《Windows 95 系統程式設計 – 虛擬機器與 VxD 程式設計》
(System Programming for Windows 95 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 (松崗) 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 ☺。

* *

只要 Windows 95/Windows 98 還大量存在的一天，只要 VxD programming 仍有大量需求的一天，本書即深具價值。

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

開放檔案如下：
檔名：system-programming-for-windows95.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
虛擬裝置與 VxD 程式設計
Systems
Programming
for Windows 95
Walter Oney
Microsoft Press

WINDOWS 95 系統程式設計

虛擬機器與 VxD 程式設計

原著：System Programming for Windows 95

Walter Oney 著

侯俊傑 譯

松崗電腦圖書資料股份有限公司

關於作者

自從 1968 畢業於 MIT（麻省理工學院）後，Walter Oney 就成爲一位專業的系統程式開發人員。1990 年開始，他專攻低階的 Windows 程式設計。在一家 DOS Extender 主要供應廠商任職期間，他協助起草 DPMI（Dos Protected Mode Interface）標準。Oney 寫出第一套 Windows extenders，可以在 Windows 3.1 上執行 32 位元 Windows 程式。現在他是一位獨立顧問，幫助許多公司及個人學習 Windows 系統程式設計。

關於譯者

自從 1983 畢業於 NCTU（交通大學），並經過兩年兵役、兩年工作、兩年研究所，於 1989 年畢業於 NTHU（清華大學）後，侯俊傑就成爲一位專業的軟體開發人員。1990 年開始，他專攻 Windows 程式設計，並逐次下探 Windows 系統深處。他曾服務於工研院，現在元智大學開課。他是一位獨立而自由的研究人員，曾經幫助許多公司及個人學習 Windows 環境上的各種技術。

譯　序

（侯俊傑）

我於 1997 年初譯完 Matt Pietrek 的 ***Windows 95 System Programming Secrets*** （中譯爲 **Windows 95 系統程式設計大奧秘**，旗標出版）之後，網路上的 CompBook（電腦書訊）版貼出了這樣一則 post：

> 作者　Imanaka (八方歸去)
> 時間　Sat Mar 15 01:05:29 1997
> ---
> Matt Pietrek　我最崇拜的作家
> ***Windows 95 System Programming Secrets***　我最愛的書
> 現在居然有中文本了...看來我又得去買一本中文本，中英對照看了☺
>
> 任何喜愛 Windows 程式的人千萬不能錯過這本書... 不過得先懂點 SDK 就是了... 看完這本書後，幾乎能解答所有關於 Windows 95 OS 方面的疑問... 大家快去買吧！侯俊傑先生既然有勇氣翻譯這種高技術性的書籍，大家也應該多多支持和鼓勵，這樣才有更好的翻譯書... 例如 *Inside OLE* 或是 ***System Programming for Windows 95*** 等等。嘿嘿！會不會太狠了點？

不，不會太狠！現在，這本 **Windows 95 系統程式設計 - VMs & VxDs** 就是我對所有需要系統層面的進階好書的讀友的回應。

這本書的原文出版日期是 1996 年第二季。我一向對 Virtual Machine 和 Virtual Device Driver（VxD）的原理甚感興趣，因此很快就享用了原文本。一年半載過去，中譯本卻一直未見面世！國內電腦書商爭取譯權的動作向來十分迅速，這麼好而且重要的一本書遲遲沒有中譯本，我唯一能夠想到的原因是，和內容深度以及市場考量有關。

事實上，市場的憂慮是多餘的，臺灣 PC 工業如此發達，對於這類書籍已經到了求書若渴的程度。臺灣電腦出版界對於高階技術領域的無能爲力，才真令人憂慮；臺灣高階技術翻譯人才的凋零難覓，能者不爲，爲者不能，才真令人憂慮。

1997 年春夏兩季，我應新竹科學園區多家廠商之邀（包括睿昱半導體、華邦電子、力捷電腦...），開授總近 100 小時的 Windows 作業系統課程。在系統軟體層面，我採用 ***Windows 95 System Programming Secrets*** 做爲教材（噢噢，當然是採用我的中譯本 ☺）；在低階 VxD 的入門層面，則爲學員大量補充這本 ***System Programming for Windows 95*** 的內容。其後，1997 年末松崗公司爭取了此書之中譯權，委由我負責翻譯。這對期待高階好書的讀者而言是一個美好的開始，對我而言則是一個艱鉅的任務。但我期許自己，藉由徹底熟讀此書的機會，提昇自己的技術水平，也藉此爲國內需要瞭解系統低階技術以及撰寫 VxD 之眾多工程師盡一份心力。是的，工程師閱讀原文書不是問題，但一本好的譯本可以在閱讀效率和吸收效率上帶來極大的幫助。

Windows 95 System Programming Secrets（Matt Pietrek 著）的讀者，大概會對這本名稱相似的 ***System Programming for Windows 95***（Walter Oney 著）乍見之下感到疑惑。基本上 Matt Pietrek 是在軟體層面討論作業系統，包括 processes、threads、modules、memory management、executable file format、windowing system、messaging system...。其中雖討論了 VxD，卻是以

使用者角度切入。Walter Oney 這本書走的是更低層次，從虛擬機器（VM）來看 Windows 95 作業系統，並以大量篇幅介紹 VxD 程式設計的技術與實作細節（包括 Plug & Play device, Communication Driver, Serial Port, VCOMM, IOS, IFS...）。其內容之詳盡豐富，技術之深刻紮實，實在是這個領域的罕見好書，是一本由淺而深，涵蓋初中高階的詳實好書。

對於任何需要撰寫 Windows 9x 驅動程式的朋友，或是想瞭解 Windows 9x 低階面貌（虛擬機器層面）的朋友，我鄭重向您推薦此書！

翻譯此書的 10 個月之中，Windows 98 問世了。我想許多人都對 Win98 WDM 和 Win95 VxD 之間的異同與定位，感到興趣。本書作者已在「前言」第一頁（頁碼 17）針對這一部份提出了他的看法，有興趣的讀者請特別注意。基本上，Win98 的 WDM 係以 VxD 模擬出來。VxD 仍然是 Windows 9x 的原生（native）技術。也因此，**本書所有技術均適用 Win95 和 Win98**。

我向來喜歡拿高階技術書籍挑戰自己的極限。這本書，實在說，後半部的 driver 程式實務已經超越了我的極限。本書原擬以合譯方式，由我負責前半，由業界朋友負責後半。後因朋友過於忙碌，遂幾乎全部由我完成。爲了讓後半部也有良好的水準，特別商請華碩公司幾位好朋友幫助檢閱。他們提出了許多技術上的見解，也提供了許多註解。這本書能夠在此刻以目前的水準出現於各位眼前，他們功不可沒。當然，本書的一切責任，仍然是屬於我的。

完成了這本書，如果還有人拿 *Inside OLE* 叫我譯，我真要「從命不如恭敬」了 ☺

侯俊傑 1998.12.01 于新竹
jjhou@ccca.nctu.edu.tw

- 本書 11~12 章之翻譯與技術推敲，得力於華邦公司陳欽燦先生甚多，謹致謝忱。

- 本書 11~16 章談到各種 drivers 的實作細節。這一部份非我專精，特別商請華碩公司的朋友擔任這些章節的檢閱工作：

  高雲慶先生：第 11, 12 章（以及 5, 6, 10, 17 各章中有關於「中斷」的部份）
  張訓賓先生：第 13 章
  吳俊良先生：第 14 章
  林居輝先生：第 15 章
  林偉博先生：第 16 章

對於他們的協助，我在此致上深深謝意。如果有更多業界朋友願意抽出時間在教育事業上出力，我們的高階翻譯人才及高階實務書籍，當不至凋零至此。

- 在我的教育理念中，原文技術名詞是不必也不該翻譯的。面對某些普及並得神韻的中文譯名，當然從眾爲佳；但許多原文技術名詞，強譯爲中文的結果是畫蛇添足。考慮到業界的習慣用語，以及此書的讀者層，我在書中保留了許多原文技術名詞（與動詞）。當然，我必需承認，用詞的選擇取捨，都是根源於我（譯者）個人的文字習慣與閱聽習慣，不見得爲每一個人所接受與喜愛。

- 一本技術書籍之索引（index），扮演十分重要的角色。由於我個人在排版軟體的操控上還沒有能力製作 index，所以此書如同絕大部份的中文書籍一樣，沒有 index。謹向讀者致歉。精良的 index 是未來中文書籍應該追求的目標之一。

以下是本書對於譯詞的處理原則，以 Q/A 方式呈現，由此可略窺本書風格：

Q: 原文名詞，時而譯之，時而未譯。

A: 由於電腦技術書籍（尤其如本書之類）有非常多的函式名稱、常數名稱、術語，在大量中英文夾雜的情況下，我個人並不絕對考量（實在也沒有能力做到）整體的譯詞統一和「該譯都譯」的層次。反倒是以「本書讀者之程度可接受，並可獲得原詞之隱含意義」爲重要考量。舉個例子，page, paging, pageable, 如果我把 page 譯爲分頁（名詞），paging 譯爲分頁（動詞），這基本上已經不太理想了。饒是如此，pageable 或可譯爲「可分頁的」，page fault 或可譯爲「分頁失敗」，page swapping 或可譯爲「分頁置換」。但因我已保留了 page 一詞，只好對於 paging, pageable, page fault, page swapping 統統保留。相信這是本書讀者樂見的。又例如，communication port error 雖可譯爲「通訊埠錯誤」，但先前我都保留 port 這個詞，所以...。此外，像 request 一詞，是我們工程師常掛在嘴上的，所以也保留了。

但是，留下這麼多原文名詞，再加上函式名稱、常數名稱、變數名稱、結構名稱...，整本書在某些地方變得蟹形字密度過高，這時候我可能又對於某些原已保留原文但若譯爲中文亦還可接受的詞，改用中文（在不造成迷惑的情況下）。

另一種情況是，我們或許可以把 entry 譯爲項目、條目、條孔、條格，但面對 IO service entry 我卻不好譯爲「IO service 項目」或「IO service 條目」。換句話說在一串完整的名詞裡，我比較不喜見到中英夾雜的現象。

再一種情況是，例如，三種 drivers："Nonompliant" drivers，"Port" drivers 和 "Generic" drivers。我們或許可將第三種譯爲「通用型 driver」，但爲與前兩者位階相等，仍沿用原文。

再拿 "hook into" 爲例，每一位有能力閱讀此書的讀者，都知道 hook 的意思。如果我前面保留了 hook 一詞，我也只好保留 "hook into" 這個詞，才有精簡的表現與足夠的意義。

cache 和 caching 也是一樣的情況。queue, queued, queueing 也是一樣的情況。

Q: 未譯之英文名詞加上 s，或未譯之英文動詞加上 ed，讀來非常不順。既然是中文書，理應依照中文單複數慣用法。

A: 中文很少有複數用法，只能加「們」，但「虛擬機器們」或「裝置們」是很滑稽的。在譯文中出現「名詞加 s」或「動詞加 ed」的情況，某些時候可以讓讀者更清楚意義。不過，我也的確發現不少非必要的 s 或 ed，這種情況下我便把它們拿掉了。

Q: 已經出現在章前註解或已中英文對照過的詞，不必每次都中英文同時出現。例如 mount 或 reentrant 或 entry。

A: 有些字我很想一直使用原文，例如 mount，例如 reentrant，例如 entry。但出現過多次之後，又使我信心動搖，覺得怪怪。尤其對於動詞，例如 mount，如果說：「我 mount 了一個硬碟」，似乎怪怪的，說「我裝載了一個硬碟」，又似乎沒有讓讀者得到 mount 的那個意思。所以，既然不太費事，我也就常常不厭其煩地讓這一類詞的中英文一起出現。

事實上大家都有經驗，讀一本書不見得從頭看起，也可能是日後需要抽出某個章節看三五頁。對於上述之類的動詞或名詞，如果突然看到，而沒有注意先前曾出現過的中英對照，就會覺得奇怪或不解。因此我常常不厭其煩地讓這一類詞中英文並現。

Q: 有些詞似乎不必再原文對照。

A: 這大約是指諸如 send 或 flush 之類的詞。send 若譯爲傳送或送出，讀者不知道它是 send 或是 post（這是兩種不同的訊息傳遞方式）。flush 若譯爲清除，讀者可能會以爲是 clean up 或 reset。舉一句爲例：

你應該掃清（flush）任何 cache，並清除（clean up）你所握有且代表 volume 的其他資源。

諸如此類的問題，在一整本書籍裡，層出不窮，很難做到符合每個人的理想。

換句話說，「統一譯詞」以及「中文形式之完整」這兩項，在此書已經難以企及了，所以我比較寧願「得義而忘言」。

當然，很多詞句的順眼與否，都根源於閱讀者的個人習慣。儘量減少讀者的疑慮，是我最大的考量（然而我的作法，其實也根源於我個人的閱聽習慣）。

本書 11~16 章，有許多檢閱者的註解。每一個大型註解皆以檢閱者爲名，例如 11,12 章爲高雲慶註，13 章爲張訓賓註... 依此類推。如果以「譯註」起頭，那就是譯者我（侯俊傑）所寫的註。

Windows 95 系統程式設計 - 虛擬機器與 VxDs 程式設計

目 錄

前 言

請注意，一項新技術（WDM）

Microsoft 在美國加州聖荷西（譯註：矽谷所在地）舉行的 1996 Windows Hardware Engineering Conference（WinHEC）研討會中宣佈了一項新技術：Win32 Driver Model（WDM）。WDM 技術讓你終於能夠寫出一個可同時適用於 Windows 95 和 Windows NT 的驅動程式。在 Windows NT kernel-mode driver 和 Windows 95 VxD 兩者之間，WDM 看起來比較像前者。由於 Microsoft 的宣佈過於簡潔，不少 WinHEC 與會人士不但認識不清，而且抱怨不知道是否該停止學習 VxD。

要瞭解 WDM 的重點，你必須先瞭解它的動機在哪裡。以我來看，WDM 的目的有兩個：**1.** 提供一種方法，擴充 Windows NT 的 Plug and Play 機能；**2.** 讓硬體廠商有能力供應消費性電子產品的驅動程式，使它們能夠連接 USB（譯註）和 1394 buses。至於「WDM 看起來比較像 Windows NT kernel-mode driver 而非 Windows 95 VxD」，這或許只是 Microsoft 內部策略下的一種措辭。

譯註：**USB**，**Universal Serial Bus**，可譯爲「通用序列匯流排」，是 Intel 於 1996 年正式對外公佈的一項技術。USB device 提供一種標準接頭，可直接與電腦串連。它本身有類似「集線器」的能力，使所有 USB device 可以視需要加掛或卸除。各種電腦設備（主機、螢幕、印表機、掃描機、數據機、滑鼠、鍵盤...）乃至於家電設備（數位相機、數位電話、麥克風、喇叭、數位烤麵包機 --- 如果有的話），只要支援 USB，就可以一節一節串接起來。此外，USB device 允許熱機安插（hot insertion）與熱機拔除（Hot removal），也就是說不必關機就裝卸週邊設備。Windows 98 宣稱將支援完整的 USB 介面。

我相信 WDM 在未來數年將只是漸進式地強化其重要性（譯註：而不是天翻地覆的大變化）。任何人今天如果將精力投注於 VxD 技術，只要世上除了 Windows NT 之外還有一項產品名爲 Windows，他就將繼續保有利益。Windows 95 及其後續繼承者將繼續支援 VxD。（譯註：在 Windows 98 已經問世的今天，我們看到，此言誠不虛也）。今天你爲哪些 devices 寫了 VxDs，未來它們仍然都需要 VxDs，直到 Microsoft 說服所有種類的 devices 都使用 WDM 的那一天來臨爲止。我想那一天沒那麼快來臨。猶有進者，Microsoft 已經宣稱，WDM 一開始只在 Windows 95 OEM releases 版有效。也就是說在一段時間之內，對於終端客戶而言，WDM 並不是一個大眾化的升級動作。

結論是，如果你即刻要擴充 Windows 95，或是要寫一個驅動程式，在 Windows 95 龐大裝機量的硬體平台上有效運作，你必須學會本書所說的技術。如果你要和 Windows 3.1 相容，本書技術也是你必須學習的。

對於 DDK 的參考

本書對 Windows 95 DDK（Device Driver Kit）的參考，是以 1996/04 版本爲主。更新的 DDK 版本或許並不內含本書所指的明確標題。此外，請你注意，所有對於 DDK 主題的參考，都是根據 MSDN（譯註）viewer 中的呈現，而不是根據 DDK 求助檔（DDPR.HLP）或 DDK Word 檔（.DOC）。

譯註：MSDN（Microsoft Developer's Network）是一個類似 user group 的組織。不過，一般談到 MSDN，其實指的是其最重要的產品：一季一期的 MSDN 光碟。MSDN 光碟分爲 level 1~4，內容份量各不相同。Level 1 內含 Microsoft Knowledge Base、各種規格書、範例、電子書、Microsoft Systems Journal 文章...，對於技術人員而言，已足夠成爲知識大寶庫。我個人非常建議走 Microsoft 路線之技術人員，購買 MSDN Level 1 光碟。價格低廉，臺灣有售，可詢問臺灣微軟公司。

Microsoft Visual C++ 4.1 的臭蟲（bug）

此書即將付梓之際，Microsoft 推出了 Visual C++ 4.1。4.1 版不幸有一個臭蟲（bug），會在處理 inline assembly statements 中的 enumerated constants 時發作。因此，你不能夠使用 4.1 版搭配 DDK 來建立 VxDs。不過你可以使用 4.0 版（譯註：VC++ 4.2 和 5.0 已經無此臭蟲）

書附光碟

Windows 95 的真實模式光碟機驅動程式會將光碟片中的長檔名截斷。如果你的 Windows 95 用的正是那種驅動程式，你就沒有辦法看到光碟片中的所有目錄和檔案。如果是這樣，你應該和你的光碟機廠商聯絡，換一個 Windows 95 版的光碟機驅動程式。不過你還是可以使用這片光碟，請這麼做：先從一個 SETUP 目錄中將檔案拷貝到你的硬碟，然後以適當的長檔名重新命名，然後你就可以觀看硬碟中的檔案了。要不然，你也可以觀看光碟 SETUP 目錄中的 8.3 檔名格式。

將光碟內容安裝到硬碟

請執行光碟根目錄中的 SETUP.EXE，然後按照螢幕上的指示進行安裝程序。DDK 求助檔（help file）以及我的註解檔將不會自動拷貝到你的硬碟中。

光碟內的求助檔（Help File）

光碟之中包含有一個 DDK 求助檔（help file），以及我對此求助檔的註解檔案。此求助檔是 DDK 1996/04 版本。MSDN 每季新片中或有更新的 DDK 版本，但我的註解檔對於新版 DDK 就不見得完全適用了。你可以從以下網站獲得最新的註解檔：

```
http://www.tiac.net/user/waltoney
```

光碟根目錄中的 README 檔對於求助檔和註解檔有更多說明，也有對於此一光碟內容的其他說明。

技術支援

我已盡最大力量確保光碟內容之正確性。Microsoft 在以下網站中提供了對此書籍的更正內容（如果有的話）：

```
http://www.microsoft.com/mspress/support/
```

如果你對此書或書附光碟有任何意見、問題、想法，請傳送 email 給上述網站，或使用：

郵政：

```
Microsoft Press
Attn : System Programming for Windows 95 Editor
One Microsoft Way
Redmond, WA 98052-6399 （譯註：別忘了加 U.S.A）
```

電子郵件：

```
MSPINPUT@MICROSOFT.COM
```

譯註：如果你對此中譯本有任何意見或想法，請傳送 email 給侯俊傑，謝謝：

```
jjhou@ccca.nctu.edu.tw
```

請注意，上述網站並不提供產品支援服務。關於 Visual C++，你可以打電話到 (206)635-7007（C/C++ Standard Support），星期之中的早上六點到晚上六點（當地時間）皆可。關於 MASM，你可以打電話到 (206)646-5109（Macro Assembler Standard Support），時段同前。如果你需要 VxD 程式設計的技術支援，請聯絡 (800)9363500，Microsoft Support Network Sales。

致謝

略

第一章

導入

Introduction

第 1 章

概觀（Overview）

六年前我就極需要這樣一本書了，那個時候我正要寫我的第一個虛擬裝置驅動程式（virtual device driver）。當時 Windows 3.0 剛面世，我必須讓一個 TSR（常駐程式）建立在一個 DOS extender 之上 -- 在 Windows 啓動以及結束（shutdown）的時候。要解決這個問題，我必須寫一個虛擬裝置驅動程式（VxD），但當時唯一一份可用的資料是 ***Virtual Device Adaptation Guide***（譯註：DDK 標準手冊）的一份草稿影本。說到如何設計、建立一個 VxD 並除錯，那份文件實在沒什麼用！我在那個專案（以及後續許多案子）中掙扎前進，破冰而行。我總是希望有人能夠寫下我經歷千辛萬苦而學來的技術。

我知道其他許多工程師並沒有足夠的 Windows 系統介面知識，足以解決他們要解決的問題。過去數年我花了許多時間在 CompuServe 的 WINSDK 論壇中，我看到許多人持續問一些我已經回答過的基本問題。Larry Niven 和 Jerry Pournelle 曾經寫下 ***The Mote in God's Eye*** （Simon and Schuster 出版，1974）這本書，講述一個外星球工程師，解決問題的能力非常好，但是溝通能力非常差。有時候我覺得 Microsoft 出版的 DDK 文件就像是給外星工程師看的，或許只有他們才能理解。

我想，如果工程師沒有辦法好好完成他們的任務，end user 就得受罪了。Microsoft 和

Windows 不必擔負罪名，因爲使用者沒辦法說出臭蟲遍地的原因。他們分不出好壞作業系統之間的區別（何況也不是這個因素），分不出好壞程式員之間的區別（何況也不是這個因素），分不出好壞文件之間的區別（這倒是重點）。我竭盡所能不要在這本書中打迷糊仗；我告訴你要做什麼（而不只單單告訴你不要做什麼），以及爲什麼這樣做。做爲一個出版者，Microsoft Press 並沒有給我任何特殊管道，讓我接觸任何「大內」資訊。我非常希望看看 Windows 95 的原始碼，以便能夠精確寫下「或許 Microsoft 希望你知道，以便你能夠好好使用其作業系統」的資訊，但是我沒有！因此，不管我多麼小心，也不管已經有多少工程師看過了我寫的東西，你還是有可能找出本書的錯誤 -- 即使那機會不大。

誰 應該閱讀本書

如果你希望學習如何使用 Windows 95 作業系統的最低階設備以完成你的工作，你應該閱讀本書。以下是本書可以幫助你完成的一些工作：

- 爲一個新的硬體撰寫 driver（驅動程式）。如果硬體廠商供應的是 32 位元保護模式驅動程式，而不是 MS-DOS 真實模式驅動程式或 16 位元 Windows DLL 驅動程式，那麼 Windows 95 會運作得比較好一些。這本書會傾囊相授，教你如何撰寫一個適合 Windows 95 系統架構的驅動程式。
- 將一個驅動程式升級。如果你已經能夠供應 Windows 3.x VxD，可能你沒有太多新東西要學。請試著閱讀第 11 章和第 12 章（關於 Plug and Play），看看是否你所需要的就只是在原來的 VxD 中加上一個 Plug and Play 外飾板（很可能唷）。如果你當初是從 16 位元驅動程式開始，那麼你就像其他程式員一樣，需得從頭學起 VxD 了。
- 撰寫一個系統工具的支援服務。除錯器啦、系統監視器啦、錯誤記錄器啦...都要求作業系統提供一些服務，而 Microsoft 卻沒有提供那些服務。寫一個 VxD 可以讓你進入作業系統最底層面 -- 用的是公開介面，不是未公開的那一套。
- 瞭解 Windows 95 的內部運作原理。你或許知道，Windows 95 依賴某些 INT 21h 擴充介面來執行檔案動作，但是你知道（舉個例子）INT 21h 的 file system

calls 最終可能由網路上另一部電腦裡頭的一個 VxD 提供服務嗎？本書可以解釋 Windows 95 中的這種神奇魔法，以及其他的神奇魔法。

所謂「系統程式設計」，需要知道許多硬體細節，以及熟用 assembly 語言。我嘗試將你閱讀本書時所需的基礎做最小化精簡處理。讀者之前，你應該已經完全透徹瞭解了你打算工作的範圍，像是你手上那塊 serial board 的某些奇行怪癖啦，或是你嘗試強化的那個應用程式的行爲啦。此外，你應該能夠毫無困難地閱讀 Intel x86 assembly 語言 -- 雖然你並不一定要寫那樣的程式。你應該能夠流暢使用 C 和 C++ 語言。本書大部份程式實例使用 C 和 C++ 語言。

本書組織

這本書有四篇。第一篇從第 1 章到第 5 章，呈現 Windows 95 系統架構的概觀，以及 Windows 95 系統程式設計的一些決定性觀念。

- 第 1 章（本章）描述整本書的架構，並告訴你如何找到自己面對此書（及書附光碟）的最佳方法。
- 第 2 章談及 Windows 95 在 Microsoft 各作業系統中的角色定位，並解釋各種裝置驅動程式的角色。
- 第 3 章以一個系統程式員的眼光，對 Windows 95 架構做一個提要。
- 第 4 章描述用來和「Virtual Machine Manager 及 Windows 95 其他核心元件」交談並互動的程式介面（programming interface）。
- 第 5 章是對 Windows 95 硬體平台（也就是 Intel 32 位元 CPU）的特性摘要。這一章假設你已經對 Intel CPU 的 16 位元程式設計有了一些基礎。如果你沒有，你可以跳過這一章，因爲本書其餘部份並不需要這一章做基礎。

本書第二篇涵蓋 VxD 程式設計基礎。不論你要處理的是什麼樣的系統程式設計專案，你都必須熟悉這一部份。

- 第 6 章描述 Virtual Machine Manager 如何管理記憶體和 CPU 時間。

- 第 7 章詳細解釋如何以 assembly、C、C++ 語言建立一個 VxD。Microsoft 和協力廠商的工具都已經達到某種程度的整合性，讓你輕鬆地以你所選擇的語言來完成一個 VxD。本章觸到了問題的核心。
- 第 8 章討論 VxD 的載入和初始化。「只是在 CONFIG.SYS 中加上一個 device= 敘述句，然後讓 MS-DOS 處理一切」的日子已經過去了。讀完本章你可以學到如何靜態或動態載入一個 VxD，以及如何將它初始化。
- 第 9 章內含一缸子程式設計技術，對於 VxD 程式員甚有補益。這一章的主題包括：VxD 如何呼叫另一個 VxD、如何被另一個 VxD 呼叫、如何配置記憶體、如何提供 asynchronous callbacks、如何面對其他的驅動程式和應用程式做同步執行（synchronous execution）。
- 第 10 章介紹如何將硬體和軟體虛擬化（因爲在虛擬機器中執行軟體，有其利益存在）。在前一版 Windows 中，「虛擬化」是你撰寫 VxD 的主要原因，此技術至今仍然重要。

有了這些基礎，本書第三篇討論 input/output programming。這些章節是硬體驅動程式撰寫人員最感興趣的部份：

- 第 11 章和第 12 章合併討論了 Windows 95 的 Configuration Manager 和 Plug and Play 架構。閱讀這兩章可以學到 Windows 95 如何辨識硬體、如何指定資源、如何載入並初始化驅動程式。
- 第 13 章說明當驅動程式「被載入、辨識中斷種類、辨識 DMA 通道（direct memory access channels）、辨識硬體所使用的其他資源」時，驅動程式應該配合硬體做些什麼事情。這一章教你如何處理硬體中斷、如何 "trap" I/O port 以管理所謂的資源搶奪、如何模擬裝置、如何處理 memory-mapped hardware、如何處理 DMA。
- 第 14 章敘述通訊驅動程式。它們和 Microsoft 的 VCOMM 驅動程式共同合作，處理 serial ports 和 parallel ports。這一章提供一個以 C++ 完成的 serial port 驅動程式擴充實例，示範如何精巧地製作屬於自己的訂製型 port driver。
- 第 15 章討論 Input/Output Supervisor（IOS），它負責處理磁碟及其他 block devices。這一章描述一個 RAM-disk 驅動程式和一個 IOS request monitor（IOS

需求監視程式），這正是大部份人可能需要撰寫的兩種 block device 驅動程式。

最後，本書第四篇討論 Windows 95 的一些額外性質，允許你擴充系統機能：

- 第 16 章涵蓋一個「可安裝檔案系統」管理程式（Installable File System Manager），這正是稍早我所說的那些 INT 21h calls 的終極標的。本章說明如何撰寫驅動程式以便在本機（local）磁碟裝置上支援一個新的檔案系統。
- 第 17 章，本書最後一章，描述 DOS Protected Mode Interface（DPMI）。雖然新程式完全不需要 DPMI，但那些古老的傳統程式，以及允許被移植到 Windows 3.1 的那些 16 位元應用程式，可能需要 DPMI。

本書程式實例

程式實例和文字敘述並不是一體兩面，這就是爲什麼你在書中很少看到程式完整列表的原因。我嘗試使用程式片段來解釋一般性的要點。書中之所以有許多程式片段，是因爲面對一本電腦書籍時，讀者常以電腦語言來思考問題。

我沒有放進一大堆程式列表的另一個原因是...坦白說...我認爲已經有太多書籍屬於那種「這是我的碼，歡迎你來看」的風格。你知道我說的這種書：作者多半是位程式員而不是位作家，寫了一些自認爲漂亮的程式，於是寫一本書把程式碼重印一次，讓大家分享。唔，我也是個程式員，我也常常認爲我的碼靈巧漂亮，但是我想你比較希望自己寫一個程式，而不只是讚嘆我的。

儘管如此，如果你沒有在一大堆文字之外還獲得一大堆程式碼的話，我想你還是會覺得受騙。這就是爲什麼本書附了一張光碟的原因。光碟內含一些可以有效運作的程式，以及許多實例，它們或者只在我的電腦環境下才能編譯，或者只在我的電腦環境下才能執行。光碟中有一個 **README** 檔案，告訴你如何在你的電腦環境下做實驗。這些實例適用於 Microsoft MASM 6.11 （Windows 95 DDK 中的一個特殊版本）和 Microsoft Visual

C++ 4.0。大部份範例使用 DDK，少部份使用 Vireo Software 公司的 VToolsD。這些範例程式的編譯及測試環境是：Pentium 90，64MB RAM，一個 ISA bus 和一個 PCI local bus，一部 3.5" 軟碟機。

光碟中另有來自 Windows 95 DDK 的「Microsoft 官方 WinHelp 檔」，以及我自己的註解檔。這些註解源自我自己的觀察以及 CompuServe 上的來信。如果你也想在下一版中爲本書略盡棉薄，我的 email 位址是 waltoney@oneysoft.com。

注意 本書「前言」談了更多有關於本書光碟的消息。

關於所有的未公開技巧（Undocumented Hacks）

關於所有的公開技巧（Documented Hacks）

我刻意讓前一節空白，是爲了突顯一個要點：這不是一本講敘未公開秘密的書籍，這本書使用 Microsoft 希望你知道的系統介面，來幫你完成任務。這些介面統統都是公開的。我唯一一次提到反組譯（disassembly）或「未公開之結構欄位」，是在當我說明 control blocks（控制區塊），而又沒有其他方法可以解釋如何使用一個「我判斷一定可以使用的介面」的時候。

不要以爲因爲這本書是 Microsoft Press 出版，所以我就亦步亦趨地踏著 Microsoft 畫好的線。不，我希望每一個人最好都能夠爲這條線塗上個人色彩。但是你要知道，Microsoft 有責任與義務在 Windows 演化過程中讓公開介面的機能保持不墜。此外，你我閱讀的公開技術文件同時也是 Microsoft 教導自己的工程師開發這個系統時所閱讀的文件。如果 Windows 95 深度依賴某一性質，例如「Virtual Communication Device 對於 serial ports 資源搶奪的管理」，那麼你可以確定這個性質一定歷久不衰，十分穩定。然而如果你依

賴那些曲折離奇的手法來做你的買賣（就像呼叫 KERNEL32 中的 *VxDCall*，完全不顧慮 Microsoft 已經不希望你這麼做），可以預期的是，當那些介面無預警地改變了，你將嚐到苦果 -- 特別是如果有某個公開技術可以獲得相同結果的話。

我不喜歡依賴未公開技術的另一項原因是，我不喜歡藉著探索和錯誤嘗試來發掘事物的真象。當我嘗試以一個不適當的方法來解決問題，我可以輕易嗅出障礙和極度複雜的味道，那是一種警訊。所以如果（舉例來說）我發現我沒辦法在我的驅動程式中輕易產生一個隱藏的虛擬機器(hidden VM)，俾能執行一個真實模式驅動程式以便完成某些任務，那麼我就會問我自己，是否有比較保險的作法。

但是，當然，VxD 程式設計的每一個面象似乎都充滿了障礙和複雜。我希望這本書能夠清除經年困擾 Windows 系統程式員的許多迷惑。我希望這本書使你不必虛擲青春。

第 2 章

各式各樣的 Windows 和 Drivers

做爲一個資深的系統程式員，我愛極了沉浸在「讓程式得以有效運作」的實用細節中。我發現，如果我能夠清楚瞭解我的系統如何和作業系統的體制相融 -- 不管是過去的或現在的作業系統，那麼我將獲得最大效率。，我要在這一章檢視一些你或許已經知道的東西。

在 Microsoft 企圖「將桌上型電腦作業系統大一統至 Windows NT」的主路線中，Windows 95 是一條分支。今天，各個 Windows（而非 Windows NT）版本上的系統程式設計份量不斷增加。也就是說，32 位元、ring0 模組的所謂 virtual device driver（VxD）份量不斷增加。VxDs 最初只是爲了替 ring3 模組（如 MS-DOS drivers 和 Windows DLL drivers）將硬體虛擬化，但今天在 Windows 95 之中，VxDs 卻變成一個與硬體溝通的主要介面，同時也提供多樣的軟體服務給各種應用程式。

在我的觀念裡，Windows 3.0 是 Windows 家族中第一個值得嚴肅面對，值得考慮做爲堅實之應用程式發展平台的版本。Windows 3.0 在一段長時間並且廣泛的 beta 測試（當然

是秘密進行）之後，於 1990 年開始了她的處女秀。那時候的 Intel 80386 還是桌上型電腦的高檔產品，而 Windows 3.0 enhanced mode（386 機器上才有）則時有異常情況發生。即使在 80386 機器上還是以 standard mode 跑起來最穩。Windows 3.0 還提供 real mode，支援「髮蒼蒼視茫茫」的 Intel 8086 CPU。終於，Windows 3.1 停掉了對 real mode 的支援，沒有人驚訝，也沒有人遺憾。Windows 95 進一步停掉了對 standard mode 的支援。今天 Windows 95 只能夠在 enhanced mode 上跑，也就是說你必須有一台 80386+ 電腦。

眞實模式 這是 Intel 80x86 CPU 的預設模式，一次只能執行一個應用程式。應用程式在此模式中可以自由使用任何系統記憶體和輸入/輸出裝置。

Windows 家族分類

Windows 95 佔據了作業系統生態中的一個特殊領域。Windows 本身以各種層次的功能，橫跨多種 CPU 設計：Windows NT 瞄準高階市場，Windows 3.x 瞄準 80386 系統，Windows 95 則瞄準今天的中階桌上系統 -- 也就是一部配備大約 8MB 記憶體的 Intel i486 機器。不過 Microsoft 希望你相信，80386 機器配備 4MB 記憶體也可以跑 Windows 95。

Windows NT

你所能買到的 Intel-based 桌上型電腦作業系統當中，Windows NT 是威力最強大的一種。但是你需要很高檔的硬體設備，才跑得動它（我並不想捲入「OS/2 是否成爲未來浪潮」的論戰中，不過以我個人觀點，我不認爲如此！）Windows NT 在數個項目上，把它自己和其他 Windows 產品做了明顯的區隔。第一點並且也是最重要的一點：NT 是個安全防護系統（secure system），只要你把你的硬碟放在 Windows NT native file system 的看管之下，它就可以阻止未經同意的操作（不論是對程式或資料）。Windows NT 同時也是唯一可以跨硬體平台的 Microsoft 作業系統。

Windows NT 導入了 32 位元 Win32 程式介面（Application Programming Interface，API）。只要程式員稍加留心，以此 API 所完成的程式，只需經過簡單的重新編譯動作便可以在任何 Windows NT 所支援的硬體平台上跑。Win32 API 支援「圖形視窗模式」和「文字模式」兩種應用程式，後者常被 end user 稱爲「不進入圖形模式的 MS-DOS 應用程式」。Win32 API 與其他 Windows APIs 的關鍵差別在於：它是爲 32 位元程式而設計的。

注意　當然啦，所謂 API 是一個抽象觀念，技術上它和任何具體實作品沒有關係。因此如果你有才又有閒，倒是可以設計 Win32 的一個 16 位元實作品（真是瘋狂）。

Win32 API 愈來愈豐富而且多樣化。1993 年 Microsoft Press 出版了兩本 ***Microsoft Win32 Programmer's Reference***，共 85 章，每一章致力說明相同類型的一組 APIs。其中有 27 章的標題是 "System Services"，內容包括虛擬記憶體的管理、如何產生並管理 processes 和 threads、四種不同的同步元件（synchronization primitives）、三種不同的檔案系統、至少三種跨越網路的 IPC（Interprocess Communication）方法...等等等。

Windows NT 第一次面世是在五年前，如今已慢慢流行起來。兩個因素造成這種與Ａ級作業系統迥異的「慢慢流行」現象。第一個因素是效率。Windows NT 3.5 比初版的 NT 3.1 效率好得多，但你需要很好的硬體才能得此高效率。你知道，昨日的高檔貨就是今日的入門產品、明日的古董！因此，效率因素會因時間而衰減。

第二個因素是，沒有太多 32 位元應用軟體需要 Windows NT。大部份 Windows 應用軟體還停留在 16 位元，而 Windows NT 對於這些軟體，坦白說不會跑得比 Windows 3.1 更好。Microsoft 於是發明了跛腳的 Win32s 子系統，用於 Windows 3.1 上，引誘廠商開發 32 位元應用軟體，可以同時在 Windows 3.1 和 Windows NT 上執行。諸多限制再加上不相容性，以及一大堆臭蟲，使得 Win32s 並未得到廠商的青睞。由於缺乏 32 位元應用軟體的的大型市場，廠商並不情願開發任何 32 位元程式。

Windows 3.1

在功能上，Windows 3.1 的位階低於 Windows NT。對於相對低檔的 80386 機器而言，Windows 3.1 其實是個 32 位元作業系統，執行 16 位元應用程式、MS-DOS 應用程式、extented DOS 應用程式。

Extented DOS 應用程式 這是一種 16 位元或 32 位元程式，建立在 DOS extender 之上，使它得以在 CPU 保護模式中執行，卻仍然使用 MS-DOS 和 BIOS 的 system services。以此觀點而言，Windows 本身（我指的是 KERNEL、USER、GDI 模組，而非作業系統本身）就是一個 16 位元 extended DOS 應用程式。

保護模式 Intel 80286+ 提供的一種操作模式，支援多工、資料保護、虛擬記憶體。

Windows 3.1 應用程式使用 16 位元 API 來和視窗系統打交道，並使用軟體中斷來和 MS-DOS 以及 BIOS 打交道。事實上，16 位元 Windows 程式用了許多和傳統 MS-DOS 程式所用相同的 runtime library。管理 Windows 3.1 應用程式的那個 32 位元作業系統，主要作用是接收保護模式所發生的軟體中斷，妥善地將它們轉到真實模式的 MS-DOS 和 BIOS 去。

在 Windows 3.1 上進行所謂的系統程式設計，意味著和 MS-DOS 驅動程式、常駐程式（TSR）、以及 DPMI（DOS Protected Mode Interface）打交道。少數冒險家則設計 VxDs 來支援高層次系統程式設計的需求。

Windows 3.1 存在許多問題。最可惱的大概就是 Windows 程式必須週期性地將控制權釋回給視窗系統，以便完成所謂的「合作型多工」（cooperative multitasking）。這種情況下一個設計不良的應用程式可能造成系統失速失控。某些種類的應用程式，特別是重度運算或依賴行程通訊（IPC，interprocess communication）者，很難完美遵循「釋放控制權」

的遊戲規則。Windows 3.1 的另一個問題是核心元件如 USER 和 GDI 的 heap 空間有限，許多使用者抱怨說因爲這個有限空間導至他們沒辦法執行更多的程式。

在系統層面，Windows 3.1 不斷受到「MS-DOS 無法重進入（reentrant）」這個事實的衝擊，以及「MS-DOS 只能在真實模式和 Virtual 86（V86）模式執行並只能夠存取 1MB 記憶體」的影響。這些事實都很重要，因爲 Windows 3.1 仰賴 MS-DOS 和 BIOS 來接觸磁碟驅動程式和檔案系統。真實模式驅動程式的束縛包含：

- 除非 end user 設定硬碟中的一個區域爲永久置換檔（permanent swap file），否則 Windows 3.1 需要一個貫穿 MS-DOS 或 BIOS 的單一執行緒（thread）來執行 paging I/O。
- 需有 MSCDEX 常駐程式的幫忙，才能夠存取 CD-ROMs，但 MSCDEX 可能干擾某些點對點網路軟體（peer networking software）。
- Peer networking servers 會干擾 Windows 處理磁碟驅動程式的能力。Server 端沒辦法使用永久置換檔（permanent swap file）以及因永久置換檔而獲得的 "smart paging" 能力。
- 檔案系統的 I/O 動作要求 CPU 從保護模式切換到 V86 模式，並在最低的 1MB 位址空間中配置一塊緩衝區。
- 由於真實模式驅動程式的普及（特別是網路連接管理程式），使得最低 1MB 位址空間常常客滿，無法執行更多程式。

可重入碼（reentrant code） 代表那些可被中斷然後被另一個執行緒（thread）執行的程式碼。

Windows 95

Windows 95 解決了 Windows 3.1 所衍生的許多問題，並對於使用介面的親和性做出了獨到的貢獻。在 end user 這一端，Windows 95 加了許多方便特性，包括長檔名和可移動工作列（dockable taskbar），以及一個較具整合性並以文件爲中心（所謂 document-centric）

的 shell。雖然它架構在和 Windows 3.1 相同的虛擬機器技術基礎上，但 Windows 95 有 32 位元檔案系統、32 位元磁碟驅動程式、32 位元網路介面。它支援 Win32 程式之間的插斷式多工（preemptive multitasking），使用和 Windows NT 一樣的 process/thread 模型。它的 Plug and Play 子系統使 end user 能夠輕易加上新的硬體週邊。

Windows 95 的一項驚人性質是它對於老舊軟體的包容性。我很喜愛的一部卡通可以把這一點表達得很傳神。這部卡通的第一格，美國陸戰隊二兵 Sack 正在閱讀連隊佈告欄，那幾乎被密密麻麻的備忘錄給塞滿；下一格，我們看到 Sack 從層層疊疊（把陽光都遮住了）的備忘錄中翻出一張紙，顯然這個連隊佈告欄從來沒有撕下過什麼東西；最後一格（圖 **2-1**），Sack 點燃一根火柴，讀到了喬治華盛頓將軍於 1776 年下達的橫渡德拉威河的命令[1]。

圖 2-1 Windows 95 的相容性(美國陸戰隊二兵 Sack 正在閱讀連隊佈告欄)

[1] 取材自 George Baker 的 *"The Bulletin Board", in The Sad Sack*（Simon & Schuster 出版），並獲得授權。

唔，Windows 95 就像這樣。橫渡德拉威河，就軟體而言，有點像 Program Segment Prefix（PSP）資料區，那是 MS-DOS 用來區分一個程式（最重要的是一組檔案 handles）和另一個程式的線索。是的，即使 Windows 95 的 Win32 程式也還含有一個 PSP，而且位在最低 1MB 位址空間，因爲那裡才是 MS-DOS 可以存取到的地方[2]。任何 MS-DOS 或 BIOS 或 Windows 的孑遺都必須可以繼續在 Windows 95 中存活得好好的。舉幾個例子：

- 如果能夠，Windows 95 將使用全新的 32 位元驅動程式來處理所有的網路工作。但如果你堅持，它也可以使用原有的真實模式網路驅動程式。
- Windows 95 將使用全新的 32 位元驅動程式來處理所有硬體，必要時才使用原先的真實模式驅動程式。爲了舉證，我在我的系統中放置一顆 CD-ROM 和一顆硬碟，並且不爲它們安裝真實模式驅動程式。在我安裝 Windows 95 之前，我沒辦法存取它們。如今我開始使用 32 位元驅動程式，但我從來沒有明白要求過它們。是的，Windows 95 Setup 程式偵測了我的硬體，自動調整我的系統組態（configuration），以配合該硬體。
- 16 位元 Windows 程式有時候會仰賴前一版 Windows 的「記憶體共享」特性，以及合作型多工特性，所以 Windows 95 本質上是以和 Windows 3.1 相同的方法來執行 16 位元程式。
- 許多重量級商業軟體以不正確的方式來偵測 Windows 版本號碼，於是 Windows 95 只好欺騙它們，使它們得以繼續存活。

所有這些相容性對於我們系統程式員都會造成不小的問題。你可以輕易寫出一個 VxD 型式的 32 位元作業系統擴充元件，但是你很難讓它面對 real-mode MS-DOS、extended DOS、16-bit Windows，32-bit Windows 等等應用程式都還能夠虎虎生風。舉個例子，如果你要支援一個磁碟設備，那麼供應一層驅動程式，靈巧地吻合 Input/Output Supervisor 架構，應該是足夠了。但是一般而言，關於硬體支援，有時候你還需要誘捕（trap）並模

[2] Andrew Schulman，*Unauthorized Windows 95 : A Developer's Guide to Exploring the Foundations of Windows "Chicago"*，IDG Books，1995。

擬來自 V86 程式的存取動作，而這就比較複雜些！

Windows 驅動程式的沿革

系統程式員的各項任務之中，有許多是「爲特別的硬體撰寫驅動程式」。一般而言，我認爲這樣的工作在 Windows 95 要比在前一版 Windows 容易得多。先瞭解一些歷史沿革，可能會對驅動程式的撰寫有所幫助。

Real-Mode Windows

從一開始，MS-DOS 和 system BIOS 就已經提供了許多硬體驅動程式。BIOS 經由一些著名的軟體中斷，開放出驅動程式的 services，像 INT 10h 之於視訊系統，INT 13h 之於磁碟子系統，INT 16h 之於鍵盤等等。BIOS 也處理硬體中斷，並承擔對於「可程式中斷控制器」（Programmable Interrupt Controller，PIC）的管理責任。MS-DOS 也經由軟體中斷（如 INT 21h、INT 25h、INT 26h）開放了 system services，並提供一個機制（CONFIG.SYS 中的 device= 敘述句），讓新的或強化後的驅動程式能夠在系統啓動時被載入。Ralf Brown 和 Jim Kyle 的 ***PC Interrupts*** 一書（Addison Wesley，1994）含有 MS-DOS 和 BIOS 介面的完整討論。此書對許多 PC 程式員而言應該不陌生。

Standard-Mode Windows

早期的 Windows 中，MS-DOS 和 BIOS 佔最高重要性。Windows 原本只是個真實模式「作業環境」，提供一個架構在 MS-DOS 之上的圖形外套而已。從系統層面看，Windows 只不過是個大的圖形應用程式。Intel 80286 的來臨，使 Windows 能夠在保護模式中跑並獲得高達 16MB 實際記憶體。藉由保護模式的切入和切出，Windows 仍然繼續使用 MS-DOS 和 BIOS 來完成所有的系統需求。這種運作模式被稱爲 Windows standard mode。

在 80286 機器上切換真實模式和保護模式，代價高昂。Intel 於是提供了一個快又有效率的指令，讓你從真實模式切進保護模式。但 Intel 卻以爲沒有什麼人還要再從保護模式切

回真實模式。結果，唯一能夠讓保護模式程式（如 Windows standard mode）取用真實模式軟體（如 MS-DOS）的方法就是 "reset" CPU。在人們發展出來的各種 "reset" 方法中，最普遍的一種就是觸發鍵盤控制器，供應「原本應該由 Ctrl-Alt-Delete 所發出」的外部訊號。於是引發所謂的 "triple fault"，這是 CPU 先天無法處理的一種 "fault"。事實上無論哪一種作法，代價都很昂貴，因爲它們至少都得繞經 BIOS 的 bootstrap code。事實上，在某些 286 機器，模式的切換要花掉好幾毫秒（milliseconds）。

顯然 Windows 需要一種方法，避免每次一有事件（event）發生，像是鍵盤被按下或滑鼠移動等等，就得切回真實模式。解決方法就是寫一個保護模式驅動程式，可以在保護模式中處理 I/O 中斷。這些驅動程式直到今天都還和我們長相左右，你在 SYSTEM 子目錄看到副檔名爲 .DRV 的檔案都是！包括 MOUSE.DRV、COMM.DRV 等等。我把它們稱爲 ring3 DLL 驅動程式，因爲它們骨子裡都是 16 位元 Windows 動態聯結函式庫（DLLs），在 ring3（Intel CPU 最不受保護的階層）執行。它們的任務是在不離開 CPU 保護模式的前提下，和 Windows KERNEL、USER、GDI 模組之間形成介面。

Enhanced-Mode Windows

Intel 80386 CPU 使 Windows 的第三種操作模式（所謂的 enhanced mode）成爲可能。在此模式中 Windows 採用 paging 和 V86 特性，創造出虛擬機器（Virtual Machines，VMs）。對一個應用程式而言，VM 就像一部一般的個人電腦，擁有自己的鍵盤、滑鼠、顯示幕...。而實際上，經由所謂的虛擬化（virtualization），數個 VMs 共享相同硬體。對 end user 而言，最大的好處是他現在能夠在視窗之中（而非全螢幕）跑 MS-DOS sessions。

「虛擬化」是 VxDs 的工作。VxD 的名稱來自於 "virtual x device"，意思是此驅動程式用來虛擬化某個（x）device。例如：VKD 用來虛擬化鍵盤，使 Windows 和任何一個 MS-DOS sessions 都自認爲擁有屬於自己的鍵盤。VMD 用來虛擬化滑鼠。某些 VxDs 並不是爲了虛擬化某些硬體，而是爲了提供各種低階系統服務。PAGESWAP 和 PAGEFILE 就屬於這種 "deviceless" VxD，它們共同管理置換檔（swap file），使 enhanced-mode

Windows 得以將磁碟空間收納爲虛擬記憶體的一部份。

儘管基礎技術令人耳目一新，enhanced-mode Windows 還是繼續在磁碟和檔案 I/O 方面使用 MS-DOS 和 BIOS。需要置換（swap）一個檔案時，它還是把 CPU 切換到 V86 模式，讓 MS-DOS 和 BIOS 來處理 I/O 動作。

在保護模式、真實模式、V86 模式之間的所有切換動作都使得 Windows 慢下來。更多的延宕則來自於「MS-DOS 和 BIOS 不可重進入」這一事實。Windows 必須強迫所有應用程式在同一條隊伍等待真實模式服務。「只有一個隊伍」，使得 overlap paging 或 file system I/O 不切實際。

Windows 95

Windows 95 將終結這一份對歷史的回憶。Windows 95 使用數種不同的驅動程式模型，大部份是使用 32 位元 ring0 VxDs，而非 rin3 DLL。Windows 95 對於 device 的處理，一般的模型是：由一個 VxD 掌管所有中斷並執行所有資料傳輸，應用程式則使用 function calls 的方式對 VxDs 發出需求。

這種「VxD 導向」的 device 規劃模型的一個好例子就是：Windows 95 的 serial communications。從前 Windows 通訊是使用一個 ring3 驅動程式（COMM.DRV），內含硬體中斷處理常式以及「驅動一個 universal asynchronous receiver-transmitter（UART）晶片」所需的全部邏輯。在未讓此驅動程式知曉的情況下，兩個 VxDs（VCD 和 COMBUFF）攔截了硬體中斷和軟體 IN/OUT 指令，爲的是虛擬化每一個 port，並且改善因多工而引起的問題。Windows 95 也有一個 ring3 元件名爲 COMM.DRV，但這個元件已經成爲新的 VxD（VCOMM）的一個薄薄外層，只用來提供 16 位元程式和 VCOMM 之間的介面。VCOMM 穩坐網中央，這個網附著到一般應用程式、附著到 VxD clients、附著到 VxD port 驅動程式。Port 驅動程式現在負責處理所有中斷，並執行真正與硬體交談的 IN/OUT 指令。

Windows 95 檔案系統是另一個好例子。過去，對檔案系統的請求（requests），源自於

16 位元保護模式程式所發出的 INT 21h。有一個 VxD 用來處理這些 INT 21h，並將它們繞轉到 V86 模式，以便 MS-DOS 處理。MS-DOS 發出 INT 13h，以求使用 BIOS 的硬碟 I/O 功能；發出 INT 2Fh，允許網路的 "redirector modules"（重新定線模組）將此 request 透過網路傳輸出去。Windows 95 提供給應用程式的，仍是回溯相容的介面，INT 21h 仍舊是導至檔案系統的動作，但是底層基礎大異其趣！

在 Windows 95 之中，一個名爲「可安裝檔案系統」（**Installable File System**，**IFS**）的管理器會處理所有 INT 21h，甚至是源自於 V86 模式者。然後它把控制權交給一個檔案系統驅動程式（**File System Driver**，**FSD**）。有一個 FSD 名爲 VFAT，係針對 MS-DOS 檔案系統（所謂 File Allocation Table，FAT）而設計；有一個 FSD 名爲 CDFSD，可以解析 CD-ROM 格式；此外還有其他 FSDs，知道如何經由各種網路彼此通訊。針對本機（local 端）FSD（如 VFAT）的磁碟動作，會流經「被 Input/Output Supervisor（IOS）監視管理」的一堆 VxDs 手中。甚至 V86 模式的 INT 13h calls 最終也是在 IOS 處理。換句話說，真實模式和保護模式所發出的對檔案系統的請求（request），不論是針對近端（local）或遠端（remote）磁碟機，有可能完全（或幾乎完全）由 VxDs 來服務。

Windows 95 這種「以 VxD 爲中心」的驅動程式模型，好處之一是，系統程式員不一定得是 MS-DOS 和 BIOS 的專家，就可以寫驅動程式。那些準備供應系統擴充元件的程式員，也同享這個好處；原本你必須瞭解 DPMI 以及 Windows 核心模組的許多神秘性質或未公開性質，現在只需瞭解 Win32 的 *DeviceIoControl* API 函式，以及那些支援所謂 "alertable waits" 的 Win32 API 即可。這兩個介面可以讓你把 VxD 當做 32 位元應用程式的擴充元件。

儘管 VxD 程式設計是「在系統層面上擴充 Windows 95」的一個明顯重點，Windows 95 還是保留了令人印象深刻的回溯相容能力。DPMI 還是存在（有些 16 位元程式還是需要它），你還是可以跑真實模式的網路驅動程式或檔案系統驅動程式 -- 如果這真是你的選擇。事實上，你往往可以把 Windows 3.1 的一整組硬體設備、網路驅動程式、16 位元應用程式及其必要的 VxDs 整個搬到 Windows 95，不至於遭遇什麼大問題。

第 3 章

Windows 95 系統架構

Windows 95 是一個 32 位元保護模式作業系統，可以在中階配備的 Intel 機器上執行 16 位元和 32 位元應用程式。Windows 95 提供虛擬記憶體（可高達 4GB，視不同機器上的實際記憶體和置換檔空間而定），並支援 Windows 程式和 MS-DOS 程式的插斷式多工（preemptive multitasking）。和 Windows 3.1 一樣，Windows 95 只能夠安裝在 Intel 機器上，不同的是，它只能在 80386+ CPU 上跑，也就是說它只能跑所謂的 "386 enhanced mode"。它和 Windows NT 的重大差別在於，Win95 並不支援安全防護，它不是個 "secure" 作業系統。Windows NT 則是個 "secure" 作業系統，不允許程式或資料未經許可就被動用。

虛擬記憶體　這是一種技術，允許作業系統提供比實際數量還要多的記憶體給應用程式使用，作法是把一部份資料放在硬碟之中。

權級（privileged level）　應用程式在某一個 CPU 權級中執行，因此決定了什麼資料可以被取用，什麼程式碼可以被執行，以及程式可以使用什麼樣的指令。ring0 是 Intel CPU 的最高權級，ring3 是最低權級。

記憶體模型（memory model） 程式對於程式碼和資料的定址方法。在所謂的 flat memory model 中，單一 segment 即映射了一個 process 位址空間所涵蓋的整個虛擬記憶體。16 位元程式常常使用 large memory model，它把虛擬記憶體切割爲一個個的 segments，每個 segment 最大 64KB。

Windows 95 作業系統元件在最高權級（ring0）執行，並使用 32 位元的 flat memory model。應用程式則在最低權級（ring3）執行，可以使用 flat memory model 或傳統上適用於 16 位元程式的任何其他 memory models （large, medium, compact 或 small）。

Windows 應用程式開發人員必須學習 Windows 作業系統的三大模組：KERNEL、USER、GDI。在應用程式的程式設計範圍裡，其實並沒有碰觸到作業系統層面。KERNEL、USER、GDI 三大模組也都是 ring3 應用程式，它們並沒有比「接龍」遊戲（打個比方）有更大的權力。Windows 95（以及先前的 Windows 3.1）的真正作業系統核心是 Virtual Machine Manager（VMM）。

本書有許多地方會詳細談到 VMM 以及它提供給系統程式員使用的 APIs。這一章只提供一個概觀，好讓你瞭解 Windows 95 的架構。VM（虛擬機器）是 Windows 95 內部運作的一個中心觀念。有一個所謂的 system VM（系統虛擬機器）內含所有的 Windows 應用程式，還有其他 VMs 內含 MS-DOS 應用程式。System VM 不只執行 16 位元和 32 位元 Windows 程式，它也藏匿了一個以上的 32 位元執行緒（execution threads），這些 threads 在一個插斷式多工排程器（preemptive multitasking scheduler）的控制之下共享 CPU。MS-DOS VM 之於相容性非常重要，但是 MS-DOS 所扮演的角色已經比過去減輕許多。

Virtual Machines（虛擬機器，VM）

VM 的出現是因爲 Windows 3.0 需要同時支援多個 MS-DOS 程式和 Windows 程式。Windows 程式員知道他們處在一個合作型多工環境下，他們有責任週期性地釋放控制權，使其他程式也有機會執行。Windows 應用程式對於輸入裝置的使用有良好的規範，因爲 Windows 核心組件（KERNEL、USER、GDI）扮演應用程式和驅動程式之間的一個仲介。MS-DOS 程式則眾所周知有比較壞的行爲：它們永遠認爲自己擁有鍵盤、擁有滑鼠、擁有螢幕、擁有 CPU、擁有使用者的注意焦點。使問題更加複雜的是，MS-DOS 本身是個單緒（single threaded）、真實模式、不可重進入（non-reentrant）的系統，不能同時執行多個應用程式。

爲了讓多個程式共享 CPU 和其他硬體資源，Windows 95 決定使用 VMs。VM 是一個完全由軟體虛構出來的東西，以和真實電腦完全相同的方式來回應應用程式所提出的需求。從某種奇怪的角度來看，你可以把一部標準 PC 的架構視爲一套 API。這套 API 的元素包括硬體 I/O 系統，和以中斷爲基礎的 BIOS 和 MS-DOS。Windows 95 常常以它自己的軟體來代理這些傳統的 API 元素，俾便能夠對珍貴的硬體多重發訊。但由於表現出「MS-DOS 程式員已經習慣了幾乎一整個年代」的同一標準介面，Windows 95 成就了史無前例的高度相容性。不只是早期 Windows 程式可以順利在其上執行，幾乎所有 MS-DOS 程式也都一樣可以順利執行。

什麼是虛擬機器（VM）？

在 1960s 年代，IBM 開發出一套名爲 VM/370 的作業系統。VM/370 在不同程式之間提供插斷式多工（preemptive multitasking），作法是在單一實際硬體上模擬多部虛擬機器。典型的 VM/370 session，使用者坐在電訊相連的遠端終端器前，經由控制程式的一個 IPL 命令，模擬真實機器的 "Initial Program Load" 動作，於是一套完整的作業系統被載入虛擬機器中，並開始爲使用者著手建立一個 session。這套模擬系統是如此地完備，系統程式員甚至可以跑 VM/370 的一個虛擬副本，來對新版本除錯。

VM/370 爲 IBM 所解決的問題，很類似 Microsoft 今天爲 Windows 3.0 解決的問題：讓每個應用程式認爲自己獨佔整部機器，以允許多工的存在。虛擬機器的觀念再一次提供了解決方案。只要是在一部虛擬機器中跑，不管應用程式或作業系統，都可以相信它們的確是從真正的鍵盤和滑鼠獲得輸入，並從真正的螢幕上輸出。稍加一點點限制，它們甚至可以認爲自己完全擁有 CPU 和全部記憶體。硬體虛擬化和軟體虛擬化便是其中的關鍵。

虛擬硬體（Virtual Hardware）

如我稍早所說，一個虛擬硬體元件的作用，就像真正的硬體一樣。舉個例子，MS-DOS 程式需要從鍵盤讀取輸入，它可以使用 runtime library 函式如 *fgets* 和 *_kbhit*，但 runtime library 函式實際上是發出軟體中斷 INT 21h 呼叫 MS-DOS，或發出軟體中斷 16h 直接呼叫 BIOS。MS-DOS 有一個鍵盤驅動程式可以經由 INT 16h 和 BIOS 交談。BIOS 使用 port I/O operations（IN 和 OUT 指令）來和鍵盤控制器晶片以及中斷控制器晶片交談，BIOS 並且處理源自鍵盤控制器的硬體中斷，以便截取使用者的個別按鍵動作。圖 **3-1** 說明在這個互動中，應用程式和系統的典型層次分佈。

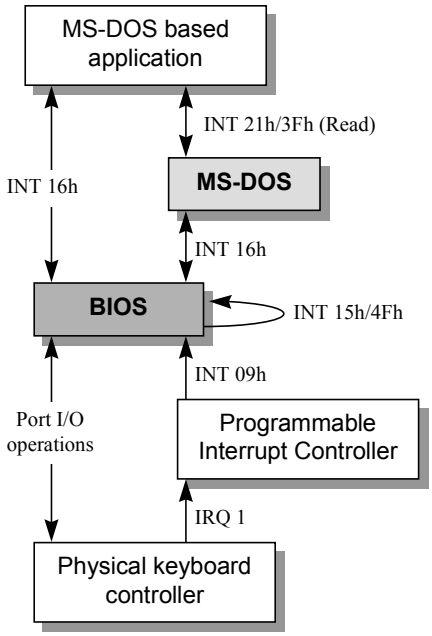


圖 3-1　一個 MS-DOS 應用程式如何從鍵盤中讀取資料

Windows 的最簡單版本（真實模式）係在 MS-DOS 控制下運作，它對鍵盤的使用非常類似前面所說的情況。但是當 Windows 移往保護模式，立刻出現一個問題：保護模式程式沒辦法直接呼叫真實模式的 MS-DOS INT 21h 處理常式，也不能夠直接呼叫真實模式的 BIOS。Microsoft 當然可以寫一個全新的保護模式作業系統，讓所有的鍵盤讀取動作都在保護模式進行，完全不仰賴真實模式的 MS-DOS 和 BIOS。但他們也可以不那麼激進，而是另想辦法，讓保護模式可以觸及真實模式，反之亦然。事實上，Microsoft 兩種方法並行，前一種方法實施於 OS/2 和 Windows NT 之中，後一種作法實施於 Windows 95 和 Windows 3.1 的某幾個運作模式中。當然，最差的是什麼都不做，徒留高檔硬體被 MS-DOS 和 640KB 記憶體綁住！

你可以使用「軟體虛擬化（Software Virtualization）」這個術語來描述「保護模式 Windows 元件」是如何能夠和真實模式 MS-DOS 和 BIOS 彼此互動。「軟體虛擬化」基本上要求作業系統能夠攔截「企圖跨越保護模式和真實模式邊界」的呼叫動作，並且在調整適

當的參數暫存器後，改變 CPU 的操作模式。Virtual Device Drivers（VxDs）就是用來將保護模式的中斷呼叫，透過真實模式中斷向量表（Interrupt Vector Table，IVT），轉換爲真實模式中斷呼叫。做爲轉換程序的一部份，VxD 必須使用置於保護模式延伸記憶體（extended memory）內的參數，產生出適當參數，並將之放在真實模式作業系統可以存取到的地方。此時的真實模式作業系統其實是在 V86 模式（而非真實模式）中運作。服務結束之後，VxDs 再把結果交給延伸記憶體（extended memory）中的保護模式呼叫端。整個過程可以由圖 **3-2** 解釋。這樣的轉換過程將產生出得以在保護模式中執行的「虛擬版」MS-DOS 和 BIOS。

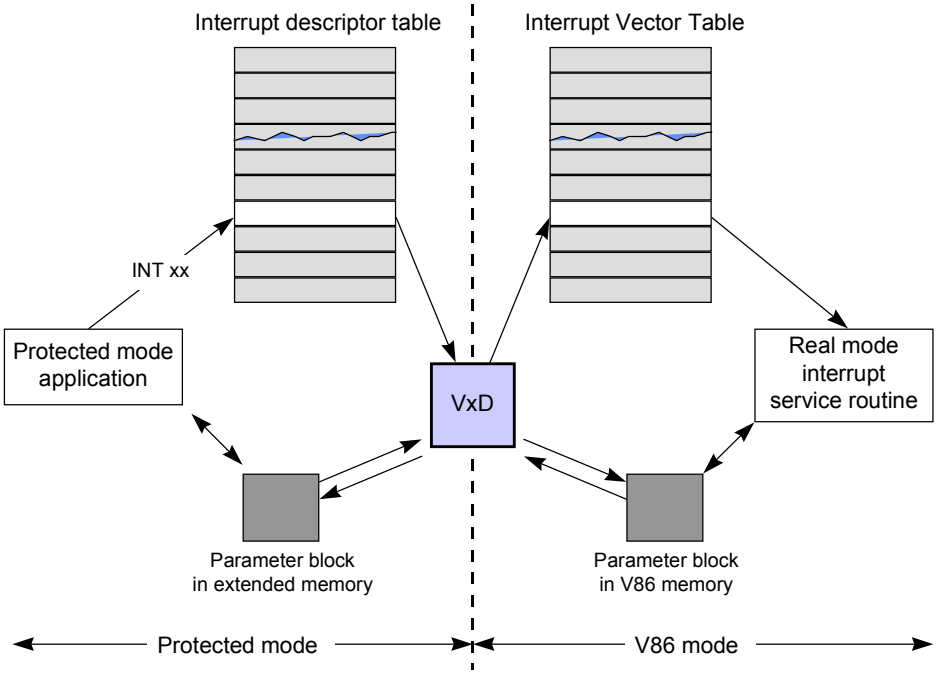


圖 3-2 MS-DOS 和 BIOS 的軟體虛擬化

「硬體虛擬化」又比「軟體虛擬化」的路走得更遠。虛擬硬體的出現是爲了在硬體中斷請求線（hardware interrupt request lines）上產生中斷請求；虛擬硬體的出現是爲了回應 IN 和 OUT 指令、改變特殊的記憶體映射位置等等原因。但是現實可能和外觀不同。**圖 3-3** 解釋兩個 MS-DOS 程式如何同時接觸虛擬鍵盤。一定要注意，這兩個程式執行完全相同的機器指令 -- 從 BIOS 到應用程式都如此，就像**圖 3-1** 單一應用程式的情況一樣。我們只是加了一些 VxDs 以提供必要的硬體虛擬化。由於我們虛擬化的是一個鍵盤和一個可程式中斷控制器（PIC），所以相關的 VxDs 可以稱爲 VKD（Virtual Keyboard Device）和 VPICD（Virtual PIC Device）。一個爲滑鼠設計的 VxD 可以稱爲 VMD（Virtual Mouse Device）。虛擬機器管理器（VMM）自己也是個 VxD。

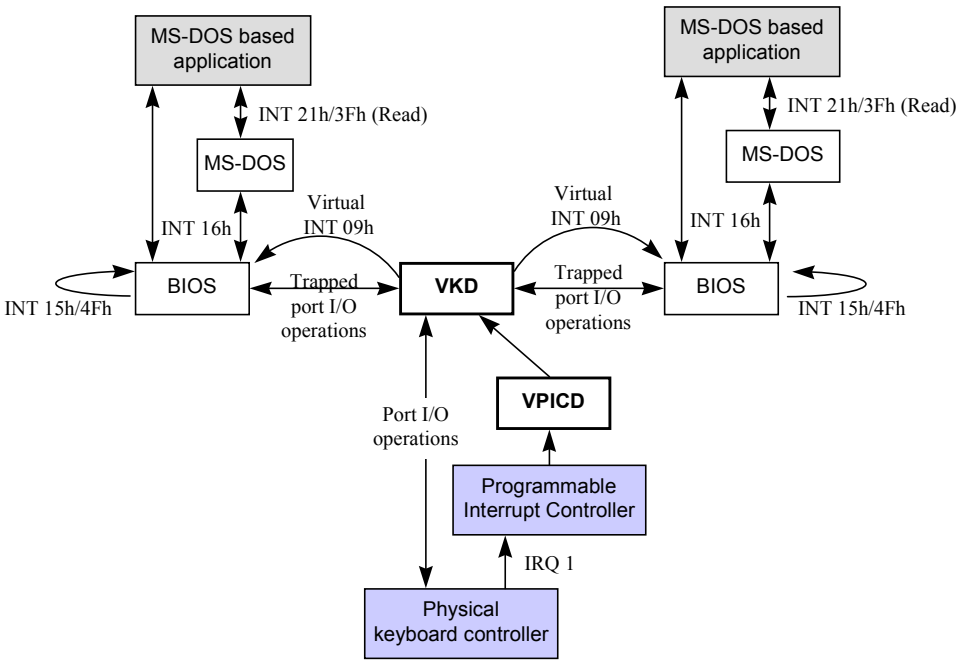


圖 3-3　鍵盤操作動作的硬體虛擬化

「硬體虛擬化」仰賴 Intel 80386+ 的幾個特性。其中一個特性是 I/O permission mask，使作業系統能夠誘捕（trap）對任何一個 port 的所有 IN/OUT 指令。由於硬體中斷的處理動作往往包含對 PIC（Programmable Interrupt Controller）的 I/O 處理，所以 "port trapping" 的能力可使得軟體較易模擬中斷子系統（interruption subsystem）。

> 高雲慶註：IBM PC 相容電腦使用 8259 系列的 PIC（可程式中斷控制器）來規劃硬體中斷向量，所以應用程式可利用規劃 8259 的方式來決定硬體觸發中斷號碼；而在中斷程序之中，亦必須向 8259 發出 EOI（End Of Interrupt）。以上二者都必須針對 PIC 發出 IN/OUT 指令。

另一個特性是：由硬體輔助的 paging 機制，使作業系統能夠提供虛擬記憶體，並攔截對記憶體位置的存取動作（如有必要）。「將 Video RAM 虛擬化」就是「因 paging 而成爲可能」的眾多事情之一。

最後一個必要的特性是 CPU V86 模式，讓 MS-DOS 程式像在真實模式（其實是在保護模式）之中那樣地執行。V86 模式允許作業系統映射一塊 extended memory 到真實模式位址空間中 -- 那是 MS-DOS 和 BIOS 的棲息地，並允許誘捕（trap）I/O 動作、接受並還擲軟硬體中斷。

Virtual Device Drivers（虛擬裝置驅動程式）

VMM 和一大群 VxDs 是硬體虛擬化的關鍵。VxDs 有兩種型態：static VxDs 和 dynamic VxDs。在 Windows 95，VMM 藉由 system registry、SYSTEM.INI 組態檔、以及 INT 2Fh function 1605h 來收集一系列的 static VxDs。當 VMM 安裝「準備用以支援虛擬機器」的架構時，它就載入所有登記有案的 static VxDs，此後那些 VxDs 就一直存在，直到 Windows session 結束爲止。

Windows 95 也允許你動態決定某個 VxD 的載入和卸除。這項能力主要用於 Windows 95

的 Configuration Manager 和 Input/Output Supervisor，兩者都是 static VxDs。靠著 registry 和各式各樣的協定，硬體得以辨識它自己，於是 Configuration Manager 得以選擇並載入與硬體對應的正確 dynamic VxDs。至於 Input/Output Supervisor 用的是比較簡單的體制，根據 VxD 的實際位置，爲硬碟載入適當的驅動程式。

Processes and Threads（行程和執行緒）

Windows 95 使用類似於 Windows NT 所使用的 process 和 thread 觀念。也就是說，每個 Windows 程式佔用一個 process、一個專用的位址空間、和一個（以上）的 threads。每個 thread 擁有許多程式執行步驟，以及與那些步驟相關的暫存器內容和系統物件。Windows 95 使用一個以優先權爲依據的體制來對各個 threads 做插斷式多工（preemptive multitasking）。

在一個以 VMs、processes、threads 爲基礎的完全一般化系統中，每一個 VM 可以內含多個 processes，每一個 process 可以內含多個 threads。然而 Windows 95 並未實現這麼一般化的系統。Windows 95 有一個比較特別的 VM：System VM，用來容納所有的 16 位元和 32 位元 Windows 程式。VMM 總是在 Windows session 開始啓動時就產生唯一的一個 system VM。在其中，每一個 Windows 程式是一個 process，而每一個 process 可以擁有一個以上的 threads。除了 system VM 之外，其他 VMs 都是爲 MS-DOS 程式產生的，其中只內含一個 process 和一個 thread。

Win16 應用程式　一個 16 位元 Windows 應用程式

Win32 應用程式　一個 32 位元 Windows 應用程式

雖然 Windows 95 以上述方式爲 MS-DOS VM 畫出界線，但是 command shell 卻又生出多少有點怪異的一種面貌。你可以在 Windows 95 的 MS-DOS 視窗中啓動一個 Windows 程式，這種啓動方式和在圖形介面中啓動沒有什麼不同。事實上當 Windows 95 在 MS-DOS 視窗啓動一個 GUI 程式，MS-DOS VM 會收回控制權以呈現另一個

command prompt。此外，不管一個 Win32 程式如何被啓動，它可以自由產生新的 threads。這是不是說，MS-DOS 突然長出了一個圖形前端？亦或是 MS-DOS VM 有了一個以上的 processes 而每一個 process 可以擁有一個以上的 threads？

如果你認爲答案是 No，你是對的。但是在搖你的脖子之前，也許你應該聽聽另一個令人迷惑的實驗結果。你可以產生一個所謂的 console mode Win32 程式，它可以使用所有的 Win32 API -- 和圖形介面有關的除外。試著執行一個 multithreading console-mode Win32 程式，例如書附碟片中的 \CHAP03\THREAD\THREAD.EXE：

THREAD.C

```
#0001  #include <windows.h>
#0002  #include <stdio.h>
#0003
#0004  DWORD WINAPI mythread(LPVOID junk)
#0005     {                     // mythread
#0006     puts("Hello from the thread!");
#0007     return 0;
#0008     }                     // mythread
#0009
#0010  int main(int argc, char *argv[])
#0011     {                     // main
#0012     DWORD tid;
#0013     HANDLE hThread;
#0014
#0015     hThread = CreateThread(NULL, 0, mythread, NULL,
#0016     0, &tid);
#0017     if (hThread)
#0018        {                  // thread has been started
#0019        puts("Hello from the main program!");
#0020        WaitForSingleObject(hThread, INFINITE);
#0021        }                  // thread has been started
#0022     return 0;
#0023     }                     // main
```

譯註：下面是 THREAD.EXE 執行結果：

當你在 Windows 95 的 MS-DOS 視窗中執行此程式，它會正確產生出 *mythread* thread。你會看到 MS-DOS 視窗中的輸出，和在 Windows NT 中的情況一樣。程式結束之前，你不會回到 MS-DOS prompt。看起來好像這個 Win32 程式以 MS-DOS 程式的風貌出現，但卻衍生出新的 threads，明顯違反先前我說的 MS-DOS VM 的限制。

實際情況是，command shell 處理這種程式時使用了一些技巧。當你發出一個 MS-DOS 命令，MS-DOS command shell（COMMAND.COM）就發出一個 MS-DOS 中斷（INT 21h, function 4Bh），載入並執行你所指定的檔案。VMM 攔下這個中斷並注意到你欲執行的是一個 Win32 程式，於是在 system VM（而不是你敲入命令的那個 MS-DOS VM）中載入該程式。如果你執行的是一個 GDI 程式，MS-DOS 會立刻重獲控制權；但如果你執行的是一個 console mode 程式，在該程式結束之前，VMM 不會讓 INT 21h, function 4Bh 的處理常式回返。但無論哪一種情況，程式都真的在一個新的 System VM process 中展開其生命。

Windows 應用程式

Windows 程式有 16 位元和 32 位元兩種。Win16 程式使用一套歷經時間考驗過的（也因而有點陳舊的）API。此 API 從 Windows 1.0 演化至 Windows 3.11。Win32 程式則使用一套從 Windows NT 3.1 開始發展的 API。兩者都使用 Intel 80386+ CPU 的保護模式，因而可以獲得 VMM 所提供的所有虛擬記憶體。Win32 程式可以擁有許多 threads，但 Win16 程式只能有一個 thread。Win32 程式在 VMM 的排程器控制之下，享有插斷

式多工（preemptive multitasking）；Win16 程式則只有合作型多工（cooperative multitasking）。

一般而言，Win32 程式跑得比 Win16 程式快，使用記憶體的效率也比較高。這是因爲 32 位元程式使用 flat memory model，所有的 code 和 data 都放在一個「涵蓋所有虛擬記憶體」的 segment 中。16 位元程式則必須不時地把 segment selectors 放進 CPU 的 segment registers 中，才能存取超過 64KB 的記憶體。「載入保護模式 selectors」比單純「載入一個 32 位元 flat 指標」要多花上 7 倍的時間。由於 Windows 三大基礎模組 KERNEL、USER、GDI 佔用的記憶體遠大於 64KB，而它們又是所有應用程式常常要用到的模組，並且由於 16 位元 Windows 的合作型多工模型保證會常常將一個程式切換到另一個程式，因此 selector 的載入動作必然是件常常發生的事情。

Win32 應用程式

32 位元程式不單只是執行得比 16 位元程式好，它們還有其他優點！Win32 API 可以在 Windows NT 能夠執行的所有平台上移植，所以一個設計良好的 Win32 程式只要經過重新編譯和聯結，便可輕易移植到任何這樣的硬體平台。Win32 API 也比 Win16 API 更豐富更合邏輯，程式員可以更容易與系統（及其他程式）交談。舉個例子，Win32 程式可以輕易產生一個 virtual file mappings，把磁碟檔案當做虛擬記憶體來使用。這可以大量簡化某些工作，像是管理一個資料庫或載入一個可執行檔以便執行等等。

只有 Windows NT 才支援完整的 Win32 API。Windows 95 支援的是其中的大部份，缺少 security、event logging、Unicode。不過 Windows 95 也擁有一些 Windows NT 所沒有的部份，主要原因是 Windows 95 係爲一個 16 位元程式壓擠出來的結果。你可以在 ***Programmer's Guide to Microsoft Windows 95***（Microsoft Press，1995）的 Article 3 中獲得這部份細節。

儘管存在上述優點，Win32 程式還是擺了一道難題給 VMM。不管你信不信，VMM 骨子裡相信 System VM 跑的是 16 位元碼。結果，大部份的系統程式介面（用來和 VMM 和 VxDs 交談）只能讓 16 位元程式使用。當 32 位元程式發出一個軟體中斷企圖喚起

那些介面，系統保證立刻當掉。Microsoft 不希望改變底層設計，爲的就是擔心出現太多不可移植的 Win32 程式。

一個想要直接與 MS-DOS 和 BIOS 打交道的 Win32 程式，會因「經由 Win32 API 能送出什麼」而受到限制。當然，API 的威力是十分強大的。應用程式有一組豐富並可自由支配的 API calls，用來處理（以及同步化）各個 processes 和 threads；system registry（控制 Windows 95 的許多狀態）也可以輕易以 Win32 API 來存取或修改；debug API 可以讓應用程式監視或控制另一個應用程式；此外還有用來量測效率的 API。

如果 Win32 API 不夠，應用程式可以利用 *DeviceIoControl*，送出資訊給 VxD，或是從 VxD 接收資訊。舉個例子，只要送 IOCTL 碼給 VWIN32 device，Win32 程式就可以做低階的磁碟 I/O。VxD 自己也可以使用 asynchronous procedure call，將 calls 回授給 Win32 程式。提醒你，應用程式如果依賴此法，恐怕不能在 Windows 95（及其後繼者）以外的任何作業平台上跑。我並不清楚是否用這兩種方法可以比使用傳統介面產生出「更好的」不可移植軟體！不，我只是在解釋什麼可以成功運作時，盡我之責。

Win16 應用程式

雖然 Windows 95 讓 Intel 機器上的 32 位元程式設計明顯比先前任何時候都有吸引力得多，16 位元程式仍然繼續在 Windows 95 中扮演一個重要角色。從相容角度來看，Windows 95 必須支援許多市場上的 Win16 應用軟體。某些重量級 Win16 商業軟體以這種方式來偵測 Windows 版本：

```
BOOL bAtLeast_3_10 = ((WORD)GetVersion() >= 0x0A03);  // 錯誤！
```

GetVersion API 以 low-byte 傳回主版本號碼，以 high-byte 傳回次版本號碼。爲了做比較，一個程式其實應該分開比較主、次版本號碼，或是先將高低 bytes 的次序對調。例如：

```
WORD wVersion = (WORD)GetVersion();
wVersion = (LOBYTE(wVersion) << 8) | HIBYTE(wVersion);
BOOL bAtLeast_3_10 = wVersion >= 0x030A;  // 正確！
```

Microsoft 最初打算以 4.00 做爲 Windows 95 的版本號碼。事實上，在 Windows 95 之中，如果 Win32 程式呼叫 *GetVersion*，將獲得 0x0004，如果 VxD 呼叫 *Get_VMM_Version*，將獲得 4.00 版。但由於 Win16 程式廣爲流行不正確的版本偵測方法，Microsoft 只好讓 16 位元 *GetVersion* 傳回 0x5F03（也就是 3.95 版），而非 0x0004！因此，下面這個程式的執行結果，視其爲 Win16 程式或 Win32 程式而有不同：

GETVER.C

```
#0001  #include <windows.h>
#0002
#0003  int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrev,
#0004     LPSTR lpCmd, int nShow)
#0005     {                       // WinMain
#0006     char msg[128];
#0007     wsprintf(msg, "Windows version is %4.4X", GetVersion());
#0008     MessageBox(GetFocus(), msg, "GetVersion Value",
#0009        MB_OK | MB_ICONINFORMATION);
#0010     return 0;
#0011     }                       // WinMain
```

原始碼及其 Win16 可執行檔和 Win32 可執行檔，分別放在書附碟片的 \CHAP03\GETVER16 和 \CHAP03\GETVER32 目錄中。

譯註：以下是 Win16 版的 GETVER.EXE 執行結果：

以下是 Win32 版的 GETVER.EXE 執行結果：

Windows 95 在管理 Win16 程式記憶體方面，以及在 Win16 程式之間的多工方面，都有良好的回溯相容性。在 Windows 3.x 中，所有 Windows 程式和 MS-DOS 程式共享同一塊 4GB 虛擬位址空間。然而在 Windows 95 中，每一個 Win32 程式擁有一個私有位址空間，從 00400000h 到 80000000h。因此，就像在 Windows NT 之中那樣，除非使用 named file mappings，否則 Win32 程式不能夠分享記憶體內容。但因爲 Win16 程式以及甚至 extended DOS 程式（使用 DPMI 來配置記憶體）彼此一視同仁，所以 Windows 95 把所有 Win16 程式、以 *GlobalAlloc* 配置的記憶體、以 DPMI 配置的記憶體等等，都安排在 80000000h~C0000000h 位址空間中。這個範圍可以被所有 processes 共享。

Win16 程式必須自動釋放控制權，使合作型多工得以實現。某些特殊設計的程式在到達它自己所定義的情況之前，並不釋放控制權。爲了回溯相容，Windows 95 爲 Win16 程式提供了相同的合作型多工機制。事實上，每一個 Win16 程式都是不同的 process 和 thread 的一部份。我們無法察覺這一事實，除非是利用一個系統除錯器（例如 SofeIce/W，它可以讓你觀察 ring0 控制結構）。一個 Win16 thread 只有在它獲得 **Win16Mutex**（一個 mutual exclusion semaphore）的主權之後，才能夠開始執行。利用 mutex 來模擬早期

Windows 的行爲，才能夠在 Windows 95 VMM 的插斷式多工（preemptive multitasking）環境下，一次只讓一個 Win16 程式有排班（scheduling）機會。

混合 16 位元碼和 32 位元碼

由於 Win16 程式和 Win32 程式共處於 system VM 之中，你可能會希望它們之間能夠輕易合作。然而，混合了 16 位元和 32 位元的程式設計模型（所謂 mixed-bitness programming）在 Windows 環境中是有困難的，理由很多！Win16 所使用的 NE（New Executable）檔案格式只能容納 16-bit segments，而 Win32 所使用的 PE（Portable Executable）檔案格式只能容納 32-bit segments。Win32 程式執行時，CS、DS、ES、SS 都指向同一個 32-bit flat segment，定址所用的暫存器和偏移（offset）也都是 32 位元；Win16 定址則是使用 16-bit segment 暫存器和 16-bit 偏移的組合。因此，如果 Win32 程式要成功呼叫 Win16 程式，其間必須有一個轉換層（thunk）。所謂 **thunk**，經過許多動作細節，將程式碼和堆疊定址從 32 位元切換到 16 位元，並把整數從 32 位元切斷爲 16 位元。反方向呼叫（16 位元呼叫 32 位元）亦需要類似的工作。關於 16/32 位元混合，還有許多許多其他複雜的事情，是我沒辦法在這裡提到的

你可以從 ***Programmer's Guide to Microsoft Windows 95*** 中獲得 thunk compiler 的資訊，它可以讓程式員輕易在 Windows 95 中做出自己的 16 位元和 32 位元之間的介面元件。Thunk compiler 比起所謂的 **universal thunk** 可說有了巨大的改善，後者允許「執行於 Windows 3.1 Win32s 子系統中的 Win32 程式」能夠呼叫 Win16 DLLs。Thunk compiler 也比所謂的 **generic thunk** 改善多了，後者允許「執行於 Windows NT 中的 Win16 程式」呼叫 Win32 函式。Universal thunk 和 generic thunk 結構都非常特殊，而且單向；Windows 95 的 thunking 機制則不但強固（robust）而且雙向（不幸的是它和前兩種 thunking 機制並不相容）。

16 位元碼和 32 位元碼之間的 thunking，在 Windows 95 中是如此地水乳交融，Microsoft 於是大量依賴它來執行 Windows 程式。先前我一直談的 KERNEL、USER、GDI，好像每一個元件只有一個模組似地。不然！它們其實各有 16 位元版和 32 位元版，兩版彼

此以 thunks 連接起來。例如，KERNEL32.DLL 會呼叫 KRNL386.EXE 執行某些動作，反向亦然。大部份和視窗管理有關的工作都在 16 位元 USER.EXE 中完成，32 位元 USER32.DLL 僅提供少部份協助。同樣地，32 位元 GDI32.DLL 也分擔了 16 位元 GDI.EXE 的工作。

但是，由於 16 位元系統元件在本質上是從 Windows 3.1 遺留下來的，不能夠「重進入」（reentrant），所以 Windows 95 必須對它們的執行做同步化控制。**Win16Mutex** semaphore 就是爲了這個目的而設計。只要有一個 32 位元系統元件（例如 GDI32）需要呼叫 16 位元模組中的一個函式，就必須先擁有 Win16Mutex。於是，一次只有一個 thread（不管是 16 位元或 32 位元）能夠在 System VM 中執行 16 位元碼。不過，同時可以有一個以上的 threads 執行 32 位元碼。關於 Windows 95 的 KERNEL、USER、GDI 子系統的運作，在 Matt Pietrek 的 ***Windows 95 System Programming Secrets***（IDG Books，1995）一書中有詳細的討論。

MS-DOS 和 MS-DOS 應用程式

Windows 最初只不過是個建立在 MS-DOS 之上的作業「環境」。事實上最早版的 Windows 只是個華麗的 command shell，爲的是比 command prompt 好用些。然而當 Windows 經年累月地成長之後，它包容了曾經由 MS-DOS 所提供的各種作業系統核心機能。早期架構圖（圖 **3-4**）顯示，MS-DOS 和一大堆真實模式驅動程式是最接近硬體的東西。到了 Windows 95，VMM 和一堆 VxDs 才是最接近硬體的東西，MS-DOS 和少量的真實模式驅動程式則成爲 VMM 的客戶（呼叫者）！

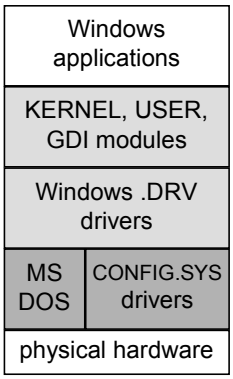


圖 3-4　Windows 95 之前，MS-DOS 所扮演的角色。

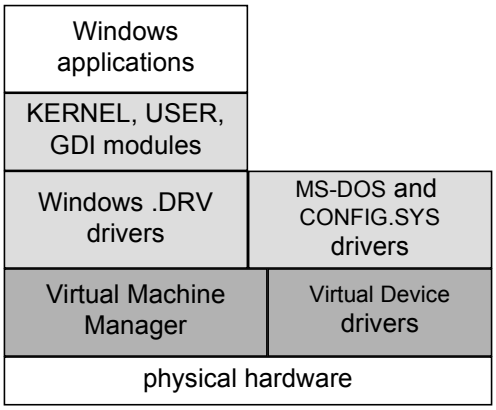


圖 3-5　Windows 95 之中，MS-DOS 所扮演的角色。

從 MS-DOS 到 VMM 的移轉現象，反應出 PC 硬體和軟體市場的演化。過去，桌上型電腦中有一大部份是低階的 8086 和 80286 機種，它們跑 MS-DOS 要比跑 Windows 順暢得多。應用軟體開發者也發現，在 MS-DOS 上發展軟體輕鬆多了，只要再加上一個 DOS extender 和其他商用函式庫，就可以達到記憶體需求和親和性需求。但是今天高階電腦已經多得多，它們跑起 Windows 386 enhanced mode 來好得很，根本沒有必要停留在 MS-DOS。此外，或許來自於 Microsoft 文件和訓練的推波助瀾，也或許來自於 end user

的壓力，應用軟體廠商現在已經持續開發出非常好的圖形導向 Windows 程式。

既然大部份新軟體都瞄準 Windows，既然大部份新硬體都有能力跑 32 位元保護模式，爲什麼還要顧忌 MS-DOS？我猜想唯一的答案是相容性和熟悉度。在我寫這一章的此刻，我在一部極具水準的電腦上跑 Windows 95，但我使用真實模式的網路驅動程式，因爲我的網路商還沒有把他的驅動程式移植到保護模式（雖然 Windows 95 beta 版已經面世兩年之久）。我常常開一個 MS-DOS 視窗，在裡面打一些習慣的 DIR、GREP 等命令。偶爾我也玩一些電腦遊戲，它們許多都只能在 MS-DOS 環境下跑。Microsoft 從 OS/2 和 Windows NT 身上學到的最大教訓或許就是：一個作業系統如果沒有殺手級應用軟體，就不可能縱橫市場。Windows 95 會成功，因爲它比 MS-DOS 和 Windows 3.1 都好，而又能執行前兩者的所有應用程式。Windows 95 也會大量培育 32 位元程式的開發，因爲它是 32 位元程式的執行平台中，第一個擁有巨大市場佔有率者（我不重視 Windows 3.1 上的 Win32s，因爲大部份廠商根本把它視爲一個玩具）。

Windows 95 與 MS-DOS 的共存是從你開機後就開始的。硬碟中的 bootstrap loader 會載入並執行一個 IO.SYS，其中內含 MS-DOS 7.0（至少，那是 DOS function 21h/30h 所記錄的號碼）。你可以像下面這樣測試（版本號碼由 AX 傳回，前半部是次版本號碼，後半部是主版本號碼）：

```
C:\>debug
-a 100
2E01:0100 mov ah,30
2E01:0102 int 21
2E01:0104 int 3
2E01:0105
-g

AX=0007  BX=FF00 ...
DS=2E01  ES=2E01 ...
2E01:0104 CC          INT    3
-q
```

MS-DOS 7.0 處理 CONFIG.SYS 和 AUTOEXEC.BAT，然後自動發出 WIN 命令，執行 WIN.COM 程式（位於 Windows 磁碟目錄中）。它會啓動 Windows 95 的 VMM，於

是接管機器並執行圖形操作環境。有數種方法可以截斷上述程序，落到 MS-DOS 的掌控中。很長一段時間，我使用一個空的 WIN.BAT，放在 WIN.COM 路徑之前。這個空的 WIN.BAT 會在 MS-DOS 自動執行 WIN 命令時獲得執行。你也可以在開機啓動程序中按下 F8，表示你要在 command prompt 之後停止啓動程序。不過呢，現在，我只要在隱藏檔 MSDOS.SYS 中做這樣的設定就可以了：

```
[Options] BootGUI=0
```

你可以在 MS-DOS command prompt（DOS 視窗）中做任何正常的 MS-DOS-like 動作，包括執行一個早版的 Windows -- 如果你還保有的話。我常常利用這個機會來組態（configuring）並啓動除錯器。其實並沒有太多時候需要停在 command prompt（DOS 視窗），除非是像上面所說的那種詭異需求。如果你要在 MS-DOS 中啓動 Windows 95，只要執行 Windows 磁碟目錄中的 WIN.COM 即可。

如果你可以更密切地驗證 VMM 以及 Windows 95 其他部份在圖形介面起而執行之後是如何地使用 MS-DOS，你的腦海才會浮現一個比較有趣的畫面。Andrew Schulman 在其 ***Unauthorized Windows 95***（IDG Books，1994）一書中深度探索了這一主題。在 Andrew 所揭露的一些有趣消息中，有一則是：Windows 95 繼續爲每一個應用程式產生一個 program segment prefix（PSP），並且仰賴 MS-DOS 完成此事。但是在大部份的系統服務範圍內，MS-DOS 已經大大撤退到幕後去了，於是中斷處理和 I/O 動作現在比較可能在 32 位元保護模式中發生。甚至連 MS-DOS 的檔案系統通常也已經由 VxD 完成。

然而，「回溯相容」要求 MS-DOS 驅動程式和常駐程式（TSR）能夠繼續運作，同時也要求 Win16 程式能夠繼續使用軟體中斷，包括 INT 21h，以求取系統服務。VMM 以及與之合作的一堆 VxDs 確保回溯相容的作法就是：將 MS-DOS 虛擬化。換句話說，當任何程式，甚至真實模式的 MS-DOS 應用程式，使用 INT 21h 來要求一個檔案系統服務時，VxDs 會挺身而出執行磁碟 I/O。當一個真實模式驅動程式嘗試對一個它自以爲管控的 port 做 I/O 動作，VxDs 會攔截並指揮那個動作。事實上 Windows 95 是把 MS-DOS 視爲一個應用程式。

第 4 章

系統程式設計之介面
Interfaces for System Programming

我曾經用過的每一套作業系統，都有一組介面，讓應用程式提出系統服務需求（system services request）。我在 IBM OS/360 上鍛鍊了一身好功夫，它控制了一整個房間的 mainframe 電腦，呈現出 60~70 年代的科技極致。OS/360 提供唯一一條通往其 system services 的管道：Supervisor Call（SVC）機器指令，以及一組非常詳盡的 assembly 巨集指令，用來將 request 安插到該管道上。IBM 還供應一套非常完整精確的文件，描述如何使用這個介面。一點也不誇張，我相信一個有經驗的程式員可以透過這一份文件，呼叫他不熟悉的介面，並保證其結果一如預期。如果偶爾有意外情況發生，系統程式員也可以諮詢邏輯手冊或是內含整個系統原始碼的微縮影片。IBM 的另一個作業系統（VM/370）甚至給我們機器可讀的原始碼，系統程式員可以不斷修改以符合當地客戶的需求。IBM 提供有正式而公開的指令，用於上述修改用途。

相對於 OS/360 系統程式員生活中的秩序和精準，Windows 95 系統程式員的生活可以說充滿了混沌和迷惑。雖然 Intel 的 INT 指令在每一個 system service 底層提供服務，但的確有數群 services 有著它們自己的規則。例如，使用 INT 21h 索求檔案系統服務，並

不能帶給你在「使用 INT 31h 索求 DPMI 服務」時什麼經驗。但如果 DPMI 不存在（也就是說在真實模式中），你可以使用 INT 21h 取代之。如果你在 32 位元 Windows 程式中使用前述的共通介面（譯註：INT 指令），程式會當掉。是的，在某些情況下，一般的 INT 21h services 沒有作用，你必須使用 DPMI 來強迫真實模式 MS-DOS 執行其機能（例如大小寫轉換 -- 那是 MS-DOS 跨國語言支援能力的一部份），或是執行不同的機能（例如配置真實模式記憶體）。

如果「數個不同的系統介面」還不夠複雜，那麼，我告訴你，這些介面之說明文件之差勁，足以打敗任何一個系統程式員新手（除非你極有毅力）。以稍早所說的 INT 21h 爲例，它的文件現在是夠詳細的了，但那是因爲有一群勤勉的「逆向」工程師，不斷地尋找並記錄許多細節。要學習 MS-DOS 程式設計，程式員必須購買一些「非官方」出版的書籍，其中充滿經驗的分享和英雄式的反組譯色彩，但那多少帶點「短暫軼聞」的味道。隨便舉個例子，有些時候程式員必須寫碼檢驗 MS-DOS 的 memory blocks chain（當常駐程式要檢查是否它已被載入時，就需這麼做）；MS-DOS 的 memory blocks 結構如今眾所周知，你可以使用 INT 21h, function 52h 來找到整個串列的頭（請參考 Ralf Brown 和 Jim Kyle 所著的 ***PC Interrupts***，p8-45，Addison Wesley 出版，1991）。但是 Microsoft 從來沒有出版任何這方面的相關資訊。是那些奇怪而瘋狂的程式員自己想辦法瞭解了它，並冒著訴訟的風險來出版他們的發現。

當然，這個作業系統的某一部份是公開的、無遺漏的。在這本書中我要大談如何撰寫 Virtual Device Drivers（VxDs）。Microsoft 的 Windows 95 DDK 解釋「如何」呼叫一個被 VxD 使用的 system interfaces，但並沒有解釋「爲什麼」呼叫。它也不解釋副作用，也不解釋怎樣把各個不同的 interfaces 整合在一起形成一個有條理的程式。這和過去我在 OS/360 的經驗完全不同。我在 Windows 上的經驗是：沒有辦法光靠閱讀官方文件就寫出一個有用的 VxD。許多時候我的努力奮鬥掙扎與痛苦，只因爲我錯用或忽略了數個 VxD services。

現在我終於說出這本書的寫作動機了。是的，是時候了，應該開始有人探索如何使用並擴充 Windows 95 作業系統。在這裡，我將對系統程式設計專案中可能使用的各種介面

做一番摘要，其中包括 VxD service API 以及數個定義給特殊種類之 devices（如 serial 和 parallel port drivers、網路檔案系統...）所使用的結構化 APIs。同時也包括一組較爲老舊的 Windows 系統介面：DPMI。Windows 保留 DPMI，純粹是爲了回溯相容。

Virtual Device Drvers（虛擬裝置驅動程式）

Windows 95 作業系統的核心是一群在 VMM 傘下運作的 VxDs。VMM 本身也是一個 VxD。VxDs 之間彼此以三種基本方法溝通：system control messages、service API calls、callback functions。

一個 VxD 也可以輕易開放出 API 進入點，給真實模式和保護模式程式使用。爲了使用這樣的 API，應用程式必須使用 INT 2Fh, function 1684h 獲得函式進入點。當應用程式呼叫此函式進入點，VMM 就會獲得控制權，並將控制權交到 VxD 手上。VxD 於是處理 VM ring3 暫存器的一份影像（image，譯註：即 **client register structure**），回應呼叫端的需求。這樣的機制適用於 16 位元 Windows 程式、16 位元 MS-DOS 程式、在 DOS extender 中執行的 16 位元或 32 位元程式。至於 Win32 程式則是以 *DeviceIoControl* 函式來索求 VxDs services。

所有 VxD service calls，都被記錄在 Windows 95 DDK 文件之中。本書大部份內容將提供「如何」以及「何時」使用本章所摘要的這些機制的相關細節。

System Control Messages（系統控制訊息）

Windows 95 以一種預先定義好的次序，將 **system control messages** 送到所有已載入的 VxDs 手中。每一個 VxD 會開放一個 **device control procedure** 接收這些訊息。這個函式要不就把訊息派送到一個專門的處理常式去，要不就將 carry flag 清乾淨，然後回返，表示成功地處理了這個訊息。目前大約有 50 個定義好的 system control messages，某些用來描述 Windows 95 啓動（startup）和停工（shutdown）時的各種狀態，或是 VM 和 thread 誕生和結束時的各種狀態。其他訊息則關係到對許多 VxDs 有重要意義的

events。

由於常常有許多 VxDs 被載入，而每一個 VxD 會獲得一大堆 system control messages，因此，如何對訊息設限，使它在「許多 VxDs 都感興趣」的情況下才發生，是很重要的。你並不會在 system control messages 中看到像一般應用程式所收到的那麼細微的訊息（諸如 WM_MOUSEMOVE 之類）。

但是 Windows 95 也定義了一些其實只與某單一 VxD 相配的 system control messages，原因是 device control procedure 是 VMM 可對 VxD 設定的唯一入口。不過，現在已有一個新的體制出現，VxDs 可以定義它們自己的 control messages，並將這些訊息往瞭解其意義的 VxDs 送去。

Device Control Procedure 在哪裡？ 就如上文所述，VMM 唯一能夠找到的一個 VxD 進入點就是 device control procedure，所以它被用來處理 system control messages。有一個 static 結構：**device description block**（DDB），其中有欄位指向 device control procedure。DDB 是 VxD 可執行檔唯一匯出（export）的一個符號。

Startup 和 Shutdown 訊息

VMM 藉由表 **4-1** 所列的 system control messages，在 Windows 95 啓動（startup）和停擺（shutdown）之前，溝通重要的 events。

訊息	說明
Sys_Critical_Init	通知 VxD 說這是初始化的第一個階段。此時中斷處於 disabled 狀態
Device_Init	通知 VxD 說這是初始化的第二個階段。此時中斷處於 enabled 狀態
Init_Complete	通知 VxD 說這是初始化的第三個階段。
System_Exit	通知 VxD 說這是停工的第一個階段。此時中斷仍處於 enabled 狀態
System_Exit2	和 System_Exit 相同，但訊息傳送次序和初始化次序相反。

訊息	說明
Sys_Critical_Exit	通知 VxD 說這是停工的第二個階段。此時中斷處於 disabled 狀態。
Sys_Critical_Exit2	和 Sys_Critical_Exit 相同，但訊息傳送次序和初始化次序相反。
Reboot_Processor	指示 VxD 說：如果它知道如何重新開機（reboot），就去做！
Device_Reboot_Notify	通知 VxD 說 VMM 正要重新啓動系統。此時中斷處於 enabled 狀態
Device_Reboot_Notify2	和 Device_Reboot_Notify 相同，但訊息傳送次序和初始化次序相反。
Crit_Reboot_Notify	通知 VxD 說 VMM 正要重新啓動系統。此時中斷處於 disabled 狀態
Crit_Reboot_Notify2	和 Crit_Reboot_Notify 相同，但訊息傳送次序和初始化次序相反。

表 4-1　Startup 和 Shutdown 時候所發生的 system control messages

下面是值得注意的重要事項：

- 有三個 startup 訊息。Sys_Critical_Init 是在 CPU 切換到保護模式之後而中斷被 enabled 之前立刻發生。Device_Init 在中斷被 enabled 之後發生。Init_Complete 發生於「所有 devices 都在 Device_Init 期間初始化」之後。
- 有兩個 shutdown 訊息。System_Exit 相當於 Windows 發出的一個宣告，說它自己將要停工了。Sys_Critical_Exit 則是在中斷被 disabled 之後但 CPU 停止動作之前發生。在 Windows 3.1 及更早版本中，此訊息送出後，CPU 會回到真實模式，於是 MS-DOS 重新得到控制權。然而在 Windows 95 中，MS-DOS 通常不會重新得到控制權。不過真正厲害的用戶（我並不視自己爲其中之一，直到我讀了 Spencer Katt 專欄所列的以下小技巧後）就知道，如果你是從 MS-DOS prompt 啓動 Windows 95，通常你可以鍵入 mode co80 這個命令，圖形畫面上出現 "It's now safe to shut down your computer"，然後回到 MS-DOS prompt。當然，Microsoft 希望大部份的使用者在這個時候把機器關掉。
- Device_Reboot_Notify 訊息與 System_Exit 訊息類似，Crit_Reboot_Notify 訊息與 Sys_Critical_Exit 訊息類似。這兩組訊息的 "Reboot" 版是在 Windows 95「爲了重新開機而關機」的時刻發生。
- shutdown 訊息有兩個變形：System_Exit 和 System_Exit2。VMM 爲每一個 VxD 維護了一個初始化序號，這樣它才知道循序送出訊息給每一個被載入的

VxDs。VMM 以相反次序送出 "2" 這一型訊息。shutdown 訊息或許本來就應該這樣，因爲任何 VxD 都可以使用由「比它更早初始化的 VxDs」所提供的 services，而這份依存關係可以維持到系統停工爲止。

Virtual Machine 訊息

表 **4-2** 所列的是和 VM 生命息息相關的 system control messages。

訊息	說明
Create_VM	通知 VxD 說，現在是 VM（不含 System VM）誕生的第一個階段。
VM_Critical_Init	通知 VxD 說，現在是 VM（不含 System VM）誕生的第二個階段。
Sys_VM_Init	通知 VxD 說，現在是 System VM 的初始化階段。
VM_Init	通知 VxD 說，現在是 VM（不含 System VM）誕生的第三個階段。
Begin_PM_App	通知 VxD 說，有一個 DPMI client 已經切換到保護模式去了。
Kernel32_Initialized	通知 VxD 說，KERNEL32.DLL 已經被初始化。
VM_Suspend	通知 VxD 說，VM 暫時不能夠執行。
VM_Suspend2	和 VM_Suspend 相同，但訊息傳送次序和初始化次序相反。
VM_Resume	通知 VxD 說，VM 重新可以執行了。
Kernel32_Shutdown	通知 VxD 說，KERNEL32.DLL 即將停擺了。
End_PM_App	通知 VxD 說，保護模式 DPMI client 已經結束。
End_PM_App2	和 End_PM_App 相同，但訊息傳送次序和初始化次序相反。
Query_Destroy	詢問 VxD 是否同意摧毀某個 VM。
Close_VM_Notify	通知 VxD 說，有一個 VM 即將因爲有人呼叫 Close_VM service 而結束。
Close_VM_Notify2	和 Close_VM_Notify 相同，但訊息傳送次序和初始化次序相反。
Sys_VM_Terminate	通知 VxD 說，這是一個正常的 System VM shutdown 訊息。
Sys_VM_Terminate2	和 Sys_VM_Terminate 相同，但訊息傳送次序和初始化次序相反。
VM_Terminate	通知 VxD 說，這是一個正常的 VM（不含 System VM）shutdown 訊息。
VM_Terminate2	和 VM_Terminate 相同，但訊息傳送次序和初始化次序相反。

訊息	說明
VM_Not_Executable	通知 VxD 說，這是摧毀一個 VM 的倒數第二階段。
VM_Not_Executable2	和 VM_Not_Executable 相同，但訊息傳送次序和初始化次序相反。
Destroy_VM	通知 VxD 說，這是摧毀一個 VM 的最後階段。
Destroy_VM2	和 Destroy_VM 相同，但訊息傳送次序和初始化次序相反。

表 4-2　Virtual machine control messages

以下是有關上述訊息的重要注意事項：

- 爲了反應出 Windows 架構中 System VM 的特殊權級(privileged)地位，System VM 有自己的初始化和 shutdown 訊息。
- 產生一個新的 VM 時，Windows 送出三個 control messages：Create_VM、VM_Critical_Init、VM_Init。只有最後那個訊息有對應的 System_VM 相關訊息(也就是 Sys_VM_Init)，因爲「產生 System VM 」是 VMM 在呼叫任何 VxD 之前自動做的事情。
- 如果一個 VM 正常結束，Windows 會送出 VM_Terminate 訊息。VxDs 也可以以不正常的手法結束一個 VM，例如呼叫 *Nuke_VM* 或 *Crash_Cur_VM* services。不管 VM 如何結束，VM_Not_Executable 和 Destroy_VM 都會伴隨而來。
- 就如前一節所言，shutdown 訊息有兩個變形。也就是說，VMM 以「和初始化次序相反的次序」送出 "2" 這一型訊息。

其他的 System Control Messages

表 **4-3** 列出其他的 system control messages。

訊息	說明
Set_Device_Focus	要求 VxD 重新指定對某個 device 或所有 devices 的擁有權。
Begin_Message_Mode	通知 VxD 說 SHELL 即將顯示一個訊息。
End_Message_Mode	通知 VxD 說 SHELL 已經完成一個訊息的顯示。
End_Message_Mode2	和 End_Message_Mode 相同，但訊息傳送次序和初始化次序相反。
Debug_Query	要求 VxD 產生 debugging 輸出。
Power_Event	要求 VxD 處理系統電力的變化（譯註：和筆記型電腦有關）
Sys_Dynamic_Device_Init	要求 VxD 將一個 dynamic VxD 初始化。
Sys_Dynamic_Device_Exit	要求 VxD 準備卸載（unload）一個 dynamic VxD。
Create_Thread	通知 VxD 說這是 thread 誕生的第一個階段。
Thread_Init	通知 VxD 說這是 thread 誕生的第二個階段。
Terminate_Thread	通知 VxD 說這是 thread 結束的第一個階段。
Thread_Not_Executable	通知 VxD 說這是 thread 結束的第二個階段。
Destroy_Thread	通知 VxD 說這是 thread 結束的第三個階段。
PNP_New_Devnode	通知 VxD 說有一個新的 device node 已經產生。
W32_DeviceIoControl	通知 VxD 說有一個 Win32 程式發出 *DeviceIoControl* 呼叫。
Get_Contention_Handler	通知 VxD 說 VCOMM 要一個 contention handler 的位址。

表 4-3 其他的 system control messages

以下是有關上述訊息的重要事項：

- Set_Device_Focus 是 Windows 改變 device(如滑鼠)擁有權的一個基本方法。你可以針對特定 device 發出這個訊息，那麼每一個 VxD 必須檢視此訊息是否和其 device 相配。要不然你也可以對所有 device 發出這個訊息，於是多個 VxDs 會有所反應。
- Begin_Message_Mode 和 End_Message_Mode 用來將 SHELL device 所發出的 system modal message 涵括起來。這些訊息常常有眾所熟悉的「藍色訊息視

窗」。雖然這樣一個訊息是可見的，device driver 不應該做任何可能「干擾此一訊息或使用者的回應」的事情。

- Debug_Query 發生於使用者在除錯器中下達特殊命令時。例如，*.V86MMGR* 命令會造成此一訊息被送往 V86MMGR device。處理這個訊息將有助於你對自己的程式碼除錯。
- Power_Event 通知有興趣的 VxDs 說，電力系統即將有所變化。以我在飛機上使用筆記電腦的經驗來說，這表示大事不妙了。
- Sys_Dynamic_Device_Init 和 Sys_Dynamic_Device_Exit 是 Windows 用來初始化和結束 dynamic VxDs 的兩個訊息。
- 五個和 threads 生命有關的訊息（Create_Thread, Thread_Init, Terminate_Thread, Thread_Not_Executable, Destroy_Thread）涵蓋了 thread 的生存跡象。
- PNP_New_Devnode 通知 VxD 說，Configuration Manager 已經產生了一個新的「device 節點」，也就是一個表現實際 device 的控制區塊。
- Win32 程式可以呼叫 *CreateFile* 來獲得一個 VxD handle，做爲 *DeviceIoControl* 的參數之一。呼叫 *DeviceIoControl* 會導至一個 W32_DeviceIoControl 訊息。
- VCOMM（管理 serial 和 parallel port 的 VxD）使用 Get_Contention_Handler 來獲得一個 callback 函式位址；該函式用來處理 "device contention"。

私有的（Private）Control Messages

凡設計用來一起合作的 VxDs，可以使用 *Directed_Sys_Control* API 送出私有訊息給其他 VxDs。這種用途的 system control message 的範圍從 70000000h 到 7FFFFFFFh（從 BEGIN_RESERVED_PRIVATE_SYSTEM_CONTROL 到 END_RESERVED_ PRIVATE_ SYSTEM_CONTROL）。VMM 並不爲那些訊息提供任何註冊服務（registry），以此方法溝通的 VxDs，必須自我瞭解這些訊息的意義才行。

VxD Service API

VxD 使用 VxD service API 來和 VMM 以及其他 VxDs 溝通。這個 API 利用 INT 20h 和一組 VxD 服務序號，提供 VxDs 之間的動態聯結。任何 VxD 的任何 service 函式的服務序號都是獨一無二的。VMM 供應了大約 400 個 services，對於許多 VxDs 而言大有用處。其他 VxDs 則提供少量特殊目的的 services APIs。

每一個開放出「service 進入點」的 VxD，都有一個獨一無二的 16 位元識別碼（也就是 VxD ID），以及一個表格（其位址出現在 Device Description Block 之中）。表格內的每個元素都指向一個函式，實作出一個可被外部呼叫的 service。Driver 設計者必須提供一個表頭檔，定義一系列的 32 位元 service table 常數，包括助憶碼如 VKD_Define_Hot_Key，以及數值常數如 000D0001h。每一個常數的前半部代表 VxD ID，後半部代表從 0 開始起算的表格索引，此索引即是 service 函式在表格中的位置（圖 **4-1**）。因此，000D0001h 表示「VxD ID 爲 000Dh（亦即 Virtual Key Device，VKD），service 編號爲 1（Define_Hot_Key）」。

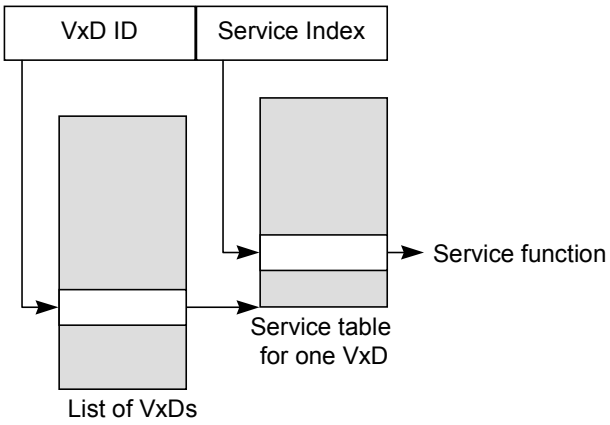


圖 4-1 一個 VxD service 識別碼

VxD 如果要呼叫 service 函式，只要發出 INT 20h 並緊跟著一個 service 識別碼即可。VMM 會利用識別碼前半部的 Device ID 找出 VxD，再利用識別碼後半部找出在 VxD service table 中的表格元素（譯註：是個函式指標）。VMM 然後就呼叫該 service 函式，並安排讓控制權回到呼叫端（calling VxD）的呼叫動作的下一行。以 assembly 語言撰寫之驅動程式可以使用 VMMCall 和 VxDCall 巨集，自動產生 INT 20h 指令及其後續資料，例如：

```
include vkd.inc
...
mov     al, scancode
mov     ah, scantype
mov     ebx, shiftstate
mov     cl, operationflags
mov     esi, offset32 callback
mov     edi, delay
VxDCall VKD_Define_Hot_Key
```

很明顯，搜尋 VxD service table 並對它進行索引動作，是件費時的工作。所以 VMM 第一次遭遇它們的時候，就毅然做了 service linkages 動作。VxDCall 巨集最初被擴展爲：

```
CD 20           INT   20h
01 00 0D 00     DD    VKD_Define_HotKey
```

這個指令第一次被執行後，VMM 以一個 "indirect CALL" 指令將這 6 個 bytes 取代掉，改爲直接呼叫 service 進入點，如下所示（其中的 x 表示 service 進入點的位址）：

```
FF 15 xx xx xx xx    CALL [$VKD_Define_Hot_Key]
```

這種聯結方式很明顯比原來快。某些 service calls 甚至因此更有效率。例如，VMM 以一個 MOV 指令取代 *Get_Cur_VM_Handle* service，使它能夠直接參考到放置 current VM handle 的記憶體位址。此之所以能夠成功，是因爲 MOV 指令佔用 6 個 bytes，正好是 INT 20h 指令加上 service ID 的長度。

譯註：Andrew Schulman 等人於 ***Microsoft Systems Journal***（***MSJ***）1992/02 發表了一篇名爲 *Call VxD Functions and VMM Services Easily Using Our Generic VxD* 的文章。雖然發表日期已久遠，但對於上述所說的動態聯結，仍然極具參考價值。

這種 function linkage 方法，提供了數個重要的優點。第一，有可能因爲改變 service table 中的元素，而將其他 VxDs 對此 VxD 的所有呼叫都攔截下來。有一個 VMM service 名爲 *Hook_Device_Service*，就是這麼做的。另一個優點是，VMM 可以動態決議 VxD 對另一個 VxD 的參考（呼叫）動作。這很類似 Windows DLLs 的動態聯結，但由於聯結動作發生於呼叫端真正發出一個 VxD service call 的時候，所以不需對那些其實並未被呼叫的符號也做聯結。也就是說，你可以建立一個 VxD，根據執行時期的結果，條件式地呼叫某些 services，不必擔心沒辦法載入 VxD。若是想在 Windows DLL 中完成類似的動態聯結，你必須自己以 *GetProcAddress* 解決「符號參考」的問題。

譯註：我們常把一般 Windows DLLs 的動態聯結（帶有 import library 的那種）稱爲「載入時期（loadtime）動態聯結」，而把藉助 *GetProcAddress* 所完成的動態聯結稱爲「執行時期（runtime）動態聯結」。

最後一個優點是，你可以輕易知道某個 VxD 是否被載入；是的，只要呼叫其 service 即可。通常每一個 VxD 都會開放一個 *Get_Version* service，索引爲 0，你可以呼叫這一個！如果失敗，就知道該 VxD 不在記憶體中了：

```
VxDCall FOO_Get_Version
jc      not_loaded
...
```

你之所以呼叫 *Get_Version*，因爲通常它不會失敗；如果失敗必是由於某些不正常因素，例如根本就沒有 VxD 回應等等。

Dynamic VxDs 中的 Services　利用 *Get_Version* 所做的上述測試，只對 static VxDs 有效，因爲它們只可能在系統啓動時載入。雖然（嚴格地說）你不應該在 dynamic VxDs 中開放 services，不過並非不可能。於是便有可能發生這樣的事情：當有人第一次嘗試呼叫 *Get_Version*，此 service 不存在；而當那個 VxD 最後出現了，呼叫端還是獲得「沒人在家」的回答，因爲 VMM 做了動態聯結。爲了避免這種困擾，請使用 *Get_DDB* service 來詢問 dynamic VxD 是否存在，並使用 *Directed_Sys_Control* service 以及某些 "cross-registration" 的協定來和它通訊。

Callback 函式

VxDs 也經由 callback 函式來通訊。在這裡，callback 意指「一個位於 VxD A 中的函式；VxD B 可藉由 VxD A 所提供的指標，呼叫該函式」。典型的例子就是 VMM 呼叫 VxD 的 device control procedure 以處理某些 system control messages（例如 Sys_Dynamic_Device_Init）。VxD 的訊息處理常式可以使用 VxD service（像是 *_VCOMM_Register_Port_Driver* 之類）來登錄一個 callback 函式位址。從此以後，這個 callback 函式就可以被呼叫以執行某些功能了。

另一個和 callback 有關的典型情況是，VxD 在處理某個 system control message 時 "hooks" 一個中斷（也就是說安裝了一個新的中斷處理常式）。VMM 或其他一些第一層（first level）中斷處理常式（譯註：第６章）會呼叫此一新函式（hook 函式），由後者處理中斷。

Win32 API

當初爲了「在 Windows NT 跑 32 位元程式」而設計出來的 Win32 API，現在已可輕易在 Windows 95 上運用。就像在 Windows NT 上一樣，現在你可以使用 Win32 API 在 Windows 95 產生一個 32 位元程式。你可以產生 32 位元的 Windows EXE 或 DLL，或是 32 位元的 "console" 程式（很像傳統的 MS-DOS 程式）。有幾組 Win32 APIs 是爲系統程式設計而準備的，我將在這裡做個摘要。

Process 和 Thread 的管理

Windows 95 針對 Win32 程式所支援的「process 模型和 thread 模型」，和 Windows NT 完全一樣。在此模型中，每個程式擁有一個 process，其中又有一個（以上）的 threads。所有 threads 都和其他 threads 一起搶奪 CPU 時間，靠的是優先權和一些演算法，以儘量保持公平。每個 process 擁有自己的 **memory context**，佔用 00400000h~80000000h 之間的線性位址空間；映射於此範圍的實際記憶體，則由同一個 process 的所有 threads 共享。其他 process 的 threads 不能存取這些記憶體。

你可以呼叫 *CreateThread* 建立新的 thread。參數之一是個函式位址，那是 thread 將執行的地點。當該函式結束，或是呼叫了 *ExitThread*，thread 即結束。*CreateThread* 的另一個參數用來指示 thread 是否產生於凍結狀態。你可以呼叫 *ResumeThread* 讓一個虛懸（凍結）的 thread 重獲生機。*CreateThread* 傳回一個 thread handle，許多場合都會用到它。例如，你可藉以呼叫 *TerminateThread*，強迫結束某個 thread，或是呼叫 *GetExitCodeThread* 獲得其結束代碼，或是呼叫 *WaitForSingleObject*，等待 thread 結束自己。

同一個 process 的所有 threads 共享相同的 virtual memory context，包括程式所擁有的 static storage。爲了讓 thread 產生出「和其他 threads 所擁有的不同」的永久性資料空間，我們可以使用所謂的 **thread local storage**（TLS）機制。在 Microsoft Visual C++ 之中，你可以使用 *__declspec(thread)* 來宣告 thread 的 local objects。執行時期，一個 master thread 可以使用 *TlsAlloc* 配置一個 TLS 索引值，給它自己以及其他 thread 使用。通常你可以儲存這個索引於全域變數之中，使它能隨時隨地被程式使用（在呼叫 *TlsGetValue* 和 *TlsSetValue* 時用到）。

> 譯註：一般人對於系統如何製作出 TLS，常感困惑。Matt Pietrek 的 ***Windows 95 System Programming Secrets***（IDG Books / 1995；中譯 **Windows 95 系統程式設計大奧秘**，侯俊傑 / 旗標 / 1997）第三章將 thread 的資料結構做了完整的剖析，其中涵蓋 TLS 設計原理。此書對於作業系統的資料結構和 API 函式動作有極詳盡的剖析，是極佳書籍。

譯註：以下出現 **priority base** 和 **priority boosts** 兩個名詞，我保留原文，因爲你會在許多原文書中看到這兩個名詞，早點熟悉的好！base 和 boost，其實就是「基礎」與「微調」的觀念。

VMM 內含一個所謂的 secondary scheduler（次排程器）VxD，用來實作 Win32 thread 的優先權模型（priority model）。次排程器是主排程器（primary scheduler）的一個 client 程式，後者在一個精準的 **priority base** 上運作。你可以藉由呼叫 *GetThreadPriority* 和 *SetThreadPriority* 來控制一個 thread 的 **base priority** 值。次排程器對 base priority 實施優先權提昇（**priority boosts**）動作，以反應主排程器的選擇。其他 VxDs 也可以加減 thread 的 priority boosts。這些 boosts 通常都比次排程器的 boosts 數值高得多。Win32 API 所設定的優先權值其實只是用來決定 thread 是否獲得 CPU 時間的因素之一而已。

你可以在 ***Microsoft Win32 Programmer's Reference Vol.2***（Microsoft Press / 1993）的第 43 章找到有關於 process 和 thread 管理的其他資訊。本書第６章對 threads 的排程還有更多描述。

Thread 的同步化控制（Thread Synchronization）

譯註：同步化物件有兩種狀態：**signaled** 和 **unsignaled**，我將之譯爲「激發狀態」和「未激發狀態」。

Windows 95 實作出數種同步控制物件：mutex objects, semaphore objects, event objects，允許 threads 藉此控制彼此的操作順序。你可以利用 API 產生任何一種同步物件，並獲得一個 handle。你可以呼叫 *CloseHandle* 將它摧毀。你可以使用 *WaitForSingleObject*、*WaitForSingleObjectEx*、*WaitForMultipleObjects*、*WaitForMultipleObjectsEx* 等 *Wait* 函式，讓 thread 等待，直到某個（或某些個）同步物件變成激發狀態爲止。除了上述所說的同步物件之外，你也可以在這些 API 中使用其他的 handles，例如 thread handle（當 thread 結束，其 handle 就處於激發狀態）。

譯註：關於同步物件的運用，在 Jeffrey Richter 所著的 ***Advacend Windows 3/e***（Microsoft Press，1997）第 10 章有極多極佳極詳盡的實例與解釋。

Mutex objet（mutex 意味 mutual exlcusion，可譯爲「互斥器」）一次只能被一個 thread 擁有。你可以利用 *CreateMutex* 產生一個 mutex object。當它未被任何 thread 擁有，就是處於激發狀態；當它被擁有，就是處於未激發狀態。因此，欲等待進入一個 mutex 的保護範圍，thread 可以使用先前所提的四個 *Wait* 函式；沒有獲得 mutex 的人就得等待繼續下去。如果要離開 mutex 保護範圍，並將 mutex 恢復爲激發狀態，你可以使用 *ReleaseMutex* 函式，於是其他等待中的 threads 就有機會獲得這個 mutex 了。

Semaphore object 本質上是一個不帶正負號、附有「同步控制語意」的整數。當 semaphore 的數值大於 0，它就處於激發狀態；當 semaphore 的數值等於 0，它就處於未激發狀態。你可以呼叫 *CreateSemaphore* 產生一個 semaphore。你可以呼叫前述任何一個 *Wait* 函式等待 semaphore 變爲非零值。任何 threads 都可以呼叫 *ReleaseSemaphore* 來增加 semaphore 的數值。這樣的特性使你能夠將 semaphore 運用在「可同時被有限多個使用者共同使用」的設備的同步控制上。

所謂 event object，是一個簡單的閘門。你可以呼叫 *CreateEvent* 產生一個 event，並使用稍早所提的任何一種 *Wait* 函式來等待 event 到達激發狀態。Event 是否進入激發狀態、進入時間有多長，完全視你是否使用 *SetEvent* 或 *PulseEvent* 來激發它，以及它是手動重置（manual reset）或自動重置（auto reset）而異。*CreateEvent* 有一個參數可以指定此 event 爲手動重置或自動重置。*SetEvent* 會使一個手動重置的 event 進入激發狀態，此狀態維持到有人對它呼叫 *ResetEvent* 爲止。當手動重置的 events 進入激發狀態，任何等待中的 threads 會如猛虎出閘一般立刻開始執行。*SetEvent* 也可以令一個自動重置的 event 進入激發狀態，但一旦有個 thread 因此獲得執行權，此 event 立刻自動恢復爲未激發狀態。如果目前沒有任何 thread 在等待，*PulseEvent* 不會產生任何影響。對一個手動重置的 event 呼叫 *PulseEvent*，會使所有等待中的 threads 都被釋出（譯註：不再被凍結）；對一個自動重置的 event 呼叫 *PulseEvent*，則只會使一個等待中的 threads

被釋出；而一旦 *PulseEvent* 完成，不論手動重置或自動重置的 event 都會回到未激發狀態。

Wait 函式的擴充型式（*Ex* 者）允許一種所謂的可變等待狀態（alterable wait state）。在 Windows NT 中，你可以利用這東西，允許系統呼叫相同 thread 中的一個 asynchronous I/O completion routine；在 Windows 95 中，你可以利用這東西等待一個被 VxD 排班的 asynchronous procedure call。

Windows 95 實作有一個名爲 *OpenVxDHandle* 的函式，可以爲 Win32 event object 產生一個 ring0 handle。你可以透過 *DeviceIoControl* 把這個 handle 交給 VxD，VxD 於是可以對此 handle 使用某些 Win32 函式的 ring0 同等服務。例如，VxD 可以使用 *_VWIN32_SetWin32Event* service 來執行一個類似 ring3 *SetEvent* 的動作。

你可以在 ***Microsoft Win32 Programmer's Reference Vol.2*** 的第 44 章發現更多有關於同步控制的細節。本書第 9, 10 兩章將描述 VxDs 如何使用 ring0 的 synchronization calls 和 asynchronous procedure calls。

> 譯註：Threads 之間的同步控制，向來是程式設計的一大關口，不論對一般程式設計或系統程式設計都如此。***Multithreading Application in Win32***（Jim Beveridge & Robert Wiener，Addison Wesley，1997）是一本專門討論多緒程式設計的書籍，可爲入門之用；***Advanced Windows 3/e***（Jeffrey Richter，Microsoft Press，1997）第 10 章 Thread Synchronization 可爲進階之用。

System Registry

Windows 95 registry 是一個樹狀階層資料庫，由 **key**s 和 **value**s 組成。每一個 key 可以有無限制的 subkeys。每一個 key 可以有唯一一個 **unnamed value**，和任何個數的 **named value**s。系統元件以及應用程式都喜歡使用 registry 來儲存安裝好的軟硬體的永久性資料。

對系統程式設計而言，system registry 的最上層有兩個分支特別重要：HKEY_LOCAL_MACHINE（縮寫爲 **HKLM**）和 HKEY_DYN_DATA。Windows 95 的 Configuration Manager VxD 使用 HKLM 中的記錄來對系統進行組態（configuring），並選擇驅動程式。Configuration Manager 也將執行時期的資訊記錄於 HKEY_DYN_DATA 的 dynamic keys 之中。

Win32 程式可以使用正規的 Win32 API 來存取並修改 registry。通常你可以利用 *RegOpenKey* 打開最上層 keys（如 HKLM）的一個 named subkey 並獲得一個 handle，或是利用 *RegCreateKey* 產生一個新的 subkey。你可以呼叫 *RegQueryValue* 來詢問和某個 key 相關聯的 unnamed value，也可以呼叫 *RegQueryValueEx* 來檢驗 named values。相似函式 *RegSetValue* 和 *RegSetValueEx* 可用來修改某個已被打開的 key。若要知道在某個已被打開的 key 之下有哪些 subkeys 或 values，可以呼叫 *RegEnumKey* 和 *RegEnumValue* 函式。

Win16 程式也可以使用 Windows 95 版的 16 位元 WINDOWS.H 表頭檔內所宣告的函式來處理 registry。那個表頭檔是 DDK 乃至於 16 位元編譯器的一部份。

你可以在 ***Microsoft Win32 Programmer's Reference Vol.2*** 第 52 章發現更多有關於 registry 的細節。本書第 11, 12 兩章描述 Windows 95 的 Plug and Play 子系統，其間 Configuration Manager 大量使用了 registry。

相容性介面（Compatibility Interfaces）

前一版 Windows 中有數個系統介面是很具重要性的。包括 DPMI、XMS、EMS，以及 Virtual DMA services。爲了回溯相容的緣故，Windows 95 繼續支援這些介面。

DPMI

DPMI 讓保護模式程式能夠取用 Windows system services 的一部份。DPMI 的原始目的是讓具商業價值的 DOS extender 產品能夠在 Windows 3.0 的 MS-DOS 虛擬機器中運

作良好。後來 Windows 系統程式員發現 DPMI 的用途超越了原來的目標。Windows 95 繼續支援 DPMI，以免破壞那些仍然依賴 DPMI 的應用程式。

DPMI 可以解決「在一部虛擬機器中跑一個 DOS extender」的需求。程式首先在 V86 模式開始其生命，然後發出一個 INT 2Fh, function 1687h，獲得「模式切換服務常式」的位址。呼叫該函式，便可將 CPU 從 V86 模式切換到保護模式。然後再使用 INT 31h 向所謂的 DPMI "host"（譯註：或稱爲 DPMI server）要求服務。程式打算結束時，呼叫 INT 21h, function 4Ch 離開保護模式，回到 MS-DOS prompt。有趣的是，Windows 95 的 KERNEL 模組就是使用相同的機制在 System VM 上開始執行 Windows 各元件。

DPMI 只在 assembly 語言中才顯實用。DPMI calls 要求有一個功能代碼放在 AX 暫存器中，用以指定數十個服務中的一個。某些 calls 要求更多的參數，放在 general 暫存器或 segment 暫存器中。執行結果通常是放在 general 暫存器中，如果回返時 carry flag 清除，代表成功；如果回返時 carry flag 設立，代表失敗。未特別說明的暫存器，其內容不會在回返時有所變化。

DPMI 的功能代碼是以 high byte 區分大類別，以 low byte 指定特定服務。**圖 4-4** 列出 DPMI 0.9 版的所有服務類別，再加上 0Exxh 服務。當初 DPMI 委員會在正式的 1.0 版未審理完成前，先推出 0.9 版。當 1.0 版終於推出，Microsoft 卻不打算實作其中除了 0Exxh 之外的任何新性質。Windows 95 支援 DPMI 0.9，就像過去 Windows 3.0 和 3.1 以及現在的 Windows NT 一樣。

功能分類	說明
00xxh	selectors 和 descriptors 的管理服務
01xxh	V86 記憶體（最低 1MB）的管理服務
02xxh	中斷攔截（hooking）服務
03xxh	「保護模式至真實模式」或「真實模式至保護模式」的服務
04xxh	取得版本（只有一個服務：0400h）
05xxh	延伸記憶體（extended memory）管理服務

功能分類	說明
06xxh	Page locking 服務
07xxh	Paging 效率調整服務
08xxh	實際（physical）記憶體的映射服務（只有一個服務：0800h）
09xxh	虛擬中斷狀態管理服務
0Axxh	老舊函式（0A00h），用來決定一個由廠商提供的進入點。
0Bxxh	除錯暫存器的管理服務
0Exxh	浮點協同處理器（floating-point coprocessor）的管理服務

表格 4-4 DPMI services 分類

如果你仔細瞧瞧 DPMI 的服務，你會注意到其中許多功能是爲 DOS extender 設計的。讓我先對 DOS extender 做個簡單的描述。我所熟悉的數個 DOS extender 基本上是程式載入器 + runtime 函式庫（用以支援標準的 MS-DOS 和 BIOS 軟體中斷介面）。切換到保護模式之後，extender 需要打開並讀取可執行檔。原本這應該使用標準的 INT 21h API，但 extender 必須使用 DPMI function 0205h 來攔截這一中斷的保護模式版，再交給真實模式 DOS：

```
mov     ax, 0205h          ; function 0205h : set PM interrupt vector
mov     bl, 21h            ; BL = interrupt number
mov     cx, cs             ; CX:DX -> interrupt handler
mov     dx, offset int21   ; ..
int     31h                ; hook interrupt in protected mode
```

這時候 DPMI host 的協助是必要的，因爲 DOS extender 不容易取得 interrupt descriptor table（IDT）。現在，當 DOS extender 發出一個 INT 21h call，上述名爲 *int21* 的那段碼會獲得控制權（尚處於保護模式）。這就像使用 DPMI function 0300h（模擬真實模式中斷）把中斷交到真實模式去一樣。此時再一次需要系統軟體的協助，因爲把 CPU 從保護模式切換出去再切換回來，CPU 暫存器必須有所改變，而它只能被一個 ring0 主控程式處理。DOS extender 的部份工作是把指標參數從保護模式所使用的 selector:offset 型式轉換爲真實模式所使用的 segment:offset 型式。轉換動作需要將資料從延伸記憶體

（extended memory）拷貝到真實（V86）模式視線所及的最低 1MB 區域。拷貝動作需利用 DPMI function 0100h 在最低 1MB 區域配置一塊記憶體、利用 DPMI function 0000h 配置一個 selector、再利用 DPMI function 0007h 將 selector 的 base 設定指向真實模式記憶體... 等等等。

雖然不爲大眾所知，不過其實 Windows 3.0 早已內含一個十分完整的 DOS extender；Windows 95 亦保有此一事實。如果你可以載入一個程式並重新定位（relocate）它，讓它能夠使用保護模式的 selector，那麼 Windows 將處理此程式可能使用的所有軟體中斷。因此，一個 DOS extender 其實並不需要 "hook" INT 21h，因爲 Windows 對於 INT 21h 的預設行爲就是轉換指標參數並將中斷交由真實模式處理。Windows 對於滑鼠驅動程式（INT 33h）、video BIOS（INT 10h）等中斷提供了適度的處理，因此很可能某人爲 Windows 3.0 所寫的 DOS extender 只是利用 DPMI 切換進入保護模式並使用正規的檔案系統呼叫將 client 程式載入 extended memory 而已。標準的 runtime 函式庫中有一小部份函式在保護模式中不能運作，需要換掉，但大部份商用的 DOS extender 做的遠比這多得多。

DPMI 的最大目的是爲了 extended DOS 程式。DPMI 在 Windows 95 中繼續存在的主要理由則是爲了回溯相容。由於你現在可以使用 Win32 API 寫 32 位元 console 程式，因此沒有什麼理由需要買一個 DOS extender。如果你的工作是在一個 extended DOS 程式身上，你應該檢討是否能夠把它重建爲一個 Win32 console 程式，這樣一來可以把你從 DOS extender 的枷鎖中釋放出來，並爲「移植到其他 Win32 平台」鋪上一條康莊大道。

在 16 位元 Windows 程式中使用 DPMI，還是有一些限制（由於技術上的理由，一個 32 位元 Windows 程式不能夠使用 DPMI calls 或任何其他以軟體中斷形式出現的介面）。每次你呼叫一個「非標準的真實模式函式」（也就是 Windows 內建之 DOS extender 並未爲此函式處理必要之指標轉換），你可能得使用 DPMI 03xxh 功能。DPMI 09xxh 功能允許你控制虛擬中斷旗標（virtual interrupt flag），它是一個控制閘門 ，控制著 VMM 是否將中斷反映至 VM。

DPMI 0800h 功能對於那些「想發展虛擬位址」以處理記憶體之 Windows 程式有些幫助。這種程式可能燒在擴充卡上，其實際位址在 extended memory 上已知。欲建立一個「必要的 selector」以爲指標所用，必須使用多個 DPMI 功能。Windows 提供的 *AllocSelector, SetSelectorBase, SetSelectorLimit* 有對等功能，又方便得多。在 Microsoft 公開這些 API 函式之前（Windows 3.1 SDK 之前），許多程式員只能依賴 DPMI，因爲他們不知道有其他方法。

最後，DPMI 還是爲「遠古孑遺」的程式（MS-DOS 程式和早期 Windows 程式）提供了一條生路。那些程式彼此共享記憶體。一個在虛擬機器中執行的 extended DOS 程式可以使用 0501h 功能配置一塊 extended memory，讓其線性位址落在 80000000h~C0000000h 之間。其他虛擬機器中的 extended DOS 程式可以經由相同的線性位址取用這塊記憶體，System VM 中的 Win16 程式也是一樣！不過 Microsoft 希望以此法通訊的程式能夠改頭換面，像 Win32 程式那樣以 named file mapping 來通訊。

DPMI 的終極原始資訊是其官方規格，你可以從許多地方（包括 Intel 公司）獲得這份規格。本書第 17 章將討論 DPMI 細節。

Extented Memory Management（XMS）

延伸記憶體規格（XMS）統治了 MS-DOS 程式取用延伸記憶體（extended memory，1MB 以外）的方法。每一個 Windows 95 系統都有一個 XMS 供應者，通常是 Windows 95 版的 HIMEM.SYS。真實模式程式可以利用 INT 2Fh, function 4310h 獲得一個 XMS 服務常式位址。這個中斷會傳回一個真實模式函式位址，該函式可以視 AH 暫存器中的功能代碼而執行各種記憶體配置。注意，你不能夠直接在保護模式中使用 XMS，像 DOS extender 這樣的程式必須先切換到真實模式才能呼叫 XMS 服務常式。

XMS 的各項服務允許你：保留或釋放從線性位址 00100000h 開始的 high memory area（HMA）部份空間、配置或釋放 extended memory、控制 A20 硬體位址線（A20 負責實際位址的第 21 個位元；由於 8086 CPU 只提供 20 個位元的實際位址，80286+ 允許程式員將 A20 位址線 disable 掉，以維護回溯相容）。MS-DOS 程式很少直接使用

XMS 服務，較多時候是依賴一個底層（如 DOS extender）和 XMS 供應者溝通。

XMS 在兩方面影響 Windows 95。第一，Windows 在系統啓動時使用 XMS 來控制所有的 extended memory。第二，Windows 95 模擬 XMS 服務，以求回溯相容於原本依賴那些服務的程式。下面這個簡單的 debug 程式就取得 XMS 進入點並使用 XMS 的 0h 功能，詢問服務供應者（HIMEM.SYS）的版本號碼：

```
C:\>debug
-a 100
2DBE:0100 mov ax, 4310
2DBE:0103 int 2f
2DBE:0105 mov [200], bx
2DBE:0109 mov [202], es
2DBE:010D mov ah, 0
2DBE:010F call far [200]
2DBE:0113 int 3
2DBE:0114
-g

AX=0300  BX=035F  CX=0000  DX=0001 ...
...
2DBE:0113 CC          INT 3
-q
```

傳回值放在 AX 暫存器中，表示 XMS 服務提供者支援 XMS 3.0。BX 暫存器值表示修正版本，本例爲 3.95 版。

關於 XMS 的更多資訊，請參考 ***Extended Memory Specification（XMS）3.0***，你可在 MSDN 光碟的 "Specifications" 區找到。

Expanded Memory Management（EMS）

所謂 expanded memory（擴充記憶體），有一段可歌可泣的歷史，並且幾乎反映出 PC 硬體與軟體的成長標準。早期有許多 MS-DOS 程式難以更加茁壯，大多因爲它們只有 640KB 記憶體可用。解決方法之一是 Lotus-Intel-Microsoft 所制定的擴充記憶體企劃。你可以爲你的 PC 加上一片擴充卡，上有數 megabytes 記憶體。這張擴充卡在某一段「可

調整之 segment 位址（所謂 page frame）」中的行爲，就像一般的 RAM 一樣；page frame 位在 ROM 區域（A000h:0000h ~ F000h:0000h 之間）。我們可以對此擴充卡撰寫程式，在 page frame 位址空間中做出數個 16KB pages。應用程式並不需要直接對硬體 I/O ports 做動作，而是呼叫軟體中斷 67h，取用一個附隨擴充卡而來的驅動程式（譯註：所謂的 XMS driver）的服務。

當 80386 面世，以軟體模擬 expanded memory 的技術變得可行了。Expanded memory 管理程式如 EMM386、QEMM、386Max 都仰賴 CPU 的 V86 模式和 paging 能力來完成這個任務。簡單地說，這些記憶體管理程式把電腦切到 V86 模式，然後將 paging 功能 "enable" 起來，並以一組 page tables 將「虛擬記憶體的 1st MB 的大部份」映射到完全相同的實際位址上。記憶體管理程式面對外界的 expanded memory 索求，解決辦法就是將 extended memory 映射到設計好的 V86 page frame 位址上。

除此之外，新的 expanded memory 管理程式，也以 extended memory 填充實際位址空間中的一些「洞」。典型的 PC，有些 RAM 映射到位址 00000h~A0000h 之間（也就是 0000h:0000h~A000h:0000h 之間），擴充卡和 BIOS 則提供額外的 RAM 和 ROM 於剩餘的 386 KB 位址空間中。其間常常有些不定大小的「洞」沒有被填入任何記憶體。記憶體管理程式可以利用這些「洞」產生所謂的 upper memory block（UMB），讓 MS-DOS 使用這些記憶體來放置驅動程式和常駐程式。

如果 PC 在 V86 模式跑，有一個不好的副作用：某些指令在真實模式沒有問題，到了 V86 模式卻因爲 privilege（CPU 權級）而產生 general protection faults（GPF）。第一個會出現這種問題的程式就是 DOS extender。此類產品的領導廠商（Phar Lap Software 和 Quarterdeck Office System）於是定義所謂的 Virtual Control Program Interface（VCPI），描述如何讓 DOS extender 和記憶體管理程式共存。VCPI 也使用軟體中斷 67h，並提供進入和退出保護模式、管理實際 pages、存取 privileged 暫存器… 等功能。

Windows 95 不能夠和 VCPI 記憶體管理程式共存。事實上，如果你在 Windows 95 的 MS-DOS 視窗中使用 INT 67h, function DE00h 偵測 VCPI 存在與否，你會被告知 VCPI 不存在。即使你在啓動 Windows 95 之前載入了一個 VCPI 記憶體管理程式亦然！Windows 95 啓動的時候，會發出中斷 2Fh, function 1605h 改變 Windows 95 接管機器時的所有常駐程式。VCPI 記憶體管理程式基本上是提供模式切換服務常式位址，稍後 Windows 95 可以用它來切換 V86 模式和真實模式。有一個未公開的介面，允許記憶體管理程式提供其 page mapping 資訊給 Windows 95。這些資訊允許 V86 memory management VxD 將 Windows 95 啓動時所用的 UMBs 複製一份。任何 extended DOS 常駐程式如果沒有 Windows 95 VxD 的協助，就沒有辦法執行。

Ralf Brown 和 Jim Kyle 合著的 ***PC Interrupts*** 第 10 章（Addison Wesley，1991）是一份很不錯的 VCPI 補充資料。

Virtual DMA Services

Direct Memory Access（DMA，直接記憶體存取）提供了一種在硬體設備和記憶體之間高速傳輸資料的方法。在我所能想到的不幸情況中，硬體 DMA 控制器有三個嚴重束縛：第一，它只能對實際（physical）記憶體操作；第二，它會妨礙某些記憶體邊界的跨越；第三，視搭配的硬體 bus 而定，DMA 或許只能夠處理前 16 MB 實際記憶體。Windows 95 應用程式都使用線性位址，而線性位址對 DMA 控制器沒有意義；而且電腦可能裝有 16MB 以上的 RAM。

雖然如此，真實模式程式員卻可以像從前一樣繼續寫他們的 DMA 傳輸程式，他們可以愉快地忽略掉 Windows 95 所帶來的可能問題。這是因爲 Virtual DMA Device（VDMAD）將 DMA 控制器的 ports 都虛擬化了，使每一件事都依然順利運作。因此，真實模式的軟碟 driver 可以啓動一個傳輸動作，傳到一個已知的 V86 模式位址上，而不必擔心 V86 記憶體管理器可能已經將兩個 16MB 以外的不同 physical pages 映射到 V86 位址空間的特定位置。

VxD 撰寫者如果要規劃 DMA，做傳輸動作，也可以呼叫 VDMAD 的許多服務函式，不必直接存取 DMA 控制器的 ports，因而可以避免許多 DMA 控制器的瑣碎細節。

保護模式程式（包括 ring3 Windows 驅動程式）形成了一種不被「VDMAD 的初始化動作或其 VxD 服務介面涵蓋」的程式。不管什麼理由，總之 VDMAD 並沒有虛擬化那種「直接由 ring3 保護模式程式發起」的 DMA 傳輸。這些程式需要的是一種較少人知道的所謂 Virtual DMA Service（簡寫爲 VDS）。VDS 的基本目標是提供一種方法，讓保護模式應用程式能夠經由軟體中斷 INT 4Bh 取用 VDMAD。

Virtual DMA Services 的細節，可以在 ***Virtual DMA Services（VDS）Specification 1.0*** 中查到。此一規格可以在 MSDN 光碟的 "Specifications" 區獲得。

第 5 章

以 assembly 語言進行系統程式設計

你或許已經熟悉了 PC assembly 語言的某些形式，以及 Intel 處理器的基本架構。處理暫存器、分析堆疊內容等等技術，無疑能夠讓你居於一個有利地位。但是，和大部份我所遇到的程式員一樣，你可能希望使用更高階一點的語言（如 C），把注意力集中在程式邏輯身上，而非機器身上。這裡有一個好消息，你可以使用 C 或 C++ 實作大部份的 Windows 95 系統程式，使用的工具是 Microsoft Windows 95 DDK 套件或 Vireo Software 公司的 VToolsD 套件。事實上，我將儘量以 C 完成本書範例程式。

但是關於語言，也有一個壞消息：爲了閱讀系統程式設計的相關技術，你還是需要瞭解 assembly 語言，有時候你甚至必須以 assembly 語言撰寫或多或少的專案內容。此外，你還必須瞭解 Intel 80386+ CPU 的數個神秘特性：

- 真實模式、保護模式和 V86 模式之間不同的定址方法
- 如何使用 16 位元和 32 位元暫存器
- Intel CPU 如何處理中斷（interrupts）

- Windows 95 如何使用各種系統層面的暫存器和資料結構，像是 task state register 和 descriptor table 。

以 assembly 語言來解釋機器層面的系統程式設計，會花掉許多篇幅，遠多於本章所能容納的程度。這一章只不過是 Intel CPU 32 位元保護模式程式設計的一個入門，終極的參考資料應該是 Intel 程式員參考手冊。我推薦 i486 手冊，它似乎比其兄弟手冊有較好的組織和較少的錯誤。Igor Chebotko 的 ***Assembly Language Master Class***（WROX Press，1995）是一本相當不錯的書籍，其中也介紹了許多系統程式設計的主題（不限定在 Windows 環境）。

定址模式（Addressing Modes）

如你所知，assembly 程式中的許多指令都會用到記憶體運算元（operand）。Intel CPU 產生「記憶體運算元的位址」的方法是：將 segment 暫存器和另一個（或兩個）general 暫存器組合起來。Intel 80386+（及其相容品）提供真實、保護和 V86 模式三種定址模式。CPU 將根據它自己處於哪一種模式而來解釋「記憶體位址中的 segment 部份」。

真實位址（Real-Address）模式

在真實位址模式（或所謂真實模式）中，CPU 是以和 8086/8088 晶片相同的方式來組合位址，也就是說，將置放於 segment 暫存器中的 physical paragraph 加上一個來自指令運算元的 offset（圖 **5-1**）。獲得的結果，理論上可以超過 1MB。IBM PC 工程師保留了前 640KB 位址給作業系統（MS-DOS）和應用程式使用，剩餘的留給 BIOS 或擴充卡上的記憶體使用。

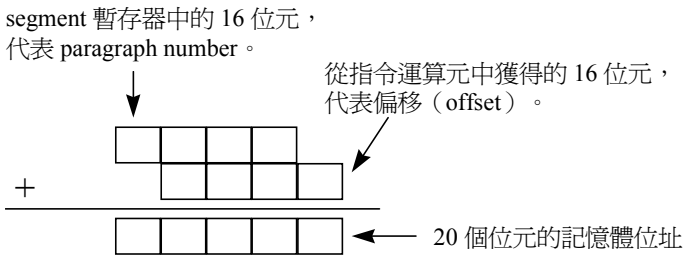


圖 5-1　眞實模式中的位址計算

從你開機開始，電腦就進入真實模式，直到 Windows 95 啓動或其他保護模式程式開始執行。記憶體管理程式如 EMM386 之流就是保護模式程式，所以切離真實模式的時機可能很快來臨；不過 BIOS 的開機自我測試（power-on self test，POST）和 MS-DOS 的初始化模組還是得在真實模式中執行。

應用軟體開發人員，特別是遊戲軟體設計者，有時候比較喜歡以真實模式做爲他們的工作平台，因爲比較沒有行爲上的限制。真實模式程式可以存取任何暫存器以及 1MB 之內的任何位址上的記憶體，沒有任何束縛！它可以執行幾乎任何指令。這樣的自由度使圖形介面遊戲軟體可以獲得最大的效率。不過即使遊戲軟體也需要大量記憶體才能玩他們的魔術，而真實模式只提供最多 640KB，這一點是誘使許多軟體決定進入保護模式的原因。

保護模式（Protected Mode）

在保護模式中，位址的形成和真實模式並不相同（圖 **5-2**）。Segment 暫存器中內含的是 selector，這麼稱呼是因爲它用來在 descriptor table 中選擇一個 descriptor。有兩個 descriptor tables：**global descriptor table**（GDT）和 **local descriptor table**（LDT）。Selector 之中有一個欄位用來指示它所索引的是上述哪一個表格。GDT 內含的 descriptors 屬於系統或全體 VMs 共有，LDT 內含的 descriptors 則屬於目前的 VM 所有。

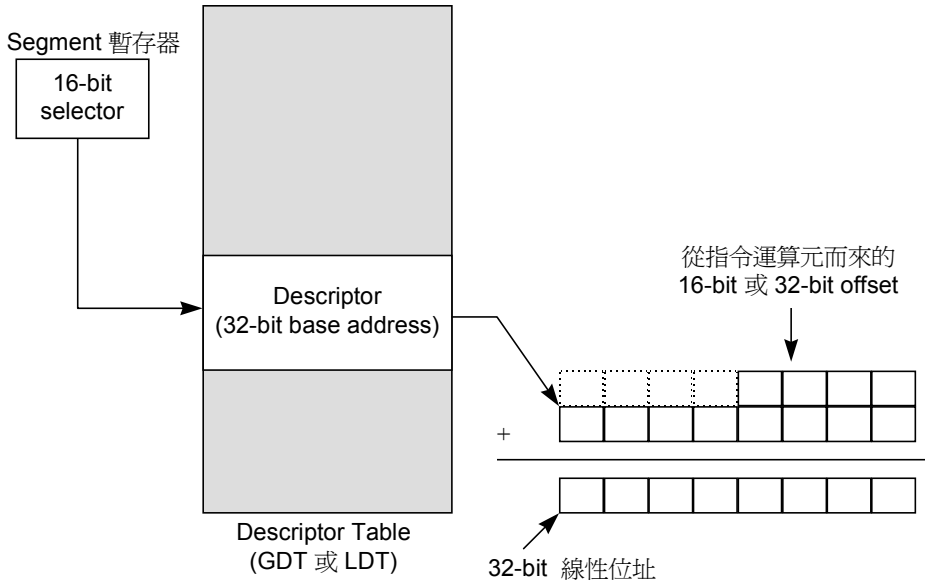


圖 5-2 保護模式中的位址計算

Descriptor 內含一個 **base** 欄位，一個 **limit** 欄位，以及一些 **access control flag**s，用來指示可被允許的存取型態（圖 **5-3**）。指令的執行過程包括「組成一個虛擬位址（有時稱爲線性位址，linear address）」，方法是把 base 欄位加上一個由指令運算元得來的 offset。由於 base 是 32 位元，所以保護模式可以存取高達 4GB（2^{32} bytes）的記憶體。目前的電腦不可能安插 4GB 記憶體，所以作業系統使用一個磁碟置換檔（disk swap file）暫時接受實際記憶體無法容納的資料。因此雖然電腦只有 8MB 或 16MB，應用程式卻可能取用 4GB 虛擬記憶體。不過，作業系統必須能夠把 page（虛擬記憶體的運作單位）搬移到實際記憶體的任何一個地點才行。虛擬位址對應用程式而言是不變的；將它轉換爲實際位址，是 CPU 的責任 。轉換方法是使用 page tables。轉換過程在圖 **5-4** 顯示。

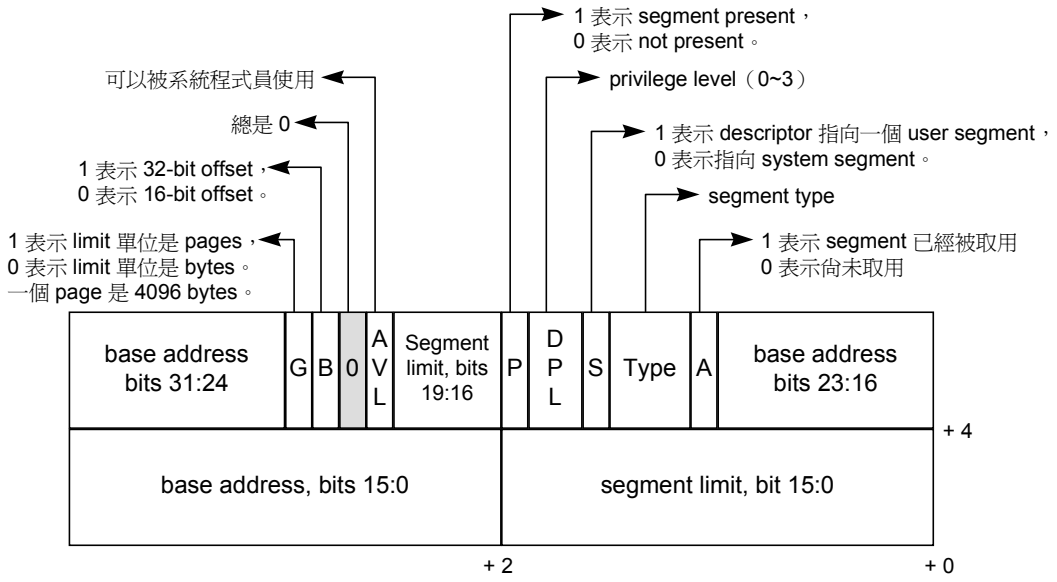


圖 5-3　Segment descriptor 的格式

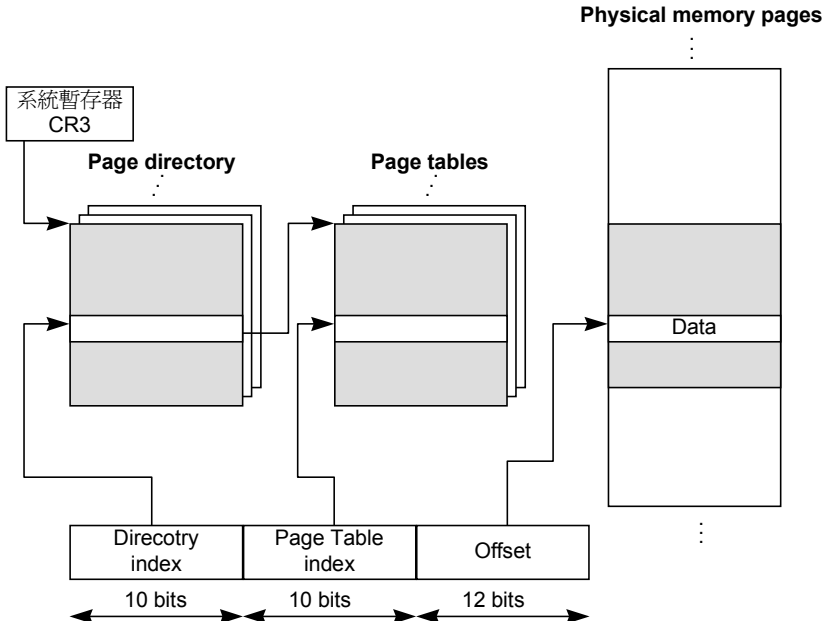


圖 5-4　虛擬位址至實際位址的轉換過程

術語的使用 當你大量接觸 assembly 語言保護模式程式設計，你有可能漸漸對術語的使用草率起來。嚴格地說，一個 segment 暫存器持有一個 segment selector，用來指出 GDT 或 LDT 中的一個 segment descriptor。爲了精準表達，你不應該使用像「一個 selector 的 limit 欄位」這樣的說法，因爲 limit 是 descriptor 的欄位，不是 selector 的欄位。我希望提醒你，不要因爲這本書而以爲我是一個賣弄學問、只重瑣屑細節的窮酸書生。我會使用比較草率的字眼，除非我需要精準表達我的意念。無論如何，在 selector 和 descriptor 之間，存在一種一對一的關係，而我年輕時學到的異種同型現象（isomorphism）告訴我們，在大部份情況下，一對一的映射關係，可以視爲完全對等（identity）的關係。所以或許 selectors 和 descriptors 之間的混淆其實並不是那麼嚴重啦！

Segment Access Controls

保護模式的「保護」來自於 CPU 執行指令時，自動完成的數個合法性檢驗動作。這些合法性在「用以描述此一 segment」的 descriptor 中有所指定。S 位元和 Type 欄位（圖 **5-3**）提供了第一層控制。只有所謂的 **user segment**（S 位元爲 1）才能直接被軟體取用；其他的 segment 屬於 **system segment**。你不需要知道 system segments 太多細節，除非你要建立作業系統內部核心。User segments 包括：內含程式指令的 **code segment**s，以及內含程式所需資料的 **data segment**s（請看圖 **5-5**）。沒有任何程式可以改變 code segments 的內容，也沒有任何程式可以把 data segments 內的資料當作指令來執行。只有當 segment 擁有 "readable" 屬性時，程式才可以從 code segment 讀取內容；只有當 segment 擁有 "writeable" 屬性時，程式才可以將資料寫入 data segment 中。一般而言 Windows 95 會爲 code segment 產生 "readable" 屬性，爲 data segment 產生 "writeable" 屬性，不過如果應用程式或 VxD 想要，它們可以忽略這些屬性不計。Type 欄位中的其他屬性，包括 Conforming 和 Expand-down，將在稍後說明。

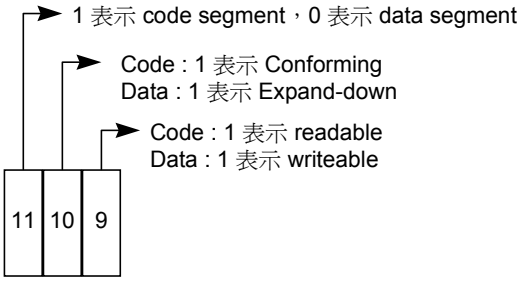


圖 5-5　一個 user segment 的 descriptor 中的 Type 欄位

除了檢驗指令的存取型態（讀、寫、或執行）對其目標物是否適當之外，CPU 還必須確定存取動作的範圍涵蓋在「此 segment 所佔有的虛擬記憶體」中。Descriptor 的 Segment Limit 欄位（圖 **5-3**）是一個 20 位元數值，通常爲 segment 的大小減 1。例如一個 64KB segment 的 Limit 欄位爲 0FFFFh；CPU 因而不允許任何指令使用 64KB 以上的 offset 來存取此 segment 資料，因爲那將超越 segment 大小。爲了供應大於 1MB（2^{20} bytes）的 segments，Intel 以 descriptor 的 G 位元（圖 **5-3**）表示計量單位爲一個 page。如果 G 位元設立，表示 limit 的計量單位是 4096 bytes；如果 G 位元清除，表示 limit 的計量單位是 1 bytes。因此，如果 limit 欄位內容爲 FFFFFh，當 G 位元設立，表示這是一個 4GB segment；當 G 位元清除，表示這是一個 1MB segment。

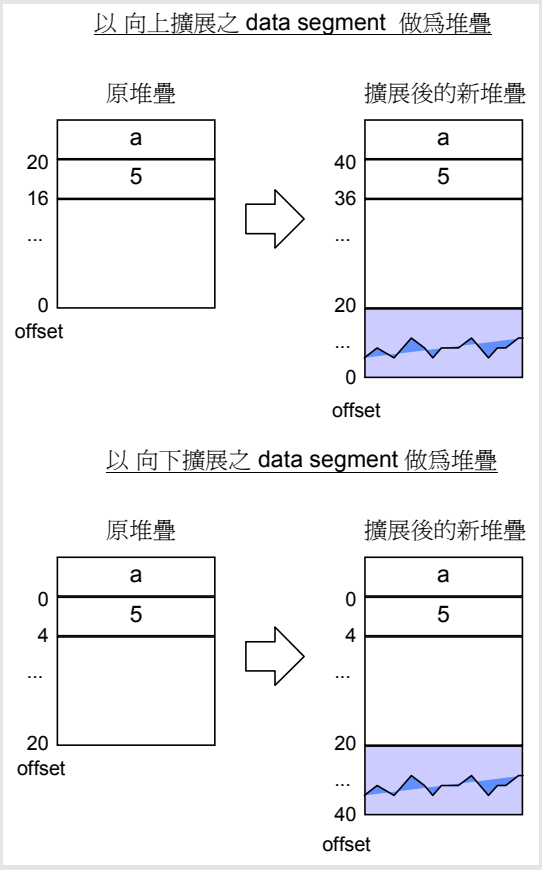
檢驗 segments 的大小，可以確保記憶體運算元落在 offset 0 和 segment 尾端之間。不過 data segment 可以設立 Expand-down 屬性（圖 **5-5**），表示這個 segment 的 limit 是「可允許之最低 offset - 1」，而不是「可允許之最高 offset」。Windows 95 利用這個十分聰明的觀念來誘捕（trap）32 位元程式對於 NULL 以及 NULL 指標的使用。如果程式有一個 data selector，base 爲 0，limit 爲 00FFFh，並且其 Expand-down 屬性設立，那麼 CPU 將不允許程式使用小於 4096 的資料指標來取得資料。這樣的誘捕（trapping）動作可以保護程式不要直接被一個 NULL 指標取得內容，或使程式避免存取某個結構中的欄位 -- 如果這個欄位的 offset 比 4096 bytes 小的話！

高雲慶註：這裡我對 expand-down 屬性再做一點補充。絕大部份的 data segment 的 descriptor 的 E 位元都是 0，代表此 segment 爲向上擴展（**expand-up**）。但 386 機器爲了滿足以下兩種堆疊的需求，提供有向下擴展（**expand-down**）的 data segment：

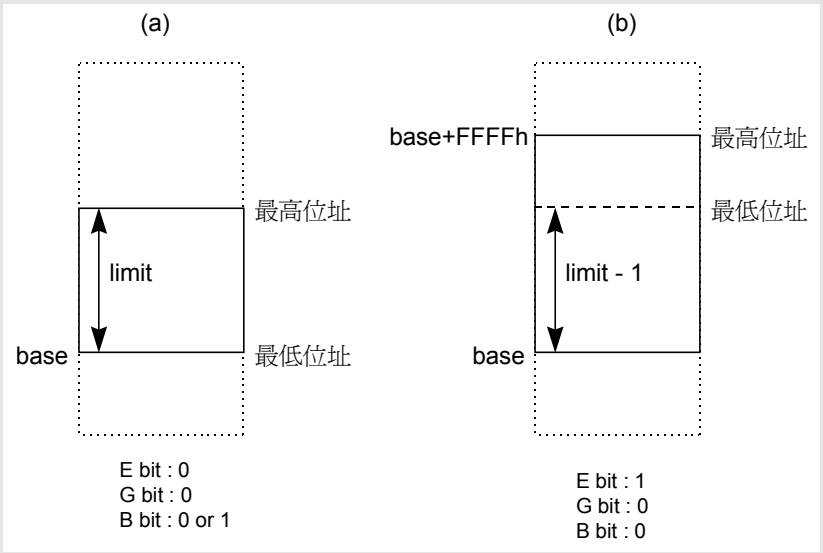
1. 程式中的 stack segment 和 data segment 是分開的（DS 和 SS 的值不同）

2. 堆疊擴展方式係將整個內容拷貝至較大 segment，而不是在尾端附加新 pages。

下圖可以說明「向上擴展」和「向下擴展」的差異。由圖可知，向上擴展之 segment 若用於堆疊，並以拷貝方式來擴展，堆疊內容的 offset 會變動，導致相同的 SP 卻參考到不同的內容，影響了程式的進行。向下擴展的 segment 則無此困擾。



此外，向上擴展與向下擴展的 segment 在計算最高位址與最低位址時，方式亦有不同。以下圖爲例：



(a) 代表向上擴展之 segment，其 base 即爲最低位址，(base+limit) 則爲最高位址。如果要擴大 segment，只要將 limit 值加大，節區即向上擴展。

(b) 代表向下擴展之 segment，其最高位址在 (base+FFFFh) 處，最低位址在 (base+limit-1) 處。換句話說此時的 limit 代表一個 64K segment 中「不可用區域」的大小。若要擴大節區，把 limit 的值降低，節區就往下擴展，這也是它之所以被命名爲「向下擴展」的原因。

注意 **NULL 指標** Windows 3.1 的 Win32s 子系統以一種和 Windows 95 不同的方法來保護 NULL 指標。在 Win32s 中，32 位元程式使用的 code selectors 和 data selectors，其 base 爲 FFFF0000h，其 limit 爲 FFFFFFFFh。由於並未安排 page tables 將最後 64KB 映射到位址空間中，所以如果使用 NULL 指標，保證會引起 page faults。如果使用的 32 位元指標，其值大於或等於 1000h，就可以轉折爲「從 0h 開始的合法線性位址」。不幸的是，這個 page faults 的處理並不一直很順利，因爲早期 Win32s 面對這個 fault 會當掉，而不是正確地將這個 page fault 視爲一個 NULL 指標參考動作。此外如果你的 VxD 希望供應 32 位元線性位址給 32 位元 client 程式，並且相容於 Windows 95 和更早的 Windows 版本，那麼你還必須知道不同的 segment base 位址。

