的信息，以得到他们试图查明的根本所在。诸如 Nu-Mega 的 BoundsChecker for Windows 和 Periscope 的 WinScope 一类程序，把侦探工具提到了一个新的水准。除了窗口消息外，这些程序还拦截一个程序或它的 DLL 产生的 API 调用。另外，一些侦探程序还监视和记录钩式回调（hook callback）、TOOLHELP 通知和其它回调（callback）。这些程序的想法是在控制进入或退出程序代码之处（窗口过程，API 调用以及其他等等）放“一枚探针”，通过这些边界的信息被定位在固定位置。例如，所有的窗口过程都通过在堆栈上（HWND 在[BP+0E]处，MSG 在[BP+0C]等等）一组固定参数来调用。侦探程序可以利用这点来存储、分析和显示信息。

最好的侦探工具是那些对所侦探的代码无需任何修改的程序。这些程序只依赖可执行文件中的信息和它们插入“探针”点的调用。如同本节后面我要显示的，这样可使侦探程序侦探任何的 EXE 或 DLL，甚至那些你用某些方法也不能重连接和修改的程序。

另外一些侦探工具要求你对所侦探代码进行重连接，这些工具通过“欺骗”连接器，把程序的 API 调用指向它本身的代码而不是操作系统的 DLL 来进行工作。还有一类与此相近的工具是修改已连接过的可执行文件，其效果是一样的。侦探程序把 API 调用指向工具本身代码，而这些代码记录了在控制传给操作系统之前的这个调用。

对 16 位 Windows 应用程序而言，侦探工具是多种多样的。尽管 BoundsChecker/W（BCHKW）的最初目的是发现错误，但它也可以通过拦截所有 Windows API 调用和程序产生的对某些 C 库的调用，以及产生有效参数来实现侦探。由于 BCHKW 已经做了拦截所有 API 函数调用的艰苦工作，因此就不需要做太多的工作以使 BCHKW 把信息保存在跟踪缓冲区中。为了给出导致错误的事件序列的一

个清楚图面，BCHKW 也观察窗口和会话信息、钩式回调、TOOLHELP 通知和其它种类的回调。

如果你选择把跟踪信息存入磁盘，你可以用 BCHKW 的 TVIEW 程序，以便得到你程序动作的两个视图：可扩展视图和可折叠的层次结构视图。TVIEW 包含了多种事件过滤器，这些过滤器可以做诸如除去你可能不感兴趣的消息和重复的 API 序列。一个典型的这种序列可能是：GetMessage/TranslateMessage/DispatchMessage/Window Message/DefWindowProc。

尽管 BCHKW 利用调试信息来实现它“找错”的角色，但这对它的侦探功能来说，调试信息不是必需的。作为结果，你可以用任何 Windows 程序来运行 BCHKW，而不是只能用你自己的正在开发过程中的程序。

另一个较为流行的、用于 16 位程序的侦探程序是 Periscope 的 WinScope。不像 BoundsChecker/W 在某一时刻只能处理一个程序，WinScope 是一个系统宽度(Systemwide)的侦探工具。WinScope 可显示出所有的 API 调用、钩式回调和发生在系统中任何位置的消息。这有时是很有用的，而有时导致信息的溢出。

幸运的是，WinScope 为你所希望的侦探提供了很高级的用户选择性，你可以允许或不允许在单个 API 或一组 API 上进行侦探，你也可以侦探或不侦探窗口消息或钩式回调。像 BoundsChecker/W 一样，WinScope 可以保存一个 API 的远指针参数所指向内存的拷贝，这就使你可以看到被传送到 CreateWindow 和 GetPrivateProfileString 等的字符串和数据结构。WinScope 也可以保存每个事件的计时信息，允许 WinScope 作为一个原始的描述器(profiler)来运行。WinScope 用 NE 文件的信息去钩 API 调用，所以你不必重新链接你所要侦探的代码。

如果你为了省钱而宁愿牺牲可用性和特点，你可以考虑使用 Microsoft 的 API 参数描述器(profiler)，尽管这种既有 16 位版本也有 32 位版本的侦探工具出现在 Windows NT SDK 中，但几乎没人知道它的存在。

Microsoft 的描述器较为原始，它要求你要对被检查的任何 EXE 或 DLL 进行修改。这种侦探工具的核心是一组 DLL(ZERNEL. DLL、ZSER. DLL、ZERNEL32. DLL、ZSER32. DLL 等)。每个 DLL 都有一个和一个操作系统 DLL 相同的基本文件名，但把第一个字母改为 Z。这些 DLL 有一个小存根，是为了保留被取代的 DLL 所引出的 API，例如 USER. EXE 引出函数 CreateWindow()，这样 ZSER. DLL 也引出一个 CreateWindow()函数。你可以用 APFCNVRT 程序(或用为 Win32 用户的 APF32CVT)把你的程序或 DLL 连接到这些 DLL 上。APFCNVRT 和 APF32CVT 要修改你的程序，这样它可以从参数描述表的 DLL 中引出其函数，而不是从操作系统的 DLL。当你运行一个修改过的程序，所有对 DLL 的调用在被传送到操作系统 DLL 之前，先经过参数侦探 DLL。参数描述器把它收集到的信息存入磁盘以供查看。对于 32 位程序，Microsoft 提供了一组可选择的 DLL，它可以做真正的描述器工作，而不是 API 登录。

除了 32 位版本的 Microsoft 参数描述器外，Nu-Mega 的 BoundsChecker32 (BCHK32)程序(在 NT、Windows 95 和 Win32 下)也可以在 Win32 程序中侦探 API 调用和窗口消息。从侦探 API 的目的上看，BCHK32 类似于 BoundsChecker/

W，然而它与它的 Win16 同类相比，还有一些新特点。首先，当一个 API 调用失败时，API 通常通过 SetLastError()来存储一个错误代码，以指示出失败原因。BCHK32 知道何时 API 失败，并且记录出错代码。第二，由于 Win32 支持线程，BCHK32 为每个 API 调用和窗口消息都保存了线程代码标识符(ID)。TVIEW 程序用线程信息去提供附加的过滤选择，例如对特殊的线程只显示事件。

我在这儿要提到的最后一个 Win32 侦探程序，是在第十章中我编写的 APISPY32，尽管 APISPY32 的特点不如 BoundsChecker32 那么全面，但它提供了 API 侦探的基本功能(包括显示函数的参数和返回值)，并且易于扩充以侦探任何你所想要侦探的 Win32 DLL，而且不要求对你的程序作任何修改。

评价一个侦探工具应考虑的一个关键，是各工具允许你观察系统的哪些部分。WinScope 可在对一打的 Windows DLL 的调用(USER，KERNEL，GDI 等)上进行侦探，更重要的是，WinScope 具有可以通过你所写的描述去侦探其它 DLL 的功能。BoundsChecker/W 可以侦探 10 个标准 DLL(粗略上与 WinScope 所包含的省缺部分相同)。而 Microsoft 参数描述器只能观察三个主要的系统 DLL(USER、KERNEL 和 GDI)。BoundsChecker32 目前可侦探 KERNEL32、USER32、GDI32 和 ADVAPI32，以及其他几个重要的 DLL。

在 API 侦探过程中，侦探工具不能给你显示正在进行的事情和使你迷惑不解的事情之间的差别。根据经验，你所记录的数据点(即 API 调用、窗口消息、钩式回调、TOOLHELP 通知等)越多越好。例如，如果你想搞清楚一个程序是如何用 TOOLHELP 来完成某一动作的，则 Microsoft 的参数描述器是没有用的。

以上对 API 侦探工具的描述已够完备的了！让我们来解决一个实际问题，这样你就可以在实际行动中看到如何使用这些工具。在编程论坛上我多次看到与 Windows 的 CLOCK.EXE 程序有关的问题，最普遍的询问是："如何可以让我的程序如同 CLOCK.EXE 一样，能在有标题与无标题栏之间进行转换?"。通过用侦探程序来检查在 CLOCK 标题栏转换时的 API 调用，你可以找到答案。这里，我将用 Windows NT 中的 32 位 CLOCK.EXE，我也可以轻易地用 16 位的 CLOCK.EXE。(Windows 95 中不包含分离的 CLOCK 程序，但你可以在 Windows 95 中运行 NT 的 CLOCK.EXE)。

至于工具，尽管我所提及的任何工具都可以，但我选用 BoundsChecker32/NT。如果你有这些工具中的一个，你可以跟着我后面的步骤。当然对于理解关键点，这并非是必不可少的。

第一步是运行程序并且收集跟踪信息。为了做到这一点，可以运行 BoundsChecker，从 FILE|LOAD 对话中选择 CLOCK.EXE，然后选择 Run。CLOCK 启动之后，进入 CLOCK 的 Settings(设置)菜单，选择 NO Title。(我想，在 CLOCK 一开始运行时，它是带有标题栏和菜单的。)然后关闭 CLOCK 程序。

下一步就要检查跟踪输出，并找到程序对 NO Title 命令的响应点。下面的结果显示了有关跟踪部分的一个文本文件。

```
WNDMSG:  HWND:0049016E  MSG:WM_COMMAND(0111)  WPARAM:00000006  LPARAM:00000000
APICALL: GetWindowLong(HWND:0049016E, WINDOWLONG:GWL_STYLE)
APIRET:  GetWindowLong returns LONG:14CF0000
APICALL: SetWindowLong(HWND:0049016E, WINDOWLONG:GWL_ID, DWORD:00000000)
APIRET:  SetWindowLong returns LONG:BE00F2
APICALL: SetWindowLong(HWND:0049016E, WINDOWLONG:GWL_STYLE, DWORD:14840000)
APIRET:  SetWindowLong returns LONG:14CF0000
APICALL: SetWindowPos(HWND:0049016E, HWND:00000000, DWORD:00000000,
           DWORD:00000000, DWORD:00000000, DWORD:00000000,
           SWP_FLAGS:00000027:
           SWP_NOSIZE:SWP_NOMOVE:SWP_NOZORDER:SWP_FRAMECHANGED)
```

你也许会要问,“我怎么知道去哪儿找我所需要的信息呢?”答案是非常的简单,无论你从菜单上选择什么,Windows 给你的程序发出 WM_COMMAND 消息,这样,为了发现事件序列,你所要做的第一件事就是去找到字符串 WM_COMMAND。如果你是按上面给定的步骤去做的,那么在整个事件登录中只有一个 WM_COMMAND 消息。但为了周全,让我们检验一下输出中的 WM_COMMAND 是不是正确。

在一个 WM_COMMAND 消息中,WPARAM 参数保存有被选中菜单项的 ID。在输出中,WPARAM 是 6。如果你用 Resource Workshop 或其它类似的程序检查 CLOCK 的资源,你会看到 No Title 项的 ID 是 6。现在我们可以确保我们看的是正确的事件登录部分。

在接收了通知 CLOCK 关掉标题栏的 WM_COMMAND 消息之后,CLOCK 所做的第一件事是调用 GetWindowLong()并传送 GWL_STYLE 参数。下一行的输出表明 GetWindowLong()返回一个 0x14CF0000 的 DWORD。这个值代表传送给 GetWindowlong()的 WS_xxx 风格位组,你可以通过查看 WINDOWS.H 来了解这些位的含义。

```
#define WS_VISIBLE          0x10000000L
#define WS_CLIPSIBLINGS     0x04000000L
#define WS_BORDER           0x00800000L
#define WS_DLGFRAME         0x00400000L
#define WS_SYSMENU          0x00080000L
#define WS_THICKFRAME       0x00040000L
#define WS_MINIMIZEBOX      0x00020000L
#define WS_MAXIMIZEBOX      0x00010000L
                            ===========
                            0x14CF0000
```

现在,暂时先忽略输出中接下来的两行(我将在稍后再谈它们)。当 CLOCK 利用 GetWindowlong()接收到它的风格位组之后,它转过来调用 SetWindowLong()设置一组稍微不同的风格位。在这个调用中,风格位组为 0x14840000,它看起来像 CLOCK 接收到它的 WS_xxx 风格位组,修改了一些位,并将风格位组反送给窗口。但 CLOCK 改变了什么风格呢?把原先的 0x14CF0000 与新的 0x14840000 相比较,新风格 DWORD 与原值相比丢掉了下列风格:

```
#define WS_DLGFRAME          0x00400000L
#define WS_SYSMENU           0x00080000L
#define WS_MINIMIZEBOX       0x00020000L
#define WS_MAXIMIZEBOX       0x00010000L
```

这与 CLOCK 的行为是一致的。我们选择了 No Title 菜单项，系统菜单和最大化、最小化按钮也消失了。

现在，我再回到我在前面越过的两行上。第一行是对 SetWindowLong()的调用，这行看起来是把窗口控制 ID(GWL_ID)设置为 0。只用信息的这一位，你也许会对代码的意图感到困惑，接下去，我将让你知道秘密所在。所有的窗口都有一个内部域，它要么是一个菜单句柄，要么是一个控制 ID。高层窗口(如 CLOCK.EXE 的窗口)用这个域保存菜单句柄(HMENU)。子窗口(如对话框控制)，用这个域保存控制 ID。如果想要对此得到正式的证实，请参考在 CreateWindow()说明资料中的 hMenu 域描述。

知道了这个事实之后，我们可以看到 CLOCK.EXE 把它窗口的 HMENU 设置为 0。如果 CLOCK 用 SetMenu()来改变它的 HMENU 值，或许会更好(也更清楚)一些。然而，CLOCK.EXE 作者不用 SetMenu()可能有潜在的原因，一个可能的原因就是 SetMenu()要强迫菜单域被改写，以反映菜单的变化。

在输出的最后一行，是对 SetWindowPos()的调用。SetWindowPos()是一个通用的例程，它可以移动窗口、改变它们的 Z 顺序(Z-order)，或者促使 Windows 重新计算和重画窗口。它的最后一部分(促使 Windows 重新计算和重画窗口的部分)可能与在 CLOCK 中不用 SetMenu()有关。下面就是为什么在 CLOCK 中不用 SetMenu()的原因：在 CLOCK 接发风格位和 HMENU 之后，它需要用新的风格重画它本身。调用 SetMenu()将引起窗口的各部分重画，而随后的对 SetWindowPos()的调用将引起窗口再次重画，这就导致窗口“颤动”。CLOCK 的作者可能也想到了通过用 SetWindowWord()直接设置窗口新的 HMENU 值，就可很好地减少“颤动”，因为他们知道窗口将会被后面的 SetWindowPos()调用重画。

CLOCK 传送给 SetWindowPos()的参数很有意思，非零参数只有 HWND 和 SWP_xxx 标志。前三个标志告诉 Windows：CLOCK 不希望改变窗口的大小、屏幕位置和 Z 顺序。最后一个参数是最重要的一个，它告诉 Windows 窗口的边框被改变了。这就迫使 Windows 重新计算用户区和非用户区，并重画整个窗口。如果我要显示更多的事件跟踪结果，你将看到 SetWindowPos()调用使消息与 API 调用混乱，而 API 的调用不下几百行。我真心希望你能自己把它检查出来。如果你也想自己搞清事件跟踪镜象的事实，Microsoft 在 Win32 SDK 的 SAMPLES\DDEML\CLOCK\目录下提供了 CLOCK 的源代码。

通过检查 CLOCK.EXE，我们已看到了侦探工具怎样给你显示一个可视效果(去掉标题栏)是如何实现的。对于了解表层之下所进行的、隐藏于视线之外的东西，侦探工具也是十分有用的。程序员中最爱用的技巧是给他们程序增加一些无说明的动作或功能。比如，程序员也许想让程序能够把调试诊断写到一个文件中，由

于这个功能只用于极少数情况，所以编程者不想在用户界面上增加额外的信息，以免最终用户迷惑。还有，把这个选择加到用户界面上则需要描述和说明它，这样就会仅为一个极少用的特点花额外的宝贵时间。最终结果就产生了没有说明的功能。

为了寻找无说明的功能，你可以用的一个技术是去查看程序的.INI 文件中的各条目，这个.INI 文件通常是不出现的。换言之，程序寻找一个特殊的.INI 条目，但在保存这些选择时却不写入条目的数值。要使用无说明的条目，程序用户必须知道该条目的存在，并且手工地把它加入到.INI 文件。尽管这个讨论着眼于.INI 文件，但同样也适用于 Win32 的登记簿(registry)，在大多数情况下它可替代.INI 文件。

用保留了一个 API 指针参数所指向内容的副本的侦探工具来找到一个像我前面所描述的情况是很容易的。尽管我可以使用 BoundsChecker/W 或 WinScope，但我选择显示一个使用 Microsoft 参数描述器(Parameter profiler)的例子。下面的输出显示了通过运行 Windows 3.1 WINMINE.EXE 得到的事件跟踪的一个片断：

```
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Ypos"  50  "winmine.ini"
01|APIRET:GetPrivateProfileInt 105
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Ypos"  50  "winmine.ini"
01|APIRET:GetPrivateProfileInt 105
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Ypos"  50  "winmine.ini"
01|APIRET:GetPrivateProfileInt 105
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Ypos"  50  "winmine.ini"
01|APIRET:GetPrivateProfileInt 105
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Sound"  0  "winmine.ini"
01|APIRET:GetPrivateProfileInt 0
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Sound"  0  "winmine.ini"
01|APIRET:GetPrivateProfileInt 0
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Sound"  0  "winmine.ini"
01|APIRET:GetPrivateProfileInt 0
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Sound"  0  "winmine.ini"
01|APIRET:GetPrivateProfileInt 0
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Tick"  0  "winmine.ini"
01|APIRET:GetPrivateProfileInt 0
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Tick"  0  "winmine.ini"
01|APIRET:GetPrivateProfileInt 0
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Tick"  0  "winmine.ini"
01|APIRET:GetPrivateProfileInt 0
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Tick"  0  "winmine.ini"
01|APIRET:GetPrivateProfileInt 0
01|APICALL:GetPrivateProfileInt "Minesweeper"  "Menu"  0  "winmine.ini"
01|APIRET:GetPrivateProfileInt 1
... 3 more "Menu" calls not shown...
```

在输出中每个“APICALL:”行的第一部分是调用嵌套级。在此输出中，所有的调用都在最高层 01 级上，这意味着 WINMINE 不能在另一个 API 函数中间调用函数。在“APICALL:”之后是函数名，后跟它的参数。Microsoft 参数描述器可以显示实际的 ASCII 码字符串，而不是指针值(例如 0x10b7:003A)。GETPrivateProfileInt 用了三个 LPSTR 参数，因此该功能在这种情况下特别有用。

检查一下 WINMINE 事件跟踪的片断，注意代码寻找每个 INI 条目寻找了四次。为什么如此，对我来说是个谜，尽力搞清如此奇怪的行为也是探密的乐趣之一。

先把这个行为放在一边，看一下每个 APICALL 行的第二参数。这个参数是在一个 INI 节中的条目名字。第一组 APICALL 寻找名为 Ypos 的条目，如果你查看一下 WINMINE. INI 文件，你就会看到，事实上存在一个名为 Ypos 的条目。然而，如果你接着看下面三个条目(Sound、Tick 和 Menu)，你在 INI 文件中任何地方都不能找到它们。你再进一步查看它的事件跟踪(在此处作为关闭序列的一部分，WINMINE 写出新的 INI 文件值)，你也同样找不到 Sound、Tick 或 Menu。

我们所揭示的是影响 WINMINE 动作的三个无资料说明的方法。我曾亲自用 WINMINE 给 WINMINE. INI 增加了三个条目。尽管我用 Tick 没有得到任何结果，但增加条目“Menu＝1”引起 WINMINE 不显示主菜单，增加条目“Sound＝3”(或更大的数字)，使得你在游戏中赢或输时，让 WINMINE 响起一首短歌。

用反汇编探密

尽管反汇编复杂且困难，但它通常是分析疑难算法或技术的唯一方法。

反汇编一个程序或 DLL 并不一定只是要分析别人的代码。当你在你的代码中遇到一个不能立刻从源程序中识别的奇怪错误时，知道怎样把高级语言和汇编语言联系起来是很重要的。反汇编你自己的代码，也让你明白编译器是否已为一个频繁使用的程序产生了优化的代码。然而另一种你可能反汇编你自己代码的情况是：当你的程序在用户执行时，出现神秘的 GP 错误。如果用户能给你程序中断的地址，你可在这个地址处反汇编你的代码，以便看看程序正在做什么。

在继续下面之前，我想强调一下，反汇编不是为那些不愿在细节上下功夫的人而设的。如果汇编代码使你感到陌生的活，反汇编也不适合你。你必须了解汇编语言程序或乐意去了解它。这并不是说，你必须用汇编语言来编程。虽然工作在高级语言这一级上相当轻松愉快，但是你必须愿意工作在很低的机器级和寄存器级上。

反汇编器的选择有时受你想处理的文件类型的限制。为了完成各种任务，反汇编程序需要知道相当多的可执行文件格式。一个最简单的文件反汇编程序并不比一个文件转储程序多费多少头脑，文件转储程序可以认为是一个用原始字节作为输入并输出汇编记忆码的反汇编“机械”，它的一个完美例子是 Visual C_{++} 32 位编辑连接器中的 DUMPBIN/DISASM 选择。更先进的反汇编程序能在用程序地址来与一个符号名相联系的符号数据中读。这些反汇编程序能产生用了真正的变量和函数名，而不是十六进制地址表示的汇编清单。

基于 PC 机，最有名的反汇编程序大概是来自 V-Communication 的 Sourcer。Sourcer 本身只能处理 DOS 的 EXE 和 COM 文件，若有产生描述文件的附加部件，Sourcer 也可处理 16 位 NE 文件、VxD(LE 文件)和 Win32 PE 文件。Eclectic Software 有 Win2Asm 反汇编程序，它处理 NE、LE 和 PE 文件。RJ Swantek 有 DisDoc Profesional 反汇编程序，它能处理的文件类型和 Win2Asm 相同。如果你只考虑 Win32 文件和价格，它是很难击败来自于 Microsoft 的 Win32 SDK 的 DUMPBIN

程序的。过一会我将给出一个使用 DUMPBIN 的简单例子。

你或许想知道我在本书中会用什么呢?我有我自己设计的两种反汇编程序(一个是对 Win16 NE 文件的,另一个对 Win32 PE 文件和 VxD)。尽管它们不像 Sourcer 一样适用于多种环境,但它们仍能很好地为我服务。写我自己的反汇编程序的好处是:我能利用我学过的一些技巧修改它们,并能使它们从种种源程序中读符号信息。目前它们都还没有投入市场。

如果你只用我下面讲的反汇编技术来进行简单的修修补补,你或许只用你调试器中的反汇编程序就行了,这假定你不是用的一个不含汇编窗口的集成开发环境调试器。一些调试器把它们窗口的内容转储到一个文件中。通过把几个反汇编窗口的内容转储到一行中,你能得到一个有一定意义的列表。然而,这样做很乏味且费时,特别是当被处理的例行程序需要调用程序中别处的函数时。如果你真正要反汇编,那么,还是用一个真正的反汇编程序,如 Sourcer、DisDoc 或 Win2Asm。考虑到它们的强大功能,它们实际上也并不昂贵。

反汇编的学习和技术

反汇编一段代码并非只有一种方法,我这儿谈谈我用什么方法,如果你还有一些其它方法,那么你试试用它!我反汇编的基本方法可总结为"分块解决"。从反汇编程序的原输出端开始,我并不一下子解决一个大块中的全部函数或部分,相反,我通过一系列步骤,把原始列表分成一些较小的易管理的块,然后我再各个击破。我反汇编的最终目的是把汇编列表变成一段带注释的 C 代码。

下面的分解一段代码各个步骤的重要性和顺序,根据你处理的代码而可能有所改变。首先,我整体上描述一下我反汇编一个函数的步骤。接着,我将转到标识参数、局部变量、分枝语句、函数调用等的细节上。最后,我将举例怎样把一个原始的反汇编列表转换为可用的东西。下面讨论反汇编一个函数的步骤:

步骤 1:反汇编文件

用反汇编程序运行可执行代码以得到一个列表文件。若反汇编程序带有附加的符号输入(例如从.SYM 文件或调试信息),那么把它送给反汇编程序。

如果你只对某个函数感兴趣,那么删除该函数前面或后面的列表代码部分,或许对寻找该函数有益,并将使这个文件在你的编辑器中更易于管理。例如,我曾经用到过有 3MB 大的一个反汇编列表,这使得我的编辑器花费很长时间去装入和存储文件。减少一些不必要的代码,确实能加速处理进程。

步骤 2:标记已知的实体

遍历该函数,把所有已知的项用更具描述性的名字标记起来。已知项,这里指的是函数的参数、局部变量和全局变量。意即先从容易的做起。当玩拼图游戏时,大多数人先拼边和清楚部分等容易拼的部分,这样做,也能为你剩余的部分提供一些启发。这种观念也适用于反汇编。

假设你知道了函数的参数和调用约定，你就可很容易地标识函数的基于堆栈的参数，并用含有一定含义的名字（例如 hWnd）取代它们。（稍后我将讨论标识基于堆栈的变量）。

除非在可执行码中有调试信息，否则为局部变量确定名字比标记参数要困难得多。如果你不能弄清楚每个名字，也不用担扰。如果某个局部变量超出了你的想象，你可以通过一切手段，用一个有含义的符号化名字把它替换掉。

如果你有全局变量的符号信息，反汇编程序可能已用符号名代替了全局变量地址，但如果它没有符号信息，你现在必须手工标上。

步骤 3：分割指令序列

反汇编列表往往包含一长串连续的指令序列。我发现，在逻辑上应连在一起的一个指令序列与另一个序列之间，插入空行是很有好处的。这听起来比较模糊，但做起来并不难。这样序列的一个例子是函数的序码，另一个逻辑上应连在一起的指令序列是把参数压栈并调用另一个函数的代码，第三个逻辑指令序列是执行一些计算并将结果存入一个变量的一段代码。一个有用的（但不是绝对的）原则是试着去创建能在程序的源代码中形成一个语句的指令序列，然后在每组指令之间插入一个空行，在外观上将指令列表分割开来。

在以后的反汇编进程中，你可能需对一些分枝语句进行译码。在高级语言中，这些语句是 if、while、do、switch 等等。我发现在一个条件转向或非条件转向指令后加一个空行，就更容易理解指令列表。若你的反汇编程序不能自动做到这一点，那你就自己做。我习惯于用一个编辑器的宏来寻找以 J 开头的指令，然后在这条指令后加一空行。最近，我已把我的反汇编程序修改成能自动完成这个功能的了。（这也是我为何愿用我自己的反汇编程序的原因之一）。

步骤 4：添加文字字符串

如果函数看起来用到一些文字字符串值，那么就增加包含了该字符串的注释，并把它放在用了该字符串的函数调用附近。以后它有助于把几行汇编代码压缩成一个 C 语句。

步骤 5：把指令压缩成单一的 C 语句

把函数调用和中断压缩成单一语句，在这一点上，函数应分成大量的小块。找到包含对你已知其名字和参数的函数调用的指令序列，研究什么要被压栈，并试着构造 c 函数的参数。

步骤 6：标识分枝语句

标识并把条件转向语句转换成相应的高级语言语句。如果你看到一个 TEST 或 CMP 指令后面紧跟一个条件转移指令（例如 JE），则你或许可以把它看成是高级语言的 if 语句。一条 Jxx 指令转向的入口往往是一个复合语句的结束。在 C 语言中，一条复合语句是指{}括起来的所有东西，而在 Pascal 语言中，一条复合语句

是用 BEGIN/END 括起来的。

如果你看到一个长系列的测试和条件转移，你或许可以把它看成是 C 语言里的 Switch 语句或 Pascal 里的 Case 语句。弄清条件分枝代码是一件比较棘手的事情。用高级语言的 if 语句进行多次测试，能真正弄清产生的汇编代码正在做什么。像下面的 C 语句：

```
if ( (GetModuleHandle("MYDLL.DLL") != 0)
     && ( (hWnd != GetDesktopWindow()) || ( styleFlags & WS_POPUP)) )
```

产生条件转向、把临时结果存储在寄存器里等一系列指令。盯着一个数字或一条指令一个多小时却仍不能理出头绪，不知道代码要干什么，这种情形并不少见。这就是我为什么把探密作为最后依靠的原因。

步骤 7：必要时重复

必要时重复前面的步骤，这听起来很令人乏味，但实际上并非如此；这是一个迭代过程。你遍历一遍代码，尽可能利用你所掌握的信息，尽可能去做。然后，返到前面的步骤，看看图面变成什么样了，接着再做一遍。通过解决某个疑难问题，可能使得许多问题都迎刃而解了。从某方面看，探密像在玩“连点”游戏，你有越多的点，连接的点就越多，图象的剩余部分就越清晰。

弄清常用代码序列和约定

前面我们已广泛地讨论了怎样反汇编一个函数，现在我将检验一些常用代码序列和产生代码约定，这将有助于你正确地把原始汇编代码转换成相应的高级语言语句。

标识函数和过程

从一个反汇编程序的原始输出端看，第一件事是要弄清楚一个函数（或 Pascal 类型的过程）在什么地方开始和结束。找一个函数的开始，最简单的办法是寻找被编译器产生的标准序码。对 16 位代码来说，这些标准序码是下面这个进程的某些变种：

- 把原始的 BP 寄存器存储到堆栈上
- 把堆栈指针赋给 BP 寄存器
- 把堆栈指针减 1，为局部变量留空间
- 把调用函数寄存器变量存入堆栈

用汇编语言表示，即为：

```
PUSH    BP      ;; Save caller's BP frame.
MOV     BP,SP   ;; Set up new BP frame.
SUB     SP,XX   ;; XX is the number of bytes need for local variables.
PUSH    SI      ;; DI and SI are commonly used as register variables.
PUSH    DI
```

或者，当用 80286 或更好的代码时为：

```
ENTER   XX,0    ;; XX is the number of bytes needed for locals
PUSH    SI      ;; DI and SI are commonly used as register variables
PUSH    DI
```

