《Windows 95 系統程式設計 大奧秘》
(Windows 95 System Programming SECRETs 繁體中文版)
電子書 開放自由下載 聲明 / 侯捷

近年來許多讀者問我，哪裡可以買到這本書。我總是給他們殘酷的回答：這本書已經絕版了。

鑑於本書仍有非常高的技術價值，鑑於還有這麼多讀者需要它，並且基於以下兩個現實的成立，我決定將本書製作成 PDF 電子檔，開放免費下載：
1. 本書英文版已絕版
2. 本書中文版已絕版，我亦已與中文版出版公司 (旗標) 簽訂解約條款。
換句話說，這本電子書的傳佈，不會造成任何人財務上的損失。

固然我不清楚，法律上或道德上是否允許我，做爲一個譯者，在不損及任何人(包括原作者、原出版公司、中文出版公司) 利益的前提下將此書製作電子檔免費傳佈，不過，基於眾多讀者的需求，尤其是大陸讀者對於系統層級的好書的殷切期盼，我決定這麼做。

希望我不會收到一張法院傳喚單 ☺。

＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊＊

本書目前仍深具價值的章節是：3,5,8,10。我所謂深具價值，並非指這些章節中所挖掘的 Windows 內部資料結構或虛擬碼 (pseudo code) 仍適用於目前最新一代的 Windows 作業系統。我的意思是，這些章節所揭露出來的 Windows 內部資料結構和虛擬碼，向我們展示了一個複雜如 Windows 的作業系統，是如何管理其記憶體、模組、行程、執行緒、又是如何完成動態聯結、如何組織其可執行檔、如何將記憶體定址空間分離。這使我們對於 Win32 作業系統這一家族系列，在作業系統基本教義的範疇內，有一個明晰的圖像與掌握。

大學課程中的 Operation System (作業系統) 一門，由於過於理論，充滿概念，令學習者難以具體瞭解實作上的可能性。透過本書的輔助，可以確實「看」到一個全球普及的作業系統的實際長像。我認爲這對 Operation System 的學習，很有幫助。

我曾在元智大學開授一門三學分的「Windows 作業系統」，即以此書爲教材。我帶領同學們分組追蹤了各章源碼，並讓同學們以第 10 章爲基礎，實際寫一個可攔截 Windows 系統的程式。對於一個大三學生，這樣的份量或許稍重了些，但我相信這是個很好的訓練，讓同學深切體會什麼叫做「深入系統的靈魂」，什麼叫做「系統程式設計」。

本書書評，見 http://www.jjhou.com/review3-7.htm（苦澀後的甘甜 - Windows 系統深耕），或簡體版 http://www.csdn.net/expert/jjhou/review3-7.htm

由於 PDF 書籤（亦即目錄連結）的製作十分費工 (或許有方便作法而我不知道)，所以我只針對上述我認爲目前仍深具價值的 3,5,8,10 各章製作書籤（亦即目錄連結）。其他各章節的頁碼，請見目錄。

開放檔案如下：
檔名：windows95-system-programming-secrets.pdf
內容：全書
不需密碼即可開啓。檔案內含書籤（亦即目錄連結）。

本中文版，從某種角度來說，或許我有權將之公開。但是說到原書所附之源碼，由於完全沒有我的參與，所以我也完全沒有立場將之放在網上供人下載。請不要有任何人來詢問或要求書附源碼，我不會回應。

-- the end


WINDOWS
95 系統程式設計
大奧秘
Windows 95 System Programming SECRETS

# WINDOWS 95 系統程式設計大典

Windows 95 System Programming SECRETS

Matt Pietrek 著

侯俊傑 譯

旗標出版有限公司

# INSIDE SECRETS

- 獲悉微軟企圖阻止你呼叫的各種 KERNEL32 未公開函式。
- 打開 Windows 95 用以通往 VxDs 的秘密後門。
- 探究 32 位元行程所具備的 16 位元 KRNL386 資料結構。
- 了解自 thunks 呼叫 16 位元碼。
- 揭發 Windows 95 目前系統資源（FSR）的謊言。
- 把行程和執行緒的 ID 轉換為指向系統資料結構的指標。
- 攔截其他程式所呼叫的 API 函式。
- 完整列出被 KERNEL32 呼叫的 KRNL386 函式。
- 瞭解行程如何彼此共享記憶體。

# 再刷感言

**大發秘**（我對此書的暱稱）第一刷 3000 本。一個半月內再刷。

我真開心。

我太開心了！

不論你是因爲書籍內容，或是因爲作者 Matt Pietrek，或是因爲譯者侯俊傑，拿起這本書，我想你可能不是第一次成爲我的讀者，可能對我有點認識，可能知道我開心爲了什麼，不爲了什麼。

這本書技術層次艱深，價格高貴，我從來不敢想像它中譯出來之後的市場反應。事實證明，好書不寂寞。一位從事系統技術的朋友以「欣喜若狂」來形容他在書店看到此譯本的情緒。一位讀者說如果不是這譯本，他雖然知道原著好，恐怕也無緣一讀。一位朋友則說，原書售價 1350，譯本才 860，又是侯 sir 製作，開玩笑，爲什麼不買？BBS 上有人說：作者 Matt Pietrek 和譯者侯俊傑，超強組合，一定要買。

而我說，我重新確立了一種信心。

這本書一完成，我也成爲它的一個讀者，而且我相信不會有人比我讀得更爛熟。我用這本書做爲大學教材，也用這本書爲科學園區的工程師朋友們開了好幾次課程。每一次翻閱這本書，就著讀者的心情，我不但感謝 Matt Pietrek 的苦心孤詣，提供我們那麼完整而難得的資料，我也感謝我們的出版社，願意花大筆簽約金，大筆製作費，在**大學秘**這樣層次的書籍上做中文化的功夫。你知道，即使初期市場反應良好，我也不敢預期出版社什麼時候才能「損益平衡」！

世有伯樂，而後千里馬出。我還特別感謝一個人。

**侯俊傑** 1997.04.12 于新竹
jjhou@ccca.nctu.edu.tw

P.S. 第一刷的圖 3-2 和圖 8-1 有誤，已在此新刷中更正。同時亦更正了幾個錯別字。關於這些錯誤，我在這裡向讀者道歉。

## 關於作者

Matt Pietrek 是 ***Microsoft Systems Journal*** 期刊的 "Under the Hood" 專欄作家。他也爲 ***PC Magazine*** 及其他刊物寫稿。他是 ***Windows Internals***（Addison-Wesley，1993）一書的作者，也是 ***Undocumented Windows***（Addison-Wesley，1992）作者群之一。除了寫作，Matt 是 Nu-Mega 公司的 BoundsChecker 系列產品的資深建構者。他住在 Nashua, New Hampshire。他的 e-mail 地址是 71774.362@compuserve.com。

## 關於譯者

侯俊傑是 ***RUN!PC*** 雜誌的「技術空間」專欄作家。他也曾經爲其他數本電腦雜誌寫稿。他著有「**深入核心 - Windows 作業系統**」（旗標，1993）和「**深入淺出 MFC**」（松崗，1996）等書籍。除了寫作，侯俊傑亦專長於電腦書籍的翻譯與評論，是一位橫跨著、譯、評三領域的資訊觀察研究員。他住在臺灣新竹。他的 e-mail 地址是 jjhou@ccca.nctu.edu.tw。

# 山川壯麗 鬼斧神工

（譯序）

這是截至目前我所譯過難度最高的一本書。所幸我一直就在這個領域裡頭做學問，也早就鑽研過了原著，才能夠在一百天的時間裡完成它。我與作者 Matt Pietrek 神交已久。不，我並不認識 Matt，我是指他在雜誌和期刊上的許多文章，早就成爲我的重要精神糧食。可以說，在作業系統這個領域，他一直是我的導師。

關於作業系統，書籍的取材可以很廣，譬如 Walter Oney 的 *Systems Programming for Windows 95*（Microsoft Press）著重在作業系統與硬體之間的介面、驅動程式與 VxDs，旨在從硬體與軔體（firmware）層面看系統。Jeffrey Richter 的 *Advanced Windows*（Microsoft Press）著重在 32 位元架構，以及與系統核心有關的 API 函式，旨在從 Win32 API 的層面看系統。但是，別忘了，Windows 95 有許多 16 位元「遺老」在其中，而且扮演吃重的角色。Matt 的這本 *Windows 95 system programming SECRETS*（IDG Books）著重在 16-/32- 位元核心資料結構的介紹，及其相關函式（含未公開函式）的內部動作。這些核心資料結構包括 modules、processes、threads、tasks。本書也對 KRNL386 / KERNEL32、GDI / GDI32、USER / USER32 三大模組做了非常徹底的挖掘，幾乎到了寸草不留的地步。此外，隱藏在三大模組背後的 VMM、VWIN32、ADVAPI32 等神秘的 VxDs，作者也有非常深入的剖析。PE 可執行檔格式對於 Win32 程式載入、DLL 模組載入、函式輸入（imported）、函式輸出（exported）、

動態聯結機制等題目，有密不可分的關聯，而 Matt 對於 PE 檔案格式的透徹分析，讓我們有醍醐灌頂之感。

系統的剖析之外，Matt 還公開了他的私房菜。第 9 章是他使用各種分析工具的經驗，以及反組譯的實務心得。第 10 章（最後一章）的 spy 軟體設計，則是對整個系統結構工程的總驗收。Matt 讓我們見識什麼叫做山川壯麗，什麼叫做鬼斧神工！我非常欣賞這一章，研讀的同時畫下數十張草圖，才徹底掌握其精神。這些技術在 Matt 而言易如反掌折枝，因爲他是頂頂有名的 BoundsChecker 除錯器的建構者。

本書深度，勿庸置疑。本書密度，勿庸置疑。以資訊份量而言，Matt 的書真正是物超所值！

關於 Windows 作業系統，雖然我自己也寫一些東西，我想我肯定及不上 Matt，但是關於知識的表現，Matt 可就未必及得上我了。這本書的缺點，就在於 Matt 惜圖如金。許多觀念，其實配合一張圖就可以表現得淋漓盡致，Matt 卻不。整本書雖然有些不錯的示意圖，但和浩瀚字海比起來，彷彿滄海一粟。有時候我有爲他畫張圖的衝動，卻總是強忍下來，畢竟我是譯者，不是作者。Matt 的文風活潑主觀，相當好看，但有點囉哩八嗦。這一點我盡力讓它洗鍊些。此外，原書有不少錯別字，想是當初爲了趕 Windows 95 上市熱潮所致。不必擔心，中譯本都改過來了（除非我沒發現）。

鑽研 Windows 95 系統，是不是一項划算的投資？不是還有 Windows NT 嗎？95 和 NT 不是有許多不同嗎？我不想談 Microsoft 的策略權謀，我想告訴你，學問是堆積起來的。基礎愈深，堆得愈高；基礎愈廣，堆得愈快。觸類旁通是我在這個千變萬化一日千里的領域中最大的感受和依侍。這本書讓我們全盤瞭解 Windows 95，而 NT 只在隔壁房間。

**侯俊傑** 1997.01.30 于新竹
jjhou@ccca.nctu.edu.tw

**後記：**

寫下這一篇序的同時，我一邊聽著世界三大男高音世紀聯演的現場錄音 CD。星光閃耀的溫潤高音衝擊我的耳鼓，令我感到暈眩悸動。萬馬奔騰的男高音當然是一種藝術的極致，靜悄悄的文字藝術具有同樣震憾人心的能力。如果一首衝上高音 C 的人聲曲是藝術，令你血脈賁張，一本讓你握拳大喊 eureka，令你整夜興奮睡不著覺的技術書籍，算不算得上也是一種藝術？

文字創作當然是一門藝術。即使是技術性的文字創作，我也當它是一門藝術。翻譯是一門藝術，即使是技術性文字的翻譯，我也視之爲一門藝術。

這本書探觸了影響當今桌上電腦生態最鉅的 Windows 95 系統深處，讓我們見識了技術之美。也許有人認爲 Windows 內部九拐十八彎的動作哪稱得上美，我要這麼說，它呈現了一種難以想像的複雜度和巨大架構，也是美的另一種形式。Matt Pietrek 這本書的內涵具備了絕對的質量，但其形式卻沒辦法彰顯出藝術的光華，離我的理想還有一大段距離。因爲溫故，也因爲知新，我從這本書中獲得許多寶貴的東西。我將以此加上我已經籌劃一整年的稿子，近百幅圖片，完成一本取材不盡相同，手法完全迥異，真正符合我心目中藝術份量的 Windows 作業系統書籍。

等著瞧，朋友。

山自高兮水自深！當塵霧消散，唯事實留傳

# 奇穴探險

（原 序）

*Windows 95 System Programming SECRETS* 是 Matt Pietrek 所寫的第三本有關於如何真正瞭解 Windows 系統的書籍。Matt 已經在上面耕耘了相當一段時間。他的技術大師生涯始自 1988 年的 Santa Cruz 大學畢業典禮。他獲得的是物理學位，只修過兩門電腦課程。在加入 Borland 技術支援部門之後，他很快因爲一次評量而看清自己 -- 他獲得的是最低分。

轉到 Borland R&D 部門的那段時光比較樂觀。在那裡 Matt 寫了 TDUMP 和 WinSpector。他甚至加入 OS/2 Turbo Debugger 的開發。他的辛勤工作獲得了豐富的報酬：在一次裁員行動中，他失業了。最後，Matt 終於在 Nu-Mega 公司找到了自我。今天他是 BoundsChecker 系列產品的主要建構者。

我第一次遇到 Matt 是在 1991 年春天，我們都參加了 Software Development conference。在那個場合中我們這些 Windows 擁護者被視爲異類。Charles Petzold 和我是 "Windows v.s. OS/2" 的小組成員。我們被其他小組成員大加撻伐，並且被觀眾激烈質問刁難，只因爲我們預測 PC 作業系統的主導榮冠將在不久的未來落在 Windows 頭上。

看來 Matt、Charles 和我是對的 -- 事實上 Windows 如今已經超越了技術領域，變成文化的一部份。在 Windows 95 開始銷售的那個週末，它的總收入超越侏儸紀公園。在所有誇大的宣傳背後，謝天謝地，還是有很多的牛肉在，而且發出嘶嘶聲。客戶從 Windows 3.1 移轉至 Windows 95，使我們終能免除記憶體模式的不安（那是我們經驗多年的痛苦回憶），使我們得以全面進入 32 位元。

如果 Windows 是一個大洞穴，這本書就是給那些不因 Win32 API 而滿足，想要探索洞穴的人的最佳禮物。在 Windows 95 巨大洞穴中，Matt 是第一流嚮導。事實上這本書原本叫做「Windows 奇穴探險」。許多其他的 Windows 95 程式開發書籍（包括名爲 Unauthorized 的那一本），都承諾爲你照明所有的黑暗，卻早在一年前或數年前就完成。那些作者爲了搶奪「第一本書」的名銜爭破頭，但從 1994 年五月的 Chicago beta1 之後就封刀了。其中一些書籍甚至資料老舊，假設錯誤。

Matt 就不一樣了。他細細觀察 Chicago 的每一個版本，包括 Windows 95 的最後問世版，帶給你最新資料，也就是本書所提供的。現在，戴上你的安全頭盔，點亮你的強力照明燈，展開驚心動魄令人難忘的奇穴探險吧！

Eric J. Maffei
Microsoft Systems Journal 主編
紐約，1995 年九月
ericm@microsoft.com

# 目錄提要

## 第 1 章　透視 Windows 95

本章溫習 Win32 作業系統（Windows NT、Win32s 和 Windows 95）的歷史背景。你可以得知每一個作業系統的優點與弱點，以及其他 Win32 環境如 OS/2 Warp 和 Phar Lap 公司的 TNT DOS extender。

## 第 2 章　Windows 95 有些什麼新東西

你將從這一章獲得廣泛的 Windows 95 架構概觀，以及爲什麼 Windows 95 從 Windows 3.1 演化而來，而不是一個全新系統的原因。本章也談到了高階主題如記憶體管理、執行緒同步化控制、視窗系統的改善等等。

## 第 3 章　模組、行程、執行緒

檢驗 Windows 95 的模組、行程、執行緒之後，你就可以解開 KERNEL32 用來實現其資料結構之謎。本章的 Win32 函式虛擬碼完全依賴這些資料結構。此外，你還會讀到 thread local storage 和 structured exception handling 兩項主題。

## 第 4 章　USER 和 GDI 子系統

微軟把 Windows 3.1 的視窗系統、訊息系統、繪圖系統重新整修，放在 Windows 95 之中。爲了更瞭解它們，你必須學習 32 位元的 USER 和 GDI heaps，以及 16 位元 USER heap 中的新資料結構對於所謂 free system resource 的影響。

## 第 5 章　記憶體管理

Windows 95 的 32 位元記憶體管理是相當複雜的一個領域。在這重量級的一章裡，你可以探索以分頁爲基礎（paged-based）的虛擬記憶體、分離位址空間、共享記憶體。每一個 Win32 記憶體管理函式也以虛擬碼描述出來。

### 第 6 章　VWINKERNEL32386

Windows 95 有三個非常基礎的核心元件：16 位元的 KRNL386、32 位元的 KERNEL32、以及 ring0 的 VWIN32.VXD。如果你把它們放在一起，就得到了 VWINKERNEL32386。檢驗三者之間的關係時，你會同時發現許多有用的函式 -- 雖然其中有一些是未公開函式。

### 第 7 章　Win16 的 Module 和 Task

千萬不要忽略了 Windows 95 的 16 位元核心資料結構。雖然 Windows 95 是一個 32 位元作業系統，它的許多資料結構早在 Windows 3.1 之中就出現了，包括 task database 以及 16 位元的 module database。這也顯示了 Windows 95 內部的紛亂狀態。

### 第 8 章　PE 與 COFF OBJ 檔案格式

如果要充份瞭解 Windows 95，你就必須瞭解 Portable Executable（PE） 檔案格式 -- 那是 Windows 95 和 Windows NT 的可執行檔格式。本章還告訴你 COFF OBJ 和 COFF LIB -- 聯結器用它們來製造 PE 可執行檔。

### 第 9 章　尋幽訪勝索目己

如果你真正想進入 Windows 95 的核心，第 9 章告訴你怎麼做。你會學到如何使用檔案傾印（dumping）工具和 API 刺探（spying）工具，以及如何檢驗反組譯碼，找出諸如區域變數、函式參數、if 句型...等等東西。本章最後以一些頗有用處的經驗提示收尾。

### 第 10 章　寫一個 Win32 API Spy

以前數章知識爲基礎，本章告訴你如何產生一個可擴充的 API 刺探工具（spying tool）。這個 spy 工具可以記錄 API 函式的運轉過程，以及它們的參數。

# 目 錄

# 簡介

晚近，微軟常會問人們一個問題：『今天你要上哪兒去？』這家公司大剌剌地推廣 Windows 95，使我們有這樣的印象，認爲它可以將我們帶到目的地。我們程式員關心的是，它是否真的是個合適的交通工具，可以把我們帶到目的地。幾乎每個人都同意 Windows NT 是凱迪拉克（或賓士，如果你喜歡的話），速度快又舒適平穩，問題是，Windows 95 是雪佛蘭呢？還只是輛手推車？唯一的答案就是自己打開引擎蓋看看。這就是你手上這本書的目的。唯有檢驗 Windows 95 作業系統的內部，才能說出它是否有尾部安定翼，是否有安全氣囊以及尊貴皮椅。

你或許會奇怪，爲什麼有像我這樣的人要把作業系統大卸八塊？專注在新技術如 OLE、MFC、OpenGL、Multimedia 不是更好嗎？雖然有些程式員喜歡只學習他所能獲得的資訊就好，另一些人則不知足地需要瞭解底部所有資訊 -- 或許是因爲我們不敢信賴未經驗證過的碼。不論什麼理由，**Windows 95 系統程式設計大奧秘**（***Windows 95 System Programming SECRETS***）是一本爲那樣的人準備的書。知識就是力量，擁有作業系統如 Windows 95 的更多知識，你也就更能夠掌控它。

**Windows 95 系統程式設計大奧秘**（***Windows 95 System Programming SECRETS***）並不是一本「當局」認可的 Windows 95 系統架構書籍。我非常放任地把焦點特別集中在我感興趣的主題上。我希望這本像磚頭一樣厚的書的某個角落裡有你感興趣的東西，或是有你寫程式時用得到的東西。

## 對你（讀者）的假設

我必須對本書讀者做點假設。總括地說，我的讀者必須是一個足夠勝任的 Windows 程式員，至少寫過一些 Windows 3.x 程式。這並不是一本教你「如何寫一個 Windows 95 應用

程式」的書籍。那樣的書已經夠多了。

是的，這本書假設你知道如何寫 Windows 3.1 或 Windows 95 程式。你現在希望達到另一個層次：瞭解 Windows 95 爲什麼（以及如何）運作。

知道 Windows 95 黑盒子中的奧秘後，你就可以清楚瞭解那些盲目執行的禮儀般的行爲。當你發現一隻臭蟲，除錯器會更有效一些 -- 如果你瞭解 Windows 95 是怎麼工作的話。怎麼辦到的？唔，如果你領悟 Windows 的行爲（或應該的行爲），你就可以比較容易在除錯器中辨識出你的程式哪裡錯了。

書中的例子都是以 C 語言完成，再混合一點點組合語言。我用來表現各式各樣 Windows 95 函式的虛擬碼也是以 C 語言爲基礎。因此，爲了獲得最大利益，你應該熟悉 C/C++。如果你使用其他語言如 Borland Pascal/Delphi，或許可以勉強通過。

## 虛擬碼（pseudo code）

由於本書最主要目標是顯示 Windows 95 如何運作，我爲 system DLLs 的各式各樣函式提供了虛擬碼。這些虛擬碼類似 C 語言。然而我打破了一些 C 的語法束縛，以求乾淨明瞭。虛擬碼是以 Windows 95 除錯版爲基礎，後者提供許多有幫助的診斷字串，以及其他有趣的訊息，使我們很容易看出 Windows 95 真正的動作。如果你不是使用 Windows 95 除錯版，或許應該重新考慮一下。如果你不喜歡除錯版，而嘗試進入 Windows 95 零售版之中，可得有心理準備 -- 從除錯器中看到的碼，和本書所列的虛擬碼不同。

# 範例程式

**Windows 95 系統程式設計大奧秘**（***Windows 95 System Programming SECRETS***）內含一些程式，用來讓你探索 Windows 95。所有這些程式（.EXE 和所有原始檔案）都放在隨書磁片中。我輕視那種一次放 30 頁程式碼列表的書籍。爲了這個理由，幾乎沒有任何一個程式的原始碼列於書頁之中。唯一例外是第 10 章的 APISPY32 程式。第 10 章的焦點是建立一個 Win32 API spy 軟體，原始碼的密切說明很有必要，用以展示觀念的應用。

如果你讀過 ***Microsoft Systems Journal*** 或 ***PC Magazine***，你或許曾經看過本書某些範例程式的前身。事實上有些章節已經摘錄刊登在前述期刊雜誌上。如果你看過那些文章，請不要跳過本書相關章節，因爲受限於雜誌期刊的格式和篇幅，它們之中有許多資料當初被割捨下來，沒有刊登。

例如，第 8 章的 PEDUMP 程式的大小幾乎是它最初在 ***Microsoft Systems Journal*** 上的兩倍。最初在 ***Microsoft Systems Journal*** 上的 APISPY32 程式只適用於 Windows 95 beta2，在後來的版本中不能跑。本書的 APISPY32 則可以順利執行。

# 透視 Windows 95（Win32 概觀）

當我下筆此刻，微軟正在瘋狂地複製 Windows 95，準備在八月份大舉問世。

Windows NT，一個已經問世兩年的產品，在許多人心目中的地位已經不斷下滑了。一般大眾對 NT 的認知是它很慢，而且對資源貪求無度。事實上 Windows NT 3.5 已經比其前身好很多，許多抱怨也都已經解決。我目前就相當能夠享受在 NT 上的開發樂趣。至於在 Windows 3.1 根基上執行的 Win32s，幾乎和 NT 同時問世，咸認爲有不可置信的臭蟲，不值得在上面發展。

看來 32 位元 Windows 程式設計的前景不是十分光明 -- 直到 Windows 95 問世。現在，不管你喜不喜歡它，只要你還想待在微軟的陣營中並跟上最新技術，你就必須面對它。微軟把所有的蛋都放到 Win32 這個籃子中。雖然微軟還是會繼續支援 16 位元程式，但是已經不允許它們取得最新性質。假設 Win32 就是未來（根據微軟的說法），我們的大問題是：應該把我們的軟體開發精力放在哪個平台上？

這本書把焦點放在 Windows 95 的架構和其內部實作上。雖然 Windows NT 和 Win32s 早已行之有年，可能你們之中的大多數人並不關心 Win32 -- 直到 Windows 95 出現。微

軟的 Win32 API 策略及其作業系統格局早就在我們眼前展開三年了。如果還自稱 Windows 95 是個沒有歷史包袱的新品種，，無異是自欺欺人。雖然 Windows 95 贏得了眾人的所有目光，在微軟本部裡頭，NT 小組才是微軟作業系統的未來希望所繫。微軟企圖將 NT 和 95 合併起來，其中 NT 所佔的技術比例比 95 高得多。因此，在挖掘 Windows 95 核心之前，我要以這一章告訴各位微軟的 Win32 策略，並讓你知道 Windows 95 在其中的位置。相信我，本書其他章節都是 Windows 95 核心結構及其內部實作的硬扎貨色，而本章則是認識 Windows 95 在 Win32 API 和 32 位元程式設計脈絡中定位的重要導引。

毫無疑問我的言論不能夠討微軟的歡心。因爲他們一再說『只有一套 Win32 API。寫一個程式可以在所有作業平台上執行』。雖然聽起來不錯，事實卻不是那麼一回事。

或許討論這個問題之前，最好把 Win32 定義清楚。Win32 定義出一組由作業系統提供的函式（API），應用程式可以利用（呼叫）它來完成某些工作。這一組函式就是 Win32 API。當微軟第一次把 Windows NT 推出舞台時，許多人對 Win32 和 Windows NT 感到困惑。Windows NT 只不過是 Win32 API 的一個實作平台。然而由於它是第一個 Win32 平台，有些程式員因此在作業系統（Windows NT）和其 API（Win32）之間混淆了。

微軟對 Win32 的一個主要目標是提供很好的移植性，所以某個範圍內的 Win32 API 非常類似 Windows 3.x API（例如在視窗管理和圖形顯示方面）。

如果微軟把 Win32 限制在 Windows NT 身上，那麼 Windows 95 可就難產了。然而微軟把 Win32 API 實作於許多個作業平台上。每一個作業系統都爲了特殊的形勢和硬體環境而有自己的最佳化動作。對高檔設備而言，穩健性和安全性是最重要的主題，Windows NT 便是如此。對於低檔並配備不多記憶體的 386 機器而言，Win32s 是通往 Win32 的最佳路徑 -- 直到 Windows 95 出現。微軟的重要觀點就是，盡情寫你的 Win32 程式，它可以在任何 Win32 平台上跑。

理論上 Win32 API 的實作層應該把硬體差異和低階的系統部份都涵蓋起來。這也就是微軟所說的 "Scaleable Architecture" -- 1992 年七月第一次 Win32 developer's conference 時推展的一個觀念，。正如 Win32 之名所強調，從 Windows 3.x API 移轉到 Win32 API 的最關鍵優點就是「32 位元」。定義 Win32 API 的同時，微軟也定出一個新的 32 位元可執行檔的大綱。這個新的可執行檔格式稱爲 PE（Portable Executable）格式，係從 UNIX 系統的 common object file format V（COFF）衍生而來。Win32 API 和 PE 格式必須互相配合。所有的 Win32 平台（甚至 non-Intel 機器上的）都以 PE 格式做爲其主要的可執行檔格式。如此一來微軟就能夠保證所有的 Win32 程式都可以在 Win32 平台上執行。當然，所謂移植性也只能到此爲止了。你還是沒有辦法在 Intel 機器上跑一個爲 DEC Alpha 機器編譯的 Win32 程式（除非有非常複雜的模擬軟體）。

推出 Windows NT 後，微軟很快宣稱有另一個 Win32 API 的實作平台，名爲 Win32s。其背後的想法是以微軟提供的一組 DLLs 和 VxDs 加到現存的 Windows 3.1 機器中，使它能夠執行 Win32 程式。不幸的是，一些極被期待的 Windows NT 特性沒有辦法在 Windows 3.1 中實現出來。這也是爲什麼稱其爲 Win32s（subset）的原因。Win32s 提供了一部份而非全部的 Win32 APIs，它的主要弱點是沒有支援現代作業系統的特質，例如執行緒和分離位址空間。執行緒是高階作業系統的特性之一，允許程式中一個以上的部份同時執行（或至少看起來是同時執行）。典型的用途是以一個執行緒處理列印，另一個執行緒處理使用者介面（UI）。Win32s 也因爲 Windows 3.1 的某些限制而顯得一跛一跛。詳情請看稍後的「Win32s」一節。

和 Win32s 一樣，Windows 95 也只是完整的 Win32 API（定義於 Windows NT 中）的一個子集。微軟最初稱之爲 Win32c（c 代表 Chicago -- Windows 95 的最初代號）。Win32c 涵蓋了所有的 Win32s API，並加入相當份量的 NT APIs。Windows 95 被賦與重大的承先啓後任務，雖然它也只是一個子集，但它已經涵蓋了高階作業系統的絕大部份特徵，包括執行緒和分離位址空間（兩者都是 Win32s 缺乏的）。程式員應該會喜歡「分離位址空間」，因爲它可以阻止不好的程式破壞別人的程式或甚至系統。Windows 95 需要的記憶體比 Windows NT 少得多，很適合在低檔機器上跑。

和 Windows NT 小組不同，Windows 95 小組並不考慮不同機器之間的移植性。因爲 Intel 市場大得足夠讓微軟並行兩條 Win32 生產線。NT 小組生產一個可移植的 Win32 平台，但對於任何特定硬體都沒有最佳化。95 小組則生產一個特定用於 Intel 386 家族的 Win32 平台，並且最佳化。如果微軟沒有對 Intel 機器做最佳化，那就喪失了對其他作業系統如 OS/2 Warp 的競爭力。事實上許多人認爲 Windows 95 和 OS/2 極類似，並認爲它是 "OS/2 killer"。

不久前微軟才放棄 Win32c 這個名詞，因爲它看起來強調了 Windows NT 和 Windows 95 之間的差異性。微軟於是宣稱『只有一套 Win32，一個 Win32 程式可以在任何 Win32 平台上執行』。事實的真象是，程式員還是得考慮到 Windows 95 只實作出 Windows NT 的 Win32 API 的一部份。微軟的顧慮似乎是，如果程式員不知道對象是哪一個平台，他們便可能對 Win32 裹足不前。微軟更進一步強調「只有一套 Win32」，作法是讓 Windows NT 和 Windows 95 的應用程式都使用相同的 Win32 Logo。

當然，文飾 API 子集之間的差異完全是無意義的事。這些子集的確有差異存在，而且不算不重要。例如，軟體開發者發現他們很難獲得微軟的 Win32 Logo，因爲他們沒有辦法符合微軟所要求的「同時支援 NT 和 95」。最後微軟甚至因爲太多的抱怨聲音而改變了他們的 Win32 logo 授權策略。第二個例子是，使用多執行緒的程式很明顯不能夠在 Win32s 環境中跑，因爲 Win32s 不支援多緒多工。結果，爲了讓程式更有效率，程式員必須對不同的 Win32s 付出注意力，並了解底層作業系統如何實作。

## 1.2 Win32 作業系統定位

爲了澄清目前紛亂的 Win32 平台架構，我以一個音訊系統（圖 **1-1**）來模擬說明。爲了討論上方便起見，我們假設沒有所謂的音訊光碟，磁卡帶是唯一的錄音型式。

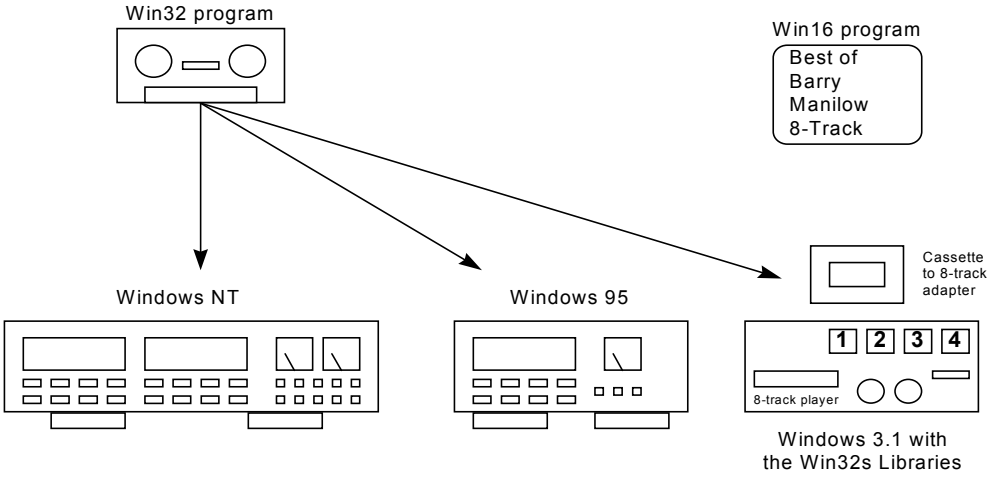



**圖 1-1　我以一個音訊系統模擬說明目前的 Win32 平台架構，並顯示它們之間的關係。**

在我的模擬之中，Win32 程式是卡式錄音帶，16 位元 Windows 3.x 程式是老舊的八磁軌錄音帶。Win32 作業系統像是一個卡式錄音座，可以播放或記錄卡式錄音帶，Windows 3.x 像是一個八磁軌錄音機，只能接受八磁軌錄音帶。

如果你是一個高級音響玩家，需要最高級的立體音響，那麼你大概會購買卡式錄音座。如果受限於預算，但是又希望聽卡式錄音帶，怎麼辦？你可以爲你的八磁軌錄音機買一個轉接器，於是就可以接聽卡式錄音帶了。在 Windows 程式設計中，Win32s 就相當於轉接器，你把 Win32 程式插入 Win32s 轉接器中，而實際上是 Windows 3.1 在服務。使用這樣的磁帶轉接器，當然也就帶來了一些限制。例如你沒辦法使用一個八磁軌轉接器錄製卡式錄音帶，聲音品質也不可能像在卡式錄音座中聽卡式錄音帶那麼理想（因爲中間有一些失真以及轉換的問題）。同樣的道理，Win32s 的能力也有限制：不支援完整的 Win32 API，也不支援像多緒多工這樣的性質。

Windows 95 扮演什麼角色？Windows 95 相當於一個入門級的卡式錄音座。它的一些元件取材自八磁軌錄音機，但是有一個很漂亮的面板。是的，Windows 95 有許多新的 32 位

元碼，但是也從 Windows 3.x 中借取了許多技術（像是視窗管理等等）。大致而言，Windows 95 能夠做到 Windows NT 的許多出類拔萃的功能，但是缺乏一些最高檔能力，如安全防護（security）和雙位元組字元集（也就是 unicode）等。然而也正因爲 Windows 95 不需要那麼高檔的能力，所以它比較便宜。它不像 Windows NT 要做那麼多服務，所以它需要的記憶體比較少，動作也比較快。

看過了這個比喻，現在我們來檢驗微軟的每一個 Win32 平台，看看它們之間的關係。

## Windows NT 코ᄀᆷᄂ

Windows NT 的主要目標在強固性與移植性（移植到其他硬體平台）。大部份碼是以 C 和 C++（而非組合語言）完成。極佳的穩定性使得 Windows NT 成爲一個理想的軟體開發平台，甚至即使你的開發目標是放在 Windows 95、Win32s、Windows 3.x 或 DOS 身上。但是另一方面來看，強固性與移植性帶來成本，需要更多記憶體才能拉動整個系統。NT 所需的最低配備是 486 機器加上 16MB RAM。在這樣的設備上，NT 不會跑得比 OS/2 或 Windows 95 更快。不過讓我持平說句話，Windows NT 3.5 比 Windows NT 3.1 好多了。

NT 之所以穩定，一個主要原因是其「受保護的子系統」（protected subsystem）架構。在此子系統中，實作出 API 的那些系統碼，係在與應用程式不同的位址空間中執行。Windows NT 中最重要的子系統便是 Win32 子系統，它有自己的行程。USER 和 GDI 碼大部份都放在一個名爲 WINSERV 的 DLL 中。當你的程式呼叫 API 函式如 *TextOut*，事實上並非直接呼叫真正的 *TextOut* 碼；NT GDI32.DLL 中的一個 stub（一小段碼）會把你的參數拷貝到一塊記憶體中，該區域可被你的行程或 Win32 子系統存取。你的執行緒然後發訊號給 Win32 子系統，告訴它有一個函式正等著被處理，然後就睡著了（冰凍了）。當 Win32 子系統看到這樣的訊號，它做出適當的處理（例如把一個字串放到螢幕上），然後通知呼叫端（執行緒）說此一函式已經完成。子系統的 client/server 模型也適用於 NT 所支援的其他作業系統，如 OS/2 和 POSIX。

「受保護子系統」的優點是，它們的位址空間可以免受記憶體覆寫以及應用程式臭蟲的侵擾。在那些沒有子系統的作業系統中（例如 Windows 3.x 和 Windows 95），作業系統的程式碼和資料都映射到所有行程的位址空間中，於是一個不良份子很可能就把整個系統搞砸了。子系統模型的缺點就是增加執行時間 -- 每次呼叫一個函式理論上要引發一個行程切換，並改變 memory context。這個代價十分昂貴，估計平均每次呼叫需要 2000~3000 個 clock cycles。爲了這個原因，NT 小組將某些最常被呼叫的函式最佳化，使它們不至於引發行程切換。此外，有些 GDI 呼叫可以批次作業，不需要每次呼叫就做一次行程切換。

所有這些用以增加 Win32 程式穩定性的措施都很棒。但 16 位元程式怎麼辦？它們在合作型多工模式下運作，並且認爲可以處理屬於其他 16 位元程式的記憶體。NT 把 16 位元程式放在所謂的 WOW（Windows On Windows）行程中跑。預設情況下所有 16 位元程式處於單獨一個 "WOW Box" 中，那基本上是個多執行緒的 DOS Box。

WOW box 很像 Windows 3.1，16 位元程式可以在其中做任何它們可以做的事。它們的動作不會影響盒子外的世界。

WOW 子系統和 Win32 子系統聯繫，以便執行顯示幕的輸出，使 16-/32- 位元 Windows 程式能夠在同一個螢幕中作用。Windows NT 3.5 甚至讓每一個 16 位元程式在單獨一個 WOW box 中跑，更增進了 Win16 程式的穩定性。

我之所以在這裡介紹 WOW 子系統，因爲它是 NT 和 95 之間的重大架構差異。WOW 的唯一缺點就是 16 位元程式在其中的速度比在同級機器下的 Windows 3.1 慢。

## Win32s 子系統

和 NT 相反，Win32s 架構在不穩定的 DOS/Windows 聯盟之上。Win32s 不是一個作業系統，而是 Windows 3.1 的一組擴充函式庫。事實上 Windows 3.1 也不是個作業系統，而是架構在 DOS 作業系統之上的一層。實作出 Win32 API 的那些 Win32s 碼讓不穩定又更加深了些，因爲它必須仰賴 VxDs 和 system DLLs。

除了極少數例外（如記憶體映射檔），幾乎可以說如果某個機能不存在於 Windows 3.1，也就不存在於 Win32s。其實 Win32s 不過就是把你的 32 位元碼下移（thunk down）至真正能夠工作的 Windows 3.1 碼而已。"Thunking" 之於程式設計，就好像用口香糖、鉛線、橡皮筋把東西綁在一塊兒一樣地不勞靠。它其實是一小段程式碼，處理 16-/32- 位元間的轉換事宜。

Win32s 的限制很大。第一個同時也是最大的限制是，它不支援執行緒。第二是它把所有 Win16 程式和 Win32 程式都擺在同一個位址空間中。由於 Windows 95 和 NT 都爲 Win32 程式準備了分離位址空間，這項弱勢使 Win32s 被歸類爲「隨時可以放棄」的平台。Win32s 中的 Win32 程式可以看到其他 Win32 程式的記憶體，以及其他 Win16 程式的記憶體，這使得記憶體被破壞的情況真的可能發生。

第三個缺點是 Win32s 缺乏每一行程專屬的 DLL 資料空間。在 Windows NT 和 95 中，DLL 的資料區在預設情況下是每個行程各有一塊。簡單地說，就是你可以安心使用 DLL 的全域變數，不必擔心其他的 DLL 呼叫者來搶奪破壞。但 Win32s 在這方面的表現和 Windows 3.1 一樣。通常你的程式及其 DLLs 在 Windows NT 和 95 中可以正常執行，到了 Win32s 和 Windows 3.1 就掛了。

Win32s 的另一個問題是行程的排程和訊息傳遞系統。在 NT 和 95 中，執行緒是強制性排程，此外每一個執行緒有自己的訊息佇列，以及自己的一個「輸入執行緒」，把滑鼠和鍵盤訊息送到適當的佇列中。這兩項設計使得一個執行緒可以從心所欲地抓取 CPU 時間，不必擔心影響到其他程式。然而 Win32s 和 Windows 3.1 一樣使用合作型多工模式。爲了讓一個程式跑起來，另一個程式必須釋放 CPU（經由呼叫像 *GetMessage* 或 *PeekMessage* 這樣的函式）。如果程式沒有獲得訊息並及時釋放控制權，使用者就沒辦法切換執行其他程式。

還有更糟的嗎？Win32s 有一種毀滅（或其他奇怪行徑）的傾向。如果你感覺出我並不認爲 Win32s 值得投資，你是對的。感謝上帝，Windows 95 終於取代它了。

## Windows 95 ㄱㄴㄷ

描述 Windows 95 Win32 平台的最好說法就是：它是更好的 Win32s。它把 Windows NT 中最好的一些性質都實現出來了。Windows 95 有充份的高階作業系統性質。它的記憶體消耗量和 Windows 3.1 + Win32s 差不多。所以 Windows 95 是 Win32s 的理想代替品。

從底層觀之，Windows 95 比較類似 Windows 3.1 + Win32s，而非 Windows NT。和 Windows 3.1 一樣，Windows 95 的最底層是 ring0 系統碼，包括虛擬機器管理器（VMM）和輔助的 VxDs。在 CPU ring0 執行的碼理論上是最穩定也最能夠信賴的。所以它負擔較多的系統資料和硬體的處理。另外，就像 Windows 3.1 一樣，Windows 95 有一個「系統虛擬機器」用來執行 Windows 程式，你所啓動的每一個 DOS session 則有個別的虛擬機器起來服務。在系統虛擬機器中，你會發現熟悉的 ring3 system DLLs：USER、KERNEL、GDI，及其 32 位元兄弟：USER32、KERNEL32、GDI32。

和 Win32s 一樣，Windows 95 有大半仰賴 16-bit system DLLs 並使用 thunking 機制，把 32 位元碼轉換爲 16 位元碼。幾乎所有的視窗系統和訊息系統碼都在 16 位元 USER.EXE 中。嘗試把巨大而複雜的視窗系統移植到 32 位元領域，會導至程式碼暴漲，而且與現有之 16 位元程式不相容。沒有一種情況能夠被微軟接受，因爲回溯相容是 Windows 95 的最高指導原則。因此，Windows 95 的視窗系統和訊息系統基本上是 Windows 3.1 的修改版。修改重點主要是允許 16 位元元件得以和 32 位元元件溝通，並加上某些新能力以便實作出 Windows 95 所支援的 Win32 函式。

32 位元 GDI API 的實體部份被分配在原有的 16 位元 GDI.EXE 和新的 GDI32.DLL 之間。只要可能，Windows 95 GDI32 函式便下移（thunk down）呼叫原有之 16 位元 GDI 碼。至於 KERNEL APIs，微軟信誓旦旦地說 KERNEL32.DLL 不會下移呼叫 16 位元的 KRNL386.EXE，但是 Andrew Schulman 在他的 ***Unauthorized Windows 95*** 一書證明了不是那麼回事。我們可以在第 3 章細細品味微軟的宣稱。

我曾經在前一節描述過 Win32s 的某些問題：沒有支援執行緒、單一位址空間、DLL 資料區沒有讓每個行程專屬一份、合作型多工。Windows 95 解決了所有問題，也就是說它

的行爲像 NT。Windows 95 的 32 位元程式可以擁有執行緒（但 16 位元程式還是不能夠），DLL 資料區也可以讓每個行程專屬一份。然而，某些部份還是打了點折扣。例如，Windows 95 的每一個行程雖然有自己的位址空間，但是 system DLLs 可以被所有行程看到，而非只被那個將它載入的行程看到。此外，給所有 Win16 程式使用的記憶體，以及 DOS 記憶體（1MB 以下）也可以被目前正在執行的 Win32 程式看到。也就是說，DOS、所有的 Win16 程式、目前正在執行的 Win32 程式，統統混在同一個位址空間中。這一點和 Windows NT 截然不同。第５章顯示，記憶體的訛誤在 Windows 95 之中仍然無可避免。不過比起 Win32s 當然是好多了。

Windows 95 的一個熱門話題就是它那並不十分平順的 Win16 多工行爲。Windows 95 真的有強制性多工能力，但是一個素行不良的 Win16 程式卻可能使其他程式阻塞在 16 位元 system DLLs（像 USER.EXE 和 GDI.EXE）的大門口。問題出在於 16 位元 system DLLs 是 nonreentrant（不可重複進入的）。也就是說它們的設計並沒有考慮到「尚未完畢一件任務時就被切換出去」的情況。由於許多 Win32 API 都下移（thunk down）至 16 位元 system DLLs，所以防止在不適當時刻發生執行緒切換是有必要的。在 Windows 95 開發初期，許多解決方案被提出來並熱烈討論。

最後決定的一個解決方法是所謂的 Win16Mutex。基本上那是一個互斥的 semaphore，當程式欲進入 16 位元的 system DLLs 如 USER.EXE 或 GDI.EXE 時即向系統索求。Win16Mutex 意味著一次只能有一個執行緒進入 16 位元 system DLLs 中。這樣，就可以解決許多問題。當一個 Win16 程式執行起來，它便擁有 Win16Mutex。糟的是如果一個 Win16 程式不適度地釋放控制權（藉由 *GetMessage* 或 *PeekMessage*），就可能妨礙 32 位元程式的 UI 執行緒的進行。

Win16Mutex 的出現暗示兩件事情。第一，愈快把你的程式移往 32 位元愈好。如果系統沒機會執行不良的 Win16 程式，當然 Win16Mutex 也就沒有機會成爲麻煩的根源啦。***Unauthorized Windows 95*** 一書曾經指出，Windows 95 不可能完全脫離 Win16 程式，因爲系統本身就使用了一兩個。然而，那些 Win16 程式在釋放控制權方面表現良好，因此對於 Win16Mutex 的釋放也是表現良好。

第二個暗示就是盡量使用執行緒。如果你的工作和時間很有關係，你或許希望把程式分裂爲數個執行緒，例如一個「UI thread」以及數個「worker threads」。Win16Mutex 不會影響那些不呼叫 16 位元碼（如 USER 或 GDI）者，它們會繼續被強制性排程 -- 甚至即使整個 UI 被素行不良的 Win16 程式綁得動彈不得。

## 微軟以外的 Win32 平台

前三小節專注在微軟提供的 Win32 平台。由於 Win32 定義完整，其他公司亦願意實作其平台。大部份人熟知的例子就是 OS/2 Warp。甚至雖然 Win32 API 直接與 OS/2 API 競爭，IBM 最新版本的 OS/2 還是支援 Win32 API 的一個子集。在我下筆的時刻，OS/2 所支援的是 Win32s。毫無疑問未來它會支援 Windows 95 API。

更可愛的是 Win32 for DOS。雖然我主要是在 Windows 95 或 NT 上跑，但我還是常常以 DOS 開機並使用 Windows 3.1。當我這麼做的時候，我恨爲什麼沒有工具讓我使用 Win32 API。幸運的是 Phar Lap 和 Borland 都有 DOS extender 產品，支援足夠的 Win32 API，讓 console 程式也能夠在 DOS 或 Windows 3.1 下執行。如果你用到任何圖形函式或訊息函式，那些 DOS extender 就沒法度了。不過通常一個 console 程式（例如我的 PEDUMP，第 8 章）就是你所要的。

Phar Lap 的 DOS extender 名爲 TNT，Borland 的 DOS extender 名爲 Borland DOS Power Pack。使用這些 extender，你可以寫一般的 C/C++ 程式並使用 *printf* 或 *fread* 等函式，不必在乎程式將在 Windows NT 或 Windows 95 或 DOS 執行。

使用 Phar Lap 或 Borland 的 DOS extender 簡單得很，只需略爲修改聯結器選項。你可以繼續使用原來的 Win32 編譯器。隱藏在那些 DOS extender 背後的觀念是，你使用了由 DOS extender 提供的特殊程式，當做 Win32 可執行檔中的 DOS stub 程式（譯註）。如果你在 Windows 95 或 NT 之下執行此程式，作業系統會忽略 stub 程式。如果你在 DOS 底下執行它，這個 stub 程式會載入 DOS extender，把提供 Win32 API 的碼帶進來。

> 譯註：不論是 Win16 的 NE 檔或 Win32 的 PE 檔，都內含一個 DOS 程式，稱爲 DOS stub 程式。你在 Windows 3.1 之下看到的 "This Program Requires Microsoft Windows" 輸出便是 DOS stub 程式的傑作。

有趣的是微軟也在它的 Visual C++ 32 位元版第一版上使用 TNT DOS extender 。由於有些程式員要爲那些沒有 Windows NT 的人開發 Win32s 程式，微軟不能夠讓 Windows NT 成爲編譯器（CL.EXE）和聯結器（LINK.EXE）的執行必要條件。藉由 TNT extender，微軟工具跑了一個 native Win32 console 程式。Borland 的主要命令列工具也是個 Win32 程式，使用 Borland Power Pack DOS extender。

## 傻瓜軟體的開發考量

如果你決定把下一個計劃放在 Windows 95 和 NT 身上，你的開發環境就很重要。如果要讓程式在 Windows 95 中正確執行，並且不用到任何 95 特性，那麼它應該也要能夠未加修改就在 NT 上執行才是。但是 Windows 95 不像 NT 那麼穩定，你可能需要比較多次的重開機（至少我是如此）。這表示，NT 是個理想的 Win32 開發平台。

在 Windows 95 或 NT 上開發軟體，是你個人的選擇。某些人憎惡像 Windows 3.1 般的 Windows NT 3.5 使用者介面，因而比較喜歡在 Windows 95 上工作。但也有些人希望在穩定的環境下工作。另一些人，像我，喜歡把系統曝露在風險之下，做一些像開發除錯器、挖掘系統資料的動作，就會愛上 NT 的強固。在我最近一兩年的工作之中，我只把 Windows NT 搞當掉過一兩次。但我也在 Windows 95 上做了不少事情，主要是因爲像 SoftIce/W 這樣的工具只有 95 版而沒有 NT 版。

## Win32 的未來

差不多在這本書出版的同時，微軟已經在開發代名 Cairo 的 NT 下一個主要版本。Cairo 採用 Win32 API，並且據聞十分地物件導向 -- 甚至其檔案系統亦是。Cairo 應該會支援類似 Windows 95 的使用者介面。由於 Cairo 是以 NT 爲基礎，其平台獨立性可望在「程

式碼增加、效率降低」的情況下達成。或許微軟認爲平均機器效能以及記憶體價格會有大幅的改善 -- 當 Cairo 到來時。

雖然 Windows 95 並不是 Intel 專屬平台的終點線，但 95 的架構恐怕只會存活數年。如果桌上型電腦的價格和能力都能夠對 Cairo 或其後代有所助益，微軟或許會停止兩個作業系統線平行發展的情況。換句話說如果使用者的硬體跟不上 Cairo 的要求，微軟會繼續開發 Intel 專屬的 Win32 平台，讓它繼續分享市場大餅。

## 摘要

旋風般地解釋「Windows 95 與其他 Win32 平台的關係，以及 Win32 的沿革」之後，現在終於要結束了。第 2 章把焦點放在 Windows 95 身上，更進一步說，它提供一個概觀，說明 Windows 95 之於 Windows 3.1 的新特質。本書的剩餘部份則是徹底挖掘 Windows 95 的細部資料。

# Windows 95 有些什麼新東西

近兩年來，人們不斷推測 Windows 95 到底是什麼。有人說 Windows 95 是 NT Lite -- 但 Windows 95 其實並不只是一個輕薄的短小 NT。也有人說 Windows 95 是注射了類固醇的 Win32s -- 雖然在兩個作業系統之間有些極爲明顯的類似，但這麼說也有欠公允。Windows 95 遠比 Win32s 龐大而且重要得多。

這一章要從程式設計以及系統架構兩個角度告訴你 Windows 95 的概觀。由於大部份使用者都是從 Windows 3.1 轉進到 Windows 95，所以我以 Windows 3.1 作爲各種比較的基準。

我所敘述的系統架構是幾乎每一個 Windows 程式都必須面對的。我的討論主要落在 KERNEL、USER、GDI 三大模組。當然我無法在有限的篇幅下將完完整整的 Windows 95 呈現給你，許多主題像是 OLE 2.0、Plug and play、MAPI（Mail API）、TAPI（Telephony API）都超出了我希望在此描述的範圍。

這一整章從頭到尾，我所描述的一些 Windows 95 的特性和架構觀念，事實上適用於整個 Win32 而不只是 Windows 95。這些性質也存在於 NT。然而由於 Windows 95 是第

一個把 Win32 程式設計觀念曝露於如此眾多程式員面前的作業系統，而我的最主要目標也是放在 Windows 95 身上，所以我在本章許多地方所說的 Windows 95，其實嚴格來說應該是 Win32 或 Windows NT and Windows 95。

Windows 95 有兩個基本條件（雖然也許有點矛盾）：

- 提供存在於 Windows NT 上的所有 Win32 API（執行緒、分離位址空間、虛擬記憶體等等等），但捨棄安全防護性（security）和 unicode。
- 在 4MB 機器上執行既有的 MS-DOS 和 16 位元 Windows 程式，效率要和在相同機器上的 Windows 3.1 一樣好，或甚至更好。

第一個基本條件表達了微軟公司對於 95 嘉年華會入場許可證的態度：並不是每一部機器都有得以執行 Windows NT 的硬體裝備。NT 是一套偉大而「毫不妥協」的作業系統，它對記憶體的需求超過了一般桌上電腦的平均值（4MB）。Windows 95 帶給使用者相當程度的 NT 功能子集，那些雖然沒有 NT 硬體配備卻也不需要 NT 或 UNIX 防彈設施（帶來大量額外負擔）的使用者咸感歡迎。由於有數百萬計的機器無法執行 NT，微軟決定放棄 NT 相容性，提供一個能夠在一般桌上電腦執行並具有足夠威力的 Win32 平台。雖然 NT 和 Windows 95 在 Win32 API 這一層十分類似，但 Windows 95 的設計牢牢綁住 Intel CPU 的 80386 家族，這把 Win32 帶往一個非常巨大的機器安裝量上。也因此，維護兩套作業系統才值得。

Windows 95 的第二個基本條件必須詳加敘述。注意，微軟並沒有宣稱你可以在一部 4MB 的 Windows 95 機器上很舒服地跑一個大程式，它是說，Windows 95 的目的在於：在一部 4MB（或更多）的機器上，Windows 95 的表現不能夠比 Windows 3.1 差勁。我想，在一部 4MB 機器上，Windows 95 要表現得比 Windows 3.1 好是頗爲困難的，但我們有理由期望至少不比較差。由於 Windows 95 不放棄 Windows 3.1 的功能，很明顯地，Win32 支援能力只好塞到那些因爲緊勒並重新調整 Windows 3.1 系統程式碼而空出來的空間。這就是 Windows 95 的設計折衷方案。

我把這一章切割爲以下四個主要段落：

- Windows 95 如何能夠和 Windows 3.1 性能一樣。
- Windows 95 如何改善既有之 Windows 3.1 性能。
- Windows 95 新的性能。
- Windows 95 的一些骯髒秘聞。

本章提供的，是對於 Windows 95 的改變和補強的高階看法。至於深入的討論將延至後續各章中出現。在那裡，我將適度地指出在本書的其他什麼地方可以獲得更多資訊。

## 與 Windows 3.1 類似之處

微軟花了許多時間讓我們相信 Windows 95 是一個新的、平地而起的作業系統。然而，你不應該相信任何「聽」來的話。如果你做一點小小的改變（稍後我將示範），你就會得到足夠的證據，證明 Windows 95 的底層其實主要是 DOS 和 Windows 系統基礎的演化。當然，Windows 95 有許多新的特質，我會在這裡以及本章其他地方訴說它們。但是爲了能夠真正了解 Windows 95，很重要的一件工作就是把誇大的宣傳擺在一旁，誠實地看看 Windows 95 的底層。

我要聲明，Windows 95 是 DOS 和 Windows 3.1 的組合。要不我就拿出證據，要不我就應該閉嘴。現在進行我們的第一個實驗，看看開機後發生什麼情況。假設你已經安裝好了 Windows 95。在重新開機之前，讓我們做一點小小改變。在你的開機磁碟的開機目錄之中，有一個隱藏的系統檔，名爲 MSDOS.SYS。如果你執行 dir /ah 命令，就會看到它：

```
C:\> dir /ah MSDOS.SYS
 Volume in drive C is MS-DOS 622
 Volume Serial Number is 3E64-17DF
 Directory of C:\
MSDOS    SYS        1,637  03-23-96  17:06 MSDOS.SYS
        1 file(s)         1,637 bytes
        0 dir(s)     148,324,352 bytes free
```

如果你曾經是 DOS 的使用者，你就不會對這個檔案有太大的驚奇。但是在 Windows 95，它竟然變成了一個 ASCII 檔。讓我們改變它的屬性，使它能夠被一個文字編輯器處理：

```
C:\> ATTRIB -r -h -s MSDOS.SYS
RHSA_ -> ___A_  C:\MSDOS.SYS （譯註：表示取消 R,H,S 屬性，只剩下 A 屬性）
```

然後在文字編輯器中你可以看到它的內容像這樣：

```
[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.)
;xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxa
... rest of file omitted...
```

你的檔案可能和上述內容輕微不同，但是我想你可以抓到重點。現在，加一行 "Logo=0" 到 [Options] 區：

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

存檔，然後把檔案屬性恢復爲原來的樣子（利用 +r +h +s）。現在重新開機。如果你有 CONFIG.SYS 和 AUTOEXEC.BAT，你會看到這些檔案被處理的過程 -- 在 Windows 95 圖形介面出現之前。少了些什麼東西？Windows 95 的 Logo 畫面是也。原來，那個 Logo 畫面是用來遮掩紛亂的開機程序用的。那些開機程序的出現可能會迷惑終端用戶。單單一行，我們就褪去了 Windows 95 親和性介面的外衣。

看起來的確是像 DOS。爲了進一步檢驗，我殺掉了我的 CONFIG.SYS 和 AUTOEXEC.BAT，再重新開機。或許我們所看到的類似 DOS 的行爲是爲了回溯相容也不一定。在沒有 CONFIG.SYS 和 AUTOEXEC.BAT 的情況下重新開機之後，我執行 MEM/DEBUG 命令，看看 1MB 位址之下有些什麼。我看到了如**圖 2-1** 的畫面。

```
 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 —
... rest of output omitted ....
```

**圖 2-1　MEM/DEBUG 命令顯示出 DOS 片斷（即使雖然想像中 Windows 95 已經不再內含 DOS）**

如果 Windows 95 真正遠離了 DOS，我們就不應該看到 DOS 的足跡。但是圖 **2-1** 中的兩行卻洩了底：

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

唔...有 5K 被記錄爲 MSDOS，有 11K 被記錄爲 IO。後者或許和 IO.SYS 有點關係，此檔案也是我們在跑 DOS 和 Windows 3.1（而非 Windows 95）時用到的。讓我們檢查看看。再次在開機目錄下使用 dir /ah 命令：

```
C:\> dir /ah IO.SYS
 Volume in drive C is MS-DOS 622
 Volume Serial Number is 3E64-17DF
 Directory of C:\
IO      SYS      225,558  11-07-95   0:00 IO.SYS
        1 file(s)       225,558 bytes
        0 dir(s)    148,324,352 bytes free
```

的確，IO.SYS 是一個大檔案。雖然它接近 220k，但是載入系統後卻只使用 DOS 記憶體中的 11K（如前所見）。11K 不是什麼大塊記憶體，但還是足以證明至少有一些類似 DOS 的碼駐留在 Windows 95 系統之中。

圖 **2-1** 還有兩行亦十分有趣：

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

這兩行似乎告訴我們有一個名爲 WIN 的程式被載入記憶體中。嘿，等一等，以前我在 DOS 提示號下進入 Windows 3.1，不就是鍵入 WIN 然後引發 WIN.COM 的嗎？讓我們看看 WIN.COM 是否還掛在 Windows 95 之中：

```
C:\>dir c:\windows\win.com
 Volume in drive C is MS-DOS 622
 Volume Serial Number is 3E64-17DF
 Directory of C:\WINDOWS

WIN      COM       22,679  11-07-95   0:00 WIN.COM
        1 file(s)        22,679 bytes
        0 dir(s)    203,374,592 bytes free
```

非常明確，WIN.COM 仍然存在於 Windows 95 之中。現在看看在 WIN 程式之後的是什麼，那是 vmm32。看起來似乎 WIN.COM 在 Windows 95 中扮演著與 Windows 3.1 中相同的角色。換句話說，WIN.COM 是把機器從真實模式（或虛擬 86 模式）切換到保護模式 Windows 環境的整個程序源頭。

讓我們在 DOS 區域中做最後一個試驗以證實我們的理論。在 CONFIG.SYS 檔中改變 DOS 的命令處理器（COMMAND.COM）。我自己喜歡的是 4DOS -- 一個和 COMMAND.COM 相容，來自 JP Software 的命令處理器，提供比 COMMAND.COM 更多的功能。爲了置換 4DOS，把這一行加到你的 CONFIG.SYS 中：

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

當我這麼做並且重新開機，我發現我置身於一個 4DOS 提示號下，而不是 Windows 95 漂亮舒適的環境中。似乎我的 4DOS.COM 不知道要在處理完 AUTOEXEC.BAT 之後啓動 WIN.COM。但是 Windows 95 套件中的 COMMAND.COM 卻知道要這麼做。天啊，看起來這似乎是這個號稱天衣無縫的整合環境的又一個縫隙。所謂的透明化，所謂的親和性，似乎是在 AUTOEXEC.BAT 的最後加上這一行就可以辦到：

```
WIN
```

現在我們身處 4DOS 提示號之下（它想必完全不知道所謂的 Windows），讓我們詢問它 DOS 版本號碼：

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

DOS 7，嗯？我猜想這應該不會造成你的驚訝。前一版 DOS 不是 6.x 嗎？如果你執行 Windows 95 的 COMMAND.COM 並且問它同樣的問題，你會得到這樣的回應：

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

真奇怪，完全沒有提到 DOS 版本。看來微軟真的不打算讓沒有技術背景的使用者知道 DOS 混雜在 Windows 95 之中。

我還可以繼續展示 Windows 95 之中存在有 DOS 的證據給你看，但 ***Unauthorized Windows 95***（譯註：Andrew Schulman 著，IDG Press 出版）對此題目探討得更詳細。如果你對此主題有特殊興趣，請看 ***Unauthorized Windows 95***。

現在讓我們看看 Windows 95 點火起動之後的情形。如果你有 WINICE（譯註：Nu-Mega 公司出品的一個極負盛名的 Windows 除錯器），請在 Windows 95 之中執行它（即使你的 WINICE 是 3.1 版亦無妨），你會以爲你置身於 Windows 3.1 中。舉個例子，Windows 95 還是架構在許多 VxDs 之上（和 Windows 3.1 一樣）。許多我們熟悉的 VxDs 都還是存在：VMM、VPICD、VTD、VDMAD、V86MMGR 等等等。當然也有不少新的 VxDs，稍後我再來談它們。此外，你依然可以在 SYSTEM.INI 的 [386enh] 區段中載入你自己的 VxDs。不過，微軟希望你把自己的 VxDs 以 Registry 的方式載入，這一點稍後我會說明。（譯註：Registry 是系統提供的一個登錄資料庫，記錄系統的所有狀態。它是一個二進位檔，而非 ASCII 文字碼）

在 WINICE 內下達 MOD 命令，你好像回到了 Windows 3.1：

```
: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
```

所有這些 DLLs 也都存在於 Windows 3.1，並且在 Windows 95 中繼續扮演它們原先的角色。同樣地，WINICE 的 HEAP 命令可以顯示給你看，16 位元的 heap 完全沒有改變。我還可以繼續提供更多的例子，用以證明 Windows 95 之中有一大塊，不僅看起來而且運作起來和 Windows 3.1 一模一樣。Windows 95 從 Windows 3.1 進化而來毫無疑

問。如果你了解 Windows 3.1，你就站在了解 Windows 95 的良好起跑點上。第７章會詳細告訴你 Windows 95 中的 WIN16 模組「類似但不完全等同」於 Windows 3.1。

讓我澄清我在這一節所說的一些話。我認爲微軟做了正確的決定，讓 Windows 95 進化自 Windows 3.1 而不是從頭開始。回溯相容是一個絕對必要的條件。雖然 Windows 95 並非百分之百相容於 Windows 3.1，但它至少是極爲明顯地在這方面表現比 Windows NT 或 OS/2 Warp 好得多。如果從頭設計 Windows 95，那麼相容性就會是工程師的午夜惡夢。再者，一個全新設計的系統其體積必然會膨脹，而一個焦點放在大眾市場的作業系統如果不能夠在平均硬體設備上執行的話，將是一件不合理的事情。微軟面對如此嚴酷的現實 -- 大部份終端用戶的設備比起軟體發展者的設備差 -- 而做的努力，其實值得我們給予掌聲。

如果你是一個作業系統純粹主義者，瞧不起 Windows 95 採行的折衷方案，那麼你可以使用 Windows NT、OS/2、或是 UNIX，但可別抱怨某些你需要的應用軟體不能夠在上面正常執行唷。老實說，批評 Windows 95 架構，我也感覺不安，然而我也曾經一絲不苟地研究過 Windows NT。我的看法是，Windows 95 和 Windows NT 都是有確實根據的作業系統，你得決定什麼是比較重要的（記憶體消耗量加上相容性，或是強固性加上安全性），並決定最適用的平台。

我的哲學是，Windows 95 和 Windows NT 都是未來數年內非常重要的作業系統，所以，我在兩者之上都投注心力。那麼爲什麼本書專注在 Windows 95？因爲我感覺到短期之內的程式設計市場，Windows 95 將大於 Windows NT。

## Windows 3.1 上的改善

即使你不在乎 Windows 95 的新性能（強制性多緒多工、受保護的位址空間等等），光只爲了它對 Windows 3.1 的改善，也值得你移植。在這一節我將以大開大闔的筆法敘述這些改善，至於細節則留給後續章節。

## DOS 已死（幾乎）

雖然你或許從沒有注意到，但 Windows 95 對於以前的 DOS/Windows3.1 組合的最大改善就是以 VxD 取代過去我們對 DOS 的呼叫。在 Windows 3.x 之中，虛擬機器管理器（Virtual Machine Manager，VMM）的作用像是一個 DOS extender。當程式呼叫 DOS 進行某些事情（例如讀檔），INT 21h 首先跳到 Ring0 WIN386，然後轉移到底層的 16 位元 DOS。在 Windows 95，一旦 VMM32.VXD 載入並運行，幾乎所有對 DOS 的呼叫都改由 VMM32 內全新的 32 位元碼負責。VMM32 由 Windows 95 中的一大堆 ring0 元件組成。VMM32 相當於 Windows 3.1 的 WIN386.EXE。

這麼做，最值得一提的效益就是，檔案 I/O 可以完全由 32 位元 ring0 碼處理，獲得戲劇性的效率改善。這些改善並不只侷限在 DOS 程式，Windows 程式如果呼叫 *_lread*，最終也是落到和 DOS 程式呼叫 INT 21h 時所使用的相同一塊 VxD 碼去。

爲了和老舊硬體裝置及其驅動程式回溯相容，Windows 95 還是會把某些危險的中斷轉移到小量的真實模式（其實是 V86 模式）DOS 碼去，這些 DOS 碼位於 Windows 95 底層，也就是我在稍早描述過的 DOS 碼。例如，當 VMM32 發現使用者要使用一個 DOS 驅動程式，它可以退回到舊的行爲模式 -- 把中斷轉移到 16 位元 DOS 虛擬機器中，使驅動程式可以完成它自己的事情。其他的中斷，例如 DOS Get Time 函式（INT 21h/2Ch）亦總是轉移到真實模式（V86 模式）的 DOS 碼去。但記住，主要的 DOS 機能都已經移轉到 Windows 的 32 位元碼了。只要再加上少量努力，微軟就可以讓 Windows 95 完全擺脫真實模式 DOS，這可能會引起作業系統純粹主義者的興趣，但卻帶來相容性的昂貴代價。如果你要那樣的東西，你應該選擇 Windows NT。

## 視窗管理系統（windowing system）

對於某些程式員而言，Windows 95 帶來的最大救贖就是視窗管理系統和繪圖系統的 32-bit heaps 了。在 Windows 3.0（以及更早），所有的視窗以及相關的資料結構都被塞擠在 USER DGROUP 中，而它受限於 64K。到了 Windows 3.1，視窗管理系統的某些資料已經被搬移到其他的 64K 節區，不過這只減輕了一些負擔而已。Windows 95 的視窗

管理系統（在 USER 模組中）使用兩個分離的 32-bit heaps 來貯存像是 HWND 之類的資料結構。於是，我們不再受限只能開數百個視窗，或是在列示清單（listbox）中只能有 8160 筆資料。在一個 listbox 中有八千筆資料其實並不是好的設計，不過如果你有你的理由，Windows 95 也會讓你滿意。

雖然 Windows 95 的視窗管理系統使用 32-bit data，你可別把它和 32 bit code 混淆在一起。所有的視窗（是的，甚至是由 32 位元程式產生出來的視窗）都是由 16 位元的 USER.EXE 管理。Windows NT 小組成員倒是有機會寫出一個全新的 Win32 碼（在 Windows NT 中相容性不比強固性重要）。 Windows 95 小組成員「修改 16 位元 USER 模組碼」的決定是否睿智？程式員之間有不少爭論。兩個重要的因素使它成爲唯一合乎邏輯的決定。我曾經在第１章討論過這個主題，這裡我要重述我的重點。

第一個理由是考慮到模組大小。如果同時存在兩份視窗管理系統，一份是 16 位元而一份是 32 位元，會爲 Windows 95 增加數百 K 的記憶體負擔。而微軟的目標是要在 4MB 機器上執行 Windows 95，因此這無法接受。還記得嗎，Windows 95 並不企圖只在軟體開發者的高檔硬體設備上跑，Windows 95 企圖在任何祖父級的 386s 機器上跑，以及在任何最近才裝滿 4MB 的辦公室機器上跑。

這就導出了第二個理由：容我率直地說，16 位元的 USER.EXE 並不是十分容易移植。重要部份都是以最佳化過的組合語言完成，此外，USER.EXE 本身也是一個不斷進化的怪物，它已經被修改、被補焊、被拉拉捏捏近一個年代了，可以想像其中包含了許多奇奇怪怪的設計。未必能有某個人把整套產品的奇行怪癖都掌握得好。如果 USER 模組被移植爲完全的 32 位元碼，我敢說所有現存的軟體全都會崩潰掉。

除了大小的限制，Windows 95 更進一步要考慮 99.44% 的回溯相容能力。NT 的視窗系統是 32 位元，並且儘可能與其 16 位元的前身相容，但是微軟仍然不敢宣稱百分之百與 Win16 相容。Windows 95 在相容性方面有更高的標準，因此把視窗管理系統保持在 16 位元的決定是合理的。

討論過高階哲學之後，讓我們下來看看視窗系統如何改變以接納 32 位元應用程式。我曾經說過 USER 模組使用兩個不同的 32-bit heaps，但這只是故事的一小部份。Windows 95 USER 事實上是以一種不尋常的佈局，使用一個混合了 16- 和 32- 位元的 heaps。就像 Windows 3.1 一樣，16 位元的 USER.EXE 仍舊使用 16 位元 DGROUP 節區，其中有一個 local heap。儲存在 local heap 中的項目有 atoms、window classes、properties（譯註）。所有你預期該有的東西都還是在 USER 的 16 位元 DGROUP 中。然而，獨缺視窗結構 -- 比較精確地說是 WND 結構（譯註）。它們到哪裡去了？啊哈，32 位元 heaps 來也。嗯...USER 一定是產生了一個特殊的 32 位元 heap 用來儲存 WND 結構，對嗎？對！但故事還沒有結束。

譯註：所謂 atom，是系統用來儲存字串並給予的一個代碼，使用於 DDE 程式設計中。所謂 window classes，就是程式員以 *RegisterClass* 完成的一份資料結構。所謂 properties，是每一個視窗可以擁有的一種儲存私有資料的區域。

譯註：WND 結構，是一個未公開的結構，內中存有視窗函式、父視窗、視窗風格...等等與視窗息息相關的資料。

如果你仔細看看 USER DGROUP 的 selector（譯註）內容，你就會發現其 limit 欄位並不接近 64K，而是遠比 64K 大。在 Windows 95 之中，USER DGROUP 的大小是 2MB+128K。Windows 95 的 32 位元 window heap 事實上包含了 USER DGROUP 節區，放在最底部。請你仔細觀想這樣的分配。USER 所使用的各式各樣資料結構，以單一一個 selector 就取用得到。處理那些「仍然放在正常 DGROUP local heap 的資料」的 USER 碼，可以繼續使用其 16-bit offsets 值，一如在 Windows 3.1。只有那些處理「位於 32-bit heap 的資料（如 WND 資料結構）」的 USER 碼，才需要改用 32-bit offsets。記住，這些 32-bit offsets 是從 USER 的資料節區算起，而不是從 32-bit 線性位址算起。

譯註：Intel CPU 保護模式定址是以 selector:offset 的形式呈現。selector 將透過 CPU 指向一個名爲 descriptor 的結構。descriptor 之中最重要的欄位就是 "base" 和 "limit" 和 "access right"。

除了有一個新的 32-bit heap 用來儲存 WND 結構，Windows 95 的 16-bit USER 還有另一個 32-bit heap，用來儲存選單（menu）和其他文字。但和 32-bit window heap 不同的是，它們的底部 64K 並沒有 16-bit local heap。這樣的觀念（把 menu 相關的項目放到另一個分離的 heap）並不是從 Windows 95 才開始。Windows 3.1 也有分離的 menu heap -- 雖然那是 16 bits。第４章會更詳細敘述 Windows 95 的 32-bit heaps。

Windows 95 移轉到 32-bit window heap 的一個立即影響就是 window handles（HWNDs）。在 Windows 3.x，一個 HWND 是一個 local heap handle，用來指出 USER DGROUP 中的一塊空間。而由於 WND 結構被儲存爲 LMEM_FIXED 區塊（譯註）所以 local handle 其實就是一個 offset 值。因此，將 USER DGROUP 的 selector 和一個 HWND 整合起來，程式就可以獲得一個指向 WND 結構的遠程指標，並可以直接處理 WND 內容。在 Windows 95 中這種情況不再允許。Windows 95 的 HWNDs 值是如 0x80、0x84、0x8C 之類的小小數值，它們並不是 offsets，而是 32-bit window heap 中各區塊的 handle。USER 模組有能力把這樣一個 handle 轉換爲 32-bit offset 並且再把它轉換回來。第４章將描述這個轉換過程。

譯註：LMEM_FIXED 是記憶體區塊的性質，表示它放在 local heap 且位置固定。

爲了讓每一個應用程式「只能看到自己」，USER 改變了維護 window classes 串列的方法。在 Windows 3.1 中，所有的 window classes 都存放在一個串列之中。你可以利用 TOOLHELP 模組提供的 *ClassFirst* 和 *ClassNext* 函式走訪整個串列並獲得每一個 window class 的名稱及其擁有者。在 Windows 95 之中，*ClassFirst* 和 *ClassNext* 還是能夠有效運作，但它們傳回來的值只包含 USER 模組在系統啓動時註冊的標準 system classes，例如 buttons。應用程式自己註冊的 window classes 被放在一個私有串列之中。每一個私有的 window class 都保留了一小部份資訊放在 USER 的 16-bit DGROUP 中，但是 TOOLHELP.DLL 完全不知道。第４章的討論將涵蓋這些 window classes 的改變。

## 訊息系統（messaging system）的改變

微軟終於在 Windows 95 中止了瘋狂的現況，提供了分離的 input message queue 給每一個行程。更精確地說，其實是提供給每一個執行緒。重要的是，不再只有單一一個 system input queue 給系統中的每一個 task 共享了。爲什麼「單獨一個 input queue」就罪大惡極呢？簡單地說，強迫所有 tasks 從單一來源取其使用者輸入訊息（例如滑鼠和鍵盤訊息）會導至它們容易受到傷害 -- 被一個沒寫好以至於不釋放控制權的行程傷害。當一個 task 作用起來，實際上等於鎖住了使用者輸入系統，直到它釋放控制權。此期間沒有其他 task 能夠取得輸入訊息。

Windows 95（以及 Windows NT）丟棄了這個過時的模型，允許訊息被立刻移交到適當行程的 input queue 中。不幸的是，備受爭議性的 Win16Mutex（稍後將有細部描述）使得 Windows 95 的行爲還是像 Windows 3.1 一樣 -- 如果 16-bit tasks 沒有及時釋放控制權（例如經由 *GetMessage* 或 *PeekMessage*）的話。Win32 行程沒有這樣的問題，它們處理訊息的過程中並不會影響其他行程。

Windows 95 將輸入訊息移交給程式的方法其實是 Windows 3.1 模型的擴充。原始的滑鼠和鍵盤訊息被滑鼠和鍵盤驅動程式的中斷服務常式放置於單一的 system queue 中。在 Windows 3.x，所有程式從這個單一的 queue 中讀取它們的輸入訊息，而一個程式讀取時會阻止另一個程式的讀取。Windows 95 有一個專屬的執行緒，也就是所謂的 Raw Input Thread（RIT），監視這個 queue，並且在訊息進來的當下就把它們移交給適當的執行緒的 queue。因此，即使某一個程式沒有釋放控制權，其他程式還是可以繼續獲得它們的輸入訊息。當然啦，Win16Mutex 和 Win16 應用程式之間還是存在著一些問題，RIT 主要是在 32 位元強制性多工環境下發揮功效。

除了爲每一個執行緒準備一個 input queue，Win32 還要求一個行程不能夠改變另一個行程的狀態或其正在使用的值。在 Windows 3.1 中，USER 維護許多個視窗系統狀態，並視之爲系統性資料。一個最主要的例子就是 focus window（譯註）。在 Windows 3.1 中，USER 有一個全域變數名爲 HWndFocus，任何 task 都可以呼叫 *SetFocus* 而把「焦點視窗」轉移過來（並因而引起 HWndFocus 內容的改變）。Window capture（譯註）以及

其他的視窗系統狀態也是如此。但 Win32 無法接受這種情況。Windows 95 之中的每一個執行緒（而不只是行程）有它自己一組視窗系統狀態變數，當你呼叫 *SetFocus*，你能夠改變的是現行執行緒（current thread）的狀態，而不是單一的全域性狀態。USER 會注意讓螢幕上的每一個視窗看起來「不逾矩」。這些狀態包括 capture window、focus window、active window，以及滑鼠游標。第４章會對此描述得更詳細一些。

> 譯註：所謂 focus window，是指擁有鍵盤焦點的視窗，也就是鍵盤訊息流往的視窗。
> 譯註：所謂 window capture，是指擁有滑鼠訊息專屬權的視窗。

除了把視窗狀態以每個執行緒爲區隔，儲存起來，Windows 95 USER 也不允許一個執行緒改變其他執行緒的視窗狀態。例如，如果你呼叫 *SetFocus*，傳給它一個其他執行緒所擁有的 HWND，USER 除錯版會給你一個警告訊息，而且這項動作也不會成功。根據傳給 *SetFocus* 函式的 HWND，USER 可以獲得擁有此一視窗之 queue，而將此 queue 與目前的 queue 相比對，於是就可以決定是否程式員企圖在執行緒之間引發視窗焦點的改變。注意：queue 之間的視窗活化（activation）動作是絕對不允許的。

既然說到了訊息和 queue，我要再補充一些。當有人在 Windows 95 中送出（post）一個訊息到某個視窗，訊息並不立刻出現在對應的 queue 之內。訊息系統把它儲存到一個串列之中，然後只有在「它們的存在可能影響 USER 做某種決定」時，才把它們傳給適當的 queue。例如，當一個 task 進入 16 位元排程器（經由 *GetMessage*、*PeekMessage*、*Yield* 等等），Windows 95 首先把這個訊息傳佈給適當執行緒的 queue 之中，如果排程器不這麼做，它就不會看到 task 有一個訊息，那麼它就不會選擇它作爲下次執行的候選人。從應用程式的層面來看，你不必憂慮訊息系統底層行爲的改變，USER 承擔了這樣的責任：確保每一件事情看起來和在 Windows 3.1 中表現的一致。

## 16- 和 32- 位元行程的互動

Windows 95 視窗管理系統之中極有趣的一部份是有關於 16- 和 32- 位元程式的視窗之間的互動關係。雖然一個 32 位元程式的視窗函式（window procedure）是 32 位元碼，

原有的 16 位元程式並不知道，也不在乎。這些程式預期任何視窗都和在 Windows 3.x 中的行爲一樣。現在，讓我們考慮 subclassing 動作。想像有一個 16 位元程式獲得了一個 32 位元程式的視窗的 HWND。16 位元程式要對 32 位元程式的視窗做 subclassing 動作，也就是把後者原本的 WNDPROC 位址儲存起來，放上一個新的 16 位元的 WNDPROC 位址取代之。如果 Windows 95 最初在這個 32 位元 WND 結構中儲存了一個 32 位元線性位址，事情就不妙了。爲了阻止這樣的問題發生，Windows 95 讓所有的視窗行爲都像在 16 位元 Windows 一樣。

另一個 USER 所做的額外努力就是訊息編碼。在 Win16，私有的控制元件訊息從 WM_USER 開始，持續遞增。其他控制元件的自定訊息可能會覆蓋相同的這些號碼。例如在 Win16 中，BM_GETSTATE 訊息被定義爲 WM_USER+2，而它和 EM_SETRECT、LB_INSERTSTRING、CB_SETEDITSEL 都相同。這種情況下，除非你知道訊息被用於哪一種控制元件，否則訊息本身不具意義。或許是爲了讓事情比較一致化，Win32 的設計者重新爲某些控制元件指定了訊息號碼，使它們落在 WM_USER 之後而又不會影響其他的自定訊息號碼。重新對應過的訊息群組是：

| 訊息群組 | 使用於 | Win32 起始訊息 |
|---|---|---|
| EM_ | Edit controls | 0x00B0 |
| SBM_ | Scroll bars | 0x00E0 |
| BM_ | Buttons | 0x00F0 |
| CB_ | Combo boxes | 0x0140 |
| STM_ | Static controls | 0x0170 |
| LB_ | List boxes | 0x0180 |

如果同一個訊息在 Win16 和 Win32 中有不同的號碼，16- 和 32- 位元視窗如何溝通？在 16- 和 32- 位元碼的轉換層（thunking layer）內，USER 把訊息轉換爲目標視窗的適當訊息號碼。如果在純粹的 16 位元程式之間傳遞訊息，就不會招致訊息重新對應的額外負擔。

然而讓 16- 和 32- 位元程式毫無痛苦地一起運作的複雜程度絕不只是在於簡單的訊息轉化而已。許多訊息使用 WPARAM 和 LPARAM 參數傳達額外的資訊。通常一個 Win16 程式的 LPARAM 內含一個遠程指標，指向一個待填充的資料區。如果一個 Win16 程式送出一個訊息給 Win32 程式，並且在 LPARAM 中放置一個遠程的 16:16 指標，會如何？再一次，USER 的轉換層（thunking layer）必須進場處理，把訊息轉換爲 32 位元視窗函式可以處理的型式。在這個例子中，轉換層把 16:16 指標轉換爲目標視窗對等的 32 位元線性位址。反方向來說，如果一個 32 位元線性位址被交給一個 16 位元視窗，32 位元線性位址就必須被轉換爲 16:16 遠程指標。這種情況下，USER 會保持一個 selector 專門做這件事，它會改變 selector 的基底位址（"base" 欄位）以吻合 32 位元線性位址。這個 selector 的 "limit" 欄位將被設定爲 0xFFFF。

除了 16 位元遠程指標與 32 位元線性位址之間的轉換，Windows 95 的視窗管理系統也需要負責參數的轉換 -- 當訊息在 16- 和 32- 位元程式之間被交換的時候。稍早我曾經說過有一些訊息必須在 Win16 和 Win32 程式中被轉換。訊息的 WPARAM 參數也需要轉換。在 Win32 中 WPARAM 也是 32 位元（在 Win16 中它是 16 位元）。一般情況下，把一個 16 位元訊息轉換爲一個 32 位元視窗可用的型式時，USER 通常把 0 放到 32 位元 WPARAM 中較高的 16 個位元。如果是相反方向，USER 就砍掉 32 位元 WPARAM 中較高的 16 個位元。這個規則有一些例外，但是我們不必在概觀性質的這一章中太深入。

## Win16Mutex

雖然 Windows 95 的執行緒排程器是強制性的，排程動作還是會受到「單一執行緒、一次只能一個」的 16 位元碼（例如 USER.EXE 模組）的影響。一個 Win32 行程可以產生出一些執行緒，既不呼叫 *GetMessage* 也不處理使用者的輸入。例如讓一個執行緒計算 PI 值（3.14159265...）到小數點 50 位。這種不做任何使用者介面活動的執行緒只被 32 位元的執行緒排程器（位於 VMM32 內）管轄。執行緒排程器持續以強制性切換方式在這些執行緒中輪流排程 -- 甚至即使有些東西因爲受到「與使用者介面有關之執行緒」的影響而阻塞。不幸的是，16 位元程式沒有辦法生育出另外的執行緒，所以它們沒有辦法

受益於強制性多工。

就在一秒鐘之前我才提到過「被阻塞的使用者介面執行緒」。我指的是什麼呢？執行緒不是能夠被強制切換嗎？這個問題的答案牽扯到聲名狼籍的 Win16Mutex。此刻，「Windows 95 是 16 位元和 32 位元的混合體」一定會使你頭痛欲裂。導出 Win16Mutex 解決方案的原始問題是，16 位元的 USER 和 GDI 碼沒辦法強制性多工。它們假設它們絕對不會以任何理由被中斷。切換到其他 tasks 的動作應該發生在一些良好設計過的地方。在 USER 和 GDI 模組中有爲數十分眾多的全域變數。如果 Windows 完全忽略這個問題，執行緒就可以被切換 -- 即使現在正執行到 USER 或 GDI 函式的中心。由於 16 位元碼並不預期會發生這樣的事，所以系統會立刻崩潰掉。

既存的 16 位元碼沒有爲強制性多工做準備，此現象不獨出現在 Windows 身上，數以千計由第三者開發的 DLLs 也完全沒有考慮到強制性多工。因此，即使微軟有神奇魔法可以解決 USER 和 GDI 的問題，其他那些 DLLs 還是會造成 Windows 系統受傷 -- 當執行緒的切換在一個不恰當的時間發生。

解決方法之一就是找出 USER 和 GDI 中易受傷害的區域，並以同步機制如 critical sections 保護之。在 USER 這麼大的模組中做這樣的事情，很容易造成錯誤並浪費時間。更重要的是，如果在 USER 和 GDI 中從頭到尾安置同步化程式碼，會使模組增胖不少。別忘了程式碼大小是 Windows 95 開發小組念茲在茲的主題。所以，在整個模組中加上 critical sections 或 mutexes 並不是可被接受的解決方案。至於 NT，資源需求不是如此窘迫，所以視窗系統和繪圖系統完整受到 critical sections 的保護，所以它們可以「重進入」（re-entrant）。

微軟對於強制性多工的執行緒問題的解決方法就是 Win16Mutex。基本上那是一個 mutual exclusion semaphore，涵蓋系統的整個 16 位元區域，這些區域是「一旦它們執行而卻在中途發生了執行緒切換時，就會導至大災難」的區域。由於 32 位元視窗系統和圖形系統大多是呼叫其 16 位元夥伴，所以甚至 Win32 執行緒也會受到影響 -- 當它們正在執行與使用者介面相關的動作或是其他會轉換到 16 位元碼的動作。如果執行緒沒有執行

與使用者介面相關的動作或是其他會轉換到 16 位元碼的動作，那麼它們就不會擁有 Win16Mutex，因而也就繼續可以被強制性排程。

無論什麼時候，只要執行緒執行了 16 位元碼，它就擁有 Win16Mutex。這東西阻止其他執行緒進入 16 位元的 USER 和 GDI 碼，直到鎖定被解開爲止。16 位元執行緒釋放 Win16Mutex 的時機是在它們釋放控制權（例如呼叫 *GetMessage*）時。而接續執行的那個執行緒則獲得 Win16Mutex。記住一件重要事情：16 位元執行緒在它的整個執行時期內都擁有 Win16Mutex，而不只是當它呼叫作業系統的那段時刻。

雖然 Win32 執行緒只在呼叫作業系統函式時才片刻擁有 Win16Mutex，但所有的 Win16 程式卻在它們的整個生命期都擁有 Win16Mutex -- 甚至即使它們只是自顧自地計算著 50 個小數位的 PI 值。也就是說一個未把 Win16Mutex 釋放出去的 16-bit task 會阻止其他執行緒取得 Win16Mutex。這裡所謂的其他執行緒，不論是 16- 或 32- 位元行程，都會被懸住未能執行，直到 Win16Mutex 被釋放出來。也就是說，一個 16 位元程式如果不釋放控制權，就會鎖住其他任何 16- 或 32- 位元程式，

一個應用程式如果不能夠敏捷地處理訊息並適時地釋放控制權，就可能帶來一些問題。Windows 95 的新特質是強制性多工，然而，對於任何會執行到 16 位元碼（像是 USER.EXE）的程式，Win16Mutex 卻是強制性多工的一個瓶頸。擺在眼前的現實是，一個未能寫好的、對 32 位元程式帶來不利影響的 16 位元程式，可能會使 Win16Mutex 的設計成爲一個大笑柄。

雖然是如此地可憎，Win16Mutex 卻幾乎不會對只含 32 位元程式的系統帶來什麼不好影響。（雖然 Windows 95 總是有一個或兩個 16 位元程式在跑，不過它們都是背景行程，不會攫取 Win16Mutex 並把持它。）32 位元程式還是可能要求 Win16Mutex，但只有在它們轉換至 16 位元碼（如 USER 或 GDI）以執行與使用者介面相關的動作時才會。USER 和 GDI 碼理論上會很快執行完畢並立刻釋出 Win16Mutex。一般而言，沒有任何 32 位元執行緒會貪婪地佔有 Win16Mutex 一段明顯時間。當然啦，你可以圖謀不軌，推翻這個規則。如果擔心 Win16Mutex 影響你的 Win32 程式，你可以產生一些執

行緒，只要保證不呼叫到 16 位元碼如 USER 和 GDI，這些執行緒就絲毫不受 Win16Mutex 的影響。

Win16Mutex 對系統的影響很簡單就可以描述清楚。如果有 16 位元程式在執行，Windows 95 應用程式的使用者介面的多工表現就還是像 Windows 3.1 一樣。如果沒有（非系統的）16 位元程式在執行，那麼使用者介面的多工表現就會十分順暢，像在 NT 中的表現一樣。結論至爲明顯：所有新的程式請以 32 位元撰寫，舊的程式請儘快移植到 Win32 環境中。

對 16 位元碼說 No，對 32 位元碼說 Yes，就對了！

## Windows 95 GDI

Windows 95 圖形系統（GDI）是一個混合了 Windows 3.1 16 位元 GDI 和新的 32 位元 GDI32.DLL 的圖形系統。一般而言，如果一個 GDI 函式存在於 Windows 3.1，它也應該存在於 Windows 95 的 GDI.EXE。新函式像 Beziers、paths、以及改良後的 metafile 支援函式，被加到原來的 GDI.EXE 之中。其他的新函式像 TrueType 字形調整器以及列印子系統則放在 GDI32.DLL 之中。

和 USER.EXE 一樣，Windows 95 的 GDI 有 16 位元碼，使用 32-bit heap。Windows 95 GDI 使用 32-bit heap 儲存 regions 和 fonts。和 USER 一樣，32-bit GDI heap 的最前端 64K 也是 16-bit GDI DGROUP。GDI 物件除了 regions（譯註）之外，都還是儲存在這 16-bit GDI DGROUP 中。這意思很明白，你不能夠貪婪地產生數以噸計的 GDI 物件。

譯註: regions 是一種 GDI 資源，就像 pen、brush 也是一種 GDI 資源一樣。

Windows 3.x 中一個最有名的限制就是，你只能同時擁有五個 DCs（Device Context）。如果一個程式攫取了所有五個 DCs，其他程式就別想繪圖了，系統也常常因此而不穩定。在 Windows 95 之中這個束縛已經被解放了。

由於大部份的 Windows 95 GDI 都還是 16 位元碼，所以其座標系統也還是 16 位元。雖然 Win32 API 以及 Windows NT 都說 32 位元座標是基準，Windows 95 事實上只對任何交付給它的座標值的底部 16 位元感興趣。

Windows 95 GDI 固守其過去的 16 位元碼的另一個地方就是驅動程式。當 GDI 要顯示某些東西在螢幕上或其他設備上，GDI 呼叫的是一個 16 位元驅動程式（一個 DLL）。雖然所有的 Windows 95 新的部份都期望獲得 32 位元 PE（Portable Executable，譯註）驅動程式的支援，GDI 還是必須回溯相容於原有的 16 位元螢幕驅動程式和印表機驅動程式。這並不是說 16 位元驅動程式就必須把它們自己完全幽閉在 16 位元之中，許多高效率驅動程式已經使用 32 位元指令 -- 甚至即使它們還是保持在 16 位元 NE（New Executable，譯註）的 DLL 格式。

譯註：PE 是 32 位元可執行檔的檔案格式。
譯註：NE 是 16 位元可執行檔的檔案格式。

## 系統資源的清除

Windows 95 爲每一個 task 準備了一個位址空間。這麼做的理由之一是爲了資源的清除。由於某些奇異的理由（例如空間的考量），先前的 Windows 並不把應用程式所使用的 USER 資源（如 icons）附貼上「擁有者」的標籤。一旦 task 結束，USER 沒有任何資訊可以得知誰擁有這些資源，所以它也就沒有辦法在 task 結束之後把資源清乾淨。重複執行那些「不把屁股擦乾淨」（譯註：抱歉，我使用這麼俗的形狀詞）的程式，會使系統最終耗盡其 heap，於是後續程式就再也不能執行了。這是 Windows 的「阿基里斯腱」（譯註），也是 Windows 還未能全面攻佔市場的一個主要因素。Windows 95 向前跨了一大步（雖然是珊珊來遲的一大步），把每一筆資源都和其擁有者產生關聯。當擁有者（行程）結束，Windows 迭代尋訪所有資源，把未被釋放的資源釋放掉。

譯註：阿基里斯腱的意思相當於金鐘罩中的罩門 -- 最脆弱的地方。

但是這樣的資源回收系統得注意失算的地方。在 Windows 3.1，一個 task 可以配置一份資源並把其 handle 傳給另一個 task 使用之。即使原主兒結束執行了，第二個 task 還是可以使用那份資源。前面所說的 Windows 95 清除垃圾的作法於是可能導至大災難。爲了回溯相容，當 Windows 95 打算釋放一個已經結束之行程所擁有的資源時，它會先檢查那是個什麼種類的行程。如果它是一個 16 位元程式並且沒有記號顯示它與 Windows 4.0 相容，Windows 95 就保留這份資源，直到沒有任何 16 位元程式執行爲止。這樣子 Windows 95 就不會殺掉一個可能還被使用的資源了。

如果 Windows 95 確能清理系統資源，那麼會對聲名狼籍的 Free System Resource（FSR）帶來什麼影響？在 Windows 3.1 中，FSR 顯示於【About】對話盒，它是根據四個 heap -- 三個 16 位元 USER heaps 以及一個 16 位元 GDI heap -- 的自由空間而決定。四個 heaps 之自由空間百分比最小者即爲 FSR。因爲 Windows 95 擁有 32-bit heaps，計算方法必須改變。大部份情況下，FSR 並不因爲 32-bit heap 的出現而改變，因爲這些 heaps 不可能有比 16 位元之 USER DGROUP 或 16 位元之 GDI DGROUP 更小的自由空間百分比。然而，因爲將某些極耗空間的物件移出 16 位元 USER DGROUP 和 16 位元 GDI DGROUP，Windows 95 還是讓 FSR 的下降率緩慢一些。第 4 章對於 FSR 的計算有詳細的探討。

## 減少 1MB 之內的記憶體消耗

終於，我們要面對聲名狼籍的 "insufficient memory to load this program" 訊息了。好消息是，微軟已經消除了 1MB 以內的問題。在 Windows 3.x 中，DLLs 的 FIXED 節區，以及 *GlobalPageLock* 所獲得的節區，都是坐落在 heap 的低位址端。通常這意味它們是在 1MB 之內。這些節區會吃掉許多 1MB 內的記憶體，導至 Windows 沒有辦法開啓其他的 tasks（每一個 task 需要至少 1MB 之內的 512 個位元組，用做其 task database）。請參考我在 ***Microsoft Systems Journal*** 1995 年五月的 Q/A（問答）專欄，那裡有對此問題的詳細描述。在 Windows 95 之中，FIXED 節區和 *GlobalPageLock* 節區仍然坐落在 heap 的低位址端，但是不再必須位於 1MB 之內。雖然還是可能有某種乖張情況使得因爲 1MB 內的記憶體不足而無法執行更多應用程式，但是我想正常情況下這將是很罕見的了。

# 全新的性能

截至目前，我已經討論過 Windows 95 與 Windows 3.x 相同或改善過的一些性質。現在是看看全新性質的時候了。可想而知，有許多性質和 Windows NT 類似。然而對於絕大多數程式員以及終端用戶而言，Windows 95 才是第一個揭露出這些性質的作業系統。

## Windows 95 的 Win32 實作部份

以程式員的角度來看，Windows 95 帶來的最大消息就是 Win32 API。微軟希望 Win32 API 能夠使程式具備移植性。理論上一個以 Win32 API 完成的程式應該能夠毫不修改地在其他平台（如 Windows NT）上執行 -- 只要對方也支援 Win32 API 並使用相同的 CPU。一個 Win32 程式也應該在重新編譯之後就能夠在不同 CPU 的 Win32 平台上執行。Win32 API 如何有效地消除作業系統之間的差異，將是未來數年的研討主題。

當我第一次聽說 Windows 95 支援 Win32，第一個浮現的問題就是：它像 NT 還是像 Win32s？當我在其上工作兩年之後，我要說它是一個「比較好的 Win32s」。和 Win32s 一樣，Windows 95 有 32-bit system DLLs，並且被下移（thunk down）到對應的 16-bit DLLs。大部份對視窗系統和訊息系統的呼叫都會被下移到 16-bit USER.EXE 去，而大部份對 Win32 圖形系統的呼叫都會被下移到 16-bit GDI.EXE 去。但 Windows NT 卻有不折不扣的 32 位元 USER 和 GDI 模組。16 位元程式如果在 Windows NT 上跑，它們的 API 呼叫會被 Windows On Windows（WOW）轉換層上移（thunk up）到 32 位元的 USER32 和 GDI32 中。

雖然 Windows 95 比較接近 Win32s 而不是 Windows NT，但它的前途肯定比 Win32s 光明。Win32s 受限於必須在原有的 Windows 3.1 之上建構。Win32s 的開發者沒有辦法改變 Windows 3.1 的碼，因爲 Windows 3.1 已經有數百萬套的安裝量，只爲了支援 Win32 而要求他們全部升級到新版，並不是好主意。所以，Win32s 比起 Windows 95 或 Windows NT 有更嚴格的桎梏。

至於 Windows 95 小組成員，則奢侈地享受了可以修改也可以接受底層基礎的權力。從 Windows 3.1 的基礎開始，不論是 ring0 元件（VMM32.VXD 中的 VMM 和 VxDs）或是 ring3 元件（KRNL386、USER、GDI）都被修改以適當地支援 Win32 system DLLs（如 KERNEL32.DLL、USER32.DLL、GDI32.DLL）。基本上可以說，Windows 95 有大部份的 NT 性質，但在實作上傾向於 Win32s。對於大多數使用者，Windows 95 提供了速度、記憶體用量、性能、以及系統穩定度上的最佳折衷。

「仍然有 16 位元碼存在於 Windows 95 之中」的事實並不就意味著它們完全不知道 32 位元性質。舉個例子，KRNL386.EXE 之中常會呼叫 KERNEL32.DLL，以協助 16 位元的 USER 和 GDI 做行程管理以及使用 32-bit heap。第 6 章對此有更詳細的介紹。

## Windows 95 的 Win32 system DLLs

Windows 95 的 Win32 API 是以一些 16- 與 32- 位元 DLLs 混合組成的。**表 2-1** 列出一些常用的 Win32 API DLLs 以及它們的實作方式。你可以從這個表看出，不管是否合理，微軟嘗試使用原有的 16 位元碼（經由轉換機制）。

這種作法有兩個好處。第一，16 位元碼一般而言比其 32 位元兄弟小。第二，16 位元 Windows 3.x 已經在真實世界中經過了考驗。一個重新寫過的 32 位元 system DLL 需要再經過無數的測試和修正，或許會延誤 Windows 95 的上市時間。16 位元 USER.EXE 中的視窗系統已經成熟了，而其大部份的奇行怪癖也已經被充份了解。如果微軟重新以 32 位元改寫之，勢必又要產生一大堆的複雜行爲，增加開發者許多的負擔。

NT 開發小組選擇重新寫一份 32 位元 USER，因此犧牲了一些與 16 位元碼的相容性。NT 的設計標準允許這種情況發生。但 Windows 95 不允許。回溯相容是 Windows 95 最重要的基本條件。

**表 2-1　Windows 95 32 位元 system DLLs 的一部份**

| DLL 名稱 | DLL 目的 | DLL 如何實作出來 |
|---|---|---|
| KERNEL32.DLL | Win32/Windows 95 核心服務 | 大部份是 Win32 碼，不過許多是呼叫到 VxDs。有一部份呼叫到 KRNL386.EXE。 |
| USER32.DLL | 視窗管理函式 | 大部份都下移到 16 位元 USER，少部份在 USER32.DLL 中實作出來。 |
| GDI32.DLL | 繪圖函式 | 大部份都下移到 16 位元 GDI，Truetype 繪製器以及列印相關的碼放在 GDI32.DLL 中。 |
| ADVAPI32.DLL | Windows Registry | 大部份都是 Win32 碼，但若與 Registry 有關則呼叫 VMM VxD。 |
| OLE32.DLL | OLE 2.0 bBase DLL | 統統都是 32 位元碼。 |
| COMDLG32.DLL | 通用對話盒（common dialogs） | 大部份都是 32 位元碼，但少部份會下移（thunk down）至 16 位元碼。 |
| SHELL32.DLL | Windows 95 shell library （32 位元） | 大部份都是 32 位元碼，但少部份會下移（thunk down）至 16 位元碼。 |
| LZ32.DLL | LZA 檔案解壓縮 | 下移（thunk down）至 16 位元碼。 |
| VERSION.DLL | version-stamping library | 下移（thunk down）至 16 位元碼。 |
| WINMM.DLL | 多媒體函式 | 16- 和 32- 位元碼的混合 |

## Windows 95 的 ring0 元件

在 system DLLs 層面之下，我們將遭遇 ring0 元件，包括虛擬機器管理器（VMM）和虛擬裝置驅動程式（VxDs）。Windows 3.1 的這些元件統統集中在 WIN386.EXE 內。Windows 95 的這些元件還是集合在一起，不過檔案叫做 VMM32.VXD。表 **2-2** 和表 **2-3** 列出在 VMM32.VXD 和 WIN386.EXE 中標準 VxDs 的改變。

**表 2-2 Windows 95 VMM32.VxD 之中新的 VxDs 檔案**

| VXD 名稱 | VXD 目的 |
|---|---|
| CONFIGMG | 組態（configuration）管理器（plug & play） |
| DYNAPAGE | 分頁管理器 |
| IFSMGR | 可安裝之檔案系統管理器 |
| IOS | I/O 監督（取代 BLOCKDEV） |
| PERF | 組態（configuration）/ 狀態（status）資訊 |
| SHELL | Shell support |
| SPOOLER | Local spooler |
| VCACHE | 磁碟快取（disk cache） |
| VCDFSD | CD 檔案系統 |
| VCOMM | COMM 驅動程式 |
| VCOND | 主控台（console）設備 |
| VDD | 顯示器 |
| VDEF | 未定義 |
| VFAT | 檔案配置表格（File Allocation Table）協助者 |
| VFBACKUP | 給備份軟體使用 |
| VFLATD | 平滑記憶體（flat memory）裝置 |
| VMM | 虛擬機器管理器 |
| VMOUSE | 滑鼠裝置 |
| VPD | 印表機裝置 |
| VSHARE | 檔案的 SHARE 支援 |
| VTDAPI | 虛擬計時器裝置 |
| VWIN32 | Win32 裝置 |
| VXDLDR | VxD 載入器 |

**表 2-3　不再出現於 Windows 95 中的 VxDs**

| VXD 名稱 | VXD 目的 |
|---|---|
| BLOCKDEV | Block Device（被 IOS 取代） |
| CDPSCSI | SCSI CD 裝置 |
| PAGEFILE | Pagefile 裝置（被 DYNAPAGE 取代） |
| QEMMFIX | 針對 QEMM 的修正 |
| VDDVGA | VGA 顯示裝置 |
| VFD | 軟碟機裝置 |
| VNETBIOS | Netbios 裝置 |
| VDCTRL | Western Digital 快速硬碟 |
| WIN386 | 被 VMM 取代 |
| WSHELL | 舊的 shell 裝置 |

VMM32 之中最有趣的新增產品就是 VWIN32。VWIN32 並不是一個裝置（device），它是 16 位元的 KRNL386.EXE 和 32 位元的 KERNEL32.DLL 執行某些低階基本動作時需要用到的 rint0 碼。Windows NT 之中最接近 VWIN32 的就是 NTDLL.DLL 了，那是一個未公開的、但明顯內含許多低階作業系統性質的東西。

VWIN32 VxD 和 VMM VxD（以及其他一些 VxDs）都提供有 ring3 程式可呼叫的函式（又稱爲 Win32 VxD services）。許多 KERNEL32 的動作十分依賴 VWIN32（或甚至 VMM）的 Win32 VxD services，這些動作包括作業系統的基本動作，如執行緒的產生、同步化物件的阻塞、新的 memory context 的產生等等。我將在第６章介紹 VWIN32 以及 Win32 VxD services。

Windows 95 開發小組用來降低記憶體消耗量的一個方法是，改善 VxD 架構。Windows 95 支援可動態載入的 VxDs。在 Windows 3.x 時代，VxDs 必須於系統啓動時一併載入，並且在系統生命期間一直存活著。Windows 95 可以在必要時候載入或踢出 VxDs，有點像是應用程式只在列印時候才載入印表機驅動程式一樣。新的 VxD 架構也支援可分頁

（pageable）的 VxDs。你的 VxD 之中一些不常用到的部份可以設計爲 pageable，於是它們在被需要時才會被載入記憶體。

移植原有的 Win16 碼，必須考慮中斷以及中斷處理常式。Windows 95 的 Win32 程式不可以安裝中斷處理常式。它們不但不能藉由中斷與其他程式溝通，也不能夠和 Win16 DLL 內的中斷處理常式聯繫。使用中斷的程式碼大部份是爲了和硬體通訊。微軟建議你寫一個 VxD 實作出中斷程式碼，應用程式則可以經由 *DeviceIoControl* 函式和 VxD 溝通。如果你需要呼叫某些中斷函式（例如 INT 21h 或 INT 31h），VWIN32 VxD 提供了一些常式，用來引發那些中斷。

## 行程管理（Process Management）

Win16 稱呼一個執行中的程式爲 task。任何時候，一個 task 的程式碼只執行一份。也許在你明白所謂執行緒之後，會更清楚這一點（稍後我會詳細說明）。Windows 16-bit KERNEL 將每一個 Win16 task 的資訊記錄在對應的一個名爲 Task Database（TDB）的節區中。這個節區的 selector 是一個 HTASK，可以被當作參數傳給那些「需要知道你的 task 是哪一個」的 API 函式。

Windows 95 中的 32 位元程式在這方面有什麼改變？是的，一個執行中的程式不再被稱爲 task，而是稱爲 process（行程）。每一個行程在它自己的位址空間中跑，這對於那些正把 Win16 碼轉移到 Win32 環境的程式員而言，代表著一些極大的牽連。我將在「Windows 95 的位址空間」那一節敘述這重大的分歧。目前，你只要這麼想：「一個行程完全看不到另一個行程的存在」就好了。它們可以看到自己的記憶體以及作業系統的資源，但是它們看不到其他行程及其記憶體。把行程區隔開來的理由是，如此一來老鼠屎才不會壞了一鍋粥。

行程的區隔程度是如此徹底，因此在 Win32 程式中，*WinMain* 的參數之一，hPrevInstance 的值總是 0，甚至即使這個程式有另一個個體（instance）正在執行。行程認爲它自己是系統中唯一的程式。當然啦，如果你真的要和其他行程通訊，還是有辦法的。

每一個 Windows 95 行程在系統中有一個獨一無二的識別碼，這個碼被稱爲 process ID。程式可以藉由 *GetCurrentProcessId* 獲得它的 process ID。process ID 大概是 Win32 中最接近 Win16 HTASK 的東西了。在 NT，process ID 並不明確地對應到什麼系統資料結構，其典型的 process ID 是像 74、77、84 這樣的數值。在 Windows 95 中，process ID 有較大的值並且看起來更隨機。你將在第３章看到，process ID 可以透過一個魔術般的轉換過程轉成一個指標，指向一個名爲 process database 的結構；KERNEL32.DLL 就是用它來管理並追蹤每一個行程。

通常我們並不會用到 process ID。一些與行程有關的 API 期望獲得的是一個 HANDLE 參數，或稱爲 hProcess。一個 hProcess 並沒有直接關聯到某個類似 Win16 task database 的東西。和 process ID 不同的是，系統之中可以存在多個不同的 hProcess 值，都對映到同一個行程。

### KERNEL32 object handles

Win32 之中到處充斥著 handles。Handle 是你從系統獲得的一小塊數值，你可以把它交給 API 函式 -- 當你需要某些服務時。理論上來說，handle 值對於應用程式是沒有意義的，只有作業系統才知道怎麼解釋它。不過，你的 Win16 程式或許知道，因為 Win16 程式中幾乎所有的 handles 都可以被視為一個 selector 或是一個近程指標。

當你使用 KERNEL32 API，你所獲得的 handles 大多是我所謂的 KERNEL32 handles。KERNEL32 handles 有特殊屬性，例如它可以被交給 *WaitForSingleObject* 函式。KERNEL32 handles 包括有行程和執行緒的 handles，檔案的 handles，mutex handles...等等等。第３章將敘述各式各樣的 KERNEL32 handles。

一個 KERNEL32 handles 只有在被一個行程擁有時，才是合法的。在一個行程中企圖使用另一個行程的 handles 並沒有意義。雖然理論上 handles 是個不透明體，但如果你對行程相關的資料結構有足夠的認識，你就有可能把一個 handle 轉換為一個有用的指標。第３章會告訴你轉換的方法。

Windows 95 之中與行程有關的最基礎函式當屬 *CreateProcess*，類似 Win16 的 *WinExec* 和 *LoadModule*。這兩個函式還是存在，但表面之下它們其實正是呼叫 *CreateProcess*。如果你在誕生出新行程之後還要操控它，那麼你一定得用 *CreateProcess*，因爲它會傳回一個 hProcess 給你。

*WinExec* 和 *LoadModule* 不會傳給你一個 hProcess。事實上，在這兩個函式呼叫了 *CreateProcess* 之後，它們立刻關閉了 *CreateProcess* 所傳回的 hProcess。這樣做是爲了避免配置給該行程一些沒有必要的系統資源。記住，關閉一個 hProcess 並不意味結束這個行程，而只是說，你放棄經由這個 handle 來處理這個行程。當行程結束，作業系統會負責清除與行程相關的資源，而所有未能解決的 handles 也都將被關閉。

產生行程，可以獲得一個 hProcess。另一個方法就是根據一個合法的 process ID 呼叫 *OpenProcess*。不論哪一種方法，有了一個 hProcess 在手，你就可以進行某些基本動作。一個程式可以利用 *TerminateProcess* 結束另一個行程，也可以利用 *SetPriorityClass* 影響其他行程的執行優先權。

學習 Windows 如何將某些 KERNEL object（例如 tasks 和 modules）在 16 位元和 32 位元圍牆的兩邊複製出來，是十分有趣的。每一個 Win32 行程都有一個 16 位元的 task database（TDB）鏈結到 TDB 串列之中。如果你利用 TOOLHELP 走訪整個 task 串列，你就會看到除了 Win16 tasks，還有每一個正在執行的 Win32 程式。或許你應該回憶一下：TDB 接近底端之處有八個位元組，用來儲存此一 task 的模組名稱。

除了 16- 或 32- 位元行程各有 TDBs，Windows 95 中的每一個 TDB 也有一個對應的 PSP。但是和 Windows 3.x 不同，Windows 95 TDB 的 PSP 不需要立刻緊跟在 TDB 記憶體之後。在長度爲 100 位元組的 TDB 和 PSP 之間，是一個存放目前目錄名稱的區域。這個區域用來放置長檔名和路徑是十分充裕的。Windows 3.x 中的目前目錄存放在 TDB 中的 65 個位元組內。第７章對此有更詳細的敘述。

## 執行緒管理（Thread Management）

執行緒是令人興奮的 Windows 95 新性質。一個執行緒是一段程式碼的執行實體。簡單地說，執行緒使得程式得以同時執行程式碼的不同部位。就好像有許多個 CPUs 一樣，每一個 CPU 執行程式的一部份。在一個單一 CPU 系統（Windows 95 只支援單一 CPU），執行緒們只是「看起來」同時執行，事實上 Windows 95 排程器在所有的執行緒之間切換著 CPU，這就是所謂的 timeslicing。由於硬體內建的計時器會以固定週期通知作業系統，因此作業系統可以選擇執行另一個執行緒。我必須告訴你，雖然 16 位元程式也出現於系統的執行緒串列中，但只有 Win32 程式才能夠在其行程之中產生新的執行緒。

一個執行緒可能因爲兩個因素而被切換掉。一個因素是這個執行緒需要另一個執行緒完成某些事情。這種情況下執行緒會釋放控制權給其他執行緒（這是在 system DLLs 層面中發生的，你不必操心）。第二個因素是當執行緒用完了它分配到的時間。Windows 95 執行緒排程器使用一個精巧的演算法，把大部份 CPU 時間給予最迫切需要的執行緒。CPU 使用硬體計時器週期性地中斷作業系統。在硬體計時器的中斷服務常式中，排程器決定是否要讓另一個執行緒執行。如果答案是 Yes，它就進行切換工作。Windows 95 的 timeslice 是 20 個毫秒（milliseconds），意思是排程器理論上可以在一秒鐘之內切換 50 個執行緒。對絕大多數人而言，這已經代表了多工。

每一個執行緒關係到一個行程。當作業系統產生一個新的行程，它也爲它設立了一個最初的執行緒。執行緒執行於其相關行程的 memory context 之中，所有執行緒都共享其所屬行程的所有資源。在接下來的討論之中，我所謂的資源係指作業系統供應的某些東西，而非一般狹義所指的對話盒啦、游標啦等等東西。行程的資源包括 memory context、file handles、current directory 等等。

一般而言行程並不改變另一行程的資源，然而執行緒卻有可能在使用行程的資源時引發衝突。你的程式可能有一段碼用來更改全域變數的值，如果在執行這段碼的中途，執行緒被切換走，而下一個執行緒用到這些全域變數，那麼它們就處於一種不一致的狀態了。

要讓多執行緒程式能夠成功順利，你必須鑑定哪些行程資源可能會被弄壞掉 -- 如果在操作它們的途中遭受了執行緒切換動作的話。這樣的資源必須特別以同步化機制（如 critical sections）保護起來，以確保它們不會被不合時宜的執行緒切換動作糟蹋掉。Critical sections 和其他的執行緒同步化機制將在後面數節中討論。

雖然執行緒分享行程資源，每一個執行緒還是有一些它自己私有的資源。其中最重要的就屬堆疊（stack）了。不，並不是每一個執行緒有它自己的一個 SS 暫存器和堆疊節區，而是說，每一個執行緒在其行程位址空間中都有一塊專屬區域。預設情況下，每一個執行緒有 1MB 的位址空間做爲其堆疊。大小可以改變，要不是在程式的 .DEF 檔的 STACK 指令中指定，要不就是在 *CreateThread* 函式中指定一個非零的堆疊大小。Windows 95 並不真正使用整整 1MB 做爲執行緒堆疊，而是使用一種名爲 guard page 的機制，以獲知何時才需委派（commit）額外的記憶體到堆疊的位址範圍內。Guard pages 是結構化異常處理（structured exception handling）的一個例子，我將在稍後的「結構化異常處理」一節中討論之。

另一個生死攸關的執行緒資源是暫存器組。只要執行緒被切換出去，作業系統就必須把中斷時的暫存器內容儲存起來。十分有用（但是很少人知道）的 *GetThreadContext* 函式，允許你取得並修改執行緒的暫存器。正常程式不需要這麼做，但是讀取暫存器並修改其內容，是除錯器的生機命脈。

在作業系統之中，每一個執行緒有獨一無二的 thread ID。就像 process ID，Windows 95 的 thread ID 有相當高的值，但顯然不是一個線性位址。不過請注意，大部份的執行緒 API 函式並不用到 thread ID，它們要的是一個 HANDLE，通常被稱爲 hThread。只有在執行緒所屬的行程之中，hThread 才有意義。有可能多個 hThread 對映到同一個執行。某些 hThreads 在這個行程中有意義，另一些 hThreads 在另一個行程中有意義。

如果你開始注意到行程與執行緒之間的一種平行關係，那是好事。記住，行程和執行緒的 ID 在系統中是獨一無二的，handles 卻不如此。每一個行程和每一個執行緒都可能對映有許多個 hProcess 和 hThread。

## 行程以及執行緒的同步問題（Synchronization）

Win32 程式設計對 DOS 或 Win16 程式員有一個全新題目，那就是行程或執行緒的同步控制。同步控制的意思就是要避免程式因爲一個不適當的行程切換或執行緒切換而受到傷害。雖然 Win16 也是多工環境，但並不提供真正的同步控制元件，因爲它是一個合作型多工。一個 Win16 程式在它還沒有以自主意識釋放控制權之前，絕不可能被切換出去。釋放控制權的方法是呼叫 *GetMessage* 或 *PeekMessage*，那是很含蓄地表示說「我現在處於可被中斷的狀態」。

Win32 程式沒有這種奢侈的享受（也許奢不奢侈要看你怎麼看它）。它們必須有「在最壞的時機被切換出去」的心理準備。一個好的 Win32 程式不應該浪費時間在一個 polling loop 中打轉，等待某些事件發生。

Win32 API 有四個同步物件：

- Events
- Semaphores
- Mutexes
- Critical sections

除了 critical sections，其他同步控制物件都是全域變數，可以面對不同行程的執行緒工作。因此，這些同步控制物件不但能控制同一行程的執行緒，還可以控制不同的行程。

### Events

同步控制物件的第一種型態是 event。如其名稱所示，events 集中於某些發生於行程或執行緒的特定動作。當你希望你的執行緒停下來（blocked），直到某件事情發生爲止，那麼請使用 event。所謂 "block"，意思是虛懸執行緒，直到某些特定狀況吻合。即使被虛懸，仍具有效率意義，因爲排程器不會浪費任何 CPU 時間在被虛懸的執行緒身上。

呼叫 *CreateEvent* 或 *OpenEvent*，程式即可獲得一個 event handle。程式然後呼叫 *WaitForSingleObject*，交出一個 event handle 以及一個時間終了週期。此後執行緒即被鎖

住，直到其他執行緒（不管是在同一個行程或不同行程上）激發那個 event 爲止。激發 event 的動作是靠 *SetEvent* 或 *PulseEvent*。Events 被激發後，原先被鎖住的執行緒就會甦醒並繼續執行。

當一個執行緒想使用另一個執行緒的排序結果，你可能就會想要使用 event 來控制。讓排序執行緒在工作完成之後設立一個旗標（一個全域變數），這並不是好方法，因爲另一個執行緒必須在一個迴圈中打轉，不斷檢查旗標，看看它是否設立了起來。如果使用 event，事情就很簡單了。排序執行緒產生一個 event，用來表示排序完成。另一個執行緒則呼叫 *WaitForSingleObject*，交給系統先前產生出來的 event。這會造成此一執行緒鎖住並且不浪費任何 CPU 時間。當排序執行緒完成工作，它呼叫 *SetEvent*，把另一個執行緒喚醒，重新開始執行。不只是 CPU 時間獲得了有效的運用，我們也因此避免了平行處理所遭遇的問題。

這是同步控制之 API 的最簡單運用實例。除了 *WaitForSingleObject*，還有一個 *WaitForMultipleObjects*，讓執行緒鎖住直到一系列的 events 都被激發爲止。一個執行緒甚至還可以呼叫 *MsgWaitForMultipleObjects*，那將會把執行緒鎖住，直到 events 的條件被滿足了，或是直到產生了一個等待被執行的視窗訊息。其他函式還可以把執行緒鎖住直到 event 條件被滿足或是 I/O 動作完成。毫無疑問，這裡有很大的彈性。

## Semaphores

第二種型態就是 semaphores。當你想限制某一個資源被使用的次數，或是限制某一段碼被多少個執行緒呼叫，你可以使用 semaphores。學校宿舍的會客通行證是一個不錯的比喻。任何時間只能夠有一定量的學生在會客室裡頭（譯註：我從來不知道有這種情況），如果你想進去而所有的通行證都已發出，你就必須等待，直到有一張通行證回籠，然後你可以獲得你的通行證並且進去。在 Win32 程式設計中，要求一個 semaphore 就好像要求一張通行證一樣。

爲了使用 semaphore，一個執行緒必須呼叫 *CreateSemaphore* 以求獲得一個 semaphore

handle。呼叫這個函式時你必須指定多少執行緒可以同時使用這份資源或程式碼。如果這個 semaphore 只在一個行程中被使用，其他的執行緒可以經由一個全域變數取其 HANDLE。如果其他執行緒是屬於另一個行程，它們可以呼叫 *OpenSemaphore* 取得 HANDLE。當一個執行緒需要處理那塊被共享的資源時，它把資源交給 *WaitForSingleObject*（或其變種，如 *WaitForMultipleObjects*）。如果這個 semaphore 不曾宣告其可容納之執行緒的最大數量，那麼 *WaitForXXX* 函式就只是吐出 semaphore 的使用量，而執行緒也將不受影響地繼續執行。但如果這個 semaphore 已經滿溢了，呼叫 *WaitForXXX* 的執行緒將被迫停下來，直到其他執行緒釋放它們對 semaphore 的主權。釋放 semaphore 主權的方法是呼叫 *ReleaseSemaphore* 函式。

## Mutexes

同步控制的第三種型態是 mutex。這是一個縮寫字，是 mutual exclusion （「彼此互斥」）兩個字的組合。當我們希望一個程式或一組程式之中一次只能有一個執行緒存取某個資源或某段碼，我們就可以使用 mutex。如果某個執行緒正在使用一份資源，其他執行緒就會被排斥於外。你可以把 mutex 視爲「使用次數（usage count）爲 1」的 semaphore。Mutex 的運用和 semaphore 十分近似，它也一樣有產生、打開、釋放等函式。

## Critical sections

Win32 同步控制的第四種物件是所謂的 critical sections。與前三者不同的是，critical sections 只能用於同屬一個行程的各執行緒內。Critical sections 用來阻止多個執行緒在同一時間內執行同一段程式碼。與其他同步控制物件相比較， critical sections 代價低廉並且容易使用。你可以把 critical sections 想像是一個輕量級的 mutex，只適用於單一行程之中。要使用 critical section，程式必須配置一個型態爲 CRITICAL_SECTION 的全域變數。你必須在使用之前先呼叫 *InitialCriticalSection*，以便將欄位填入初值。然後，執行緒可以呼叫 *EnterCriticalSection* 進入這「關鍵地帶」。呼叫 *LeaveCriticalSection* 則是告訴作業系統：另一個執行緒可以進來了。

如我稍早所說，critical sections 十分容易使用。在 Windows 95 之中，如果一個執行緒呼叫 *EnterCriticalSection* 而彼時沒有其他執行緒進入，*EnterCriticalSection* 就只是調整並設定 CRITICAL_SECTION 的某些欄位。只有當另一個執行緒已經位於「關鍵地帶」之中，*EnterCriticalSection* 才會呼叫 VWIN32 VXD 並致使執行緒虛懸住。

### WaitForXXX 函式

到目前爲止我已經涵蓋了執行緒或行程同步化的四個主要方法，我還要再說一些其他方法。除了 event、semophore、mutext handles 之外，*WaitForXXX* 函式家族還可以接受其他的 handles，它們都是 Win32 KERNEL handles，曾經在前面的「KERNEL32 object handles」方塊中敘述過。把一個 hProcess 丟給 *WaitForXXX* 函式會使得執行緒停下來，直到 hProcess 所代表的那個行程結束才繼續執行。如果該行程早已結束，*WaitForXXX* 函式會立刻回返。同樣道理，把一個 hThread 丟給 *WaitForXXX* 函式會使得執行緒停下來，直到 hThread 所代表的那個執行緒結束才繼續執行。

另一個會引起 *WaitForXXX* 函式將執行緒停下來的 handle 是所謂的 "file change notification HANDLE"。這樣的一個 HANDLE 可以用來測知是否某個特定的改變已經在一個已知的磁碟目錄（及其子目錄）中發生。另一個 *WaitForXXX* 函式可以接受的 HANDLE 是 console input device 的 file HANDLE。一旦 console 的輸入緩衝區中有一個未讀取的輸入，*WaitForXXX* 函式就會回返，讓執行緒繼續執行下去。

## 模組管理（Module Management）

在行程和執行緒之後，剩下的關鍵性的 KERNEL 觀念就是模組（module）了。所謂模組，就是一個可執行檔（或 DLL）的程式碼、資料、資源在記憶體中的版本。每一個行程都有一個模組，被行程使用的每一個 DLL 也各有一個模組。如果兩個或多個行程使用同一個 DLL，它們就是共享相同的 DLL 模組。同樣道理，如果一個行程執行了兩份副本，這些副本就是共享了相同的 EXE 模組。

在 Win16 中，每一個 task 經由一個 NE 可執行檔的程式碼和資料而誕生。Win16 在一個節區中保持了一份可執行檔表頭，此節區名爲 module database，其 selector 稱爲 HMODULE。每一個 Win16 DLL 也有一個 module database，因爲 Win16 EXEs 和 DLLs 使用相同的檔案格式。Win16 程式把 HMODULE 交給那些「需要知道你所使用之 EXE 或 DLL」的 API 函式。

Windows 95 從 PE 檔中產生出一個 32 位元行程。PE 格式是 UNIX Common Object File Format（COFF）的更新版。第８章將非常詳細地探討 PE 格式，所以這裡我就跳過不說了。

在 Windows 95 之中，與 Win16 的 module database 最接近的東西當屬 EXE 或 DLL 的 PE 檔案表頭部份了。每一個 EXE 或 DLL 的表頭都會出現在記憶體中，因爲 Windows 95 使用記憶體映射檔（memory mapped files）來載入程式碼和資料。我將在稍後以一整節的空間討論所謂的記憶體映射檔。目前，請你把它想像爲記憶體中的一塊區域，作業系統從那裡讀取檔案的一部份（甚至全部）。

Windows 95 的 HMODULE 只不過是一個線性位址，指向記憶體映射檔的位置而已。當你獲得一個 HMODULE 並賦予小量計算，你就可以把 HMODULE 轉換爲指向 PE 表頭的一個指標。有了這個指標，程式就可以在記憶體中搜尋程式碼、資料、資源等東西。

Win16 對待 HMODULE 和 HINSTANCE 之間的差異有點草率。兩者其實並不相同，Win16 HMODULE 是一個 task（或 DLL DGROUP 節區）的 selector。然而 Win16 也使用 HINSTANCE 來區分兩個不同的 tasks。在 Windows 95 的 32 位元行程中，一個 HMODULE 和一個 HINSTANCE 則是完全相同的東西 -- 代表記憶體中的模組的基底位址。

就像對待 Win16 task 和 Win32 process 一樣，Windows 95 把模組相關資訊貯存在 16- 和 32- 位元籬笆的兩端。每一個 32 位元行程模組都有一個對應的 16 位元 NE module database。然而，這些 16 位元的表現已經縮減至最小，並非所有欄位都被填滿。我把這最小化的 HMODULEs 稱爲 "pseudo-HMODULEs"。pseudo-HMODULEs 並沒有出現在

正常的 16 位元模組串列之中。如果你以 TOOLHELP 走訪整個模組串列，看不到任何一個 pseudo-HMODULE。第 7 章的 SHOW16.EXE 告訴你如何找到 Win32 EXEs 和 DLLs 的 16 位元 module database。

## Windows 95 的位址空間

Windows 95 與 Windows NT 之間的一個主要差異就是，在 Windows 95 之中，16- 和 32- 位元程式在同一個虛擬機器和同一個位址空間中跑。爲了增加系統的強固性，NT 把 16 位元 Windows 程式放在另一個名爲 Windows on Windows（WOW）的虛擬機器中。NT 3.5 之後的版本甚至可以在個別的虛擬機器中執行個別的 16 位元程式。但它有一個缺點，那就是它把 32 位元行程和 16 位元行程分開，這使得 32 位元碼和 16 位元碼之間的移轉（thunking）比較困難。理想世界裡你其實不需要移轉動作，不幸的是許多 16 位元的 Windows DLLs 還沒有 32 位元版本。

從 16 位元程式的角度來看，位址空間和 Windows 3.1 那個時代並沒有什麼改變。所有 16 位元程式還是使用 ring3 的 16 位元 selector，全部來自一個共用的 LDT（local descriptor table）。這些程式可以使用（也可以分享）記憶體至其他的 16 位元程式。這是因爲所有 16 位元程式所使用的位址處於一個共用的位址空間中。一個 16 位元 task 總是能夠看到另一個 16 位元 task -- 只要它有一個合法的 selector，指向其他 task 的記憶體。虛擬記憶體管理器可以把一頁（page）記憶體標上記號，表示它不存在。但只要一觸及該記憶體，就會透明化地把那一塊記憶體帶回來。雖然微軟建議你在配置「可被 tasks 共用」之記憶體時，使用 GMEM_SHARE 屬性，但 Windows 3.x 程式常常會忽略這個忠告。是的，Windows 95 底下的 16 位元程式還是可以繼續忽略這個忠告。

存放 32 位元行程的位址空間可就大大不同了。就像 Windows NT，每一個 32 位元 Windows 95 行程的私有記憶體，只在它爲現行（current）行程時，才放在 CPU 的 page mapping tables 中。當排程器切換另一個 32 位元行程，前一個行程的私有記憶體就不再可以被任何其他行程存取。這麼做可以杜絕一個行程在另一個行程的記憶體中塗鴉 -- 不論是意外或故意。

由於 Win16 tasks 在共享區域中配置記憶體給其程式碼和資料使用，所以任何時候現行的 32 位元 Windows 95 行程必然可以看到所有 16 位元程式所使用的記憶體。然而，一個 32 位元行程沒有辦法看到另一個 32 位元行程的記憶體。同一時間之內只有一個行程的 memory context 會映射到系統中。但是把望遠鏡反轉過來看看，16 位元碼可以看到所有的共享區域，以及現行的 Win32 行程記憶體。如果不是這樣，要把 32 位元碼下移（thunk down）爲 16 位元碼可就困難重重囉！

將行程保護起來是一個很好的構想，但是有時候你又真的需要共享記憶體。在行程之間共享記憶體的主要方法是所謂的記憶體映射檔（memory mapped files）。這個名稱有時會帶來一些誤解，因爲它其實可以完全和磁碟檔案無關。存在於 NT 和 Windows 95 之間一個有趣的架構差異就是對於檔案映射的「能見度」。在 NT 之中，記憶體映射檔只能夠被那些針對檔案呼叫了 *CreateFileMapping* 和 *MapViewOfFile* 的行程使用。此外，檔案記憶體範圍可以根據不同行程的不同虛擬位址而改變。而在 Windows 95 之中，一旦某個程式產生出一個記憶體映射檔，其他所有程式都可以共用之。因此，Windows 95 的記憶體映射檔總是使用存在於所有行程之中的相同虛擬位址。毫無疑問這簡化了 Windows 95 的虛擬記憶體管理。我將在稍後以一整節介紹「記憶體映射檔」。

Windows 95 程式不再需要用到 selectors。所有 32 位元程式都在啓動的時候被初始化，使用相同的 code selector 和 data selector。應用程式本身從不需要改變 segment selectors。Windows 95 的 system DLLs 會在下移（thunk down）至 16 位元碼時暫時性地改變 segment registers。當我執行 Win32 程式，每一個程式使用一個 ring3 LDT code selector，其值總是 0x013F。這個 selector 的 "base" 欄位爲 0，"limit" 欄位爲 0xFFFFFFFF（4GB）。Windows 95 程式使用於 DS、SS、ES 暫存器的 data selector 有一點不尋常，它是一個 expanded-down selector，其"limit" 值低於 1MB。

在 Windows 95 之前，你很少能夠遇到 expand down selectors ，所以對它做點解釋是必要的。一個 expand down selector（或說 descriptor）的 "limit" 值其實就是「可以使用此一 selector 之程式」的最低 offset 值。最高的可用 offset 值則是此一 selector 的可定址記憶體的尾端。在 Windows 95 之中，data segment selector 是一個 32 位元的 LDT

selector，"base" 欄位爲 0。這意味著此一 selector 的合法定址範圍是在「一個小於 1MB 的值」和 4GB 之間。Windows 95 將虛擬機器的最低 4KB 位址設爲不可存取，如此一來它就可以讓那些「帶有 NULL 指標錯誤」的程式引發 GP fault，而不是默默地覆寫到記憶體上頭去。

所有 32 位元行程都使用相同的 selectors，這對擁有 16 位元基礎背景的人而言，常常帶來困擾。你如何能夠在不同的程式中使用相同的 code selector 呢？稍早我曾經提過，Windows 使用 CPU 的分頁映射（page mapping）特質，把實際的 RAM 映射到線性位址中。每一個行程有它自己一組分頁映射表。一旦 Windows 95 要切換行程，它就改變 CPU 的分頁映射表，以反映出新行程的記憶體佈局。因此，即使兩個程式有著相同的 selector，它們在相同的線性位址中還是有著完全不同的碼。這也就是爲什麼我們說「如果只知道位址卻不知道其行程，此位址一點用也沒有」的原因。

## Windows 95 的記憶體管理

表面上看，Windows 95 的 32 位元記憶體管理架構類似 NT。而在底層，KERNEL32 十分依賴由 VMM32.VXD 提供的服務，實作出 Win32 記憶體管理函式。在 16 位元這一邊，KRNL386 現在也直接呼叫 VMM32 中的 VWIN32 VxD 了，用以提供低階服務如配置大塊記憶體並鎖住（pagelocking）等等。在 Windows 3.1 中，KRNL386 使用的是由 WIN386 提供的 DPMI 函式，供應許多（但非全部）相同的服務。

在大部份程式員每天接觸的那一層次，最大的消息就是 Win32 以及 Windows 95 不再有節區（segment）。移轉到 32 位元程式設計後，你可以把近程和遠程指標統統忘掉，你也可以忘掉 *GlobalLock*、*LocalLock*、以及任何和記憶體模式有關的東西。Windows 95 程式中的每一樣東西都是小型模式（small model）-- 32 位元的小型模式。當然，如果你要對記憶體管理器做一些惡作劇，Win32 API 和 Windows 95 有一組新的函式，一定能夠讓低階駭客很高興。

在 Windows 95 中，低階的記憶體操作是由 *VirtualXXX* 函式提供，第 5 章對此有詳細的敘述。*VirtualAlloc* 允許你以 4K 爲單位配置大量記憶體。4K 是 80386 CPU 的分頁

單位。雖然其間有些重大的差異，不過 Windows 3.1 之中和 *VirtualAlloc* 最接近的就數 *GlobalAlloc* 了。兩個函式都企圖配置大塊記憶體，但同時，兩個函式的配置單位也都使得它們的經常性開消（overhead）不小。或許你並不想使用任何一個來取代 *malloc* 或 *new*。

當你以 *VirtualAlloc* 配置記憶體，你可以（也可以不）把指定的一段位址空間和實際的 RAM 繫結在一起，用的是 MEM_COMMIT 狀態旗標。什麼原因使你不想讓配置來的位址空間立刻和實際的記憶體發生關係呢？稀疏記憶體（sparse memory）是主要原因。例如，你的程式可能需要大量記憶體（以 MB 爲單位），但一開始你並不知道真正需要多少。這種情況下你可以呼叫 *VirtualAlloc* 配置一大塊位址空間，大到絕對足以應付你的需求。一旦你的程式需要更多的位址範圍，你可以再呼叫 *VirtualAlloc* 以委派（commit）記憶體。本章稍後的「結構化異常處理（Structured exception handling）」一節對此自動程序有更詳細的敘述。這種「要吃才拿」的哲學正是 Windows 95 如何能夠實作出大塊程式堆疊而又不浪費記憶體的原因。

高階的 Windows 95 記憶體管理是以 heap 函式的形式出現（第５章細述）。當 Windows 95 產生一個新的 32 位元行程，會在其位址空間中爲它產生一個 heap。這個 32-bit heap 大致相當於 16 位元的 Windows local heap，因爲每一個行程都有一個。然而，32-bit heap 當然不受限於 64KB。Windows 95 可同時支援許多個 heaps，所以當你要操作 heap 時，必須將一個 heap handle 交給函式。你可以使用 *GetProcessHeap* 取得這預設的 heap handle，它代表的正是這個 heap 的起始位址。

和 *VirtualXXX* 函式不同的是，以 Win32 heap 函式配置而得的記憶體，其基本單位極小（4 個位元組，而不是 4KB 位元組）。每一次配置的最大額外負擔也不過就是 4 個位元組。這使得 *HeapAlloc* 成爲比較好的 *malloc* 替代品。這 4 個位元組以 DWORD 的型式緊鄰出現在 *HeapAlloc* 所傳回的位址之前。此一 DWORD（不計最底部的兩個位元）內含的是後面緊鄰所接的記憶體區塊的大小。

永遠不能忘記回溯相容！數以千計的 16 位元程式使用 *GlobalAlloc* 和 *LocalAlloc*，如何移植？是的，微軟繼續保留這些重要的記憶體配置函式，然而，這些函式的意義及其底層動作已經有所改變。首先，並且也十分重要的一點是，global heap 和 local heap 如今已不分家，你可以呼叫 *GlobalAlloc* 配置一塊記憶體再以 *LocalHeap* 釋放它。第二，不論 global heap 或 local heap 都使用前面所說的 32-bit heap。因此，下面的呼叫所傳回的指標：

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

應該和下面這個呼叫所傳回的指標是一樣的：

```
LocalAlloc (LMEM_FIXED, 0x100);
```

一旦成功，*HeapAlloc* 總是傳回一個可用指標。所以，以 *HeapAlloc* 配置的每一塊記憶體都相當於 LMEM_FIXED。固定而不能移動的區塊可能會導至 heap 空間碎裂。在 Win16 中，你必須呼叫 *LocalLock* 才能獲得一塊可搬移區塊的指標。Win16 有一項鮮爲人知的事實，那就是可搬移區塊的 handle 的 bit1 總是設立，所以其 handle 值總是以 2 或 6 或 0xA 或 0xE 收尾（譯註）。如果你把一個可搬移之區塊的 handle 當成指標的指標來看，你可以經由它指出它所指示之區塊位址。微軟「回溯相容」的承諾在這裡可以看出，因爲這些規則也都適用於 32-bit heaps。

> 譯註：這裡的解釋不夠清楚。Selector 的 16 個位元中，bit0 和 bit1 用做 RPL（Requestor Privilege Level），此值對於 Windows 應用程式必爲 3；而 Windows 3.1 的 selector 值比對應的 handle 值多 1，所以 handle 的最低兩個位元必爲 10，所以才得到文中的結論：handle 值總是以 2 或 6 或 0xA 或 0xE 收尾。

Local heap 函式和新的 Win32 *HeapXXX* 函式之間的對應關係十分簡單。*HeapAlloc* 和 *HeapFree* 取代 *LocalAlloc* 和 *LocalFree*，*HeapReAlloc* 和 *HeapSize* 取代 *LocalReAlloc* 和 *LocalSize*。*HeapCreate* 則差不多相當於先以一個 *GlobalAlloc* 攫取一塊 global heap 記憶體，然後再呼叫 *LocalInit* 把它設定成爲一個 heap。至於 *HeapDestroy*，沒有對應的 Win16 函式。第５章將詳細地描述記憶體管理。

## 記憶體映射檔（Memory mapped files）

Win32 和 Windows 95 中最酷的一項性質當屬記憶體映射檔了。Win16 之中並沒有對等的東西，Windows 95 的 16 位元程式也不能夠使用它。記憶體映射檔有三個主要用途。第一同時也是最明顯的用途就是讓你很方便地以指標讀寫檔案。檔案映射動作將指定磁碟檔案的一部份（或全部）對映於虛擬位址空間中某一範圍的記憶體。當你對著這段記憶體讀寫資料，作業系統會把它轉到磁碟檔案去。

第二個用途就是在不同的 Win32 行程之間共享記憶體。一個 Win32 行程可以爲一個 NULL 檔案設定一個 "file mapping"，以保留一塊位址空間，卻不指定給它一塊磁碟檔案。其他行程可以打開對此 "file mapping" 的視野（view）。與此一 "file mapping" 之位址範圍產生連接的那些實際記憶體於是就可以被其他行程看見。因此，一個行程若要分享另一個行程的記憶體，只要對相同的 "file mapping" 取得視野（view）即可。這個過程中不需要磁碟檔案的參與。

記憶體映射檔的第三個用途是用於模組載入。當 Windows 95 的 32 位元載入器需要載入一個 EXE 或 DLL，它使用記憶體映射檔，把檔案內容映射到行程的位址空間中。由於記憶體映射檔可以被其他行程看見，所以 Windows 95 十分輕鬆並有效地可以在不同行程之間分享 EXE 或 DLL 的程式碼、資料、資源。經由儲存在 PE 檔的數值，Windows 載入器把可執行檔中各式各樣的段落（sections）映射到特定的位址上。第８章詳細地描述了這一部份。

## 結構化異常處理（Structured exception handling）

Win32 和 Windows 95 之中最有用但是最不容易了解的元件就是「結構化異常處理」。在 Windows 3.1 之前，API 中並沒有正式的機制讓程式得以處理中斷。Windows 3.1 引入 TOOLHELP.DLL，那是偉大的一步，但要稱其爲「結構化」，還是太誇張了些。TOOLHELP 攔截小量但是有用的中斷，像是中斷點中斷（breakpoint interrupt，INT 3）、GP fault（exception 13h）。當一個異常情況（exception）發生，TOOLHELP 的內部處理常式就取得控制權。處理常式會設定一個相同的堆疊框架（stack frame），然後呼叫由應

用程式設立的處理常式（我們稱之爲 interrupt callback）。

雖然 TOOLHELP 允許許多彈性，它也留下許多問題。每一個 task，只要安裝一個 interrupt callback，就可以看到所有來自 TOOLHELP 的中斷和異常情況。而且，callback 函式可以告訴 TOOLHELP 是否要呼叫下一個 callback 函式。因此，一個 task 有能力阻止另一個 task 看到某個中斷，而那可能是後者非常依賴的一個中斷。此外，萬一 callback 函式有臭蟲，可能會引發巢狀的 GP fault 以及其他造成系統當機的行爲。對於 32 位元行程，Windows 95 取消這種「每一個 task 只爲自己」的方式，以一種更良好的行爲規範來處理異常情況。

除非你要設計除錯器，否則爲什麼一個行程需要處理異常情況（exception）呢？原因之一是，行程需要做一個可能會引起 GP fault 或「除以零」的動作。如果行程知道如何彌補這種事態，它就不會被作業系統強制結束掉。情況之二是因爲稀疏記憶體。一個程式可能需要使用大量記憶體，但是卻無法預先得知到底需要多少。利用 *VirtualAlloc*，程式可以保留一大塊足夠的虛擬位址空間，當行程存取了位址範圍中的一頁記憶體，而它沒有映射到實際的 RAM 時，CPU 會產生一個 page fault。有了結構化異常處理，一個 Win32 行程就可以處理 page fault，將 RAM 指定出去，然後告訴作業系統重新將失敗的指令做一次。

技術上來說，結構化異常處理是建立在作業系統之上，和程式語言沒有關係。然而，結構化異常情況在作業系統層面十分地麻煩並且複雜。事實上，就在我下筆的時刻，我並不知道這個主題有任何正式文件公佈於世。爲了這些理由，大部份程式員讓他們的編譯器以及執行時期函式庫放一個可愛而有趣的介面在程式裡頭，用來處理結構化異常。第３章有更進一步的說明。

如果某一個異常情況沒有被行程中的任何處理常式處理，就會被交給作業系統的一個預設處理常式。這個處理常式的工作就是把行程結束掉，並釋放未被釋放的資源。爲了增加強固性，Windows 95 將這一系列動作放在另一個執行緒中。這是因爲當執行緒面對一個非預期的毀滅（例如由於違反存取規定，access violation），thread context 可能處於一

種不穩定狀態。以另一個執行緒（良好的 thread context）進行清除工作，顯然能夠減少麻煩。

## Registry（登錄資料庫）

在 Windows NT 之前，不論系統或應用程式，都把它們的永久性資料儲存在一個雜亂的 .INI 檔中。還記得那巨大的 WIN.INI 嗎？還記得裡頭令人目眩神迷的一大堆項目嗎？Windows 95 使用一個所謂的 registry，使混亂的資訊大步邁向中央集權管理。

過去你放在 .INI 中的資訊，現在應該放在 Windows 95 的 registry 中。Registry 是一個有階層架構的資料庫，圖 **2-2** 的 REGEDIT 程式可以顯示其階層架構。最頂層有數個預先定義的 key（鍵值名稱）。每一個 key 有一些 subkey。在 registry 的任何地方，subkey 可以有一筆或多筆內容（文字或二進位值），也可以有一個或多個 subkeys。有一組 APIs（像是 *RegCreateKeyEx*、*RegQueryValue...*）用來加入、刪除、修改、搜尋 registry。

六個預設的、最高層次的 keys 是：

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

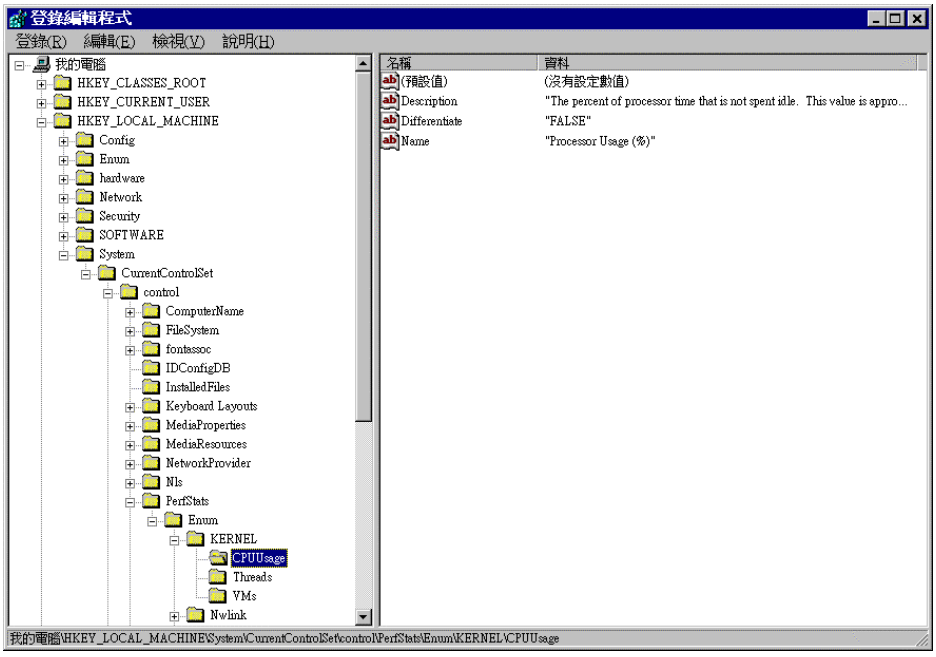



**圖 2-2 Windows 95 REGEDIT 程式可以顯示 registry 的階層架構。**

我們特別對 HKEY_DYN_DATA 感興趣（圖 **2-2**）。不斷往下追尋其節點，可以獲得不少有用的資訊。例如，HKEY_DYN_DATA\PerfStats\StartStat\ 將你引導到一個名爲 KERNEL\CPUUsage 的數值，以及一個名爲 VFAT\ReadsSec 的數值。