利用 NULL 指標來取用資料，在 16 位元程式上也會引起 faults，不過 Windows 95 依賴一種不同的保護手段來誘捕它們。你不能夠使用 0 selector 來表示一個記憶體運算元。16:16 NULL 指標是 0000h:0000h，所以嘗試藉由一個 NULL 指標取得內容，或是存取某個結構中位址爲 NULL 的欄位，會引發一個 fault。上述所謂 16:16 表示法是指一個 16 位元的 paragraph（或 selector）和一個 16 位元的 offset。

General Protection Fault（GPF）

我一度極不明白 CPU 如何在保護模式程式上強制加諸「存取限制」。現在我明白了，違反存取規則的報應通常是獲得一個 general protection fault（GP fault）。GP fault 是 Intel CPU 的一支瑞士和平部隊。產生這種異常的原因非常非常多，其中也許代表程式臭蟲，但也有許多發生在正常程式身上。以下簡列一些 GP faults 的發生原因：

- 「爲一個 segment 暫存器（如 DS, ES, FS, GS）而設定」的 data selector 不合法；或者該 selector 對應的 descriptor 的 privileged level 比現行的程式更高權級（譯註：也就是說現行程式沒有資格使用那個 segment）。
- 企圖以一個 16:16 NULL 指標（其 selector 爲 0）取用某些資料。是的，你可以爲一個 segment 暫存器設定一個 NULL selector，但是你不能夠以它來取用資料。

- 企圖呼叫一個不同權級（privilege level）的程式。
- 企圖取用某 segment 的 limit 範圍（記錄在其 descriptor 中）以外的程式碼或資料。
- 企圖將資料寫入一個只可讀取的 data segment 中，或企圖從一個只能執行的 code segment 中讀取資料；或是執行一個 data segment，或是將資料寫到 code segment。
- 企圖在 interrupt descriptor table（IDT）範圍之外引發一個中斷。只要程式發出軟體中斷 60h 以上，就一定會引起 GP fault。這並不一定代表程式有錯。
- 任何保護模式程式企圖執行一個 CLI 或 STI 指令。這會不時發生；Windows 會誘捕這個 fault 並爲受影響之虛擬機器設立虛擬中斷旗標（virtual interrupt flag）。

不吉利的數字 雖然有點突兀，但我要說一件很有趣的事情。請注意，GP fault 就是 CPU 0Dh exception，這是傳統的不吉利數字（譯註：西人視 13 爲不吉利），IBM 用此數字在 OS/360 上表示 supervisor call 的不正常結束（ABEND 或 SVC 13）。

Privilege Ring

Intel CPU 提供四種權級（**privileged level**），通常稱爲 **privilege ring**s，爲的是將某些需要系統資源的軟體，以「被控制的程度」加以區隔。Windows 只使用兩種權級：作業系統在 **ring0** 執行，這是權力最大的等級。Ring0 碼可以改變任何位址的記憶體內容，也可以改變任何暫存器。應用程式在 **ring3** 執行，這是權力最小的等級。Ring3 程式不能夠存取系統控制暫存器（譯註：例如 CR0 ~ CR3），也不能夠讀寫被作業系統規劃爲保護區的記憶體。CPU 可以誘捕（trap）ring3 程式的某些動作（例如稍早所說的 CLI 和 STI 指令），讓 supervisor 得以模擬那些動作，或是得以控制應用程式如何（以及何時）執行它們。

爲了讓你瞭解稍後對於權級（privileged level）的討論，我必須先介紹一些略顯複雜的術語。首先要說的是，Windows 95 使用 CPU 權級，使你幾乎可以忽略掉所謂的複雜性。

Selector 的最低兩個位元（圖 **5-6**）內含所謂的 requested privileged level（**RPL**），而一個 segment 的 descriptor 內含 descriptor privileged level（**DPL**），理論上可以不同於對應之 selector 的 RPL。CS segment register 中的 RPL 又有些特別：它被稱爲 current privileged level（**CPL**），代表目前正在執行的程式的權限等級。

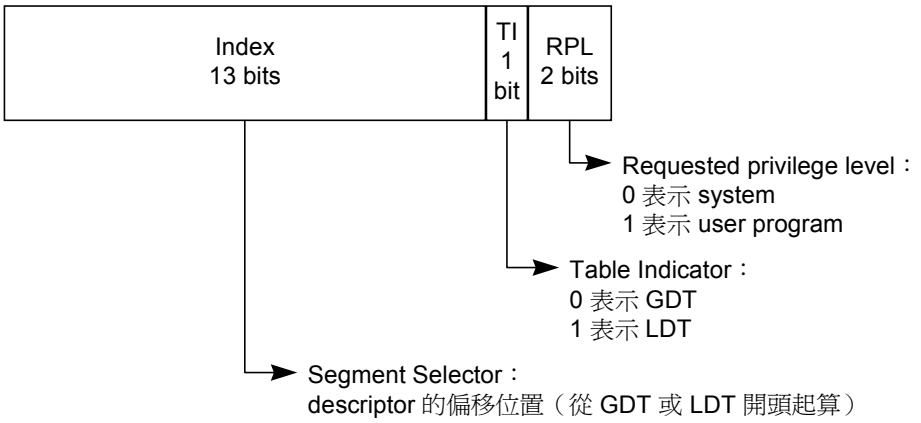


圖 5-6　Selector 的格式

當一個程式以 CPL 權級執行，如何載入相同或不同 RPL / DPL 的 selectors 呢？這之間有一些值得詳細敘述的規則。（還記住先前我曾提過有關於 selector 和 descriptor 的不同嗎？你我現在都應該清楚，RPL 是 selector 的屬性，DLP 是 descriptor 的屬性，但我無法忍受這樣冗長的敘述。）如我所說，在 Windows 95 中大部份時候你可以忽略這些規則，因爲一個 ring0 程式總是以 CPL 0 執行之，並總是能夠使用 RPL 0 和 DPL 0。應用程式則總是以 CPL 3 執行之，並總是能夠使用 RPL 3 和 DPL 3。Ring0 程式總是使用 GDT 而應用程式總是使用 LDT。因此，你會發現系統軟體使用 code selector 28h（一個 ring0 GDT selector）和 data selector 30h（也是一個 ring0 GDT selector），而應用程式使用 xxx7h 或 xxxFh 型式（表示 ring3 LDT selectors）的 code selector 和 data selector。

CPU 不讓程式處理那些「屬於另一程式所有，而又不在相同權級」的 code 和 data。Ring3

code 不能夠處理 ring0 data，也不能夠直接呼叫一個 ring0 程式。爲了和作業系統溝通，ring3 程式必須引發中斷以觸發 ring 的轉換。Intel 架構提供所謂的 **gate descriptor**（一種 system segment）以及所謂的 **conforming code segment**s（它和一般 code segment 不同之處在於，其 descriptor 的 Type 欄位的 Conforming 屬性是設立的），兩者都可以被 ring3 程式直接呼叫。不過 Windows 並未使用這兩種特性。

Ring0 程式可以處理任何權級的 data，但不能夠直接執行任何其他權級的 code。若要執行 user-mode code，ring0 supervisor 必須在一個特殊格式的堆疊上執行 IRET 指令，該堆疊內含執行對象的完整環境（context）。

虛擬 8086 模式

虛擬 8086（V86）模式是一種爲 MS-DOS 及其他真實模式程式而存在的保護模式。在 V86 模式中，CPU 以和真實模式相同的方法來組合一個位址：將放在 segment 暫存器 中的 paragraph number 與指令之運算元所指出的 offset 搭配在一起。然而和真實模式有一點不同，這 20 個位元是虛擬位址而非實際位址。因此，V86 位址必須經歷和保護模式位址一樣經歷過的 page 轉換。這一事實使 Windows 和其他系統軟體能夠把真實模式程式所能存取的 1MB 位址空間映射到 extended memory 中。使用 paging，使程式能夠經由 1MB 位址來存取 extended memory，是 QEMM, 386MAX 及其他 80386 記憶體管理程式的基本技巧，而它也構成了 Windows 95 MS-DOS VM 的基礎。

如有必要，作業系統可以非常密切地管理 V86 程式。從大部份角度來看，V86 程式都是在 ring3 執行，這意味著若要使用 privileged（高權級）指令，或用到 privileged（高權級）暫存器或資料，會引起 General Protected（GP）faults。Supervise 可以設定 EFLAGS 暫存器中的 I/O privilege level（IOPL）位元，讓它小於 3，於是讓一些正常而無害的指令（如 INT、PUSHF、POPF、IRET、CLI、STI）產生 GP faults。Intel 意圖讓作業系統使用這一特性來虛擬化中斷旗標（interrupt flag）。你大概注意到了，上述指令的共同點就是它們會儲存或復原中斷旗標。Microsoft 發現如果這麼做的話，MS-DOS 程式大約會慢 15%。因此 Windows 95 在執行 V86 模式時通常將 IOPL 設爲 3，除非它很特別地

企圖誘捕（trap）一個 VM 的 IRET 指令。所以一個 MS-DOS 程式可以在「中斷被 disabled」的情況下因執行一個份量很重的迴圈而使電腦停擺。PS/2 架構支援一種所謂的 watchdog timer，可以在這種情況下繼續產生中斷，但是標準 PC 架構卻使電腦曝露在這種程式錯誤之下！

偵測 V86 模式

如果你知道你的程式只能夠在 MS-DOS 底下跑，那麼你可以使用冷僻的 SMSW 指令來分辨真實模式和 V86 模式。這是一個 80286 指令，不過在 80386+ 中還是存在。它會取得 CR0 暫存器的較低部份，而其中的最低位元即用以指示是否處於保護模式：

```
pe_mask equ 1            ; protected-execution bit in CR0
smsw    ax               ; capture low-order part of CR0
test    ax, pe_mask      ; test PE-bit
jnz     InV86Mode        ; branch if in protected mode
```

爲了回溯相容，SMSW 指令並不是 privileged（高權級），也就是說 ring3 程式也可以執行它。上述程式碼背後的邏輯是：我們假設程式只能夠在 MS-DOS 底下跑（意味著它在真實模式或 V86 模式下跑）；此時如果 PE 位元設立，CPU 不可能處於真實模式，那麼必定處於 V86 模式。

如果你知道你的程式能夠在 Windows 底下跑，而你想要分辨 V86 模式和保護模式，就不能夠使用上述技巧，因爲 SMSW 指令會告訴你說你在保護模式之下。這時候你必須使用 INT 2Fh, function 1686h：

```
mov     ax, 1686h        ; function 1686h
int     2Fh              ; detect mode
test    ax, ax           ; any bits set ?
jz      ProtectedMode    ; if no, program is in protected mode
```

Intel 希望，對程式而言，V86 模式儘可能透明。事實上，沒有什麼直接方法可以讓程式知道它處於 V86 模式。這是真的，甚至雖然 EFLAGS 暫存器中有一個位元（VM 位元，**圖 5-8**）可以控制 CPU 是否處於 V86 模式。不論是好是壞，Intel 實作了一個 PUSHFD 指令，把 EFLAGS 暫存器內容壓進堆疊之中，然後讓 VM 位元爲 0。所以，下面這些碼不能夠有效運作（如果你在 16 位元程式中使用 32 位元暫存器的功力不足，請參考下一節）：

```
pushfd                  ; push EFLAGS register onto stack
pop     eax             ; pop EFLAGS image into EAX
bt      eax, 17         ; test VM bit in EFLAGS image
jc      InV86Mode       ; branch never taken!
```

雖然對程式而言我們希望 V86 模式是透明而無影響的，但有時候你並不想要這樣的透明度，特別是當你要引發一個由 Windows 95 作業系統提供的服務時。你可以從一個 V86 程式中執行一個 ARPL（Adjust Requested Privilege Level）指令，它會引發一個 "invalid operation" CPU exception。Virtual Machine Manager 於是會檢驗這個 exception 的 segment:offset 位址，並把控制權交到某些 VxD 服務常式手中。IBM 在 CP-67 和 VM/370 身上使用相同技巧，兩者的 VM 碼都以高權級（privileged）的 Diagnose 指令與控制程式溝通。

32 位元程式設計

使用一部 80386+ 機器和一套如 Windows 95 之類的作業系統，主要是希望獲得 32 位元程式設計的重要特性。這些特性包括效率的改善（對 end user 有直接幫助），以及程式設計上的方便（對 end user 雖沒有直接幫助，卻能夠讓他們及時收到較好的軟體產品）。32 位元和 16 位元程式設計的基本不同在於擴充暫存器的使用。擴充暫存器中放的是 32 位元的指令運算元和位址。此外，16 位元程式常常使用四種記憶體模式（small, compact, medium, large）之一，視你有沒有超過 64KB 的 code 或 data 而定；32 位元程式通常使用 flat 模式，可以輕易處理 4GB 的 code 和 data，不需重新設定 segment 暫存器。

擴充暫存器（Extended Registers）

在 80386+ 中，程式可以處理 8 個 general purpose 暫存器（有時候簡稱爲 general 暫存器），和六個 segment 暫存器（圖 **5-7**），以及一個和記憶體模式無關的 flags 暫存器（圖 **5-8**）。General 暫存器和 flags 暫存器都是 32 位元，assembly 程式員以大寫的 E（表示 extended）放在暫存器名稱前面做爲記號。例如，要處理 EAX 暫存器，程式員可以這麼寫：

```
xor     eax, eax   ; clear 32-bit extended AX register
```

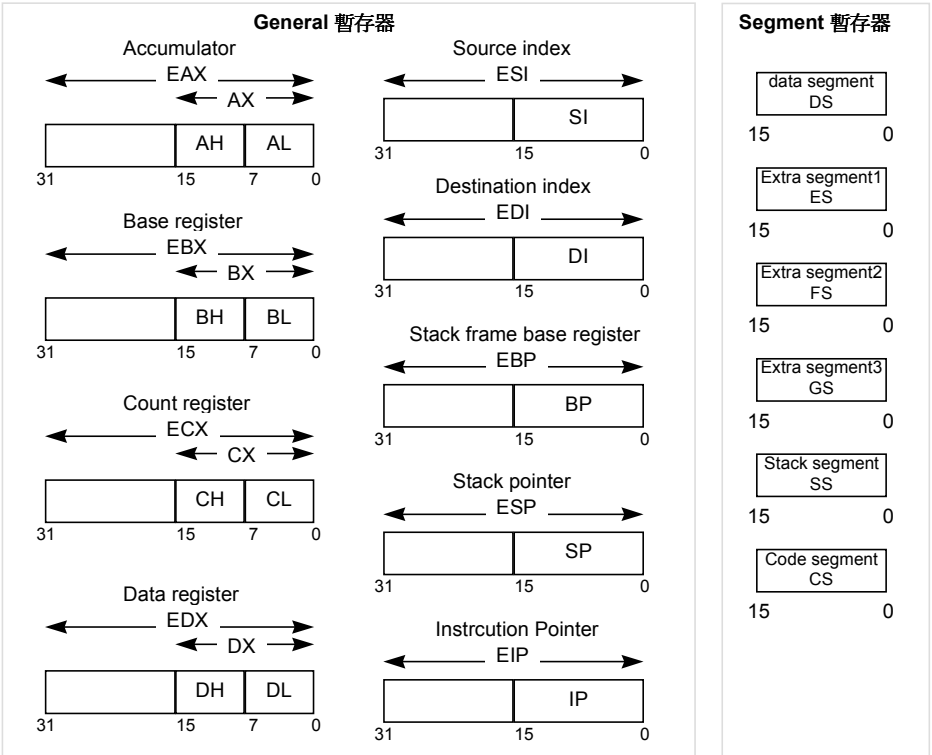


圖 5-7　General 暫存器和 segment 暫存器

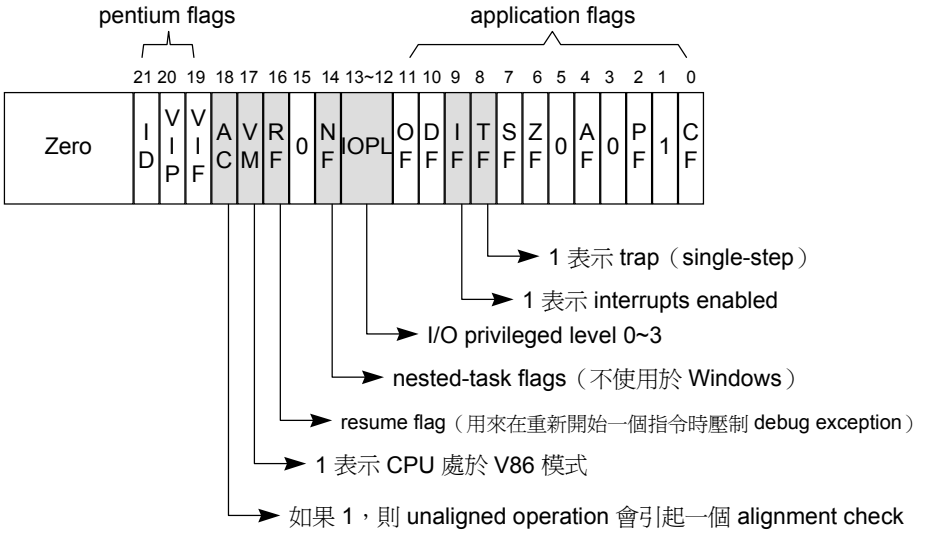


圖 5-8　flags 暫存器

我們在 16 位元程式中所使用的 16 位元暫存器只不過是對應的「32 位元擴充暫存器」的後半部而已。AX 暫存器代表的是 EAX 暫存器的較低 16 位元。同樣道理，8 位元暫存器如 AH 和 AL 也佔用擴充暫存器的一部份。32 位元碼中大量存在那種「在 8 位元、16 位元、32 位元暫存器之間搬移資料」的指令，下面是三個例子：

```
movzx   ecx, al      ; zero-extend 8 bits from AL to ECX

cdq                  ; sign-extend EAX into EDX:EAX

rol     eax, 16      ; interchange high and low halves of EAX
```

這或許已經明白顯示出使用 32 位元指令的理由了。是的，32 位元指令更容易做 32 位元數值的運算！例如，以 16 位元指令把兩個 32 位元數值加起來，要這麼做：

```
mov     ax, word ptr a          ; low-order part of "a"
mov     dx, word ptr a + 2      ; high-order part of "a"
add     ax, word ptr b          ; accumlate 32-bit result
add     dx, word ptr b + 2      ; (including carry from low-order)
mov     word ptr c, ax          ; low-order part of sum
mov     word ptr c+2, dx        ; high-order part of sum
```

若以 32 位元指令就簡單多了：

```
mov     eax, a
add     eax, b    ; add a + b
mov     c, eax    ; store the result
```

如果你只能使用 16 位元暫存器，那麼就必須寫一個十分複雜的軟體模擬器，模擬「不斷重複的硬體減法演算」（repeated-subtraction hardware algorithm），才能完成一次 32 位元除法。但使用 32 位元指令來做兩個 32 位元長整數除法，簡直是輕而易舉：

```
mov     eax, dividend
cdq                         ; sign-extend the dividend
idiv    divisor             ; divide it by the divisor
mov     quotient, eax       ; store the result
```

32 位元定址模式也能夠使你的生活比較容易一些，因爲你不再需要在跨越 64KB 邊界時重新設定 segment 暫存器的值。舉個例子，有一些資料結構在 Windows 中大於 64KB（例如 bitmap 圖），在 16 位元程式中你必須宣告 huge 指標才能夠存取它。Windows 的 *GlobalAlloc* 會指定連續的 selectors（就是像 7E7h, 7EFh...等等間隔爲 8 的 selectors）來涵蓋整塊記憶體，而程式中爲了處理資料所產生的 selector 亦將如預期般運作。假設你想要計算一張 256 色 bitmap 中的黑色圖素有幾個，並假設 bitmap 中數值爲 0 的圖素代表黑色，那麼你可以 C 語言完成這份工作如下：

```
LONG CountBlack(BYTE __huge * lpBits, LONG nbits)
   {                        // CountBlack
   LONG result = 0;
   LONG i;
```

```
    for (i=0; i< nbits; i+)
        if (!lpBits[i])
            ++result;

    return result;
    }                       // CountBlack
```

如果你事先不知道 C 編譯器會把對 lpBits[i] 的取值動作以什麼樣的 assembly 碼在 16 位元平台上表現出來，那麼你可能會大吃一驚：

```
mov     ax, WORD PTR i
mov     bx, WORD PTR i+2
mov     cx, WORD PTR lpBits
mov     dx, WORD PTR lpBits+2
add     ax, cx
adc     dx, 0
mov     cx, OFFSET __AHSHIFT
shl     dx, cl
add     dx, bx
mov     bx, ax
mov     es, dx
mov     al, BYTE PTR es:[bx]
```

其中 *__AHSHIFT* 是一個外部常數（其實是個匯入符號），其值爲 3。這段程式碼的淨效益是把 (i / 65536) * 8 加到 *lpBits* 記憶體區塊的第一個 selector 去，並使用 (i % 65536) 做爲該 selector 的 offset。如果你對於爲什麼 *BitBlt* 在以前的 Windows 版本中是如此之慢感到奇怪，現在知道原因了吧！

上述程式的 32 位元版會產生明顯較小較快的碼：

```
mov     eax, DWORD PTR i
mov     ecx, DWORD PTR lpBits
xor     edx, edx
mov     dl, BYTE PTR [eax+ecx]
```

並不是只有在面對大塊記憶體時，32 位元定址方式才使你輕鬆些，事實上它也幫你管理陣列。原本的 808x CPU 架構嚴重限制了你組成位址的方法，使你只能夠將 base 暫存

器和 index 暫存器與指令中的運算元（被當做 offset）組合，形成位址。但在 32 位元位址中，你可以使用任何暫存器做爲 base 暫存器，並使用除了 ESP 以外的任何暫存器做爲 index 暫存器。猶有進者，你還可以將 index 暫存器乘以 1 或 2 或 4 或 8。上一段碼示範的是「將暫存器組合，形成一個位址」，下面是另一個實例，示範以乘法對陣列做索引動作（陣列的每個元素都是 4 bytes）：

```
mov     ecx, array              ; ECX points to base of long[] array
mov     edx, i                  ; EDX = loop index (0 to n)
mov     eax, [ecx + 4*edx]      ; EAX <- i'th element of array
```

USE32 和 USE16 Segments

雖然 general 暫存器的長度是 32 位元，程式卻並不一定非得處理其中全部位元不可。從現行的 code segment 所對應的 descriptor 中的 bitness（16 位元或 32 位元），可以得知運算元和指令位址究竟預設爲 16 位元還是 32 位元。在一個所謂 USE32 segment 中，指令係使用 16:32 位址（一個 16 位元的 selector 加上一個 32 位元的 offset），並處理 32 位元運算元。而在一個所謂 USE16 segment 中，指令係使用 16:16 位址（一個 16 位元的 selector 加上一個 16 位元的 offset），並處理 16 位元運算元。不論哪一種情況，爲一個指令而產生的虛擬位址都是 32 位元，那正是 segment descriptor 的 base 欄位的長度。80386+ 保護模式的 16 位元定址模式和 32 位元定址模式之間的唯一差異是，在 base 位址之上究竟要加多少位元（16 位元或 32 位元）的 offset。

爲了說明「code segment 是多少位元」的重要性，請看看機器碼 8B 07。即使你有能力對 Intel 指令編碼，你也沒辦法一下子就知道 8B 07 到底是什麼動作。在一個 USE16 segment 中，它代表的是：

```
mov ax, [bx]   ; 16-bit address, 16-bit operand
```

而在一個 USE32 segment 中，它代表的是：

```
mov eax, [edi]   ; 32-bit address, 32-bit operand
```

位址和運算元的 "Size Override Prefixes"

指令的「前置操作」（prefix operations）允許程式員在一個指令的生命期內更改預設的運算元和（或）位址寬度。因此，在指令流中安插一個 operand-override prefix 66h，可以使接下來的指令使用另一種運算元寬度；而安插 address-size override 67h，可以使接下來的指令使用另一種位址寬度。在 USE16 segment 中，我剛剛提過的 8B 07 指令可以代表四種動作，視 override prefixes 存在與否而定：

```
8B 07          mov ax, [bx]         ; no overrides
66 8B 07       mov eax, [bx]        ; operand-size override
67 8B 07       mov ax, [edi]        ; address-size override
67 66 8B 07    mov eax, [edi]       ; both overrides
```

在 USE32 segment 中，則有以下四種可能性：

```
8B 07          mov eax, [edi]       ; no overrides
66 8B 07       mov ax, [edi]        ; operand-size override
67 8B 07       mov eax, [bx]        ; address-size override
67 66 8B 07    mov ax, [bx]         ; both overrides
```

這是否意味，如果你不想要預設的位址寬度和運算元大小，你就必須明白地安插 override prefixes 到程式中？不！幸運的是你的組譯器知道如何使用 override prefixes，使你的日子輕鬆一些。首先你必須宣佈你的 code segment 是 16 位元（USE16）還是 32 位元（USE32）：

```
_TEXT  SEGMENT BYTE PUBLIC USE16 'CODE'
```

或

```
_TEXT  SEGMENT BYTE PUBLIC USE32 'CODE'
```

從此以後，程式碼中的運算元和位址就會使用對應的暫存器（32 位元碼使用擴充型暫存器，16 位元碼使用非擴充型暫存器），一如前面的範例所示。組譯器會將必要的 prefixes 放在正確的位置，確保執行時的正確性。

請特別注意，有一些指令，包括 PUSHF、PUSHA、MOVS 等等，有不明顯的暫存器運算元。如果你希望組譯器爲它們產生 32 位元版機器碼，你得明白地在這些運算子後面加一個 'D'（也就是 PUSHFD、PUSHAD、MOVSD 等等），甚至即使在 USE32 segment 中也必須如此。我曾經看過某個商用 VxD 有個駭人的錯誤，它在應該使用 PUSHAD/POPAD 的地方竟然使用 PUSHA/POPA。不要在你自己身上發生這種菜鳥錯誤呀！

16 位元程式碼中的 32 位元運算元和 32 位元位址

在 16 位元程式中使用 32 位元暫存器可能會帶來一些好處。某些 16 位元 C 編譯器會爲 long 運算元產生 32 位元整數運算 -- 如果它知道這個程式將在 32 位元硬體平台上執行的話。此外，在一個保護模式 Windows 程式中使用 32 位元位址，對於 __huge 資料陣列如 bitmaps 也很有用。

在真實模式中使用 32 位元位址（有些人稱作 "big real mode"）或許看起來比較沒道理，其實不然。要知道，CPU 會 "cache" 每一個 segment 的 descriptor；當你正處於真實模式，那些 caches 一般而言會表現出「limit 爲 64KB 而 base 爲 20 位元 physical paragraph 位址」的一個 segment。如果你切入保護模式，將一個 selector 載入到某 segment 暫存器中然後切回真實模式，而沒有先恢復那個 segment 暫存器，那麼 CPU 的 descriptor cache 會繼續持有新的 descriptor，其 base 位址可以指向實際記憶體的任何位置（包括 1MB 以外）；於是你可以存取保護模式 selector 所指向的任何記憶體，並以 32 位元位址存取 64KB 以外的資料。然而只要 segment 暫存器一被更改（例如被一個隨時可能發生的中斷處理常式），原先被 cached 的 base 位址會重新被載入，而短暫的自由也就消失了。

注意 Microsoft 的 MASM 組譯器會根據你指定的 .286 或 .386 directive 來指定預設的 bitness（16 或 32 位元），。如果你指定 .286，組譯器便使用 16 位元暫存器和指令，它們都是從 80286 就開始使用的；所有 segments 都將是 USE16，因爲這是唯一可以在 80286 使用的 segment。如果你指定 .386，組譯器便使用 80386 所支援的 32 位元暫存

器和指令；所有的 segments 都將是 USE32，因爲這或許正是你所希望的。這種安排通常有其便利性，但也可能爲你帶來未曾預期的困擾。假設你以 .386 做爲程式的開始，然後使用 large 模式（因爲你正在寫 large-model Windows 程式），那麼你的 segment 將會不正確。爲了避免這種情況發生，請把 model directive 放置在 .386 directive 之前。

Flat Memory Model

32 位元程式通常使用 flat 記憶體模式。在 flat 模式中，CS 暫存器持有一個 code selector，其 base 爲 0，limit 爲 4GB。DS, ES, SS 暫存器則持有另一個 selector（之所以不同，因爲它必須宣告爲 data selector 而非 code selector），其 base 也是 0，其 limit 也是 4GB。（我沒有忘記「Windows 95 使用一個 Expand-down data segment，對於 ring3 程式的 limit 是 4KB」這一事實，我只是不想再加重這裡的複雜度）。因此，一個 32 位元 offset 就足以提供一切資訊，定出 4GB 虛擬位址空間的任何一點。32 位元 flat 模式的位址計算方式有時候我們稱爲 **0:32** 指標（以強調 segment selector 的無關輕重），或者稱爲 flat 指標或 linear 指標。所以 flat 模型下的程式不需要改變 segment registers 的內容。如果是 16 位元程式，就需不時改變 segment 暫存器的內容，以存取散亂各處的記憶體。在保護模式中設定一個 segment descriptor，再加上相關的所有合法性檢驗，成本十分高昂，幾乎是「只設定 flat 指標」的 7 倍。所以 large-model 16 位元程式常常比 flat-model 32 位元程式慢，就是這個道理。

系統程式設計的特質

Intel 80386+ CPU 有數個特質是只給作業系統 supervisor 使用的，這些特質包括 **control** 暫存器和 **debugging** 暫存器，以及 **system table**s 如 global/local descriptor tables 和 interrupt descriptor table。對於 Intel CPU 如何在真實模式處理中斷，以及真實模式中斷服務常式如何使用 IRET 指令讓原先的流程重新獲得控制權，你或許已經有一些認識，但是保護模式的中斷處理和真實模式之間有一些重大差異。也許你的生命中不需要對我這裡所說的東西有所瞭解，不過如果你瞭解它，你就比較能夠清楚 Windows 95 的 Virtual Machine Manager（VMM）。

Control 暫存器

Intel CPU 包含至少 4 個 control 暫存器：CR0~CR3。只有 ring0 程式才能夠存取它們。CR0 中的位元（圖 **5-9**）控制著 CPU 的運作模式（真實或保護），以及 paging 機置是否打開、算術協同處理器（math coprocessor）指令是否可以被 "trapped" 和被模擬...等等。

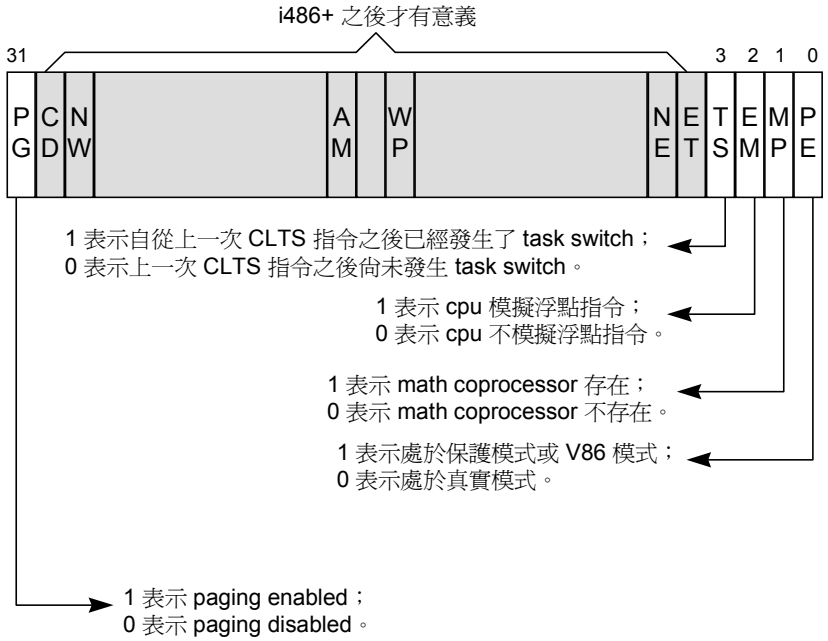


圖 5-9 CR0 的格式

CR3 內含 **page directory** 的實際位址。CPU 根據它來找到所有作用中的 **page table**s（用來把虛擬位址轉換爲實際位址，請看圖 **5-4**）。CR2 儲存一個線性位址，此位址和 page fault 的發生有關。當一個被求取的 virtual memory page 不存在於實際記憶體中，就會發生 page fault。其他 control 暫存器的編號和作用視 CPU 模式而定，Windows 並沒有特別使用它們。各式各樣的 MOV 指令可以用來存取 control 暫存器，下面是兩個例子：

```
mov     cr0, eax   ; copy eax to cr0
mov     eax, cr2   ; copy cr2 to eax
```

如果你要在 Microsoft 的 MASM 程式中寫這樣的 ring0 指令，你必須告訴組譯器說你正在使用高權級（privileged）指令，也就是在 processor-type directive 後面加上一個 'P'：

```
.386p                ;譯註：這就是 processor-type directive
mov     eax, cr0
```

如果 ring3 程式企圖讀寫 control 暫存器，會發生 general protection fault。 以 ring3 程式讀取 CR0 實在有點愚蠢，因爲不受權級限制的 SMSW 指令就已經可以把 CR0 的較低 16 位元取出了。

雖然 CPU 可以阻止 ring3 程式改變 CR0，Windows 卻允許經由 trapping 和 emulating 做某些良性變化。舉個例子，在一台擁有真正算術協同處理器（math coprocessor）的電腦上，V86 程式「將算術協同處理器之模擬動作 "enable"」的唯一方法，就是強迫 CR0 的 EM 位元設立起來：

```
em_mask equ 4            ; emulate coprocessor bit
mov     eax, cr0         ; normally privileged, but OK
or      eax, em_mask     ; enable coprocessor emulation
mov     cr0, eax         ; normally privileged, but OK
```

這種作法可以在 Windows 95 上有效運作，原因是 VMM 分析了兩個 MOV 所產生的 GP faults 之後發現，將 CR0 搬移到 EAX 是安全的，所以 Windows 爲 ring3 程式模擬此一動作；至於將 EM 位元設立起來也是安全的，所以 Windows 也模擬了將 EAX 搬移到 CR0 的動作。

Debugging 暫存器

Intel CPU 之中還有 8 個 debugging 暫存器，其中四個（DR0~DR3）內放 breakpoint 位址。另四個控制著「當程式在一個 breakpoint 處存取記憶體時，處理器如何產生 debugging exceptions」。Pentium 處理器還讓你以此法誘捕（trap）port I/O 動作。系統軟體使用特

殊型式的 MOV 指令來存取 debug 暫存器，下面是兩個例子：

```
mov     eax, dr0    ; copy DR0 to EAX
mov     dr7, eax    ; copy EAX to DR7
```

Windows 允許除錯器經由 DPMI 使用這些暫存器。系統除錯器如 WIDEB386 和 SoftIce/W 也使用這些暫存器來協助系統程式員。

Global Descriptor Table（GDT）

> 譯註：以下談到 GDT 暫存器和 IDT 暫存器和 LDT 暫存器，它們都有對應的 "store" 指令和 "load" 指令。在這裡，store 是指從 register 中取出資料，load 是指把資料放進暫存器中。store 指令可以在 ring0 和 ring3 執行，load 指令只能夠在 ring0 執行。

GDT 暫存器內含 GDT（稍早前我說過的兩個 descriptor tables 之一）的長度和虛擬位址。GDT 內含 segment descriptors（圖 **5-3**），用來描述「可被任何一個 task 使用」之每一個 segment 的起始位址和長度、型態、存取權利（access right）。Windows 係在啓動程序（boot process）中建立 GDT，並在其中內植「給 system segment 使用的 ring0 selectors」。除了特殊的 selector 40h（用來參考實際位址 0400h 中的 BIOS 資料），GDT selectors 的 DPL 通常爲 0，因此一般的 ring3 應用程式不能夠取用 GDT descriptors 所描述的 segments。只有 ring0 程式或真實模式程式可以使用 LGDT 指令改變 GDT 暫存器。倒是，任何權級（privilege level）的軟體都可以使用 SGDT 指令來取得 GDT 暫存器的內容，像這樣：

```
gdtlimit  dw  ?                   ; GDT's length - 1
gdtaddr   dd  ?                   ; linear base addr of GDT
...
sgdt      fword ptr gdtlimit      ; any ring
lgdt      fword ptr gdtlimit      ; ring0 only
```

請注意 SGDT 和 LGDT 的運算元都是 6 bytes。也就是說，上面程式片段中的 SGDT 指令把 GDT 的 limit（譯註：GDT 的大小減 1）存放在 *gdtlimit* 變數中，把 GDT 的虛擬位址存放在鄰近的 *gdtaddr* 變數中。而 LGDT 指令則是從兩個變數中取出 6 bytes 做爲 GDT 暫存器內容。

Interrupt Descriptor Table（IDT）

IDT 暫存器內含 IDT 的長度和虛擬位址。IDT 之中含有 gates，用來指定處理器的中斷處理常式（請看**圖 5-10**）。Windows IDT 只含有 0~5Fh 中斷，至於 60h~FFh 中斷可以因爲軟體執行 INT 指令而產生出來，那會引發一個 general protection（GP）fault -- 因爲 IDT segment 的長度之故。IDT 暫存器的存取動作很類似 GDT 暫存器：ring0 程式和真實模式程式可以利用 LIDT 指令來改變 IDT 暫存器內容，而任何 ring 碼都可以使用 SIDT 指令來獲得 IDT 暫存器內容，像這樣：

```
idtlimit  dw  ?                    ; length - 1 of IDT
idtaddr   dd  ?                    ; linear base addr of IDT
...
sidt      fword ptr idtlimit       ; any ring
lidt      fword ptr idtlimit       ; ring0 only
```

和 SGDT 和 LGDT 一樣，SIDT 和 LIDT 的運算元也都是 6 bytes。

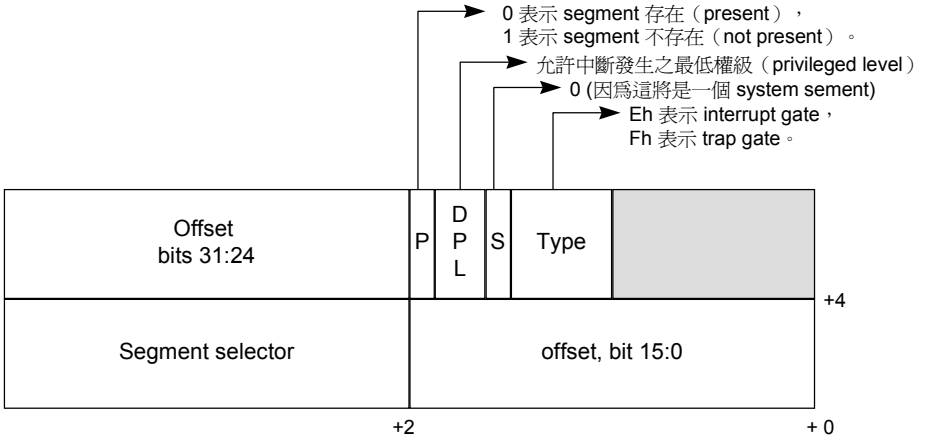


圖 5-10 IDT 中的 Gate 格式

扯翻保護

一知半解可能會帶來危險。某些讀者現在可能已經摩拳擦掌地計劃「如何利用無權級限制的 SGDT 和 SIDT 指令來接管機器」了。我要以經驗來反對這種企圖心，原因至爲明顯：SSDT 和 SIDT 的確會傳回 GDT 和 IDT 的長度和線性位址給你，你可以輕易修改那些表格的內容。雖然那很容易，但那不是可以拿來玩玩的。請注意你以 SIDT 獲得的 IDT 位址可能屬於除錯器而非屬於 Windows。如果你要，你是可以建立一個 gate descriptor，它可以把你神奇地轉移到 ring0 去 -- 儘管你可能會因爲沒有正確處理中斷而當掉，唔，反正電腦是你的！

Windows 已經將內含如此重要表格的 pages 保護起來了，因爲 CPU 的保護機制禁止 ring3 程式處理「被標示爲屬於 supervisor 所有」的 pages。Windows 並不企圖保護它自己的 pages，畢竟它面對的只是一台個人電腦。如果你買一輛車，你有充份的自由將機油流光，看看引擎會怎麼樣，只不過稍後你得爲你的好奇心付出代價就是了。在 Windows 中道理相同。我將繼續扮演逆向工程師的角色，以求知道程式是怎麼跑起來的，但是我會將管理機器的殘酷細節留給知道要怎麼做的系統軟體工程師。

Local Descriptor Table（LDT）

LDT 暫存器內含 LDT 的 selector。Windows 95 爲每一個虛擬機器產生一個 LDT。用以設定 LDT 暫存器內容的 LLDT 指令，只有 ring0 程式才可呼叫；但用以取得 LDT 暫存器內容的 SLDT 指令則否。代表 LDT segment 的那個 descriptor（譯註：位在 GDT 中）的 Type 欄位顯示，LDT 是一個 **system segment**，換句話說你只能夠把 LDT "load" 到 LDT 暫存器中（如果你的權級夠的話）。這個 descriptor table 在記憶體中的長度和位址，被記錄於 LDT 的 segment descriptor 之中。你需要另一個 selector（標示爲一般的 **user data segment**）來存取表格本身。LDT 內的 descriptors 看起來和 GDT 的類似。一如稍早所述，當 selector 的 bit2 設立，它所選擇的是 LDT descriptor；當 selector 的 bit2 清除，它所選擇的是 GDT descriptor。

中斷（Interrupts）

有三種 events 會中斷一個應用程式的正常流程，使控制權交到作業系統手上：硬體中斷、軟體中斷、以及處理器異常（processor exception）。

- 硬體中斷源自 I/O device，表示對於可程式中斷控制器（PIC）的一個中斷請求（一個 IRQ）。
- 軟體中斷發生於程式執行 INT *n* 指令時；目的是要和作業系統溝通。
- 當 CPU 辨識出某個問題或異常情況（例如一個 page fault 或一個 general protection fault），而它們需要作業系統的介入，這時便會發生處理器異常（processor exception）。

這三種中斷隨時都可能發生。我們還可以在上述名單中增加 non-maskable 中斷（NMI），但是一般而言它並不發生於一個運作中的系統。

不論起源爲何，所有中斷都會把當時正執行的程式環境（context）記錄於堆疊之中，然後切換到 IDT 中某個 gate 所描述的處理常式去。CPU 如何正確儲存當時的執行環境（context）？這必須視被中斷程式和處理常式之間的相對權級（privilege level）而定：

中斷來自同一個 Privilege Level 許多情況下，中斷服務常式和被中斷程式處於同一權級（privilege level），這時候 CPU 會把 flags 和目前的指令計數器（instruction counter）壓到目前的堆疊之中，並把控制權交給處理常式。處理常式最後會執行一個 IRET 指令（16 位元碼）或 IRETD 指令（32 位元碼）以便回到被中斷地點。這個程序和真實模式非常類似，只不過真實模式的中斷是透過中斷向量（位於實際記憶體的 0:0 位址）而不是透過 IDT 中的 gate 來引導。

中斷來自較外部（較低）的 Privilege Level 當中斷來自 ring3 而中斷處理常式處於 ring0，事情比較有趣一些。CPU 會先檢查一個名爲 task state segment（TSS）的特殊 system segment（Intel 企圖讓每一個不同的 thread 有自己的一個 TSS，成爲多工的基礎架構。然而透過不同的 TSSs 而完成 task switch，相當耗成本，所以 Windows 並不使用這種方法。取而代之的是 Windows 維護單獨一個 TSS 並使用比較暴力的指令來完成 context switches）。CPU 在處理一個跨越 ring 的中斷時，必須觀察 TSS，因爲 TSS 內含一個指標指向 ring0 堆疊頂端，而該堆疊儲存了舊有的環境（context）。這是一個 static 指標，每次有「需切換至 ring0」的中斷發生，指標內容都相同。

切換到 ring0 堆疊之後，CPU 把舊的（ring3）堆疊指位器（SS:ESP）和一個標準的 interrupt frame（內含 EFLAGS 暫存器和 CS:EIP 指令指位器）壓進堆疊之中（圖 **5-11**）。當 supervisor 稍後執行了一個 IRETD 指令，CPU 便把標準的 interrupt frame 從堆疊中吐出來，然後通知說新的 code selector 的 RPL 需要一個切換動作，回到較外部（較低）的權級（privilege level），然後再吐出堆疊指位器。

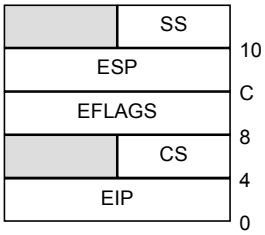


圖 5-11 由一個「向內部（較高）權級發出」的中斷所產生的 stack frame

中斷來自 V86 模式　當中斷發生於 V86 模式，情形比較複雜。在中斷發生時刻，segment 暫存器內含真實模式的 paragraph 號碼，它對於保護模式是沒有意義的。切換到 ring0 堆疊之後，CPU 保存 GS, FS, DS, ES 的值，然後把它們統統設爲 0，這麼一來就可以阻止一個不可重複的 GP fault，因爲它會「在嘗試存取任何資料之前，強迫中斷處理常式載入合法的 selector」。然後 CPU 把舊的 SS:ESP, EFLAGS, CS:EIP 壓進堆疊之中。圖 **5-12** 顯示這樣的 stack frame。中斷處理常式最終呼叫的 IRETD 會把每一樣東西歸定位。當我們後續討論 VxD 程式設計所需要的 **client register structure**（CRS） 時，我會提醒你回頭看看圖 **5-12**，因爲兩者長得很像。

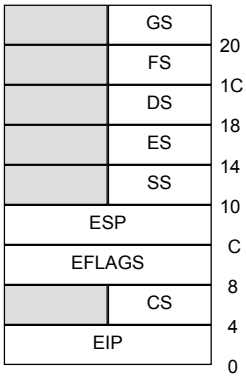


圖 5-12　由一個來自 V86 模式的中斷所產生的 stack frame

中斷的處理

關於 Windows 95 的中斷處理，另有四件事情是你應該知道的。當我描述這些瑣碎細節，我其實是在討論大部份 Windows 95 系統程式員需要知道的 assembly 語言本質。

第一件事情是「釐清所謂的硬體中斷」。真實模式 BIOS 對 PIC 所做的規劃，使得 IRQs 0h~7h 的中斷向量是 8h~Fh，IRQs 8h~Fh 的中斷向量是 70h~77h。一個中斷處理常式，例如 IRQ 5，必須分辨自己是因硬體產生的 INT 0Dh 而觸發，或是由 CPU 產生的

general protected fault 觸發，抑或是因爲某個不受約束的程式所發出的 INT 0Dh 指令而觸發。上述最後一種情況（軟體的 INT 0Dh）是一種脫離正軌的行爲，發出者可能會因它所產生的混亂而受到報應。至於前兩種情況，發生於正常系統之中。中斷處理常式通常必須檢查 PIC 的 in-service 暫存器，以便決定中斷發生的原因。

Windows 藉由「把硬體中斷移除到正常範圍之外」，來避免「釐清硬體中斷」的必要性。在 Windows 中，IRQs 0~Fh 的中斷向量是 50h~5Fh，因而不可能和 processor exceptions 所生的硬體中斷混淆。你的真實模式驅動程式和常駐程式突然需要 "hook" 新的中斷嗎？別擔心，VPICD 會把硬體中斷表現到虛擬機器的慣常地方。

即使對 PIC 重新規劃，在 IRQ 0Ch（通常是一個 bus 滑鼠中斷）和 NetBIOS 中斷之間還是有一個混亂情況，因爲兩者都以 INT 5Ch 出現。關於中斷處理的第二件有趣事情就是要解釋 Windows 如何處理這個混亂情況。事實上一個 IDT gate 關聯兩個不同的 權級（privilege levels），目前我們只談到中斷處理常式本身的權級，它可以決定當 CPU 還擲一個中斷時，是否需要做 ring 轉換。在 gate descriptor 中有一個 DPL 欄位，可以決定 CPU 是否允許發生一個 INT n 指令。中斷 5Ch 的 gate 之中的 DPL 爲 0。如果 ring3 程式嘗試發出 INT 5Ch，CPU 會產生一個 GP fault，因爲權級不吻合。GP fault 處理常式於是會把控制權交到 NetBIOS 處理常式手上。

第三件你應該知道的事情是，某些（不是全部）processor exceptions 還會把一個錯誤代碼推到堆疊裡頭。這個代碼對於 Windows 沒有什麼用處，因爲它所傳輸的資訊非常有限。「Exceptions 有時候會把錯誤代碼推到堆疊去」這一事實正是爲什麼一個人必須精確撰寫第一層（first level）中斷處理常式（譯註：請參考第 6 章）的原因之一。第一層中斷（或 exception）處理常式如果不將一個錯誤代碼推入堆疊，就必須推入一個傀儡值（dummy value），如此一來所有的處理常式才能夠面對統一的堆疊格式。系統必須在準備回返至被中斷點時，在恢復暫存器時跳過那些錯誤代碼。

最後一件應該注意的事情是，IDT gates 有數種不同風貌。**Interrupt gates** 是常見的一種，它們會引發中斷處理常式，獲得控制權，並將中斷 "disable" 掉。另有一種 **trap**

gates， 它和 interrupt gates 唯一的不同是，它會讓中斷繼續 "enable"。對於軟體中斷如 INT 21h 者，Windows 會使用 trap gates 直接誘導到 user-privilege（譯註：ring3）碼去。這麼做可以讓處理常式省掉「經由 STI 指令（那會產生一個成本高昂的 GP fault）將中斷重新 "enable"」的義務動作。Gates 甚至還被區分爲 16 位元和 32 位元兩種，這樣的劃分並不用來代表處理常式於 USE16 或 USE32 segment 執行，而是用來控制「被儲存在堆疊中的環境（context）」的大小。Windows 硬體中斷總是能夠經由 32 位元 gates 來導引，因爲一個人沒辦法事先知道執行的是 16 位元碼還是 32 位元碼。然而 System VM 中的許多軟體中斷是由 16 位元 gates 導引，這個事實也使得 16 位元 gates 被 Win32 程式使用的機會大大地降低了。

第二篇

虛擬裝置驅動程式基礎技術

Virtual Device Driver Basics

第 6 章

虛擬機器管理器
The Virtual Machine Manager

Virtual Machine Manager（VMM）是 Windows95 作業系統的核心，建立並維護一個用以管理 virtual machine（VM）的架構。Virtual device drivers（VxD）配合 VMM 將硬體裝置虛擬化，並提供系統服務給應用程式和其它的 VxDs 使用。如果你喜歡，你可以想像 VMM 是整台電腦的 virtual driver，因爲它允許多個程式同時運作，就好像每個程式單獨在一台真實機器上運作一樣。VMM 需要比典型的 VxD 負擔更多的責任，此外，它也提供幾乎所有 VxDs 都會用到的公用函式，並管理其它所有 VxDs。

這一章將從三方面來描述 VMM 的運作（這些運作事實上是 Windows 95 中發生的每一件事情的基礎）：**1**. memory management，**2**. interrupt handling，**3**. thread scheduling。關於 memory management，你需要知道的基本知識是「VMM 如何分割位址空間」；稍後章節會討論製作 VxD 時用以配置或控制記憶體的各種 service calls。你還需要明瞭 VMM 如何處理中斷，如果沒有這項知識基礎，你就很難知道如何攔截一些可能發生的中斷。最後，你必須明瞭 VMM 如何選擇它所要執行的 threads，因爲應用程式就是經由這種基本機構，才得以執行。

記憶體管理

不知道爲什麼，一台電腦上的資源永遠不夠。是的，CPU 時間（processor cycles）總是不夠分配；螢幕總是不夠大到足以容納我們希望開啓的視窗；視訊卡總是不夠快；總是沒有足夠的解析度來技援現在流行的影像；數據機總是不能滿足資訊高速公路不斷暴增的頻寬要求。然而，每一台電腦上最缺乏的資源似乎總是記憶體，因此，Windows95 繼續朝向更複雜的記憶體管理技術邁進，也就不足爲奇了。

系統記憶體映射狀態（The System Memory Map）

VMM 使用 Intel 80386+ 處理器的 paging（分頁）功能來完成 32 位元虛擬位址空間。爲了簡化記憶體管理，VMM 將所獲得的位址空間細分成四個區域，如圖 **6-1** 所示。這四個重要區域是：

- **V86 region**，從線性位址 0h 延伸至 10FFEFh，屬於目前執行的 VM 所有（然而就像稍後即將出現的方塊文字所說的一樣，10FFF0h ~ 003FFFFFh 位址沒有被使用）。這個區域所含括的記憶體可以用一個範圍從 0000h:0000h 至 FFFFh:FFFFh 的 16:16 指標來定址。
- **private application region**，從 00400000h 開始延伸至 7FFFFFFFh。這部份虛擬記憶體對一個特定的 **memory context** 而言是私有的，基本上對應於一個 Win32 process。每一個 VM 以及每一個 Win32 應用程式均隸屬不同的 process，因此各自擁有自己的一塊 private application region。
- **shared application region**，從 80000000h 至 BFFFFFFFh。Windows 將系統本身的一些 ring3 DLLs（如 KERNEL32.DLL 和 USER32.DLL）以及所有 Win16 應用程式載入這一區域。Windows 也使用這個範圍的線性位址來滿足 System VM 中的 Win32 程式對於 file-mapping（以 *CreateFileMapping* 建立之）的要求，以及應付 VM 之中來自 DPMI client 的記憶體配置需求。
- **shared system region**，從 C0000000h 至虛擬記憶體的頂端。此區域擁有系統資料以及「被所有 processes 和 VMs 共享」的程式。這也就是 VMM 和

其他所有 VxDs 生存的地方。每一個 VM 的 V86 region，其 pages 也是被映射到位址空間的這個範圍內。這種所謂的 "high-linear mapping" 允許 VxD 存取特定 VM 的 V86 記憶體 -- 甚至即使這個 VM 並不是當班的那個（這種情形下，隸屬其它 VM 的一些記憶體將會佔據位址空間的第一個 MB）。

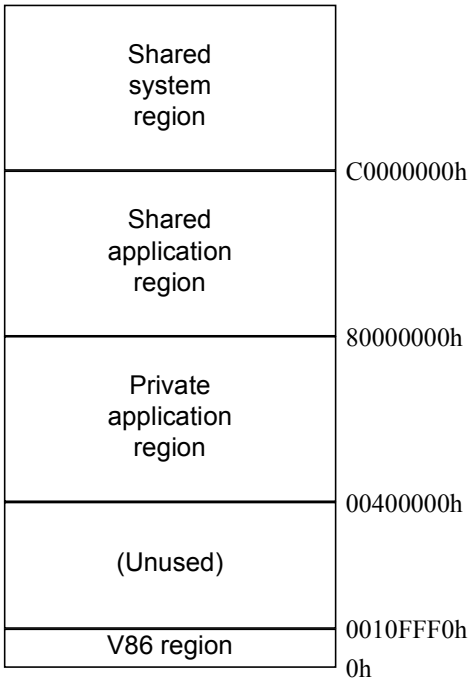


圖 6-1　Windows95 虛擬位址空間映射圖

記憶體共享（Memory Sharing）

Context-specific memory（譯註：因 processes 不同而不同的記憶體）和 shared memory（共享記憶體）之間的差異，對系統程式員而言相當重要。將所有 Win16 程式放在 shared application region，有其危險性，但是爲了讓 Windows 95 保持與早期 Windows 版本相容，這又是必須的選擇。這種情形下應用程式可以自由地彼此存取對方的記憶體。爲了

滿足 DPMI 在這個區域中對記憶體的配置需求，Windows 95 允許 Win16 程式和 extended DOS 程式繼續分享這一塊（譯註：shared application region）線性位址。

然而 Win32 程式會遇到另外一些相容性限制。兩個 Win32 程式分享記憶體的唯一合法方式就是建立 named file mapping，因爲即使你使用 *GMEM_SHARE* 這個動人的屬性，*GlobalAlloc* 還是不能夠建立 shared memory，它只會在每一個 process 的 private application region 裡配置記憶體。有經驗的 Windows NT 程式員應該對此事實不感意外才是！因此，如果記憶體對 Win32 程式是可共享的，自然而然地 Win16 程式和 extended DOS 程式也就可以取用它。

相容性注意事項 你不應該在沒有檢查 Windows 版本號碼之前，就全然相信 Win32 程式和 Win16（或 DOS）程式之間的記憶體共享關係。Microsoft 可能會在未來版本中改變位址分佈。此外，Windows 95 目前也從 shared application region 中配置可共享的（具名的）和不可共享的（未具名的）memory-mapped file，但 Microsoft 卻視此爲一個病癥。最終，不可共享的 mappings 將移至位址空間中的 private application region，以避免被共用。

最前面的 4MB

你可能很好奇到底 Windows 95 在 V86 region 尾端至 private application region 起點（0010FFF0h~003FFFFFh）大約 3MB 之中放了些什麼？答案是，什麼也沒有！Windows 95 放棄這塊位址空間的原因是爲了配合 Intel 晶片對於 page tables 的編組。

我們知道，system control register CR3 指向一個 page table directory（4096 bytes），其中每一筆記錄（4 bytes）代表一個 page table（也是 4096 bytes）的實際位址。Page table 中的每一筆記錄（32 位元）如果不是代表實際記憶體中的一塊 page（4096 bytes），就是用來讓 VMM「將 page 初始化，或是從 swap file（置換檔）中把內容叫出來」。

請注意單單 page table directory 中的一筆記錄（entry）便可以控制多少記憶體：「1024 筆 page table entries」乘以「每個 page 有 4096 bytes」，得到 4MB。換句話說 page table directory 中的一個 DWORD 可以描述虛擬記憶體中的 00400000h bytes，那正是 VMM 爲一個 VM 的 V86 region 所保留的空間。在 Windows 3.0 和 Windows 3.1 中，並沒有所謂的 private application region，VMM 從一個 VM memory context 切換到另一個 VM memory context 的方式，就是簡單地把 page directory 中的第一個「page table 指標」換掉。由於 V86 記憶體只佔用 1MB，所以剩下的 3MB 就失去作用了。Windows 95 還是使用 page table directory 的一筆記錄來指向 V86 記憶體，但 Windows 95 需要切換 page directory 中的許多指標才能完成 context switch。

VxDs 必須注意，不要在特別的時段對 context 以外的記憶體做出參考（取值）動作。例如在硬體中斷的過程中，VxD 不應該參考「被中斷的這個 process」之外的 processes 的記憶體。如果不遵循這個規則，當 VMM 不能容忍 page fault 時就可能導至 page fault 的發生，並因而使系統當掉。*_LinPageLock* service 允許 VxD 開發人員將一塊 private page 鎖定在記憶體中，並得到一個 shared region alias，此 alias 不管在哪一個 context 中都可用。

記憶體管理之 API 函式

VxDs 可使用三大類 VMM services 來配置和釋放虛擬記憶體。**Paging subsystem** 提供 *_PageAllocate* 和 *_PageFree* services，用來配置虛擬記憶體中以 4096 bytes 爲單位的大塊區域。**Heap manager** 提供 *_HeapAllocate* 和 *_HeapFree* services，用來從 process heaps 中配置小塊記憶體；一共有三種不同的 heap 記憶體：一種是 pageable 記憶體，一種是 page-locked 記憶體，另一種是當 device 初始化之後（也就是在 *Init_Complete* system control message 被處理完畢之後）會被拋棄的記憶體。**Linked-list manager** 允許 VxD 產生一堆固定大小的記憶體區塊，每一塊都因 *List_Allocate* 而獲得。

對於那些將在 ring0 碼使用的記憶體，VxDs 是以 heap 和 linked-list services 配置之。至於 *_PageAllocate* 主要用來配置那些給一般應用程式使用的記憶體。「對於單一 VM

而言是區域性的」並且「通常在 MS-DOS 記憶體和 A0000h 之間以自由空間呈現」的 V86 記憶體，都是使用 *_PageAllocate* 配置而得，例如每一個 VM 中的那些「video RAM 位址之後」的 pages。應用程式如果呼叫 *VirtualAlloc* 或是 DPMI 記憶體配置函式，也會導至 *_PageAllocate* calls。Heap manager 和 linked-list manager 所做的次階配置（suballocate）動作是從 *_PageAllocate* 所獲得的 pages 邊界開始。

中斷（interrupts）的處理

一部電腦如果正在執行一個運算，而其結果是對使用者有益的，我們說它處在正常狀態。所謂中斷（interrupts），是一種用來凍結正常運算的一種基礎機制，使系統能夠處理異常情況（exception），或對應用程式提供一個作業系統的 service。面對中斷，VMM 的行爲大致可以描述如下：當中斷發生，VMM 被喚醒以處理中斷，然後執行一個 IRETD 指令，回到中斷發生之前的執行點。請看**圖 6-2**。

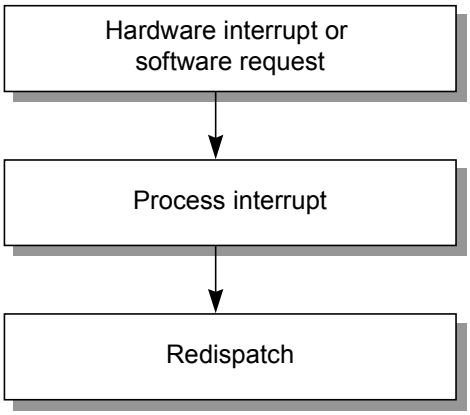


圖 6-2 VMM（Virtual Machine Manager）的主迴路。

第一層和第二層（First-Level and Second-Level）中斷處理常式

VMM 內含大量的第一層中斷處理常式，每一個常式服務一個可能的中斷。第一層中斷處理常式的工作是把被中斷程式的狀態儲存起來，然後把控制權交到第二層中斷處理常式手中 – 這才是真正提供中斷服務的人。「狀態儲存」相當於利用 PUSHAD 和 MOV 指令將所有 general 暫存器和 segment 暫存器儲存於堆疊之中。第二層中斷處理常式做它應該做的事情，然後把控制權交到一個 redispatch 常式手中。**Redispatcher** 把原先儲存的狀態恢復回來，然後執行一個 IRETD 指令，重新回到被中斷處繼續執行。

儲存 Coprocessor Context　當你讀到「VMM 使用 PUSHAD 和 MOV 指令將目前的程式狀態儲存起來」，你是否想到 math coprocessor？FSAVE 和 FRESTORE 指令可以儲存和恢復 coprocessor 的狀態，不過它們非常耗時。因此 Windows 並不在中斷發生的時候自動儲存 coprocessor 狀態。事實上它甚至不在一個 thread 切換至另一個 thread 時自動儲存 coprocessor 狀態。取而代之的是，它依賴 CR0 暫存器中朦朧難懂的 Task Switched flag。在一個 context switch 過程期間設立這個位元，會使得下一個 coprocessor 指令失敗，於是 Windows 才切換 coprocessor contexts。這是一種 "just in time"（及時）程式設計！

在中斷發生之後做 redispatching 動作，事實上十分複雜。第二層中斷處理常式可能會排班等候一個 event，以執行它目前沒有辦法執行的工作。這麼一來，redispatcher 就必須在完成目前的中斷之前，呼叫相關的 event callback 函式。此外，redispatch 可能有一些好理由去執行非被中斷的 threads，原因可能是：被中斷之 thread 的 CPU 時間配額（time slice）終了，或是因爲第二層中斷處理常式凍結了該 thread。因此，排程器就有了一個機會，決定誰應該接下去執行。

Interrupt Descriptor Tables（IDT）

每一個 VM 有自己的一個 Interrupt Descriptor Table（IDT）。由於 Windows 95 總是在保護模式中執行（如果不考慮特殊的 Single MS-DOS Application Mode），VM 的所有中斷係從它自己的 IDT 被導引到 VMM 的第一層處理常式去。

VMM 初次產生一個 VM 時，同時也將其 IDT 初始化。初始值來自一個預設表格，那是 VMM 和其他 VxDs 在系統初始化時產生出來的。VxDs 可以使用 *Set_PM_Int_Vector* 和 *Set_PM_Int_Type* 兩個 services，指定 IDT 各筆記錄中的任何初值，使它不同於 VMM 的預設值。

有多少 IDTs？ *Set_PM_Int_Vector* service 爲「來自保護模式應用程式之中斷」指定第一層中斷處理常式。這個處理常式並不能被 V86 模式的應用程式使用。每一個 VM 因而事實上擁有兩個 IDTs：一個給 V86 程式使用，另一個給保護模式程式使用。V86 中斷向量（interrupts vector）指向第一層中斷處理常式，其預設行爲就是映射 0000h:0000h 開始的中斷向量表格（**Interrupt Vector Table，IVT**）。

中斷（Interrupts）的種類

Windows 95 將硬體中斷、軟體中斷、處理器異常（processor exceptions）分門別類。硬體中斷表示有一個 I/O device 需要服務；軟體中斷是由程式以 INT *n* 指令產生；異常（exception）中斷則是當處理器決定作業系統必須處理某些異常情況時發出。

中斷的分類左右了「VxDs 和保護模式應用程式如何 hook 中斷」以及「VMM 如何爲中斷服務」的行爲。應用程式使用 DPMI services 來 hook 中斷和異常情況。VxDs，包括爲 DPMI calls 服務的那一個，使用數個不同的 VMM services 來 hook 硬體中斷、軟體中斷、處理器異常：

- *Set_PM_Int_Vector* 和 *Set_PM_Int_Type* 會改變 IDT 中的 gate entry，進而改變來自保護模式應用程式的第一層中斷處理常式。一般而言 VxD 只有在安裝一個 ring3 處理常式時才會使用這兩個函式（因爲一個 ring0 第一層中斷處理常式需要做許多狀態儲存和堆疊置換工作）。然而 ring3 處理常式常常只是一個因呼叫 *Allocate_PM_Call_Back* 而產生的 INT 30h 動作。當中斷發生，對應的 ring0 callback 常式會在 VMM 設定正常的 VxD 執行環境後取得控制權。

- *Set_V86_Int_Vector* 會改變 0000h:0000h 中斷向量表格（IVT）中的一個遠程函式位址。這會改變一個已知的虛擬中斷的 V86 模式服務常式。

- *Hook_PM_Fault* 會對發生於保護模式應用程式中的處理器異常（processor exception）或軟體中斷，建立起一個第二層處理常式。預設的第一層處理常式（可由 IDT 觸及）在儲存了 context 之後便呼叫這個處理常式。任何以 *Set_PM_Int_Vector* 安裝的非標準第一層處理常式會先取得控制權。如果它和預設處理常式串鏈（chains）在一起，使用 *Hook_PM_Fault* 的 VxDs 就可以在這個中斷之中獲得一些機會。

- *Hook_V86_Fault* 會對發生於 V86 模式應用程式中的處理器異常（processor exception）建立起第二層處理常式。預設的處理常式會把中斷反映到 0000h:0000h 中斷向量表（IVT）中。

- *Hook_V86_Int_Chain* 把 VxD 加到第二層處理常式所形成的串列中，此 VxD 將在 V86 模式發出軟體中斷時首先當掉（crash）！

- *Hook_VMM_Fault* 爲「發生於 ring0 碼正在執行時」的處理器異常建立起第二層處理常式。VxD 如果需要此一 service，其實是很不尋常的，因爲 VMM 之中已經內含了 ring0 fault 處理常式。你可以使用 *Install_Exception_Handler* 來誘捕（trap）VxD 之中誤入歧途的指標（也可使用 VMM service *_Assert_Range* 在第一時間就阻止 VxD 產生不良的指標）。

預設處理方式（Default Handling）

VMM 的原先目標是將電腦虛擬化，使 Windows 和多個 MS-DOS sessions 可以同時運作。你已經知道，MS-DOS 內含一大組中斷處理常式；Windows 的 KERNEL, USER, GDI 三大模組以及這些模組所用到的其他 ring3 驅動程式，也都能夠處理中斷。爲了讓虛擬化的工作正確發展，VMM 對幾乎每一個中斷的預設處理方式是，把它映射（reflect）到一個 VM，使某個 ring3 處理常式能夠處理它。映射（reflecting）一個中斷，相當於改變該 VM 的「被儲存之暫存器內容（saved register images）」，於是當 VMM 下次 dispatched 該 VM 時，適當的中斷服務常式能夠獲得控制權。

然而，立刻將中斷映射（reflect）出去，並不永遠是正確的作法。當硬體中斷發生，正確的硬體中斷處理常式可能位於另一個 VM，而不是在中斷發生時處於 active 狀態的那個 VM。許多 processor fault，特別是 page fault，必須由 VMM（而不是存在於任何 VM 中的任何軟體）處理。VMM 和其他 VxDs 可能至少得審查一下軟體中斷，爲的是將該中斷所關聯到的電腦資源加以虛擬化。

硬體中斷（Hardware Interrupts）

有一個系統元件名爲 VPICD（Virtual Programmable Interrupt Controller Device），內含 50h~5Fh 中斷的第一層處理常式。VPICD 將真正的中斷控制器重新規劃，使得上述編碼（50h~5Fh）的中斷請求 0h~Fh，能夠取代真實模式 BIOS 的預設值（08h~0Fh 以及 70h~77h）。那些中斷所對應的 IDT gates 的 DPL（Descriptor Privilege Level）爲 0，所以 ring3 程式如果企圖對它們發出軟體中斷，會引發 GP fault。至於 Ring0 碼，當然不會嘗試對它們發出軟體中斷。

預設情況下，VPICD 會將一個硬體中斷映射到一個最合適的 VM 的 ring3 處理常式去。究竟哪一個 VM 獲得中斷，則有賴數個因素決定，包括當 Windows 95 啓動時 IRQ 被 masked 或否！第 13 章將詳細討論虛擬中斷的繞行（routing）。

VxDs 可以呼叫 *VPCID_Virtualize_IRQ* 以改寫一個 IRQ 的預設處理常式（對某個硬體

中斷直接改變其 IDT entry 是非常不好的作法）。有時候 VxD 只不過是想選擇一個「不同於 VPICD 所選擇的」VM 來服務此一中斷。將 IRQ 虛擬化，有一個特別好的理由，那就是爲一個硬體裝置提供完整的 ring0 服務 -- 這種情況下 VxD 不該把中斷映射（reflect）到任何 VM 去。

處理器異常（processor exceptions）

Intel 企圖保留 0h~1Fh 中斷給它自己使用，用以描述處理器異常。有趣的故事發生了，IBM 讀了 Intel 技術文件中的「保留」一詞，以爲是爲 IBM 保留，於是指定了一些不使用於 8088 的硬體中斷和 BIOS 軟體中斷給它。因此，一個真實模式作業系統或是 DOS extender，面對 INT 0Eh，無法立刻判別究竟是因 page fault 而引起，或是因 floppy controller 在 IRQ 6 而引起。爲避免模稜兩可，Windows 95 於是把硬體中斷搬移到 50h~5Fh，並明智地使用 DPL=0 的 gate 來阻止 INT *n* 指令。

只要呼叫 *Hook_PM_Fault*，VxDs 便可以改寫保護模式異常情況的預設處理。Windows 95 kernel 使用 DPMI 子功能 0203h 來 "hook" 一個異常情況，就是這麼回事。你不應該藉由 *Set_PM_Int_Vector* 來改變一個異常（exception）中斷的 IDT gate，因爲這樣會將 fault handler 的正常串鏈打斷掉。VxD 可以呼叫 *Hook_V86_Fault* 來改寫 V86 模式異常情況的預設處理行爲。由於標準的 VxDs 會 "hook" 某些異常情況，所以在你加入自己的 VxD 之前，知道它們大概會如何被處理，頗有助益：

Exception 00h（Divide Error）　預設行爲是將中斷反射（reflect）回 VM，因此一個已由應用程式之 runtime 環境所安裝妥當的處理常式，可能會使該程式當掉。

Exception 01h（Debugger Call）　預設行爲是將中斷反射（reflect）回 VM，該 VM 往往會忽略此中斷。Application-level debuggers 會 hook 此一中斷，以便單步執行一個應用程式，並處理 debugging 暫存器中的 breakpoint 位址。

Exception 02h（NonMaskable Interrupt） 預設行爲是忽略此一中斷。

Exception 03h（BreakPoint） 預設行爲是將中斷反射（reflect）回 VM，該 VM 往往會忽略此中斷。Application-level debuggers 會 hook 此一中斷以設定 single-byte breakpoints。

Exception 04h（INTO-Detected Overflow） 預設行爲是將中斷反射（reflect）回 VM。

Exception 05h（BOUND Range Exceeded） 預設行爲是將中斷反射（reflect）回 VM。

Exception 06h（Invalid Opcode） 一般而言這發生在 V86 碼執行 ARPL 指令時（Windows 95 以它做爲一個 breakpoint）。這種情況下，預設行爲是呼叫 breakpoint callback 函式（在某些 VxD 中），它們會以適當方式重新安排 VM 暫存器。至於其他情況，預設行爲是把這個 VM 結束掉！然而 Windows 95 kernel 總是爲 System VM "hook" 這個 fault，以避免 VM 當掉。

Exception 07h（Coprocessor Not Present） 預設行爲是把這個異常情況反映（reflect）到 VM；不過 Virtual Coprocessor Device（VCPD）已經 hooks 這個異常情況。VCPD 使用 CR0 暫存器中的 Task Switched flag 觸發這個異常，爲的是將真實的 coprocessor（如果有的話）虛擬化，並且安排 VWin32（一個爲 KERNEL 執行低階功能的 VxD）爲 Win32 應用程式模擬一個其實並不存在的 coprocessor。

Exception 08h（Double Fault） 這個異常表示 system control tables 有一個嚴重的問題。它是獨一無二的中斷，中斷向量會經過一個 task gate，保證一組安全而調和的暫存器內容。這個中斷的處理常式並不做任何事情，意思是不管發生什麼事，此一異常註定要重複，直到使用者累了，將機器關閉爲止。

Exception 0Ah（Invalid Task State Segment）　這個錯誤不應該發生。如果它發生了，Windows 應該會當掉 -- 不過事實上它被反射（reflect）回 VM。

Exception 0Bh（Segment Not Present）　這個異常不會在 V86 模式中發生。預設行爲會毀滅一個 VM。Windows 95 kernel 總是爲 System VM "hook" 此一異常，因爲它依賴 "**not-present**" exception，以載入 16 位元程式的 LOADONCALL segments。

Exception 0Ch（Stack Exception）　預設行爲是毀滅這個 VM，不過 Windows 95 kernel 總是爲 System VM "hook" 此一異常，以避免悲劇發生。你必須對此異常付出努力，否則你將獲得一個無窮迴路的 faults，因爲預設處理常式只是簡單地重新執行那個引起錯誤的指令。

Exception 0Dh（General Protection）　沒有單一的預設行爲，因爲有太多不同的情況；大部份情況會引起 GP faults。如果 VMM 和 VxDs 都沒有處理這個 fault，那麼唯一的可能就是這個 VM 當掉！Windows 95 kernel 總是會爲 System VM "hooks" 這個 faults，以阻止慘劇發生。

Exception 0Eh（Page Fault）　預設行爲是從暫存器 CR2 中讀取 faulting virtual address，並指定一個 physical memory page。

Exception 10h（Floating-Point Error）　這個異常不會發生在 Windows 95 之中，因爲 Windows 95 並不改變暫存器 CR0 的 enabling 位元。

Exception 11h（Alignment Check）　這個 exception 不會發生於 Windows 95 之中，因爲 Windows 95 並不改變暫存器 CR0 的 enabling 位元。

Exception 12h（Machine Checks）　VMM 會將此 Pentium exception 反射（reflects）給 current VM，而不是當掉。

V86 模式軟體中斷（V86-Mode Software Interrupts）

V86 軟體中斷的預設處理方式是把此中斷經由 0000h:0000h 的 Interrupt Vector Table（IVT）反射（reflect）到 VM 之中。VxD 可以經由 *Hook_V86_Int_Chain* 介入之。舉個例子，VMM 和其他標準的 VxDs 通常會 "hook" INT 2Fh 以濾出 16xxh 子功能。SHELL virtual device 也會 "hooks" INT 2Fh 以濾出和 clipboard 存取有關的 17xxh 子功能。任何可以將樁腳安插到（所謂 "pile onto"）INT 2Fh 的 VxD 則可以尋找子功能 168Ah （取得廠商自定之 API Entry Point。譯註）。

譯註：一個頂有名的例子就是 Microsoft 提供的對 DPMI 的擴充服務，用法如下。以下規格可以從 Ralf Brown & Jim Kyle 所著的 ***PC Interrupts***（Addison Wesley 出版，1991）第 11 章獲得：

呼叫時的暫存器內容：

AX=168Ah

DS:(ESI) = selector:offset of ASCII vendor name（本例爲 "MS-DOS"）

呼叫後的傳回值：

AL = 執行結果代碼（8Ah 表示失敗；00 表示成功，於是 ES:(EDI) 指向 extended API 的 entry point）

Microsoft 提供的 DPMI 擴充函式中，第 100h 號服務是：get LDT alias。如果我們獲得一個 alias LDT，我們就可以自由自在地在上面做任何動作，包括取出所有 LDT descriptors、改變它們的內容等等。這些動作都會真實反應到 LDT 上頭。當我們希望配置一個 286 CallGate 時，通常採用此法。

保護模式軟體中斷（Protected-Mode Software Interrupts）

保護模式軟體中斷的預設處理方式是把此中斷反射到 VM（在 V86 模式中）。Windows 3.0 和 3.1 就是如此，在其中，MS-DOS 和 BIOS 處理了應用程式所發出的大部份軟體

中斷。嚴格來說這仍然是 Windows 95 的預設行爲，但是 VxDs "hook" 它們之中的大部份，爲的是在保護模式中儘可能完成最多機能。一個很重要的例子就是 INT 21h，這是 MS-DOS 和 Win16 程式用以要求系統服務的介面。在先前的 Windows 版本中，VMM 送出大部份的 INT 21h（特別是和檔案存取有關者），進入 V86 模式下的 MS-DOS（或許是在適當改變指標參數以描述 V86 記憶體之後）。在 Windows 95 之中，Installable File System Manager 攔截對檔案系統的請求，並完全在 ring0 服務它們。

Events

就像大部份作業系統一樣，VMM 需要一個內部機制，用來產生並服務一系列的 events。Event 代表在需求發生時刻，VMM 沒辦法滿足（或不方便執行）的「一個單位的工作」。舉個例子，某些 VM 正等待一個表示「運算完成」的硬體中斷，而此中斷卻在另一個 VM active 時發生；這種情況下中斷處理常式可能會排班等候一個 event callback，以便在其他 VM 的環境（context）中執行。

爲什麼一個 VxD 需要將一個 event 放進 queue 之中？最常見的原因是爲了和 paging 合作。硬體中斷可以發生在任何時間，只要 CPU 處於可中斷的情況下，而大部份時候的確如此。爲了避免中斷遺失，VMM 在處理與 paging 有關的資料結構時仍然將中斷保持在 enabled 狀態。事實上，VMM 和其他 VxDs 通常都會讓中斷保持在 enabled 狀態（只有面對非常短暫的 critical section code 除外，不過這一節我們只關心 paging）。由於 paging 系統並非可重複進入（reentrant），即使 Windows 95 亦然（其中 ring0 碼已經代替 MS-DOS 來處理 paging I/O 了）；所以「不讓硬體中斷觸發遞迴現象」就成了重要關鍵。於是，硬體中斷處理常式只能使用那些被特別設計爲非同步（asynchronous）的 VxD services。Asynchronous services 保證不會引起任何 paging 動作，意思是它們佔用 locked memory pages 並且只參考（使用）locked memory structures。更進一步說，它們只使用 asynchronous VxD services（如果有用到的話）。

大部份硬體中斷處理常式需要執行的動作並不是非同步的，例如許多處理常式需要通知 ring3 碼說有一個動作已經完成，但是用以執行這些任務的 services 很明顯在一個硬體

中斷處理常式中並不安全。「event queuing 機制」提供了活門閥的功能，允許硬體中斷處理常式在安全時刻內將「VMM 將要執行」的延遲工作項目加以排程。

> 譯註：我恐怕我譯得不好，因此爲這一段附上原文：The event queuing mechanism provide the escape valve that allows hardware interrupt handlers to schedule deferred work items that the VMM will perform when it's safe to do so.

Event queue 背後的基礎觀念是，VxD 可以在任何時刻產生一個 event object，特別是當它服務一個硬體中斷的時候。當 VMM 到達「可以容忍一個 page fault」的穩定度時，便掃描 event queue 並呼叫 events 所關聯的 callback 函式。這時候這個 callback 函式可以被任何 VxD services 呼叫，所以不管什麼事情，只要必須發生，都可以發生！一旦所有 events 都被服務過了，VMM 就執行一個 IRETD 指令，"redispatch" 某些應用程式。重新獲得控制權的，可能是最初被中斷的那個應用程式，也可能是排程器（scheduler）決定應該執行的那個應用程式。

VMM 把 events 分爲兩類（以「必須在什麼 context 之中執行」爲基準）：

- **Global events** - 施行於整個系統，而不是特定某個 VM 或 process 或 thread。VMM 會首先處理 queued global events。
- **Local events** - 施行於特定某個 VM 或 process 或 thread，並且必須在那個 context 之中執行。在處理完 global events 之後，VMM 爲目前的 VM、process 或 thread 處理任何的 queued local events。這些 event callbacks 可能會觸發一個 VM switch 或 thread switch，於是 VMM 將爲新的 local context 處理一個新的 event queue。

巢狀執行（Nested Execution）

要讓 VMM 的主執行路線可以不同於簡單的「服務然後回返（service and IRETD）」，events 是一種方法，Nested Execution 是另一種方法。當 VMM "redispatches" 一段碼而它不同於被中斷的碼時，nested execution 就會發生。"nesting" 之所以發生，是因爲特殊

的 dispatched code 回返（return）到 VMM 中（經由其他中斷的產生），VMM 然後可以重新開始其原來的主線，並且最終回返（return）到最外層中斷。爲了技術上的理由，nested execution blocks 通常發生在一個 event callback 函式中。

如果沒有一個好例子，nested execution 的觀念會使你眼花撩亂。圖 **6-3** 及此處的解釋將有助於你搞清楚這個觀念。假設一個 Windows 程式發出一個 INT 5Ch 以完成一個 NetBIOS request，而唯一的 NetBIOS 供應者是一個雜亂的、陳舊的、真實模式的網路驅動程式。你應該還記得 INT 5Ch 會引起一個 GP fault -- 因爲 DPL 不吻合。最後由 VNETBIOS driver 獲得控制權，因爲它 "hooked" INT 5Ch。它進入一個 nested execution block 以執行真實模式的 INT 5Ch 處理常式。當真實模式處理常式終於執行了一個 IRET 回到它「以爲是一個真實模式 NetBIOS call」的地方，事實上它到達的是一個 ARPL 指令，於是引發 **Invalid Opcode** exception（中斷 06h）。VMM 知道這意味此一 return 來自 VM，於是把控制權還給 VNETBIOS，於是退出 nest execution block 並將控制權送還給 VMM。或快或慢，VMM 會 "redispatches" 原來的「INT 5Ch 保護模式呼叫者」。

對執行緒（Threads）進行排程

VMM 有兩個排程器（scheduler），用以在多個 threads 和 VMs 之間實現插斷式多工（preemptive multitasking）。**Primary scheduler** 的責任是選擇下一個將被執行的 thread。基本上 primary scheduler 只是從一系列符合資格的 threads 中選擇最高優先權的那個。選擇動作發生在 VMM 處於 "in control" 狀態（正在服務一個中斷）時，而其結果決定了「當 VMM 終於要 redispatches "user code"（譯註）時」所必須儲存的暫存器內容。VxDs 使用 primary scheduler 所提供的服務來調整 threads 優先權，以控制選擇結果，並使用各種同步元件（synchronization primitives）讓 threads 具有（或卸除）排班資格。

譯註：這裡所謂 user 是指使用 primary scheduler 的使用者，不特定指誰；不是 USER 模組，當然更不是 end user。

Primary Scheduler 的客戶之一是所謂的 time-slicing（或稱爲 secondary）scheduler。**Time-slicing scheduler** 使用 primary scheduler 提供的服務，在一個複雜的優先權體制之下，實現對於 CPU 的 "round-robin" 配置方式。它也經由 KERNEL32.DLL 和 VWIN32 VxD 之間的內部介面之助，對 Win32 API calls 如 *SetThreadPriority* 和 *SetPriorityClass* 做出回應。

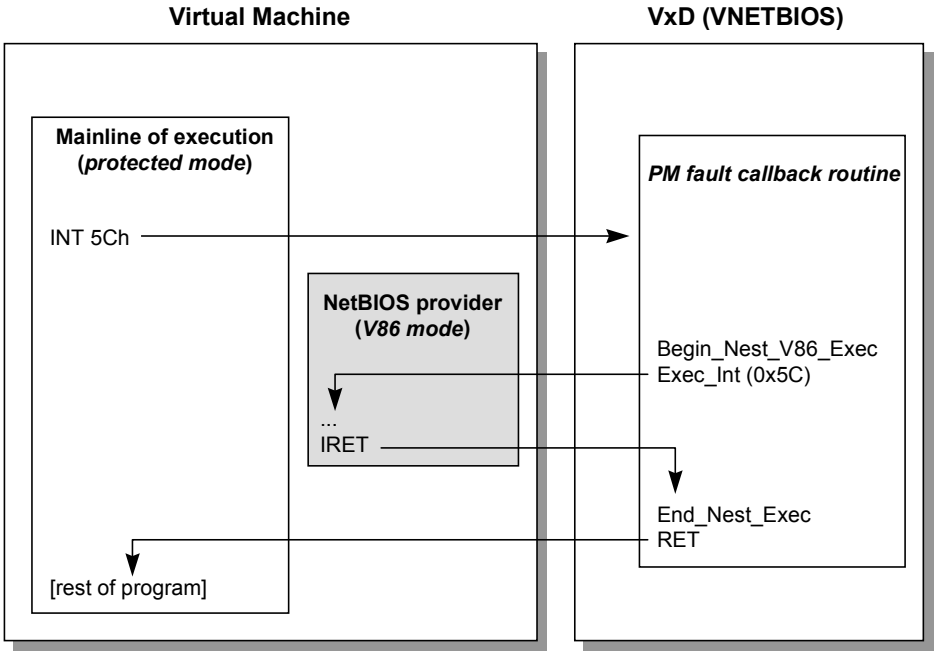


圖 6-3 VNETBIOS 如何使用巢狀執行（Nested Execution）。VMM 藉以獲得控制權的那些內部動作並未顯示出來。

執行緒（Execution）與 Win32 優先權

Primary scheduler 和 time-slicing scheduler 分別使用不同的優先權體制。Primary scheduler 根據 thread 的「執行優先權」決定是否給它機會成爲下一個得到控制權的幸運傢伙。這種決定「要不就有，要不全輸」：最高優先權的 threads 才有可能被執行。由於 Windows 95 是一個單處理器作業系統，沒有任何 threads 可以在某個 threads 正執行時同時被執行。然而，優先權常會改變，因爲各式各樣的 VxDs 會實施（或移除）priority boosts（優先權微調提昇）。舉個例子，基本優先權爲 8h 的 thread 進入了一個 critical section，爲了讓它完成動作並釋放 critical section，VMM 將其優先權調高，增加 *CRITICAL_SECTION_BOOST*（00100000h）。於是，它的優先權變成 00100008h，較先前有更多機會被執行。Time-slicing scheduler 也會爲了挑出下一個（即將被執行之）thread 而實施優先權提昇（priority boost）動作。

表 **6-1** 列出 Windows 95 中的優先權調昇值。你也可以以另外方式來調昇優先權，但很少會有人這麼做。Primary scheduler 使用 *RESERVED_LOW_BOOST* 和 *RESERVED_HIGH_BOOST* 做爲其內部確認值，你我都不能夠以這兩種方式來調昇優先權。Time-slicing scheduler 使用 *CUR_RUN_VM_BOOST* 來暫時調昇那個「根據 round-robin 體制，被選中成爲下一個執行體」的 thread 的優先權。*Begin_Critical_Section* service 會自動以 *CRITICAL_SECTION_BOOST* 調昇 thread 優先權，而 *End_Critical_Section* service 會自動解除此次調昇。類似的道理，當 VPICD 於某一 VM 的硬體中斷處理常式中執行起來，它會暫時實施 *TIME_CRITICAL_BOOST* 以求儘快結束此次中斷。Time critical boost 在數值上大於 critical-section boost，意思是「硬體中斷處理」優先於「完成一個被 critical section 所保護起來的動作」。

符號名稱	常數定義
RESERVED_LOW_BOOST	00000001h
CUR_RUN_VM_BOOST	00000004h
LOW_PRI_DEVICE_BOOST	00000010h
HIGH_PRI_DEVICE_BOOST	00001000h
CRITICAL_SECTION_BOOST	00100000h
TIME_CRITICAL_BOOST	00400001h
RESERVED_HIGH_BOOST	40000001h

表 6-1　優先權的提昇值

譯註：以下將出現 **priority class**、**priority level** 等名詞，雖然翻譯它們毫無問題，但我認爲還是保留原文的好，因爲它們都是專用術語。

欲瞭解另外兩個優先權調昇動作（*LOW_PRI_DEVICE_BOOST* 和 *HIGH_PRI_DEVICE_BOOST*），我們必須先探索 time-slicing scheduler 的優先權模型。Time-slicing scheduler 使用「針對 Windows NT 而發展」的 Win32 優先權模型。在此模型之中，thread 使用五種 priority classes 之一，每一種 class 內含某個範圍的 priority levels（請看表 **6-2**）。優先權範圍可能彼此重疊，所以一個 *IDLE_PRIORITY_CLASS* 的 high-priority thread，其「可能優先權」高於 *NORMAL_PRIORITY_CLASS* 的 low-priority thread。應用程式（包括 ring3 碼如 KERNEL32）可以使用 Win32 API 函式 *SetPriorityClass* 改變 thread 的 priority class，使用 *SetThreadPriority* 函式改變 thread 的 priority level。

	Priority Class（*xxx_PRIORITY_CLASS*）				
Thread Priority（*THREAD_PRIORITY_xxx*）	*IDLE*	*NORMAL* (Background)	*NORMAL* (Foreground)	*HIGH*	*REALTIME*
IDLE	01h	01h	01h	01h	10h
LOWEST	02h	05h	07h	0Bh	11h~16H
BELOW_NORMAL	03h	06h	08h	0Ch	17h
NORMAL	04h	07h	09h	0Dh	18h
ABOVE_NORMAL	05h	08h	0Ah	0Eh	19h
HIGHEST	06h	09h	0Bh	0Fh	1Ah
TIME_CRITICAL	0Fh	0Fh	0Fh	0Fh	1Bh~1Fh

表 6-2　Win32 priority classes。表中所列是優先權值。

例如，下面的 Win32 calls 會把一個 Win32 thread 的優先權設爲 10h：

```
SetPriorityClass(hThread, REALTIME_PRIORITY_CLASS);
SetThreadPriority(hThread, THREAD_PRIORITY_IDLE);
```

Time-slicing scheduler 會把 CPU 時間分割給所有擁有最高優先權的 Win32 threads，給予每個 thread 一個時間配額（time-slice）。其基本演算法就是建立一個「最高優先權 threads」串列，並給予每個 thread 一個優先權調漲值（priority boost）。同時，time slicer 利用一個計時器中斷（timer interrupt）來排班，當 current thread 的時間配額終了，就強制插斷（preempt）其執行，於是排程器就給予下一個 thread 一點點時間。依此類推。Time-slicing scheduler 另外還應用其他一些附加演算法，讓系統運轉得更平順。例如：

- 如果有一個 thread 在某個合理的時間長度內一直未執行，排程器便提昇其優先權，以避免 starvation（飢餓）情況發生。至於何謂「合理的時間長度」，必須視時間配額（time slice）的平均值、排班中的 threads 個數、系統內部的演算法而定。

- 如果一個 thread 擁有一筆資源（例如一個 semaphore），而在該資源上頭有一個較高優先權的 thread 被凍結（blocked），那麼排程器會將前者（owning thread）的優先權提昇到和後者（blocked thread）一樣。「優先權繼承性質」使得「優先權較低之資源擁有者」在「優先權較高之資源需求者」面前仍可揚長而去。
- 如果一個 thread 被凍結（blocked），稍後它被解凍時，排程器會將其優先權提高，這可以令「搶奪資源的 thread」和「不搶奪資源的 thread」獲得相同的 CPU 平均時間。
- 當時間配額（time slice）終了，排程器會將 thread 的優先權從提昇值衰減至基礎值。而在其他情況，當引起優先權提昇的工作被完成之後，排程器會取消提昇（unboost）。

簡言之，time-slicing scheduler 服從 Win32 應用程式給它的指令，以及它自己的演算法規則，產生出 0h~1Fh 的優先權值。它呼叫 primary scheduler 的 *Adjust_Thread_Priority* service 建立起 thread 的優先權，然後實施優先權提昇，以說服 primary scheduler 將最適當的 thread 列爲接下來要執行的目標。硬體中斷和 critical section boosts 會影響優先權，於是也就實質影響了 Win32 優先權體制。你可以產生出兩個 threads，擁有相同的 Win32 優先權，而且都擁有 time-critical boost，那麼，由於時間配額之故，它們將共享 CPU 時間。

VxD 可以使用 *LOW_PRI_DEVICE_BOOST* 來增加「thread 適時完成任務」的可能性，不過這樣的調昇值只有 10h。把這樣的調昇施行於 *NORMAL_PRIORITY_CLASS* 且 *THREAD_PRIORITY_IDLE* 的 thread，會導至優先權從 1h 跳昇爲 11h。於是它就被放在 *REALTIME_PRIORITY_CLASS* threads 以外的所有 threads 的前面。如果你以 *HIGH_PRI_DEVICE_BOOST* 來調昇同一個 thread，導至優先權從 1h 跳昇爲 1001h，就會甚至移到 *REALTIME_PRIORITY_CLASS* threads 的前頭。不管哪一種情況，其餘 threads 仍然落在那些有較高調昇值（boosts）的 thread 身後。

第 7 章

虛擬裝置驅動程式之相關技術
Virtual Device Driver Mechanics

我絕對敢說，虛擬裝置驅動程式（VxD）的設計技術很神秘。不只因爲 VxDs 在 Ring0 執行（在那裡它們常常使用大部份程式員只聽過而未用過的 CPU 高權級性質），同時也因爲它們使用 API calls 來完成一些奇怪而難以理解的工作。不僅如此，VxD 程式設計還用到一些絕大多數程式員感覺陌生的非主流工具。別以爲你能夠走進軟體超級市場，從架上拿起一盒 Visual VxD IDE（Integrated Development Environment，整合開發環境）。不，沒這回事，你所能找到的最接近產品是 Vireo Software 公司出品的 VToolsD，我將在本章稍後描述這個產品。除非你使用這個產品，否則目前的 VxD 程式設計工作必須以 assembly 語言或 C 語言完成，並以 Microsoft Device Driver Kit（DDK）來處理你的程式檔。

這一章中，我要討論如何建造 VxD。如你所見，這將包括如何產生一個可執行檔（有著老舊而奇特的檔案格式）。我還會討論如何以 assembly, C, C++ 語言來撰寫 VxD 程式。即使你打算以高階語言來開發 VxD，你也得讀 assembly 語言這一部份，因爲當你擁有這部份基礎，你才能夠清楚兩個重要的觀念：**Device Description Block**（**DDB**）和 **Device Control Procedure**（**DCP**）。讀完此章，你將知道怎麼樣建造一個 VxD，然後你可以在

後續各章中獲得一些可以放進你的原始碼中的有用東西。

可執行檔格式（Executable File Format）

不管使用哪一種程式語言，你都需要建造出一個可執行檔（也就是你的驅動程式）。Windows 特別爲 VxD 使用了一種特殊的檔案格式，稱爲 **linear executable**（**LE**）格式。此格式由 OS/2 v2.0 首倡，可以同時內含 16 位元碼和 32 位元碼 -- 這的確是 VxD 所需要的，因爲有些 VxD 內含真實模式下的初始化區段（initialization section）。雖然再也沒有其他軟體使用 LE 格式了，Windows 還是爲了 VxD 而繼續使用它們。有很長一段時間，程式開發人員使用古老的 LINK386 聯結器（來自早期的 OS/2 2.0 beta）來產生 VxD 檔案。現在，微軟公司的語言工具已經內含一個 LINK 程式，可以製造出 LE 檔。其他語言工具所附的聯結器（特別值得注意的是 Watcom C/C++ 的 WLINK）也可以做出 LE 檔案。

聯結器和可執行檔格式

我想我正是「生活並呼吸於物件檔案格式之中」的那種人[1]，因此我想，瞭解可執行檔格式的分類，恐怕比瞭解各種工具的特性和限制更好一些：

- COM 檔案：內含 16 位元程式的二進位映像（binary image），沒有重定位資訊（relocation information）。DOS 可以載入並執行一個 COM 檔，但此檔案之所以能夠執行，是因爲它不需要重定位（relocation）。
- MZ 格式：這是 DOS 所瞭解的可重定位格式。MZ 可執行檔內含 16 位元程式碼和資料，它們統統和重定位資訊共處，所以 DOS 可以把程式放在任何它想要的地方。本方塊說明中的每一個更複雜的檔案格式，一開始都是以一個 MZ-format stub 爲前導。這個 stub（一小段存根聯、戳記） 通常只用來輸出一個警告訊息，但它也可以是個複雜程式。MZ 表頭中有一個欄位，指向檔案中真正的檔案表頭。

[1] 語見 Adrian King 的 *Inside Windows 95*（Microsoft Press，1994）p143, n26。

- NE（new executable）格式：內含 16 位元 Windows 程式或 OS/2 程式。NE 格式的關鍵特性是：它把程式碼、資料、資源隔離在不同的可載入區塊中。它也藉由符號輸入和輸出（symbolic imports and exports），實現所謂的執行時期動態聯結（runtime dynamic linkage），
- LE（linear executable）格式：內含 16 或 32 位元的程式碼和資料（或是其混合體），以及資源。OS/2 2.x 就是使用此一格式。LE 格式本是線性可執行檔的老祖宗，但是當 IBM 成爲 OS/2 的唯一發展者之後，它卻風華漸失，只在 VxDs 上繼續發揮光采。
- PE（portable executable）格式：內含 32 位元程式碼和資料，是 UNIX Common Object File Format（COFF）的演化品種。微軟的 32 位元作業系統（Windows NT, Windows 95 和 Win32s）都使用此一格式。PE 格式比其他格式優越的關鍵點在於它有依字母次序排列的 exports，以及一個可以「直接將程式影像映射到虛擬記憶體」的分頁映射組織（direct page mapping of the image into virtual memory）。

知道 LE 格式的歷史之後，我敢打賭，微軟正在尋找機會以 PE 取代 LE。如果不是因爲： (1) 偶而還需要在真實模式中執行 16 位元初始化程式碼 (2) 重新寫一個載入器畢竟有點困難 (3) 所有的 VxDs 都已經使用 LE 格式，我敢說瓜代的事情在 Windows 95 就已經發生了。據我所知，這樣的轉變其實已經發生在微軟自己的 SCSI 驅動程式和 network miniport 驅動程式身上。

VxDs 的節區架構（Segmentation of VxDs）

Windows 不只是爲 VxDs 使用了一個非標準的檔案格式，它也以非標準的方式使用那些檔案。一般而言，保護模式可執行檔會在不同的 sections 中放置程式碼和資料，檔案表頭中的各種屬性旗標即是用來導引載入器面對這些 sections 時的各種細節動作。但是 VxDs 卻將程式碼和資料混雜在 segments 之中，其屬性反映出它們（程式碼和資料）不

希望在執行時期有所變化。程式碼和資料之所以能夠混雜在一起，而仍然能夠有效運作，是因爲 VxD 所使用的 flat code selector 和 flat data selector 有相同的基底位址與大小。因此不論使用上述哪一個 segment 暫存器都可以取用程式碼或資料。表 7-1 顯示出一個 Windows 95 VxD 可能內含的 segments 及其屬性，每一個屬性都可以在 VxD 的模組定義檔（.DEF）中指定，稍後我會示範一個 .DEF 檔給你看。檢視這個表格時請注意，segment 屬性名稱並不一定代表你所以爲的意義。舉個例子， DBOCODE segment 的 CONFORMING 屬性並不會產生一個 conforming code segment，它只是一個標記，告訴 VxD 載入器說這個 segment 內含「只適於除錯」的程式碼和資料。

Segment 類別	說明	.DEF 檔的屬性
LCODE	Page-locked code 和 data	PRELOAD, NONDISCARDABLE
PCODE	Pageable code	NONDISCARDABLE
PDATA	Pageable data	NONDISCARDABLE, SHARED
ICODE	初始化所用的 code 和 data	DISCARDABLE
DBOCODE	除錯用的 code 和 data	PRELOAD, NONDISCARDABLE, CONFORMING
SCODE	Static code 和 data	RESIDENT
RCODE	真實模式的初始化 code 和 data	無，這是 VxD 中唯一一個 16 位元 segment
16ICODE	USE16 保護模式的初始化 data	PRELOAD, DISCARDABLE
MCODE	Locked message strings	PRELOAD, NONDISCARDABLE, IOPL

表 7-1 VxD 的 segmentation

VxD 中大部份的程式碼和資料是放在 LCODE, PCODE 和 PDATA 之中。需得一直存在於記憶體中的程式碼和資料則放在 LCODE 內，可被置換（page in 和 page out）的程式碼和資料放在 PCODE 和 PDATA 內。例如處理硬體中斷的程式碼，必須放在 locked pages（也就是 LCODE segment）之中，因爲當 Windows 嘗試服務硬體中斷的時候，它

沒有辦法處理 page fault。硬體中斷服務常式所使用的資料，也必須放在 locked pages 中。然而，Windows 可以對「用來處理 application time events」的程式碼做 paging 動作。做爲一個 VxD 的撰寫者，你得自行決定你的程式碼和資料是可以安全地被 paged 呢，或是它必須被鎖定在實際記憶體（physical memory）中。

VxD 載入器也接受以下幾個特殊目的的 segments。ICODE segment 內含初始化程式碼和資料，一旦所有驅動程式都完成了初始化工作，VMM 可以把 ICODE 丟棄，節省虛擬記憶體（可能是實際記憶體，也可能是置換檔空間）。DBOCODE segment 內含的程式碼和資料只有當你在除錯器控制之下執行時才派得上用場。例如，*Debug_Query*（系統控制訊息）的處理常式就屬於這個 segment。RCODE segment 內含 16 位元碼和資料，用於真實模式的初始化動作。MCODE segment 內含的竟是訊息字串（而非名稱所暗示的 "CODE"），將和 MSGMACRO.INC 中的巨集一起被編譯（請看 Windows 95 DDK 的 ***Kernel Services Guide*** 手冊中的 "Message Macros" 一節，以瞭解所謂的訊息巨集。它可以幫助你產生國際版的驅動程式）。

16ICODE segment 讓你能夠方便地定義一些「打算在初始化時，從保護模式拷貝到 V86 模式」的程式碼。這是一個 USE16 segment，將在初始化之後被丟棄。別被其名稱迷惑了，你並不能夠在這個 segment 中執行程式碼，反倒是你可以在 VxD 的 32 位元保護模式初始化程式碼中，把這個 segment 的內容視爲 data。舉個例，假設你要產生一段小型的真實模式碼做爲一個精緻體系的一部份，用以修改某個真實模式驅動程式（或常駐程式，TSR）的行爲，你的一小段碼只不過是要將 DS:BX 中的一個 word 載入到 AX 暫存器然後回返。你可以嘗試把下面的碼放在一個正規的 USE32 code segment 或 data segment 中：

```
VxD_INIT_DATA_SEG
fragment: mov ax, [bx]    ; don't do this in a USE32 data segment!
          ret
VxD_INIT_DATA_ENDS
```

然而你得不到預期的結果。當你把這段碼從 USE32 segment 拷貝到 USE16 segment（真實模式）中，第一個指令的意義有了巧妙的變化，它會變成 MOV EAX, [EDI]。如果把這段碼放到 16ICODE segment 中，由於 16ICODE segment 是一個 USE16 segment，組譯器（assembler）會爲這個 16 位元指令產生正確的 bit pattern，如你所願。

VxD 還可以內含一些 static code 和 data，放在 SCODE segment 中。動態載入驅動程式時，可能需要用到這個 segment，以解決一個獨特的問題。有時候同一 VxD 會在某個 Windows session 中被動態載入（和卸載）一次以上，這時候它需要記住一個個 instances 的狀態資訊（state information）。舉個例，VMM 不能夠摧毀應用程式的 callback 物件，所以被動態載入的 VxD 如果要產生一個 callback，它必須提供一個 static callback 函式，並在 static data 中記錄該 callback 的位址。

注意 Windows 3.1 只提供 LCODE 和 ICODE segments。雖然 Windows 3.1 的 DDK 提供了 segmentation 巨集，可以區分 "locked" 和 "pageable" 等不同特性的 segments，但那些巨集卻總是產生出 LCODE segments。

VxDs 的聯結選項

由於你即將在 VxD 開發過程中使用命令列（command-line）工具，所以你必須自己動手寫 MAKE 檔。Microsoft NMAKE 程式可以讀取下面這樣的標準 VxD 聯結動作（使用 Microsoft Visual C++ 聯結器）：

```
myvxd.vxd : $*.obj $*.def
    link /vxd /nod /map:$*.map /def:$*.def $*.obj
```

如果你對 MAKE 語法不熟，請看 MASM 的 ***Environment and Tools*** 手冊。

譯註：讓我對出現在上面的 MAKE-file 的語法做一點說明。$* 是 MAKE 工具的預設巨集，意思是「展開成爲欲建立之檔案名稱，但不含副檔名」，所以上述兩行相當於：

```
myvxd.vxd : myvxd.obj myvxd.def
    link /vxd /nod /map:myvxd.map /def:myvxd.def myvxd.obj
```

Makefile 最主要的精神就是把 ':' 左右兩端的檔案拿來比一比，如果右邊的任何一個檔案比左邊的檔案新，就執行下一行指令（一個 DOS 命令）。於是上述的 makefile 內容可解釋爲：如果 myvxd.obj 或 myvxd.def 比 myvxd.vxd 新（表示原始碼改變過），就執行 link 動作。

Microsoft Visual C++（甚至是 4.0 版）所附的聯結器並不能夠用來建造 VxDs。你必須使用由 Windows 95 DDK 所附的另一個聯結器（2.60.5046 版）（譯註：此話明顯與上一段牴觸，而且我使用 VC++ 所附之聯結器並無問題）。你會獲得大量的聯結警告訊息，但因爲它們並不影響驅動程式的運作，所以你可以忽略那些警告。

對於聯結器 LINK386（必須是 1.02.004 之後的版本），你可以寫這樣的 MAKE 語法：

```
myvxd.vxd : $*.obj $*.def
    link386 /noi /nod /map /li $*,$@,$*,,$*;
```

不論哪一種情況，你都必須提供一個模組定義檔。你可以使用放在 Windows 95 DDK 的 \BASE\SAMPLES\GENERIC 子目錄下的一個標準的 GENERIC.DEF，內容如下：

```
#0001  VXD MYVXD
#0002
#0003  DESCRIPTION 'MYVXD VxD for Microsoft Windows'
#0004
#0005  SEGMENTS
#0006    _LPTEXT     CLASS 'LCODE'   PRELOAD NONDISCARDABLE
#0007    _LTEXT      CLASS 'LCODE'   PRELOAD NONDISCARDABLE
#0008    _LDATA      CLASS 'LCODE'   PRELOAD NONDISCARDABLE
#0009    _TEXT       CLASS 'LCODE'   PRELOAD NONDISCARDABLE
#0010    _DATA       CLASS 'LCODE'   PRELOAD NONDISCARDABLE
#0011    CONST       CLASS 'LCODE'   PRELOAD NONDISCARDABLE
#0012    _TLS        CLASS 'LCODE'   PRELOAD NONDISCARDABLE
#0013    _BSS        CLASS 'LCODE'   PRELOAD NONDISCARDABLE
#0014    _LMSGTABLE  CLASS 'MCODE'   PRELOAD NONDISCARDABLE IOPL
#0015    _LMSGDATA   CLASS 'MCODE'   PRELOAD NONDISCARDABLE IOPL
#0016    _IMSGTABLE  CLASS 'MCODE'   PRELOAD DISCARDABLE IOPL
```

```
#0017    _IMSGDATA   CLASS 'MCODE'   PRELOAD DISCARDABLE IOPL
#0018    _ITEXT      CLASS 'ICODE'   DISCARDABLE
#0019    _IDATA      CLASS 'ICODE'   DISCARDABLE
#0020    _PTEXT      CLASS 'PCODE'   NONDISCARDABLE
#0021    _PMSGTABLE  CLASS 'MCODE'   NONDISCARDABLE IOPL
#0022    _PMSGDATA   CLASS 'MCODE'   NONDISCARDABLE IOPL
#0023    _PDATA      CLASS 'PDATA'   NONDISCARDABLE SHARED
#0024    _STEXT      CLASS 'SCODE'   RESIDENT
#0025    _SDATA      CLASS 'SCODE'   RESIDENT
#0026    _DBOSTART   CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
#0027    _DBOCODE    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
#0028    _DBODATA    CLASS 'DBOCODE' PRELOAD NONDISCARDABLE CONFORMING
#0029    _16ICODE    CLASS '16ICODE' PRELOAD DISCARDABLE
#0030    _RCODE      CLASS 'RCODE'
#0031
#0032  EXPORTS
#0033    MYVXD_DDB @1
```

上面這個檔案也放在書附光碟的 \CHAP07\ASM 子目錄中。你的 VxD 名稱出現在三個地方，但都不甚重要。.DEF 的內容重點有二：

- 你必須爲每一個 segments 指定正確的屬性。上述範例適用於 C 和 C++ 程式。對於 assembly 程式而言這個檔案或許太過了，不過依然適用就是！
- 你必須將 device description block（DDB）匯出（export），序號必須爲 1。

注意：對於「確認真正的驅動程式名稱」，我有一些經驗談。VMM 藉由 DDB 中記錄的名稱，才得以確認某個驅動程式。VMM 藉著尋找模組中的第一筆匯出（export）資料，才能夠找到 DDB。曾有其他書籍寫道，DDB 位於 VxD 的固定偏移位置上，那並不正確，你只需將它匯出就好，至於 DDB 本身叫什麼名字，無關宏旨！不過，Soft-ICE/W 卻會使用你在 .DEF 的 VXD 敘述句中指定的名稱。如果這個名稱和輸出檔名（.VXD）不符合，聯結器會給予一個警告，但這並不影響 VMM 和除錯器的歡喜接受。如果你在三個地方使用相同名稱（8 個字元）：(1) 在 *Declare_Virtual_Device* 巨集中（那最終將被放進 DDB 結構）(2) 在 VXD 敘述句 (3) 驅動程式檔案名稱，並且你唯一匯出（export）的符號就是 DDB，那麼就絕對沒有問題。

以 assembly 語言撰寫 VxDs

最原始的方法就是以數個 assembly 語言檔，加上表頭檔，並利用 DDK 提供的工具，建造出 VxD。我所謂的工具包括 32 位元組譯器（assembler）。MASM 6.0 （或更新版本）及其他廠牌的組譯器，現在已經具備處理 16 位元程式和 32 位元程式的能力。稍早時候微軟公司曾經在其 DDK 中夾帶 MASM 5.10B，因爲當時 32 位元組譯器很少。

假設你要以 assembly 語言產生一個 MYVXD.VXD（請看**圖 7-1**）。一開始你需要寫一個 MYVXD.ASM，看起來像這樣：

```
    .386p
    include vmm.inc             ; 一定要
    include debug.inc           ; 可有可無
Declare_Virtual_Device ...      ; 稍後詳述

Begin_Control_Dispatch myvxd
    [Control_Dispatch macros]
End_Control_Dispatch   myvxd

VXD_IDATA_SEG
    [Initialization-only data]
VXD_IDATA_ENDS

VXD_ICODE_SEG
    [Initialization-only code]
VXD_ICODE_ENDS

VXD_LOCKED_DATA_SEG
    [Page-locked data]
VXD_LOCKED_DATA_ENDS

VXD_LOCKED_CODE_SEG
    [Page-locked code]
VXD_LOCKED_CODE_ENDS

    end
```

爲這個骨幹程式添枝添葉之後，請對此 MYVXD.ASM 做組譯動作，產生一個 COFF 格

式的 OBJ 檔，然後再以聯結器處理這個 OBJ 檔，成爲一個 LE 格式的 VxD 可執行檔。你還可以使用其他工具爲此 VXD 產生一個符號檔，以備除錯之用。最終目標是一個副檔名爲 .VXD 的驅動程式。

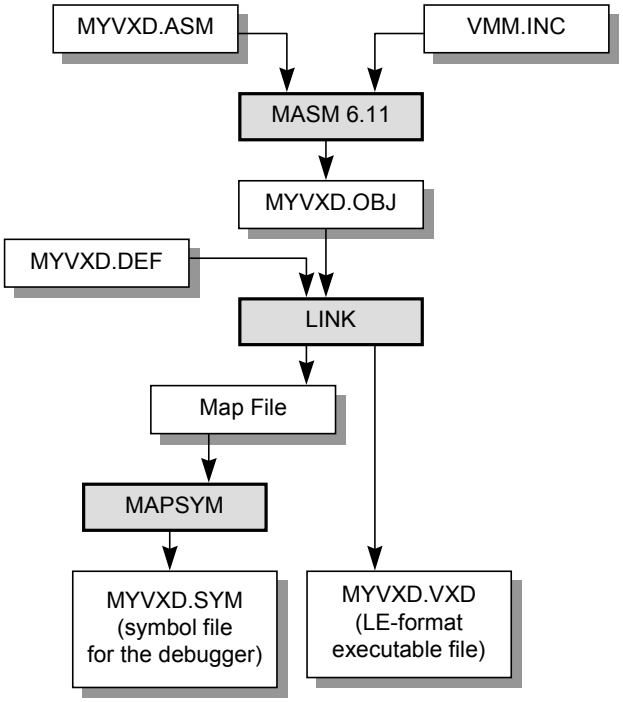


圖 7-1　建造一個以 assembly 語言完成的 VxD

注意：COFF 雖然是 UNIX 標準，但如今的 COFF 已經被 Microsoft 做了大量的變化，並且常常有一些未公開的擴充內容。稱呼這樣的格式爲 COFF，就好像稱呼 end user 爲哺乳動物一樣 -- 技術上沒有錯，但實際上不能帶給你太多資訊。

Declare_Virtual_Device 巨集

VMM 從哪裡獲得 VxD 的重要資訊？device description block（DDB）是也！請看**圖 7-2**。你可以使用 *Declare_Virtual_Device* 巨集爲你的驅動程式產生一個 DDB，然後在 .DEF 檔中將此 DDB 匯出。*Declare_Virtual_Device* 接受數個參數：

```
Declare_Virtual_Device Name, MajorVer, MinorVer, CtrlProc, DeviceNum,
   InitOrder, V86Proc, PMProc, RefData
```

```
struct VxD_Desc_Block {
   ULONG DDB_Next;                 // 00h chain to next DDB (reserved)
   USHORT DDB_SDK_Version;         // 04h DDK version used to build driver
   USHORT DDB_Req_Device_Number;   // 06h unique device ID
   UCHAR DDB_Dev_Major_Version;    // 08h major version number
   UCHAR DDB_Dev_Minor_Version;    // 09h minor version number
   USHORT DDB_Flags;               // 0Ah flags used by VMM
   UCHAR DDB_Name[8];              // 0Ch 8-byte name of driver
   ULONG DDB_Init_Order;           // 14h initialization ordinal
   ULONG DDB_Control_Proc;         // 18h control procedure address
   ULONG DDB_V86_API_Proc;         // 1Ch V86 API entry (if any)
   ULONG DDB_PM_API_Proc;          // 20h PM API entry (if any)
   ULONG DDB_V86_API_CSIP;         // 24h V86 version of API entry addr
   ULONG DDB_PM_API_CSIP;          // 28h ring-3 version of API entry addr
   ULONG DDB_Reference_Data;       // 2Ch reference data from RM init
   ULONG DDB_Service_Table_Ptr;    // 30h address of service table (if any)
   ULONG DDB_Service_Table_Size;   // 34h number of services
   ULONG DDB_Win32_Service_Table;  // 38h address of VWIn32 service table
   ULONG DDB_Prev;                 // 3Ch chain to previous DDB (reserved)
   ULONG DDB_Size;                 // 40h size of this block (reserved)
   ULONG DDB_Reserved1;            // 44h reserved
   ULONG DDB_Reserved2;            // 48h reserved
   ULONG DDB_Reserved3;            // 4Ch reserved
};
```

圖 7-2　Device Description Block（DDB）的結構佈局

Name 參數是一個 8 bytes 長的 device 名稱。此名稱在系統之中應該獨一無二。這個名稱一般而言應該和 VXD 的檔名以及你在 .DEF 檔的 VXD 敘述句中指定的名稱相同。*Declare_Virtual_Device* 巨集會把 *_DDB* 附加到此名稱之後，產生一個內部使用的符

號，代表此一 DDB。因此，MYVXD 將會有一個 DDB 名爲 *MYVXD_DDB*。

參數 *MajorVer* 和 *MinorVer* 用來指定驅動程式的版本號碼。你可以任意指定其值。通常開發人員會指定此一驅動程式的執行平台（Windows）的版本號碼。請注意 DDB 欄位是以 big-endian 方式來儲存這兩個值，所以程式碼在處理它們時必須先將其值顛倒過來，像這樣：

```
mov     ax, word ptr MYVXD_DDB.DDB_Dev_Major_Version
xchg    ah, al          ; AH = major, AL = minor
```

所謂 big-endian order 這是資料在記憶體中的一種組織方式，"most significant byte" 放在最前，而不是最後。Intel 處理器是以 little-endian 方式來放置資料，所以 "most significant byte" 放在最後面。

CtrlProc 參數指明 **device control procedure**，用來處理流到你的驅動程式中的系統控制訊息（system control messages）。稍後我將詳細討論這個重要的函式。如果你遵循一般規則，你應該指定類似 *MYVXD_Control* 這樣的名稱。

某些驅動程式需要一個獨一無二的識別碼（ID），如此一來其他軟體才能夠找到它們。你可以在 *DeviceNum* 欄位指定此一識別碼。這是一個 16 位元整數。如果你不需要這樣的識別碼，可以在此欄位中放置 *UNDEFINED_DEVICE_ID*。只有以下情況才需要指定一個獨一無二的識別碼（ID）：

- 某些常駐程式（TSR）使用 INT 2Fh 子功能 1605h 來載入你的驅動程式。這種情況下驅動程式需要一個真實模式的初始化區段（initialization section）以避免重複發生，也需要一個獨一無二的識別碼讓 Windows 能夠決定是否有重複發生。
- 你的驅動程式在初始化階段使用 INT 2Fh 子功能 1607h 向所有真實模式驅動程式廣播它的存在。這時候，凡是看到你的廣播的程式，會將暫存器 BX 的值拿來與你的驅動程式識別碼做比較。

- 你的驅動程式匯出(exports)VxD services 給其他驅動程式使用。INT 20h service API 以 service ID 的較高一半（16 位元）做爲驅動程式識別碼。

如果因爲上述理由而需要一個識別碼，你可以根據 VXDID.TXT（位於 Windows 95 DDK 的 \BASE\SAMPLES 子目錄）中的說明，要求 Microsoft 提供一個。事實上現在已經少有驅動程式需要擁有自己的識別碼了，以前之所以需要，是爲了支援 INT 2FH 子功能 1684H，此介面用來找出一個驅動程式的 V86 模式或保護模式的 APIs。在 Windows 95 之中，如果你知道一個驅動程式的 DDB 名稱，你就可以找出它的 API。你也可以使用 INT 2Fh 子功能 168Ah，這是 DPMI 規格中的一個由廠商指定的 16 位元應用程式服務項。在這項服務中，應用程式使用 INT 2Fh 子功能 168Ah，並讓 DS:[E]SI 指向一個你所定義的字串（以 null 爲結束字元）。這個字串取代了識別碼，識別碼因而成爲多餘。開放 services 給 32 位元程式的唯一方法是經由 *DeviceIoControl* 函式，而此函式需要知道的是你的驅動程式檔名，不是識別碼。

警告　在 Windows 3.0 DDK 中，Microsoft 建立了一個規則，提供 16 位元識別碼中的 11 個位元給最多 2048 家軟體廠商，剩餘的 5 個位元用來指定最多 32 個 VxDs。Microsoft 很快就發現這不是個好方法，於是它決定不再給任何提出要求的人 32 個識別碼空間。如今，你必須清楚說明你爲什麼需要識別碼，並且一次只能獲得一個。如果其他人盜用了你的識別碼，你就得承擔破壞他人的程式的風險。

每一個驅動程式需要一個初始化序號，指定自己與其他驅動程式在系統中的次序關係。*Declare_Virtual_Device* 巨集的 *InitOrder* 參數允許你指定這個數值。如果你不在乎其值，請指定 *UNDEFINED_INIT_ORDER*。

驅動程式可以匯出（export）一個 API 給 V86 模式或保護模式（或兩者共同）使用。*Declare_Virtual_Device* 巨集的第 7 和第 8 個參數用來指定這個 API 的位址。如果你不想供應 API，可以忽略這兩個參數。

你可以利用 *Declare_Virtual_Device* 巨集的最後一個參數指定一個 reference data。這麼

做的唯一理由是你想要建立一層驅動程式，和 I/O Supervisor（IOS）搭配使用。這種情況下你可以宣告你的 virtual device 帶有一個 reference data 欄位，指向一個特殊的 driver registration packets，於是 IOS 可以在呼叫你的任何一個常式之前，快速探查有關於你的驅動程式的一些事實。

一般的寫碼規矩

以 assembly 語言撰寫 VxD 程式，必須遵循某些規矩。幸運的是比起撰寫 Windows 程式，這些規矩少得多了。不過你還是得知道它們。

暫存器（Register）和旗標（Flag）的使用

大部份時候，VxD 可以自由使用 general 暫存器，以及 FS 和 GS 兩個 segment 暫存器。許多 VxD-level 常式都會假設 direction flag 是空的，所以字串動作像 MOVS, CMPS 等等都會累增其運算元（operand）位址。你可以暫時改變 direction flag，但應該在 return 的時候或呼叫另一個 VxD 的時候，把 direction flag 恢復原狀。你也可以在短暫時間內將中斷 disable，但你應該記得恢復 interrupt flag 的原先狀態，而不只是做一個 STI 動作：

```
pushfd          ; save current interrupt-enable state
cli             ; disable interrupts BRIEFLY
...
popfd           ; restore interrupt flag
```

一般人可能會告訴你說絕對不要在 VxD 中改變 CS, DS, ES, SS 暫存器。我不打算說明爲什麼，我只要告訴你，絕對不要改變 CS（例如進行一個 far call）或 SS，除非在非常不尋常的情況下，也就是說當你確定絕對不會有 page faults 或 interrupts 發生時。同時你應該知道，觸及一個恐怕已被鎖定（locked）的 V86 page，而其中內含你要引用的資料時，可能會引發 page fault。你可以改變 DS 和 ES，只要在呼叫任何 VMM 或 VxD services 之前恢復其原值就好。不過，在改變這些暫存器之外，你其實有更安全的替代方案。舉個例，*Map_Flat* 是一個被大家認可的方法，以 VM 的 selector 和 offset 產生一個 flat 位址，它能夠做到 native LDS 或 LES 指令做不到的事。如果你只是要改變堆

疊，可使用 *_Call_On_My_Stack* 以避免 VMM 在行程切換（context switches）時會執行的各種繁瑣的簿記工作。

Service 的呼叫規則

一旦你寫出自己的 VxD service entry points（第 9 章主題），你應該公開一份文件，告訴大家你保留了哪些暫存器。每一個函式都必然會復原 ESP 暫存器，或其他那些沒有辦法 return 給呼叫者的暫存器。「以暫存器來傳遞參數」的 services，通常會保存所有暫存器（內含輸出參數者，除外）。C-style services 則必須保存 EBX, ESI, EBP 暫存器。

Segmentation 巨集

我們不再直接指定 segment 名稱，而是使用定義於 VMM.INC 和 VMM.H 中的各式各樣巨集（陳列於表 **7-2**）。

Segment	VMM.INC 或 VMM.H 中的巨集
_LTEXT	VxD_LOCKED_CODE_SEG
_LDATA	VxD_LOCKED_DATA_SEG
_PTEXT	VxD_PAGEABLE_CODE_SEG
_PDATA	VxD_PAGEABLE_DATA_SEG
_DBOCODE	VxD_DEBUG_ONLY_CODE_SEG
_DBODATA	VxD_DEBUG_ONLY_DATA_SEG
_ITEXT	VxD_INIT_CODE_SEG
_IDATA	VxD_INIT_DATA_SEG
_STEXT	VxD_STATIC_CODE_SEG
_SDATA	VxD_STATIC_DATA_SEG
_RCODE	VxD_REAL_INIT_SEG

表 7-2　Segmentation 巨集（使用於 assembly 語言或 C 語言）

這些巨集使你得以輕鬆改變 segment 名稱或 grouping conventions，只要重新編譯你的原始碼就好。這些巨集的使用看起來像這樣：

```
VxD_LOCKED_CODE_SEG
...
VxD_LOCKED_CODE_ENDS
```

Flat Addresses（平澤式位址）

使用 OFFSET32 巨集，就可以指定 flat 位址。這個巨集會被擴展爲 *offset flat:*，我們可以稍微節省一些鍵盤輸入動作。如果使用早期的開發工具，你必須小心，不要光只輸入 *offset* 卻忘了加上 *flat:*，組譯器和聯結器會因此共謀產生不正確的重定位資訊（relocation information）。下面是一個簡單的例子：

```
functbl   label     dword
          dd        offset32 function0
          dd        offset32 function1
          ...
          jmp [functbl + 4*eax]
```

爲 Procedures 劃界

現在我們改用 BeginProc 和 EndProc 巨集（不再使用 *proc* 和 *endp* 這些 assembly directives），爲函式劃清界線：

```
BeginProc      OnDeviceInit, init
...
EndProc        OnDeviceInit

BeginProc      MYVXD_Get_Version, SERVICE, HIGH_FREQ
...
EndProc        MYVXD_Get_Version
```

組譯或除錯時，這些巨集都會產生有用的 “logging information”。

參數	說明
HIGH_FREQ	表示這個函式常常被呼叫，並指示組譯器（assembler）對此函式做 DWORD-align 的動作。
PUBLIC LOCAL	表示這個函式的生存範圍（scope）
PCALL　　SCALL CCALL ICALL	指定這個函式使用一個特殊的呼叫慣例（calling convention，如 Pascal, Standard, C 或 default）
ESP	指示組譯器（assembler）在處理 *ArgVar* 和 *LocalVar* 變數時使用 ESP 暫存器做爲 stack frame。
HOOK_PROC, *label*	表示此函式可以經由 *Hook_xxx_Fault* 或 *Hook_Device_Service* 安裝。使用 *label* 變數是爲了 chaining。
LOCKED PAGEABLE INIT etc.	指示組譯器（assembler）把這個函式放在 VxD_xxx_CODE_SEG segment 之中。
SERVICE ASYNC_SERVICE	表示這個函式是被 *VxDCall* 呼叫。
NO_LOG NO_PROFILE NO_TEST_CLD NO_PROLOG	將各種檢查項 disable 掉，否則除錯版中會做這些檢查。
TEST_BLOCK TEST_REENTER NEVER_REENTER NOT SWAPPING	指定呼叫端的各種環境特性（在建立除錯版時）
W32SVC	表示這是一個 Win32 service

表 7-3　BeginProc 巨集的一些可有可無的參數

表 **7-3** 列出你可以在 BeginProc 巨集中指定的一些可有可無的參數。其中有一些可以讓你寫程式時更輕鬆些，譬如說你可以指定函式所在之 segment 名稱，這可避免明顯使用那些 segmentation 巨集，並給你自己更多的彈性，以更邏輯的方式來安排原始碼。

以 C 語言呼叫 VxD 函式

如果想要寫一個 assembly 函式，而又能夠被 C 程式呼叫，你可以使用 ArgVar 和 LocalVar 巨集來定義參數變數和區域變數。你可以使用 EnterProc、LeaveProc、return 等巨集來產生 prolog 和 epilog 碼。下面就是 Microsoft 的 SERIAL.VXD 對於其 *PortSetState* 函式（一個被 VCOMM 呼叫的 port driver 函式）的定義：

```
BeginProc  PortSetState, CCALL, PUBLIC

ArgVar  hPort, DWORD
ArgVar  pDcb, DWORD
ArgVar  ActionMask, DWORD

LocalVar  BaudRateChange, DWORD

       EnterProc
       [body of procedure]
       LeaveProc
       return

EndProc  PortSetState
```

其中 BeginProc 的參數之一 *CCALL* 表示這個函式將由 C 程式來呼叫，*PUBLIC* 表示函式名稱可以在此 assembly 模組之外被看到（如此才能夠被呼叫啊）。

上述三個 ArgVar 巨集宣告出此函式的三個參數，LocalVar 巨集則定義了一個放置在 stack 的區域變數。在這些巨集之中，你可以使用 *BYTE, WORD, DWORD* 做爲第二參數，指示變數的大小。你也可以使用其他述句（expression）來指定變數長度。

對於一般的 Windows 程式，這些巨集是多餘的，因爲 MASM 6.x 在 *proc* directive 中已內建這些機能。不過你一定不會在 VxDs 中使用 *proc* directives，因爲 BeginProc 內含了其他一些必要機能。

Device Control Procedures

每一個 VxD 都需要一個 **device control procedure** 來對 **system control message**s 做出

反應。把它想像是個 *switch* 或是 MFC 的 message map 吧！你的碼看來應該這樣子：

```
Begin_Control_Dispatch vxdname

Control_Dispatch message, function
...
End_Control_Dispatch vxdname
```

這個 Begin_Control_Dispatch 巨集會強迫將程式碼組譯到 locked code segment 中，並宣告一個函式名爲 *vxdname_Control*。每一個 Control_Dispatch 巨集都用來指定某個函式處理某個 system control messages。End_Control_Dispatch 巨集用來關閉整個 control procedure，它會清除 carry flag（表示成功）並回返。這樣的巨集組合也可以在一個 if-then-else 或是一個 branch table 中做自動選擇，完全視怎麼做最合理而定。

一個完整的 assembly 語言骨幹程式

光靠上面提供的一點點觀念，就想寫出一個具代表性而又富意義的 VxD 範例程式，實在是有困難！但如果把這一節的所有觀念放在一起，就獲得下面這個 assembly 語言的 VxD 骨幹程式（你可以在書附光碟的 \CHAP07\ASM 子目錄中找到它）：

MYVXD.MAK

```
#0001  # MYVXD.MAK -- MAKE file for MYVXD.VXD
#0002
#0003  all: myvxd.vxd
#0004
#0005  myvxd.obj:
#0006     ml -coff -DBLD_COFF -DIS_32 -W2 -c -Cx -DMASM6 -Zd -DDEBUG $*.asm
#0007
#0008  myvxd.vxd: $*.obj $*.def
#0009     c:\ddk\bin\link @<<
#0010  /vxd /nod
#0011  /map:$*.map
#0012  /def:$*.def
#0013  $*.obj
#0014  <<
```

MYVXD.ASM

```
#0001  ;==============================================================
#0002  ;  MYVXD.ASM -- The (almost) simplest possible VxD in assembly language
#0003  ;  Written by Walter Oney
#0004  ;==============================================================
#0005
#0006     name  myvxd
#0007     .386p
#0008     include vmm.inc
#0009     include debug.inc
#0010
#0011  Declare_Virtual_Device MYVXD, 1, 0, MYVXD_Control, \
#0012     Undefined_Device_ID, Undefined_Init_Order
#0013
#0014  Begin_Control_Dispatch MYVXD
#0015     Control_Dispatch Device_Init, OnDeviceInit
#0016  End_Control_Dispatch   MYVXD
#0017
#0018  BeginProc OnDeviceInit, init
#0019     clc           ; indicate no error
#0020     ret           ; return to VMM
#0021  EndProc  OnDeviceInit
#0022
#0023     end
```

組譯這個範例程式　爲了對上述程式進行組譯與聯結，你必須安裝好 Windows 95 DDK 並設定好環境變數。請遵循書附光碟根目錄中的 README.TXT 的指示，然後在 DOS 視窗中鍵入以下命令：

```
C:\projectfolder> nmake -F myvxd.mak
```

聯結器會產生一些警告，不過沒有關係。

數頁之前我已爲你顯示了這個程式所需要的模組定義檔（.DEF）。

當然這個驅動程式什麼也沒做！它出現在這裡，只是爲了示範一個符合規則的完整 VXD 骨幹。

Windows 3.1 相容性　如果你要建立一個在 Windows 3.1 也能跑的 VxD，請在含入 VMM.INC 之前定義 WIN31COMPAT 常數，像這樣：

```
name myvxd
.386p
win32compat = 1
include vmm.inc
...
```

WIN31COMPAT 能使你的 VxD 與 Windows 3.1 相容，它會抑制 DDK 各個含入檔中與 Windows 95 有關卻不容於 Windows 3.1 的所有宣告。

以 C 或 C++ 撰寫 VxDs

我想大部份程式員比較喜歡以高階語言來寫 VxDs。C 或 C++ 比 assembly 語言更容易閱讀和維護，而且高階語言的程式邏輯（通常）比較佔優勢。但是以 C 或 C++ 語言來開發驅動程式一直有相當的困難，因爲：

- DDK 中的成份，像表頭檔以及說明文件，都是以 assembly 語言爲對象。
- 不容易找到 32 位元的 C/C++ 編譯器。
- 某些廠商的 C 編譯器很難使用 inline assembly。
- 大部份 VxD services 需要把參數放在暫存器中，而且它們都依賴 INT 20h（我在第 4 章描述過這個中斷）。
- 系統層次（system level）的除錯器沒有能夠處理高階語言，這種情況到今天還是一樣。

如今有兩個不錯的解決方案：Windows 95 DDK 和 Vireo Software 出品的 VToolsD。我將在這一節敘述這兩種方法。總的來說，對於那些已經知道底層在幹什麼的 C 程式員，DDK 是個相當好的工具。VToolsD 比 DDK 更容易使用，因爲它有一套 C++ class library。不幸的是，目前還沒有非常理想的 VxD source-level 除錯器，Nu-Mega Technologies 公司的 Soft-Ice/W 算是相當接近的一個產品了。

Windows 95 DDK

Windows 95 DDK 與早期的 DDK 不同，它非常鼓勵大家以 C 開發 VxD。C 表頭檔宣告了所有 service APIs 和資料結構，也內含所有你可能需要的註解。雖然其中還是有 assembly 語言的 .INC 檔，不過那是 Microsoft 以一個名爲 H2INC 的內部工具將 C 語言的 .H 檔轉換而得，也因此捨棄了 .H 檔內的文字註解（並因此破壞了可讀性）！

欲使用 DDK 來建立一個以 C 語言完成的 VxD，你必須這麼做：

- 撰寫用來處理 system control messages 的 C 函式，並加入任何適當的（必要的）"layered device architectures"（VCOMM、Plug and Play...等等）。
- 對於那些「你必須呼叫，但 Microsoft 卻沒有提供其外包函式（wrappers）」的 VxD services，請撰寫屬於你自己的外包函式（wrappers）。讓我告訴你吧，這可能成爲你的主要煩惱。
- 把你的「Declare_Virtual_Device 巨集呼叫」和你的 device control procedure 放進一個小小的 assembly 模組中（DDK 並沒有內含對等於此物的東西）。你必須在 Control_Dispatch 巨集中使用一些額外參數，以便將控制權移轉至 C 函式去。
- 爲那些接受「暫存器參數」的 callback 函式撰寫 assembly 語言的外包函式（wrappers）。這可能會成爲你的主要痛苦所在！

C 表頭檔

當你以 C 語言發展 VxD，至少會用到兩個表頭檔。這兩個必要的檔案是：

- BASEDEF.H - 定義基本的 typedef 符號以及常用常數如 BOOL, TRUE 等等。
- VMM.H - 定義出每個 VxD 都幾乎一定會用到的資料結構和常數。

表 **7-4** 列出 VxD 程式設計過程中可能用到的其他 DDK 表頭檔。其中最有用的是 DEBUG.H，允許你使用各式各樣的除錯函式和巨集，幫助你對 VxD 除錯。

表頭檔	目的
BASEDEF.H	提供基本的 typedefs（幾乎每個 VxD 都會用到）
BLOCKDEV.H	定義 block device interface
CONFIGMG.H	定義 Configuration Manager interface
DBT.H	定義 WM_DEVICECHANGE 訊息和 *BroadcastSystemMessage* 函式所需的常數
DEBUG.H	提供對大部份 VxDs 而言極有用的除錯服務
DOSMGR.H	定義 DOS virtualization manager interface
INT2FAPI.H	定義 Windows INT 2Fh interface 所需常數
IOS.H	定義 I/O Supervisor interface（IOS components 所需的眾多表頭檔之一）
NETVXD.H	提供 network VxD module 所需的 VxD 識別碼和初始化序號。
PCCARD.H	定義 PCMCIA card manager interface
SHELL.H	定義 SHELL device interface
VCOMM.H	定義 Virtual Communications Driver interface
VFBACKUP.H	定義 backup device interface
VMCPD.H	定義 math coprocessor device interface
VMM.H	定義所有 VxDs 都需要的 VMM interfaces
VMMREG.H	定義 VxDs 所需的 registry interface
VPICD.H	定義 Programmable Interrupt Controller（PIC）interface
VPOWERD.H	定義 power management interface
VTD.H	定義 timer chip interface
VWIN32.H	定義 Win32 application manager interface
VXDLDR.H	定義 VxD loader interface
VXDWRAPS.H	提供「外包函式的原型」（function wrapper prototypes）

表 7-4　DDK 中的 VxD 表頭檔

C 驅動程式中的分段（segmenting）

如果使用 DDK 並以 C 語言開發 VxD，你可以利用前置處理器(preprocessor)的 pragma 敘述句來控制分段（segmentation）問題。例如：

```
#pragma VxD_LOCKED_CODE_SEG
#pragma VxD_LOCKED_DATA_SEG
```

這會導至：

```
#pragma code_seg("_LTEXT", "LCODE")
#pragma data_seg("_LDATA", "LDATA")
```

熟悉 Microsoft C 編譯器的人應該知道這些 pragma 敘述句用來控制隨後出現之 segments 的功能和資料定義。表 **7-2** 列出你可以在 pragma 句子中使用的常見 segment 名稱。務請注意在 C 和 assembly 中對同一 segment 使用相同的符號。

DDK 表頭檔 VMM.H 之中也定義了一些 pragma 句子，使用於數個新增的 segments，像是 *VxD_VMCREATE_CODE_SEG, VxD_VMDESTROY_CODE_SEG* 等等。這些 segments 允許 Microsoft 將大型 VxD（例如 VMM）中的相關東西群組在一起，藉此降低那些 VxDs 的 working set（譯註：運轉時的一個單元）。你我可能都尚未需要操心這件事情！

外包函式（Function Wrappers）

VXDWRAPS.H 是極重要並且常常被用到的一個表頭檔。爲了解釋它做些什麼服務，我必須先說明程式如何喚起VxD services。在 assembly 語言之中，你可以使用 VMMCall 巨集或 VxDCall 巨集：

```
VMMCall Get_VMM_Version
VxDCall SHELL_Get_Version
```

這些巨集都會產生一個 INT 20h 指令，後面緊跟著一個數值，指出一個特定的 VxD 和

一個索引號碼，此號碼表示 VxD service table 中的函式編號。C 語言 VxD 還是會發出 INT 20h，但是 DDK 提供了一些東西，爲上述的醜陋形式做了點美化。第一個東西就是定義在 VMM.H 之中的「VxDCall 巨集」，。這個巨集是 interlocking scheme 的一部份，允許你這樣寫 C 語言驅動程式：

```
WORD version;
VxDCall(Get_VMM_Version)
_asm mov version, ax
```

任何人都能夠諒解我們不願看到程式中到處散落 inline assembly 碼(用來在 VxDCall 之後搬移暫存器值）的心情！所以，DDK 提供了一組所謂的 **wrappers**（外包函式），允許你以更自然的方式寫碼：

```
WORD version = Get_VMM_Version();
```

大部份 wrapper 的原型宣告都放在 VXDWRAPS.H 之中。函式碼本身如果不是以 inline 形式出現在程式中，就是以 wrapper library（例如 VXDWRAPS.CLB）的形式和你的程式聯結在一起。你可能會希望只要含入 VXDWRAPS.H 就可以立刻存取你所需要的 service calls 和資料結構。噢，DDK 沒讓你的日子這麼好過！我相信 Microsoft 已經爲那些必須生產硬體驅動程式的程式員（尤其是那些在特殊架構如 Plug & Play 或 VCOMM 之下飽受束縛的可憐人）好好地整修了 DDK，但如果你不在那些領域內，我想你會非常有挫敗感，除非你瞭解如何把 DDK 表頭檔平安地整合在一起。

你必須知道的第一件事情就是：VxD wrappers 的含入次序很重要，*WANTVXDWRAPS* 符號的出現也很重要。如果你需要含入 VXDWRAPS.H，請在含入任何 DDK 表頭檔之前先定義 *WANTVXDWRAPS*，並在最後才含入 VXDWRAPS.H：

```
#define WANTVXDWRAPS      // 抑制性宣告
#include <basedef.h>
#include <vmm.h>
...                       // 其他的 DDK 含入檔
#include <vxdwraps.h>     // 應該最後含入
```

上述的 #define 那行使 VXDWRAPS.H 和其他表頭檔之間不可能出現函式重覆定義的情況。舉個例子，VMM.H 和 VXDWRAPS.H 都宣告了 *Get_VMM_Version*，我們在最後才含入 VXDWRAPS.H，因爲它只宣告那些先前曾含入的表頭檔中的函式。唯有先前曾含入 VCOMM.H 檔，我們才能夠取得 VCOMM 的 wrappers 函式。

DDK wrappers 的另一個事實是，它迫使你必須做一些 segmentation 動作。VXDWRAPS.H 中所含的前置處理器巨集（preprocessor macro）可以爲真正有 wrappers 的函式宣告出六個不同的 wrappers。以 *Begin_Critical_Section service* 爲例，它就有**表 7-5** 所展示的六個 wrapper 函式。

Wrappers 名稱	內含於（某個 segment）
LCODE_Begin_Critical_Section	LCODE（Locked code）
ICODE_Begin_Critical_Section	ICODE（initialization code）
PCODE_Begin_Critical_Section	PCODE（pageable code）
SCODE_Begin_Critical_Section	SCODE（static code）
DCODE_Begin_Critical_Section	DCODE（debugging code）
CCODE_Begin_Critical_Section	CCODE（Configuration Manager）

表 7-5 Begin_Critical_Section service 的 wrappers

稍早對 VxD segmentation 的討論之中，我並沒有談到 CCODE。CCODE 內含和 Configuration Manager 互動的程式碼，做爲 Windows 95 的 Plug and Play 架構的一部份。就如我們在第 11, 12 章學習 Configuration Manager 時所將看到的，系統的這一部份非常重要，我建議你先不要急著嘗試瞭解這個主題，應該等到進行到那些章節再說。

擁有這些 wrappers 的重點在於，你可以輕鬆地在任何一個 code segment 中呼叫它們。如果你的碼放在 initialization segment 中，你可以呼叫 *ICODE_Begin_Critical_Section*；而在 locked code 中你可以呼叫 *LCODE_Begin_Critical_Section*。很明顯，這些以 segment 之名開頭的前置詞（prefix）會讓你的碼不容易寫也不容易讀，所以 VXDWRAPS.H 只

定義出一個名稱（如 *Begin_Critical_Section*）讓你的日子輕鬆一些。它看 CURSEG 巨集傳回什麼 segment（預設情況下是 LCODE）就定義對應的名稱。你可以重新定義這個巨集以改變預設情況，例如：

```
#include <vxdwraps.h>
...
// code that uses wrappers in LCODE
...
#pragma VxD_INIT_CODE_SEG
#pragma VxD_INIT_DATA_SEG
#undef CURSEG
#define CURSEG() ICODE
```

於是 *Begin_Critical_Section* 會變成 *ICODE_Begin_Critical_Section*。如果你打算對許多 segments 做改變，可能遵循 Vireo's VTOOLSD（一種工具產品）的模式比較好，它產生出一些含入檔，內含需改變之 segments 的「四聯句」（如上所示）。

你必須知道的關於 VXDWRAPS.H 的另一件事情是，除了 wrappers 函式原型，該檔案還含有一些 inline 函式宣告。例如，*End_Critical_Section* 函式（*Begin_Critical_Section* 的對應）被宣告成這樣：

```
VOID VXDINLINE End_Critical_Section(VOID)
    {
    VMMCall(End_Critical_Section);
    }
```

VXDINLINE 會被擴展爲 *static __inline*，其基本意義是說你自己可以擁有一份函式拷貝，不必從一個 object library 中取得。VMMCall 巨集和其 assembly 兄弟很像：產生一個 inline INT 20h 指令，後面跟著一個數值，指出某個 VxD service。

最後，令人抓狂的是，VXDWRAPS.H 竟然不夠完全，只涵蓋某些標準 VxDs。因此，如果你想呼叫 Virtual Keyboard Driver（VKD），你必須寫出自己的 wrappers。甚至它所支援的 VxDs 中，也不是每個 services 都定義有 wrappers。例如 VMM 大約有 400 個 services，但 VXDWRAPS.H 只支援其中大約 80 個。你可能不想呼叫

Call_When_Thread_Switch service，但如果你想，就得自己寫一個 wrapper。我還注意到某些詭異的不一致性和愚蠢表現，例如雖然 *Get_Profile_Hex_Int* service 有一個 wrapper，*Get_Profile_Decimal_Int* service 卻沒有！不僅如此，每一個 segment 都自稱要內含 *Get_Profile_Hex_Int* 的一個 wrapper，即使它只適用於初始化時間。你其實可以想像，只會有一個 *ICODE_Get_Profile_Hex_Int* wrapper。

我把這個表頭檔的細節都告訴你了，因爲我想在你嚴肅地以 C 和 DDK 著手一份 VxD 專案之前，你需要知道什麼是可以期望的。我所寫過的任何一個驅動程式都必須呼叫那些「DDK 並未提供 C wrapper」的 services，毫無例外。對 DDK 光碟中的 \DDK\BASE\VXDWRAPS 目錄內容研究一個小時，就足以讓你以簡單的 assembly 語言建立起你自己的 wrappers。不過我想完全避開 assembly 語言還是最理想的，你說是嗎？

以 C 語言完成 Device Control Procedures

以 assembly 語言完成的驅動程式，內含一個 device control procedure，並針對驅動程式所欲處理的每一個 system control message 呼叫 Control_Dispatch 巨集。以 C 語言完成的驅動程式，也有一個 assembly 語言所寫的 device control procedure，不過它的 Control_Dispatch 巨集用來指定 C 語言的訊息處理常式：

```
Control_Dispatch message, function, type, <arguments>
```

其中第三個參數可以是 *sCall*，表示是個 *__stdcall* 函式；或 *cCall*，表示是個 *__cdecl* 函式；或 *pCall*，表示是個 *__pascal* 函式。第四參數是一系列的函式參數，前後以 <> 括起來。

理論上你可以自由選擇要傳遞哪些參數給訊息處理常式，因爲 device control procedure 和訊息處理常式都是你寫的。很快你就會在訊息控制上面臨 assembly 語言規格的一個困境。以 Device_Init 爲例，當此訊息發生，EBX 暫存器將指向 System VM 的 VM control block，EXD 暫存器則內含由你的真實模式初始化函式所供應的參考資料（如果有的話）；assembly 語言寫成的 control procedure 可能長得像這樣：

```
Begin_Control_Dispatch myvxd
   Control_Dispatch Device_Init, OnDeviceInit, sCall, <ebx, edx>
End_Control_Dispatch myvxd
```

C 語言寫成的處理常式（由 Control_Dispatch 巨集的第二個參數指出）則長得像這樣：

```
BOOL __stdcall OnDeviceInit(PVMMCB hVM, DWORD refdata)
   {              // OnDeviceInit
   ...
   return TRUE;
   }              // OnDeviceInit
```

我宣告了一個 BOOL C 函式，用以處理 Device_Init。然而你可能還記得，device control procedure 應該是以「carry flag 的設立與否」來表示它的成功或失敗。Control_Dispatch 巨集會爲我們在「Boolean 傳回值」和系統要求的「carry flag 設立與否」之間做轉換：

```
   cmp eax, 1
```

如果 EAX（被視爲一個 unsigned 32 位元整數）大於或等於 1，上述指令會清除 carry flag。換句話說，如果 C 函式傳回任何非零值（譯註：回返值會被放在 EAX 之中），control procedure 就會在 return 時將 carry flag 清除，表示成功；如果 C 函式傳回 0，control procedure 就會在 return 時設立 carry flag，表示失敗。

注意　真正的事實是，Control_Dispatch 通常只測試 EAX。以 PNP_NEW_DEVNODE 訊息爲例，它總是在回返前設立 carry flag。這是因爲「在訊息回返時將 carry flag 設立」正是訊息處理常式通知呼叫端「我真正處理了訊息」的一個方式。這會觸發對「處理常式之中你所註冊過的函式」的呼叫。

Device_Init 訊息發生時，ESI 暫存器會指向原本呼叫 WIN.COM 時的命令列的尾部（command-tail）。這在 Windows 95 之中並不是太有用，因爲 Windows 95 處理完 AUTOEXEC.BAT 後會自動呼叫 WIN.COM，因此我沒有在上述範例中傳遞 ESI。我想這樣的彈性卻也導至了一個問題，那就是不同的程式員可能實作出不同的、不完整的訊

息處理常式來。

全域變數（Global Variables）

通常在定義全域變數時，如果其初值爲 0，我就不設初值。舉個例，如果一個全域變數名爲 *counter*，我通常是這麼定義：

```
int counter;   // 噢，不，不要在 VxD 中這麼做
```

但是在一個 C VxD 中，請不要省略初始化動作，因爲沒有誰會幫你設初值爲 0。所以請總是明白地這麼做：

```
int counter = 0;   // 這才是正確的作法
```

你需要一個明顯的初值設定動作，原因是唯有 C 編譯器、聯結器、runtime library 通力合作，才能將你未設初值的變數初始化。編譯器會把未設初值的全域變數放在 block-started-by-symbol（BSS）segment 中，而不幸的是，爲了節省可執行檔在磁碟中的空間，聯結器並不爲 BSS data 產生任何 data records。

Runtime library 提供的標準 startup 函式在呼叫你的 *main* 函式之前，會使用一個 block-store 指令將 BSS segment 零值化。然而當你以 C 語言產生一個 VxD，你的程式並未含入任何標準的 runtime library。結果，沒有任何人爲你執行 BSS segment 的零值化動作！你可以自己寫一個這樣的函式碼，或是在任何需要的地方明白設定初值（譯註：那當然是後者容易得多了）

Runtime Library 函式呼叫

以高階如 C 語言來撰寫程式，好處之一就是你能夠取用大量有用的 library 函式。我相信當我想在 VxD 中呼叫一個外部函式如 *strxfrm*（用以轉換字串）時，這個好處會讓我額手稱慶。不幸的是 runtime library 通常夾帶巨大的包袱，包括錯誤記錄和異常處理所需的碼，而且它通常需要多方面的初始化工作始能有效運作。被含入於大多數 C 程式內的

標準 *crt0* 函式，並沒有機會爲 VxD 服務（對初學者而言，程式該做的第一件事情是呼叫一個 Win32 API 以獲得 command line，然而這是一個對 VxD 而言完全沒有意義的觀念。此外還有許多其他的複雜因素），因此，你根本不該將標準的 runtime library 和 VxD 聯結在一起。

即使不能夠在 VxD 中使用標準的 runtime library，你還是可以使用編譯器提供的任何帶有 **"intrinsic implementation"** 的函式。所謂 "intrinsic" 函式就是編譯器可以爲它產生 inline code 的函式。舉個例子，*memcpy* 就是！編譯器知道如何產生一個 block-move（區塊搬移）指令以完成 *memcpy* 函式呼叫。它甚至知道如果只是對付少量幾個 bytes 的話，該如何產生簡單的 MOV 指令就完成 *memcpy*。當你編譯 release 版程式，通常會選擇最佳化，強迫儘可能使用 intrinsic。甚至當你編譯 debug 版程式（喔，面對 VxD，你常常需要這麼做 ☺），你也可以使用 pragma 句子完成相同願望。例如，在我的 C 語言驅動程式中，我通常包含下面兩個敘述句，以便能夠自由使用字串相關函式和 byte 相關函式：

```
#include <string.h>
#pragma intrinsic(memcmp, memcpy, memset, strcat, strcmp, strcpy, strlen)
```

Callback Wrappers

我必須調整一下炮口，對準以 C 開發 VxD 時的一個比較困難的問題。當你提供一個 callback 函式位址給 VMM 或其他 VxDs 時，會有許多緊張場面出現。所謂 callback 函式就是：當某些 event 發生，會有其他的系統元件來呼叫你 。我將在後面章節再詳細討論 callback 函式，但是我現在要先給一個簡單的例子。

你可以利用 *Set_Global_Time_Out* service 讓某個函式在特定時間週期被呼叫起來。若使用 assembly 語言，你可以這樣設定一秒週期：

```
mov     eax, 1000               ; 1000 miliseconds
mov     edx, refdata            ; "reference" data
mov     esi, offset32 timeout   ; callback 函式的位址
VMMCall Set_Global_Time_Out
```

相當於 C 語言的：

```
Set_Global_Time_Out(timeout, 1000, refdata);
```

時間一到，VMM 就以表 **7-6** 的暫存器呼叫你的 callback 函式。請特別注意此時的 EBP 指向一塊 current VM 暫存器影像（譯註）。也請注意此時的 EDX 內含一些對 callback 函式有意義的參考資料。通常你會使用 EDX 做爲指標，指向一些小量資料，當做參數來傳遞。

> 譯註：所謂暫存器影像（registers image）就是一塊可以記錄 CPU 暫存器值的空間。你知道，Windows 可以跑許多 VMs，每一個 VM 都該有一組暫存器，但真正的 CPU 暫存器只有一組，所以每一個 VM 都有一個 "registers image"，用以儲存 VM 被切換前一刻的暫存器值。事實上，由於 CPU 的排程單元是 thread，所以每一個 thread（而不只是每一個 VM）有自己的一個 "registers image"。

暫存器	內容
EBX	current VM handle
ECX	從「指定之時間間隔」到達之後，一直至今的 milliseconds 數（譯註：指定之時間間隔終了時，CPU 執行權可能沒有辦法立刻切換到 callback 函式來）
EDX	由 *Set_Global_Time_Out* 提供的參考資料
EBP	client（VM）的暫存器影像（registers image）的位址

表 7-6 進入一個 timer callback 函式時，暫存器的內容。

如果以 assembly 語言撰寫 callback 函式，你可以清楚看到暫存器的使用情況。但如果以 C 語言來寫，情況如何？C 編譯器以 EBP 暫存器做爲函式的 stack frame 的指標，並假設 ECX 和 EDX 暫存器是易變（volatile）的，可用於任何用途；並將 EBX 視爲一個工作暫存器，在回返前恢復其原值。似乎沒有什麼明顯而安全的方法，可以在那些暫存器被改變之前，取得其值。

由於這些問題是因暫存器而起，所以你必須以 assembly 寫 callback 函式。有兩個基本作法，第一個是完全以 assembly 來寫，你可以把它放到獨立的 ASM 檔中，也可以在 C 檔案中使用一些 Microsoft C 語言擴充性質，再加上一些 inline assembly，像這樣：

```
void __declspec(naked) timeout()
   {
   _asm
      {
      ...     ; assembly code
      ret     ; return to caller
      }
   }
```

__declspec(naked) 會令編譯器忽略所有正常該有的 prolog 和 epilog 碼（譯註：所謂 prolog 碼和 epilog 碼是編譯器爲每一個函式所加的前置碼和後置碼，通常用來處理堆疊、儲存暫存器，以及恢復暫存器、清理堆疊）。所有的暫存器儲存與恢復，甚至是最終的 RET 指令，都由你自己打理。inline *_asm* 段落中的是一般的 32 位元 assembly 語言，再加上一些額外限制，例如不能使用高權級（privileged）指令（譯註：諸如 `mov eax, cr3` 這樣的 ring0 動作），因爲 Microsoft 編譯器的 inline assembler 不認識它們。

撰寫 callback 函式的第二個作法是提供一個 assembly wrapper，呼叫 C 函式。以前述的 timeout 爲例，你可以這麼做：

```
void __declspec(naked) timeout()
   {
   _asm
      {
      push    ecx         ; 自從 timeout 後的時間
```

```
        push    edx         ; 來自 Set_ call 的參考資料
        push    ebp         ; client register 指標
        push    ebx         ; current VM handle
        call    OnTimeout   ; 呼叫 C 函式
        ret                 ; 回返呼叫端
        }
    }

void __stdcall OnTimeout(PVMMCB hVM, PCRS pRegs, PVOID refdata, DWORD extra)
    {
    ...
    }
```

你可以根據自己的生產力以及對高效率的需求情況，決定使用哪一種作法。第二種作法明顯比較慢，但你卻可以使用更多的 C。

一個完全的 C 範例

和先前的警告一樣，我沒辦法現在就給你一個富有意義的範例。下面是我以 C + DDK 完成的一個基本的 VxD。你可以在書附碟片的 \CHAP07\C-DDK 目錄中找到它。

MYVXD.MAK

```
#0001  # MYVXD.MAK -- MAKE file for sample VxD
#0002
#0003  all: myvxd.vxd
#0004
#0005  devdcl.obj: $*.asm
#0006     ml -coff -DBLD_COFF -DIS_32 -W2 -c -Cx -DMASM6 -Zd -DDEBUG $*.asm
#0007
#0008  myvxd.obj: $*.c
#0009     cl -c -Gs -Zdpl -Od -D_X86_ -YX -W3 -DDEBLEVEL=1 -DBLD_COFF -DDEBUG -DIS_32 $*.c
#0010
#0011  myvxd.vxd: devdcl.obj $*.obj $*.def
#0012     c:\ddk\bin\link @<<
#0013  -machine:i386 -debug:none -pdb:none -def:$*.def -out:$@
#0014  -map:$*.map -vxd vxdwraps.clb
#0015  devdcl.obj myvxd.obj
#0016  <<
```

DEVDCL.ASM

```
#0001  ; DEVDCL.ASM -- Required assembly-language part of C driver
#0002     .386p
#0003     include vmm.inc
#0004     include debug.inc
#0005
#0006  Declare_Virtual_Device MYVXD, 1, 0, MYVXD_control,\
#0007     Undefined_Device_ID, Undefined_Init_Order
#0008
#0009  Begin_Control_Dispatch MYVXD
#0010     Control_Dispatch Device_Init, OnDeviceInit, sCall, <ebx, edx>
#0011  End_Control_Dispatch   MYVXD
#0012
#0013     end
#0014
```

MYVXD.C

```
#0001  // MYVXD.C -- C-language skeleton using DDK tools
#0002  #define WANTVXDWRAPS
#0003
#0004  #include <basedef.h>
#0005  #include <vmm.h>
#0006  #include <debug.h>
#0007  #include <vxdwraps.h>
#0008
#0009  #pragma VxD_ICODE_SEG
#0010  #pragma VxD_IDATA_SEG
#0011
#0012  BOOL _stdcall OnDeviceInit(PVMMCB hVM, DWORD refdata)
#0013     {                          // OnDeviceInit
#0014     return TRUE;
#0015     }                          // OnDeviceInit
```

使用 C++ 搭配 Windows 95 DDK

我曾經長期以爲，以 C++ 撰寫 VxD 必定有著非常不合理的困難度。畢竟第一個印象是，似乎需要一個外部的 runtime library，才能讓 C++ 程式正常運作。是呀，你當然會選擇 C++，如果你要建立的是一個 MFC 應用程式的話。

後來我發現，寫一個 C++ VxD 其實需要的不多，也不過就是一般的 C++ 編譯器（像是 Visual C++ 4.0）再加上 DDK 表頭檔就夠了。我在第 14 章談到如何建立一個 VCOMM port driver 時，會呈現給你一個周延的 C++ 驅動程式範例。除了和 C 語言的情況一樣，另還有兩個問題需要解決：(1) 如何處理 static 變數的初始化動作，(2) 如何使用 *new* operator。此外你還需要將 DDK 表頭檔以 *extern "C"* 括起來，以避免在產生 VxD service wrapper 函式時有奇怪的修飾名稱（譯註：C++ 編譯器會自動爲函式名稱加上許多修飾，稱爲 name mangling）。

警告 Visual C++ 4.1 有一個 bug，所以不能夠用來製作 VxDs。當其 inline assembler 參考 enumerated constants 時，會產生不正確的碼。這個 bug 會造成每個 VxDCall 巨集（或其變形）產生錯誤的碼而導至當機！

管理「static 初始器」（Handling Static Initializers）

如果你要寫出相當程度的 C++ 驅動程式，你真的得好好管理「static 初始器」。所謂「static 初始器」是指一段用來初始化 static 或 global 變數的碼，通常其中會呼叫 class 的 static instance（靜態實體）的 constructor（建構式）。舉個簡單的例子，假設你的驅動程式使用 *CPort* object，宣告如下：

```
class CPort
    {
public:
    CPort(DWORD address);
    CPort*       m_next;
    DWORD        m_address;
    static CPort* First;
    };
```

再進一步假設你的驅動程式要求在 static 記憶體中建立一個 *CPort* objects 的串列（linked list）。你大概會寫出這樣的程式碼：

```
CPort::First = NULL;

CPort::CPort(DWORD address)
   {                    // CPort::CPort
   m_next    = First;
   First     = this;
   m_address = address;
   }                    // CPort::CPort

CPort(0x3F0);
CPort(0x3F1);
[etc.]
```

（這是一個特別設計的例子。請絕對不要在 Windows 95 驅動程式中硬寫一個 I/O port 位址；你應該和 Plug and Play 架構並存，動態測知你正在使用哪一個 port。不過這個特別設計的例子可以告訴你如何實作出一個「static 初始器」，比使用 floppy disk 驅動程式爲例好，所以忍受一下吧！）

正常情況下，runtime library 初始化程式碼會爲你找出並執行「static 初始器」。它之所以能夠找到它們是因爲編譯器把它們放進一個特別的 segment 之中（.crt$xcu）。由於你的 VxD 並未和一個「知道如何執行 static 初始器」的 startup 碼聯結在一起，所以你得自己提供一份相同價值的東西。因著一些無聊的技術上的理由，你必須先說服編譯器將「static 初始器」放進另一個 segment 之中，並且不論是 COFF 或 OMF（譯註）對此 segment 的命名都必須相同。因此你必須：

> 譯註：COFF 和 OMF 都是模組檔（.OBJ）的格式種類。COFF 的全名是 Common Object File Format，原是 Unix 的標準，後爲 Microsoft 採用並修改，應用於 Win32 環境中。OMF 的全名是 Object Module Format，由 Intel 主導，主要採用廠商有 Borland。

```
#pragma warning(disable:4075)  // nonstandard init seg
#pragma init_seg("INITCODE")
#pragma warning(default:4075)
```

譯註：類似上述的 #pragma 程式技巧，在 Jeffrey Richter 的 ***Advanced Windows*** 一書也不時出現。該書第二版的附錄 B 和第三版的附錄 C 都有相關說明。

採用 "INITCODE" 名稱並沒有什麼特別原因，只要和「負責初始化動作」的那個 assembly 模組中所使用的名稱相同即可。

接下來，你應該在內含 device control procedure 的 assembly 程式中加入：

```
VxD_LOCKED_DATA_SEG
didinit  dd    0
VxD_LOCKED_DATA_ENDS

initcode segment dword public flat 'code'
beginit  dd    0
initcode ends
...
initend segment dword public flat 'code'
endinit  dd    0
initend ends

_bss    segment dword public flat 'lcode'
startbss dd    0
_bss ends
...
_ebss   segment dword public flat 'lcode'
endbss   dd    0
_ebss ends
```

（我以 *_BSS* segment 代表「static 初始器」勢力範圍之外的那些未被初始化的全域變數）

你還必須在你的 device control procedure 中插入一些負責初始化動作的碼：

```
Begin_Control_Dispatch MYVXD

       bts     didinit, 0                ; been here before ?
       jc      skipinit                  ; if yes, skip init
       pushad                            ; save all registers
       mov     esi, offset32 beginit+4   ; point to first entry
@@:
       cmp     esi, offset32 endinit     ; reached end of list ?
       jae     @F                        ; if yes, leave the loop
       call    dword ptr [esi]           ; call init function
       add     esi, 4                    ; process all of them
       jmp     @B                        ; ...
@@:
       cld
       mov     edi, offset32 startbss    ; point to start of BSS
       mov     ecx, offset32 endbss      ; compute length
       sub     ecx, edi                  ; ...
       shr     ecx, 2                    ; convert to DWORDs
       xor     eax, eax                  ; get const zero
       rep     stosd                     ; zero-fill BSS area

       popad                             ; restore registers

skipinit:

Control_Dispatch ...
[etc.]
End_Control_Dispatch MYVXD
```

這些新加的碼一開始先測試一個旗標位元，看看我們是否已經做過了額外的初始化動作（BTS 指令用來測試一個位元。它會設立此位元，而如果該位元原先就已設立，那麼 BTS 會將 carry flag 也設立）。*initcode* segment 中內含一些指標，指向所有需要呼叫的初始化函式。這些 segments directives 的安排保證 *beginit* 和 *endinit* 符號會整齊地排列 *initcode* 指標。我使用一個類似的計倆來找出 _BSS 的開始和結束。（順帶一提，沒有必要非得在額外的那些 segments 中定義資料物件不可！但是當我企圖定位 *initcode* 和 BSS 的開始和結束而沒有那麼做時，我遭遇了聯結器當掉的問題）

最後，你還得加上一些 segment 定義，放在 VxD 的 .DEF 檔中，以便聯結器能夠將所有的 segments 適當群組起來：

```
VXD MYVXD

SEGMENT
       ...
       _EBSS    CLASS 'LCODE'   PRELOAD NONDISCARDABLE
       INITCODE CLASS 'ICODE'   DISCARDABLE
       INITEND  CLASS 'ICODE'   DISCARDABLE
       ...
```

改寫 new 和 delete

標準 runtime library 用以實現 *new* 和 *delete* operator 所需的一個 heap management subsystem，沒有辦法（也不應該）在一個 VxD 中存在。如果你願意慎重使用這些 operators，那麼你可以簡單地利用 VMM heap manager 自行設計一些函式來完成它們：

```
void* ::operator new(unsigned int size)
   {                    // operator new
   return _HeapAllocate(size, 0);
   }                    // operator new

void ::operator delete(void* p)
   {                    // operator delete
   if (p)
      _HeapFree(p, 0);
   }                    // operator delete
```

但是請小心，不要到處配置和釋放 objects。不只是因爲你可能需要承擔「heap 被弄得支離破碎」的風險，而且你可能在不安全的情況下呼叫 heap manager。舉個例子，當你正在處理一個硬體中斷，不經意地（隱含地）呼叫 *_HeapAllocate* 是非常不好的事情。是的，你很可能因爲下面的碼而不經意（隱含地）呼叫了 *_HeapAllocate*：

```
CPort CPort::GetPort(DWORD address)
   {                          // CPort::GetPort
   return CPort(address);     // oops! calls new CPort
   }                          // CPort::GetPort
```

VToolsD

如果說以 Microsoft DDK 撰寫 C VxD，就像把 11 號的腳塞進 9 號鞋子裡的話，那麼 VToolsD 可說是一隻合腳的鞋了，特別是如果你用它的 class library 來開發 C++ 驅動程式的話。VToolsD 是兩位 DOS extender 開發者的獨到觀念，他們創立了一家名爲 Vireo Software 的小公司，實現他們的想法，將 VxD 程式設計推廣給更多程式員。VToolsD 內含三個基本元件：

- QuickVxd - 它會向使用者詢問 VxD 規格，並產生出一個骨幹程式，然後你可以爲它長肉。它很類似 Microsoft 的 AppWizard，不過沒那麼大的雄心，因此留下較多的細節給你完成。
- 表頭檔以及 libraries - 允許你完全以 C 來開發 VxD，不需要任何 assembly 語言，除非偶爾用到的一些 inline assembly 碼。
- 表頭檔和 C++ class library - 允許你以 C++ 開發 VxD。