这些堆栈结构是编译器为那些只在 16 位保护模式下运行的代码产生的。在旧的实模式下，当在内存中来回移动一个段时，Windows 本身需多次进出一个程序的堆栈。因为进出一个包括远近函数调用的程序堆栈是相当困难的，所以编译器通过这个奇(odd)BP 堆栈结构来帮助进出。当奇 BP 结构代码生成被许可时，所有远程函数在 BP 寄存器压栈之前先把它加 1(近程函数不影响 BP)。在远程函数收尾程序恢复 BP 寄存器的原始值之后，代码把 BP 寄存器的值减 1。当进出栈结构时，如果 Windows 看见被保存的 BP 是奇数值，就知道这是一个远程函数。一个奇 BP 类型远程函数的标准栈结构看起来像这样：

```
INC     BP      ;; Indicate a far frame.
PUSH    BP      ;; Save caller's BP frame.
MOV     BP,SP   ;; Set up new BP frame.
SUB     SP,XX   ;; XX is the number of bytes need for local variables.
PUSH    SI      ;; DI and SI are commonly used as register variables.
PUSH    DI
```

对于 32 位程序，标准序码看起来像如下：

```
PUSH    EBP         ;; Save caller's EBP frame.
MOV     EBP, ESP    ;; Set up new EBP frame.
SUB     ESP, XX     ;; Make space for local variables on stack.
PUSH    ESI         ;; ESI, EDI, and EBX are commonly used as
PUSH    EDI         ;; register variables.
PUSH    EBX
```

或者：

```
ENTER   XX,0    ;; XX is the number of bytes needed for locals.
PUSH    ESI
PUSH    EDI
PUSH    EBX
```

前面的序列是完整的序码。在实际代码中，序码的部分或全部可被省去或不同：

- 如果函数代码不改变一个寄存器变量的寄存器(例如，ESI、EDI 和 EBX)，则在序码中就没有保存该寄存器的操作。还有，在 32 位代码中，EBX 有时被用作寄存器变量，而在 16 位代码中，它通常不用作寄存器变量。
- 在 16 位代码中，如果函数不带任何参数或不用任何局部变量，编译器可能省略 PUSH BP/MOV BP,SP 序列。
- 在 32 位代码中，即使函数带参数并用局部变量，编译器可能仍建立 EBP 结构。

386 或更高档 CPU 的 32 位寻址模式，允许编译器用 ESP 寄存器来寻址参数和局部变量，例如：

```
MOV EAX,[ESP+1C].
```

辨别函数的收尾也有点难度。如果编译器进行了优化，则函数内部可能有好几处在做返回到调用者的 RET 或 RETF 命令。设想函数只在其末尾处有一个单一的收尾，则完整的 16 位收尾程序将看起来像这样：

```
POP     DI  ;; Restore caller's register variables
POP     SI
LEAVE       ;; or ADD SP,XX / POP BP
RETF        ;; far return. Near return is a RET.
```

对 32 位代码，收尾程序看起来像这样：

```
POP     EBX ;; Restore caller's register variables.
POP     EDI
POP     ESI
LEAVE
RET         ;; A 32-bit near return.
```

当确定一个例行程序的结束和另一个的开始之处时，记住，在一个例程结束的地方，你很可能会找到另一个的开始。如果你看到一些代码像是结尾程序，那么通过寻找它后另一函数的序码来验证。如果你看不到序码，那么要么是编译器把另一个函数的序码优化到其他地方了，要么就是当前函数有多个出口。

函数返回值

当函数返回一个值时，它把值送到一个寄存器或一组寄存器中。要确定例行程序的返回值是否正被使用，检查一下调用了该例行程序的代码中寄存器的使用情况，如果你看到一段代码调用了例行程序，并且然后在没有明确设置其值的情况下就使用了存放返回值的寄存器，你就可以知道这个代码使用函数的返回值。例如，如果你看到一段代码调用了一个函数，并且在没给 AX 寄存器设置值的情况下使用了 AX，那么你就可以知道被调函数将把值返回到 AX 寄存器中。

在 32 位代码中，一般情况下函数把值返回到 EAX 中。16 位代码用 AX 存放 16 位的返回值，而 DX:AX 组合起来存放 32 位的返回值。如果代码是用汇编语言写的，那么前面都白说了，因为汇编语言程序员能按其需要来返回值。一个常用的汇编器约定是：如果例行程序仅需返回一个正确或失败代码，则例行程序相应地设置或清除进位标志(CF)。你可通过寻找直接跟在 CALL 指令后的 JC 和 JNC 指令来找到这些例行程序。

标识参数

如果你知道你所分解函数的参数，在汇编代码中给这些参数做上标记特别容易。除了一个例外(我将在这节后面阐述它)外，编译器总是在堆栈上传递函数或过

程的参数，通过累加每一个被传递的参数的尺寸，你能很快确定堆栈中每一个参数的地址。然而在举这样的一个例子之前，我首先需要回顾一下在 Windows 和 Win32 中被使用的编译器调用约定。

在 16 位 Windows 代码中，许多引出函数使用 Pascal 调用约定。在 Pascal 调用约定中，调用代码从左到右地把参数压入到堆栈中。作为一个例子，对“foo(0x10，0x20，0x30)”的一个调用而产生的 16 位代码看起来像这样：

```
PUSH 0010h
PUSH 0020h
PUSH 0030h
CALL FAR PTR FOO
```

除了指明参数从左到右进行传递外，Pascal 调用约定也指示被调函数在返回之前必须从堆栈中移走其调用参数。在我刚举的这个例子中，该 foo 函数在返回前需从堆栈中弹出 6 个字节，它可能用一个 RETF 6 指令来完成这个操作。

和 Pascal 调用约定相反的是 C 调用约定。标堆 C/C++运行时间库函数使用 C 调用约定。在 C 调用约定中，参数是从右到左被传递的。（从右到左传递参数的最大好处是支持诸如 printf 一类具有可变数量的参数的函数）。调用 C 风格函数的代码，负责在被调函数返回后从堆栈中移出参数。用 C 调用约定的对“foo(0x10，0x20，0x30)”的调用看起来像这样：

```
PUSH 0030h      ;; Parameters pushed right to left.
PUSH 0020h
PUSH 0010h
CALL FAR PTR FOO
ADD  SP,06h     ;; Remove parameters from the stack.
```

你不能希望在一个 C 风格调用之后总能看到一个“ADD (E)SP,XX”指令。如果编译器只压入了一个或两个参数，它有时从堆栈中把它们弹进一个未用的寄存器中。Borland C++编译器就具有这样的代码生成功能。

对于 Win32，Microsoft 对所有被操作系统 DLL 引出的函数都采用了标准调用约定，这个标准调用约定是 C 和 Pascal 约定的一个混合。调用者从右到左压入参数，如同在 C 风格中一样，而被调函数从堆栈中清除参数，这又像 Pascal 风格一样。偶而当你在 Microsoft 的 C++中使用标准调用函数时，编译器内部地把一个“@xx”加到函数名的尾部。这个 xx 是一个字符串，表示函数期望的参数的字节数，例如：_GetWindowLong@8 或者_PeekMessage@20。

在你清楚了所检验函数的调用约定之后，你能确定这些参数在堆栈中的位置。知道了这些参数相对于堆栈主体的偏移量后，你就可寻找出访问了这些内存单元的指令，并且然后用一个符号名字取代汇编语言地址。在阅读一个汇编语言的列表时，拥有符号名字是极为有用的。

在一个函数执行它的序码之后，堆栈主体看起来像如图所示：

Parameters	
return address	(placed here by the CALL instruction)
previous (E)BP	(pushed by the prologue code)

正如你所看到的，(E)BP 寄存器指向先前的(E)BP 值所存放的位置。在函数内部中，所有的参数现在都能通过 BP 或 EBP 的值间址访问。这点非常重要，值得再说一下：使用了诸如[BP+xx]或[EBP+xx]的地址的内存访问，很可能是正在使用该例程的参数。

像被 16 位 Windows 引出的 API 一类的有 16 位的远函数，它的实际堆栈主体像如下所示：

Parameters	(starting at BP+06)
return CS	(at BP+04)
return IP	(at BP+02)
Previous BP	(at BP+00)

假定每个参数都是一个 WORD 的并且用 Pascal 调用约定，则函数的最后参数将在[BP+06]处，倒数第二个参数在[BP+08]处，以此类推。如果有任何 DWORD 参数，则这个计算需要作相应的调整。还有，如果这个函数是一个近函数，这个给定的位置也需要调整，因为在堆栈中仅仅返回 IP，而不返回 CS。

现在让我们来看一个真正的例子，以便对我们刚才描述的内容有一个更好的认识。一个 16 位程序的窗口过程具有以下描述：

```
LRESULT WINAPI WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
```

在 WndProc 代码中，堆栈像这样：

```
hWnd            WORD PTR  [BP+0E]   ;; Parameters pushed left to right.
msg             WORD PTR  [BP+0C]
wParam          WORD PTR  [BP+0A]
lParam          DWORD PTR [BP+06]
return CS       WORD PTR  [BP+04]
return IP       WORD PTR  [BP+02]
previous BP     WORD PTR  [BP+00]
```

具备了这些知识，你就能使用编辑器检索和代替功能来找到对[BP+0E]的所有访问，并且用更具含义的[hwnd]来取代它们。例如，你可用[msg]代替[BP+0C]，等等。

现在让我们看看对于上述窗口过程等价的 32 位程序。在 Win32 中，所有参数都是 32 位的，返回地址是一个 32 位的近程指针，并且代码使用 EBP 而不是 BP。别忘了该窗口过程使用标准调用约定，使得这些参数是以和 16 位代码相反的顺序

出现的。32 位的窗口过程的堆栈因此看起来像这样：

```
lParam          DWORD PTR [EBP+14]  ;; Parameters pushed right to left.
wParam          DWORD PTR [EBP+10]
msg             DWORD PTR [EBP+0C]
hWnd            DWORD PTR [EBP+08]
return EIP      DWORD PTR [EBP+04]
previous EBP    DWORD PTR [EBP+00]
```

我已经描述了 32 位函数的常用堆栈格式，现在我将告诉你一些坏消息。32 位编译器有一个不产生标准的 EBP 格式的选择项。它们做此事是为了通过不包括建立和记录堆栈的代码来节省时间和空间。

问题是：产生的代码不能用来自于 EBP 的一个偏移量来寻址参数和局部变量，却可用来自于 ESP 的一个偏移量（例如[ESP+14]）来寻址参数和局部变量。如果这给你一个惊醒，是理所当然的。当 ESP 为准备调用其他例程而把参数压栈时，它的值在整个函数中变化。因此在一个函数中，如果代码把一个 DWORD 压栈的话，早先在[ESP+14]处的一个[lParam]则随后可在[ESP+18]处找到。如果代码压入第二个 DWORD，则[lParam]将在[ESP+1C]处。这就使得查找像 MOV EAX,[ESP+14]的内存访问并用像 MOV EAX,[lParam]一样的进行替换，几乎成为不可能的事。因为这个原因，你得在整个函数中寻找 ESP 的相对位置，通过指令基值给参数一个符号名字。现在你大概真的只希望编译器把一个参数拷贝到一个寄存器中，并且在任何需要的地方使用寄存器中值的复制品。

如果你正在分解你不知道其参数的函数，你仍可做点事情，使处理过程变得简易些。例如，你一定最想知道传递给函数的参数的字节数。为了做到这一点，看看函数退出的收尾程序。它是否是用像 RETF 8 一样的指令来弹栈？如果是这样，你就可知道这个函数所带的参数有多少个字节（在这个情形中是 8 个字节）。如果函数的退出代码不从堆栈中移走任何东西，那么在代码中找到调用该函数的地方，在 CALL 指令之后的下一个指令是不是像 ADD ESP,12 类似的指令？如果是，那么这个函数带了 12 字节的参数。

除了知道函数参数的字节数外，你能经常地通过研究为准备调用该函数而把参数压栈的代码来搜集更多的信息。例如，假设你看到了下面的从一个 Win32 程序反汇编的列表片断：

```
CALL    GetFocus
PUSH    EAX
CALL    GetCurrentThread
PUSH    EAX
CALL    DoSomething
```

从这个代码片断中，你能确定这个 DoSomething 函数带两个参数：一个 HWND 和一个线程 HANDLE。我怎样能知道这些呢？GetFocus()和 GetCurrentThread()这两个函数都是在 EAX 中返回一个值的 Win32 API。在调用 GetFocus()之后，EAX 保存一个 HWND 值。在调用 GetCurrentThread()之后，EAX 保存一个线程 HANDLE。根据逻辑推理，DoSomething()函数期望一个 HWND 和一个

HANDLE 作为其参数。

尽管参数通常在栈上传递，但它也可能在寄存器中传递，这个调用约定通常被称为“快速调用”(fastcall)约定，因为在寄存器中传递参数比在栈中传递要快。例如，许多 KRNL386 内部堆管理例程在寄存器中传递参数以提高速度。编译器或汇编语言程序员根据函数基体来决定寄存器参数是否用在该函数上。Microsoft 编译器把用了快速调用约定的函数名字前面加一个“@”符号。在你源代码中的“Foo”函数，如果编译器用了快速调用约定，则它在 MAP 文件或调试信息中就以“@Foo”形式出现。快速调用风格函数不仅仅局限于寄存器参数。编译器能把一些参数传给寄存器而把另一些参数传到栈上。

最后，如果你正检验的代码使用了中断，则取出中断表文件说明并且查询哪些参数在哪些寄存器中。在 INI 指令处加一个注释，描述这个指令要做什么，例如：

```
MOV AX,0500
LES DI,[myBuffer]
INT 31
```

将变成这样的形式：

```
MOV AX,0500
LES DI,[myBuffer]   ; DPMI function 0500h - Get Free Memory Information
INT 31              ; ES:DI -> structure to fill with information
```

标识局部变量

像函数参数一样，一个例程的局部变量也通常在栈中找到。参数和局部变量之间的一个关键区别是：代码从栈中用一个负偏移量来访问局部变量，例如，在 16 位代码中的[BP-04]，或者 32 位代码中的[EBP-04]。

不同于参数，没有半机械化的方法来决定局部变量的类型、用途和地址，你必须检查函数的代码是怎样使用一个特定内存单元的。有时确定一个局部变量的含义是相当容易的，例如，看下面的 Win32 代码片断：

```
PUSH DWORD PTR [EBP+08]
CALL GetParent
MOV  [EBP-0C],EAX
```

这个 GetParent()函数是一个 Win32 API 函数，它用了一个 HWND 参数，并且返回给 EAX 一个窗口的父 HWND。因为这个代码片断把 EAX 拷贝到[EBP-0C]中，因此[EBP-0C]明显是一个 HWND。另外，你能作一个大胆的猜测，认为这个变量可能在源代码中像是“hWndParent”那样被称呼。一旦你到达了这一步，就该是使用编辑器的查找和替换功能把所有的[EBP-0C]替换成[hWndParent]的时候了。在你这样做了之后，看看你的反汇编列表，它开始变得比较清晰了。

也许你要说，“这是一个好的例子，但是并非每一个局部变量都这么容易获取。”的确，但是有不止一种方法来解决这样的问题。有时，从局部变量被用作为到

另一个函数的参数的情况中，来标识这些局部变量要更容易一些。下面这个 Win32 汇编片断是这样的一个例子：

```
LEA  EAX,[EBP-30]       ; Get address of EBP-30h into EAX.
PUSH EAX                ; Push it as an LPRECT.
PUSH [EBP+08]           ; Push an HWND (a parameter).
CALL GetWindowRect      ; Call into USER32 to get the RECT coordinates.
```

在 SDK 文献中查看 GetWindowRect()，我们知道它使用一个 HWND 和一个 RECT 结构的指针。由于 GetWindowRect 是一个标准调用函数，那么 RECT 指针应先被压栈，接着是 HWND 压栈。在这个列表中我们看到，代码为 LPRECT 参数把一个低于 EBP 值 30h 字节的地址压入栈中，因此在[EBP-30]处必然有一个 RECT 类型的局部变量。由于 WINDEF.H 包含了 RECT 结构(4 个 DWORD)的格式，我们能想象出所有的 RECT 域在栈中的位置：

```
RECT.left   = [EBP-30]
RECT.top    = [EBP-2C]
RECT.right  = [EBP-28]
RECT.bottom = [EBP-24]
```

和前面一样，现在可利用这个机会查找[EBP-xx]，并用更具含义的符号名字替换它们。编译器能暂时地将局部变量(和参数)拷贝到寄存器中，代码在需要变量值的地方使用寄存器。这节省了代码空间和时间。当你处理一个反汇编列表时，你必须找到代码开始使用寄存器变量的地方。在后面用到了该寄存器的任何地方，都替换成你已经想象出的变量名。然而需注意的是：编译器(或汇编语言程序员)可能在函数内部不同的地方对不同的变量使用相同的寄存器。

在 16 位程序中，SI 和 DI 变量是最常被用作寄存器变量的。由于寄存器只有 16 位长，因此它们通常不被用作指针，因为在 16 位 Windows 代码中的大多数指针是 32 位远指针。为此 SI 和 DI 典型地被用于 16 位值，诸如 HWND 和 DC。在 Win32 程序中，ESI、EDI 和 EBX 寄存器是最常用的寄存器变量。在 Win32 中，指针都是 32 位近指针，因此这些寄存器除了作为其它类型的变量外，也常作为指针。然而这些准则中没有一个是绝对可靠和快速的，当处理寄存器变量时，得使用你的直觉和判断力。

标识全局变量

确定一个程序在使用一个全局变量是特别的容易，几乎任何使用了一个硬代码(hardcoded)的地址的内存访问都是一个全局变量。用另一种方式讲，全局变量寻址时不需借助寄存器(如 EBP)。在 32 位代码中，一个全局变量的访问看起来像这样：

```
MOV EAX,[00464398]
```

如果你走运并且有符号信息，那么反汇编器可能已经用在源程序中使用的名字取代了“[00464398]”。如果不是这样，你应找到所有使用了该内存单元的指令，

并且用符号名字来取代地址。如果你没有符号信息，尽力想出这个变量是干什么用的，并为它取一个名字。

在 16 位代码中，标识全局变量与在 32 位代码中极为相似，尽管是具有 16 位地址而不是 32 位地址。然而，如果你正在处理的代码具有多数据段，你也需要格外小心。这个问题是同样的偏移量可能在不同的数据段中用到。当在一个非默认的 DGROUP 段中存取全局变量时，代码建立了一个段寄存器，用它来指向这个段。代码然后在这个段中用硬代码偏移量访问变量 - 例如，MOV AX,ES:[001C]。需要提起的是：当用符号名字替代全局变量地址时，应当小心谨慎。

如果你有一个可执行文件的符号信息，但是碰到一个不在全局变量表中的内存单元，你可能面临两个相似情形之一。在第一种情形中，内存单元可能用于一个静态变量。如果你的符号信息仅包含公共符号，那么变量不会在表中有显示。在第二种情形中，你可能要查看一个结构或数组的成员。例如，一个 16 位程序有一个全局变量“MSG MyMsg;”，它结束于程序的 DGROUP 段中偏移地址 0364h 处。进入 MSG 结构的 4 个字节存于 wParam 域，因此 MyMsg. wParam 将在该数据段中偏移地址 0368h 处。为这个可执行文件产生的符号信息将在 0364h 偏移地址处含一个被称为“MyMsg”的公共符号，但是在 0368h 偏移地址处却什么也不包含。

为了说明这个过程，假设当我检查一个反汇编列表时，你也在一起观看。当我工作时，我遇到一个读出在 0368h 偏移地址处的值的指令。令我懊丧的是，符号信息并不显示这个偏移地址处的任何符号。

这也并不是没有办法了。通过寻找出现在 0368h 地址之前的最近符号，我看见了在 0364h 偏移地址处有被称为“MyMsg”的东西。建立在一个名字和预感基础之上，我假设在 0364h 偏移地址上的 MyMsg 符号是一个 MSG 结构。我然后需要验证这个理论。如果 0364h 偏移地址处真的是一个 MSG 结构，那么 0368h 偏移量是结构的一个域的地址吗？在这种情形中，是的。

然而，在我假设我已正确地猜测了之前，我将寻找违背该理论的其它的代码。这个 0368h 内存单元像是作为 WPARAM 使用的吗？下一个结构域看起来像作为一个 LPARAM 使用的吗？不幸的是，这儿我没有硬的和快的技术来使用，我不得不对要发生的作合理的猜想，并且测试这些猜想，直到我对于我的理论完全有把握为止。

全局变量一个好的方面是编译器很少把它们放在寄存器变量中。使全局变量寄存器化通常不是一个好的主意。如果只是把全局变量正确的拷贝放到一个寄存器中，中断服务程序和回调函数可能失败，如果它们试图使用全局变量的内存版本的话。

标识文字字符串

许多 API 函数把串作为参数。通过用使用它们的函数来匹配 ASC Ⅱ 码字符串，你常能得到关于代码干什么用的一个更好的主意。例如，在 16 位程序中你或许遇到下面的指令序列：

```
PUSH DS
PUSH 0437
CALL GETMODULEHANDLE
```

或在32位程序中，像下面这样的：

```
PUSH 00471784
CALL GETMODULEHANDLE
```

打开你可信赖的API文献，你会看到GetModuleHandle()有一个参数，即指向一个字符串的一个指针。PUSH指令把该串的地址压入栈中，作为给GetModuleHandle()的参数。因此，在00471784(或者16位类型下的DS:0437)地址上，一定有一个以null作为结束的串(例如“USER32”)。如果你的反汇编器为这个文件的数据节做了一次16进制/ASCII转储，则到这个地址上并检索这个串。退回到代码中访问该串的地方，做一个包含被检索串的注释。例如：

```
PUSH 00471784           ;; "USER32"
CALL GETMODULEHANDLE
```

如果你正在反汇编的代码使用了大量的文字字符串，在做完这些后，你将因代码变得多么的清楚而惊叹。填充文字字符串是反汇编的一个技巧和比较耗时的一个方面。

一些可执行代码在代码段包含了文字字符串。通常字符串在内存中紧跟在访问该串的代码后面。

一个好的反汇编器能够识别这种情形，并能临时地转换成十六进制转储格式。然而，反汇编器也常犯错误。有时，你需要检查周围代码，以便寻找可以告诉你代码在什么地方开始和数据在什么地方结束的线索。通常像开关语句JMP表一类的嵌入数据在你的反汇编列表中产生临时的“垃圾”。通过查看周围代码，你能获得在代码区域内哪些是真正的代码，哪些是嵌入数据的线索。你然后可以把这些信息反馈给反汇编器，并做一个可正确区分代码和数据的第二个列表。不过没有人说这样做起来很简单！

标识if语句

条件执行代码最简单类型应该是一个简单的if语句：

```
if ( some test ) {
    do some sequence of code
}
```

在讨论这种语句的变化前，看一下它在汇编语言中看起来像什么。从反汇编列表层次来看，有三种你能遇到的主要测试类型：

- 相等测试：if(a==b)，if(a != b)，等等……

• 布尔 TRUE/FALSE 测试:if(a),if(! a),等等……

• 位域测试:if(a & 0x0040),等等……

尽管编译器为各种测试类型产生不同的代码序列,但每种情形最终目的还是设置或清除 CPU 的零标志(ZF)。在设置或清除零标志(ZF)后,代码使用 JZ(零跳转)或 JNZ(非零跳转)条件转移指令去执行或越过代码的下一部分。尽管显得有些混乱,但 JZ 指令助记符可以被表示为 JZ(相等跳转),JNZ 也可写成 JNE(不等跳转)。

"测试,条件地转移"的基本算法模式是:如果测试表达式结果为 FALSE,CPU 执行条件转移,后面在{}或 BEGIN/END 块内的代码不被执行。如果检测结果为 TRUE,则条件转移不执行,接着执行{}块内的代码。

警告:刚才叙述的仅仅是一个简单情形。实际上,产生的代码要复杂得多。例如在 16 位代码中,或许有这样的一个 JZ 或 JNZ 指令:它唯一功能是跳过一个 JMP 语句。如果 if 块中的代码要比 127 字节长,则会发生这样的事情,因为在 16 位代码中条件转移指令限制在 127 字节内。当然,我刚叙述过的基本前提仍适用。

关于相等测试,编译器使用 CMP 指令。用"DUMPBIN/DISASM"所产生的输出片断显示了一个例子:

```
0000101E:  cmp  dword ptr [ebp-04],04
00001022:  jne  0000102E
00001028:  inc  byte ptr [ebp-04]
0000102B:  inc  byte ptr [ebp-08]
0000102E:  ...
```

第一条指令把在[EBP-04]处的 DWORD 与 4 比较,如果相同,CMP 指令设置零标志;否则清除零标志。下条指令(JNE)仅当零标志被清除时,跳过其后的代码。因此,两个 INC 指令仅当零标志被设置时才执行。而零标志仅当[EBP-04]与 4 相等时被设置。用 C 代码表达,上面片断类似于以下片断:

```
if ( SomeVariable1 == 4 )
{
    SomeVariable1++;    // INC [EBP-04]
    SomeVariable2++;    // INC [EBP-08]
}
```

当 if 语句中表达式只关心表达式是 TRUE 还是 FALSE 时,编译器有一个代码生成选项的选择。在某些情形下,生成的代码像前面描述的 if 语句代码那样,例如表达式 if(MyVariable)也可写成"if(MyVariable ! = 0)"。另一些要考虑的情形是当表达式的值在寄存器中时。这种情形下,编译器可用一个短指令来确定值是 TRUE(非零)还是 FALSE(零)。更短的指令是"OR register,register"指令,如下所示:

```
0000102E:  call  00001000
00001033:  or    eax,eax
00001035:  je    0000103E
0000103B:  inc   byte ptr [ebp-04]
0000103E:  ...
```

在这段代码中，第一条指令调用一个返回值在EAX中的函数。编译器没有使用三字节的“CMP EAX，0”指令，而是使用了一个OR指令。OR指令在EAX的所有位上做了一次“或”逻辑运算。仅仅当其他位没被设置时，零标志才被设置（因此EAX == 0）。

你将看到同if语句有联系的第三个指令序列，是在各个单独位被使用时出现的。Windows编程中的许多WORD和DWORD是由一组1位标志组成的，例如以一个DWORD传给CrerteWindow()的WS_xxx风格位组。需要检查某位是否被置位的代码，使用逐位的AND操作符（C语言中“&”操作符）。考虑下面的C程序段：

```
DWORD winFlags = GetWinFlags();
if ( winFlags & WF_CPU386 )
    is386 = TRUE;
```

这个if语句产生的汇编代码如下所示：

```
0000102E:  test  byte ptr [ebp-08],04   ;; WF_CPU386 == 0004h
00001032:  je    0000103F
00001038:  mov   dword ptr [ebp-0C],00000001
0000103F:  sub   eax,eax
```

第一条指令使用CPU的TEST（测试）指令来检查表示4的那一位是否设置了。TEST指令在两个操作数间执行一次相应位的逻辑AND运算，但不改写任何一个操作数。如果结果每一位都未被设置，则TEST设置Zero（零）标志。若某一位被设置，则TEST指令清除Zero标志。若[EBP-08]中表示4的位被设置，则Zero标志将被清除。作为结果，JE指令不能跳转，在[EBP-0C]处的DWORD将被加1。

如果你仔细看一下前面代码段的TEST指令，你将注意到一些奇怪的东西。C语言代码中，“winFlags”是一个DWORD，然而生成汇编代码看起来是位于最底层的BYTE（字节）。编译器进行了优化，并尽可能使用最短指令。在不优化情形下，用TEST来测试一个完整DWORD要用3个以上字节，如下所示：

```
TEST DWORD PTR [EBP-08],00000004
```

并不是因为优化我才提及这一点，而是因为你需要认真看看TEST指令。地址和被TEST测试的位可能和你想的不一样。例如，在前面的例子中，让我们改变测试去寻找值为00000400h的WF_80x87。TEST指令因此应该变成“test [ebp-08],00000400h”，对吗？错了！编译器生成的实际指令是“test [ebp-07],04”。该代码在内存中比winFlags DWORD（在EBP-08处）的开始要高一个字节。为了补偿，编译器把要测试的位右移8位。甚至被测试位进入到变量中，编译器也能找到地址并相应地移动测试位。

我们已经看到简单if语句的三种编码方法，现在让我们再进一步。比一个if语句稍微复杂一点的是if-else语句。考虑如下例子：

```
if ( i == 4 )
{
    i++;
    j++;
}
else
    j-;
```

编译器生成下面代码：

```
0000101E:  cmp  dword ptr [ebp-04],04
00001022:  jne  00001033
00001028:  inc  byte ptr [ebp-04]
0000102B:  inc  byte ptr [ebp-08]
0000102E:  jmp  00001036
00001033:  dec  byte ptr [ebp-08]
00001036:  ...
```

头两个指令看起来和一个简单 if 语句是相同的。后面的 JMP 指令是关键。当执行了表达式是“真”时的代码后，JMP 指令跳过由 else 子句生成的代码。JMP 指令的目的地址是 else 子句的代码在什么地方结束的重要线索。

当你试图识别 if-else 语句时，有两个重要的事情要弄清：原始的 JE/JNE 指令是否是跳到紧跟在一个 JMP 指令后的指令？JMP 指令是否转到一个更高的地址上(即在内存向前而不是向后)？

另一个更复杂的 if 语句形式是含多条件的表达式。例如：

```
if ( (i == 4) && (j == 2) && (k == 6) )
{
    i++;
    j++;
}
```

编译器生成下列代码：

```
0000101E:  cmp  dword ptr [ebp-08],04
00001022:  jne  00001042 ;; Jump past code inside {}'s.
00001028:  cmp  dword ptr [ebp-0C],02
0000102C:  jne  00001042 ;; Jump past code inside {}'s.
00001032:  cmp  dword ptr [ebp-04],06
00001036:  jne  00001042 ;; Jump past code inside {}'s.
0000103C:  inc  byte ptr [ebp-08]
0000103F:  inc  byte ptr [ebp-0C]
00001042:  ...
```

这里的代码是直接向前的，其中有三个成功的测试。若有一个失败，代码将跳过其余的测试和{}中的代码。若你看到都跳到同一个地点的一系列测试和分枝组合，你或许正在处理一个具有多条件的 if 语句，并且每个条件都必须为“真”。