Registry 事實上是 VMM VXD 實作出來的。把其中的內容放到第一個被載入的 VxD（VMM VxD）中，registry 的資訊就可以被 VxDs 使用。你不妨觀察 Windows 95 DDK 的 VMM.H 檔，在其中，你會發現下面的 VxD services 可被其他 VxDs 使用：

```
// 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 層面，registry API 實作於 ADVAPI32.DLL 之中。Windows 95 的這個檔案十分短小。挖進去看看，就會恍然大悟：原來所有的 registry API 函式只不過是一層包裝，底部呼叫的是 VMM 的 registry 函式。當然，由於 ADVAPI32.DLL 是 ring3 碼，所以不能夠直接呼叫 VMM 函式；它還是使用 KERNEL32 所使用的相同的 Win32 VxD services（稍早描述過）。關於 VxD services，第６章將有敘述。

## USER 的新增性質

Windows 95 中的視窗系統有些什麼新東西？第一道菜是，有爲數眾多的新的視窗風格，讓 Windows 95 應用程式能夠雕塑三維空間的外觀。新的視窗風格包括：

| 風格 | 目的 |
|---|---|
| WS_EX_MDICHILD | 產生一個 MDI 子視窗 |
| WS_EX_TOOLWINDOW | 給工具列視窗使用 |
| WS_EX_CLIENTEDGE | 視窗有一個下沉的邊緣 |
| WS_EX_RIGHT | 視窗文字一律靠右排列 |
| WS_EX_LEFTSCROLLBAR | 捲軸在左邊 |

其他令人興奮的新東西還包括一組新的控制元件，它們是：

| 控制元件 | 目的 |
|---|---|
| Animate | 顯示一個 .AVI 檔案 |
| DragListBoxes | Drags listbox items between lists |
| Header | Header bar |
| HotKey | 熱鍵 |

| | |
|---|---|
| ImageList | 影像串列 |
| ListView | List view |
| Progress | 進度計量器 |
| Property Sheets | Edit item Properties |
| RichEdit | Rich Format Text |
| StatusWindow | 狀態視窗 |
| TabControl | 有附頁的對話盒 |
| ToolBar | 自定的附含圖形按鈕的工具列 |
| Tooltips | 突冒式輔助系統 |
| TrackBar | Customizable column-width tracking |
| TreeView | 樹狀圖形 |
| UpDown | 上下箭頭（遞增/遞減） |

和標準控制元件（出現於 Windows 3.1 中者）不同，這些新的控制元件並不由 USER.EXE 提供，而是實作於 COMCTL32.DLL 和 COMMCTRL.DLL 之中。所以，這些新的控制元件只適用於 32 位元行程。16 位元程式沒辦法參加這場嘉年華會。

## 系統除錯和除錯

Windows 95 實作的 Win32 debugging API 遠比出現在 Win16 中的正式得多。Windows 3.1 或 Windows 95 中的 16 位元除錯器一般而言是使用 TOOLHELP 來安裝一個 interrupt callback 或 notification callback。經由觀察 interrupt 或 notification 串列，除錯器就可以知道除錯對象的所作所爲。然而，除錯器的 callback 函式必須將一些不感興趣的 events 或是屬於其他行程的 events 過濾掉。此外，當除錯對象執行至中斷點（break point）或是引發一個異常情況（exception），除錯器處理常式需要在某種迴圈中打轉，直到除錯器願意讓除錯對象繼續執行下去。一句話啦，Win16 除錯器十分麻煩。

Windows 95 的 debug API 以 *WaitForDebugEvent* 爲中心。在產生或搭上一個行程之後，除錯器呼叫 *WaitForDebugEvent*，把指向 DEBUG_EVENT 結構的指標交過去。這個函

式會阻止行程的進行，直到除錯對象發生了某些除錯器所在意的事情。能夠促使 *WaitForDebugEvent* 回返的 debug event 列於表 **2-4** 中。

**表 2-4　能夠促使 WaitForDebugEvent 回返的 debug event。**

| 除錯事件 | 說明 |
|---|---|
| EXCEPTION_DEBUG_EVENT | 告訴除錯器說遇到了中斷點、access violations、以及其他的異常情況。 |
| CREATE_THREAD_DEBUG_EVENT 和 EXIT_THREAD_DEBUG_EVENT | 將除錯器致能（enable），使它有能力追蹤除錯對象所屬的執行緒。 |
| LOAD_DLL_DEBUG_EVENT 和 UNLOAD_DLL_DEBUG_EVENT | 讓除錯器知道除錯對象使用了哪些 DLLs。除錯器可以利用這些通知來動態載入或剔除 DLLs 的符號表格。 |
| OUTPUT_DEBUG_STRING_EVENT | 讓你能夠看到你的 OutputDebugString 訊息。更細節的部份，請參考本節對於 OUTPUT_DEBUG_STRING_EVENT 和 WaitForDebugEvent 的討論。 |
| CREATE_PROCESS_DEBUG_EVENT 和 EXIT_PROCESS_DEBUG_EVENT | 告訴除錯器說除錯對象產生了一個新的行程，或是結束了一個行程。 |
| RIP_EVENT | 此訊息似乎不曾被產生。 |

和每一個 debug event 息息相關的，是一個內含該 event 詳細資訊的結構。除錯器可以利用這些 notifications 做某些事情，像是動態地載入或剔除 DLL 的符號表格。OUTPUT_DEBUG_STRING_EVENT 應該是大家比較有興趣的一個 event。在 Win32 中，這是能夠看到你的 *OutputDebugString* 訊息的唯一方法。你必須在一個除錯環境中執行你那個擁有 *WaitForDebugEvent* 的程式。而在 Win16，任何程式都可以看到系統中所有的 *OutputDebugString* 訊息 -- 只要程式曾經輕扣 ToolHelp *NotifyRegister* 的大門。這正是 Win16 DBWIN 程式之所爲。

不管 *WaitForDebugEvent* 何時帶著一個 event 回返除錯器，子行程（除錯對象）中的所有行動都將完全凍結。除錯器不需擔心「虛懸所有子執行緒」這件事情。它只要負責處理這個 event，並於最後呼叫 *ContinueDebugEvent*，讓除錯對象繼續執行。Win32 除錯

器的心臟是一個迴圈，呼叫 *WaitForDebugEvent* 和 *ContinueDebugEvent*，直到除錯器收到 EXIT_PROCESS_DEBUG_EVENT 方休。

除了知道除錯對象發生的 events，除錯器還需要能夠改寫除錯對象的暫存器。*WriteProcessMemory* 和 *ReadProcessMemory*（第 5 章）解決了這個問題。同樣地，*GetThreadContext* 和 *SetThreadContext*（第 3 章）也讓除錯器得以讀寫除錯對象的執行緒中的暫存器內容。

除了提供 interrupts 以及 system events 等資訊，Windows 3.1 的 TOOLHELP.DLL 也提供一條方便的路，讓程式很容易獲得各式各樣的系統資料結構，像是 modules、tasks、heap 等等。在 Windows 95 中，這些資料結構之於 32 位元程式已經有了明顯的改變。微軟又提供了一套 32 位元 TOOLTHELP，名爲 TOOLHELP32。以下這些函式定義在 TLHELP32.H 之中：

```
CreateToolhelp32Snapshot
Heap32ListFirst
Heap32ListNext
Heap32First
Heap32Next
Toolhelp32ReadProcessMemory
Process32First
Process32Next
Thread32First
Thread323Next
Module32First
Module32Next
```

這些 API 函式類似(但當然不等同於)Win16 的 TOOLHELP.DLL 函式。如果你的 16 位元碼使用了這些 TOOLHELP 函式，你需要做點移植工作。此外，Win16 的 TOOLHELP.DLL 獨立於 KRNL386 之外，而 TOOLHELP32 函式卻是實作於 KERNEL32.DLL 之中。

在一個像 Windows 95 這樣的強制性多工環境中完成這種曝露系統資訊的函式，必須特別小心，否則會遭遇一些問題。例如，在走訪執行緒串列的途中，執行走訪動作的執行緒可能會被切換出去，而在它再次回來執行之前，執行緒串列可能有所改變。爲了避免

這樣的（以及類似的）問題，TOOLHELP32 函式有一種所謂「快照（snapshot）」的觀念。當你要走訪一個串列，首先得呼叫 *CreateToolhelp32Snapshot*，產生一張快照，它會在一塊緩衝區中填入一份與系統狀態完全一致的資訊；然後你可以把這張「快照」的 handle 傳給 TOOLHELP32 的列舉函式中，而那些列舉函式就會從「快照」之中取出它們所要的資訊。

與 Win16 的 TOOLHELP.DLL 比較，TOOLHELP32 少了一些函式，像是走訪「視窗類別」串列、獲得 system heap 的使用情況、對另一個行程執行堆疊追蹤（stack trace）的動作等等。Windows 95 有其他方法可以取代這些消失的函式。我發表於 ***Microsoft Systems Journal*** 1995 年九月的一篇文章（譯註："*An Exclusive Tour of the New TOOLHELP32 Functions for Windows 95*"）詳細地描述了 TOOLHELP32 函式，並建議其他一些方法，用以完成 TOOLHELP32 沒有提供的功能。

## Windows 95 的一些骯髒秘聞

在結束這一章之前，我想我已經丟給大眾一大串糟糕的設計決策，以及一些令人困窘的資訊，那是微軟不希望在短時間之內公開出來的。

這一節中我可以談的許多主題已經在本章其他地方或其他書籍文章中有過討論，包括：

- 一些零零碎碎的真實模式的 DOS 碼還在使用。
- 可共享的位址空間（4MB 以下，2GB 以上）幾乎完全沒有保護。不論 Win16 或 Win32 程式都可以在敏感的系統資料區中胡亂塗鴉。
- Win16Mutext 再加上素行不良的 16-bit tasks，可以顛覆整個系統的多工體制。
- 不管矛盾與否，KERNEL32 還是呼叫了 KRNL386。其規模值得注意，將在第６章討論。

不要再提上面那些話題了，現在我想說一些其他有趣的題目 -- 一些直到目前還沒有被大量注意的題目。下面先給你一些重點提示：

- 新的「反挖掘程式碼（anti-hacking code）」，企圖阻止你取用未公開的 KERNEL32 函式。
- 由於 Windows 95 和 Windows NT 開發小組之間缺乏溝通與合作，導至 Win32 API 並非唯一。
- 自由系統資源（Free System Resource，FSR）計算方式的改變，使 Windows 95 看起來戲劇性地有了更多的 USER heap 和 GDI heap 空間。但事實並非如此。
- 新的 16 位元碼靜悄悄地加入了系統 -- 雖然微軟說 32 位元碼才是正確道路。

## 反挖掘程式碼（anti-hacking code）

***Unauthorized Windows 95*** 一書對於 KERNEL32.DLL 的未公開函式做了多方面的使用。雖然這些函式並沒有 .H 檔提供其原型，但它們出現於 KERNEL32.DLL 的 import library 中。所以呼叫這些函式極容易：提供一個函式原型，並與 KERNEL32.LIB 聯結。

在 ***Unauthorized Windows 95*** 一書問世之後推出的 Windows 95 版本中，這些函式從 KERNEL32.LIB 中消失了。真令人驚訝！真令人驚訝！這些未公開函式其實還是開放的，但它們是以序號（而非函式名稱）開放。

這原本只會帶來一點小小漣漪。你還是能夠呼叫 *GetProcAddress*，並把函式序號當作函式名稱來傳遞（HIWORD 放 0，LOWORD 放序號），並獲得函式位址。這應該是可行的，然而，微軟卻修改了 *GetProcAddress* 的碼，如果它發現函式是以序號的形式被指定，並且 HMODULE 又是代表 KERNEL32.DLL，那麼 *GetProcAddress* 就不會成功。在 KERNEL32.DLL 的除錯版中，這段碼會吐出一個訊息：*GetProcAddress*: kernel32 by id not supported"。

現在，讓我們想想這款情事。由於未公開函式並未以函式名稱開放出來，你不能夠把一個 KERNEL32 函式交給 *GetProcAddress* 以取其位址（函式進入點）。而如今 *GetProcAddress* 又拒絕你使用函式序號。顯然微軟不希望人們呼叫那些未被公開的

KERNEL32 函式。那麼，很明顯地，除非你有「神奇的」KERNEL32 import library（微軟的 Win32 SDK 並不提供這玩意兒，它提供的是一個剝過皮的版本），你才能夠呼叫它們。

不要擔憂，你將在這本書中看到，我對 KERNEL32 未公開函式做了多方面的使用（有益的，不是有害的）。加上一點點努力，我就能夠強迫 Visual C++ 產生一個內含「向公開文件挑戰」的 KERNEL32 import library。附錄Ａ含有這些函式的資訊，以及它們的 import library。

「反挖掘」的另一個例子是所謂的 Obsfucator 旗標。在 Windows 95 的早期版本中，*GetCurrentProcessId* 和 *GetCurrentThreadId* 傳回指標，指向適當行程和執行緒的資料結構（第３章將討論之）。然而就在 ***Unauthorized Windows 95*** 一書問世之後，這些函式開始傳回一些明顯不是指標的東西。經過一些研究，我發現其實傳回來的還是原來的指標，只不過又 XOR 了一個似乎是隨機的數值。這隨機值從何而來？每次系統啓動，就利用系統時鐘計算一個隨機值。有趣的是，在 KERNEL32.DLL 除錯版中，此一隨機值名爲 Obsfucator。看來 KERNEL32 開發人員把 "obfuscator" 誤拼爲 "obsfucator" 了。依我看呢，KERNEL32 原始碼可能需要來一次拼字檢查。

微軟使用 XOR 來修飾 *GetCurrentThreadId* 和 *GetCurrentProcessId* 的回返值，其實沒有理由，唯一的理由就是要阻止人們取得系統資料結構。微軟當然可以設法隱藏那些東西，但當有人真正需要這個資訊並且進去挖掘，微軟不應該抱怨。第３章描述了一個技術，可以在程式執行時期計算 Obsfucator 的值，這麼一來你就能夠取得執行緒和行程的資料結構了。

## Win32 API 的鬧劇

雖然微軟要你相信，只有單一一套 Win32 API，但事實上 NT 和 Windows 95 開發小組之間的整合並不是很好。這種缺乏協調的結果就是，NT 和 Windows 95 的 Win32 API 個數不一致。下面三個證據可以證明我的觀點。

**證據之一** ToolHelp32 函式。我從許多管道聽來，說 NT 小組發誓絕不會把這些 APIs 實作出來。如果你仔細看看 ToolHelp32 函式，你會發現它們只不過是少數幾個函式。主要的趣味集中在行程和執行緒的列舉函式上頭。這些資訊也可以從 Windows NT 的 registry 中獲得，Win32 SDK 所提供的 PVIEW 程式就明白地做了示範。我心中的疑問是：爲什麼 Windows 95 小組不也設計像 NT 一樣的 registry key，使得 PVIEW 可以直接發揮效用？爲什麼 NT 小組不能夠在 registry 函式之上再加一層，把 ToolHelp32 函式實作於 Windows NT 之中？如果有任何一邊行動了，系統資訊的列舉函式方面就有了一組可移植的 Win32 API。當我快要完成這本書的時候，我聽到 NT 小組的一員咕噥說 ToolHelp32 函式可能有一天會出現在未來版本的 Windows NT 上。

**證據之二** heap 函式。有數個 Win32 heap 函式是 Windows 95 沒有提供的 -- 雖然把它們設計出來也許要不了一個小時。最主要的例子就是 Windows NT 的 *HeapWalk* 函式。這個函式並沒有出現於 Windows 95，但如果你看看 TLHELP32.H，你會發現有兩個函式事實上做相同的事情：*Heap32First* 和 *Heap32Next*。不把既有的一個 Win32 API 實作出來，卻去開發兩個全新的函式，NT 小組一定會反擊說他們也不打算支援其他一些函式。哎，神經病！

**證據之三** *HeapLock* 函式。Windows NT 的此一函式獲得一個特定的 Win32 heap 的 mutex。你將在第５章看到，Windows 95 也有一個對等函式。然而，KERNEL32 開發小組並沒有開放（export）這個函式，因此 Windows 95 沒有實作出 *HeapLock* 的最可能理由就是，某些人不喜歡把既有的函式改名爲 *HeapLock* 並把它從 KERNEL32.DLL 中開放出來。

重點在於，雖然微軟嘗試讓每一個人覺悟，寫出標準的 Win32 API，微軟自己的兩個小組卻只做他們喜歡做的事。長久下來微軟一定會大受其害。我已經提出我這一部份的 WINBUG 報告，並送出非常多的 e-mails。現在要看市場的了，看看什麼回應會發生在這「想像中合爲一體」的 Win32 API 身上。

## 目前系統資源（FSR）的謊言

如果，在啓動 Windows 95 之後，你立刻執行 Explorer 並去看看它的【Help|About】對話盒，你會看到一個相當高的 FSR 值，大約是 95%。這遠高於你在 Windows 3.1 中所看到的。難道 Windows 95 突然自 16 位元 USER 和 GDI 的 heaps 中獲得大量自由記憶體嗎？不，事實上，有更多的東西被加到 USER 的 DGROUP 節區中。再怎麼說，FSR 似乎應該變小或持平才是！

那麼到底是怎麼回事？一如我將在第４章說明的，Windows 95 啓動時，Explorer 令桌面視窗計算正確的、類似 Windows 3.1 的 FSR 值。後來所有對 *GetFreeSystemResources* 的呼叫都被這些最初值給偏斜了。因此，當 Explorer 宣稱 FSR 有 95% 時，它的意思是在 Explorer 以及其他程式啓動之「後」（譯註）有 95% 的資源。FSR 計算的改變主要企圖是「炫耀」，告訴大家 Windows 95 優於 Windows 3.1。

> 譯註：原文是 When the Explorer says that there's 95 percent of the system resources available, it means 95 percent of the resources **after** the Explorer and other programs have started. 作者特別強調 **after**，但我卻感覺似乎應該是 **before**，不然前後意思就有點矛盾了。

## Win16 沒有死

雖然微軟強烈地要把每一個人推往 Win32，這麼多的 Win32 底層卻依賴 16 位元碼。這不是秘密，也不值得再提一次。奇怪的是，微軟在 16 位元 DLLs 中增加新函式，並沒有引起太多的雜音。這些函式有許多其實是正式的 Win32 API 的 16 位元對等品，例如 *CreateDirectory* 和 *GetPrivateProfileSection*（譯註：這兩個都是 Win16 不曾有的函式）。某些情況下，這些新函式被無聲無息地加到 16 位元 Windows.h 中。某些情況下，這些新函式被 16 位元 DLL 開放出來，但對應的 .H 檔中卻沒有其函式原型。這種情況下，一份 Win32 文件再加上一些常識，通常可以讓你過關。

如果微軟沒有公佈這些 16 位元附加物，誰會去用它？如果每一個人都應該去寫 Win32 碼，微軟何必加上這些新的 Win16 APIs？很明顯，微軟知道 Win16 會繼續存在一段相

當長的時間 -- 甚至在 Windows 95 推出之後。但是現在所有的程式員都議論紛紛地說 Win16 是一條死胡同，Win32 才是活路。我個人同意，程式員將把焦點放在 Win32 上頭 -- 如果可能的話。但是強迫把他們統統拉往 Win32，卻似乎不是一步好棋。

## 摘要

Windows 95 是一個明確的作業系統。雖然大部份碼衍生自 Windows 3.1，Windows 95 的 16 位元部份已經重新修改過，以去除許多 16 位元的限制，以及處理 Win32 多緒執行的需要。Windows 95 並不是 Win32s，它的執行緒以及多個位址空間的設計遠比 Win32s 好。Windows 95 也不是一個輕量級的 Windows NT。Windows 95 的碼經過效率以及空間（記憶體消耗量）的最佳化處理 -- 針對 Intel X86 CPUs 而言。NT 的焦點則是放在移植性與強固性。雖然 Windows 95 和 Windows NT 的架構在某些關鍵部份有相當程度的不同，它們在微軟的作業系統策略上的重要性卻是等量齊觀。未來數年它們的重要性將不至於衰褪。

# 模組、行程、執行緒

## （Modules, Processes, Threads）

大部份人都有偏好的顏色。你可以說我有病，但我的確是有一個偏好的資料結構。事實上，精確地說，我偏好三個緊密相關的資料結構，它們構成 ring3 Windows 95 的核心。我指的是 module（模組）、process（行程）和 thread（執行緒）。一般而言，你很難找到哪一個重要的 Windows API 函式沒有與它們有關聯。不相信？看看 *ShowScrollBar* 好了！它的第一個參數是擁有捲軸的視窗的 HWND。每一個 HWND 和一個特定的訊息佇列（message queue）有關係。而，稍後你將看到，Windows 95 的每一個訊息佇列又和一個執行緒有關係。因此在 *ShowScrollBar* 函式碼中，thread database 的資訊是必須的。

這一章中，我們將察看模組、行程、執行緒的核心資料結構。當我們觀察這些資料結構，我們常常又會遭遇另一些資料結構，迫使我們再細究下去。例如，每一個行程內含一個指標指向一個 handle table，很像 DOS 的 Program Segment Prefix（PSP）中的 handle table。而一旦進入 handle table，我們無意中發現所有重要的 KERNEL32 物件。同樣地，觀察執行緒時，我們很難忽略 Thread Information Block（TIB）的存在。TIB 在結構化異常處理中扮演重要角色。

這一章滿滿的盡是資訊。除了三個關鍵資料結構，我還丟出與它們直接發生關係的 API 函式的虛擬碼。這使你有機會看到這些資料結構的運轉情況，以及看到 KERNEL32 如何處理像執行緒同步化控制之類的題目。最後我還提供了一個 WIN32WLK 程式，那是我寫的一個用來幫助我研究關鍵資料結構的程式，讓你很容易觀察系統中的模組、行程、執行緒，並審查它們的每一個資料欄位。

如果你是一位 Windows 3.x 程式員，或許你已經熟悉所謂的 module 和 task。在 Win32 中，task 的觀念被分割爲兩部份：行程和執行緒。除此之外，表面上看 Win16 和 Win32 在 module 和 task/process 的觀念上是十分近似的。然而實際上它們相當不同。一個 Win32 的 module database 不同於一個 Win16 的 module database，而一個 Win16 的 task 結構也完全迥異於 Win32 的 thread 結構或 process 結構。

Windows 95 架構中十分有趣的所謂 "mirroring"（鏡射）性質，並沒有出現在 Windows NT 身上。Windows 95 中的每一個程式（不論 16 位元或 32 位元）執行時都會現出一個 Win16 task 和一個 Win32 process。真的，你可以利用 Win16 的 TOOLHELP.DLL 走訪 task list 並在其中看到 Win32 程式。你也可以利用 Win32 的 TOOLHELP32 走訪 process list 並在其中看到 Win16 程式。除了 task/process 互相鏡射之外，Windows 95 也爲每一個 EXE 或 DLL（不論 16- 或 32- 位元）維護一個 Win16 模組資訊。不幸的是 Win16 的 TOOLHELP.DLL 不能夠觀看 Windows 95 針對 Win32 模組所產生的 Win16 module database。然而第 7 章的 SHOW16.EXE 能夠找到它們。本章以及 WIN32WLK.EXE 集中在 Win32 這邊，第 7 章以及 SHOW16.EXE 則集中在 Win16 那邊。

在深掘模組、行程以及執行緒的細節之前，我必須先聲明，這些資訊的透露並未經過微軟公司的核准。微軟希望你不要在自己的程式碼中放進與這些資料結構有關的資訊。對於那些需要處理模組、行程、執行緒的應用程式，微軟提供的解決方案是定義在 TLHELP32.H 中的 TOOLHELP32 API 函式。

TOOLHELP32 函式提供了對於模組、行程、執行緒資料結構的有限處理能力，侷限在微軟認爲安全的籬笆之內。我必須強調，這樣的處理只是一種唯讀處理。但是，常常，微

軟認爲足夠的資料，對於系統程式員如我者是不夠的。例如，TOOLHELP32 沒有提供「列舉一個行程的 handle table」的能力。如果你需要這樣的動作，你就必須直接取其資料，一如 WIN32WLK 所做的那樣。如果你能夠使用 TOOLHELP32 達到目的而不要直接攫取資料，請那麼去做。注意，系統資料結構的盛宴應該留給訓練有素的...黑猩猩...喔不，我是指技術專家。

## Win32 模組（Modules）

就像在 Win16 一樣，一個 Win32 模組代表的是一個被 Win32 載入器載入的 EXE 或 DLL 的程式碼、資料、資源。因此，記憶體中的一個模組都對應到磁碟中的一個檔案。EXE 或 DLL 本身並不是模組。是載入器把它們讀進記憶體並產生出模組。Win32 PE 檔案格式的一個良好性質就是：將它們載入記憶體是十分簡單的。載入器利用記憶體映射檔，將 PE 檔中的某一段落映射到線性記憶體中。作業系統把一個被載入模組的所有高階資訊保持在一個結構之中，此結構被我稱之爲 module database。第 8 章詳細地描述了 PE 表頭和 module database。

應用程式使用 HMODULEs 來代表被載入的模組。在 Win16 中，HMODULE 只是一個 global heap handle，該 heap 屬於「內含 16 位元 module database 之節區」所有。第 7 章會詳細描述 Win16 模組。Win32 並沒有所謂的節區，所以需要一些其他方法來參考被載入模組。微軟的作法是讓一個 HMODULE 成爲 Win32 載入器映射 PE 檔案時的起始線性位址。例如，大部份 EXE 程式被載入於位址 0x400000（4MB）處，所以它們的 HMODULE 就是 0x400000。是的，這意味著多個 EXE 同時執行時，擁有相同的 HMODULE。這不是問題，因爲 Windows 95 和 NT 爲每一個行程維護了一個分離位址空間。

Module database 非常靠近 EXE 或 DLL 被載入後的記憶體起頭處，並且內含一些像是檔案中的 code/data sections 被載入到記憶體何處等等的資訊。模組中的碼和資料並不僅只於編譯器爲你的程式所產生的碼，還包括 import table、export table、resource directory... 等等。import table（放在 .idata section）告訴載入器說這個模組要動態聯結哪一個 DLL 中

的哪一個函式；export table 恰相反，告訴作業系統說本模組有哪一個函式要開放給別的模組呼叫。Resource section 內含一個類似磁碟目錄的階層架構，使系統能夠快速尋找到特定的資源。Module database 內含如何尋找這些 sections 的資訊，以及需要的作業系統版本，以及此一程式是否爲 console 模式...等等等。

現在，戴上你的保護面罩，拿起你的乙炔噴管，讓我們把 module database 大卸八塊，看看微軟試圖隱藏什麼。

令人驚訝的是，module database 的格式是公開的。 Win32 中的一個 module database 其實就是 EXE 或 DLL 的 PE 表頭。看看 WINNT.H，你會發現 IMAGE_NT_HEADERS 結構，它由一個 DWORD 和兩個子結構組成。IMAGE_NT_HEADERS 結構中的資訊就是 Windows 95 內部用來尋找被載入之 EXE 或 DLL 中的程式碼、資料、資源用的。

雖然我可以用滿滿數頁揭露 IMAGE_NT_HEADERS 結構中的每一個欄位，我卻不打算這些做。爲什麼？因爲 IMAGE_NT_HEADERS 結構和 PE 表頭是如此重要，值得有專門的一章探討之 -- 本書的第 8 章。

Win32 要求每一個行程有自己的模組串列。如果模組沒有隱式聯結（implicitly link，譯註）DLLs，或說它是以 *LoadLibrary* 載入 DLLs，行程就沒有辦法在記憶體中看到那些 DLL 模組 -- 甚至即使其他行程載入了它們也一樣。這在 Win16 是相當不同的，Win16 的每一個被載入模組可以被所有其他行程看到，不論它們有沒有參考到該 DLL。雖然，「每一個 Win32 行程有自己的模組串列」這件事對於安全防護性和強固性有好的影響，從「利用可共享之程式碼和資源以節省空間」的角度來看卻不怎麼好。畢竟，如果你執行三份 WINHELP，WINHELP 的碼不應該被載入三次，對不對？

> 譯註：所謂 implicitly link（隱式聯結），是指程式在聯結時期即與 DLLs 所對應的 import libraries（.LIBs）做靜態聯結，於是可執行檔中便對所有 DLL 函式都有一份重定位表格（relocation table）和待修正記錄（fixup record）。當程式被 Windows 載入器載入記憶體中，載入器會修正所有的 fixup records，使它記錄 DLLs 函式的真正位址，於是動態聯結才可以順利進行。

KERNEL32 必須面對一個費力的選擇。從應用程式眼光來看，每一個行程有自己的模組串列是不錯；但從 KERNEL32 的角度來看，單一模組串列比較容易達到程式碼和資源的共享。只要有一個新的行程開始執行，或一個新的 DLL 被載入，KERNEL32 就可以快速檢查獨一無二的全域性模組串列，看看那個 EXE 或 DLL 是否已經載入？如果是，KERNEL32 就簡單地改變其計數值。如果不是，KERNEL32 才需要在記憶體中爲它產生新的模組。

KERNEL32 利用兩個資料結構來維護一個全域性模組串列，並且使它看起來好像每一個行程有自己的一個串列。第一個是資料結構 IMTE（Internal Module Table Entry），第二個資料結構是 MODREF。

## IMTEs（Internal Module Table Entries[?]）

如圖 **3-1** 所示，全域性 KERNEL32 模組串列其實只不過是 IMTEs 指標所組成的陣列。稍後所列的虛擬碼中，我將以 pModuleTableArray 代表指向此一陣列的指標。這塊陣列所使用的記憶體是從 KERNEL32 heap 中配置而來，那是一般的 *HeapAlloc*（第 5 章描述之）所獲得的一個區塊。當新模組被載入或踢出記憶體，KERNEL32 利用 *HeapReAlloc* 動態擴張或縮減陣列大小。當 KERNEL32 產生一個新的 IMTE，它會搜尋 pModuleTableArray 中的空白元素；找到了一個，就把 IMTE 指標放進去。這個元素的陣列索引值稍後在我們搜尋 MODREFs 時將有吃重的演出。pModuleTableArray 的第一個元素（索引爲 0）用來表示 KERNEL32.DLL 模組。

讓我快速敘述一下重點。pModuleTableArray 中的每一個非零元素都代表系統中一個被載入的 EXE 或 DLL。每一個這樣的元素都是一個 IMTE 指標（在虛擬碼中我以 PIMTE 表示）。雖然 module database 的格式是公開的（其實就是 IMAGE_NT_HEADERS 結構），IMTE 的格式卻沒有公開。

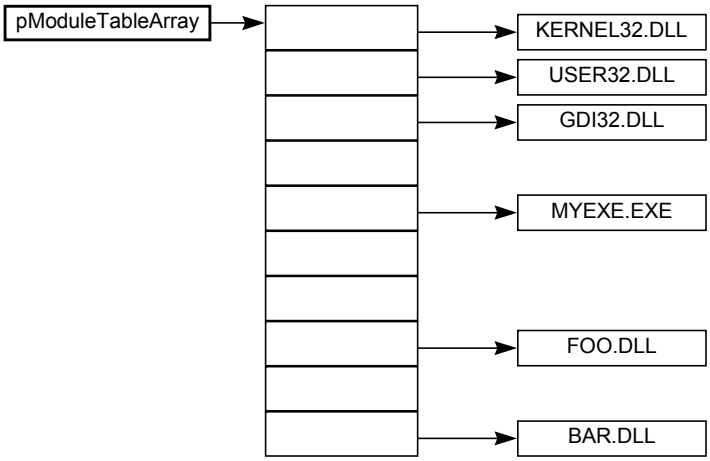



**圖 3-1　全域性的模組串列是一個由 IMTEs 指標所組成的陣列。**

## IMTE 結構

WIN32WLK 原始碼中的 MODULE32.H 內含一個 IMTE 結構定義。每一個 IMTE 有以下欄位：

**00h　DWORD　un1**

這個欄位用來放置某種旗標值。

**04h　PIMAGE_NT_HEADERS　pNTHdr**

這個指標指向一個 IMAGE_NT_HEADERS 結構。然而，它只是該結構內容的一份拷貝。這份拷貝所用的空間是從 KERNEL32 heap 配置來的，所以對所有行程而言都是可見的。而主要的那個 IMAGE_NT_HEADERS 結構位於模組的基底位址附近（可能在 2GB 之下），只能給「載入此一模組」之行程存取之。經由拷貝的動作，KERNEL32 就可以輕易地把任何被載入模組的資訊讓所有行程看見，不需要呼叫 ring0 碼以切換 memory context。

**08h　DWORD　un2**

此欄位的意義不清楚。其值似乎總是 -1。

**0Ch　PSTR　pszFileName**

內含一個指標，指向用以建立這個模組的 EXE 或 DLL 的完整檔名。你可以呼叫 *GetModuleFileName* 獲得此完整檔名，更可以利用 *GetModuleHandle* 再將檔名轉換爲 handle。放置完整檔名的區塊是從 KERNEL32 heap 中配置來的。

**10h　PSTR　pszModName**

內含一個指標，指向模組名稱。Win32 的模組名稱就是 EXE 或 DLL 名稱。例如，C:\WINDOWS\CALC.EXE 的模組名稱就是 CALC.EXE。pszModName 事實上是指入前面所說的 pszFileName 之中。以前例而言，它將指向第二個 '\' 之後的 "CALC.EXE"。

**14h　WORD　cbFileName**

此值表示 pszFileName 所指之字串的字元個數。*GetModuleHandle* 可以利用此一欄位快速決定字串是否吻合。

**16h　WORD　cbModName**

此值表示 pszModName 所指之字串的字元個數。*GetModuleHandle* 可以利用此一欄位快速決定字串是否吻合。

**18h　DWORD　un3**

此欄位的意義不清楚。

**1Ch　DWORD　cSections**

此一模組所含之 sections（.text、.idata 等等）的個數。這個值也可以從 IMAGE_FILE_HEADERS 結構（第８章）中的 NumberOfSections 欄位獲得。

**20h DWORD un5**

此欄位的意義不清楚。其值總是 0，但是在 COMCTL32.DLL 中，它內含一個指標，指向 KERNEL32 heap 中的一塊區域。

**24h DWORD baseAddress/Module Handle**

這個欄位內含模組的基底位址。在 Win32 中模組的基底位址和 HMODULE 以及 HINSTANCE 相同，所以此欄位也可以被解釋爲模組的 HMODULE 和 HINSTANCE。EXE 的基底位址幾乎總是爲 0x40000。system DLLs 的基底位址在 2GB 之上，是共享記憶區。第８章對於基底位址有更多敘述。

**28h WORD hModule16**

此欄位內含一個 selector，其線性位址指向一個 Win16 NE module database（其格式在第７章詳述）。對於 Win32 程式，NE module database 內含重要資訊，包括在哪裡可以找到資源等等。這是必要的，因爲處理資源的程式碼位在 Win16 的 KRNL386.EXE 和 USER.EXE 中。請注意，hModule16 並非是以 Win16 的 *GlobalAlloc* 配置而得，所以這個 selector 並不像 Win16 的全域性記憶體 handle，因爲這個因素和一些其他因素，Win16 TOOLHELP 沒辦法看到 Win32 模組所鏡射的那個 NE 模組。

**2Ah WORD cUsage**

此欄位內含模組的參用計數。如果有三個 CALC.EXE 正在執行，CALC.EXE 的 module database 的此一欄位即爲 3。

如果 Win32 有一個名爲 *GetModuleUsage* 的函式，我相信它一定會報告此欄位的值。然而，Win32 SDK 文件卻這麼說：

> *GetModuleUsage* 是一個陳腐老舊的函式。它被用來簡化 16 位元 Windows 程式的移植過程。每一個 Win32 程式在它自己的位址空間中跑。

你該何去何從？相信文件？還是相信 KERNEL32 真正的行爲？

**2Ch　DWORD　un7**

此欄位的意義不清楚。通常它內含一個合法的指標，指向一塊 KERNEL32 heap 區域。

**30h　PSTR　pszFileName2**

這個 PSTR（以及後續三個欄位）有點神秘。它們似乎爲那些「處理本結構之 0Ch 至 16h 偏移位置」的相同函式服務。這個欄位指向 EXE 或 DLL 完整檔名的一份不同拷貝。pszFileName 和 pszFileName2 所指的字串似乎總是相同。

**34h　WORD　cbFileName2**

這個欄位內含 pszFileName2 的長度。它總是和 cbFileName 欄位相同。

**36h　DWORD　pszModName2**

這個欄位指向 pszFileName2 之中的模組名稱。它總是和 pszModName 相同。

**3Ah　WORD　cbModName2**

這個欄位內含 pszModName2 的長度。它總是和 cbModName 欄位相同。

IMTE 之中維護兩個指標分別指向模組的檔案名稱和模組名稱，這是很奇怪的事。我不確定是爲了什麼，然而關於模組名稱倒是有些好消息。Win16 之中的 EXE 或 DLL 模組名稱是 resident names table 的第一筆資料項，設定在應用程式的 .DEF 檔中。由於 Win16 載入器假設模組名稱和檔案基本名稱（去除路徑之後的那個）相同，所以如果情況並非如此的話，會引起一些奇奇怪怪的問題。例如，如果你有兩個相同名稱的 DLLs，分別在不同的磁碟目錄，載入器只可能載入其中一個。企圖載入另一個只會引起前一個的參用計數值加 1 而已。真糟糕！

幸運的是 Win32 已經解決這個問題。你交給 *GetModuleHandle* 的模組名稱必須和 EXE 或 DLL 的檔案名稱相同。因此，程式A可以載入 \BAR 子目錄下的 FOO.DLL，程式B可以載入 \BAZ 子目錄下的 FOO.DLL。

但是有一種情況微軟還沒有解決：程式企圖在同一時間使用有著相同名稱的兩個不同的DLLs。例如，程式A隱式（implicitly）聯結 FOO.DLL，載入器在 \BAR 目錄中找到它；稍後程式又使用 *LoadLibrary* 企圖載入 C:\BAZ\FOO.DLL。那麼到底是C:\BAZ\FOO.DLL會被載入，還是 C:\BAR\FOO.DLL 的參用值加 1？微軟的文件上沒有說。在請教 Windows 95 載入器的設計者之後，他說兩個不同的 FOO.DLL 會被載入記憶體中。我自己也在操作 SoftIce/W 觀察模組串列時，看到了這種行爲。

## MODREF 結構

現在你已經知道 KERNEL32 如何以一個全域陣列（內含 IMTEs 指標）維護所有模組。我將把其他的謎底一併揭開。稍早我曾提過，一個行程如何擁有它自己的模組串列，我也說它對其他行程所載入的其他模組一無所知。把每個行程都有的模組串列和全域性模組表格連接在一起的就是 MODREF 結構。每個行程（除了奇怪的 KERNEL32）都有的模組串列事實上是一個 MODREF 串列，其中一個 MODREF 是針對行程本身（也構成一個模組），其他 MODREFs 是針對行程所使用的每一個 Win32 DLLs。MODREFs 的記憶體來自 KERNEL32 heap，那是處於 2GB 之上 -- 可共享區域。因此，即使 MODREFs 厲行「每一個行程有一個模組串列」的概念，MODREF 串列事實上是誰都看得到的。WIN32WLK 能夠走訪每一個行程的模組列印就足以證明這一點。

MODREFs 串列的頭放在 process database（稍後討論）之中。每一個 MODREFs 結構內含一個索引，指向 pModuleTableArray 表格。**圖 3-2** 顯示 MODREFs 和 IMTEs 的關係。

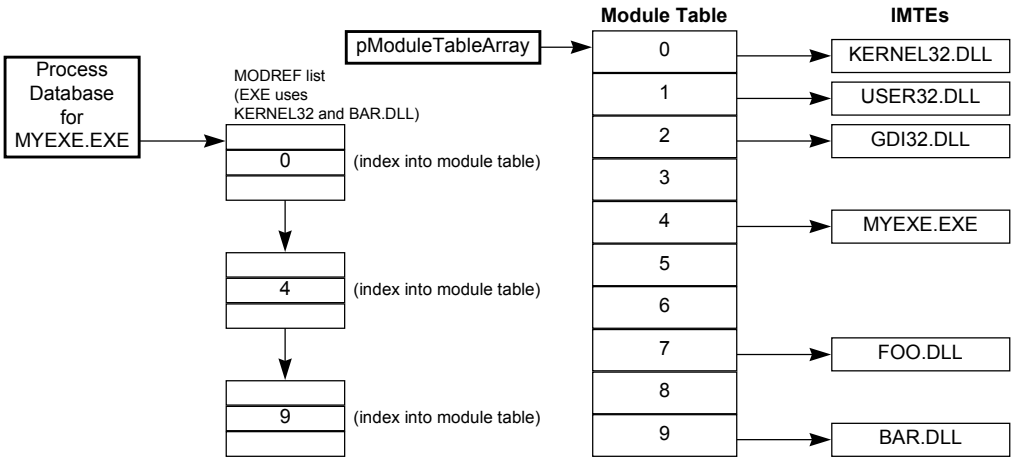



**圖 3-2　全域性的 IMTEs 表格，和「每一行程都有一個」的 MODREFs 串列。**

WIN32WLK 原始碼中的 MODULE32.H 內含一個 MODREF 結構定義。每一個 MODREF 有以下欄位：

### 00h　PMODREF　pNextModRef

這個指標指向串列中的下一個 MODREF 結構。串列最後以 NULL 收尾。只要從行程的 process database 中取得 MODREFs 串列的頭，然後一一追蹤下去，就可以把行程所用到的每一個模組找出來。WIN32WLK 之中就有這樣的示範。

### 10h　WORD　mteIndex

一個以 0 爲基準的值，代表 pModuleTableArray 陣列的索引。

### 18h　PVOID　ppdb

這是一個 PPROCESS_DATABASE，也就是一個指向 PROCESS_DATABASE 結構的指標，它提供一個從 MODREF 回頭聯結至「擁有此一 MODREF 之行程」的資料。稍後我會探討 PROCESS_DATABASE。

由於 Windows 95 必須看起來像是每一個行程有自己的一個模組串列，所以和模組有關的 API 函式如 *GetModuleHandle* 不能夠在全域模組表格（pModuleTableArray）上動作，它們只能夠在自己的 MODREF 串列上動作，然後再藉由獲得的索引參考到全域模組表格的元素項目。例如，*GetProcAddress* 只能夠尋找陳列於 current MODREF 串列中的模組。即使這個模組已經被其他行程載入，*GetProcAddress* 還是無法找到它 -- 除非現行（current）行程也載入了它。

# 與模組有關的 API 函式

現在讓我們看看一些使用到 module database 的 Win32 函式。

## GetProcAddress 和 IGetProcAddress

*GetProcAddress* 是 Win32 程式設計的一個關鍵函式，因爲它是動態載入 DLL 的方法。（相反的方法則是所謂隱式載入，implicitly link）。只要指定模組（HMODULE）和函式（名稱或序號皆可），*GetProcAddress* 就會傳回函式進入點位址。爲了完成任務，*GetProcAddress* 首先必須找出特定模組的 module database，然後再走訪其輸出函式表格（exported function table），以獲得位址。

*GetProcAddress* 其實只不過是進行參數驗證而已。它先確認 lpszProc 參數是個字串還是個序號。如果高字組（high WORD）是 0，低字組（low WORD）就是序號。如果高字組不是 0，lpszProc 就被認爲是一個 PSTR，於是函式碼掃描這個字串，尋找 NULL 結束字元。如果 PSTR 不符規定，掃描過程中就會發生異常情況，被結構化異常處理常式捕捉，傳回 0（表示失敗）。稍後我討論執行緒時，會再補充所謂的結構化異常處理。如果一切順利，控制權被交給 *IGetProcAddress*，那裡才真正取得函式位址。

**GetProcAddress 的虛擬碼**

```
// 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* 把尋找函式位址的動作管理在一個高層面上，留下瑣碎的工作給兩個低階函式。*IGetProcAddress* 首先做一些執行緒同步控制動作，確保目前的執行緒不會在不適當時機被中斷掉。接下來它呼叫 *MRFromHLib* 取得一個指標，指向 MODREF。*MRFromHLib* 是 KERNEL32 的內部函式，掃描此一行程的 MODREFs 串列，搜尋看看誰的 HMODULE 吻合要求。找到之後 *IGetProcAddress* 即利用 MODREF 結構中的模組表格索引值，尋找對應的 IMTE。

*IGetProcAddress* 的第二個動作就是尋訪函式位址。由於 *IGetProcAddress* 可能獲得函式名稱或序號以爲參數，所以它必須先決定是哪一種型態，才能呼叫下層函式。如果傳來的是序號，就呼叫 *x_FindAddressFromExportOrdinal*；如果傳來的是字串，就呼叫 *x_FindAddressFromExportName*。這兩種情況如果都未能找到函式位址，*IGetProcAddress* 就會吐出一個錯誤的診斷訊息並傳回 0。

如果你搜尋的是 KERNEL32 函式，*IGetProcAddress* 並不允許你以序號來指定函式。爲什麼微軟要做出這種令人厭惡的動作？因爲在 KERNEL32 之中有一堆未公開函式，只以序號輸出（請看附錄A）。由於這些函式名稱不在 KERNEL32.DLL 之中，所以它們也不存在於 KERNEL32 的 import library 之中，因此應用程式就不能夠呼叫這些微軟保留品。在 ***Unauthorized Windows 95*** 一書中 Schulman 寫了一些程式，呼叫到未公開的 KERNEL32 函式。在 Windows 95 後期版本中這些程式全部失靈。

自從 beta3 之後，直接呼叫 KERNEL32 未公開函式的方法已經不再管用。然而，還是有許多靈巧的程式員可以破繭而出。大家都知道 *GetProcAddress* 可以獲得函式指標。如果你知道未公開函式的序號，不就可以這麼做了嗎？不！*IGetProcAddress* 中一段可怕的碼阻斷了這樣的企圖，它不允許 *GetProcAddress* 使用 KERNEL32 函式序號。因此，如果 Schulman 嘗試以 *GetProcAddress* 修改他那失靈的程式，他會愈行愈遠。情節會更加複雜...。

以個人觀點來說，我認爲 *IGetProcAddress* 的改變有些孩子氣的成份在。任何稱職的 Windows 95 系統程式員都可以根據 PE 模組的格式（第 8 章）寫出他們自己的 *GetProcAddress* 版本。我所使用的另一個方法是利用 Visual C++ 的 LIB.EXE 和一個 .DEF 檔，產生 KERNEL32 的 import library（含未公開函式）。WIN32WLK 就是使用這個 import library。附錄 A 對此有所討論。

讓我們回到 *IGetProcAddress* 理性的那一部份。成功找到函式位址之後，你會認爲 *IGetProcAddress* 功德圓滿。不，沒那麼快。爲了某些詭異的理由，當一個行程在 Windows 95 中被除錯器載入，它所呼叫的任何 system DLLs（位於 2GB 之上）會先通過一段特殊的碼(由載入器動態產生出來)。這些碼的目的是爲了阻止除錯器進入 ring3 system DLLs 之中。對於隱式聯結，載入器會處理每一個必要動作；然而如果使用顯式聯結（也就是利用 *GetProcAddress*），程式呼叫由 *GetProcAddress* 傳回的函式指標，將因此跳過那些小段碼。因此，*GetProcAddress* 必須先檢查是否程式正處於被除錯狀態；如果 *IGetProcAddress* 獲得的位址高於 2GB，*IGetProcAddress* 會先尋找對應的小段碼，並傳回那一小段碼的位址。

如果函式沒有找到，*IGetProcAddress* 設定錯誤代碼，讓 *GetLastError* 傳回 ERROR_PROC_NOT_FOUND。最後，*IGetProcAddress* 離開 critical section（那是它在函式一開始所採行的同步控制機制）。

**IGetProcAddress 的虛擬碼**

```
// 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*（這是我取的名稱，不是微軟的名稱）是 KERNEL32 的一個核心函式。它不只被 *GetProcAddress* 呼叫，也被 PE 載入器呼叫（在修正對隱式載入之函式的呼叫時使用）。

*x_FindAddressFromExportOrdinal* 十分依賴 PE 檔中的 IMAGE_NT_HEADERs 和 .edata 的內容。它們都被映射到記憶體中，用以建造模組。所以我要再次強調第8章 PE 格式的重要性 -- 即使你不打算直接在 PE 格式上做點什麼動作。

雖然 *x_FindAddressFromExportOrdinal* 之中有相當量的碼，但其觀念其實相當簡單。在模組的 export table（也就是 .edata）中，你可以獲得一個 RVA（Relative Virtual Address）陣列，用來描述模組中的每一個輸出函式。這個陣列被稱爲 export address table。陣列中的第一個元素內含輸出序號爲 1 的函式的 RVA，第二個元素內含輸出序號爲 2 的函式的 RVA。依此類推。*x_FindAddressFromExportOrdinal* 唯一要做的就是進入陣列之中取得 RVA，然後把 RVA 加上模組基底位址，使它成爲一個可用的線性位址。不過這其中又有兩點需要注意。

第一點是，*x_FindAddressFromExportOrdinal* 需要計算序號基底。在 PE 檔中，最低輸出序號是爲基底。這可以使放置輸出函式位址的表格比較小一些。例如我們說一個 DLL 的輸出序號是 100 至 109。直觀想法則需要一個 110 個元素的陣列，而只有最後 10 個

有用。爲了節省空間，聯結器設定序號基底爲 100，於是陣列只要 10 個元素就夠了。找到一個輸出函式後，*x_FindAddressFromExportOrdinal* 要記得把序號基底加到索引值上面，才成爲真正可用的索引值。

第二點是，*x_FindAddressFromExportOrdinal* 必須處理轉交函式（forwarded function）。轉交函式在第 8 章有比較詳細的解釋。目前你只要知道，所謂轉交函式就是針對「位於另一 DLL 中的輸出函式」的一個別名。例如 Windows NT KERNEL32.DLL 的 *HeapAlloc* 函式就被轉交到 NTDLL.DLL 的 *RtlAllocateHeap* 函式。輸出函式位址陣列中的轉交函式的位址總是放在 .edata section 中。它並不是輸出函式的位址，而是指向一個像是 NTDLL.RtlAllocateHeap 之類的字串。如果 *x_FindAddressFromExportOrdinal* 看到這樣的事情發生，它就分別取出模組名稱和函式名稱，然後以之呼叫 *GetProcAddress*。噢是的，如果用 *GetProcAddress* 尋找一個轉交函式，會陷入遞迴（recursive）之中。

**x_FindAddressFromExportOrdinal 的虛擬碼**

```
// 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

這是 *x_FindAddressFromExportOrdinal* 的同伴。兩個函式之間的差異在於，本函式以名稱（而非序號）爲尋找對象。其第一部份與前一個函式相同。

*x_FindAddressFromExportName* 函式的實質意義在於它從哪裡尋找函式名稱以吻合其 lpszProc 參數。如果找到一個吻合字串，此函式就根據 AddressOfNameOrdinals 陣列，把字串陣列索引轉換爲輸出函式位址表格的索引。然後，*x_FindAddressFromExportName* 就可以尋找輸出的 RVA 並傳回給其呼叫者。然而這麼做會使它跳過前一節所說的兩種特殊情況（一是序號基底，一是除錯戳記）。因此，這個函式其實是把它所發現的序號交給 *x_FindAddressFromExportOrdinal*，讓後者做它該做的事情。

簡單再說一次，函式位址可以根據名稱或序號來取得。然而在底層動作中，位址總是根

據序號來搜尋。當你把一個函式名稱交給 *GetProcAddress*，或是以名稱輸入一個函式，KERNEL32 只不過是注入一個額外步驟，把字串先轉換爲序號罷了。

**x_FindAddressFromExportName 的虛擬碼**

```
// 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* 接受一個 HMODULE 參數，傳回對應的 EXE 或 DLL 的完整路徑。*GetModuleFileNameA*（譯註）本身很簡單，只不過做參數確認的動作。在確認過 lpszPath 參數（準備用來放置檔名）爲合法值之後，*GetModuleFileName* 跳到 *IGetModuleFileName* 函式中。

> 譯註：函式名稱之最後帶 'A' 者，表示這是個 ANSI 字串。若爲 'W' 表示這是個 unicode 字串。

如果不是爲了考慮 ANSI 和 OEM 兩種檔名，*IGetModuleFileName* 會簡單得多。KERNEL32 中的 *SetFileApisToANSI* 和 *SetFileApisToOEM* 兩函式讓呼叫者指定使用 ANSI 字串或 OEM 字串。Windows 95 內部以 ANSI 字元儲存所有的檔名，有必要時再轉換爲 OEM 字元。*IGetModuleFileName* 的實質內容中有一部份就是負責轉換動作。

除了檔名這一主題，*IGetModuleFileName* 是十分簡單的。所有它需要做的事情就是把完整檔名從正確的 IMTE 拷貝到輸出緩衝區內。然而由於每一個行程認爲它有自己的模組串列，*IGetModuleFileName* 不能夠直接搜尋 pModuleTableArray。*IGetModuleFileName* 使用 *MRFromHLib*（稍早我曾經描述過）找出這個模組的 MODREF。有了 MODREF，*IGetModuleFileName* 使用其 mteIndex 欄位進入 pModuleTableArray 並獲得其 IMTE 指標。一旦有了 IMTE 指標，剩下的事情就是把 IMTE 的 pszFileName 欄位內的字串拷貝到緩衝區中 -- 那是 *GetModuleFileName* 的一個參數。

**GetModuleFileNameA 的虛擬碼**

```
// 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 的虛擬碼**

```
// 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* 函式執行 *GetModuleFileName* 的相反工作。給予一個模組名稱，這個函式會傳回其對應的 HMODULE（也就是基底位址）。不幸的是微軟文件中對於模組名稱的解釋有點模糊。不過，虛擬碼可以澄清所有疑問。總的來說，模組名稱可以是 EXE 或 DLL 的一個基礎名稱，也可以是一個完整路徑。如果副檔名是 DLL，則可省略不現。因此，下列四種情況都可以說是 C:\WINDOWS\SYSTEM\USER32.DLL 的模組名稱

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

真正的 *GetModuleHandle* 函式碼很短，它只是確認 lpszModule 參數爲一個合法的字串

指標。如果確是如此，*GetModuleHandle* 就跳到 *IGetModuleHandle* 去。就像 *IGetModuleFileName* 一樣，*IGetModuleHandle* 的核心也有一部份用來執行ANSI 和 OEM 字串之間的轉換（如果需要的話）。這部份碼首先把模組名稱全部改爲大寫，以便稍後的比對可以快速一些。接下來檢查是否有一個副檔名在最後面。如果沒有副檔名，就自動加上一個 .DLL。

接下來的函式核心碼呼叫兩個輔助函式：*x_GetMODREFFromFilename* 和 *x_GetHModuleFromMODREF*。首先 *x_GetMODREFFromFilename* 掃描此一行程的 MODREFs 串列，直到發現一個吻合的檔名，然後傳回該 MODREF 的指標。接下來 *x_GetHModuleFromMODREF* 獲得一個 PMODREF 參數，傳回對應的 HMODULE。這兩個輔助函式描述於下。

**GetModuleHandleA 的虛擬碼**

```
// 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 的虛擬碼**

```
// 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

這個函式（這是我的命名）掃描某一行程的 MODREFs 串列，把其中的檔案名稱拿來和函式參數 lpszModName 比對。如果吻合，就傳回 PMODREF，否則傳回 NULL。

有趣的是，*x_GetMODREFFromFilename* 做的字串比對動作不只是一個，也不只是兩個，而是四個。第一次比對基礎名稱（例如 KERNEL32.DLL），如果失敗，再比對完整路徑。如果又失敗，再比對基礎名稱的第二份副本，以及完整路徑的第二份副本（譯註）。只要有一次比對成功，就傳回 MODREF 指標。

> 譯註：什麼是第二份副本？請看 IMTE 結構中的 pszFileName2 和 pszModName2 欄位。

爲了加速比對，*x_GetMODREFFromFilename* 首先計算字串參數的長度。由於儲存在 MODREF 結構中的字串，其長度已記錄於結構之中，所以 *x_GetMODREFFromFilename* 先比對長度是否吻合。如果長度不吻合，也不必比了。

**x_GetMODREFFromFilename 的虛擬碼**

```
// Parameters:
//      PSTR    lpszModName
// Locals:                    譯註：原文書少了這一行
//      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

這個函式獲得一個 PMODREF 做爲參數，傳回對應的 HMODULE（也就是基底位址）。函式只要從 MODREF 中取出指向 module database （一個 IMAGE_NT_HEADERS 結構）的指標，然後從該結構中取出模組基底位址，那就是一個 HMODULE。

**x_GetHModuleFromMODREF 的虛擬碼**

```
// 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 物件

此刻我很希望能夠立刻跳去討論行程和執行緒，但是我不能夠在討論所謂的 KERNEL32 物件（或稱 K32 物件）之前這麼做。雖然佔用 KERNEL32 heap 記憶體的任何東西我們都可以說它是 KERNEL32 物件，但我的心中另有一套定義。

K32 物件是關鍵性的系統資料結構，放在 KERNEL32 heap 之中。有各式各樣的 K32 物件，統統都以相同的表頭開始。決定它是否爲一個 K32 物件的方法就是詢問一個問題：應用程式之中可有 handles 代表此一物件？例如，應用程式可以擁有 file handles 或 event handles，所以 file 和 event 都是 K32 物件。我不曾看過任何應用程式碼擁有 MODREF 或 IMTE 的 handles，所以它們並不是 K32 物件。

每一個 K32 物件都以一個共通的表頭開始。此表頭擁有以下欄位：

### 00h　DWORD

物件型態。此值決定後續的結構成員如何解釋。

### 04h　DWORD

這是物件的參用次數（reference count），代表物件被使用的次數。例如，當你呼叫 *GetFileInformationByHandle*，被詢問的檔案物件的參用次數就會累加 1；而在函式回返之前，參用次數又會減 1。

現在，你或許渴望知道有哪些 K32 物件型態。以下就是一份清單：

```
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; 請看 FindFirstChangeNotification)
K32OBJ_CONSOLE(0x9)
K32OBJ_SCREEN_BUFFER(0xA)
K32OBJ_MEM_MAPPED_FILE(0xB；請看 CreateFileMapping)
K32OBJ_SERIAL(0xC)
K32OBJ_DEVICE_IOCTL(0xD；請看 DeviceIoControl)
K32OBJ_PIPE(0xE)
K32OBJ_MAILSLOT(0xF)
K32OBJ_TOOLHELP_SNAPSHOT(x10；請看 CreateToolhelp32Snapshot)
K32OBJ_SOCKET(0x11)
```

本章剩餘部份，我們主要的焦點放在行程物件和執行緒物件（IDs 5 和 6）。一個 process database 其實就是一個 K32_PROCESS 物件，而一個 thread database 其實就是一個 K32_THREAD 物件。就像你在「什麼是 process handle？什麼是 process ID？」一節即將看到的，一個 process handle table 其實就是一個指標陣列，每一個指標指向各式各樣的 K32 物件。整個 KERNEL32 和 VWIN32.VXD 中，程式碼會先檢查物件的第一個 DWORD，以確定它所處理的物件型態。

如果你熟悉 Win16 Kernel，你可能會注意到這裡所說的一切與 Win16 task 和 module 不同之處。這裡的 8 位元組 Win32 物件表頭之中沒有任何欄位可以儲存串列指標。而在 Win16 中，當你獲得第一個 task 或 module，你需要的每一樣資訊（用以繼續追蹤下一個 task 或 module）俱在。Windows 95 的 KERNEL32 有它自己的 code section（LSTMGR.C）用來維護 K32 物件串列。

# Windows 95 行程（Processes）

進行到這裡，是探尋行程（processes）的一般定義的時候了。讓我們開始吧。行程其實就是一大堆物件的擁有權的集合。也就是說，行程擁有物件。行程可以擁有記憶體（更精確說是擁有 memory context），可以擁有 file handle，可以擁有執行緒，可以擁有一串列的 DLL 模組（被載入於此一行程的位址空間中）。我還可以說上一大堆，但我想你已經有了一個概念。

> 注意，行程並不代表「執行事實」（執行緒才是）。行程也不是 EXE 檔。在被載入之前，一個 EXE 檔案只不過是一個程式。只有在被載入之後，Windows 95 才爲它產生一個行程。換句話說一個行程關係到一個磁碟檔案（但 KERNEL32 是個例外。本章最後討論 WIN32WLK 時我們再來說它）。

一旦 Windows 95 產生一個行程，它也產生一個 memory context，容納行程的執行緒在其中執行。此外，Windows 95 也產生出第一個執行緒，用來執行行程本身。如有必要，行程可以再產生執行緒。系統還會產生一個 file handle table，行程可以在其中持有一些開啓的檔案。最後，也是最重要的，Windows 95 產生一個 process database，用以表現這個行程。

Process database 是一種 K32 物件，內含與行程有關的大量資訊。稍後我們將在「Windows 95 Process Database（PDB）」一節中詳細觀察其欄位。Process database 所使用的記憶體來自 KERNEL32 heap，因此所有的 process database 都可以被其他行程看到。WIN32WLK 就是這麼做出來的。

Process database 含有一系列的執行緒、一系列的被載入模組、預設之 process heap 的 handle、指向 process handle table 的指標、以及指向 memory context 的指標（請看第５章）。此外還有更多更多的東西。

# 什麼是 process handle？什麼是 process ID？

在我再跨一步之前，我需要先釐清常感困惑的 process handles 和 process IDs。兩個看起來類似的 Win32 函式：*GetCurrentProcess* 和 *GetCurrentProcessId* 迷惑了不少程式設計者。事實上其間差異是很容易分辨的。

Process handle 基本上和 file handle 一樣。它是一個不被明瞭的數值，你不能夠說它是任何東西的指標。系統內部事實上是把 K32 物件的 handle 當作 process handle table 的索引。而從該陣列中獲得的，才是一個 K32 物件指標。然而，由於應用程式不需要直接處理 handle table，所以 process handle 是沒有用的。

記住，因爲每一個應用程式有它自己的 handle table，所以不同的行程在各自的位址空間中擁有相同的 process handle 是絕對可能的。例如，正常而言每一個行程都有一個 process handle 爲它而開，其值總是 1。也就是說，process handle 並不是可以賴以判別行程的資料。另一個例子是：如果程式爲自己這個行程打開另一個 process handle，那麼就有兩個 handles，對應至同一個行程。

在 *GetCurrentProcess* 函式碼中你更可以確定，process handle 並不是可以用來判別行程的資料：

**GetCurrentProcess 函式的虛擬碼**

```
// 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;
```

是的，忽略掉「它另外又呼叫其他函式」的事實，*GetCurrentProcess* 只不過是固定傳回 0x7FFFFFFF 罷了。不管誰呼叫這個函式，它都獲得 0x7FFFFFFF。0x7FFFFFFF 是一個魔術數字，KERNEL32 把它解釋爲「使用現行的行程」。面對 0x7FFFFFFF，那些個需要 process handle 的 KERNEL32 函式會把它替換爲現行的 process handle。還需要更多證據以證明 process handle 只在自己的 context 中才有作用嗎？我想不需要了！

現在讓我們看看 process IDs。***Unauthorized Windows 95*** 一書中說，早期的 Windows 95 版本（從 beta1 開始）把 process database 的位址當做一個 process ID。由於 process database 被放置在可共享的記憶體中，所以 process database 的位址必須保證絕不衝突。***Unauthorized Windows 95*** 書中擴充了 *GetCurrentProcessId* 的使用，利用它來獲取一個指向現行之 process database 的指標，於是就可以根據它來取得關鍵性的欄位內容。不幸的是，微軟的 KERNEL32 小組在新版本中改變了 *GetCurrentProcessId* 碼：

**GetCurrentProcessId 函式的虛擬碼**

```
x_LogSomeKernelFunction( function number for GetCurrentProcessId );

   return PDBToPid( ppCurrentProcess );
```

再一次，如果我們忽略掉它另外呼叫其他函式的話，*GetCurrentProcessId* 只不過是把一個全域變數 ppCurrentProcess 交給 *PDBToPID* 函式。讓我們停下來對這裡多了解一些，因爲它對於後續章節十分重要。ppCurrentProcess 是一個「指標的指標」，指向現行的 process database。以 C 語言來說，就是 **ppCurrentProcess 指向現行的 process database。

之所以需要兩次間接指向，基於一個魅惑的原因，第 6 章我會提到。現在，你只要記住 ppCurrentProcess 是 KERNEL32.DLL 中的全域變數，允許 KERNEL32 搜尋現行行程的 process database 即可。爲了盡量簡化事情，我將在虛擬碼中使用 ppCurrentProcess 變數，我假裝它是一個指向 process database 的指標，而不是一個指標的指標。

好，如果 KERNEL32 有一個指標，指向目前的 process database，爲什麼 *GetCurrentProcessId* 不直接把它傳回就好呢？我們得看看 *PDBToPID* 做什麼事：

**PDBToPID 函式的虛擬碼**

```
// 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;
```

喔喔，是真的嗎？是的！"Obsfucator" 一詞竟然赤裸裸地出現在微軟的二進位碼中。（雖然 "Obsfucator" 其實是 "Obfuscator" 的誤拼。**譯註**）。除了檢驗傳進來的是一個合法的 process database 指標之外，*PDBToPID* 的基本行爲就是將現行之 process database 指標和 ObsfucatorDWORD 兩者做 XOR 動作。這個企圖非常明顯，微軟希望阻絕「駭客」（hacker）們刺探系統內部資料結構的行爲。然而，一如我在 WIN32WLK 所展示，這只不過是一個小小的、暫時的障礙而已。

譯註：Obfuscate 是「使模糊、使困惑」的意思。

如果你驚訝 ObsfucatorDWORD 來自何處，你會狼狽地發現它在每一次系統啓動時動態產生出來。這又爲系統做了一層更好的保護。事實上不只是 process database 被保護，thread database 也被保護了。稍後我將在 *GetCurrentThreadId* 函式中顯示給你看，*GetCurrentProcessId* 和 *GetCurrentThreadId* 有近乎神秘的相似程度。

讓我做個總結。一個 process handle 就像一個 file handle。在其行程之外別無意義。至於一個 process ID，則是在各行程之間獨一無二不會衝突的數值，它是一個指標，指向 process database 結構，甚至雖然微軟在其中加了點料（Obsfucator -- 這是他們選的字眼，可不是我）。WIN32WLK 示範神奇的轉換公式，把 process ID 轉換爲一個有用的指標。

如果你看過 TOOLHELP32 的 *Process32First* 和 *Process32Next* 兩個函式，可能你會注意 PROCESSENTRY32 結構中的 th32ProcessID 欄位。這和 *GetCurrentProcessID* 傳回來的東西有關係嗎？幸運的是，答案是 Yes。WIN32WLK 程式利用了這項性質，讓 TOOLHELP32 把一些瑣屑事情先處理掉。

# Windows 95 Process Database（PDB）

Windows 95 的每一個 process database 都是一塊從 KERNEL32 heap 配置而來的記憶體。KERNEL32 通常以 PDB 縮寫字取代又臭又長的 "process database"。不幸的是在 Win116 中 PDB 是 DOS PSP 的縮寫。這會造成混淆嗎？是的，本章之中，我將以 KERNEL32 所賦予的意義來解釋 PDB。每一個 PDB 就是一個「第一字組爲 5（K32OBJ_PROCESS）」的 K32 物件。WIN32WLK 程式的 PROCDB.H 中有一個 PDB 的 C 語言定義，我們把每一個欄位看清楚些：

### 00h DWORD Type

此值必爲 5（K32OBJ_PROCESS）

### 04h DWORD cReference

參用次數（reference count）。也就是此一 PDB 被使用的次數。

### 08h DWORD un1

此欄位之真實意義未知。似乎總是 0。

### 0Ch DWORD someEvent

這是一個指向 K32OBJ_EVENT 物件的指標。Event 物件用於 *WaitForSingleObject* 這樣的函式。

### 10h DWORD TerminationStatus

當你呼叫 *GetExitCodeProcess*，傳回的就是這個值。所謂退出代碼（exit code）就是 *main* 或 *WinMain* 的回返值。它也可以被 *ExitProcess* 或 *TerminateProcess* 指定。當一個行程還在執行時，此欄位爲 0x103（STILL_ACTIVE）。

### 14h DWORD un2

此欄位之真實意義未知。似乎總是 0。

### 18h DWORD DefaultHeap

Default process heap 的位址。*GetProcessHeap* 傳回的就是這個值。

### 1Ch DWORD MemoryContext

一個指標，指向行程的 memory context。所謂 memory context，內含 page directory mapping，用以提供行程在 4GB 位址空間中的私人區域。第５章對於 memory context 有更多描述。

### 20h DWORD flags

這個旗標值的意義如下：

| 旗標名稱與內容 | 說明 |
|---|---|
| fDebugSingle 0x00000001 | 如果設立，表示行程正被除錯中 |
| fCreateProcessEvent 0x00000002 | 設立於除錯行程中（在起始之後） |
| fExitProcessEvent 0x00000004 | 可以在除錯行程中（在結束時候）被設立 |
| fWin16Process 0x00000008 | 表示一個 16 位元程式 |
| fDosProcess 0x00000010 | 表示一個 DOS 程式 |
| fConsoleProcess 0x00000020 | 表示一個 console（文字模式）Win32 程式 |
| fFileApisAreOem 0x00000040 | 請看 API 文件中的 SetFileApisToOEM 說明 |
| fNukeProcess 0x00000080 | |

| | |
|---|---|
| fServiceProcess　0x00000100 | 例如 MSGSRV32.EXE |
| fLoginScriptHack　0x00000800 | 可能是一個 Novell 網路簽入行程 |
| fSendDllNotifications　0x00200000 | |
| fDebugEventPending　0x00400000 | 例如，在除錯器中被停下來 |
| fNearlyTerminating　0x00800000 | |
| fFaulted　0x08000000 | |
| fTerminating　0x10000000 | |
| fTerminated　0x20000000 | |
| fInitError　0x40000000 | |
| fSignaled　0x80000000 | |

### 24h　DWORD　pPSP

這個值是此一行程之 DOS PSP 的線性位址。Win16 和 Win32 程式都會設定此欄位。此一線性位址總是在 1MB（真實模式 DOS 碼所能看到的最高位址）之下。請參考 28h 欄位。

### 28h　WORD　PSPSelector

這是一個 selector，指向此一行程之 DOS PSP。Win16 和 Win32 程式都有 DOS PSP。請參考 24h 欄位。

### 2Ah　WORD　MTEIndex

這裡內含一個全域模組表格（pModuleTableArray）的索引值。經由此索引值而取出之 IMTE 正是此一行程對應的 IMTE。IMTE 和 pModuleTableArray 已於本章先前討論過。

### 2Ch　WORD　cThreads

此欄位記錄此一行程擁有的執行緒個數。

### 2Eh WORD cNotTermThreads

此欄位記錄屬於行程所有而尚未結束之執行緒個數。它怎麼看都應該和上一欄位相同。

### 30h WORD un3

此欄位之真實意義未知。似乎總是 0。

### 32h WORD cRing0Threads

此欄位記錄由 VMM32.VXD 管理的 ring0 執行緒個數。對於一般程式而言，其值應該與 cThreads 欄位值相同。然而在 KERNEL32.DLL 之中，此值比 cThreads 欄位值多 1。

### 34h HANDLE HeapHandle

此欄位是一個 HEAP handle，此 HEAP 內含屬於這個行程的表格（或可能其他東西）。這裡記錄的總是 KERNEL32 的 shared heap handle。

### 38h HTASK W16TDB

這是一個 selector，指向行程相關的 Win16 Task Database（TDB）。Win16 和 Win32 程式都有 TDB selector，並且維護一個合法的 TDB。

### 3Ch DWORD MemMapFiles

一個指標，指向「此一行程所使用之記憶體映射檔所組成的串列」中的第一個節點。每一個記憶體映射檔都出現爲串列中的一個節點。節點的格式是：

```
DWORD  記憶體映射檔的基底位址
DWORD  指向下一個節點；或是 0。
```

### 40h PENVIRONMENT_DATABASE pEDB

一個指標，指向 environment database。Environment database 中內含目前的子目錄、環境變數、行程命令列、標準 handles（例如 stdin）、以及其他項目。我將在「Environment Database」一節中描述其詳細格式。

### 44h　PHANDLE_TABLE pHandleTable

這是一個指標，指向 process handle table。所有的 handles 都在這裡面，包括 file handles、event handles、process handles 等等等。在 DOS/Win16 環境中的對等物品是 DOS 的 System File Table（SFT）。關於 SFT 請參考 Schulman 等人所著的 ***Undocumented DOS***，第二版。

然而畢竟有所不同。SFT 適用於整個系統，Win32 process handle table 則只適用於一個行程。Win32 handle table 的佈局將在「Process Handle Tables」那一節描述。

### 48h　struct _PROCESS_DATABASE * ParentPDB

這是一個指標，指向父行程的 PROCESS_DATABASE。對一般程式而言父行程是 Windows 95 的檔案總管（Explorer）。MSGSRV32 則又是 Explorer 和 initial "service" processes 的父行程。

### 4Ch　PMODREF MODREFlist

這個欄位指向行程的模組串列的起頭。這也就是稍早描述過的 MODREFs 串列。

### 50h　DWORD　ThreadList

一個指標，指向此一行程所擁有的執行緒的串列。目前我還不知道這個串列的真正格式。

### 54h　DWORD　DebuggeeCB

這是一個被除錯程式的 context。當一個行程被除錯，這個欄位即指向 2GB 以上的一個區域，該區域包含一個指標，指向被除錯者的 process database。

### 58h　DWORD　LocalHeapFreeHead

這個指標指向「此一行程預設之 heap」的自由區塊串列起頭。第 5 章對於其格式有詳細的描述。

#### 5Ch DWORD InitialRing0ID

此欄位意義未知。似乎總是爲 0。

#### 60h CRITICAL_SECTION crst

這個欄位是一個 CRITICAL_SECTION，被各式各樣的 API 用來同步控制同一行程中的各執行緒。稍後在許多虛擬碼之中你會看到這個 critical section 的作用。

#### 78h DWORD un4[3]

這三個 DWORD 之真實意義未知。似乎總是 0。

#### 84h DWORD pConsole

如果這個行程使用 console（也就是說它是一個文字模式程式），此一欄位即指向一個用於輸出的 console 物件（K32OBJ_CONSOLE）。

#### 88h DWORD tlsInUseBits1

這 32 個位元表示最低的 32 個 TLS（Thread Local Storage）的索引。如果某個位元設立，表示對應的 TLS 索引被用掉了。每一個 TLS 索引都不斷累加其值，例如：

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

稍後將有一節專門討論 TLS。

#### 8Ch DWORD tlsInUseBits2

這個 DWORD 表示 TLS 之中第 32~63 個索引的狀態。請參考前一欄位。

#### 90h DWORD ProcessDWORD

此欄位之意義未明。有一個未公開函式（*GetProcessDword*）可以取出其值。

### 94h　struct _PROCESS_DATABASE * ProcessGroup

此欄位要不爲 0，要不就指向一個「行程群」中的爲首行程。所謂「行程群」是一群行程，彼此互屬。當此一群組被摧毀，其中的所有行程也一併被摧毀。注意，每一個行程都認爲自己在自己的「行程群」之中，而也因此這個欄位指向行程自己的 PDB。如果行程處在除錯狀態，它就屬於除錯器「行程群」。

### 98h　DWORD　pExeMODREF

這個欄位指向 EXE 的 MODREF（前面描述過此一結構，是模組串列中的資料項）。一般而言，EXE 的 MODREF 是模組串列中的頭，所以這個欄位通常和欄位 4Ch 吻合，除非行程又經由 *LoadLibrary* 或 *LoadModule* 載入了其他 DLLs。

### 9Ch　DWORD　TopExcFilter

這個 DWORD 內放行程的 "Top Exception Filter"。如果行程沒有安裝任何異常情況處理常式，那麼就使用這一個。這個值是經由 *SetUnhandledExceptionFilter* 函式設立的。結構化異常處理將在稍後討論。

### A0h　DWORD　BasePriority

這個 DWORD 存放的是行程的排程優先權。Windows 95 支援 32 個優先權，分爲四個等級（右邊所列是其預設優先權）：

```
Idle        4
Normal      8
High        13
Realtime    18
```

在每組之中，優先權還可以略爲調高或調低。稍後對此亦有詳細說明。

### A4h　DWORD　HeapOwnList

這個欄位指向「行程所有的 heaps」所形成的串列。預設情況下每一個行程只有一個 heap，可經由 *GetProcessHeap* 取得。然而行程也可以呼叫 *HeapCreate* 產生另一個 heap。這些 heaps 都放在串列之中。第５章對此一主題有較多的敘述。

### A8h DWORD HeapHandleBlockList

heap 之中的可搬移區塊係由 moveable handle table 來管理。每一個 heap 對應一個 table。許多個 tables 則形成一個串列。本欄位指向串列的頭。第５章對於 moveable handle table 有較多的敘述。

### ACh DWORD pSomeHeapPtr

本欄位的真正意義不十分明確。通常它是 0，如果不是 0 那麼就是一個指標，指向本行程之 default heap 的 moveable handle table。請看 A8h 欄位。

### B0h DWORD pConsoleProvider

此欄位要不爲 0，要不就是一個指標，指向 KERNEL32 的主控台物件（K32OBJ_CONSOLE）。對於 Win32 console 程式而言它似乎總是爲 0，但對於 WINOLDAP 行程而言就不是 0。WINOLDAP 係用來管理 Windows 中的 DOS 程式。

### B4h WORD EnvironSelector

這是一個 selector，指向行程的環境區。這個 selector 的基底位址（base 欄位）和 Environment Database 的 pszEnvironment 欄位的值相同

### B6H WORD ErrorMode

這個欄位內含由 *SetErrorMode* 設定的數值。KERNEL32 的 *SetErrorMode* 會下移（thunk down）至 KRNL386 的同名函式，所以這個欄位反應出 Win16 錯誤模式代碼。它們可能是：

```
SEM_FAILCRITICALERRORS
SEM_NOALIGHMENTFAULTEXCEPT
SEM_NOGPFAULTERRORBOX
SEM_NOOPENFILEERRORBOX
```

**B8h　DWORD　pevtLoadFinished**

這個欄位指向 KERNEL32 的 Event object（K32OBJ_EVENT）。當行程載入之後，此 event 即被激發。

**BCh　WORD　UTState**

此欄位之意義不明確，但根據其名稱，似乎和 Universal Thunk（譯註）有關。通常是 0。

> 譯註：所謂 Universal Thunk，是微軟爲了讓 16 位元 DLLs 能夠被 32 位元 EXEs 呼叫而開發的一套工具，包括一個 thunk compiler 以及一些 DLLs 和 LIBs。

請特別注意和 DOS 有關的欄位，一個是 PSP selector，另一個是 DOS PSP （必在 1MB 之下）的線性位址。我們已經知道 Windows 會把 INT 21h 移轉給虛擬 86 模式中的 DOS 碼（請參考 ***Unauthorized Windows 95*** 第 8 章），所以這些欄位的出現不值得驚訝。Windows NT 的 process database 中就不含 PSP 資訊。我們可以確定一件事：DOS 沒有死 -- 至少在演化自 Windows 3.1 的平台上。

我們已經看過了 process database 的結構，現在讓我們看看相關的函式。

## GetExitCodeProcess 和 IGetExitCodeProcess

*GetExitCodeProcess* 取得行程的結束狀態。它需要一個 hProcess 參數。函式的主要功能其實只是確認第二個參數是否爲合法的指標，真正的重要動作則交給 *IGetExitCodeProcess* 去做。後者利用 hProcess 尋找對應的指標（指向 PROCESS_DATABASE）。由於 hProcess 是一個 handle，所以上述動作意味著先利用索引進入行程的 handle table，再取出行程指標。*x_ConvertHandleToK32Object* 負責「增加 process database 的參用次數」等等雜務。

有了一個 PROCESS_DATABASE 指標在手，函式就可以取出 TerminationStatus 欄位的值，並將它存放在呼叫端指定的一個緩衝區中。*IGetExitCodeProcess* 會減少 process

database 的參用次數（reference count） 1 次，並留下 "must complete" 狀態。

**GetExitCodeProcess 函式的虛擬碼**

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

   Set up structured exception handling frame

   if ( lpdwExitCode )        // If 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 函式的虛擬碼**

```
// 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

此函式設定 KERNEL32 的 *UnhandledExceptionFilter* 函式位址 -- 後者將在沒有其他異常情況過濾器（exception filter）被用來處理異常情況時使用。這個函式把 TopExcFilter 的現值藏匿在 process database 之中，然後以參數中的值取代之，並傳回原先的值。

**SetUnhandledExceptionFilter 函式的虛擬碼**

```
// 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

此函式要求一個 process ID 做爲參數，並傳回一個 process handle。process handle 然後可以被交給像 *ReadProcessMemory* 或 *VirtualQueryEx* 之類函式。而你知道，TOOLHELP32 有能力給你任何行程的 process ID。因此如果你能組合這兩股力量，你就可以有大作爲。奇怪的是 Windows 95 允許你打開一個 process handle 卻不允許你打開一個 thread handle。或許微軟認爲執行緒這邊一旦打開會造成大破壞，無法承擔。

*OpenProcess* 首先把 process ID 轉換爲一個 PPROCESS_DATABASE。「轉換 process ID 爲 process 指標」的演算法，和「轉換 thread ID 爲 thread 指標」的演算法完全相同。接下來參數傳來的旗標值會被檢驗。最後，*OpenProcess* 呼叫一個內部函式，在目前行程的 handle table 中配置一個空間，並把 PPROCESS_DATABASE 指標存放進去。

**OpenProcess 函式的虛擬碼**

```
// 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

這個函式改變與檔案有關的 KERNEL32 函式對於檔案名稱的解釋方法。預設情況下 KERNEL32 使用 ANSI 字串做爲檔名。如果呼叫了 *SetFileApisToOEM*，就可以改用 OEM 字串。請參考本章稍早出現的 *GetModuleFileName* 和 *GetModuleHandle* 兩函式。

本函式的內部動作並不簡單。它攔截一個指向現行之 process database 的指標，並把 fFileApisAreOem 旗標設立起來。

**SetFileApisToOEM 函式的虛擬碼**

```
x_LogKernelFunction( function number for SetFileApisToOEM )

   ppCurrentProcess->flags |= fFileApisAreOem;
```

# Environment Database

Process database 的 40h 欄位中是一個指標，指向一個重要的資料結構，內含與行程有關的資訊。KERNEL32 內部稱此指標爲 pEDB，我把它解釋爲 "pointer to Environment Database"。就像對待 PROCESS_DATABASE 一樣，我在 PROCDB.H 中描述了 ENVIRONMENT_DATABASE 的結構佈局。現在我們來看看這些欄位：

### 00h　PSTR　　pszEnvironment

這個欄位指向行程的環境區。所謂環境區是標準的 DOS 環境（形式一如 string=value; string=value）。行程環境區是一塊記憶體，位於每一個行程私有的資料空間中，通常就在模組被載入的位址之上。

### 04h　DWORD　　un1

此欄位意義未明。通常總是 0。

#### 08h PSTR pszCmdLine

此欄位內含 *CreateProcess* 函式中的命令列參數內容。大部份情況下這個命令列是一個完整的 EXE 檔名。有時候它會指向空字串（\0）。

#### 0Ch PSTR pszCurrDirectory

此欄位指向目前的磁碟目錄。

#### 10h LPSTARTUPINFOA pStartupInfo

這是一個指標，指向行程的 STARTUPINFOA 結構（定義在 WINBASE.H 之中）。STARTUPINFOA 結構是 *CreateProcess* 的參數之一，可用來指定視窗的大小、標題、標準的 file handles...等等等。這個欄位所指的是該結構的一個副本。

#### 14h HANDLE hStdIn

這是一個 file handle，行程用它當做標準的檔案輸入裝置。如果沒有用到（例如一個 GUI 程式），此值爲 -1。

#### 18h HANDLE hStdOut

這是一個 file handle，行程用它當做標準的檔案輸出裝置。如果沒有用到（例如一個 GUI 程式），此值爲 -1。

#### 1Ch HANDLE hStdErr

這是一個 file handle，行程用它當做標準的錯誤輸出裝置。如果沒有用到（例如一個 GUI 程式），此值爲 -1。

#### 20h DWORD un2

此欄位意義未明。通常總是 1。

#### 24h　DWORD　InheritConsole

從名稱可以推測，此一欄位表示行程是否繼承自 console 程式。請參考 *CreateProcess* 函式的 CREATE_NEW_CONSOLE 旗標。在我的觀察中，此欄位總是 0。

#### 28h　DWORD　BreakType

這個欄位最可能用來指示 console event（例如 CTRL+C）如何處理。在我所執行過的程式中，它通常爲 0，但偶而會是 0xA。

#### 2Ch　DWORD　BreakSem

通常此爲 0，但如果程式呼叫 *SetConsoleCtrlHandle*，此欄位就會指向一個 KERNEL32 semaphore object（K32OBJ_SEMAPHORE）。

#### 30h　DWORD　BreakEvent

通常此爲 0，但如果程式呼叫 *SetConsoleCtrlHandle*，此欄位就會指向一個 KERNEL32 EVENT object（K32OBJ_EVENT）。

#### 34h　DWORD　BreakThreadID

通常此爲 0，但如果程式呼叫 *SetConsoleCtrlHandle*，此欄位就會指向一個執行緒物件（K32OBJ_THREAD），而該執行緒正是安裝此一處理常式的執行緒本身。

#### 38h　DWORD　BreakHandlers

通常此爲 0，但如果程式呼叫 *SetConsoleCtrlHandle*，此欄位就會指向一個從 KERNEL32 shared heap 中配置得來的資料結構，內放一系列安裝好的主控台控制函式（console control handler）。

現在讓我們看看一些函式的虛擬碼。這次是與 ENVIRONMENT_DATABASE 有關。

## GetCommandLineA

其實這個函式沒有太多東西可以說。它傳回命令列指標，命令列字串存放在 environment database 中。

**GetCommandLineA 函式的虛擬碼**

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

## GetEnvironmentStrings

這個函式也沒有太多東西可以說。它傳回的是 environment database 的相關指標。值得我們注意的是這個函式的真正實作碼和 SDK 說明文件之間有兩個差異。SDK 文件上說：

> 當 *GetEnvironmentStrings* 被呼叫，它會配置記憶體做爲一塊環境區。當此環境區不再被需要，它應該呼叫 *FreeEnvironmentStrings*。

這對 Windows NT 也許成立，對 Windows 95 卻不正確。

**GetCommandStrings 函式的虛擬碼**

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

## FreeEnvironmentStringA

這個函式比較有趣些。由於 *GetEnvironmentStringA* 並不真正配置記憶體，所以其實沒有什麼是 *FreeEnvironmentStringA* 必須做的事情。然而，也許純粹爲了消遣，這個函式檢查其字串參數，看看它是否吻合 environment database 中的環境區指標。如果不吻合，*FreeEnvironmentStringA* 會將 LastError 值設定爲 ERROR_INVALID_PARAMETER。

**FreeEnvironmentStringA 函式的虛擬碼**

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

   x_LogSomeKernelFunction( function number for FreeEnvironmentStringsA );
```

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

    return TRUE;
```

## GetStdHandle

這個函式和你所能想像的一樣直接。給它一個 Device ID（stdin、stdout、stderr 等等），這個函式會傳回對應的 file handle。如果你給的是一個冒牌的 device ID，此函式會失敗，並設定 LastError 代碼。

**GetStdHandle 函式的虛擬碼**

```
// 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

這個函式比 *GetStdHandle* 有趣一些。它首先驗證 handle 的確代表一個合法的 KERNEL32 物件。怎麼做呢？請 *x_ConvertHandleToK32Object* 代勞！後者會傳回一個

指標，指向對應的 KERNEL32 物件 -- 如果 handle 合法的話。*SetStdHandle* 從不使用 K32 物件指標，簡單的 NULL 檢驗是唯一需要的動作。在檢驗過 hHandle 參數的合法性之後，其餘函式碼把 hHandle 塞進 environment database 結構的適當欄位去。

**SetStdHandle 函式的底擬碼**

```
// 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;
```

# Process Handle Tables

PROCESS_DATABASE 的 44h 偏移處是一個指標，指向行程的 handle table。我將使用 handle 一詞代表可以從 handle table 中取得的東西。除了 file handles，Windows 95 還會產生其他的系統物件的 handles，像是行程啦、執行緒啦、event 啦、mutex 啦等等。請參考「KERNEL32 物件」一節。

handle 的內容理論上來講是不透明的，也就是說 handle 本身沒有辦法告訴你它究竟代表什麼東西。如果它的值是 5，你判斷不出這是一個 file handle 還是一個 mutex handle。然而，一旦你了解 Windows 95 行程的 handle tables，你就可以輕易地將一個 handle 值和其參考到的資料產生關係。

Windows 95 行程的 handle table 構造十分簡單。第一個 DWORD 放的是這個表格的最大容量（項目個數）。此值初始爲 0x30（48）。然而這並不意味行程最多只能有 48 個打開的 handles。當行程需要更多的 handles，KERNEL32 會重新配置一塊記憶體，使表格有成長空間。每次增加 0x10 個 handles。似乎並沒有明顯的上限。我寫了一個小程式，以迴路不斷打開 file handles，在超過 255 個 handles 之後仍然安好 -- 255 是 DOS 的限制。

第一個 DWORD 之後，是由許多結構所組成的陣列。每一個結構都由兩個 DWORD 構成：

```
DWORD   flags
DWORD   pK32Object
```

其中第二欄位是個指標，指向 17 種可能的 K32 物件型態（稍早我曾在「KERNEL32 物件」一節中描述過）。至於第一個欄位則是此一物件的 access control flags。這些旗標的意義視物件是何種型態而定。對於一個 K32OBJ_PROCESS 物件，這些旗標將是 PROCESS_xxx（定義在 WINNT.H 中），像是 PROCESS_TERMNATE、PROCESS_VM_READ 等等。

進行到這裡，也許你已經可以感覺到 handle 是什麼東西了。如果你猜想 handle 是一個索引，指向行程的 handle table，你對了！一旦這麼認爲，你就很容易把一個 handle 值

比對其所參考的 KERNEL32 物件型態。一個沒有用的 handle，其兩個 DWORD 一定都填滿 0。當程式配置一個新的 handle，KERNEL32 就使用 handle table 中的第一筆空白項的索引值做爲 handle。雖然瀏覽行程的 handle table 並不是一個被微軟建議的程式動作，Win32Wlk 還是提供了這項能力。使用 Win32Wlk 時，請注意 KERNEL32 所使用的 handle 型態的個數。

# Thread（執行緒）

你已經看過了模組和行程，只要再看過執行緒，就可以完成整個 KERNEL32 基礎結構之旅。行程主要是表達對 file handles、位址空間等等的擁有權，執行緒則主要表達來自模組的碼的執行事實。你看，有這麼多的東西相互關聯，我很難把什麼東西從另一個東西中完全抽離出來。例如稍早在討論行程時，我必須先提到執行緒和同步控制物件。

從抽象層面來說，執行緒是一種方便的表達方式，讓你的某一部份碼執行 -- 當其他部份碼正在等待某些外部事件發生時。將行程的各項工作進一步分配給執行緒之後，你幾乎可以消除像 "pooling loop" 這樣的動作。Pooling loop 浪費許多 CPU 時間。

任何時候，執行緒可能處於三種狀態之一。第一種是「執行中」狀態（running state）。這個時候 CPU 暫存器內容就是執行緒的暫存器值。當某個執行緒處於執行狀態，其他執行緒就處於虛懸狀態。

第二種情況稱爲「準備執行」狀態（ready to run state）。這種狀態下的執行緒沒有什麼理由不會被執行 -- 時間早晚而已。它終有一刻能夠控制 CPU。

第三種情況稱爲「阻塞」狀態（blocked state）。執行緒如果被阻塞，表示它正在等待某件事情發生。在那之前排程器不會配置執行緒執行起來。引起執行緒阻塞的東西稱爲同步控制物件（synchronization objects）。Windows 95 的同步控制物件有 critical sections、events、semaphores、mutexes 四種。

我曾經在第 2 章描述過 Windows 95 同步控制物件的基本機能，所以不打算在此重複。本書之中我不會對待這些同步控制物件像我對待行程、執行緒、模組那樣。有許多好書，像是 Jeffrey Richter 的 ***Advanced Windows***，對於同步控制物件的使用就有詳細描述。本書假設你知道同步控制物件存在，並且知道它們如預期般運作。

最初，每一個行程以一個執行緒開始。如果需要，行程可以產生更多執行緒，使 CPU 可以在同一時間執行行程中不同區段的碼。標準的例子就是文書處理軟體。當文書處理軟體要列印，它把列印工作交給另一個執行緒，讓主執行緒依然能夠對使用者的動作有所回應。於是使用者就能夠在列印時繼續工作。

當然，如果你熟悉 CPU 基礎架構，你就會知道，一部只有一顆 CPU 的機器根本不可能同時執行兩件工作。「許多個執行緒同時執行」的幻覺是靠 VMM 排程器提供。它使用一個硬體計時器和一組複雜的規則，在不同的執行緒之間快速切換。

微軟宣告 Windows 95 的時間切片（timeslice）是 20 毫秒（milliseconts）。也就是說，如果不考慮其他因素（例如執行緒優先權），每一個執行緒執行 20 毫秒，然後換別人執行。我將在「Thread Priority（執行緒優先權）」一節中說得更詳細一點。不過我得先聲明，本書並不打算深入討論執行緒排程和 VMM 排程器。就像同步控制物件一樣，這些主題應該留待另一本書討論。

和行程一樣，執行緒是以一塊從 KERNEL32 共享記憶體中配置而來的記憶體表現出來。這塊記憶體持有所有必要的資訊，讓 KERNEL32 用來維護一個執行緒。雖然我說「所有必要的資訊」，實際上這塊記憶體中有一些指標指向其他資訊，不過你懂得我的意思就好。這塊記憶體在本書中被稱爲 thread database，或 TDB（注意，在不同的時間，微軟分別使用 TDB 代表 Task Database 和 Thread Database 兩種意義）。就像 process database 一樣，thread database 是一個 KERNEL32 物件，它的第一個 DWORD 值爲 6，表示這是一個 K32OBJ_THREAD 物件。

如果你是一位高級程式員，能夠改寫 DDK 或使用 WDEB386 或 SoftIce/W，你可能遭遇過另一個與執行緒有關的資料結構，名爲 THCB（Thread Control Block）。THCB 是

執行緒在 ring0 中的呈現。在 Windows 95，執行緒呈現 ring0 和 ring3 兩份資料結構。ring0 碼如 VMM VXD 者經由 THCB 來處理執行緒。ring3 碼如 KERNEL32 者則經由 thread database 來處理執行緒。本章描述 ring3 執行緒行爲和機制，並不打算涵蓋 ring0 那一端。

執行緒本身擁有一些東西。第一樣東西是暫存器組（register set）。一如稍早我說過，執行緒要不是正在執行，要不就是並未執行（這可不是廢話嗎）。當執行緒正在執行，它的暫存器組內容被放到 CPU 暫存器中，也就是說執行緒的 EIP 值就是暫存器 EIP 值。當執行緒不在執行狀態，它的暫存器必須儲存在記憶體某處。因此，每一個執行緒有一個指標指向一塊記憶體緩衝區，執行緒的暫存器內容就存放在那裡。

與每一執行緒都有關係的另一樣東西是行程。行程中的所有執行緒分享行程的每一樣東西，例如，行程擁有 memory context 和一個私有的位址空間，所以其下的所有執行緒都在相同的位址空間中執行。行程有一個 handle table，用來參考檔案、主控台（console）、記憶體映射檔（memory mapped files）、events 等等，行程中的所有執行緒也共享相同的這些 handles。如果 handle 3 代表一個記憶體映射檔，行程中的任何一個執行緒都可以使用 handle 3 來參考這個記憶體映射檔。

執行緒還擁有許多其他東西。每一個執行緒有一個專屬的堆疊，一個專屬的訊息佇列，一個專屬的 Thread Local Storage（TLS）以及一個專屬的結構化異常處理串鏈（如果你不知道後兩者是什麼，稍後我將描述它們）。此外，執行緒在執行過程中可能會索求、釋放同步控制物件的擁有權。在看過 thread database 之後，我會解釋所有這些東西。

## 什麼是 Thread Handle？什麼是 Thread ID？

本章稍早我曾說過 process handle 和 process ID 的不同。我的說明可以輕易地套到 thread handle 和 thread ID 身上 -- 只要把「行程」改爲「執行緒」就好。如果你不確定，請回頭去看「什麼是 process handle？什麼是 process ID？」那一節。

*GetThreadHandle* 傳回一個常數（微軟說是一個「虛擬 handle」），可以使用於任何真正的 thread handle 可以用的地方：

**GetThreadHandle 函式的虛擬碼**

```
x_LogSomeKernelFunction( function number for GetCurrentThread );

   return 0xFFFFFFFE;
```

就像 *GetCurrentProcessId* 那樣，*GetCurrentThreadId* 傳回一個指標，指標目前的 thread database（但 KERNEL32 小組會加上一個令人迷惑的數值）：

**GetCurrentThreadId 函式的虛擬碼**

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

KERNEL32 如何迷惑世人呢？我們看看：

**TDBToTID 函式的虛擬碼**

```
// 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;
```

如果你認爲這一段看起來真像先前提過的 *PDBToPID* 函式，你是對的。KERNEL32 使

用同一個 ObsfucatorDWORD 把 process database 指標和 thread database 指標轉換爲 IDs。一旦你了解 ObsfucatorDWORD 的值（並且記住微軟拼錯了這個字），你就可以把行程或執行緒的 ID 轉換爲有用的指標了。我要再說一次，這並不是被鼓勵的程式動作，但是爲了多瞭解系統的底層動作，我們沒有太多選擇。

# Thread Database

Thread Database 是一個 K32 物件（K32OBJ_THREAD），從 KERNEL32 共享資料區中配置而來。和 process database 一樣，thread database 也並不是直接成爲一個串列形式。Win32Wlk 的 THREADDB.H 檔中有 thread database 的 C 語言定義，格式如下：

#### 00h DWORD Type

此欄爲 6，表示 K32OBJ_THREAD 物件。

#### 04h DWORD cReference

此欄內含執行緒的參用次數。

#### 08h PPROCESS_DATABASE pProcess

這是一個指標，指向執行緒所屬的行程。

#### 0Ch DWORD someEvent

一個指標，指向 K32OBJ_EVENT 物件。Event 物件通常被交給 *WaitForSingleObject*。這個 Event 物件正是你呼叫 *WaitForSingleObject* 函式時給予的 event。

#### 10h DWORD pvExcept

這是一個指標，指向結構化異常處理的串鏈頭（結構化異常處理將在稍後討論）。請注意這個欄位也標記了 task database 中的 TIB（Thread Information Block）巢狀結構的起始處。TIB 亦將在稍後討論。

#### 14h　DWORD　TopOfStack

這個欄位放的是執行緒堆疊的最高位址。一般而言保留給執行緒的堆疊大小是 1MB。

#### 18h　DWORD　StackLow

這個欄位放的是執行緒堆疊的低水位標記（以 page 爲單位）。把 TopOfStack 減去 StackLow 就可以知道執行緒目前使用了多大的堆疊。

#### 1Ch　WORD　W16TDB

這裡存放的是 Win16 task database 的 selector。第７章會告訴你，每一個 Win32 行程都有一個 Win16 task database 和一個 Win32 process database。

#### 1Eh　WORD　StackSelector16

Win32 碼下移（thunk down）至 16 位元碼之前，必須先切換到一個 16 位元堆疊。這個欄位存放的即是該 16 位元堆疊的 selector。

#### 20h　DWORD　SelmanList

一個指標，指向執行緒的 SelmanList。"Selman" 意指 "Selector Manager"。KERNEL32 中的 Selman 似乎有責任管理 selectors，執行緒可以從中配置，作爲各種用途。

#### 24h　DWORD　UserPointer

此欄位的精確意義未明。然而 TIB 結構的文件中說，這個欄位可以給應用程式使用。別忘了，TIB 結構在 thread database 中是巢狀的。

#### 28h　PTIB　pTIB

這個欄位指向 TIB（Thread Information Block）。Windows 95 的 TIB 位在 thread database 之中，所以此一指標其實是指向 thread database 的另一個欄位：pvExcept 欄位（偏移位置 10h）。

### 2Ch WORD TIBFlags

這個欄位內含一些旗標值，給 TIB 使用。它們是：

| 旗標名稱和其數值 | 說明 |
|---|---|
| TIBF_WIN32（0x0001） | 此一執行緒來自 Win32 程式 |
| TIBF_TRAP （0x0002） | 某種異常處理（exception handling） |

### 2Eh WORD Win16MutexCount

這個欄位和 Win16Mutex（也被稱爲 Win16Lock）有點關係。通常此欄位在 Win32 執行緒中爲 -1，在 Win16 執行緒中爲 0。

### 30h DWORD DebugContext

如果執行緒所關係的行程正被除錯，此欄位將指向一個 debug context 結構。該結構之格式未知，但似乎有一些暫存器值放在其中。如果行程並非處在除錯狀態，此欄位爲 0。

### 34h PDWORD pCurrentPriority

此欄位指向一個 DWORD，內含執行緒的優先權。該 DWORD 位於 0xC0000000 之上，那結結實實正是 VxD 的勢力範圍。

### 38h DWORD MessageQueue

此欄位的低字組（low WORD）放置一個 Win16 global heap handle，用作執行緒的訊息佇列。訊息佇列是存放系統轉來的視窗訊息的場所，將在第 4 章介紹。這個欄位和 W16TDB 的 1Ch 欄位有密切關係。

### 3Ch DWORD pTLSArray

這個指標指向執行緒的 TLS 陣列。陣列中的項目被用於 *TlsSetValue* 函式家族。TLS 將在本章稍後描述。TLS 陣列所用的記憶體位於 thread database 之後。

### 40h　PPROCESS_DATABASE pProcess2

這個欄位容器指標，指向此一執行緒所屬之行程。它似乎總是和 08h 欄位重複。

### 44h　DWORD　Flags

此園地內含各式各樣的旗標值。下面是我所知道的一些旗標意義：

| 旗標名稱和其數值 | 說明 |
|---|---|
| fCreateThreadEvent　0x00000001 | 執行緒正被除錯中 |
| fCancelExceptionAbort　0x00000002 | |
| fOnTempStack　0x00000004 | |
| fGrowableStack　0x00000008 | |
| fDelaySingleStep　0x00000010 | |
| fOpenExeAsImmovableFile　0x00000020 | |
| fCreateSuspended　0x00000040 | 呼叫 *CreateProcess* 時指定了 CREATE_SUSPENDED |
| fStackOverflow　0x00000080 | |
| fNestedCleanAPCs　0x00000100 | APC = Asynchronous Procedure Call |
| fWasOemNowAnsi　0x00000200 | ANSI/OEM 檔案功能 |
| fOKToSetThreadOem　0x00000400 | ANSI/OEM 檔案功能 |

### 48h　DWORD　TerminationStatus

這個欄位將被 *GetExitCodeThread* 傳回。執行緒的退出碼可以經由 *ExitThread* 或 *TerminateThread* 指定。如果執行緒尚在執行，此欄位將爲 0x103（STILL_ACTIVE）。

### 4Ch　WORD　TIBSelector

此欄位十分重要，內含一個 selector，代表現行執行緒之 TIB（Thread Information Block）。TIB 內含極重要的資訊，像是此一執行緒的異常處理的串鏈頭（head of the exception handler chain）。當 Windows 95 切換執行緒，會更改 FS 暫存器，存放此值。如此一來執行緒才能夠經由 FS 暫存器獲得這些重要資料。

#### 4Eh WORD EmulatorSelector

這個欄位可能是一個 selector，代表一塊記憶體空間，其中存放 80387 模擬器狀態（可能是一個 FSAVE-style 的結構）。如果機器使用算術協同處理器（math coprocessor），此欄將爲 0。

#### 50h DWORD cHandles

這個欄位的意義未明，似乎總是 0。

#### 54h DWORD WaitNodeList

如果執行緒正在等待一個（以上）的 event 被激發，這個欄位就會指向 VxD 區域中的一個由 event 節點所組成的串列。每一個節點內含一個指標，指向一個 event 物件；另含一個指標，指向正在等待 event 的那個執行緒。

#### 58h DWORD un4

這個欄位的意義未明。通常它不是 0 就是 2。

#### 5Ch DWORD Ring0Thread

這個欄位內含指標，指向執行緒的 ring0 THCB（Thread Control Block）。

#### 60h PTDBX pTDBX

這個欄位內含指標，指向 TDBX 結構。TDBX 是執行緒在 VWIN32.VXD 中的呈現，第 6 章對此結構有比較詳細的描述。

#### 64h DWORD StackBase

對 Win32 執行緒，此值表示執行緒堆疊的最低可能位址。以此值減堆疊最大位址（本結構的 14h 欄位）你就可以計算出有多少位址空間保留給堆疊。對於 Win16 執行緒，此欄爲 0。

### 68h　DWORD　TerminationStack

根據其名，這個欄位應該內含執行緒結束程序中最初使用的 ESP 值。對 Win32 執行緒而言，此欄位和 TopOfStack（14h 欄位）相同。對於 Win16 執行緒，此欄位內含一個位址，恰在 shared KERNEL32 heap 之下。

### 6Ch　DWORD　EmulatorData

我推測此欄位是 80387 模擬器資料的 32 位元線性位址。如果確實如此，那麼此一欄位就和 EmulatorSelector 欄位（4Eh 欄位）有關聯。

### 70h　DWORD　GetLastErrorCode

此欄位內含 *GetLastError* 的傳回值。此值可以經由 *SetLastError* 設定。

### 74h　DWORD　DebuggerCB

如果執行緒是一個除錯器執行緒（也就是說它呼叫 *WaitForDebugEvent*），則此欄位內含指標，指向一塊被除錯器使用的資料，其中包括除錯器的 process database，thread database，以及被除錯者的 thread database。

### 78h　DWORD　DebuggerThread

如果執行緒正被除錯中，此欄位內含一個 non-NULL 值。其真正意義不得而知，因爲它的值太低了，不是一個合法的指標。

### 7Ch　PCONTEXT　ThreadContext

這個指標指向 Intel 的 CONTEXT 結構（定義於 WINNT.H）。該結構內放的是非處於執行狀態的執行緒的暫存器值。結構內容可以經由 *GetThreadContext* 函式和 *SetThreadContext* 函式讀寫之。

### 80h　DWORD　Except16List

此欄位意義未明。從名稱看它似乎應該和異常處理有點關係。我的觀察顯示，它總是 0。

### 84h DWORD ThunkConnect

意義未明。從名稱看它似乎應該和 thunking 動作有點關係。我的觀察顯示，它總是 0。

### 88h DWORD NegStackBase

如果你把這個欄位加上 StackBase 欄位（64h 欄位），總是得到 FFEF9000。不要問我爲什麼。

### 8Ch DWORD CurrentSS

這個欄位內含一個 16 位元堆疊 selector ，此與 32 位元碼下移（thunk down）至 16 位元碼有關。此欄位似乎非常近似 StackSelector16 欄位（1Eh 欄位），我不太清楚兩者之間的差異。

### 90h DWORD SSTable

這個指標指向一塊記憶體，內含的資訊與「32 位元碼下移（thunk down）至 16 位元碼」有關。

### 94h DWORD ThunkSS16

這個欄位又是內含一個與下移（thunk down）動作有關的資訊。在某些執行緒中，它和 StackSelector16 欄位（1Eh 欄位）吻合，而在另一些執行緒中，它又和 CurrentSS 欄位（8Ch 欄位）相同。

### 98h DWORD TLSArray[64]

這是 64 個 DWORDs 組成的一個陣列。每一個 DWORD 是「*TLSGetValue* 函式根據已知之 TLS ID 而傳回的值」。例如，第一個 DWORD 是 *TLSGetValue*(0) 的傳回值，第二個 DWORD 是 *TLSGetValue*(1) 的傳回值，依此類推。TLS 將於稍後描述。

### 198h　DWORD　DeltaPriority

這個欄位內含「此執行緒之優先權」與其「所屬行程之優先權等級」之間的差異。它可能是：

```
THREAD_PRIORITY_LOWEST          -2
THREAD_PRIORITY_BELOW_NORMAL    -1
THREAD_PRIORITY_NORMAL           0
THREAD_PRIORITY_ABOVE_NORMAL     1   （譯註：原書爲 2，是錯誤的）
THREAD_PRIORITY_HIGHEST          2   （譯註：原書爲 1，是錯誤的）
```

### 19Ch　DWORD　un5[7]

這個欄位的意義未明。它似乎總是 0。

### 1B8h　DWORD　pCreateData16

如果此欄位不是 0，它就是指向一個結構。該結構有兩個 32 位元指標：

```
00h     pProcessInfo    一個 PPROCESS_INFORMATION
04h     pStartupInfo    一個 PSTARTUPINFO
```

不過根據我的觀察，此欄位總是爲 0。

### 1BCh　DWORD　APISuspendCount

每當呼叫 *SuspendThread* 一次，這個欄位就累加 1；每當呼叫 *ResumeThread* 一次，這個欄位就累減 1。

### 1C0h　DWORD　un6

這個欄位的意義未明。

### 1C4h　DWORD　WOWChain

此欄位根據推測與 Windows 95 對於 WOW（Windows on Windows）的支援有關。WOW 是 Windows NT 在其保護位址空間中執行 16 位元程式的方法，可以阻止它們破壞 32 位元程式。根據我的觀察，此欄位總是爲 0。

### 1C8h WORD wSSBig

從其名稱來看，此欄位內含一個平滑模式（flat modal）的 32 位元 selector，當作是堆疊節區。然而根據我的觀察，此欄位總是爲 0。

### 1CAh WORD un7

這個欄位的意義未明。它可能只是用來填充空間，以使後續欄位得以 DWORD 爲邊界。

### 1CCh DWORD lp16SwitchRec

這個欄位的意義未明。從其名稱來看，它可能和 Win16 thunking 動作有關。

### 1D0h DWORD un8[6]

這五個欄位似乎總是 0。

### 1E8h DWORD pSomeCritSect1

這樣欄位指向一個 K32OBJ_CRITICAL_SECTION 物件。此一物件在每一個行程中都不一樣，它的目的我還不十分清楚。這個欄位似乎總是和 pSomeCritSect2 相同。

### 1ECh DWORD pWin16Mutex

這個指標指向 KRNL386.EXE 中的 Win16Mutex。

### 1F0h DWORD pWin32Mutex

這個指標指向 KERNEL32.DLL 中的 Win32Mutex。

### 1F4h DWORD pSomeCritSect2

這樣欄位指向一個 K32OBJ_CRITICAL_SECTION 物件。此一物件在每一個行程中都不一樣。這個欄位似乎總是和 pSomeCritSect1 相同。

**1F8h　DWORD　un9**

這個欄位的意義未明。似乎總是 0。

**1FCh　DWORD　ripString**

從其名稱來看，這個欄位應該是一個字串指標，該字串將在 FatalAppExit RIP 中使用。然而，大部份情況下此欄位爲 0；如果不是 0，根據我的觀察，它卻又不指向一個字串。

**200h　DWORD　LastTlsSetValueEIP[64]**

64 個 DWORDs 所組成的這個陣列，平行於 TLS 陣列（thread database 的 98h 偏移處）。陣列中的每一筆項目和一個 TLS 索引值有關，含有 EIP 值。EIP 值是從堆疊中取出（由 *TlsSetValue* 設定）。

最後我要再補充一點：獲得 thread database 指標的方法不只一種。除了前面我提過的 XOR 把戲外，每一個 Win16 task database 也內含一個指標指向對應的 thread database。Task database 的 54h 欄位放置有一個線性位址，代表第一個執行緒的 thread database。

# Thread Information Block（TIB）

在 thread database 之中，有一些欄位對於執行中的程式極爲有用。事實上，它們是如此有用，以至於 Win32 架構讓它們可以立刻被取用，不需經過 thread database。這些欄位被放置到一個名爲 Thread Information Block（TIB）的結構中。Thread database 的 10h~3Ch 欄位統統被放到 TIB 之中。

應用程式如何取用 TIB？如果你看過 Win32 程式的組合語言碼，你會發現 FS 暫存器有頻繁的使用率。等等，Win32 不是會搬移節區嗎？雖然答案是 Yes，但是 Win32 底層（包括 Windows NT、Windows 95、Win32s）都奉獻出 FS 暫存器，用以指向目前執行中的執行緒的 TIB。Win32 並不是第一個這麼做的作業系統，OS/2 2.0 早就如此了。就像你所感覺到的，是的，當 Windows 95 切換執行緒，排程器必須更改 FS 暫存器值，讓它內含一個 selector，指向新的 TIB。

FS 暫存器和 TIB 的主要用途就是增加結構化異常處理串鏈（structured exception handling chain）的項目個數。串鏈的頭放在 TIB 的 0 偏移處，所以當你看到組合語言碼使用 FS:[0]，你就知道它正在做某些與結構化異常處理有關的動作了。

TIB 的另兩個欄位也十分廣泛地被使用，它們是 pvQueue 和 pvTLSArray （分別是 28h 和 2Ch 欄位）。pvQueue 欄位內放的是現行執行緒的訊息佇列的 handle，此欄位常常被 USER.EXE 的視窗系統使用。因爲在 Windows 95 之中，「焦點視窗（focus window）」並非整個系統只一個。pvTLSArray 則指向 thread database 中的 TLS 陣列。編譯器廠商把它和可執行檔的 .tls section 聯用，提供透明化的所謂 ”per-thread global variable”。

雖然 TIB 結構的佈局可以從 thread database 中推算而來，我要在這裡提供一點摘要。Win32Wlk 的 TIB.H 檔中有 TIB 的 C 語言結構定義。微軟對此的第一份文件是 Windows NT 3.5 DDK 提供的 NTDDK.H（附帶一份嚴厲警告，警告這些欄位必須與 OS/2 2.0 相容）。這明顯是早期 NT 的一個遺跡，那時候的微軟還嘗試給大家一個印象說它十分在乎 OS/2（請看 Z.Pascal Zachary 的書 ***Showstopper***，裡面有一些與這個主題相關的有趣故事）。

Windows 95 的 TIB 內容如下：

```
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 DWORD   pTLSArray
```

如果想知道每一個欄位的意義，把其偏移值加 10h，然後看看上一節的結構內容，便知道了。請注意，其中只有一些欄位對於其他 Win32 平台是共通的。

# Thread Priority（執行緒優先權）

Windows 95 的 VMM 核心排程系統並不真正在乎行程，它只在乎執行緒的優先權，而不管執行緒屬於哪一個行程。換句話說，行程也不真正擁有優先權。當然啦，對於執行緒排程服務的終端使用者（也就是應用程式）而言，「行程有優先權」是一種比較有用的抽象想法。

任何時候，擁有最高優先權的執行緒，而且它又沒有什麼東西要等待的話，將會是即將執行的一個。爲了確保平順地讓系統運轉並避免許多問題，執行緒優先權可以動態地被系統改變。例如，當一個執行緒正在進行 I/O 動作，它的優先權可以暫時性地提高。更詳細的談論這個題目，會耗掉大半章。因此，我將把對執行緒優先權的討論放在另一本書籍中（或是未來的雜誌文章中）。

Windows 95 的 VMM 排程器支援 32 種優先權。這 32 個優先權被區分爲四個等級，稱爲優先權等級（priority classes），每一個等級關係到一組優先權。在一個等級之中，優先權可以上下增減 2。但是也有特殊情況如 THREAD_PRIORITY_LEVEL，它可以使優先權完全跳出其等級的規範之外。除非有特別指定，否則作業系統產生一個行程時給予的優先權等級是 NORMAL_PRIORITY_CLASS。

四個優先權等級的預設優先權以及上下擺盪範圍是：

| 優先權 | 預設值 | 上下擺盪範圍 |
|---|---|---|
| IDLE_PRIORITY_CLASS | 4 | 2~6 |
| NORMAL_PRIORITY_CLASS | 9 或7（前景爲 9，背景爲 7） | 1~10 |
| HIGH_PRIORITY_CLASS | 13 | 11~15 |
| REALTIME_PRIORITY_CLASS | 24 | 16~31 |

執行緒優先權爲 1 是一種特殊情況。凡是 IDLE_PRIORITY_CLASS、NORMAL_PRIORITY_CLASS 或 HIGH_PRIORITY_CLASS 三個等級，都可以經由

*SetPriorityClass* 函式設定優先權爲 1。

注意，Windows 95 排程器中的這 32 個優先權，其數值並非對應於 WINBASE.H 的定義。例如，NORMAL_PRIORITY_CLASS 在 WINBASE.H 中定義爲 0x20。KERNEL32.DLL 將這些值映射爲適當的 Windows 95 排程器優先權。

## GetThreadPriority

*GetThreadPriority* 是一個簡單的函式。給予一個 thread handle（可以是行程中的任何執行緒），這個函式會把 handle 轉換爲指標，指向 thread database。如果轉換沒有問題，函式就傳回 thread database 中的 DeltaPriority 欄位（198h 欄位）。所有的碼都被一個 EnterSysLevel 和一個 LeaveSysLevel 包裹起來，以避免不適當的執行緒切換。

**GetThreadPriority 函式的虛擬碼**

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

   x_LogSomeKernelFunction( function number for GetThreadPriority );

   _EnterSysLevel( pKrn32Mutex );

   retValue = 0x7FFFFFFF;

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

   if ( ptdb )
      retValue = ptdb->DeltaPriority;

   _LeaveSysLevel( pKrn32Mutex );
```

## SetThreadPriority

*SetThreadPriority* 的主體被切割爲四個部份。首先，把 thread handle 轉換爲一個 thread database 指標。然後，檢查傳進來的優先權（參數之一），看看是否落在合理範圍內。然後，再以內部函式 *CalculateNewPriority* 將傳進來的優先權轉換爲 Windows 95 排程器認識的優先權值。我將在下一節探討 *CalculateNewPriority* 函式。

最後，*SetThreadPriority* 呼叫 VWIN32.VXD，將新的優先權通知 ring0 元件。KERNEL32 藉由 *VxDCall* 函式呼叫 ring0 碼。Ring3 碼就是使用 *VxDCall* 函式才能夠喚起 Win32 VxD 的服務。在這個例子中，VWIN32.VXD 提供了一個 ring3 可呼叫的服務函式（service），用以設定優先權。Win32 VxD service 是 Windows 95 的新特質，提供 ring0 和 ring3 之間的互動。事實上，由於這項特質是如此重要，第 6 章有一大部份用在這方面的描述上。由於 Win32 VxD 的服務將在稍後涵蓋，所以這裡我並不專注於 VxD 的真正機制。

**SetThreadPriority 函式的虛擬碼**

```
// 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

這個函式封裝了 Windows 95 排程器的執行緒優先權計算規則。給予一個行程和一個執行緒，它就可以計算執行緒應該有的優先權（範圍 1~31）。從 process database 中，這個函式取出執行緒的優先權等級（normal、idle、high 或 realtime 四種）。面對優先權基底，此函式再補上差額。優先權差額在 +-2 之間。補上差額之後，此函式確定新值也是一個合法的優先權。擁有 realtime 優先權者不值得在此做特別處理，因爲 realtime 優先權等級的範圍比一般等級大得多。

### CalculateNewPriority 函式的虛擬碼

```
// 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

呼叫此函式可以改變執行緒的優先權等級。它首先把 hProcess 轉換爲一個 PPROCES_DATABASE 指標，然後根據這個指標決定行程的目前的優先權等級。如果和原等級相同，就不需要再做任何事。

如果新等級和舊等級不同，*SetPriorityClass* 會把新等級的預設優先權值寫入 process database 的 BasePriority 欄位中。但是，等等，還有咧，稍早我曾說過所謂行程的優先權只是一種概念，因爲 VMM 排程器只關心執行緒，而不是行程。爲了在兩個視野之間搭起橋樑，*SetPriorityClass* 一一找到行程所轄的每一個執行緒，然後呼叫 VMM32.VXD 爲每一個執行緒設定新的優先權。

然而有一點需要注意。執行緒的優先權可以與其優先權等級之標準基值之間有著輕微差額。這個差額放在 thread database 的 DeltaPriority 欄位中。*SetPriorityClass* 必須取得每一個執行緒的優先權差額 -- 當它計算新優先權的時候。*CalculateNewPriority*（前面剛說過）負責這樣的計算。

**SetPriorityClass 函式的底擬碼**

```
// 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

此函式針對某個行程，傳回其優先權等級。在把 hProcess 轉換爲一個 PPROCESS_DATABASE 之後，它從 process database 中取出其優先權等級。優先權的值應該是在 1~31 之間，那和 WINBASE.H 所定義的 xxx_PRIORITY_CLASS 並不相同。因此，*GetPriorityClass* 必須把 VMM 排程器的優先權轉換爲對應的 xxx_PRIORITY_CLASS 旗標。

**GetPriorityClass 函式的虛擬碼**

```
// 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;
```

# 執行緒控制函式

Win32 API 提供小量函式，用來修改和詢問執行緒的執行狀態。低階方面，執行緒可以讀寫另一個執行緒的暫存器（假設它有其他執行緒的 handle 的話）。而在高階方面，有些 Win32 API 允許你冰凍或融解其他執行緒的執行。讓我們看看這些執行緒控制函式。

## GetThreadContext 和 IGetThreadContext

*GetThreadContext* 使執行緒有能力獲得另一個執行緒的暫存器的一份副本。任何時候，執行緒要不是正在執行，要不就是虛懸。即使執行緒處在虛懸狀態，其暫存器值還是保留在一個名爲 thread context 的結構中。*GetThreadContext* 函式讓你能夠讀取一個被虛懸的執行緒的 thread context。

*GetThreadContext* 函式其實只是做參數檢驗的工作。它檢查傳來的指標是否指向足夠大小以容納 CONTEXT 結構。如果答案是肯定的，程式碼就跳到內部函式 *IGetThreadContext* 去。

*IGetThreadContext* 首先轉換 hThread 參數爲一個 thread database 指標，然後呼叫 *x_ThreadContext_CopyRegs*（稍後描述），把輸入的暫存器組放到 ring3 的 CONTEXT 結

構中。除了拷貝暫存器內容到 ring3 CONTEXT 結構，*IGetThreadContext* 也呼叫 VWIN32.VXD 取得那些暫存器的 ring0 版本。至於暫存器爲什麼要有 ring0 和 ring3 版，不甚清楚。

在將 CONTEXT 結構填充完畢之後，*GetThreadContext* 檢查 CS 以及旗標暫存器的值，看看是否合法。旗標暫存器的檢查很簡單，只是確定目前不處於 V86 模式。

**GetThreadContext 函式的虛擬碼**

```
// 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 函式的虛擬碼**

```
// 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

這個函式把被選中的暫存器內容，從某個 CONTEXT 結構拷貝到另一個 CONTEXT 結構。CONTEXT 定義於 WINNT.H，是針對 x86 晶片而定義；其第一個欄位是個 DWORD 旗標，用來指出 CONTEXT 中的哪些暫存器應該被拷貝。

爲 x86 晶片而設定的暫存器群組有：

```
CONTEXT_DEBUG_REGISTERS     除錯暫存器 0~3,6,7
CONTEXT_FLOATING_POINT      算術協同處理器的狀態
CONTEXT_SEGMENTS       DS、ES、FS、GS
CONTEXT_INTEGER             EAX、EBX、ECX、EDX、ESI、EDI
CONTEXT_CONTROL             SS:ESP、CS:EIP、EFLAGS、EBP
```

*x_ThreadContext_CopyRegs* 函式十分直接了當。它忠實地檢查來源端的每一個旗標，如果設立，就拷貝對應的暫存器值到目的端的 CONTEXT 中。程式碼中沒有什麼令人震驚的發現。

**x_ThreadContext_CopyRegs 函式的虛擬碼**

```
// 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* 使執行緒有能力改變另一個執行緒的暫存器。任何時候，這個函式與 *GetThreadContext* 形成互補。視輸入的 CONTEXT 結構中有哪些旗標被設立，*SetThreadContext* 函式於是拷貝其中的某些欄位，放到「Windows 95 用來保持被虛懸執行緒的暫存器」的 CONTEXT 中。

*SetThreadContext* 函式其實只是做參數檢驗的工作。它檢查傳來的指標是否指向足夠大小以容納 CONTEXT 結構。如果答案肯定，程式碼就跳到內部函式 *ISetThreadContext* 去。

*ISetThreadContext* 首先轉換 hThread 參數爲一個 thread database 指標，然後，視外部條件而定，它呼叫 *x_ThradContext_CopyRegs* 以及 VWIN32.VXD。再一次我要說，暫存器爲什麼要有 ring0 和 ring3 版，不甚清楚。

**SetThreadContext 函式的虛擬碼**

```
// 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 函式的虛擬碼**

```
// 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* 會影響執行緒的虛懸次數。如果虛懸次數不是 0，ring0 排程器絕不可能讓該執行緒執行。*SuspendThread* 函式碼其實只不過是我所謂的 *VWIN32_SuspendThread* 內部函式的外包裝而已。*VWIN32_SuspendThread* 預待獲得一個 thread database 指標，所以在呼叫它之前，*SuspendThread* 首先把 hThread 轉換爲有用的指標。如果 *VWIN32_SuspendThread* 成功，*SuspendThread* 會在 thread database 的 1BCh 欄位增加虛懸次數。

然而，重要的是，當排程器欲決定哪一個執行緒可以執行，虛懸次數並不被考慮爲一個決定因素。倒是，VWIN32 在 TDBX 結構中保持了真正的虛懸次數（請見第 6 章）。*VWIN32_SuspendThread* 其實也只不過是低階函式的一個外包裝而已，它經由一個未公開的 *VxDCall* 函式呼叫 VWIN32.VXD，交過去的 VxD service ID 是 0x002A001A。

**SuspendThread 函式的虛擬碼**

```
// 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 函式的虛擬碼**

```
// 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

這個函式和 *SuspendThread* 有互補作用。它會將某個執行緒的虛懸次數減 1（不論是在 ring0 的 TDBX 結構中，或是在 ring3 的 thread database 1BCh 欄位中）。當虛懸次數降爲 0，它就有資格被排程器視爲可能的執行對象。

*ResumeThread* 首先把 hThread 參數轉換爲一個 thread database 指標，然後檢查 ring3 的虛懸次數，確定它不是 0（如果是 0，*ResumeThread* 就什麼也別做了）。接下來，呼叫一個 VWIN32.VXD 服務函式，將虛懸次數減 1。這個服務函式的 ID 是 0x002A001B（比前面提過用於 *SuspendThread* 的那個服務函式的 ID 多 1）。如果這個 VWIN32.VXD 服務函式能夠成功地減低 ring0 的虛懸次數，*ResumeThread* 就減低 ring3 的虛懸次數。

**ResumeThread 函式的虛擬碼**

```
// 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;
```

## 結構化異常處理（Structured Exception Handling）

結構化異常處理（Structured Exception Handling，簡寫爲 SEH）在現代化作業系統如 OS/2、Windows NT、Windows 95 中，是一個被大肆宣傳並且常常被誤解的主題。大部份談到它的書籍和文章都把它放在編譯器層面來說。編譯器利用一些如 __try、__except、catch、throw 等的保留字，把零亂的作業系統基礎介面包裝起來。截至目前，我還沒有看過一篇能夠解釋如何在作業系統層面實作出 SEH 的好文章。所以我試著做點救濟。

已有許多書籍和文章描述如何在應用程式中使用 SEH，所以我不打算重彈老調。第２章曾經大略告訴你如何在 C/C++ 碼中使用 __try 和 __except。這一份討論將假設你熟悉 SEH 的基礎。如果你不是，我建議你讀 Jeffrey Richter 的 ***Advanced Windows*** 或 Brian Meyer 的 ***Mastering Windows NT Programming***。

當一個異常情況（例如 page fault）發生，CPU 會立刻把控制權轉給 ring0 的異常處理函式，後者位址存放在中斷描述表格（interrupt descriptor table）中。ring0 處理函式才能夠決定該如何善後。如果這是一個系統知道怎麼處理的異常情況，ring0 碼就做必要的措施，然後讓指令繼續下去。這些異常情況基本上對 ring3 碼以及 system DLLs 是不可見的，而且此處我們也不關心它們。

此處我們所關心的是，萬一系統不知道如何處理這個異常情況，怎麼辦？老舊的作業系統的典型反應就是把引起異常情況的行程砍掉。這也就是爲什麼你會看到有臭蟲的程式引發一個 UAE 對話盒（在 Windows 3.0 中）或一個 GPF 對話盒（在 Windows 3.1 中）然後壯烈身亡的畫面。

> 譯註：UAE 是 Unrecoverable Application Error 的縮寫。GPF 是 General Protection Fault 的縮寫。兩者都是程式員的最怕 ☺。

雖然「將任何引起不可預期之錯誤的程式結束掉」的哲學無可挑剔，但它畢竟沒有包容性。比較好的作法是通知應用程式（或其他應用程式）讓它們自己決定怎麼辦。如果你使用 Win16 TOOLHELP 的 *InterruptRegister* 函式，你就會看到這種策略的實現。應用

程式可以經由它向系統登記一個 callback 函式，用以處理中斷和異常情況（但是每一個行程只能有一個處理函式）。當預期的中斷或異常情況發生，TOOLHELP 就會呼叫登記過的 callback 函式，並根據其傳回值決定如何處理這個異常。WinSpector 或 Dr. Watson 這類程式就是利用這種方式，把異常情況下的機器狀態記錄下來。然後，它們再告訴 TOOLHELP 把異常情況交給下一個處理函式。假設沒有任何一個 TOOLHELP callback 函式將引發異常的指令重新執行起來，系統預設的異常處理函式就會被喚起，砍掉罪魁禍首。

雖然 TOOLHELP 的 callback 體制算是向前邁進一大步（比起完全沒有控制好多了），它仍然不夠好。32 位元 PC 作業系統如 OS/2 和 NT 引入一個比較更有彈性的處理方法。新的方法就是我們所說的結構化異常處理（SEH）。在處理多執行緒以及利用 C++ catch/throw 機制等方面，SEH 表現比以前的方法好得多。對於 C++ 異常情況，應用程式本身可以引發一個和 CPU 異常情況截然不同的異常情況。假設 C++ 的 new 運算子失敗，它會丟出一個代表記憶體不足的異常情況。32 位元作業系統的 SEH 機制有足夠彈性，以相同的碼一併處理語言的異常和硬體的異常。

在繼續進行之前，我要再次強調我要談的 SEH 是它們在作業系統層面的實作情況。下面的敘述可能和你在 C/C++ 課程中所受的訓練完全不同。

在一個擁有 SEH 的系統中，每一個執行緒有它自己的私有串列，內含安裝好異常處理函式。當一個異常情況發生，作業系統走訪該串列，並呼叫其中的適當函式。這樣的動作一直持續到某個函式傳回代碼，表示它要處理這個異常情況為止。這就是第一階段：找到正主兒。如果這些函式無一可以處理此一異常情況，系統就會出面，把鬧事兒的行程砍掉。我們不關心最後這種情節，因為作業系統把行程砍掉是很簡單的事情。

當你獲得了一個函式用來處理異常情況，第二階段就是重新再走訪串列一遍。未公開函式 *RtlUnwind* 為我們做這檔事，它被那個「決定處理此一異常情況的處理函式」所呼叫。當 *RtlUnwind* 一一觸發那些串列中的函式，系統會交給它們一個旗標。這個旗標告訴函式說執行緒的堆疊目前正被 "unwound"（譯註）。將堆疊 "Unwinding"，是「把程式狀

態恢復到異常處理函式被安裝時的狀態」的一種方法。不只在 __except 區塊中重新恢復執行，系統還給予每一個「被安裝，但是不處理此一異常情況」的函式一個機會清理自己。給予這個機會之後，重要的事情如「呼叫堆疊中的 C++ 物件的解構式」就可以在一種有紀律的情況下完成。

> 譯註：所謂 "unwind"，就是當一個異常情況發生時，將堆疊中的物件解構。這是 ANSI 的標準要求。據我所知，OWL 支援完整的 "stack unwinding"，而早期的 MFC 並不包含自動的 stack unwinding。新版 MFC 表現如何我就不清楚了。

理論夠多了，Windows 95 真正使用的結構和介面到底是什麼呢？先前我介紹 TIB 時曾說過，FS:[0] 總是指向現行執行緒之異常處理函式的串列頭，異常處理函式所組成的串列是一個 EXCEPTIONREGISTRATIONRECORD 串列。這個長長的名稱來自於 OS/2 2.0 的 BSEXCEPT.H 檔。爲了某些理由，微軟似乎企圖對一般人隱藏 SEH 在作業系統層面的資訊。EXCEPTIONREGISTRATIONRECORD 結構看起來像這樣：

```
DWORD prev_structure   // A pointer to the previously installed
                       // EXCEPTIONREGISTRATIONRECORD
DWORD ExceptionHandler // Address of the exception handler function.
```

串列最後是以 -1（prev_structure）表示結束。

正常情況下，程式會依需要從堆疊中挖出空間來製造 EXCEPTIONREGISTRATIONRECORD。在 C/C++ 程式中，每一個 EXCEPTIONREGISTRATIONRECORD 對應一個 __try/__except 區塊。當程式進入 __try 區塊，編譯器就在堆疊中產生一個新的 EXCEPTIONREGISTRATIONRECORD 並把它放到串列起頭處。離開 __except 區塊之後，編譯器設定 FS:[0] 指向串列中的下一個 EXCEPTIONREGISTRATIONRECORD。**圖 3-3** 顯示這些被串鏈起來的資料。

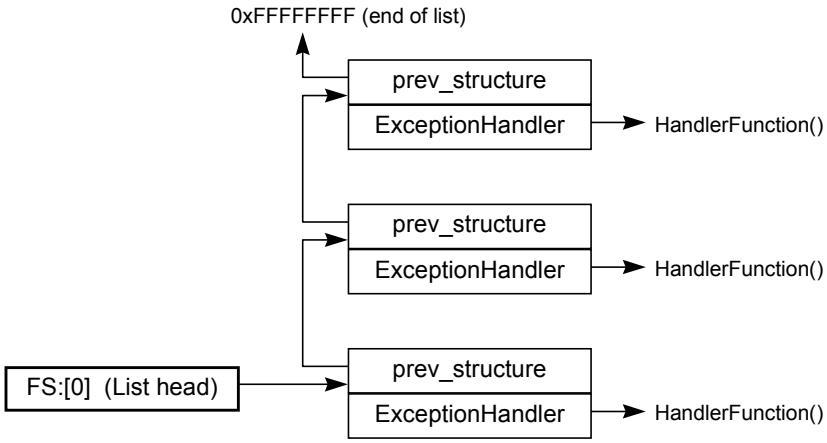



**圖 3-3　從作業系統層面看 SEH 串列。**

記住，上述 8 位元組的結構只是作業系統的最小需求而已。沒有什麼可以阻止編譯器在堆疊之中產生更大的結構並且把 EXCEPTIONREGISTRATIONRECORD 放在結構起首。編譯器從上述結構中獲得的其他欄位可以提供足夠的資訊，使單獨一個異常處理函式適用於所有的 __try 區塊。微軟公司和 Borland 公司的編譯器都使用 EXCEPTIONREGISTRATIONRECORD 的擴充結構。

說到異常處理函式，到底它長得什麼樣子？再一次，微軟似乎企圖隱藏某些資訊，但至少 Win32 表頭檔提供了一個函式原型。在 EXCPT.H 檔中，你可以看到這樣的原型：

```
EXCEPTION_DISPOSITION __cdecl _except_handler (
   struct _EXCEPTION_RECORD *ExceptionRecord,
   void * EstablisherFrame,
   struct _CONTEXT *ContextRecord,
   void * DispatcherContext
   );
```

乍見之下它彷彿太過複雜了。傳回值 EXCEPTION_DISPOSITION 其實只不過是個 enum，告訴系統說此一函式如何被用來處理異常情況：

```
typedef enum _EXCEPTION_DISPOSITION {
    ExceptionContinueExecution,
    ExceptionContinueSearch,
    ExceptionNestedException,
    ExceptionCollidedUnwind
} EXCEPTION_DISPOSITION;
```

最後兩項很少會遇到。至於第一項 ExceptionContinueExecution 是告訴系統說異常處理函式已經處理了該異常情況，並打算讓執行繼續下去。ExceptionContinueSearch 則是告訴系統說異常處理常式不打算處理這個異常情況，系統應該繼續走訪 EXCEPTIONREGISTRATIONRECORD 串列，直到某個處理函式傳回 ExceptionContinueExecution。

把 *_except_handler* 函式原型重寫一遍，看起來就可接受多了：

```
int _except_handler (
    PEXCEPTION_RECORD ExceptionRecord,
    PVOID EstablisherFrame,
    PCONTEXT ContextRecord,
    PVOID DispatcherContext );
```

我們發現，一個異常處理函式需要四個指標參數，指向異常情況以及機器狀態等資訊。這個函式傳回一個整數，告訴系統它是否把異常處理好了。EXCEPTION_RECORD 結構內含異常代碼，以及其他東西。WINNT.H 對此有些說明。CONTEXT 結構內含異常發生時的暫存器內容，WINNT.H 對此也有說明。EstablisherFrame 參數內含一個指標，指向堆疊 -- EXCEPTIONREGISTRATIONRECORD 結構的設定處。DispatcherContext 參數則似乎沒有用到。

稍早我曾說過，處理函式會被呼叫兩次。第一次是系統尋找適當處理者的時候。第二次是爲了系統要 "unwinding"，而處理函式被認爲會進行任何必要的清理工作（像是呼叫堆疊物件的解構式等等）。這兩次啓動之間的差別在哪裡？ExceptionRecord 結構（被第一個參數所指）內含一個 ExceptionFlags 旗標，如果 EH_UNWINDING（0x2）或 EH_EXIT_UNWIND（0x4）旗標並未設立，那麼就是前述第一種情況；如果兩個旗標之中有一個設立，那麼就是前述第二種情況。

雖然我所說的不足以讓你寫一個自己的作業系統層面的異常情況處置碼，但是因爲足夠讓你了解 SEH 是如何運作的了。爲了證明我不是在胡亂吹大氣，我寫了一個 SHOWSEH 程式，放在書附磁片之中。SHOWSEH 利用 __try 段落來設立異常處理串鏈。統統設立好之後，這個程式即走訪 SEH 串列並印出每一個節點的內容。

SHOWSEH 的輸出結果請看圖 **3-4**。我要你注意數點。第一，請注意 "next rec" 欄位總是遞增，那是編譯器用來放置 EXCEPTIONREGISTRATIONRECORD 的堆疊區域。最前面四筆資料反應出 SHOWSEH.C 中的 __try 段落。第二，請注意前四筆資料有一個不變的 ESP 值。輸出畫面的最後一行則顯示 SHOWSEH.C 之中每一個函式的 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 = 0063FD84
in    b(), ESP = 0063FD78
in    a(), ESP = 0063FDA8
in main(), ESP = 0063FDD8
```

**圖 3-4　SHOWSEH.EXE 的輸出結果**

最後一件值得注意的事情是，最前面五筆資料的 handler 欄位都一樣，位址落於 SHOWSEH 的程式區域（而非資料區域）中，顯示出編譯器所產生的碼對於每一個 __try 段落都使用相同的異常處理函式。剛才我說過了，前四筆是 SHOWSEH.C 的四個 __try 形成的。至於第 5 個則是在呼叫 *main* 之前由執行時期函式庫安裝的。這些處理常式的位址統統都是 *__except_handler3* -- Visual C++ 的執行時期函式庫中的一個函式。最後一個異常處理函式是預設的系統處理函式，位於 KERNEL32.DLL 中。

## 結構化異常處理與參數確認

Windows 95 SEH 的一個最重要的用途就是提供快速並且容易的方法，檢驗 API 函式的參數。基本觀念是：假設參數是正確的，然後執行一系列測試。然而在執行這些測試之前，程式碼首先得在 SEH 串列頭部加上一個新的異常處理函式。如果參數是合法的，就不會有什麼壞事情發生，而新添的那個異常處理函式也可以拿掉。這些只需要花費極小的執行時間。

如果參數是不正確的，就會引起 CPU 異常情況，於是我們新安裝的異常處理函式就會挺身處理。這個函式告訴作業系統，繼續進行執行緒的執行，即使是在困難的處境下（可能引發 API 函式的失敗）。讓我們看看一些 Win32 API 函式的虛擬碼。本書對其他 API 的描述中，我對於 SEH 在參數檢驗的角色上都只是輕描淡寫。對於 *GetCurrentDirectoryA*，我將展示其虛擬碼，以顯示作業系統如何使用 SEH。

除了顯示 SEH 的觀念，*GetCurrentDirectoryA* 函式虛擬碼也詳細示範了什麼是一個典型的參數確認層。擁有參數確認層的函式被分割爲兩部份。輸出（exported）位址上的那一部份只是用作參數檢驗的一小段碼而已。如果參數正確，那一小段碼就會跳到真正的大角色去（位在同一模組中）。在 Windows 3.1 中，參數檢驗層之外的那個真正的函式本體，其函式名稱前面多一個 'I'。例如，本章稍早談到 *GetProcAddress* 時曾經看到，它被分爲兩部份：

```
GetProcessAddress stub
       Validate the procedure name string parameter
       JMP IGetProcessAddress

IGetProcessAddress
       Meat of the code that looks up a function address
```

爲了讓事情比較一致化，我也採用 Windows 3.1 的原則（函式前面加個 'I'）。

## GetCurrentDirectoryA

*GetCurrentDirectoryA* 就是個典型的參數檢驗層。它一開始在堆疊之中產生一個 EXCEPTIONREGISTRATIONRECORD（利用 PUSH 指令）。這個結構中的 prev_structure 成員內容，經由 push FS:[0] 被放到堆疊之中。EXCEPTIONREGISTRATIONRECORD 指定的異常處理函式是一個我命名爲 *x_invalid_param_2_handler* 的函式。函式如果有兩個參數，並且有一個參數檢驗層，就會在其檢驗過程中使用這個 *x_invalid_param_2_handler* 函式。

把 EXCEPTIONREGISTRATIONRECORD 壓進堆疊之後，程式碼讓一個指標指向 FS:[0] 所指的結構。這使得剛剛說的新的 EXCEPTIONREGISTRATIONRECORD 成爲異常情況處理函式串鏈中的第一個。就定位之後，函式碼就可以安全接觸 lpszCurDir 參數而不必管它是否合法了。這個例子中的參數檢驗動作包括 lpszCurDir 所指的記憶體是否可用，是否可寫，長度是否足夠等等。

一旦參數檢驗過程中發生異常情況，*x_invalid_param_2_handler* 就會獲得控制權。這個函式將於下一節描述。如果參數是正確的，執行路徑不會偏離。*GetCurrentDirectoryA* 最後會把異常處理函式移走。POP FS:[0] 會恢復異常處理函式串列的原始頭部，ADD EPS,4 則會抹去異常處理函式的位址。

假設每一件事情都做對了（也就是沒有異常情況發生），*GetCurrentDirectoryA* 的最後一個動作是跳到 *IGetCurrentDirectoryA* 去。令人驚訝的是，KERNEL32 似乎並不使用儲存在 environment database 中的 current directory pointer，它進入 Win16 task database 並使用 VWIN32 的 int 21h 分派機制，呼叫 INT 21h 的第 19h 號功能（Get Current Directory）和第 7147 號功能（用於處理長檔名）。

**GetCurrentDirectoryA 函式的虛擬碼**

```
// 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

這個函式是「參數不合法」這種異常情況的典型終站。這個函式並不是被直接呼叫。KERNEL32 中有 10 個小程式，每一個小程式交給 *x_invalid_param_handler* 不同的參數。擁有 10 個小程式的原因是，*x_invalid_param_handler* 必須負責移除那個「擁有不合法參數的函式」在堆疊中的所有參數。

*x_invalid_param_handler* 函式執行三個主要動作。第一是在除錯視窗中印出"Invalid parameter passed to: XXXXXXXX" 訊息。第二，函式呼叫 *RtlUnwind*，負責將所有「在參數檢驗碼安裝之後才安裝的異常處理函式」清理好。第三，也是最重要的一點，呼叫 *ReturnFailureCode* 函式（這是我的命名）。後者計算正確的失敗代碼，準備交給原函式（例如 *GetCurrentDirectoryA*），然後跳到原函式的退出程序（exit prologue）中。

所有這些背後的故事都只是使得一個獲得不合法參數的 Win32 函式執行失敗而已。如果你在 Windows 95 除錯版下，你將可以獲得不合法參數的診斷。

**x_invalid_param_X_param 函式的虛擬碼**

```
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 函式的虛擬碼

```
// 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;
```

# Thread Local Storage（執行緒區域儲存空間）

TLS 是一個良好的 Win32 特質，讓多執行緒程式設計更容易一些。TLS 是一個機制，經由它，程式可以擁有全域變數，但處於「每一執行緒各不相同」的狀態。也就是說，行程中的所有執行緒都可以擁有全域變數，但這些變數其實是特定對某個執行緒才有意義。例如，你可能有一個多執行緒程式，每一個執行緒都對不同的檔案寫檔（也因此它們使用不同的檔案 handle）。這種情況下，把每一個執行緒所使用的檔案 handle 儲存在 TLS 中，將會十分方便。當執行緒需要知道所使用的 handle，它可以從 TLS 獲得。重點在於：執行緒用來取得檔案 handle 的那一段碼在任何情況下都是相同的，而從 TLS 中取出的檔案 handle 卻各不相同。非常靈巧，不是嗎？有全域變數的便利，卻又分屬各執行緒。

當然，你可以使用串列，讓一個檔案 handle 與一個 thread ID 產生關係，每一個執行緒有一個節點。用此來模擬 TLS。當執行緒需要知道它使用哪一個檔案 handle，它可以從串列中尋找檔案 handle。你當然可以把檔案 handle 儲存在區域變數中（位於執行緒的堆疊）。但是卻因此必須把這個 handle 在函式與函式之間傳來傳去。那多痛苦。TLS 可以利用簡單的 alloc/set/get/free 函式消除這些問題。

雖然 TLS 很方便，它並不是毫無限制。在 Windows NT 和 Windows 95 之中，有 64 個 DWORD slots 供每一個執行緒使用。這意思是一個行程最多可以有 64 個「對各執行緒有不同意義」的 DWORDs。爲了在每個執行緒中保留一個 slot，程式應該呼叫 *TlsAlloc*。每次呼叫 *TlsAlloc* 就傳回一個可被所有執行緒使用的索引值。這個索引值常常被儲存在全域變數中。當執行緒要對一個 slot 寫入資料，它使用 *TlsSetValue*，交待一個 TLS 索引以及一筆資料。稍後當執行緒要取出此值，它呼叫 *TlsGetValue*，再次交待一個 TLS 索引。最後，程式呼叫 *TlsFree* 並交待一個 TLS 索引，將 slot 釋放掉。這麼一來當然也就讓 slot 不再能夠被任何執行緒使用，因爲 TLS 索引值在各執行緒之間是共通的。

> 譯註：如果你想明瞭上一段話的意義和 TLS 在系統層面上的實際影響，請詳讀稍後的四個 TLS 函式細部說明。

雖然 TLS 可以存放單一數值如檔案 handle，更常的用途是放置指標，指向執行緒的私有資料。有許多情況，多執行緒程式需要儲存一堆資料，而它們又都是與各執行緒相關。許多程式員對此的作法是把這些變數包裝爲 C 結構，然後把結構指標儲存在 TLS 中。當新的執行緒誕生，程式就配置一些記憶體給該結構使用，並且把指標儲存在爲執行緒保留下來的 TLS 中。一旦執行緒結束，程式碼就釋放所有配置來的區塊。

這種程式風格的最佳示範就是第 10 章的 APISPY32。APISPY32.DLL 需要保持一個堆疊，用來傳回它所攔截的函式位址（我在這裡使用古典的計算機科學術語「堆疊」，事實上我指的是一個結構陣列，以及一個堆疊指標）。由於被攔截的程式可能有許多執行緒，APISPY32.DLL 必須針對每一個執行緒保留各自的傳回位址。

如果每一個執行緒有 64 個 slots 用來儲存執行緒自己的資料，這些空間打哪兒來？稍

早我曾說過，每一個 thread database 有 64 個 DWORDs 給 TLS 使用。當你以 TLS 函式設定或取出資料，事實上你真正面對的就是那 64 DWORDs。沒有任何公開文件告訴我們可以存取其他執行緒的 TLS。讓我們更詳細地看看這些 TLS 函式。

## TlsAlloc

由於 TLS 只提供最多 64 slots 給每一個執行緒使用，所以必須有某種方法追蹤哪一個 slot 已被使用。KERNEL32 使用兩個 DWORDs（總共 64 個位元）來記錄哪一個 slot 是可用的、哪一個 slot 已經被用。這兩個 DWORDs 可想像成爲一個 64 位元陣列，如果某個位元設立，就表示它對應的 TLS slot 已被使用。

這 64 位元 TLS slot 陣列存放在 process database 中（可能你會猜想在 thread database 中，不，不是這樣）。記住，當你配置一個 TLS slot，這個 slot 可以在行程所屬的任何執行緒中被該索引值參考到。64 位元的 TLS slots 陣列放在 process database 的 0x88 和 0x8C 兩個 DWORD 欄位。雖然下面的 *TlsAlloc* 函式虛擬碼可能看起來有點複雜，事實上並不會。其中所做的只不過是掃描 64 位元陣列中的位元，看看有沒有哪一個是 0。如果找到，就把它改爲 1，並傳回其陣列位置（索引值）。因此，如果第 5 個位元是 0，*TlsAlloc* 就把它改爲 1 並傳回 4（索引值從 0 開始）。

**TlsAlloc 函式的虛擬碼**

```
// 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

*TlsSetValue* 可以把資料放入先前配置到的 TLS slot 中。兩個參數分別是 TLS slot 索引值以及欲寫入的資料內容。函式首先檢查陣列索引是否合法（小於 64）。Windows 95 的早期版本還會檢查此一索引是否的確被配置了，但是在 beta3 版本中就只做上述的最簡單檢查。如果索引值的確小於 64，*TlsSetValue* 就把你指定的資料放入 64 DWORDs 所組成的陣列（位於目前的 thread database）的適當位置中。

除此之外，*TlsSetValue* 還更新第二個 64 DWORDs 陣列。這個陣列內含 EIP 值，*TlsSetValue* 上一次就是在那裡被呼叫。很明顯，這些 EIP 是爲了除錯用的。微軟並沒有提供什麼方法讓應用程式取用這塊資料。

**TlsSetValue 函式的虛擬碼**

```
// 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

這個函式幾乎是 *TlsSetValue* 的一面鏡子，最大的差異是它取出資料而非設定資料。和 *TlsSetValue* 一樣，這個函式也是先檢查 TLS 索引值合法與否。如果是，*TlsGetValue* 就使用這個索引值找到 64 DWORDs 陣列（位於 thread database 中）的對應資料項，並將其內容傳回。

**TlsGetValue 函式的虛擬碼**

```
// 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

這個函式將 *TlsAlloc* 和 *TlsSetValue* 的努力全部抹消掉。*TlsFree* 先檢驗你交給它的索引值是否的確被配置過。如果是，它將對應的 64 位元 TLS slots 位元關閉。然後，爲了避免那個已經不再合法的內容被使用，*TlsFree* 巡訪行程中的每一個執行緒，把 0 放到剛剛被釋放的那個 TLS slot 上頭。於是呢，如果有某個 TLS 索引後來又被重新配置，所有用到該索引的執行緒就保證會取回一個 0 值，除非它們再呼叫 *TlsSetValue*。

**TlsFree 函式的虛擬碼**

```
// 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;
```

# 執行緒的雜項函式

本節描述的各個函式無法歸類於前面各個主題中。雖然如此，它們都是十分重要的函式，可以強調「KERNEL32.DLL 使用 thread database」這一事實。

## GetLastError

*GetLastError* 是一個機制，應用程式可以利用它決定爲什麼某個系統呼叫會失敗。當 Windows 95 函式失敗，它可以選擇性地在目前的執行緒中設定「最後錯誤代碼」（last error code），指示失敗的原因。WINERROR.H 中定義有許多錯誤代碼。*GetLastError* 有點像 C 函式庫中的 errno 變數。

除了系統函式，應用程式也可以參與這場派對，並且呼叫 *SetLastError*。那些錯誤代碼最好是和系統定義的錯誤代碼不同。

*GetLastError* 函式動作很簡單。在確定的確有備詢對象（某個執行緒）之後，它就傳回該執行緒的 thread database 的 GetLastErrorCode 欄位（70h 欄位）內容。如果沒有備詢對象（某個執行緒），此函式就傳回 KERNEL32 全域變數中的值。該全域變數出現在下面的虛擬碼中（我沒有給它命名）。

**GetLastError 函式的虛擬碼**

```
if ( ppCurrentThread )
       return ppCurrentThread->GetLastErrorCode
   else
       return x_LastErrorIfNoCurrentThread;    // A global variable.
```

## SetLastError

*SetLastError* 也非常簡單。如果目前存在有執行緒，這個函式就設定其 thread database 中的 GetLastError 欄位。

**SetLastError 函式的虛擬碼**

```
// Locals:
//      DWORD   fdwError

   if ( ppCurrentThread )
       ppCurrentThread->GetLastErrorCode = fdwError;
```

## GetExitCodeThread 和 IGetExitCodeThread

*GetExitCodeThread* 傳回由 hThread 所指定的執行緒的退出狀態。執行緒退出狀態記錄在 thread database 的 TerminationStatus 欄位（48h 欄位）中。正常執行情況下，其值爲 0x103（STILL_ACTIVE）。

*GetExitCodeThread* 只不過是參數檢驗而已，在檢查過傳進來的指標合法之後，它就呼叫 *IGetExitCodeThread*。

*IGetExitCodeThread* 首先進入一個「必須完成（must complete）」的段落中，然後把 hThread

轉換爲 thread database 指標。在取出 TerminationStatus 欄位後，它離開那個「必須完成」的段落。

**GetExitCodeThread 函式的虛擬碼**

```
// 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 函式的虛擬碼**

```
// 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 程式

爲了幫助搜尋各式各樣的資料結構，以及驗證我在前面提到的觀念，我寫了一個 Win32Wlk 程式。根據最初設定給它的任務，它將是一個很不錯的工具，用來探索 KERNEL32 資料結構。

如你在圖 **3-5** 所見，它是一個 GUI Win32 程式，顯示三個主要的資料結構：process database，thread database，以及 IMTE（modules）。此外，Win32Wlk 還可以顯示另三個資料結構：process handle table、process module list、Thread Information Block（TIB）。