Vireo 可謂是面對哥利亞巨人（我當然是指 Microsoft）的少年大衛。你也可以說他們穿著優雅的露趾鞋而不是笨重的橡膠鞋跳舞。對於 VxD 開發人員而言，Vireo 帶來一個重要的好處：他們的確支援並回應他們的所有客戶。就我的經驗，我的確得到了諮詢服務以及除錯報告。另一個好處是，你終於可以在 Microsoft 之外有其他選擇。

從另一個角度看，Vireo 公司的規模使它沒辦法把 VToolsD 帶到一個更優雅的境界。我曾經遇到過令人懊惱的問題，像是發現某些 derived class 中我所亟需的一些 class members 竟然被宣告爲 private！不過我要提醒你，DDK 也絕對不可能和優雅扯上一點邊。

使用 QuickVxD

我只能在本書中給一點點篇幅來解釋 VToolsD 的用法，讓你嚐一點它的滋味。我們將從 QuickVxd 開始（**圖 7-3**）。對話盒中的第一個活頁（tab）允許你指定驅動程式的基本資料，像是名稱、ID、初始化次序等等。

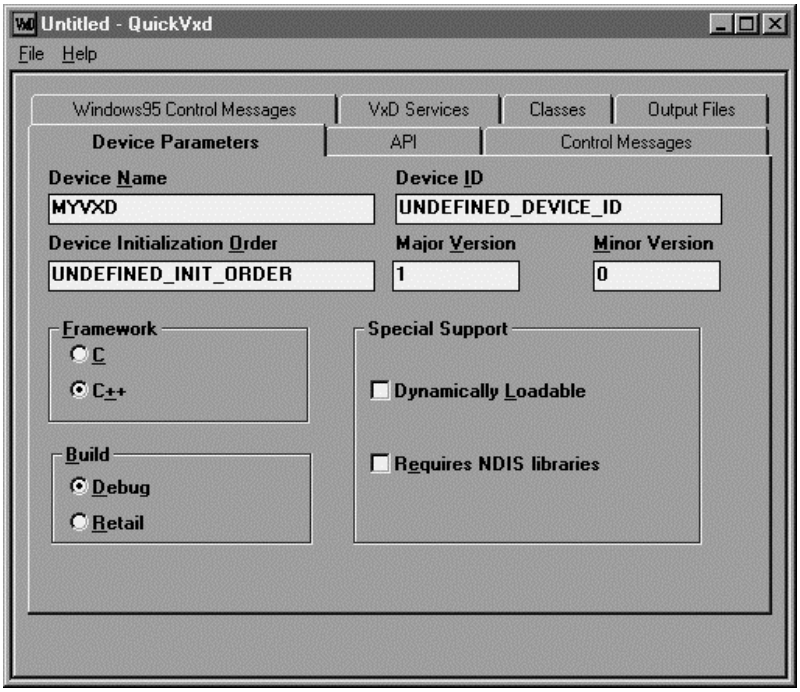


圖 7-3 QuickVxd 用來定義一個 VxD 的規格

當你進入另兩個 "Control Messages" 活頁（圖 **7-4** 和圖 **7-5**），QuickVxd 會立刻顯露出它優於 DDK 的部份。這兩個活頁都讓你以圈選的方式決定驅動程式要處理哪些 system control messages。QuickVxd 也會做少量的常識判斷。舉個例，如果你說你要產生一個可動態載入的驅動程式，但是卻沒有選擇 Sys_Dynamic_Device_Init 和 Sys_Dynamic_Device_Exit（它們是任何動態驅動程式都必須處理的訊息），那麼 QuickVxd 會給你一個警告。

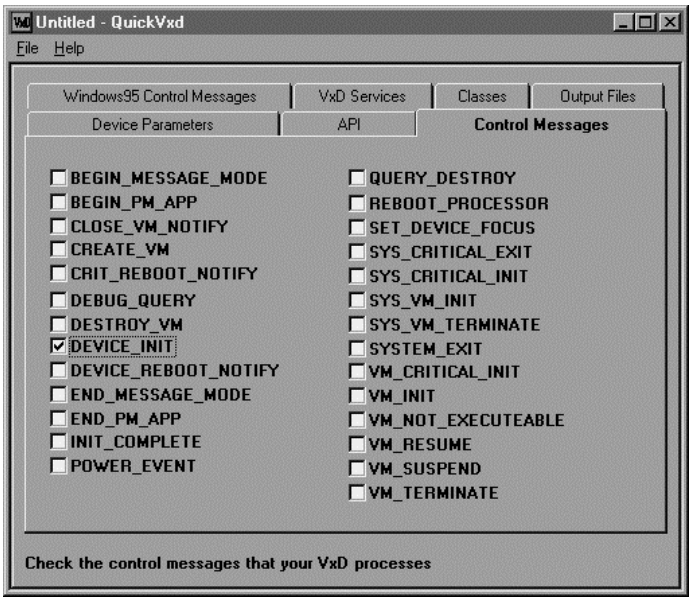


圖 7-4　QuickVxd 的 "Control Message" tab

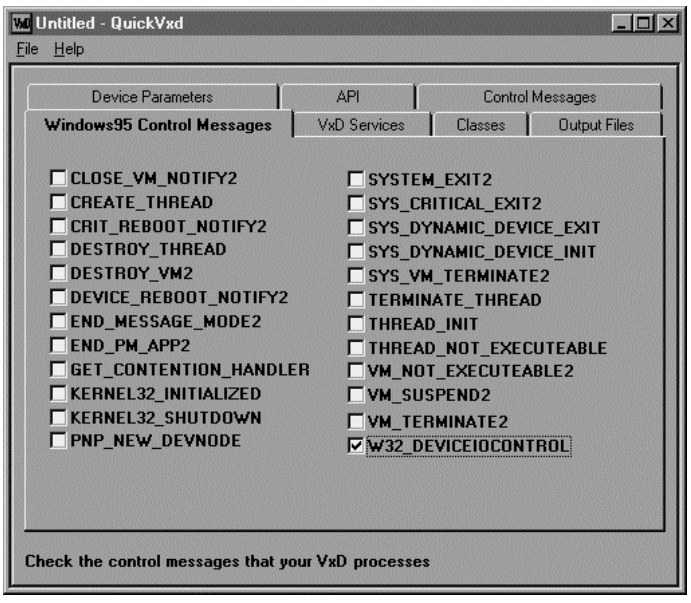


圖 7-5　QuickVxd 的 "Windows 95 Control Message" tab

QuickVxd 所產生的 C 骨幹程式

QuickVxd 會爲你產生出一個 MAKE 檔，一個表頭檔，和一個 C 原始碼檔案（如果你指定的是 C framework）。以下這些檔案放在書附碟片的 \CHAP07\C-VTOOLSD 目錄中：

MYVXD.MAK

```
#0001  # MYVXD.mak - makefile for VxD MYVXD
#0002
#0003  DEVICENAME = MYVXD
#0004  FRAMEWORK = C
#0005  DEBUG = 1
#0006  OBJECTS = myvxd.OBJ
#0007
#0008  !include $(VTOOLSD)\include\vtoolsd.mak
#0009  !include $(VTOOLSD)\include\vxdtarg.mak
#0010
#0011  myvxd.OBJ:      myvxd.c myvxd.h
```

MYVXD.H

```
#0001  // MYVXD.h - include file for VxD MYVXD
#0002
#0003  #include <vtoolsc.h>
#0004
#0005  #define MYVXD_Major            1
#0006  #define MYVXD_Minor            0
#0007  #define MYVXD_DeviceID         UNDEFINED_DEVICE_ID
#0008  #define MYVXD_Init_Order UNDEFINED_INIT_ORDER
```

MYVXD.C

```
#0001  // MYVXD.c - main module for VxD MYVXD
#0002
#0003  #define   DEVICE_MAIN
#0004  #include  "myvxd.h"
#0005  #undef    DEVICE_MAIN
#0006
#0007  Declare_Virtual_Device(MYVXD)
#0008
```

```
#0009  DefineControlHandler(DEVICE_INIT, OnDeviceInit);
#0010  DefineControlHandler(W32_DEVICEIOCONTROL, OnW32Deviceiocontrol);
#0011
#0012  BOOL __cdecl ControlDispatcher(
#0013   DWORD dwControlMessage,
#0014   DWORD EBX,
#0015   DWORD EDX,
#0016   DWORD ESI,
#0017   DWORD EDI,
#0018   DWORD ECX)
#0019  {
#0020   START_CONTROL_DISPATCH
#0021
#0022          ON_DEVICE_INIT(OnDeviceInit);
#0023          ON_W32_DEVICEIOCONTROL(OnW32Deviceiocontrol);
#0024
#0025   END_CONTROL_DISPATCH
#0026
#0027   return TRUE;
#0028  }
#0029
#0030
#0031  BOOL OnDeviceInit(VMHANDLE hVM, PCHAR CommandTail)
#0032  {
#0033   return TRUE;
#0034  }
#0035
#0036  DWORD OnW32Deviceiocontrol(PIOCTLPARAMS p)
#0037  {
#0038   return 0;
#0039  }
```

從這個小小範例之中我們可以看出，QuickVxd 和 VToolsD framework 爲你完成 DDB 的定義以及 device control procedure 的一切細節，不需你寫任何 assembly 碼或知道 system control messages 發生時暫存器內容的意義。爲了從這些建造好的碼中獲得最高利益，你必須在專案一開始就完整指定你的 VxD 介面，如此可儘量避免稍後爲 *ControlDispatcher* 函式加上 control messages 時任何可能的困難。

這個例子也顯示出你可能會在面臨 VToolsD 時遭遇的兩個潛在問題。第一個問題就是 DDK 和 VToolsD 之間不同的資料結構名稱。例如 VToolsD 使用 typedef *VMHANDLE* 來描述一個指向 VM control block 的指標，而 DDK 的 VXDWRAPS.H 則使用

PVMMCB 來描述同一種東西；Vireo 稱呼 *OnW32DeviceIoControl* 的參數結構爲 *IOCTLPARAMS*，而 DDK 則稱之爲 *DIOCPARAMETERS*；此外不同的工具對於結構中的成員亦有各自的命名。因此我們很難將自己的程式碼從一個工具平台移植到另一個工具平台。

第二個問題是，Vireo framework 會因爲 Device_Init 這個 system control message 而呼叫 *OnDeviceInit* 函式，然而如我稍早所說，這個訊息發生時 EDX 暫存器指向一塊由真實模式初始化函式所提供的參考資料。稍早的例子中我放棄命令列尾端資訊（command-tail）而改以參考資料做爲參數，不幸的是要把 Vireo framework 整修成相同的樣子，不是件容易的事！

注意 「EDX 暫存器內含真實模式初始器之參考資料」這一事實在技術文件中只明載是針對 Sys_Critical_Init，並不針對 Device_Init 或 Init_Complete；後者是驅動程式的另兩個初始化訊息，或許參考資料在那時候也有用！這是 Microsoft VxD 技術文件撰寫者的失察。你可以在 DDB 的 *DDB_Reference_Data* 欄位說明中獲得稍微好一點點的介紹。在 VToolsD 驅動程式中，你可以使用外部名稱 *The_DDB* 來參考（代表）DDB。

VToolsD Framework 中的 callback 函式

VTOOLSD 使你生活更輕鬆的另一種情況是：callback 函式的使用。稍早我曾顯示給你看過，如何以 DDK 的 C 語言介面外包一個 timer event callback 函式。VToolsD 更簡單，因爲它讓你定義一個 callback thunk，然後負責在其中填寫必要的機器語言，令它呼叫你的 C callback 函式。你的碼看起來像這樣：

```
static TIMEOUT_THUNK thunk;      // in locked data
TIMEOUT_HANDLER OnTimeout;       // forward declaration
...
Set_Global_Time_Out(1000, refdata, OnTimeout, &thunk);
...
void __stdcall Ontimeout(VMHANDLE hVM, PCLIENT_STRUCT pRegs,
   PVOID refdata, DWORD extra)
{                                // OnTimeout
...
}                                // OnTimeout
```

許多程式員發現，這種 thunk-based 作法比使用 *__declspec(naked)* 更簡單，不過這個領域是 Vireo 技術文件和其表頭檔不一致的地方（至少前一版是如此）。C++ framework 的方法更簡單，稍後我爲你介紹。

Segments 和 Wrappers

你不需指定含入任何檔案，就可以呼叫 VxD services；VToolsD 會自動爲你宣告。雖然實際情況和 VToolsD 的報導還有一些距離，不過 VToolsD 在這一點的確遠比 DDK 優異得多。

就像 DDK 一樣，你可以這樣輕鬆控制你的 segmentation：

```
#include LOCKED_CODE_SEGMENT
#include LOCKED_DATA_SEGMENT
```

這些常數會表現出表頭檔的名稱，用以完成 segment 名稱的必要改變。

VToolsD 不只包含所有 VxD services 的 wrappers，它還包含每個 wrapper 的五個版本，每個版本係針對你可能的呼叫地點（某個 segment）而設計，因而有所謂的 *INIT_Get_VMM_Version*、*LOCK_Get_VMM_Version*...等等等。當你建立一個特別的 code segment，屬於...唔...例如 *LOCKED_CODE_SEGMENT*，你也就編譯了一些 #define 句子，於是 *Get_VMM_Version* 就變成了 *LOCK_Get_VMM_Version*。VToolsD 的 runtime library 在五個可能的名稱之下都有 wrappers，每一個 wrapper 內含相同的碼，引發正確的 INT 20h。

含入多個 wrappers，意味你的驅動程式可能最終含有每一 wrapper 的多個副本，每個副本用來對付一個 segment（你可能在其中做呼叫動作）。如果對你而言空間很重要，不能這麼浪費，那麼就在必要的 segment 中使用 wrapper 全名，例如：

```
#include INIT_CODE_SEGMENT
...
DWORD version = LOCK_Get_VMM_Version();
...
#include LOCKED_CODE_SEGMENT
...
DWORD version = LOCK_Get_VMM_Version();
```

但是請小心，你不應該在 locked code segment 中呼叫 *INIT* 或 *PAGEABLE* 兩種 wrapperr。

Runtime Library Calls

VToolsD 和 DDK 之間的另一個重要差別是，VToolsD 允許你使用更多的 runtime library 函式。兩者都支援那些有 "compiler intrinsics"（也就是編譯器能夠產生 inline 碼以實作出函式內容的那種）函式。例如 *strcmp* 和 *memcpy* 對於這兩個工具都是可用的，但 VToolsD 還允許你使用 nonintrinsic 函式如 *atol*、*sprintf*、*malloc* 等。

QuickVxd 產生的 C++ 骨幹程式

當你使用 C++ framework，QuickVxd 爲你產生以下的程式骨幹：

MYVXD.MAK

```
#0001  # MYVXD.mak - makefile for VxD MYVXD
#0002
#0003  DEVICENAME = MYVXD
#0004  FRAMEWORK = CPP
#0005  DEBUG = 1
#0006  OBJECTS = MYVXD.OBJ
#0007
#0008  !include $(VTOOLSD)\include\vtoolsd.mak
#0009  !include $(VTOOLSD)\include\vxdtarg.mak
#0010
#0011  MYVXD.OBJ:     MYVXD.cpp MYVXD.h
```

MYVXD.H

```
#0001  // MYVXD.h - include file for VxD MYVXD
#0002
#0003  #include <vtoolscp.h>
#0004
#0005  #define DEVICE_CLASS          MyvxdDevice
#0006  #define MYVXD_DeviceID         UNDEFINED_DEVICE_ID
#0007  #define MYVXD_Init_Order       UNDEFINED_INIT_ORDER
#0008  #define MYVXD_Major           1
#0009  #define MYVXD_Minor           0
#0010
#0011  class MyvxdDevice : public VDevice
#0012  {
#0013  public:
#0014     virtual BOOL  OnDeviceInit(VMHANDLE hSysVM, PCHAR pszCmdTail);
#0015     virtual DWORD OnW32DeviceIoControl(PIOCTLPARAMS pDIOCParams);
#0016  };
#0017
#0018  class MyvxdVM : public VVirtualMachine
#0019  {
#0020  public:
#0021        MyvxdVM(VMHANDLE hVM);
#0022  };
#0023
#0024  class MyvxdThread : public VThread
#0025  {
#0026  public:
#0027        MyvxdThread(THREADHANDLE hThread);
#0028  };
```

MYVXD.CPP

```
#0001  // MYVXD.cpp - main module for VxD MYVXD
#0002
#0003  #define DEVICE_MAIN
#0004  #include "myvxd.h"
#0005  Declare_Virtual_Device(MYVXD)
#0006  #undef DEVICE_MAIN
#0007
#0008  MyvxdVM::MyvxdVM(VMHANDLE hVM) : VVirtualMachine(hVM) {}
#0009
#0010  MyvxdThread::MyvxdThread(THREADHANDLE hThread) : VThread(hThread) {}
#0011
```

```
#0012  BOOL MyvxdDevice::OnDeviceInit(VMHANDLE hSysVM, PCHAR pszCmdTail)
#0013  {
#0014      return TRUE;
#0015  }
#0016
#0017  DWORD MyvxdDevice::OnW32DeviceIoControl(PIOCTLPARAMS pDIOCParams);
#0018  {
#0019      return 0;
#0020  }
```

- ***MyvxdDevice*** 衍生自基礎類別 *VDevice*，用以實作出你的 virtual device driver。此類別有許多對應於 system control messages 的成員函式，因此你只要改寫基礎類別的成員函式如 *OnDeviceInit* 或 *OnW32DeviceIoControl*，就可以處理特定訊息。
- ***MyvxdVM*** 衍生自 *VVirtualMachine*，用以實作出你需要的任何與 VM 有關的特定處理。VToolsD framework 將大部份與 VM 有關的 system control messages 都繞行（routes）到這個類別來，唯一例外是 Create_VM 訊息，它由你的 *MyvxdDevice* 類別處理，爲的是要先產生出一個 *MyvxdVM* 物件。
- ***MyvxdThread*** 衍生自 *VThread*，用以處理你的 VxD 中任何與 thread 有關的訊息。和 *MyvxdVM* 的情況一樣，你的 *MyvxdThread* 處理所有與 thread 有關的 system control messages，唯一一個例外就是 Create_Thread，它由你的 *MyvxdDevice* 處理，爲的是要先產生出一個 *MyvxdThread* 物件。

許多時候你可能只需要 *VDevice* 衍生類別，另兩種類別並不是在每一個 VxD 中都用得上。當然啦，即使你從未產生後兩種類別的實體，它們的出現並不會帶給你什麼傷害。

C++ 的 callback 函式

VToolsD 對於 callbacks 體制是以 framework 中的其他基礎類別爲中心。你可以從 *VGlobalTimeOut* 衍生出自己的類別，用以解決我們先前提過的 timeout 問題：

```
class MyTimeOut : public VGlobalTimeOut
    {
public:
    MyTimeOut(DWORD msec) : VGlobalTimeOut(msec) {}
    virtual VOID handler(VMHANDLE, PVOID, CLIENT_STRUCT*, DWORD);
    };
```

爲了能夠設定 timeout，你應該寫這一行碼：

```
(new MyTimeOut(1000))->Set();
```

爲了能夠在 timeout 終於發生時處理之，你應該這樣子寫 callback 函式：

```
VOID MyTimeOut::handler(VMHANDLE hVM, PVOID junk,
    CLIENT_STRUCT* pRegs, DWORD extra)
    {                // MyTimeOut::handler
    ...
    }                // MyTimeOut::handler
```

第 8 章

Virtual Device Drivers 的起始與終結

Windows 95 根據 virtual device drivers（VxDs）被載入的時間，將之區分爲靜態和動態兩種。VMM 在 Windows 95 啓動的時候載入 static VxDs，並在此 Windows 95 session 生命期間，將它們一直保留在虛擬記憶體中。至於 dynamic VxDs，是應某個程式或某個 VxD 的要求，動態地（不然還能怎麼樣）被載入。在一個 Windows 95 session 生命期間，dynamic VxDs 可以反複多次地被卸載（unloaded）再被重新載入（reloaded）。一般規則是：如果 VxDs 係用來搭配某個硬體裝置或某個應用程式，通常會被設計爲動態；如果它是 VMM 的核心元件（包括 VMM 自己），則會被設計爲靜態。

static VxD 的生命週期可以概述如下。VMM 從 registry database 或從 SYSTEM.INI 檔中找出哪些 static VxDs 需要被載入。真實模式常駐程式，像是 TSR utility 或 MS-DOS 驅動程式，也可以 "hook" 一個中斷以要求 VMM 載入某個特定的 static VxDs。Static VxDs 會在系統啓動時刻收到三個訊息：Sys_Critical_Init, Device_Init 和 Init_Complete，分別對應 VMM 的三個重要階段。Static VxDs 可以內含真實模式初始化程式碼，在 VMM 切換至保護模式之前獲得執行權。當 VMM 準備終結 Windows 95，回到真實模式之前，

static VxDs 會收到一系列的停工（shutdown）訊息。

至於 dynamic VxDs，VMM 倒是無法自己發現。應用程式或 VxD（例如 Confiucatin Manager 或 Input/Output Supervisor）可以要求載入一或多個 VxDs，並由 VXDLDR（一個 static VxD）將它們載入。和 static VxDs 不同的是，dynamic VxDs 只會收到一個初始訊息和一個停工（shutdown）訊息。除了這些不同，兩種 VxDs 使用相同的工具和相同的 APIs 來開發。事實上在 static VxDs 和 dynamic VxDs 之間存在的唯一技術性差異是：dynamic VxD 必須在 .DEF 檔中宣稱它自己是 *dynamic*，而 static VxDs 不必。

靜態的（static）VxDs

有三種方法可以引發 VMM 載入一個 static VxD。大部份時候，你或你的安裝程式會修改 registry database 或是位於 Windows 95 磁碟目錄中的 SYSTEM.INI。然而某些情況下你可能會有一些 TSR（常駐程式）沒辦法在缺少某個 VxD 的情況下發揮功能；這時候你的常駐程式必須 "hook" 軟體中斷 2Fh，尋找其 1605h 子功能（"startup broadcast"）。

使用 System Registry

在 Windows 95 中較受歡迎的 static VxD 載入法，是在 system registry database 中加上一筆記錄：

```
HKEY_LOCAL_MACHINE
    System
        CurrentControlSet
            Services
                VxD
                    key
                        StaticVxD=pathname
```

在這份 registry 樹狀架構的局部內容之中，*key* 是個 registry key，*pathname* 是將被靜態載入的 VxD 名稱。VMM 會將 VxD key 的所有 subkeys 列舉出來，尋找 *StaticVxD* 這個 values；每一個這樣的 value 表示一個要被靜態載入的 VxD。

舉個例子，Configuration Manager 擁有圖 **8-1** 的 registry entry。圖中 "CONFIGMG" device 之前的 * 符號代表這個驅動程式實際位於 VMM32.VXD 之內，而不是在另外某個檔案中。你當然不會對你自己的 VxDs 使用這種命名方式！圖中各式各樣的 named values，像是 *Detect*，可以示範 registry 如何能夠包含「與 device 相關的參數」。稍後我會告訴你如何查詢這些參數（例如 *Detect*）以便在你的驅動程式初始化時加以組態（configure）。

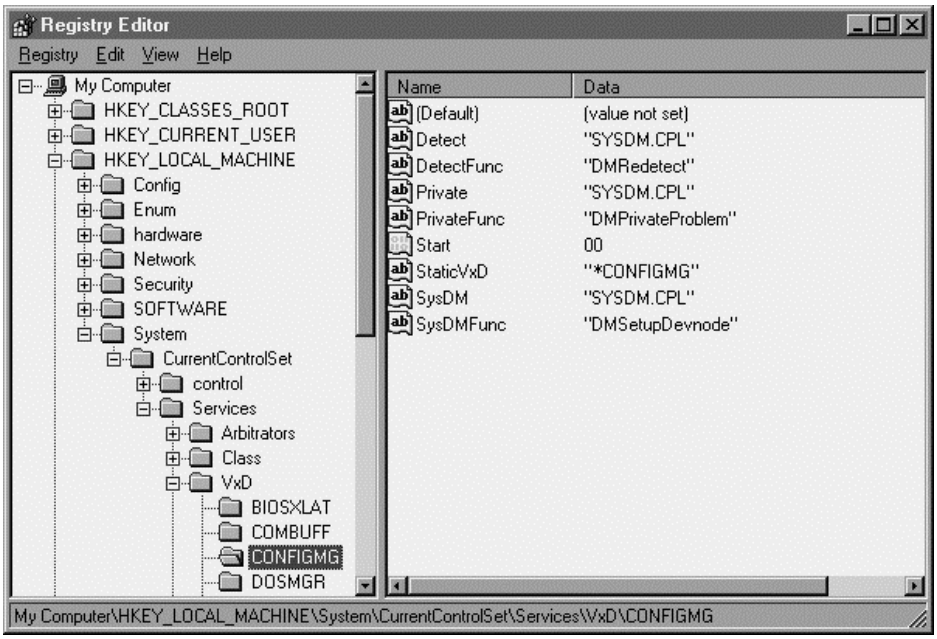


圖 8-1　CONFIGMG device 的 registry entries。

使用 SYSTEM.INI

在 Windows 3.1 或更早之前，SYSTEM.INI 中的 [386enh] 區段可以控制 Windows 的許多低階系統功能。你可以像下面這樣，引導 Windows 載入 VxDs：

```
device=pathname
```

假設你的 VxD 名爲 MYVXD.VXD，被安裝在 C 磁碟的 MYAPP 目錄中，你的

SYSTEM.INI 中應該有這樣的句子：

```
device=c:\MYAPP\MYVXD.VXD
```

現在你還是可以在 Windows 95 中使用 SYSTEM.INI，但是像前一節那樣修改 registry 會比較理想。儘管如此，我卻發現在發展 static VxD 期間，改變 SYSTEM.INI 檔是比較簡單的方法。SYSTEM.INI 有一個更實用性的理由：一個不良的 VxD 可能會導至 Windows 95 無法啓動，而你卻只有在一個運作中的 Windows 95 session 中才能夠修改 registry。至於 SYSTEM.INI，噢，你永遠能夠在 Windows 95 啓動之前，於真實模式中編輯 SYSTEM.INI。

啟動進入（Booting into）MS-DOS

當你打開你的電腦，Windows 95 會進入圖形式 Windows 環境。或許你已經知道，至少有一種方法可以讓這個 boot 程序短路，進入 MS-DOS。在 boot 過程中按下 F8，你會獲得一份選單，你可以選擇文字命令列模式代替圖形式 Windows 環境。在 MS-DOS 提示號下你可以對一個真實模式的網路加以組態（configure）、安裝 DDK 除錯版或零售版、或是啓動一個如 SoftIce/W 那樣的除錯器。當然，你也可以發出一個 WIN 命令，進入 Windows 95 圖形環境。

我發現我幾乎總是得在進入 Windows 95 之前先進入 MS-DOS，所以我學了幾招，讓 boot 程序在 MS-DOS 階段暫停下來。我們知道，Windows 95 之所以獲得控制權是因爲 MS-DOS 在處理完 AUTOEXEC.BAT 之後，發出了一個 WIN 命令。如果你有一個 WIN.BAT，能夠在 path 搜尋路徑中比 WIN.COM(位於 Windows 子目錄)更早被發現，你的 WIN.BAT 就能夠在 Windows 95 執行之前先被執行。

一個比較精緻的作法是修改 MSDOS.SYS。怪了，MSDOS.SYS 不是 MS-DOS 的兩個系統隱藏檔之一（另一爲 IO.SYS）嗎？早期版本中 MSDOS.SYS 的確是一個巨大的二進位檔，內含一些你別想改變的程式碼（唔，或許有少數讀者曾經修改過它，不過絕大部份的讀者不會嘗試這麼做）。然而在 Windows 95 版的 MS-DOS 中，這只是一個短短的

ASCII 文字檔，放在你的啓動磁碟（boot disk）的根目錄中。它內含 Windows 95 的 startup options，格式大約和 Windows 的 .INI 檔相同。其中的 BootGUI option 控制著 MS-DOS 是否自動發出 WIN 命令。你可以這樣子改變它：

```
[Options] BootGUI=0
```

爲了完成這項編輯工作，你必須先將 MSDOS.SYS 的三個檔案屬性：隱藏、唯讀、系統（Hidden、Read-Only、System）去除。編輯完畢後，你可以恢復檔案原屬性，也可以不恢復，因爲不管 Windows 95 或 MS-DOS 都不在乎那些屬性是否成立。

順帶一提，如果你使用 dual boot（雙重開機特性，由 MSDOS.SYS 檔的 [Options] BootMulti=1 控制）來啓動一個較早版本的 MS-DOS，請不要嘗試編修其 MSDOS.SYS 檔，因爲此時 MSDOS.SYS 真的內含一部份 MS-DOS 碼；Windows 95 會把某些檔案重新命名，而你想編修的那個檔案現在變成了 MSDOS.W40。

譯註：我曾經收到讀者來函詢問：「爲何洋人把電腦開機動作叫做 "boot"？ 可否請侯先生從英文字義或字源的觀點，或者有什麼典故，爲我解釋一下這個長久以來在我心中的疑問」。

根據我的瞭解，boot 程式必須非常小，但又必須足夠 smart 以讀取稍稍複雜的另一個程式；該程式又必須足夠 smart 以從磁碟或磁帶中讀取作業系統。這樣「一個拉拔一個」，連續不斷的程序就像拔靴帶一樣，所以被叫做 "bootstrap loader"。後來就慢慢演變爲 "boot"。

使用 INT 2Fh, Function 1605h（Startup Broadcast）

Windows 95 之中還是有許多真實模式應用軟體 -- 雖然它們的重要性日漸衰減。爲了回溯相容（如果沒有其他原因了的話）， Windows 95 繼續支援數種介面，允許真實模式驅動程式與 TSR 常駐程式能夠與之共存。INT 2Fh / function 1605h 是其中最重要的一個介面。

VMM 會在系統啓動時刻發出軟體中斷 2Fh，並將暫存器 AX 設定爲 1605h。常駐軟體可以攔截（hook）軟體中斷 2Fh 以接收廣播（broadcast）訊息。你的處理常式會和其他的 2Fh 中斷處理常式串鏈(chain)在一起，並傳回 ES:BX，指向一串由 startup information structures 所組成的串列。程式碼大約是這個樣子：

```
sis       db     3, 0          ; SIS_Version
          dd     0             ; SIS_Next_Dev_Ptr
          dd     vxdname       ; SIS_Virt_Dev_File_Ptr
          dd     refdata       ; SIS_Reference_Data
          dd     0             ; SIS_Instance_Data_Ptr
vxdname   db     'pathname', 0

          assume ds:nothing, cs:@curseg

          align  4
org2f     dd     ?
int2f:    cmp    ax, 1605h
          je     @F
          jmp    [org2f]
@@:       pushf
          call   [org2f]
          mov    word ptr sis+2, bx
          mov    word ptr sis+4, es
          mov    bx, cs
          mov    es, bx
          mov    bx, offset sis
          iret
```

在這段程式碼中，*pathname* 代表 VxD 的名稱，*refdata* 代表被傳遞給「VxD 之真實模式初始化函式」的一筆 32 位元資料。

通常你會爲你的驅動程式指定路徑全名。假設你的驅動程式名爲 MYVXD.VXD，放在 C 碟的 MYVXD 子目錄中，你可以這樣指定：

```
vxdname db      'C:\MYVXD\MYVXD.VXD', 0
```

然而如果你的驅動程式安裝在 Windows 95 目錄或其 system 子目錄中，或是由 path 所指定的目錄中，那麼你可以只提供主檔名和副檔名就好。

也有可能你的常駐程式需要兩個（或更多）VxDs。這種情況下只要在你的 INT 2Fh 處理常式回返之前，產生額外的 startup information structures 即可。

我的驅動程式在哪裡？

如果 TSR（常駐程式）需要一個 VxD，你可以提供 end user 一些彈性，並同時簡化你的程式碼。作法是給予 VxD 一個名稱，只與該 TSR 的副檔名不同，並總是把 VxD 放置在和 TSR 相同的磁碟目錄中。TSR 只要掃描環境區塊（environment block）的尾端即可決定其路徑全名，像這樣：

```
        mov     es, es:[2Ch]    ; ES:DI -> environment block
        xor     di, di          ; ...
        cld                     ; force forward direction
        xor     al, al          ; AL = 0 to compare against
        xor     cx, cx          ; CX = essentially infinite count
        dec     cx              ; ...
@@:     repne scasb             ; find end of current variable
        scasb                   ; is next byte 0 ?
        jne     @B              ; if not, skip next variable
        add     di, 2           ; ES:DI now points to our name
```

你所需要的 VxD 的主檔名，就是 TSR 的主檔名，只有副檔名需要改變。例如你可能得把 .COM 改爲 .VXD。

一個比較特別的技倆是把真實模式常駐程式寫成一個 .EXE 檔，並以它做爲 VxD 的

stub 程式。也就是在你的 VxD 的 .DEF 檔中，將 STUB 敘述句指定爲你的 TSR 可執行檔（請參考 MSDN 的 ***Binding a TSR to a VxD*** 一文）。這種情況下你只能夠提供唯一一個檔案。你還是可以掃描環境區塊以獲知自己的名稱，就像上述程式碼的動作那樣，但是不需要改變副檔名。

載入次序（load order）與重複載入（duplication）

VMM 不應該載入同一個驅動程式兩次。VMM 以驅動程式的 DDB（Device Description Block，也就是你以 *Declare_Virtual_Device* 巨集所產生出來的一個結構）所記錄的識別碼（ID）來區分不同的驅動程式。因此，即使兩個驅動程式有著不同的磁碟檔案、不同的 NAME 指定句（在 .DEF 檔中）、不同的名稱（在 *Declare_Virtual_Device* 巨集中），它們還是可能重複！不過，要是兩個驅動程式的 ID 皆爲 0（也就是 *Undefined_Device_ID*），它們將永遠不會被 VMM 視爲重複 -- 甚至即使它們其實是完全一樣的！是的，重複載入一個「undefined-ID 之驅動程式」並不算錯誤。

由於「被要求載入同一個驅動程式一次以上」的情況很可能發生（事實上是非常容易發生），你恐怕有必要瞭解 VMM 載入驅動程式的次序：

1. VMM 先載入被 INT 2Fh, function 1605h 中的 Startup Information Structures 所指定的那些 VxD；按照「此結構出現在串列中的次序」載入之。這個次序正與常駐程式 "hooked" INT 2Fh 的次序相反，所以最後一個被 hook 的程式反而有較高的優先權。
2. VMM 然後搜尋 registry database 中的 *StaticVxD*，如稍早所述。
3. 最後，VMM 載入那些在 SYSTEM.INI 中指名的驅動程式，並以出現的次序爲次序。

一個靜態（static）VxD 的起始與終結

Windows 95 以四個步驟將一個被靜態載入的驅動程式初始化。在 CPU 被切換到保護模式之前，VMM 先呼叫 VxD 中的真實模式初始化函式（如果有的話）。切入保護模式之後且 CPU 尚處於 interrupts disabled 狀態時，VMM 會送給驅動程式一個 *Sys_Critical_Init* 訊息。一旦 CPU 被 enabled，VMM 送給驅動程式一個 *Device_Init* 訊息。大部份驅動程式在這時候執行其初始化動作。然後，VMM 再送給驅動程式一個 *Init_Complete* 訊息。

真實模式的初始化動作

當你啓動電腦，CPU 原先處於真實模式。當 MS-DOS 初始化，CPU 仍處於真實模式，此時 MS-DOS 會載入 CONFIG.SYS 所指定的驅動程式，隨後並處理 AUTOEXEC.BAT。這些動作都完成之後，MS-DOS 自動發出一個 WIN 命令，將 Windows 95 載入。所以，Windows 95 是在真實模式中開始其生命。當它決定哪些 static VxDs 要被載入之後，才將 CPU 切換到保護模式。

注意　如今提這個注意事項，有點賣弄學問的嫌疑，不過我還是要提醒你。如果你利用 AUTOEXEC.BAT 載入一個 expanded memory manager（EMM）如 EMM386，電腦會在 Windows 95 啓動時進入 V86 模式。這會影響你「在初始化時期存取線性位址空間中的最前面 1MB」的能力，因爲 Windows 的 V86MMGR device 需要從 EMM 手中接管對 Upper Memory Blocks （譯註：UMB，640K~1MB 之間的區域）的控制權。此事發生於 V86MMGR 的 *Sys_Critical_Init* 處理常式之中。

在尚未進入保護模式之前，VMM 會執行 VxD 的真實模式初始化函式。這個階段的一部份就是 INT 2Fh, function 1605h（startup broadcast），使常駐程式有機會指定它所需要的 VxDs。在這個階段中，VMM 還會讀取 registry database 以及 SYSTEM.INI 的

[386enh] 區段中的 device 敘述句。VMM 會檢查其中所指定的每一個驅動程式，呼叫其可能存在（也可能不存在）的真實模式初始化函式。

真實模式初始化函式可以做下列任何（或所有）事情：

- 檢查是否重複載入同一個驅動程式；若是，選擇到底要載入哪一個。
- 提供 instancing data（見以下說明）所需區域。
- 指定某些 conventional memory（譯註：640K 以前的記憶體）給 virtual device。
- 和另一些真實模式程式聯絡。
- 阻止 Windows 95 啓動。

instance data 所謂 instance data 是指「佔用每一個虛擬機器中的相同虛擬位址、但對於不同的虛擬機器卻可以有不同內容」的資料。在 Windows 95 中，這個術語總是代表那些「最初置於位址空間中的第一個 MB，但不屬於任何硬體裝置」的資料。例如，DOSKEY 的 recall buffer 可被視爲是；但 video RAM 就不是 -- 甚至即使兩個區域有著相同的位址而且在不同的 VMs 中擁有不同的內容。

並不是每一個 VxD 都需要一個真實模式初始化函式。通常它存在的理由是用來獲得那些「你必須呼叫真實模式碼才能決定」的資訊，而那些資訊又是在 *Sys_Critical_Init* 時才需要（彼時真實模式暫時無法運作）。另一個理由是爲了偵測是否重複載入。

程式 結構

你必須把你的真實模式初始化函式碼及其用到的所有資料，包裝在 VxD 的一個 16 位元 segment 之中：

```
VxD_REAL_INIT_SEG       ; starts _RCODE segment
...
VxD_REAL_INIT_ENDS      ; ends _RCODE segment
```

也就是說，你使用 VxD_REAL_INIT_xxx 巨集來爲真實模式初始化函式劃定界線。如果你使用 Windows 3.1 DDK 中的聯結器 LINK386，你還必須指定函式的起始點（方法是在 END 敘述句中指定它），做爲 VxD 的主要進入點。但如果你使用的是 Visual C++ 或 Windows 95 DDK 的聯結器，就不需要指定任何東西！

進入（entry）狀態與退出（exit）狀態

進入真實模式初始化函式時，暫存器內容如表 **8-1** 所示。置於 AX 中的版本號碼，在 Windows 95 是 0400h，在 Windows 3.1 則是 030Ah。你可以呼叫 ECX 所指示的服務常式，檢查 registry 或 SYSTEM.INI 中的記錄，或是和真實模式載入器（real mode loader）聯絡。如果驅動程式是因 SYSTEM.INI 的 device= 或 registry 中的記錄而被載入，EDX 中的參考資料將是 0，否則它將是 INT 2Fh, function 1605h（引發驅動程式被載入）的 startup information structure 中的 *SIS_Reference_Data* 欄位內容。

暫存器	內容
AX	VMM 版本號碼（例如 0400h 或 030Ah）
BX	旗標值，意義如下： **Bit0**：*Duplicate_Device_ID*。如果設立，表示另一個相同 ID 的驅動程式已被載入。 **Bit1**：*Duplicate_From_INT2F*。如果設立，表示已經有一個驅動程式因爲 INT 2Fh, function 1605h 的指定而被載入。 **Bit2**：*Loading_From_INT2F*。如果設立，表示這個驅動程式是因 INT 2Fh, function 1605h 的指定而被載入。
ECX	初始化服務常式的進入點（segment:offset），適用於 Windows 3.1 以上。
EDX	0，或是 INT 2Fh, function 1605h, startup information structure 中的參考資料。
SI	指向「內含 MS-DOS environment」之 segment。
CS, DS, ES	「內含 _RCODE segment」之 paragraph。

表 8-1　進入眞實模式初始化函式時，暫存器的內容

版本號碼 Windows 95 以 0004h（也就是 4.00 版）回應 Win32 程式對於版本號碼的詢問，但卻以 0400h 回應 VxD 的詢問。Win16 程式如果呼叫 *GetVersion*，會獲得 5F03h（也就是 3.95 版）。Windows 95 欺騙 Win16 程式的原因是，有太多 Win16 程式沒有正確使用 *GetVersion*。某些程式詢問「這是版本３嗎？」其實意思是要詢問「這是版本２之後的版本嗎？」，另一些程式可能沒有在比較數字之前先保留好 *GetVersion* 傳回值的 byte 次序，所以它們會獲得「0004h（4.00 版）小於 0A03h（3.10 版）」的結論，這當然是不正確的！

暫存器 BX 中的「驅動程式重複旗標」（bit0 和 bit1）表示這個驅動程式的 ID 和另一個已被載入的驅動程式相同。請注意 bit1（*Duplicate_From_INT2F*）表示先前有一驅動程式和本驅動程式的 ID 相同。你可以檢查 bit2 （*Loading_From_INT2F*）以確定你的驅動程式是以 INT 2Fh, function 1605h 載入。

你的真實模式初始化函式應該以一個 near return 結束，此時暫存器內容應該如**表 8-2** 所示。只要設定 AX 爲 0（*Device_Load_Ok*），就表示正常結束。

暫存器	內容
AX	回返值，由以下位元組成： **Bit 0**：*Abort_Device_Load*。如果設立，告訴 VMM 說不要載入此一 VxD（但是相同 ID 的其他 VxDs 還是可以載入）。 **Bit 1**：*Abort_Win386_Load*。如果設立，告訴 VMM 說不要啓動 Windows 95。 **Bit 15**：*No_Fail_Message*。如果設立，告訴 VMM 說不要在載入此一 VxD 失敗時顯示錯誤訊息。
BX	指向由 owned pages 所組成的一個串列。
EDX	給「保護模式初始化函式」的參考資料。
SI	指向由 instance data 所組成的串列。

表 8-2 眞實模式初始化函式結束時，暫存器的內容

我很驚訝竟然需要使用一個 *near* return。似乎 VMM 在執行時期放了某些碼到真實模式

的 initialization segment 之中，然後做一個 far call，呼叫那些碼。彼等再做一個 near call，呼叫你的初始化函式。你的 near return 也因此會到達一個 far return 指令，把控制權交還給 VMM。

Owned pages 串列是一個由短整數所組成的向量，最後以 0 值結束。**Instance data**（見稍早說明）的 item 串列則是一個由 *Instance_Item_Struc* 結構所組成的向量，最後以 32 位元的 0 值結束（請看圖 **8-2**）。這兩個向量都和你的真實模式初始化程式碼以及資料一起置於 _RCODE segment 中。稍後我會討論 owned pages 的觀念。至於 instance data，將在第 10 章探討。

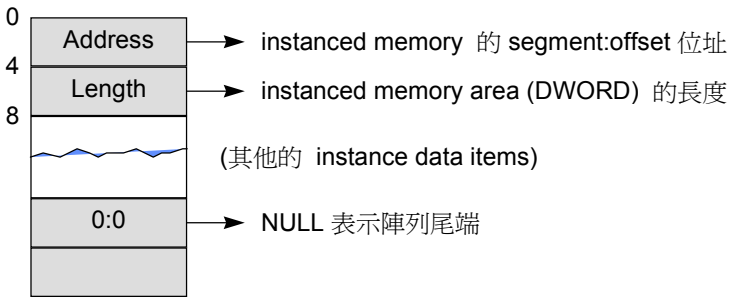


圖 8-2　Instance item structure 所組成的串列

檢查是否重複載入

如果你懷疑你的 VxD 會被重複載入，可以提供一個真實模式初始化函式檢查之。例如：

```
rminit: test    bx, Duplicate_Device_ID       ; duplicate ?
        setnz   al                            ; yes, set code
        cbw                                   ; ..
        xor     bx, bx                        ; no owned pages
        xor     si, si                        ; no instance data
        ret                                   ; return
```

如果前一個 TEST 指令的結果不是 0，SETNZ 指令會將 AL 設定爲 1。暫存器 AL 的

值相當於 *Abort_Device_Load* 旗標。

你可能會驚訝爲什麼 Windows 95 不乾脆自動偵測「重複載入」這件事，並載入指定的第一個驅動程式。爲什麼它要呼叫你的真實模式初始化函式，由你做決定呢？原因是，每次對驅動程式提出請求，可能表示需要宣告一個 **owned page**，或是將 conventional memory（譯註：低於 640KB）的某些區域標示做爲 "instancing" 之用。Windows 95 呼叫你的初始化函式爲的就是完成這個可能性。但這也意味你必須在函式回返時設定 BX 和 SI 爲 owned page 和 instance-data -- 甚至即使你在回返時設立了 *Abort_Device_Load* 旗標。

Owned Pages

Windows 95 在 conventional memory region（譯註：低於 640K）中爲每一部 VM 建立一個 page table。每一個這樣的 page table 用來將某些虛擬位址直接映射到對應的實際位址上，並將其他一些虛擬位址映射到 extended memory（譯註：高於 1MB）的 pages 身上。極其罕見的情況下，你需要改寫預設的虛擬位址映射關係，而「owned page 串列」允許你這麼做！

利用 Windows 95 的 EBIOS 驅動程式，我可以對 owned page 做最好的說明。在一台 PS/2 機器上，BIOS 配置一塊 extended BIOS data area，或稱爲 **EBIOS area**，做爲讀寫暫存區。「回溯相容性」使得 EBIOS area 必須落在 0040:0000h 位置上，PS/2 BIOS 因而必須使用實際記憶體中的這塊額外範圍。每一部虛擬機器需要以一個「虛擬即爲真實」的位址來指向此一 EBIOS area，否則 BIOS 就沒辦法在這部虛擬機器上工作。

其他的 Windows 95 系統組件可能並不明白 EBIOS area 的存在，因爲 BIOS 可以經由軟體中斷 INT 12h 通知大家說，實際記憶體結束於 EBIOS area 的起始處之前。預設的映射行爲將因此把 extended memory pages 指定至 EBIOS 位址範圍中。由於這樣的行爲是不正確的，EBIOS 驅動程式必須改寫它。

EBIOS 驅動程式的真實模式初始函式首先確定 EBIOS area 是否落在 640KB

conventional memory 區域的最後 40KB 之中。如果是，則宣稱「從 EBIOS area 起始位址至 A000h:0000h」之間都是它的領土，作法是傳回 BX，指向一個由 16 位元整數組成的陣列，最後一個整數是 0。舉個例，如果 EBIOS area 起始於 9E80h:0000h，陣列將內含三筆資料：

```
exc_bios_page dw 9Eh, 9Fh, 0
```

此後，每有一個新的 VM 被產生出來，EBIOS 驅動程式就利用 *_PhyIntoV86* service，將 physical pages 9Eh 和 9Fh 分別映射到虛擬機器的 virtual page 9Eh 和 9Fh。

Device Callout Protocol

在真實模式初始化函式之中，你可以以一般方式使用 BIOS 和 MS-DOS 提供的功能函式。如果你要和一個常駐程式（例如 MS-DOS 驅動程式或 TSRs）溝通，你可以使用幾乎任何你想用的方法。Microsoft 定義了一組 device callout 公約，架構在軟體中斷 INT 2Fh 之上，將此通訊方法標準化。

爲了使用這個 device callout，請準備暫存器內容如下：AX 等於 1607h，BX 等於你的 VxD 識別碼。然後發出軟體中斷 2Fh。那些期待此一 callout 之常駐程式一定已經 hooked 了這個軟體中斷，它們將搜尋 function 1607h，以及「BX 暫存器等於你的 VxD ID」者。其他暫存器以及 INT 2Fh 回返時的暫存器內容，都由你自行設定。

眞實模式的 initialization services

進入真實模式初始化函式時，ECX 暫存器內含一個服務常式位址（形式爲 paragraph:offset ），此常式提供數個公用函式，用來解析 registry 和 SYSTEM.INI 中的記錄。爲能夠使用這些函式，你首先得將服務常式的位址儲存在一個 DWORD 之中：

```
rminit: mov     _ServiceEntry, ecx
...
_ServiceEntry   dd   0
```

呼叫 *_ServiceEntry* 變數中的位址，會比使用 VMMREG.H 所提供的巨集更簡單些。

要喚起某一函式，請將函式號碼放進 AX 暫存器中，然後執行一個 far call：

```
mov     ax, n             ; AX = function index
call    [_ServiceEntry]   ; call service routine
```

表 **8-3** 顯示可用的服務常式。我發現很難找出那些服務常式的技術說明文件。前 6 個服務常式（0h~06h）在 Windows 95 DDK 的 "Programmer's Guide" 中有說明（在 "Real Mode Initialization" 那一節，標題爲 "Reference"）；在 MSDN 中搜尋 *LDRSRV_COPY_EXTENDED_MEMORY* 關鍵字，也可以找到它們。Registry services 可以在 Windows 95 DDK 的 "Kernel Services Guide" 中找到說明 -- 在 "Registry Services Reference" 那一節，標題爲 "Real-Mode Functions"。你也可以在 MSDN 中搜尋 *LDR_RegCloseKey* 關鍵字找到它們。

AX 內容	功能名稱
0h	LDRSRV_GET_PROFILE_STRING
01h	LDRSRV_GET_NEXT_PROFILE_STRING
03h	LDRSRV_GET_PROFILE_BOOLEAN
04h	LDRSRV_GET_PROFILE_DECIMAL_INT
05h	LDRSRV_GET_PROFILE_HEX_INT
06h	LDRSRV_COPY_EXTENDED_MEMORY
100h	LDR_RegOpenKey
102h	LDR_RegCloseKey
105h	LDR_RegQueryValue
106h	LDR_RegEnumKey
108h	LDR_RegEnumValue
109h	LDR_RegQueryValueEx

表格 8-3　眞實模式初始化函式的 service calls

舉個例子，假設你在你的 SYSTEM.INI 中加入一個 [myvxd] 區段，並指定你的驅動程

式的 I/O port 位址如下：

```
[myvxd]
port = 1234h
```

你的程式碼可以這樣讀取上述內容：

```
rminit: mov     _ServiceEntry, ecx              ; save service return address
        ...
        mov     ax, LDRSRV_GET_PROFILE_HEX_INT  ; i.e. 05h
        mov     ecx, -1                         ; ECX = default value
        mov     si, offset secname              ; DS:SI -> section name
        mov     di, offset varname              ; DS:DI -> setting name
        call    [_ServiceEntry]                 ; get SYSTEM.INI setting
        mov     ioaddr, ecx                     ; (return in ECX)
        ...
_ServiceEntry dd 0
secname db    'myvxd', 0
varname db    'port', 0
ioaddr  dd    0
```

如果你要處理 registry，就比較複雜些，部份原因是你必須先知道 registry 的結構，尤其是引起你的驅動程式被載入的那一部份。下面是個小範例，用來在真實模式初始化過程中從 registry 取出 port 的位址（你可以在書附光碟的 \CHAP08\REALMODEINIT 目錄中找到這些程式碼）：

```
#0001  ; RMREG.ASM -- Test of real-mode registry access
#0002
#0003      .386p
#0004      include vmm.inc
#0005      include vmmreg.inc
#0006      include regstr.inc
#0007
#0008  Declare_Virtual_Device RMREG, 1, 0, rmreg_control,\
#0009      Undefined_Device_Id, Undefined_Init_Order
#0010
#0011  Begin_Control_Dispatch RMREG
#0012  End_Control_Dispatch RMREG
#0013
#0014  VxD_REAL_INIT_SEG
```

```
#0015     mov   _ServiceEntry, ecx
#0016
#0017     LDR_RegOpenKey HKEY_LOCAL_MACHINE, <offset namevxd>, ds,\
#0018        <offset hvxd>, ds
#0019
#0020     test  ax, ax
#0021     jnz   fail1
#0022
#0023     LDR_RegOpenKey hvxd, <offset myname>, ds, <offset hme>, ds
#0024
#0025     test  ax, ax
#0026     jnz   fail2
#0027
#0028     Ldr_RegQueryValueEx hme, <offset portname>, ds, 0, 0, 0,\
#0029        <offset port>, ds, <offset portsize>, ds
#0030
#0031     test  ax, ax
#0032     jnz   fail3
#0033
#0034     ...        ; do something with "port" value
#0035
#0036  fail3:
#0037     LDR_RegCloseKey hme
#0038  fail2:
#0039     LDR_RegCloseKey hvxd
#0040  fail1:
#0041
#0042  alldone:
#0043     xor   ax, ax
#0044     xor   bx, bx
#0045     xor   si, si
#0046     ret
#0047
#0048  _ServiceEntry dd 0
#0049
#0050  namevxd  db    REGSTR_PATH_VXD, 0
#0051  hvxd     dd    0
#0052  myname   db    'RMREG', 0
#0053  hme      dd    0
#0054  portname db    'port', 0
#0055  port     dd    0
#0056  portsize dd    size port
#0057  VxD_REAL_INIT_ENDS
#0058
#0059     end
```

在這段程式碼中我用了數個定義於 VMMREG.INC 的巨集，它們統統都希望找到一個名爲 *_ServiceEntry* 的 DWORD，內含初始化服務函式的進入點位址。

要打開「管理某一硬體裝置」的 registry key，你必須先確定那個 key 的名稱。*REGSTR_PATH_VXD* 常數（定義於 REGSTR.INC）被定義爲\System\CurrentControlSet\Services\VxD。要打開 RMREG key，最簡單的作法就是先打開一個指向此一路徑的 key，然後再打開在它之下的一個名爲 *RMREG* 的 subkey。這也就是 *LDR_RegOpenKey* 所完成的事情。你可以使用 *LDR_RegQueryValueEx*，在一個已被打開的 key 中獲得一個設定值。

上述真實模式範例中的一個潛在問題是，它必須依賴編譯時期的決定，才知道你將把 registry 項目放在何處。稍後你會看到，VxD 的保護模式初始化函式會以比較穩健也比較輕鬆的態度來處理 registry key，這是因爲 *_GetRegistryPath* service 會計算出「使你的 VxD 被載入」的那個 registry path，你不再需要具備那麼多的 registry 知識。

和保護模式初始化函式溝訊

「真實模式初始化函式」能夠和驅動程式的「保護模式部份」交換資訊，只要在 return 之前把 EDX 設爲一個任意的 32 位元 reference data 即可。當 Windows 95 送出保護模式初始化訊息給驅動程式，這個 reference data 佔用 EDX 暫存器。Windows 95 也把這個值儲存在 DDB 結構的 *DDB_Reference_Data* 欄位中。DDB 就是你以 Declare_Virtual_Device 巨集所產生出來的一個所謂的 **Device Descriptor Block** 結構。

在 Windows 95 之中，你可以使用 *LDRSRV_Copy_Extended_Memory* service 來配置並初始化一塊 extended memory（譯註：1MB 以上），然後以 EDX 將記憶體位址傳回，供驅動程式的保護模式部份使用。例如，Windows 95 Configuration Manager 的真實模式初始化函式就利用這個 service 來傳遞一塊相當長的資料給其保護模式初始化函式；這塊資料內含目前之硬體的 profile 資訊。在前一版 Windows 中（彼時 *LDRSRV_Copy_Extended_Memory* 尚未存在），你必須配置真實模式記憶體，並以 EDX 將其位址交給保護模式初始化函式。由於在 Windows 95 退出之前，沒有任何人會釋放這

塊記憶體，所以它會在每一個 MS-DOS 虛擬機器中產生一個「永久的污漬」。*LDRSRV_Copy_Extended_Memory* 不只可以避免真實模式記憶體的破碎（fragmenting）情況，而且你「必須」使用它，因爲 Windows 95 會在進入保護模式之前，將所有在真實模式初始化函式中配置而來的記憶體釋放掉。

保護模式的初始化

決定載入哪一些驅動程式之後，VMM 把 CPU 從真實模式切換到保護模式。這是在 CPU 處於中斷 disabled 狀態時發生的。然後 VMM 送給每個驅動程式的 control procedure 一個 Sys_Critical_Init 訊息。所有驅動程式都處理過此一訊息之後，VMM 令 CPU 進入中斷 enabled 狀態，並送給每個驅動程式一個 Device_Init 訊息。然後它再送給每個驅動程式一個 Init_Complete 訊息，完成整個初始化程序。

VMM 依照遞增的「初始化次序」（在 Declare_Virtual_Device 巨集中指定）送出那些訊息。Microsoft 供應的每一個驅動程式都有自己的一個初始化次序常數，定義在 VMM.H 檔。下面是一段節錄：

```
#define VMPOLL_INIT_ORDER      0x064000000
#define UNDEFINED_INIT_ORDER   0x080000000
#define WINDEBUG_INIT_ORDER    0x081000000
#define VDMAD_INIT_ORDER       0x090000000
```

換句話說，WINDEBUG 在 VMPOLL 之後（但在 VDMAD 之前）被初始化。

有時候，就像 MS-DOS 驅動程式或常駐程式（TSRs）的載入次序有其重要性一樣，VxDs 的初始化次序亦有其重要性：最後一個對中斷做 hook 動作的程式，有較高優先權來處理該中斷。此外，你或許需要依賴其他 VxD 提供的 services，因此你必須比那個 VxD 更晚載入。如果你不在乎初始化次序，請選用 *Undefined_Init_Order*。如果你要在某個 device 之前或之後初始化，請選擇稍小於或稍大於「該 device 之初始化次序」的一個常數。爲了讓相鄰的「初始化次序常數」之間還有空間存在，你最好和相鄰的 device 保持距離，讓別人還有機會安插在你和你的相鄰 device 之間。請保持大約 1000h 空間。

Calling Sequences

針對三個保護模式初始化訊息，VMM 使用相同的 calling sequence。表 **8-4** 表示那三個訊息出現時，general 暫存器的內容。

暫存器	內容
EAX	0（Sys_Critical_Init），1（Device_Init）或 2（Init_Complete）
EBX	System VM 的 handle（也就是 VM control block 的位址）
EDX	來自真實模式之初始化函式的 reference data。如果真實模式初始化函式沒有定義之，此值爲 0。
ESI	VMM386.EXE 命令列尾（command tail）的位址。這是 Program Segment Prefix（PSP）的命令列尾的位址，其格式是先有一個前導的計數位元組，然後是命令列參數字串，然後是一個 0Dh 記號。

表 8-4　保護模式初始化訊息發生時，暫存器的內容。

如果你的 VxD 中的上述三個訊息處理常式在回返時將 carry flag 設立（或是一個 C 函式傳回 FALSE），VMM 會放棄載入這個 VxD。你的 VxD 應該清除 carry flag（或傳回 TRUE）以表示正常地完成。

下面是個以 C 完成的 VxD 骨幹，有一個由 assembly 語言所寫的 control procedure，用來處理上述三個初始化訊息（程式碼放在書附光碟的 \CHAP08\C-DDK 目錄中）：

DEVDCL.ASM

```
#0001  ; DEVDCL.ASM -- Assembly language interfaces for Sample VxD
#0002
#0003     .386p
#0004     include vmm.inc
#0005     include debug.inc
#0006
#0007  Declare_Virtual_Device MYVXD, 1, 0, MYVXD_control,\
#0008     Undefined_Device_ID, Undefined_Init_Order
#0009
```

```
#0010  Begin_Control_Dispatch MYVXD
#0011
#0012  Control_Dispatch Sys_Critical_Init, OnDeviceInit,     sCall, <ebx, edx>
#0013  Control_Dispatch Device_Init,       OnSysCriticalInit, sCall, <ebx, edx>
#0014  Control_Dispatch Init_Complete,     OnInitComplete,    sCall, <ebx, edx>
#0015
#0016  End_Control_Dispatch   MYVXD
#0017
#0018     end
```

MYVXD.C

```
#0001  // MYVXD.C -- Sample Virtual Device Driver
#0002
#0003  #define WANTVXDWRAPS
#0004
#0005  #include <basedef.h>
#0006  #include <vmm.h>
#0007  #include <debug.h>
#0008  #include <vxdwraps.h>
#0009
#0010  #pragma VxD_ICODE_SEG
#0011  #pragma VxD_IDATA_SEG
#0012
#0013  BOOL _stdcall OnDeviceInit(PVMMCB hVM, DWORD refdata)
#0014         {
#0015         return TRUE;
#0016         }
#0017
#0018  BOOL _stdcall OnSysCriticalInit(PVMMCB hVM, DWORD refdata)
#0019         {
#0020         return TRUE;
#0021         }
#0022
#0023  BOOL _stdcall OnInitComplete(PVMMCB hVM, DWORD refdata)
#0024         {
#0025         return TRUE;
#0026         }
```

Sys_Critical_Init

VMM 把 CPU 切換到保護模式，並在中斷 enabled 之前將 Sys_Critical_Init 訊息送出給各個驅動程式。大部份驅動程式並不需要處理這個訊息。以下是處理這個訊息的原因：

- 你的驅動程式對某個軟體中斷而言，作用像 DOS extender。也就是說，你的驅動程式有責任接受保護模式所發出的 MS-DOS 或 BIOS 軟體中斷，並在 V86 模式中執行此一中斷（必須先將指標參數從保護模式格式轉爲真實模式格式）。你應該在 Sys_Critical_Init 訊息期間，使用 *Set_PM_Int_Vector* 來 "hook" 你的中斷，於是其他所有 VxDs 都可以在獲得 Device_Init 訊息時透過 *Exec_VxD_Int* 自由呼叫你的中斷。
- 對其他某些「在 Device_Init 期間會被某些 VxDs 呼叫」的 services 而言，你的驅動程式負有責任。例如，V86MMGR device 在 Sys_Critical_Init 期間接管 upper memory blocks（譯註：UMB，640K~1024K），於是其他 VxDs 就有可能在 Windows 95 啓動之前，存取真實模式程式正在使用的所有記憶體。
- 你想要提供一個預設的中斷處理常式，並且是最後一個被呼叫。只要在 Sys_Critical_Init 期間 hooking 那個中斷，你的驅動程式就可以比 VMM 安裝的預設處理常式更優先執行（因爲 VMM 比你的 VxD 更早初始化），但是晚於那個在 Device_Init 期間（或更晚）"hook" 此一中斷之驅動程式。
- 你的驅動程式匯出(export)一些 VxD services 給較早初始化的驅動程式使用。你或許因此需要初始化某些資料，以便對於 Device_Init 期間發出的 service calls 做出回應。

我想我不必再強調，驅動程式處理這些訊息時，一定不能夠將中斷 enable。還有，不要使用那些「會執行真實模式碼」的 services（例如 *Exec_Int*）。最後，請注意，儘可能不要在這個訊息期間做太多工作，以免遺失硬體中斷。

Device_Init

Windows 95 在中斷 enable 之後，送出 Device_Init 訊息。大多數驅動程式回應此訊息的方式是：執行初始化工作的大部份。由於中斷已經被 enabled 了，你可以執行耗時的工作，不必擔心會影響系統。你也可以執行真實模式碼。我無法明白告訴你應該在這訊息期間做些什麼事，因爲那完全視 VxD 的設計目的而定。

Init_Complete

在將所有驅動程式初始化之後，但還沒有釋放 initialization segments 並取得 instance snapshot（第 10 章介紹）之前，VMM 送出 Init_Complete 訊息。

很少驅動程式需要處理這個訊息。我們可以從 DOSMGR device 來思考處理此訊息的一個理由：通常真實模式 MS-DOS 驅動程式需要獲得所有（或部份）它所需要的記憶體。較新的驅動程式會 "hook" INT 2Fh 並使用 "INT 2Fh, function 1605h, broadcast" 來完成此一任務，但是 VxD 可能需要面對一個 Windows 95 並不認識的真實模式驅動程式做管理等細節動作。*DOSMGR_Instance_Device* service 就做這個事情，但它只在 Init_Complete 期間有效。

初始化過程中一些有用的 Services

初始化過程中，你的驅動程式可能需要詢問 registry 或 SYSTEM.INI 中的資料項。你可以使用以下幾組 services：

- registry services（例如 *_RegQueryValue*），用來處理 registry database。
- get profile services（例如 *Get_Profile_Decimal_Int*），用來讀出 SYSTEM.INI 的內容。
- convert services（例如 *Convert_Boolean_String*），用來將字串轉換爲數值。通常你會以 *Get_Profile_String* 讀取 SYSTEM.INI 中的字串。

除此之外還有其他選擇。例如 *Get_Name_Of_Ugly_TSR*，允許驅動程式看看是否有一個不合作（因而可能妨礙驅動程式）的 TSR。

這裡所描述的所有 services，除了 registry services 之外，都被置於 initialization code segment 內，因此在 Init_Complete 訊息之後就不可再用。

使用 Registry Services 以下內容說明用來對驅動程式組態（configure）的 initialization services 的用法。假設你正在寫一個名爲 MYVXD.VXD 的驅動程式，它用

來將一個 I/O port 虛擬化。爲了方便討論，讓我們假設 MYVXD 是一個傳統程式，其中沒有 Windows 95 的 plug & play 架構。更進一步假設你遵循前述行爲，使用 system registry 來載入驅動程式。在 \\HKLM\System\CurrentControlSet\Services\VxD\ MYVXD 這個 key 之下，你應該有這樣的 values：

```
StaticVxD=c:\MyProd\Myvxd.vxd
Port=1234h
```

其中的 *StaticVxD* 導至你的驅動程式被載入。*Port* 應該內含二進位資料，經由 registry services，提供決定性的組態資料給你的驅動程式：

```
extern VxD_Desc_Block MYVXD_DDB;

BOOL __stdcall OnDeviceInit(PVMMCB hVM, DWORD refdata)
   {                                // OnDeviceInit
   char regpath[256];
   char *p;
   HKEY hkey;
   DWORD port;
   DWORD cbdata = sizeof(port);

   _GetRegistryPath(&MYVXD_DDB, regpath, sizeof(regpath));

   p = regpath + strlen(regpath);
   while (p > regpath && p[-1] == ' ')
      --p;   // trim trailing blanks from name
   *p = 0;

   _RegOpenKey(HKEY_LOCAL_MACHINE, regpath, &hkey);
   _RegQueryValueEx(hkey, "Port", NULL, NULL, (PBYTE)&port, &cbdata);
   _RegCloseKey(hkey);
   return TRUE;
   }                                // OnDeviceInit
```

_GetRegistryPath 可以取得被載入之 MYVXD 的相關 registry path，本例之中它應該是：

```
System\CurrentControlSet\Services\VxD\Myvxd
```

_GetRegistryPath 傳回的 path 最終是以你的 device 名稱（8-bytes，不足者以空白補足）做爲結尾；device 名稱係直接從 DDB 取得。在你將它用於 *_RegOpenKey* 之前，你必須先去除尾部的空白字元。打開這個 key 之後，你可以利用 *_RegQueryValueEx* 詢問組態值（像是 Port 之類）。完成之後，你必須呼叫 *_RegCloseKey* 關閉這個 registry key。

在初始化過程中，你只需要處理 registry 中的 HKEY_LOCAL_MACHINE 這一支。而在 Device_Init 訊息之後，你可以存取整個 registry。

使用 Profile Services 載入驅動程式的舊方式是藉由 SYSTEM.INI 中的 device= 敘述句。記錄組態資料的一個老舊作法是利用額外的 .INI 設定句。例如你可以讓 SYSTEM.INI 內含以下內容：

```
[386enh]
device=c:\Myprod\myvxd.vxd
...
[myvxd]
port=1234h
```

那麼下列程式碼的作用將相當於先前對 registry 的動作：

```
BOOL __stdcall OnDeviceInit(PVMMCB hVM, DWORD refdata)
   {                                // OnDeviceInit
   DWORD port = Get_Profile_Hex_Int(-1, "myvxd", "port");
   return TRUE;
   }                                // OnDeviceInit
```

這雖然比較簡單，卻比較不牢靠，而且也不符合對 device 組態時的標準使用者介面。此法亦適用於 Windows 3.1。

結束（Termination）

我們花了許多精力來討論系統的啓動。當 Windows 95 結束，回到真實模式，static VxDs 一樣有許多動作要做。VMM 會送出四個 system control messages 給每一個還存活的 VxDs：

- System_Exit，宣佈 Windows 95 即將停工。此時 CPU 仍處於保護模式，在 System VM 中執行真實模式碼仍然安全。所有其他的 VMs 將在此時被摧毀，Windows KERNEL module 則已經結束。
- System_Exit2，緊跟著上個訊息出現，意義相同。VMM 以「初始化次序的相反次序」送出 System_Exit2，讓 VxDs 在必要時候得以依賴那些由「有著較早初始化次序的 VxDs」所提供的 services。
- 所有 VxDs 都處理完上述兩個訊息之後，VMM 把中斷 disable 掉，並送給每個 VxD 一個 Sys_Critical_Exit 訊息。
- 最後，VMM 送給每個 VxD 一個 Sys_Critical_Exit2 訊息。就像 System_Exit2 一樣，這個訊息以「初始化次序的相反次序」送出。

大部份 VxDs 不需要處理任何 system shutdown 訊息。然而如果你極不尋常地要對 real-mode image 做任何動作，你得處理其中的訊息。例如 V86MMGR device 在 Windows 95 啓動時利用一個 breakpoint 改變了 XMS server 的進入點；它必須在 Windows 95 停工時恢復該進入點，使 XMS server 回復正常機能。

動態（dynamic）VxDs

Windows 95 可以動態載入並卸除（unload）某個驅動程式。此一特性主要的推動力量來自於對硬體的動態重新組態（dynamic hardware reconfiguration）支援，但你也可以利用這個性質來載入或卸除純粹自己使用的驅動程式。

爲了讓你的驅動程式有被動態載入的能力，你必須做兩件事情：

- 將此驅動程式建立爲一個只能在 Windows 95 中使用的程式。Windows 3.1 驅動程式不能夠被動態載入。也就是說，不要在你組譯和編譯時，定義前置處理器巨集（preprocessor macro）*WIN31COMPAT*。
- 在你的驅動程式 .DEF 檔的 VXD 敘述句中加上 DYNAMIC 關鍵字如下：

```
VXD MYVXD DYNAMIC
```

這一節的其他部份，我要描述一個應用程式如何動態載入和卸除驅動程式，我也要描述 Windows 95 如何初始化和終結一個可被動態載入的驅動程式。

32 位元應用程式

32 位元程式的動態載入介面是經由一個名爲 *CreateFile* 的正規 Win32 API 完成。爲了載入一個驅動程式，你必須呼叫 *CreateFile* 並指定以下參數：

```
HANDLE hDevice = CreateFile("\\\\.\\pathname", 0, 0,
                            NULL, 0, FILE_FLAG_DELETE_ON_CLOSE, NULL);
```

當你以這種方式來使用 *CreateFile*，大部份參數並不重要，你可以把它們都設爲 0。奇怪的前置符號 \\.\（注意，C 會把兩個倒斜線變成一個倒斜線）表示你要求 Windows 95 載入驅動程式並傳回一個 device handle。通常 *CreateFile* 用來打開一個可讀或可寫的檔案，我並不想得罪這個介面的設計者，但我實在很驚訝竟然會以這麼常用的 API 來負擔新任務。「實作並載入一個 dynamic VxD」的範例碼放在書附光碟的 \CHAP08\DYNALOAD-VTOOLSD 目錄中。

CreateFile 的倒數第二個參數是屬性與旗標。使用 *FILE_FLAG_DELETE_ON_CLOSE* 會使得 Windows 95 在你呼叫 *CloseHandle* 時自動卸載這個驅動程式：

```
CloseHandle(hDevice);
```

和以前一樣，*CreateFile* 如果不能夠成功載入驅動程式，會傳回一個 *INVALID_HANDLE_VALUE*（也就是 -1）。

欲以此介面動態載入驅動程式，你還必須做另一件事情：你必須在你的 VxD 中處理 W32_DEVICEIOCONTROL 訊息。VWIN32 VxD 利用這個訊息處理來自 Win32 程式的 *DeviceIoControl* calls，並將此訊息送給因爲應用程式呼叫 *CreateFile* 而被動態載入的驅動程式。此訊息的處理常式最少應該做下列事情：

```
#include <vwin32.h>
DWORD __stdcall OnW32DeviceIoControl(PDIOCPARAMETERS p)
   {                          // OnW32DeviceIoControl
   switch (p->dwwIoControlCode)
      {                       // process control call
      case DIOC_GETVERSION:
      return 0;
      }                       // process control call
   }                          // OnW32DeviceIoControl
```

就像許多發生於 Windows 的事情一樣，我實在想不出，爲什麼「送出一個 W32_DEVICEIOCONTROL 訊息」必須成爲「實作 *CreateFile*」的一部份。

16 位元應用程式

16 位元應用程式（不管在真實模式或保護模式中）可藉由呼叫 VXDLDR 的 API 來動態載入一個 VxD。這比起 32 位元程式所做的動作，有許多不同，也比較複雜。第一個步驟是利用 INT 2Fh, function 1684h 定出 API 進入點（我將在第 10 章詳細描述這個重要介面）。下面是程式碼大要：

```
include vmm.inc      ; for VXDLDR_DEVICE_ID
...
vxdldr dd 0          ; address of VXDLDR API entry point
...
mov    ax, 1684h     ; function 1684h: find VxD API entry point
xor    di, di        ; clear ES:DI first to detect failure
mov    es, di        ; ...
mov    bx, VXDLDR_DEVICE_ID
int    2Fh           ; locate loader's API entry point
mov    ax, es        ; was entry point found ?
or     ax, di        ; ...
jz     fail          ; if not, can't dynamically load drivers

mov    word ptr vxdldr, di
mov    word ptr vxdldr+2, es
```

一旦載入並儲存 VXDLDR 的 API 位址，你就可以將 AX 暫存器設定爲 VXDLDR 的一個 service 代碼，然後就可以呼叫此進入點以載入或卸除一個 VxD。欲動態載入一個

驅動程式，請含入 VXDLDR.H 或 VXDLDR.INC 並寫這樣的碼：

```
include vxdldr.inc                      ; for VXDLDR services
...
mov    ax, VXDLDR_APIFUNC_LOADDEVICE    ; i.e., 1
mov    dx, offset filename              ; DS:DX -> filename + 0
call   [vxdldr]                         ; try to load driver
jc     fail                             ; CF set if unable to load
```

在此片段中，*filename* 代表一個以 null 爲結束字元的字串，內含你希望載入之驅動程式的路徑。如果你的驅動程式名爲 MYVXD.VXD，你可以這樣寫：

```
filename db   'myvxd.vxd', 0
```

欲卸除一個驅動程式，你可以這樣寫：

```
mov    ax, VXDLDR_APIFUNC_UNLOADDEVICE  ; i.e., 2
mov    bx, -1
mov    dx, offset name                  ; DS:DX -> driver name + 0
call   [vxdldr]                         ; try to unload driver
jc     fail                             ; CF set if unable to unload
```

name 代表一個以 null 爲結束字元的字串，內含驅動程式的內部名稱。如果你在 Declare_Virtual_Device 巨集中把你的驅動程式命名爲 MYVXD，就可以這樣寫：

```
name db   'MYVXD', 0
```

注意，在載入和卸除兩動作中，你用了兩個不同的命名格式。*LOADDEVICE* 函式需要的是 path 名稱，*UNLOADDEVICE* 則只需要 device 名稱（也就是出現在 Declare_Virtual_Device 巨集中的名稱）。

一個動態（dynamic）VxD 的起始與終結

Windows 95 初始化一個 dynamic VxD 的作法是，送給它一個 Sys_Dynamic_Device 訊息。而在卸載一個 dynamic VxD 之前，送出的是 Sys_Dynamic_Device_Exit 訊息。

由於 Windows 95 在結束之前早就拋棄了 initialization segments 並取得 instance snapshot，dynamic VxD 不可以使用下面任何一個「只能於初始化過程中使用」的 services：

```
_Add_Global_V86_Data_Area          Get_Next_Arena
_AddFreePhysPage                   Get_Next_Profile_String
_AddInstanceItem                   Get_Profile_Boolean
_Allocate_Global_V86_Data_Area     Get_Profile_Decimal_Int
_Allocate_Temp_V86_Data_Area       Get_Profile_Fixed_Point
_Free_Temp_V86_Data_Area           Get_Profile_Hex_Int
_GetGlblRng0V86IntBase             Get_Profile_String
_SetFreePhysRegCalBk               GetDOSVectors
_SetLastV86Page                    Locate_Byte_In_ROM
Allocate_PM_App_CB_Area            MMGR_SetNULPageAddr
Convert_Boolean_String             OpenFile
Convert_Decimal_String             PageFile_Init_File
Convert_Fixed_Point_String         Set_Physical_HMA_Alias
Convert_Hex_String                 V86MMGR_NoUMBInitCalls
DOSMGR_BackFill_Allowed            V86MMGR_Set_Mapping_Info
DOSMGR_Enable_Indos_Polling        V86MMGR_SetAvailMapPgs
DOSMGR_Instance_Device             V86MMGR_SetLocalA20
Get_Name_Of_Ugly_TSR               VDMAD_Reserve_Buffer_Space
```

dynamic VxD 應該明白地 "unhook" 任何它所 "hooked" 的中斷，並且（通常）取消任何「不再應該有持續影響力」的動作。

關於「dynamic VxDs 應該自行清理乾淨」這一戒條，有一個例外，那就是分別經由 *Allocate_V86_Call_Back* 和 *Allocate_PM_Call_Back* 兩個 services 配置而來的「V86 模式 callback」和「保護模式 callback」。並沒有 service 可以釋放這兩種 objects！如果驅動程式每次被動態載入時都得重新配置一個 callback，會浪費資源。我們可以使用 static code 和 data segments 來解決這個問題：

```
VxD_STATIC_DATA_SEG
mycallback dd 0
loaded dd    0
VxD_STATIC_DATA_ENDS

BeginProc myfunction, static
       cmp   loaded, 1     ; is VxD really loaded ?
       je    @f           ; if yes, good
       [failsafe code that doesn't leave static segment]
@@:    jmp   realcallback  ; transfer to real callback
       ...
EndProc myfunction

BeginProc OnSysDynamicDeviceInit, init
       mov   loaded, 1
       cmp   mycallback, 0
       jne   @f
       mov   esi, offset32 myfunction
       VMMCall Allocate_PM_Call_Back
       mov   mycallback, eax
@@:    ...
EndProc OnSysDynamicDeviceInit

BeginProc OnSysDynamicDeviceExit, locked
       mov   loaded, 0
       ...
EndProc OnSysDynamicDeviceExit

BeginProc realcallback, locked
       ...
EndProc realcallback
```

（我使用 assembly 語言，因爲純粹以 C 語言來完成 callbacks 碼實在是件痛苦的事）

最後值得注意的一點是，不管我在本章中如何賣力地區別，dynamic VxDs 其實並不真的和 static VxDs 有多大不同。Dynamic VxDs 收到兩個 system control messages（Sys_Dynamic_Device_Init 和 Sys_Dynamic_Device_Exit），那是 static VxDs 絕不會收到的；除此之外，以及除了在 Init_Complete 之後對於 services 的呼叫有些限制之外，任何在 static VxDs 出現的碼都可以放到 dynamic VxDs 之中，反之亦然。此外，當一個 dynamic VxD 被載入，它會收到所有可能發生的 system control messages。大部份 dynamic VxDs 不在乎那些訊息，但如果它們在乎，它們有機會處理之。

第 9 章

VxD 程式設計基礎技術

前數章已經把你帶到一個足以讓你撰寫頗具意義的 VxD 的位置上。撰寫 VxD 時，你所使用的程式介面（Application Programming Interface，API）是以 Microsoft 所謂的 **VxD service**s 爲中心。這一章我首先要談如何以 assembly 和 C 語言使用這些 API。然後我要探討三個主要的資料結構；是的，做爲一個 VxD 撰寫者，你必須處理這三個結構：**VM control block**，**client register structure**（CRS），以及 **thread control block**（TCB）。然後，我會介紹一些對大部份 VxDs 而言十分普遍的 API services，範圍涵蓋記憶體管理、event 管理，以及 thread synchronization（執行緒同步控制）。

VxD Service 介面

你可以在這一節學到如何呼叫其他 VxD 所提供的 service 進入點，以及如何匯出（export）你自己的一個 service API。我們已經由前數章獲得了一些基礎，特別是你已經知道一個以 assembly 語言撰寫的驅動程式如何使用 *VMMCall* 和 *VxDCall* 巨集來產生一個 INT 20h 指令，動態聯結 service 函式。這一節將描述這些巨集的內部機制。你也已經知道，在 VxD 程式設計領域中，C 或 C++ frameworks 提供了 INT 20h 這一介面

的 wrappers（外包函式），但是你得學習如何撰寫你自己的 wrappers 以彌補那些 framework 的疏忽和錯誤。最後，你會學到如何在自己的 VxD 中定義並實作出一個 service API。

VMMCall 巨集和 VxDCall 巨集

Assembly 驅動程式使用 *VMMCall* 和 *VxDCall* 巨集來喚起 services。*VMMCall* 用來是呼叫 VMM 所開放的 services，*VxDCall* 則用來呼叫其他 VxDs 所開放的 services。兩個巨集有完全相同的語法：

```
VMMCall service          ; assembly-language services
VMMCall _service, <args> ; C-convention services
```

或

```
VxDCall service          ; assembly-language services
VxDCall _service, <args> ; C-convention services
```

VxD services 有兩個呼叫習慣。Assembly 語言呼叫習慣（應用於大部份從 Windows 3.0 和 Windows 3.1 遺留下來的 services）藉由暫存器和旗標做爲參數和回返值的橋樑。例如，*Get_Cur_VM_Handle* 是一個「暫存器導向」的 service，其傳回值放在 EBX 之中，並且不改變其他任何暫存器：

```
VMMCall Get_Cur_VM_Handle
[code that references VM control block via EBX]
```

C 語言呼叫習慣則是把參數放在堆疊之中，傳回值由 EAX 傳遞。這種 service 會保護 EBX, ESI 和 EDI 暫存器，並可能改變其他暫存器值以及旗標值。所有這類 services 都以一個底線 '_' 做爲名稱前導。因此，除了某些 VWIN32 services 之外，任何 services 如果其名稱是以一個底線做爲前導，它就是所謂的 **C-convention service**s。舉個例子，*_HeapAllocate* service 就是，它從 ring0 heap 中配置一塊記憶體給 VxD 使用：

```
VMMCall  _HeapAllocate, <<size somestruc>, HeapZeroInit>
test     eax, eax
jz       failure
mov      address, eax
```

當你使用 *VMMCall* 巨集呼叫一個 C-convention service，你應該使用角括弧來標示參數。爲了讓組譯器（assembler）的巨集處理器將指定的各個參數正常解析出一個個"token"，請使用成對的角括弧。傳回值放在 EAX 暫存器中。所以在這個例子中，我們指定兩個參數給 *_HeapAllocate* service：第一個參數是 *size somestruc*，指定我們想要配置的 bytes 個數；第二個參數是常數 *HeapZeroInit*，表示我們希望 *_HeapAllocate* 將配置來的記憶體先清理乾淨（設爲 0）。

當你想要呼叫一個由 VMM 以外的 VxD 提供的 service，通常你需要含入一個對應的表頭檔。這些 services 總是以 VxD 的名稱做爲前導字；如果是 C-convention service，更有一個底線 '_' 在最前面。你可以使用 *VxDCall*（代替 *VMMCall*）來呼叫它們。例如，SHELL device 有一個 service，可用來決定版本號碼：

```
include shell.inc
...
VxDCall SHELL_Get_Version
...
```

我要解釋 *VMMCall* 和 *VxDCall* 的一個變種。有時候 VxD service 會又呼叫另一個 VxD service，爲了尋求最佳效率，Microsoft 提供了 *VMMJmp* 和 *VxDJmp* 巨集。一如你所預期，這些巨集直接 jump 到目標點，不需要先 push 一個回返位址到堆疊中。當被呼叫之 service 回返，它會迴避其最近的呼叫者，並因此節省了一個 RET 指令。例如，trapped port 的 I/O service routine 傳統上是以一個 jump to *Simulate_IO* 做爲結束：

```
BeginProc IOCallback
   ...
   VMMJmp Simulate_IO   ; doesn't return
EndProc IOCallback
```

注意　回憶一下 BeginProc 和 EndProc，在除錯版（debug build）中它們內含 call logging code。爲了避免太過份，*VMMJmp* 和 *VxDJmp* 內含條件式的 assembly 指令，如此一來在除錯版（debug build）中，它們發出一次 call 之後，緊跟著一個 return。而在零售版本（retail build）中，它們就只是 jump。

針對 C 語言呼叫設計的 Wrappers（外包函式）

第7章花了不少篇幅介紹 DDK 和其他開發工具（如 Vireo Software 公司的 VToolsD）如何（以及爲什麼）提供 VxD services 的 C wrappers。一般而言，你不會明顯地在 C 程式中使用 *VxDCall* 或 *VMMCall* 巨集，而是呼叫某個 wrapper 函式。下面就是使用 DDK 的典型呼叫動作：

```
#define WANTVXDWRAPS
#include <basedef.h>
#include <vmm.h>
#include <shell.h>
#include <vxdwraps.h>
...
PVOID address = _HeapAllocate(sizeof(somestruc), HEAPZEROINIT);
if (!address)
    // handle error
...
WORD wShellVer = SHELL_Get_Version();
```

我在這個例子中示範如何呼叫 *_HeapAllocate*。這是一個 C-convention service，在 assembly 語言中原本該爲它使用 *VMMCall*。我也示範如何呼叫 *SHELL_Get_Version*，這是一個習慣使用暫存器的 service（也就是說 service 接受暫存器值做爲參數，並以暫存器來傳回數值），在 assembly 語言中原本該爲它使用 *VxDCall*。請注意，這些不同的 service 呼叫法在 C 程式中已經消失無蹤了。VToolsD C framework 的作法也很類似，不過需要含入的表頭檔更少，而且你得注意，在宣告 *SHELL_Get_Version* 的傳回值時有一點不同：

```
#include <vtoolsc.h>
...
PVOID address = _HeapAllocate(sizeof(somestruc), HEAPZEROINIT);
if (!address)
    // handle error
...
DWORD dwShellVer = SHELL_Get_Version();
```

注意　在 VToolsD 和 DDK 之間有個令人煩惱的不一致性，那就是對 *VMMCall* 的拼字。Vireo 拼爲 *VMMcall*（小寫 c），而 Microsoft 拼爲 *VMMCall*（大寫 C）。至於 *VxDCall* 倒是沒有紛歧。不過，倒是只有 Microsoft 才定義有 *VMMJmp* 和 *VxDJmp*。

有時候你得爲自己的 service 函式撰寫 wrappers。當你使用 DDK，這份需求可能常常湧現，不過即使你使用 VToolsD，也不能夠完全避免這項需求。大部份時候你可以依賴 inline assembly 來完成這份工作。舉個例子，你使用 DDK 並且想藉由 *Get_Profile_Decimal_Int* 取出 SYSTEM.INI 檔中的一筆項目。*Get_Profile_Decimal_Int* 是一個「沒有預先定義 wrapper」的 service，你可以在 C 檔案中放入以下宣告：

```
#pragma warning(disable:4035)
DWORD VXDINLINE Get_Profile_Decimal_Int(PCHAR pszProfile,
    PCHAR pszKeyName, DWORD dwDefault)
    {
    _asm mov eax, dwDefault
    _asm mov esi, pszProfile
    _asm mov edi, pszKeyName
    VMMCall(Get_Profile_Decimal_Int);
    }
#pragma warning(default:4035)
```

VXDINLINE 會被展開爲 *static __inline*，致使編譯器在你呼叫上述函式的地點產生 inline code。由於這個 wrapper 並沒有明顯 return 一個值，所以 Microsoft 編譯器通常會產生一個警告訊息（#4035）。如果你不喜歡它，只要將此警告訊息 disable 即可，要不就是把 EAX 值儲存到一個傀儡變數（dummy variable）中，然後明顯 return 之。

當你定義自己的 inline function wrappers，有時候你也需要通知編譯器說哪些暫存器會被 service 改變內容。否則編譯器的最佳化邏輯可能會誤以爲它可以重複使用那些暫存器內容。VMM.H 內有一個 *Touch_Register* 巨集，可以讓這個修改動作顯現出來。這個巨集只是簡單地產生出一個 *XOR* 指令，清理某個暫存器。下面就是一個可能會破壞 ECX 和 EDX 的 C-convention service wrapper 示範：

```
DWORD VXDINLINE _TestGlobalV86Mem(DWORD VMLinAddr,
    DWORD nBytes, DWORD flags)
    {
    DWORD result;
    Touch_Register(ECX)
    Touch_Register(EDX)
    _asm push flags
    _asm push nBytes
    _asm push VMLinAddr
    VMMCall(_TestGlobalV86Mem)
    _asm add esp, 12
    _asm mov result, eax
    return result;
    }
```

此例中我使用一個傀儡變數（dummy variable）做爲傳回值，以避免警告訊息 #4035。*Touch_Register* 巨集所產生的 XOR 指令其實是沒有必要的，但是呼叫這個巨集可以確保避免最佳化所導至的錯誤。

為 VxD Service Call 建立一個 Wrapper

我沒有辦法描述純粹的技術程序，讓你將 assembly 語言的 VxD services 轉換爲 C 語言的 wrapper，但我可以給一些提示，幫助你完成這個常常需要做的工作。

有一個非常簡單的程序可以幫助你包裝 C-convention service。回憶一下，C-convention service 通常以底線做爲名稱前導符號，並且預期它們的參數放在堆疊之中。對於這樣的 services，只要宣告一個 inline 函式，有著相同的參數及排列次序。面對任何型別爲 mask 或 integer 的參數，則以 DWORD 取代之。只要可能，就對指標參數使用特定的 *typedef* 名稱（在說明文件中，指標參數通常帶有一個 offset32 directive），以便從編譯器錯誤檢驗中獲得最大利益。面對沒有指定資料型別或函式型態的指標，則使用 *PVOID* 或 *PFN*。 定義函式時，使用 *__declspec(naked)* directive 並提供一個軀殼，內含一個 *VMMJmp* 或 *VxDJmp*（它們在 C 程式中是完全一樣的），後面緊跟著適當的 service 名稱。

舉個例，如果你需要爲 *_HeapAllocate* 寫一個 wrapper（你其實不需要，因爲標準的 DDK 已經爲這個常被使用的 service 提供了一個 wrapper），可以查詢線上說明文件，並且發現有這麼一段 assembly 原型：

```
include vmm.inc
VMMcall _HeapAllocate, <nbytes, flags>

or   eax, eax          ; zero if error
jz   not_allocated
mov  [Address], eax    ; address of memory block
```

根據前述方法，你應該在你的表頭檔中準備一個函式宣告：

```
PVOID VXDINLINE __declspec(naked) _HeapAllocate(DWORD nbytes,
   DWORD flags)
   {                      // _HeapAllocate
   VMMJmp(_HeapAllocate)
   }                      // _HeapAllocate
```

對於習慣以暫存器做爲介面（所謂 register-convention）的 services，你可以遵循一個稍微明顯的程序，將參數規格轉換爲一個函式原型。

舉個例，*Get_Profile_Decimal_Int* 的說明文件這樣描述自己：

```
mov    eax, Default                ; default value
mov    esi, OFFSET32 Profile       ; points to section name
mov    edi, OFFSET32 Keyname       ; points to entry name
VMMcall Get_Profile_Decimal_Int

jc     not_found                   ; carry set if entry not found
jz     no_value                    ; zero set if entry has no value

mov    [Value], eax                ; entry value
```

這份說明文件進一步說明 *Profile* 和 *Keyname* 都是以零值字元做爲結束的字串。這些說明導引我們寫出這樣的函式原型（未完工）：

```
DWORD VXDINLINE Get_Profile_Decimal_Int(PCHAR pszProfile,
   PCHAR pszKeyName, DWORD dwDefault
```

譯註：以下的 section 和 key 和 value 都是 .INI 檔案格式的術語，所以我仍然保持本書原有風格，不譯它，閱讀時請注意。

我不加上右括弧，因爲這個 service 有一個輸出。這個 service 傳回的旗標值用來表示兩種錯誤和一種幾近錯誤的狀態。如果 carry flag 設立，要不是進入點不存在，就是其 value 不是個合理的十進位整數。如果 carry flag 被清除但 zero flag 被設立，表示這個 section 和 key 存在，但沒有 value （如果 SYSTEM.INI 有一個 Keyname= 句子，但是等號右邊沒有任何東西，就是這種情況）。如果 carry flag 和 zero flag 都被清除，表示 value 存在，於是暫存器 EAX 將內含轉換後的數值。上述任何一種錯誤情況，EAX 都將內含預設值。

面對這麼複雜的錯誤狀態，你可以加上一個參數，用來儲存錯誤代碼。於是完整的函式原型如下：

```
DWORD VXDINLINE Get_Profile_Decimal_Int(PCHAR pszProfile,
   PCHAR pszKeyName, DWORD dwDefault, PDWORD pError);
```

或者，你也可以把目標放在「只要獲得數值就好」，至於那是讀自 SYSTEM.INI 的值，或是預設值，不在乎！這樣的話你就不必加上最後面那個參數。

一旦你知道函式原型，要以 inline assembly 來寫出函式內容就不會太困難了。#220 頁所顯示的例子中，我基本上只是把說明文件的內容轉換爲程式碼而已。

這些 wrapper 的地理中心位置就是 *VMMCall* 巨集。*VMMCall* 是一個前置處理巨集，會被展開爲一個 INT 20h 指令，以及必要的 inline data。例如，*VMMCall(Get_Profile_Decimal_Int)* 會產生出以下的碼（*Get_Profile_Decimal_Int* 的 service ID 爲 000100ABh）：

```
_asm
    {
    int 20h
    _emit 0abh
    _emit 0
    _emit 1
    _emit 0
    }
```

Microsoft 定義了大量像 *Get_Profile_Decimal_Int* 這樣的常數。DDK 已經巧妙地將這個（以及其他）service ID 定義爲一種 enum 型態。表頭檔內含下面這樣的東西：

```
Begin_Service_Table(VMM, VMM)
VMM_Service(Get_VMM_Version, LOCAL)
...
VMM_Service(Get_Profile_Decimal_Int, VMM_ICODE)
VMM_Service(Convert_Decimal_String, VMM_ICODE)
...
End_Service_Table(VMM, VMM)
```

這些碼會被展開爲：

```
enum VMM_SERVICES {
VMM_dummy = (VMM_DEVICE_ID << 16) - 1,  // i.e. 0000FFFF
__Get_VMM_Version,                      // i.e., 00010000
...
__Get_Profile_Decimal_Int,              // i.e., 000100AB
__Convert_Decimal_String,               // i.e., 000100AC
...
Num_VMM_Services };
```

這項技術很明顯是依賴「C 會自動爲每一個 enum 常數加 1」的事實，除非你又指定了另一個起始點。

請注意，*VMM_Service* 巨集所定義的符號和 service 名稱相同，但卻有兩個前導底線。這個名稱上的差異，使你得以定義一個和 service 相同名稱的函式。如果 enum 常數上的名稱沒有加以修飾的話，你會在編譯時獲得「符號重複定義」的錯誤訊息。

雖然前面的討論定位在 Microsoft DDK，但你可以把完全相同的討論搬到 VToolsD 來，因爲 Vireo Software 公司也釋出了許多和 DDK 相同的表頭檔。有一個情況需要注意，如果你對 VxD service 做了一個 inline call，而 VToolsD 已爲該 service 定義了 wrapper，那麼你必須做一個奇怪的動作：

```
#undef Get_VMM_Version
VMMcall(Get_VMM_Version)
...
```

原因是 VToolsD 表頭檔內已經爲每一個 wrapped 函式在當時的（current）code segment 中安插了一個 #define 句子，因此當你切換到...唔...假設說是 locked code segment 好了，你將獲得：

```
#define Get_VMM_Version  LOCK_Get_VMM_Version
```

正常情況下，這樣的定義會妨礙你使用 *VMMCall* 巨集中的符號，因爲編譯器會把它展開爲 *LOCK_Get_VMM_Version*。因此當 *VMMCall* 嘗試使用它自己展開的未定義符號 *__LOCK_Get_VMM_Version*，會出現錯誤。

定義你自己的 Services

到目前爲止，我已經談過如何呼叫 VxDs 所開放的 services。你當然也有可能爲自己的 static VxD 定義一些 services 供其他 VxDs 呼叫。再一次我得說，除非你使用 VToolsD，否則就得重新磨亮你的 assembly 技巧。爲了讓事情儘量簡化，讓我們假設，你的 MYVXD device 要匯出（export）一個 service，傳出它自己的版本號碼。傳統作法有兩個步驟：先產生一個表頭檔，準備給呼叫端含入；然後在內含 Declare_Virtual_Device 巨集的那個模組中定義一個 service table。表頭檔（MYVXD.INC）有以下內容：

```
MYVXD_DEVICE_ID equ 4242h
Begin_Service_Table MYVXD
MYVXD_Service MYVXD_Get_Version, LOCAL
End_Service_Table MYVXD
```

一般而言你應該產生一個表頭檔（含入檔），定義你的 device ID（一個獨一無二的識別碼，必須詢問 Microsoft 以獲得此號碼），並列出你要匯出的 services。在這個例子中，*Begin_Service_Table* 巨集定義了一個新的巨集，名爲 *MYVXD_Service*。你將呼叫此巨集一次或多次，以指定自己的 services。這個巨集內部定義了一個明確的常數，可以經由 device ID 和 service 序號識別出一個個 services。此例只呼叫 *MYVXD_Service* 一次，定義出符號 *@@MYVXD_Get_Version*，其值等於 42420000h。

習慣上你應該準備一個 *Get_Version* 做爲第一個 service。如果 service 以 C convention 的方式被呼叫，你應該爲 service 名稱加上一個前導底線；如果是以暫存器來傳遞參數，那麼就不要在 service 名稱前加底線。對於一般的 services，再沒有什麼其他規則或建議要說了。注意，如果你改變 service table 中的排列次序，你的 VxD 的所有使用者就都必須重新編譯。因爲這個理由，你不可以在販售你的 VxD 之後又隨意更改 services 次序，除非你能夠完全掌握你的客戶。

應用程式如果要呼叫你的 VxD service，應該這麼做：

```
include myvxd.inc
...
VxDCall MYVXD_Get_Version
```

你必須定義一個 service table，以及用來完成那些 service 的函式。Service table 內含指向 service 函式的指標。你的 DDB（Device Description Block）內有一個欄位指向 service table，這也就是 VMM 找到 service table 的第一個起點。產生 service table 的最簡單方法就是定義所謂的致能符號（enabling symbol）*Create_xxx_Service_Table*，然後含入自己的表頭檔，然後再定義你的 DDB：

```
        .386p
        .xlist
        include vmm.inc
        .list

        Create_MYVXD_Service_Table = 1
        include myvxd.inc

Declare_Virtual_Device MYVXD, 1, 0, MYVXD_control, \
              MYVXD_Device_ID, undefined_init_order

Begin_Control_Dispatch MYVXD
End_Control_Dispatch MYVXD

BeginProc MYVXD_Get_Version, service, locked
        mov   ax, word ptr MYVXD_DDB.DDB_Dev_Major_Version
        xchg  ah, al
        clc
```

```
        ret
EndProc MYVXD_Get_Version

        end
```

BeginProc 和 *EndProc* 兩個巨集幫助你定義了自己的 service 函式。參數 *service* 表示 device service table 內含一個指標，指向此函式本身。Service 函式的細節絕大部份操之在你，不過以 *Get_Version* 而言，最少應該在 return 時將 carry flag 清除乾淨，以表示成功。這樣的習慣符合我在第 4 章所描述的：應用程式若想知道某個驅動程式是否被載入，就呼叫其 *Get_Version* service。如果你的驅動程式尚未被載入，VMM 會出面，將 carry flag 設立，表示失敗。所以如果你不把 carry flag 清乾淨，就會混淆呼叫者的視聽，以爲你的驅動程式並未載入。

在哪裡放置你的 service 函式，是你的自由，但是你的決定會影響該函式在表頭檔中的宣告。如果你是在與 service table 相同的原始碼檔案中定義 service 函式，請在 *xxx_Service* 巨集中使用 *LOCAL* 關鍵字。否則就不要使用任何關鍵字，那麼便會被自動加上 *ENTERN* 宣告。

稍早我曾說過，static VxD 可以匯出 services 給別人呼叫。一般而言 dynamic VxD 不行，原因是兩個和 services 有關的重要資料結構（DDB 和 service table）所在的 segments 都不能夠在 VxD 被卸載（unload）後仍然存在。由於 VMM 快速地動態聯結 service 函式，其他許多 VxDs 可能在此 VxD 被卸載之後仍保有「穿透 service table 之直接指標」。問題是，沒有任何機制可以打斷這些聯結。如果這些指標的存在還不夠讓你麻煩，請注意，其他驅動程式還可能做了 *Hook_Device_Service* calls，因而改變了 service table。

讓你的 services 得被 C 語言處理

爲了讓其他以 C 或 C++ 完成的 VxDs 能夠使用你的 VxD，你必須在 assembly 含入檔（.INC）之外再提供一個 C 語言表頭檔（.H）。此外你還需要供應適當的 wrapper 函式。延續前一個例子，你必須產生這樣的 MYVXD.H 檔：

```
#define MYVXD_DEVICE_ID 0x4242

#define MYVXD_Service Declare_Service
#pragma warning(disable:4003) // not enough parameters is okey

Begin_Service_Table(MYVXD)
MYVXD_Service(MYVXD_Get_Version)
End_Service_Table(MYVXD)

WORD VXDINLINE MYVXD_Get_Version(void)
    {
    WORD ver;
    VxDCall(MYVXD_Get_Version)
    _asm mov ver, ax
    return ver;
    }
```

如果你熟讀 DDK，你就會知道，Microsoft 爲作業系統的許多組件（components）都提供了 assembly 含入檔和 C 表頭檔。很明顯 Microsoft 使用一種自動程序，將 C 表頭檔轉爲 assembly 含入檔，不過他們並未使用 MASM 6.11 所附的 H2INC 標準工具。面對一個大型專案，你可能得製造自己的工具，將 .H 檔轉換爲 .INC 檔。

VToolsD 驅動程式中的 Services

如果使用 VToolsD，就可以免除我前面所說的一些細碎動作。首先，你可以在執行 QuickVxd（圖 **9-1**）時宣告你的 service 進入點。QuickVxd 於是會在你的表頭檔中產生必要的 service table，並在你的原始碼檔案中產生必要的函式骨幹：

MYVXD.H

```
#0001  // MYVXD.h - include file for VxD MYVXD
#0002
#0003  #include <vtoolsc.h>
#0004
#0005  #define MYVXD_Major      1
#0006  #define MYVXD_Minor      0
#0007  #define MYVXD_DEVICE_ID   0x4242
#0008  #define MYVXD_Init_Order  UNDEFINED_INIT_ORDER
#0009
```

```
#0010  DWORD __cdecl MYVXD_Get_Version();
#0011
#0012  Begin_VxD_Service_Table(MYVXD)
#0013      VxD_Service(MYVXD_Get_Version)
#0014  End_VxD_Service_Table
```

MYVXD.C

```
#0001  // MYVXD.c - main module for VxD MYVXD
#0002
#0003  #define   DEVICE_MAIN
#0004  #include  "myvxd.h"
#0005  #undef    DEVICE_MAIN
#0006
#0007  Declare_Virtual_Device(MYVXD)
#0008  DWORD __cdecl MYVXD_Get_Version()
#0009      {
#0010          return (MYVXD_Major << 8) | MYVXD_Minor;
#0011      }
```

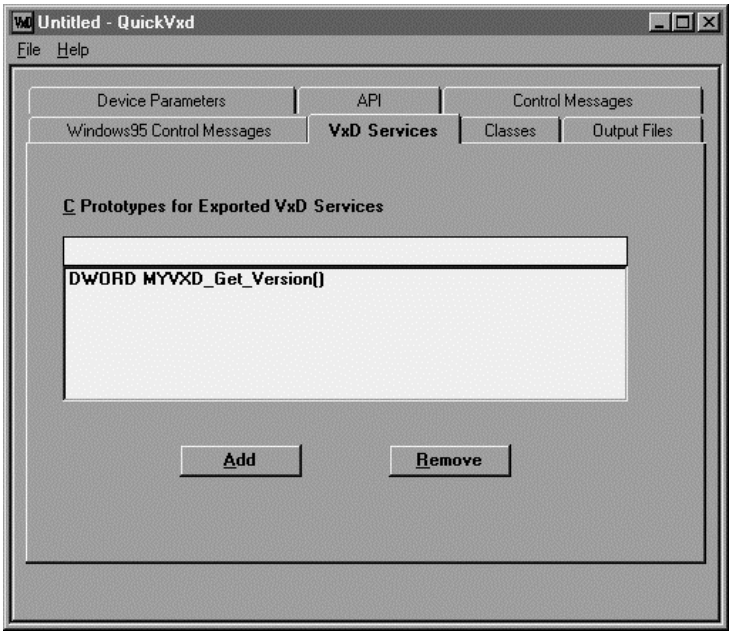


圖 9-1　QuickVxd 爲你產生 service 函式

麻煩的是，這些碼並不正確！注意，*MYVXD_Get_Version* 使用 C 呼叫習慣，所以其名稱應該有一個前導底線。但是 QuickVxd 所產生的函式名稱沒有前導底線。因爲這個 service 沒有參數，並且只會改變 EAX 暫存器，所以不同的呼叫習慣並沒有什麼大關係。比較嚴重的是這個函式並沒有在 return 之前清除 carry flag。你可以這樣修改 *MYVXD_Get_Version* 函式：

```
DWORD __declspec(naked) MYVXD_Get_Version()
   {
   _asm
       {
       mov  ax, (MYVXD_Major shl 8) or MYVXD_Minor
       clc
       ret
       }
   }
```

其中 *__declspec(naked)* 要求編譯器不要產生任何「可能在 return 之前改變 carry flag」的指令。如果你以 C 撰寫驅動程式，你所匯出之 services 大部份會以 C 呼叫慣例被呼叫之，所以你不需擔心 carry flag，但是你應該改變 service 函式名稱，爲它加上一個前導底線。

Hooking「Service 進入點」

DOS 程式員必定十分熟悉爲了改變系統行爲而做的中斷攔截（hook）動作。對 VxD services 而言，類似的動作（稱爲重導向，redirection）也是可能的。只要呼叫 *Hook_Device_Service*，VxD 就可以接管另一個 VxD 的 service 的進入點。例如下面這段碼（請參考書附光碟的 \CHAP09\SERVICEHOOK-ASM 目錄）就攔截了 *DOSMGR_Begin_V86_App* service：

```
#0001  VxD_LOCKED_DATA_SEG
#0002  next_begin dd 0
#0003  VxD_LOCKED_DATA_ENDS
#0004
#0005  BeginProc OnSysDynamicDeviceInit, locked
#0006          GetVxDServiceOrdinal eax, DOSMGR_Begin_V86_App
#0007          mov   esi, offset32 BeginHook
```

```
#0008          VMMCall Hook_Device_Service
#0009          jc    initfail
#0010          ...
#0011          clc
#0012          ret
#0013
#0014  initfail:
#0015          stc
#0016          ret
#0017  EndProc OnSysDynamicDeviceInit...
#0018
#0019  BeginProc BeginHook, service, hook_proc, next_begin, locked
#0020          ...
#0021          jmp   [next_begin]   ; chain to next hooker
#0022  EndProc BeginHook
```

這個例子使用 *Hook_Device_Service* 來重新導引對 *DOSMGR_Begin_V86_App* service 的呼叫，導向你自己的 *BeginHook* 常式。*GetVxDServiceOrdinal* 巨集將 32 位元的 service 識別代碼放到 EAX 暫存器，識別代碼內含 device ID 以及 service table 索引。如果該 device 用的是 *Undefined_Device_ID*，還是有可能攔截其 services，只要將該 device 的名稱（8 bytes，不足則填 0，區分大小寫）的起始位址放進 EDI 暫存器即可。

你必須以 *Hook_Proc* 宣告你的 hook 函式，其後必須緊跟著一個 page-locked DWORD 變數名稱（本例爲 *next_begin*）。VMM 會將原先的 service 進入點儲存在這個 DWORD 之中，它的存在使 VMM 得以實作出 *Unhook_Device_Service*。你的 hook 函式可以爲所欲爲，最後再經由 *next_begin* 間接跳到原來的處理常式，完成串鏈（chain）動作。

注意　*GetVxDServiceOrdinal* 巨集只是簡單地在 service 名稱前面加上 @@ 字首。你應該使用這個巨集，不要自己動手編出如此特別的命名習慣，因爲 Microsoft 有可能在往後版本中改變這個習慣。修飾 service 名稱，是爲了讓你得以指定與 service 名稱相同的 service 函式名稱，不至於遭受「符號重複」的錯誤打擊。

如果你攔截（hook）dynamic VxD 中的一個 service，你必須在卸除它之前，做 unhook 動作。下面是個例子：

```
BeginProc OnSysDynamicDeviceExit, locked
        GetVxDServiceOrdinal eax, DOSMGR_Begin_V86_App
        mov   esi, offset32 BeginHook
        VMMCall UnHook_Device_Service
        ...
        clc
        ret
EndProc OnSysDynamicDeviceExit
```

攔截 *DOSMGR_Begin_V86_App* 是學習「應用程式何時於虛擬機器中啓動」的官方作法。你也可以攔截其他 services，監視它們，或是代之以自己的碼。我曾經做過的案子中，有一個 pre-Windows extended DOS TSR 需要執行 Direct Memory Access（DMA）傳輸動作。VDMAD device 假設，任何來自虛擬機器的傳輸動作一定要使用 low-memory 位址（譯註：低於 640K），然後再自動爲每一個位址加上 *CB_High_Linear*。這個加法在我的案子中產生錯誤結果，因爲我已經在 DOS extender（它支援該 TSR）提供的實際（physical）位址中工作；攔截 *VDMAD_Lock_DMA_Region* 可以使我有機會避免不正確的位址調整，使 DMA 傳輸行爲正確運作。

除了 unhooking 條款，Windows 95 所實作的 *Hook_Device_Service* 還提供另一個優點，讓我們避免攔截一個 asynchronous（非同步的）service。在 Windows 3.1 之中，除非你很小心，否則下面情況有可能發生：「你呼叫了 *Hook_Device_Service*，它成功地修改了 service table 中的項目，於是成功地攔截了你所指定的 service。然而在你成功地將前一個函式位址儲存於變數之前，不幸發生了一個中斷，你的 hook 函式於是被呼叫起來。然後它嘗試串接那個尚未被設定內容的變數，於是系統就完蛋了」。Windows 95 把這個洞補了起來，它在儲存新舊處理常式位址的前後，將中斷 disabled 掉。於是，只有 nonmaskable 中斷（NMI）才有可能導至問題發生。

「你的 hook 函式取得控制權」時的進入狀態，和原先的 service 是一樣的；而它也必須在 return 時有著和原先 service 一樣的離開狀態。你必須保護原先 service 所保護的所有暫存器（和旗標）。你可以在你的 hook 函式中串接原先的 service，只要呼叫前面提過的那個 DWORD 就行。當你串接另一個 service 處理常式，你必須確定複製出它所期望的進入狀態。對一個靠暫存器來傳遞參數（所謂 register-based）的 service 而言，

你必須清除所有暫存器（而不只是做爲參數的那幾個）。這項要求反應到現實生活就是，你必須使用 assembly 函式來攔截一個 register-oriented service，這樣你才可以在適當地點放置 PUSHAD 和 POPAD 指令。

在 Windows 3.1 身上攔截 device services，技術上稍微有些不同。第一，沒有 *Unhook_Device_Service*，因爲根本就沒有 dynamic VxD。再者，*Hook_Device_Service* 利用 ESI 傳回前一個 service 的進入點位址，通常你必須把這個值儲存下來，才能夠在自己的 hook 函式中串接原來的 service。第三，前一版 DDK 是以公開說明的符號來定義 service ID，所以你只要像下面這麼做，不需使用 *GetVxDServiceOrdinal* 巨集：

```
mov eax, DOSMGR_Begin_V86_App
```

最後一點，*BeginProc* 沒有 *Hook_Proc* 選項，因爲攔截（hooking）和串接（chaining）的細節都是由被喚起的 VxD 來管理。

Windows 95 可以接納一個「攔截了某個 service」的 Windows 3.1 VxD -- 雖然它並不能夠從 hook chain 中移除一個 Windows 3.1 hook 函式。「攔截了某個 service」的 Windows 3.1 VxD 之所以能夠在 Windows 95 中正常運作，原因是 VMM 可以區分 hooking services 的新舊不同風格。*Hook_Proc* 屬性引起 *BeginProc* 產生兩個額外的 *JMP* 指令，放在 service 進入點之前。第一個 *JMP* 指令跳到 hook 函式去，第二個 *JMP* 指令迂迴經過你所提供的「前一個函式的指標」。靠著這兩個指令，VMM 就可以辨識出是 Windows 95（新風格）或 Windows 3.1（舊風格）的 hook 函式。

基礎資料結構

這一節我要介紹三個基礎資料結構，它們在 VxD 程式設計過程中到處可見：

1. VM control block（VMCB）
2. client register structure（CRS）
3. thread control block（TCB）

VM Control Block（VMCB）

VMM 持續追蹤一部 VM 的方法是，爲它維護一個 VM control block。這個冗長的名詞沒有官方縮寫字，所以我自己以 VMCB 來稱呼它。**圖 9-2** 列出 Microsoft 公開說明的少數幾個 VMCB 欄位。

```
struct cb_s {
   ULONG CB_VM_Status;        // 00 status flags
   ULONG CB_High_Linear;      // 04 unique address of V86 memory map
   ULONG CB_Client_Pointer;   // 08 address of client register structure
   ULONG CB_VMID;             // 0C virtual machine ID
   ULONG CB_Signature;        // 10 'VMcb' = 0x62634D56
};
```

圖 9-2　VM control block 中公開說明的欄位

欲解釋 VMCB 中的 *CB_High_Linear* 欄位，我必須暫時脫離主題，先說明 Windows 95 如何提供給 VM「V86 記憶體定址能力」。當 VM 正在執行，線性位址 000000h ~ 10FFEFh（也就是 0000h:0000h ~ FFFFh:FFFFh）映射到「內含此一 VM 之 V86 記憶體」的 pages 身上。VxD 可以存取這一組 pages，只要把 *CB_High_Linear* 加到一個 V86 線性位址上即可。假設某個 VM 有一個 "High_Linear" 位址爲 C1C00000h，VM 的 video RAM 可以同時在 000B8000h 和 C1CB8000h 被存取到，因爲 VMM 把描述「C1CB8000h 所在的那個 4MB」的 page directory entry 拷貝到 page table 中，所以它也相當於從 0 開始描述了這 4MB。

CB_Client_Pointer 欄位內含所謂的 client register structure（CRS）位址。此結構內含在此 VM 中執行之應用程式的「V86 模式和保護模式的 general 暫存器和 segment 暫存器」內容。

CB_VMID 欄位內含此一 VM 的識別代碼，是一個小整數。System VM 的 ID 總是 1。VMM 每產生一個 VM，就會指定一個 ID 給它，從此不改變。當你關閉 VM，其 ID 可以給新的 VM 使用。

CB_Signature 內含簽名符號 "VMcb"。當你對一個驅動程式除錯時，此字串可以派上用

場。除錯巨集 *Assert_VM_Handle* 也會檢查這個字串，不過該巨集只在 *DEBUG* 符號有被定義時才起作用。

VM Control Block 的狀態旗標（Status Flag）

表 **9-1** 列出各式各樣的 *VM_Status* 旗標值。你應該把這些旗標以及 VMCB 的其他所有欄位視爲唯讀（read only）。

名稱	Mask	說明
VMSTAT_EXCLUSIVE	00000001	要求獨佔執行（exclusive execution）
VMSTAT_BACKGROUND	00000002	允許背景執行（background execution）
VMSTAT_CREATING	00000004	VM 正被產生
VMSTAT_SUSPENDED	00000008	VM 被凍結（suspended）
VMSTAT_NOT_EXECUTEABLE	00000010	VM 已經到達了停工（shutdown）前的不可執行狀態
VMSTAT_PM_EXEC	00000020	VM 目前正在執行保護模式碼
VMSTAT_PM_APP	00000040	目前執行的是保護模式程式
VMSTAT_PM_USE32	00000080	保護模式程式是 32 位元程式（USE32）
VMSTAT_VXD_EXEC	00000100	*Exec_VxD_Int* 或 *Exec_PM_Int* 目前正 active
VMSTAT_HIGH_PRI_BACK	00000200	VM 是一個高優先權的 background task
VMSTAT_BLOCKED	00000400	VM 被一個 semaphore 凍結（blocked）住
VMSTAT_AWAKENING	00000800	VM 被凍結（blocked）後醒來
VMSTAT_PAGEABLEV86	00001000	部份 V86 記憶體被 DPMI 功能 0602h 標示爲 pageable
VMSTAT_V86INTSLOCKED	00002000	Nonpageable V86 記憶體被鎖住（不管 pager type 爲何）
VMSTAT_IDLE_TIMEOUT	00004000	VM 被 time slicing scheduler 排班
VMSTAT_IDLE	00008000	VM 已經釋放了它的 time slice
VMSTAT_CLOSING	00010000	VM 由於 Close_VM 訊息而被關閉

表 9-1 VM status 欄位的位元意義

如果 VM 中執行的程式碼使用 DPMI 的「模式切換常式」進入保護模式，*VMSTAT_PM_APP* 旗標會設立。如果 DPMI client 表示它是一個 32 位元程式，*VMSTAT_PM_USE32* 旗標會設立。請注意，Windows 95 本身是一個 USER16 DPMI client。上述兩個旗標之中，只有 *VMSTAT_PM_APP* 會爲 system VM 而設。*VMSTAT_PM_EXEC* 旗標表示 VM 現在正在執行保護模式碼，其反相是 VM 正在執行 V86 碼。

VMSTAT_BACKGROUND 旗標和 "MS-DOS Prompt" 屬性頁（Properties sheet）中的 Misc 附頁內的一個設定有關（譯註）。如果設爲 TRUE，表示當另一個 VM 獲得焦點，這個退至背景的 VM 還可以繼續運作。前版 Windows 還有一個 *VMSTAT_EXCLUSIVE* 設定：當一個 VM 獲得焦點時，把所有的 CPU 時間都給它。

譯註：這就是 Windows 95 中文版中的 "MS-DOS Prompt" 屬性頁的 Misc 附頁畫面：

你可以使用一種不明顯的方法來改變這些設定：

```
VMMCall Get_Time_Slice_Priority
mov     eax, statusflags
VMMCall Set_Time_Slice_Priority
```

在這段碼中，*statusflags* 是我們所期望的 *VMSTAT_EXCLUSIVE, VMSTAT_BACKGROUND* 和 *VMSTAT_HIGH_PRI_BACK* 的混合。由於改變一個 VM 的背景優先權可能會有預想不到的副作用，只有 VMM 才能瞭解，所以你應該使用上述方法，而不要直接改變旗標位元。

如果 VM 正在處理一個 *Exec_VxD_Int* 或 *Exec_PM_Int* service call，*VMSTAT_VXD_EXEC* 旗標將是 1。如果你已經 hook 了一個保護模式中斷，有可能（但未必）需要在你的中斷處理常式中檢查這個旗標值，以判斷是比較常見的情況（也就是「由保護模式程式發出此一中斷」）或其他情況。

VMSTAT_PAGEABLEV86 旗標表示 VM 中的一個保護模式程式使用了 DPMI 功能 0602h，令 V86 region 的一部份成爲 pageable（可分頁）。除非這個旗標設立，否則呼叫 *_GetV86PageableArray* 並沒有什麼意義，因爲所有的位元都將爲 0。這個旗標及其相關邏輯是如此的曖昧不明，我描述它們時總是深覺遺憾。

稍後我會討論 *VMSTAT_V86INTSLOCKED* 旗標，並和 *Begin_Critical_Section* service 產生關連。相信我，此刻你不需要對此旗標有任何認識。

其餘許多旗標似乎很明顯是給 VMM 及其他系統組件內部使用，你我都沒有必要詳細瞭解它們。

位元與遮罩（Bits and Masks）

DDK 表頭檔有時候不容易閱讀，因爲它們針對一個旗標位元同時定義了一個位元號碼和一個 mask。例如，VMM.H 定義 *VMSTAT_EXCLUSIVE_BIT* 爲 0，然後定義 *VMSTAT_EXCLUSIVE* 爲 `1L << VMSTAT_EXCLUSIVE_BIT`。這兩個定義在 assembly 語

言中比較有意義，你可以這樣測試位元：

```
bt     [ebx+CB_VM_STATUS], VMSTAT_EXCLUSIVE_BIT
jc     is_exclusive
```

或

```
test   [ebx+CB_VM_STATUS], VMSTAT_EXCLUSIVE
jnz    is_exclusive
```

第一個例子使用 BT 指令來測試一串位元中的某個位元，如果位元爲 1 就設立 carry flag。第二個例子使用一般的 TEST 指令來設定狀態碼；它將一個 DWORD 和一個隨後的遮罩（mask）做 AND 運算。兩種方法都可以獲得相同結果。TEST 指令比 BT 快，但是佔用較多的程式空間 -- 如果它後面緊跟的運算元（operand）需要一個 byte 以上的空間的話。

使用一個位元號碼而非使用遮罩來測試一個旗標欄位，會比較容易「測試一個位元然後（同時）改變其狀態」。例如 BTS 會根據某一位元的目前設定狀態而設立 carry flag，然後將該位元設爲 1。反之，BTR 會測試某一位元然後將該位元設爲 0。

找到 VM Control Block

大部份時候 VMM 會令 EBX 指向當班的（current） VM 的 VMCB 結構。如果你使用 assembly 語言，便總是能夠經由 EBX 存取到 current VM 的 VMCB 欄位。如果使用 C 或 C++，下列兩個 services 都可以取得 VMCB 位址：

```
PVMMCB hCurrentVM = Get_Cur_VM_Handle();
PVMMCB hSystemVM  = Get_Sys_VM_Handle();
```

你可以使用表 **9-2** 列出的各種 services 來尋找或列舉 VM control blocks。舉個例子，你或許偶爾需要確認自己正在某特定 VM 中執行一個動作，那麼你可以使用這兩個 test services：

```
if (Test_Cur_VM_Handle(hVM))
   ...                  // is it the current VM ?
if (Test_Sys_VM_Handle(hVM))
   ...                  // is it the system VM ?
```

service	說明
_Allocate_Device_CB_Area	在每一個 VMCB 中保留 device 空間
_Deallocate_Device_CB_Area	在每一個 VMCB 中釋放 device 空間
Get_Cur_VM_Handle	取得 current VMCB 位址，放進 EBX 中
Test_Cur_VM_Handle	測試是否 EBX 指向 current VMCB
Get_Sys_VM_Handle	取得 system VMCB 位址，放進 EBX 中
Test_Sys_VM_Handle	測試是否 EBX 指向 system VMCB
Get_Next_VM_Handle	取得串鏈（chain）中的下一個 VMCB 位址，放進 EBX

表 9-2 可用以找出或列舉 VM control blocks 的一些 services

VM Control Block 之中「爲每一個 Device 所準備的空間」

每一個 VxD 都有機會擴大 VMCB 的空間 -- 只要使用 *_Allocate_Device_CB_Area* 爲自己保留一塊空間即是。當你呼叫這個 service，你可以指定要保留的 bytes 個數，傳回值是一個 offset，稍後你可以加到某個 VMCB 的位址上，定出保留區域。由於這種使用方式，通常你會把 offset 儲存於全域變數內。在 static VxD 中通常我們是在 *Device_Init* 訊息期間爲自己保留空間。

下面例子就是在 *Device_Init* 訊息發生時，在 VMCB 中保留足夠的空間放置一塊私有的 *MYSTUFF* control block。這個例子並顯示如何存取 *MYSTUFF* 結構：

```
typedef struct tagMYSTUFF
   {...} MYSTUFF, *PMYSTUFF
DWORD cboffset;

BOOL __stdcall OnDeviceInit(PVMMCB hVM, DWORD refdata)
   {                    // OnDeviceInit
```

```
    cboffset = _Allocate_Device_CB_Area(sizeof(MYSTUFF), 0);
    ...
    return TRUE;
    }                       // OnDeviceInit

somefunction()
    {                       // somefunction
    PMYSTUFF p = (PMYSTUFF)((DWORD)Get_Cur_VM_Handle() + cboffset);
    ...
    }                       // somefunction
```

此處的 *_Allocate_Device_CB_Area* 在每一個 VMCB 中保留 *sizeof(MYSTUFF)* 個 bytes，並傳回被保留區域的 offset，於是我們將它儲存在 *cboffset* 變數中。*somefunction* 函式片段示範如何把這個 offset 加到 current VMCB 的位址（經由 *Get_Cur_VM_Handle* 獲得），於是取得你內嵌進去的 *MYSTUFF* 區塊。

上個例子對於 static VxD 是適當的。Dynamic VxD 也可以在 VMCB 中爲自己配置空間，但是由於 VMM 從來不會送 *Device_Init* 訊息給 dynamic VxD，配置空間的適當時機因而落在 *Sys_Dynamic_Device_Init* 訊息身上。此外，驅動程式應該確定在卸載之前呼叫 *_Unallocate_Device_CB_Area* 釋放那塊空間：

```
_Deallocate_Device_CB_Area(cboffset, 0);
```

VMM 會將你的私人空間清爲 0，不論是每一個新的 VMCB，或是每一個原已存在的 VMCB。稍後你會看到，這件事情在 thread control block 中並不成立。

Client Register Structure（CRS）

> 譯註：以下所謂暫存器影像（**register images**）是指某一時刻儲存下來的當時暫存器值。

Client register structure（圖 **9-3**）曝露出 VxD 程式設計中最難領悟的一面。VxD 中的程式碼並非一般應用程式所棲息的世界的一部份，它其實是一種 "metaprogram"，監視著 ring3 應用程式，偶爾修正程式路線。還記得嗎？VMM 只有在回應某種中斷時才獲得控

制權，而它會立刻將暫存器值儲存起來。VMM 唯一能夠影響被中斷程式的方法是：修改儲存起來的暫存器影像（register images）。當 VMM 終於要 "redispatch"（重新回到）被中斷程式，會將更改過的暫存器影像放進真正的暫存器中。其影響對於 ring3 環境而言，就好像執行動作被突然轉向到另一個地方、另一種狀態。

當 VMM 的第一層中斷處理常式儲存 general 暫存器和 segment 暫存器，client register structure 將表現出各暫存器的值。這個結構的許多部份吻合「由 V86 模式向 ring0 發出一個中斷」時的堆疊佈局（請看圖 **5-12**）。事實上，CPU 自己的中斷週期（interrupt cycle）會 push 一個 PUSHAD 指令，用以建立起 client register structure。

在 client register structure 中的「代理」暫存器（譯註：圖 **9-3** 中的 "*xxx_Alt_xxx*"）有 segment 暫存器、stack pointer 及 instruction pointer，供其他模式執行時使用。如果 VM 正在執行一個保護模式程式，它的「代理」暫存器將描述出 V86 程式的狀態。相反地，如果 VM 正在執行一個 V86 模式程式，它的「代理」暫存器將描述出保護模式程式的狀態。我在這裡對於「代理」暫存器的說明，應該使你很清楚地瞭解，一個 VM 同時有 V86 模式和保護模式兩個 threads，其中一個可能在某一時刻成爲作用狀態（active）。注意，thread A 的 segment 暫存器對 thread B 並不適用，相同的道理也發生在 stack pointers 和 instruction pointers 身上。

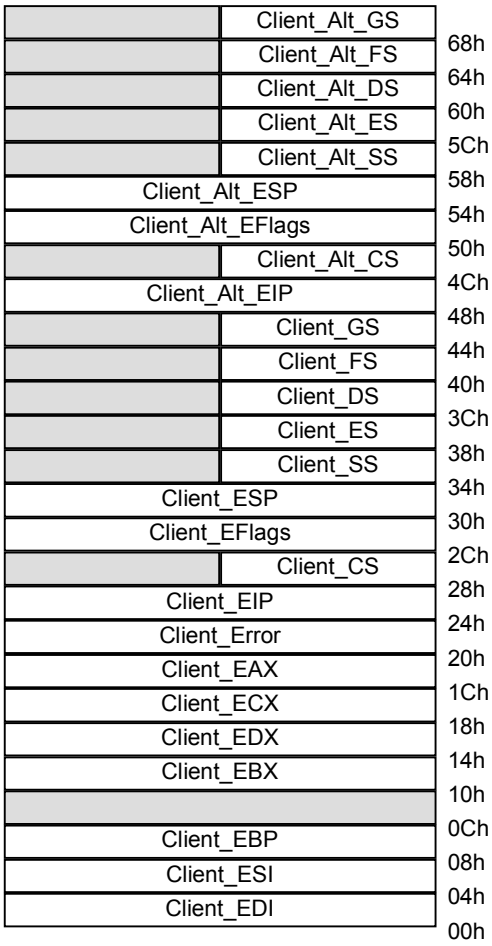


圖 9-3　Client Register Structure（CRS）

存取 Client Registers

一個 assembly VxD 通常會以 EBP 指向 current VM 的暫存器影像（**register images**）。當外界呼叫 VxD，通常 VMM 會讓 EBP 指向暫存器影像。你也可以使用下面這段碼自行完成此事：

```
VMMCall Get_Cur_VM_Handle            ; sets EBX = current VMCB
mov     ebp, [ebx+CB_Client_Pointer]   ; EBP -> client regs
```

從此以後就可以這樣存取屬於此一 VM 之暫存器值（作法是取出 client register structure 中的欄位）：

```
mov     eax, [ebp+Client_ECX]   ; EAX = VM's ECX
movzx   ecx, [ebp+Client_AX]    ; ECX = VM's AX, 0-extended
lar     edx, dword ptr [ebp+Client_DS]
[etc.]
```

在這個例子中，我慎重地將不同的 VM 暫存器值放進 VxD 暫存器中，以突顯一個事實：不管 VxD 暫存器中目前是什麼值，都不會影響「VM 被中斷時暫存器內原本是什麼值」或「VMM 重新回到 VM 時暫存器內是什麼值」。

C 驅動程式通常會使用指標變數來指向那些暫存器影像（register images）。DDK 表頭檔定義了一個 union，名爲 *CLIENT_STRUCT*，其中定義有 *CRS*（針對 32 位元暫存器）、*CWRS*（針對 16 位元暫存器）以及 *CBRS*（針對 8 位元暫存器）。存取暫存器內容因而需要十分冗長的碼，像這樣：

```
CLIENT_STRUCT* pRegs;
DWORD a = pRegs->CRS.Client_EAX;
WORD  b = pRegs->CWRS.Client_BX;
BYTE  c = pRegs->CBRS.Client_CH;
```

譯註：以下就是定義在 VMM.H 中的 *CLIENT_STRUCT* 結構：

```
typedef union tagCLIENT_STRUC {
   struct Client_Reg_Struc         CRS;
   struct Client_Word_Reg_Struc    CWRS;
   struct Client_Byte_Reg_Struc    CBRS;
   } CLIENT_STRUCT;
typedef struct Client_Reg_Struc    CRS;
typedef CRS *PCRS;
```

由於 Microsoft Visual C++ 編譯器允許我們不需指定 members 名稱就定義 union members，所以你可以加上一個定義如下，簡化程式碼：

```
typedef union tagMYCLIENT_STRUCT  {
    struct Client_Reg_Struc;
    struct Client_Word_Reg_Struc;
    struct Client_Byte_Reg_Struc;
} MYCLIENT_STRUC, MYCRS, *PMYCRS;
```

於是程式碼可以變成：

```
PMYCRS pRegs;
DWORD a = pRegs->Client_EAX;
WORD  b = pRegs->Client_BX;
BYTE  c = pRegs->Client_CH;
```

如果使用 VToolsD，或如果寫一組 #define，就像我使用於第 13, 14, 16 章範例程式中的 CRS.H 檔那樣（可以在書附光碟的 \CHAP13\RING0DMA 目錄中找到該檔），你就可以以更經濟的方式來寫碼。你可以總是宣告一個名爲 *pRegs* 的變數，是一個指標，指向 *CLIENT_STRUCT*，然後使用 VToolsD 供應的許多巨集來存取 client register：

```
CLIENT_STRUC* pRegs;
DWORD a = _ClientEAX;
WORD  b = _ClientBX;
BYTE  c = _ClientCH;
```

找出 Client Data 的位址

有時候你必須處理 VM 中的程式所擁有的資料。假設你有一個 API，DS:DX 指向一個字串，下面是一段看似合理其實缺乏經驗的程式碼，用來定址出 VxD 中的字串：

```
char __far *p = MAKELP(_ClientDS, _ClientDX);  // WRONG!
```

或許你在 Win16 程式中寫過類似的碼，但是在 VxD 中行不通。第一，你或許以爲編譯器會將你的 far 指標放到 DS 或 ES 中，再搭配一些 general 暫存器。唔，這麼做可能會使 Windows 95 當掉，因爲之後你所呼叫的任何 VxD services 都會以爲 DS 和 ES 持有 ring0 flat data selector。此外，許多 32 位元 C 編譯器並不讓你使用 far 指標，更別說是 16:16 指標（牽扯到一個 selector 和一個 16 位元 offset）了。某些編譯器可以解釋（接受）16:32 far 指標，但是上面那行碼還是不能正常運作，因爲它有把系統當掉的潛在可能。另一個問題是，你所使用的表頭檔也許並沒有像 Windows.h 那樣定義有 *MAKELP* 巨集（絕大部份原因是沒有人認爲你會使用 far 指標）。

唔，假設你選擇以 assembly 來寫碼，你可以這樣寫嗎：

```
mov  dx, [ebp+Client_DX]   ; WRONG!
mov  gs, [ebp+Client_DS]
[access string via GS:DX]
```

這段碼有時候可以運作，因爲 VxD 之中沒有誰會在乎 GS 的內容。但它仍然不正確，原因是：

- 假設當 VM 呼叫你的驅動程式時，驅動程式正在執行 V86 碼，*Client_DS* 的值因此是一個 paragraph number 而不是一個 selector。當你嘗試把此值放進一個 segment 暫存器，可能會引發一個 GP fault。
- 假設 VM 正在 DOS Extender 的控制下執行 32 位元保護模式程式。此例的 API 應該需要 DS:EDX 來指向字串，但是你只將指標中的 16 位元放入 client 的 EDX 暫存器中。
- 假設 *Client_DS* 中的 selector 是不合理的（或是 null），當你處理它的時候，會引起一個 GP fault，於是 Windows 95 當掉。

欲處理 client 中的字串，正確方法是使用 *Map_Flat* service。在 assembly 語言中，這個 service 使用 AH 來存放 segment 暫存器中的 client structure offset 的編碼，並使用 AL 來存放 offset 暫存器中的 client structure offset 的編碼。它會傳回一個線性位址，其內容是 segment base 加上指定的 offset；也可能傳回 -1，表示 selector 不是合法值。例如：

```
mov     ah, Client_DS
mov     al, Client_DX
VMMCall Map_Flat
cmp     eax, -1
je      error
```

Map_Flat 以一種不尋常的方式使用 *Client_DS* 和 *Client_DX*。AX 暫存器最後會內含 3C14h，3Ch 就是 *Client_DS* 在 client structure 中的偏移位置，14h 則是 *Client_DX* 的偏移位置。

Map_Flat 之所以如此有用，因爲不論 VM 正在執行 V86 碼或保護模式碼，它都能夠正確解釋 segment 暫存器。它並且根據保護模式的 client 是 16 位元程式或 32 位元程式而決定使用 16 位元或 32 位元的 general 暫存器。如果 *Map_Flat* 獲得的是一個合理的 selector，它總是會傳回一個可以直接被 VxD 使用的 32 位元線性位址，不需再設定任何 segment 暫存器。

由於以此方式呼叫 *Map_Flat* 是如此普遍，Microsoft 甚至提供了一個 *Client_Ptr_Flat* 巨集，讓你更輕鬆：

```
Client_Ptr_Flat eax, DS, DX
```

如果使用 Microsoft DDK，你必須自己爲 *Map_Flat* 設計一個 wrapper（外包函式）。我在我的 CRS.H 中定義了一個。VToolsD 也有一個 *Map_Flat* wrapper 和一個 *MAPFLAT* 巨集（爲什麼不把它命名爲 *Client_Ptr_Flat* 呢）。下面是我的巨集使用方法：

```
char *p = (char *)Map_Flat(CLIENT_DS, CLIENT_DX);
```

或

```
char *p = (char *)Client_Ptr_Flat(DS, DX);
```

請注意，即使你懷疑你正在處理一個 32 位元程式，還是請你使用 DX。*Client_Ptr_Flat* 巨集會把你的指示轉換爲一個結構偏移位置，而 DX 和 EDX（以及 DL）有著相同的偏移位置。

與 Win32s 相容的資料參考方式

如果你的 VxD 必須繼續和 Windows 3.1 相容，而且必須能夠和 Win32s 程式共同合作，你就不能夠只使用應用程式傳遞來的線性位址，因爲 Win32s 是以基底位址 FFFF0000h（而不是 0h）來使用 flat selectors。你或許會想呼叫 *Map_Flat* 將一個 Win32 資料位址線性化，不幸的是那會失敗，因爲 VMM 相信 System VM 內含一個 16 位元的 DPMI client，因此它只爲基底位址加上「來自 client DX 暫存器」的 16 位元 offset。

下面是運應之道：

```
VMMCall _SelectorMapFlat, <ebx, <dword ptr [ebp+Client_DS]>, 0>
mov     edx, [ebp+Client_EDX]
lar     ecx, dword ptr [ebp+Client_CS]
test    ecx, 00400000h   // 譯註：判斷 USE16 或 USE32
jnz     @F
and     edx, 0000FFFFh   // 譯註：只取底部 16 個位元（也就是 DX）
@@:
add     eax, edx
```

這個例子利用 *_SelectorMapFlat* 獲得 client data segment 的基底位址，然後根據 client 程式在哪一個 segment（USE16 或 USE32）執行，將基底位址加上 client DX 或 EDX。

我所示範的碼可能對你而言根本沒有用，因爲你可能從來不在 Win32s 程式和 VxDs 之間傳遞線性位址。Microsoft 已經明白表示，希望人們停止在 Win32s 上再做任何努力。

在 Windows 95 中存取 Win32 程式資料，可就簡單多了。一般而言你只需對 *DeviceIoControl* call 有反應就好，你也只需對其中的 input、output、以及 output-length 等等指標（譯註：都是 *DeviceIoControl* 的參數）感興趣就好，它們都是經由 VWin32 傳遞來的。如果你想存取的資料中有指標內嵌於內，VxD 這一端可以使用 Win32 程式所使用的線性位址。但是請小心，務必將「你想要非同步（asynchronously）處理的任何 VM 資料」複製一份，因爲呼叫端的 address context 只有在它是 current process 時才有效力。

譯註：上一句話我用了許多原文術語，其整個意思就是：「當 VxD 的呼叫者目前正獲得 CPU 執行權，VxD 自呼叫端獲得（接受到）的位址才具意義」。一旦發生 context switch，那個位址將暫時失去意義（視非所視矣），因爲整個 memory context 已經改變了。

Thread Control Block（TCB）

Windows 95 引入另一個到處可見的控制區塊：**thread control block**（**TCB**，**圖 9-4**），用以追蹤系統中的每一個 threads。**表 9-3** 顯示此一結構中的 *TCB_Flags* 的個別位元意

義。TCB 對大部份驅動程式而言是唯讀（read only）的資料，這一點和 client register structure 不同，但和 virtual machine control block 相似。

```
struct tcb_s {
   ULONG   TCB_Flags;              // 00 Thread status flags
   ULONG   TCB_Reserved1;          // 04 Used internally by VMM
   ULONG   TCB_Reserved2;          // 08 Used internally by VMM
   ULONG   TCB_Signature;          // 0C 'THCB' (0x42434854)
   ULONG   TCB_ClientPtr;          // 10 register images for this thread
   ULONG   TCB_VMHandle;           // 14 VM that contain this thread
   USHORT  TCB_ThreadId;           // 18 Unique Thread ID
   USHORT  TCB_PMLockOrigSS;       // 1A Original SS before stack lock
   ULONG   TCB_PMLockOrigESP;      // 1C Original ESP before stack lock
   ULONG   TCB_PMLockOrigEIP;      // 20 Original EIP before stack lock
   ULONG   TCB_PMLockStackCount;   // 24 count of stack locks for this thread
   USHORT  TCB_PMLockOrigCS;       // 28 original CS before stack lock
   USHORT  TCB_PMPSPSelector;      // 2A PSP selector
   ULONG   TCB_ThreadType;         // 2C DWORD passed to VMMCreateThread
   USHORT  TCB_pad1;               // 30 padding
   UCHAR   TCB_pad2;               // 32 padding
   UCHAR   TCB_extErrLocus;        // 33 extended error Locus
   USHORT  TCB_extErr;             // 34 extended error Code
   UCHAR   TCB_extErrAction;       // 36 extended error Action
   UCHAR   TCB_extErrClass;        // 37 extended error Class
   ULONG   TCB_extErrPtr;          // 38 extended error pointer
};                                 // 3C
```

圖 9-4　Thread Control Block 的佈局

名稱	Mask	說明
THFLAG_SUSPENDED	00000008	Thread 已被凍結（suspended）
THFLAG_NOT_EXECUTEABLE	00000010	Thread 已被部份摧毀（partially destroyed）
THFLAG_THREAD_CREATION	00000100	Thread 屬於其產生者（一個 process）所有
THFLAG_THREAD_BLOCKED	00000400	Thread 被一個 semaphore 凍結（blocked）
THFLAG_CHARSET_MASK	00030000	預設字元組：

名稱	數值	說明
THFLAG_ANSI	0	ANSI
THFLAG_OEM	1	OEM

THFLAG_UNICODE	2	Unicode
THFLAG_EXTENDED_HANDLE	00040000	Thread 使用 extended file handles
THFLAG_OPEN_AS_IMMOVABLE_FILE	00080000	磁碟離斷（disk fragement）旗標尚未導至檔案被搬移
THFLAG_RING0_THREAD	10000000	Thread 只在 ring0 執行

表 9-3 Thread Control Block 中的旗標位元

習慣上，assembly 驅動程式會把 current TCB 位址放在 EDI 暫存器中。有一組 services 用來尋找並掃描 TCBs，和應用在 VMCBs 上的那一組十分類似，我把它整理於表 **9-4**。

Service	說明
_AllocateThreadDataSlot	爲每一個 TCB 保留 4 bytes 給 device 使用
_FreeThreadDataSlot	釋放 TCB 中的 device 空間
Get_Cur_Thread_Handle	傳回 current TCB 位址（放在 EDI）
Test_Cur_Thread_Handle	測試是否 EDI 指向 current TCB
Get_Sys_Thread_Handle	取得 system thread 的 TCB
Test_Sys_Thread_Handle	測試是否 EDI 指向 system TCB
Validate_Thread_Handle	測試是否 EDI 指向一個合法的 TCB
Get_Initial_Thread_Handle	取得 EBX 所代表之 VM 的 initial thread
Test_Initial_Thread_Handle	測試是否 EDI 指向一個 VM 的 initial thread
Get_Next_Thread_Handle	取得串鏈（chain）中的下一個 TCB 位址，放進 EDI

表 9-4 可用以找出或列舉 thread control blocks 的一些 services

就像你以 *_Allocate_Device_CB_Area* 在 VM control block 中保留給每一個 device 一塊空間一樣，你可以使用 *_AllocateThreadDataSlot* 在每一個 TCB 中配置一個 DWORD 供自己使用：

```
DWORD tcboffset = _AllocateThreadDataSlot();
...
PDWORD p = (PDWORD)((DWORD)Get_Cur_Thread_Handle() + tcboffset);
```

雖然 thread data slots 有點類似給 Win32 程式使用的 thread local storage，但是它們佔用不同的實際記憶體。因此，你不可以將你的 slot offset 交給一個 Win32 程式使用於...譬如說...呃...*TlsGetValue* 上面。同樣道理，你也不可以將 Win32 *TlsAlloc* 獲得的結果當做一個 ring0 thread data slot。

使用 thread data slot 會產生一個重大併發症：VMM 無論如何不會爲那些 slots 設初值。因此當你配置一個 thread data slot，你必須將所有目前存在的以及未來誕生的 TCBs 中的 slots 都加以初始化。此外，由於你很可能使用 slot 來儲存一個指向 heap memory 的指標，所以你必須確定不要引發 memory leak。也就是說要記得在 thread 結束之前釋放該指標。所以如果你打算處理 *Thread_Init* 和 *Destroy_Thread* 這兩個系統控制訊息，你可能需要自己寫一個輔助常式來配置 thread data slots。下一節會示範這個動作。

配置記憶體

Device 驅動程式常常需要爲了各種目的配置小塊記憶體。VMM 提供一個 memory heap 滿足這個需要，並提供兩組 services 用以管理這個 heap。通常你可以使用 *_HeapAllocate* 做爲基本的記憶體配置工具。當你需要許多塊小量記憶體，爲了強化效率並爲了讓中斷處理常式也能獲得記憶體，VMM 另外還提供了一組 linked-list services。

Heap Manager

表 **9-5** 列出用以管理小塊記憶體的 heap manager services。你可以使用 *_HeapAllocate* 來保留一塊空間，稍後再呼叫 *_HeapFree* 釋放之：

```
PVOID p = _HeapAllocate(nbytes, flags);
...
_HeapFree(p, 0);
```

傳回值是一個 flat 指標，指向一塊記憶體，大小至少等同於你所指定的 bytes 數。有一點我想不必說明你一定清楚，那就是千萬不要溢過（overflow）一塊記憶體的邊界，因爲這個 memory manager 並沒有提供此類保護。當你不再需要這塊記憶體，必須將它釋放

（這點和 Windows 程式不同；Windows kernel 會自動釋放 global 記憶體 -- 當其擁有者結束時）。在 VxD 程式設計領域中，沒有誰會幫你追蹤被 VxD 配置的記憶體。

Service	說明
_HeapAllocate	配置一塊記憶體
_HeapFree	釋放一塊記憶體
_HeapGetSize	決定（得知）一塊記憶體的大小
_HeapReAllocate	重新配置一塊記憶體

表 9-5 Heap management services

_HeapAllocate 的 *flags* 參數是 *HEAPZEROINIT* 旗標和另一旗標的組合。前者表示以 0 爲初值，後者指定你所希望的記憶體位置，共有三種可能：

- **HEAPINIT** 只有在 device 初始化時這個旗標才能使用，表示你希望把記憶體放在 initialization data segment 中。VMM 會在初始化結束後自動釋放記憶體，因爲它會刪除含有此塊記憶體的 segment（簡直就是釜底抽薪）。
- **HEAPSWAP** 表示你需要的是一塊 pageable（可分頁的）記憶體。小心使用它，因爲它會導至 deadlock(死結)發生，請參考稍後的 "Pageable Data Areas" 方塊文字。
- **HEAPLOCKEDIFDP**（意思是 "locked if DOS paging"） 表示如果 Windows 95 在 ring0 做 paging I/O，那麼你要的是 pageable 記憶體；如果是由 MS-DOS 或 BIOS 處理 paging I/O，那麼你要的是 locked 記憶體。

可分頁的資料區（Pageable Data Areas）

欲讓 Windows 95 能夠對 VxD code 和 data 做 paging 動作，你必須在配置 data objects 時特別小心。因爲即使 Windows 95 內含對磁碟 I/O 的 ring0 完全支援，end user 還是可能強迫系統經由 MS-DOS 和 BIOS 來做 paging。這種情況下 VMM 會自動鎖住（locks）其他的 pageable code 和 data segment，以避免在執行 VxD code 時重複進入

MS-DOS。然而 VMM 不會自動鎖住動態配置的 data objects。如果一個 VxD 企圖在「MS-DOS 或 BIOS 正在 paging」時存取 pageable data，就會發生 deadlock（死結）。避免這個問題的方法就是在呼叫 *_HeapAllocate* 時使用 *HEAPLOCKEDIFDP* 旗標，以及在呼叫 *_PageAllocate* 時使用 *PAGELOCKEDIFDP* 旗標。

如果你不指定任何一個旗標，*_HeapAllocate* 會配置 page-locked 記憶體。不管哪一種情況，記憶體都是從可共享的系統區域（C0000000h 以上）配置而來。

爲了示範 heap 的使用，讓我假設，你的驅動程式要爲每一個 thread 將某資源虛擬化，方法是儲存各個重要的 per-thread 資料於你所定義的一個 *MYSTUFF* 結構中。實作策略大約是這樣：

- 每次 VMM 產生一個新的 thread，就從自由記憶體中配置一塊 *MYSTUFF* 結構，並把指向「每個 thread 擁有一份的記憶體」的指標儲存在 thread control block（TCB） 的一個 slot 之中。
- 每次 VMM 結束一個 thread，就釋放你先前配置的記憶體。

第一個步驟是配置一個 thread data slot 供自己使用。如果你的驅動程式是一個 static VxD，你應該在處理 *Device_Init* 時配置這個 slot。如果你的驅動程式是一個 dynamic VxD，程式碼會有趣一些，因爲你必須在被卸載（unload）時釋放那個 data slot。此外，就如同前一節所說，VMM 並不設定 data slots 的初值。你的初始化函式因而應該看起來像這樣（取自書附光碟的 \CHAP09\HEAPMANAGEMENT-DDK 目錄）：

```
#0001  DWORD tcboffset;
#0002
#0003  BOOL OnSysDynamicDeviceInit()
#0004     {                        // OnSysDynamicDeviceInit
#0005     PTCB first, thread;
#0006     tcboffset = AllocateSlot();
#0007     if (!tcboffset)
#0008           return FALSE;
#0009
#0010     first = thread = Get_Sys_Thread_Handle();
#0011     do  {   // allocate stuff for each thread
#0012        if (!(*(PMYSTUFF*)((DWORD) thread + tcboffset) =
```

```
#0013            _HeapAllocate(sizeof(MYSTUFF), HEAPZEROINIT)))
#0014            {                        // can't allocate
#0015            OnSysDynamicDeviceExit();  // cleanup
#0016            return FALSE;             // fail device load
#0017            }   // can't allocate
#0018         thread = Get_Next_Thread_Handle(thread);
#0019         }   // allocate stuff for each thread
#0020      while (thread != first);
#0021      return TRUE;
#0022      }                          // OnSysDynamicDeviceInit

#0001  DWORD AllocateSlot()
#0002      {                       // AllocateSlot
#0003      PTCB first, thread;
#0004      DWORD offset = _AllocateThreadDataSlot();
#0005      if (!offset)
#0006         return 0;
#0007      first = thread = Get_Sys_Thread_Handle();
#0008      do {   // initialize slot in each thread
#0009         *(PDWORD) ((DWORD) thread + offset) = 0;
#0010         thread = Get_Next_Thread_Handle(thread);
#0011         }   // initialize slot in each thread
#0012      while (thread != first);
#0013      return offset;
#0014      }                       // AllocateSlot
```

其中 *OnSysDynamicDeviceInit* 負責處理 *Sys_Dynamic_Device_Init* 訊息，用以初始化一個 dynamic VxD。函式中使用了 *AllocateSlot* 輔助函式來獲得一個 thread data slot 的偏移位置，然後以迴路方式走訪目前作用中的所有 threads，爲每個 thread 配置一個 *MYSTUFF* 結構。迴路是從 system thread 開始，你可以呼叫 *Get_Sys_Thread_Handle* 獲得其 handle 值。由於 thread control blocks 形成一個圓形串列（ring），這個迴路將一次次地呼叫 *Get_Next_Thread_Handle* 直到又回返 system thread 爲止。*AllocateSlot* 以 *_AllocateThreadDataSlot* 來獲得一個 data slot，然後執行迴路，走訪所有的 thread control blocks，將該 data slot 初始化爲 0。範例中所呈現的作法是組織過的，如果我們配置一個 thread data slot 成功，那麼每一個 TCB 中的它內容若不是 0 就是一個 *MYSTUFF* 結構位址。你可以依賴此一事實來幫助證明你的 VxD 的正確性。

第二個步驟是，每當一個新的 thread 產生出來就配置一個 *MYSTUFF* 結構；每當一個 thread 結束就釋放其 *MYSTUFF* 結構。你可以這樣完成任務：

```
#0001  BOOL OnCreateThread(PTCB thread)
#0002     {             // OnCreateThread
#0003     PMYSTUFF* pcell = (PMYSTUFF*) ((DWORD) thread + tcboffset);
#0004     PMYSTUFF pstuff = (PMYSTUFF) _HeapAllocate(sizeof(MYSTUFF),
#0005                                          HEAPZEROINIT);
#0006     if (!pstuff)
#0007        return FALSE;    // fail thread creation
#0008     *pcell = pstuff;
#0009     return TRUE;
#0010     }             // OnCreateThread
#0011
#0012  VOID OnDestroyThread(PTCB thread)
#0013     {             // OnDestroyThread
#0014     PMYSTUFF* pcell = (PMYSTUFF*)((DWORD) thread + tcboffset);
#0015     PMYSTUFF pstuff = *pcell;
#0016     *pcell = NULL;
#0017     if (pstuff)
#0018        _HeapFree(pstuff, 0);
#0019     }             // OnDestroyThread
```

其中 *OnCreateThread* 用來處理 *Create_Thread* 訊息，此訊息通知大家說有一個 thread 被產生出來了。請設定你的 thread data slot 內容，使它指向 thread 自己的 *MYSTUFF* 結構。如果你不能夠爲這塊空間配置記憶體，就應該傳回 FALSE 表示 thread 產生失敗！在上市前的最後一個開發版本中，你應該使用 SHELL device service 來產生一個訊息，告訴使用者爲什麼你不允許新的 thread 誕生。

OnDestroyThread 用來處理 *Destroy_Thread* 訊息，此訊息通知大家說 VMM 打算摧毀一個已結束的 thread。你應該在這裡釋放 *MYSTUFF* 記憶體。我要雞婆地建議你，在呼叫 *_HeapFree* 之前先把你的 thread data slot 設爲 NULL。這麼一來如果 *_HeapFree* 遭遇了某些問題，你的 thread data slot 會是 NULL，而你的驅動程式絕不會成爲某個 thread 的某個禍因[1]。

[1] 在釋放記憶體之前，先將指標設爲 NULL，是一種避免錯誤的技術，用以避免無窮迴路的記憶體錯誤。假設 *_HeapFree* 中出了某些錯誤，由於我們的 thread 尚未完全結束，很有可能 *Destroy_Thread* 訊息處理常式會再被呼叫一次。甚至可能是遞迴情況（recursively）。於是又再觸發相同的失敗 -- 如果我們沒有預先留意的話！

你還需要在你的驅動程式卸載（unloads）時清除所有的 *MYSTUFF* 記憶體。下面是個實例，很簡單：

```
#0001  BOOL OnSysDynamicDeviceExit()
#0002      {       // OnSysDynamicDeviceExit
#0003      PTCB first, thread;
#0004      if (!tcboffset)
#0005         return TRUE;   // shouldn't be possible
#0006      first = thread = Get_Sys_Thread_Handle();
#0007      do  {   // cleanup existing threads
#0008         OnDestroyThread(thread);
#0009         thread = Get_Next_Thread_Handle(thread);
#0010         }   // cleanup existing threads
#0011      while (thread != first);
#0012      _FreeThreadDataSlot(tcboffset);
#0013      return TRUE;   // i.e., okay to unload
#0014      }       // OnSysDynamicDeviceExit
```

這段碼依賴一個事實：*OnDestroyThread* 釋放 thread 的 *MYSTUFF* 記憶體並將你的驅動程式的 thread data slot 設爲 NULL。這個函式也展示了「走訪所有 threads」的另一種迴路形式，和前兩種形式都不相同。

這些都是 Thread-Safe（對 thread 而言安全的）嗎？　如果你夠仔細也夠努力，你可以在前面的碼中看到一個明顯的破綻。如果一個新的 thread 誕生或結束，三種迴路形式中的任何一種都會被打斷而跳出。然而就像稍後要討論的，每一個 system control messages 的處理常式都有 Windows 的 critical section 保護著，因此當顯示於此的任何一個函式正在進行時，絕不可能有某個 thread 啓動或停止，所以上面的碼面對 threads 畢竟還是安全的！

與串列（linked-list）有關的 services

與串列（linked-list）有關的這一組 services（請看表 **9-6**），用來管理固定大小的記憶體區塊，這些區塊可能（也可能不）串接在一起。使用這些 APIs 的第一個步驟是首先呼叫 *List_Create* 產生出一個新的 list object。當你最後完成任務時，請呼叫 *List_Destroy* 將串列清除掉：

```
VMMLIST list = List_Create(flags, nodesize);
...
List_Destroy(list);
```

Service	說明
List_Allocate	配置串列中的一個新元素（new element）
List_Attach	爲串列加上一個新元素（放在串列頭）
List_Attach_Tail	爲串列加上一個新元素（放在串列尾）
List_Create	產生一個新的 list object
List_Deallocate	釋放串列中的一個新元素（new element）
List_Destroy	摧毀一個新的 list object
List_Get_First	取得串列中的第一個元素（first element）
List_Get_Next	取得串列中的下一個元素（next element）
List_Insert	在串列中插入一個新元素（new element）
List_Remove	在串列中移除一個新元素（new element）
List_Remove_First	移除串列中的第一個元素（first element）

表 9-6 與串列（linked-list）有關的 services

List_Create 會配置一塊空間，此空間可以滿足稍後的記憶體需求。*nodesize* 參數用來指定每一個元素的固定大小。*flags* 參數可以控制稍後這個 list object 的行爲。*flags* 參數必須是 0 或下列常數的組合：

- **LF_USE_HEAP** 表示這個 list 應該被 *_HeapAllocate* 從 system heap 中配置而來。以此選項來配置 list，沒有什麼特性需要說明。
- **LF_ASYNC** 表示你可以在中斷處理過程中配置串列元素。以此選項產生一個 list object，是唯一能夠讓你的中斷處理常式動態配置記憶體的方法。
- **LF_ALLOC_ERROR** 通知 list manager 說，後續的元素配置動作如果失敗，請以「設立 carry flag 並 return」來表示，而不要直接讓 Windows 95 當掉！

■ **LF_SWAP** 表示你希望串列元素從 swappable heap 中配置得來。

一旦你產生了一個 list object，你就可以使用 *List_Allocate* 和 *List_Deallocate* 來獲得或釋放固定大小（先前已指定過）的串列元素了：

```
VMMLISTNODE node = List_Allocate(list);
...
List_Deallocate(list, node);
```

如果你要，你可以單只使用我所描述的這些 list management services，就完成一個簡單而快速的記憶體配置機制。如果串列是以 *LF_ASYNC* 產生出來，中斷處理常式或其他 asynchronous（非同步的）service 也可以從其中配置記憶體。然而如果有需要，你也可以使用**表 9-6**所列的其他 services 以組成一個雙向串列。例如一個 FIFO（譯註：First In First Out，先進先出）queue 的 services 可以因爲「中斷處理常式將一個元素加到串列尾端」而完成：

```
VMMLIST stufflist;   // created with LF_Async
...
void SomeInterruptHandler()
    {
    PMYSTUFF p = (PMYSTUFF)List_Allocate(stufflist);
    List_Attach_Tail(stufflist, p);
    }
```

在這個例子中，我假設 *SomeInterruptHandler* 函式處於一個 locked code segment 之中，並且是在處理硬體中斷時被呼叫，當時的中斷已經被 disables 了（中斷之所以會被 disabled，因爲你是面對一個 *LF_ASYNC* 串列呼叫 list manager services）。*List_Allocate* 從一大塊 page-locked 記憶體中配置一個新元素，大小如同 *MYSTUFF*。如果你在產生這個串列（list）時曾經指定 *LF_ALLOC_ERROR*，assembly 語言的 *List_Allocate* service 會在沒辦法成功配置一塊新元素的時候，令 carry flag 設立起來，然後 return。但要是你沒指定 *LF_ALLOC_ERROR*，*List_Allocate* 會在配置失敗時直接導至 Windows 95 當機。如果有一個 *List_Allocate* 的 C wrapper（外包函式），那麼當它配置記憶體失敗時應該傳回 NULL。

List_Allocate 所做的無非是配置記憶體。至於記憶體做何用途，由你決定。在上述程式片段中，我使用 *List_Attach_Tail* 把新元素加到串列尾端。下面是個 callback 函式，走訪整個串列（list）：

```
void SomeCallback()
    {
    PMYSTUFF p;
    while ((p = (PMYSTUFF)List_Remove_First(stufflist)))
        {
        ...
        List_Deallocate(stufflist, p);
        }
    }
```

迴路的每一次迭代過程中，我們使用 *List_Remove_First* 來解除串列的第一個元素。然後在某一個適當地點呼叫 *List_Deallocate*，將元素的記憶體歸還。

處理 Events

和其他作業系統一樣，Windows 95 提供一種方法，讓 VxD 能夠要求 VMM 稍後回呼（callback）它，以便執行一些不方便當下就執行的工作。在 Windows 之中用來做這種延遲工作的東西就是所謂的 events。你可能必須爲了兩種常見的狀態，將一個 event callback 納入排程（schedule）之中：

- 如果你正對某個 VM（或某個 thread）執行一個動作，有可能你需要在另一個 VM（或 thread）的 context 中做一些事情。你不能夠直接觸發 task switch 或 thread switch，然而你可以將一個 event callback 納入排程（schedule），於是當那個 VM（或 thread）下次作用起來（譯註：也就是獲得 CPU 時間），你的 event callback 就會被執行。
- 如果你正在處理硬體中斷，你的工作受到兩個限制：你的 code 必須放在 locked pages，而且它只能夠呼叫那些被設計爲 asynchronous（非同步）的 services。所謂的 **asynchronous service**s 是指可以安全地在中斷處理期間被呼叫的 services。這種 services 很少，請看**表 9-7**。於是，爲了完整處理中斷，

你可以先將一個 callback 函式納入排程，等到 VMM 到達穩定狀態了，不再受什麼束縛了，才執行你的 callback 函式。

Event 的排程作法是，由你呼叫一個 VMM service，將 event callback 納入排程。Events 有數種（表 **9-8**），你可以挑選你認爲對你的工作比較適合的一種。根據你所呼叫的 scheduling API，VMM 有可能立即處理這個 event。是的，如果你呼叫 *Call_xxx_Event*，就有可能立即處理。VMM 也可能將 event 放到 queue 裡，稍後再處理 -- 如果你呼叫 *Schedule_xxx_Event*，就是這種情況；甚至即使你呼叫的是 *Call_xxx_Event*，也有可能是這種情況。

無論如何，VMM 稍後會呼叫你的 callback 函式，此函式完成其任務後回返至 VMM。請注意，event 具有「僅只一次」的觀念，也就是說一旦 VMM 呼叫了 event 函式，它便會清除先前你所產生的 event object。你當然也可以在 VMM 呼叫 event 函式之前，自己先行取消這個 event object。排程和取消所使用的 API 函式是成雙成對的，你必須使用某個特定的函式才能取消某種特別的 event。

當 VMM 正要從 ring0 移轉控制權到 ring3，它會處理 events。這種移轉發生在最外部中斷（outermost interrupt）的處理常式回返至 VMM 時。Event 的處理也發生於任何 VxD 呼叫 *Resume_Exec* 或 *Exec_Int* 時。*Resume_Exec* 有點像 Windows 的 *PeekMessage* 函式（帶有 *PM_NOYIELD* 旗標），允許 event 函式執行起來，而不需要先有一個 context switch！

所有的 Event services	Get_Next_Thread_Handle
Linked-list memory allocation ♦	Get_Next_VM_Handle
大部份的 VPICD services ♠	Get_Sys_Thread_Handle
各種除錯用的 services ♥	Get_Sys_VM_Handle
Call_When_Idle	Get_System_Time
Crash_Cur_VM	_GetThreadExecTime
Disable_Touch_1st_Meg	Get_Time_Slice_Info
DOSMGR_Get_DOS_Crit_Status	Get_VM_Exec_Time
Enable_Touch_1st_Meg	Get_VMM_Reenter_Count
Fatal_Error_Handler	Get_VMM_Version
Fatal_Memory_Error	GetSetDetailedVMError
Get_Crit_Status_No_Block ✦	_lmemcpy
Get_Crit_Status_Thread	_lstrcpyn
Get_Cur_Thread_Handle	_lstrlen
Get_Cur_VM_Handle	Set_Async_Time_Out
Get_Execution_Focus	Set_Global_Time_Out
Get_Initial_Thread_Handle	Set_Thread_Time_Out
Get_Last_Updated_System_Time	Set_VM_Time_Out
_GetLastUpdatedThreadExecTime	_SHELL_CallAtAppyTime ♣
Get_Last_Updated_VM_Exec_Time	_SHELL_QueryAppyTimeAvailable
SHELL_Update_User_Activity	Test_Sys_Thread_Handle
Signal_Semaphore	Test_Sys_VM_Handle
Signal_Semaphore_No_Switch	Update_System_Clock
Test_Cur_VM_Handle	Validate_Thread_Handle
Test_Initial_Thread_Handle	Validate_VM_Handle

♦ 呼叫 *List_Create* 時請指定 *LF_Async*。
♠ *VPICD_Vritualize_IRQ* 並不是非同步（asynchronous）。
♥ Windows 95 DDK 線上說明文件所說的各種 *_Debug_* 和 *Debug_* services。*_Debug_Flags_Service* 特別有用，可以在執行環境中提供 assertion 功能。
✦ *Get_Crit_Status_No_Block* 是 *Get_Crit_Status* 的非同步兄弟。
♣ 注意，*_SHELL_CancelAppyTimeEvent* 並不是非同步（asynchronous）。

表 9-7　Asynchronous（非同步）services

Event 種類	Services
Global	Call_Global_Event, Cancel_Global_Event, Schedule_Global_Event
Virtual machine	Call_VM_Event, Cancel_VM_Event, Schedule_VM_Event
Thread	Cancel_Thread_Event, Schedule_Thread_Event
Priority VM	Call_Priority_VM_Event, Cancel_Priority_VM_Event
Restricted	Call_Restricted_Event, Cancel_Restricted_Event

表 9-8 Events 及其 services 的種類

Events 的種類

Windows 有數種不同的 events：

- **Global events** 其 callback 函式可以在任何 VM context 或任何 thread context 中執行。硬體中斷常式可以利用 global event 來完成其工作，不需等待任何 task switch 或 thread switch。
- **VM events** 其 callback 函式必須在特定的 VM context 中才能執行。通常你可以利用 VM events 做爲「影響一個 MS-DOS 視窗或所有 Windows 應用程式」的動作的一部份。以 VM event 來完成工作的極佳例子就是「進入 System VM 然後 "post" 一個訊息」。不過更好的例子是使用一個 priority VM event（稍後描述），因爲它在執行之前還提供了一些條件！
- **Thread events** VMM 在特定的 thread context 中才能爲此種 events 服務。當你（VxD）和一個 Win32 程式合作時，一定會常常用到這種 events。
- **Priority VM events** 這是 VM events 的一種，不過當 callback 發生時，它另外還提供一組選擇性的限制條件，而且當 callback 執行時，它會引起優先權提昇（priority boost）。可有可無的限制條件包括：時間終了週期（也就是自動發射此一 event 的期限）、此 VM 的所有 threads 是否都令虛擬中斷 enabled 起來、無任何 VM 擁有唯一的 Windows critical section、指定執行優先權時無其他任何 VM 正在執行一個耗時的運算、VM 不處於「正在處理一個模擬的硬體中斷」過程之中。在 Windows 3.1，這種 event 最常被使用，因爲用了它你就可以指定條件並提昇優先權。

- **Restricted events** 這是 Windows 95 新添的 events，提供對 callbacks 的時間限制 -- 比 priority VM events 的限制更多。你可以指定一個受限的（restricted）event 在某個 thread context 中執行（藉由產生一個 priority thread event），並限制 critical section 和虛擬中斷的狀態能夠符合該 thread 的情況。你也可以指定 callback 不要在 VM 正執行 V86 程式時發生。最後，你也可以產生一個此種型態的 global events。還有一些限制條件是 Microsoft 所宣稱，並明載於 DDK 中以求完備，但我們從不需要用到的。大部份的我們現在都已經改用這種受限的（restricted）events，因爲它可以解決一些令人煩惱的死結（deadlock）問題。

當 VMM 開始處理 events，它首先處理 global events，然後是 current VM（也就是有最高優先權之 VM）的 VM events，然後是 current thread 的 thread events。VMM 處理 events 的次序並無其他指定方式。請注意，一個 VM event 或 thread event 的處理常式可能會因爲讓其他 VM 或 thread 開始當班而改變了執行優先權，或是釋放了被凍結的 thread，VMM 於是會把注意力放在新的 VM 或 thread 的任何 pending events 身上。