OR 情形中有多项测试，只要求有一项为“真”，它的生成代码类似于 AND 情形的生成代码。你将看到一连串连续的测试和分枝。如果测试结果为“真”，除了最后一个外的所有测试都跳到{}内的代码中。如果一个测试失败，代码将简单地进入下一个测试。若最后一个测试结果为“真”，它将进入{}内的代码中，若测试失败，它跳过{}中的代码。

这节仅涉及到了基本的东西；例如，我没有讨论“for 循环”或“while 循环”。你一定会遇到更复杂的东西。然而不管怎么说，几乎你碰到的所有事都可分解为我在这描述的代码序列的组合和变种。

标识开关语句

在有像 MFC 和 OWL 一样的类库以前，大多数 Windows 程序在它们的窗口过程开始处有一个大的开关语句。该开关语句把各种窗口信息给相应的处理代码。如果你需要看一个程序的窗口过程用一个特定的消息来做什么，你必须应该知道怎样分析一个开关语句。幸好，这不难做到。

分析一个开关语句的一般过程是这样的：对于每个条件跳转，都跳到目标代码所在的地方。直接在代码前加注释来标记该代码处理的情形，这对于经常出现在窗口过程中的开关语句的译码特别有帮助。对代码检查的每个消息，把相应的 WM_xxx 消息名字放在处理它的代码段前面。例如：

```
; CASE WM_NCHITTEST
00413254:   XOR     EAX,EAX
00413256:   JMP     00413454

; CASE WM_GETTEXTLENGTH
0041325B:   MOV     EAX,[cbTextBuffer]
00413260:   JMP     00413454
```

识别一个开关语句是极为容易的，尽管它的编码有三个常用的变种。最简单的开关语句译码是我们所称的“傻瓜编码”，它是非常容易跟踪的，但在过程中浪费了大量的空间。其汇编代码看起来是像这样的一些东西：

```
MOV EAX,[EBP+0C]
CMP EAX,00000045
JE  someAddress
CMP EAX,00000169
JE  someAddress2
CMP EAX,00000265
JE  someAddress3
```

第一条指令把开关语句的结果变量装入寄存器中。在这个例子中，寄存器是 EAX，但是它也可是其他一些寄存器，例如 EDI。但 16 位代码似乎总是用 AX。

在把测试值装入寄存器后，代码进入一连串的 CMP/JE 组合。对于开关语句的每一个“情况(case)”子句，有一个相应的 CMP/JE 组合。作为结果，就很容易找到一个给定开关输入值下的操纵代码。如果一个程序在窗口过程中用一个开关语句来分发消息，则正好查看一下感兴趣的窗口消息的 WM_xxx 值。然后再找到测

试该值的 CMP 指令，这是很简单的。接下来的 JE 指令有处理这个消息的代码的地址。如果你想分解整个例程，以便看看它是怎样处理每个消息的话，找到每个消息的处理代码，并用消息名字来标记它，是很有帮助的。

开关语句的第二个变种与我们刚才描述的紧密相关。不同之处是检测指令使用很少字节，并需要你保持对中间值的跟踪。看下面代码序列：

```
MOV EAX,[EBP+0C]
SUB EAX,2
JE  someAddress
DEC EAX
JE  someAddress2
DEC EAX
JE  someAddress3
SUB EAX,5
JE  someAddress4
```

猛一看，代码比较混乱。它不像第一个开关语句变种那样比较每一个值。唯一真正的行动是使 EAX 的值一直减少。为了弄清代码的含义，你必须知道如果 DEC 指令和 SUB 指令的操作结果为 0，则它们设置 Zero(零)标志。每个 SUB 和 DEC 指令都减掉输入值的一部分，如果值恰好被减到 0，则表示要处理的正是它，并且一个 JE 指令把它发送到相应的处理代码中。较低的初始值将较早被传送，而较高的初始值较晚被传送。

要在给定的 JE 指令处看被测试的是什么值，你需要加上前面所有被减去的值。在分析这类开关语句时，我发现用当前运行总和来标记 JE 指令，是大有帮助的。下面就是我给上面序列作的注释：

```
MOV EAX,[EBP+0C]    ; Load EAX with the switch() argument.
SUB EAX,2
JE  someAddress     ; 2 (Jumps only if EAX was initially 2.)
DEC EAX
JE  someAddress2    ; 3 (Jumps only if EAX was initially 3.)
DEC EAX
JE  someAddress3    ; 4 (Jumps only if EAX was initially 4.)
SUB EAX,5
JE  someAddress4    ; 9 (Jumps only if EAX was initially 9.)
```

你将遇到的第三种开关语句编码类型叫做“跳转表(jump table)”。如果一系列输入值足够相近的话，编译器可决定建立一个地址数组。数组的每一项对应一种情形值。跳转表的好处是非常快。代码不必包括对每一种可能的情形值的测试。下面的 C 代码就显示了一个编译器能使用跳转表的开关语句：

```
switch ( i )
{
    case 0x0:   i = 2; break;
    case 0x1:   j = 2; break;
    case 0x2:   k = 3; break;
```

```
    // Cases 3 through 8 not shown.
    case 0x9:   j = j + k + i; break;
}
```

编译器生成如下的代码：

```
00001008:  mov  eax,dword ptr [ebp-0C]
0000100B:  cmp  eax,09
0000100E:  ja  00001068
00001010:  jmp  dword ptr [eax*4+0040108F]
```

第一条指令把开关语句的输入值装入 EAX。接着两条指令确定输入值是否在情形值列表范围之内。若不在，JA 指令跳到开关语句之后的代码。最后的指令使用 EAX 作为处理代码地址数组的索引，并跳到那个位置上。

在前面的代码中，编译器把处理代码地址数组放在数据区中。然而，如果该数组直接跟在 JMP 指令之后，也不必感到惊奇。这在 16 位程序中特别普遍。说出这个情况在什么时候出现是很容易的，因为 JMP 指令把 CS 用作为内存地址的一个部分。若处理代码数组跟在 JMP 语句之后，一会儿你可以看到一些“垃圾”指令。这是因为反汇编器并不知道这些字节是数据而不是代码。一个好的反汇编器应该能区分这种形式，或者至少能让你告诉它在代码区中哪些是真正的数据。

一个反汇编例子

现在我已经谈及了反汇编器的基础东西，让我们举一个真实的例子来看看这些概念是怎样应用的。对于这个例子，我用了 Windows NT CLOCK.EXE 的一个例程，它让程序在带标题栏和不带标题栏之间进行转换。我选择这个函数有两个原因。首先，由于我在前面的侦探程序分析过程中已经检查了这个例程，我们可以通过对这两个方法下所得的结果进行比较，来做一细致检查。其次，CLOCK 的源程序随 Microsoft 为 Win32 编程所提供的样板一起被提供，因此，你将能够判断反汇编过程的准确性。

对于这个例子，我将用我自己反汇编器的输出结果。我当然也可简单地用 Microsoft 的 DUMPBIN 程序。然而，一些用 DUMPBIN 程序必须手工来做的事，我的反汇编器可自动完成，特别是用符号名来匹配一个对 API 函数的调用时。下面是对正在讨论的这个例程的反汇编器原始输出：

```
12F3B00:    PUSH    ESI
12F3B01:    PUSH    EDI
12F3B02:    MOV     ESI,DWORD PTR [ESP+0C]
12F3B06:    PUSH    F0
12F3B08:    PUSH    ESI
12F3B09:    CALL    GetWindowLongA
12F3B0E:    MOV     EDI,EAX
12F3B10:    CMP     DWORD PTR [012F612C],00
12F3B17:    JE      012F3B30
```

```
12F3B19:    AND     EDI,FFB4FFFF
12F3B1F:    PUSH    00
12F3B21:    PUSH    F4
12F3B23:    PUSH    ESI
12F3B24:    CALL    SetWindowLongA
12F3B29:    MOV     [012F6000],EAX
12F3B2E:    JMP     012F3B44
12F3B30:    OR      EDI,00CF0000
12F3B36:    MOV     EAX,[012F6000]
12F3B3B:    PUSH    EAX
12F3B3C:    PUSH    F4
12F3B3E:    PUSH    ESI
12F3B3F:    CALL    SetWindowLongA
12F3B44:    PUSH    EDI
12F3B45:    PUSH    F0
12F3B47:    PUSH    ESI
12F3B48:    CALL    SetWindowLongA
12F3B4D:    PUSH    27
12F3B4F:    PUSH    00
12F3B51:    PUSH    00
12F3B53:    PUSH    00
12F3B55:    PUSH    00
12F3B57:    PUSH    00
12F3B59:    PUSH    ESI
12F3B5A:    CALL    SetWindowPos
12F3B5F:    PUSH    05
12F3B61:    PUSH    ESI
12F3B62:    CALL    ShowWindow
12F3B67:    POP     EDI
12F3B68:    POP     ESI
12F3B69:    RET     0004
```

开始两行和结尾三行是作为序码和收尾程序的，这很容易识别。除了下面两点外，它们并没有什么令人感兴趣的地方："RET 0004"说明此函数带有一个参数(32位代码中的所有参数均为4个字节)。第二，此代码并设有建立EBP堆栈结构，因此我们必须要保持跟踪堆栈以决定参数所在的位置。

幸运的是，整个例程仅有一条指令访问栈上的参数，那就是序码后的"MOV ESI,DWORD PTR [ESP+0C]"。这条指令把一个参数拷入ESI寄存器中，而该寄存器在整个函数的好几处地方出现，这使得ESI类似某种寄有器变量。嗯，ESI还会有什么别的含义呢？查看整个例程，我们看到ESI作为一个参数被传递给GetWindowLong()、SetWindowLong()、SetWindowPos()和ShowWindow()函数。ESI有可能保存一个HWND吗？看来它确实如此！

让我们用这个机会，利用已经发现的东西，重写前面的列表，把这些指令变为易于理解的序列，并且去掉序码和收尾。

```
12F3B02:    MOV     hWnd(ESI),DWORD PTR [ESP+0C]

12F3B06:    PUSH    F0
```

```
12F3B08:    PUSH    hWnd(ESI)
12F3B09:    CALL    GetWindowLongA
12F3B0E:    MOV     EDI,EAX

12F3B10:    CMP     DWORD PTR [012F612C],00
12F3B17:    JE      012F3B30

12F3B19:    AND     EDI,FFB4FFFF
12F3B1F:    PUSH    00
12F3B21:    PUSH    F4
12F3B23:    PUSH    hWnd(ESI)
12F3B24:    CALL    SetWindowLongA
12F3B29:    MOV     [012F6000],EAX
12F3B2E:    JMP     012F3B44

12F3B30:    OR      EDI,00CF0000
12F3B36:    MOV     EAX,[012F6000]
12F3B3B:    PUSH    EAX
12F3B3C:    PUSH    F4
12F3B3E:    PUSH    hWnd(ESI)
12F3B3F:    CALL    SetWindowLongA

12F3B44:    PUSH    EDI
12F3B45:    PUSH    F0
12F3B47:    PUSH    hWnd(ESI)
12F3B48:    CALL    SetWindowLongA

12F3B4D:    PUSH    27
12F3B4F:    PUSH    00
12F3B51:    PUSH    00
12F3B53:    PUSH    00
12F3B55:    PUSH    00
12F3B57:    PUSH    00
12F3B59:    PUSH    hWnd(ESI)
12F3B5A:    CALL    SetWindowPos

12F3B5F:    PUSH    05
12F3B61:    PUSH    hWnd(ESI)
12F3B62:    CALL    ShowWindow
```

现在，几个函数调用，特别是 GetWindowLong()、第一个 SetWindowLong()、SetWindowPos()和 ShowWindow()，可很快转变为 C 等价语句。给这些例程的每个参数或者是我们已经发现的 hWnd，或者是我们能在 WINDOWS.H 中查到的数值。让我们再次重写此例程，把其中一些指令序列压缩成单个 C 语句：

```
12F3B02:    MOV     hWnd(ESI),DWORD PTR [ESP+0C]

GetWindowLong( hWnd, GWL_STYLE );   // GWL_STYLE == -16 == 0F0h
12F3B0E:    MOV     EDI,EAX

12F3B10:    CMP     DWORD PTR [012F612C],00
12F3B17:    JE      012F3B30
```

```
12F3B19:    AND     EDI,FFB4FFFF

SetWindowLong( hWnd, GWL_ID, 0 );   // GWL_ID == -12 == 0F4h
12F3B29:    MOV     [012F6000],EAX

12F3B2E:    JMP     012F3B44

12F3B30:    OR      EDI,00CF0000
12F3B36:    MOV     EAX,[012F6000]
12F3B3B:    PUSH    EAX
12F3B3C:    PUSH    F4
12F3B3E:    PUSH    hWnd(ESI)
12F3B3F:    CALL    SetWindowLongA

12F3B44:    PUSH    EDI
12F3B45:    PUSH    F0
12F3B47:    PUSH    hWnd(ESI)
12F3B48:    CALL    SetWindowLongA

SetWindowPos( hWnd,0,0,0,0,0,   // 0x27 == the flags on the next line
            SWP_NOSIZE | SWP_NOMOVE | SWP_NOZORDER | SWP_FRAMECHANGED);

ShowWindow( hWnd, SW_SHOW );
```

在我们能用 C 等价语句重写一些函数调用时，我们还不能足够了解给最后两个 SetWindowLong()的参数并压缩它们。一方面我们不知道 EDI 包含什么，另一方面，我们需要知道在地址[012F6000]处的全局变量是什么。

先等一下！我们已经看到函数 GetWindowLong()检索窗口的风格值，并把它拷入 EDI 中。EDI 可能是用于保存窗口风格位组的另一个寄存器变量。至于全局变量 [012F6000]，注意代码把函数 SetWindowLong(GWL_ID)的返回值保存在它里面。前面我已经描述过了窗口 ID 域是怎样被用来保存高层窗口的 HMENU 的。结合 SetWindowLong()返回这个域先前的值这一事实，你可以猜到[012F6000]是一个含有一个菜单句柄(HMENU)的全局变量。

下面让我们利用这两个新发现(在 EDI 中的窗口风格变量和 HMENU 全局变量)再重新写此例程：

```
winStyle = GetWindowLong( hWnd, GWL_STYLE );

12F3B10:    CMP     DWORD PTR [012F612C],00
12F3B17:    JE      012F3B30

winStyle &= ~(WS_DLGFRAME | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX);

HMenu = SetWindowLong( hWnd, GWL_ID, 0 );

12F3B2E:    JMP     012F3B44
```

```
12F3B30:
winStyle |= (WS_BORDER | WS_DLGFRAME | WS_SYSMENU |
             WS_THICKFRAME|WS_MINIMIZEBOX|WS_MAXIMIZEBOX);
SetWindowLong( hWnd, GWL_ID, HMenu );

12F3B44:
SetWindowLong(hWnd, GWL_STYLE, winStyle);

SetWindowPos( hWnd,0,0,0,0,0,
            SWP_NOSIZE | SWP_NOMOVE | SWP_NOZORDER | SWP_FRAMECHANGED);

ShowWindow( hWnd, SW_SHOW );
```

这样，留给我们的只是在地址 012F3B10 处的条件分枝语句。CMP 指令把地址[012F612C]处的全局变量与 0 相比较。一个 JMP 指令直接指出了条件转跳的目的地。这看来与我前面所述的 if-else 语句相似。在[012F612C]处的全局变量看起来像布尔型。让我们给它取个名字(例如："MyBool")，写在{}中并呈锯齿状，并且看看这样的代码是否好读了：

```
winStyle = GetWindowLong( hWnd, GWL_STYLE );

if ( MyBool != 0 )
{
    // Turn off the style bits need for the titlebar, boxes, and menu.
    winStyle &= ~(WS_DLGFRAME | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX);
    // Set the window's HMENU field to 0.
    HMenu = SetWindowLong( hWnd, GWL_ID, 0 );
}
else
{
    // Turn on the style bits needed for a titlebar, boxes, and menu.
    winStyle |= (WS_BORDER | WS_DLGFRAME | WS_SYSMENU |
                 WS_THICKFRAME|WS_MINIMIZEBOX|WS_MAXIMIZEBOX);
    // Set the window's HMENU field back to whatever it was before.
    SetWindowLong( hWnd, GWL_ID, HMenu );
}

// Blast the style bits into the window.
SetWindowLong(hWnd, GWL_STYLE, winStyle);

// Force Windows to recalculate and repaint what the window should look like.
SetWindowPos( hWnd,0,0,0,0,0,
            SWP_NOSIZE | SWP_NOMOVE | SWP_NOZORDER | SWP_FRAMECHANGED);

ShowWindow( hWnd, SW_SHOW );
```

我们拿到一些原始的汇编代码，并把它返回成可读的 C 代码。如果你把这些代码与从侦探工具获得的相比较，你将会看到它们是完全一致的。然而，反汇编列表包含的信息比你通过侦探工具获得的信息要更多。例如，侦探工具不给出有两个全局变量(HMenu 和这个布尔型变量)的任何指示。

对一些人来说，从源汇编代码至 C 代码的这个系列步骤有一点快了。的确，并

非每一反汇编操作都这样顺利或这样快。然而，我希望我已经显示了反汇编一个函数是一个反复过程。当你假定和找出关于此代码一些信息时，你把这些信息反馈进入列表中，希望它将释放其他信息。

关于反汇编最后要注意的是，要毫不犹豫地把要分析的代码装入调试器并独自单步运行它。通过看用了真实值的代码执行情况，常能冲破关于代码目的的思想阻塞。许多次我不能估计出一个函数要返回什么。通过在调试器中单步执行代码，并看见实际的返回值，我便能推出返回值的格式。例如，一个例程总返回一个全局内存句柄。这说明每一丝信息都可能有所帮助。一件很微小的事情能助你“打开”一片代码，你也不必感到惊奇。

高级技巧

在结束本章之前，我认为再谈一些前面题目未包含的通用技巧还是有用的。

使用 SoftIce/Windows

如果你要深入探密，则 SoftIce/Windows 是必备的。在我深入展开之前，我需要声明我在 Nu-Mega 公司工作，SoftIce/Windows 是该公司生产的。可与 SoftIce/Windows 相比的工具只有 Microsoft 的 WDEB386 系统级调试器，但它没有前者那么多的命令来转储数据和使用符号化的调试信息。

SoftIce/Windows 有这样的性能是因为它是一个系统级调试器。与像 Turbo Debugger、CodeView 或在编辑器 IDE 中的调试器等一类的应用调试器不同，SoftIce/W 不依赖 Windows 中任何东西，它在 Windows 和硬件之间运行。正因如此，SoftIce/W 能在系统中单步执行任何代码，包括 ring 0 VxD 和实模式 DOS 代码。这对研究诸如 Windows 调度程序和转换内存上下文(memory context)的例程是很有用的。不要考虑用一个普通调试器来尝试，它们都不行。

公正地说，WDEB386 在这个特殊领域中具有相似性能。Window 95 的一个“圈套”是你不能用应用调试器来单步执行 ring 3 系统 DLL(如 KERNEL32.DLL)。问题是由于 Windows 95 所有进程共享这个代码，因此把 INT 3 加在代码中几乎总与系统相冲突。因为 SoftIce/W 的运行不依赖该系统，所以它没有这些限制，能轻易单步执行任何系统代码。

与普通调试器不同，你不必在 Windows 内装入 SoftIce/W，你也不必非用它调试一段程序不可。相反，SoftIce/W 装在 Windows 层下，并且总是存在的。除非你用它的热键来引出它的用户界面，SoftIce/W 静静地“躲在”Windows 层下面。因此，你能在每次装入 Windows 时都装入 SoftIce/W。当你需要时，就把它弹出。否则，你可以忽略它的存在。它与其说是一段程序，例不如说它更像一个 Windows 的超级版本，你可根据需要停止或启动它，类似于最终的 Windows TSR。

与 WDEB386 不同，SoftIce/W 包括含了大量的用于转储出数据结构和在系统所有级上的列表的命令。在最低层，它能转储出 CPU 的页表以及在全局和局部描

述表中的选择器。往上一层，它能显示与 VXD 相关的重要项，例如 VxD 表、对 VxD 的设备描述块，以及用于维持各进程地址空间的上下文表。事实上，SoftIce/W 甚至能把地址上下文转换成任何任意的内存上下文，让你了解系统中任何进程的所有内存。再往上一层，SoftIce/W 能列出所有进程、线程、模块和 Win16 任务，并伴有每项的详细信息。当你的代码用系统提供的句柄和你需要清楚该句柄访问什么时，这是无价值的。在最高层，SoftIce/W 能显示窗口和窗口类的详尽信息。所有这些并不是要赞扬 SoftIce/W，而是为了让你感觉到你手边到底有多少系统信息。

SoftIce/W 有两个功能特别有用，在这里必须简要提起。在单步执行 16 位代码时，你将常常用看来像全局堆句柄的句柄。简单地把那个句柄给 SoftIce/W HEAP 命令，你能立即证明它是否是一个有效的全局堆句柄。如果是一有效的堆句柄，SoftIce/W 将告诉你这句柄目的是什么（谁拥有它？它是代码、数据还是资源？如果是代码，在 NE 文件中，它对应哪个段？）。

在 16 位或 32 位代码中，SoftIce/W U 和 D 命令用起来很方便。对给定的代码地址，你可以把它供给 U 命令，能很快在此地址反汇编。同样，D 命令以各种格式让你查看内存的任何事物。与应用调试器和其它程序不同，这些命令对它们反汇编或检查内存的位置不受限制。

SoftIce/W 超过其它调试器的另一件事是：它能从 16 位和 32 位 EXE/DLL 文件中装载引出函数信息，并且用它作为伪符号表。因此，在 Windows 中无论你停在什么地方，SoftIce/W 都能告诉你你在什么 EXE/DLL/UxD 中，并且常能指出精确的例程。它甚至使用它反汇编器中的信息，以便把像这样的：

```
CALL  BFFB0149
```

替换为：

```
CALL  GetModuleHandleA
```

另外一个巧妙的 SoftIce/W 命令（为 Win32 和 Windows 95 支持而增加的）是 MAP32 命令。用 MAP32，你能很容易确定一个 EXE 文件和它的 DDL（包括系统 DLL）的所有节驻留在内存的位置。SoftIce/W 另外一个方便的特征是 STACK 命令，无论你停在系统中的任何地方，你总能得到显示你如何到达调用栈的。而且，在任何给定时刻，如果你需要知道当前任务或线程是什么，试用 TASK 和 THREAD 命令，这些命令让你很快知道你正在执行的进程和线程。

应用硬件断点

如果在某一条件是“真”时你需要单步执行代码，你常可用 CPU 的硬件断点来快速到达代码中那一点。例如，我的一个合作者想要知道什么时候某一很少用的线程被激活，通过用 SoftIce/W 的 THREAD 命令，我们能发现我们感兴趣线程的线程 ID。我们可以在线程转换后执行的代码上设一断点，但这种方法证明是不行的，

因为系统不断转换线程。断点不断地“跑开”，但当前线程绝不是我们所感兴趣的线程。

围绕这个问题，我发现了系统保存当前线程 ID 的内存中的这个 DWORD(在未汇编的 GetCurreutThread 函数处)，我于是在保存线程 ID 的这个 DWORD 上设一条件硬件断点。条件是：只有当我们感兴趣的线程 ID 被写入线程 ID 的 DWORD 时，断点才触发。问题就解决了。在我们采取行动把感兴趣的线程“唤醒”之前，系统运行正常。

另一个例子是：我的一个程序正用 SetThreadContext 来改变另一个程序的 EIP。SetThreadContext 报告成功，而另一进程却“爆炸”了。为了查明发生了什么，我在线程上下文结构中新 BIP 值要被写到的 DWORD 上设一硬件断点。通过运行程序，我发现 SetThreadContext 的确把 EIP 值拷到正确位置上。不幸的是，断点在一会儿后又“跑开”了，我想可能是 KERNEL32. DLL 用一个“垃圾”数据改写了我的 EIP 值。如果没有使用硬件断点，我可能仍在猜想是我的代码出错了呢，还是 Windows95 中有错误。

VxD .(园点)命令

WDEB386 和 SoftIce/W 用户通过运行“.”(园点)命令，可得到相当多的对他们有用的系统信息。园点命令(这样称呼是因为它们都用“.”开头)在各种 VxDs 中都能执行。要使用它们，应进入你的系统调试器，在提示下，输入命令名(总以“.”开头)。一些命令所有时候都有效，而其它一些命令只在调试版本中才有效。可以一试的一些命令有：

. . ?	. vtd
. m?	. dosmgr
. vmm	. vmpoll
. vxdldr	. vtdapi
. vpicd	

VAR2MAP 工具

在 Windows95 中，Win32 系统 DLL，诸如 USER32、KERNEL32 等，是“基准的”(based)的。也就是说每当你启动 Windows95 时，它们总装在相同的线性地址上。你能利用这一事实，把难以获得的关于函数和变量的定位地址的知识加到 WDEB386 或 SoftIce/W 中。这样在你单步执行系统代码时，就允许你使用这些符号。怎么样？看一下在 KERNEL32. DLL 中 GetSystemDefaultLangID 函数的代码：

```
GetSystemDefaultLangID proc
BFFB69FD:       MOV     AX,[BFFD44D0]
BFFB6A03:       RET
```

显然地址 BFFD44D0h 保存了一个叫作 SystemDefaultLangID(或具有那种效果的名字)的全局变量。因为 KERNEL32.DLL 在线性地址空间中有一唯一的基地址,SystemDefaultLangID 变量将总在地址 BFFD44D0h 处。如果你能使你的系统调试器了解这一事实,并且在它反汇编过程中用"SystemDefaultLangID"自动替换地址"BFFD44D0h",这难道不是很好吗?我也这样认为,因此我写了 VAR2MAP 程序。

为了使用 VAR2MAP,你应建立包括 32 位地址表和它们相关的名字(你必须给出名字)的文件,该文件包含变量名和函数名。唯一限制是所有名字和地址必须在同一 EXE 或 DLL 文件中。VAR2MAP 把你建立的文件作为输入,并输出一 MAP 文件。MAP 文件有什么好处呢? 你可以通过像 Microsoft 的 MAPSYM 或 Nu-Mega 的 MSYM 一样的程序来运行.MAP 文件,这样的每个程序都能从.MAP 文件产生-个.SYM 文件。WDEB386 和 SoftIce/W 都知道如何装 SYM 文件以便使用符号化反汇编。我在这本书的整个写作过程中,用了 VAR2MAP 来给 KERNEL.DLL 和其他系统模块中的函数确定具有一定含义的符号名。

VAR2MAP 的一个典型输入文件在下面一段代码中显示。文件第一行必须包含到含有这些地址的 EXE 或 DLL 文件的路径。为什么是必须的? 如果你看一下.MAP 文件,你会看到所有公共符号地址都以逻辑地址形式给出,即:ObjectNumber:Offset(例如,0004:00013484)。VAR2MAP 需要 EXE 或 DLL 文件来计算要被映射到内存中的代码和数据节所在的位置。这就允许 VAR2MAP 把你给的线性地址转换成像 0004:00013484 一样的逻辑地址。

```
FILE = C:\WINDOWS\SYSTEM\KERNEL32.DLL
IGetProcAddress = BFF81DC1
IGlobalHandle = BFF76E78
ILocalReAlloc = BFF833C8
ILocalSize = BFF890CB
ppCurrentThread = BFFCB3D4
ppCurrentProcess = BFFCB3D8
ppCurrentTDBX = BFFCB3DC
pWin16Mutex = BFFD34D0
pK16SysVar = BFFD33A4
pKrn32Mutex = BFFCB3FC
```

余下部分应是这样形式:

```
SymbolName = AddressInHex
```

创建输入文件后,运行 VAR2MAP 并通过命令行把输入文件名传给它(例如 VAR2MAP KERNEL32.VAR)。输出的.MAP 文件将放在和文件名在输入文件的"FILE="行的文件名相同的目录。这可以理解。因为你创建的.SMY 文件必须与它相对应的 EXE 或 DLL 在同一目录,否则,调试器将不知道装入.SYM 文件。有关装载.SYM 文件的信息请参见你的调试器文献。

识别 VxD 服务程序

了解对 VxD 服务程序的调用是怎样产生和实现的，对这种探索是有帮助的。一个 VxD 服务程序是一个从 VxD 引出的被别的、VxD 使用的函数。一个 VxD 并不直接调用其它 VxD 中的服务程序(至少不在第一次装入时)。因此，你应用 INT 20h 来从另一个 VxD 中调出一个 VxD 服务程序。看一下 DDK 的 VMM.INC 中的宏 VxDCall，就能知道实际动作了。INT 20h 由 VMM 控制，它把 INT 20h 指令转变成对实际 VxD 服务程序代码的地址的一个调用。

INT 20h 控制者怎样知道你正调出哪个 VxD 服务程序呢？紧跟 INT 20h 的是指明被调出服务程序的一个 DWORD，该 DWORD 的高 WORD 是含被调服务程序的 VxD 的设备 ID，低 WORD 含的是该服务程序在 VxD 的服务程序表中的序号。低 WORD 中的 0 值指该 VxD 中的第一个服务程序，值为 1 指第二个服务程序，依此类推。

在高 WORD 中的设备 ID 要么是在 VMM.INC 中定义的一个标准 VxD ID，要么是一个你公司的 VxD ID(由 Microsoft 赋值)。头 16 个 VxD 设备 ID 列表如下：

VMM_DEVICE_ID	1h	REBOOT_DEVICE_ID	9h
DEBUG_DEVICE_ID	2h	VDD_DEVICE_ID	0Ah
VPICD_DEVICE_ID	3h	VSD_DEVICE_ID	0Bh
VDMAD_DEVICE_ID	4h	VMD_DEVICE_ID	0Ch
VTD_DEVICE_ID	5h	VKD_DEVICE_ID	0Dh
V86MMGR_DEVICE_ID	6h	VCD_DEVICE_ID	0Eh
PAGESWAP_DEVICE_ID	7h	VPD_DEVICE_ID	0Fh
PARITY_DEVICE_ID	8h	BLOCKDEV_DEVICE_ID	10h