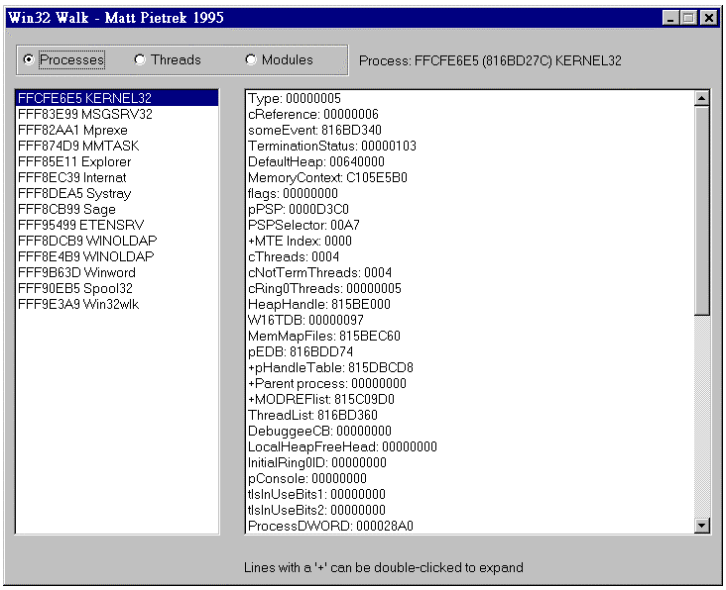



**圖 3-5a　The Process List**

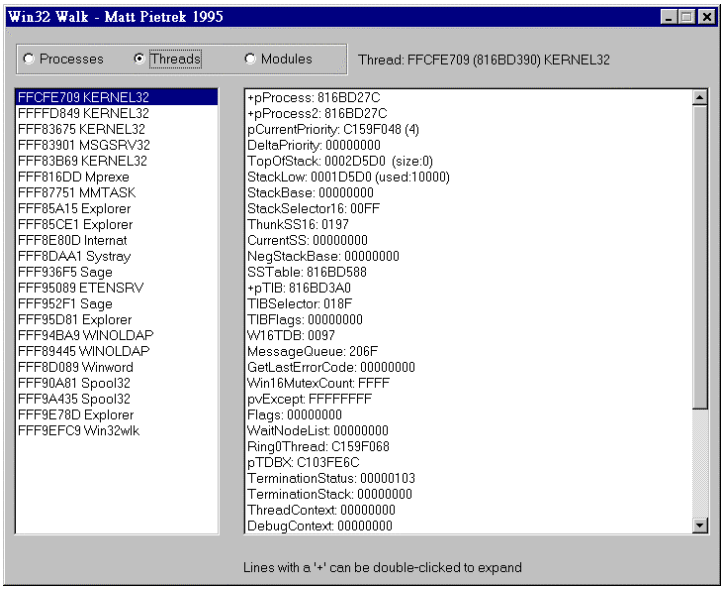



圖 3-5b The Thread List

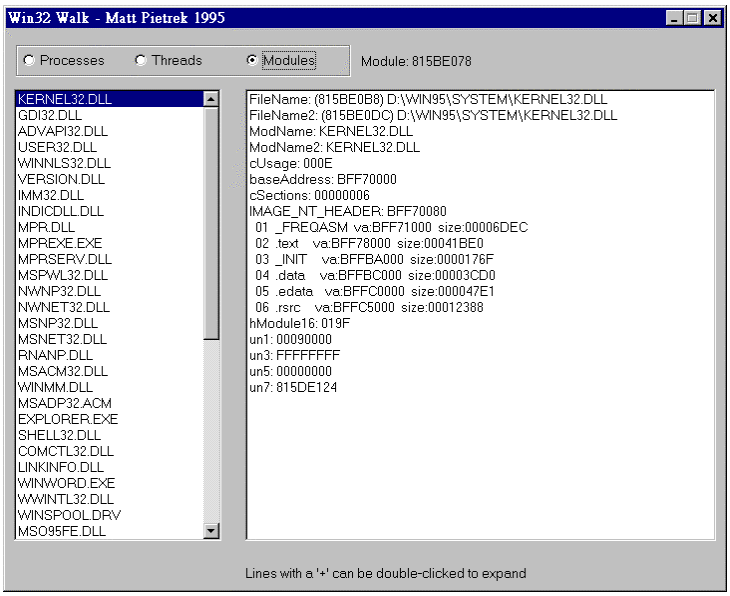



圖 3-5c The Module List

Win32Wlk 只有兩個視窗（列示清單）。左邊的那個總是顯示目前的串列（可能是行程、執行緒或模組）。你可以按下視窗左上方的圓鈕，切換這三種資料。注意，這些資料無法對新產生的行程、執行緒或模組做動態更新。任何時候你想要更新資料，請按下適當的圓鈕，Win32Wlk 就會重新計算內容。

右邊串列是細部視窗。任何時候它可以顯示 process database 或 thread database 或 IMTE 或 process handle table 或 MODREF 串列或 TIB 的欄位。細部視窗的內容以兩種方式更新。第一種，如果你點選左側視窗的其中一行，細部視窗就會現出被選中的行程（或執行緒，或模組）的各個有意義欄位。第二種方法就是使用熱聯結（hot link）。例如，當顯示行程資料時，pHandleTable 其實是 +pHandleTable。如果你在標有 + 號的那一行快按兩下，細部視窗就會顯示和該項目有直接關連的細部內容。如果你選擇的是 +pHandleTable，細部視窗會出現 process handle table。記住，細部視窗中出現的欄位並不和它們在結構中的欄位次序一致，而且我也並沒有把每一個欄位都秀出來。在佈置細部視窗時，我嘗試把資訊以相關趣味性排序。朦朧的欄位 -- 幾乎沒有人會在意的那種 -- 被我安排在清單底部。對於那些意義不明，以及總是爲 0 的欄位，我根本就不列出它們。我對每一個欄位執行 assertion 動作，判定它是否爲 0。

## Win32Wlk 程式的運作

我必須對一些技術障礙以及實作細節做更詳細的說明。當我開始寫 Win32Wlk 時，我希望我能夠走訪所有的資料結構，不使用任何 TOOLHELP32 函式。Win32Wlk 達成了大部份目標。然而，有兩個地方我終於還是不免要藉 TOOLHELP32 之助。行程串列和執行緒串列都是很難追蹤的，這是因爲它們並非像 Win16 KRNL386 所使用的那種簡單串列。幸運的是，TOOLHELP32 的 th32ProcessID 和 th32ThreadID 兩個值正是 *GetCurrentProcessId* 和 *GetCurrentThreadId* 傳回來的 PID 和 TID 值。因此，我得以利用 *Process32First*、*Process32Next*、*Thread32First*、*Thread32Next* 等函式獲得一系列的 process IDs 和 thread IDs。此後，Win32Wlk 即完全使用自己對資料結構的知識來找出任何東西。

找出 KERNEL32 把 IMTE 陣列（也就是系統模組串列）的指標放在什麼地方，是另一項挑戰。這個指標放在 KERNEL32.DLL 中，也就是先前我說過的 pModuleTableArray。在經過一些搜尋之後，我意外發現一個 KERNEL32 未公開函式 *GDIReallyCares*（輸出序號 #23）。當它被呼叫，ECX 暫存器中即放置 pModuleTableArray 指標。是的，這樣的策略實在令人作嘔。另有一些其他方法，可以找到 pModuleTableArray 陣列，但它們都太繁瑣討厭了。微軟沒有能夠提供一個 TOOLHELP32 函式以傳回系統模組串列，實在是很糟糕，否則我們就不必這樣硬砍硬剁了。

另一個令人作嘔的就是轉換 process IDs 和 thread IDs 成爲 process database 指標和 thread database 指標的過程。本章一開始沒多久，我就曾經說過 process ID 和 thread ID 其實就是一個 K32 物件指標經過 XOR（對象是 KERNEL32 的 Obsfucator DWORD）後的結果。XOR 的良好性質之一就是，它符合交換律，也就是說，如果：

```
(A XOR B) == C
```

那麼：

```
(A XOR C) == B
```

也就是說如果你知道 XOR 運算的任兩個值，你就可以知道第三個值。

現在，讓我們把真正的 KERNEL32 值套到上述公式中：

```
(PTHREAD_DATABASE XOR Obsfucator) == ThreadID
```

在 Win32Wlk 中，一開始我只知道公式中的一個值（*GetCurrentThreadId* 函式傳回的 Thread ID）。我們不知道 Obsfucator 是多少，因爲它每次開機後都不同。然而如果我們能夠獲得一個指標指向 THREAD_DATABASE，我們就可以重新安排計算公式：

```
(ThreadID XOR PTHREAD_DATABASE) == Obsfucator
```

有數個方法可以取得 THREAD_DATABASE 指標。我選擇最簡單的一個：取現行的 Thread Information Block（TIB）並減掉 0x10。TIB 由 FS 暫存器指出。在 TIB 偏移位置 18h 之處你可以發現 TIB 的線性位址。演算法十分單純：

```
pTIB = FS:[18h]
pThreadDatabase = pTIB - 0x10
```

你會在 Win32Wlk 的 WIN32WLK.C 中發現少量組合語言碼。一旦獲得一個名爲 Unobsfucator 的變數值，你就可以計算 process ID 和 thread ID 了。

最後請注意一點：Win32Wlk 是否能持續在 Windows 95 的更新修定版上有效運作呢？只要看看 Win32Wlk 多麼輕易就找出並解開那些系統資料結構，Windows 95 小組可能會再放置第二個 XOR，或是改變 *GDIReallyCares* 函式，使 IMTEs 陣列指標不再放在 ECX 暫存器上。

如果微軟選擇無償地破壞別人的程式（像我的 Win32Wlk），只因爲後者做了一些非經認可的行爲，那麼我還有第二個方法找出我要的資訊。我希望這之間的格調不要降低到諜對諜的狀態。微軟應該了解，微軟以外的人可以使用這些資訊做出更好的產品。以 Win32Wlk 來說，程式並不企圖改變任何系統行爲，只是純粹幫助程式員對系統有較好的了解。微軟希望我們使用黑盒子來寫像 "Hello World" 的程式，但是對於玩具以外的程式，很遺憾黑盒子不管用。

如果 Windows 95 小組真是因爲考慮到安全性而希望系統資料結構不要開放，他們應該把 Windows 95 設計得像 Windows NT 那樣。如果程式員需要的資訊嚴重不足，他們就會想辦法獲得它。爲什麼要在作業系統上加數層垃圾以防堵那些努力呢？Windows 95 並不是 Windows NT，它也從來沒有被認爲應該是。爲什麼要把那些真正需要這些資訊的人的生活弄得這麼複雜？

## 摘要

模組、行程和執行緒組成了 Windows 95 的核心。在這一章中我們看到了它們的資料結構，也看到了使用這些結構的各個 API 函式。第 8 章講述 PE 檔案格式時對於模組有更詳細的說明。後續各章中你還會遭遇模組、行程、執行緒的一些觀念。了解它們，你也就打開了了解 Windows 95 的大門。

# USER 和 GDI 子系統

什麼都還沒說，就先來一個道歉啓事，似乎有點奇怪，不過這的確是我想做的事。一如本章標題所示，我要從各個角度挖掘並描述 Windows 95 的 USER 和 GDI 模組。USER 模組負責傳遞訊息並管理視窗，GDI 則是 Windows 圖形系統的核心。在螢幕上產生一個視窗，需要 USER 模組和 GDI 模組的大量配合。因此，你可以想像，光是描述兩個模組的最上層動作就大約可以寫出兩本書。這也是爲什麼我道歉並請求各位免除我描述兩個模組魔術般行爲的原因。這一章焦點放在 Windows 95 的 USER 和 GDI 模組從 Windows 3.1 以來的進化，我放棄描述 Windows NT 的 USER 和 GDI 模組。

Windows 95 之中嶄新而重要的 USER 和 GDI 機能，例如通用控制元件（common controls），也不是我打算放進來的題目。我甚至開玩笑地建議本書的技術檢閱者，或許寫一本 WndProc Internals，並在這本（純粹假設性的）書籍中，詳細列出所有標準的系統視窗函式（例如 button 的視窗函式、tooltips 的視窗函式）虛擬碼，可能會有市場。這一章裡頭，最接近這個目標的該算是 desktop 的視窗函式了，稍後我將展示它的一段虛擬碼給你看。

你已經知道這一章不談什麼了。那麼，這一章到底談些什麼？喔，很多，而且馬上會證明給你看。光只是重新撰寫 Windows USER 和 GDI 模組的核心碼以應付 Win32 API 的需求，就足以使這些模組的內部有著乾坤扭轉的變化。如果你對 Windows 3.1 上的這些模組有適當認識，這一章可以幫助你了解 Windows 95 的新模組。我要把這一章切割爲兩部份。是的，你猜對了，一個是 USER，一個是 GDI。USER 這一部份忒大，因爲 USER 的改變比較戲劇性。如果你了解 Windows 95 對 USER 的改變，那麼要領悟 GDI 的改變就不難。

# Windows 95 USER 模組

寫作期間，我一直爲了如何將 USER 的改變做一個漂亮的分類而掙扎。事實證明，要把 USER 的改變歸納爲一或二個特定類別，並不容易。Windows 95 的 USER 模組既非游魚亦非鳥禽：訊息傳遞系統的主要部份還是在 16 位元 USER.EXE，但是有一些 32 位元碼散落在整個 16 位元模組中。這個 16 位元 USER.EXE 的一部份和 Windows 3.1 相同，其他部份則從根本上重新翻寫。

Windows 95 的 USER 模組也包含 32 位元的 USER32.DLL。Win32 的 EXEs 或 DLLs 即係以此做爲介面。或許你已經聽說過，USER32.DLL 只是一堆用來下移（thunk down）至 16 位元 USER.EXE 的程式碼。雖然 USER32.DLL 中大部份碼的確只是用來做下移動作，但其中也不乏表現不俗的函式，完全不需要下移至 16 位元碼。稍後我們會看到一些這類函式。

嘗試以優雅的設計和實作來完成 16 位元 USER.EXE 和 32 位元 USER32.DLL，似乎是件不可能的任務。我對 Windows 95 開發人員的最大讚美就是，他們竟然能夠維持「回溯相容」和「滿足 Win32 API 規格」兩個目標之間的美妙平衡。很多情況下，「回溯相容」和「堅守 Win32 API 規格」是不可得兼的事。這會導至無可避免的妥協並做出無法讓任何人都滿意的決定（或許，這個時候你腦中浮現的是 Win16Mutex？）。所有的狀況都被考慮過了，我想 Windows 95 開發小組做出了值得喝采的成績。

爲了對 Windows 95 USER 模組有些感覺，我想，看看另一個 Win32 弟兄應該是頗有助益的。Windows NT USER 是一個全然的 32 位元模組，它的最重要任務就是完成 Win32 API。回溯相容固然好，卻不是絕對必要。Windows NT 中的 16 位元 USER.EXE 將會上移（thunk up）到 NT 的 USER32.DLL 中。

至於幾乎已被人遺忘的 Win32s，嘗試儘可能以原有的 16 位元 Windows 3.1 USER.EXE 提供儘可能多的 Win32 API。16 位元 USER.EXE 則完全沒有改變。

Windows 95 落在這兩個極端之中的哪一點呢？雖然 Win32 純粹主義者（我自己就是其一）希望 Windows 95 走上 NT 之路，但事與願違。Windows 95 爲迎合承繼自 Windows 3.1 的主流市場口味，不可能犧牲回溯相容性。世界上有太多太多軟體依賴 16 位元 USER.EXE 的奇行怪癖。我知道微軟對此的標準態度是：『嘿，早就告訴過你們，不要用那些未公開的東西...。』

除了回溯相容，另一個因素也使微軟決定保留 16 位元 USER.EXE 的核心碼：程式碼「大小」問題！是的，一般而言，32 位元碼需要比較多的空間，因爲許多指令的操作元（operands）的大小都增加了（老實說，這個觀念曾經被激烈討論過，有些動作，使用 32 位元指令反而比較節省空間）。整體而言，微軟程式員估量過，把 USER 模組以 32 位元全部重寫，大約會使大小膨脹 40%。而你知道，Windows 95 一直就希望能夠在 4MB 機器上跑得不比 Windows 3.1 差（嚇，微軟企劃人員一定希望我寫「跑得比 Windows 3.1 好）。因此，將 USER 重寫爲 32 位元模組（像 Windows NT 那樣）並非可行之道。

於是，一個純粹的 Win32 USER 模組就此出局。Windows 95 小組接下來倒是做了件好事。他們從 Windows 3.1 USER.EXE 出發，並且被允許更改其中的碼。由於 Windows 95 需要至少 80386 機器，所以 Windows 95 USER 工作小組狂暴地修改了 USER.EXE。在 USER 的 16 位元程式節區（code segments）中到處都有 32 位元指令。這也就是爲什麼你會在 Windows 95 的 USER 程式節區中發現這麼多的 size-override opcodes（66h）的原因了。

要知道，許多 USER 程式碼是以 C 語言完成的，PC 上的 C 編譯器根據記憶體模式完成其編碼動作。一般的 16 位元 C 編譯器如 Borland C++ 吐出 16 位元指令，並使用 16 位元偏移值（offset）抓取資料。即使 16 位元編譯器能夠產生 32 位元指令，它還是不可能取 64KB 節區以外的資料。

32 位元編譯器就不同了。它們使用 flat 記憶體模式，這個模式使編譯器忘了節區的存在。它們所產生出來的碼根本不會使用到 code selector、data selector，或是 stack selector（CS、DS、SS 暫存器）。Windows 95 的 USER.EXE 就像是 16 位元碼和 flat 模式的混合。也就是說，USER.EXE 的碼存在 16 位元節區中，並且明顯使用節區暫存器。但它內含一些指令，使用的位址超越 64KB 節區。看看下面這段來自 USER.EXE 的碼：

```
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
```

這些指令的大小（例如第一個指令佔三個位元組）證明它們的確是 16 位元碼。前兩個指令明顯設定一個節區暫存器，攫取一個位在 USER DGROUP 的 062Eh 處的全域變數。第四個指令使用 EAX 暫存器做爲位址計算的一部份。這段碼真正執行時，EAX 內含的值超過 128K。我從來不曾看過有哪個編譯器可以產生出 16 位元碼，卻又使用 32 位元的資料偏移值（offset）。我懷疑 Windows 95 小組是不是用了一個來自微軟程式語言部門的特殊編譯器。喔，是的，根據不願具名的消息來源指出，的確有這麼一個編譯器存在。

雖然 16 位元 USER.EXE 的許多改變是爲了提供更進步的功能（畢竟耗盡 heap 空間是 Windows 3.1 長久以來的問題），也有許多改變是爲了支援 Win32 API。這也是 95 小組企圖趕上 NT 小組的地方。例如 Win32 的 *AttachThreadInput* 函式（把某個執行緒的輸入狀態和另一個執行緒關聯起來），並沒有 16 位元弟兄。甚至沒有任何一個 Win16 API 有類似的功能。Windows 95 的 16 位元 USER.EXE 硬是重頭做出了 *AttachThreadInput* 函式，但是它很謙虛，沒有開放出來。反倒是 USER32.DLL 開放了它。事實上

USER32.DLL 的 *AttachThreadInput* 函式只是做一個簡單的下移（thunk down）動作。USER32.DLL 獲得了所有榮光，USER.EXE 則扮演灰姑娘的角色。

另一個例子是資源的取用。你將在第8章看到，儲存在 Win32 PE 檔案中的資源是以一種完全迥異於 NE 檔案資源格式的方式排列。而又如第7章所示，Windows 95 爲 32 位元模組產生的 16 位元 NE 模組資料庫，內含一個指標，指向「Win32 模組中的資源」載入記憶體後的基底位址。原因如下：16 位元 USER.EXE 的負擔加重了，既支援舊的 16 位元 NE 格式中的資源，又支援新的 32 位元 PE 格式中的資源。USER32.DLL 之中與資源相關的函式已被貶至純粹做下移（thunk down）工作而已。

## USER32 thunking 實例

這一節的主題是 thunking，現在正是解釋 thunking 在 Windows 95 之中如何運作的好時機。Windows 95 極爲依賴 16 位元和 32 位元之間的 thunking（移轉），所以要真正了解 Windows 95，絕對不能避談 thunking。讓我們看一個從 USER32 函式下移（thunk down）至 USER.EXE 的典型實例。我選擇的是 *SetFocus*，此函式有一個參數，這個參數並不需要轉換其值，就可以在 USER.EXE 中使用（在 Windows NT 中則有不同的故事，但那應該是另一本書的責任）。

### SetFocus

*SetFocus* 和許多其他的 USER32 函式一樣，被下移（thunk down）至 USER.EXE。在 USER32.DLL 的除錯版中，此函式一開始即呼叫一個「logging 函式」，也就是專門負責記錄運轉過程的程式。如果 USER32 資料區域中曾經設立某個特殊旗標，*SetFocus* 就會吐出字串 "[F]SetFocus" 到除錯埠中。USER32 的 *SetFocus* 函式將一個索引值載入 CL 暫存器，用來索引一個基本上是 16:16 位址的 "jump table"。在 *SetFocus* 函式中，這個索引值是 0x7E，意味著 "jump table" 的第 0x7E 個元素是一個指標，指向 16 位元版的 *SetFocus*。

載入 CL 之後，*SetFocus* 跳到一小塊程式碼去，我稱之爲 *ThunkToUSER16_One_Param*。這一小塊碼是許多 USER32 函式都可以呼叫的一個共同進入點，有一個參數，下移（thunk down）至 USER.EXE。*ThunkToUSER16_One_Param* 把呼叫端的參數以及所謂的 "thunk index" 壓到堆疊去，然後呼叫一個我稱之爲 *CommonThunk* 的函式：

**SetFocus 虛擬碼（32 -> 16）**

```
LogWin16ThunkFunction1("[F] SetFocus");

CL = 0x7E  // Thunking index for SetFocus

goto ThunToUSER16_One_Param
```

**ThunkToUSER16_One_Param 虛擬碼**

```
// Parameters :
//  DWORD  param1
//  DWORD  thunkIndex  // Actually in CL register

   return CommonThunk ( Param1, thunkIndex );
```

*CommonThunk* 是如此簡單，把它轉換爲 C 虛擬碼反倒模糊了其動作。由於某些未知的理由，這段碼被放在 USER32 的資料區域中。或許這段碼是啓動時動態產生的也說不定。它首先取出 thunk index（本例爲 0x7E），並以它做爲陣列索引，指向一個由 16:16 指標構成的陣列元素。然後把該指標取出放到 EDX 暫存器中。最後，*CommonThunk* 跳到 KERNEL32.DLL 的 *QT_Thunk* 函式（稍後描述）。

**CommonThunk 函式碼**

```
// 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

*QT_Thunk* 由 KERNEL32.DLL 開放出來。這是一個一般化函式，供那些需要做下移（thunk down）動作的 Win32 碼使用。換句話說它並不只是侷限給 KERNEL32 或 USER32 使用。事實上如果你觀察 Win32 SDK 的 thunk compiler （THUNK.EXE）的輸出，你就會看到其中對 *QT_Thunk* 的使用和呼叫。

*QT_Thunk* 是以組合語言撰寫，並在空間和速度上做了最佳化。我本來考慮在本節中使用原原本本的組合語言碼，但是這麼一來很容易就會失去控制，除非你是一個組合語言高手。因此，稍後你看到的虛擬碼混合了 C 語言和組合語言。我已盡力把一個複雜的函式表達出來。如果你真的想看到底發生了什麼事，請你在 SoftIce/W 中爲 *QT_Thunk* 設立一個中斷點（break point），我保證不會等太久這個中斷點就會被喚起。

*QT_Thunk* 很簡單：從 EDX 中取出 16:16 位址，然後把控制權交給該位址。當然啦，不可能這麼簡單，一定有什麼其他要注意的！一開始就把「控制權回返之後要接續執行的位址」記錄下來，應該饒有助益。將堆疊從一個 flat 32-bit stack selector 切換到一個 16-bit selector，也是個好主意。

讓我們近一點看這個函式。*QT_Thunk* 被切分爲五個階段。首先，在除錯版，*QT_Thunk* 呼叫一個負責做運轉記錄的函式 -- 假設正確的旗標值設立了的話，不過通常並不。然後驗證 TIB（Thread Information Block，第3章）的 selector 和 FS 暫存器是否相同。如果不相同，*QT_Thunk* 會發出抱怨（這是除錯版，別忘了）。

第二階段，*QT_Thunk* 把 16:16 位址（thunk 的終極目標）壓到堆疊中。我們將在第五階段再回到這裡來。第二階段並且完成了回返位址以及 32 位元暫存器的保存動作。從 16 位元碼回返時的那個回返位址，儲存在堆疊中一個碰觸不到的地方。被儲存的暫存器則是 ESI，EDI 和 EBX。這些都是常常被使用的暫存器，Win32 編譯器希望它們被保存起來（請看第3章）。

*QT_Thunk* 的第三階段是索求 Win16Mutex。我想現在全世界都知道了，當 32 位元碼下

移（thunk down）至 16 位元碼，系統必須要求 Win16Mutex，那是一個 mutex semaphore，駐在 KRNL386.EXE 的資料節區中。只要強迫讓所有的 Win32 碼在下移至 16 位元世界時都能夠索求 Win16Mutex，Windows 95 就能夠保證一次只執行一個穿越 Win16 system DLLs（以及其他 16 位元 DLLs）的執行緒。

這就是微軟「讓 16-bit system DLLs 在多執行緒環境下正常工作」的方法。Win16Mutex 的使用具有高度爭議性，我可以爲這個題目輕易寫出一整章。我將在「訊息系統的改變」一節中對這個傢伙多談一點。這裡我只能告訴你，*QT_Thunk* 是索求 Win16Mutex 的眾多地點之一。

*QT_Thunk* 的第四階段是把 flat 32 位元堆疊切換爲 Win16 所使用的 16:16 堆疊。由於 Win32 執行緒通常有 1MB 堆疊，而這時候的 ESP 可能位於這 1MB 的任何位置，所以你看到的切換動作十分詭異。*QT_Thunk* 不能夠只是在執行緒啓動時配置一個 16-bit stack selector 並設定其基底位址，它必須在移轉至 16 位元碼時，調整 selector 的基底位址，此 selector 將在執行緒執行 16 位元碼時使用。此一 16 位元 selector 的基底位址被設定與 ESP 暫存器所使用的線性位址相同。在欺騙了 stack selector 之後，*QT_Thunk* 終於計算出一個適當的 16 位元 SS:SP 組合，並把它們儲存到 SS 和 SP 暫存器中。

最後一個（第五）階段是把控制權交給 16:16 位址，那也正是 thunk 的目標地。一如階段二所示，16:16 目標位址儲存在進入 *QT_Thunk* 當時的 EDX 之中，然後被壓入堆疊。*QT_Thunk* 經由標準的 RETF 技巧跳到 16:16 位址。然而在把控制權交到該位址之前，*QT_Thunk* 把所有非必需的節區暫存器（DS、ES、FS、GS）統統清爲 0。It wouldn't do to hand the target 16:16 function a DS register set up with a nice, juicy flat 32 selector for the function to scribble on. It's expected that the 16:16 function will set up the segment registers however it needs to.

譯註：很抱歉我不懂上一小段的真正意義，不敢妄加翻譯。原文保留給您。

**QT_Thunk 虛擬碼**

```
// 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 位元碼完成了它的工作之後，必須回到 32 位元碼去。另有一大段碼負責這件事情。雖然我也可以詳詳細細地說它，但其中並沒有什麼令人興奮的地方。注意，在這個下移（thunk down）至 16 位元的例子中，並沒有任何 flat 32-bit 位址需要被轉換爲 16:16 位址。如果必須做這件事情，thunking 碼就更複雜了。我不想在這裡討論它。

## 32-bit heaps

或許 USER 子系統中最大最劇烈的改變就是 32 bit heaps 了。你也許知道，任何 Win32 程式都可以經由 Win32 *HeapXXX* API 函式處置 32-bit heaps。但你或許不知道，16-bit USER.EXE 和 16-bit GDI.EXE 也使用 32-bit heaps 儲存某些資料。事實上 16-bit USER.EXE 和 GDI.EXE 上移（thunk up）至 32-bit KERNEL32.DLL，從特別的 32-bit heaps 中配置記憶體。那些 32-bit heaps 是特別設計給 16-bit USER.EXE 和 GDI.EXE 使用的。雖然那些 heaps 是特別要給 USER 和 GDI 使用，它們的格式和 Win32 程式呼叫 *GetProcessHeap* 所獲得的 heap 格式相同。你可以使用第5章的 WALKHEAP 程式走訪 USER 或 GDI 的 32-bit heaps（當然你得先找到它們。稍後我將示範怎麼做）。

爲什麼要使用 32 bit heaps？在 Windows 95 之前，所有配置給 USER 和 GDI 使用的記憶體，都是來自 *LocalAlloc* 所取得的 heap，最大值爲 64K。不消說，這使得同一時間存在的視窗物件和圖形物件的個數受到很大的限制。如果能夠把其中的大塊物件搬移到 32-bit heaps 去，Windows 95 就能夠明顯改善系統的能力。每一個這樣的 heap 大小是 2MB，所以容量不再是問題。

USER.EXE 使用兩個 32-bit heaps。其中一個用來儲存 WND 結構。系統的每一個視窗都有一個對應的 WND 結構。稍後我會告訴你 WND 結構的詳細內容。另一個 32-bit heap 用來存放 menu。GDI.EXE 只有一個 32-bit heaps。用來儲存 fonts 和 regions。就像 WNDs 和 MENUs 之於 USER 是十分巨大的物件，fonts 和 regions 之於 GDI 也是十分巨大的物件，所以把它們搬移到 32 bit heap 十分合理。

如果說 16-bit 模組擁有 32-bit heaps 是個大消息，那麼這些 heaps 位在哪裡就更有趣了。當處理 32-bit heaps 中的資料，USER 和 GDI 並不使用 flat 線性位址，而是繼續使用相同的 DS selector，那是它們用來存取其正規 128K DGROUP 用的。爲什麼能夠如此？因爲經由十分有趣的重新安排，32-bit WND heap 和 32-bit GDI heaps 接續在 128K 16-bit DGROUP 區域之後。如果這聽起來很怪異，**圖 4-1** 或許能夠讓你清醒一些。

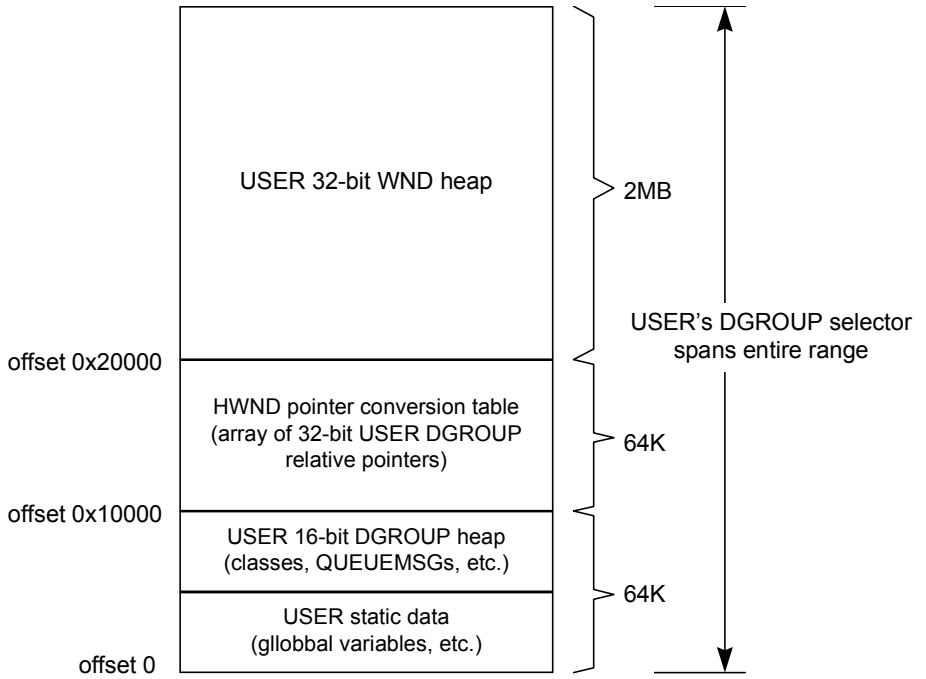



**圖 4-1　USER.EXE 的 16-bit heap 和 32-bit heap 組態**

一如我說，USER 和 GDI 並不使用 32-bit flat 指標指向 32-bit heap 中的物件。取而代之的是，它們儲存「以 USER 或 GDI 的 DGROUP selector 爲基底位址」的偏移位置。這些偏移位置當然是 32 位元。舉個例子，USER 的 16-bit（128K）DGROUP 的極大值爲 64K，32-bit WND heap 接續在 128K 16-bit DGROUP 區域之後。這意思是你所能獲得的 WND 結構偏移值的最低可能是 0x20000。實際應用上（你可以在第５章看到），Win32 heap 的最前面區域用來當做簿記區。所以一個典型的 WND 結構的偏移值是像 0x20924 這樣的值。由於偏移值並不是 flat 線性位址，它對於 selector（我指的是 USER 或 GDI 的 DGROUP）毫無意義。當然如果你知道 USER 或 GDI 的 DGROUP 線性位址，你就可以直接把該偏移值加上去，當做是一個 flat 線性位址。稍後的 SHOWWND 程式就是這麼做。

現在我要證明 32-bit WND heap 真的是從正規 DGROUP 的 128K 以上開始，並且證明它真的是一個標準的 Win32 heap。爲了這麼做，我們使用 SoftIce/W。首先我必須找出 USER's DGROUP 的基底位址。爲了找到它，我必須先找到 USER's DGROUP 的 handle 或 selector。第7章會告訴你，一個模組的 DGROUP 可以從其 module database 中取得。

在 SoftIce/W 中對 USER 下達 MOD 命令，導至下面結果：

```
:mod   user
hMode  PEHeader       Module Name    EXE File Name
17CF                  USER           C:\WINDOWS\SYSTEM\user.exe
1857   0147:81537DB8  USER32         C:\WINDOWS\SYSTEM\USER32.DLL
```

我們得知 USER 的 module handle 是 17CF。而 module database 的 #8 偏移處是一個近程指標，指向一塊 10 個位元組的「DGROUP 節區記錄」，所以讓我把那些記錄傾印出來看：

```
:dw 17cf:8
17CF:00000008 0180 10D9 C341 0021 157C 0000 1F42 0015    ....A.!.|...B...
```

好，17CF:180 放的就是 10 個位元組的 DGROUP 節區記錄，其中最後一個 WORD 是該節區的 handle。現在我把這 10 個位元組的節區記錄再傾印出來：

```
:dw 17cf:180
17CF:00000180 4042 0B02 0177 157C 16C6 0005 800C 000F    ....A.!.|...B...
```

現在我們知道 USER 模組的 handle 是 16C6 了（而其對應的 selector 將是 16C7）。讓我們再利用 SoftIce/W 的 LDT 命令取其線性位址（請注意其大小超過 64KB）：

```
:ldt 16c6
16C7  Data16     Base=81D09000   Lim=0021FFFF    DPL=3  P   RW
```

於是我們知道 USER 的 DGROUP 線性位址是 81D09000。再加上 0x20000 就可以獲得 USER32 window heap。讓我們測試看看（使用 SoftIce/W 的 "Heap 32" 命令）：

```
: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...
```

你看，SoftIce/W 的確接受了我們餵給它的位址，並且印出看起來相當合理的結果。特別請你注意，每一塊資料都是大約 0x58 個位元組。稍後我們將看到，0x58 正是一個 WND 結構的最小可能。如果有比 0x58 個位元組大的區塊，應該是因爲它們使用了 window extra words（這個值在我們註冊類別時以 WNDCLASS 結構的 cbWndExtra 欄位指定之）。從各個角度來看，這的確是一個 Win32 heap，駐在 USER's DGROUP 的最初 128K 之上。

也許你會奇怪爲什麼 32-bit heaps 要駐紮在 USER 或 GDI 的 128K DGROUP 之上（難道你不覺得奇怪嗎？）爲什麼不就在原來的 16-bit 64K DGROUP 之上？概括地說，答案就是 Handles！雖然 WND 結構是根據 32 位元偏移值（以 USER DGROUP 爲基準）定位出來，但是麻煩的回溯相容使得 HWNDs 必須是 16 位元。

在 Windows 3.x 及更早之前，HWND 只不過代表 USER DGROUP 節區中的一個偏移位置。很明顯，這樣的 HWND 不適用於新的 16-/32- 位元混合體。爲了讓一個 16 位元偏移值能夠轉換爲一個 32 位元偏移值，USER 和 GDI 以 16-bit DGROUP 和 32-bit heap 之間的一個 64K 空間做爲 handle table。Handle（例如 HWND）於是成爲此一 handle table 的偏移值。圖 **4-2** 顯示，在 handle 所指出的位置上，你可以發現一個 32 位元偏移值（相對於 DGROUP 起始位址），指向真正的物件所在。

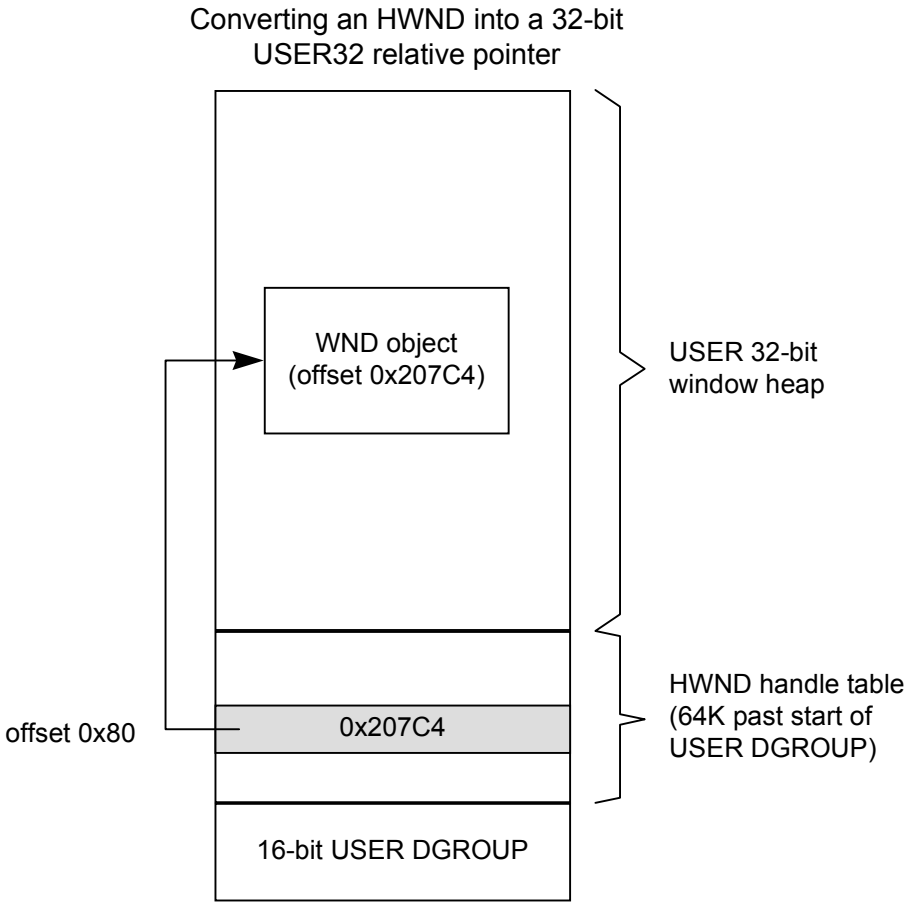



**圖 4-2 眞正資料的 32 位元偏移值(相對於某個 DGROUP)可以從 handle table 中獲得。**

爲了證明 handle table 的存在,我再次使用 SoftIce/W。我選擇 desktop 視窗的 HWND,並經由 handle table 看個仔細。SoftIce/W 的 WND 命令提供對視窗串列的一種階層架構式的觀察,desktop 視窗是最上層。下面的輸出中,desktop 視窗的 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...
```

如果我先前所說是真的，我們就應該能夠把 HWND 值加上 0x10000，並且在該處獲得一個 DWORD，代表 WND 結構的位置。0x10000+0x0080=0x10080，所以再讓我們看看 16C7:10080 的記憶體內容：

```
:dd 16c7:10080
16C7:00010080 0002:0178 0002:01E0 0002:0248 0002:02B4    x.......H.......
```

冒號（:）是 SoftIce/W 插進來的，企圖使上面這些值看起來像 16:16 指標。不要理它。我們獲得 WND 結構的偏移位置是 0x20178。而由於 USER 的 DGROUP 基底位址爲 0x81D09000，所以我們算出 WND 結構位於線性位址 81D29178 處。回頭看看稍早我所顯示的 SoftIce/W 巡訪 32-bit USER heap 的結果，可以看到 0x81D29178 的確是其中一個區塊的起始位址。好極了，每一樣東西似乎都通過了檢驗。

GDI 模組也使用相同的 handle table 機制，把某些大型的 GDI 物件移置到 32-bit heap 中。例如，regions 就被放在 32-bit heap 中，可以一個 HRGN 結構代表之。你可以遵循類似上述的步驟，找出這個 region 物件的線性位址。

如果 handle table 大小是 64K 而每一個 handle 是 4 個位元組，那麼 handles 最多可爲 16384 個（65536/4=16384）。微軟宣稱他們現在可以提供 32767 個視窗和 32767 份選單（menus），我不知道那數字是怎麼來的。不論如何，在你產生一萬六千個（或三萬兩千個）視窗之前，其他的系統限制也是一大問題。

稍早以前，我曾經說過 USER 還有一個 32-bit menu heap。Menu heap 的佈局及其 handle table 機制和 window heap 差不多。唯一欠缺的是 handle table 之下的那個 64K 16-bit DGROUP。或許你會認爲微軟把 menu 搬移到另一個 32-bit heap 是一件不得了的功績，其實沒有這麼驚天動地。Windows 3.1 的 menu 早就已經被搬移到另一個 16-bit heap 去了。Windows 95 的唯一改變就是增加 menu heap 的大小。附帶一提，menu heap 的基

底位址的 selector 可以經由 *UserSeeUserDo* 函式獲得。稍後我會介紹這個函式。

如果 USER 和 GDI 所使用之 32-bit heaps 的機能和 Win32 heaps 相同，那麼用來操作 Win32 heaps 的 KERNEL32 函式就應該也能夠用來操作 USER 和 GDI 的 32-bit heaps。事實上的確如此。當 USER 爲 WND 結構配置一塊記憶體，KERNEL32.DLL 中的 *HeapAlloc* 會被呼叫（經由一個 thunk）。然而 USER 和 GDI 並不是直接上移（thunk up）到 KERNEL32.DLL 去，而是由 KRNL386.EXE 提供一組未公開函式，負責呼叫 KERNEL32 的 heap 函式。這些 KRNL386.EXE 的未公開函式有：

KRNL386.209 - Local32Alloc
KRNL386.210 - Local32ReAlloc
KRNL386.211 - Local32Free
KRNL386.213 - Local32Translate（把一個 handle 轉化爲一個 16:16 位址）
KRNL386.214 - Local32FreeQuickly

雖然函式名稱都以 *Local32* 起頭，它們事實上是呼叫對應的 *HeapXXX* 函式，例如 Local32Alloc 呼叫 *HeapAlloc*。第5章將顯示，Win32 的 local heap 函式其事只是 Win32 *HeapXXX* 函式的一層薄薄包裝而已。請特別注意上述的 KRNL386.214 函式。這個函式看來是要釋放一個區塊，而且不上移（thunk up）至 KERNEL32。然而它沒有完成某些十分關鍵的動作，例如把釋放掉的區塊加到自由串列之中等等。

## 神秘的 GetFreeSystemResources

知道 32-bit heaps 之後，現在我們有足夠的知識基礎來討論神秘的 Free System Resource（FSR）了。我說神秘，是因爲 FreeSystemResource 的值似乎在 Windows 95 中獲得提昇，但從底層來看，沒有理由如此。對大部份非程式員而言，FreeSystemResource 是他們對 USER 和 GDI 模組的唯一概念。如果 FreeSystemResource 提昇，就一定是好事。是嗎？噢，不要那麼快下定論。

所謂 FreeSystemResource，只不過是個漂亮的術語，代表各個 system heaps 的剩餘記憶體（以百分率表示）。在 Windows 3.1 中，這個值是許多個百分率的最小值，這些百分

率代表以下各 heaps 的自由率：USER DGROUP heap、USER menu heap、USER string heap（此 heap 對於 Windows 95 不再重要）、GDI DGROUP heap。

在 Windows 95 之中，FreeSystemResource 的計算一開始與從前類似，但是愈到後面愈是有一些變化。總括地說，Windows 的 FreeSystemResource 計算方式是首先看看以下哪個 heap 的自由百分率最低：

1. USER 16-bit DGROUP heap
2. 32-bit window heap
3. 32-bit menu heap
4. 16-bit GDI heap
5. 32-bit GDI heap

由於三個 32-bit heap 大小都是 2MB，它們的自由百分率常常處於 99% 的高水位。因此它們可以說並不會影響 FreeSystemResource 的計算。剩下的是 16-bit USER 和 GDI 的 DGROUP heaps，誰的自由百分率比較小，誰就成爲 FreeSystemResource。畢竟因爲還是有一些東西浮游於其中，所以它們無論如何不應該有接近 96% 的高水位 -- 這卻是你一啓動 Windows 95 後在 Explorer 的【About】對話盒中看到的數值。

請你做個小實驗。啓動 Windows 95 然後立刻執行 CALC，或 Explorer，或其他一些由系統提供的程式。選按【Help/About】，獲得【About】對話盒，它會顯示 FreeSystemResource。一般而言你會獲得 96% 左右。這聽起來是不是太高了呢？是的，稍後當你看到 *GetFreeSystemResources* 的虛擬碼，你就會知道，USER 和 GDI 的 heaps 任何時候都不可能有 96% 的自由率。

那麼這到底是怎麼回事？讓我長話短說，Windows 95 竄改事實！它記錄的並不是各個 heaps 自由率的最小值，而是一個相對值。毫無疑問你會問：相對什麼呢？Windows 95 的 FreeSystemResource 記錄的是相對於系統啓動之後的自由率。也就是說，在系統啓動後並且 Explorer（中文名爲「檔案總管」） 也做完它該做的工作之後，系統才記錄其剩餘資源，並且當你詢問 FreeSystemResource，系統是以此記錄值爲基準來計算。

讓我再舉個例子。假設 Windows 95 啓動後，其真正（原 Windows 3.1 計算方式）的 FreeSystemResource 是 75%。然後你執行一些程式，那些 system heaps 之中只剩 50% 的自由率。這時候 Windows 95 對於 FreeSystemResource 的報告是 66%（50/75），而不是 50%。如果不是爲了企圖讓事情往正面表現，我不知道這意味著什麼。或許微軟認爲應該讓其客戶相信 Windows 95 真的消除了 FreeSystemResource 的問題。當然，Windows 95 的確以 32-bit heaps 改善了一些問題，但沒那麼多！

爲免遭受「挑微軟麻煩」的控訴，下面是我對微軟處理 FreeSystemResource 的行徑的另一種解釋。或許存在一個有著良好定義的、供系統使用的資源記憶體量。當你啓動系統，產生 desktop 或 tray 視窗，所消耗的就是其中的記憶體。由於沒有辦法要求（或開墾）這些記憶體，何必把它們顯示給系統使用者看呢？新的計算方法或許可以被視爲是「從使用者的眼光來看，比較精確的數值」-- 如果 FreeSystemResource 剩下 50%，表示他們用掉了可用資源的 50%。使用者不知道（或許不在乎）系統本身也用了一些資源。

## GetFreeSystemResources 函式

現在我們知道爲什麼新的 FreeSystemResource 會昇高了，讓我們看看 Windows 95 內部如何計算這個值。*GetFreeSystemResources* 由 16 位元 USER.EXE 提供。如果必要，SHELL32.DLL 會下移（thunk down）呼叫此函式，取其值，然後顯示在系統公用程式的【About】對話盒上。這個函式，一如我在第3章提過的許多函式，只是一個標準的參數檢驗層。檢驗完畢之後，它就跳到 *IGetFreeSystemResources* 去。

*IGetFreeSystemResources* 有三段功能迥異的碼。第一段碼計算 USER 和 GDI 的自由率。USER 自由率是取 16-bit DGROUP、32-bit window heap 以及 32-bit menu heap 的最小自由率而得，GDI 自由率則是呼叫 GDI.EXE 的 *GDIFreeResource* 函式而得。

*IGetFreeSystemResources* 的第二段碼根據系統啓動時使用掉多少 heap 空間而調整自由率的值。關鍵在於 USER.EXE 的兩個全域變數：我稱之爲 base_USER_FSR_percentage 和 base_GDI_FSR_percentage。這兩個變數最初爲 0，當 *IGetFreeSystemResources* 被呼叫

時，如果它們還是 0，函式就不調整其原先計算的 USER 和 GDI 自由率（假設其值爲Ａ）。然而如果兩個全域變數不是 0，它們內含的就是 Windows 95 啓動之後的 USER 和 GDI heaps 的自由率（假設其值爲Ｂ）。然後，*IGetFreeSystemResources* 就會把Ａ除以Ｂ，視爲 Free System Resource。

當我看到這兩個全域變數，我的第一個念頭是：誰來設定它？你相信是 Explorer 嗎？（即使你沒有看到 Explorer 視窗，Explorer 仍然是一個執行中的行程）。告訴你吧，Explorer 並不進入 USER 的 DGROUP 節區中設定 base_USER_FSR_percentage 和 base_GDI_FSR_percentange，而是讓 USER.EXE 自己去做。怎麼辦到的？當 Explorer 認爲它自己已經充份設定好，就送出一個 WM_USER 訊息給 desktop 視窗的視窗函式。稍後你就會看到，desktop 的視窗函式接受 WM_USER 訊息後設定前述兩個全域變數。如果你有一個行程或一個 DLL 搶在 desktop 視窗收到 WM_USER 之前呼叫了 *GetFreeSystemResources*，你會獲得一個完全不同的值。

*IGetFreeSystemResources* 的第三段碼將用到它所收到的參數。如果你要求的是 USER 或 GDI 的自由資源（參數指定爲 GFSR_USERRESOURCES 或 GFSR_GDIRESOURCES），函式會傳回對應的值；如果你要求的是 GFSR_SYSTEMRESOURCES，函式就傳回 USER 和 GDI 自由率中比較小的值。

**GetFreeSystemResources 虛擬碼**

```
// 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 虛擬碼

```
// 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 函式

*GetPercentFree16BitHeap* 和 *GetPercentFree32BitHeap* 是 *IGetFreeSystemResources* 的兩個輔助函式。它們都希望獲得一個參數，指定感興趣的 heap。*GetPercentFree16BitHeap* 使用未公開的 *GetHeapSpaces* 函式（***Undocumented Windows*** 第5章介紹過），它把自由空間和總體空間的比率視爲自由率。

*GetPercentFree32BitHeap* 比較複雜。它使用的是和 Windows 95 16-bit TOOLHELP 的 *Local32Info* 函式相同的一段碼。這段碼傳回你所詢問的 heap 的三個欄位：dwMemCommitted、dwTotalFree、dwMemReserved。其中 dwMemCommitted 和 dwMemReserved 幾乎總是相同。將 dwMemCommitted 減去 dwTotalFree 之後再除以 dwMemReserved，即爲自由率。由於上述這些值大小差不多，所以 *GetPercentFree32BitHeap* 傳回的通常是 98% 或 99%。

**GetPercentFree16BitHeap 虛擬碼**

```
// 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 虛擬碼**

```
// 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;
```

## 從 32 位元碼取 FreeSystemResource，跳過 Thunk Compiler

相不相信，Windows 95 並沒有提供任何容易的方法，讓 32 位元程式取得 Free System Resources（FSR）。截至標準的 Windows 95 公用程式在其【About】對話盒中顯示的 FSR 值，都是經由 SHELL32.DLL 中的 32-to-16-bit thunk 完成的。如果你寫了一個 32 位元程式，卻想要呼叫一個 16 位元系統函式（例如 *GetFreeSystemResources*），你可以花幾個鐘頭（或幾天）學習 Windows 95 的 thunk compiler，然後寫些 thunking DLLs。其實還有其他較好的作法。

一如我在討論 *SetFocus* 時一節所說的，USER32.DLL 總是下移（thunk down）到 USER.EXE，但是它並沒有任何獨立的 16- 和 32-bit DLLs 以為 thunking 之用。取而代之的是，32 位元的 *SetFocus* 使用 *QT_Thunk* 函式。你也可以在自己的程式中使用之，只不過有些技巧需要注意一下，不像使用標準的 Win32 函式那樣。這是一個未公開函式（雖然你也可以看到 thunk compiler THUNK.EXE 製造出對它的一些參考動作），你必須使用組合語言才能呼叫它。

呼叫 *QT_Thunk* 需要兩件事情。第一，你必須把欲呼叫的 16:16 位址放在 EDX 暫存器中。第二，你必須確定。呼叫 *QT_Thunk* 的這一段碼設定了 EBP stack frame，並且比你所需要的區域儲存空間多出 0x3C 個位元組。這是因為 *QT_Thunk* 為呼叫端（16 位元碼）建立了一個 convoluted stack frame，就位在你的 EBP 暫存器所指位置的下方。

為了示範如何在應用程式中呼叫 *QT_Thunk*，我寫了 FSR32 程式，它呼叫 *QT_Thunk* 取得 USER 和 GDI 的 FSR（Free System Resource）。FSR32 的原始碼就只有一個檔案，FSR32.C，很短，所以我把它全部列出來。編譯動作如下（使用 Visual C++）：

```
cl fsr32.c k32lib.lib thunk32.lib
```

或者，你也可以使用書附磁碟中的 BUILDFSR.BAT 檔案來建造這個程式。

```
#0001  //================================
```

```
#0002  // FSR32 - Matt Pietrek 1995
#0003  // FILE: FSR32.C
#0004  //================================
#0005  #define WIN32_LEAN_AND_MEAN
#0006  #include <windows.h>
#0007  #include <stdio.h>
#0008  #pragma hdrstop
#0009
#0010  typedef int (CALLBACK *GFSR_PROC)(int);
#0011
#0012  // Steal some #define's from the 16 bit WINDOWS.H
#0013  #define GFSR_GDIRESOURCES     0x0001
#0014  #define GFSR_USERRESOURCES    0x0002
#0015
#0016  // Prototype some undocumented KERNEL32 functions
#0017  HINSTANCE WINAPI LoadLibrary16( PSTR );
#0018  void WINAPI FreeLibrary16( HINSTANCE );
#0019  FARPROC WINAPI GetProcAddress16( HINSTANCE, PSTR );
#0020  void __cdecl QT_Thunk(void);
#0021
#0022  GFSR_PROC pfnFreeSystemResources = 0;   // We don't want these as
#0023  HINSTANCE hInstUser16;           // locals in main(), since
#0024  WORD user_fsr, gdi_fsr;          // QT_THUNK could trash them...
#0025
#0026  int main()
#0027  {
#0028      char buffer[0x40];
#0029
#0030      buffer[0] = 0;  // Make sure to use the local variable so that
#0031                      // the compiler sets up an EBP frame
#0032
#0033      hInstUser16 = LoadLibrary16("USER.EXE");
#0034      if ( hInstUser16 < (HINSTANCE)32 )
#0035      {
#0036          printf( "LoadLibrary16() failed!\n" );
#0037          return 1;
#0038      }
#0039
#0040      FreeLibrary16( hInstUser16 );   // Decrement the reference
count
#0041
#0042      pfnFreeSystemResources =
```

```
                                                "GetFreeSystemResources");
#0044     if ( !pfnFreeSystemResources )
#0045     {
#0046        printf( "GetProcAddress16() failed!\n" );
#0047        return 1;
#0048     }
#0049
#0050     __asm {
#0051        push    GFSR_USERRESOURCES
#0052        mov     edx, [pfnFreeSystemResources]
#0053        call    QT_Thunk
#0054        mov     [user_fsr], ax
#0055
#0056        push    GFSR_GDIRESOURCES
#0057        mov     edx, [pfnFreeSystemResources]
#0058        call    QT_Thunk
#0059        mov     [gdi_fsr], ax
#0060     }
#0061
#0062     printf( "USER FSR: %u%%  GDI FSR: %u%%\n", user_fsr, gdi_fsr );
#0063
#0064     return 0;
#0065  }
```

以上有些動作需要討論一下。首先，如何在 32 位元碼中取得 16 位元的 *GetFreeSystemResources* 函式位址呢？FSR32.C 使用三個未公開的 KERNEL32 函式 *LoadLibrary16*、*FreeLibrary16* 以及 *GetProAddress16* 完成此事。附錄A提供了一張十分完整的 KERNEL32 未公開函式列表。為了讓 FSR32 能夠成功聯結這些未公開函式，它需要 K32LIB.LIB。這個函式庫也在附錄A中討論。

為了確定堆疊上有足夠的空間給 *QT_Thunk* 使用，FSR32.C 宣告了一個 0x40 字元的區域性陣列，但完全不使用它。*QT_Thunk* 碼可以使用這段記憶體而不會引爆什麼災難。FSR32 中任何要的變數都必須宣告為全域變數，才不會被 *QT_Thunk* 破壞掉（我可是花了很大代價才學會了這一課唷）。

FSR32.C 使用行內(inline)組合語言碼來呼叫 *QT_Thunk*。FSR32 不使用一般的 C 呼叫，是因為 EDX 在呼叫之前必須被設定為 16:16 位址。理論上你可以在呼叫 *QT_Thunk* 之前使用一行組合語言碼設定 EDX 的值就好。然而你必須確定編譯器不會在 CALL 指令執行之前改變 EDX 的值。

最後注意一點，這些碼並沒有做任何難以處理的動作，像是把指標交給 16 位元碼等等。不是 Win32 API 函式下移(thunk down)到 16 位元碼，並且把指標交給 16-bit DLLs 時，都必須竭盡心力去做諸如設定 alias selectors 之類的事情。如果你需要處理這些事情，我勸你還是使用 thunk compiler 吧，那才是正途。上述例子只傳遞一個參數，而且該參數也不需要為了 16 位元碼做任何轉換。需要做轉換的參數包括有指標、視窗訊息等等。簡單地說，在你決定直接使用 Windows 95 的 thunks 之前，先慎重考慮一下 thunk compiler。

## 視窗系統的 16/32 位元混合性質

早先我曾經說過，WND 結構儲存在 32-bit heaps 中，其偏移位置（相對於 USER's DGROUP）因此一定大於 64K。我也說過 HWNDs 受限爲 16 位元，所以在 16-bit DGROUP 和 window heap 之間有一塊區域用來做爲 handle table，將 HWNDs 轉換爲有用的指標，指向 WND 結構。

這時候，視窗系統中的這種「16-/32- 位元程式碼與資料」雙重形式的處理就十分重要。首先要釐清的一個觀念是，16 位元 HWND 值在整個系統中都適用。不論是在 Win16 程式中或 Win32 程式中，HWND 就是一個 16 位元值，被當做是 window heap 的 handle table 的索引。讓我再說一次，一個 HWND 就是一個 HWND。怎麼說它就是一個 HWND，不管是在 Win16 或 Win32 之中。HWND 是 16 位元值，但它並不像在 Windows 3.1 中那樣只是代表 USER's DGROUP 中的偏移位置而已。

現在你知道 HWNDs 在任何地方都是 16-bit handles 了，讓我告訴你內部情況：USER.EXE 通常把 HWNDs 轉換爲 32 位元指標。這些 32 位元指標是相對指標（相對

於 USER's DGROUP），不是 flat 指標。USER 使用這些 32 位元特殊指標的一個絕佳例子就是在 WND 結構之中，其最前面的四個欄位分別是視窗的 parent、owner、child 和 sibling 視窗。在這四個欄位中，USER 儲存了 32 位元指標（而不是 16 位元 HWND 值）指向適當的 parent、owner、child 和 sibling 視窗。這很可能是爲了效率考量，因爲 USER 無論如何必須在視窗階層架構來來回回移動時，把 HWNDs 轉換爲指標。稍後我將回到視窗的階層架構來。

當然，雖然 USER 可以在內部使用 32 位元指標指向 WND 結構，它還是必須在對外介面上使用 16 位元 HWNDs。因此，必須有一個很快速很容易的方法，在 16 位元 HWND 和 32 位元指標之間做轉換。的確是有這麼一個方法。稍後當我們觀察某些視窗相關函式的虛擬碼時（16 和 32 位元都有），你就會看到。

在同一系統中支援 Win16 和 Win32 程式的另一個棘手問題就是，視窗函式的差異性。Win16 程式有一個典型的視窗函式如下：

```
WndProc16(  unsigned short hWnd,
            unsigned short wMsg,
            unsigned short wParam,
            unsigned long  lParam );
```

而 Win32 程式的典型視窗函式如下：

```
WndProc32(  unsigned long hWnd,
            unsigned long wMsg,
            unsigned long wParam,
            unsigned long lParam );
```

問題是，當一個 Win32 程式呼叫 *SendMessage*，傳送訊息給一個 Win16 程式中的視窗，會有什麼結果？很明顯，這一定會造成問題，除非參數被重新安排或被截斷。另一方面，如果 Win16 程式呼叫 *SendMessage*，傳送訊息給一個 Win32 程式，大部份壓到堆疊中的參數（hWnd、wMsg、wParam）都必須放寬。程式不可能做這些細部工作，所以責任就落到 USER.EXE 身上。

另一個相關問題就是視窗的 subclassing。Subclassing 的基本觀念是利用 *GetWindowLong*(GWL_WNDPROC) 取得某個視窗的 WNDPROC callback 位址，把該位

址儲存下來，然後再利用 *SetWindowLong*(GWL_WNDPROC) 改變視窗的 WNDPROC 位址爲某個 subclass 函式。如今問題來了：32 位元程式的視窗函式位址是 32 位元線性位址，如果一個 16 位元程式要將它改變爲 16:16 位址，很顯然是一個困難的局面。32 位元碼呼叫 WNDPROC 時，預期的是一個 flat 線性位址，以一個 16:16 位址取而代之當然不可能正常工作。

爲了避免這些問題，USER.EXE 爲每一個它所產生的 32 位元 WNDPROC 產生一小塊碼，這是一塊 16 位元碼，內含 Win32 程式真正用來做爲其 WNDPROC 的 32 位元線性位址。下面就是 Explorer 的 tray 視窗的那一小塊碼：

```
:u 1457:140
1457:00000140   PUSH    00401DFA        ; A 32-bit WNDPROC address
1457:00000146   PUSH    00030000
1457:0000014C   JMP     0127:7555
```

再之後則是另一小塊碼：

```
:u 1457:156
1457:00000156   PUSH    0040180D        ; A 32-bit WNDPROC address
1457:0000015C   PUSH    00030000
1457:00000162   JMP     0127:7555
```

你可以想像，位址 0127:7555 是某種 thunk routine（於 KRNL386.EXE 中），把 Win16 WNDPROC 的參數轉換爲 Win32 WNDPROC 期望的參數，然後呼叫 thunk 中指定的位址。這些 thunks 所駐留的節區，都是 USER.EXE 配置的 global heap，而 code segment alias selector（0x1457）用來指向它。

到底這整個意味著什麼呢？如果你觀察任何視窗的 WND 結構，你總會發現其 WNDPROC 的 16:16 位址。根據其記憶體內容，你可以判斷這是一個正規的 Win16 WNDPROC 位址，還是一個用來上移（thunk up）至 Win32 WNDPROC 的位址。於是 *GetWindowLong*(GWL_WNDPROC) 就有兩條路可以走：它必須視呼叫動作來自 Win16 或 Win32 程式，然後才能夠決定適當的位址。

## 訊息傳遞系統的改變

Windows 95 的 USER 子系統中，與先前版本比較，最戲劇性的變化，就是視窗訊息的傳遞方式。我把 post、send、以及處理訊息的機制稱爲訊息傳遞系統（messaging system）。Windows 95 訊息傳遞系統的最好消息就是它消除了對 Win32 程式傳遞訊息時的同步（synchronous）性質。在 16 位元 Windows 中，一次只能有一個 task 執行，這個 task 必須明白呼叫訊息系統中的某個 API。一般而言一個 task 要釋放控制時可以在其主迴路中呼叫 *GetMessage* 或 *PeekMessage*。*SendMessage* 也可以釋放控制權。

問題在於，一個不能夠規律釋放控制權的程式會妨礙其他程式執行。而這將造成輸入系統的凍結。如果 Win16 系統沒有呼叫 *GetMessage* 或 *PeekMessage*，就沒有其他人能夠執行。於是一個長期佔有 CPU 的程式將使系統不具有多工能力。

Windows NT 小組把 USER 模組重新設計過，使得控制權的釋放以及排程的進行不受程式是否呼叫 *GetMessage* 或 *PeekMessage* 的影響。一個 Win32 可以取得 CPU 時間，處理訊息，不會對其他程式造成不利影響。Windows NT 達成了這個目標之後，Windows 95 更是義無反顧，不容許走回頭路了。

當然，如果 Win16 繼續在 Windows 95 中正常執行，訊息系統的改變就沒辦法適用到 Win16 程式上。太多 Win16 程式依賴合作型多工模式，在其中，程式不會釋放控制權，直到它準備好了爲止。因此，只有 Win32 程式才能夠完全以自己的步調處理訊息而不影響系統的其他部份。

Windows 95 產生這樣的雙模式行爲（Win16 採用合作型多工而 Win32 採用強制性多工）是經由 Win16Mutex 和執行緒排程獲得的。任何時候，Windows 95 排程器把最高優先權且準備好要執行的執行緒，推爲第一執行順位。有一件事情可以使一個執行緒處於「尚未準備好要執行」的狀態：當它等待一個 mutex semaphore（亦即 Win16Mutex）時。

只要一個 Win16 程式執行起來，它就獲得 Win16Mutex（更精確應該這樣說：當任何 16 位元碼執行起來，Win16Mutex 就被擁有）。只要有 16 位元碼執行起來，舊日規則如

呼叫 *GetMessage* 或 *PeekMessage* 就仍然適用，以便其他 16 位元程式也有機會執行。然而，正因爲執行緒排程器不會因爲一個 Win16 task 握有 Win16Mutex 就不把執行權從其身上轉移，所以即使 Win16 task 沒有適時地唧入訊息，至少 Win32 執行緒仍然能夠繼續執行。至於其他的 Win16 程式當然是被阻塞住啦。

現在，出現了一個問題。由於訊息系統位於 16 位元 USER.EXE 內，一個使用訊息迴路的 32 位元程式一定會在它下移（thunk down）到 Win16 USER.EXE 之前獲得 Win16Mutex。因此，強制性多工變成是個半調子。如果執行緒的內容純粹是計算，無關使用者介面（也就不必下移至 16 位元 USER.EXE）,那麼執行緒就不受任何壞份子（不釋放控制權的那些程式）的影響。然而如果一個 Win32 執行緒必須下移至 USER 或 GDI 或其他 16 位元模組，它必須索求 Win16Mutex，而執行緒會因此被堵住，直到獲得 Win16Mutex 爲止。因此，不良行爲的 Win16 程式還是會影響其他 Win32 程式的執行（如果那些程式也使用訊息系統或相關機能的話）。

我們在 Windows 95 中到底擁有什麼呢？是的，我們擁有的是強制性多工，但是有個死角。這個死角就是「不及時唧取訊息」之 Win16 程式。你沒有辦法擺脫 Win16Mutex，但是你可以想辦法儘量減少 Win16 程式。只要減少花費在等待 Win16Mutex 的時間上，你就可以減少不良的 Win16 程式對系統造成的不良影響。

微軟有一個設計上的改善，就是加了所謂的 Raw Input Thread（RIT）。在所有微軟用來描述訊息進入系統的圖片中，都顯示硬體中斷處理常式把訊息安置到一個中央佇列去。有一個執行緒（也就是 RIT）不斷監視這個佇列，取出訊息，分配到適當的執行緒訊息佇列去。下一節我會比較詳細地介紹訊息佇列。

雖然 RIT 聽起來很不錯，而且恐怕 Windows NT 的確是這般行爲，但我卻沒有辦法確定它存在於 Windows 95。我發現 KERNEL32.DLL 有一個名爲 *DispatchRITInput* 的函式。然而當我在上面設立中斷點（breakpoint）並且在它被觸發時檢查當時的執行緒，我發現它並不是被單一執行緒呼叫。*DispatchRITInput* 最終下移（thunk down）至 16 位元 USER.EXE 的 *DispatchInput* 函式。我嘗試在這裡設立中斷點，雖然它不時被觸發，它

還是被各個不同的執行緒呼叫。我嘗試在其他的 USER 訊息系統內部函式中進行類似的實驗，卻沒辦法發現一個只被單一執行緒呼叫的函式。於是我承認失敗並寫信給 Windows 95 小組成員，詢問 RIT。他們這麼說：

> 的確有一個 RIT。不過如果我們能夠在某些隨機執行緒中處理一般事務，我們真的會那麼做，而不將 RIT 納入排程考慮。爲的是速度和效率。這就是爲什麼你看到 *DispatchInput* 被各式各樣執行緒呼叫的原因。我們把 RIT 視爲最後的手段。

## 未被公開證實的 Windows 95 和 Win16Mutex 的問題

在 ***Unauthorized Windows 95*** 一書第 552 頁，作者 Schulman 引用了一些我在 ***PC Magazine*** 和 ***Microsoft Systems Journal*** 中發表的文章，其中關於 Win16Mutex 有些內容是不正確的。他引用了這一段我發表於 ***PC Magazine*** 的話：

愈快移往 32 位元愈好。如果系統之中沒有任何 16 位元程式，Win16Mutex 就不會成為困擾的來源...

然後他繼續說道：『一個 Windows 95 系統（至少在 beta1）總是有兩個 Win16 tasks 會執行起來。一個是 TIMER，一個是 MSGSRV32。』

Schulman 也引用了我在 ***Microsoft Systems Journal*** 上的一段話：

USER 和 GDI 將迅速執行然後釋放 Win16Mutex。沒有任何 32 位元執行緒能夠會堅持有 Win16Mutex 一段時間。

然後他展示一個小小的 Win32 程式（W16LOCK），抓住 Win16Mutex 不放，維持了一段時間。

這兩點都是有確實根據的，而我要再做一些申述。第一點（Windows 95 系統中總是至少有兩個 Win16 tasks 在跑）已經有了輕微的改變。最近版本中，Windows 95 對

實只用到一個 Win16 task：MSGSRV32。或許你的系統中會有一個 MMTASK task 在跑，不過那非必要。你可以把它關閉，於系統毫髮無損。MSGSRV32 允許你從 DOS 視窗中啟動程式。為了觀察 MSGSRV32 是否造成問題，我使用 SoftIce/W 並在 KRNL386.EXE 的 CurTDB 變數上設定一個硬體寫入中斷。這個中斷點很迷人－－我被限制資格，只有當 MSGSRV32 的 HTASK 被寫進該變數時才會響起。於是只要 MSGSRV32（系統中唯一的 Win16 task）變成 active task，中斷點就會觸發。在大多數情況下，我唯一能夠讓 MSGSRV32 成為 active task 的機會就是在 DOS 提示號下啟動程式。當然它也偶而會自己地會在其他時間成為 current task。

觀察 MSGSRV32 程式碼，我並沒有注意到任何東西可以顯示 MSGSRV32 企圖凍住系統（因為未及時處理訊息）。唯一一個比較接近的情況是當 MSGSRV32 以 *WinExec* 啟動另一個程式。在 *WinExec* 執行期間，Win16Mutex 會被抓住不放。是的，你不可能完全消除 16 位元程式，而 MSGSRV32 看起來頗還值得信賴，不會無端持有 Win16Mutex 太長時間。

對於第二點（W16LOCK 程式從一個 Win32 程式中取得 Win16Mutex 並佔有相當一段時間），我的感覺是，W16LOCK 是一種乖張的情況。是的，它的確是在「Windows 95 允許取用系統函式和時步化物件」的路上揭露了一個大洞，但是 W16LOCK 必須很明顯地從一個 Win32 程式中搶過 Win16Mutex 並持有它，而非因為 Win32 程式怠情或不在乎以致於成功。如果它們下移（thunk down）到 16-bit DLLs，那就是另一回事了。我承認，Win16Mutex 可能是個瓶頸，可能造成系統停滯。如果你放棄所有 non-system Win16 程式，而且沒有故意找系統的碴兒，或許你根本不會注意到 Win16Mutex 的影響。換句話說，認識 Win16Mutex，但不要因為它而睡不安穩！

## 每個執行緒都有一個訊息佇列

Windows 95 的每一個執行緒都可以有它自己的訊息佇列。總括來說，訊息佇列是一個結構，控制著執行緒呼叫 *GetMessage* 或 *PeekMessage* 時究竟獲得哪一個訊息。在

Windows 3.1 以及更早之前，每一個 Win16 程式有自己的訊息佇列，在程式啓動之時產生。Windows 95 的每一個執行緒都可以有自己的訊息佇列，但只有當執行緒真正需要一個時才會產生。由於每一個 Win16 程式在 Windows 95 中有一個執行緒，所以每一個 Win16 程式還是擁有單一一個訊息佇列。

讓我們更密切地看看訊息佇列，因爲它們是 USER 模組中的一個主要的資料結構。當執行緒呼叫 *GetMessage* 或 *PeekMessage*，它會檢查所屬訊息佇列。「執行緒所擁有之訊息佇列」標示在 *GetMessage* 和 *PeekMessage* 碼之中。你不可能檢查到另一個執行緒的訊息佇列。訊息佇列也在「送訊息到另一個程式」時發揮部份功能。從 USER 的角度來看，*SendMessage* 就是從一個訊息佇列轉移到另一個訊息佇列。

我並不打算深入 *GetMessage*、*PeekMessage*、或 *SendMessage* 的所有細節，我把那些細節放在 ***Windows Internals*** 一書。雖然從 Windows 3.1 到 Windows 95 有一些改變，但我不認爲在此重複那些知識有什麼好處。我只想闡述 Windows 3.1 到 Windows 95 之間的改變。

## 訊息佇列的格式

一開始讓我們看看訊息佇列的格式。每一個訊息佇列位在一個來自 16-bit USER.EXE global heap 的節區之中。每一個 thread database（第 3 章）以及 task database（第 7 章）都內含一個 selector，指向對應的訊息佇列。訊息佇列的欄位被我定義在 SHOWWND 程式的 MSGQUEUE.H 檔案中，細節如下（以下每一筆資料分別是偏移位置、資料型態、欄位名稱）：

**00h WORD nextQueue**

指向下一個訊息佇列。系統中所有的訊息佇列都被串爲一個串列（linked list）。串列中的最後一個訊息佇列的此一欄位爲 0。

**02h WORD hTask**

內含對應之 HTASK。我將在第7章介紹，每一個 Win32 程式都擁有一個 16-bit task database。

**04h WORD headMsg**

一個近程指標（相對於 USER's DGROUP），指向 QUEUEMSGs 串列的頭。QUEUEMSGs 將於下一節描述。

**06h WORD tailMsg**

一個近程指標（相對於 USER's DGROUP），指向 QUEUEMSGs 串列的尾。

**08h WORD lastMsg**

一個近程指標（相對於 USER's DGROUP），指向一個 QUEUEMSG，它正是程式呼叫 *GetMessage* 或 *PeekMessage* 時即將取得的訊息。

**0Ah WORD cMsgs**

訊息佇列中等待被處理的訊息個數。也就是 headMsg（04h）所指之 QUEUEMSGs 串列中的元素個數。

**0Dh BYTE sig[3]**

對於 Win32 執行緒的訊息佇列，這三個位元組內放的是 "MJT" 的 ASCII 碼。那或許是 Jon Thomason（一位微軟程式員）的縮寫。對於 Win16 程式的訊息佇列，這裡面放的都是 0。

**10h WORD npPerQueue**

一個近程指標（相對於 USER's DGROUP），指向 PERQUEUEDATA 結構。該結構中持有每一執行緒的 active 視窗、focus 視窗、capture 視窗。我將在下一節描述這樣的觀念。

### 16h WORD npProcess

一個近程指標（相對於 USER's DGROUP），指向 QUEUEPROCESSDATA 結構。如果一個行程有許多執行緒和佇列，所有佇列中的這個欄位就都指向同一個 QUEUEPROCESSDATA 結構。這個結構內含的資訊（像是與此佇列相關聯的 process ID），稍後我會描述。

### 24h DWORD messageTime

如果程式呼叫 *GetMessageTime*，將獲得此欄位內容。這也就是訊息被 posted 的時刻。這個值是在訊息被 *GetMessage*/*PeekMessage* 抓走時，從 QUEUEMSG 結構中拷貝而來的。

### 28h DWORD messagePos

如果程式呼叫 *GetMessagePos*，將獲得此欄位內容。這也就是訊息產生時滑鼠游標的 x,y 位置。這個值是在訊息被 *GetMessage* / *PeekMessage* 抓走時，從 QUEUEMSG 結構中拷貝而來的。

### 2Eh WORD lastMsg2

一個近程指標（相對於 USER's DGROUP），指向最新將被取走的 QUEUEMSG 結構。

### 30h DWORD extraInfo

如果程式呼叫 *GetMessageExtraInfo*，將獲得這個欄位內容。這個值是在訊息被 *GetMessage* / *PeekMessage* 抓走時，從 QUEUEMSG 結構中拷貝而來的。

### 3Ch DWORD threadId

與此佇列相關的執行緒的 thread ID。thread ID 和 thread database 的關係在第 3 章說過。

### 42h WORD expWinVer

這是程式所期望的 Windows 版本。通常爲 0x300、0x30A 或 0x400，分別代表 Windows 3.0、3.1 或 4.0。這個值是在程式載入之初從其可執行檔表頭中獲得。USER 有時候會

根據這個值決定訊息應該如何處理，或是哪個訊息應該傳遞。換句話說它允許 USER 在眾多 Windows 版本之間有相容行爲。

### 48h WORD ChangeBits

這個欄位由許多 QS_XXX 旗標值組成，用來表達自從前一次 *GetQueueStatus* 之後的所發生的各種訊息型態。下面是定義於 WINUSER.H 中的 QS_ 常數：

```
QS_KEY          0x0001
QS_MOUSEMOVE    0x0002
QS_MOUSEBUTTON  0x0004
QS_POSTMESSAGE  0x0008
QS_TIMER        0x0010
QS_PAINT        0x0020
QS_SENDMESSAGE  0x0040
QS_HOTKEY       0x0080
```

*GetQueueStatus* 會傳回這個欄位的值，放在 DWORD 的較低字組( low word )中。***Windows Internals*** 第7章有許多關於 QS_xxx 旗標意義的解釋。

### 4Ah WORD WakeBits

這個欄位由許多 QS_XXX 旗標值組成，用來表達佇列中的各種訊息型態。QS_XXX 的意義已在前一欄位描述過。*GetQueueStatus* 會傳回這個欄位的值，放在 DWORD 的較高字組（high word）中。

### 4Ch WORD WakeMask

如果執行緒在呼叫 *GetMessage* 或 *PeekMessage* 時被阻塞住，這個 WORD 就放置著 QS_XXX 旗標，指出它正等待的訊息的型態。一般而言，程式會在呼叫 *GetMessage* 時被阻塞住，所以這個欄位將是 QS_ALLINPUT，那是所有 QS_XXX 的組合。

### 50h WORD hQueueSend

如果這個執行緒正處理的訊息是被其他執行緒 "send" 過來的，這個 WORD 將放置傳輸端執行緒的訊息佇列的 handle。

#### 56h WORD sig2

這個欄位內放 0x5148，那是 "HQ"（Handle Queue？）的 ASCII 碼。每一個訊息佇列都關聯一個特定的執行緒，每一個執行緒又關聯一個行程。因此，在訊息佇列和行程之間可能存在有「多對一」的關係。訊息系統之中，如果某些資訊要讓行程的所有佇列共享，它將被存放在一個我所謂的 QUEUEPROCESSDATA 結構中。這個結構放在一塊由 16-bit USER heap 配置而來的空間中。訊息佇列的 0x16 偏移處儲存有此一結構的指標。在 Windows 95 的 16-bit TOOLHELP.H 檔案中，這樣的資料結構被標示以 LT_USER_PROCESS（0x1D）。只有 USER.EXE 除錯版才會對此區塊貼上這樣的標籤。

QUEUEPROCESSDATA 結構的欄位被記錄在 SHOWWND 程式的 MSGQUEUE.H 檔案中。細節如下（每一筆資料分別是偏移位置、資料型態、欄位名稱）：

#### 00h WORD npNext

一個近程指標（相對於 USER's DGROUP），指向下一個 QUEUEPROCESSDATA 結構。

#### 02h WORD un2

這個指標所指之資料意義未明。但在 Windows 除錯版中，該資料有著 LT_USER_SUBSYSTEM 的型別。

#### 04h WORD flags

某種旗標值，意義未明。

#### 08h DWORD processId

與此佇列有關聯的 process ID。

#### 0Eh WORD hQueue

此欄位儲存著 hQueue。其真實意義未明。

## QUEUEMSG 結構

在 Windows 3.1 及更早版本，所謂訊息佇列內含的就是訊息。基本上訊息佇列是一個大型的 MSG 結構陣列。在佇列結構中，前兩個 WORD 扮演頭尾指標。由於訊息是儲存在陣列之中，所以有容量限制。預設是 8 個訊息，但你可以呼叫 *SetMessageQueue* 改變其大小。

Windows 95 徹底改變了訊息的儲存方式。在 Windows 95 的訊息佇列中，有一個近程指標指向某一結構串列的頭。每一個結構代表一個訊息。我給這些結構取了個名字：QUEUEMSG。QUEUEMSG 係從 16 位元 USER's DGROUP 中配置而來。這真令人驚訝，因爲 Windows 95 的許多改善就是要把資料從 USER's DGROUP 中移走。所以這樣的安排似乎有點「反動」。附帶一提，Windows 95 的 16 位元 TOOLHELP.H 把這些結構叫做 LT_USER_QMSG（0x1A）。

如果你很難相信佇列中的訊息已不再是存在於陣列之中，請看看 Windows 95 的 *SetMessageQueue* 函式碼：

```
SETMESSAGEQUEUE proc
C0ED:  XOR  AX,AX
C0EF:  INC  AX
C0F0:  RETF 3702
```

對於那些沒辦法讀組合語言的人，我告訴你，這個函式傳回 1。那是因爲佇列之中不再有 MSG 結構陣列。但 Windows 3.1 就不同，其 *SetMessageQueue* 計算新佇列有多大（容納多少訊息），並爲它配置新的 global heap 空間。

QUEUEMSG 結構的佈局在 SHOWWND 程式的 MSGQUEUE.H 中以 C 語言形式呈現出來。各欄位細節如下（每一筆資料分別是偏移位置、資料型態、欄位名稱）：

**00h WORD hWnd**

訊息將被派往的視窗。

#### 02h WORD msg

訊息號碼。只有底部 16 個位元會被儲存起來。這在 Win16 不是問題，因爲它的訊息本來就是 16 位元。但在 Win32 中訊息號碼是個 DWORD，所以較高的 WORD 會遺失掉。

#### 04h WORD wParamLow

對 Win16 程式而言，此欄位內含訊息的 WPARAM。對 Win32 程式而言，此欄位內含訊息的 WPARAM 的較低字組。

#### 06h DWORD lParam

此欄位內含訊息的 LPARAM。

#### 0Ah DWORD messageTime

訊息被放置到佇列時的時刻。根據 SDK 文件上說，訊息時刻是從系統啓動之後以毫秒（millisecond）計算。*GetMessageTime* 可以獲得此欄位的值。*GetMessage* 和 *PeekMessage* 會把此值拷貝到訊息佇列的 24h 偏移位置處，而事實上 *GetMessageTime* 是從該處取值。

#### 0Eh DWORD messagePos

此欄位內含訊息產生時滑鼠游標的 x,y 位置。*GetMessagePos* 可以取出此值。*GetMessage* 和 *PeekMessage* 會把此值拷貝到訊息佇列的 28h 偏移位置處，而事實上 *GetMessagePos* 是從該處取值。

#### 12h WORD wParamHigh

對 Win32 程式而言，此欄位內含訊息的 WPARAM 的較高字組。對於 Win16 程式，此欄位將被忽略。

#### 14h DWORD extraInfo

此欄位內含訊息的額外資訊（不一定會有）。*GetMessageExtraInfo* 可以取出此值。*GetMessage* 和 *PeekMessage* 會把此值拷貝到訊息佇列的 30h 偏移位置處，而事實上 *GetMessageExtraInfo* 是從該處取值。

#### 18h WORD nextQueueMsg

一個近程指標（相對於 USER's DGROUP），指向串列中的下一個 QUEUEMSG 結構。此值若爲 0，代表串列結束。

## 每一個訊息佇列都有的「系統視窗」

Windows NT 信奉的一個設計理念是，一個行程不應該有能力對另一個行程造成不利影響（至少不應該在未獲同意的情況下）。Windows 3.1 及其更早版本未能遵循此一哲學，特別是在視窗系統上面。在 Windows 3.1 中，整個系統只有一個 active 視窗，一個 focus 視窗，一個 capture 視窗。任何程式都可以從另一個程式手上偷走 focus，只要呼叫 *SetFocus* 就行。呼叫 *SetActiveWindow* 則可以改變 active 視窗。

Windows NT 解決之道是，每個程式有自己的一個 active、focus、capture HWNDs （我這麼說實在有點粗糙，但目前足夠了）。讓每一個程式擁有這些「系統狀態視窗」，Win32 程式就不必擔心其他程式有意或無意地影響自己的行爲。這樣的觀念如此美好，於是 Windows 95 沿用了它。

「每個佇列都擁有的資訊」係保持在一個從 USER's DGROUP 中配置而來的結構。嘿，我曾說過 Windows 95 把某些東西移出 USER's DGROUP，現在卻加入了一些新東西。訊息佇列的 0x10 偏移位置處存放著一個指標，指向這個結構。我把這個結構命名爲 PERQUEUEDATA。Windows 95 的 16-bit TOOLHELP.H 把此結構稱爲 LT_USER_VWININFO（type ID = 0x1B）。

順帶一提，當 Windows 95 進入最後測試版（M8）時，1995.03.27 的 ***InfoWorld*** 雜誌曾以頭條消息發佈這則消息： "Win95 beta lays an egg"。這篇文章是後續許多辯論的濫觴，這些辯論最終都和 PERQUEUEDATA 結構有關。在 Windows 95 的最後測試版中，PERQUEUEDATA 長達數百位元組。這導至稍微執行多一點執行緒就造成 USER 64K DGROUP 的大暴滿。最後，微軟減少了相當量的 PERQUEUEDATA 長度，辯論才得以平息。

SHOWWND 程式的 MSGQUEUE.H 中定義有 PERQUEUEDATA 的 C 語言定義。各欄位細節如下（每一筆資料所呈現的分別是偏移位置、資料型態、欄位名稱）：

#### 00h WORD npNext

近程指標（相對於 USER's DGROUP），指向下一個 PEQUEUEDATA 結構。很明顯地，PEQUEUEDATA 結構形成一個串列。

#### 06h WORD npQMsg

近程指標（相對於 USER's DGROUP），指向一個 QUEUEMSG 結構。QUEUEMSG 結構曾描述於前一節。

#### 14h WORD somehQueue1

這是一個訊息佇列的 handle。其真實意義目前未知。

#### 16h WORD somehQueue2

這是另一個訊息佇列的 handle。其真實意義目前未知。

#### 18h DWORD hWndCapture

32 位元指標（相對於 USER's DGROUP），指向目前擁有 capture 的視窗。

#### 1Ch　DWORD　hWndFocus

32 位元指標（相對於 USER's DGROUP），指向目前擁有 focus 的視窗。

#### 20h　DWORD　hWndActive

32 位元指標（相對於 USER's DGROUP），指向目前 active 的視窗。

## 改變 Windows 95 中的 WND 結構

WND 結構或許是 Windows 95 中最被使用的系統資料結構。系統中的每一個視窗（不論可見或不可見），都有一個對應的 WND 結構。Windows 3.1 的 HWND 是一個近程指標，指向 USER's DGROUP 中的 WND 結構。至於 Windows 95 的 HWND，一如稍早我所說，是一個偏移位置，指出一個由「32 位元 USER32 相對指標」所組成的陣列，每一個相對指標都指向一個 WND 結構。

由於每一個 WND 結構都內含一個指標，指向其父視窗、其兄弟視窗、及其第一個子視窗，你可以輕易看出視窗其實是被維護成一個階層架構。**圖 4-3** 顯示視窗樹狀階層架構並對每一層有一些說明。最根源是 desktop 視窗，其下第一層視窗的屬性必是 WS_OVERLAPPED 或 WS_POPUP，這也就是程式員一般所謂的「最上層」視窗或「主」視窗。較低層視窗的屬性是 WS_CHILD，典型的例子是對話盒中的控制元件（例如一個按鈕）。由於視窗是以階層架構來管理，你可以從 desktop 開始，漫游系統中的所有視窗。SHOWWND 程式就是這麼做（稍後詳述）。

雖然這一事實並不常常爲人所知，但視窗的 z-order（前後覆疊關係）事實上是藉由它們在階層架構中的相關位置而決定。在一群兄弟視窗中，串列中的第一個視窗就是 z-order 最高者，第二個視窗則有次高的 z-order。例如，所有最上層視窗（WS_OVERLAPPED 和 WS_POPUP）彼此都是兄弟視窗，也都是 desktop 的子視窗。第一個子視窗（也就是第一個 WS_OVERLAPPED 或 WS_POPUP 視窗）擁有最高的 z-order。

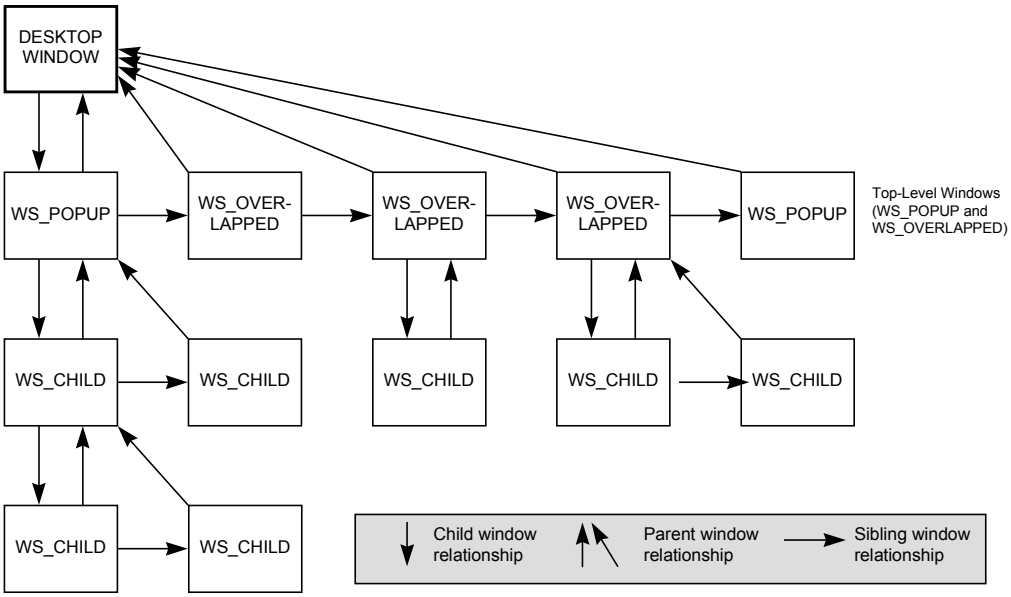



**圖 4-3　WND 結構的樹狀階層架構，使你能夠從 desktop 出發，巡訪系統中的每一個視窗。**

## WND 結構的細節

雖然 Windows 95 的 WND 結構定義相較於 Windows 3.1 有一些變化，但並不是十分劇烈。欄位排列並沒有改變（雖然欄位的大小改變了）。新增了一些欄位，其中最主要的是一個 WORD，內含一個 16 位元 HWND，代表視窗本身。這個 WORD 使視窗得以輕易被 16-bit HWND 或是 32 bit USER's DGROUP 的相對指標參考到。

SHOWWND 程式的 HWND32.H 有一個關於 Windows 95 WND 結構的 C 語言定義。下面是其詳細內容（每一筆資料所呈現的分別是偏移位置、資料型態、欄位名稱）：

### 00h　struct _WND32 *hWndNext

32 位元指標（相對於 USER's DGROUP），指向本視窗之兄弟視窗。此兄弟視窗是樹狀階層架構中的下一個視窗，並且與本視窗共有同一父視窗。呼叫 *GetWindow* 並指定 GW_HWNDNEXT，即可獲得此一兄弟視窗的 16 位元 HWND。

#### 04h struct _WND32 *hWndChild

32 位元指標（相對於 USER's DGROUP），指向本視窗之第一個子視窗。呼叫 *GetWindow* 並指定 GW_CHILD，即可獲得此一子視窗的 16 位元 HWND。若再根據後者呼叫 *GetWindow* 並指定 GW_HWNDNEXT，就可以依序獲得每一個子視窗。

#### 08h struct _WND32 *hWndParent

32 位元指標（相對於 USER's DGROUP），指向本視窗之父視窗。呼叫 *GetParent*，即可獲得此父視窗的 16 位元 HWND。唯一一個沒有父視窗的視窗就是 desktop。

#### 0Ch struct _WND32 *hWndOwner

32 位元指標（相對於 USER's DGROUP），指向本視窗之 owner 視窗。owner 視窗是接收 notification 訊息（例如 BN_CLICKED）之視窗。對於 WS_OVERLAPPED 和 WS_POPUP 視窗而言，其 owner 視窗和父視窗不一定相同 -- 雖然它們常常是。對於 WS_CHILD 視窗，父視窗就是 owner 視窗（也就是說它接收所有的 notification 訊息）。呼叫 *GetWindow*(GW_OWNER）可以獲得一個 16-bit HWND，代表該視窗的 owner 視窗。

#### 10h RECTS rectWindow

16 位元 RECT 結構（4個短整數），用來定義視窗邊界，包括非客戶區（nonclient area）。

#### 18h RECTS rectClient

16 位元 RECT 結構（4個短整數），用來定義視窗客戶區的邊界。所謂視窗客戶區是視窗的部份面積，應用程式可以在其上利用一個 device context（以 *BeginPaint* 或 *GetDC* 獲得）作畫。

#### 20h WORD hQueue

這是一個 16-bit glohal heap handle，代表訊息佇列。這個欄位的存在，證明在 Win32 之中，視窗被限制在單一訊息佇列之內活動，也因此被限制在單一執行緒內活動。也因此，Win32 有一個 *GetWindowThreadProcessId* 函式。

### 22h WORD hrgnUpdate

如果視窗有部份需要重繪，這個欄位就內含一個 HRGN，用以描述需要重繪的區域。Regions 是 GDI 資料結構，儲存在 Windows 95 的另一個 32-bit heap 中。

### 24h WORD wndClass

一個近程指標（相對於 USER's DGROUP），指向 USER_DGROUP_WNDCLASS 結構。這樣的一個結構其實是 USER 常常取用的「視窗與類別相關資訊」的最小量資訊。較不常被取用的類別資訊則存放在另一個結構中（也是位於 32-bit heap）-- 我將在「Windows 95 視窗類別的改變」一節中再來談論它。總的來說，WND 結構中的這個欄位指定了視窗類別型態。

### 26h WORD hInstance

大部份情況下，這個 WORD 內含產生此一視窗之程式的 16-bit hInstance（指向 DGROUP）。然而，對於像 edit control 那樣需要維護一個很大緩衝區（接近 64K）者而言，這裡放的是將被其 WNDPROC 使用的 DS 值。呼叫一個視窗函式之前，USER 會將此欄位內容載入到 AX 之中，在某些 Win16 輸出函式（exported function）的前置碼（prologues）中，它們期望 AX 內含的是將用於 DS 暫存器的值。一般而言，程式將使用的 DS 是指 DGROUP 節區，但是像 edit control 那種特殊情況，就會使用另一個節區。

### 28h WNDPROC lpfnWndProc

內含與此視窗有關之視窗函式位址。它似乎總是一個 16:16 遠程位址。如果視窗所宣告的函式是 Win32 函式，這個欄位將內含一個指標，指向一小段程式碼，用來上移至 Win32 視窗函式，

### 2Ch DWORD dwFlags

內含一堆旗標值，用以表示視窗內部狀態。其個別位元之定義並未公開。

#### 30h DWORD dwStyleFlags

內含一堆 WS_XXX 旗標值，用以表示視窗風格。旗標值定義於 16 位元系統的 WINDOWS.H 和 32 位元系統的 WINUSER.H 中。

#### 34h DWORD dwExStyleFlags

內含一堆 WS_EX_XXX 旗標值，用以表示視窗風格。旗標值定義於 16 位元系統的 WINDOWS.H 和 32 位元系統的 WINUSER.H 中。Windows 95 新增數個風格旗標，稍後我將在「視窗系統的其他改變」小節中做較多的說明。

#### 38h DWORD moreFlags

這個欄位明顯用做旗標，但是其意義未明。

#### 3Ch HANDLE ctrlID（or hMenu）

對於上層視窗（WS_OVERLAPPED 或 WS_POPUP）而言，此欄位內含本視窗之 hMenu。其值可由 *GetMenu* 獲得。對於 WS_CHILD 視窗，此欄位內含本視窗之 control ID。如果視窗是 WS_CHILD，這個欄位內容可以經由 *GetDlgCtrlId* 函式取出。

#### 40h WORD some32BitHandle

這是一個 32 bit handle，指向視窗文字。這個 handle 類似 HWND，但其適用的 heap 既非 window heap 也非 menu heap。

#### 42h WORD scrollBar

這個欄位內含的資訊關係到視窗的捲動桿屬性。

#### 44h WORD properties

這是一個 handle，指向視窗之「properties 串列」中的第一個 property。Properties 其實就是 atoms，允許你把署名之 16 位元數值「綁」在一個視窗身上。欲知更多細節，請看 SDK 文件中的 *SetProp* 和 *GetProp* 兩函式，

### 46h WORD hWnd16

這是一個關鍵性欄位，內含此視窗之 16 位元 HWND。當 USER 獲得一個 32 位元指標，指向一個 WND 結構，它就能夠取得此欄位值，傳回給需要真正 16 位元 HWND 的碼。這使得 USER 得以接受 32 位元 WND 指標而不需要對應之 16 位元 HWND。只要有必要，HWND 可以在 WND 結構中查閱獲得。

### 48h struct _WND32 *lastActive

這是一個 32 位元指標（相對於 USER's DGROUP），指向最近的一個與本視窗有關聯的 active popup 視窗。*GetLastActivePopup* 函式抓此此欄位內容，以之獲取一個 WND 結構，然後傳回儲存在其 0x46 偏移位置的 16 位元 HWND。

### 4Ch HANDLE hMenuSystem

一個 handle，代表此視窗所使用之系統選單（system menu）。請看 SDK 手冊中有關於 *GetSystemMenu* 的說明，以求獲得更詳細的資料。

### 56h WORD classAtom

此欄位內含一個 atom，關係到視窗的類別名稱。它可以是一般的 atom （> 0xC000），也可以是個預先定義好的視窗類別型態：

```
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
```

這個欄位的內容通常和 wndClass 指標（WND 結構的 24h 偏移位置處）所指之結構的第２偏移位置處之欄位內容相同。

### 58h DWORD alternatePID
### 5Ch DWORD alternateTID

這兩個欄位似乎並不實際上含有 process ID 或 thread ID。然而，有一條經過

*GetWindowThreadProcessId* 的路徑，指示出這兩個欄位能夠內含一個 PID 和一個 TID。

最後請你注意一點。當我告訴人們，產生一個視窗並不需要從 USER 的 64K DGROUP 中挖取任何空間，他們常常瞪大眼睛。如果你根據一個既有的類別產生一個視窗，唯一需要配置的空間就是給 WND 結構用的。由於 WND 結構來自 32-bit heap，根本不會影響 USER's 16-bit DGROUP。爲了證明我的說法，我寫了一個小程式，產生數千個視窗，然後檢查 USER's DGROUP 的自由空間。我發現自由空間根本沒有改變！

## 視窗系統的其他改變

對某些使用者而言，Windows 95 最讓人失望的就是沒有能夠把「一個標準視窗最多擁有 64K 文字」的限制撤走。如果你知道 Windows 95 的設計目標，你大概就不會驚訝了。處理並顯示視窗文字的是 16 位元碼。有不少 USER 16 位元碼被轉換爲 32 位元碼以求取打破 64K 的限制。但考慮到大小和回溯相容性，我們可以理解爲什麼 Windows 95 沒有在這一點上這麼做。Windows NT 擁有全然的 32 位元 USER 和 GDI，所以它沒有這種限制。64K 視窗文字限制是終端用戶可以看出的 Windows 95 與 Windows NT 之間的一個主要差異。

正面的評價有沒有？有，Windows 95 提供不少新的視窗風格，讓 Win32 程式看起來更正點。新的風格定義於 WINUSER.H 中：

```
#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
```

我不打算背誦 SDK 手冊上對這些視窗風格的說明。不過，其中有一些十分有趣而並不十分明顯的地方。如果你觀察 16 位元的 WINDOWS.H，你不會看到上面的任何一個定義。這些新視窗風格只有在 32 位元的 WINUSER.H 中才出現。這一整章中我一再強調，幾乎所有 USER 子系統的功能（包括視窗系統）都在 16 位元 USER.EXE 中實作出來。那麼，某些念頭是不是此刻在你心中縈繞不去，嘎嘎作響？如果 16 位元 USER.EXE 實作出視窗系統，它就應該實作出這些視窗風格來，不是嗎？爲什麼 16 位元程式不能夠使用這些新風格呢？事實上在某些非正式的測試中，有些 16 位元 Windows 95 工具程式的確使用了這些新的擴充風格。

## Windows 95 視窗類別的改變

在下海討論 Windows 95 視窗類別的改變之前，我想應該先對視窗類別做一次檢閱。所謂視窗類別，是一組屬性，用以產生視窗。這些屬性包括視窗函式、視窗風格、extra bytes 的大小...等等。雖然理論上 USER 子系統可以不需要「視窗類別」這樣的東西，但每次產生一個視窗就指定每一個屬性，實在是一件痛苦的事情。特別是當某種視窗（例如 button）常常不斷地被製造出新個體的時候。

視窗類別可以被視爲一個模板（template）。在視窗被產生之後，某些類別的屬性會被系統拷貝到 WND 結構中並允許修改。最重要的例子就是視窗函式的更改。同一類別衍生出來的所有視窗基本上都使用相同的視窗函式，然而程式可以利用 *SetWindowLong* 改變某個視窗的視窗函式。這就是所謂的 subclassing。

當 Windows 啓動，它會先產生一些標準類別：

```
Button      ListBox        ComboBox       MDIClient
ComboLBox   PopupMenu      Desktop        ScrollBar
Dialog      Static         Edit           WinSwitch
```

這些類別通常在應用程式中被不斷地重複使用，所以把它們設計爲公用視窗類別是很合適的。*RegisterClass* 函式可以產生（註冊）屬於應用程式的新類別。不管你使用系統視窗類別，或私人的視窗類別，你都必須在呼叫 *CreateWindow* 或 *CreateWindowEx* 時指定一個類別名稱。

在 Windows 3.1 之前，USER 把所有註冊過的類別維護在一個串列之中。TOOLHELP 函式（例如 *ClassFirst* 和 *ClassNext*）可以列舉所有註冊過的類別。在 Windows 95 之中，串列不再。當然，你還是可以呼叫 *ClassFirst* 和 *ClassNext*，獲得某些類別的資訊，然而它們只是系統類別（例如 button、listbox 等等），那是系統啓動時被 USER 模組註冊的類別（請參考 ***Windows Internals*** 第1章）。

和 Windows 3.1 一樣，Windows 95 USER 仍然是從 16-bit heap 中配置記憶體給視窗類別使用（所以，當然啦，每一個視窗類別都會消耗掉一些系統資源）。在 USER 除錯版中，針對類別結構而配置的記憶體，最前面有 LT_USER_CLASS(1) 的標記。

Win32 的哲學思想，就是儘量讓行程對其他行程知道得愈少愈好。是的，Windows 95 的每一個 32 位元行程現在有了它自己私有的類別串列。這個私有的類別串列包括由 system DLLs（例如 COMCTL32.DLL）註冊的類別。每次一有新的行程使用 COMCTL32.DLL，就有數打的類別被註冊。如果你認爲這些私有類別會很快吃光 USER 的 64K heap，你是對的。

我們已經知道有這麼一個私有類別串列，如果能夠走訪它就更好了。遺憾的是 16- 和 32-位元的 TOOLHELP 都沒有提供適用的函式。到目前爲止，我能夠使用的唯一方法就是列舉系統中的所有視窗，然後從其 WND 結構中取其類別指標。類別指標放在 WND 結構的 24h 偏移位置。使用 SHOWWND，你可以手動方式走訪一個行程的類別串列（我的意思是，你必須先找出類別串列的頭，然後雙擊串列中各類別的 next 欄位）。

SHOWWND 的 WNDCLASS.H 中有 Windows 95 視窗類別結構的 C 語言定義。我把此結構命名爲 USER_DGROUP_WNDCLASS。這個結構內含 16 位元 USER 常常必須取用的最小量欄位。不常用的欄位則被移到 32-bit heap 的另一個結構去。USER_DGROUP_WNDCLASS 欄位的詳細內容如下（每一筆資料所呈現的分別是偏移位置、資料型態、欄位名稱）：

**00h DWORD lpIntWndClass**

一個遠程指標（16:16），指向 window heap。這個指標對象是一個 INTWNDCLASS 結構，稍後我會解釋它。基本上，INTWNDCLASS 內含的類別資訊是 USER 不需要立刻取用者。

**04h WORD hcNext**

一個近程指標（在 USER's DGROUP 中），指向下一個類別。下一個類別若不是系統註冊的類別，就是程式私有的類別。

**06h ATOM classNameAtom**

一個 atom，用來代表類別名稱。它若不是一般性的 atom（> 0xC000），就是一個標準的 class atom（0x8000、0x8001 等等）。SHOWWND.C 中有一個 *GetClassNameFromAtom* 函式，用來把 atom 解碼爲類別名稱。

**08h DWORD style**

這裡都是 CS_xxx 旗標值，例如 CS_VREDRAW。這個欄位比起 Windows 3.1 時代擴大了兩倍，以前只是個 WORD。

把所有欄位加總起來，你會發現，一個類別消耗掉 USER's DGROUP 共 0x0C 個位元組。在 Windows 3.1 中，WNDCLASS 的所有資訊都儲存在 USER's DGROUP 中。爲了釋放 USER's DGROUP 的記憶體，微軟把視窗類別結構的大部份欄位移到另一個 32-bit heap 去。USER_DGROUP_WNDCLASS 的第一個欄位內含一個遠程指標指向該 heap，而它所指的就是我所謂的 INTWNDCLASS 結構（"INTernal WNDCLASS" 的縮寫），INTWNDCLASS 結構類似（但不完全等同）你丟給 *RegisterClass* 的那個 WNDCLASS 結構。其格式也在 SHOWWND 的 WNDCLASS.H 中有定義：

**00h DWORD cClsWnds**

截至目前，依此類別而產生之視窗的個數。

#### 04h DWORD lpfnWndProc

視窗函式的位址。所謂 superclassing，就是使用 *SetClassLong* 改變此一欄位。

#### 08h WORD cbClsExtra

此值表示依附於 INTWNDCLASS 結構尾端的所謂 extra bytes 的大小（位元組數）。應用程式可以使用這塊空間儲存私密資料。這些資料可以經由 *SetClassWord*/*Long* 和 *GetClassWord*/*Long* 函式處理之。

#### 0Ah WORD hModule

模組的 HMODULE（該模組註冊此一視窗類別）。請注意這和 SDK 文件中所說的不同，後者把此說成 HINSTANCE。USER 例行性地使用 16 位元 *GetExePtr* 函式（第7章描述），把 HINSTANCE 轉換爲 HMODULE。

#### 0Ch WORD hIcon

依此類別所產生之視窗，將使用此一圖示（icon）。

#### 0Eh WORD hCursor

依此類別所產生之視窗，將使用此一滑鼠游標。

#### 10h WORD hBrBackground

視窗重畫時，以此畫刷做爲背景。

#### 12h DWORD lpszMenuName

依此類別所產生之視窗，將使用此一選單（menu）。這個欄位通常是 0，但偶而它是一個 16:16 指標。

#### 16h WORD hIconSm

依此類別所產生之視窗，將使用此一小型圖示。如果它不是 0，你就可以在 Explorer 中

看到它被用來表示一個視窗。

**18h WORD cbWndExtra**

依此類別所產生之視窗，其 WND 結構之尾端將附加上此欄位所指定之位元組數。應用程式可以利用這塊空間儲存與視窗相關的資料。這些資料可以經由 *SetWindowWord/Long* 和 *GetWindowWord/Long* 存取到。

## SHOWWND 程式

爲了示範我所說過的資料結構，我寫下了 SHOWWND 程式。我也使用 SHOWWND 搜尋出我所描述的資料結構的一些更細節內容。SHOWWND 的重心放在視窗階層架構上。你可以點選系統的任何一個視窗，顯示視窗的每一個欄位內容。如果某個欄位關係到另一個重要的資料結構，你也可以雙擊該欄位，觀察更細部的內容。如此一來，SHOWWND 可以顯示 WND 結構、視窗類別結構、訊息佇列結構中的每一個欄位。由於這三個結構都內含鏈結指標，指向其他同一結構型態之資料實體，所以你可以輕易將視窗、類別、訊息佇列的串列巡訪一遍。

爲了展示資料結構真的如我稍早所述，SHOWWND 儘可能少用 USER 函式，儘可能直接處理資料結構。SHOWWND 可以呼叫 *EnumWindows* 和 *EnumChildWindows* 以顯示視窗的階層架構，但是這種方式無法證明我所描述的一切。當然，進入系統資料結構之中甚至改寫內容，並不是一個程式員應該做的實驗。你應該儘量避免這麼做。然而爲了證明 USER 內部結構，這是我唯一的選擇。

和本書其他程式一樣，SHOWWND 是一個以對話盒爲基礎的程式。你可以從圖 **4-4** 看出，左邊的 listbox 內含一個巢狀的階層圖，表示系統中目前的視窗。按下【Refresh】鈕可以重新顯現一次。右手邊的 listbox 用來展現細部內容。當你選擇左手邊 listbox 中的某視窗，右邊的 listbox 就顯示其 WND 結構。

仔細觀察右邊的 listbox，你會看到數行前面有 + 標記，表示此行可以被滑鼠雙擊以展現更細部的內容。在一個 WND 細部視窗中，你可以進入另一個 WND，或視窗類別，

或視窗的訊息佇列。而從類別的細部視窗中，你可以循著 hcNext 指標，進入串列的下一個類別。訊息佇列視窗也是相同的情況，允許你走訪每一個訊息佇列。

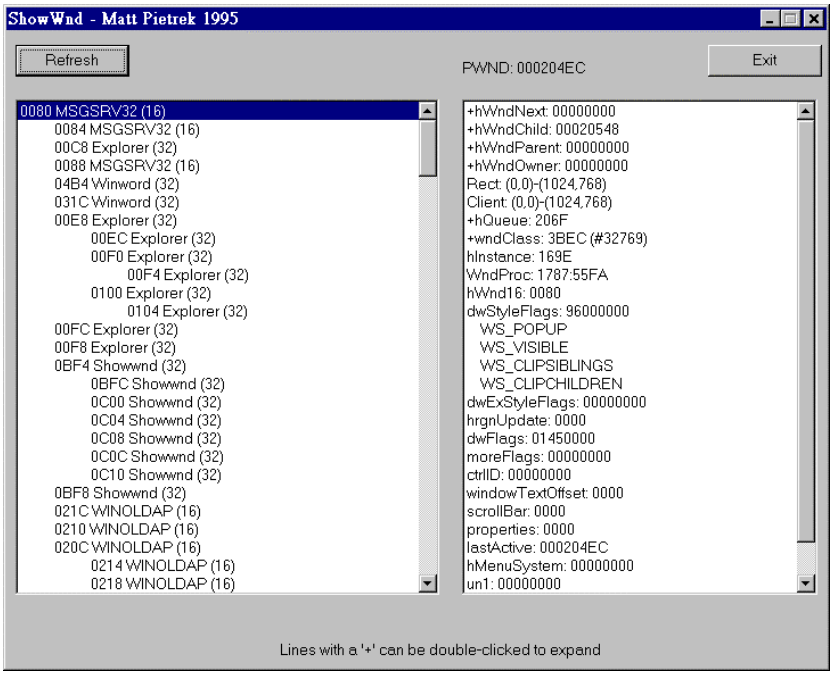



**圖 4-4　SHOWWND 程式是一個以對話盒爲主的程式，內含兩個 listboxes，分別顯示目前的視窗，以及每一個視窗的細部資料。**

大體而言 SHOWWND.C 十分直接了當，所以我不會解釋它們或列出其程式碼來煩你。有一點必須注意。SHOWWND 是一個 Win32 程式，而我相信現在你應該已經知道，Win32 程式可能被其他執行緒強制接收控制權。於是，可能 SHOWWND 還在巡訪其視窗階層架構的半途，另一個執行緒插進來改變了視窗階層。這或許不常發生，但不是不可能。

爲了避免這種事情發生，SHOWWND 在走訪視窗階層時，要求獲得 Win16Mutex。SHOWWND.C 以三個未公開函式完成此項任務：*GetpWin16Lock*、*EnterSysLevel*、*LeaveSysLevel*。*GetpWin16Lock* 將 Win16Mutex 的位址（位於 KRNL386.EXE）填入一個 DWORD 中。如果把此位址交給 *EnterSysLevel*，程式就可以求取 Win16Mutex 並以 *LeaveSysLevel* 釋放 Win16Mutex。這個技術有點類似 ***Unauthorized Windows 95*** 書中的 W16LOCK 程式。關鍵性的差異在於 W16LOCK 使用這些函式來證明系統可以因 Win32 程式而打死結（deadlocked），SHOWWND 卻是使用它們來適當處理執行緒的同步控制。

## 某些 16 位元 USER.EXE 函式的虛擬碼

到目前爲止，我們已經看過了一些關鍵性的 16-bit USER 資料結構，以及 USER 所使用的 32-bit heaps，讓我們看看 USER.EXE 中某些函式的虛擬碼。下面數節在本質上具有實際性，因爲我要顯示給你看，前述的資料結構以及觀念如何運用。

### IsWindow 和 IsWindow16

*IsWindow* 函式取一個 16-bit HWND 做爲參數，並驗證它是否真的是一個合法的 HWND。*IsWindow* 函式碼在 USER.EXE 的除錯版中只不過是個運轉記錄碼而已，如果某個適當的 USER trace mode flag 被設立的話，它就會向除錯埠吐露出函式名稱。真正的 HWND 驗證工作發生在 *IsWindow* 進入 *IsWindow16* 之後。

*IsWindow16* 一開始就快速否決掉不可能爲合法的 HWND 值。如我稍早所說，HWNDs 在 Windows 95 中總是 4 的倍數，所以面對任何一個「最低兩個位元中有一設立」的 HWNDs，*IsWindow16* 立刻傳回 FALSE。小於 0x80 的值也立刻被否決掉。爲什麼單挑 0x80？因爲 handle table 區域中最初的 0x80 個位元組（在 USER's DGROUP 基底位址 +0x10000 處）被用來儲存其他資訊（與 32-bit window heap 有關）。第一個可用的視窗指標是在 handle table 區的 0x80 偏移位置處，而且它總是使用於 desktop 視窗。這很合理，因爲 desktop 是系統的第一個視窗。

*IsWindow16* 接下來又否決掉太高的 HWND 值。在 handle table 區的 0x70 偏移處是一個 DWORD，內含 handle table 的最大可用偏移位置。如果 *IsWindow* 接收的 HWND 比該值大，它就不可能是一個合法的 HWND，所以 *IsWindow16* 傳回 FALSE。

最後，*IsWindow16* 使用 16 位元 HWND 尋找 32 位元的 WND 結構指標。還記得嗎，這個 32 位元指標是相對於 USER's DGROUP，而非 flat 指標。*IsWindow16* 對此指標做兩個檢驗動作。首先，指標必須比 0x10000 大（在 USER32 的 *IsWindow* 中，它必須比 0x20000 大。這才是比較精確的）。第二，這個指標必須不爲 0。如果兩個狀況都吻合，*IsWindow16* 傳回 TRUE，表示這個 HWND 是合法的。

**IsWindow 的虛擬碼**

```
// 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、GetActiveWindow

如我稍早所說，Windows 95 之中的 capture 視窗、focus 視窗、active 視窗是以「每個訊息佇列都有」的方式儲存著。因此，和 Windows 3.1 不同，*GetCapture*、*GetFocus*、*GetActiveWindow* 不能夠只是掐起 USER's DGROUP 中的相關值。換句話說這三個 HWNDs(事實上是三個 USER 32 位元指標)儲存在稍早我說過的 PERQUEUEDATA 結構之中。也就是說，「取出這三個 HWNDs」的程式碼可以共用同一份。

這三個函式都把它們所想要的視窗指標（在 PERQUEUEDATA 結構中的偏移位置）寫進一個暫存器中（虛擬碼中稱此爲 perQueueOffset）。然後統統跳到一個共同點去（虛擬碼中稱此爲 Get_XXX_common）。該共同點首先呼叫 KRNL386 取得一個指標，指向現行執行緒的訊息佇列。佇列中有一個指標指向 PERQUEUEDATA 結構。有了這個指標在手，就把稍早經由 *GetCapture* 或 *GetFocus* 或 *GetActiveWindow* 而決定的偏移值加上去。計算出來的是一個 32 位元指標(相對於 USER's DGROUP)，指向我們所要的視窗。共同碼的剩餘部份就是進入該視窗的 WND 結構，取其 0x46 偏移處所放置的 16 位元 HWND。這段動作表現在虛擬碼中對 *HWnd32ToHWnd16* 的呼叫內。

**GetCapture, GetFocus, GetActiveWindow 的虛擬碼**

```
// 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

*GetWindowThreadProcessId* 是一個新的 Win32 API 函式。Windows 3.x 中最接近的對等函式是 *GetWindowTask*。雖然 *GetWindowThreadProcessId* 是被 32 位元 USER32.DLL 開放出來，事實上卻是由 16 位元的 USER.EXE 實作它。*GetWindowThreadProcessId* 基本上只是參數的檢驗而已。真正的工作放在 *IGetWindowThreadProcessId* 之中。然而在呼叫 *IGetWindowThreadProcessId* 之前，*GetWindowThreadProcessId* 必須先把 16-bit handle 轉換爲與 USER32 相關的 32 位元指標，然後才丟出去。

*IGetWindowThreadProcessId* 必須從兩個不同的地點抓出 process ID 和 thread ID。視窗所關聯的 thread ID 被儲存在此一執行緒的訊息佇列之中。由於訊息佇列是每個執行緒有一個（而不是你所可能想像的，一個行程有一個），所以 process ID 不可能儲存在訊息佇列之中。它儲存在稍早我曾描述過的 QUEUEPROCESSDATA 結構中。*IGetWindowThreadProcessId* 使用訊息佇列取得一個指標，指向 QUEUEPROCESSDATA，然後再從中取出 process ID。

*IGetWindowThreadProcessId* 之中有一段碼我無法解釋。外觀上，似乎如果 QUEUEPROCESSDATA 結構中有一些旗標設立，視窗所關聯的 thread ID 和 process ID 就會儲存在 WND 結構的尾端。我從來不曾發現這樣的實例。

**GetWindowThreadProcessId 虛擬碼**

```
// 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 虛擬碼**

```
// 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

當我正打算決定什麼函式值得觀察，我很快就被 *DesktopWndProc* 吸引了。理由有二。第一，這函式十分簡單，而我打算顯示一個有效運作的、由系統提供的視窗函式。第二，*DesktopWndProc* 啓動了前面我說過的有關於 free system resource 的一些記事行爲。

關於此函式，第一個值得注意的就是，它是一個「準 32 位元」WNDPROC。也就是說，hWnd 和 msg 欄位都是 16 位元，WPARAM 是 32 位元。另一件重要的事情是，此函式會立刻把 16 位元的 hWnd 轉換爲一個與 USER32 有關的 32 位元指標。它使用該指標處理 WND 結構。關於這一點，其他所有標準的 system WNDPROCs 也做相同的事情，它們也使用相關於 USER's DGROUP 的 32 位元指標。

*DesktopWndProc* 的核心是一個 switch 指令（噢，Windows 95 小組還沒有開始用 MFC）。*DesktopWndProc* 所處理的 Windows 訊息列出於下：

- WM_USER：當 desktop 第一次（也唯有第一次）收到這個訊息，它呼叫 *GetFreeSystemResources* 取得 USER 和 GDI heaps 的自由率。後面再有呼叫 *GetFreeSystemResources* 者即以此值爲基備。是誰送出 WM_USER 給 desktop 呢？是 Explorer 自己 -- 在它做完它的初始化動作之後。WM_USER 的作用很明顯是要建立一個 system resource 的用量底線，後續再有呼叫 *GetFreeSystemResources* 的話才能拿來比較。雖然這聽起來頗爲合理，但卻是 Windows 3.1 以來的大轉變。如果微軟曾經描述過這個轉變，那就更好了。目前，眾多使用者會以爲只要把系統更新爲 Windows 95，free system resource 就會暴漲。其實呢，沒有那麼神奇啦。
- WM_ERASEBKGND：此訊息將背景清除，並把整個被指定的四方形合法化。沒有什麼令人興奮的事發生。
- WM_CANCELMODE：如果目前沒有什麼 system modal window（譯註，把整個系統凍結住的那種視窗），這個訊息會落到預設處理常式去。
- WM_NCCREATE：這個訊息的處理常式似乎只是用於穩健性檢查，以確定沒有另一個 desktop 視窗，並確定 desktop 沒有父視窗。
- WM_LBUTTONDBLCLK：這個訊息的處理常式改變了訊息，將它改裝爲 WM_SYSCOMMAND，並將 WPARAM 的較高字組設爲 SC_TASKLIST。在 Windows 3.1，面對 desktop 做出滑鼠雙擊動作會帶出 task manager（譯註：一個程式）。在 Windows 95，當 *DefWindowProc* 收到 SC_TASKLIST 命令，它會呼叫 shell，帶出 Explorer 的【start】選單。

- WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED：這兩個訊息處理常式會呼叫 USER 中的某些函式，進入某些與調色盤相關的動作。

任何訊息進入 *DesktopWndProc* 之中而沒有被上述處理常式攔下來的話，就會進入 *DefWindowProc*（而 *DefWindowProc* 的內部動作恐怕又得另一本書才能說明了）。

**DesktopWndProc 函式虛擬碼**

```
// 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 );
```

## USER32 並不純粹只是轉運站

整章之中，我一再強調 Windows 95 USER 子系統的真正工作是由 16-bit USER.EXE 完成。真的，USER32.DLL 絕大部份只是個轉運站，把函式下移（thunk down）至 USER.EXE。但如果你把 USER32.DLL 想像全然是一個轉運站，並不正確。看看 USER32 的列表，顯然有時候微軟會決定哪一個 USER 函式因爲太常被呼叫，以至於最好直接實作出來，免得下移至 USER.EXE。在爲數不多的幾個情況下（例如稍後我所顯示的幾個），Windows 95 小組認爲，消除一個 thunk，可以獲得速度上更多的利益，所以在 USER32.DLL 中又增加了一些額外的碼。下一節我所描述的函式並不是完整的列表，它們只是視窗系統函式中的幾個典型例子。

### USER32 的 IsWindow 函式

USER32 的 *IsWindow* 函式比其 USER.EXE 版稍稍複雜一些。由於 USER32 的 *IsWindow* 可以被一個 Win32 執行緒呼叫，而該執行緒目前沒有擁有 Win16Mutex，所以這個函式使用兩個輔助函式（*GrabWin16Mutex* 和 *ReleaseWinMutex*）把呼叫到此函式之核心者加以分類。函式所呼叫的 *GetWndPtr32*（稍後描述）在整個 USER.DLL 中一再用到。如果 *GetWndPtr32* 傳回 0，*IsWindow* 就傳回 FALSE，代表放進來的 HWND 是不合法的。否則，*GetWndPtr32* 傳回非零值而 *IsWindow* 傳回 TRUE，

**IsWindow 函式虛擬碼**

```
// in USER32.DLL
// Parameters:
//  HWND    hWnd       // The 16-bit version.
// Locals:
//  BOOL    retValue;

    GrabWin16Mutex();

    retValue = GetWndPtr32( hWnd );     // Pass 16-bit HWND version.

    ReleaseWin16Mutex();
```

**GrabWin16Mutex 函式虛擬碼**

```
EnterSysLevel( pWin16Mutex );   // Call KERNEL.97 to acquire the Win16.
```

**ReleaseWin16Mutex 函式虛擬碼**

```
LeaveSysLevel( pWin16Mutex );   // Call KERNEL.98 to release the Win16
                                // mutex semaphore.
```

## USER32 的 GetWndPtr32 函式

*GetWndPtr32* 是一般性的 USER32 內部函式。給予一個 16-bit HWND，它就會傳回與 USER32 相關的 32 位元指標，指向 WND 結構。其中檢查 16 位元 HWND 並尋找 WND 結構的過程，和 16 位元 USER.EXE 中的 *IsWindow* 函式幾乎相同。唯一的不同是在函式末尾：16 位元的 *IsWindow* 傳回 TRUE 或 FALSE，32 位元的 *GetWndPtr32* 傳回的是 WND 結構指標。

**GetWndPtr32 函式虛擬碼**

```
// 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 的 GetCapture、GetFocus、GetActiveWindow 函式

稍早我曾展示過 USER.EXE 中的 *GetCapture*、*GetFocus*、*GetActiveWindow* 函式虛擬碼。其 32 位元版基本上是相同的，但有兩點不同，需要注意。第一，USER32 版在處理 USER 資料結構時都索求 Win16Mutex。處理完畢後釋回。16 位元版就沒有這麼做，因爲它們是 16 位元碼，根據定義，Win16Mutex 本來就會被抓取。第二個不同是 USER32 版中沒有錯誤檢驗。16 位元版中會在開始處理 PERQUEUEDATA 結構之前先確定確實存在有一個佇列。

**GetCapture、GetFocus、GetActiveWindow 函式虛擬碼**

```
// 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 的 GetMessagePos、GetMessageTime、GetMessageExtraInfo

USER32 中的這三個函式基本上和在 USER.EXE 中的 16 位元版相同。由於它們都只是抓取執行緒訊息佇列中的一個欄位，所以它們都是把某特定偏移位置儲存在一個暫存器中，然後就呼叫一個共同函式。該共同函式獲得一個指標指向執行緒佇列，然後再從其中取出對應的 DWORD。有趣的是，這些函式並不像 USER 的 *GetCapture*、*GetFocus*、*GetActiveWindow* 那樣會索求 Win16Mutex。

**GetMessagePos、GetMessageTime、GetMessageExtraInfo 函式虛擬碼**

```
// 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 的 SendMessage 函式

你或許會驚訝我竟然提供 USER32 的 *SendMessage* 函式虛擬碼。*SendMessage* 是 USER 子系統中最複雜的一個函式，所以它當然是下移（thunk down）至 16 位元 USER.EXE，對不對？大部份時候這個假設是正確的。然而 *SendMessage* 是一個被繁重使用的函式，如果狀況吻合，它也可以自己完成工作而不必下移。我談的正是主效率的改善。

USER32 *SendMessage* 一開始就要求 Win16Mutex。然後進行一系列的偵測，看看此一訊息是否可以安全地被送出去而不需要真正的 USER.EXE 中的 *SendMessage* 函式的幫

忙。奪其全功並迫使其下移至 USER.EXE 的情況有：

- HWND 爲 0。
- 標的視窗的訊息佇列不同於現行執行緒的佇列。
- USER.EXE 的 DGROUP 中的某些變數不爲 0。

如果某個被 sent 過來的訊息經過種種測試後必須下移（thunk down），*SendMessage* 就開始爲呼叫 WNDPROC 做準備。尤其是，*SendMessage* 需要它即將呼叫的 WNDPROC 的位址。

一如我稍早所說，WND 結構本身並未儲存有指向 WNDPROC 的 32 位元 flat 指標。如果 WND 結構所代表的是一個 32 位元視窗，那麼該結構中的 WNDPROC 位址將指向一塊 16 位元碼（或稱之爲 stub），彼端將把控制權轉到 32 位元領域去。*SendMessage* 知道這些特別的 stubs，也能夠從其中讀出 32 位元 WNDPROC 位址。最後，在跳至該目標位址前，*SendMessage* 釋放 Win16Mutex。這整個程序似乎是有點奇怪，但如果它可以有效運作並且改善效率，何樂不爲！

**SendMessage 函式虛擬碼**

```
// 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 的 GetDlgItem 函式

這是 Win32 中另一個被繁重使用的函式，特別是在與對話盒相關的碼。給予一個 HWND 和一個 child control ID，這個函式必須傳回 child control 的 HWND。如果你還記得 WND 結構，你就會了解 *GetDlgItem* 所做的事情只不過就是走訪 WND 階層架構，尋找某個擁有吻合之 control ID 的視窗罷了。

USER *GetDlgItem* 函式一開始先設法取得 Win16Mutex(畢竟我們不希望視窗階層架構在此函式工作期間有所改變)。對話盒中的 controls 都是對話盒的子視窗。因此，*GetDlgItem* 需要做的就是走訪此對話盒的子視窗串列，並將每一個 control ID 和輸入的 idControl 拿來比較。一旦有吻合者，就查詢並傳回 16-bit HWND。當然啦，在回返之前，此函式會將 Win16Mutex 釋放掉。

**GetDlgItem 函式虛擬碼**

```
// 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 的 GetDlgCtrlID 函式

*GetDlgCtrlID* 是 *GetDlgItem* 的補充。給予一個 16-bit HWND，它會傳回對應的 control ID。和 *GetDlgItem* 一樣，此函式會在握有 Win16Mutex 之後才開始動作。

*GetDlgCtrlID* 把輸入的 16-bit HWND 參數交給 GetWndPtr32，獲得一個與 USER32 相關的 32-bit WND 指標。如果那是一個 non-null 指標，*GetDlgCtrlID* 就從該 WND 結構的適當偏移處取出 control ID 並傳回。噢，當然不要忘記釋放 Win16Mutex。

**GetDlgCtrlID 函式虛擬碼**

```
// 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 對 Unicode 的支援（哦？有嗎）

相不相信，Windows 95 的確有些微的 unicode 支援能力。不相信嗎？檢查下面這個我名之爲 WIN95UNI.C 的小程式：

```
#define UNICODE
#include <windows.h>

int main()
{
    MessageBox( 0,
              TEXT("Yes! Really!"),
              TEXT("Unicode in Windows 95?"),
              MB_ICONQUESTION );
    return 0;
}
```

編譯之後，這段碼會產生一個 unicode 程式。我們甚至可以利用第 8 章的 PEDUMP 工具程式驗證其 EXE 檔：

```
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....
```

很明確地其中有一個 unicode 版本的 *MessageBox* 函式呼叫動作。執行時會發生什麼事？請看**圖 4-5**。


**圖 4-5　WIN95UNI 程式證明了 Windows 95 支援 unicode。**

**譯 4-5** 到底是怎麼回事？Windows 95 並不支援 unicode，但你看到了，WIN95UNI 證明了其實存在有一些對 unicode 的支援。下面是 unicode *MessageBoxW* 的呼叫串鏈：

```
MessageBoxExW
        WideCharToMultiByte // Convert the 2nd parameter to ASCII.
        WideCharToMultiByte // Convert the 3rd parameter to ASCII.
        MessageBoxExA       // Invoke the ASCII MessageBoxEx.
```

爲什麼 Windows 95 要淌這趟 unicode 的渾水呢（至少從簡約生活的角度來看）？微軟對於 "Windows 95 logo" 的條件之一是，程式應該能夠在「未能全數提供應用程式所需之支援」的系統中，姿態優雅地把自己降一個級數。程式能夠做的一個降級動作就是丟出一個 *MessageBox*，然後說「抱歉，無法執行」。提供適度的 unicode 支援，Windows 95 就能夠讓那些 unicodee 應用程式至少得以說明它沒有辦法執行（或是沒有辦法完全執行）。

## USER.EXE 的 UserSeeUserDo 函式

尚未探討過 *UserSeeUserDo* 函式之前，我沒有辦法把 USER 子系統的討論做一個結束。這個函式從 Windows 3.1 開始就提供了，是一個未公開函式，提供許多 USER 變數和函式的後門。到了 Windows 95，可以進出後門的事物更多了。審視 *UserSeeUserDo* 所提供的功能，可以讓我們對 USER 建構者認爲重要的關鍵事物也有一個掌握。

*UserSeeUserDo* 實作於 16 位元 USER.EXE 中，有四個參數。第一個參數指示 *UserSeeUserDo* 的任務，剩餘三個參數的意義則依第一個參數而定。

前三個子函式允許呼叫者配置、釋放、或緊密（compact）來自 USEE's 16-bit DGROUP 的記憶體。接下來五個子函式用來取出重要的 USER 全域變數：

1. menu heap handle
2. head of the system class list
3. USER's DGROUP handle
4. head of the device context entry chain（請參考 ***Undocumented Windows*** 第 5 章）
5. desktop window 的指標

最後一個變數並非 desktop 視窗的 16-bit HWND，而是相關於 USER32 的 32 位元指標，指向 desktop 視窗的 WND 結構。

最後兩個子函式用來配置和釋放來自新的 USER's 32-bit heaps 的記憶體。前一個函式配置，後一個函式釋放。如果 *UserSeeUserDo* 的第二個參數不是 0，就從 32-bit menu heap 配置記憶體。否則就從 32-bit window heap 配置記憶體。

**UserSeeUserDo 函式虛擬碼**

```
// 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 模組

在我描述過 Windows 95 的 USER 子系統之後，我對 GDI 的報導或許會被你譏爲虎頭蛇尾 -- 如果你是一個在螢幕上追逐圖素（pixel）的 GDI 狂熱者的話。並不是 GDI 不重要，事實上 Windows 95 的繪圖模組裡頭加入了許多新而令人振奮的東西。問題是，對於一個 kernel 狂熱者如我，繪圖模組實在是淡而無味。現在我要開誠佈公我的 GDI 經驗。

如果我只能傳授 Windows 95 的 GDI.EXE 和 GDI32.DLL 一個訊息，那一定是：USER 子系統和 GDI 子系統是平行的。兩個子系統都管理一些來自其 heaps 的物件。在 USER 之中，主要的物件是 windows、menus、classes，在 GDI 之中，主要的物件是 pens、brushes、bitmaps 等等等。在 Windows 3.1 之中，兩者都深受 64K heap 之苦（縱使 USER.EXE 已經把 menus 放到另一個 64K heap 去）。在 Windows 95 之中，兩者還是強烈依賴從其 DGROUP heaps 中配置而來的資料結構。USER 和 GDI 也都以 Win32 heaps 處理 2MB 記憶體，用來放置大塊資料。USER DGROUP 的佈局、handle table 區域、以及 32-bit window heap 完全平移到 GDI 來，如圖 **4-6**。

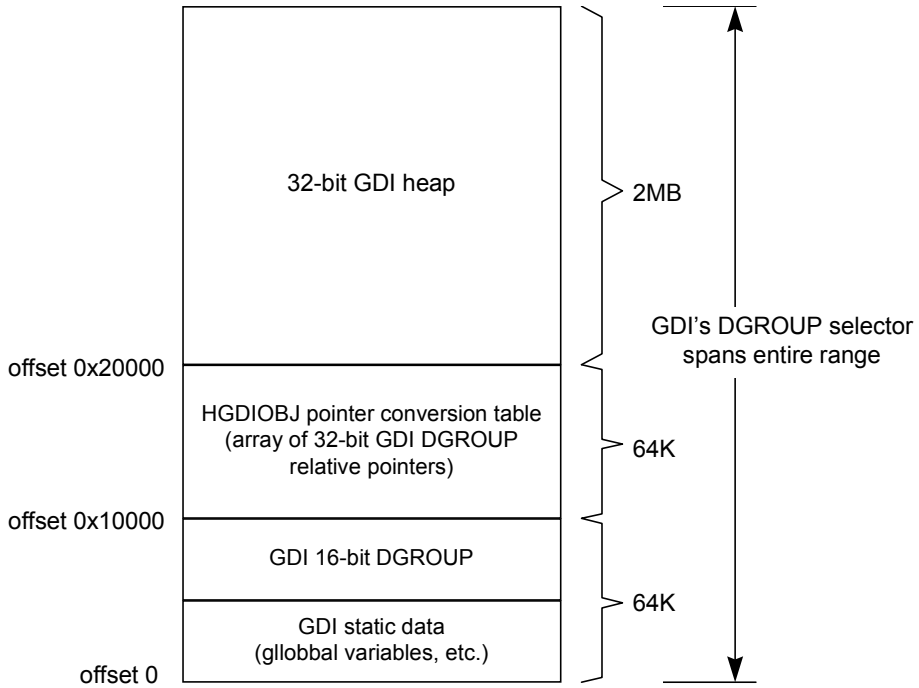



**圖 4-6 在 Windows 95 之中，GDI 和 USER 有著幾乎相同的結構。從此圖你可以看出，GDI 的 DGROUP 佈局及 handle table 區域和 USER 極爲相似。**

正如你以 handles（HWNDs、HMENUs 等等）處理 USER ，你也可以以 handles 處理 GDI 物件（HPENs、HBRUSHs 等等）。稍早，我曾說過 16-bit HWNDs 如何用來查閱陣列之中的一個 32 位元指標，以找出一個 WND 結構的精確位置。對於儲存在 32-bit GDI heap 中的 GDI 物件而言，16-bit handles 到 32-bit 指標的轉移過程就像是 HWNDs 在 USER 端的轉移過程一樣。如果 handle 所表示的一個 GDI 物件是從 GDI DGROUP 中配置而來，那麼 handle 就是一般的 16-bit heap local handle，可以輕易被解析爲一個 GDI 16-bit DGROUP 的偏移位置。

整個重點在於，想像 USER 和 GDI 兩個子系統在操作上是平行的 -- 至少在它們處理資料結構的這一層面是如此。如果你真的了解 USER 如何處理 handles 和指標，你應該就可以探究 GDI 碼並知道怎麼一回事，不會有太大困難。

什麼資料是 Windows 95 GDI 強迫一定要移到 32-bit heap 去的呢？唔，根據 Win32 SDK 提供的 HeapWalk 工具程式的觀察結果，fonts 和 regions 被丟出去了。此外還有一些物件是 HeapWalk（以及我自己）沒辦法識別的。

另一個 USER 和 GDI 的平行機制是 thunking。USER 子系統中最主要的碼實作於 16-bit USER.EXE。USER32 主要（但非全部）是做轉運站。GDI 的情況類似，但不完全相同。大部份的 GDI 還是在 16-bit GDI.EXE 中實作出來，然而微軟加了許多與 GDI 相關的新特質，並且支援 Win32，放到 GDI 子系統中。某些新的碼放在 GDI.EXE，某些新的碼則放在 GDI32.DLL。微軟說，置於 GDI32.DLL 的那一部份是 TrueType rasterizer，spooler 和 printing 子系統，以及 DIB 引擎。我並沒有確實地證明其真假，然而仔細觀察 GDI32.DLL 程式碼，我發現其中有相當程度的碼並不只是下移（thunk down）至 GDI.EXE 而已。

16-bit GDI.EXE 值得注意的一點就是，在 16 位元模組中竟然有 32 位元碼。第 7 章中我將描述 16 位元 NE 檔案中的一些 segment table entries，它們用來告訴 Windows 95 載入器：針對這個 segment，製作出一個 32 位元 code selector。也就是說，當 CPU 把那樣的 selector 放到 CS 暫存器中，它所指向的那一段碼會被解釋爲 32 位元碼，而不是大部份 Win16 EXEs 和 DLLs 的 16 位元碼。16 位元 GDI.EXE 使用四個 32 位元節區。雖然其中並沒有任何輸出函式（export function），我還是檢查了那些碼，並且獲得以下結論：

```
GDI.EXE Segment 0x20：Bezier
GDI.EXE Segment 0x23：Paths，Enhanced Metafile（EMF）
GDI.EXE Segment 0x24：???（未知）
GDI.EXE Segment 0x26：字串 "engine font" 出現在此節區中。
```

微軟對於 Windows 95 的 16- 和 32- 位元模組的描述中說，Beziers、paths、enhanced metafiles 都位於 16 位元 GDI.EXE。符合我的發現。

## GDI 物件

成爲 GDI 專家的關鍵之一就是了解 GDI 物件。GDI 子系統處理一打以上的不同物件型態。它們大部份都有自己獨一無二的 handle 名稱。毫無疑問你一定已經熟悉了它們。例如，一個 device context（DC）就是一種 GDI 物件，你把一個 HDC（DC handle）交給許許多多的 GDI 函式。Pen 也是一個 GDI 物件，你可以以一個 HPEN（pen handle）表示某支特定的筆。凡是能夠接受任何型態之 GDI 物件的函式，都有一個 HGDIOBJ 參數。HGDIOBJ 可被視爲特定型態之 GDI 物件如 HDCs、HBRUSHs 等之基礎類別。你可以在 Windows 3.1 中找到 GDI 物件的列表，只要在 TOOLHELP.H 中尋找 LT_GDI_xxx #defines 就好。不幸的是，這些 #defines 顯然沒有針對 Windows 95 新的 GDI 物件型態而更新。

可以這麼說：GDI 嘗試以一致的方式處理其物件，因爲它擁有像 *SelectObject* 和 *DeleteObject* 這樣的函式，不需要你告訴它傳進去的是什麼物件。GDI 會檢查傳入的物件，決定其型態，並做出適當的動作。GDI 如何根據一個物件得到其型態？每一個 GDI 物件一開始都有一個標準的表頭，內含一個 WORD，標示物件的型態。Windows 95 GDI 物件串列（及其對應的標示值）列出於下：

```
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 內的 IsGDIObject 函式

技術文件上說，如果傳入的 GDI object handle（一個 HGDIOBJ）不合法， *IsGDIObject* 將傳回 FALSE。有趣的是，文件上又說，如果 *IsGDIObject* 傳回 TRUE，傳入的 handle 也有可能不是一個真正的 GDI object handle。雖然如此，此函式的公開目的還是爲了決定一個 HGDIOBJ 是否合法。文件中唯一沒有告訴你的是，如果 HGDIOBJ 合法，函式傳回值可以表示其真實型別。這對應用軟體如 Bounds-Checker/W 很方便，因爲那種軟體必須識別 HDCs 和 HBRUSHs 等 handles。

一如我稍早描述的，GDI 在其 16-bit DGROUP heap 中儲存了一些物件，在其 32-bit heap 中儲存了另一些物件。*IsGDIObject* 函式的第一要務就是了解它應該從哪個 heap 中尋得物件，如此一來它才能夠讀出代表物件型別的 WORD（例如 0x4F47）。幸運的是，這件事情對 GDI 並不困難。GDI 物件中凡是從 16-bit DGROUP heap 中配置而來的，都配以 LMEM_MOVEABLE 屬性。讓我長話短說，16-bit LMEM_MOVEABLE handles 總是以 2, 6, 0xA, 0xE 結束。而請你回憶一下本章前面所說，在 32-bit USER 或 GDI heap 中，handles 總是 4 的倍數。

知道這個關鍵性的區別後，*IsGDIObject* 只需要驗證 handle 的倒數第二個位元就好：如果該位元設立，此 handle 必定是以 2, 6, 0xA, 0xE 結束，所以物件必定是從 GDI 的 16-bit DGROUP 配置而來。如果倒數第二個位元是 0，handle 的值是以 0, 4, 8, 0xC 結束，所以物件必定是來自 32-bit GDI heap。不管哪一種情況，*IsGDIObject* 都會算出可以尋獲物件的位址，並建構一個指標，指向該物件。利用這個指標，*IsGDIObject* 就可以取出其「型別 WORD」。

有了「型別 WORD」在手，*IsGDIObject* 於是把一些無關的位元遮罩住，遮罩結果必定在 0x4F47~0x4F52 之間。如果不是這樣，這就不是一個合法的 GDI 物件，於是 *IsGDIObject* 傳回 0。如果在合法範圍之內，*IsGDIObject* 就把該值減去 0x4F46，使其成爲一個從 1 起算的整數，並傳回。

**IsGDIObject 函式虛擬碼**

```
// 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 函式

Win32 API 並沒有 *IsGDIObject* 這一函式。幸運的是，Win32 更進一步提供了一個可以傳回 HGDIOBJ 型別的函式。是的，*GetObjectType* 比 *IsGDIObject* 又更精巧一些。

*GetObjectType* 先檢查 HGDIOBJ handle，看看是否它真正是一個 selector。metafile object 的 handle 就真正是一個 selector，指向 metafile 中的資料。如果 HGDIOBJ 看起來像是一個 selector，*GetObjectType* 會摸索探進這個節區之內。而如果它又發現一些它認爲應該要有的欄位，它就會傳回 OBJ_METAFILE。

測試 selector 的工作完畢之後，*GetObjectType* 進入一段碼，看起來明顯類似 GDI.EXE 的 *IsGDIObject* 的所作所爲。如果 handle 值以 2, 6, 0xA, 0xE 結束，, *GetObjectType* 就假設這是一個 16-bit local heap handle，代表一個存在於 USER's 16-bit DGROUP 的物件。真是這樣的話，*GetObjectType* 就索取 Win16Mutex，避免被哪個執行緒改變 USER heap 的狀態。如果 HGDIOBJ 不是以 2, 6, 0xA, 0xE 結束，*GetObjectType* 就認爲這個物件是 font 或 region，存在於 32-bit heap 中。無論如何，它都會產生一個 32 位元指標，指向該 GDI 物件。

有了這個指標，*GetObjectType* 取出「型別 WORD」，並進行類似 *IsGDIObject* 的位元遮罩行爲和減數行爲。然後檢查物件型別，看是否在合法範圍內。如果不是，就傳回 0，如果是 6（一個 HDC），*GetObjectType* 更進一步進入物件資料之中，看看它是不是一個 enhanced metafile DC 或一個 memory DC。如果真是，就傳回 OBJ_XXX（定義於 WINGDI.H）。

*GetObjectType* 的最後一步就是把 16 位元的物件型別（如 TOOLHELP.H 所定義的 LT_GDI_xxx）轉換爲對應的 32 位元 OBJ_XXX。因爲某些奇怪的理由，OBJ_XXX 並沒有一對一對應到物件型態。這或許是因爲 OBJ_XXX 最初是由 Windows NT GDI 小組所定義，而他們並不以 Windows 3.1 GDI.EXE 爲基礎）。任何情況下，物件型態都必須從 GDI.EXE 所使用的值轉換爲 WINGDI.H 所定義的 OBJ_XXX。這個轉換步驟是藉由一個查詢表格的協助而完成。最後，*GetObjectType* 釋放 Win16Mutex（如果先前它有取得的話）。

**GetObjectType 函式虛擬碼**

```
// 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;
```

## 適用於 Win16 程式的新的 Win32 GDI 函式

做爲對 GDI 的最後一瞥，我好奇地想知道有多少個新的 Win32 GDI 函式。爲了解是否任何 Win32-only GDI 函式都能夠被 16 位元碼呼叫，我所必須做的就是從 Windows 95 的 GDI.EXE 中傾印出其輸出函式（export function），再與 Windows 3.1 的 GDI.EXE 輸出函式做比較。過濾掉一些未公開函式之後，留下來的就是新增、並且可以被 Win16 呼叫的 Win32 GDI 函式，比較兩個 GDI.EXE 版本的工作，可以由 ***Undocumented Windows*** 一書所提供的一個卓越的 EXEUTIL 程式，不費吹灰之力地完成。進行下面這個命令：