撰寫 Event Callbacks 函式

最初產生一個 event 時，你必須指定 event callback 函式的位址。例如：

```
BeginProc hwIntProc, locked
...
mov  esi, offset32 OnMyEvent
VMMCall Schedule_Global_Event
...
EndProc hwIntProc
```

VMM 會以表 **9-9** 的暫存器內容來呼叫你的 callback 函式。Direction flag 會被清除（C 程式向來如此），CPU 中斷會被 enabled 起來。你的 callback 函式必須保持 EBP 和 ESP 以及 interrupt flag 和 direction flag 不變。和系統的其他部份一樣，你不能夠改變 DS, ES（或 SS）暫存器，但是必要的時候可以使用（並恢復）FS 或 GS 暫存器。

暫存器	內容
EBX	current VMCB 的位址
EDX	*Call_xxx_Event* 或 *Schedule_xx_Event* 的 reference 參數內容
EDI	current thread control block（TCB）的位址
EBP	current thread 的 client register structure。

表 9-9　進入一個 event callback 函式時的暫存器內容

一般而言，event callback 函式可以使用任何 VxD services。這不就是你使用 event 機制的原因嗎！但是有一些限制你必須遵守。由於 global event callback 函式可以在任何 VM context 和任何 thread context 中被執行，所以隨著參數而來的 VMCB 和 TCB 位址，可能沒有什麼用。甚且一個 VM 或 thread callback 並不見得對此 VM 或 thread 同步（synchronized），於是 client 的暫存器內容變得不可預期！Callback 函式因此不應該改變 client 的暫存器值，除非它先儲存起來並於最後記得恢復。第 10 章介紹這種程式方法，此法最適用於「在 event callback 函式中執行 ring3 碼」。

Event callback 函式的另一個主要約束在於避免死結（deadlock）的發生。這些問題之所以發生，是因爲如果你在呼叫一個 synchronization service 時指定 *BLOCK_SVC_INTS* 旗標，thread 可能會在某個資源（譯註）上被凍結住，而 event callbacks 還是可能在該 thread 中發生。以下三點解釋死結發生的環境，以及如何避凶趨吉：

> 譯註：這裡所謂的資源，當然不是 Windows 程式設計中 .RC 檔內的資源（選單、對話盒、字串、圖示、加速鍵表、字形...），而是指 memory、printer、file...等等屬於作業系統層次的資源。

- 一般而言，event callback 函式不應該被一個 semaphore 或其他同步控制元件（critical section 或 event 或 mutex 或...）凍結（block）。它可能在擁有相同資源的 thread context 中被呼叫，於是帶來死結的可能性。如果沒有死結問題發生，凍結（blocking）也可能對平順的多工效應帶來不利影響，因爲處理 event 的那個 thread 可能擁有其他 thread 所需的資源。

- 死結可能在存取 registry 時發生。假設有一個 event callback 函式在某個 thread 中獲得控制權，但是同一 thread 的 ring3 程式碼正在處理 registry，如果 event callback 函式也企圖存取 registry，就會把這個 thread 帶到死結狀態。避免之道就是只在 restricted event callbacks 函式中（並且指定 *PEF_Wait_Not_Nested_Exec* 旗標）才存取 registry。
- 如果 MS-DOS 被用於 paging，event 函式可能會因爲存取一塊動態配置而來的 pageable 資料，或是因爲呼叫一個記憶體配置函式，不小心引起一個死結問題。如果你的 thread 已經被凍結(blocked)並等待 paging，但它又必須 paging 某些資料，死結問題就會發生。爲了避免問題，請你在呼叫 *_HeapAllocate* 時總是指定 *HEAPLOCKEDIFDP* 旗標，並且在呼叫 *_PageAllocate* 時總是指定 *PAGELOCKEDIFDP* 旗標，而不要配置總是可分頁（pageable）的資料空間。此外，請避免在 event callback 函式中呼叫記憶體配置函式。如果你不確定某個資料指標的源頭，你可以呼叫 *PAGESWAP_Test_IO_Valid*，並在其結果顯示「MS-DOS paging 並未執行」時，重新將你的 event 排程。或者你可以將一個有著 *PEF_Wait_Not_Crit* 旗標的 event 加入排程行列。注意，這個限制只適用於動態配置所得的資料，因爲如果 MS-DOS 或 BIOS 被用於 paging，VMM 會自動將 VxD 的 code segment 和 data segment 加以 page-locks。

取消 Events

任何一個 service 如果產生 event，都會經由 ESI 暫存器傳回 event handle。在 VMM 呼叫過其 callback 函式之後，event handle 就會變成 NULL。如果你要保留「在 callback 發生之前取消 event」的權利，你可以把 handle 儲存在某個地方：

```
hEvent dd 0
...
mov     esi, offset32 OnSomeEvent
VMMCall Call_xxx_Event
mov     hEvent, esi
```

稍後若要取消這個 handle，你可以呼叫 *Cancel_xxx_Event* service。你必須小心行事，避免被強制插斷（preemption）。事實上，event callback 和主線程式碼（取消 event 者）必須合作才能讓這個動作絕對安全：

Mainline Code（主線程式碼，取消 event 者）

```
xor     esi, esi
xchg    esi, hEvent       // 譯註：取出 hEvent，放到 esi
VMMCall Cancel_xxx_Event
```

Callback 函式

```
BeginProc OnSomeEvent
xor     eax, eax
xchg    eax, hEvent       // 譯註：取出 hEvent，放到 eax
test    eax, eax          // 譯註：hEvent 是否爲 null？
jnz     @F
ret                       // 譯註：若 hEvent 是 null 則回返
@@:
...
EndProc OnSomeEvent
```

看這段程式之前，你必須知道，cancellation 函式（譯註：*Cancel_xxx_Event*）可以容忍一個 NULL handle 而沒有任何怨言。Event callback 函式使用最原始的 XCHG 指令來取出 *hEvent* 的值，並同時將 *hEvent* 設爲 NULL。如果 *hEvent* 原本就是 NULL，意味主線程式已經取消了這個 event，於是 callback 函式兩手空空地回返，沒有做任何事情。如果主線程式進行到 XCHG 指令並發現 *hEvent* 不是 NULL，意味 callback 函式尚未被執行過，於是會取消這個 event。

VToolsD 中如何處理 Events

VToolsD C++ class library 提供了數個類別（classes），可以在高階語言中簡化 events 的產生和處理。例如，只要從 *VGlobalEvent* 類別衍生出你自己的類別，便可以將一個 global event 納入排程之中：

```
class MyEvent : public VGlobalEvent
{
public:
   MyEvent(PVOID refData = 0) : VGlobalEvent(refData){}
   virtual void handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs, PVOID refData);
   BOOL CancelSafely();
};
```

VGlobalEvent 基礎類別以 *Call_Global_Event* 來呼叫其成員函式，並使用 *Schedule_Global_Event* 將其成員函式納入排程。其作法都是將目標瞄準一個 assembly thunk（譯註：所謂 thunk 就是一小段程式碼，通常做爲轉換層使用），這個 thunk 內部才呼叫你的 event 處理函式。一個 event object 會在呼叫其處理函式之後自動解構。因此，要在程式中扯進一個 event，就不再是太痛苦的事了：

```
// mainline code
(new MyEvent)->call();   // schedule or call event
...
void MyEvent::handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs, PVOID refData)
   {
   ... // operations not requiring you to delete this
   }
```

由於已經衍生出自己的類別，所以你可以自由產生你的成員變數，儘情地做任何 C++ class 允許你做的美好事情。

Vireo event classes 也內含 *cancel* 成員函式。你並不需要使用它們，因爲它們並沒有使用 "thread-safe" 碼來決定究竟要取消 event 還是要繼續其 callback。如果你希望能夠取消 event，就必須自己寫一段適當的 assembly 碼。而由於 *cancel* 函式很不幸地並不是一個 virtual 函式，所以你必須控制所有可能取消 events 的地方，才能做出一個屬於你自己的 "thread-safe" 函式。下面是個實例（原始碼可從書附光碟的 \CHAP09\EVENTHANDLING-VTOOLSD 目錄中獲得）：

表頭檔

```
class MyEvent : public VGlobalEvent
{
...
public:
   BOOL CancelSafely();
};   // 譯註：原文書這裡有誤，少了一個 ;
```

Mainline Code（主線程式碼）

```
MyEvent* pEvent = new MyEvent;
pEvent->call();
```

```
...
pEvent->CancelSafely();
```

Callback 函式

```
#0001  BOOL MyEvent::CancelSafely()
#0002    {             // MyEvent::CancelSafely（譯註：原文書這裡有誤，少了 //
#0003    _asm  mov  edx, this
#0004    _asm  xor  eax, eax   // 譯註：原文書這裡使用 esi，我以碟片中的原始碼爲準
#0005    _asm  xchg eax, [edx]VEvent.m_handle   // 譯註：原文書這裡使用 esi
#0006    #ifdef Cancel_Global_Event
#0007       #undef Cancel_Global_Event
#0008    #endif
#0009    VMMcall(Cancel_Global_Event);
#0010    BOOL result;
#0011    _asm  mov result, eax
#0012    if (result)
#0013       delete this;
#0014    return result != 0; // TRUE means actually cancelled
#0015    }                // MyEvent::CancelSafely
#0016
#0017  void MyEvent::handler(VMHANDLE hVM, CLIENT_STRUCT* pRegs,
#0018    PVOID refData)
#0019    {             // MyEvent::handler
#0020    _asm  mov  edx, this
#0021    _asm  xor  eax, eax
#0022    _asm  xchg eax, [edx]VEvent.m_handle
#0023    BOOL not_cancelled;
#0024    _asm  mov not_cancelled, eax
#0025    if (!not_cancelled)
#0026       return;      // event was cancelled, so abort callback
#0027       ...
#0028    }             // MyEvent::handler
```

執行動作的同步控制（Synchronizing Execution）

VMM 可以在 Windows 95 執行的任何時刻岔斷（preempt）一個 VxD。因此，VxD 常常需要以同步元件（synchronization preemption）保護那些共享資料（sharable data object）。本節討論「VxD 何時可能因爲被插斷而喪失控制權」，然後再描述一個 VxD 如何將其執行事實與其他 VxD 同步化，以及與它自己的其他 instances（在其他 threads 中）同步化。

VxD 之所以這麼輕易就被岔斷（preempt），原因之一與 paging 有關。如果 VxD 引起一個 page fault，current thread 可能會停下來（block），等待 paging subsystem 取來一個 page。另外，如果 VxD 呼叫一個 nonasynchronous service（也就是一個並非設計用來在「其他 services 被中斷」時被使用的 service ），也會導至岔斷（preemption）的發生，甚至即使 VxD 碼是 page-locked。由於不論硬體中斷是否被 enabled 或被 disabled，page faults 都可能發生，所以將一個 CLI 指令放在一段敏感程式碼的前面，並不足以使你獲得保護。

某些情況下，你可能預期自己會被岔斷（preempted）。如果你的 VxD 做了以下事情，它就可能被岔斷：

- 調整另一個 thread 的執行優先權。這可能會引起 primary scheduler 立刻將控制權給予另一個 thread。
- 配置記憶體而可能帶來 paging I/O 或置換檔（swap file）的改變。這種情況也可能觸發岔斷。
- 呼叫 *Resume_Exec* 會使得 queued events 被處理，並可能因爲許多理由而產生一個 thread switch。
- 最明顯的情況：呼叫一個 synchronization services（本節稍後介紹）會凍結住（block）呼叫端的 thread。

一旦 VxD 通過了處理硬體中斷的最初階段，它通常會在中斷 enabled 的情況下執行。然而硬體中斷如果岔斷（preempt）一個正在執行的 VxD，並不會引發 context switch，因爲硬體中斷處理常式的規則規定，它們（中斷處理常式）必須在 locked code 中執行、使用 locked data、並且不能呼叫任何 nonasynchronous services。猶有進者，當硬體中斷處理常式完成其動作並回返 VMM 時，VMM 重新執行起先前被中斷的那個 VxD。因此，一個硬體中斷最可能發生的結果是：當中斷處理常式正起而服務，VxD 會暫時失去其控制權。

凍結（suspending）及恢復（Resuming）- 部虛擬機器

在 Windows 3.0 環境中，唯一的岔斷式多工（preemptive multitasking）發生於 VMs 之間。那時候的 VMM 只提供初級而發育不全的方法來令 VM 同步化：你可以凍結（suspend）一部 VM，使它沒有資格被排程（scheduling），稍後你又可以將它喚醒。不同的 VxD 撰寫者所寫出來的碼可以自行決定是否要凍結一部 VM。Windows 3.1 增加了對 semaphore objects 的支援，Windows 95 增加了更豐富的同步元素（synchronization primitives），稍後我就要開始討論它們。然而，最初那種凍結和喚醒（suspending and resuming）的粗糙機制仍然存在於 Windows 95 之中 -- 即使已經沒有什麼特別理由需要它們了。

有時候 VMM 必須凍結某個 VM，阻止它在當某些 task 發生時繼續執行下去。例如當使用者做一個 clipboard（剪貼簿）拷貝動作時，視窗化的 MS-DOS VM 會被凍結住。如果不是這樣，螢幕畫面可能會在動作過程中改變，造成使用者的挫折感。這裡我所說的是凍結（suspension）的一個不同用法，凍結的確是「將 threads 同步化」的一種方法。

由於你可能會使用老舊的 VxDs，而那些 VxDs 使用傳統方法來同步控制 tasks，所以你應該瞭解那些傳統方法（services）做些什麼事情。*Suspend_VM* service 會累加某一指定之 VM 的虛懸計數器：

```
BOOL okay = Suspend_VM(PVMMCB hVM);
```

傳回值表示此 VM 是否已經成功地被凍結了。VMM 不允許你凍結 System VM 或 critical section 的擁有者。如果 VM 並非已被凍結，VMM 會在 *Suspend_VM* 回返之前送出一個 *VM_Suspend* 系統控制訊息給所有 devices。爲了回應此訊息，device driver 可能 unlock 某些資源（例如 V86 記憶體），這些資源係爲 VM 而預備。

一個被凍結的 VM 不會獲得 CPU 時間。此外，一個被凍結的 VM 的 V86 記憶體可能會被置換出去（swapped out），這意味一個 VxD 應該不要嘗試探觸這樣的記憶體（甚至在 high-linear 位址上的記憶體也一樣），除非這個 VxD 知道這塊記憶體是 page-locked，或者其 paging I/O 是安全的。

爲了恢復一個 VM 的被排程資格，你可以呼叫 *Resume_VM*：

```
BOOL okay = Resume_VM(PVMMCB hVM);
```

Resume_VM 會將此 VM 之凍結計數器減 1。如果計數器到達 0，VMM 就送出一個 *VM_Resume* 系統控制訊息給每一個 device。任何 VxD 收到這個訊息，可以在回返時令 carry flag 設立，表示 *Resume_VM* call 失敗。如果 VxD 不可能在這時候將 VM 中執行的程式所需的資源 lock 住，那麼令 *Resume_VM* call 失敗的確是正確作法。如果你不想容忍失敗的發生，你可以使用另一個 service：*No_Fail_Resume_VM*。因爲如果新「出匣」（released）的 VM 有最高執行優先權，*Resume_VM* 會引起一個 task switch，那麼它就不能夠立刻回返。當 current VM 下一次移動到 execution queue 前面，回返動作才會發生。

執行焦點（execution focus）是你需要瞭解的另一個觀念。這個術語和前景（foreground）VM 有所關聯。end user 最終是以滑鼠點選、taskbar、或 Alt-Tab 熱鍵來控制哪一個 VM 有執行焦點。VMM 的回應是發出一個 *Set_Execution_Focus* call，它會把其他所有「無法在背景執行」的 VMs 都凍結住。

Time-slicing scheduler 還知道另一個 VM 狀態：idle（閒置）狀態。它很類似凍結（suspension）。Idle 的意思是，VM 沒有什麼有用的工作需要執行，但理論上它仍然處於執行的有效狀態下。VxD 可以呼叫 *Release_Time_Slice* service，強迫一個 VM 進入 idle 狀態。於是（藉由 INT 2Fh, function 1680h）VM 中正在執行的碼會釋出其 time slice。具有 Windows 意識的 DOS 程式，應該釋出它們的 time slices，而不是繼續 "polling" 某些預期的 event。一旦預期的 event 發生，一個協同合作的 VxD 應該將 VM 的優先權提昇，並發出 *Wake_Up_VM* call，將 VM 帶離 idle 狀態。如果沒有人做這兩件事情，VM 將有一段時間得不到 CPU 服務。

如果呼叫 *Time_Slice_Sleep* service，VxD 也可以令 VM 進入 idle 狀態一段時間。如果時間終了，或是某些人呼叫了 *Wake_Up_VM*，這個 VM 就會醒來。其實 VM 還可以因爲其他理由而更早醒來。換句話說，睡眠週期只是一個最大值，VxD 的使用者必須對提早醒來有所準備。

Critical Sections

VMM 實作有唯一一個 critical section，扮演大量系統組件的「敏感動作的守門員」。你可以呼叫 *Begin_Critical_Section* service 進入此一 section。如果 thread A 嘗試進入一個 section 而 thread B 已經擁有該 section，thread A 會被擋在門外，直到那個 section 恢復自由。如果 thread 嘗試進入一個它所擁有的 critical section，VMM 會將計數器累加 1。呼叫 *End_Critical_Section* 會使計數器減 1。當 *End_Critical_Section* 的呼叫次數和 *Begin_Critical_Section* 的呼叫次數一樣，這個 section 就自由了。請看表 **9-10**，其中列出與 critical sections 有關的所有 services。

當你進入 critical section 時，你可以指定某些 flag 位元，控制「thread 是否仍然可以被排程」以及「何時被排程」（甚至當它被迫暫停以等待 critical section）：

```
Begin_Critical_Section(flags);
```

其中的 *flags* 參數是下列位元的組合：

- *BLOCK_THREAD_IDLE* 意思是 thread 應被視爲 idle。
- *BLOCK_SVC_INTS* 允許 events 以及模擬中斷（simulated interrupts）得以被服務。
- *BLOCK_ENABLE_INTS* 使 VMM 將中斷反映（reflect）到 VM，甚至即使此 VM 處於 disabled 狀態。這個 flag 只有在 *BLOCK_SVC_INTS* 也設立的情況下才有意義。

Service(s)	說明
Begin_Critical_Section End_Critical_Section	進入或離開 critical section
Call_When_Not_Critical Cancel_Call_When_Not_Critical	產生或取消一個 callback，該函式是在 critical section 未被擁有時執行
Claim_Critical_Section Release_Critical_Section	增加或減少 critical section 計數器（可指定其值）

End_Crit_And_Suspend	凍結（Blocks）直到另一個 VM 處理了一個 event（此 event 乃傳統 event)
Get_Crit_Section_Status Get_Crit_Status_No_Block Get_Crit_Status_Thread	決定 critical section 目前的擁有者。

表 9-10 管理 critical section 的各式各樣 services

Windows 3.0 唯一提供的同步元素（synchronization primitive）只有 critical section 一種。於是，許多 VxDs 不管三七二十一地將任何它們想保護的東西，以宣示 critical section 主權的方式保護之。這樣的過度使用無疑是有殺傷力的，因爲沒有什麼特別理由使我們需要因爲「其他 thread 碰巧由於一個 "page fetch" 而停滯下來（blocked）」而護衛你自己 VxD 中的某小段程式碼。Critical section 的使用在 Windows 中仍然十分普遍，你恐怕得好好弄清楚它何時可能在你背後被別人宣示主權。下面是一個不完整的列表，列出 VxD 可能會宣示 critical section 主權的幾種常見情況。

- 在 paging 動作過程中。
- 在呼叫記憶體配置函式如 *_HeapAllocate*, *_HeapFree*, *_PageAllocate*, *_PageFree* 等的過程中。
- 在呼叫 selector 管理函式如 *_Allocate_LDT_Selector* 等的過程中。
- 在呼叫 *VDMAD_Scatter_Lock* 的過程中（因爲它依賴 *_LinPageLock* service）。
- 在 System VM 內取用 V86-mode services 的過程中。一個 thread 於 System VM 之中取用 V86 services 是可能的，即使其他 VM 擁有 critical section 也沒有關係（只要 System VM 中的其他 threads 不要擁有即可）。VMM 施行此一規則的作法是，維護一個 V86 mutex 和一個 critical section mutex：呼叫 *Begin_V86_Serialization* 可以取得 V86 mutex，在 System VM 中宣示 critical section 主權則可以取得兩個 mutexes。
- 在進入 DOS 之時。DOS 仍然是不可重進入的（non reentrant），即使在 Windows 95 中亦然。因此，VMM 組件中凡是需要呼叫 DOS services 者，都應該宣示 critical section，以避免重進入（reentrance）的發生。

- 當 DOS 程式發出 INT 2Fh, function 1601h。
- 在呼叫 *System_Control* 期間。換句話說，只要 VxD 處理系統控制訊息，它便是進入了 critical section。因此，它應該不會停滯（blocked）下來。從好的方面來說，你不再需要保護任何在你的「系統控制訊息處理常式」中執行的 code sections。保留這個 Windows 3.1 以來的行爲，很明顯是爲了回溯相容，因爲某些 non-Microsoft VxDs 依賴此一性質。

由於 critical section 可以控制極敏感的動作以及對執行時間十分在乎（time-critical）的動作，所以 VMM 會自動提昇 critical section 擁有者的執行優先權。提昇量 *CRITICAL_SECTION_BOOST* 等於 00100000h。這個量小於 *TIME_CRITICAL_BOOST*（00400000h，施行於 VMs 處理硬體中斷之時），但大於其他常見的優先權提昇量。因此 critical section 的擁有者幾乎確定會起而執行，唯一可能被延緩執行的原因是硬體中斷。

BLOCK_SVC_IF_INTS_LOCKED

另有一個 critical section flag 名爲 *BLOCK_SVC_IF_INTS_LOCKED*。我相信你像我一樣渴望知道它的意思。指定這個 flag 可以使 VM 只在 *VMSTAT_V86INTSLOCKED* flag 被設立時才爲中斷服務。現在你知道它的意思啦！

我打賭你還是不滿足。你可能還想知道上述那個 VM status flag 的意思。此 flag 是 Windows 3.0 留下來的，Microsoft 恐怕遺漏了它，即使他們程序就不知道有什麼人真正使用過它。此觀念得溯及「一個 paging driver 在等待磁碟 I/O 完成時必須處理 event」的要求。呼叫 *SetResetV86Pageable* 可設定 *VMSTAT_V86INTSLOCKED* flag，它不但會鎖住 local 記憶體，還會鎖住所有內含 global 記憶體和 instanced 記憶體的 pages。*SetResetV86Pageable* 使我們知道，被模擬的（simulated）硬體中斷不能夠引起巢狀（nested）page faults，因此「pager 允許被模擬之硬體中斷發生」是安全的。所以，如果你指定 *BLOCK_SVC_IF_INTS_ LOCKED*，就表示如果某些人以這種方式來使用 *SetResetV86Pageable*，那麼你可以容忍中斷發生。

Critical Sections 和 Event Callbacks

稍早我曾提醒你，一個 event callback 函式如果停滯下來（block），是非常危險的事情。由於 *Begin_Critical_Section* 可能停滯並因而產生一個死結（deadlock）狀態，所以你可能猜想沒有任何 event callback 敢呼叫這個函式。這在 Windows 3.1 的確是一個問題。程式開發人員於是演化出一種 Microsoft 稱之爲 "chasing（巡狩）the critical section" 技術，有了它，callback 函式將於新的 task 中保持自己的排程資格，直到它終於在一個「可以進入 critical section」的 task 中結束：

```
BeginProc OnSomeEvent
VMMCall Get_Crit_Section_Status
VMMCall Test_Cur_VM_Handle
jz      @F
mov     esi, offset32 OnSomeEvent
VMMCall Schedule_VM_Event
ret
@@:
mov     ecx, Block_Svc_Ints
VMMCall Begin_Critical_Section
...
VMMCall End_Critical_Section
ret
EndProc OnSomeEvent
```

一開始你可以以一個 global event 或是一個 VM event 來對此函式排程，前者或後者都無關緊要。此函式中的碼極爲簡潔，運作如下。一開始呼叫 *Get_Crit_Section_Status*，傳回時 EBX 內含的若不是 current VM handle（如果沒有任何人擁有 critical section）就是「擁有 critical section」的那個 VM 的 handle。這個函式也會引發 task switch，因爲 VMM 的一個最佳化動作會延緩釋出 critical section，直到它將 events 處理完畢爲止。*Test_Cur_VM_Handle* 會測試是否 EBX 指向 current VM 的控制區塊（control block）。如果是，呼叫 *Begin_Critical_Section* 將是安全的，理由有二：如果此刻無人擁有 critical section，就可以很安全地宣示其主權，否則如果 current VM 已經擁有主權，那麼增加宣示次數也是安全的。然而如果 EBX 指向其他 VM，呼叫 *Schedule_VM_Event* 便會將目前該 VM 相同的 event 處理函式納入排程，等待控制權重回 callback 函式。我們希望

當控制權下一次重回 callback 函式時，其他 VM 仍然擁有 critical section。如果不是如此，這樣的程序會重複。

在 Windows 95 中，你不需要寫出這麼複雜的碼，你可以只是以 *PEF_Wait_Crit* flag 來對一個受限的（restricted）global event 排程：

```
mov     eax, <>PriorityBoost        ; 0 or something else
xor     ebx, ebx                    ; indicate global event
mov     ecx, PEF_Wait_Crit          ; we need to enter critical section
mov     edx, <>ReferenceData        ; pass-through data to callback
mov     esi, offset32 OnSomeEvent   ; callback function
mov     edi, <>TimeOut              ; in milliseconds
VMMCall Call_Restricted_Event
...
```

只有當「無人擁有 critical section」或是當「current VM 擁有 critical section」時，event callback 函式才會以此方式排程。就像在較爲複雜的 Windows 3.1 例子中一樣，不論上述哪一種情況，都意味 callback function 可以安全進入 critical section。

Synchronization Objects（同步物件）

這一節中，我要討論 VxD 撰寫者可以運用的三種 synchronization objects（同步物件）：semaphores, mutual exclusion（mutex）objects, 以及 blocking identifiers。進行討論之前，我要提出一些程式寫作技巧，給你一個高效率的「同步執行」實踐法 -- 甚至在一個高度變換的多工環境下。

實作一個私有的（Private）Critical Section

第一個寫碼策略，其實不怎麼像什麼策略。在一部單 CPU 系統中（這是 Windows 95 目前唯一支援的機種），非浮點數指令（non-floating-point instructions）不能夠被中斷。即使在一部多 CPU 系統中，在一個指令之前使用 LOCK prefix 也可以保證任何其他 CPU 都不得處理共享記憶體，直到目前的指令完成爲止。因此，程式員常常使用 XCHG 和 INC 或 DEC 指令來完成所謂的不可切割（atomic）動作，因爲他們知道岔斷

（preemption）的發生不會影響運算結果。

假設你要實作出一個 private critical section，用來保護一筆共享資源，此資源將與「執行同一份程式碼」之其他 threads 共享。請設定一塊記憶體，內含一個計數器。此計數器初值爲 -1，意思是目前沒有人進入 critical section。數值 0 表示有一個 thread 進入了此 critical section。大於 0 的數值表示「目前阻塞而正等待進入 critical section」的 threads 個數。在正常（沒有發生搶奪）的情況下，你的碼可以宣示擁有此 section 的主權：

```
inc claimcount
jz  claimed           ; jump if claim counter == 0
```

如果當時 critical section 已被宣示主權，你所增加的計數值就不會是 0，因此必須明白地等待一個 semaphore（號誌）。你應該以 0 值來初始化此一 semaphore，如此一來第一個呼叫者便會停滯下來（blocks）。

如果沒有發生搶奪（competition）的情況，那麼 critical section 的釋出是一樣地簡單：

```
dec claimcount
jl  released          ; jump if claim counter < 0 now
```

如果 DEC 指令使計數器值爲 0 或更大值，意味某個 thread 已在某處被阻塞，正等待進入此 critical section。這種情況下會激發 semaphore，於是釋出一個等待中的 thread。順帶一提，不要依賴 C 語言的 ++ 和 -- 運算子，因爲編譯器可能不會產生 INC 和 DEC 指令，這意味你不確定能夠在單一指令上獲得不可切割的（atomic）行爲。

使用 CMPXCHG

如果你知道你的碼將在 Intel i486 或更新的 CPU 上執行，你可以使用「比較並交換（compare-and-exchange）」指令，以求獲得更多好處。舉個例子，假設多個 threads 共同維護一個共享的計數器變數，此變數用來指定識別碼，而識別碼必須在整個系統中獨一無二。我想你會希望以計數器值做爲一個識別碼，然後累加此計數器，並確保在這兩

個步驟之間沒有其他動作會增加計數器值。

一個安全的作法就是，以 CMPXCHG 指令來增加計數器值，像這樣：

```
     mov   eax, idcounter
@@:  lea   ecx, [eax + 1]
     cmpxchg idcounter, ecx
     jnz   @B
```

當這段碼完成，ECX 內含一個獨一無二的識別碼。CMPXCHG 指令可以神奇地令這段碼爲 "thread-safe"（譯註：意指不受岔斷影響）。在迴圈的每次迭代中，EAX 內含 *idcounter* 的假設現值，CMPXCHG 會將 EAX 拿來和 *idcounter* 比較。如果它們相等，意味我們的假設是正確的，而 ECX 此刻內含著下一個值。這時候 CMPXCHG 將 ECX 的內容儲存於 *idcounter* 並設立 zero flag 以結束迴圈。然而如果它們並不相等，CMPXCHG 會從 *idcounter* 中將資料載入 EAX，並清除 zero flag，於是迴圈繼續進行。由於 CMPXCHG 係在一個不可切割（atomic）的動作中做掉所有事情，所以此迴圈可以隨時被岔斷（preempted），不至於讓不同的 threads 獲得相同的識別碼。只要你爲 CMPXCHG 指令加上一個 LOCK prefix，這段碼便能夠在多 CPU 系統中有效運作。

下面是使用 CMPXCHG 的另一策略。假設你有一個串列（linked list），被其他 processes 共享。串列中的每一個元素有一個串鏈（chain）欄位，佔用第一個 DWORD；最後一個元素的串鏈欄位內容是 NULL。爲了安全地在串列頭部加上一個新元素，而此元素的位址係記錄於 ESI 之中，你可以這麼做：

```
     mov   eax, listhead       ; EAX -> head of list
@@:  mov   [esi], eax          ; ESI's chain -> head
     cmpxchg listhead, esi     ; make [ESI] head if EAX still head
     jnz   @B                  ; repeat with new head in EAX
```

但這豈不是將時間浪費於 ring0 碼中潛在性的無窮迴圈了嗎？不，因爲系統不太可能重複岔斷此一迴圈而不讓這個 thread 執行至少一兩個指令。只要執行一兩個指令，就足以跳脫迴圈了。事實上，此迴圈甚至不太可能重複一次，因爲以機率而言，幾乎不可能有另一個 thread 碰巧在 context switch 發生時執行相同範圍的碼。我們知道「共享變數」

在某些時候需要保護，而在壓倒性的常見情況中並且又沒有真正發生「競爭」時，使用低價方法來達到保護，是一件很棒的事。

Semaphore Objects

Windows 3.1 導入 ring0 semaphores，用來做爲同步控制 VM 的一種方法。Windows 95 採納 semaphore 觀念，放進其「以 thread 爲基礎的多工模型」中。表 **9-11** 列出你可以用來處理 semaphores 的各個 VMM services。Semaphores 基本上是個計數器，帶有相關的特殊語意。當你呼叫 *Create_Semaphore*，你可以指定一個初值：

```
VMM_SEMAPHORE sem = Create_Semaphore(initcount);
```

Service	說明
Create_Semaphore	產生一個新的 semaphore object
Destroy_Semaphore	摧毀一個 semaphore object
Signal_Semaphore	激發（signals）semaphore 以釋出一個等待中的 thread
Signal_Semaphore_No_Switch	激發（signals）semaphore，但不立即執行 task switch
Wait_Semaphore	等待某個 semaphore 被激發（signaled）

表 9-11　Semaphore services

semaphore 的初值通常是 0，於是產生出來的 semaphore 會使「第一次企圖等待它的某個 thread」停滯（block）下來。任何人如果需要等候取得「被 semaphore 控制的資源」，應該呼叫 *Wait_Semaphore*：

```
Wait_Semaphore(sem, flags);
```

其中 *flags* 參數可以指定一些項目。如果這個 semaphore 恰巧被停滯（block），那麼中斷在呼叫端（thread）中如何被服務，便與這些項目設定有關。這些 flags 和我稍早針對 *Begin_Critical_Section* 所描述的相同(表 **9-10** 之下)。這個函式會降低並測試 semaphore

的計數值，如果新值小於 0，*Wait_Semaphore* service 會凍結此 semaphore，直到再有什麼人激發（signals）它爲止。如果新值不小於 0，就立刻回返。

爲了釋出一個等待中的 thread，另一個 thread 必須呼叫 *Signal_Semaphore*：

```
Signal_Semaphore(sem);
```

Signal_Semaphore 會增加計數值然後檢查是否有任何 thread 正在等待這個 semaphore。如果有，它就釋放被凍結的 threads 之中最高優先權的那個，並可能因此立刻引發 task switch。一旦 *Signal_Semaphore* 終於回返(可能是立刻發生 -- 如果沒有 task switch 的話），中斷會被 enabled。

如果你不希望立刻發生一個 thread switch，或如果中斷被 disabled 而你希望它們繼續保持，請呼叫 *Signal_Semaphore_No_Switch* 而非 *Signal_Semaphore*。當 VMM 下一次處理 events 時，任何該發生的 context switch 就會發生。

注意 事實上 Windows 95 以完全相同的碼來實作 *Signal_Semaphore* 和 *Signal_Semaphore_No_Switch*，這意味著呼叫 *Signal_Semaphore* 並不會改變中斷旗標，也不會引起 context switch。但這種情況可能會在未來有所變化，所以你應該在「*Signal_Semaphore* 會帶來副作用」的假設下寫你的碼。

Mutex Objects

Mutex 這個字眼代表 "**mutual exclusion**"。你可以利用 mutex 來阻止兩個 threads 同時執行某些程式碼。表 **9-12** 列出你可以用來處理 mutex 的所有 VMM services。呼叫 *CreateMutex* 可以產生一個 mutex：

```
PVMMMUTEX mutex = _CreateMutex(boost, flags);
```

此處 *boost* 是一個優先權提昇值，自動施行於擁有 mutex 的 thread 身上。*flags* 參數包括我稍早在介紹 *Begin_Critical_Section* 時所描述的位元。flags 參數也可以內含

MUTEX_MUST_COMPLETE，表示不管誰擁有 mutex，時常會有一個所謂的 "must complete" 狀態，會阻止凍結（suspending）或摧毀（destroying）的施行。

Service	說明
_CreateMutex	產生一個新的 mutex
_DestroyMutex	摧毀一個 mutex
_EnterMutex	進入一個 mutual exclusion section
_GetMutexOwner	決定（判斷）哪一個 thread 擁有一個 mutex
_LeaveMutex	離開一個 mutual exclusion section

表 9-12 Mutex services.

一般而言，你應該以 *_EnterMutex* 和 *_LeaveMutex* 將不可重入（nonreentrant）的程式碼包夾起來，像這樣：

```
_EnterMutex(mutex);
[sensitive code that only one thread can run at a time]
_LeaveMutex(mutex);
```

Blocking Identifiers

Ring3 Win32 程式可以使用 event objects 來同步化多個 threads。Event object 的目的是允許一個 thread 等待，直到某些 event 發生；也就是說直到其他某些 thread 完成了一個動作（event 這個字眼，在這裡，和平常對 VxD 程式員的意義稍有不同）。VMM 並未提供類似 Win32 event object 那樣的 object。VMM 提供的是一對奇怪的同步化（synchronization）services，名爲 *_BlockOnID* 和 *_SignalID*，你可以用來實作出一個低成本的 event object。

_BlockOnID 的使用十分單純。只要選擇一個隨機的 32 位元數值（再不會與他人相同），做爲 blocking ID，然後呼叫這個 service：

```
VMMCall _BlockOnId, <random-32-bit-number, 0>
```

（*_BlockOnID* service 的第二參數內含的 flags，與你在 *Begin_Critical_Section* 時所用的相同，用以控制正被凍結中的 VM 的中斷處理）。現在你的 thread 被凍結住了，直到其他 thread 呼叫 *_SignalID*（使用相同的 blocking ID）：

```
VMMCall _SignalID, <same-random-32-bit-number>
```

我可曾說過這些 services 的使用很簡單？唔，真的實在是太簡單了。首先，*_SignalID* 就像 Win32 *PulseEvent* 函式一樣，它只釋放那些「在你呼叫的那一刻，正處於等待狀態」的 threads。因此，你必須當心其他亦處於等待狀態者更早激發此一 ID 的可能性。第二，*_SignalID* 會喚醒每一個等待此一 blocking ID 之 thread，因此如果你和別人的 VxDs 恰巧選擇了相同的 blocking ID，你會喚醒其他那些 threads。因此，你必須有一些額外手段以知曉某特別的 *_SignalID* 是爲你而發。通常，VxDs 會使用某個函式或它所控制的資料區塊的 flat 位址做爲 blocking ID，降低與其他 VxD 衝突的可能，並簡化工作。

如果你需要比 *_BlockOnID* 和 *_SignalID* 更強固（robust）的 event object，可以自己寫碼來產生一個。你應該遵循三個步驟，實作出一個以 blocking ID services 產生而得的 event object：

1. 將記憶體中的兩個位元設初值爲 0。其中一個位元我稱之爲 **wait bit**，表示有人正在等待此一 event。另一個位元我稱之爲 **post bit**，表示這個 event 已經發生（這些術語係 OS/360 用來描述它針對 objects 類似物所提供的動作。本書讀者大概都太年輕，不知道 OS/360 是什麼東西。OS/360 是 IBM 在軟體界尚具主宰力時候的產品）。初始化動作完成之後，就可以安全地開始一個「其執行結果會被其他 thread 等待」的動作。
2. 當 thread 需要某個動作的結果（不管是什麼動作），它會執行 wait 動作如下：如果 post bit 設立，表示該動作已完成，程式可以前進下去，否則，就設立 wait bit 並呼叫 *BlockOnID* 將自己凍結。一旦 *_BlockOnID* 回返，就回頭再次測試 post bit，避免「重複 blocking ID」的可能性。測試 post bit、設立 wait bit、呼叫 *_BlockOnID*，三者必須一氣呵成，其他 thread 不能夠在其間呼叫

_SignalID，否則 *_SignalID* call 可能會反而在 *_BlockOnID* call 之前，那麼 thread 就會被永遠凍結住（如果此處所討論的碼是 page-locked，那麼將中斷 disabled 掉，便足以保護之）。

3. 一旦程式所執行的那個動作真正完成了，就執行一個 post 動作如下：設立 post bit。如果 wait bit 也已設立，就呼叫 *_SignalID* 來喚醒目前因此 event 而被凍結的 thread。其實在任何情況下呼叫 *_SignalID* 都是安全的。只不過在必要時才呼叫它，速度會比較快。

這些解釋似乎有點難以領會。嘿，開心一點，程式碼其實十分簡單：

```
ecb       db    0
wait_bit  equ   1
post_bit  equ   2
...
          mov    ecb, 0        ; initialize event control block
...
wait_for_event:
          pushfd               ; save interrupt flag
@@:       cli                  ; disable interrupts
          test   ecb, post_bit ; event already posted?
          jnz    @F            ; if yes, proceed
          or     ecb, wait_bit ; indicate we're waiting
          VMMCall _BlockOnId, <<offset32 ecb>, 0>
          jmp    @B            ; guard against spurious wakeup
@@:       popfd                ; restore interrupt flag
...
post_event:
          or     ecb, post_bit ; indicate event is complete
          test   ecb, wait_bit ; anyone waiting?
          jz     @F            ; if not, no signal needed
          VMMCall _SignalID, <<offset32 ecb>>
@@:
```

Ring3 vs. Ring0 Synchronization Objects

截至目前，我已經討論過 ring0 的 VxDs 同步基礎元件。結束本章之前，我還有其他同步物件(synchronization object)要討論。你可以和一個 Win32 程式共享同一個 ring3 event object -- 只要使用 VWIN32 提供的 services（表 **9-13**）即可。請加強一個觀念：VxD 和 Win32 程式是同時並存的，ring3 程式必須先呼叫 *CreateEvent* 產生出一個 event

object，然後呼叫 *OpenVxDHandle*，爲此 event object 產生一個 ring0 handle。Win32 程式然後應該使用任何方便手法，把這個 ring0 handle 交給 VxD。VxD（而非 Win32 程式）最終必須呼叫 *_VWIN32_CloseVxDHandle* 以釋放此一 handle。

Service	說明
_VWIN32_CloseVxDHandle	關閉一個 ring3 event 的 ring0 handle
_VWIN32_PulseWin32Event	"Pulses" 一個 ring3 event
_VWIN32_ResetWin32Event	將一個 ring3 event "reset" 爲未激發（unsignaled）狀態
_VWIN32_SetWin32Event	將一個 ring3 event "set" 爲激發狀態（signaled state）
_VWIN32_WaitMultipleObjects	等待一個或多個 ring3 events 被激發
_VWIN32_WaitSingleObject	等待一個 ring3 event 被激發

譯註：上述的 "set" 和 "reset" 是討論激發（signaled）或未激發（unsignaled）狀態時的一種專用術語。"set" 表示「使處於激發狀態」，"reset" 表示「使處於未激發狀態」。"pulse" 是指先 set 然後立刻 reset，很像所謂的「脈衝」。

表 9-13 Ring3 event services

注意 有件事很重要，你得知道，儘管這些 VWIN32 services 的名稱前面都有底線前導，外觀上會以爲它們都是 C-convention services，但其實它們都是以暫存器來傳遞參數（所謂的 register-oriented）。

在稍早某些版本的 Windows 95 SDK 中，*OpenVxDHandle* 是 KERNEL32.DLL 的 import library 中的一個符號。由於 KERNEL32.LIB 基本上可以在 Windows 95 和 Windows NT 之間互相移植，所以 *OpenVxDHandle* 就不再可用了（我是指在 Visual C+ 4.0 中）。你必須以類似下面這樣的碼做明白的聯結動作：

```
typedef DWORD(WINAPI *OPENVXDHANDLE)(HANDLE);
OPENVXDHANDLE OpenVxDHandle = (OPENVXDHANDLE)
   GetProcAddress(GetModuleHandle ("KERNEL32"), "OpenVxDHandle");
DWORD hr0Event = OpenVxDHandle(hEvent);
```

VxD 也將 ring0 event handle 應用在表 **9-13** 以外的其他 VWIN32 services。這些行爲就好像 Win32 程式在對應的 Win32 call 身上的作爲一樣。

第 10 章

協助 Windows 95 管理虛擬機器

在先前的 Windows 作業系統版本之中，撰寫 VxD 的複雜度主要在於對 MS-DOS VM 的支援，讓 Windows 彷彿消失不見。雖然圖形導向的 Windows 應用軟體很明顯地比字元模式的 MS-DOS 應用軟體有更多優勢，我們週遭的 MS-DOS 應用軟體還是很多，因此繼續需要上述的支援。而且由於 Windows 應用軟體本身也是在一個幽禁的 VM 中跑，系統程式設計者還是有必要對 VM 有足夠的瞭解。

VMs 有一個明確的生命週期，由一組系統控制訊息左右。這些訊息由 SHELL virtual device 在一部 VM 產生和結束時，傳送給所有的 VxDs。由於在每一部 VM 中有著完全分離的「真實模式可定址之 1st MB 記憶體」，將是一種浪費，更不必說邏輯上的不正確性了，所以 VMM 在管理位址空間的 V86 region 時，遇到很大的麻煩。在前一版 Windows 中，MS-DOS 是一個隱藏在 Windows 作業環境背後的真正作業系統。由於真實模式驅動程式和常駐程式（TSR）組成了系統的中樞骨幹，VMM 於是竭心盡力地提供對軟體中斷的支援，讓應用程式藉它獲得系統服務。爲了回溯相容，這樣的支援在 Windows 95 中依然存在，不過 Win95 另外提供了一個方法，給那些沒辦法轉往「以 VxD

做爲解決方案之主要」的廠商一條生路。

VMM 還提供一個與「虛擬化需求」完全無關的機制，允許應用程式直接與 VxDs 通訊。16 位元程式使用經由 INT 2Fh 獲得的 API 位址，32 位元程式則使用一個正規 Win32 API 的變種，來達到目的。VxD 可以 "post" Windows 訊息，以求獲得應用程式的注意，也可以在獲得 "application time"（或是比較古怪的 "AppyTime"）的時候，直接呼叫 16 位元碼。與應用程式有較多的合作並克服較多的困難之後，VxDs 也可以藉著 Windows 95 的非同步函式呼叫設施（asynchronous procedure call facility）呼叫 32 位元函式。

虛擬機器（VM）的生命週期

將一個 device 虛擬化，意味著爲每一部 VM 表現出一塊假想硬體。欲如此做，通常意味你必須持續追蹤 VM 的誕生與死亡。在 VM 的生命期間，VMM 會送出數個系統控制訊息，幫助你將你的 device 虛擬化。我曾在第 4 章對這些訊息做了一份摘要（表 **4-2**），現在我要做進一步的解釋。

> 譯註：以下出現的 **MS-DOS prompt**，是 Windows 95 的【開始/程式集】功能表所提供的一個項目，Win95 中文版名爲「MS-DOS 模式」。我仍沿用其原名。這個 item 聯結到 \WIN95\COMMAND.COM，執行它就會開啓一個 MS-DOS Box （有時稱爲一個 Win95 console），也就是開啓一個 DOS VM。

使用者觸發一部新 VM 的方法是：打開（執行）一個 **MS-DOS prompt**，或從 Windows shell（例如檔案總管 Explorer）中執行一個 MS-DOS 應用程式。Windows 程式可以利用 *WinExec* 產生新的 VMs。整合開發環境如 Microsoft Developer Studio 則使用一些未公開方法產生一個隱藏的 VM。不管起源爲何，VMs 都是經由 SHELL virtual device 的調解才得以誕生。SHELL virtual device 執行了許多工作，最後呼叫 *System_Control* service。VxD 係藉由對一系列系統控制訊息的回應，來參與一部新 VM 的誕生：

1. SHELL device 產生一個 *Create_VM* 訊息。VMM 對此的回應是產生一個 control blocks（VMCB），這是描述 VM 的必須品。然後 VMM 送出 *Create_VM* 給所有的 devices。如果你以 *_Allocate_Device_CB_Area* 在 VMCB 中保留一塊空間，此時正是爲它設定初值的時機。以 Virtual Display Driver（VDD）爲例，它會爲它的「每個 VM 都該有一份」的資訊設定初值，並初始化任何必要的記憶體或 port traps （第 13 章將討論之），如此一來就可以將「對 Video 硬體的 I/O 動作」虛擬化。

2. VMM 最初是以 suspended state（凍結狀態）來產生 VM，並將 events 納入排程，以便在新的 VM context 之中完成初始化動作。然後才呼叫 *Resume_VM* 讓 VM（更重要的是用以完成 VM 誕生動作之 event 函式）有起而執行的資格。

3. SHELL device 接下來送出一個 *VM_Critical_Init* 訊息。目前爲止你還不能夠在此 VM 中執行 ring3 程式。Ring3 程式碼第一次有機會執行 port I/O、發出軟體中斷、觸及 mapped memory locations 之前，所有的必備設定動作都必須在 *VM_Init* 訊息前完成，以解除 ring3 魔咒的束縛。所以你必須在這個時候或是在處理 *Create_VM* 訊息時完成那些設定。這個訊息和 *Create_VM* 訊息之間的差異十分微小。VMM 的 *System_Control* service 要求在它呼叫 VxD device control procedures 的前後，以一個 critical section 保護之。*Create_VM* 訊息發生在某個 VM （通常是 System VM）之中，而不在目前正被產生的 VM 之中。所以「處理 *VM_Critical_Init* 訊息」處，就是進行初始化步驟的地方，而它可能需要對新的 VM 宣告一個 critical section。舉個例子，VDD 將 physical video BIOS pages 映射到一個新的 VM 之中，允許真實模式程式能夠真正使用 video system；顯然這是個好主意。由於映射（mapping）動作需要 critical section 並且可能會被凍結（blocking），所以 VDD 在此階段執行之。除此之外，你大概不需要在意這一訊息。

4. SHELL device 送出一個 *VM_Init* 訊息。到了這個時刻，VM 已經完全有作用了，現在是和真實模式碼聯絡的時候，靠著那些碼 VM 才能夠執行其真實模式應用程式。舉個例子，VDD 會發出一些 INT 10h 函式呼叫，將真實模式的 video subsystem 初始化。

5. 最後，SHELL 送出一系列的 *Set_Device_Focus* 訊息，引導共享裝置（如螢幕、滑鼠、鍵盤）之驅動程式，將擁有權指定給新的 VM。當然啦，所謂「切換 device

focus」，必須將 execution focus 指定給新 VM，這比產生一個新的 VM 的工作量多。

注意 你可能讀過 DDK 文件中所說的，「在 *VM_Critical_Init* 訊息期間，中斷被 disabled；訊息處理常式千萬不能將中斷 enable 起來」。這段敘述在 Windows 3.1 中並不確實，在 Windows 95 中也不確實，我不知道爲什麼 DDK 文件中這麼寫。

將 VM 初始化之後，VMM 便讓它進入正常的岔斷式多工（preemptive multitasking）系統之中，和其他 VMs 一起排班。產生 VM 的那個應用程式可藉由呼叫 *_DOSMGR_Set_Exec_VM_Data* 將程式執行起來。程式於是開始執行。當它結束並回返，VMM 重獲控制權，於是依序摧毀 VM。訊息產生的次序分別是 *VM_Terminate*, *VM_Terminate2*, *VM_Suspend*, *VM_Suspend2*, *VM_Not_Executable*, *VM_Not_Executable2*, *Set_Device_Focus*, *Destroy_VM*, *Destroy_VM2*（請記住，VM 收到 "*2*" 之類訊息的次序恰與初始化次序相反）。

注意 如果你監視那些伴隨 "MS-DOS prompt" 的誕生和毀滅而出現的系統控制訊息，你一定會注意到有一個 thread 也被產生和摧毀。WINOLDAP("MS-DOS prompts" 的 GUI 介面）就是在此 thread 中執行。與此新誕生之 VM 有關的 thread 另外還有一個，VMM 利用它來爲新 VM 向系統索求 CPU 時間（也就是讓這個 thread 進入排程隊伍），但是絕不會送一般的系統控制訊息給它。

停工（shutdown）程序就不是這麼井然有序了，那是在 VM 出現不良狀況（例如 general protection fault）時發生的。VxD 可以呼叫 *Crash_Cur_VM* 或策略上並不頂好的 *Nuke_VM*，粗暴結束一個 VM。在此情況下 VxD 接獲的第一個「VM 正在停工」的警報是 *VM_Not_Executable* 訊息。因爲這個原因，大部份驅動程式並不會分心去處理 *VM_Terminate*，因爲不一定能收到它。

保護模式概引程式

令人困擾的記憶體大小限制問題一度衍生出所謂的 DOS extenders 次級工業。DOS extenders 是一個系統層次的軟體，使用保護模式和 extended memory（譯註：1MB 以上），允許極大的程式在 MS-DOS 中執行。當 Windows 3.0 進入市場時，大部份主要的應用軟體都在使用 DOS extender，因爲 640KB 記憶體實在太少了。Microsoft 把 DOS extender 的技術以 DPMI 的型式內建於 Windows 之中，以各式各樣的 VxDs 來擴充 DPMI，作法是提供保護模式至真實模式的 API 轉換，這些以中斷爲導向的 API 在 MS-DOS 程式和 16 位元 Windows 程式中被大量使用。

DOS extenders 的世界和 VxDs 有些衝突。稍後我會談到 VxD 如何補充存在於 Windows 95 中的 DOS extender 的機能。現在我想談的是兩個附屬於 DPMI 的系統控制訊息。在一部只有 MS-DOS 而沒有 Windows 的機器中，DOS extender 簡單地使用一些高權級（privileged）指令如 LGDT 或 MOV into CR0，將 CPU 切換到保護模式去。但如果 Windows 存在，它就必須改用 INT 2Fh, function 1687h 來獲得一個模式切換函式的入口（位址）。呼叫該函式，便可將 VM 從 V86 模式切換到保護模式。最後，應用程式應該呼叫 INT 21h, function 4Ch 結束自己，於是 VM 回返到 V86 模式。

當真實模式程式第一次呼叫 DPMI 的模式切換函式，VMM 會送給所有驅動程式一個 *Begin_PM_App* 訊息。一旦保護模式程式結束，VMM 會送出一個 *End_PM_App* 訊息。VxD 子系統中那些被統稱爲 DPMI 的驅動程式，會對上述訊息有反應。舉個例子，如果離開保護模式而未釋放以 DPMI function 0501h 配置而來的記憶體，便會觸發記憶體的自動釋放機制。

應用程式切換到保護模式之後，VM control block 的 *VMSTAT_PM_APP* 狀態旗標會設立起來。如果應用程式在模式切換的當初表明自己是一個 32 位元程式，那麼 *VMSTAT_PM_USE32* 旗標也會被設立起來。當 VM 處於保護模式，*VMSTAT_PM_EXEC* 旗標會被設立起來；但是在 VM 進入 V86 模式的那段期間，*VMSTAT_PM_EXEC* 旗標會被清除。

System VM

上述討論適用於 Windows 啓動之後所產生的 MS-DOS VMs。然而至少有一個 VM 是在 Windows 執行時就存在的，那就是 System VM，也就是 Windows 安身立命之處。VMM 自動產生 System VM，做爲正常啓動程序的一部份。因此 VxDs 絕不可能看到 System VM 發出一般 VM 啓動和停工時所發出的那些訊息。因此：

- device 針對 System VM 的 *Create_VM* 或 *VM_Critical_Init* 訊息所打算進行的各種初始化動作，都應該在 device 的初始化三階段中完成。
- device 可以處理 *Sys_VM_Init* 訊息（類似 *VM_Init* 訊息）。然而較常見的作法是在 *Device_Init* 處理常式中簡單呼叫一個自己的 *VM_Init* 處理常式。由於驅動程式的初始化動作發生在 System VM context 之中，這個時機適合用來觸發必要的函式。
- 除了 *VM_Terminate* 訊息外，device 還可以處理 *Sys_VM_Terminate*（以及 *Sys_VM_Terminate2*）訊息。不過較常見的情況是忽略它們，改處理另一個訊息：*System_Exit*。甚至在 VMM 沒有產生 *Sys_VM_Terminate* 訊息的情況下，你還是能夠收到 *System_Exit* 訊息。

V86 Region

記憶體中的 V86 region 係由「V86 程式能夠以 segment:offset 定址到的所有線性位址」組成，合計大約有 1.1MB，從 0000:0000 開始，延伸到 High Memory Area（HMA，譯註）的 FFFF:FFFF。圖 **10-1** 說明了眾所周知的真實模式記憶體佈局。當一台 PC 正在執行 MS-DOS，情況就是如此。MS-DOS、系統啓動時所載入的 device driver、以及使用者所載入的任何常駐程式（TSRs）都佔據記憶體的最低位址。從常駐程式及其資料區的尾部到配接卡記憶區（adapter memory area）起始處 A000:0000 之間，是一塊廣大空間，給 MS-DOS 應用程式使用。配接卡記憶區（adapter memory area）的內容完全視硬體的存在而變化，不過某些特定位置是可以預期的。例如 video 卡上的記憶體一定佔用 A000:0000 ~ Cx00:0000，ROM BIOS 一定佔用 F000:0000 ~ F000:FFFF 之間。BIOS 底下有多達 128K 的位址空間可能並沒有映射到任何實際記憶體。

譯註：MS-DOS 一直爲位址空間短缺所苦。其 5.0 版成功利用邏輯定址 FFFF:FFFF 所獲得的高於 1MB 的空間（10FFEF-FFFFF=FFF0，也就是 64K-16bytes），大量挹注了幾近 64K 定址空間。這項創舉在當時是技術上的大突破。1MB 以上的這幾近 64K 的區域就被稱爲 High Memory Area（HMA）。由於定址 1MB 以上的位址空間前必須先將 A20 位址線打開，所以 MS-DOS 5.0 提供了一個 WINA20.386。.386 就是 LE（Linear Executable）檔案格式，也就是日後的 .VXD。

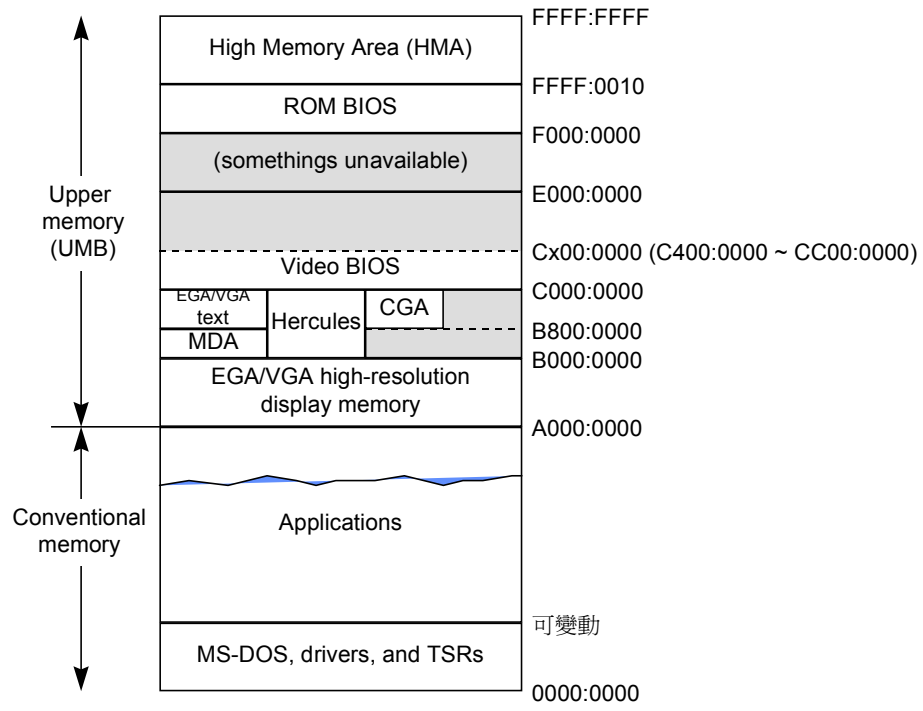


圖 10-1　MS-DOS 之下的眞實模式記憶體映射狀況

在 V86 region 這一主題上，VMM 的工作主要是模仿圖 **10-1** 的佈局，讓 V86 程式能夠如以往一般地在多個 VMs 上正常執行。表 **10-1** 列出相關的 services。

Service	說明
_AddInstanceItem	任命一段 V86 記憶體，使它內含所謂的 instance data（也就是每個 VMs 看到的資料都不相同）。
_Allocate_Global_V86_Data_Area	配置一小塊記憶體，藉由它，VxD 和真實模式中的通訊對手溝通。
_Assign_Device_V86_Pages	將一段 V86 pages 標記爲「被指定」（不論是全域性指定，或是指定給某個 VM）。
_DeAssign_Device_V86_Pages	將一段「被指定」的 V86 pages 標記爲「不再被指定」。
_GetFirstV86Pages	取得某特定 VM 中的第一個 local page 的號碼。
_Get_Device_V86_Pages_Array	取得用以描述「被指定之 V86 pages」的 bit map 副本。
_GetLastV86Pages	取得某特定 VM 中的最後一個 local page 的號碼。
GetSet_HMA_Info	取出或改變與 high memory area（HMA）有關的資訊。
Hook_V86_Page	對一個特定的 V86 位址建立一個 page fault 處理常式。
_LinMapIntoV86	將一塊線性位址已知的記憶體，映射到 V86 page table 中。
_MapIntoV86	將一塊 block handle 已知的記憶體，映射到 V86 page table 中。
_PhysIntoV86	將一塊實際位址已知的記憶體，映射到 V86 page table 中。
_TestGlobalV86Mem	記錄某範圍之位址中的 global, local 或 instanced 的狀態

表 10-1 V86 區域的管理函式

預設情況下，VMM 會將 Windows 啓動時使用的所有記憶體歸類爲 global。Global 的意思是，每一個 VM 在相同的線性位址上內含相同的資料。MS-DOS 程式碼以及 BIOS 就是 global，意味 System VM 和每一個 "MS-DOS prompt" 共享同一份 MS-DOS 碼和 BIOS 碼。真實模式驅動程式和常駐程式（TSRs）也屬於這一類。

相反的，local 記憶體表示在不同的 VM 中，相同的線性位址映射到不同的實際記憶體。當你啓動 Windows，大部份的 conventional memory（譯註：640 以內）是自由而可用的，並被歸類爲 local。因此，每次你打開一個 "MS-DOS prompt"，你所擁有的 conventional memory 大小完全一樣。讓這些記憶體成爲 local，可以使每一個代表 MS-DOS sessions

的 VM 各自獨立。

Local 記憶體以 pages 爲界，你可以呼叫 *_GetFirstV86Page* 和 *_GetLastV86Page* 獲得那些 pages。舉個例，SHELL device 在產生一個新的 VM 期間，動作之一就是呼叫 *_V86MMGR_Allocate_V86_Pages*，設定 V86 記憶體基底（base）。該 service 內部其實就是呼叫 *_GetFirstV86Page* 和 *_GetLastV86Page* 來決定 local 記憶體的區域範圍。

在 device 初始化期間，也有可能配置一塊記憶體並最終成爲 global 屬性而非 local 屬性。*_Allocate_Global_V86_Data_Area* 可保留一塊記憶體給 VxD 以及「和 VxD 通訊的真實模式碼」共同使用。由於真實模式驅動程式或常駐程式沒有能力定址到 V86 region 以外的位址，所以上述 service 事實上是允許真實模式程式和 VxD 共享一塊記憶體。如果你的程式架構需要一塊永久性的 V86 記憶體，或如果真實模式碼呼叫 VxD（在填寫一塊真實模式的記憶體緩衝區之後），你就需要上述 service。然而如果 VxD 呼叫真實模式碼，並且需要提供一個指向暫時資料區的指標，V86MMGR（被設計用以擴充 DOS）的 services 會比較適合這個任務。

Instance Memory

> 譯註：以下出現的 "instance" 字眼，很難簡明譯出。此字可爲動詞亦可爲名詞，在這一節的意思是「每個 VM 特有一份」。例如 **instance** 記憶體是指每個 VM 專有一份」的記憶體，**instancing** 或 **instanced** 則是指爲完成「每個 VM 專有一份」而做的動作。

Instance 記憶體是一種「雖爲 global，但在每個 VM 中內容各不相同」的記憶體。所謂的 **command recall buffer** 就是個好例子，這種東西存在於像 DOSKEY 之類的程式中。你在某個 MS-DOS prompt 中所鍵入的命令並不會出現於另一個 MS-DOS prompt，但是 DOSKEY 是一個常駐程式，佔用被所有 VMs 共享的 global 記憶體。如果讓 DOSKEY 的 recall buffer 成爲「每個 VM 特有一份」，所有的 VMs 在存取同一位址時就會獲得不同的（屬於自己的）資料內容。

我知道有五種不同的方法可以做出上述的 instance 記憶體。在 VxD 的保護模式初始化期間（也就是說在 *Init_Complete* 訊息發生之前和發生期間），你可以呼叫 *_AddInstanceItem*：

```
#pragma VxD_IDATA_SEG
static InstDataStruc ids = {0, 0, 0, 0, ALWAYS_FIELD};
ids.InstLinAddr = linaddr;
ids.InstSize = size;
_AddInstanceItem(&ids, 0);
```

這個呼叫動作會將從 *linaddr* 開始並且大小爲 *size*（bytes）的整個區域定義爲 instance data，使每一個 VM 在此有自己的一份私有內容。請注意，傳給 *_AddInstanceItem* 的 *InstDataStruc* 結構必須一直存在，直到 *Init_Complete* 訊息處理完畢，因此最簡單的作法就是在 initialization data segment 之中把該結構宣告爲 static。

你也可以在真實模式初始化函式（第 8 章）動作期間將記憶體定義爲 "instancing"。

如果「你希望成爲 "instancing" 的那塊記憶體」是一個完整的真實模式 device driver，你可以在處理 *Init_Complete* 訊息期間只呼叫 *DOSMGR_Instance_Device*。例如，爲了找出並 "instancing" 一個名爲 MS$MOUSE 的 device，你可以這樣做：

```
DOSMGR_Instance_Device("MS$MOUSE");
```

Windows 所認知的驅動程式和常駐程式都應該爲你省下麻煩，使你不必在 VxD 中在反應 INT 2Fh, function 1605h（startup broadcast）時設定 instance 記憶體。上述中斷回返時，ES:BX 指向一個由 **startup info** 結構所形成的串列，其中描述了被靜態載入的 VxDs，也描述了 instance 記憶體區域。舉個例子，Windows 認知的驅動程式應該包含類似下面的碼：

```
sis       db     3, 0              ; SIS_Version
          dd     0                 ; SIS_Next_Dev_Ptr
          dd     0                 ; SIS_Virt_Dev_File_Ptr
          dd     0                 ; SIS_Reference_Data
          dd     ids               ; SIS_Instance_Data_Ptr
```

```
ids       dd     inststart         ; FAR* to start of region
          dw     instlength        ; length of instance region
          dd     0                 ; fence to end instance list
...
inststart equ $                    ; start of instance area
...[This area of memory is instanced in each VM]
instlength equ $-inststart         ; length of instance area
...
          assume  ds:nothing, cs:@curseg
          align   4
org2f     dd     ?
int2f:    cmp    ax, 1605h         ; Windows startup?
          je     @F                ; if yes, skip ahead
chain2f:  jmp    [org2f]           ; chain to next handler
@@:       test    bx, 1            ; standard mode startup?
          jnz    chain2f           ; if yes, ignore
          pushf                    ; call next handler
          call    [org2f]          ; ...
          mov    word ptr sis+2, bx  ; build SIS chain
          mov    word ptr sis+4, es  ; ...
          mov    bx, cs            ; set ES:BX -> SIS chain
          mov    es, bx            ; ...
          mov    bx, offset sis    ; ...
          iret                     ; return to VMM32 startup
```

Windows 3.1 的 Virtual Mouse Driver（VMD）可以解釋數種這類方法。我將以早期的 VMD 爲例，因爲對 Microsoft 開發工具的用戶而言，它十分普及（Windows 3.1 DDK 中就有其完整的原始碼）。務請注意，Windows 95 的 VMOUSE 就截然不同，但爲了對 Windows 3.1 的滑鼠驅動程式回溯相容，才例外執行了一些相同的函式。

在處理 *Init_Complete* 訊息期間，VMD 發出一個 INT 2Fh, function 1607h（device call out），看看真實模式的滑鼠驅動程式是否是 Windows 所認知的品種。如果是，真實模式驅動程式一定已經因爲正確處理了 INT 2Fh, function 1605h（如前所述）而 "instanced" 了它自己！如果不是，VMD 會檢查看看是否真實模式的 INT 33h（mouse services）處理常式是個常駐程式（TSRs）。如果是，就呼叫 *_AddInstanceItem*，將內含 INT 33h 處理常式碼的那塊記憶體給 "instancing"！如果兩個檢查的答案都是否定的，VMD 就呼叫 *DOSMGR_Instance_Device* 做爲最後手段，將一個名爲 MS$MOUSE 的真實模式驅動程式給 "instancing" 起來。從這段描述你可以知道，一個滑鼠供應商可以讓 VMD 的

「instancing 企圖」無效 -- 只要它讓 hook INT 2Fh 的動作失敗，並且安裝一個不叫做 MS$MOUSE 的 MS-DOS 真實模式滑鼠驅動程式即可。

產生 instance 記憶體的第五個（也是最後一個）方法是，經由 *_Allocate_Global_V86_Data_Area* 的一個旗標參數來指定：

```
DWORD linaddr = _Allocate_Global_V86_Data_Area(nbytes, GVDAInstance);
```

配置一塊 global V86 資料區的主要理由是爲了和真實模式常駐程式通訊，所以通常你會在上一行程式碼之下，把線性位址轉換爲 segment:offset 型式。轉換邏輯十分簡單：

```
WORD segment = (WORD)(linaddr >> 4);
WORD offset  = (WORD)linaddr & 0xF;
```

爲了產生一個可以被 16 位元 Windows 程式使用的保護模式位址，請確定你的驅動程式是在 System VM 的某個保護模式程式的環境（context）中進行，並使用 *Map_Lin_To_VM_Addr*：

```
WORD selector;
DWORD offset;
PVMMCB hVM = Get_Cur_VM_Handle();
ASSERT(Test_Sys_VM_Handle(hVM) && (hVM->CB_VM_Status & VMSTAT_PM_EXEC));
Map_Lin_To_VM_Addr(linaddr, nbytes-1, &selector, &offset);
```

注意 Microsoft 官方並不鼓勵使用 *Map_Lin_To_VM_Addr*，因爲它總是會消耗掉一個 selector。Windows 自己以這個 service 建立起永久性的映射關係，映射到 400h, A0000h, B0000h, B8000h 等區域。通常，你會配置一個 global V86 資料區，是因爲你要在真實模式驅動程式和 VxD 之間共享記憶體。如果你要在共享關係中納入一個 Win16 DLL 驅動程式，產生一個指向共享記憶體的永久指標應該是比較完美的作法。

你可能會好奇 VMM 如何實作出 instance 記憶體。如果一個 V86 page 內含 instance data，VMM 每次執行 task switch 切換到新 VM 時，會把那個 page 所隸屬的 page table entry（PTE）標示爲 not present。只要任何人企圖碰觸那個 page，page-fault 處理常式就

會把「對 current VM 而言合適之 instance data 內容」先拷貝到 physical page 中。這麼一來，任何觀察或改變 instance data 的人，看到的總是正確的私有資料。當然啦，要藉由一個 page 的共享，讓 noninstance data 成爲 instance data，必須付出一些代價。不過比起「每次 task switch 時就把所有的 instance data 拷貝一次」，代價並不算重！

Device Memory

V86 region 中的每一個 page 都可能屬於某特定的 VxD 所有。舉個例子，local memory area 屬於 V86MMGR device 所有，各個和 video display 有關的 RAM areas 則屬於 Virtual Display Driver（VDD）所有。VMM 維護了一些 bit maps（由位元組成的對應表），用以描述 page 的被擁有狀況；它還提供一個簡單的預約措施，讓 VxDs 管理那些對應表。

爲了管理一個 page 的被擁有權，首先請你呼叫 *_Get_Device_V86_Pages_Array*，獲得一個 "reservation map"（預約對應表）的區域性拷貝。然後檢測其中適當的位元，看看是否其他某些 VxDs 預約了相同的 page。最後再呼叫 *_Assign_Device_V86_Pages* 完成真正的指定動作。以下程式片段就指定了從 000B8000h（標準 VGA 文字緩衝區的起始位址）開始的 page：

```
DWORD devpages[9];    // room for all 110h bits needed
if (_Get_Device_V86_Pages_Array(NULL, devpages, 0)
     &&!(devpages[0xB8 / 0x20] & (1 << (0xB8 % 0x20))))
   _Assign_Device_V86_Pages(0xB8, 1, NULL, 0);
```

每一個 VM 有一個區域性的指定對應表（assignment map）和一個全域性的指定對應表。「全域性指定」會施行於後續所有新產生的 VMs。一般而言 VxD 會在初始化期間， 也就是在處理 *Sys_Critical_Init*, *Device_Init* 或 *Init_Complete* 訊息期間，確定其「全域性指定」。至於區域性指定只施行於某特定的 VM 身上，而且往往發生於 *Create_VM* 的處理過程中。

V86 page 的擁有者有責任提供一些實際記憶體，用以備份（backup）虛擬位址。這就是三個 *xxxIntoV86* service 的設計原由。舉個例，V86 memory manager device 開放出一個名

爲 *V86MMGR_Allocate_V86_Pages* 的 services，SHELL device 用它來設定一個新 VM 的基底記憶體。這個 service 主要的功績是定義 VM 的 local memory region 的內容。要這麼做，V86MMGR 首先呼叫 *_PageAllocate* 配置出一塊線性位址空間，然後呼叫 *_MapIntoV86*，把這塊空間映射到 V86 region。在這個例子中，V86MMGR 對其所映射的 pages 做了個「區域性指定」。

再舉另一個例子，考慮從 000B8000h 開始的 VGA 文字緩衝區。Virtual Display Device（VDD）全域性地指定這塊記憶體以完成顯示幕虛擬化。它使用 *Hook_V86_Page* 來建立一個 page-fault 處理常式，當 VM 嘗試碰觸這塊記憶體時，該處理常式就接管控制權。如果 VM 正在執行一個視窗，display device 會呼叫 *_MapIntoV86* 將一塊 private region 的記憶體映射到 video RAM 位址上。這樣的映射使得數個 VMs 可以同時執行，每一個都認爲自己將資料（字元和屬性）放進 video RAM 時是直接寫入硬體。然而如果 VM 是以全螢幕方式執行，就真的允許直接存取硬體。

Upper Memory

任何努力，只要能夠突破 640KB 的封鎖，基本上 MS-DOS 應用程式開發者都有意願去嘗試。關於記憶體不足，一個較大的突破就是：在一部 Intel 80386 相容機器的 V86 模式中，真實模式應用程式也可以發生 paging 行爲。將一些原本不能夠被 MS-DOS 處理的 extended memory（譯註：1MB 以上）映射進入 V86 位址空間後，MS-DOS, BIOS, 以及其他真實模式程式就有可能使用機器上所安裝的 1MB 以上的記憶體了。當然啦，至此還是沒有跳脫 1.1MB 的如來佛掌！expanded memory manager（EMM）產品如 EMM386, QEMM, 386Max 等等，都會尋找 0A0000h 至 100000h 之間的位址空間漏洞。它們會產生一個 page table，以「實際即虛擬」的原則，映射真正的記憶體。於是就把 extended memory 映射到那些位址空間破洞中了。這些產品也會產生 upper memory blocks（UMBs），允許 MS-DOS 配置虛擬記憶體給自己及 MS-DOS 應用程式使用。

你或許以爲 expanded memory manager 已經是歷史陳蹟，不然！Windows 95 的整個景觀中還是有很大一部份由它構成，因爲網路廠商很難把它們的驅動程式搬移到保護模式

中。此刻我所使用的網路是一個需要幾乎 100KB 的真實模式驅動程式。我不知道爲什麼網路廠商不提供保護模式版本給 Windows 95 使用，但現實就是如此！安裝 EMM386 使我多得大約 80KB 的 conventional memory（譯註：640KB 以下），這將成爲某些真實模式程式能否執行的關鍵。

要將「對 paging 硬體或 UMBs 的控制」從 expanded memory manager 移轉到 Windows 手上，確實有點棘手。對 memory manager 而言，第一個步驟是使用 INT 2Fh, function 1605h（所謂的 startup broadcast）。回返之前，memory manager 可以將 CPU 切換回真實模式，也可以設定 DS:SI 暫存器，使它指向一個模式切換函式，於是 Windows 便可在較佳時機呼叫該函式，執行模式切換動作。Windows 使用另一個介面（是個未公開介面）來決定記憶體的目前狀態。基本上，V86MMGR 發出一個 IOCTL call 給 memory manager，要求取得一個資料結構，用以模仿被 memory manager 所建立的記憶體佈局[1]。

上述模仿時機對某些 VxDs 而言十分重要。直到 V86MMGR 在處理 *Sys_Critical_Init* 訊息過程中設妥「UMBs 的定址機制」前，遵循 MS-DOS 記憶體串鏈（從 conventional 記憶體到 upper 記憶體）其實並不安全。由於真實模式的 drivers 和 TSRs 或甚至 MS-DOS 自己，可能佔用 UMBs，所以甚至連「檢驗記憶體是否內含一個真實模式中斷服務常式」也不安全。V86MMGR 有一個非常高的初始化次序值（A0000000h），如果在面對 *Sys_Critical_Init* 訊息時你需要處理 1st MB 記憶體，你應該使用特殊的初始化次序常數 *UNDEF_INIT_TOUCH_MEM_INIT_ORDER*（A8000000h）。如果這個初始次序對你不適用，那就直到你收到 *Device_Init* 訊息再來存取那塊記憶體吧。

---

[1] 細節請參考 *Dr. Dobb's Journal* 於 1994/7 刊出的一篇文章："The Windows Global EMM Import Interface"，作者 Taku Okazaki。

存取 V86 記憶體

每一個 VM 擁有一個 4MB 線性位址空間，用來映射線性位址空間的 1st MB 區域。當某個 VM 成爲 active VM，VMM 就將這 4MB 區域的 page table entries 複製於位址 0。也就是說，當某個 VM 正執行時，其 V86 region 可以從線性位址 0 開始定址，也可以從獨一無二的 high-linear 位址來定址。Windows 完成這所謂雙重映射（dual mapping）的方法是，放置一個 32 位元指標到 page directory 中，因爲每一個 page directory entry（PDE）控制著 1024 個 page table entries（PTEs），相當於 1024 * 4096 byte 虛擬記憶體。VMCB 結構中的 *CB_Hight_Linear* 欄位內含這個 high-linear 基底位址。

面對一個位址，VxD 應該總是能夠經由其「high-linear 版」存取其 V86 記憶體。以下是一種經過證實的作法，存取 BIOS data area（40:0h）中記錄的 COM1 port 位址：

```
PVMMCB hVM = Get_Cur_VM_Handle();
WORD address = *(WORD *)(0x400 + hVM->CB_High_Linear);
```

限制存取 low memory 的原因是，VMM 除錯版一般而言並不將 1st MB 記憶體映射給 VxD 存取。這可以幫助你捕捉 NULL 指標，或是捕捉那些「欄位與 NULL 指標只有小量差移（offset）」的結構的 reference（譯註：此 reference 是指 C++ 中與指標相提並論的那個 reference；無適當譯詞，故保留原文）。如果你真的爲了某些因素需要使用 V86 位址，請在這個 reference 前後包夾 *Enable_Touch_1st_Meg* 和 *Disable_Touch_1st_ Meg*。若不直接呼叫 services，另一個替代方案是使用 DEBUG.H 中的兩個巨集：

```
#include <debug.h>
...
WORD address;
Begin_Touch_1st_Meg();   // enables first MB in debug build
address = *(WORD*)0x400;
End_Touch_1st_Meg();     // disables first MB in debug build
```

這些巨集的優點是，它們不會在 Windows 95 零售版（retail build）中產生任何碼。是的，零售版並不限制 VxD 存取 1st MB。

軟體中斷（Software Interrupts）

在 MS-DOS 的世界中，程式之間藉由發出軟體中斷或共享某些資料結構，來達到通訊目的。Windows 3.1 繼續使用 MS-DOS 和 BIOS 中斷，這一點恰足以反映出其源自真實模式。16 位元 Windows 程式還是使用軟體中斷做爲系統主要的程式介面 。我已經在第 5 章和第 6 章描述過一個中斷如何經由 CPU 的中斷描述表格（interrupt descriptor table），在保護模式中被導引到中斷處理常式去。我也提示了一些 VxD services，允許我們安裝自己的中斷處理常式。這一節，我要繼續討論軟體中斷的處理。

我即將說的任何內容，沒有一個適用於 Win32 程式。32 位元 Windows 程式使用的是 Win32 API，完全建立在「函式呼叫」的基礎上。在 KERNEL32 最核心處，有一個費盡心力完成的反推引擎，可以找出「對 VWin32 VxD 的呼叫動作」其內所呼叫的一個或數個 ring0 services。 Win32 程式之於軟體中斷，非「不爲」也，根本是「不能」也。VMM 相信任何來自 System VM 的中斷，必定是來自 16 位元程式，並因此只儲存（和恢復）CPU 的部份狀態。

於是，此處討論只與 16 位元 Windows 程式、MS-DOS 程式（也是 16 位元程式）以及 extended DOS 程式（可能是 16 位元或 32 位元程式）有關。這類程式不再是現今的商業焦點，但爲了回溯相容，Windows 必須在一段時間內繼續支援它們。

爲了讓事情更具體化，我假設你要開發一個虛擬滑鼠驅動程式（VMD）。在 MS-DOS 中，你可以使用 INT 33h 來和真實模式滑鼠驅動程式交談。然而，做爲一個 VxD 撰寫者，你的工作之一是要處理來自保護模式程式的 INT 33h。在 Windows 3.1 中，Windows 滑鼠驅動程式（MOUSE.DRV）使用 INT 33h。在 Windows 95 中，extended DOS 程式還是可以繼續使用 INT 33h。

攔截一個中斷（Hooking an Interrupt）

如果想要攔截（hook）保護模式軟體中斷，*Set_PM_Int_Vector* 是正確的選擇。這個 service 直接在 IDT 中安裝一個 gate。如果你在 *Sys_VM_Init* 訊息產生出來之前使用這個

service，你就可以爲每個 VM 指定一個預設處理常式。否則你就只能夠處理 current VM。真正的 Virtual Mouse Driver 是在處理 *Sys_Critical_Init* 訊息時安裝其 INT 33h 處理常式，所以如果沒有其他人攔截（通常如此），它就是「hook 串鏈」中的最後一個。如果你的中斷使用了任何指標參數，而此指標需要從保護模式轉換到真實模式，並且如果其他 VxD 處理 *Device_Init* 訊息時使用 *Exec_VxD_Int* 喚起你的轉換碼（translation code），那麼你應該在處理 *Sys_Critical_Init* 時安裝一個處理常式。

你還應該利用 *Set_PM_Int_Vector* 將一個 ring3 處理常式的位址安裝到 IDT 中。你可以安裝一個 ring0 位址，但這麼一來你的碼就有責任設定 ring0 執行環境，就像 VMM 第一層中斷處理常式的所做所爲一樣（譯註：所謂第一層中斷，第 6 章有說明）。爲了攔截中斷，俾能夠在 VxD 中處理它，你應該利用 *Allocate_PM_Call_Back* 產生一個 object，稱爲「保護模式 callback」：

```
mov     edx, refdata             ; arbitrary pass-through data
mov     esi, offset32 i33callback     ; address of callback routine
VMMCall Allocate_PM_Call_Back         ; create callback
jc      error                         ; handle error
```

然後將 IDT 導引到「保護模式 callback」的位址：

```
movzx   edx, ax                       ; cx:edx = first level handler
mov     ecx, eax                      ; ...
shr     ecx, 16                       ; ...
mov     eax, 33h                      ; eax = interrupt number
VMMCall Set_PM_Int_Vector             ; install default INT 33h handler
```

上一段碼將 *Allocate_PM_Call_Back* 的 32 位元傳回值（EAX）分解爲 selector:offset 型式，放在 CX:EDX 中，做爲 *Set_PM_Int_Vector* 的參數，代表你的中斷處理常式的位址。完成這兩個動作之後，任何時候 ring3 保護模式程式只要執行一個 INT 33h 指令，VMM 就會把控制權移交到你的 *i33callback* 常式，暫存器內容則如**表 10-2** 所示。Callback 常式可以改變任何 general 暫存器，並使用任何 VxD service。雖然 callback 常式用來處理中斷，但此中斷並非硬體中斷，因爲硬體中斷會有殘酷的非同步（asynchronous-only）限制。

暫存器	內容
EBX	current VMCB 的位址
EDX	*Allocate_PM_Call_Back* 時指定的 reference data
EDI	current TCB（thread control block）的位址
EBP	client register structure（CRS）位址。CRS 內含「當某個 VM 呼叫 callback 常式時，該 VM 之各暫存器值」。

表 10-2　進入一個保護模式 callback 常式時，暫存器的內容

以實用角度來說，callback 函式必須以 assembly 語言來寫，才能夠使用暫存器參數。如果你要，你可以爲你的 callback 常式寫一個 C wrapper（外包函式）：

```
        extern  _OnInt33@16:near
BeginProc i33callback, pageable
        push    ebp             ; client registers
        push    edi             ; current thread
        push    edx             ; reference data
        push    ebx             ; current VM
        call    _OnInt33@16     ; handle interrupt
        ret                     ; return to caller

void __stdcall OnInt33(PVMMCB hVM, DWORD refdata, PTCB tcb, PCRS pRegs)
{
...
}
```

這種方法並不是最好，因爲用來將指標轉換至真實模式的那些 API 函式，在高階語言中並不好使用。我將以 assembly 語言繼續完成本章範例。

「V86 模式 callbacks」和「保護模式 callbacks」

「保護模式 callback」是一個「允許 ring3 保護模式程式碼 V 86 ring0」的 object。呼叫 *Allocate_PM_Call_Back* 時，你得提供一個目的函式的位址，及一份 reference data。VMM 會將位於 segment 3Bh 的一群 INT 30h 指令中的一個，指定給你。Segment 3Bh 內含的全都是 INT 30h 指令。當 ring3 程式執行一個 INT 30h，VMM

就把控制權繞行（routes）到你的函式去了，並將 reference data 以參數傳遞過去。

有了保護模式 callback 位址，你可以達成兩件事情。第一，攔截（hooking）ring3 中斷須使用一個 callback 位址。第二，你可以將 callback 位址交給應用程式，當做函式指標。當應用程式使用這個指標，它就執行了 INT 30h 並且而讓控制權轉到你的去了。「保護模式 callback」於是給予你一種能力，相當於為 ring3 應用程式提供一個 ring0 函式。VMM 內部另為了其他理由使用「保護模式 callbacks」，例如 VMM 使用一個保護模式 callback 位址，以便在 ring3 碼鼠狀執行（nested execution）之後，重獲控制權。當應用程式使用 INT 2Fh, function 1684h 尋找一個 VxD 的保護模式 API 位址時，VMM 也會產生一個保護模式 callback。

VMM 利用 V86 callbacks 提供類似的一組 services 給 V86 程式使用。你可以使用 V86 callbacks 來攔截（hook）V86 中斷，並讓 V86 程式呼叫你。你可以呼叫 *Allocate_V86_Call_Back* 來產生一個 V86 callback，獲得的 segment:offset 位址視同一個 ARPL（adjust requested privilege level）指令。在 V86 模式中執行 ARPL 指令會引起一個 **Invalid Operation** 異常（INT 06h）。根據這個異常的位址，VMM 可以辨識出此 callback 屬於你，並將控制權繞行（route）給你的 ring0 callback 函式。某種情況下，ARPL 指令事實上是 BIOS 的一個字母，只不過是湊巧有正確的 "bit pattern" -- 碰巧是 63h，也就是字母 c。如果你奇怪 *Locate_Byte_In_ROM* service 是做什麼用的，它其實就是用來搜尋 BIOS，尋找 63h bit pattern，因為此 bit pattern 可用來擋住另一個名叫 ：ARPL。

讓我再重述一次重點。剛剛那個程式片段和「在 MS-DOS 中使用 INT 21h, function 35h，或是呼叫 *_dos_setvect* 攔截一個中斷」差可比擬。當 VM 中的保護模式碼發出我們所攔截（hook）的中斷時到底會發生什麼事，圖 **10-2** 有不錯的解釋。此一中斷最初會被誘導到 INT 30h 指令，而指令所需的位址則是我們呼叫 *Set_PM_Int_Vector* 時所安裝。VMM 使用 INT 30h 指令中的位址，找出我們的 *i33callback* 常式。當處理常式回返，VMM 會 "redispatch" VM（使用 CS:IP 位址）。如果我們的 callback 常式沒有加以干涉，同一個 breakpoint INT 30h 會一再一再地執行起來。爲了避免無窮迴路發生，callback

必須使用 *Simulate_Iret* 將 VM 重新導到原 INT 33h 之後的指令：

```
BeginProc i33callback, pageable
VMMCall Simulate_Iret    ; arrange for VM to return
...
EndProc i33callback
```

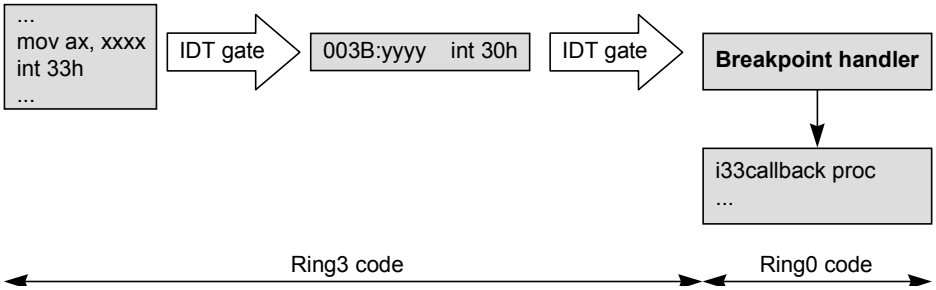


圖 10-2　保護模式中斷過程中的控制流程

將中斷反射出去（Reflecting the Interrupt）

你可以在各式各樣的地方找到 INT 33h API 的相關說明，例如 Ralf Brown & Jim Kyle 的 *PC Interrupts*（Addison Wesley，1991）。讓我們先看一個最簡單的功能：

```
mov   ax, 2     ; function 2 : hide mouse cursor
int   33h       ; issue mouse function call
```

就像上面的註解所示，呼叫滑鼠驅動程式時，暫存器 AX 必須放置功能代碼。#2 功能是把滑鼠游標隱藏起來。如果缺乏一個真實模式滑鼠驅動程式，ring0 元件必須在保護模式中完全處理這個中斷。然而如果使用者安裝了一個真實模式驅動程式，Windows 會將滑鼠控制權轉移給它（例如硬體滑鼠可能會要求某特定軟體，而該軟體卻尚未移植到 Windows 95）。將中斷送到真實模式，稱爲 **"reflecting"**。

將中斷 "reflecting"，其機制中隱含 nested execution（巢狀執行）的觀念。你可以呼叫 *Begin_Nest_V86_Exec*（執行真實模式碼）或 *Begin_Nest_Exec*（執行保護模式碼）以建立

一個巢狀執行區塊（nested execution block）。此例我將使用 *Begin_Nest_V86_Exec*，因爲我們要執行真實模式驅動程式，完成「隱藏游標」的目的。這個 service 會爲 client 暫存器拷貝一份副本，並設定虛擬的 CS:IP，使它指向一個 V86 callback 位址。你可以使用 *Exec_Int* service 執行真實模式中斷處理常式。*Exec_Int* 將 client flags 和 instruction pointer（IP）推入真實模式堆疊之中，並改變 client 的 CS:IP，指向某一中斷的真實模式處理常式。然後它引發 VMM 迅速處理（dispatch）VM，於是執行起真實模式中斷處理常式。真實模式碼執行一個 IRET，回返到屬於巢狀執行管理器（nested execution manager）的一個 V86 中斷點（breakpoint），於是你在 *Exec_Int* 之後重新獲得控制權。最後，你應該使用 *End_Nest_Exec* 恢復 VM 狀態，回到巢狀執行區塊（nested execution block）被執行前的狀態。欲執行上述這些步驟，你的 INT 33h 處理常式大約需要以下內容：

```
VxD_PAGEABLE_DATA_SEG
i33func label   dword
       dd     offset32       i33_00
       dd     offset32       i33_01
       dd     offset32       i33_02
       ...
VxD_PAGEABLE_DATA_ENDS

BeginProc i33callback, pageable
       VMMCall Simulate_Iret             ; arrange for VM to return
       movzx   eax, [ebp + Client_AX]    ; get function code
       jmp     [i33func + 4 * eax]       ; dispatch handler
       ...
i33_02: call reflect                     ; hide cursor
       jmp   done
       ...
done:   ret
EndProc i33callback

BeginProc reflect, pageable
       VMMCall Begin_Nest_V86_Exec
       mov     eax, 33h
       VMMCall Exec_Int
       VMMCall End_Nest_Exec
       ret
EndProc reflect
```

你可以把 *Exec_Int* 想像是一個在真實模式中執行的函式，因爲在 VM 回返之前，你沒辦法重新獲得控制權。事實上 *Exec_Int* 處理任何等待中的 events，並且可以觸發 task switch，切換到其他 VM。換句話說在 VM 真正執行中斷之前，可能需要一段長時間。如果時間很緊要，你可以使用 *Adjust_Exec_Priority* 來提昇 VM 的優先權以強化其留住控制權的機會。

雖然此例是正確的，但它卻粉飾了你需要知道（關於何時使用巢狀執行區塊, nested execution block）的某些複雜度及一般性。第一件值得注意的事情是，真實模式中斷處理常式對 general 暫存器的任何改變都將波及保護模式中的 client 端。如果中斷處理常式將新值覆寫於原來的暫存器上，必然是故意的，因爲真實模式處理常式將結果提供於 general 暫存器之中。如果你不希望「"nested" 程式」改變 general 暫存器，你可以在巢狀執行區塊（nested execution block）的前後儲存它們並恢復之：

```
...
Push_Client_State          ; save registers
VMMCall Begin_Nest_V86_Exec
...
VMMCall End_Nest_Exec
Pop_Client_State           ; restore registers
```

Push_Client_State 是一個巨集，會調整堆疊指標，留出一塊資料區域空間，並將位址傳遞給 *Save_Client_State* service。*Pop_Client_State* 是另一個巨集，它會呼叫 *Restore_Client_State* 並 "pops" 堆疊，使堆疊回到原先位置。你應該成雙成對地使用這兩個巨集，如上所示。

另一件你需要知道的事情是，有一個比 *Exec_Int* 更一般化的 service，名曰 *Resume_Exec*。*Resume_Exec* 處理 events 並迅速處理（dispatches）VM，當 VM 執行其中斷點（breakpoint）時便回返到你的 driver。爲了讓 *Resume_Exec* 有用，你需要事先使用其他一些 services 來處理 client 暫存器和堆疊（表 **10-3**）。例如，你可以呼叫 *Simulate_Int* 和 *Resume_Exec* 來實作 *Exec_Int*。如果不是因爲 *Simulate_Int* 和 *Exec_Int* 會導至一個重大副作用（它們會呼叫被 *Hook_V86_Int_Chain* 設立的 hook 函式），你

可以使用下面這份稍嫌累贅但「不可能被切割（也就是所謂 atomic）」的碼：

```
mov     eax, 33h                ; eax = interrupt number
VMMCall Get_V86_Int_Vector      ; cx:edx = real-mode INT 33 vector
VMMCall Build_Int_Stack_Frame   ; build IRET frame using cx:edx
VMMCall Resume_Exec
```

或是使用下面這樣荒謬的動作來「重新發明輪子」：

```
mov     ax, [ebp + Client_Flags]
VMMCall Simulate_Push
mov     ax, [ebp + Client_CS]
VMMCall Simulate_Push
mov     ax, [ebp + Client_IP]
VMMCall Simulate_Push
mov     eax, 33h
VMMCall Get_V86_Int_Vector
mov     [ebp + Client_CS], cx
mov     [ebp + Client_IP], dx
VMMCall Resume_Exec
```

意外地是，所有這些暫存器模擬服務函式（register simulation services）都能夠認知記錄於 VMCB 中的「應用程式 bitness」（16 或 32 位元），並採取適當行爲。如果 VMM 認爲某個 VM 正在執行 16 位元 DPMI client（例如 Windows），那麼不管 client 堆疊以及 code segments 的真正 bitness 是什麼，*Simulate_Push* 會將 2 bytes 推入 SS:SP 堆疊之中（正確的行爲其實是將 2 或 4 bytes（視 client CS 是 USE16 或 USE32 而定）推入至 SS:SP 或 SS:ESP（視 client SS 是 USE16 或 USE32 而定））。這是因爲 VMM 既然認定 System VM 內含 16 位元保護模式程式，那麼你應該只爲了與 16 位元程式打交道才使用那些處理 client 暫存器的 services。

Service	說明
Build_Int_Stack_Frame	將 VM 佈置妥當，準備執行一個中斷服務常式。
Simulate_Far_Call	佈置 CS:IP 和堆疊，就像 VM 執行了一個 far CALL 指令似的。
Simulate_Far_Jmp	佈置 CS:IP，就像 VM 執行了一個 far JMP 指令似的。
Simulate_Far_Ret	佈置 CS:IP 和堆疊，就像 VM 執行了一個 RETF 指令似的。

Simulate_Far_Ret_N	佈置 CS:IP 和堆疊，就像 VM 執行了一個 RETF n 指令似的。
Simulate_Int	呼叫 hook 函式，然後佈置 CS:IP 和堆疊，就像 VM 被中斷似的。
Simulate_Iret	佈置 CS:IP 和堆疊，就像 VM 執行了一個 IRET 指令似的。
Simulate_Pop	佈置堆疊，就像 VM 執行了一個 POP 指令似的。
Simulate_Push	佈置堆疊，就像 VM 執行了一個 PUSH 指令似的。

表 10-3　用來操作 client registers 的一些 services

最後，當你使用 *Begin_Nest_Exec*（不是 *Begin_Nest_V86_Exec*）爲保護模式碼啓動一個巢狀執行區塊，你也可能會在 VM 之中引起堆疊切換。VMM 爲每一個 VM 維護一個特殊的 4-KB locked stack，用於巢狀執行之中。此 locked stack 的主要目的是允許虛擬的異常處理常式（virtual exception handlers）和 DPMI callbacks 忽略掉 paging。所以不論你何時發出一個 *Begin_Nest_Exec* call，VMM 會將 VM 切換到這個 locked stack 上 -- 除非 VM 本就已經使用了它。

Exec_VxD_Int 和 Exec_PM_Int

VMM 內含另外個 services，可以執行某個 VM 的軟體中斷處理常式。*Exec_VxD_Int* 讓你發出一個 inline call，呼叫真實模式中斷處理常式。你應該爲此中斷適當地填寫 general 暫存器內容，然後將中斷號碼推入堆疊，然後呼叫這個 service。但是你不可以改變 segment 暫存器。舉個例子，如果真實模式中斷處理常式希望 ES:BX 持有某些指標，那麼就簡單地將該指標的 flat 版本放到 EBX 就好。

經由某些計策，*Exec_VxD_Int* 模擬中斷的方法使得 VxD 處理常式有機會轉換指標參數，並將此中斷映射（reflect）到真實模式。其行爲相當於一個 32 位元保護模式應用程式在 current VM 中發出中斷。由於這個 service 在 *Device_Init* 過程中可用，許多 VxD 系統組件用它來和真實模式的影像們交談。

你可能已經在其他地方聽人家說過，*Exec_VxD_Int* 不是非常有用，因為 *Begin_Nest_Exec* 有一個潛伏的 bug。如果以下兩個條件為真，這個 bug（或說弱點，或什麼形容詞都好）就會曝露出來：

- current VM 不含有一個保護模式應用程式。除非使用者啟動一個以 DOS extender 完成的應用程式，否則便沒有任何 MS-DOS VM 內含保護模式應用程式。同樣道理也發生在 System VM 中，一直到 Windows KERNEL 使用了 DPMI 的模式切換函式進入保護模式。

- 你所執行的中斷，其 VxD 處理常式做了一個 *Begin_Nest_Exec* 動作來執行保護模式碼。*Begin_Nest_Exec* 在此處並非十分有用，因為這些中斷處理常式通常會呼叫 *Begin_Nest_V86_Exec*，將其中斷映射（reflect）到真實模式。其實不必憂心其可能性，因為如果中斷處理常式決定停滯（block）於一個 semaphore 之上，某些類似 *Begin_Nest_Exec* 的事情就將發生。

這種情況不應該發生在 Windows 95 身上（但可能發生在 Windows for Workgroups 3.11 的一個 *Init_Complete* 訊息處理常式中 -- 如果其 32 位元檔案存取能力被 enabled 的話）。Microsoft 認為 *Exec_VxD_Int* 夠安全，所以它自己的系統元件在 *Device_Init* 期間及其後面的時間，都自由使用這個 service。

另一個中斷模擬服務是 *Exec_PM_Int*。*Exec_PM_Int* 就像 *Exec_VxD_Int*，只不過你應該使用 current VM 的 client register structure（而非真正的 general 暫存器）做為參數。它常常被當做一種用來幫助 32 位元 ring3 程式發出中斷的方法；也常被當做另一種方法，保護自己免受某些「未將 32 位元暫存器正較高半部保存下來」的真實模式碼牽連。

這兩個 services 都在 *Exec_Int* 呼叫期間於 current VMCB 內設立 *VMSTAT_VXD_EXEC* 狀態旗標。雖然一個保護模式中斷處理常式應該不必關心它所處理的中斷的發源處，但如果真的需要區分中斷來自 VM 或來自 VxD，它可以檢閱這個旗標。

> Windows 95 DDK 說明文件中要求，你為保護模式中斷所安裝的任何處理常式，都會在 *Sys_Critical_Init* 訊息處理期間被裝設妥當。這項要求的理由是，比你的 VxD 更早初始化的其他那些 VxDs，可能在 *Device_Init* 期間開始呼叫 *Exec_VxD_Int*，而在那個時刻，傳入來的處理常式必須已經就定位。

為真實模式做指標轉換

如果你所要做的就只是將你的中斷映射（reflect）到真實模式，其實沒有理由寫自己的 hook，因爲中斷的映射是預設行爲。然而許多中斷介面內包含有遠程指標參數，所以你需要供應一個位址轉換服務。轉換是必須的，因爲保護模式程式使用 selector:offset 來定址記憶體中的資料，而真實模式程式要求的是 segment:offset 位址（於 1st MB 中）。最好的策略就是把指標所指的資料拷貝一份到 V86 記憶體中，然後產生一個指標指向這副本。從真實模式回返時，再把任何改變過的資料拷貝回原來的緩衝區中。

處理指標參數，最容易的方法就是使用專爲此目的而設計的 V86MMGR services （請看表 **10-4**）。V86MMGR 在 low memory 區控制一塊轉化緩衝區（translation buffer），它將資料拷貝到（或從）此緩衝區中，以回應其他 VxDs 的請求。你可以明白呼叫轉換用的 services，或建立一個轉換用的 script（描述文字）給 *V86MMGR_Xlat_API* 執行。透過 *V86MMGR_Set_Mapping_Info*，你可以將你對轉換緩衝區空間的需求通知 translation service manager。雖然 DDK 說，收到 *Init_Complete* 訊息之前你不可以使用 V86 translation services，Microsoft 自己的 drivers 卻在 *Device_Init* 期間呼叫它們。所以，我猜想我們也可以這麼做。

Service	說明
V86MMGR_Allocate_Buffer	在轉化緩衝區（translation buffer）中保留空間
V86MMGR_Free_Buffer	釋放轉化緩衝區（translation buffer）中的空間
V86MMGR_Get_Xlat_Buff_State	取得轉化緩衝區（translation buffer）的相關資料
V86MMGR_Load_Client_Ptr	取得 client 緩衝區的位址
V86MMGR_Set_Mapping_Info	指定轉換服務需求。

V86MMGR_Set_Xlat_Buffer_State	切換到一個新的轉化緩衝區（translation buffer）
V86MMGR_Xlat_API	直譯（interprets）「轉化描述文字（translation script）」

表 10-4　V86MMGR 提供的指標轉化服務

使用這些 services 的第一個步驟就是呼叫 *V86MMGR_Load_Client_Ptr* 來設定 FS:ESI 暫存器，指向一個保護模式資料區的位址，此區內容正是你打算拷貝到真實模式去的。接下來使用 *V86MMGR_Allocate_Buffer* 在轉換緩衝區中保留空間並將 client 資料拷貝進去。真實模式程式完成任務之後，你就使用 *V86MMGR_Free_Buffer* 將資料拷貝回到 client 緩衝區去，並釋放轉換緩衝區中的空間。

舉個例子，考慮一下滑鼠功能 09h，它用來定義滑鼠游標的圖形 bitmap 。這個函式使用 ES:DX 指向新的 bitmap。你可以下列程式碼掌握這個函式：

```
BeginProc i33callback, pageable
        ...
i33_09:  push    [ebp + Client_EDX]      ; save client's EDX (1)
        push    dword ptr [ebp + Client_Alt_ES]             ; (2)
        mov     ax, (Client_ES * 256) or Client_DX
        VxDCall V86MMGR_Load_Client_Ptr  ; FS:ESI ->> bitmap
        mov     ecx, 64                  ; bitmap is 64 bytes
        stc                              ; copy data to buffer
        VxDCall V86MMGR_Allocate_Buffer  ; reserve space and copy
        mov     [ebp + Client_DX], di    ; EDI = pointer to copy
        shr     edi, 16                  ;   ..
        mov     [ebp + Client_Alt_ES], di ; ..
        call    reflect                  ; reflect to real mode
        clc                              ; don't copy back
        VxDCall V86MMGR_Free_Buffer      ; release ECX bytes
        pop     dword ptr [ebp + Client_Alt_ES]     ; (2)
        pop     [ebp + Client_EDX]       ; restore EDX (1)
        jmp     done                     ; done with function 09h
```

現在我要一步一步把這段碼走一遍。一開始，VM 處於保護模式，這是我們即將呼叫之 V86MMGR services 的一個必要條件。由於我們將暫時改變 client DX 暫存器，使它指向一個滑鼠游標（圖形）的拷貝本，所以我們先把 DX 值儲存在我們的 ring0 堆疊之中。

倒是沒有必要將整個 32 位元 EDX 都儲存下來，因爲真實模式 driver 不應改變此暫存器的較高半部。不過在 VxD 中處理 32 位元數值總是比較快些，況且多儲存 EDX 的上半部並不會帶來什麼傷害。而且，有些真實模式碼還真的會改變擴充（extended）暫存器的較高半部呢！*V86MMGR_Load_Client_Ptr* service 使用 AH 和 AL 來索引 EBP 所指出的 client register structure（CRS），並企圖在 FS:ESI 中組成一個位址。此處我們需要的是位址的保護模式對等品（記錄於 client ES:DX 中）。

這段碼之中，決定性的呼叫動作是 *V86MMGR_Allocate_Buffer*。我們將 ECX 設定爲即將傳遞給真實模式的資料長度（bytes）。設立 carry flag，表示我們要把資料從 FS:SI 所指的緩衝區拷貝到轉化緩衝區（translation buffer）。EDI 中的傳回值是一個遠程指標，指向資料的真實模式副本。我們把 EDI 存放到 client ES:DX 之中。由於 VM 還處於保護模式，主要的（**primary**）client ES（*Client_ES*）是個保護模式 selector，被目前的呼叫者使用，因此我們改變候補的（**alternate**）client ES 暫存器。*Begin_Nest_V86_Exec* 會將 "primary" 和 "alternate" 的內容對調。General 暫存器（例如 *Client_DX*）倒是只有一份。

V86MMGR_Allocate_Buffer 可能會失敗。如果回返時 carry flag 設立，就表示失敗。它也可能保留比我們所要求的數量更少的緩衝區空間，這可以從回返後的暫存器 ECX 值得知。我們應該檢測出這些失敗情況並彌補之，才是穩健的設計。

設定好轉化緩衝區（translation buffer）後，我們呼叫 *reflect* 來實際執行真實模式中斷。回返時 ECX 仍然持有我們在轉化緩衝區中保留的長度（bytes），而 VM 則回到了保護模式。我們呼叫 *V86MMGR_Free_Buffer*，將指定於 ECX 中的 bytes 個數從轉化緩衝區的堆疊中推出（pop）。事先清除 carry flag，表示我們不希望令真實模式緩衝區的內容又被拷貝回保護模式。

最後，我們可以恢復被我們改變的 client 暫存器，並離開中斷處理常式。

這個例子示範出如何保留轉化緩衝區（translation buffer）空間，以及如何從保護模式拷貝一個資料參數到真實模式。另有其他滑鼠相關服務（如 function 16h, Save Driver State）

的動作反方向而行。也就是說它們提供一個指標，指向一塊由真實模式 driver 負責填寫的緩衝區。這種情況下你需要呼叫 *V86MMGR_Allocate_Buffer*，將 carry flag 清除並且沒必要設定 FS:ESI。當你呼叫 *V86MMGR_Free_Buffer*，你首先將 FS:ESI 設定爲保護模式緩衝區位址，並設立 carry flag，表示你希望資料能夠拷貝回到保護模式。

另一個方法是建立一個 script 給 *V86MMGR_Xlat_API* 執行，這可以讓你自己明白呼叫 V86MMGR 提供的「指標轉換服務函式」。script 內含用以表現 runtime calls 的動作碼（operation codes），但它以一種比較緊湊而易讀的型式對那些呼叫做了封裝。**表 10-5** 列出用來建立 script 的一些 V86MMGR.INC 巨集。下面的程式片段說明如何使用它們來處理 INT 33h, function 09h （這是稍早出現過的例子）：

```
VxD_PAGEABLE_DATA_SEG
xlat_09  label   byte
        Xlat_API_Fixed_Len es, dx, 64
        Xlat_API_Exec_Int 33h
VxD_PAGEABLE_DATA_ENDS

BeginProc i33callback, pageable
        ...
 i33_09: mov     edx, offset32 xlat_09
        VxDCall V86MMGR_Xlat_API
        jmp     done                    ; done with function 09h
        ...
EndProc  i33callback
```

巨集	說明
Xlat_API_ASCIIZ	轉換一個以 null 爲結束字元的字串（從保護模式到真實模式）
Xlat_API_ASCIIZ_InOut	轉換一個以 null 爲結束字元的字串（雙向）
Xlat_API_Calc_Len	轉換一個緩衝區，其長度可由你所提供的函式決定之（雙向）
Xlat_API_Exec_Int	在一個巢狀的 V86 執行區塊（nested V86 execution block）中執行 *Exec_Int*
Xlat_API_Fixed_Len	轉換一個固定長度的緩衝區（雙向）
Xlat_API_Jmp_To_Proc	將 script processor 的控制權轉移到你提供的函式身上，以實施特殊處理。

Xlat_API_Return_Ptr	產生某一指標（由真實模式程式傳回）的保護模式版本。
Xlat_API_Return_Seg	產生一個 selector，映射到一個由真實模式程式傳回的 segment 位址。
Xlat_API_Var_Len	轉換一個緩衝區，其長度記錄於某暫存器中（雙向）

表 10-5　V86MMGR 的 translation API 巨集

這些「轉換用的 API 巨集」比較容易使用，原因是：它們儲存並恢復任何它們所使用的 client 暫存器。它們自動發出必要的 *Begin_Nest_V86_Exec* 和 *End_Nest_Exec* calls，圍繞在 *Exec_Int* call 前後。它們並且將「使用 FS:ESI 做爲一個緩衝區指標」的前後惡作劇式的動作隱藏了起來。這些便利性使得以 C 來撰寫中斷處理常式似乎頗爲合理，但你必須寫自己的 C 語言巨集來產生 "translation script"。

函式指標

有一些 MS-DOS API 函式會使用函式指標。INT 21h, function 38h（Get Country-Specific Information），就是將一個「大寫轉換函式」的函式位址（以及其他東西）填入某資料結構。INT 33h, function 0Ch 則是供應一個「mouse event callback 函式」的位址。這類函式都需要一個轉換用的 VxD 做額外工作，因爲很明顯真實模式程式不能夠直接呼叫保護模式函式，反之亦然。

爲了處理第一種情況（真實模式系統碼傳回一個函式指標），你必須在回返保護模式的 client 程式之前，以一個保護模式位址取代之。最實際的經驗就是產生一個保護模式 callback 函式。保護模式 client 程式經由函式指標，會呼叫到這個 callback 函式，此函式模擬一個遠程呼叫（far call），呼叫位於巢狀執行區塊（nested execution block）中的一個真實模式函式。由於「保護模式 callback objects」有個數限制，而且沒有辦法釋回，我想你或許不會想要在每次處理中斷的時候都配置一個新的；你應該記住此函式的真實模式位址。INT 21h function 38h 是個特殊案例，由於其對「大小寫變換函式」的位址轉化有著甚低的效用，以至於 Windows 完全不管事，只在傳回的結構中留下真實模式位址。誰想呼叫它，誰去傷腦筋！

處理第二種情況（由保護模式程式提供一個函式指標）也是一樣棘手。你得在呼叫真實模式時以一個 V86 callback 位址取代你原本儲存的保護模式位址。一旦 callback 發生，你就使用 *Begin_Nest_Exec* 強迫 VM 進入保護模式，並模擬一個遠程呼叫（far call），呼叫保護模式函式（其位址已經被你記下）。

V86 中斷

V86 程式的軟體中斷經由 IDT 被導引到一個 VMM 處理常式去，通常會被轉向並映射（reflects）回到 V86 模式。做爲一個 VxD 撰寫者，你有兩種方法可以影響 events。你可以使用 *Set_V86_Int_Vector* service 來改變 0:0 中斷向量表。配合這個 service，你必須安裝一個 V86 遠程指標。如果你嘗試在一個 VxD 內處理中斷，你必須產生一個 V86 callback 函式並安裝其位址：一個真實模式的 segment:offset 位址。

另一個比較方便的 V86 中斷處理方法是使用 *Hook_V86_Int_Chain* service。任意多個 VxDs 都可以爲一個「V86 中斷」註冊其私人的 hook 函式。 hook 函式可以檢驗與此中斷相關的參數，並決定「燒毀」它（清除 carry flag 並回返）或是把它交給 hook 串鏈（hook chain）中的下一個 hook 函式（回返並令 carry flag 設立）。如果沒有任何 hook 函式「燒毀」此中斷，VMM 就把它映射（reflects）到 V86 模式中由 0:0 中斷向量所指定的位址上。Hook 函式也可以對於中斷 "hook the back end"，方法是：使用 *Call_When_VM_Returns* 建立一個 callback 函式；當 VM 終於執行了一個 IRET 指令，從處理常式回返，callback 函式便可獲得控制權。總的來說，這些性質允許 VxD 檢閱、修改、或完全實作出一個 V86 中斷。

VxDs 可以使用 *Simulate_Int* 或 *Exec_Int* services 來產生 V86 中斷，這兩個 services 送出的中斷，在呼叫虛擬處理常式（virtual handler）之前，都會先通過 V86 hook chain。

舉個例子，Windows 3.1 Virtual Mouse Driver 需要知道真實模式碼下什麼指令給真實模式滑鼠驅動程式，才能正確地將它們轉換爲寫實的 mouse events。因此它必須攔截（hooks）V86 INT 33h chain。大部份情況下，VMD 的 hook 函式只是從中斷參數萃取出一些資訊，然後就將中斷交給真實模式驅動程式去服務。然而當面對一個不具 Windows 意識

的真實模式驅動程式時，它會濾除來自背景 VMs 的 *Set Cursor* calls，以避免滑鼠突然出現在一個非焦點視窗上。至於具有 Windows 意識的驅動程式，知道它們何時擁有焦點，並能夠適時行動。

另一個函式（與 V86 中斷的處理有點離題）是 *Install_V86_Break_Point*。這個 service 將指令儲存於一個指定位址，並搭配一個 ARPL 指令，使控制權交到你提供的 callback 函式手上。這個 service 的目的是誘捕（trap）真實模式碼。例如，考慮一個在 Windows 風行之前就完成並使用 extended memory（譯註：1MB 以上）的真實模式驅動程式或 TSR。它應該使用 INT 2Fh, function 4310h 來獲得 XMS 管理程式的 API 進入點位址，然後將此位址儲存於某個位置上。由於應用程式會直接呼叫 XMS 管理程式，不發出任何中斷，所以 V86MMGR 安裝一個 V86 breakpoint 於 XMS 管理程式中以誘捕（trap）那些呼叫。不消說，這種手法需要許多事前的假設檢驗。

給傻司程式呼叫的 APIs

寫 VxD 的一個主要理由是爲了擴充 Windows 作業系統的能力。爲了避免產生出一個程式而卻從未被執行起來 -- 只因沒有人知道如何呼叫它，很顯然你會想要提供 API 給能夠觸及你的驅動程式的那些應用程式使用，讓它索求驅動程式提供的服務。爲此目的，你可以視你的呼叫者是 16 位元或 32 位元應用程式而定，使用兩種不同的技術。這一節中我將描述一個 Win32 程式如何使用 *CreateFile* 和 *DeviceIoControl* 來與 VxD 交談，然後我會談到 Windows 3.0 當初的 Win16 介面。

來自 32 位元傻司程式的呼叫

Win32 應用程式如果要呼叫你的 VxD，必須使用 *CreateFile*, *DeviceIoControl*, 以及 *CloseHandle* 函式。此外，你的 VxD 必須處理 *W32_DeviceIoControl* 這個系統控制訊息。*CreateFile* 是一個 Win32 API 函式，其功能正常而言是爲一個應用程式開啓一個檔案。但自從有了所謂的 Universal Naming Convention（萬能的命名習慣）之後，Microsoft 提供了一個方法，讓應用程式由此獲得一個特殊的 device handle：

```
HANDLE hDevice = CreateFile("\\\\.\\name", ...);
```

如果呼叫時檔名前面有奇怪的 "\\.\" 符號，那麼它就不是用來開啓一個檔案，而是找出指定的 device driver，並傳回其 handle，於是此一 driver 稍後便可被使用。

大部份時候，如果一個 Win32 應用程式需要用到一個 device driver，那麼由此應用程式動態載入該 driver 是合理的。下面就是 *CreateFile* 的完整呼叫動作，打開並載入一個 device driver：

```
HANDLE hDevice = CreateFile("\\\\.\\pathname", 0, 0,
    NULL, 0, FILE_FLAG_DELETE_ON_CLOSE, NULL);
```

在此語法中，*pathname* 指定 driver 檔名。*FILE_FLAG_DELETE_ON_CLOSE* flag 表示當你稍後呼叫 *CloseHandle* 關閉此 handle 時，得卸載（unload）驅動程式。

於是你可以在 *DeviceIoControl* 中使用這個 device handle。此一 API 規格如下：

```
BOOL result = DeviceIoControl(
    HANDLE hDevice,         // handle of device driver
    DWORD code,             // I/O control code
    LPVOID inbuffer,        // input buffer
    DWORD cbinbuffer,       // size of input buffer
    LPVOID outbuffer,       // output buffer
    DWORD cboutbuffer,      // size of output buffer
    LPDWORD pnumreturned, // where to put the number of output bytes
    LPOVERLAPPED povl);     // overlapped I/O structure
```

此函式的參數該如何使用，完全由 driver 的作者決定。一般而言 *code* 參數用來指定一個要求 driver 完成的動作，input buffer 內含要給驅動程式的參數，output buffer 內含驅動程式將傳回的數值。但即使是這些參數，其使用也都是約定成俗，沒有強制規定。如果你要，你也可以把功能代碼放在 *cboutbuffer* 之中，只不過這麼一來，你可能會使閱讀你的程式碼的朋友感到迷惑。唔，這還不是唯一缺點，上述安排表示你的 VxD 必須以傳回 0 的方式來回應 I/O control code 0，否則「動態載入介面」無法正確運作。

所以，讓我們假設你遵循傳統，以慣常的方式來使用 *DeviceIoControl* 參數。在系統內部，*DeviceIoControl* 被轉換爲對 VWIN32 驅動程式的一個呼叫（經由一個秘密的未公開介面）。你的 VxD 應該有一個針對 *W32_DeviceIoControl* 系統控制訊息的處理常式。此一

訊息頗爲離奇古怪，VWIN32 利用它來通知你：有一個 ring3 程式呼叫了 *DeviceIoControl*。你可以輕易以 C 語言處理這個訊息：

```
#include <vwin32.h>
DWORD __stdcall OnW32DeviceIoControl(PDIOCPARAMETERS p)
    {                               // OnW32DeviceIOControl
    switch (p->dwIoControlCode)
        {
    case DIOC_GETVERSION:       // i.e., control code 0
        return 0;
    ...
        }
    }                   // OnW32DeviceIOControl
```

此函式的傳回值是一個 Win32 錯誤代碼；0 表示沒有錯誤。此函式獲得一個 *DIOCPARAMETERS* 結構位址，基本上那是 *DeviceIoControl* 的參數複製品。此結構定義於**圖 10-3**。你的 driver 必須處理 *DIOC_GETVERSION* 子功能，如上所示，以便成就「可經由 *CreateFile* 被動態載入」這一事實。其他的碼一如我曾經說過的，隨你高興啦。假設你要實作一個 *GetVersion* service 給你的 ring3 clients 使用，當做是功能 1，你的 VxD service 函式可以內含這段碼：

```
...
switch (p->dwIoControlCode)
    {
    ...
case 1:
    if (!p->lpvOutBuffer || p->cbOutBuffer < 2)
        return ERROR_INVALID_PARAMETER;
    *(WORD*)(p->lpvOutBuffer) = 0x0100; // major/minor
    if (p->lpcbBytesReturned)
        *(PDWORD)(p->lpcbBytesReturned) = 2;
    return 0;
    ...
    }
```

此程式首先執行一個 switch，判斷 *dwIoControlCode* 值（*DIOCPARAMETERS* 結構的一個欄位）。這個值和 *DeviceIoControl* 函式的 *code* 參數是相同的，程式檢驗 *DeviceIoControl* 呼叫者所提供的輸出緩衝區（*lpvOutBuffer* 不應該是 NULL），並檢驗其大小至少應有 2 bytes（*cbOutBuffer* >= 2）。如果這些測試有一個沒有通過，意味呼叫端以錯誤方式呼叫 *DeviceIoControl*，於是便傳回 *ERROR_INVALID_PARAMETER*。這個

錯誤代碼在 WINERROR.H（Win32 SDK 表頭檔）中有定義。如果傳回的是此錯誤代碼或任何其他非零數值，會引起原來的 *DeviceIoControl* 函式傳回 FALSE，應用程式於是可以呼叫 *GetLastError* 並取回 *ERROR_INVALID_ PARAMETER*。

DeviceIoControl 的 *pnumreturned* 參數可有可無；如果呼叫端給予此參數一個 NULL 指標，*DIOCPARAMETERS* 結構中的 *lpcbBytesReturned* 欄位便也是 NULL。然而如果那個指標不是 NULL，你可以用它來儲存真正被拷貝到輸出緩衝區中的 bytes 個數。

```
typedef struct DIOCParams {
    DWORD Internal1;           // 00 pointer to cient registers
    DWORD VMHande;             // 04 VM handle
    DWORD Internal2;           // 08 pointer to DDB
    DWORD dwIoConrolCode;      // 0 control code fom DeviceIoContol call
    DWOD lpvInBuffer;          // 10 inpu buffer address from call
    DWORD cbInBuffer;          // 14 size of input buffer from call
    DWORD lpvOutBuffer;        // 18 output buffer address from call
    DWORD cbOutBuffer;         // 1C size of outpu buffer from cal
    DWORD lpcbBytesReturned;   // 20 where to put output count (can be NULL)
    DWORD lpOverlapped;        // 24 overlapped I/O block from call
    DWORD hDevice;             // 28 ring-three device handle
    DWORD tagProcess;          // 2C pocess tag
} DIOPARAMETERS;
```

圖 10-3　DIOCParams 結構，使用於 W32_DeviceIoControl 訊息過程之中

Win32 程式如果想知道 driver 版本號碼，可以這麼做：

```
WORD version;
HANDLE hDevice = CreateFile(...);
DeviceIoControl(hDevice, 1, NULL, 0, &version,
    sizeof(version), NULL, NULL);
```

在這裡，輸入緩衝區（input buffer）參數是 NULL，因爲此功能並不需要輸入資料。輸出緩衝區（output buffer）正是用來存放版本號碼的變數。設定「overlapped I/O 結構（譯註：最後一個參數）」的指標爲 NULL，表示這份程式碼提出要求，在控制動作完成之前不要重新獲得控制權。

Windows 95 有可能做到非同步（asynchronous）的 *DeviceIoControl* calls。爲了實現這一

點，你必須在打開這個 device handle 時指定 *FILE_FLAG_OVERLAPPED* flag，並提供一個 *OVERLAPPED* 結構的位址，做爲 *DeviceIoControl* 的最後一個參數。VxD 會立刻從其 *W32_DeviceIoControl* 處理常式中回返。稍後（推測大約是在一個 event callback 函式中），你的 VxD 呼叫 *VWIN32_DIOCCompletionRoutine* service 來激發與此 overlapped I/O operation 相關的 event。這個 VWIN32 call 允許你與 ring3 應用程式同步化。

來自 16 位元應用程式的呼叫

自 3.0 以降的每一個 Windows 版本，VxDs 都可以匯出（exports）APIs（譯註：此處 API 是指給應用程式直接呼叫的函式），只要簡單地在 device description block（DDB）中宣告它們即可。你可以提供一個 API 函式給 V86 程式使用，另一個 API 函式給保護模式程式使用。你也可以爲它們指定相同的函式。但是有兩個基本底線：

- 應用程式應該使用軟體中斷 2Fh, function 1684h 來獲得函式位址。當此函式被呼叫，你的 API 函式將獲得控制權。
- 你的 API 函式會檢驗並改變 client 暫存器（屬於呼叫端程式）。

在應用程式這一邊，程式員需要知道：出現在 DDB 之中的究竟是 VxD 獨一無二的 16 位元識別碼還是 VxD 名稱。如果程式員知道獨一無二的識別碼（*DriverId*），可以這麼做：

```
DWORD apiaddr;                  // address of VxD's API
_asm
    {                           // get API address
    xor di, di
    mov es, di
    mov ax, 1684h
    mov bx, DriverId
    int 2fh
    mov word ptr apiaddr, di
    mov word ptr apiaddr+2, es
    }                           // get API address
```

如果程式員不知道 *DriverId*，或如果此 ID 是 *Undefined_Device_ID*，應用程式會將 BX 暫存器設爲 0，並將 ES:DI 指向一個 8-byte、以空白字元補齊、區分大小寫的驅動程式

名稱。例如：

```
char drivername[9] = "MYVXD   ";      // three trailing blanks
DWORD apiaddr;                  // address of VxD's API
_asm
    {                                 // get API address
    xor bx, bx    ; or use mov bx, Undefined_Device_ID
    mov di, ss
    mov es, di
    lea di, drivername
    mov ax, 1684h
    int 2fh
    mov word ptr apiaddr, di
    mov word ptr apiaddr+2, es
    }                                 // get API address
```

如果中斷回返時 *apiaddr* 非零，意味著應用程式所指名的 VxD 已被載入並匯出一個 API。應用程式藉這個指標發出呼叫，並遵循雙方同意的暫存器使用習慣，與驅動程式交談。例如：

```
WORD version;
_asm
    {                       // call API
    mov ax, 0             ; function 0: get version
    call [apiaddr]
    mov version, ax
    }                       // call API
```

在 VxD 這一端，當你最初定義 device driver 時，首先指明 API 進入點：

```
Declare_Virtual_Device MYVXD, ..., v86api, pmapi
```

此巨集的最後兩個參數用來指明保護模式 API 和 V86 模式 API 的進入點。只要應用程式透過 INT 2Fh, function 1684h, 以 *apiaddr* 指標呼叫你的 VxD，VMM 就會呼叫某些 VxD 函式。

接下來就是定義 API 函式本身。VMM 呼叫你的函式時，以 EBX 放置目前的 VM handle，以 EDI 放置 current thread handle，以 EBP 指向 current thread 的 client register structure（CRS）。下面是一個以 assembly 語言完成的 *GetVersion* API 函式：

```
BeginProc pmapi, pageable
          movzx  eax, [ebp+Client_AX]
          test   eax, eax
          jnz    error
          mov    ax, word ptr MYVXD_DDB.DDB_Dev_Major_Version
          xchg   ah, al
          mov    [ebp+Client_AX], ax
          and    [ebp+Client_EFlags], not CF_Mask
          ret
error:    or     [ebp+Client_EFlags], CF_MASK
          ret
EndProc   pmapi
```

這個例子從 client AX 中取出一個功能代碼。如果代碼爲 0，就把主版本和次版本號碼組合起來放進 client AX，並清除 client 的 carry flag。如果功能代碼不是 0，它就設立 client 的 carry flag，表示有錯誤發生。在此 API 函式回返至 VMM 之後，VMM 會 "redispatch" 原 VM，並夾帶被更改過的 flags 和暫存器。

這些都是應用程式 APIs 的基礎。下面是兩項整理。第一，我顯示給你看的應用程式可以是一個 V86 模式中的 DOS 程式或是一個 16 位元 Windows 程式（當然是在保護模式中執行），或是一個 extended DOS 程式（在 DPMI 或其他 DOS extender 中執行）。INT 2Fh, function 1684h 介面對上述任何一種程式都能夠有效運作。在 V86 client 和保護模式 client 之間唯一的不同就是，你可以（但是非必要）使用不同的 API 函式來爲它們服務。使用我在第 9 章說過的 *Map_Flat* service，便可以定址出由 client segment 暫存器所指的資料，而不需要知道 client 處於哪一種執行模式。

API 機制的第二個重點是，它對 32 位元 extended DOS 程式有效，對 32 位元 Windows 程式卻無效。USE32 client 會取回一個 16:32 遠程指標（而不是一個 16:16 指標），指向應用程式 API 進入點；它應該使用 32 位元暫存器來進行參數傳遞，因爲那正是 *Map_Flat* 所期望的。除了這些，USE32 client 對 VxD 的介面，與其 16 位元版完全一致。32 位元 Windows 程式不能夠使用此介面的理由是，VMM 認爲 System VM（其中執行的是 32 位元程式）正在執行一個 16 位元保護模式的 DPMI client 程式。VMM 陳腐地認爲，所有應用程式都像某些真實模式程式（亦即 Windows session 啓動之初作用起來的）一樣，都是呼叫 DPMI 的「模式切換函式」來進入保護模式。Windows 95 KERNEL

是一個 16 位元程式，所以 VMM 也就假設所有的 Windows 95 應用程式都是 16 位元應用程式。

在 32 位元應用程式中呼叫 VxDs

你可能會懷疑，系統初始化過程中所做的 bitness 決定，就是由 application API 完成。如果軟體中斷來自 Win32 程式，VMM 無法總是正確地處理它們。是的，某些軟體中斷在 System VM 的 IDT 中有 16 位元 gates；而 16 位元 gate 這成 CPU 只儲存 FLAGS（而非 EFLAGS）、CS:IP（而非 CS:EIP）以及 SS:SP（而非 SS:ESP）暫存器。如果 32 位元程式發出一個那樣的中斷，VMM 只會在堆疊中儲存一部份 program context。於是 IRET 指令執行之後，會回返到一個隨機地點，並夾帶一個隨機的堆疊指標，這可不妙。INT 2Fh 是 API 公共建設的一部份，卻沒有這個特殊問題，因為它使用 32 位元 gate，會保存完整的 context。不幸的是從第一章 INT 2Fh 處理常式順流而下的碼毫無招架能力地假設它正面對一個 16 位元呼叫者，並因而不正確地回返到 VM。為此，Win32 應用程式甚至不能夠獲得一個 API 進入點的位址。如果你想要自己攔截（hook）INT 2Fh 並檢查呼叫端是 16 位元或 32 位元程式，倒是可以繞過這個問題。然而最終你將面對一個無法解決的問題：應用程式需有一個 flat 位址準備被呼叫，但是 INT 30h 處理常式不會允許任何一個函式呼叫進入 3Bh 以外的任何 segment 之中。此外，處理常式的回返也不正確。

這個故事的教訓是，當你手上是整潔漂亮的 32 位元應用程式，就不該嘗試使用那些給怪異老舊的 16 位元應用程式使用的方法。Microsoft 意欲 Win32 跨機器平台和作業平台，而「以軟體中斷做為通訊工具」會妨礙可移植性。此外，Microsoft 從不要求你面對 VxD 碼像面對一般應用程式那樣。這就是「回呼（call back）進入 VMs」為什麼如此困難的原因。如果你絕對需要在 32 位元應用程式中做這些事情，你應該使用 *DeviceIoControl* 和 *_VWIN32_QueueUserApc*，它們至少允許應用程式像過去一樣地具移植性。

呼叫 Windows 應用程式

由於 VxDs 是作業系統的一部份，而作業系統是 Windows 圖形作業環境的底層，所以你可以假設，VxD 可以輕易完成應用程式的行爲。奇怪的是，這樣的假設並不正確。事實上直到 Windows 95 問世之前，VxD 非常難以讓它自己被 ring3 碼看見，更別提與使用者有所互動了。前一節我展示了 VxD 如何匯出（export）API，讓應用程式能夠申請作業系統的服務。這一節，我要展示另一半面貌，解釋 VxD 如何能夠回頭和相關的應用程式交談。

要瞭解爲什麼系統程式設計是如此困難，請記住，VMM 的原始任務是在 VMs 之間傳佈（distributed）中斷。由於係在底層運作，並且未與執行於 VMs 之中的應用程式碼有任何關聯，所以 VMM 沒有必要太過於靠近那些碼。因此 VMM 對 VM 內部生命的唯一貢獻就是中斷有紀律的 ring3 執行流程，並且暫時將執行點導引到其他地方。情況之所以如此複雜，是因爲 Windows 本身並非「可重入（reentrant）」，中斷了 System VM 之後，你實在無法做太多事情。在前一版 Windows 中，你所能夠做的就是呼叫 *PostMessage*，這是極少數可以在中斷時期內呼叫的 Windows API 函式。

Windows 3.1 的 PostMessage

你可能需要寫一個相容於 Windows 3.1 的 VxD，你也可能需要瞭解或維護爲 Windows 3.1 所寫的碼，所以你應該瞭解那些中世紀風格的 VxD 撰寫者是如何地被桎梏於 *PostMessage*。這個難題可分爲兩部份，Windows 應用程式的主要工作是告訴 VxD 說 *PostMessage* 至何處，VxD 則在必要時候將 "interrupt-like calls" 排班（schedules）至 *PostMessage* 身上。VxD 開放出一個保護模式 API，給輔助的 Windows 程式用以傳輸關鍵性的 *PostMessage* 位址。

PostMessage 的更多細節

如果在 VxD 中你只能夠呼叫一個 Windows API 函式，*PostMessage* 將是你唯一的選擇。其宣告如下：

```
BOOL WINAPI PostMessage(HWND hwnd, UINT msg,
                        WPARAM wParam, LPARAM lParam);
```

hwnd 參數表示一個視窗 handle，*msg* 參數表示某個 Windows 訊息。*wParam* 和 *lParam* 參數內含一些資料，其意義視不同的訊息而不相同。做為一個 Windows 函式設計者，你可以定義自己的訊息，也可以決定是否要讓 *lParam* 在某種情況下扮演指標的角色。有了指標在手，你就可以指定任何複雜的資料做為函式參數了。

PostMessage 嘗試將一個 Windows 訊息放到擁有此視窗之 task 的訊息佇列（msg queue）中。如果成功就傳回 TRUE。傳回 FALSE 表示視窗不存在，或者 task queue 已滿（應用程式可以使用 *SetMessageQueue* API 函式增加 queue 的大小）。應用程式碼中有一個訊息迴圈，使用 *GetMessage* 或 *PeekMessage* 從 queue 中取出訊息，並有一個 *DispatchMessage* API 函式，將訊息送往適當的 Windows 函式。當訊息迴圈下一次獲得 CPU 時間，它將再次抓取訊息。最終它總會抓到你所 "post" 的那一個，於是 ring3 Windows 函式就可以獲得控制權並執行你的命令了。

一個輔助的 Windows 程式，會定位出 VxD 的保護模式 API 函式，並將 Windows *PostMessage* 函式的位址傳給它。一般而言，輔助程式也需要供應一個視窗 handle，有了此 handle，VxD 就可以對它 "post" 訊息。例如（以下程式碼可在書附光碟的 \CHAP10\POSTMESSAGE-VTOOLSD 目錄中找到）：

The Windows Application

```
FARPROC aPostMessage = (FARPROC) PostMessage;
void (*apiaddr)();

_asm
```

```
   {                // get API entry
   xor di, di
   mov es, di
   mov ax, 1684h
   mov bx, 4242h      ; use your own VxD ID here
   int 2fh
   mov word ptr apiaddr, di
   mov word ptr apiaddr+2, es
   }                // get API entry

if (apiaddr)
   _asm
   {                // register this application with the VxD
   mov ax, 1
   mov bx, hwnd
   mov cx, WM_USER+256
   mov di, word ptr aPostMessage
   mov si, word ptr aPostMessage+2
   call [apiaddr]
   }                // register this application with the VxD
```

這段程式碼的第一部份使用中斷 2Fh, function 1684h 獲得某個 VxD （識別碼爲 4242h） API 位址。第二部份則是將參數置入數個 general 暫存器之後，呼叫 API 功能 01h。

VxD 對於功能 01h (registration) 實作如下：

The VxD (for Windows 3.1)

```
DWORD hwnd;
DWORD msg;
DWORD aPostMessage;

VOID PM_Api_Handler(VMHANDLE hVM, PCLIENT_STRUCT pRegs)
   {                        // PM_Api_Handler
   _clientEFlags &= ~CF_MASK;
   switch (_clientAX)
      {                     // select API function
      ...

   case 1:
      hwnd = _clientBX;
```

```
        msg = _clientCX;
        aPostMessage = ((DWORD) _clientSI << 16) |
                       ((DWORD) _clientDI & 0xFFFF);
        break;
        ...
        }                       // select API function
    }                           // PM_Api_Handler
```

根本上，API 功能 01h 只是簡單地將參數從 client 的 general 暫存器拷貝到 VxD 的全域變數中。所以應用程式所做的這個 registration call，其實就是給予 VxD 一個視窗 handle、一個訊息代碼、以及 Windows *PostMessage* 函式的 ring3 位址。VxD 沒有其他方法可以在 Windows 3.1 中獲得這些東西，這雖然令人驚訝，卻是真的。

爲了說明你面對這些資訊應該做些什麼，讓我假設，你打算寫一個 VxD，在你產生和摧毀一個 VM 時記錄（logging）其間發生的系統控制訊息。你需得爲每個系統控制訊息處理常式加上一個呼叫，呼叫共同的區域函式（本例稱之爲 *DoPostMessage*），將一個訊息 "post" 至輔助的 Windows 應用程式中。我使用 VToolsD 來撰寫這個例子，以免陷入 assembly 語言的沼澤。

The VxD（續）

```
BOOL OnCreateVm(VMHANDLE hVM)
    {
    DoPostMessage(CREATE_VM, hVM);
    return TRUE;
    }

...

typedef struct tagMSGSTUFF {
    DWORD hwnd;
    DWORD msg;
    DWORD wParam;
    DWORD lParam;
} MSGSTUFF, *PMSGSTUFF;

PriorityVMEvent_THUNK PostMessageThunk;
TIMEOUT_THUNK TimeoutThunk;
VOID DoPostMessage(DWORD event, VMHANDLE hVM)
```

```
    {
    PMSGSTUFF p;

    if (!aPostMessage) // a global variable
        return;

    p = (PMSGSTUFF) _HeapAllocate(sizeof(MSGSTUFF),
                                  HEAPZEROINIT);
    Assert(p);
    p->hwnd = hwnd;    // hwnd is a global variable
    p->msg = msg;      // msg is a global variable
    p->wParam = event;
    p->lParam = (DWORD) hVM;

    Call_Priority_VM_Event(0, Get_Sys_VM_Handle(),
        PEF_WAIT_FOR_STI | PEF_WAIT_NOT_CRIT,
        p, PostMessageCallback, 0, &PostMessageThunk);
    }

VOID __stdcall PostMessageCallback(VMHANDLE hVM, PVOID refdata,
    PCLIENT_STRUCT pRegs, DWORD flags)
    {
    DoCallback((PMSGSTUFF) refdata, pRegs);
    }

VOID __stdcall TimeoutCallback(VMHANDLE hVM, PCLIENT_STRUCT
    pRegs, PVOID refdata, DWORD extra)
    {
    DoCallback((PMSGSTUFF) refdata, pRegs);
    }

BOOL DoCallback(PMSGSTUFF p, PCLIENT_STRUCT pRegs)
    {
    CLIENT_STRUCT saveregs;
    BOOL okay;

    Assert(aPostMessage); // aPostMessage is a global variable
    Save_Client_State(&saveregs);
    Begin_Nest_Exec();

    Simulate_Push(p->hwnd);
    Simulate_Push(p->msg);
    Simulate_Push(p->wParam);
    Simulate_Push(HIWORD(p->lParam));
    Simulate_Push(LOWORD(p->lParam));
    Simulate_Far_Call(HIWORD(aPostMessage),
```

```
    LOWORD(aPostMessage));
Resume_Exec();

okay = _clientAX;

End_Nest_Exec();
Restore_Client_State(&saveregs);

if (okay)
    _HeapFree(p, 0);
else
    Set_VM_Time_Out(100, Get_Sys_VM_Handle(), p,
                TimeoutCallback, &TimeoutThunk);

return okay;
}
```

DoPostMessage 內部對於 System VM 中的一個 priority VM event callback 做了排班（schedule）動作，以便真正執行這個 call。它使用一小塊記憶體（從 system heap 中配置而得）來放置 Windows 訊息的重要參數。

DoCallback 是真正呼叫 *PostMessage* 的函式。由於呼叫 *PostMessage* 本質上會中斷 System VM 目前正在執行的任何動作，所以第一個步驟就是呼叫 *Save_Client_State*，將目前的 client register image 保存起來。*Begin_Nest_Exec* 開始一個巢狀執行區塊（nested execution block），並強迫 VM 進入保護模式（如果它不是已經處於保護模式的話）。*Simulate_Push* 把要給 *PostMessage* 的參數放進虛擬堆疊中，次序符合 *__pascal* 呼叫習慣。*Simulate_Far_Call* 把一個回返位址推入（pushes）虛擬堆疊中並將虛擬的 CS:IP 暫存器設爲 *PostMessage* 位址（那是稍早由測試程式提供的）。*Resume_Exec* 允許 System VM 繼續執行，於是 *PostMessage* 發揮作用並將一個 Windows 訊息放進測試程式的 queue 之中。*End_Nest_Exec* 和 *Restore_Client_State* 將 System VM 恢復爲原先狀態。測試程式（書附光碟的 \CHAP10\POSTMESSAGE 目錄）執行結果如圖 **10-4**。

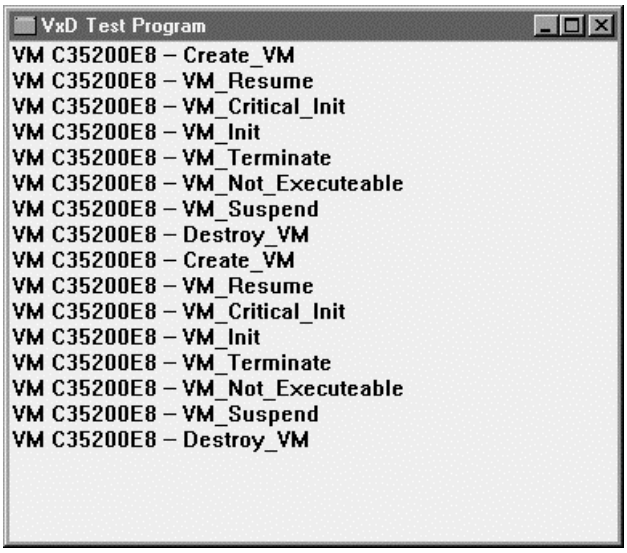


圖 10-4　PostMessage 測試程式的執行結果

注意，*DoPostMessage* 函式是將一個 event callback 函式排班（schedule）至 *PostMessageCallback* 身上，再由其內部呼叫 *DoCallback*。安排這額外一層子函式的理由是，Windows *PostMessage* 函式可能會失敗（例如在應用程式的 msg queue 已滿的情況下）。爲了處理這種情況，*DoCallback* 在發出 "cleanup calls"（那會覆蓋掉 *PostMessage* 的傳回值，也就是 client AX 的內容）之前，先捕捉其值。如果這個值是 TRUE，*DoCallback* 就釋放用以儲存參數的那塊記憶體。然而如果其值是 FALSE，一旦 System VM 已經執行了 100 毫秒（ms），*DoCallback* 便對另一個 event callback 函式進行排班。VtoolsD 要求這個 event 必須使用不同的 callback thunk 和一個不同的 callback 函式，這就是爲什麼我寫下 *TimeoutCallback* 的原因，它的內容事實上和 *PostMessageCallback* 完全相同。

這段碼有兩個潛在問題。第一，除了 msg queue 滿了之外，*PostMessage* 還可能因其他理由而失敗。例如應用程式可能結束並摧毀其視窗。如果交給 *PostMssage* 的視窗 handle 不再合法，*PostMessage* 就會失敗而這段碼也就每當時間終了重試一次，永遠在迴圈中出不來了。一個 VxD 不可能判斷視窗 handle 是否依然合法，也不可能判斷 *PostMessage* 爲什麼失敗。因此，你可能得在傳達訊息的嘗試次數上做點限制。依賴「應用程式在結

束之前將其視窗 handle 解除註冊」也不可行，因爲如果是不正常當機，應用程式就沒有正常的結束管道(除非應用程式開發者使用 ToolHelp 發展出一個所謂的 task termination callback，解脫這個問題）。

第二個潛在問題是，如果你把這段碼放進一個 dynamic VxD 中，那麼萬一有人卸載（unload）了這個 VxD，而此時尚有一個或多個被 *DoPostMessage* 產生的 events 尚未解決，怎麼辦？除非你把 event 的處理碼放進 static code segment 中，否則你將會置身於「當 event 發生，找不到 callback」的風險之中。這種情況的處理有點複雜，因爲任一時刻可能有一個以上的 event 尚未獲得解決。我不想讓這個例子的重心因爲加上所有的複雜性而模糊掉了，但是當你寫一個真正的 VxD，你的確需要考慮這些主題。

Windows 95 的 PostMessage

我希望能夠很高興地說，在一個 Windows 95 VxD 中呼叫 *PostMessage* 變得簡單多了。但是我不能夠，因爲只有簡單一些些而已。Windows 95 有一個新的 service 名爲 *_SHELL_PostMessage*，用來負責找出並呼叫 *PostMessage*（夾帶正常的四個參數）。然而這個 service 並不保證絕對能夠遞送訊息，你還是必須應付 *PostMessage* 失敗所造成的影響。使用這個 service，你可以將稍早出現的 VxD *DoPostMessage* 函式重寫如下（這段碼可以從書附光碟的 \CHAP10\SHELL_POSTMESSAGE-VTOOLSD 目錄中獲得）：

The VxD (for Windows 95)

```
...
VOID DoPostMessage(DWORD event, VMHANDLE hVM)
   {
   PMSGSTUFF p;

   if (!hwnd)      // hwnd is a global variable
      return;

   p = (PMSGSTUFF) _HeapAllocate(sizeof(MSGSTUFF),
                                 HEAPZEROINIT);
   Assert(p);
   p->hwnd = hwnd; // hwnd is a global variable
```

```
    p->msg = msg;   // msg is a global variable
    p->wParam = event;
    p->lParam = (DWORD) hVM;

    _SHELL_PostMessage((HANDLE) hwnd, msg, (WORD) event, (DWORD) hVM,
                        (PPostMessage_HANDLER) CheckPostMessage, p);
    }

VOID __cdecl CheckPostMessage(DWORD rc, PMSGSTUFF p)
    {
    if (rc)
        _HeapFree(p, 0);
    else
        Set_VM_Time_Out(100, Get_Sys_VM_Handle(), p,
                        TimeoutCallback, &TimeoutThunk);
    }
VOID __stdcall TimeoutCallback(VMHANDLE hVM,
                        PCLIENT_STRUCT pRegs, PVOID refdata, DWORD extra)
    {
    PMSGSTUFF p = (PMSGSTUFF) refdata;
    _SHELL_PostMessage((HANDLE) p->hwnd, p->msg, (WORD) p->wParam,
    p->lParam, (PPostMessage_HANDLER)
    CheckPostMessage, p);
    }
```

_SHELL_PostMessage 會將一個 event 納入排程，並執行呼叫 *PostMessage* 的必要動作。然後它呼叫 *CheckPostMessage* callback 函式以記錄 *PostMessage* 的傳回值。如果 *PostMessage* 傳回 FALSE，*CheckPostMssage* 就將一個 timeout 納入排程，在那之後，應用程式重複對 *_SHELL_PostMessage* 的呼叫。

這個例子比我先前所展示的例子，在兩方面稍微簡單些：第一，VxD 不需要記住 *PostMessage* 的位址。第二，它不需要讓所有的 service calls 都設定巢狀執行區塊（nested execution block）、push 參數、並實際產生呼叫。然而它還是需要先從應用程式手中獲得一個視窗 handle 以及訊息代碼，然後將 timeout 納入排程以便再試一次（如果 *PostMessage* 失敗的話）。

AppyTime Events

與先前版本不同的是，Windows 95 提供了一個方法，讓 VxD 可以呼叫除了 *PostMessage* 之外的其他正規 16 位元 Windows 函式。這個方法圍繞著 AppyTime（application time）event 打轉。首先，你得將一個 event callback 納入排程：

```
_SHELL_CallAtAppyTime(OnAppyTime, NULL, 0, 0);
```

過一段時間，當 Windows 95 處於穩固狀態下（此時呼叫 API 是被允許的），SHELL device 呼叫你的 callback 函式。然後你可以使用一些特殊的 SHELL application time services（如表 **10-6** 所列）來發出 Windows API calls。舉個例子，下面的 callback 函式會啓動 *WinHelp*，顯示 *_SHELL_CallDll* service 的線上說明（這段碼可在書附光碟的 \CHAP10\APPYTIME-DDK 目錄中找到）：

```
VOID __cdecl OnAppyTime(DWORD refdata)
    {                       // OnAppyTime
    #pragma pack(1)
    struct {
        DWORD dwData;
        WORD  fuCommand;
        DWORD lpszHelpFile;
        WORD hwnd;
    } WinHelpArgs;
    #pragma pack()

    WinHelpArgs.hwnd = 0;
    WinHelpArgs.lpszHelpFile = _SHELL_LocalAllocEx(LPTR +
        LMEM_STRING + LMEM_OEM2ANSI, 0,
        "D:\\DDK\\DOCS\\VMM.HLP");
    WinHelpArgs.fuCommand = 0x101;      // HELP_KEY
    WinHelpArgs.dwData = _SHELL_LocalAllocEx(LPTR +
        LMEM_STRING + LMEM_OEM2ANSI, 0,
        "_SHELL_CallDll");

    _SHELL_CallDll("USER", "WinHelp", sizeof(WinHelpArgs),
        &WinHelpArgs);

    _SHELL_LocalFree(WinHelpArgs.lpszHelpFile);
    _SHELL_LocalFree(WinHelpArgs.dwData);
    }                       // OnAppyTime
```

Service	說明
_SHELL_CallDll	呼叫 16 位元 DLL 中的一個函式
_SHELL_FreeLibrary	卸載（unload）一個 DLL
_SHELL_GetProcAddress	定位出 DLL 中的一個進入點（entry point）
_SHELL_LoadLibrary	載入一個 DLL
_SHELL_LocalAllocEx	配置 ring3 可定址記憶體
_SHELL_LocalFree	釋放 ring3 可定址記憶體

表 10-6　在 application time（AppyTime）中可運用的 services

_SHELL_CallDll 是你用來發出 Windows API calls 的基本 service。你可以像上例那樣使用它：指定一個 DLL 名稱以及 DLL 中的一個進入點名稱，或是以一個匯出序號（export ordinal）來取代名稱。要不然你也可以利用其他方法來決定一個函式的 16:16 位址（例如使用 *_SHELL_GetProcAddress*，或是令一個應用程式傳給你）並直接指定該位址。此外，你還需對此 service 提供一塊記憶體位址，你必須先將函式參數依照「被呼叫之 16 位元函式期望於其堆疊中找到」的次序，排列於該塊記憶體中。

如果要把字串參數交給 16 位元函式，你需要一個 16:16 遠程指標，指向該字串。*_SHELL_LocalAllocEx* 函式爲你產生必要的指標，它先在 local heap 中配置記憶體。當你不再需要這塊記憶體，必須呼叫 *_SHELL_LocalFree* 明白地釋放它。如果你嘗試傳遞一個以 null 爲結束符號的字串，可以使用 *LMEM_STRING* flag，令 SHELL 爲你計算字串。請記住，VxDs 使用 OEM 字元集，如果你所呼叫的 API 函式期望獲得的是 ANSI 字元集（本例就是如此），請加上 *LMEM_OEM2ANSI* flag。

Application time events 並不特別方便使用，因爲呼叫 ring3 函式的機制是如此地累贅又麻煩。在可能於 application time callback 中使用的各個 VMM services 之中，有一個重要的限制是：不要呼叫 *Begn_Nest_Exec*，因爲如果它被使用於 application time，可能會使 Windows 95 當掉。此外，此方法中你只能夠呼叫 16 位元程式。你必須使用一個非同步函式呼叫（asynchronous procedure call）才能呼叫 32 位元程式（而且你不需一直等

到 application time 才能對如此一個呼叫動作進行排程）。

非同步函式呼叫（Asynchronous Procedure Calls）

到目前爲止，我告訴你的每一件事都解釋了 Win32 碼的呼叫問題。一如稍早所見，你可以使用 *DeviceIoControl* 和 *W32_DeviceIoControl* 訊息，在 Win32 程式中呼叫 VxD。這一小節中我要告訴你 VxD 如何使用一個名爲「非同步函式呼叫（asynchronous procedure call 或 APC）」的新機制來呼叫 Win32 碼。簡單地說，Win32 應用程式利用 *DeviceIoControl* 送出一個 callback 函式位址給 VxD，然後進入所謂的 **alertable wait state**。VxD 使用 *_VWIN32_QueueUserApc* 來對一個「非同步 callback 函式」進行排程。然後 callback 發生，Win32 應用程式在等待中甦醒過來。

這一節用以說明 APC 機制的例子，示範的是個簡單的監視器，觀察 V86 應用程式的啓動與結束。你可以在書附光碟的 \CHAP09\SERVICEHOOK-ASM 目錄中找到程式碼。Win32 應用程式負責列印出啓動與結束時的 event。至於 VxD，一如你所想像，藉由攔截（hooking）DOSMGR APIs，誘補（traps）啓動和結束時的 events，並將 APCs 放進佇列（queue）之中，以求改變應用程式。

在 Win32 這一端，應用程式動態載入這個 VxD 並使用 *DeviceIoControl* 來供應兩個 callback 函式的位址：

```
HANDLE hDevice;
void (WINAPI *abegin)(DWORD) = beginapp;
void (WINAPI *aend)(DWORD) = endapp;

hDevice = CreateFile("\\\\.\\myvxd.vxd", 0, 0, NULL, 0,
   FILE_FLAG_DELETE_ON_CLOSE, NULL);
DeviceIoControl(hDevice, 1, &abegin, sizeof(abegin),
   NULL, 0, NULL, NULL);
DeviceIoControl(hDevice, 2, &aend, sizeof(aend),
   NULL, 0, NULL, NULL);
```

這個 VxD 使用前一章所描述的 *Hook_Device_Service* service 來監視

DOSMGR_Begin_V86_App call 和 *DOSMGR_End_V86_App* call。當其中之一發生，VxD 就把一個 APC "queues to" 某個被 *DevceIoControl* calls 註冊過的函式：

```
BeginProc BeginHook, service, hook_proc, next_begin, locked
        pushad
        mov   eax, beginfunc
        call  DoApc
        popad
        jmp   [next_begin]
EndProc BeginHook

BeginProc EndHook, service, hook_proc, next_end, locked
        pushad
        mov   eax, endfunc
        call  DoApc
        popad
        jmp   [next_end]
EndProc EndHook

BeginProc DoApc, locked
        test  eax, eax
        jz    doapc_done
        VxDCall _VWIN32_QueueUserApc, <eax, esi, appthread>

doapc_done:
        ret
EndProc  DoApc
```

在此片段之中，*beginfunc* 和 *endfunc* 都是 DWORD 變數，VxD 把應用程式的 *beginapp* 和 *endapp* 函式位址儲存於其中；*appthread* 持有應用程式的 thread control block 位址。這個例子中最有效果的部份是對 *_VWIN32_QueueUserApc* 的呼叫，第一個參數是即將被呼叫的 Win32 函式位址，第二個參數是個 DWORD，可被傳遞給該函式（成爲該函式的參數），第三個參數是個 thread，此一呼叫將在其中發生。術語 "asynchronous procedure call" 中的 "asynchronous" 表現於一個事實：此 API 會立刻回返，不會等到這個 call 真正發生於 VM 才回返。

此其間應用程式經由呼叫 *SleepEx*, *WaitForSingleObjectEx* 或 *WaitForMultipleObjectsEx*，已經把自己置於一個 **alertable wait state** 中。爲求簡單，我選擇使用 *SleepEx*：

```
while (SleepEx(INFINITE, TRUE) == WAIT_IO_COMPLETION)
   ;           // i.e., wait forever or until error
```

SleepEx 的第二參數 TRUE 表示我們要讓這個等待成爲 "alertable"。一個 alertable wait 的意思是，VxD 的 APC 會引起 thread 醒來並執行指定的 callback 函式，於是 *SleepEx* 會傳回 *WAIT_IO_COMPLETION* 代碼。一如程式碼所示，應用程式於是會一再重複等待。按下 Ctrl-Break 可以結束這個應用程式，並關閉其所有 handles。關閉 device driver handle 會動態卸載 VxD，因爲我們使用 *FILE_FLAG_DELETE_ON_CLOSE*。

應用程式的 callback 函式被宣告爲 void __stdcall，它會收到一個 DWORD 參數，其值與 *_VWIN32_QueueUserApc* 的第二個參數相同。例如：

```
void __stdcall beginapp(DWORD psp)
   {                            // beginapp
   char name[9];
   printf("Starting V86 App %s\n", getname(psp, name));
   }                            // beginapp

void __stdcall endapp(DWORD psp)
   {                            // endapp
   char name[9];
   printf("Ending  V86 App %s\n", getname(psp, name));
   }                            // endapp

char *getname(DWORD psp, char *name)
   {                            // getname
   char *p  = (char *) (psp - 8);
   memcpy(name, p, 8);
   name[8] = 0;
   return name;
   }                            // getname
```

在這裡，我們正攔截的兩個 DOSMGR services 以 ESI 指向 V86 程式的 Program Segment Prefix（PSP）的 high-linear 位址。這個指標成爲 *beginapp* 和 *endapp* 函式的參數。然後我們依賴 DOS 4.0（及更新版本）的性質：PSP 之前的 arena header 的偏移位置 8 處含有程式名稱。因此，這個極小的程式可以列印出作用期間的每一個啓動或結束的 V86 程式名稱。

這個例子在實用方面或許太過簡單，主要是因爲搭配 APC 所使用的唯一一個 DWORD 參數，沒辦法傳達太多資訊 。除非那是一個指標。我們不妨假設讓 VxD 配置一些記憶體（或許是使用 *_HeapAllocate*）並將指標傳給 Win32 應用程式。但是應用程式沒辦法釋放這塊記憶體，它必須 call back 回到 VxD 來（使用另一個 *DeviceIoControl* call）以執行必要的 *_HeapFree*。這個邏輯，不僅其簿記工作傷腦筋，其程式碼也很難產生以及被大眾理解。第 15 章的 MONITOR 實例有比較週延的作法。

第三篇

I/O 程式設計

Input / Output Programming

第 11 章

Plug and Play
（即插即用）

譯註：本章多次出現 configuration（名詞）或 configure（動詞）字眼，有時候我會將它們譯爲「組態」或「設置」（視上下文而定）。

你知道更換一個電燈泡需要多少位系統程式員嗎？我也不知道，聽起來這應該是個硬體問題。

這個笑話告訴我們，個人電腦的硬體問題十分詭異而令人困惑。擁有軟體專門知識的人，大部份對於設置網路卡或音效卡的工作仍然望而生畏。如果沒有 Plug and Play，那麼就算只是將一台 PC 升級以便玩多媒體遊戲，這麼簡單的事情，你也必須精通 IRQ 衝突、DMA 通道配置、I/O port 定址、以及記憶體分配等課題。

除了對可攜式電腦以及可動態更換之 PCMCIA 卡的組態設定傷腦筋之外，Windows 面對所有連接在 PC 上的週邊裝置（device），也都需要一個裝置驅動程式（driver）來運作。這就是爲什麼 Microsoft 亟需發明一個好方法來解決 device 的設置問題。答案就是

Windows 95 的 Plug and Play 架構。

Plug and Play 包括一組硬體標準及一個軟體標準。硬體標準決定「devices 如何確認本身」，以及「它們在作業系統裡的資源需求」；軟體標準決定「系統程式員如何撰寫驅動程式」，以便一旦 Windows 95 必須在電腦上確認特定硬體時，可以找到並載入驅動程式。我將在這一章和下一章，以兩種不同的角度來說明 Plug and Play。由於這是一本系統程式設計方面的書，所以我專注於軟體方面。如你所見，規劃一些與 Plug and Play 軟體架構緊密相連的元件，會引起程式員不少的困擾，但是保證會讓使用者更覺方便。

硬體如何運作

我即將說明的各種硬體標準，有兩個目的。第一個目的是允許作業系統處理「電腦硬體的自動辨識」，第二個目的是允許作業系統「自動重設硬體以協調資源的爭奪」。由於電腦硬體演進的緣故，這些硬體標準伴隨著 hardware bus 的標準而演變，包括 clock rates、pin assignments、以及其餘的電氣特性。我將以 bus 種類來分類討論，並以古老的 ISA bus 開始。

ISA Bus

ISA bus 是使用最久、最普遍、最便宜的 bus 標準，當然也是最笨拙的。早在明瞭「設置一個現代化系統是多麼困難」以前，Industry Standard Architecture （ISA）bus 就已經存在了。因此若想要讓 ISA bus 支援 Plug and Play，作業系統必須使用一種新方法來自動辨認硬體介面卡。老舊的介面卡並不認識我即將描述的 Plug and Play 協定，它們總是以既有的方式運作，使用者想必得在上面一邊咒罵，一邊嘗試設定 DIP switches 或 jumpers 以避免安裝時發生的衝突。

簡單地說，系統軟體首先輸出一筆特定的 32-byte 資料到保留的 I/O port 上，以此啓動硬體偵測程序。所有接在 ISA bus 上的 Plug and Play 裝置，會同時並不斷地尋找這個所謂的 initiation key，然後以進入 configuration mode 做爲回應。系統軟體接著會依次區隔

每塊 Plug and Play 介面卡，並分別爲它們指定獨一無二的 handle（一個 Card Select Number，簡稱爲 CSN）。接下來系統軟體可以從介面卡上讀取資源需求及其他資料，決定如何以最佳方式來配置電腦資源，然後將每一塊介面卡設置（configure）妥當。最後，系統軟體使所有的 Plug and Play 介面卡發生作用。

我不準備討論「Plug and Play 介面卡如何確認資源需求並接受設置指令」的所有細節，你可以從 *Plug and Play ISA Specification 1.0A*（Microsoft MSDN 內含）得到完整的資料。以下我將詳細描述初始化（initiation）及區隔（isolation）的方法。

爲了使所有 Plug and Play 介面卡進入 configuration mode，系統軟體送出兩個 0 bytes 加上下列 32-byte 資料流到 I/O port 279h 去：

```
6A B5 DA ED F6 FB 7D BE DF 6F 37 1B 0D 86 C3 61
B0 58 2C 16 8B 45 A2 D1 E8 74 3A 9D CE E7 73 39
```

和「PC 軟體的偉大傳統 ☺」不同的是，這個數列並不是任何人的名字，或是開發小組常去的餐廳名稱。它們是一個二元多項式（binary polynomial）的連續數值，其隨機發生的可能性微乎其微。Port 279h 通常是 LPT1 的狀態埠，通常處於唯讀狀態。因此，Plug and Play 協定可避免因爲新功能的加入而引起的疏忽。

當所有的 Plug and Play 介面卡處於 configuration state，系統軟體開始一次一個地區隔它們，以便指定 handles。其演算法取決於每塊介面卡上擁有的唯一一個 64 位元識別碼，其中包括 32 位元的廠商編號，以及 32 位元的介面卡編號。64 位元識別碼已足夠讓下個千禧年的每一個人在每一秒鐘製造出來的介面卡使用（如果真是這樣，大概宇宙中所知的矽、氧、金都被消耗光了，銀河的重力場也將崩潰）。每一塊卡上還包含 8 位元的檢查碼（checksum)，以確定識別碼無誤。（有關科幻小說家 Douglas Adams 已經保留廠商識別碼 0000002Ah 的傳聞，並不正確）

介面卡區隔（card isolation），是一種高科技的 "odd man out" 遊戲（譯註：一種擲錢幣遊戲）。爲了區隔介面卡，系統軟體從 Plug and Play 的 READ_DATA port 讀入 144 個

bytes（64 位元識別碼及 8 位元檢驗碼，每一位元需 2 bytes，一共需要 144 bytes）。視電腦所允許的 I/O configuration 而定，READ_DATA port 可以位在 203h~3FFh 之間。Plug and Play 規格中對於「作業系統如何不斷地嚐試不同的 port 位址，以找出電腦製造商所設定的位址」有所描述。

介面卡的區隔（isolation）步驟，係從 "sleep" state 開始。軟體送出一個 *WAKE(0)* 命令，令所有尚未區隔的卡（也就是其 CSN 仍爲 0 者）進入 isolation state（請看圖 **11-1**）。在軟體讀入資料的過程中，每一塊介面卡會查看本身識別碼的位元（由低至高），以判斷如何回應：

- 1-bit 若查看的位元值爲 1，則介面卡送出 55h 以回應接下來的 even-bytes 讀取，或送出 AAh 以回應接下來的 odd-bytes 讀取。
- 0-bit 若查看的位元值爲 0，則介面卡會監視 data bus 上最後的兩個位元（而不是送資料到 bus 上）以明白其它介面卡在做什麼事。如果它發覺其它介面卡回應 01 或 10（即 55h 和 AAh 的最後兩個位元），它會退出這場區隔（isolation）遊戲，回到 sleep state。

由於介面卡只針對識別碼裡頭的 1-bits 產生資料到 bus，所以這個演算法在「所有介面卡皆爲 0-bit，只有一塊卡是 1-bit」時，區隔出第一塊介面卡。如果系統軟體需要其識別碼，可以把讀到的 55h AAh 數列以 1-bit 表示，把任何其他的 2-byte 數列以 0-bit 表示，重建出介面卡的識別碼。無論如何，這已足夠讓我們知道介面卡的識別碼，因爲當演算法結束時只有一個介面卡會停留在 isolation state。

讓我們考慮一個簡化的例子。假設介面卡有 4 位元的識別碼並且沒有檢查碼（checksum），而我們有三塊介面卡 A，B 及 C，其識別碼分別爲 1010h，1011h 以及 0111h。圖 **11-2** 顯示軟體逐步區隔出介面卡 C，然後是介面卡 B，然後是介面卡 A。再來的區隔程序便會產生 0000 ID，表示沒有其他 Plug and Play 裝置存在。

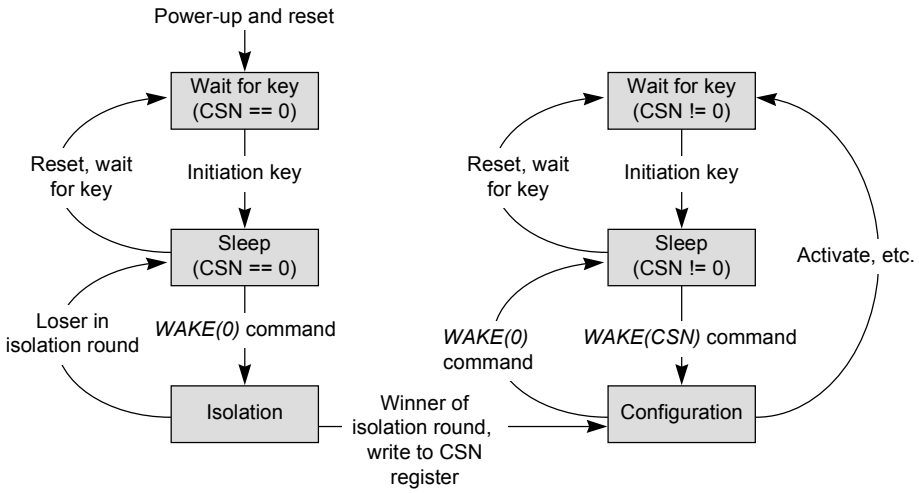


圖 11-1　ISA Plug and Play 介面卡的狀態（state）圖

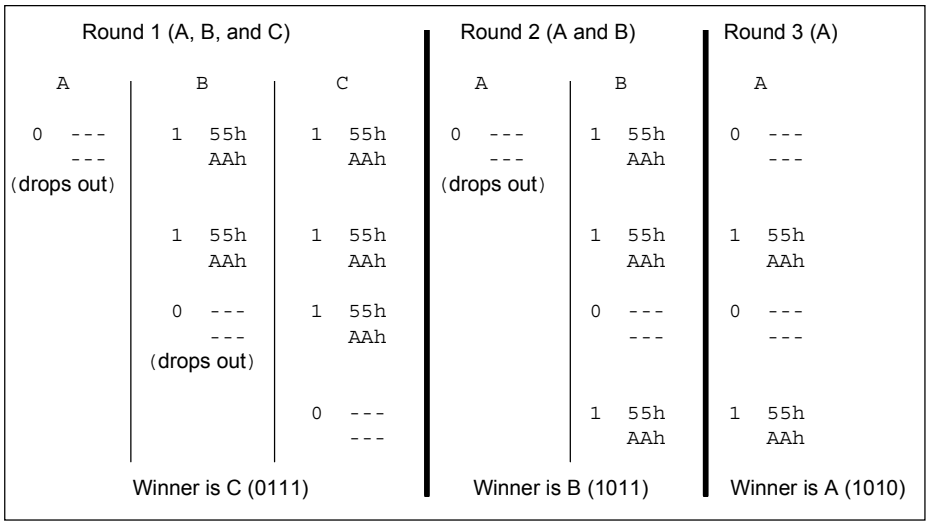


圖 11-2　ISA 介面卡的區隔演算法（isolation algorithm）（簡化情況）

成功區隔出一塊介面卡後，軟體爲它指定一個 8 位元的 CSN，當作以後運作時所需的 handle。藉著 *WAKE* 命令與 CSN，作業系統接下來可以使介面卡進入 configuration state。一旦進入 configuration state，介面卡將會發出資源需求（IRQ，DMA，memory 及 port），然後接受 configuration 指令。由於任何介面卡係以電子線路來表示自己的資源需求，所以不需要製造商特別提供什麼 configuration 檔案。