如果你正在使用一个标准的 VxD，在 DDK 中通过查看.H 或.INC 文件以寻找该 VxD，你通常能发现一个 VxD 服务程序表。Begin_Service_Table 宏指示服务程序表的开始。那个 VxD 中提供的每一个服务程序有它自己的一行，它用<VxD Name>_Service 作为开始，(例如 VPICD_Service)。在表中的第一个服务程序在 INT 20h DWORD 中的低 WORD 将是一个 0，下一个服务程序的该 WORD 值为 1，依次类推。具备了这些知识，你能轻易计算出跟在 INT 20h 后的 DWORD 值，以便调出一个给定的函数。例如，知道了 VMM VxD 有一个为 1 的设备 ID(在高 WORD 中)，你能轻易计算出每个 VMM 的服务程序的 DWORD，如下所示：

```
Begin_Service_Table VMM,VMM                 00010000h
VMM_Service     Get_VMM_Version             00010001h
VMM_Service     Get_Cur_VM_Handle           00010002h
```

VMM_Service	Test_Cur_VM_Handle	00010003h
VMM_Service	Get_Sys_VM_Handle	00010004h
VMM_Service	Test_Sys_VM_Handle	00010005h

调试器或反汇编器给出由 INT 20h 调出的服务程序的实际名字，并简单地保存在一大表中，该表是一个将 DWORD 值与函数名相匹配的表。前面我谈到过，INT 20h 操纵者把 INT 20h 指令和它后面的 DWORD 拼入到一个 CALL 指令中。严格地讲这并不正确。如果在低 WORD 中的服务程序序号的高位被设置，那么 INT 20h 将被拼入到 JMP 指令。例如，用上表中的 VMM 服务程序，后跟一个 DWORD 为 00018001 的 INT 20h 将变成一个转到 Get_Cur_VM_Handle 函数的 JMP 指令，而不是一个 CALL 指令。为了产生转到一个 VxD 服务程序的 JMP 指令，你应该使用 VMM.INC 中的 VxDJmp 宏，而不是一般的 VxDCall 宏。

识别 Win32 VxD 服务程序

Windows 95 新特征之一是增加了 Win32 VxD 服务程序。这个 Win32 服务程序除了能被 Ring 3 应用代码调用外，类似于 VxD 服务程序。为了调出一个 Win32 服务程序，程序必须调用由 KERNEL32.DLL 引出的 VxDCall0 函数。

VxDCall0 函数的参数之一是一个 DWORD，它与我在前面一节中描述过的 VxD 服务程序 ID 相似。这个 DWORD 中的高 WORD 是设备 ID（像 INT 20h DWORD 一样），低 WORD 是 Win32 服务程序的序号，这个 WORD 对应于 Win32 服务程序表中的该 Win32 服务程序的相对次序，0 表示是第一个服务程序，1 为第 2 个，等等。

为了使这更清楚，让我们看一个例子。下面这段代码来自 KERNEL32.DLL 的 GetThreadContext：

```
BFFABD8D:   PUSH    EAX
BFFABD8E:   PUSH    DWORD PTR [EBX+5C]
BFFABD91:   PUSH    002A0014
BFFABD96:   CALL    VxDCall0
```

前两个 PUSH 指令是 Win32 服务程序的参数，最后一个 PUSH 002A0014 通知 VxDCall0 函数应该调出 2Ah 设备的第 21 个 Win32 服务程序。2Ah 设备是什么？除了 VWIN32 外不会再有别的，它是 KERNEL32 大部分功能的源。

识别参数有效性检查和 Ixxx 函数

在 Windows 3.1 中，Microsoft 引进了检验参数有效性的函数。它们是 API 函数，用来检查你传递过来的参数的有效性。如果你的参数不合标准（例如，指针无效），那么函数不做任何事就返回。在调试版本中，函数也可能发一个警告信息给调试终端。

带有参数有效性检查的函数的完整代码一般存于两个不同的地方，检查参数有效性的代码总出现在函数的开头。假如所有有效性测试都通过了，代码然后JMP(跳转)到实际的代码中。在Windows 3.1中，函数的真正代码有它自己的名子在一个I后跟上函数名。例如，引出函数WinExec检查它的参数后，跳转到一个内部IWinExec函数。在本书中，我对Win32代码也遵循相同的约定。

Windows 95的32位WinExec函数的一个注解列表如下所示。这个代码建立一个结构化的异常处理(SEH)框架，并检验由LpszCmdLine参数指向的字符串中的所有字符能否被存取。如果有效性检查失败，则会出现一个页面错误，SEH框使函数通过在这里没有显示的另一条代码路径返用。如果有效性检查成功，代码去掉SEH框，然后跳转到IWinExec代码上。

```
WinExec proc
BFFB2569:   PUSH    EDI                         ; Preserve EDI.

BFFB256A:   PUSH    22                          ; Set up a structured exception
BFFB256C:   SUB     EDX,EDX                     ; handler frame in case the
BFFB256E:   PUSH    BFFB1172                    ; following validations cause a
BFFB2573:   PUSH    DWORD PTR FS:[EDX]          ; fault.
BFFB2576:   MOV     DWORD PTR FS:[EDX],ESP

BFFB2579:   MOV     EDI,DWORD PTR [ESP+14]      ; Validate the lpszCmdLine
BFFB257D:   SUB     EAX,EAX                     ; parameter by touching every
BFFB257F:   LEA     ECX,[EAX-01]                ; character in the string. A
BFFB2582:   REPNE   SCASB                       ; page fault will trigger the
                                                ; exception handler above.

BFFB2584:   POP     DWORD PTR FS:[EDX]          ; If we got here, everything was
BFFB2587:   ADD     ESP,08                      ; OK. Remove the SEH frame.

BFFB258A:   POP     EDI                         ; Restore EDI.

BFFB258B:   JMP     IWinExec                    ; JMP to the real WinExec code.
```

这里的关键点是识别函数的I版本。当你看到与如上所示类似的代码，且最后用JMP转到别处时，它可能是一个带有参数有效性检查的函数。这个JMP指令的目的地址极可能是代码的I版本的地址。你可以利用这个知识来构造附加符号的地址。例如，在“VAR2MAP工具”一节中，我显示了属此类的四个函数。

IGetProcAddress = BFF81DC1
IGlobalHandle = BFF76E78
ILocalReAlloc = BFF833C8
ILocalSize = BFF890CB

如果你把这些内部名加到你的调试器中，单步执行系统代码会容易得多。参数有效性检查存根显然是一起聚在某个位置上，而函数的真正代码确散布于整个模

块。为内部函数添加附加的符号，给了你一个搞清你在给定系统模块中的位置的机会，这也促使你使用堆栈跟踪。

使用调试版本

除了在你的代码中找错外，Windows 的调试版本可以使你很容易地清楚 Windows 正在做什么。SDK 系统 DLL 的各种调试版本具有有用的诊断跟踪信息。这些字符串通常包含函数名，它们是作为消息部分被发出的。同样，还有许多打印出或注释系统变量值的消息。你常可以在代码中向后退几条指令，以便找出它们访问的变量。例如，考虑下面代码。

```
BFFC788A:   PUSH    DWORD PTR [ESI+18]
BFFC788D:   PUSH    BFFDBF9C ;;     Default Heap: %8x\n
BFFC7892:   CALL    BFFC6092
```

这里 ESI 指向 Win32 进程数据库结构。第二条指令把一个指针传给一个 printf 风格的格式串。利用这个信息，你可以很容易地发现进程的堆句柄位于 18h 偏移处。

Windows 调试版本另一个有用的地方是在所有的正确性检查和声明代码中。调试 DLL 通常检查传给它们函数的参数和系统变量的状态。你可以用正确性检查代码来证实或否定你关于某段代码正在做什么或处理什么的猜测。

Pentium 优化的代码

Microsoft 关于 Windows 95 的声明之一就是它是为 Pentium 而优化的。在这里我告诉你这个声明是确实的。编译器针对 Pentium CPU 做的主要优化是：它把指令序列重新组织以便让 Pentium 的两个执行部件无停歇地同时工作。考虑 KERNEL32.DLL 中的一段代码：

```
1) PUSH     EBP
2) MOV      EBP,ESP
3) SUB      ESP,04
4) CMP      DWORD PTR [EBP+0C],0FFFFF98
5) PUSH     EBX
6) PUSH     ESI
7) PUSH     EDI
8) JBE      BFF741AB
```

第 1、2、3、5、6 和 7 条指令一起组成了序码，它建立了函数的堆栈并且保存寄存器变量的登记。第 4 和第 8 条指令构成了标准 if 语句的指令。如果没做 Pentium 优化，代码看起来要更容易明白一些。JBE 指令将直接出现 CMP 指令之后，而不是在四条指令之后。

我为什么提起这个？当单步执行 Pentium 优化代码（如 Windows 95 系统

DLL)时，指令序列也许不能立即弄清楚，你必须意识到这一点，并去找一下正在做两件不同事情的指令序列。我曾将指令重新分成两组，从而试着保持这样的指令序列。在大多数情况下，这两组指令对应于两个不同的C源代码语句。经过重新安排指令，我先集中精力看第一组，然后再看另一组。

小结

在这一章中，我展示了几种不同的探密方法。开始时，方法的中心是围绕文件转储程序的。在中间是侦探工具，如果你对程序与操作系统之间的相互作用感兴趣的话，它是一个非常有用的工具。在最后，你和你的反汇编器，可以利用一个优化的编译器，来探明一个程序的方方面面。反汇编虽然是一个杂乱无章、不精确和易让人受挫的工作，但它又可能是极有价值的资产，只不过很少有人化时间去学习它。

如果你还没有使用过我所描述的工具和技巧，我希望这个讨论可以除去它们的“神秘面具”，以便使你在需要时愿意去试一试。尽管这些工具做的一些事似乎像“魔术”，实际上它们不是。如果你有硬件和操作系统知识的坚实基础，这些工具和技巧可以作为你工具箱中的一部分，而不是被作为编程“巫术”而保留。

第 十 章

编写 Win32 API 侦探程序

* *

作为程序员，我们经常看到别人的程序，而且想知道它怎样工作。在这种情况下，那种允许你观察一个应用程序在运行中干什么的工具程序是非常有价值的，它能跟踪某个程序或 DLL（动态链接库）在做什么。就现存的对运行中程序进行分析的手段而言，没有什么比一个 API 侦探程序（a spy）更好的了。

API 侦探工具给你报告一个程序及其 DLL 调用了哪些 Windows 函数。除按 API 函数被调用的顺序报告它们的名字外，还记录传递给它们的参数值和它们的返回值。更高级的 API 侦探程序甚至可以记录一些附加的相关信息，如窗口信息，钩式回调函数，以及其他程序事件。有了这些信息，就很容易知道一段代码在做什么。第十章讨论了较为流行的 API 侦探程序，并举了一个例子来说明如何利用 API 侦探程序的输出结果。在本章中，我将建立一个简单实用且易于扩展的 Win32 API 侦探工具。

我在这个 API 侦探程序中用到的拦截 API 函数调用的技术，很容易加以改写而用在你自己的 Win32 程序中。比如，你想用你自己的 lstrcpy 函数来代替标准的 Win32 lstrcpy 函数，你只需在我后面介绍的代码中调用它即可。如果你只是想得到拦截函数调用的程序，可直接翻到本章的最后一部分“在你自己的程序中拦截函数”。如果你还对 Win32 系统级编程和如何使用这项技术感兴趣，请往下读。

为什么在可以买到如 BoundsChecker32 之类功能更强的工具时，还要费力去编写一个简单的 API 侦探程序呢？通过亲自编写 Win32 程序的 API 侦探程序，你可以对 Win32 操作系统的机理有一个完整的理解；而且可以更深入地认识不同的 32 位视窗系统（Windows NT、Windows 95、Win32）之间的重要差别。

表面上，本章的主题是“如何建立一个 API 侦探程序”。然而我的真正目的在于介绍一系列 Win32 编程的实际问题和怎样解决它们。在这个过程中，你将看到 Win32 体系结构的很多侧面。正如你即将认识到，编写一个 Win32 程序的侦探程序，迫使你去面对如地址空间、多线程、动态链接、调试机制、进程管理及线程控制之类的问题。简言之，我要建立的程序将在很多有关 Win32 核心的概念方面给你一个很好的引导。

讨论其细节之前，我需要在此列出这个 API 侦探程序的设计目标：

1. 对一个给出的 Win32 进程，这个程序应该记录对一个给定的函数集合中的函数的调用。

2. 用户可以通过一个配置文件来改变需要监视的 DLL 集合。
3. 调用时参数已知的函数可以在配置文件中说明，且应将这些参数和函数名记录在一起
4. 这个侦探程序必须记录函数的返回值。
5. 这个侦探程序应该能够在 Windows 95、Windows NT 和 Win32 上运行。
6. 不需要更改被侦探程序的源程序或可执行文件。
7. 记录的结果应形成一个磁盘文件而不是即时显示。

这个程序建好后(在本章的末尾)，能做下面几项工作：让你选择要侦探的程序，运行该程序，产生含有记录结果的 ASCII 文本文件。选定的程序运行结束后，你可选用一个编辑器或阅读器来浏览其内容。需要指出的非常重要的一点限制是本侦探程序是一个单进程 API 侦探程序。与 Win16 WinScope 侦探程序之类程序不同之处在于，我的 Win32API 侦探程序并不观察系统中所有进程的函数调用。相反，它只观察某单个进程的函数调用。在一个每个进程均有各自相互独立的地址空间的系统中，编写一个全局(对整个系统)的 API 侦探程序是一个繁重的负担，而且超出了本章的范围。(这样的广告怎么可信?)

拦截函数

所有侦探程序的基本思想都是这样的：侦探程序将它自己插入被侦探程序的控制流程中，在调用到达预定的目标之前得到控制权，并在将控制交还该目标之前做所有要做的工作。我们面对的第一个问题是怎样在“侦探目标”调用一个 DLL 函数之后，到该函数开始执行之前在某处取得控制权。

过去曾用过的一个方法是建立一个你自己的 DLL，它给出你想要拦截的函数的同名函数。例如，假如你想拦截对 KERNEL32.DLL 中 GetProcAddress 函数的调用，你可以建立一个 DLL，在里面放入你自己的 GetProcAddress 函数。将这个 DLL 的引入函数库(imported library)放在库列表的最前面，链接程序将会把对 GetProcAddress 的调用指向你的拦截 DLL 而不是 KERNEL32.DLL。拦截 DLL 在转到真正的函数(如 KERNEL32.DLL 中的 GetProcAddress)之前记录该函数调用的有关信息。作为一种代替创建引入函数库的方式，你可以仅在你的.DEF 文件中给该引入函数起个别名。然而，上面两种方法都存在需要在链接时建立 API 拦截——当你侦探的程序不是你自己编写的时就不能用这种方法。

这种拦截 API 函数调用的方法正好是 Win32 SDK 中的 Microsoft API 参数描述器(parameter profiler)所用的。该描述器包含 5 个.DLL 文件：ZDI32.DLL、ZDVAPI32.DLL、ZERNEL32.DLL、ZRTDLL.DLL 和 ZSER32.DLL；这些 DLL 分别拦截指向 GDI32.DLL、ADVAPI32.DLL、KERNEL32.DLL、CRTDLL.DLL 和 USER32.DLL 中的函数调用。作为代替链接这些 DLL 的一种方法，你运行一个程序(APF32CVT)来修改你要侦探的程序。其结果与你链接那些引入函数库相同。至少，用 APF32CVT 无须修改源代码。

就我们的目标而言，这种方法有两个问题。首先，扩展它去处理新的 DLL 并不

容易。对每个你想拦截的新 API 函数，你需要修改拦截 DLL 并重新建立它。你还必须重新链接或修改要侦探的 EXE 文件。第二个问题更为严重，这种方法需要修改被侦探的程序，这与我们的设计目标之一直接冲突。

另一个拦截 API 函数调用的途径是或多或少地修改被调用的函数。修改一个被调用函数的开始部分，一个程序可以在函数体被执行之前获得控制权。有两种方法可以修改函数开始部分的代码以把控制交给别的程序。第一种方法，也是最显然的方法，是在函数代码的第一个字节处放一条断点指令（操作码为 0xCC）。当函数被调用时，由侦探程序安放的断点处理程序将获得控制权并做登记工作。接着，侦探程序在 CPU 执行一条指令（通过单步中断实现）之前恢复该字节的原来内容。在其单步异常处理程序中，侦探程序接着再次插入断点指令以便捕获对该函数的下一次调用。

虽然某些 Win16 侦探程序用断点来拦截对 DLL 函数的调用，试图在 Win32 环境中用同样的方法却困难得多。对初学者来说，在 Win32 环境中，一个进程不可能得知另一进程的断点，除非它对该进程起调试程序的作用。再说，迫使对 API 函数的每次调用都通过 Win32 结构化的异常处理代码可能严重地影响程序的执行。还有，Win32 环境中，每个进程拥有各自独立的地址空间使侦探程序必须用 ReadProcessMemory 来获得目标应用程序的函数参数。这比能够直接读取内存要艰巨得多。

第二种修改函数开始部分代码来移交控制权的方法是在函数代码的开始部分插入一条 JMP 或 CALL 指令。这种方法有一个问题，即 32 位的代码中，一条 JMP 或 CALL 指令要占至少 5 个字节。如果该 DLL 有某个函数少于 5 个字节（确实有这样的情况），补上一条 JMP 或 CALL 指令是不切实际的，因为即将运行的函数会在这条 JMP 或 CALL 指令的中间开始。断点可以被别的进程处理，但补上的 JMP 或 CALL 指令需要你自己的代码在被侦探程序的进程环境中运行。你的代码要在 DLL 中才可在无关的进程中运行。不过，如你所将看到的，在被侦探进程的环境中运行你的侦探代码并不是很糟糕。然而，补上 JMP 或 CALL 指令确实毫无意义，尤其在需要在你的 JMP/CALL 指令和原来的代码间频繁切换时更是如此。

看过而且讨论过两种明显的拦截方法（链接自己编写的 DLL 和修改 API 函数代码）之后，让我们来看看第三种并不明显的方法。目标说明中没有说必须修改目标 API 函数的代码。修改函数调用同样有效。如果侦探程序能以某种方式找到对 API 函数的调用指令，它就可以修改该 CALL 指令使之指向侦探程序的登记代码。如前面讨论过的一种方法，侦探程序的登记代码需要在被侦探进程的环境中运行。“在另一个进程内接种一个 DLL”小节中说明了怎样使“接种”一个 DLL 到另一个进程的地址空间成为可能。这里，我们将注意力集中在问题的函数拦截部分。

你自己也许会想，“一个程序可能只是对系统的 DLL 就有几百甚至几千次 API 函数调用。我到底怎样才能找到那些 CALL 指令？”不用担心，Win32 EXE 与 DLL 之间动态链接的方式使得这一点难以置信地简单：所有对给定 API 函数的调用总通过可执行文件的同一地方结束转移。修改那个位置使之指向侦探程序的登记代码，你可以拦截该 EXE 文件对该函数的所有调用。

为了明白这是怎样做的，让我们来看看对 KERNEL32. DLL 中 API 函数 GetVersion()的三次分别调用而产生的实际代码。我们以下面这小段 C 语言程序开始：

```
int main()
{
    GetVersion();
    GetVersion();
    GetVersion();
}
```

对这段程序编译产生如下汇编代码：

```
410052: CALL    0042003C
410057: CALL    0042003C
41005C: CALL    0042003C
...
42003C: JMP     DWORD PTR [00440064]
```

此处值得注意的一点是 CALL 指令并不直接调用 KERNEL32. DLL 中的 GetVersion()函数代码。每次调用都将控制转向了在 EXE 文件中某处的一条 JMP 指令。该 JMP 指令引用内存中一个 DWORD(双字)的内容作为操作数而转向该地址。在上面的例子中，DWORD 的地址是 00440064。存放在这个 DWORD 中的地址指向何处？你也许已经猜到，那就是 KERNEL32. DLL 中 GetVersion()函数的开始地址。所有对 API 函数的调用结束于这一仅由 JMP DWORD PTR [XXXXXXXX] 指令构成的程序片段。对可执行程序链入的每个函数，均有一条对应的 JMP DWORD PTR [XXXXXXXX]指令。是谁产生这些片段？当使用 Microsoft 的编译器时，JMP 程序片段是供链接至 DLL 引入函数库的代码。对 Borland C_{++}而言，则由链接程序(TLINK)产生这些 JMP 程序片段。

由这种 JMP 程序片段机制自然引出的问题是“在哪里能发现存储该函数开始地址的那个 DWORD?”和“谁负责对它初始化?”包含函数开始地址的 DWORD 可在称为输入地址表(import address table，简写为 IAT)中找到。IAT 通常存放在每个可执行文件的. idata 部分(import data)。对每个链接至一个可执行文件的 DLL，有一个包含在链入的 DLL 中函数的开始地址为元素的 DWORD 数组。当 Win32 装载程序将一个可执行文件装载至内存时，它将在这个数组中填入正确的地址，如图 10-1 所示。在装载该可执行文件之前，每个 DWORD 存放目标函数的函数名 ASCIIZ 字符串的位移(如 GetVersion)，装载该可执行文件时，装载程序用函数的实际地址改写之。

知道一个可执行文件如何链入其它 DLL 中的函数后，很容易明白一个侦探程序如何以最小的麻烦和开销来拦截和登记那些函数调用。侦探程序只须找到可执行文件入口部分中的函数开始地址数组，并用登记例程的开始地址来代替它们。无需修改代码本身，也就无需在原先的代码和修改后的代码之间不断切换。可执行文件调用 API 侦探程序的代码后就结束，所以唯一的问题是登记函数。侦探程序完

成对该函数的登记后转向原来的 JMP 程序片断的目标地址。很简单，不是吗？

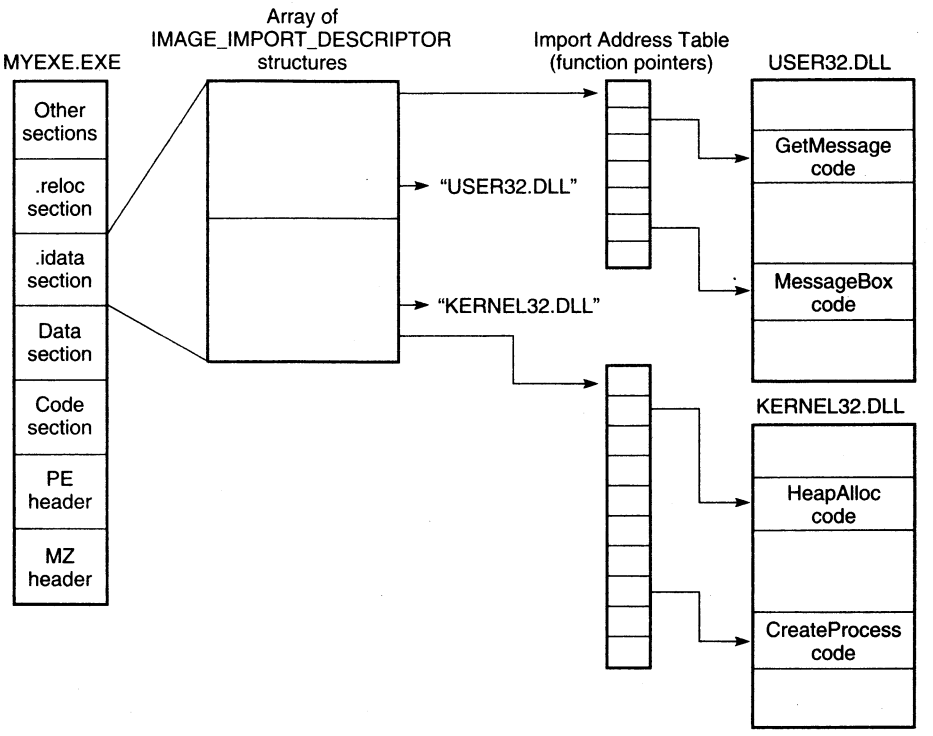


图 10-1　可执行文件的.idata 部分通常存放着一个引入函数的地址 DWORD，虽然也可在别处找到链入表

即使你的目的不在于侦探别的程序，你同样可以使用这种修改链入部分中地址的技巧来有选择地拦截函数。例如，你也许想有你自己的代码来取代某 DLL 中的一个函数。很容易用一个以 DLL 名和该函数名为参数的函数来获取一个指向含有该函数开始地址的链入数据部分中 DWORD 的指针。接着，你的代码将有你自己的函数的地址来改写那个 DWORD。如果你想链接到原来的地址，只须在改写前简单地将 DWORD 原来的内容保存到别处。

我将在本章建立的侦探程序中选择刚才描述的那种方式来拦截对 API 函数的调用。在做这个决定的过程中，我曾答应负责在被侦探进程的环境中运行侦探程序的登记代码。由于设计目标不允许对目标应用程序重新链接或修改，我需要以某种方式迫使登记代码运行在目标程序的地址空间中。这还意味着 API 侦探程序的代码的主要部分必须在一个 DLL 中。

在另一个进程内接种一个 DLL

现在我们已知道如何拦截对 API 函数的调用，下一个障碍是强制侦探代码进入目标程序的地址空间中运行。在 16 位 Windows 中，所有程序共享一个地址空

间，因而这并不是问题。但在 Win32 中，每个进程拥有各自独立的地址空间及自己的 DLL。仅因一个进程正在使用某 DLL 意味着别的进程不能使用这个 DLL，或者通过隐式地链接在一起，或者通过调用 LoadLibrary()。由于我们要侦探的程序并不知道我们的 API 侦探 DLL，我们需要用低层的技巧来使这个 DLL 进入被侦探程序的地址空间。

至少有三种方法可以将一个 DLL 接种到一个独立进程的环境中去。Jeffrey Richter 在 94 年 5 月的《微软系统》杂志(Microsoft System Journal)上有篇文章非常详细地描述了每一种方法。这里，我将简要介绍我们不用的两种，然后把主要的篇幅集中在侦探程序实际采用的那种方法上。Richter 为一个通用的应用所选的方法与我将用在侦探程序中的方法近似(但不完全相同)。关键的不同点在于 Richter 的方法用了 Win32s 和 Windows 95 中没有的 CreateRemoteThread 函数。我的接种方法则可在这三个平台上任意移植。

第一个方法，也是最普遍的方法，是用 SetWindowsHookEx()函数安装一个“窗口钩调函数(windows hook)”来强制一个 DLL 进入另一个进程的地址空间。如果你为一个不同的进程说明一个 hThread，或请求一个系统范围内的“钩调函数(hook)”，操作系统自动将“钩调过程(hook procedure)”的 DLL 装载到受到钩调函数影响的所有进程的地址空间中。有两点理由可以说明装一个窗口钩调函数来使我们的 API 侦探 DLL 装载是不起作用的。第一点理由是你必须在一个现存的进程上安装钩调函数。当这点实现时，目标进程毫无疑问已经调用过 API 函数。侦探程序将错过在此以前目标进程的所有 API 函数调用。第二点理由是在目标进程有引发钩调函数的调用之前，钩调函数 DLL 不会被实际装载到目标进程上。企图使用钩调函数来强制一个 DLL 进入另一进程的环境恰好不能保证把握 DLL 被装载的准确时机。

第二种方法则流于仅仅是归档一类的东西。看起来好象在寄存器堆中深藏着一个不可见的寄存器关键字值。

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\APPINIT_DLLS
```

将你的 DLL 的文件名加入这个关键字区(key)，操作系统将自动地在每个进程启动时将 DLL 装载到它的地址空间中去。有几条理由说明这种方法对一个侦探程序来说并不适合。主要的一点是对登记项(registry)的修改要到系统下一次启动时才起作用。为了侦探一个程序，你必须先重新启动系统。这不可行！这种方法的另一缺点是侦探程序如果想侦探刚进入其环境的进程，它就需要一个一个例子去判断。对你不想侦探的程序，侦探程序应该在它的 DllMain()过程中返回 0 来对应 DLL_PROCESS_ATTACH 的值。从 DllMain 返回的 0 告诉操作系统不应该为这一进程装载这个 DLL。然而，另一个问题是操作系统将试图在每个进程(甚至在那些被隐藏的进程)中拦截这个 DLL，你不能对其干预(象 MPREXE.EXE)。

第三种方法是一种暴力方式，这就是我们用在 API 侦探程序中的方法。理想情况下，我们以某种方式告诉目标进程它应该调用 LoadLibrary 来装载我们的侦探 DLL，并且应该一启动就调用 LoadLibrary。当我们不能直接这么做时，没有理

由说我们不能用某种技巧来使目标进程为我们装载侦探 DLL。

让我们看一个例子以更好地说明我在建议些什么。假如你想进入一个用声纹识别锁锁住的银行金库。只有一个人的声音模式可以打开那道锁，而你不能。那个人不愿为你开那道锁。然而，你可以对他进行催眠。等他进入催眠状态后，让他说话打开那道门。在使他脱离催眠状态之前，叫他忘记刚发生的一切。

既然如此，怎样把它应用到装载一个 DLL 呢？如果我们可以冻结(或催眠，你喜欢叫什么都行)目标进程。我们可以修改该进程的内存和寄存器值，使它看起来象是在自愿地调用 LoadLibrary。适当设置其内存和寄存器值后，我们解冻该进程并使它运行。最后结果是目标进程调用 LoadLibrary，操作系统通过装载 API 侦探 DLL 而迫使它进入该进程的地址空间。LoadLibrary 返回后，我们再次冻结目标进程，恢复其内存和寄存器原来的值，使该进程继续执行，就象什么也没有发生过。

你也许会想到，冒充目标进程调用 LoadLibrary 的代码很复杂。它要修改目标进程的代码。所以第一步是算出需要修改第几代码页，并将它保存在另一区域以备以后恢复之用。接种的代码还需要修改目标进程的寄存器内容，所以它也需要保存一份原来寄存器值的拷贝。幸运的是，Win32 提供了 GetThreadContext 函数，它可以将一给定进程的所有寄存器值保存到一个 C 结构中。

接下来，我的代码产生一段代码来从目标进程的环境中来调用 LoadLibrary。这段代码包含一个含有侦探 DLL 文件名(APISPY32.DLL)的 ASCII 字符串。这段代码中紧跟在调用 LoadLibrary 代码之后的是一条断点指令。这样装载程序可以在 LoadLibrary 一返回就获得控制权。一旦这段代码产生后，我用 WriteProcessMemory 函数把它写到目标进程的第一页中。紧接着，我改变目标进程的 EIP 寄存器的内容，使它将从我的那段代码开始恢复执行。

如此设置内存和寄存器内容之后，API 侦探程序使目标进程执行。如果一切正常，目标进程将成功地执行 LoadLibrary 的代码并返回到我设的断点处。又遇到这个断点，目标进程再次被暂时冻结。侦探程序抓住这次机会恢复它保存的代码页的原来值，并恢复原来的寄存器值(再次使用 SetThreadContext())。一切恢复原状后(除了我们附加在目标进程地址空间中的侦探 DLL 外)，断点处理程序使目标进程恢复执行。回头我在“APISPY32 程序”一部分给出侦探程序的代码时将更详细地讨论这种强迫另一进程装载一个 DLL 的方法。

用 DEBUG API 来控制目标进程

在另一进程的环境中装载一个 DLL 时，对于进程的执行进行精确的控制是最基本的。Win32 调试 API 提供了所有我们需要的基本信息。特别地，我们需要准确地知道目标进程执行第一条指令的时刻，这样我们才能接种侦探 DLL。另外，我们必须保证当我们在目标进程的地址空间中动手术时，它不会突然开始执行。使用调试 API 可以解决这个问题。无论被调试的进程何时向调试程序报告什么事情，被调试的进程中所有线程将被挂起，直到调试程序告知操作系统让它们恢复执行。

如果我们编写一个 16 位 Windows 程序的侦探程序，TOOLHELP NotifyReg-

ister 和 InterruptRegister 函数就可以做这些事。TOOLHELP NFY_STARTTASK 报告将允许我们在一个新任务真正开始执行之前知道它何时开始执行。不巧，TOOLHELP 报告回调模块假定所有进程运行在同一地址空间。TOOLHELP 报告回调模块在 Windows NT 或 Windows 95 的地址空间各自独立的情况下不起作用，所以我们要用最相近的代用品——Win32 调试 API。

使用 Win32 调试 API 监视目标进程的执行在我们的 API 侦探程序上加了一个特定的体系。API 侦探程序将由两部分组成。第一部分是在目标进程中拦截和登记函数的代码。由于我们将把这些代码接种到被监视进程的地址空间中，它必须在一个 DLL 中。第二部分由装载被侦探程序的装载程序组成。装载该进程后，侦探程序的可执行代码进入一个调试循环，它主要由对 WaitForDebugEvent()和 ContinueDebugEvent()函数的调用组成。当 WaitForDebugEvent()返回调试事件时，装载程序检查它们并采取必要的行动。可能由 WaitForDebugEvent()返回的事件有：