```
EXEUTIL -diff C:\WIN31\SYSTEM\GDI.EXE  C:\WINDOWS\SYSTEM\GDI.EXE
```

可以帶給我兩個版本的 GDI.EXE 之間的輸出函式的差異列表。只有三個未公開函式從 Windows 95 的 GDI.EXE 中被移走。列表之中有一些新的 16 位元 GDI 函式被加到 Windows 95 之中。我過濾掉所有的未公開函式，以及那些沒有對等之 Win32 API 的函式。經過重新排列，我把結果顯示於**表格 4-1**。這些函式都是輸出函式（export function），並且可以被 Win16 碼安全地呼叫。

**表格 4-1　新的 GDI 函式，可以從 Win16 碼中呼叫之。**

| 函式種類 | 函式名稱 |
|---|---|
| Printing 這些函式統統都被上移（thunk up）至 GDI32.dll | ABORTPRINTER, CLOSEPRINTER, ENDDOCPRINTER, ENDPAGEPRINTER, OPENPRINTERA, STARTDOCPRINTERA, STARTPAGEPRINTER, WRITEPRINTER |
| Device-Independent Bitmaps 這些函式實作於 GDI.EXE 中 | CREATEDIBSECTION, GETDIBCOLORTABLE, SETDIBCOLORTABLE |
| Enhanced Metafiles 這些函式實作於 GDI.EXE 中，並獲得 32 位元碼的協助 | CLOSEENHMETAFILE, COPYENHMETAFILE, CREATEENHMETAFILE, DELETEENHMETAFILE, GDICOMMENT, GETENHMETAFILE, GETENHMETAFILEBITS, GETENHMETAFILEDESCRIPTION, GETENHMETAFILEHEADER, |

| | |
|---|---|
| | GETENHMETAFILEPALETTEENTRIES, PLAYENHMETAFILERECORD, SETENHMETAFILEBITS, SETMETARGN |
| line drawing 這些函式實作於 GDI.EXE 中，並獲得 32 位元碼的協助 | GETARCDIRECTION, POLYBEZIER, POLYBEZIERTO, SETARCDIRECTION |
| Paths 這些函式實作於 GDI.EXE 中，並獲得 32 位元碼的協助 | ABORTPATH, BEGINPATH, CLOSEFIGURE, ENDPATH, FILLPATH, FLATTENPATH, GETMITERLIMIT, GETPATH, PATHTOREGION, SELECTCLIPPATH, SETMITERLIMIT, STROKEANDFILLPATH, STROKEPATH, WIDENPATH |
| Miscellaneous 這些函式實作於 GDI.EXE 中 | CREATEHALFTONEPALETTE, ENUMFONTFAMILIESEX, EXTCREATEPEN, EXTCREATEREGION, EXTSELECTCLIPRGN, ETCHARACTERPLACEMENT, GETFONTLANGUAGEINFO, GETREGIONDATA |

## 摘要

我用一整章來表現 Windows 95 的 USER 和 GDI 模組奇怪而混合的性質。雖然它們明顯繼承自 Windows 3.1，它們還是有相當份量的 32 位元碼。其結果就是程式員可以從中獲利的許多改良性質（不管 16- 或 32- 位元程式）。此外，將重量級資料（如 WND）搬離 16-bit heaps，是 Windows 95 一個非常好的改良。雖然 Windows 95 的 USER 和 GDI 模組並不像 Windows NT 那樣全功能以及強固穩健，它還是對於滿受挫折的 Windows 3.1 程式員帶來了很不錯的信心。

# 記憶體管理（Memory Management）

正當程式員逐漸習慣 Windows 3.x 的記憶體管理特性，微軟祭出 Win32 API，又帶給我們這些本來就已經快被眾多函式淹死的程式員一組新的挑戰。

理論上 Win32 記憶體管理在其三個平台化身（NT、Windows 95、Win32s）之間應該是十分類似的。然而，了解微軟在此一領域的軌跡之後，你應該可以預期 Windows 95 記憶體管理和 NT 和 Win32s 之間還是有許多差異（包括好的和壞的）。這一章我要討論 Win32 記憶體管理的 Windows 95 版。注意，此處描述的許多觀念也適用於 NT 和 Win32s。

我把記憶體管理的子題目分爲兩個。第一個子題目與行程的位址空間、memory contexts、分頁行爲（例如 "copy on write"）有關。第二個子題目是作業系統提供的記憶體管理函式。

如果你正在尋覓 16 位元（或稱 DOS 虛擬機器）記憶體管理的相關資料，本章並不是你要的。我很果斷地把此章定位在 32 位元方面。如果你對 Windows 95 的 16 位元記

憶體管理有興趣，請看我的前一本書 ***Windows Internals*** 第2章。Windows 95 的 16 位元記憶體管理幾乎和 Windows 3.1 之間完全沒有改變（除了臭蟲清除了不少之外）。有了這些準備，現在讓我們跳到...

# 以分頁為基礎之 Windows 95 記憶體管理

如果你打算真正了解 Windows 95 的記憶體架構，那麼你就不可能對 Intel 80386 CPU 的記憶體分頁（paging）避而不談。記憶體分頁技術其實遠早於 80386，但因爲我們只對 Windows 95 如何在 80386 上使用分頁機制感興趣，所以我只使用 80386 的那些術語。如果你稍稍了解分頁（paging），你可以跳過此節。如果你對記憶體分頁感覺十分陌生，或是你需要快速充電，那麼就請安心地讀下去。

## 記憶體分頁（Memory Paging）

分頁（paging）的主要理由是提供一種方法，讓作業系統和 CPU 聯手欺騙程式，使程式誤以爲它所擁有的記憶體比真正安裝在機器上的還多。當程式讀寫記憶體中的一個位元組，它可能是（也可能不）存取實際 RAM 上的一個位元組。如果程式碰觸一個位址，而該位址卻沒有對應到實際的 RAM，CPU 就會通知作業系統。作業系統於是採取必要措施，把該位址和實際的 RAM 關聯起來。

如果所有執行中的程式的記憶體用量超過了實際記憶體安裝量，作業系統可能需要從甲程式中抽出一塊記憶體給乙程式使用。盲目地將一塊使用中的記憶體另作它用，是災難的開始，所以 Windows 95 必須把原來的內容另外儲存在其他地方。這所謂的其他地方，就是硬碟。任何時候，作業系統以及所有應用程式所使用的記憶體內容，不是儲存在 RAM 之中，就是儲存在硬碟之中。我這麼說雖然略嫌粗糙，但目前爲止還堪夠用。

上面所說的，以分頁方式以及第二儲存空間來模擬記憶體，一般稱之爲「虛擬記憶體」。Windows 95 虛擬機器管理器（VMM32.VXD 中的 VMM 模組）的一個基礎工作就是提供虛擬記憶體，使應用程式受到的驚擾達到最小。

最令許多人迷惑的就是，分頁會影響 CPU 的記憶體定址。如果沒有分頁，程式交給 CPU 的位址，將等同於電腦記憶體匯流排（memory bus）上的位址。舉個例，在真實模式應用程式中，你很容易根據 segment:offset 計算出實際位址：把 segment 乘以 16 再加上 offset 就是了。但是當分頁機制啓動，程式所使用的位址可能和 CPU 送往記憶體匯流排的位址不同。分頁機制爲所有位址導入了一層（事實上是兩層）間接性。當程式交給 CPU 一個位址，CPU 使用這 32 位元位址中的某些位元，尋找實際 RAM 的位址（它將在記憶體匯流排上通行）。CPU 用以轉換位址的表格，由作業系統控制，這麼一來作業系統就可以告訴應用程式使用 4GB 定址空間的任何一個地方，而不必在乎那裡是不是真的有實際的 RAM。

之所以使用「分頁（paging）」這個名詞，是因爲 CPU 並不提供以 byte 爲單位的間接定址。記憶體位址的轉化都是以 4K 爲單位。如果你使用分頁機制來指定實際位址 0x1000 對應到程式的 0x400000，實際位址 0x1001 就會對應到程式的 0x400001，實際位址 0x1FFF 就會對應到程式的 0x400FFF。然而，下一個程式位址（也就是 0x401000）是另一個 4K 的開始，所以實際位址 0x2000 並不一定會對應到程式位址 0x401000。也許程式位址 0x401000 映射到實際位址 0x6000，或者根本沒有映射到任何實際的 RAM。所有的映射關係都由作業系統的分頁機制控制之。

除了允許作業系統提供虛擬記憶體，CPU 對分頁的支援還因此提供了一個很大的彈性，讓作業系統在記憶體中安排各式各樣的物件。所謂物件，我指的是像作業系統碼、應用程式碼、程式資料區、記憶體映射檔等等。作業系統所使用的記憶體佈局稱爲定址空間佈局（address space layout）。稍後我將描述 Windows 95 的定址空間。

分頁的好處是，作業系統得以把各種作業系統物件散佈在 CPU 的整個定址範圍中。以 Intel 386 CPU 家族而言，定址範圍是 4GB。理論上可定址的整個記憶體範圍稱爲 CPU 的位址空間（address space）。由於分頁機制起作用，因而必須先被轉換的那些位址，稱爲線性位址（linear address）。至於轉換後的，才是真正放到匯流排上的位址，稱爲實際位址（physical address）。記住一點，任何情況下應用程式以及 API 所使用的，幾乎都是線性位址，而非實際位址。

一旦擁有分頁能力，作業系統就可以指定位址空間中各式各樣的區段，給特定的資料項目使用，並預留空間以備成長或增加項目。例如，當應用程式一啓動，預設情況下 Windows 95 會保留 1MB 的位址空間做爲應用程式的 stack。這並不意味 Windows 95 馬上就要把 1MB RAM 映射到這 1MB 位址空間中，而是說這塊空間最大是 1MB。實際上 Windows 95 是 4K 4K 地將 RAM 映射到那塊保留的位址空間中。

分頁機制使得作業系統得以保留巨大的記憶體位址範圍，卻不需要先付出代價（我是指 RAM），直到真正需要 RAM 爲止。這很像是在音樂會中保留 12 張座位，但你不知道有多少朋友要來。如果只來三位朋友，你只需付三張票。（**譯註**：這個比喻適當嗎？我持保留態度）

任何時候，CPU 的 4GB 位址空間中，每一個 4K 區段（一頁）有四種可能的狀態：

- 狀態１：**Available**，表示這頁記憶體並沒有保留給任何人。任何人可以經由配置動作擁有它。企圖讀寫這塊記憶體將會導至 "page fault" 異常情況（exception 0Eh）。稍後我將描述 "page fault" 異常情況。
- 狀態２：**Reserved**，表示這一頁已經被某人要走了。然而，RAM 還沒有被映射到這個位址，也沒有任何硬碟空間保留用來複製其內容。企圖讀寫這塊記憶體將會導至 "page fault" 異常情況（exception 0Eh）。作業系統給予這個區塊的擁有者一個機會，把狀態改變爲 Committed and Present。
- 狀態３：**Committed and present**，表示這位址已經被配置給某人，並且有一個程式已經用它來儲存資料。CPU 的分頁機制也已經把 4K RAM 映射到此位址。讀寫此塊位址也就是讀寫映射之實際記憶體。這個狀態還有一個子狀態稱爲 pagelocked，表示這一頁是 committed、present、並保證絕不會被置換（swapped out）出去。當 page 處於 pagelocked 狀態，永遠會有實際記憶體映射到該 page，直到它變成 unpagelocked 狀態。
- 狀態４：**Committed and not-present**，這和前一狀態十分類似，程式已經配置此塊記憶體並用它來儲存資料。不同之處在於作業系統已經決定，其他地方更迫切需要映射至此區的 RAM 。因此，CPU 已經把此區內容拷貝到硬碟之中，並把此區的每一頁標記爲 Not Present（**譯註**：Not Present 表示位址並沒有映

射到實際記憶體）。

和狀態１、狀態２一樣，如果這樣的 page 被存取，會發生 page fault。不同之處在於當程式存取這樣的位址，作業系統會自動處理 page fault 異常情況並重新映射一塊 4MB RAM 過來。然後作業系統會讀取硬碟中的原來資料，再重新執行發生 page fault 之前的那個指令。於是程式完全不知道有 page fault 發生。這樣透明化地以硬碟模擬 RAM 正是虛擬記憶體的基礎。

Windows 95 提供了一些 *VirtualXXX* APIs，使你得以配置許多 pages 並改變其屬性。例如 *VirtualAlloc*、*VirtualFree* 等等等。稍後我將描述這些函式。

## 記憶體分頁與選擇器（Paging v.s. Selector）

如果你曾經寫過 Windows 3.x 應用程式，你或許會懷疑分頁動作如何與 selector 產生關係。在 Intel CPU 保護模式中執行的 16 位元程式必須使用 selector 才能存取 CPU 位址空間中的某一塊記憶體。每一個 Win16 程式節區（code segment）都關聯到一個 selector；資料節區（data segment）也是如此，從 global heap 中取出的任何一塊記憶體也是如此。只要你寫 Win16 程式，絕對不可能和 selector 老死不相往來。

Selector 中最基本的資料就是它所指向的位址（也就是 "base" 位址）。在 386 機器上，base 總是在 0~(4GB-1) 之間。換句話說一個 selector 有能力指向 CPU 位址空間的任何一處。然而，base 位址是一個線性位址而非實際位址，因此 CPU 的分頁機制係隱藏在 selector 的支配之下。在 Windows 3.1 和 Windows 95 中，16 位元碼並沒有想到分頁和虛擬記憶體，它只是想到有大量記憶體可以用。16 位元的 global heap 管理器從 ring0 作業系統元件中配置大塊記憶體出來，並把它切割爲小塊小塊，使它可以經由 selector 被程式取用。這些 selector 的 base 位址並不需要從 4KB 邊界開始，而且也並非記憶體節區中的每一頁都必須映射到實際記憶體。

如前所述，selector/segment 管理系統並沒有在分頁這一層面上有什麼貢獻。它讓底層的分頁系統提供虛擬記憶體，並假設當它需要時，記憶體會在那兒。***Windows Internals*** 第

2章就描述過 Windows 3.1 之中的 16 位元的 selector/segment 管理系統。這一部份在 Windows 95 之中並沒有改變。

如果你的程式在保護模式中執行，你也不可能避開 selector。存取記憶體時它們是絕對必需品。Windows 95 要求至少在一部 386 機器上跑，而 386 的一個關鍵性質就是你可以製作一個節區（segment），使它橫跨整個 4GB 位址空間。也就是說可以做出一個 selector，base 爲 0，limit 爲 4GB。如果你把這樣的 selector 載入 CS、DS 暫存器中，你就可以忘記所謂「節區」這回事，應用程式可以根據 32 位元偏移值（offset）就指出任何一個地點。這種情況下 32 位元偏移值其實就是線性位址。這種模式（使用 base 爲 0、limit 爲 4GB 的 selector）被稱爲 flat memory model（平滑記憶體模式），它與過去 16 位元的 small、medium、compact、huge 模式都不相同。注意，雖然 flat 模式使得 Win32 程式中似乎不再出現節區，CPU 還是在底層使用節區管理。如果你混合 16 位元碼和 32 位元碼，這一點尤其重要。因爲 16 位元碼無法隱藏醜陋的節區。

有了廣大而開放的節區，程式得以接觸 CPU 位址空間中的任何位置。你可能會奇怪作業系統如何保護其內部資料結構以及其他不可以被應用程式弄亂的地方。對 16 位元程式而言，這並不困難，因爲 selector 決定了一個特定位址範圍，讓程式接觸，而理論上作業系統絕不會給出一個 selector，使其 base 位址存取到它不該存取的地方。然而，Windows 3.1 和 Windows 95 並不阻止你產生自己的 selector 然後進入「遊客止步區」晃盪一番。本章稍後我將反過來利用這個破綻。

如果一個 Win32 程式使用 flat 模式，作業系統如何能夠約束其他程式，不讓它們進入某些位址區域呢？事實上作業系統不再依賴節區大小，而是設定每一頁的屬性。例如，程式不應該盲目地對其程式區域寫入東西，所以作業系統就把程式區的屬性設定爲唯讀。應用程式可以讀它們，但任何寫入動作都會導至 page fault。同樣道理，程式持有一個已被丟棄的指標，就好像把東西寫到一個尚未被配置的 page 一樣。

作業系統把那些尚未被配置的 pages 統統標記爲 not-present。企圖接觸這些位址將導至 page fault。此外，作業系統可以把某一範圍內的 pages 標上 supervisor 屬性，那麼它們

就只能被更高權限等級的碼（也就是作業系統的一部份，以及 VxDs）處理。企圖以較低權限等級的碼處理之，會導至 page fault。如你所見，即使沒有節區，Windows 95 還可以使用分頁來有效保護敏感區域。分頁的唯一缺點就是記憶體配置的最小單位是 4KB，而不是 16 位元節區中的一個 byte。

## Windows 95 之中的 Win32 行程位址空間

Windows 3.x 之中，所有程式都在同一個位址空間中執行。於是任何程式都很容易讀取另一個程式的記憶體。更糟的是，程式還可以改變其他程式的記憶體內容，這就提供給那些有臭蟲的程式一張通往地獄的車票囉。例如，16 位元 Windows 程式甚至在 Windows 95 中可以取得 16 位元 USER DGROUP 的 selector 並隨意寫些垃圾進去。於是視窗系統只好對你說拜拜了。

Windows 95 給予每一行程一個獨立位址空間。所謂「獨立位址空間」，我意思是程式只能看到它自己的記憶體，其他行程所使用的記憶體是不可處理的。更精確地說，Windows 95 記憶體管理器使用 CPU 「以分頁爲基礎」的記憶體管理哲學，確保只有目前行程所擁有的記憶體才會被映射到 CPU 的 4GB 位址空間中。其他行程所擁有的 RAM 並不會出現在目前行程的 page tables 中。這樣做的最大好處是，一個問題程式最多只能破壞它自己，不會影響其他人。

爲恐你對這個 Windows 新性質太過興奮，我必須告訴你，它其實一點也不新。UNIX 行之有年矣，Windows NT 亦復如此。我們只能說，微軟目前主推的桌上型作業系統擁有了高級作業系統的最基礎性質。至於 Win32s，這個 Win32 家族中同父異母的姊妹，並不使用分離位址空間。

雖然把每一個行程的記憶體分隔開來是很重要的，但某些記憶體還是需要被所有行程共享。也就是說所有行程的線性位址空間中的某些頁應該被映射到相同的 RAM 身上。爲什麼？最好的例子就是 system DLLs。每一個行程都需要 KERNEL32.DLL，如果每一個行程都載入一份嶄新的 KERNEL32.DLL，那將是無可置信的超級大浪費。因此

KERNEL32(以及 USER32 等其他 system DLLs)應該駐紮在共享區域中。當 Windows 作業系統切換 page tables 以便執行另一個行程，它會把映射到共享記憶體的那些 page table 留下不動。我將以其他例子說明共享記憶體的必要性。

由於 Windows 95 把不同行程的記憶體都分隔開來，所以對「Windows 95 如何佈置 4GB 位址空間」的任何討論都將離不開所謂的 memory context 觀念。Memory context 基本上是一系列的 RAM pages，以及它們所映射的線性位址。用另一句話說，memory context 是作業系統給予一個行程的線性位址的視野（view）。

每一個行程有它自己一套 memory context。當 Windows 95 排程器將某個行程暫停而讓另一個行程執行，它必須把 memory context 也切換過來。由於每一個行程有自己一套 memory context，所以有時候它又被稱爲 process context。有時候它也被稱爲 address context。不論你把它叫作什麼，記住一點，位址本身沒有意義，除非你指明這個位址在哪一個 memory context 中。

從最上層來看，Windows 95 的 Win32 行程的記憶體佈局十分簡單。在 4GB 位址範圍中，最底部的 2GB(0~7FFFFFFFh)保留給應用程式，2GB 以上(80000000h~FFFFFFFFh)則保留給作業系統。這兩部份又都有細部切割。圖 **5-1** 顯示 4GB 位址空間中的各個細目。如果你有 Windows 95 DDK （譯註：Device Development Kit），請你閱讀線上說明文件中的 "Arenas" 主題中的 “Page Mapping and Address Spaces” 一節。

第一個 4MB 位址空間是給系統虛擬機器中的每一個行程共享。其中位於 1MB 之下的那一部份，內含 MS-DOS 的記憶體映像（memory image），在 Windows 95 啓動時載入。1MB 之下的有趣東西還包括 16 位元 global heap 的較低部份。一如我在 ***Windows Internals*** 第2章所說，Windows 3.1 中的所有 16 位元 heap 節區的線性位址，不是在 1MB 之下就是在 2GB 之上。如果它是以 GMEM_FIXED 屬性配置而得，那麼常常就是在 1MB 之下。你會在位址空間的最初 4MB 中看到許多 16 位元 system DLLs，因爲它們之中有許多（例如 KRNL386）需要「fixed 並且 pagelocked」的記憶體。這是很重要的一點，稍後我還會再討論。

下一個區域是 4MB 到 2GB。這是 Win32 行程所使用的位址空間。每一個 Win32 行程把它自己的碼、自己的資料、自己的資源映射到這將近 2GB 的範圍來。當 memory context 的切換動作發生，其實就是換另一組 pages，映射到這個範圍。除非特別指定，否則映射到此區域的 RAM pages 不能夠被其他行程存取。除了應用程式的碼和資料，它所用到的任何 DLLs 的碼和資料也放在此區。在這裡面你還可以發現應用程式的 heap 和 stack（每一個執行緒有一個 stack）。

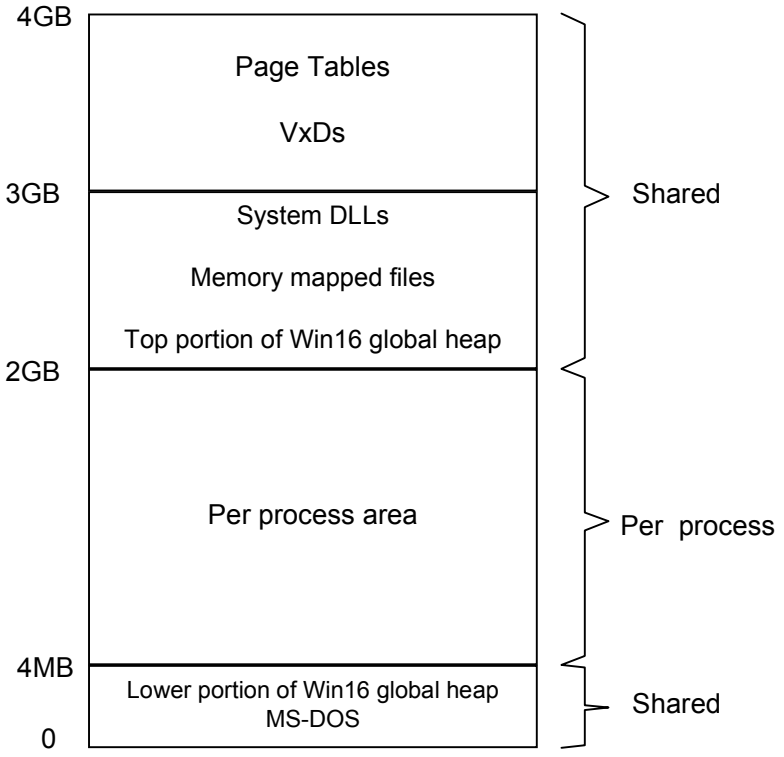



**圖 5-1　Windows 95 的線性位址空間**

Win32 程式預設載入於非常低的位置（4MB）。除非你真的了解分頁動作，否則這樣的概念有點不協調。怎麼能夠有一個以上的程式載入到同一個位址呢？答案是：它們共享了相同的線性位址，但卻不是相同的實際位址。一般而言，行程中的線性位址並不會映射到相同數值的實際位址。由於分頁運算的關係，每一個行程可以認爲自己擁有的是 4MB~2GB 整個空間。它無法看到其他行程的記憶體，其他行程也無法看到它 -- 即令彼此其實享用同一個線性位址。分頁「魔術」使它們在實際上有所區別。

上述規則（爲每一個行程保存個別的 4MB~2GB 位址空間）的例外情況就是：Windows 95 認爲「把相同的實際記憶體開放給同一程式的多份副本（執行個體，instances）共享」是安全的。拿程式碼來說好了，因爲程式通常不會修改其程式碼，如果你執行同一程式的多份副本，那麼 Windows 95 節省記憶體的作法就是：把內含程式碼的實際記憶體映射到每一個程式副本的位址空間中。

從最純淨的作業系統觀點來看，如果每一個 16 位元行程也都有它自己的位址空間，類似 32 位元行程那樣，真是最好不過。不幸的是大量 16 位元程式都依賴「能夠看到其他程式的記憶體」這種能力而生存下去。爲了保留 16 位元程式的相容性，Windows 95 勢必得提供比 Win32 行程更大的權力給它們。Windows NT 3.5 讓每一個 Win16 行程在它自己的位址空間中跑，但是因此消耗更多記憶體並導至更高的複雜性。Windows 95 的設計人員似乎感覺這樣的效益不值得其所付出的代價。

自從我看過 Windows 95，有一個問題就引起我的興趣：16 位元程式如何以不同行程的身份而仍能夠分享其位址空間？結論是，16 位元程式所使用的記憶體總是來自 4MB 以下和 2GB 以上，所謂的可共享區域。

現在讓我們把眼睛移到 4GB 的上半部。從圖 **5-1** 你可以看出它被切割爲兩部份。2GB 至 3GB 之間給所有行程共享，並意圖給 ring3 作業系統碼使用。在這個區域的最低部份，你將發現 16 位元的 global heap。而在它之上，你看到的是記憶體映射檔。這相當有趣，並且值得深思。

如果記憶體映射檔位於可被所有行程共享的區域，很顯然任何行程都可以看到它，甚至不需要對它做映射動作（譯註：指的是 Win32 *MapViewOfFile* 這個動作）。是的，這樣的假設是正確的。在 Windows 95 之中，一個記憶體映射檔可以被所有行程存取得到。這個情況與 Windows NT 不相同。Windows NT 使用更精巧的分頁模式，使記憶體映射檔只能夠被「對此檔案做了映射動作」的行程看到。

2GB~3GB 區域之最上層爲 32 位元 system DLLs（KERNEL32、USER32 等等）的藏身處。爲了保留最多的空間給記憶體映射檔使用，ring3 system DLLs 從 3GB 開始往低處載入。下面是 SoftIce/W MOD 命令的輸出片段，明白表示了這個事實：

```
:mod
hMod   Base      PEHeader        Module Name     EXE File Name
019F   BFF700000 0147:BFF70080   KERNEL32        C:\WINDOWS\SYSTEM\KERNEL32.DLL
01A7   BFF200000 0147:81525AF4   GDI32      C:\WINDOWS\SYSTEM\GDI32.DLL
186F   BFEF00000 0147:81525E98   ADVAPI32        C:\WINDOWS\SYSTEM\ADVAPI32.DLL
1827   BFC000000 0147:815270F0   USER32          C:\WINDOWS\SYSTEM\USER32.DLL
```

其中第二欄是模組的載入位址。KERNEL32 是第一個被載入的 32 位元 system DLL，極端接近 3GB（位址 BFF700000）。接下來是 USER32，位於 BFF200000，並儘量和 KERNEL32 接壤。也許你會以爲這些位址是在載入的時候臨場計算出來的，不，不是這樣。微軟有一個工具程式（Win32 SDK 中的 REBASE.EXE），可以算出一個 DLL 需要多少位址空間，然後算出最佳載入位址，使這些 system DLLs 可以儘量緊密地連接在一起。當這些 system DLLs 被編譯聯結之後（譯註：當然不是被你），微軟接著又修改 DLLs，使它們擁有由 REBASE.EXE 所計算出來的較佳載入位址。這使得所有 systems DLLs 都能夠以最快時間載入，Windows 95 載入器不需要再對它們做「重定位（relocation）」的工作。

Windows 95 位址空間的最後一大塊是 3GB~4GB（C0000000h~FFFFFFFFh）。最後這 1GB 是給 ring0 系統元件（也就是 VxDs）用的。這可以從 SoftIce/W VxD 命令的輸出中觀察得知：

```
: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      C009BD20  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
```

SoftIce/W 的 VxD 命令的完整輸出大約在 360 行以上。我曾經突發奇想地把上述大小加總起來，看看所有的 VxDs 共耗掉多少記憶體。扣除 SoftIce/W 之後，約爲 1MB 左右，其中一些記憶體很可能是可分頁（pageable）的。有許多系統碼隱藏在這 ring0 的天地裡 -- 大部份程式員到達不了的一個神秘地方。

你可能會以爲 Windows 95 使用分頁性質來保護 C0000000h 以上的 VxD 專屬區域，以防阻 ring3 系統碼的刺探或某些笨拙行爲。不，並非如此。KERNEL32 中的許多地方就握有指標，指向 ring0 元件中的變數。VxD 中也有許多地方握有指標，指向 KERNEL32 中的變數，或甚至 KRNL386 中的變數。最糟糕的莫過於 VWIN32.VXD 了，它甚至於開放了兩個 Win32 服務函式（請見第 6 章），其中一個把指向 VWIN32 某位置的指標交給 ring3；另一個函式接受 ring3 位址（在 KERNEL32 和 KRNL386 中）。

## 記憶體共享（Sharing Memory）

Win16 中的所有程式和所有 DLLs 所擁有的所有記憶體都可以被其他程式和 DLLs 存取。這是因爲每一個 Win16 行程使用的都是同一個區域描述表格（local descriptor table，LDT）。因此，行程之間共享記憶體是很輕易的事：只要讓兩個（以上）程式使用相同的 selector 即可。將欲給別人共享之記憶體設定爲 GMEM_SHARE 屬性，其實並非必要。是啊，不必理會微軟信誓旦旦的警告。

現在讓我們比對一下 Windows 95 的記憶體管理，它把每一個 Win32 行程的位址空間都區分開來，除非你特別指定哪一塊要共享。不幸的是，指定共享並不只是像使用 GMEM_SHARE 屬性那麼簡單 -- 事實上在 *GlobalAlloc* 中使用 GMEM_SHARE 屬性是沒有用的。也就是說 GMEM_SHARE 毫無用處：Win16 根本不需要它，因爲每一樣

東西都可共享；Win32 則根本忽略它。

可能你曾經聽一些所謂的 Win32 權威人士說過，在 Windows 95 或 NT 中共享記憶體的唯一方法就是使用記憶體映射檔（memory mapped file）。那的確是一個方法，但不是唯一方法。如果你只是想在同一程式的不同執行個體（instance）中分享小量的記憶體，殺雞何必用牛刀？雖然本書把焦點放在程式與程式之間的可讀/可寫資料的共享，但是別忘了，4GB 位址空間有一半保留給系統使用，它們總是可以被所有行程共享。

從低層來說，所謂記憶體共享，只不過就是把一頁頁的 RAM 映射到一個以上的行程位址空間中。這些 RAM 可以被映射到相同的線性位址，也可以被映射到不同的線性位址。

在 Windows 95 中，經由記憶體映射檔（memory mapped file）而完成的記憶體共享區域，總是在不同的行程中有著相同的線性位址。稍後的 PHYS 程式會揭露此一事實。然而，在你的 Win32 程式中做此假設是很危險的，因爲 Windows NT 並不保證記憶體映射檔在每一個行程中有相同的線性位址。許多 Win32 程式設計書籍都涵蓋有記憶體映射檔這個主題，所以我不打算在這裡說太多。

最簡單的記憶體共享辦法反而沒有被太多人提起。事實上，只要在聯結時指定程式的 data sections 爲 SHARED 屬性，你就可以輕易地在同一程式的每一個執行個體（instance）之間，或是 DLL 的每一使用者之間，共享這份資料。只要將 Win32 DLL 的 data section 指定爲 SHARED，其性質就會像 Win16 DLL 一樣。真幸運，Windows 95 給我們這麼簡單又有彈性的資料共享方式。你可以在 EXE 或 DLL 中產生多個 data sections，把所有你打算共享的資料放到其中一個 data section，然後把它的屬性設爲 SHARED。至於其他的 data sections 仍然使用預設的屬性（nonshared）。PHYS 程式會示範這一切。

一般而言微軟編譯器會把所有初始化過的資料放進一個名爲 .data 的 section 中，然後留給它一個 IMAGE_SCN_MEM_SHARED 以外的屬性。這會使得每當有一個執行個體（instance）產生，該資料就會複製一份資料，專屬給執行個體使用。爲了共享記憶體，你可以要求編譯器產生一個新的 section，名稱隨你取，但只有前８個字元有意義。例如：

```
#pragma data_seg("sharedat")
```

在 #pragma 之後，你可以宣告任何你想要被共享的資料變數。你應該初始化這些資料，否則它們會被編譯器放到另一個專放未初始化資料的 data section 去。

變數宣告完之後，如果你要恢復原來的 data section 屬性，只要加上一行即可：

```
#pragma data_seg()
```

最後，你必須將你的共享心願傳達給聯結器知道。你有兩種方法，傳統作法是在 DEF 檔中設定 section 屬性：

```
SECTIONS
       SHAREDAT  READ WRITE SHARED
```

另一個作法是在聯結器命令列參數中指定屬性。RWS 代表 Read、Write、Shared：

```
LINK /SECTION:SHAREDAT, RWS  <其他的聯結器選項及檔案名稱>
```

我應該告訴你一些「使用者需知」之類的警告。如果你將你的資料初始化爲程式碼或資料符號的指標，那麼當 DLL 被載入於不同行程的不同線性位址上，事情會變得頗爲有趣。看看這個表面上沒有什麼問題的資料宣告（在一個可共享的 data section 中）：

```
int i;
int *AddressOf_i = &i;
```

問題出在 DLL 被載入之前，AddressOf_i 無法確定下來。因此，DLL 必須內含一個待修正記錄（fixup record），告訴載入器記得修正 AddressOf_i 的值。當 DLL 第一次載入，沒有問題。但是如果另一個行程隨後也載入此 DLL，而載入位址卻沒有與前一行程相同的話，由於 AddressOf_i 已被用於第一個行程（它是被共享的，不是嗎），載入器不能夠插手修改 AddressOf_i 的值。於是，對於第二個行程而言，AddressOf_i 的值是錯誤的。利用指標，可以解決這個問題。我可以使用一個非共享的資料變數，內放一個指標指向共享資料。由於此一指標是每個行程皆有一份，所以載入器可以修正其值，使它在每一個行程中都正確無誤。

除了將你的資料分享出來，Windows 95 還可以共享其他記憶體。我已經說過了，2GB 以上全都是共享的。然而，Windows 95 也微微開放了 2GB 以下的一部份區域。如果你執

行一個程式的多份副本（instance），或是在一個以上的行程中使用相同的 DLL，那麼每重複一份碼都是一種浪費。雖然 code section 並沒有 IMAGE_SCN_MEM_SHARED 屬性，Windows 95 還是只載入一份程式碼，然後使用 CPU 的 page table，把程式碼映射到其他的 memory context 之中。

這種分享 code section 的作法很好，唯一例外就是當 DLL 沒有辦法載入到不同行程中的相同線性位址時。假設 FOO.DLL 被兩個行程使用，行程A載入 FOO.DLL 並放到線性位址 X 處。行程B使用另一組 DLLs（其中包括 FOO.DLL）。當行程B載入 FOO.DLL，某些其他的 DLLs 已經佔用了位址 X，於是 FOO.DLL 只好使用其他位址。如果你的程式處於這種情況，解決之道是重新設定 DLL 的基底載入位址，設到一個從沒有被其他行程使用的線性位址上。

## Windows 95 的 "Copy on Write"（寫入時才拷貝）

既然知道 Windows 95 極盡可能地共享程式碼，我們很自然就會關心：除錯器對此如何因應。有什麼問題嗎？噢，除錯器會在你的碼內寫入中斷點（breakpoint）指令（INT 3，opcode 0xCC）。如果除錯器寫入中斷點指令的那個 code page 是被兩個行程共享的話，就會有潛在問題。要知道，除錯器只對一個行程除錯，另一個行程即使碰到中斷點，也不應該受影響。當作業系統看到 INT 3 並且得知該行程並非處於被除錯狀態時，它就把該行程結束掉，因爲這是一種無法處理的異常情況。好，如果 Windows 95 的記憶體管理系統果真如上節我說的那樣，你就沒有辦法對一個「同時被多個行程所使用」的 DLL 除錯了 -- 那樣將無可避免地導至其他行程莫名其妙被結束掉。更別說是對某個執行個體（instance）除錯，而另一個執行個體還能正常運作了。

高級作業系統如 UNIX 之流，對付此問題的方法是所謂的 "copy on write" 機制。一個擁有 copy on write 機制的系統（如 Windows NT），記憶體管理器會使用 CPU 的分頁機制，儘可能將記憶體共享出來，而在必要的時候又將某些 RAM page 複製一份。

給個實際例子會比較清楚一些。假設某程式的兩個個體（instances）都正在執行，共享相

同的 code pages（都是唯讀性質）。其中之一處於除錯狀態，使用者告訴除錯器在程式某處放上一個中斷點（breakpoint）。當除錯器企圖寫入中斷點指令，會觸發一個 page fault（因爲 code page 擁有唯讀屬性）。作業系統一看到這個 page fault，首先斷定是除錯器企圖讀記憶體內容，這是合法的。然而，隨後「寫入到共享的 code page 中」的動作就不應該被允許了。系統於是會先將受影響的各頁拷貝一份，並改變被除錯者的 page table，使映射關係轉變到這份拷貝版。一旦記憶體被拷貝並被映射，系統就可以讓寫入動作過關了。寫入（中斷點）的動作只影響該份拷貝內容，不影響原先內容。

"Copy on write" 並不只在分享程式碼時才派上用場。在 Windows NT 中，可寫入的 data pages 一開始也是唯讀屬性，當應用程式對其中一個 page 寫入資料，CPU 會產生 page fault。作業系統於是把這個 page 改登記爲「可讀可寫」。爲什麼要這麼麻煩呢？因爲如此一來記憶體管理器還是可以把其他唯讀的 data pages 共享給大家。如果稍後有人對這些 data pages 做寫入動作，”copy on write” 機制會拒絕之，並另外提供 RAM pages 給每一個行程。

Copy on write 機制的最大好處就是儘可能讓記憶體獲得共享效益。只有在必要時刻，系統才會對共享記憶體做出新的拷貝。不幸的是，copy on write 機制需要一個精巧的記憶體管理系統，和一個精巧 page table 管理系統，而 Windows 95 還夠不上格，因爲 Windows 95 並非直接在分頁層面支援 copy on write。這對於 Windows 95 早期使用者而言是極大的苦惱，畢竟微軟一直推銷說所有 Win32 程式在 Windows 95 和 NT 上都執行得一樣好。當主要特質（如 ”copy on write”）缺席，「執行得一樣好」這句話可就有漏洞囉。

Windows 95 並不是盲目而愚蠢地就把資料寫入共享記憶體中。由於必須有某些動作以使除錯器能夠工作，Windows 95 支援一個所謂的「copy on write 虛擬機制」。在這個虛擬機制中，當共享記憶體身上出現 page fault，*WriteProcessMemory* 動作就會發生。作業系統首先確定你要寫入的位址是否落於共享記憶體中，如果是，系統會將原來的 pages 拷貝一份，然後把新的 pages 映射到相同的線性位址，然後再進行寫入動作。PHYS 程式證明，copy on write 虛擬機制的確有效地運作著。

雖然 *WriteProcessMemory* 足夠讓除錯器得以對大部份的 DLLs 除錯，它卻不能夠對 2GB 以上的區域除錯。由於 system DLLs 如 KERNEL32 者位於 2GB 之上，所以一般的應用程式除錯器沒有辦法像在 Windows NT 之中那樣地對它們除錯。試看看，在 Windows 95 中啓動你最熟悉的應用程式除錯器，嘗試進入（step into，譯註）一個系統呼叫之中。不論 Visual C++ 除錯器或 Turbo Debugger 都沉默地跳出（step out，譯註）該系統呼叫 -- 甚至即使你是在一個反組譯視窗並要求進入呼叫之中。如果你希望走入 Windows 95 系統碼之中，你需要一個系統層面的除錯器，像是 SoftIce/W 或 WDEB386 之類。

> 譯註：step into 和 step out 都是除錯器上的命令，前者表示要進入一個函式之中，後者表示要跳離目前的函式。

# PHYS 程式

爲了展示我所討論過的所有 Windows 95 記憶體管理細節，我寫了 PHYS 程式。PHYS 沒有亮麗的外表，但是它能夠有效顯示記憶體佈局、共享記憶體、以及 Windows 95 對「copy on write 虛擬機制」的支援。

PHYS 的觀念十分簡單。它尋找並顯示記憶體中各個資料項的線性位址。當你執行 PHYS 的一個個體（instance），它可以顯示 Windows 行程的記憶體佈局。但是這個程式的功能不止於此，它還顯示映射至各線性位址的實際 RAM 位址，及其保護屬性。如果你執行兩個（以上）PHYS 個體，你就可以看出哪一段記憶體被多個行程共享。此外，PHYS 也示範寫入 code page 的前後，位址的變化，證明 *WriteProcessMemory* 的確有效實現了 copy on write。

完整的 PHYS 原始碼放在書附磁片之中。最主要的動作顯示於**列表 5-1**。*ShowPhysicalPages* 計算各個記憶體物件的線性位址和實際位址，並把它們顯示出來。然而，PHYS 並不打算顯示其位址空間中的所有記憶體物件，它只顯示幾個我認爲重要的物件，這幾個物件對認識行程的記憶體佈局十分有幫助。

```
#0001  void ShowPhysicalPages(void)
#0002  {
#0003     DWORD linearAddr;
#0004     MEMORY_BASIC_INFORMATION mbi;
#0005
#0006     //
#0007     // Get the address of a 16 bit DLL that's below 1MB (KRNL386's DGROUP)
#0008     //
#0009     linearAddr = Get_KRNL386_DGROUP_LinearAddress();
#0010     printf( "KRNL386 DGROUP     - Linear:%08X  Physical:%08X  %s\n",
#0011           linearAddr,
#0012           GetPhysicalAddrFromLinear(linearAddr),
#0013           GetPageAttributesAsString(linearAddr) );
#0014
#0015     //
#0016     // Get the starting address of the code area.  We'll pass VirtualQuery
#0017     // the address of a routine within the code area.
#0018     //
#0019     VirtualQuery( ShowPhysicalPages, &mbi, sizeof(mbi) );
#0020     linearAddr = (DWORD)mbi.BaseAddress;
#0021     printf( "First code page    - Linear:%08X  Physical:%08X  %s\n",
#0022           linearAddr,
#0023           GetPhysicalAddrFromLinear(linearAddr),
#0024           GetPageAttributesAsString(linearAddr) );
#0025
#0026     //
#0027     // Get the starting address of the data area.  We'll pass VirtualQuery
#0028     // the address of a global variable within the data area.
#0029     //
#0030     VirtualQuery( &callgate1, &mbi, sizeof(mbi) );
#0031     linearAddr = (DWORD)mbi.BaseAddress;
#0032     printf( "First data page    - Linear:%08X  Physical:%08X  %s\n",
#0033           linearAddr,
#0034           GetPhysicalAddrFromLinear(linearAddr),
#0035           GetPageAttributesAsString(linearAddr) );
#0036
#0037     //
#0038     // Get the address of a data section with the SHARED attribute
#0039     //
#0040     MySharedSectionVariable = 1;    // Touch it to force it present
#0041     linearAddr = (DWORD)&MySharedSectionVariable;
#0042     printf( "Shared section     - Linear:%08X  Physical:%08X  %s\n",
#0043           linearAddr,
#0044           GetPhysicalAddrFromLinear(linearAddr),
#0045           GetPageAttributesAsString(linearAddr) );
```

```
#0046
#0047     //
#0048     // Get the address of a resource within the module
#0049     //
#0050     linearAddr = (DWORD)
#0051           FindResource(GetModuleHandle(0), MAKEINTATOM(1), RT_STRING);
#0052     printf( "Resources          - Linear:%08X  Physical:%08X  %s\n",
#0053           linearAddr,
#0054           GetPhysicalAddrFromLinear(linearAddr),
#0055           GetPageAttributesAsString(linearAddr) );
#0056
#0057     //
#0058     // Get the starting address of the process heap area.
#0059     //
#0060     linearAddr = (DWORD)GetProcessHeap();
#0061     printf( "Process Heap       - Linear:%08X  Physical:%08X  %s\n",
#0062           linearAddr,
#0063           GetPhysicalAddrFromLinear(linearAddr),
#0064           GetPageAttributesAsString(linearAddr) );
#0065
#0066     //
#0067     // Get the starting address of the process environment area.
#0068     //
#0069     VirtualQuery( GetEnvironmentStrings(), &mbi, sizeof(mbi) );
#0070     linearAddr = (DWORD)mbi.BaseAddress;
#0071     printf( "Environment area    - Linear:%08X  Physical:%08X  %s\n",
#0072           linearAddr,
#0073           GetPhysicalAddrFromLinear(linearAddr),
#0074           GetPageAttributesAsString(linearAddr) );
#0075
#0076     //
#0077     // Get the starting address of the stack area.  We'll pass
#0078     // the address of a stack variable to VirtualQuery
#0079     //
#0080     VirtualQuery( &linearAddr, &mbi, sizeof(mbi) );
#0081     linearAddr = (DWORD)mbi.BaseAddress;
#0082     printf( "Current Stack page  - Linear:%08X  Physical:%08X  %s\n",
#0083           linearAddr,
#0084           GetPhysicalAddrFromLinear(linearAddr),
#0085           GetPageAttributesAsString(linearAddr) );
#0086
#0087     //
#0088     // Show the address of a memory mapped file
#0089     //
#0090     linearAddr = (DWORD)PMemMapFileRegion;
#0091     printf( "Memory Mapped file  - Linear:%08X  Physical:%08X  %s\n",
```

```
#0092            linearAddr,
#0093            GetPhysicalAddrFromLinear(linearAddr),
#0094            GetPageAttributesAsString(linearAddr) );
#0095
#0096     //
#0097     // Show the address of a routine in KERNEL32.DLL
#0098     //
#0099     linearAddr = (DWORD)
#0100        GetProcAddress( GetModuleHandle("KERNEL32.DLL"), "VirtualQuery" );
#0101     printf( "KERNEL32.DLL       - Linear:%08X  Physical:%08X  %s\n",
#0102            linearAddr,
#0103            GetPhysicalAddrFromLinear(linearAddr),
#0104            GetPageAttributesAsString(linearAddr) );
#0105  }
```

**列表 5-1　PHYS.EXE 程式中的 ShowPhysicalPages 函式**

PHYS 所顯示的記憶體物件包括：16 位元 DLL 中的一個函式、一個記憶體映射檔、32 位元 DLL 中的一個函式、PHYS 的 heap、stack、程式碼、資料、資源、共享記憶體。我選擇 KRNL386 的 DGROUP，告訴大家 Win16 DLLs 其實是映射到一個 Win32 程式的位址空間中（如果不是這樣，下移（thunk down，譯註）就很難實現）。至於 KERNEL32 和記憶體映射檔，可以證明它們位於共享的 ring3 碼區域（2GB~3GB）。

> 譯註：所謂 thunk down，係指 32 位元碼透過某些轉換層，呼叫到 16 位元碼。所謂 thunk up，則指 16 位元碼透過某些轉換層，呼叫到 32 位元碼。

**圖 5-2** 顯示執行兩個 PHYS 個體（instances）的結果。你必須遵循下面的執行步驟：先執行第一個 PHYS，當它出現 Press any key...，再執行第二個 PHYS。這樣可以保證兩個 PHYS 同時存在。最後，回到第一個 PHYS，按下任意鍵，獲得第一個 PHYS 的後半部輸出。

現在，我們集中焦點在第一個 PHYS 的上半部輸出結果。這些位址以線性位址爲主，依序排列。請檢查線性位址和實際位址之間的關聯性。理不出頭緒，對不對？是的，Windows 95 持有的一大缸子 RAM pages 並不一定映射到任何特定的線性位址。

輸出結果中的第一項是 KRNL386 的 DGROUP。接下來四個項目分別是 PHYS.EXE 的四個 sections。稍早我曾說過，在 Windows 95 之中，一個 32 位元行程的預設載入位址是 4MB（0x400000）。如果你利用第 8 章的 PEDUMP 工具程式列出 PHYS.EXE 的表頭，你會發現 code section 從相對虛擬位址（relative virtual address，RVA）0x1000 開始。把 4MB 加上 0x1000，正是 PHYS 的 code section 起始位址 0x401000。你還可以驗證 data section、shared data section、resource section，把它們的 RVA 加上 4MB，就是 PHYS 的線性位址輸出結果。

下一個輸出項目是 PHYS 的 heap，位於 0x410000。它距離上一個位址並不太遠，看來 KERNEL32 似乎是以「由下而上」（bottom-up）的方式配置線性位址。預設的 heap 大小是 1MB+4KB，因此下一個位址應該是 0x511000。但因爲 Windows 95 對每一個記憶體物件的線性位址是以 64KB 爲邊界，所以下一個位址應該是 0x520000。啊哈，那不正是 PHYS 的環境區起始位址嗎！看來「由下而上」的配置方式果然沒錯。

絕大部份的環境區不會擁有 64K 的字串，但規則就是規則，所以下一個位址應該是在 64KB 之後，也就是 0x530000。但是我們卻看到 PHYS 的 stack 位於 0x63F000。乍見之下似乎違反由下而上的配置方式。稍做思考之後，我們發現其實沒有違反。記得嗎，stack 的成長是由高位址而低位址，所以我們必須將 stack 的頂部減去 stack 的大小，才是 stack 的起始位置。如果目前的 stack 是在 0x63F000，而如果我們沒有使用太多的 stack 空間，那麼 stack 區應該是到 0x640000。PHYS 的預設 stack 大小是 1MB，所以 0x640000 減去 1MB 之後得到 0x540000。那比起我所猜測的 0x530000 還多出 64KB。如果我呼叫 *VirtualQuery*，詢問 stack 位址，*VirtualQuery* 傳回的 AllocationBase 值是 0x530000。很明顯地，載入器計算程式所需的 stack 大小，並以 64KB 爲單位，因而爲 PHYS 配置了 1MB+64KB（而非 1MB）的 stack 空間。從這裡我們可以看出，由下而上的配置方式仍然是適用的。

```
// 以下是第一個 PHYS 的輸出
//
***** FIRST INSTANCE *****
KRNL386 DGROUP      - Linear:00036F60  Physical:00245F60  Read/Write USER
First code page     - Linear:00401000  Physical:0247D000  ReadOnly USER
First data page     - Linear:00407000  Physical:02477000  Read/Write USER
Shared section      - Linear:0040A000  Physical:02470000  Read/Write USER
Resources           - Linear:0040C088  Physical:0248B088  ReadOnly USER
Process Heap        - Linear:00410000  Physical:02486000  Read/Write USER
Environment area    - Linear:00520000  Physical:02484000  Read/Write USER
Current Stack page  - Linear:0063F000  Physical:02490000  Read/Write USER
Memory Mapped file  - Linear:82DCC000  Physical:02ADD000  Read/Write USER
KERNEL32.DLL        - Linear:BFF9CECB  Physical:014B8ECB  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:0248A000  Read/Write USER
First data page     - Linear:00407000  Physical:02477000  Read/Write USER
Shared section      - Linear:0040A000  Physical:02470000  Read/Write USER
Resources           - Linear:0040C088  Physical:01D53088  ReadOnly USER
Process Heap        - Linear:00410000  Physical:02486000  Read/Write USER
Environment area    - Linear:00520000  Physical:02484000  Read/Write USER
Current Stack page  - Linear:0063F000  Physical:02490000  Read/Write USER
Memory Mapped file  - Linear:82DCC000  Physical:02ADD000  Read/Write USER
KERNEL32.DLL        - Linear:BFF9CECB  Physical:014B8ECB  ReadOnly USER

// 以下是第二個 PHYS 的輸出
//
***** SECONDARY INSTANCE *****
KRNL386 DGROUP      - Linear:00036F60  Physical:00245F60  Read/Write USER
First code page     - Linear:00401000  Physical:0247D000  ReadOnly USER
First data page     - Linear:00407000  Physical:0240E000  Read/Write USER
Shared section      - Linear:0040A000  Physical:02470000  Read/Write USER
Resources           - Linear:0040C088  Physical:0248B088  ReadOnly USER
Process Heap        - Linear:00410000  Physical:02450000  Read/Write USER
Environment area    - Linear:00520000  Physical:02440000  Read/Write USER
Current Stack page  - Linear:0063F000  Physical:0246B000  Read/Write USER
Memory Mapped file  - Linear:82DCC000  Physical:02ADD000  Read/Write USER
KERNEL32.DLL        - Linear:BFF9CECB  Physical:014B8ECB  ReadOnly USER
Press any key...
```

**圖 5-2　兩個同時執行的 PHYS 程式的輸出結果。**

接下來 PHYS 顯示的是它所產生的記憶體映射檔的位址。基底位址在 0x82DCC000，比 shared ring3 region 的底線（2GB）高出 45MB。由於 2GB~3GB 區域映射給所有行程，所以每一個行程都可以看到任何記憶體映射檔。是的，甚至於那些並沒有對記憶體映射檔做出映射動作的行程也是。這是 Windows 95 之中潛在性的「錯誤指標」問題之所在。

Windows NT 擁有較精巧的記憶體管理器，不允許這種嚴重破壞位址空間隱私的行爲。

最後一項是 KERNEL32 中的 *VirtualQuery* 位址。位址 0xBFF9CECB 十分接近 3GB，爲什麼在這麼高的位置呢？Windows 95 設定 system DLLs 的基底位址儘可能靠近 3GB，以便盡量保留 2GB~3GB 之間的空間給記憶體映射檔使用。你也可以自己驗證一下 USER32 和 GDI32 的位址。

## 以 PHYS 檢驗共享記憶體

爲了看看 Windows 95 讓行程們共享哪些記憶體，我執行兩個 PHYS，並比較其輸出。這也是**圖 5-2** 爲什麼結合兩份輸出的原因。讓我們將第一個 PHYS 的第一組輸出和第二個 PHYS 的輸出做一比較。擁有相同實際位址者，就是共享記憶體。爲了更清楚說明比較結果，我在**圖 5-3** 中列出共享和非共享的項目。

| 共享記憶體 | 非共享記憶體 |
|---|---|
| KRNL386 DGROUP | First data page |
| First code page | Process Heap |
| Shared section | Environment area |
| Resource | Current stack page |
| Memory mapped file | |
| KERNEL32.DLL | |

**圖 5-3 一個 Win32 程式同時執行兩份個體（instances），其共享與非共享部份。**

共享部份應該不會令你驚訝。KRNL386 DGROUP 和 KERNEL32 都是 system DLLs，你當然希望它們被共享。PHYS 的程式碼和資源也被共享，意味著 Windows 95 希望以最佳效率運用記憶體。另外，PHYS 刻意產生兩個共享區域，一是 shared section，一是

記憶體映射檔。非共享部份應該也不會令你太驚訝，其中所有項目都是可讀可寫的程式資料。如果 Windows 95 連這些資料都要共享，只要執行多份 PHYS 就會導至系統崩潰掉。

## 以 PHYS 檢驗 "Copy on Write" 性質

PHYS 的最後一項功能是展示由 *WriteProcessMemory* 提供的 copy on write 虛擬機制。請仔細看看下面三行輸出：

```
***** FIRST INSTANCE *****
First code page    - Linear:00401000  Physical:0247D000  ReadOnly USER
...
Now modifying the code page
First code page    - Linear:00401000  Physical:0248A000  Read/Write USER
...
***** SECONDARY INSTANCE *****
First code page    - Linear:00401000  Physical:0247D000  ReadOnly USER
...
```

爲了讓輸出結果看起來合理，你必須謹慎遵守兩份 PHYS 的執行程序。上述第一行和第三行來自不同的 PHYS 輸出，是對 code page 做寫入動作之前的狀態。它們的實際位址都是 0247D000，證明這塊記憶體的確是被共享的。中間那一行是在第一份 PHYS 以 *WriteProcessMemory* 函式將資料寫入 code page 之後的狀態，是否注意到它有一個不同於前的實際位址？這表示 *WriteProcessMemory* 會把底層的 RAM page 轉換掉。雖然這裡沒有顯示出來，不過我可以告訴你，此時第二份 PHYS 的 First code page 實際位址仍然是 0247D000。

## PHYS 程式中的酷刑

潛伏在 PHYS 表面下的，是一些低階的系統碼，微軟可能不希望你知道那麼多。在一個良好設計的作業系統之中，應用程式不應該能夠處理實際記憶體和線性位址之間的對映關係。一般而言也沒有需要這麼做。但這是 PHYS 程式機能的核心。由於 Windows 95 並未提供方法讓我們獲得 page mapping 的關係，PHYS 只好繞過作業系統自己來。一部份的 PHYS 碼走在刀鋒邊緣，促使在 ring0（CPU 的最高權限等級）中執行。一般應用程式係在 ring3 執行，未經作業系統的謹慎控制，不可能進入 ring0。由於 PHYS 所需的

ring0 碼未經作業系統核准，我必須寫一個一般性的機制，讓 ring3 Win32 程式可以呼叫 ring0 碼。 你可以輕易修改 PHYS ring0 碼，放入你自己的應用程式中。

爲了把線性位址映射到實際位址，*GetPhysicalAddrFromLinear* 函式必須 "party with" page tables。"party" 是微軟的官方術語，意思是做某些你不應該做的事。Page tables 是一個複雜的主題，我將在下一節 "Memory Contexts" 中描述它。如果現在你不知道什麼是 page tables，只要把它想像是一個資料結構，用來描述線性位址與實際位址之間的對映關係即可。Page tables 由作業系統維護，給 CPU 使用。打開 CPU 手冊，你會發現 page directory 由暫存器 CR3 指出。不幸的是，你必須有很高的權限才能取出 CR3 的值。企圖在 ring3 取 CR3 的值只會導至一個 general protection fault（異常情況 0Dh）。當 Windows 95 看到這個異常情況，它會分析這個指令並發現後者並沒有足夠的權限。Windows 95 並不會把發出這個指令的程式結束掉，它只是無聲無息地把控制權交還給該應用程式。但，當然不含 CR3 的值。

這意味什麼？Windows 95 不讓應用程式偷襲 page tables。當然啦，我可以寫一個 VxD（它在 ring0 執行）取出 CR3 值，但是我不喜歡我的系統中圍繞太多的 VxDs。此外，即使我能夠取得 CR3 值，還有一個大問題。CR3 值代表一個實際位址，卻沒有什麼好方法可以把實際位址轉換爲線性位址（PHYS 只能使用線性位址）。除非我把分頁機制關閉（譯註：於是實際位址等於線性位址），否則我似乎不能對 CR3 值做點什麼。

另一個想法是，看看 Windows 95 能不能夠把 page tables 映射爲 ring3 碼能夠直接使用的線性位址。我們知道整個 4MB page tables 總是被映射在線性位址 0xFF800000 處（距離線性位址頂端 8MB）。那麼你是不是可以產生一個指標，指向該處，就可以直接讀取 page tables 的內容？不，沒這麼好，這些表格並非如你所想像般地沒有保護。page directory 以及每一個 page table 中的每一筆資料都有一個 user/supervisor 位元，指示「任何權限等級的碼都可以存取它」或是「只有 ring0 碼才可以存取它」。每一個 page table 的 user/supervisor 位元都是 0，表示整個 4MB page tables 對於 ring3 碼而言是個禁地。

由於整個 4MB page tables 對於 ring3 碼而言是個禁地，我們勢必得將我們的碼在 ring0 執行，才能取得 page tables。我曾經在 1993 年五月的 ***Microsoft Systems Journal*** 發表一篇有關於 ring 權限等級的文章，並且寫了一個 RIN0.EXE。RING0 使用了 Windows 記憶體管理的一些技巧，從一個 ring3 Windows 程式中呼叫 ring0 16 位元碼。重點其實在於所謂的 CPU call gates，它提供一種讓「低權限的碼呼叫高權限的碼，例如 ring3 呼叫 ring0」的方法。由於 Windows 並沒有給你這樣一個 call gate，RING0 就自行進入 LDT 並產生一個 call gate。爲了進入 LDT，RING0 和 KRNL386 一樣，使用 INT 2Fh 子功能。

RING0 問世之後，Alex Schmidt 寫了一篇很棒的文章（***Dr. Dobb's Journal***，1994 年三月），擴充 RING0，使它能夠呼叫 32 位元 ring0 碼。Alex 甚至於走得更遠，使用這些 call gate 技巧寫了一個可以動態載入的 VxD（譯註：彼時 Windows 95 還在 "Chicago" 時代，還沒有動態載入 VxD 的能力）。幸運的是現在的 Windows 95 已經能夠支援 VxD 動態載入，不再需要像我和 Alex 使用的那些可怕動作。當我看到 PHYS 需要呼叫 ring0 碼，我就想到要把原來的 RING0 程式升級，使它適用於 Win32 程式。也就是說我們需要一個 32-bit call gate，而不是 16-bit call gate。其結果就是本書所附的 PHYS。

在一個 Win32 程式中以一般化的機制呼叫 ring0 碼，當然是有點棘手，但不會太棘手。**圖 5-4** 所列的 *GetPhysicalAddressFromLinear* 就是個好例子。首先，你必須呼叫 *GetRing0Callgate* 以產生一個 call gate selector。*GetRing0Callgate* 函式將在稍後的 32 位元區討論之，它接受兩個參數，第一個是「你所希望執行的 ring0 碼」的 32 位元線性位址，第二個是 DWORD 參數的個數，它們將被壓到 stack 之中，供 ring0 碼取用。

一旦你擁有 call gate selector，下一個動作就是把它儲存在一個 6 位元組的遠程指標中（也就是一個 FWORD）。6 位元組？是的，在 32 位元模式中，遠程呼叫是藉由一個 16 位元 selector 和一個 32 位元偏移值。偏移值是 32 位元，暗喻 selector 將是針對 32 位元節區而非 16 位元節區，這就有點像 Win32 程式的 flat 定址模式。我們希望利用這個 call gate selector 做出一個遠程呼叫，爲的是讓 CPU 切換至 ring0。在**圖 5-4** 中，call gate selector 被儲存在 6 位元組的陣列中的較高三個 WORDs。指標的偏移值並不重要，因

爲 CPU 會忽略它，並從 call gate 描述器所記錄的偏移值中載入 EIP。產生這個指標之後，程式碼改用行內組譯器（inline assembler）呼叫此一指標（因爲 C 編譯器只知道 32 位元近程呼叫）。我在呼叫 call gate 的動作前後包夾了 cli 和 sti 兩個指令，爲的是避免在 ring0 發生中斷（interrupts）。

```
#0001  DWORD GetPhysicalAddrFromLinear(DWORD linear)
#0002  {
#0003      if ( !callgate1 )
#0004          callgate1 = GetRing0Callgate( (DWORD)_GetPhysicalAddrFromLinear, 1 );
#0005
#0006      if ( callgate1 )
#0007      {
#0008          WORD myFwordPtr[3];
#0009
#0010          myFwordPtr[2] = callgate1;
#0011          __asm   push    [linear]
#0012          __asm   cli
#0013          __asm   call    fword ptr [myFwordPtr]
#0014          __asm   sti
#0015
#0016          // The return value is in EAX.  The compiler will complain, but...
#0017      }
#0018      else
#0019          return 0xFFFFFFFF;
#0020  }
```

**圖 5-4　PHYS.EXE 中使用 32 位元 call gate。**

從 Win32 程式中進入 ring0 有一些奇特的需求。爲了某些因素，我必須寫一個 PAGETABL.ASM。第一，16:32 遠程呼叫會使得 CPU 放置 8 個位元組到 stack 中，而不是傳統的 4 個。因此在設定 EBP frame 之後，第一個參數是 EBP+0Ch 而不是 EBP+08。更重要的是，欲回返至 ring3 時，需要一個 16:32 RETF 而不是 32 位元的近程回返。和 16:32 遠程呼叫一樣，編譯器也不知道怎麼產生一個 16:32 RETF。

現在讓我做一個整理。當你要從 Win32 程式中呼叫 ring0，第一步是寫 ring0 碼，並注意上述的警告。接下來在 Win32 碼中呼叫 *GetRing0Callgate*，把你的 ring0 函式名稱及

其參數個數傳過去。然後根據這個 call gate 產生一個 16:32 遠程指標，並呼叫之。最後，當你不再需要那個 ring0 函式，呼叫 *FreeRing0Callgate* 釋放即。整個過程並不是很精緻，但總比全部由作業系統支配好。

# Memory Contexts

雖然抽象說明 memory contexts 也是不錯，不過有時候來點實務經驗更好。Windows 95 必須維護一些資料結構，用以記錄哪一頁的 RAM 映射到行程的哪一塊線性位址。爲了了解 Windows 95 的 memory contexts，你必須了解 CPU 的分頁機制。我將帶你快速瀏覽 80386 的分頁機制，至於更先進的細節就省略不提了。如果你對分頁有興趣，請參考 Intel 手冊或其他 386 架構的書籍。

80386 級的 CPU 使用雙層查詢表格，將一個線性位址轉換爲一個實際位址，再送往位址匯流排（address bus）。第一層查詢表格稱爲 page directory，有 4KB 那麼大，可視爲 1024 個 DWORDs 組成的陣列。每一個 DWORD 內含一個實際位址，指向一個名爲 page table 的 4KB 空間 -- 它同樣也是 1024 個 DWORDs 組成的陣列，每一個 DWORD 內含一個實際位址，指向 4KB 實際記憶體（RAM）。

爲了使用 page directory 和 page table，CPU 把 32 位元線性位址切割爲三部份，如圖 **5-5** 所示。最高 10 個位元給 CPU 當做 page directory 的陣列索引，選出一個 page table。接下來的 10 位元則當做該 page table 的陣列索引，選出一筆資料，內含 4KB RAM 的起始位址。最後 12 個位元則用來做爲這 4KB RAM 的偏移值，精確指出一個位元組。

CPU 到哪裡找出 page directory？CR3 暫存器是也！這是 80386 所引進的一個特殊暫存器。memory contexts 的最粗糙生產方式就是爲每一個行程產生一個 page directory 以及 1024 個 page tables，然後在適當時刻改變 CR3 暫存器內容，使它指向當時行程的 page directory。

這種作法的問題是，爲了映射整個 4GB 位址空間，你需要 1024 個 page tables，每個

大小是 4KB。每一個行程光爲這個就耗掉 4MB 記憶體，不符合經濟效益。Windows 95 的作法是只維護單獨一塊 4MB 區域當做 page tables，並時時修改 page directory 中的資料項，使 CPU 能夠快速改變 pages 的映射。

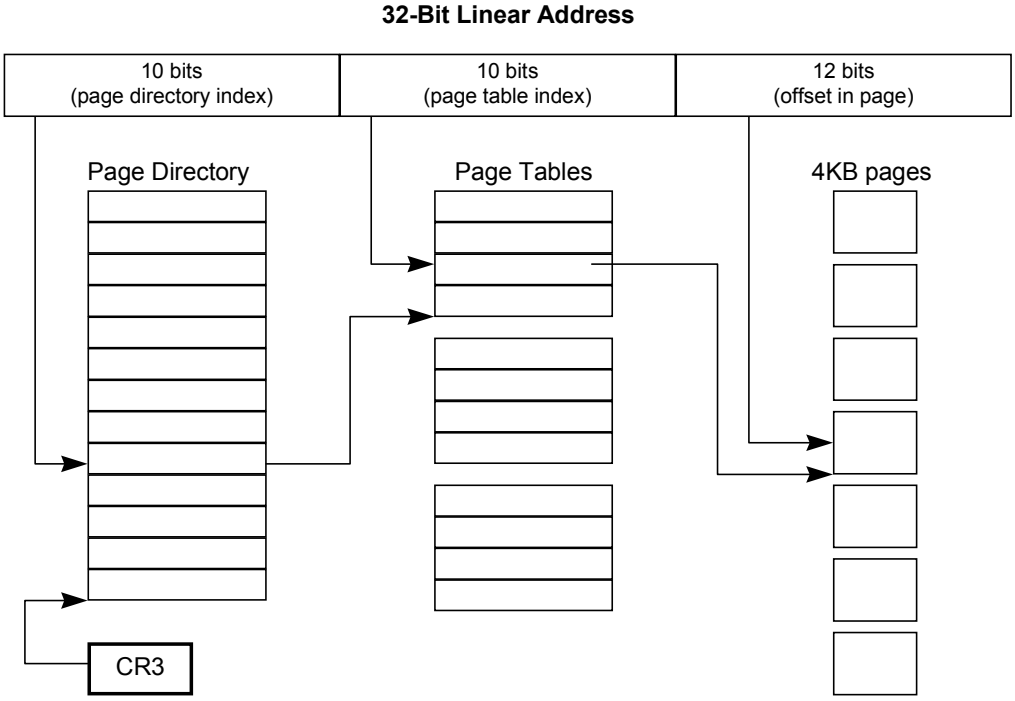



**圖 5-5　CPU 如何把一個線性位址轉換爲一個實際位址**

也許你擔心，光爲了分頁就用掉 4MB，是不是太多了些。噢，不必擔心，作業系統可以藉由 page directory 這一層，告訴 CPU 說某一個 page table（佔用 4KB）不在記憶體中（not present），於是就可以省下 4KB RAM。Page directory 和 page tables 很少真正使用將近 4MB 的實際記憶體，但它們的確是使用 4MB 位址空間，就從 FF800000h 開始。Page directory 也位於這 4MB 之中。利用 SoftIce/W 可以觀察得到它們。

你可以輕易找出 page directory 的線性位址：只要利用 SoftIce/W 的 CR 命令取出 CR3 值。在我的機器上，CR3 爲 6EE000h。這是一個實際位址，所以你必須先將它轉換爲線性位址才能夠在程式中使用。SoftIce/W 的 PHYS 命令可以輕易完成此事，它會搜尋所有的 page tables，找出所有與「你所指定之實際位址」有映射關係的線性位址。下達 PHYS 6EE000h 命令，我獲得兩個線性位址，其中第二個是 FFBFE000h，正落於保留給 page tables 使用的 4MB 位址空間中。

既然我們能夠藉由 SoftIce/W 找到 page directory，我們應該能夠在 page directory 中設立一個硬體寫入中斷（write breakpoint），以證明或推翻前面我所說的有關於 memory context switching 的論點。如果中斷點沒有響起，表示 context switching 或許是由其他方法完成。如果中斷點確實響起，表示 context switching 的確是因爲對 page tables 的操作而完成，同時，前述的寫入位址也可以給我們一個線索，讓我們更清楚 context switching 的反應。

我在 SoftIce/W 上做了個小小實驗，證實 page directory 的確會被改寫。爲了觀察，我回頭看看寫入動作發生前的一些指令，如下所示：

```
_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* 的核心動作是 REPZ MOVSD 和 REPZ STOSD 兩個指令。REPZ MOVSD 之前的三個 MOV 指令用來設定某個東西，用以將一塊記憶體內容從某處拷貝到另一處。拷貝的對象是 FFBFE000h，正是我們稍早所見的 page directory 起始位址。這表示，此一常式產生一組新的 page tables，映射到 page directory 之中。它所拷貝的每一個 DWORDs 都對應於 page tables（最大可能爲 1024 個）中的一個。

另一件有趣事情是，被搬移的 DWORDs 個數並未寫死。相反地，程式碼載入 ECX，內含 DWORDs 個數。第二個指令 REPZ STOSD 的效果並不明顯，它用來比較「這一次被拷貝的 DWORDs 個數」和「前一次 *_ContextSwitch* 被呼叫時所拷貝的 DWORDs 個數」。如果本次比前次少，表示有一些 page tables 是前一個 memory contexts 專屬的，新的 memory context 不應該看到。如有必要，REPZ STOSD 會把其他的 page directory 資料項（譯註）標記爲 "non-present"。

> 譯註：我所謂的「page directory 資料項」就是 "page directory entries"，有人簡稱爲 PDE。至於 page table entries，有人簡稱之爲 PTE。

SoftIce/W 很好心地把 *_ContextSwitch* 標記放在程式列表的頂端。*_ContextSwitch* 是 VMM 的一個 services ，其位址出現在 VMM services 表格中，此表格係由 VMM 的 Device Descriptor Block（DDB）的一個欄位指出。SoftIce/W 如何知道這個 service 的名稱呢？請看 Windows 95 DDK 中的 VMM.INC。每一行若以 VxD_Service 起頭，就是 VMM VxD 的 service。接近底部的地方你會看到 *_ContextSwitch*。另兩個鄰近的 services：*_PageModify* 和 *_PageModifyPermissions*，也很有趣。

我們發現，Windows 95 必須保持一組 pages，以及一個「頁數」值，給每一個 memory context 使用。再一次我們可以利用 SoftIce/W 的 Addr 命令驗證之：

```
: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
```

在這個列表之中，FREECELL、WINMINE、MMTASK 以及 HEAPWALK 都是 Win16 程式。有趣的是，即使 Win16 程式可以彼此看到對方，Windows 95 一樣以分離的行程待之，並以不同的 memory context 伺候。然而這只是一種理論，因爲 Win16 的程式碼和資料節區總是被載入到共享區域（0~4MB 以及 2GB 以上）。因此，Win16 程式總是能夠看到彼此，甚至雖然技術上它們擁有不同的 memory context。

上述列表中的其他行程都是 Win32 行程。Tables 欄位容易引起誤會，它其實是指構成該 memory context 所需的 page tables 個數。每一個 page directory 映射 1024 個 page tables，每一個 page table entry（PTE）映射一個 4KB 區域。因此，每一個 page directory entries（PDE）映射 4MB 線性位址空間。請注意 16 位元程式只使用兩個 page tables，這是因爲 16 位元程式不需要 Win32 行程區域（0x00400000~0x7FFFFFFF）。至於 Win32 行程就一定需要這個區域了，不過其中大部份也都是 "not present"。

每一個 memory context 的 handle 看起來和線性位址頗相像。讓我們傾印出其中一個 handle 所代表的記憶體內容。我選擇第一個 handle（C0FE5D04）：

```
:dd c0fe5d04
0030:C0FE5D04  C103C6F8  00000004  C0FD4D1C  C103C9B0    .........M......
```

唔…我們可以輕易地把前兩個 DWORDs 和剛剛 SoftIce/W 的 addr 輸出產生關聯。第一個 DWORD（C103C6F8）是 addr 輸出中的 PGTPTR（Page Table Pointer）欄位值。第

二個 DWORD（00000004）是 Tables 欄位值。回頭看看 *_ContextSwitch* 的碼，你會發現 *_ContextSwitch* 希望獲得一個指標，其格式正是這裡的格式：一個指標（指向欲拷貝之各個 page directory entries，PDEs），後面緊跟著待拷貝的頁數。

第四個 DWORD（C103C9B0）也很容易在 addr 輸出中找到。它正是下一個 context 的 handle。事實上 contexts 的確是一個串列。剩下的第三個 DWORD（C0FD4D1C）呢？它看起來像是一個指標，所以讓我把其內容傾印出來看看：

```
:dd c0fd4d1c
0030:C0FD4D1C  00000400  0007FFFF  C0E0E310  C0E0E31C    ................
```

多麼有趣啊，如果你把前兩個 DWORDs 分別乘以 0x1000（一頁的大小），就得到 addr 輸出中的 Min Addr 和 Max Addr 兩欄位內容。看來我們已經完全挖出了 Windows 95 中的 contexts 管理系統的秘密。

如果你對於深掘 Windows 95 的 memory context 有興趣，那麼 DDK 絕對不可或缺。和 SDK 文件不同，DDK 文件並不企圖隱藏某些資訊。DDK 說 memory context 由 VMM.VXD 的 *_CreateContext* 產生，由 *_DestroyContext* 摧毀掉。只要寫 VxD，你就可以真正產生、切換、摧毀你自己的 memory contexts。當然啦，讓 Windows 95 知道你在幹什麼是更重要的工作。

其他的 VMM 酷哥函式還包括 *_CopyPageTable* 和 *_PageAttach*。前者讓你獲得 memory context 中的邏輯位址/實際位址之間的對應，從此不需要再像我一樣在 PHYS 中走入 page tables。後者的函式文件中說，它被用來把一個 context 中的記憶體映射到另一個 context 的相同線性位址中。Windows 95 就是使用這個機制才能在同一程式的多個執行個體之間共享程式碼和資料。

# Windows 95 的記憶體管理函式

Windows 95 的記憶體管理函式分爲四層，上層函式依賴下層函式。最底層是 VMM 提供的函式，用來配置大塊記憶體並在其中操作 pages。應用程式並不直接呼叫這一層函式，KERNEL32.DLL 才會用到它們，用以成就較高階的函式。

第二層是 KERNEL32 提供的 *VirtualXXX* 函式：*VirtualAlloc*、*VirtualFree*、*VirtualProtect* 等等。這些函式係以 VMM 函式爲基礎，用來管理大塊記憶體，並以 page 爲單位。

更上一層是 KERNEL32 的 *HeapXXX* 函式，包括 *HeapAlloc*、*HeapFree*、*HeapCreate* 等等。它們大約相當於 C 函式庫中的記憶體相關函式（如 *malloc*、*free* 等等）。事實上，在 Windows NT SDK 的 C 函式庫中，*malloc* 只是 *HeapAlloc* 的另一個包裝而已。最上一層是 *LocalXXX* 和 *GlobalXXX* 函式。但和 Win16 不同的是，這兩組函式基本上沒有差別，例如 *LocalAlloc* 就和 *GlobalAlloc* 完全相同。KERNEL32 開放出這兩個函式，但使用同一個函式位址。*LocalXXX* 和 *GlobalXXX* 其實只是 *HeapXXX* 的上層包裝而已，其實沒有太多理由需要在 Win32 程式中使用 *LocalAlloc* 和 *GlobalAlloc* 這樣的函式。這些記憶體管理函式不再像 Win16 的 *GlobalAlloc* 一樣需要和 selector 打交道，也不再像 Win16 的 *LocalAlloc* 一樣從程式的資料節區中挖空間。它們之所以繼續存在於 Win32，主要理由就是讓原來的 Win16 程式更容易生存。本章剩餘部份會深入挖掘這四層函式。除了最底層的 VMM 函式之外，我會提供每一個記憶體管理函式的虛擬碼。某些 Win32 函式可能沒有在 Windows 95 中實作出來，或者可能只是單純對映到其他函式。我都會一一指出。

# VMM 函式

最低階的記憶體管理函式存在於 VMM VxD 之中，提供對線性位址空間的保留（reserve）、委派（commit）、解除委派（decommit）、釋放（free）等動作。以及查詢 pages 狀態、管理 memory contexts、安裝 page fault 處理常式、堆積函式（給其他 VxDs 使用）等等。表 **5-1** 內含 VMM 記憶體相關函式的 DDK 說明。

如果你熟悉 VxDs，可能你會想，這個表格中的函式很棒，但對 ring3 程式又有什麼幫助？畢竟，一般的 ring3 程式不能夠呼叫 VxD 函式。呵，我會給你一個好理由：每一個函式都可以被 ring3 碼呼叫，只是並非直接呼叫罷了。

Windows 95 小組認爲這一組函式對 KERNEL32 非常重要，所以他們爲這每一個函式實作出一個 Win32 VxD services。Win32 VxD services 是 Windows 95 的一種新機制，允許 ring3 應用程式使用 C 語言呼叫習慣，呼叫 ring0 VxD。這裡所說的 services 和 Windows NT services 沒有關係，後者是一種特殊的行程。

第6章會比較詳細地說明 Windows 95 的 Win32 VxD services。在這裡我們只要知道一件事實就好：每一個由 VxD（例如 VMM）提供的 Win32 VxD services，都以一個獨一無二的號碼識別之。較高字組是 VxD service ID，較低字組則是 Win32 VxD services 表格中的索引值。圖 **5-6** 顯示表 **5-1** 所列之 VMM 函式的 Win32 VxD services ID。第6章會描述 Win32 VxD services，並且有一個更詳細的 services ID 列表。

**表 5-1　VMM 記憶體相關函式的說明（DDK）**

| VMM 函式名稱 | 目的 |
|---|---|
| _PageReserve | 在目前的 context 中保留一段線性位址，但不配置任何 RAM。 |
| _PageFree | 釋放指定的記憶體區塊。 |
| _PageCommit | 把實際記憶體（RAM）委派（committed）給某一範圍的線性位址。 |
| _PageDecommit | 把某範圍內的線性位址所映射的實際記憶體（RAM）解除委派（decommitted）。 |
| _PageAttach | 在目前的 memory context 中映射一段線性位址到相同的實際儲存裝置上。所謂「相同」是指和該些被映射到特定之 context（source context）的 pages 相同。 |
| _PageFlush | 將某個範圍內的 committed pages 寫到一個被鎖住的檔案。這個 service 不會把 pages 標記爲 not-present。 |
| _PageModifyPermissions | 修改某個範圍的 pages 的處理權（permission）。 |
| _PageQuery | 取得某個範圍內的 pages 的相關資訊。這些資訊的格式和 VirtualQuery 傳回的相同。 |
| _PageRegister | 通知系統說，有一個新型態的 pager。 |
| _PagerQuery | 取得一個註冊過的 pager 的資訊 |
| _ContextCreate | 產生一個新的 memory context。Windows 95 的 tasking 系統和 scheduling 系統就是利用這個 service 來爲新的 Win32 程式產生一個私有的線性位址空間。 |
| _ContextDestroy | 摧毀一個由 _ContextCreate 產生的 memory context |
| _ContextSwitch | 改變 memory context |
| _GetCurrentContext | 取得目前的 memory context |
| _HeapAlloc | 從系統堆積中配置一塊記憶體 |
| _HeapReAlloc | 在系統堆積中重新配置或重新初始化一塊記憶體 |
| _HeapFree | 從系統堆積中釋放一塊記憶體 |

```
0x00010000 _PageReserve
0x00010001 _PageCommit
0x00010002 _PageDecommit
0x00010003 _PageRegister
0x00010004 _PageQuery
0x00010005 _HeapAlloc
0x00010006 _ContextCreate
0x00010007 _ContextDestroy
0x00010008 _PageAttach
0x00010009 _PageFlush
0x0001000A _PageFree
0x0001000B _ContextSwitch
0x0001000C _HeapReAlloc
0x0001000D _PageModifyPermissions
0x0001000E _PageQuery
0x0001000F _GetCurrentContext
0x00010010 _HeapFree
```

**圖 5-6　VMM 的 Win32 VxD service IDs，用來呼叫 ring0 VMM 函式。**

爲了呼叫這些 VMM 函式，KERNEL32 把參數壓入 stack 之中，後面再緊跟著 Win32 VxD service ID，然後呼叫 *VxDCall* 函式（***Unauthorized Windows 95*** 一書稱此函式爲 *VxDCall0*）。例如，VMM.VXD 的 *_PageReserve* 函式原型如下：

```
ULONG EXTERNAL _PageReserve(ULONG page, ULONG npages, ULONG flags);
```

下面是 KERNEL32 載入器的碼，顯示 *_PageReserve* 如何被 ring3 呼叫：

```
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
```

我並沒有像對待高階 API 函式那樣，提供這些 VMM 函式的虛擬碼給你看。應用程式不能夠直接呼叫它們，請你把它們想像是基礎工程。ring3 記憶體管理函式就是利用它們堆砌起來的。我之所以列出它們是因爲有些讀者沒有 DDK。我並不打算完全忽視它們，後續各節中我還會提到這些 VxD 函式。

# Win32 的 Virtual 函式

> 譯註：此處的 virtual 函式是指以 *Virtual* 開頭的 Win32 API（都與記憶體管理有關）。和 C++ 中的虛擬函式（virtual function）沒有任何關聯。

Win32 記憶體管理 API 函式中最底層的就是 virtual 函式，例如 *VirtualAlloc* 和 *VirtualProtect*。這些 virtual 函式用來配置和管理大塊記憶體。在 Windows 95 之中，virtual 函式對記憶體的基本量是 4KB，這使得它們不適合用來取代 C/C++ 的 *malloc* 和 *new*。它們之中大部份都是 VMM 函式的一層薄包裝。關於這一點，當我展示 virtual 函式的虛擬碼時你就會看出來。

Win16 之中與這些 virtual 函式最接近的要算是 global heap 函式了，例如 *GlobalAlloc*。Win116 的 global heap 函式和 Win32 的 virtual 函式都允許你配置大塊記憶體。但是和 global heap 函式不同的是，virtual 函式並不使用 selector 來指向記憶體。它們以 4KB 區塊爲記憶體單位，不使用 selector。而 Win16 global heap 函式允許你配置甚至像 20h 位元組那麼小的區塊。

## VirtualAlloc

*VirtualAlloc* 有多重功能。任何時候 VMM 記憶體管理器對待線性記憶體的每一個 page 的態度是 free、reserved、或 committed。*VirtualAlloc* 使你得以單向改變某一個範圍內的 pages 的狀態。它可以改變 page 的狀態，從 free 到 reserved，或是從 free 到 committed。此外，它也可以改變原本 reserved 的 pages 成爲 committed 狀態。

最後一種改變，從 reserved 到 committed，對於稀疏記憶體和 stack 的實作極有價值。程式首先利用 *VirtualAlloc* 保留一塊夠大的記憶體空間，足夠滿足程式中的任何需求。然後程式設定一個結構化異常處理常式（Structured Exception Handler）搜尋被保留的記憶體範圍內的 page faults。當那些 page faults 發生，程式再度呼叫 *VirtualAlloc*。這一次 *VirtualAlloc* 把引起 page fault 的 pages，從 reserved 狀態改爲 committed 狀態。這麼一來程式就可以配置巨量記憶體而不需要先獲得實際的 RAM。只有當那些 pages 真正被碰觸了，才需要映射到實際的 RAM。

一般而言，*VirtualAlloc* 被作業系統和應用程式使用，在程式的位址空間（也就是 2GB 以下）配置記憶體。然而，它有一個未公開的旗標（0x8000000），允許它抓取 2GB 以上的記憶體。那是被所有程式共享的，所以這是一個未公開的「在程式之間共享記憶體」的方法。你可以使用記憶體映射檔完成相同的事情。事實上，粗略地說，記憶體映射檔所使用的位址範圍，正相當於 *VirtualAlloc* 以 0x8000000 旗標配置而得的位址。

Win32 *VirtualAlloc* 函式保留記憶體時，是以最接近的 64KB 邊界爲始。然而，並不是 *VirtualAlloc* 做這個「切齊」動作，是 *VirtualAlloc* 所呼叫的 *_PageReserve* 完成的。

*VirtualAlloc* 首先檢查索求的記憶體是否太大。這裡的「太大」意味著超過 2GB-4MB，這是每一個應用程式的線性位址保留區的大小。然後，*VirtualAlloc* 計算出需要多少 pages，然後把起頭的位址下移到最接近的 4KB 邊界，也把最尾端的位址上移到最接近的 4KB 邊界。因此，如果你要求 2 位元組，一個在某 page 的最後，另一個在另一 page 的最前，那麼 *VirtualAlloc* 會嘗試保留兩個 pages。

接下來 *VirtualAlloc* 處理來自 fdwAllocationType 參數（譯註：原文書中寫爲 fdwProtect，應爲筆誤）的各種旗標值。首先，它看看是否有未公開的 0x8000000 旗標，那意味要配置 2GB 以上的記憶體。*VirtualAlloc* 忽略 MEM_TOP_DOWN 旗標。然後它再測試是否你只傳遞 MEM_COMMIT 或 MEM_RESERVED 旗標過去。任何其他的旗標都會引發除錯版的一個警告訊息。最後，函式碼呼叫 *mmPAGEToPC*，那是一個輔助函式（下節描述），把 fdwProtect 參數旗標轉換爲 VMM 的 *_PageReserve* 所使用的旗標。

這時候，函式兵分兩路。如果不在乎哪一段記憶體要保留，就執行其中一路。如果呼叫端指定了某特定範圍的記憶體，那麼就執行另一路。不管是哪一路，如果記憶體要被保留，*VirtualAlloc* 就呼叫 Win32 service 00010000，那是 VMM's *_PageReserve* 函式的一個外包函式。保留住這段記憶體之後，如果呼叫端還有指定 MEM_COMMIT 旗標的話，*VirtualAlloc* 就呼叫 Win32 service 00010001，那是 VMM's *_PageCommit* 函式的一個外包函式。如果呼叫端指定了一個特定的位址範圍，*VirtualAlloc* 會檢查它是不是在 0xC0000000（VxD 領域的起始處）之下。

整個函式碼中，*VirtualAlloc* 不斷周到地檢查傳回值 _PageReserve 和 _PageCommit。如果任何事情失敗，它就會吐出一個除錯診斷訊息，然後從某個唯一出口退出。該出口處釋放先前保留的記憶體。

**VirtualAlloc 的虛擬碼**

```
// 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

這個函式被 *VirtualAlloc*、*VirtualAllocEx*、*VirtualProtect* 等函式使用。它把定義在 WINUSER.H 中的 PAGE_ 旗標（例如 PAGE_READONLY）轉換爲對應的 PC_ 旗標。PC_ （Page Commit）旗標定義於 VMM.INC，被 VMM 的 *_PageCommit* 函式使用。

Windows 95 所使用的旗標之中，有一個用來表示此一 page 是受保護的 page。當程式企圖對受保護之 page 做寫入動作，會引發一個 page fault，於是作業系統必須委派（commit）額外的記憶體於 stack 的底部，允許 stack 向下成長。然而，很明顯你不能夠以 *VirtualAlloc* 要求一個受保護的 page，因爲 *mmPAGEToPC* 把 PAGE_GUARD 旗標給濾掉了。這個函式也忽略 PAGE_NOCACHE 旗標。*mmPAGEToPC* 的主要內容就是簡單地對映各式各樣的 PAGE_ 旗標值。除了 PAGE_NOACCESS 之外，被轉換的旗標

都含入 PC_USER 位元，意思是此一 page 可以被 ring3 碼（user level）存取。如果 page 應該是可寫入的，PC_WRITEABLE 也會被放到傳回的旗標內。換一個說法，除了 PAGE_NOACCESS 之外，所有的 PAGE_ 旗標都會被對映爲 PC_USER 或 PC_USER | PC_WRITEABLE。

**mmPAGEToPC 的虛擬碼**

```
// 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* 執行 *VirtualAlloc* 的相反機能。它可以把 pages 狀態從 committed 改變爲 reserved，或從 committed 改變爲 free，或從 reserved 改變爲 free。函式的第一部份首先檢查傳進來的是否爲合法的位址以及合理的大小。位址必須在 3GB 之下，大小必須小於 2GB-4MB（應用程式私有位址空間的大小）。

你可以指定 MEM_RELEASE 或 MEM_DECOMMIT 旗標，但不能兩者同時指定。MEM_RELEASE 使得 *VirtualFree* 呼叫 VMM's *_PageFree* 函式，將整個範圍內的所有 pages 都 "decommit"（如果需要的話）並 "unreserve"。這種情況下，你必須指定大小爲 0，使得 *VirtualFree* 釋放原先以 *VirtualAlloc* 配置的整塊空間。如果你傳遞 MEM_DECOMMIT 進去，會使得 *VirtualFree* 呼叫 VMM's *_PageDecommit*，將某個區塊的 pages "decommit" 掉。

**VirtualFree 的虛擬碼**

```
// 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 )
```

```
{
    _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

這或許是 Windows 95 之中最俏皮討好的函式了。它針對某個位址，提供記憶體型態方面的豐富資訊。例如，給予行程位址空間中的任意位址，*VirtualQueryEx* 可以告訴你哪一個 EXE 或 DLL 擁有此塊記憶體。*VirtualQueryEx* 也是 Windows NT PWALK 程式的核心，該程式爲任何一個指定的行程顯示一張記憶體佈局圖。

*VirtualQueryEx* 最初並不在 Windows 95 的 Win32 函式名單中，這對於開發系統層面工具軟體（如除錯器）的廠商而言，真是令人震驚。幸運的是，Windows 95 小組從善如流，最終還是納入了 *VirtualQueryEx*。

*VirtualQueryEx* 將某個位址的資訊填入 MEMORY_BASIC_INFORMATION 結構中。這個結構看起來像這樣：

```
PVOID BaseAddress;
PVOID AllocationBase;
DWORD AllocationProtect;
DWORD RegionSize;
DWORD State;
DWORD Protect;
DWORD Type;
```

這些欄位都在 Win32 文件中有所描述。這裡我必須對其中一個欄位特別加以說明。AllocationBase 聽起來或許不怎麼樣，然而它卻是最重要的一個欄位。技術上來說，它代表最初以 *VirtualAlloc* 配置而來的記憶體的基底位址。更重要的是，當 *VirtualQueryEx* 的 lpvAddress 參數落在一個 EXE 或 DLL 模組之中，AllocationBase 就成爲此 EXE 或 DLL 的基底位址。也就是說，AllocationBase 和 EXE 或 DLL 的 HINSTANCE / HMODULE 是一樣的。NT SDK 所附的 PWALK 程式就是利用這一點來走訪一個行程的位址空間，並且爲各個區域貼上其擁有者（EXE 或 DLL）的名稱。除錯器則可以使用這個功能算出哪一個 EXE 或 DLL 關聯到某個有問題的位址。

*VirtualQueryEx* 基本上只是呼叫 VWIN32.VXD 的第 40h 號 Win32 service。（也就是 *VxDCall* 002A0040）。這個 service 內部呼叫 VMM 的 *_PageQuery* 函式。DDK 文件中說 *_PageQuery* 函式需要一個參數，指向 MEMORY_BASIC_INFORMATION 結構。

或許是爲了避免因爲一個不適當的執行緒切換動作而傳回不調和的值（在 MEMORY_BASIC_INFORMATION 結構中），所以 *VirtualQueryEx* 一開始就先取得 Krn32Mutex，並在離開時釋放之。它是以未公開的 *KERNEL32_EnterSysLevel* 和 *_LeaveSysLevel* 函式完成這些工作。

第 43h 號 VWIN32 service，也是填寫 MEMORY_BASIC_INFORMATION 結構，就比單單的 *_PageQuery* 外包函式更多一些。目前我不能夠真確地說它到底做了什麼事情，然而很顯然這個函式必須知道查詢對象（行程）之現行執行緒的 ring0 stack 位址。因此，在呼叫 VWIN32 service 之前，*VirtualQueryEx* 使用 hProcess 參數取得一個指標，指向行程的結構（請看第 3 章的「Windows 95 Process Database（PDB）」一節）。從那個地方，*VirtualQueryEx* 取得現行執行緒的 thread database，交給 VWIN32 service。有趣的是，在仔細觀察第 43h 號 VWIN32 service 數次之後，老實說，我未曾發現這程式碼除了呼叫 *_PageQuery* 之外還做了些什麼。

**VirtualQueryEx 的虛擬碼**

```
// 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* 只不過是 *VirtualQueryEx* 的一個特例。*VirtualQuery* 取得目前行程中某位址的資訊，而 *VirtualQueryEx* 可以取任何行程的位址資訊。

*VirtualQuery* 碼幾乎沒有什麼值得說的，它只是檢驗參數的合法性。它看看一個被指標所指的緩衝區是否大到足夠容納 MEMORY_BASIC_INFORMAT 結構。假設答案是肯定的，*VirtualQuery* 跳到 *IVirtualQuery* 去。*VirtualQuery* 先做參數檢驗，再跳到一個內部函式去，這是 system DLL 的許多函式的典型作爲，例如 *VirtualProtect* 也是如此（稍後會提到）。

不比某些在除錯版中只作運轉記錄用的函式，*IVirtualQuery* 其實只是呼叫 *VirtualQueryEx*，並以目前行程的虛擬 handle 做爲第一參數。請注意，在 Windows 95 之中，*IVirtualQuery* 呼叫 *VirtualQueryEx*。這和 Win32s 不同，後者的 *VirtualQueryEx* 呼叫 *VirtualQuery*。其間的關鍵差異在於 Win32s 中每一個行程共享相同的位址空間，所以 *VirtualQuery* 應該等同於 *VirtualQueryEx*。

**VirtualQuery 的虛擬碼**

```
// 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 的虛擬碼**

```
// 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* 可以改變一個 committed page 或一系列 pages 的存取保護狀態。它可以處理任何行程，只要你有行程的 handle。*VirtualProtectEx* 和 *VirtualAlloc* 之間的關鍵差異在於前者假設你已經 "committed" 你「正打算改變其狀態」的那些 pages，而 *VirtualAlloc* 允許你配置、委派（commit）、然後指定處理某一個 page 或某一些 pages。

*VirtualProtectEx* 的碼十分直接了當。就像我所說過的其他 virtual 函式一樣，它先以一些錯誤檢驗揭開序幕。函式碼檢查要修改的位址範圍是否小於 2GB-4MB，起始位址是否小於 0xC0000000。*VirtualProtectEx* 的中心是呼叫 VWIN32 service 0x3F。這個 service 最終呼叫 VMM's *_PageModifyPermission*。就像在 *VirtualQueryEx* 中一樣，這個 VWIN32 call 爲了某些理由，期望獲得一個指標，指向指定之行程的現行執行緒的 ring0 stack。有一些碼用來決定這個 ring0 stack 是否即是我們在 *VirtualQueryEx* 中獲得的。*VirtualProtectEx* 也和 *VirtualQueryEx* 一樣，在 VWIN32 call 執行期間，取得並持有 Krn32Mutex。

VWIN32 service 0x3F 傳回被改變之 pages 之前一狀態（如果呼叫成功的話）。然而，這個狀態是以 VMM 的 PC_ 旗標值記錄，而不是呼叫端所期望的 PAGE_ 旗標。*VirtualProtectEx* 因此做一次快速的轉換。最後，如果呼叫端有指定一個指標用來儲存舊的 page 屬性，函式碼就把那些 PAGE_ 旗標拷貝過去。

**VirtualProtectEx 的虛擬碼**

```
// 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* 的簡化版本。只對現行行程才有作用。*VirtualProtect* 事實上只做參數檢驗，真正的碼在 *IVirtualProtect* 之中。唯一在 VirtualProtect 中進行的檢驗工作是決定 pfdwOldProtect 指標是一個合法的 DWORD 或是 0。

*IVirtualProtect* 是 *VirtualProtectEx* 的外包函式。它所傳遞的 hProcess 是一個虛擬 handle，用以表達目前的行程（0x7FFFFFFF）。在除錯版中，*IVirtualProtect* 也呼叫一個專門做運轉記錄的函式，把某些 API calls 輸出到除錯終端機上。

**VirtualProtect 的虛擬碼**

```
// 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 的虛擬碼**

```
// 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

這兩個函式並不存在於 Windows 95。在支援它們的 Win32 平台（如 Windows NT）上，它們允許行程 "pagelock" 一個範圍的 pages。系統保證那些 pages 總是能夠映射到實際的 RAM。對於那種承擔不起 page fault 的代價者（例如對於時間非常吹毛求疵的裝置驅動程式），幫助頗大。

在 Windows 95 之中，*VirtualLock* 和 *VirtualUnlock* 都跳到 *CommonUnimpStub* 碼中。那是一小段程式碼，所有未實現的 Win32 APIs 都會跳到那兒。*CommonUnimpStub* 的影響是雙重的，第一，在除錯版本中，KERNEL32 吐出一個診斷訊息到除錯終端機上，像

這樣：

```
*** Unimplemented Win32 API : VirtualLock
```

第二個影響是清除 stack 中適當的參數。在 *VirtualLock*/*Unlock* 一例中，清除的是 8 個位元組。由於 *CommonUnimpStub* 所處理的 APIs 有多寡不同的參數個數，所以需要清除的 stack 大小必須先讓 *CommonUnimpStub* 知道。這可以經由 CL 暫存器的傳遞而獲知。CL 暫存器中放的是個經過編碼的值，不是直接的位元組個數。

**VirtualLock 的虛擬碼**

```
EAX = "VirtualLock"
    CL = 12
    JMP CommonUnimpStub
```

**VirtualUnlock 的虛擬碼**

```
EAX = "VirtualUnlock"
    CL = 12
    JMP CommonUnimpStub
```

# Win32 的 Heap 函式

微軟終於在 Win32 作業系統中放入了一些高級的 heap 管理函式。DOS 記憶體配置體制針對一個 heap 建立起一塊空間，往往太大又太慢。Win16 *GlobalAlloc* 的最小配置限額爲 20h 位元組，並且受限於 8192 個 selectors。Win16 的 *LocalHeap* 比較適用於小量配置，但它最大又只能是 64KB。此外，這些函式都沒有破洞追蹤（leak tracking）或記憶體覆疊（memory overrun）的能力。

Win32 heap 函式優秀多了。在 Windows 95 之中，每一個區塊只需額外消耗 4 位元組，而理論上你可以產生一個 heap 高達 2GB-4MB 那麼大。此外，Windows 95 的 Win32 heaps 爲各種大小的區塊維護了四個分離的自由串列，爲的是避免記憶體過度支離破碎。另一個優點只在除錯版才發作，那是，每一個被配置的區塊都貼上額外的資訊，使你能夠輕易找出記憶體破洞、覆疊情況，並知道是誰配置了這塊記憶體。稍後我還會介紹如

何使用這些額外的除錯資訊。不幸的是，唯一能夠讓記憶體覆疊檢驗發生效力的是，使用一個晦暗的，只 Windows 95 才有的函式：*HeapSetFlags*。在我下筆的此刻，微軟的任何文件中都沒有提到過這個函式，但是我獲得消息說它將會被提供出來。我將在 ***Microsoft Systems Journal*** 1995 年十月份我的專欄中描述 *HeapSetFlags*（我發現這個函式的時間太晚，以至於沒有能夠在這裡描述它）。

除了這些極佳的函式外，Windows 95 也允許行程擁有一個以上的 heaps。這使你得以方便地將你的記憶體配置以 heap 區分不同的型態。這常常是避免記憶體破碎的一個好策略。由於 Windows 95 支援一個以上的 heaps，所以你總是必須傳一個 heap handle 給任何 Win32 heap 函式，用來表示你打算操作的對象。Heap handle 其實就是 heap 的起始位址（線性位址）。

Windows 95 heaps 的另一個好性質是，它們可以成長。這種情況下，KERNEL32 配置額外的記憶體，並將之與 heap 產生關聯。我稱此額外記憶體爲 subheaps。**圖 5-7** 顯示一個複雜的 heap 設定。

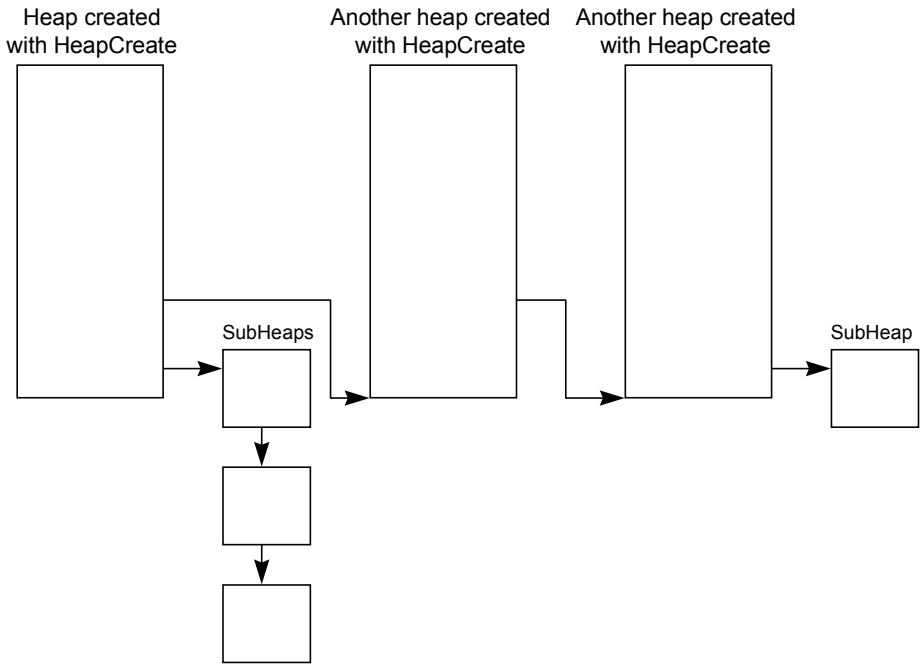



**圖 5-7　一個行程，擁有許多個 Win32 heaps。**

Win32 heap 函式包括 *HeapAlloc*、*HeapFree*、*HeapReAlloc* 等等。或許你會以爲，對於需要實作出 *malloc*、*realloc*、*free*、*new*、*delete* 函式的編譯器廠商而言，這些基本的 heap 函式將是必然的選擇。不，真實情況並非如此。Borland 和 Microsoft 都迴避將 Win32 heap 函式用於其 runtime library 中。Win32 SDK runtime library（CRTDLL.DLL）是一個例外，它的 *malloc* 和 *free* 函式分別使用 *HeapAlloc* 和 *HeapFree*。注意，NT 和 95 所使用的 CRTDLL.DLL 是不同的版本。

更正啓事：本書即將付梓之前，我發現 Visual C++ 4.0 使用 Win32 heap 函式來完成其 C/C++ runtime heap。

在 Windows 95 的 Win32 heaps 服務之更上層，你會發現 *GlobalAlloc* 和 *LocalAlloc*。它們都是以 *HeapAlloc* 家族函式完成的。*LocalAlloc* 並不只是 *HeapAlloc* 的另一個包裝，因爲有些 Win16 程式員針對 *LocalAlloc* 配置而來的記憶體玩了一些難纏的把戲（譯註：所謂的 sub-allocation），Win32 版的 *LocalAlloc* 必須顧慮到回溯相容。我將在後續數節中詳細討論這個主題。在 Win32 heap 函式的下層，使用的是 VMM 提供的有關於記憶體管理的 Win32 VxD services。然而，我並沒有看到這些函式中有任何東西沒辦法在 virtual 函式（稍早我描述過的）實作出來。因爲這一點，我相信 Win32 heap 函式其實是 Win32 virtual 函式的上層。有趣的是，VMM 的 *_HeapXXX* 函式（提供 heap 機能給 VxD）所使用的 heap 結構，與 KERNEL32 用於 ring3 行程的 heap 結構相同。

## Win32 的 heap header 和 head arenas

一個 Windows 95 heap 的所有零組件都是從 VMM *_PageReserve* 所配置的記憶體中產生出來。這塊記憶體被區分爲兩塊。起頭是一個 heap 表頭。這個表頭（稍後我們會詳細地看個清楚）內含一些用以管理 heap 的資訊，像是自由串列啦、heap 大小啦、heap 生成旗標啦等等。表頭之下便是 heap 區塊。每一個 heap 區塊一開始是一個所謂的 arena 結構，內含此一區塊的資訊。區塊的頭緊跟著前一區塊的尾。所有的區塊一直延伸到 heap 空間的尾端，但並不是其中的每一個 page 都一定得映射到實際的 RAM。**圖 5-8** 顯示典型的 heap 佈局。

記住，每一個 heap 區塊，不管是自由或是使用中，都以一個 arena 結構開始。其格式在 Windows 95 的除錯版和零售版中並不相同。如果 heap 區塊是自由的，arena 中還會出現一些額外的欄位。這於是導至 arena 佈局的四種變化：retail free、retail in-use、debug free、debug in-use。

每一個 heap arena 一開始都有一個 DWORD，內含 heap 的大小。這個大小包括被 arena 自己使用的空間。然而你可以簡單地取出第一個 DWORD 並以它當做區塊的大小。爲什麼？因爲在這第一個 DWORD 中有一些位元被用於和區塊大小無關的項目。這個 DWORD 的較高字組總是 0xA0，其意義不甚清楚，我猜想它是一個 "bit pattern"，用來

告訴 KERNEL32 說這個 arena 是否被覆寫。其他與區塊大小無關的位元還有，因爲所有的 heap 區塊的大小總是 4 的倍數，所以最低兩個位元用不到，可以當做某種旗標。這兩個位元（位元 1 和位元 2）的實際意義是：

- 位元 1：若設立，即表示此區塊爲自由。0 表示此區塊已被配置。
- 位元 2：若設立，即表示此區塊之前一區塊爲自由。這個位元只有在已被配置的區塊中才會設立。若此一區塊是自由的，它就可以和前一自由區塊聯結在一起。如果它未被設立，表示前一區塊不自由，所以不需要企圖聯結本區塊。

把所有位元納入考慮，很容易就可以算出區塊大小。只要把第一個 DWORD 和數值 0x5FFFFFFC 做位元之間的 AND 運算即可，這個運算把 DWORD 中所有非用於指示區塊大小的位元都遮罩掉。早先的一種作法是把第一個 arena DWORD 和數值 ~0xA0000003 做邏輯的 AND 運算。爲計算出有多少記憶體還可被呼叫端使用，只要把區塊大小減去 arena 大小即可。

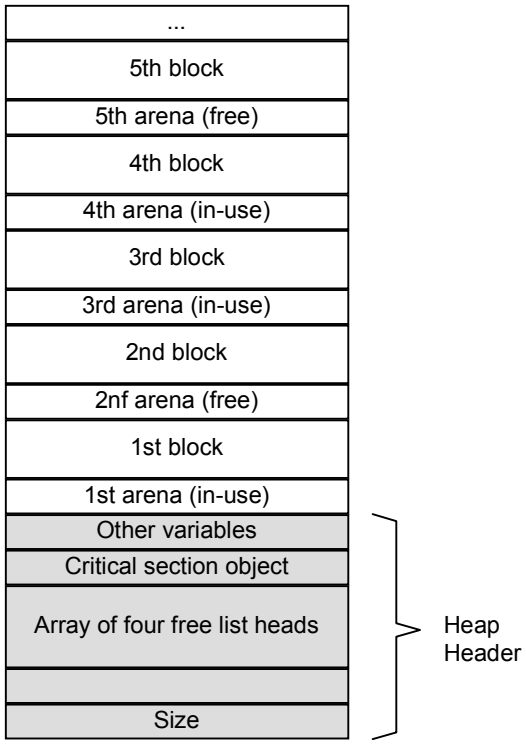



**圖 5-8　一個典型的 Win32 heap**

### Windows 95 零售版（Retail Version）的 in-use 區塊

這種區塊的 arena 型態最是簡單：

```
DWORD  size   // OR'ed with 0xA0000000 or 0xA0000002
```

### Windows 95 零售版（Retail Version）的 free 區塊

這種區塊的 arena 型態與前一種相同，但是增加了 prev 和 next 欄位：

```
DWORD  size   // OR'ed with 0xA0000001
DWORD  prev   // Pointer to the previous heap arena
DWORD  next   // Pointer to the next heap arena
```

## Windows 95除錯版（Debug Version）的 in-use 區塊

和零售版類似，但增加了一些欄位：

```
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.
```

這些額外欄位用以追蹤記憶體覆疊以及 heap 被破壞等情況。"allocating EIP" 欄位儲存了一個程式位址，區塊就是在該處被配置的。這可以用來做定點測試（where a block of code that somehow wasn't free was allocated）。Thread number 欄位的作用類似，不過它是用來識別哪一個執行緒配置了這塊空間。請注意，thread number 和 thread ID 並不相同，後者是 GetCurrentThreadId 傳回的值。thread number 是一個索引，指向目前的執行緒串列。你可以利用 SoftIce/W 的 "THREAD" 命令看到這個索引值。對於 in-use 區塊而言，signature 欄位應該總是 0x4842。如果不是，這個 arena 可能是被破壞了。

Arena 的最後一個欄位提供更有力的 heap 破壞防護。這個欄位內含前三個 DWORDs 的檢驗總和。其演算法在稍後說明 ChecksumHeapBlock 時有所描述。雖然這個欄位永遠有在維護，但是 KERNEL32 除錯版從來不會自動驗證它，你必須推一下，它才動一下。這一性質，稱爲 "paranoid heap corruption checking"，靠 *HeapSetFlags* 函式而決定作用與否。在我寫的一個簡單測試中，當我違反 checksum 規則，收到以下輸出：

```
hpWalk: bad busy block checksum trashed addr between 560014 and 560020
heap handle=460000
```

## Windows 95除錯版（Debug Version）的 free 區塊

這些區塊的 arena 是零售版 free arena 和除錯版 in-use arena 的混合。它也有 prev 和 next 欄位，也有 thread number、signature、checksum 等欄位。signature 從 0x4842 改爲 0x4846。checksum 的演算法也有輕微改變。由於比除錯版 in-use arena 多出一個 DWORD，所以當 KERNEL32 檢驗這個 arena 時，它使用前四個（而非前三個）DWORDs。

```
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 的 heap header（表頭結構）

每一個 heap 的起頭是一個 heap 表頭結構。所謂 heap handle，例如你以 *GetProcessHeap* 所獲得者，只不過是個指標，指向 heap 的表頭結構而已。*HeapCreate* 函式的主要工作，除了保留記憶體給 heap 使用之外，就是將表頭結構初始化。Windows 95 的零售版和除錯版的 heap 表頭結構大小是會變化的（但其格式變化不大）。緊跟在表頭之後的就是第一個 heap 區塊的 arena。

### Windows 95 零售版（Retail Version）的 heap 表頭

#### 00h　WORD　dwSize

保留給此 heap 的記憶體總量。每一個行程有一個 default heap，大小是 1MB+4KB。

#### 04h　DWORD　nextBlock

如果呼叫 *HeapCreate* 時將 dwMaximum 參數指定爲 0，那麼此 heap 產生之後，大小還可以擴充。這種情況下，如果呼叫端要求的區塊對此 heap 而言太大，KERNEL32 就會保留其他的記憶體，並設定 subheaps。Subheaps 也使用 heap arena，但不使用整個表頭結構。爲了追蹤這些 subheaps，KERNEL32 以串列存放它們。串列頭就記錄在主表頭結構的此一欄位之中。指向下一個 subheap 的指標則放在每一個 subheap 的 04 偏移位置處。一旦 heap 被摧毀，KERNEL32 會走訪串列中的每個 subheaps，將它們的 pages 釋放給系統。

### 08h FREE_LIST_HEADER_RETAIL freeListArray[4]

爲了盡量減低記憶體破碎的情況並加速對自由區塊的搜尋，每一個 heap 表頭都維護有四個自由串列。分別掌管 0x20 位元組以下、0x80 位元組以下、0x200 位元組以下、0xFFFFFFFF 位元組以下的自由區塊。KERNEL32 會針對最適當的串列展開搜尋動作。如果你需要一塊 0x18 大小的空間，KERNEL32 搜尋第一個串列。如果你需要一塊 0x100 大小的空間，KERNEL32 搜尋第三個串列。

這四個自由串列是以四個簡單結構所組成的陣列來表示。結構格式如下：

- **DWORD maxBlockSize** 此一串列之最大區塊。可以是 0x20、0x80、0x200、0xFFFFFFFF。
- **free arena** 這個 arena 和零售版的 free arena 類似，但是其 size 欄位是 0。prev 欄位指向第一個 free arena。由於 size 是 0，所以搜尋過程中從來不會把此 arena 列入考慮。

### 48h PVOID nextHeap

在 Windows 95 零售版的 heap 中，這個位置是一個指標，指向下一個以 *HeapCreate* 產生出來的 heap。請注意，下一個 heap 並不是下一個 subheap（記錄在 04 偏移位置），而是一個完完整整如假包換的 heap。除非你呼叫 *CreateHeap*，否則此欄位爲 0。

### 4Ch HCRITICAL_SECTION hCriticalSection

這個欄位內含 critical section handle，那是 heap 函式做爲同步化控制用的。請注意這欄位放的並不是 CRITICAL_SECTION 本身，而是一個指標，指向 KERNEL32 之中與 critical sections 有關的一個內部結構。

### 50h CRITICAL_SECTION criticalSection

此欄位內含 CRITICAL_SECTION 本身（定義於 WINBASE.H 中）。當程式碼需要同步化控制，KERNEL32 就把一個指向此欄位的指標交給 *EnterCriticalSection*。這一結構

的各個成員的初始化動作是在行程呼叫 *InitializeCriticalSection* 時完成。如果你不需要同步化控制（例如你只有一個執行緒），你可以傳 HEAP_NO_SERIALIZE 給 *HeapAlloc*、*HeapCreate* 等函式。

#### 68h DWORD unknown1[2]

此欄位之意義未明。

#### 70h BYTE flags

此欄位內含旗標值，可以交給 HeapCreate：

```
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 技術文件中只解釋了其中的兩個項目：HEAP_NO_SERIALIZE 和 HEAP_GENERATE_EXCEPTIONS。

#### 71h BYTE unknown2

此欄位之意義未明。可能是保留給擴充的 HEAP_ 旗標使用。

#### 72h WORD signature

在合法的 Windows 95 heap 中，此欄位應該是 0x4948（"HI"）。

### Windows 95除錯版（Debug Version）的 heap 表頭

除錯版的 heap 表頭十分類似零售版的 heap 表頭。然而，內嵌在其中的 free arena 結構比較大些，也多了一些額外的欄位。下面是其佈局：

#### 00h WORD dwSize

請看前一節說明。

#### 04h DWORD nextBlock

請看前一節說明。

#### 08h FREE_LIST_HEADER_RETAIL freeListArray[4]

請看前一節說明。其 free arena 是除錯版而不是零售版。結構佈局如下：

- **DWORD maxBlockSize** 此一串列之最大區塊。可以是 0x20、0x80、0x200、0xFFFFFFFF。
- **free arena** 這個 arena 和除錯版的 free arena 類似，但是其 size 欄位是 0。

#### 68h PVOID nextHeap

請看前一節說明。

#### 6Ch HCRITICAL_SECTION hCriticalSection

請看前一節說明。

#### 70h CRITICAL_SECTION criticalSection

請看前一節說明。

#### 88h DWORD unknown1[14]

請看前一節說明。

#### C0h DWORD creating EIP

這個欄位內含「呼叫內部函式 *HPInit* 以初始化此一 heap」的程式碼的 EIP 值。很顯然它總是記錄著 *HeapCreate* 呼叫 *HPInit* 的那一點。

#### C4h　DWORD　　checksum

此欄位內放的是 heap 表頭的第一個 DWORD（size 欄位）和 0x17761965 數值的 XOR 結果。我推測這大概是幫助 KERNEL32 偵測 heap 表頭是否覆疊。

#### C8h　WORD　　creating thread number

記錄著產生此一 heap 的執行緒的 thread number（不是 thread ID）。請看前面小節中對於 thread number 的解釋。

#### CAh　WORD　　unknown2

這個 WORD 似乎沒有用到。

#### CCh　BYTE　　flags (HEAP_xxx flags)

請看前一節說明。

#### CDh　BYTE　　unknown3

請看前一節說明。

#### CEh　WORD　　signature (0x4948)

請看前一節說明。

## WALKHEAP 程式

爲了實際顯示 Windows 95 heap 的表頭和 arena 結構，我寫了一個 WALKHEAP 程式。原始碼放在書附碟片中。這個程式包括兩個檔案：WALKHEAP.C 和 WALKHEAP.H，後者內含 heap 表頭和 heap arena 的結構定義。WALKHEAP 必須在 Windows 95 除錯版中執行。磁片中的另一個類似的程式 WALKHP2.EXE 則可以在 Windows 95 零售版中執行。是的，我當然可以利用 32 位元 TOOLHELP32 函式，但是那一點都不好玩。

而且它也不會帶來什麼有益的情報。TOOLHELP32 函式有隱藏某些有趣細節的傾向。

當你執行 WALKHEAP 而沒有指定任何命令列參數，此程式走訪並顯示它的所有 heaps。爲了讓事情有趣些，WALKHEAP 首先對著 default heap 做一些配置和刪除動作，然後產生第二個 heap。使用 heap 表頭中的 Next Heap 欄位，WALKHEAP 就可以走訪該行程的所有 Win32 heaps。

如果你知道你有興趣的特定 heap 的位址，你可以把該位址（也就是 heap handle）當做 WALKHEAP 的命令列參數。它應該以 16 進位呈現，不需要標示 0x 或 h。例如：

```
WALKHEAP 81CEC000
```

**圖 5-9** 顯示的是沒有任何指定位址時，WALKHEAP 的輸出。Block 欄位中的是區塊的線性位址。請注意 heap 中的最前面四個區塊大小都是 0。同時，請注意，你也可以走訪自由串列，只要循著 prev 指標，從第一個區塊開始。

```
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
```

**圖 5-9　WALKHEAP 的輸出結果**

現在我們已經看過了 Win32 heap 表頭以及區塊的佈局，是深入虛擬碼的時候了。從以下討論，我們可以了解 KERNEL32 如何產生、管理、摧毀 heaps。這些虛擬碼都是 Windows 95 除錯版的虛擬碼。

## GetProcessHeap

使用一個 Win32 heap 函式，首先你得有一個 heap handle。大部份程式都使用 KERNEL32 在程式產生時所給予的一個 default heap。你可以呼叫 *GetProcessHeap* 獲得其 heap handle。這個函式很簡單，它取出 KERNEL32 的一個全域變數，指向目前行程的 process database（第 6 章有詳細介紹）。其中放有行程的 default heap 的 handle。

**GetProcessHeap 的虛擬碼**

```
return  ppCurrentProcessId->lpProcessHeap;
```

## HeapAlloc 和 IHeapAlloc

*HeapAlloc*，如其名稱所示，你可以利用它從某個 heap 中配置一塊記憶體。它其實只是做參數檢驗的工作，真實配置記憶體是由 *IHeapAlloc* 和 *HPAlloc* 完成。*HeapAlloc* 檢查 hHeap 所代表之 heap 的大小是否足夠容納一個 heap 表頭。雖然它也可以檢查其他

欄位如 signature 和 checksum，但很奇怪的是它忽略那些欄位。如果 hHeap 通過測試，*HeapAlloc* 就呼叫 *IHeapAlloc*。

**HeapAlloc 的虛擬碼**

```
// 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* 其實只是 *HPAlloc* 的外包函式。後者才是真正的 *HeapAlloc*「本尊」。在呼叫 *HPAlloc* 之前，*IHeapAlloc* 對 dwFlags 做了一些處理。唯一可能倖存的是 HEAP_ZERO_MEMORY 和 HEAP_GENERATE_EXCEPTIONS 兩旗標。前者的值（如果倖存的話）比其原值左移三個位元。

**IHeapAlloc 的虛擬碼**

```
// 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

這才是 *HeapAlloc* 的「本尊」。它首先檢查，要求的區塊大小是否太大。太大意味 0x0FFFFF98（接近 256 MB）。接下來 *HPAlloc* 呼叫 *hpTakeSem*，這使得 heap 表頭中的 critical section 被取得。從此，行程中就沒有其他執行緒可以呼叫 *HPAlloc*，直到 *HPAlloc* 結束爲止。在除錯版中，*hpTakeSem* 也會隨性檢驗 heap 是否被摧毀。*hpTakeSem* 還可以走訪整個 heap，檢查 checksum 以及 signature。你可以利用 *HeapSetFlags* 切換這些能力。*HeapSetFlags* 加入 Windows 95 陣營的時間有點太晚，以至於我沒有辦法把它放到這本書中來講。

*HPAlloc* 接下來取出區塊大小參數，調整使它接近 4 的倍數（也要考慮 arena 的大小喔）。最小值是 0x18。在減去 arena 的大小之後，留給使用者的只剩 8 個位元組。知道區塊大小之後，*HPAlloc* 就能夠決定搜尋四個自由串列中的哪一個。找到正確的串列之後，*HPAlloc* 走訪整個串列（使用自由區塊的 prev 指標）以找出第一個夠大的區塊。

這時候，讓我們假設 *HPAlloc* 找到了一個夠大的區塊。於是它呼叫 *hpCarve*（稍後我將展示其虛擬碼）。*hpCarve* 函式檢驗一個區塊，看看它是剛好夠大，或是可以分裂爲兩塊。如果需要分裂，*hpCarve* 處理所有的工作，包括產生新的 arena、設定 prev 和 next 欄位...等等等。分裂出來的其中一塊剛好夠大，滿足 *HPAlloc* 之所需。另一塊將被放到自由串列之中。

在 hpCarve 回返之後，*HPAlloc* 將新的區塊的 arena 欄位初始化。除了「取得 *HPAlloc* 呼叫端的 EIP」，以及「計算前三個欄位的 checksum」之外，其餘只是一些簡單的指定動作。最後，*HPAlloc* 釋放 heap 的 critical section，傳回一個指標，指向 arena 之後的第一個位元組。

現在讓我們回頭看看萬一 *HPAlloc* 沒有在自由串列中發現一個自由區塊時該怎麼辦。如果 heap 允許成長（也就是說當 heap 產生時 dwMaximumSize 被指定爲 0），*HPAlloc* 需要產生一個 subheap。先前我說過，一個 subheap 是另一個獨立空間，內含一些 heap 區塊。KERNEL32 負責追蹤所有的 subheaps，作法是把它們統統維護在一個串列之中。如果需要產生 subheap，KERNEL32 會決定其初始大小（通常是 4KB），並呼叫 VMM 保留一些 pages。接下來 *HPAlloc* 呼叫 *HPInit* 將新的 subheap 的表頭初始化。稍後當我描述 *HeapCreate* 時你會比較清楚地看到 *HPInit* 的動作。初始化之後，*HPAlloc* 把它安插到 subheaps 所組成的串列中。最後，*HPAlloc* 跳回到「搜尋自由串列」的起始處。我想，這一次應該可以找到足夠大小的區塊了。

### HPAlloc 的虛擬碼

```
// 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

*hpCarve* 從 *HPAlloc* 手上獲得一個自由區塊並把它分裂爲兩塊。第一塊的大小必須如 *HPAlloc* 所要求者。*hpCarve* 先做一些測試工作，確保區塊不會比 *HPAlloc* 所要求者小。並確定這個區塊並非處於使用狀態。

*hpCarve* 的主要部份十分直接了當，大約就是設定新的 arena 內容以及 prev 和 next 欄位。虛擬碼顯示出這一切。

比較有趣的是 "memory committing" 動作。一如我們先前在 HPAlloc 中看過的，Win32 heap 其實處於稀疏狀態。也就是說，heap 中的記憶體只是 "reserved"，並未 "committed"。

如果我們有 1MB heap 並且在還不需要這 1MB 時就把 1MB RAM 委任給它，那麼這是一種浪費。當程式觸及被 reserved 但尚未被 committed 的 page 時，會引發 page fault。因此，heap 函式必須確定對所有使用中的區塊所涵蓋的 pages 都做 commit 動作。*hpCarve* 就是這個動作的發生地點。

當一個區塊被分裂爲兩塊，*hpCarve* 必須將第一塊的所有 pages 都做 commit 動作。此外，*hpCarve* 產生並填寫第二個區塊的 arena，所以那個 page 也必須被 committed。committing 動作由 *hpCommit* 函式完成。該函式決定 pages 的狀態，必要的時候就呼叫 VMM 的 *_PageCommit* 函式。如果你以爲 Windows 95 使用「結構化異常處理（Structured Exception Handling）」來做 commit 動作，那你就錯了！（至少在 Windows 95 除錯版中並非如此）

第二塊的 arena 被設定好之後，*hpCarve* 以常數值將第一塊內容初始化。這就是爲什麼 *hpCarve* 不設定其 arena 欄位的原因。如果 *HeapAlloc* 獲得 HEAP_ZERO_MEMORY 旗標，它會把整個區塊設爲 0。否則它把整個區塊設爲 0xCC，那是中斷點（breakpoint）的 opcode。*hpCarve* 的呼叫者有責任在被切開的區塊的一開始處產生 arena 結構。

**hpCarve 的虛擬碼**

```
// 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

這是我所討論的有關於 *HeapAlloc* 相關函式中的最後一個。這個函式只在 Windows 95 除錯版中才會用到。它需要兩個參數，一個是指向起始 arena 的指標，另一個是所使用的 DWORDs 個數。如果要處理一個使用中的區塊，第二個參數是 3，如果要處理一個自由區塊，第二個參數是 4。從 0 開始，*ChecksumHeapBlock* 每次對下一個 DWORD 做 XOR 運算，儲存到一個 checksum DWORD 中。最後再與 0x17751965 做 XOR 動作，並傳回其結果。

**ChecksumHeapBlock 的虛擬碼**

```
// 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

*HeapSize* 獲得一個指標，指向先前配置的一塊記憶體，並傳回其大小（不含 arena）。

*HeapSize* 其實只負責參數檢驗，然後就呼叫 *IHeapSize*。*IHeapSize* 一開始先將 lpMem 減掉 0x10，以便指向區塊的 arena。然後取得 heap 的 critical section 以阻止其他的執行緒干擾。*IHeapSize* 的重心是取出 arena 的 size 欄位，把它和 0xA0000003 做 AND 動作，然後再減 0x10。減 0x10 是爲了扣除 arena 的大小。最後，*IHeapSize* 放棄 heap 的 critical section 並傳回區塊大小（不含 arena）。

**HeapSize 的虛擬碼**

```
// 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;
```

**HeapSize 的虛擬碼**

```
// 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

*HeapFree* 也是只負責參數檢驗。它檢查 hHeap 所代表的記憶體是否大到足以容納一個 heap 表頭。它也檢查 lpMem 是否爲一個合法的 heap 區塊指標。lpMem 的前端應該有 0x10 位元組的 arena 結構，而 lpMem 本身所指應該至少有 8 個位元組（不含 arena）。因此 *HeapFree* 檢查看看這塊記憶體是否在 lpMem 之前的 0x10 個位元組和之後的 0x7 個位元組都還可以存取。測試之後，它跳到一個奇怪的 *x_HeapFree* 函式去（這是我的命名。稍後描述）。

**HeapFree 的虛擬碼**

```
// 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* 坐落在 *HeapFree* 的參數檢驗和 *IHeapFree* 的真正釋放動作之間。爲了某些理由，並不是每一個 heap 區塊都能夠直接呼叫 *IHeapFree*。*x_HeapFree* 的工作就是要決定誰能夠呼叫 *IHeapFree*。如果不是被某個特殊常式呼叫，*x_HeapFree* 就可以跳到 *IHeapFree* 去（大部份是這種情況）。如果 *x_HeapFree* 是被某特殊常式呼叫（目前我還不知道那是什麼常式），它就先呼叫一個函式，然後才跳到 *IHeapFree* 去。

**x_HeapFree 的虛擬碼**

```
// 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* 有兩個功能。首先，如果被釋放的區塊的前一區塊也是自由的，*IHeapFree* 會聯合這些區塊。*IHeapFree* 如何知道前一區塊是否自由呢？區塊的 arena 的 size 欄位的位元 1 如果設立，表示前一區塊自由。那麼，*IHeapFree* 又如何找到前一區塊的 arena 呢？原來啊，前一區塊的最後一個 DWORD 是一個指標，指向前一區塊的 arena。因此，*IHeapFree* 只要把欲被釋放的 arena 減掉 4，該處就是一個指標，指向 heap 之中的前一區塊。*IHeapFree* 聯合兩區塊的作法是，對前一區塊呼叫 *hpFreeSub*，並告訴它聯合後的區塊大小是兩區塊的和。

*IHeapFree* 的第二個功能就是呼叫 *hpFreeSub*。*hpFreeSub* 是 *HPAlloc* 的相反函式，把記憶體釋放還給 heap（稍後描述）。當釋放動作開始，*IHeapFree* 抓住 heap 的 semaphore，並在呼叫 *hpFreeSub* 之後才放掉。

**IHeapFree 的虛擬碼**

```
// 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* 將一塊記憶體釋放還給 heap。它被賦予將被釋放的 arena 的位址，以及長度。這個函式可以從數個地方呼叫之，包括 *IHeapFree* 和 *hpCarve*。後者使用 *hpFreeSub* 頗爲有趣，因爲被釋放的區塊是一個已經自由的區塊的一部份。

*hpFreeSub* 虛擬碼相當長，所以我讓虛擬碼展示一切細節。高階來看，*hpFreeSub* 由兩部份組成。第一部份負責 decommitting（如果需要的話），當程式把一大塊記憶體還給作業系統，卻還讓它們保持 committed 狀態，並不合理。因此，*hpFreeSub* 看看是否有任何 pages 可以被 decommitted，而不需要和 heap 中的其他區塊有所牽扯。如果有符合條件者，*hpFreeSub* 就呼叫 VMM 的 *_PageDecommit*，自由該區塊。但如果 *hpFreeSub* 是被 *hpCarve* 呼叫，這個規則就會出現例外。這種情況下 *hpFreeSub* 不能夠 decommit 任何 pages，它只能夠看看受影響的 pages 是否處於 reserved 狀態。

*hpFreeSub* 的第二部份負責 arena 的更新。首先，在自由區塊之後的區塊必須是使用中的區塊，否則它應該已經成爲將被釋放的區塊的一部份。*hpFreeSub* 因而將下一個 arena

的 size 欄位的位元 1 設立起來。接下來，*hpFreeSub* 決定被釋放的區塊應該加到四個自由串列的哪一個之中。確定之後，*hpFreeSub* 掃描整個串列，尋找正確的點，插入新的自由區塊。最後，*hpFreeSub* 爲此新的自由區塊完成一個新的 arena，包括填寫 prev 和 next 欄位，以及計算 checksum。

**hpFreeSub 的虛擬碼**

```
// 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

*HeapReAlloc* 對原已存在的一個 Win32 heap 重新配置大小。*HeapReAlloc* 只負責參數檢驗，其動作和 *HeapFree* 所做的一樣。hHeap 參數必須指向一塊長度爲 0xD0 的記憶體。lpMem 參數必須滿足其前 0x10 個位元組和後 0x7 個位元組都是合法空間。如果測試通過，*HeapReAlloc* 就呼叫 *IHeapReAlloc*。

*IHeapReAlloc* 有一點詭異。在它呼叫 *HPReAlloc* 之前，它重新安排了要傳給 *HPReAlloc* 的 dwFlags 參數。爲什麼要這麼做，我很納悶。唯一能夠通過這台搗碎機的旗標值是：

HEAP_GENERATGE_EXCEPTIONS
HEAP_NO_SERIALIZE
HEAP_ZERO_MEMORY
HEAP_REALLOC_IN_PLACE_ONLY

**HeapReAlloc 的虛擬碼**

```
// 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;
```

**lHeapReAlloc 的虛擬碼**

```
// 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

*HPReAlloc* 其實是 *HeapReAlloc* 的核心。其中的碼很長，但不難理解。虛擬碼可以表達出所有的細節。從高階來看，*HeapReAlloc* 有四個情況需要考慮：

- 新區塊比原區塊小。
- 新區塊和原區塊大小相差不多。
- 新區塊比原區塊大。而 heap 中的下一個區塊是自由的，並且可以和原區塊聯結起來，組成夠大的區塊。
- 新區塊比原區塊大。而 heap 中的下一個區塊不是自由的，或者雖然它是自由區塊，但整合原區塊之後還是不夠大。

對於第一種情況，*HPReAlloc* 使用 *hpFreeSub* 將原區塊分裂爲兩半。第一部份爲新區塊，第二部份標記爲自由區塊。

對於第二種情況，*HPReAlloc* 不對原區塊做什麼改變。「不動」門檻大約是 8 個位元組。如果你把 arena 算進去，那就是 0x18 個位元組。

對於第三種情況，*HPReAlloc* 計算它需要下一個區塊的多少空間，然後利用 *hpCarve* 把下一區塊分裂爲兩部份，第一部份必須大到足夠和原區塊整合起來滿足需求。第二部份則標示爲自由區塊。

對於最後一種情況，*HPReAlloc* 嘗試以 *HPAlloc* 配置一個新區塊，滿足所需。如果配置成功，*HPReAlloc* 將原內容複製到新區塊內。然後呼叫內部版本的 *HeapFree* 釋放原區塊。

**HPReAlloc 的虛擬碼**

```
// 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

*HeapCreate* 是所有 Win32 heap 函式的根源。每一個 Win32 程式在開始之前都有一個爲它而生的 heap。此外，程式也可以呼叫 *HeapCreate* 產生另外的 heaps。除了被應用程式使用之外，KERNEL32 也呼叫 *HeapCreate* 在共享記憶體之中產生 heaps，並用這些記憶體來對系統資料結構（諸如執行緒或行程相關資料）排序。雖然未曾公開，應用程式也可以使用此函式產生出共享的 heap -- 只要在呼叫 *HeapCreate* 時指定旗標值 fOptions 爲 0x4000000 即可。

產生一個 Win32 heap 的過程分爲兩部份。*HeapCreate* 處理高階部份（保留記憶體並將即聯結到行程的 heap 串列），另一部份是初始化 heap 表頭的每個欄位。關於第二部份，*HeapCreate* 會呼叫 *HPInit* 完成之（稍後我會描述）。

*HeapCreate* 一開始檢查並修改傳進來的 size 參數（如果必要的話）。首先，它看看是否 dwInitialSize 參數比 dwMaximumSize 參數小。接下來它把 dwMaximumSize 加到最接近的 4KB 邊界。如果 dwMaximumSize 是 0，就需要特別處理 -- 它表示 heap 可以隨時成長。如果 *HeapAlloc* 沒辦法在目前的 heap 中找到足夠空間，它可以保留另一大塊記憶體並設立 subheap。*HeapCreate* 函式碼檢查是否 dwMaximumSize 是 0，如果是，就將 fOptions 參數的 0x40 位元設立起來（那或許是 HEAP_FREE_CHECKING_ENABLED）。最後一項測試是看看 fOptions 參數的 0x04000000 位元是否設立，若是，代表 heap 將建立在一塊共享記憶體中（2GB 之上）。

在 *HeapCreate* 決定它要產生的是什麼之後，它呼叫 VMM 的 *_PageReserve*，保留足夠的線性位址空間。然後呼叫 *HPInit* 將 heap 表頭初始化。*HPInit* 回返之後，*HeapCreate* 又檢查看看是否它要產生的是 KERNEL32 的 shared heap，並採取一些必要動作。最後，它把新產生出來的 heap 加到此行程的 heap 串列之中。如果這是第一個被行程產生的 heap，它會成爲串列的頭，並且被記錄到 process database（第 3 章）之中。

**HeapCreate 的虛擬碼**

```
// 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

*HPInit* 負責初始化一個新的 heap 的欄位。它的對象可以是 main heap，也可以是 subheap。如果是後者，其表頭明顯較小。

在某些邊界條件通過之後，*HPInit* 呼叫 *hpCommit* 將第一個 page 委以實際記憶體。爲什麼？因爲表頭位在第一個 page 之中。在此動作之前，整個 heap 還只處於 reserved 狀態。*HPInit* 然後開始填寫表頭內容。如果面對的是一般的 heap，*HPInit* 必須填寫的欄位包括 size，signature，thread ID。如果 HEAP_NO_SERIALIZE 沒有指定（一般是如此），*HPInit* 會呼叫 *InitializeCriticalSection*，把 heap 表頭中的 CRITICAL_SECTION 物件位址交過去。*HPInit* 並在此時設定自由串列表頭的陣列。

如果 *HPInit* 初始化的是一個 subheap，動作就小多了。這時候表頭中只有兩個 DWORDs：heap 大小，以及指向下一個 subheap 的指標

初始化表頭之後，*HPInit* 產生一個 arena，描述長度爲 0 的區塊，位在 heap 的最尾端 4KB 處。由於最後一個 page 最初也是在 uncommitted 狀態，所以 *HPInit* 呼叫 *hpCommit* 先做 commit 動作。由此，我們可以知道，每一個新的 heap 都用了至少 8KB 的實際記憶體（RAM）：4KB 用於頭，4KB 用於尾。

在產生了長度爲 0 的標兵之後，HPInit 做出了一個巨大的自由區塊。這個區塊跨越從 arena 到標兵區塊（長度爲 0）之間的整個範圍。爲了這項工作，*HPInit* 使用 *hpFreeSub* 函式（稍早描述過）。你可以利用 WALKHEAP 的輸出，檢驗第一個 heap，看到 Win32 heap 中的這些區塊的初始佈局。

**HPInit 的虛擬碼**

```
// 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;
```

## HeapDestroy 和 IHeapDestroy

*HeapDestroy* 只做參數檢驗工作。摧毀 Win32 heap 的工作是由 *IHeapDestroy* 完成。*HeapDestroy* 唯一做的檢驗工作就是看看 hHeap 所代表的區域是否有至少 0xD0 那麼大。

**HeapDestroy 的底擬碼**

```
// 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;
```

和你所猜想的相反，摧毀一個 Win32 heap 並不像釋放 heap 還給作業系統那麼簡單。兩件事情使它比較複雜。第一，所有未以 HEAP_NO_SERIALIZE 屬性產生出來的 heap，都持有一個 critical section 物件。*IHeapDestroy* 檢查是否 heap 擁有如此的物件並適當地釋放之。

另一個複雜因素是 heaps 串列。如果 *IHeapDestroy* 只是單單釋放 heap 的 pages，這個 heap 串列會被破壞掉。*IHeapDestroy* 必須走訪整個串列，並更新之。

在串列被更新之後，*IHeapDestroy* 呼叫 VMM 的 *_PageFree*，釋放 heap 所擁有的 pages。光呼叫一次可能不足以釋放所有的 pages。爲什麼？如果 heap 使用者做了許多的配置，或是一個非常大的配置，*HeapAlloc* 可能產生出額外的 subheaps 並加到 subheap 串列去（記錄在 heap 表頭的 04h 偏移處）。因此，*IHeapDestroy* 必須使用迴路來釋放 main heap 以及任何一個 subheap。

最後請注意一點，程式退出前，系統不會自動呼叫 *HeapDestroy*。我推測如果行程的位址空間消失了，所有的 heap 記憶體也就被釋放了吧。

### IHeapDestroy 的虛擬碼

```
// 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

這是一個 Windows NT 函式，它掃描一個 Win32 heap，檢查其一貫性。其實我看不出有什麼理由它不應該出現在 Windows 95 API 之中。

關於虛擬碼中出現的 *CommonUnimpStub*，請看 *VirtualLock* 那一節，我在那兒有一些敘述。

**HeapValidate 的虛擬碼**

```
EAX = "HeapValidate"
    CL = F3
    JMP CommonUnimpStub
```

## HeapCompact

這是一個 Windows NT 函式，企圖聯合自由區塊並將 Win32 heap 中未使用的 pages 統統 decommit 掉。Windows 95 已經在日常雜務中完成了這些事情，所以它沒有存在的必要。

**HeapCompact 的虛擬碼**

```
EAX = "HeapCompact"
    CL = 2
    JMP CommonUnimpStub
```

## GetProcessHeaps

這是一個 Windows NT 函式，傳回一個由 heap handle 組成的陣列。奇怪的是，Windows 95 API 之中沒有它，雖然它是如此容易實作出來。事實上，TOOLHELP32 的 *Heap32ListFirst* 和 *Heap32ListNext* 函式就是給你這樣的資訊。

**GetProcessHeaps 的虛擬碼**

```
EAX = "GetProcessHeaps"
    CL = 2
    JMP CommonUnimpStub
```

## HeapLock

這是一個 Windows NT 函式，取得 Win32 heap 的 critical section 物件。這又是一個我想不出有什麼理由的缺席函式。*hpTakeSem* 函式所作所爲似乎正是 *HeapLock* 被預期該做的行爲。

**HeapLock 的虛擬碼**

```
EAX = "HeapLock"
    CL = 1
    JMP CommonUnimpStub
```

## HeapUnlock

這是一個 Windows NT 函式，釋放 Win32 heap 的 critical section 物件。和 *HeapLock* 一樣，它也在 Windows 95 API 中缺席了。

**HeapUnlock 的虛擬碼**

```
EsAX = "HeapUnlock"
    CL = 1
    JMP CommonUnimpStub
```

## HeapWalk

這是一個 Windows NT 函式，迭代走訪一個 Win32 heap 的所有區塊。這是 Win32 API 鬧劇的一個絕佳例子。Windows 95 設計者決定忽略 *HeapWalk*，因爲時間不夠。然而，在做下這個決定的時候，他們卻又增加了 TOOLHELP32 的 *Heap32First* 和 *Heap32Next* 兩函式。

**HeapWalk 的虛擬碼**

```
EAX = "HeapWalk"
    CL = 1
    JMP CommonUnimpStub
```

# Win32 的 Local Heap 函式

Win32 的 local heap 和 global heap 函式都是 Win16 的遺跡。在 Windows 95 之中其實已無需要。Win16 的 local heap 之所以產生是爲了讓 EXE 或 DLL 可以不需要改變 selector 就找到其 heap 內容。Global heap 之所以存在則是因爲沒有辦法在不處理 selector 的情況下配置巨大區塊。Windows 95 中的 Win32 程式沒有這兩個限制，所以 Win32 API 可以免除這兩組函式。

然而我們知道，Win32 API 因爲回溯相容而做了某些妥協。有太多的 Win16 程式使用

global heap 和 local heap 函式。如果把它們從 Win32 API 中去除，會使得程式的移植工作走不出實驗室。因此微軟決定保留它們。

絕大部份而言，Windows 95 的 local heap 和 global heap 函式是完全一致的。也就是說，*GlobalAlloc* 和 *LocalAlloc* 雖然同爲輸出函式，但使用相同的 KERNEL32.DLL 位址。*GlobalFree* 和 *LocalFree* 也是相同的情況。稍後，在這些函式的虛擬碼中，我會指出其間的差異。在檢驗 Windows 95 的過程之中，我已經找到共同的 *Global*/*Local* 函式，並將以其 *Local* 名稱來稱呼它們。往後我將遵此慣例。

Win32 local heap 函式對於 Windows 95 的 Win16 元件亦有用途。Windows 95 的 USER 和 GDI 都是 16 位元碼，但許多情況下它們使用 32 位元指標，指向 HWNDs、menus、GDI 物件…。這些物件都放在 Win32 local heaps 中，位於 USER 與 GDI 的 DGROUP 之下。第 4 章有許多有關它們的真實佈局的資料。KRNL386 的 16 位元輸出函式向上呼叫 KERNEL32 的 Win32 local heap 函式。例如未公開的 K209 函式（KRNL386.209）就是向上呼叫 KERNEL32 的 *LocalAlloc* 函式。16 位元的 USER 和 GDI 呼叫 KRNL386.K209 函式以配置記憶體給視窗、DC 等等使用。同樣地，K211 函式向上呼叫 KERNEL32 的 *LocalFree*。

## Win32 local heaps

Local heap 在 Windows 95 之中比在 Windows 3.1 之中簡單。Win32 的 local heap 函式使用我在前面描述過的底層的 Win32 heap 碼。這大量簡化了 local heap 函式碼。例如，呼叫 *LocalAlloc* 並指定 LMEM_FIXED，基本上和呼叫 *HeapAlloc* 是相同的。而在底層，*LocalAlloc* 和 *HeapAlloc* 都是呼叫 KERNEL32 的 *HPAlloc*。

另一個比較簡化的地方在於其較小的 local heaps 個數。在 Win16 中，EXE 和 DLLs 都可以有自己的 local heap（只有存放字形的 DLL 例外）。但預設情況下每一個 Windows 行程只有一個 Win32 local heap。經由 Win32 local heap 函式的配置行爲，記憶體來自 default process heap（稍早前曾描述過）。如果不是爲了 LMEM_MOVEABLE 屬性，你甚至可以這樣就完成 *LocalAlloc* 的設計：

```
HLOCAL WINAPI LocalAlloc(UINT uFlags, UINT cbBytes)
{
    return (HLOCAL) HeapAlloc( GetProcessHeap(), 0, cbBytes );
}
```

稍後你將在虛擬碼中看到，*LocalAlloc* 搭配 LMEM_FIXED 並不是多麼地複雜。倒是 LMEM_MOVEABLE 使事情複雜了一些。你可能會問：那爲什麼還要LMEM_MOVEABLE 呢？幹嘛不乾脆忽略此旗標，把它視爲 LMEM_FIXED？要知道，以 LMEM_MOVEABLE 配置而來的記憶體並不真的能夠在 heap 之中移動。答案是因爲還有許多程式（包括 Windows 本身）使用這樣的事實：LMEM_MOVEABLE handle 是一個記憶體區塊的指標的指標：

```
pMemoryBlock = *(void *)_LMEM_MOVEABLE_handle;
```

把 handle 視爲指標並且使用它，這些程式很可能就不經由 *LocalLock* 而直接獲得一個指標。雖然這並不是好的程式寫法，但是它被廣泛地使用。爲了相容，新的 Windows 必須支援它。

Win32 local heap 函式維護了與其前身的語意相容。如果你要，你真的可以根據 LMEM_MOVEABLE handle 得到一個指標。關鍵性的差別在於，在 Win16 中那是一個兩位元組的近程指標，在 Win32 中那是一個四位元組的近程指標。爲了保持相容的表象，Win32 local heap 函式也使用 handle tables。我在 ***Windows Internals*** 一書第２章描述過它們，只不過格式有些不同。

每一個 Win32 local heap 的 handle table 保有最多 8 個 local handles 的資訊。當程式一次使用的 local handles 超過 8 個，*LocalAlloc* 就再配置另一個 handle table。這些 handle table 佔用它們所代表的區塊的記憶體空間。它們被維護成串列的型式，以便輕易找到自由項目。指向「handle table 串列的頭」的那個指標，記錄在 process database 之中。Handle table 看起來像這樣：

```
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];
};
```

每一個 LOCAL_HANDLE_TABLE_ENTRY 看起來像這樣：

```
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.
}
```

當你以 LMEM_MOVEABLE（或 GMEM_MOVEABLE）配置記憶體，*LocalAlloc* 必須找到一個可用的 LOCAL_HANDLE_TABLE_ENTRY。然後配置一塊符合大小的區塊，並把區塊位址記錄到 LOCAL_HANDLE_TABLE_ENTRY 的 pBlock 欄位去。*LocalAlloc* 傳回的 handle 正是 LOCAL_HANDLE_TABLE_ENTRY.pBlock 的位址。

我們已經知道，LMEM_FIXED 是一個指標而 LMEM_MOVEABLE 不是。你或許會奇怪 KERNEL32 怎麼知道你用的 handle 是哪一種？你可能交給 *LocalFree* 任何一種 handle，不是嗎？容易得很，Local heap handles 如果以 0, 4, 8, 0xC 結束，就是代表 fixed 區塊。如果以 2, 6, 0xA, 0xE 結束，就是代表 moveable 區塊。*HPAlloc* 傳回的所有區塊

位址都是以 0, 4, 8, 0xC 結束，爲了讓 moveable 區塊能夠以 2, 6, 0xA, 0xE 結束，微軟公司「put the pBlock pointer two bytes into the LOCAL_HANDLE_TABLE_ENTRY structure」。附帶一提，Win16 的 local heap handle tables 也有類似的設計。

## LocalAlloc 和 ILocalAlloc

*LocalAlloc* 並沒有太多好看。它呼叫了 KERNEL32 的內部函式 *HouseCleanLogicallyDeadHandles* 之後，即跳到 *ILocalAlloc* 函式去。*HouseCleanLogicallyDeadHandles* 的行爲也許如其名稱所言，然而我從未看出其意義過。所謂 "logically dead handles" 的意思實在不甚明瞭。

**LocalAlloc 的虛擬碼**

```
HouseCleanLogicallyDeadHandles();
    goto ILocalAlloc;
```

*ILocalAlloc* 一開始在 process database 中查詢 default process heap 的位址，接下來，它要求 heap semaphore，並允許程式碼把 HEAP_NO_SERIALIZE 旗標交給稍後將使用的下層函式。這時候，*ILocalAlloc* 分裂爲兩部份，一個處理 LMEM_MOVEABLE，另一個處理 LMEM_FIXED。

如果配置一個 LMEM_MOVEABLE，*ILocalAlloc* 就從 process database 中查詢 free handle list 的頭。如果後者是空的，*ILocalAlloc* 就利用 *HPAlloc* 爲新的 handle table 配置記憶體，並初始化其內容。最終會取得一個 free handle table entry。有了它，*ILocalAlloc* 就可以填寫其欄位，表示它是一個 in-use handle。

在幾乎填寫所有的 handle table entry 之後，*ILocalAlloc* 呼叫 *HPAlloc* 取得一塊記憶體，符合 *LocalAlloc* 所要求的大小。*ILocalAlloc* 再加索 4 個位元組，如此一來它就可以把第一個 DWORD 做爲己用。*ILocalAlloc* 可能會在其中放置什麼東西呢？還不就是 handle table entry 的指標嘛。於是，local heap 函式就可以把一個 moveable 區塊的指標轉換回其 handle。由於第一個 DWORD 是給 local heap 函式用的，*ILocalAlloc* 會把區塊位址

加上 4，才儲存到 handle table 的 slot 之中，當做指標。

其他的碼用來處理 LMEM_FIXED handle。這種情況下呼叫 *HPAlloc* 配置記憶體區塊。區塊位址就是 *ILocalAlloc* 的傳回數（也就是一個 handle）。這和 Win16 的情況是一樣的。

**ILocalAlloc 的虛擬碼**

```
// 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

Win16 的 *LocalLock* 有兩個目的：阻止區塊移動並傳回其位址。Win32 的 *LocalLock* 主要是 handle 檢驗函式 -- 雖然它也傳回位址。在 Win32 之中，local heap 的區塊並不能移動，所以不需要鎖住這些區塊。並且也由於你沒辦法真的移動區塊，所以沒有理由以 LMEM_MOVEABLE 屬性配置記憶體。不過，KERNEL32 還是維護了一個鎖定計數器。

真正的 *LocalLock* 函式只做檢驗工作，它檢驗 hLocal 指標之後的 0x10 個位元組以及之前的 0x7 個位元組是否合法。任何 handle -- 不論是 LMEM_FIXED 或 LMEM_MOVEABLE 都應符合這條規則。如果這項檢驗工作通過，就跳到 *ILocalLock* 。

如果交給 *ILocalLock* 的 handle 是一個 LMEM_MOVEABLE handle，函式會將 handle 減去 2，以獲得指標，指向此區塊之 LOCAL_HANDLE_TABLE_ENTRY 結構。有了這個指標，*ILocalLock* 檢驗 signature（BS）欄位並取出目前的鎖定計數（一個位元組）。如果其值爲 0xFE，*ILocalLock* 將拒絕增加其值。否則函式會增加該值並傳回指標。

如果交給 *ILocalLock* 的 handle 是一個 LMEM_FIXED handle，就沒有所謂的鎖定計數值跟著它。這時候 handle 應該和以 *HeapAlloc* 配置者相同。因此，應該有一個 *HPAlloc* 風格的 arena（0x10 個位元組）在 handle（其實就是位址）之前。*LocalLock* 從此 arena 的 size 欄位中取值，並檢查某些位元。針對 LMEM_FIXED，*LocalLock* 傳出的位址，和其所接收的 handle 其實是相同的。

**LocalLock 的虛擬碼**

```
// 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 的虛擬碼**

```
// 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

這也是一個檢驗層。它檢驗 hLocal 指標之後的 0x10 個位元組以及之前的 0x7 個位元組是否合法。任何 handle -- 不論是 LMEM_FIXED 或 LMEM_MOVEABLE 都應該符合這條規則。如果這項檢驗工作通過，就跳到 *ILocalUnlock* 去。

*ILocalUnlock* 事實上重新播演 *ILocalLock* 的戲碼，只是完全相反罷了。如果交給它的是一個 LMEM_FIXED handle，沒有任何事需要做。甚至於不需要驗證 handle。如果交給它的是一個 LMEM_MOVEABLE handle，*ILocalUnlock* 會先檢查 handle table entry 中的 signature，確定這是一個合法的 handle。然後再檢查區塊的鎖定次數，看看是否可以安全地降低。如果是，就減少鎖定次數並傳回 BOOL 值，表示這一區塊還在鎖定狀態或是已經完全解鎖。

**LocalUnlock 的虛擬碼**

```
// 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
```

**lLocalUnlock 的虛擬碼**

```
// 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

Win32 的 *LocalFree* 有點詭異。在它真正釋放一個由 *LocalAlloc* 獲得的 handle 之前，它會先檢查一個特殊狀況。KERNEL32 和 KRNL386 無意間共謀產生並使用所謂的 handle groups。我並不清楚 handle group 是什麼，因爲我沒辦法找到任何一個。但不論如何，handle groups 是存在於一個 Win16 Task database 和一個 Win32 *LocalAlloc* handle 和一個 handle group 之間的某種三方關係。當 *LocalFree* 偵測到是這種特殊的 local

handle 要被釋放，它會呼叫 *GlobalNukeGroup* 以擺脫 handle group。handle group 串列是由 KRNL386 維護，所以 *GlobalNukeGroup* 會下移呼叫 KRNL386。這是微軟宣稱「KERNEL32 不會移樽就教於 KRNL386」的一個反證。一般而言 *LocalAlloc* 傳回的不會是一個 handle group handle，所以 *LocalFree* 直接呼叫 *ILocalFree*。

**LocalFree 的虛擬碼**