EISA Bus

在 Extended ISA（EISA）系統中，PC 上的每一個擴充槽（expansion slot）有屬於自己獨一無二的 I/O port 位址範圍。每一個 EISA 介面卡並包含電子線路用以設置（configuring）介面卡可能需要的額外資源。EISA 介面卡製造商必須提供一張磁片，內含 configuration 檔，詳細描述這個介面卡的資源需求。Configuration 工具軟體可以檢查 configuration 檔，指出如何協調所有介面卡。這個工具然後會在 nonvolatile memory（非揮發性記憶體，像是由電池供電力的 CMOS 記憶體）中記錄 configuration。在開機自我測試（POST）期間，system BIOS 會讀取此 configuration 資料，檢查正確的介面卡是否在所預期的擴充槽（slot）中，並動態設置（configures）此介面卡。

軟體可以藉由從 port xC80h 至 xC83h 所讀取的 byte，來確認 EISA 介面卡是否位在 slot x 上。傳回的資料表示 5 個被壓縮的 bytes：3 個 ASCII 字母用來表示製造商代碼，4 個十六進制數字（2 bytes）用來表示裝置和修訂號碼，如**圖 11-3** 所示。由於資料在記憶體中的排列次序是 "big-endian"，所以這張簡圖看起來是如此特別。

Windows 95 繼續仰賴 BIOS 來確認 EISA 擴充介面卡。也就是說，當所謂的 Configuration Manager 啓動時，它只會發現記錄在 nonvolatile memory 的 devices。如果要增加新的 EISA 介面卡，使用者必須執行 configuration 工具，然後重新開機。

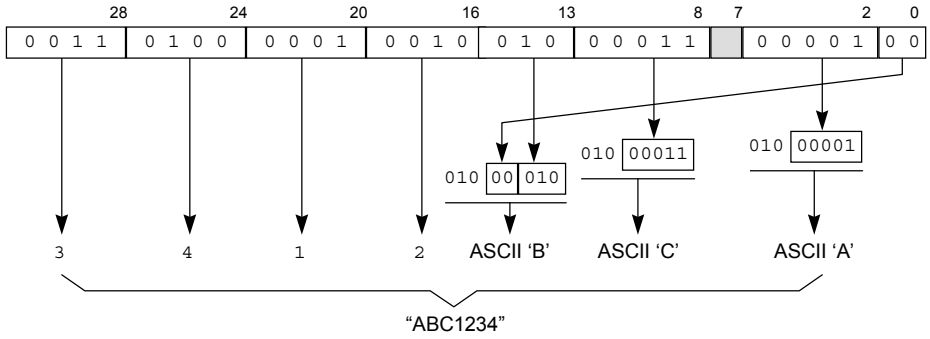


圖 11-3　被壓縮的 EISA 介面卡識別碼

MCA Bus

IBM PS/2 電腦的 Micro Channel Architecture（MCA）bus 使用一個十分受限的介面卡（adapter）識別方法。每一個 MCA 介面卡有 8 個 programmable option select（POS）暫存器，且可由軟體定址於 ports 100h ~ 107h。軟體藉由寫入 I/O port 096h 的 adapter activation register（如圖 **11-4** 所示）來選擇特定的 POS 暫存器，然後軟體可以從 ports 100h 和 101h 讀取 16 位元的識別資料。目前 IBM 負責維護介面卡識別碼的註冊。

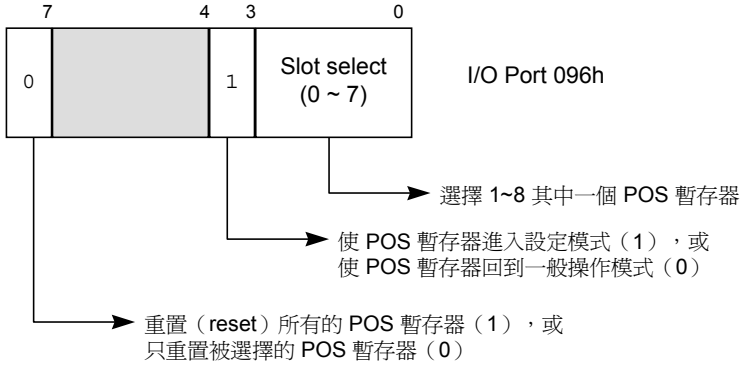


圖 11-4　MCA 的 adapter activation register

除了 16 位元識別碼以外，MCA 介面卡本身並不包含任何其他資訊（例如所需資源之指示等等）。每塊介面卡附有一磁片，其中包含一組設定檔（setup files），用以詳細說

明如何設置（configure）介面卡。設定用的工具程式（setup utility）會解析這些檔案內容，並在 CMOS RAM 裡記錄必要的 configuration 資料。當使用者啓動機器（booting），system BIOS 即從 CMOS 中將 configuration 複製到介面卡的 POS 暫存器。除了確認預期的介面卡是否位於所設置的 slot 上面，BIOS 基本上並不會檢查其他的 configuration 資料。

你可以花相當高的費用從 Micro Channel Developers Association (916)222-2262 取得 MCA bus 及其介面卡的更多相關資訊。你需要一些手冊（每一份價值 US$50 ~ US$100）以徹底了解 bus、POS 暫存器、及 setup utility。更好並且更經濟的瀏覽方法是參考 Hans-Peter Messmer 所著的 *Indispendable PC Hardware Book*：*Your Hardware Questions Answered*（Addison-Wesley，1994），pages 265~281。

PCI Bus

Peripheral Component Interconnect（PCI）local bus 定義有一個自我辨識用的 devices 硬體標準。接著於 PCI bus 上的 devices，有一個 256 bytes 的 configuration space （**圖 11-5**），包含以下資料：

- 一個 16-bit 廠商識別碼
- 一個 16-bit 裝置識別碼及 8-bit 版本識別碼
- 一個 24-bit 裝置類別碼（device class code）
- 最多 6 個 I/O addresses 或 memory base addresses。
- 一個可選擇的 ROM address
- 一個 IRQ 號碼

軟體有兩種方法，可以查詢特定的 configuration 暫存器。比較好的方法是首先將一個 32 位元的 device and register address（**圖 11-6**）寫入 CONFIG_ADDRESS 暫存器中（這是 I/O 位址 CF8h 處的一個 32 位元 port），然後從 CONFIG_DATA（這是 I/O 位址 CFCh 的一個 port）讀取暫存器的內容。你可以以位址間隔 0,4,8...的方式來讀取 64 個暫存

器，以便獲得某一 device 的所有 256 bytes configuration space。你可以讀取 device number 0~31 的資料，列舉出所有接著於 PCI bus 的裝置。

```
typedef struct tagCONFIG {     // PCI configuration space
    WORD vendorid;             // 00h vendor ID
    WORD deviceid;             // 02h device ID
    WORD command;              // 04h command register
    WORD status;               // 06h status register
    BYTE revision;             // 08h revision ID
    BYTE class[3];             // 09h class code
    BYTE cachesize;            // 0Ch cache line size
    BYTE latency;              // 0Dh latency number
    BYTE hdrtype;              // OEh header type
    BYTE bist;                 // 0Fh built-in self-test control
    DWORD baseaddr[6];         // 10h base address registers
    DWORD cis;                 // 28h CardBus CIS pointer
    WORD subvendor;            // 2Ch subsystem vendor ID
    WORD subsystem;            // 2Eh subsystem
    DWORD romaddr;             // 30h expansion ROM base address
    DWORD reserved[2];         // 34h reserved
    BYTE irq;                  // 3Ch interrupt line
    BYTE    ipin;              // 3Dh interrupt pin
    BYTE min_gnt;              // 3Eh minimum burst on 33 MHz bus (1/4-
                               //    microsecond units)
    BYTE max_lat;              // 3Fh maximum latency on 33 MHz bus (1/4-
                               //    microsecond units)
    BYTE device[196];          // 40h device-specific information
} CONFIG, *PCONFIG;            // [100h] PCI configuration space
```

圖 11-5　PCI device 的 configuration space

高雲慶註：在此我補充說明上述 vendorid，deviceid，subvendor，subsystem 四者之間的關係，因爲它們在 Windows 98 中特別重要，而本書原著只提到 Windows 95 對於廠商識別碼（vendorid）和裝置識別碼（deviceid）的使用。以 Intel 740 3D 繪圖卡爲例，Intel 的 vendorid 爲 8086h，i740 的 deviceid 爲 7800h；而 ASUS 的 vendorid 爲 1043h，v2740 的 deviceid 爲 0100h。因此，ASUS 以 Intel i740 chip 所製造的 v2740 3D 繪圖卡就可依此識別出來。Windows 95 僅以 vendorid 和 deviceid 來辨識，而 Windows 98 依上述法則不僅可辨識出 chip 類別，更可辨識出 adaptor board 的製造商。如此一來，系統才可有效辨識出同樣使用 i740 chip 的不同廠商（例如 ASUS 和 Diamond）所製造的繪圖卡。

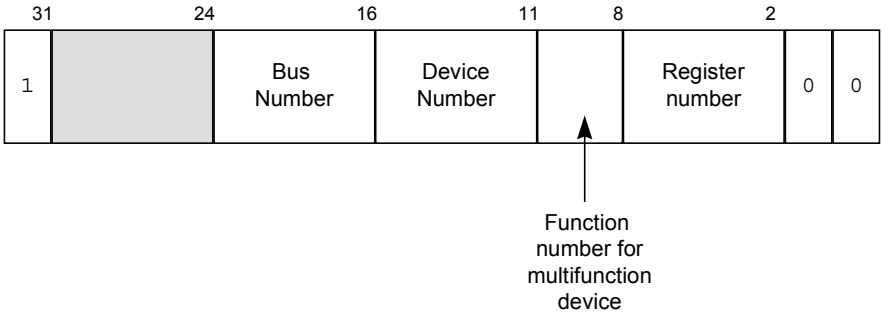


圖 11-6 PCI configuration address register 中的位元分配

另一個取得 configuration 資料的方法，純粹是爲了回溯相容。首先將一個 16-bit key 寫入 port CF8h，用以 "enable" configuration space，然後在 I/O ports C000h~CFFFh 處讀取 configuration 暫存器（每個 bus 最多 16 個 devices）。當 configuration space 被 "disabled"，這些 ports 的功能如同一般的 I/O ports。注意，根據你在 CF8h 寫入的是 4 bytes 或 2 bytes，你會得到不同的結果。

你可以從 PCI Special Interest Group（美國境內請打 (800)433-5177，美國境外請打 (503)797-4207），取得一份 PCI bus 標準，費用大約是 $25。除此之外，你可能還需要 PCI BIOS 及 PCI-to-PCI Bridge 規格。

Configuration Manager（組態管理器）

有一種新的 Windows 95 元件（component），稱爲 **Configuration Manager** (CONFIGMG)，用來控制裝置驅動程式的動態載入（dynamic loading）和硬體的設置（configuration）。其實甚至在 Windows 95 發行以前，新的硬體標準和軟體標準就已經開始運作了。如你所知，電腦啓動時需要某些裝置在場，例如顯示器、鍵盤、磁碟機，這些裝置即使能夠動態的重新設置（re-configuration），也必須以一組合理的預設值來啓動。爲了這個目的，Plug and Play 硬體規格允許製造商使用 jumpers 或 DIP switches。

如果你的電腦正好有一個爲了 Plug and Play 而設計的新型 BIOS，當你啓動電腦，這個 BIOS 會自動爲你設置部分系統。爲了保持最大彈性，BIOS 應該在系統運作的階段將沒有必要的 devices 給 "disable" 掉。然後，當 Windows 95 最終於保護模式中接管控制權，Configuration Manager 就可以將許多情報運用於整個系統的設置工作上。

System Registry（系統登錄資料庫）

在引導 Configuration Manager 完成其工作方面，registry 扮演了一個關鍵角色。Windows 95 registry 是一個階層化的資料庫，其中每個節點（node）是一個「可能擁有 named values 及 subkeys」的 **named key**。**Named values**（譯註）是一對對的「names 及 binary（或 ASCII）values」，而 **subkeys** 則是本身包含「更深層的 named values 及 subkeys」的 named objects。節點之中也可能只有一個 **unnamed value**，但是 Windows 95 系統軟體並不會去利用這種資料。Windows 95 的 Setup 程式一開始會建立一個 registry，並將它所發現的硬體環境和軟體環境的資料放入 registry。此後即由各種不同的系統元件來維護這個 registry。使用者可以藉由 REGEDIT.EXE 來檢查及修改 registry。圖 **11-7** 是 REGEDIT 的執行畫面，其中所顯示的是 registry 中非常重要的一個分支：HKEY_LOCAL_MACHINE（簡稱 **HKLM**）。

> 譯註：**named value** 代表成對的資料項名稱（name）及其值（value）。爲求精簡通暢，本書皆保留原文名詞 named value。

由於 registry 對 Windows 95 的系統管理如此重要，很值得我們好好探討一下這個資料庫的各個分支：

HKEY_CLASSES_ROOT	OLE class 的相關資訊（等同於 HKLM\Software\Classes）。
HKEY_CURRENT_USER	訂製型（customization）資訊，配合此電腦目前使用者的 logon ID（那是 HKEY_USERS 分支下的一個別名(alias)）。
HKEY_LOCAL_MACHINE	硬體相關資訊。
Config	硬體 profiles 的細節。
0001	硬體 profile 1 的細節。

Enum	所有曾經安裝於此電腦的硬體的相關資訊。
enumerator	被所謂的 bus enumerator 所找到的 devices(例如 ISAPNP、PCI 等）的相關資訊。
device-id	某個 device 的相關資訊。
instance	父類型（譯註：指上一行 device-id）之下的某個 device 的相關資訊。這個 key 也被稱爲 device 的 *hardware* key。其 named value "Driver" 特別重要。
ROOT	legacy（老舊的）device 的相關資訊。
**device-id*	某種 device 的相關資訊。例如 *PNP0500 與串列埠（serial ports）有關。
0000	某種型態之下的第一個 device 的相關資訊。這個 key 也被稱爲 device 的 *hardware* key。其 named value "Driver" 特別重要。
LogConfig	device 的邏輯組態（logical configuration）的相關資訊。
Software	應用程式的相關資訊。這份資料可以取代 INI 檔案。
Classes	OLE class 的相關資訊。
Manufacturer	此系統上某一廠商的軟體資料。
ProductName	某項產品的資料。
CurrentVersion	此一產品目前的版本。
version	此一產品的其它版本的相關資訊。
System	可用的 devices 的詳細資料。
CurrentControlSet	可用的 devices 的詳細資料。
Control	可用的 devices 的詳細資料。
IDConfigDB	硬體 profiles 的相關資訊。其 named value "FriendlyNamennnn" 特別重要。
Services	裝置驅動程式的相關資訊。
Arbitrators	資源仲裁器（resource arbitrators）的控制參數。
Class	每一種裝置的相關資訊。
classname	某種 device 的相關資訊。
0000	某種 device 的第一筆資料（第一個 device）。這個 key 也稱爲 device 的 *software* key。其 named values 如 DevLoader,

	Enumerator, EnumPropPages, PortDriver 等等，對 Configuration Manager 而言很重要。
VxD	static device drivers（靜態載入之 device drivers）的相關資訊。
name	關一特定 driver 的資訊，涵蓋原本在 SYSTEM.INI 檔案裡頭的各項設定。其 named value "StaticVxD" 指出一個被靜態載入的 driver 的檔案名稱。
HKEY_USERS	這部電腦的所有使用者的相關資訊。
HKEY_CURRENT_CONFIG	目前硬體 profile 的相關資訊。這是 HKLM\Config 某些分支的別名。
HKEY_DYN_DATA	一些動態變化的資料，可由 registry API 函式存取之。
Config Manager	由 Configuration Manager 維護的資料。
Enum	硬體樹狀圖（hardware tree）。
xxxxxxxx	在特定線性位址上的 DEVNODE。
PerfStats	效率記錄器（Performance counters）。

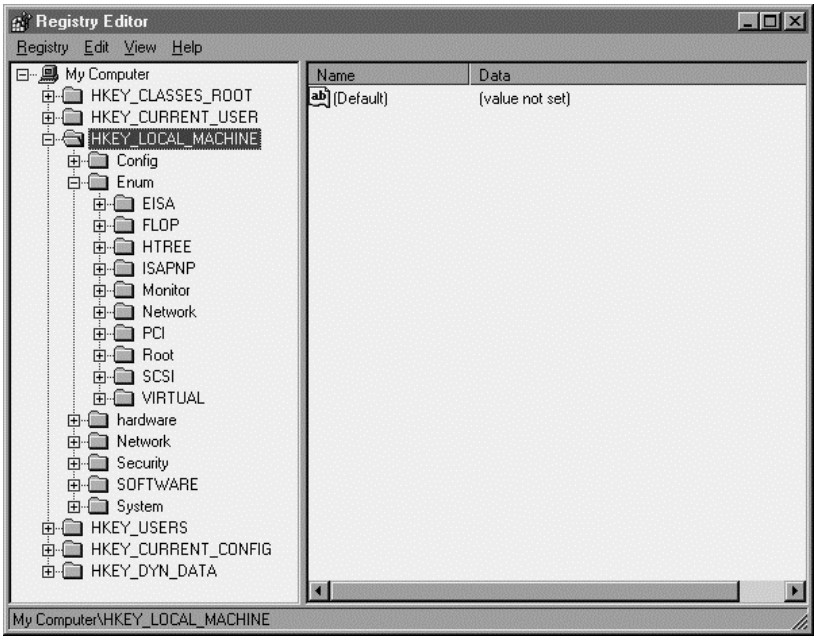


圖 11-7 Registry 的 HKEY_LOCAL_MACHINE 分支

Plug and Play 驅動程式的機制

若想要建立驅動程式，使其成爲 Configuration Manager 系統的一部份，有一些特別的要求必須遵守。其中有不少內容，未經解釋實在難以理解。Microsoft DDK 允許你以高階語言撰寫 Plug and Play 驅動程式，所以我決定提供一個以 DDK/C 撰寫而成的範例。

Plug and Play 驅動程式是一個 dynamic VxD，因此你需要提供 *Sys_Dynamic_Device_Init* 及 *Sys_Dynamic_Device_Exit* 這兩個系統控制訊息（system control messages）的處理常式。這些處理常式應該簡單地傳回 TRUE 以示成功：

```
BOOL OnSysDynamicDeviceInit()
   {                    // OnSysDynamicDeviceInit
   return TRUE;
   }                    // OnSysDynamicDeviceInit

BOOL OnSysDynamicDeviceExit()
   {                    // OnSysDynamicDeviceExit
   return TRUE;
   }                    // OnSysDynamicDeviceExit
```

DDK 提供了一些表頭檔，對於開發一個 Plug and Play 驅動程式頗有幫助：

```
#define WANTVXDWRAPS
#include <basedef.h>
#include <vmm.h>
#include <debug.h>
#include <vmmreg.h>
#include <vxdwraps.h>
#include <configmg.h>
#include <regstr.h>
```

其中的 VMMREG.H 宣告了一些 services，用來檢查及修改 registry。REGSTR.H 針對很多複雜的 registry pathname，定義了容易明瞭的常數。CONFIGMG.H 宣告 Configuration Manager 的介面。我在第 7 章曾經指出「所有 VxD 表頭檔應該在 VXDWRAPS.H 之前含入」，這個規則並不適用於 REGSTR.H 及 CONFIGMG.H。

DDK 定義了一些以 *_CONFIGMG_* 爲字首的 Configuration Manager services，這些就是你在 VxD 中以 C 呼叫風格所呼叫的、字首爲 *CONFIGMG* 的 VxD services。然而，撰寫驅動程式時，你不能使用文件上的名稱，應該以 *CM_* 字首取代 *_CONFIGMG_* 字首。因此，如果要呼叫 *_CONFIGMG_Register_Enumerator* service，你真正寫的程式碼是呼叫 *CM_Register_Enumerator*。（注意，這些 services 的文件中甚至使用一個開頭沒有底線的 *CONFIGMG_* 字首，真是混亂極了）

Configuration Manager 藉由呼叫一個或多個特別的 callback 函式來與 Plug and Play driver 溝通。Configuration Manager 需要某種方法以得知這些 callback 函式的位址。然而 Windows 只能經由 VxD *Declare_Virtual_Device* 巨集中的 device control procedure 來存取 VxD。因此，你應該在 device control procedure 裡撰寫處理 *PNP_New_Devnode* 這一系統控制訊息的程式碼。Configuration Manager 會呼叫 *VXDLDR_LoadDevice* 動態載入你的驅動程式。VxD loader 會傳回驅動程式的 DDB（device-descriptor block）位址，Configuration Manager 於是呼叫 *Directed_Sys_Control* 以傳送 *PNP_New_Devnode* 訊息給你。因此，你可以使用一個或多個 registration API 函式，將你的 callback 處理常式告知 Configuration Manager。

PNP_New_Devnode 訊息有兩個參數。一個參數是 device node 的位址，簡稱爲 DEVNODE。另一個參數是 load type code（見表 **11-1**），用以表示 Configuration Manager 對你的驅動程式的期望（期望做些什麼）。如果要處理這個訊息，C 程式碼應該看起來像下面這樣，其中的傳回值 *CR_DEFAULT* 表示你希望以預設方式來處理你不想要處理的 load type code：

```
CONFIGRET OnPnpNewDevnode(DEVNODE devnode, DWORD loadtype)
   {                    // OnPnpNewDevnode
   ...
   return CR_DEFAULT;
   }                    // OnPnpNewDevnode
```

PNP_New_Devnode Load Type Codes	說明
DLVXD_LOAD_ENUMERATOR	你的驅動程式被視爲一個 enumerator
DLVXD_LOAD_DEVLOADER	你的驅動程式被視爲一個 device loader
DLVXD_LOAD_DRIVER	你的驅動程式被視爲一個 device driver

表 11-1 PNP_New_Devnode 訊息的原因

在表 **11-1** 所示的三種可能中，Configuration Manager 希望你採取以下的對應動作：

譯註：以下常出現的「登錄」一詞，原文爲 register，有時被譯爲「註冊」。

- **Enumerator** 呼叫 *CM_Register_Enumerator* 登錄（註冊）一個 enumeration 函式。
- **Device Loader** 呼叫 *CM_Load_DLVxDs*，對特定的 DEVNODE 動態載入其 driver。這個呼叫的副作用是送出「有著 *DLVXD_LOAD_DRIVER* 型態碼的 *PNP_New_Devnode* 訊息」給最近載入的驅動程式。傳回 CR_SUCCESS 尚不足以讓 Configuration Manager 相信你已成功地成爲一個 device loader，有些東西必須一併登錄才行。
- **Device Driver** 呼叫 *CM_Register_Device_Driver* 以登錄一個 configuration 函式。你可以登錄一個 NULL 的 configuration 函式，意謂著 Configuration Manager 將會對所有 configuration 的 events，執行預設的處理動作。

我建議你最好在驅動程式內含入一份版本資源（version resource），如此驅動程式將能夠更充分的配合整個系統架構。舉個例子，我建立下面兩個檔案，做爲範例的一部份（這些檔案被放在書附光碟的 \CHAP11\SCHOOLBUS 目錄之中）：

SCHOOL.RC

```
#0001  ; SCHOOL.RC
#0002
#0003  #include <winver.h>
```

```
#0004  #include <version.h>
#0005
#0006  VS_VERSION_INFO VERSIONINFO
#0007  FILEVERSION     VERMAJOR, VERMINOR, 0, BUILD
#0008  PRODUCTVERSION  VERMAJOR, VERMINOR, 0, BUILD
#0009  FILEFLAGSMASK   VS_FFI_FILEFLAGSMASK
#0010  FILEFLAGS       VS_FF_DEBUG | VS_FF_PRERELEASE
#0011  FILEOS          VOS_DOS_WINDOWS32
#0012  FILETYPE        VFT_VXD
#0013  FILESUBTYPE     0
#0014  BEGIN
#0015      BLOCK "StringFileInfo"
#0016      BEGIN
#0017          BLOCK "040904E4"
#0018          BEGIN
#0019              VALUE "CompanyName", "Walter Oney Software\0"
#0020              VALUE "FileDescription", "School Bus Driver\0"
#0021              VALUE "FileVersion", PRODVER
#0022              VALUE "InternalName", "SCHOOL\0"
#0023              VALUE "LegalCopyright", \
#0024                   "Copyright (C) 1995 by Walter Oney Software\0"
#0025              VALUE "OriginalFilename", "SCHOOL.VXD\0"
#0026              VALUE "ProductName", "School Bus Sample Program\0"
#0027              VALUE "ProductVersion", PRODVER
#0028          END
#0029      END
#0030      BLOCK "VarFileInfo"
#0031      BEGIN
#0032          VALUE "Translation", 0x409, 1252
#0033      END
#0034  END
```

VERSION.H

```
#0001  // VERSION.H
#0002
#0003  #define VERMAJOR  1
#0004  #define VERMINOR  0
#0005  #define BUILD     003
#0006
#0007  #define PRODVER   "1.0.003\0"
```

當終端使用者啓動 Windows 95【控制台】（Control Panel）中的【系統】（System）項

目，並查詢 School Bus device 的屬性（properties）時（圖 **11-8**），使用者會看到上述些資料。爲了在你的可執行檔中加入版本資源，你必須使用 16 位元的資源編譯器，以及一個被稱爲 ADRC2VXD 的 DDK 工具。你的 NMAKE 劇本可能包含以下內容：

```
#0001  all: school.vxd
#0002  ...
#0003  .rc.res:
#0004     c:\msvc\bin\rc -r $*.rc   # must use RC16 for this
#0005
#0006  school.vxd : s_ctl.obj $*.obj $*.def $*.res
#0007     link @<<
#0008  -machine:i386 -def:$*.def -out:$@
#0009  -debug -debugtype:map
#0010  -map:$*.map -vxd vxdwraps.clb -nodefaultlib
#0011  s_ctl.obj $*.obj
#0012  <<
#0013     adrc2vxd $*.vxd $*.res
```

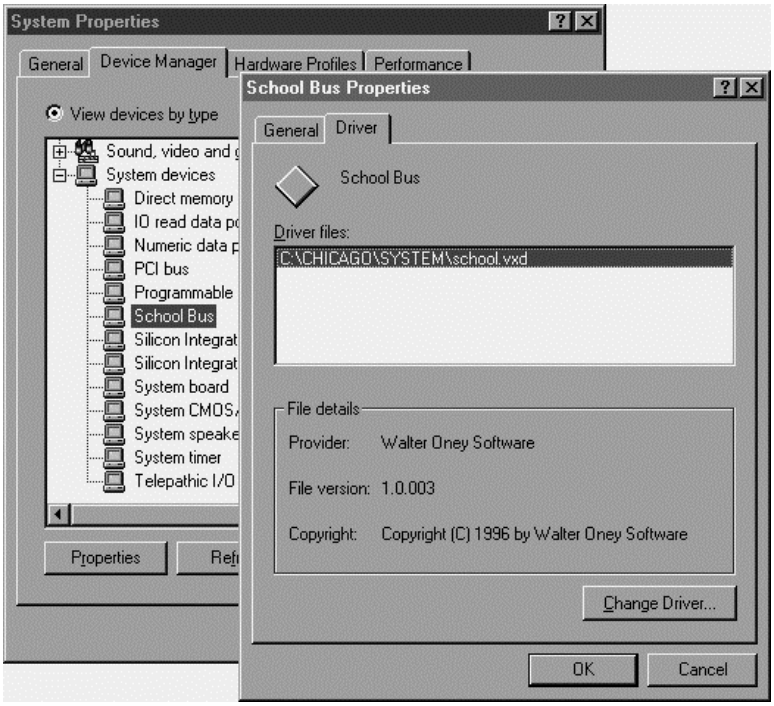


圖 11-8　控制台（control panel）顯示出版本資源

載入正確的裝置驅動程式

爲了反映實際硬體，Configuration Manager 將電腦塑造成「包含一個或多個硬體」的 bus，有 devices 附著於其上。Configuration Manager 會建立起一個階層化的資料結構，稱爲硬體樹狀圖（**hardware tree**），用來描述 buses 和其上的 devices。樹狀圖的節點（nodes）叫做 DEVNODEs。此樹狀圖可能有數個節點的深度，因爲某些 devices 可能其實是附著於其它 devices 身上。例如 SCSI 控制器接到一個 bus 上，但它用來控制其它的 devices。

Configuration Manager 十分依賴 registry 來建立 hardware tree。其 root enumerator（Configuration Manager 的內部元件）首先將 HKLM\ENUM\ROOT 所列出的 devices 統統加入 hardware tree。如此的 device 應該是在這部電腦的 primary bus 上面。例如我目前正在使用的系統之中，有一個 PCI local bus 做爲其 primary bus。HKLM\ROOT\ENUM 也列出所謂的 "legacy"（傳統的、老舊的）devices。"Legacy" devices 包括標準元件如 Direct Memory Access（DMA）Controller 及 Programmable Interrupt Controller（PIC）。由於這些元件被歸類爲 Plug and Play devices，所以 Configuration Manager 並不需要有特別的程式碼就可以爲每個 Windows 95 session 設定它們。所謂的 "Legacy" devices 還包括一些不容易被自動偵測到的 devices，像是鍵盤和標準序列埠（standard serial ports）。Windows 95 Setup 程式使用一個複雜（更別說費時以及脆弱）的偵測演算法來定位這些 legacy devices。Configuration Manager 每次啓動你的電腦時如果需要重新執行偵測演算法，那不只是愚蠢，而且不經濟，風險亦高。

當 root enumerator 結束時，hardware tree 包含所有 legacy devices 的 DEVNODEs。其中一些 DEVNODEs 應該與 enumerators 有關係。Configuration Manager 呼叫這些額外的 enumerators 來擴大 hardware tree。Enumerator 還可以發現更多的 devices，其中有較低階的 devices 連接於上。舉個例子，我的電腦有一個 ISA bus，透過橋接器（bridge chip）連接到 PCI local bus，所以 PCI bus enumerator 會對這個 ISA bus 建立一個 DEVNODE。ISA bus enumerator 然後會取得機會來檢查 ISAPNP 介面卡。這個列舉程序以遞迴的方法繼續進行，直到 hardware tree 反映出電腦真實的形態（topology）。

裝置識別碼（Device Identifiers）

Configuration Manager 以一個「由倒斜線切割爲三部分」的 ASCII 字串來確認 devices，這個字串在任何一台電腦裡都是獨一無二的。Device ID 不但是 device 的 DEVNODE 名稱，也是用以描述 device 的那個 registry key 的名稱。字串的第一部份指出用以偵測 device 之 enumerator（如 ISAPNP 或 PCI）。第二部份指出 device 類型，第三部份指出此類型之 device 在電腦上的一個實體（instance）。

例如，在我的電腦上，IDE 硬碟的 ID 字串爲 PCI\VEN_1095&DEV_0640\BUS_00&DEV_0D&FUNC_00。其中 PCI 表示 bus，VEN_1095&DEV_0640 表示 device type（也就是廠商編號爲 1095 的一塊 PCI-IDE bridge），BUS_00&DEV_0D&FUNC_00 則表示這個 device type 的一個實體（instance）。instance 識別碼其實只是「內含於 PCI configuration space 中的資料」的連鎖表示而已，它表示在編號 0h 的 PCI bus 上，device 0Dh 的 function 0。這個磁碟裝置的 registry key 是 HKLM\Enum\PCI\VEN_1095&DEV_0640\BUS_00&DEV_0D&FUNC_00。

> 高雲慶註：上述所說的 PCI ID 是 Windows 95 表示法；在 Windows 98 中尚包括 subvendor ID 及 subsystem ID。例如：
>
> `VEN_8086&DEV_7800&subsystem_1043&subven_0100`。

Device ID 時常以 * 開始，用來表示它們是 EISA 識別碼。EISA 識別碼包含 3 個字母（代表廠商字首），以及 4 個十六進位數字。Device 製造廠商可以從 BCPR Services 公司（位於 Spring, Texas, USA）所集中維護的 registry 之中取得一個 EISA 字首。對於缺乏「標準 EISA IDs」的 devices，以及「不適用 EISA IDs」的 devices，Microsoft 爲它們保留字首 *PNP*，並指定一個 pseudo-IDs 給大多數的 devices。舉個例子，以 *PNP0A00 起始的 device ID 就是一個 ISA bus。表 **11-2** 列出 hardware buses 上可能的 device IDs。本書附錄列有所有 device 種類的完整列表。你也可以從 CompuServe PLUGPLAY forum 下載一個名爲 DEVIDS.TXT 的檔案，取得最新列表。

Bus Enumerators

爲了說明 Configuration Manager 如何處理 buses，我將提供一個簡單的例子。此例純爲教學之用，所以我將示範如何支援一個稱爲 School Bus 的新的 hardware bus。我假設已申請此虛構之 device ID 爲 PNP0A05。由於我所準備的 device information file 內容之故（我將在本章後面加以說明），Windows 95 Setup 程式會把 bus driver（SCHOOL.VXD）複製到 SYSTEM 目錄，並建立如圖 **11-9** 的 registry entries。

Device ID	說明
PNP0A00	Industry Standard Architecture（ISA）bus
PNP0A01	Extended Industry Standard Architecture（EISA）bus
PNP0A02	Micro Channel Architecture（MCA）bus
PNP0A03	Peripheral Component Interconnect（PCI）bus
PNP0A04	Video Electronics Standards Association Local（VL）bus

表 11-2 Hardware bus device identifiers

當 root enumerator 在 HKLM\Enum\Root 的列舉過程進行到 *PNP0A05 key 時，它會讀取 named value "Driver" 以找出 bus enumerator 的位置。在圖 **11-9** 中，這個值是 System\0011 -- 因爲 School Bus 是我的電腦中的第 12 個「類別爲 System」的 device。於是 root enumerator 打開 HKLM\System\CurrentControlSet\Services\Class\System\0011（圖 **11-10**）。此 key 指出 enumerator 的 device loader（DevLoader）爲 SCHOOL.VXD。

關於 Pseudo-EISA 識別碼 如果你從書附 CD 片中安裝 School Bus 驅動程式，它將不會如書上所說成爲 HKLM\Enum\Root*PNP0A05，而是成爲 HKLM\Enum\SCHOOL。因爲只有經由自動偵測程序所建立的 registry entries，才會根據 pseudo-EISA 識別碼名稱，最終成爲 root 的分支。以人工方式加入的 devices entries，最後會成爲名稱更易識別的分支。爲了產生圖 **11-9** 的螢幕畫面，我用人工方式將自動產生的 registry key 重新命名。（譯註：我的實際安裝經驗是，的確會產生 HKLM\Enum\SCHOOL，但圖 **11-9** 的內容也會出現，出現於 HKLM\Enum\Root\System）

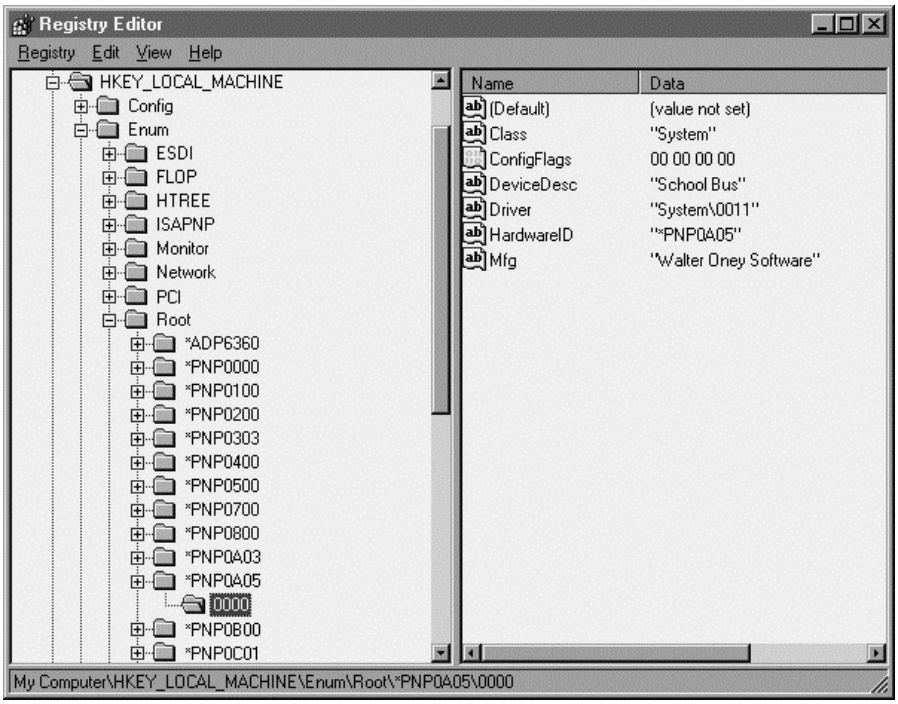


圖 11-9 School Bus 的 registry entries

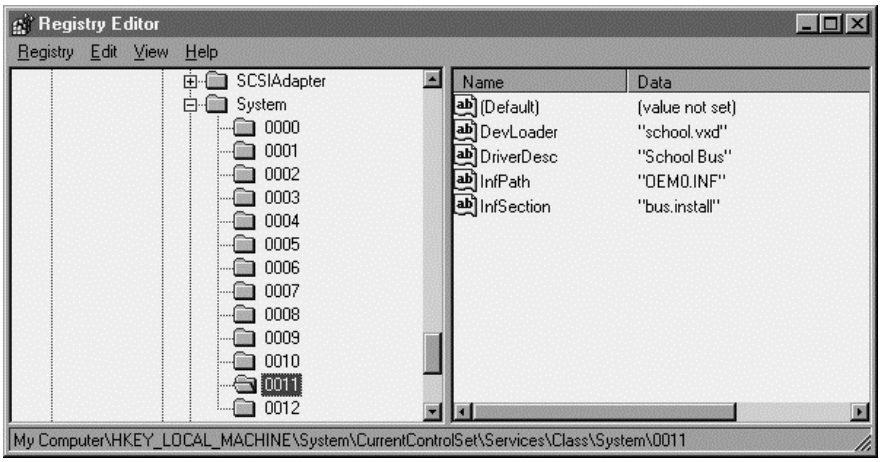


圖 11-10 School Bus device driver 的 registry entries

現在，Root enumerator 建立起一個 DEVNODE 來說明這個新 bus，並動態載入名爲 SCHOOL.VXD 的 device loader。這個 VxD 其實是 bus enumerator，但是 root enumeration 的處理邏輯要求它在這個階段扮演 device loader。SCHOOL.VXD 收到一個 *Sys_Dynamic_Device_Init* 訊息，對此訊息，一般是以「傳回 TRUE」回應之：

```
BOOL OnSysDynamicDeviceInit()
   {                    // OnSysDynamicDeviceInit
   return TRUE;
   }                    // OnSysDynamicDeviceInit
```

到目前爲止，我所說的只不過是動態載入 device driver 的一種特殊機構。接下來是 Plug and Play 所要發生的事情：Configuration Manager 送給 SCHOOL.VXD 一個 *PNP_New_Devnode* 訊息，指示它載入「表示此 bus 之 DEVNODE」的 driver。其實這個 driver（也就是 SCHOOL.VXD）已被載入（譯註：由 root enumerator 載入），但是 Configuration Manager 並不知道。SCHOOL.VXD 利用收到此訊息的機會，將本身登錄爲爲一個 bus enumerator：

```
#0001  CONFIGRET OnPnpNewDevnode(DEVNODE devnode, DWORD loadtype)
#0002     {                        // OnPnpNewDevnode
#0003     CONFIGRET code;
#0004     switch (loadtype)
#0005        {                     // select function to perform
#0006
#0007     case DLVXD_LOAD_DEVLOADER:
#0008        code = CM_Register_Enumerator(devnode, OnEnumerate,
#0009           CM_REGISTER_ENUMERATOR_HARDWARE);
#0010        if (code != CR_SUCCESS)
#0011           return code;
#0012        return CM_Register_Device_Driver(devnode, NULL, 0,
#0013           CM_REGISTER_DEVICE_DRIVER_REMOVABLE);
#0014        }                     // select function to perform
#0015     return CR_DEFAULT;
#0016     }                        // OnPnpNewDevnode
#0017
```

至此，SCHOOL.VXD 已經將本身登錄，表示自己是 School Bus DEVNODE 的

enumerator 及 device driver。我將在本章稍後完成 enumeration 機能。*OnEnumerate* 的基礎版本看起來像這樣：

```
CONFIGRET _cdecl OnEnumerate(CONFIGFUNC cf, SUBCONFIGFUNC scf,
   DEVNODE tonode, DEVNODE aboutnode, ULONG flags)
   {                          // OnEnumerate
   return CR_DEFAULT;
   }                          // OnEnumerate
```

注意，School Bus enumerator 回應 *PNP_New_Devnode* 訊息時似乎並不遵守我先前曾經提過的規則。它是一個 enumerator，但它只對 load type 爲 *DLVXD_LOAD_DEVLOADER* 的訊息做回應。而且又不遵循規則動態載入 drivers，反而執行另外兩種動作（譯註：指的是 load type 爲 *DLVXD_LOAD_ENUMERATOR* 及 *DLVXD_LOAD_DRIVER* 所該做的動作）。這些結果都是基於「想把此一 bus driver 的所有功能塞進一個 VxD 裡」的貪心念頭所製造成。Microsoft 自己的 PCI.VXD 正是以相同的方式運作，因此我爲你示範的這個狀況並不特別奇怪。

表 **11-3** 列出一個 enumerator 可以接收到的 configuration 訊息。面對不想處理的 enumeration 訊息，你不用僞稱處理成功；你可以傳回 *CR_DEFAULT* 要求執行預設動作。事實上，Configuration Manager 的 debug 版本，在呼叫 *CM_Register_Enumerator* 時，會送給 enumeration 函式一個沒有意義的 0x12345678 訊息，以明白是否會被正確回應。

Configuration Manager 會送出許多的「功能申請（function requests）」到 device 的 configuration 函式（下一章討論）以及擁有此 device 的 enumerator 去。這樣的重複性允許你以最適當的方法來分配「處理特定 events」的責任。例如，一個 device driver 可能認爲移除自己是沒問題的，但是 enumerator 可能會否決掉那個決定。

Enumeration 函式會收到兩個 DEVNODEs 參數。參數 *tonode* 用以表示此 bus 或其他 parent device 的 DEVNODE。參數 *aboutnode* 則通常表示 child device 的 DEVNODE。有一些函式只需要一個 DEVNODE，此時的 *tonode* 和 *aboutnode* 應該相同。

下面數頁將更詳細地說明 enumerators 的 configuration 機能。

訊息	說明
CONFIG_APM	通知 enumerator，發生一個電源管理（power management）的 event
CONFIG_CALLBACK	通知 enumerator，*CM_CallBack_Enumerator* 已被呼叫
CONFIG_ENUMRATE	通知 enumerator，列舉出直接的 children
CONFIG_FILTER	通知 enumerator，過濾(限制)其中一個 device 的 logical configuration
CONFIG_PREREMOVE	通知 enumerator 說有一個 device 將被移除（從 tree 的底部往上送）
CONFIG_PREREMOVE2	通知 enumerator 說有一個 device 將被移除（從 tree 的頂部往下送）
CONFIG_PRESHUTDOWN	通知 enumerator 說系統將要關機
CONFIG_READY	通知 enumerator 說有一個 device 已被設定（set up）
CONFIG_REMOVE	通知 enumerator 說有一個 device 正從系統中移除
CONFIG_SETUP	通知 enumerator 說有一個 device 已被首次設定
CONFIG_SHUTDOWN	通知 enumerator 說系統正在關機
CONFIG_TEST	詢問 enumerator 是否可以停止使用 configuration，或是否其中一個 device 可以被移除
CONFIG_TEST_FAILED	通知 enumerator 說先前的 CONFIG_TEST 失敗
CONFIG_TEST_SUCCEEDED	通知 enumerator 說先前的 CONFIG_TEST 成功

表 11-3 Enumerators 的 Configuration messages

CONFIG_APM 機能 Configuration Manager 送出 *CONFIG_APM* 訊息作爲 Advanced Power Management 協定的一部份。其 *subfunction* 參數（見表 **11-4**）表示這個 event 的發出原因。例如當你從【開始】功能表中選擇【暫停（Suspend）】命令時，Configuration Manager 會送出一系列如下的訊息給 drivers 及其 enumerators：

■ CONFIG_APM_TEST_SUSPEND 送給 enumerator 以便瞭解是否可以

"suspend"（虛懸、凍結、暫停）每一個附著的 device。

- CONFIG_APM_TEST_SUSPEND 送給每一個 device。
- CONFIG_APM_TEST_SUSPEND 送給 enumerator，以便瞭解是否可以 "suspend"（虛懸、凍結、暫停）bus 本身。

假設 suspend 訊息導至傳回 CR_SUCCESS，那麼 Configuration Manager 會再送出這一系列訊息：

- CONFIG_APM_TEST_SUSPEND_SUCCEEDED 送給每一個 device
- CONFIG_APM_TEST_SUSPEND_SUCCEEDED 送給每一個 device 的 enumerator
- CONFIG_APM_TEST_SUSPEND_SUCCEEDED 送給 bus 本身的 enumerator

子機能（subfunction）	內容
CONFIG_APM_TEST_STANDBY	詢問是否可以改爲 standby（低耗電）
CONFIG_APM_TEST_SUSPEND	詢問是否可以改爲 suspended（無需耗電）
CONFIG_APM_TEST_STANDBY_FAILED	表示上一次的 standby request 失敗
CONFIG_APM_TEST_SUSPEND_FAILED	表示上一次的 suspended request 失敗
CONFIG_APM_TEST_STANDBY_ SUCCEEDED	表示上一次的 suspended request 成功
CONFIG_APM_RESUME_STANDBY	表示 standby 作業結束
CONFIG_APM_RESUME_SUSPEND	表示 suspended 作業結束
CONFIG_APM_RESUME_CRITICAL	表示 critical 作業重新開始（resuming）

表 11-4 CONFIG_APM 的 subfunctions（子機能）

Enumerator 對於上述這些所謂的 TEST requests 的認可方式是，傳回 CR_SUCCESS。否決方式則是傳回 CR_FAILURE。

CONFIG_CALLBACK 機能 Configuration Manager 有一個不易了解的 service，用來回呼（calling back）一個 enumerator：

```
CM_CallBack_Enumerator(OnEnumrate);
```

在 enumerator 裡呼叫這個函式，會使得 Configuration Manager 針對每一個 DEVODE，以 *CONFIG_CALLBACK* 訊息呼叫上述指定的 enumeration 函式。換句話說，如果你的 School Bus 連接十個 devices，則呼叫 *CM_CallBack_Enumerator* 並指定以 School Bus 的 bus enumerator 函式，會引發唯一一個 *CONFIG_CALLBACK* 訊息，其中的 *tonode* 及 *aboutnode* 參數都指向 bus 的 DEVNODE。沒有任何一個連接於上的 devices 會被回呼（called back）。

就我所知，沒有任何人真的用過 *CM_CallBack_Enumerator*，所以你或許不必去操心這個 configuration 機能。

CONFIG_ENUMERATE 機能 *CONFIG_ENUMERATE* 訊息是 enumerator 所收到的最重要訊息。這個訊息要求 enumerator 致力使 hardware tree 能夠符合實際連接至此 bus 的各個 devices。假設 School Bus 可以有一個以上的 **telepathic**（有精神感應力的）I/O channels。這是虛構的 devices 可以讀取使用者的心思，它們擁有相同的 device ID WCO1234。真實情況下應該有某種方法，可以知道有多少個 devices 連接到此 bus。也許像先前所說的 ISA 或 EISA buses 那樣，使用某種協定，動態指出確實存在的 devices。或者，可能像 root enumerator 的行爲那樣，讀取偵測程式過去所遺留下來的 registry 各筆內容。

無論什麼 devices，enumerator 都應該爲每一個 device 製造出獨一無二的 device ID，並且建立 DEVNODEs。然後它應該指定 logical configuration 以說明 devices 的資源需求。我將在下一章討論 configuration 的程序。Enumerator 建立 DEVNODE 的部份過程可能看起來像這樣：

```
#0001  case CONFIG_ENUMERATE:
#0002      {
#0003      DEVNODE device;
```

```
#0004     CONFIGRET code = CM_Create_DevNode(&device,
#0005         "SCHOOL\\WCO1234\\0000", tonode, 0);
#0006     if (code == CR_SUCCESS)
#0007         {             // devnode added okay
#0008         [add configuration to new devnode]
#0009         }             // devnode added okay
#0010     else if (code == CR_ALREADY_SUCH_DEVNODE)
#0011         code = CR_SUCCESS;  // not an error
#0012     return code;
#0013     }
```

在這個例子裡，*CM_Create_DevNode* 爲 **telepathic device** 建立新的 device node，當作 bus 的 child。使用 WCO1234 做爲 device ID（由三部份組成）的中間部份，基本上只是一種巧合，因爲 Configuration Manager 無法從名稱本身蒐集任何情報。也就是說 device ID 在系統中是獨一無二的，它會被複製，從一個 session 複製到下一個 session。

這段程式碼把從 *CM_Create_DevNode* 傳回的 CR_ALREADY_SUCH_DEVNODE 錯誤視爲正常情形來處理。之所以如此，是因爲 Configuration Manager 可以在 Windows 95 session 之中多次呼叫 enumerator，以便偵測最新安裝的硬體。每次被呼叫，enumerator 必須確定 hardware tree 與安裝的硬體群數一致。有一些 enumerators，像是 PCMCIA 介面卡，知道硬體何時被移除，並且會即時呼叫 *CM_Remove_SubTree*，以便從 tree 之中移除不再使用的 DEVNODEs。

然而，大部份的 enumerators 並不知道使用者何時移除了其所對應的硬體。它們有一個自動方法，用來檢查其 devices 的存在。爲了在 enumeration 階段刪除 hardware tree 的多餘部分，這些 enumerators 依靠附屬於每個 DEVNODE 之下的記號來達成，如下所示：

```
#0001  case CONFIG_ENUMERATE:
#0002      CM_Reset_Children_Marks(tonode, 0);
#0003      [enumerate the bus]
#0004      CM_Remove_Unmarked_Children(tonode, 0);
#0005      return CR_SUCCESS;
```

CM_Reset_Children_Marks 會清除 *tonode* DEVNODE（代表 bus）裡頭的每個 child DEVNODE 的記號。在 enumerating bus 的過程中，每次呼叫 *CM_Create_DevNode* 都會

設定記號，即使 DEVNODE 已經存在。因此在 bus enumeration 完成以後，唯一未被做記號的那個 DEVNODE 就是表示在 enumeration 過程中未曾被拜訪過者。*CM_Remove_Unmarked_Children* 會刪除那些個 DEVNODEs，以刪除 tree 的多餘部分，使 tree 符合實際的硬體裝置。

但是 root enumerator 並不符合前面兩種描述：它既不知道硬體何時被移走，也沒有辦法在 re-enumeration 時決定硬體是否仍然存在。Root enumerator 完全不知道這些事，因爲它只是簡單的讀取 registry 以決定哪些 devices 可能存在。因此 root enumerator 送出 *CONFIG_VERIFY_DEVICE* 子機能碼到其 child nodes 的 device drivers 手中，而 child drivers 如果不是回應 CR_SUCCESS 表示硬體仍然存在，就是回應 *CR_DEVICE_NOT_THERE* 表示硬體不存在。

CONFIG_FILTER 機能 Configuration Manager 允許一個 enumerator 過濾（filter）其 child device nodes（連續的 *aboutnodes*）的組態（configuration）-- 在 device nodes 本身過濾完畢之後。過濾的目的是爲了要限制 device 的資源。例如，PCMCIA 介面卡可以要求 32 位元定址範圍內的任何一塊記憶體，但 PCMCIA enumerator 卻可能將其範圍限制在最初的 16MB 位址空間內。

CONFIG_PREREMOVE 和 CONFIG_PREREMOVE2 機能 這些機能表示 *aboutnode* DEVNODE 即將被移除。Configuration Manager 送出的 *CONFIG_PREREMOVE* 是從 DEVNODE 開始，循著 hardware tree 往上遞送；至於 *CONFIG_PREREMOVE2* 則是從 hardware tree 的頂端開始往下遞送。Enumerator 利用這兩個 events 時機，可以完成適當的清理（cleanup）動作。

CONFIG_PRESHUTDOWN 機能 Configuration Manager 送出這個訊息給 enumerator，做系統關機前的準備。*tonode* 及 *aboutnode* 參數都指向 enumerator 自己的 DEVNODE。這個訊息是在 enumerator 及其 devices 的 *CONFIG_REMOVE* 及 *CONFIG_SHUTDOWN* 訊息之前被送出。

CONFIG_READY 機能 當 *aboutnode* 參數所指向的 DEVNODE 初次被完整設定

但尚未設置（configured）妥當時，enumerator 會收到此一訊息。也就是說，這個訊息是在 *CONFIG_SETUP* 之後、*CONFIG_FILTER* 之前被送出。此時的 Enumerator 可以確定 logical configuration 的資料是正確的。

CONFIG_REMOVE 機能 當 *aboutnode* 參數所指向的 DEVNODE 即將從系統中被移除時，首先是 device drivers，然後是其 enumerators，會收到這個訊息。這時候的 Configuration Manager 可能已經送出 *CONFIG_TEST* 及 *CONFIG_TEST_SUCCEEDED* 訊息。子機能 *CONFIG_REMOVE_DYNAMIC* 表示 device 被動態移除，子機能 *CONFIG_REMOVE_SHUTDOWN* 則表示系統即將關機。不管哪一種情況，device 都會先收到 *CONFIG_REMOVE* 訊息，然後才由其 enumerators 收到。

CONFIG_SETUP 機能 當 enumerator 自己的 DEVNODE 第一次建立，會收到這個訊息。Enumerator 應該載入這個 device 所需的任何其他 drivers，它們可能來自 device ROM。Enumerator 也應該爲新的 device 建構識別碼（ID）及描述字串（description）。使用者可以在 Windows 95 中文版的【新增硬體精靈】對話盒中看到這些字串。在 enumerator 處理這個訊息以後而 Configuration Manager 開始設置（configuring）device 之前，enumerator 會收到 *CONFIG_READY* 訊息。

舉個例子，我爲 School Bus driver 實作一個 *CONFIG_SETUP* 機能，以便讓 Windows 95 的【新增硬體精靈】知道，連接到這個 bus 上的 device 到底是什麼種類：

```
#0001  case CONFIG_SETUP:
#0002      {                    // CONFIG_SETUP
#0003      ULONG length;          // length of class name
#0004      char class[64];        // device class
#0005      length = sizeof(class);
#0006
#0007      code = CM_Read_Registry_Value(aboutnode, NULL, "Class",
#0008            REG_SZ, class, &length, CM_REGISTRY_HARDWARE);
#0009      if (code == CR_NO_SUCH_VALUE)
#0010            {              // new device
#0011            CM_Write_Registry_Value(aboutnode, NULL, "Class",
#0012               REG_SZ, "System", 6, CM_REGISTRY_HARDWARE);
#0013            CM_Write_Registry_Value(aboutnode, NULL,
#0014               "HardwareID",
```

```
#0015               REG_SZ, "WCO1234", 6, CM_REGISTRY_HARDWARE);
#0016           }              // new device
#0017     return CR_SUCCESS;
#0018     }                    // CONFIG_SETUP
```

呼叫 *CM_Read_Registry_Value* 可以讀取 device 的 hardware key 裡頭的 named value "Class" 的內容。這個 key 總會存在，除非 device 處於「正在加入系統」階段，這種情況下的傳回值應該是 CR_NO_SUCH_VALUE。爲了涵蓋那種可能性，我們使用 *CM_Write_Registry_Value* 來記錄 named value "Class" 及 "HardwareID" 的內容。從【新增硬體精靈】可以看到這些內容，藉以知道使用那一個 .INF 檔來安裝連接於此 bus 的 telepathic device。

CONFIG_SHUTDOWN 機能 Enumerator 將在其 device 收到此訊息之後收到這個訊息。此訊息表示系統準備關機。屬於 boot configuration 的那些 devices（也就是說，在 Windows 95 啓動前的真實模式中就已存在者）會收到 *CONFIG_SHUTDOWN* 訊息而不是 *CONFIG_REMOVE* 訊息。

CONFIG_START 機能 Enumerator 會收到這個訊息，並由參數 *aboutnode* 指出某一個 DEVNODE，表示該 device 已經被設置（configured），現在應該開始使用其被指定的 configuration 了。Enumerator 會先收到這個訊息，然後 device 才收到。如有必要，子機能（不論是 *CONFIG_START_FIRST_START* 或 *CONFIG_START_DYNAMIC_START*）可讓你區別究竟是 device 在其 DEVNODE 建立之後的第一次 start，或是後續的 starts。

CONFIG_STOP 機能 Enumerator 會收到這個訊息，並由參數 *aboutnode* 指出某一個 DEVNODE，表示該 device 應該停止使用其被指定的 configuration（組態）。Enumerator 會先收到這個訊息，然後才是 device 收到。

CONFIG_TEST 機能 Configuration Manager 利用 *CONFIG_TEST* 訊息來詢問是否 enumerator 容許 device 移除（removal，子訊息爲 *CONFIG_TEST_CAN_REMOVE*）或關機（shutdown，子訊息爲 *CONFIG_TEST_CAN_STOP*）。不管那一種情形，該 device 都

是以參數 *aboutnode* 所指之 DEVNODE 做爲代表。Device 將在 enumerator 傳回 CR_SUCCESS 之後才收到這個訊息。此外，enumerator 也可以「代表自己的那個 DEVNODE」來接收此訊息，然後傳回 CR_SUCCESS，認可這項申請，或是傳回 CR_FAILURE，否定這項申請。

CONFIG_TEST_FAILED 機能 當某些 driver 或 enumerator 面對 CONFIG_TEST request 時回覆以失敗，即會送出這個訊息。

CONFIG_TEST_SUCCEEDED 機能 當每一個 driver 及 enumerator 的 CONFIG_TEST request 被認可後，即會送出此一訊息。

詳細談 Device Loaders

前面所談到 Plug and Play driver 的例子有一點不合規則，即 device driver 本身也是 device loader。Bus drivers 以及獨立的 device drivers 必須以這種方法加以組織。例如我的 Enum\Root*PNP0A03（PCI bus）key 的 named value "Device"，指到一個 registry key，其 named value "DevLoader" 爲 "PCI.VXD"，那既是個 bus driver 也是個 enumerator。同樣地，serial mouse（Enum\Root*PNP0F0C）的 named value "Device" 爲 "Mouse\0000"，其 named value "DevLoader" 爲 "*vmouse"。"*vmouse" 表示內建於 VMM32.VXD 中的 VMOUSE.VXD，它主要是個 driver 而不是一個 loader。

你所寫的 device drivers，能夠良好地符合新式的 Windows 95 layered device 架構中的一種。對於這些 drivers，通常你會指定一個 Microsoft system component（例如 VCOMM 或 IOS）做爲你的 device loader。那個 component 期待能夠在 named value "PortDriver" 處發現 driver 的名稱。例如我的 COM1 key（Enum\Root*PNP0500\0000）內含的 named value "Driver" 爲 "ports\0000"。圖 **11-11** 指出 driver 的 registry key。注意其中的 named value "DevLoader" 爲 "*vcomm"，意謂著 VCOMM.VXD driver 內建於 VMM32.VXD 之中；而其 named value "PortDriver" 內容爲 "serial.vxd"。

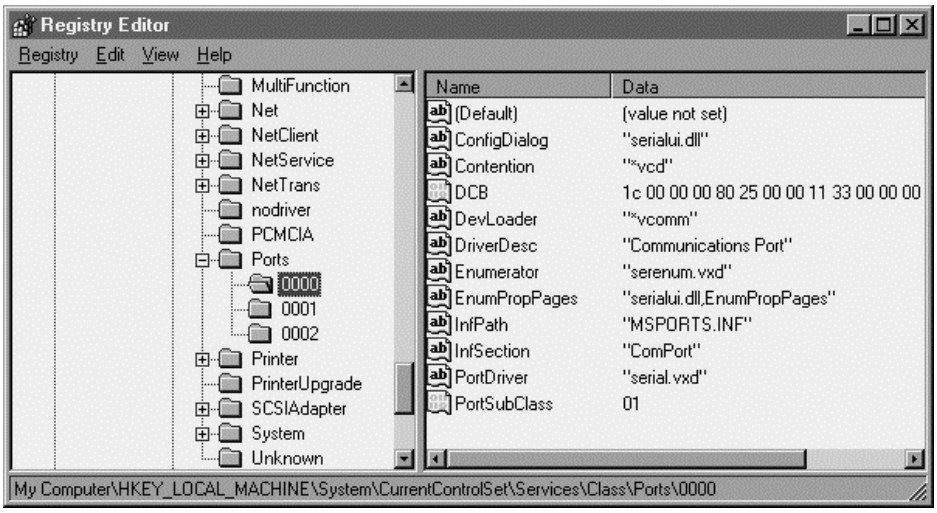


圖 11-11　COM1 的 software key

形成 COM1 謎團的部分資料，係以下列方式相互配合：Root enumerator 在進行 legacy（傳統）devices 的 enumeration 時，遇到 *PNP0500\0000 key，因而建立 COM1 的 DEVNODE。Configuration Manager 依照 "Driver" 指標，指向 HKLM\System\CurrentControlSet\Services\Class 的 "ports\0000" subkey。從那裡，它得知 "*vcomm" 是這個 DEVNODE 的驅動程式載入器（device loader），於是送給 VCOMM 一個伴有 DLVXD_LOAD_ DEVLOADER 代碼的 *PNP_New_Devnode* 訊息（控制流程並不全然如此簡單，但我所描述的是淨結果）。VCOMM 呼叫 *CM_Read_Registry_Value*，從 software key 取得 "PortDriver" 的值，亦即 "serial.vxd"。當 COM1 第一次被使用，VCOMM 直接呼叫 *VXDLDR_LoadDevice*，載入 SERIAL.VXD。爲了回應 *Sys_Dynamic_Device_Init* 訊息，SERIAL.VXD 呼叫 *_VCOMM_Register_Port_Driver*，將 VCOMM 註冊爲其 device control procedure。於是後續對 port driver 的呼叫就流往 VCOMM。就 Configuration Manager 而言，VCOMM 是 COM port 的 device driver，因而使 port driver 卸除「必須處理 configuration events」的責任。

在其它場合，你可能會希望使用各自的 device loader 來載入 device drivers。爲了回應

PNP_New_Devnode 訊息，你應該呼叫 Configuration Manager 的動態載入函式：

```
code = CM_Load_DLVxDs(devnode, filenames, type, 0);
```

這個函式會傳回 CONFIGRET 代碼，一如我們曾經討論過的其他 Configuration Manager 函式一樣。參數 *devnode* 表示你準備載入的 driver 的 DEVNODE，參數 *filenames* 係以空白分隔或以逗點分隔的一些檔案名稱（包含延伸檔名），參數 *type* 是你通常會在 *PNP_New_Devnode* 訊息處理的一種 DLVXD_LOAD_xxxx 代碼（表 **11-1**）。當然，不一定要指定一個以上的檔案，單一檔名（以 null 爲結束字元）也可以。

對於一個同樣是 Plug and Play device loader 的 static VxDs 而言，可能會有一種怪異的狀況發生。VMOUSE.VXD 就是一個例子。VMOUSE.VXD 被靜態載入，因爲它需要在處理 *Sys_Critical_Init* 訊息的同時，攔截（hook）及虛擬化（virtualize）INT 33h。直到收到 *Device_Init* 訊息之後，VMOUSE.VXD 才知道設置 mouse 的所有必要資訊。然而 Configuration Manager 係在 VMOUSE.VXD 以前初使化。因此，在 VMOUSE.VXD 準備成爲 device loader 以前，會先收到 *PNP_New_Devnode* 訊息。爲了處理這種狀況，VMOUSE.VXD 一開始會傳回 CR_DEVLOADER_NOT_READY，使 Configuration Manager 延緩處理 mouse。當 VMOUSE.VXD 收到 *Device_Init* 訊息，它就呼叫 *CM_Register_DevLoader*，表示自己已準備成爲 device loader。於是在下次機會中，Configuration Manager 重新發出 *PNP_New_Devnode*，將 mouse driver 載入。

Device Information Files（裝置資訊檔）

如果你曾經爲 Windows 應用程式建立過安裝程式，你應該很高興知道，你不需要爲你的 device driver 建立自己的安裝程式。Microsoft 已經在 Windows 95 的【新增硬體精靈】裡爲你完成所有工作。你需要做的只是提供一個 **device information file**（裝置資訊檔，一個 .INF 檔），用來描述你的 device 和隨附軟體。

DDK 中包含有廣泛的資訊，告訴我們如何使用 DDK 所附的 INFEDIT 工具程式來建立一個 .INF 檔。在此我將概括說明有關 .INF 檔的基礎觀念。

即使你已經會運用 INFEDIT 來建立和維護你的 .INF 檔，明瞭原始 (raw) 檔案本身的基本結構及語法，也會使你有所收獲。一個 .INF 檔大體上看來像是傳統的 Windows .INI 檔。它包含了一些方括號括起來的節區，每一個節區包括一些 named values （本章最後有一個範例）。.INF 語法的設計要點，目的是使它容易定位出（localize）使用者可以看到的字串（出現在訊息盒和對話盒中）。所以無論你在那裡看到像 *%String0%* 之類以百分比符號分隔的關鍵字，只要檢查檔案中的 [Strings] section 就可以找到對應的替代文字。因此，如果要指出 WCO.INF 檔案（本章最後會顯示）的建立者，你可以在 [Version] section 中尋找 "Provider" 的 value。"Provider" value 會帶引你到 [Strings] section 的 "String0" value，在那裡你得知是由 Walter Oney Software 提供這個檔案。以下是此例的相關部份：

```
[Version]
Signature=$CHICAGO$
Class=System
Provider=%String0%

...

[Strings]
String0="Walter Oney Software"
String1="Walter Oney Software"
String2="School Bus"
```

一個 .INF 檔案實際上表現的是一個與指標交織在一起的階層化資料結構。[Manufacturer] section 是主要的結構（見圖 **11-12**）。這個 section 列出在一組安裝磁片中出現的軟體廠商（.INF 檔出現在這組磁片的第一片或唯一一片之中）。這些廠商名稱也是使用者在【新增硬體精靈】裡，指定一種 devices 或是按下【從磁片安裝（Have Disk）】按鍵並指定一台磁碟機後，所看到的第一個畫面的名稱來源（圖 **11-13**）。

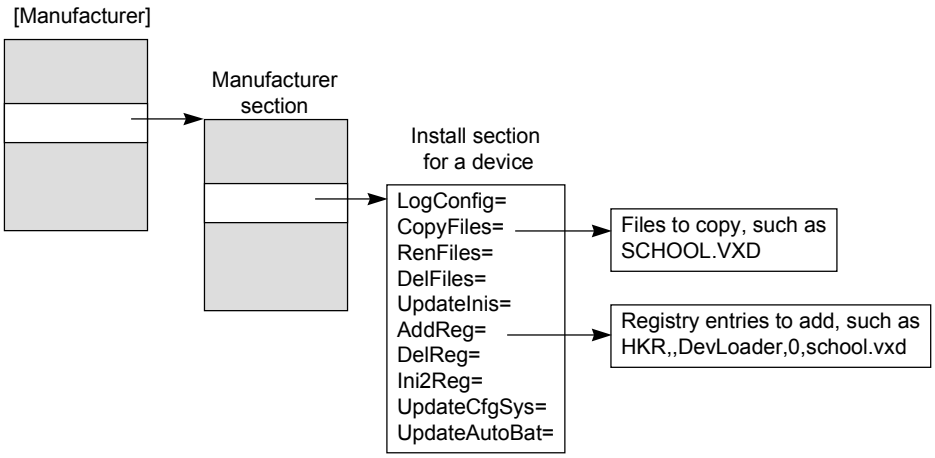


圖 11-12 .INF 檔案的階層化結構

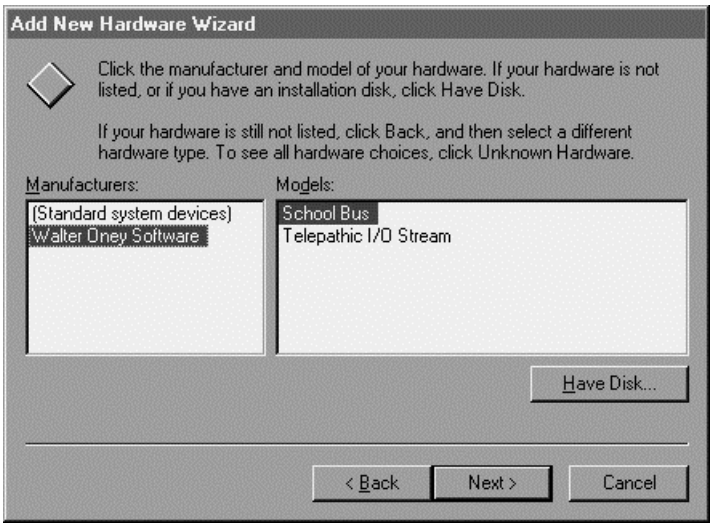


圖 11-13 製造商列表畫面

[Manufacturer] section 內的 values，其格式是 *company-name=section-name*。例如，*Walter Oney Software=SECTION_0* 這一行表示你可以在 [SECTION_0] 發現有關 Walter Oney Software device 的資料。每個 manufacturer section 以一行一行的 *device line* 列出廠商提

供的 devices，每一行把 device 的 install section 名稱及 hardware ID 結合在一起。本章最後提供的 WCO.INF 檔案，你會看到 Walter Oney Software 提供了一個 School Bus，在那裡面有一個名爲 "bus.install" 的 install section 以及一個內容爲 *PNP0A05 的 hardware ID。每一個 install section 內含進一步的指標，指向其他 sections，以便詳細說明如何安裝特定的 device。以下是本例的相關部份：

```
...
[Manufacturer]
%String1%=SECTION_0

[SETCTION_0]
%String2%=bus.install,*PNP0A05
...
```

Install Sections

一個 device 的 install section 可以在安裝（或更新）軟體，以及指定組態資訊上，給你相當多的彈性。在 install section 裡的每筆記錄都是一個指標，指到另一個 section，其中包含真正的安裝指示。大部分時候你肯定需要 AddReg 以及 CopyFiles 這兩個項目，其它項目有更多深奧的目的。一個 install section 的語法看起來如下，其中的 install-name 是你在某些 manufacturer sections 的 device line 中所使用的名稱：

```
[install-name]
AddReg=AddReg-name
CopyFiles=CopyFiles-name
...
```

你可以指定的「subsection 指標」有以下數種：

AddReg AddReg section 用以指示在安裝這個 device 時，如何更新 registry。這個 section 的項目有以下格式：

```
root-key, subkey, name, flag, value
```

參數 *root-key* 以及 *subkey* 用以說明你希望加入或修改的 registry key。它們最常見的內

容分別是 HKR 以及空白字串，表示你希望在 device 的 software key （在 \HKLM\System\CurrentControlSet\Services\Class 之下）新增或修改一筆 entry。參數 *name* 以及 *value* 表示 subkey 之下的 named value。參數 *flag* 表示 subkey 的資料種類及替換屬性（replacement option）：位元 #0 如果是 0，表示 value 是一個 ANSI 字串，如果是 1，表示 value 是 16 進位。位元 #1 如果是 0，表示你希望替換目前的 registry key，如果是 1，表示你不希望替換它。不必爲了保持這些位元的正確性而操心，稍後我將介紹的 INFEDIT 工具程式，會爲你保持它們的正確性。

本例之中，安裝一個 School Bus 會引起安裝程式處理 [bus.registry] section，其中包含：

```
HKR,,DevLoader,0,school.vxd
```

此行會插入一個 "DevLoader school.vxd" 項目到 bus 的 software key 中。如前所描述，就是這一行造成 Configuration Manager 載入 School Bus enumerator 的 registry key。

CopyFiles CopyFiles section 用以說明你希望從安裝磁片複製到硬碟中的所有檔案。這個 section 的每一行都有這樣的格式：

```
destination-file, source-file, temporary-file, flag
（目的檔名，來源檔名，暫存檔名，旗標值）
```

除了參數 *destination-file* 以外，每一個參數都可有可無。參數 *destination-file* 表示複製後的檔案名稱及副檔名。參數 *source-file* 指出安裝磁片中的檔案名稱；如果 *source* 及 *destination* 名稱相同，可省略 *destination* 名稱。如果檔案可能在安裝期間被開啓，則使用 *temporary-file* 所指定的檔名。Windows 95 就是以這個名稱來安裝檔案，並在下一次系統啓動時更改名稱。如果這個檔案的安裝成功與否，對整個安裝過程至爲重要，你可以指定 flag 值爲 2。

在 WCO.INF 之中，School Bus 的 CopyFiles section 被指名爲 [bus.files]，並指定從安裝磁片中複製 SCHOOL.VXD 到硬碟裡頭。

現在你可能會想知道兩件事：【新增硬體精靈】如何找出檔案在安裝磁片中的位置，它又如何知道將檔案放在硬碟的什麼目錄下？有一些 .INF section 可以用來描述安裝磁

片，與我們曾討論過的 [Manufacturer] 分支完全無關。[SourceDiskNames] section 用來列舉安裝磁片的組成；這個 section 包括磁片的 labels 以及記錄於磁片中的磁片名稱，所以安裝者可以確認是否插入正確的磁片。[SourceDiskFiles] section 用來列出安裝套件中的所有檔案，以及在哪張磁片中可以找到哪些檔案。以 WCO.INF 爲例，SCHOOL.VXD 存放在標籤爲 School Bus Installation Disk 的 DISK1 內。

爲了決定檔案的目的地（磁碟目錄），【新增硬體精靈】會參考 [DestinationDirs] section。如果沒有其它資料，"DefaultDestDir" 項目表示預設的目錄。以 WCO.INF 爲例，這個項目的值爲 11，那是 SYSTEM 目錄的所謂助憶碼（mnemonic code）-- 雖然其實並不怎麼有助於記憶。**表 11-5** 列出這個參數的各種可能數值。如果使用 INFEDIT，你不需要參考這個表，因爲 INFEDIT 會提供一個你可以選擇的目錄挑選表。

Destination Code	說明
1	檔案來源的磁碟機及路徑名稱
2	暫時的 setup 目錄（只在 setup 時有效）
3	移除（Uninstall）目錄（例如 \UNINSTAL.000）
4	備份（Backup）目錄
10	Windows 目錄（例如 \WINDOWS）
11	System 目錄（例如 \WINDOWS\SYSTEM）
12	I/O subsystem 目錄（例如 \WINDOWS\SYSTEM\IOSUBSYS）
13	Command 目錄（例如 \WINDOWS\COMMAND）
14	Control Panel（控制台）目錄
15	印表機目錄
16	工作群組（Workgroup）目錄
17	.INF 目錄（例如 \WINDOWS\INF）
18	Help 目錄（例如 \WINDOWS\HELP）
19	Administration 目錄（例如 \WINDOWS）
20	字型（Font）目錄（例如 \WINDOWS\FONTS）

Destination Code	說明
21	Viewers 目錄（例如 \WINDOWS\SYSTEM\VIEWERS）
22	VMM32 目錄（例如 \WINDOWS\SYSTEM\VMM32）
23	Color 目錄（例如 \WINDOWS\SYSTEM\COLOR）
25	共享（Shared）目錄（例如 \WINDOWS）
26	Windows 開機（boot）目錄（例如 \WINDOWS）
27	由機器指定（Machine specific）
28	主機（Host）開機（boot）目錄（例如 \WINDOWS）
30	開機磁碟機（boot drive）的根目錄
31	虛擬開機磁碟機（virtual boot drive）的 host drive 根目錄
32	舊的根目錄
33	舊的 Windows 目錄
34	舊的 DOS 目錄

表 11-5 Destination Directory 代碼

你也可以指定這種格式的項目：

```
section-name=code[,subdir]
```

在這筆項目中，*section-name* 是一個 CopyFiles section 的名稱，code 是表 **11-5** 的 destination codes 之中的一個，而 *subdir* 是在 code 所代表的目錄底下，一個可有可無的子目錄。這個 section 名稱也可以是 RenFiles section 或 DelFiles section 的名稱。

注意，在某個 CopyFiles section 中的所有檔案，最後都存放在硬體的同一個目錄中。如果你需要把檔案複製到不同目錄，必須爲 install section 增加數個 CopyFiles 項目。

DelFiles 你可以依據 DelFiles section 內的記錄，刪除一個以上的事先存於硬碟的檔案。每一筆項目有著這樣的格式：

```
filename[,,,flag]
```

這裡的 *filename* 是目的檔（target file）的名稱及副檔名，*flag* 是一個可有可無的旗標。

如果 *flag* 設立，Windows 95 暫時不刪除正在使用的檔案，直到下次系統重新啓動再刪除之。檔案名稱的目錄部分來自於 [DestinationDirs] section。安裝程式在安裝期間會刪除你在 DelFiles section 指定的檔案。當使用者移除 device 時，Windows 95 並不會自動刪除檔案。

DelReg 你可以依據這個 section 的記錄，刪除一個以上的 values 或是全部的 key。每一筆項目有這樣的格式：

```
root-key[,subkey][,name]
```

其中的參數與 AddReg section 的一樣，有著相同的意義。就像 DelFiles section 中的項目一樣，這些項目與 device 的安裝有關，不與移除有關。

Ini2Reg Ini2Reg section 允許你把舊式的 .INI 檔案轉換至對應的 registry 項目。Ini2Reg section 裡的項目有這樣的格式：

```
%dest%\filename,ini-section,[ini-key],root-key,[subkey][,flag]
```

其中 *dest* 是一個 destination directory 代碼（表 **11-5**），*filename* 是 .INI 檔名（包括副檔名），*ini-section* 是 .INI 檔案中以方括號框住的 section 名稱，可有可無的 *ini-key* 則用以指出 .INI 檔案裡頭等號左邊的關鍵字，*root-key* 是一個 registry root key，可有可無的 *subkey* 用以指出一個 registry subkey，可有可無的 *flags* 代表旗標。如果 *flags* 的位元 #0 爲 0，表示你不希望刪除 .INI 的 entry（或 section），如果爲 1，表示希望刪除 entry（或 section）。如果 *flags* 的位元 #1 爲 0，表示你不希望替換已存在的 registry subkey，如果爲 1，表示你希望替換它。INFEDIT 會爲你正確設定這些旗標。

LogConfig LogConfig section 用以表示邏輯組態（logical configuration）的內容。如果 device driver 或其 enumerator 都沒有任何方法知道 device 的資源需求，你就必須指定這份資料。由於只有 Plug and Play ISA 介面卡能夠提供所需資源的資料，而且你也許會懷疑介面卡廠商是否完全遵守標準，因此你大概總是必須在 .INF 檔案裡提供邏輯組態。

這個 section 可以內含一個 ConfigPriority 項目，以及一個或多個用以指定資源需求的項目。如果 device 所需的資源不只一個實體，你可以重複資源關鍵字。可能的項目有：

```
ConfigPriority={HARDWIRED | DESIRED | NORMAL | SUBOPTIMAL |
                DISABLED | RESTART | REBOOT | POWEROFF | HARDRECONFIG}
DMAConfig=[{D | W}:]channel-number[,channel-number]...
IOConfig=io-range[,...]
IRQConfig=[S:]irq-number[,irq-number]...
MemConfig=mem-range[,...]
```

DMAConfig 裡頭的 D 和 W 分別表示大小爲 DWORD（32 位元）及 WORD（16 位元）；如果沒有指定，預設爲 8 位元。IRQConfig 裡頭的 S 表示此 IRQ 和其它 device 共享，意思是說 driver 會將其虛擬化爲 shared IRQ，並搞清楚每一個中斷（interrupt）的來源。

io-range 使用以下兩種格式中的一種：

```
start-end[([device-mask]:[alias-offset]:)]
size@min-max[%align-mask][([decode-mask]:[alias-offset]:)]
```

其中符號 "-" 用以分隔 *start* 和 *end* 以及 *min* 和 *max*，是語法的一部份。下面的例子表示 device 需要 8 個連續的 ports，從 1F8h 或 2F8h 或 3F8h 開始：

```
IOConfig=1F8-1FF(3FF::), 2F8-2FF(3FF:), 3F8-3FF(3FF::)
```

下一個例子表示 device 需要 8 個連續的 ports，從 300, 308, 310, 318, 320 或 328 的 8-byte 邊界開始：

```
IOConfig=8@300-328%FF8(3FF::)
```

參數 *mem-range* 以類似的語法表示記憶體範圍，只不過你得使用整個 32 位元來說明 alignment mask：

```
[size@]min-max[%align-mask]
```

下面的例子表示 device 使用 C000:0000h 至 C000:7FFFh 記憶體：

```
MemConfig=C000-C7FFF
```

下一個例子表示的是更有彈性的 device，它需要以 C000:0000h，C800:0000h，D000:0000h，或 D800:0000h 開始的 8000-byte aligned block：

```
MemConfig=8000@C000-D8000%FFFF8000
```

如果你使用 .INF 檔案的 LogConfig section 來建立邏輯組態（logical configuration），它們會以 hardware registry key 的 LogConfig entries 的方式出現。Root enumerator 會讀取這些 registry entries 來取得 BASIC_LOG_CONF 數值。MCA enumerator 也會做同樣的動作。

RenFiles RenFiles section 讓你在安裝過程中，以下面的語法更改檔案名稱：

```
new-name,old-name
```

你可以在 [DestinationDirs] section 裡指定一個目錄，準備放置將被改名的檔案。

UpdateAutoBat UpdateAutoBat section 讓你在 AUTOEXEC.BAT 裡增加或修改 registry entries。通常你並不需要修改 AUTOEXEC.BAT，因爲你所做的每一件事，都應該能夠由 Windows 95 自我完成。如果你真的要做，可以使用以下方式：

```
CmdAdd=command-name[,"parameters"]
CmdDelete=command-name[,flag]
PrefixPath=destination-code[,...]
RemOldPath=destination-code[,...]
TmpDir=destination-code[,subdirectory]
UnSet=environment-variable-name
```

所謂刪除一個 command，意思是刪除內含特定 command 的任何一行；如果加上可有可無的 flag 值 1，就會使該行被變成註解，而不是被刪除。由於你通常並不希望刪除所有的 SET command，所以你可以使用 *UnSet* 來刪除環境設定值。

RemOldPath 用來表示應該被移除的 directories path，*PrefixPath* 用來指出在【新增硬體精靈】中應該增加的 directories path。例如，SETUP.INF（幫助驅動 Windows 95 Setup 程式的一個 .INF 檔案）就內含以下兩行 path 維護指令：

```
PrefixPath=26,25,10,13
RemOldPath=33,32
```

這些指令移除了原來的 Windows 及 boot 磁碟目錄，加上新的 Windows boot、Windows shared、Windows、以及 Command 磁碟目錄。這些磁碟目錄有一部份是相同的。在我的系統中，這些指令的最後結果是把目錄：

```
path C:\DOS;C:\TOOLS;C:\WINDOWS
```

更改爲：

```
path C:\WINDOWS;C:\WINDOWS\COMMAND;C:\TOOLS
```

UpdateCfgSys UpdateCfgSys section 讓你在 CONFIG.SYS 檔案內增加或修改 entries。如同 AUTOEXEC.BAT 一樣，你通常不需要在 Windows 95 裡改變 CONFIG.SYS。但是如果你想改變，你可以使用以下方式：

```
Buffers=number
DelKey=key
DevAddDev=driver-name,{device|install}[,flag][,"parameter"]
DevDelete=driver-name
DevRename=old-name,new-name
Files=number
PrefixPath=destination-code[,destination-code]
RemKey=key
Stacks=number
```

UpdateIniFields UpdateIniFields section 允許你選擇 .INI 檔案裡頭那些「屬於不只一個元件（component）」的欄位做修改。每一筆 entry 有以下格式：

```
%dest%\filename,ini-section,key,[old-field],[new-field][,flags]
```

如果 flags 的位元 #0 數值爲 0，表示你希望逐字處理 * 字元，如果其值爲 1，表示你希望 * 是一個萬用(wildcard)字元。當你加入一個新欄位時，如果位元 #1 的數值爲 0，表示你希望使用空白做爲分隔，如果數值爲 1，表示你希望使用逗點做爲分隔。

例如，WINPAD.INF 有以下的 UpdateIniFields section：

```
[WinPadFields]
system.ini,boot,drivers,hhsystem.dll
```

這筆記錄（entry）的意思是，在 SYSTEM.INI 裡頭屬於 [boot] section 中的 drivers= 設定行中，若欄位爲 "hhsystem.dll" 則應以空白來取代（換句話說，它應該被移除）。

又例如，如果你懶於更改【啓動（StartUp）】程式集（program group），下面這一行會在 WIN.INI 檔案裡頭的 [Windows] section 中加上一筆 load= 的設定行：

```
win.ini,Windows,load,,myapp.exe,2
```

UpdateInis 你應該在 .INI 檔案裡頭使用 UpdateInis section 來增加或修改整筆記錄（entries）。一般而言，你應該使用 registry 來儲存在 Windows 95 中的參數值。但是，如果你需要使用舊式的 .INI 檔案，可以這麼做：

```
%dest%\filename,ini-section,[old-entry],[new-entry][,flags]
```

參數 *flags* 讓你有相當大的彈性來處理目前 .INI 檔案的 entries。Windows 95 DDK 的 "Update INI File Sections" 對於這個參數有更詳細的說明。其預設值爲 0，適合大部份的需求，表示【新增硬體精靈】將以 new-entry 取代 old-entry。如果省略 old-entry 參數，會導至增加一個 new-entry。而且，只有 old-entry value 的關鍵字部份被考慮。因此，下面這一行：

```
win.ini,Desktop,Wallpaper=0,Wallpaper=1
```

將會改變 WIN.INI 檔案裡頭屬於 [Desktop] section 的 Wallpaper= 設定值，從 0 改變爲 1。

使用工具軟體 INFEDIT

要持續關心所有 .INF 項目的語法，各 sections 之間的連繫，以及每一階段的選擇，是非常困難的。幸好，Microsoft 在 DDK 裡頭提供了一個 .INF 編輯器：INFEDIT.EXE。INFEDIT 有一個圖形環境，在那裡你可以建立並編輯 .INF 檔案，INFEDIT 會自動檢查語法的正確性。

我將引導你經由建立一個簡單的 .INF 檔案來學習如何使用 INFEDIT。步驟如下：

1. 完成 [Version] section 與檔頭資料（file header information）。

2. 藉由加上一個新的 manufacturer 以及一個新的 device，來完成 [Manufacturer] section。
3. 建立 CopyFiles section 以描述哪些檔案應該複製到使用者硬碟之中。
4. 更新 Disk Names section 以表示所有的「安裝檔案」應該如何安排在「安裝磁片」上。
5. 建立一個 Add Registry section。
6. 完成 device 的 Install section。

你必須從 Windows 95 DDK 的 \BIN 目錄中執行 INFEDIT.EXE，以啓動 INFEDIT。請選按【File/New】命令，然後選擇最上層的 <NEWINF.INF> 資料夾（**11-14**）。

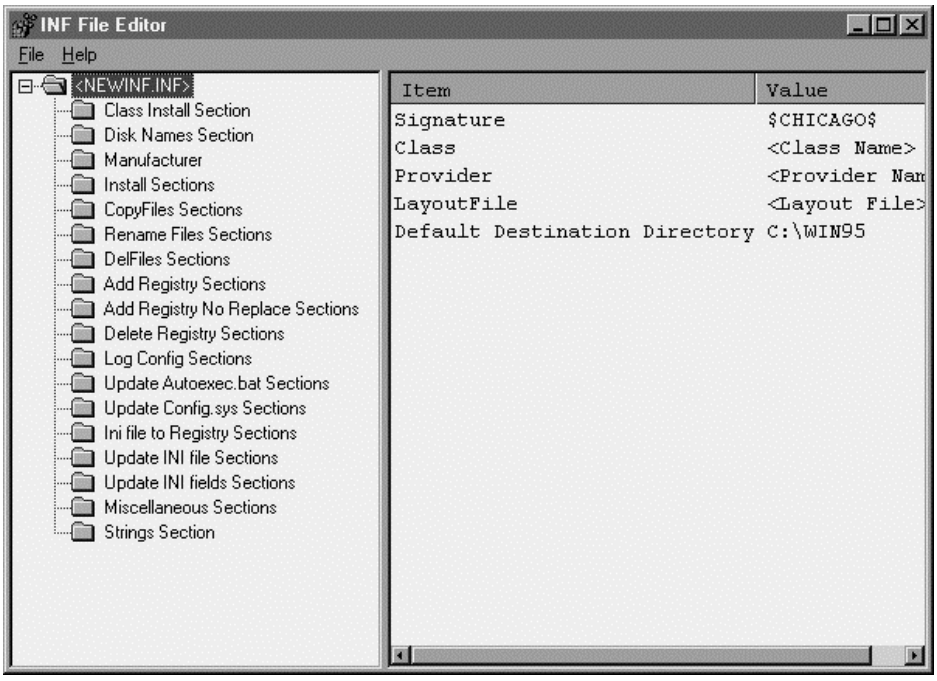


圖 11-14 INFEDIT 的初始畫面

你應該在【Class】項目中填寫 .INF 檔選定的 device class 名稱（本例爲 *System*），並在【Provider】項目中填寫你的公司名稱（因爲是你在建立這個 .INF 檔案）。如果你有一組複雜的安裝磁片，你可能會希望在這裡指出一個磁片配置檔（layout file）。本例並沒有配置檔，所以你應該將【LayoutFile】項目設爲空白。

爲了改變上述各項目的值，請在 INFEDIT 視窗的右邊對各個項目（Item）名稱做滑鼠雙擊動作。INFEDIT 會顯示一個對話盒，讓你指定新值（圖 **11-15**）。

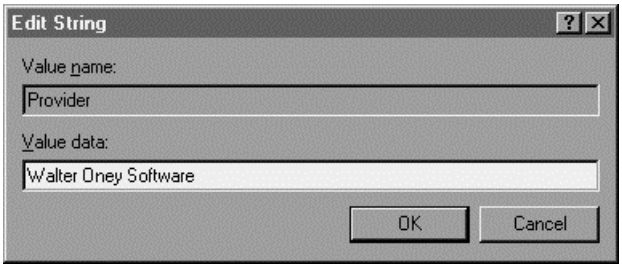


圖 11-15　數值編輯對話盒

INFEDIT 會給你類似的對話盒，讓你指定所有數值。有一些項目會出現更複雜的對話盒，其中包括許多欄位及選單，如圖 **11-16**。

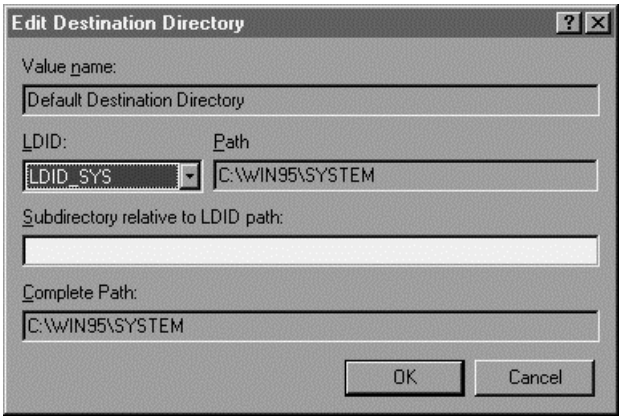


圖 11-16　編輯 default destination directory

現在，用滑鼠右鍵按一下 INFEDIT 左視窗的【Manufacturer】資料夾，你會獲得一個捷徑功能表（shortcut menu）。選擇其中的【New Manufacturer】，並在視窗的 tree 中下移一層，也就是選擇【New Manufacturer】資料夾，把【Company Name】項目更改爲自己公司的名稱，如圖 **11-17** 所示。滑鼠右鍵按一下左邊的資料夾，你可以在 INFEDIT 中建立新的或刪除舊的 sections。因爲 INFEDIT 知道有哪些可能的選擇，所以捷徑功能表（shortcut menus）會在每個步驟裡適當的限制你可以使用的選項。

現在滑鼠右鍵按一下你剛才建立的【company name】資料夾，然後選擇【New Device】。選按【Device description】資料夾，並修改這個項目。在我的例子中，我將建立 School Bus 這筆項目，其內容包括一個名爲 bus.install 的 install section，以及一個值爲 *PNP0A05 的 hardware ID，並表示它不相容於任何其它 device。當你在右邊視窗指定 bus.install，INFEDIT 即以那個名稱自動建立一個空白的 install section（圖 **11-18**）。

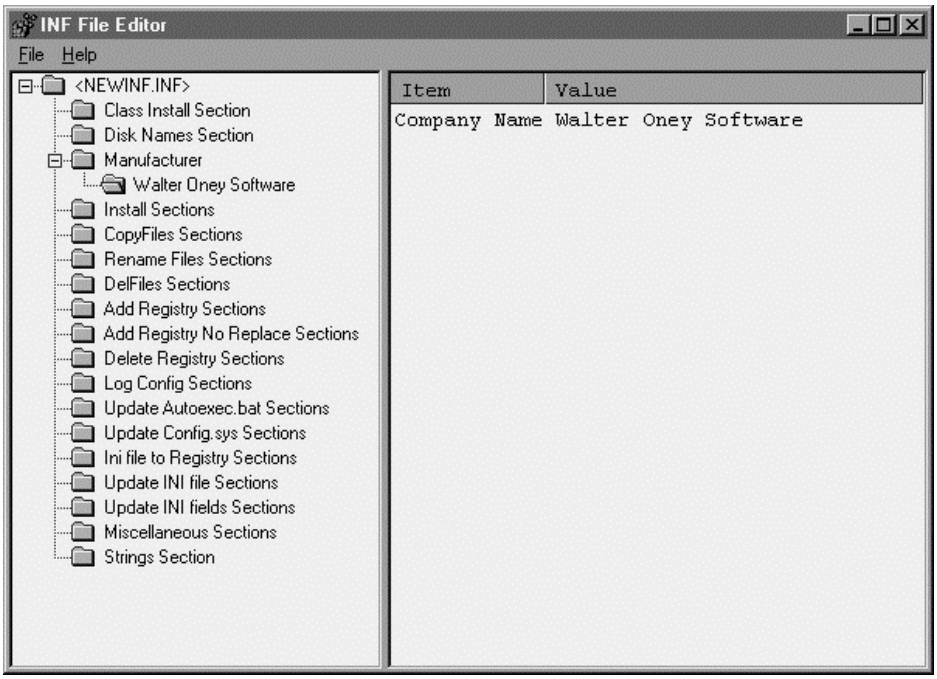


圖 11-17　新的 manufacturer section

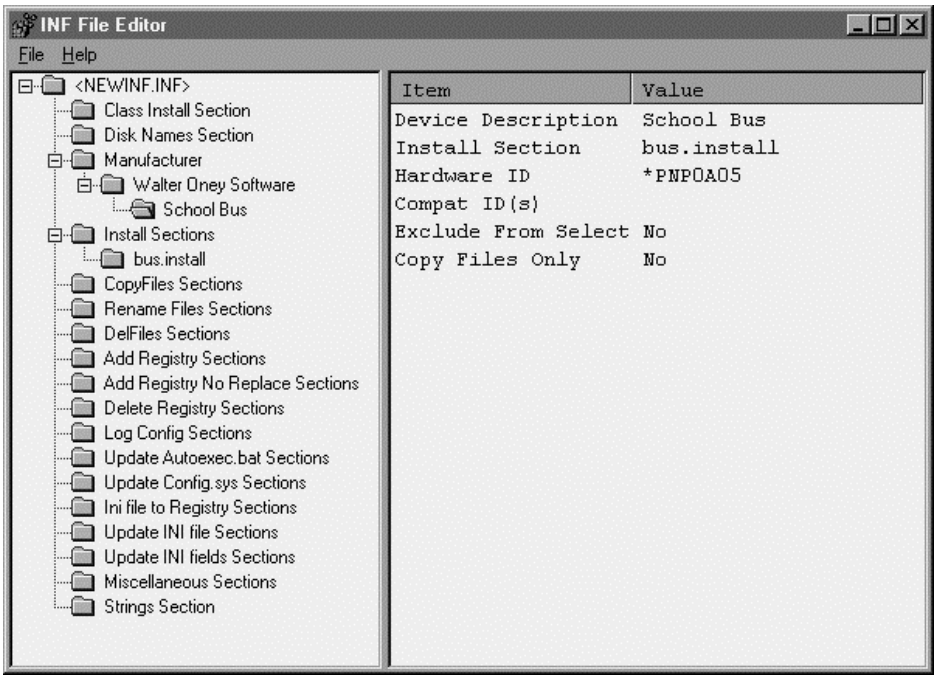


圖 11-18　完成 new-device 的所有資訊

在試著填寫 bus.install 以前，先指定安裝磁片的資料及其內容，會更方便些。爲此，滑鼠右鍵按一下【CopyFiles Sections】資料夾，然後增加一個新的 section。下移到那個資料夾，將【Section Name】項目改變爲 bus.files。然後滑鼠右鍵按一下剛才建立的【bus.files】資料夾，選擇【Add File Name】。將【File Name】項目改爲 SCHOOL.VXD 的 Destination Name。現在，移到【Disk Names section】資料夾（接近左視窗上面），在其中增加新磁片名稱。磁碟的可列印標籤（也就是【Disk Description】項目）應該是 School Bus Installation Disk，磁片標籤（也就是【Disk Label】項目）應該是 DISK1。【Disk serial number】應改成空白，代替 0000-0000（圖 **11-19**）。

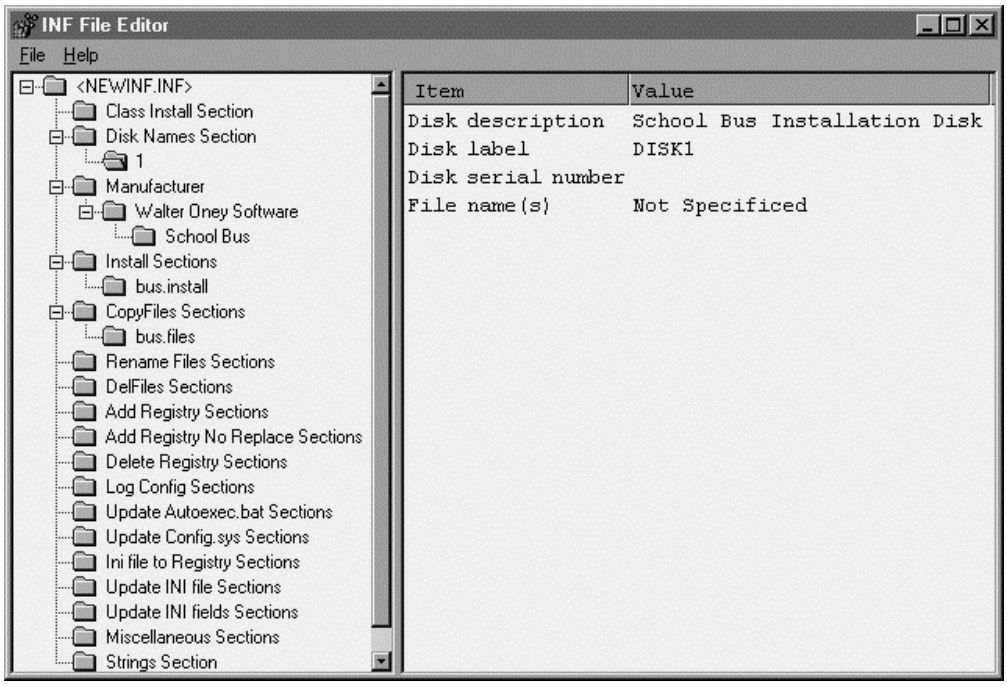


圖 11-19　增加新的安裝磁片

這時候你可以把檔案劃分到不同的安裝磁片上。當你在 Disk Names section 裡頭改變 File Name(s) 項目時，INFEDIT 從 CopyFiles sections 得知所有檔案並給你一個列表（☞**11-20**）。如果你先完成了 CopyFiles sections，就可以在這裡獲得一份列表。請在【Edit Disk File List】對話盒裡頭將 SCHOOL.VXD 加入【Include Files】列表之中，然後按【OK】鈕。

回到 install section 以前，你需要建立一個新的 Add Registry section。請把【Section Name】項目改成 bus.registry，然後建立一個新的 Section Value。在 software 分支中，你應該有一個 DevLoader 項目。雖然 INFEDIT 沒有很清楚的表示，不過在這裡使用 HKR 做爲 root key 是正確的。請設定所有 Section Value 的項目，如☞**11-21** 所示。

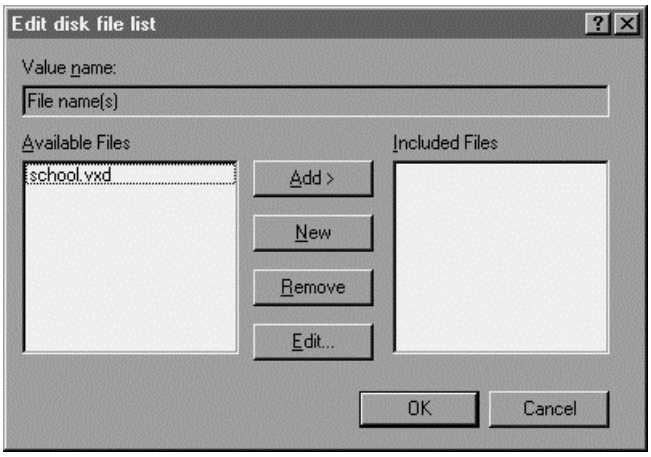


圖 11-20　檔案列表

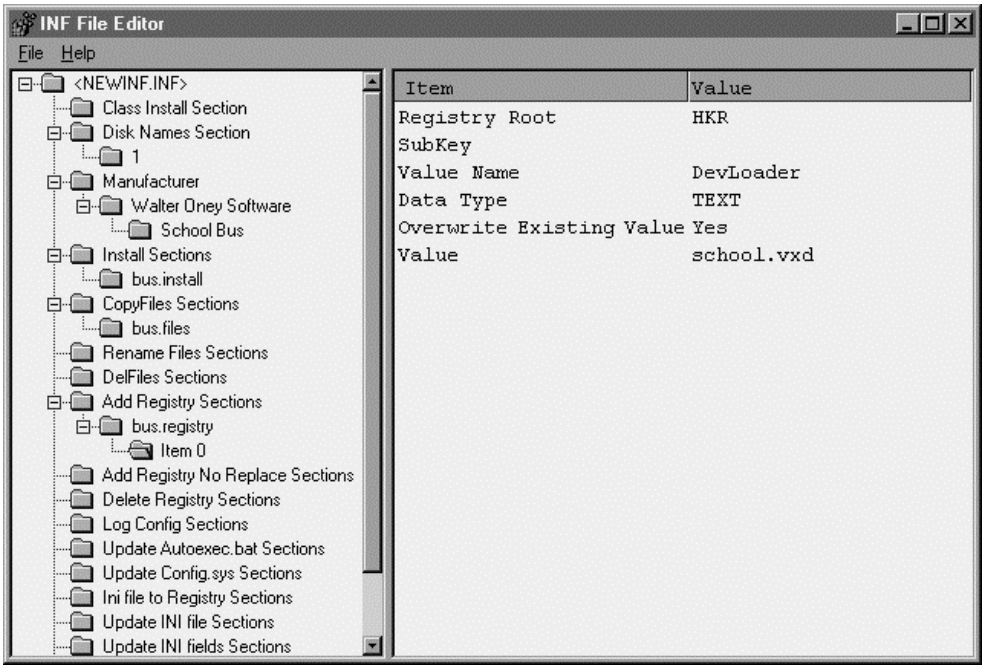


圖 11-21　增加 registry entries

現在回到 bus.install section，填寫 CopyFiles Sections 項目（增加 bus.files），以及 AddReg 項目（增加 bus.registry）。INFEDIT 提供列表給這些項目挑選，這就是爲什麼我們要先建立輔助的 sections 的原因。

然後，你可以儲存這個新的 .INF 檔案。我以 WCO.INF 名稱儲存之。完整的 .INF 檔案看起來像這樣：

```
#0001  [Version]
#0002  Signature=$CHICAGO$
#0003  Class=System
#0004  Provider=%String0%
#0005
#0006  [DestinationDirs]
#0007  DefaultDestDir=11
#0008
#0009  [Manufacturer]
#0010  %String1%=SECTION_0
#0011
#0012  [SECTION_0]
#0013  %String2%=bus.install,PNP0A05
#0014
#0015  [bus.install]
#0016  CopyFiles=bus.files
#0017  AddReg=bus.registry
#0018
#0019  [bus.files]
#0020  school.vxd
#0021
#0022  [bus.registry]
#0023  HKR,,DevLoader,0,school.vxd
#0024
#0025  [ControlFlags]
#0026
#0027  [SourceDisksNames]
#0028  1=School Bus Installation Disk,DISK1,
#0029
#0030  [SourceDisksFiles]
#0031  school.vxd=1
#0032
#0033  [Strings]
#0034  String0="Walter Oney Software"
#0035  String1="Walter Oney Software"
#0036  String2="School Bus"
```

第 12 章

Configuring Devices
（對裝置做組態規劃）

譯註：本章多次出現 configuration（名詞）或 configure（動詞）字眼，有時候我將它們譯爲「組態」或「設置」（視上下文而定）。

前一章中，我爲你說明了 Configuration Manager 如何判斷出現在電腦上的硬體，以及如何載入支援此一硬體之 drivers。這一章，我將更詳細討論 configuration 的過程。當你設置（configure）連接於系統的 devices 時，你必須先決定它們的資源需求，然後爲它們配置配置可用的資源。簡單地說，這個過程步驟如下：

1. 建立每個 device 的 logical configuration，用以說明 device 的資源需求，並允許有不同的選擇。
2. 過濾（filter）logical configuration 以進一步限制可容許的不同資源分配。
3. 以如此方法來分配資源給 device，避免衝突並確保最高效率。
4. 設置（configure）硬體本身

邏輯組態（Logical Configuration）

每一個需要資源的 device，最後都會在其 DEVNODE 上附有一個或多個 logical configuration。一個 logical configuration 表示一組可能的資源分配。每個 DEVNODE 可以擁有五種不同的 logical configuration：

- **Boot** configuration（起始組態） 用以表示 system BIOS 在 boot 時分配給 device 的資源。
- 使用者可以在「控制台（Control Panel）」指定資源分配，以建立 **forced** configuration（強制組態）。
- Enumerator 可以建立 **basic** configurations 串列（linked list），以說明可能的資源分配。
- Enumerator 及 device driver 可以複製並修改 basic configuration 串列，產生另一個 **filtered** configurations 串列 。
- Configuration Manager 及其 resource arbitrators（資源仲裁器）最後會建立一個 **allocated** configuration，描述真正指定給 device 的資源。

由於可能有不止一個的 logical configurations，放在 **basic** 及 **filtered** configuration 串列裡，Configuration Manager 需要一些方法來處理 configurations。每一個 logical configuration 擁有一個優先權（priority，表 **12-1**），其中的 *LCPRI_FORCECONFIG* 及 *LCPRI_BOOTCONFIG* 兩數值分別只在 **forced** 及 **boot** configurations 裡使用，它們唯一的目的是讓有經驗的程式員在除錯時發現 configuration 的來源。其他的 priority 數值表示 configurations 之間的真正優先權等級。

乍看之下，擁有多個 logical configuration 並各有不同的優先權，對目前的硬體來講似乎多餘。然而，現在的作業系統可以處理更複雜的硬體，因此這種彈性可能會變得更爲重要。例如，一張「畫面捕捉卡」如果可以直接定址大量記憶體，運作最爲良好。如果換成經由 DMA 傳送，雖然速度變慢，但是仍舊可以運作。這樣的 device 應該有一個「索

求記憶體」的 *LCPRI_DESIRED* configuration，以及一個「索求 DMA 通道」的 *LCPRI_NORMAL* 或 *LCPRI_SUBOPTIMAL* configuration。

優先權（priority）	說明
LCPRI_FORCECONFIG	這個 configuration 是使用者透過「控制台」強制設定的
LCPRI_BOOTCONFIG	這個 configuration 是 BIOS 或介面卡在開機時建立的
LCPRI_DESIRED	這個 configuration 比較被喜歡，並有最好的效率
LCPRI_NORMAL	這個 configuration 的效率還可以接受
LCPRI_SUBOPTIMAL	這是一個可使用的 configuration，但效率不佳
LCPRI_RESTART	這個 configuration 要求重新啓動 Windows
LCPRI_REBOOT	這個 configuration 要求電腦 soft reboot
LCPRI_POWEROFF	這個 configuration 要求電腦 hard reboot
LCPRI_HARDRECONFIG	這個 configuration 要求使用者改變 jumper
LCPRI_HARDWIRED	這個 configuration 是唯一可能的一個

表 12-1　邏輯組態（Logical configuration）的優先權

發現 device 的資源需求並建立 logical configurations，一般而言是「建立對應之 DEVNODE」的 enumerator 的責任。很多 devices 可以自動提供資源需求資料，enumerator 通常取得這些需求以便建立 logical configurations。Enumerator 使用 *CM_Add_Empty_Log_Conf* 建立每個 logical configuration，並使用 *CM_Add_Res_Des* 增加每一組資源需求（以 resource descriptor 的形式）。

爲了再一次具體說明事實，讓我們假設第 11 章的 School Bus，支援一種可以察覺使用者想法的 telepathic I/O stream device。School Bus 的 *OnEnumerate* 函式可能包含下面的 enumeration 程式碼：

```
#0001  case CONFIG_ENUMERATE:      // cf == 5
#0002     {                     // CONFIG_ENUMERATE
#0003     DEVNODE device;         // DEVNODE for device
```

```
#0004     LOG_CONF logconf;      // logical configuration
#0005     RES_DES resource;      // resource descriptor handle
#0006
#0007     static IRQ_DES irq = {{0, 0, 0xFFFF, 0}};
#0008
#0009     CM_Create_DevNode(&device, "SCHOOL\\WCO1234\\0000",
#0010        tonode, 0);
#0011     CM_Add_Empty_Log_Conf(&logconf, device,
#0012        LCPRI_NORMAL,
#0013        BASIC_LOG_CONF | PRIORITY_EQUAL_LAST);
#0014     CM_Add_Res_Des(&resource, logconf, ResType_IRQ,
#0015        &irq, sizeof(irq), 0);
#0016     return CR_SUCCESS;
#0017     }                  // CONFIG_ENUMERATE
```

CM_Add_Empty_Log_Conf 可以建立一個空白的 logical configuration。如果我們只是建立空白的 configuration，device 並不會要求資源。*CM_Add_Res_Des* 可以在 logical configuration 身上增加一筆資源需求。此例之中，資料結構 *irq* 裡的第三個欄位表示 device 需要一個範圍 0~15 的 IRQ。資料結構 IRQ_DES 將在下一節 "Resource Descriptors" 中說明。

在 device 用以登錄資料的 LogConfig subkey 裡頭，其 registry 記錄著傳統（老舊的）devices 的 logical configurations，見圖 **12-1**。Root Enumerator 使用 *CM_Read_Registry_Log_Confs* 並根據 registry entries，自動建立起存放於記憶體中（而非 registry 中）的 logical configurations。

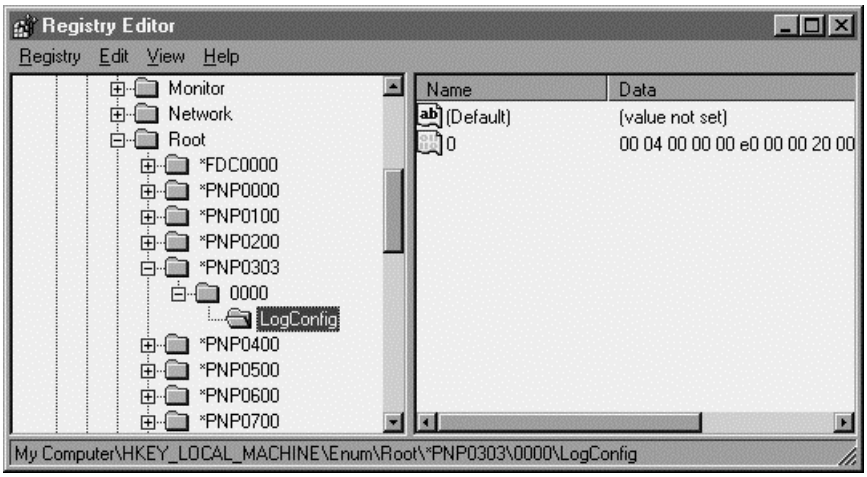


圖 12-1　Registry 中的 logical configuration（邏輯組態）實例

一個 logical configuration 允許每種標準資源（如 IRQ、記憶體、I/O 位址、DMA）可以有一些選擇。事實上，*Hardware Design Guide for Microsoft Windows 95*（Microsoft Press 出版，1994）中要求 Plug and Play 擴充卡及主機板盡可能提供多種不同的 configurations，以確保最大彈性。以下是能夠符合 "PC 95" 需求規格的建議：

- 至少 8 個可選擇的 IRQ，每一個都可以 disabled。
- 至少 8 個 memory base addresses（記憶體基底位址），每一個都可以 disabled，用於直接記憶體存取（direct memory access，DMA）及 option ROM。
- 至少 8 個 I/O base addresses，每一個都可以 disabled。
- 至少 3 個可選擇的 DMA 通道，加上一個 disabled 組態。

理想的 device 應該有一個 logical configuration，允許每一種可能分配。然而，讓 devices 有這樣的彈性會使它們更爲昂貴，這也就是爲什麼 "PC 95" 的認證（certification）要求並不那麼嚴格的原因。

Resource Descriptors（資源描述器）

Configuration Manager 在一個所謂的 **resource descriptor** 結構中說明每個資源的需求及分配。一個 logical configuration 可以有許多種資源，每一種資源可以有自己的 descriptor。這些 descriptors 就某方面來說有兩種作用，一方面你可以藉由完成 resource descriptor 並把它加到 logical configuration，將你的 device 的資源需求告知 Configuration Manager。Configuration Manager 會在內部複製一份 resource descriptor 結構，前面加以一個標準的表頭。另一方面，你可以在稍後取得並使用此 descriptors 在 Configuration Manager 中的版本。

依據所描述的資源種類的不同，resource descriptors 有不同的格式。下面說明四種 resource descriptors。

Interrupt Request Lines（IRQs）

一個 *IRQ_DES* 結構用以描述一個 IRQ：

```
typedef struct IRQ_DES_s
   {
   WORD    IRQD_Flags;         // 00 shared/unshared flags
   WORD    IRQD_Alloc_Num;     // 02 IRQ actually allocated
   WORD    IRQD_Req_Mask;      // 04 mask of acceptable IRQs
   WORD    IRQD_Reserved;      // 06 (reserved)
   } IRQ_DES;                  // 08
```

其中 *IRQD_Flags* 欄位可以設定爲 *fIRQD_Share*，表示 device 可分享一個 IRQ，或設定爲 0，表示 device 需要獨佔的 IRQ。*IRQD_Req_Mask* 是一個 bit mask，如果 device 支援 IRQ *n*，則 bit *n* 爲 1。一個可以處理任何 IRQ 的 device 應該將此欄位設爲 *FFFFh*，而一個需要 IRQ3 或 IRQ4 的 device 應該將此欄位設爲 *0018h*。在 Configuration Manager 最後建立起來的 allocated logical configuration 之中的 *IRQ_DES* 結構裡，*IRQD_Alloc_Num* 欄位應該存放真正指定給 device 的 IRQ。

memory

一個 **memory** resource descriptor 用到一個 *MEM_DES* 結構，並緊接著一個以上的 *MEM_RANGE* 結構：

```
typedef struct Mem_Des_s
    {
    WORD  MD_Count;        // 00 後面接著的 MEM_RANGEs 個數
    WORD  MD_Type;         // 02 總是等於 MTypeRange
    ULONG MD_Alloc_Base;   // 04 記憶體的起始位址
    ULONG MD_Alloc_End;    // 08 記憶體的結束位址
    WORD  MD_Flags;        // 0C flags
    WORD  MD_Reserved;     // 0E 保留
    } MEM_DES;             // 10

typedef struct Mem_Range_s
    {
    ULONG  MR_Align;       // 00 base alignment 的 mask
    ULONG  MR_nBytes;      // 04 bytes 數目
    ULONG  MR_Min;         // 08 最小位址
    ULONG  MR_Max;         // 0C 最大位址
    WORD   MR_Flags;       // 10 flags
    WORD   MR_Reserved;    // 12 保留
    } MEM_RANGE;           // 14
```

MD_Count 欄位是緊接在 *MEM_DES* 結構後面的 *MEM_RANGE* 結構個數。*MD_Type* 欄位在發展過程中可能有過不同的意義，但現在等於 *MTypeRange*，實際上即是 *MEM_RANGE* 結構的大小。在每一個 *MEM_RANGE* 結構裡，*MR_Align* 欄位用以說明配置所得的記憶體應如何排列，例如 FFFFF000h 表示 page alignment（邊界爲 4096 byte）。*MR_nBytes* 欄位表示所需記憶體區塊的大小（bytes）。*MR_Min* 及 *MR_Max* 欄位指出基底位址（base address）的範圍。舉個例子，爲了迫使一塊 page-aligned 記憶體在可接受的記憶體範圍內，你必須分別指定 000A0000h 及 000FFFFFh 給 *MR_Min* 及 *MR_Max*。*MR_Max*（此例爲 000FFFFFh）是這塊記憶體的最大結束位址的一個可能值。*MR_Flags* 欄位可能是 *fMD_ROM*（或 *fMD_RAM*）加上 *fMD_24*（或 *fMD_32*）。MR_Reserved 欄位應該是 0。

在 Configuration Manager 最後建立起來的 allocated logical configuration 之中的

MEM_DES 結構裡，*MD_Alloc_Base* 及 *MD_Alloc_End* 欄位應該存放被配置之記憶體的起始位址及結束位址。*MD_Flags* 欄位存放的將是 resource arbitrators（資源仲裁器）最終爲了滿足此一申請所使用之 *MEM_Range* 結構內的 flags 值。

I/O Port Address

如同 **memory** resource descriptors，**I/O port address** descriptors 使用一個 "header" 結構，後面跟著許多 "range" 結構：

```
typedef struct IO_Des_s
   {
   WORD IOD_Count;            // 00 後面接著的 IO_RANGEs 個數
   WORD IOD_Type'             // 02 總是等於 IOType_Range
   WORD IOD_Alloc_Base;       // 04 配置而得的基底位址
   WORD IOD_Alloc_End;        // 06 配置而得的結束範圍
   WORD IOD_DesFlags;         // 08 flags
   BYTE IOD_Alloc_Alias;      // 0A 配置而得的 alias offset
   BYTE IOD_Alloc_Decode;     // 0B 配置而得的 alias decode mask
   } IO_DES;                  // 0C

typedef struct IO_Range_s
   {
   WORD IOR_Align;            // 00 base alignment 的 mask
   WORD IOR_nPorts;           // 02 所需的 ports 個數
   WORD IOR_Min;              // 04 可允許的最低位址
   WORD IOR_Max;              // 06 可允許的最高位址
   WORD IOR_RangeFlags;       // 08 flags
   BYTE IOR_Alias;            // 0A alias offset
   BYTE IOR_Decode;           // 0B alias decode mask
   } IO_RANGE;                // 0C
```

大部份欄位的意義類似 **memory** resource descriptors 的對應欄位。例如，一塊 ISA 介面卡需要 16 個 ports 並以 16-byte 邊界開始，那麼就應該指定：

```
IOR_Align = 0xFFFF0;
IOR_nPorts = 0x10;
IOR_Min = 0x0100;
IOR_Max = 0x03FF;
```

這個例子所指出的 *IOR_Min* 及 *IOR_Max* 數值，遵循一個事實，那就是「standard ISA bus 只提供 10 位元的 I/O port 定址能力並保留位址 00h~FFh」。*IOR_Max* 表示配置所得的 ports 的最大可能結束位址。換句話說由於最大的 16-byte-aligned 基底位址是 3F0h，所以被索求的 16 個 ports 中的最後一個應該是在 3FFh。

Configuration Manager 最終爲 **allocated** logical configuration 建立的 *IO_DES* 結構裡，*IOD_Alloc_Base*, *IOD_Alloc_End*, *IOD_DesFlags*, *IOD_Alloc_Alias*, *IOD_Alloc_Decode* 等欄位可以指出真正指定給 device 的 port(s)。

DDK 對於「如何使用 *IOR_Alias* 及 *IOR_Decode* 欄位來完成一些不同的事情」提供了一個令人混淆的說明。這些欄位的用途之一是用來說明介面卡應該對 I/O port 位址中的多少個位元做解碼（decode）動作。一個標準的 ISA 介面卡只對最後 10 個位元解碼，這表示 3F8h，7F8h，BF8h 等等位址全都定址到相同的 physical hardware port。*IOR_Alias* 和 *IOR_Decode* 欄位的另一個用途是用以說明某些介面卡，這些卡使用多個位址，但又不是 400h, 800h 或某些大於 1024（2^{10}）的數值。這個設計太過複雜，Microsoft 計畫在 Windows 95 的未來版本中加以改變。表 **12-2** 說明這些欄位。

IOR_Alias	*IO_Decode*	意義
0	0	介面卡應該對 I/O 位址的所有 16 個位元解碼
4	3	介面卡應該對 I/O 位址的 10 個位元解碼
16	15	介面卡應該對 I/O 位址的 12 個位元解碼
255	0	介面卡是一張 PCI 卡，但可以使用原本被保留的「port 位址的最後 10 個位元」（它們被視爲 ISA 卡的 10 位元）

表 12-2　對於一項 I/O 資源，可能的 IOR_Alias / IOR_Deocde 設定

IOR_Alias=255 及 *IOR_Decode*=0 這兩項設定需要更多的說明。通常 **I/O 仲裁器**（I/O arbitrator，用來在彼此競爭的 devices 之間配置 I/O port 位址）會自動保留任何 port 位址的最後 10 個位元，這是 ISA 卡的解碼位址。也就是說，如果一個介面卡接受 ports 3F8h~3FFh，仲裁器通常不會提供像 7F8h~7FFh 或 BF8h~BFFh 這樣的位址給其它介面

卡，因爲這些位址無法以最後 10 個位元加以區別。在一個同時擁有 PCI bus 及 ISA bus 的系統裡，PCI bus 是主要的 bus，ISA bus 則係透過 bridge（橋接器）接到 PCI bus。這種情況下，ISA bus 不會看到大於 3FFh 的 I/O 位址，這表示 PCI 介面卡可以安全地使用「最後 10 位元相同」的 port 位址。爲了利用 PCI bus 的這項特性，將 *IOR_Alias / IOR_Decode* 設爲 255/0 就是告訴 I/O 仲裁器說，指定 port 位址時，不必擔心「最後 10 位元相同」。

最後要注意的是：EISA 卡使用的 I/O port 位址是由它們所佔據的 slot 決定。這樣的介面卡不需指定所有的 I/O 資源，因爲 EISA enumerator 會自動推測之。

Direct Memory Access（DMA）Channels

你可以使用一個 *DMA_RES* 結構來描述 DMA 資源：

```
typedef struct DMA_Des_s
   {
   BYTE DD_Flags;             // 00 flags
   BYTE DD_Alloc_Chan;        // 01 allocated channel
   BYTE DD_Req_Mask;          // 02 mask for supported channels
   BYTE DD_Reserved;          // 03 (reserved)
   } DMA_RES;                 // 04
```

DD_Flags 表示所需通道的寬度：*fDD_BYTE*, *fDD_WORD*, 或 *fDD_DWORD*。*DD_Req_Mask* 表示可以使用哪些通道，例如 60h 表示 device 可以使用通道 5 或 6。實際分配得到的通道編號，應該記錄在 Configuration Manager 最後爲 **allocated** logical configuration 所建立的 *DMA_RES* 結構裡頭的 *DD_Alloc_Chan* 欄位內。

裝置驅動程式（Device Drivers）

Plug and Play device driver 是一個 dynamic（動態載入的）VxD，除了有低階的硬體支援程式碼，以及 virtualization 函式以外，還提供 configuration 函式（用以取代「偵測 device 硬體資源」的傳統方法）。

在 Windows 95 之前，VxDs 必須查詢 SYSTEM.INI 裡的設定，或是其它地方的設定，才能決定使用哪些資源，否則 VxDs 就採取某個特別安排的假設。例如 Windows 3.1 Virtual Mouse Device（VMD）必須靠真實模式中的初始化段落，才能獲知它所操作的是 bus mouse 或 serial mouse，以及使用哪一個 IRQ（如果有的話）。VMD 的真實模式初始化段落，則是依賴真實模式的 mouse driver 才能得到那些資料。VMD 及其它 drivers 以這種方法所搜集到的資料，最後還得靠使用者在 CONFIG.SYS、SYSTEM.INI 及其它檔案裡頭輸入正確的設定值。

對終端使用者而言，「正確設置 devices」以及「傳送想要的設定值給 driver」是一項高難度工作，所以 Configuration Manager 就設法讓這個過程自動化。這就是 device driver 的 configuration 函式粉墨登場的時候。

Driver 藉由「註冊自己成爲一個 driver」以及「提供 configuration 函式位址」的動作，回應 PNP_New_Devnide 系統控制訊息。例如：

```
#0001  CONFIGRET OnPnpNewDevnode(DEVNODE devnode, DWORD loadtype)
#0002      {                        // OnPnpNewDevnode
#0003      switch (loadtype)
#0004          {                    // select function to perform
#0005
#0006      case DLVXD_LOAD_DEVLOADER:  // loadtype == 2
#0007          return CM_Register_Device_Driver(devnode, OnConfigure,
#0008              0, (CM_REGISTER_DEVICE_DRIVER_REMOVABLE
#0009              | CM_REGISTER_DEVICE_DRIVER_DISABLEABLE));
#0010          }                    // select function to perform
#0011
#0012      return CR_DEFAULT;
#0013      }                        // OnPnpNewDevnode
```

這個函式（位於書附光碟的 \CHAP11\SCHOOLBUS 目錄之中，TELEPATH.C 的一部份）首先檢查 *loadtype* 參數，藉以瞭解爲什麼 Configuration Manager 呼叫它。常數 *DLVXD_LOAD_DEVLOADER* 意謂著 Configuration Manager 希望這個函式找出並載入正確的 driver。Driver 時常會違反這個指示。本例的回應是呼叫函式 *CM_Register_Device_Driver*，爲傳入的 DEVNODE 登錄一個 configuration 函式。最後的

flag 參數表示這個特定的 driver 支援動態移除（dynamic removal）及重新組態（reconfiguration）。

Configuration 函式

Configuration 函式負責回應由 Configuration Manager 所發出的 configuration events。這些 event 代碼與上一章所討論的 enumeration 函式有著相同的 *CONFIG_* 常數。表 **12-3** 列出與 configuration 函式有關聯的訊息。一個 configuration 函式看起來像這樣：

```
#0001  CONFIGRET OnConfigure(CONFIGFUNC cf, SUBCONFIGFUNC scf,
#0002      DEVNODE devnode, DWORD refdata, ULONG flags)
#0003      {                          // OnConfigure
#0004      switch (cf)
#0005          {                      // select on message
#0006          ...
#0007      default:
#0008          return CR_DEFAULT;
#0009           }                      // select on message
#0010       }                         // OnConfigure
```

和 enumeration functions 一樣， configuration functions 在 debug 版中會收到一個假的 0x12345678 configuration 訊息，以檢查它們是否可以正確傳回 CR_DEFAULT。你應該只在真正成功處理一個特定的 request 後，才傳回 CR_SUCCESS。以下我將詳述 configuration function 應該如何回應這些訊息。

訊息	內容
CONFIG_APM	通知驅動程式，發生一個電源管理（power management）event
CONFIG_CALLBACK	通知驅動程式，*CM_CallBack_Device_Driver* 已經被呼叫
CONFIG_FILETR	通知驅動程式，過濾（限制）logical configurations
CONFIG_PREREMOVE	通知驅動程式，device 將被移除（從 tree 的底部往上傳遞）
CONFIG_PREREMOVE2	通知驅動程式，device 將被移除（從 tree 的頂部往下傳遞）
CONFIG_PRESHUTDOWN	通知驅動程式，系統將要關機
CONFIG_REMOVE	通知驅動程式，device 正從系統中移除
CONFIG_SHUTDOWN	通知驅動程式，系統正在關機
CONFIG_START	通知驅動程式，開始使用配置的組態
CONFIG_STOP	通知驅動程式，停止使用目前的組態
CONFIG_TEST	測試是否驅動程式可以停止使用此組態，或是否 device 可被移除
CONFIG_TEST_FAILED	通知驅動程式，先前的 CONFIG_TEST 失敗
CONFIG_TEST_SUCCEEDED	通知驅動程式，先前的 CONFIG_TEST 成功
CONFIG_VERIFY_DEVCICE	測試是否有傳統的（legacy）device 存在

表 12-3　Device drivers 的 configuration 相關訊息

CONFIG_START 機能　Device 獲得其組態，並應從中取出有關於資源的資料，以回應此一訊息。事實上，一般的 Plug and Play VxD 在處理 *CONFIG_START* 訊息時，會完成大部份的初始化工作。如果 device 只使用標準的資源種類，取得資源的最容易方法就是呼叫 *CM_Get_Alloc_Log_Conf*：

```
int irq;
...
CMCONFIG config;
CM_Get_Alloc_Log_Conf(&config, devnode,
    CM_GET_ALLOC_LOG_CONF_ALLOC);
irq = config.bIRQRegisters[0];
return CR_SUCCESS;
```

CM_Get_Alloc_Log_Conf 將以被指定之資源來填寫 *CMCONFIG* 結構（見圖 **12-2**）。這個結構記錄最多 7 個 IRQ，9 個記憶體區塊，20 個 I/O 基底位址，以及 7 個 DMA 通道。

```
typedef struct Config_Buff_s {
   WORD  wNumMemWindows;                        // 00 memory windows 的個數
   DWORD dMemBase[MAX_MEM_REGISTERS];           // 02 Memory window base [9]
   DWORD dMemLength[MAX_MEM_REGISTERS];         // 26 Memory window length [9]
   WORD  wMemAttrib[MAX_MEM_REGISTERS];         // 4A Memory window Attrib [9]
   WORD  wNumIOPorts;                           // 5C IO ports 的個數
   WORD  wIOPortBase[MAX_IO_PORTS];             // 5E I/O port base [20]
   WORD  wIOPortLength[MAX_IO_PORTS];           // 86 I/O port length [20]
   WORD  wNumIRQs;                              // AE IRQ info 的個數
   BYTE  bIRQRegisters[MAX_IRQS];               // B0 IRQ list [7]
   BYTE  bIRQAttrib[MAX_IRQS];                  // B7 IRQ Attrib list [7]
   WORD  wNumDMAs;                              // BE DMA channels 的個數
   BYTE  bDMALst[MAX_DMA_CHANNELS];             // C0 DMA list [7]
   WORD  wDMAAttrib[MAX_DMA_CHANNELS];          // C7 DMA Attrib list [7]
   BYTE  bReserved1[3];                         // D5 保留
} CMCONFIG;                                     // D8
```

圖 12-2　CMCONFIG 結構

CMCONFIG 結構裡的 *wNumIRQs* 欄位，表示有多少個 IRQs 已經分配給 device，陣列 *bIRQRegisters* 裡則記錄著那些 IRQs。陣列 *bIRQAttrib* 記錄每一個 IRQ 的屬性；目前唯一的屬性是 *fIRQD_Share* 旗標，表示與其它 devices 共享。通常你需要呼叫 *VPICD_Virtualize_IRQ* 以虛擬化你所擁有的 IRQs。

wNumMemWindows 表示分配多少個記憶體區塊，*dMemBase* 記錄其基底位址。陣列 *dMemLength* 記錄記憶體區塊的大小，陣列 *wMemAttrib* 說明區塊的屬性。一般而言，基底位址就是實體位址（physical address），可爲 32 位元位址空間的任何位置。通常 devices 會使用 1st MB 裡頭的配接卡區域（adapter region，也就是位址 000A0000h~000FFFFFh）的記憶體。陣列 *wMemAttrib* 裡頭的資料表示分配所得的記憶體位置，其內容可能是 *fMD_ROM* 或 *fMD_RAM*，以及 *fMD_24* 或 *fMD_32*。

爲了在一個 VxD 中存取記憶體區塊，你必須將 *dMemBase* 裡的 physical address（實體位址）轉換爲 virtual address（虛擬位址）。在 Windows 3.1 之中，你必須呼叫

_MapPhysToLinear，其所獲得的位址在 Windows session 生存期間將總是指向相同的 physical 位置。你也可以在 Windows 95 中呼叫 *_MapPhysToLinear*，但是有個主要的缺點：如果你的 device 被重新設置（reconfigured）爲「使用不同的記憶體位址」，那麼你正在使用的線性位址空間中的區域，對接下來的 Windows session 實際沒有效用。在 Windows 95 之中，你應該使用新的 page mapping services 來取代 *_MapPhysToLinear*：

```
ULONG npages = (physsize + 4095) >> 12;
ULONG firstpage = physaddr >> 12;
DWORD linaddr = _PageReserve(firstpage, npages, PF_FIXED);
_PageCommitPhys(linaddr >> 12, npages, firstpage,
    PC_INCR | PC_WRITEABLE);
_LinPageLock(linaddr >> 12, npages, 0);
```

其中的 *physsize* 內含的是一塊 physical memory 的長度，以 byte 爲單位（例如 *config.dMemLength[something]*）；*physaddr* 內含的則是起始的 physical address（例如 *config.dMemBase[something]*）。這段碼假設 physical address 以 page（4096-byte）爲邊界，使用 *_PageReserve* 來配置足夠的 virtual address pages，以便能夠涵蓋 device memory。*PR_FIXED* flag 表示與記憶體區塊有關的 physical addresses 不能被改變。*_PageCommitPhys* 會建立好 page tables，透過那裡，記憶體可以被存取。*PC_INCR* flag 表示 *_PageCommitPhys* 在建立 page tables 時，要同時增加 linear 及 physical 位址（譯註：也就是說 linear pages 映射到同樣數目的連續 physical pages）。*PC_WRITEABLE* flag 表示記憶體可以讀寫。如果 ring3 程式需要存取這塊記憶體，你可以指定 *PC_USER* flag。

CMCONFIG 結構裡的 *wNumIOPorts* 欄位表示，有多少塊的 port 位址已經被指定給 device。每一個指定包括基底位址（記錄於 *wIOPortBase*）及 port 個數（記錄於 *wIOPortLength*）。

最後，*wNumDMAs* 欄位表示有多少個 DMA channels 已經分配給 device，陣列 *bDMALst* 之中內含它們的編號。陣列 *wDMAAttrib* 則使用 *fDD_BYTE*, *fDD_WORD*, *fDD_DWORD* 來表示被指定的通道寬度。

如果不使用 *CM_Get_Alloc_Log_Conf*，你可以使用更麻煩的方法：針對 allocated configuration，列舉所有的 resource descriptors：

```
LOG_CONF logconf;
RES_DES hres;
RESOURCEID restype;

CM_Get_First_Log_Conf(&logconf, devnode, ALLOC_LOG_CONF);
hres = (RES_DES) logconf;
While (CM_Get_Next_Res_Des(&hres, hres,
    ResType_All, &restype, 0) == CR_SUCCESS)
    {                   // for each resource
    switch (restype)
        {               // select on resource code
    case ResType_IRQ:
        {               // IRQ resource
        IRQ_DES* pirq = (IRQ_DES*) hres;
        [do something with pirq->IRQD_Alloc_Num]
        break;
        }               // IRQ resource
    case ResType_Mem:
        ...
    case ResType_IO:
        ...
    case ResType_DMA:
        ...
        }               // select on resource code
    }                   // for each resource
```

呼叫 *CM_Get_First_Log_Conf* 可以取得第一個 logical configuration（最後參數將指定其種類）。我們對 "allocated" configuration 有興趣，其中內含我們已經分配到的資源。每次呼叫 *CM_Get_Next_Res_Des* 即可取得一個新的 resource descriptor。你可以設定 *hres* 等於 logical configuration handle，執行迴圈，以走遍所有的 resource descriptors。在迴圈的每次迭代期間，你應該提供前一個 resource descriptor（或 logical configuration）的 handle，做爲第二個參數，並以你感興趣的資源種類當做第三個參數。如果指定 *ResType_All*，表示你對所有的資源種類感興趣。第一個參數應該指向存放下一個 resource handle 的位置，第四個參數應該指向存放 resource type 的位置；如果你對這些資訊不感興趣，可以分別（或全部）指定爲 NULL。最後一個參數用以指定 flags，目前應該爲 0。

每次 *CM_Get_Next_Res_Des* 傳回一個 resource descriptor，前面例子中便以 switch 敘述來處理 *restype* 傳回值。傳回值 *hres*（resource handle）只是 resource descriptor 在 Configuration Manager 內部複製的 flat address，可由 *restype* 知道其格式。舉個例子，如果 *CM_Get_Next_Res_Des* 取得一個 *ResType_IRQ* 資源，則 *hres* 應該指向 *IRQ_DES* 結構。

取得組態資料的第二種方法（也就是上述走訪所有 resource descriptors 的方法）看起來比第一個方法（呼叫 *CM_Get_Alloc_Log_Conf*）困難許多，你可能想知道爲什麼要那樣做。我能夠想到的唯一理由是，你也許想要使用 private resource（也就是你自己的 arbitrator 所提供的資源）。上述這種「走訪整個 configurations」的方法，是唯一獲得 private resource 的手段，因爲 *CMCONFIG* 結構無法應付這種情況。

CONFIG_FILTER 機能　在分配資源以前，Configuration Manager 會提供一個機會給 devices 及其 enumerators，讓它們藉由送出 *CONFIG_FILTER* 訊息來 "filter"（過濾）它們的 logical configurations。過濾程序所要解決的問題是，device 或其 drivers 對於組態設置的彈性，最初可能不切實際地過於樂觀。爲了處理這種情況，你應該執行迴圈，取得所有的 logical configurations 及所有資源，修改你不喜歡的 resource descriptors：

```
LOG_CONF logconf;
code = CM_Get_First_Log_Conf(&logconf, devnode,
   FILTER_LOG_CONF);
while (code == CR_SUCCESS)
   {                    // for each configuration
   RESOURCEID restype;
   RES_DES hres = (RES_DES) logconf;
   while (CM_Get_Next_Res_Des(&hres, hres, ResType_All,
                              &restype, 0) == CR_SUCCESS)
      {                 // for each resource
      switch (restype)
         {              // process this resource
         [filter resource requirements]
         }              // process this resource
      }                 // for each resource
   code = CM_Get_Next_Log_Conf(&logconf, logconf, 0);
   }                    // for each configuration
```

這個例子使用一個外部迴圈來走訪所有的 logical configurations。我們首先呼叫 *CM_Get_First_Log_Conf* 取得第一個指定類型的 logical configuration。在這裡，我們要求第一個 configuration 是一個 "filtered" configuration。呼叫 *CM_Get_Next_Log_Conf* 可以反覆執行外部迴圈。另有一個內部迴圈，呼叫 *CM_Get_Next_Res_Des* 取得連續的 resource descriptors。此迴圈的程式邏輯和前面討論的 *CONFIG_START* 訊息的迴圈一樣。在內部迴圈裡，你可以把 *hres* 變數看成一個指向「類別爲 *restype*」的 resource descriptor 的指標，並直接在 descriptor 中完成必要的改變。

PCMCIA 可以說明爲什麼 configuration 可能需要過濾（filtered）。一塊對記憶體有需求的介面卡，可以在 32 位元線性位址空間的任何地方要求記憶體，並指定任何的 alignment 及大小。然而在 ISA bus 機器中，DMA 只能在起始的 16 MB 記憶體範圍內使用。另外，將記憶體配置爲 page-aligned，並讓其大小爲 page 倍數，將對效率有幫助。因此 PCCARD device 必須修改所有的 **memory** resource descriptors 以限制可允許的記憶體範圍。前一個例子中，我們可以在內部迴圈使用以下的 switch 句子，完成整個修改（PCCARD 所做的修改事實上更複雜，但此例傳達了一般觀念）：

```
case ResType_Mem:
    {                   // filter memory resource
    MEM_DES* mem = (MEM_DES*) hres;
    int i;
    MEM_RANGE* range = (MEM_RANGE*) (mem+1);
    ULONG maxmax;

    for (i = 0; i < mem->MD_Count; ++i)
        {               // for each memory range
        range->MR_Align &= ~4095; // require page alignment
        range->MR_nBytes = (range->MR_nBytes + 4095 ) & ~4095;
        maxmax = 0x00FFFFFF - range->MR_nBytes;
        if (range->MR_Max > maxmax)
            range->MR_Max = maxmax;
        ASSERT(range->MR_Min <= range->MR_Max);
        range = (MEM_RANGE*) ((DWORD) range + mem->MD_Type);
        }               // for each memory range
    }                   // filter memory resource
```

CONFIG_REMOVE, CONFIG_SHUTDOWN, CONFIG_STOP 機能　當 Configuration Manager 希望 (**1**) 移除 device（*CONFIG_REMOVE*），(**2**) 將系統關機（*CONFIG_SHUTDOWN*），或 (**3**) 改變資源配置以配合新的 device（*CONFIG_STOP*）時，它會送出對應的一個申請。驅動程式收到請求時，應該停止使用目前的組態。舉個例子，如果你的驅動程式目前正在虛擬化一個 IRQ，它應該停止做這個動作。當 Configuration Manager 想要重新裝置（configure）你的驅動程式以便取得資源，驅動程式可能會在 Windows session 之中收到 *CONFIG_STOP* 訊息。驅動程式可能也會在 Windows session 之中收到 *CONFIG_REMOVE* 訊息，表示使用者正在移除 device。事實上，Windows 95 以 *CONFIG_REMOVE* 代替 *CONFIG_STOP*，但是後繼的 Windows 版本會使用 *CONFIG_STOP*。當系統關機時，你的驅動程式應該收到 *CONFIG_SHUTDOWN*（如果它是 boot configuration 的一部份）或 *CONFIG_REMOVE*（如果它不是 boot configuration 的一部份）。

CONFIG_TEST 機能　*CONFIG_TEST* 訊息是一項申請，詢問是否可以將目前的組態視爲無效（配合 *CONFIG_TEST_CAN_STOP* 子機能），或是否可以移除 device（配合 *CONFIG_TEST_CAN_REMOVE* 子機能）。你的驅動程式應該傳回 *CR_SUCCESS* 以示許可，或傳回 *CR_FAILURE* 以示否決。

CONFIG_VERIFY_DEVICE 機能　當 root enumerator 列舉傳統（legacy）的 devices 並收到 *CR_DEVNODE_ALREADY_THERE* 傳回值時，它會發出一個 *CONFIG_VERIFY_DEVICE* 訊息。這項機能的目的是爲了找出 device 是否仍然存在。如果 device 還存在，驅動程式應該傳回 *CR_SUCCESS*，否則應該傳回 *CR_DEVICE_NOT_THERE*。

資源仲裁器（Resource Arbitrators）

每一種資源（IRQ，memory address，I/O address，以及 DMA）都有一個資源仲裁器，負責在所有彼此競爭的 devices 之間配置資源。Microsoft 認爲很少有開發人員需要撰寫仲裁器（arbitrators）；所以針對這方面的說明並不多。爲了說明仲裁（arbitration）的過程，

我將舉出一個應用於虛構之 telepathic channel resource 的仲裁器。此一資源是連接到 School Bus 的每一個 telepathic I/O stream device 所必需。爲了辨識這項資源，我使用一種在 CONFIGMG.H 中有所記錄的習慣：Microsoft 指定 10 位元的 OEM ID，我再加上 5 位元的 resource code。假設我的 OEM 編號爲 10h（Microsoft 保留 0~F），而這是我曾經定義過的第六筆資源：

```
#define ResType_Telepath ((0x10 << 5) | 5)
```

注意，這個資源確認方法，只考慮 32 位元裡的 15 個位元。定義在 CONFIGMG.H 裡的 00008000h-bit 表示 *ResType_Ignored_Bit*；在 resource ID 中設定這個位元，表示此項資源沒有仲裁器。不使用剩餘的 16 位元是因爲，Microsoft 希望 16 位元程式碼一樣能夠與 Configuration Manager 一起工作。

呼叫 *CM_Register_Arbitrator*（譯註：原文誤寫爲 *CM_Register_Enumerator*）可以登錄一個資源仲裁器：

```
REGISTERID arbid;
CONFIGRET OnArbitrateTelepath(ARBFUNC af, ULONG refdata,
   DEVNODE devnode, NODELIST_HEADER h);
...
CM_Register_Arbitrator(&arbid, ResType_Telepath,
   OnArbitrateTelepath, 0, NULL, ARB_GLOBAL);
```

在這個例子中，我要爲新的全域資源登錄一個仲裁器，所以我在呼叫 *CM_Register_Arbitrator* 時使用 *ARB_GLOBAL* flag，並將倒數第二個參數設爲 NULL。你也可以擁有一個屬於特定 DEVNODE 的資源，這種情況下，請使用 *ARB_LOCAL* 並以 DEVNODE 位址做爲數第二個參數。你應該這樣定義仲裁函式（arbitration function）：

```
CONFIGRET OnArbitrateTelepath(ARBFUNC af, ULONG refdata,
   DEVNODE devnode, NODELIST_HEADER h)
   {                        // OnArbitrateTelepath
   switch (af)
      {                     // select on function
      ...
      }                     // select on function
   }                        // OnArbitrateTelepath
```

其中的 *af* 參數是表 **12-4** 所列出的訊息之一，*refdata* 是由 *CM_Register_Arbitrator* 第四個參數所提供的資料（本例爲 0），*devnode* 是 *CM_Register_Arbitrator* 的第五個參數，*h* 是一個指向 *nodelistheader* 結構的指標，此結構內部指向需要設定之 DEVNODEs 串列的頭尾兩端（見圖 **12-3**）。此串列中的每個 node 指向一個 DEVNODE（以 *nl_ItsDevNode* 欄位表示）及一個 logical configuration（以 *nl_Test_Req* 欄位表示）。每個 node 也擁有一個 *nl_ulSortDWord* 欄位，讓仲裁器排列順序時使用。與每個 node 都有關係的 logical configuration，內含 resource descriptors，其中一些可能是我們要仲裁的對象。

訊息	內容
ARB_TEST_ALLOC	指示仲裁器處理資源的 trial allocation
ARB_RETEST_ALLOC	指示仲裁器檢查前一次的 trial allocation
ARB_FORCE_ALLOC	指示仲裁器重新測試但是不要拋棄前一次的 trial allocation
ARB_SET_ALLOC	指示仲裁器接受前一次的 trial allocation
ARB_RELEASE_ALLOC	指示仲裁器放棄前一次的 trial allocation
ARB_QUERY_FREE	要求有關 free resource 的資訊
ARB_REMOVE	指示仲裁器準備移除自己

表 12-4　仲裁函式（Arbitrator function）的訊息

除非真的發生錯誤，否則仲裁函式對於所有標準的機能代碼都應該傳回 *CR_SUCCESS*，對於任何無法辨識的機能代碼則應該傳回 *CR_DEFAULT*。未來的 Windows 版本中，Configuration Manager 可以使用 *CR_DEFAULT* 傳回值來測試版本的相容性。

Configuration Manager 基本的資源分配演算法，是以 *ARB_TEST_ALLOC* 訊息，呼叫所有的資源仲裁器，以執行一次 trial allocation。一旦仲裁器產生一組有用的資源分配，Configuration Manager 會送出 *ARB_SET_ALLOC* 訊息以表示接受此 trial allocations。爲了處理這些及其它仲裁器機能，我的範例中維護了一個配置映射表（allocation map），

稱爲 *free_map*，由 short bit string 組成。如果編號 *n* 的 telepathic channel 沒有使用，則映射表裡頭的位元 *n* 爲 1，如果編號 *n* 的 channel 已經分配給某個 device，則映射表裡的位元 *n* 爲 0。下一節將說明資源仲裁器如何回應每一個仲裁器訊息。

```
struct nodelistheader_s
    {                                   // node list header
    struct nodelist_s* nlh_Head;        // 00 head of list
    struct nodelist_s* nlh_Tail;        // 04 tail of list
    };                                  // 08 node list header

struct nodelist_s
    {                                   // node list element
    struct nodelist_s* nl_Next;         // 00 chain to next element
    struct nodelist_s* nl_Prev;         // 04 chain to previous element
    struct devnode_s* nl_ItsDevNode;    // 08 a device node that needs
                                        //    resource
    struct Log_Conf* nl_Test_Req;       // 0C a configuration for that DEVNODE
    ULONG nl_ulSortDWord;               // 10 sort order for CM_Sort_NodeList
    };                                  // 14 node list element
```

圖 12-3　Arbitrator node 結構

ARB_TEST_ALLOC 機能　對於所要處理的仲裁訊息，*ARB_TEST_ALLOC* 是最基本也是最複雜的。在我們的例子中，它的任務是設法分配一個 telepathic channel 給每個有需要的 device。下面的碼（位於光碟 \CHAP11\SCHOOLBUS 目錄中，是 SCHOOL.C 的一部份）實作了一個典型的演算法（變數及輔助函式將在這一節稍後說明）：

```
case ARB_TEST_ALLOC:        // af == 0
    {                     // ARB_TEST_ALLOC
    sortnodes((NODELISTHEADER) h);
    free_copy = free_map;
    release((NODELISTHEADER) h, &free_copy);

    ALLOCPLACE place = {((NODELISTHEADER)h)->nlh_Head,
        NULL};
    if (allocate(place, &free_copy))
        return CR_SUCCESS;
    else
        return CR_FAILURE;
    }                     // ARB_TEST_ALLOC
```

這個資源分配演算法的第一步驟是排列 devices 的順序，使愈挑剔（譯註：意指可供選擇的資源數目愈少）的 devices 排在串列的最前面。我們根據有多少個不同的 channels 可滿足 device 的需求，來指定每個 device 的挑剔指數（fussiness index）。挑剔的 devices 比不挑剔的 devices 擁有更少的選擇性。一個 telepathic resource descriptor 可能看起來像這樣：

```
#define ResType_Telepath ((0x10 << 5) | 5)

typedef struct
    {
    int allocated;   // index of allocated channel (-1 --> none)
    ULONG requested; // mask for requested channels
    } TELEPATH_RESOURCE;
```

我設計了一些輔助函式來處理排序動作。*bitcount* 函式只是簡單地計算參數 mask 之中位元值 1 的位元個數，*sortnodes* 函式則根據這個個數來排列 nodes 順序。

```
typedef struct nodelistheader_s *NODELISTHEADER;
typedef struct nodelist_s *PNODE;

int bitcount(ULONG mask)
    {                          // bitcount
    int nbits = 0;
    while (mask)
        {                      // count bits
        if (mask & 1)
            ++nbits;
        mask >>= 1;
        }                      // count bits
    return nbits;
    }                          // bitcount

void sortnodes(NODELISTHEADER h)
    {                          // sortnodes
    PNODE node = h->nlh_Head;
    while (node)
        {                      // for each node
        RES_DES hres = (RES_DES) node->nl_Test_Req;
        #define pres ((TELEPATH_RESOURCE *) hres)
```

```
        node->nl_ulSortDWord = 0;
        while (CM_Get_Next_Res_Des(&hres, hres,
            ResType_Telepath, NULL, 0) == CR_SUCCESS)
            node->nl_ulSortDWord += bitcount(pres->requested);
        node = node->nl_Next;

        #undef pres
        }                       // for each node
    CM_Sort_NodeList((NODELIST_HEADER) h, 0);
    }                           // sortnodes
```

排序函式在 node 串列上執行一個迴圈，以便根據 telepathic descriptor 的 *requested* 欄位計算排列序號（sorting ordinals），然後呼叫 *CM_Sort_NodeList* 來處理真正的排序動作。在這個迴圈裡，排序函式執行 *CM_Get_Next_Res_Des* 以遍訪 telepathic resources 並累計可供選擇的 channels 個數。*bitcount* 輔助函式會計算 TELEPATH_RESOURCE *requested* 欄位裡位元爲 1 的個數，其中的 *requested* 欄位是一個 bit mask，用來表示這個 device 所支援的 channels。

現在讓我們複製實際的配置映射表（allocation map）。我們應該使用這份複製品來記錄最後的 trial allocation。假設電腦可以容納最多 8 個 channels，那麼 "實際" 映射表和複製品中使用 static DWORD 變數就可以了。一個簡單的指定句就足以建立有用的複製品：

```
static DWORD free_map = 0x000000FF;
static DWORD free_copy;
...
free_copy = free_map;
```

free_map 的初始值表示 channels 0~7 一開始是自由可用的。

Trial allocation 的第三步驟是更新 trial allocation 映射表，使衝突的 devices 所使用的資源可以釋放出來。畢竟，我們是要分配新的 channels 給這些 devices。我寫了另一個輔助函式來處理這個步驟：

```
void release(NODELISTHEADER h, PULONG pmap)
    {                       // release
    PNODE node = h->nlh_Head;
    while (node)
        {                   // for each node
        LOG_CONF logconf;

        if (CM_Get_First_Log_Conf(&logconf,
            (DEVNODE) node->nl_ItsDevNode, ALLOC_LOG_CONF)
            == CR_SUCCESS)
            {               // release channel(s)
            RES_DES hres = (RES_DES) logconf;
            #define pres ((TELEPATH_RESOURCE *) hres)
            while (CM_Get_Next_Res_Des(&hres, hres,
                ResType_Telepath, NULL, 0) == CR_SUCCESS)
                if (pres->allocated >= 0)
                    *pmap |= 1 << pres->allocated;
            #undef pres
            }               // release channel(s)
        node = node->nl_Next;
        }                   // for each node
    }                       // release
```

對於串列中的每一個 node，我們呼叫 *CM_Get_First_Log_Conf* 找出唯一適合此 device 的 "allocated" logical allocation。配置得來的 channel 被儲存在 telepathic resource descriptor 的 allocated 欄位中。如果真找到一個，我們遍訪所有的 telepathic channel resource，以清除在 *pmap* bit map 中用以表示「目前被配置之 channel」的位元。

Trial allocation 的第四步驟（最後步驟）是試著找出沒有被使用的 channels 給有需要的 devices 使用。理論上一個 device 可以要求一個以上的 channels，但這種情況只在走訪 logical configuration 中的多個 telepathic channel resource descriptors（與 node list 中的這個 device's entry 有關）時才會出現。我們指定 channels 的那個實體（entity），真的是一個 resource descriptor，而不是個 configuration 或是個 device，而且 resource descriptors 位於一個 interrupted list 之中，如圖 **12-4** 所示。

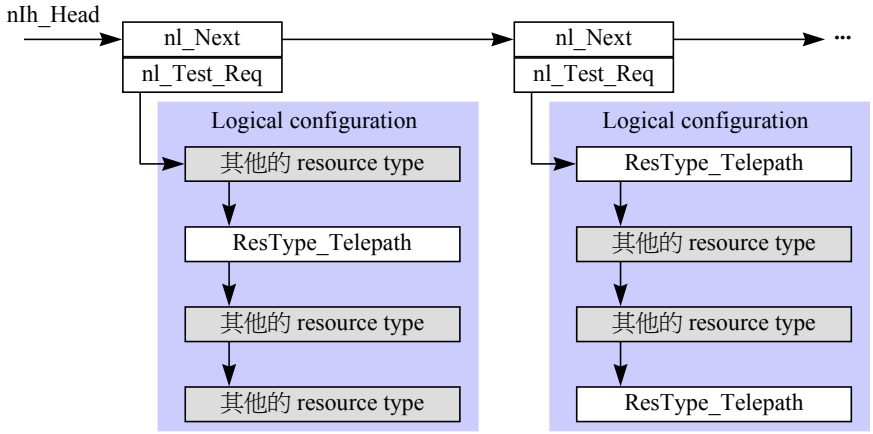


圖 12-4 Resource descriptors 的 interrupted list

爲了使走訪 interrupted list 儘可能容易些，我定義了一個輔助用的資料結構來表示串列內的位置：

```
typedef struct
    {                                   // allocation placeholder
    PNODE node;                         // current node
    TELEPATH_RESOURCE* pres;            // current resource descriptor
    } ALLOCPLACE, *PALLOCPLACE;         // allocation placeholder
```

其中 *node* 欄位表示一個 node list 元素的位址，*pres* 欄位表示一個 resource descriptor 的位址，此 descriptor 屬於 *nl_Test_Req* 所示之 logical configuration 所有。下面的輔助函式從在 list 中的一個位置進行到下一個位置：

```
BOOL nextres(PALLOCPLACE p)
    {                           // nextres
    ASSERT(p->node && p->node->nl_Test_Req);
    if (!p->pres)
        p->pres = (TELEPATH_RESOURCE *) p->node->nl_Test_Req;
    while (CM_Get_Next_Res_Des((RES_DES*) &p->pres,
        (RES_DES) p->pres, ResType_Telepath, NULL, 0)
        != CR_SUCCESS)
        {                       // no more descriptors of our resource
```

```
        if (!(p->node = p->node->nl_Next))
            return FALSE;       // no more nodes in the list
        p->pres = (TELEPATH_RESOURCE *) p->node->nl_Test_Req;
        }                       // no more descriptors of our resource
    return TRUE;
    }                           // nextres
```

下面這個函式真正完成 trial allocation；它被我們的 *ARB_TEST_ALLOC* 訊息處理碼（先前曾出現過）呼叫起來：

```
BOOL allocate(ALLOCPLACE place, PULONG pmap)
    {                           // allocate
    if (!nextres(&place))       // no resource descriptor...
        return TRUE;            // ...nothing to do

    place.pres->allocated = -1;

    for (int channel = 0; channel < 8; ++channel)
        {                       // try to allocate a channel
        ULONG mask = 1 << channel;
        ULONG tempmap = *pmap;
        if ((tempmap & mask) && (place.pres->requested & mask))
            {                   // do trial allocation
            tempmap &= ~mask;
            if (allocate(place, &tempmap))
                {               // successful allocation
                *pmap = tempmap;
                place.pres->allocated = channel;
                return TRUE;
                }               // successful allocation
            }                   // do trial allocation
        }                       // try to allocate a channel
    return FALSE;               // all channels in use
    }                           // allocate
```

這個函式在 resource descriptors 的 interrupted list 中以遞迴的方式執行，以便爲每一個 device 指定 channels。每執行一次函式遞迴，表示對 telepathic channel 的一個需求。函式巡訪所有可能的 channels，看看迴圈索引值所對應的 channel 是否可以滿足需求。第一個測試動作（*tempmap* & *mask*）決定前次遞迴所提供的 allocation map 中所記錄的 channel 是否可用。還記得嗎，bit 如爲 1 表示 channel 可用。最外層遞迴會收到仲裁器

所擁有的 *free_copy* map，然而內層遞迴會收到屬於上一層遞迴的 map。如果這項檢查通過（譯註：亦即 *tempmap & mask* 不爲 0），下一個檢查（*place.pres->requested & mask*）會決定是否 channel 符合 device 的需求（由其 telepathic resource descriptor 指定）。如果這項檢查也通過（譯註：亦即 *place.pres->requested & mask* 不爲 0），我們就在自己的暫時映射表中標記「已使用」符號（*tempmap &= ~mask*），並藉由遞迴呼叫 *allocate* 函式來滿足所有後續的要求。如果遞迴呼叫成功，我們就將我們的暫時映射表複製到呼叫者的映射表中（譯註：即 *pmap = tempmap 這個動作），並且在 resource descriptor 中記錄下目前的 channel。如果遞迴呼叫失敗，我們就對下一個 channel 重複所有步驟。

每一次遞迴呼叫 *allocate*，會收到呼叫者的 *ALLOCPLACE* 結構的一份複製品（也就是變數 place）。這麼做的理由是，如此一來，每次遞迴呼叫 *allocate* 函式，在其中（一開始處）呼叫 *nextres* 函式，可以獲得相同的 resource descriptor。

如果最外層配置順利完成，仲裁器將成功的 trial allocation 結果儲存於 *free_copy* map 時，應該傳回 *CR_SUCCESS*。否則，仲裁器應該傳回 *CR_FAILURE*，表示無法滿足記錄在 node list 中的所有要求。

ARB_RETEST_ALLOC 與 ARB_FORCE_ALLOC 機能 Configuration Manager 利用 *ARB_RETEST_ALLOC* 訊息，檢查看看記錄在一組特別之 logical configurations 裡頭的資源分配，是否仍然可以使用。*ARB_FORCE_ALLOC* 的功能類似，但一定不能以失敗收場。一旦你知道如何實作這兩個訊息的處理函式以後，你就很容易明白爲什麼 Configuration Manager 要使用這兩個訊息了。以下是個例子：

```
case ARB_RETEST_ALLOC:      // af == 1
case ARB_FORCE_ALLOC:       // af == 6
   free_copy = free_map;
   release((NODELISTHEADER) h, &free_copy);

   if (reallocate((NODELISTHEADER) h, &free_copy,
      af == ARB_FORCE_ALLOC))
      return CR_SUCCESS;
   else
      return CR_FAILURE;
```

這段碼使用輔助函式 *release* 來釋放記錄在 temporary allocation map 裡頭的資源，然後呼叫 *reallocate*，配置出在 node list 裡頭由 *ul_Test_Req* configuration 所描述的資源：

```
BOOL reallocate(NODELISTHEADER h, PULONG pmap, BOOL forced)
   {                        // reallocate
   PNODE node = h->nlh_Head;

   while (node)
      {                     // for each node
      RES_DES hres = (RES_DES) node->nl_Test_Req;
      #define pres ((TELEPATH_RESOURCE *) hres)

      while (CM_Get_Next_Res_Des(&hres, hres,
         ResType_Telepath, NULL, 0) == CR_SUCCESS)
         {                  // requires our resource
         ULONG mask;

         ASSERT(pres->allocated >= 0);
         mask = 1 << pres->allocated;
         if ((*pmap & mask) && !forced)
            return FALSE;   // one or more still in use
         *pmap &= ~mask;
         }                  // requires our resource
      node = node->nl_Next;

      #undef pres
      }                     // for each node
   return TRUE;
   }                        // reallocate
```

reallocate 函式執行一個迴圈，將 nodes 巡訪一遍，該迴圈裡有另一個迴圈，走訪 telepathic channel resources。每個 resource descriptors 應該包含被指定的 channel（*pres->allocated*）。如果 channel 可使用，我們便在上面標記「已被配置」的記號。如果 channel 不可用，我們的行爲將依據收到的訊息是 *ARB_RETEST_ALLOC* 或 *ARB_FORCE_ALLOC* 而定。在 "retest" 情況下，我們令此呼叫失敗。在 "force" 情況下，我們允許重複指定。

ARB_RETEST_ALLOC 訊息有一個可想而知的用途，那就是傳回 Configuration Manager 初期認爲可使用，但還不是很理想的組態。Configuration Manager 會送出這個訊息給仲裁器，另有兩個原因。一個原因是爲了以開機（boot time）之後剩餘的任何資源，初始化 allocation map。爲了執行這個初始化動作，Configuration Manager 送給每個仲裁器一個 *ARB_RETEST_ALLOC* request，夾帶一個 node list，指向 boot logical configurations。與此一呼叫有關之 DEVNODEs，可能並未擁有 "allocated" logical configurations，此情況下 *release* 函式將什麼也不做。

Configuration Manager 使用 *ARB_RETEST_ALLOC* 的第二個情況是在移除 DEVNODE 的時候。這種情況下，Configuration Manager 送出含有某一 node list 元素的申請，其中說明了預備移除的 device。由於 *ul_Test_Req* logical configuration 並沒有任何資源，所以 *reallocate* 函式不做任何事情。是的，由 Configuration Manager 送出這個訊息，只是爲了釋放在 DEVNODE 的 "allocated" configuration 中所聲稱的資源。

ARB_FORCE_ALLOC 訊息的目的是爲了要實作出由使用者在「控制台（control panel）」中強制設定的 configuration。在此情況下，重複指定是允許的，因爲使用者必定知道什麼才是最佳設定。

ARB_SET_ALLOC 機能　在成功呼叫 *ARB_TEST_ALLOC, ARB_RETEST_ALLOC,* 或 *ARB_FORCE_ALLOC* 之後，Configuration Manager 會以 *ARB_SET_ALLOC* 呼叫仲裁器，委派以 trial allocation：

```
case ARB_SET_ALLOC:        // af == 2
   ASSERT(free_copy != 0xDEADBEEF);
   free_map = free_copy;
   free_copy = 0xDEADBEEF;
   return CR_SUCCESS;
```

這項機能會把目前的 trial allocation map（*free_copy*）複製到「真實的」map （*free_map*）中。本例之中，我將 *free_copy* 重新設定爲可辨識的位元格式（bit pattern），以便有人想要處理未初始化版本的 trial map 時，可以明顯地發現它。常數 *DEADBEEFh* 出現在

Configuration Manager 的很多地方，用來表示尚未初始化的資料。

ARB_RELEASE_ALLOC 機能　在不完美的真實世界裡，滿足每一個 device 是不可能的。舉個例子，我們可能有兩個 devices，都只能使用 channel 0。這種情況下，*allocate* 函式應該傳回 FALSE，而仲裁器應該傳回 *CR_FAILURE*，表示配置失敗。Configuration Manager 應該發出 *ARB_RELEASE_ALLOC* 訊息，允許仲裁器放棄目前無用的 trial allocation，然後試著使用其他存在的 logical configurations 來進行 trial allocation。如果無法完成配置，Configuration Manager 應該 disable 其中一個 device。Device Manager 應該記錄此一衝突狀況，好讓使用者明白爲什麼被 disable 的 device 不能使用。

一般而言，你可能有記憶體或其它程式資源與 trial map 結合。現在是釋放它們的時候。在我們所考慮的仲裁器例子中，這項機能幾乎是沒有用途的：

```
case ARB_RELEASE_ALLOC:     // af == 3
   free_copy = 0xDEADBEEF;
   return CR_SUCCESS;
```

ARB_QUERY_FREE 機能　當一個 VxD 或一個應用程式呼叫 *CM_Query_Arbitrator_Free_Size* 或 *CM_Query_Arbitrator_Free_Data* 時，Configuration Manager 會送出 *ARB_QUERY_FREE* 訊息。這些機能會取得仲裁器用以控制配置行爲的內部資料大小，或是資料本身。對於這項機能，Configuration Manager 會以 *arbitfree_s* 結構位址來取代 *NODELIST_HEADER* 參數：

```
struct arbitfree_s {
   PVOID *af_PointerToInfo;  // arbitrator 的資料
   ULONG af_SizeOfInfo;      // 資料的長度
};
```

仲裁器會以內部資料的位址及大小填滿這個結構。例如：

```
case ARB_QUERY_FREE:           // af == 4
   {                           // ARB_QUERY_FREE
   struct arbitfree_s *p = (struct arbitfree_s *) h;
   p->af_SizeOfInfo = sizeof(ULONG);
```

```
    p->af_PointerToInfo = (PVOID*) &free_map;
    return CR_SUCCESS;
    }                       // ARB_QUERY_FREE
```

爲了使用傳回的資料，最後的呼叫者必須知道仲裁器所使用之內部資料的結構。除了 DMA 仲裁器（原始碼由 DDK 提供）以外，Microsoft 並沒有公佈標準仲裁器所使用的格式。因此，這項機能主要使用在私有資源上面。