```
EXCEPTION_DEBUG_EVENT
CREATE_THREAD_DEBUG_EVENT
CREATE_PROCESS_DEBUG_EVENT
EXIT_THREAD_DEBUG_EVENT
EXIT_PROCESS_DEBUG_EVENT
LOAD_DLL_DEBUG_EVENT
UNLOAD_DLL_DEBUG_EVENT
OUTPUT_DEBUG_STRING_EVENT
RIP_EVENT
```

确切地说，如果你将 Win32 调试事件与 Win16 NotifyRegister 函数返回的报告比较一下，你将注意到两者惊人的相似。还有，如果你想在一个程序中使用 WaitForDebugEvent 函数并显示所有可能的返回信息，可以查一下 Win32 SDK 中的 DEB 样本程序。

一旦我们的装载程序处理完调试事件报告，它就调用 ContinueDebugEvent()去通知操作系统可以让被调试的程序恢复执行。将 WaitForDebugEvent()和 ContinueDebugEvent()放在一个循环中，装载程序可以看到被侦探进程生存期中的所有重要的事件。

对我们的 API 侦探程序来说，最重要的调试事件是 EXCEPTION_DEBUG_EVENT。就在一个进程刚要执行之前，由函数 WaitForDebugEvent()返回一个除类型为 STATUS_BREAKPOINT 之外的 EXCEPTION_DEBUG_EVENT 报告。API 侦探程序的装载程序把这作为用我先前描述的方式迫使侦探 DLL 进入该进程地址空间的线索。当 LoadLibrary 返回到我们插在该进程代码区的断点时，装载程序将看到另一个 STATUS_BREAKPOINT 异常。装载程序用这种方法来知道何时应该恢复寄存器和内存页的原来值。

一旦装载程序执行了经过两次断点异常的目标进程，它的工作就基本做完了。然而，Win32 侦探程序并没有提供一种方法来让一个调试程序告诉操作系统它不想再接收调试事件报告。一旦你将调试 API 用在一个进程上，该进程就会在每次

产生调试事件时被挂起。调试程序调用 ContinueDebugEvent 处理每一调试事件是保持被调试程序运行的唯一方法。因此，API 侦探程序需在由 WaitForDebugEvent 和 ContinueDebugEvent 组成的循环中转来转去，直到目标进程终止。即使我们确实只需要几个调试事件，我们也被迫接收所有的。我们可以忽略我们不感兴起的调试事件，并只调用 ContinueDebugEvent 而不作进一步的处理。API 侦探程序的装载程序可用伪码描述如下：

```
Load Process to be spied on
while ( TRUE )
{
    WaitForDebugEvent()

    if ( debug event is a breakpoint )
    {
        if ( first breakpoint )
            modify debuggee to make it load the spy DLL
        else if ( second breakpoint )
            restore original register and data pages of debuggee
    }
    else if ( debug event is an EXIT_PROCESS )
        break out of loop

    ContinueDebugEvent()
}
```

编写登记 API 函数的程序段

至此，我们已解决了与操作系统主要体系有关的问题：

- 如何拦截 API 函数
- 如何将侦探 DLL 装载到目标进程的地址空间中
- 如何精确地控制目标进程的执行

当然还有问题需要处理，但它们和操作系统并没有直接的关系。一个这样的问题就是处理被重定向的 API 函数调用的那段代码。虽然试图为所有被我们重定向到我们的侦探 DLL 的函数调用做一个单一的入口是很有吸引力的，但也正是不可行的。没有办法可以从侦探 DLL 中一个单一的入口点知道它正登记那个函数。相反，我们应该为我们拦截的每一个函数产生一个独立的程序片段。转换程序片(thunk)这个词通常被用来刻画一段将控制权转移之前做某些处理的很短的程序片段。虽然也可以把我将产生的一段段的代码叫作转换程序片，但为了将我的代码和 Windows 的转换程序片相区别，我使用存根(stub)这个术语，所有用在我们的侦探程序中的代码存根大体相似而稍有不同。当每个存根找到一个重定向的函数调用，它将该函数的独特的信息压栈，然后调用一个公共例程来登记这次调用。

如果我们想做的仅仅是对一个给定不变的函数集进行拦截，很容易写出一些宏并在编译期间产生所有的代码存根。由于我们的设计目标指出了这个 API 侦探

程序应该是可扩展的，在编译期间产生这些存根就不可行。也就是说，我们需要动态地产生基于一个配置文件中信息的存根，在 Win32 环境中这并不难。

对我们需要的每一存根，我们可简单地为其分配一定的空间并写入我们的代码，在 16 位 Windows 环境中这要困难些，因为我们需要以某种方式在代码段中分配一块内存，而不是由内存分配函数从数据段中分配。即使我们有了代码段中某部分空间，我们也不能写入我们的存根，因为不允许向代码段写。为了向代码段写存根，我们需要用别名选择器(alias selectors)或 TOOLHELP MemoryWrite()函数。在 Win32 环境中不会有这个问题，因为代码段和数据段都映象到同一地址范围。我们可用平常的正常模式数据指针来写入我们的代码，稍后去执行它。

为了产主那些存根，侦探 DLL 从一个输入文件(APISPY32.API)读入关于每一被拦截函数的下列信息：

- 包含该函数的 DLL
- 函数名
- 函数参数

侦探 DLL 为每个函数产生一个包含代码和数据的存根，其形式如图 10-2。存根的代码部分首先保存所有的 32 位通用寄存器。这并不是必要的，但良好的编码习惯意味着你需要将一切恢复到你发现它们时的状态。接着，存根将三个指针压栈，为调用登记函数作好准备。这三个指针分别指向函数名、返回地址和栈内函数及函数参数的选项信息。(过一会儿我将讨论参数信息。)登记函数做完工作后，存根恢复所有通用寄存器并转移到原来应该被调用的代码(就象我们没有拦截它一样)。

```
DWORD RealAddressOfInterceptedFunction;

pushad                      ; Preserve all registers.
lea     EAX, [ESP+32]
push    EAX                 ; Push pointer to the return addr and params.
push    [pParamInfo]        ; Push pointer to byte-encoded parameter info.
push    [pszFunctionName]   ; Push pointer to the function's name.
call    LogCall             ; Call function that does logging.
popad                       ; Restore original registers.
jmp     [RealAddressOfInterceptedFunction]  ; Jump to the original code.

char    szFunctionName[]    ; ASCIIZ name of the function.
BYTE    paramInfo[]         ; Optional byte-encoded parameter info.
                            ; First byte is the length of the info.
```

图 10-2　侦探 DLL 为每个被拦截的函数产生一个包含代码和数据的存根

你也许会想到，所有存根需要在我们能够重定向 API 函数调用之前产生。在侦探程序产生每个存根时，它将该存根的地址加到一个存根的指针数组中。很容易将目标程序中的函数调用重定向到对应的存根。侦探 DLL 将找到每个目标进程中引入函数的地址。引入函数的地址保存在一个由 PE 头文件中的 DataDirectory

[IMAGE DIRECTORY ENTRY IMPORT]. VirtualAddress 指向的表中。接下来，侦探 DLL 重新在它刚产生的数组中找一个在其第一个 DWORD 中含有同样地址的存根。如果找到一个相同的，侦探程序用该存根的第一条指令的地址来代替目标程序的入口表中对应的 DWORD。这个过程如图 10-3 所示。

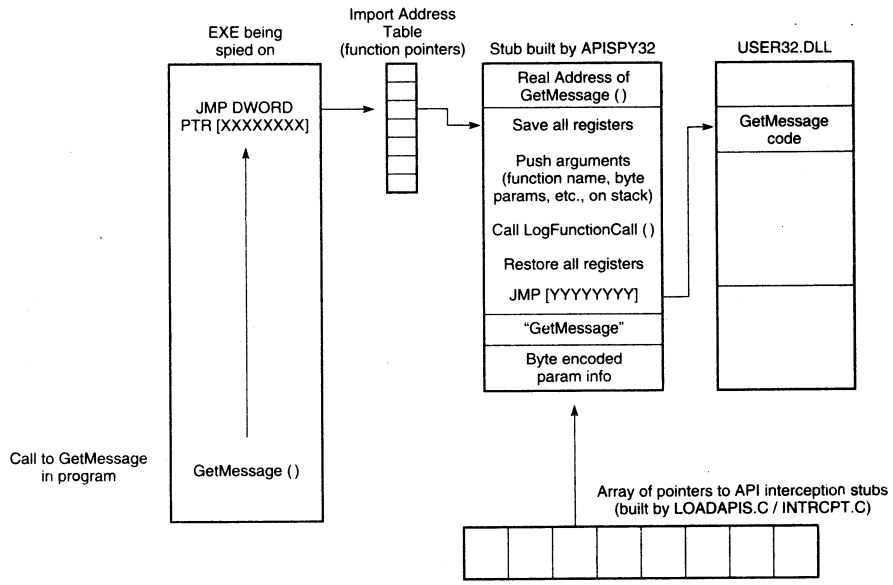


图 10-3　引入函数的地址通过建立在浮动空间上的存根被重定向。这些存根调用 APISPY32 的代码来登记那些函数调用的函数名及其参数

参数信息编码

API 侦探程序的用途的很大部分在于它们报告 API 函数参数的实际值。登记函数无法支付为每个 API 函数及其参数保存一个独立编号部分的开销。另外，你必须加入代码并重新编译 API 侦探程序以加入新函数。

将函数参数表示成一些可以被解释为登记一个函数调用的一部分的紧缩格式是一种较好的方法。由于 Win32 编程中参数的类型是有限的。我决定用一个单独的 BYTE 来表示每种基本类型。这些基本类型包括 BYTE、WORD、DWORD 和 LPSTR。为保持简单性，我将所有类型的指针用 LPDATA 来表示，但 LPSTR 和 LPWSTR 除外。如果你想使侦探程序更好用，你可以扩展参数类型以包含其他类型，包括指向特定数据结构的指针(例如 LPRECT)。登记代码可以用这些附加的参数类型信息来显示更多关于参数的细节(如 LPRECT 指向的结构有哪些域)。然而，正如我说过的，我的目的是保持问题简单，所以提供的代码中只有十种不同的参数类型。

在我们的 API 侦探程序中，参数信息放在为每个 API 函数产生的存根的最后面。参数信息的第一个 BYTE 说明该函数有几个参数。紧跟其后的是说明参数类

型的那些 BYTE，按它们出现在函数申明中的顺序依次给出。例如，如果我们定义 BYTE 值 8 代表 HWND 而 1 代表 DWORD。函数 GetWindowWord(HWND, DWORD)的参数信息编号为：2,8,1(两个参数，第一个是 HWND(8)，第二个是 DWORD(1))。没有参数的函数用一个单独的 0 表示。

将参数编码信息解码以显示其值是很简单的。API 函数的存根传给登记函数的值之下就是指向栈顶的指针(ESP 寄存器)，也就是该函数的入口。栈顶的 DWORD 就是 API 函数执行完后应该转向的地址。紧靠着的内存区域中放着调用者压在栈中的那些参数。登记函数解释 BYTE 编码的参数信息。对每一参数编码，登记函数从栈里找出对应的 DWORD 并输出一个含参数类型及值的字符串(如 LPSTR:00410068)。

每处理完一个参数，存根将指针加 4 以指向下一个参数。对 Win32 API 来说，一个好处在于参数是从后向前压栈，这使得第一个参数最先出现。如果 Win32 API 用 pascal 调用惯例(从前向后)，就象 Win16 API 那样，对参数编码的解释将因为第一个参数随不同的 API 函数而出现在不同的位置而变得更困难。

函数返回值

到此我们已掌握了拦截函数和登记它们的参数的机制。我们可以就此打住并开始应用我们的代码。不过，我们的设计目标指出我们还需要登记函数的返回值。这使问题从多个方面变得更困难。对一个函数的调用必须经过一点(这使我们可以拦截它们)，而 API 函数可能从好几个地方返回。我们怎样才能在那一点得到控制权而找出保存着返回值的 EAX 寄存器呢？

在我们让 API 函数真正的代码执行之前，我们所知唯一关于该函数返回的是它将返回的地址。跳进脑中的一个显而易见的答案是在那个地址上设一个断点。一个与之有关的方法是在那里补上一条转向我们的代码的 JMP 指令。虽然这两种方法都可用(通常是这样的)，它们是非常混乱的，且存在着与我先前讨论的拦截函数同样的问题。一种不那么明显但比较清楚的方法(而且不需要修改代码)是将栈内的返回地址改成侦探程序的返回值登记代码的开始地址。当然，你应该暂时保存它原来的值，并应在 API 函数的真正代码执行之前做到这一点。让原来的函数代码执行后，返回值登记代码就进去了。登记该函数的返回值后，登记代码将原来的返回地址拷贝到栈内，这样登记代码返回时目标进程可正确返回。

恐怕你会认为上述获取返回值的方法过于简单，但注意这儿有一个圈套。在 Win16 和 Win32 中，一个 API 函数在执行中也许需要调用其他的 API 函数。典型的例子是 DispatchMessage。DispatchMessage 是一个调用你的程序中窗口过程的函数。当你在你的窗口过程中调用 Windows 函数时，你实际上是在从一个 API 函数(在这个例子中是 DispatchMessage)的执行过程中调用另一 API 函数。这有什么不对吗？在我刚才描述的拦截函数返回值的简单方法中，一个单独的变量保存着原来的地址。假如你遇上嵌套的 API 函数调用，将只有最近一次调用的返回地址被保存下来。更深一些的嵌套函数的返回地址就丢掉了。

为了解决这个嵌套 API 函数的问题，我用栈来保存返回地址。无论存根何时将一个返回地址改成指向我们的登记代码，它将返回地址原来的值压到我们的返回地址栈的栈顶。当返回值登记代码准备好返回调用者时，它将栈顶的地址弹出并按它返回。在某些方面上，我们的返回地址栈很象真正的程序栈。关键的不同点是我们的返回地址栈不包含参数，而且只包含那些被侦探 DLL 拦截的函数的返回地址。

连嵌套 API 函数调用也考虑到后，我们就可以编程了，对吗？没有这么快。Windows NT 和 Windows 95 环境下的程序支持多线程执行。每个线程使用独立的栈，并且并不关注别的线程正在做什么。为了处理多个线程，API 侦探 DLL 为目标程序启动的每个线程维持一个独立的返回地址栈。由于侦探程序不能预先得知目标进程将启动多少条线程，无论何时启动一个新的线程，就要为它分配一个返回地址栈的内存空间。对我们的侦探程序来说，幸运的是，Win32 使我们很容易知道何时创建了一个线程。操作系统每次调用所有 DLL 的入口点（如 DllMain）来启动一个新的线程。你在下一部分即将看到，DllMain 函数为每个新启动的线程分配用作返回地址栈的内存。我将在下一部分给出侦探程序的代码。

指向每一线程的返回地址的指针用 Win32 的线程局部存储区（Thread Local Storage，TLS）机制来保存。线程局部存储区允许你为每一线程保存一个独立 DWORD 的集合，但以一种恒定的方式来检索，而不管哪个线程正在执行。第三章详细讨论过线程局部存储区的应用。要使用线程局部存储区，你需要先用 TlsAlloc 函数分配一个索引值并把它保存到一个全局变量中。这样，每个线程可以通过将这个索引值传递给 TlsGetValue 来找出它的线程说明数据。如要将一个线程值保存到别处，你可以调用 TlsSetValue，将 TLS 索引和你想保存的当前正在执行的线程的值传递给它。在我们的 API 侦探 DLL 的情形中，我们想要保存的线程值是指向该线程返回地址的指针。API 侦探 DLL 在首次装载并正在 DllMain 中处理 DLL_PROCESS_ATTACH 码时分配 TLS 索引值。

某些 Win32 程序员也许已经意识到__declspec（线程）编译器指令。用__declspec（线程）是一种不使用 TlsXXX 函数而产生线程的方便的方法。（参看第八章可知道更多关于__declspec（线程）如何作用的信息。）难道将线程栈做成一个__declspec（线程）变量不比用 TlsXXX 函数更容易吗？不幸的是，__declspec（线程）变量不能在用 LoadLibrary 函数装载的 DLL 中正常工作（虽然它在隐式装载的 DLL 中工作得很好）。我们的 API 侦探 DLL 是用 LoadLibrary 装载的，所以__declspec（线程）对我们来说毫无用处。

你也许想知道我们的侦探程序在 Win32s 下将做些什么，因为 Win32s 并不支持多线程。Microsoft 已做好了这些事并将 TLS 函数包含在 Win32s 的库中。虽然一个 Win32s 程序的 TLS 数据本质上仅是全局数据，但很重要的一点是我们的侦探程序可以用 TlsXXX 函数而不必担心运行在哪一个操作系统上。

你可能已经明白，获取 API 函数的返回值要比初看起来难得多。我们不仅要保持一个返回地址栈，而且要为目标进程的每个线程分别保持一个这样的栈。不可能有比这更复杂的事了，是吗？再揣摩一下。

Win32 API 的简洁的特性之一是结构化的异常处理，有点象 C++ 中的异常处理，但它不完全相同。（更详细关于结构化异常处理如何工作的信息见第三章。）问题是结构化异常处理会破坏我们的返回地址栈。比如说你在对 DispatchMessage() 调用的前后放一个 try/except{}。在 DispatchMessage()最终调用的窗口过程中，你的代码将产生一个异常（如一个 STATUS_ACCESS_VIOLATION）。处理该异常的 except 块就是在 DispatchMessage 代码后的 except{}块。问题就在这里。出于高效，CPU 并不等 DispatchMessage 返回就跳到 except{}块。由于我们的返回值登记代码不会被从 DispatchMessage 的返回调用，我们不知道应该移去线程返回地址栈中 DispatchMessage 的返回地址。这种情况如果反复发生，线程返回地址栈很快就会溢出。

与我们遇到的其他关于返回值登记的问题不同，由结构化异常处理引起的问题没有优雅而简洁的解决方法。我在商用程序（BoundsChecker32）中曾使用过的方法非常混乱，且十分复杂而难以理解，但我没有在 API 侦探 DLL 中写入类似的代码，因为这将使程序大大地复杂化。我决定忽略结构化异常处理的困难的依据在于很少有弹出嵌套的 API 函数而不返回的程序。到现在为止，除了我特意设计后写在这儿的测试程序外，我还从未见过有哪个程序在用这里给出的登记代码时遇到结构化异常处理的问题。

返回地址栈的一个良性的副作用在于我们可以通过栈指针算出我们在第几层 API 函数的嵌套调用中。登记代码利用这点来对函数调用进行缩进，并返回在另一 API 函数中调用的函数的行号。一个 API 函数嵌套得越深，它在登记文件中出现的位置就缩进得越多。当登记代码准备输出一个函数调用或返回值记录行时，它检查返回地址栈的指针并相应地缩进该行的起始位置。在登记文件中，通过找开始位置相应的记录行很容易将返回值和函数调用对应起来。例如，你可以很容易地看到调用行（顶行）与返回值行（底行）相对应：

```
DispatchMessageA(LPDATA:80B6AE68)
  LoadCursorA(HANDLE:00000000,LPSTR:00007F00)
  LoadCursorA returns: 2CE
  SetCursor()
  SetCursor returns: 2CE
DispatchMessageA returns: 0
```

APISPY32 程序

现在我们已清楚我们将用在侦探代码中的那些原理，该来讨论我们将实际编写的 API 侦探程序代码了。我首先将给出组成该 DLL 的函数，然后给出加载程序。不用担心，这儿不会有一大堆代码。我曾为程序量很小而感到惊讶。

我编写的 API 侦探程序的名字是 APISPY32，它也是侦探 DLL 入口点源文件的名字。图 10-4 给出了非常重要的第一部分：

```
HINSTANCE HInstance;
BOOL FChicago = FALSE;
#if defined(_ _BORLANDC_ _)
#define DllMain DllEntryPoint
#endif

INT WINAPI DllMain
(
    HANDLE hInst,
    ULONG ul_reason_being_called,
    LPVOID lpReserved
)
{
    // OutputDebugString("In APISPY32.C\r\n");

    switch (ul_reason_being_called)
    {
        case DLL_PROCESS_ATTACH:
            HInstance = hInst;
            FChicago = (BOOL)((GetVersion() & 0xC0000000) == 0xC0000000);

            if ( InitializeAPISpy32() == FALSE )
                return 0;
            if ( InitThreadReturnStack() == FALSE )
                return 0;
            break;

        case DLL_THREAD_ATTACH:
            if ( InitThreadReturnStack() == FALSE )
                return 0;
            break;

        case DLL_THREAD_DETACH:
            if ( ShutdownThreadReturnStack() == FALSE )
                return 0;
            break;

        case DLL_PROCESS_DETACH:
            ShutDownAPISpy32();

            if ( ShutdownThreadReturnStack() == FALSE )
                return 0;
            break;
    }

    return 1;
}
```

图 10-4　APISPY32.C 中 DllMain 函数的第一部分

DllMain 函数用了一个 switch 语句来处理四种重要的进程/线程事件。我在下面的描述中说明“事件”时，我实际上说的是 DllMain 将 dwReason 置为特定值而

发出的一个请求。DLL_PROCESS_ATTACH 事件是我们拦截所有目标进程函数调用和设置函数登记的线索。

对一个进程的第一个线程来说，操作系统不会用一个 DLL_THREAD_ATTACH 事件为参数来调用 DllMain。实际上，你需要认为 DLL_PROCESS_ATTACH 还隐含了一个 DLL_THREAD_ATTACH 事件。我们需要为目标进程的所有线程分配一个线程返回地址栈，所以 DLL_PROCESS_ATTACH 和 DLL_THREAD_ATTACH 的处理程序都调用 InitThreadReturnStack 来建立线程返回地址栈。这儿隐含了一个假定，即这两个通知都是在最新创建的线程中产生的。DLL_THREAD_DETACH 事件的处理程序调用 ShutDownReturnStack 来释放线程返回地址栈使用的内存。最后的事件，DLL_PROCESS_DETACH，它的处理程序调用 ShutDownAPISpy32。当下这个函数除了关闭登记文件以使操作系统内部缓冲区中的内容确实被写到磁盘中以外什么也不做。实际上我们还可以恢复该 EXE 文件入口部分中原来的地址，但没有理由说我们必须这样做。对该进程的最后一个线程来说，并不随线程的创建而产生一个显式的 DLL_THREAD_DETACH。而且 DLL_PROCESS_ATTACH 事件处理程序也会调用 ShutDownThreadReturnStack 来释放最后一个返回地址栈。

APISPY32.C 的其余部分在图 10-5 中给出。InitializeAPISpy32 函数的第一个动作是调用 LoadLibrary 例程。LoadAPIConfigFile 读入含有 API 函数及其参数信息的 .API 文件并用这些数据来产生存根。(稍后我将在代码浏览中更详细地讨论这个函数。)

```
BOOL InitializeAPISpy32(void)
{
    HMODULE hModExe;
    DWORD moduleBase;

    if ( LoadAPIConfigFile() == FALSE )
        return FALSE;

    if ( OpenLogFile() == FALSE )
        return FALSE;

    hModExe = GetModuleHandle(0);
    if ( !hModExe )
        return FALSE;

    if ( (GetVersion() & 0xC0000000) == 0x80000000 )    // Win32s???
        moduleBase = GetModuleBaseFromWin32sHMod(hModExe);
    else
        moduleBase = (DWORD)hModExe;

    if ( !moduleBase )
        return FALSE;

    return InterceptFunctionsInModule( (HMODULE)moduleBase );
```

```
}

BOOL ShutDownAPISpy32(void)
{
    CloseLogFile();

    return TRUE;
}
```

图 10-5　APISPY32.C 中的初始化函数和关闭函数

建立起 API 函数的存根后，初始化的下一步是打开输出文件为输出函数调用及返回值信息作好准备。初始化的最后部分是调用 InterceptFunctionModule 将目标程序中的函数调用重定向到前面建立的存根。InterceptFunctionModule 函数需要知道目标进程在内存中的装载基地址(base load address)。这样它才能找到引入函数部分的表。在 Windows NT 和 Windows 95(Windows 4.0)中，一个运行中的程序的 HMODULE 与它的装载基地址相同。由于我们的 DLL 不是在主 EXE 文件中，所以它的 HMODULE 不是我们所想要的。实际上，我们调用 GetModuleHandle(0)，在 Win32 环境中不管你在哪里调用它都返回该 EXE 的 HMODULE。在 Win32s 中，我们还需再做一步工作，因为一个 HMODULE 的值并不就是基地址。我编写了 GetModuleBaseFromWin32s 函数来获取一个 Win32s 模块的基地址。这个在 W32SSUPP.C 中的例程使用了两个未公布的 Win32s 函数来将 Win32s HMODULE 转化为基地址。关闭 API 侦探程序要比初始化简单多了，仅由一个对 CloseLogFile 调用组成。

图 10-6 给出了负责读取 APISPY32.API 文件的代码。每读完一个函数的描述，它就调用 INTRCPT.C 中的 AddAPIFunction 来为该函数的存根分配内存并以一定的格式初始化。APISPY32.API 是以行为单位的 ASCII 文本文件。行前的空白和空行是允许的，但在一个有效的行后面的额外字符是不允许的。不可识别的行将被忽略并继续处理下一行。

定义 API 的语法非常简单。对你想拦截的每个函数，只须按下面的形式加一行：

```
API:ModuleName:FunctionName
```

例如：

```
API:USER32.dll:GetMessageA
```

你可以紧接在一个新的函数定义之后放参数信息，每个参数占一行。例如：

```
API:USER32.dll:GetMessageA
LPDATA
HWND
DWORD
DWORD
```

合法的参数关键字由下面的内容组成，并存放在数组 ParamEncodings 中：

```
DWORD       ; Any general-purpose 4-byte value (DWORD, UINT, int, etc.)
WORD        ; Any general-purpose 2-byte value (WORD, USHORT, short, etc.)
BYTE        ; Any general-purpose 1-byte value (BYTE, char, etc.)
LPSTR       ; Pointer to a null terminated ASCII string.
LPWSTR      ; Pointer to a null terminated unicode (wide) string.
LPDATA      ; Pointer to any data, other than LPSTRs and LPWSTRs.
HANDLE      ; A handle value (other than HWNDs).
HWND        ; An HWND.
BOOL        ; A BOOL parameter.
LPCODE      ; Pointer to code (e.g., FARPROC, WNDPROC, etc.).
```

为了尽量减少你第一次使用 APISPY32 的紧张感，我在提供的 APISPY32.API 中预置了包含在 KERNEL32.DLL、USER32.DLL、GDI32.DLL 和 ADVAPI32.DLL 中的函数及参数信息。你可以在这个文件中加入另外的函数定义。COMCTL32.DLL 就是一个可取的选择。假如你想和几个不同的项目一起使用 APISPY32，你也许想扩展 LoadAPIConfigFile 函数的功能以使它可从多个 .API 文件中读取数据。

在我已定义的参数类型中有些重迭。例如，HWND 也可以编码成 HANDLE 或 DWORD。我定义这些关键字的目的在于包含大多数常见的类型以便在输出文件中显示参数的方式上灵活些。将 LPSTR 和 LPDATA 两种参数类型分开，我们就可以在遇到一个 LPSTR 类型的参数时显示一个字符串。如果我们将 LPSTR 并入 LPDATA，我们就不知道哪个参数可以被显示成有意义的字符串。另一个我未使用的可能性是将 BOOL 类型的变量显示成 TRUE 或 FALSE 而不是数值。同样 HWND 也可用于在输出文件中显示窗口标题的一部分。这样在察看登记文件时就很容易将一个 HWND 类型的值和一个具体的窗口联系起来。

如果你想让它更好，你可以随意扩展我定义的编码方式。加入一个新的参数类型是很容易的。在文件 PARAMTYPE.H 中定义了一个称作 PARAMTYPE 的枚举类型变量，将你的新参数类型加到该变量的后面。然后，因为参数名称应该在 .API 文件中，将它加进去，并把你的枚举值加到数组 ParamEncodings 的尾部。最后，在 LOG.C 的登记代码中加入你想为你的新参数类型打印数据的语句。很明显的一个方向是定义更多具体的指针类型。如 LPMSG 是一个常见的类型，定义一个 LPMSG 参数，登记代码就可以辩认出 LPMSG 指针而在登记文件中加入该 MSG 结构成员的值。图 10-6 给出了 LOADAPIS.C 的开始部分，即 .API 文件分析代码：