```
// 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* 才是真正做釋放動作的地方。和其他大部份 local heap 函式一樣，處理 LMEM_FIXED 區塊時會比較簡單些。通常面對這種區塊就是直接呼叫一個對應的 *HeapXXX* 函式。本例而言呼叫的是 *IHeapXXX*。

如果面對的是 LMEM_MOVEABLE 區塊，就比較複雜了。在檢查過 handle 的合法性之後，*ILocalFree* 檢查鎖定次數。鎖定次數如果不是 0 是不行的。接下來 *ILocalFree* 釋放 handle 所代表的區塊，經由 *IHeapFree* 把它還給 heap。最後，*ILocalFree* 把對應的 handle table entry 放回可用串列中的第一個位置。

請注意，當 handle table 中的 8 筆資料項目都被釋放之後，*ILocalFree* 並沒有把它清除掉。也就是說它並不是一個良好設計的回收筒。爲了證明我並沒有漏看什麼東西，我修改 WALKHELP 程式，先做 50 次 *LocalAlloc* 動作，然後再全部釋放。輸出結果顯示，所有 handle table 都還在記憶體之中。我們只好說，看來 handle table 有一個良好而規律的「記憶體破碎避免體制」。我們只好說，也許這些 handle tables 可以被重複使用於稍後可能出現的 moveable 區塊的配置動作上。

**ILocalFree 的虛擬碼**

```
// 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

*LocalReAlloc* 只是檢驗層。它檢驗 hLocal 指標之後的 0x10 個位元組以及之前的 0x7 個位元組是否合法。任何 handle -- 不論是 LMEM_FIXED 或 LMEM_MOVEABLE 都應該符合這條規則。如果這項檢驗工作通過，就跳到 *ILocalReAlloc* 去。

**LocalReAlloc 的虛擬碼**

```
// 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 中最長而且最複雜的 heap 函式。和其他 local heap 函式一樣，它也分爲對 LMEM_FIXED 區塊和對 LMEM_MOVEABLE 區塊的兩種處理方式。前者比較簡單，並呼叫 *HPReAlloc*，那是一個內部函式。但是在呼叫之前，*ILocalReAlloc* 會先檢查是否呼叫端嘗試修改 LMEM_FIXED 旗標。那可是不允許的唷。

至於在 LMEM_MOVEABLE 這一方面，*ILocalReAlloc* 先檢查看看是否呼叫端只不過是要修改旗標值。如果確是如此，函式碼就修改 handle 的 LOCAL_HANDLE_TABLE_ENTRY，然後結束。接下來函式碼再檢查 size 參數是否爲 0。如果是，表示呼叫端要求拋棄（discard）這個區塊。*ILocalReAlloc* 的應允方式是，把區塊的 handle 交給 *IHeapFree*。而在做這個動作之前，*ILocalReAlloc* 會檢查區塊是否被鎖住，並且在「是」的情況下發出抱怨。

如果 size 參數不是 0，表示呼叫端要求配置一個新區塊。而如果目前此一 handle 所代表的記憶體區塊大小是 0，表示先前它已經被拋棄了。這時候函式就呼叫 *HPAlloc* 取得一塊符合大小的空間。如果 handle 已經關聯到一塊記憶體，*ILocalReAlloc* 會把此區塊的位址交給 *HPReAlloc*，讓後者去做重新配置的工作。

### ILocalReAlloc 的虛擬碼

```
// 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

*LocalHandle* 只是檢驗層。它檢驗 hLocal 指標之後的 0x10 個位元組以及之前的 0x7 個位元組是否合法。任何 handle -- 不論是 LMEM_FIXED 或 LMEM_MOVEABLE 都應該符合這條規則。如果這項檢驗工作通過，就跳到 *ILocalHandle* 去。

*ILocalHandle* 取得區塊位址並傳回對應的 local heap handle。對於 LMEM_FIXED 而言這是一個簡單的工作，因爲區塊位址和 handle 根本就是相同的。然而，*ILocalHandle* 至少還是做了點事兒：它檢驗 pMem 區塊位址真的是 *HPAlloc* 所獲得的區塊。

另一個劇本用來對付 LMEM_MOVEABLE handles。這裡面有點計謀，但不太多。在 ILocalAlloc 的虛擬碼中，我曾經顯示，針對 LMEM_MOVEABLE 區塊，*ILocalAlloc* 將配置空間增加了 4 個位元組，在最前面的 4 個位元組中，*ILocalAlloc* 放入一個指標，指向 local handle table entry。現在是這 4 個位元組上場的時候了。*ILocalHandle* 只需要將指標減掉 4，就可以讀取一個 DWORD，它應該是個指向 local handle table entry 的指標。*ILocalHandle* 會先檢驗這件事情，然後傳回「handle table entry 的位址加 2」。而我們早已知道，在那個位置內是個指標，指向記憶體區塊。

**LocalHandle 的虛擬碼**

```
// 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 的虛擬碼**

```
// 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

*LocalSize* 只是檢驗層。它檢驗 hLocal 指標之後的 0x10 個位元組以及之前的 0x7 個位元組是否合法。任何 handle -- 不論是 LMEM_FIXED 或 LMEM_MOVEABLE 都應該符合這條規則。如果這項檢驗工作通過，就跳到 *ILocalSize* 去。

*LocalSize* 傳回記憶體區塊的大小。真正的工作是由 *HeapSize* 完成。如果區塊是 LMEM_FIXED，*LocalSize* 幾乎直接就呼叫 *HeapSize*。

如果區塊是 LMEM_MOVEABLE，*LocalSize* 必須首先將 handle 轉換爲指向區塊的指標。這種情況下，*LocalSize* 必須先檢驗 handle 是否爲合法的 local handle。如果是，*LocalSize* 從其 LOCAL_HANDLE_TABLE_ENTRY 結構中取出指標。

最後還有一段碼，但只適用 LMEM_MOVEABLE 區塊。一如我在 *ILocalAlloc* 中所示，LMEM_MOVEABLE 區塊大小比真實需要的大小還多出 4 個位元組，用來放置回指 handle table entry 的指標。爲了讓 *LocalSize* 所傳回的值符合 *LocalAlloc* 所傳回的值，*LocalSize* 必須將「*HeapSize* 針對 LMEM_MOVEABLE 區塊的傳回值」再減掉 4，才傳回。

**LocalSize 的虛擬碼**

```
// 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 的虛擬碼**

```
// 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

*LocalFlags* 傳回值中，最低的位元組是 local heap 區塊的鎖定次數，次低的位元組是區塊的旗標。*LocalFlags* 一開始先檢驗 handle。接下來函式碼兵分兩路。如果 handle 是 LMEM_FIXED handle（也就是以 0, 4, 8, 0xC 結束），函式傳回 0（也就是 flag = LMEM_FIXED，鎖定次數 = 0）。*LocalFlags* 還會檢查 handle 是否指向 *HPAlloc* 區塊，如果不是，會傳回 LMEM_INVALID_HANDLE。

如果 handle 是 LMEM_MOVEABLE handle，*LocalFlags* 會把 handle 值減 2，製造出一個指向 LOCAL_HANDLE_TABLE_ENTRY 的指標。然後，取其 cLock、flags、pBlock 欄位。cLock 直接填入傳回值中，不做任何改變。然而 flag 欄位卻不是由 LMEM_xxx 組成，因此 *LocalFlags* 必須根據 flag 和 pBlock 合成 LMEM_xxx，以便傳回。如果 pBlock 是 0，表示區塊已經被拋棄了（這應該只有在「*LocalReAlloc* 被呼叫時 size 參數爲 0」才會發生）。另外，就像 LMEM_FIXED 一樣，如果 handle 不正確，*LocalFlags* 會傳回 LMEM_INVALID_HANDLE。

**LocalFlags 的虛擬碼**

```
// 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_DISCARDABLE 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

Win32 的 *LocalShrink* 不會影響 heap 本身，因爲 Win32 heap 是不可搬移的。Win16 的 *LocalShrink* 則會傳回 heap 的大小。因此，爲了相容起見，Win32 *LocalShrink* 傳回 default heap 的大小。這或許也是不錯的，因爲 Win32 中沒有好的、公開的方法，取得 default heap

的大小。

**LocalShrink 的虛擬碼**

```
// 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

和 *LocalShrink* 一樣，這個函式的存在只是爲了回溯相容。由於 Win32 heap 不能夠搬移，所以它們也就不能夠緊壓（compact）。

**LocalCompact 的虛擬碼**

```
return 0;       // Easy enough?
```

# Win32 的 Global Heap 函式

Windows 95 的 global heap 函式幾乎已沒有功能。大部份時候，它們只是直接跳進其對應的 local heap 函式去。要不就是像 *GlobalAlloc* 那樣，根本就和對應的 local heap 函式共享同一個進入點。大部份函式都會接受一個 HGLOBAL 參數。*GlobalAlloc* 或 *LocalAlloc* 配置的記憶體都是從 *HPAlloc* 而來。因此，在區塊之前的 0x10 個位元組和之後的 0x07 個位元組之間，應該總是有一塊合法記憶體。

由於 Global heap 函式是如此簡單，虛擬碼即可表明一切，我不需要再多做說明了。

## GlobalAlloc

*GlobalAlloc* 和 *LocalAlloc* 共享相同的函式進入點。

## GlobalLock

**GlobalLock 的虛擬碼**

```
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 的虛擬碼**

```
same tests as GlobalLock
   goto GlobalUnwire;            // JMPs to ILocalUnlock.
```

## GlobalFree

**GlobalFree 的虛擬碼**

```
same tests as GlobalLock
   goto LocalFree;
```

## GlobalReAlloc

**GlobalReAlloc 的虛擬碼**

```
// Parameters:
//     HGLOBAL hGlobal
   same tests as GlobalLock
       goto IGlobalReAlloc;      // JMPs to ILocalReAlloc.
```

## GlobalSize

**GlobalSize 的底層碼**

```
same tests as GlobalLock
       goto IGlobalSize;     ß    // JMPs to ILocalSize.
```

## GlobalHandle

**GlobalHandle 的底層碼**

```
same tests as GlobalLock
   goto IGlobalHandle;           // JMPs to ILocalHandle.
```

## GlobalFlags 和 IGlobalFlags

**GlobalFlags 的底層碼**

```
same tests as GlobalLock
   goto IGlobalFlags;
```

**IGlobalFlags 的底層碼**

```
// 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 的底層碼**

```
goto ILocalLock;
```

## GlobalUnWire

**GlobalUnWire 的虛擬碼**

```
goto ILocalUnlock;
GlobalFix
```

## GlobalFix

**GlobalFix 的虛擬碼**

```
// Parameters:
//      HGLOBAL hMem

    if ( hMem != 0xFFFFFFFF )
        return GlobalLock( hMem );  // GlobalLock ultimately calls ILocalLock.
```

## GlobalUnfix

**GlobalUnfix 的虛擬碼**

```
// Parameters:
//      HGLOBAL hMem

    if ( hMem != 0xFFFFFFFF )
        return GlobalUnlock( hMem );    // GlobalUnlock ultimately
                                        // calls ILocalUnlock.
```

## GlobalCompact

**GlobalCompact 的虛擬碼**

```
goto LocalCompact;
```

# 雜項函式

最後這幾個函式沒有辦法符合前面的分類標準，但它們也相當重要。我沒有辦法涵蓋每一個記憶體函式，取而代之，我將選擇一些有趣的函式。畢竟這一章實在有夠長的了。

## WriteProcessMemory 和 ReadProcessMemory

這是兩個被核准用來讀寫另一行程之記憶體的函式。爲了使用它們，你必須先獲得另一行程的 handle。然而 Win32 API 不提供什麼方便的作法，讓你輕易達此目標。這兩個函式是 Win32 除錯器的關鍵函式。除錯器是屬於那種「必須讀寫其他行程之記憶體」的另類軟體。

兩個函式的底層十分類似。因此我決定只顯示其中一個的虛擬碼。唯一明顯的差異在於 *WriteProcessMemory* 呼叫 VWIN32 的 0x002A0017 service，而 *ReadProcessMemory* 呼叫 VWIN32 的 0x002A0016 service。

*WriteProcessMemory* 首先做同步化控制。它先確定它沒有持有 Win16Mutex 和 Krn32Mutex，然後進入「必須完成」狀態中 -- 意思是不能在中途被切換出來。然後確定來源端的位址（source address）位於應用程式私有的 arena 中（也就是 VMM 文件中說的 4MB~2GB 之間）。

接下來 *WriteProcessMemory* 取得指標，指向來源端的行程結構，再從其身上獲得執行緒串列。爲了某些理由，「實際執行記憶體複製動作」的 VWIN32 service，希望獲得目標行程（target process）之現行執行緒的 ring0 stack 位址。一旦每一項東西都到手，它就索求 Krn32Mutex 並呼叫 VWIN32 service 0x002A0017。在 VWIN32 完成其 memory context 之神奇魔法後，*WriteProcessMemory* 釋放 Krn32Mutex，並呼叫 *LeaveMustComplete* 以退出「必須完成」狀態。如果這個程序中有某些事情錯了，就呼叫 *SetLastError*，讓呼叫端知道錯在哪裡。

**WriteProcessMemory 的虛擬碼**

```
// 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

*GlobalMemoryStatus* 可以很方便地觀察記憶體的狀態。這個函式填充 MEMORYSTATUS 結構內容，像是有多少 pages 已經映射至實際的 RAM、置換檔（swap file）的大小等等。這個函式頗類似 Win16 的 *MemManInfo*。

*GlobalMemoryStatus* 其實只是個參數檢驗層。它只確定一件事：傳給函式的指標指向一塊足以放置 MEMORYSTATUS 結構的記憶體。別管文件上說什麼，你不需要在呼叫 *GlobalMemoryStatus* 之前先初始化 MEMORYSTATUS 結構中的 dwLength 欄位。

**GlobalMemoryStatus 的虛擬碼**

```
// 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* 的工作就是利用一個縮簡版的 DemandInfoStruc 結構內容，填寫 MEMORYSTATUS 結構。前者內容係呼叫 VMM.VXD 的 *_GetDemandPageInfo* 函式而獲得。由於 ring3 程式不能夠直接呼叫 VxDs，*IGlobalMemoryStatus* 使用 VMM 的 0x0001001E service 做爲代理人。爲了那些沒有 DDK 文件的讀者，我列出 DemandInfoStruc 結構給你看：

**DemandInfoStruc 結構**

```
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
```

毫無疑問，將會有許多程式被設計出來，出現在螢幕的一角，告訴你目前的「記憶體負荷」。什麼是「記憶體負荷」？從虛擬碼中你可以看出，它是「committed page 頁數的 50 倍」除以「被 Windows 95 記憶體管理器所掌管的實際記憶體」。換句話說，它是 committed pages 比上 physical pages 的百分比的一半。假設系統有 8 MB 的 RAM，11MB 的 committed pages，那麼記憶體負荷是：

```
(11 x 50) / 8 == 68.75
```

是的，你可以有比實際記憶體更多的 committed pages。一個 committed page 並不意味就一定映射到 RAM。除非你將它 "pagelock"，Windows 95 可以自由自在地將它 "page out" 出去。

**IGlobalMemoryStatus　Procedure**

```
// 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

當我看到 *GetThreadSelectorEntry*，我很震驚它也出現在 Win32 API 隊伍中。這個函式並沒有對執行緒有任何動作，事實上，hThread 參數只是爲了檢驗，卻從來沒有用到。*GetThreadSelectorEntry* 給你一個唯讀機會，處理 system VM 的 local descriptor tables（LDTs）。這是特定的 descriptor table，用來記錄 Win32 程式的 flat 程式碼節區和資料節區。Win16 程式也是經由它取得它們的程式碼節區和資料節區，以及 *GlobalAlloc* handles。這個函式對於任何探索系統的工具軟體，都是非常有用的一個利器。

如果你傳給 *GetThreadSelectorEntry* 一個合法的 selector，就可以取回一個 8 位元組的結構，格式與 LDT descriptor 相同。由於 Win32 程式有一個 flat 指標，可以到達任何地方，所以可利用此函式將 16:16 位址轉換爲一個 flat32 位址，於是 Win32 程式可以讀取該處的內容，也可以將資料寫入該處。你甚至於可以將 Win16 的 *GetSelectorBase* 和 *GetSelectorLimit* 函式建構成屬於你自己的 Win32 版。

說到 *GetSelectorLimit*，在 ***Unauthorized Windows 95*** 一書第 449 頁，這一段碼用來獲得 selector 的 base（基底位址）和 limit（節區大小）。這段碼使用一個 VWIN32 VxD service call，引發 DPMI #6 號子功能。該子功能傳回指定之 selector 的基底位址。這個作法在技術上予人印象深刻，而 *GetThreadSelectorEntry* 可以做得一樣好，而且更簡單。更好的是，*GetThreadSelectorEntry* 是個公開函式，應該比任何未公開函式優先使用。

*GetThreadSelectorEntry* 函式中主要的趣味在 LDTAlias 和 LDTPtr 兩變數身上。這是 KERNEL32.DLL 的兩個全域變數。LDTPtr 內含 system VM 的 LDT 的線性位址，LDTAlias 則是一個 selector 值，擁有對 selector table 的記憶體的讀寫權力（請看 ***Windows Internals*** 第2章）。

**GetThreadSelectorEntry　procedure**

```
// 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 procedure

```
// 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++編譯器提供的 malloc 和 new

許多時候，程式員會忽略作業系統提供記憶體管理函式，使用 C runtime library（特別是 *malloc* 和 *free* 函式）來做記憶體管理。C++ 呢？就我所知，所有 PC 上的 C++ 編譯器，*new* 運算子都被直接對應到 *malloc*，*delete* 運算子則被對應到 *free*。問題在於，這些函式如何使用底層的作業系統能力？

在這一章中，我已經告訴你，heap 函式（如 *HeapAlloc* 和 *HeapFree*）是多麼接近 *malloc* 和 *free*。這意思是說 *malloc* 和 *free* 只是 *HeapAlloc* 和 *HeapFree* 的另一個包裝嗎？至少直到 Visual C++ 4.0 爲止，答案是否定的。唯一的例外是 CRTDLL.DLL -- 微軟的 C runtime library。在 CRTDLL.DLL 之中，*malloc* 和 *new* 都只是簡單地呼叫 *HeapAlloc*，*free* 和 *delete* 則呼叫 *HeapFree*。CRTDLL.DLL 使用於許多標準的 Windows NT 和 Windows 95 的 EXEs 和 DLLs 身上。這是一個偉大的想法，讓微軟不必爲了「不同的 EXE 和 DLL 身上有不同的 C runtime library」而傷腦筋。

不幸的是，C 編譯器廠商不捧場，他們沒有讓每一個人都使用 CRTDLL.DLL。因此，每一個 EXE 之內還是可能各有一個 C runtime llibrary。這種情況短時間內不會改變，所

以我們最好知道這些 runtime library 的底層在做些什麼。

我並不打算細探 *malloc* 和 *free*。取而代之的是，我將給你足夠的概觀，讓你足以自我判斷，該如何設計你自己的記憶體管理體系。

到目前爲止我已經能夠確定，Borland 和 Microsoft 以類似的方法實作其 runtime library 的 heaps。事實上，除了大小之外，整個形勢並沒有和 Windows 3.1 時代有太多改變。每一個 EXE 和 DLL 都有它自己的 heap。程式如果使用三個 DLLs，就會有四個分離的 heaps（EXE 有一個，DLLs 各有一個）。DLL 所配置的記憶體將來自 DLL 的 heap。這和 Win32 的 *HeapAlloc* 不同，後者一定是從 EXE heap 取記憶體（假設你總是將程式的 default heap handle 丟給它）。

C 編譯器的 RTLs（RunTime Librarys）並不使用作業系統的高階 heap 函式如 *HeapAlloc*，它們使用自己的資料結構和記憶體管理碼。於是，要把 *malloc* 所獲得的記憶體和 *HeapAlloc* 所獲得的記憶體混合起來使用就很困難了。

只要挖掘 C/C++ RTL 夠深，我們就可以了解如何將 *malloc* 對應到底層的作業系統函式。我潛伏到 Borland C++ 4.5 RTL 原始碼中，那是件困難的工作，我很高興你不必再做一次。下面這段 call stack 顯示，*malloc* 如何實作於 Windows 95 函式之上：

```
malloc (HEAP.C)
    _getmem (GETMEM.C)
        _virt_reserve (VIRTMEM.C)
            VirtualAlloc( NULL, size, MEM_RESERVE, PAGE_NOACCESS )
```

啊哈，C RTL 利用 *VirtualAlloc* 向作業系統配置大塊記憶體，那正是 *HeapAlloc* 的行爲。被配置的區域最初處於 reserved 狀態，然後再以 _virt_reserve 將它們 “commit” -- 在它們被使用之前一刻。*_virt_reserve* 其實只是 *VirtualCommit* 的一個外包函式。這個過程聽起來熟悉嗎？是的。這正是 Windows 95 的作法。如果你需要複習，請回頭看看 *hpCarve* 和 *hpCommit* 那一節。

RTL 並不會在每次你呼叫 *malloc* 時就呼叫 *VirtualAlloc*。它們需要設定並維護內部資料結構並持續追蹤什麼區塊被配置，什麼不是...。同時也保持一個自由串列，以便做快速配置之用。這些 heap 區塊看起來像什麼呢？以下是 HEAP.C 中的說明：

```
/*-----------------------------------------------------------------------
 * 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.
```

嗯...，Windows 95 零售版管理 heap 區塊的作法類似，但並不完全一樣。*HeapAlloc* 對每一區塊的額外負擔只有 4 個位元組，Borland C++ 的 RTL 卻需要 8 個位元組。Note also the similarities in how Borland C++ and the *HeapAlloc* function use memory in a free block to point to another free block.（譯註：抱歉，我對這段話沒有感覺，只好留原文給您）

當你使用 RTL 提供的 heap，一個潛在性問題是其生命期。當一個 DLL 被剔出記憶體並且收到了 DLL_PROCESS_DETACH 通告訊息，runtime library 會呼叫 *VirtualFree*，將 heap 的記憶體釋放掉。如果另一個 DLL 有個指標指向這記憶體中的區塊，指標突然之間就失效了。如果該 DLL 稍後也被剔出記憶體，並在其 DLL_PROCESS_DETACH 處理過程中使用了這些指標，程式會完蛋，而且不容易除錯。這段心得是從有痛苦經驗的人那兒學來的。

那麼，回答我最初的問題，*malloc* 基本上是 Windows 95 的 *HeapAlloc* 函式的編譯器版本，但至少有兩個關鍵差異。第一，每一個 EXE 和 DLL 都有它們自己的 heap，由 RTL 提供，而 *HeapAlloc* 所配置的記憶體一概來自系統所設定的 default process heap。在某種操作程序下，使用 RTL 的 heap 可能會造成問題。這並不是說你應該避免使用 *malloc* 或 *new*，只是要告訴你它們是什麼，以及讓你了解其中的潛在性問題。

## 摘要

哇歐，這一章真是漫長的一役，赤裸裸的地碰觸了 Windows 95 記憶體管理中的各種觀念。我們已經驗證了 CPU 的分頁機制，每一個行程的位址空間，以及分享給每一個行程的那些位址空間。在 Win32 API 方面，我們看到了 *VirtualXXX* 函式如何管理 pages，也看到了 *HeapXXX* 函式如何提供更高階的記憶體管理。從 Win16 衍生而來的 heap 函式如 *GlobalXXX* 和 *LocalXXX* 其實只不過是 *HeapXXX* 的輕薄包裝而已。下一章我們將看到 ring3 KERNEL32.DLL 如何與 ring0 的虛擬記憶體管理器溝通，以便獲得基本的服務，並在其上構築 heap 相關函式。

# VWINKERNEL32386

（VWIN32.VXD,　KERNEL32.DLL,　KRNL386.EXE）

有一天，我和幾位同事待在辦公室裡，思考 Windows 95 的核心架構。和平常一樣，我們的主題圍繞在 Windows 95 的各個元件如何能夠對其他元件有詳細而直接的知識。我的一位同事就很奇怪：爲什麼微軟要自找麻煩地把 16- 和 32- 位元核心分開來，再加上一個核心的 VxD？爲什麼不把它們統統塞到一個檔案裡頭去？

這一章，我們就來討論這個主題。事實上這一章的章名（VWINKERNEL32386）正是我即將說明的元件的簡略稱呼：VWIN32.VXD，KERNEL32.DLL，KRNL386.EXE。我得警告你，這一章有許多進階的東西。你並不需要先了解這一章才能夠進行後續各章。

Windows 程式員可以立刻分辨出 KRNL386.EXE 是 16 位元核心，KERNEL32.DLL 是 32 位元核心。這些 DLLs 開放出一些核心函式（像 *LoadLibrary*、*_lread* 等等），是每一個 Win16 和 Win32 程式都必須用到的。這些函式絕大部份都在 SDK 或其他編譯器所提供的標準系統表頭檔中有些說明。對 16 位元程式而言，WINDOWS.H 檔有 KRNL386.EXE 的所有函式原型。而在 Win32 這一邊，WINBASE.H 和 WINCON.H 描述了大部份的 KERNEL32.DLL 函式。

不幸的是，上述的第三個核心元件（VWIN32.VXD）只勉勉強強在微軟的說明文件或表頭檔中有些許介紹。事實上，以我所知，唯一一份和 VWIN32.VXD 有關的資料就是 Windows 95 DDK 中的 VWIN32.H 檔案。稱 VWIN32.H 爲文件實在有點誇張，特別是當你知道 VWIN32.VXD 是兩個最重要的 VxDs 之一時（另一個是 VMM）。VWIN32.VXD 提供關鍵性的 ring0 作業系統元素，給 16 位元的 KRNL386 和 32 位元的 KERNEL32 使用。我已經一再說過，只要你是以嚴肅的態度檢視 KRNL386 和 KERNEL32，很快你就會進入 VWIN32.VXD 的領域。

VWIN32.VXD 的文件如此短缺，我企圖在本章中彌補這種情況。首先，我將描述 VWIN32.VXD 及其介面，然後我將顯示這三個核心元件是如何地互相關聯，並且互相了解。我區分下列數個子題：

- KRNL386.EXE 了解並且呼叫 VWIN32.VXD
- KRNL386.EXE 了解並且呼叫 KERNEL32.DLL
- KERNEL32.DLL 了解並且呼叫 KRNL386.EXE
- KERNEL32.DLL 了解並且呼叫 VWIN32.VXD
- VWIN32.VXD 了解並且和 KERNEL32.DLL 交換資訊
- VWIN32.VXD 了解並且和 KRNL386.EXE 交換資訊

在這些排列組合中，我們最感興趣的是「KERNEL32.DLL 呼叫 KRNL386.EXE」。微軟發誓說那是不可能的，但 ***Unauthorized Windows 95*** 證明微軟說了謊話。我會在本章提供一張詳細名單，告訴你哪一個 KERNEL32 函式呼叫了 KRNL386.EXE。

***Unauthorized Windows 95*** 觸及的另一個題目是 Win32 VxD services。這些 services 提供給 Win32 程式一種「使用標準的 "C 呼叫習慣" 來呼叫 VxDs」的簡易方法。Win32 VxD services 是 Windows 95 架構的主要部份，例如每一個 file I/O call 就是都轉換爲一個 Win32 VxD services call（事實上它們最終是呼叫 VMM.VXD 的 *Exec_PM_Int*，並給予中斷號碼 21h。聽起來滿耳熟的？是呀，DOS 沒有死，不是嗎？）

不幸的是，微軟從未正式公開 Win32 VxD services。由於 ***Unauthorized Windows 95*** 只貢獻出數頁，談及一些重要的 Win32 VxD services，所以我要在這一章深入地討論它。通常，學習並刺探未公開介面的最佳作法就是寫工具程式，因此這一章包括一個名爲 W32SVSPY 的程式，監視對 Win32 VxD services 的呼叫。我必須克服一些障礙 -- 某些是微軟故意造成的 -- 才能夠讓 W32SVSPY 有效運作。在本章最後，我會說明 W32SVSPY 如何玩弄神奇的手法。

## 臨時抱佛腳，談談 VxD

由於我即將在後續數頁中討論 VWIN32.VXD，先說點 VxD 的基礎應該至少有點幫助。對於並不專注於 VxD 的讀者（我自己也是），快速做一次概觀是適當的。如果你已經寫過 VxD，我想你可以跳過這一節。

正如其名所示，VxD 是一個 Virtual Device Driver（虛擬裝置驅動程式）。也就是說它可以用來將某個硬體裝置虛擬化，使許多程式都可以用它。沒有人說過 VxD 一定得和一個實際裝置產生關聯。VxD 事實上只不過是一個 DLL，在最高權限（ring0）中執行。由於 VxD 在 ring0 執行，基本上沒有什麼是它不能夠做的。當然，很公平地，它比較難寫，也比較不容易被 ring3 程式呼叫。

我可不打算描述如何撰寫 VxDs，或是告訴 VxD 撰寫者一些漂亮的技術。***Unauthorized Windows 95*** 以及 Dave Thielen 的 ***Writing Windows Device Drivers*** 都涵蓋了 VxD 主題，並且比這裡的討論深入得多。我解說 VxD 的目的只是爲了解說 VWIN32.VXD。

被載入記憶體之後，VxDs 有一個獨一無二的 16 位元識別碼（ID）。VMM.VXD 的 device ID 是 1，Virtual Keyboard Device（VKD）的 device ID 是 0Dh，VWIN32.VXD 的 device ID 是 2Ah。你可以從 Windows 95 DDK 的 VMM.H 或 VMM.INC 檔案中找到 xxx_DEVICE_ID 定義，從而獲得標準並預先定義的 VxDs IDs 的完整列表。請注意其中 512 以下的 IDs 保留給微軟使用。其他公司如果撰寫 VxDs，必須向微軟申請專用的 VxD IDs。

## 從另一個 VxD 中呼叫 VxD 函式

就像 ring3 system DLLs 有一個標準方法將函式輸出（exports）給其他 EXEs 或 DLLs 呼叫一樣，當 VxD 呼叫另一個 VxD 函式，也有一些規範必須遵守。VxD 的所有可被呼叫的函式都列於一個陣列之中。每一個函式稱爲一個 service。當 VxD 呼叫另一個 VxD 的 service，它並不使用其名稱，而是使用陣列中的索引號碼。以 VMM.INC 的片斷爲例：

```
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
```

Get_VMM_Version 就是 VMM service 0，Get_Cur_VM_Handle 就是 VMM service 1，Test_Cur_VM_Handle 則是 VMM service 2，依此類推。

VxD service 的呼叫機制十分有趣。和 ring3 system DLLs 不同，VxD 載入器並不修補原先的 VxD 碼的 CALL 指令以放入呼叫目標（service）的真正位址。事實上在 VxD 之中根本沒有 CALL 指令。取代 CALL 指令的是：

```
INT 20h
DD  device_and_service_number  ;; 每一個 VxD service 都有不同的值
```

INT 20h 之後的 DWORD 並不是任意數值，它的較高字組內含 device 號碼（稍早我說過了），較低字組內含 service 號碼。因此，一個呼叫 Test_Cur_VM_Handle 的動作應該是這樣：

```
INT 20h
DD  00010002h ;; 0001 = VMM device ID, 0002 = service # for Test_Cur_VM_Handle
```

另一個例子呼叫 VWin32 的 GetSystemTime service：

```
INT 20h
DD  002A0002h ;; 002A = VWin32 device ID, 0002 = service # for GetSystemTime
```

當 ring0 的 INT 20h 處理常式被喚起，它先檢查中斷指令後面的那個 DWORD 內容，使用其中的 device 號碼和 service 號碼來查詢位址。如果你怕這會太慢，別擔心。一旦一個 INT 20h 被喚起，INT 20h 處理常式就會把程式碼修補爲一個真正的 CALL 指令。這樣的發展很理想，因爲 INT 20h 之後緊跟著一個 DWORD，共耗掉 6 個位元組，那不正是一個近程的 32 位元間接呼叫（也就是 call DWORD PTR[xxxxxxxx]）所耗用的記憶體數量嗎？VxD 載入器並不會在載入時間修補所有的 CALL 指令，它只修補真正被使用的程式碼位置。

當 service 號碼的較低字組的最高位元（0x8000）被設立，VxD 中的 INT 20h 的動態修正動作有一些改變。這種情況下，程式碼會被修正爲 JMP，而非 CALL：

```
INT 20h
DD  00018002h
```

## 從 Win16 保護模式碼叫呼叫 VxD 函式

如果只有 VxD 才能夠呼叫其他的 VxDs，Windows 就相當無聊了。由於 VxDs 可以爲所欲爲，暢所欲行，一定要有一種「ring3 程式碼呼叫 VxD」的方法存在，一切才顯得有趣。這樣的能力使 ring3 程式做到原本不可能做到的事情。只要你在寫應用程式時踢到鐵板，你就可以寫一個 VxD 並從應用程式中呼叫它。

某些人（包括我）主張這樣的策略應該謹慎使用。任何人都可以寫一個 VxD 爲所欲爲 -- 甚至是對系統有害的行爲（不管是故意的或不是故意的）。我個人認爲如果你能夠避免寫 VxD，你應該儘量避免。系統之中無限權力的東西到底還是愈少愈好。我非常憂慮有一天我的硬碟之中充斥著一大堆 VxDs，只因爲沒有經驗的程式員以爲 VxD 是解決問題的唯一途徑。

啊，把我的個人看法放在一邊吧。從 DOS 程式或 Win16 程式中呼叫 VxDs 雖然有點痛苦，但並不困難。VxD 可以開放一組被 V86 模式、真實模式、ring3 保護模式（Win16 碼）等各種程式呼叫的函式。VxD 把「來自 V86 程式的呼叫」和「來自 ring3 保護模式程式的呼叫」的進入點分開，不過它也可以把它們併在一起。

爲了從 V86 或 16 位元保護模式碼中呼叫 VxD，應用程式首先必須索求一個位址，它可以在此位址上做出一個遠程呼叫。這個位址可以經由 INT 2Fh（AX 放置 1684h）獲得。BX 則應該放置 16 位元的 VxD ID。回返之後，ES:DI 內含一個 16:16 遠程指標。應用程式可以呼叫該指標，將控制權轉換到 ring0 VxD。

讓我們看看 KRNL386 的片斷內容，它顯示 KRNL386 如何獲得 VWIN32.VXD 的進入點：

```
XOR     DI, DI          ; Zero out ES:DI in case the operating 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
```

那些熟悉 Intel 保護模式架構的人，可能會搔破頭，驚訝爲什麼上述的碼能夠有效運作。要知道，ring3 碼是不能夠呼叫 ring0 碼的，有一些保護機制會阻止它（對於各個 ring 層級以及保護機制的討論已經超越了本章的範圍）。Ring3 碼如果嘗試呼叫（或說載入）一個有著 ring0 層級限制的選擇器，會引發 GP fault -- 除非經過重新安排。Intel 架構支援一種很少被使用的機制，稱爲 call gates。Call gates 允許 ring3 碼在受控制的情況下呼叫 ring0 碼。

如果你啓用一個系統除錯器（如 SoftIce/W 或 WDEB386），並將前面程式片斷中的 INT 2Fh AX=1684h 傳回的位址反組譯，你會看到像這樣的東西：

```
:u 3B:03d0
003B:000003D0   INT     30      ; #0028:C025DB52        VWIN32(04)+0742
003B:000003D2   INT     30      ; #0028:C0002BC9        VMM(01)+1BC9
003B:000003D3   INT     30      ; #0028:C022F713        VMM(0D)+0713
```

唔...看起來很奇怪。INT 2Fh 所傳回的函式進入點竟指向一個 INT 30h 指令。怎麼回事？原來呀，Windows 使用一個 INT 30h 強迫 CPU 從 ring3 提昇到 ring0。要知道，任何中斷或異常情況都會暗中引發 CPU 把控制權交給記錄在中斷描述表（Interrupt Descriptor Table，IDT）中的 ring0 處理常式，而 Windows 95 的 INT 30h 處理常式就使用被引發的 INT 30h 指令的 CS:IP，尋找一個 ring0 位址，此一位址便是控制權即將轉移過去的地方。在上述列表中，逗點之後的位址就是每一個 INT 30h 處理常式的位址（SoftIce/W 知道如何找出 INT 30h 處理常式所使用的派送表格並解碼，所以它能夠顯示那些位址）。VWIN32.VXD 的 INT 30h 處理常式位址落在 VWIN32.VXD 之內，這一點也不令人驚訝。如果我們更進一步將該一 ring0 碼位址反組譯，我們會獲得：

```
: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
```

第一個指令需要解釋。當一個 VxD V86/PM API 函式被呼叫，應用程式並不會把參數壓入堆疊。主要原因是 ring0 VxD 使用的堆疊與 ring3 程式使用的堆疊不同。當 CPU 在保護層級中切換，它也把堆疊暫存器切換一個新值，使它指向另一個堆疊，該堆疊乃特別設計用來給新的 ring 層級使用。

由於 ring3 碼不能夠把打算給 VxD 函式的參數壓入堆疊之中，只好在 INT 30h 之前先把參數放到暫存器。當 INT 30h 處理常式呼叫適當的 ring0 函式，它會把一個指標交出去，該指標指向一個結構，內含 ring3 的暫存器值。這是一個 32 位元的 flat 指標，放

在 EBP 暫存器中，所指向的結構稱爲 Client Register Structure（CRS，請參考 VMM.INC 中的 Client_Reg_Struc）。那些提供 APIs（可被 V86 或 16 位元 ring3 保護模式呼叫）的 VxDs 知道，它們能夠經由 EBP 所指的結構，讀取並修改 ring3 暫存器的值。

上述列表的第一個指令，是以 INT 30h 發生當時的 AH 暫存器值寫入 EAX。習慣上，從 V86 或 ring3 16 位元保護模式呼叫 VxD 函式時，函式號碼正是放於 AH 暫存器。如果函式號碼在服務範圍之內，就使用一個 JMP table 把控制權交給適當的 VWIN32.VXD 函式進入點。如果函式號碼不在範圍之內，就顯示一段錯誤訊息。

## 從 Win32 碼中呼叫 VxD 函式

剛才我所描述的兩個 VxD 介面可以回溯到 Windows 3.0。Windows 95 並沒有對它們做任何基本上的修改，反倒是添了一個新介面。由於 Windows 95 不但支援 DOS 程式和 Win16 程式，它還能夠跑 Win32 程式，所以毫不令人訝異地，微軟提供了一款方式，讓 Win32 碼呼叫 VxD。現在，VxD 介面有了四個（一是 ring0 VxD services，二是從 V86 程式中呼叫，三是從 ring3 Win16 程式中呼叫，四是新介面，馬上就要談到）。

新介面只對 ring3 Win32 程式有效，所以其中的函式被稱爲 Win32 VxD services。可別把這個名稱和一般所謂的 VxD services 搞混，後者是指可以被其他 VxDs 呼叫的 VxD 函式。也別把 Win32 VxD services 和 Win32 services 搞混，後者存在於 Windows NT 之中，比較像是一個 daemon process，和 Win32 VxD services 一點關係也沒有。

爲了某些沒有道理的原因（至少我這麼認爲），微軟決定把 Win32 VxD services 介面隱藏起來。也許這是爲了阻止人們寫移植性低的程式吧，因爲 Windows NT 不支援 VxDs。微軟要你透過 *DeviceIoControl* 這個 Win32 API，那是一個在 Windows 95 和 Windows NT 之間半移植性的函式。問題是，*DeviceIoControl* 的速度比起直接使用 Win32 VxD services 慢。事實上在 Windows 95 之中 *DeviceIoControl* 最後會呼叫一個 Win32 VxD service。

比起過去的版本，Windows 95 有更高比例的內容是以 C 語言完成，所以 Win32 VxD service 介面的唯一自然風格就是做成 C 語言的可被呼叫函式。也就是說，Win32 VxD services 可以輕易地以 ring3 碼中的 C 語言呼叫之。Win32 VxD service 的參數是以堆疊傳遞，就像正常函式一樣。對於其他的 VxD 介面而言，這是一個明顯的改善，因爲其他介面通常是以組合語言喚起，參數因而必須放在暫存器中。

### 以 SoftIce/W 觀察 VxD Interfaces

SoftIce/W for Windows 95 知道我在本節所說的每一種 VxD 呼叫介面，包括最新的 Win32 VxD services 介面。你可以使用 VxD 命令，搭配一個 VxD 名稱。例如：

```
:VXD REBOOT
VxD Name Address  Length  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
```

第一組訊息提供我們豐富的資料。從 PM 欄為 "Y" 我們得知，REBOOT 為 ring3 16 位元保護模式程式提供了一個呼叫介面。從 V86 欄為 "N" 我們亦得知，REBOOT 未提供呼叫介面給 V86 程式。另外我們也看到了，REBOOT device 有四個一般的 VxD services（可被其他 VxDs 呼叫）和兩個 Win32 VxD services。

請注意最後三行，表示有兩個 Win32 VxD services。每一行都包括了該 service 的進入點，以及 DWORD 參數的個數。從這些資訊（或是 VMM.INC），你可以推演出每一個 Win32 VxD services 有一個 8 bytes 的結構：

```
DWORD  pfnService;   // service 函式的位址
DWORD  cParams;      // DWORD 參數的個數
```

我稱此結構為一個 service table entry。當一個 VxD 啟動，它必須向系統註冊其 Win32 VxD services。它必須呼叫 VMM.VXD 的 *_Register_Win32_Services* 以完成此事。*_Register_Win32_Services* 的參數之一就是一個指向 Win32 VxD service table 的指標。此一指標儲存在 VxD 的 DDB（Device Description Block）之中，而我們就是 SoftIce/W 取得所有資訊的地方。

呼叫一個 Win32 VxD service，其方法和呼叫其他介面的函式完全不同。我們不再引發一個中斷（INT）或是呼叫一個函式指標，而是呼叫一個未公開的 KERNEL32 函式，*VxDCall*。呼叫之前，呼叫端先把所有參數推進堆疊之中，最後放進一個 DWORD，然後才放進 *VxDCall* 動作。DWORD 的較高字組是 VxD 號碼，較低字組則是一個從 0 開始的索引值。這個索引值指向 Win32 VxD service table，而不是指向一般的 ring0 VxD service table。

實際看個例子會比較清楚些。下面的碼引發 VWIN32.VXD 中的 *VWIN32_sleep* 函式。那是 VWIN32.VXD 所提供的第 10 個 Win32 VxD service，所以其函式號碼爲 9。

```
PUSH    DWORD PTR [EBP+08]  // Push a parameter.
PUSH    002A0009            // 002A = VWIN32, 0009 = VWIN32_sleep
CALL    VxDCall
```

*VxDCall* 是一個 stdcall 函式，意思是參數由右而左傳遞，並且由被呼叫端負責清除堆疊。上面的碼如果以 C 寫成，應該是這樣：

```
VxDCall(0x002A0009, parameter);
```

如果你把 KERNEL32.DLL 的所有輸出（exported）函式都傾印出來看，你會發現前 8 個函式進入點統統映射到同一位址，這個位址就是 *VxDCall*。爲什麼如此？長話短說，基本上那些進入點是 *VxDCall@0*、*VxDCall@4*、*VxDCall@8...VxDCall@28*。微軟的 C 編譯器把 stdcall 函式（例如 *VxDCall*）的名稱加料，加上一個 @ 符號，再附上參數的總位元組個數。由於不同的 Win32 VxD services 有不同的參數，所以呼叫 *VxDCall* 時有可能一個被轉換爲 *VxDCall@4*，另一個卻被轉換爲 *VxDCall@16*。讓多個進入點的名稱有些輕微變化，聯結器就能夠分析出不同參數的 *VxDCall* 呼叫。本章之中我將把所有這些 *VxDCall@xx* 統統稱爲 *VxDCall*。如果你看過 ***Unauthorized Windows 95***，請注意，該書把 *VxDCall* 稱爲 *VxDCall0*。

讓我再整理一次。Win32 碼呼叫 Win32 VxD service 時，首先把每一個參數壓入堆疊之中，然後再壓 DWORD service ID。這個 DWORD ID 可以定義出被呼叫的 VxD，以及其中被呼叫的 Win32 VxD service。最後，程式碼呼叫 KERNEL32.DLL 中的 *VxDCall* 函式。當 Win32 VxD service 回返，執行權立刻到達 *VxDCall* 之後的下一個動作，而所有參數也都在堆疊中被清乾淨。

這就是從外部觀察到的 Win32 VxD service 呼叫動作。現在讓我們跳進去看看 Win32 VxD services 是如何被完成。首先我們看 *VxDCall* 函式碼：

```
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 ring0.
```

最前面兩個指令是把 DWORD VxD service ID 從堆疊中取出，並放到 EAX；而放置傳回值的空間（被 *VxDCall* 這個 32 位元近程呼叫壓入）會在堆疊中滑動，佔據原先被 Win32 VxD service ID 佔有的地方。第三個指令是一個 32 位元遠程呼叫，呼叫 INT 30h。

嘿，我們不是已經看過 INT 30h 了嗎？它是「V86 碼和 Win16 保護模式碼呼叫 VxDs」的一種方法。然而，這裡並不是一個正常的 INT 30h 指令：

```
u 3B:000003DE:         // The 16:32 pointer found at BFFC9004

003B:000003DE   INT    30 ; #0028:C02301E4  VMM(0Dh)+11E4
```

這個被 *VxDCall* 使用的 INT 30h 用來把控制權交給 ring0，並跳到 VMM.VXD 中的某處。讓我們看看該處的 VMM.VXD 虛擬碼：

```
//———————————————————————————————————————————————————————-
// 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 中的這個 *VxDCall* 處理常式相當複雜。然而如果你研讀夠久，你會發現這些碼被分解爲一些較小的工作：

1. 從 EBP 所指向的 CRS 結構中讀入重要的暫存器值，包括 ring3 EAX （內含 service ID）和 ring3 ESP（指向 ring3 堆疊，那裡面放有參數）。
2. 修改 CRS 中的 CS 和 EIP 暫存器值，使得當 ring0 碼回返，控制權能夠交給 KERNEL32 中的 *VxDCall* 下一行繼續執行。ring3 的 ESP 暫存器也會被改變，如此才能有效地吐出原先被壓入堆疊中的參數。
3. 取得代表 Win32 VxD service ID 的那個 DWORD，並切割爲兩部份（16 位元的 VxD ID 和 16 位元的 service ID）。然後檢查 VxD ID 是否爲標準的

system VxDs，並檢查這被指名的 VxD 是否真正提供了 Win32 VxD services。如果是，繼續檢查 16 位元的 service ID 是否在 VxD 提供的 service 範圍內。

4. 把 ring3 堆疊的內容拷貝到 ring0 堆疊去。
5. 搜尋被指定之 Win32 VxD service 的進入點，並 JMP 過去。由於該 service 可能需要使用 VM handle 或 CRS，所以這裡必須先將這兩個值壓入堆疊之中。

當 Win32 VxD service 函式完成並回返，控制權回到 VMM.VXD。VMM.VXD 負責把 CPU 切回 ring3，並使用 CRS 中記錄的暫存器值。

## 我可以在哪裡找到 Win32 的 VxD Services

稍早我就說過了，微軟沒有正式公開 Win32 VxD services。所以 DDK 並沒有告訴我們哪些 VxDs 提供 Win32 VxD Services。根據我使用 SoftIce/W 的觀察結果，我確定下面這些 VxDs 開放了 Win32 VxD services（可能還有其他未列入）：

| VxD | ID | Services | Description |
|---|---|---|---|
| VMM | 0001h | 41 | Virtual Machine Manager |
| REBOOT | 0009h | 2 | Reboot device |
| VNETBIOS | 0014H | 2 | Virtual NetBios device |
| VWIN32 | 002Ah | 79 | Virtual Win32 "device" |
| VCOMM | 002Bh | 27 | Virtual COMM device |
| VCOND | 0038h | 53 | Virtual Console device |

KERNEL32.DLL 大量使用了 VWIN32、VMM、VCOND、VCOMM。某些 KERNEL32 函式事實上不過就是一個 Win32 VxD service 的外包裝而已。在 ADVAPI32.DLL 之中這種情況更多。VMM.VXD 所提供的 Win32 VxD services 之中的 registry 函式，與 Win32 API registry 函式是並行的。事實上後者正是前者的一層薄薄包裝。

# VMM 提供的 Win32 VxD Services

雖然本章焦點放在 Win32 VxD services 和 VWIN32.VXD 身上，然而如果我沒有至少列出一張 VMM.VXD 的 VxD services IDs 列表的話，我想我就實在是太不夠細心了：

```
00010000h  PageReserve              00010014h  RegDeleteKey
00010001h  PageCommit               00010015h  RegSetValue
00010002h  PageDecommit             00010016h  RegDeleteValue
00010003h  PagerRegister            00010017h  RegQueryValue
00010004h  PagerQuery               00010018h  RegEnumKey
00010005h  HeapAlloc                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
```

我之所以沒有告訴各位每一個函式的參數，唯一的原因是我自己也不知道。對於本書而言，知道 VMM Win32 VxD services 名稱足堪夠矣。毫無疑問，隨著時間過去，這些 services 的參數會慢慢爲我們所知。至於與 registry 有關的 services 的參數，可以經由對應的 Win32 函式的參數推想而知。

許多時候，一個 Win32 VxD service 其實是對應到一個 ring0 VxD service，詳載於 VMM.VXD 的說明文件之中。

VMM.VXD 提供的 Win32 VxD services 可以被區分爲下列數大類：

| 類別 | Win32 VxD services |
|---|---|
| 以分頁爲基礎的記憶體管理 | _GetDemandPageInfo, _PageAttach, _PageCommit _PageDecommit, _PageFlush, _PageFree, _PageModifyPermissions, _PageQuery, _PageReserve |
| 虛擬記憶體之分頁支援 | _PageChangePager, _PagerRegister, _PagerQuery |
| Ring0 的堆積（heap）管理 | _HeapAllocate, _HeapFree, _HeapReAllocate |
| Memory Context 管理 | _ContextCreate, _ContextDestroy, _ContextSwitch, _GetCurrentContext |
| Registry 函式 | _RegCloseKey, _RegCreateKey, _RegDeleteKey, _RegDeleteValue, _RegEnumKey, _RegEnumValue, _RegFlushKey, _RegLoadKey, _RegOpenKey, _RegQueryMultipleValues, _RegQueryValue, _RegQueryValueEx, _RegRemapPreDefKey, _RegReplaceKey, _RegSaveKey, _RegSetValue, _RegSetValueEx, _RegUnloadKey |
| 同步化控制 | _BlockOnID, _SignalID |

一如你在第 3 章和第 5 章所見，KERNEL32 的的確確使用了「分頁記憶體管理」和「context 管理」等等 services。另一些未在本書描述的 KERNEL32.DLL 功能則使用其他類型的 services。Registry services 是一個例外，它們被 ADVAPI32.DLL 喚起（經由 KERNEL32.DLL 中的 *VxDCall* 函式）。

## 自己呼叫 Win32 VxD Services

就我所知，唯一呼叫 Win32 VxD services 的微軟程式碼就是 Windows 95 的 system DLLs 了。然而，沒有理由說一般程式就不能夠呼叫 Win32 VxD services。爲了證明這一點，我寫下**圖 6-1** 的 WIN95MEM 程式（完整程式碼放於書附磁片之中）。

```
//=================================
// WIN95MEM - Matt Pietrek 1995
// FILE: WIN95MEM.C
//=================================
#include <windows.h>
#include "win95mem.h"

// The DemandInfoStruc struct below is excerpted from the VMM.H file
// in the Windows 95 DDK

struct DemandInfoStruc {
   ULONG DILin_Total_Count;    /* # pages in linear address space */
   ULONG DIPhys_Count;        /* Count of phys pages */
   ULONG DIFree_Count;        /* Count of free phys pages */
   ULONG DIUnlock_Count;      /* Count of unlocked Phys Pages */
   ULONG DILinear_Base_Addr;   /* Base of pageable address space */
   ULONG DILin_Total_Free;     /* Total Count of free linear pages */

   /*
    *  The following 5 fields are all running totals, kept from the time
    *  the system was started
    */
   ULONG DIPage_Faults;       /* total page faults */
   ULONG DIPage_Ins;          /* calls to pagers to page in a page */
   ULONG DIPage_Outs;         /* calls to pagers to page out a page*/
   ULONG DIPage_Discards;     /* pages discarded w/o calling pager */
   ULONG DIInstance_Faults;    /* instance page faults */

   ULONG DIPagingFileMax;     /* maximum # of pages that could be in paging file */
   ULONG DIPagingFileInUse;    /* # of pages of paging file currently in use */

   ULONG DICommit_Count;      /* Total committed memory, in pages */

   ULONG DIReserved[2];       /* Reserved for expansion */
};

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;
}

int APIENTRY WinMain( HANDLE hInstance, HANDLE hPrevInstance,
                      LPSTR lpszCmdLine, int nCmdShow )
{
    DialogBox(hInstance, "WIN95MEM_DLG", 0, (DLGPROC)Win95MemDlgProc );
    return 00;
}
```

**圖 6-1　Win95Mem 程式**

WIN95MEM 使用了 VMM.VXD 的 *_GetDemandPageInfo*。這個 service 只是一個同名的 ring0 VxD service 的外包裝。我曾在第5章說過，Win32 的 *GlobalMemoryStatus* 函式使用這個 Win32 VxD service 並簡單傳回一塊空間，放置一些由 *_GetDemandPageInfo* 傳回的資訊。好，如果我們能直接拿第一手資料，何必要二手貨？

*_GetDemandPageInfo* 需要一個指標參數，指向 DemandInfoStruc 結構。這個 service 負責填寫該結構中的欄位，包括：

```
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
```

我不打算詳述每一個欄位。如果你有興趣，請看 Windows 95 DDK 中的相關資料。在 WIN95MEM 之中我只在意兩個欄位：DICommit_Count 和 DIPhys_Count。DICommit_Count 代表已經被 VMM 記憶體管理器配置（或委派，committed）的頁數。請注意，一個 committed page 並不一樣要映射到一塊 RAM，它的意義好像是保留一塊空間以備將來使用。DIPhys_Count 則代表被虛擬記憶體管理器控制的所有 RAM pages。這些 RAM 也就是當 Windows 95 保護模式啓動後的所有可用記憶體。其中並不包括任何常駐程式（TSRs），或任何「被驅動程式在 DOS 載入時期透過 DPMI 配置獲得」的記憶體。

由於 Windows 95 支援虛擬記憶體，通常，被委派（committed）的記憶體會超過實際的記憶體。WIN95MEM 程式顯示出兩者的數量。這兩個值以及它們的比例，每秒修改一次。**圖 6-2** 顯示 WIN95MEM 程式的執行畫面。是的，畫面不怎麼特別，但重點不在 UI 畫面，不是嗎？

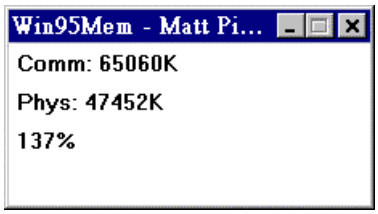



**圖 6-2　WIN95MEM 執行畫面，它示範如何在一般程式中呼叫 VxDCall。**

WIN95MEM 的重點在於對 *VxDCall* 的呼叫。*VxDCall* 是一個未公開函式，所以 WIN95MEM 必須先對 *VxDCall2* 做一次原型宣告。2 意味著兩個參數。由於實際上有三個參數（當你加上 Win32 VxD service ID DWORD 0x0001001E），所以編譯器會產生一個 *VxDCall2@12* 呼叫指令，@12 是因爲此函式宣告爲 stdcall [WINAPI] 函式。我所產生的 K32LIB.LIB 能夠讓我呼叫未公開的 KERNEL32 函式 *VxDCall2@12*，聯結器因此才能夠把一切事情料理妥當。附錄A有對 K32LIB.LIB 的介紹。

以下是實際的 *VxDCall* 呼叫：

```
VxDCall2(0x0001001E, (DWORD)&dis, 0);
```

第一個參數是 DWORD service ID，結合了 VMM device ID 和 _GetDemandPageInfo 函式 ID。第二個參數指向一個 DemandInfoStruc，此結構被宣告在堆疊之中（因爲是區域變數）。最後一個欄位的意義未明，我之所以傳 0 是因爲當 *GlobalMemoryStatus* 呼叫這個 service 時也是傳 0。

# 檢視 VWIN32.VXD

我們已經旋風般地完成了對 VxD 介面的旅行，現在我要把焦點縮小，討論 VWIN32.VXD。這個 VxD 是新的（所謂新是指對 Windows 3.1 而言），其 16 位元 VxD ID 是 0x002A。爲什麼要分割出 VWIN32 和 VMM 兩個 VXDs，原因不是十分明顯（至少在微軟的「神聖殿堂」之外不是十分明顯）。一般對 VWIN32.VXD 的認知是，它內

含的 Win32 VxD services 影響行程和執行緒的排程及同步化。我把 VMM.VXD 和 VWIN32.VXD 當做是一個團隊，做一些 Windows 賴以生存的 ring0 動作。

VWIN32.VXD 並不開放出 V86 API，但它開放有 ring0 VxD services、16 位元保護模式 services、以及 Win32 VxD services。在所有 VxDs 之中，VWIN32 擁有最多數量的 Win32 VxD services。如果不是 VWIN32.VXD 提供了這麼多的 services，如果不是這些 services 如此重要，我也就不會自討苦吃寫下前數節內容了。現在就讓我們跳進去看看 VWIN32.VXD 的核心吧。

## VWIN32.VXD 的 ring0 VxD service API

我們首先要看的是 VWIN32.VXD 提供給其他 VxDs 的介面。很幸運，微軟在 Windows 95 DDK 的 VWIN32.H 和 VWIN32.INC 中留下了一份列表。爲了造福那些沒有 DDK 的讀者，我在**圖 6-3** 也提供了一份列表。SoftIce/W 的 VxD 命令知道這些 services，所以它很容易就能夠取得一份 service 名單，及其函式 ID。

在這份列表中，左邊欄位是 VWIN32.VXD 的 service ID，中間欄位是其函式位址，右邊欄位是 service 名稱，一如 VWIN32.INC 中所記錄的。

Ring0 VWIN32 services 列表看起來十分混雜。然而你可以看出一堆與執行緒同步化有關的函式。另有一些函式（如 *VWIN32_GetCurrentProcessHandle*、*VWIN32_GetCurrentDirectory*、*VWIN32_TerminateApp*）暗示我們，VWIN32.VXD 擁有某種程度的行程（process）知識。這些函式之所以饒富趣味，因爲 VMM.VXD 只知道所謂的執行緒，所以它沒有辦法提供與行程管理有關的函式。這一部份必須由 VWIN32.VXD 提供。

```
: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, LOCAL
  0001  C0075776  VWIN32_DIOCCompletionRoutine
  0002  C007678F  _VWIN32_QueueUserApc
  0003  C0261002  _VWIN32_Get_Thread_Context
  0004            _VWIN32_Set_Thread_Context
  0005            _VWIN32_CopyMem, LOCAL
  0006            _VWIN32_Npx_Exception
  0007            _VWIN32_Emulate_Npx
  0008            _VWIN32_CheckDelayedNpxTrap
  0009            VWIN32_EnterCrstR0
  000A            VWIN32_LeaveCrstR0
  000B            _VWIN32_FaultPopup
  000C            VWIN32_GetContextHandle
  000D            VWIN32_GetCurrentProcessHandle
  000E            _VWIN32_SetWin32Event
  000F            _VWIN32_PulseWin32Event
  0010            _VWIN32_ResetWin32Event
  0011            _VWIN32_WaitSingleObject
  0012            _VWIN32_WaitMultipleObjects
  0013            _VWIN32_CreateRing0Thread)
  0014            _VWIN32_CloseVxDHandle
  0015            VWIN32_ActiveTimeBiasSet
  0016            VWIN32_GetCurrentDirectory
  0017            VWIN32_BlueScreenPopup
  0018            VWIN32_TerminateApp
  0019            _VWIN32_QueueKernelAPC
  001A            VWIN32_SysErrorBox
  001B            _VWIN32_IsClientWin32
  001C            VWIN32_IFSRIPWhenLev2Taken
```

**圖 6-3　VWIN32 Ring0 VxD services**

## VWIN32.VXD 的 16 位元保護模式 API

微軟並沒有在 DDK 中公開 VWIN32.VXD 的 16 位元保護模式 API 函式。然而我早在 1993 年八月份的 ***Microsoft Systems Journal*** 中就以一篇名爲 "Stepping Up to 32 Bits: Chicago's Process, Thread, and Memory Management." 的文章揭露了那些函式。寫那篇文章的時候，那些函式尚還未被從 VWIN32.INC 檔案中取走。這裡我把它們再次列出來。

```
VWIN32_GET_VER                          AH = 0
VWIN32_THREAD_SWITCH                    AH = 1
VWIN32_DPMI_FAULT                       AH = 2
VWIN32_MMGR_FUNCTIONS                   AH = 3
   子功能：
   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
```

誰將呼叫這些函式呢？就是 KRNL386 自己囉。這些函式正是 Windows 95 的 ring3 16 位元碼能夠與 ring0 VWIN32 元件接觸的關鍵。這些函式大部份都落在一個大範籌之中：執行緒排程、執行緒同步化、記憶體管理、錯誤處理（fault handling）。

前面所列的函式之中，VWIN32_MAKE_IDLE_SYS 特別引起我的注意。這個函式是當 KRNL386.EXE 的 ring3 Win16 排程器發現沒有任何 task 需要排程時，被呼叫的（如果你有我的 ***Windows Internals*** 一書，請看其中的 *Reschedule* 函式）。當 16 位元 KRNL386 排程器掉進所謂的閒置迴圈（idle loop）之中，KRNL386 就會呼叫 VWIN32_MAKE_IDLE_SYS。控制權不會交還給 KRNL386，直到 16 位元程式有某些動作發生。

附帶一提，VWIN32_EXIT_TIME 屬於錯誤處理（fault handling）一類。如果你曾讀過 ***Undocumented Windows***，或許你還記得一個名爲 *Bunny_351* 的函式。在 Windows 3.1 和 Windows 95 之中，*Bunny_351* 在系統結束時（關機前）被呼叫。它的唯一目的就是改變預設的 unhandled exception 的處理常式的位址。在 Windows 95 之中，*Bunny_351* 其實是 VWIN32_EXIT_TIME 的一個外包裝而已。

## VWIN32.VXD 的 32 位元 VxD service API

我們很幸運，微軟公開了前兩組 APIs 的 service 名稱和 ID 號碼。但是在 Win32 VxD service 介面中我們就沒有這麼幸運了。我花了一些時間，建立下面這個表格，列出一些 services 的進入點。它並不完全，有些函式名稱甚至是猜的 -- 根據我所觀察的 KERNEL32 和 VWIN32 碼而猜。

| Service ID | 目的 |
|---|---|
| 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     ??? (unknown)
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     ??? (unknown)
0x002A002E     dwBlockSingleWnod (WaitForSingleObject)
0x002A002F     dwBlockMultipleWnod (WaitForMultipleObjects)
0x002A0030     VWIN32_SetEvent
0x002A0031     Something to do with delivering APCs
0x002A0032     ??? (unknown)
0x002A0033     InitUserAPCList
0x002A0034     ??? (unknown)
0x002A0035     Calls VMM Signal_Semaphore_No_Switch
0x002A0036     Calls System_Control(KERNEL32_INITIALIZED)
0x002A0037     VWIN32_CommonFaultPopup
0x002A0038     VWIN32_ForceCrsts
0x002A0039     ??? (unknown)
0x002A003A     VWIN32_FreezeAllThreads
0x002A003B     VWIN32_UnFreezeAllThreads
0x002A003C     Calls IFSMgr_Ring0_FileIO
0x002A003D     Calls Get_Initial_Thread_Handle, Attach_Thread_To_Group, and
               Boost_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      ??? (unknown)
0x002A004A      VW32_AddSysCrst
0x002A004B      VW32_SetTimeOut
0x002A004C      VW32_Cancel_Time_Out
0x002A004D      ??? (unknown)
0x002A004E      Something to do with setting and reflecting hotkeys
```

注意：Crst 意思是 Critical Section
APC 意思是 Asynchronous Procedure Call
VMCPD 是 Virtual Math Coprocessor Device
IFSMgr 是 Installable File System Manager
System_Control 是 VMM.VXD ring0 service，可以把系統控制訊息廣播給 VxDs。

VWIN32 提供了許多 Win32 VxD services，多到我簡直可以爲它們另外寫一本書。本書之中，我把焦點放在其中三個 services 身上：

```
0x002A0010 - VWIN32_Int21Dispatch（DOS 中斷）
0x002A0029 - VWIN32_Int31Dispatch（DPMI 中斷）
0x002A002A - VWIN32_Int41Dispatch（除錯器的 notification 中斷）
```

微軟已經宣佈，Win32 不能夠觸發如 Win16 碼中所做的那種中斷。然而，這並不意味不再需要使用中斷（interrupt）了。當 Win32 碼必須觸發 INT 21h、31h、41h 時，VWIN32.VXD 中的這些 Win32 VxD services 就可以用來做正確而適當的動作。KERNEL32.DLL 在各處使用這些 interrupt dispatching functions。讓我們觀察 *VWIN32_Int31Dispatch* 的函式內容，以便知道 VWIN32.VXD 是怎麼工作的：

**VWIN32_Int31Dispatch 的虛擬碼**

```
// 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_ECX = ring3_ECX

Exec_PM_Init( EAX = 0x31 );

if (carry set)
    _Debug_Out_Service( "VW32_Int31Dispatch: Exec_PM_Int Failed!\r\n" );
```

VWIN32 之中並沒有太多的 interrupt dispatching services。用來派送中斷指令的 Win32 VxD services 只不過是 ring0 Exec_PM_Init service 的一個外包裝而已。許多圍繞在 Windows 95 週圍的宣傳花招都說 DOS 已經不見了。唔，由於幾乎所有用來呼叫 DOS 的碼現在都放在 VxD 之中，如果 DOS 真的消失了，interrupt dispatching Win32 VxD services 就不應該被使用得那麼頻繁，不是嗎？讓我們看看 KERNEL32.DLL 中的 *FindClose* 函式，然後你可以自己決定孰是孰非。

**FindClose 虛擬函式碼**

```
// 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 虛擬函式碼**

```
return VxDCall( 0x002A0010, EAX, ECX);
```

我要告訴各位，KERNEL32 呼叫了數以打計的 *VWIN32_Int21Dispatch* service。以下列表揭露 KERNEL32.DLL 呼叫的 INT 21h（DOS services）：

| DOS Subfunction | Purpose |
|---|---|
| 0E00 | Set default drive |
| 1900 | Get current drive |
| 2A00 | Get system date |
| 2B00 | Set system date |
| 2C00 | Get system time |
| 2D00 | Set system time |
| 3600 | Get disk free space |
| 3D00 | Open existing file — read only |
| 3D02 | Open existing file — read/write |
| 3E00 | Close file |
| 3F00 | Read file |
| 4000 | Write file |
| 4200 | Set current file position — relative to start of file |
| 4201 | Set current file position — relative to current position |
| 4202 | Set current file position — relative to end of file |
| 4400 | IOCTL — get device information |
| 4401 | IOCTL — set device information |
| 4408 | IOCTL — check if block device removable |
| 4409 | IOCTL — check if block device remote |
| 440D | IOCTL — generic block device request |
| 4B00 | Exec program |
| 4D00 | Get return code |
| 5000 | Set current PSP |
| 5700 | Get file date/time |
| 5701 | Set file date/time |
| 5704 | Set extended file attributes |
| 5705 | ??? (unknown) |
| 5706 | ??? (unknown) |
| 5707 | ??? (unknown) |
| 5900 | Get extended error info |
| 5C00 | Lock file region |
| 5C01 | Unlock file region |
| 5E00 | Network functions |
| 5F32 | ??? (unknown) |
| 5F33 | ??? (unknown) |
| 5F34 | ??? (unknown) |
| 5F35 | ??? (unknown) |
| 5F36 | ??? (unknown) |
| 5F37 | ??? (unknown) |
| 5F38 | ??? (unknown) |

```
5F3B                ??? (unknown)
5F3C                ??? (unknown)
5F4D                ??? (unknown)
5F4F                ??? (unknown)
5F52                ??? (unknown)
6800                Commit file
7139                LFN create directory
713A           LFN remove directory
713B                LFN change directory
7141                LFN delete file
7143           LFN get/set file attributes
7147                LFN get current directory
714E                LFN find first file
714F                LFN find next file
7156                LFN rename file
7160                LFN get canonical filename
716C                LFN Extended open/create
71A0                LFN Get Volume Information
71A1                LFN Find Close
71A3                ??? (unknown)
71A4                ??? (unknown)
71A5                ??? (unknown)
71A6                LFN Get File Info By Handle
71A7                LFN File Time To DOS Time
B400                ??? (unknown)
EA00                ??? (unknown)
```

哇嗚！KERNEL32.DLL 發出好多的 INT 21h。看來 DOS 的陰魂繼續纏繞著我們，並沒有因爲 Windows 95 把 DOS 幹掉而結束。唯一的改變就是，現在的 INT 21h 是被 KERNEL32.DLL 呼叫，而不是被你的程式直接呼叫。

爲什麼微軟要這麼做呢？難道它不能夠直接呼叫低階的作業系統函式，避過有 15 年歷史的 INT 21h 介面嗎？答案當然是可以，然而如果微軟這麼做，那些和 INT 21h calls 掛勾的驅動程式以及 VxDs 就會出問題。這些驅動程式和 VxDs 沒有辦法看到基本的作業系統行爲，而那是它們被期望能夠看到的。再一次，微軟爲了與過去的應用程式和驅動程式相容而做了一些妥協。

現在回到 VWIN32.VXD dispatching services 的探討。你或許已經熟悉 INT 21h（DOS）和 INT 31h（DPMI），然而 INT 41h 是什麼？那是作業系統用來告訴系統級除錯器（如 WDEB386、SoftIce/W）某些重要訊息的管道。例如，KERNEL32.DLL 引發下面這些 INT 41h（詳列於 DDK 中的 DEBUGSYS.INC 中）：

```
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 breakpointt
```

# VWIN32 TDBX

我曾經在第 3 章討論過 KERNEL32 維護的 process database 和 thread database。VWIN32 是如此私密地牽扯到行程和執行緒，我想如果 VWIN32 有它自己的資料結構用以追蹤行程和執行緒，我們一點也不會覺得驚訝。是的，這個資料結構稱爲 TDBX。

系統中的每一個執行緒有一個 TDBX。毫無疑問，某些 VxD 高手現在就會猜想了：指向 TDBX 的指標應該是放在 THCB（Thread Control Block）的一個 TLS（Thread Local Storage）slot 之中吧。THCB 是 VMM 的執行緒管理器用來追蹤其所產生的所有執行緒的一種資料結構。別的 VxDs 可以索求 THCB 中的 slots 做爲它們自己的 per-thread storage。這個索求動作是經由 VMM 的 *_AllocateThreadDataSlot* 完成，傳回的是一個 THCB 的偏移位置。在那個地方，有一個指標指向「執行緒專有資料」可以存放的空間。KERNEL32 呼叫 Win32 VxD Service 0x002A0008（VWIN32_CreateThread），配置一個 VWIN32 TDBX。TDBX 指標放置在 THCB 的 DWORD slot 之中。

現在讓我們看看 VWIN32 的 TDBX 結構吧。和本書許多結構不同的是，TDBX 的許多欄位只能夠根據其名稱做一點猜想，沒有確切的答案。

**00h DWORD ptdb**

一個指標，指向與此執行緒有關的 ring3 PROCESS_DATABASE 結構。後者的格式在第３章的「Windows 95 Process Database（PDB）」一節中有所描述。

**04h DWORD ppdb**

一個指標，指向與此執行緒有關的 ring3 THREAD_DATABASE 結構。後者的格式在第３章的「Thread Database」一節中有所描述。

**08h DWORD ContextHandle**

一個指標，指向執行緒所屬的行程的 memory context 結構。後者在第５章描述過。

**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**

程式中針對此一執行緒，呼叫 *SuspendThread* 函式的次數。

### 24h　DWORD　SuspendHandle

未知。

### 28h　DWORD　MustCompleteCount

當此值爲 0，此執行緒不能夠被中斷。*EnterMustComplete* 和 *LeaveMustComplete* 兩函式（在第 3 章和第五章描述過）分別增減此欄位的值。

### 2Ch　DWORD　WaitExFlags

此一執行緒的旗標值。下面是已知的一些數值及其意義：

```
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      FORZEN
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 的意思是 Asynchronous Procedure Call）

**3Ch DWORD KernAPCList**

（APC 的意思是 Asynchronous Procedure Call）

**40h DWORD pPMPSPSelector**

一個指標，指向保護模式的 PSP selector。

**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 DWORD TraceOutLastCS**

未知。

**72h DWORD K16TDB**

與此執行緒所屬之行程有關係的 Win16 Task Database（TDB）的 selector。

### 74h　DWORD　K16PDB

與此執行緒所屬之行程有關係的 Win16 Program Segment Prefix（PSP）的 selector。

### 76h　DWORD　DosPDBSeg

與此執行緒所屬之行程有關的真實模式下的 PSP 的節區值（segment value）。

### 78h　DWORD　ExceptionCount

未知。

TDBX 的最前面兩個欄位最是有趣。它們爲 ring0 VWIN32.VXD 提供指標，指向 ring3 KERNEL32.DLL 所使用的行程與執行緒結構體。請你回憶第3章，KERNEL32 爲每一個執行緒在 ring3 THREAD_DATABASE 結構中保留了一個指標，指向 TDBX。把兩者關聯起來，你就會看到 KERNEL32.DLL 的 THREAD_DATABASE 和 VWIN32.VXD 的 TDBX 結構兩者彼此互相參考的情況。

另一些欄位也很值得注意。偏移位置 8 的地方是一個指標，指向執行緒所屬行程的 memory context。此外，一個執行緒如果絕對不能夠被中斷，那麼欄位 MustCompleteCount 對於低階的執行緒同步化動作非常重要。

TDBX 的尾端，是一個指向 Win16 Task Database 的 selector。最後兩個欄位則是保護模式指標和真實模式指標，指向執行緒所屬之行程的 PSP。很明顯，ring0 VWIN32.VXD 需要知道 DOS 的資料結構和 Win16 KRNL386 的資料結構。這一節的重點就是，Windows 95 的三個核心元件（16 位元 KRNL386.EXE、ring3 KERNEL32.DLL、ring0 VWIN32.VXD）彼此都知道對方。接下來我們要討論其間的交互關係。

# Windows 95 的三個核心元件如何聯繫

在帶領你逛過 VWIN32.VXD 的漫漫長路之後，我相信你終於有足夠的基礎，可以接觸 Windows 95 三個核心元件如何彼此聯繫的主題了。這三個核心元件的通訊和互動的範圍令人驚訝，至少對我而言是如此。爲什麼？依照我的看法，我認爲一個作業系統核心是獨立的實體，不能夠依賴本體以外的任何東西。核心元件是每一樣東西建立時的基礎，所以它不應該依賴外界的任何事務。另一方面，我一直喜歡把核心元件想像是一個黑盒子，了解黑盒子的運作不應該需要先了解黑盒子以外的元件。然而，有三個黑盒子其實不夠黑。我打算證明給你看，每一個核心元件對另一核心元件有明明白白的認知以及互動。

## VWIN32 對 KRNL386 的認知

「VWIN32 知悉 KRNL386」的第一個徵候就是在 TDBX 結構的尾端有一個 Win16 Task Database（TDB）選擇器，與 TDBX 所屬的行程有關聯。更多的徵候則是出現在 VWIN32.VXD 中一個我稱之爲 *ThreadSwitchCallback* 的函式。

當執行緒排程器切換一個新的執行緒時，VMM 就會呼叫 *ThreadSwitchCallback*。這個函式是 Win32 強制性多緒多工和 Win16 KRNL386.EXE 非強制性多工的交會點。它也是多工 Win32 連接單工 MS-DOS 的地方。*ThreadSwitchCallback* 扮演一個令人矚目的角色，使 Windows 95 成爲一個（看起來）完全的多工系統。

**ThreadSwitchCallback 虛擬碼**

```
// 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 );
}
```

在完成一些穩健性初步檢查之後，*ThreadSwitchCallback* 更改了兩個全域變數，那是 VWIN32 用來指向 current ring3 process database 和 thread database 的指標。同時，它也

更改了 VWIN32 用來指向 current TDBX 結構的指標（也是一個全域變數）。接下來這個函式做了一些讓我初次見到時（在 SoftIce/W 之中）目瞪口呆的事情：ring0 VWIN32.VXD 竟然粉碎了 KRNL386.EXE 中的全域變數 CurTDB。

在 Windows 95 來到這個世界之前，CurTDB 是神聖不可侵犯的。它唯一可能被更改的時機是在 Windows 呼叫 Windows 3.1 的核心排程器（也就是 *Reschedule* 函式）時。這時候原本井然有序的 Windows 3.1 合作型多工受到 Windows 95 的強制性多工的衝擊，而強制性多工贏了。這是一個有問題的架構，一個基礎物質（例如 current task 的全域變數）竟然會受到另一個極少數程式員知道的元件的影響（當然啦，你已經知道了 VWIN32，因爲你正在讀這一章嘛！但是我的觀點還是有確實根據的）。

*ThreadSwitchCallback* 之中剩下的雜務就是清除與多工無關的東西。即將退出之執行緒的 PSP 被記錄在其 TDBX 結構中。接下來函式碼取得即將進入的執行緒的 PSP，並用它來設定 system VM 中的 current PSP。這樣的 PSP 切換並不是始自 Windows 95，Windows 3.1 也做類似的事情，只不過是在 ring3 KRNL386 之中爲之。*ThreadSwitchCallback* 所做的最後一件事情就是把 memory context 切換爲即將進入之執行緒的 memory context。就像第 5 章所說，memory context 的切換可使得每一個行程擁有自己的私有位址空間。在 Windows 95 之中，行程的私有空間是在線性位址 4MB~2GB 之間。

## VWIN32 對 KERNEL32 的認知

「VWIN32 知悉 KERNEL32」的最主要徵候在於 VWIN32 所使用的 TDBX 結構中出現 THREAD_DATABASE 和 PROCESS_DATABASE 的指標。VWIN32 也保有一些全域變數，用來指向 KERNEL32 所維護的 current process database 和 thread database 結構。此外，VWIN32.VXD 還獲得一系列指標，指向 KERNEL32.DLL 啓動時的一些函式。KERNEL32.DLL 自動提供這些函式位址，做爲 Win32 VxD service 0x002A0003 的參數。

## KERNEL32 對 VWIN32 的認知

顯然，「KERNEL32 知悉 VWIN32」的最大證據就在於 KERNEL32 呼叫了 VWIN32 所提供的眾多 Win32 VxD services。這個事實已經在本書的許多地方展示過了，特別是在本章的前數節。

此外，兩造之間的合作關係還發生在當某個 Win32 VxD service 被呼叫時，VWIN32.VXD 送交一系列的函式位址給 KERNEL32。我所說的某個 Win32 VxD service 是指 0x002A0001，它被 KERNEL32 的 *FInitPager* 函式呼叫。我們推測 KERNEL32.DLL 將在 page fault 發生時回呼（call back）這些 VWIN32.VXD 位址。就像 ***Unauthorized Windows 95*** 書中所說，VMM 的 page fault 處理常式在 ring0 呼叫 KERNEL32.DLL。我們必須相信，VWIN32 會把它的函式位址交給 KERNEL32.DLL -- 在 KERNEL32.DLL 進行分頁初始化動作的時候。

## KERNEL32 對 KRNL386 的認知（微軟的來源認此點）

翻遍微軟的 Windows 95 技術資料，從來不會看到 KERNEL32 對 KRNL386 有任何機能上的依賴（但對於 USER 和 GDI 兩個模組，微軟承認 32 位元碼下移呼叫了 16 位元碼）。***Unauthorized Windows 95*** 完全撕碎了微軟的那些宣言。然而，到底哪些 KERNEL32 函式呼叫了哪些 KRNL386 函式，至今未明。

我不打算重複 ***Unauthorized Windows 95*** 書中對於 KERNEL32 呼叫 KRNL386 的絕佳解釋。我要提供一些新東西：一些被 KERNEL32 呼叫的 KRNL386 函式。如果這還不夠有趣，我真不知道什麼才叫有趣了。這份清單對於微軟宣稱「呼叫任何 KERNEL32 函式將不會導至呼叫端癡癡等待 Win16Mutex」特別適切。你會發現有一組不算少的 KERNEL32 函式會堵住 Win16Mutex。

爲恐你認爲這些函式不可能引起問題，請你想想「profile 相關函式」，例如 *GetPrivateProfileSection*。它們將存取你的硬碟並且快速回返。如果它們所要尋找的檔案

是在光碟上，怎麼辦？光碟機的速度比較慢，於是 Win16Mutex 會被呼叫端擁有一段不短的時間（我是在 Windows 95 beta 版上實測這個情況）。

下表中，左邊欄位是 KERNEL32 內部使用的函式名稱。如果某個特殊的 KERNEL32 內部位元被設立，KERNEL32 就會在除錯終端機上吐露這些字串。有些函式是 KERNEL32 的一般輸出函式，另一些則是未公開的，或 KERNEL32 輸出函式的變種。一般應用程式是否會呼叫它們，頗有疑問。例如，*LoadLibrary16* 呼叫 16 位元的 KRNL386 *LoadLibrary* 函式。*LoadLibrary16* 是 KERNEL32 的輸出函式，輸出序號爲 35，它和 KERNEL32 的 *LoadLibrary* 並不相同。*WritePrivateProfileSection32A* 也和一般的 KERNEL32 *WritePrivateProfileSectionA* 不相同。

下表右邊欄位的函式名稱，是 KERNEL32 直接下移（thunk down）時所呼叫的 KRNL386 函式。如果欄位是空的，表示下移（thunk down）時並不直接呼叫輸出函式，而是呼叫 KRNL386 的內部函式。

| KERNEL32.DLL 的內部函式名稱 | KRNL386.EXE 的輸出函式名稱 |
|---|---|
| TerminateZombie | |
| DiagOutput16 | DiagOutput |
| DispatchRITInput | |
| GetFastQueue | KERNEL.625 |
| SetVolumeLabel16 | |
| PK16FNF | |
| CommConfigThk | |
| InitAtomTable | |
| GetAtomNameA | |
| 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
KERNEL32.DLL                        KRNL386.EXE
Internal Name  Exported Name
WOWGlobalFree16                     GlobalFree
WOWGlobalAlloc16
WOWDirectedYield16                  DirectedYield
WOWYield16                          Yield
Yield16
FreeLibrary16ByName
SSChk
UTThunkLSHelper
UTUnregisterInt
UTRegisterInt
UTProcessExit
FreeCB
```

我已經指出，許多 Windows 95 程式員渴望知道哪一個 KERNEL32 函式會堵住 Win16Mutex -- 在該函式周遊 KRNL386 期間。微軟宣稱沒有任何一個函式會堵住 Win16Mutex，這是天大的謊言。下面這個表格將這些會堵住 Win16Mutex 的函式整理出幾個類別。

| 函式類別 | 函式名稱 |
|---|---|
| Atom functions | AddAtomA, DeleteAtom, FindAtomA, GetAtomNameA, GlobalAddAtomA, GlobalDeleteAtom, GlobalFindAtomA, GlobalGetAtomNameA, InitAtomTable |
| Directory functions | GetSystemDirectoryA, GetWindowsDirectoryA |
| Selected WIN.INI file functions | GetProfileSectionA, GetProfileStringA, WriteProfileSectionA, WriteProfileStringA, |

KERNEL32 知道 KRNL386 中的某些全域變數。稍早我們已經看到 VWIN32.VXD 如何影響 KRNL386 中的全域變數 CurTDB。在 KERNEL32 這邊，最令人側目的例子是它使用了 KRNL386 的 Win16Mutex 全域變數。Win16Mutex 事實上是一個 CRITICAL_SECTION 結構，位於 KRNL386.EXE 的 DGROUP 節區中。KERNEL32 如

何能夠獲得 Win16Mutex 的位址呢？原來 KRNL386 在載入 KERNEL32 之後，引發了一個 initialization call，這時候它把 Win16Mutex 一起交過去。

## KRNL386 對 KERNEL32 的認知

剛剛我們才看到了 KERNEL32 之中所有呼叫 KRNL386 的函式列表。同樣地 KRNL386 也有一張相對列表，其中的函式呼叫了 KERNEL32：

| 函式類別 | 函式名稱 |
|---|---|
| 行程管理函式 | CreateProcessFromWinExec, IFatalAppExit, NukeProcess, RegisterServiceProcess, ThunkExitProcess, ThunkMapProcessHandle, ThunkCreateProcessWin16, WinExecWait |
| 執行緒管理函式 | IsThreadId, ThunkCreateThread16, ThunkTerminateThread, |
| 模組管理函式 | GetModuleHandle32, GetNEPEBuddyFromFileName32, LoadLibraryEx32W, ThunkFreeLibrary32 (free a Win32 DLL), ThunkGetHModK16FromHModK32 (get the Win16 HMODULE from a Win32 HMODULE), ThunkGetModuleFileName, ThunkLoadLibrary32 (load a Win32 DLL), |
| 磁碟目錄管理函式 | GetCurrentDirectory (stored in the KERNEL32 process database for each process), SetCurrentDirectory32, ThunkGetCurrentDirectory |
| 檔案 I/O 函式 | FindClose, FindFirstFile, FindNextFile (16-bit versions of the Win32 FindXXX functions), FileTimeToDosDateTime, OpenFileEx16And32, ThunkCloseDOSHandles, ThunkCloseW32Handle, |
| 32-bit heap 函式 | LocalAlloc32NG, ThunkLocal32Alloc, ThunkLocal32Free, ThunkLocal32Init, ThunkLocal32ReAlloc, ThunkLocal32SizeThkHlp, ThunkLocal32Translate, ThunkLocal32ValidHandle |
| 同步化控制函式 | ThunkCreateW32Event, ThunkResetW32Event, ThunkSetW32Event, WaitForMultipleObjects, WaitForSingleObject |
| 錯誤處理函式 (fault handling） | CreateFaultThread, FaultRestore, FaultSave, |

| | |
|---|---|
| WINOLDAP 支援函式 | WOAAbort, WOACreateConsole, WOADestroyConsole, WOAFullScreen, WOAGimmeTitle, WOASpawnConApp, WOATerminateProcesses |
| 資源清除函式 （cleanup functions） | FreeInitResources32, HGCleanupDepartingHTask, NotifyDetachFromWin16, ThunkDeallocOrphanedCrsts |
| 雜項函式（Misceellaneous functions） | CallProc32WFixHelper, CallProc32WHelper, FlatCommonThunkConnect16, FullLoRes, GetProcessDword, GetVersionEx, InitK32AfterSysDllsLoaded, ISetErrorModeEx, InvalidateNLSCache, LateBindWin32ThunkPartner, SetProcessDword, SmashEnvironment, ThunkConvertToGlobalHandle, ThunkGetProcAddress32, ThunkTheTemplateHandle, VirtualFree |

如果你驚訝上個表格中的名稱是哪來的，我告訴你，它們內嵌於 KERNEL32.DLL 之中。當我研究 KERNEL32.DLL 時，我注意到有個固定模型出現在 KRNL386 上移（thunk up）的那些函式中。這個固定模型包括一個指標，指向函式名稱。寫一個文字編輯器巨集，找出 KERNEL32.DLL 中含有此種固定模型的所有地點，並且把函式名稱拷貝到一個檔案之中，對我而言是很簡單的一件事。

## KRNL386 對 VWIN32 的認知

KRNL386 對 VWIN32 的認知係包含在它對 VWIN32 PM API services 的呼叫之中。我曾經在「VWIN32.VXD 的 16 位元保護模式 API」一節中描述過那些 services。

# Win32 VxD Service Spy（W32SVSPY 程式）

如果沒有一個程式讓你探索我所說的那些東西，這一章就不算完整。我寫了一個 W32SVSPY 程式用來監視 Win32 VxD Service calls。從某方面來說，它很像第 10 章的 API spy 程式，只是不那麼完整而已。在撰寫 W32SVSPY 的過程中有一些技術上的障礙，我是以有趣的、不明顯的方法來解決。因此，我將另外花一點時間告訴你，可以從 W32SVSPY 中學到哪些奇技淫巧。

完整的原始碼放在書附磁片之中。我不打算描述 W32SVSPY 的所有內部動作，因爲它十分複雜，而且或許不是大多數人感興趣的對象。W32SVSPY 的輸出才是大多數人感興趣的東西。

**圖 6-4** 顯示 W32SVSPY 的初始畫面。大視窗準備放置刺探結果。如果要開始觀察 Win32 VxD service calls，請按【Start】鈕。資料將立刻開始展示，直到你按下【Stop】鈕，或是直到 W32SVSPY 的緩衝區滿了爲止。我任意選了 16KB 做爲 W32SVSPY 的緩衝區，你可以改變它，但記得重新編譯。【Save...】按鈕允許你將輸出結果儲存到一個文字檔中。

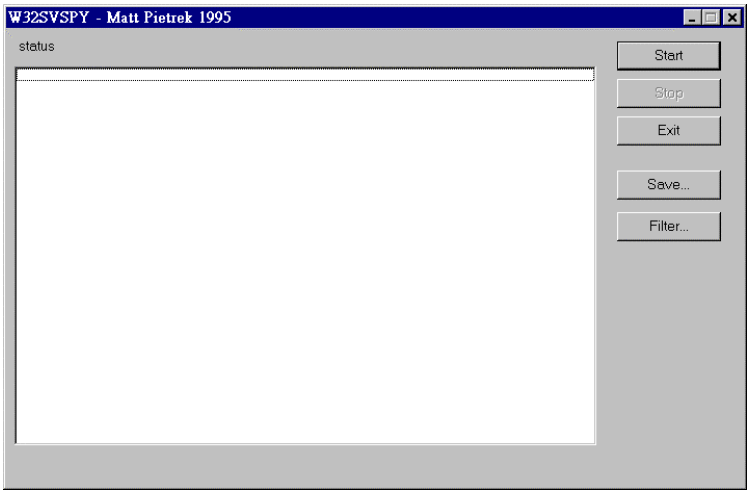



**圖 6-4　W32SVSPY 的初始畫面**

【Filter...】按鈕帶出一個對話盒如**圖 6-5**，讓你過濾任何 Win32 VxD services -- 如你稍後所見，Win32 VxD services 之中可能有一些雜質，把它們濾掉可能有助觀察。Filter 對話盒有兩個列示視窗（listbox），只要在左邊視窗上選擇一個 VxD，右邊視窗就會出現該 VxD 中的 Win32 VxD services。帶有+ 號者，表示該 service 已經致能（enable）。如果雙擊其中某一個 service，你就可以切換致能（enable）狀態和除能（disable）狀態。

預設情況下所有的 services 都是致能的。W32SVSPY 眼中只有 VMM 和 VWIN32。當然它也可以刺探所有的 Win32 VxD service calls，但是只能夠對那些它所知道的 Win32 VxD services 提供名稱。如果你要加上其他 VxDs 的 Win32 VxD services 的知識，請修改 W32SRVDB.C。

你在對話盒中的任何過濾設定都會被保存在 .FLT 檔案中。如果你要關閉過濾功能（也就是說如果你要看每一樣東西），請在啓動 W32SVSPY 之前先殺掉 .FLT 檔，要不就是回到【Filter...】對話盒中重新將所有函式致能起來。

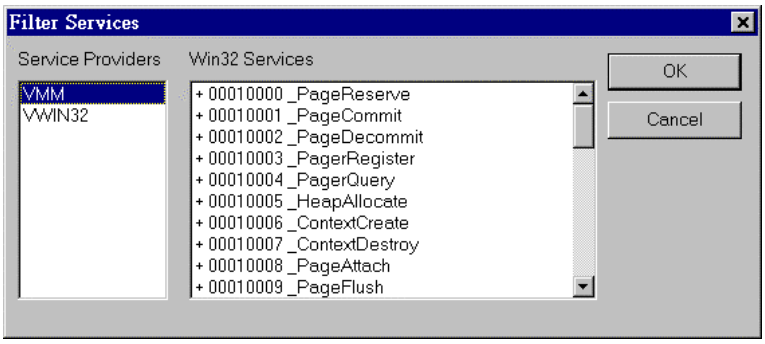



**圖 6-5 W32SVSPY 的 Filter 對話盒。**

W32SVSPY 的輸出格式如下：

```
<CurrentTaskName>  <Service Name> (<parameter 1 value>)
```

例如，下面這一行：

```
Explorer VWIN32_SetEvent(8154A230)
```

表示目前的行程（也就是呼叫該 service 者）是 Explorer，被呼叫的 service 是 VWIN32_SetEvent，第一個參數是 8154A230h。W32SVSPY 並不顯示所有參數，因爲那會使運送資料的程序過於複雜。如果你要這種能力，不妨自行練習。

如果 Win32 VxD service call 是 VWIN32_Int21Dispatch，W32SVSPY 會把第一個參數解碼爲一個字串，用以代表將被喚起的 DOS 函式。這種情況下，DOS 函式名稱將出現在 service 名稱（也就是 VWIN32_Int21Dispatch）之後，第一個參數之前。例如：

```
Calc    VWIN32_Int21Dispatch LFN File Time To DOS Time(8154A230)
```

偶而你也會看到像這樣的輸出：

```
FFFF56F3  VWIN32_Int41Dispatch(00000150)
```

第一個欄位是 process ID。這是因爲 W32SVSPY 沒辦法從 Win16 的 TDB 中萃取出行程名稱（Win16 TDB 是 Windows 95 爲每一個行程所準備的）。這種情況通常只會發生在應用程式啓動的時候，並且在 KERNEL32.DLL 把正確的行程名稱放到 task database 之前。這些 ID 究竟對應哪一個行程，通常不至於太難了解，只要尋遍所有的項目即知。

## 一個 W32SVSPY 輸出範例

爲了展示 W32SVSPY 能夠做哪些事情，讓我們看看當我們在桌面上以捷徑（short cut）啓動一個應用程式時，所引發的 Win32 VxD services。我們首先製作出 CALC.EXE 的捷徑，然後啓動 W32SVSPY 並按下【Start】鈕，然後雙擊 CALC.EXE 的捷徑圖示，最後再按下 W32SVSPY 的【Stop】鈕。執行結果將出現在 W32SVSPY 的主視窗中，如圖 **6-6**。此刻你可以好好瀏覽其內容，也可以將它存檔。

我取得了一份輸出結果，並去除其中一些雜訊以及重複資料，列於圖 **6-7**。

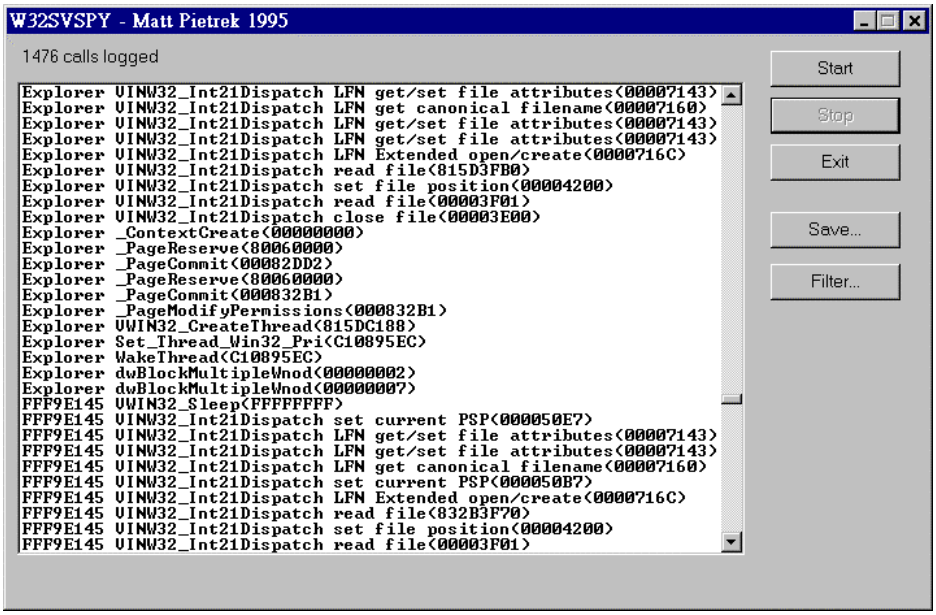



**圖 6-6　在 W32SVSPY 控制之下執行 CALC 程式。**

```
... Many preliminary calls to DOS and the registry by the Explorer omitted

// Create the memory context for the new process.
Explorer _ContextCreate(00000000)

Explorer _PageReserve(80060000)
Explorer _PageCommit(00083FD1)
Explorer _PageReserve(80060000)
Explorer _PageCommit(000861D7)
Explorer _PageModifyPermissions(000861D7)

// Create the initial thread for the new process.
Explorer VWIN32_CreateThread(8160A188)

// Set the priority of the initial thread of the new process.
Explorer Set_Thread_Win32_Pri(C10A3D50)

// Wake up the initial thread of the new process. This should cause
// the new memory context to be switched to.
Explorer WakeThread(C10A3D50)
```

```
Explorer WakeCrst(0001BDA4)
Explorer dwBlockMultipleWnod(00000002)

// The first act of the new process (take a nap!)
FFC44585 VWIN32_Sleep(FFFFFFFF)

// Do file I/O (presumably with the EXE file of the process.
FFC44585 VINW32_Int21Dispatch set current PSP(00005007)
FFC44585 VINW32_Int21Dispatch LFN get/set file attributes(00007143)
FFC44585 _PageCommit(0008160B)
FFC44585 VINW32_Int21Dispatch LFN get/set file attributes(00007143)
FFC44585 _PageDecommit(0008160B)
FFC44585 VINW32_Int21Dispatch LFN get canonical filename(00007160)
FFC44585 VINW32_Int21Dispatch set current PSP(000050C7)
FFC44585 VINW32_Int21Dispatch LFN Extended open/create(0000716C)
FFC44585 VINW32_Int21Dispatch read file(861D3F70)
FFC44585 VINW32_Int21Dispatch set file position(00004200)
FFC44585 VINW32_Int21Dispatch read file(00003F01)
FFC44585 VINW32_Int21Dispatch set file position(00004200)
FFC44585 VINW32_Int21Dispatch read file(00003F01)
FFC44585 VINW32_Int21Dispatch LFN get canonical filename(00007160)

// Reserve and commit the memory to be used by the process.
FFC44585 _PageReserve(00000400)
FFC44585 _PageCommit(00000400)    ← repeat similar calls 11 times
FFC44585 _PageReserve(80000400)
FFC44585 _PageCommit(00000420)
FFC44585 _PageReserve(80000400)
FFC44585 _PageCommit(00000440)
FFC44585 _PageFree(00440000)
FFC44585 _PageFree(00420000)

FFC44585 VINW32_Int21Dispatch set current PSP(0000500B)

// Tell the system debugger (WINICEE) about the EXE's 6 sections.
FFC44585 VWIN32_Int41Dispatch(00000150)   ← repeat this sequence 6 times

FFC44585 _RegQueryValueEx(C11C2EF0)

// Reserve and commit the memory range where SHELL32.DLL is mapped in.
// Since some other process has already loaded SHELL332.DLL, certain
// SHELL32.DLL pages can be shared with this process. Share the pages
// by calling PageAttach.
FFC44585 _PageReserve(0007FE00)
FFC44585 _PageCommit(0007FE00)
```

```
FFC44585 _PageAttach(0007FE01)
FFC44585 _PageAttach(0007FE62)
FFC44585 _PageAttach(0007FE66)
FFC44585 _PageAttach(0007FE69)
FFC44585 _PageCommit(00000420)
FFC44585 _PageCommit(00000520)
FFC44585 _PageAttach(0007FE6A)
FFC44585 _PageAttach(0007FE6B)

// Tell the system debugger (WINICE) about all the DLL sections
FFC44585 VWIN32_Int41Dispatch(00000150)   ← repeat this sequence 33 times

// Checks if the debugger wants control during a fault
FFC44585 VWIN32_Int41Dispatch(0000007F)

FFC44585 _VWIN32_Get_Thread_Context(00000000)
FFC44585 _VWIN32_Get_Thread_Context(00000000)
... Omitted rest of output
```

**圖 6-7　萃取後的 W32SVSPY 輸出結果（啓動 CACL.EXE 之後）**

研究圖 **6-7**，你可以看到一個新行程的誕生。注意，出現在每一個 event 左手邊的行程名稱是現行行程名稱。大部份對 Win32 VxD services 的呼叫是由 KERNEL32.DLL 發出，而非現行行程（本例是指 Explorer）。

這份輸出結果是從 Explorer 產生一份新的 memory context 開始記錄。然而，memory context 不是一個行程能夠獨力完成的，有更多事情要做。後續數行表示新行程的第一個執行緒的誕生，以及執行緒優先權的設定。KERNEL32 的下一個動作是對新執行緒呼叫 *WakeThread*，使新執行緒成爲現行（current）執行緒。

一旦新執行緒啓動，它便開始執行檔案 I/O。可想而知這時候執行緒開始檢查其 EXE 檔案並載入到記憶體中。欲將 EXE 載入記憶體，行程必須呼叫用於分頁記憶體管理的 Win32 VxD services，在 process context 之中保留並委派記憶體。在此過程中，系統知道它可以共享某些 DLLs 的 pages，那些 DLLs 先前已被載入到其他行程的私有位址空間。本例中這樣的 DLL 是 SHELL32.DLL。一旦系統了解某些 pages（主要是 code pages）可以被共享，它就使用 VMM_PageAttach service 真正實現共享。

## 撰寫 W32SVSPY 的技術挑戰

當我最初撰寫 W32SVSPY 時，第一跳出的問題是，KERNEL32 *VxDCall* 函式必須能夠在系統的任何程式的 memory context 中被呼叫。而我稍早說過，Win32 VxD services 是經由 *VxDCall* 喚起的。因此我必須把這個「處理 service call 重導向」的碼放置在可以被所有行程都取用到的記憶體之中。換句話說，它應該被放在共享記憶體中。

另一個問題是，我不能夠使用檔案 I/O 來記錄 W32SVSPY 的運轉記錄（記錄哪些個 Win32 VxD services 被呼叫）。因爲檔案的 handle 只在開啓該檔案的 process context 中才有效。每次一有 Win32 VxD service 被喚起，就開檔、關檔，顯然並不理想。KERNEL32 利用 Win32 VxD service 以實現檔案 I/O，因此更會引發 reentrancy （重複進入）。這個問題的解答還是要靠共享記憶體。W32SVSPY 把所有運轉記錄都放在一個記憶體緩衝區中，並從中取出、顯示。

第三個問題是如何攔截 Win32 VxD services。稍早我曾經說過，*VxDCall* 函式看起來像這樣：

```
VxDCall:
MOV     EAX,DWORD PTR [ESP+04]
POP     DWORD PTR [ESP]
CALL    FWORD PTR CS:[BFFC8004]
```

欲攔截這樣一個 Win32 VxD service，似乎只要簡單地在 BFFC8004 位址上安插一個我的函式位址，然後再串接原來位址就好。事實上那不太困難，困難的是如何找出 *VxDCall* 的位址。

爲什麼不直接對著 *VxDCall* 呼叫 *GetProcAddress* 就好？因爲這個函式是未公開的，並未以函式名稱開放（experted）出來。我不能夠以函式序號呼叫 *GetProcAddress*，第 3 章已經說過，不能夠以序號呼叫 KERNEL32 的函式。爲什麼微軟這麼做呢？毫無疑問就是要阻止像 W32SVSPY 這樣的程式被發展出來。現在你看到了，微軟的企圖失敗了，W32SVSPY 獲得了勝利。

## 把 W32SVSPY 放到共享記憶體中

第一個重要動作就是把 W32SVSPY 放到共享記憶體中。在第 3 章中我曾經示範一種利用 DLL 的作法，因此 W32SVSPY 除了 EXE 之外還有一個 DLL 兄弟（W32SPDLL.DLL）。然而，光只是把我的碼放入 DLL 還不夠，DLL 還需要被載入到某塊被所有行程共享的記憶體中。在 Windows 95 中這意味著 2GB~3GB 位址之間，那是像 KERNEL32 或 USER32 等 system DLLs 駐紮的地方。我們必須讓 Windows 95 把 W32SPDLL.DLL 也載入到那塊區域去。

好，你的第一個衝動就是將 W32SPDLL.DLL 的基底位址（2GB~3GB）告訴聯結器。當然你可以這麼做，但是我的經驗顯示，那還不夠。哦不，DLL 是可以正確載入，問題是作業系統又把 DLL 重新載入到應用程式的私有位址空間去。很顯然，Windows 95 不希望我的 DLLs 佔用保留給 system DLLs 以及 shared memory 的位址空間。

研究過微軟的那些 system DLLs 之後，我發現，每一個能夠被 Windows 95 載入器成功載入到 2GB 以上（可共享之位址空間）的 DLLs，它們的可寫入資料節區（writeable data section）都是可共享的。回想一下，其實這是十分明白的道理，如果載入器把 DLLs 載入到共享區域中，Windows 95 當然就不能夠在其資料節區中提供給每一個行程私有空間了 -- 那會引起問題的。

經由下面兩個步驟，我終於能夠把 W32SPDLL.DLL 載入 2GB 以上：

- 告訴聯結器說這個 DLL 的基底位於 2GB 以上。我使用 KERNEL32 的基底位址，因爲我知道 Windows 95 載入器會對重疊的基底位址做重新定位的動作。
- 讓 DLL 的 .data、.idata、.bss 節區都成爲共享節區。我在 makefile 中向聯結器發出 /section: 命令而完成這項任務。RWS 是 readable、writeable、shared 的意思：

```
-BASE:0xBFF70000
/section:.data,RWS
/section:.idata,RWS
/section:.bss,RWS
```

## 找出 VxDCall 的位址

稍早我曾經說過，Windows 95 的 KERNEL32 強烈阻止應用程式取得 KERNEL32 中的未公開函式（如 *VxDCall*）的位址。*GetProcAddress* 對那些函式起不了作用。然而，如果你能夠以隱寓（implicitly）方式聯結一個函式，你就可以輕易獲得其位址。在 C 或 C++ 之中，你可以使用函式名稱（不要加上括號）。

附錄 A 中，我展示我爲將近 100 個 KERNEL32 未公開函式所建造的一個 import library。利用這個 library，你可以根據序號來輸入（import）一個函式。在 W32SPDLL.C 中，我必須先做這樣的原型宣告：

```
__declspec(dllimport) int WINAPI VxDCall0(void);
```

然後我必須取得其函式位址（利用函式名稱）：

```
pfnVxDCall0 = VxDCall0;
```

一旦知道函式位址，要進入（呼叫）該函式碼就十分簡單了，我也可以取出指向 INT 30h 指令的 16:32 位址：

```
ppfnOriginalVxDCall = (PBYTE)*(PDWORD)((DWORD)pfnVxDCall0 + 0xA);
```

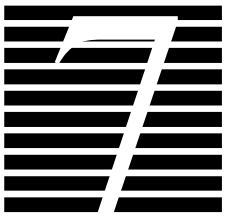
這一行看起來很不聰明，但其實它不差。它取走一個 DWORD（在 *VxDCall* 函式之後的 0A 個位元組），然後把它轉型爲一個指標。

是的，依賴 *VxDCall* 函式的一個固定偏移位置以找出我要的一個指標，真是令人發嘔。但是當你寫像 W32SVSPY 這樣的低階系統工具，這麼做是很平常的。微軟只要重新安排 *VxDCall*，讓我所要的碼不出現在 0xA 偏移處，很容易就可以使 W32SVSPY「掛」了。但這麼一來它們就真的不只是惡意破壞 W32SVSPY，而且是毫無理由地弄亂了 *VxDCall*。讓我們拭目以待 Windows 95 的表現。

# 摘要

我在本章丟出了一大堆未公開函式以及極端技術性的東西。把它們統統記憶下來倒非絕對必要，重要的是我嘗試轉達這樣的觀念：Windows 95 有三個足以被稱爲核心的元件，分別是 KRNL386.EXE、KERNEL32.DLL、VWIN32.VXD。每一個元件都對其他元件有十足的認識與充份的互動。

了解其中任何一個元件，卻不了解其他兩個元件，是很困難的。你可以一再一再地回頭重讀那些比較複雜的部份。我希望我已經清楚顯示了這三個 Windows 95 核心元件之間的彼此互動關係，並且促使你有能力自力挖掘出更多東西。我知道，在我面前一樣有更多的探險活動需要進行。



# Win16 的 Modules 和 Tasks

譯註：本文之中如果 module 和 task 聯用，我就都使用原文。如果單獨使用，我就把 module 譯爲「模組」。"task" 沒有什麼好譯詞，姑且使用原文。

在一本專注於 Windows 95 32 位元架構的書籍中放一章 16 位元 KERNEL 資料結構，是不是有些詭異？然而，很快你就會看到，這些資料結構在 Windows 95 之中扮演非常重要的角色 -- 對 16- 或 32- 位元程式都是如此。

我們將在這一章對 KRNL386 所維護的 16 位元 modules 和 tasks 做一次旅遊。如果你熟悉 Windows 3.1 的 modules 和 tasks，驚鴻一瞥之下它們和 Windows 95 中的弟兄似乎沒有不同。在新而令人興奮的 Win32 元件尚未探索完畢之前，爲什麼要操心那些舊的 16 位元元件？哈，只要稍稍挖深一些，你就會看到 16 位元 KRNL386.EXE、32 位元 KERNEL32.DLL 以及 VWIN32 VxD 彼此之間互通氣息，糾葛不清。檢閱 16 位元的 modules 和 tasks，是學習 Windows 95 架構的重要部份。

我們將從模組開始，那是 Windows 95 16 位元端用以追蹤所有載入於系統之中的 EXEs 和 DLLs 的機制。有趣的是，Windows 95 不僅止爲 16 位元的 EXEs 和 DLLs 產生 16 位元模組，它也爲 32 位元 EXEs 和 DLLs 這麼做。在描述過 16 位元模組規格之後，我會展示一些有用的 16 位元 KRNL386 函式虛擬碼給你看，用以示範如何使用模組。

接下來我要進入 16 位元 tasks 主題，以及 KRNL386 用以維護 tasks 的資料結構。Tasks 由 modules 產生，所以當然先描述 module 才符合自然。如果「Windows 95 爲 32 位元 EXEs 和 DLLs 產生 16 位元 modules」這個事實還不夠你驚訝，「Windows 爲每一個 Win32 行程維護一個 16 位元 task」應該可以補上一腳吧！描述過 tasks 的佈局和特徵之後，我會展示一些 KRNL386 函式虛擬碼給你看，用以示範如何使用 tasks。

在本章最後的「SHOW16 程式」一節，我寫了一個 16 位元 SHOW16.EXE 程式，讓你能夠方便地瀏覽系統中的 16 位元 modules 和 tasks。雖然我可以利用 TOOLHELP 的幫助，但我選擇直接存取 tasks 和 modules 資料結構，用以證明它們並不是什麼只有微軟程式員才能碰的奇怪事物。SHOW16 的輸出結果可能會令你驚訝。

一頭鑽入細節之前，我必須先說明一點。整章之中我常常提到 global memory handle 和 CPU selector，並似乎把它們當做同一個東西。在 Windows 3.1 和 Windows 95 中，一個 16 位元的 FIXED global heap handle 的確是一個 ring3 selector。至於 MOVEABLE handle 則可以輕易轉換爲一個 selector：把最低位元改爲 1 即可，這也正是 *GlobalLock* 的行爲。我事先做此聲明以免處處都要再說一遍。在本章的討論中，它們的確可被視爲相同的東西。

## 為什麼 32 位元模組和行程有對 16 位元表象？

或許你會奇怪爲什麼微軟要以 16 位元的舊東西表現其 32 位元 EXEs 和 DLLs。答案很簡單：Windows 95 不是 NT，它並沒有把 16 位元程式隔離在另外的虛擬機器中，而是讓 16-/32- 位元程式共存於同一個虛擬機器，甚至共享某些位址空間（詳情請看第 5

章）。此外，被所有 Windows 95 程式所使用的大部份碼都放在 16 位元 DLLs 中（例如 USER、GDI、COMMDLG...等等，甚至於 KRNL386）。

# 16-bit Modules

在 Windows 95 的 16 位元世界裡，模組（modules）是「KRNL386 爲了表達一個 EXE 或 DLL 被載入記憶體後的程式碼、資料、資源」而使用的一種資料結構。這裡所謂的 DLL 涵蓋不同的副檔名，例如 .DRV 和 .FON。每一個模組都和磁碟某處的一個檔案有關聯。

這一節我要描述從 16 位元 Windows 3.1 模組演化而來的 Windows 95 16 位元模組。每一個被載入的 EXE 或 DLL，不論它是 16 位元或 32 位元，都擁有一個 16 位元模組。32 位元 EXE 或 DLL 另外還擁有一個 32 位元模組，由 KERNEL32.DLL 掌管。一般而言，16 位元模組給 16 位元 system DLLs 使用，32 位元模組給 32 位元 system DLLs 使用。第3章曾經介紹過 32 位元模組。

16 位元模組的資料存放在一個以 *GlobalAlloc* 配置而來的節區中，名爲 module database。這塊記憶體的 handle 則被稱爲 module handle，或 HMODULE。*GetModuleHandle* 所獲得的就是這樣一個 handle。

在 Windows 3.1 之中，所有模組都由 *LoadModule* 函式產生。*LoadLibrary* 和 *WinExec* 其實都是呼叫 *LoadModule*。在 Windows 95 之中，16 位元 EXEs 和 DLLs 的 16 位元模組還是由 KRNL386 的 *LoadModule* 產生，32 位元 EXEs 和 DLLs 的 16 位元模組則由 KERNEL32.DLL 產生。後者並不列名於 global heap handle list 之中，所以要找到它們有些棘手。

16 位元 module database 的格式是以 Windows 和 OS/2 1.x 所使用的 16 位元可執行檔格式爲基礎。這個檔案格式稱爲 New Executable（NE）格式。從現在開始，我將以「NE 模組」表示 16 位元模組，以「PE 模組」表示 32 位元模組。PE 是 Portable Executable 的縮寫，是 Win32 可執行檔格式。我不在這一章描述 NE 格式，因爲微軟文件及其他地方都有不錯而且充份的說明。如果你把 NE 檔案格式和 NE module database 拿來做比

較，你會發現雖然它們類似，卻有幾個重要的差異。

NE 檔一開始是個表頭，佔據 0x40 個位元組，稱爲 NE 表頭。這個結構之所以有此名稱是因爲最前面兩個位元組是 0x454E，正是 "NE" 的 ASCII 碼。*LoadModule* 把 NE 表頭讀進來放到它所建構的 module database 中，所以 NE 檔的許多欄位和記憶體中的 NE 模組一模一樣。然而 KRNL386 將其中一些只對磁碟中的 NE 檔有意義的欄位拿來再利用，賦與它們不同的內容和意義。

Module database 的 NE 表頭之後，放的是 segment table。那是一個結構陣列，內含許多對於模組之程式碼和資料而言極其重要的資訊。再下來是 resource table，內含 NE 檔中的資源相關資訊。KRNL386 可以根據它找出 NE 檔中的每一筆資源。

兩個 tables 之後，緊接著是有關於模組的輸入（imports）和輸出（exports）資訊。呼叫（calling）一個函式，又稱爲輸入（importing）一個函式。例如「USER.EXE 呼叫 KRNL386 函式」，我們也可以說是「USER.EXE 輸入了 KRNL386 函式」。所謂輸出一個函式，意思是你開放一個函式供別人（EXEs 或 DLLs）呼叫。譬如說：KRNL386 輸出了一個函式，而 USER 將它輸入進來。

所謂輸出一個函式，意思是你把函式位址放到一個名爲 entry table 的地方。當你載入 NE 檔，而它從另一個模組輸入函式，Windows 載入器（也就是 *LoadModule* 函式）會根據 entry table 尋找該模組的輸出函式位址。載入器怎麼知道要看表格中的哪一個元素呢？是的，當你輸出一個函式，聯結器會指定一個序號給它，當做 entry table 的索引。

以名稱（而不是序號）輸入一個函式也是可能的。這正是 resident name tables 和 nonresident names tables 發揮功效的地方。這兩個表格把模組所開放的函式的「位址」和「名稱」做一個關聯。Module database 內含整份 resident names table，但只含有一個「指向 nonresident names table」的檔案偏移位置（這也就是 “resident” 和 “nonresident” 的區別）。

Windows 3.1 的 KRNL386 把 module database 維護成一個串列。當 LoadModule 產生新

模組，便加到串列尾端。串列頭是 KRNL386 -- 第一個進入系統的模組。你很容易就可以走訪模組串列（像我在 SHOW16 中的作爲），或是讓 TOOLHELP 的 *ModuleFirst* 和 *ModuleNext* 爲你服務。基本上 TOOLHELP 的動作和 SHOW16 相同，但它被微軟正式核准使用，而「自行探險」則未獲官方承諾。

在 Windows 95 之下，KRNL386 以相同的方法維護 16 位元 EXEs 和 DLLs 的 NE 模組。然而 32 位元 PE 檔的 16 位元 module database 卻不會被系統加到串列之中。它們統統被掛在 global heap 之內，和 16 位元的模組串列沒有什麼關聯。在 16 位元程式中我不知道有什麼方法可以列舉這些 32 位元程式的 NE 模組。然而暴力法可以得逞，SHOW16 會示範給你看。

深入其實際格式之前，讓我們從高階眼光看看 16 位元 module database 的內容。如圖 **7-1** 所示，NE 表頭在最前面，然後是 segment table 和 resource table。然後是與函式輸入有關的表格（imported name table 和 module reference table），然後是與函式輸出有關的表格（entry table 和 resident names table）。

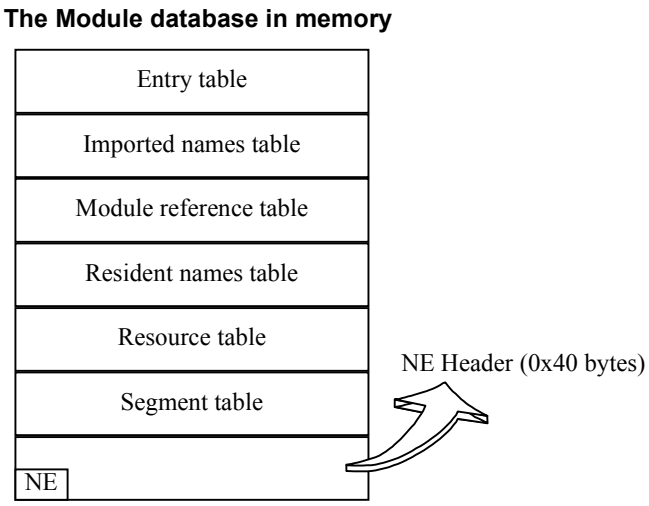



**圖 7-1　16 位元 module database 中的各式元件。**

# NE 表頭（NE Header）

如果你只想快速瀏覽 module database 的所有欄位，請看 SHOW16 的 HMODULE.H 檔案。這裡，我將詳細描述 NE 表頭的 0x40 個位元組。每一個欄位的第一行表示其偏移位置、型態、及簡短說明。

### 00h WORD Signature

此欄總是爲 0x454E，代表 "NE"。在可執行檔檔案起始處放一個標記，是 Microsoft 和 IBM 作業系統的傳統。其他標記還包括有 PE（Win32 Portable Executable）、LE（Linear Executable，使用於 Windows 3.1 和 Windows 95 的 VxD），LX（也是 Linear Executable 之意，使用於 32 位元 OS/2 2.x 程式）。

### 02h WORD module usage (reference) count

此一模組被其他模組使用的次數。如果此 DLL 被三個程式使用，此值爲 3。如果你以 *LoadLibrary* 載入一個 DLL，該 DLL 的 module database 的此一欄位將爲 1；以後每次呼叫 *LoadLibrary* 或 *LoadModule*，此值即累加 1；每次呼叫 *FreeLibrary*，此值即累減 1。然而，用以決定其值之規則其實不很清楚。例如一個程式使用了 DLL A 和 DLL B，它們兩個又都使用了 DLL C，那麼 DLL C 的此一欄位竟是 1 而不是 2。我在 1994 年五月份 ***Microsoft Systems Journal*** 上的 Q&A 專欄中介紹過 *IncExeUsage* 和 *DecExeUsage*，那是 KRNL386 內部函式，用來增減模組的使用次數值。其中對於循環使用的模組，有粗暴的立場。PE 檔所產生的 NE 模組中，此值被設爲 1 之後永不改變。

當此值變爲 0，KRNL386 釋放模組的節區和資源，呼叫 WEP 函式（如果它是一個 DLL，並且如果它有 WEP 函式的話）。最後，呼叫 *GlobalFree* 釋放 module database 節區。

### 04h WORD near pointer to entry table

這是一個近程指標（相對於 HMODULE 節區），指向模組的 entry table。後者是模組之中開放給其他模組使用的函式列表。每一筆資料包括函式位址、序號、以及某些旗標值，詳細請看「Entry Table」一節。Win32 可執行檔的 NE 模組並不含有 entry table，因爲 entry

table 使用 16 位元遠程位址，而 PE 模組並不支援之。

### 06h　HMODULE　next module database

NE 模組串列中的下一個 HMODULE。KRNL386 永遠在串列的頭。兩種方法可以獲得 KRNL386 的 HMODULE：要不就呼叫 *GetModuleHandle*(KERNEL)，要不就對任何模組呼叫 *GetModuleHandle*，而 KRNL386 的 HMODULE 將放在 DX 中傳回。當新模組被載入，它們附在串列尾端。串列中最後一個模組的此一欄位爲 0。Win32 可執行檔的 NE 模組不在這個串列之中，所以它們的這一欄位內容都是 0。

### 08h　WORD　near pointer to DGROUP segment entry

這是一個近程指標（相對於 HMODULE 節區），指向 segment table 中與模組 DGROUP 節區有關的項目。segment table 的格式將在「Segment Table」一節描述。DGROUP 是模組一般性資料的駐足場所。DGROUP 通常內含一個 local heap。而如果模組是 EXEs，DGROUP 還內含一個堆疊（stack）。Win32 EXEs 和 DLLs 所產生的 NE 模組，此欄位爲 0。

### 0Ah　WORD　near pointer to OFSTRUCT with file name

這是一個近程指標（相對於 HMODULE 節區），指向一個極類似 OFSTRUCT 的結構（可在 Win16 的 WINDOWS.H 中找到其定義）。在 Windows 95 DDK 中，16 位元 WINDOWS.H 檔稱此結構爲 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;
```

它與傳統 OFSTRUCT 之間的主要不同在於 cBytes 欄位是個 WORD，而不是 BYTE。

爲什麼？因爲 Windows 95 支援長檔名（超過 260 個字元）。因此單一 BYTE 無法表示結構全長。此外，結構的尾端（內含路徑名稱）是 260 個位元組，而不是 OFSTRUCT 的 128 個位元組。

### 0Ch WORD module flags

內含一些位元旗標，可以儲存模組在記憶體中的相關資訊。這些旗標中許多位元的意義與它們原本在 NE 檔中的意義不同。Windows 95 的 NE 模組旗標意義如下：

| 旗標名稱和位元值 | 說明 |
|---|---|
| MODFLAGS_DLL<br>0x8000 | 對真正的 NE 模組而言，此旗標表示它是個 DLL，而不是個 EXE。但 Win32 模組的 NE module database 中此位元總是設立。 |
| MODFLAGS_CALL_WEP<br>0x4000 | 此旗標只對 DLL 模組有效，表示 WEP 函式應該在 DLL 退出記憶體時被呼叫。此旗標總是設立，除非是 task 模組或 Win32 模組。 |
| MODFLAGS_SELF_LOADING<br>0x0800 | 此旗標表示這個模組使用自我載入（self-loading）機制。模組提供自己準備好的節區載入器，給 LoadModule 呼叫。微軟強烈不鼓勵大家使用自我載入機制，也很少公開說明它的用法。然而微軟過去的數個產品（包括 Word 和 Fortran）都使用這項特性。<br><br>SLR systems 公司（現隸屬於 Symantec 公司）的 OPTLINK 5.x 所產生的程式都使用自我載入機制，以縮減可執行檔大小。當 OPTLINK 將節區資料寫到 NE 檔，會先經過壓縮。一旦模組被載入，程式內建的自我載入碼會把資料解壓縮。 |
| MODFLAGS_APPTYPE<br>0x0300<br>(0x0200 \| 0x0100) | 這兩個位元是 OS/2 1.x 的遺物。那個時候程式的使用者介面有三種可能。位元值 0x0300 表示使用作業系統的 GUI windowing API，位元值 0x0200 表示程式是個 console（文字模式）程式，有點像是在 Windows 中跑一個 MS-DOS prompt。位元值 0x0100 表示應用程式直接處理視訊記憶體，所以它必須在全螢幕模式下執行。<br><br>一般的 Windows NE 模組總是有值爲 0x0300，意思是使用 GUI API。在 Windows 95 中，Win32 檔案的 NE 模組並不設立任何旗標值，也就是說此旗標爲 0，此值在 NE 規格書中沒有定義。 |

| | |
|---|---|
| MODFLAGS_IMPLICIT_LOAD<br>0x0040 | 此旗標表示模組現在正在記憶體中。Tasks 模組無此旗標，DLL 若被 LoadLibrary 載入的話，也無此旗標。然而如果某個 DLL 被 LoadLibrary 載入，而它又隱式載入（implicitly loads）其他 DLLs，那麼那些 DLLs 將有此旗標。 |
| MODFLAGS_WIN32<br>0x0010 | 這是 Windows 95 新旗標，表示此 NE 模組代表一個 Win32 PE 檔。 |
| MODFLAGS_AUTODATA<br>0x0002 | 這個旗標告訴 Win16 載入器說，每一模組都應該有一個分離的 DGROUP。此旗標只出現於 EXE 模組，因爲 EXE 的每一個執行個體（instance）需要自己的一個 DGROUP。 |
| MODFLAGS_SINGLEDATA<br>0x0001 | 這個旗標表示，此模組的所有使用者共用同一個 DGROUP。此旗標只針對 DLL 模組設立，因爲不管誰呼叫它，Win16 DLL 總是使用相同的 DGROUP。<br><br>如果 MODFLAGS_AUTODATA 和 MODFLAGS_SINGLEDATA 都沒有設立，表示此模組沒有 DGROUP，也就沒有 local heap。有趣的是 SYSTEM 模組（在 KERNEL 之後被載入）就是屬於此一類型。<br><br>32 位元檔的 NE 模組總是內含 0x8010 旗標值，也就是 MODFLAGS_DLL 和 MODFLAGS_Win32。 |

### 0Eh　WORD　segment index of DGROUP segment

從 1 開始起算的索引值，用以挑出 segment table 中所記錄的 DGROUP 節區。這個欄位有時候嫌累贅，因爲 08h 欄位就已經可以將 DGROUP 節區定位出來。在 Win32 NE 模組中，此欄位恆爲 0。

### 10h　WORD　initial local heap size

Windows 載入器保留給「DGROUP 節區中的 local heap」的大小。必要時 local heap 可以成長。如果 heap 成長，這個欄位不會隨之改變（改變它並不合理，因爲其他執行副本啓動時仍需此初值）。許多標準的 16 位元 system DLLs 的此一欄位爲 0。有趣的是，Windows 95 中有許多 system DLLs，其 heap 的初始大小比對應的 Windows 3.1 DLLs

小。這或許是爲了儘可能降低 Windows 95 記憶體的初始用量。在 Win32 NE 模組中，此欄位恆爲 0。

### 12h WORD stack size

Windows 載入器保留給「DGROUP 節區中的堆疊（stack）」的大小。此值只對 EXE 模組有效，因爲 DLL 模組係使用呼叫端的堆疊。16 位元程式的堆疊最小爲 5K。如果此欄位小於 5K，載入器會以 5K 視之。在 Win32 NE 模組中，此欄位恆爲 0。

### 14h DWORD entry point of module

程式進入點。對 EXE 模組而言，此欄位所指之處即爲程式開始之處。如果是 C/C++ 程式，此欄位即指向 C/C++ startup code，後者再呼叫 *WinMain*。對 DLL 模組而言，程式進入點將呼叫 *LibMain*。EXE 模組必須有一個非零的進入點，DLL 模組（例如字形檔）則允許有 NULL 進入點 -- 這種情況下載入器不會呼叫 DLL 的任何碼。

儲存於此欄位的是一個邏輯位址，也就是 16:16 位址，但前半部並不是真正的 selector，而是一個索引值，用以挑出 segment table 中的一個元素。因此，如果進入點位於模組第３個節區的 0x017C 處，此欄位將是 0003:017C。當載入器真正需要進入點（位址）時，它必須計算真正的 selector。

在 Win32 NE 模組中，此欄位恆爲 0。這很合理，因爲 32 位元碼使用 32 位元線性位址，而不是 16:16 遠程指標。

### 18h DWORD initial stack pointer value

此欄位內含「當 EXE 進入點被呼叫時」的 SS:SP 值。和前一欄位一樣，它也是個邏輯位址，而不是真正的 selector:offset。其前半部份應該和 0Eh 偏移處所指的 DGROUP 節區相同。在 Win32 程式的 NE 模組中，此欄位恆爲 0。

### 1Ch WORD number of segments in module

模組的節區個數。NE 表頭之下的 segment tables 中有一筆筆的項目，都是 10 位元組（譯

註）。那些項目的總個數就是此一欄位的值。模組也可能沒有任何節區，例如字形模組。字形模組通常只含資源，沒有程式碼也沒有資料。在 Win32 NE 模組中，此欄位恆爲 0。

> 譯註：如果你瞭解 NE 檔案格式，可能你會在這裡得意地指出作者的錯誤：「應該是 8 個位元組」。但，你錯了，這裡說的是 NE 模組而非 NE 檔案，而兩者在此一欄位有些差異。請看下一節「Segment Table」的第一段文字說明。

### 1Eh　WORD　number of imported modules

模組隱式聯結（implicitly links）的 DLLs 個數。如果一個 EXE 呼叫了 KERNEL、USER、GDI，這個欄位就是 3。此欄位和 module reference table 合作（請看 28h 欄位），後者的項目個數必須與此欄位相符合。在 Win32 NE 模組中，此欄位恆爲 0。

### 20h　WORD　size of nonresident names table

這個欄位內含 nonresident names table 的大小。該表格（以及 resident names table，請參考欄位 26h）把一個輸出符號（通常是函式名稱）和其序號關聯起來。爲了處理 nonresident names table，KERNEL 必須尋找表格在 NE 檔中的起始位置（請參考欄位 2Ch），並讀入此處所指定的大小。請參考稍後的「Resident/Nonresident Names Tables」一節。在 Win32 NE 模組中，此欄位恆爲 0。

### 22h　WORD　near pointer to segment table

這是一個近程指標（相對於 HMODULE 節區），指向 segment table。segment table 的每一元素都是 10 個位元組，每一個元素代表一個 code 節區或 data 節區。由於 segment table 總是位於 NE 表頭的後方，所以此欄位總是爲 0x40。在 Win32 NE 模組中，此欄位恆爲 0x4C -- 但那是沒有意義的，因爲 Win32 檔案沒有 16 位元節區，也沒有 segment table。

### 24h WORD near pointer to the resource table

這是一個近程指標（相對於 HMODULE 節區），指向 resource table。Resource table 是一種表格中的表格，用以表示真正的資源，其格式我會在後面小節中提到。有趣的是，這個欄位在 16- 和 32- 位元 NE 模組中都可用。所以我猜想處理資源的 16 位元碼也適合處理 32 位元 PE 檔案。

### 26h WORD near pointer to resident names table

這是一個近程指標（相對於 HMODULE 節區），指向 resident names table。這個表格用來讓一個輸出函式的名稱和其序號產生關係。其格式與 nonresident names table 一樣，關鍵差異在於前者總是駐留於記憶體中，後者可能被置換到磁碟去。這兩個 name tables 的格式將於後面小節中描述。

所有 NE 模組都有 resident names table -- 不論此模組來自 NE 檔或 PE 檔。這是因爲每一個模組都必須有名稱（例如 KERNEL、USER、TOOLHELP 等等等），而 resident names table 的第一筆資料就是本模組名稱。因此，當 KERNEL32 建立它自己的最小的 NE module database 時，其 resident names table 總是只有一個元素，就是自己的模組名稱。

### 28h WORD near pointer to the module reference table

這是一個近程指標（相對於 HMODULE 節區），指向 module reference table，其中記錄著本模組所使用的所有模組。其實這就是一個 HMODULE 陣列。在可執行檔的重定位表格（relocation table）中，你可以找到每一個輸入函式（imported function）的重定位資訊，內含一個索引值，挑出 module reference table 中的一項（代表一個模組）。假設程式呼叫了 GDI.EXE 的 *SetPixel* 函式，序號 31。而在 module reference table 中 GDI.EXE 排行第 4。那麼 *SetPixel* 的重定位資訊中將包括 4（代表 GDI.EXE）和 31（代表 *SetPixel*）。

### 2Ah WORD near pointer to the imported names table

這是一個近程指標（相對於 HMODULE 節區），指向 imported names table。該表格內

是一些 pascal 字串（譯註：pascal 字串的第一個位元組表示字串長度，字串最後不需結束字元。又稱爲 LString），記錄本模組用到的所有 DLLs 名稱。這個表格也可以記錄那些以名稱而非序號輸入進來的函式(但這種情況極少)。當 KRNL386 產生一個 module database，它會尋找（找不到就載入）此表格所記錄的 DLLs。一旦找到（或載入）DLLs，便把其 HMODULE 填到 28h 欄位去。模組產生之後，這個欄位就用不上了。Win32 NE 模組中，此一欄位不是零，但沒有什麼意義，因爲彼時根本就沒有 imported names table。

### 2Ch　DWORD　file offset of the nonresident names table

這個欄位內含「nonresident names table 在 NE 檔的位置」（以位元組計）。它必須與 20h 欄位互相配合，才能把表格載入記憶體中。在 Win32 NE 模組中，此欄位恆爲 0。

### 30h　WORD　number of moveable entries in the entry table

這個欄位已經老舊，並且隨著 Windows 真實模式的終止而停止使用了。在 Windows 保護模式中，selectors 和 descriptors 將節區搬移的事實隱藏起來，所以這個欄位不再需要。

### 32h　WORD　alignment shift

在 NE 模組中，節區真實資料的檔案位置不是以一個 DWORD 偏移值記錄著，而是以 sector 爲單位。這裡的 sector 並不是磁碟的 sector。它總是 2 的冪次方，例如 2, 4, 8, 16, 32...。爲了獲知一個 NE 模組的 sector 大小，你必須以此欄位做爲 2 的冪次方。

典型的 NE 模組中，此欄位爲 9，所以 sector 大小爲 512。如果節區從檔案的 1536 位元組開始，那麼它的位置應該記錄爲 sector 3。另一個常見的數值是 4，那麼 sector 大小爲 16。你可以在聯結時期指定 sector 的大小。在 Win32 NE 模組中，此欄位恆爲 1。

一般而言，此值較小比較有利。如果 alignment 過大，會浪費磁碟空間，因爲聯結器必須裝填一些數值，才能讓每一個節區和資源都從 sector 的倍數位置開始。節區和資源的 sector 倍數值儲存在一個 WORD 之中，所以如果你的 sector 是 16，節區最大可能是 1MB（65536 * 16 = 1MB）；如果你的 sector 是 512，節區最大可能是 32MB（65536 * 512 = 32MB）。

### 34h WORD unknown

在 Windows 3.1 中，如果模組內含 truetype 字形，此欄位爲 2。在 Windows 95 中，此欄位沒有用到，總是爲 0。

### 36h BYTE intended operating system

此值表示這個模組期望的作業系統，包括：

0. 未知（但其實 Windows 1.0 使用此值）
1. OS/2
2. Windows
3. European DOS 4（一個多工版的 DOS，但不曾在美國發行）
4. Windows/386（只在 Windows 2.x 時代出現過）

在 Win32 NE 模組中，此值爲 0。

### 37h BYTE other module flags

這是 Windows 1.x 之後新增的旗標。在 Windows 3.x 中， 0x02 和 0x04 用來表示「在 Windows 2.x 中開發的模組可以在 Windows 3.x 中執行無誤」。由於 Windows 95 並不支援 2.x 或更早的版本，所以此值沒有意義。

0x08 表示 NE 檔有一個 gangload area（快速載入區）。所謂 gangload area 是指某些節區和資源，被放在檔案的同一個（或連續的）cluster 中。Windows 載入器可以使用單一讀檔動作把它們全部讀出，不必多番尋找再讀取。

另有一個 Windows 95 中的新位元旗標（0x10）出現在某些 16 位元模組身上。如果該值設立，KRNL386 就不再搜尋並呼叫 16 位元 DLLs 中的 *DllEntryPoint* 函式。Windows 95 的 Win16 DLL 的 *DllEntryPoint* 函式說明文字出現在與 thunk compiler（THUNK.EXE）有關的微軟文件中。

Win32 NE 模組中的此一欄位總是爲 0。

#### 38h　WORD　near pointer to imported names table

這個欄位用來指向 imported names table，它和 2Ah 欄位意義相同。

#### 3Ah　WORD　near pointer to imported names table

這個欄位用來指向 imported names table，它和 2Ah 欄位以及 38h 欄位意義相同。只有對第一個模組（KRNL386）有點例外，這個例外其實是無傷大雅的失察而已，因爲 KRNL386 是被一段類似 Windows 載入器的碼載入，而此碼的行爲在 Windows 95 中仍然沒變。

#### 3Ch　WORD　unknown

此欄位意義未明，似乎總是 0x10 的倍數。在 Win32 NE 模組中，此欄恆爲 0。

#### 3Eh　WORD　expected windows version

能夠使用此一模組之 Windows 最小版本。一般是 Windows 3.0（0x0300）或 Windows 3.1（0x0310）或 Windows 95（0x0400）或 Windows NT 3.5（0x0350）。此值的較高位元組是 Windows 主版本，較低位元組是 Windows 次版本。正確的列印格式是：

```
%u.%02u
```

SHOW16 將示範其用法。

## Windows 95 中新的 module database 欄位

下面三個是 Windows 95 新增欄位，只存在於 Win32 NE 模組中。對於 16 位元 NE 模組，這三個欄位都不存在。

#### 40h　DWORD　address of associated PE file

這是一個相對虛擬位址（一個 flat 32 位元指標），指向記憶體中的 PE 程式的 32 位元端。這個值和 32 位元模組的 HMODULE 相同。

#### 44h DWORD base address of associated PE file

此值似乎總是和前一個欄位相同。

#### 48h DWORD base address of resource section in memory mapped PE file

這是 32 位元線性位址，指向與此 HMODULE 相關聯的 PE 檔中的資源區域（.rsrc section）。稍後你會在「Resource Table」一節中看到，Windows 95 的 16 位元元件可以知曉 32 位元 PE 檔的資源。

## Segment Table

NE 表頭之下立刻就是 segment table（雖然 Win32 檔的 NE 模組沒有 segment table）。這表格是個結構陣列，每個結構描述一個 code 節區或 data 節區。前 8 個位元組和 NE 檔案中的 segment table 內容一樣，後兩個位元組用來儲存 16 位元 Windows 載入器指定的 selector（指向目標節區）。這一點很重要，KRNL386 總是能夠在一個「NE 檔案的節區」載入記憶體時，把它和「用以指向該節區」的這個 selector 關聯起來。

每一筆 segment table 的結構格式如下：

#### 00h WORD sector offset in NE file

節區原始資料在 NE 檔案中的位置。其單位不是 bytes，而是 sectors。sector 的大小可能在不同檔案中不盡相同，視 NE 表頭的 32h 欄位而定。典型的 sector 大小是 16 和 512。如果此欄爲 0，表示這是個用於未初始化資料的節區，在 NE 檔中沒有原始資料。

#### 02h WORD sector length in file

NE 檔中的節區原始資料的長度。注意，這並不是 KRNL386 應該爲此節區配置的記憶體大小（那應該看 06h 欄位）。爲什麼兩個和大小有關的欄位卻有不同的值呢？最常見的情況是你在 data 節區之後又放上未初始化資料（BSS）。假設你有 3K 真正的資料，另需要 4K 給未初始化資料（例如一個陣列）使用，那麼此欄將是 3K，06h 欄將是 7K。

### 04h　WORD　　flags

一些旗標值，表示節區的某些性質。以下所列的旗標意義與它們在 NE 檔案中完全一樣。然而如果你檢查記憶體中的這個欄位，你會發現有些旗標被 KRNL386 額外設立起來。已知的旗標意義如下：

| 旗標名稱和位元值 | | 說明 |
|---|---|---|
| DATA | 0x0001 | 這是個 data 節區。如果此旗標未設立，則表示這是個 code 節區。 |
| ITERATED | 0x0008 | 此節區內含 iterated（run length encoded，RLE 壓縮法）data。 |
| MOVEABLE | 0x0010 | 此節區可在線性記憶體中移動。如果此旗標未設立，則節區爲 FIXED。Windows 載入器會把 EXE 模組的此一旗標關閉，因爲載入器很少需要 FIXED 記憶體。（譯註：作者的這兩句話矛盾，但是我沒有能力訂正） |
| PRELOAD | 0x0040 | 此節區隨著模組被載入而載入，不是等到被使用了才載入。 |
| RELOC | 0x0100 | 此節區在其原始內容之後緊跟著重定位資訊（relocation information）。 |
| DISCARDABLE | 0x1000 | 此節區可被拋棄。如果記憶體不足，KRNL386 可以將此節區的 descriptor 記錄爲 not-present（不存在），然後爲 RAM 指定其他用途。 |
| 32BIT | 0x2000 | 此節區爲 32-bit code 節區。當載入器爲這樣的節區配置記憶體時，它會將 descriptor 中的 "big" 位元設立起來，以表示此爲 32-bit code 節區。 |

### 06h　WORD　　allocation size

這是 KRNL386 應該爲節區配置的大小。它可能比 NE 檔中真正的節區原始資料還大。請參考 02h 欄位的說明。

### 08h　WORD　　global memory handle

這是 KRNL386 爲節區配置的記憶體（來自 global heap）。如果 handle 以 6 或 E 結

尾（例如 0476h 或 047Eh），這是一個可搬移節區。如果 handle 以 7 或 F 結尾，代表這是個 FIXED 節區。

Segment table 中的結構排列次序是有意義的，因爲那與邏輯位址有關。當聯結器或除錯器需要在模組的節區位址上做事，它們使用的是邏輯位址而非真正的 selectors 和 offsets。因爲用來指出模組節區的 selectors 會隨著每次載入而不同。邏輯位址是從 1 起算的索引值，segment table 中第一個結構所代表的就是邏輯節區 1，第二個結構所代表的就是邏輯節區 2，依此類推。如果你觀察聯結器製作出來的 .MAP 檔，也許可以看到這樣的記錄：

```
0001:5F46       _free
0001:5F5C       __GetSubAllocClientData
0002:0030       _errno
0002:0032       __protected
```

一個模組最多可有 253 個節區，這是因爲 entry table（稍後描述）以邏輯位址形式儲存了輸出函式（exported functions）的位址，並只以一個位元組來儲存邏輯節區號碼。而邏輯節區 0、0FEh、0FFh 對 Windows 載入器有特殊意義，所以最大數目是 253 而非 256。

# Resource Table

除了節區資訊，每一個 module database 還包含所有資源（icons、bitmaps 等等）的位置和屬性。資源並未被視爲登記在 segment table 中的節區，所以你可以擁有超過 255 筆資源。

通常 resource table 緊跟在 segment table 之後，但它不是一個陣列。你必須大量計算，才能動態獲得一筆資源的位置。module database 中的資源格式十分接近 NE 檔中的 resource table。

第一個 WORD 是個 alignment shift count，關係到 sector 的大小，其意義和 NE 模組第 32h 欄位的 sector 大小意義相同，值也應該相同，否則就可能是什麼地方出現了錯誤。

第一個 WORD 之後是可變長度的 sections。每一個 section 放置某一特定型態的資源資訊。例如 USER.EXE 之中就有 cursors、icons、bitmaps、menus、dialogs、string tables、version info 等資源的 sections。每一個 section 之中是一個資料結構陣列，每一個結構關係到一份資源實體。如果你的 NE 檔有 5 個 icons，你就會有一個 icon section，內含 5 個結構。

每一個 section 緊跟在前一份資源之後。因此，爲了找出某份資源，你必須知道一個 section 有多大，而那又得視其中有多少資源而定。SHOW16 程式就示範了巡訪動作。每一種資源型態都以下面的結構開始（其對應的 C 語言定義請看 HMODULE.H）：

### 00h　WORD　resource ID

這個欄位是資源識別碼。最高位元（0x8000）如果設立，代表這是一個預先定義的資源。把最高位元遮罩之後，代表資源的型態：

```
1  - Cursor
2  - Bitmap
3  - Icon
4  - Menu
5  - Dialog
6  - String table
7  - Font directory
8  - Font
9  - Accelerator
10 - RC data (user-defined data)
11 - Error table
12 - Group cursor
13 - Unknown
14 - Group icon
15 - Name table (went away in Windows 3.1)
16 - Version info 請看 NE 格式規格書，以求做更詳盡的瞭解。
```

如果最高位元沒有設立，這就是一個使用者自定的資源，這時候 ID 就是資源型態名稱的偏移值（從 NE 模組中的 resource table 起算）。那是一個 pascal 字串（第一個字元用來表示字串長度）。

### 02h WORD number of resources of this type

某特定資源型態的實體個數。此欄位用於決定 resource type section 的長度，因爲代表各份資源的資訊就接續在本結構之後。

### 04h DWORD resource handler function

這個欄位內含資源的處理常式。處理常式很顯然是用於「必要時候把資源鎖定在記憶體中」。微軟不希望我們碰低階的資源處理，因爲那可能會帶來災難，所以相關文件非常稀少。SDK 中有關於 *SetResourceHandler* 和 *LoadProc* 的文件是僅見的微軟資料。

你可以改變某個模組的某種資源的處理常式。*SetResourceHandler* 可以辦到這一點。這個函式的文件上說它需要一個 HINSTANCE，而你手上只可能有 HMODULE。沒關係，就交給它 HMODULE。這是微軟把 HMODULE 和 HINSTANCE 混爲一談的又一個例子。後面還有好多這樣的例子呢！

緊跟在每一個 resource type section 之後的，是一個結構陣列。每一個結構（12 個位元組）關係到一份資源實體。結構個數已登記在 resource type section 的 02h 欄位中。每一個結構有以下格式（其對應的 C 語言定義請看 HMODULE.H）：

### 00h WORD offset in NE file

對於 NE 模組，這是資源的實際資料在 NE 檔中的偏移位置，單位是 sectors，不是 bytes。對於 PE 模組，這個欄位是某個 DWORD 的偏移位置（相對於全部 resource sections 的起始處）。該 DWORD 代表 PE 檔案中的 IMAGE_RESOURCE_DATA_ENTRY 結構位置（相對於 .rsrc section）。在 IMAGE_RESOURCE_DATA_ENTRY 結構中，你會找到資源實際資料在 PE image 中的位置和大小。請看第 8 章的 PE 檔案格式。如果 Windows 95 的 16 位元部份需要使用這份資料，它們必須知道 PE 檔的 .rsrc section 落在記憶體何處。如何決定 .rsrc section 的基底位址？簡單啦，請看 PE 檔的 module database 中的 NE 表頭第 48h 欄位。

### 02h　WORD　length

對於 NE 模組，此欄位是資源大小（單位爲 sectors）。對於 PE 模組，這是資源真正大小（單位爲 bytes）。這個欄位應該和 PE 檔的 .rsrc section 中的 IMAGE_RESOURCE_DATA_ENTRY.Size 吻合。

### 04h　WORD　flags

此一特定資源型態的旗標。一般而言，這些旗標和節區的旗標（Segment table 中的 04h 欄位）相同。然而 KRNL386 把某些額外位元設立起來了，而其意義未明。已知的幾個旗標位元意義如下：

| 旗標名稱和位元值 | 說明 |
|---|---|
| OADED 0x0004 | 此一資源目前被載入記憶體中。 |
| OVEABLE 0x0010 | 此節區在線性記憶體中是可移動的。如果此旗標未設立，節區固定不動（FIXED）。 |
| EADONLY　0x0020 | 這份資源沒辦法在記憶體中被更改。 |
| RELOAD 0x0040 | 當模組被載入，此節區即被載入，而非等用到了才載入。 |
| ISCARDABLE 0x1000 | 此節區可被拋棄。如果記憶體不足，KRNL386 可以將此節區的 descriptor 記錄爲 not-present（不存在），然後指定 RAM 做其他用途。 |

### 06h　WORD　ID

這是資源編譯器給予的資源 ID。如果高位元（0x8000）設立，表示以整數代表這份資源，否則它是一個有名稱的資源，於是此欄位就是資源名稱的偏移位置（相對於全部的 resource table 起始處）。資源名稱是個 pascal 字串（第一個字元用來表示字串長度）。典型以名稱表示的資源是對話盒，例如 .RC 中這樣的宣告：

```
Show16Dlg DIALOG 8, 18, 360, 280
```

**08h WORD handle**

如果資源已被載入，此欄位是其 global heap handle。如果資源尚未載入，此欄位是 0。這個欄位和資源旗標（04h 欄位）的 LOADED 旗標有關，如果 LOADED 旗標未設立，此欄位爲 0。

**0Ah WORD usage**

資源被使用的次數。呼叫 *LockResource* 可以增加其值，呼叫 *FreeResource* 可以降低其值。

# Entry Table

Entry table 記錄著本模組開放給其他模組使用的各個函式。在 Windows 真實模式的時代，entry table 也做爲 MOVEABLE 節區中所有遠程函式的 "central thunking location"。這裡，我將略而不談 entry table 的長像，只討論其角色的扮演。

和 segment table 或 resource table 不同，entry table 與其 NE 檔中的弟兄只有粗略相似。NE 檔中的 entry table 被最佳化以求空間節省，記憶體中的 entry table 則被最佳化以求快速掃描。和 resource table 一樣，entry table 最外層是以可變長度的區塊組成，需要動態計算才能在上面來回移動。

由於模組函式的輸出序號不必連續，也不需要從 1 開始，所以 entry table 可以由一束束的連續序號組成。要在其中尋找某個函式，你必須先找到適當的一束，然後才在其中尋找目標。每一束連續序號都以一個表頭（有著以下佈局）出現：

**00h WORD first export ordinal value in this bundle -1**

如果有一束序號從 3 到 14，此欄位爲 2（起始值減 1）。

**02h WORD last export ordinal value in this bundle**

如果有一束序號從 3 到 14，此欄位爲 14（最後一個序號）。利用這個欄位和上個欄位，我們就可以知道這一束序號究竟有多少個。本例爲 14-2=12 個序號。

表頭之後緊跟著便是一個結構陣列，每個結構對應一個輸出函式。結構佈局如下：

#### 00h BYTE segment type

如果此值爲 0FFh，表示它是一個 MOVEABLE 節區。如果此值爲 0FEh，表示此一輸出項比較特殊。這種項目沒有真正的遠程位址，其

offset 欄位（03h 欄位）被當做一個全域變數，用來聯結此輸出項。唯一知道的例子是 KRNL386 的輸出數值：__AHSHIFT 和 __0000H 等等。請看 ***Undocumented Windows*** 第５章，其中有此種輸出項的完整列表。如果此欄位既不是 0FFh 也不是 0FEh，那麼它就是輸出函式位址的邏輯節區號碼，這時候它應該和 02h 欄位吻合。