ARB_REMOVE 機能　當 Configuration Manager 取消仲裁器之登錄，會送出 *ARB_REMOVE* 訊息。對於一個「連結特定 DEVNODE」的 local 仲裁器而言，當 DEVNODE 消失，即自動取消登錄。但對於一個 global 仲裁器而言，取消登錄的動作只有在 VxD（可能是第一個登錄此仲裁器之 VxD）呼叫 *CM_Deregister_Arbitrator* 時才會發生。總之，仲裁器的工作是要釋放被使用過的任何一塊記憶體或其它資源。在 telepathic channel 仲裁器之中，並沒有什麼是需要做的：

```
case ARB_REMOVE:            // af == 5
    return CR_SUCCESS;
```

硬體設定檔（Hardware Profiles）

Windows 95 允許使用者爲電腦定義多重組態。多重組態對擁有 docking（可駐泊）硬體的膝上型電腦可能相當有用，因爲它們允許系統自動設定本身，以適合駐泊於任何工作站（docking station）上。桌上型系統的使用者或許也可以找到此一構想的某些理由。

高雲慶註：此處對於膝上型電腦之描述，似乎較適合應用於筆記型電腦。docking station 目前多用於筆記型電腦，使筆記型電腦在辦公室中可連結外接之顯示幕、鍵盤、或其他設備（如網路設備）。

Windows 95 Setup 程式一開始以 "Original Configuration" 來建立硬體設定檔。如果你將你的膝上型電腦駐泊（dock）到新的工作站上，Windows 95 會察覺到並自動展開完整的

硬體偵測以便發現新環境的特性。你也可以複製一份原始設定檔，用以建立新的硬體設定檔，然後更改其中的 devices 設定。例如，你可能希望 telepathic I/O stream device 只在 "Intellectual Configuration" 中才可使用，那麼首先你必須在【控制台（Control Panel）】的【系統（System）】圖示中使用【硬體設定檔（Hardware Profiles）】附頁（tag），建立新的設定檔，接著在【系統】圖示中使用【裝置管理員（Device Manager）】附頁（tag）來顯示 device 的屬性表，在那裡你可以從 "Original Configuration" 中移除 device（圖 **12-5**）。

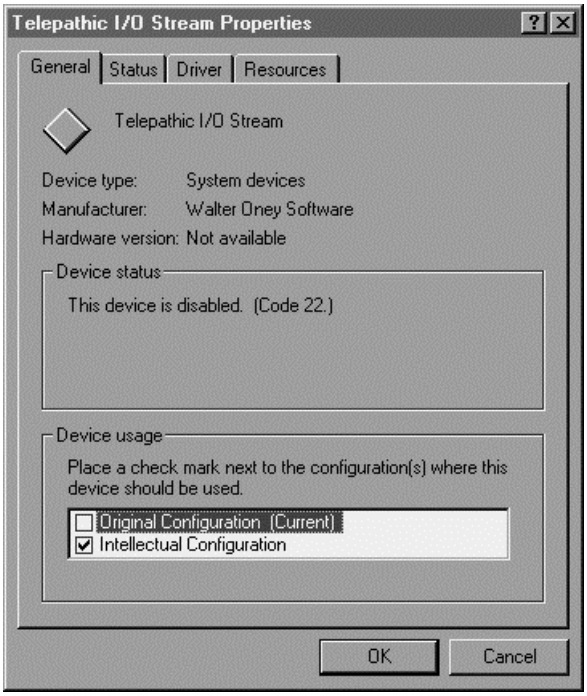


圖 12-5　從硬體設定檔（hardware profile）中移除 device

新的硬體設定檔的存在，會造成電腦啓動時的一些改變。如前面所說，通常定義新的設定檔，是爲了要支援一種新的 docking mode。如果你的電腦有 Plug and Play BIOS，它會詢問硬體以計算一個 docking ID。MS-DOS 會搜尋 Windows 95 registry 以選擇包含此

docking ID 的硬體設定檔（見圖 **12-6**）。如果它在 registry 中發現，合適的設定檔不止一個，它會準備適當的選單供你選擇。甚至即使沒有 Plug and Play BIOS，同樣的情形還是會發生（只不過你不再總是能夠獲得 configuration 選單），因爲 MS-DOS 總是會發現相同的預設 docking ID：

```
Windows cannot determine what configuration you computer is in.
Select one of the following:

1. Original Configuration
2. Intellectual Configuration
3. None of the above
```

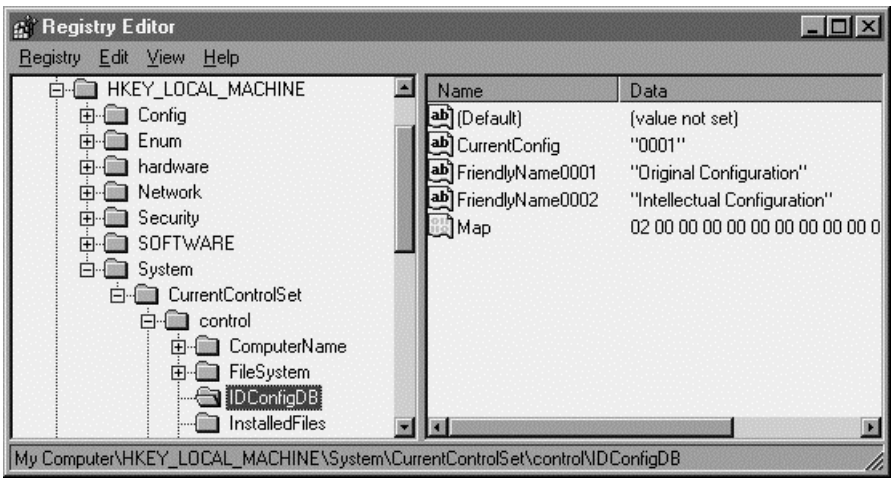


圖 12-6　registry 中的組態資料庫（configuration database）

選擇了一個組態後，MS-DOS 會從 registry 中取出硬體設定檔的名稱。本例之中，名稱不是 "Original Configuration" 就是 "Intellectual Configuration"。現在，MS-DOS 搜尋你的 CONFIG.SYS，看看能否在 [menu] section 中找到擁有相同名稱的 menuitem 項目。如果找到，MS-DOS 便自動選擇 CONFIG.SYS 的那個分支。如果你有一個 [menu] section，但是沒有吻合的 menuitem 項目，MS-DOS 會給你標準的 CONFIG 選單，讓你選擇。這種選單只有 MS-DOS version 6 或更後面的版本才有。決定了適當的組態以後，MS-DOS 會執行所選分支中的 CONFIG.SYS 命令，並且也會將組態的簡稱設爲 "CONFIG"

環境變數。例如，你的 CONFIG.SYS 檔有以下內容：

```
[menu]
menuitem=original, Original Configuration
menuitem=smart, Intellectual Configuration

[original]
...(normal configuration commands)

[smart]
include=original
...(special configuration commands)
```

接下來你的 AUTOEXEC.BAT 可能會依據 CONFIG 環境變數是 ORIGINAL 還是 SMART 來做進一步決定。

當 Windows 95 啓動，CONFIGMG.VXD 裡頭的真實模式初始化常式（real-mode initializer）送出 IOCTL request 給真實模式的 CONFIG$ device，以便得知是哪一個硬體設定檔生效。它再將這份資料傳遞給 Configuration Manager 中的保護模式初始化常式（protected-mode initialization routines）當作參考資料。Configuration Manager 然後就自動將任何不存在於目前設定檔中的 devices 給 disable 掉。

如果所有的設定程序都吻合，使用者並不會注意到這個過程。BIOS 確認 docking station，然後 MS-DOS 找出符合 docking ID 的硬體設定檔，並選擇 CONFIG.SYS 中正確的區塊，接著再由 AUTOEXEC.BAT 執行所有正確的命令。Configuration Manager 絕不會 "enables" 不存在的 devices，Device Manager 則指出電腦目前的組態。此機制唯一不方便的地方是，device 屬性表中的【一般（General）】附頁內的【裝置狀態（Device Status）】所顯示的說明不易讓人明白："This device is disabled. (Code 22)"（譯註：中文 Windows 95 所顯示的是：這個裝置已經停止使用（code 22））。你必須有足夠的知識，才能明瞭：它是因爲不在目前的硬體設定檔中，所以才被 disabled。

與使用者之間的介面

Windows 95 內含一個 device 安裝程式(【新增硬體精靈】)和一個 Device Manager（【裝置管理員】），兩者皆可在 Windows 95 的【控制台】中找到。這些元件提供了一致的介面給使用者，使他們比較不會對硬體感到困惑。device drivers 的設計者必須多少知道一些一般應用程式的概念，才能發展出這些 UI 介面。這一節我將說明其中兩個因素，並從「如何訂製 device properties 畫面」開始。

Property Page Providers

Windows 95 的【裝置管理員】可在【控制台】的【系統】圖示中找到。你可以使用【裝置管理員】來檢查系統的組態。當使用者從【控制台/系統/裝置管理員】附頁（tab）中選擇【內容（Property）】按鍵時，【裝置管理員】提供兩種方法，讓你能夠更換或新增 properties pages，並在其中放置要供應給使用者的資訊。第一種方法是爲 device 準備一個 custom property sheet provider。第二種方法是針對 device 所屬的 device class，呼叫其 class installer 的進入點（entry points）。通常這是在沒有 custom provider 下的應急辦法。我將在這裡討論第一種方法，也是最簡單的一種方法。

【裝置管理員】的 property sheet provider 是一個 16 位元動態連結函式庫（DLL）。在這種最簡單的形式下，provider 匯出（expert）一個名爲 *EnumPropPages* 的函式。它並且內含一個 property page resource，以及管理附頁所需的 dialog 函式。在顯示 property sheet 以前，【裝置管理員】先呼叫 *EnumPropPages* 函式，讓你有機會增加自己的 pages 上去，或是更換標準的 pages。接下來，Windows 95 的 property sheet manager 以平常方式呼叫你的 dialog 函式。下面是一個 *EnumPropPages* 函式例，用於本章所處理的 telepathic I/O stream device（可在書附光碟的 \CHAP11\SCHOOLBUS 目錄中找到，是 SCHOOLUI.C 的一部份）：

```
#0001  #include <windows.h>
#0002  #include <commctrl.h>
#0003  #include <setupx.h>
```

```
#0004
#0005  #include "resource.h"
#0006
#0007  BOOL WINAPI EXPORT StatusDlgProc(HWND, UINT, WPARAM, LPARAM);
#0008
#0009  BOOL WINAPI EXPORT EnumPropPages(LPDEVICE_INFO pdi,
#0010     LPFNADDPROPSHEETPAGE AddPage, LPARAM lParam)
#0011     {                        // EnumPropPages
#0012     PROPSHEETPAGE status;       // status property page
#0013     HPROPSHEETPAGE hstatus;
#0014
#0015     status.dwSize = sizeof(PROPSHEETPAGE);
#0016     status.dwFlags = PSP_USETITLE;
#0017     _asm mov status.hInstance, ds
#0018     status.pszTemplate = MAKEINTRESOURCE(IDD_STATUS);
#0019     status.hIcon = NULL;
#0020     status.pszTitle = "Status";
#0021     status.pfnDlgProc = StatusDlgProc;
#0022     status.lParam = (LPARAM) pdi->dnDevnode;
#0023     status.pfnCallback = NULL;
#0024
#0025     hstatus = CreatePropertySheetPage(&status);
#0026     if (!hstatus)
#0027        return TRUE;  // display property sheet even if we fail
#0028
#0029     if (!AddPage(hstatus, lParam))
#0030        DestroyPropertySheetPage(hstatus);
#0031     return TRUE;
#0032     }                        // EnumPropPages
```

其中 WINDOWS.H、COMMCTRL.H、SETUPX.H 表頭檔是 DDK 的一部份；你可以在 INC16 子目錄中找到它們。SETUPX.H 也內含 PRSHT.H，其中定義了 property sheet interface，以及 DEVICE_INFO 結構。

這個函式除了兩個小地方之外，對有經驗的 Windows 程式員應該沒有什麼困難。首先，當使用者第一次看到我們的 custom page 時，Windows 95 會送給我們的 dialog 函式一個 *WM_INITDIALOG* 訊息，其 *lParam* 爲一個 property page 結構副本的位址。*DEVICE_INFO* 結構中的 *dnDevnode* 欄位是我們正欲獲知的 DEVNODE 的 flat 位址。將 *status.lParam* 欄位設爲 *DEVICE_INFO* 結構中的 DEVNODE 位址，可以使 dialog 函式知道 DEVNODE 位址。第二個值得注意的是，*EnumPropPages* 函式裡的

AddPage 參數是一個函式指標，指向一個可以將我們的 custom page 加到 property sheet 中的函式。當我們呼叫 *AddPage* 函式時，必須將所收到的 *lParam* 參數毫無改變的傳給它。如果 *AddPage* 成功地爲我們加上 page，便會傳回 TRUE。

你的 provider 可以經由保護模式 API 來呼叫 Configuration Manager。在 16 位元程式中含入 CONFIGMG.H，便可以呼叫你原本在 VxD 中才能呼叫的函式（譯註：這裡指的是 *CM_Get_First_Log_Conf* 函式及 *CM_Get_Res_Des_Data* 函式）。爲了達到這個目的，你需要以一個特別的方式來含入 CONFIGMG.H：

```
#define Not_VxD
#define <vmm.h>
#define MIDL_PASS
#define <configmg.h>
```

第一行定義 *Not_VxD*，如此便會忽略掉 VMM.H 及 CONFIGMG.H 之中「只有 VxD 程式員才會有興趣」的宣告。定義 *MIDL_PASS* 則可以避免 16 位元編譯器對 #pragma pack(push) 敘述句產生警告訊息。

下面這個例子可從 Configuration Manager 裡取出資料。設計這個 dialog 函式是爲了一個簡單的 custom page（**圖 12-7**）：

```
#0001  BOOL WINAPI EXPORT StatusDlgProc(HWND hdlg, UINT msg,
#0002     WPARAM wParam, LPARAM lParam)
#0003     {                        // StatusDlgProc
#0004     switch (msg)
#0005        {                     // process message
#0006
#0007     case WM_INITDIALOG:
#0008        {                     // WM_INITDIALOG
#0009
#0010        #define ResType_Telepath ((0x10 << 5) | 5)
#0011
#0012        typedef struct
#0013           {
#0014           ULONG allocated;
#0015           ULONG requested;
#0016           } TELEPATH_RESOURCE;
```

```
#0017
#0018         LOG_CONF logconf;
#0019         RES_DES hres;
#0020         DEVNODE devnode = (DEVNODE)
#0021            ((LPPROPSHEETPAGE) lParam)->lParam;
#0022
#0023         if (CM_Get_First_Log_Conf(&logconf, devnode,
#0024                ALLOC_LOG_CONF) == CR_SUCCESS
#0025             && CM_Get_Next_Res_Des(&hres, (RES_DES) logconf,
#0026                ResType_Telepath, NULL, 0) == CR_SUCCESS)
#0027             {                 // has telepath channel
#0028             TELEPATH_RESOURCE res;
#0029             int channel;
#0030             DWORD mask;
#0031
#0032             CM_Get_Res_Des_Data(hres, &res, sizeof(res), 0);
#0033             if (res.allocated >= 0)
#0034                SetDlgItemInt(hdlg, IDC_CHANNEL, res.allocated,
#0035                   FALSE);
#0036             }                 // has telepath channel
#0037         break;
#0038         }                     // WM_INITDIALOG
#0039
#0040         }                     // process message
#0041      return FALSE;
#0042      }                        // StatusDlgProc
```

這個 dialog 函式只處理 *WM_INITDIALOG*。它取得先前提過的 telepathic device 的 DEVNODE 位址，然後使用 Configuration Manager call，找出所配置的 logical configuration 的位置，及其中的 telepathic channel resource descriptor。雖然我們可以使用 DEVNODE 的 ring0 flat address 當作 *CM_Get_First_Log_Conf* 函式參數，但是我們並沒有使用從 *CM_Get_Next_Res_Des* 取回的 telepathic resource descriptor 的 ring0 flat address。那是 *CM_Get_Res_Des_Data* 應該處理的工作。這個函式只能由 ring3 程式使用，並且像本例這樣，希望取得一份 Configuration Manager 的 resource descriptor 副本。一旦我們擁有自己的一份 telepathic resource descriptor，便可以使用它的 *allocated* 欄位來設定 property page 中的適當文字。

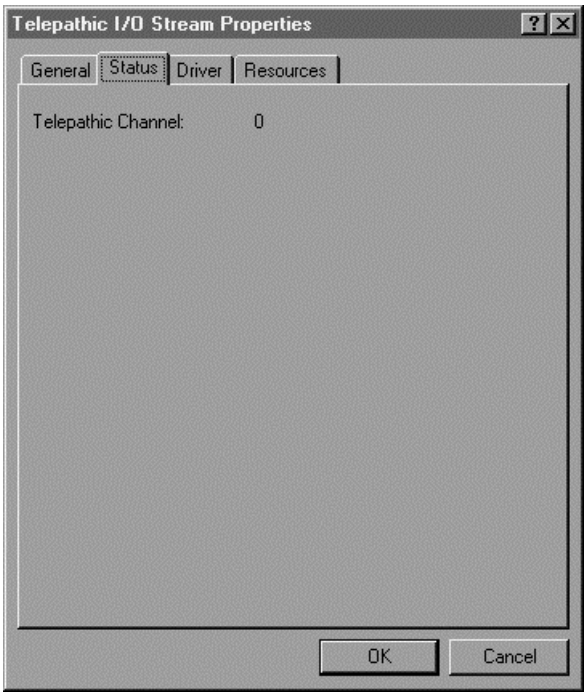


圖 12-7 一個最簡單的 custom property page

處理私有資源（Handling Private Resources）

剛才所說明的例子，對使用者而言並不是很理想。因爲我定義了一個 private telepathic resource，爲了讓使用者更方便，我應該在 property sheet 的 Resource page 中將 channel 連同其他資源一併列出。你可以一開始就加上內含所有資源資料的 property page，然後在 *DEVICE_INFO* 結構中設立 *DI_RESOURCEPAGE_ADDED* flag，以完成此事：

```
pdi->Flags |= DI_RESOURCEPAGE_ADDED;
```

（你可以設立 *DI_GENERALPAGE_ADDED* flag 來取代 General page）

然而，如果你取代掉 Resource page，很不幸的你也必須複製 Resource page 中的所有機能。這是一件痛苦的工作，因爲其中包括處理組態設定表（configuration profile list），提出資源衝突報告，著手重設組態等等工作。而且，用以顯示資源資料的 list box，又是一

個大家比較陌生的 owner-draw 選單。

WM_DEVICECHANGE 訊息

除了讓你有能力訂製一部份與 device 有關的使用者介面外，Windows 95 也爲應用軟體設計者提供一個 *WM_DEVICECHANGE* 訊息。這個訊息如同 *WM_WININICHANGE* 或 *WM_SYSCOLORCHANGE* 一樣，當系統層次的改變發生時，Windows 將這個訊息傳送給所有上層（top level）視窗，因爲它可能會引起某些程式的興趣。像其它的 broadcast 訊息一樣，大部份程式會忽略這個訊息並交給預設的處理程序。在我說明如何於程式中反應這個訊息之後，我們就準備結束 Plug and Play 這個主題了。

WM_DEVICECHANGE 訊息的 *wParam* 參數表示傳送此訊息的原因，*lParam* 參數則內含因 *wParam*（表 **12-5**）不同而不同的資料（通常是一個指標）。表格裡的符號可在 DBT.H 找到，其中也定義了一些與此訊息之處理有關的結構及常數。

wParam 數值	內容
DBT_CONFIGCHANGED	表示目前組態已經改變
DBT_DEVNODESCHANGED	表示 hardware tree 已經改變
DBT_DEVICEARRIVAL	表示已經安裝新的 device 或已經安插新的媒體
DBT_DEVICEQUERYREMOVE	查詢是否 device 的移除沒有問題
DBT_DEVICEQUERYREMOVEFAILED	表示上一次 QUERYREMOVE 失敗
DBT_DEVICEREMOVEPENDING	表示 device 正在移除
DBT_DEVICEREMOVECOMPLETE	表示 device 已經移除
DBT_DEVICETYPESPECIFIC	表示特定 device 的 event 已經發生

表 12-5　WM_DEVICECHANGE 的參數

對於所有的 *DBT_DEVICE* 訊息，*lParam* 都指向數種 self-identifying 結構中的一個。每一種可能的結構，其最初 12 個 bytes 相同，所以你可將 *lParam* 做型別轉換，使其成

爲一個指向 *_DEV_BROADCAST_HEADER* 結構的指標（**12-8**）。結構中的 *dbcd_devicetype* 欄位表示緊跟在固定表頭之後的額外資料。以下數小節說明這個欄位的可能值。

```
struct _DEV_BROADCAST_HEADER
   {
   DWORD dbcd_size;           // 00 整個結構的大小
   DWORD dbcd_devicetype;     // 04 資料結構真正的類別
   DWORD dbcd_reserved;       // 08 保留給未來使用
   };                         // 0C
```

圖 12-8　DBT_DEVICE message 資料結構的表頭

DBT_DEVTYP_OEM　如果數值等於 *DBT_DEVTYP_OEM*，則 *lParam* 指向一個 *DEV_BROADCAST_OEM* 結構：

```
typedef struct _DEV_BROADCAST_OEM
   {
   DWORD      dbco_size;
   DWORD      dbco_devicetype;
   DWORD      dbco_reserved;
   DWORD      dbco_identifier;
   DWORD      dbco_suppfunc;
   } DEV_BROADCAST_OEM, DBTFAR* PDEV_BROADCAST_OEM;
```

其中 *dbco_identifier* 和 *dbco_suppfunc* 的數值取決於「誰送出訊息」以及「送出的原因」。

DBT_DEVTYP_DEVNODE　如果數值等於 *DBT_DEVTYP_DEVNODE*，則 *lParam* 指向一個 *DEV_BROADCAST_DEVNODE* 結構：

```
typedef struct _DEV_BROADCAST_DEVNODE
   {
   DWORD      dbcd_size;
   DWORD      dbcd_devicetype;
   DWORD      dbcd_reserved;
   DWORD      dbcd_devnode;
   } DEV_BROADCAST_DEVNODE, DBTFAR *PDEV_BROADCAST_DEVNODE;
```

其中 *devnode* 表示訊息傳送給 device node 的 ring0 線性位址（linear address）。例如，*DBT_DEVICEARRIVAL* 訊息表示一個新的 DEVNODE 已誕生。

如你所知，Configuration Manager 會給你一個 DEVNODE 的全部資訊。爲了在 16 位元程式中呼叫 Configuration Manager，你可以使用稍早提過的 custom property page providers 技術。也就是說，#define *Not_VxD* 並含入兩個 VMM.H 及 CONFIGMG.H，然後就可以呼叫諸如 *CM_Get_Alloc_Log_Conf* 等函式，就像在 VxD 中的作爲一樣。

高雲慶註：Windows 95 的 Configuration Manager 及 Driver Installer 皆爲 16 位元程式。若你要在 32 位元程式中使用它們，必須自行撰寫 thunk DLL。

奇怪的是，在 Windows 95 裡頭，你無法在一個 Win32 程式中直接接觸 Configuration Manager。未來的版本應該可以。爲了更瞭解某個 DEVNODE 指標所表示的意義，你可以查詢 registry 內所謂的 dynamic key：

```
#0001  #define tp ((struct _DEV_BROADCAST_DEVNODE*) lParam)
#0002  DWORD devnode = tp->dbcd_devnode;
#0003  HKEY devkey;
#0004  char keyname[256];
#0005  DWORD lkeyname;
#0006
#0007  _snprintf(keyname, sizeof(keyname)),
#0008      "Config Manager\\Enum\\%8.8lX", devnode);
#0009  RegOpenKeyEx(HKEY_DYN_DATA, keyname, 0, KEY_READ, &devkey);
#0010  lkeyname = sizeof(keyname);
#0011  RegQueryValueEx(devkey, "HardwareKey", NULL, NULL, keyname,
#0012      &lkeyname);
#0013  ...
#0014  if (devkey)
#0015      RegCloseKey(devkey);
#0016  #undef tp
```

這段碼係根據文件中記載的一個事實：registry 的 HKEY_DYN_DATA 分支包含 Config Manager\Enum 分支，其中的每一筆記錄（entry）代表 hardware tree 中的一個 DEVNODE。一個 DEVNODE 的 named value "HardwareKey" 代表一個 registry 路徑，

記錄著如何從 HKLM\Enum 到達 device hardware key（在這裡你可以發現其它資料）。然而，如果沒有獲得未公開的 control block 內容知識而想知道 device 的目前資源分配情況，是不可能的。爲了這個原因，Microsoft 並不鼓勵程式處理那種「*dbcd_devicetype* 被設定爲 *DBT_DEVTYP_DEVNODE*」的訊息。無論如何，這不會發生在 Windows NT 之中，而這也正是不要依賴它們的另一個原因。

DBT_DEVTYP_VOLUME 如果 *dbcd_devicetype* 欄位的值是 *DBT_DEVTYP_VOLUME*，那麼 *lParam* 將指向一個 *DEV_BROADCAST_VOLUME* 結構：

```
typedef struct _DEV_BROADCAST_VOLUME
    {
    DWORD       dbcv_size;
    DWORD       dbcv_devicetype;
    DWORD       dbcv_reserved;
    DWORD       dbcv_unitmask;
    DWORD       dbcv_flags;
    } DEV_BROADCAST_VOLUME, DBTFAR *PDEV_BROADCAST_VOLUME;
```

其中 *dbcv_unitmask* 是一個 bit mask，表示這個訊息與哪一個磁碟裝置有關；位元 0 表示磁碟機 A，依此類推。*dbcv_flags* 可以是 *DBTF_MEDIA*（表示這項改變影響的是磁碟機的媒體，而不是磁碟機本身）也可以是 *DBTF_NET*（表示磁碟機是一個網路磁碟機）。

例如，在我的磁碟機 D 中插入一片新的 CD-ROM 光碟，會產生一個 *DBT_DEVICEARRIVAL* 訊息以及一個 *DBT_DEVTYP_VOLUME* subtype、一個 *DBTF_MEDIA* flag，以及一個 00000008h 的 unit mask。

按下 CD-ROM 光碟機的退出鍵，卻不會產生任何有關於「是否可以移除光碟」的詢問訊息（譯註：指 *DBT_DEVICEQUERYREMOVE* 訊息）。我對於這一點感到失望。我只得到 *DBT_DEVICEREMOVECOMPLETE* 訊息，告訴我說光碟已經移除。對於軟式磁碟機，你甚至不會收到這兩個訊息，因爲硬體本身並沒有能力通知你磁片更換了，只能藉由不斷的查詢（polling）來偵測。換句話說，*WM_DEVICECHANGE* 訊息無法幫你以較聰明的方法來處理「使用者正打算移除使用中的光碟」的情況。你所能夠做的最佳辦法是，首先偵測到你正在處理的檔案係位於光碟片中，然後使用 21/440D 媒體上鎖（media

locking）函式（譯註：21/440D 是指 INT 21h / AX=440Dh 之 IOCOL 中斷呼叫），阻止使用者移除光碟。21/440D media locking 函式在 ***Programmer's Guide to Microsoft Windows 95***（Microsoft Press，1995）裡頭有說明。噢，當然，這個介面只適用於 16 位元呼叫者。

DBT_DEVTYP_PORT　如果 *dbcd_devicetype* 欄位等於 *DBT_DEVTYP_PORT*，則 *lParam* 指向一個 *DEV_BROADCAST_PORT* 結構：

```
typedef struct _DEV_BROADCAST_PORT
    {
    DWORD       dbcp_size;
    DWORD       dbcp_devicetype;
    DWORD       dbcp_reserved;
    DWORD       dbcp_name[1];
    } DEV_BROADCAST_PORT, DBTFAR *PDEV_BROADCAST_PORT;
```

其中 *dbcp_name* 是一個以 null 爲結尾的字串，內含容易瞭解的名字，像是 Communications Port（COM3）等等，表示與此訊息有關的通訊埠。舉個例子，如果你使用【控制台】增加一個虛構的通訊埠，你會收到兩個 *DBT_DEVICEARRIVAL* 訊息：*DBT_DEVTYP_PORT* 及 *DBT_DEVTYP_DEVNODE*。

我發現當我新增並刪除 School Bus 及其 telepathic device 時，監視所發生的 *WM_DEVICECHANGE* 訊息，頗富啓發性，並且其結果有點令人驚訝（監視軟體可以從書附光碟的 \CHAP12\WM_DEVICECHANGE 目錄中取得）。**圖 12-9** 顯示當我新增此一 bus 時所發生的訊息，**圖 12-10** 顯示當我刪除此一 bus 時所發生的訊息。如果你留意**圖 12-9**，你會看到有些 DEVNODEs 似乎什麼都沒做 -- 在 School Bus 產生以及毀滅的時候。爲了驗證新 device 所需的資源的確是備妥可用的，Windows 95 會建立一個暫時性 DEVNODE。除此之外，重新處理這個 DEVNODE 時，Windows 95 會對整個系統做重新列舉（re-enumeration）的動作。ISAPNP 驅動程式總是重建 read-data port DENODE，而 Virtual Display Device 總是移除並重建 MONITOR DEVNODE。

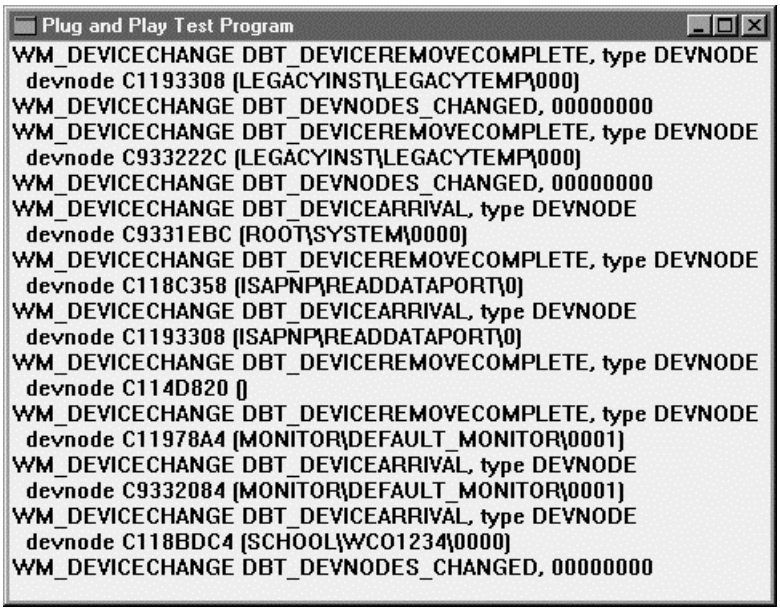


圖 12-9　當一個 device 被新增到系統上，出現的 WM_DEVICECHANGE 訊息

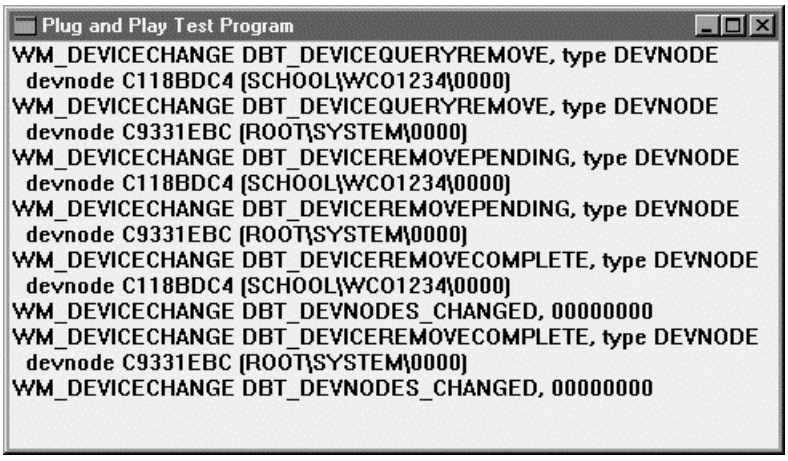


圖 12-10　當一個 device 被移除，所出現的 WM_DEVICECHANGE 訊息

第 13 章

Input/Output 程式設計

譯註：
EOI：end-of-interrupt command
IDT：interrupt descriptor table IMR：interrupt mask register
IRQ：interrupt request
IRR：interrupt request register
ISR：interrupt service routines
PIC：programmable interrupt controller

前兩章的 Plug and Play 說明 Windows 95 Configuration Manager 如何將 I/O 資源分配給 devices 和 drivers。這一章將告訴大家如何在 Windows 95 driver 中實際使用這些資源。你將會學到如何使用「直接映射至 device」之記憶體、如何控制 device 之「I/O ports 存取」、如何處理硬體中斷、如何對一個 device 使用 DMA 傳輸。

最初，Windows VMM 的主要目的是爲了在多個 VMs 的情況下將硬體虛擬化。早期 Windows 3.0 時代，真實模式驅動程式對於 enhanced mode 非常重要，因此 VMM 給予「真實模式驅動程式之 I/O」以極大的優先權，只有在需要虛擬化動作時才加以干涉。慢慢地，真實模式驅動程式變得愈來愈不重要，並且終將完全消失。但是要瞭解如何撰寫目前當道的 VxD，你還是必須瞭解 IBM PC/AT I/O 架構，以及真實模式驅動程式如何在上面做動作。

在討論完大部份 I/O 程式設計之下的硬體特性後，我會說明如何撰寫 VxD 來虛擬化這些特性，好讓多個 VMs 以爲它們對硬體擁有獨佔使用權。然而，如果你希望跟著 Microsoft 前進，你應該考慮撰寫可以在 ring0 完整處理硬體的那種 VxD，而不是爲了 ring3 驅動程式而對硬體做虛擬化。因此，本章最後一個主題安排的便是 ring0 驅動程式的撰寫。

硬體基礎

在 PC/AT 相容電腦上的 I/O device 程式設計，牽涉到對 mapped memory 與 I/O port 的存取，硬體中斷 requests（IRQs），以及所謂的直接記憶體存取（DMA）。Intel CPU 及一些標準化的輔助晶片互相作用以完成這些工作。這一節我要探討驅動程式使用輔助晶片及 CPU 特性的標準方法。

Mapped Memory

雖然 DMA 意味 "直接記憶體存取"，某些 devices 甚至使用一種更直接的記憶體存取方法：它們將記憶體直接映射（map）到 CPU 位址空間中。於是軟體可以經由讀寫 mapped memory 來存取硬體，就如同讀寫一般資料一樣。此方法最常見的例子就是 Video Graphics Array（VGA）顯示幕的文字模式（text mode）。起始於實際位址 B8000h 的一些記憶體，控制著出現於顯示幕上的字元。因此，下面這段簡單的 DEBUG 指令：

```
C:\>debug
-e b800:0 41 20
-
```

可以把黑色字母 A 放在螢幕的左上角，且以綠色爲背景（譯註：原書爲藍綠色，錯誤）。測試這個例子時，請確定在執行 DEBUG 以前先清除螢幕畫面，否則測試結果可能會在被你看見以前就被捲出螢幕之外。

I/O Ports

CPU 除了實作出一個記憶體位址空間，又實作了一個 I/O 位址空間。I/O 位址空間裡的位置叫做 ports。IN/OUT 指令及其數個變化，就是用來存取 I/O ports（而非記憶體）。就像面對真正的記憶體那樣，port 的存取單位可能是 1 byte, 2 bytes 或 4 bytes。此外，軟體可以使用有（或沒有）REP 爲前導的 INS 及 OUTS 指令，以便在 port 資料轉移之後自動調整位址暫存器及計數暫存器。

I/O port 位址的寬度爲 16 位元。然而一些標準的設計特性，導致你不能夠連接 65,536 個可定址的 devices 到實際的 PC/AT 相容電腦上。標準架構保留了 port 位址 00h~FFh 給某些標準 devices（表 **13-1**）使用。Industry Standard Architecture（ISA）bus 只對一個 I/O 位址解碼其最後 10 位元，使得可用的 port 位址數減至 768（2^{10} 減去 256 個保留位址）。然而，其它的硬體 bus 可以對完整的 16 位元 I/O 位址解碼。除此之外，內含 PCI bus 的系統，通常擁有 bridge chip（橋接器晶片），連接 PCI bus 及 ISA bus。橋接器（bridge）使 ISA bus 只看到 I/O 位址 0FFh~3FFh。因此直接連到 PCI bus 的 devices 可以使用 16 位元的 port 位址，但其最後 10 位元可能在其它地方與某個 ISA device 衝突。

> 張訓賓（本章技術檢閱）註：新的 PNP（plug and play） 規格要求，ISA 必須對 16 個位元完全解碼（decode），以解決此種 IO port 衝突問題。

Port 位址（16 進制）	說明
00	DMA channel 0，address 暫存器
01	DMA channel 0，count 暫存器
02	DMA channel 1，address 暫存器
03	DMA channel 1，count 暫存器
04	DMA channel 2，address 暫存器
05	DMA channel 2，count 暫存器

Port 位址（16 進制）	說明
06	DMA channel 3，address 暫存器
07	DMA channel 3，count 暫存器
08	DMA status 暫存器，for channel 0~3
0A	DMA mask 暫存器，for channel 0~3
0B	DMA mode 暫存器，for channel 0~3
0C	DMA clear-byte 指標
0D	DMA master clear byte
0E	DMA clear-mask 暫存器，for channel 0~3
0F	DMA write-mask 暫存器，for channel 0~3
18	DMA extended function 暫存器
1A	DMA extended function execute 暫存器
20	In-service 暫存器，for master PIC
21	Master PIC 的 command and mask 暫存器
40	Interval timer counter 0
42	Interval timer counter 2
43	control 暫存器，for interval timer counters 0, 2
44	Interval timer counter 3
47	control 暫存器，for interval timer counter 3
60	鍵盤控制器（keyboard controller）
61	System control port B（PS/2）
64	PS/2 週邊控制器的 command and status 暫存器
70	CMOS address 暫存器
71	CMOS data 暫存器
74~75	Extended CMOS address 暫存器
76	Extended CMOS data 暫存器
80	DMA address 暫存器
81	DMA page 暫存器，for channel 2

Port 位址（16 進制）	說明
82	DMA page 暫存器，for channel 3
83	DMA page 暫存器，for channel 1
87	DMA page 暫存器，for channel 0
89	DMA page 暫存器，for channel 6
8A	DMA page 暫存器，for channel 7
8B	DMA page 暫存器，for channel 5
8F	DMA page 暫存器，for channel 4
90	DMA arbitration 暫存器
91	DMA feedback 暫存器
92	System control port A（PS/2）
94	系統主機板 setup enable 暫存器（PS/2）
96	可程式的 option select 暫存器（PS/2）
A0	In-service 暫存器，for slave PIC
A1	Slave PIC 的 command and mask 暫存器
C0	DMA address 暫存器，for channel 4
C2	DMA count 暫存器，for channel 4
C4	DMA address 暫存器，for channel 5
C6	DMA count 暫存器，for channel 5
C8	DMA address 暫存器，for channel 6
CA	DMA count 暫存器，for channel 6
CC	DMA address 暫存器，for channel 7
CE	DMA count 暫存器，for channel 7
D0	DMA status 暫存器，for channel 4~7
D4	DMA mask 暫存器，for channel 4~7
D6	DMA mode 暫存器，for channel 4~7
D8	DMA clear-word 指位器（pointer）
DA	DMA master clear word
DC	DMA clear-mask 暫存器，for channel 4~7
DE	DMA write-mask 暫存器，for channel 4~7
F0~FF	Math coprocessor 暫存器

表 13-1　保留的 I/O port

硬體中斷（Hardware Interrupts）

大部份 devices 使用中斷（interrupt）來通知作業系統說，它們已經完成了一個動作。以這種方法來使用中斷，可以讓 CPU 在 I/O 發生時處理其它的事，藉以改善整個系統的效率。電腦內部包含一個可程式中斷控制器（Programmable Interrupt Controller，PIC）來管理中斷系統。圖 **13-1** 以簡單的方塊圖來說明 PIC。請注意，這張圖顯示 PIC 有 8 條「中斷請求訊號線（interrupt request lines）」，然而你可能已經知道，IRQs 事實上高達 15。目前的電腦裡頭實際上有兩個 PICs，每一個處理 8 個輸入訊號線。有關 PIC 更多的資料，可參閱 Intel 的 *Peripheral Components*（Intel Corporation，1993）及 Hans-Peter Messmer 的 *Indispensible PC Hardware Book; Your Hardware Questions Answered*（Addison-Wesley，1994）。

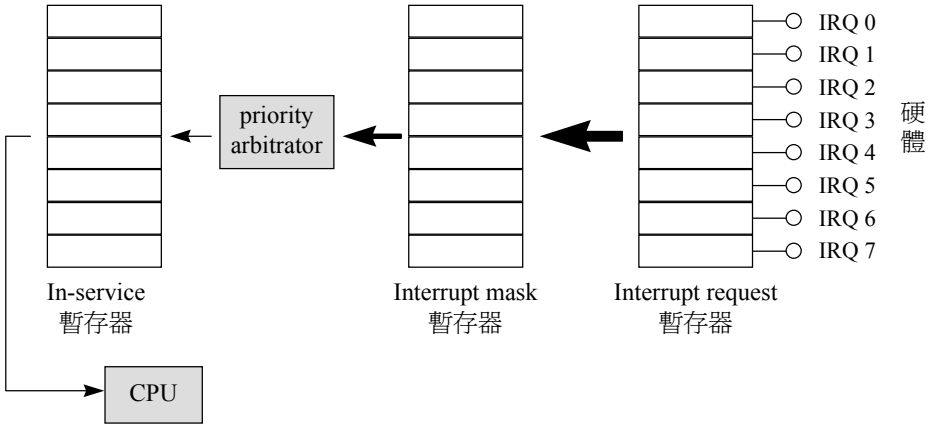


圖 13-1　可程式中斷控制器（PIC）架構

每一個「可產生中斷」的硬體裝置，對 PIC 的 IRQ 訊號線都有一個邏輯上的連接。在所有 PC/AT 相容電腦裡，某些 IRQs 被固定分配給一些標準的 device（表 **13-2**）。當 device 需要中斷 CPU 時，它會對被指定給它的 IRQ 訊號線發出信號，藉以在 PIC 的 **interrupt request register**（**IRR**）裡將對應的位元設立起來。IRR 對每個「目前想要中斷

的 IRQ」，會將該 IRQ 在 IRR 中的對應位元設爲 1，對每個「目前不想要中斷的 IRQ」，則將 IRR 中的對應位元設爲 0。PIC 會參考 **interrupt mask register**（**IMR**），決定是否允許該中斷發生。如果 IMR 中對應的位元是 1，表示對應的中斷是 disabled 或 "masked"；如果對應的位元是 0，表示中斷是 enabled 或 "unmasked"。

IRQ	標準分配
0	系統計時器（Interval timer）
1	鍵盤
2	概括表示來自於 salve PIC 的 IRQs 8~15（請看內文說明）
3	COM2
4	COM1
5	LPT2（如果安裝的話），通常可供自由使用
6	軟式磁碟機控制器
7	LPT1
8	Real-time clock
13	數學輔助運算器（Math coprocessor）錯誤訊號
14	硬式磁碟機控制器

表 13-2　IRQ 標準分配

如果 IMR 指出某個 IRQ 是 "unmasked" 狀態，PIC 會試圖中斷 CPU，然而只有當 CPU flag 暫存器的中斷旗標（interrupt flag）爲 1，中斷才得以發生。如果中斷得以發生，PIC 會在目前要求服務的所有 "unmasked" 中斷之間選擇最高優先權之 IRQ，並在 in-service register 中設立對應之位元時，同時清除該 IRQ 在 interrupt request register（IRR）的對應位元。PIC 也會引發 CPU 產生某個中斷號碼，此號碼是由 IRQ 編號加上內部的中斷基底位址計算而得。

我曾提過，一台電腦裡有兩個中斷控制器。第一個 PIC，也就是 **master PIC**，爲 interrupt requests 0~7 提供服務，並導引至標準真實模式系統之中斷向量 08h~0Fh。第二個 PIC，

也就是 **slave PIC**，爲 interrupt requests 8~15 提供服務，並導引至中斷向量 70h~77h。但由於 CPU 只有一條中斷訊號線（interrupt line），所以 slave PIC 連接到 master PIC 的 IRQ 2 訊號線（而不是連接到 CPU）。由於 master PIC 可以察覺 IRQ 2，中斷 IRQs 8~15 於焉可以發生。

標準的真實模式 BIOS 以所謂的 "fully nested mode" 來規劃 PICs。在這個模式中，每個 IRQ 有絕對的優先權，阻斷同一個 PIC 上較高號碼的 IRQs。因爲 slave PIC 連接到 master PIC 的 IRQ 2 訊號線，所以 interrupt requests 的優先順序從最高至最低排行爲 0~1，8~15，3~7。舉個例子，當鍵盤中斷（IRQ 1）的服務常式正在執行，只有 timer 中斷（IRQ 0）可以發生。沒有任何硬體可以中斷 timer 的中斷服務常式。

用於硬體中斷的中斷服務常式（interrupt service routines，ISRs）一般而言需要處理許多相同的工作：

1. ISR 應該儘快執行 STI 指令以便儘快 "enable" 中斷。在真實模式作業系統裡，CPU 開始執行 ISR 之前，會將中斷 "disable"（CLI），這就是 Intel 處理器的運作方式。在保護模式環境裡，如果 IDT 內含一個針對此中斷之 interrupt gate，那麼 CPU 也會先 "disable" 中斷，再開始執行此 ISR。（作業系統有可能安裝一個 trap gate，使得 ISR 在 "interrupt enabled" 情況下取得控制權。不過系統通常不會對硬體中斷使用 trap gate，因爲硬體 ISRs 之中通常會有一些碼必須在絕不會被中斷的情況下執行）
2. ISR 也應該儘快對 PIC 送出一個 end-of-interrupt（EOI）命令以解除中斷。由於 BIOS 對 PIC 的規劃是完全根據中斷優先權來巢化（nest）中斷，所以大部份真實模式軟體使用一個未特定的 EOI 命令來解除中斷，於是當 EOI 命令執行，就沒有其他中斷會 "pending"（懸而未決）。「Master-PIC 中斷」的 ISR 應該使用 OUT 指令將 byte 20h 寫到 I/O port 20h，以求送出一個 EOI 命令。「Slave-PIC 中斷」的 ISR 則應該將 byte 20h 分別寫到 port 20h 及 A0h，以求送出一個 EOI 命令（應該先對 slave PIC 送出，再對 master PIC 送出）。Windows 使用特定的 EOI 命令來解除某個特定中斷，以確保該中斷的確獲得解除。不論使用哪一種 EOI 命令，它都會「鬆開」PIC，以便較低優先權的中斷可以發生，也使相同來源的中斷能夠再一次發生。

3. 當中斷的處理結束時，ISR 執行一個 IRET 指令，回到原先被中斷的程式。

IRQ 9 的處理　IBM PC/XT 只有一個 PIC。爲了維持軟體的相容性，當 PC/AT 增加第二個 PIC 時，BIOS 做了一件很奇怪的事：送出一個 EOI 命令給 slave PIC 以處理 INT 71，然後發出 INT 0Ah。INT 71 來自於 IRQ 9，而 INT 0Ah 來自於 IRQ 2。由於 IRQ 2 已被用來串接（cascading）master PICs 和 slave PICs，所以它不再能夠讓其他硬體使用。於是過去在 XT 中原本使用 IRQ 2 的硬體，現在被重新導向至 IRQ 9。如果沒有人 hook INT 71h，BIOS 提供了一個預設處理常式。這個預設處理常式首先送出一個 EOI 命令給 master PIC，然後把控制權交到原來的 IRQ 2 ISR 手中。如果你 "hook" INT 71h 以處理 IRQ 9，就有責任送出兩個 EOI 命令。如果你 "hook" INT 0Ah 以處理 IRQ 2，那麼即使硬體真正激發於 IRQ 9，您的程式碼也應該執行無誤。

直接記憶體存取（Direct Memory Access）

DMA 傳輸允許 I/O device 在主記憶體之間移動資料，不需藉由軟體來完成每一次單位資料的傳輸。藉由輪流地 locking 及 unlocking 記憶體，DMA 控制器使得「資料傳輸」與「程式執行」可以同時發生。簡單地說，軟體使用 port I/O operations， 在 DMA 控制器暫存器中儲存起始位址、長度、傳輸方向（to or from 記憶體）。軟體然後將通道（channel）"unmask" 掉，以便啓動傳輸動作。後繼的硬體中斷則是提醒軟體說，傳輸已經完成。DMA 控制器晶片的基本資料可從 Intel 的 *Peripheral Components* 獲得。你可以在 Messmer 的 *Indispensible PC Hardware Book; Your Hardware Questions Answered*，以及 Frank Van Gilluwe 的 *The Undocumented PC*（Addison-Wesley，1994）發現其他有用的討論文字。然而請注意，這兩本書的程式範例有一些錯誤。

圖 13-2 是一個簡化方塊圖，說明 PC 裡頭標準的 DMA 硬體。兩個 DMA 控制器晶片各有 4 個通道，合計共 8 個 DMA 通道。通道 0~3 提供 8-bit 傳輸，通道 5~7 提供 16-bit 傳輸。通道 4 被用來串接（cascade）兩個實際的 DMA 控制器晶片，因此無法被規劃使用。傳統上，通道 0 保留給 DRAM 記憶體所需的 "memory refresh cycles" 驅

動之用，通道 2 保留給軟式磁碟機使用。除了這些被保留的通道以外，兩個 8-bit 通道（1 及 3）以及三個 16-bit 通道（5, 6, 7）提供給擴充介面卡使用。

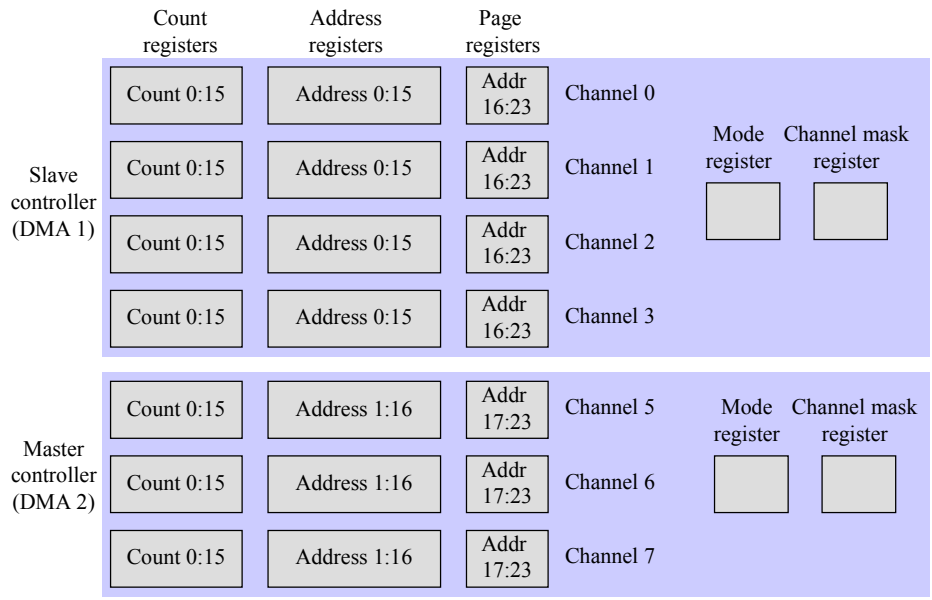


圖 13-2 DMA 硬體架構（通道 4 用來串接 master 控制器與 slave 控制器）

每一個 DMA 通道有一個 16 位元 address 暫存器，一個 8 位元 page 暫存器，以及一個 16 位元 count 暫存器。結合 page 及 address 兩暫存器，可提供最前面之 16MB 實際記憶體存取（張訓賓註：page 與 address 兩暫存器共 24 位元，可定址 2^{24} = 16MB）。每一個 DMA 控制器有一個 mode 暫存器和一個 channel mask 暫存器。此外還有其他一些暫存器，被設計給 BIOS 在系統開機過程中使用；我不會在這裡討論這些暫存器。

爲了傳輸資料，系統軟體必須爲某一硬體設備預備使用的 DMA 通道設定基底位址（base address），資料總數（data count），以及傳輸模式。例如下面的程式片段將 DMA 的通道 2 初始化，以便從軟式磁碟機傳送一個 512-byte sector 資料到記憶體中。（粗體數字表示後繼數頁的討論）

1.

```
                    ; DMA channel mask register
mov al, 06h         ; xxxxx 1 10 => mask channel 2
out 0Ah, al         ;   ..
iodelay             ;   ..
```

2.

```
                    ; DMA mode register:
mov   al, 46h       ; 01 0 0 01 10
out   0Bh, al       ; |  | | |  |
iodelay             ; |  | | |  +-> channel 2
                    ; |  | | +-> write transfer mode
                    ; |  | +-> autoinitialize inactive
                    ; |  +-> increment address
                    ; +-> single transfer mode
```

3.

```
movzx eax, word ptr buffer+2  ; segment portion of buffer address
movzx ecx, word ptr buffer    ; offset portion
shl   eax, 4              ; compute physical address
add   eax, ecx            ;   ..

out   0Ch, al             ; output anything at all to 0C
iodelay                   ;   to reset the even/odd flip-flop
out   04h, al             ; low byte of address
iodelay                   ;   ..
shr   eax, 8              ; high byte of address
out   04h, al             ;   ..
iodelay                   ;   ..
shr   eax, 8              ; page address
out   81h, al             ;   ..
iodelay                   ;   ..
```

4.

```
mov   al, 0FFh            ; set count register to 511
out   05h, al             ;   ..
iodelay                   ;   ..
mov   al, 1               ;   ..
out   05h, al             ;   ..
iodelay                   ;   ..
```

5.

```
mov   al, 02h             ; xxxxx 0 10 => unmask channel 2
out   0Ah, al             ;   ..
```

在此程式片段中，*iodelay* 是一個產生出兩個 short jumps 的巨集，用以卸載（unload）處理器的指令管線（instruction pipeline），以便允許 OUT 指令可以在下一個 OUT 指令執行前完成。變數 buffer 是一個真實模式遠程指標，指向一個資料緩衝區。程式邏輯如下：

1. Port 0Ah 是第一個 DMA 控制器的 channel mask 暫存器，控制通道 0~3（見圖 **13-3**）。將 06h 寫入 port 0Ah，可以 "masks" 通道 2，於是我們便可以安全的規劃它。最低兩個位元（10b）指出通道 2，位元 2（1b）表示我們希望 "mask" 這個通道。

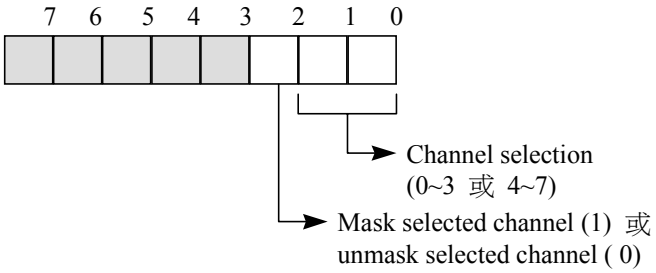


圖 13-3　DMA channel mask register

2. Port 0Bh 是第一個 DMA 控制器的 mode 暫存器（見圖 **13-4**）。將 46h 寫入 port 0Bh，代表以下意思：最低兩個位元（10b）表示我們要設定通道 2 的模式（mode）。位元 2 和位元 3（01b）表示這是一個寫入動作；也就是說我們會從 I/O device 讀出資料並且寫入記憶體。位元 4 表示我們不希望 DMA 通道在此動作完畢後自動對額外的資料傳輸重新初始化。稍後我會解釋爲什麼在 Windows 95 之下不使用 "autoinitialization" 模式（位元 4 設爲 1）。位元 5（0b）表示我們希望在每一次資料傳輸之後，控制器可以累加 address 暫存器內容（另一個選擇是累減）。位元 6 和 7（01b）選出軟式磁碟機控制器所期望的 "single" 傳輸模式，其他傳輸模式包括 00b（demand mode）及 10b（block mode）；數值 11b 則表示「串接（cascade）」（只用於通道 4）。

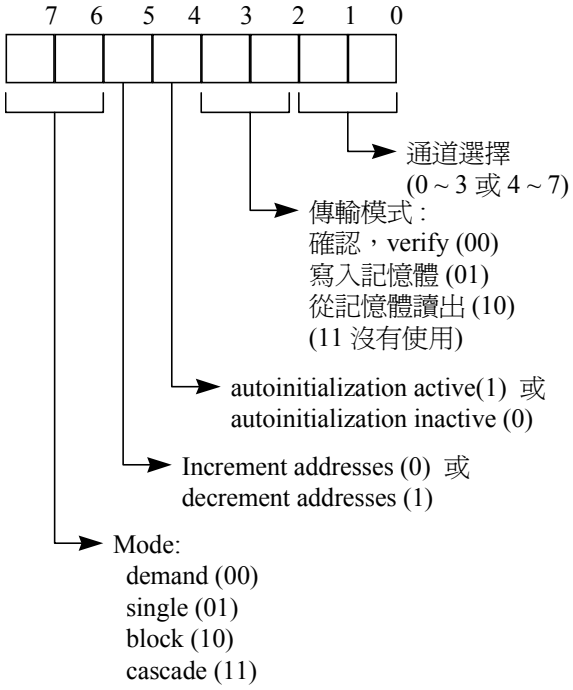


圖 13-4　DMA mode register

3. Port 04h 是通道 2 的 address 暫存器，port 81h 則是通道 2 的 page 暫存器。我們先將實際緩衝區位址的最低 16 個位元一次一個 byte 地寫至 port 04h，然後再將較高一個位元寫至 page 暫存器。爲了節省電路（註），DMA 晶片有一個內部正反器（flip-flop）表示接收的資料是「16 位元 address 暫存器及 count 暫存器」的 low-order byte 或 high-order byte。爲了讓正反器（flip-flop）進入一個已知狀態，表示下一筆資料是 "low-order byte"，我們先將一個任意數值寫入 port 0Ch。

> 張訓賓註：使用正反器（flip-flop）共需 3 bytes。若直接將 address 暫存器和 count 暫存器都以兩個 port 表示，則只需 4 bytes，節省 25%。

4. Port 05h 是通道 2 的 count 暫存器。我們將希望傳送的資料總數減 1，一次一個 byte 寫到這個暫存器。前面所提的同一個正反器被用來控制「寫入此 port

的 byte 被解釋爲 low-order 或 high-order」。上述步驟 3 中第二次對 port 04h 的寫入動作，使正反器保持了步驟 4 所需的正確狀態。

5. 最後，將 02h 寫入 port 0Ah，以便 "unmask" 通道 2，於是 device 一準備妥當就可以開始傳送資料。

如果你真的希望從軟碟機讀取一個 sector 的資料，你現在應該寫程式規劃軟式磁碟機控制器，使能夠啓動馬達並使用 DMA 完成一個 sector 的讀取。一旦控制器收到 sector 讀取命令序列的最後一個 byte，它就應該移動磁碟讀寫頭到指定的 track 上頭，找出被指定的 sector，開始傳送資料到記憶體匯流排（memory bus）。每次軟碟機控制器讀取一個新的 data byte，就會通知 DMA 控制器。DMA 控制器會禁止 CPU（及其它記憶體的使用者）存取記憶體，並讓資料進入下一個緩衝區位置。在軟碟機控制器「移動磁頭尋找正確的 sector」和「真正傳送資料」的空檔期間，CPU 可以正常存取記憶體。當傳輸結束，不管是因爲 DMA 控制器傳送完資料數目或是因爲軟碟機控制器結束了磁碟動作，軟碟機控制器會透過保留的 IRQ 訊號線通知 PIC（通常是 IRQ 6，對應真實模式系統的中斷 0Eh）。系統軟體會「送出 EOI 命令到中斷控制器」來處理此中斷，並從軟碟機控制器身上讀取 status bytes。

檢視 DMA 架構，幾乎可以看出 PC 設計的歷史淵源。DMA 晶片使用實際位址（physical address），而非虛擬位址（virtual address）。像 Windows 95 之類擁有 paging 功能的作業系統，面對 DMA 傳輸時，就必須做大量的轉換和 page-locking 的工作。**13-5** 說明當一個 device driver 希望傳送 512 bytes 資料時，可能必須完成的準備工作。這個 device driver 從一個虛擬緩衝區位址著手。由於 DMA 控制器只認識實際位址，所以 driver 一開始必須先參考一組 page tables，將虛擬緩衝區位址轉換爲實際位址。驅動程式必須鎖住（lock）記憶體中的這個緩衝區，避免 paging subsystem 在資料傳送時爲了其他某些目的而使用相同的實際記憶體。此外，一個緩衝區可能跨越兩個 virtual pages（如圖所示）邊界，這種情況下，驅動程式尚需處理「緩衝區不座落於相鄰 physical pages」的可能性。視硬體而定，驅動程式可能需要規劃兩次傳送，或使用硬體的 scatter/gather 機制來安排一次不連續的傳輸。

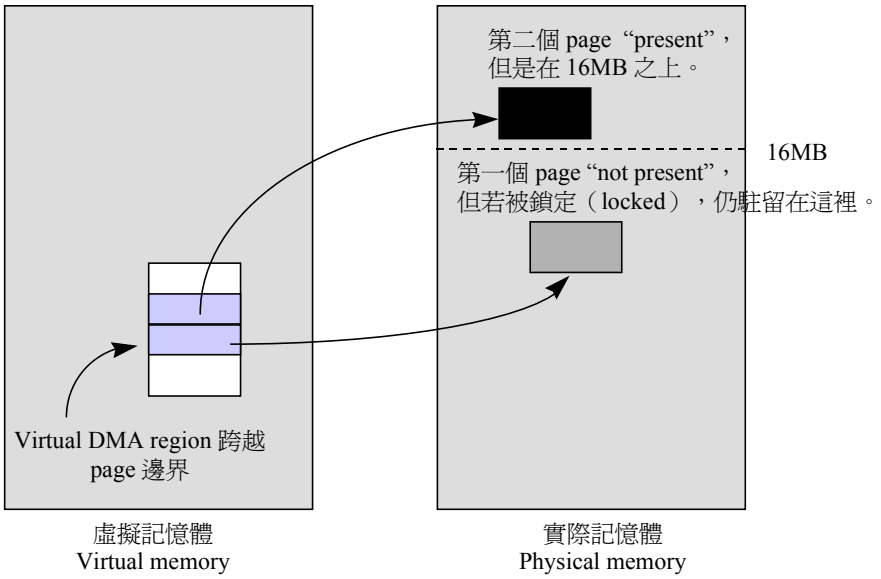


圖 13-5　DMA 傳輸，從一個緩衝區傳往虛擬機器

因虛擬記憶體而導至的困難並不是只有這麼一點。回想一下，DMA 控制器有一個 24-bit 位址，只夠定址出 16MB 實際記憶體。萬一內含緩衝區的那些 pages 碰巧落在 16-MB 界線以上呢？Windows 95 VDMAD VxD 負責處理 16MB 界線以上的記憶體傳送，作法是在一個預先配置的 DMA 緩衝區上複製資料，此緩衝區被鎖進（locked into）16MB 界線下的實際記憶體。

張訓賓（本章技術檢閱）註：由於 DMA 控制器在做 16 位元傳輸時，會先將 page 暫存器與 address 暫存器所組成的 24 位元位址左移一位，所以程式在取得緩衝區的實際（physical）位址時，需先右移一位後再寫入 page 暫存器及 address 暫存器。此時的 page 暫存器與 address 暫存器以「記憶體空間中的第幾個 word」視之。

面對 16-bit DMA 通道，需要注意一件事情。此時的 DMA 控制器要求你傳送資料時必須以記憶體的 word（16-bit）邊界開始。事實上控制器是對 24-bit 數字（由 page 及 address 暫存器組成）左移一個位元（損失最高位元）以產生記憶體位址，並將 count 暫

存器的計數單位解釋爲 words 而非 bytes。因此，你可以使用 16-bit DMA 通道中的一個，以 word 爲單位，傳送最多 128KB 資料（譯註：count 暫存器爲 16 位元，一個 word 有 2 個 bytes，所以 2^16 * 2 = 128KB）。

如果 address 暫存器在傳送期間 overflow 或 underflow，DMA 控制器並不會自動增加或減少 page 暫存器。所以，假設你規劃一個 512-byte 資料傳輸，並且以實際位址 4FFFh:0000h（4FFF0h）開始，你必須設定 page 暫存器爲 4，address 暫存器爲 FFF0h。傳送 16 bytes 之後，DMA 控制器對 address 暫存器的增量導至超過其 16-bit 最大值，於是下一個記憶體位置變成了 40000h 而非正確的 50000h（張訓賓註：offset 暫存器不會進位至 page 暫存器）。因此，你必須確保 DMA 傳輸時不跨越 64-KB 邊界（如果是 16-bit DMA 則爲 128-KB）。

在一部擁有 EISA bus 的電腦上，DMA 會比較簡單一點。EISA 除了 8-bit 及 16-bit 資料傳送外，還支援 32-bit 資料傳送。而且，EISA 也支援 32-bit 緩衝區定址。因此，EISA bus 允許 DMA 資料傳送到記憶體的任何地方，不論是在 16 MB 之上或之下。

硬體資源虛擬化

在 Windows 95 之前，VxD 的目的是要提供「硬體行爲之虛擬化」給「執行於各個 VM 中的 ring3 驅動程式」。虛擬化背後的觀念是要讓已存在的真實模式驅動程式繼續能夠處理硬體，於是硬體廠商就可以不必馬上開發出 VxDs。雖然我所談到的「馬上」是指六年前 Windows 3.0 初次面世時，不過許多硬體廠商至今仍未跟上腳步。因此，Windows 95 只好繼續支援我即將說明的虛擬化技術。

虛擬化 Mapped Memory

記憶體映射裝置（memory-mapped devices）所遭遇的問題是：硬體與某特定範圍之實際位址對應，而使用此硬體之軟體，必須使用「與該實際位址完全等同」的虛擬位址。我將使用「硬體位址」來描述硬體所對應的位址；例如 VGA 文字模式（text-mode）的硬

體位址爲 B8000h （DDK 稱此爲實際位址）。將映射記憶體（mapped memory）虛擬化的方法有兩個步驟。首先，對於共享相同 device 的每一個 VM，各自提供不同的記憶體區域。其次，如果 device 的單一擁有者能夠爲同時存在的每一個 VM 模擬硬體（對顯示幕而言正是如此 -- MS-DOS session 視窗就是這種情況），你應該讓每一個 VM 的 page tables 將虛擬硬體位址映射到每個 VM 自己的記憶體區域去。如果一次只有一個 VM 可以擁有此 device，則每次 device 擁有者改變時，你可以置換映射記憶體（mapped memory）的全部內容。

切換 Memory-Mapped Device 的狀態

Windows 95 Virtual Display Device（VDD）VxD 結合了前面所提的兩個方法。爲了處理視窗化的 MS-DOS VM，VDD 允許各個 VM 一如往常地從 B0 或 A0 segment 存取它們各自的 display memory（這些 display memory 其實是虛擬的）。VDD 與兩個其它元件緊密配合（一個是 video "grabber"，另一個是 WINOLDAP Windows 程式），以適時的方式更新 VM 視窗。WINOLDAP 擁有「顯示（代表）VM」的那個螢幕視窗，它呼叫 grabber 執行螢幕更新動作以及視窗中的選取動作（譯註：指的是類似 copy/paste 時的螢幕內容選取），grabber 內部會呼叫 VDD API 函式。*Device Driver Adaptation Guide* （Windows 3.1 DDK [Microsoft Corporation，1992] 的一部份）第三章有關於 video grabbers 的更多資料。

使用者經由 Windows 95 工作列（taskbar）或其他的使用者介面元件，選擇一個 VM，使它擁有焦點（focus）。這個 focus VM 會被移到前景。SHELL VxD 利用 *Set_Device_Focus* 這一系統控制訊息，將硬體的所有權移轉到新的 focus VM 身上。如果新的 focus VM 在視窗中執行，SHELL 會轉移很多 devices（其中包括顯示幕）的所有權給 System VM（張訓賓註：即使此時的 focus VM 並不是 system VM。詳見本段之後的註解）。如果新的 focus VM 在全螢幕中執行， SHELL 則把顯示幕指定給該 VM。VDD 回應 *Set_Device_Focus* 訊息的作法是：將顯示幕前一個擁有者的狀態記錄下來，並安裝新擁有者的狀態（張訓賓註：所謂安裝，包括「初始化」或「回復先前狀態」）。

張訓賓註：其中最大的問題在於高解析度圖形模式下需要很大的記憶體來虛擬化該 display memory，這會使得 VM 間的切換變得緩慢而不切實際。因此 VDD 通常只對 mode 13h 以下的模式做虛擬化。

一般而言，如今的 super VGA 顯示系統中狀態資料的儲存及恢復非常複雜，尤其當一個或兩個 VMs 裡的顯示幕處於圖形模式之下。爲了簡化我的討論，假設兩個 VMs 都使用 video mode 3（標準 25 行文字模式）。在兩個狀態之間實際上唯一不同的東西，應該是從虛擬位址 B8000h 開始的 video RAM 的內容。

當全螢幕 VM 在前景執行，VDD 將 VM 的 virtual B8h, B9h ... 等等 pages 直接映射至相同的實際記憶體位置（因爲一個 page 是 1000h bytes，page B8h 相當於線性位址 B8000h）。因此，不論什麼時候，只要前景程式在 video RAM 中儲存一個字元或屬性，實際顯示幕就會跟著立即改變。然而當全螢幕 VM 在背景執行，VDD 將其 video RAM pages 映射到 extended memory pages。因此，背景程式不論什麼時候在 video RAM 中儲存一個字元或屬性，使用者看不見變化。爲了從一個前景 VM 切換到另一個，VDD 只需將 physical video RAM 拷貝到「被切換出去之 VM 所屬的 extended memory buffer」中，並將「被切換進來之 VM 所屬的 extended memory buffer」拷貝到 physical video RAM 中。切換完成之後，切換進來的 VM 的 video RAM 中的任何變化即能爲使用者所見。

實作記憶體置換（memory-swapping）的方法可以概述如下。不過這離真正的驅動程式還很遠；這只是一個例子，說明如何處理一個 device。對真實電腦而言這恐怕太過簡單，沒有什麼用。事實上，我能夠給你的所有承諾就是，這個範例（取自書附光碟的 \CHAP13\MEMORYVIRTUALIZATION 目錄）可以正確編譯。

```
PVMMCB hOwner;
PBYTE GetPrivateVideoBuffer(PVMMCB);
PBYTE GetRealVideoBuffer(void);
void MapPrivateVideoPages(PVMMCB);
void MapRealVideoPages(PVMMCB);
#define BUFSIZE (8*4096) // Eight pages of data
```

```
BOOL OnSetDeviceFocus(PVMMCB hVM, DWORD devid, DWORD flags,
    PVMMCB hProblemVM)
    {               // OnSetDeviceFocus
    if (devid && devid != MYVXD_DEVICE_ID)
        return TRUE;   // ignore request for another device
    memcpy(GetPrivateVideoBuffer(hOwner), GetRealVideoBuffer(),
        BUFSIZE);
    memcpy(GetRealVideoBuffer(), GetPrivateVideoBuffer(hVM),
        BUFSIZE);
    MapPrivateVideoPages(hOwner);
    MapRealVideoPages(hVM);
    hOwner = hVM;
    return TRUE;
    }               // OnSetDeviceFocus
```

這段碼需要一些說明及推敲。*Set_Device_Focus* 系統控制訊息夾帶一個 VxD ID 做爲參數，其用意是，被指定的 VxD 必須將它所管理的 devices 的 focus 切換到指定的 VM 上 -- 因爲每一個 VxD 會收到所有（而不只是它所管理的）devices 的 *Set_Device_Focus* 訊息。習慣上，一個夾帶 device ID 0 的 *Set_Device_Focus* 訊息會使得「focus 的重設是有意義的」的所有 devices，將 focus 重新設定到指定的 VM 身上。上述函式的第一行就是偵測這些情況，以便能夠略去其它驅動程式所發出的 requests。

我使用一個名爲 *GetRealVideoBuffer* 的輔助函式，其目的是傳回實際的 video buffer（定址於實際位址 B8000h）的線性位址。這個函式如果只是傳回 B8000h 並不正確，它必須根據目前使用的 page table 傳回虛擬位址。1MB 以下的記憶體虛擬位址（即 V86 位址空間），可能直接對應到相同的實際位址，也可能被映射至 extended memory 中的某些 pages -- 視 VM 是否爲「當班」的 VM 而定。在我們收到 *Set_Device_Focus* 訊息時之前，我們無法知道目前是哪一個 VM「當班」，所以我們不應該假設虛擬位址 B8000h 一定會映射到實際位址 B8000h。*GetRealVideoBuffer* 的正確作法是使用 *_MapPhysToLinear* VxD service：

```
PBYTE GetRealVideoBuffer(void)
    {               // GetRealVideoBuffer
    return _MapPhysToLinear(0xB8000, BUFSIZE, 0);
    }               // GetRealVideoBuffer
```

以此方法所獲得的線性位址會在 Windows 95 session 期間保持不變。事實上，我們可能會在 device 初始化時就呼叫一次 *_MapPhysToLinear*，並將傳回值存在全域變數裡備用。在初始化期間呼叫，可以節省時間的重複浪費，因爲 *_MapPhysToLinear* 會檢查是否映射已經完成。

_MapPhysToLinear 會產生線性位址與實際位址之間的永久映射，這個事實在 Windows 95 會產生問題，因爲在 Windows 95 session 期間，physical device 位址可能會改變。Video 顯示幕的位址是不會改變啦，因爲 video driver 的組態（configuration）在 boot time 時就已決定，並且在機器重新開機前，不可能改變。但是一般而言 Configuration Manager 可以對所有其它 I/O 資源重新指定記憶體位址。爲了避免破壞線性位址空間，導致無法恢復，面對 Plug and Play 驅動程式你必須使用不同的方法：在你的 configuration 函式的 *CONFIG_START* 處理常式中使用 *_PageReserve*, *_PageCommitPhys*, *_LinPageLock* 來映射被指定的記憶體；在 *CONFIG_STOP* 及 *CONFIG_REMOVE* 處理常式中再將這些函式恢復原狀。下面程式碼說明如何完成這件事：

```
case CONFIG_START:
    [determine physical memory address "physaddr" and length "npages"]
    ULONG linaddr = _PageReserve(PR_SYSTEM, npages,
        PR_FIXED);
    _PageCommitPhys(linaddr >> 12, npages, physaddr >> 12,
        PC_INCR | PC_WRITEABLE);
    _LinPageLock(linaddr >> 12, npages, 0);
    ...
case CONFIG_STOP:
case CONFIG_REMOVE:
    _LinPageUnlock(linaddr >> 12, npages, 0);
    _PageFree(linaddr, 0);
    ...
```

在處理 *CONFIG_START* 的時候，呼叫 *_PageReserve* 配置一些虛擬位址，使我們可以定址 device 的記憶體。*_PageCommitPhys* 指示 paging manager 建立起一些 page table entries（PTEs），用來將實際記憶體映射到虛擬位址中。*_LinPageLock* 可以確保「虛擬至實際」映射不會改變。順便一提，DDK 文件說「你既已使用 *_PageCommitPhys* 映射記憶體，則不能 lock 其結果」並不正確；你可以如本例所示做 lock 動作。處理

CONFIG_STOP 及 *CONFIG_REMOVE* 的時候，*_LinPageUnlock* 和 *_PageFree* 這兩個動作會恢復 *CONFIG_START* 時所做的動作。

我使用另外一個名爲 *GetPrivateVideoBuffer* 的輔助函式，存取一份「8-page private 記憶體區塊」，其中內含某個 VM 的 video RAM 的副本（張訓賓註：一個 page 有 4K，8 個 pages 共 32K，爲 B800:0000～B800:FFFF 的虛擬化提供了足夠的記憶體空間）。你可以在回應 *Create_VM* 訊息時，呼叫 *_PageAllocate* 以配置這塊緩衝區，並將其位址儲存在 VM control block 內。在 VM control block 內保留這塊空間（張訓賓註：指儲存緩衝區位址所需的 4 bytes），是你在 device 初始化時必須做的工作。因此，爲了實作出 *GetPrivateVideoBuffer* 函式，你還需要這些碼：

```
typedef struct tagMYSTUFF
    {
    PVOID pVidBuff;       // linear address of video buffer
    } MYSTUFF, *PMYSTUFF;
DWORD cboffset;           // offset of MYSTUFF in VMCB
#define GetStuff(hvm) ((PMYSTUFF) ((DWORD) hvm + cboffset))

BOOL OnDeviceInit(PVMMCB hVM, DWORD refdata)
    {                   // OnDeviceInit
    if (!_Assign_Device_V86_Pages(0xB8, 8, NULL, 0))
        return FALSE;     // can't globally assign pages
    cboffset = _Allocate_Device_CB_Area(sizeof(MYSTUFF), 0);
    if (!cboffset)
        return FALSE;
    return OnCreateVM(hVM); // initialize system VM data now
    }                     // OnDeviceInit

BOOL OnCreateVM(PVMMCB hVM)
    {                     // OnCreateVM
    PVOID pVidBuff;
    pVidBuff = (PVOID) _PageAllocate(8, PG_VM, (HVM) hVM, 0, 0, 0, 0,
        PAGEZEROINIT);
    if (!pVidBuff)
        return FALSE;
    GetStuff(hVM)->pVidBuff = pVidBuff;
    return TRUE;
    }                   // OnCreateVM

BOOL OnDestroyVM(PVMMCB hVM)
```

```
    {                   // OnDestroyVM
    _PageFree(GetStuff(hVM)->pVidBuff, 0);
    return TRUE;
    }                   // OnDestroyVM

PBYTE GetPrivateVideoBuffer(PVMMCB hVM)
    {                   // GetPrivateVideoBuffer
    return GetStuff(hVM)->pVidBuff;
    }                   // GetPrivateVideoBuffer
```

OnDeviceInit 首先使用 *_Assign_Device_V86_Pages*，將 V86 region 中的 video RAM pages 標記爲屬於我們的 VxD 所有。*_Assign_Device_V86_Pages* 會檢查一個 bit map，在那裡記錄每個 V86 page 是否由某些 VxD 擁有；bit map 不會指出誰擁有 page，只會表示該 page 是否被擁有。如果我們準備虛擬化的 8 個 pages 並未被其它 VxD 擁有，*_Assign_Device_V86_Pages* 會把它們標記爲「被擁有」，並傳回 TRUE；否則傳回 FALSE。實際情況中，此一呼叫很少會失敗，因爲 V86 中的任一 page，應該只會有一個 VxD 對它感興趣。

保留了 video RAM pages 以後，*OnDeviceInit* 函式利用 *_Allocate_Device_CB_Area*，在每一個 VM control block 裡頭保留 device 的特定空間。我們打算利用這個空間來記載我們所配置的 private video buffer 位址。最後，*OnDeviceInit* 呼叫 *OnCreateVM* 來初始化 System VM；還記得嗎，沒有任何一個 *Create_VM* 訊息會爲 System VM 發生，所以 VxDs 通常會在 *Device_init* 的時候爲 System VM 做「per-VM initialization」的任何必要動作。

_PageAllocate 函式　這是所有 VxD 都可使用之記憶體配置函式中，最基本的一個。這個函式保留一塊連續的線性位址空間，並委派（commits）足夠的記憶體來映射那些位址。所謂委派一塊（committing）記憶體，意謂有足夠的實際記憶體來持有區塊內容。實際記憶體可能來自 RAM 或置換檔（swap file）。Windows 95 之所以委派（commits）記憶體，是爲了確保 page fault 絕不會因爲「無法將一個 page 寫到 swap file 中」而失敗。如果你喜歡，你可以分別呼叫 *_PageReserve* 及 *_PageCommit*，以兩個步驟來「保留位址空間」及「委派 pages」，以取代單一呼叫 *_PageAllocate*。

_PageAllocate（定義於 VXDWRAPS.H）的函式原型是：

```
PVOID _PageAllocate(DWORD nPages, DWORD pType, HVM hvm,
   DWORD AlignMask, DWORD minPhys, DWORD maxPhys,
   PVOID *PhysAddr, DWORD flags);
```

其中參數 *nPages* 表示你希望配置的 pages 數目。參數 *pType* 表示常數 PG_SYS 或 PG_VM；PG_SYS pages 係配置於位址空間裡頭的 shared system region（C0000000h 以上），而 PG_VM pages 則是配置於 shared application region（80000000h~C0000000h）。Windows 95 允許使用第三個常數：PG_HOOKED，其意義與 PG_VM 相同。第三個參數是一個 VM handle（也就是說，一個指向 VM control block 的指標）；請注意，DDK 將 HVM 定義爲一個 DWORD，但 HVM 實際上與 PVMMCB 有著相同的本質。如果你以 PG_VM 或 PG_HOOKED 兩種屬性來配置 pages，你所指定的 VM handle 將擁有這些 pages。如果你以 PG_SYS 來配置 pages，這個（第三）參數應是 NULL，因爲這些 pages 不屬於任何 VM。

參數 *flags* 允許你根據特別的目的來調整這次的配置行動。你可以將一些描述特殊需求的常數，以 OR 串接起來，產生這個參數。PAGEZEROINIT flag 的意義非常明顯，PAGEFIXED 及 PAGELOCKED 兩者會鎖住（lock）配置得來的 pages；也就是說它們會使得 page manager 立即保留那些實際記憶體給這些 pages，避免被「偷走」（被對應給其他的 page）。這兩個 flag 的差異在於，PAGEFIXED 表示 pages 不需要解鎖（unlocked），而 PAGELOCKED 表示你必須呼叫 *_PageUnlock* 來解除鎖定（unlocked）。

PAGELOCKEDIFDP flag 也會像 PAGELOCKED 一樣地鎖住 pages，但是只用於「在 Windows 95 中使用 MS-DOS 函式或 BIOS 函式處理 paging I/O」時，這並不常見。如果你需要在一個 event callback 期間存取記憶體區塊，你應該使用這個 flag，因爲彼時你事先並不知道 paging 是否沒問題。VMM 直到 *Init_Complete* 訊息被送出才知道是 MS-DOS 還是 ring 0 VxD 應負起 paging 的責任。Static driver 可以等待直到 *Init_Complete* 出現才配置一塊需要這種條件式鎖定行爲的記憶體區塊。Dynamic driver 可以藉由以下的碼達到相同結果：

```
BOOL OnSysDynamicDeviceInit()
    {                   // OnSysDynamicDeviceInit
    if (VMM_GetSystemInitState() >= SYSSTATE_VXDINITCOMPLETED)
        return OnInitComplete(Get_Sys_VM_Handle(), 0);
    else
        return TRUE;
    }                   // OnSysDynamicDeviceInit

BOOL OnInitComplete(PVMMCB hVM, DWORD refdata)
    {                   // OnInitComplete
    _PageAllocate(..., PAGELOCKEDIFDP);
    ...
    }                   // OnInitComplete
```

在此程式片段中，我使用 *VMM_GetSystemInitState* 來判斷 *Init_Complete* 是否已經發生。如果還沒發生，就等待該訊息的到來。如果發生了，就呼叫該訊息處理常式。

_PageAllocate 所使用的 PAGEUSEALIGN flag 允許你控制「PAGELOCKED pages 應該被放在實際記憶體的什麼地方」。如果你指定 PAGEUSEALIGN 及 PAGELOCKED，則我尚未說明的 4 個參數就有作用。*AlignMask* 可以是 0h, 01h, 03h, 07h, 0Fh, 或 1Fh 數值中的任一個，用以指定一個特定的 page 邊界上的 alignment。例如，指定 *AlignMask* 爲 0Fh，表示實際位址必須是 16 pages 的倍數，也就是說 alignment 爲 64-KB。參數 *minPhys* 及 *maxPhys* 用來指定記憶體區塊的最小/最大實際位址。參數 *PhysAddr* 如果不是零，就代表一個位址 -- *_PageAllocate* 應該將配得的實際位址儲存於該處。最後，如果你指定 PAGEUSEALIGN 和 PAGELOCKED，你也可以指定 PAGECONTIG 以確保配置而得的實際記憶體是連續的。

這些額外參數的目的之一是要讓 VDMAD 可以使用 *_PageAllocate* 來配置 DMA buffer。VDMAD 的典型要求是：最大實際位址 16MB、alignment 爲 128-KB（*AlignMask* 等於 1Fh）。VDMAD 也會要求連續的實際記憶體（使用 PAGECONTIG flag），並且必須知道實際位址（*PhysAddr* 不爲 NULL）。

_PageAllocate 的傳回值表示被配置的記憶體區塊的線性位址，當你呼叫其它 page allocation service 時，可以把此線性位址當做 handle 使用。在以前的 Windows 版本中，

_PageAllocate 除了傳回線性位址外，還傳回一個 handle。如果你必須寫一個相容於 Windows 3.x 的 VxD，你需要寫自己的 C wrapper 函式，夾帶一個額外的指標參數，如此一來就可以同時傳回 handle 和線性位址。

將 Pages 映射進入 VM　剩下兩個未解釋的輔助函式是 *MapPrivateVideoPages* 和 *MapRealVideoPages*，它們會分別設置 private video RAM 與 real video RAM 的 page table entries（PTEs）：

```
void MapPrivateVideoPages(PVMMCB hVM)
    {                           // MapPrivateVideoPages
    _MapIntoV86(GetStuff(hVM)->pVidBuff, (HVM) hVM, 0xB8, 8,
        0, 0);
    }                           // MapPrivateVideoPages

void MapRealVideoPages(PVMMCB hVM)
    {                           // MapRealVideoPages
    _PhysIntoV86(0xB8, (HVM) hVM, 0xB8, 8, 0);
    }                           // MapRealVideoPages
```

_MapIntoV86 會將記憶體區塊內的一些（或所有）pages 映射到 VM 中的一個指定的虛擬位址上。*_PhysIntoV86* 則是將 physical pages 映射到 VM 中。所謂映射（"mapping），意指設置 page table entries（PTEs），使它們指向特定的 physical pages。*MapPrivateVideoPages* 將虛擬位址 B8000h~BF000h 的 PTEs 設定指向 private video RAM buffer 裡的 pages。*MapRealVideoPages* 將虛擬位址 B8000h~BF000h 的 PTEs 設定指向實際位址 B8000h~BF000h。

動態配置記憶體

我所解釋的這個範例程式在實際情況下有一個很大的問題：浪費虛擬記憶體。看看在 *OnCreateVM* 裡呼叫的 *_PageAllocate*，你就會瞭解，記錄顯示幕的完全狀態（含圖形及文字模式），需要的 pages 個數遠大於 8。假設每次我們建立一個新的 VM 時就配置所有這些記憶體。由於記憶體的配置需要 Windows 95 在置換檔（swap file）中委派（commit）空間，置換檔會有顯而易見的成長，虛擬位址空間也會被大量貢獻給 private

display memory buffers。如果使用者只是打算在 25 行文字視窗中輸入簡短的 ver 命令，這不是很愚蠢嗎？（不但如此，你還無法從 ver 的結果獲得任何有用的資料呢！☺ ）

至少，在記憶體映射到虛擬位址空間的 V86 region 時，我們可以安排讓 VM 利用 page faults 通知 device，告知其所需要的 mapped memory，然後我們再將對應的記憶體配置給它。欲完成這種「隨需求而配置」的記憶體，標準方法是呼叫 *Hook_V86_Page*，以便爲此 device 位址範圍建立起一個 page-fault callback。不幸的是我現在要說的這個方法，不適用於「devices 的 mapped memory 在 V86 region 之外」的情況。因爲沒有 VxD services 可以讓你處理「映射至那些位址」的 page tables。如果你打算虛擬化這樣的 device，你必須事先配置「打算用來儲存 device 狀態」的緩衝區。

繼續先前的例子，你的 *Device_Init* 處理常式可能會像這樣地 "hook" VGA 文字緩衝區：

```
BOOL OnDeviceInit(PVMMCB hVM, DWORD refdata)
    {                           // OnDeviceInit
    DWORD page;

    if (!_Assign_Device_V86_Pages(0xB8, 8, NULL, 0))
        return FALSE;           // can't globally assign pages

    cboffset = _Allocate_Device_CB_Area(sizeof(MYSTUFF), 0);
    if (!cboffset)
        return FALSE;

    for (page = 0xB8; page < 0xC0; ++page)
        if (!Hook_V86_Page(page, OnPageFault))
            {                   // couldn't hook page
            while (--page >= 0xB8)
                Unlock_V86_Page(page, OnPageFault);
            return FALSE;
            }                   // couldn't hook page

    return OnVMCriticalInit();

    }                           // OnDeviceInit
```

你不再需要於 *Create_VM* 期間配置緩衝區，但你需要處理 *VM_Critical_Init* 訊息，以便

準備 mapped pages，以求真正產生 page faults：

```
BOOL OnVMCriticalInit(PVMMCB hVM)
   {                        // OnVMCriticalInit
   return _ModifyPageBits((HVM) hVM, 0xB8, 8, ~P_PRES, 0,
      PG_HOOKED, 0);
   }                        // OnVMCriticalInit
```

_ModifyPageBits 函式原型如下：

```
BOOL _ModifyPageBits(HVM hVM, DWORD VMLinPgNum,
   int nPages, const int bitAND,
   const int bitOR, DWORD pType, DWORD flags);
```

這個函式修改 hVM 虛擬機器中的 *nPages* 個 PTEs（page table entries），從 page 號碼 *VMLinPgNum* 開始，它先將之與 *bitAnd* mask 做 AND 運算，再與 *bitOr* mask 做 OR 運算。我們在此函式中清除（reset）video RAM pages 的 "present" 位元，如此一來當下次參考到範圍 B8000h~BFFFFh 的任何虛擬位址時，就會引發 page fault。當我們 "hook" B8h~BFh pages 時，我們已經將我們的 *OnPageFault* 函式指定爲這些 page 的 page-fault callback 函式。

爲什麼我要在 *VM_Critical_Init* 訊息處理常式中呼叫 *_ModifyPageBits* 呢？有一個理由。回想一下，一個新的 VM 建立時，會引發三個系統控制訊息送至每一個 VxD：*Create_VM*, *VM_Critical_Init*, *VM_Init*。*Create_VM* 發生於「新 VM 誕生當時的 current VM」的 context 之中。通常這 current VM 就是 System VM，因爲產生一個新 VM 的訊號通常是來自一個像 *WinExec* 的函式。*VM_Init* 發生於新 VM 的 context 之中，任何 VxD 都可自由使用 *Resume_Exec* 在此 VM 中執行 V86 碼。一般而言，我們希望我們的 device 在新的 VM context 中設置妥當 -- 在任何 V86 碼有機會用到我們的 device 之前。這就是 *VM_Critical_Init* 訊息之所由來：它發生於新 VM 的 context 之中，但是在所有 VxDs 處理過它之前，沒有任何一個 VxD 可以執行 V86 碼。

終於，有一些執行於新 VM 中的 V86 程式嘗試使用我們的 device，它們觸及 B8000h~BFFFFh 記憶體。由於我們將這個範圍的所有 pages 都標示爲 "not present"，所

以 page fault 會發生。於是控制權來到 page-fault callback 函式身上，那是我們經由 *Hook_V86_Page* 建立起來的：

```
void __declspec(naked) OnPageFault()
   {                            // OnPageFault
   _asm
       {                        // handle page fault
       push    ebx              ; current VM handle
       push    eax              ; page number that caused the fault
       call    HandlePageFault ; call real handler
       add     esp, 8           ; lose args
       ret                      ; return to caller
       }                        // handle page fault
   }                            // OnPageFault

void HandlePageFault(DWORD page, PVMMCB hVM)
   {                            // HandlePageFault
   PBYTE pVidBuff = (PBYTE) _PageAllocate(8, PG_HOOKED,
       (HVM) hVM, 0, 0, 0, 0, PAGEZEROINIT);
   if (pVidBuff)
       {                        // map private buffer
       GetStuff(hVM)->pVidBuff = pVidBuff;
       _MapIntoV86(pVidBuff, (HVM) hVM, 0xB8, 8, 0, 0);
       }                        // map private buffer
   else
       Nuke_VM(hVM);
   }                        // HandlePageFault
```

這個 page fault 處理常式需要以 assembly 語言來寫，因爲它的參數是從暫存器中獲得，這在一個標準的 C 函式是沒有辦法的。我使用 *__declspec(naked)* 指令，強迫編譯器省去所有的 prolog 和 epilog 碼，所以我可以爲 *HandlePageFault* 函式只寫一個小小的 inline assembler wrapper。後繼的函式才負責所有的工作。它首先嘗試配置一個 8 page 緩衝區。如果配置成功，就呼叫 *_MapIntoV86*，將配置得來的緩衝區安裝到 VM 的 page tables 中。當然，配置也可能失敗，於是我們就會獲得一個藍色畫面，解釋爲什麼需要在呼叫 *Nuke_VM* 結束這個 VM 之前，將 VM "crash" 掉。

我所描述的這個動態配置實例需要其他一些邏輯，才適用於真正的 video display device。我會解釋什麼是必要的，並將細節留給你做爲練習。請注意，我等待一個 page fault，然

後將一個私有的緩衝區位址安裝到 VM 的「與 B8000h 有關的 PTEs」中。只有當 VM 處於視窗之中，這樣的動作對於一個 display driver 而言才是正確的。如果 VM 處於全螢幕狀態，driver 需要做點不同的事。你還是需要誘捕（trap）page faults，但是不再配置私有緩衝區並將緩衝區的 pages 於 page fault 發生時安裝至 VM page table，而是設立一個 flag，表示此一 VM 可以處理 video 文字模式記憶體。然後就呼叫 *_PhysIntoV86* 將 VM 解放於真實硬體。萬一這個 VM 稍後失去了 focus，你的 *Set_Device_Focus* 處理常式便會看到這個 flag 被設立起來，你必須配置緩衝區以儲存 這些 video RAM pages。相反的，如果 focus 被切換至某一 VM 而它從未有爲此而配置的緩衝區，你會得到通知說這個 flag 沒有設立，表示並沒有被儲存下來的狀態可以恢復，所以你必須直接將 video 硬體初始化。

將 I/O Ports 虛擬化

將 I/O ports 虛擬化，必須仰賴 Intel 處理器的 I/O permission mask 特性。Windows 95 使用的唯一一個 task state segment（TSS）內含一個可變長度的 permission mask，其中每一個 bit 對應於一個 8-bit I/O port。兩個鄰近的 bits 控制了一個 16-bit port，而四個鄰近的 bits 控制了一個 32-bit port。在允許「於 ring3 中執行的 V86 模式或保護模式程式」存取一個 port 之前，處理器會檢查 TSS 上對應的 bit(s)。如果所有對應到該 port 的 bits 都是 0，處理器就允許該動作進行。如果有任何 bit 爲 1，處理器會產生一個 general protection fault，讓 VMM 進場干涉。至於 ring0 程式，可自由存取所有的 I/O ports。

更進一步討論 I/O Permission Mask　Intel CPU 如果處於 V86 模式，那麼它總是會在允許存取 I/O port 之前先檢查 I/O permission mask。如果 CPU 處於保護模式，存取控制機制就比較複雜。Flags 暫存器內含有兩個位元，稱爲 I/O privilege level（IOPL），還記得嗎，current code selector 可以決定 current privilege level（CPL），那是執行中的程式的 "privilege ring"。如果這個 CPL 小於或等於 IOPL，I/O 不必被檢查就可以執行。相反地，如果 CPL 比 IOPL 大，CPU 就會查閱 permission mask。你已經知道了，VMM 和其他的 VxDs 都在 ring0 保護模式下跑（CPL == 0），所以這些規則意味著任何 VxD 可以存取任何 port。你也已經知道，Windows 應用程式在 ring3 保護模式下跑（CPL ==

3）。現在你需要知道的是，當執行保護模式應用程式時，VMM 會迫使 IOPL 爲 0。因此，應用程式存取 port 時就會受到 permission mask 的影響。（如果是 V86 程式，則 IOPL 總是 3，其理由已在第５章說明過，但是 IOPL 對 V86 模式中的 I/O 並沒有任何影響）

VMM 將 I/O permission mask 初始化爲 0。因此在預設情況下，任何 I/O port 對任何程式基本上都是可用的。我猜想這意味著我可以在兩個不同的 MS-DOS 視窗中啓動 serial 通訊程式，並令它們都存取 COM1 ports，藉此混雜資料並完全迷惑 universal asynchronous receiver-transmitter（UART），是嗎？（答案：是的，只不過 I/O permission mask 並不會停留於預設狀態，所以有些人會捕捉（trap）這些存取動作）

捕捉（Trapping）I/O Ports

要避免 VMs 之間因爲存取同一個未共享的 ports 而發生的不幸衝突，你必須捕捉你的 ports，方法是呼叫 *Install_IO_Handler* 以便在 VMM 中註冊一個 I/O callback 函式。一旦 ring3 碼嘗試存取一個你所 trap 的 port，VMM 便會呼叫你這個函式。此一 callback 函式應該做以下事情：

- 執行程式的 I/O 動作。意思是直接發出 IN 或 OUT 指令到硬體。由於這個 callback 函式是 ring0 VxD 的一部份，這些指令並不會被 "permission mask checking" 所影響。
- 模擬 I/O 動作。這種情況是在當你擁有一個 device 而它不能夠容忍「當它發出中斷，至 driver 回應」之間的一個延遲（latency）。畢竟完全於 ring0 之中回應一個中斷，遠比先切換至 ring3 快多了。另一個理由是所謂「模擬一個動作（operation）」也就是產生一個假想硬體。第三個理由是允許一個程式在其他程式擁有真實硬體時執行，並與之互動。舉個例子，將 video display 虛擬化，使你能夠在圖形視窗中執行一個 MS-DOS session。
- 發現兩個不同的 VMs 正嘗試要存取相同的硬體。這種情況下，你的 VxD 或許得警告使用者說「發生了搶奪情況」，並將搶奪狀態下的 VMs 中的某一個結束掉。

- 如果沒有其他的 VM 也在執行，就將 port trap "disable"。爲一個 VM "disable" trap，事實上就是將那個 port 的擁有權給予該 VM 並改善效率，因爲可以避免你的 callback 函式後續又被喚起。

一如我即將說明的，以 assembly 語言寫一個 I/O callback 函式，比用 C 語言容易多了，所以本節使用 assembler。VxD 應該在初始化期間建立起 port trap，在結束期間取消之。一個 Plug and Play driver 應該在其 configuration 函式中執行上述動作，以反應 CONFIG_START 和 CONFIG_STOP（和/或 CONFIG_REMOVE）等 requests。一個動態載入且非 Plug and Play 的 driver，會在 *Sys_Dynamic_Device_Init* 和 *Sys_Dynamic_Device_Exit* 期間執行它們。一個靜態的 VxD 或許會在 *Device_Init* 期間建立 port traps，並且不取消 traps。事實上，Windows 3.x（它只支援 static VxDs）並不提供任何方法讓你移除一個 I/O trap，它並要求你不得在 *Init_Complete* 之後才安裝 traps。

你可以呼叫 *Install_IO_Handler* 來建立一個 port trap：

```
mov     esi, offset32 <>IoCallback  ; I/O callback routine
mov     edx, <>port                 ; port number
VMMCall Install_IO_Handler
jc      error
```

其中 *IoCallback* 是你的 I/O callback 函式名稱，*port* 是你要捕捉的 port 號碼。一如往常，回返時設立 carry flag 表示有錯誤發生（通常意味其他 VxD 已在捕捉此 port），回返時清除 carry flag 則表示成功。由於你通常需要捕捉一個以上的 port，所以有一個名爲 *Install_Mult_IO_Handlers* 的 service 可以讓你安裝多個 traps。例如：

```
mov     edi, offset32 iotable     ; I/O table address
VMMCall Install_Mult_IO_Handlers
jc      error
...
Begin_VxD_IO_Table iotable
VxD_IO 3F8h, TrapXmitHoldingReg
VxD_IO 3F9h, TrapInterruptEnableReg
       [etc.]
End_VxD_IO_Table iotable
```

如果這個函式失敗，EDX 將內含失敗發生時的 port 位址。爲了正確清除，你必須移除稍早在表格中的 traps，而不是 EDX 中記錄的那一個。

取消 traps 可以使用類似的 Remove_ services：

```
mov     edx,<> port             ; port address
VMMCall Remove_IO_Handler
```

或

```
mov     edi, offset32 iotable   ; same one we installed with
VMMCall Remove_Mult_IO_Handlers
```

I/O Callback 函式

當一個 V86 或 ring3 程式嘗試處理一個 trapped port，也許是要讀取，也許是要寫入，VMM 會把控制權交到你爲此 port 指定的 I/O callback 函式手上。你的 callback 的目的是爲了執行或模擬引起 trap 的那個指令。表 **13-3** 列出 callback 函式的暫存器參數。如果你正在執行單一的輸入動作，用以讀取 8, 16, 或 32 bytes，應該以 EAX 傳回輸入資料。至於其他情況，回返值並不重要。如果你正在執行單一的輸出動作，用以寫入 8, 16, 或 32 bytes，你應該以 EAX 放置準備輸出的資料。以 assembly 語言來撰寫程式，一部份理由是，有如此多的參數利用暫存器傳遞給你。不過還有一個更重要的理由，稍後我再來說。

暫存器	內容
EAX	輸出動作所需的輸出資料。在輸入動作中沒有意義。
EBX	目前的（current）VM handle（VMCB 的位址）
ECX	Flags，指示 I/O 動作型態。可能是以下之一：

Flag	說明
BYTE_INPUT	Single-byte 輸入動作
BYTE_OUTPUT	Single-byte 輸出動作
DWORD_INPUT	Double-word 輸入動作

暫存器	內容	
	DWORD_OUTPUT	Double-word 輸出動作
	WORD_INPUT	Word 輸入動作
	WORD_OUTPUT	Word 輸出動作
	再加上以下設定：	
	Flag	說明
	ADDR_32_IO	INS 或 OUTS 動作所使用的是 32 位元位址
	REP_IO	INS 或 OUTS 動作，有一個重複的 prefix
	REVERSE_IO	INS 或 OUTS 動作，有把 direction flag 設立起來（也就是說，位址遞減）
	STRING_IO	一個 INS 或 OUTS 動作（HIWORD(ECX) 將持有資料緩衝區的 selector）
EDX	I/O port 位址	
EBP	current VM 的 Client registers。	

表 13-3　一個 I/O callback 函式的暫存器參數

ECX 中的 flag 位元(s) 告訴你 ring3 程式正嘗試執行哪一種 I/O 動作。這些 I/O 動作一共有 54 種不同的組合。程式可能使用一個 INS 或 OUTS 指令來執行一個字串 I/O 動作，如此情況下 ADDR_32_IO, REP_IO, 和 REVERSE_IO flags 共有 8 個不同組合（這三個 flag 位元只有在 STRING_IO 設立的情況下才有意義）。第 9 個可能（往往是最可能的）是程式嘗試做單一的 IN 或 OUT 動作。以這 9 個可能再乘上 3 種大小（8, 16, 或 32 bytes）和 2 個方向（輸入或輸出），就得到 54。

寫一個帶有 54 個 switch-case 的 I/O callback 函式（以便處理所有的可能）是十分令人氣餒的事。幸運的是，VMM 提供了一些 service 能夠讓這項工作輕鬆一些。首先，有一個 service 名爲 *Simulte_IO*，你可以呼叫它來處理所有「比 single-byte IN/OUT 動作更複雜」的情況。*Simulate_IO* 期望獲得「如同一個 I/O callback 所獲得的」相同暫存器內容。它會將 ECX flag bits 解碼，並遞迴呼叫你的 I/O callback 以完成一系列的 BYTE_INPUT 或 BYTE_OUTPUT 動作，爲的是模擬更複雜的動作（註）。通常你不會

呼叫 *Simulate_IO*，通常你會利用 *VMMJmp* 巨集「跳」過去。以此方法，它直接回返到「呼叫你的 IO callback 函式」的呼叫端。此外，有一個巨集名爲 *Dispatch_Byte_IO*，會檢查 ECX 暫存器，看看是否你正面對一個複雜情況。如果是，這個巨集就跳到 *Simulate_IO* 去。然而如果你遭遇 BYTE_INPUT 或 BYTE_OUPUT operations，這個巨集會跳到你自己的程式中的一個 label，讓你自己去完成。

> 張訓賓註：有些硬體暫存器必須以 32-bit 來存取；對於這些暫存器，我們就不應該交給 *Simulate_IO* 做 byte IO 模擬。

使用 *Dispatch_Byte_IO* 巨集可以簡化寫碼工作，成爲下面這樣的骨幹：

```
BeginProc IOCallback, locked
        Dispatch_Byte_IO Fall_Through, byteout
        in      al, dx
        ret
byteout: out     dx, al
        ret
EndProc  IOCallback
```

在這裡我走了一條捷徑：以 *Fall_Through* 代替 "byte input routine label"。以這個關鍵字來代替 labels，可以壓制明白的（explicit）JMP 指令，於是提昇效率（張訓賓註：當 IO operation 爲 byte input 時，程式接著執行 macro 的下一行，省去一個 JMP 指令）。我也寫了個簡單的函式，用來處理 *Dispatch_Byte_IO* 無法交給 *Simulate_IO* 去做的那兩種情況（譯註：前述的 BYTE_INPUT 和 BYTE_OUPUT）。注意，VMM 已經將 port 位址設定給 DX 暫存器（多麼巧妙呀，IN 和 OUT 正需要 DX 內含位址！），所以你可以只是發出 IN 或 OUT 指令，然後就回返。程式碼效率是「爲什麼以 assembly 語言來寫 I/O callbacks 會比較好」的另一個理由。

以 assembler 而非 C 來撰寫一個 I/O callback，最重要的理由是爲了直接 JMP 到 *Simulate_IO*。一如我所說，*Simulate_IO* 期望獲得「如同一個 I/O callback 所獲得的」相同暫存器內容，其中包括 EBP。而我知道的所有 C 編譯器都使用 EBP 指向目前函式

的 stack frame，透過它來存取參數和函式的 auto 變數。因此如果你想從 C callback 函式中跳到 *Simulate_IO*，精確的寫碼過程是無法避免的。如果你決定以 C 來寫碼，以下片段可能有用（節錄自我以 VToolsD 完成的一個 driver）：

```
DWORD __stdcall IOCallback(VMHANDLE hVM, DWORD IOType,
   DWORD Port, PCLIENT_STRUCT pRegs, DWORD Data)
   {                           // IOCallback
   if (IOType == BYTE_INPUT)
      {                         // read character
      char ch;
      <>[read input somehow into "ch"]
      return ch;
      }                         // read character
   else if (IOType == BYTE_OUTPUT)
      {                         // write character
      <>[output LOBYTE(Data) somehow]
      return 0;
      }                         // write character

   _asm mov eax, Data
   _asm mov ebx, hVM
   _asm mov ecx, IOType
   _asm mov edx, Port
   _asm push ebp
   _asm mov ebp, pRegs

   VxDCall(Simulate_IO);

   _asm pop ebp
   }                           // IOCallback
```

注意，你重新設定 EBP -- 就在呼叫 *Simulate_IO* 之前，因爲你不會在重新設定之後再存取參數了。此外，你是以「呼叫」而非「跳躍」的方式進入 *Simulate_IO*，因爲它（譯註：指的是 *Simulate_IO*）最後的 RET 指令會引起大破壞 -- 如果執行於你所接收的相同的 stack 身上的話。最後，你得小心儲存並恢復 EBP 暫存器，如此一來你的函式的 epilog 碼才能夠正確運作（epilog 碼內含一個 LEAVE 或是一個 MOV ESP, EBP 指令）。

模擬一個 Device

有時候你會想要在你的 I/O callback 函式中模擬一個 device，而不只是簡單地爲應用程式執行 IN 和 OUT 指令。常見的理由是因爲「效率」。有一些 devices 可以輕易產生遠比 Windows 95 所能容忍（而不遺失資料）的極限還快的中斷(s) -- 如果 Win95 還必須先切回 ring3 ISR 的話。標準的 serial port 就是一個例子。當它以適度的速度（以今天的標準來說是 9600 baud）收到資料，port 會每 950 毫秒（ms）產生一個中斷。軟體必須在下一個中斷之前從晶片中將資料讀走，否則就得承擔資料遺失的風險。但是 Windows 95 不可能在無落後或是沒有怠慢其他工作的情況下，處理中斷並回到 ring3 完成輸入動作。

Windows 95 的虛擬通訊裝置（virtual communications device，VCD）driver 就是用來解決這種 serial ports 的潛在問題：它完全在 ring0 處理中斷，並且模擬應該是由應用程式所做的 serial port I/O 動作。這兩個工作在早期 Windows 版本中是靠兩個 VxDs（VCD 和 COMBUFF）完成。只要收到一個 data byte，VCD 立刻將這個 byte 讀取到它自己的緩衝區中，然後釋放 serial 晶片給下一個 byte 使用。Ring3 碼在自己最適當的時機執行一個 IN 指令，取得下一個 byte。VCD 會捕捉（traps）這個指令並從其內部緩衝區傳回下一個 byte。VCD 以類似手法來緩衝輸出資料。

模擬 I/O port 的另一個理由是爲了產生假想硬體。新型通訊卡廠商可能會使用 VxD，將「被導引至標準 8250-class serial 晶片」的 I/O，轉換爲對自己硬體的一個 requests。這種模擬的結果就好像把 serial port interface 視爲一個非傳統方法實作出來的 API。

這裡有一個 VxD 實例，藉由 "trapping a port" 來模擬硬體設備。這段碼的擴充版本（出現於下一節），附於光碟的 \CHAP13\PORTVIRTUALIZATION 目錄中）。被模擬的 device 有一個唯讀的 port 在位址 1234h 處。我要讀這個 port 以取出字串 "Hello, world!" 的連續字元：

```
name  myvxd
.386p
include vmm.inc
include debug.inc
```

```
Declare_Virtual_Device MYVXD, 1, 0, MYVXD_control, \
       Undefined_Device_ID, Undefined_Init_Order

Begin_Control_Dispatch MYVXD
Control_Dispatch Sys_Dynamic_Device_Init, \
   OnSysDynamicDeviceInit
Control_Dispatch Sys_Dynamic_Device_Exit, \
   OnSysDynamicDeviceExit
End_Control_Dispatch   MYVXD

VxD_LOCKED_DATA_SEG
data    db    'Hello, world!', 0
pdata   dd    offset32 data
VxD_LOCKED_DATA_ENDS


BeginProc OnSysDynamicDeviceInit, locked
        mov   esi, offset32 IOCallback
        mov   edx, 1234h
        VMMCall Install_IO_Handler
        ret
EndProc OnSysDynamicDeviceInit

BeginProc OnSysDynamicDeviceExit, locked
        mov   edx, 1234h
        VMMCall Remove_IO_Handler
        ret
EndProc OnSysDynamicDeviceExit

BeginProc IOCallback, locked
        Dispatch_Byte_IO Fall_Through, byteout
bytein:
        mov   esi, pdata
        xor   eax, eax
        lodsb
        test  al, al
        jnz   @F
        mov   esi, offset32 data
@@:
        mov   pdata, esi
byteout:
        ret
EndProc  IOCallback

        end
```

下面是個小測試程式，你可以用來驗證上面的 VxD 範例的正確性：

```
D:\CHAP13\PortVirtualization>debug testport.com
File not found

-a 100
30CD:0100 mov di,200
30CD:0103 mov dx,1234
30CD:0106 in al,dx
30CD:0107 test al,al
30CD:0109 jz 10e
30CD:010B stosb
30CD:010C jmp 106
30CD:010E mov byte ptr [di], 0d
30CD:0111 mov byte ptr [di+1], 0a
30CD:0115 mov byte ptr [di+2], 24
30CD:0119 mov dx,0200
30CD:011C mov ah,9
30CD:011E int 21
30CD:0120 mov ax,4c00
30CD:0123 int 21
30CD:0125
-g
Hello, world!
```

這個測試程式不斷在 106h 位置處執行 IN 指令，直到讀到一個 null byte 爲止。然後就使用 INT 21h, function 09h 將字串輸出到顯示幕上並結束生命。

現在讓我們追蹤 TESTPORT 搭配 VxD 對 port 1234h 的 trapping 動作。每次 TESTPORT 嘗試在 106h 位置執行一個 IN 指令，CPU 就諮詢 "I/O privilege mask" 並決定 port 1234h 是 "trapped"。於是會引起一個 general protection fault，使得控制權最終移轉到 MYVXD 的 *IOCallback* 函式。此時 ECX 暫存器中的 flag bits 將會等於 BYTE_INPUT，因爲被捕捉的指令只是一個 simple IN，對象是 AL 暫存器。EDX 暫存器會等於 1234h。其他暫存器無關緊要。*Dispatch_Byte_IO* 巨集會檢查 ECX 並因爲失敗而到達 byte input 模擬碼（沒有其他情況會在 TESTPORT 程式中發生）。爲了模擬 input 動作，callback 函式將一個靜態指標（*pdata*）設定給 "next input byte"（由 ESI 掌管），然後以 LODSB 指令同時設定 next byte 並令指標前進。

如果 input byte 是 0，表示我們到達了字串的終點，應該重新將資料指標設回原起始處。這麼一來，下次我們再呼叫 TESTPORT 時，就可以再一次取得完全字串。最後，callback 函式利用 AL 傳回新的 input byte，VMM 會負責將這個 data byte 放進 client AL 暫存器 -- 在回返至 VM 之前。

搶奪（Contention）情況的處理

通常，一次應該只有一個 VM 存取一個特定的 I/O port。提供搶奪（競爭）解決方案，是一個 trapped port 的 I/O callback 函式的另一功能。欲解決搶奪的問題，你必須持續追蹤目前擁有 device 的是哪一個 VM。你只能夠允許目前的擁有者使用這個 device。如果其他 VM 嘗試存取這個 device，你必須告訴終端使用者這個問題，並允許他（或她）決定怎麼辦。*SHELLResolve_Contention* 是一個用來和終端使用者交談的 service。在詢問完使用者的決定之後，你可能需要做某些殘酷的行爲，例如結束某個 VM，以便中止一個可能會永不終止的 I/O service 要求。

下面是稍早的 I/O callback 函式的擴充版，用來處理所謂的搶奪（競爭）問題：

```
BeginProc IOCallback, locked
        cmp   ebx, owner
        je    okayio
        cmp   owner, 0
        jne   contend
        mov   owner, ebx
        jmp   okayio

contend:
        push  eax
        mov   eax, owner
        mov   esi, offset32 MYVXD_DDB + DDB_Name
        VxDCall SHELL_Resolve_Contention
        pop   eax

        jc    cantresolve
        cmp   ebx, owner
        je    cantresolve
        mov   owner, ebx
```

```
okayio:
        Dispatch_Byte_IO Fall_Through, byteout

bytein:
        mov   esi, pdata
        xor   eax, eax
        lodsb
        test  al, al
        jnz   @F
        mov   esi, offset32 data
@@:
        mov   pdata, esi

byteout:
        ret

cantresolve:
        VMMCall Crash_Cur_VM
        ret
EndProc  IOCallback
```

13-6 說明解決方案背後的邏輯。基本想法是檢查 current VM（其 handle 是 I/O callback 函式的一個參數，放在 EBX）是否爲 device 擁有者，或是否沒有任何一個 VM 擁有該 device。不管哪一種情況，都可以順利繼續。如果其他某些 VM 擁有這個 device，我們呼叫 *SHELL_Resolve_Contention* 讓使用者來決定誰出線。回返時，carry flag 用來表示是否紛爭已解決，如果是的話，EBX 將內含這場競爭的勝利者（VM）的 handle。如果原來的擁有者勝利，我們必須做某些事情，阻止 current VM 繼續存取此 port。此例中我只是將這個落敗的 VM 強制結束掉。沒有任何訊息告訴使用者發生了什麼事，所以你最好在一個真正的 driver 中爲此做一點親和性動作。例如，你可以呼叫 *SHELL_Message* 來顯示一些說明。

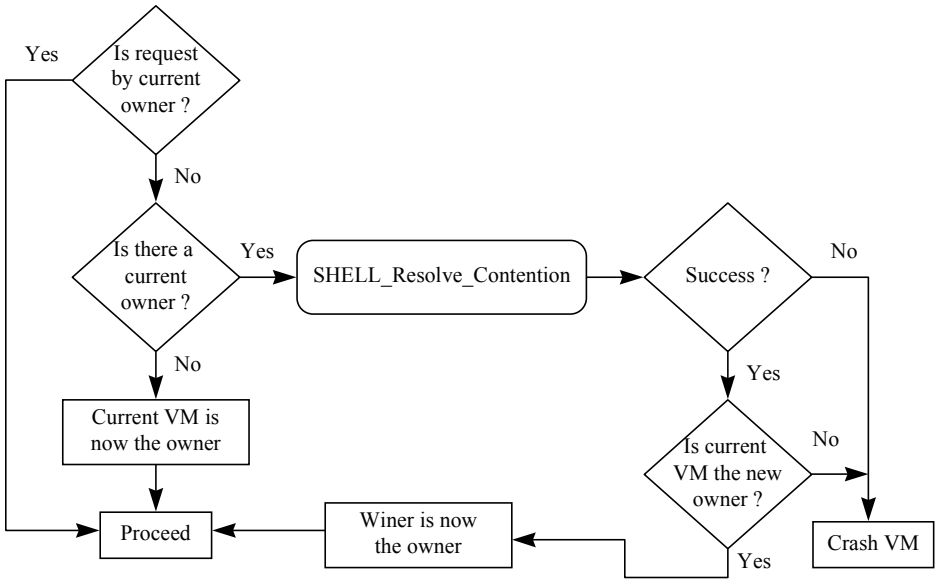
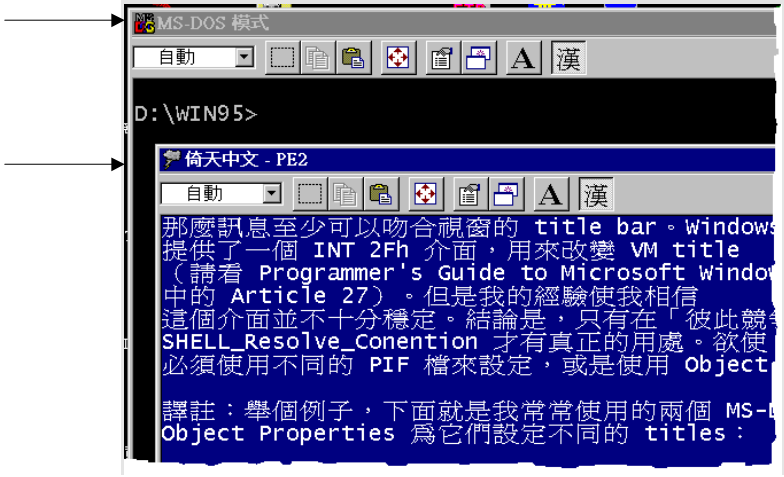


圖 13-6　搶奪（競爭，Contention）解決方案的邏輯

爲了測試「搶奪解決方案」，你可以在兩個不同的 MS-DOS 視窗中執行 TESTPORT。測試程式可以在第一個 MS-DOS 視窗中正常運作。當你嘗試在第二個 MS-DOS 視窗中做實驗，**圖 13-7** 便顯示出 *SHELL_Resolve_Contention* 所展現的對話盒。你能夠告訴我你應該點選哪一個 MS-DOS 視窗（代表一個 VM）嗎？（答案：上面那個是目前的擁有者）。對話盒中的訊息是如此地不帶價值，因爲 SHELL 所顯示的是 VM 的 title 字串。如果 SHELL 所顯示的也包含應用程式的 title 字串，就會比較好一些，那麼訊息至少可以吻合視窗的 title bar。Windows 95 爲 MS-DOS 應用程式提供了一個 INT 2Fh 介面，用來改變 VM title （請看 *Programmer's Guide to Microsoft Windows 95* [Microsoft Corporation, 1995] 中的 Article 27）。但是我的經驗使我相信這個介面並不十分穩定。結論是，只有在「彼此競爭的 VMs 有不同的 titles」情況下，*SHELL_Resolve_Conention* 才有真正的用處。欲使 VMs 有不同的 titles，必須使用不同的 PIF 檔來設定，或是使用 Object Properties。

譯註：舉個例子，下面就是我常常使用的兩個 MS-DOS VMs，我利用 Object Properties 爲

它們設定不同的 titles：



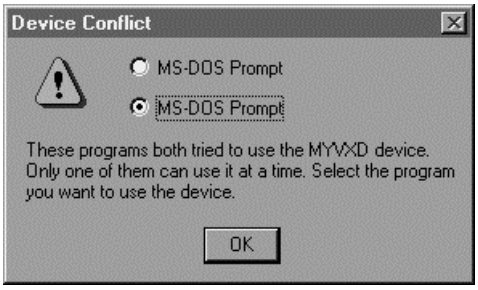


圖 13-7 Contention 解決方案的對話盒

Enabling/Disabling Trapping

「I/O port 虛擬化」的一個比較精鍊的方法，與 trap state 有關。一個 trap state 若不是 on 就是 off。VMM 爲每一個 port 和 VM 維護一個 trap state。你可以在某個 VM 中爲某個 port 改變其 trap state，方法是呼叫 *Enable_Lcal_Trapping* 或 *Disable_Local_Trapping*。你也可以一次爲所有的 VM 改變 trap state，方法是呼叫 *Enable_Global_Trapping* 或 *Disable_Global_Trapping*。經由呼叫某個 global trapping calls 所建立起來的 trap state，將是後續所產生的每一個 VM 的最初 state。

張訓賓註：由於 VGA driver 的效率非常重要，所以通常的作法是：VDD 建立 global IO trapping，而將 system VM（即 display driver 所在的 VM）的 IO trapping 給 disable 掉。

你可能會懷疑爲什麼需要控制 trap state。答案是，如果一個 ring3 driver 的 IN/OUT 指令直接奔至硬體，而不是先被一個 VxD 捕捉，這個 driver 的執行會快許多。因此，許多情況下，一個 driver 會在它「視之爲 device 之擁有者」的 VM 中，爲它自己的 ports 將 local trapping "disable" 掉。在其他 VMs 中仍保持 local trapping enabled，這個 driver 於是就可以信心滿滿地說它確實可以辨識出這個 device 的 contention（搶奪）狀態。

將硬體中斷虛擬化

Virtual Programmable Interrupt Controller Device（VPICD）負責處理 Windows 95 的所有硬體中斷。它並不直接 "hooking" 一個中斷向量以接管一個 device 中斷，而是呼叫一個 VPICD service 將 IRQ 虛擬化。表 **13-4** 列出所有涉及硬體中斷處理的 VPICD services。在深入這些 services 的細節之前，你必須先瞭解當 VPICD 收到一個硬體中斷時通常會奉行的程序。你也必須長記在心，VPICD 所在的 Windows 95，是一個以 VM 爲中心的世界。也就是說，VPICD 的主要任務是將硬體中斷繞行（route）到坐落於各 VM 內的 device drivers 手上，並允許它們有效運作，就好像沒有 Windows 一樣。VPICD 也爲了 VM 的 drivers，將 interrupt controller 虛擬化，允許 drivers "mask" 和 "unmask" 中斷申請（interrupt requests）、操作虛擬的 in-service 暫存器...等等等。

Service	說明
VPICD_Auto_Mask_At_Inst_Swap	引起一個 IRQ 在一個 instance data swap 期間被 masked。
VPICD_Begin_Inst_Page_Swap	通知 VPICD 說一個 instance data swap 開始進行（只用於內部）。
VPICD_Call_When_Hw_Int	安裝一個 callback 函式，以備每次硬體中斷時被呼叫。
VPICD_Clear_Int_Request	爲一部 VM 清除一個 virtual interrupt request。

Service	說明
VPICD_Convert_Handle_To_IRQ	爲一個虛擬化的 IRQ 取出 IRQ 號碼
VPICD_Convert_Int_To_IRQ	決定哪一個 IRQ 對應於一個已知 VM 中的一個已知中斷號碼。
VPICD_Convert_IRQ_To_Int	決定哪一個中斷號碼對應於一個已知 VM 中的一個已知 IRQ。
VPICD_End_Inst_Page_Swap	通知 VPICD 說一個 instance data swap 完成了（只用於內部）。
VPICD_Force_Default_Behavior	將一個 IRQ 反虛擬化。
VPICD_Force_Default_Owner	爲一個 IRQ（以 IRQ 號碼識別之）設定 global/local 狀態、擁有權、以及某些預設處理選項。
VPICD_Get_Complete_Status	取得一個虛擬化的 IRQ 的 complete status。
VPICD_Get_IRQ_Complete_Status	取得某個 IRQ（以其號碼識別之）的 complete status。
VPICD_Get_Status	取得一個虛擬化的 IRQ 的 complete status 中最常被使用的部份
VPICD_Get_Version	取得 VPICD 的版本號碼、PIC 組態（configuration）、以及對 IRQ 的最大支援個數。
VPICD_Get_Virtualization_Count	根據 IRQ 編號取得此 IRQ 被虛擬化的次數。
VPICD_Phys_EOI	表示一個中斷的處理完畢（實際上是 "unmasks" IRQ，而不是送出一個 EOI 命令）
VPICD_Physically_Mask	Masks 一個被虛擬化的 IRQ。
VPICD_Physically_Unmask	Unmasks 一個被虛擬化的 IRQ。
VPICD_Set_Auto_Masking	Enables 一個虛擬化 IRQ 的 "automatic masking"。如此一來，實際的（physical）IRQ 只有在某個 VM 將它 "unmasked" 時才會被 "unmasked"。
VPICD_Set_Int_Request	爲一個虛擬化 IRQ 在某個指定的 VM 中設定一個虛擬的 interrupt request（模擬一個中斷）。
VPICD_Test_Phys_Request	爲一個虛擬化 IRQ 測試實體之 in-service 暫存器.
VPICD_Virtual_EOI	爲一個虛擬化 IRQ 在某個指定的 VM 中執行一個 virtual EOI 命令

Service	說明
VPICD_Virtualize_IRQ	爲一個 IRQ 建立起非預設的處理常式。
VPICD_VM_SlavePIC_Mask_Change	一個內部 service，當 VM "masks" 或 "unmasks" IRQ 2 時被使用。如果你想知道這情況何時會發生，可使用 *Hook_Device_Service*。

表 13-4　VPICD services

預設的 IRQ 處理程序

圖 13-8 表現出一個硬體中斷的預設處理動作的大體程序。硬體中斷向量在 IDT entries 中的編號爲 50h~5Fh。這些中斷號碼與標準真實模式的向量號碼不同，因爲 VPICD 將 master PIC 和 slave PIC 的基底向量分別改爲 50h 和 58h。改變基底向量可以使 VPICD 比較容易並且比較快地識別出硬體中斷和軟體中斷或處理器異常（processor exception）之間的差異 -- 它們一致使用相同的真實模式向量號碼。否則，要識別出一個 page fault（exception 0Eh）和 IRQ 6 之間的不同就比較困難；反之，INT 56h 就毫無疑問地充任 IRQ 6 的中斷向量。

INT 50h~5Fh 的一份備忘錄　中斷 50h~5Fh 在 IDT gate entries 之中的 Descriptor Privilege Level （DPL）都是 0。如果 ring3 程式發出一個 INT 指令，指明這些中斷之一，就會發生一個 general protection fault。於是 VMM 會檢查並發現，這個程式正嘗試執行一個 INT 中斷，於是將控制權繞行（routes）到此一中斷的軟體中斷處理常式去。由於沒有任何一個 ring0 程式應該以一個 INT 指令使用這些中斷，所以 VPICD 可以輕鬆假設，如此的情況暗示著一個硬體中斷。我在本章範例程式中使用了這一事實，我在一個 VxD 中利用 INT 5Ch 來模擬一個 IRQ 12，用以測試一個 drivers 對 IRQ 12 的虛擬化。你千萬不要在一個發行的產品中冒充硬體中斷，因爲 Windows 未來版本並不保證會像 Windows 95 這樣處理硬體中斷。

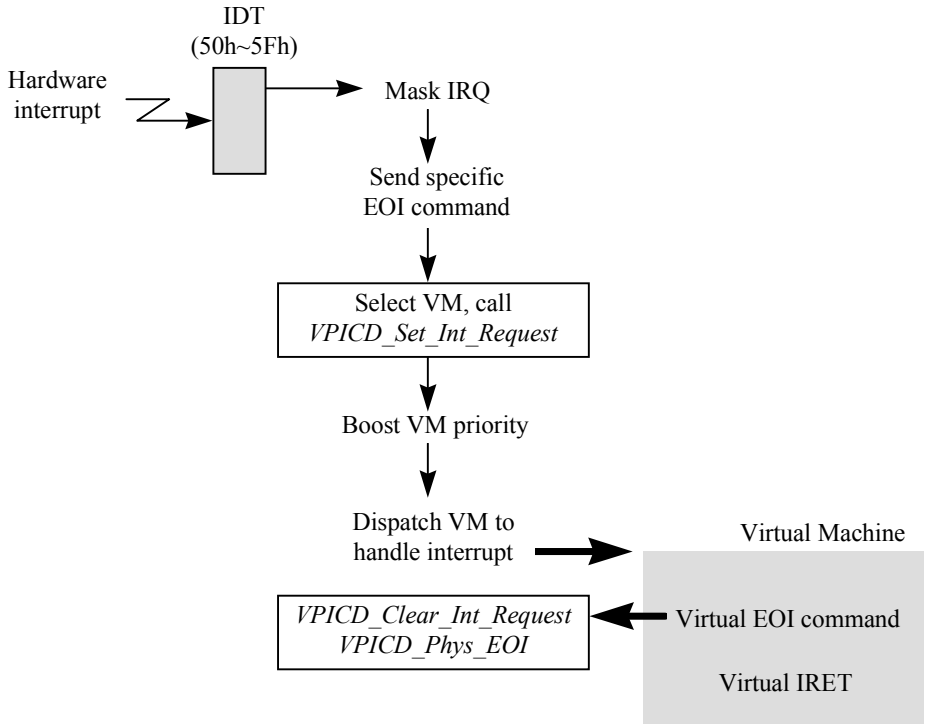


圖 13-8　處理一個硬體中斷時的預設程序

VPICD 內含每一個硬體中斷的第二層中斷處理常式。每一個處理常式會透過對 PIC 的 IMR 的寫入動作，將其 IRQ "mask off"。然後它會發出一個特定的 EOI 命令，將中斷清除。從本章稍早的討論可知，這兩個步驟允許較低優先權或較高優先權的中斷發生，但是阻止在相同的 IRQ 身上發生遞迴中斷。經過實際量測，VPICD 的第一層和第二層中斷處理常式，再加上 VMM 核心中的共通的狀態儲存（state-saving）程式碼，在到達 EOI 這一點之前共執行了 24 或 31 個指令 -- 視中斷來自 V86 模式或保護模式而定。在一顆 i486 CPU 上，這些指令消耗大約 100~150 個 clock cycles。

VPICD 接下來會選擇某個 VM 來處理這個中斷。VM 會不會被選中，視 IRQ 是否被視爲 global 或 owned（已被擁有）而定。所謂 global IRQ，是指當 Windows 95 啓動時它是 "unmasked"（也就是說，IMR 爲它內含一個 0-bit）。當 Windows 95 啓動，這個 IRQ

的真實模式 ISR 已經就定位，並佔用 global V86 記憶體。換句話說，其 ISR 存在於每一個 VM 之中，因此也就不在乎由哪一個 VM 來處理此一中斷。於是，VPICD 通常會選擇 current VM 來處理。然而，如果其他某些 VM 碰巧已經聲稱（擁有）了 critical section，VPICD 會將中斷處理權給予 critical section 的擁有者。

所謂 local IRQ，在 Windows 95 啓動之後才被 "unmasked"。某些 VM 碼必定會對 IMR（不論是 master PIC 的或 slave PIC 的）做寫入動作。VPICD 會捕捉（traps）這些寫入動作，於是就知道 VM 什麼時候嘗試操控 PIC，並使「新被 unmasked 的 IRQ」成爲此 VM 的 local IRQ。預設情況下，VPICD 也會將一個 local IRQ 上的中斷(s) 送至擁有者（VM）手中。

你可以經由呼叫 *VPICD_Force_Default_Owner* 來改變 IRQ 的分類，從 global 改爲 local，反之亦可。該 service 的一個參數並允許你導引 VPICD 選擇 "execution-focus VM"（而非 "current VM"）來處理一個 global IRQ -- 如果沒有任何 VM 擁有 critical section 的話。例如，我有一個客戶，他需要確保一個中斷總是在 System VM 中被服務，因爲他的碼依賴一個「DPMI 真實模式 callback」來觸及 Windows 應用程式。由於這個 IRQ 在 Windows 95 啓動時是 "unmasked"，VPICD 的預設處理方式是將此 IRQ 視爲 global，並將它繞行（route）到 current VM 去。簡單的 VxD 只需呼叫 *VPICD_Force_Default_Owner* 並給予 System VM handle，就可以解決這個問題。相反的，如果 BIOS 對軟碟 IRQ 的 "unmask" 動作失敗，問題就會發生，所以 Virtual Floppy Driver 利用這個 service 來強迫軟碟 IRQ 成爲 global。

選擇了一個 VM 爲此中斷服務之後，VPICD 使用它自己的一個 service：*VPICD_Set_Int_Request*，將此 VM 標記爲「在此 IRQ 上有一個 pending 中斷」。這個中斷將被模擬至 VM 中（使用一般的真實模式向量號碼）-- 如果不是立刻，就是當以下所有情況都爲真之後：

- 中斷在此 VM 之中必須是被 "enabled" 的。VMM 會捕捉(traps)來自 ring3 保護模式程式的 CLI 和 STI 指令，並記錄一個 virtual enable state，那並不會與真實機器的 interrupt flag 有任何關連。由於 V86 碼的 IOPL 通常是 3，一個

來自 V86 模式的 CLI 指令一般而言會 "disables" 真實機器，不過有些情況下一個 V86 程式可能只是被 virtually disabled 而已。不論哪種情況，對硬體中斷模擬而言，virtual interrupt state 才是重要的。

- IRQ 在此 VM 之中是未被 "masked" 的（藉由寫入一個 1-bit 到其 virtual IMR 中）。VPICD 爲每一個 VM 捕捉（traps）PIC control ports 並維護 virtual mask 和 in-service registers。當某個 IRQ 正被 VPICD 服務，它將是被實際 "masked" 的。不過，中斷是否能被模擬，是由 virtual mask state 決定。
- 沒有任何「較高優先權之 IRQ」可以在此 VM 中提供服務。這是模仿真實機器的運作方式。
- VM 必須準備好要執行了，意思是它並非被虛懸（suspended）或被凍結（blocked）於一種「不允許中斷」的情況下。還記得嗎，同步化基礎元件（synchronization primitives）如 *Begin_Critical_Section*, *Wait_Semaphore* 等等都有一個 flag 參數，允許 VM 接受中斷 -- 甚至在被凍結（blocked）的情況下。
- 中斷一定不曾因爲某些 VxD 呼叫 *VPICD_Clear_Int_Request* 而取消。

如果呼叫 *VPICD_Set_Int_Request*，VM ISR 最終會獲得控制權。基本上，ISR 會在某一時刻企圖送出一個 EOI 命令給 PIC，最後將執行一個 IRET 指令以回返至被中斷的那一點。藉由對 PIC control ports 的捕捉（trap），VPICD 在 EOI 命令那一點上橫加干涉。當 virtual EOI 命令發生，VPICD 呼叫它自己的多個 services：*VPICD_Clear_Int_Request* 和一個 *VPICD_Phys_EOI*（其名稱容易令人誤解）*VPICD_Clear_Int_Request* 清除來自 VM 的 virtual interrupt request。清除這個 request 是必要的，因爲 VPICD 模擬了一個 level-triggered PIC，只要 interrupt request 仍然存在，就會保持 interrupting 狀態。由於真正的 physical EOI 命令在中斷之後即刻發生，所以 *VPICD_Phys_EOI* 並不發出一個 EOI 命令。取而代之的是，它將 physical IRQ "unmasks" 掉。（*VPICD_Physically_Unmask* service 也會 "physically unmasks" 一個 IRQ。這兩個 services 的差異在於它們如何維護內部資料結構，所以誰也不能取代誰）

VPICD 也對來自 ring3 ISR 的 IRET 加以干涉。捕捉（trapping）這個 IRET 使 VPICD 得以取消（undo）所謂優先權提昇（下一段即將描述）。此外，VPICD 允許相同的 VM

處理同一個 IRQ 上的任何中斷，只要是發生於 IRET 之前（否則可能會導至對於 "cascading" 中斷的不正確處理）。捕捉（trapping）IRET 使得 VPICD 清除它對「哪個 VM 正在處理此一中斷」的記憶。如此一來普通的 VM 選擇規則就可以施行於同一 IRQ 的下一個中斷了。

儘快爲硬體中斷服務，通常是個好觀念，所以 VPICD 將那個「即將處理中斷」的 VM 的優先權提昇至甚大值 TIME_CRITICAL_BOOST。這個提昇量比系統中的任何其他可能發生的提昇量都大。一個 VM 經此提昇之後，幾乎可以保證在下一次機會立刻獲得執行權（除非有數個 VMs 統統正在處理不同的中斷，這種情況下排程器會讓它們共享 CPU）。這次提昇一直維持到 virtual ISR 回返或直到 500 毫秒（ms）結束爲止。

將一個 IRQ 虛擬化

如果你要改寫一個 IRQ 的預設處理方式，你可以利用 *VPICD_Virtualize_IRQ* 提供一表格的函式指標，允許你在你所定義的點上加以干涉。這個 service 可以這樣呼叫：

```
include vpicd.inc
...
mov     edi, offset32, vid      ; virtual irq descriptor
VxDCall VPICD_Virtualize_IRQ
jc      error                   ; jump if error
mov     hIrq, eax               ; handle of virtualized IRQ
```

圖 13-9 顯示 Virtual IRQ Descriptor（VID）結構的格式，它用來描述 IRQ 的虛擬化。*VID_IRQ_Number* 和 *VID_Hw_Int_Proc* 兩欄位是必要的；其他欄位可有可無，如果不需要就應該被設爲 0。稍後我將討論 VID 結構中各式各樣的 flag bits 意義。*VPICD_Virtualize_IRQ* 的傳回值是一個 IRQ handle，你應該將它儲存於一個全域變數中，以便稍後使用。這個 handle 對許多 VPICD services 而言是個參數，而這些 services 中最值得注意的當屬 *VPICD_Force_Default_Behavior*，它被用來將一個 IRQ 解除虛擬化（unvirtualize），這應該是你的 driver 的清理工作（cleanup）的一部份。

VPICD 會在**圖 13-10** 所指示的各點，呼叫你的虛擬化函式。如果你不改寫 *Virt_Int_Proc*,

EOI_Proc, *IRET_Proc* 等函式中的一個或多個（改寫方式是提供一個非零指標），當相對的 events 發生，VPICD 不會做任何事情。當你正將一個 IRQ 虛擬化，以允許它在一個 VM 中被服務，你通常會提供一個 *Hw_Int_Proc* 函式和一個 *EOI_Proc* 函式。（一如我們將在本章最後一節所見，當你虛擬化一個 IRQ 以便在 ring0 提供一個中斷處理，你只需要一個 *Hw_Int_Proc* 函式，因爲你不會把這個中斷反映（reflecting）至一個 VM 上頭）。

```
VPICD_IRQ_Descriptor STRUC
   VID_IRQ_Number        dw  ?      ; 00 number of IRQ (0-15) (required)
   VID_Options           dw  0      ; 02 option flags (listed below)
   VID_Hw_Int_Proc       dd  ?      ; 04 address of hardware interrupt
                                    ; procedure (required)
   VID_Virt_Int_Proc     dd  0      ; 08 virtual interrupt procedure
                                    ; (optional)
   VID_EOI_Proc          dd  0      ; 0C EOI procedure (optional)
   VID_Mask_Change_Proc  dd  0      ; 10 mask-change procedure (optional)
   VID_IRET_Proc         dd  0      ; 14 IRET procedure (optional)
   VID_IRET_Time_Out     dd  500    ; 18 maximum time to leave VM's priority
                                    ; boosted
   VID_Hw_Int_Ref        dd  ?      ; 1C reference data for hardware
                                    ; interrupt procedure
VPICD_IRQ_Descriptor ENDS

; Flags in VID_Options:

VPICD_OPT_READ_HW_IRR      EQU 0001h     ; Read physical IRR to see if the
                                         ; interrupt is in service
VPICD_OPT_CAN_SHARE        EQU 0002h     ; Virtual IRQ can be shared
VPICD_OPT_REF_DATA         EQU 0004h     ; Pass reference data to hardware
                                         ; interrupt procedure
VPICD_OPT_VIRT_INT_REJECT   EQU 0010h    ; Don't allow interrupt to be
                                         ; reflected to ring three
VPICD_OPT_SHARE_PMODE_ONLY  EQU 0020h    ; Interrupt can be shared only if
                                         ; handled entirely in ring zero
```

圖 13-9　Virtual IRQ Descriptor (VID) 結構，用來傳遞給 VPICD_Virutal_IRQ

這一節的剩餘部份，我要說明當你虛擬化一個 IRQ 時，可以提供的各種 callback 函式。

硬體中斷函式（Hw_Int_Proc）　一個硬體中斷函式會在硬體中斷發生之後很快收到控制權。VPICD 會先將相關的 IRQ "masked" 起來，並發出一個特定的 EOI 命令給 PIC。真實機器的中斷會被 disabled。EAX 內含 virtual IRQ handle，那是先前被 *VPICD_Virtualize_IRQ* 傳回的，EDX 內含你指定於 VID 結構中的參考資料（如果有的話）。EBX 出人意表地指向 current VM control block，EDI 指向 current thread control block，EBP 指向 current VM 的 client registers。幾乎沒有理由你需要去參考 EBX, EDI, 和 EBP，因爲對於處理一個硬體中斷而言，被中斷者（VM）的身份和 context 是無關緊要的。

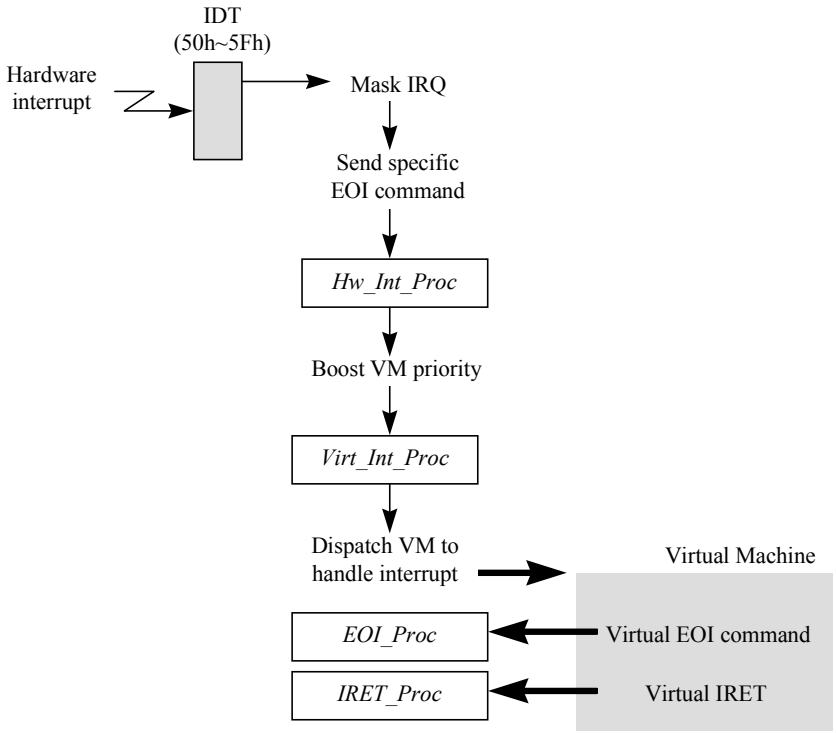


圖 3-10　處理一個被虛擬化後的 IRQ

你的硬體中斷函式如果將中斷處理好了，應該在回返時將 carry flag 清除。如果你在你的 VID 結構中指示說這個 IRQ 是可被共享的(將 VID_OPT_CAN_SHARE flag 設立即

可），那麼你可以在回返時令 carry flag 設立，表示你的函式並未處理這個中斷。這種情況下，VPICD 會呼叫此 IRQ 服務常式串列的下一個處理常式。

VPICD_OPT_SHARE_PMODE_ONLY Flag　呼叫 *VPICD_Virtualize_IRQ* 時，你可以在 Virtual IRQ Descriptor 中設立此一 flag，表示這個 IRQ 只有在「處理常式於 ring0 執行時」才可被共享。（這或許是 "IRQ sharing" 的唯一辦法）。在零售版 Windows 95 中這個 flag 不做任何事情，在除錯版 Windows 95 中它會產生一個除錯警告 -- 如果有人呼叫 *VPICD_Set_Int_Request* 在 ring3 中爲此 IRQ 服務的話。你必須確定那不會發生，因爲硬體中斷函式不會知道回返時 carry flag 是設立還是清除；它不知道是否要設立 carry flag，因爲它不知道 ring3 處理常式是否真的會處理這個中斷。

雖然你的硬體中斷函式可以使中斷 "enabled"，它必須只使用 page-locked code 和 data，並且只能呼叫那些 "asynchronous" VxD services。由於 VPICD 已送出一個 physical EOI 命令給 PIC，所以不論較高優先權或較低優先權的中斷，都可以發生 -- 如果你的函式將中斷 "enables" 的話。你的 IRQ 上沒有任何中斷可以發生，因爲 VPICD 已經把它 "masked" 掉了。雖然你的函式可能會因其他硬體而被中斷，你確定可以在內部中斷的硬體中斷函式回返之後，儘快重獲控制權。特別一提，VMM 並不處理 events，直到巢狀硬體中斷函式(s) 的最外圈回返。

將一個「你終究要反映（reflecting）至 VM 中」的 IRQ 虛擬化，一個常見的理由是爲了改變 VPICD 對於「用以處理中斷」之 VM 的選擇。舉個例子，假設你的 device 有一個擁有者（通常你會因爲 *Set_Device_Focus* 訊息或爲了一個搶奪（競爭，contention）演算法而指定擁有權），你或許希望這個中斷在擁有此 device 的 VM 中被服務。一個硬體中斷函式以下列作法完成這樣的願望：

```
BeginProc Hw_Int_Proc, locked
          cmp   owner, 0            ; anyone own the device?
          je    @F                  ; if not, reflect to current VM
          mov   ebx, owner          ; point to device owner
@@:       VxDJmp VPICD_Set_Int_Request ; set request for owner
EndProc  Hw_Int_Proc
```

這個函式的影響是將此中斷送往擁有 device 的 VM 中 -- 如果有的話。如果沒有任何 VM 擁有這個 device，此函式便將中斷送進 current VM 之中。如果你打算模仿 VPICD 的預設處理，而不只是將中斷送到 current VM 手中，你需要更多更詳盡的碼。

虛擬中斷函式（Virt_Int_Proc）　當 VPICD 即將要把一個硬體中斷反映（reflect）至一個特定的 VM 時，它會呼叫你的虛擬中斷函式。EAX 內含 IRQ handle，EBX 內含將收到這個中斷的 VM 的 handle。提供一個虛擬中斷函式的通常理由是，如此一來，你可以在這個虛擬處理常式的外圍包夾一個 critical section，如下所示：

```
BeginProc Virt_Int_Proc, locked
        xor     ecx, ecx
        VMMCall Begin_Critical_Section
        ret
EndProc  Virt_Int_Proc
```

如果你在虛擬中斷函式中聲稱擁有 critical section，你也應該提供一個 IRET 函式（譯註：稍後描述）以釋放這個 section。

提供 *Virt_Int_Proc* 函式的另一理由是，選擇性地阻止中斷被反映（reflected）到一個 VM 手上。VPICD_OPT_VIRT_INT_REJECT flag 會阻止任何中斷被反映。爲了只阻止「被選中的中斷」不要被反映，請讓該 flag 爲 0，並且在你的 *Virt_Int_Proc* 回返時將 carry flag 設立，以阻止反映（reflection）；或是將 carry flag 清除，以允許反映（reflection）。

中斷結束函式（EOI_Proc）　一個虛擬中斷的服務常式通常會送出一個 EOI 命令給 interrupt controller -- 當它已經準備好接受來自相同 device 的另一個中斷。面對一個使用 IRQ 0~7（master PIC 上的 interrupt request）的 device，ring3 中斷處理常式會發出一個 EOI 命令，像這樣：

```
out  20h, 20h   ; EOI for our IRQ on the master PIC
```

面對一個使用 IRQ 8~15（slave PIC 上的 interrupt request）的 device，ring3 中斷處理常式會發出一個 EOI 命令，像這樣：

```
out  0A0h, 20h  ; EOI for our IRQ on the slave PIC
out  20h, 20h   ; EOI for IRQ 2 to the master PIC
```

我要強調，我所顯示的是一個 ring3 中斷處理常式的所做所爲。除了 VPICD 以外，任何 VxD 都不應該將 EOI 命令直接送往 PIC。它應該利用 *VPICD_Phys_EOI* service 來表示一個中斷的完成。

VPICD 會捕捉（traps）這些 PIC 暫存器的讀寫，並呼叫你的 *EOI_Proc* 函式，並以 EAX 暫存器放置 IRQ handle，以 EBX 暫存器放置 current VM handle。通常你會清除這個 virtual interrupt request 並通知 VPICD 說這個中斷已經被處理完畢：

```
BeginProc EOI_Proc, locked
        VxDCall VPICD_Clear_Int_Request
        VxDJmp VPICD_Phys_EOI
EndProc  EOI_Proc
```

如果你忘記清除虛擬中斷，VPICD 會嘗試一次又一次地將此中斷反映（reflect）到 VM。這是因爲 VPICD 模擬了一個 level-triggered interrupt controller，它會要求一個中斷 -- 只要硬體的 request line 尚還保持在 asserted 狀態。

還記得嗎，儘管名稱頗令人困惑，但 *VPICD_Phys_EOI* 並不會真正做一個 physical EOI。取而代之的是，它會 "unmasks" physical interrupt。Physical EOI 應該在中斷老早之前、以及中斷後的極短時間發生。

「Mask 已改變」之處理函式（Mask_Change_Proc） 只要有一個 VM 嘗試改變你已經虛擬化之 IRQ 的 mask，VPICD 就會呼叫你的 *Mask_Change_Proc* 函式。函式被呼叫時 EAX 內含 IRQ handle，EBX 內含 current VM handle，而 ECX 內含一個 flag，表示此 IRQ 是否已被 masked（ECX 非零）或處於 unmasked 狀態（ECX 爲 0）。你可能會提供一個 *Mask_Change_Proc* 函式以管理你的 device 搶奪（競爭）問題。我們

已經討論過 device 搶奪和 port trapping 的相關問題，我就不再重複了。

請注意，沒有什麼方法可以讓你的 *Mask_Change_Proc* 函式將 virtual IMR 恢復回到 trapped 指令（製造成你的處理函式被呼叫）之前的狀態。

IRET 函式（IRET_Proc）　還有一個 callback 函式是你在虛擬化一個 IRQ 時可以指定的：IRET 函式。虛擬中斷服務常式最終會發出一個 IRET 指令以結束其對中斷的處理。VPICD 然後就呼叫我們的 IRET 函式 -- 如果你有提供的話。通常只有在你擁有一個 *Virt_Int_Proc* 函式的情況下，才會也供應一個 *IRET_Proc* 函式，而其目的是要恢復（undo）*Virt_Int_Proc* 的任何作爲。舉個例子，如果你的 *Virt_Int_Proc* 函式聲稱擁有 critical section，你的 *IRET_Proc* 函式就應該呼叫 *End_Critical_Section* 釋放它。

DMA 虛擬化

如果你因爲先前對 DMA 傳輸的設定與執行的種種討論而氣餒畏縮，你大概很擔心你的 VxD 必須爲了支援 DMA 傳輸而做許多工作，幸運的是，VDMAD 做掉了所有的工作。一個在 VM 中跑的 V86-mode driver，可以以一般方式來規劃 DMA 傳輸（只需寫入 DMA 的 page, address, count 等暫存器），並可經由對 DMA channel 的 "unmasking" 以及對 device 的規劃，將傳輸程序初始化。VDMAD 捕捉（traps）所有的 DMA controller ports，爲的是將 DMA 傳輸虛擬化。VDMAD 取得並鎖定涉及資料傳輸的 pages，並將虛擬位址轉換爲實際位址。如果必要，VDMAD 還提供它自己的緩衝區（16-MB 實際記憶體之下），並在必要的時候拷貝資料至（或從）VM 緩衝區。

VDMAD 可以處理一個真實模式 driver 所丟出的任何的標準 DMA programming 問題。唯一的例外發生於自動初始化（autoinitialization）模式。在 autoinitialization 模式中規劃一個 DMA channel，表示你希望每當這個 channel 將計數耗盡，位址和計數便會被自動恢復。一般情況下，這個模式僅被用來造成 channel 0 持續地重掃（refresh）記憶體。如果你規劃一個「自動初始化傳輸行爲」，並使用不連續緩衝區，或是超越了 16MB 限制，VDMAD 會拒絕傳輸，因爲簿記工作太複雜了，不值得。

然而，保護模式 ring3 程式不能夠以一般方式來使用 DMA。VDMAD 爲「保護模式 DMA」支援 Virtual DMA Standard（VDS）。VDS 規格可從 MSDN 獲得，我不打算在本書討論它。

你唯一需要爲了你的 device 而在意並關心的 DMA 問題是：VDMAD 將使用的 DMA 緩衝區的大小和位置。VDMAD 在處理 *Init_Complete* 訊息期間配置緩衝區。在 VDMAD 處理此一訊息之前的任何時刻，你都可以呼叫 *VDMAD_Reserve_Buffer_Space* 來指示你的 device 所需的緩衝區大小和最大實際位址。例如，假設你知道你的 device 的 V86-mode driver 將進行 8192 bytes 的 DMA 傳輸，那麼在你的 *Device_Init* 處理常式中，你應該告訴 VDMAD 這項需求：

```
        include vdmad.inc

BeginProc Device_Init, init
        mov     eax, 2          ; 8192 bytes = 2 pages
        xor     ecx, ecx        ; we don't care about buffer addr
        VxDCall VDMAD_Reserve_Buffer_Space
        ...
EndProc  Device_Init
```

順帶一提，很少 VxDs 會對 DMA 緩衝區的實際位置感興趣。這個緩衝區的位置主要視電腦使用什麼硬體 bus 而定，而 VDMAD 已在其演算法中考慮了這項因素。

進階的虛擬化技術

有兩個額外的虛擬化技術可以在特殊狀況下使用。撰寫一個普通的 16 位元 DLLs，在 ring0 提供中斷處理常式，是有可能的。寫雙模式（bimodal）中斷處理常式，在「任何中斷發生時 CPU 所處的模式」內執行，也是有可能的。這些技術的目的都是爲了將 ring 之間的轉換次數降到最低，以便改善高速 devices 的效率。前述兩項技術在 DDK 中都沒有良好的說明，不過 Karen Hazzah 的 *Writing Windows VxDs and Device Drivers*（R&D Press, 1995）內含一份卓越的討論，並附一份範例程式碼。

在我的觀念中，這兩個高階的中斷處理技術，基本上都是一種特異功能，爲的是讓你將

「其實應該重寫爲一個 VxD」的程式碼做轉換。這就是爲什麼我不打算在本書中進一步討論它們的原因。

撰寫 Ring0 Drivers

你在前面的冗長討論中所讀到的是 VxD 虛擬化 I/O 資源的作法。我有一個秘密要分享給你：你所獲得的知識恐怕已經老舊過時了。在所有美好可能的未來世界中，沒有 ring3 device drivers 的一席之地，你不必再爲它們的 port I/O、中斷處理、或是 DMA 傳輸規劃操心。如果不是爲了回溯相容，你或許根本就不會想到要虛擬化一個 device。

Microsoft 正在移往一個新架構，在其中，所有的 device drivers 都在 32 位元保護模式中跑，不再有真實模式的 drivers。這個架構是否最終會涵蓋 VxDs 的傳統型式，在我下筆此刻尚不清楚，但可以確定的是，至少會有一個過渡階段，其中由 VxDs 實作大部份的 device driver 功能。這一節我要討論如何撰寫一個純粹的 ring0 device driver.