```
BOOL IsNewAPILine(PSTR pszInputLine);
BOOL ParseNewAPILine(PSTR pszInput, PSTR pszDLLName, PSTR pszAPIName);
PARAMTYPE GetParameterEncoding(PSTR pszParam);
PSTR SkipWhitespace(PSTR pszInputLine);

extern HINSTANCE HInstance;
```

```
BOOL LoadAPIConfigFile(void)
{
    FILE *pFile;
    char szInput[256];
    BYTE params[33];
    BOOL fBuilding = FALSE;
    char szAPIFunctionFile[MAX_PATH];
    PSTR p;

    // Create a string with the path to the API function file. This
    // file will be in the same directory as this DLL.
    GetModuleFileName(HInstance, szAPIFunctionFile, sizeof(szAPIFunctionFile));
    p = strrchr(szAPIFunctionFile, '\\')+1;
    strcpy(p, "APISPY32.API");

    pFile = fopen(szAPIFunctionFile, "rt");
    if ( !pFile )
        return FALSE;

    //
    // Format of a line is moduleName:APIName
    // (e.g., "KERNEL32.DLL:LoadLibraryA")
    //
    while ( fgets(szInput, sizeof(szInput), pFile) )
    {
        PSTR pszNewline, pszInput;
        char szAPIName[128], szDLLName[128];

        pszInput = SkipWhitespace(szInput);

        if ( *pszInput == '\n' )    // Go to next line if this line is blank.
            continue;
        pszNewline = strrchr(pszInput, '\n');   // Look for the newline.
        if ( pszNewline )
            *pszNewline = 0;                    // Hack off the newline.

        if ( IsNewAPILine(pszInput) )
        {
            // Dispense with the old one we've been building.
            if ( fBuilding )
                AddAPIFunction(szDLLName, szAPIName, params);

            if ( ParseNewAPILine(pszInput, szDLLName, szAPIName) )
                fBuilding = TRUE;
            else
                fBuilding = FALSE;

            params[0] = 0;  // New set of parameters.
        }
        else    // A parameter line
        {
            BYTE param = (BYTE)GetParameterEncoding(pszInput);
            if ( param != PARAM_NONE )
            {
```

```
                params[ params[0] +1 ] = param; // Add param to end of list.
                params[0]++;                    // Update the param count.
            }
            else
            {
                if ( (*pszInput != 0) && (stricmp(pszInput, "VOID") != 0) )
                {
                    char errBuff[256];
                    wsprintf(errBuff, "Unknown param %s in %s\r\n",
                            pszInput, szAPIName);
                    OutputDebugString(errBuff);
                }
            }
        }
    }

    fclose( pFile );

    return TRUE;
}


// Returns TRUE if this line is the start of an API definition. It assumes
// that any whitespace has already been skipped over.
BOOL IsNewAPILine(PSTR pszInputLine)
{
    return 0 == strnicmp(pszInputLine, "API:", 4);
}

// Break apart a function definition line into a module name and a function
// name. Returns those strings in the passed PSTR buffers.
BOOL ParseNewAPILine(PSTR pszInput, PSTR pszDLLName, PSTR pszAPIName)
{
    PSTR pszColonSeparator;

    pszDLLName[0] = pszAPIName[0] = 0;

    pszInput += 4;  // Skip over "API:"

    pszColonSeparator = strchr(pszInput, ':');
    if ( !pszColonSeparator )
        return FALSE;

    *pszColonSeparator++ = 0;   // Null terminate module name, bump up
                                // pointer to API name.

    strcpy(pszDLLName, pszInput);
    strcpy(pszAPIName, pszColonSeparator);

    return TRUE;
}


typedef struct tagPARAM_ENCODING
{
```

```
    PSTR        pszName;    // Parameter name as it appears in APISPY32.API
    PARAMTYPE   value;      // Associated PARAM_xxx enum from PARMTYPE.H
} PARAM_ENCODING, * PPARAM_ENCODING;

PARAM_ENCODING ParamEncodings[] =
{
{"DWORD", PARAM_DWORD},
{"WORD", PARAM_WORD},
{"BYTE", PARAM_BYTE},
{"LPSTR", PARAM_LPSTR},
{"LPWSTR", PARAM_LPWSTR},
{"LPDATA", PARAM_LPDATA},
{"HANDLE", PARAM_HANDLE},
{"HWND", PARAM_HWND},
{"BOOL", PARAM_BOOL},
{"LPCODE", PARAM_LPCODE},
};
// Given a line that's possibly a parameter line, returns the PARAM_xxx
// encoding for that parameter type.  Lines that don't match any of the
// strings in the ParamEncodings cause the function to return PARAM_NONE.
PARAMTYPE GetParameterEncoding(PSTR pszParam)
{
    unsigned i;
    PPARAM_ENCODING pParamEncoding = ParamEncodings;

    for ( i=0; i < (sizeof(ParamEncodings)/sizeof(PARAM_ENCODING)); i++ )
    {
        if ( stricmp(pParamEncoding->pszName, pszParam) == 0 )
            return pParamEncoding->value;

        pParamEncoding++;
    }

    return PARAM_NONE;
}

// Given a pointer to an ASCIIZ string, return a pointer to the first
// non-whitespace character in the line.
PSTR SkipWhitespace(PSTR pszInputLine)
{
    while ( *pszInputLine && isspace(*pszInputLine) )
        pszInputLine++;
    return pszInputLine;
}
```

图 10-6　LOADAPIS.C 中 .API 文件分析处理的开始部分

INTRCPT.C 中的源程序包含了所有与拦截目标进程发出的函数调用相关的代码。INTRCPT.C 中的第一个例程是 AddAPIFunction。当 LOADAPIS.C 的代码读完一个函数的所有信息后，它将函数名、包含此函数的 DLL 文件名及 BYTE 编码的参数信息传递给 AddAPIFunction 函数。AddAPIFunction 做的两件事是产生函数拦截存根并将它加入存根序列中。AddAPIFunction 把产生存根的繁琐的工

作交给 BuildAPIStub 例程。

BuildAPIStub 先用函数及模块名来调用 GetProcAddress 来找出该函数的地址。假如 GetProcAddress 成功返回，BuildAPIStub 算出需要为该存根分配多少内存(API 名和参数编码长度不定)，并为它分配内存。接着 BuildAPIStub 填写第一部分已由在 INTRCPT2.H 中给出的 APIFunction 结构定义好的存根的那些域。在分配好的内存尾部，BuilAPIStub 拷贝一份函数名和 BYTE 编码的参数信息。

除了建立和维护用于函数拦截的存根外，INTRCPT.C 的另一重要任务是在目标进程的内存映象中搜索它的 JMP DWORD PTR [XXXXXXXX]并将它指向先前建立的存根。InterceptFunctionsInModule 只需要该模块在内存中的加载地址来找到我在前面部分中描述过的引入函数的表。这个函数首先通过在文件中寻找 DOS MZ 和 Win32 PE 符号来确定是否能找到一个有效的基地址。一旦它得知存在一个有效的基地址，它就用 IMAGE_NT_HEADERS 结构尾部的数据目录来获取一个指向该模块 .idata 部分的指针。

为了找到该 EXE 文件链入的函数，InterceptFunctionsInModule 解释在.idata 部分开始处的 IMAGE_IMPORT_DESCRIPTOR 结构数组。第八章中的 PEDUMP 做某些与此类似的工作。对就每个由该 EXE 文件隐式链接的 DLL 有一个 IMAGE_IMPORT_DESCRIPTOR。每个 IMAGE_IMPORT_DESCRIPTOR 包含一个与 IMAGE_THUNK_DATA 结构数组相关的位移，而每个 IMAGE_THUNK_DATA 对应一个链入的函数。

在一个两重循环中，InterceptFunctionsInModule 扫描该 EXE 文件链入的所有函数并找出它们的地址。这些地址作为 IMAGE_THUNK_DATA 结构的一部分被存储起来。我们的例程将每个函数的地址作为参数传递给 LookupInterceptedAPI。LookupInterceptedAPI 扫描我们建立的函数存根数组，寻找第一个 DWORD 与该地址相同的那个存根。如果找到，InterceptFunctionsInModule 用我们刚查到的存根代码的地址来改写在 IMAGE_THUNK_DATA 结构中的引入函数原来的地址(如图 10-7 所示)。从这时起，无论被侦探的进程何时调用链入的函数，它的 JMP DWORD PTR [XXXXXXXX]指令段将转到我们的拦截存根而不是它想转到的 API 函数。仅当我们的登记代码对该函数调用登记完毕后才把控制权交还给链入的函数。

```
PAPIFunction BuildAPIStub(PSTR pszModule, PSTR pszFuncName, PBYTE params);

// MakePtr is a macro that allows you to easily add to values (including
// pointers) together without dealing with C's pointer arithmetic.  It
// essentially treats the last two parameters as DWORDs. The first
// parameter is used to typecast the result to the appropriate pointer type.
#define MakePtr( cast, ptr, addValue ) (cast)( (DWORD)(ptr)+(DWORD)(addValue))

#define MAX_INTERCEPTED_APIS 2048
unsigned InterceptedAPICount = 0;
PAPIFunction InterceptedAPIArray[MAX_INTERCEPTED_APIS];
```

```
extern BOOL FChicago;
extern FILE * PLogFile;

BOOL AddAPIFunction
(
    PSTR pszModule,     // Exporting DLL name.
    PSTR pszFuncName,   // Exported function name.
    PBYTE params        // Opcode encoded parameters of exported function.
)
{
    PAPIFunction pNewFunction;

    if ( InterceptedAPICount >= MAX_INTERCEPTED_APIS )
        return FALSE;

    pNewFunction = BuildAPIStub(pszModule, pszFuncName, params);
    if ( !pNewFunction )
        return FALSE;

    InterceptedAPIArray[ InterceptedAPICount++ ] = pNewFunction;

    return TRUE;
}


PAPIFunction BuildAPIStub(PSTR pszModule, PSTR pszFuncName, PBYTE params)
{
    UINT allocSize;
    PAPIFunction pNewFunction;
    PVOID realProcAddress;
    UINT cbFuncName;
    HMODULE hModule;

    hModule = GetModuleHandle(pszModule);
    if ( !hModule )
        return 0;

    realProcAddress = GetProcAddress( hModule, pszFuncName );
    if ( !realProcAddress )
      return 0;

  cbFuncName = strlen(pszFuncName);
  allocSize = sizeof(APIFunction) + cbFuncName +1 + *params + 1;

  pNewFunction = malloc(allocSize);
  if ( !pNewFunction )
      return 0;

  pNewFunction->RealProcAddress = realProcAddress;
  pNewFunction->instr_pushad = 0x60;
  pNewFunction->instr_lea_eax_esp_plus_32 = 0x2024448D;
  pNewFunction->instr_push_eax = 0x50;
  pNewFunction->instr_push_offset_params = 0x68;
  pNewFunction->offset_params = (DWORD)(pNewFunction + 1) + cbFuncName + 1;
  pNewFunction->instr_push_offset_funcName = 0x68;
  pNewFunction->offset_funcName = (DWORD)(pNewFunction + 1);
  pNewFunction->instr_call_LogFunction = 0xE8;
  pNewFunction->offset_LogFunction
```

```
        = (DWORD)LogCall - (DWORD)&pNewFunction->instr_popad;
    pNewFunction->instr_popad = 0x61;
    pNewFunction->instr_jmp_dword_ptr_RealProcAddress = 0x25FF;
    pNewFunction->offset_dword_ptr_RealProcAddrss = (DWORD)pNewFunction;

    strcpy( (PSTR)pNewFunction->offset_funcName, pszFuncName );
    memcpy( (PVOID)pNewFunction->offset_params, params, *params+1 );

    return pNewFunction;
}


PAPIFunction LookupInterceptedAPI( PVOID address )
{
    unsigned i;
    PVOID stubAddress;

    for ( i=0; i < InterceptedAPICount; i++ )
    {
        if ( InterceptedAPIArray[i]->RealProcAddress == address )
            return InterceptedAPIArray[i];
    }

    // If it's Windows 95, and the app is being debugged (as this app is)
    // the loader doesn't fix up the calls to point directly at the
    // DLL's entry point. Instead, the address in the .idata section
    // points to a PUSH xxxxxxxx / JMP yyyyyyyy stub. The address in
    // xxxxxxxx points to another stub: PUSH aaaaaaaa / JMP bbbbbbbb.
    // The address in aaaaaaaa is the real address of the function in the
    // DLL. This ugly code verifies we're looking at this stub setup,
    // and if so, grabs the real DLL entry point, and scans through
    // the InterceptedAPIArray list of addresses again.
    // ***WARNING*** ***WARNING*** ***WARNING*** ***WARNING***
    // This code is subject to change!
    if ( FChicago )
    {

        if ( address < (PVOID)0x80000000 )  // Only shared, system DLLs
            return 0;                       // have stubs.

        if ( IsBadReadPtr(address, 9) || (*(PBYTE)address != 0x68)
             || (*((PBYTE)address+5) != 0xE9) )
            return 0;

        stubAddress = (PVOID) *(PDWORD)((PBYTE)address+1);

        for ( i=0; i < InterceptedAPICount; i++ )
        {
            PVOID lunacy;

            if ( InterceptedAPIArray[i]->RealProcAddress == stubAddress )
                return InterceptedAPIArray[i];

            lunacy = InterceptedAPIArray[i]->RealProcAddress;

            if ( !IsBadReadPtr(lunacy, 9) && (*(PBYTE)lunacy == 0x68)
                && (*((PBYTE)lunacy+5) == 0xE9) )
            {
```

```
                lunacy = (PVOID)*(PDWORD)((PBYTE)lunacy+1);
                if ( lunacy == stubAddress )
                    return InterceptedAPIArray[i];
            }
        }
    }

    return 0;
}

BOOL InterceptFunctionsInModule(PVOID baseAddress)
{
    PIMAGE_DOS_HEADER pDOSHeader = (PIMAGE_DOS_HEADER)baseAddress;
    PIMAGE_NT_HEADERS pNTHeader;
    PIMAGE_IMPORT_DESCRIPTOR pImportDesc;

    if ( IsBadReadPtr(baseAddress, sizeof(PIMAGE_NT_HEADERS)) )
        return FALSE;

    if ( pDOSHeader->e_magic != IMAGE_DOS_SIGNATURE )
        return FALSE;

    pNTHeader = MakePtr(PIMAGE_NT_HEADERS, pDOSHeader, pDOSHeader->e_lfanew);
    if ( pNTHeader->Signature != IMAGE_NT_SIGNATURE )
        return FALSE;
    pImportDesc = MakePtr(PIMAGE_IMPORT_DESCRIPTOR, baseAddress,
                          pNTHeader->OptionalHeader.
                          DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].
                          VirtualAddress);

    // Bail out if the RVA of the imports section is 0 (it doesn't exist)
    if ( pImportDesc == (PIMAGE_IMPORT_DESCRIPTOR)pNTHeader )
        return FALSE;

    while ( pImportDesc->Name )
    {
        PIMAGE_THUNK_DATA pThunk;

        pThunk = MakePtr(PIMAGE_THUNK_DATA,
                         baseAddress, pImportDesc->FirstThunk);

        while ( pThunk->u1.Function )
        {
            PAPIFunction pInterceptedFunction;

            pInterceptedFunction = LookupInterceptedAPI(pThunk->u1.Function);

            if ( pInterceptedFunction )
            {
                DWORD cBytesMoved;
                DWORD src = (DWORD)&pInterceptedFunction->instr_pushad;

                // Bash the import thunk.  We have to use WriteProcessMemory,
                // since the import table may be in a code section (courtesy
                // of the NT 3.51 team!).

                WriteProcessMemory( GetCurrentProcess(),
```

```
                                &pThunk->u1.Function,
                                &src, sizeof(DWORD), &cBytesMoved );
            }

            pThunk++;
        }

        pImportDesc++;
    }

    return TRUE;
```

图 10-7 INTRCPT.C 中建立存根和函数拦截的开始部分

LOG.C 文件包含所有将函数调用和返回值信息写入输出文件的相关代码。第一个例程，OpenLogFile，在被侦探程序所在的子目录中打开那个输出文件。输出文件的文件名与可执行文件的文件名相同，但扩展名是.OUT。以目标程序的 HMODULE 为参数调用 GetModuleFileName，我们可以很方便地得到该程序的目录和文件名，所以我们所要做的只是将扩展名 .EXE 换为 .OUT。OpenLogFile 没有考虑只读的存储介质，所以如果你是从 CD-ROM 上运行程序，fopen 函数就会失败，而你将不能得到一个跟踪记录文件。

LOG.C 中的 LogCall 例程是负责将函数调用的相关信息加入输出文件的高层函数。LogCall 由函数拦截存根调用，并需要一个指向函数名的指针，一个指向 BYTE 编码的参数信息的指针，以及一个指向在存根代码入口处的栈的指针作为参数。LogCall 做的第一件事就是将解释和格式化函数参数信息的繁琐工作交给 DecodeParamsToString 函数去做。

然后，LogCall 将函数和用()括起来的译好的参数信息写到跟踪记录文件中并换行。如果被拦截的函数嵌套于另一被拦截函数中，该行将按嵌套层数按比例缩进若干空格符。LogCall 最后的工作是调用 InterceptFunctionReturn，这个例程(在 RETURN.C 中)修改被拦截函数的返回地址以指向我们的返回值登记代码。

辅助函数 DecodeParamsToString 有三个参数：一个指向在函数拦截存根代码入口处的栈的指针，一个指向 BYTE 编码的参数信息的指针和一个指向存放格式化后字符串的缓冲区的指针。这个函数首先将框架指针上移 4 个 BYTE 以传递被拦截函数的返回地址。接着，一个 for 循环对所有 BYTE 编码的参数信息进行解释，从栈框架中抓出对应的 DWORD，并相应格式化参数。参数的一般形式为 <类型>:<值>，如 HWND:000200AC。如果参数是 LPSTR 类型的，这个函数调用辅助函数 GetLPSTR 获取这个参数指向的字符串的一小片(最多 10 个 BYTE)。如果所指向的是一个有效的字符串。该字符串作为附加的信息用一个 :(冒号)隔开放在后面，如下所示：

```
LPSTR:80B70018:"FreeCellIc"
```

DecodeParamsToString 一边格式化每个参数，一边将该字符串添加到传递给

这个函数的缓冲区的末尾。如果有多个参数，则用“,”(逗号)分隔，就象在真的 C/C++程序中一样。目的在于使参数信息尽可能有实际意义，例如：

```
LoadAcceleratorsA(HANDLE:00001C9E,LPSTR:80B70004:"FreeMenu")
```

与登记函数参数相对的是登记它的返回值，这将由 LogReturn 函数来执行。LogReturn 比 LogCall 要简单得多，仅由相应缩进该行和打印函数名及返回值的语句组成。

有些读者可能已注意到，在 LOG.C 中完全没有处理线程同步的代码。通常在一个有文件 I/O 操作的多线程程序中，你需要转折片或哑代码来防止线程在不适当的时机进行切换而产生的问题。LOG.C 无需处理线程同步是因为它不使用任何它修改的全局变量(无论是 PLogFile 指针还是 TlsIndex 变量在程序执行中都不改变)。但如果调用 fprintf()会产生什么样的结果呢？一个线程在它们之中切换不会产生问题吗？如果你不加注意，你会说会。然而，APISPY32 DLL 是用多线程运行库链接的(对 Visual C++来说则是 LIBCMT.LIB)。这些多线程库内部使用同步机制，所以使用这些函数不必处理同步问题。非常有趣的是，如果你通篇浏览 APISPY32 的代码，你不会找到任何同步处理的代码。这很大程度上是因为全局变量中在初始化时赋值，然后永不改变。图 10-8 给出了 LOG.C，它将函数名和参数写入输出文件。

```
// Helper function prototypes
void MakeIndentString(PSTR buffer, UINT level);
void DecodeParamsToString(PBYTE pParams, PDWORD pFrame, PSTR pszParams);
BOOL GetLPSTR( PSTR ptr, PSTR buffer );

FILE *PLogFile = 0;
extern DWORD TlsIndex;          // Defined in RETURN.C

BOOL OpenLogFile(void)
{
    char szFilename[MAX_PATH];
    PSTR pszExtension;

    GetModuleFileName( GetModuleHandle(0), szFilename, sizeof(szFilename) );
    pszExtension = strrchr(szFilename, '.');
    if ( !pszExtension )
        return FALSE;

    strcpy(pszExtension, ".out");

    PLogFile = fopen(szFilename, "wt");

    return (BOOL)PLogFile;
}


BOOL CloseLogFile(void)
```

```
{
    if ( PLogFile )
        fclose( PLogFile );
    return TRUE;
}


void _ _stdcall LogCall(PSTR pszName, PBYTE pParams, PDWORD pFrame)
{
    char szParams[512];
    char szIndent[128];
    PPER_THREAD_DATA pStack;

    if ( !PLogFile )
        return;

    DecodeParamsToString(pParams, pFrame, szParams);

    pStack = (PPER_THREAD_DATA)TlsGetValue(TlsIndex);
    if ( !pStack )
        return;

    MakeIndentString(szIndent, pStack->FunctionStackPtr);

    fprintf(PLogFile, "%s%s(%s)\n", szIndent, pszName, szParams);
    fflush(PLogFile);

    // Patch the return address of this function so that returns to us.
    InterceptFunctionReturn(pszName, pFrame);
}

void DecodeParamsToString(PBYTE pParams, PDWORD pFrame, PSTR pszParams)
{
    unsigned i;
    unsigned paramCount;
    unsigned paramShowSize;
    PSTR pszParamName;

    pszParams[0] = 0;   // Null out string in case there's no parameters.

    paramCount = *pParams++;    // Get number of parameters and advance
                                // to first encoded param.
    pFrame++;                   // Bump past the DWORD return address.

    for ( i=0; i < paramCount; i++ )
    {
        switch ( *pParams )
        {
            case PARAM_DWORD:
                pszParamName = "DWORD"; paramShowSize = 4; break;
            case PARAM_WORD:
                pszParamName = "WORD"; paramShowSize = 2; break;
            case PARAM_BYTE:
                pszParamName = "BYTE"; paramShowSize = 1; break;
            case PARAM_LPSTR:
```

```
                pszParamName = "LPSTR"; paramShowSize = 4; break;
            case PARAM_LPWSTR:
                pszParamName = "LPWSTR"; paramShowSize = 4; break;
            case PARAM_LPDATA:
                pszParamName = "LPDATA"; paramShowSize = 4; break;
            case PARAM_HANDLE:
                pszParamName = "HANDLE"; paramShowSize = 4; break;
            case PARAM_HWND:
                pszParamName = "HWND"; paramShowSize = 4; break;
            case PARAM_BOOL:
                pszParamName = "BOOL"; paramShowSize = 4; break;
            case PARAM_LPCODE:
                pszParamName = "LPCODE"; paramShowSize = 4; break;
            default:
                pszParamName = "<unknown>"; paramShowSize = 0;
        }

        pszParams += wsprintf(pszParams, "%s:", pszParamName);

        switch ( paramShowSize )
        {
            case 4: pszParamName = "%08X"; break;
            case 2: pszParamName = "%04X"; break;
            case 1: pszParamName = "%02X"; break;
        }
         pszParams += wsprintf(pszParams, pszParamName, *pFrame) ;

         // Tack on the string literal value if it's a PARAM_LPSTR
         if ( *pParams == PARAM_LPSTR )
         {
             char buffer[30];

             if ( GetLPSTR( (PSTR)*pFrame, buffer ) )
             {
                 strcpy(pszParams, buffer);
                 pszParams += strlen(buffer);
             }
         }

         if ( (paramCount - i) != 1 )    // Tack on a comma if not last
             *pszParams++ = ',';         // parameter.

         pFrame++;   // Bump frame up to the next DWORD value
         pParams++;  // advance to next encoded parameter.
    }               // End of for() statement.
}


BOOL GetLPSTR( PSTR ptr, PSTR buffer )
{
    PSTR p = buffer;
    int i;

    *p++ = ':';     // Write out initial -> :" <-
    *p++ = '\"';
```

```
    for ( i=0; i < 10; i++ )
    {
        if ( !IsBadReadPtr( ptr, 1 ) && *ptr )
        {
            *p = *ptr++;
            if ( *p == '\r' ) { *p++ = '\\'; *p = 'r'; }
            else if ( *p == '\n' ) { *p++ = '\\'; *p = 'n'; }
            else if ( *p == '\t' ) { *p++ = '\\'; *p = 't'; }

            p++;
        }
        else
            break;
    }
     if ( i == 0 )   // Not a valid string
         return FALSE;

     *p++ = '\"';    // Valid string ptr - end quote and null
     *p++ = 0;       // terminate the string

     return TRUE;
}

void LogReturn(PSTR pszName, DWORD returnValue, DWORD level)
{
    char szIndent[128];

    if ( !PLogFile )
        return;

    MakeIndentString(szIndent, level);
    fprintf(PLogFile, "%s%s returns: %X\n", szIndent, pszName, returnValue);
    fflush(PLogFile);
}


void MakeIndentString(PSTR buffer, UINT level)
{
    DWORD cBytes = level * 2;
    memset(buffer, ' ', cBytes);
    buffer[cBytes] = 0;
}
```

图 10-8　LOG.C 将 API 函数名和参数写入输出文件

RETURN.C 中的代码是关于拦截从 API 函数的调用返回以获取它的返回值的。头两个函数，InitThreadReturnStack 和 ShutdownThreadReturnStack，对目标进程的每个线程被调用两次。InitThreadReturnStack 申请一块 PER_HREAD_DATA 结构大小的内存并将其初始化(参见 PERTHRED.H)。PER_THREAD_DATA 结构中有对该线程返回地址栈极重要的两部分：一个 HOOKED_FUNCTION 结构数组和一个栈指针(见图 10-9)。

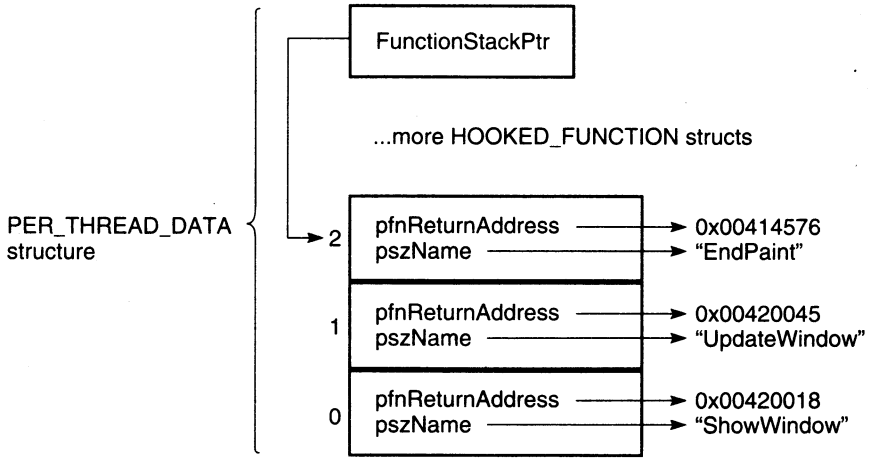


图 10-9　含指向存放线程返回地址栈的 HOOKED_FUNCTION 结构数组指针的 PER_THREAD_DATA 结构

每当一个被拦截的函数被调用时，它原来的返回地址和函数名指针被写入下一个可用的 HOOKED_FUNCTION 结构元素中。然后，栈指针加 1。这样运用栈使栈指针(其实是一个索引，不是指针)与当前嵌套层数相对应。登记例程利用这一点来根据当前嵌套层数适当地缩进调用及返回行。指向 PER_THREAD_DATA 结构的指针存储在 DLL 初始化时为它分配的线程局部存储区夹缝中。

InterceptFunctionReturn 由函数调用登记代码在将控制交还原来的 API 函数代码之前的那一刻调用。首先，InterceptFunctionReturn 将被拦截函数的返回地址和名字加入返回地址栈。然后它用 ASMRETRN. ASM 中 AsmCommonReturnPoint 例程的起始地址来改写该函数的返回地址。

RETURN. C 中最后一个函数是 CCommonReturnPoint，它被 AsmCommonReturnPoint 中的汇编代码调用。虽然我可以将这些都用汇编代码来做，但我想尽量用 C 来编写 APISPY32。CCommonReturnPoint 先调用 LogReturn 登记被拦截函数的返回值。然后它将原来的返回值写入一个特定的由汇编程序为回传返回值保存的栈中的内存区，并且返回汇编程序。图 10-10 给出了 RETURN. C，它处理登记函数返回值的具体细节。

```
void AsmCommonReturnPoint(void);

DWORD TlsIndex = 0xFFFFFFFF;

BOOL InitThreadReturnStack(void)
{
    PPER_THREAD_DATA pPerThreadData;

    static BOOL firstTime = TRUE;
```

```
    if ( firstTime )
    {
        TlsIndex = TlsAlloc();
        firstTime = FALSE;
    }

    if ( TlsIndex == 0xFFFFFFFF )
        return FALSE;

    pPerThreadData = malloc( sizeof(PER_THREAD_DATA) );
    if ( !pPerThreadData )
        return FALSE;

    pPerThreadData->FunctionStackPtr = 0;