#### 01h BYTE flags

此一輸出項的旗標。下面是已知旗標：

| 旗標 | 說明 |
|---|---|
| 0x01 | 此函式爲輸出函式（exported function）。除非程式必須在真實模式下執行，否則此旗標應該總是設立。如果沒有設立，表示函式需要一個 real mode thunk，那就不應該開放給其他模組使用了。 |
| 0x02 | 此函式以一個共同的資料節區給所有呼叫者使用。這應該在 DLL 模組中發生。預設情況下此旗標在 EXE 模組中關閉，在 DLL 模組中設立。然而你也可以強迫這個旗標對 DLL 模組中的某個輸出函式關閉 -- 如果該 DLL 的 DEF 檔中的 EXPORTS 敘述句指定爲 NODATA 的話。 |

#### 02h BYTE logical segment number

此欄位內含輸出函式的位址的邏輯節區號碼。這個值應該被當做 segment table 的一個索引，用以決定節區的真正 selector。

#### 03h WORD offset

這是前一欄位所指定之節區中的偏移位置，代表輸出函式的起始位址。

爲了查詢輸出函式的位址（就像 KRNL386 所做所爲），你可以掃描所有「序號束」的表頭，找出內含此輸出位址的「序號束」。一旦找到，你就可以決定輸出函式的陣列索引。以前面所舉的 3~14「序號束」爲例，序號 7 之輸出函式應該在第 5 個陣列元素中。

在 entry table 中，沒有任何地方談到函式名稱。然而 *GetProcAddress* 允許你指定函式名稱。因此，一定有什麼方法讓函式名稱和函式序號產生關係，那就是下一節所介紹的主題啦。

# Resident/Nonresident Names Tables

這兩個表格是 NE 模組建立函式名稱和函式序號關係的依靠。它們有相同的格式。表格中每一項目都有以下佈局：

| 偏移（offset） | 說明 |
|---|---|
| 01 BYTE | 輸出函式的名稱長度 |
| ?? char | 輸出函式的名稱（沒有 null 結束字元） |
| ?? WORD | 輸出函式的序號 |

例如，GDI 的 *SetPixel* 函式序號爲 31，在 GDI 的 nonresident names table 某處將存在這樣的資料：

```
8, 'S', 'E', 'T', 'P', 'I', 'X', 'E', 'L', 31
```

Resident/nonresident names table 的第一筆資料有著特殊意義。其輸出序號皆爲 0。Resident names table 中第一筆資料是模組名稱（如 KERNEL、GDI、TOOLHELP 等等），這字串正是你在 .DEF 檔的 NAME 或 LIBRARY 指令處指定的名稱。

Nonresident names table 的第一筆資料是 description 欄位。那是你在 .DEF 的 DESCRIPTION 敘述句所指定的字串，做爲此一模組的簡短敘述。如果 .DEF 中沒有 DESCRIPTION 這一行，聯結器會以 EXE 或 DLL 的名稱代替。Windows 95 的

KRNL386 和 USER 和 GDI 模組的 description 字串如下：

```
KRNL386: 'Microsoft Windows Kernel Interface Version 4.00'
USER:    'Microsoft Windows User Interface'
GDI:     'Microsoft Windows Graphics Device Interface'
```

爲什麼要有兩個 names tables 呢？唯一的理由是爲了節省空間。大部份 EXEs 和 DLLs 是以序號而非名稱來輸入函式，因此 DLL 沒有理由放一大堆用不到的函式名稱在記憶體中。這些名稱應該放在 nonresident names table 中，只在必要時才載入（例如在 *GetProcAddress* 被呼叫時）。至於那些需要快速搜尋或不希望引起磁碟動作的函式，才放在 resident names table 中。

你可以利用 .DEF 控制「輸出函式（export function）」放到哪一個 names table 去。如果你輸出一個函式並在 .DEF 檔中明白給予其序號，它會被放到 nonresident names table。如果你沒有明白給予序號，它就會被放到 resident names table。你也可以在輸出函式那一行加上 RESIDENTNAME，聯結器就會把它放到 resident names table 中。一般而言，如果你有一個 DLL，開放出許多函式，你應該以 TDUMP 或 EXEHDR 之類的工具看看那些函式名稱放在哪個表格中。除非你有好理由，否則應該使用 nonresident names table。這樣你就不會被一大堆 DLL 函式名稱耗掉寶貴的記憶體了。

# HMODULEs v.s. HINSTANCEs

Win16 程式設計中最令人困擾的一件事就是 module handle（HMODULE）和 instance handle（HINSTANCE）的差別。我才剛剛表示過，HMODULE 代表一個被載入的 EXE 或 DLL。而下一節將說明，HINSTANCE 代表正在執行的 task 或 DLL 的 DGROUP 節區的 global heap handle。觀念上它們兩個是十分不同的。HMODULE 可以將你帶到一個被載入之可執行檔的豐富資訊去，HINSTANCE 卻只能給你 DGROUP 的內容。

前面我說令人困擾的事情是，許多 Win16 API 函式指定需要一個 HINSTANCE 時，其實它們真正要的是 HMODULE。以 *DialogBox* 爲例，它的第一個參數是 HINSTANCE，然而請你仔細想想它需要什麼。*DialogBox* 必須知道哪裡找得到它的對話盒資源，而資

源放在 EXEs 或 DLLs 中，所以 *DialogBox* 獲得 NE 檔的 HMODULE 才是比較合理的。把一個 HINSTANCE 交給 *DialogBox* 並不能幫助 *DialogBox* 找到對話盒資源。然而，或許你知道，把一個 HINSTANCE 交給 *DialogBox* 一樣可以正常運作。因此，檯面下一定有某些東西不爲人知。

未公開函式 *GetExePtr* 提供了一個瞭解 *DialogBox*（及其他函式）之所以能夠使用 HINSTANCE 的關鍵：

```
HMODULE GetExePtr ( HANDLE );
```

*GetExePtr* 是一個神奇的函式，它盡一切力量傳回 HMODULE 給你。如果你傳給它的是個 HINSTANCE，它會掃描所有的 DLLs 和所有的 tasks，尋找有沒有吻合的 HINSTANCE。如果找到了，*GetExePtr* 就傳回對應的 HMODULE。如果你傳給它的是個 HMODULE，它立刻傳回相同的 HMODULE。如果你觀察 *DialogBox* 函式動作，你會發現它呼叫 *GetExePtr*，然後再使用獲得的 HMODULE 找出對話盒資源。因此，不論你交給 *DialogBox* 的是 HINSTANCE 或 HMODULE，都可以正常運作。相同情況也發生在許多需要 HINSTANCE 的 API 函式上。

知道 HMODULE 和 HINSTANCE 的意義之後，我想你可以輕易回答這個常見的問題了：我要在我的 EXE 中產生一個對話盒，而對話盒資源放在 DLL 之中，那麼我應該把哪一個 HINSTANCE 交給 *DialogBox*？答案當然是 DLL 的 HINSTANCE（或 HMODULE）。如果微軟文件中能夠清楚說明參數的用途，類似的困惑就不會發生了。

或許你會驚訝爲什麼許多 API 函式的說明文件上都指定要 HINSTANCE 而其實它們需要的是一個 HMODULE 呢？我所知道的最好理由是，HINSTANCE 比 HMODULE 容易獲得。一般而言，EXE 或 DLL 不知道它的 HMODULE，直到它呼叫了 *GetModuleHandle*。但是 EXE 和 DLL 都會在執行時刻便獲得它們的 HINSTANCE。你也可以在主程式中取出 SS 暫存器值，那便是 HINSTANCE。當 EXE 開始執行，SS 暫存器和 DS 暫存器內容相同。雖然 DS 暫存器可能會隨程式的進行而改變，但 SS 暫存器不變。

在 Win32 程式設計中（Win32s 除外），HINSTANCE 和 HMODULE 是相同的東西，它們都代表 EXE 或 DLL 被載入的基底位址。

# 模組相關函式

現在你已經看過了 16 位元 module database，讓我們看看一些處理 module database 的函式。我選擇一組函式，並揭露其虛擬碼。有些函式（例如 *LoadModule*）沒有涵蓋進來的原因是它們太過複雜，而我希望在 20 世紀結束之前完成這本書。

## GetModuleHandle

當你檢驗與 module database 相關的函式，*GetModuleHandle* 應該是第一個選擇，因爲它可以展示大量的模組觀念，又不需要巨量的虛擬碼來表現。*GetModuleHandle* 接受一個模組名稱，傳回模組的 database segment 的 global heap handle（也就是 HMODULE）。技術文件上並沒有清楚定義什麼是模組名稱：是真實的模組名稱呢（在 .DEF 中指定）？還是模組的檔名？同時，如你在虛擬碼中所見，文件遺漏了其他一些出現在 *GetModuleHandle* 行爲中的東西。

*GetModuleHandle* 一開始先做參數檢驗工作，確定唯一的一個參數是個合法的字串指標。如果沒有通過，除錯版本將以 0x700A（ERROR_BAD_STRING_PTR）發出 RIPs。如果的確是個字串指標，就跳到 *IGetModuleHandle* 去。字串參數留在堆疊中。

*IGetModuleHandle* 的第一段有點令人驚訝。它先檢查模組名稱指標的 HIWORD 是否爲 0。如果是，就跳過一大段碼，把字串參數的偏移位置傳給 *GetExePtr*，並傳回 *GetExePtr* 的回返值（一個 HMODULE）。如何使模組名稱指標的 HIWORD 成爲 0，呵呵，那是未公開的秘密。你幾乎可以把與模組有關的任何一種 global handle 交給 *GetModuleHandle*，並獲得 HMODULE -- 只要記住把 HIWORD 設爲 0，並在 LOWORD 放置你的 handle。這些 handle 可以是 HINSTANCE、模組中的 code 節區或 data 節區、或 *GetExePtr* 能夠處理的任何其他 handles。

*IGetModuleHandle* 的主體用來搜尋 module list：以傳進來的模組名稱尋找某個 module database。它會以下列次序檢查三種可能性：

- 可能性之一：*GetModuleHandle* 獲得一個模組名稱，吻合 resident names table 的第一個項目。比對動作在 *FindExeInfo* 函式中發生。其虛擬碼也被我列出，十分單純，無需多做解釋。
- 可能性之二：*GetModuleHandle* 獲得一個模組名稱，吻合 resident names table 的第一個項目，但兩個字串的大小寫不一致。這和上一情況類似，*IGetModuleHandle* 會先把字串轉換爲大寫，再呼叫 *FindExeInfo*。
- 可能性之三：程式獲得一個檔案名稱。這也分兩種子情況：一個基本檔名（例如 KRNL386.EXE）或是一個完整路徑名稱（例如 C:\WINDOWS\SYSTEM\KRNL386.EXE）。*IGetModuleHandle* 在呼叫 *FindExeFile* 之前先把基本檔名萃取出來。*FindExeFile* 類似 *FindExeInfo*，但它比對的是檔案名稱而非模組名稱。

*IGetModuleHandle* 的最後一段碼隱藏了兩個未公開的秘密。Windows 3.1 有一個 DLL 稱爲 TIMER.DRV，這個 DRV 在 Windows 95 中已經消失了。假設某些程式呼叫 *GetModuleHandle*(TIMER.DRV) 檢查 TIMER.DRV 是否存在，微軟應該讓 *GetModuleHandle* 傳回 1（TRUE）。當然啦，若程式嘗試使用 TIMER.DRV 可就沒那麼幸運囉。第二個未公開的秘密是，module list 的第一個元素（KERNEL）會放在 DX 暫存器中一起傳回。

最後請你注意一點，不要使用 *GetModuleHandle* 尋找 Win32 模組的 16 bit module database。它們並沒有被安插到 HMODULEs 串列中。SHOW16 程式係使用暴力法才勉強找出那些 HMODULEs。

### GetModuleHandle 的反擬碼

```
// 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 的反擬碼

```
// 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 的虛擬碼（被 FarFindExeInfo 呼叫，使用相同的參數）**

```
// 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

*GetExePtr* 可以說是 Windows 95 之中最有用的 16 位元未公開函式。它的介面簡單，而且似乎不會改變。不知道爲什麼微軟要把這個令人驚嘆的函式隱藏起來。檢驗 *GetExePtr*，你可以獲得 modules、tasks、instances、global memory handles 之間的關係，基本上相當於對 16 位元 KERNEL 資料結構做了一次迷你旅程。

*GetExePtr* 的任務是根據傳進來的參數（一個 handle），傳回其對應的 HMODULE。通常 *GetExePtr* 是被 KRNL386 內部使用，將 HINSTANCEs 轉換爲 HMODULEs。任何需要 HINSTANCE 做爲參數的 Windows 函式，其內部幾乎都呼叫 *GetExePtr* 以取得 HMODULE。然而 *GetExePtr* 並不只限於處理 HINSTANCEs，幾乎任何型態的 global memory handle 都行。除了 HINSTANCEs，*GetExePtr* 也接受 HTASK。你也可以把模組中的一個 code selector 或 data selector 傳進去。你甚至可以傳給它一個以 *GlobalAlloc* 函式配置到的 handle，傳回的是擁有此塊記憶體之 task 的對應 HMODULE。簡單地說，*GetExePtr* 是個全功能函式，不會輕言放棄。

我還沒有正式說明 tasks 和 task databases（TDBs），它們在 *GetExePtr* 中佔有顯著地位。下一節我才會提到它們。

*GetExePtr* 一開始先把傳進來的 handle 轉換爲一個 selector。基本上這意味把 handle 的最底部位元設立起來。接下來 *GetExePtr* 檢查最好的一種可能：傳進來的是一個 HMODULE。只要檢查該節區的第一個 WORD 看看有無 "NE" 標記即可。如果測試成功，*GetExePtr* 任務完成。如果不成功，可就有些搜尋動作要做了。首先它看看參數是不是個 HINSTANCE，作法是檢查 task list，尋找有沒有吻合的值。如果有，*GetExePtr* 就傳回該 task 對應的 HMODULE。

如果傳進來的 handle 既非 HMODULE 也非 HINSTANCE，*GetExePtr* 呼叫輔助函式，對系統資料結構做更多的檢查。在虛擬碼中我把這輔助函式喚爲 *GetExePtrHelper*。它首先使用 CPU 的 LAR 指令，檢查輸進來的 handle 是否爲合法的 ring3 selector。如果不是合法值，立刻傳回 0。

檢驗成功之後，下一步是檢查誰擁有此 handle。大部份 global heap 區塊的擁有者不外乎 HMODULEs 或 PDB 節區（PDB 類似 DOS PSP）。例如 EXE 或 DLL 的 code 節區就是被一個 HMODULE 擁有。以 *GlobalAlloc* 配置，並指定 GMEM_SHARE 旗標的記憶體區塊則屬於配置時的 current task 的 PDB 擁有。知道 handle 的擁有者之後，*GetExePtrHelper* 檢查它是否爲一個 HMODULE，若是則大功告成。

如果擁有者不是 HMODULE，*GetExePtrHelper* 接下來又檢查它是否爲一個 HTASK，作法是搜尋 task database 中的 TD 標記。如果 handle 並非一個 HTASK，那麼其擁有者可能是一個 active task 的 PDB 節區。爲了檢查這種可能性，*GetExePtrHelper* 掃描 task list，取出每一個 task 的 PDB selector，拿來和輸入的 handle 比較。如果吻合，*GetExePtrHelper* 就傳回此 task 對應的 HMODULE。

當 *GetExePtrHelper* 回到 *GetExePtr*，如果傳回的是一個 HMODULE，*GetExePtr* 就原樣不動傳回給其呼叫者。否則 *GetExePtr* 傳回 0。這時候如果在 KRNL386 除錯版中，將有 RIP 錯誤訊息如下：

```
wn K16 GetExePtr(#ax) invalid parameter
```

其中的 #ax 將以 *GetExePtr* 的實際回返值取代。

自從 Windows 3.1 之後，*GetExePtr* 有了一些改變。在 Windows 3.1 之中，如果你傳給它的是一個 HTASK，*GetExePtr* 無法處理。但 Windows 95 版可以處理（前面你已看到了不是？）。這個改變說不定是因爲我在 ***Windows Internals*** 一書中的抱怨呢。

**GetExePtr 的虛擬碼**

```
// 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 的虛擬碼**

```
// 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

我在「模組相關函式」這一節中介紹 *GetProcAddress*，有三個理由。第一，這個函式提供非常好的例子，示範 entry table 和 resident/nonresident names table 的連接。第二，驗證 *GetProcAddress* 可以幫助你瞭解 Windows 載入器如何解決對其他模組的 fixups 問題（譯註）。第三，你可以利用 *GetProcAddress* 深度挖掘 Windows 的動態聯結機制，那是 Windows 作業系統最具威力的性質。

> 譯註：所謂 fixups，是指聯結器無法對程式所呼叫的 DLLs 函式指定實際位址（因爲是動態聯結而非靜態聯結），所以留下一個「待修正記錄」（fixup records），提喚載入器在載入時刻把它修補爲正確的函式位址，以利動態聯結順利完成。

*GetProcAddress* 和其他許多 API 函式一樣，先有一小段碼檢驗參數的正確性。它必須確定你傳給它一個合法的 selector（或 0 或 -1），做爲 HINSTANCE 參數。至於第二個參數（函式名稱）必須是合法的 LPSTR，或是一個 MAKEINTATOM 型態的字串：HIWORD 是 0，LOWORD 則爲非零值。如果你知道函式的輸出序號，可以把序號放在 LPSTR 的 LOWORD 中。只要有一個參數檢驗失敗，*GetProcAddress* 便立刻回返。如

果你使用 KRNL386 除錯版，你會得到 0x6002 或 0x700A 的 RIP 錯誤。如果兩個參數都沒問題，函式就跳到 *IGetProcAddress* 去，那才是真正的「本尊」。

*IGetProcAddress* 的第一個動作就是呼叫 *GetExePtr*，把你交過去的 HINSTANCE 轉換爲 HMODULE。如我先前所述，*GetExePtr* 有能力把許多種 handle 轉換爲 HMODULE，所以你不必受限只能傳遞 HINSTANCE。任何和模組扯上關係的 global handle 都可以派上用場。一旦 *IGetProcAddress* 知道欲搜尋之函式所屬的模組，它會檢查那是不是個 DLL。如果不是，*IGetProcAddress* 會以失敗收場。在 KRNL386 除錯版中，你會得到這樣的訊息：

```
Can not GetProcAddress a task.
```

爲什麼如此？因爲 Windows 不希望 EXEs 的函式被其他程式呼叫。只有 EXE 本身才能夠呼叫自己的函式，並且屆時的堆疊暫存器（SS）指向程式的 DGROUP。但是如果 *GetProcAddress* 的第一個參數爲 0，*IGetProcAddress* 會使用呼叫者（一個 task）所關聯的 HMODULE。換句話說，*GetProcAddress* 可以找到你自己這個 EXE 的函式，或所有的 DLLs 函式，但不能夠找其他 EXEs 的函式。

接下來 *IGetProcAddress* 必須計算出目標函式在 entry table 的序號。如果你傳遞過去的 LPSTR 參數的 HIWORD 爲 0，表示序號放在 LOWORD，那麼 *IGetProcAddress* 可以直接查詢模組的 entry table 中的輸出序號。但通常，*GetProcAddress* 接受的是一個 ASCII 字串，因此它必須把字串轉換爲適當的序號。

把 LPSTR 轉換爲序號是 *FarFindOrdinal* 的工作。我沒有提供 *FarFindOrdinal* 的虛擬碼，因爲不難瞭解它的動作。*FarFindOrdinal* 呼叫 *FindOrdinal*，掃描 resident names table 和 nonresident names table，一一比對其中每一個字串。如果吻合，序號（WORD）就緊鄰在字串之後。另一個未公開的 *GetProcAddress* 用法是，傳遞像 "#97" 這樣的字串。這時候 *FindOrdinal* 會把 '#' 拿掉，直接把剩餘字串轉換爲數值 97。

於是，我們發現一共有三種不同的表示法可以獲得相同的結果。假設你想要 *GetMessage* 函式位址（USER.EXE 的 #108 輸出函式），下面這三行都適用：

```
GetProcAddress( GetModuleHandle("USER"), "GetMessage" );
GetProcAddress( GetModuleHandle("USER"), MAKELONG(108, 0) );
GetProcAddress( GetModuleHandle("USER"), "#108" );
```

有了 HMODULE 以及函式序號之後，*GetProcAddress* 掃描 entry table，尋找此一序號所對應的資料項。這個資料項內含的資訊可以幫助我們計算函式位址。掃描 entry table 並取回函式位址的工作由 *FarEntProcAddress* 負責，而它其實只是 *EntProcAddress* 的外包裝而已。

在 *IGetProcAddress* 虛擬碼之後，你會看到 *EntProcAddress* 的虛擬碼。*EntProcAddress* 掃描「entry table 束」所組成的串列。所謂「entry table 束」其實是「一組 entry table 元素」，有連續的輸出序號。*EntProcAddress* 掃描每一「束」看看有沒有它要尋找的序號。如果找到，它就做出一個指標，指向該「束」中的 entry table 資料項。這筆資料項將被用來計算函式位址。

回到「Entry Table」一節看看 entry table 資料項的格式，你會看到它內含一個「函式偏移位置（offset）」，但是沒有 selector。不過它記錄了一個邏輯節區號碼。因此，*EntProcAddress* 必須把邏輯節區號碼轉換爲該節區的 selector。

怎麼做？簡單，每一個 module database 都有一個 segment table 陣列，邏輯節區號碼被視爲該陣列的一個索引，於是我們可以輕易從「挑出的 segment table 資料項的最後一個 WORD」得知 selector。剩下的工作就是把 selector 和 offset 組合成一個遠程指標。*EntProcAddress* 把遠程指標傳給 *FarEntProcAddress*，後者再傳回給 *IGetProcAddress*，最後再傳回給 *GetProcAddress*。

**GetProcAddress 的虛擬碼**

```
// 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 的反汇编**

```
// 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 的虛擬碼**

```
// 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-bits Tasks

如果 modules 被想像爲一個沒有生命的軀殼，task 可以想像是帶給軀殼生命的火花。在描述 task 如何掌管生命之前，我必須先釐清一些術語。Tasks 有時候被稱爲 programs，但在 16 位元 Windows 中正確的稱謂是 tasks，不是 programs。Win32 以 process 取代了 task 這一名詞。16-bit tasks 的意義相當於 32-bit process。在這一節中，我將在 Windows 95 的 16 位元端繼續使用 tasks 一詞。

Tasks 描述兩件事情。第一，task 描述程式碼的執行。在 Windows 3.x 或更早，task 就是排程單位，任何時候只有一個 task 正在執行。Task 所描述的第二件事情是擁有權。每一個 task 擁有自己一組 file handles、所產生的視窗、所配置的記憶體...等等。稍後我還會回來說這個主題。

每次 Windows 95 啓動一個程式，KRNL386 就產生一個新的 task。如果你執行兩個 CALC.EXE，Windows 95 把兩個 tasks 加到 KRNL386 的 tasks list 之中。甚至對於 Win32 processes，Windows 95 也產生一個 16-bit task 代表物。這或許可以讓 Windows 16 位元端感到高興，因爲 Win32 process 有了一種舊的 16 位元碼所認識的形象。

Task 是否存在，主要是看所謂的 Task Database（或稱 TDB）的資料結構是否存在。TDB 內含某個程式的某個執行個體（instance）的相關資訊。個別的 TDB 欄位描述於下一節，本節焦點放在一般性的、與 task 有關的觀念。

TDB 是一系列欄位，自成一個節區，來自 16-bit global heap。這個節區的 handle 稱爲 HTASK。我們知道，一個 HTASK 其實不過就是一個 selector。你可以直接讀取 TDB 的內容。這時候一個 task database 和其 HTASK 有點類似 module database 和其 HMODULE。*GetCurrentTask* 傳回一個 HTASK，你可以把它交給像 *PostAppMessage* 或 *EnumTaskWindows* 這樣的函式。

某一方面來說，Windows task 類似一個 DOS 程式。DOS 的每一個執行中的程式有它自己的 Program Segment Prefix（PSP）區域，內含一個 file handle table，以及其他資料，像是程式的命令列等等。由於 Windows 原本是 DOS 的一個延伸，所以 Windows task 總是在其 HTASK 節區中支援一個 PSP。在 95 版之前，Windows 執行真實模式下的 DOS 動作（如檔案 I/O）時，真的用到這個 PSP。以 Windows 的說法，TDB 中的 PSP 稱爲 PDB（Process Database）。別把它和 Win32 的 process database 搞混了。傳回 current task's PSP 的函式稱爲 *GetCurrentPDB* 而不是 *GetCurrentPSP*。Windows TDB 混合了「真實模式的 DOS 資料」和一些「只對 16 位元保護模式有意義的資料」。此外，Windows 95 的 TDB 還內含一個指標，指向 Win32 Thread database。所以 Windows 95 的 Task Database 事實上是 DOS、Win16、Win32 資訊的整合。

就像一個軀體不能沒有生命，一個 task 不能沒有 module 而單獨存在。每一個 task 都和一個 module 有關聯，反之不一定成立。當你第一次執行起一個程式，Windows 95 產生一個 16-bit module database 然後再爲新的 task 產生一個 task database。如果你再執行一個副本（原來的那個持續在跑），Windows 95 會產生另一個 task database，但不會再產生新的 module database。兩個 tasks 都關聯到同一個 module database（HMODULE）。Modules 表現的項目有程式碼、資源、以及其他可被共用的東西。Tasks 表現的則是個別程式副本的東西，例如堆疊、目前磁碟目錄等等。

此刻請你暫時忘掉 32-bit processes 和 threads，任何時候 Windows 95 只執行一個 task。所有其他的 threads 都被凍住，直到這個正在執行的 task 自動放棄對 CPU 的控制。這就是所謂的合作型多工（cooperative multitasking）。每一個 task 可以盡情執行，當它放棄控制權時其他 tasks 才能接棒。

Task 如何放棄控制權？通常它是利用 *GetMessage*、*PeekMessage*、*SendMessage* 或 *WaitMessage*。如果那些函式認爲此一 task 沒有必要繼續執行（例如沒有訊息等待被處理的話），它們便呼叫 16 位元排程器，爲其他有事等著做的 task 服務。大部份時候釋放控制權這事情隱藏在程式員的視線之外，因爲像 *GetMessage* 之類的函式會透明化地處理合作型多工。

Windows 95 持續追蹤 task list，類似追蹤 16-bit module database list 那樣。TDB 中有一個 WORD，內含串列內的下一個 task 的 selector。Tasks list 並不像 modules list 那般靜態，它的次序常會改變以幫助排程器做決定。有趣的是，用來表達「Win32 processes in 16-bit land」的那些 TDBs，似乎不會因 16 位元排程器的作用而移動其在 task list 中的位置。那些爲 Win32 processes 而產生的 TDBs 似乎固定在串列的頭部，不會移動，直到其 task/process 結束。請看稍後的「Task Database（TDB）」一節中對於 08h 欄位的描述。

如果要走訪整個 task list，微軟希望你採用 TOOLHELP 的 *TaskFirst* 和 *TaskNext* 函式。如果你要直接探詢它，像 SHOW16 那樣，你可以在呼叫 *GetCurrentTask* 之後從 DX 暫存器獲得串列的頭。然後你可以找出串列中的第一個 TDB，作法是利用 *GetProcAddress* 獲得 THHOOK 函式位址，加上 0xE，然後讀取該位置上的 WORD。THHOOK 內含其他數個有用的 KRNL386 全域變數：

**THHOOK+0　　hGlobalHeap**

這是 Burgermaster 資料結構的 handle（selector-1）。該資料結構用以維護 16-bit global heap。***Windows Internals*** 第 2 章對此有詳細說明。此值可以從 KRNL386 的未公開函式 *GlobalMasterHandle* 回返時的 AX 暫存器中獲得。

### THHOOK+2　　pGlobalHeap

這是 Burgermaster 資料結構的 selector，基本上和 hGlobalHeap 欄位的作用差不多。此值可從 *GlobalMasterHandle* 回返時的 DX 暫存器中獲得。

### THHOOK+4　　hExeHead

內含 16-bit module list 中的第一個 module 的 HMODULE。第一個 module 總是 KERNEL（KRNL386.EXE）。此值可以從 *GetModuleHandle* 回返時的 DX 暫存器中獲得。請看 module database 的 06h 欄位以獲得更多資訊。

### THHOOK+8　　topPDB

這是 KRNL386.EXE 的 PSP（或說 PDB）的 selector。這是 KRNL386 從真實模式 DOS 中被載入（並被視爲一個 DOS 可執行檔）的 PSP。此值可從 *GetCurrentPDB* 回返時的 DX 暫存器中獲得。

### THHOOK+0Ah　　headPDB

這是 PDBs 串列中的第一個 PDB/PSP 的 selector。

### THHOOK+0Eh　　headTDB

指向 TDBs 串列中的第一個 TDB。此值可從 *GetCurrentTask* 回返時的 DX 暫存器中獲得。

### THHOOK+10h　　CurTDB

指向 TDBs 串列中的 current TDB。它也幾乎總是串列中的最後一個元件。此值可從 *GetCurrentTask* 回返時的 AX 暫存器中獲得。

### THHOOK+12h　　LoadTDB

這欄位通常是 0，除非有一個新的 task 要產生。這時候這個欄位內含 new task 的 TDB selector。

**THHOOK+16h SelTableLen**

這是 Burgermaster 節區中的 selector table（DWORDs 陣列）的長度。請看 ***Windows Internals*** 第 2 章以知詳情。

**THHOOK+18h SelTableStart**

這是 Burgermaster 節區中的 selector table（DWORDs 陣列）的起始位置。請看 ***Windows Internals*** 第 2 章以知詳情。

和 module databases 不一樣，Windows 95 能夠把所有的 task databases（不管 16- 或 32-位元程式的）集中在一個 tasks list 中管理。因此你可以利用 16 位元 TOOLHELP.DLL 所提供的 *TaskFirst* 和 *TaskNext* 把所有執行中的程式走訪一遍。同時，和 16 bit module 不同的是，所有 tasks 在 Windows 95 的 32 位元端都有一個對應的表象。此外，每一個 Windows 95 task database 都內含一個 flat 線性指標，指向 thread database。甚至 16 位元程式亦然。讓我最後做個摘要：每一個程式（NE 程式或 PE 程式）都有一個 16-bit task database 和一個 32-bit thread database。

# 關於 Tasks 的一些常見錯誤觀念

Tasks 有時候不太容易理解，難怪程式員對它常會有些錯誤觀念。這一節要描述並澄清我常遇到的一些錯誤觀念。

最常見的一個有關於 task 的錯誤觀念是，每一個 task 有一個視窗。雖然視窗可以表現一個 task 的生命跡象，但 task 和視窗完全沒有必要關聯，更不應該混爲一談。Task 表現出來的意義就是「執行現象」，僅此而已。它要不要有視窗，完全由程式員決定。我們很容易產生出沒有視窗但仍然有用途的 tasks -- 當你觀察 Windows 95 的「task list」（譯註），這一點要牢記在心。該程式列出最上層視窗，它們和 KRNL386 所維護的 task list 風馬牛不相及。如果你執行本章的 SHOW16 程式，你就會看到未曾出現在 Explorer（中文名爲「檔案總管」）上的一些 tasks。

譯註：這裡所說的 "task list" 是個程式，就是 \WIN95\TASKMAN.EXE。畫面如下：


另一個常見的錯誤觀念是，把 DLL 想像爲一個有著 task 性質的東西。他們常說的話是：『我希望我的 DLL 產生一個視窗，讓所有的呼叫者都能享用』，或是說：『我希望我的 DLL 產生一個 file handle，讓所有的呼叫者都能享用』。這些說法表示程式員以爲 DLL 擁有它所產生的視窗或檔案。但其實是 task 而非 DLL 擁有它們。DLL 只不過是 task 所使用的一堆程式碼。雖然 DLL 有獨立的檔案，但它們並不能夠擁有系統資源。

一個 DLL 產生視窗或開啓檔案，都是爲了 task 而做。DLL 沒有權力擁有它們。因此如果你在 DLL 中呼叫 *CreateWindow*，產生出來的視窗由 current task 擁有。如果該 task 結束了，視窗也必隨之帶走 -- 即使 DLL 還留在記憶體中。同樣的道理適用於檔案身上。如果 DLL 爲第一個 task 開啓一個檔案，然後第二個 task 呼叫此 DLL 並使用該檔案，便會發生錯誤（更糟的情況是存取到另一個檔案）。爲什麼？因爲該檔案屬於第一個 task 擁有。

# Task Database（TDB）

前數節已經說明了 task 的一些一般觀念，這一節我要詳細列出 TDB 中的每一個欄位並解釋之。如果你只想快速瀏覽 TDB 的結構，請看 SHOW16 程式的 TDB.H 檔。和以前的習慣相同，我所列出的每一行的三項資料分別是欄位偏移位置、欄位型態、簡短說明。

### 00h WORD next TDB

這是 Win16 tasks list 內的下一個 task 的 HTASK。串列頭被記錄在 KRNL386 的全域變數 HeadTDB 中，此外你也可以從 *GetCurrentTask* 回返時的 DX 暫存器中獲得。串列以 0 做爲結束標記。

### 02h DWORD task SS:SP

當 task「停泊」在 16 位元排程器時，這個 DWORD 內含其 SS:SP。從這裡開始算起的一個固定位置處，放有 CPU 暫存器值 -- 當這個 task 再次獲得 CPU 控制權，那些暫存器值就會重新被使用。事實上 TOOLHELP 的 *TaskSwitch* 和 *TaskSetCSIP* 完全依賴這組資料才能遂其神妙。這個欄位對於執行中的 task 沒有意義，因爲執行中的 task 並未「停泊」在 16 位元排程器內。

### 06h WORD number of events

這是等著被 task 處理的事件（events）個數。通常 Windows 程式設計書籍中不會使用 event 一詞，因爲 event 會帶來訊息（message）。如果你 "post" 一個訊息到某個視窗，訊息會被存入該 task 的訊息佇列中，而此欄位會累加 1。然而，event 畢竟不是 message（訊息）的同義字，一個 task 可能有 events 等待處理，卻沒有訊息等著被處理。Events 是量測指標，16 位元排程器用以決定是否要喚醒某個 task 繼續執行。排程器通常只對那種有 events 等待被處理的 tasks 感興趣。

### 08h BYTE priority

這個欄位內含 task 的相對排程優先權。然而此值似乎未真正被使用。理論上，此值可爲 -32~15，由 KRNL386.EXE 的未公開函式 *SetPriority* 設定。KRNL386 把 tasks 保持在

「以優先權排序」的狀態。數值低者在前。由於 Win16 排程器演算法之故，較低的優先權值，較早被檢查「是否有等待中的 events 要處理」。然而，調整你的優先權通常不會給你帶來什麼東西，因爲排程器只青睞那些「有 events 等待被處理」的 task。你可以把你的 task 優先權設爲 -32，但如果它沒有任何 events 等待被處理，還是沒輒兒。

#### 09h　BYTE　unused fields

此欄位未被使用。

#### 0Ah,0Eh,10h,12h　WORD　unused fields

這些欄位可能是在 OS/2 1.x 時使用，當時 OS/2 1.x 和 Windows 共用不少碼。到了 Windows 3.x 和 Windows 95，這些欄位統統爲 0。

#### 0Ch　WORD　This TDB

此欄位指向自己這個 TDB。

#### 14h　WORD　floating-point control word

在 Windows 3.x 中，這個 WORD 內含 task 被切換時的浮點運算控制字組。所謂「浮點運算控制字組」內含 80x87 狀態旗標，被 FLDCW 和 FSTCW CPU 指令儲存和回復。在 Windows 95 中此欄位沒有用處，或許是因爲 Windows 95 的 task switch 也引發了 thread switch，而浮點運算控制字組的儲存和回復可以在 ring0 thread switch 層面完成。

#### 16h　WORD　task flags

這個欄位內含以下旗標：

| 旗標名稱與數值 | 說明 |
|---|---|
| TDBF_WIN32<br>0x0010h | 如果設立，表示此 task 爲 Win32 程式。在 Win32s 環境中執行的 Win32 程式也會設立此旗標。 |
| TDBF_NEWTASK<br>0x0008 | 當 Win16 task 產生，此旗標設立。當這個 task 第一次進入 16 位元排程器（Reschedule 函式），此旗標被清除。 |

| | |
|---|---|
| TDBF_WINOLDAP 0x0001 | 表示這個 task 是 WINOA386.MOD（模組名稱爲 WINOLDAP）。WINOLDAP 模組用來在 Windows 95 的某個虛擬機器中執行 DOS 程式。WINOLDAP 有點像是 DOS prompt 的一個外包器，在 task list 中，你看到的是 WINOLDAP，而不是 DOS 程式的名稱。 |

### 18h WORD error mode

這個欄位內含一組旗標值，用來訂定 Windows 95 對於某些錯誤的回應。這些旗標可以 *SetErrorMode* 函式設定之。公開說明的旗標意義如下：

| 旗標名稱與數值 | 說明 |
|---|---|
| SEM_FAILCRITICALERRORS 0x0001 | 當 DOS 函式呼叫失敗（遭遇一個嚴重錯誤，原本會以 "Abort, Retry, Ignore?" 回應）時，不要聲張。如果這個旗標沒有設立，Windows 95 會出現一個對話盒，要求指導。 |
| SEM_NOGPFAULTERRORBOX 0x0002 | 當 GP fault 發生，不要出現一般的 GP fault 對話盒。在 Windows 3.1 中，此旗標主要給除錯器使用，用來默默結束被除錯程式。除錯器爲被除錯程式設立此旗標，然後修改被除錯程式，使它一旦再次執行起來就產生 GP fault，於是 Windows 默默將之結束。請參考 TOOLHELP 的 erminateApp 函式說明，該函式可以選擇性地設立這個旗標。 |
| SEM_NOOPENFILEERRORBOX 0x8000 | 當檔案找不到時，不顯示對話盒。當你希望 LoadLibrary 失敗時不要聲張（不要出現 File Not Found 對話盒），就應該將此旗標設立。 |

### 1Ah WORD expected Windows version

此欄位表示能夠執行此程式之 Windows 最小版本。它和 module database 的 0x3E 欄位相同。

### 1Ch WORD HINSTANCE of this task

此欄位內含 task 之 HINSTANCE。HINSTANCE 其實就是一個 global heap handle，指向 task 的 DGROUP（default data segment）。這個 HINSTANCE 被當做 *WinMain* 的第一

個參數。DGROUP 節區內含堆疊。每一個 task 有它自己的 HINSTANCE，這些 HINSTANCEs 常被用以區分不同的程式(雖然 TDB 也相當適合這個用途)。Win32 tasks 的 HINSTANCE 和其 HMODULE（1Eh 欄位）相同。

#### 1Eh　WORD　module handle of this task

此欄位內含被載入之可執行檔的 HMODULE -- task 就是從該可執行檔產生出來的。此值可以交給 *GetModuleFileName* 以取得對應的 EXE 檔案名稱。

#### 20h　WORD　message queue

這個欄位內含一個 selector，指向 task 的訊息佇列。和前一版本不同的是，Windows 95 的訊息佇列沒有固定大小。第4章曾對此有細部說明。

#### 22h　WORD　parent TDB

這個欄位內含一個 TDB selector，代表「以 *WinExec* 將此 task 執行起來」的 task。如果你正在對一個程式除錯，那麼 parent TDB 就是指除錯器。如果你在 Explorer（檔案總管）中啓動一個程式，其父行程即爲 EXPLORER.EXE。如果你在 DOS 視窗中啓動一個程式，其父行程即爲 MSGSRV32.EXE。對 Win32 程式而言，parent TDB 總是 0。

#### 24h　WORD　application signal action

在 Windows 3.1 中，此值會影響 task 的「application signal procedure」的行爲，但真正的意義不明確。在 Windows 95 中，「application signal procedure」的位址（26h 欄位）似乎沒有用。

#### 26h　DWORD　Windows 3.1 application signal procedure

在 Windows 3.1 中此欄爲「application signal procedure」的指標。程式可藉由「application signal procedure」在 Ctrl-Break 被按下時獲得回呼。它是經由未公開函式 *SetSigHandler* 函式設定的。在 Windows 95 中該函式不再存在，而此欄位總是爲 0。

### 2Ah DWORD USER signal procedure

這個欄位有一個指標，指向 USER 的 signal procedure。當 DLL 被載入或被踢出記憶體，USER 的 signal procedure 就會被呼叫。這使得 USER 有機會清除未釋放的系統資源。在 DLL 退出記憶體時所呼叫的 callback 過程中，USER 也呼叫 GDI 的 *SignalProc* 函式，給 GDI 一個清除未釋放之 GDI 資源的機會。

你可以藉由呼叫未公開的 *SetTaskSignalProc*（KERNEL.38）函式，改變 TDB 中的 signal procedure。該函式的原型是：

```
FARPROC SetTaskSignalProc ( HTASK hTask, FARPROC lpfnNewSignalProc );
```

傳回的是原來的 signal procedure 位址。

USER 的 signal procedure callback 函式如下：

```
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 以自己的處理常式取代了 USER signal procedure。在該處理常式中，TOOLHELP 把所有已安裝之中斷服務常式或通告處理常式（notification handler）都「解除掛勾（unhooked）」。在 Windows 95 中，TOOLHELP 不再和 signal procedure 混雜相處，而是使用新的 DLLENTRYPOINT 機制（描述於 Windows 95 的 thunk compiler 文件中）。

### 2Eh DWORD GlobalNotify callback（譯註：原文爲 2Ch，是錯誤的）

此欄位內含一個指標，指向 task 的 *GlobalNotify* callback procedure。當 KRNL3866 即將

拋棄一個 DISCARDABLE 區塊時，它會呼叫此一函式。這個函式可以阻止拋棄工作的順利進行。當 task 新誕生，此欄位初始化爲 0。

### 32h　DWORD[7]　task interrupt handlers（INTs 0,2,4,6,7,3Eh,75h）

（譯註：原文的 offset 爲 **30h**，是錯誤的）

對於大部份中斷（interrupts），Windows 95 有一個全域性處理常式供所有 tasks 使用。然而它也允許 tasks 安裝它們自己的處理常式（經由 INT 21h 的第 25h 號功能）。當那樣的一個中斷發生，Windows 95 會尋找 current task 之 TDB 中的中斷服務常式，然後呼叫之。在 TDB 中則是一個 DWORDs 陣列，每一個 DWORD 持有一個中斷服務常式的位址。「每個 task 自己有一個處理常式」的中斷包括：

```
0   - Divide by Zero
2   - NMI
4   - INTO
6   - Invalid Opcode
7   - Coprocessor Not Available
3Eh - 80x87 emulator
75h - 80x87 error
```

CALC.EXE 是 task 改變中斷向量的一個好例子。SHOW16 可以讓你觀察哪一個 task 安裝了哪些中斷服務常式。

### 4Eh　DWORD　compatibility flags

這個欄位在 Windows 3.1 時代引入，告訴 Windows 保留前一版行爲，給依賴那些行爲的程式運用。當 Windows 發現它執行的是這樣一個程式，就檢查這些旗標值，然後適當地調整自己。如果你觀察 WIN.INI 中的[Compatibility] 區，你會看到需要這種相容性旗標的各個模組名稱（譯註）。令人驚訝的是，其中許多程式是微軟的產品。***Undocumented Windows*** 第 5 章有一張列表，列出每一旗標在 Windows 3.1 的意義。Windows 95 又增加了一些旗標。你可以利用未公開的 *GetAppCompatFlags* 函式取得某個 task 的相容性旗標：

```
DWORD FAR PASCAL GetAppCompatFlags(HTASK hTask);
```

譯註：以下便是 Windows 95 的 WIN.INI [Compatibility] 一瞥：

```
[Compatibility]
AMIPRO=0x04800010
CASMONEY=0x00200000
CAVOIDE=0x00200000
CCMAIL=0x00200000
COSTAR=0x0004
DIRECTOR=0x00800000
EXCEL=0x1000
FREEHAND=0x8000
...

[Compatibility32]
CLWORKS=0x00A00000
MCAD=0x00600000
MT=0x00400000
PHOTOSHP=0x00208000
VISIO=0x00000000
...
```

### 52h WORD TIB selector

這是 Win32 threading code 用來處理 TIB（thread information block）結構的 FS 暫存器值。所有的 tasks（甚至 16-bit task）都維護有 Win32 processes 和 threads。用以「存取 task 的 Win32 thread information」的指標和 selectors，都有一個副本放置在每個 task 的 TDB 節區中。

Thread Information Block（TIB）內含每一執行緒專屬的一份資訊，包括：

```
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.
                         //譯註：我懷疑 Matt Pietrek筆誤。這裡是否應爲 28h？
```

TIB 結構從 thread database 的 0x10 位元組開始。Thread database 的 32 位元線性位址放在 TDB 的 54h 欄位中。關於 TIB，第 3 章有更多資訊。

### 54h　DWORD　linear address of the task's THREAD_DATABASE

此欄位是線性位址，指向與此 task 關聯的 ring3 thread database。這個 thread database 包含 Thread Information Block（TIB，請看 52h 欄位），以及在 TIB 之前的 10h 個位元組。關於 THREAD_DATABASE，請參考第3章。

### 58h　WORD DGROUP　handle of task

對於 16-bit tasks 而言，這個 WORD 是其 DGROUP 節區的 global heap handle。根據 KRNL386 除錯版的錯誤訊息顯示，此欄位可以在 16-/32- 位元移轉（thunking）過程中用來獲得 task 的 atom table 所在節區（也就是 DGROUP 節區）的 handle。對於 Win32 tasks，此欄位總是 0。

### 5Ah　BYTE[6]　unused

這 6 個位元組在 Windows 95 中不起作用。

### 60h　WORD　PDB of task

這是 PDB（PSP）節區的 selector。PDB 內含 task 的 file handle table、命令列（command line）、以及在各式 DOS 程式設計書籍中曾提過的其他欄位。每一個 Win16 tasks 的 PDB 儲存在 HTASK 所指之記憶體尾端；很特別的一點是，其 PDB selector 的基底位址總是比 HTASK selector 的基底高出 0x210。至於 Win32 tasks，PDB 總是在 1MB 線性位址之下 -- 雖然 TDB 節區總是在 2GB 之上。

### 62h　DWORD　DOS Disk Transfer Area

這個 DWORD 指向 MS-DOS Disk Transfer Area（DTA）。請參考 DOS 程式設計書籍以獲得 DTA 的詳細資料。對於 Win16 tasks，DTA 爲 PDB 節區中的 80h 個位元組。也就是說，此欄位的 selector 部份和上一欄位（60h 欄位）的內容應該相符。至於 Win32 tasks，所有的 Win32 tasks 共享同一個 DTA。

### 66h BYTE current drive

此值內含 task 目前目錄中的磁碟機代號。它偏移 0x80，所以你可以將它減去 0x80，便可獲得磁碟機代號（0x80=磁碟機 A，0x81=磁碟機 B，依此類推）。在 Windows 3.x 中，路徑的目錄部份即儲存在此欄位之後，但在 Windows 95 中目錄被移到 0x100 欄位處。

### 67h char[65] unused

在 Windows 3.1 中此區域放置 task 目前工作目錄的路徑。路徑名稱最大爲 65 字元。Windows 95 之中由於長檔名來臨，這個區域顯然太小，所以路徑名稱被放到 0x100 欄位處。

### A8h WORD initial task validity check

在 Windows 3.1 中此欄位放的是程式啓動後的 AX 值。然而由於似乎沒有任何函式會檢查此值，所以它似乎是個無用的欄位。

### AAh WORD next task to schedule（DirectedYield）

此值如不爲 0，即代表 Win16 排程器即將喚醒之 HTASK。實際上此值總是 0，除非你呼叫 *DirectedYield* 指定某個 task 接續在後執行。*DirectedYield* 會把其參數儲存到此欄位中，然後呼叫 Win16 排程器（Reschedule 函式）。Reschedule 函式一開始會檢查此欄位，如果不是 null，就跳過排程器正規的「下一個 task」搜尋程序。Reschedule 函式會把此欄位重設爲 0，所以你很難得觀察到這個欄位不是 0。

### ACh DWORD selector:offset to list of DLLs to initialize

程式啓動時，此欄位內含一個指標，指向一個以 0 爲結束記號的「DLL module handle 陣列」。其中記錄的所有 DLLs 都將初次被載入記憶體，所以它們的 *LibMain* 都會被呼叫。如果 implicitly linked DLL（譯註：以 import library 與 EXE 聯結在一起的那種 DLLs）早已在記憶體中，其 HMODULE 不會出現於此陣列。*InitTask* 會尋訪整個陣列，呼叫對應的 *LibMain* 函式。然後 *InitTask* 把陣列所佔的記憶體釋放掉，把此欄位設爲 0。注意：欄位中的遠程指標有正常的格式，selector 放在低字組而 offset 放在高字組。

### B0h　WORD　code segment alias for this TDB

Windows 最初把 *MakeProcInstance* thunks 生產於 TDB 本身之中（請看 BAh 欄位）。由於 CPU 沒辦法以一個 data selector 執行程式碼（而你知道，TDB 是一個 data selector），所以 KRNL386 產生一個 alias selector，基底位址以及節區長度都和 TDB 相同，唯一的不同是使用 code selector 而非 data selector。你所產生的最初 7 個 *MakeProcInstance* thunks 都有一個與此欄位相同的 selector 值。

### B2h　WORD　selector of segment with additional thunks

如果有超過 7 個 *MakeProcInstance* thunks 被產生，KRNL386 會配置另一塊 code 節區，安置另外 7 個 thunks。該節區的格式和 TDB 的 B0h~F1h 格式相同。如果另有超過 7 個 thunks 被產生，KRNL386 就再配置另一塊 code 節區安置之。這些節區會被放在串列的尾端，而此一欄位就像是個 "next" 指標。

### B4h　WORD　PT signature（5450h）

這個欄位內含 5450h，也就是 "PT" 的 ASCII 碼。 "PT" 意指 Procedure Thunks 或 ProcInstance Thunk。

### B6h　WORD　unused

此欄位沒有用處，總是爲 0。

### B8h　WORD　offset of next available thunk slot + 6

將此值減 6，你就可以獲得 TDB 的偏移位置 -- *MakeProcInstance* thunk 將在那裡被產生。

### BAh　char[0x38]　MakeProcAddress thunk area

這裡放置 7 個 *MakeProcAddress* thunks。每一個 thunk 有 8 位元組那麼長，內容如下：

```
MOV AX, hInstance       ; hInstance == parameter 2 to MakeProcInstance
JMP FAR PTR lpfnProc    ; lpfnProc == parameter 1 to MakeProcInstance
```

#### F2h char[8] module name for task

內含 task 的模組名稱，拷貝自 module database。如果模組名稱是滿滿的 8 個字元，此欄位就沒有 NULL 結束字元。

#### FAh WORD TD signature

這個欄位內含 0x4454，也就是 "TD" 的 ASCII 碼（表示 Task Database）。*IsTask* 函式以及其他 KRNL386 函式將根據此一標記保證它們正在處理的是一個合法的 task database。

#### FCh DWORD unused

此欄位沒有用處，總是爲 0。

#### 100h char[110h] current directory of task

由於 Windows 95 支援長檔名，task 的目前工作磁碟目錄沒辦法塞到 67h 欄位處，所以其路徑（不含磁碟機代號）將放在這裡。

#### 210h char[110h] PDB/PSP of task（Win16 tasks only）

對 16 位元程式而言，此欄內含 task 的 PDB/PSP。這些區域也被 TDB 60h 欄位中的 selector 所指。它的長度 110h 個位元組，是有點奇怪，因爲在 Windows 95/DOS 7 之前，PSP 只需要 100h 個位元組。

## Task 相關函式

現在我們已經看過了 Windows 95 16-bit TDB 的長像，讓我們看看一些存取 TDB 結構的函式。我所選擇的是十分簡單的函式，主要是因爲像 Windows 核心排程器（Reschedule）那樣的函式，很容易就會佔掉一整章。

## GetCurrentTask

這是 task 相關函式中最基本的一個。此函式的回返值放在 AX，task list 的頭放在 DX。該兩個值其實都保持在 KRNL386 的全域變數中。因爲 KRNL386 的資料節區是 FIXED 並且 pagelocked，此函式所取回的兩個數值總是駐留於記憶體中。因此，在一個中斷服務常式中呼叫 *GetCurrentTask* 很是安全。這足以反駁微軟的警告，他們說在中斷服務常式中唯一能夠安全呼叫的只有 *PostMessage*。你打算相信誰呢？請看看證據然後自己做決定。

由於 *GetCurrentTask* 是如此簡單的一個函式，直接展示其組合語言碼我想會比顯示其 C 語言虛擬碼更清楚些：

**GetCurrentTask 函式碼**

```
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

這是一個很方便的函式，你可以用它來驗證你的確獲得一個合法的 task handle。驗證過程並不十分嚴格，只不過是檢查 FAh 欄位的值是否爲 0x4454（TD）。你隨時可以建構一個節區，輕鬆矇騙過關。

有趣的是，並沒有任何測試動作，確保這個 handle 是合法的 global memory handle。你或許可以想像，丟給它一個不合法的 selector，會引起 GP fault，於是 Windows 強制結束你的程式。當此事發生，的確會產生 GP fault，但 KRNL386 已經做好了準備。

爲了處理可能發生 GP fault 的程式碼，KRNL386 有一個表格，內含 GP fault 可能發生的位址範圍。和每一位址範圍有關聯的，是一個安全的復元位址。如果 KRNL386 的 GP fault handler 看到一個 GP fault 在那些位址範圍中發生，它會把控制權移轉到「復元位址」上。*IsTask* 的「復元位址」只是簡單放了個 0（也就是 FLASE）在 AX 暫存器中，然後回返至 *IsTask* 的呼叫者。這聽起來有點像 Win32 的結構化異常處理（structured exception handling），是的，基本上它是，雖然其間有些重大差異。如果想知道更多有關於 KRNL386 結構化異常處理的知識，請參考 ***Undocumented Windows*** 書中對 __GP 和 HasGPHandler 的介紹。

**GetCurrentTask 的底層碼**

```
// 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

*GetTaskQueue* 是一個未公開函式，它需要一個 HTASK 參數，傳回的則是與該 HTASK 關聯的訊息佇列的 handle。如果 HTASK 是 0，此函式傳回 current task 的訊息佇列。第 4 章曾對訊息佇列有較深刻的介紹。

*GetTaskQueue* 可以有效決定一個 task 是否能夠接收視窗訊息（因爲一個訊息佇列能夠接受被 posted 或被 sent 的訊息）。程式的訊息佇列一開始並不存在，直到 startup 碼呼叫了 *InitApp* 後才產生。呼叫 *InitApp* 的動作一直到 implicitly loaded DLLs（譯註：KERNEL、GDI、USER 都是）的 *LibMain* 被呼叫之後才發生，因此 task 生命中有相當份量的碼是在沒有訊息佇列的情況下執行的。Windows 除錯器特別需要知道它們所面對的程式是否有一個訊息佇列，因爲這會影響它們處理被除錯者及其視窗訊息的行爲 -- 在被除錯者結束之後。

*GetTaskQueue* 並不做嚴厲的參數檢驗。如果你放一個非零值做爲參數，而它不是一個合法的 selector，你會得到一個 GP fault。

**GetTaskQueue 的虛擬碼**

```
// 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.
```

**GetATaskSomehow 的虛擬碼**

```
// Parameters:
//  HTASK   hTask

    if ( hTask )        // If any nonzero hTask passed in, return it;
        return hTask;   // otherwise, return the current task.
    else
        return CurTDB;
```

## MakeProcInstance

雖然許多新的編譯器已經去除了 *MakeProcInstance* 的必要性，但是如果你的 callback 函式位在 EXE 檔而非 DLL 檔，它還是需要的。例如，你可能希望在 EXE 中使用 ToolHelp 的 *NotifyRegister* 或 *InterruptRegister* callback。如果你使用 __loadds 函式修飾詞，你會限制此程式只能有單一執行實體（instance）。*MakeProcInstance* thunks 可以解決這樣的事態。

*MakeProcInstance* thunks 的工作很簡單：把 AX 設定到 DS，然後跳到指定位址。函式的前置碼（prologue）被期待能夠取出 AX 並把它放到 DS 去。

*MakeProcInstance* 的參數檢驗層首先確定函式所收到的是個合法的位址和 HINSTANCE。*ValidateHInstance* 和 *ValidateCodePtr* 的虛擬碼顯示出對 *MakeProcInstance* 而言所謂的合法參數。*ValidateHInstance* 檢驗 HANDLE 而 *ValidateCodePtr* 檢驗 FARPROC。如果有任一參數不合法，*MakeProcInstance* 即回返，不產生一小段碼（所謂的 thunk）。如果這時候是在 KRNL386 除錯版，你會獲得一個 RIPs 以及適當的錯誤碼，告訴你哪裡不對勁兒。

檢驗過參數之後，*MakeProcInstance* 跳到 *IMakeProcInstance* 去 -- 真正的 thunk 在那裡產生出來。*IMakeProcInstance* 一開始又做了一些檢驗。如果 HINSTANCE 和呼叫端的 DS 暫存器不相同，你會獲得一個 "MakeProcInstance only for current instance" 的訊息，意思是你不能夠爲別的 EXE 模組製作 thunk -- 除非你在呼叫 *MakeProcInstance* 之前改變了 DS （有夠卑鄙的了）。

*IMakeProcInstance* 的另一個重要檢查工作是看看你所要求製作的，是否爲 DLL 函式的 thunk。DLLs 不需要 *MakeProcInstance* thunks，因爲它們可以利用輸出函式（export function）的前置碼（prologue）設定一個固定的 DS 值，例如：

```
MOV AX, 17C7h
MOV DS, AX
```

如果你真的把一個 DLL 函式交給 *MakeProcInstance*，它會默默地把原值傳回給你，完全不改。

*IMakeProcInstance* 的下一個主要部份將決定它是否要產生新的 thunk。如果到目前爲止你所產生的 thunks 個數少於 7，這個 thunk 將來自 current task 的 TDB 節區中的一個區域。否則 *IMakeProcInstance* 將尋覓另一節區（利用 *GlobalAlloc*）以放置新的 thunks，並將此新節區與原節區串成一個串列。

一旦 *IMakeProcInstance* 知道在哪裡產生 thunk，真正的生產動作倒是簡單得很。*MakeProcInstance* thunk 看起來像這樣：

```
MOV AX, hInstance
JMP FAR PTR lpfnProc
```

產生這個 thunk 其實就是產生完整的指令。位元組 0 和 3 分別填入常數（opcode 爲 0xB8 和 0xEA）。偏移位置爲 1 的 WORD 放的是 *MakeProcInstance* 的 hInstance 參數。偏移位置爲 4 的 DWORD 放的則是 *MakeProcInstance* 的 lpProc 參數。

譯註：一圖解千言萬語啦，thunk 的真實內容如下：

| offset | type | value |
|---|---|---|
| 0 | BYTE | B8h |
| 1 | WORD | hInstance |
| 3 | BYTE | Eah |
| 4 | DWORD | lpProc |

**MakeProcInstance 的虛擬碼**

```
// 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 的虛擬碼**

```
// 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 的虛擬碼**

```
// 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 的虚拟码

```
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

我決定在這一節中涵蓋 TOOLHELP 的 *TaskFindHandle* 函式，因爲許多程式員有這樣的印象，認爲 TOOLHELP 函式相當神奇。其實如你即將見到的虛擬碼所示，*TaskFindHandle* 只不過是一種比較方便的作法，用來處理 task database 中的某些欄位。缺點是你將獲得一大串資訊 -- 甚至即使你只想知道一部份。如果你有一段非常關鍵的碼，而它一秒鐘要被呼叫許多次，你恐怕必須在 *TaskFindHandle* 之前直接讀出 Task Database 的資料。會有人辯稱你這麼做犧牲了可移植性。但在這場 Windows 遊戲中，*TaskFindHandle* 所收集的 TDB 欄位並不會改變 -- 因爲改變它會使太多程式因此完蛋。

和幾乎所有的 TOOLHELP 函式一樣，*TaskFindHandle* 首先檢驗參數的合法性。這意味著你必須傳過去一個合法指標，指向 TASKENTRY 結構，而其第一個欄位（dwSize）必須設定爲 TASKENTRY 結構的大小。測試過後，*TaskFindHandle* 才呼叫內部函式真正地把 TDB 內容拷貝到 TASKENTRY 結構中。我在虛擬碼中稱此內部函式爲 *CopyTaskInformation*。

*CopyTaskInformation* 會檢查你是否傳入一個合法的 HTASK。它會看看 TDB 的 FAh 欄位的 TD 標記。這個作法實在太懶了些，你可以輕易做個假節區來矇騙它（還記得前面提過的 IsTask 嗎？它也好不到哪裡去）。假設 TD 檢驗成功，*CopyTaskInformation* 便開始五鬼搬運啦。最後它還會做一些離題的工作：把 task 的堆疊節區的頂端位置、底端位置、目前指標位置放到 TASKENTRY 結構中。

*CopyTaskInformation* 的碼自從 Windows 3.1 TOOLHELP 以來有兩個改變，都和 32-bit tasks 有關。第一個改變是「爲了 Win32 processes 而產生的 pseudo tasks」沒有 HINSTANCE 節區，於是 TOOLHELP 以「task 的 TDB 節區」填入 TASKENTRY.hInst 欄位中。第二個改變和堆疊的邊界有關（wStackTop...等等）。Win32 processes 所屬的 TDBs 中的這些欄位總是 0。*CopyTaskInformation* 不需操心 Win32 task 的 wStackTop、wStackMinimum、wStatckBottom 等欄位。

**TaskFindHandle 的虛擬碼**

```
// 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 的虛擬碼**

```
// 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 程式

我寫了 SHOW16 程式，用來示範我在這一章所做的敘述。SHOW16 原始碼放在書附磁片中。這個程式可以顯示 task list、module list、以及現行的 task 和 module 的詳細情況。此外你也可以雙擊視窗中的某一行輸出結果，取得更深入的資訊。

SHOW16 是個 Windows 95 程式，幾乎可以確定沒辦法在 Windows 3.x、Windows NT、

OS/2 2.x 中執行。它的目標是要儘可能顯示 Windows 95 的 tasks 和 modules 資訊，而不是爲了彰揚移植能力。

SHOW16 開始執行的畫面像圖 **7-2**。左邊 listbox 顯示的若不是 task list 就是 module list。上方有兩個圓鈕可以讓你選擇哪一個 list 才是你的目標。每次你選按圓鈕，畫面就重新整理一次。

右邊 listbox 顯示出你在左邊視窗所選擇的某一項目的詳細內容。這些內容直接從 module database 或 task database 中取出，不是利用 TOOLHELP 函式獲得。前面有著 '+' 符號者，表示可以雙擊之以展開內部資料。如果這項目是個 TDB 或 HMODULE，那麼右邊視窗就會改顯示 TDB 或 HMODULE 的詳細內容。

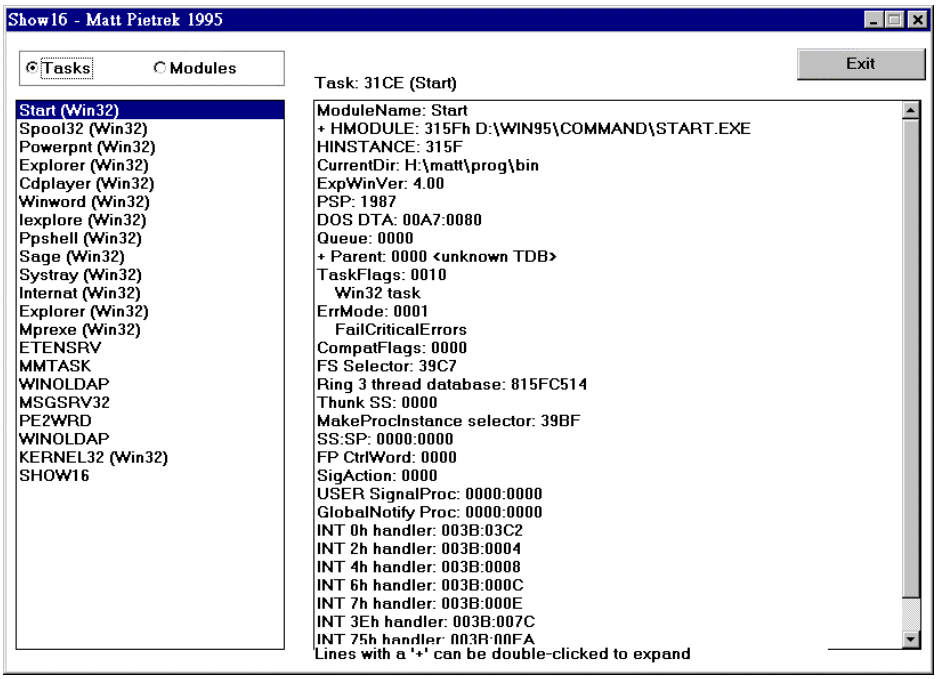



**圖 7-2　SHOW16 程式視窗有兩個 listboxes，提供 tasks 或 modules 資訊。左邊的 listbox 讓你觀察 task list 或 module list，右邊的 listbox 讓你看更細部的內容。**

圖 **7-2** 所顯示的 task 資料中有幾項請特別注意。左邊視窗的 task list 中，凡最後有 (Win32) 字樣者，表示是 32 位元行程。右邊視窗第二行的 HMODULE 有一個 '+' 符號。由於 HMODULE 已經被我交給 *GetModuleFileName*，所以這裡也一併顯示對應的 EXE 或 DLL 的路徑名稱。如果你雙擊此行，右邊視窗會改顯示 HMODULE 的 module database 詳細內容。另一個可被擴展開來的是 parent task。

Task 細節視窗顯示 task database 的所有欄位。但那些沒有用的資訊已被我排除，例如本章稍早介紹 TDB 時所說過的那些沒有用的欄位。負責顯示 task database 的程式碼中有一堆 assert 動作，每一個 assert 用來檢查那些無用的欄位是否爲 0。如果 assert 失敗，表示它可能使用於一些我不曾發現的用途上。

圖 **7-3** 顯示 SHOW16 的其他主要畫面。那是一個 module list。前一半代表 16 位元 EXEs 和 DLLs，是靠 module database 的欄位追蹤而來。後一半（每一行最後有 Win32 者）代表 32 位元 EXEs 和 DLLs，它們是我用暴力方式取得的（因爲它們並不在 module list 之中）。所謂暴力法就是，SHOW16.C 的 *UpdateModuleList* 函式碼檢查每一個 ring3 LDT selector，尋找是否有某個節區爲 module database。如果找到，再檢查其 0xC 位置是否爲 MODFLAGS_WIN32。如果該旗標設立，就把這個 module 加進視窗的顯示列中。

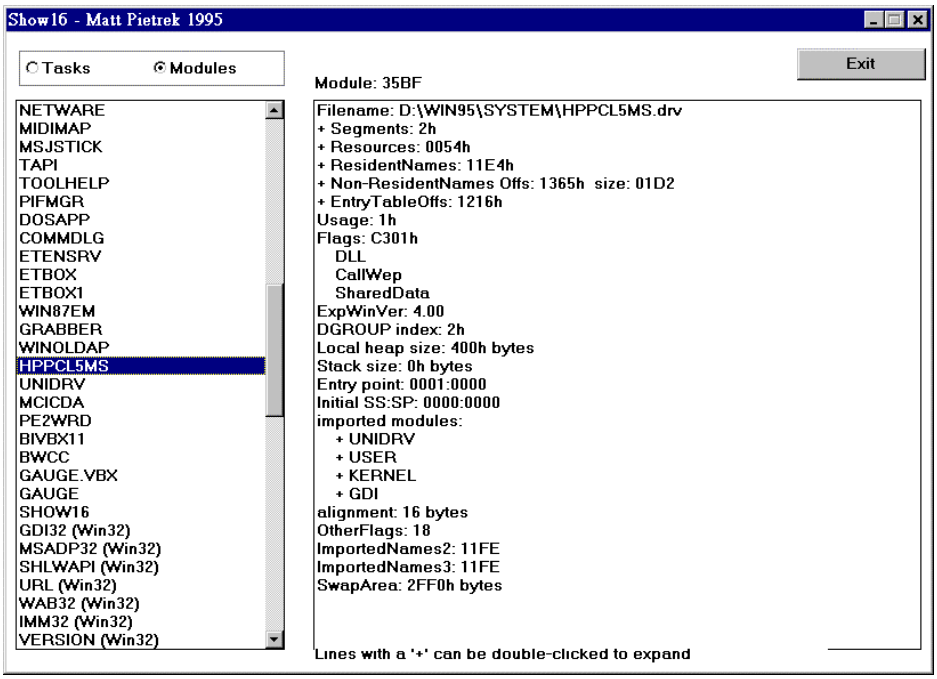



**圖 7-3　左邊視窗中的 module list 包括常規性的(16 位元)以及虛擬的(32 位元）兩種。右邊視窗顯示細部資料。**

圖 **7-3** 的右邊視窗有一些有趣的東西。首先請你找到 "imported modules" 那一行。其下的每一縮排行都代表該模組所聯結的 DLLs。雙擊其中任一行，便會顯示更詳細資料。另外視窗上方也有好幾行是可以展開的：

- Segment table
- Entry table
- Resources
- Resident names
- Nonresident names

**圖 7-4** 顯示一個典型的 segment table。它所記錄的每一個節區，都會顯示出節區序號（邏輯位址的前半部）、global heap handle、型態（code 或 data）、以及大小。雄心勃勃的你或許可以修改 SHOW16 原始碼，允許使用者在雙擊某一行之後，顯示一個記憶體傾印視窗。附帶一提，字形模組沒有 segment table。

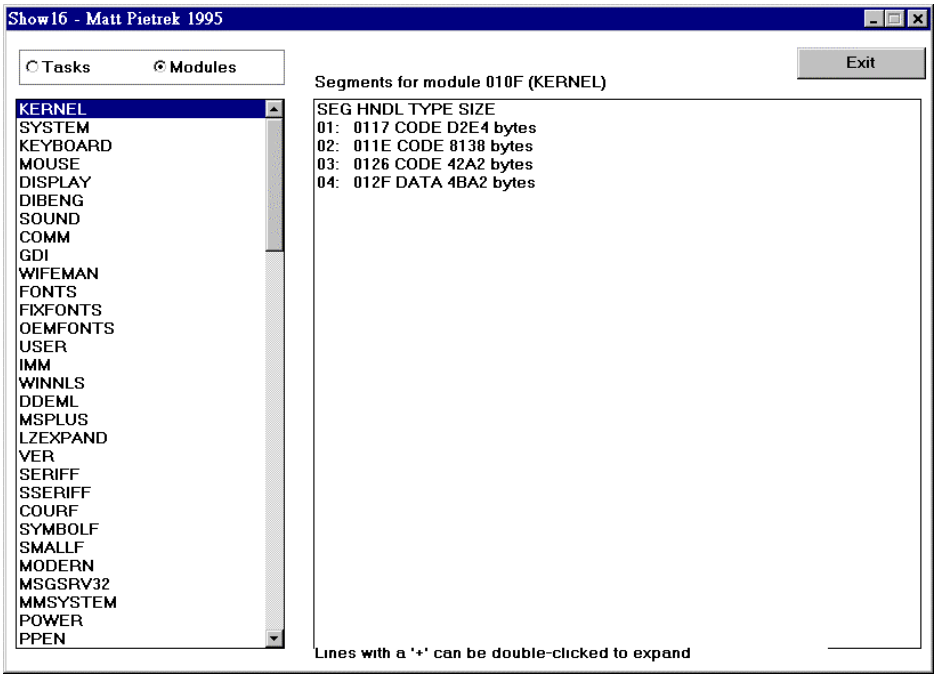



**圖 7-4　細部視窗顯示 segment table 中所記錄的每一個節區的序號、global heap handle、型態、大小。**

圖 7-5 顯示的是資源。如你所見，視窗中的顯示格式類似 resource table 在 module database 中的佈局。緊跟在 resource type line 之下的是一系列的縮排行，每一行代表一筆資源（也就是一張 bitmap、一個 cursor 等等）。每一縮排行提供了以下資訊：資源在檔案中的位置（以 sector 爲單位 -- 如果是 Win16 模組的話）、資源大小（以 sector 爲單位）、資源的 ASCII 名稱或 ID、資源的 global heap handle（如果已載入記憶體的話）。你也可以修改 SHOW16 原始碼，允許使用者在雙擊某一行之後，看到實際的資源圖形。

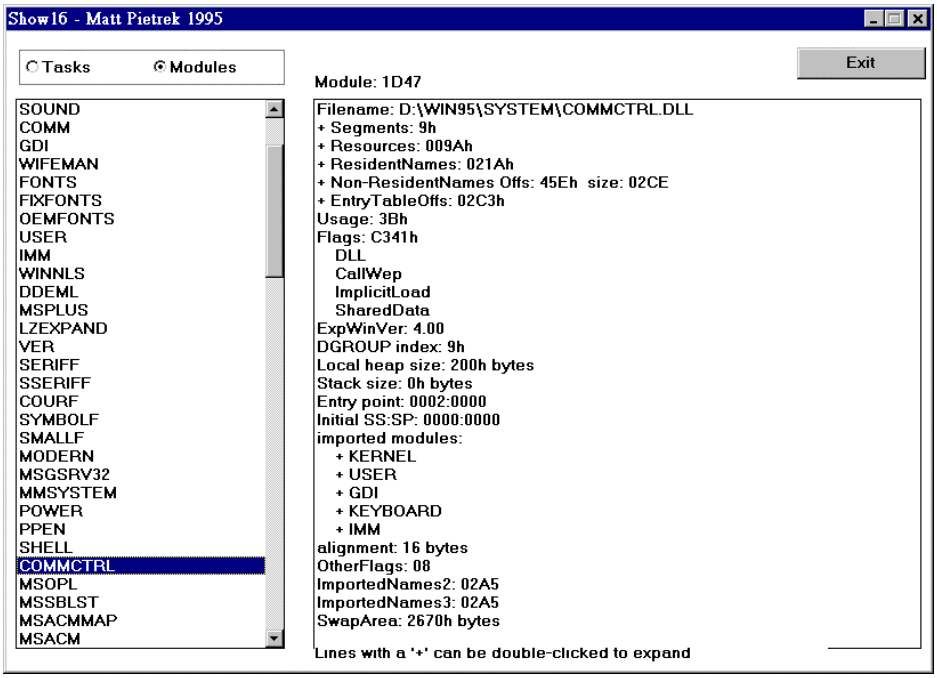



**圖 7-5 Resource 細部視窗所顯示的格式，類似 module database 中的 resource table 的佈局。**

圖 **7-6** 顯示的是 resident names table。它和 nonresident names table 有相同的格式。唯一的不同的，一個要從記憶體中讀取，一個則從檔案中讀取。每一行都以序號開始，然後是輸出函式（或變數）的名稱。Resident names table 的第一行序號爲 0，代表模組名稱，例如 "USER"。Nonresident names table 的第一行序號爲 0，代表模組的 description 字串，例如 "Microsoft Windows User Interface"。

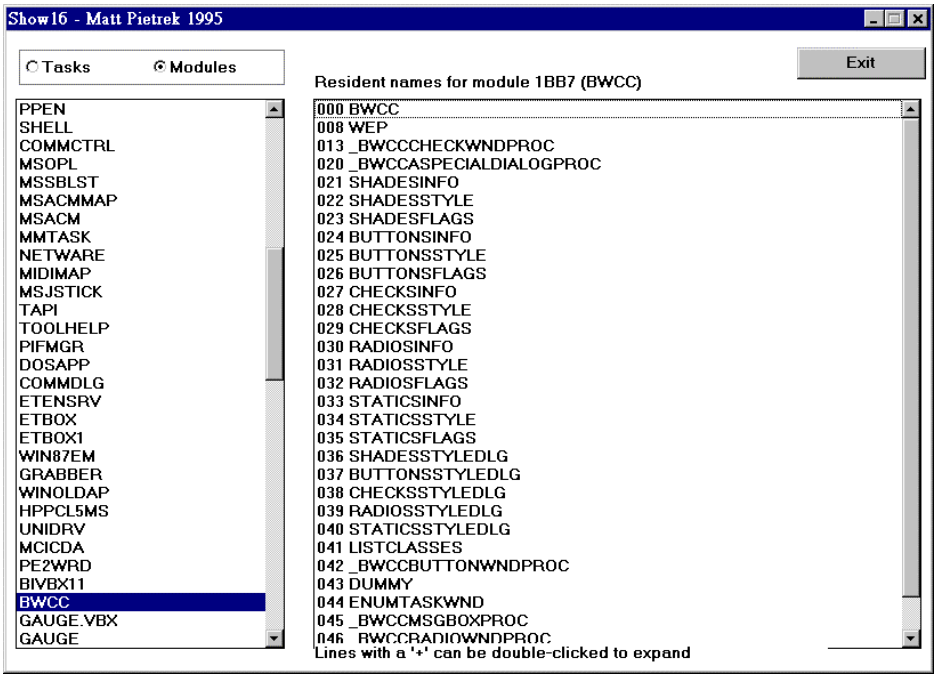



**圖 7-6　Resident names table 的細部視窗，顯示輸出函式及其序號。它和 nonresident names table 有相同格式。**

**圖 7-7** 顯示 entry table 的細部內容。每一行代表模組的 entry table 中的一筆結構。一開始是輸出序號和邏輯位址，然後是旗標值：若不是 MOVEABLE 就是 FIXED，並且通常是 EXPORTED。如果能夠把函式名稱也顯示出來才稱得上十全十美，但這需要花費許多時間，因爲函式名稱通常存放在 nonresident names table 中，需要磁碟讀取動作。此外沒什麼好方法可以快速找到某個輸出序號所對應的函式名稱（在 resident/nonresident names tables 中）。如果面對的是一個小型的 DLL，花費的時間可能不算什麼；如果是大型 DLL 如 USER.EXE 之類（有數百個輸出函式），可能會把系統綁住一段時間唷。

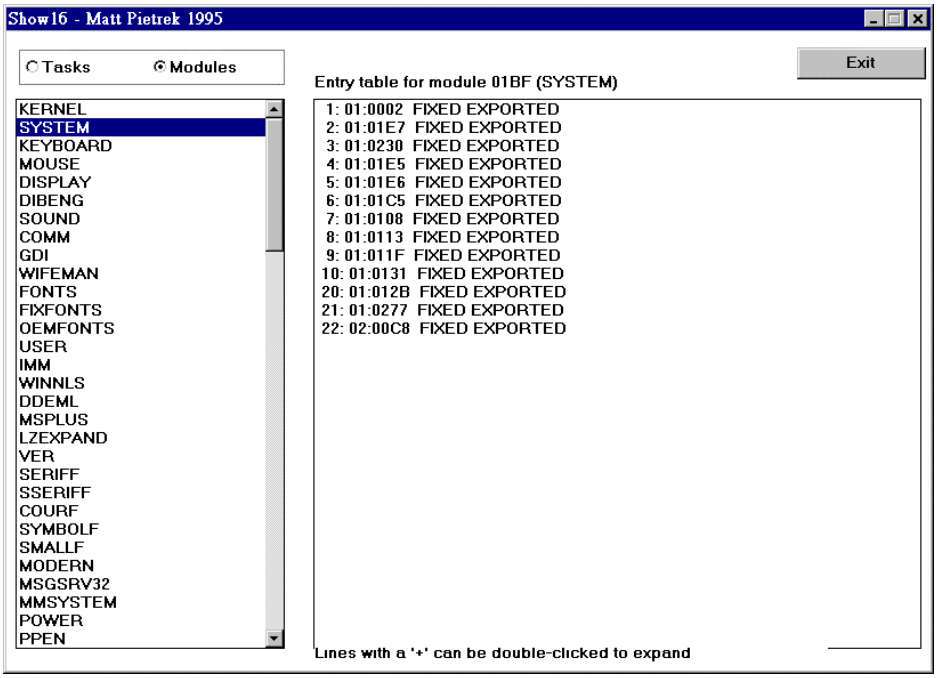
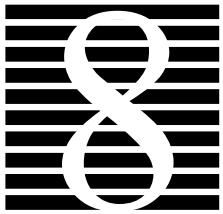



**圖 7-7 Entry table 細部視窗顯示輸出序號、邏輯位址、旗標。**

# 摘要

雖然 Windows 95 被視爲 32 位元作業系統，它還是有許多部份是 16 位元碼。此外，除非絕大部份的程式開發工程移轉到 32 位元，否則 Windows 95 還是需要照顧 16 位元程式。因此，瞭解 Windows 95 的 16 位元元件如何工作，並不是浪費時間。這一章我們看了兩個關鍵性的 16 位元資料結構（module database 和 task database）。我已經顯示它們在某些地方與其 32 位元弟兄(第3章)平行。雖然我沒有涵蓋 Windows 95 對 16 位元 modules 和 tasks 的所有狀況，我想我在這裡揭露的資訊對於任何一位最富冒險精神的探險家而言，都足夠了。



# PE 與 COFF OBJ 檔案格式

一個作業系統的可執行檔格式，從許多角度來看，都是作業系統內建行爲的一面鏡子。雖然可執行檔格式通常並不是一個程式員認爲迫切需要學習的東西，但作業系統的許多有用的知識卻可以在這個過程中獲得。動態聯結、載入器行爲、以及記憶體管理，是特別容易在這個學習過程中推理而得的三個主題。

在這一章，我要帶你走一趟真正的旅遊 -- 遊覽 Portable Executable 檔案格式。PE 檔是微軟設計用於其所有 Win32 位元作業系統（Windows NT、Windows 95、Win32s）的可執行檔格式。

你或許會驚訝爲什麼我要在這本書中涵蓋這一章，畢竟在 Microsoft Developer Network CD-ROM 之中已有數篇相關文章。主要的原因是，PE 檔案中的結構，同時也是 Windows 95 的關鍵性資料結構。例如，Windows 95 把 PE 檔的表頭段（header section）映射到記憶體中，並以之表示一個被載入的模組。爲了瞭解 Windows 95 的核心如何工作，你需要了解 PE 格式。就是這麼簡單！

我探討 PE 檔案的另一個理由是，就像所有出自於微軟的文件一樣，微軟的 PE 文件彷佛假設你已經和 PE 檔案融爲一體，因此它非常地不夠充份，不夠詳細。我這一章的目標是要補充那些文件，並且把它和你每天的經驗關聯在一起。沿著這樣的方向，我會展示 PE 檔影響作業系統的各種方式。

在可見的未來，PE 格式將在微軟的作業系統（包括 Cairo）中扮演一個吃重的角色。甚至即使你使用 Visual C++ 在 Windows 3.1 中開發程式，你還是要使用 PE 檔案（因爲 Visual C++ 的 32 位元 DOS 擴充元件也是使用這種格式）。如果你打算在 Windows 95 內設計任何層面的系統程式，PE 格式的知識更是不可或缺。

在討論 PE 格式的過程中，我不打算費勁地顯示無窮無盡的 hex dump（16 位元傾印碼），然後解釋個別位元的重要性（譯註）。取而代之的是，我要解釋內嵌在其中的觀念，並把它和你每天的 Win32 程式經驗關聯起來。例如，執行緒區域變數（thread local variable）的概念就幾乎把我逼瘋，直到我了解它是如何精緻優雅地在可執行檔中實作出來。由於許多 Win32 程式員擁有 Win16 背景，我會想辦法讓 PE 格式與其 16 位元兄弟（NT 格式）儘量產生一些關聯。

> 譯註：事實上，適當追蹤具代表性的 16 位元傾印碼，不但不枯躁（我已經在我所主持的多次研討會中驗證了這個論點），而且經由「鐵證如山」的追蹤，我們才能從模模糊糊的視野中解脫出來。我將在我自己的新書中詳細解剖 PE 檔案格式。

在微軟引入這個可執行檔格式的同時，它也引入了一個新的目的模組（object module）格式和函式庫（library）格式。這些格式由新的編譯器和組譯器產生。新的 LIB 格式基本上只不過是一群 OBJ 檔綁在一起，再加上一些索引。所以當我說 OBJ 檔時，我說的是 COFF OBJ 和 LIB 檔。這些新的 OBJ 和 LIB 格式分享了 PE 格式的許多觀念。直到最近，都還沒有公開可用的資訊，介紹微軟的 OBJ 和 LIB 檔。甚至在我下筆的時刻，資料都還很缺乏。所以，把 OBJ 和 LIB 檔案格式也涵蓋進來是值得的。

大家都知道，Windows NT（第一個 Win32 作業系統）有 VAX VMS 和 UNIX 的血統。NT 開發小組中的許多關鍵人物在進入微軟之前就是在那些平台上設計並寫碼。當他們

開始設計 NT，很自然地他們會想到使用過去寫過並且測試過的工具。那些工具所產生的可執行檔以及目的模組（object module）的格式稱爲 COFF（Common Object File Format）。

COFF 格式本身已有良好的基礎，但還需要擴充以滿足現代作業系統如 Windows NT 的需要。更新的結果就是 PE（Portable Executable）格式。稱爲可移植性（portable）是因爲任何機器（Intel 386、MIPS、Alpha、Power PC 等等）上的 NT 都可以使用相同的可執行檔格式。當然啦，CPU 指令的二進位編碼是完全不同的，你不可能把一個在 MIPS 機器編譯好的 PE 檔拿到 Intel 系統來跑。重要的是，程式載入器以及程式開發工具不需要針對每一個新的作業系統重寫。

微軟對於 Windows NT 的高度企圖心可以從一件事獲得證明：它放棄了自己原有的 32 位元工具和檔案格式。Windows 3.x 的虛擬裝置驅動程式（VxDs）就是以不同的 32 位元檔案佈局（LE 格式）呈現，而它早在 NT 跳上舞台之前就出現了。由於 Windows 的性格是 "if it ain't broke, don't fix it"，所以 Windows 95 同時使用 PE 和 LE 兩種格式。這使得微軟能夠繼續大規模地使用既有的 Windows 3.x 碼。

雖然，預期一個全新的作業系統（如 Windows NT）有一個全新的可執行檔格式是合理的，但面對 object module（.OBJ 和 .LIB）又不盡然是這麼回事。在 Visual C++ 32 位元版 1.0 之前，所有的微軟編譯器都使用 Intel 的 OMF（Object Module Format）規格，但 Win32 編譯器改採 COFF 格式的 OBJ 檔。微軟的一些競爭者（如 Borland 公司）則堅持使用 Intel 的 OMF 格式。

一些微軟陰謀論者可能會認爲改變 OBJ 格式正是微軟企圖打擊對手的證據。因爲如果要宣稱能夠「與微軟相容」於 OBJ 層次，其他廠商就必須把它們的所有 32 位元工具都轉換爲 COFF OBJ 和 LIB 格式。簡單地說，OBJ 和 LIB 檔案格式可視爲是微軟爲求某種利益而放棄既有標準的又一個例子。

WINNT.H 之中有一些 PE 格式的說明（雖然是極其模糊曖昧），它定義了一些 COFF OBJ 用的結構。我將在稍後細部解釋時使用這些 WINNT.H 中定義的欄位名稱。在 WINNT.H

之中有一段名爲 "Image Format"，從這裡開始就是舊的 MZ-DOS 格式和 NE 格式的表頭，然後才是 PE 格式資料。WINNT.H 提供了 PE 檔所需要的資料結構的明確定義，但只以少量文字說明那些結構、欄位、旗標的意義。PE 格式的表頭設計者 Michael J. O'Leary 想必是一位「長名稱，具自身說明意義」的忠實擁護者，以及一位「巢狀結構和巨集」的忠實擁護者。當面對 WINNT.H 寫碼時，你得面對像這樣的句子：

```
pNTHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_DEBUG].VirtualAddress;
```

除了讀取 PE 格式的成份，你還需要把一些 PE 檔傾印出來，看看呈現其中的觀念。如果你使用微軟的 Win32 開發工具，Visual C++ 和 Win32 SDK 提供的 DUMPBIN 程式可以解析 PE 檔和 COFF OBJ/LIB 檔，並以人類可讀的形式輸出。DUMPBIN 甚至有一項極佳能力，可以反組譯檔案中的 code section。有鑑於微軟曾經宣稱你不可以反組譯其產品，這就有趣了。它竟然供應一個工具，如此輕易地反組譯它的程式和 DLLs。如果反組譯 EXEs 和 OBJs 的能力沒有用，爲什麼微軟要把這項功能加到 DUMPBIN 之中呢？似乎這又是一個 "Do as we say，not as we do"（照著我們所說的做，不要照著我們所做的做）的例子。

Borland 的使用者可以使用 TDUMP 觀察 PE 檔，但 TDUMP 並不了解 COFF OBJ 檔，這不是個大問題，因爲 Borland 根本就不產生 COFF OBJ 檔。我自己也寫了一個 PE 檔和 COFF OBJ/LIB 檔的傾印工具，名爲 PEDUMP，並且我認爲我提供了比 DUMPBIN 更容易了解的輸出格式。它沒有反組譯能力，但其他功能都類似 DUMPBIN，並且還有一些新功能使它身價高一些。PEDUMP 的原始碼放在書附磁碟中，所以我不打算在書中列出太多。取而代之的是，我要列出 PEDUMP 的一些輸出結果，用來解釋我所要說的觀念。

# PEDUMP

PEDUMP 是一個命令列程式，用來傾印 PE 檔和 COFF OBJ/LIB 檔的格式。它使用 Win32 console 能力，去除了使用者介面的負擔。PEDUMP 的使用語法如下：

```
PEDUMP [switches] filename
```

執行 PEDUMP 但不給任何參數，你就可以看到選項（switches）有多少種。PEDUMP 支援下列選項：

```
/A    include everything in dump (也就是把所有選項致能起來)
/H    include hex dump of sections
/I    include Import Address Table thunk addresses
/L    include line number information
/R    show base relocations（只適用於 PE 檔案）
/S    show symbol table（適用於 PE 檔和 COFF OBJ 檔）
```

預設情況下沒有任何選項是致能的。

PEDUMP 把它的輸出送到標準輸出檔（例如螢幕），所以它的輸出可以被重導向（利用 > 符號）。PEDUMP 原始碼附含在本書磁片之中，它是以 Visual C++ 2.0 建造完成的，但我也以 Borland C++ 4.x 編譯它，一切順利。

# Win32 和 PE 的基礎觀念

在探討 PE 檔的佈局之前，我需要察看一些基礎觀念，以便滲透其設計。我將使用模組（module）一詞表示一個 EXE 或 DLL 被載入記憶體後的程式碼、資料和資源。除了程式碼和資料是你的程式直接使用的之外，模組還內含一些支援性資料，Windows 用它來決定程式碼和資料放在記憶體的什麼地方。在 Win16 之中，這些支援性資料結構位於 module database（一個由 HMODULE 指出的節區）。在 Win32，這些資訊保留在 PE 表頭中（以一個 IMAGE_NT_HEADERS 結構），稍後我將有詳細的說明。

關於 PE 檔，你必須知道的一件最重要的事情就是：磁碟中的可執行檔格式非常類似記憶體中的模組。因此 Windows 載入器不需非常辛苦地工作才能根據磁碟檔案產生一個

行程。載入器使用 Win32 記憶體映射檔，把適當的 PE 檔案載入程式的位址空間中。以建築學的方式來說，PE 檔就像是一棟預先製造組合配件的房屋：配件不多，每一個配件可以啪答一聲就定位 -- 只要經過一些些努力。並且，就像組合屋很容易接上電路和水管一樣，PE 檔也很容易與外界產生關係（我是指像連接到 DLLs 之類的動作）。

載入 DLLs 也是一樣地簡單。一旦一個 DLL 或 EXE 模組被載入記憶體，Windows 就把它們當成其他記憶體映射檔一樣。這與 16 位元 Windows 是明顯不同的對比。16 位元 NE 檔案載入器讀入檔案的一部份，並產生個別的資料結構，以便在記憶體中顯現模組。當一個 code segment 或 data segment 需被載入，載入器必須從 global heap 中配置記憶體，找出資料存放在可執行檔中的什麼地方，讀入資料，並做任何必要的修訂。此外，每一個 16 位元模組有責任記住它目前使用的所有 selectors、哪一個節區被拋棄（discarded）了等等。

Win32 就不是這樣了。用來放置模組的碼、資料、資源、import tables、export tables、以及其他東西的記憶體，是位於一塊連續的線性位址空間中。所有你需要知道的，就是這個位址在哪裡。然後你就可以輕輕鬆鬆根據儲存在這個 "image"（譯註）中的指標，找到模組的每一樣資料。

> 譯註：檔案中的模組資料被載入記憶體後，我們稱其爲模組的 "image" 。我保留這個原文，不譯它。

在我們開始之前，你必須知道的另一個觀念是所謂的相對虛擬位址（Relative Virtual Address，RVA）。PE 檔案中的許多欄位內容都是以 RVA 表示。一個 RVA 是某一資料項的 offset（偏移）值 -- 從檔案被映射進來的起點算起。舉個例子，我們說 Windows 載入器把一個 PE 檔映射到虛擬位址空間的 0x400000 處，如果此一 image 有一個表格開始於 0x401464，那麼這個表格的 RVA 就是 0x1464：

```
虛擬位址 0x401464 - 基底位址 0x400000 = RVA 0x1464
```

只要把 RVA 加上基底位址，RVA 就可以被轉換爲一個有用的指標。「基底位址（base address）」是另一個重要觀念，用來描述被映射到記憶體中的 EXE 或 DLL 的起始位址。爲了方便，Windows NT 和 Windows 95 都以模組的基底位址做爲模組的 instance handle（HINSTANCE）。在 Win32 之中，稱呼一個模組的基底位址爲一個 HINSTANCE 似乎有點混淆，因爲 instance handle 一詞根本就是來自於 16 位元 Windows。Win16 每一個程式的副本都擁有它自己的資料節區（以及一個對應的全域性 handle），用以和同一程式的其他副本有所區分，因此才有 instance handle 這樣的詞兒。

在 Win32，程式並不需要和其他人有所區分，因爲它們根本就不共享位址空間。但是 HINSTANCE 這個術語還是保留，以便至少在外觀上使得 Win16 和 Win32 有著一致性。在 Win32 中重要的是你可以針對行程所使用之 DLLs，呼叫 *GetModuleHandle*，取得一個指標。你可以用它來處理模組元件。我所謂的元件是指被輸入（imported）函式或被輸出（exported）函式、它們的位置、它們的 code section 和 data section...等等等。

另一個觀念是 section。PE 檔或 COFF OBJ 檔的一個 section 大略相當於 16 位元 NE 檔的一個 segment。Sections 內含程式碼或資料。某些 section 內含的是你的程式宣告並直接使用的程式碼或資料，另有一些存放資料的 sections 是由聯結器和 librarian（製造 LIB 的工具）產生，內含對作業系統很重要的資訊。在微軟的某些文件中，也把 sections 稱爲 objects。後者已經是如此地氾濫，所以我還是使用 sections 一詞。我將在「通常會遇到的 Sections」一節再詳細討論 sections，目前，你只要知道 section 到底是什麼就好了。

在跳到 PE 檔細節之前，好好看看圖 **8-1**，它顯示出 PE 檔案的大致佈局。我將分別解釋每一塊內容，但是大局觀是很有幫助的。

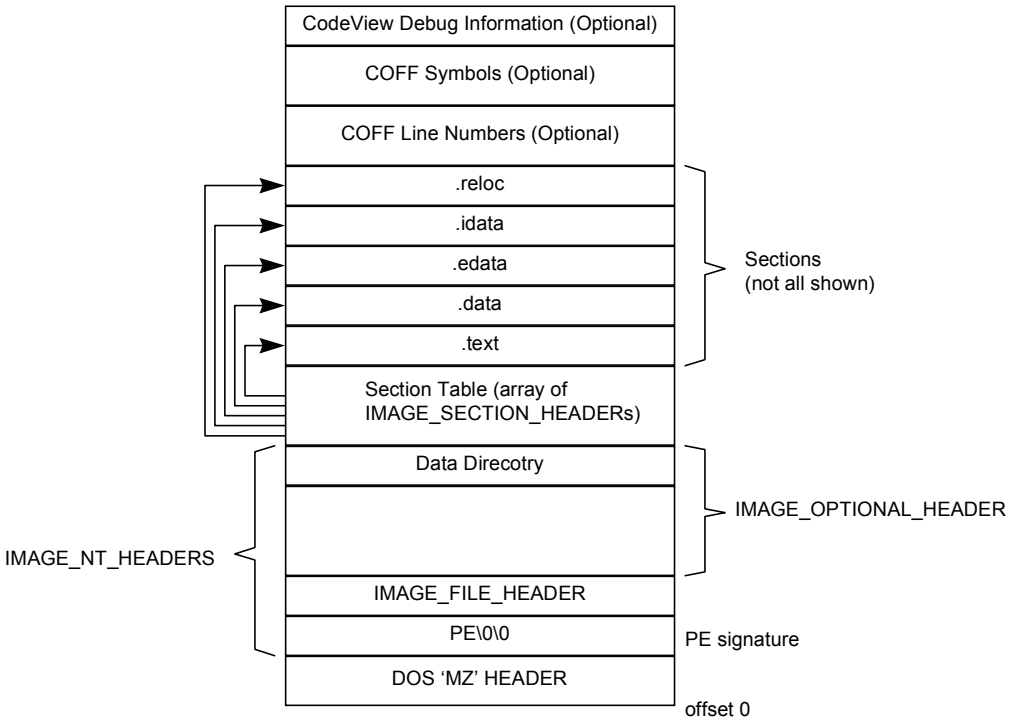



**圖 8-1　PE 檔案的大致佈局**

# PE 表頭（PE Header）

我們的 PE 之旅第一站是 PE 表頭。像其他的微軟可執行檔格式一樣，PE 檔有一系列的欄位，固定在一個已知（並且很容易找到）的位置上，定義檔案的其他部份。PE 表頭內含的重要資訊包括程式碼和資料區域的大小位置、適用的作業系統、堆疊（stack）的最初大小等等。

和其他的微軟可執行檔格式一樣，PE 表頭並非在檔案的最起頭處。檔案最前面的數百個

位元組是所謂的 DOS stub：一個極小的 DOS 程式，用來輸出像 "This Program cannot be run in DOS mode" 這樣的訊息。如果你在一個不支援 Win32 的作業系統上跑一個 Win32 程式，就會獲得這個錯誤訊息。當 Win32 載入器把一個 PE 檔映射到記憶體，記憶體映射檔（memory mapped file）的第一個位元組對應到 DOS Stub 的第一個位元組。好得很，於是，你每執行一個 Win32 程式，就獲得了一個免費的 DOS 程式（在 Win16 中這個 DOS stub 並不載入記憶體）。

和其他的微軟可執行檔格式一樣，你可以在 DOS stub 的表頭中找到「真正的表頭」。WINNT.H 爲 DOS stub 表頭定義了一個結構，循此我們將非常容易找到 PE 表頭。e_lfanew 欄位是一個相對偏移值（或說是 RVA），指向真正的 PE 表頭。爲了獲得指標，你必須爲 RVA 加上 image 的基底位址：

```
// Ignoring typecasts and pointer conversion issues for clarity...
pNTHeader = dosHeader + dosHeader->e_lfanew;
```

一旦你擁有指向 PE 表頭的指標，真正的趣味就開始了。PE 表頭整個是個 IMAGE_NT_HEADERS 結構，定義於 WINNT.H。這個結構正是 Windows 95 的 module database。每一個被載入的 EXE 或 DLL 都以一個 IMAGE_NT_HEADERS 結構表現出來。此一結構有一個 DWORD 和兩個子結構：

```
DWORD Signature;
IMAGE_FILE_HEADER FileHeader;
IMAGE_OPTIONAL_HEADER OptionalHeader;
```

Signature 欄位內容應該是 ASCII 的 PE\0\0。如果 DOS stub 表頭中的 e_lfanew 欄位指向一個 NE Signature 而不是一個 PE Signature，表示你面對的是一個 Win16 NE 可執行檔。如果是 LE Signature 則表示你面對的是一個 VxD 檔。如果是 LX Signature 則表示 OS/2 檔案。

再來是一個 IMAGE_FILE_HEADER 結構。此結構內含最基礎的檔案資訊。和 COFF 比較起來，它似乎沒有做什麼更改。除了是 PE 表頭的一部份，它也出現在微軟 32 位元編譯器所產生的 COFF OBJs 檔的最前端。IMAGE_FILE_HEADER 的欄位如下。

### WORD Machine

檔案使用於哪一種 CPU。下面是 CPU 識別碼的定義：

```
Intel I386            0x14C
Intel i860            0x14D
MIPS R300             0x162
MIPS R400             0x166
DEC Alpha AXP         0x184
Power PC              0x1F0（little endian）
Motorola 68000        0x268
PA RISC               0x290（Precision Architecture）
```

譯註：但是在我的 Visual C++ 4.2 WINNT.H 檔案中，只有如下定義：

```
#define IMAGE_FILE_MACHINE_I386     0x14c   // Intel 386.
#define IMAGE_FILE_MACHINE_R3000    0x162 //MIPS little-endian, 0x160 big-endian
#define IMAGE_FILE_MACHINE_R4000    0x166   // MIPS little-endian
#define IMAGE_FILE_MACHINE_R10000   0x168   // MIPS little-endian
#define IMAGE_FILE_MACHINE_ALPHA    0x184   // Alpha_AXP
#define IMAGE_FILE_MACHINE_POWERPC 0x1F0   // IBM PowerPC Little-Endian
```

### WORD NumberOfSections

EXE 的 OBJ 中的 sections 個數

### DWORD TimeDateStamp

聯結器產生此一檔案的時刻。其格式是自從 1969 年 12 月 31 日 4:00 P.M. 之後的總秒數。

### DWORD PointerToSymbolTable

COFF 符號表格的偏移位置。此欄位只對 COFF 除錯資訊有用。PE 檔支援數種除錯格式，所以除錯器應該參考 data directory（稍後描述）欄位中的 IMAGE_DIRECTORY_ENTRY_DEBUG。

### DWORD NumberOfSymbols

COFF 符號表格中的符號個數。請看前一欄位。

### WORD SizeOfOptionalHeader

一個可有可無的表頭（出現在本結構之後）的大小。在 EXE 檔中，這也就是 IMAGE_OPTIONAL_HEADER 的大小。在 OBJ 檔中，微軟說它總是 0。然而，觀察 KERNEL32.LIB，有一個 OBJ 在其中，而它的這個值不是 0。所以呢，微軟的話總是要打點折扣。

### WORD Characteristics

描述此一檔案的性質。一些比較重要的性質如下：

```
0x0001  檔案中沒有重定位（relocation）
0x0002  檔案是個可執行檔（也就是說不是 OBJ 或 LIB）
0x2000  檔案是動態聯結函式庫，不是程式。
```

PE 表頭的第三個成份是 IMAGE_OPTIONAL_HEADER 結構。對於 PE 檔而言，這一部份其實並不是可有可無。要知道，COFF 格式允許不同的生產者在標準的 IMAGE_FILE_HEADER 之後定義一個結構。IMAGE_OPTIONAL_HEADER 正是 PE 設計者認爲在基本的 IMAGE_FILE_HEADER 資訊之外還需要的一些重要資訊。

你並不需要知道 IMAGE_OPTIONAL_HEADER 的所有欄位。最重要的兩個欄位是 ImageBase 和 Subsystem。如果你喜歡，你可以跳著讀或是整個略過下面這些欄位說明。

### WORD Magic

這值用來定義 image 的狀態：

```
0x0107  一個 ROM image
0x010B  一個正常的（一般的）EXE image。大部份 PE 檔都含此值。
```

### BYTE MajorLinkerVersion
### BYTE MinorLinkerVersion

產生此 PE 檔案之聯結器版本。以十進位而非十六進位表示。例如 2.23 版。

**DWORD SizeOfCode**

所有 code sections 的總和大小。大部份檔案只有一個 code section，所以此欄位通常也就是 .text section 的大小。

**DWORD SizeOfInitializedData**

所有內含「初始值資料」的 sections（但不包括 code sections）的總和大小。然而它似乎並不包括 initialized data sections 在內。

**DWORD SizeOfUninitializedData**

這是「需要載入器將之付諸虛擬位址空間，但卻不佔用磁碟檔案」的所有 sections 大小總和。這些 sections 在程式啓動時並不需要什麼特別內容，所以才導至 Uninitialized data 這一稱呼。未初始化的資料通常集中在 .bss section 中。

**DWORD AddressOfEntryPoint**

這是 image 開始執行的位址。這是一個 RVA（Relative Virtual Address），通常會落在 .text section。此欄位對於 DLLs 或 EXEs 都適用。

**DWORD BaseOfCode**

這是一個 RVA（Relative Virtual Address），表示檔案中的 code section 從何開始。code section 通常在 data section 之前，在 PE 表頭之後。微軟聯結器所產生的 EXEs 中，此值通常爲 0x1000。Borland 的 TLINK32 則通常指定此值爲 0x10000，因爲預設情況下它是以 64KB 爲排列邊界，不像微軟的聯結器採用 4K 爲邊界。

**DWORD BaseOfData**

這是一個 RVA（Relative Virtual Address），表示檔案中的 data section 從何開始。data section 通常在 code section 和 PE 表頭之後。

### DWORD　ImageBase

一旦聯結器產生了一個可執行檔，它就假設這個檔案將被映射到特定位置的記憶體中。這個欄位放的就是記憶體起始位址。假設出這樣一個位址，聯結器才能完成最佳化。如果檔案真的被載入到這個位址，程式碼就完全不需要修補。稍後我將對此有更多的討論。在 NT 3.1 EXE 中，預設的 image base 是 0x10000，DLL 則是 0x400000。在 Windows 95，0x10000 不再適用，因爲那將落在一個被所有行程共享的位址空間中。因而到了 NT 3.5，微軟把預設的 Win32 EXE ImageBase 改爲 0x400000。原來那些 NT EXE 程式在 Windows 95 之中將需要比較長的載入時間，因爲載入器必須重新定位。稍後我會詳細討論所謂的「重新定位（relocation）」。

### DWORD　SectionAlignment

一旦映射到記憶體中，每一個 section 保證從一個「此值之倍數」的虛擬位址開始。

### DWORD　FileAlignment

在 PE 檔中，組成每一個 section 的原始資料（raw data）保證是從一個「此值之倍數」的虛擬位址開始。預設值是 0x200，如此一來 section 總是能夠從磁碟的 sector 開始（因爲 sector 長度總是 0x200 個位元組）。這個欄位相當於 NE 檔中的 segment/resource alignment 大小。與 NE 檔不同的是，PE 檔不會有上百個 sections，所以因其特定排列而浪費的空間不大。

### WORD　MajorOperatingSystemVersion
### WORD　MinorOperatingSystemVersion

使用此一可執行檔之作業系統的最小版本。這個欄位與稍後即將出現的另一欄位 "subsystem" 有類似的用途，因此頗令人困惑。Win32 檔案的這兩個欄位通常指出 1.0。

**WORD MajorImageVersion**
**WORD MinorImageVersion**

使用者自定的欄位，允許你擁有不同版本的 EXE 或 DLL。你可以利用聯結器的 /VERSION 選項設定其值。例如：

```
LINK /VERSION:2.0 myobj.obj
```

**WORD MajorSubsystemVersion**
**WORD MinorSubsystemVersion**

使用此一可執行檔的子系統的最小版本。典型的數值是 4.0，代表 Windows 4.0，也就是 Windows 95。

**DWORD Reserved1**

似乎總是 0。

**DWORD SizeOfImage**

載入器必須在意的整個 image 大小。它的範圍從 image base 開始，直到最後一個 section 爲止。最後一個 section 的尾端必需是 SectionAlignment 的倍數。

**DWORD SizeOfHeaders**

PE 表頭以及 section table 的大小。各個 sections 的資料就從其後展開。

**DWORD CheckSum**

此檔案的一個 CRC checksum。就像微軟的其他可執行檔一樣，此欄位通常被忽略並被設爲 0。然而，所有的 driver DLLs、所有在開機時間載入的 DLLs、以及 server DLLs 都必須有一個合法的 checksum。其演算法可以在 IMAGEHLP.DLL 中獲得。IMAGEHLP.DLL 的原始碼可以在 Win32 SDK 中找到。

### WORD　Subsystem

此一可執行檔用來作爲使用者介面的子系統。WINNT.H 定義了下列常數：

```
NATIVE=1          不需要子系統（例如驅動程式）
WINDOWS_GUI=2     在 Windows GUI 子系統中執行
WINDOWS_CUI=3     在 Windows 字元模式子系統中執行（也就是 console 應用程式）
OS2_CUI=5         在 OS/2 字元模式子系統中執行（也就是 OS/2 1.x 應用程式）
POSIZ_CUI=7       在 Posix 字元模式子系統中執行
```

### WORD　DllCharacteristics

一組旗標值，用來指示 DLL 的初始化函式（例如 *DllMain*）在什麼環境下被呼叫。這個值總是 0，但作業系統卻還是會在四個 events 發生時呼叫 DLL 的初始化函式。此值的四個數值分別意義如下：

```
1：當 DLL 被載入一個行程的位址空間時呼叫之。
2：當一個執行緒結束時呼叫之。
4：當一個執行緒開始時呼叫之。
8：當 DLL 退出時呼叫之。
```

### DWORD　SizeOfStackReserve

執行緒初始堆疊的保留大小。然而並不是所有這些記憶體都被系統委派（committed）。此值預設爲 0x100000（1MB）。如果你在程式中呼叫 *CreateThread* 並指定其堆疊大小爲 0，獲得的執行緒就有一個與此值相同大小的的堆疊。

### DWORD　SizeOfStackCommit

一開始即被委派（committed）給執行緒初始堆疊的記憶體數量。微軟的聯結器預設此一欄位爲 0x1000（一個 page），Borland 的 TLINK32 則把它設爲 0x2000（兩個 pages）。

### DWORD　SizeOfHeapReserve

保留給最初的 process heap 的虛擬記憶體數量。這個 heap 的 handle 可以利用 *GetProcessHeap* 獲得。注意，並不是所有這些記憶體都已被委派（committed）。

### DWORD SizeOfHeapCommit

一開始即被委派（committed）給 process heap 的記憶體數量。此值預設爲 0x1000 個位元組。

### DWORD LoaderFlags（在 Windows NT 3.5 中被註明爲老舊欄位）

這個欄位似乎和除錯有關。我從來沒有看過哪一個可執行檔的這些位元是設立的，也不曾清楚知道聯結器如何設定它們。下面是其可能的數值與意義：

1：在開始這個行程之前先引發一個中斷點（breakpoint）？
2：在行程被載入之後引發一個除錯器執行起來？

### DWORD NumberOfRvaAndSizes

在 DataDirectory（下一欄位）陣列中的項目個數。目前的工具總是把此值設爲 16。

### IMAGE_DATA_DIRECTORY DataDirectory [IMAGE_NUMBEROF_DIRECTORY_ENTRIES]

陣列一開始的元素內含可執行檔重要部位的 RVA（Relative Virtual Address）及大小。陣列最後的一些元素目前還用不著。陣列的第一個元素代表 exported function table（如果有的話）的位址和大小，第二個元素代表 imported function table 的位址和大小，依此類推。完整的列表請看 WINNT.H 中的定義。

譯註：以下就是完整的列表。

```
// Directory Entries
#define IMAGE_DIRECTORY_ENTRY_EXPORT         0   // Export Directory
#define IMAGE_DIRECTORY_ENTRY_IMPORT         1   // Import Directory
#define IMAGE_DIRECTORY_ENTRY_RESOURCE       2   // Resource Directory
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION      3   // Exception Directory
#define IMAGE_DIRECTORY_ENTRY_SECURITY       4   // Security Directory
#define IMAGE_DIRECTORY_ENTRY_BASERELOC      5   // Base Relocation Table
#define IMAGE_DIRECTORY_ENTRY_DEBUG          6   // Debug Directory
#define IMAGE_DIRECTORY_ENTRY_COPYRIGHT      7   // Description String
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR      8   // Machine Value (MIPS GP)
#define IMAGE_DIRECTORY_ENTRY_TLS            9   // TLS Directory
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG    10   // Load Configuration Directory
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT   11   // Bound Import Directory in headers
#define IMAGE_DIRECTORY_ENTRY_IAT            12   // Import Address Table
```

這個陣列企圖讓載入器能夠迅速在 image 中找到特定的 section，不需要一一經過每一個 sections、比對名稱、再繼續尋找...。

大部份的陣列元素描述一整個 section 資料，但 IMAGE_DIRECTORY_ENTRY_DEBUG 卻只包含 .rdata section 中的一小部份。更多資訊將在「The .rdata section」一節中披露。

# The Section Table

在 PE 表頭和真正的 section 資料之間，橫躺著一個 section table。其中內含 image 的每一個 sections 的資訊。sections 是以其起始位址來排列，而不是以其字母次序來排列。

現在是澄清 section 意義的時候了。在 NE 檔中，程式碼和資料儲存在檔案的不同節區（segments）之中。NE 表頭的一部份是一個結構陣列（譯註：作者指的是 segment table），陣列的每一個元素恰對應一個節區。每一個結構內含節區的一些資訊，包括節區的型態（code 或 data）、大小、位置。PE 檔的 section table 正類似於 NE 檔的 segment table。

但是，和 NE 檔 segment table 不同，PE 檔的 section table 並不爲每一塊 code 或 data 存放一個對應的 selector 值。取而代之的是，section table 的每一筆資料貯存了一個位址，在那裡，檔案的原始資料被映射到記憶體。雖然 sections 類似 32 位元的 segments，事實上它們不是一個個的 segments，而是行程虛擬位址空間中一塊記憶體範圍。

PE 和 NE 之間的另一個差異就是它們對於那些「不爲程式所用，而爲作業系統所用」的資料的管理方式。我以兩筆資料爲例，一是 EXE 所用到的 DLLs 的列表，一是 fixup table 的位置。在 NE 檔中，資源並未被視爲一個節區，甚至雖然有 selector 指派給資源，但 NE 表頭的 segment table 中並未存有資源的相關資訊。資源被歸類爲一個個別表格，接近 NE 表頭的尾端。imported 函式資訊和 exported 函式資訊也不能擔保有自己的節區，它們只能夠填塞在 NE 表頭的邊緣。

PE 檔的故事就不同了。任何被視爲重要的 code 或 data 都會被存放在一個如假包換的 section 之中。因此，imported 函式的資訊被儲存在自己一個 section 中，exported 函式的資訊也被儲存在自己一個 section 中。relocation data 的情況亦復如此。程式或作業系統所需的任何 code 或 data 都可能有它自己的 section。

馬上我要開始討論各式各樣的 sections，但首先我必須描述作業系統管理 sections 時所用的資料。就在 PE 表頭之後是一個 IMAGE_SECTION_HEADER 結構陣列，該陣列的元素個數已記錄在 PE 表頭之中（也就是 IMAGE_NT_HEADER.FileHeader.NumberOfSections 欄位）。PEDUMP 程式可以輸出 section table 以及所有 section 的欄位和屬性。**圖 8-2** 顯示 PEDUMP 對於一個典型 EXE 檔案的 section table 的輸出。**圖 8-3** 顯示的則是 PEDUMP 對於一個 OBJ 檔的 section table 的輸出。

```
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
```

**圖 8-2 一個典型的 EXE 檔的 section table**

```
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
```

**圖 8-3　一個典型的 OBJ 檔的 section table**

每一個 IMAGE_SECTION_HEADER 是 EXE 檔案或 OBJ 檔案中的一個 section 的完整資料，格式如下：

**BYTE　　Name[IMAGE_SIZEOF_SHORT_NAME]**

這是一個 8 八位元組的 ANSI 名稱（不是 unicode），表示 section 名稱。大部份 section 名稱都以句點開始（例如 ".text"），但此非必要條件。你也可以自己命名：或是在組合語言中以 segment directive 方式，或是在 C/C++ 之中以 #pragma code_seg 或 #pragma data_seg 方式，都行。Borland C++ 的使用者則應該以 #pragma codeseg 爲之。注意，如果 section 名稱全長 8 位元組，那麼就沒有 NULL 結束符號。Borland C++ 4.0x 的 TDUMP 程式忽略了這一點，以至於在某些情況下會吐出一些垃圾。如果你是 *printf* 的熱愛者，可以使用 "%.8s" 列印格式。

```
union {
        DWORD    PhysicalAddress;
        DWORD    VirtualSize;
} Misc;
```

這個欄位的意義視其應用於 EXE 或 OBJ 而定。在 EXE 中它代表 code section 或 data section 的虛擬大小，這是在它們被調整爲「最接近之 file-alignment 倍數」之前的大小。稍後出現的 SizeOfRawData 欄位則是調整後的大小。有趣的是，Borland 的 TLINK32 把這個欄位和 SizeOfRawData 欄位的意義顛倒了，卻仍能正常運作。對於 OBJ 檔，這個欄位則是代表 section 的實際位址。第一個 section 從位址 0 開始。上一個 section 的實際位址再加 SizeOfRawData 值即爲下一個 section 的起始位址。

**DWORD　　VirtualAddress**

在 EXE 中，這個欄位代表載入器應該將 section 映射過去的 RVA 位置。爲了計算 section 的真正起始位址，你必須將此值再加上基底位址。微軟的工具把第一個 section 的此一欄位設爲 0x1000。對於 OBJ 檔，這個欄位沒有意義，總是爲 0。

### DWORD SizeOfRawData

在 EXE 中，這個欄位代表 section 大小被調整爲「最接近之 file-alignment 倍數」後的值。假設 file-alignment 大小爲 0x200，VirtualSize 的值是 0x35A，那麼這個欄位的值就將會是 0x400（位元組）。在 OBJ 中，這個欄位表示由編譯器或組譯器指定的真正 section 大小。換句話說，對於 OBJ，這個欄位等同於 SizeOfRawData 欄位。

### DWORD PointerToRawData

這是「以檔案起頭爲基準」的偏移值，section 的原始資料可以在該處尋獲。如果你自己以記憶體映射的方式映射了一個 PE 檔或 COFF 檔（而不是經由作業系統的載入器），這個欄位就比 VirtualAddress 重要多了。因爲這種情況下你對整個檔案有一個完全的線性映射，所以你必須根據此值找到 section 的資料，而不是根據 VirtualAddress 中的 RVA（Relative Virtual Address） 值。

### DWORD PointerToRelocations

在 OBJs 中，這是「以檔案起頭爲基準」的偏移值，用來指向 section 的「重定位資訊」。每一個 OBJ section 的「重定位資訊」緊跟在 section 資料之後。在 EXEs 中，這個欄位（以及下一欄位）沒有意義，總是爲 0。當聯結器產生一個 EXE，它會決定大部份的 fixups（待修正記錄），只剩下基底位址的重定位位址以及 imported 函式的重定位位址，留待載入時再解決。兩份相關資訊放在 base relocation section 和 imported functions section 之中，所以 EXEs 不需要在每一個 section 之後又有「重定位資訊」。

### DWORD PointerToLinenumbers

這是行號表的偏移值（以檔案起頭爲基準）。行號表把原始碼行號和其被映射至記憶體的位址扯上關係。在 CodeView 除錯格式中，行號被儲存爲除錯資訊的一部份，然而在 COFF 除錯格式中，行號在觀念上和符號名稱以及符號型態等資訊是分開來的。通常，只有 code sections（例如 .text 或 CODE）才有行號資訊。在 EXE 檔中，行號資訊被收集放在檔案的最尾端。在 OBJ 檔中，行號表就放在每一個 section 的原始資料以及重定位表格的後面。我將在本章的「COFF 除錯資訊」一節詳細描述行號表的格式。

### WORD　NumberOfRelocations

重定位表格（由 PointerToRelocations 指向）中的重定位項目個數。此欄位只用於 OBJ 檔。

### WORD　NumberOfLinenumbers

行號表格（由 PointerToLinenumbers 指向）中的行號個數。

### DWORD　Characteristics

大部份程式員口中說的 "flags"，在 COFF/PE 格式中稱爲 "characteristics"。這個欄位是一組旗標值，用來表示 section 中的屬性（code 或 data、可讀或可寫…等等等）。完整列表請看 WINNT.H 中的常數定義（IMAGE_SCN_XXX_XXX）。最重要的旗標值列於表 **8-1**。

有沒有注意到什麼東西遺失未提？首先，請注意，沒有提到 PRELOAD 屬性。NE 格式允許你針對節區指定 PRELOAD 屬性，使它能夠在模組載入之後立刻被載入記憶體。OS/2 2.0 的 LX 格式也有類似屬性，允許你指定最多 8 個 pages 爲 PRELOAD。但是 PE 格式沒有！我們只好假設微軟對於其在 Win32 平台上「有需要才載入（demand-paged loading）」的效率頗有自信。

**表 8-1　COFF section 屬性旗標值**

| flag（旗標值） | 用途 |
|---|---|
| 0x00000020 | 這是一個 code section。常與 0x80000000（executable flag）聯用。 |
| 0x00000040 | 這個 section 內含初始化資料。幾乎所有 sections（除了可執行者以及 .bss）都有此旗標設定。 |
| 0x00000080 | 這個 section 內含未初始化資料（例如 .bss section）。 |
| 0x00000200 | 這個 section 內含文字說明或其他型態的資訊。典型的例子就是編譯器製造出來的 .drectve section，其中內含的是給聯結器看的命令。 |
| 0x00000800 | 這個 section 的內容不應該被放到最終的 EXE 檔案中。這是編譯器或組譯器用來傳遞資訊給聯結器的場所。 |

| | |
|---|---|
| 0x02000000 | 表示這個 section 可以被拋棄，原因是一旦被載入後行程再也不需要用到它。最常見的例子就是 base relocations section（.reloc） |
| 0x10000000 | 可共享的 section。如果在 DLL 中，儲存於此 section 的資料將可被所有使用此 DLL 之行程共用。預設情況下並不設立此旗標，也就是說同一 DLL 的每一個雇主（一個行程）有它們自己的一份資料拷貝。<br><br>以更技術的角度來說，一個 shared section 告訴記憶體管理器說，請爲此 section 設定 page mappings，使所有用到這個 DLL 的行程們都能夠參考到相同的實際記憶體。爲了做出一個可共享的 section，請在聯結時使用 SHARED 屬性，例如：<br><br>`LINK /SECTION:MYDATA,RWS...`<br><br>告訴聯結器說名爲 MYDATA 的 section 應該是可讀的（R）、可寫的（W）、可共享的（S）。Borland C++ DLL 的 data section 預設使用此一屬性。 |
| 0x20000000 | 這個 section 可執行。此屬性通常和 0x00000020 旗標聯用。 |
| 0x40000000 | 這個 section 內容可讀。EXE 的任何一個 sections 幾乎都設立此旗標。 |
| 0x80000000 | 這個 section 允許被讀入資料。如果 EXE 之中沒有任何一個 section 設立此一旗標，載入器就應該把所有的記憶體映射頁都做上「唯讀」或「僅能執行」的記號。.data 和 .bss 通常會設立此旗標。 |

另一個不見蹤影的東西是所謂的 page lookup table（頁次查詢表格）。IMAGE_SECTION_HEADER 在 OS/2 LX 格式中的對等物並不直接指出一個 section 可以在檔案的何處找到，反而是，OS/2 LX 檔案內含一個 page lookup table，用來指定 section 之中某些 pages 的屬性和位置。PE 格式免除了這些東西，保證一個 section 的資料一定連續儲存於檔案之中。比較這兩種格式，LX 允許較多的彈性，但是 PE 比較簡單也比較容易使用 -- 在寫過兩種格式下的檔案傾印器以及反組譯器之後，我可以這麼說。

另一個受歡迎的改變是，在原來的 NE 格式中，各個項目的位置只以一個 DWORD 表示。NE 檔案中幾乎每一樣東西的位置都以 sector 爲單位。爲了找到真正的檔案偏移值，你必須首先從 NE 表頭中找出 alignment 的單位大小，把它轉換爲 sector 大小（通常不是 16 就是 512 個位元組）。然後你必須把每一個 sector 偏移值再乘以 sector 大小，才是真正的檔案偏移位置。PE 檔案中各個項目的位置一律是以「距離記憶體映射位址起

始處」的偏移值表示，相當簡單。結論是，PE 格式遠比 NE、LX 或 LE 格式簡單易用 -- 假設你會使用記憶體映射檔的話。

# 常會遇到的 Sections

關於 sections 的意義以及它如何定位，相信你已有個概念。現在我們要看看在 EXE 和 OBJ 檔中的一些常見的 sections。雖然我所列的並不是全部，但已經涵蓋了你每天會接觸到（但也許你自己並不知道）的 sections。排列次序是根據其重要性以及遭遇它們的頻繁度。

## .text section

.text 內含所有一般性的程式碼。由於 PE 檔在 32 位元模式下跑，並且不受約束於 16 位元節區，所以沒有理由把程式碼分開放到不同的 sections 中。聯結器把所有來自 .OBJ 的 .text 集合到一個大的 .text 中。如果你使用 Borland C++，其編譯器製作出來的 code section 名爲 CODE 而不是 .text。請看稍後的「Borland CODE 以及 .icode sections」一節。

我很驚訝地發現，在 .text 中除了編譯器製作出來的碼，以及 runtime library 的碼之外，還有一些其他東西。在 PE 檔中，當你呼叫另一模組中的函式（例如 USER32.DLL 中的 *GetMessage*），編譯器製造出來的 CALL 指令並不會把控制權直接傳給 DLL 中的函式，而是傳給一個 JMP DWORD PTR [XXXXXXXX] 指令，後者也位於 .text 中。JMP 指令跳到一個位址去，此位址儲存在 .idata 的一個 DWORD 之中。這個 DWORD 內含該函式的真正位址（函式進入點），如圖 **8-4** 所示。

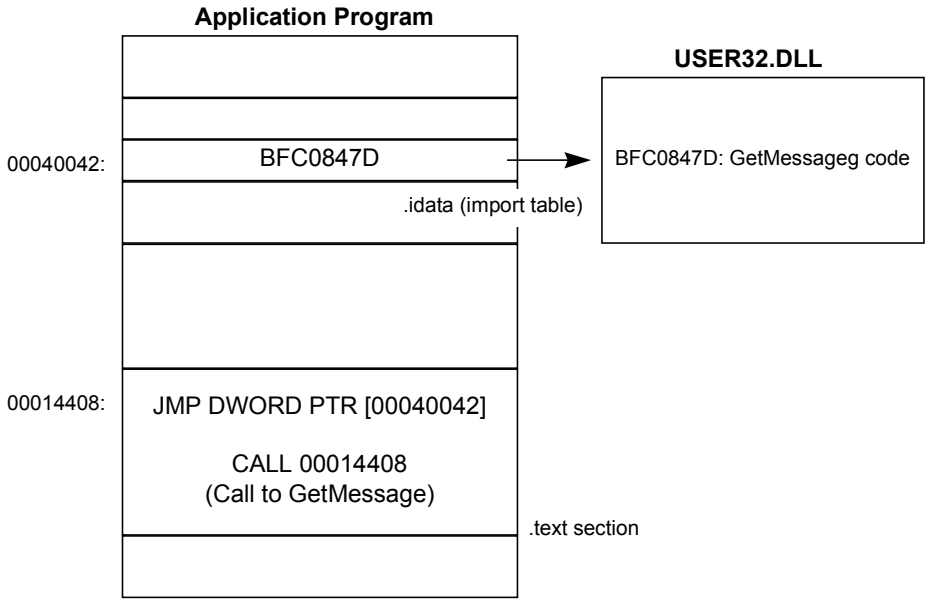



**圖 8-4　一個 PE 檔呼叫 imported function。**

沉思良久，我終於瞭解爲什麼 DLL 的呼叫需要以這種方式實現。把對同一個 DLL 函式的所有呼叫都集中到一處，載入器就不再需要修補每一個呼叫 DLL 的指令。PE 載入器需要做的，就只是把 DLL 函式的真實位址放到 .idata 的那個 DWORD 之中，根本就沒有程式碼需要修補。這和 NE 檔有極明顯的差異。NE 檔的每一個節區內含一串待修正記錄（fixup records），如果某一節區呼叫同一個 DLL 函式 20 次，載入器就必須忙碌 20 次，將函式位址拷貝到待修正記錄之中。PE 檔這種處理方式也有缺點：你不能夠以 DLL 函式的真正位址初始化一個變數。例如：

```
FARPROC pfnGetMessage = GetMessage;
```

是把 *GetMessage* 函式位址放到 pfnGetMessage 變數中。在 Win16 這沒問題，在 Win32，變數中放的其實將是稍早我說過的 JMP DWORD PTR [XXXXXXXX] 指令的位址。如果你根據這個函式指標來呼叫函式，事情會如你所預期。但如果你要以此指標讀取 *GetMessage* 的前數個位元組，幸運之神不會站在你那邊。稍後我將在「PE 檔案的輸出（exports）」一節中再繼續討論這個主題。

在我寫完本章的第一個版本之後，Visual C++ 2.0 推出了。它介紹另一種新的呼叫方式。如果你看過 Visual C++ 2.0 的系統表頭檔（例如 WINBASE.H），你將看到和過去不同的東西。在 Visual C++ 2.0 中，API 函式原型都有一個 __declspec(dllimport）作爲原型的一部份。當你呼叫一個這樣的函式，編譯器不會在模組的另一個地方產生 JMP DWORD PTR [XXXXXXXX] 指令，而是產生一個 CALL DWORD PTR [XXXXXXXX] 函式呼叫。XXXXXXXX 位址位於 .idata 內，作用與原先在 JMP DWORD PTR [XXXXXXXX] 指令中的位址相同。就我所知，Borland C++ 4.5 編譯器並沒有這樣的性質。

## Borland CODE 與 .icode sections

Borland C++ 4.5 編譯器和聯結器不能夠使用 COFF OBJ 檔，它們固守 Intel OMF 32 位元格式。Borland 編譯器當然可以吐出一個名爲 .text 的 section，但它卻選擇 "CODE" 這個名稱。爲了決定 PE 檔中的一個 section 名稱，Borland C++ 聯結器（TLINK32.EXE）從 OBJ 檔中取出 section 名稱並把它攔斷爲 8 個字元（如果必要）。所以，Borland C++ 有一個 CODE section 而不是 .text section。

名稱不同不算什麼，更重要的不同存在於 Borland 工具聯結出來的 PE 檔中。稍早我說過，所有對 OBJ 的呼叫都經由一個 JMP DWORD PTR[XXXXXXXX] 指令。在微軟的系統中，這指令來自一個 import library 的 .text section。也就是說聯結器不需要知道如何產生這個指令。import library 可視爲「需要聯結到 PE 檔中」的更多的碼和資料。

Borland 系統的處理方式就不一樣，它比較類似 16 位元 NE 檔案所採行的方法。Borland 聯結器所使用的 import library 真正只是函式名稱和 DLL 名稱的列表而已。TLINK32 有責任決定哪一些待修正記錄（fixups）是針對外部 DLLs，然後爲它們產生 JMP DWORD PTR [XXXXXXXX] 指令。Borland C++ 4.0 的 TLINK32 把它所產生的這個指令存放在 .icode section 中，但是到了 Borland C++ 4.02，TLINK32 又改變了，把所有這些 JMP 指令放到 CODE section 中。

## .data section

這是你的初始化資料的存放區。所謂初始化資料，包括全域變數和靜態變數（global and static variable），在編譯器時期就給定初值。它也包括字串常數，像是 C/C++ 程式中的 "Hello World"。聯結器把 OBJ 和 LIB 檔案中所有的 .data 組合起來放到 EXE 檔的 .data。區域變數（local variable）位於執行緒堆疊之中，不佔用 .data 或 .bss 空間。

## DATA SECTION

Borland C++ 以 DATA 作爲其預設的資料區域。相當於微軟編譯器所製作的 .data。

## .bss section

這是任何未初始化的靜態變數和全域變數的存放區。聯結器把 OBJ 和 LIB 檔案中所有的 .bss 組合起來放到 EXE 檔的 .bss。在 section table 中，.bss 的 RawDataOffset 欄位總是爲 0，表示這個 section 不佔用檔案的任何一點空間。TLINK32 並不吐出一個 .bss，它的作法是擴充 DATA section 的虛擬大小，以接納未初始化的資料。

## .CRT section

這是微軟的 C/C++ runtime library（CRT）所使用的另一個初始化的 data section。這裡所放的資料用於「在 *main* 或 *WinMain* 之前執行的 static C++ 類別建構式」中。

## .rsrc section

此處內含模組資源。早期的 NT，16 位元 RC.EXE 所輸出的 .RES 檔並不被微軟的聯結器所瞭解，那個時候的 CVTRES 程式就是用來把一個 .RES 檔轉換爲一個 COFF OBJ，把資源放到 OBJ 檔的一個 .rsrc 之中。聯結器於是就可以產生一個 resource OBJ。也就是說，聯結器不需要知道任何有關於資源的事情。後來的微軟聯結器已經能夠直接處理 .RES 檔。我將在「PE 檔案的資源」一節中涵蓋資源 section 的格式。

## .idata section

這個 section 內含有關於「模組從其他 DLLs 中輸入（import）函式和資料」的相關資訊。它相當於 NT 檔的 module reference table。關鍵性的差異是，每一個輸入函式都被列在這個 section 之中。如果要在 NE 檔中找出對等的資訊，你必須深掘每一個節區的原始內容的重定位資料。我將在「PE 檔案的輸入（imports）」一節中涵蓋 import table 的格式。

## .edata section

這是 PE 檔輸出函式（export function）的相關資訊。它的 NE 對等物是 entry table、resident names table 和 nonresident names table 的組合。和 Win16 不同的是，很少有機會從一個 EXE 中輸出一個函式出去，所以通常你只在 DLL 中才會看到 .edata。Borland C++ 所產生的 EXE 是個例外，它總是有一個輸出函式（__GetExceptDLLinfo）給 runtime library 的內部使用。

export table 的格式將於本章的「PE 檔案的輸出（exports）」一節討論。如果使用微軟工具，.edata 的資料來自 .EXP 檔，但是聯結器沒有能力產生這個檔案，必須依賴函式庫管理器 LIB32.EXE 掃描 OBJ 檔然後才產生 EXP 檔，然後才能交給聯結器。是的，那是真的，EXP 檔其實就是擁有不同副檔名的 OBJ 檔罷了。使用 PEDUMP /S 觀察 EXP 檔，你可以看到其中的輸出函式（export functions）。

## .reloc section

這個 section 內含一表格的 base relocations。所謂 base relocation 是一個指令或初始化變數的調整值。如果載入器沒有辦法把 EXE 或 DLL 檔案載入到預設的位址的話，就必須做這樣的調整；否則載入器可以忽略「重定位」這件事情。

如果你希望載入器總是能夠把 image 載入到預定的基底位址，你可以使用 /FIXED 選項，告訴聯結器剝除本項資訊。雖然這可以節省 EXE 的檔案空間，卻可能使得 EXE 檔

沒辦法在其他 Win32 平台上執行。例如，你爲 NT 開發了一個 EXE，基底位址爲 0x10000。如果你告訴聯結器把這資訊剝除，這個 EXE 就沒有辦法在 Windows 95 上跑，因爲 0x10000 不適用（Windows 95 的最低載入位址是 0x400000，也就是 4MB）。

注意一點，編譯器所產生的 JMP 和 CALL 指令，其所使用的 offset 值是與該指令成相對位址關係，而不是真正的 32 位元平滑節區的 offset 值。如果 image 被載入到一個並非聯結器指定的基底位址去，JMP 和 CALL 指令不需修改，因爲它們用的是相對定址。也就是說，其實沒有如你想像中那麼多的重定位動作要做。只有使用 32-bit offset 的指令才需要重定位動作。假設你有下面的全域變數宣告：

```
int i;
int *ptr = &i;
```

如果聯結器設定基底位址是 0x10000，變數 i 的位址是 0x12004。在被用來存放 ptr 的記憶體中，聯結器將寫入 0x12004，因爲那是變數 i 的位址。如果載入器爲了某種理由把檔案載入到 0x70000 處，i 的位址將是 0x72004，然而，預先初始化過的 ptr 值變成錯誤值，因爲 i 現在的位置已經提升了 0x60000。

這就是需要重定位資訊參一腳的場合了。.reloc 用來表示「聯結器所假設的載入位址」和「真正的載入位址」之間的差異。我將在「PE 檔的 Base Relocations」一節有比較詳細的討論。

## .tls section

當你使用編譯器的 "__declspec(thread)" 性質，你定義的資料並沒有進入 .data 或 .bss 之中，倒是有一份拷貝進入 .tls 之中。.tls 的名稱是因爲 thread local storage 而來，和 *TlsAlloc* 函式家族有密切關係。

爲了簡單描述所謂的 thread local storage，請把它想像成「讓每一個執行緒擁有各自的全域變數」的一種方法。也就是說，每一個執行緒可以擁有它自己的一組靜態資料，使用這些資料的程式碼，不需在意現在是哪一個執行緒正在執行。假設某程式有數個執行緒，處理相同的工作。也因此執行相同的碼。如果你宣告一個 tls，像這樣：

```
__declspec(thread) int i = 0;  // this is a global variable declaration
```

每一個執行緒將因此擁有變數 i 的一個副本。

你可以明白地在執行時期索求並使用 tls，相關函式是 *TlsAlloc*、*TlsSetValue*、*TlsGetValue* 等（第 3 章對於 *TlsXXX* 函式的描述比較詳細）。通常，以 __declspec(thread) 在程式中宣告你的資料，比使用 *TlsAlloc* 簡單得多。

這裡有一個壞消息。在 NT 和 Windows 95 中，tls 機制不能夠有效運作 -- 如果運作對象是以 *LoadLibrary* 動態載入的 DLL。至於在一個 EXE 或是一個隱式載入(implicitly loaded，譯註)的 DLL 之中，每一件事情都沒問題。如果你不能夠以隱晦方式載入 DLL，但又需要讓每一個執行緒有自己的資料，那你只好使用 *TlsAlloc* 和 *TlsGetValue*。注意，每一執行緒真正的記憶體區塊並不是放在 .tls section 中，也就是說，當切換執行緒的時候，記憶體管理器並不改變「實際映射至模組之 .tls section」的記憶體。.tls 內只不過是一些資料，用來初始化真正的執行緒專屬區塊。初始化動作是靠作業系統與 runtime library 的合作，過程之中需要另外一些儲存在 .rdata 之中的資料：TLS directory。

> 譯註：如果程式與 DLL 的 import library 聯結，我們說這是 implicitly link，並導至 DLL 被 implicitly loaded。如果程式沒有與 DLL 的 import library 聯結，而是在需要時（執行時期）呼叫 *LoadLibrary* 和 *GetProcAddress* 以取得函式位址，再呼叫之，我們稱此爲 explicitly linked，並導至 DLL 被 explicitly loaded。

## .rdata section

.rdata 至少有四個用途。第一，在被微軟聯結器產生的 EXEs 之中，.rdata 內含 debug directory（OBJ 檔中並沒有 debug directory）。而在 TLINK32 所產生的 EXEs 之中，debug directory 是一個名爲 .debug 的 section。debug directory 是一個由 IMAGE_DEBUG_DIRECTORY 結構所組成的陣列。這些結構持有檔案之中各種除錯資訊的型態、大小、位置。除錯資訊可能有三種型態：CodeView、COFF、FPO。**圖 8-5** 顯示 PEDUMP 對一典型的 debug directory 的輸出結果。

| Type | Size | Address | FilePtr | Charactr | TimeData | Version |
|---|---|---|---|---|---|---|
| COFF | 000065C5 | 00000000 | 00009200 | 00000000 | 2CF83F3D | 0.00 |
| (unknown) | 00000114 | 00000000 | 0000F7C8 | 00000000 | 2CF83F3D | 0.00 |
| FPO | 000004B0 | 00000000 | 0000F8DC | 00000000 | 2CF83F3D | 0.00 |
| CODEVIEW | 0000B0B4 | 00000000 | 0000FD8C | 00000000 | 2CF83F3D | 0.00 |

**圖 8-5　一個典型的 debug directory。**

debug directory 並不一定會在 .rdata 的起始處被發現。要找到它，你必須使用 data directory 的第 7 筆資料（IMAGE_DIRECTORY_ENTRY_DEBUG）。還記得嗎，data directory 位於 PE 表頭的尾端。爲了確定微軟聯結器所做出來的 debug directory 的項目個數，請把 debug directory 的大小（可從 debug directory 的 "size" 欄位獲得）除以 IMAGE_DEBUG_DIRECTORY 的結構大小。至於 TLINK32 則是把 debug directories 的真正數量記錄在 "size" 欄位中，而不是位元組總長度。PEDUMP 可以處理這兩種情況。

.rdata 的第二個有用部份是 description string。如果你在程式的 .DEF 檔中指定 DESCRIPTION，被指定的字串就會出現在 .rdata 之中。在 NE 檔中，description string 總是 nonresident names table 的第一個項目。description string 主要是用來設定一個有用的字串，用以描述這個檔案。不幸的是我還沒有發現什麼好方法來找到它。我曾經看過有些 PE 檔的 description string 放在 debug directory 之前，有些卻在 debug directory 之後。

.rdata 的第三個用途是爲了 OLE 程式設計所需的 GUIDs。UUID.LIB 內含一系列的 128 位元 GUIDs，當作 interface IDs。這些 GUIDs 都放在 EXE 或 DLL 的 .rdata 中。

.rdata 的最後一個用途是用來放置 TLS（Thread Local Storage）的 directory。TLS directory 是一個特殊資料結構，被編譯器的 runtime library 使用，以便能夠透明化地提供 TLS 給程式中宣告的變數。TLS directory 的格式可以在 MSDN（Microsoft Developer Network）光碟片中找到："Portable Executable and Common Object File Format"。我們對 TLS directory 的主要興趣是指向資料（用來初始化每一個 tls 區塊）的起頭和結尾的指標，TLS directory 的 RVA（Relative Virtual Address）可以在 PE 表頭的 data directory 的

IMAGE_DIRECTORY_ENTRY_TLS 項目中獲得。至於真正用來初始化 TLS 區塊的資料可以在 .tls section 中找到。

## .debug$S 和 .debug$T sections

.debug$S 和 .debug$T 只出現於 COFF OBJs 之中，內含 CodeView 的符號和型態資訊。看來十分奇怪的 section 名稱係衍生自前一版微軟編譯器的節區名稱（$$SYMBOLS 和 $$TYPES）。.debug$T 的唯一目的是爲了放置 .PDB 檔（內有專案中所有 OBJs 的 CodeView 型態資訊）的路徑名稱。聯結器利用 .PDB 爲 EXE 檔產生出一部份的 CodeView 資訊。

## .drectve section

這個 section 只出現在 OBJ 檔，內含聯結器命令列參數的文字表達。例如，在微軟的 Visual C++ 編譯器，下面字串一定會出現在 .drectve 中：

```
-defaultlib:LIBC -defaultlib:OLDNAMES
```

當你在程式碼中使用 __declspec(export) ，編譯器會製造出命令列上的對應東西，放在 .drectve 之中（例如 export:MyFunction）。

## 含有 $ 的 sections（只針對 OBJs/LIBs）

在 OBJ 檔中，名稱含有 $ 的 sections（例如 .idata$2）將被聯結器特別對待。聯結器把所有擁有相同名稱（直至 $ 字元）的 sections 組合成爲單一一個 section。例如，如果聯結器遭遇 .idata$2 和 .idata$6，它會把它們整合爲一個 .idata。

被整合的 sections 的次序是以 $ 之後的字元爲準。聯結器以字母順序排列之，所以 .idata$2 在 .idata$6 之前。.idata$A 則在 .idata$B 之前。

那麼到底帶有 $ 的 section 做什麼用？最普遍的用法就是 import library 利用它們來存放最終的 .idata（import section）的各部份資料。這可有趣了，聯結器本身並不需要從頭產生 .idata，最終的 .idata 是由 OBJ 和 LIB 各貢獻一部份而來。

## 雜項的 sections

有時候我會從 PEDUMP 的輸出中看到其他一些 sections。例如 Windows 95 的 GDI32.DLL 內含一個名爲 _GPFIX 的 data section，我們推測它大概與 GP fault 的處理有關。

這有雙重意義。第一，不要以爲你只能使用編譯器或組譯器提供的標準 sections。若有需要，別猶豫不決。在微軟的 C/C++ 編譯器中，你可以使用 #pragma code_seg 和 #pragma data_seg。Borland 的使用者則可以使用 #pragma codeseg 和 #pragma dataseg。若是組合語言，你只要產生一個 32 位元節區並給予不同於「標準 sections」的名稱即可。TLINK32 會把同類別的 code segments 組合在一起，所以你要不就得爲每一個 code segment 指定一個類別名稱，要不就關閉 "code segment packing" 這個性質。

# PE 檔案的輸入（imports）

> 譯註：我把 import 譯爲「輸入」，import function 譯爲「輸入函式」；export 譯爲「輸出」，export function 譯爲「輸出函式」。

稍早我曾說過，呼叫外部的 DLLs 其實並不是直接呼叫 DLL 本身，而是跳到一塊存放著 JMP DWORD PTR [XXXXXXXX] 指令的區塊去（可能放在 .text 或是 .icode）。如果你在 VC++ 中使用 __declspec(dllimport)，函式呼叫就會變成 CALL DWORD PTR [XXXXXXXX]。不論哪一種情況，JMP 或 CALL 指令中的位址都存放在 .idata。JMP 或 CALL 指令會把控制權轉給該位址。如果你仍然不清楚這一部份，請回頭看看圖 **8-4**。

在被載入記憶體之前，存放在 PE 檔的 .idata 中的資訊是給載入器用來決定函式位址並修補它們，以便完成 image 用的。而在被載入之後，.idata 內含的是指標，指向 EXE/DLL 的輸入函式。請注意，這裡我所討論的所有陣列和結構都內含在 .idata 之中。

.idata section（或稱爲 import table）是以一個 IMAGE_IMPORT_DESCRIPTOR 陣列開始。被 PE 檔隱晦聯結（implicitly link）的 DLLs 都會在此有一個對應的 IMAGE_IMPORT_DESCRIPTOR 結構。最後一個 IMAGE_IMPORT_DESCRIPTOR 結構的內容全部爲 NULL，以此做爲結束符號。IMAGE_IMPORT_DESCRIPTOR 的格式描述於下。

#### DWORD Characteristics/OriginalFirstThunk

這是一個偏移值（一個 RVA，Relative Virtual Address），指向一個 DWORD 陣列。每一個 DWORD 事實上是一個 IMAGE_THUNK_DATA union，對應於一個輸入函式。稍後我會描述 IMAGE_THUNK_DATA DWORD 的格式。

#### DWORD TimeDateStamp

這是檔案的產生時刻。通常此欄爲 0。然而微軟的 BIND 程式可以將此一 IMAGE_IMPORT_DESCRIPTOR 所對應的 DLL 的產生時刻寫到這裡來。

#### DWORD ForwarderChain

這個欄位關係到所謂的 forwarding（轉交），意味一個 DLL 函式再參考（呼叫、利用）另一個 DLL。例如在 Windows NT 中，KERNEL32.DLL 將它的某些輸出函式轉交給 NTDLL.DLL。應用程式可能以爲它呼叫 KERNEL32.DLL，而事實上呼叫的是 NTDLL.DLL。這個欄位內含一個索引，指向 FirstThunk 陣列（稍後敘述）。被這個索引所指定的函式就是一個轉交函式。不幸的是，函式如何轉交並未在微軟的文件中公開。關於轉交，請看稍後「輸出函式的轉交（Export forwarding）」一節。

#### DWORD Name

這是一個 RVA（Relative Virtual Address），指向一個以 null 爲結束字元的 ASCII 字串，內含 imported DLL 的名稱。

### PIMAGE_THUNK_DATA FirstThunk

這是一個 RVA，指向一個 DWORDs（IMAGE_THUNK_DATA）陣列。大部份情況下，該 DWORD 被解釋爲一個指向 IMAGE_IMPORT_BY_NAME 結構的指標。然而，以函式序號（而非函式名稱）輸入，也是有可能的。

IMAGE_IMPORT_DESCRIPTOR 結構中，最重要的部份是 imported DLL 的名稱以及兩個 IMAGE_THUNK_DATA DWORDs 陣列。每一個 IMAGE_THUNK_DATA DWORDs 對應到一個輸入函式。在 EXE 檔案中，兩個陣列（分別由 Characteristics 和 FirstThunk 欄位指向）平行存在，並且都以 NULL 爲結束符號。

爲什麼需要兩個平行陣列呢？第一個陣列（由 Characteristics 指向）從不被修改，有時候它又被稱爲 hint-name table。第二個陣列（由 FirstThunk 指向）則被載入器改寫。載入器一一檢閱每一個 IMAGE_THUNK_DATA 並找出它所記錄的函式的位址，然後把位址寫入 IMAGE_THUNK_DATA 這個 DWORD 之中。

稍早我曾經說過對 DLL 函式的呼叫會導至一個 JMP DWORD PTR [XXXXXXXX] 指令。[XXXXXXXX] 事實上參考到 FirstThunk 陣列中的一個元素。由於這個 IMAGE_THUNK_DATA 陣列內容已被載入器改寫爲輸入函式的位址，所以它又被稱做 Import Address Table（IAT）。圖 **8-6** 顯示這兩個陣列。

對於 Borland 使用者，上述敘述需要一點點修改。TLINK32 所產生的 PE 檔遺漏了一個陣列：Characteristics（也就是 hint-name array）總是 0。顯然 Win32 載入器根本不需要它。

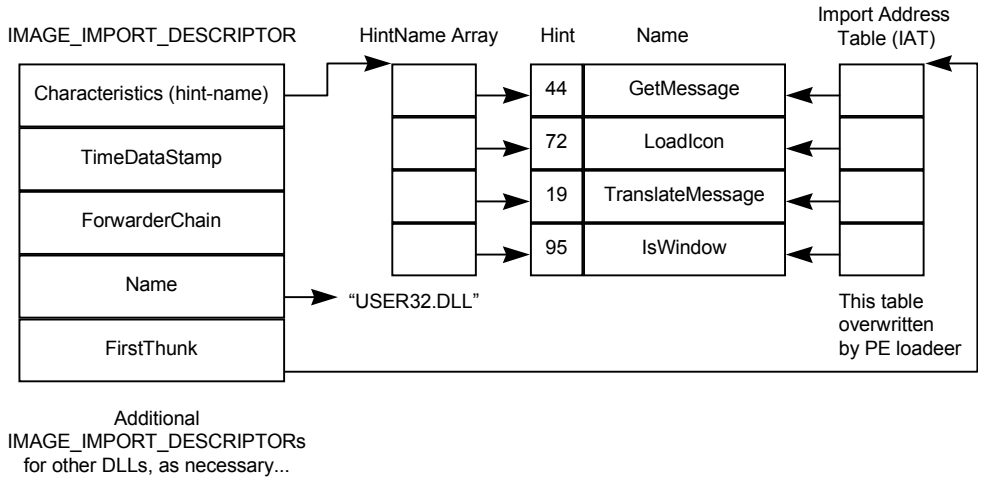



**圖 8-6　PE 檔如何處理輸入函式（imported functions）**

故事即將結束，但當我撰寫 PEDUMP 時遭遇了一個有趣的問題。在「從不停止追求最佳化」的腳步中，微軟把 Windows NT system DLLs 的 IMAGE_THUNK_DATA 陣列也給最佳化了。於是，IMAGE_THUNK_DATA 不再內含「能夠找到輸入函式」的資訊，而是內含輸入函式的位址。換句話說載入器不再需要辛苦尋找函式位址並改寫陣列內容，Win32 SDK 的 BIND 可以執行此一最佳化動作。不幸的是這會造成 PE 傾印工具程式一些問題。可能你會說『嘿 Matt，爲什麼你不使用 hint-name table 呢？』主意是不錯，問題是 Borland 檔案沒有提供 hint-name table。好，別爲我擔心，PEDUMP 可以處理所有的情況，但程式碼無可避免會有些零亂，不夠優雅。

由於 import address table 通常是一個可寫區域，所以要攔截 EXE 或 DLL 對其他 DLL 的呼叫就相當容易了。你只要修改 import address table 中指向欲攔截函式的那個陣列元素即可。不需要修改呼叫端或被呼叫端的程式碼。我在第 10 章建立了一個 Win32 API spy 程式，就是依賴這個性質。

微軟所產生的 PE 檔並不是完全由聯結器合成。事實上，呼叫 DLL 函式所需的所有資訊都是由相關的 DLL import library 提供。當你對 DLL 做聯結動作，函式庫管理器（LIB.EXE）掃描所有 OBJ 檔並產生一個 import library。它可不同於 16 位元 NE 檔的 import library。這個由 32 位元 LIB.EXE 產生的 import library 有 .text section 和數個 .idata$ sections。.text 內含 JMP DWORD PTR [XXXXXXXX] 指令，每一個這樣的程式碼段落有一個名稱記錄在 OBJ 檔的符號表格中。符號名稱與 DLL 輸出函式的名稱相同（例如 _DispatchMessage@4）。

import library 中的一個 .idata$ section 內含上述 JMP 指令中的 DWORD。另一個 .idata$ section 提供一個空間給 "hint ordinal"（後面緊跟著輸入函式的名稱）。這兩個欄位組成一個 IMAGE_IMPORT_BY_NAME 結構，當你使用這個 import library 聯結 PE 檔時，這個 import library 的 sections 會被加到來自 OBJ 的其他 sections 之中。由於這個「JMP 指令區段」有著與輸入函式相同的名稱，所以聯結器認爲這個指令區段其實就是輸入函式。

除了提供輸入函式的「JMP 指令區段」外，import library 還提供 PE 檔的 .idata section（或稱爲 import table）。那些片段都是來自於 .idata$ sections。總之，聯結器並不知道輸入函式（imported functions）和「位於其他 OBJ 之函式」之間的差異。聯結器只是遵循其預先調整好的規則，產生並組合 sections，然後每一件事情自然就會各就各位。

## IMAGE_THUNK_DATA DWORD

一如我稍早所說，每一個 IMAGE_THUNK_DATA DWORD 都關係到一個輸入函式。這個 DWORD 的解釋視檔案被載入與否，以及函式被以名稱輸入或以序號輸入而定。以名稱輸入是比較共同的方式。

當一個函式以序號輸入，EXE 檔的 IMAGE_THUNK_DATA DWORD 中的最高位元（0x80000000）設立。例如，考慮一個 IMAGE_THUNK_DATA，其值爲 0x80000112，放在 GDI32.DLL 陣列中。表示這 IMAGE_THUNK_DATA 將輸入 GDI32.DLL 中的第 112 號輸出（exported）函式。以序號輸入的一個問題是，微軟並未讓 Win32 API 函式的序

號在 Windows NT 、Windows 95 和 Win32s 中保持一致。

如果函式以名稱輸入，IMAGE_THUNK_DATA DWORD 就內含一個 RVA（Relative Virtual Address），指向 IMAGE_IMPORT_BY_NAME 結構。這個結構非常簡單：

**WORD　Hint**

對此欄位的最佳揣測就是輸入函式（imported function）的輸出序號（exported ordinal）。載入器利用它做爲一個建議起始值，用於對函式的二元搜尋法上面。

**BYTE[?]**

這是一個 ASCII Zstring（譯註），內放輸入函式的名稱。

IMAGE_THUNK_DATA DWORD 的最後意義(譯註:別忘了它是一個 union)是在 PE 檔被載入後才決定的，Win32 載入器使用 IMAGE_THUNK_DATA 的原始內容（可能是函式名稱也可能是函式序號）來尋找輸入函式的位址。然後，載入器就以獲得的位址改寫 IMAGE_THUNK_DATA 的內容。

> 譯註：所謂 Zstring，是指字串最後以 Null 爲結束符號。C 語言的字串便是 Zstring。所謂 Lstring，是指字串最前有一個字元計數器。Pascal 語言的字串便是 Lstring。

## 把 IMAGE_IMPORT_DESCRIPTORs 和 IMAGE_THUNK_DATAs 放在一起

你已經看過了 IMAGE_IMPORT_DESCRIPTOR 和 IMAGE_THUNK_DATA 兩種結構。把一個 EXE 或 DLL 所用到的輸入函數做出一份列表來已經不是什麼困難的事，你只要一一察看 IMAGE_IMPORT_DESCRIPTORs 表格元素（每一個元素對應一個 DLL），然後找出其相關的 IMAGE_THUNK_DATA DWORD 並適當解釋它的意義即可。**圖 8-7** 顯示 PEDUMP 的輸出結果，未列出函式名稱者，表示是以序號輸入。