一個現代的 ring0 driver 應該有以下組件（components）：

- 一個機制，用以連接 Configuration Manager，以便與 I/O 資源管理保持合作。你的組態函式（configuration function）的 *CONFIG_START* 處理常式應該取得系統分配給你的資源並將 device 初始化。你的 *CONFIG_TOP* 和 *CONFIG_REMOVE* 處理常式應該採取步驟來終止對資源的使用。
- 一個 VPICD 硬體中斷函式，用以處理任何指定的 IRQs 上的硬體中斷。這個中斷函式將完全處理這個中斷，不會反映（reflecting）至一個 VM 身上。
- 一個允許應用程式透過函式呼叫來和 device 交談的 API。這個 API 取代了「I/O port trapping」以及「模擬 memory-mapped devices 行爲的詳細方法」。（你或許願意允許應用程式直接存取 mapped memory，但是你可以提供 API 並讓應用程式呼叫，以此明白決定存取順序）。
- Ring0 碼，爲任何需要的 DMA 傳輸使用 VDMAD services。

注意 這一節所展現的範例涵蓋於書附碟片的 \CHAP13\RING0DMA 目錄中。如果你建造並執行之，務請小心，因爲如果你不小心使用錯誤的 I/O 命令或是錯誤的傳輸方向，它會輕易地覆寫掉你的硬碟上的重要資料。

除了最後一個，我們已經討論過上述所有內容。爲了解釋 ring0 中的 DMA，我將展示一個例子，它從一個軟碟中讀取 boot sector。Plug and Play connection 由以下子函式組成：

```
DMAHANDLE dmahandle;          // virtualized DMA channel handle
HIRQ irqhandle;               // virtualized IRQ handle
DWORD baseport;               // I/O base address (i.e., 3F2)

CONFIGRET OnPnpNewDevnode(DEVNODE devnode, DWORD loadtype)
   {                          // OnPnpNewDevnode
   return CM_Register_Device_Driver(devnode, OnConfigure,
      0, CM_REGISTER_DEVICE_DRIVER_REMOVABLE
      | CM_REGISTER_DEVICE_DRIVER_DISABLEABLE);
   }                          // OnPnpNewDevnode

CONFIGRET CM_HANDLER OnConfigure(CONFIGFUNC cf, SUBCONFIGFUNC
   scf, DEVNODE devnode, DWORD refdata, ULONG flags)
   {                          // OnConfigure
   switch (cf)
      {                       // handle configuration function

   case CONFIG_START:
      {                         // CONFIG_START
      CMCONFIG config;
      DWORD channel;
      DWORD irq;
      VID vid = {0, 0, (DWORD) Hw_Int_Proc_Thunk};

      CM_Get_Alloc_Log_Conf(&config, devnode,
         CM_GET_ALLOC_LOG_CONF_ALLOC);
      baseport = config.wIOPortBase[0];      // 3F2h
      channel = config.bDMALst[0];           // 2
      irq = config.bIRQRegisters[0];         // 6
      dmahandle = VDMAD_Virtualize_Channel(channel, NULL);
      vid.VID_IRQ_Number = (USHORT) irq;
      irqhandle = VPICD_Virtualize_IRQ(&vid);
```

```
        return CR_SUCCESS;
        }                   // CONFIG_START

    case CONFIG_STOP:
    case CONFIG_REMOVE:
        if (dmahandle)
            {               // unvirtualize DMA channel
            VDMAD_Unvirtualize_Channel(dmahandle);
            dmahandle = NULL;
            }               // unvirtualize DMA channel
        if (irqhandle)
            {               // unvirtualize IRQ
            VPICD_Force_Default_Behavior(irqhandle);
            irqhandle = NULL;
            }               // unvirtualize IRQ
        return CR_SUCCESS;

    default:
        return CR_DEFAULT;
        }               // handle configuration function
    }                   // OnConfigure
```

這個組態函式（configuration function）結合一個有著硬體組態的 .INF 檔（要求 ports 3F2h~3F5h, DMA channel 2, IRQ 6），將接管與標準軟碟控制器有關的 I/O 資源。

本例之中我們對 IRQ 6 的虛擬化只設計了一個 *Hw_Int_Proc* 函式，內含以下的碼，用以從控制器中取出 status bytes 並將中斷結束掉：

```
void __declspec(naked) Hw_Int_Proc_Thunk()
    {                           // Hw_Int_Proc_Thunk
    _asm
        {                       // call Hw_Int_Proc
        push edx                ; reference data (if any)
        push eax                ; IRQ handle
        call Hw_Int_Proc        ; call C program
        add  esp, 8             ; lose arguments
        cmp  eax, 1             ; turn TRUE return into CLC
        ret                     ; return to VPICD
        }                       // call Hw_Int_Proc
    }                           // Hw_Int_Proc_Thunk

VMM_SEMAPHORE waitsem;          // semaphore to wait on
```

```
BOOL Hw_Int_Proc(HIRQ hirq, DWORD refdata)
   {                            // Hw_Int_Proc
   int i;
   BYTE status[7];              // 7 status bytes

   for (i = 0; i < 7; ++i)
      {                         // read status from controller
      BYTE c;
      FdcWait(baseport);
      _asm
         {                      // read next status byte
         mov edx, baseport      ; DX = port to read from
         add dx, 3              ;   ..
         in  al, dx             ; read status byte
         mov c, al              ;   ..
         }                      // read next status byte
      status[i] = c;
      }                         // read status from controller

   VPICD_Phys_EOI(hirq);
   Schedule_Global_Event(FdcWakeup, waitsem);
   return TRUE;
   }                            // Hw_Int_Proc

void __declspec(naked) FdcWakeup()
   {                            // FdcWakeup
   _asm mov eax, edx            ; EDX (ref data) = semaphore handle
   VMMCall(Signal_Semaphore_No_Switch) // wakeup thread
   _asm ret
   }                            // FdcWakeup

void FdcWait(DWORD baseport)
   {                            // FdcWait
   _asm
      {                         // wait for floppy disk controller
      mov edx, baseport         ; base port address
      add dx, 2                 ; point to status port
   waitloop:
      in  al, dx                ; read status byte
      test al, 80h              ; wait for ready bit to be 1
      jz  waitloop              ;   ..
      }                         // wait for floppy disk controller
   }                            // FdcWait
```

在這個例子中，*FdcWait* 是一個輔助函式，等待軟碟控制器準備好接收或傳送另一個

status byte 或 command byte。對 *Schedule_Global_Event* 的呼叫動作，說明了一個硬體中斷函式如何通知一個等待中的 thread 說某個 I/O 動作已經完成。*FdcWakeup* 是一個 event callback 函式，激發一個 semaphore（其 handle 被傳入做爲參考資料）。由於 *Signal_Semaphore* 並不是一個 asynchronous service，硬體中斷函式不可能直接呼叫它。就像本書稍早的討論一樣，如果一個中斷函式需要執行 nonasynchronous services，它會將一個 event callback 納入排程。這個 event callback 函式可以自由呼叫任何 service call，但受到先前所討論之限制條件的約束（也就是「當 MS-DOS 被使用於 paging I/O 時會引起 page faults」）。

我爲此一簡單的 VxD 建立了一個簡單的 API 函式，它會將軟碟機控制器初始化，並在 channel 2 規劃一個 DMA 傳輸。我將在下一段討論 DMA 傳輸概況。我也寫了一個測試程式，使用此 API 函式。它將一個 512-byte 緩衝區的位址設定給 DS:BX，並呼叫 INT 2Fh, function 1684h（API 函式進入點）。回返時，應用程式的緩衝區會內含 drive A 的磁碟的 boot sector 內容。這個 API 函式看起來像這樣：

```
void ApiEntry(HVM hVM, PTCB tcb, PCRS pRegs)
   {                          // ApiEntry
   static BYTE cmd[] = {3, 0xAF, 2, 0xE6, 0, 0, 0, 1, 2,
      0x12, 0x1B, 255};
   DWORD buffer;
   DWORD code;

   _asm
      {                         // initialize floppy disk controller
      mov edx, baseport         ; DX = base port (3F2)
      mov al, 1Ch               ; turn on drive A motor
      out dx, al                ;   ..
      jmp delay1
   delay1:
      jmp delay2
   delay2:
      add dx, 5                ; DX = 3F7
      xor al, al
      out dx, al
      }                        // initialize floppy disk controller

   // INITIALIZE FOR DMA TRANSFER
```

```
    FdcWrite(baseport, cmd, sizeof(cmd));

    Wait_Semaphore(waitsem, BLOCK_SVC_INTS);

    _asm
        {                   // turn motor off
        mov edx, baseport
        mov al, 0Ch
        out dx, al
        }                   // turn motor off
    }                       // ApiEntry

void FdcWrite(DWORD baseport, BYTE* data, int length)
    {                       // FdcWrite
    while (length-- > 0)
        {                   // for each byte
        BYTE c = *data++;     // next byte to write
        FdcWait(baseport);    // wait until okay to write
        _asm
            {                 // write data byte
            mov edx, baseport  ; compute port address
            add dx, 3          ;   ..
            mov al, c          ; AL = next data byte
            out dx, al         ; output data byte
            }                 // write data byte
        }                     // for each byte
    }                         // FdcWrite
```

ApiEntry 函式對軟碟控制器做動作，讀取 drive A 的 sector 0。軟碟控制器一旦收到最後一個 command byte 便開始動作，所以我們有義務提早將 DMA 控制器準備好。一旦 I/O 動作開始進行，*ApiEntry* 函式便等待一個 semaphore。指定 BLOCK_SVC_INTS 爲 *Wait_Semaphore* 的一個旗標參數，使我們自己的 thread 有可能在必要的時候處理 events。這個 semaphore 會被一個 global event callback 激發，此 event callback 被 IRQ 6 硬體中斷處理常式納入排程。換句話說，當讀取動作完成，*ApiEntry* 函式甦醒過來，於是立刻將馬達停轉並回返至應用程式。

使用 VDMAD Services

雖然 VDMAD 將「由 VM 發動的 DMA 傳輸」虛擬化了，如果你打算在一個 VxD driver 中使用 DMA，VDMAD 還是要求你使用它對外開放的（exported）services。因此，我需要對 VDMAD 做更詳細的描述。一旦你瞭解了對 VDMAD 的讀取動作，你也就知道了「爲 Windows 95 撰寫一個典型的硬體 device driver」所必須知道的一切知識。

表 **13-5** 列出 VDMAD 所提供的 services。

Service	說明
VDMAD_Copy_From_Buffer	將資料從 VDMAD 緩衝區拷貝到相關的 DMA 區域
VDMAD_Copy_To_Buffer	將資料從一個 DMA 區域拷貝到相關的 VDMAD 緩衝區
VDMAD_Default_Handler	使用 VM 所指定的一些數據以及預設的演算法，執行一次傳輸
VDMAD_Disable_Translation	將一個標準的 DMA channel 的自動傳輸能力 "disable" 掉.
VDMAD_Enable_Translation	取消先前的 *VDMAD_Disable_Translation* service 的效果
VDMAD_Get_EISA_Adr_Mode	決定一個 channel 的定址和傳輸模式
VDMAD_Get_Region_Info	將「目前指定給某一個 DMA channel」之區域相關資訊取出
VDMAD_Get_Version	詢問 VDMAD 版本
VDMAD_Get_Virt_State	取得一個 DMA channel 的 virtual state。
VDMAD_Lock_DMA_Region	如果可能，將「被用於一個 DMA 傳輸」的區域的 pages 鎖住
VDMAD_Mask_Channel	"Masks" 一個 DMA channel
VDMAD_Phys_Mask_Channel	"Masks" 一個 DMA channel 而不檢查其結束計量（terminal count）
VDMAD_Phys_Unmask_Channel	"Unmasks" 一個 DMA channel 而不檢查其結束計量（terminal count）
VDMAD_Release_Buffer	釋放一個 VDMAD 緩衝區（先前由 *VDMAD_Request_Buffer* 所指定）
VDMAD_Request_Buffer	保留 VDMAD 緩衝區以便傳輸一個已知區域

Service	說明
VDMAD_Reserve_Buffer_Space	在初始化期間，通知 VDMAD 關於緩衝區空間的需求。
VDMAD_Scatter_Lock	將映射至一個 DMA 區域的所有 pages 鎖住，並提供一個 "scatter translation map"
VDMAD_Scatter_Unlock	解除先前被 *VDMAD_Scatter_Lock* 鎖住的那些 pages。
VDMAD_Set_EISA_Adr_Mode	設定 EISA 擴充定址模式
VDMAD_Set_EISA_Phys_State	爲一個 DMA channel 設定傳輸模式（EISA 才有）
VDMAD_Set_IO_Address	改變「與一個 DMA channel 相關聯」的 port（PS/2 機器才有）
VDMAD_Set_Phys_State	爲一個 DMA channel 設定傳輸模式
VDMAD_Set_PS2_Phys_State	爲一個 DMA channel 設定傳輸模式（PS/2 機器才有）
VDMAD_Set_Region_Info	爲「目前與一個 DMA channel 相關聯」的區域設定資訊
VDMAD_Set_Virt_State	修改一個 DMA channel 的 virtual state。
VDMAD_Unlock_DMA_Region	將「與一個 channel 相關聯」的 DMA 區域解鎖（unlock）
VDMAD_Unlock_DMA_Region_No_Dirty	將 DMA 區域解決鎖定（unlock）。但是不要將 pages 標示爲 "dirty"
VDMAD_UnMask_Channel	"Physically unmasks" 一個 DMA channel
VDMAD_Unvirtualize_Channel	解除一個 DMA channel 的虛擬化。
VDMAD_Virtualize_Channel	取得一個 DMA channel 的擁有權。

表 13-5 VDMAD services

如果你要在你的 driver 中規劃 DMA 傳輸，第一個步驟就是獲得一個被虛擬化之 DMA channel 的 handle：

```
DMAHANDLE dmahandle = VDMAD_Virtualize_Channel(channel, NULL);
```

VDMAD_Virtualize_Channel 的第一個參數是 channel 號碼（0~7）。第二個參數是一個 callback 函式位址，當一個 VM 改變了該 channel 的 virtual state 時，VDMAD 就會呼叫此 callback 函式。以目前的目標而言，我並不打算允許 VMs 規劃我們的 channel，所以我們可以提供一個 NULL 位址。將 channel 虛擬化，有個副作用，就是也會在 DMA

arbitrator（仲裁器）中保留 DMA channel。當然你不應該依賴這個副作用：你應該允許 Configuration Manager 首先指定一個 DMA channel，然後將它虛擬化。

你必須在 driver 起始的時候（最可能的情況是處理 *CONFIG_START* 時）呼叫 *VDMAD_Virtualize_Channel*，並且在 driver 結束時（大部份是在處理 *CONFIG_STOP* 和 *CONFIG_REMOVE* 時）呼叫 *VDMAD_Unvirtualize_Channel*。

VDMAD 的一條終端捷徑

一般而言，在 ring0 中規劃 DMA 傳輸可以成爲一件非常複雜的工作。我將以我們的範例中的事實爲基礎，爲你展示一個快捷作法。然後我會解釋你如何使用低階的 VDMAD services 來完成相同的結果。這個捷徑的大體策略就是使用 *VDMAD_Set_Virt_State* 來設定 DMA channel 的 virtual state，好似以此 VM 中的碼規劃我們即將執行的傳輸。 然後我們呼叫 *VDMAD_Default_Handler* 來處理實體傳輸。我們要使用應用程式緩衝區的虛擬位址，所以我們利用 *Client_Ptr_Flat*，根據 client registers 來決定那個位址：

```
buffer = Client_Ptr_Flat(DS, BX);
```

Client_Ptr_Flat 是 *Map_Flat* 的一個外包巨集。在 assembly 語言中，已經有這樣一個巨集。我定義此巨集完全是爲了這個例子，因爲使用它遠比直接呼叫 *Map_Flat* 方便多了：

```
#define Client_Ptr_Flat(seg, off) Map_Flat(\
    FIELDOFFSET(struct Client_Reg_Struc, Client_##seg), \
    FIELDOFFSET(struct Client_Word_Reg_Struc, Client_##off))
```

> 譯註：## 是 C/C++ 所謂的 merging operator，其用法可參考 《C++ 與虛擬機》（侯俊傑 / 松崗）p.185，或查詢任何 C++ 編譯器之 online 手冊。

我們也必須絕對確定一件事：當我們呼叫 *VDMAD_Set_Virt_State* 時它期望獲得的是一個虛擬位址而非一個實際位址。VDMAD 在每一個 VM 中爲每一個 channel 維護了一個 "translation disable counter"。*VDMAD_Disable_Translaton* 會累加這個 counter，*VDMAD_Enable_Translation* 會遞減這個 counter。當 counter 不爲 0，VDMAD 預期你

「透過 *VDMAD_Set_Virt_State* 送給它的任何緩衝區位址」是個實際位址。爲確保 translation 處於 "enabled" 狀態，我們必須一再呼叫 *VDMAD_Enable_Translation*，直到它回返時的 carry flag 是設立狀態，那表示 translation 已經在這次呼叫之前就被 "enabled" 了。由於我必須寫一個 VDMAD.H 檔（DDK 中並沒有），所以我爲此函式定義了一個外包函式（wrapper）如下：

```
#define ET_WASENABLED   0x0100 // translation already enabled
#define ET_NOTENABLED   0x0001 // not yet enabled

VXDINLINE DWORD VDMAD_Enable_Translation(DMAHANDLE hdma,
    HVM hVM)
    {
    _asm mov eax, hdma
    _asm mov ebx, hVM
    VMMCall(VDMAD_Enable_Translation)
    _asm mov eax, 0
    _asm setz al
    _asm setc ah
    }
```

於是，*ApiEntry* 函式中用以備妥 DMA 傳輸的程式碼，應該像這樣：

```
do  {
    code = VDMAD_Enable_Translation(dmahandle, hVM);
    }
while ((code & ET_WASENABLED) != ET_WASENABLED);
```

最後，這個程式可以發出以下兩個呼叫，用以規劃和初始化 DMA 傳輸：

```
VDMAD_Set_Virt_State(dmahandle, hVM, buffer,
    512, DMA_type_write | DMA_single_mode);
VDMAD_Default_Handler(dmahandle, hVM);
```

請注意我並沒有在呼叫 *VDMAD_Set_Virt_State* 時提供 *DMA_masked* flag。結果，這個呼叫把 virtual channel 給 "unmasks" 了。*VDMAD_Default_Handler* 意味著這個 channel 實際上等於是被 "unmasked" 了，並繼續著手鎖定緩衝區、規劃實際的 DMA 暫存器、"unmask" 實際的 channel。

低階的 VDMAD Programming

很明顯地，*VDMAD_Default_Handler* 運用了精心的機構來設定並執行一個 physical DMA 傳輸。不幸的是，那個函式只在某特定 VM（其 DMA controllers 處於 virtualized 狀態）之中才能運作。如果你的 driver 需要在該 VM 之外動作，你得自己重寫大部份 *VDMAD_Default_Handler* 所處理的許多作爲。幸運的是，VDMAD 原始碼是 DDK 的一部份，所以你我都有機會理解如何做必要的步驟。

你的 driver 需要如下進行，重複 *VDMAD_Default_Handler* 所處理的行爲：

1. 定位出一個可被 DMA controller 接受的實際緩衝區。如果可能的話，你應該以「page-locking 你自己的緩衝區」來完成這件事情。VDMAD 將你的緩衝區視爲「DMA 區域」。如果這個區域位於 controller 可見的位址範圍之外（也就是說在一個 ISA 系統上超過了 16 MB），或如果它佔用了不連續的記憶體，或如果它跨越了一個 DMA page 邊界（對 "byte transfers" 來說是 64 KB，對 "word transfers" 來說是 128 KB），你需要「借用」 VDMAD 的緩衝區。你可能還必須將一次的傳輸打散爲數個可被 controller 接受的小塊。
2. 針對你的傳輸，以適當的位址、計量、模式來規劃 physical DMA controller 暫存器。
3. 將 physical channel "unmask" 起來。如你所知，這可使 DMA 傳輸在 device 一旦開始移交或索求資料並通知 DMA controller 後，立刻開始進行。
4. 安排在傳輸完成後取得控制權，然後立刻 "mask" DMA channel。如果你需要規劃多次傳輸，現在的你應該開始下一次傳輸。如果你「借用」了 VDMAD 緩衝區，你應該交還它。如果你鎖定了任何 pages，你應該將它們解除鎖定（unlock）。
5. 在 "unmasking" channel 之後，你也應該安排一個 timeout，以便處理由於某些 device 問題使得傳輸無法發生的情形。如果 timeout 到期，你可以放棄 DMA 傳輸。最重要的是，如果你借了 VDMAD 緩衝區，你可以將它傳回。

鎖住 DMA 區域

上述 5 個步驟中，第一個最是複雜。你首先得決定 DMA 區域的位址、大小、以及 alignment 的需求，並使用類似下面的碼來鎖住這個區域：

```
DWORD code;              // return code
DWORD address;           // linear address of DMA region
DWORD size;              // byte size of DMA region
DWORD result;            // result of lock attempt
BYTE align;              // 1 for 64 KB, 2 for 128 KB
code = VDMAD_Lock_DMA_Region(address, size, align, &result);
```

如果鎖定成功，上述的 *code* 將爲 0 而 *result* 將是被鎖定區域的實際位址。如果鎖定失敗，*code* 可能是 *DMA_Not_Contiguous* 或 *DMA_Not_Aligned* 或 *DMA_Lock_Failed*，而 *result* 將是「可能已被鎖定於區域一開始處」的 bytes 個數。如果你從 *VDMAD_Lock_DMA_Region* 手中獲得一個失敗代碼，你可以決定是否要把傳輸打散爲數小塊。即使函式傳回的訊息是成功，你也尚未脫離險境，因爲你仍然必須驗證這個實際位址可被 DMA controller 使用。VDMAD 提供了一個未公開的 service，可以決定針對 DMA 的最大 physical page 個數：

```
DWORD maxpage = VDMAD_Get_Max_Phys_Page();
if (code == 0 && result >= (maxpage << 12))
   {                 // beyond maximum
   VDMAD_Unlock_DMA_Region(address, size);
   code = DMA_Invalid_Region;
   }                 // beyond maximum
```

以自己之力來決定最大的 physical DMA 位址是很複雜的，因爲這和你所擁有的硬體 bus 有關（EISA bus 允許任何 32 位元位址，ISA 和 MCA buses 只允許 16MB），也和你所擁有的 CPU 晶片有關（一顆 PC/XT 只允許 1MB，其他的則無限制），也和使用者在各種 SYSTEM.INI 中的設定值有關。但是請注意，*VDMAD_Get_Max_Phys_Page* 是自從 Windows 95 之後才有的，所以你不能夠使用它-- 如果你的 driver 必須與 Windows 3.1 相容的話。

如果你順利完成鎖定（locking）並通過最大位址測試，你可以繼續規劃 DMA 傳輸如下：

```
VDMAD_Set_Region_Info(dmahandle, 0, TRUE, address, size, result);
VDMAD_Set_Phys_State(dmahandle, mode);
VDMAD_UnMask_Channel(dmahandle, Get_Sys_VM_Handle());
```

我的 *VDMAD_Set_Region_Info* wrapper（外包函式）的第二個參數是一個緩衝區識別碼；0 表示我們並沒有使用 VDMAD 緩衝區。第三個參數表示我們是否已經鎖定此次傳輸所牽扯的 pages。address 參數用來指定 DMA 區域的虛擬位址，這對 VDMAD 並沒有真正用途 -- 當我們以低階作法規劃 DMA 傳輸時。size 和 result 參數用來指定真正 DMA 緩衝區的實際大小和位址，它們最終將被規劃到 physical DMA controller 之內。

VDMAD_Set_Phys_State 的 mode 參數內含模式和擴充模式的設定（表 **13-6**），它們基本上和你在規劃一個真正的 DMA controller 時所使用的設定相同，只不過於最低兩個位元中省略了 channel 號碼（譯註：請參考圖 **13-4**）。*VDMAD_UnMask_Channel* 的第二個參數是一個 VM handle。你必須提供一個合法的 VM handle -- 即使傳輸並非針對某個特定的 VM。這個 VM handle 在大部份情況下最終會被忽略，但若你指定 NULL（你可能會想以此取代本例中的 System VM handle），卻有一個特別的意義。

Mode Flag	說明
DMA_type_verify	驗證這個行動（operation）
DMA_type_write	寫到記憶體去
DMA_type_read	從記憶體中讀出
DMA_AutoInit	指定 autoinitialization mode
DMA_AdrDec	指定 address decrement mode
DMA_demand_mode	指定 demand mode
DMA_single_mode	指定 single transfer mode
DMA_block_mode	指定 block transfer mode
DMA_cascade	串接（cascades）自 master DMA controller
Programmed_IO	一種擴充模式

Mode Flag	說明
PS2_AutoInit	一種擴充模式
Transfer_Data	一種擴充模式
Write_Mem	一種擴充模式
_16_bit_xfer	一種擴充模式

表 13-6 VDMAD_Set_Phys_State 的模式旗標（Mode flags）

借用 VDMAD 的緩衝區

如果你沒有安然通過鎖定測試和最大位址測試，你必須借用一個緩衝區，才能進行傳輸。VDMAD 會在它處理 *Init_Complete* 訊息時配置一個有正確 alignment 與 location 的 page-locked 緩衝區。你可以在任何時刻（只要是 VDMAD 處理 *Init_Complete* 之前）呼叫 *VDMAD_Reserve_Buffer_Space* 對此緩衝區設定一個最小尺寸。這個緩衝區的目的是當 DMA 區域（DMA region）爲了某些因素而無法使用時，提供一塊記憶體空間以執行 DMA 傳輸。DMA 區域（DMA region）如果是非連續的，或是它跨越了一個 DMA page 邊界，或是高於最大位址，或是爲了某些原因不能夠鎖定，都將無法使用。

爲了借用 DMA 緩衝區，你必須呼叫 *VDMAD_Request_Buffer*：

```
BYTE id;                // buffer ID for VDMAD_Set_Region_Info
code = VDMAD_Request_Buffer(address, size, &result, &id);
```

如果成功，*VDMAD_Request_Buffer* 傳回 0 並將 *result* 設爲被指定之緩衝區的實際位址，將 *id* 設定爲緩衝區的識別碼（ID）。稍後你應該將此緩衝區識別碼和實際位址交給 *VDMAD_Set_Region_Info*。如果 *VDMAD_Request_Buffer* 失敗，回返值應該不是 *DMA_Buffer_Too_Small* 就是 *DMA_Buffer_In_Use*。面對前者，你必須將傳輸打散爲小塊狀。面對後者，你必須等待緩衝區恢復自由身。VDMAD 的預設處理常式簡單地呼叫 *Time_Slice_Sleep* 等待 100 毫秒（ms），然後就再次嘗試。

一旦你獲得了一部份的「VDMAD 的 DMA 緩衝區」的主權，應該接下去將傳輸設定妥

當，大約就像簡單地鎖定自己的緩衝區一樣。唯一不同的是，你提供一個緩衝區識別碼給 *VDMAD_Set_Region_Info*，如果必要，你必須將資料拷貝到該緩衝區，像這樣：

```
VDMAD_Set_Region_Info(dmahandle, id, TRUE, address, size, result);
if (mode & DMA_type_read)
   VDMAD_Copy_To_Buffer(id, address, 0, size);
VDMAD_Set_Phys_State(dmahandle, mode);
VDMAD_UnMask_Channel(dmahandle, Get_Sys_VM_Handle());
```

VDMAD_Copy_To_Buffer 會把資料從位於線性位址 address 處的 DMA 區域中的 size 個 bytes，拷貝到 id 所指定之緩衝區中的某個位置（第三個參數將指定偏移位置）。如果你正在做一個 *DMA_type_read* 動作，也就是說如果你正在讀取記憶體內容並寫到 device 中，你應該在 "unmasking" channel 之前，將資料拷貝到緩衝區。

傳輸完畢後的處理

當 DMA 傳輸完成，你的 device 會產生一個硬體中斷。你應該將剛才所使用的 DMA channel "mask" 起來，阻止庶出的傳輸。如果你從 VDMAD 手中借了一塊緩衝區，你應該將資料從緩衝區中拷貝到 DMA 區域，此外，你應該釋放你所使用的緩衝區，像這樣：

```
VDMAD_Mask_Channel(dmahandle);
if (id)
   {                 // finish buffered transfer
   if (mode & DMA_type_write)
      VDMAD_Copy_From_Buffer(id, address, 0, size);
   VDMAD_Release_Buffer(id);
   }                 // finish buffered transfer
else
   if (mode & DMA_type_write)
      VDMAD_Unlock_DMA_Region(address, size);
   else
      VDMAD_Unlock_DMA_Region_No_Dirty(address, size);
```

這段善後工作揭露出我們必須在 "I/O programming with VxDs" 這個主題中考慮的最後一件煩人的事務。如果這次傳輸是一個 *DMA_write_type* 動作（從 device 到記憶體），我們就是改變了某些 pages 的內容。通常，當你將資料寫入記憶體，CPU 中的 paging 邏

輯會自動將某個 page 標記爲 "dirty" -- 在 PTE（page table entry）中標記。這個 "dirty" flag 告訴 paging 系統說，如果它要將相同的 "page frame"（譯註：意指實際記憶體中的一個 page）給其他 page（譯註：意指虛擬定址空間中的 page）使用，就請先將記憶體中的 page 內容寫到置換檔（swap file）中。但是 paging 系統絕不會知道我們對 DMA 區域所做的內容更改，因爲 DMA controller 繞過了 CPU 中所有的 page 傳輸邏輯。因此，*VDMAD_Unlock_DMA_Region* 爲每一個與此次傳輸有關的 page 設立 "dirty" flag。然而，如果我們做了一次 *DMA_type_read* 動作（從記憶體到 device），我們並不需要將涉及的 pages 也標示爲 "dirty"。因此我們呼叫的是另一個 service（*VDMAD_Unlock_DMA_Region_No_Dirty*），它不會將 "dirty" flags 設立起來。

選擇性 複雜

我要以一些忠告（簡化原則）結束這漫長的一章。很明顯地，如果你要做 DMA 傳輸（到任意緩衝區中），你有很複雜的瑣事要進行。如果你以適當的大小保留自己的緩衝區，並在自己的 device 初始化過程中爲它定位，那麼就可以避免鎖定一塊區域、向 VDMAD 借一個緩衝區 ... 等等等。你可以使用 PAGEUSEALIGN, PAGEFIXED, 和 PAGECONTIG 等選項來指定 *_PageAllocate* 如何配置緩衝區；你可以指定 64-KB 或 128-KB alignment，視你要進行的是 8-bit 或 16-bit 傳輸而定。此外，你可以指定最大位址爲 16 MB（page 號碼是 01000h）以確定在 ISA, MCA, 和 EISA 系統中的相容性。這個作法的唯一缺點（可能是個大號缺點）就是，只要你的 driver 被載入，就得有一塊實際記憶體做爲你的 DMA 緩衝區之用。當然啦，對於一個活動頻繁的 device 而言，將資源永久貢獻給 device 是完全適當的。提醒你，在 Windows 95 中你可以在任何時候使用 PAGEUSEALIGN 和相關選項（Windows 3.1 將這些選項侷限於 device 初始化時使用），但是你得承擔「*_PageAllocate* 因不適當的使用時機而失敗」的風險。

第 14 章

通訊驅動程式
（Communications Drivers）

電腦的 serial port 是使用者對外的一個窗口。經由它，以各種令人迷惑的格式，流動著各種不斷累加的資訊。配備有一個 serial port（序列埠）和一個 modem（數據機）之後，使用者就可以傳送並接收電子郵件和傳真，管理其投資，並考慮以下行爲：收看外太空影像，求愛，金融侵佔，甚至顛覆政府。簡直可以說，通訊硬體使得個人電腦成爲我們在商務生活中的另一個工具，用在好的方面或壞的方面，全憑個人意向。無遠弗屆的網際網路（Internet）和全球資訊網（World Wide Web），以及高速數位通訊技術如整體服務數位網路（Integrated Services Digital Network，ISDN）的部署，使得個人電腦使用者的對外通訊比過去任何時刻都顯得更爲重要。

Parallel port 提供個人電腦使用者另一種窗口。最早其目的只是爲了成爲一個輸出專用的印表機連接器，但現在 parallel port 已經成長，爲更多的需求提供服務。靈巧的程式設計，使我們得以利用 Traveling Software's LapLink 將兩部 PCs 連接起來，以 parallel port 進行檔案傳輸。External SCSI adapters 可以連接 parallel port。Xircom 開發出一個網路介面，可以插在 parallel port 上面。對於膝上型電腦的使用者而言，這無異是一份天賜之

物，因爲他們面對的是一個已經被過度縮小，難以再膨脹的硬體空間。數家公司販賣所謂的 "dongles"，可以插在 parallel port 上頭，針對昂貴的軟體，形成一個安全防護鍵（security keys）。如今印表機也變得更有智慧一些，可以傳遞訊息給作業系統，說明它們的需求和能力。

當然，硬體需要軟體的協助，才能順利運作。在 Web 瀏覽器支配下的，是 drivers 和 APIs，那是系統程式員的國度。一個名爲 VCOMM 的 static VxD，是 Windows 95 之中涉及 serial ports 和 parallel ports I/O 動作的焦點。VCOMM 和 port driver 以及 port virtualization VxDs 一起運作，以便協調 physical port 的使用，並提供應用程式一層與 device 無關的介面。

架構概觀

就像 Windows 的其他領域一樣，要瞭解我們身置何處，最好是先瞭解我們曾經身在何處。所以我要先討論 Windows 早期版本中的標準 serial 晶片的動作及其 serial communications drivers 的架構。然後才討論 Windows 95 的 VCOMM 架構及附屬話題。

透視 8250

早期的 PCs 使用者，有時候會加上一些可有可無的介面卡，其中內含 serial 和 parallel ports。早期的 serial ports 是以 Intel 8250 universal asynchronous receiver-transmitter（UART）晶片爲基礎。標準的 system BIOS 提供頗爲麻煩的 INT 14h 以處理 serial port。MS-DOS 透過 AUX device 或經由 COM1, COM2 等 devices 來支援 serial 通訊。但不論 BIOS 或 MS-DOS 提供的介面，速度都太慢了。而且那些介面依賴「連續不斷地 "polling" input data」，使它們不適用於全雙工（full duplex）通訊，也不適用於多工系統。

爲了克服 BIOS 和 MS-DOS serial APIs 的限制，程式員開始學習如何直接對 8250 晶片做程式規劃。這麼做的主要好處是，當一個新的輸入字元到達，或是當晶片已經送出一

個輸出字元，你可以讓晶片產生一個硬體中斷，而非不斷地對 port 做 "polling" 動作。另一個好處是，你可以對 baud-rate generator 做程式規劃，使它到達比 9600（BIOS 的極限）更高的 baud rate。

Windows 內含一個 DLL driver（COMM.DRV）用來做爲 USER.EXE 和 8250 晶片之間的介面。應用程式使用低階的 API 如 *OpenComm*, *ReadComm* 等等來和 USER.EXE 通訊。USER.EXE 呼叫 COMM.DRV 的進入點，指定名稱如 *inicom* 和 *reccom*。Windows 3.x 的 COMM.DRV 直接將資料寫往 8250 ports 並持續擲出硬體中斷。Enhanced-mode Windows 3.0 內含一個 VxD，名爲 VCD （Virtual Communications Device），用以處理不同 VMs 之間對於 ports 的搶奪。實驗證明，在 Windows 3.0 之中要提供比 4800 baud 更快的通訊速度，極爲困難。所以 Windows 3.1 加了一個名爲 COMBUFF 的 VxD。COMBUFF 的任務是將 UART 的所有輸入和輸出做緩衝（buffering）處理。COMBUFF 也將 serial ports 和 IRQs 虛擬化，以便能夠提供「對 VM 程式（包括 COMM.DRV）之輸入」的預先緩衝處理，以及「對 UART 輸出」的緩衝處理。**圖 14-1** 說明 Windows 3.1 中的 serial 通訊架構。

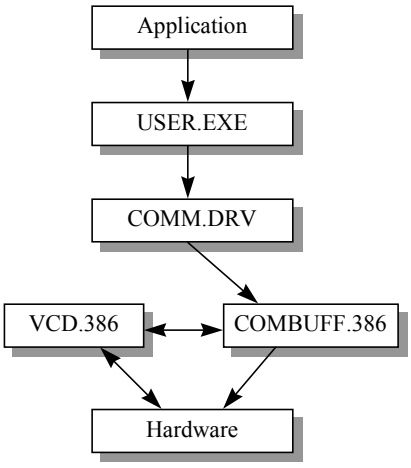


圖 14-1　Windows 3.1 中的 serial 通訊架構

有了圖中所示的 COMBUFF，較高速率的通訊就有可能達到了。但是 Windows 應用程式仍然受制於效率的懲罰，因爲通過 USER.EXE 和 COMM.DRV 的路徑中有過多的裝飾。廠商們例行性地供應他們自己版本的 COMM.DRV, VCD.386, COMBUFF.386。情況混沌不明，因爲各獨立廠商對於所有問題的最佳解決方案，當然沒有共識。

8250 晶片的程式規劃（Programming the 8250 Chip）

由於有如此多的 serial 通訊軟體架構支援一顆特定晶片（8250 UART），瞭解此晶片的基礎操作顯然是件重要的事情。圖 **14-2** 是一張簡化的 8250 區塊圖（實際上此圖顯示的是其較高效率的後代產品，也就是 16550），用以強調程式人員可以存取的特質。此晶片需要 8 個 8-bit I/O ports。在一個標準組態（configuration）中，COM1 使用 ports 3F8h~3FFh，COM2 使用 ports 2F8h~2FFh。此晶片也需要一個 IRQ，通常 COM1 使用 IRQ 4，COM2 使用 IRQ 3。

爲了讓這顆晶片能夠運作，你必須寫組態資訊（configuration information）到 Line Control、FIFO Control、以及 Interrupt Enable 等暫存器中。如果要改變 baud rate，你必須輸出一個 16-bit clock divisor 到 divisor register pair 上頭。通常 divisor register pair 是不可見的，如果你設立 Line Control register 的 bit 7，divisor register pair 就能夠從晶片的 base address 以及 base address + 1 中看到。baud rate 的決定法是，將 115,200（最大的 data rate）除以 divisor 的值。也就是說，如果要將晶片設定爲 9600 baud，你應該將 divisor register pair 設爲 12。

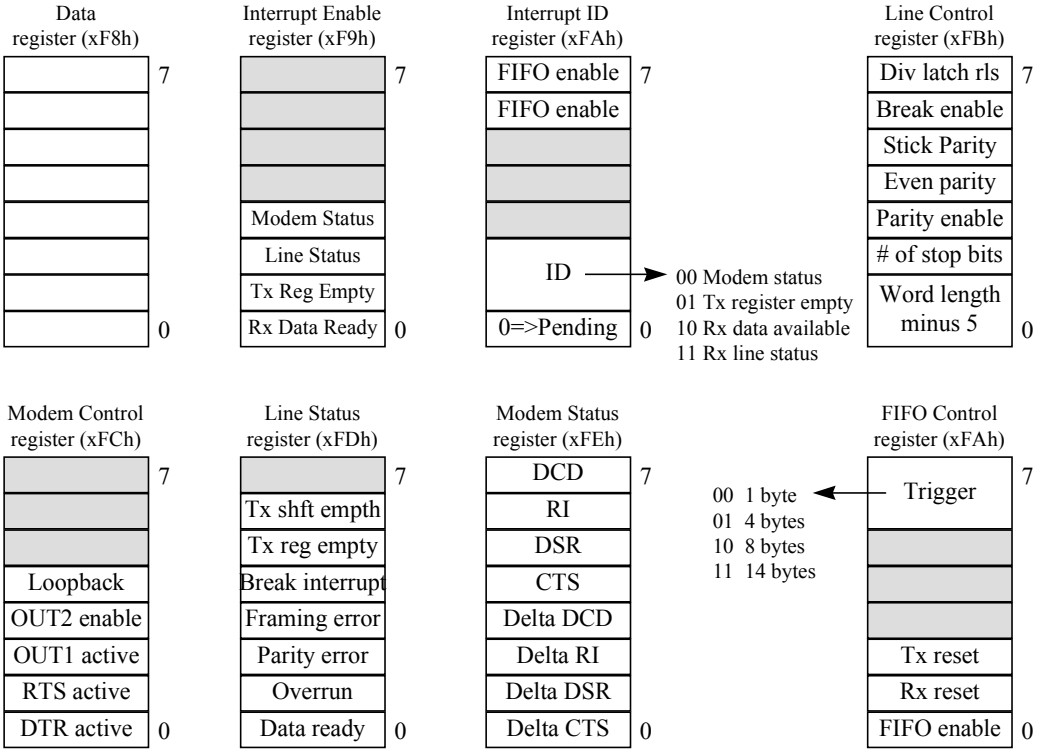


圖 14-2　16550 UART 的 programming 架構

UART 會在某些 events 發生時產生中斷。藉由設定 Interrupt Enable register 中的位元，你可以表示要處理哪些 events。當 Modem Status 或 Line Status 暫存器中有任何改變，便會引發 Modem Status 和 Line Status 中斷。一個 Modem Status 中斷表示連接於 UART 的 RS-232 控制訊號線有了變化，而一個 Line Status 中斷一般而言是表示一個錯誤。當此晶片完整送出一個輸出字元，便會發生 Transmit Register Empty 中斷。這一中斷表示，晶片有能力再傳送另一個字元了。當晶片接收到一個輸入字元，便會發生 Receive Data Ready 中斷，軟體必須在下一個 bytes 到達之前先讀取這個輸入字元，以便清除中斷並避免發生 "overrun" 錯誤。

Modem Control 和 Modem Status 暫存器用來管理 RS-232 通訊協定，UART 靠它們與

PC 以外的世界溝通。常常，UART 會被連接到一台數據機（modem），這也就是爲什麼這些暫存器的名稱中有著 Modem 這個字的原因。這些暫存器中的 6 個 bits 分別是 DTR (Data Terminal Ready), RTS (Request To Send), DCD (Data Carrier Detect), RI (Ring Indicate), DSR (Data Set Ready), 和 CTS (Clear To Send)，對應於 RS-232 規格所指定的 6 條標準控制線。

由於各種 modems 和通訊協定之間有如此廣泛的差異，大部份通訊程式把低階觀念（像是 line control settings 以及 RS-232 signals 等等）帶到使用者介面上來。終端用戶和應用軟體需要控制 UART 的運作細節，所以通訊用的 device drivers 就必須提供一個讓資訊在應用軟體和硬體之間的傳輸管道。舉個例子，終端用戶能夠選擇使用所謂的 "hardware flow control"。Flow control 代表一種機制，視「接收器是否已經準備好接收資料」來管制資料的傳輸。Hardware flow control 必須依賴 RTS 和 CTS 控制線。爲了讓一個合作型多工應用程式能夠以此方法有效執行 flow control，程式必須幾乎即時地處理 Modem Status 暫存器的現況（CTS signal 出現在其中）。因此，Windows 通訊驅動程式總是提供一種方法，讓應用程式取得一個位址指標，而這個位址正是硬體中斷處理常式（hardware interrupt handler）在一次中斷之後儲存 Modem Status 暫存器的地方。

VCOMM 架構

Virtual Communications Driver（VCOMM）是 Windows 95 架構之中處理 serial 通訊和 parallel 通訊的中心。圖 **14-3** 顯示 VCOMM 和 Windows 95 的其他系統元件之間的交互關係。

爲了執行 serial 通訊，16 位元 Windows 應用程式通常使用 16 位元 USER.EXE 所實作的標準 Windows 3.x API 函式。USER.EXE 把這些 API calls 轉而呼叫 COMM.DRV driver。雖然早期版本的 Windows COMM.DRV 擲出 serial port 中斷並發出 port I/O 動作，但 Windows 95 版的 COMM.DRV 卻只是簡單地經由一個標準的 INT 2Fh/function 1684h 呼叫 VCOMM（請看表 **14-1**）。

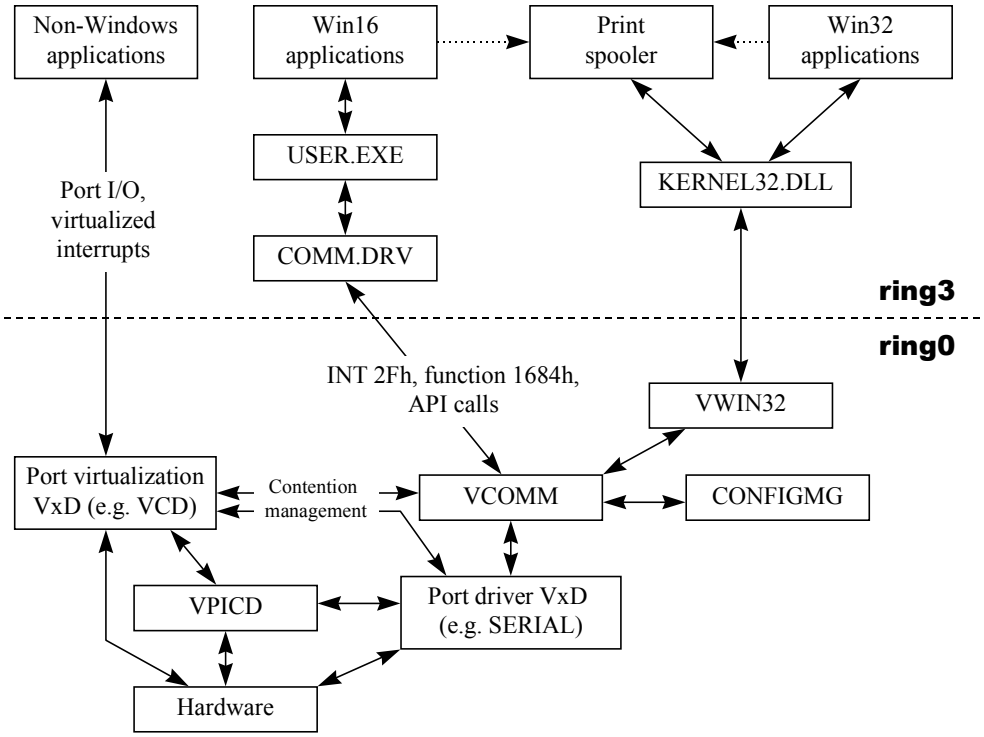


圖 14-3　Serial communications with VCOMM

函式號碼	助憶碼（Mnemonic）	說明
0	VCOMM_PM_API_OpenCom	打開一個 communications device
1	VCOMM_PM_API_SetCom	以一個 DCB 結構設定 port 狀態 (類似 Win16 *SetCommState* 函式）
2	VCOMM_PM_API_Setup	設定輸入用和輸出用的 queue 的大小
3	VCOMM_PM_API_ctx	送出一個高優先權的字元
4	VCOMM_PM_API_TrmCom	關閉一個 port
5	VCOMM_PM_API_StaCom	決定（偵測）port 的狀態
6	VCOMM_PM_API_cextfcn	執行一個經過擴充的控制函式（類似 Win16 *EscapeCommFunction* 函式）
7	VCOMM_PM_API_cFlush	清掃（flush）輸入用或輸出用的 queue

函式號碼	助憶碼（Mnemonic）	說明
8	VCOMM_PM_API_cevt	註冊一個 event word 和一個 event mask（類似 Win16 *SetCommEventMask* 函式）
9	VCOMM_PM_API_cevtget	取得目前的 event word 並清除被選中的 events（類似 Win16 *GetCommEventMask* 函式）
10	VCOMM_PM_API_SetMSRShadow	註冊一個 byte 以便在 Modem Status 暫存器改變時即能夠收到一份最新副本。
11	VCOMM_PM_API_WriteString	送出輸出資料到 port
12	VCOMM_PM_API_ReadString	從 port 中讀取輸入資料
13	VCOMM_PM_API_EnableNotify	註冊一個 notification 函式並接收和傳送 queue size notification thresholds

表 14-1 Windows 應用程式所使用的 VCOMM API 函式

32 位元 Windows 程式使用正規的 *CreateFile* API 函式，爲一個通訊埠開啓一個檔案 handle，並使用其他的 Win32 API 來執行 serial I/O 動作（請看**表 14-2**）。在其內部，KERNEL32.DLL 使用一個未公開的介面（藉由 VWIN32 VxD）以取得 VCOMM 的 services。VCOMM 的 VxD services 顯示於**表 14-3**。Print spooler 是一個 Win32 程式，使用相同的未公開 VWin32 介面，在 GDI 和 VCOMM 之間做聯繫。

VCOMM 藉由呼叫 port driver VxD，爲「來自 Windows 程式和 VxD 呼叫者」的 serial I/O requests 服務。**表 14-4** 列出 port driver 開放出來給 VCOMM 使用的每一個函式。

在 MS-DOS 虛擬機器中執行的程式，會 "hooks" 標準的 serial port 硬體中斷，並發出讀（或寫）動作到標準的 I/O ports 上頭。一個 port virtualization VxD 會將 interrupts 和 I/O ports 虛擬化。標準的 port virtualization VxD 名爲 VCD，和早期的 Windows 版本一樣。除了針對虛擬機器中的程式，將 serial ports 虛擬化之外，一個 port virtualization VxD 也爲 VCOMM 提供競爭管理（contention management）的機能。

Win32 API 函式	說明
BuildCommDCB	對通訊埠（communications port）建立一個控制結構
BuildCommDCBAndTimeouts	建立一個 DCB 和一個 COMMTIMEOUTS 結構
ClearCommBreak	清除一個 break condition
ClearCommError	獲得並清除（resets）通訊錯誤
CloseHandle	關閉一個通訊埠（communications port）
CommConfigDialog	從使用者手上獲得組態參數（configuration parameters）
CreateFile	打開一個通訊埠（communications port）
EscapeCommFunction	執行一個擴充的控制函式（extended control function）
GetCommConfig	取得通訊組態（communication configuration）
GetCommMask	取得 event mask
GetCommModemStatus	取得目前的 modem status
GetCommProperties	獲得 communications device properties
GetCommState	獲得目前的 device state
GetCommTimeouts	取得 timeout 的設定情況
GetDefaultCommConfig	取得預設組態（default configuration）
PurgeComm	清理 input and/or output queues
ReadFile	從檔案中讀取資料
SetCommBreak	送出一個 break signal
SetCommConfig	設定 port configuration
SetCommMask	設定 event mask
SetCommState	設定 port state
SetCommTimeouts	設定 port timeout 值
SetDefaultCommConfig	設定預設組態（default configuration）
SetupComm	建立 input queue 和 output queue 的大小
TransmitCommChar	送出一個優先權字元（priority character）
WaitCommEvent	等待一個 overlapped (asynchronous) port operation 的完成
WriteFile	寫資料到一個檔案中

表 14-2　用於 serial 通訊的 Win32 API 函式

Service	說明
_VCOMM_Acquire_Port	要求絕對控制這個 port
_VCOMM_Add_Port	新增加一個 port
_VCOMM_ClearCommError	獲得並清理通訊錯誤
_VCOMM_CloseComm	關閉一個 port
_VCOMM_EnableCommNotification	註冊一個 event notification callback
_VCOMM_EscapeCommFunction	執行一個擴充的控制函式（extended control function）
_VCOMM_GetCommEventMask	取得目前的 events 並清除被選中的 events
_VCOMM_GetCommProperties	取得一個 port 的 properties
_VCOMM_GetCommQueueStatus	取得一個 port 的 status 和 queue sizes
_VCOMM_GetCommState	取得一個 port 的目前狀態
_VCOMM_GetLastError	取得最近發生的錯誤代碼
_VCOMM_GetModemStatus	取得目前的 modem 狀態
_VCOMM_GetSetCommTimeouts	取得或設定 port 的 timeout 值
_VCOMM_OpenComm	打開一個 port
_VCOMM_PurgeComm	清理 input queue 或 output queue
_VCOMM_ReadComm	讀取 input data
_VCOMM_Register_Enumerator	爲 Plug and Play devices 註冊一個 enumerator
_VCOMM_Register_Port_Driver	註冊一個 port driver
_VCOMM_Release_Port	釋放對一個 port 的控制權
_VCOMM_SetCommEventMask	註冊一個 event word 並設定 event mask
_VCOMM_SetCommState	設定 port state
_VCOMM_SetReadCallback	註冊一個 callback 並設定 input queue size threshold
_VCOMM_SetupComm	設定 queue 的大小和位址
_VCOMM_SetWriteCallback	註冊一個 callback 並設定 output queue size threshold
_VCOMM_TransmitCommChar	送出一個優先權輸出字元（priority output character）
_VCOMM_WriteComm	送出 output data

表 14-3 爲 VxD 而準備的 VCOMM API services

Port Driver 函式	說明
ClearError	取得並清除錯誤
Close	關閉一個 port
EnableNotification	註冊 event notification procedure
EscapeFunction	執行一個擴充的控制函式（extended control function）
GetCommConfig	取得 port configuration
GetCommState	取得 port state
GetError	取得最近的錯誤代碼
GetEventMask	取得目前的 events 並清除被選中的 events
GetModemStatus	取得目前的 modem status
GetProperties	取得 port properties
GetQueueStatus	取得 port status 和 queue counts
Open	打開一個 port
Purge	清理 input queue 或 output queue
Read	讀取 input data
SetCommConfig	設定 port configuration
SetCommState	設定 port state
SetEventMask	註冊一個 event word 並設定 event mask
SetModemStatusShadow	註冊一個 byte 以便在 Modem Status 暫存器改變時即能夠收到一份最新副本。
SetReadCallback	註冊 callback 並設定 input queue threshold
Setup	設定 queue 的大小和位址
SetWriteCallback	註冊 callback 並設定 output queue threshold
TransmitChar	送出一個優先權字元（priority character）
Write	送出 output data

表 14-4　Port driver 函式

Serial Port Drivers

這一節中我將討論 serial port driver 的細節。再一次參考**圖 14-3**，請注意，VCOMM 使用 port driver VxDs 做爲 physical ports 的 device drivers。VCOMM 本身扮演兩個角色，它不但是 port drivers 和 Windows 應用程式之間的媒介物，也是 Windows 95 Configuration Manager 和 port drivers 之間的聯繫點。

Microsoft 提供一個名爲 SERIAL.VXD 的 port driver，做爲 Windows 95 的一個標準元件。SERIAL.VXD 的原始碼（以組合語言完成）可從 Windows 95 DDK 中獲得。要探索 port drivers 如何運作，你通常必須解讀組合語言版的 SERIAL.VXD。然而在本書之中，我將顯示一個 C++ 版的 SERIAL.VXD，用以說明 port drivers 的內部工作。

關於 SERIAL.VXD 範例程式　和本書的其他範例程式不同的是，SERIAL.VXD 是一個完整的，可有效運作的 serial port driver。我選擇以 C++ 來撰寫它，因爲 port drivers 的架構似乎適合使用 class 的繼承和虛擬函式。我也決定使用標準的 DDK 表頭檔，而不使用如 VToolsD 之類的工具，因爲我希望避免在一個複雜的 class library 身上傷腦筋。本章出現的程式碼可從書附光碟的 \CHAP14\PORTDRIVER 目錄中獲得。

雖然本節的討論很明顯是和 serial port 的 port driver 有關，但其中許多也適用於 parallel port drivers。兩種 drivers 之間有類似的性質，是因爲這兩種 ports 都是以緩慢的速度搬移資料（進出電腦）。印表機技術涵蓋 serial 和 parallel 兩種，所以如果爲 ring3 程式提供廣泛不同的 drivers interfaces，將是一件愚蠢的事情。

載入 Port Drivers

將 port drivers 載入記憶體之中，並使它有能力控制一個實際的 port，整個程序說起來是有點複雜。讓我們以一部有著兩個標準的 serial ports （COM1 和 COM2）的電腦爲例。Windows 95 Setup 程式會偵測到這兩個 ports，視之爲傳統的（legacy）devices 並且在

registry 的 HKLM\Enum\Root 分支中加上 subkeys（圖 **14-4**）。在這些 hardware subkeys 中，named value "Driver" 的值（Data）係指向 HKLM\System\CurrentControlSet\Services\Class 的 software subkeys（圖 **14-5**）。每一個 software key 的 named values 都任命 VCOMM 爲 device loader，並任命 SERIAL.VXD 爲 port driver。

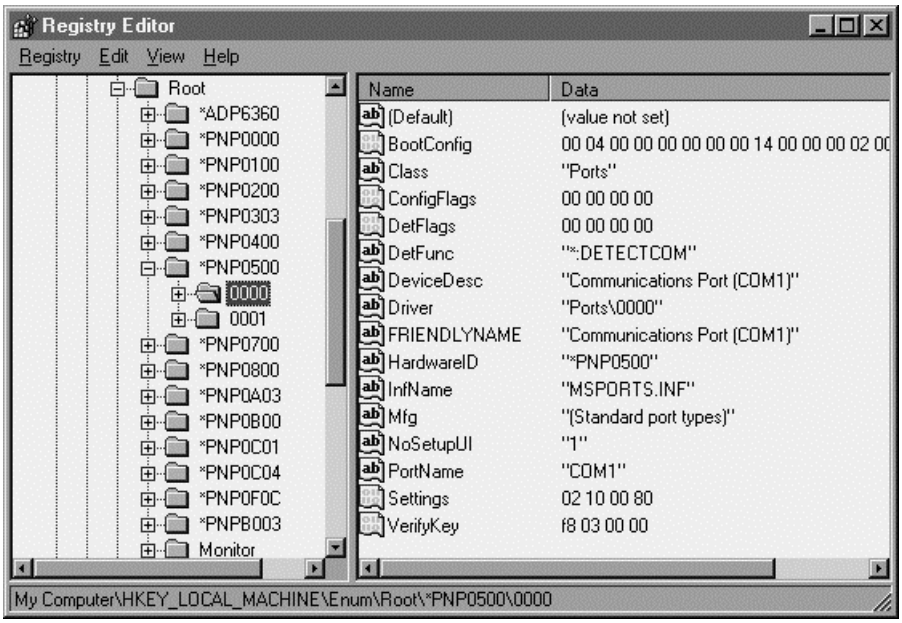


圖 14-4　COM1 的 hardware key

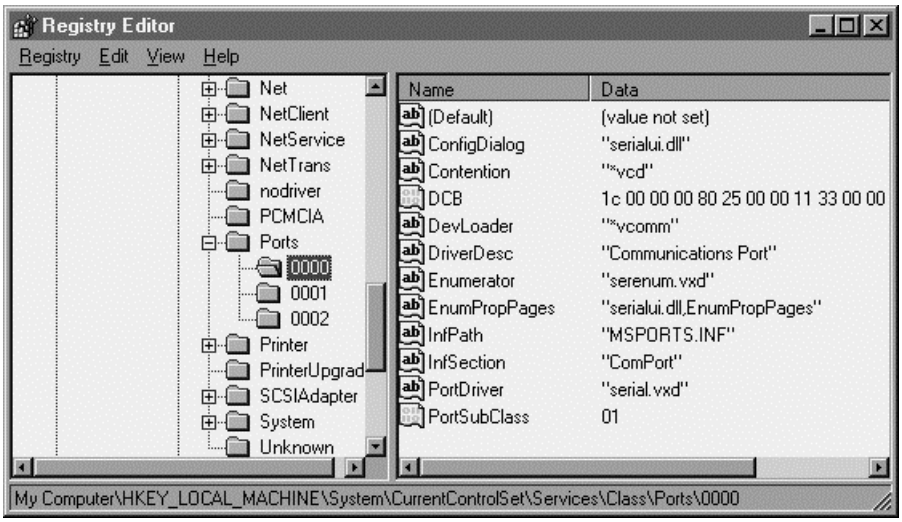


圖 14-5 COM1 的 software key

就像前一章 Plug and Play 中所討論的，root enumerator 會爲每一個 serial port 產生一個 DEVNODE，在那上面，Configuration Manager 會送給 VCOMM 一個系統控制訊息 *PNP_New_Devnode*，並含有 *DLVXD_LOAD_DEVLOADER* 參數。這個參數告訴 VCOMM 爲這個 DEVNODE 扮演 device loader 的角色。VCOMM 於是不將自己註冊爲 DEVNODE 的 device driver，並提供一個 configuration function 位址。當 Configuration Manager 組態（configures）一個 serial port 時，它就呼叫 VCOMM 的 configuration function，在其中，VCOMM 決定哪一個 I/O ports 以及哪一個 IRQ 屬於這個 serial port。一般而言，COM1 擁有 I/O ports 3F8h~3FFh 以及 IRQ 4，COM2 擁有 I/O ports 2F8h~2FFh 以及 IRQ 3。VCOMM 也從每一個 DEVNODE 的 software key 中讀取 named value "PortDriver" 的值，並因此決定 SERIAL.VXD 就是兩個標準的 serial ports 的 port driver。

終於，VCOMM 的一些 client 程式企圖打開一個 port。如我們所見，"port open request" 最終會導至 *_VCOMM_OpenComm*（這是一個 VxD service call）。Client 程式可以是一個 16 位元 Windows 程式，那麼就會使用 Win16 API *OpenComm* 函式。*OpenComm* 是

USER.EXE 的一個函式進入點，會呼叫 COMM.DRV（一個 16-bit DLL）的 *inicom* 函式進入點，於是呼叫到 VCOMM 的保護模式 API 進入點，執行 *VCOMM_PM_API_OpenCom* 函式。更內部而言，這個 API 進入點發出一個 VxD 層面的 service call，喚起 *_VCOMM_OpenComm*。

VCOMM client 程式也可能是一個 Win32 程式。欲打開 port，必須呼叫正規的 Win32 API *CreateFile* 函式，並指定 COM1 或 COM2 爲開檔名稱。KERNEL32.DLL 所實作的 *CreateFile* 函式，使用一個未公開介面來呼叫 VWIN32 VxD。VWIN32 內部以另一個未公開的進入點，呼叫 VCOMM，最終導至呼叫相同的 *_VCOMM_OpenComm* service call 以打開 port。

最後一種情況是， VCOMM client 程式可能是另一個 VxD，它可以直接發出一個 *_VCOMM_OpenComm* call。我所知道的唯一一個標準 VxD 並扮演 VCOMM client 程式者，是爲 AT-command-compatible modems 而寫的 UNIMODEM driver。你可能需要撰寫自己的 VCOMM client VxD 以支援應用程式。

不論是上述哪一種情況，總之， *_VCOMM_OpenComm* call 包括一個字串參數，指示要打開哪一個 port。這時候 VCOMM 會諮詢其內部資料結構，確定 SERIAL.VXD 就是這個 port 的 port driver。VCOMM 於是動態載入 SERIAL.VXD。在動態載入過程之中，VMM 會送給 SERIAL.VXD 一個系統控制訊息 *Sys_Dynamic_Device_Init*。SERIAL.VXD 於是呼叫 VCOMM 將自己註冊爲一個 port driver：

```
SYSCTL BOOL OnSysDynamicDeviceInit()
   {                        // OnSysDynamicDeviceInit
   return VCOMM_Register_Port_Driver((PFN) DriverControl);
   }                        // OnSysDynamicDeviceInit
```

這一小段碼之中，有三件事需要解釋。在我的範例程式裡，符號 SYSCTL 被簡單地以 #defined 定義爲 *extern "C"*，使你比較容易從組合語言的 device control procedure 中參考到系統控制訊息處理常式（system control message handlers）。標準的 VXDWRAPS.H 表頭以 PFN 做爲一般的函式指標的型別，但是 PFN 難以吻合它所希望描述的函式的真正

參數。因此，通常需要把函式參數轉型爲一個 VxD service calls，像上述程式碼所示。最後一點，VXDWRAPS.H 爲 VCOMM 定義了所謂的外包函式（wrappers），沒有前導的底線字元，和傳統的習慣不同。例如，VCOMM 開放（exports）一個與 C 習慣相符的 service，名爲 *_VCOMM_RegisterPort_Driver*，而 VXDWRAPS.H 針對它所宣告的 wrapper 函式並沒有前導的底線字元。

VCOMM_Register_Port_Driver 的目的是要向 VCOMM 註冊一個所謂的 driver control function。此函式的原型（prototype）如下：

```
BOOL DriverControl(DWORD cf, DEVNODE devnode, DWORD refdata, ...)
```

其中參數 *cf* 表示一個 control function code；參數 *devnode* 是一個指標，指向一個 device node；參數 *refdata* 是一塊所謂的 "reference data"，要被交給底層的 VCOMM 函式。剩下的可變參數（variable arguments）則視 *cf* 而定。

儘管我剛剛以誇張的語句描述了 driver control function，但目前只有一個 control function 被定義出來，名爲 *DC_Initialize*。此函式的可變參數（variable arguments）包括有 base I/O port address（一個 DWORD），IRQ（也是一個 DWORD），以及 device 名稱（一個指標，指向一個字串）。爲了對 *DC_Initialize* 回應，port driver 應該呼叫 *VCOMM_Add_Port* 以樹立管理此一 port 的機制。在我的例子中，我以下面的方法完成這個步驟：

```
#0001  #include <stdarg.h>
#0002  ...
#0003  BOOL DriverControl(DWORD cf, DEVNODE devnode,
#0004      DWORD refdata, ...)
#0005      {                          // DriverControl
#0006      va_list ap;
#0007      va_start(ap, refdata);
#0008
#0009      DWORD iobase = va_arg(ap, DWORD);
#0010      DWORD irq    = va_arg(ap, DWORD);
#0011      char *name   = va_arg(ap, char *);
#0012
```

```
#0013     va_end(ap);
#0014
#0015     CSerialPort* port = new CSerialPort(name,
#0016        iobase, irq, devnode);
#0017     if (!port->AddPort(refdata))
#0018        {                    // unable to add port
#0019        delete port;
#0020        return FALSE;
#0021        }                    // unable to add port
#0022
#0023     return TRUE;
#0024     }                       // DriverControl
#0025  ...
#0026  BOOL CPort::AddPort(DWORD refdata)
#0027     {                       // CPort::AddPort
#0028     if (!VCOMM_Add_Port(refdata, (PFN) CPort::PreOpen, m_name))
#0029        return FALSE;
#0030     if ((m_contend = (PCONTENTIONPROC)
#0031        VCOMM_Get_Contention_Handler(m_name)))
#0032        m_resource = VCOMM_Map_Name_To_Resource(m_name);
#0033     return TRUE;
#0034     }                       // CPort::AddPort
```

VCOMM_Add_Port 的第一個參數是我們要交給 driver control procedure 的一份參考資料。第二個參數是個函式位址，VCOMM 可以呼叫它以打開 port。本例之中，它將是 *CPort* class 的 *PreOpen* member function。第三個參數是 port 名稱（例如 "COM2"），其值與 driver control procedure 的可變參數中的第三個參數相同。我將在稍後討論呼叫 *VCOMM_Get_Contentin_Handler* 的目的。

爲了對這段碼有完整的感覺，你必須看下面這段節錄。此節錄來自 *CSerialPort* 和 *CPort* classes 的完整宣告。*CPort* 的完整宣告出現於本章稍後。*CSerialPort* 的宣告在書附碟片中是以機器才能閱讀的型式存在。以下就是這段節錄：

```
#0001  class CPort
#0002     {
#0003     public:
#0004        CPort(char *name, DWORD iobase, DWORD irq, DWORD devnode);
#0005        char           m_name[8];
#0006        DWORD          m_iobase;
#0007        DWORD          m_irq;
```

```
#0008         DEVNODE        m_devnode;
#0009         PCONTENTIONPROC m_contend;
#0010         DWORD          m_resource;
#0011     };
#0012
#0013     class CSerialPort : public CPort
#0014     {
#0015     public:
#0016         CSerialPort(char *name, DWORD iobase, DWORD irq,
#0017             DWORD devnode);
#0018     };
```

這個片段說明一個事實，那就是，我定義了兩個 C++ classes，用以實作此 driver 範例。*CPort* 封裝了 VCOMM port driver 之中所有獨立於 device 的機能。*CSerialPort* 衍生自 *CPort*，實作一個 8250 相容的 serial port driver。一如你所想像，*CPort* 的建構式（constructor）會將成員變數初始化，並執行其他雜務，但並不喚起 VxD services（不像 *_HeapAllocate* 那樣會被 operator *new* 暗中呼叫）。

技術文件注意事項 有關 VCOMM service 及其保護模式 API，以及 port driver 的資料結構和函式，可以在 Windows 95 DDK 光碟中的 VCOMM.DOC 找到。但是這份文件中有許多錯誤。VCOMM.DOC 說，driver control function 不會傳回一個數值，這是對的，但其傳回值事實上變成了 *VCOMM_Register_Port_Driver* 的傳回值。如果 driver control function 真的是一個 void 函式，那麼如果呼叫 *VCOMM_Add_Port* 失敗，或其他什麼事發生錯誤的話，將沒有辦法放棄對 port driver VxD 的載入。VCOMM.DOC 也將唯一一個函式名稱誤拼爲 *DC_InitializePort*，事實上應該是 *DC_Initialize*。最後一點，VCOMM.DOC 把可變參數的次序搞亂掉了，它說 port 名稱是第一個參數，而事實上那是最後一個參數。我已經盡力在這本書中記錄正確的事情，因爲沒有其他方法可以讓一個真實的 driver 有效運作。書附光碟中的 WinHelp 檔案也已經正確註解了我所發現的事實。

打開 Port

在你呼叫 *VCOMM_Add_Port* 之後，VCOMM 終於有了一個函式位址，它可以呼叫此函式，打開 port。在我的 driver 實例之中， 我將此函式寫爲 *CPort* 的一個 static member function，如下所示：

```
#0001  PortData* CPort::PreOpen(char *name, HVM hVM, int* pError)
#0002      {                        // CPort::PreOpen
#0003      CPort* port;
#0004
#0005      for (port = CPortAnchor; port; port = port->m_next)
#0006          if (strcmp(name, port->m_name) == 0)
#0007              {                // try to open port
#0008              if (port->m_open)
#0009                  {            // already open
#0010                  *pError = IE_OPEN;
#0011                  return NULL;
#0012                  }            // already open
#0013              if (port->Open(hVM, pError))
#0014                  {            // opened okay
#0015                  port->m_open = TRUE;
#0016                  return &port->m_pd;
#0017                  }            // opened okay
#0018              port->Release();
#0019              return NULL;
#0020              }                // try to open port
#0021
#0022      *pError = IE_HARDWARE;
#0023      return NULL;
#0024      }                        // CPort::PreOpen
```

PreOpen 函式的第一個參數是 VCOMM 嘗試打開的 port 名稱。這個名稱最初在應用程式中是 *OpenComm* 或 *CreateFile* 的一個參數，在 client VxD 中則是 *VCOMM_OpenComm* 的一個參數。這個名稱應該吻合我們的 driver 向 VCOMM 註冊（經由呼叫 *VCOMM_Add_Port*）的 ports 之中的一個。爲了滿足 VCOMM 對此 open 函式的呼叫，一個 port driver 需要某種方法，用以找到它自己的私有資料結構（此結構以此名稱開始）。在我的 driver 中，係使用一個串列（linked list），以一個 static member

variable（*CPortAnchor*）表示，並以 member variables（*m_next* 和 *mprev*）串接起來。*CPort* 建構式和解構式則用來維護這個串列。

open 函式的第二參數是一個 VM handle，我將以之做爲競爭管理協定中的一部份（這個主題將在「競爭管理」這一節中討論）。第三參數（也就是最後一個參數）是某個 DWORD 的位址，其中記錄著 open 函式的任何失敗原因。可能的結果包括 IE_OPEN（port 已被打開）和 IE_HARDWARE（port 不存在，或是有其他硬體問題，使它無法被打開）。

open 函式的目的是要完成資料結構的初始化、在一個競爭管理協定中保留 device、並傳給 VCOMM 一個已經完全設好初值的 *PortData* 結構的位址。VCOMM 和 port driver 共享這個 *PortData* 結構，以便在 device 被打開期間能夠管理它。事實上，VCOMM 使用這個位址做爲呼叫所有 port driver 函式（open 函式除外）的一個 handle 參數，

PortData 結構

瞭解 *PortData* 結構，是學習 VCOMM 和一個 port driver 如何共同管理一個 open port 的關鍵。**圖 14-6** 說明這個重要結構的佈局。由於 VCOMM.DOC 對於 *PortData* 結構的說明是如此地不完整，我將一個欄位一個欄位地對此結構進行解釋。

PDLength 欄位內含你的結構長度，如此一來 VCOMM 就能夠檢驗你的確使用了正確的 *PortData* 結構 -- 和你的 driver 建造時所宣告的一樣。你可以簡單地設定其值爲 *sizeof(PortData)*。*PDVersion* 欄位是版本號碼，目的十分類似。我使用 1.10 做爲版本號碼，所以我們可以把這個欄位設爲 0x010A。

```
typedef struct _PortData {
  WORD PDLength;                // 00 sizeof (PortData)
  WORD PDVersion;               // 02 version of struct
  PortFunctions *PDfunctions;   // 04 table of port driver function
  DWORD PDNumFunctions;         // 08 number of functions in PDfunctions table
  DWORD dwLastError;            // 0C 0 or error resulting from last port driver function
  DWORD dwClientEventMask;      // 10 event mask set by client
```

```
 DWORD lpClientEventNotify;     // 14 client's event notification procedure
 DWORD lpClientReadNotify;      // 18 client's receive notification procedure
 DWORD lpClientWriteNotify;     // 1C client's transmit notification procedure
 DWORD dwClientRefData;         // 20 reference data for lpClientEventNotify
 DWORD dwWin31Req;              // 24 reserved for use by VCOMM
 DWORD dwClientEvent;           // 28 reserved for use by VCOMM
 DWORD dwCallerVMId;            // 2C reserved for use by VCOMM
 DWORD dwDetectedEvents;        // 30 default location to store currently detected events
 DWORD dwCommError;             // 34 current communication error flags
 BYTE bMSRShadow;               // 38 default shadow of Modem Status Register
 WORD wFlags;                   // 39 reserved for use by VCOMM
 BYTE LossByte;                 // 3B contention management flag
 DWORD QInAddr;                 // 3C input queue address
 DWORD QInSize;                 // 40 input queue size
 DWORD QOutAddr;                // 44 output queue address
 DWORD QOutSize;                // 48 output queue size
 DWORD QInCount;                // 4C number of bytes now in input queue
 DWORD QInGet;                  // 50 input queue offset to get next byte from
 DWORD QInPut;                  // 54 input queue offset to put next byte into
 DWORD QOutCount;               // 58 number of bytes now in output queue
 DWORD QOutGet;                 // 5C output queue offset to get next byte from
 DWORD QOutPut;                 // 60 output queue offset to put next byte into
 DWORD ValidPortData;           // 64 reserved for use by VCOMM
 DWORD lpLoadHandle;            // 68 reserved for use by VCOMM
 COMMTIMEOUTS cmto;             // 6C reserved for use by VCOMM
 DWORD lpReadRequestQueue;      // 80 reserved for use by VCOMM
 DWORD lpWriteRequestQueue;     // 84 reserved for use by VCOMM
 DWORD dwLastReceiveTime;       // 88 default location to store time of last receive
 DWORD dwReserved1;             // 8C reserved for use by VCOMM
 DWORD dwReserved2;             // 90 reserved for use by VCOMM
} PortData;                     // [94]
```

圖 14-6　PortData 結構佈局（譯註：你可以在 DDK\INC32\VCOMM.H 中找到）

PDfunctions 欄位是「函式指標表格」的位址，*PDNumFunctions* 欄位是表格中的指標個數。雖然 *PDfunctions* 被定義爲一個指標指向一個 *PortFunctions* 結構，但它其實指向一個函式指標陣列。這個函式指標表格的目的有點類似 C++ class object 中的一個虛擬函式表格，因爲它會導引 VCOMM 到你的 port driver 服務函式上。我將在稍後的「Port Driver Functions」中對此表格做更詳細的說明。

dwLastError 欄位內含 0 或是 port driver 函式最近所發生的錯誤代碼。表 **14-5** 列出此

變數的可能數值。如我們所見，所有的 port driver functions（除了 *Open*）都是 Boolean 函式，如果成功就傳回 TRUE，如果失敗就傳回 FALSE。雖然有一個 port driver 函式名爲 *GetError*，但它只是傳回 *dwLastError* 而已。Microsoft SERIAL.VXD 所實作的 *GetError* 函式事實上什麼也沒做。VCOMM 確實是使用 *dwLastError* 來爲它自己的呼叫者診斷錯誤，不需要呼叫 port driver！

錯誤代碼	說明
IE_BADID	表示這個 device 是一個非法的或未被支援的 device
IE_OPEN	表示 port 已被打開
IE_NOPEN	表示 port 未被打開
IE_MEMORY	表示沒有足夠的記憶體
IE_DEFAULT	表示預設的參數中有一個錯誤
IE_INVALIDSERVICE	表示 port driver 不支援此函式（功能）
IE_HARDWARE	表示硬體並不存在
IE_BYTESIZE	表示 DCB 中有一個不合理的 byte size
IE_BAUDRATE	表示 DCB 中有一個不合理的 baud rate
IE_EXTINVALID	表示使用了一個非法的 escape 函式
IE_INVALIDPARAM	表示傳遞了一個非法參數
IE_TRANSMITCHARFAILED	表示有一個優先權字元（priority character）正等待被送出

表 14-5 PortData 的 dwLastError 欄位的可能內容

VCOMM 會維護 *lpClientEventNotify*, *lpClientReadNotify*, 和 *lpClientWriteNotify* 三個欄位。每一個欄位內含一個 client callback 函式的位址，此位址可以在 ring3 中定址出來。Port driver 應該爲 event, receive, 和 transmit notifications 維護個別的函式指標。VCOMM 呼叫你的 *EnableNotification*, *SetReadCallback*, 和 *SetWriteCallback* 函式以註冊這些函式指標。這些指標代表 VCOMM 中的函式，它們把 client callbacks 做了一層外包裝（wrap）。每一個 callback 各有一個（各不相同的）DWORD，表示對應的 reference

data，在註冊動作後提供給你。你必須保留這三個 DWORDs 以便持有 callbacks 的 reference data values。

由 *EnableNotification* 所註冊的 event notification callback function 負責處理被 port driver 所偵測的 communication events。表 **14-6** 列出可能的 events flags。一旦 port driver 偵測到一個 event，它就在目前的 event word 中設定適當的 event flag。呼叫了 *GetEventMask* 函式之後，driver 會清除 events。*PortData* 結構中的 *dwDetectedEvents* 欄位並非如你所想像是放置（記錄）events 的地方，port driver 應該另外維護一個 LPDWORD 變數，以它指向目前的 event word，後者可以因爲呼叫 *SetEventMask* 而被改變。這個指標變數的初值可能是 *dwetectedEvents* 欄位位址，因爲這個欄位並不被使用於任何其他用途上。

Port driver 使用 *dwCommError* 來記錄通訊錯誤。表 **14-7** 列出各種 flag bits 及其代表的各種錯誤意義。直到呼叫了 *ClearError* 這個 port driver function，錯誤才會被清除。如果存在任何錯誤，就會中止 Read 動作，除非有人發出 IGNOREERRORONREADS escape function。

Event Flag	說明
EV_RXCHAR	表示收到了一個字元（任何字元）
EV_RXFLAG	表示收到了特殊的 "event character"（在 DCB 中有指定）
EV_TXEMPTY	表示傳輸用的 queue 是空的
EV_CTS	表示 CTS bit 處於「已被改變」狀態（changed state）
EV_DSR	表示 DSR bit 處於「已被改變」狀態（changed state）
EV_RLSD	表示 carrier detect line 的狀態已被改變
EV_BREAK	表示收到了一個 break signal
EV_ERR	表示發生了一個 Line Status interrupt（代表一個錯誤）
EV_RING	表示 RI (Ring Indicate) line 變成 active
EV_PERR	表示發生了一個 printer error
EV_CTSS	表示 CTS 是 active

Event Flag	說明
EV_DSRS	表示 DSR 是 active
EV_RLSDS	表示 carrier detect line 是 active
EV_RingTe	表示 ring trailing-edge 被偵測到了
EV_TXCHAR	表示傳輸了一個字元
EV_DRIVER	表示發生了一個與某特定 driver 相關的 event
EV_UNAVAIL	表示這個 port 被「偷」了
EV_AVAIL	表示一個先前被「偷」的 port 現在又再度備妥可用了

表 14-6 Communication event flags

Error Flag	說明
CE_RXOVER	表示 input queue 已滿
CE_OVERRUN	表示有一個 receive overrun error
CE_RXPARITY	表示有一個 parity error
CE_FRAME	表示有一個 framing error
CE_BREAK	表示有一個 break
CE_CTSTO	表示有一個 CTS timeout
CE_DSRTO	表示有一個 DSR timeout
CE_RLSDTO	表示有一個 carrier detect timeout
CE_TXFULL	表示 output queue 已滿
CE_PTO	表示有一個 printer timeout
CE_IOE	表示有一個 printer I/O error
CE_DNS	表示有一個「印表機未選取」的錯誤
CE_OOP	表示印表機沒紙了
CE_MODE	表示所要求的模式並未被支援

表 14-7 dwCommError 之中可能出現的 Error flags

一個標準的 UART 會產生一個中斷，使 CPU 注意 Modem Status 暫存器中的變化。Port driver 應該維護一個 PBYTE 變數，用它指向一個位置，使中斷處理常式能夠儲存 Modem Status 暫存器中即將成爲「現行值」的內容。。*PortData* 結構的 *bMSRShadow* 欄位就是這個「暫存器之 shadow copy」的最初位置。呼叫 port driver function *SetModemStatusShadow* 可以改變這個位置。

官方文件上說，*LossByte* 被保留給 VCOMM 使用。這並不是真的，標準 COM ports 的一個 port driver 必須檢閱並修改這個 byte 的較低位元，以便能夠正確處理 VCD 所實作的競爭協定。我將在下一節詳細討論所謂的競爭協定（contention protocol）。Port driver 使用 *PortData* 結構中所保留的欄位，以 "ring buffers" 的方式來管理 input queue 和 output queue。**圖 14-7** 說明各個 buffering 參數之間的關係。Port driver 的 Setup 函式會記錄每一個 queue 的 buffer 位址（*QInAddr* 和 *QOutAddr*）及大小（*QInSize* 和 *QOutSize*）。Buffers 可以屬於一個外部呼叫者所有，也可以由 port driver 爲它配置空間。每一個 queue 有一個計數器（*QInCount* 和 *QOutCount*），表示目前在 buffer 之中存在有多少 bytes 的資料。程式如果要加資料到 buffer 內部，應該加在 put 指標（*QInPut* 或 *QOutPut*）所指之處，此指標然後會累加 1。程式如果要從 buffer 之中移除資料，應該自 get 指標（*QInGet* 或 *QOutGet*）處移除，然後它也會累加 1。程式也可能調整相關的計數器值，表示它剛剛新增或移除資料。"get" 和 "put" 兩指標相減，應該等於 buffer 中的 data bytes。由於 buffer 是一個 ring buffer，"get" 指標可能大於 "put" 指標，意思是 buffer 之中被 data 所佔據的空間會轉折回到 buffer 的頭部。

爲了能夠使一個閒置的（idle）port 發生 "timing out"，port driver 必須記錄最後收到資料時的系統時間。有一個 escape 機能（*SETUPDATETIMEADDR*）可以指定某個 DWORD 位置，用以接收 time stamp，所以 driver 必須提供它自己的 PDWORD 變數，指向目前的 time stamp 位置。Driver 應該爲此變數設定初值，使它指向 *PortData* 結構的 *dwLastReceiveTime* 欄位。

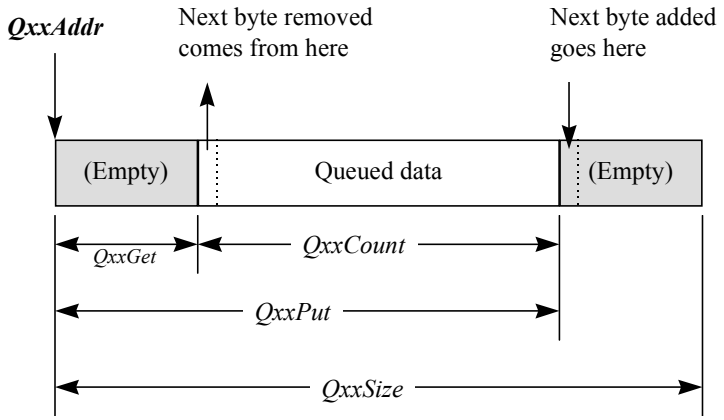


圖 14-7 「port buffering 參數」之間的關係

VCOMM 保留了上述說明之外的其他 *PortData* 結構欄位，以爲己用。除了欄位名稱已明白宣示爲保留者之外，這樣的欄位還包括 *dwClientEventMask*, *dwWin31Req*, *dwClientEvent*, *dwCallerVMId*, *wFlags*, *ValidPortData*, *lpLoadHandle*, *cmto*, *lpReadRequestQueue*, 以及 *lpWriteRequestQueue*。

競爭管理（Managing Contention）

Windows 不允許兩個應用程式同時共享一個 serial port。這似乎顯而易見，因爲兩個應用程式有不同的通訊設定，它們因而不能夠透過一個共享的 serial port 完成輸入和輸出的多重通訊。Contention management 是一個一般化的術語，用來描述 Windows 如何確保一次只能有一個應用程式存取某個 port。

每個 port 的 software registry key 內含一個 named value "Contention"，用來任命某個 VxD 管理 VCOMM port drivers 之間對於 port 的搶奪。VCD 提供 serial ports 的競爭管理，而 Virtual Printer Device（VPD）則對 parallel ports 提供相同的服務。Contention VxD 會將 port 虛擬化，以便讓虛擬機器軟體可以直接使用 port。將 port 虛擬化，意味著使用前一章介紹的技術來 trapping "port I/O operations" 並 hooking 相關的硬體中斷。把虛擬化動作和競爭管理組合到單一的 VxD 之中是合理的，因爲一個 VxD 之所以能夠發

現 MS-DOS 或其他虛擬機器軟體嘗試使用 port，唯一的方法就是 trapping 一個 I/O operation。此外，一如我們在前一章所見，將硬體虛擬化，一部份工作就是處理不同虛擬機器之間的硬體搶奪戰。

Contention VxD 內含一個 contention function。VCOMM 送出一個 *GET_CONTENTION_HANDLER* 系統控制訊息給 VxD，於是獲得這個函式的位址。VxD 的回應是在 EAX 暫存器中傳回 contention function 的位址，並清除 carry flag。Port driver 應該呼叫 VCOMM 以求獲得這個 contention function 位址。在我所舉的實例中的 *CPort::AddPort* 函式裡頭，我使用以下呼叫：

```
m_contend = (PCONTENTIONPROC)
   VCOMM_Get_Contention_Handler(m_name);
```

其中 *m_contend* 是一個成員變數，用來放置 contention function 位址（當 port 保持開啓狀態）。Contention function 事實上是 6 個不同的函式，以一個子機能代碼（subfunction code）做爲第一參數而區別之（請看**表 14-8**）。這些函式的細節將在以下數頁討論。

子機能代碼（Subfunction Code）	說明
MAP_DEVICE_TO_RESOURCE	取得一個具名的 port 的 "contention resource handle"
ACQUIRE_RESOURCE	獲得一個 port 的擁有權
STEAL_RESOURCE	偷一個 port 的擁有權（從其目前的擁有者手上）
RELEASE_RESOURCE	釋放先前獲得的 port
ADD_RESOURCE	將一個 port 加到串列中（串列由 contention function 管理）
REMOVE_RESOURCE	從串列中移除一個 port(串列由 contention function 管理）

表 14-8　Contention functions 的子機能代碼（Subfunction codes）

MAP_DEVICE_TO_RESOURCE 子機能 此一子機能的呼叫情形如下：

```
DWORD hResource = Contend(MAP_DEVICE_TO_RESOURCE, PCHAR name,
   DEVNODE devnode, DWORD IOBase);
```

如果 contention function 不知道 name port，就傳回 0。否則傳回一個對 contention function 有意義的 resource handle。Port driver 不需要直接呼叫 contention function 以執行資源映射（resource mapping），因爲 VCOMM 提供一個 service，可以隱藏這個特別的介面。在我的實例中，*CPort::AddPort* 函式內含這樣的呼叫動作：

```
m_resource = VCOMM_Map_Name_To_Resource(m_name);
```

其中 *m_resource* 是一個成員變數，當 port 持續開啓時，它內含 resource handle。請注意，這裡的子機能代碼並不真正吻合 VCOMM service 名稱。

ACQUIRE_RESOURCE 子機能 此一子機能的呼叫情形如下：

```
DWORD hContend = Contend(ACQUIRE_RESOURCE, DWORD hResource,
   PFN NotifyProc, DWORD NotifyData, BOOL bSteal);
```

它要求 contention function 將「*hResource* 所代表之 port」的擁有權指定給呼叫者。*hResource* 是前一次呼叫 *MAP_DEVICE_TO_RESOURCE* 時所傳回的值，如果 *bSteal* 參數是 TRUE，contention function 會企圖從 port 目前的擁有者手中「偷取」這個 port（如果這個 port 目前被擁有的話）。但如果 *bSteal* 是 FALSE 而 port 目前已被擁有，contention function 並不會嘗試重新指定其主人。*NotifyProc* 是一個 notification function 位址，它將在這場指定擁有權的過程中被呼叫。*NotifyData* 是一個任意值，被傳遞給 notification function 做爲參考資料。

如果 contention function 能夠指定 resource，它會傳回一個 DWORD handle，代表這個 port，否則就傳回 NULL。事實上，從一次成功呼叫中所傳回的 handle，和「被當做 *hResource* 參數」的 resource handle 是一樣的。在 VCD 的 contention function 中，事實上，這兩個 handles 都是 VCD 內部針對這個 port 的 *VCD_COM_Struc* 結構位址。

因此，你可以視 *ACQUIRE_RESOURCE* 爲一個 Boolean function，並測試傳回值是否爲 0。

在我的例子中，我寫了一個 *CPort::Acquire* 成員函式，用以獲得一個 port 的擁有權，此 port 是在下面這個程序中被打開的：

```
BOOL CPort::Acquire(HVM hVM)
   {                          // CPort::Acquire
   if (m_contend)
      {                       // have a contention procedure
      if (!m_resource)
         return FALSE;        // but no resource handle
      m_hContend = (*m_contend)(ACQUIRE_RESOURCE, m_resource,
         pPortStolen, this, TRUE);     // try to steal if owned
      if (m_hContend)
         {                    // we've got it
         m_owner = hVM;
         return TRUE;
         }                    // we've got it
      return FALSE;
      }                       // have a contention procedure
   m_owner = hVM;
   return TRUE;
   }                          // CPort::Acquire
```

（*pPortStolen* 變數是一個 static 指標，指向 *CPort::OnPortStolen* 函式，稍後我將呈現此一函式。我需要這樣一個指標，才能克服 C++ 編譯器的一個限制，將一個指向 member function 的指標當做 contention function 的參數）

Notification function 收到兩個參數：第一個是 *NotifyData* reference data，第二個是個 Boolean 值 -- 如果是 TRUE，表示 port 已經被此 driver 獲得；如果是 FALSE，表示別的 driver 失去了這個 port。contention VxD 會在 *ACQUIRE_RESOURCE* call 的過程中呼叫這個函式，也會在 port 擁有權移轉的時候呼叫它。在範例程式中，我使用一個 static 成員函式來處理擁有權移轉事宜（它必須是一個 static，因爲 VCB 直接呼叫它，就像面對一個一般的 C 函式一樣）：

```
BOOL CPort::OnPortStolen(CPort* port, BOOL owned)
   {                              // OnPortStolen
```

```
if (owned)                          // we own the port
   port->m_pd.LossByte &= ~1;       // set low bit to 0
else                                // we lost the port
   port->m_pd.LossByte |= 1;        // set low bit to 1
return TRUE;                        // answer !owned with "OK"
}                                   // OnPortStolen
```

（*CPort* 中的 *m_pd* 成員，是這個 port 的 *PortData* 結構）只有當 *owned* 參數是 0（意思是 contention VxD 打算偷取這個 port），此函式的傳回值才有意義。如果你傳回 TRUE，就表示允許這個 port 被偷，如果傳回 FALSE，就表示不允許它被偷。

當 VCOMM 通知說擁有權已經改變時，我的 *OnPortStolen* 函式模仿官方的 SERIAL.VXD 的作爲。當 port driver 擁有這個 port，*LossByte* 的低位元爲 0；否則爲 1。Port driver 中的各個函式，像是中斷處理常式以及其他函式，如果要對 UART 進行 I/O operations，都必須先檢查這個位元。如果位元爲 1，意思是 contention VxD 已經「偷走」了這個 port，此時 port driver 可以嘗試發出一個 STEAL_RESORCE call，把它再偷回來。

STEAL_RESOURCE 子機能 此一子機能的呼叫情形如下：

```
BOOL bStolen = Contend(STEAL_RESOURCE, DWORD hResource,
                       PFN NotifyProc);
```

其中 *hResource* 參數是 *MAP_DEVICE_TO_RESOURCE* call 所傳回的 resource handle。而 *NotifyProc* 則是先前在 *ACQUIRE_RESOURCE* call 中所使用的同一個 notification function 位址。Notification function 使用 *NotifyProc* 位址，不但可以鑑定呼叫者爲 resource 的先前擁有者，也可以在它重獲擁有權時通知它。所以對呼叫端而言，對於 *ACQUIRE_RESOURCE* call 和 *STEAL_RESOURCE* call，總是使用相同的函式位址是十分重要的。

如果 contention VxD 能夠重新指定 port 的控制權，*STEAL_RESOURCE* 就傳回 TRUE，否則就傳回 FALSE。在回返之前，它會呼叫前一個擁有者的 notification function，表示遺失了這個 port（owned 參數將被設爲 FALSE），它也會呼叫新擁有者的

notification function，以表示控制權的恢復（owned 參數將被設爲 TRUE）。由於 *STEAL_RESOURCE* 呼叫 notification functions，所以一個 port driver 並沒有強烈必要去測試傳回值。在我的例子中，我模仿官方的 SERIAL.VXD 的行爲來「偷取」port：

```
BOOL CPort::StealPort()
   {                          // CPort::StealPort
   if (!(m_pd.LossByte & 1))
      return TRUE;            // we never lost it
   if ((*m_contend)(STEAL_RESOURCE, m_hContend, pPortStolen))
      {                       // stole it back
      m_pd.LossByte &= ~1;
      return TRUE;
      }                       // stole it back
   return FALSE;
   }                          // CPort::StealPort
```

爲什麼不使用 VCOMM_Acquire_Port？

按官方文件 VCOMM.DOC 中的說法，一個 port driver 應該能夠呼叫 *VCOMM_Acquire_Port* 以期獲得一個 port 擁有權，並呼叫 *VCOMM_Release_Port* 以釋放 port 擁有權。不敢相信這段話！首先，VCOMM.DOC 中對 *VCOMM_Acquire_Port* 的說明是錯誤的。正確的方式是：

```
DWORD hContend = VCOMM_Acquire_Port((HANDLE)hPort, DWORD lPortNum,
                                    HVM hVM, DWORD lFlags, char *vxdname);
```

其中 *hPort* 是針對此一 port 的 VCOMM port handle（也就是 *PortData* 結構的位址），*lPortNum* 若不是以 1 開始的 port 編號（1 表示 COM1），就是這個 port 的基礎 I/O 位址；到底爲何，視 *lFlags* 的 bit2 而定。*hVM* 是即將擁有此一 port 的 VM 的 handle，也可能是 -1，表示你的 VxD 按望了這個 port。*lFlags* 是以 OR 動作將 01h 和 02h 組合起來，01h 表示「即使 port 已被擁有，也要把它偷過來」，02h 表示「*lPortNum* 代表的是 I/O 基礎位址，而非 port 號碼」。*vxdname* 是一個以 null 爲結尾的字元，代表你的 VxD 名稱（如果你指定 *hVM* 爲 -1 的話）。如

果你的 *hVM* 指定真正的 VM handle，那麼 *vxdname* 參數就派不上用場。你大概不會想接管 port，因為這麼一來你就得扛起 port 虛擬化的責任。如果你真的有這個勇氣，那麼請你先仔細閱讀 VCD 原始碼（在 Windows 95 DDK 光碟片的 \COMM\SAMPLES\VCD 目錄中）。

如果我要在我的例子中呼叫這個函式，動作大約像這樣：

```
m_hContend = VCOMM_Acquire_Port((HANDLE)&m_pd, m_iobase, hVM, 3, NULL);
```

VCOMM 將呼叫 *VCD_Acquire_Port_Windows_Style* 小聲稱對此 port 的主權。VCD 並且清除你的 *PortData* 結構中的 *LossByte* 欄位的較低位元組（low-order byte），讓你知道你的 port 已經被「偷」走了。不幸的是，你沒有任何方法可以把 port 再偷回來。如果你直接呼叫 contention function(這也是我所批評的方法)，你的 callback 函式（例如 *CPort::OnPortStolen* 函式）會警告你說你的 port 已經被偷走了。稍後你可以再次發出 *STEAL_RESOURCE call*，重得控制權，一如 SERIAL.VXD 的作為。

RELEASE_RESOURCE 子機能　此一子機能的呼叫情形如下：

```
Contend(RELEASE_RESOURCE, DWORD hResource, PFN NotifyProc);
```

其中 *hResource* 參數是 *MAP_DEVICE_TO_RESOURCE* 傳回的 resource handle，*NotifyProc* 則是先前在 *ACQUIRE_RESOURCE* call 中所使用的同一個 notification function 位址。回返之前，*RELEASE_RESOURCE* 會呼叫 notification function，表示失去了這個 resource（*owned* 參數會被設爲 FALSE）。在我的實例當中，我寫了一個成員函式，在關閉 port 的程序中撤回控制權：

```
void CPort::Release()
   {                        // CPort::Release
   m_owner = NULL;
   if (m_hContend)
      {                     // release port
```

```
        (*m_contend)(RELEASE_RESOURCE, m_hContend,
           pPortStolen);
        m_hContend = NULL;
        }                       // release port
    }                           // CPort::Release
```

ADD_RESOURCE 子機能　此一子機能的呼叫情形如下：

```
BOOL bAdded = Contend(ADD_RESOURCE, DWORD iobase, DWORD irq,
                      DEVNODE devnode);
```

其中的 *iobase* 是 port 的 base I/O address，*irq* 是被配置的 IRQ 號碼，*devnode* 是一個 Configuration Manager device node 的位址。VCOMM 使用 *ADD_RESOURCE* 來通知一個 contention VxD 說，Configuration Manager 已經偵測到一個新的 port。傳回值若爲 TRUE，表示 contention VxD 能夠更新其內部資料結構以反映出新 port 的存在，否則爲 FALSE。

REMOVE_RESOURCE 子機能　此一子機能的呼叫情形如下：

```
void Contend(REMOVE_RESOURCE, DWORD iobase);
```

其中的 *iobase* 是 port 的 base I/O address。當 Configuration Manager 移除在其下的 DEVNODE，VCOMM 即呼叫 *REMOVE_RESOURCE*。

Port Driver Functions

一旦 VCOMM 成功呼叫了 port driver 的 open 函式，打開了一個 port 之後，它就可以呼叫某「函式指標表格」中的函式位址（此表格由 *PortData* 結構的 *PDfunctions* 欄位指出），以便和 port driver 溝通。

一共有 22 個這樣的函式，列出於稍早的**表 14-4** 中。但函式指標表格中並不包括第 23 號函式 *Open*。我將在這一節中描述它們，以便澄清官方文件 VCOMM.DOC 中的錯誤。我將藉由一個 driver 實例中的 *CPort* class，說明這些函式動作中獨立於 device 的部份。我將簡短說明由 *CSerialPort*（衍生自 *CPort*）所提供的一些與 device 無關的機能。

與 Virtual Functions 打交道

一如先前我曾經說過，一個 VCOMM port driver 的架構似乎很適合以一個 C++ derived class 加上虛擬函式來完成。將 VCOMM 的 C 函式呼叫轉換爲 C++ 成員函式，面臨一些小小的挑戰。所有這些 port driver 函式都期望獲得一個 port handle（那事實上是一個指標，指向一個 *PortData* 結構）做爲第一參數；它們都傳回一個 Boolean 值，表示成功或失敗。這 22 port driver 函式只有在型態上以及其他參數的個數上有所不同，爲了產生必要的函式指標表格，我首先建立一個巨集，把一個 port handle 轉換爲一個 this 指標，準備給成員函式使用：

```
#define CPORT(hp) ((CPort *) ((DWORD) hp - \
   FIELDOFFSET(CPort, CPort::m_pd)))
```

其中 *FIELDOFFSET* 是一個標準巨集，用來計算結構中的成員偏移位置：

```
#define FIELDOFFSET(type, field) ((DWORD)(&((type *)0)->field))
```

CPORT 巨集利用一個事實，那就是，一個成功的 *CPort::PreOpen* 函式呼叫，傳回給 VCOMM 的是一個 *PortData* 結構位址，此結構將成爲 *CPort* 物件中的 *m_pd* 成員。VCOMM 以此位址當做 port handles，用來呼叫其他的 port driver 函式。爲了在此 driver 的除錯版中提供一些功能，我也實作了一個 *m_signature* 欄位，內含一個人類可解讀的字串，以及一個 *ASSERT_VALID_CPORT* 巨集，用來在每一個成員函式中進行測試。

由於懶於打太多的字，所以我寫了一些巨集，幫助我宣告 C 函式，以便在 VCOMM 和我的 C++ 成員函式之間提供介面：

```
#define PF0(f) BOOL f(PortData* hp) {return CPORT(hp)->f();}
#define PF1(f, a1t) BOOL f(PortData* hp, a1t a1)\
   {return CPORT(hp)->f(a1);}
[etc.]
```

PF0 用於無參數的成員函式身上，*PF1* 用於有一個參數的成員函式身上，依此類推。然後我使用這些巨集，寫下真正的函式定義：

```
PF2(ClearError, _COMSTAT*, int*)
PF0(Close)
[etc.]
```

這些句子產生出以下函式：

```
BOOL ClearError(PortData* hp, _COMSTAT* a1, int* a2)
    {
    return CPORT(hp)->ClearError(a1, a2);
    }

BOOL Close(PortData* hp)
    {
    return CPORT(hp)->Close();
    }

[etc.]
```

爲了宣告函式指標表格，我寫了另一個巨集，於是就可以不必一再重複地輸入相同的型別轉換動作：

```
#define PF (BOOL (*)())
```

最後，讓這個函式指標表格由 *PortData* 結構中的 *PDfunctions* 欄位指出，

```
PortFunctions functions =
{PF SetCommState, PF GetCommState, PF Setup, PF TransmitChar,
PF Close, PF GetQueueStatus, PF ClearError,
PF SetModemStatusShadow,
PF GetProperties, PF EscapeFunction, PF Purge, PF SetEventMask,
PF GetEventMask, PF Write, PF Read, PF EnableNotification,
PF SetReadCallback, PF SetWriteCallback, PF GetModemStatus,
PF GetCommConfig, PF SetCommConfig, PF GetError, NULL};
```

> 譯註：以下這段文字說明，用到 C++ 物件模型的知識。所謂 C++ 物件模型（C++ Object Model），範圍極廣，幾乎可以說 C++ 編譯器在程式底層所做的動作都可涵蓋於內。但以此處之應用而言，集中在 C++ 物件之記憶體佈局（memory layout）身上。欲對此一主題有更多瞭解，可參考 ??型與虛擬－物件導向的精髓（侯俊傑著/松崗/1998）第２章，或 *Inside The C++ Object Model* (Lippman /Addison Wesley/1996)。

讓我們追蹤一個 port driver 函式的呼叫實例，看看上述複雜的聯結關係實際上如何有效地運作。VCOMM 呼叫我們的 *CPort::PreOpen* 以求打開一個 port，該函式位址是透過

我們呼叫 *VCOMM_Add_Port* 而得。*PreOpen* 函式會呼叫 *CPort::Open* 函式，後者設定 *CPort* 物件的 *m_pd.PDfunctions* 成員，使它指向剛剛所顯示的函式表格。表格中的第五個元素（對應於 *PortFunctions* 結構中的 *pPortClose* 成員）指向稍早所顯示的 *Close* 函式。當 VCOMM 想要關閉這個 port，它便經由 *pPortClose* 呼叫 *Close* 的函式指標，並把我們的 *CPort* 結構中的 *m_p* 成員的位址當做唯一一個參數（*hp*）傳過去。*Close* 利用 *CPORT* 巨集將 *hp* 反算出 *CPort* 實體之起始位址，並呼叫該實體（instance）的 *CPort::Close* 成員函式。

我將所有的 port driver 成員函式宣告爲 virtual，允許它們在衍生類別中被改寫：

```
class CPort
{
...
virtual BOOL    ClearError(_COMSTAT* pComstat, int* pError);
virtual BOOL    Close();
...
};
```

因此，VCOMM 直接與之交談的 C 語言外包函式（wrappers），最終呼叫的若不是 base-class 函式，就是 derived-class 函式，視這些函式如何（根據一個特定的 port）設計而定。

順帶一提，你可以在 *PortFunctions* 陣列中使用一個 NULL 指標來代表任何你沒有實作的 port driver 函式 -- *SetModemStatusShadow* 除外（如果你的 driver 是一個 serial port driver 的話）。VCOMM 並不企圖呼叫一個其實是 NULL 的 port driver 函式（不幸的是，VCOMM 也不會執行任何預設處理動作）。剛剛說的那個例外，是因爲 UNIMODEM modem driver 經由函式指標表格呼叫你的 *SetModemStatusShadow* 函式，沒有先檢查它是否爲 NULL。如果你使用任何透過電話線的軟體（telephony applications），例如 HyperTerminal，最終你將獲得一個路徑，進入 UNIMODEM.VXD，完成此一呼叫。

我的 *PortFunctions* 表格中的最後一個項目（一個 NULL 指標），只是爲 port driver 的 IOCTL 函式保留的一個空間。Microsoft 並沒有公開說明此函式的呼叫方式。

CPort Class

我在 VPORT.H 檔中宣告 *Cport* 這個類別。完整檔案置於書附光碟片中。在你瞭解我所說明的這些函式之前，我想你需要以下的 .H 檔：

VPORT.H

```
#0001  // Vport.h -- Interface to CPort class
#0002  // Copyright (C) 1996 by Walter Oney
#0003  // All rights reserved
#0004
#0005  #ifndef VPORT_H
#0006  #define VPORT_H
#0007
#0008  typedef void (*PCOMMNOTIFYPROC)(PortData* hPort, DWORD refdata,
#0009     DWORD lEvent, DWORD lSubEvent);
#0010  typedef DWORD (*PCONTENTIONPROC)(int, ...);
#0011
#0012  ////////////////////////////////////////////////////////////
#0013  // Base class for serial ports
#0014
#0015  class CPort
#0016  {
#0017  public:
#0018    static CPort*   CPortAnchor;     // anchor of CPort structures
#0019
#0020  // Non-standard flags in m_NfyFlags (#define so we can use inline
#0021  // assembler with them):
#0022
#0023  #define CN_IDLE     0x80           // host has gone idle
#0024  #define CN_NOTIFY   0x40           // notifications enabled
#0025
#0026  // Flags in m_MiscFlags:
#0027
#0028  enum MISCFLAGS
#0029    {MF_TXQSET          = 0x0001     // transmit queue has been setup
#0030    ,MF_RXQINTERNAL     = 0x0002     // we allocated Rx queue
#0031    ,MF_TXQINTERNAL     = 0x0004     // we allocated Tx queue
#0032    ,MF_CLRTIMER        = 0x0008     // timer logic disabled
#0033    ,MF_IGNORECOMMERROR  = 0x0010    // ignore read errors
#0034    ,MF_DISCARD         = 0x0020     // discard input characters
#0035    ,MF_STATESETONCE     = 0x0040    // SetCommState done once
#0036    };
```

```
#0037
#0038  // Attributes
#0039
#0040  public:
#0041    CPort*          m_next;          // chain to next port
#0042    CPort*          m_prev;          // chain to previous port
#0043
#0044    PortData        m_pd;            // VCOMM PortData structure
#0045    _DCB            m_dcb;           // ring-0 device control block
#0046    DWORD           m_signature;     // signature "PORT"
#0047    char            m_name[8];       // name of port (COMx)
#0048    DWORD           m_iobase;        // base I/O address
#0049    DWORD           m_irq;           // IRQ number
#0050    DEVNODE         m_devnode;       // our device node
#0051    PCONTENTIONPROC  m_contend;      // contention function
#0052    DWORD           m_resource;      // resource handle
#0053    DWORD           m_hContend;      // contention handle
#0054    BOOL            m_open;          // true when port is open
#0055    HVM             m_owner;         // handle of VM that owns port
#0056    PDWORD          m_pRxTime;       // last receive time
#0057    PDWORD          m_pEvent;        // address of event word
#0058    DWORD           m_eventmask;     // mask for notification events
#0059    PBYTE           m_pMsrShadow;    // address of MSR shadow byte
#0060    DWORD           m_TxTrigger;     // transmit trigger
#0061    DWORD           m_RxTrigger;     // receive trigger
#0062
#0063    DWORD           m_EvData;        // reference data for EvNotify
#0064    DWORD           m_RxData;        // reference data for RxNotify
#0065    DWORD           m_TxData;        // reference data for TxNotify
#0066    PCOMMNOTIFYPROC  m_EvNotify;     // ring-zero notification
#0067                                     // procedure
#0068    PCOMMNOTIFYPROC  m_RxNotify;     // ring-zero read notification
#0069                                     // procedure
#0070    PCOMMNOTIFYPROC  m_TxNotify;     // ring-zero write notification
#0071                                     // procedure
#0072    WORD            m_MiscFlags;     // miscellaneous flags
#0073
#0074    BYTE            m_NfyFlags;      // flags for notifications
#0075
#0076  // Methods
#0077
#0078  public:
#0079    CPort(char *name, DWORD iobase, DWORD irq, DWORD devnode);
#0080    virtual ~CPort();
#0081
#0082    static void DeleteAll();
```

```
#0083   static PortData* PreOpen(char *name, HVM hVM, int* pError);
#0084   static BOOL OnPortStolen(CPort* port, BOOL stolen);
#0085   static void ManageTimer();
#0086   static void OnTimeout(DWORD extra, DWORD refdata);
#0087
#0088   BOOL AddPort(DWORD refdata);
#0089
#0090  // Overridable functions:
#0091
#0092   virtual BOOL Acquire(HVM hVM);
#0093   virtual void CallNotifyProc(int code);
#0094   virtual BOOL ClearError(_COMSTAT* pComstat, int* pError);
#0095   virtual BOOL Close();
#0096   virtual BOOL EnableNotification(PCOMMNOTIFYPROC pCallback,
#0097                                   DWORD refdata);
#0098   virtual BOOL EscapeFunction(DWORD lFunc, DWORD InData,
#0099                               PVOID pOutData);
#0100   virtual BOOL GetCommConfig(PCOMMCONFIG lpCC, PDWORD lpSize);
#0101   virtual BOOL GetCommState(_DCB* pDCB);
#0102   virtual BOOL GetEventMask(DWORD mask, PDWORD pEvents);
#0103   virtual BOOL GetProperties(_COMMPROP* pCommProp);
#0104   virtual BOOL GetQueueStatus(_COMSTAT* pComstat);
#0105   virtual BOOL GetError(int* pError);
#0106   virtual BOOL GetModemStatus(PDWORD pModemStatus);
#0107   virtual BOOL Open(HVM hVM, int* pError);
#0108   virtual BOOL Purge(DWORD qType);
#0109   virtual BOOL Read(PCHAR buf, DWORD cbRequest, PDWORD pRxCount);
#0110   virtual void Release();
#0111   virtual BOOL SetCommConfig(PCOMMCONFIG lpCC, DWORD dwSize);
#0112   virtual BOOL SetCommState(_DCB* pDCB, DWORD ActionMask);
#0113   virtual BOOL SetEventMask(DWORD mask, PDWORD pEvents);
#0114   virtual BOOL SetModemStatusShadow(PBYTE pShadow);
#0115   virtual BOOL SetReadCallback(DWORD RxTrigger,
#0116                           PCOMMNOTIFYPROC pCallback, DWORD refdata);
#0117   virtual BOOL Setup(PCHAR RxQueue, DWORD cbRxQueue,
#0118                      PCHAR TxQueue, DWORD cbTxQueue);
#0119   virtual BOOL SetWriteCallback(DWORD TxTrigger,
#0120                           PCOMMNOTIFYPROC pCallback, DWORD refdata);
#0121   virtual BOOL StealPort();
#0122   virtual BOOL TransmitChar(CHAR ch);
#0123   virtual BOOL Write(PCHAR buf, DWORD cbRequest, PDWORD pTxCount);
#0124
#0125  // Mini-driver callouts:
#0126
#0127   virtual BOOL  cextfcn(DWORD lFunc, DWORD InData,
#0128                           PVOID pOutData); // part of EscapeFunction
```

```
#0129    virtual BOOL  inicom(int* pError);    // part of Open
#0130    virtual BOOL  trmcom();              // part of Close
#0131
#0132    virtual void  BeginSetState();
#0133    virtual void  EndSetState(DWORD ChangedMask);
#0134    virtual void  Flush(DWORD qType);     // part of Purge
#0135    virtual BOOL  CheckState(_DCB* pDCB, DWORD ActionMask);
#0136    virtual DWORD GetProviderSubType();  // for filling in COMMCONFIG
#0137    virtual void  KickTx();
#0138  };
#0139
#0140  #define ASSERT_VALID_CPORT(p)
#0141    ASSERT(((CPort *) p)->m_signature == 'TROP')
#0142
#0143  #endif // VPORT_H
```

The Port Driver Functions

這一小節將對 22 個 port driver functions 做詳細的描述。我並沒有把除錯用的碼列出來，爲的是讓程式更簡明。我也放棄列出那些太過瑣屑而且不是主要說明目標的東西。如果你要完整程式碼，請從書附光碟片中取得。

ClearError *ClearError* 傳回目前的通訊錯誤，並將所有的錯誤指示器清爲 0：

```
BOOL CPort::ClearError(_COMSTAT* pComstat, int* pError)
   {                        // CPort::ClearError
   if (pComstat)
      GetQueueStatus(pComstat);
   *pError = (int) m_pd.dwCommError;
   m_pd.dwCommError = 0;
   m_pd.dwLastError = 0;
   return TRUE;
   }                        // CPort::ClearError
```

一個應用程式之所以會產生 *ClearError* calls，可能是因爲呼叫 *GetCommError*（Win16）或 *ClearCommError*（Win32）。如果有任何錯誤指示器被設立起來，*Read* 函式通常會立刻中斷執行。應用程式只要使用 *IGNOREERRORONREADS* escape 函式，就可以忽略這項錯誤檢查功能。

pComstat 可有可無。如果它不是 NULL，這個 driver 應該像 *GetQueueStatus* 那樣地填寫它。我將在討論 *GetQueueStatus* 時一併討論 *_COMSTAT*。請注意，設立 *dwLastError* 爲 0 意思是 *ClearError* 本身成功。所有的 port driver 函式如果成功，都必須清除 *dwLastError*。我不會在後面一再重複這一點。

一個衍生類別一般而言沒有必要改寫 *ClearError*，因爲所有的機能都與 device 無關。

Close　*Close* 用來關閉一個 port。應用程式觸發一個 *Close* call 的方式是呼叫 *CloseComm*（如果是一個 Win16 程式）或 *CloseHandle*（如果是一個 Win32 程式）。我的例子中這樣實作 *Close*：

```
BOOL CPort::Close()
    {                           // CPort::Close
    m_pd.dwLastError = 0;       // assume no problems
    if (!m_open)
        return TRUE;            // not open in the first place

    if (!trmcom())
        return FALSE;           // mini-driver couldn't close
    Release();                  // release from contention manager
    m_open = FALSE;

    if (m_MiscFlags & MF_RXQINTERNAL)
        _HeapFree((PVOID) m_pd.QInAddr, 0);

    if (m_MiscFlags & MF_TXQINTERNAL)
        _HeapFree((PVOID) m_pd.QOutAddr, 0);

    m_MiscFlags |= MF_CLRTIMER;
    ManageTimer();
    return TRUE;
    }                           // CPort::Close
```

這個基礎類別的函式呼叫一個名爲 *trmcom* 的虛擬函式，以便在關閉 port 時執行與 device 相關的動作。我的 *CSerialPort::trmcom* 函式將等待 output queue 乾涸掉，然後命令 UART 阻止更進一步的中斷，並將相關的「IRQ 虛擬化」取消掉。

關閉一個 port 的過程之中，與 device 無關的部份已經顯示於上。由於我們是從 contention VxD 手中獲得 port，所以有必要現在釋放它。我已於稍早討論競爭管理（contention management）時顯示過 *CPort::Release* 函式碼。我們的 *CPort::Setup* 函式可能已經配置了接收緩衝區和（或）傳輸緩衝區，所以 *CPort::Close* 必須釋放它們。最後，我們可能已經產生了一個 timer，用以處理一個或多個 timeout events，這些 events 與我們的 driver 所管理的眾多 ports 中的一個有關聯。*ManageTimer* 函式會取消 timeout event（如果沒有其他的 port 依附於它的話）。由於基礎類別的 *Close* 函式在適當時機呼叫虛擬函式 *trmcom*，所以衍生類別應該沒有必要改寫 *Close* 函式。

EnableNotification *EnableNotification* 允許 VCOMM 針對列於表 **14-6** 中的任何 events 註冊一個 notification 函式。下面是我爲 *CPort* class 設計的實作碼：

```
BOOL CPort::EnableNotification(PCOMMNOTIFYPROC pCallback, DWORD refdata)
   {                        // CPort::EnableNotification
   ClrNfyFlag(CN_NOTIFY);
   m_EvNotify = pCallback;
   m_EvData = refdata;
   if (pCallback)
      {                     // wants notifications
      _asm pushfd
      _asm cli
      m_NfyFlags |= CN_NOTIFY;
      DWORD pending = m_eventmask & *m_pEvent;
      _asm popfd
      if (pending)
         CallNotifyProc(CN_EVENT);
      }                     // wants notifications
   m_pd.dwLastError = 0;
   return TRUE;
   }                        // CPort::EnableNotification
```

m_eventmask 成員變數內含一個 bit mask，用以表示「client 程式希望收到 notifications」的一些 events。*m_pEvent* 變數指向一個由 client 提供的 DWORD，port driver 會在其中記錄未審理的 events。*SetEventMask* 會將這些變數都設定好。

一個 Win16 程式可以呼叫 *EnableCommNotification*，告訴 USER.EXE 從此送來 *WM_COMMNOTIFY* 訊息。USER.EXE 呼叫 COMM.DRV 的一個進入點，將一個

VCOMM_PM_API_EnableNotify request 轉播給 VCOMM。VCOMM 在代之以一個適當的 ring0 callback 函式（它使用巢狀執行以喚起應用程式）之後，呼叫 port driver。Win32 程式不能夠建立一個 event notification 函式 -- 雖然它們可以使用 *WaitCommEvent* 來等待 event 的發生。

Port driver 中的函式如果發現 events，應該檢查看看是否有一個 notification callback 已經註冊好了。如果是，就呼叫之。例如，*CSerialPort* 中的硬體中斷處理常式內含下面的 event 辨識邏輯：

```
BOOL CSerialPort::HwIntProc()
   {                          // CSerialPort::HwIntProc
   DWORD oldevents = *m_pEvent;    // current event word

   do  {                      // handle pending interrupts
       [interrupt handling code that may set *m_pEvent bits]
       }                      // handle pending interrupts
   while ( [expression indicating pending interrupts]);

   VPICD_Phys_EOI(m_irqhandle);

   // Notify client of any new events

   *m_pEvent &= m_eventmask;       // clear uninteresting events
   if ((m_NfyFlags & CN_NOTIFY) && (*m_pEvent & ~oldevents))
       CallNotifyProc(CN_EVENT);
   return TRUE;                    // i.e., we've handled it
   }                               // CSerialPort::HwIntProc
```

CPort::CallNotifyProc 用來呼叫 client notification 函式：

```
void CPort::CallNotifyProc(int code)
   {                       // CPort::CallNotifyProc
   _asm mov ebx, this
   _asm mov al, byte ptr code
   _asm or [ebx]CPort.m_NfyFlags, al
   DWORD events = *m_pEvent;
   switch (code)
       {                   // select notify proc to call

   case CN_EVENT:
       (*m_EvNotify)(&m_pd, m_EvData, code, events);
       break;
```

```
case CN_RECEIVE:
    (*m_RxNotify)(&m_pd, m_RxData, code, events);
    break;

case CN_TRANSMIT:
    (*m_TxNotify)(&m_pd, m_TxData, code, events);
    break;
    }                   // select notify proc to call
}                       // CPort::CallNotifyProc
```

由於當硬體中斷正被處理的時候，event notification 還是可能發生，所以 notification 函式必須能夠被重進入（reentrant）並且必須是 page-locked，而且它只能夠使用 asynchronous VxD services。請注意，我小心地避免 *EnableNotification* 和 *CallNotifyProc* 函式與一個中斷處理常式（它可能也要改變 *m_NfyFlags*）之間陷入可能的競速狀態（races conditions）。*ClrNfyFlag* 是一個巨集，使用 inline assembler 在一個不可分割的動作（atomic operation）中改變 *m_NfyFlags*。如果你不知道 interrupt flag 原來的狀態是什麼，*EnableNoification* 也示範了清除並於稍後恢復 interrupt flag 的正確作法：你應該使用 PUSHFD 先將目前的 flags 儲存於堆疊之中，然後執行 CLI 以清除中斷，然後再執行 POPFD 恢復 interrupt flag。

同時也請注意，client 可以提供一個 NULL notification 函式位址，表示要將 event notification 除能（disable）。

爲什麼我要在最初三行使用 EBX 暫存器呢？有一個特殊的理由。雖然 notification callback 函式（*m_EvNotify*, *m_RxNotify*, 和 *m_TxNotify*）被假設爲可由 C 程式呼叫（C-callable），但是它們有可能未能恢復 EBX 暫存器原值。像上例那樣地使用 EBX，會使編譯器在 *CPort::CallNotifyProc* 函式的 prolog 和 epilog 碼之中儲存並恢復 EBX，因而避免對呼叫端造成問題。

由於 *EnableNotification* 獨立於任何 device 之外，所以一個衍生類別應該沒有什麼理由去改寫它。然而衍生類別仍然需要花心思於如何維護 event word 和觸發 notification calls。

EscapeFunction　*EscapeFunction* 針對 port 執行一個擴充的控制函式。表 **14-9** 列出每一個可能的 escape 函式，以及一個代碼，表示該函式是與 device 有關（device-dependent）或與 device 無關（device-independent）。應用程式呼叫 *EscapeCommFunction*（不論是 Win16 或 Win32）就可以觸發 *EscpeFunction* 函式。

Escape 函式	Device-Dependent(D) 或 Device-Independent(I)	說明
SETXOFF	D (COM)	作用如同一個 X-Off (DC3) 字元已被收到（來自 host）
SETXON	D (COM)	作用如同一個 X-On (DC1) 字元已被收到（來自 host）
SETRTS	D (COM)	設定（sets）RTS line
CLRRTS	D (COM)	清除（clears）RTS line
SETDTR	D (COM)	設定（sets）DTR line
CLRDTR	D (COM)	清除（clears）DTR line
RESETDEV	D (LPT)	重置（Resets）device
GETCOMBASEIRQ	D	決定 I/O base address 和 IRQ
SETBREAK	D (COM)	在 RS-232 line 上維護（asserts）一個 break signal
CLEARBREAK	D (COM)	去除（removes）break signal
CLRTIMERLOGIC	I	Disables timeout logic
GETDEVICEID	D (LPT)	取得 device identifier
SETECPADDRESS	D (LPT)	設定 ECP channel address
ENABLETIMERLOGIC	I	Enables timeout logic
IGNOREERRORONREAD	I	允許 read 動作，甚至即使有 errors 尚未被處理
SETUPDATETIMEADDR	I	儲存一個新位址，用以記錄最新收到的 time value
PEEKCHAR	I	在 input queue 中檢驗下一個字元（如果有的話）

表 14-9　Device escape functions

在我的範例程式中，我以下列方式實作出與 device 無關的 escape 函式：

```
BOOL CPort::EscapeFunction(DWORD lFunc, DWORD InData, PVOID pOutData)
   {                          // CPort::EscapeFunction
   m_pd.dwLastError = 0;

   switch (lFunc)
      {                       // process escape function

   case PEEKCHAR:             // lFunc == 200
      if (!m_pd.QInCount)
         return FALSE;        // no pending input character
      *(PBYTE) pOutData = ((char *) m_pd.QInAddr)[m_pd.QInGet];
      break;

   case ENABLETIMERLOGIC:     // lFunc == 21
      m_MiscFlags &= ~MF_CLRTIMER;
      break;

   case IGNOREERRORONREADS:   // lFunc == 20
      m_MiscFlags |= MF_IGNORECOMMERROR;
      break;

   case CLRTIMERLOGIC:        // lFunc == 16
      m_MiscFlags |= MF_CLRTIMER;
      break;
   case SETUPDATETIMEADDR:    // lFunc == 19
      m_pRxTime = (PDWORD) InData;
      break;

   default:
      if (cextfcn(lFunc, InData, pOutData))
         break;            // mini-driver handled it
      m_pd.dwLastError = (DWORD) IE_EXTINVALID;
      return FALSE;          // unknown escape
      }                     // process escape function
   return TRUE;
   }                         // CPort::EscapeFunction
```

其中的 *cextfcn* 虛擬函式用來處理與 device 相關的 escapes。由於基礎類別在適當時間呼叫 *cextfcn*，所以衍生類別並沒有什麼特別理由需要改寫 *EscapeFunction*。

下面是與 device 無關之 escape 函式的意義：

- **PEEKCHAR**　這個 escape 函式允許呼叫者檢驗 input queue 中的下一個字元（如果有的話）。如果有任何字元正在 queue 之中，這個 port 的 *PortData* 結構中的 *QInCount* 欄位將爲非零值，於是下一個字元可以在 offset 爲 *QInGet* 的位置找到。offset 是從 input buffer 的起頭（也就是 *QInAddr* 所指示）開始起量。

- **ENABLETIMERLOGIC**　這個 escape 函式會令一個 timer 起作用（enables），使它成爲 client 程式的一個「擋球網」。timer 的運作方式如下：每 100 毫秒（ms），driver 就醒來並檢驗它所管理的所有 ports。如果有一個 port 已經因爲呼叫 *SetReadCallback* 而要求 input queue notifications，並且當時的確有資料滯留在 port's input queue 之中，但最後收到的資料是在前一個 timeout 之前到達，那麼 port driver 就會呼叫 client 的 input callback 函式 -- 甚至即使 queue 尚未達到指定的 threshold。這個想法可以讓應用程式一直保持在「忙碌讀取 input data」的狀態，甚至當 host 並非快速傳送資料，也是如此。

- **IGNOREERRORONREADS**　這個 escape 函式告訴 port driver，即使有尚未被清除的錯誤狀態存在，也要執行 read 動作。正常情況下，如果有任何錯誤被偵測到並記錄於 *dwCommError* 之中，client 一呼叫 *Read* 便會立刻失敗。這時候 client 必須呼叫 *ClearError* 以清除 error flags，才能讓 read 動作再次順利執行。

- **CLRTIMERLOGIC**　這個 escape 函式告訴 port driver，請它停止每 100 毫秒（ms）送給 client 程式一個 input notifications。

- **SETUPDATETIMEADDR**　這個 escape 函式指定一個新位址，用以儲存一個 DWORD。Port driver 中斷處理常式會將每一個 input byte 到達時間記錄在這個 DWORD 中。此一時間值（time stamp）和 *Get_Last_Updated_System_Time* 所獲得的一樣，其不精確度可能高達 50 毫秒（ms），但其價值是不必花費任何時間在計算上面。

GetCommConfig　*GetCommConfig* 會填寫一個 *COMMCONFIG* 結構，用以描述 port 的目前組態。Win32 API 有一個同名的函式。Win16 程式並不使用這個 port driver 函式。我的實作方式如下：

```
BOOL CPort::GetCommConfig(PCOMMCONFIG lpCC, PDWORD lpSize)
   {                          // CPort::GetCommConfig
   DWORD size = *lpSize;
   *lpSize = sizeof(COMMCONFIG);

   if (size < sizeof(COMMCONFIG) || !lpCC)
      return TRUE;

   lpCC->dwProviderOffset = 0;
   lpCC->dwProviderSize = 0;
   lpCC->dwSize = sizeof(COMMCONFIG);
   lpCC->wVersion = 0x0100;
   lpCC->dwProviderSubType = GetProviderSubType();
   VCOMM_Map_Ring0DCB_To_Win32(&m_dcb, &lpCC->dcb);

   return TRUE;
   }                          // CPort::GetCommConfig
```

其中 *lpSize* 參數指向一個 DWORD，這個 DWORD 最初內含 *COMMCONFIG* 結構（由 *lpCC* 指出）的大小。*GetCommConfig* 會更新 DWORD 內容，放置真正結構體的大小。如果原先的大小太小，*GetCommConig* 會傳回 TRUE 並且不做任何事情（除了設定大小之外）。然而如果大小充裕，*GetCommConfig* 就會填寫該結構。

COMMCONFIG 結構（請看**圖 14-8**）在 Win32 SDK online help 的 "Win32 Programmer's Reference" 這一節中有所說明。如果將 *dwProviderOffset* 和 *dwProviderSize* 設初值爲 0，表示這個結構之中的 *wcProviderData* 欄位內並沒有由 provider 所指定的資料。*dwProviderSubType* 值與 device 相依，所以我提供了一個名爲 *GetProviderSubType* 的虛擬函式，用以決定其值。*CSerialPort::GetProviderSubType* 會傳回 *PST_RS232*，表示 *CSerialPort* 實作出一個 serial port driver。

```
typedef struct _COMM_CONFIG {
  DWORD    dwSize;                 // 00 size of the structure
  WORD     wVersion;               // 04 version of the structure (0x0100 = 1.00)
  WORD     wAlignDCB;              // 06 padding to DWORD boundary
  Win32DCB  dcb;                   // 08 Win32-format device control block
  DWORD    dwProviderSubType;      // 24 provider subtype (PST_xxx)
  DWORD    dwProviderOffset;       // 28 offset of provider data from
                                   //    start of structure
  DWORD    dwProviderSize;         // 2C length of provider data that
```

```
                                    //    follows structure
} COMMCONFIG, *PCOMMCONFIG;         // [30]
```

圖 14-8　COMMCONFIG 結構

最有趣的組態資訊（configuration information）是 COMMCONFIG 結構中的 *dcb* 欄位。這個欄位代表一個 Win32 格式的 device control block（DCB）。由於 ring0 DCB 的格式（VCOMM 和 port drivers 都靠它來運作）與 Win32 格式不同，VCOMM 提供了其間的轉換服務。本例之中我們希望把 *CPort* object 中的 ring0 DCB 轉換爲 *COMMCONFIG* 結構中的 Win32 DCB，所以我們呼叫 *VCOM_Map_Ring0DCB_To_ Win32*。

由於 provider subtype 是唯一一個因不同之 devices 而有變化的資料，所以沒有什麼特殊理由需要衍生類別改寫（overrided）*GetCommConfig*。提供一個適當的 *GetProviderSubType* 函式應該就足夠了。

GetCommState　*GetCommState* 傳回一個 communications port 的目前狀態，它會填寫一個由呼叫端傳來的 device control block。Win16 和 Win32 程式使用它們各自的 *GetCommState* API 函式來觸及這個 port driver 函式。以下是我的設計：

```
BOOL CPort::GetCommState(_DCB* pDCB)
   {                        // CPort::GetCommState
   *pDCB = m_dcb;
   m_pd.dwLastError = 0;
   return TRUE;
   }                        // CPort::GetCommState
```

很簡單，不是嗎？同樣地似乎沒有理由需要去改寫（override）*GetCommState* 函式。

GetError　每個人都會猜想 *GetError* 到底做些什麼事情。SERIAL.VXD 程式中的這個函式什麼也沒做，LPT.VXD 程式中的這個函式則檢驗 *PortData* 的 *dwCommError* 欄位，將「與印表機有關的 error flag」轉換爲正規的 WINERROR.H（也就是 Win32 *GetLastError*）碼。VCOMM.DOC 並未說明這個 port 函式，但它聲稱 *_VCOMM_GetLastError* service 會傳回 IE-series 中的一個 communication port errors。事

實上，*_VCOMM_GetLastError* 傳回的是 port driver 遺留在 *PortData* 結構中的 *dwLastError* 欄位內容。那理所當然是一個 IE-series error code。實在是有點混亂！我是這樣設計這個函式的：

```
BOOL CPort::GetError(int* pError)
   {                        // CPort::GetError
   *pError = m_pd.dwLastError;
   return TRUE;
   }                        // CPort::GetError
```

衍生類別沒有什麼理由需要改寫（override）*GetError* 函式。

GetEventMask *GetEventMask* 執行兩個功能。第一，它將目前的 event word（其中記錄著 client 感興趣的 events（藉由呼叫 *SetEventMask*））拷貝到一個由呼叫者提供的地點。第二，它把某些 events 從 event word 中遮罩掉（mask off）。16 位元程式使用 *GetCommEventMask* 來觸發此函式，Win32 程式則根本用不到它。下面是我的設計：

```
BOOL CPort::GetEventMask(DWORD mask, PDWORD pEvents)
   {                           // CPort::GetEventMask
   _asm pushfd
   _asm cli
   *pEvents = *m_pEvent;       // return all current events
   *m_pEvent &= ~mask;         // clear selected events
   _asm popfd
   m_pd.dwLastError = 0;
   return TRUE;
   }                           // CPort::GetEventMask
```

技術文件中的另一個混亂之處 VCOMM 文件中說，*GetEventMask* 函式的 *pEvents* 參數可以是 NULL。事實上它不能夠！好吧，我承認，這個函式的 LPT.VXD 版會檢驗該指標，但 SERIAL.VXD 版不會。這份文件似乎暗指一個 non-NULL *pEvents* 參數將成爲 event word 的新位址。不，不是這樣！唯一能夠改變 event word 位址的辦法是呼叫 *SetEventMask*。猶有進者，Microsoft 對於此一函式以及 *SetEventMask* 函式的說明文字中，使用了互不一致的 "mask" 意義。爲了精準，我以 "event word" 這個字眼來描述「port

driver 將用以記錄 enabled events」的地方，我以 "mask" 這個字眼來描述一個 bits 陣列，用以表示哪一個 events 目前被 enabled。當 Microsoft 命名 *GetEventMask*（以及對應的 Win16 API 函式），他們卻以 event mask 表示 event word。Win32 API 之中有兩個函式：*SetCommMask* 和 *GetCommMask*，它們操作的是 "mask"，而不是 event word。我要說的是，只是退回 mask，卻不知道哪一個 event 已經發生，並沒有什麼用途，不過至少 API 是一致的。

由於 *GetEventMask* 機能與 device 無關，所以沒有什麼理由讓一個衍生類別改寫（override）這個函式。

GetModemStatus　*GetModemStatus* 傳回目前的 modem 狀態。此函式乃爲 Win32 *GetCommModemStatus* API 函式服務。一個衍生類別可以改寫（override）此一函式以便從硬體取得目前狀態，例如：

```
BOOL CSerialPort::GetModemStatus(PDWORD pModemStatus)
   {                        // CSerialPort::GetModemStatus
   *pModemStatus = in(m_iobase + MSR) & MS_Modem_Status;
   return TRUE;
   }                        // CSerialPort::GetModemStatus
```

GetProperties　*GetProperties* 會將 port 的相關資訊填入一個 *_COMMPROP* 結構（圖 **14-9**）。這個函式用以爲 Win32 API *GetCommProperties* 服務。我的 driver 係以下列方式完成 properties 結構之中與 device 無關的部份：

```
BOOL CPort::GetProperties(_COMMPROP* pCommProp)
   {                        // CPort::GetProperties
   memset(pCommProp, 0, sizeof(_COMMPROP));
   pCommProp->wPacketLength = sizeof(_COMMPROP);
   pCommProp->wPacketVersion = 2;
   pCommProp->dwServiceMask = SP_SERIALCOMM;
   pCommProp->dwCurrentRxQueue = m_pd.QInSize;
   pCommProp->dwCurrentTxQueue = m_pd.QOutSize;
```

```
    m_pd.dwLastError = 0;
    return TRUE;
    }                       // CPort::GetProperties
```

```
typedef struct _COMMPROP {       // cmmp
  WORD   wPacketLength;          // 00 length of this structure
  WORD   wPacketVersion;         // 02 version of this structure
  DWORD  dwServiceMask;          // 04 bit mask indicating services provided
  DWORD  dwReserved1;            // 08 reserved
  DWORD  dwMaxTxQueue;           // 0C maximum transmit queue size
  DWORD  dwMaxRxQueue;           // 10 maximum receive queue size
  DWORD  dwMaxBaud;              // 14 maximum baud support
  DWORD  dwProvSubType;          // 18 specific COMM provider type
  DWORD  dwProvCapabilities;     // 1C flow control capabilities
  DWORD  dwSettableParams;       // 20 bit mask indicating parameters that can be set
  DWORD  dwSettableBaud;         // 24 bit mask indicating baud rates that can be set
  WORD   wSettableData;          // 28 bit mask indicating number of data bits that can be set
  WORD   wSettableStopParity;    // 2A bit mask indicating allowed stop bits and parity checking
  DWORD  dwCurrentTxQueue;       // 2C current size of transmit queue
  DWORD  dwCurrentRxQueue;       // 30 current size of receive queue
  DWORD  dwProvSpec1;            // 34 used if clients have intimate knowledge of format
  DWORD  dwProvSpec2;            // 38 used if clients have intimate knowledge of format
  WCHAR  wcProvChar[1];          // 3C used if clients have intimate knowledge of format
  WORD   filler;                 // 3E to make it multiple of 4
} COMMPROP;                      // [40]
```

圖 14-9 _COMMPROP 結構

SP_SERIALCOMM 是唯一一個爲 *dwServiceMask* 而定義的常數。呼叫 *GetProperties* 企圖取得一個 parallel port 資訊，並不合理。LPT.VXD 勇敢地嘗試以各種方式來處理這個呼叫，但是它只設定 *dwCurrentxxQueue* 值。

衍生類別應該改寫（override）這個函式，呼叫基礎類別函式，然後填寫這個結構中與 device 相依的部份，例如：

```
BOOL CSerialPort::GetProperties(_COMMPROP* pCommProp)
    {                       // CSerialPort::GetProperties
    if (!CPort::GetProperties(pCommProp))
        return FALSE;

    pCommProp->dwMaxBaud = BAUD_USER;
    pCommProp->dwProvSubType = PST_RS232;
```

```
    ...
    return TRUE;
    }                       // CSerialPort::GetProperties
```

GetQueueStatus　這個函式的命名並不適當，它傳回的是一個 *_COMSTAT* 結構（圖 **14-10**），其中有 status flag，以及 queue 中的目前資料數量。應用程式通常是呼叫 *GetComError*（Win16）或 *ClearCommError*（Win32），並提供一個 non-NULL 的 _COMSTAT 指標，以此方式間接性地觸發這個函式。這個函式之中與 device 無關的部份如下：

```
BOOL CPort::GetQueueStatus(_COMSTAT* pComstat)
    {                       // CPort::GetQueueStatus
    m_pd.dwLastError = 0;
    pComstat->cbInque = m_pd.QInCount;
    pComstat->cbOutque = m_pd.QOutCount;
    return TRUE;
    }                       // CPort::GetQueueStatus
```

衍生類別應該改寫（override）這個函式，填寫 *BitMask* 欄位，並呼叫基礎類別的函式。

```
typedef struct _COMSTAT {
  DWORD BitMask;     // 00 flags
  DWORD cbInQue;     // 04 number of bytes in input queue
  DWORD cbOutQue;    // 08 number of bytes in output queue
} _COMSTAT;          // 0C
```

圖 14-10　_COMSTAT 結構

Purge　*Purge* 會拋棄 input buffer 或 output buffer 中的目前內容。*Purge* 爲 Win32 API *PurgeComm* 提供服務。這個函式大多與 device 無關：

```
#0001  BOOL CPort::Purge(DWORD qType)
#0002      {                        // CPort::Purge
#0003      switch (qType)
#0004          {                    // purge requested queue
#0005
#0006      case 0:                   // Tx queue
#0007          {                    // flush Tx queue
#0008          DWORD count = m_pd.QOutCount;
```

```
#0009         _asm pushfd
#0010         _asm cli
#0011         m_pd.QOutCount = 0;
#0012         m_pd.QOutGet = 0;
#0013         m_pd.QOutPut = 0;
#0014         _asm popfd
#0015
#0016         if (count && m_TxTrigger)
#0017            CallNotifyProc(CN_TRANSMIT);
#0018
#0019         Flush(qType);
#0020         break;
#0021         }                    // flush Tx queue
#0022
#0023     case 1:                   // Rx queue
#0024         _asm pushfd
#0025         _asm cli
#0026         m_pd.QInCount = 0;
#0027         m_pd.QInGet = 0;
#0028         m_pd.QInPut = 0;
#0029         _asm popfd
#0030
#0031         Flush(qType);
#0032         ClrNfyFlag(CN_RECEIVE);
#0033
#0034         break;
#0035
#0036     default:
#0037         break;
#0038         }                    // purge requested queue
#0039     m_pd.dwLastError = 0;
#0040     return TRUE;
#0041     }                        // CPort::Purge
```

衍生類別可以改寫（override）*Flush* 成員函式以提供額外的處理。例如我的 *CSerialPort::Flush* 函式就會檢查看看是否先前填寫 input queue 的動作會引起 driver 使用一個 flow control 機制（例如送出一個 X-off 字元）來中止傳輸。如果是，對 read queue 的清理（purge）動作意味著「host 應該被通知重新傳輸」。

Read *Read* 會從 input queue 中取得資料。應用程式呼叫 *ReadComm*（Win16）或 *ReadFile*（Win32）以觸發此一 port driver 函式。一個 read 動作，其大部份工作其實是管理 ring buffer。我的 driver 以這樣的方式來完成 *Read* 函式：

```
#0001  BOOL CPort::Read(PCHAR buf, DWORD cbRequest, PDWORD pRxCount)
#0002     {                        // CPort::Read
#0003     if (!(m_MiscFlags & MF_IGNORECOMMERROR) && m_pd.dwCommError)
#0004        {                     // pending errors
#0005        m_pd.dwLastError = m_pd.dwCommError;
#0006        return FALSE;
#0007        }                     // pending errors
#0008
#0009     m_pd.dwLastError = 0;
#0010     DWORD numread = m_pd.QInCount;
#0011     if (!numread)
#0012        {                     // quick out if queue empty
#0013        *pRxCount = 0;
#0014        return TRUE;
#0015        }                     // quick out if queue empty
#0016
#0017     if (numread > cbRequest)
#0018        numread = cbRequest;
#0019     DWORD get = m_pd.QInGet;
#0020     DWORD ncopy = m_pd.QInSize - get;
#0021     if (ncopy > numread)
#0022        ncopy = numread;
#0023     memcpy(buf, (PCHAR) m_pd.QInAddr + get, ncopy);
#0024
#0025     if (ncopy == numread)
#0026        get += ncopy;
#0027     else
#0028        {                     // wraparound to start
#0029        buf += ncopy;
#0030        ncopy = numread - ncopy;
#0031        memcpy(buf, (PCHAR) m_pd.QInAddr, ncopy);
#0032        get = ncopy;
#0033        }                     // wraparound to start
#0034     m_pd.QInGet = get;
#0035
#0036     _asm pushfd
#0037     _asm cli
#0038     m_pd.QInCount -= numread;
#0039     m_NfyFlags &= ~CN_RECEIVE;
#0040     _asm popfd
#0041
#0042     *pRxCount = numread;
#0043     return TRUE;
#0044     }                        // CPort::Read
```

如果曾經有人舉發一個 *IGNOREERRORONREADS* escape 函式，*MF_IGNORECOMMERROR* flag 會被設立起來。這表示即使尚有 communication error 還未被 *ClearError* 清除，read 動作還是應該被處理。我從 SERIAL.VXD 中拷貝了一段碼，用來將 *dwLastError* 設定等於 *dwCommError*。*dwCommError* 中的 CE-series communication error bits 並不等於 IE-series error codes（隸屬於 *dwLastError*），所以這段碼當然是錯誤的。我曾經在稍早的 *GetError* 相關討論中對此做了一些陳述，述及一些令人迷惑的 error codes，它們被誤以爲「被 port drivers 儲存在 *dwLastError* 之中」。既然有了前車之鑑，我想你面對這裡又一次的混淆，不會太過驚訝。

Read 函式中的緩衝區管理以及拷貝邏輯十分明顯易懂。請回頭參考圖 **14-7**，該圖告訴我們這些用於緩衝管理的各個參數是如何彼此產生關係。「更新緩衝區計數及 notify callback flag」的動作必須在不會被中斷處理常式干擾的情況下進行，所以我以「清除 interrupt flag」的方式保護那些步驟。

一個衍生類別應該改寫（override）Read 函式，在其中呼叫基礎類別函式，然後檢查「從 input buffer 中移取資料」是否有可能通知 host 重新傳輸。

SetCommConfig *SetCommConfig* 是 *GetCommConfig* 的相反動作。它會根據一個 Win32-format device control block 來設定 port 狀態：

```
BOOL CPort::SetCommConfig(PCOMMCONFIG lpCC, DWORD dwSize)
   {                        // CPort::SetCommConfig
   if (dwSize < sizeof(COMMCONFIG))
      {                     // too small
      m_pd.dwLastError = (DWORD) IE_INVALIDPARAM;
      return FALSE;
      }                     // too small
   _DCB r0dcb;
   VCOMM_Map_Win32DCB_To_Ring0(&lpCC->dcb, &r0dcb);
   return SetCommState(&r0dcb, 0xFFFFFFFF);
   }                        // CPort::SetCommConfig
```

這個函式將爲 Win32 API 函式 *SetCommConfig* 服務（Win16 並沒有對應函式）。衍生類別沒有理由改寫(override)此函式，因爲基礎類別的 *SetCommState* 已經內含「與 device 相依」之功能的處理邏輯。

SetCommState　*SetCommState* 是設定通訊參數的基本 API 函式。這個函式使用一個 device control block（*_DCB*）結構（圖 **14-11**）來描述通訊協定。Win16 和 Win32 程式呼叫 *SetCommState* API 函式以觸及此一 port driver 函式。

```
typedef struct _DCB {
  DWORD DCBLength;        // 00 sizeof (DCB)
  DWORD BaudRate ;        // 04 Baudrate at which running
  DWORD BitMask;          // 08 flag DWORD
  DWORD XonLim;           // 0C Transmit X-ON threshold
  DWORD XoffLim;          // 10 Transmit X-OFF threshold
  WORD wReserved;         // 14 reserved
  BYTE ByteSize;          // 16 Number of bits/byte, 4-8
  BYTE Parity;            // 17 0-4=None,Odd,Even,Mark,Space
  BYTE StopBits;          // 18 0,1,2 = 1, 1.5, 2
  char XonChar;           // 19 Tx and Rx X-ON character
  char XoffChar;          // 1A Tx and Rx X-OFF character
  char ErrorChar;         // 1B Parity error replacement char
  char EofChar;           // 1C End of Input character
  char EvtChar1;          // 1D special event character
  char EvtChar2;          // 1E Another special event character
  BYTE bReserved;         // 1F reserved
  DWORD RlsTimeout;       // 20 Timeout for RLSD to be set
  DWORD CtsTimeout;       // 24 Timeout for CTS to be set
  DWORD DsrTimeout;       // 28 Timeout for DSR to be set
  DWORD TxDelay;          // 2C Amount of time between chars
} _DCB;                   // [30]
```

圖 14-11　device control block（DCB）格式

SetCommState 的參數包括一個 *_DCB* 結構位址，以及一個 bit mask，用以表示哪一個欄位被改變。爲了使可共用的程式碼愈多愈好，我寫了下面這個基礎類別函式：

```
#0001  BOOL CPort::SetCommState(_DCB* pDCB, DWORD ActionMask)
#0002     {                        // CPort::SetCommState
#0003     if ((m_pd.LossByte & 1) && !StealPort())
#0004        {                     // port stolen
#0005        m_pd.dwLastError = (DWORD) IE_DEFAULT;
#0006        return FALSE;
#0007        }                     // port stolen
#0008     if (!CheckState(pDCB, ActionMask))
#0009        return FALSE;    // error in miniport-specific parts
#0010     BeginSetState();    // prepare to change state of port
```

```
#0011
#0012     DWORD ChangedMask = 0;    // assume nothing changed yet
#0013
#0014     #define ss(m) if (ActionMask & f##m) { \
#0015                      if (m_dcb.m != pDCB->m) ChangedMask |= f##m; \
#0016                      m_dcb.m = pDCB->m;}
#0017
#0018  ss(BaudRate)
#0019     ss(BitMask)
#0020     ss(XonLim)
#0021     ss(XoffLim)
#0022     ss(ByteSize)
#0023     ss(Parity)
#0024     ss(StopBits)
#0025     ss(XonChar)
#0026     ss(XoffChar)
#0027     ss(ErrorChar)
#0028     ss(EofChar)
#0029     ss(EvtChar1)
#0030     ss(EvtChar2)
#0031     ss(RlsTimeout)
#0032     ss(CtsTimeout)
#0033     ss(DsrTimeout)
#0034     ss(TxDelay)
#0035
#0036     EndSetState(ChangedMask);   // install new parameters
#0037     m_MiscFlags |= MF_STATESETONCE;
#0038     m_pd.dwLastError = 0;
#0039     return TRUE;
#0040     }                        // CPort::SetCommState
```

譯註：程式碼中的 ## 是所謂的 merging operator，可以從 Visual C++ Online Help 中查到其意義與用途，也可以參考「多型與虛擬 - 物件導向的精髓」（侯俊傑著/松崗）第三章最後一節。這是 C++ Standard 所規範的一個 operator，在 Bjarne Stroustrup 的 *The C++ Programming Language* 2/e p.609 或 3/e p.162 中都有提到。

其中的 *ss* 巨集主要是爲了節省打字時間而設計，它會爲 *_DCB* 結構中的每一個欄位產生下面這樣的碼：

```
if (ActionMask & fBaudRate)
   {
```

```
if (m_dcb.BaudRate != pDCB->BaudRate)
    ChangedMask |= fBaudRate;
m_dcb.BaudRate = pDCB->BaudRate;
}
```

於是，在所有的 *ss* 呼叫之後，port 的 *_DCB* 結構被更新了，而 *ChangedMask* 則指出哪一個欄位真的被改變了。

有三個與 device 相依的步驟用來輔助 *SetCommState*：*CheckState* 用來驗證 *_DCB* 結構中的參數的正確性和一致性，*BeginSetState* 用來調整 device 以認可對 *_DCB* 的改變，*EndSetState* 對 device 做 "reprogramming" 動作（一旦所有的改變都完成之後）。衍生類別應該改寫（override）所有這三個函式。

SetEventMask　*SetEventMask* 和 *EnableNotification* 一起合作，用來提供 client notification of communications events。應用程式呼叫 *SetCommEventMask*（Win16）或 *SetCommMask*（Win32）以觸發此一 port driver 函式。這個函式會設立一個 event mask，用以管理哪一個 event 將被記錄下來。它也會給你一個機會，讓你設定 event word（內含 masked event flags）的位址：

```
#0001  BOOL CPort::SetEventMask(DWORD mask, PDWORD pEvents)
#0002      {                        // CPort::SetEventMask
#0003      m_eventmask = mask;
#0004      if (pEvents)
#0005          m_pEvent = pEvents;
#0006      m_pd.dwLastError = 0;
#0007      return TRUE;
#0008      }                        // CPort::SetEventMask
```

呼叫 *SetEventMask* 並傳進一個 NULL *pEvents* 參數，並不是一種錯誤：這正是一個 client 如何能夠「只改變 mask，不影響 events 儲存場所」的辦法。此外，port driver 必須維護一個 PDWORD（本例中爲 *m_pEvent*），指向目前的 event word。Driver 應該設此指標之初值爲 *PortData* 結構中的 *dwDetectedEvents* 欄位位址。

我相信 *SetEventMask* 的這份實作碼正確地解釋了 application level API 函式的目的。然而如果嘗試爲一個 parallel port 設定 event mask，他會大吃一驚：LPT.VXD 中的

SetEventMask 實作碼會把參數 mask 與 event word 做 OR 動作，而不是把它儲存爲一個真正的 mask。LPT.VXD 也視 *dwDetectedEvents* 爲一個指標，指向 event word，而不把它視爲 event word 本身。

衍生類別沒有理由需要改寫（override）*SetEventMask*。

SetModemStatusShadow *SetModemStatusShadow* 用以設定一個 BYTE 的位址；每當 Modem Status 暫存器的內容有所變化，port driver 中斷處理常式就會把該值儲存在那個 BYTE 之中：

```
#0001  BOOL CPort::SetModemStatusShadow(PBYTE pShadow)
#0002     {                        // CPort::SetModemStatusShadow
#0003     m_pMsrShadow = pShadow;
#0004     m_pd.dwLastError = 0;
#0005     return TRUE;
#0006     }                        // CPort::SetModemStatusShadow
```

衍生類別沒有理由改寫（override）這個函式。應用程式沒有什麼直接方法可以觸及（喚起）此一常式。

SetReadCallback *SetReadCallback* 會爲 input queue 建立一個 threshold queue size （一個觸發點），並註冊一個 callback 函式；當 queue 成長超過了該 threshold，callback 函式就會被呼叫：

```
#0001  BOOL CPort::SetReadCallback(DWORD RxTrigger,
#0002     PCOMMNOTIFYPROC pCallback, DWORD refdata)
#0003     {                        // CPort::SetReadCallback
#0004     if (RxTrigger != 0xFFFFFFFF && RxTrigger > m_pd.QInSize)
#0005        RxTrigger = m_pd.QInSize;   // make it sensible
#0006     if (!pCallback)
#0007        RxTrigger = 0xFFFFFFFF; // reset if no callback function
#0008
#0009     _asm pushfd
#0010     _asm cli
#0011     m_RxNotify = pCallback;
#0012     m_RxData = refdata;
#0013     m_RxTrigger = RxTrigger;
```

```
#0014     _asm popfd
#0015
#0016     if (!m_pd.QInCount)
#0017        *m_pRxTime = 0;
#0018     ManageTimer();
#0019     return TRUE;
#0020     }                          // CPort::SetReadCallback
```

Threshold 數值的意義是：每當 input queue 成長到某個大小（或更多），port driver 就會呼叫指定的 callback 函式。呼叫 callback 函式會緩和觸發行動，直到 input queue 降到 threshold 值之下。此外，如果 timer logic 對 port 而言係處於作用狀態（enabled），那麼即使 input queue 從未到達 threshold 值，callback 也會每 100 毫秒（ms）發生一次。預設情況下 timer logic 處於作用狀態，但也可以由 *ENABLETIMERLOGIC* 和 *CLRTIMERLOGIC* 這兩個 escape 函式來控制。每 100 毫秒（ms）呼叫一次 client，可以在 host 無法快速取走資料時，讓資料在各週期之間移出 input buffer。

如果指定一個 trigger 數值爲 -1，或是一個 NULL callback 位址，就會將 callback 機制除能（disables）。預設的 input trigger 是 -1，意思是當有人打開 port 時，input callbacks 最初處於 disabled 狀態。

衍生類別沒有理由需要改寫（override）*SetReadCallback*。Win16 呼叫 *EnableCommNotification* 可以間接觸及此一函式。Win32 程式並不使用這個函式。

Setup　*Setup* 會爲 input queue 和 output queue 的大小和位置設定初值。Win32 程式可以呼叫 *SetupComm* 以控制 queue 的大小，Win16 程式則是以 *OpenComm* 中的一個可有可無的參數來指定 queue sizes。我的 *Setup* 函式如下：

```
#0001  BOOL CPort::Setup(PCHAR RxQueue, DWORD RxLength,
#0002     PCHAR TxQueue, DWORD TxLength)
#0003     {                          // CPort::Setup
#0004     m_pd.dwLastError = 0;      // no error
#0005     m_pd.QInCount = 0;
#0006     m_pd.QInGet = 0;
#0007     m_pd.QInPut = 0;
#0008     m_pd.QOutCount = 0;
```

```
#0009     m_pd.QOutGet = 0;
#0010     m_pd.QOutPut = 0;
#0011
#0012     if (!RxQueue)
#0013        {                    // need internal buffer
#0014        if (m_MiscFlags & MF_RXQINTERNAL)
#0015           {                 // reallocate existing buffer
#0016           RxQueue = (PCHAR) _HeapReAllocate((PVOID)
#0017              m_pd.QInAddr, RxLength, 0);
#0018           if (!RxQueue)
#0019              return FALSE;
#0020           }                 // reallocate existing buffer
#0021        else
#0022           {                 // allocate buffer the first time
#0023           RxQueue = (PCHAR) _HeapAllocate(RxLength, 0);
#0024           if (!RxQueue)
#0025              return FALSE;   // means no change made
#0026           m_MiscFlags |= MF_RXQINTERNAL;
#0027           }                 // allocate buffer the first time
#0028        }                    // need internal buffer
#0029     m_pd.QInAddr = (DWORD) RxQueue;
#0030     m_pd.QInSize = RxLength;
#0031
#0032     if (!TxQueue)
#0033        if (TxLength)
#0034           {                 // need internal buffer
#0035           if (m_MiscFlags & MF_TXQINTERNAL)
#0036              {              // reallocate existing buffer
#0037              TxQueue = (PCHAR) _HeapReAllocate((PVOID)
#0038                 m_pd.QOutAddr, TxLength, 0);
#0039              if (!TxQueue)
#0040                 return FALSE;
#0041              }              // reallocate existing buffer
#0042           else
#0043              {              // allocate buffer the first time
#0044              TxQueue = (PCHAR) _HeapAllocate(TxLength, 0);
#0045              if (!TxQueue)
#0046                 {          // can't allocate
#0047                 if (m_MiscFlags & MF_RXQINTERNAL)
#0048                    {       // release internal buffer
#0049                    _HeapFree((PVOID) m_pd.QInAddr, 0);
#0050                    m_MiscFlags &= ~MF_RXQINTERNAL;
#0051                    }       // release internal buffer
#0052                 return FALSE;
#0053                 }          // can't allocate
#0054              m_MiscFlags |= MF_TXQINTERNAL;
```

```
#0055               }            // allocate buffer the first time
#0056           }                // need internal buffer
#0057
#0058     m_pd.QOutAddr = (DWORD) TxQueue;
#0059     m_pd.QOutSize = TxLength;
#0060     if (TxQueue)
#0061         m_MiscFlags |= MF_TXQSET;
#0062
#0063     return TRUE;
#0064     }                        // CPort::Setup
```

這個函式看起來比它實際上複雜。如果 client 指定一個緩衝區位址，port driver 將使用那個緩衝區。如果 client 指定一個緩衝區大小和一個 NULL 緩衝區位址，driver 會以指定的大小配置自己的緩衝區。如果 client 既沒有指定緩衝區大小也沒有指定位址，*Setup* 不會改變 queue 的位址或大小。

基本上我是從 Microsoft 的 SERIAL.VXD 中拷貝出對等函式的邏輯。上述實作碼可能導至一個記憶體破洞（memory leak），因爲如果 Setup 在某次呼叫期間配置了一個內部緩衝區，而在下次被呼叫時收到一個由 client 擁有的緩衝區，那麼 Setup 並不會釋放原來的緩衝區。

衍生類別沒有理由改寫（override）*Setup*。

SetWriteCallback　*SetWriteCallback* 會爲 output queue 建立一個 threshold queue size（一個觸發點），並註冊一個 callback 函式；當 queue 低於該 threshold，callback 函式就會被呼叫：

```
#0001  BOOL CPort::SetWriteCallback(DWORD TxTrigger,
#0002     PCOMMNOTIFYPROC pCallback, DWORD refdata)
#0003     {                        // CPort::SetWriteCallback
#0004     if (TxTrigger == 0xFFFFFFFF)
#0005         TxTrigger = 0;         // -1 => no callback
#0006     if ((m_MiscFlags & MF_TXQSET)
#0007         && TxTrigger > m_pd.QOutSize)
#0008         TxTrigger = m_pd.QOutSize;
#0009     if (!pCallback)
#0010         TxTrigger = 0;
```

```
#0011
#0012     _asm pushfd
#0013     _asm cli
#0014     m_TxNotify = pCallback;
#0015     m_TxData = refdata;
#0016     m_TxTrigger = TxTrigger;
#0017     if (m_pd.QOutCount < TxTrigger)
#0018        m_NfyFlags |= CN_TRANSMIT;
#0019     _asm popfd
#0020
#0021     return TRUE;
#0022     }                        // CPort::SetWriteCallback
```

Threshold 數值的意義是：每當 output queue 低於特定的 data bytes，port driver 就會呼叫指定的 callback 函式。呼叫 callback 函式會緩和觸發行動，直到 output queue 上昇到 threshold 值。Callback 函式的目的是允許 client 程式能夠重新填寫 output 緩衝區，以便對 host 保持一個穩定的資料流。

我列出一段奇怪的邏輯（來自 Microsoft 的 SERIAL.VXD）。其中允許一個 *TxTrigger* 值爲 -1，意思是 "no trigger"。其實 0 才是代表正確的 "no trigger" 意義，因爲 queue 的大小不可能爲負。事實上，0 是指定給 *m_TxTrigger* 的初值，所以 transmit notifications 最初處於 disabled 狀態。

衍生類別沒有理由需要改寫（override）*SetWriteCallback*。Win16 呼叫 *EnableCommNotification* 可以間接觸及此一函式。Win32 程式並不使用這個函式。

TransmitChar *TransmitChar* 會把一個高優先權的輸出字元放到 output queues 之中。應用程式可以呼叫 *TransmitCommChar*（Win16 或 Win32）探及 *TransmitChar*。這個函式天生就是與 device 相依。嚴格地說，一次只能存在一個高優先權字元，如果有第二個到來，port driver 應該診斷出一個錯誤（也就是 *IE_TRANSMITCHARFAILED*）。然而 Microsoft 的 SERIAL.VXD 並不檢查這種情況，所以我在我的 driver 中也不那麼做：

```
BOOL CSerialPort::TransmitChar(CHAR ch)
   {               // CSerialPort::TransmitChar
   m_ImmedChar = ch;  // save char to transmit BEFORE setting flag
```

```
SetFlag(fTxImmed); // remember we have a character to transmit
KickTx();
return TRUE;
}               // CSerialPort::TransmitChar
```

其中 *KickTx* 是一個與 device 相依的函式，企圖將 pending output 重新開始。

Write　Write 會把資料傳輸到 host 端。應用程式呼叫 *WriteComm*（Win16）或 *WriteFile*（Win32）以觸及此函式。和 *Read* 函式一樣，*Write* 函式的大部份工作都放在 ring buffer 的管理上面：

```
#0001  BOOL CPort::Write(PCHAR buf, DWORD cbRequest, PDWORD pTxCount)
#0002     {                       // CPort::Write
#0003     DWORD nwritten;
#0004     m_pd.dwLastError = 0;
#0005
#0006     if (m_MiscFlags & MF_TXQSET)
#0007        {                    // using an output buffer
#0008        nwritten = m_pd.QOutSize - m_pd.QOutCount;
#0009        if (nwritten > cbRequest)
#0010           nwritten = cbRequest;
#0011
#0012        DWORD put = m_pd.QOutPut; // where we can start copying
#0013        DWORD ncopy = m_pd.QOutSize - put; // most we can copy
#0014        if (ncopy > nwritten)
#0015           ncopy = nwritten;
#0016        memcpy((PCHAR) m_pd.QOutAddr + put, buf, ncopy);
#0017
#0018        if (ncopy == nwritten)
#0019           put += ncopy;      // only need one copy
#0020        else
#0021           {                 // wraparound
#0022           buf += ncopy;
#0023           ncopy = nwritten - ncopy; // amount left to do
#0024           memcpy((PCHAR) m_pd.QOutAddr, buf, ncopy);
#0025           put = ncopy;
#0026           }                 // wraparound
#0027
#0028        m_pd.QOutPut = put;
#0029
#0030        _asm pushfd
#0031        _asm cli
#0032        if ((m_pd.QOutCount += nwritten) >= m_TxTrigger)
```

```
#0033            m_NfyFlags &= ~CN_TRANSMIT;
#0034        _asm popfd
#0035        }                    // using an output buffer
#0036    else
#0037        {                    // no output buffer set up
#0038        _asm pushfd
#0039        _asm cli
#0040        m_pd.QOutAddr = (DWORD) buf;
#0041        m_pd.QOutSize = cbRequest;
#0042        m_pd.QOutCount = cbRequest;
#0043        m_pd.QOutPut = 0;
#0044        m_pd.QOutGet = 0;
#0045        if (cbRequest >= m_TxTrigger)
#0046            m_NfyFlags &= ~CN_TRANSMIT;
#0047        _asm popfd
#0048
#0049        nwritten = cbRequest;   // pretend all of it written
#0050        }                    // no output buffer set up
#0051
#0052    KickTx();                 // try to restart output
#0053    if (nwritten < cbRequest)
#0054        m_pd.dwCommError |= CE_TXFULL; // buffer became full
#0055    *pTxCount = nwritten;
#0056    return TRUE;
#0057    }                        // CPort::Write
```

其中第一個 if 句子的 else 分支處理的是「client 沒有設定一個 output queue」的情況。它會欺騙 driver 的剩餘部份，讓它們將呼叫端的 output buffer 視爲一個遺失的 queue。

衍生類別應該改寫 Write，並在驗證 host 不曾妨礙「經由目前之流程控制機制而完成的傳輸」之後，呼叫基礎類別函式。

VCOMM 的其他 VxDs

一如我稍早所說，VCOMM 和其他的 VxDs 合作，處理通訊方面的工作。我們已經討論過了 serial port driver VxDs。剩下的一些和 VCOMM 合作的 VxDs 是 parallel port drivers，port virtualization drivers，以及 modem drivers。

Parallel Port Drivers

本章前面各節，藉由一個例子，描述了大量的實作細節。該實例幾乎可以取代官方提供的 SERIAL.VXD port driver（在標準的 serial ports 方面）。針對 parallel ports，VCOMM 也使用 ports drivers。標準的 LPT.VXD driver 的運作模式很像我們剛剛所討論過的 serial port driver，不過比較簡單，因爲 parallel port 程式設計比 serial port 程式設計容易。

我很驚訝 Microsoft 如何把一個 parallel ports driver 塞進一個「針對較爲複雜之 serial ports 而設計」的架構之中。例如，我就很驚訝 LPT.VXD 如何處理類似 _DCB 和 COMMCONFIG 的資料結構，它們嚴重傾向於 serial 通訊所用。

不幸的是，Microsoft 並未在 Windows 95 DDK 光碟片中提供 LPT.VXD 原始碼。我必須退而使用逆向工程技術，才能學到 LPT.VXD 的細節。然後我發現了我早該預想得到的答案：LPT.VXD 忽略那些對它而言不適當的結構和欄位。舉個例子，*GetCommState*, *SetCommState*, *GetCommConfig*, 和 *SetCommConfig* 在 LPT.VXD 中都沒有作用：如果你呼叫它們，它們會設定 last-error 爲 0，並傳回 TRUE。*GetQueueStatus* 會填寫 *_COMSTAT* 結構中的 queue count 欄位，但也只是將 *BitMask* 欄位設爲 0 而已。

LPT.VXD 也受益於一件事實：VCOMM 並不呼叫那些「在 *PortFunctions* 陣列中的對應條目（entry）爲 NULL」的 port driver 函式。LPT.VXD 並沒有 *SetModemStatusShadow* 和 *GetModemStatus* 函式，所以它在對應的 *PortFunctions* 條目中使用 NULL 指標。將 *SetModemStatusShadow* 指標設爲 NULL 並不會有危險，因爲 UNIMODEM 沒有責任要載入一個 parallel port driver。

Port Virtualization Drivers

Windows 95 內含兩個 port virtualization VxDs：VCD 用於 serial ports，VPD 用於 printer ports。虛擬化（Virtualization）在 Windows 95 環境中是一種「用以保持與傳統應用程式相容」的方法。在一個理想世界裡，所有應用程式都應該使用新的 VCOMM 介面，以處理 serial 和 parallel ports，不應該有任何碼直接處理 I/O ports 或直接 "hooking" 硬體

中斷。但畢竟我們尚未到達這樣的烏托邦境界，我們需要更瞭解 VCD 和 VPD 如何完成其工作的細節部份，以便能夠揣測系統的行爲。由於 Windows 95 DDK 光碟中附有那兩個 VxDs 的原始碼，我們很容易展開探索。

Printer Port Trapping

VCD 和 VPD 都使用 *Install_IO_Handler* 來 "trap" 它們對應的 I/O ports。這兩個 VxDs 的顯著差異在於它們對於資源搶奪的策略。一般而言 VPD 會全面性地針對 printer ports，將 "trapping" 除能（disabled）。一旦 trapping 被除能（disabled），任何 VM 中的 ring3 碼就可以自由地和 printer 交談。這種全面性的自由導至詭異而不可預期的後果。圖 14-12 顯示兩個不同的 MS-DOS boxes 同時對印表機輸出，所獲得的怪異結果。

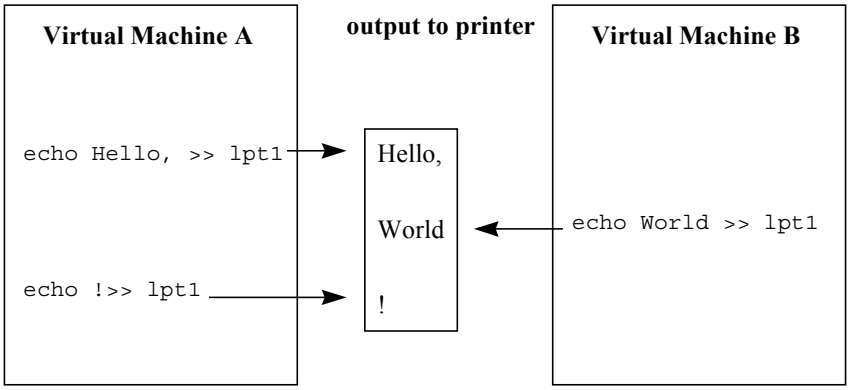


圖 14-12　兩個不同的虛擬機器同時列印

當有人使用 VCOMM 競爭機制（contention mechanism）來保留 port，也就是說，當某些 VxD 呼叫 VPD 的 contention function 並指定 *ACQUIRE_RESOURCE* 代碼，VPD 會呼叫 *Enable_Global_Trapping*，在所有的虛擬機器上 "trap" printer I/O ports。Windows 95 spooler 使用 VCOMM services 並且明白地保留 printer。所以當 spooler 正在列印一份文件，VPD 不會允許來自 MS-DOS 的干擾。如果一個 MS-DOS 程式企圖寫東西到該印表機上，VPD 會顯現一個「contention 對話盒」，讓使用者在 MS-DOS 程式和 Windows

95 spooler 之間做選擇。但是反方向並不成立：VPD 會允許 Windows 95 spooler 岔斷一個 MS-DOS 正列印中的程式。這樣的強制岔斷之所以發生，是因爲 VPD 不知道印表機有一個 ring3 使用者（因爲 "trapping" 一直處於除能（disabled）狀態，直到此刻），因而把印表機的擁有權指定給 system VM。而來自 MS-DOS 程式的下一個 I/O 動作會被 "trapped"，VPD 會模擬一個 "busy status"，導至發生一個 MS-DOS critical error。

VPD 的競爭管理（contention management）策略有兩個反常之事，值得注意。第一，當 Windows 95 spooler（官方提供的列印程式，在 system VM 執行）正在列印，而某些 Windows 程式企圖直接存取印表機時，你可能會遭遇一種病態情況，這種情況將導至錯誤，因爲「印表機擁有權」是一種「每個 VM 都可以有」的觀念。也就是說，VPD 會 捕捉（trap）來自非官方程式的 I/O，檢查它的確來自一個擁有印表機的 VM，才允許該 I/O 動作執行。第二，如果有許多 VM 使用 VCOMM contention 機制，VPD 便把擁有權指定給 System VM。因爲 contention 架構並不準備交給 *ACQUIRE_RESOURCE* 函式一個 VM handle。

Serial Port Trapping

和 VPD 相反，VCD 通常讓一個 serial device 的 I/O ports 被 "trapped"，除非有一個 VCOMM port driver 正在控制它。任何 VM 只要嘗試一個 I/O 動作，便會成爲 serial port 的擁有者。VCD 的 Windows 95 版也包含 Windows 3.1 COMBUFF driver 的機能，作法是對 physical I/O 做緩衝處理，並模擬來自 VMs 的 port I/O requests。這些處置確保作業系統能夠儘快反應 serial 中斷，但仍允許 virtual interrupt handlers 相信，它們正在服務真正的硬體（這是多工環境下對於排程的固有奇想）。如果另一個 VM 緊接著企圖處理硬體，VCD 不會讓它成功，並且不通告終端使用者。使用者可以藉由改變 SYSTEM.INI 檔中的 *COMxAutoAssign* 設定，來改變「資源搶奪」的行爲。

Serial Port Virtualization 與 VCOMM

你必須真正知道，VCD 並不是一個 VCOMM client。換句話說，它所虛擬化的任何 I/O 動作都將直接送向硬體，而不是經由 VCOMM 再送入一個 port driver VxD。如果你企圖提供一個應用於網路的 modem redirection，或如果你想 "trap" 對於標準的 ports 的 serial 動作（來自 MS-DOS 程式），你必須從 VCD 手上接管標準的 ports，而且你必須準備自己的 contention 機制。

VCD_Virtualize_Port 允許你扮演「將一個 port 虛擬化」的責任，如此一來你就可以將 I/O 動作經由 VCOMM 重新導向。你可以在 Windows 3.1 DDK 或是 Windows 95 DDK 光碟片中的 VCD 原始碼中找到此一服務的說明。但是與該說明相反的是，你必須在 CONFIGMG 的 *Device_Init* 之後、VCD 的 *Device_Init* 之前呼叫這個 service。這個奇特的時機是因為 VCD 建立它自己的內介資料結構，用以追蹤對其 contention 函式的 *ADD_RESOURCE* calls，然後在它自己的 *Device_Init* 過程中丟掉最後的資料結構。*VCD_Virtualize_Port* 只能處理那個內介資料結構，因此除非是在兩個 events 之間，否則就沒有用。這樣的需求意味你應該指定一個初始化次序（initialization order），指定於 VCD 之前，並把你的 *VCD_Virtualize_Port* call 放在你的 *Device_Init* 處理常式之內。

Contention 的處理是一個重點，因為 VCD 的 *ACQUIRE_RESOURCE* 函式會把 port 的擁有權交給 system VM。如果你要攔截來自 MS-DOS VM 的一個 serial I/O 動作，你必須將擁有權給予該 VM。為了這麼做，你必須針對 MS-DOS VM 呼叫 *VCD_Acquire_Port_Windows_Style*，並指定「VCD contention 函式將會用到」的其他參數。

COMxAutoAssign 項目　針對每一個 COM port，使用者可以在 SYSTEM.INI 檔案中設定 *COMxAutoAssign* 項目（*COM1AutoAssign*, *COM2AutoAssign* 等等）爲一個 timeout 值（以秒爲單位）。例如，*COM2AutoAssign=2* 的意思是，如果上一次 I/O 動作已經過

去了 2 秒，VCD 應該判決 COM2 的前一個擁有者已經完成了對 COM2 的使用。如果其值爲 0，表示 VCD 應該總是重新指定擁有權，不需任何的時間等待。如果其值爲 -1，表示 VCD 應該總是顯現一個「contention 對話盒」。未公開值 -2 是個預設值，意思是 VCD 應該無論如何總是不讓搶奪行爲成功。

當 VCOMM port driver 使用 VCD 的競爭函式（contention function）來要求一個 port 的擁有權，「虛擬化」有了輕微的改變。Contention function 使用 *VCD_Acquire_Port_Windows_Style* service，將 port 的擁有權給予 System VM。在處理這個 service call 的過程中，VCD 將新擁有者（一個 VM）內的 "trapping" 解除作用（disabled），但仍然讓其他所有 VMs 的 "trapping" 繼續保持作用。解除 "trapping"，理論上的效果是允許 System VM 直接處理硬體。但是使用 COMM .DRV Windows 95 版之應用程式沒有得到這項自由所帶來的好處：它們將呼叫 VCOMM，而所有的 port I/O 動作無論如何都在 ring0 port driver 中發生。

最後值得注意的一點是，VCD Windows 95 版繼續保持與 Windows 3.0 和 3.1 之通訊驅動程式的相容性。這項支援包括古董級的 16-bit ring DLLs。這項支援是如何做到的呢？噢，這個題目超過了本書範疇。

Modem Drivers

沒有人需要在 Windows 95 中寫一個特別的 VxD 來處理 modem（數據機）。幾乎市場上所有的 modems 都接受 Hayes 公司所發明的 AT command set，Microsoft 也已經提供了一組 ring3 drivers 和 ring0 drivers 給 AT-command 相容數據機使用。大部份 modem 製造廠商需要做的（爲了獲得 Windows 95 相容性）是產生一個 .INF 檔，其中定義其 modem 所瞭解的命令和回應。

UNIMODEM.VXD 就是那個支援 AT-command 相容數據機，給 telephony applications（也就是使用 TAPI -- Telephony API -- 之應用程式）所用的 VxD。Microsoft 並沒有公開 UNIMODEM.VXD 的原始碼，但或許其實也沒有必要這麼做，因爲幾乎沒有人需要

複製它。UNIMODEM.VXD 是一個奇妙的東西：它既是一個 VCOMM port driver，也是一個 VCOMM client。我將在稍後數段解釋我的意思，在那之後，我們便將結束對 Windows 95 通訊議題的討論。

爲了測定 UNIMODEM.VXD 如何裝配到系統架構之中，我使用【Add New Hardware】（新增硬體精靈）來安裝一個一般的 28,800 bps modem。最終我登錄了一些條目（entries），任命 VCOMM 爲 device loader（圖 **14-13**）。請注意，這張圖中，modem 的 software key 有一些 subkeys，其名如 Answer, Hangup 等等。這些 subkeys 內含此一 modem 的命令字串及回應。例如，Hangup 有唯一一個 named value (1)，被設定爲 ATH<cr>。你或許知道，這便是標準的 Hayes modem 命令，意思是掛上（hanging up）電話。

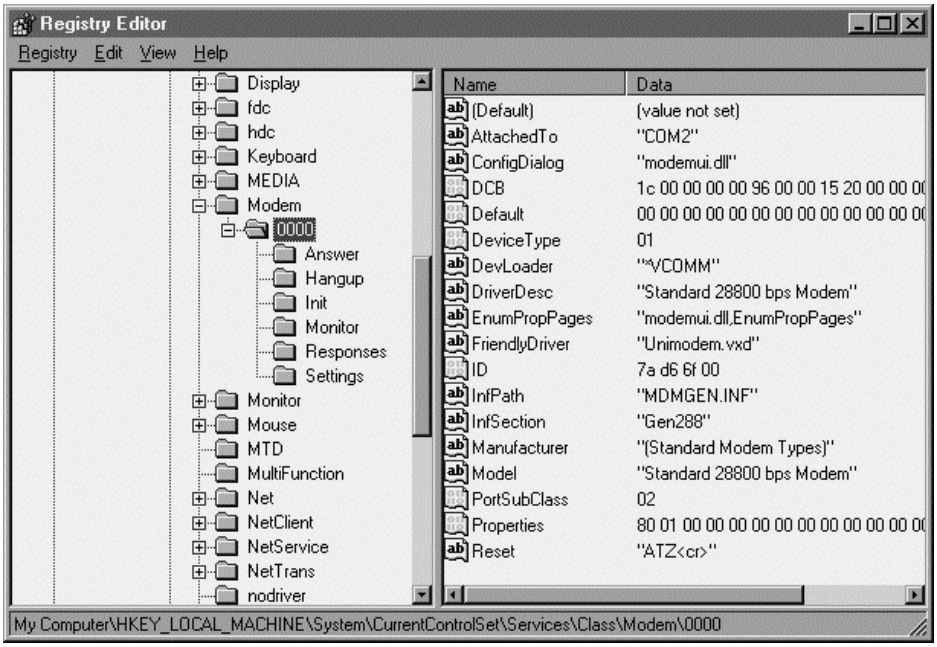


圖 14-13　一個 Modem 的 Software key

Modem 的 software key 和我們在本書所學到的其他 Plug and Play devices 不同，它並沒有一個 named value "PortDriver"。對此處而言，重要的 named values 是 "FriendlyDriver"（被設定爲 UNIMODEM.VXD）和 "AttachedTo"（被設定爲 COMx）。這些數值告訴 VCOMM 說，以此親和性之 modem 名稱所打開之 device，其 port driver 名爲 UNIMODEM.VXD；該 modem 附著在 COM*x* port 上面。