    TlsSetValue(TlsIndex, pPerThreadData);

    return TRUE;
}

BOOL ShutdownThreadReturnStack(void)
{
    PPER_THREAD_DATA pPerThreadData;

    if ( TlsIndex == 0xFFFFFFFF )
        return FALSE;

    pPerThreadData = TlsGetValue( TlsIndex );
    if ( pPerThreadData )
        free( pPerThreadData );

    return TRUE;
}

BOOL InterceptFunctionReturn(PSTR pszName, PDWORD pFrame)
{
    PPER_THREAD_DATA pStack;
    DWORD i;

    pStack = (PPER_THREAD_DATA)TlsGetValue(TlsIndex);
    if ( !pStack )
        return FALSE;
    if ( pStack->FunctionStackPtr >= (MAX_HOOKED_FUNCTIONS-1) )
        return FALSE;

    i = pStack->FunctionStackPtr;

    pStack->FunctionStack[i].pfnReturnAddress = (PVOID)pFrame[0];
    pStack->FunctionStack[i].pszName = pszName;
    pStack->FunctionStackPtr++;

    pFrame[0] = (DWORD)AsmCommonReturnPoint;

    return TRUE;
}

// return_address <- pFrame[8]
// EAX            <- pFrame[7]
// ECX            <- pFrame[6]
// EDX            <- pFrame[5]
```

```
    EBX             <- pFrame[4]
// ESP              <- pFrame[3]
// EBP              <- pFrame[2]
// ESI              <- pFrame[1]
// EDI              <- pFrame[0]

//
// Common return point for all functions that we've intercepted.
// Called by _AsmCommonReturnPoint in ASMRETRN.ASM
// pFrame is a pointer to the stack frame set up by the PUSHAD
// (see above comment for the layout of this frame)
//
void CCommonReturnPoint( PDWORD pFrame )
{
    PPER_THREAD_DATA pStack;
    DWORD i;

    // Get the function stack for the current thread
    pStack = (PPER_THREAD_DATA)TlsGetValue(TlsIndex);
    if ( !pStack )
        return;

    i = -pStack->FunctionStackPtr;

    // Emit the information about the function return value to the logging
    // mechanism.
    LogReturn(pStack->FunctionStack[i].pszName, pFrame[7], i);

    // Patch the return address back to what it was when the function
    // was originally called.
    pFrame[8] = (DWORD)pStack->FunctionStack[i].pfnReturnAddress;
}
```

图 10-10　RETURN.C 登记函数返回值

在我动手编写 APISPY32 时，我想全用 C。不巧，为了在被拦截的函数返回时对我们玩的堆栈游戏进行很好的控制，我无法编写一个清晰的 C 程序。C 例程也可以存取寄存器，我想使 APISPY32 代码对目标进程的影响尽可能小，所以我选择了在调用 C 例程之前将所有通用寄存器压栈，之后将它们弹出以恢复环境。ASMRETRN.ASM 的代码做了最小的一部分工作。它首先将 ESP 寄存器的值减 4 以保持用于存放原来的返回地址的空间。最终 C 代码将那个 DWORD 填上正确的地址，这样汇编代码返回时就会转到正确的地址。而且栈指针也和该例程进入时相同。代码的其余部分，如图 10-11 所示，仅仅是在调用 CCommonReturnPoint 函数之前的一条 PUSHAD 指令和调用之后的一条 POPAD 指令。

Win32s 特有的代码

APISPY32.DLL 中最后剩下一点是关于 Win32s 的，其源程序在 W32SSUPP.C 中。这个模块中有一个函数，GetModuleBaseFromWin32sHMod，在图 10-12 中给出。这个函数的任务是取一个 Win32s 的 hModule(它并不是内存中模块的基地址)，并将它转换成一个基地址。翻完 Win32s 的文档，我都未能

找到一个简洁的(即使是公布的)方法来做这个转换。不过,我确实知道象 Win32 GetProAddress 这样的函数也要做类似的工作。一步步地在 SoftIce/W 中的 Win32s 库中找出两个现有的(但未公布的)函数在 W32SKRNL. DLL 中,它们正好是我需要的。第一个是_ImteFromHModule@4,它取走一个 Win32s 的 HMODULE 并返回一个内部句柄,称为 IMTE。(IMTE 在第三章中描述过。)第二个是 BaseAddressFromImte,它用一个 IMTE 作为一个参数并返回一个 32 位的线性基地址,该模块就在此处被装载。

```
.386
.model flat

extrn _CCommonReturnPoint:proc

.code

public _AsmCommonReturnPoint

_AsmCommonReturnPoint proc
    SUB     ESP,4   ; Make space for return address
    PUSHAD
    MOV     EAX,ESP
    PUSH    EAX
    CALL    _CCommonReturnPoint
    ADD     ESP,4
    POPAD
    RET
_AsmCommonReturnPoint endp

END
```

图 10-11　函数返回捕获代码在 ASMRETRN 中的汇编代码部分

由于这些函数是具体针对 Win32s 的,我不能直接从 APISPY32. DLL 中调用它们,因为如果这样,这个 DLL 就不能在 Windows 95 或 Windows NT 环境中装载。直接调用这些函数就是将它们固定在这个 DLL 中,这样当装载程序把 APISPY32. DLL 装载到内存中时,它无法找到它们。因此,我用调用 GetProAddress 来获取指向这两个函数的指针的标准技术,然后通过指针来调用它们。

APISPYLD 的代码

有了支撑我们的 API 侦探 DLL 的代码后,剩下的问题就是程序装载器了。它本身是用 CreateProcess 来启动要侦探的进程的一个程序。在目标进程实际运行前,程序装载器将侦探 DLL(APISPY32. DLL)接种到目标进程中。接种完毕后,程序装载器将在一个 WaitForDebugEvent 循环中打转直到目标进程终止,除此之外什么也不做。我将这个程序加载器命名为 APISPYLD。它的源文件比工程中别的

源文件都大，所以我把 APISPYLD.C 分成几小块来说明。

```
typedef DWORD (_ _stdcall *XPROC)(DWORD);

DWORD GetModuleBaseFromWin32sHMod(HMODULE hMod)
{
    XPROC ImteFromHModule, BaseAddrFromImte;
    HMODULE hModule;
    DWORD imte;

    hModule = GetModuleHandle("W32SKRNL.DLL");
    if( !hModule )
        return 0;

    ImteFromHModule = (XPROC)GetProcAddress(hModule, "_ImteFromHModule@4");
    if ( !ImteFromHModule )
        return 0;
    BaseAddrFromImte = (XPROC)GetProcAddress(hModule, "_BaseAddrFromImte@4");
    if ( !BaseAddrFromImte )
        return 0;

    imte = ImteFromHModule( (DWORD)hMod);
    if ( !imte )
        return 0;

    return BaseAddrFromImte(imte);
}
```

图 10-12　W32SSUPP.C 是 Win32s 特有的代码

APISPYLD.C 的第一部分是由一个极小化的用户界面的代码和装载目标进程的代码组成。WinMain 是一个简单的 while 循环，它循环到一个程序成功地被侦探或用户选择退出侦探程序为止。这个 while 循环首先打开一个对话框来获取要侦探的程序名。如果 DialogBox 返回非零，WinMain 调用 LoadProcessForSpying。如果成功地启动了该进程，WinMain 调用 DebugLoop，APISPYLD 剩下的部分就在其中，来从所有的调试信息中抽取信息直到目标进程终止。

如你在图 10-13 中所见，APISPYLD 的对话框是极小化的。单行编辑框用于键入待侦探程序的命令行(包括参数)。为方便起见，File... 按钮打开公用的 GetOpenFileName 对话框，使你能通过逐步地查找来找到文件名，而不用手工键入冗长的路径。单击 Run 按钮使对话框消失并试图装载选定的进程来进行侦探。单击 Cancel 按钮使对话框消失并退出程序。

APISPY32DlgProc 中的对话过程代码极为简单，只与三种消息有关：WM_INITDIALOG、WM_COMMAND 和 WM_CLOSE。WM_INITDIALOG 处理程序使我们可以找出我们上次给 APISPYLD 的命令行，并用它填充编辑区。上一个命令存放在一个名为 APISPY32.INI 的私有 .INI 文件中。

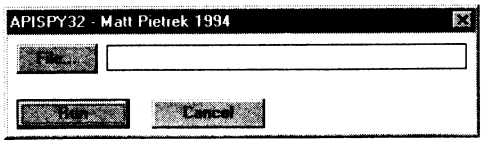


图 10-13　APISPYLD 对话框供你指定要侦探的程序

对话代码的主要内容用于处理 WM_COMMAND 消息；它位于 Handle_WM_COMMAND 帮助函数中。只有三个按钮产生的 WM_COMMAND 消息才会被处理。File... 按钮的代码调用 GetProgramName 函数，它是内含公用的对话框函数 GetOpenFileName 函数的一个封套。如果 GetProgramName 成功，程序名将出现在对话框中的编辑区内。单击 Run 按钮就是告诉 APISPYLD 将编辑区的内容拷贝到全程变量 SzCmdLine，退出对话框并返回代码 1。单击 Cancel 也会退出对话框，但告诉它返回 0，这样 WinMain 就不会试图去装载程序。

如果用户正确地输入一个有效的命令行并选择了 Run 按钮，对话框消失且控制返回 WinMain。然后 WinMain 调用 LoadProcessForSpying，并将全程变量 SzCmdLine 的值传递给它。LoadProcessForSpying 只是 Win32s 的 CreateProcess API 函数的一个外壳。这个特定的 CreateProcess 调用只对 fdwCreate 参数说明标志 DEBUG_ONLY_THIS_PROCESS 感兴趣。这告诉操作系统我们的程序(APISPYLD)要以调试程序的身份来调试正被装载的程序。它还告诉操作系统我只对这个特定进程的调试事件感兴趣，对进程的任何子进程的调试事件都不关心。如果前面我用 DEBUG_PROCESS 代替 DEBUG_ONLY_THIS_PROCESS，APISPYLD 将还会得到该目标进程的所有子进程的调试事件报告。图 10-14 给出了 APISPYLD.C 的开始部分，用户界面和进程装载过程。

```
char SzINISection[] = "Options";
char SzINICmdLineKey[] = "CommandLine";
char SzINIFile[] = "APISPY32.INI";
char SzCmdLine[MAX_PATH];

BOOL FFirstBreakpointHit = FALSE, FSecondBreakpointHit = FALSE;

PROCESS_INFORMATION ProcessInformation;
CREATE_PROCESS_DEBUG_INFO ProcessDebugInfo;

CONTEXT OriginalThreadContext, FakeLoadLibraryContext;
PVOID PInjectionPage;

#define PAGE_SIZE 4096
BYTE OriginalCodePage[PAGE_SIZE];
BYTE NewCodePage[PAGE_SIZE];
```

```
//======================== Code ==========================================

//
// Function prototypes
//
BOOL  CALLBACK APISPY32DlgProc(HWND, UINT, WPARAM, LPARAM);
void  Handle_WM_COMMAND(HWND hWndDlg, WPARAM wParam, LPARAM lParam);
void  Handle_WM_INITDIALOG(HWND hWndDlg, WPARAM wParam, LPARAM lParam);
BOOL  GetProgramName(HWND hWndOwner, PSTR szFile, unsigned nFileBuffSize);
BOOL  LoadProcessForSpying(PSTR SzCmdLine);
void  DebugLoop(void);
DWORD HandleDebugEvent( DEBUG_EVENT * event );
void  HandleException(LPDEBUG_EVENT lpEvent, PDWORD continueStatus);
void  EmptyMsgQueueOfUselessMessages(void);
BOOL  InjectSpyDll(void);
BOOL  ReplaceOriginalPagesAndContext(void);
PVOID FindUsablePage(HANDLE hProcess, PVOID PProcessBase);
BOOL  GetSpyDllName(PSTR buffer, UINT cBytes);


int APIENTRY WinMain( HANDLE hInstance, HANDLE hPrevInstance,
                      LPSTR lpszCmdLine, int nCmdShow )
{
     // This dialog returns 0 if the user pressed cancel
     while ( 0 != DialogBox(hInstance, "APISPY32_LOAD_DLG", 0,
                            (DLGPROC)APISPY32DlgProc) )
     {
         if ( LoadProcessForSpying(SzCmdLine) )
         {
             DebugLoop();
             break;
         }

         MessageBox(0, "Unable to start program", 0, MB_OK);
     }

     return 0;
}

BOOL CALLBACK APISPY32DlgProc(HWND hWndDlg, UINT msg,
                              WPARAM wParam, LPARAM lParam)
{
    switch ( msg )
    {
        case WM_COMMAND:
            Handle_WM_COMMAND(hWndDlg, wParam, lParam);
            return TRUE;
        case WM_INITDIALOG:
            Handle_WM_INITDIALOG(hWndDlg, wParam, lParam);
            return TRUE;
        case WM_CLOSE:
            EndDialog(hWndDlg, 0);
            return FALSE;
    }
```

```
    return FALSE;
}

void Handle_WM_COMMAND(HWND hWndDlg, WPARAM wParam, LPARAM lParam)
{
    if ( wParam == IDC_RUN )
    {
        if ( GetWindowText( GetDlgItem(hWndDlg, IDC_CMDLINE),
                            SzCmdLine, sizeof(SzCmdLine)) )
        {
            WritePrivateProfileString(SzINISection, SzINICmdLineKey,
                                      SzCmdLine, SzINIFile);
            EndDialog(hWndDlg, 1);  // Return TRUE
        }
        else
        {
            MessageBox( hWndDlg, "No program selected", 0, MB_OK);
        }
    }
    else if ( wParam == IDC_FILE )
    {
        if ( GetProgramName(hWndDlg, SzCmdLine, sizeof(SzCmdLine)) )
            SetWindowText( GetDlgItem(hWndDlg, IDC_CMDLINE), SzCmdLine );
    }
    else if ( wParam == IDCANCEL )
    {
        EndDialog(hWndDlg, 0);
    }
}

void Handle_WM_INITDIALOG(HWND hWndDlg, WPARAM wParam, LPARAM lParam)
{
    GetPrivateProfileString(SzINISection, SzINICmdLineKey, "", SzCmdLine,
                            sizeof(SzCmdLine), SzINIFile);
    SetWindowText( GetDlgItem(hWndDlg, IDC_CMDLINE), SzCmdLine );
}

static char szFilter1[] = "Programs (*.EXE)\0*.EXE\0";

BOOL GetProgramName(HWND hWndOwner, PSTR szFile, unsigned nFileBuffSize)
{
    OPENFILENAME ofn;

    szFile[0] = 0;

    memset(&ofn, 0, sizeof(OPENFILENAME));

    ofn.lStructSize = sizeof(OPENFILENAME);
    ofn.hwndOwner = hWndOwner;
    ofn.lpstrFilter = szFilter1;
    ofn.nFilterIndex = 1;
    ofn.lpstrFile= szFile;
    ofn.nMaxFile = nFileBuffSize;
    ofn.lpstrFileTitle = 0;
    ofn.nMaxFileTitle = 0;
```

```
    ofn.lpstrInitialDir = 0;
    ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;

    return GetOpenFileName(&ofn);
}
BOOL LoadProcessForSpying(PSTR SzCmdLine)
{
    STARTUPINFO startupInfo;

    memset(&startupInfo, 0, sizeof(startupInfo));
    startupInfo.cb = sizeof(startupInfo);

    return CreateProcess(
                0,                          // lpszImageName
                SzCmdLine,                  // lpszCommandLine
                0,                          // lpsaProcess
                0,                          // lpsaThread
                FALSE,                      // fInheritHandles
                DEBUG_ONLY_THIS_PROCESS,    // fdwCreate
                0,                          // lpvEnvironment
                0,                          // lpszCurDir
                &startupInfo,               // lpsiStartupInfo
                &ProcessInformation         // lppiProcInfo
                );
}
```

图 10-14　APISPYLD.C 的开始部分：用户界面和进程装载函数

APISPYLD.C 的中间部分是一个调试循环：一个调用 WaitForDebugEvent 和 ContinueDebugEvent 直到我们侦探的进程结束的循环。每次 WaitForDebugEvent 返回意味着产生了新的调试事件 XXX_DEBUG_EVENT（如 EXCEPTION_DEBUG_EVENT 或 CREATE_THREAD_DEBUG_EVENT）。DebugLoop 将每个调试事件传递给 HandleDebugEvent 帮助函数，让它做必要的处理。大部分情况下，HandleDebugEvent 中的代码忽略大多数事件而将 DEBUG_CONTINUE 传递给 ContinueDebugEvent。然而，两个发生在目标进程生存期间的 EXCEPTION_DEBUG_EVENT 是我们的程序装载器所感兴趣的。正因如此，我将异常处理单独分出来交给另一个帮助函数——HandleException。

我们的程序装载器应该看到的第一个 EXCEPTION_DEBUG_EVENT 是断点异常——EXCEPTION_BREAKPOINT（在 WINBASE.H 中）。这个断点并不在目标进程的代码中。事实上，在操作系统代码中有一条内嵌的 INT 3 指令，它正好在新进程的第一条指令执行之前执行。我们的 HandleException 例程显式地寻找第一个断点异常；当它看到该异常时，它将侦探 DLL 接种到目标进程的地址空间中（用在这部分中稍后介绍的 InjectSpyDll）。

HandleException 寻找的第二个异常是 InjectSpyDll 接种到目标进程代码中以使我们在 APISPY32.DLL 装载后获取控制的断点。当这个断点发生时，APISPYLD 就知道目标进程已经完成了对 APISPY32.DLL 的装载。被 InjectSpyDll 修改的原来的内存页面和线程环境需恢复到第一个断点发生时的状态。Han-

dleException 调用 ReplaceOriginalPagesAndContext 函数来执行这一任务。

APISPYLD 在这一部分的最后一点代码是专为 Win32s 写的。早些时候我曾说过在 WaitForDebugEvent 返回时，有一个新的调试事件等待处理。在 Win32s 中这不总是对的。实际上，Win32s 的 WaitForDebugEvent 函数在有另一条消息等待处理时返回 TRUE，否则返回 FALSE。在 Win32s 中另一未公布的怪事是系统将窗口消息和一 NULL 窗口控制柄传给调试程序(APISPYLD)的消息队列。传给该队列的窗口消息具有由 Win32s 用如下调用获得的消息号：

```
RegisterWindowMessage("W32S_Debug_Msg");
```

如果你不把这些消息从你的队列中清出去，你的消息队列将被塞满而真正的窗口消息反而不能进入该队列。为了处理 Win32s 这两种奇怪的行为，我们的调试循环当程序在 Win32s 中运行且 WaitForDebugEvent 返回 FALSE 时调用 EmptyMsgQueueOfUselessMessage。EmptyMsgQueueOfUselessMessage 是一个简单的例程，它调用 PeekMessage(PM_REMOVE)直到 PeekMessage 返回 FALSE。任何 HWND 为非零的消息被传给 DispatchMessage——但到现在为止，我尚未见过从这个例程传回 HWND 有效的消息。将这些消息从队列中消除后，DebugLoop 再次调用 WaitForDebugEvent。这次 WaitForDebugEvent 停住，直到确实有一个等待处理的调试事件。图 10-15 给出了 APISPYLD.C 中调试循环和调试事件处理的开始部分。

```
void DebugLoop(void)
{
    DEBUG_EVENT event;
    DWORD continueStatus;
    BOOL fWin32s;
    BOOL fWaitResult;

    fWin32s = (GetVersion() & 0xC0000000) == 0x80000000;

    while ( 1 )
    {
        fWaitResult = WaitForDebugEvent(&event, INFINITE);

        if ( (fWaitResult == FALSE) && fWin32s )
        {
            EmptyMsgQueueOfUselessMessages();
            continue;
        }

        continueStatus = HandleDebugEvent( &event );

        if ( event.dwDebugEventCode == EXIT_PROCESS_DEBUG_EVENT )
            return;

        ContinueDebugEvent( event.dwProcessId,
                            event.dwThreadId,
                            continueStatus );
    }
```

```
}

PSTR SzDebugEventTypes[] =
{
"",
"EXCEPTION",
"CREATE_THREAD",
"CREATE_PROCESS",
"EXIT_THREAD",
"EXIT_PROCESS",
"LOAD_DLL",
"UNLOAD_DLL",
"OUTPUT_DEBUG_STRING",
"RIP",
};

DWORD HandleDebugEvent( DEBUG_EVENT * event )
{
    DWORD continueStatus = DBG_CONTINUE;
    // char buffer[1024];

    // wsprintf(buffer, "Event: %s\r\n",
    //          SzDebugEventTypes[event->dwDebugEventCode]);
    // OutputDebugString(buffer);


    if ( event->dwDebugEventCode == CREATE_PROCESS_DEBUG_EVENT )
    {
        ProcessDebugInfo = event->u.CreateProcessInfo;
    }
    else if ( event->dwDebugEventCode == EXCEPTION_DEBUG_EVENT )
    {

        HandleException(event, &continueStatus);
    }

    return continueStatus;
}

void HandleException(LPDEBUG_EVENT lpEvent, PDWORD continueStatus)
{
    // char buffer[128];
    // wsprintf(buffer, "Exception code: %X  Addr: %08X\r\n",
    //          lpEvent->u.Exception.ExceptionRecord.ExceptionCode,
    //          lpEvent->u.Exception.ExceptionRecord.ExceptionAddress);
    // OutputDebugString(buffer);


    if ( lpEvent->u.Exception.ExceptionRecord.ExceptionCode
            == EXCEPTION_BREAKPOINT )
    {
        if ( FFirstBreakpointHit == FALSE )
        {
            InjectSpyDll();
            FFirstBreakpointHit = TRUE;
        }
        else if ( FSecondBreakpointHit == FALSE )
        {
            ReplaceOriginalPagesAndContext();
            FSecondBreakpointHit = TRUE;
        }
```

```
        *continueStatus = DBG_CONTINUE;
    }
    else
    {
        *continueStatus = DBG_EXCEPTION_NOT_HANDLED;
    }
}

void EmptyMsgQueueOfUselessMessages(void)
{
    MSG msg;        // See PeekMessage loop for explanation of idiocy.

    // Win32s idiocy puts W32s_Debug_Msg message in our message queue.
    // Dispose of them! They're useless!
    while ( PeekMessage(&msg, 0, 0, 0, PM_REMOVE) )
    {
        if ( msg.hwnd )
            DispatchMessage(&msg);
    }
}
```

图 10-15 APISPYLD.C 中调试循环和调试事件处理的开始部分

我们的 APISPYLD 代码的最后一部分用于在它遇到第一个断点后将 APISPY32.DLL 接种到被侦探进程的地址空间中。InjectSpyDll 是一个复杂的例程，可大致分为三个部分。InjectSpyDll 的第一部分用于找到目标进程的入口地址。这一阶段主要关注目标进程中不在 .idata 部分的第一个可写的数据页面。KERNEL32.DLL 中 LoadLibrary 例程的地址，要装载的 DLL 文件名(APISPY32.DLL)，及保存进程的初始线程的事前线程环境也都很重要。

在它的第二部分中，InjectSpyDll 将第一个可写页的内容拷贝到一个名为 OriginalCodePage 的全程变量中。(顺便说一下，这个变量的命名很糟糕。APISPY32 的早些版本用第一个代码页(而不是第一个可写数据页)来存放这个变量的名称。)注意在这里为了给我们将修改的内存页面做一个备份，有必要调用 ReadProcessMemory。这一点非常重要。我们要备份的页面在另一进程中，装载程序不能直接存取那片内存。

InjectSpyDll 的第三部分设置环境，这样当目标进程恢复执行时就会调用 LoadLibrary，通知操作系统装载 APISPY32.DLL。调用 LoadLibrary 的代码是通过填写 FAKE_LOADLIBRARY_CODE 结构来构造的。这个结构的每个域不是一条汇编指令就是前一条指令的操作数。在这个结构的尾部是侦探 DLL 文件名的一份拷贝。我将 DLL 的名字放在这个结构中是因为这个 DLL 的名字需要在目标进程的环境中可见，而不是对 APISPYLD 程序可见。填好这个结构后，我用 WriteProcessMemory 来将这个结构拷贝到目标进程的相应页面中。之后，InjectSpyDll 函数用 SetThreadContext 来改变 EIP 寄存器的值，目标进程的线程在恢复运行时要用到它。特别地，当它出现在目标进程的地址空间中时，EIP 寄存器将被设置成该代码片段第一条指令的地址。

假如 InjectSpyDll 正确地工作，目标进程将在恢复运行时执行 LoadLibrary 的

代码。在 LoadLibrary 返回时，CPU 将在调用 LoadLibrary 后暂停在断点指令上，这使目标进程再度被冻结，且使 APISPYLD 进程中的 WaitForDebugEvent 调用返回一个 EXCEPTION_DEBUG_EVENT。HandleException 函数一看见这个特殊的异常，就知道这时该恢复我们先前修改的页面及线程环境。恢复线程原来状态的代码在 ReplaceOriginalPagesAndContext 帮助函数中。在这个例程中，我们用 WriteProcessMemory 来写回被修改的页面，然后用 SetThreadContext 将我们接种该 DLL 前保存到一旁的线程环境传递回去。图 10-16 给出了 APISPYLD.C 中 InjectSpyDll 部分。

```
#pragma pack ( 1 )
typedef struct
{
    WORD    instr_SUB;
    DWORD   operand_SUB_value;
    BYTE    instr_PUSH;
    DWORD   operand_PUSH_value;
    BYTE    instr_CALL;
    DWORD   operand_CALL_offset;
    BYTE    instr_INT_3;
    char    data_DllName[1];
} FAKE_LOADLIBRARY_CODE, * PFAKE_LOADLIBRARY_CODE;

BOOL InjectSpyDll(void)
{
    BOOL retCode;
    DWORD cBytesMoved;
    char szSpyDllName[MAX_PATH];
    FARPROC pfnLoadLibrary;
    PFAKE_LOADLIBRARY_CODE pNewCode;

    // ====================================================================
    // Phase 1 - Locating addresses of important things
    // ====================================================================

    pfnLoadLibrary = GetProcAddress( GetModuleHandle("KERNEL32.DLL"),
                                     "LoadLibraryA" );
    if ( !pfnLoadLibrary )
        return FALSE;

    PInjectionPage = FindUsablePage(ProcessInformation.hProcess,
                                    ProcessDebugInfo.lpBaseOfImage);
    if ( !PInjectionPage )
        return FALSE;

    if ( !GetSpyDllName(szSpyDllName, sizeof(szSpyDllName)) )
        return FALSE;

    OriginalThreadContext.ContextFlags = CONTEXT_CONTROL;
    if ( !GetThreadContext(ProcessInformation.hThread,&OriginalThreadContext))
        return FALSE;
```

```
    // ====================================================================
    // Phase 2 - Saving the original code page away
    // ====================================================================

    // Save off the original code page
    retCode = ReadProcessMemory(ProcessInformation.hProcess, PInjectionPage,
                                OriginalCodePage, sizeof(OriginalCodePage),
                                &cBytesMoved);
    if ( !retCode || (cBytesMoved != sizeof(OriginalCodePage)) )
        return FALSE;

    // ====================================================================
    // Phase 3 - Writing new code page and changing the thread context
    // ====================================================================

    pNewCode = (PFAKE_LOADLIBRARY_CODE)NewCodePage;

    pNewCode->instr_SUB = 0xEC81;
    pNewCode->operand_SUB_value = 0x1000;

    pNewCode->instr_PUSH = 0x68;
    pNewCode->operand_PUSH_value = (DWORD)PInjectionPage
                          + offsetof(FAKE_LOADLIBRARY_CODE, data_DllName);

    pNewCode->instr_CALL = 0xE8;
    pNewCode->operand_CALL_offset =
            (DWORD)pfnLoadLibrary - (DWORD)PInjectionPage
            - offsetof(FAKE_LOADLIBRARY_CODE,instr_CALL) - 5;

    pNewCode->instr_INT_3 = 0xCC;

    lstrcpy(pNewCode->data_DllName, szSpyDllName); // Copy DLL name.