```
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...
```

**圖 8-7 一個典型的 import table（來自 EXPLORER.EXE）**

# PE 檔案的輸出（exports）

「輸入一個函式」的相反動作就是「輸出一個函式」給其他的 EXE 或 DLL 使用。PE 檔把它的輸出函式相關資訊存放在 .edata。一般而言，微軟的 LINK.EXE 所產生的 PE EXE 檔並不輸出任何東西，所以它們也就不會有 .edata。TLINK32.EXE 則輸出一個符號，所以它們有一個 .edata。大部份的 DLLs 都會輸出一些函式，並因此有.edata。.edata（或

稱爲 export table）中的主要成份是一個表格，內含函式名稱、進入點位址、輸出序號。NE 檔中與此 export table 對等的東西是 entry table、resident names table 和 nonresident names table。在 NE 檔中，這些表格是 NE 表頭的一部份而不是節區或資源的一部份。

在 .edata 一開始處是一個 IMAGE_EXPORT_DIRECTORY 結構，之後緊跟著的是由該結構的某個欄位所指向的資料。IMAGE_EXPORT_DIRECTORY 結構長像如下：

**DWORD Characteristics**

此欄位沒有用途，總是爲 0。

**DWORD TimeDateStamp**

檔案被產生的時刻。

**WORD MajorVersion**
**WORD MinorVersion**

這兩個欄位沒有用途，總是爲 0。

**DWORD Name**

一個 RVA（Relative Virtual Address）值，指向一個 ASCIIZ 字串（DLL 名稱，例如 MYDLL.DLL）。

**DWORD Base**

被此模組輸出的函式的起始序號。如果輸出函式的序號是 10, 11, 12，此欄位即爲 10。

**DWORD　NumberOfFunctions**

AddressOfFunctions 陣列（稍後描述）中的元素個數。此值同時也是輸出函式的個數。通常這個值和 NumberOfNames 欄位相同，但也可以不同。

**DWORD　NumberOfNames**

AddressOfNames 陣列（稍後描述）中的元素個數。此值表示以名稱輸出的函式個數。通常（但不總是）和輸出函式的總數相同。

**PDWORD　*AddressOfFunctions**

這是一個 RVA（Relative Virtual Address）值，指向一個由函式位址所構成的陣列。我所謂的函式位址是指此一模組中的每一個輸出函式的進入點的 RVA 值。

**PDWORD　*AddressOfNames**

這是一個 RVA 值，指向一個由字串指標所構成的陣列。字串的內容是此一模組中的每一個「以名稱輸出的輸出函式」的名稱（譯註，天啊，有點拗口）。

**PWORD　*AddressOfNameOrdinals**

這是一個 RVA 值，指向一個 WORD 陣列。WORD 的內容是此一模組中的每一個「以名稱輸出的輸出函式」的序號。別忘了要加上 Base 欄位中的起始序號。

Export table 的佈局有點詭異。稍早我就說過，輸出一個函式的基本需求就是函式位址和輸出序號。如果你是以名稱輸出，那麼還需要一個函式名稱。你會以爲 PE 格式的設計者應該把這三樣資訊放在一個結構中，然後設計一個由此結構組成的陣列。結果，你卻必須在三個分離的陣列中把它們找齊！

最重要的一個陣列是 AddressOfFunctions 所指的陣列。這是一個 DWORD 陣列，每一個 DWORD 是一個輸出函式的 RVA 位址。每一個輸出函式的序號都與函式在此陣列中的位置一致。假設起始序號爲 1，那麼輸出序號爲 1 者其函式位址在此陣列中排在第

一個位置。輸出序號爲 2 者其函式位址在此陣列中則排在第二個位置，依此類推。

關於 AddressOfFunctions 陣列，有兩件重要的事情需記住。第一，輸出序號必須加上 Base 值。如果 Base 是 10，則此陣列之第一個 DWORD 應該對應序號 11，第二個 DWORD 應該對應序號 12，依此類推。第二，序號可以跳號。如果此一模組輸出兩個函式，序號爲 1 和 3，AddressOfFunctions 陣列還是需要三個元素。陣列中的元素如果不對應到一個輸出函式，那麼元素內容應該是 0。

當 Win32 載入器欲修正（fixs up）一個函式呼叫，而此函式係以序號輸入，它的工作其實很少。載入器只要以序號做爲陣列索引，找到正確的 DLL 中的 AddressOfFunctions 陣列的元素即可。當然，載入器還需考慮起始序號是否爲 1。

然而大部份函式是以名稱（而非序號）輸出。這時候前面說的另兩個結構才準備發揮它們的效用。AddressOfNames 和 AddressOfNameOrdinals 兩個陣列允許載入器根據函式名稱快速找到對應的序號。這兩個結構有相同的元素個數（記錄在 IMAGE_EXPORT_DIRECTORY 的 NumberOfNames 欄位中）。AddressOfNames 是由函式名稱的指標所構成的陣列，AddressOfNameOrdinals 陣列則是 AddressOfNames 陣列索引值所構成的一個陣列。

讓我們看看 Win32 載入器如何修正（fixs up）對於一個「以名稱輸出」的函式的呼叫。首先，載入器搜尋 AddressOfNames 陣列所指的各個字串，假設在第 3 個元素中找到目標。接下來載入器利用它獲得的這個索引值（3）尋找 AddressOfNameOrdinals 陣列中的對應元素（第 3 個元素），獲得的 WORD 值代表 AddressOfFunctions 陣列索引。最後一步就是到 AddressOfFunctions 陣列中以此索引找出函式位址。

在 C 語言中，你可以這樣獲得一個「以名稱輸出」的函式的位址：

```
WORD nameIndex = FindIndexOfString( AddressOfNames, "GetMessageA" );
WORD functionIndex = AddressOfNameOrdinals[ nameIndex ];
DWORD functionAddress = AddressOfFunctions[ functionIndex - OrdinalBase ];
```

**8-8** 顯示 export section 的格式及其中的三個陣列。**8-9** 顯示 PEDUMP 對 KERNEL32.DLL 觀察得到的 export section。

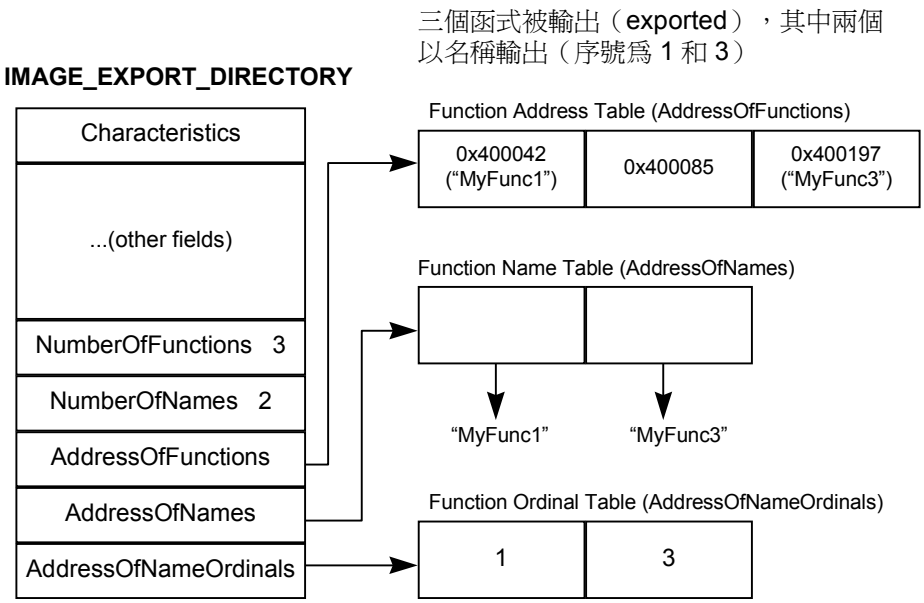



**圖 8-8　一個典型的 EXE export table**

```
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...
```

**圖 8-9　PEDUMP 對 KERNEL32.DLL 觀察獲得的 export section。**

附帶一提，如果你傾印 system DLLs（例如 KERNEL32.DLL 和 USER32.DLL）的輸出函式，你會看到許多函式只有最後一個字元的差別。例如 *CreateWindowExA* 和 *CreateWindowExW*。這是「unicode 透明化支援」的一種方式。以 'A' 結束的是 ASCII（或 ANSI）的相容函式，以 'W' 結束的則是其 unicode 版本。你的程式不需要明白指出使用哪一種版本，WINDOWS.H 中有許多 #ifdef 編譯前置動作會處理。下面就是 NT 的 WINDOWS.H 的一小片段：

```
#ifdef UNICODE
#define DefWindowProc DefWindowProcW
#else
#define DefWindowProc DefWindowProcA
#endif //!UNICODE
```

譯註：Windows 95 亦復如此處理。但 Windows 95 並不支援 unicode。

## 輸出函式的轉交（Export forwarding）

有時候會發生「經由一個 DLL 輸出一個函式，而此函式實際上位於另一個 DLL 之中」的情況。這時候 DLL 可以將此函式轉交（forward）給另一個 DLL。當 Win32 載入器遇到這樣的函式，它必須把函式呼叫修正爲對真正含有函式碼的那 DLL 的呼叫。

舉個例子就清楚多了。下面是 PEDUMP 對 Windows NT 3.5 KERNEL32.DLL 的輸出列表：

```
00043FC3       335    HeapAlloc (forward -> NTDLL.RtlAllocateHeap)
00044005       339    HeapFree (forward -> NTDLL.RtlFreeHeap)
0004402C       341    HeapReAlloc (forward -> NTDLL.RtlReAllocateHeap)
0004404D       342    HeapSize (forward -> NTDLL.RtlSizeHeap)
0004466F       442    RtlFillMemory (forward -> NTDLL.RtlFillMemory)
00044691       443    RtlMoveMemory (forward -> NTDLL.RtlMoveMemory)
000446AF       444    RtlUnwind (forward -> NTDLL.RtlUnwind)
000446CD       445    RtlZeroMemory (forward -> NTDLL.RtlZeroMemory)
```

其中的每一個函式都轉交給 NTDLL 中的另一個函式。因此，呼叫 *HeapAlloc* 事實上是呼叫 NTDLL.DLL 中的 *RtlAllocateHeap*，呼叫 *HeapFree* 事實上是呼叫 NTDLL.DLL 中的 *RtlFreeHeap*。

好，你如何告訴聯結器說某個函式需要轉交呢？唯一可能的地方是 export table（.edata）中的函式位址。這種情況下，所謂的函式位址將放的是一個 RVA，指向一個字串，內含轉交目標的 DLL 名稱和函式名稱。例如，在上面的 PEDUMP 輸出中我們看到 *HeapAlloc* 的 RVA 是 0x43FC3，而 KERNEL32.DLL 的 0x43FC3 偏移處正位於 .edata 之中。其值指向一個字串，內容爲 "NTDLL.RtlAllocateHeap"。PEDUMP 中的 *DumpExprtsSection* 函式可以示範轉交函式的鑑定方法。

雖然轉交輸出看起來不賴，微軟並沒有公開說明如何在你自己的 DLL 中使用這個性質。而且到目前爲止我也只在 Windows NT 的 KERNEL32.DLL 中看到這種用法。雖然我沒有在 Windows 95 的任何 DLLs 中看到使用這個性質，Windows 95 載入器還是有支援這項特性（我曾在第 3 章展示過）。

# PE 檔的資源

和 NE 做比較，在 PE 檔中尋找資源是比較麻煩的。資源的格式並沒有什麼改變，但是你必須在一個複雜的階層架構中移來移去，才能找到目標。

航行於資源目錄的階層架構中，有點像是航行在磁碟之中。一個根目錄，許多子目錄，子目錄又有許多子目錄。你可能在子目錄中找到檔案。檔案就類似於資源的原始資料，如對話盒面板（dialog template）。在 PE 檔中，根目錄和子目錄都是一種 IMAGE_RESOURCE_DIRECTORY 結構，長像如下：

**DWORD Characteristics**

理論上這個值可以放置資源的特性旗標。不過它總是 0。

**DWORD TimeDateStamp**

資源的產生時刻。

**WORD MajorVersion WORD MinorVersion**

理論上這裡可以放置資源的版本號碼。不過它總是 0。

**WORD NumberOfNamedEntries**

陣列之中「使用名稱的元素」的個數。此陣列緊跟在這個結構之後。請看稍後的 DirectoryEntries[] 解說。

**WORD NumberOfIdEntries**

陣列之中「使用整數 ID 的元素」的個數。此陣列緊跟在這個結構之後。請看稍後的 DirectoryEntries[] 解說。

### IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[ ]

這個欄位並非正式的 IMAGE_RESOURCE_DIRECTORY 結構的一部份，而是緊跟在它之後的一個由 IMAGE_RESOURCE_DIRECTORY_ENTRY 組成的陣列。其中的元素個數是 NumberOfNamedEntries 和 NumberOfIdEntries 的總和。「帶有名稱之元素」統統位在「帶有 ID 之元素」的前面。

> 譯註：在 Windows 95 的 WINNT.H 中，並未視 DirectoryEntries 爲有用欄位。

一筆目錄資料（directory entry）可以指向一個子目錄（也就是另一個 IMAGE_RESOURCE_DIRECTORY）或是一個 IMAGE_RESOURCE_DATA_ENTRY。後者用來描述在檔案什麼地方找出資源的原始資料。一般而言，在你找到 IMAGE_RESOURCE_DATA_ENTRY 之前最少需經過三層目錄。最上層目錄（只有一個）一定在 .rsrc 一開始處獲得。其子目錄對應於檔案中的資源型態。如果 PE 檔內有三筆資源：對話盒、字串表格和選單，那麼就會有三個對應的子目錄。每一個這樣的「型態子目錄」又會有一個「ID 子目錄」。一種型態的資源如果產生出三份實體，就有三個「ID 子目錄」。以上例而言，如果程式中有四個對話盒，那麼就應該有四個「對話盒 ID 子目錄」。每一個 ID 子目錄中要不是有名稱，就是有資源 ID，用以識別 RC 檔中的資源。**圖 8-10** 顯示資源目錄的階層架構。

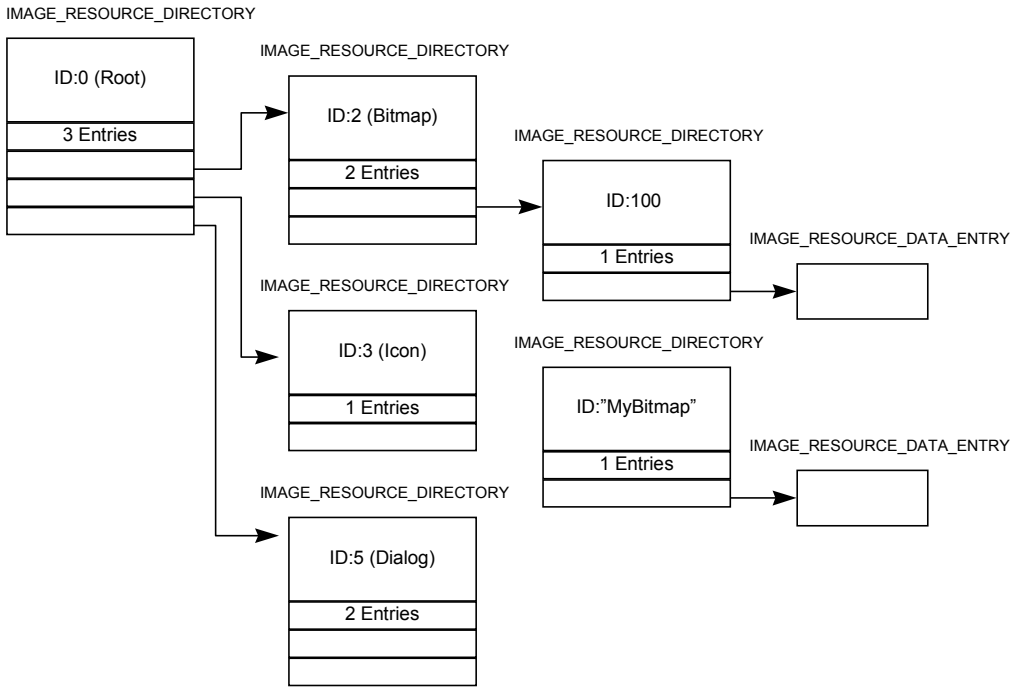



**圖 8-10　一個典型的 PE 檔資源架構**

**圖 8-11** 是 PEDUMP 對於 Windows NT 3.5 的 CLOCK.EXE 的資源傾印輸出。請注意縮排的第二層，你將看到 icons、menus、dialogs、stringtables、group icons、version 等等資源。而在第三層，有兩個 icons（ID 爲 1、2），兩個 menus（名爲 CLOCK 和 GENERICMENU），兩個 dialogs（一個名爲 ABOUTBOX，另一個 ID 爲 0x64）。縮排第四層告訴你，icon 1 的資料位於 RVA 0x9754，有 0x130 個位元組。CLOCK menu 的資料位於 RVA 0x952C，有 0xEA 個位元組。

```
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
```

**圖 8-11　CLOCK.EXE 的資源階層架構**

每一筆資源目錄資料是一個 IMAGE_RESOURCE_DIRECTORY_ENTRY（嘿，這名稱可真長）。IMAGE_RESOURCE_DIRECTORY_ENTRY 的格式如下：

**DWORD Name**

這個欄位內含一個整數 ID，或是一個指標，指向一個內含字串名稱的結構。如果最高位元（0x80000000）是 0，這欄位就被解釋爲一個整數 ID。如果最高位元不是 0，剩餘的 31 個位元就是 IMAGE_RESOURCE_DIR_STRING_U 結構的偏移位置。該結構內含一個 WORD 字元計數器，然後是代表資源名稱的 unicode 字串。是的，即使 PE 檔適用於 non-Unicode 的 Win32 平台，這裡還是使用了 unicode。欲將 unicode 轉換爲 ANSI 字串，請參考 *WideCharToMultiByte* 函式。

**DWORD OffsetToData**

這欄位若不是另一個資源目錄的偏移位置，就是指向特定資源實體的一個指標。如果最高位元（0x80000000）設立，表示這個目錄項目指向另一個子目錄，而剩餘的 31 個位元就是另一個 IMAGE_RESOURCE_DIRECTORY 的偏移位置（從 resource section 算起）。如果最高位元沒有設立，剩餘的 31 個位元就是某一個 IMAGE_RESOURCE_DATA_ENTRY 的偏移位置（從 resource section 算起）。IMAGE_RESOURCE_DATA_ENTRY 結構內含資源原始資料的位置、大小、以及 code page。

如果要再進一步挖掘資源格式，我就必須探討各種類型的資源（dialog 啦、menu 啦等等）。這些題目很容易就會填滿另一大章，而且我也應該多拯救幾棵樹才是。如果你有興趣，請看 Win32 SDK 中的 RESFMT.TXT 檔，那裡面對於所有資源型態的格式都有詳細的描述。PEDUMP 可以顯示資源的階層架構，但是沒有分解個別的資源實體。

# PE 檔的基址重定位（Base Relocations）

當聯結器產生一個 EXE 檔，它假設這個檔案會被載入記憶體的某處，並且把 code 和 data 的相關假設位址都寫入 EXE 檔案中。如果可執行檔最終被載入到虛擬位址空間的另一個位址，沒有按預期來，聯結器所登記的那個位址就是錯誤的。儲存在 .reloc 的資訊就是用來幫助 PE 載入器修正載入模組的位址。如果載入器能夠把模組載入到預定位址

上，.reloc 就可以棄而不用。.reloc 中的資料項被稱爲「基底重定位（base relocations）資料項」，原因是它們的用途視被載入模組的基底位址而定。

和 NE 格式的重定位方式不同，PE 檔的作法十分簡單。它們並不參考到外部 DLLs 或模組中的其他 sections，而是把 image（譯註）中所有可能需要修改的位址串成一個串列。

> 譯註：檔案中的模組資料被載入記憶體後，我們稱其爲模組的 "image" 。我保留這個原文，不譯它。

下面是一個例子。假設有一個可執行檔，基底位址是 0x400000。在這個 image 偏移位置 0x2134 處是一個指標，指向一個字串。字串始於實際位址 0x404002 處，所以指標內容應該是 0x404002。你可以把檔案載入，但是載入器決定把它映射到實際位址 0x600000 處。聯結器假設的基底位址和實際載入的起始位址之間的差額稱爲 delta。此例之 delta 爲 0x20000。整個 image 的位置提高了 0x20000，其中的字串當然也是（現在應該是 0x604002）。所以指向字串的指標就錯誤了，delta 應該加到指標值中。

爲了讓 Windows 載入器有能力做這樣的調整，可執行檔內含許多個「基底重定位資料項」，給那些存放指標的位置（本例爲 0x2134）使用。載入器必須把 delta 加到各個位址上。本例之中載入器應該把 0x20000 加給原來的指標值（0x404002），並將結果 0x604002 寫回原處。圖 **8-12** 顯示這個過程。

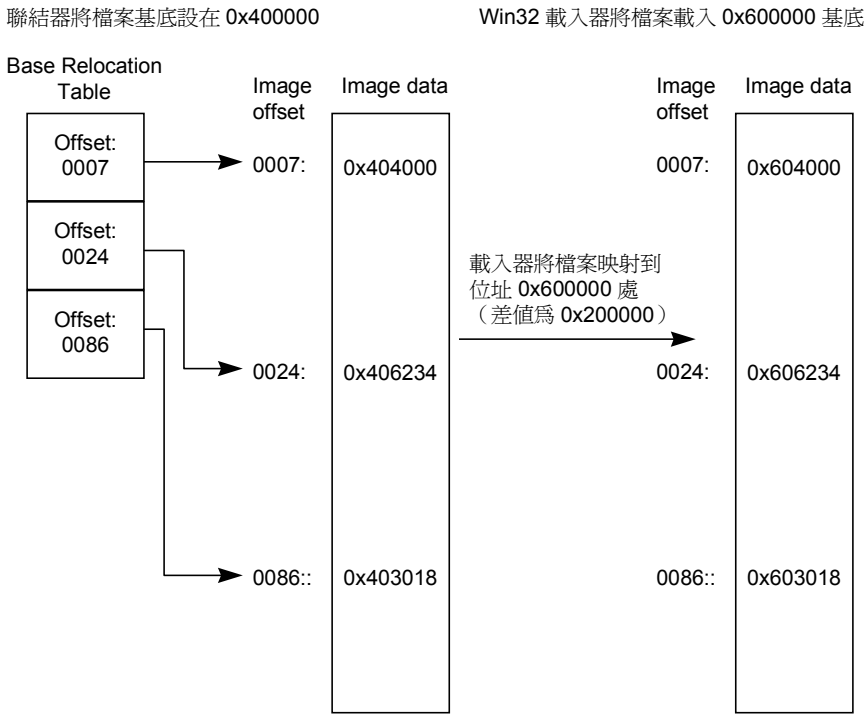



**圖 8-12　PE 檔案的基底重定位動作。**

「基底重定位資料項」的格式有點兒奇怪。它們被包裝爲一系列連續區段，長短不一。每一個區段描述 image 中的一個 4K page 的重定位資訊。它們以一個 IMAGE_BASE_RELOCATION 結構做爲開始，格式如下：

**DWORD　VirtualAddress**

此一欄位內含這些個「基底重定位資料項」的起始 RVA 值。每一個「基底重定位資料項」的偏移位置必須加上此值才能夠構成一個真正的 RVA，指向「基底重定位資料項」。

### DWORD　SizeOfBlock

結構大小，再加上所有跟隨在後的「基底重定位資料項」（都是 WORDs）。爲了決定區塊中的「基底重定位資料項」的個數，先把此值減去 IMAGE_BASE_RELOCATION 結構大小（8 個位元組），再除以 2（WORD 大小）。如果此欄位爲 44，就表示有 18 個「基底重定位資料項」：

```
( 44 - sizeof(IMAGE_BASE_RELOCATION) ) / sizeof(WORD) = 18
```

### WORD　TypeOffset

這並不是單獨一個 WORD，事實上它是一個 WORDs 陣列。陣列元素個數可由上一個式子計算獲得。每一個 WORD 的最底部 12 個位元代表「基底重定位資料項」偏移位置，但必須再加上「基底重定位資料項」區塊表頭的 VirtualAddress 欄位值。最高的 4 個位元是「基底重定位資料項」的型態。對於在 Intel CPUs 中執行的 PE 檔，你將看到兩種型態：

- 0（IMAGE_REL_BASED_ABSOLUTE）：此一「基底重定位資料項」無意義，只是用來充數而已，使所有「基底重定位資料項」的總共大小成爲 DWORD 的倍數。
- 3（IMAGE_REL_BASED_HIGHLOW）：把 delta 值加到欲計算的 RVA 值去。

另外還有其他型態，在 WINNT.H 中定義。它們大部份是給 i386 以外的 CPU 使用。

**範例 8-13** 描繪出一些「基底重定位資料項」，這是 PEDUMP 的輸出結果。請注意圖中的 RVA 值已經被 IMAGE_BASE_RELOCATION 結構中的 VirtualAddress 欄位校正過了。

```
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 chunk omitted...
```

**圖 8-13　一個 EXE 檔的「基底重定位資料項」（base relations）**

## COFF 符號表格 （略）

如果你只對被作業系統使用的 PE 檔案內容感興趣，你可以跳過這一節和下兩節（「COFF 除錯資訊」和「COFF 行號表格」）。你可以直接跳到「PE 檔和 COFF 檔之間的差異」那一節繼續閱讀。

## COFF 除錯資訊 （略）

## COFF 行號表格 （略）

譯註：此處將原文書中的「COFF 符號表格」、「COFF 除錯資訊」以及「COFF 行號表格」三節略去。這三節都與除錯資訊有關，在原文書的 p.605~p.614，共約 9 頁。這是本書唯一略去原文內容的部分。

# PE 檔和 COFF 檔之間的差異

前面的討論之中，我已經告訴你，在 COFF OBJ 檔和 PE 檔之間有許多共通的結構和表格。兩者一開始都有一個 IMAGE_FILE_HEADER，後面緊跟著一個 section table。兩者共享相同的行號資訊格式以及符號表格格式 -- 雖然 PE 檔可以有額外的 non-COFF 符號表格。存在於 PE 和 COFF OBJ 之間的共通性，其規模可以從 PEDUMP 原始碼看得出來。這個程式中最大的一個檔案是 COMMON.C，內含所有可以共用於 PE- 和 OBJ-檔案的傾印函式。

兩個檔案格式的相似性並非偶然。這樣設計的目的是爲了讓聯結器的工作降至最少。理論上，從單獨一個 OBJ 檔產生一個 EXE 檔，應該只是安插一些表格，並修改一些偏移值而已。你可以因此想像 COFF 檔是 PE 檔的胚胎。只有一些東西不一樣，我把它們列在下面：

- COFF OBJ 一開始就是 IMAGE_FILE_HEADER 結構，沒有 DOS stub。而且該結構一開始也沒有 PE 簽名字元。
- OBJ 檔沒有 IMAGE_OPTIONAL_HEADER。PE 檔中的這個結構則是緊跟在 IMAGE_FILE_HEADER 之後。有趣的是，某些存放於 COFF LIB 檔內的 OBJs 倒是有此結構。
- OBJ 檔沒有「基底重定位資料項」。但是它們有正規的符號修正表。我沒有討論 COFF OBJ 中的重定位方式，因爲那十分艱澀。如果你要挖掘這一部份，section table 資料項中的 NumberOfRelocations 和 PointerToRelocations 兩個欄位就指向每一個 section 的重定位資訊。這些重定位資訊是 IMAGE_RELOCATION 結構（定義於 WINNT.H）所組成的陣列。PEDUMP 可以顯示 OBJ 檔的重定位資訊 -- 如果你選用適當的命令列選項的話。
- OBJ 檔中的 CodeView 資訊存放在 .debug$S 和 .debug$T 兩個 sections 中。當聯結器處理 OBJ 檔案，它不把這兩個 sections 放到 PE 檔中。而是收集所有的 sections 並建立一個簡單的符號表格，放在檔案尾端。這個符號表格並不是正式的 section（也就是說在 PE 檔的 section table 中沒有和它相關的資料項）。

# COFF LIB 檔

一旦你了解 COFF OBJ 檔，使用 COFF LIB 檔就不會太困難。COFF LIB 檔基本上只是 COFF OBJ 的集合，再加上一些初始資料，讓你能夠很快速地搜尋 LIB 中的某個 OBJ 的位置。有些技術文件把 COFF LIB 說成是一個 archives，這點我必須先告訴你。

所有的 LIB 檔以相同的 8 個簽名字元做爲開始。這 8 個字元定義在 WINNT.H 中：

```
#define IMAGE_ARCHIVE_START      "!<arch>\n"
```

檔案的剩餘部份是一大堆可變長度的記錄，每一個記錄以一個 IMAGE_ARCHIVE_MEMBER_HEADER 打頭陣：

```
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;
```

每一個 IMAGE_ARCHIVE_MEMBER_HEADER 對應到的，要不是 LIB 中的一個 OBJ，要不就是一些特殊記錄的集合。這些特殊記錄來自 LIB 的最前頭，爲的是讓 LIB 能夠快速尋找 OBJ。在 IMAGE_ARCHIVE_MEMBER_HEADER 之後的資料是一個個的記錄（稱爲 archive member records）。大部份的 archive member records 就是 OBJ 檔的內容。事實上當你傾印 LIB 檔，PEDUMP 呼叫的傾印函式也用於處理 OBJ 檔。圖 8-16 顯示 LIB 檔案格式。

讓我們仔細看看 IMAGE_ARCHIVE_MEMBER_HEADER 的欄位：

**BYTE　　Name[16]**

這是 archive member 的名稱。如果一個 '/' 出現在一個 ASCII 字串最後（例如 "FOO.OBJ/"），那麼在 '/' 之前的就是 member 的名稱。如果字串是以 '/' 開始並緊跟著十進位數字（例如 "/104"），這個號碼就是 archive member 名稱的偏移位置。所謂偏移是指對 LIB 中的 Longnames member 而言。因此 "/104" 的意思就是表示 member 名稱出現在 Longname 區域的第 104 個位元組。

另外還有一些特殊名稱，用來識別特殊的 archive members：

```
#define IMAGE_ARCHIVE_LINKER_MEMBER         "/               "
#define IMAGE_ARCHIVE_LONGNAMES_MEMBER      "//              "
```

至於那些記錄在 import library 中的 OBJ 檔，此欄位放的是 DLL（內含被輸入函式）的名稱。

| “!<arch>\n” |
| --- |
| IMAGE_ARCHIVE_MEMBER_HEADER (“/”) |
| First Linker Member data |
| IMAGE_ARCHIVE_MEMBER_HEADER (“/”) |
| Second Linker Member data |
| IMAGE_ARCHIVE_MEMBER_HEADER (“//”) |
| Longnames Member Data |
| IMAGE_ARCHIVE_MEMBER_HEADER (“FOO.OBJ”) |
| OBJ file data |
| IMAGE_ARCHIVE_MEMBER_HEADER (“/104”) |
| OBJ file data |
| ... |

**圖 8-16　COFF LIB 檔的格式佈局**

**BYTE Date[12]**

Member 被產生的時刻。以 ASCII 十進位格式儲存。

**BYTE UserID[6]**

使用者的 ID，以 ASCII 十進位格式儲存。是一個以 NULL 爲結束字元的字串。

**BYTE GroupID[6]**

Group ID，以 ASCII 十進位格式儲存。是一個以 NULL 爲結束字元的字串。

**BYTE Mode[8]**

檔案模式，以 ASCII 十進位格式儲存。總是爲 0。

**BYTE Size[10]**

跟隨在後的 member 資料的大小，以 ASCII 十進位格式儲存。至於資料的格式，則視其型態而定（請參考前面的 Name 欄位）。

**BYTE EndHeader[2]**

ASCII 字串 \n。

## Linker members

每一個 LIB 檔有兩個 Linker member sections，作用像是檔案中的其他內容的資訊表格。這兩個 member 都有名稱 '/'，可以經由出現在檔案中的前後次序區分之。第一個 linker member 是第一個 archive member 的名稱再加上 '/'，第二個 linker member 是第二個 archive member 的名稱再加上 '/'。

兩個 Linker member 基本上都用來列出 LIB 檔中的公開符號，以及「內含公開符號之 OBJ members」的偏移位置。兩個 Linker member 有著不同的格式。爲什麼同一份資訊要有兩份拷貝？第一個 Linker member 所放資訊是以 OBJ 出現次序排列，導至無法進

行最佳搜尋。第二個 Linker member 所放資訊是以符號的字母次序排列，因此對聯結器而言它是比較有用的。微軟的文件上說，聯結器會忽略第一個 Linker member，使用第二個 Linker member。

第一個 Linker member 有以下格式：

**DWORD　NumberOfSymbols**

這是 LIB 之中公開符號的個數。這是一個 big-endian 格式（譯註），反應出「COFF 並不是繼承自 i386 機器」這一點。PEDUMP 的 LIBDUMP.C 中的 *ConvertBigEndian* 函式可以處理 big-endian 和 little-endian 之間的轉換。

> 譯註：所謂 big-endian 是把「most significant 位元組」放在前面，例如 Motorola 和 IBM 370。所謂 little-endian 是把「least significant 位元組」放在前面，例如 Intel 和 DEC VAX。所以如果你以工具軟體如 PC-Tools 或 DEBUG.EXE 觀察 Intel 機器的記憶體內容或是檔案內容，螢幕上出現的 "00 40" 其實真正是 "40 00"。

**DWORD　Offsets[NumberOfSymbols]**

這是一個檔案偏移值構成的陣列，每一個偏移值指向另一個 archive members。偏移值是以 big-endian 形式出現。每一個 member 都是一個 OBJ-type member。這個陣列中的每一個元素對應於相同排列次序的陣列（請看下一個欄位），後者的元素是 ASCII 字串，代表符號名稱。

**BYTE　StringTable[?]**

這是一系列沒有被打斷的 C-style 字串（譯註：所謂 C-style 字串就是 ZString，也就是以 0 爲結束字元的字串）。

基本上，Offsets 陣列中的每一個元素都對應到一個公開符號，而其名稱出現於 StringTable 之中。例如，Offsets 的第三個元素關係到 StringTable 的第三個字串。看看 PEDUMP 的輸出，你會比較清楚：

```
First Linker Member:
  Symbols:     00000006
  MbrOffs      Name
  --------     ----
  00000180     _DumpCAP@0
  00000180     _StartCAP@0
  00000180     _StopCAP@0
  ...
```

第二個 Linker member 的格式比較複雜，因爲要提供快速搜尋的資料。格式如下：

**DWORD NumberOfMembers**

LIB 檔案中的 OBJ archive members 個數。

**DWORD Offsets[NumberOfSymbols]**

「其他 archive members 的檔案偏移位置」所組成的陣列。與第一個 Linker member 不同的是，這些偏移位置都是機器原生（native）格式，也就是說，如果在 i386 機器上，它將是 little-endian 形式。

**DWORD NumberOfSymbols**

這是 StringTable 中的公開符號個數，同時也就是整個 LIB 的公開符號個數。

**WORD Indices[NumberOfSymbols]**

這個陣列內含「從 1 起算」的 Offsets 陣列索引。這個陣列與 StringTable 陣列平行運作。

**BYTE StringTable[NumberOfSymbols]**

這是一系列沒有被打斷的 C-style 字串。

爲了以第二個 Link member 找出使用某一公開符號的 OBJ，聯結器首先尋找 StringTable 並獲得其相關索引值，然後聯結器以此索引值查詢 Indices 陣列。最後，聯結器再將 Indices 陣列元素的值（WORD）減 1，當做是 Offsets 陣列的索引。找到 Offsets 陣列

元素（DWORD）之後就是 OBJ 的偏移位置了。PEDUMP 的 LIBDUMP.C 中的 *DumpSecondLinkerMember* 函式實作出這樣的尋找程序。

## The Longnames member

存放在 Longnames archive member section 中的資料其實只是一些 C-style 字串，一個跟著一個。如果一個字串太長，沒辦法塞入 IMAGE_ARCHIVE_MEMBER_HEADER 的 Name 欄位（16 位元組）的話，它就會被放到這裡來。這種情況下，Name 欄位會放置一個 '/' 字元，後隨著一個字串偏移位置（Longnames section 的偏移位置）。偏移值是以 ASCII 十進位值表示。

# 摘要

由於 Win32 的降臨，微軟對於 OBJ 和 EXE（DLL）格式做了全盤的改變。這些改變建立在其他作業系統上的既成結果，使微軟得以節省時間。改頭換面的主要目的是爲了強化在不同平台之間的可移植性。COFF OBJ 格式在 Win32 誕生之前就存在了。PE 格式則是 COFF 格式的延伸，使用於 Win32 平台上。

OBJ 和 EXE（DLL）檔案的有用部份是從 IMAGE_FILE_HEADER 結構展開。在那結構之下是一個 section table，此一表格內含檔案中所有 sections 的屬性和位置。所謂 section 是一群「邏輯上應該放在一起的資料或程式碼」。爲了讓資料的搜尋能夠更快速，PE 檔內含一個 data directory，指向檔案中各個有用的區段（例如檔案的 export table）。除了表頭、section tables，以及 sections 的原始資料外，COFF OBJ 檔以及 PE 檔也內含符號名稱和行號資訊。這些資訊存放在檔案尾端。

# 尋幽訪勝靠自己

和本書其他各章都不一樣，這一章並不把焦點放在 Windows 95 的動作或架構上。而是希望描述一些比較基本的自力探索方法。請你想想這句箴言：『給某人一條魚，他只夠吃一天；教某人釣魚，他可以吃一輩子』。其他各章給你的是魚，這一章則是教你如何釣「Windows 秘密」這條魚。

當然，這裡所學得的技術與技巧，也可以施行於裝置驅動程式或應用軟體身上。

理想中，你應該在說明文件裡頭找到所有的作業系統資訊，使你能夠以黑盒子的心情看待作業系統。你不需要了解作業系統的內部行爲或其資料結構，因爲了解並使用公開的介面就已經足夠寫任何程式、函式庫、驅動程式了。

如果完整的說明文件並不存在（噢，理想世界開始解構了），作業系統的原始碼可以做爲資訊的另一來源。雖然你必須看其他人寫的碼（聽起來令人毛骨悚然），但是作業系統的每一個問題都可以從其原始碼中獲得解答。事實上，Eric S. Raymond 的 ***The New Hacker's Directory*** 書中有一詞爲 UTSL，就是 “Use The Source Luke” 的縮寫（譯註）。

譯註："Use The Source Luke" 是 "Use The Force Luke" 的演繹。"Use The Force Luke"，「使用原力，路克」，是電影「星際大戰」的一句台詞（如果我沒記錯的話。18 歲以下的朋友不可能看過這部電影，28 歲以下的朋友對這部電影可能不復記憶）。路克是劇中男主角的名字，他與星際中的惡魔（黑元帥？）奮戰時，路克的導師要求他使用原力 -- 生命中的潛能吧我想！

茲列出 ***The New Hacker's Directory*** 書中對 UTSL 一詞的說明（本書在臺灣沒得買，我是旅遊紐約時順便買的）：

UTSL // [UUNIX] n. On-line acronym for "Use the Source, Luke" (a pun on Obi-Wan Kenobi's "Use the Force, Luke!" in Star Wars) --- analogous to RTFS, but more polite. This is a common way of suggesting that someone would be better off reading the source code that supports whatever feature is causing confusion, rather than making yet another futile pass through the manuals, or broadcasting questions on USENET that haven't attracted wizards to answer them.

看倌可能不知 RTFS 是什麼，那是 "Read the Fucking Source" 的縮寫。噢，粗話上場了，我老天！

在 UNIX 世界中，取得 UNIX 原始碼是很平常的一件事。不幸的是，對於我們這些在 Windows 環境下工作的人，原始碼不易獲得。是的，有一些 Windows 95 的碼出現在 DDK 之中。大部份時候，程式設計的主要問題都不是 SDK 或 DDK 提供的原始碼所能解決的。微軟的文件雖然在最近幾年有顯著的改善，但還是存在著許多「黑洞」，唯有作業系統的原始碼才能夠填滿它。

不足的文件和未可運用的原始碼，並不是你在與作業系統共處時唯一遭遇的問題。你的程式可能需要和另一個未知的程式互動。典型的例子就是程式員被迫花費大量時間，嘗試確定微軟的 Excel 送出什麼 DDE 訊息、以什麼順序送出。另一個例子是在 Windows 3.1 之中執行某個程式時，沒辦法再跑其他程式。這可能是因爲該程式吸光了 1MB 以內的所有記憶體。

我要在這一章討論下面三種 Windows 程式員用來探索奧秘的方法：

- 檔案傾印工具（file dumping utilities）
- API 函式及 Windows 訊息的窺視工具（spy programs）
- 反組譯（disassembly）

我會一節一節地描述常用工具，並且示範如何使用這些工具獲得有用的資訊。最後一節關於反組譯技術特別詳盡，因爲反組譯被視爲一種神奇魔法，很少被公開討論。閱讀組合語言列表並走進除錯器中的碼，看似有許多藝術在裡頭，其實只要知道編譯器的程式碼產生樣式（code-generation patterns）即可。

這一章描述的是一般性的探索觀念，登堂者與入室者應該都會感到助益。本章最後一節則是高級技巧與計謀，適合給高段的探險家閱讀。畢竟，我在這一條艱辛道上一路走來，而大多數的你們可能只是對我的成果（而非過程）感到興趣。

雖然 Windows 95 是一個 Win32 作業系統，其中極重要的部份卻還是使用 16 位元碼以及 NE 檔案格式。因此描述 Win16 的探索技術也是有必要的。雖然某些探索技術繼續存在於 Win32，但其中可能有明顯的變化。因此，我也會涵蓋 Win32 程式的工具及其使用技術。

## 探險行動概觀

學習一段碼，最簡單也最容易起頭的方法，就是使用檔案傾印工具，如 Borland 的 TDUMP，Microsoft 的 EXEHDR 和 DUMPBIN，或本書第8章的 PEDUMP。這些工具可以告訴你一個程式使用了哪些 DLLs，哪些 API 函式，但是沒有辦法告訴你更內部的演算法和資料結構。檔案傾印動作就像是，你透過前窗，監視街角房子的動靜，但是你沒辦法獲得屋內更多的活動訊息。

如果要更精巧地刺探一個程式的活動，你可以使用 spy 工具。Microsoft 的 Windows SDK 附有一個 SPY 程式，Borland C++ 也附有一個 WinSight 程式，都可以顯示一個程式送

出和接收的訊息。Nu-Mega 公司的 BoundsChecker 產品以及 Periscope 公司的 WinScope 產品，也能夠觀察你的程式呼叫了哪些作業系統 API 函式。本書第 10 章也提供了一個擴充的 Win32 API spy 程式。有了這些資訊再加上一點點努力，你就幾乎可以理解任何一塊碼。稍後我會示範如何刺探一個程式的活動。以前面我所做的比喻而言，spy 就像是在房子外攔截信件或在房子內截聽電話一樣。

最後，當你需要知道程式或動態聯結函式庫的內部演算法和資料結構，你必須使用反組譯技術。雖然你可以利用除錯器做某種程度的反組譯，但恐怕還得借助以檔案爲根據的反組譯工具才行，像是 V-Communication 公司的 Sourcer 或 Eclectic Software 公司的 Win2Asm。加上私人標記以及將輸出列表格式化，是這類反組譯工具比一般除錯器強的地方。如果再以前面所提的比喻來說，反組譯就像是踢破前門上膛攻堅的行動。我將在「以反組譯工具探險」那一節描述各種反組譯工具和技術。

## 以檔案傾印（Dumping）工具探險

通常對一個程式的探險動作，第一步就是傾印出其檔案內容。這個步驟使你能夠迅速知道你的探險對象的檔案型態，以及它可能的用途。表 **9-1** 列出幾個常見的檔案解剖工具及其能力。

如果你的開發工具是 Borland C++，試試使用 BIN 子目錄下的 TDUMP.EXE。如果你的開發工具是 16-bit Microsoft C++，EXEHDR 或許是你的選擇。如果你使用 Microsoft Win32 SDK 或 Visual C++，BIN 子目錄下的 DUMPBIN 可以處理 Portable Executable（PE）檔，以及微軟 32 位元編譯器所產生的 COFF format OBJs 檔。如表 **9-1** 所示，沒有一個程式可以通吃所有功能。所以同時擁有數個工具在手是不錯的主意。TDUMP 和 DUMPBIN 是絕佳組合。

**表 9-1　幾個常見的檔案解剖工具的能力。**

| 檔案型態 | DUMPBIN | DUMPEXE | EXEHDR | TDUMP |
|---|---|---|---|---|
| MZ 檔案（DOS） | | X | X | X |
| NE 檔案（Win16） | | X | X | X |
| PE 檔案（Win32） | X | X | | X |
| 除錯資訊 | X | X | | X |
| 反組譯 | X | | | |
| OBJs | X | | | X |
| 資源（Resources） | X | X | | X |

註：DUMPBIN 來自 Microsoft Win32 SDK 和 Visual C++
DUMPEXE 來自 Symantec C++
EXEHDR 來自 Visual C++
TDUMP 來自 Borland C++

你從檔案傾印工具的輸出中所能獲得的最有用資訊通常是這個可執行檔所用到的 DLL 和 API 函式。通常知道一個程式使用某些函式之後，往往就能揭開許多困惑。例如，Windows 3.0 並沒有公開說明如何改變桌面壁紙（desktop wallpaper），但是控制台（control panel）中的程式卻可以改變之。這能力一定存在於 Windows 的某個角落。只要對著 Windows 3.0 的控制台程式執行 TDUMP 或 EXEHDR，你就會看到這個程式使用了一個未公開函式 *SetDeskWallPaper*（Windows 3.1 有一公開函式 *SystemParameterInfo* 取代了其功能）。

欲找出 16 位元 NE 檔或 DLL 所使用的函式，需要兩個步驟。由於 NE 檔案中並未含有來自其他 DLLs 的所有函式的列表，所以第一個步驟是找出可執行檔節區中的 fixup（待修正）資料。如果你的工具是 EXEHDR，你必須加上 /VERBOSE 選項才能取得 fixup 資料。以下是以 TDUMP 處理 Windows 3.1 CALC.EXE 的部份結果：

```
...
PTR     0AD9h   GDI.91
PTR     0121h   GDI.93
PTR     00EAh   USER.89
PTR     0223h   USER.90
PTR     04ADh   USER.91
PTR     1DCAh   USER.92
...
```

其中所含的重要資訊是模組名稱以及輸入序號（import ordinal）。以本例而言，輸入 6 個函式，2 個來自 GDI，4 個來自 USER。但是，像 GDI.91 這樣的函式名稱沒有什麼用，所以第二步驟就是把模組名稱（GDI）和函式序號（91）轉換爲真正的函式名稱。

當一個 EXE 或 DLL 輸出（export）函式，函式名稱及其相關序號儲存在可執行檔中，以及 import library 中。由於並沒有什麼簡易的作法可以傾印 16 位元 import library 的內容，所以我示範的是直接從 DLL 取出函式名稱。回到我們的第二個步驟來，我們必須先了解 GDI.91 的意義。那意思是傾印出 GDI.EXE 的內容，尋找輸出序號爲 91 的函式。下面顯示的是對著 GDI.EXE 執行 TDUMP 之後，Non-Resident Name Table 的輸出片斷（如果使用 EXEHDR 工具，會在"Exports" 節區中發現類似的資訊）：

```
Non-Resident Name Table     offset: 0C41h
    Module Description: 'Microsoft Windows Graphics Device Interface'
    Name: GETWINDOWEXTEX              Entry:   474
    ....
    Name: GETTEXTEXTENT               Entry:    91
```

我們看到 *GetTextExtent* 正是 GDI.91。事實極爲明顯，CALC.EXE 呼叫了 *GetTextExtent*（GDI.91）。其他的函式名稱也可以經由這兩個步驟的不斷重複而決定。請注意，這個方法沒辦法發現「程式執行時利用 *GetProcAddress* 而聯結的函式」。那種情況唯有靠反組譯才能解決。

要找出 32 位元 PE 檔所輸入（import）的函式，比較容易些。對著一個 Win32 程式執行 DUMPBIN 或第 10 章提供的 PEDUMP，就可以獲得 PE 檔的輸入函式列表。這個表看似單純，實則檔案中的 imports section 十分複雜，請參考第 8 章的介紹。輸入函式的列表甚至以模組名稱排序。下面這份列表就是對著 Windows NT 的 USER32.DLL 執行 DUMPBIN 的結果：

```
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
```

含有函式名稱的每一行的第一個號碼，是所謂的提示序號（hint ordinal）。Win32 作業系統雖然是以函式名稱來做輸入（import）行爲，但序號可以加快處理過程。它給予載入器一個提示，告訴它從哪裡開始其二元搜尋 -- 搜尋那些輸出（export）函式的名稱。

觀察 USER32.DLL 的 import table 之後，我們發現它呼叫 KERNEL32 函式如 *GetAtomNameW* 和 *LocalReAlloc*。有趣的是，請注意，當 Windows NT USER32.DLL 有機會呼叫一個 ASCII API 或一個 unicode API 時，它呼叫的是 unicode 版本（*GetAtomNameW*、*LoadLibraryW* 等等）。這符合微軟所宣稱的「Windows NT 內部使用 unicode 字串」。

這一份輸出也顯示 NT 的 USER32.DLL 使用許多來自 NTDLL.DLL 的函式。NTDLL.DLL 裡面全都是未公開函式。有趣的是，NT 的 KERNEL32.DLL 非常依賴 NTDLL.DLL 中的函式。NTDLL.DLL 也存在於 Windows 95，但後者的 KERNEL32.DLL

就沒有使用 NTDLL.DLL 的跡象。事實上恰恰相反，Windows 95 的 NTDLL.DLL 非常依賴 KERNEL32.DLL。

雖然，知道一個 EXE 或 DLL 使用哪些個 API 函式頗有用處，但反方向也同樣重要。檔案傾印工具可以告訴你某個 DLL 輸出哪些函式給其他 EXEs 或 DLLs 使用。輸出函式（export functions）通常是 DLL 的目的和能力的洩露點。有時候光是函式名稱就足以讓你猜出一個未公開函式的參數。另些時候也許你需要一個反組譯器，才能瞭解如何呼叫未公開函式。

以下列表是 TDUMP 對著 SHELL.DLL（一個 16 位元 NE 檔）的輸出結果。這個 DLL 來自微軟的 Word for Windows 2.0，但它的 APIs 並沒有公開：

```
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
```

其中一些函式的名稱給了我們十分明白的線索，告訴我們這個 DLL 做些什麼事，以及如何呼叫它。例如 *SpellVer* 大概不需要任何參數，大概會傳回一個版本號碼，放在 AX 或 DX:AX 之中。寫個小程式測試一下，太簡單了，你只要以 *LoadLibrary* 載入 SHELL.DLL，呼叫 *GetProcAddress* 取得 *SpellVer* 的函式位址，然後呼叫之即可。

最新報導：我的確寫了這麼一個小程式，測試 *SpellVer*。我發現它總是傳回 0。經由反組譯，我發現它需要三個指向 WORD 的遠程指標（LPWORD），用來讓它填寫資料。從這裡我們得到一個教訓：雖然檔案傾印是最簡單的探險方式，但它並不總是能夠給你足夠的資訊。

回頭再看看 SPELL.DLL 的其他函式。請注意有 *SpellInit*、*SpellCheck*、*SpellTerminate* 等函式。所以，這個 DLL 或許希望初始化、檢查某些文字、然後停工。我們所不知道的是這些函式需要什麼參數。再一次，這又非反組譯不爲功了。

如果你想看看某個產品不同版本之間有些什麼改變，比較其對應的 DLLs 的輸出函式（export functions）是個簡單的開始。**表 9-2** 顯示 KRNL3386.EXE 的輸出函式在 Windows 3.1 和 Windows 95 之間的差異。爲了獲得這份資料，我利用 EXEHDR 傾印出 Windows 3.1 的 KRNL386.EXE 和 Windows 95 的 KRNL3386.EXE，把結果分別儲存到不同的檔案，然後再把每一個函式以字母排序，最後再執行 DIFF 程式，顯示其間的差異。

**表 9-2　KRNL386.EXE 的輸出函式在 Windows 3.1 和 Windows 95 中有些差異**

| | KRNL386 的輸出函式（export functions） |
|---|---|
| Windows 95 新增函式 | CALLPROC32W, CREATEDIRECTORY, DELETEFILE, FINDCLOSE, FINDFIRSTFILE, FINDNEXTFILE, FREELIBRARY32W, GETCURRENTDIRECTORY, GETDISKFREESPACE, GETFILEATTRIBUTES, GETLASTERROR, GETMODULENAME, GETPRIVATEPROFILESECTION, GETPRIVATEPROFILESECTIONNAMES, 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 淘汰函式 | DIAGOUTPUT, DIAGQUERY, DOSIGNAL, EMSCOPY, GETFREEMEMINFO, GETTASKQUEUEDS, GETTASKQUEUEES, GETWINOLDAPHOOKS, INITTASK1, K327, K329, K403, K404, REGISTERWINOLDAPHOOK, RESERVED1, RESERVED2, RESERVED3, RESERVED4, RESERVED5, SETSIGHANDLER, SETTASKQUEUE, SETTASKSIGNALPROC, WINOLDAPCALL |

從表 **9-2** 可看出，有些不合時宜的函式已經被淘汰了，有趣的函式則加了一堆。某些新函式有公開文件（例如 *GetPrivateProfileStruct*），某些新函式則未公開（例如 *Piglet_361* 和 *GetVDMPointer32W*）。值得注意的是有些新函式只以序號輸出，沒有名稱，例如 K209 和 K210。如果我們能夠知道其函式名稱，肯定比較容易猜出其作用。我發現有些 *Kxxx* 函式（例如 K209）用來爲 Win16 程式在 Win32 heap 中配置或釋放記憶體。使用這些 *Kxxx* 函式的一個好例子是 USER.EXE，它把 WND 結構儲存在 USER DGROUP 的高處（高於 64KB，所以程式員頗感苦惱）。第 5 章對這些函式有更多介紹。

下面是以 DUMPBIN 檢視 Windows 95 的 KERNEL32.DLL 的結果：

```
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)
```

有兩件重要的事情值得注意。第一，某些函式有兩個變種，例如 *AddAtomA* 和 *AddAtomW*。前者是 *AddAtom* 的 ASCII 版本，後者是 unicode 版本。在 Windows 95 和

Win32s 中，大部份 unicode 版的函式都只是把它們的參數從 stack 中取出，然後就回返，因爲這兩個 Win32 平台並不支援 unicode。

第二件值得注意的事情是每一行最後面的號碼。它代表函式在此模組的 RVA（Relative Virtual Address）。這是個偉大的消息：exports section 中內含足夠的資訊，足以將符號名稱和程式碼位址連接起來。稍後你就會看到，擁有符號名稱，在探險行動中不啻是增加了一支大燭光的手電筒。

下面是 DUMPBIN 從 Windows NT 3.5 NTDLL.DLL 獲得的輸出函式列表片斷。

```
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)
    ...
```

有了像 *DbgPrint*、*NtCreateProcess* 和 *NtQuerySystemInformation* 這樣的函式，NTDLL.DLL 就可以埋藏許多有趣的機能在裡頭。在 Windows NT 之中，許多被 DUMPBIN 揭露的未公開函式其實負責的是產生行程、管理記憶體等重頭戲。Windows NT 的 KERNEL32.DLL 只不過是 NTDLL.DLL 的一層薄薄包裝而已。『好啊，但我或許不會用到 NTDLL.DLL』，錯！如果你以 PEDUMP 或 DUMPBIN 檢視某些 NT 程式（如 WPERF.EXE），你會看到它們直接呼叫未公開的 NTDLL.DLL 函式，像是 *NtQuerySystemInformation*。

常常你還可以經由檢驗檔案中的文字而獲得更多的洞察機會。最有用的一段文字就是 description field。聯結器把你在模組定義檔（.DEF）中的 DESCRIPTION 指令所指定的文字放到可執行檔的 description field 中。在 16 位元 NT 檔，這個字串是 nonresident names table 的第一個項目。下面的輸出顯示某些典型的 description 字串（取材自 Windows 95 的 \WINDOWS 目錄）：

```
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
```

對於 32 位元 PE 檔案，聯結器是把 description 字串放到可執行檔的 .rdata section 中。不幸的是沒有什麼一致的模式可以表現其位置。如果你想看該字串，唯一的方法大概是把 .rdata 以 16 進位碼傾印出來，再觀察嵌在其中的 ASCII 文字。由於微軟的 Win32 工具並不需要.DEF 檔，你可能會發現有些可執行檔其實沒有 description 字串。

另一個取得有用字串的地方是 EXE 或 DLL 的 resource section。不論在 Win16 或 Win32 程式設計中，你都可以以名稱或序號指定資源。有時候，對話盒會有一些有趣的

名稱，或是有些隱藏的 controls，位於對話盒四方形之外。Stringtable 常常放了一些你從來沒有想到去看看的東西。例如，在微軟的遊戲軟體 TAIPEI.EXE 中（譯註：好像是個麻將軟體），如果你贏了，程式會以一句箴言獎賞你。如果你想看看所有的箴言，要不你就一直贏，要不你就像我一樣地作弊，把 Stringtable 傾印出來看。

有數種方法可以取得檔案的資源。Borland 的 Resource Workshop 允許你以互動方式觀察並編輯任何檔案中的資源。如果你喜歡命令列方式，Eclectic Software 公司的反組譯器 Win2Asm 附有一個工具程式，可以讀取可執行檔中的二進位資源。它會吐出一個 .RC 檔，你可以直接把它餵給 RC 編譯器 -- 如果你要的話。

檔案傾印工具如果遇到一個內含除錯資訊的檔案，那麼你就中獎了。除錯資訊含有一個程式所有的東西。現代化編譯器的除錯資訊包羅萬象，包括所有變數和函式的名稱、原始碼檔案名稱、結構定義的佈局、類別的階層架構，以及許多其他資料。簡短地說，對於那些知道怎麼看除錯資訊的人而言，除錯資訊和原始碼沒有兩樣。

Borland 的 TDUMP 可以傾印出兩種風味的除錯資訊（16 位元和 32 位元）。以及 Microsoft C7 的除錯資訊。微軟客戶可以使用 CVDUMP 把 CodeView 資訊變成可讀文字。除了 CodeView 資訊之外，微軟的 32 位元編譯器還產生另一種型態的除錯資訊，名爲 COFF -- DUMPBIN 和第８章的 PEDUMP.EXE 可以破解其格式。Nu-Mega 公司的 SoftIce/W 所附的 DBG2MAP 可以根據 Borland 和 Microsoft 32 位元除錯資訊，產生一個可讀的 .MAP 檔。

.SYM 檔是另一種格式的除錯資訊。對於探險行動也有幫助。雖然 .SYM 歷史悠久並且有點笨拙，它還是頗有用處 -- 如果你習慣的話。微軟把某些 .SYM 檔隨著 Windows 95 SDK 一起出貨。但是其中並沒有 system DLL's SYM 檔，而那是許多人想要的。

除錯資訊（除了 .SYM）告訴你的第一個並且也是最明顯的訊息，就是哪一家公司的聯結器成就了這份除錯資訊。你也可以從編譯器的 runtime libraries 版權字樣看到這份訊息 -- 它會被放到程式的資料區中。最重要的是，你可以獲得所有函式和變數的名稱。此外，除錯資訊也內含符號位址。如果你需要反組譯，這些符號名稱會增加許多成功的機會。

除了符號名稱，除錯資訊也含有變數型態、函式參數串列、結構和類別的佈局。簡單地說，除錯資訊內含你不希望你的競爭者知道的一切秘密。我曾經讓一個程式員吃驚得說不出話來，因爲我告訴他他的程式碼在某一行會發生一個 GP fault。我並沒有他的原始碼，只不過是有一個從 BBS 下載的除錯版程式而已。那對我已經足夠了。你的競爭對手可能不會像我這樣仁慈以對，所以千萬不要讓除錯版產品流到市場去。許多公司，包括 Microsoft、Borland、Delrina，都曾經犯下這種錯誤。呵呵，你不妨自己以 TDUMP 或 CVDUMP 檢視 Windows 3.1 的 SOUNDREC.EXE 看看。

即使你沒有讓除錯資訊流出去，傾印檔案並分析其中內容還是可以學到許多東西。1993 年七月份的 ***Microsoft Systems Journal*** 中，我發表了一個名爲 EXESIZE 的工具程式，它可以掃描一個 16 位元 NE 檔，把因爲不良的程式方法而造成的浪費空間找出來。它可以判斷是否檔案的 alignment 應該更小一些，是否有低效率的程式碼產生，是否檔案中留有除錯資訊。有時候 EXESIZE 可以找出 100K 的浪費。我發現那些浪費情況很嚴重的程式是由草率無知的程式員做出來的。如果一個可執行檔通過 EXESIZE 的各項考驗，它一定是出自一個有專業水準的程式員之手，他們不放過任何細節部份。

雖然我把焦點放在 EXEs 和 DLLs 的傾印，但別忽略了其他相關的檔案所能提供的豐富資訊，特別是 OBJ 和 LIB 檔。Borland 的 TDUMP 可以分解 Intel OMF OBJ 檔案格式，告訴你 public 和 external 符號、節區名稱等資料。Symantec C++ 附有一個 OBJ2ASM 工具，可以把 Intel OMF OBJ 檔以符號反組譯出來。微軟的 DUMPBIN 和我的 PEDUMP 都能夠執行一般性的 COFF OBJ 和 LIB 傾印功能。DUMPBIN 甚至可以反組譯 COFF OBJ/LIB 檔。

## 以 Spying 工具探索

雖然檔案傾印很有趣，也富含資訊意義，但常常無法告訴你你需要知道的每一件事情。能夠刺探程式活動程序的工具，才能夠勝任上述要求。最有名的 Windows 刺探工具是訊息刺探軟體：Microsoft 的 SPY 和 Borland 的 WINSIGHT。訊息刺探軟體可以顯示視窗收到的任何訊息，以及對此訊息的反應。

雖然這樣的資訊頗有用處，但程式員通常還需要更多的資訊，以探觸他們希望了解的事物的根源。Nu-Mega 公司的 BoundsChecker for Windows 和 Periscope 公司的 WinScope 把刺探工具提昇到新的層次。除了 Windows 訊息，這些工具還攔截 EXE 或 DLL 發出的 API 呼叫。此外，有些刺探工具監視並記錄 hook callbacks、TOOLHELP notifications、以及其他的 callbacks。這些工具背後的理念是，在控制權進入或離開程式碼的地方（例如視窗函式啦、API 呼叫啦等等）放一根探針。通過這些邊界的資訊都需以一致的格式擺置，例如所有的視窗函式都有一組固定參數，放在堆疊之中（HWND 放在 [BP+0E]，MSG 編號放在 [BP+0C]...等等等），spy 軟體取得這些知識，然後儲存、分析、顯示這些資訊。

最好的 spy 工具不會要求被刺探的對象做任何修改。這些工具只依賴可執行檔中的資訊來安插探針。這將使得所有這些工具得以刺探幾乎任何的 EXEs 或 DLLs，甚至是那些你沒有辦法重新聯結或修改者。

另一種 spy 軟體要求你重新聯結刺探對象（應用程式）。這類工具之所以發揮效用是因爲它愚弄聯結器，把程式發出的 API 呼叫導到工具所提供的碼，而不是作業系統的 DLLs 之中。這些工具通常會改變可執行檔內容 -- 在它們被重新聯結之後。

16 位元 Windows 上有許多 spy 工具軟體。BoundsChecker/W（BCHKW）的目的是找出程式臭蟲，它完成任務的方式是攔截程式發出的所有 API calls 以及某些 C library calls，並檢驗其參數。由於 BCHKW 已經完成了最困難的工作，因此把那些資訊保留到一個追蹤用的記憶緩衝區中不過是舉手之勞。爲了更清楚知道事件的過程以便推導出臭蟲的來源，BCHKW 也監視視窗和對話盒訊息、hook callbacks、TOOLHELP notifications、以及其他有關的 callbacks。

如果你決定把這些追蹤訊息儲存到磁碟中，你可以使用 BCHKW 的 TVIEW 程式，獲得對程式動作的兩種不同方式的觀察：一種是擴展模式，一種是摺疊模式。TVIEW 內含各種事件過濾裝置，可以濾掉某些你不感興趣的訊息或某些一再重複的 API 呼叫動作（這一類 APIs 的典型名單包括 *GetMessage*/*TranslateMessage*/*DispatchMessage*/Window

Message/*DefWindowProc*）。

雖然 BCHKW 取得除錯資訊，協助其「臭蟲探照燈」的角色扮演，但那並非必要。你可以面對任何 Windows 程式使用 BCHKW，不只是你自己開發中的程式。

另一個 16 位元程式的刺探工具是 Periscope 公司的 WinScope。和 BoundsChecker/W 一次關心一個程式不同，WinScope 是一個系統層級的刺探工具。WinScope 告訴你所有的 API calls、hooks、以及發生在系統任何一處的訊息。有時候這非常有用，有時候嘛，呵呵，資訊爆炸！

幸運的是 WinScope 提供了非常高階的訂製能力。你可以癱瘓任何一個（或一組）API 的監視能力，你也可以癱瘓任何一個訊息或 hooks 的監視能力。和 BoundsChecker/W 一樣，WinScope 也可以把一個 API 的遠程指標參數所指的一塊記憶體內容拷貝出來。這使你得以看到交給 *CreateWindow*、*GetPrivateProfileString* 等函式的資料結構和字串內容。WinScope 也可以儲存每一個 event 的發生時間，使 WinScope 因此成爲一個粗淺的效率評論器。WinScope 使用 NE 檔中的資訊來對 API calls 做 "hook" 動作，所以你不需要對刺探對象重新聯結。

如果你有意犧牲一些功能，節省一些金錢，可以考慮微軟的 API parameter profiler。16 位元版和 32 位元版都附在 Windows NT SDK 之中，但很少人知道它們的存在。

Microsoft profiler 的功能嫌粗淺了些，而且它要求你修改檢視的對象。這個工具的核心是一組 DLLs（ZERNEL.DLL、ZSER.DLL、ZERNEL32.DLL、ZSER32.DLL 等等）。每一個 DLL 對應於一個基本名稱相同的 system DLL，只有第一個字元改爲 'Z'。針對它們所取代的 DLL 的每一個輸出函式（export function），這些 'Z' 開頭的 DLLs 都準備了一個小小程式（所謂的 stub）。例如 USER.EXE 有 *CreateWindow*，所以 ZSER.DLL 也有一個 *CreateWindow*。把你的 EXEs 或 DLLs 以 APFCNVRT（或 APF32CVT）程式和這些 'Z' DLLs 連接起來。APFCNVRT 和 APF32CVT 會修改你的程式碼，使後者從這些 'Z' DLLs 中輸入（import）函式，而不是從 system DLLs 中輸入函式。於是，執行時候，應用程式的所有呼叫動作都會先跑到這些 'Z' DLLs 來。這個 parameter profiler 可以

把它所收集到的資訊儲存到磁碟中以便觀察。至於 32 位元程式，微軟提供了另一組 DLLs，具有真正的評論能力，而不只是記錄 API 的呼叫狀況。

除了 32 位元版的 Microsoft parameter profiler，Nu-Mega 的 BoundsChecker32（BCHK32）程式（有 NT 版、95 版、Win32s 版）也刺探 Win32 程式的 API calls 和視窗訊息。關於「刺探」，BCHK32 的表現類似 BoundsChecker/W。然而，它還提供了一些新功能。第一，當函式呼叫失敗，API 通常會儲存 *SetLastError* 所傳回的錯誤碼，指示失敗原因。BCHK32 知道一個 API 何時失敗，並記錄其錯誤代碼。第二，由於 Win32 支援執行緒，BCHK32 針對每一個 API call 和視窗訊息，儲存其執行緒 ID。TVIEW 使用這個執行緒資訊提供額外的過濾選項，像是「只顯示某一執行緒之 events」等等。

我要說的最後一個 spy 工具是本書第 10 章提供的 APISPY32。雖然它並不像 BoundsChecker 那般功能完整，不過的確提供了 spy 工具的根本元素（包括顯示函式參數和傳回值）。它很容易擴充監視任何你想要的 Win32 DLL，而且不需修改被監視對象。

這些 spy 工具的評論關鍵在於，你被允許觀看系統的哪一部份。WinScope 允許你刺探好多 system DLLs（USER、KERNEL、GDI 等等等）。更重要的是，WinScope 有能力刺探經由你描述過的其他 DLLs。BoundsChecker/W 刺探 10 個 system DLLs（差不多和 WinScope 預設情況下相同）。Microsoft parameter profile 只能夠觀察三個標準的 system DLLs（USER、KERNEL、GDI）。BoundsChecker32 目前能夠刺探 KERNEL32、USER32、GDI32 和 ADVAPI32，以及另外幾個重要的 DLLs。

工具的描述我想夠了。讓我們真正解決一個問題，你才能夠更清楚這些工具的效能。我在 Internet 的程式設計論壇中看到多次有關於 CLOCK.EXE 的討論。最頻繁的問題就是：『我如何能夠讓我的程式在有標題和無標題之間切換，像 CLOCK.EXE 那樣？』你可以利用 spy 工具，找出 CLOCK.EXE 在切換標題時呼叫了哪一個 API 函式而獲得答案。這裡，我面對的是 Windows NT 的 32 位元 CLOCK.EXE，但 16 位元 CLOCK.EXE 也是一樣。Windows 95 並沒有提供 CLOCK.EXE，但可以拿 NT 的那一個來執行，沒問題。

至於工具，任何我說過的工具都可以，不過我的選擇是 BoundsChecker32/NT。如果你使用的是其他工具，不妨照著我的步驟做。

第一個步驟是把刺探對象執行起來，並且收集其追蹤資訊。爲了這麼做，先執行 BoundsChecker，從【File/Load】對話盒中選擇 CLOCK.EXE，再按下【Run】。CLOCK 啓動之後，選按其【Settings/No Title】選單項目（我假設一開始的 CLOCK 有一個標題和一個選單）。現在把 CLOCK 結束掉。

下一個步驟是檢視追蹤資訊，並找出程式中反應【Settings/No Title】動作的那一點。下面是輸出片斷：

```
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)
```

你可能會懷疑：『我怎麼知道到哪裡找出我所需要的資訊？』答案非常簡單。只要你在選單上做了一些選擇，Windows 就會送出 WM_COMMAND 到你的程式中。因此你第一個要找的就是 WM_COMMAND 字串。如果你按照上述步驟去做，那麼在整個運轉記錄中就只會出現一個 WM_COMMAND 字串。然而，保險起見，我們還是得驗證這個 WM_COMMAND 的正確性。

在一個 WM_COMMAND 訊息中，WPARAM 放的是選單項目的 ID。在這份輸出列表中，WPARAM 是 6。如果你以 Resource WorkShop 或其他類似工具檢查 CLOCK.EXE 的資源內容，你會看到【No Title】選單項目的 ID 的確是 6。現在我們能夠確定我們找到了正確的目標。

在 CLOCK.EXE 收到 WM_COMMAND 訊息，被告知要收起其標題後，CLOCK 所做的第一個反應是呼叫 *GetWindowLong*，以 GWL_STYLE 爲參數。下一行輸出顯示，*GetWindowLong* 傳回 DWORD 值 0x14CF0000。這個值表現出當初 *CreateWindow* 收到的 WS_xxx，代表視窗風格。你可以在看過 WINDOWS.H 之後解開這些位元意義：

```
#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
```

現在，暫時忽略下兩行輸出（我會馬上回來）。在 CLOCK 利用 *GetWindowLong* 獲得其風格位元之後，它呼叫 *SetWindowLong*，做了一個輕微的改變。這一次視窗風格是 0x14840000。看來 CLOCK.EXE 是把其 WS_xxx 位元取回去，修改之後再設回。好，它到底改變了什麼風格呢？比較原來的 0x14CF0000 和新的 0x14840000，新值少了下面幾個風格：

```
#define WS_DLGFRAME         0x00400000L
#define WS_SYSMENU          0x00080000L
#define WS_MINIMIZEBOX      0x00020000L
#define WS_MAXIMIZEBOX      0x00010000L
```

這符合 CLOCK 的行爲。當你選擇【No Title】選項，系統選單以及極大極小盒就不見了。

現在我要回到剛剛被我忽略的那兩行輸出。第一行呼叫 *SetWindowLong*，設定視窗的 control ID（GWL_ID）爲 0。你一定會對此感到奇怪。讓我告訴你這個秘密：所有視窗都有一個欄位，既可以是 menu handle 也可以是 control ID。top-level 視窗（如 CLOCK.EXE）放的是一個 HMENU，child 視窗（例如對話盒中的 controls）放的則是 control ID。官方手冊上把這個欄位稱爲 hMenu 欄位。

知道這鮮少被人注意的細節後，我們看看 CLOCK 爲什麼要把其 HMENU 設爲 0。或

許最好也最清楚的方式是用 *SetMenu* 改變其 HMENU。然而，可能有一些檯面下的理由，使 CLOCK 作者不使用 *SetMenu*。一個可能的理由是，*SetMenu* 會強迫選單區被重繪，因而影響了選單中的改變。

輸出片斷中的最後一行顯示，CLOCK 呼叫 *SetWindowPos*。這是一個功用廣泛的函式，可以移動視窗、改變其 Z-order（譯註：視窗前後覆疊關係）、引發 Windows 重新計算並重繪視窗。最後這一點（重新計算並重繪視窗）可能是導至 CLOCK 不使用 *SetMenu* 的原因：在 CLOCK 改變風格位元以及 HMENU 之後，它必須負責以新風格重繪視窗。呼叫 *SetMenu* 會引起部份視窗被重繪，後續再呼叫 *SetWindowPos* 將引起視窗再被重繪。這會造成螢幕閃爍。CLOCK 的作者可能意識到，直接以 *SetWindowLong* 對 HMENU 做猛烈的一擊，可以解決螢幕閃爍的問題。他們知道稍後的 *SetWindowPos* 會重繪視窗。

CLOCK 傳給 *SetWindowPos* 的參數頗爲有趣。僅有的非零參數是 HWND 和各個 SWP_xxx 旗標值。前三個旗標告訴 Windows 說 CLOCK 不希望改變視窗的大小、位置、Z-order。最後一個參數告訴 Windows 說 CLOCK 的視窗外框已經改變。這會強迫 Windows 重新計算客戶區（client area）和非客戶區，並且重繪整個視窗。CLOCK 的原始碼在 Win32 SDK 的 SAMPLES\DDEML\CLOCK 子目錄中有提供，你可以自行驗證。

以 CLOCK 爲例，我們看到了 spy 工具能夠顯示「視覺效果的變化（標題欄的消失）是如何辦到的」。spy 工具也能夠對於揭露底層秘密做出貢獻。程式員常有一個癖好，就是爲他們的程式加上一些隱藏的行爲或機能。例如他們可能希望程式能夠把除錯診斷訊息寫到檔案去。由於這個性質只在罕見情況下才使用，程式員不要在使用者介面上增加新的選項以迷惑終端使用者。再者，加上新的功能選項就得描述它，說明它。爲了不想把寶貴的時間花在罕見的情況，結果就帶來了所謂的「未公開性質」。

一個用來尋找未公開性質的技術是，搜尋程式的 .INI 檔中的記錄項，看看有沒有一般情況下不會出現的東西。程式使用者必須知道確實有此項目，並以手動方式把該項目加到 .INI 檔。雖然這些討論專注在 .INI 檔，相同的情況也出現在 Win32 registry。

如果有個 spy 工具能夠儲存 API 指標參數所指向之物體的一份副本，對於我所描述的那些事態的發現就會很大的幫助。雖然我可以使用 BoundsChecker/W 或 WinScope，但我選擇微軟的 Parameter profiler 做一個示範。下面是執行 Windows 3.1 WINMINE.EXE（譯註：中文名爲「踩地雷」的一個遊戲軟體）後的一份結果：

```
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...
```

每一個 "APICALL:" 的第一部份是函式呼叫的巢狀層級。上例的所有巢狀層級都是 01，也就是最上層級。這意味著 WINMINE 不會在函式進行至一半時呼叫另一函式。"APICALL:" 之後的是函式名稱，再來是其參數。微軟的 Parameter profiler 可以顯示字串參數的內容，而不是只有一個位址（像是 0x10B7:003A 之類）。*GetPrivateProfileInt* 有三個 LPSTR 參數，所以此性質對它特別有用。

注意上述的 WINMINE 事件追蹤記錄，程式碼尋找每一個 INI 項目達四次之多。我不知道爲什麼如此。嘗試解開像這樣的碼，是探險行動最有趣的地方。

把這個問題置之一旁。請看每一個 APICALL 的第二個參數。那是 INI 某一區內的某個項目名稱。第一組 APICALL 尋找一個名爲 Ypos 的記錄項。如果你去看 WINMINE.INI 檔，的確有一個 Ypos 記錄項。然而後三個記錄項（Sound、Tick、Menu）就都沒有在 .INI 檔的任何地方出現。更進一步看看 WINMINE 結束之前所寫的一個新的 INI 檔案內容，你也看不到 Sound、Tick、Menu 三個記錄項。

我們發現的是三個影響 WINMINE 行爲的未公開方法。我曾經把這三筆項目加到 WINMINE.INI 中，Tick 沒有帶來任何影響，但 "Menu=1" 卻使 WINMINE 不顯示主選單，"Sound=3"（或更高值）使得 WINMINE 在你遊戲輸贏時播放一些聲音。

## 反組譯（Disassembly）工具探索

雖然反組譯很複雜也很困難，它卻常常是解決神秘演算法或技術的唯一辦法。

並不是只有面對其他人的碼，才要考慮反組譯。當你遭遇一隻頑強的臭蟲，沒辦法從原始碼中一眼看出其出處，這時候如果有能力建立並觀察高階語言與低階的編譯器產生碼之間的關係，將是一種非常有價值的技巧。反組譯自己的碼也可以讓你瞭解編譯器是否爲你做了適當的最佳化。如果你的程式執行時獲得出人意料的 GP fault，也是反組譯的好時機。你的客戶把令程式翹辮子的 GP fault 的位址告訴你，而你可以反組譯該處的碼，看看程式做了什麼好事。

在繼續下去之前，我必須強調，反組譯並不是給那些對細節沒有興趣的人準備的。如果你對組合語言碼無動於衷，反組譯工具並不適合你。你要不已經熟悉組合語言，要不就得開始學習它。這倒不是說你必須以組合語言寫程式。使用高階語言是很好的。我是說你得在反組譯器所產生的非常低階的機器碼和暫存器中打滾。

你所選擇的反組譯工具有時候會因探索對象（檔案格式）的不同而不同。反組譯器必須知道它所對付的可執行檔格式。最簡單的反組譯器就是一個檔案傾印工具連線到一個反組譯引擎，以傾印資料爲輸入，輸出組合語言助憶碼。最好的例子就是 Visual C++ 32 位元聯結器中的 /DUMPBIN /DISASM 選項。更高級的反組譯器可以讀入符號資料（它將程式中的符號名稱和一個位址關聯起來）。這些反組譯器可以產生組合語言列表，並使用真正的變數和函式名稱，而不只是一堆十六進位碼。

最爲人熟知的 PC 反組譯器當推 V Communication 公司的 Sourcer。Sourcer 可以處理 DOS EXE 和 COM 檔案。如果加上額外的一些描述檔，Sourcer 還可以處理 NE 檔、VxDs（LE 檔）、以及 Win32 PE 檔。Eclectic Software 公司有一個 Win2Asm 反組譯產品，可對付 NE、LE、PE 檔。RJ Swantek 公司有 DisDoc 反組譯產品，它的火力範圍和 Win2Asm 相同。如果你只在乎 Win32 檔案而且價格是個考慮因素的話，微軟 Win32 SDK 所附的 DUMPBIN 是個選擇。稍後我將以 DUMPBIN 做一次反組譯示範。

想知道我寫作這本書時所依賴的反組譯器嗎？我有一對自己發明的反組譯器，一個用於 Win16 NE 檔，一個用於 Win32 PE 檔和 VxDs。雖然它們並不像 Sourcer 那樣有多層檢驗，對我卻已足夠。自己寫一個反組譯器是因爲如此一來我可以視特殊情況而修改工具。

如果你只是想要以我即將描述的反組譯技術對程式做一些粗淺的修補工作，那麼除錯器上的反組譯工具或許就夠了。當然這是假設你用的不是那種連一個組合語言視窗都不提供的整合環境除錯器。某些除錯器可以把它們的傾印內容寫入檔案中。將數個相關的組合語言視窗內容都印出來，你或許可以獲得一些極有價值的資料。然而這是十分沉悶而令人生厭的工作，而且耗時頗多，特別是如果你要觀察的常式呼叫了另一個程式的函式的話。如果你很嚴肅面對反組譯，你應該擁有一個真正的反組譯器，像 Sourcer、Win2Asm、DisDoc 等等。它們並不昂貴 -- 特別是想到它們帶來的強大威力的時候。

## 反組譯的環境與藝術

沒有什麼單一準則適用於所有的反組譯動作。這裡我所描述的，是對我有效的準則。如果其他準則適用於你，別猶豫。我的反組譯基本哲學，一言以蔽之，就是「分散征服，

各個擊破」。拿到一份反組譯碼之後，我並不處理整個函式或整段碼，而是經由一系列程序把範圍打破、縮小，再進攻，才有較大的成功率。我的終極目標就是把反組譯碼轉換爲日後方便易讀的 C 形式碼。

依照你所處理的碼的不同，以下數點的重要性以及次序可能會有變化。首先我會說明反組譯過程中的一般術語。然後我就跳到核心問題去：參數鑑定啦、區域變數啦、條件判斷啦、函式呼叫啦...等等。最後我會示範一個實作例子，利用一些反組譯輸出結果，轉化爲有用的資訊。反組譯一個函式時，你所需要的步驟列於下面。

### 步驟１：將檔案反組譯

對著你的可執行檔，執行反組譯器，獲得一份列表檔案。如果你的反組譯器可以處理額外的符號輸入（例如來自 .SYM 或除錯資訊），一併餵給它。

如果你只對某個函式感興趣，可以把反組譯列表的其他部份殺掉，使你的列表檔案容易管理些。我曾經獲得某些反組譯器的列表檔案超過 3MB，這讓我的編輯器疲於奔命。把不在乎的東西砍掉可以加速程序的進行。

### 步驟２：為已知物體做上標記

爲已知意義的物體標上更有敘述性的名稱。所謂已知物體，我意思是函式參數啦、區域變數啦、全域變數啦等等。這麼做可以先把最簡單工作解決掉。能處理的先處理，能丟棄的先丟棄。這樣可以降低未知數的個數，也可以讓你對剩餘的部份有較好的認識。

假設你知道函式所使用的語言和呼叫習慣（譯註：例如 C 或 Pascal 呼叫習慣），你可以輕易辨識出堆疊中的參數，並爲它們冠上有意義的名稱（例如 hWnd）。稍後我將討論堆疊參數。

除非可執行檔中有除錯資訊，否則決定區域變數的名稱比決定參數名稱困難。如果在這個階段中你尚未理解每一個名稱，不必擔心。但你應該想盡一切辦法爲它取一個有意義

的符號名稱。

如果你提供了全域變數的符號資訊，或許反組譯器就把全域變數的位址以符號名稱取代了。然而，如果它沒這麼做，你得自己動手。

### 步驟3：將指令序列（instruction sequences）區隔開來

反組譯列表常常出現一長串指令，而且中間沒有什麼空白行。我發現插入一些空白行，將指令劃分爲邏輯群組，對於理解是很有幫助的。雖然聽起來有點模糊，事實上不難做到。函式前置碼（prologue）就是一個好例子。另一個可以劃分出來的指令集就是「把參數推入堆疊然後呼叫函式」的動作。第三個例子就是「執行某些運算後把結果儲存到變數」的動作。一個有助的（但不是嚴格規定的）指導方針就是，嘗試把一系列組合語言指令組成一個原始碼指令。放一空白行將一群指令區隔出來，也是個好主意。

反組譯過程中你需要面對條件敘述，也就是高階語言中的 if、while、do、switch 等等。我發現放一空白行在每一個有條件的或是無條件的 jump 指令之後，可以讓我們比較容易瞭解。如果你的反組譯器沒有自動這麼做，那麼就動手完成它吧。我通常以文字編輯器的巨集輕鬆而快速地完成這項任務：搜尋 'J' 開頭的指令，然後在其下方插入一空白行。最近，我已經修改了我自己的反組譯器，自動完成這項工作（這也是爲什麼我喜歡自己寫反組譯器的原因之一）。

### 步驟4：加上字串（string literals）

如果函式看起來使用了一些字串，請爲它加上註解。把註解放在使用這個字串的函式呼叫動作附近。這可以幫助我們把好幾行組合語言碼轉換爲 C 指令。

### 步驟5：將指令凝結爲單一的 C 語言敘述

把函式呼叫和中斷（interrupts）凝結爲單一的 C 語言敘述。找出呼叫某個已知函式（包括函式名稱和參數）的地方，研究堆疊中的參數並重建 C 語言參數。

### 步驟 6：鑑定條件句

鑑定條件句並轉換爲高階語法。如果你看到 TEST 或 CMP 指令之後緊跟著 JMP 指令（例如 JE），那大概就是個高階的 if 動作。Jxx 指令跳過去的地方通常是一個複合敘述的尾端。所謂複合敘述，在 C 語言就是 { } 之間的每樣東西，在 Pascal 語言則是指 BEGIN/END 之間的每樣東西。

如果你看到一長串的 test 和條件式 jumps，你大概遇到了 C 語言的 switch 或 Pascal 語言的 case 句型。瞭解這些條件句是個棘手的工作。高階語言的多層 if 敘述可能產生非常複雜的組合語言碼，例如：

```
if ( (GetModuleHandle("MYDLL.DLL") != 0)
     && ( (hWnd != GetDesktopWindow()) || ( styleFlags & WS_POPUP)) )
```

會產生非常令人討厭的巢狀條件式 jumps，暫得結果則儲存在暫存器中。凝視 20 行（或更多）的組合語言碼而腦袋空空沒有任何線索，並不是罕見的事。這也就是爲什麼我強調反組譯器是最後手段的原因。

### 步驟 7：重複（如果必要的話）

如果必要的話，重複前述步驟。聽起來可能有點陳腔濫調，但並不是一定非這麼做不行。這是個一再一再重複的程序。你可以先走過一遍，把你可以做的做掉，然後回頭，看看整個有什麼改變，然後再走另一遍。理解了某一段令人困惑的碼之後，或許其他段落就跟著土崩瓦解了。

## 辨識常見的程式碼和習慣動作

討論過如何反組譯一個函式後，現在我要檢驗某些常見的碼，以及組合語言的一些習慣。這可以幫助你把組合語言碼翻譯爲對等的高階語言碼。

## 辨識函式和程序

看反組譯輸出碼，第一件事情就是辨識出哪裡是一個函式的開始和結束。最容易找到函式起始位置的方法就是尋找編譯器產生的標準 prologue 碼。對於 16 位元碼，標準 prologue 是函式的一些變化：

- 將原來的 BP 暫存器儲存到堆疊。
- 將堆疊指標指定給 BP。
- 爲函式區域變數留下堆疊空間。
- 將呼叫端的暫存器變數儲存到堆疊中。

以組合語言解釋之，prologue 碼如下：

```
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，如果 80286 或更好的程式碼產生選項發生效能的話，prologue 碼如下：

```
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
```

這些堆疊組織（stack frames）是編譯器爲 16 位元保護模式程式所產生的碼。在真實模式那個糟糕老舊的日子裡，當可移動節區遍佈於記憶體各處，Windows 本身常常需要走訪一個程式的堆疊。由於對於一個既有近程呼叫又有遠程呼叫的程式，走訪其堆疊十分棘手，編譯器經由單數的 BP 堆疊組織帶來幫助。當「產生單數 BP-frame」發揮作用，所有遠程函式先將 BP 暫存器加 1，然後才把它推入堆疊（近程函式則不這麼做）。而在（遠程函式的 epilogue 碼中）恢復原來的 BP 後，程式碼將 BP 減 1。於是，在走訪堆疊組織時，如果 Windows 看到被儲存的 BP 是單數，它就知道函式是個遠程函式。單數 BP 風格的遠程函式看起來像這樣：

```
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
```

如果在 32 位元程式，標準的 prologue 碼看起來像這樣：

```
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

```
ENTER   XX,0    ;; XX is the number of bytes needed for locals.
PUSH    ESI
PUSH    EDI
PUSH    EBX
```

上述是完整功能的 prologues。在真實世界中， prologue 可能會有部份遺漏或不同：

- 如果函式碼不改變當做變數使用的暫存器（例如 ESI、EDI、EBX）內容，它就不需在 prologue 中把它們的值先儲存起來。注意，在 32 位元碼中，EBX 有時候也用來當做暫存器變數，雖然在 16 位元碼中很少這麼做。
- 在 16 位元碼中，如果函式沒有任何參數或使用任何區域變數，編譯器可能會省略 PUSH BP / MOV BP,SP 這兩個動作。
- 在 16 位元碼中，甚至即使函式使用了參數或區域變數，編譯器還是可能不設定 EBP-frame。386 CPU（或更高）之 32 位元定址模式允許編譯器以 ESP 暫存器來定址參數和區域變數，例如：

```
MOV EAX, [ESP+1C]
```

辨識函式的 epilogue 碼比較棘手些。如果編譯器做了最佳化設定，函式內可能會出現許多個 RET 或 RETF。假設函式在其尾端有單一一個完整的 16 位元 epilogue，看起來將是這個模樣：

```
POP     DI  ;; Restore caller's register variables
POP     SI
LEAVE       ;; or ADD SP,XX / POP BP
RETF        ;; far return. Near return is a RET.
```

32 位元的 epilogue 則是這幅模樣：

```
POP     EBX ;; Restore caller's register variables.
POP     EDI
POP     ESI
LEAVE
RET         ;; A 32-bit near return.
```

決定一個函式的起始和結束之後，記住，緊跟在後面的，就是另一個函式的開始。如果你看到某些碼像是 epilogue，請尋找另一個看起來像 prologue 的段落，與它湊成對。如果你沒找到，那麼一種可能是編譯器的最佳化動作使 prologue 消失了，另一種可能是前一個函式有多重出口。

## 函式回返值

當函式傳回一個數值，它是把數值放在一個暫存器（或一組暫存器）中傳回。爲了確定函式傳回值有沒有被使用，請你檢查函式呼叫者，看它們有沒有使用那些暫存器。如果你看到某處呼叫一個函式，然後使用那些專門用來傳遞回返值的暫存器，而卻沒有事先設定其值，你就知道，它正在使用函式回返值了。舉個例子，如果你看到程式碼中呼叫一個函式，然後使用 AX 而未先設定其值，你可以確定，被呼叫函式利用 AX 傳回其回返值。

在 32 位元碼中，函式習慣以 EAX 做爲傳回值的媒介。16 位元碼則使用 AX 傳遞 16 位元數值，以 DX:AX 傳遞 32 位元數值。如果程式是以組合語言完成，程式員可以使用任何他們喜歡的暫存器完成傳遞工作。組譯器通常遵循的一個習慣是，如果函式只是傳回一個真假值（代表成功或失敗），它通常會設定或清除 carry flag（CF）。搜尋緊跟在 CALL 指令之後的 JC 和 JNC 指令，便很容易嗅出這類函式。

## 參數的辨識

如果你已經知道你正在拆卸的函式所需要的參數，在組合語言碼中爲它們貼上標籤將十分容易。除了一個例外（稍後我會帶到），編譯器總是利用堆疊來傳遞函式參數。只要知道每一個參數的大小，你就可以輕易定位出堆疊中的每一個參數。然而在我展示範例之前，我首先必須解釋 Windows 和 Win32 所使用的「函式呼叫習慣（calling convention）」。

在 16 位元 Windows 碼中，大部份的被輸出函式（exported function）都使用 pascal 呼叫習慣 -- 呼叫端負責把參數推入堆疊，推入的次序是最左邊的參數至最右邊的參數。舉個例子，16 位元碼的 "foo(0x10, 0x20, 0x30)" 經過編譯後產生這樣的結果：

```
PUSH 0010h
PUSH 0020h
PUSH 0030h
CALL FAR PTR FOO
```

除了參數推入堆疊的次序從左到右，Pascal 呼叫習慣也要求「被呼叫端」必須在回返之前負責清除堆疊。以我所引用的這個 foo 函式而言，它必須在回返之前將堆疊的 6 個位元組推出堆疊。或許它會使用一個 "RETF 6" 指令。

Pascal 呼叫習慣的相反是 C 呼叫習慣。標準的 C/C++ runtime library 使用 C 呼叫習慣 -- 參數由最右至最左推入堆疊（這樣做的好處是它可以處理像 *printf* 那種參數不定的函式）。至於清除堆疊的工作則由「呼叫端」負責。16 位元碼的 "foo(0x10, 0x20, 0x30)" 如果使用 C 呼叫習慣，會得到這樣的結果：

```
PUSH 0030h      ;; Parameters pushed right to left.
PUSH 0020h
PUSH 0010h
CALL FAR PTR FOO
ADD  SP,06h     ;; Remove parameters from the stack.
```

你不應該以爲在 C 呼叫習慣中總是能夠看到 "ADD (E)SP,XX" 這樣的動作。如果編譯器只推入（push）一個或兩個參數到堆疊裡頭去，有時候它會把它們推出（pop）到一個暫存器中，以清除堆疊。我知道 Borland C++ 編譯器就會這麼做。

至於 Win32，微軟接受了 stdcall 呼叫習慣，應用於幾乎所有的 system DLLs 輸出函式（exported function）。Stdcall 是 C 和 Pascal 的混合，它的參數傳遞次序是像 C 那樣由右至左，清除堆疊的責任則落在「被呼叫端」身上，這一點又像 Pascal。此外，當你在 Microsoft C++ 中使用 stdcall 呼叫習慣，編譯器會自動（內部地）在函式名稱之後加上 "@xx" 字串，xx 用以表示參數的總位元組個數。例如 *_GetWindowLong@8* 或 *_PeekMessage@20*。

瞭解函式呼叫習慣之後，你就可以決定參數在堆疊中的位置了。知道參數在堆疊的偏移位置後，你就可以尋找是否有哪個指令參考到該記憶體位址，然後把該組合語言位址以符號名稱取代之。符號名稱對於反組譯有莫大幫助。

在函式執行其 prologue 之後，堆疊看起來像這樣：

```
Parameters
return address （這是由 CALL 指令完成的）
previous (E)BP （這是由 prologue 碼完成的）
```

如你所見，(E)BP 現在指向 (E)BP 原值的儲存場所。在此函式之中，所有參數都能夠以一個正數位移（以 BP 或 EBP 爲基礎）參考到。這是值得再敘述一次的重點：若有某個指令存取某塊記憶體，而使用的記憶體位址是 [BP+xx] 或 [EBP+xx]，大概就是用到了函式參數。

對於一個 16 位元遠程函式，例如 Win16 API 函式，堆疊看起來是這個模樣：

```
Parameters  （starting at BP+06）
return CS   （at BP+04）
return IP   （at BP+02）
previous BP （at BP+00）
```

假設每一個參數都是 WORD，使用 Pascal 呼叫習慣，函式的最後一個參數應該放在 [BP+06] 處，倒數第二個參數應該放在 [BP+08] 處。如果參數中有 DWORD，位置就應該有所調整。如果這是近程函式，參數在堆疊中的位置也應該調整，因爲這時候只有 IP 需要放在堆疊中，CS 不需要。

讓我們看一個真實例子，以便對我所說的有更好感受。16 位元視窗函式有下列型式：

```
LRESULT WINAPI WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
```

在 WndProc 函式碼中，堆疊狀態如下：

```
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]
```

有了前面所說的知識，你可以利用文字編輯器中的「搜尋並取代」功能，找出所有的 [BP+0E]，代之以有意義得多的符號 [hWnd]。同樣地你也可以把 [BP+0C] 代之以 [msg]，依此類推。

現在，讓我們看看上述例子的 32 位元情況。在 Win32 中，所有參數都是 32 位元，傳回值是一個 32 位元近程指標，而且程式碼中使用的是 EBP 而非 BP。同時也別忘了，WndProc 使用 stdcall 呼叫習慣，所以推入堆疊的次序與 16 位元情況恰恰相反。32 位元視窗函式的堆疊因此像這個模樣：

```
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]
```

上面就是 32 位元函式的正常堆疊狀態。現在我要告訴你一個壞消息。32 位元編譯器有個選項，允許不產生出標準的 EBP frame。這麼做可以節省時間和空間。

問題就在於，這樣做出來的碼不以 EBP 做爲參數和區域變數的定位基準，而可能以 ESP 爲基準，像 [ESP+14] 這樣。這使你恐慌嗎？應該是。要知道函式中的 ESP 隨時會改變 -- 只要有東西被推入堆疊。因此，如果 [lParam] 原本位於 [ESP+14]，稍後在你推入一個 DWORD 到堆疊之後卻變成了 [ESP+18]。如果再推入一個 DWORD，那就變成 [ESP+1C] 了。因此幾乎不可能把 MOV EAX, [ESP+14] 這樣的組合語言碼轉換爲 MOV

EAX, [lParam] 這樣的形式。因此你必須在整個函式中憑藉著個人的腦力去追蹤 ESP 的相對位置，給予參數和區域變數的符號名稱。噁！你唯一真正的希望就是編譯器把參數拷貝到暫存器中並在任何它需要其值的時候，使用該暫存器。

如果你磨刀霍霍的對象是一個你並不知道參數的函式，好，還是有一些小事情可以做，用以幫助你的反組譯工程。你可以確定這個函式的參數總共有多少個位元組。你可以先觀察函式的 epilogue 碼，它是否使用類似 RETF 8 這樣的指令？如果是，表示參數共 8 個位元組。如果 epilogue 碼並沒有從堆疊中推出任何東西，你可以尋找程式碼中對此函式的呼叫處。在 CALL 指令的下面是不是有像 ADD ESP,12 這樣的動作？如果是，這函式需要 12 個位元組的參數。

除了參數的總位元組個數之外，你還可以研究「函式呼叫之前，參數推進堆疊的動作」，以蒐集更多資訊。假設你看到了下面的 Win32 程式反組譯碼片斷：

```
CALL    GetFocus
PUSH    EAX
CALL    GetCurrentThread
PUSH    EAX
CALL    DoSomething
```

從上述片斷中你可以知道 *DoSomething* 有兩個參數，一是 HWND，一是 thread HANDLE。怎麼知道呢？*GetFocus* 和 *GetCurrentThread* 都是 Win32 API，傳回一個值，放在 EAX 中。呼叫 *GetFocus* 之後，EAX 放的是個 HWND。呼叫 *GetCurrentThread* 之後，EAX 放的是個 thread HANDLE。經過邏輯推演，*DoSomething* 應該接納兩個參數，一是 HWND，一是 HANDLE。

雖然參數通常是經由堆疊傳遞，但也不是沒有可能經由暫存器。這樣的呼叫習慣稱爲 fastcall convention，因爲經由暫存器傳遞比經由堆疊快。許多 KRNL386 的內部 heap 函式都採用這種方法以加快速度。編譯器或組合語言程式員視函式的個別差異而決定要不要用暫存器來傳遞參數。微軟編譯器以 @ 字元做爲 fastcall 函式的起頭，你的 Foo 函式在 .MAP 檔中將變成 @Foo。Fastcall 函式並不一定要讓所有的參數都藉由暫存器傳遞，編譯器可以決定讓部份參數仍然透過堆疊傳遞。

如果你遇到中斷（interrupt），請準備好你的中斷表以及參考文件，找尋哪一個參數放在哪一個暫存器中，然後爲 INT 指令加上一些註解。你一定會準備 Ralf Brown 的中斷列表，對不？（譯註：Ralf Brown 和 Jim Kyle 合著有 ***PC Interrupts*** 一書，是很詳盡的參考資料）。下面是個例子：

```
MOV AX,0500
LES DI,[myBuffer]
INT 31
```

將會變成：

```
MOV AX,0500
LES DI,[myBuffer]   ; DPMI function 0500h - Get Free Memory Information
INT 31            ; ES:DI -> structure to fill with information
```

## 辨識區域變數

和函式參數一樣，區域變數通常也棲身於堆疊中。函式參數和區域變數之間最關鍵性的差別在於，區域變數在堆疊係以負偏移值來定位。例如 16 位元碼中的[BP-4] 或 32 位元碼中的 [EBP-4]。

和函式參數不同的是，區域變數沒有什麼制式作法可以決定變數的型態、使用、位置。你必須斟酌函式碼如何使用一個特殊的位址。有時候決定一個區域變數的意義很容易，例如下面這段 Win32 碼：

```
PUSH DWORD PTR [EBP+08]
CALL GetParent
MOV  [EBP-0C],EAX
```

*GetParent* 是一個 Win32 API，需要一個 HWND 參數，以 EAX 傳回父視窗的 HWND。由於程式片斷中把 EAX 拷貝到 [EBP-0C]，很顯然 [EBP-0C] 是一個 HWND。由此你可以做一次狂野的猜測：參數或許名爲 "hWndParent"。到了這一步，你可以利用你的文字編輯器的「搜尋並取代」功能，把所有的 [EBP-0C] 改變爲[hWndParent]。完成之後再看看你的反組譯碼，是不是清爽多了？

你們之中可能有人要問：『好得很，Matt 先生，但不是每一個區域變數都這麼容易被摘取出來唷！』是的，我們還有其他方法。有時候我們很容易因爲某個區域變數用做其他函式的參數而把它們辨識出來。下面就是個 Win32 實例：

```
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.
```

看看 SDK 手冊中的 *GetWindowRect* 資料，我們知道它需要一個 HWND 參數，一個指向 RECT 結構的指標參數。由於 *GetWindowRect* 是一個 stdcall 函式，所以 RECT 指標應該先被推入堆疊，然後才是 HWND。在上述列表中我們看到，針對 LPRECT 參數，程式碼將一個 EBP-30h 位址推到堆疊中，因此，在 [EBP-30h] 位址處一定是個 RECT 型態的區域變數。這是一份意想不到的資訊，因爲 WINDEF.H 內含 RECT 結構格式（4 個 DWORDs），所以我們可以獲知堆疊中所有的 RECT 欄位：

```
RECT.left   = [EBP-30]
RECT.top    = [EBP-2C]
RECT.right  = [EBP-28]
RECT.bottom = [EBP-24]
```

再一次，我們可以較有意義的符號，搜尋並取代那些 [EBP-xx] 位址。

編譯器可以暫時把區域變數（和函式參數）拷貝到暫存器中。這可以節省一些程式碼空間和執行時間。對著反組譯碼工作時，你必須對那些開始使用暫存器變數的地方保持警惕。只要隨後你又看到那個暫存器被使用，就以有意義的變數名稱取而代之。注意，編譯器（或組合語言程式員）可能在不同地點以同一個暫存器存放不同的變數。

在 16 位元程式中，SI 和 DI 最常被用來當做暫存器變數。由於這兩個暫存器只有 16 位元長，它們通常不用在指標身上，因爲 16 位元程式中的指標大部份是 32 位元遠程指標。SI 和 DI 通常用於 16 位元值如 HWNDs 或 DCs。在 Win32 程式中，ESI、EDI 和 EBX 最常被用來當做暫存器變數。Win32 的指標是 32 位元近程指標，所以這三個暫存器也可以用來放置指標。啊，沒有任何一條路是堅固或快速的，處理暫存器變數時，請運用你的直覺和判斷。

## 辨識全域變數

決定程式所使用的全域變數就容易得多。幾乎任何一個被寫死的記憶體位址處，就是一個全域變數。全域變數不需要像 EBP 這樣的東西輔助才能夠定出位置來。在 32 位元碼中，全域變數可能像這個樣子：

```
MOV EAX, [00464398]
```

如果你夠幸運，而且反組譯器有符號表格可以參考，或許 [00464398] 會以原始碼的變數名稱取代之。如果不是這樣，你必須搜尋整個反組譯列表，自行以符號名稱取代 [00464398]。如果你手中沒有符號表格，可嘗試自行決定有意義的名稱。

在 16 位元碼中，辨識全域變數的工作和 32 位元碼差不多，差只差在 16-/32- 位元。但如果該程式有多個資料節區，你必須特別小心。問題出在於相同的偏移位址可能出現在好幾個資料節區中。當你處理 DGROUP 以外的資料節區的全域變數時，程式碼會設定一個節區暫存器（通常是 ES）指向該節區，然後程式碼才以該暫存器配合原先寫死的偏移值，例如 MOV AX, ES:[001C]。

如果你手上有符號表格，但是你所遭遇的記憶體位置並不在全域變數列表之中，可能有兩種情況。第一種情況是該記憶體位址被用於一個 static 變數。如果你的符號表格只含 public 符號，該變數就不會顯示出來。第二種情況是，你可能看到了一個結構或陣列中的一個欄位。例如 16 位元程式有全域變數 MSG MyMsg; 位於 DGROUP 節區的 0364h 位置，其中的 wParam 欄位佔 4 個位元組，那麼 MyMsg.wParam 就應該在 0368h 處。此程式的符號表格將列出 MyMsg 位於 0364h 處，但不會列出 0368h 處有什麼東西。

不會什麼收穫都沒有！尋找最接近 0368h 位址的符號，我看到有個名叫 MyMsg 的傢伙在 0364h 處。根據這個名稱我假設那是一個 MSG 結構。當然我必須測試我的假設。如果 0364h 處真的是一個 MSG 結構，0368h 處就應該是這結構的一個欄位，對嗎？對！

但是在我尚未確定我的假設正確之前，我應該尋找其他的擔保。0368h 處看起來像不像 WPARAM？其下一個欄位（036Ah）看起來像不像一個 LPARAM？不幸的是沒有堅固而

且快速的技術能夠派上用場。我必須繼續維持我的假設，直到我有了足夠的自信。

好消息是，編譯器很少把全域變數放到暫存器中。

### 辨識字串

許多 API 函式都需要字串參數。把函式的字串參數對應爲 ASCII，你就比較能夠對函式在做什麼有些念頭。例如在 16 位元碼中你可能遭遇以下的指令：

```
PUSH DS
PUSH 0437
CALL GETMODULEHANDLE
```

在 32 位元碼中你可能遭遇以下的指令：

```
PUSH 00471784
CALL GETMODULEHANDLE
```

打開你的 API 手冊，你會發現 *GetModuleHandle* 有一個參數，是個字串指標。那些 PUSH 指令就是把字串指標推入堆疊，當做 *GetModuleHandle* 的參數。因此在位址 00471784 處（或 16 位元碼中的 DS:0437 處）應該有一個以 null 做爲結束字元的字串（例如 "USER32"）。如果你的反組譯器把資料節區傾印出來，請你前進到該位址並取出其字串內容，然後回到程式碼中，加上一段註解，例如：

```
PUSH 00471784          ;; “USER32”
CALL GETMODULEHANDLE
```

如果你所反組譯的程式使用一大堆字串，你可能會對經過這樣處理之後的結果大感驚訝，因爲實在是清爽太多了。這個過程在反組譯程序中是比較棘手也比較耗時的。

有些可執行檔把字串放在 code 節區中。通常該字串會放在使用它的程式碼的下方。好的反組譯器可以注意到這種情況並且暫時切換到 hex 傾印模式。然而一般的反組譯器常常會出錯。有時候你必須看看週遭的碼，才能決定函式從哪裡開始、資料在哪裡結束。通常，內嵌資料（如 switch 指令所產生的 JMP 表格）會在你的反組譯碼中產生一些暫時性的垃圾。看看週遭的碼，常可以獲得一些線索，告訴你什麼是真正的碼，什麼是內

嵌資料。你可以把這些結果回饋給反組譯器，做出第二次列表，區分程式碼和資料。沒有人說過反組譯是件容易的差事，對不對！

## 辨識 if 指令

最簡單的條件判斷是 if 動作：

```
if ( some test ) {
    do some sequence of code
}
```

在討論其各種變化之前，我要顯示其組合語言碼。從反組譯碼看來，你會遭遇三種主要的測試形式：

- 相等測試：if(a==b)、if(a!=b) ...等等。
- 布林值測試：if(a)、if(b) ...等等。
- 位元測試：if(a & 0x0040) ...等等。

雖然編譯器對不同型態的測試產生不同的碼，,其目標都在設立或清除 CPU 的 Zero Flag（ZF）。在設立或清除 Zero Flag 之後，程式碼使用 JZ（Jump if Zero）或 JNZ（Jump if Not Zero）做條件判斷，看是要執行或跳過後續的碼。你也可以把 JZ 想成是 JE（Jump if Equal），把 JNZ 想成是 JNE（Jump if Not Equal）。

「測試，然後有條件地跳離」這種模式的基本演算法是：如果測試結果爲否定，CPU 就進行條件式跳離，後續的 { } 或 BEGIN/END 中的碼就不會被執行。如果測試結果爲肯定，執行順序就不會跳脫，於是 { } 或 BEGIN/END 中的碼就會被執行。

警告：我在這裡所描述的只是最最簡單的一種情況。真實世界中的碼可能複雜得多。例如在 16 位元程式中，可能有一個 JZ 或 JNZ 用以跳離正規的 JMP 指令。這種情況發生在「if 區塊中的碼比 127 位元組長」時 -- 那是 16 位元碼對一個條件式跳躍的限制。當然啦，前面我所描述的基礎前提並沒有改變。

對於「相等測試」，編譯器使用 CMP 指令。以下是 "DUMPBIN/DISASM" 的輸出片斷：

```
0000101E:  cmp  dword ptr [ebp-04],04
00001022:  jne  0000102E
00001028:  inc  byte ptr [ebp-04]
0000102B:  inc  byte ptr [ebp-08]
0000102E:  ...
```

第一個指令把 [EBP-04] 處的 DWORD 拿來和 4 比較。如果相同，CMP 指令就設立 Zero flag，否則它就清除 Zero flag。下一個指令（JNE）跳過後面的碼 -- 但只有在 Zero flag 被清除時才如此。因此，兩個 INC 指令只在 Zero flag 設立時才得以執行。以 C 語言重寫，我們獲得：

```
if ( SomeVariable1 == 4 )
{
   SomeVariable1++;    // INC [EBP-04]
   SomeVariable2++;    // INC [EBP-08]
}
```

如果 if 指令只要測試 TRUE 或 FALSE，編譯器對於機器碼的產生有兩種選擇。一種選擇是做出像前述 if 指令那樣的碼，例如 if(MyVariable) 也可以被視爲 if(MyVariable!=0)。另一種情況是當詞句中的數值位在暫存器中。這種情況下編譯器可以使用較少的指令來決定是否其值爲 TRUE 或 FALSE。所謂較少的指令是一個 "OR register, register" 指令，像這樣：

```
0000102E:  call  00001000
00001033:  or    eax,eax
00001035:  je    0000103E
0000103B:  inc   byte ptr [ebp-04]
0000103E:  ...
```

其中第一個指令呼叫一個函式，傳回值放在 EAX 中。然後編譯器不再使用 CMP EAX, 0 這樣的指令，改用 OR 指令。OR 指令對 EAX 中的每一個位元都做 logical OR 運算。如果沒有任何一個位元處於設立狀態（也就是說 EAX==0），Zero flag 便會設立。

第三種情況是位元測試。Windows 程式設計中有許多 WORD 和 DWORD 是以一個位元一個位元的旗標值組成，例如 *CreateWindow* 所需要的 WS_xxx。程式常常利用 AND 運算來檢查是否哪一個旗標設立起來。看看下面這一段 C 程式碼：

```
DWORD winFlags = GetWinFlags();
if ( winFlags & WF_CPU386 )
     is386 = TRUE;
```

產生出來的組合語言碼像這樣：

```
0000102E:  test  byte ptr [ebp-08],04   ;; WF_CPU386 == 0004h
00001032:  je    0000103F
00001038:  mov   dword ptr [ebp-0C],00000001
0000103F:  sub   eax,eax
```

第一個指令使用 CPU 的 TEST 指令看看是否某個位元有設立起來。TEST 指令會對兩個運算元執行 logical AND 運算，但不會改變任何一個運算元的值。如果運算結果並沒有任何位元設立，Zero flag 會設立起來，否則就會清除爲 0。當 Zero flag 設立，JE 指令就不會跳離，於是 [EBP-0C] 中的 DWORD 值會變成 1。

如果你小心地觀察前面那個片斷中的 TEST 指令，你會發現某些奇怪的東西。在 C 程式碼中，winFlags 是一個 DWORD，但是組合語言碼卻只在意其最底部的 BYTE。這是因爲編譯器做了最佳化，儘可能使用最少量指令。如果沒有最佳化，上述的第一行應該是：

```
TEST DWORD PTR [EBP-08], 00000004
```

我提出這一點並不是要強調最佳化與否，而是告訴你觀察 TEST 指令時必須機伶些：被 TEST 的位址和位元遮罩可能不是你預期的樣子。前一例中如果我們改測試 WF_80x87，其值爲 00000400h，TEST 指令應該變成 "test [ebp-08], 00000400h"，是嗎？錯！它會變成 "test [ebp-07], 04"。看起來記憶體位址比 winFlags DWORD（在 ebp-08 處）高了一個位元組。爲了補償這一點，編譯器把將被測試的位元旗標向右移 8 個位元。這簡直是有點卑鄙！如果被測試的位元放在變數中，編譯器還會把它的位址調出來，依樣畫葫蘆。

現在我們看過了所有三種基本的 if 判斷型態。讓我們再來點大的。比單獨的 if 句子稍稍複雜些的是 if-else 句子。考慮下面這段碼：

```
if ( i == 4 )
    {
        i++;
        j++;
    }
    else
        j—;
```

編譯器爲它產生這樣的碼：

```
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:  ...
```

前兩個指令看起來和單獨一個 if 敘述句的情況一樣。較遠的地方有個 JMP 指令，那是關鍵所在。JMP 使得執行完「判斷句爲真」的回應動作之後，就跳過 else 子句。JMP 所跳的位址是「else 子句何處結束」的重要線索。

當你嘗試辨識一個 if-else 句型時，注意兩件事情：一開始的 JE/JNE 指令所跳過去的位址，是否是緊接在一個 JMP 指令之後？JMP 指令是否跳到一個更高的位址（也就是說向後跳而非向前跳）？

另一種比較複雜的 if 判斷句型是多重條件。例如：

```
if ( (i == 4) && (j == 2) && (k == 6) )
    {
        i++;
        j++;
    }
```

編譯器產生如下的碼：

```
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:  ...
```

這些碼十分直接了當，有三個測試接續發生。其中任何一個失敗，程式碼就會跳離 { } 區域。如果你看到許多「測試並跳離」的組合，可能你面對的就是一個有多重條件的 if 句型。

多重條件的 if 句型中，以 OR 串連子句和以 AND 串連子句的情況相差無幾。你同樣會看到一系列連續的「測試並跳離」動作。所有的測試，除了最後一個之外，都跳到 { } 之外 -- 如果其測試結果都是 TRUE 的話。如果測試結果爲 FALSE，就會掉到 { } 之內的下一個指令。如果最後一個測試失敗，也是跳離 { }。

這一節涵蓋的是基礎句型。我並沒有討論 for 迴路或 while 迴路。你或許會遭遇更複雜的東西，然而幾乎你所遭遇的每一種句型都可以被分解成我所描述過的這幾種的組合和變形。

### 辨識 switch 指令

在 class library 產品如 MFC 和 OWL 降臨之前，大部份 Windows 程式都還是在其視窗函式的一開始有一個大型的 switch 句型。switch 句型把各式各樣的視窗訊息導引到適當的處理常式去。如果你需要觀察一個程式的視窗函式以決定它如何處理某個訊息，你就必須知道怎麼剖析一個 switch 句型。幸運的是，那並不太困難。

剖析一個 switch 句型的一般程序是：對於每一個條件式 jump，都到目標地去看看，並在該處放一個大大粗粗的註解。這對於發生在視窗函式中的 switch 句型特別有幫助。對每一個 WM_xxx 訊息處理常式，就以 WM_xxx 訊息名稱做爲註解。例如：

```
; CASE WM_NCHITTEST
00413254:   XOR     EAX,EAX
00413256:   JMP     00413454

; CASE WM_GETTEXTLENGTH
0041325B:   MOV     EAX,[cbTextBuffer]
00413260:   JMP     00413454
```

辨識一個 switch 句型非常非常地簡單 -- 雖說它也有三種常見的變形。最簡單的一種就是我所謂的白痴型編碼。它浪費許多空間，非常容易推斷，組合語言碼看起來像這樣：

```
MOV EAX,[EBP+0C]
CMP EAX,00000045
JE  someAddress
CMP EAX,00000169
JE  someAddress2
CMP EAX,00000265
JE  someAddress3
```

第一個指令把 switch 的參數載入暫存器中 -- 本例爲 EAX，但也可以是其他暫存器如 EDI。16 位元碼則似乎總是使用 AX。

將欲測試之數值載入暫存器後，程式碼進入一系列的 CMP/JE 組合。對於 switch 句型中的每一個 case，都有一個對應的 CMP/JE 組合。於是我們就很容易找到某個 switch 測試值的處理常式了。如果程式使用 switch 來分派視窗函式中的訊息，你只要在組合語言碼中尋找你感興趣的 WM_xxx 即可。那是一項簡單的工作 -- 只要搜尋 CMP 指令並比對其測試值即可。緊跟其後的 JE 指令內含有訊息處理常式的位址。如果你要把整個句型拆卸開來，看看它如何處理每一個訊息，那麼以訊息名稱做爲處理常式的標記是很有幫助的。

switch 句型的第二種變形十分接近第一種變形，差別在於測試指令使用較少的位元組，並且要求你對中間過程的數值保持注意。看看下面這段碼：

```
MOV EAX,[EBP+0C]
SUB EAX,2
JE  someAddress
DEC EAX
JE  someAddress2
DEC EAX
JE  someAddress3
SUB EAX,5
JE  someAddress4
```

一瞥之下，這段碼看起來令人迷惑。它並不像第一種類型那樣去比較任何數值，唯一真正的動作是 EAX 的值不斷往下掉。爲了讓這段碼看起來比較合理，你必須知道，DEC 和 SUB 指令的運算結果若是 0，Zero flag 會設立起來。每一個 DEC 和 SUB 指令都會把

輸入值吃掉一些。當該值爲 0，時候便到了，JE 指令便把它分派到適當的處理常式去。輸入值較低，分派出去的時間較快；輸入值較高，分派出去的時間較慢。

爲了看看 JE 指令中到底是哪一個值被測試，你必須將先前所有被減掉的值做個總和。面對這種 switch 句型，我發現在每個 JE 指令旁做上「目前數值」的標記很有幫助。下面就是我對上一段碼的註解：

```
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.)
```

switch 句型的第三種變形稱爲跳躍表格（jump table）。如果輸入值十分接近，編譯器也許會建立一個位址陣列，每一個陣列元素都對應一個 case 值。這種作法的好處是執行速度很快，因爲不需要對每一個可能的輸入值做檢驗。請看下面這段碼：

```
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;
}
```

編譯後的結果如下：

```
00001008:  mov  eax,dword ptr [ebp-0C]
0000100B:  cmp  eax,09
0000100E:  ja  00001068
00001010:  jmp  dword ptr [eax*4+0040108F]
```

第一個指令把 switch 的輸入值放到 EAX 中。後面兩個指令判斷輸入值是否在合法範圍內。如果不是，JA 指令會跳離 switch 句子。最後一行碼利用 EAX 做爲陣列索引，找出處理常式的位址，然後跳到該處去。

> 譯註：寫過 VxD 的朋友們，一定不會對此法陌生。許多 VxD 入門範例就是以這種組合語言寫法來判斷系統傳給 VxD 的 events。

前面程式碼中，編譯器把處理常式的位址陣列放在可執行檔的資料節區中。然而如果陣列緊跟在 JMP 指令之後，你也別太驚訝。這在 16 位元程式中特別普遍。這時候 JMP 指令使用 CS 做爲記憶體位址的一部份。這種情況下你可能會看到一些沒用的暫時性指令，那是因爲反組譯器不知道那些位元組到底是程式碼還是資料。一個好的反組譯器應該能夠辨識出這種形勢，或者至少讓你告訴它「code 節區中的某一部份其實是資料」。

## 一個反組譯實例

我已經涵蓋反組譯器的基本觀念了，讓我們看一個實際例子，示範如何運用這些觀念。我將以 Windows NT 的 CLOCK.EXE 爲例，它可以切換程式狀態爲有（或沒有）視窗標題。我選擇這個程式有兩點理由。第一，我已經從 spy 的角度檢驗過這個程式了，我們可以更進一步測試，然後比較兩個方法所得的結果。第二，微軟提供 CLOCK.EXE 原始碼給 Win32 程式員，所以你可以判斷反組譯的精準度。

對這個例子，我使用自己的反組譯器。微軟的 DUMPBIN 當然也可以，但我的反組譯器會自動做某些事情（特別是把 API 呼叫與其符號名稱相配起來），而那卻是使用 DUMPBIN 做爲工具時你必須手動完成的。下面是反組譯器的最初輸出結果：

```
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
```

最前面兩行和最後面三行就是 prologue 碼和 epilogue 碼。其中只有兩件事有點趣味："RET 0004" 告訴我們這個函式需要一個參數（Win32 中的所有參數都是 4 個位元組）。第二，程式碼沒有設立 EBP 堆疊框架，所以我們必須追蹤堆疊上的東西，以便決定參數在哪裡。

夠幸運的了，這段碼之中只有一個指令用到堆疊中的參數。那是 prologue 碼之下的：

```
MOV ESI, DWORD PTR [ESP+0C]
```

這個指令把參數拷貝到 ESI 中，ESI 會在其他許多地方被用到。看起來 ESI 似乎是某種暫存器變數。唔...ESI 可能是什麼呢？掃描整段碼，我們發現 ESI 被用做 *GetWindowLong*、*SetWindowLong*、*SetWindowPos*、*ShowWindow* 的參數。會不會 ESI 是個 HWND？看起來的確是。

讓我們以我們所發現的東西，重寫上述程式碼，並消除 prologue 和 epilogue 碼：

```
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
```

現在，我們要把數個函式呼叫（*GetWindowLong*、*SetWindowLong*、*SetWindowPos* 和 *ShowWindow*）轉換爲 C 型式。這些函式的參數要不是我們所發現的 hWnd，就是一個可以從 WINDOWS.H 中查到的數值。讓我們重寫程式碼如下：

```
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 );
```

雖然我們可以把一些函式改寫爲 C 型式，卻還沒有足夠的資訊了解最後兩個 *SetWindowLong* 的參數是什麼。對於其中之一，我們必須知道 EDI 內含什麼，對於另一個我們則必須知道 [012F6000] 位址處是哪一個全域變數。

等等！我們已經看到，*GetWindowLong* 取出代表視窗風格的旗標值，拷貝到 EDI 去。EDI 或許是另一個暫存器變數，用來儲存視窗風格旗標位元。而對於 [012F6000] 位址，請注意程式碼將 *SetWindowLong*(GWL_ID) 的回返值儲存到其中。稍早前我曾說過，對 top-level 視窗而言，視窗識別碼欄位（GWL_ID）係用來儲存 HMENU。把這些事實串在一起，你可以猜得出來，[012F6000] 是一個全域變數，內含一個 menu handle（HMENU）。

讓我們根據這兩項新發現重寫整段程式碼：

```
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 );
```

現在只剩下 [012F3B10] 處的條件句了。CMP 指令比較 [012F612C] 中的全域變數是否爲 0，然後立刻以一個 JMP 指令進行條件式跳躍。看起來這似乎是個標準的 if-else 動作。[012F612C] 看起來似乎是某種布林值（boolean）。暫時給它一個名稱吧，就叫 "MyBool" 好了。安插一些 { } 並縮排，程式碼現在變成這樣：

```
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 );
```

真令人驚訝。我們把一段組合語言碼轉換爲易讀的 C 語言碼。如果你把這段碼拿來和 spy 的觀察結果比較，你會發現一切吻合。然而反組譯所獲得的資料比你從 spy 工具處所得到的豐富得多。例如，spy 軟體就絕對不會告訴你有兩個全域變數牽扯在其中（一個 HMENU 和一個布林值）。

對某些人而言，這段動作（組合語言碼到 C 語言碼）或許是太快了些。並不是每一次反組譯都能夠這麼順暢這麼快速。我希望我已經給你留下這個印象了：反組譯過程需要一次一次地修正。做一些假設、找出程式碼顯示之外的資訊、然後把它回饋到程式碼中、再做一些修正...。

最後我要提醒你一點，對於「把有問題的碼放到除錯器去深入觀察」這一點，千萬不要猶豫不決。看到程式碼以活生生的數值運轉著，常常可以突破心理上的障礙。許多時候我沒辦法理解某個函式傳回什麼東西，但是以除錯器進去看實際的傳回值之後，我常常就能夠演繹出某種模型，例如「某個函式總是傳回一個 global memory handle」等等。重點是，每一小片線索都可能有幫助。你或許會驚訝那麼小的東西可以幫助你打開那麼大的視野。

# 高階秘訣

在本章結束之前，我想，給你一些一般性的錦囊妙計，應該會有幫助。

## 使用 SoftIce/Windows

如果你對挖寶工作有興趣，SoftIce/Windows 是必需品。在我更進一步討論之前，我有義務指出我爲 Nu-Mega 工作，這家公司是 SoftIce/Windows 的出品公司。這是唯一可以和微軟的 WDEB386 系統級除錯器相抗衡的產品，而後者並沒有許多方便的命令可以用（用以傾印資料結構啦、使用除錯符號資訊啦等等）。

SoftIce/W 之所以有強大威力，因爲它是系統層級的除錯器。和一般應用程式層級的除錯器（例如 Turbo Debugger、CodeView、或編譯器整合環境中附帶的除錯器）不同，SoftIce/W 不依賴 Windows 的任何事物，它在 Windows 和硬體之間動作。所以，SoftIce/W 可以進入系統任何碼之中，包括 ring0 VxDs 和真實模式的 DOS 碼。這在研究 Windows 排程器及 memory context 切換時非常有用。一般除錯器想都別想這一點。

微軟的 WDEB386 在這一部份有類似功能。Windows 95 的一個缺點是，你沒辦法以應用程式除錯器進入 ring3 system DLLs（如 KERNEL32.DLL），因爲 Windows 把它們讓各程式共享，如果在其間擺上一個 INT 3，幾乎總是會把整個系統搞當掉。至於 SoftIce/W，因爲在系統底層下工作，沒有這種限制。

和一般除錯器不同，你不需要在 Windows 中載入 SoftIce/W，也不需要刻意透過它對某個程式除錯。SoftIce/W 載入於系統底層之下，並總是存在。除非你按下熱鍵，否則它就安靜地待在一旁。因此，你能夠在每次啓動 Windows 時就載入 SoftIce/W。當你需要它，就讓它突然冒出來；平時就忽略它的存在。它像一個超級的 Windows，你可以隨時停下來自由意願地改寫一些東西。有點像是 Windows 中的 TSR（常駐程式）。

和 WDEB386 不同，SoftIce/W 內含許多命令，可以傾印出各種層級的資料結構和串列。在最低層級，它可以傾印出 CPU 的 page tables，以及 Global Descriptor Table 和 Local Descriptor Tables 中的 selectors。較高一層，它可以顯示重要的 VxD 相關項目，像是 VxDs 串列、VxD 的 Device Descriptor Block 等等，以及用來維護位址空間的 context tables 內容。事實上，SoftIce/W 甚至可以把 address contexts 切換到任何一個 memory context 中，使你能夠看到系統中所有行程的所有記憶體。再上一層，SoftIce/W 可以列出所有的 processes、threads、modules、Win16 tasks，以及其中的每一個細節項目。其價值簡直無法估計 -- 如果你的碼使用一個由系統提供的 handle 而你需要知道該 handle 參考到什麼東西的話。更上一層，SoftIce/W 可以顯示視窗及其類別（window class）的詳細資料。我說這些不是要爲 SoftIce/W 贏得多少讚美，而是要告訴你，你的手邊可以擁有這麼多的系統資訊。

有些 SoftIce/W 功能特別有用又十分簡單，我要在這裡提出說明。當你進入 16 位元碼，常常會遇到一些 handles，看起來像是 global heap handles。只要在 SoftIce/W 的 HEAP 命令中以該 handle 做爲參數，你立刻就可以驗證那是否爲一個合法的 global heap handle。如果它是，SoftIce/W 會告訴你那個 handle 的目的是什麼（擁有者是誰？它是程式碼還是資料？還是資源？如果是程式碼，位在 NE 檔的哪一個節區內？）

不論 16- 或 32- 位元，SoftIce/W 的 U 和 D 命令都非常方便。已知一個程式碼位址，你可以把它當做 U 命令的參數，並因而快速獲得一段反組譯碼。D 命令則允許你以各種格式觀看記憶體內容。和其他的除錯器或應用程式不同，這兩個命令對於觀察對象的身份可說是百無忌憚。

SoftIce/W 超越其他除錯器的另一個地方是，它可以從 16-/32- 位元的 EXE/DLL 檔案中載入輸出函式（exported functions）的資訊，然後把它們當做虛擬符號表格。於是，不論你在 Windows 的何處停下來，SoftIce/W 都能夠告訴你你目前正使用哪些 EXE/DLL/VxD，並且也常常能夠指出精確的位置。它甚至把這份表格用在反組譯過程中，因此你看到的將不是：

```
CALL  BFFB0149
```

而是：

```
CALL  GetModuleHandleA
```

另一個極好的 SoftIce/W 命令（存在於 Win32s 和 Windows 95 版）是 MAP32 命令。有此命令，你就可以輕易決定 EXE 及其 DLLs（包括 system DLLs）要駐留在記憶體何處。另一個好東西是 SoftIce/W 的 STACK 命令，不論在什麼地方停下來，你都可以獲得一個 call stack 報表，讓你知道你自己是怎麼進行到那一點上的。任何時候如果你想知道目前的 task 或 thread，請用 TASK 和 THREAD 命令。

## 使用硬體中斷點（hardware breakpoint）

如果希望在某種情況下進入某段程式碼去看看，你可以利用 CPU 的硬體中斷點。例如，我想知道某個很少被用到的執行緒何時作用起來，首先我利用 THREAD 命令獲得這個執行緒的 ID，然後在執行緒切換之後的執行點設立一個中斷點（breakpoint）。有效嗎？沒有，因爲系統不斷地在切換執行緒，中斷點不斷地被觸發，但是"current thread" 卻總不是我們所要觀察的那個執行緒。

爲了解決這個問題，我們找到系統用以記錄 current thread ID 的那個 DWORD （提示：請看 *GetCurrentThreadId* 的反組譯碼），然後在該處設定一個有條件的硬體中斷點。條件如下：當我們感興趣的 thread ID 被放到那個 DWORD 之後，硬體中斷點才得被觸發。問題解決了，系統會正常執行，直到我們做了某些動作使「我們感興趣的那個執行緒」醒過來爲止。

另一個例子是：我的程式使用 *SetThreadContext* 改變另一個程式的 EIP。*SetThreadContext* 總是告訴我成功，但另一個程式總是當掉。爲了瞭解到底發生什麼事，我在「thread context 結構中用以記錄 EIP」的那個 DWORD 設了一個硬體中斷點。執行該程式後，我發現 *SetThreadContext* 的確是把 EIP 拷貝到正確位置，不幸的是，在稍後被觸發的中斷點，我看到 KERNEL32.DLL 以一個莫名其妙的東西改寫了我的 EIP。如果沒有使用硬體中斷點，可能到現在我還在懷疑我有什麼臭蟲，或 Windows 95 有什麼臭蟲呢。

## 使用 VxD 的 .（句點）命令

WDEB386 和 SoftIce/W 的使用者可以經由 .（句點）命令獲得一些系統資訊。句點命令（之所以這麼稱呼是因爲它們都以一個句點開始）被實現於各個 VxDs 中。爲使用它們，請進入你的系統除錯器中，在提示號下輸入命令名稱（總是以句點開始）。某些是隨時有作用，某些則只在除錯模式中才有作用。下面是句點命令的一部份：

```
..?         .vtd
.m?         .dosmgr
.vmm        .vmpoll
.vxdldr     .vtdapi
.vpicd
```

## VAR2MAP 工具程式

在 Windows 95 中，Win32 system DLLs 如 USER32、KERNEL32 等等都算是基礎函式庫。也就是說每次你啓動 Windows 95，它們總是被載入到相同位址。你可以根據這一事實加上一些很難被別人超越的知識：這些函式和變數被載入到 WDEB386 或 SoftIce/W 的什麼地方。這可以讓你在進入那些 system DLLs 時使用符號。怎麼做？看看 KERNEL32.DLL 中的 *GetSystemDefautLangID* 函式：

```
GetSystemDefaultLangID proc
BFFB69FD:      MOV      AX,[BFFD44D0]
BFFB6A03:      RET
```

很明顯 BFFD44D0h 位址處有一個全域變數，名爲 SystemDefaultLangID。由於 KERNEL32.DLL 在線性位址空間中有一個獨一無二的基底位址，SystemDefaultLangID

將總是位於位址 BFFD44D0h 處。如果你能夠告訴你的系統除錯器這個事實並讓它在反組譯時自動把 BFFD44D0h 改變爲 SystemDefaultLangID，豈不甚妙？我也這麼想，所以我寫了一個 VAR2MAP 程式。

欲使用 VAR2MAP，你必須寫一個檔案，內含一系列的 32 位元位址及其相關名稱。檔案內含可以是變數名稱也可以是函式名稱。唯一限制是所有的名稱和位址都必須在同一個 EXE 或 DLL 中。VAR2MAP 以此檔案做爲輸入，吐出一個 .MAP 檔。.MAP 檔有什麼了不起？喔，你可以使用微軟公司的 MAPSYM 或 Nu-Mega 公司的 MSYM 處理 .MAP 檔，它們都會吐出一個 .SYM 檔。WDEB386 和 SoftIce/W 都知道如何載入.SYM 檔，用於符號式反組譯。我寫這本書期間不斷使用 VAR2MAP 取得 KERNEL32.DLL 和其他 system DLLs 中有意義的函式符號名稱。

VAR2MAP 程式的一個典型輸入檔顯示於下：

```
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
```

所有其他各行都應該是這樣的型式：

```
SymbolName = AddressInHex
```

檔案的第一行必須指定內含這些位址的 EXE 或 DLL 的磁碟路徑。爲什麼需要這樣？如果你看過 .MAP 檔，你會發現所有 public 符號的位址都是邏輯位址，也就是 ObjectNumber:Offset （例如 0004:00013484）。VAR2MAP 需要 EXE 或 DLL 檔名以求瞭解是否每一個 code section 或 data section 都將映射到記憶體中。這使得 VAR2MAP 得以把你給予的線性位址轉換爲邏輯位址如 0004:00013484。

產生輸入檔後，執行 VAR2MAP 並在其命令列中指定輸入檔名稱（例如 VAR2MAP KERNEL32.VAR）。吐出的 .MAP 檔將放在與第一行（ FILE= 那一行）所指定檔案的同一個磁碟目錄。這很合理，因爲你所產生出來的 .SYM 檔案必須和其對應的 .EXE 或 .DLL 位於同一個磁碟目錄。否則除錯器不知道要把 .SYM 載入記憶體。請參考你的除錯器文件中有關於載入 .SYM 檔的部份。

## 辨識 VxD services

對於目前這種探險型態，如果能夠知道 VxD services 如何被呼叫，以及呼叫動作是如何被完成，將頗有助益。所謂 VxD services 是指由 VxD 開放出來給其他 VxDs 呼叫的函式。一個 VxD 不會直接呼叫 VxD service（至少在它初次被載入時），你必須在 VxD 中使用 INT 20h 才能呼叫 VxD services。DDK 的 VMM.INC 中有一個 *VxDCall* 巨集就做這件事情。INT 20h 由 VMM 處理。VMM 會把 INT 20h 轉換爲「呼叫一個真正的 VxD service 位址」。

INT 20h 如何知道你指定了哪一個 VxD service？緊跟在 INT 20h 之後是一個 DWORD，它透露出一些線索。這個 DWORD 的較高字組（high WORD）代表「內含此一 service 之 VxD」的 device ID，較低字組（low WORD）代表該 VxD 的 service table 中的 service 編號（從 0 開始）。

較高字組（high WORD）所代表的如果不是個標準的 VxD IDs（定義於 VMM.INC 中），就是某個公司的 VxD IDs（必須由微軟公司指定）（譯註：如果只是寫個 VxD 玩玩，或只做爲公司內部使用，當然就不需微軟指定 VxD IDs）。最前 16 個 VxD IDs 是：

```
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
```

如果你使用某個標準 VxDs，通常你會在 DDK 的對應檔案（.H 或 .INC）中找到一系列的 VxD services。VxD 程式中的 Begin_Service_Table 巨集就是用來表示 service 列表開始。VxD 所供應的每一個 service 都在程式碼中有一行以 <VxD Name>_Service 開頭的敘述句對應之。第一行所列的 service 編號爲 0，第二行編號爲 1，依此類推。有了這些知識，你就很容易知道如何在 INT 20h 之後的 DWORD 中標記出一個已知的 VxD service 了。例如我們知道 VMM 的 device ID 爲 1，於是可以輕易計算出 VMM 中每一個 service 所對應的 DWORDs 值：

```
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
```

除錯器或反組譯器只要保持一個巨大的對應表格，便能夠把程式碼中的 DWORD 編號對應爲實際的符號名稱。稍早我曾說過 INT 20h 處理常式把整個動作轉換爲一個 CALL 指令，這其實不完全是事實。如果 DWORD 較低字組（service 編號）的最高位元被設立，INT 20h 會被修補爲 JMP 指令。例如 INT 20h 00018001 將變成 JMP Get_VMM_Version。如果要產生一個 JMP 指令而不是 CALL 指令，你應該使用 *VxDJmp* 巨集而非 *VxDCall* 巨集。

## 辨識 Win32 VxD services

Windows 95 的極佳特色之一就是它有額外的 Win32 VxD services。Win32 services 類似 VxD services，但它能被 ring3 應用程式呼叫。要喚起一個 Win32 service，請你呼叫 KERNEL32.DLL 提供的 *VxDCall0* 函式。

VxDCall0 函式有一個 DWORD 參數，類似前述的 VxD service ID。其較高字組（high WORD）是 device ID，較低字組（low WORD）則是 Win32 service 編號（從 0 開始）。

讓我們看個例子。下面這一片斷來自 KERNEL32.DLL 的 *GetThreadContext*：

```
BFFABD8D:   PUSH    EAX
BFFABD8E:   PUSH    DWORD PTR [EBX+5C]
BFFABD91:   PUSH    002A0014
BFFABD96:   CALL    VxDCall0
```

前兩個 PUSH 指令用來將 Win32 service 的參數推入堆疊。最後一個 PUSH 指令告訴 *VxDCall0* 將 #14(從 0 開始)的 Win32 service 喚起。什麼是 device 2Ah？就是 VWin32 嘛 -- KERNEL32 眾多機能的來源。

## 辨識參數之合法性以及 Ixxx 函式

在 Windows 3.1 中，微軟引進參數檢驗函式，它們會檢查你（程式員）傳給 API 函式的參數是否合法。如果你的參數不合法(例如傳進一個 null 指標)，這個函式會立刻回返，啥也不做。在除錯版中，這個函式甚至還會吐出一些診斷用字串到除錯終端機上。

一個擁有參數檢驗功能的函式通常分爲兩部份。第一部份總是在函式一開始處，用來檢驗參數。如果所有檢驗都通過，就 JMP 到真正的函式碼去。在 Windows 3.1 中，真正的函式碼另有一個名稱 -- 以 'I' 開頭，再緊接著原函式名稱。例如 *WinExec* 檢驗過其參數之後跳到 *IWinExec* 去。本書之中我一直沿用這樣的習慣。

下面是一段有註解的 Windows 95 32 位元 *WinExec* 函式碼。此碼設定一個結構化異常處理（Structured Exception Handling，SEH），然後檢驗 lpszCmdLine 所指的字串。如果檢驗失敗，產生 page fault，於是 SEH 令此函式回返（經由本碼未曾顯示的路徑）。如果檢驗成功，函式將移除 SEH，然後 JMP 到 *IWinExec* 去。

```
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.
```

關鍵點在於辨識 I 開頭的函式。如果你看到類似上面這樣的函式，最終以 JMP 做爲結束，那或許就是個參數檢驗函式。JMP 的目標位址也就是「I 函式」的起頭。你可以利用這份知識建立起額外的符號位址，例如在「VAR2MAP 工具程式」一節中我曾顯示四個「I 函式」：

```
IGetProcAddress = BFF81DC1
IGlobalHandle = BFF76E78
ILocalReAlloc = BFF833C8
ILocalSize = BFF890CB
```

如果你爲你的除錯器加上這四個內部函式名稱，察看系統碼時會比較輕鬆些。似乎參數檢驗層都集中在某個區域，而實際函式碼則散佈在模組各處。加上額外的內部函式符號，你更有機會理解自己身處某個系統模組的何處。它也強化了「堆疊追蹤」作法的合用性。

## 使用除錯版本

除了能夠幫助你找出程式臭蟲，Windows 除錯版還可以讓你比較容易瞭解 Windows 在做什麼。SDK 所提供的 system DLLs 除錯版會產生出有幫助的診斷訊息。那些訊息字串中常會出現函式名稱。此外，有許多訊息會列印（或註解）系統變數的值，通常你回頭看看前幾個指令便能夠發現該變數。舉個例子，看看下面這段碼：

```
BFFC788A:   PUSH    DWORD PTR [ESI+18]
BFFC788D:   PUSH    BFFDBF9C ;;     Default Heap: %8x\n
BFFC7892:   CALL    BFFC6092
```

在這裡，ESI 指向 Win32 process database 結構。第二個指令正把一個指標交給類似

*printf* 那種型式的格式化字串。從這份資料你就可以輕易瞭解 process heap handle 在偏移位置 18h 處。

Windows 除錯版的另一貢獻就是它所提供的強固性檢驗以及類似 assert 那樣的檢驗碼。除錯版 DLLs 總是會檢查傳進來的參數以及系統變數的狀態。你可以利用所有的強固性檢驗碼來確認或反舉證你所猜想的某一段碼是否有效運作。

## Pentium 的最佳化碼

微軟曾宣稱 Windows 95 對 Pentium 做了最佳化。我要告訴你那宣稱是真的。編譯器爲 Pentium CPU 所做的主要最佳化動作是辨識某些指令，讓 Pentium 的兩個執行單位能夠一起執行沒有拖延。請看看 KERNEL32.DLL 的一小片斷：