    // Write out the new code page
    retCode = WriteProcessMemory(ProcessInformation.hProcess, PInjectionPage,
                                &NewCodePage, sizeof(NewCodePage),
                                &cBytesMoved);
    if ( !retCode || (cBytesMoved != sizeof(NewCodePage)) )
        return FALSE;

    FakeLoadLibraryContext = OriginalThreadContext;
    FakeLoadLibraryContext.Eip = (DWORD)PInjectionPage;

    if ( !SetThreadContext(ProcessInformation.hThread,
                          &FakeLoadLibraryContext) )
        return FALSE;

    return TRUE;
}
BOOL ReplaceOriginalPagesAndContext(void)
{
    BOOL retCode;
    DWORD cBytesMoved;
```

```
    retCode = WriteProcessMemory(ProcessInformation.hProcess, PInjectionPage,
                                 OriginalCodePage, sizeof(OriginalCodePage),
                                 &cBytesMoved);
    if ( !retCode || (cBytesMoved != sizeof(OriginalCodePage)) )
        return FALSE;

    if ( !SetThreadContext(ProcessInformation.hThread,
                           &OriginalThreadContext) )
        return FALSE;

    return TRUE;
}

PVOID FindUsablePage(HANDLE hProcess, PVOID PProcessBase)
{
    DWORD peHdrOffset;
    DWORD cBytesMoved:
    IMAGE_NT_HEADERS ntHdr;
    PIMAGE_SECTION_HEADER pSection;
    unsigned i;

    // Read in the offset of the PE header within the debuggee
    if ( !ReadProcessMemory(ProcessInformation.hProcess,
                            (PBYTE)PProcessBase + 0x3C,
                            &peHdrOffset,
                            sizeof(peHdrOffset),
                            &cBytesMoved) )
        return FALSE;


    // Read in the IMAGE_NT_HEADERS.OptionalHeader.BaseOfCode field
    if ( !ReadProcessMemory(ProcessInformation.hProcess,
                            (PBYTE)PProcessBase + peHdrOffset,
                            &ntHdr, sizeof(ntHdr), &cBytesMoved) )
        return FALSE;

    pSection = (PIMAGE_SECTION_HEADER)
               ((PBYTE)PProcessBase + peHdrOffset + 4
               + sizeof(ntHdr.FileHeader)
               + ntHdr.FileHeader.SizeOfOptionalHeader);

    for ( i=0; i < ntHdr.FileHeader.NumberOfSections; i++ )
    {
```

```
        IMAGE_SECTION_HEADER section;

        if ( !ReadProcessMemory( ProcessInformation.hProcess,
                                 pSection, &section, sizeof(section),
                                 &cBytesMoved) )
            return FALSE;

        // OutputDebugString( "trying section: " );
        // OutputDebugString( section.Name );
        // OutputDebugString( "\r\n" );

        // If it's writeable, and not the .idata section, we'll go with it
        if ( (section.Characteristics & IMAGE_SCN_MEM_WRITE)
             && strncmp(section.Name, ".idata", 6) )
        {
            // OutputDebugString( "using section: " );
            // OutputDebugString( section.Name );
            // OutputDebugString( "\r\n" );

            return (PVOID) ((DWORD)PProcessBase + section.VirtualAddress);
        }

        pSection++; // Not this section. Advance to next section.
    }

    return 0;
}

BOOL GetSpyDllName(PSTR buffer, UINT cBytes)
{
    char szBuffer[MAX_PATH];
    PSTR pszFilename;

    // Get the complete path to this EXE - The spy dll should be in the
    // same directory.
    GetModuleFileName(0, szBuffer, sizeof(szBuffer));

    pszFilename = strrchr(szBuffer, '\\');
    if ( !pszFilename )
        return FALSE;

    lstrcpy(pszFilename+1, "APISPY32.DLL");
    strncpy(buffer, szBuffer, cBytes);
    return TRUE;
}
```

图 10-16　APISPYLD.C 中 DLL 接种例程

使用 APISPY32 时的注意事项

如果要用 APISPY32 侦探一个程序，就要先运行 APISPYLD 程序。在编辑区中键入一个命令行，或用 File... 按钮来寻找想要的可执行文件。找到后，单击 Run 按钮。APISPYLD 对话框将会消失，选定的程序将会开始运行。在你的目标程序运行完毕后，在可执行文件所在的子目录中，将形成一个以 .OUT 为扩展名的同名输出文件。图 10-17 给出了在 Win32CLOCK 程序上运行 APISPY32 得到的输出文件的一部分。

```
KillTimer(HWND:000026F4,DWORD:00000001)
KillTimer returns: 1
SetTimer(HWND:000026F4,DWORD:00000001,DWORD:000001C2,LPDATA:00000000)
SetTimer returns: 1
CheckMenuItem(HANDLE:00001EF0,DWORD:00000008,DWORD:00000008)
CheckMenuItem returns: 0
wsprintfA(LPSTR:80E3AD68,LPSTR:80DEE190:"%s - %s")
wsprintfA returns: F
SetWindowTextA(HWND:000026F4,LPSTR:80E3AD68:"Clock - 4/")

    DefWindowProcA(HWND:000026F4,DWORD:0000000C,DWORD:00000000,DWORD:80E3AD68)
    DefWindowProcA returns: 0
SetWindowTextA returns: 1
GetSystemMenu(HWND:000026F4,BOOL:00000000)
GetSystemMenu returns: 1F68
AppendMenuA(HANDLE:00001F68,DWORD:00000800,DWORD:00000000,LPSTR:00000000)
AppendMenuA returns: 1
```

图 10-17　运行 APISPY32 得到的 CLOCK 的输出文件

大部分时间中，OUT 文件中的一个函数调用行通常跟着这个函数的返回值行。然而，事情并非总是如此。注意（在图 10-17 中）DefWindowProcA 函数及其返回值行是如何缩进的。这说明这个函数是在另一函数（在这个例子中是 SetWindowTextA）的执行中被调用的。这样的顺序很有意义，因为 DefWindowProc 的第二个参数（信息参数）显示出值为 0xC。在 WINUSER.H 中查找这个号码，你将发现这条信息是 WM_SETTEXT。由于 DefWindowProc 是在调用 SetWindowText 的过程中被调用的，我们可以作这样的假定，SetWindowText 给该程序的窗口过程发送一条 WM_SETTEXT 消息，而该程序并不处理这条消息，而是简单地把它传递给 DefWindowProc。在 APISPY32 的这个输出文件中只有一层嵌套。嵌套 4 或 5 层的函数并不常见，尤其在主窗口接到一条 WM_CLOSE 消息时程序的关闭顺序中。

在看到 LPSTR 型参数时，记住也许没有给出完整的字符串。由于象缓冲区之类无意义的字符串也许会作为参数被传递，我无法预先知道对每个 LPSTR 型的参数需要显示多少个字符。于是我决定只打印前 10 个字符或遇到空格为止。还

有，制表符、回车符、换行符分别用 \t、\r、\n 代替。如果我照原样打印这些字符，OUT 文件中的各行就没有了恰当的格式。

如果你在 Win32s 的早些版本上(向前到 Win32s 1.2)运行 APISPY32，如果你运行调试 16 位的 USER.EXE，你将得到一大堆 RIP。这是 Win32s 的一个与 Win16 和 Win32 之间消息传递相关的缺陷。Win32s 的调试程序在 16 位代码和 32 位代码之间有消息传递时需要转换某些消息的代号。为了知道要转换哪条消息，Win32s 需要知道窗口的级别，所以 Win32s 在 USER.EXE 中调用 GetClassName。这样当遇到一条 HWND 为 0 的消息时就会出问题。如果一个 0 HWND 传给 GetClassName 的调试版本，它就会 RIP。我们的侦探程序会在什么地方接到 HWND 为 0 的消息呢？如前所述，是在 Win32s WaitForDebugEvent 函数将 RegisterWindowMessage(W32S_Debug_Msg)消息放到调试程序的消息队列中时。

如果你是用 Borland C++编写这个 APISPY32 程序，你在侦探多线程程序时就不会那么顺利。Borland C++的多线程库中某些函数(在 APISPY32 中相关的函数是 fprintf)使用单一进程数据。在 Borland 运行库中，那些代码不理会 DLL_THREAD_ATTACH 事件报告。实际上，运行库靠程序对_beginthread 函数的调用来得知何时分配它的进程数据。不幸的是，这种方法在一个线程不是在你的模块中创建时就失效了。拿我们的侦探程序来说，APISPY32.DLL 中的 Borland 运行库代码不会发现由被侦探的 EXE 文件对 beginthread 的调用。Borland 公司承认这是一个缺陷，但这个问题直到 BC++ 4.5 中都未解决。

在你自己的程序中拦截函数

在本章的开始，曾答应提供一个你可以在你自己的代码中使用 APISPY32 函数拦截技术的方法。做这类工作的代码在 HOOKAPI.C 中，如图 10-18 所示。HOOKAPI.C 中的 HookImportedFunction 使你可以拦截在一个模块中对另一模块中的给定函数的所有调用。例如，你使用一个叫 FOO.DLL 的 DLL，你可以拦截 FOO.DLL 对 MessageBeep 例程的所有调用(即使你没有 FOO.DLL 的源程序)。如果你还想拦截 BAR.DLL 和 BAZ.DLL 对 MessageBeep 的调用，你还需对每个 DLL 调用一次 HookImportedFunction 例程。

另一点需要记住的很重要的是这种拦截技术只拦截在你自己的进程中的引入函数。它不能拦截由其他进程发出的 API 函数调用。换句话说，你只能拦截你的 EXE 及其 DLL 发出的调用。你不能用它来做象拦截 WINFILE 对 OpenFile 的所有调用之类的事。你的拦截代码不会映象到 WINFILE 进程的地址空间中。

HookImportedFunction 的第一个参数是你想从中拦截函数调用的 EXE 或 DLL 的模块句柄(在前面的例子中是 FOO.DLL)。第二个参数是包含在你想拦截的函数的模块的模块名。最后一个参数是你的替身函数的地址。HookImportedFunction 返回你刚才拦截的函数的原来地址。如果有必要，你可以用这个地址连到原来的代码上。用上面的例子，可以看到调用 HookImportedFunction 的情形如下：

```
    pfnOriginalProc = HookImportedFunction( GetModuleHandle("BAR.DLL"),
                                            "USER32.DLL",
                                            "MessageBeep",
                                            MyMessageBeepHandler );


// Macro for adding pointers/DWORDs together without C arithmetic interfering.
#define MakePtr( cast, ptr, addValue ) (cast)( (DWORD)(ptr)+(DWORD)(addValue))

DWORD GetModuleBaseFromWin32sHMod(HMODULE hMod); // Prototype (defined below)

PROC WINAPI HookImportedFunction(
        HMODULE hFromModule,        // Module to intercept calls from.
        PSTR    pszFunctionModule,  // Module to intercept calls to.
        PSTR    pszFunctionName,    // Function to intercept calls to.
        PROC    pfnNewProc          // New function (replaces old function).
        )
{
    PROC pfnOriginalProc;
    PIMAGE_DOS_HEADER pDosHeader;
    PIMAGE_NT_HEADERS pNTHeader;
    PIMAGE_IMPORT_DESCRIPTOR pImportDesc;
    PIMAGE_THUNK_DATA pThunk;

    if ( IsBadCodePtr(pfnNewProc) ) // Verify that a valid pfn was passed.
        return 0;

    // First, verify the module and function names passed to use are valid.
    pfnOriginalProc = GetProcAddress( GetModuleHandle(pszFunctionModule),
                                      pszFunctionName );
    if ( !pfnOriginalProc )
        return 0;

    if ( (GetVersion() & 0xC0000000) == 0x80000000 )
        pDosHeader =                                            // Win32s
            (PIMAGE_DOS_HEADER)GetModuleBaseFromWin32sHMod(hFromModule);
    else
        pDosHeader = (PIMAGE_DOS_HEADER)hFromModule;            // other

    // Tests to make sure we're looking at a module image (the MZ header).
    if ( IsBadReadPtr(pDosHeader, sizeof(IMAGE_DOS_HEADER)) )
        return 0;
    if ( pDosHeader->e_magic != IMAGE_DOS_SIGNATURE )
        return 0;

    // The MZ header has a pointer to the PE header.
    pNTHeader = MakePtr(PIMAGE_NT_HEADERS, pDosHeader, pDosHeader->e_lfanew);

    // More tests to make sure we're looking at a "PE" image.
    if ( IsBadReadPtr(pNTHeader, sizeof(IMAGE_NT_HEADERS)) )
        return 0;
    if ( pNTHeader->Signature != IMAGE_NT_SIGNATURE )
        return 0;
```

```
// We know have a valid pointer to the module's PE header. Now go
// get a pointer to its imports section.
pImportDesc = MakePtr(PIMAGE_IMPORT_DESCRIPTOR, pDosHeader,
                      pNTHeader->OptionalHeader.

                      DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].
                      VirtualAddress);

// Bail out if the RVA of the imports section is 0 (it doesn't exist).
if ( pImportDesc == (PIMAGE_IMPORT_DESCRIPTOR)pNTHeader )
    return 0;

// Iterate through the array of imported module descriptors, looking
// for the module whose name matches the pszFunctionModule parameter.
while ( pImportDesc->Name )
{
    PSTR pszModName = MakePtr(PSTR, pDosHeader, pImportDesc->Name);

    if ( stricmp(pszModName, pszFunctionModule) == 0 )
        break;

    pImportDesc++;  // Advance to next imported module descriptor.
}

// Bail out if we didn't find the import module descriptor for the
// specified module. pImportDesc->Name will be nonzero if we found it.
if ( pImportDesc->Name == 0 )
    return 0;

// Get a pointer to the found module's import address table (IAT).
pThunk = MakePtr(PIMAGE_THUNK_DATA, pDosHeader, pImportDesc->FirstThunk);

// Blast through the table of import addresses, looking for the one
// that matches the address we got back from GetProcAddress above.
while ( pThunk->u1.Function )
{
    if ( pThunk->u1.Function == (PDWORD)pfnOriginalProc )
    {
        // We found it! Overwrite the original address with the
        // address of the interception function. Return the original
        // address to the caller so that they can chain on to it.
        pThunk->u1.Function = (PDWORD)pfnNewProc;
        return pfnOriginalProc;
    }

    pThunk++;   // Advance to next imported function address.
}

return 0;   // Function not found.


typedef DWORD (_ _stdcall *XPROC)(DWORD);

// Converts an HMODULE under Win32s to a base address in memory.
DWORD GetModuleBaseFromWin32sHMod(HMODULE hMod)
{
```

```
    XPROC ImteFromHModule, BaseAddrFromImte;
    HMODULE hModule;
    DWORD imte;

    hModule = GetModuleHandle("W32SKRNL.DLL");
    if( !hModule )
        return 0;

    ImteFromHModule = (XPROC)GetProcAddress(hModule, "_ImteFromHModule@4");
    if ( !ImteFromHModule )
        return 0;

    BaseAddrFromImte = (XPROC)GetProcAddress(hModule, "_BaseAddrFromImte@4");
    if ( !BaseAddrFromImte )
        return 0;

    imte = ImteFromHModule( (DWORD)hMod);
    if ( !imte )
        return 0;

    return BaseAddrFromImte(imte);
}
```

图 10-18　HOOKAPI.C 让你拦截你自己的程序发出的函数调用

你用来代替原来的 API 函数的函数应该有与你拦截的函数完全相同的原型说明。这使你可以存取该函数的所有参数，并使编译器在函数返回时正确地从栈中弹出相应的 BYTE 数。如果你想把控制交回原来的函数作为你处理的一部分，你可通过 HookImportedFunction 返回的函数指针来调用原来的函数。典型地，你在你的函数中最后做这件事，你将原来的 API 函数返回的值返回给你自己的代码。

为了说明这个 HookImportedFunction 的用途，我写了这个 SimonSez 程序。这个程序极为简单，由拦截 SimonSez 对 MessageBox 的调用和将“SimonSez:”放在信息的前面显示组成。由于 SimonSez 已预先拦截了 MessageBox 函数，安装好的处理函数(MyMessageBox)将被调用，而不是 USER32.DLL 中的 MessageBox 函数。然后 MyMessageBox 调用 USER32.DLL 中的 MessageBox 函数。SIMONSEZ.C 程序在图 10-19 中给出。

```
//==================================
// SIMONSEZ - Matt Pietrek 1995
// FILE: HOOKAPI.C
//==================================
#include <windows.h>
#include <malloc.h>
#include "hookapi.h"

// Make a typedef for the WINAPI function we're going to intercept
typedef int (_ _stdcall *MESSAGEBOXPROC)(HWND, LPCSTR, LPCSTR, UINT);
```

```
MESSAGEBOXPROC PfnOriginalMessageBox;   // For storing original. address

//
// A special version of MessageBox that always prepends "Simon Sez: "
// to the text that will be displayed.
//
int WINAPI MyMessageBox( HWND hWnd, LPCSTR lpText,
                         LPCSTR lpCaption, UINT uType )
{
    int retValue;               // Real MessageBox return value.
    PSTR lpszRevisedString;     // Pointer to our modified string.

    // Allocate space for our revised string - add 40 bytes for new stuff.
    lpszRevisedString = malloc( lstrlen(lpText) + 40 );

    // Now modify the original string to first say "Simon Sez: ".
    if ( lpszRevisedString )
    {
        lstrcpy(lpszRevisedString, "Simon Sez: ");
        lstrcat(lpszRevisedString, lpText);
    }
    else                                    // If malloc() failed, just
        lpszRevisedString = (PSTR)lpText;   // use the original string.

    // Chain on to the original function in USER32.DLL.
    retValue = PfnOriginalMessageBox(hWnd,lpszRevisedString,lpCaption,uType);

    if ( lpszRevisedString != lpText )  // If we successfully allocated string
        free( lpszRevisedString );      // memory, then free it.

    return retValue;    // Return whatever the real MessageBox returned.
}

int APIENTRY WinMain( HANDLE hInstance, HANDLE hPrevInstance,
                      LPSTR lpszCmdLine, int nCmdShow )
{
    MessageBox(0, "MessageBox Isn't Intercepted Yet", "Test", MB_OK);

    // Intercept the calls that this module (TESTHOOK) makes to
    // MessageBox() in USER32.DLL. The function that intercepts the
    // calls will be MyMessageBox(), above.

    PfnOriginalMessageBox = (MESSAGEBOXPROC) HookImportedFunction(
                    GetModuleHandle(0),     // Hook our own module
                    "USER32.DLL",           // MessageBox is in. USE32.DLL
                    "MessageBoxA",          // Function to intercept.
                    (PROC)MyMessageBox);    // Interception function.

    if ( !PfnOriginalMessageBox )   // Make sure the interception worked.
    {
        MessageBox(0, "Couldn't hook function", 0, MB_OK);
        return 0;
    }
```

```
// !!!!!!!!!!!!!!!!!!!!!!!!  WARNING  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// When built with optimizations, the VC++ compiler loads a
// register with the address of MessageBoxA, and then makes all
// subsequent calls through it. This can cause the MessageBox call
// below to not go through the Import Address table that we just patched.
// For this reason, the .MAK file for this program does not use the
// /O2 or /O1 switches. This usually won't be a problem, but it
// was in this particularly simple program.  ACCKK!!!!

// Call MessageBox again. However, since we've now intercepted
// MessageBox, control should first go to our own function
// (MyMessageBox), rather than the MessageBox() code in USER32.DLL.

MessageBox(0, "MessageBox Is Now Intercepted", "Test", MB_OK);

return 0;
```

图 10-19　说明 HookImportedFunction 用途的简单的 SIMONSEZ.C 程序

HookImportedFunction 是如何工作的？如前所述，当一个 Win32 程序调用从另一个模块中链入的函数时，这个调用实际上将控制交给一条 JMP DWORD PTR [XXXXXXXX]指令。在内存地址为 XXXXXXXX 处的 DWORD 包含了该引入函数的地址(例如，USER32.DLL 中 MessageBox 的地址)。

HookImportedFunction 的代码所做的一切就是在链入地址表中搜索这个你想拦截的函数的地址 DWORD。一旦它发现这个位置，HookImportedFunction 用你的处理函数的地址来改写该引入函数的地址。

HookImportedFunction 的第一个参数是标志你想从中拦截函数调用的 EXE 或 DLL 的一个 HMODULE。在 Windows NT 和 Windows 95 中，一个 HMODULE 只不过是这个模块在内存中的线性起始地址。这就是我们知道的模块基地址。由于 Win32 使用内存映象文件，在该模块基地址处的内存值是一个 DOS MZ 头部(参见 WINNT.H 中的 IMAGE_DOS_HEADER)。HookImportedFunction 用 e_lfanew 域的值来找到 PE 头的地址(WINNT.H 中的 IMAGE_NT_HEADERS 结构)。在 IMAGE_NT_HEADERS 结构的尾部是一个包含模块中重要区域的地址的结构数组。我们的 HookImportedFunction 例程对链入地址表的开始部分尤其感兴趣。这个表(通常在模块的 .idata 部分)包含了被这个模块链入的函数的有关信息。HookImportedFunction 需要用你的处理函数的地址改写的那个 DWORD 就在这个链入地址表的某个地方。

在这个链入地址表的开始部分是一个 IMAGE_IMPORT_DESCRIPTOR 结构数组(再看看 WINNT.H)。这些结构中有一个是针对这个模块从中引入函数的 DLL 的。IMAGE_IMPORT_DESCRIPTOR 有一个相关 DLL 名字的指针，以及指向链入地址表的指针(先前提到过的函数地址数组)。HookImportedFunction 从头到尾搜索 IMAGE_IMPORT_DESCRIPTOR 数组直到它找到其名字与传给 HookImportedFunction 的 pszFunctionModule 相匹配的那一个。该例程用这个 IM-

AGE_IMPORT_DESCRIPTOR 中的信息来产生一个指向链入地址表的指针。

在 HookImportedFunction 开始部分附近的代码调用 GetProcAddress 来获取我们想要拦截的函数的地址。这一地址应该在我们刚找到的链入地址表中。HookImportedFunction 的代码在这个地址数组中从头到尾搜索，直到它发现带有一个与 GetProcAddress 返回的地址相同的地址的夹缝为止。接下来只须将这个新的函数的地址(pfnNewProc 参数)拷贝到那个夹缝中并返回原来函数的地址。

小结

Win32 程序设计对 Win16 环境中的程序员提出了一整套新的挑战。总之，Win32 系统程序设计因有了象独立地址空间与多线程执行之类问题而更严格且更复杂。在本章中，我们已经在编写 API 侦探程序和建立一套通用的拦截机制上遇到了这些问题。我们也看到了，尽管 Microsoft 宣称“只有一个 Win32 API”，在你涉及到细节时仍偶而有些不同。无论如何，通过理解这些问题，你可以写 出在所有 Win32 平台上都能运行得很好的具有专业水平的程序。

附 录 A

未公布的 KERNEL32.DLL 链接库

* *

Windows 95 KERNEL32.DLL 的链接表的前 100 个函数仅公布了其序号，与此相对，所有 Microsoft 提供的普通的 Win32 API 函数都同时公布了函数名和序号。公布一个函数的名字意味着您可以将这个函数名作为 GetProcAddress 的参数来得到这个函数的地址，您可以通过这个地址来调用它。

显而易见，既然 Microsoft 未公布这 100 个函数的名字，它就没有打算让您调用或使用它们。换个说法，它们是“未公布的”函数。就我们所知，未公布的函数可能非常有用。事实上，有时使用一个未公布的函数是您能完成某个特殊任务的唯一途径。然而，由于未公布这些函数的名字，您不能只调用 GetProcAddress 来象您也许会指望的那样使用它们。

通常在这种情况下，有经验的程序员不会灰心，因为他们知道可以将一个函数的链接序号代替其名字传给 GetProcAddress。可惜，如您在第三章中所见，Microsoft 编写 KERNEL32 的程序员将这个暗门也封死了。GetProcAddress 函数对尝试用序号值在 KERNEL32.DLL 中查找函数地址的任何调用都宣告失败。有趣的是，GetProcAddress 只禁止在 KERNEL32.DLL 用序号来查找地址。所以很明显 Microsoft 试图阻止人们使用 KERNEL32.DLL 中的这 100 个函数。

不用担心。象您在本书中别处已看到的，这些对调用未公布的 KERNEL32.DLL 函数的人为限制是可以克服的。一种途径是编写您自己的 GetProcAddress 函数。这一点不难做到，因为内存中的一个已装载的 PE 模块的格式是详细公布了的。如果您想明白 Windows 如何应用它，第三章已给出了 GetProcAddres 的伪码描述。

这种“包上您自己的”GetProcAddress 的方法的问题在于不得不调用 GetProcAddress，然后还得将它的返回值保存到一个函数指针中，这一点非常令人头疼。使用这些未公布的函数的一个简单得多的方法是用一个包含这些函数的链接库。因而，本附录提供给您建立可以同 Visual C++或其它的 Microsoft 编辑/链接程序一起使用的链接库的工具。图 A-1 给出了 K32LIB.DEF，它包含这 100 个左右已有但未公布的 KERNEL32.DLL 函数中的大部分。

```
LIBRARY KERNEL32
EXPORTS
    VxDCall0@0              @1
    VxDCall1@8              @2
    VxDCall2@12             @3
    VxDCall3@16             @4
    VxDCall4@20             @5
    VxDCall5@24             @6
    VxDCall6@28             @7
    VxDCall7@32             @8

    CharToOemA@8            @10 ; USER32's version calls straight here.
    CharToOemBuffA@12       @11 ; USER32's version calls straight here.
    OemToCharA@8            @12 ; USER32's version calls straight here.
    OemToCharBuffA@12       @13 ; USER32's version calls straight here.
    LoadStringA@16          @14 ; USER32's version calls straight here.
    wsprintfA@8             @15 ; USER32's version calls straight here.
    wvsprintfA@4            @16 ; USER32's version calls straight here.
    CommonUnimpStub@0       @17 ; Non-implemented APIs call here.
    GetProcessDWORD@8       @18

    DosFileHandleToWin32Handle@4        @20
    Win32HandleToDosFileHandle@4        @21
    DisposeLZ32Handle@4                 @22
    GDIReallyCares@4                    @23
    GlobalAlloc16@8                     @24
    GlobalLock16@4                      @25
    GlobalUnlock16@4                    @26
    GlobalFix16@4                       @27
    GlobalUnfix16@4                     @28
    GlobalWire16@4                      @29
    GlobalUnWire16@4                    @30
    GlobalFree16@4                      @31
    GlobalSize16@4                      @32
    HouseCleanLogicallyDeadHandles@0    @33
    GetWin16DOSEnv                      @34
    LoadLibrary16@4                     @35
    FreeLibrary16@4                     @36
    GetProcAddress16@8                  @37
    AllocMappedBuffer                   @38
    FreeMappedBuffer                    @39
    OT_32ThkLSF                         @40
    ThunkInitLSF@20                     @41
    LogApiThkLSF@4                      @42
    ThunkInitLS@20                      @43
    LogApiThkSL@4                       @44
    Common32ThkLS                       @45
    ThunkInitSL@20                      @46
    LogCBThkSL@4                        @47
    ReleaseThunkLock@4                  @48
    RestoreThunkLock@4                  @49

    W32S_BackTo32                       @51
    GetThunkBuff@0                      @52
```

```
GetThunkStuff@8                    @53
K32WOWCallback16@8                 @54
K32WOWCallback16Ex@20              @55
K32WOWGetVDMPointer@12             @56

WOWGlobalAlloc16@8                     @59
WOWGlobalLock16@4                      @60
WOWGlobalUnlock16@4                    @61
WOWGlobalFree16@4                      @62
WOWGlobalAllocLock16@12                @63
WOWGlobalUnlockFree16@4                @64
WOWGlobalLockSize16@8                  @65
WOWYield16@0                           @66
WOWDirectedYield16@4                   @67
K32WOWGetVDMPointerFix@12              @68
K32WOWGetVDMPointerUnfix@4             @69
K32WOWGetDescriptor@8                  @70
IsThreadId@4                           @71
K32RtlLargeIntegerAdd@16               @72
K32RtlEnlargedIntegerMultiply@8        @73
K32RtlEnlargedUnsignedMultiply@8       @74
K32RtlEnlargedUnsignedDivide@16        @75
K32RtlExtendedLargeIntegerDivide@16    @76
K32RtlExtendedMagicDivide@20           @77
K32RtlExtendedIntegerMultiply@12       @78
K32RtlLargeIntegerShiftLeft@12         @79
K32RtlLargeIntegerShiftRight@12        @80
K32RtlLargeIntegerArithmeticShift@12   @81
K32RtlLargeIntegerNegate@8             @82
K32RtlLargeIntegerSubtract@16          @83
K32RtlConvertLongToLargeInteger@4      @84
K32RtlConvertUlongToLargeInteger@4     @85

FT_PrologPrime          @89
QT_ThunkPrime           @90
PK16FNF@0               @91
GetPK16SysVar@0         @92
GetpWin16Lock@4         @93 ; Returns a pointer to the Win16Mutex.
_CheckNotSysLevel@4     @94
ConfirmSysLevel@4       @95
_ConfirmWin16Lock@0     @96
EnterSysLevel@4         @97 ; Acquire a mutex (e.g., Win16Mutex).
LeaveSysLevel@4         @98 ; Release a mutex (e.g., Win16Mutex).
```

图 A-1　您可用这些未公布的 KERNEL32.DLL 函数来建立您自己的链接库来和 Visual C++或其它编译器和链接器一起使用

您可能已经注意到我在图 A-1 中所列未公布的函数并未全部给出函数原型。虽然可以写象“未公布的 Windows”这样完整的一章来公布并给出这些函数的原型，但这已不是本书的目的。毫无疑问在别的书中这些函数将被详细地说明和描述。注意某些函数，如 VxDCall0@0 在本书中其他地方已经用到，而您可以很容易地查出某些别的函数的参数和作用。

K32LIB. DEF 及 K32LIB. LIB(适用于 Microsoft VC++)包含在随本书一起出版的软盘中;您可以在 APPENDIX 子目录中找到它们。正常情况下,Microsoft 链接程序在链接一个 DLL 时为它建立一个链接库。然而 Mierosoft LIB. EXE 可以由一个 .DEF 文件建立一个链接库。要由 K32LIB. DEF 建立链接库,您可以在 K32LIB. DEF 所在的子目录中用 MAKE. BAT。MAKE. BAT 不过是下面这行命令:

```
lib /MACHINE:IX86 /DEF:K32LIB.DEF
```

如要在您的工程中使用 K32LIB. LIB,您应该将它紧接着 KERNEL32. LIB 放在链接库列表中。这迫使 Microsoft 链接程序将 K32LIB. LIB 中的代码和数据与 KERNEL32. LIB 中的代码和数据连续地放在一起。在形成的可执行文件中您将看到对 KERNEL32. DLL 的两处调用。不用太在意,您会得到 KERNEL32. DLL 的两个 IMAGE_IMPORT_DESCRIPTOR 头部,但并不是描述每一链接函数的所有数据的两份拷贝。(解释 Microsoft 链接程序在这个层次做些什么需要很长的篇幅,所以不要问……)

Borland C++的用户可用 K32LIB. DEF 并通过 IMPORT. LIB 运行它来建立一个符合 TLINK 使用格式的链接库,这种情况下的命令行如下:

```
IMPLIB K32LIB.LIB K32LIB.DEF
```

您可以将 K32LIB. LIB 放在链接库列表的任何地方,TLINK 不关心它们出现的顺序。

如您在本书中许多其它的程序中所见的,K32LIB. LIB 对调用 Microsoft 不想让您使用的 Windows 95 函数是非常有用的。当然,您应该尽可能避免使用未公布的函数。一个非常好的理由是如果您用了未公布的函数,您的程序不能在 Windows NT 上运行,而且也许在将来的 Windows 版本上也不能运行。本书中的程序是与 Windows 95 不可分的,而且明确地设计成展示在 Windows 95 中实际发生着什么。没有理由让我避免在本书的程序中使用这些函数。

如果您绝对需要在您的程序中使用这些函数,在您的代码中加入版本测试或其它的安全性检查,这样即使您的程序出错也不会失去控制。要做到这点(同时应避免装载失败),您将必须使用 GetProcAddress 而不是直接调用这些函数。这就是在已公布的操作系统 API 的边缘上工作的危险所在。