```
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
```

指令 1,2,3,5,6,7 組合成 prologue，設定函式的堆疊並保存某些暫存器值。指令 4,8 則組合成一個 if 標準動作。如果沒有做 Pentium 最佳化動作，程式碼看起來會容易理解一些。JBE 指令會立刻在 CMP 之後發生，而不是在四個指令之後。

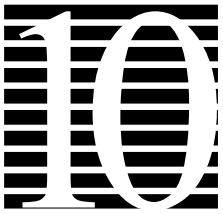
我說這個幹嘛？當我們進入 Pentium 最佳化碼（例如 Windows 95 的 system DLLs），其指令可能不會一眼望去便覺有理。你必須知道這種情況並 look for instructions sequences that are doing two different things（譯註）。一般而言我會嘗試讓這些程式碼比較率直了當一些，作法是把它們重新安排爲兩組碼。大部份時候，這兩組碼對應不同的 C 原始碼。重新安排之後，我先看第一組，再看第二組。

譯註：這一句我不是很有感覺，不敢造次，保留原文。請原諒。

# 摘要

這一章顯示了數種不同的探險方法。低檔方面，我使用最簡單的檔案傾印工具。中級層面用的是刺探工具（SPY）。如果你對於應用程式和作業系統之間的互動感興趣，SPY 非常管用。高檔方面，反組譯器可以破解任何程式。反組譯是零亂的、不嚴謹的、令人深感挫敗的工作。但它可能是十分有價值的資產，只是問津者日鮮矣！

如果你還未曾使用過我說的工具和技術，我希望這樣的討論能夠拿掉它們神秘的面貌，使你在需要的那一天願意使用它們。雖然這些工具在某方面的表現似乎過於神奇，但其實並非如此。如果你在硬體和作業系統方面有堅固的基礎，那些工具和技術可以視爲你工具箱中的另一部份，而不是爲某些程式魔術師保留的什麼奇妙法門。



# 寫一個 Win32 API SPY 軟體

譯註：本章常出現「輸入函式」和「輸出函式」兩個名詞，我指的是 import function 和 expert function。如果你在程式 A 中呼叫 USER32.DLL 的 *GetMessage* 函式，那麼 *GetMessage* 就是程式 A 的「輸入函式」，它也必然是 USER32.DLL 的「輸出函式」。

如果你看到「輸入函式庫」，我指的是 import library。例如 USER32.LIB 就是 USER32.DLL 的「輸入函式庫」。

這一章可以用鬼斧神工來形容。要完成一個非常有彈性，而且面面俱到的 spy 軟體，你需要許多作業系統的知識，以及編譯器的知識。你必須熟悉 PE 可執行檔格式（尤其在 PE header 以及 import section 方面），你必須熟悉函式呼叫時的參數傳遞方式以及回返位址的設定（也就是堆疊的狀態），你必須知道中斷點與除錯訊息，以及除錯器與被除錯程式之間的關係，你必須知道有哪些 API 可以跨越行程，處理別人的位址空間中的內容，你也必須知道 thread context 中的 EIP 作用。雖然我已把本章的 spy 程式徹底看懂，也儘全力修潤了許多原本艱澀的說明文字，加了不少譯註，但全章仍嫌瑣碎，示意圖不夠多。我畫了數十張圖，才得以全盤掌握整個 spy 程式的架構。建議你也邊看邊畫邊參透。這裡面沒有提到 C 函式呼叫後的堆疊狀態，而那是 spy 軟體成功的關鍵點（也是極容易出錯的地方）。我已在適當處做了補充。

做爲一個程式員，我們常常在看到別人的程式後驚訝得大叫：他是怎麼辦到的！這時候你需要一個工具，幫助你仔細察看一個執行中的程式表面下的動作，追蹤 EXE 或 DLL 的行爲。要分析執行中的程式，沒有什麼比得上一個 API spy 軟體了。

API spy 軟體可以告訴你 EXE 或 DLL 呼叫了哪一個 Windows 函式。除了顯示函式名稱（依照被呼叫的次序），spy 軟體還可以記錄 API 函式的參數其及回返值。更高級的 API spy 軟體還可以記錄其他資料，如視窗訊息、hook callbacks 等等。有了這些資料之後，我們就可以輕易解讀某一段碼的行爲。第 9 章討論過常見的 SPI spy 軟體，並以一個例子說明如何運用其輸出。這一章，我要建構一個簡單但是頗具威力，並且很容易擴充的 Win32 API spy 軟體。

我用來攔截 API 函式呼叫的方法，也可以應用到你自己的 Win32 程式中。假設你想要以自己的函式取代 Win32 *lstrcpy* 函式，只要一個動作就能夠完成這個目的。如果你只對這個應用感興趣，請直接跳到最後一節「在你自己的程式中攔截函式」。如果你對 Win32 系統程式設計感興趣，並且打算學習這項技術的運作，請繼續讀下去。

爲什麼在眾多威力強大的 API spy 商業軟體（例如 BoundsChecker32）環伺下，我們還要自尋煩惱寫一個簡易型的 API spy 軟體呢？因爲自己動手之後，你就可以全然瞭解 Win32 作業系統的哲學，並對三個不同的 Win32 平台（Windows NT、Windows 95 和 Win32s）之間的差異有清楚的認知。

表面上看，這一章的目標是「如何建立一個 API spy 軟體」。然而我真正的目的是要表現一組真實世界中的 Win32 程式設計問題，並告訴你解決之道。在這個過程中，你會獲得 Win32 架構的面面觀。很快你就會發現，寫一個 API spy 軟體可以強迫你面對像位址空間、多執行緒、動態聯結、除錯機制、行程管理、執行緒控制等題目。簡單地說，我即將創造的程式，會帶領你對許多 Win32 核心觀念好好做一趟瀏覽。

在進入程式細節之前，我必須先列出 API spy 軟體的規格：

1. 對於一個已知的 Win32 行程，spy 軟體應該記錄該行程針對某一組 DLLs 所發出的函式呼叫。
2. 使用者應該能夠經由一個組態檔（configuration file），擴充被監視之 DLLs 的名單。
3. 如果使用者知道某個函式的參數，他可以在組態檔中指定之，於是參數值將和函式名稱一併被記錄下來。
4. spy 軟體必須記錄函式的回返值。
5. spy 軟體應該能夠在 Windows NT、Windows 95 和 Win32s 上執行。
6. 不可以要求被刺探之程式修改其原始碼或可執行檔。
7. 運轉記錄應該能夠儲存到檔案中，不能只是曇花一現。

當這個 spy 軟體在本章結束前完成，它應該可以做下列服務：它允許你指定一個刺探對象，然後執行之，然後產生一個運轉記錄（ASCII 文字檔）。被指定之程式結束後，你可以從任何一個文字編輯器中看到那份運轉記錄。

這個 spy 軟體的重要限制是，它只能夠在某個指定的行程範圍內活動。不像 Win16 的 WinScope 那樣，我的 spy 軟體沒有辦法監視系統中每一個行程的動作。是的，它只能監視單一行程發出的呼叫。

## 攔截函式

任何一個 spy 軟體的基本觀念就是，把自己安插到被監視者的控制流程中。在呼叫動作到達預定目標之前，spy 軟體先一步取得控制權，記錄它所需要的資訊，再把控制權轉給原預定目標。我們的第一個問題就是，如何讓 spy 軟體在適當時機取得控制權，以及如何執行函式呼叫的預定目標。

方法之一是讓你自己的 DLL 開放出與攔截目標完全相同的函式名稱。如果你要攔截 KERNEL32.DLL 的 *GetProcAddress* 函式，你得讓你自己的 DLL 也開放一個

*GetProcAddress*。把這個 DLL 的 import library 放在聯結時的最前端，聯結器就會把對 *GetProcAddress* 的呼叫動作導到你的這個 DLL 來。這個 DLL 做好記錄之後，跳到真正的函式（KERNEL32.DLL 的 *GetProcAddress*）去。如果不想產生一個 import library，你也可以只是在 DEF 檔中 "alias" 那些輸入函式。然而，不論哪一種作法，都要求聯結器在聯結時期就設定 API 攔截動作 -- 如果你要刺探的程式不是你自己寫的，問題就來了。

這種攔截方法正是 SDK 所附之 Microsoft API parameter profiler 所採行的作法。這個 profiler 含有五個 DLLs：ZDI32.DLL、ZDVAPI32.DLL、ZERNEL32.DLL、ZRTDLL.DLL、ZSER32.DLL。這些 DLLs 分別攔截 GDI32.DLL、ADVAPI32.DLL、KERNEL32.DLL、CRTDLL.DLL、USER32.DLL。只不過你的程式不需要和它們聯結，你只要對著欲被監視的程式執行 APF32CVT 就夠了。這和「與上述 DLLs 的 import library 聯結」的效果一樣，但過程之中不需要原始碼。

以我要求的規格來看，這種所謂 dummy DLL 的作法，有兩個問題。第一，不太容易擴充容納其他的 DLLs。每次你需要攔截一個新的 API，你必須修改對應的「DLL 攔截版」並重新製作。你也必須重新聯結你的刺探目標。第二，同時也是比較大的問題是，你必須改變被刺探程式的可執行檔，而這直接違反了我的一個設計原則。

攔截 API 的另一個方法是修改呼叫對象。只要改變被呼叫函式的最初一部份碼，spy 軟體就可以讓自己在該函式執行之前先獲得控制權。兩種方法可以改變程式的前置碼（prologue code），用以轉換控制權。第一同時也是最明顯的作法就是在第一個位元組上放一個中斷點（breakpoint）指令，opcode 爲 0xCC。當此函式被呼叫，spy 軟體所安裝的一個中斷服務常式就會獲得執行權並記錄它所要的資訊。然後 spy 軟體再經由 trap flag single-step 機制，在 CPU 真正執行一個指令之前，把原來的內容寫回第一個位元組中。而在那個 single-step 異常處理常式中，API spy 軟體重新插入中斷點的 opcode，使後續對此函式的呼叫能夠再被捕捉。

雖然某些 Win16 spy 軟體使用中斷點來攔截函式呼叫，但在 Win32 之中這項技術比較困難。Win32 行程看不到另一行程的異常情況（exception），除非它是該行程的除錯器。此外，如果強迫每一個 API 呼叫都必須經過 Win32 結構化異常處理函式，可能會嚴重減弱效率。同時，Win32 行程的分離位址空間使得 spy 軟體必須使用 *ReadProcessMemory* 取刺探目標的函式參數。而這比直接讀記憶體遠遠地沒效率也不方便得多。

改變程式前置碼（prologue code）以便「將控制權轉換到我們的 spy 軟體」的第二個方法是，在函式一開始處插入一個 JMP 或 CALL 指令。這個作法的問題是，32 位元碼的一個 JMP 或 CALL 指令需要至少 5 個位元組。如果 DLL 有個函式小於 5 位元組（是的，有可能），補進一個 JMP 或 CALL 就不可行，因爲稍後而來的程式碼將從 JMP 或 CALL 指令的半途開始（譯註）。中斷點可以被另一個行程處理，但是 JMP 和 CALL 指令的補綴動作卻必須在被刺探程式的行程範疇（process context）內。爲了能夠進入任何行程，你的碼必須成爲一個 DLL。然而，稍後你就會知道，把你的窺視碼安排在被刺探行程中執行，不是什麼好主意。在被刺探行程中補進 JMPs 或 CALLs 是一件很麻煩的事兒，特別是因爲需要在原來的碼和你的 JMP/CALL 指令之間不斷地切換。

> 譯註：很抱歉，因爲我個人對此處所言不太有把握，爲恐失真，我把原文列給你：The second method of modifying a functions's prologue code to transfer control to the spying code is to insert a JMP or CALL instruction at the start of the function. One problem with this approach is that in 32-bit code, a JMP or CALL instruction will take at least 5 bytes. If the DLL has functions less than 5 bytes apart (yes, this has happened!), patching in a JMP or CALL becomes impractical because the function that comes later in the code will start in the middle of a JMP or CALL instruction.

看過並且討論過兩個明顯的攔截方法後（一個是與 custom DLL 聯結，一個是修補 API 函式碼），讓我們看看不是那麼明白的第三個方法。手冊上並沒有說被偵測的目標（API 函式）可以被修補，那麼，修改呼叫端一樣是正當性十足。如果 spy 軟體能夠找出 API 呼叫動作，它就可以修改它們，使它們指向 spy 軟體運轉記錄碼（logging code，一個函式）來。而與前面所討論的一樣，spy 軟體的運轉記錄碼必須在被刺探程式的行程範疇

中執行。稍後的一節「把 DLL 注射到行程之中」，顯示如何注射一個 DLL 進入另一個行程的位址空間。在這裡，我先集中焦點在攔截部份。

或許你會想：一個程式有成百上千個 API 呼叫，我怎麼可能找出所有呼叫？別怕，Win32 的 EXEs 和 DLLs 的動態聯結方式使事情簡單得幾乎不可置信。對同一個 API 的呼叫動作都會導到可執行檔中的某一點，只要修補這一點，使它導向 spy 軟體的運轉記錄碼（logging code），你就可以攔截程式之中對此函式的每一個呼叫動作。

爲了明白這是怎麼回事，讓我們看看三個 *GetVersion* 動作所獲得的真正的碼。下面是個簡單的 C 程式：

```
int main()
{
   GetVersion();
   GetVersion();
   GetVersion();
}
```

根據這個小程式，編譯器產生以下的碼：

```
410052: CALL   0042003C
410057: CALL   0042003C
41005C: CALL   0042003C
...
42003C: JMP    DWORD PTR [00440064]
```

注意，CALL 指令並非直接呼叫 KERNEL32.DLL 中的 *GetVersion*。取而代之的是，每一個呼叫都把控制權轉移到 EXE 檔案中的一個 JMP 指令。那個 JMP 指令參考了某記憶體位置處的一個 DWORD，然後跳至該 DWORD 所指之處。本例的 DWORD 位於位址 00440064 處，而儲存在那個 DWORD 中的是什麼呢？是的，正是 *GetVersion* 的真正位址。所有對 *GetVersion* 的呼叫最終都導至 JMP DWORD PTR[XXXXXXXX] 這「一小段碼」（譯註：所謂的 "thunk"）。每一個被可執行檔輸入（import）的函式，都有對應的這一小段碼。誰產生它呢？在微軟編譯器裡頭，這段 JMP 碼是「將被聯結之 DLL 的 import library」中的碼。在 Borland C++ 裡頭則是由聯結器（TLINK.EXE）產生之。

譯註："thunk" 一詞，一般詞典中查不到。但你可以在 The New Hacker's Directory 中查到它的意義：A piece of coding which provides an address。這與本書第４章不斷用到的 "thunking"、"thunk up"、"thunk down"、"thunking layer" 意義不同，但也有點關聯。前述四個詞的意義都有「包裝後移轉」的意思。"thunk up" 是指 16 位元碼「包裝後移轉」到 32 位元碼，"thunk down" 是指 32 位元碼「包裝後移轉」到 16 位元碼。"thunking layer" 就是轉換層的意思。

隨之而來的問題就是：我們從哪裡找出這個 DWORD 函式位址？另一個問題則是：誰負責它的內容？DWORD 函式位址可以從 import address table（簡稱 IAT）獲得。IAT 通常置於可執行檔的 .idata section 中。可執行檔所聯結的每一個 DLL 都有一個對應的 DWORDs 陣列，內含「來自此 DLL 之輸入函式」的位址。當 Win32 載入器把一個可執行檔帶入記憶體，它會以適當的位址填寫此 DWORD 陣列，如圖 **10-1** 所示。在可執行檔被載入之前，每一個 DWORD 內含的則是一個 ASCII Z-string（譯註：以零值做爲結束標記的字串）的偏移位置，該字串表現出函式名稱（例如 "GetVersion"）。一旦載入器把可執行檔帶入記憶體，它就以函式之實際位址改寫此一 DWORD 陣列。

現在你可以瞭解一個 spy 軟體如何能夠攔截並記錄函式的運作了吧。Spy 軟體只需要在可執行檔的 imports section 中找到 import address table，改寫其內容，使其中的 DWORD 改指向 spy 程式碼即可，根本不必做什麼縫縫補補的工作。也不需要在原來的碼和被 spy 軟體修改過的碼之間不斷地切來切去。可執行檔最終是直接呼叫 spy 碼，所以唯一會加重系統負荷的就是 spy 軟體的運轉記錄碼（一個函式）本身。記錄完畢之後，spy 軟體就跳到原目標（利用 JMP DWORD PTR [XXXXXXXX]）。很簡單，不是嗎？

甚至即使「刺探」不是你的目的，你也可以利用這樣的計謀選擇性地攔截 APIs。舉個例子，你可能想要以你自己的碼取代 DLL 中的一個函式。你很容易根據上述觀念做出一個函式，兩個參數分別是被攔截的 DLL 名稱和函式名稱。回返值則爲一個指標，指向 .idata 中那個內含函式位址的 DWORD。你的程式然後應該改寫這個 DWORD，把準備接手的那個函式的位址填進去。如果你要串連原呼叫函式，記得在改寫 DWORD 之前先儲存其內容（代表原函式位址）。

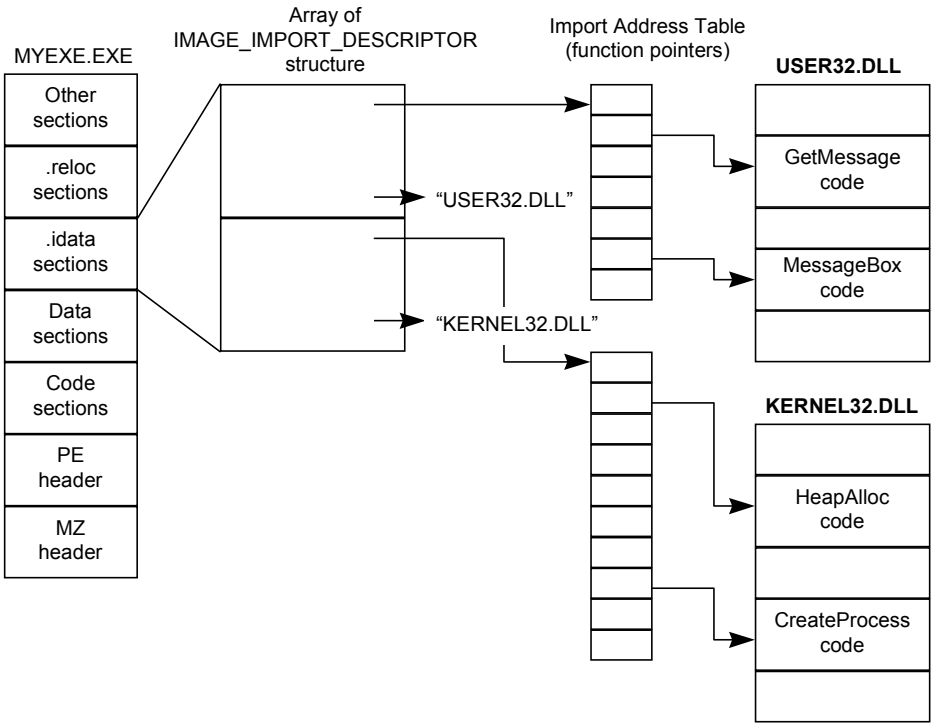



**圖 10-1　可執行檔的 .idata section 通常持有內含「輸入函式之位址」的 DWORD。（其實 imports table 可以被放在任何地方）**

本章建立的這個 spy 軟體，我所選擇的攔截方式就是上面所說的這種。我必須確實讓我的 spy 碼進入被刺探程式的行程範疇中。由於 API spy 軟體的設計目標之一是「不得重新聯結或修改刺探對象」，我必須強迫把 spy 碼放進刺探目標的位址空間中。這也意味著 spy 軟體必須是一個 DLL。

# 把 DLL 注射到行程之內

現在我們知道如何攔截 API 函式了，下一個障礙就是強迫把 spy 碼放進目標程式的位址空間中。在 16 位元 Windows 環境中這根本不是問題，因爲所有程式共享同一位址空間。然而 Win32 程式各有自己的位址空間，各有自己的一組 DLLs。一個行程使用某個 DLL 並不表示另一個行程也可以使用它。每一個行程若要使用 DLLs 都必須自己載入之 -- 不管是隱式聯結（implicitly link），或明白呼叫 *LoadLibrary*。由於我們的刺探對象根本就不知道 spy 軟體的存在，我們需要玩一點「權謀」才有辦法強迫 DLL 進入其位址空間中。

至少有三種方法，可以把 DLL 注射到任意行程中。Jeffrey Richter 在 1994 年五月發表於 ***Microsoft Systems Journal*** 的一篇文章中揭示了每一種方法。這裡我將先對兩種不適用的方法做一次瀏覽，然後花比較多的時間在 spy 軟體比較會考慮採用的第三種方法。Richter 的第三種作法是做爲一般性目的使用，類似（但不完全相同）於我的作法。差別在於 Richter 使用 *CreateRemoteThread*，但這個 API 不存在於 Win32s 和 Windows 95 環境上。我的這個方法則可適用於所有 Win32 平台。

第一個也是最爲人熟知的作法是，利用 *SetWindowsHookEx* 安裝一個 window hook。如果你指定一個 hThread 而它屬於另一行程，或者如果你設定的是系統層面的 hook，作業系統便會自動載入內含 hook 函式之 DLL，放到所有受 hook 影響的行程位址空間中。這種方法很沒有效率，原因有二。第一，你必須有一個原就存在的行程，用以安裝 hook。而如果確是如此，該行程必定也會呼叫 API。那麼 spy 軟體會遺漏所有該行程發出的 API 呼叫。第二個理由是，直到行程有某些動作，使得 hook callback 被喚起，hook DLL 才會被載入行程位址空間中。我的結論是，這種作法無法提供足夠的嚴謹度（足以確定 DLL 被載入的時間）。

迫使 DLL 被載入行程位址空間的第二個作法是，有一個鮮爲人知的 registry key，被埋藏在層層疊疊的 registry 架構中：

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\APPINIT_DLLS

只要在這個 key 後面加上你的 DLL 名稱，作業系統便會自動載入 DLL 到每一個行程的位址空間中 -- 在行程啓動時刻。數個理由使此法也不適用於 spy 軟體。最主要的理由是 registry 的改變必須在下次開機後才能顯現效果。爲了刺探一個程式，你竟然得重新開機。唔，不實際！另一個缺點是，spy DLL 必須視情況決定它是否要刺探這個載入它的行程。對於那些你並不想刺探的程式，spy DLL 的 *DllMain* 應該在反應 DLL_PROCESS_ATTACH 時傳回 0，告訴作業系統說這個 DLL 不應該被此行程載入。另一個問題是，作業系統會嘗試爲每一個行程載入此 spy DLL，甚至是那些隱藏在背景、你絕不可能和它們有互動關係的行程，像是 MPREXE.EXE。這會讓整個系統的速度降下來。

迫使 DLL 被載入行程位址空間的第三個作法相當暴力，但同時也是我所使用的方法。理想情況下，我們應該通知目標行程，要求它呼叫 *LoadLibrary* 載入我們的 spy DLL，而且它應該在啓動之後立刻呼叫 *LoadLibrary*。雖然我們沒辦法直接這麼做（你不能改變被刺探的對象不是嗎），但是沒理由說我們不能夠玩點計謀，讓目標行程載入我們的 spy DLL。

讓我以一個比喻來說明我的想法。假設你要打開一個以「聲音辨識鎖」鎖住的金庫。只有一個人能夠通過聲音辨識，而你不是那個人。那個人也沒有意思要爲你開門。然而，你可以對他施以催眠術，在他恍惚之際，誘他說出那個開鎖關鍵字。而在將他帶離催眠狀態之前，告訴他什麼都沒有發生。

這將如何實施於 DLL 的載入動作？如果我們可以將目標行程冰凍（或說催眠，如果你喜歡），我們就可以修改行程的記憶體和暫存器，使情況看起來像是行程以其自我意志呼叫 *LoadLibrary*。設定好記憶體和暫存器之後，我們將該行程解凍，讓它繼續執行。於是乎，該行程呼叫了 *LoadLibrary*，作業系統被迫載入 spy DLL 至其位址空間。*LoadLibrary* 回返之後，我們再次將目標行程冰凍，將記憶體和暫存器恢復原狀。船過水無痕，目標行程以爲什麼事兒也沒有發生。

正如你所想像，假冒聖旨去呼叫 *LoadLibrary* 是十分複雜的行爲。它必須改變目標行程的碼，所以第一步就是計算出要修改的碼放在哪個 page，並且將該 page 內容儲存起來以便稍後回復。「注射」動作還得修改暫存器值，所以也應該先把所有暫存器值儲存起來。幸運的是 Win32 提供了 *GetThreadContext* 函式，它會把所有暫存器值寫入一個 C 結構中。

接下來我的程式會產生一小段碼，用以在目標行程中呼叫 *LoadLibrary*。該小段碼含有 spy DLL 的名稱 "APISPY32.DLL"，緊接在 *LoadLibrary* 呼叫之後，是一個中斷點（breakpoint）指令，允許載入器（譯註：不是系統的載入器，而是作者另寫的一個，APISPYLD.EXE）在 *LoadLibrary* 執行完畢後立刻取得控制權。此一小段碼製作出來後，我利用 *WriteProcessMemory* 把它寫入目標行程的第一個 page 中。然後我立刻改變目標行程的 EIP 暫存器（譯註），讓執行權從我這一小段碼開始。

> 譯註：EIP 就是 Extended Instruction Pointer，所指之處即是程式進行之處。

設定好記憶體和暫存器（如上）之後，spy 軟體就讓目標行程（被刺探對象）開始執行。如果一切都在計劃之中，目標行程會成功執行起 *LoadLibrary*，然後觸及我所設定的中斷點。中斷點一被觸發，這個行程就暫時被冰凍起來，於是 spy 軟體有機會恢復目標行程的原狀（包括記憶體和暫存器），用的是極爲便利的 *SetThreadContext* 函式。於是每一件事情回歸原位（但我們的 spy DLL 已經安置於目標行程的位址空間中了），而我們的中斷點處理常式想辦法讓目標行程重獲控制權。稍後在「APISPY32 程式碼」一節中我會更詳細地說明整個作法。

## 使用 Debug API 控制另一個行程

要將一個 DLL 載入某個行程之中，必須對該行程的進行有精確的控制。Win32 的 debug API 提供了我們所需的一切基礎資訊。尤其是我們必須精確知道目標行程執行第一個指令的時機，然後才能夠注射我們的 spy DLL。我們也必須靠 debug API 獲知目標行程的結束時間。此外，當我們正在對目標行程的位址空間進行外科手術，我們必須確定這個

行程不會起身下床活動 -- 在我們還沒把線縫好的時候。debug API 可以處理這些問題。被除錯端的所有執行緒都會被凍結，直到除錯器告訴作業系統說「讓它動吧」爲止。

如果我們爲 Win16 程式寫一個 spy 工具，TOOLHELP 的 *NotifyRegister* 和 *InterruptRegister* 兩函式將是一張通往目的地的車票。TOOLHELP 的 NFY_STARTTASK notification 讓我們知道何時是新的 task 將行未行之際。不幸的是 TOOLHELP 的「notification callbacks 模型」係建立在單一位址空間的假設基礎上，在 Windows NT 和 Windows 95 行不通。所以我們必須另謀生路。最接近的解決方案就是 Win32 debug API。

欲使用 Win32 debug API 來監視目標行程的執行，API spy 軟體就必須承擔某種設計架構。API spy 軟體需要兩個元件，一個元件用來攔截 API 函式並做運轉記錄。它必須處於一個 DLL 之中，我們要把它注射到目標行程裡頭。第二個元件是個 loader 程式，把目標行程載入記憶體。載入之後，spy 軟體進入一個所謂的除錯迴路，其中主要的動作就是呼叫 *WaitForDebugEvent* 和 *ContinueDebugEvent*。當 *WaitForDebugEvent* 傳回一個除錯事件，loader 程式檢查它並決定採取什麼行爲。除錯事件的型別可以由 *WaitForDebugEvent* 傳回：

```
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
```

把它們和 Win16 *NotifyRegister* callback 函式所傳回的 notifications 比較，你就會發現它們多麼相似。如果希望程式使用 *WaitForDebugEvent* 並顯示所有可能被傳回的資料，請你參考 Win32 SDK 所附的 DEB 範例程式。

我們的 loader 程式處理完除錯事件之後，呼叫 *ContinueDebugEvent*，告知作業系統說被除錯端可以繼續執行了。把 *WaitForDebugEvent* 和 *ContinueDebugEvent* 安排在一個迴路之中，loader 於是就可以在目標行程處於被剌探狀態的整個生命期中，看到它發生的所有重要事件（events）。

對我們的 spy 軟體而言，最重要的除錯事件就是 EXCEPTION_DEBUG_EVENT。在行程開始執行的前一刻，*WaitForDebugEvent* 會獲得一個 EXCEPTION_DEBUG_EVENT notification，異常（exception）型態則是 STATUS_BREAKPOINT。我們的 loader 程式取得它，當做是「強迫把 spy DLL 注射到目標行程之位址空間中」的一個暗示。當 *LoadLibrary* 回返，回到我們所設的中斷點，loader 程式將收到另一個 STATUS_BREAKPOINT 異常狀態。loader 程式因此知道它何時應該把原先的暫存器值和記憶體內容恢復過來。

當 loader 程式處理完目標行程傳來的兩個中斷點異常後，它的工作幾乎已經完成了。然而，Win32 並沒有提供什麼方法讓除錯器告訴作業系統說它不要再接收除錯訊息。如果你在行程中使用 debug API，每當行程產生一個除錯事件，行程就會被系統凍住。「除錯器對所收到的每一個除錯事件都呼叫 *ContinueDebugEvent*」，是讓被除錯端繼續執行的唯一方法。因著這個原因，API spy loader 程式必須在迴路中不斷地呼叫 *WaitForDebugEvent* 和 *ContinueDebugEvent*，直到目標行程結束爲止。雖然我其實只需要截下某幾個除錯事件，卻被迫全盤接受。所有我不感興趣的除錯事件都將被我忽略掉：直接呼叫 *ContinueDebugEvent* 而不做任何處置。如果以虛擬碼來表示，API spy loader 看起來像這樣：

```
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()
    }
```

## 建立 Stubs 以記錄 API 函式

現在我們已經解決了主要的架構問題：

- API 如何被攔截
- spy DLL 如何被載入到目標行程的位址中
- 如何謹慎而精確地控制目標行程的執行

還有其他一些題目需要處理，但它們並非直接與作業系統有關。例如我們需要一段碼用來處理被重新導向後的函式呼叫。雖然，在 spy DLL 中產生單一一個函式進入點以應付我們導到 spy DLL 的所有函式，似乎是個誘人的想法，但其實單一一個進入點不可能厲害到竟然能夠知道它正在處理哪一個函式。我們必須爲我們所攔截的函式產生一塊專屬的碼。"thunk" 這個字眼常被用來表示「在轉換控制權至其他地方之前，用於某些處理」的一小塊碼（譯註：請看本章稍早前的一個譯註，那是 ***The New Hacker's Directory*** 中對於 thunk 的定義）。雖然我所產生的這小塊碼可以被稱爲 thunks，但是我將使用 "stub" 這個字眼以避免和 Windows 系統所謂的 thunks 混淆。我的 spy 軟體中的所有 code stubs 都很類似，只有輕微不同。當 stub 收到一個被重新導向的函式呼叫，它會在呼叫一個共用的運轉記錄碼（logging code）之前，把該函式所需的資訊壓到堆疊之中。

如果我們需要的只是攔截一組固定的、已知的函式，我們可以輕易產生一些巨集並在編譯時期創造出所有的 code stubs。然而由於我們的規格上明明白白顯示，API spy 軟體應該可以擴充，所以在 spy 軟體的編譯時期創造這些 code stubs 並不適當。我們應該根據組態檔（configuration file）中的記錄，動態產生它們。幸運的是，在 Win32 中這不困難。

對於每一個 stubs，我們可以配置記憶體並寫入適當的內容。在 16 位元 Windows 環境中這很困難，因爲我們必須在 code 節區（而非 data 節區）中配置記憶體。一旦配置成功，我們又不能夠直直地寫入內容，因爲寫入 code 節區是不被允許的。欲寫入 code 節區，我們必須使用 alias selector 或 TOOLHELP 的 *MemoryWrite* 函式。在 Win32 中這不會成爲絆腳石，因爲 code 節區和 data 節區都映射到完全相同的位址範圍上。我們可以使用一般的 flat model data pointer，把我們的碼寫進去，然後執行這些碼。

爲了產生 stubs，spy DLL 讀入一個組態檔（APISPY32.API），內含以下資訊（針對每一個被攔截的函式）：

- 此函式所屬之 DLL 名稱
- 函式名稱
- 函式參數相關資料

spy DLL 針對每一個函式產生一個 stub，內含程式碼和資料，形式如圖 10-2。程式碼部份首先保存所有 general-purpose 32 位元暫存器。這並非絕對必要，但是好的程式風格總是希望我們怎麼來怎麼去。接下來 stub 程式碼將三個指標壓入堆疊之中，然後呼叫運轉記錄碼（logging code）。三個指標分別指向 API 函式名稱、堆疊中的回返位址、以及函式的「參數資訊」。稍後我會說明所謂的「參數資訊」。在運轉記錄碼完成了它的工作之後，stub 碼把所有 general-purpose 暫存器還原，並 JMP 到原本應該被呼叫的函式去。

```
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.
```

**圖 10-2　對於每一個被攔截函式，SPY DLL 建立一個 stub，內含程式碼和資料。**

就像你所預期的，所有 stubs 必須在我們開始重導「API 函式呼叫」之前先行建立好。當 spy DLL 建好一個 stub，它會把 stub 位址加入一個 stub 指標陣列中。將目標行程中的函式呼叫動作重新導向很是簡單。對於目標行程中的每一個輸入函式，spy DLL 首先取其函式位址。這些位址被維護於一個表格（譯註：就是前面提過的 import address table）中，該表格由 PE 表頭的 DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress 指出（譯註）。接下來，spy DLL 檢查它所建立的 stubs 陣列，搜尋是否有哪一個 stub 的第一個 DWORD 內容與剛才所取得的位址相同。如果找到了，spy DLL 就以「stub 的第一個指令的位址」，寫入目標行程的 imports address table 中的適當 DWORD。整個過程顯示於圖 **10-3**。

譯註：事實上 import address table（IAT）並非直接由 PE 表頭的 DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress 指出，而是還要經過一些牽引。我想這是作者的一個小疏忽。稍後出現在圖 **10-7** 中的 *InterceptFunctionsInMoudle* 函式，會明白顯示尋找 IAT 的整個過程。請你詳細推敲。

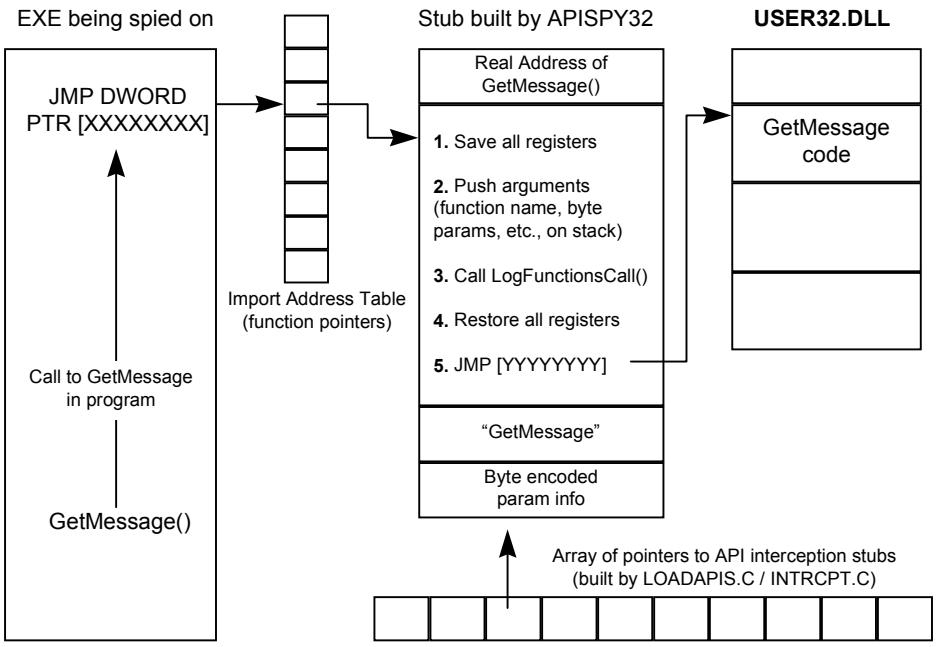



**圖 10-3　imported function 的位址被 stubs 重新導向。這些動態產生的 stubs 喚起 APISPY32 的運轉記錄碼（一個函式），記錄 API 函式的名稱和參數。**

## 「參數資訊」的編碼

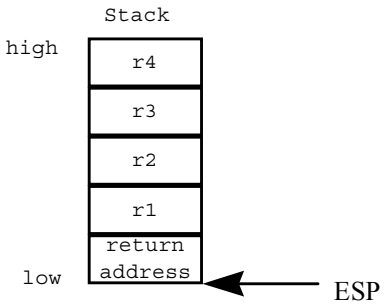
大部份 API spy 軟體都具備了函式參數實際數值的顯示能力。如果 spy 軟體爲每一個被攔截函式準備一份運轉記錄碼（一個函式），那將是十分昂貴的代價。而且，這麼一來萬一要增加新的刺探對象（函式），你必須增加程式碼並重新編譯。

比較好的作法是以壓縮方式表現函式的參數。當然壓縮格式必須被運轉記錄碼看懂。由於 Win32 程式設計中的參數型態個數有限，我決定把每一種基本型態編碼爲一個獨一無二的 BYTE。這些基礎型態包括 BYTEs、WORDs、DWORDs、LPSTRs。爲了簡化，除了 LPSTRs 和 LPWSTRs，我把其他所有指標編碼爲 LPDATA 型態。如果你要讓 spy

軟體更精緻，你可以擴充參數的編碼，使它涵蓋其他型態，包括指向特定資料結構的指標（例如 LPRECT）。運轉記錄碼可以利用這些額外的參數型態資訊，顯示參數的更多細節（像是 LPRECT 中的欄位等等）。然而，就如我說，我的目標是讓事情簡單些，所以在我的碼中只有 10 個參數型態。

譯註：Matt Pietrek 把本章讀者定位得相當高，然而我相信已經不再有太多程式員有這麼低階的實際經驗。進入下一節之前，讓我在這裡加強你的基礎，否則對於 Matt 程式中針對堆疊所玩的五鬼搬運技巧（以下會陸續出來），肯定一頭霧水。

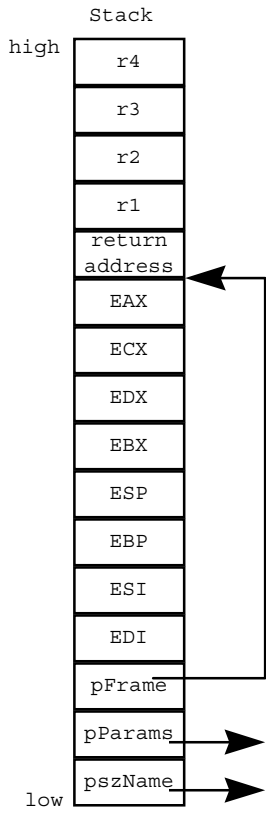
當程式以 C 或 stdcall 呼叫慣例（calling convention）呼叫一個函式，編譯器對於函式參數以及回返位址的處理（堆疊中）如下：




也就是說，進入該函式之前，ESP 正指向「回返位址」那一項目。當 spy DLL 利用 *InterceptFunctionsInModule* 把被刺探程式的 API call 導引到該函式所對應的 stub，經過了以下前五個動作（請參考圖 **10-2**）：

```
pushad
lea     EAX, [ESP+32]
push    EAX
push    [pParams]
push    [pszName]
call    LogCall
```

堆疊變成：




這也就是進入 *LogCall* 瞬間的堆疊狀態。有了這個知識基礎，你在 spy DLL 許多函式中看到堆疊處理動作，就可以輕鬆理解了。另外還有兩點必須記住：

**1.** 堆疊是由高位址往低位址成長。

**2.** 函式傳回值記錄在 EAX 之中。

在我這個 API spy 軟體中，參數資訊被放置在對應之 stub 的尾端。其中第一個位元組放的是參數個數，每一個參數的編碼（BYTE code）緊臨其後，次序一如它們出現在函式宣告時一樣。例如，我們說 HWND 參數以 8 表示，DWORD 參數以 1 表示，於是 *GetWindowWord*(HWND, DWORD) 函式的參數編碼將是 2,8,1（２個參數，第一個參數

是 HWND（8），第二個參數是 DWORD（1））。如果函式沒有參數，就以單一一個 0 表示。

解碼並顯示參數內容並不困難。stub 交給運轉記錄碼（logging code）的資料中，有一個指標指向原堆疊的尾端。原堆疊尾端的 DWORD 是回返位址（當 API 函式完成任務之後，控制權便會回到該處）。原堆疊中的回返位址之上是函式參數（由呼叫端壓入堆疊）。爲了解讀參數資訊，運轉記錄碼從堆疊中取回一個 DWORD（對應一個被編碼的參數），解讀後吐出一個字串，內含參數型態及其實值（例如：LPSTR:00410068）。

讀過一個參數之後，運轉記錄碼將堆疊指標累加 4，指向下一個參數。對 Win32 API 而言，好消息是，參數壓入堆疊的次序是從最後一個到第一個，這使得第一個參數出現在最低位址（譯註：堆疊出口）。如果 Win32 API 使用和 Win16 API 一樣的 pascal 呼叫慣例（參數壓入堆疊的次序是從第一個到最後一個），參數資訊的解碼過程就會比較困難一些，因爲不同的 APIs 的第一個參數將可能出現在堆疊的不同位置。

## 函式傳回值

現在，我們要實際攔截函式並記錄其參數。然而，我們的設計規格說，我們還需要記錄函式的傳回值。這使得事情又更困難了些。雖然對已知某個函式的所有呼叫動作最終都要繞經單一地點（這才使我們得以將之一舉成擒），API 函式卻可能回返到許多不同地點。我們如何從天女散花式的各個地點取得控制權並取出 EAX 暫存器的值（函式傳回值的放置處）？

在我們讓 API 函式真正執行之前，我們所知道有關於「函式回返」的唯一資料就是其回返位址。所以上述問題的最明顯解答便是在該位址設立一個中斷點（breakpoint）。另一個方法是安插一個 JMP，跳到我們的運轉記錄碼去。雖然兩種方法通常都可以有效運作，但它們零亂而不優雅，並且有某些問題（稍早我在敘述如何攔截函式時談過）。一個比較不明顯但比較乾淨（不需要改變被刺探程式的碼）的作法是，改變堆疊中的回返位址，使指向 spy DLL 的「回返值記錄碼」（一個函式）。當然，你必須暫時記住原來

的位址，並且必須在你讓真正的 API 函式執行之前，完成一切準備。於是，API 執行之後便會回返到 spy DLL 的「回返值記錄碼」中。記錄下 API 回返值後，「回返值記錄碼」把原回返位址拷貝回堆疊之中。於是當「回返值記錄碼」結束，控制權又回到了目標行程。

爲恐你以爲上述方法輕輕鬆鬆，請注意其中有個陷阱。不論 Win16 或 Win32，一個 API 函式可能在執行過程中需要呼叫另一個 API 函式。典型的例子就是 *DispatchMessage*。這個 API 函式呼叫你的視窗函式。當你在你的視窗函式中又呼叫 Windows API 函式，你就是「在一個 API 之中又呼叫另一個 API」。有什麼問題嗎？是的，前面我所描述用以攔截回返值的簡單方法中，「回返值記錄碼」中有一個變數持有原來的回返位址。如果你掉進巢狀式 API 呼叫的泥淖，只有最近一次呼叫的回返位址會被儲存起來。較早呼叫的回返位址會遺失不見。

爲了處理巢狀 API 呼叫的情形，我設計了一個堆疊用以存放回返位址。只要 spy 軟體修補了一個函式回返位址，使它指向 spy 軟體的「回返值記錄碼」（一個函式），就把原先的回返位址記錄到這個堆疊之中。當這個「回返值記錄碼」準備回到呼叫端，它會取走「回返位址堆疊」的最上一筆資料，並回到該資料所表示的位址去。我的這個「回返位址堆疊」的作用在某些方面類似真正的程式堆疊，關鍵差異在於「回返位址堆疊」中放的不是參數，而是被 spy DLL 攔截的函式的位址。

巢狀 API 函式呼叫的問題解決之後，我們可以開始寫碼了吧？不，沒那麼快。Windows NT 和 Windows 95 支援多緒多工，每一個執行緒有它自己的堆疊。爲了處理多個執行緒，API spy DLL 爲目標行程的每一個執行緒各自維護一個「回返位址堆疊」。由於 spy 軟體事先並不知道目標行程將啓動多少執行緒，所以「回返位址堆疊」所需之記憶體就動態地在新執行緒產生時才配置。幸運的是，Win32 使我們輕易可以得知執行緒的產生。作業系統會在開啓一個新執行緒時，呼叫所有 DLLs 的進入點（例如 *DllMain*）。下一節（那裡有 spy 原始碼）你就會看到，*DllMain* 在執行緒開啓之時，配置記憶體，做爲每個執行緒專屬堆疊之用。

執行緒專屬之「回返位址堆疊」的指標，被儲存在 Win32 的 Thread Local Storage（TLS）中。TLS 機制允許你爲執行緒儲存一串不同的 DWORDs。第 3 章曾經描述過 TLS 細節。欲使用 TLS，首先你得利用 *TlsAlloc* 配置一個索引，並把此值儲存在一個全域變數中。然後，每一個執行緒都可以以此索引，利用 *TlsGetValue* 取出與執行緒相關的資料。你也可以利用 *SetTlsValue*，指定索引以及欲儲存值，那麼現行執行緒的 TLS 內容就會被你改變。在我的 spy 軟體中，與執行緒相依相隨而必須儲存起來的是此一執行緒的「回返位址堆疊」的指標。當 DLL 初次被載入，它配置 TLS 索引，並在每次 DLL_PROCESS_ATTACH 進入 *DllMain* 時處理之。

有些 Win32 程式員可能知道 __declspec(thread)，這是個編譯器使用的指令。利用 __declspec(thread) 可以很方便產生出執行緒專屬變數，不需要藉 *TlsXXX* 函式之助（請參考第 8 章對於 .tls section 的介紹）。難道把執行緒專屬堆疊做成一個 __declspec(thread) 變數，不會比使用 *TlsXXX* 函式方便嗎？不幸得很，的確如此，因爲 __declspec(thread) 不適用於以 *LoadLibrary* 載入之 DLL（但它可適用於隱式載入之 DLL 中。譯註）。我的 spy DLL 是以 *LoadLibrary* 載入，所以 __declspec(thread) 對我毫無用處。

> 譯註：所謂隱式載入（implicitly loaded），是指程式在聯結時期就聯結了 DLLs 的 import library。所謂顯式載入（explicitly loaded），是指以 *LoadLibrary* 載入 DLLs。

或許你會好奇我的 spy 軟體在 Win32s 中會有什麼表現，因爲 Win32s 並不支援多緒多工。微軟做對了一件事：他們讓 Windows 函式庫包含 TLS 函式。雖然 Win32s 程式的 TLS 資料其實是全域變數，重要的是我們的 spy 軟體不必更改。

如你所見，攔截 API 函式的回返值遠比我們想像中困難得多。我們不僅需要維護一個「回返位址堆疊」，還必須讓目標行程中的每一個執行緒都有一個這樣的堆疊。還有更複雜的嗎？猜猜看！

Win32 API 中極優雅的一個性質就是結構化異常處理（structured exception handling，SEH）。它和 C++ 的異常處理有些類似，但不全然相同（SEH 的細節請看第 3 章）。SEH 帶來的問題是，它可能對我們的 spy 軟體產生巨大的破壞 -- 在「回返位址堆疊」

這一點上。假設你對 *DispatchMessage* 佈置了一個 try/except{}。在 *DispatchMessage* 最終呼叫的那個視窗函式中，你的碼產生了一個異常情況（例如 STATUS_ACCESS_VIOLATION）。處理此一異常情況者，將是你佈置在 *DispatchMessage* 下方的那個 except{}。問題就在於，CPU 是如此效率地直接跳到該處，甚至沒有讓 *DispatchMessage* 回返。spy 軟體的「回返值記錄碼」（一個函式）因而沒有被呼叫到，所以我們無法從我們所設計的「回返位址堆疊」中移除 *DispatchMessage* 的回返位址。如果這樣的事情一再發生，上述堆疊很快就會滿溢（overflow）。

和以往所遭遇的其他問題不同，這一次我不再有精緻而乾淨的解決方法。我在商業化軟體（BoundsChecker 系列產品）中使用了一些零亂的、複雜的、不完全的解決方法，但是我不打算把它們應用到這一章的 spy 軟體上，因爲那會使程式複雜許多。事實上「巢狀 API 呼叫，而且沒有回返」的情況是十分稀少的。直到今天，還沒有一個程式會在 Structured Exception Handling 的潛在性問題上難倒我的這個 spy 軟體 -- 當然我專爲此而設計的測試程式例外。

「回返位址堆疊」的一個良好副作用是，我們可以利用堆疊指標，瞭解 API 函式呼叫的巢狀深度。運轉記錄碼會利用這項優點，以縮排方式記錄「API 又呼叫 APIs」的情況。巢狀程度愈烈，運轉記錄輸出檔中的縮排愈深。當運轉記錄碼要記錄一個函式呼叫動作及其回返值時，它會檢查「回返位址堆疊」的指標，並且賦與輸出文字對等程度之縮排。在運轉記錄檔中，你很容易就可以將「呼叫行」和「回返行」配對：只要尋找「呼叫行」之後的下一個相同縮排程度者就對了。例如，下面輸出中的第一行（呼叫行）和最後一行（回返行）就是一對兒：

```
DispatchMessageA(LPDATA:80B6AE68)
  LoadCursorA(HANDLE:00000000,LPSTR:00007F00)
  LoadCursorA returns: 2CE
  SetCursor()
  SetCursor returns: 2CE
DispatchMessageA returns: 0
```

# APISPY32 程式碼

我們已經看過一大堆理論，現在是討論真正程式碼的時候了。首先我要描述 DLL 中的函式，然後再說明 loader 程式碼。別害怕，沒有一卡車的原始碼等著你。我很高興（也很驚訝）這麼少的碼就完成了全部工作。

我所建構的 API spy 軟體名爲 APISPY32，那也是 spy DLL 的原始碼檔案名稱。APISPY32.C 中的第一個重要部份顯示在圖 **10-4**。

```
#0001  HINSTANCE HInstance;
#0002  BOOL FChicago = FALSE;
#0003
#0004  #if defined(__BORLANDC__)
#0005  #define DllMain DllEntryPoint
#0006  #endif
#0007
#0008  INT WINAPI DllMain
#0009  (
#0010      HANDLE hInst,
#0011      ULONG ul_reason_being_called,
#0012      LPVOID lpReserved
#0013  )
#0014  {
#0015      // OutputDebugString("In APISPY32.C\r\n");
#0016
#0017      switch (ul_reason_being_called)
#0018      {
#0019          case DLL_PROCESS_ATTACH:
#0020              HInstance = hInst;
#0021              FChicago = (BOOL)((GetVersion() & 0xC0000000) == 0xC0000000);
#0022
#0023              if ( InitializeAPISpy32() == FALSE )
#0024                  return 0;
#0025              if ( InitThreadReturnStack() == FALSE )
#0026                  return 0;
#0027              break;
#0028
#0029          case DLL_THREAD_ATTACH:
#0030              if ( InitThreadReturnStack() == FALSE )
#0031                  return 0;
#0032              break;
```

```
#0033
#0034         case DLL_THREAD_DETACH:
#0035             if ( ShutdownThreadReturnStack() == FALSE )
#0036                 return 0;
#0037             break;
#0038
#0039         case DLL_PROCESS_DETACH:
#0040             ShutDownAPISpy32();
#0041
#0042             if ( ShutdownThreadReturnStack() == FALSE )
#0043                 return 0;
#0044             break;
#0045     }
#0046
#0047     return 1;
#0048 }
```

**圖 10-4　APISPY32 中的 DllMain 函式。**

*DllMain* 函式中有一個 switch 指令，用來導引四個重要的 process/thread 事件至適當的處理常式。以下敘述中當我說到事件（event），我說的也就是 *DllMain* 被喚起時的 dwReason 參數內容。

DLL_PROCESS_ATTACH 事件是我們用以「攔截目標行程的所有函式呼叫，並設定其他與函式運轉記錄有關事務」的一個線索。對於行程中的第一個執行緒，作業系統並不以 DLL_THREAD_ATTACH 訊息呼叫 *DllMain* 函式。因此，你必須考慮到，DLL_PROCESS_ATTACH 其實隱含有一個 DLL_THREAD_ATTACH 意義在裡頭。每一個執行緒需要一個專屬的「回返位址堆疊」，所以不論是 DLL_PROCESS_ATTACH 或 DLL_THREAD_ATTACH，處理時都需要呼叫 *InitThreadReturnStack* 以產生我所說的「回返位址堆疊」。這其中還有一個不明顯的假設：這兩個通告訊息都是在新產生的執行緒的 thread context 中產生。

DLL_THREAD_DETACH 處理時呼叫 *ShutDownThreadReturnStack*，釋放堆疊所用的記憶體。DLL_PROCESS_DETACH 處理時則呼叫 *ShutDownAPISpy32*，關閉運轉記錄檔，使作業系統將緩衝區內容全部寫入檔案。我們可以追奔逐北，把目標行程的 imports section 中所有的位址還原，但其實沒有需要花這個力氣。

新執行緒產生時，作業系統並沒有明顯地爲前一個執行緒發出 DLL_THREAD_DETACH。所以 DLL_PROCESS_DETACH 處理時也必須呼叫 *ShutdownThreadReturnStack*，解除最後遺留的回返位址堆疊空間。

APISPY32.C 的其餘部份顯示於**圖 10-5**。*InitializeAPISpy32* 的第一個動作是引發 *LoadAPIConfigFile*，把 .API 檔載入並根據其中的 API 函式和參數相關資料創造出 stubs。稍後我會更詳細介紹 *LoadAPIConfigFile*。

```
#0001  BOOL InitializeAPISpy32(void)
#0002  {
#0003      HMODULE hModExe;
#0004      DWORD moduleBase;
#0005
#0006      if ( LoadAPIConfigFile() == FALSE )
#0007          return FALSE;
#0008
#0009      if ( OpenLogFile() == FALSE )
#0010          return FALSE;
#0011
#0012      hModExe = GetModuleHandle(0);
#0013      if ( !hModExe )
#0014          return FALSE;
#0015
#0016      if ( (GetVersion() & 0xC0000000) == 0x80000000 )    // Win32s???
#0017          moduleBase = GetModuleBaseFromWin32sHMod(hModExe);
#0018      else
#0019          moduleBase = (DWORD)hModExe;
#0020
#0021      if ( !moduleBase )
#0022          return FALSE;
#0023
#0024      return InterceptFunctionsInModule( (HMODULE)moduleBase );
#0025  }
#0026
#0027  BOOL ShutDownAPISpy32(void)
#0028  {
#0029      CloseLogFile();
#0030
#0031      return TRUE;
#0032  }
```

**圖 10-5　APISPY32 中的初始化函式以及停工前的函式。**

建好 API 函式的 stubs 之後，下一步是打開輸出檔。spy DLL 將把函式呼叫以及傳回值等資料寫到該檔。初始化的最後一個動作是呼叫 *InterceptFunctionsInModule*，將目標行程的函式呼叫動作重導至我們剛剛產生的 stubs 身上。*InterceptFunctionsInModule* 必須知道目標行程的載入基底位址，這樣才能夠找出其 imported functions section table。在 Windows NT 和 Windows 95 中，程式的 HMODULE 就是其基底位址。但由於我們的 DLL 並不是 EXE，所以其 HMODULE 不是我們所要的東西。我們必須呼叫 *GetModuleHandle*(0)，這時候 Win32 系統給你 EXE 的 HMODULE，而非呼叫者的 HMODULE。若是在 Win32s 環境下，我們需要一個額外步驟，因爲它的 HMODULE 不等於基底位址。爲了取得 Win32s 程式的基底位址，我寫了一個 *GetModuleBaseFromWin32sHMod* 函式。此函式使用兩個未公開的 Win32s 函式，把一個 Win32s HMODULE 轉換爲基底位址。此函式位於 W32SSUPP.C 檔案中。至於停工動作，遠比初始化動作簡單，只是呼叫 *CloseLogFile*。

**圖 10-6** 所展示的碼負責讀取運轉記錄檔。在讀取一個函式定義之後，它呼叫 *AddAPIFunction*（位於 INTRCPT.C 檔案中）爲函式的 stub 配置記憶體，並將之初始化。APISPY32.API 檔是一個以行爲單位的 ASCII 文字檔，允許每一行前端有空白字元，也允許出現空白行。任何無法辨識的文字行都會被忽略掉，然後下一行文字緊接被處理。

.API 檔的語法十分簡單。對於你打算攔截的每一個函式，請在檔案中加上這麼一行：

```
API:ModuleName:FunctionName
```

例如：

```
API:USER32.DLL:GetMessageA
```

在函式定義之後，你可以緊接著放置參數資訊。一個參數佔一行，例如：

```
API:USER.DLL:GetmessageA
LPDATA
HWND
DWORD
DWORD
```

合法的參數保留字儲存在 ParamEncodings 陣列中，內容如下：

```
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.).
```

爲了讓你一開始就能夠使用 APISPY32，我附了一個名爲 APISPY32.API 的檔案給你，其內定義了 KERNEL32.DLL、USER32.DLL、GDI32.DLL 以及 ADVAPI32.DLL 中的函式和參數資訊。你可以在這個檔內加入其他的函式定義，例如 COMCTL32.DLL 的函式。你也可以擴充 *LoadAPIConfigFile* 的能力，讓它能夠讀入多個 .API 檔。

我所定義的參數型態中有一些重複。例如一個 HWND 可以被編碼爲一個 HANDLE，也可以是一個 DWORD。我定義這一組保留字的目的是爲了表現最常遭遇的型態。我希望在參數的輸出顯示上，允許一些彈性。有了 LPSTR 和 LPDATA 兩種參數型態，我們就可以在遭遇一個 LPSTR 時顯現真正字串的一小片斷。如果我們以 LPDATA 總括所有的指標型態（包括 LPSTR），就沒有辦法分辨出哪一個是字串型態並顯示其部份內容。另一個可能性（但我沒有實作出來）就是以 TRUE 和 FALSE 顯示 BOOL 參數，而不是顯示其真實數值。還有一個可能是面對一個 HWND 參數，顯示部份的視窗標題內容。這可使我們在觀看運轉記錄檔時，對於 HWND 和某個視窗的聯想力更強。

如果你雄心勃勃，那麼就放手擴充我所定義的參數型態吧。增加一個新型態很簡單，PARMTYPE.H 檔中有一份列舉資料，名爲 PARAMTYPE。把你的新型態加到 PARAMTYPE 的尾端，然後在 .API 檔中加上你的參數名稱，在 ParamEncodings 陣列中加上你的列舉常數。最後，在 LOG.C 的運轉記錄碼中，加上你想要對這個新參數的輸出動作。如果你定義一個 LPMSG，運轉記錄碼就可以解釋其指標所指之內容，把 MSG 結構的欄位填寫到運轉記錄檔。**圖 10-6** 顯示 LOADAPI.C 的開始部份，有關於 .API 檔的解析動作。

```
#0001  BOOL IsNewAPILine(PSTR pszInputLine);
#0002  BOOL ParseNewAPILine(PSTR pszInput, PSTR pszDLLName, PSTR pszAPIName);
#0003  PARAMTYPE GetParameterEncoding(PSTR pszParam);
#0004  PSTR SkipWhitespace(PSTR pszInputLine);
#0005
#0006  extern HINSTANCE HInstance;
#0007
#0008  BOOL LoadAPIConfigFile(void)
#0009  {
#0010      FILE *pFile;
#0011      char szInput[256];
#0012      BYTE params[33];
#0013      BOOL fBuilding = FALSE;
#0014      char szAPIFunctionFile[MAX_PATH];
#0015      PSTR p;
#0016
#0017      // Create a string with the path to the API function file.  This
#0018      // file will be in the same directory as this DLL
#0019      GetModuleFileName(HInstance, szAPIFunctionFile, sizeof(szAPIFunctionFile));
#0020      p = strrchr(szAPIFunctionFile, '\\')+1;
#0021      strcpy(p, "APISPY32.API");
#0022
#0023      pFile = fopen(szAPIFunctionFile, "rt");
#0024      if ( !pFile )
#0025          return FALSE;
#0026
#0027      //
#0028      // Format of a line is moduleName:APIName
#0029      // (e.g., "KERNEL32.DLL:LoadLibraryA")
#0030      //
#0031      while ( fgets(szInput, sizeof(szInput), pFile) )
#0032      {
#0033          PSTR pszNewline, pszInput;
#0034          char szAPIName[128], szDLLName[128];
#0035
#0036          pszInput = SkipWhitespace(szInput);
#0037
#0038          if ( *pszInput == '\n' )    // Go to next line if this line is blank
#0039              continue;
#0040
#0041          pszNewline = strrchr(pszInput, '\n');   // Look for the newline
#0042          if ( pszNewline )
#0043              *pszNewline = 0;                // Hack off the newline
#0044
#0045          if ( IsNewAPILine(pszInput) )
#0046          {
```

```
#0047            // Dispense with the old one we've been building
#0048            if ( fBuilding )
#0049               AddAPIFunction(szDLLName, szAPIName, params);
#0050
#0051            if ( ParseNewAPILine(pszInput, szDLLName, szAPIName) )
#0052               fBuilding = TRUE;
#0053            else
#0054               fBuilding = FALSE;
#0055
#0056            params[0] = 0;  // New set of parameters
#0057         }
#0058         else    // A parameter line
#0059         {
#0060            BYTE param = (BYTE)GetParameterEncoding(pszInput);
#0061            if ( param != PARAM_NONE )
#0062            {
#0063               params[ params[0] +1 ] = param; // Add param to end of list
#0064               params[0]++;                 // Update the param count
#0065            }
#0066            else
#0067            {
#0068               if ( (*pszInput != 0) && (stricmp(pszInput, "VOID") != 0) )
#0069               {
#0070                  char errBuff[256];
#0071                  wsprintf(errBuff, "Unknown param %s in %s\r\n",
#0072                        pszInput, szAPIName);
#0073                  OutputDebugString(errBuff);
#0074               }
#0075            }
#0076         }
#0077     }
#0078
#0079     fclose( pFile );
#0080
#0081     return TRUE;
#0082  }
#0083
#0084
#0085  // Returns TRUE if this line is the start of an API definition.  It assumes
#0086  // that any whitespace has already been skipped over.
#0087  BOOL IsNewAPILine(PSTR pszInputLine)
#0088  {
#0089     return 0 == strnicmp(pszInputLine, "API:", 4);
#0090  }
#0091
#0092  // Break apart a function definition line into a module name and a function
```

```
#0093  // name.  Returns those strings in the passed PSTR buffers.
#0094  BOOL ParseNewAPILine(PSTR pszInput, PSTR pszDLLName, PSTR pszAPIName)
#0095  {
#0096      PSTR pszColonSeparator;
#0097
#0098      pszDLLName[0] = pszAPIName[0] = 0;
#0099
#0100      pszInput += 4;  // Skip over "API:"
#0101
#0102      pszColonSeparator = strchr(pszInput, ':');
#0103      if ( !pszColonSeparator )
#0104          return FALSE;
#0105
#0106      *pszColonSeparator++ = 0;   // Null terminate module name, bump up
#0107                                  // pointer to API name
#0108
#0109      strcpy(pszDLLName, pszInput);
#0110      strcpy(pszAPIName, pszColonSeparator);
#0111
#0112      return TRUE;
#0113  }
#0114
#0115
#0116  typedef struct tagPARAM_ENCODING
#0117  {
#0118      PSTR       pszName;    // Parameter name as it appears in APISPY32.API
#0119      PARAMTYPE   value;      // Associated PARAM_xxx enum from PARMTYPE.H
#0120  } PARAM_ENCODING, * PPARAM_ENCODING;
#0121
#0122  PARAM_ENCODING ParamEncodings[] =
#0123  {
#0124  {"DWORD", PARAM_DWORD},
#0125  {"WORD", PARAM_WORD},
#0126  {"BYTE", PARAM_BYTE},
#0127  {"LPSTR", PARAM_LPSTR},
#0128  {"LPWSTR", PARAM_LPWSTR},
#0129  {"LPDATA", PARAM_LPDATA},
#0130  {"HANDLE", PARAM_HANDLE},
#0131  {"HWND", PARAM_HWND},
#0132  {"BOOL", PARAM_BOOL},
#0133  {"LPCODE", PARAM_LPCODE},
#0134  };
#0135
#0136  // Given a line that's possibly a parameter line, returns the PARAM_xxx
#0137  // encoding for that parameter type.  Lines that don't match any of the
#0138  // strings in the ParamEncodings cause the function to return PARAM_NONE.
```

```
#0139  PARAMTYPE GetParameterEncoding(PSTR pszParam)
#0140  {
#0141      unsigned i;
#0142      PPARAM_ENCODING pParamEncoding = ParamEncodings;
#0143
#0144      for ( i=0; i < (sizeof(ParamEncodings)/sizeof(PARAM_ENCODING)); i++ )
#0145      {
#0146          if ( stricmp(pParamEncoding->pszName, pszParam) == 0 )
#0147              return pParamEncoding->value;
#0148
#0149          pParamEncoding++;
#0150      }
#0151
#0152      return PARAM_NONE;
#0153  }
#0154
#0155  // Given a pointer to an ASCIIZ string, return a pointer to the first
#0156  // non-whitespace character in the line.
#0157  PSTR SkipWhitespace(PSTR pszInputLine)
#0158  {
#0159      while ( *pszInputLine && isspace(*pszInputLine) )
#0160          pszInputLine++;
#0161      return pszInputLine;
#0162  }
```

**圖 10-6　LOADAPI.C 之中有關於 .API 檔的解析動作。**

譯註：作者曾經在**圖 10-5** 之前說過，要對 *LoadAPIConfigFile* 多做點說明。結果並沒有。由於作者使用 fBuilding 做流程控制，其間有點蹊蹺，所以讓我做個補充。這個函式開檔之後，以一個 while 迴路讀.API 中的每一行。清理其中的空白字元之後，首先呼叫 *IsNewAPILine* 檢查是否帶有 "API:"。如果有，則呼叫 *ParseNewAPILine*，獲得 szDLLName[] 和 szAPIName[]，分別代表 DLL 名稱和 API 函式名稱。注意，因爲 fBuilding 原先爲 FALSE，所以會跳過 *AddAPIFunction* 不執行。接下來再讀入 .API 檔的下一行，如果未帶有 "API:"，表示是參數行，於是呼叫 *GetParameterEncoding*，填裝 params[]。然後再讀入 .API 檔的下一行，如果仍然未帶有 "API:"，表示又是參數行，於是再呼叫 *GetParameterEncoding* 填裝 params[]。最後，params[] 可能會成爲像 "281" 或是 "18" 之類的字串。再次讀入 .API 檔的下一行，發現它帶有 "API:"，表示是一個新的「API 函式行」，這時候 fBuilding 的值是 TRUE，所以先進入 *AddAPIFunction*，做出 stub。*AddAPIFunction* 馬上就要出場了。

所有與「攔截函式呼叫」有關的碼都放在 INTRCPT.C 裡頭。第一個函式是 *AddAPIFunction*，它獲得 API 名稱、API 所屬之 DLL 名稱、API 參數編碼等三個參數。它的兩件工作就是創造出對應的 stub 並且把 stub 加到 stubs 串列之中。*AddAPIFunction* 將建構 stub 的細節工作委派給 *BuildAPIStub* 去做。

*BuildAPIStub* 首先根據函式名稱和模組名稱呼叫 *GetProcAddress*，取得函式位址。成功之後，*BuildAPIStub* 計算需要多大的記憶體給 stub 使用（因爲 API 名稱和參數編碼的大小是可變的），然後配置記憶體。接下來 *BuildAPIStub* 填寫 stub 的欄位。其初始內容定義在 INTRCPT2.H 的 APIFunction 結構中。在 stub 的記憶體尾端，*BuildAPIStub* 把函式名稱和參數編碼拷貝進去。

> 譯註：*BuildAPIStub* 函式中對於 stub 欄位的填寫，是根據圖 **10-2** 的 stub 內容量身定做。這一部份我想是這個 spy 程式困難度最高也最容易出錯的地方，因爲實在太低階，曾經涉足於此層面的程式員恐怕已經寥寥無幾。

除了產生並設定 stubs 之外，INTRCPT.C 的另一重要工作就是在目標行程的記憶體映象（memory image）中到處翻尋並重導其 JMP DWORD PTR[xxxxxxxx] ，使它改呼叫我們所建立的 stubs 身上。*InterceptFunctionsInModule* 只需要「模組基底位址」一個參數，用以找出 import address table（IAT，這東西我曾在前一節描述過）。此函式首先驗證傳進來的參數是個合法的模組基底位址，作法是尋找檔案中的 DOS MZ 和 Win32 PE 記號。一旦確定合法身份，它就使用 IMAGE_NT_HEADERS 結構尾端的 data directory，取得一個指向 .idata section 的指標。

爲了找出 EXE 的所有輸入函式，*InterceptFunctionsInModule* 一一造訪 .idata 中的 IMAGE_IMPORT_DESCRIPTOR 結構。第 8 章的 PEDUMP 也有類似的動作。EXE 隱式聯結的每一個 DLLs 都有一個對應的 IMAGE_IMPORT_DESCRIPTOR 結構，其中內含一個 IMAGE_THUNK_DATA 結構的相對偏移位置。每一個 IMAGE_THUNK_DATA 結構對應一個輸入函式。

利用巢狀的兩個迴路，*InterceptFunctionsInModule* 找出每一個輸入函式並取出其函式位址。函式位址其實是保存在 IMAGE_THUNK_DATA 之內。針對每一個輸入函式，我將其函式位址交給 *LookupInterceptedAPI*。後者掃描我們所建立的每一個 stubs，比對是否哪一個 stub 有相同的函式位址。如果比對成功，*InterceptFunctionsInModule* 便以「剛剛找到的這個 stub 內的程式碼的位址」覆蓋掉 IMPORT_THUNK_DATA 中的函式位址（如圖 **10-7** 所示，利用 *WriteProcessMemory*）。從此以後，任何被刺探之程式如果呼叫該輸入函式，便會被導到我們的 stub 內的程式碼來。運轉記錄完成之後，我們的 stub 才會把控制權交給原輸入函式的手上。

```
#0001  PAPIFunction BuildAPIStub(PSTR pszModule, PSTR pszFuncName, PBYTE params);
#0002
#0003  // MakePtr is a macro that allows you to easily add to values (including
#0004  // pointers) together without dealing with C's pointer arithmetic.  It
#0005  // essentially treats the last two parameters as DWORDs.  The first
#0006  // parameter is used to typecast the result to the appropriate pointer type.
#0007  #define MakePtr( cast, ptr, addValue ) (cast)( (DWORD)(ptr)+(DWORD)(addValue))
#0008
#0009  #define MAX_INTERCEPTED_APIS 2048
#0010  unsigned InterceptedAPICount = 0;
#0011  PAPIFunction InterceptedAPIArray[MAX_INTERCEPTED_APIS];
#0012
#0013  extern BOOL FChicago;
#0014  extern FILE * PLogFile;
#0015
#0016  BOOL AddAPIFunction
#0017  (
#0018      PSTR pszModule,     // exporting DLL name
#0019      PSTR pszFuncName,   // exported function name
#0020      PBYTE params       // opcode encoded parameters of exported function
#0021  )
#0022  {
#0023      PAPIFunction pNewFunction;
#0024
#0025      if ( InterceptedAPICount >= MAX_INTERCEPTED_APIS )
#0026          return FALSE;
#0027
#0028      pNewFunction = BuildAPIStub(pszModule, pszFuncName, params);
#0029      if ( !pNewFunction )
#0030          return FALSE;
#0031
```

```
#0032     InterceptedAPIArray[ InterceptedAPICount++ ] = pNewFunction;
#0033
#0034     return TRUE;
#0035  }
#0036
#0037
#0038  PAPIFunction BuildAPIStub(PSTR pszModule, PSTR pszFuncName, PBYTE params)
#0039  {
#0040     UINT allocSize;
#0041     PAPIFunction pNewFunction;
#0042     PVOID realProcAddress;
#0043     UINT cbFuncName;
#0044     HMODULE hModule;
#0045
#0046     hModule = GetModuleHandle(pszModule);
#0047     if ( !hModule )
#0048        return 0;
#0049
#0050     realProcAddress = GetProcAddress( hModule, pszFuncName );
#0051     if ( !realProcAddress )
#0052        return 0;
#0053
#0054     cbFuncName = strlen(pszFuncName);
#0055     allocSize = sizeof(APIFunction) + cbFuncName +1 + *params + 1;
#0056
#0057     pNewFunction = malloc(allocSize);
#0058     if ( !pNewFunction )
#0059        return 0;
#0060
#0061     pNewFunction->RealProcAddress = realProcAddress;
#0062     pNewFunction->instr_pushad = 0x60;
#0063     pNewFunction->instr_lea_eax_esp_plus_32 = 0x2024448D;
#0064     pNewFunction->instr_push_eax = 0x50;
#0065     pNewFunction->instr_push_offset_params = 0x68;
#0066     pNewFunction->offset_params = (DWORD)(pNewFunction + 1) + cbFuncName + 1;
#0067     pNewFunction->instr_push_offset_funcName = 0x68;
#0068     pNewFunction->offset_funcName = (DWORD)(pNewFunction + 1);
#0069     pNewFunction->instr_call_LogFunction = 0xE8;
#0070     pNewFunction->offset_LogFunction
#0071        =  (DWORD)LogCall - (DWORD)&pNewFunction->instr_popad;
#0072     pNewFunction->instr_popad = 0x61;
#0073     pNewFunction->instr_jmp_dword_ptr_RealProcAddress = 0x25FF;
#0074     pNewFunction->offset_dword_ptr_RealProcAddrss = (DWORD)pNewFunction;
#0075
#0076     strcpy( (PSTR)pNewFunction->offset_funcName, pszFuncName );
#0077     memcpy( (PVOID)pNewFunction->offset_params, params, *params+1 );
```

```
#0078
#0079     return pNewFunction;
#0080  }
#0081
#0082
#0083  PAPIFunction LookupInterceptedAPI( PVOID address )
#0084  {
#0085     unsigned i;
#0086     PVOID stubAddress;
#0087
#0088     for ( i=0; i < InterceptedAPICount; i++ )
#0089     {
#0090        if ( InterceptedAPIArray[i]->RealProcAddress == address )
#0091           return InterceptedAPIArray[i];
#0092     }
#0093
#0094     // If it's Chicago, and the app is being debugged (as this app is)
#0095     // the loader doesn't fix up the calls to point directly at the
#0096     // DLL's entry point.  Instead, the address in the .idata section
#0097     // points to a PUSH xxxxxxxx / JMP yyyyyyyy stub.  The address in
#0098     // xxxxxxxx points to another stub: PUSH aaaaaaaa / JMP bbbbbbbb.
#0099     // The address in aaaaaaaa is the real address of the function in the
#0100     // DLL.  This ugly code verifies we're looking at this stub setup,
#0101     // and if so, grabs the real DLL entry point, and scans through
#0102     // the InterceptedAPIArray list of addresses again.
#0103     // ***WARNING*** ***WARNING*** ***WARNING*** ***WARNING***
#0104     // This code is subject to change, and is current only as of 9/94.
#0105
#0106     if ( FChicago )
#0107     {
#0108
#0109        if ( address < (PVOID)0x80000000 )  // Only shared, system DLLs
#0110           return 0;                    // have stubs
#0111
#0112        if ( IsBadReadPtr(address, 9) || (*(PBYTE)address != 0x68)
#0113             || (*((PBYTE)address+5) != 0xE9) )
#0114           return 0;
#0115
#0116        stubAddress = (PVOID) *(PDWORD)((PBYTE)address+1);
#0117
#0118        for ( i=0; i < InterceptedAPICount; i++ )
#0119        {
#0120           PVOID lunacy;
#0121
#0122           if ( InterceptedAPIArray[i]->RealProcAddress == stubAddress )
#0123              return InterceptedAPIArray[i];
```

```
#0124
#0125           lunacy = InterceptedAPIArray[i]->RealProcAddress;
#0126
#0127           if ( !IsBadReadPtr(lunacy, 9) && (*(PBYTE)lunacy == 0x68)
#0128               && (*((PBYTE)lunacy+5) == 0xE9) )
#0129           {
#0130               lunacy = (PVOID)*(PDWORD)((PBYTE)lunacy+1);
#0131               if ( lunacy == stubAddress )
#0132                   return InterceptedAPIArray[i];
#0133           }
#0134       }
#0135   }
#0136
#0137   return 0;
#0138 }
#0139
#0140 BOOL InterceptFunctionsInModule(PVOID baseAddress)
#0141 {
#0142   PIMAGE_DOS_HEADER pDOSHeader = (PIMAGE_DOS_HEADER)baseAddress;
#0143   PIMAGE_NT_HEADERS pNTHeader;
#0144   PIMAGE_IMPORT_DESCRIPTOR pImportDesc;
#0145
#0146   if ( IsBadReadPtr(baseAddress, sizeof(PIMAGE_NT_HEADERS)) )
#0147       return FALSE;
#0148
#0149   if ( pDOSHeader->e_magic != IMAGE_DOS_SIGNATURE )
#0150       return FALSE;
#0151
#0152   pNTHeader = MakePtr(PIMAGE_NT_HEADERS, pDOSHeader, pDOSHeader->e_lfanew);
#0153   if ( pNTHeader->Signature != IMAGE_NT_SIGNATURE )
#0154       return FALSE;
#0155
#0156   pImportDesc = MakePtr(PIMAGE_IMPORT_DESCRIPTOR, baseAddress,
#0157                         pNTHeader->OptionalHeader.
#0158                         DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].
#0159                         VirtualAddress);
#0160
#0161   // Bail out if the RVA of the imports section is 0 (it doesn't exist)
#0162   if ( pImportDesc == (PIMAGE_IMPORT_DESCRIPTOR)pNTHeader )
#0163       return FALSE;
#0164
#0165   while ( pImportDesc->Name )
#0166   {
#0167       PIMAGE_THUNK_DATA pThunk;
#0168
#0169       pThunk = MakePtr(PIMAGE_THUNK_DATA,
```

```
#0170                       baseAddress, pImportDesc->FirstThunk);
#0171
#0172         while ( pThunk->u1.Function )
#0173         {
#0174             PAPIFunction pInterceptedFunction;
#0175
#0176             pInterceptedFunction = LookupInterceptedAPI(pThunk->u1.Function);
#0177
#0178             if ( pInterceptedFunction )
#0179             {
#0180                 DWORD cBytesMoved;
#0181                 DWORD src = (DWORD)&pInterceptedFunction->instr_pushad;
#0182
#0183                 // Bash the import thunk.  We have to use WriteProcessMemory,
#0184                 // since the import table may be in a code section (courtesy
#0185                 // of the NT 3.51 team!)
#0186
#0187                 WriteProcessMemory( GetCurrentProcess(),
#0188                             &pThunk->u1.Function,
#0189                             &src, sizeof(DWORD), &cBytesMoved );
#0190             }
#0191
#0192             pThunk++;
#0193         }
#0194
#0195         pImportDesc++;
#0196     }
#0197
#0198     return TRUE;
#0199 }
```

**圖 10-7　INTRCPT.C 中的 stub 建立以及函式攔截。**

將「函式呼叫」和「參數資訊」寫入檔案的所有相關動作都包含在 LOG.C 檔案裡頭。它的第一個函式 *OpenLogFile* 將輸出檔打開（指定在被刺探程式的同一磁碟目錄中）。輸出檔的名稱和目標行程之執行檔名稱相同，但副檔名是 .OUT。以目標行程的 HMODULE 爲參數，呼叫 *GetModuleFileName*，可以輕易獲得程式的目錄和檔名。所以我們唯一要做的就是把 .EXE 副檔名改爲 .OUT。*OpenLogFile* 對輸出檔並不使用唯讀屬性，所以如果你刺探的是從光碟片中載入的程式，*fopen* 會失敗，而你也因此不能獲得一個運轉記錄檔。

*LogCall* 是一個高階函式，它的責任是把一個函式呼叫動作的相關資訊加進輸出檔中。*LogCall* 由「被攔截函式的 stub 碼」呼叫之，期待獲得一個函式名稱指標，一個參數編碼指標，以及一個指向原堆疊（進入 *LogCall* 之前）的指標。*LogCall* 的第一件事就是把 API 函式參數的解碼動作以及格式化動作的沉悶工作交給 *DecodeParamsToString* 函式。

然後，*LogCall* 對輸出檔吐出一行文字，內含被攔截之 API 函式的名稱，及其解碼後的參數（放在小括號中）。如果這個 API 函式是被另一個 API 函式巢狀呼叫 ，這一行文字將內縮，內縮空格多寡則視巢狀呼叫的層級而定。*LogCall* 的最後一個動作是呼叫 *InterceptFunctionReturn*。這個函式（在 RETURN.C 中）修補被攔截之 API 函式的回返位址，使它指向 spy 軟體的回返值記錄碼（logging code）。

*DecodeParamsToString* 函式接受三個參數。一個是指向「函式參數編碼」的指標，一個是指向「原堆疊」的指標，另一個是指向緩衝區的指標。這個函式首先將原堆疊指標移高四個位元組，以便跳過被攔截函式的回返位址。然後以一個 for 迴路檢查每一個被編碼後的參數，並從堆疊中抓取相關的 DWORD（譯註：也就是真正的參數值），並將它們格式化。參數的文字輸出格式是 <type>:<value>，例如 HWND:000200AC。如果參數型態是 LPSTR，這個函式又呼叫 *GetLPSTR* 取得字串內容（最多 10 個字元），輸出型式如下：

```
LPSTR:80B70018:"FreeCellIc"
```

一旦 *DecodeParamsToString* 將每一個參數格式化，它就把字串添加到緩衝區的尾部。如果有一個以上的參數，就以逗號（,）分隔它們，好像真實的 C/C++ 碼一樣。我們的目標是儘可能做出逼真的東西，像這樣：

```
LoadAcceleratorsA(HANDLE:00001C9E, LPSTR:80B70004:"FreeMenu")
```

記錄函式參數之外，還要記錄函式回返值，那由 *LogReturn* 負責。*LogReturn* 比 *LogCall* 簡單得多，並且也根據巢狀層級縮排輸出文字。

你們之中某些人可能會注意到，LOG.C 完全缺乏執行緒的同步化控制動作。一般而言在一個有檔案 I/O 動作的多緒程式中，你需要索求 critical section 或 mutexes，以便執行緒在非適當時間被切換時，阻止問題發生。LOG.C 不需要任何執行緒同步化控制，因爲它不會更改任何全域變數（PLogFile 指標和 TlsIndex 變數都不會在程式執行過程中改變）。但 *fprintf* 怎麼說？如果執行緒在 *fprintf* 未完成時切換，會有問題嗎？粗想之下你的答案大概是 yes。然而 APISPY32 DLL 聯結時使用多緒版本的 runtime library（Visual C++ 的 LIBCMT.LIB）。這些多緒版函式庫內部已經使用了同步化控制機制，所以我們不需多此一舉。有趣的是，即使你看過整個 APISPY32 碼，你也找不到任何同步化控制碼。因爲全域變數只在初始化階段被寫值進去，此後再也不會改變。**圖 10-8** 展示 LOG.C，它把 API 函式名稱和參數寫到輸出檔去。

```
#0001  // Helper function prototypes
#0002  void MakeIndentString(PSTR buffer, UINT level);
#0003  void DecodeParamsToString(PBYTE pParams, PDWORD pFrame, PSTR pszParams);
#0004  BOOL GetLPSTR( PSTR ptr, PSTR buffer );
#0005
#0006  FILE *PLogFile = 0;
#0007  extern DWORD TlsIndex;         // defined in RETURN.C
#0008
#0009  BOOL OpenLogFile(void)
#0010  {
#0011      char szFilename[MAX_PATH];
#0012      PSTR pszExtension;
#0013
#0014      GetModuleFileName( GetModuleHandle(0), szFilename, sizeof(szFilename) );
#0015
#0016      pszExtension = strrchr(szFilename, '.');
#0017      if ( !pszExtension )
#0018          return FALSE;
#0019
#0020      strcpy(pszExtension, ".out");
#0021
#0022      PLogFile = fopen(szFilename, "wt");
#0023
#0024      return (BOOL)PLogFile;
#0025  }
#0026
#0027
#0028  BOOL CloseLogFile(void)
```

```
#0029  {
#0030      if ( PLogFile )
#0031          fclose( PLogFile );
#0032      return TRUE;
#0033  }
#0034
#0035
#0036  void __stdcall LogCall(PSTR pszName, PBYTE pParams, PDWORD pFrame)
#0037  {
#0038      char szParams[512];
#0039      char szIndent[128];
#0040      PPER_THREAD_DATA pStack;
#0041
#0042      if ( !PLogFile )
#0043          return;
#0044
#0045      DecodeParamsToString(pParams, pFrame, szParams);
#0046
#0047      pStack = (PPER_THREAD_DATA)TlsGetValue(TlsIndex);
#0048      if ( !pStack )
#0049          return;
#0050
#0051      MakeIndentString(szIndent, pStack->FunctionStackPtr);
#0052
#0053      fprintf(PLogFile, "%s%s(%s)\n", szIndent, pszName, szParams);
#0054      fflush(PLogFile);
#0055
#0056      // Patch the return address of this function so that returns to us
#0057      InterceptFunctionReturn(pszName, pFrame);
#0058  }
#0059
#0060  void DecodeParamsToString(PBYTE pParams, PDWORD pFrame, PSTR pszParams)
#0061  {
#0062      unsigned i;
#0063      unsigned paramCount;
#0064      unsigned paramShowSize;
#0065      PSTR pszParamName;
#0066
#0067      pszParams[0] = 0;   // Null out string in case there's no parameters
#0068
#0069      paramCount = *pParams++;    // Get number of parameters and advance
#0070                                  // to first encoded param
#0071      pFrame++;                   // Bump past the DWORD return address
#0072
#0073      for ( i=0; i < paramCount; i++ )
#0074      {
```

```
#0075         switch ( *pParams )
#0076         {
#0077            case PARAM_DWORD:
#0078               pszParamName = "DWORD"; paramShowSize = 4; break;
#0079            case PARAM_WORD:
#0080               pszParamName = "WORD"; paramShowSize = 2; break;
#0081            case PARAM_BYTE:
#0082               pszParamName = "BYTE"; paramShowSize = 1; break;
#0083            case PARAM_LPSTR:
#0084               pszParamName = "LPSTR"; paramShowSize = 4; break;
#0085            case PARAM_LPWSTR:
#0086               pszParamName = "LPWSTR"; paramShowSize = 4; break;
#0087            case PARAM_LPDATA:
#0088               pszParamName = "LPDATA"; paramShowSize = 4; break;
#0089            case PARAM_HANDLE:
#0090               pszParamName = "HANDLE"; paramShowSize = 4; break;
#0091            case PARAM_HWND:
#0092               pszParamName = "HWND"; paramShowSize = 4; break;
#0093            case PARAM_BOOL:
#0094               pszParamName = "BOOL"; paramShowSize = 4; break;
#0095            case PARAM_LPCODE:
#0096               pszParamName = "LPCODE"; paramShowSize = 4; break;
#0097            default:
#0098               pszParamName = "<unknown>"; paramShowSize = 0;
#0099         }
#0100
#0101         pszParams += wsprintf(pszParams, "%s:", pszParamName);
#0102
#0103         switch ( paramShowSize )
#0104         {
#0105            case 4: pszParamName = "%08X"; break;
#0106            case 2: pszParamName = "%04X"; break;
#0107            case 1: pszParamName = "%02X"; break;
#0108         }
#0109
#0110         pszParams += wsprintf(pszParams, pszParamName, *pFrame) ;
#0111
#0112         // Tack on the string literal value if it's a PARAM_LPSTR
#0113         if ( *pParams == PARAM_LPSTR )
#0114         {
#0115            char buffer[30];
#0116
#0117            if ( GetLPSTR( (PSTR)*pFrame, buffer ) )
#0118            {
#0119               strcpy(pszParams, buffer);
#0120               pszParams += strlen(buffer);
```

```
#0121            }
#0122        }
#0123
#0124        if ( (paramCount - i) != 1 )    // Tack on a comma if not last
#0125            *pszParams++ = ',';        // parameter
#0126
#0127        pFrame++;   // Bump frame up to the next DWORD value
#0128        pParams++;  // advance to next encoded parameter
#0129    }             // End of for() statement
#0130 }
#0131
#0132
#0133 BOOL GetLPSTR( PSTR ptr, PSTR buffer )
#0134 {
#0135    PSTR p = buffer;
#0136    int i;
#0137
#0138    *p++ = ':';     // Write out initial -> :" <-
#0139    *p++ = '\"';
#0140
#0141    for ( i=0; i < 10; i++ )
#0142    {
#0143        if ( !IsBadReadPtr( ptr, 1 ) && *ptr )
#0144        {
#0145            *p = *ptr++;
#0146            if ( *p == '\r' ) { *p++ = '\\'; *p = 'r'; }
#0147            else if ( *p == '\n' ) { *p++ = '\\'; *p = 'n'; }
#0148            else if ( *p == '\t' ) { *p++ = '\\'; *p = 't'; }
#0149
#0150            p++;
#0151        }
#0152        else
#0153            break;
#0154    }
#0155
#0156    if ( i == 0 )   // Not a valid string
#0157        return FALSE;
#0158
#0159    *p++ = '\"';    // Valid string ptr - end quote and null
#0160    *p++ = 0;       // terminate the string
#0161
#0162    return TRUE;
#0163 }
#0164
#0165 void LogReturn(PSTR pszName, DWORD returnValue, DWORD level)
#0166 {
```

```
#0167     char szIndent[128];
#0168
#0169     if ( !PLogFile )
#0170        return;
#0171
#0172     MakeIndentString(szIndent, level);
#0173     fprintf(PLogFile, "%s%s returns: %X\n", szIndent, pszName, returnValue);
#0174     fflush(PLogFile);
#0175  }
#0176
#0177
#0178  void MakeIndentString(PSTR buffer, UINT level)
#0179  {
#0180     DWORD cBytes = level * 2;
#0181     memset(buffer, ' ', cBytes);
#0182     buffer[cBytes] = 0;
#0183  }
```

**圖 10-8　LOG.C 把 API 函式名稱和參數寫到輸出檔去。**

RETURN.C 所有的碼都與「攔截 API 函式的回返值」有關。最前面兩個函式，*InitThreadReturnStack* 和 *ShutdownThreadReturnStack*，會被每一個執行緒各呼叫一次（譯註：請看圖 **10-4** 的 *DllMain*）。*InitThreadReturnStack* 配置一塊大小與 PER_THREAD_DATA 結構相同的記憶體，並將它初始化。PER_THREAD_DATA 內含兩個元件，是執行緒的「回返位址堆疊」所必須的：一個 HOOKED_FUNCTION 結構陣列和一個堆疊指標（請看圖 **10-9**）。

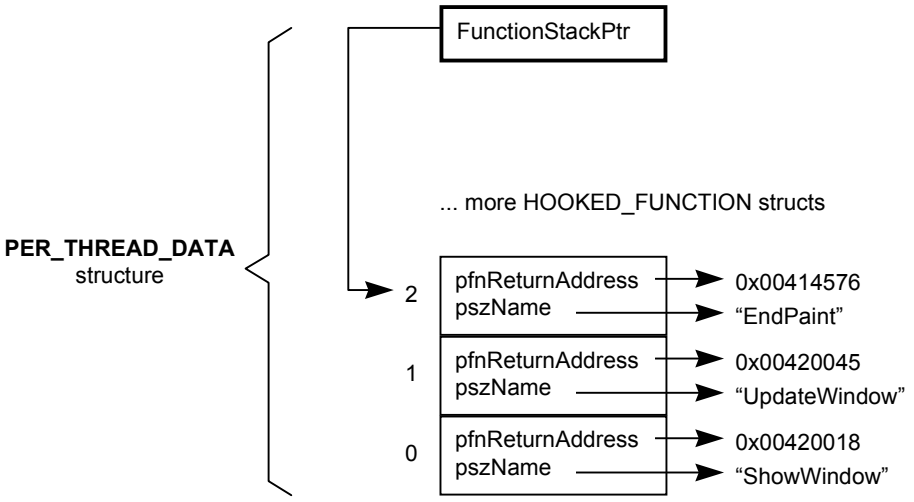



**圖 10-9　PER_THREAD_DATA 結構內含堆疊指標和 HOOKED_FUNCTION 結構陣列，後者持有執行緒的「回返位址堆疊」。**

每當一個被攔截的 API 函式被呼叫，它原先的回返位址和名稱指標就被寫到一個可用的 HOOKED_FUNCTION 結構中。然後，堆疊指標被累加 1。以此方式實作堆疊，可允許堆疊指標（事實上是一個索引，不是指標）對應到正確的「巢狀呼叫」層級。運轉記錄碼可以利用這個優點，將輸出文字縮排。PER_THREAD_DATA 結構被儲存在 Thread Local Storage 的一個 slot 之中，那是 DLL 初始化時配置的（譯註：請看 *InitThreadReturnStack* 內的 if (firstTime) 的行爲）。

*InterceptFunctionReturn* 被圖 **10-8** 的 *LogCall* 呼叫 -- 就在控制權即將跳回原 API 函式之前。這個函式首先把被攔截函式的回返位址和名稱，放到「回返位址堆疊」之中。然後以 *AsmCommonReturnPoint*（位於 ASMRETRN.ASM）的函式位址，改寫回返位址。

RETURN.C 的最後一個函式是 *CCommonReturnPoint*，它被 *AsmCommonReturnPoint* 函式呼叫。雖然我能夠以組合語言碼完成所有的事情，但我希望儘可能保持 APISPY32 的 C 語言風貌。*CCommonReturnPoint* 首先呼叫 *LogReturn* 以記錄被攔截函式的回返值。

然後它將這個值寫到保留在堆疊中的一個特別空間（因爲組合語言函式之故），然後回到組合語言碼。**10-10** 顯示 RETURN.C。

```
#0001  void AsmCommonReturnPoint(void);
#0002
#0003  DWORD TlsIndex = 0xFFFFFFFF;
#0004
#0005  BOOL InitThreadReturnStack(void)
#0006  {
#0007      PPER_THREAD_DATA pPerThreadData;
#0008
#0009      static BOOL firstTime = TRUE;
#0010
#0011      if ( firstTime )
#0012      {
#0013          TlsIndex = TlsAlloc();
#0014          firstTime = FALSE;
#0015      }
#0016
#0017      if ( TlsIndex == 0xFFFFFFFF )
#0018          return FALSE;
#0019
#0020      pPerThreadData = malloc( sizeof(PER_THREAD_DATA) );
#0021      if ( !pPerThreadData )
#0022          return FALSE;
#0023
#0024      pPerThreadData->FunctionStackPtr = 0;
#0025
#0026      TlsSetValue(TlsIndex, pPerThreadData);
#0027
#0028      return TRUE;
#0029  }
#0030
#0031  BOOL ShutdownThreadReturnStack(void)
#0032  {
#0033      PPER_THREAD_DATA pPerThreadData;
#0034
#0035      if ( TlsIndex == 0xFFFFFFFF )
#0036          return FALSE;
#0037
#0038      pPerThreadData = TlsGetValue( TlsIndex );
#0039      if ( pPerThreadData )
#0040          free( pPerThreadData );
#0041
```

```
#0042     return TRUE;
#0043  }
#0044
#0045  BOOL InterceptFunctionReturn(PSTR pszName, PDWORD pFrame)
#0046  {
#0047     PPER_THREAD_DATA pStack;
#0048     DWORD i;
#0049
#0050     pStack = (PPER_THREAD_DATA)TlsGetValue(TlsIndex);
#0051     if ( !pStack )
#0052        return FALSE;
#0053
#0054     if ( pStack->FunctionStackPtr >= (MAX_HOOKED_FUNCTIONS-1) )
#0055        return FALSE;
#0056
#0057     i = pStack->FunctionStackPtr;
#0058
#0059     pStack->FunctionStack[i].pfnReturnAddress = (PVOID)pFrame[0];
#0060     pStack->FunctionStack[i].pszName = pszName;
#0061     pStack->FunctionStackPtr++;
#0062
#0063     pFrame[0] = (DWORD)AsmCommonReturnPoint;
#0064
#0065     return TRUE;
#0066  }
#0067
#0068  // return_address <- pFrame[8]
#0069  // EAX          <- pFrame[7]
#0070  // ECX          <- pFrame[6]
#0071  // EDX          <- pFrame[5]
#0072  // EBX          <- pFrame[4]
#0073  // ESP          <- pFrame[3]
#0074  // EBP          <- pFrame[2]
#0075  // ESI          <- pFrame[1]
#0076  // EDI          <- pFrame[0]
#0077
#0078  //
#0079  // Common return point for all functions that we've intercepted.
#0080  // Called by _AsmCommonReturnPoint in ASMRETRN.ASM
#0081  // pFrame is a pointer to the stack frame set up by the PUSHAD
#0082  // (see above comment for the layout of this frame)
#0083  //
#0084  void CCommonReturnPoint( PDWORD pFrame )
#0085  {
#0086     PPER_THREAD_DATA pStack;
#0087     DWORD i;
```

```
#0088
#0089     // Get the function stack for the current thread
#0090     pStack = (PPER_THREAD_DATA)TlsGetValue(TlsIndex);
#0091     if ( !pStack )
#0092         return;
#0093
#0094     i = --pStack->FunctionStackPtr;
#0095
#0096     // Emit the information about the function return value to the logging
#0097     // mechanism.
#0098     LogReturn(pStack->FunctionStack[i].pszName, pFrame[7], i);
#0099
#0100     // Patch the return address back to what it was when the function
#0101     // was originally called.
#0102     pFrame[8] = (DWORD)pStack->FunctionStack[i].pfnReturnAddress;
#0103 }
```

**圖 10-10　RETURN.C 記錄函式的回返值。**

當我開始寫 APISPY32 時，我希望全部使用 C 語言。不幸的是，在我所玩的這個堆疊遊戲中（希望在一個被攔截函式回返時，取得控制權），我沒辦法以 C 語言乾乾淨淨地完成任務。而且，C 函式會破壞暫存器值（譯註：而你知道，我們所希望獲知的被攔截函式的回返值係放在 EAX）。我希望 APISPY32 儘可能不要影響被刺探程式，所以我選擇在 C 函式前後自行 push 和 pop「general-purpose 暫存器」。ASMRETRN.ASM 非常簡短，它首先將 ESP 減去 4，保留空間給原先的回返位址。C 函式碼基本上會以正確的位址填入該 DWORD，所以當組合語言碼回返，會回到正確的位置，並且堆疊指標也會像進入此函式時一樣。

```
#0001 .386
#0002 .model flat
#0003
#0004 extrn _CCommonReturnPoint:proc
#0005
#0006 .code
#0007
#0008 public _AsmCommonReturnPoint
#0009
#0010 _AsmCommonReturnPoint proc
```

```
#0011      SUB     ESP,4   ; Make space for return address
#0012      PUSHAD
#0013      MOV     EAX,ESP
#0014      PUSH    EAX
#0015      CALL    _CCommonReturnPoint
#0016      ADD     ESP,4
#0017      POPAD
#0018      RET
#0019  _AsmCommonReturnPoint endp
#0020
#0021  END
```

**圖 10-11　ASMRETRN.ASM 中的函式傳回 hooking code。**

# Win32s 碼

剩餘的 APISPY32.DLL 碼是專爲 Win32s 準備的，放在 W32SSUPP.C 之中。其中的唯一函式 *GetModuleBaseFromWin32sHMod* 顯示於圖 **10-12**。此函式的任務是取得 Win32s hModule（那並不是模組的基底位址），把它轉換爲基底位址。瀏覽過 Win32s 文件後，我找不到任何乾淨的（或甚至只是公開的）方法來做此轉換。然而我知道像 Win32s 的 *GetProcAddress* 需要做某些類似的事情。利用 SoftIce/W 進入 Win32s 函式庫，我發現兩個位於 W32SKRNL.DLL 的函式，正是我所需要。一個是 *_ImteFromHModule@4*，它取一個 Win32s HMODULE 並傳回一個被稱爲 IMTE 的內部 handle（第 3 章曾有描述）。第二個函式是 *BaseAddrFromImte*，它需要一個 IMTE 做爲參數，傳回一個 32 位元線性基底位址，那兒正是模組載入之處。

由於這些函式都是專屬於 Win32s，我並不直接在 APISPY32.DLL 中呼叫它們，因爲如此一來這個 DLL 就不能夠在 Windows 95 和 Windows NT 中被載入了。直接呼叫它們將使聯結器爲它們放置對應的待修正記錄（fixup），而載入器沒有辦法在載入該 DLL 之時發現它們的蹤跡。因此，我使用標準技術，也就是先呼叫 *GetProcAddress* 取得兩個函式的指標，再經由指標呼叫它們。

```
#0001  typedef DWORD (__stdcall *XPROC)(DWORD);
#0002
#0003  DWORD GetModuleBaseFromWin32sHMod(HMODULE hMod)
#0004  {
#0005      XPROC ImteFromHModule, BaseAddrFromImte;
#0006      HMODULE hModule;
#0007      DWORD imte;
#0008
#0009      hModule = GetModuleHandle("W32SKRNL.DLL");
#0010      if( !hModule )
#0011          return 0;
#0012
#0013      ImteFromHModule = (XPROC)GetProcAddress(hModule, "_ImteFromHModule@4");
#0014      if ( !ImteFromHModule )
#0015          return 0;
#0016
#0017      BaseAddrFromImte = (XPROC)GetProcAddress(hModule, "_BaseAddrFromImte@4");
#0018      if ( !BaseAddrFromImte )
#0019          return 0;
#0020
#0021      imte = ImteFromHModule( (DWORD)hMod);
#0022      if ( !imte )
#0023          return 0;
#0024
#0025      return BaseAddrFromImte(imte);
#0026  }
```

**圖 10-12　W32SSUPP.C 之中是 Win32s 專屬函式。**

# APISPYLD 碼

我們剩下的唯一問題就是 spy 軟體的程式載入器了（譯註：不是系統的那個程式載入器）。載入器本身也是一個程式，利用 *CreateProcess* 啓動被刺探程式。在目標行程開始執行任何碼之前，載入器把 spy DLL（APISPY32.DLL）注射到目標行程中。然後，載入器就不需要再做什麼事兒了，只要守住一個 *WaitForDebugEvent* 迴路，直到目標行程結束。我把載入器命名爲 APISPYLD，其原始檔比其他任何一個原始檔都大，所以我把它切爲數段來解說。

APISPYLD 的第一部份是使用者介面。*WinMain* 是個簡單的 while 迴路，不斷循環直到程式被成功刺探完畢或直到使用者選擇離開 spy 軟體。while 迴路首先引發一個對話盒，取得欲刺探的程式名稱。如果 *DialogBox* 傳回值不是 0，*WinMain* 呼叫 *LoadProcessForSpying*。如果行程被成功啓動，*WinMain* 呼叫 *DebugLoop*，喞取所有除錯訊息，直到目標行程結束。

APISPYLD 的對話盒極爲小巧（圖 **10-13**）。文字輸入區用來輸入命令列，包括程式名稱以及可有可無的參數。【File...】按鈕會引發一個 *GetOpenFileName* 對話盒，使你可以做選擇題而不是填充題。按下【Run】會使得指定之行程被載入。按下【Cancel】可以解散對話盒並將程式結束。

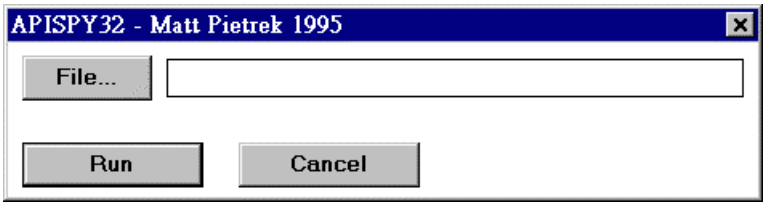



**圖 10-13　APISPYLD 對話盒讓你指定欲刺探之程式**

對話盒函式 *APISPY32DlgProc* 十分簡單，只反應三個訊息：WM_INITDIALOG、WM_COMMAND、WM_CLOSE。WM_INITDIALOG 使我們得有機會取出對話盒所獲得的命令列字串。這個命令列會被儲存在一個名爲 APISPY32.INI 內。下次當 APISPYLD 再被執行起來，這個命令列字串會被自動放進對話盒的 edit control 內。

對話盒函式的實質內容用於處理 WM_COMMAND 訊息。它把這訊息交給 *Handle_WM_COMMAND* 輔助函式。【File...】按鈕會令程式呼叫 *GetProgramName* 函式，那只不過是 *GetOpenFileName* 函式的一層包裝而已。如果 *GetProgramName* 成功，程式名稱會被放到 edit control 中。【Run】按鈕告訴 APISPYLD 把 edit control 的內容拷貝到 SzCmdLine 全域變數中，並退出對話盒，回返值 1。【Cancel】按鈕也是退出對話盒，但回返值爲 0，使 *WinMain* 不企圖載入任何東西。

如果使用者正確輸入一個適當的命令列，並按下【Run】鈕，對話盒會結束，控制權回到 *WinMain*。*WinMain* 於是呼叫 *LoadProcessForSpying*，把全域變數 SzCmdLine 遞交過去。*LoadProcessForSpying* 其實只是 *CreateProcess* 的一層外殼，唯一有趣的部份是，*CreateProcess* 的 fdwCreate 旗標參數被指定爲 DEBUG_ONLY_THIS_PROCESS。這告訴作業系統說，我們的程式（APISPYLD）打算成爲即將被載入的這個程式的除錯器。它也通知作業系統說，我們只對這個行程所發出的除錯訊息感興趣，不關心該行程所產生之子行程。如果我指定的不是 DEBUG_ONLY_THIS_PROCESS 而是 DEBUG_PROCESS，那麼除錯器（本例爲 APISPYLD）會獲得目標行程及其所有子行程的除錯訊息。**10-14** 展示 APISPYLD.C 的前段，包括使用者介面和行程載入部份。

```
#0001  //======================== Global Variables ================================
#0002  char SzINISection[] = "Options";
#0003  char SzINICmdLineKey[] = "CommandLine";
#0004  char SzINIFile[] = "APISPY32.INI";
#0005  char SzCmdLine[MAX_PATH];
#0006
#0007  BOOL FFirstBreakpointHit = FALSE, FSecondBreakpointHit = FALSE;
#0008
#0009  PROCESS_INFORMATION ProcessInformation;
#0010  CREATE_PROCESS_DEBUG_INFO ProcessDebugInfo;
#0011
#0012  CONTEXT OriginalThreadContext, FakeLoadLibraryContext;
#0013  PVOID PInjectionPage;
#0014
#0015  #define PAGE_SIZE 4096
#0016  BYTE OriginalCodePage[PAGE_SIZE];
#0017  BYTE NewCodePage[PAGE_SIZE];
#0018
#0019  //======================== Code ============================================
#0020
#0021  //
#0022  // Function prototypes
#0023  //
#0024  BOOL  CALLBACK APISPY32DlgProc(HWND, UINT, WPARAM, LPARAM);
#0025  void  Handle_WM_COMMAND(HWND hWndDlg, WPARAM wParam, LPARAM lParam);
#0026  void  Handle_WM_INITDIALOG(HWND hWndDlg, WPARAM wParam, LPARAM lParam);
#0027  BOOL  GetProgramName(HWND hWndOwner, PSTR szFile, unsigned nFileBuffSize);
#0028  BOOL  LoadProcessForSpying(PSTR SzCmdLine);
#0029  void  DebugLoop(void);
```

```
#0030  DWORD HandleDebugEvent( DEBUG_EVENT * event );
#0031  void  HandleException(LPDEBUG_EVENT lpEvent, PDWORD continueStatus);
#0032  void  EmptyMsgQueueOfUselessMessages(void);
#0033  BOOL  InjectSpyDll(void);
#0034  BOOL  ReplaceOriginalPagesAndContext(void);
#0035  PVOID FindUsablePage(HANDLE hProcess, PVOID PProcessBase);
#0036  BOOL  GetSpyDllName(PSTR buffer, UINT cBytes);
#0037
#0038
#0039  int APIENTRY WinMain( HANDLE hInstance, HANDLE hPrevInstance,
#0040                       LPSTR lpszCmdLine, int nCmdShow )
#0041  {
#0042      // This dialog returns 0 if the user pressed cancel
#0043      while ( 0 != DialogBox(hInstance, "APISPY32_LOAD_DLG", 0,
#0044                             (DLGPROC)APISPY32DlgProc) )
#0045      {
#0046          if ( LoadProcessForSpying(SzCmdLine) )
#0047          {
#0048              DebugLoop();
#0049              break;
#0050          }
#0051
#0052          MessageBox(0, "Unable to start program", 0, MB_OK);
#0053      }
#0054
#0055      return 0;
#0056  }
#0057
#0058  BOOL CALLBACK APISPY32DlgProc(HWND hWndDlg, UINT msg,
#0059                               WPARAM wParam, LPARAM lParam)
#0060  {
#0061      switch ( msg )
#0062      {
#0063          case WM_COMMAND:
#0064              Handle_WM_COMMAND(hWndDlg, wParam, lParam);
#0065              return TRUE;
#0066          case WM_INITDIALOG:
#0067              Handle_WM_INITDIALOG(hWndDlg, wParam, lParam);
#0068              return TRUE;
#0069          case WM_CLOSE:
#0070              EndDialog(hWndDlg, 0);
#0071              return FALSE;
#0072      }
#0073
#0074      return FALSE;
#0075  }
```

```
#0076
#0077  void Handle_WM_COMMAND(HWND hWndDlg, WPARAM wParam, LPARAM lParam)
#0078  {
#0079     if ( wParam == IDC_RUN )
#0080     {
#0081        if ( GetWindowText( GetDlgItem(hWndDlg, IDC_CMDLINE),
#0082                        SzCmdLine, sizeof(SzCmdLine)) )
#0083        {
#0084           WritePrivateProfileString(SzINISection, SzINICmdLineKey,
#0085                                     SzCmdLine, SzINIFile);
#0086           EndDialog(hWndDlg, 1);  // Return TRUE
#0087        }
#0088        else
#0089        {
#0090           MessageBox( hWndDlg, "No program selected", 0, MB_OK);
#0091        }
#0092     }
#0093     else if ( wParam == IDC_FILE )
#0094     {
#0095        if ( GetProgramName(hWndDlg, SzCmdLine, sizeof(SzCmdLine)) )
#0096           SetWindowText( GetDlgItem(hWndDlg, IDC_CMDLINE), SzCmdLine );
#0097     }
#0098     else if ( wParam == IDCANCEL )
#0099     {
#0100        EndDialog(hWndDlg, 0);
#0101     }
#0102  }
#0103
#0104  void Handle_WM_INITDIALOG(HWND hWndDlg, WPARAM wParam, LPARAM lParam)
#0105  {
#0106     GetPrivateProfileString(SzINISection, SzINICmdLineKey, "", SzCmdLine,
#0107                           sizeof(SzCmdLine), SzINIFile);
#0108     SetWindowText( GetDlgItem(hWndDlg, IDC_CMDLINE), SzCmdLine );
#0109  }
#0110
#0111  static char szFilter1[] = "Programs (*.EXE)\0*.EXE\0";
#0112
#0113  BOOL GetProgramName(HWND hWndOwner, PSTR szFile, unsigned nFileBuffSize)
#0114  {
#0115     OPENFILENAME ofn;
#0116
#0117     szFile[0] = 0;
#0118
#0119     memset(&ofn, 0, sizeof(OPENFILENAME));
#0120
#0121     ofn.lStructSize = sizeof(OPENFILENAME);
```

```
#0122     ofn.hwndOwner = hWndOwner;
#0123     ofn.lpstrFilter = szFilter1;
#0124     ofn.nFilterIndex = 1;
#0125     ofn.lpstrFile= szFile;
#0126     ofn.nMaxFile = nFileBuffSize;
#0127     ofn.lpstrFileTitle = 0;
#0128     ofn.nMaxFileTitle = 0;
#0129     ofn.lpstrInitialDir = 0;
#0130     ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST;
#0131
#0132     return GetOpenFileName(&ofn);
#0133  }
#0134
#0135  BOOL LoadProcessForSpying(PSTR SzCmdLine)
#0136  {
#0137     STARTUPINFO startupInfo;
#0138
#0139     memset(&startupInfo, 0, sizeof(startupInfo));
#0140     startupInfo.cb = sizeof(startupInfo);
#0141
#0142     return CreateProcess(
#0143               0,                        // lpszImageName
#0144               SzCmdLine,                 // lpszCommandLine
#0145               0,                        // lpsaProcess
#0146               0,                        // lpsaThread
#0147               FALSE,                     // fInheritHandles
#0148               DEBUG_ONLY_THIS_PROCESS,    // fdwCreate
#0149               0,                        // lpvEnvironment
#0150               0,                        // lpszCurDir
#0151               &startupInfo,              // lpsiStartupInfo
#0152               &ProcessInformation        // lppiProcInfo
#0153               );
#0154  }
```

**圖 10-14　APISPYLD.C 的前段，包括使用者介面和行程載入部份。**

APISPYLD.C 的中段專心在所謂的「除錯迴路」上：一個呼叫 *WaitForDebugEvent* 和 *ContinueDebugEvent* 的迴路，直到我們所刺探的那個行程結束爲止。每次 *WaitForDebugEvent* 回返，就是一個新的 XXX_DEBUG_EVENT（例如 EXCEPTION_DEBUG_EVENT 或 CREATE_THREAD_DEBUG_EVENT）發生時刻。*DebugLoop* 把每一個除錯訊息交給 *HandleDebugEvent* 函式，讓它去做必要的回應。

*HandleDebugEvent* 會忽略大部份 events，並把 DBG_CONTINUE 交給 *ContinueDebugEvent*。然而，在目標行程執行期間所發生的兩個異常情況對於我們的程式載入器十分重要。爲了這個理由，我把異常情況的處理（exception handling）另外獨立爲一個函式：*HandleException*。

我們的程式載入器應該看到的第一個異常情況是中斷點異常：EXCEPTION_BREAKPOINT（在 WINBASE.H 中定義）。這個中斷點並不是設定在目標行程內，而是內嵌在作業系統中的 INT 3。它將在新行程的第一個指令之前執行。當我們的 *HandleException* 看到這個異常情況，並且確定這是中斷點的第一次觸發，就呼叫 *InjectSpyDll*（稍後詳述），將 spy DLL 注射到子行程的位址空間中。

*HandleException* 應該看到的第二個異常是 *InjectSpyDll* 注射到程式碼中的那個中斷點，它使 APISPYLD 能夠在 APISPY32.DLL 載入後重獲控制權。這個中斷點被觸發後，APISPYLD 就知道目標行程已經完成了 APISPY32.DLL 的載入工作，原先被 *InjectSpyDll* 儲存下來的 memory pages 和 thread context 現在必須回存到目標行程去。*HandleException* 呼叫 *ReplaceOriginalPagesAndContext* 以完成這些瑣事。

這段碼的最後一部份是特別爲 Win32s 而準備。稍早我曾說過當 *WaitForDebugEvent* 回返，表示有一個新的 debug event 正等待被處理。在 Win32s 之下這不一定是真的。取而代之的是，Win32s 的 *WaitForDebugEvent* 傳回 TRUE -- 如果有另一個訊息正在等待。如果沒有任何訊息正在等待則傳回 FALSE。另一個不曾公開的 Win32s 怪癖是，系統 "post" 到除錯器（本例爲 APISPYLD）訊息佇列中的視窗訊息，其視窗 handle 都是 NULL。被放到該訊息佇列中的訊息，訊息號碼是這麼來的：

```
RegisterWindowMessage("W32S_Debug_Msg");
```

如果你沒有把這些訊息從佇列中移走，你的訊息佇列將被填滿，而真正的視窗訊息沒辦法進入。爲了處理這奇怪的 Win32s 行爲，如果 *WaitForDebugEvent* 傳回 FALSE 並且如果程式在 Win32s 下執行。我們的除錯迴路將呼叫 *EmptyMsgQueueOfUselessMessage*。

*EmptyMsgQueueOfUselessMessage* 是一個簡單的函式，呼叫 *PeekMessage*（PM_REMOVE），直到後者傳回 FALSE。任何訊息若夾帶著非零的 HWND，就會被交到 *DispatchMessage* 手上。到目前爲止，我還不曾看過這裡所獲得的訊息有合法的（非零的）HWND。在清空訊息佇列之後，*DebugLoop* 再次呼叫 *WaitForDebugEvent*，等待下一個 debug event 的發生。**範例 10-15** 顯示 APISPYLD.C 中的除錯迴路以及 debug event 的處理。

```
#0001  void DebugLoop(void)
#0002  {
#0003      DEBUG_EVENT event;
#0004      DWORD continueStatus;
#0005      BOOL fWin32s;
#0006      BOOL fWaitResult;
#0007
#0008      fWin32s = (GetVersion() & 0xC0000000) == 0x80000000;
#0009
#0010      while ( 1 )
#0011      {
#0012          fWaitResult = WaitForDebugEvent(&event, INFINITE);
#0013
#0014          if ( (fWaitResult == FALSE) && fWin32s )
#0015          {
#0016              EmptyMsgQueueOfUselessMessages();
#0017              continue;
#0018          }
#0019
#0020          continueStatus = HandleDebugEvent( &event );
#0021
#0022          if ( event.dwDebugEventCode == EXIT_PROCESS_DEBUG_EVENT )
#0023              return;
#0024
#0025          ContinueDebugEvent( event.dwProcessId,
#0026                              event.dwThreadId,
#0027                              continueStatus );
#0028      }
#0029  }
#0030
#0031  PSTR SzDebugEventTypes[] =
#0032  {
#0033  "",
#0034  "EXCEPTION",
```

```
#0035  "CREATE_THREAD",
#0036  "CREATE_PROCESS",
#0037  "EXIT_THREAD",
#0038  "EXIT_PROCESS",
#0039  "LOAD_DLL",
#0040  "UNLOAD_DLL",
#0041  "OUTPUT_DEBUG_STRING",
#0042  "RIP",
#0043  };
#0044
#0045  DWORD HandleDebugEvent( DEBUG_EVENT * event )
#0046  {
#0047      DWORD continueStatus = DBG_CONTINUE;
#0048      // char buffer[1024];
#0049
#0050      // wsprintf(buffer, "Event: %s\r\n",
#0051      //         SzDebugEventTypes[event->dwDebugEventCode]);
#0052      // OutputDebugString(buffer);
#0053
#0054
#0055      if ( event->dwDebugEventCode == CREATE_PROCESS_DEBUG_EVENT )
#0056      {
#0057          ProcessDebugInfo = event->u.CreateProcessInfo;
#0058      }
#0059      else if ( event->dwDebugEventCode == EXCEPTION_DEBUG_EVENT )
#0060      {
#0061          HandleException(event, &continueStatus);
#0062      }
#0063
#0064      return continueStatus;
#0065  }
#0066
#0067  void HandleException(LPDEBUG_EVENT lpEvent, PDWORD continueStatus)
#0068  {
#0069      // char buffer[128];
#0070      // wsprintf(buffer, "Exception code: %X  Addr: %08X\r\n",
#0071      //         lpEvent->u.Exception.ExceptionRecord.ExceptionCode,
#0072      //         lpEvent->u.Exception.ExceptionRecord.ExceptionAddress);
#0073      // OutputDebugString(buffer);
#0074
#0075
#0076      if ( lpEvent->u.Exception.ExceptionRecord.ExceptionCode
#0077             == EXCEPTION_BREAKPOINT )
#0078      {
#0079          if ( FFirstBreakpointHit == FALSE )
#0080          {
```

```
#0081            InjectSpyDll();
#0082            FFirstBreakpointHit = TRUE;
#0083        }
#0084        else if ( FSecondBreakpointHit == FALSE )
#0085        {
#0086            ReplaceOriginalPagesAndContext();
#0087            FSecondBreakpointHit = TRUE;
#0088        }
#0089
#0090        *continueStatus = DBG_CONTINUE;
#0091    }
#0092    else
#0093    {
#0094        *continueStatus = DBG_EXCEPTION_NOT_HANDLED;
#0095    }
#0096 }
#0097
#0098 void EmptyMsgQueueOfUselessMessages(void)
#0099 {
#0100    MSG msg;       // See PeekMessage loop for explanation of idiocy
#0101
#0102    // Win32s idiocy puts W32s_Debug_Msg message in our message queue
#0103    // Dispose of them!  They're useless!
#0104    while ( PeekMessage(&msg, 0, 0, 0, PM_REMOVE) )
#0105    {
#0106        if ( msg.hwnd )
#0107            DispatchMessage(&msg);
#0108    }
#0109 }
```

**圖 10-15　APISPYLD.C 中除錯迴路的開始以及 debug event 的處理。**

APISPYLD 的最後一段碼用來把 APISPY32 DLL 注射到被刺探行程的位址空間中 -- 在第一個中斷點被觸發之後。*InjectSpyDll* 是一個複雜的函式，大致上可以切割爲三個階段。第一階段用來定出子行程（也就是被刺探程式）中的重要位址。其中主要的趣味在於，目標行程中的第一個 writeable data page 的位址並不在 .idata section 中。此外像 KERNEL32.DLL 的 *LoadLibrary* 函式位址、被載入之 DLL 的名稱（APISPY32.DLL），以及將行程的第一個執行緒的 thread context 儲存起來，也都是很重要的事情。

第二階段，*InjectSpyDll* 把第一個 writeable data page 拷貝到名爲 OriginCodepage 的全域變數中（這個命名不好。早期的 APISPY32 以第一個 code page 而非第一個 writeable data page 儲存其注射碼。我懶得改變命名）。注意，爲了把我們即將修改的 page 做出一份副本，我們必須呼叫 *ReadProcessMemory*。要知道，我們正在製作副本的這一個 page，位在另一個行程中，我們沒辦法直接存取它。

*InjectSpyDll* 的第三階段是設定某些動作，使目標行程「獲得執行權時，會呼叫 *LoadLibrary* 將 APISPY32.DLL 載入」。我們所要製造的一小段碼是靠著填寫 FAKE_LOADLIBRARY_CODE 結構而建構起來的（譯註：十分類似 APISPY32.DLL 中的 stub）。結構中的欄位若不是一個組合語言指令（opcode），就是前一指令的運算元（operand）。結構尾端放的是 spy DLL 名稱。我把 DLL 名稱放進結構中是因爲 DLL 名稱必須在目標行程中（而不是 APISPYLD 行程中）被看見。這個結構填寫完畢之後，我使用 *WriteProcessMemory* 把整個結構拷貝到目標行程的適當 page 之中（譯註：以 *FindUsablePage* 找出來的那一個）。然後，*InjectSpyDll* 利用 *SetThreadContext* 改變 EIP 值 -- 那是當目標行程握有執行權時將使用的指令指標（Instruction Pointer）。這個 EIP 將被設定指向我們所準備的一小段碼（譯註：也就是 FAKE_LOADLIBRARY_CODE 結構）的第一個指令處。

> 譯註：經過這一折騰，我們所準備的這一小段碼現在看起來，就好像渾然天成地出現在目標行程的位址空間中一樣。

假設 *InjectSpyDll* 一切順利，目標行程重獲控制權時（譯註：在 APISPYLD 做了那麼多偷天換日的工作之後）將執行 *LoadLibrary*。當 *LoadLibrary* 回返，指令指標（EIP）將落在 *LoadLibrary* 呼叫動作之後的那個中斷點上（是我們自己加上的），這會使得目標行程再次被凍住，而 APISPYLD 行程中的 *WaitForDebugEvent* 獲得一個 EXCEPTION_DEBUG_EVENT。當 *HandleException* 看到這個異常情況（譯註：這是第二個異常），它就知道要將稍早所修改的 page 恢復原狀，thread context 也要恢復原狀。恢復執行緒狀態的碼放在 *ReplaceOriginalPagesAndContext* 函式中，利用

*WriteProcessMemory* 寫回原來內容，然後呼叫 *SetThreadContext*，將我們在注射 DLL 之前所儲存的 thread context 傳交過去。**範例 10-16** 顯示 APISPYLD.C 中的 *InjectSpyDll* 相關部份。

```
#0001  #pragma pack ( 1 )
#0002  typedef struct
#0003  {
#0004     WORD    instr_SUB;
#0005     DWORD   operand_SUB_value;
#0006     BYTE    instr_PUSH;
#0007     DWORD   operand_PUSH_value;
#0008     BYTE    instr_CALL;
#0009     DWORD   operand_CALL_offset;
#0010     BYTE    instr_INT_3;
#0011     char    data_DllName[1];
#0012  } FAKE_LOADLIBRARY_CODE, * PFAKE_LOADLIBRARY_CODE;
#0013
#0014  BOOL InjectSpyDll(void)
#0015  {
#0016     BOOL retCode;
#0017     DWORD cBytesMoved;
#0018     char szSpyDllName[MAX_PATH];
#0019     FARPROC pfnLoadLibrary;
#0020     PFAKE_LOADLIBRARY_CODE pNewCode;
#0021
#0022     // ================================================================
#0023     // Phase 1 - Locating addresses of important things
#0024     // ================================================================
#0025
#0026     pfnLoadLibrary = GetProcAddress( GetModuleHandle("KERNEL32.DLL"),
#0027                                      "LoadLibraryA" );
#0028     if ( !pfnLoadLibrary )
#0029        return FALSE;
#0030
#0031     PInjectionPage = FindUsablePage(ProcessInformation.hProcess,
#0032                                     ProcessDebugInfo.lpBaseOfImage);
#0033     if ( !PInjectionPage )
#0034        return FALSE;
#0035
#0036     if ( !GetSpyDllName(szSpyDllName, sizeof(szSpyDllName)) )
#0037        return FALSE;
#0038
#0039     OriginalThreadContext.ContextFlags = CONTEXT_CONTROL;
```

```
#0040     if ( !GetThreadContext(ProcessInformation.hThread,&OriginalThreadContext))
#0041        return FALSE;
#0042
#0043     // ====================================================================
#0044     // Phase 2 - Saving the original code page away
#0045     // ====================================================================
#0046
#0047     // Save off the original code page
#0048     retCode = ReadProcessMemory(ProcessInformation.hProcess, PInjectionPage,
#0049                              OriginalCodePage, sizeof(OriginalCodePage),
#0050                              &cBytesMoved);
#0051     if ( !retCode || (cBytesMoved != sizeof(OriginalCodePage)) )
#0052        return FALSE;
#0053
#0054     // ====================================================================
#0055     // Phase 3 - Writing new code page and changing the thread context
#0056     // ====================================================================
#0057
#0058     pNewCode = (PFAKE_LOADLIBRARY_CODE)NewCodePage;
#0059
#0060     pNewCode->instr_SUB = 0xEC81;
#0061     pNewCode->operand_SUB_value = 0x1000;
#0062
#0063     pNewCode->instr_PUSH = 0x68;
#0064     pNewCode->operand_PUSH_value = (DWORD)PInjectionPage
#0065                         + offsetof(FAKE_LOADLIBRARY_CODE, data_DllName);
#0066
#0067     pNewCode->instr_CALL = 0xE8;
#0068     pNewCode->operand_CALL_offset =
#0069            (DWORD)pfnLoadLibrary - (DWORD)PInjectionPage
#0070            - offsetof(FAKE_LOADLIBRARY_CODE,instr_CALL) - 5;
#0071
#0072     pNewCode->instr_INT_3 = 0xCC;
#0073
#0074     lstrcpy(pNewCode->data_DllName, szSpyDllName); // Copy DLL name
#0075
#0076     // Write out the new code page
#0077     retCode = WriteProcessMemory(ProcessInformation.hProcess, PInjectionPage,
#0078                              &NewCodePage, sizeof(NewCodePage),
#0079                              &cBytesMoved);
#0080     if ( !retCode || (cBytesMoved != sizeof(NewCodePage)) )
#0081        return FALSE;
#0082
#0083     FakeLoadLibraryContext = OriginalThreadContext;
#0084     FakeLoadLibraryContext.Eip = (DWORD)PInjectionPage;
#0085
```

```
#0086     if ( !SetThreadContext(ProcessInformation.hThread,
#0087                            &FakeLoadLibraryContext) )
#0088        return FALSE;
#0089
#0090     return TRUE;
#0091  }
#0092
#0093  BOOL ReplaceOriginalPagesAndContext(void)
#0094  {
#0095     BOOL retCode;
#0096     DWORD cBytesMoved;
#0097
#0098     retCode = WriteProcessMemory(ProcessInformation.hProcess, PInjectionPage,
#0099                                  OriginalCodePage, sizeof(OriginalCodePage),
#0100                                  &cBytesMoved);
#0101     if ( !retCode || (cBytesMoved != sizeof(OriginalCodePage)) )
#0102        return FALSE;
#0103
#0104     if ( !SetThreadContext(ProcessInformation.hThread,
#0105                            &OriginalThreadContext) )
#0106        return FALSE;
#0107
#0108     return TRUE;
#0109  }
#0110
#0111  PVOID FindUsablePage(HANDLE hProcess, PVOID PProcessBase)
#0112  {
#0113     DWORD peHdrOffset;
#0114     DWORD cBytesMoved;
#0115     IMAGE_NT_HEADERS ntHdr;
#0116     PIMAGE_SECTION_HEADER pSection;
#0117     unsigned i;
#0118
#0119     // Read in the offset of the PE header within the debuggee
#0120     if ( !ReadProcessMemory(ProcessInformation.hProcess,
#0121                             (PBYTE)PProcessBase + 0x3C,
#0122                             &peHdrOffset,
#0123                             sizeof(peHdrOffset),
#0124                             &cBytesMoved) )
#0125        return FALSE;
#0126
#0127
#0128     // Read in the IMAGE_NT_HEADERS.OptionalHeader.BaseOfCode field
#0129     if ( !ReadProcessMemory(ProcessInformation.hProcess,
#0130                             (PBYTE)PProcessBase + peHdrOffset,
#0131                             &ntHdr, sizeof(ntHdr), &cBytesMoved) )
```

```
#0132         return FALSE;
#0133
#0134     pSection = (PIMAGE_SECTION_HEADER)
#0135                ((PBYTE)PProcessBase + peHdrOffset + 4
#0136                + sizeof(ntHdr.FileHeader)
#0137                + ntHdr.FileHeader.SizeOfOptionalHeader);
#0138
#0139     for ( i=0; i < ntHdr.FileHeader.NumberOfSections; i++ )
#0140     {
#0141         IMAGE_SECTION_HEADER section;
#0142
#0143         if ( !ReadProcessMemory( ProcessInformation.hProcess,
#0144                                  pSection, &section, sizeof(section),
#0145                                  &cBytesMoved) )
#0146            return FALSE;
#0147
#0148         // OutputDebugString( "trying section: " );
#0149         // OutputDebugString( section.Name );
#0150         // OutputDebugString( "\r\n" );
#0151
#0152         // If it's writeable, and not the .idata section, we'll go with it
#0153         if ( (section.Characteristics & IMAGE_SCN_MEM_WRITE)
#0154              && strncmp(section.Name, ".idata", 6) )
#0155         {
#0156            // OutputDebugString( "using section: " );
#0157            // OutputDebugString( section.Name );
#0158            // OutputDebugString( "\r\n" );
#0159
#0160            return (PVOID) ((DWORD)PProcessBase + section.VirtualAddress);
#0161         }
#0162
#0163         pSection++; // Not this section.  Advance to next section.
#0164     }
#0165
#0166     return 0;
#0167 }
#0168
#0169 BOOL GetSpyDllName(PSTR buffer, UINT cBytes)
#0170 {
#0171     char szBuffer[MAX_PATH];
#0172     PSTR pszFilename;
#0173
#0174     // Get the complete path to this EXE - The spy dll should be in the
#0175     // same directory.
#0176     GetModuleFileName(0, szBuffer, sizeof(szBuffer));
#0177
```

```
#0178     pszFilename = strrchr(szBuffer, '\\');
#0179     if ( !pszFilename )
#0180        return FALSE;
#0181
#0182     lstrcpy(pszFilename+1, "APISPY32.DLL");
#0183     strncpy(buffer, szBuffer, cBytes);
#0184     return TRUE;
#0185  }
```

**圖 10-16　APISPYLD.C 中的 InjectSpyDll 相關部份**

# APISPY32 的使用注意事項

欲使用 APISPY32 刺探程式，請執行 APISPYLD 程式，在編輯欄位中輸入一個命令列，或使用【File...】按鈕瀏覽可執行檔。一旦檔名輸出完成，按下【Run】鈕。APISPYLD 對話盒會消失，你所選擇的可執行檔會跑起來。在刺探目標結束執行之後，應該可以在相同的磁碟目錄中找到一個 .OUT 檔。**圖 10-17** 顯示以 APISPY32 刺探 Win32 CLOCK 程式的一部份輸出結果。

```
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
```

**圖 10-17　以 APISPY32 刺探 Win32 CLOCK 的輸出結果**

大部份時候，.OUT 檔中的一行函式呼叫記錄之後，會緊跟著一行函式回返值記錄。然而，並不是絕對如此。請注意**圖 10-17** 中的 *DefWindowProcA* 有縮排現象。這表示此一函式是在執行外圍函式（例如本例的 *SetWindowTextA*）時被呼叫的。這個特殊的程序其實倒也合理，因爲 *DefWindowProc* 的第二個參數（訊息編號）是 0x0C。看看 WINUSER.H，你會發現 0x0C 代表 WM_SETTEXT。由於 *DefWindowProc* 是在 *SetWindowText* 中被呼叫，我們可以安全地假設 *SetWindowText* 送出一個 WM_SETTEXT 訊息給程式的視窗函式。這個函式沒有處理它，於是交到 *DefWindowProc* 手上。在這一份 APISPY32 輸出，我們只看到一層縮排。四、五層縮排不是不可能，特別容易發生於程式的結束過程（收到 WM_CLOSE 之後）。

注意 LPSTR 參數。完整的字串不一定會全部展現出來。我決定列印最前 10 個字元，或是遇到 null 字元爲止。tab 字元、carriage return 字元和 linefeed 字元被我轉換爲 \t、\r 和 \n -- 如果列出原始字元，OUT 檔案的格式就會亂掉。

如果你在 Win32s 1.2 版（或更早版本）執行 APISPY32，面對 16 位元的 USER.EXE 除錯版，你會得到許多 RIPs。這是 Win32s 之中有關於 Win16 和 Win32 之間的訊息傳遞的一個臭蟲。在 Win16 至 Win32 的演進過程中，有一些訊息被重複編碼了。Win32s 的 thunking layer 必須知道是哪些訊息被重複編碼。爲了知道哪一個訊息要被轉化，Win32s 必須知道訊息所屬視窗的類別。所以 Win32s 呼叫 USER.EXE 的 *GetClassName*。但是當視窗的 HWND 是 0，問題來了，*GetClassName* 除錯版會因此產生 RIPs。我們的 spy 軟體在哪裡產生 HWND 爲 0 的訊息？先前我說過，Win32s 的 *WaitForDebugEvent* 會將 *RegisterWindowMessage*(W32S_Debug_Msg) 訊息 "post" 給除錯器的訊息佇列。

如果你以 Borland C++ 建立 APISPY32，刺探那些多緒程式，幸運之神不會常常站在你這一邊。Borland C++ 的多緒函式庫對某些函式（例如 APISPY32 所使用的 *fprintf*）使用執行緒專屬資料。在 Borland C++ 的 runtime library 中，程式碼並沒有對 DLL_THREAD_ATTACH notification 特別付出關心。取而代之的是，它們依賴程式呼叫 *_beginthread* 函式以知道何時配置其執行緒專屬資料。不幸的是這個方法應用在那種「被其他行程所產生的執行緒」身上就失效了。在我的 spy 軟體中，APISPY32.DLL 身上的

Borland runtime library 不會看到被刺探程式呼叫 *_beginthread*。Borland 已經知道這一問題，但是在 BC++ 4.5 中還沒有修正過來。

## 在你自己的程式內攔截函式

本章一開始，我曾經承諾過，你可以將 APISPY32 的函式攔截技術，應用在自己的程式碼中。這段碼就是**圖 10-18** 的 HOOKAPI.C。其中的 *HookImportedFunction* 允許你攔截「一個模組對著另一模組之某函式所發出的所有呼叫」。假設你使用一個 DLL 名爲 FOO.DLL，你可以攔截 FOO.DLL 中所有對 *MessageBeep* 的呼叫-- 甚至即使你沒有 FOO.DLL 的原始碼。如果你也想攔截 BAR.DLL 和 BAZ.DLL 對 *MessageBeep* 的呼叫，你必須針對每一個 DLL 都呼叫一次 *HookImportedFunction*。

另一個必須記住的重點是，這個攔截技術只能攔截你自己行程中的輸入函式（import function），不能攔截被其他行程呼叫的 API 函式。換句話說你只能夠攔截你的 EXE 及其 DLLs 所發出的呼叫。你不能夠希望它「攔截 WINFILE 對 *OpenFile* 的所有呼叫」，因爲你的攔截碼沒有辦法映射到 WINFILE 的行程位址空間。

*HookImportedFunction* 的第一個參數是 EXE 或 DLL（你要攔截的模組，以上例而言就是 FOO.DLL）的 module handle。第二個參數是你希望攔截的函式的所屬模組（名稱）。第三個參數是你希望攔截的函式名稱。最後一個參數是你希望導引過去的函式位址。*HookImportedFunction* 會傳回被攔截函式的原來位址。如有必要，你可以利用這個位址串連原函式。如果依上面的舉例，*HookImportedFunction* 使用如下：

```
#0001  pfnOriginalProc = HookImportedFunction(GetModuleHandle("BAR.DLL"),
#0002                                          "USER32.DLL",
#0003                                          "MessageBeep",
#0004                                          MyMessageBeepHandler );
#0005
#0006  // Macro for adding pointers/DWORDs together without C arithmetic interfering
#0007  #define MakePtr( cast, ptr, addValue ) (cast)( (DWORD)(ptr)+(DWORD)(addValue))
#0008
#0009  DWORD GetModuleBaseFromWin32sHMod(HMODULE hMod); // Prototype (defined below)
```

```
#0010
#0011  PROC WINAPI HookImportedFunction(
#0012         HMODULE hFromModule,       // Module to intercept calls from
#0013         PSTR    pszFunctionModule,  // Module to intercept calls to
#0014         PSTR    pszFunctionName,    // Function to intercept calls to
#0015         PROC    pfnNewProc         // New function (replaces old function)
#0016         )
#0017  {
#0018     PROC pfnOriginalProc;
#0019     PIMAGE_DOS_HEADER pDosHeader;
#0020     PIMAGE_NT_HEADERS pNTHeader;
#0021     PIMAGE_IMPORT_DESCRIPTOR pImportDesc;
#0022     PIMAGE_THUNK_DATA pThunk;
#0023
#0024     if ( IsBadCodePtr(pfnNewProc) ) // Verify that a valid pfn was passed
#0025        return 0;
#0026
#0027     // First, verify the the module and function names passed to use are valid
#0028     pfnOriginalProc = GetProcAddress( GetModuleHandle(pszFunctionModule),
#0029                                 pszFunctionName );
#0030     if ( !pfnOriginalProc )
#0031        return 0;
#0032
#0033     if ( (GetVersion() & 0xC0000000) == 0x80000000 )
#0034        pDosHeader =                                        // Win32s
#0035            (PIMAGE_DOS_HEADER)GetModuleBaseFromWin32sHMod(hFromModule);
#0036     else
#0037        pDosHeader = (PIMAGE_DOS_HEADER)hFromModule;          // other
#0038
#0039     // Tests to make sure we're looking at a module image (the 'MZ' header)
#0040     if ( IsBadReadPtr(pDosHeader, sizeof(IMAGE_DOS_HEADER)) )
#0041        return 0;
#0042     if ( pDosHeader->e_magic != IMAGE_DOS_SIGNATURE )
#0043        return 0;
#0044
#0045     // The MZ header has a pointer to the PE header
#0046     pNTHeader = MakePtr(PIMAGE_NT_HEADERS, pDosHeader, pDosHeader->e_lfanew);
#0047
#0048     // More tests to make sure we're looking at a "PE" image
#0049     if ( IsBadReadPtr(pNTHeader, sizeof(IMAGE_NT_HEADERS)) )
#0050        return 0;
#0051     if ( pNTHeader->Signature != IMAGE_NT_SIGNATURE )
#0052        return 0;
#0053
#0054     // We know have a valid pointer to the module's PE header.  Now go
#0055     // get a pointer to its imports section
```

```
#0056     pImportDesc = MakePtr(PIMAGE_IMPORT_DESCRIPTOR, pDosHeader,
#0057                           pNTHeader->OptionalHeader.
#0058                           DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].
#0059                           VirtualAddress);
#0060
#0061     // Bail out if the RVA of the imports section is 0 (it doesn't exist)
#0062     if ( pImportDesc == (PIMAGE_IMPORT_DESCRIPTOR)pNTHeader )
#0063        return 0;
#0064
#0065     // Iterate through the array of imported module descriptors, looking
#0066     // for the module whose name matches the pszFunctionModule parameter
#0067     while ( pImportDesc->Name )
#0068     {
#0069        PSTR pszModName = MakePtr(PSTR, pDosHeader, pImportDesc->Name);
#0070
#0071        if ( stricmp(pszModName, pszFunctionModule) == 0 )
#0072           break;
#0073
#0074        pImportDesc++;  // Advance to next imported module descriptor
#0075     }
#0076
#0077     // Bail out if we didn't find the import module descriptor for the
#0078     // specified module.  pImportDesc->Name will be non-zero if we found it.
#0079     if ( pImportDesc->Name == 0 )
#0080        return 0;
#0081
#0082     // Get a pointer to the found module's import address table (IAT)
#0083     pThunk = MakePtr(PIMAGE_THUNK_DATA, pDosHeader, pImportDesc->FirstThunk);
#0084
#0085     // Blast through the table of import addresses, looking for the one
#0086     // that matches the address we got back from GetProcAddress above.
#0087     while ( pThunk->u1.Function )
#0088     {
#0089        if ( pThunk->u1.Function == (PDWORD)pfnOriginalProc )
#0090        {
#0091           // We found it!  Overwrite the original address with the
#0092           // address of the interception function.  Return the original
#0093           // address to the caller so that they can chain on to it.
#0094           pThunk->u1.Function = (PDWORD)pfnNewProc;
#0095           return pfnOriginalProc;
#0096        }
#0097
#0098        pThunk++;   // Advance to next imported function address
#0099     }
#0100
#0101     return 0;   // Function not found
```

```
#0102  }
#0103
#0104  typedef DWORD (__stdcall *XPROC)(DWORD);
#0105
#0106  // Converts an HMODULE under Win32s to a base address in memory
#0107  DWORD GetModuleBaseFromWin32sHMod(HMODULE hMod)
#0108  {
#0109      XPROC ImteFromHModule, BaseAddrFromImte;
#0110      HMODULE hModule;
#0111      DWORD imte;
#0112
#0113      hModule = GetModuleHandle("W32SKRNL.DLL");
#0114      if( !hModule )
#0115          return 0;
#0116
#0117      ImteFromHModule = (XPROC)GetProcAddress(hModule, "_ImteFromHModule@4");
#0118      if ( !ImteFromHModule )
#0119          return 0;
#0120
#0121      BaseAddrFromImte = (XPROC)GetProcAddress(hModule, "_BaseAddrFromImte@4");
#0122      if ( !BaseAddrFromImte )
#0123          return 0;
#0124
#0125      imte = ImteFromHModule( (DWORD)hMod);
#0126      if ( !imte )
#0127          return 0;
#0128
#0129      return BaseAddrFromImte(imte);
#0130  }
```

**圖 10-18　HOOKAPI.C 讓你攔截你自己程式中的函式呼叫。**

你自己那個用來取代原函式的函式，應該和被攔截函式有完全一樣的型式。這樣你才得以處理所有的函式參數，並使得編譯器在函式回返時從堆疊中吐出正確的位元組個數。如果你要把控制權最後再交回給被攔截的那個函式，請呼叫由 *HookImportedFunction* 傳回的位址。

爲了示範 *HookImportedFunction* 的使用，我寫下了 SimonSez 程式。這個程式非常簡單，它攔截 SimonSez 對 *MessageBox* 的呼叫，將 "SimonSez:" 字串加在原字串之前，然後才顯示出來。當 SimonSez 呼叫 *MessageBox*，我的 *MyMessageBox* 會起來，而不是

USER32.DLL 中的那個 *MessageBox*。然後 *MyMessageBox* 才呼叫 USER32.DLL 的 *MessageBox*。**範例 10-19** 是 SIMONSEZ 程式碼。

```
#0001  //=================================
#0002  // SIMONSEZ - Matt Pietrek 1995
#0003  // FILE: HOOKAPI.C
#0004  //=================================
#0005  #include <windows.h>
#0006  #include <malloc.h>
#0007  #include "hookapi.h"
#0008
#0009  // Make a typedef for the WINAPI function we're going to intercept
#0010  typedef int (__stdcall *MESSAGEBOXPROC)(HWND, LPCSTR, LPCSTR, UINT);
#0011
#0012  MESSAGEBOXPROC PfnOriginalMessageBox;   // for storing original address
#0013
#0014  //
#0015  // A special version of MessageBox that always prepends "Simon Sez: "
#0016  // to the text that will be displayed.
#0017  //
#0018  int WINAPI MyMessageBox( HWND hWnd, LPCSTR lpText,
#0019                       LPCSTR lpCaption, UINT uType )
#0020  {
#0021      int retValue;             // real MessageBox return value
#0022      PSTR lpszRevisedString;     // pointer to our modified string
#0023
#0024      // Allocate space for our revised string - add 40 bytes for new stuff
#0025      lpszRevisedString = malloc( lstrlen(lpText) + 40 );
#0026
#0027      // Now modify the original string to first say "Simon Sez: "
#0028      if ( lpszRevisedString )
#0029      {
#0030          lstrcpy(lpszRevisedString, "Simon Sez: ");
#0031          lstrcat(lpszRevisedString, lpText);
#0032      }
#0033      else                                // If malloc() failed, just
#0034          lpszRevisedString = (PSTR)lpText;   // use the original string.
#0035
#0036      // Chain on to the original function in USER32.DLL.
#0037      retValue = PfnOriginalMessageBox(hWnd,lpszRevisedString,lpCaption,uType);
#0038
#0039      if ( lpszRevisedString != lpText )  // If we sucessfully allocated string
#0040          free( lpszRevisedString );      // memory, then free it.
#0041
```

```
#0042     return retValue;    // Return whatever the real MessageBox returned
#0043  }
#0044
#0045  int APIENTRY WinMain( HANDLE hInstance, HANDLE hPrevInstance,
#0046                        LPSTR lpszCmdLine, int nCmdShow )
#0047  {
#0048     MessageBox(0, "MessageBox Isn't Intercepted Yet", "Test", MB_OK);
#0049
#0050     // Intercept the calls that this module (TESTHOOK) makes to
#0051     // MessageBox() in USER32.DLL.  The function that intercepts the
#0052     // calls will be MyMessageBox(), above.
#0053
#0054     PfnOriginalMessageBox = (MESSAGEBOXPROC) HookImportedFunction(
#0055                   GetModuleHandle(0),     // Hook our own module
#0056                   "USER32.DLL",          // MessageBox is in USE32.DLL
#0057                   "MessageBoxA",         // function to intercept
#0058                   (PROC)MyMessageBox);    // interception function
#0059
#0060     if ( !PfnOriginalMessageBox )   // Make sure the interception worked
#0061     {
#0062        MessageBox(0, "Couldn't hook function", 0, MB_OK);
#0063        return 0;
#0064     }
#0065
#0066     // !!!!!!!!!!!!!!!!!!!!!!!!  WARNING  !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
#0067        // When built with optimizations, the VC++ compiler loads a
#0068        // register with the address of MessageBoxA, and then makes all
#0069        // subsequent calls through it.  This can cause the MessageBox call
#0070        // below to not go through the Import Address table that we just patched.
#0071        // For this reason, the .MAK file for this program does not use the
#0072        // /O2 or /O1 switches.  This usually won't be a problem, but it
#0073        // was in this particularly simple program.  ACCKK!!!!
#0074
#0075     // Call MessageBox again.  However, since we've now intercepted
#0076     // MessageBox, control should first go to our own function
#0077     // (MyMessageBox), rather than the MessageBox() code in USER32.DLL.
#0078
#0079     MessageBox(0, "MessageBox Is Now Intercepted", "Test", MB_OK);
#0080
#0081     return 0;
#0082  }
```

**圖 10-19　SIMONSEZ.C，示範 HookImportedFunction 的使用。**

*HookImportedFunction* 是怎麼運作的？先前我說過，當一個 Win32 程式呼叫另一個模組所提供的函式，呼叫動作事實上是把控制權轉換到 JMP DWORD PTR[XXXXXXXX] 那一小段碼身上。在 XXXXXXXX 位址上的那個 DWORD 內含輸入函式（import function）的位址，例如 USER32.DLL 的 *MessageBox* 的位址。

*HookImportedFunction* 所做的就是搜尋 import address table（IAT），找出某個 DWORD 吻合被攔截函式的位址。一旦找到，*HookImportedFunction* 就以你的函式位址改寫之。

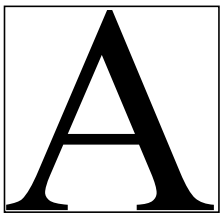
*HookImportedFunction* 的第一參數是 HMODULE，代表你打算攔截的對象（EXE 或 DLL）。在 Windows NT 和 95 之中，一個 HMODULE 只不過是一個線性位址，代表模組載入的基底，稱爲模組基底位址。由於 Win32 使用記憶體映射檔（memory mapped file），在基底位址處是一個 DOS MZ 表頭（WINNT.H 中稱爲 IMAGE_DOS_HEADER）。利用其中的 e_lfanew 欄位，*HookImportedFunction* 可以定出 PE 表頭（WINNT.H 中稱爲 IMAGE_NT_HEADER）的位置。IMAGE_NT_HEADER 尾端是一個結構陣列，每個結構內含模組中重要區域的位址。我們特別感興趣的是 import address table 的起始點。這個表格（通常位在 .idata section 內）內含 imported functions 的相關資訊。表格中的某處便是剛剛所說的 *HookImportedFunction* 必須改寫的 DWORD。

Import address table 是一個由 IMAGE_IMPORT_DESCRIPTOR 結構組成的陣列（請看 WINNT.H 以了解該結構內容）。這個模組如果與某個 DLL 聯結，該 DLL 就會在模組記憶體內擁有一個 IMAGE_IMPORT_DESCRIPTOR 結構。陣列的尾端是以全部爲 0 的 IMAGE_IMPORT_DESCRIPTOR 結構做爲終結記號。每一個 IMAGE_IMPORT_DESCRIPTOR 有一個指標指向 DLL 名稱、一個指標指向 import address table。*HookImportedFunction* 尋訪各個 IMAGE_IMPORT_DESCRIPTORs 直到發現有 DLL 名稱與本函式的 pszFunctionModule 參數相符，然後使用 IMAGE_IMPORT_DESCRIPTOR 內的資訊產生一個指標，指向 import address table。

*HookImportedFunction* 一開始便呼叫 *GetProcAddress* 取得我們打算攔截的函式的位址。這個位址應該和我們剛剛所說的 import address table 之中的一筆相符。*HookImportedFunction* 搜尋整個位址陣列，直到它找到一筆位址與我們以 *GetProcAddress* 所獲得的位址相符。剩餘的工作就是把新函式位址（pfnNewProc）拷貝到該筆資料上，並傳回原函式位址。

# 摘要

Win32 程式設計爲 Win16 環境下工作的程式員帶來一大組新的挑戰。一般而言，Win32 系統程式設計比較更有限制也更複雜，主要是因爲像分離位址空間和多執行緒等等題目。這一章中當我們設計一個 spy 軟體以及設計一個一般性的 API 函式攔截機制時就遇到了這些問題。我們同時也看到，儘管微軟曾經大聲疾呼：只有一套 Win32 API，事實上在你走入細節之後，偶而你會發現些許的差異。瞭解了這些主題之後，現在的你已經可以開發出具備工業強度，可以在所有 Win32 平台上跑的軟體了。



# KERNEL32.DLL 未公開函式之 Import Library

Windows 95 KERNEL32.DLL 輸出函式表格（exports table）的前 100 個項目是以序號開放，而所有正常的 Windows API 函式都是以函式名稱和序號開放出來。以名稱開放，可以允許你呼叫 *GetProcAddress* 並交給它一個函式名稱，然後獲得該函式的位址。

很明顯，由於微軟沒有以名稱開放其 KERNEL32 的前 100 個函式，表示他不希望你使用它們。換個說法，那些就是未公開函式。我們都知道，未公開函式可能有大用途。事實上有時候你完成某個特殊目標的唯一方法就是使用未公開函式。然而，由於這些函式不以名稱開放，你不能夠藉由 *GetProcAddress* 使用它們。

一般而言這種情況阻止不了高手，因爲他們知道 *GetProcAddress* 不只接受函式名稱，也接受函式序號。而且，就如第 3 章所說，微軟的 KERNEL32 設計者也開放了一個後門。但是，*GetProcAddress* 就是不接受 KERNEL32 函式的序號。微軟阻止人們使用 KERNEL32.DLL 前 100 個函式的企圖至爲明顯。

別怕。一如你在本書其他地方看到的，這些人爲限制可以打破。方法之一是自己寫一個 *GetProcAddress*。這並不如想像中那麼難，因爲一個載入後的 PE 模組的格式已經有了詳細的公開。第 3 章甚至已經給你 *GetProcAddress* 的虛擬碼了。

問題在於，如果你親手打造 *GetProcAddress*，呼叫它然後把其傳回值儲存爲一個函式指標恐怕是一種痛苦。另一個方法就簡單得多：使用一個包含這些未公開函式的 import library。我們知道微軟並未提供這樣的東西，因此，這個附錄告訴你如何使用 Visual C++ 或其他的微軟編譯器/聯結器，製作出這樣一個 import library。**圖 A-1** 展示的 K32LIB.DEF，內含 100 個未公開的 KERNEL32.DLL 函式。

```
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).
```

**圖 A-1　你可以以這些未公開的 KERNEL32.DLL 函式列表，使用 Visual C++ 或其他編譯器和聯結器，產生自己的 import library。**

或許你已注意到，我並沒有在圖 **A-1** 中列出所有未公開函式的原型（prototype）。雖然我可以另寫一章，完整列出未公開函式的原型並說明它們，但這並不是本書的目標。這些函式的說明說不定會出現在未來其他文章中。請注意其中一些函式如 *VxDCall0@0*，在本書其他地方用過。

K32LIB.DEF 和 K32LIB.LIB（for Visual C++）被我放在書附磁片中。你可以在磁片的 APPENDIX 子目錄中找到它們。一般而言微軟的聯結器會在聯結一個 DLL 時產生其 import library。然而微軟的 LIB.EXE 可以只根據一個 .DEF 檔就產生出一個 import library。爲了以 K32LIB.DEF 重建 K32LIB.LIB，你可以使用光碟片同一檔案目錄中的 MAKE.BAT。MAKE.BAT 內容如下：

```
lib /MACHINE:IX86 /DEF:K32LIB.DEF
```

爲了能夠在你的專案中使用 K32LIB.LIB，你應該在指定給聯結器的一長串 import libraries 中，把 K32LIB.LIB 緊鄰放在 KERNEL32.LIB 之後，這會強迫微軟聯結器把 K32LIB.LIB 的程式碼和資料續接在 KERNEL32.LIB 的程式碼和資料之後。你將會在可執行檔中看到對 KERNEL32.DLL 有兩份參考。別太在意那個！EXE 檔中「與 KERNEL32.DLL 對應的 IMAGE_IMPORT_DESSCRIPTOR 表頭」將有兩個。但並不是每一份拷貝的所有內容都用來描述輸入函式（import functions）。關於微軟聯結器在這一層面上的動作，說來話長，我就不提了。

Borland C++ 使用者可以將 K32LIB.DEF 餵給 IMPLIB.EXE 以獲得一個 TLINK 識得的 import library。命令列如下：

```
IMPLIB K32LIB.LIB K32LIB.DEF
```

你可以把 K32LIB.LIB 放在聯結器命令列的一長串 import libraries 的任何位置上。TLINK 並不在乎它的出現次序。

一如你在本書所看到的許多程式所示範，對於呼叫 Windows 95 未公開函式，K32LIB.LIB 有極高價值。當然啦，你應該儘可能避免使用未公開函式，除非那是唯一的路。因爲如果你使用未公開函式，你的程式就有可能無法在 Windows NT 或未來的 Windows 環境下執行。本書程式和 Windows 95 有緊密關係並且設計用來顯示 Windows 95 內部動作，所以我沒有辦法不使用未公開函式。

如果你有絕對的理由使用未公開函式，請使用版本控制以及其他的強固性（robustness）測試，以便讓你的程式可以姿態優雅地結束。爲了這麼做，你必須使用 *GetProcAddress* 而不要直接呼叫它們。這就是走在刀鋒邊緣的代價。