當一個 telephony application 以「指定給此 modem 之親和性名稱」打開一條電話線，VCOMM 最終會載入 UNIMODEM.VXD 做爲 port driver。UNIMODEM.VXD 會像我們先前所討論的一樣，註冊一個 driver control procedure、呼叫 *_VCOMM_Add_Port*、等等等。然而當它被要求打開 port，UNIMODEM.VXD 轉而呼叫 *_VCOMM_OpenComm* 以打開 *AttachedTo* port。這個巢狀式的 open 方式會使得 VCOMM 載入 SERIAL.VXD。接下來發生於 COMx 的事情相信你已經知道了。

UNIMODEM.VXD 的許多 port driver 函式其實是直接轉交到底層的 serial port driver 手上。事實上，UNIMODEM.VXD 打破了公開規則，它直接呼叫隸屬於 serial driver 的 *PortFunctions* 表格。除了一種情況，它會首先檢查函式指標是否爲 0。（唯一的例外是 serial driver 的 *SetModemStatusShadow* 函式，這是一個早期函式，並不先檢查是否爲 0。其原因是，其他函式都可有可無，唯獨這個函式一定要由 serial port driver 供應。UNIMODEM.VXD 從 *PortData* 結構中獲得 *PortFunctions* 表格位址，而 *PortData* 的位址是呼叫 *_VCOMM_OpenComm* 獲得的。雖然官方文件上說 *_VCOMM_OpenComm* 傳回的 handle 不一定和一個 port driver 的 open 函式所傳回的相同，但你可以看到，它們其實是相同的。我可以想像一個 UNIMODEM.VXD 的化身，使用公開的 VCOMM service API 來獲得 port driver，但效率上的考量卻暗示它應該使用一種拼湊但快速的作法。

至於那些並不只是「簡單地轉交到 serial port driver 手上」的 UNIMODEM.VXD port driver 函式，會涉及少量的額外處理程序。舉個例子，在 UNIMODEM.VXD 呼叫 serial driver 以處理 *SetModemStatusShadow* 之前，它會做出自己的一份 MSR shadow byte 副本；它的 *Read* 和 *Write* 函式會更新統計數字，其他方面則延至 serial port driver 再來

做。事實上，UNIMODEM.VXD 很像一個 C++ 類別，衍生自某個 serial port 類別（此類別類似本章先前所討論的 *CSerialPort*）。

第 15 章

Block Device Drivers

譯註：synchronous：同步　asynchronous：非同步　mount：裝載　reentrant：重進入

你對 Windows 95 的內部知道得愈多，你愈可能大聲地說：「難道沒有人可以讓我擺脫真實模式作業系統（MS-DOS）的陰影嗎」？早版的 Windows 系統之中，兩大主要的束縛就是底層的 FAT 檔案系統和磁碟 I/O 子系統。我將在下一章介紹 Windows 95 的 Installable File System Manager，它終於取代了 MS-DOS 的檔案系統。這一章我要討論的是新的 32 位元磁碟 I/O 保護模式子系統（protected mode subsystem）。

很難概略說明「允許 Windows 95 不必離開 32 位元保護模式就能夠進行磁碟 I/O」這項技術有多重要。「不斷切換至 V86 模式，俾使 MS-DOS 的 disk drivers 得以執行 page swapping 動作」，這個事實替早版的 Windows 穿上了一件緊身衣！你知道，MS-DOS 一直不能夠重進入（reentrant），即使現在也一樣，所以你不能夠允許遞迴的 page fault 發生。真的，當 MS-DOS 正在作用的時候，你不能夠允許任何 page faults 發生。程式如果違反這條規則，可能會使系統進入「需要遞迴進入 MS-DOS 以取得（fetch）一個 page」的狀態，這會導至「沒有任何 ring0 碼可以是 pageable」；這麼一來，可能會有些 event

callbacks 在單純地呼叫理應合法的 nonasynchronous VxD services 時，不經意地失敗。另一個影響是你不能夠在 paging I/O 的過程中禁止 event service，因爲系統無法處理發出 I/O completion 的那個中斷。真實模式下的 paging 所造成的另一個影響是，page manager 無可避免會長時間擁有唯一一個 ring0 critical section，以保護 paging I/O 不受 "reentrance" 的影響。所以，Windows 95「在 ring0 之中完成 paging I/O」的能力，讓整個系統獲得了救贖，也放寬了許多原本用來阻止死結（deadlock）的規則。

儘管一個 32 位元保護模式磁碟子系統可以帶來這麼多好處，Windows VxDs 程式員還是得因爲兩個相容特性而憂心老舊的規則。第一，真實模式仍然在我們週遭，而且 Windows 95 有可能最終因爲協力廠商（third-party vendors）尚未把驅動程式轉爲 VxDs，而以真實模式碼處理 page swapping。第二，end user 可能會破壞整個 ring0 端的好事 -- 如果他們強迫 Windows 95 使用真實模式驅動程式的話。

儘管「永遠不死的真實模式」帶來不方便，檔案系統以及其他依賴 Input/Output Supervisor（IOS）來存取磁碟的系統元件，還是使用 32 位元介面來觸及 IOS。真實模式在 IOS 處理外界需求的時候有時也會這麼做。我將在這一章詳細討論 IOS，並顯示如何建造一個 port driver （它將運作於驅動程式分層架構中幾乎最底階之處）以及一個由廠商供應的 driver（它將在分層架構的中間層運作）。

IOS 的架構

Windows 95 把磁碟抽象地視爲一個 block devices，也就是一種「高速搬移固定大小之資料區塊」的 device。一個被稱爲 Input/Output Supervisor（IOS）的 VxD 實作了一個所謂的 "layered device driver model"（分層裝置驅動程式模型），容納目前以及未來可預知的磁碟技術。IOS 和許多由 Microsoft 以及其他軟硬體廠商供應的 VxDs 一起運作，協力解決 client 端（如 page swapper 和 file system manager）所要求的 block I/O 動作。IOS 甚至處理「V86 模式中執行之 MS-DOS 程式」所產生的 I/O requests。IOS 的官方文件是 Microsoft 的 "Layered Block Device Drivers"（請參考 Windows 95 DDK 中的 "Design and Implementation Guide" 一節），但你或許會發現，本章才能夠帶給你有用的資訊。

IOS 提供小量的 services（表 **15-1**）給 Windows 95 的其他系統元件使用。IOS 的主要客戶是檔案系統驅動程式，這些驅動程式把應用程式所發出的有關於檔案存取的 requests 轉化爲適當的 I/O 動作，準備執行於已經 "mounted"（裝載）到某 block device 上的 volume。*IOS_SendCommand* 是其中主要的 service，藉由它，檔案系統及其他 IOS clients 可以發出 I/O requests 並監視其狀態。IOS 使用其他的 VxD components（統稱爲 layer drivers）以執行來自 client 端的 requests。本章所討論的故事，大部份都和這些 "layer drivers" 有關。

Service	說明
IOS_BD_Command_Complete	表示一個使用了 BlockDev interface 的 I/O request 已完成。
IOS_BD_Register_Device	註冊一個與 BlockDev 相容的 VxD
IOS_Exclusive_Access	鎖住一個 drive 以便由單獨一個虛擬機器獨佔性地存取。
IOS_Find_Int13_Drive	找出某個 device（它在 Windows 95 啓動之前的真實模式中可見）的 BlockDev device descriptor。
IOS_Get_Device_List	取得由 BlockDev device descriptors 所組成的串列頭
IOS_Get_Version	取得 IOS 的版本號碼
IOS_Register	註冊一個 IOS layer driver
IOS_Requestor_Service	代替 I/O requests 發起人執行一個函式。
IOS_Send_Next_Command	送出下一個命令給某個 device
IOS_SendCommand	送出一個 I/O request 給某個 device（這是用來對 block devices 做 I/O 動作的方法）
IOS_Set_Async_Timeout	產生一個 asynchronous timeout（和 VMM 的 *Set_Async_Timeout* service 一樣）
IOS_Signal_Semaphore_No_Switch	和 VMM 的 *Signal_Semaphore_No_Switch* service 一樣
IOSIdleStatus	決定 IOS 是否爲閒置狀態（idle）

IOSMapIORSToI21	將一個 IOS 錯誤代碼映射爲其 INT 21h 對應碼
IOSMapIORSToI24	將一個 IOS 錯誤代碼映射爲其 INT 24h 對應碼

表 15-1 IOS services

IOS Clients

雖然本章談的是「如何撰寫 VxDs 以符合 IOS 內部架構」，但對於「外部的 client VxDs 如何呼叫 IOS」先有一些瞭解，倒是頗有助益。Clients 程式得和 logical disk drives 以及 volumes 打交道。所謂 **volume**，是一個由許多固定大小之資料區塊所組成的抽象容器，裝載(mounted)於「client 可以個別定位」的另一種抽象單位上。也就是說，對一個 **physical disk drive** 而言，**logical disk drive** 是軟體，而 **volume** 則是旋轉於磁碟機軸上的 physical disk 的軟體部份。 大部份時候你可以把一個 logical drive 及其所容納的 volume 視爲相同實體（entity）。

在真實世界中，logical disk drives 相當於 physical disks 裡頭的 **partitions**。我們通常使用 FDISK 工具程式在一顆新硬碟上產生一個 logical drives。Small computer system interface（SCSI）硬碟和 CD ROM 照例擁有控制權以及其他實際呈現的資訊，但是隱藏了在 logical volume 外表下的因果關係。

除非先探討 IOS 的中心資料結構，否則很難描述一個 IOS client 如何使用 IOS services。這些資料結構包括 device control block（DCB）、volume request parameters block（VRP），以及 I/O request descriptor（IOR）。但是在討論這些結構之前，最好先討論如何使用它們。因此，我決定先簡潔地以一一點名的方式來解釋一個 IOS client 執行 I/O 動作時所經歷的路程。也就是說，我將像面對老朋友似地面對各式各樣的資料結構。然後，在剩餘各節，我才詳細地討論那些資料結構。本章最後一節，我再回頭說明程式員如何使用這些資料結構搭配各種 IOS services，完成有用的結果。

Windows 95 IOS Clients

一個 IOS client 需要一些非固定的方法來知道系統中現存著什麼 devices。Installable File System（IFS）Manager 藉由 volume mounting protocol 來提供這樣的知識給 file system drivers（FSDs）知道。稍後我再來討論 IFS。目前你必須知道的就是，FSDs 使用 DCB 來描述一個 logical disk drive，並使用 VRP 來描述裝載於 logical drive 上的 volume。IFS Manager 使用 *IOS_Requestor_Service* 的各個子功能來產生並管理 VRPs。從 DCB 和 VRP 出發，client 可以經歷以下步驟，執行一個 I/O 動作（像是讀取一個 sector 之類的動作）：

1. 產生一個 IOR，並以欲執行之運算的相關資訊來填寫其內容。Client 使用一個 IOS service 函式來配置 IOR 所需記憶體。
2. 喚起一個所謂的 "criteria" 函式以修改 I/O request，以便使它適合「即將處理此 request 之 drivers」的需求。這個 criteria 函式做的事情包括將傳輸單位從 sectors 轉換爲 bytes，並將虛擬位址轉換爲實際位址的 scatter/gather descriptors。
3. 將 IOR 位址交給 *IOS_SendCommand*。儘管文件中建議你也應該將 DCB 位址傳過去，但 IOS 事實上是自己去找 DCB。IOR 中有一個 flag 位元用來指示是否這個 IO 動作應該被同步處理。如果是同步動作，IOS 會等待，直到整個動作完成了才回返。但如果是非同步動作，IOS 把動作放進 queue 之後便立刻回返。而當此動作於稍後完成，IOS 將呼叫一個位址記錄於 IOR 中的 completion 函式。
4. 分析這個 IO 動作的 completion 狀態，並採取適當的反應。
5. 呼叫一個 IOS service，釋放被 IOR 佔用的記憶體。

書附光碟中有一個小程式（IOSCaller）可解釋上述程序。你可以在 \CHAP15\IOSCALLER 目錄中找到它。

Windows 3.1 Block Device Clients

不屬於 Windows 95 file system drivers 的那些 VxDs，可以使用 *IOS_Find_Int13_Drive* 來搜尋一個特定的 physical drive，或使用 *IOS_Get_Device_List* 來列舉所有的 physical drives。這兩個 IOS services 全是爲了相容的目的才存在，俾能夠支援那些使用 Windows 3.1 block device driver 的 VxDs。那些「以舊式之 block device 介面來使用 INT 13h drives」的 VxDs，使用一個 BlockDev device descriptor（BDD）來描述一個 physical drive（注意，並沒有純粹的 logical INT 13h drive）。它們並不使用 VRP 結構，因爲 Windows 3.1 之中沒有相應的觀念存在。

IOS 維護 BDD 結構，就好像是它自己的 DCBs 的一部份。事實上，BDD 是從 DCB 的 offset C3h 開始。由於幾乎 Windows 95 使用的所有控制區塊（control blocks）都是以 4-byte 邊界開始，經由這種嚴謹而奇特的位址，你可以輕易在除錯的時候找到正確的 BDDs 位址。

使用「Windows 3.1 之 block device interface」的 VxDs，執行以下步驟以完成 I/O 動作：

1. 產生一個 *BlockDev_Command_Block*（BCB）結構來描述此一 I/O 動作。Clients 使用它們自己的空間來安放這些結構。
2. 將 BCB 和 BDD 位址傳遞給 *IOS_SendCommand*。IOS 可以藉由一個 flag 位元來分辨這種伴隨著 BCB 和 BDD 指標的呼叫，和一個伴隨著 IOR 和 DCB 指標的 Windows 95-style 呼叫之間的差異，IOR 結構中的 *IOR_flags* 欄位位於 offset 08h 處，其 400h 位元被稱爲 *IORF_VERSION_002*。BCB 結構中的 *BD_CB_Flags* 欄位也位於 offset 08h 處，但在 Windows 3.1 中並不會設立 400h 位元。因此，如果 *IOS_SendCommand* 看到 400h 位元於 offset 08h 處（從 request structure 的起頭算起）被設立，就表示它有一個 Windows 95-style request；否則它會認爲它有一個 block device request。
3. 等待，直到 IOS 呼叫 BCB 所指定的 callback 函式以檢查該 request 的 completion 狀態（所有使用 block device interface 的動作都是非同步的）。

IOS Layer Drivers

IOS 定義了 32 層 driver 機能（圖 **15-1**）。Layered architecture（分層架構）的目的是希望根據一個可能的抽象模型，模塑出「block devices 如何運作」，以便將各機能分離，並安裝到整個系統之中。IOS 以 DCB 表示每一個 physical device。一個 DCB 內含一個有序的、由函式組成的 calldown list，這些函式分攤處理相關之 physical device 的 I/O requests。Calldown list 中的每一個函式，佔用 32 個可能的 layers 中的一層，並從上一層 layer 接收 I/O request packets。一個函式可以完全處理 packet，也可以修改 packet 內容然後傳遞給 calldown list 中的下一層（較低層）layer，或者它也可以執行一些全新的 requests，這些 requests 累積實現了原來的 request。

雖然在完整的 IOS driver model 中有 32 個 layers，但只有其中一些 layers 是真正被 driver 佔用。圖 **15-2** 表現出我的系統中的三個 disk drives 的實際 drivers 分層情況。我的軟碟機包括五層（layers）drivers：一個 volume tracker（level 5），一個 type-specific driver（level 7），一個 IOS 內部秘密元件（level 18），一個 floppy driver（level 27），以及一個 handler of last resort（level 31）。layers 不僅可以空缺無物，而且不同的 devices 可以在某些 layers 內含相同或不同的 drivers。例如，所有的 devices 共享 IOS 的第 31 層 "last-resort handler"。我的 CD-ROM drive 和 floppy drive 共享 volume tracker（level 5），因爲兩者都有可抽取的儲存媒體。我的系統中的一個 Integrated Device Electronics drives（IDE）硬碟機就沒有 level 5（因爲其儲存媒體絕不可能改變），但它有一個 "miscellaneous" port driver（level 19）以及一個 Enhanced Small Device Interface（ESDI）port driver（level 22）。

Layer Drivers 的功能

每一個 layer 中的 drivers 有自己明確的責任。只有一個例外，那就是最低 levels 中的 drivers，你可以把它們總括標示爲 "**port drivers**"，Microsoft 並沒有對它們的責任做清楚的規範。「沒有細節規範」或許其實是必要的，因爲 Microsoft 自己提供了高階和中階的大部份 drivers，而許多由廠商供應的 drivers 則有少量而寬廣的特權。

File System Drivers

File system drivers（FSDs）管理來自應用程式的高階 I/O requests。我將在第 16 章探討 IFS Manager 時，對 file system drivers 做更多討論。

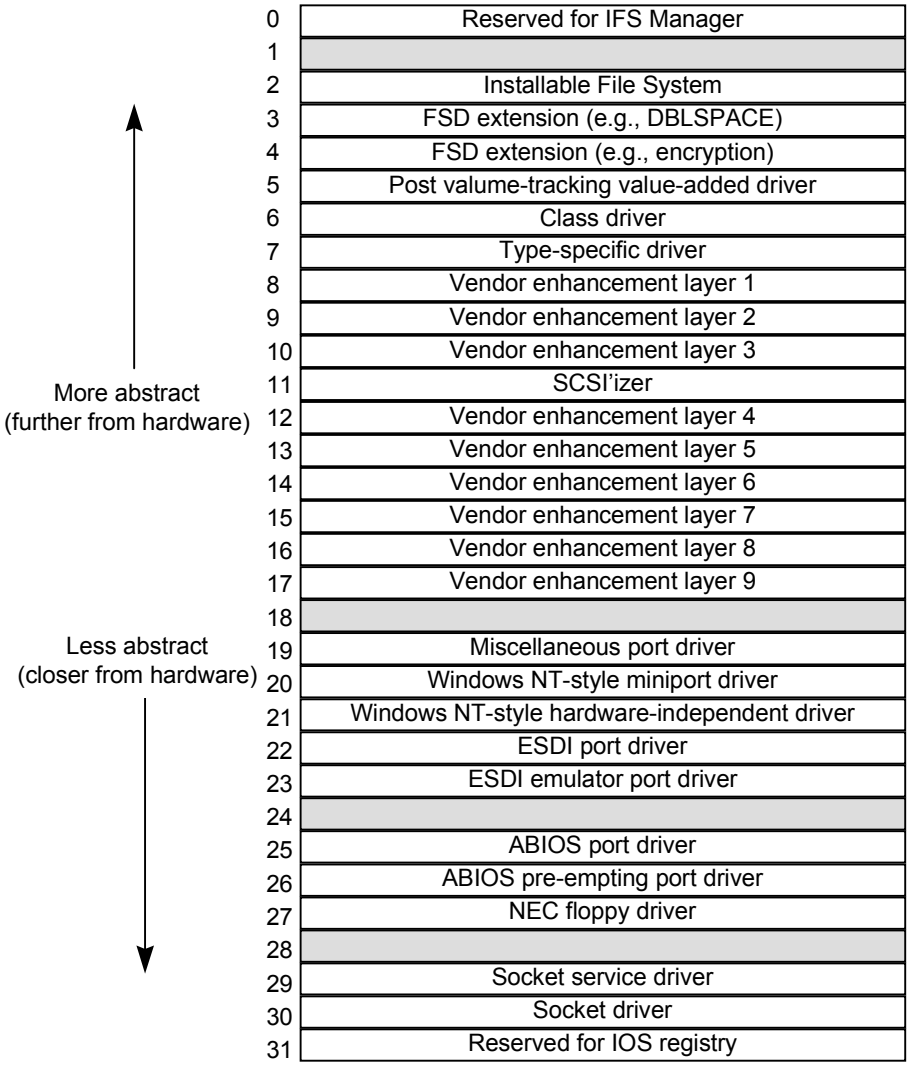


圖 15-1　32 層的 driver 機能

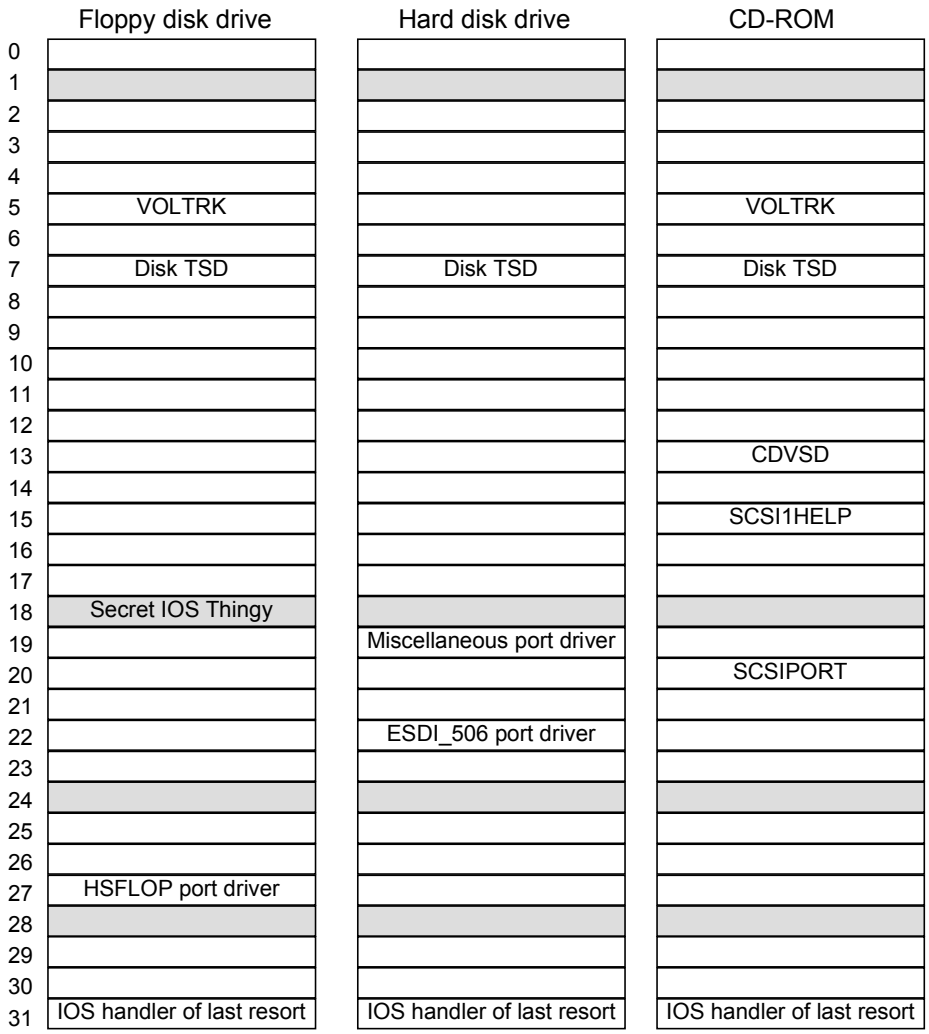


圖 15-2　我的系統中的三個 disk drives 的實際 drivers 分層情況

Volume Trackers

一如其名稱所指，volume tracker（例如 VOLTRACK.VXD）負責追蹤哪一個 volume 目前被安插至一個「有著可搬移之儲存媒體」的 device 上。Volume tracker 監視著 device，看看儲存媒體有無更動，並與 end user 交談，使他們得以確定預期之儲存媒體是否真的存在。I/O request 有一個隱含參數，是個結構（VRP），用來任命「IOS client 欲與之交談」的某一媒體。另有其他描述 physical devices 之資料結構，又含入 VRPs。所以 volume tracker 將驗證 device 是否有一個「client 想要與之合作」的 volume，並採取任何步驟來修正任何錯誤。

特類型的（Type-Specific）Drivers

一個 type-specific driver（TSD）擔負著「某一類型之 devices」的全部職責。典型的系統將含有一個 Disk TSD（DISKTSD.VXD）以及一個 CD-ROM TSD（CDTSD.VXD）。一個 TSD 的作爲包括：

- 檢驗 BIOS 和 MS-DOS 之中有關於 disk devices 的資料，它們在 Windows 95 啓動之前的真實模式中便已可見，以便能夠提供給保護模式的 driver 去處理相同的 devices。
- 從一個硬碟的 master boot record 中讀取 partition table，並建立一些資料結構，用以描述共享 physical media 的那些 logical devices。
- 將 requests 從「以 logical sector 爲基礎」轉換爲「以 physical sector 爲基礎」。

Vendor-Supplied Drivers

Vendor-supplied drivers（VSDs）不太容易分類。VSD 的目的之一是允許硬體廠商或軟體廠商實作出「非標準機能」或所謂的「專屬機能」。不過有些 VSDs 是系統的標準配備。請注意，VSD 的 layer 號碼（8~10 以及 12~17）被 layer 11 打斷，那是一個 SCSI'izer。因此，一個 VSD 可以將它自己適當地安插到 calldown stack 中的 SCSI'izer 的上層或下層。

大部份 VSDs 用來與 SCSI devices 相配合。Non-SCSI devices 則主要被實作於 port driver。其他 layers 的 drivers 都沒有真正的必要。標準的 CDVSD, DISKVSD 以及 SCSI1HLP drivers 就是所謂的 SCSI-related VSDs。CDVSD 會針對 SCSI-2 CD-ROM 和 audio devices 產生 SCSI 命令，DISKVSD 則是針對 SCSI-2 disk drives 產生相同的動作。SCSI1HLP 會把 SCSI-2 packets 轉換爲一個 SCSI-1 device 需要的任何格式（即使是非標準的格式）。

我將在本章討論如何建立一個 VSD，其作用只爲了監視 I/O requests 的流程，並以此來教育系統程式員。你可以寫一個 VSD 來執行壓縮、加密、或任何你可能想像的其他用途。不幸的是，由於 IOS 毫無把關地載入任何它在 IOSUBSYS 磁碟目錄中發現的 VxDs，並把它們視爲一個 VSD，因此提供了電腦病毒的一個可乘之機！

SCSI'izers

一個 SCSI'izer（例如 APIX.VXD）爲特定類型的 device 建立起 SCSI command descriptor blocks，並執行 device 層級的錯誤復元和運轉記錄（logging）。

Port Drivers

針對 SCSI devices，Windows 95 使用 Windows NT-style miniport drivers。SCSI miniport drivers 的副檔名是 .MPD。Windows 95 的 miniport drivers 將 Windows NT 的 miniport drivers 做了輕微的改善。它使用 portable executable（PE）檔案格式，因此可直接以二進位形式在 Windows NT 中派上用場。miniport driver 取代「由 Microsoft 提供之與 device 無關的 SCSI port driver」，處理一些因 adapter 而異的工作。這些工作包括追蹤狀態的轉變、在動作特定點導入必要的延遲、將狀態碼翻譯爲標準值等等等。如果想要學習 SCSI miniport drivers 的更多知識，請參考 Windows NT 3.51 DDK 標題 "Part 3 SCSI Drivers" 的說明文件內，名爲 "Kernel Mode Drivers" 那一節的 "Reference" 小節（請利用搜尋工具尋找 "Part 3 SCSI Drivers"）。我不打算在本書中討論這個主題。

對於 non-SCSI devices，IOS 使用副檔名爲 .PDR 的 port drivers。一個 port driver 會對

一個 physical device 產生 I/O 命令、規劃 DMA 傳輸、解釋狀態資訊，並處理硬體中斷。Microsoft 已經針對一些一般性的 devices 提供了 port drivers。例如 ESDI_506.PDR，就是 ESDI 和 IDE disk driver 的 port driver，HSFLOP.PDR 是 NEC 軟碟機的 port driver。硬體製造商也可能必須供應個別的 port drivers。

我將在本章談到如何爲一個 RAM disk 建立一個 port driver，謝天謝地，這並不需要精通 port 程式設計技巧或中斷處理技術。

Real-Mode Mappers（眞實模式映射器）

某些 devices 仍然繼續需要真實模式 device drivers 的幫助。可能原因是，這個 device 沒有一個可被大家接受的保護模式 driver，或者是，Windows 啓動時所找到的真實模式 driver 的機能，沒有任何保護模式 driver 模擬得出來（補上被遺漏的機能，應該是產生 VSD 的一個好理由）。這種情況下，Windows 95 必須將 I/O requests 傳遞到原已存在的 MS-DOS drivers 手中，使用的是我在第 10 章所討論的「中斷下傳機制」。真實模式中的對應者必須負責完成下傳後的任務。

載入 Layer Drivers

IOS 使用一種富有彈性的體制來載入「爲完成一個特別的 configuration」所需要的所有 device drivers。Configuration Manager 驅動整個程序的方式是：辨識系統中的所有 adapters 和 controllers。一個 block device controller 的 software key，會把 *IOS 任命爲 device loader，並將某一 port driver 任命爲 port driver。請參閱**圖 15-3**（林居輝註：注意，圖 15-3 的 **DevLoader** 值爲 "IOS"，**PortDriver** 值爲 "HSFLOP.pdr"）。所有 IOS port driver VxDs 的副檔名皆爲 .PDR，被放置於 Windows system 目錄下的 IOSUBSYS 子目錄中。（SCSI miniport drivers 也被放在相同地點，副檔名爲 .MPD。不過它們並非 VxDs）。當 IOS 將每一個 port driver 初始化，port driver 便驗證其 controller hardware 是否真的存在。IOS 然後便爲每一個附著於此 controller 的 physical device 產生一個 DCB。

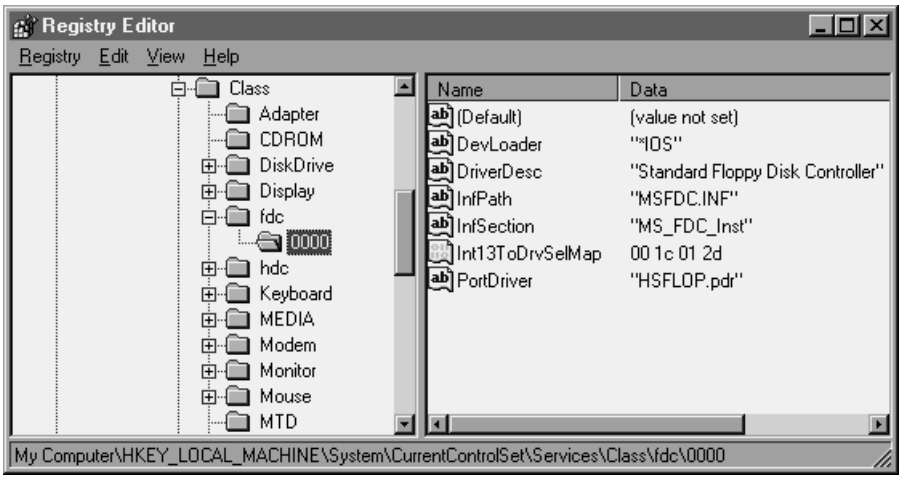


圖 15-3　Sample port driver software registry key

IOS 也會載入它在 IOSUBSYS 目錄中所發現的所有 value-added drivers （副檔名爲 .VXD）。這些檔案都是駐留在 IOS driver hierarchy 較高層的 VxDs。IOS 給予每一個 value-added drivers 一個機會，讓它們將自己安插到每一個 DCB 的 calldown list 之中。那些最終沒有被附著到任何既存之 device 上的 VxDs，則被卸載並釋放記憶體。

和 IOS 共事的技術

如果你從頭到尾一路讀過本書，現在的你應該已經對於如何建造一個 VxDs 有了堅固的基礎。這裡另有一些額外的規則，是在與 IOS 搭配時必須遵循的。第一，你最好有心理準備，要打很多字：大約有兩打的表頭檔需要含入，而它們所定義的結構，欄位名稱都很長。不僅如此，結構中大部份的指標值被定義爲 ULONG 或 PVOID，以至於你必須時常將之轉型爲正確的指標型別。不要再使用正規的 VxD service calls 來申請 IOS 服務，現在的你應該使用一個 function-based 機制（依賴 IOS 提供的一個函式指標表格）。Service calls 使用參數 "packets" 來取代堆疊中的參數，這使它們與正常的副程式（函式）稍有不同。

IOS 的表頭檔

和其他的 Windows 95 元件不同，IOS 傾向於針對每一個主要的資料結構使用各別的表頭檔。表 **15-2** 列出你可能需要的表頭檔。其中一些檔案（檔名有星號者）可以在 DDK 光碟的 \DDK\BLOCK\INC 目錄找到。DDK Setup 程式並不會把它們拷貝到你的硬碟的 \DDK\INC32 目錄中。

表頭檔（.H 或 .INC）	內容
AEP †	Asynchronous event 代碼和結構
BLOCKDEV	針對 Windows 3.x block devices 而做的相容性宣告
CONFIG (*)	IOS 設計領域中的常數
DCB (*)	Device control block（physical 或 logical device 所使用的主要結構）
DDB (*)	Device data block（controller 的主要結構）
DEFS (*)	一些隨機（零亂）的型別定義，加上一些 8250 相關常數
DRP	Driver registration packet.
DVT (*)	Driver vector table (與 IOS 共同運作之 VxD 所使用的主要結構)
ERROR (*)	標準的 runtime library 表頭，定義 MS-DOS 錯誤代碼。
IDA (*)	IOS 的內部資料結構
ILB	IOS linkage block（定義函式指標，用來呼叫 IOS）
IODEBUG (*)	IOS VxDs 的除錯宣告
IOP (*)	I/O request packet（IOP）結構（每一個 I/O request 的主要結構）
IOR	I/O request descriptor（IOP 的一部份，卻成爲分離的結構和表頭檔）
IOS	IOS service 的宣告。 只有 .INC 檔可任意使用
IRS	*IOS_Requestor_Service* 的函式碼及結構
ISP	IOS service request 函式碼及結構（VxDs 和 IOS 之間的主要介面）
IVT (*)	內部所用的 IOS vector table 結構
MINIPORT (*)	SCSI miniport drivers 的宣告
SCSI (*)	SCSI port and class driver 的宣告

表頭檔（.H 或 .INC）	內容
SCSIDEFS (*)	SCSI request 宣告
SCSIPORT (*)	關於 SCSI port driver 的更多宣告
SGD	Scatter/gather descriptor.
SRB (*)	SCSI request block.

† Windows 95 DDK 中的 AEP.H 在 *AEP_rpm_change* 結構定義中有一個語法錯誤。我已經在本書光碟的 \CHAP15 和 \CHAP16 目錄中放了一個正確版本。你應該將此正確版本拷貝到你的 \DDK\INC32 目錄中。

表 15-2　IOS header files

我很希望能夠發現一個名爲 IOS.INC 或 IOS.H 的綜合表頭檔，把所有必要的檔案都含進來。由於並沒有這樣的檔案，所以你要不就自己寫一個，要不就把各表頭檔一個個含入你的專案中。如果你打算以 C 或 C++ 來撰寫，你還必須提供自己的 service table 宣告以及外包函式（wrappers），因爲 DDK 並沒有那些東西。事實上，DDK 中的 IOS.H 檔內全都是註解文字（而那些文字並非很有用）。爲了本章所呈現的一個實例，我建立自己的 IOSDCLS.H 檔（含入本書光碟 \CHAP15 目錄中的所有檔案），它看起來像這樣：

```
#ifndef IOSDCLS_H
#define IOSDCLS_H

#ifdef __cplusplus
extern "C" {
#endif

#define WANTVXDWRAPS
#include <basedef.h>
#include <vmm.h>

#include <aep.h>
#include <blockdev.h>
#include <config.h>
#include <dcb.h>
#include <ddb.h>
#include <drp.h>
#include <dvt.h>
```

```
#include <ida.h>
#include <ilb.h>
#include <ior.h>
#include <iop.h>
#include <irs.h>
#include <isp.h>
#include <ivt.h>
#include <sgd.h>
#include <vrp.h>

#undef _IRS_H          // suppress extra IOS_Requestor_Service
#include <vxdwraps.h>

// Some annoying inconsistencies in typedef spelling:

typedef pIOP PIOP;
typedef pDDB PDDB;

...

[MANY additional declarations]
...

#ifdef __cplusplus
}
#endif

#endif // IOSDCLS_H
```

Driver 的初始化

一個與 IOS 一起運作的 VxD，將以一種十分不同於一般 VxD 的方式來初始化。這些 VxDs 可以被動態載入，並且必須被安裝於 IOSUBSYS 目錄中，副檔名爲 .PDR 或 .VXD。你可以在你的 driver 的 *Declare_Virtual_Device* 巨集中放置一個指向特殊資料結構（也就是 driver registration packet，DRP）的指標。你需要處理的唯一一個系統控制訊息（system control message）就是 *Sys_Dynamic_Device_Init*。在你的處理常式中，你應該呼叫 *IOS_Register* service。你應該檢查 *IOS_Register* 的結果以決定你的 driver 是否應該暫停載入。你的 *Sys_Dynamic_Device_Init* 處理常式的傳回值將反映出這項決定。

Driver Registration Packet（DRP）

圖 15-4 顯示 DRP 結構格式。DRP 是一個暫時性的結構，只有在呼叫 *IOS_Register* 時才有用。你可以把這個結構放到你的 initialization data segment 中（林居輝註：此只用一次）。其中的 *DRP_eyecatch_str* 欄位內含 'XXXXXXXX'，可以幫助你在除錯時輕易識別出此一結構。

```
typedef struct DRP
  CHAR   DRP_eyecatch_str[8];  // 00 EyeCatcher 'XXXXXXXX'
  ULONG  DRP_LGN;              // 08 flags for levels at which driver
                               //    operates
  PVOID  DRP_aer;              // 0C asynchronous event procedure
  PVOID  DRP_ilb;              // 10 address of permanent IOS linkage block
  CHAR   DRP_ascii_name[16];   // 14 name of driver
  BYTE   DRP_revision;         // 24 revision level
  ULONG  DRP_feature_code;     // 25 options for how asynchronous events are
                               //    sent to driver
  USHORT DRP_if_requirements;  // 29 compatible system bus types
  UCHAR  DRP_bus_type;         // 2B controller bus type (port drivers only)
  USHORT DRP_reg_result;       // 2C return code from IOS_Register
  ULONG  DRP_reference_data;   // 2E reference data for AEP_INITIALIZE
  UCHAR  DRP_reserved1[2];     // 32 alignment
  ULONG  DRP_reserved2[1];     // 34 reserved
} DRP, *PDRP;                  // 38
```

圖 15-4　Driver Registration Packet（DRP）的結構佈局

DRP_LGN 是一個 bit mask，你可以利用它指定 32 個可能的 driver levels 中的一個（如**圖 15-1**），那是你打算在上面動作的一層。這個 mask 有可能（但通常不）設立一個以上的位元。**表 15-3** 列出各個位元的意義。

DRP_aer 指向 locked code 中的一個函式，IOS 將呼叫它以處理非同步事件（asynchronous events）。*DRP_ilb* 指向一個 IOS linkage block（ILB）結構，你必須在「locked 記憶體」中配置它，由 IOS 填寫其內容。稍後當你喚起 IOS service 時，這個 ILB 將扮演中心角色。*DRP_ascii_name* 和 *DRP_revision* 是可有可無的的欄位，IOS 不會在乎其內容。

Level	Load Group Masks	說明
0	DRP_IFS	IFS Manager
2	DRP_FSD	Installable File System
3,4	DRP_FSD_EXT_1, DRP_FSD_EXT_2	FSD extensions
5,	DRP_VOLTRK	Volume tracker
6,	DRP_CLASS_DRV	Class driver
7,	DRP_TSD	Type-specific driver
8~10	DRP_VSD_1~DRP_VSD_3	Vendor-specific driver
11	DRP_SCSI_LAYER	SCSI'izer
12~17	DRP_VSD_4~DRP_VSD_9	Vendor-specific driver
19	DRP_MISC_PD	Miscellaneous port driver
20	DRP_NT_MPD	Windows NT-style miniport driver
21	DRP_NT_PD	Windows NT-style hardware-independent port driver
22	DRP_ESDI_PD	ESDI port driver
23	DRP_ESDIEMUL_PD	ESDI emulator port driver
25	DRP_ABIOS_PD	ABIOS port driver
26	DRP_ABIOS_PREMPT_PD	ABIOS pre-empting port driver
27	DRP_NEC_FLOPPY	NEC floppy driver
29	DRP_SOC_SER_DRV	Socket service driver
30	DRP_SOC_DRV	Socket driver
31	DRP_IOS_REG	IOS registry

表 15-3　load group numbers 的 Mask values

DRP_feature_code 是一個 bit mask，用來指定 options，關係到你想要如何讓 asynchronous events 送到你的 driver 手上。我將在探討 asynchronous events 時再來討論這些 bits。*DRP_if_requirements* 代表 "interface requirements"，但不再被使用。*DRP_bus_type* 用來指示你的 driver 與哪一種 I/O bus 交涉（請看表 **15-4**）。你設定此

欄位的方式，將告訴 IOS 如何送給你 asynchronous *AEP_DEVICE_INQUIRY* events（稍後討論）。IOS 設立 *DRP_reg_result* 做爲 *IOS_Register* 的傳回值。這個欄位的內容基本上是告訴你，是否要保持載入。*DRP_reference_data* 可以是個 32 位元任意值，IOS 會把它傳遞給你的 asynchronous *AEP_INITIALIZE* event handler。

Bus Type Code	說明
DRP_BT_ESDI	ESDI bus 或 ESDI emulator（包括 IDE）
DRP_BT_SCSI	SCSI bus 或 SCSI emulator
DRP_BT_FLOPPY	NEC floppy bus 或 floppy emulator
DRP_BT_SMART	Smart device bus
DRP_BT_ABIOS	ABIOS bus 或 ABIOS emulator

表 15-4　DRP_bus_type 欄位的可能值

一個以 C 撰寫並使用 DDK 的 port driver，通常包括一個小的 assembly 程式，用以宣告 VxD 並定義其 control procedure。你可以把 DRP 放在該檔案中，例如：

DEVDCL.ASM

```
#0001  ;    DEVDCL.ASM -- Device declaration and control proc for IOS VxDs
#0002          .386p
#0003          include vmm.inc
#0004          include drp.inc
#0005          include ilb.inc
#0006
#0007          extrn _OnAsyncRequest:near
#0008
#0009  VxD_IDATA_SEG
#0010          public _theDRP
#0011
#0012  ;    Declare ourselves as a level 19 port driver:
#0013
#0014  _theDRP  DRP   <EyeCatcher, DRP_MISC_PD, \
#0015                 offset32 _OnAsyncRequest, \
#0016                 offset32 _theILB, 'Sample Port Dvr', 0, 0, 0>
```

```
#0017
#0018  VxD_IDATA_ENDS
#0019
#0020  VxD_LOCKED_DATA_SEG
#0021          public _theILB
#0022  _theILB  ILB   <>       ; I/O subsystem linkage block
#0023  VxD_LOCKED_DATA_ENDS
#0024
#0025  Declare_Virtual_Device RAMDISK, 1, 0, RAMDISK_control,\
#0026     Undefined_Device_ID, Undefined_Init_Order,,,_theDRP
#0027
#0028  Begin_Control_Dispatch RAMDISK
#0029  Control_Dispatch Sys_Dynamic_Device_Init, \
#0030     _OnSysDynamicDeviceInit, cCall
#0031  End_Control_Dispatch   RAMDISK
#0032
#0033          end
```

這個檔案可從書附光碟的 \CHAP15\PORTDRIVER 目錄中找到，其關鍵在於 *Declare_Virtual_Device* 巨集的最後一個參數。這個最後參數應該是 VxD device description block（DDB）中的 *DDB_Reference_Data* 欄位，目的是給予 IOS 一個快速方法，讓它動態載入此 driver 時，能夠儘快找到 DRP。

註冊 Driver

Sys_Dynamic_Device_Init 的處理有點簡單得過火：

```
extern DRP theDRP;            // device registration packet

BOOL OnSysDynamicDeviceInit()
   {                          // OnSysDynamicDeviceInit
   IOS_Register(&theDRP);
   return theDRP.DRP_reg_result == DRP_REMAIN_RESIDENT;
   }                          // OnSysDynamicDeviceInit
```

就像字面所顯示的那樣，你呼叫 *IOS_Register*，把你的 DRP 位址當做參數傳進去。然後檢查 *DRP_reg_result* 中是什麼樣的傳回值，並適當設定自己的傳回值。*DRP_REMAIN_RESIDENT* 的意思是：你的 driver 保持載入狀態。

注意　如果你觀察 DRP.H 檔，你會發現一個名爲 *DRP_MINIMIZE* 的傳回值代碼。DDK 範例程式把它當做 *DRP_REMAIN_RESIDENT* 的同義字。不過 IOS 已經不再使用這個傳回值代碼。

你不可以完全以字面意義來看待 *OnSysDynamicDeviceInit*！當 *IOS_Register* 結束，IOS 會呼叫你的 asynchronous event procedure （也就是 *DRP_aer* 指標所指的一個函式），以處理 *AEP_INITIALIZE* event。你應該在處理那個 event 時做你真正的初始化動作。

呼叫 IOS Service Functions

一如你所預期，IOS 開放（匯出，exports）許多 services 給 port drivers 和 value-added drivers 使用。然而，和其他系統元件不同的是，IOS 並不讓這些 services 像一般 VxD services 似地可被自由呼叫。相反地，IOS 給你一些指標，指向它的內部函式，並定義一個 "packet-based" 的方法，經由那些指標來呼叫那些函式。這些函式指標被放在 ILB 中，你的 DRP 會指出來。**表 15-5** 列出由 ILB 所指出的 service 函式。在認識這些函式之前，你必須先對 IOS 程式設計有些瞭解。現在你必須先知道 *ILB_service_rtn* service。

Service	說明
ILB_service_rtn	主要的 IOS service entry
ILB_dprintf_rtn	使用 IOS 除錯版時用來除錯的輸出函式
ILB_Wait_10th_Sec	延遲 100 個毫秒（milliseconds）
ILB_internal_request	執行一個 I/O request
ILB_io_criteria_rtn	將一個 IOR 中的 scatter/gather descriptors 從線性（linear）位址轉換爲實體（physical）位址。供外部的 IOS clients 使用。
ILB_int_io_criteria_rtn	和 *ILB_io_criteria_rtn* 相同，但卻是以一個 IOP 而非 IOR 開始。供 layer drivers 用以準備內部的 requests。
ILB_enqueue_iop	把一個 IOP 放進 queue 中排隊
ILB_dequeue_iop	從 queue 中將一個 IOP 取出

表 15-5　ILB 中的 IOS service 函式

最主要的 IOS service 函式是由 *ILB_service_rtn* 所指出的那一個。要申請一個 service，你首先得完成一個適合該 service 所用的 IOS service packet（ISP）結構，然後再呼叫該 service 函式。執行結果被放在 ISP 的各欄位中。你可能觀察到了，這個過程對於 C 程式特別麻煩。例如，要配置一個 device data block（DDB），你可能得寫出以下的碼：

```
PDDB ddb;
{
ISP_ddb_create isp;
isp.ISP_ddb_hdr.ISP_func = ISP_CREATE_DDB;
isp.ISP_ddb_size = sizeof(DDB);
isp.ISP_ddb_flags = 0;
// Call the service:
(* (void (*)(ISP_ddb_create*)) theILB.ILB_service_rtn)(&isp);
if (isp.ISP_ddb_hdr.ISP_result == 0)
    ddb = (PDDB) isp.ISP_ddb_ptr;
else
    ddb = NULL;
}
```

注意 不要把 IOS DDB 結構和 VxD 的 device description block 混淆了，後者常常也被簡稱爲 "DDB"。

順帶一提，「產生一個 DDB」是你通常會在處理 *AEP_INITIALIZE* 時做的一件事情。不同的 ISP-series services 有 20 個以上（表 **15-6**），每一個有自己的 packet 結構（但都隸屬於 ISP 系列）。

Service	說明
ISP_ALLOC_MEM	配置記憶體。 只用於與 I/O requests 相關的小量記憶體。
ISP_ASSOCIATE_DCB	將 logical DCB 與 physical device 產生關聯
ISP_BROADCAST_AEP	對外廣播 asynchronous event
ISP_CREATE_DCB	產生新的 DCB
ISP_CREATE_DDB	產生新的 DDB
ISP_CREATE_IOP	產生新的 IOP
ISP_DEALLOC_DDB	釋放 DDB
ISP_DEALLOC_MEM	釋放由 *ISP_ALLOC_MEM* 所配置的記憶體
ISP_DELETE_LDM_ENTRY	摧毀 logical device map entry.
ISP_DESTROY_DCB	摧毀 DCB
ISP_DEVICE_ARRIVED	宣佈 device 登場
ISP_DEVICE_REMOVED	宣佈 device 被移除（抽換）
ISP_DISASSOCIATE_DCB	將 logical DCB 脫離 device
ISP_DRIVE_LETTER_PICK	爲 DCB 選擇一個磁碟機代碼
ISP_FIND_LDM_ENTRY	找出 logical device map entry.
ISP_GET_DCB	針對 physical drive，找出其 DCB
ISP_GET_FIRST_NEXT_DCB	在涵蓋所有 DCBs 的迴圈中，找出第一個或下一個 DCB
ISP_INSERT_CALLDOWN	在 DCB 的 calldown list 中增加一筆項目（entry）
ISP_QUERY_MATCHING_DCBS	找出可吻合某 physical DCB 的 logical DCBs
ISP_QUERY_REMOVE_DCB	詢問是否可以移除某個 DCB
ISP_REGISTRY_READ	從 registry 中讀取 named value

表 15-6　IOS service request function codes.

沒有什麼規則可以讓我們「根據 service 函式名稱猜測對應的 packet 結構名稱」。也沒有什麼規則可以讓我們猜測那些 packet 結構中的成員前置名稱（prefixes）。我估算過，每次寫一小段下面那樣的碼，我至少會出現一個 bug 或一個編譯錯誤。爲了不要展示出

不易閱讀的實例，我在我的 IOSDCLS.H 檔案中定義了一些 inline 函式，用來產生對於 service routine 的一個有理的、以函式爲導向的介面。例如，我的表頭檔中包括下面兩個定義：

```
VXDINLINE DWORD NAKED IlbService(PVOID isp)
{
   _asm jmp [theILB.ILB_service_rtn]
}
...
VXDINLINE PDDB IspCreateDdb(USHORT size, UCHAR flags)
{
   ISP_ddb_create isp = {{ISP_CREATE_DDB, 0}, size, 0, flags};
   IlbService(&isp);
   if (isp.ISP_ddb_hdr.ISP_result == 0)
      return (PDDB) isp.ISP_ddb_ptr;
   return NULL;
}
```

那麼，用以產生一個 DDB 的程式片段就簡單多了：

```
PDDB ddb = IspCreateDdb(sizeof(DDB), 0);
```

由於我認爲官方的 service 介面對於 C 而言是如此難以使用，所以我不打算解釋任何更多的使用細節。取而代之的是，我將使用定義於我的 IOSDCLS.H 中的所謂外包（wrapping）函式。如果你想看看殘酷的細部內容以寫出 assembly 碼（或是爲了個人的好強因素），請參考光碟片中的 IOSDCLS.H。

重要的資料結構

雖然 IOS 使用大量的資料結構，但其中三個（DCB、IOP、VRP）特別重要。如果你瞭解這三個結構，就很容易瞭解 IOS 和 layer drivers 如何運作。

Device Control Block

圖 **15-5** 顯示 device control block(DCB)的格式。圖中的每一個子結構其實在 DCB.H 中都是個別被定義的結構，我把它們放在一起是爲了便於說明。DCB 結構可以有四種不同大小，視它們所表達的 device 種類而定：

- logical device 只會有一個 *DCB_COMMON* 結構。TSD 爲 physical drive 的每一個 partition 產生一個 logical DCB。因此，Logical DCB 是與相關連之磁碟機代碼最接近的一個結構。如果 *DCB_device_flags* 之中設立有 *DCB_DEV_LOGICAL* flag，表示你正在處理 logical DCB。
- physical device 將擁有一個 *DCB_COMMON* 結構，以及圖中被我標註爲 "extension for physical DCBs" 的欄位。Port driver 在回應 *AEP_DEVICE_INQUIRY* events 時，順便間接產生 physical DCB。Physical DCB 會在 *DCB_device_flags* 中設立 *DCB_DEV_PHYSICAL* flag。
- physical device 也擁有一個 *_DCB_BLOCKDEV* extension。此擴充結構提供 Windows 3.1 drivers 的相容性。
- 一個 CD-ROM device 有上述三個 DCB 子結構，再加上一個特殊的 CD-ROM extension（一個 *DCB_cdrom* 結構，並未在圖中顯現）。

給各種 **DCB** 用的公用表頭，其中只含有表示 **logical DCBs** 的部份。

```
typedef struct _DCB {
  struct _DCB_COMMON {
    ULONG   DCB_physical_dcb;       //  00 DCB for physical device
    ULONG   DCB_expansion_length;   //  04 total length of IOP extensions
    PVOID   DCB_ptr_cd;             //  08 pointer to calldown list
    ULONG   DCB_next_dcb;           //  0C link to next DCB
    ULONG   DCB_next_logical_dcb;   //  10 pointer to next logical DCB
    BYTE    DCB_drive_lttr_equv;    //  14 logical drive number
                                    //     (A = 0, etc.)
    BYTE    DCB_unit_number;        //  15 physical drive number
    USHORT  DCB_TSD_Flags;          //  16 flags for TSD
    ULONG   DCB_vrp_ptr;            //  18 pointer to VP for this DCB
    ULONG   DCB_dmd_flags;          //  1C demand bits of the topmost layer
    ULONG   DCB_device_flags;       //  20 device flags
```

```
  ULONG   DCB_device_flags2;       //  24 more device flags
  ULONG   DCB_Partition_Start;     //  28 sector where partition starts
  ULONG   DCB_track_table_ptr;     //  2C (internal) track table pointer
  ULONG   DCB_bds_ptr;             //  30 DOS BDS corresponding to this DCB
  ULONG   DCB_Reserved1;           //  34
  ULONG   DCB_eserved2;            //  38
  BYTE    DCB_apparent_blk_shift;  //  3C log of apparent_blk_size
  BYTE    DCB_partition_type;      //  3D partition type
  USHORT  DCB_sig;                 //  3E padding and signature
  BYTE    DCB_device_type;         //  40 device type
  ULONG   DCB_Exclusive_VM;        //  41 handle for exclusive access to
                                   //     this device
  UCHAR   DCB_disk_bpb_flags;      //  45 BPB flags
  UCHAR   DCB_cAssoc;              //  46 count of logical drives
  UCHAR   DCB_Sstor_Host;          //  47 Super Store volume
  USHORT  DCB_user_drvlet;         //  48 user dive letter settings
  USHORT  DCB_Reserved3;           //  4A
  ULONG   DCB_Reserved4;           //  4C
} DCB_cmn;
```

針對 **physical DCBs** 的擴充部份

```
ULONG   DCB_max_xfer_len;          //  50 maximum transfer length
ULONG   DCB_actual_sector_cnt[2]; //  54 number of sectors as seen below
                                   //     TSD
ULONG   DCB_actual_blk_size;       //  5C actual block size of the device
ULONG   DCB_actual_head_cnt;       //  60 number of heads as seen below TSD
ULONG   DCB_actual_cyl_cnt;        //  64 number of cylinders as seen below
                                   //     TSD
ULONG   DCB_actual_spt;            //  68 number of sectors per track as
                                   //     seen below TSD
PVOID   DCB_next_ddb_dcb;          //  6C link to next DCB on DDB chain
PVOID   DCB_dev_node;              //  70 pointer to DEVNODE
BYTE    DCB_bus_type;              //  74 type of I/O BUS (ESDI, SCS, etc.)
BYTE    DCB_bus_number;            //  75 channel (cable) within adapter
UCHAR   DCB_queue_freeze;          //  76 queue freeze depth counter
UCHAR   DCB_max_sg_elements;       //  77 maximum number of scatter/gather
                                   //     elements
UCHAR   DCB_io_pend_count;         //  78 number of requests pending
UCHAR   DCB_lock_count;            //  79 number of media locks minus unlocks
USHORT  DCB_SCSI_VSD_FLAGS;        //  7A flags for SCSI'izer
BYTE    DCB_scsi_target_id;        //  7C SCSI target ID
BYTE    DCB_scsi_lun;              //  7D SCSI logical unit number
BYTE    DCB_scsi_hba;              //  7E SCSI host bus adapter number
BYTE    DCB_max_sense_data_len;    //  7F maximum sense data length
USHORT  DCB_srb_ext_size;          //  80 miniport SRB extension length
BYTE    DCB_inquiry_flags[8];      //  82 device inquiry flags
```

```
  BYTE    DCB_vendor_id[8];          //  8A vendor ID string
  BYTE    DCB_product_id[16];        //  92 product ID string
  BYTE    DCB_rev_level[4];          //  A2 product revision level
  BYTE    DCB_port_name[8];          //  A6
  UCHAR   DCB_current_unit;          //  AE used to emulate multiple logical
                                     //     devices
  ULONG   DCB_blocked_iop;           //  AF pointer to requests for an
                                     //     inactive volume
  ULONG   DCB_vol_unlock_timer;      //  B3 unlock timer handle
  UCHAR   DCB_access_timer;          //  B7 used to measure time between
                                     //     accesses
  UCHAR   DCB_Vol_Flags;             //  B8 flags for volume tracking
  BYTE    DCB_q_algo;                //  B9 queuing algorithm (FIFO, sorted)
  BYTE    DCB_unit_on_ctl;           //  BA relative device number on
                                     //     controller
  ULONG   DCB_Port_Specific;         //  BB bytes for port driver use
  ULONG   DCB_spindown_timer;        //  BF timer for drive spin-down
```

針對 **INT 13h drives** 的擴充部份

```
  struct _DCB_BLOKDEV {
    ULONG   DCB_BDD_Next;                 //  C3 chain to next BDD (also
                                          //     embedded in a DCB)
    BYTE    DCB_BDD_BD_Major_Version;     //  C7 major version of diver
    BYTE    DCB_BDD_BD_Minor_Version;     //  C8 minor version of driver
    BYTE    DCB_BDD_Device_SubType;       //  C9 BlockDev device type
                                          //     (usually 0, in fact)
    BYTE    DCB_BDD_Int_13h_Number;       //  CA INT 13h unit number
    ULONG   DCB_BDD_flags;                //  CB BDF series flags from
                                          //     BLOCKDEV.H
    ULONG   DCB_BDD_Name_Ptr;             //  D3 name of device
    ULONG   DCB_apparent_sector_cnt[2];   //  D7 number of sectors as seen
                                          //     by TSD and above
    ULONG   DCB_apparent_blk_size;        //  DF block size of device as
                                          //     seen by TSD and above
    ULONG   DCB_apparent_head_cnt;        //  E3 number of heads as seen
                                          //     by TSD and above
    ULONG   DB_apparent_cyl_cnt;          //  E7 number of cylinders as seen
                                          //     by TSD and above
    ULONG   DCB_apparent_spt;             //  EB number of sectors per track
                                          //     as seen by TSD and above
    ULONG   DCB_BDD_Sync_Cmd_Proc;        //  EF command handling procedure
    ULONG   DCB_BDD_Command_Proc;         //  F3 command handling procedure
    ULONG   DCB_BDD_Hw_Int_Proc;          //  F7 interrupt procedure
    ULONG   DCB_BDP_Cmd_Queue_Ascending;  //  FB head of ascending cylinder
                                          //     queue
    ULONG   DCB_BDP_Cmd_Queue_Descending; //  FF head of descending cylinder
```

```
                                        //     queue
   ULONG   DCB_BDP_Current_Flags;       // 103 flags
   ULONG   DCB_BDP_Int13_Param_Ptr;     // 107
   ULONG   DCB_BDP_Current_Command;     // 10B
   ULONG   DCB_BDP_Current_Position[2]; // 10F
   ULONG   DCB_BDP_Reserved[5];         // 117
   ULONG   DCB_fastdisk_bdd;            // 12B
  } DCB_bdd;
} DCB, *PDCB;                           // 12F
```

圖 15-5 device control block 的格式

DCB 結構中有大量的細節資料，其中大部份並非一般 port drivers 和 VSDs 撰寫者所關心。「學習其中哪些欄位是你需要關心的」，正是學習「如何撰寫與 IOS 一起運作之 VxDs」的主要戰鬥目標。下面列表提供一些資訊，告訴你 DCB 的重要欄位。我所沒有涵蓋的欄位，就表示被 IOS 或其他一些 Microsoft 元件做爲私人用途。由於我並不打算提供 Windows 95 未公開功能之描述，所以一旦我確定哪些欄位對於你的工作沒有絕對必要，我就不說明那些欄位（不過有些欄位對於除錯有幫助，我會列出）。此外，除非我有特別說明，以及除非是爲了除錯，否則你我都可以忽略掉 DCB 的 95% 內容。

DCB_cmn.DCB_physical_dcb 這個欄位指向與此 DCB 相關聯之 physical device 的 DCB。如果這個 DCB 設立有 *DCB_DEV_PHYSICAL* flag，這個指標將指向此 DCB。TSD（它會產生一個 logical DCBs 並將之與 physical DCBs 產生關聯）有責任填寫這個欄位。

DCB_cmn.DCB_expansion_length 這個欄位內含 IOP expansion area 的總長度。Expansion area 是 IOS 在配置 IOP 結構時所保留的記憶體空間。IOS 在所有「打算將自己安插到 calldown stack」的 layer drivers 完成動作之後，計算此值。

DCB_cmn.DCB_ptr_cd 此欄位指向以 *DCB_cd_entry* 結構所組成的串列，該結構描述此 device 的 calldown stack。IOS 維護此值，並在初始化 IOP packet 時派上用場。注意，layer drivers 從一個 IOP 中（而不是從 DCB 的這個欄位）來尋找目前的 calldown pointer。

DCB_cmn.DCB_next_logical_dcb　在一個 physical DCB 之中，此欄位指向與此 device 有關聯的第一個 logical DCB。而在一個 logical DCB 中，此欄位指向下一個「關聯至相同的 physical DCB」的 logical DCB。如果這個 DCB 既是 logical 也是 physical，此值將爲 NULL。

DCB_cmn.DCB_drive_lttr_equiv　此欄位內含從 0 開始起算的索引值，表示一個 logical DCB 的磁碟機代碼。也就是說，0 代表 drive A，等等等。

DCB_cmn.DCB_TSD_Flags　這個欄位內含一些 flags，供 TSD 內部使用。Port drivers 應設立 *DCB_TSD_ACTUAL_PRE_SET* 和（或）*DCB_TSD_APPARENT_PRE_SET* flags -- 如果它們在 *AEP_ONFIG_DCB* 期間完成真正（或表面上）的 DCB "geometry" 欄位的話。如果這些 flags 被設立起來，TSD 就不會嘗試孤立無助地去計算 “geometry”（因爲可能會不正確）。

DCB_cmn.DCB_dmd_flags　這個欄位內含「與最上層 calldown stack entry 相同」的 demand flags。當一個 driver 在 *AEP_CONFIG_DCB* 期間 "hooks into" calldown stack，它會傳遞一個 "demand mask"。它會將其中的 lower-level demands 相關位元清除掉，因爲那些 demands 是它要自己實作出來的；它也會把額外 demands 的相關位元設立起來，因爲那是它打算安置於 higher-level drivers 上的。IOS 內含一個所謂的 criteria 函式，其責任就是將出現在 DCB level 上的 demands 實作出來。

當 calldown stack 在初始化過程中成長，IOS 會維護 *DCB_dmd_flags* 欄位。Layer driver 的 *AEP_CONFIG_DCB* 處理常式會使用當時的值做爲它傳遞給 ISP_INSERT_CALLDOWN 的 demand flags 的一個起點，而不是直接改變 DCB 中的欄位。**表 15-7** 列出對 port drivers 和 VSDs 有意義的 demand flags。

DCB_cmn.DCB_device_flags　這個欄位內含與此 device 有關的 flags。此 DWORD 之中的所有 32 個位元都有定義，但你我只關心列於**表 15-8** 中的那一些。

Demand Flag	說明
DCB_dmd_srb_cdb	這個 flag 表示 I/O packets 需要一個 SCSI request block（SRB）和一個 command description block（CDB）。SCSI'izer 會 "hooks into" calldown stack 以實作此一需求，也因此此需求不應該在高階中再被表現出來。
DCB_dmd_physical	這個 flag 表示一個 driver 需要的是 physical 而非 logical DCB。TSD 會提供此項服務，因此此需求不應該在高階中再被表示出來。事實上，甚至即使這個 flag 沒有設立起來，TSD 還是會放置一個 physical DCB 位址到 IOP 結構中。所以沒有理由一開始就聲明之。
DCB_dmd_small_memory	這個 flag 表示資料緩衝區必須在實際記憶體的前 16 MB 中。當 port driver 處於一部擁有大量記憶體的 ISA bus 電腦中，這個旗標便派上用場，因爲 DMA controller 只認識前 16 MB 位址。BIGMEM.DRV（一個由 Microsoft 提供的 VxD）會實作出此需求。
DCB_dmd_word_align DCB_dmd_dword_align	這個 flag 表示資料緩衝區必須有指定的 alignment。這些需求也與 DMA 有關，而且 BIGMEM.DRV 也能滿足它們。
DCB_dmd_phys_sgd	這個 flag 表示 port driver 視「 scatter/gather descriptor (SGD) 結構所組成的 *IOR_sgd_lin_phys* 陣列是否完成」而定。
DCB_dmd_do_a_b_toggling	這個 flag 被 TSD 設立起來，告訴 volume tracker 追蹤一部單一軟碟機系統中的軟碟機是 drive A 或 drive B.
DCB_dmd_lock_unlock_media	這個 flag 由 port driver 設立，用來表示此一硬體支援「由軟體控制，決定是否使用者可以移走儲存媒體」。這是一個 "announcement"，不是一個 "demand"。
DCB_dmd_load_eject_media	這個 flag 由 port driver 設立，用來表示軟體可以載進或退出（eject）儲存媒體。這也是一個 "announcement"，不是一個 "demand"。
DCB_dmd_serialize	這個 flag 表示一個 driver 要求所有的 requests 在到達之前必須先經過循序排列（serialized）。有一個 IOS 內部元件會處理此一需求，方法是自動將 request 放在 queue 之中。所以此一需求（demand）應該不會在高階中再被表現出來。

表 15-7 Demand flags

Device Flag	說明
DCB_DEV_SPINDOWN_SUPPORTED	這個 flag 被 IDE port driver 設立，表示一個硬碟支援 "spin-down"。
DCB_DEV_SPUN_DOWN	當磁碟似乎是被 “spin down”，這個 flag 即被 IDE port driver 設立。
DCB_DEV_IO_ACTIVE	這個 flag 被 port driver 設立，用來表示它正忙碌。負責 "queuing" 這些 requests 的元件必須在傳遞 request 給 port driver 之前先檢查此一 flag。
DCB_DEV_ASYNC_MED_CHG_SUPPORT	如果 physical device 可以非同步地（經由一個中斷）報告說儲存媒體已經被移走，這個 flag 就會被 port driver 設立。
DCB_DEV_SYNC_MED_CHG_SUPPORT	如果 device 可以同步地（synchronously）報告說儲存媒體已經改變（經由下一次處理的一個錯誤代碼），這個 flag 就會被 port driver 設立。
DCB_DEV_PHYSICAL	這個 flag 表示說，這是一個 "physical" DCB（*DCB_DEV_LOGICAL* 也會被設立）
DCB_DEV_LOGICAL	這個 flag 表示說，這是一個 "logical" DCB（*DCB_DEV_PHYSICAL* 也會被設立）
DCB_DEV_REMOVABLE	這個 device 有可抽取的儲存媒體，此一 flag 即會被 port driver 設立。這個 flag 表示，應該有一個 volume tracker "hook into" calldown stack。
DCB_DEV_WRITEABLE	這個 flag 表示 device 是可寫入的。通常它是在 TSD 檢查了硬體之後，被 TSD 設定。
DCB_DEV_INT13_DRIVE	這個 flag 表示 device 在 Windows 95 啓動之前的真實模式中是可見的。由 TSD 設立。

表 15-8　DCB_device_flags 中的重要 flags

DCB_cmn.DCB_Partition_Start　這個欄位記錄一個 physical sector（logical device 的 partition 即是從那裡開始）。它只被 TSD 維護和使用。

DCB_cmn.DCB_partition_type 這個欄位表示 partition 的型別。它是被 TSD 從 partition 的 type indicator 處（記錄於 master boot record）拷貝而來。最常見的值是 4（16-bit FAT partition）和 6（MS-DOS Large File System partition）。

DCB_cmn.DCB_sig 這個欄位內含 0x4342，也就是字元 "BC"。這個簽名符號幫助你在除錯的時候鑑定 DCBs。對於 assembly 程式員，有一組 IOS 除錯巨集，名稱像是 *AssertDCB* 之類，會檢查像這樣的簽名符號欄位，以幫助判斷指標的合法性。

DCB_cmn.DCB_device_type 這個欄位內含 device type（請看**表 15-9**）。IOS 自動爲 SCSI devices 決定 device type，方法是讀取 SCSI inquiry data。non-SCSI devices 的 port drivers 應該在處理 *AEP_CONFIG_DC* 時設立此一欄位。VSDs 通常會檢查 device type 以便稍後當它們收到 *AEP_CONFIG_DCB* 時決定是否要 "hook into" calldown stack。

Device 型別	數值	說明
DCB_type_disk	0h	通用的不可移除（抽換）的直接存取裝置（例如硬碟）
DCB_type_tape	01h	通用的循序存取裝置
DCB_type_printer	02h	印表機（Printer device）
DCB_type_processor	03h	Processor-type device
DCB_type_worm	04h	可寫一次，可讀多次的光學機器（譯註：例如 CD-R）
DCB_type_cdrom	05h	光碟機（CD-ROM device）
DCB_type_scanner	06h	掃描機（Scanner device）
DCB_type_optical_memory	07h	光學磁碟（optical disc），但不是一般光碟機
DCB_type_changer	08h	Disk changer device
DCB_type_comm	09h	通訊裝置（Communication device）
DCB_type_floppy	0Ah	軟碟機（Floppy disk device）
DCB_type_optical_nec	84h	NEC 5.25 吋光碟機

表 15-9. DCB device types

DCB_max_xfer_len　這個欄位內含資料傳輸最大長度。只有 port driver 可以決定這個值。通常的選擇是 0xFFFFFFFF 和 0x00FFFFFF。這個值表示 port driver 準備處理的最大單一傳輸量。IOS 爲了遵守這個設定，會把大塊資料量打碎爲小塊。

DCB_actual_xxx　所謂的 "actual" 欄位（*DCB_actual_sector_cnt*, *DCB_actual_blk_size*, *DCB_actual_head_cnt*, *DCB_actual_cyl_cnt*, 以及 *DCB_actual_spt*）用來描述一個 device 的真正幾何屬性(actual geometry)。Port driver 可以在處理 *AEP_CONFIG_DCB* 或回應 *IOR_COMPUTE_GEOM* request 時設定這些欄位。Disk TSD 會在初始化期間根據 BIOS tables 來設定這些欄位（除非 port driver 先完成這項工作）並設定 *DCB_TSD_ACTUAL_PRE_SET* flag。

DCB_next_ddb_dcb　這個欄位內含「隸屬相同 controller 的下一個 physical DCB」的位址。一個 port driver 會在處理 *AEP_INITIALIZE* 期間產生一個 DDB。此 DDB 描述一個 adapter，有一個或多個附著的 devices，每一個有其 physical DCB。DDB 中的 *DDB_dcb_ptr* 欄位指向第一個 DCB，這個欄位則被用來指向下一個 DCB。

DCB_dev_node　這個欄位內含此 device 之 Configuration Manager DEVNODE 的位址。你也可以從 DDB 觸及 DEVNODE，但使用這個欄位可能比較方便。

DCB_max_sg_elements　這個欄位被 port driver 在處理 *AEP_CONFIG_DCB* 時設定爲「所能夠處理的 scatter/gather elements 的最大數量」。無論如何其值不會超過 17。任何較高 level 的 driver 可以在處理 *AEP_CONFIG_DCB* 時使此一數值小一點。

DCB_srb_ext_size　這個欄位內含 SCSI request blocks（由 device 產生）中的 miniport extension 的大小。SCSI port driver 查詢 miniport driver 以決定此值，沒有任何其他人需要爲這個欄位操心。

The Input/Output Packet

圖 **15-6** 表現出一個 I/O request packet（IOP）的格式。IOS clients（不管是內部的或外部的）必須負責配置和釋放內含 IOP 的記憶體，但它們是以一種慣例方式來完成。例如，一個存取 physical DCB 的 layer driver 將以此法完成任務：

```
USHORT offset = (USHORT) (dcb->DCB_cmn.DCB_expansion_length
   + FIELDOFFSET(IOP, IOP_ior));
USHORT size = offset + sizeof(IOR)
   + dcb->DCB_max_sg_elements * sizeof(SGD);
PIOP iop = IspCreateIop(size, offset, ISP_M_FL_MUST_SUCCEED);
PIOR ior = &iop->IOP_ior;

[do something that needs an IOP, such as IlbInternalRequest]

IspDeallocMem((PBYTE) iop - dcb->DCB_cmn.DCB_expansion_length);
```

```
typedef struct _IOP {
  ULONG   IOP_physical;             // 00 physical address of IOP
  ULONG   IOP_physical_dcb;         // 04 pointer to physical DCB
  ULONG   IOP_original_dcb;         // 08 pointer to DCB designated by IOR
  USHORT  IOP_timer;                // 0C current timeout value
  USHORT  IOP_timer_orig;           // 0E original timeout value
  ULONG   IOP_calldown_ptr;         // 10 pointer to next calldown routine
  ULONG   IOP_callback_ptr;         // 14 pointer to current callback
                                    //    address
  ULONG   IOP_voltrk_private;       // 18 private to volume tracker
  ULONG   IOP_Thread_Handle;        // 1C owning thread
  ULONG   IOP_srb;                  // 20 SRB address
  ULONG   IOP_reserved[2];          // 24
  IOP_callback_entry IOP_callback_table[IOP_CALLBACK_TABLE_DEPTH];
                                    // 2C the callback stack
  BYTE    IOP_format_head;          // 5C for use in low-level formatting
  BYTE    IOP_format_xfer_rate;     // 5D
  USHORT  IOP_format_track;         // 5E
  ULONG   IOP_format_num_sectors;    // 60
  struct _IOR {
    ULONG   IOR_next;               // 64/00 chaining pointer
    USHORT  IOR_func;               // 68/04 function to perform
    USHORT  IOR_status;             // 6A/06 status of request
    ULONG   IOR_flags;              // 6C/08 flags
    CMDCPLT IOR_callback;           // 70/0C client callback function
```

```
   ULONG   IOR_start_addr[2];       // 74/10 starting sector
   ULONG   IOR_xfer_count;          // 7C/18 bytes or sectors to transfer
   ULONG   IOR_buffer_ptr;          // 80/1C buffer or BlockDev
                                    //   scatter/gather pointer
   ULONG   IOR_private_client;      // 84/20 reserved for use by client
   ULONG   IOR_private_IOS;         // 88/24 reserved for use by IOS
   ULONG   IOR_private_port;        // 8C/28 reserved for use by port
                                    //   driver
   union   urequestor_usage _ureq;  // 90/2C IOCTL parameters
   ULONG   IOR_req_req_handle;      // A4/40 reference data for callback
                                    //   routine
   ULONG   IOR_req_vol_handle;      // A8/44 media handle supplied by
                                    //   requestor
   ULONG   IOR_sgd_lin_phys;        // AC/48 address of SGD array
   UCHAR   IOR_num_sgds;            // B0/4C number of SGDs in array
   UCHAR   IOR_vol_designtr;        // B1/4D drive letter (0 = A, etc.)
   USHORT  IOR_ios_private_1;       // B2/4E padding
   ULONG   IOR_reserved_2[2];       // B4/50
  } IOP_ior;                        //   /58
} IOP, *pIOP;                       // BC
```

圖 15-6　一個 input/output request packet 的格式

外部 IOS client 以現有之 VRP 做爲一個合適的 volume 的起始。從那裡，整個過程類似一個內部 client 的作爲。唯一不同的是，外部 client 只單獨配合「內嵌於 IOP 中之 IOR」一起工作：

```
USHORT size = vrp->VRP_max_req_size
    + vrp->VRP_max_sgd * sizeof(SGD);
USHORT offset = (USHORT) vrp->VRP_delta_to_ior;
PIOR ior = IspCreateIor(size, offset, ISP_M_FL_MUST_SUCCEED);

[do something that needs an IOR, such as IOS_SendCommand]

IspDeallocMem((PBYTE) ior - offset);
```

不管哪一種情況，發出請求者使用的是 *ILB_service_rtn* service 的 *ISP_CREATE_IOP* 子函式，它會傳回一塊未被初始化的記憶體的位址。如果你直接呼叫 service，你得在傳回的指標上面增加 expansion area 大小，以取得 IOP。我的外包函式（wrapper functions）（*IspCreateIop* 和 *IspCreateIor*，定義於 IOSDCLS.H）會在回返之前執行這些加法動作，

這正是爲什麼你沒有在這些程式片段中看到那些混亂局面的原因。**圖 15-7** 顯示內含一個 I/O request 的記憶體的大體佈局。

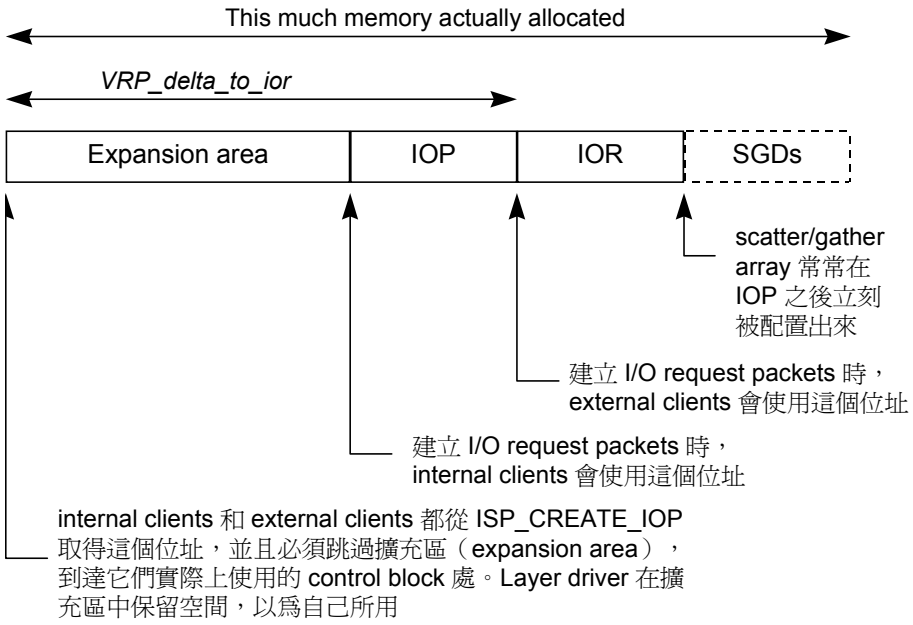


圖 15-7 一個 IOP 中的記憶體是如何被使用的

IOP 擴充區（expansion area）是一塊記憶體，位在 IOP 結構之前。當 Layer drivers 呼叫 *ISP_INSERT_CALLDOWN* 準備把它們自己安插到 calldown stack 時，它們會告訴 IOS 說它們需要多少 expansion data。IOS 會把需求加總起來，並將總額記錄在 *DCB_expansion_length* 之中。IOS clients 最終會在配置新的 IOPs 時保留 expansion area 的記憶體。Layer drivers 的 request routines 使用它們各自的 calldown list entries 來定出它們自己的 expansion areas 的偏移量（從 IOP 起始處算起，負值）。

以下列表提供 IOP 和 IOR 之中的重要欄位相關資訊。

IOP_physical 這個欄位內含 IOP 的實際位址。此欄位截至目前尚未知其用途。

IOP_physical_dcb　這個欄位內含與 request 相關之 physical DCB 的位址。在 TSD 之下的 layer drivers 將會發現，不管它們是否聲明了 *DCB_dmd_physical* demand，這個欄位永遠會被完成。發出 internal requests 並使用 internal criteria 函式的 layer drivers 必須在呼叫 criteria 函式之前完成 IOP 的這個欄位和下一個欄位，否則會使系統當機。

IOP_original_dcb　這個欄位內含由 IOS client 傳遞過來的 DCB 位址，那是 volume（request 發生地）的 logical DCB。 發出 internal requests 並使用 internal criteria 函式的 layer drivers 必須在呼叫 criteria 函式之前完成 IOP 的這個欄位和前一個欄位，否則會使系統當機。

IOP_timer　這個欄位內含以半秒爲單位的數值，一直持續到 IOS 所允許的 7.5 秒到期爲止（過了 7.5 秒表示此一 request 可能出了麻煩）。IOS 每半秒鐘即將此欄位減 1。Layer drivers 可能會爲了重新開始一個 request，或因爲身爲 *AEP_IOP_TIMEOUT* event 的一部份，而更改此欄位。

IOP_timer_orig　這個欄位內含「IOS 放進 *IOP_timer* 的初值」。如果你要給予一個 timed-out request 第二次機會的話，這個值就會被重新設到 *IOP_timer* 欄位中。

IOP_calldown_ptr　這個欄位內含指標，指向目前的 calldown stack entry，那是一個 *DCB_cd_entry* 結構，像這樣：

```
typedef struct _DCB_cd_entry {
  PVOID   DCB_cd_io_address;        // 00 address of request routine
  ULONG   DCB_cd_flags;             // 04 demand bits for this layer
  ULONG   DCB_cd_ddb;               // 08 driver's DDB pointer
  ULONG   DCB_cd_next;              // 0C pointer to next cd entry
  USHORT  DCB_cd_expan_off;         // 10 offset of expansion area
  UCHAR   DCB_cd_layer_flags;       // 12 flags for layer's use
  UCHAR   DCB_cd_lgn;               // 13 load group number
} DCB_cd_entry, *pDCB_cd_entry;     // 14
```

一個 layer driver 會使用並修改 calldown pointer，向下呼叫下一個 layer，例如：

```
void __declspec(naked) DoCallDown(PIOP iop)
   {                          // DoCallDown
   _asm
      {                       // call down to next layer
      mov ecx, [esp+4]
      mov eax, [ecx]IOP.IOP_calldown_ptr
      mov eax, [eax]DCB_cd_entry.DCB_cd_next
      mov [ecx]IOP.IOP_calldown_ptr, eax
      jmp [eax]DCB_cd_entry.DCB_cd_io_address
      }                       // call down to next layer
   }                          // DoCallDown
```

注意　在 Windows 95 DDK 的 "Layered Block Device Drivers" 一節之中，有些碼表現出一種不同的 calldown 方法：把一個常數加到目前的 calldown pointer 之中以到達下一個 entry。那些碼是不正確的，你必須像上面程式碼那樣，將一個 calldown entry 串鏈到下一個 calldown entry。

IOP_callback_ptr　這個欄位內含指標，指向目前的 callback stack entry。每一個 entry 看起來像這樣：

```
typedef struct IOP_callback_entry {
  ULONG  IOP_CB_address;       // 00 callback address
  ULONG  IOP_CB_ref_data;      // 04 reference data for callback
} IOP_callback_entry;
```

IOP 本身內含一個以此結構組成的陣列（*IOP_callback_table*），而此指標應該指向陣列中的一個元素。當一個 layer driver 處理完一個 I/O request，它並不只是回返其呼叫端而已。畢竟，這個 request 可能已經被 "queued" 了一段時間，而中間的呼叫者可能是一個 VMM event function 或是其他一些「對 I/O 動作的結果不感興趣」的代理程式。因此，driver 會 "calls the request back"（這是 IOS 官方說明文件的用語），方法是對 callback stack 做取出（popping）動作，然後呼叫此時位於 stack 頂端的函式。雖然有 "reference data" 在 callback entry 之中，callback 函式還是只有一個參數（一個 IOP 指標）並且沒有傳回任何值。下面是「執行一個 callback」的實例：

```
void __declspec(naked) DoCallBack(PIOP iop)
    {                       // DoCallBack
    _asm
        {                   // call back to previous layer
        mov ecx, [esp+4]
        sub [ecx]IOP.IOP_callback_ptr, size IOP_callback_entry
        mov eax, [ecx]IOP.IOP_callback_ptr
        jmp [eax]IOP_callback_entry.IOP_CB_address
        }                   // call back to previous layer
    }                       // DoCallBack
```

stack 之中永遠有至少一個「IOS 提供的 callback 函式」，使 layer driver 不可能意外地跳到萬劫不復之地。如果你希望你的 driver 被 called back，當它的下層 layer 完成了這個 request，你必須 "push" 一個 entry 到 callback stack 之中：

```
void InsertCallBack(PIOP iop, VOID (*callback)(PIOP),
    ULONG refdata)
    {                       // InsertCallBack
    IOP_callback_entry* cbp = (IOP_callback_entry*)
        iop->IOP_callback_ptr;
    ASSERT(cbp >= iop->IOP_callback_table
        && cbp < iop->IOP_callback_table
        + arraysize(iop->IOP_callback_table));
    cbp->IOP_CB_address = (ULONG) callback;
    cbp->IOP_CB_ref_data = refdata;
    iop->IOP_callback_ptr += sizeof(IOP_callback_entry);
    }                       // InsertCallBack
```

如果你打算處理你所儲存的 reference data，你必須自己設法取出 callback 結構。

IOR_next　這是一個 linking 欄位，被任何想要把 IOR 放進 queue 中的人使用。Client 應該將此值設初值爲 0。但如果我是你，我不會假設此值曾被初始化過。

IOR_func　這個欄位內含需要執行的 I/O 動作（表 **15-10**）。Client 程式會填寫它。Layer drivers 會在其 request 函式中檢閱它，看看是否需要處理特殊需求。

I/O Function Code	說明
IOR_READ	從 device 中讀取資料
IOR_WRITE	寫資料到 device 去
IOR_VERIFY	對寫入 device 的資料做驗證
IOR_WRITEV	寫資料到有驗證程序的 device 中
IOR_MEDIA_CHECK_RESET	檢查看看媒體（碟片）是否已經改變
IOR_LOAD_MEDIA	載入可抽換（removable）之媒體（碟片）
IOR_EJECT_MEDIA	退出（ejects）可抽換之媒體（碟片）
IOR_LOCK_MEDIA	鎖住 drive 中的可抽換媒體（增加計數器值，此值與 UNLOCK_MEDIA 共享）。
IOR_UNLOCK_MEDIA	將 drive 中的可搬移媒體解鎖（減少計數器值，此值與 LOCK_MEDIA 共享）。
IOR_REQUEST_SENSE	從 device 中讀取 "sense data"（SCSI 才有）
IOR_COMPUTE_GEOM	重新評估 volume 和 device 的特性
IOR_GEN_IOCTL	執行一個一般的 I/O 控制動作
IOR_FORMAT	發出一個低階格式化的 packet
IOR_SCSI_PASS_THROUGH	將一個 SCSI 命令原封不動地交給一個 SCSI port driver
IOR_CLEAR_QUEUE	讓一個 queue 流乾（也就是說，等待它自然成空）
IOR_DOS_RESET	在一個 INT 13h reset request 之後重設（reset）device (目前只被 HSFLOP 使用）
IOR_SCSI_REQUEST	送出一個 SRB 到一個 SCSI device 手上（起源於 layer drivers）
IOR_SET_WRITE_STATUS	依據實際媒體送出 DCB_DEV_WRITEABLE flag（目前只用於 HSFLOP）
IOR_RESTART_QUEUE	被用於一個 IOR 中，目的是在 request queue 中做一個標記。只能被 IOS 內部組件使用
IOR_ABORT_QUEUE	將 request queue 清空（flush）
IOR_SPIN_DOWN	"Spin down" 磁碟機
IOR_SPIN_UP	"Spin up" 磁碟機

IOR_FLUSH_DRIVE	強迫取出佇列中的資料（queued data）
IOR_FLUSH_DRIVE_AND_DISCARD	清空（flush）並丟棄佇列中的資料
IOR_FSD_EXT	送出一個私人命令給一個 FSD。較低的 levels 應該絕不會收到。

表 15-10　IOR function codes

IOR_status　這個欄位內含狀態碼。*IORS_ERROR_DESIGNTR*（亦即 16）以下的值表示動作成功，而大於或等於 *IORS_ERROR_DESIGNTR* 表示有錯誤發生。「動作成功」的代碼包括三種：*IORS_SUCCESS* (0), *IORS_SUCCESS_WITH_RETRY* (1), *IORS_SUCCESS_WITH_ ECC* (2)。3~15 沒有被使用。

IOR_flags　在一個 I/O request 範圍內，只有少數 flag 位元對於 clients 或 VSDs 或 port drivers 有重要性。*IORF_CHAR_COMMAND* 用來告訴 port driver，request 中的資料計數是以 bytes 爲單位，否則是以 device 的 block size 爲單位（通常是 512）。如果要的話，clients 可以設定 *IORF_DATA_IN* 和 *IORF_DATA_OUT*。這些 flags 只有在 BIGMEM.DRV 無法告知某一動作中的資料移動方向時，才有重要性，它們允許 BIGMEM.DRV 僅執行必要的拷貝動作。

Clients 會設定 *IORF_DOUBLE_BUFFER* 以要求 IOS 將 client 的資料拷貝進出於 IOS 內部緩衝區中，並執行「會用到那些內部緩衝區」的動作。Clients 也可以設定 *IORF_HIGH_PRIORITY*，將一個 request 移到任何內部 queues 的起頭處 -- 雖然通常 layer drivers 比較常要求的是 internal request 的優先權。IOS 自己的 queue manager 會禮遇這個 flag，所以 layer drivers 不需要檢查它。

當它發生，大部份 IOS clients 會提供一個 physical sector 位址，作法是爲 partition 偏移值（bias，也就是一個 partition 起始的 physical sector 號碼）加上一個 logical sector 位址（表示它們所定址的 volume）。如果你想提供一個 logical 位址取代之，你必須設定 *IORF_LOGICAL_START_SECTOR* flag。TSD 將加到適當的 partition bias 一次，並設立 *IORF_PARTITION_BIAS_ADDED* flag 以阻止再做一次（由於再一次 request 的緣故）。

一個 client 可以設定 *IORF_QUIET_VOLTRK* 來阻止 volume tracker 警告使用者說有不正確的媒體。檔案系統會在「預讀資料以便填入其 caches」時使用這個 flag，因爲如果預讀的資料並非真的被需要，干擾使用者是不正確的。

如果 *IOR_buffer_ptr* 指向一個以 scatter/gather descriptors 組成的陣列，client 便設定 *IORF_SCATTER_GATHER*。否則，*IOR_buffer_ptr* 指向一個資料緩衝區。

IORF_SYNC_COMMAND 表示有一個同步命令（synchronous command）。IOS 會等待，直到一個同步命令完成，才回返至提出要求的 client 端。如果這個 flag 被清除，IOS 會將外界的 requests 放進 queue 中排隊，然後立刻回返。當此命令稍後被完成，它也會呼叫 *IOR_callback* 函式。大部份 IOS clients 只會產生非同步（asynchronous）requests 並抑制它們自己的程式碼，直到事件完成。因此這個 flag 在 IOS 之外被用得很節制。

Clients 會設立 *IORF_VERSION_002* 以表示它們正在供應一個 IOR 結構。在一個 *BlockDev_Command_Block*（BCB）結構之中，對應的 offset（亦即 8）的對應位元（亦即 400h）是 0。如果 *IOS_SendCommand* 發現這個 flag 被設立，它會忽略從 EDI 暫存器傳遞來的參數（其中持有一個 DCB 的位址）。這種情況下，IOS 使用 *IOR_vol_designtr* 來找出正確的 DCB。

IOR_callback 這個欄位指向 IOS 應該呼叫的 completion routine（當一個非同步命令完成時）。此函式的型態如下：

```
void __cdecl callback(DWORD refdata);
```

由 IOS 供應給此 callback 函式的 "reference data" 是 client 放進 IOR 之 *IOR_req_req_handle* 欄位的東西。Layer drivers 不使用這個 callback pointer。它們使用 callback stack（由 *IOP_callback_ptr* 指向）來回呼一個 request。*IOR_callback* 則是給 *IOS_SendCommand* 或某些其他的 top-level IOS 函式使用。

IOR_start_addr 這個欄位內含 request 中的起始 sector 位址。這是一個 64 位元整數，佔用兩個 DWORDS。第一個 DWORD 持有位址的較低部份，第二個 DWORD 持

有位址的較高部份。如果 *IORF_LOGICAL_START_SECTOR* 被設立，這個位址將是相對於 partition 的起始處。否則，它將相對於 physical surface 的起始處。

IOR_xfer_count　這個欄位內含搬移數量，單位是 bytes（如果 *IORF_CHAR_COMMAND* 設立）或 sectors（如果 *IORF_CHAR_COMMAND* 清除）。如果 *IORF_SCATTER_GATHER* 被設立，這個欄位就是多餘的，因爲你可以將每一個 scatter/gather descriptor 的數量加總，來推測總量 。這個欄位必須由 client 填寫，因爲有些程式會使用它。

IOR_buffer_ptr　如果 *IORF_SCATTER_GATHER* 被設定，*IOR_buffer_ptr* 是一個「由 BlockDev scatter/gather descriptors 所組成」之陣列的線性位址。陣列中的每一個元素有如下格式：

```
typedef struct BlockDev_Scatter_Gather {
  ULONG BD_SG_Count;      // 0 number of bytes or sectors
  ULONG BD_SG_Buffer_Ptr;  // 4 address of buffer
} _BlockDev_Scatter_Gather;
```

注意，此結構的成員和 IOS 內部所用的 SGD 結構有著相反的次序。在 scatter/gather descriptors 中的緩衝區指標總是線性位址，然而如果 *IORF_SCATTER_GATHER* 被清除，*IOR_buffer_ptr* 就是一塊資料緩衝區的線性位址。

IOR_private_xxx　*IOR_private_client*、*IOR_private_IOS* 和 *IOR_private_port* 欄位分別內含 client 程式、IOS、以及 port driver 的私有資料。Microsoft 的 drivers 常常使用 *IOR_private_client* 欄位，當做一個方便地方，儲存 IOR 的偏移位置（從內嵌它的那個記憶體區塊開始起算）。當釋放一段被一個 IOR 所使用的記憶體時，它們使用這個欄位，例如：

```
ior->IOR_private_client = vrp->VRP_delta_to_ior;
...
IspDeallocMem((PVOID) ((DWORD) ior - ior->IOR_private_client));
```

_ureq　這個欄位內含一個 IOCTL request 的參數

IOR_req_req_handle 這個欄位內含 32 位元任意值，那是要被當做唯一參數，傳遞給 IOR_callback 函式用的（當一個 asynchronous request 完成）。

IOR_req_vol_handle 這個欄位會被 client 設定爲想要獲得的 VRP 的位址，並且被 volume tracker 用來驗證正確的儲存媒體的確存在。

IOR_sgd_lin_phys 這個欄位內含由 SGDs 所組成的一個陣列的線性位址。每一個 SGD 有以下格式：

```
typedef struct _SGD {
ULONG SG_buff_ptr;      // 0 physical address of buffer
ULONG SG_buff_size;     // 4 size of the buffer in bytes
} SGD, *PSGD;
```

不要受到此欄位名稱中的 *lin_phys* 的暗示所影響，事實上，SGD 結構中的指標總是 physical 位址。所謂的 "criteria" 函式將會建立起 SGD 陣列並將傳輸個數從 sectors 轉換爲 bytes -- 如果 port driver 要求 *DCB_dmd_phys_sgd* demand 的話。

有一個名爲 *DCB_dmd_phys_sgd_ptr* 的 demand flag，想像上會使得 *IOR_sgd_lin_phys* 位址成爲一個 physical 位址而非 linear 位址。physical 位址對於一個硬體 DMA controller 很有用，可以直接支援 scatter/gather 動作。不過，我得說，這個 demand 目前尚未被支援。

不要重蹈我的覆轍。我最初對於由此欄位所指出的 SGD 結構和由 *IOR_buffer_ptr* 所指出的 *_BlockDev_Scatter_Gather* 結構有點混淆。不只是這兩個結構的欄位有不相容的次序，而且這個結構使用 physical 位址，而 BlockDev 結構使用 linear 位址。

IOR_num_sgds 這個欄位被 criteria 函式設定，使它等於由 SGDs 所組成的 *IOR_sgd_lin_phys* 陣列的元素個數。

IOR_vol_designtr 這個欄位被 client 設定，代表目標物（一個 volume）的 logical drive 索引值（0 表示 A，依此類推）。

The Volume Request Parameters Block

IOS 使用一個所謂的 volume request parameters（VRP）結構來描述被裝載 （mounted）到某個特定 device 上的一個 volume（圖 **15-8**）。

```
typedef struct _VRP {
  ULONG   VRP_demand_flags;         // 00 demand flags
  ULONG   VRP_event_flags;          // 04 even flags
  USHORT  VRP_max_sgd;              // 08 maximum number of SGDs supported by
                                    //    port driver
  USHORT  VRP_max_req_size;         // 0A size of I/O request packets
  ULONG   VRP_delta_o_ior;          // 0C offset of IO from start of packet
                                    //    memory
  ULONG   VRP_block_size;           // 10 sector size on this device
  ULONG   VRP_fsd_hvol;             // 14 FSD's handle for this volume
  ULONG   VRP_fsd_entry;            // 18 FSD's IFS request handler
  ULONG   VRP_device_handle;        // 1C BDD address
  ULONG   VRP_partition_offset;     // 20 physical sector where partition starts
  ULONG   VRP_next_vrp;             // 24 internal VRP chaining field
  ULONG   VRP_logical_handle;       // 28 DCB for logical device
  ULONG   VRP_reserved;             // 2C (reserved)
} VRP, *PVRP;                       // 30
```

圖 15-8　一個 volume request parameter block 的格式

IOS 產生一個 VRP 以回應來自 IFS Manager 的 *IOR_Requestor_Service* 呼叫。IFS 和各式各樣的 file system drivers 把它到處傳遞，視之爲動作目標（儲存媒體）的一個 handle。這個結構也持有足可左右「外部 client 如何建立 I/O requests」的一些數值。VRP 中的欄位討論於下：

VRP_demand_flags　這個欄位內含的 flags 用來指示 client 必須滿足（藉由呼叫外部 criteria 函式）的 demands。欄位中的位元和 *DCB_demand_flags* 中的位元有相同意義，但是 VRP 中的 demands 並不一定必須與對應的 DCB 中的欄位完全一致。舉個例子，在我的系統中，CD-ROM 的 DCB 表示出 *DCB_dmd_load_eject_media*，但 VRP 表示的卻是 *DMD_dmd_query_remov*。

VRP_event_flags　這個欄位內含的 flags 用以警告 client 一些事實（表 **15-11**）。

當然，只有 *VRP_ef_media_changed* 和 *VRP_ef_media_uncertain* flags 對 client 有用。

Flag	說明
VRP_ef_media_changed	device 上的儲存媒體已經被改變（自從最近的 I/O 動作之後）
VRP_ef_media_uncertain	儲存媒體可能已被改變
VRP_ef_prompting	volume tracker 目前正提示使用者（重新）安插「內含此一 volume」的儲存媒體。
VRP_ef_input_share	字元的輸入流裝置（input stream device）是可共享的
VRP_ef_output_share	字元的輸出流裝置（output stream device）是可共享的
VRP_ef_user_canceled	使用者取消了 volume tracker 的 request，以（重新）安插 volume
VRP_ef_write_protected	這個 volume 禁止寫入
VRP_ef_real_mode_mapped	這個 volume 目前正被一個真實模式 driver 而非 32-bit driver 處理
VRP_ef_ios_locked	內含此 volume 的那個 device 已經被鎖住了

表 15-11 VRP_event_flags flags

VRP_max_sgd 這個欄位內含一個 I/O request packet 所需的 scatter/gather descriptors 的最大個數。Client 要不使用此值，要不就使用 17（看哪一個比較小）來決定一個新的 I/O packet 的大小：

```
USHORT numsgd = min(vrp->VRP_max_sgd, 17);
USHORT size = vrp->VRP_max_req_size + numsgd * sizeof(SGD);
```

Scatter/gather descriptors 並不需要佔用 I/O packet 記憶體的一部份，但它們的確需要佔用一個 locked page 以符合 DMA controller 的要求。因此，如果只是爲這些 descriptors 在 IOP 尾端湊和一些空間，對任何人來說都是比較容易的。

VRP_max_req_size 這個欄位內含每一個 I/O request packet 所需的 bytes 個數，並

用以管理「內含此 volume」之 device。每一個 device 只有一個大小，在 calldown stack 完成之後決定。所以我並不瞭解這個欄位名稱中的 "max" 意指爲何。這個大小包括 IOP expansion area, IOP, 以及 embedded IOR，但不包括 scatter/gather descriptors 所需空間。

VRP_delta_to_ior　這個欄位內含 IOR 的偏移位置（從記憶體區塊的頭部起算）。記憶體區塊內含一個 I/O packet，此 packet 是爲「內含此一 volume」之 device 所準備。**圖 15-7** 顯示像這樣的一塊記憶體長什麼樣子。"delta" （差值）讓你通過 expansion area 和 IOP，到達 IOR 的起始處。Clients 利用這個值來定出一個 request packet 的 IOR 部份，而不需要知道 expansion area 和 IOP portions 有多大。

VRP_block_size　這個欄位內含此一 volume 的每一個 sector 的大小（單位是 bytes）。通常其值爲 512。它必須是 2 的冪次方，因爲需要「將傳輸數量轉換爲 bytes」的程式碼（例如 criteria 函式）使用 *DCB_apparent_blk_shift* 值做爲移位位元數。但我可以和你打賭：Windows 95 的所有元件可以穩健地處理非標準的大小。

VRP_fsd_hvol　這個欄位內含一個 handle，透過它，client VxD 才知曉此一 volume。此 handle 所指之結構格式由不同的 client 決定。

VRP_fsd_entry　這個欄位內含 client file system driver 中的 *FS_MountVolume* 函式位址。IOS 元件如 volume tracker 會直接呼叫那個函式，一如下一章所討論。

VRP_device_handle　這個欄位內含一個 BDD 的位址，此 BDD 描述「內含此一 volume」之 device。這個位址將指入此一 device 之 physical DCB 中的 C3h bytes 處。

VRP_partition_offset　這個欄位內含 physical volume 中的 logical sector 0 的 sector 號碼。雖然 client 有可能傳遞一個 logical sector 號碼到 IOS 手中，當做一個起始位址，但大部份 clients 真正會做的是親自從 logical 轉換爲 physical sectors：加上 partition offset。這種詭異行爲的理由是，許多 clients 呼叫 *IOS_SendCommand* 時夾帶的是 BDD 位址，而非 logical DCB。由於 BDD 描述一個 physical device，IOS 沒辦法輕易地回頭找出 partition 起始位址。

VRP_next_vrp 這個欄位內含下一個 VRP 的位址。IOS 使用此一欄位來維護一個 VRPs 串列，串列由所有 volumes（它們被 "mounted" 到任何 device）構成。

VRP_logical_handle 這個欄位內含 DCB 位址，DCB 描述「內含此一 volume」之 logical device。

處理 Asynchronous Events

IOS 使用 asynchronous events 來初始化和結束 layer drivers，並且在系統運行期間警告它們有一些 events 發生。你的 VxD 將擁有一個 asynchronous event 處理函式。這個函式可以在中斷時間被呼叫，所以它必須處於 locked code segment。它是一個 void 函式，只需要一個參數，是一個指向 asynchronous event packet（AEP）的指標：

```
void OnAsyncRequest(PAEP aep)
     {
     // do something based on aep->AEP_func
     // store result of doing it in aep->AEP_result
     }
```

和其他被 IOS 所使用的 packet 結構一樣，AEP 事實上是一大家族之結構的父源，它們共享一個共同的表頭：

```
typedef struct AEPHDR {
  USHORT  AEP_func;       // 00 function code
  USHORT  AEP_result;     // 02 result
  ULONG   AEP_ddb;        // 04 pointer to DDB
  UCHAR   AEP_lgn;        // 08 current load-group number
  UCHAR   AEP_align[3];   // 09 alignment
} AEP, *PAEP;             // 0C
```

由於 AEP 結構中的函式代碼有 24 個可能，所以通常會有 24 個子結構，其中的共同表頭有著不同的名稱，以及不同的內部修飾名稱，例如在 AEP_NITIALIZE 結構中有 *AEP_bi_i* 字首。這些結構的繁殖以及其中類似辭典編纂的成員名稱，正是爲什麼先前我說，當你撰寫一個 VxD 準備和 IOS 配合工作時，你必須有「打許多字」的心理準備的原因。

雖然有些 asynchronous event 處理函式必須置於 locked code 之中，但其他大部份並不需要如此。因此我可以把我的 *OnAsyncRequest* 函式這樣完成：

```
VOID OnAsyncRequest(PAEP aep)
    {                          // OnAsyncRequest
    typedef USHORT (*PEF)(PAEP);
    static PEF evproc[AEP_MAX_FUNC+1] =
        {(PEF) OnInitialize     //  0 AEP_INITIALIZE
        ,NULL                   //  1 AEP_SYSTEM_CRIT_SHUTDOWN
        ,(PEF) OnBootComplete   //  2 AEP_BOOT_COMPLETE
        ...
        ,NULL                   // 23 AEP_CHANGE_RPM
        };
    PEF proc;

    ASSERT(aep->AEP_func < arraysize(evproc));
    if (aep->AEP_func < arraysize(evproc)
        && (proc = evproc[aep->AEP_func]))
        aep->AEP_result = proc(aep);
    else
        aep->AEP_result = (USHORT) AEP_FAILURE;
    }                          // OnAsyncRequest
```

也就是說，我爲每一個「我想處理的 asynchronous events」寫了一個函式，並在此函式中將每一個 event 派送（dispatch）出去。我讓各個 event 函式傳回一個結果，俾得以避免下面這種繁複的裝飾行爲：

```
void OnInitialize(PAEP_bi_init aep)
    {
    ...
    aep->AEP_bi_i_hdr.AEP_result = AEP_SUCCESS;
    }
```

而以下面這樣的程式碼取代：

```
USHORT OnInitialize(PAEP_bi_init aep)
    {
    ...
    return AEP_SUCCESS;
    }
```

當然，還有一些荒唐行爲，因爲 *AEP_FAILURE* 錯誤代碼（提醒你，它是一個 USHORT 欄位）被定義爲 -1，這差不多保證編譯器一定會發出抱怨。

當 IOS 決定「如何」以及「何時」送出 asynchronous event notifications 時，它會分辨三種 drivers。"**Noncompliant**" drivers 包括 IFS、FSD 以及 Windows NT port drivers（layers 0, 2, 21）；它們並不被認爲是「註冊過」，所以也絕不會收到 asynchronous events。"**Port**" drivers 包括 layers 19~30 之間（但 21 除外）的所有 layer drivers。"**Generic**" drivers 則涵蓋其他每一個 layer drivers。一般而言，如你所見，port drivers 負責特定的硬體，generic drivers 是個愛管閒事者，可能會和任何（所有）的 devices 有關。

以下篇幅描述 AEP events：

AEP_1_SEC IOS 送出一個 *AEP_1_SEC* event 通知你說：自從前一次 *AEP_1_SEC* event 之後，已經過去了 1 秒。你可以在呼叫 *IOS_Register* 之前先在 *DRP_feature_code* 欄位中設定 *DRP_FC_1_SEC* flag，以索求這些 notifications。但甚至即使你要求 1 秒的 notifications，你收到的卻是每半秒一次，因爲在 DRP.H 之中，*DRP_FC_1_SEC* flag 和 *DRP_FC_HALF_SEC* flag 的數值相同。你可以根據一個事實來評價這個 bug 的重要性：自從 IOS 開始以來這個 bug 就存在了。

AEP_2_SECS IOS 送出一個 *AEP_2_SEC* event 通知你：自從最近一次 *AEP_2_SEC* event 之後，已經過去了 2 秒。你可以在呼叫 *IOS_Register* 之前先在 *DRP_feature_code* 欄位中設定 *DRP_FC_2_SECS* flag，以索求這些 notifications。

AEP_4_SECS IOS 送出一個 *AEP_4_SEC* event 通知你：自從最近一次 *AEP_4_SEC* event 之後，已經過去了 4 秒。你可以在呼叫 *IOS_Register* 之前先在 *DRP_feature_code* 欄位中設定 *DRP_FC_4_SECS* flag，以索求這些 notifications。

AEP_ASSOCIATE_DCB TSD 會初始化這個 event 以發掘任何額外的 volumes。這些 volumes 雖然隸屬於某個 physical device，卻無法自我演繹而得。Event 函式會收到一個 *AEP_assoc_dcb* packet:

```
typedef struct AEP_assoc_dcb {
  struct AEPHDR      AEP_a_d_hdr;    // 00 standard header
  PVOID              AEP_a_d_pdcb;   // 0C physical DCB address
  ULONG              AEP_a_d_drives; // 10 bit map of associated drives
} AEP_assoc_dcb, *PAEP_assoc_dcb;
```

Layer driver 如果有額外的資訊是要給 TSD 看的，應該修改 *AEP_a_d_drives* bit map。假設你的 driver 是一個 FSD，負責壓縮由 *AEP_a_d_pdcb* DCB 所表現的 physical drive，而你知道 Q drive 是你的 drive 之一。你應該將 bit mask 中的 bit 16 設立起來（bit 0 對應於 drive A 而 Q 是第 17 個字母）。

AEP_BOOT_COMPLETE　當 IOS 完成一系列的動作而這些動作可能增加或去除系統中的 DCBs，IOS 便送出一個 *AEP_BOOT_COMPLETE* event。Event 函式會收到一個 *AEP_boot_done* packet，其中只內含標準的 AEP 結構表頭。Driver 應該決定是否它對所有的 devices 仍然負有責任，如果是，它應該傳回 *AEP_SUCCESS*；如果不是，應該傳回 *AEP_FAILURE*。

Layer driver 應該維護一個 static 變數，用來記錄 driver 要負責的 DCBs 個數。*AEP_CONFIG_DCB* 處理常式會累加此一變數，而 *AEP_UNCONFIG_DCB* 處理常式則會遞減此一變數。*AEP_BOOT_COMPLETE* 處理常式的工作於是就很簡單了：

```
static int ndevices = 0;
...
USHORT OnBootComplete(PAEP_boot_done aep)
    {                       // OnBootComplete
    return ndevices ? AEP_SUCCESS : AEP_FAILURE;
    }                       // OnBootComplete
```

AEP_CONFIG_DCB　IOS 送出 *AEP_CONFIG_DCB* event 給一個 layer driver，以設定一個 DCB。只有對於代表 physical devices 且爲 port driver 之直接責任對象的那些 DCBs，port drivers 才會收到這個 event。至於 **generic** drivers 則會爲系統中的每一個 DCB 收到此 event。Event 函式會收到一個 *AEP_dcb_config* packet：

```
typedef struct AEP_dcb_config {
  struct AEPHDR AEP_d_c_hdr;   // 00 standard header
  ULONG  AEP_d_c_dcb;          // 04 DCB address
} AEP_dcb_config, *PAEP_dcb_config;
```

在這裡，*AEP_d_c_dcb* 是正被組態（configuring）的一個 DCB 的位址。

一個 **port** driver 應該填寫 device 的 "geometry"（幾何屬性）以及其他與 device 相關的資訊。例如，在我的 RAM-disk driver（範例碼在書附碟片的 \CHAP15\PORTDRIVER 目錄中），我決定產生一個 2-MB drive，擁有一個 cylinder、一個 track、4096 個 sectors、每一個 sector 有 512 bytes：

```
dcb->DCB_cmn.DCB_device_type = DCB_type_disk;
dcb->DCB_cmn.DCB_device_flags |= DCB_DEV_WRITEABLE;

dcb->DCB_max_xfer_len = 0xFFFFFFFF;
dcb->DCB_actual_sector_cnt[0] = 4096;
dcb->DCB_actual_sector_cnt[1] = 0;
dcb->DCB_actual_blk_size = 512;
dcb->DCB_actual_head_cnt = 1;
dcb->DCB_actual_cyl_cnt = 1;
dcb->DCB_actual_spt = 4096;
dcb->DCB_cmn.DCB_TSD_Flags |= DCB_TSD_ACTUAL_PRE_SET;
```

port driver 可能也會想要設立其他的 device flags，例如 *DCB_DEV_SPINDOWN_SUPPORTED*, *DCB_DEV_SPUN_DOWN*, *DCB_DEV_ASYNC_MED_CHG_SUPPORT*, *DCB_DEV_SYNC_MED_CHG_SUPPORT* 和 *DCB_DEV_ REMOVABLE*。如果你的極限小於預設值 17，你也可以設立 *DCB_max_sg_elements*。

對於「可將儲存媒體移除（抽換）」之裝置的一份警告　爲了確保不會把資料寫入錯誤的 volume 之中，Microsoft 的 file system drivers 做了某些動作，而我認爲這對於可抽取之儲存媒體而言，很有問題。每當偵測到 device 中出現一個新的 read/write volume，它們就在磁碟的 boot sector 中重寫廠商識別碼（8-byte）。而校正識別碼（咸被認爲獨一無二）是以 CHI 字元開始，再加上根據時間、日期以及其他因素隨機取得的 5 個 bytes。寫下一個獨一無二的識別碼，目的是爲讓 Windows 95 分辨大量生產的軟碟（floppy

disk），因爲它們沒有獨一無二的序號。如果你撰寫的是病毒偵測軟體，你只能夠允許讓「boot sector 的改寫動作」用來改變廠商識別碼。但如果你有一個軟體建立於「在一片軟碟的 boot sector 上尋找廠商識別碼（如 "MSDOS5.0"）」的基礎上面，這個軟體就完蛋了。的確有一些重要的商業化軟體依賴著類似的廠商識別碼。舉個例子，Stacker（譯註：一種磁碟自動壓縮、解壓軟體）就會以這種方法辨識其被壓縮的 volumes。File system drivers 因此避免改寫記錄於 registry 之 HKLM\System\CurrentControlSet\Control\FileSystem\NoVolTrack 分支上的 boot sectors。

Layer driver 利用 *AEP_CONFIG_DCB* event 做爲時機，藉由呼叫下面的函式，將自己安插到感興趣的 DCB 的 calldown stack 中：

```
BOOL IspInsertCalldown(PDCB dcb, VOID (*calldown)(PIOP),
    PDDB ddb, USHORT expand, DWORD demand, UCHAR loadgroup);
```

在這個函式原型宣告之中，*dcb* 是一個 DCB（physical 或 logical 或兩者），你有興趣在其 calldown stack 中安插一筆項目。calldown 參數是你的 I/O request 處理函式的位址；我將在下一個主要小節中（當我描述如何建立一個 port driver 時）詳細介紹這個非常重要的函式。

ddb 參數是這個 driver 的 DDB 位址；我之所以提這個事實，因爲 **generic** layer drivers 有一個 DDB （其位址應該被用於此處之第三個參數），這和「產生此一 DCB」之 VxD 的 DDB 不同。更進一步說，這個參數指向一個 IOS DDB 結構而非一個 VxD 的 device description block（那常常也被稱爲 DDB）。

expand 參數表示「driver 用來做爲其 expansion area」的 bytes 個數，此 expansion area 將出現在所有「爲此 device 而產生的 IOP packets」中。IOS 將爲 expansion area 計算出一個負的 offset 值，並將之放置於它爲你的 calldown stack entry 所產生的 *DCB_cd_entry* 結構中的 *DCB_cd_expan_off* 欄位內。你的 calldown 函式可以把這個 offset 加到 IOP 位址，以找出一塊區域給它自己使用，一如 *expand* 參數所指定的大小。

你可以在 *demand* 參數中表現出你對於 I/O packet 轉換服務的需求。建構這個參數的方式是，以既存於 *DCB_dmd_flags* 中的 flags 做爲起始。如果你的 driver 可以滿足特別的需求，你可以將對應的位元清除掉。舉個例子，如果你的 driver 是個 SCSI'izer，你應該將你所建立的這個參數之中的 *DCB_dmd_srb_cdb* demand 給清除掉。如果你希望較高的 layer 滿足某些特別的需求，就必須將對應的位元設立起來。

最後一個參數 *loadgroup*，是 layer 的號碼，你應該在該 layer 中安插你的 *calldown* entry。大部份的 layer drivers 會使用 *AEP_lgn* 欄位上的號碼，它總是對應於唯一的 load group bit（設定在 *DRP_LGN* 欄位中）。理論上，你可以在這裡指定任何的 load group 索引號碼。一個以上的 driver 可以被 "hook" 於相同的 level 之上。不過，如果是這樣，IOS 呼叫這些「共享同一個 level」的 drivers 的次序並沒有明確定義出來。

成功的 *ISP_INSERT_CALLDOWN* call 會把一個 *DCB_cd_entry* 結構加進 DCB 的 calldown list 內（加在 *DCB_ptr_cd* 處）並更改 DCB 中的 *DCB_expansion_length* 和 *DCB_dmd_flags* 兩欄位。當所有的 layer drivers 已經經歷了 "hook into" calldown stack 的機會後，留在 DCB 中的是最頂層的 calldown entry 的位址、所有 layer driver expansion-area 大小的總和、以及那些仍未被任何 layer drivers 所滿足的 demand flags。

在我的 RAM-disk driver 中，我使用以下呼叫，將我的 port driver "hook into" calldown stack：

```
if (!(IspInsertCalldown(dcb, OnRequest,
    (PDDB) aep->AEP_d_c_hdr.AEP_ddb, 0,
    dcb->DCB_cmn.DCB_dmd_flags | DCB_dmd_serialize,
    aep->AEP_d_c_hdr.AEP_lgn)))
    return (USHORT) AEP_FAILURE;

++ndevices;
return AEP_SUCCESS;
```

爲了避免操心 "queueing requests" 的問題，我宣佈了 *DCB_dmd_serialize* 這樣一個 demand。上述的 *ndevices* 變數是前數頁中出現的一個 global static 整數，用來記錄我們決定要 "hook into" 多少個 DCBs。這個計數器的目的是爲了告訴 *AEP_BOOT_COMPLETE* 處理常式如何回應。

AEP_CREATE_VRP　IFS Manager 在爲一個新被裝載（mounted）上去的 parent volume 產生一個 VRP 之後，會初始化此一 event。其處理常式會收到一個 *AEP_vrp_create_destroy* packet：

```
typedef struct AEP_vrp_create_destroy {
  struct AEPHDR AEP_v_cd_hdr;  // 00 standard header
  PVOID AEP_v_cd_pvrp;         // 0C address of the VRP
  ULONG AEP_v_cd_drive;        // 10 drive number
} AEP_vrp_create_destroy, *PAEP_vrp_create_destroy;
```

你我所寫的 drivers 可以忽略這個 event，這純粹只做爲資訊傳遞之用。

AEP_DCB_LOCK　當 IOS 正在服務一個 *IRS_QUERY_VOLUME_LOCK* requestor service call 時，會傳出這個 event，用以表示它正在服務一個獨佔的（exclusive）volume lock request。這個 event 的處理常式會收到一個 *AEP_lock_dcb* packet：

```
typedef struct AEP_lock_dcb {
  struct AEPHDR AEP_d_l_hdr;   // 00 standard header
  PVOID AEP_d_l_pdcb;          // 0C logical DCB being locked
  ULONG AEP_d_l_drives;        // 10 bit map of logical drives
  UCHAR AEP_d_l_designtr;      // 14 volume index (0 = A)
  UCHAR AEP_d_l_align[3];      // 15
} AEP_lock_dcb, *PAEP_lock_dcb;
```

driver（例如一個磁碟壓縮驅動程式）如果知道這個事實，就應該更新 *AEP_d_l_drives* bit map 以表示究竟其他哪一些 logical drives 也需要被鎖定（locked）。如果 *AEP_d_l_pdcb* DCB 是壓縮碟 Q 的 host volume，那麼壓縮驅動程式應該 bit map 中的 bit 16 設立起來（因爲 Q 是第 17 個字母）。

此外，任何 driver 如果對那些被鎖定的 DCB 負有責任，就應該注意鎖定的發生，並停止做任何不應該在一個被鎖定的 volume 上發生的事情（例如 caching，快取）。

順帶一提，packet 中的 *AEP_d_l_designtr*，其值與 *IRS_QUERY_VOLUME_LOCK* call 所供應的 "designator" 相同。由於 IOS 已經使用此一數值來尋找適當的 DCB，layer drivers

或許不需要在任何場合用它。

AEP_DESTROY_VRP IFS Manager 在確定「被一個特定 VRP 所表現出來的 volume」已經被 "dismounted"（卸載）之後，便會發出這個 event。Event 處理常式會收到一個 *AEP_vrp_create_destroy* packet：

```
typedef struct AEP_vrp_create_destroy {
  struct AEPHDR AEP_v_cd_hdr;  // 00 standard header
  PVOID AEP_v_cd_pvrp;         // 0C address of the VRP
  ULONG AEP_v_cd_drive;        // 10 drive number
} AEP_vrp_create_destroy, *PAEP_vrp_create_destroy;
```

你我所寫的 drivers，可以忽略此一 event，它純粹只做資訊傳輸用。

AEP_DEVICE_INQUIRY IOS 藉由「將一系列 *AEP_DEVICE_INQUIRY* events 傳給你的 port driver」，決定有多少 disk drive units 被附著於你的控制器上頭。其 event 處理常式會收到一個指標，指向一個 *AEP_inquiry_device* 結構：

```
typedef struct AEP_inquiry_device {
  struct AEPHDR AEP_I_d_hdr;        // 00 standard header
  ULONG  AEP_I_d_dcb;               // 0C address of DCB
} AEP_inquiry_device, *PAEP_inquiry_device;  // 10
```

被此 packet 指出的 DCB，其 *DCB_unit_on_ctl* 欄位是以 0 爲基準的索引值，指出 IOS 正在詢問的 unit。也就是說，對於第一個 inquiry call，此欄位爲 0，對於第二個 inquiry call，此欄位爲 1，依此類推。一個 non-SCSI port driver 應該以實際的 controller 個數來檢查這個 unit 索引值，看看 unit 是否存在。如果不存在，driver 應該傳回 *AEP_NO_INQ_DATA*。Driver 也可以傳回 *AEP_NO_MORE_DEVICES*，通知 IOS 說沒有更多的 units 了。這時候 IOS 就應該停止送出 inquiry events。無論如何 IOS 會在 128 個 inquiries 之後停止。如果 unit 的確存在，driver 應該完成 DCB 的數個欄位，以便描述這個 device。vendor ID 和 product ID 最終會出現在 Device Manager 的 devices 列表之中（以及其他地方）。舉個例子：

```
USHORT OnDeviceInquiry(PAEP_inquiry_device aep)
   {                          // OnDeviceInquiry
   PDCB dcb = (PDCB) aep->AEP_i_d_dcb;
   ASSERT(dcb);
   if (dcb->DCB_unit_on_ctl > 0)
      return AEP_NO_MORE_DEVICES;

   memcpy(dcb->DCB_vendor_id, "WALTONEY", 8);
   memcpy(dcb->DCB_product_id, "RAM Disk       ", 16);
   memcpy(dcb->DCB_rev_level, "0001", 4);

   return AEP_SUCCESS;
   }                          // OnDeviceInquiry
```

SCSI port driver 的動作有些不同。在 *AEP_INITIALIZE* 回返之前，port driver 設定 *AEP_bi_i_max_target* 和 *AEP_bi_i_max_lun* 欄位，指定被這個 port 服務的 unit 個數。這個 port driver 也可以在 *AEP_bi_flags* 中設定可有可無的 *AEP_BI_FL_SCSI_SCAN_DOWN* flag，於是 IOS 就會只送出指定個數的 inquiry events。如果 *AEP_BI_FL_SCSI_SCAN_DOWN* flag 被設立，IOS 會依「從最大到最小」的 unit 號碼來送出 events，否則就依「從最小到最大」的次序送出。在未來某些作業系統中，或有必要以反向來模仿某些 BIOSs「指定 drive 代碼」的行爲。在 Windows 95 中，IOS 計算 drive 代碼的方式與掃描次序無關。因此，或許沒有理由需要設立 *AEP_BI_FL_SCSI_SCAN_DOWN* flag。

經由特殊作法（hack），IOS 也能接納直接來自 Windows NT 的 SCSI miniport drivers。如果 *AEP_INITIALIZE* 處理常式設立了 *AEP_BI_FL_SEND_CONFIG_AGAIN* flag，IOS 會再一次啓動 inquiry 程序：它會送出一個 *AEP_INITIALZE*，後面緊跟著一組完整的 *AEP_DEVICE_INQUIRY* events。這樣的順序允許一個 miniport driver 做出一個類似這樣意義的回返動作:『我找到我的一個 adapters 了，但是可能還有更多個』。已經爲 Windows 95 做適當校訂的 miniport drivers，並不需要實作出這個特殊行爲。

最後，一個 SCSI port driver 應該傳回 *AEP_NO_INQ_DATA*（而不是 *AEP_NO_MORE_DEVICES*）以結束對 device 的列舉動作。並且應該以來自 device 的 SCSI-2 inquiry data 來填寫 *DCB_inquiry_flags* 欄位。

AEP_HALF_SEC IOS 會送出一個 *AEP_HALF_SEC* event，通知你說自從上一個 *AEP_HALF_SEC* event 之後，0.5 秒已經過去。你可以在呼叫 *IOS_Register* 之前於 *DRP_feature_code* 欄位中設立 *DRP_FC_HALF_SEC* flag，申請這些通知。如果你要求每半秒通知，還是會獲得 *AEP_1_SEC* 通知，但頻率只有一半。

AEP_INITIALIZE *AEP_INITIALIZE* 是 IOS 送給一個 port driver 的第一個 asynchronous event，它的發生甚至於在 driver 自己的 *IOS_Register* call 回返之前。這個 event 的處理常式會收到一個指標，指向一個 *AEP_bi_init* 結構，其中內容如下：

```
typedef struct AEP_bi_init {
  struct AEPHDR      AEP_bi_i_hdr;            // 00 standard header
  ULONG              AEP_bi_reference_data;   // 0C reference data from DRP
  UCHAR              AEP_bi_flags;            // 10 flags
  CHAR               AEP_bi_i_max_target;     // 11 maximum SCSI ID
  CHAR               AEP_bi_i_max_lun;        // 12 maximum SCSI LUN
  ULONG              AEP_bi_i_dcb;            // 13 initial DCB
  PVOID              AEP_bi_i_hdevnode;       // 17 DEVNODE address
  PVOID              AEP_bi_i_regkey;         // 1B registry key
  UCHAR              AEP_bi_i_align[1];       // 1F DWORD alignment
} AEP_bi_init, *PAEP_bi_init;                 // 20
```

AEP_bi_reference_data 欄位內含的值與 driver 呼叫 *IOS_Register* 時 *DRP_reference_data* 的值一樣。*AEP_bi_i_hdevnode* 的值正是此 controller device 的 "Configuration Manager DEVNODE handle"。*AEP_bi_flags*, *AEP_bi_i_max_target* 和 *AEP_bi_i_max_lun* 欄位都是輸出欄位，由 SCSI device drivers 使用。我在稍早描述 *AEP_DEVICE_INQUIRY* event 時已一併介紹過它們的目的了。這個結構中的 *AEP_bi_i_dcb* 和 *AEP_bi_i_regkey* 兩數值沒有意義。

Port driver 應該產生一個 DDB 用來描述 physical controller。IOS 定義了一個基本的 DDB 結構，格式如圖 **15-9**。你唯一可能使用過的一個標準 DDB 欄位是 *DDB_devnode_ptr*，它指向一個 DEVNODE，與「*AEP_INITIALIZE* event 所帶來的 *AEP_bi_i_hdevnode* 參數」所指的 DEVNODE 相同。你恐怕常常得定義屬於自己私人使用的較大結構，放置一些爲你的 device 所準備的額外欄位。例如，你可能會想記住

Configuration Manager 傳給你的 IRQ 和 base I/O 位址，那麼你可以使用這樣的結構：

```
typedef struct tagRAMDISKDDB
  {                             // RAMDISKDDB
  struct DDB;                   // 00 basic DDB
  DWORD   irq;                  // 20 assigned IRQ
  DWORD   iobase;               // 24 base port address
  HIRQ    irqhandle;            // 28 virtualized IRQ handle
  BOOL    busy;                 // 2C busy with a request?
} RAMDISKDDB, *PRAMDISKDDB;     // 30
```

當然啦，我所使用的 RAM-disk driver 只不過是個範例，並不需要一個 IRQ 或一個 base I/O port。我只是要告訴你如何面對那些 resources，免得這個例子一點用都沒有。順帶一提，請注意上個宣告中使用了一個無命名的結構成員（譯註：指的是 struct DDB）。將一個無命名的結構內嵌到另一個結構或 union 之中，是 Microsoft 對於 C/C++ 語言的一個巧妙擴充，允許你參考子結構中的欄位，又不需要再加一層名稱限制。這個特性可以使我們少打一些字。

```
typedef struct DDB {
  ULONG   DDB_phys_add;         // 00 physical address of this structure
  ULONG   DDB_Next_DDB;         // 04 next DDB for this device (internal use)
  ULONG   DDB_Next_DDB_init;    // 08 next DDB on init chain (internal use)
  ULONG   DDB_dcb_ptr;          // 0C first DCB for this device
  UCHAR   DDB_number_buses;     // 10 number of buses supported by device
  UCHAR   DDB_ios_flags;        // 11 flags for internal use
  USHORT  DDB_sig;              // 12 signature 0x4442 ('BD')
  PVOID   DDB_dvt;              // 14 address of DVT for owning driver
  PVOID   DDB_devnode_pr;       // 18 DEVNODE for this device
  PVOID   DDB_reserved;         // 1C (reserved)
} DDB , *pDDB;                  // 20
```

圖 15-9　IOS device data block 的格式

下面這個例子示範如何在 *AEP_INITIALIZE* 處理常式中配置一個 DDB：

```
USHORT OnInitialize(PAEP_bi_init aep)
   {                            // OnInitialize
   PRAMDISKDDB ddb;             // pointer to new DDB
   CMCONFIG config;             // allocated configuration info
```

```
    CONFIGRET code;            // configmg return code

    ddb = (PRAMDISKDDB) IspCreateDdb(sizeof(RAMDISKDDB), 0);
    if (!ddb)
        return (USHORT) AEP_FAILURE;     // couldn't create DDB
    return AEP_SUCCESS;        // pointer to new DDB is stored in ISP
    }                          // OnInitialize
```

對於一個真正的 device，你還需要確定 Configuration Manager 已經指定哪一個 I/O resources 給你，並且在上面做點事情。例如：

```
...
CMCONFIG config;
CONFIGRET code;

ddb->irqhandle = NULL;
code = CM_Get_Alloc_Log_Conf(&config,
    (DEVNODE) aep->AEP_bi_i_hdevnode,
    CM_GET_ALLOC_LOG_CONF_ALLOC);

if (code != CR_SUCCESS)
    {                    // no configuration
error:
    IspDeallocDdb((PDDB) ddb);
    return (USHORT) AEP_FAILURE;
    }                    // no configuration

if (config.wNumIRQs)
    {                // have an IRQ
    VID vid;          // IRQ descriptor

    ASSERT(config.wNumIRQs == 1); // should only be 1

    ddb->irq = vid.VID_IRQ_Number = config.bIRQRegisters[0];
    vid.VID_Options = VPICD_OPT_REF_DATA;
    if (config.bIRQAttrib[0] & fIRQD_Share)
        vid.VID_Options |= VPICD_OPT_CAN_SHARE;
    vid.VID_Hw_Int_Proc = (ULONG) HwIntProc;
    vid.VID_EOI_Proc = 0;
    vid.VID_Mask_Change_Proc = 0;
    vid.VID_IRET_Proc = 0;
    vid.VID_IRET_Time_Out = 500;
    vid.VID_Hw_Int_Ref = (ULONG) ddb;
```

```
    if (!(ddb->irqhandle = VPICD_Virtualize_IRQ(&vid)))
        goto error;
    }                   // have an IRQ

if (config.wNumIOPorts)
    {                   // have an I/O port
    ASSERT(config.wNumIOPorts == 1); // should only be 1
    ddb->iobase = config.wIOPortBase[0];
    }                   // have an I/O port
...
```

在這份程式片段中，我使用 *CM_Get_Alloc_Log_Conf* 來填寫 *config* 結構，填的是「爲我們的 DEVNODE 而配置的 logical configuration」的相關資訊。如果一個 IRQ 被指定到我的 device 上，我就建立一個 VID 結構並呼叫 *VPICD_VirtualizeIRQ* 將它虛擬化。我也將結果（一個 IRQ handle）儲存於 DDB 中，如此一來就可以在 *AEP_UNINITIALIZE* 期間將此 IRQ 的虛擬化解除。如果被指定的是一個 base I/O port，我同樣也把它記憶於 DDB 中。想必，我們得在這個 port 上做 IN/OUT 動作以處理 I/O requests。很重要的一點是，你必須明瞭 *AEP_INITIALIZE* 以及 DDB 結構（施行於一個可定址之 device 之上，例如典型的 disk controller）。如果同一個 VxD 要處理一個以上的 controller，它會收到針對每一個 controller 的個別 *AEP_INITIALIZE* events，並爲每一個 controller 產生一個 DDB。

AEP_IOP_TIMEOUT　當一個 I/O request（不管對任何 device）無法在某時間週期內（預設 7.5 秒）完成，那麼不管有無發生錯誤，IOS 都會送出一個 *AEP_IOP_TIMEOUT* event 給系統中的每個 driver。Event 函式會收到一個 *AEP_iop_timeout_occurred* 結構：

```
typedef struct AEP_iop_timeout_occurred {
  struct AEPHDR AEP_i_t_o_hdr;      // 00 standard header
  ULONG AEP_i_t_o_iop;              // 0C IOP that timed out
} AEP_iop_timeout_occurred, *PAEP_iop_timeout_occurred;
```

這個 event 主要預定被 port drivers 處理。它的目的是爲保護系統免於 "lockup"，通常那發生在「port driver 發出一個 I/O request 但 adapter 卻沒有反應」的時候。處理這個 event 時，如果有可能重新設定（reset）adapter，port driver 應該這麼做。然後，在重新設定

IOP_timer 的值等於 *IOP_timer_orig* 之後，再次嘗試這個 request。其他可能的動作包括：

- 將一個 completion code 儲存於 *IOR_status* 中並回呼 request。
- 清除（reset）*IOP_timer* 並且不要做任何其他事情。如果 request 雖存在於 queue 之中而實際上已處理過，那麼雖已 timeout，這樣的處理仍是正確的。

但是在採取以上任何動作之前，收到此 event 的 layer driver 應該先判斷它是否曾經看過目前所討論的 IOP。如果它不曾，那就應該傳回 *AEP_FAILURE*（或任何其他非零值）。如果它曾經看過 IOP，就應該進行任何它能夠完成的矯正行爲，並傳回 *AEP_SUCCESS*。一旦有人傳回 AEP_SUCCESS，你就會期望 IOS 停止送出（send）*AEP_IOP_TIMEOUT* events 給 drivers。然而，在我下筆此刻，IOS 仍然繼續送出（send） *AP_IOP_TIMOUT* events 給其餘的 drivers。雖然如此，你還是應該傳回正確的代碼，因爲這個不成熟的行爲可能在某一天改變掉。

AEP_MOUNT_NOTIFY IFS Manager 成功地裝載（mounted）一個新的 volume 之後，便會發出一個 *AEP_MOUNT_NOTIFY* event。其處理常式會收到一個 *AEP_mnt_notify* packet：

```
typedef struct AEP_mnt_notify {
  struct AEPHDR AEP_m_n_hdr;        // 00 standard header
  PVOID AEP_m_n_pvrp;               // 0C VRP of new volume
  ULONG AEP_m_n_drivemap;           // 10 child volume map
  ULONG AEP_m_n_drive;              // 14 drive just mounted
  ULONG AEP_m_n_effective_drive;    // 18 effective drive number
  ULONG AEP_m_n_atual_drive;        // 1C actual drive number
} AEP_mnt_notify, *PAEP_mnt_notify;
```

這個 event 的主要目的是通知壓縮管理者如 DRVSPACE 之流，說有一個新的 volume 已經到來。管理器會把特殊檔案（其中儲存著被壓縮過的資訊）視爲新的 volume，並爲它們安排一個磁碟機代碼。這一部份的實作說明已經逾越了本書範疇。

AEP_PEND_UNCONFIG_DCB　當一個 DCB 即將被移除（抽換），IOS 送出 *AEP_PEND_UNCONFIG_DCB* event。其處理常式會收到一個 *AEP_dcb_unconfig_pend* packet：

```
typedef struct AEP_dcb_unconfig_pend {
  struct AEPHDR AEP_d_u_hdr;   // 00 standard header
  ULONG AEP_d_u_dcb;           // 0C address of DCB
} AEP_dcb_unconfig_pend, *PAEP_dcb_unconfig_pend;
```

由於 DCB 很快就要結束了，任何 driver 都該停止使用它，並禁止對此 device 的任何更進一步的 I/O。如果一個動作已在進行，port driver 應該中止它。如果有任何動作被 "queued" 起來，負責 "queueing" 的人應該負責掃清（flush）queue。

如果你的 driver 將一個 IRQ 虛擬化，此一 event 並非停止中斷處理的正確時機，因爲 controller（實際產生中斷的那一實體）仍然存在。請等待直到 *AEP_UNINITIALIZE* event 出現爲止，那表示該 IRQ 最明顯關聯的 DDB 的確已經遠颺了。

AEP_REAL_MODE_HANDOFF　當 IFS Manager 正打算要從「只在初始化時才執行的碼（它把 MS-DOS 檔案系統呼叫下傳至到真實模式）」手中接管 INT 21h 的處理，IOS 送出一個 *AEP_REAL_MODE_HANDOFF* event。這個 "handoff" 是從真實模式到保護模式。Event 函式會收到一個 *AEP_rm_handoff* packet，其中只含標準表頭。或許沒有任何一個你我寫出來的 driver 會在乎這個 event。

AEP_REFRESH_DRIVE　當 volume 的鎖定者解除其鎖定（lock），IOS 會送出一個 *AEP_REFRESH_DRIVE* event。這個 event 的處理常式會收到一個 *AEP_drive_refresh* packet：

```
typedef struct AEP_drive_refresh {
  struct AEPHDR AEP_d_r_hdr;   // 00 standard header
  ULONG AEP_d_r_drive;         // 0C drive number
} AEP_drive_refresh, *PAEP_drive_refresh;
```

如果你記錄了「某筆資料於一個 drive 中的絕對位置」，現在你可能會想要重新清理（refresh）你的記錄，因爲雖然 volume 被鎖住，該筆資料可能已經被搬移。這個 event 和 *AEP_DCB_LOCK* 相反，所以你現在可以清除你所設立的任何 flag，提醒大家這個 volume 被鎖住了。

AEP_SYSTEM_CRIT_SHUTDOWN 在處理 *Sys_Critical_Shutdown* 這個系統控制訊息期間，IOS 送出 *AEP_SYSTEM_CRIT_SHUTDOWN* event 給所有登錄過的 drivers。這個 event 的處理常式會收到一個 *AEP_sys_crit_shutdown* packet，其中只內含標準的表頭。這個 packet 的內容並沒有什麼重要性。請小心不要將中斷 enable 起來，或是嘗試執行任何 V86 模式碼，因爲系統並不處於「可以容忍那樣的事情」的狀態下。當然，你可以只處理 *Sys_Critical_Init* 這個控制訊息，使用 IOS 非同步通告（asynchronous notification）的唯一理由就是你可能需要根據 IOS（而非你）對於 shutdown 次序的觀點，做某些事情。

AEP_SYSTEM_SHUTDOWN 在處理 *System_Exit* 系統控制訊息期間，IOS 送出 *AEP_SYSTEM_SHUTDOWN* event 給所有登錄過的 drivers。這個 event 的處理常式會收到一個 *AEP_sys_shutdown* packet，其中只內含標準的表頭。這個 packet 的內容沒有什麼重要性。

AEP_UNCONFIG_DCB 當 DCB 所描述的 device 消失，IOS 會送給所有的 drivers 一個 *AEP_UNCONFIG_DCB* event。Driver 會收到一個 *AEP_dcb_unconfig* packet：

```
typedef struct AEP_dcb_unconfig {
   struct AEPHDR AEP_d_u_hdr;      // 00 standard header
   ULONG AEP_d_u_dcb;              // 0C DCB being deleted
}  AEP_dcb_unconfig, *PAEP_dcb_unconfig;
```

這個 event 的主要目的是通知 vendor-supplied drivers（VSDs）說：有一個 DCB 消失不見了。VSD 的處理方式是查看是否它已經把自己放進 DCB 的 calldown stack 中。如果是，我們就減少計數器值，那是 *AEP_BOOT_COMPLETE* 處理常式要檢查的：

```
USHORT OnUnconfigDcb(PAEP_dcb_unconfig aep)
    {               // OnUnconfigDcb
    PDCB dcb = (PDCB) aep->AEP_d_u_dcb;
    if (<expression that's TRUE if we hooked this dcb>)
        --ndevices;
    return AEP_SUCCESS;
    }               // OnUnconfigDcb
```

IOS 稍後會送出一個 *AEP_BOOT_COMPLETE* event，VSD 可以再次決定是否要繼續被載入。

AEP_UNINITIALIZE　當一個 adapter 即將被拔除，IOS 便送給相關之 port driver 所轄的每一個 DDB 一個 *AEP_UNINITIALIZE* event。此訊息的處理常式會收到一個 *AEP_bi_uninit* packet，其中只內含標準表頭。雖然在 *AEP_INITIALIZE* event 期間你爲 DDB 配置記憶體，但並不需要明白地釋放這個 DDB，因爲 IOS 會自動做這件事情。但是你應該清理其他所有在 *AEP_INITIALIZE* event 期間所產生的介面和資料結構。如果 driver 已經在處理一個硬體中斷，你應該將 IRQ 恢復至其原來的行爲。

處理 *AEP_UNINITIALIZE* 時，你必須小心地只恢復那些在 *AEP_INITIALIZE* 期間真正完成的動作。之所以給你這份警告，是因爲如果初始化函式呼叫 *ISP_DEALLOC_DDB* service（要是它沒有辦法成功完成初始化動作，就會這麼做），IOS 便會送出一個 *AEP_UNINITIALIZE* event。假設你的 driver 將一個 IRQ 虛擬化，正常情況下你會將 virtual IRQ handle 儲存於 DDB 之中，如此一來便可以稍後再取消虛擬化。如果在「呼叫 *ISP_DEALLOC_DDB* 的任何可能性」之前便爲 DDB 的 IRQ handle 欄位設定初值，你的 *AEP_UNINITIALIZE* 處理常式便可以輕鬆避免錯誤發生：只要檢查這個 IRQ handle 是否已經與其原值有所改變：

```
USHORT OnInitialize(PAEP_bi_init aep)
    {                           // OnInitialize
    PRAMDISKDDB ddb;

    if (!(ddb = (PRAMDISKDDB) IspCreateDdb(sizeof(RAMDISKDDB), 0)))
        return (USHORT) AEP_FAILURE;
    ddb->irqhandle = NULL;      // initialize IRQ handle field
```

```
    ...
    ddb->irqhandle = VPICD_Virtualize_IRQ(&vid);
    ...
error:
    IspDeallocDdb((PDDB) ddb);
    return (USHORT) AEP_FAILURE;
    }                           // OnInitialize

USHORT OnUninitialize(PAEP_bi_uninit aep)
    {                           // OnUninitialize
    PRAMDISKDDB ddb = (PRAMDISKDDB) aep->AEP_bi_u_hdr.AEP_ddb;

    if (!ddb)
        return AEP_SUCCESS;

    if (ddb->irqhandle)
        {                       // unvirtualize IRQ
        VPICD_Force_Default_Behavior(ddb->irqhandle);
        ddb->irqhandle = NULL;
        }                       // unvirtualize IRQ

    return AEP_SUCCESS;
    }                       // OnUninitialize
```

以此方式來組織程式碼，*OnUninitialize* 便能夠正確地將 IRQ「反虛擬化」而不必在乎 *OnUninitialize* 如何被觸及（執行）。因爲 DDB 的 *irqhandle* 欄位只可能是 0 或虛擬化後的 IRQ handle，絕不可能是一個未設初值的假貨。

實作幾個 VxD – 個 Drivers

在這一節中，我要討論兩個小型程式專案，它們使用前數節中辛苦呈現的資訊。大部份程式員一開始並不需要寫出一個 VxD 來和 IOS 交打道。他們需要做的是爲特定硬體寫一個 port driver 或 vendor-supplied driver（VSD）以補充或修正既存之 drivers 的行爲。

建造一個 Port Driver

IOS driver hierarchy 的最低真實層次內含一個 port driver，用來控制一個硬體磁碟控制器（hardware disk controller）及其一組 disk drives。一個 IOS port driver 包括有一個 asynchronous event handler、一個 I/O request handler （那是一個或多個 DCBs 之 calldown stack 的一部份），以及（通常）一個 VPICD 硬體中斷處理常式。

爲了產生一個有意義的 port driver 實例，而又不需要太涉入 control signals 和 interrupt timing 細節，我決定寫一個 RAM-disk driver。RAM disk 其實就是一個虛擬磁碟 driver，利用記憶體（而非磁碟）來放置資料，對系統而言它就像個真正的磁碟一樣。早期在個人電腦上，RAM disk 十分普及，因爲它們提供了一個非常好的效率提昇（記憶體當然遠比旋轉碟片快得多了），但也要求你的電腦必須加上一些 extended memory。現代的虛擬記憶體系統再加上穩健的磁碟快取（caching）技術，使得 RAM disk 基本上已經過氣了，因爲它消耗虛擬記憶體，而系統本身卻可能更有效率地使用虛擬記憶體。但是對我們的目的而言，它仍然是個好例子，如果你想要建立自己的檔案系統，RAM disk 是一個好的試鍊場所，直到你的檔案系統健全到可以施行於實際硬體爲止。

雜務管理（Housekeeping）

我已經透過範例程式碼，說明建立一個 RAM-disk port driver 時所需的雜務管理工作。書附碟片中的 \CHAP15\PORTDRIVER 目錄下的這個 port driver 做了以下這些你現在都已知曉的事：

- 它處理 *Sys_Dynamic_Device_Init* 訊息的方式是呼叫 *IOS_Register* 並夾帶一個 DRP，以之表示我們的 driver 是一個 *DRP_MISC_PD*（level 19）layer driver。
- 它處理 *AEP_INITIALIZE* 的方式是產生一個 DDB。
- 它處理 *AEP_CONFIG_DCB* 的方式是填寫 geometry 屬性，用來表示有「一個 cylinder、每個 cylinder 有一個 track、每個 track 有 4096 sectors（每個 sector 是 512 bytes）」。也就是說，這個磁碟含有 2 MB 空間。*AEP_CONFIG_DCB*

處理常式也把這個 driver 安插到 calldown stack 的 level 19，並更新一個全域性計數器的值（計算受影響的 DCBs）。此外它還做更多事情（包括配置記憶體以模擬磁碟空間），我將在下一節描述。

- 它處理 *AEP_UNCONFIG_DCB* 的方式是將「受影響之 DCBs」的計數器減小。它還做更多事情，我將在稍後提到。
- 它處理 *AEP_BOOT_COMPLETE* 的方式是檢驗「受影響之 DCBs」的計數器。如果計數器不是 0，就請求繼續被載入。

我也建立了一個標準的 .INF 檔來安裝這個 driver（RAMDISK.PDR）於 IOSUBSYS 目錄中，做爲一種 hdc（hard disk controller）device。你可以在書附碟片中找到這個 .INF 檔。

模擬一顆磁碟

稍早我對這個 driver 的討論中，只剩下如何真正模擬一個 disk drive 還沒談。在 *AEP_CONFIG_DCB* event 期間，這個 driver 配置一個 8-MB 虛擬記憶體區塊，用來模擬儲存媒體：

```
USHORT OnConfigDcb(PAEP_dcb_config aep)
    {
    ...
    if (!(dcb->DCB_Port_Specific = (ULONG) _PageAllocate(2048,
        PG_SYS, NULL, 0, 0, 0, NULL,
        PAGEZEROINIT | PAGELOCKEDIFDP)))
        return (USHORT) AEP_FAILURE;
    ...
    }
```

我呼叫 *_PageAllocate*，配置並「委派（commits）」（譯註：將實際記憶體指定給虛擬位址）2048 個 pages （8MB）記憶體。如果 MS-DOS 被用於 paging（在我的系統中絕不如此），這塊記憶體也會被 page locked。*AEP_CONFIG_DCB* 處理常式會將結果（線性位址）儲存在 DCB 的 *DCB_Port_Specific* 欄位中以備稍後使用，並在配置而得的記憶體區塊起始處，初始化一個 master boot record，以及一個雛形規模的 file allocation table（在 512 bytes 處），但是這段初始化動作實在太過無聊，我就不在這裡重複了。重要

的是，記得納入一個合法的 partition table，以及一個 AA55h 簽名符號於 boot record 尾端，否則 disk TSD 會拒絕爲此磁碟裝載（mount）一個 logical volume。由於我正確地完成了所有步驟，我得到的報酬是每當使用【新增硬體（Add New Hardware）】精靈來安裝它，在 Windows 95 MS-DOS 視窗以及 Windows 95 shell 中就會出現一個磁碟。

記憶體配置步驟在 *AEP_UNCONFIG_DCB* event 期間可以獲得「平反」：

```
USHORT OnUnconfigDcb(PAEP_dcb_unconfig aep)
   {                        // OnUnconfigDcb
   PDCB dcb = (PDCB) aep->AEP_d_u_dcb;
   ASSERT(dcb);

   if (!(dcb->DCB_cmn.DCB_device_flags & DCB_DEV_PHYSICAL)
      || memcmp(dcb->DCB_vendor_id, "WALTONEY", 8) != 0)
      return AEP_SUCCESS;

   if (dcb->DCB_Port_Specific)
      {                     // release memory
      _PageFree((PVOID) dcb->DCB_Port_Specific, 0);
      dcb->DCB_Port_Specific = 0;
      }                     // release memory
   --ndevices;
   ASSERT(ndevices >= 0);
   return AEP_SUCCESS;
   }                        // OnUnconfigDcb
```

處理 I/O Requests

我們所討論的分層架構（layered architecture）的整個目的就是要把 I/O requests 繞行（route）進入你所寫的與 device 相關的碼中。本例這個 RAM-disk 的「request 處理常式」只需處理 read 和 write 兩種 requests，而對這兩種型態的 requests 而言，程式碼是相同的。下面就是完整的函式：

```
#0000  #pragma VxD_LOCKED_CODE_SEG
#0001  VOID OnRequest(PIOP iop)
#0002     {                        // OnRequest
#0003     #define ior iop->IOP_ior
#0004     DWORD funcode = ior.IOR_func;
#0005     PDCB dcb = (PDCB) iop->IOP_physical_dcb;
#0006
#0007     ior.IOR_status = IORS_SUCCESS; // assume it'll succeed
#0008     dcb->DCB_cmn.DCB_device_flags |= DCB_DEV_IO_ACTIVE;
#0009
#0010     switch (funcode)
#0011        {                 // dispatch function processor
#0012
#0013     ////////////////////////////////////////////////////////
#0014
#0015     case IOR_READ:             // IOR_func == 0
#0016        {                       // IOR_READ
#0017        DWORD sector = ior.IOR_start_addr[0];
#0018        PBYTE diskdata = (PBYTE) (dcb->DCB_Port_Specific +
#0019           sector * dcb->DCB_actual_blk_size);
#0020
#0021        ASSERT(sector << dcb->DCB_actual_sector_cnt[0]);
#0022        ASSERT(ior.IOR_start_addr[1] == 0);
#0023
#0024        if (ior.IOR_flags & IORF_SCATTER_GATHER)
#0025           {              // have scatter/gather structures
#0026           _BlockDev_Scatter_Gather* sgd =
#0027              (_BlockDev_Scatter_Gather*) ior.IOR_buffer_ptr;
#0028           PBYTE memdata;
#0029           DWORD nbytes;
#0030           while ((nbytes = sgd->BD_SG_Count))
#0031              {           // for each s/g structure
#0032              memdata = (PBYTE) sgd->BD_SG_Buffer_Ptr;
#0033              if (!(ior.IOR_flags & IORF_CHAR_COMMAND))
#0034                 nbytes *= dcb->DCB_actual_blk_size;
#0035              memcpy(memdata, diskdata, nbytes);
#0036              diskdata += nbytes;
#0037              ++sgd;
#0038              }           // for each s/g structure
#0039           }              // have scatter/gather structures
#0040        else
#0041           {              // have simple buffer address
#0042           PBYTE memdata = (PBYTE) ior.IOR_buffer_ptr;
#0043           DWORD nbytes = ior.IOR_xfer_count;
#0044           if (!(ior.IOR_flags & IORF_CHAR_COMMAND))
#0045              nbytes *= dcb->DCB_actual_blk_size;
```

```
#0046           memcpy(memdata, diskdata, nbytes);
#0047           }             // have simple buffer address
#0048
#0049       break;
#0050       }                 // IOR_READ
#0051
#0052   ////////////////////////////////////////////////////////
#0053
#0054   case IOR_WRITE:           // IOR_func == 1
#0055       {                     // IOR_WRITE
#0056       DWORD sector = ior.IOR_start_addr[0];
#0057       PBYTE diskdata = (PBYTE) (dcb->DCB_Port_Specific +
#0058           sector * dcb->DCB_actual_blk_size);
#0059
#0060       ASSERT(sector);   // rewriting boot sector??
#0061       ASSERT(sector << dcb->DCB_actual_sector_cnt[0]);
#0062       ASSERT(ior.IOR_start_addr[1] == 0);
#0063
#0064       if (ior.IOR_flags & IORF_SCATTER_GATHER)
#0065           {             // have scatter/gather structures
#0066           _BlockDev_Scatter_Gather* sgd =
#0067               (_BlockDev_Scatter_Gather*) ior.IOR_buffer_ptr;
#0068           PBYTE memdata;
#0069           DWORD nbytes;
#0070           while ((nbytes = sgd->BD_SG_Count))
#0071               {         // for each s/g structure
#0072               memdata = (PBYTE) sgd->BD_SG_Buffer_Ptr;
#0073               if (!(ior.IOR_flags & IORF_CHAR_COMMAND))
#0074                   nbytes *= dcb->DCB_actual_blk_size;
#0075               memcpy(diskdata, memdata, nbytes);
#0076               diskdata += nbytes;
#0077               ++sgd;
#0078               }         // for each s/g structure
#0079           }             // have scatter/gather structures
#0080       else
#0081           {             // have simple buffer address
#0082           PBYTE memdata = (PBYTE) ior.IOR_buffer_ptr;
#0083           DWORD nbytes = ior.IOR_xfer_count;
#0084           if (!(ior.IOR_flags & IORF_CHAR_COMMAND))
#0085               nbytes *= dcb->DCB_actual_blk_size;
#0086           memcpy(diskdata, memdata, nbytes);
#0087           }             // have simple buffer address
#0088
#0089       break;
#0090       }                 // IOR_WRITE
#0091
```

```
#0092    ////////////////////////////////////////////////////////
#0093
#0094    default:
#0095       dcb->DCB_cmn.DCB_device_flags &= ~DCB_DEV_IO_ACTIVE;
#0096       DoCallDown(iop);
#0097       return;
#0098       }                // dispatch function processor
#0099
#0100    dcb->DCB_cmn.DCB_device_flags &= ~DCB_DEV_IO_ACTIVE;
#0101    DoCallBack(iop);       // we're done with this request
#0102
#0103    #undef ior
#0104    }                       // OnRequest
```

我們早就已經看過上面所使用的 *DoCallDown* 和 *DoCallBack* 兩函式，這裡我要討論 *OnRequest* 的其他細節。

OnRequest 位於 locked code segment。對此 driver 而言這並非必要，因爲當它正在處理一個對此 RAM drive 的需求（request）時，如果發生 page fault，並不會帶來什麼傷害。畢竟我們並未把置換檔（swap file）放在這個模型磁碟中。然而，一般而言，你的 driver 會有一些「request 處理動作」，在你自己的硬體中斷處理期間被呼叫。你知道的，像這樣的碼必須放在 page-locked 記憶體中，並且只能使用 asynchronous VxD services。

OnRequest 的唯一參數是一個 IOP 位址。我對其中的 DCB 和 IOP 所指的 calldown stack 有興趣。除了那個之外，我只對內嵌於 IOP 中的 IOR 結構感興趣。我使用一個巨集（*ior*）來縮短對「內嵌之 IOR」的參考指令，因爲，我想你現在已經認識我了，我憎惡無意義的長字串、雙倍的語法。

在此 driver 中，當我把 *OnRequest* 安插到 calldown stack 之中，我聲明了 *DCB_dmd_serialize* demand。這個 demand 引起程式碼逆流而上（upstream），從 driver 到 queue 並且將 requests 逐出 queue 之中。我在此 driver 中必須做的工作就是在 request 活動前設立 *DCB_DEV_IO_ACTIVE* flag，活動後清除之。其他情況下，你可能需要呼叫 *IlbEnqueueIop* 和 *IlbDequeueIop* 以自己處理 queueing。你當然需要一個硬體中斷處理常式，至少用以清除 *DCB_DEV_IO_ACTIVE* 以便釋放 queue。

面對 read 和 write 兩種 requests，處理方式大致相同，所以我一併描述它們。即將被讀取或被寫入的磁碟位置（sector），是一個 physical sector 號碼，它應該是個 0~4095 的整數。資料來源或資料目的地（變數 *diskdata*）座落於以 512 bytes（*DCB_actual_blk_size*）爲單位的 sector 上，並且在我們於 *AEP_CONFIG_DCB* 期間配置得來的記憶體之中。

我們從不要求 *IOR_sgd_lin_phys* 陣列被建構好，因爲我們並不使用線性位址。在一個比較寫實的 driver 中，你應該宣稱 *DCB_dmd_phys_sgd* demand，以便要求更高的 layers 產生 SGD 陣列，於是陣列中便持有資料緩衝區的實際位址。你或許也會使用我在 13 章討論過的 VDMAD services 來設定針對你的 device 的 DMA transfers。

在此簡化範例中，我們直接使用被原始的 IOS client 所設定的緩衝區指標。正常情況下只有 criteria 函式會查看這個緩衝區指標。Client 提供單一緩衝區，或是一個由 scatter/gather 結構（BlockDev 格式）所組成的陣列。RAM-disk driver 根據 *IORF_SCATTER_GATHER* flag 來區分兩者。如果 flag 設立，我們就走遍 *_BlockDev_Scatter_Gather* 結構陣列，直到到達一個其值爲 0 的 DWORD 爲止。對於每一個 scatter/gather 結構，我們寫入（或讀出）定量（*BD_SG_Count*）的資料至（從）指定的緩衝區（*BD_SG_Buffer_Ptr*，那是一個線性位址而非實際位址）。如果 *IORF_CHAR_COMMAND* 設立，每一個 scatter/gather descriptor 中的傳輸計數便是以 bytes 表示，否則便是以 device 的 block size （本例爲 512 bytes）爲單位。

如果 *IORF_SCATTER_GATHER* 是清除狀態，我們被強迫使用單獨一個資料緩衝區，而非一系列緩衝區。這種情況下，*IOR_xfer_count* 便是我們應該傳輸的數量（bytes 或 sectors）。

這個 driver 總是能夠完全並成功地處理 read 和 write requests，所以 *OnRequest* 最終回呼 request，並帶以一個 0（*IORS_SUCCESS*）代碼。此一 driver 並不處理其他任何事情，所以它簡單地往下呼叫（calls down）至下一個 layer（如果有的話）以處理其他 requests。舉個例子，我們可以在獲得 requests 後重新計算 drive 的 geometry 屬性，或是檢查儲存媒體，但我們也可以忽略之。

建造一個廠商供應的 Driver（Vendor-Supplied Driver）

廠商供應的 Driver（Vendor-Supplied Driver，VSDs）在 IOS 體制中提供多種目的的服務。寫一個 VSD 可以讓你對你所選擇的任何 device 的任何 I/O request 的處理加以干涉。由於有 9 個 layers 保留給 VSDs，你可以在 calldown stack 中選擇一個最適合你的插入點。舉個例子，一個 SCSI VSD 可以在 SCSI'izer 建立一個 SCSI request block 之前或之後加以干涉。這當然不是你寫 VSD 的原因。真正的原因就像硬體那麼繁多，而且你可以加上一些想像空間。

架構上，一個 VSD 和一個 port driver 之間的主要不同在於：VSD 收到 *AEP_CONFIG_DCB* event 多少次。一個 port driver 只能夠針對它所產生的 devices 看到這個 event，一個 VSD 則能夠針對系統中的每一個 DCB 都看到這個 event。這個事實允許一個 VSD 能夠針對任何一個 devices，"hook into" 其 calldown stack。

爲了提供一個 VSD 實例，我寫了一個小型的 VSD，用來監視 I/O requests。作法是針對每一個 request 送出一個資訊包裹給一個 Windows-based 應用程式。這個 VSD 以及應用程式和描述於 "*Examining the Windows 95 Layered File System*" 一文（作者 Mark Russinovich 和 Bryce Cogswell，發表於 Dr. Dobb's Journal，12/1995）中的那個實例十分類似，**圖 15-10** 顯示我的系統中的一個 DIR A: 命令所引發的 "request log"（譯註：log 是運轉記錄的意思）。

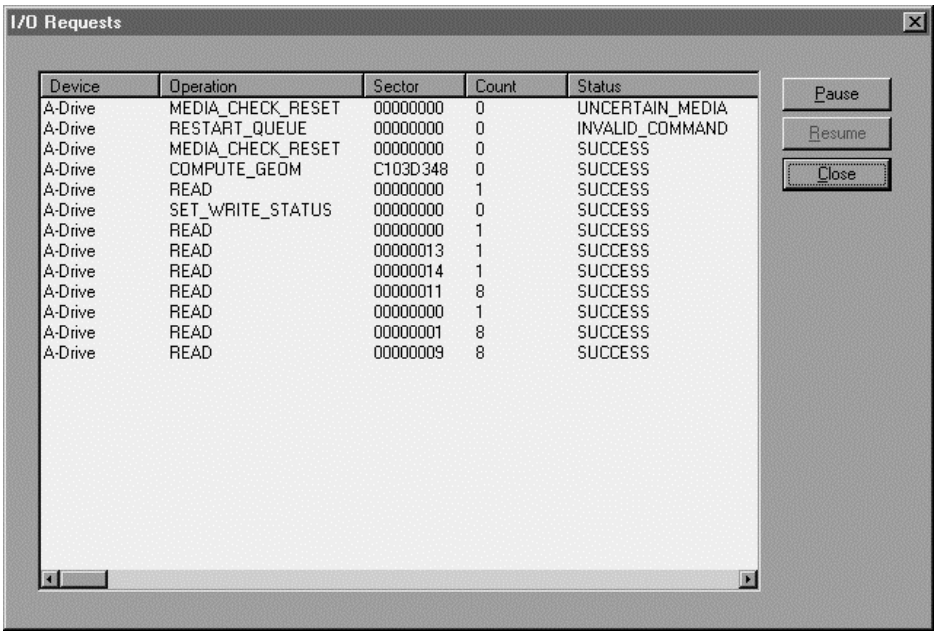


圖 15-10　MONITOR 範例程式的輸出畫面

雜務管理（Housekeeping）

這個範例（可從書附碟片的 \CHAP15\IOSMONITOR 目錄中取得）的大部份工作是在 ring3 應用程式之中，它負責螢幕輸出。但我不打算在此描述這個部份。VSD 本身倒是相對地簡單，它做了以下這些你已經瞭解的事情：

- 它處理 *Sys_Dynamic_Device_Init* 訊息的方式是呼叫 *IOS_Register* 並夾帶一個 DRP，以之表示我們的 driver 是一個 *DRP_VSD_9*（level 17）layer driver。此外，這個 VSD 使用正規的 *List_Create* service 配置一串列（linked list）的記憶體區塊，可以在 I/O request 函式運行期間存取之：

```
extern DRP theDRP;              // device registration packet

BOOL OnSysDynamicDeviceInit()
   {                          // OnSysDynamicDeviceInit
   list = List_Create(LF_ASYNC | LF_ALLOC_ERROR,
      sizeof(MONINFO));
```

```
    if (!list)
        return FALSE;
    IOS_Register(&theDRP);
    return TRUE;
    }                       // OnSysDynamicDeviceInit
```

其中的 *MONINFO* 是我宣告的一個結構，用來放置所有令人感興趣的 I/O request 相關資料，也是對頭的 Windows 應用程式將顯示出來的資料。串列變數是一個全域指標，指向串列物件。Driver 在 *Sys_Dynamic_Device_Exit* 期間摧毀此一串列：

```
BOOL OnSysDynamicDeviceExit()
    {                       // OnSysDynamicDeviceExit
    if (list)
        List_Destroy(list);
    return TRUE;
    }                       // OnSysDynamicDeviceExit
```

- 它忽略 *AEP_INITIALIZE* 和 *AEP_UNINITIALIZE* events。也就是說，面對這兩個 events 它都傳回 *AEP_SUCCESS*，並沒有真正做什麼事。
- 它處理 *AEP_CONFIG_DCB* 的方式是將每一個 physical DCB "hooking into" calldown stack：

```
USHORT OnConfigDcb(PAEP_dcb_config aep)
   {                        // OnConfigDcb
   PDCB dcb = (PDCB) aep->AEP_d_c_dcb;
   if (!(dcb->DCB_cmn.DCB_device_flags & DCB_DEV_PHYSICAL))
       return AEP_SUCCESS;
   if ((IspInsertCalldown(dcb, OnRequest,
       (PDDB) aep->AEP_d_c_hdr.AEP_ddb, 0,
       dcb->DCB_cmn.DCB_dmd_flags, aep->AEP_d_c_hdr.AEP_lgn)))
       ++ndevices;
   return AEP_SUCCESS;
   }                        // OnConfigDcb
```

- 它處理 *AEP_UNCONFIG_DCB* 的方式是：總是爲每一個 physical DCB 減少 *ndevice* 計數器的值：

```
USHORT OnUnconfigDcb(PAEP_dcb_unconfig aep)
    {                       // OnUnconfigDcb
```

```
    PDCB dcb = (PDCB) aep->AEP_d_u_dcb;
    if (!(dcb->DCB_cmn.DCB_device_flags & DCB_DEV_PHYSICAL))
       return AEP_SUCCESS;
    ASSERT(ndevices > 0);
    --ndevices;
    return AEP_SUCCESS;
    }                       // OnUnconfigDcb
```

- 它處理 *AEP_BOOT_COMPLETE* 的方式是，如果已經 "hooked into" 任何 DCB 的 calldown stack，就傳回 *AEP_SUCCESS*：

```
USHORT OnBootComplete(PAEP_boot_done aep)
    {                       // OnBootComplete
    return ndevices ? AEP_SUCCESS : AEP_FAILURE;
    }                       // OnBootComplete
```

監視所有的 Requests

這個 VSD 的 request 處理常式也十分簡單：

```
VOID OnRequest(PIOP iop)
    {                           // OnRequest
    if (userproc)               // APC callback routine
       InsertCallBack(iop, OnRequestComplete, 0);
    DoCallDown(iop);
    }                           // OnRequest
```

如你所見，這個 request 函式只是簡單地針對此一 request，將一筆項目（entry）加到 callback list 之中。當一個較低層級的 driver 回呼此一 request 表示任務完成，這個 VSD 才處理其真正的工作：

```
VOID OnRequestComplete(PIOP iop)
    {                       // OnRequestComplete
    #define ior iop->IOP_ior
    DWORD funcode = ior.IOR_func;
    PDCB dcb = (PDCB) iop->IOP_physical_dcb;
    PMONINFO mip = GetBlock();

    if (mip)
```

```
        {                   // notify ring-three app
        mip->dcb = dcb;
        mip->opcode = (BYTE) ior.IOR_func;
        mip->status = (BYTE) ior.IOR_status;
        mip->sector = ior.IOR_start_addr[0];

        mip->nbytes = 0;
        if (ior.IOR_func <= IOR_WRITEV)
            {               // compute transfer length
            if (ior.IOR_flags & IORF_SCATTER_GATHER)
                {       // request used scatter/gather records
                _BlockDev_Scatter_Gather* sgp =
                    (_BlockDev_Scatter_Gather*)
                    ior.IOR_buffer_ptr;
                while (sgp->BD_SG_Count)
                    mip->nbytes += sgp->BD_SG_Count, ++sgp;
                }       // request used scatter/gather records
            else
                mip->nbytes = ior.IOR_xfer_count;
            }               // compute transfer length

        if (!userproc || !_VWIN32_QueueUserApc(userproc,
            (DWORD) mip, thread))
            ReturnBlock(mip);
        }                   // notify ring-three app

    DoCallBack(iop);
    }                       // OnRequestComplete
```

其中的 *GetBlock* 和 *ReturnBlock* 都是輔助函式，用來從非同步串列（asynchronous linked list）中配置和釋放記憶體，該串列是在 *Sys_Dynamic_Device_Init* 處理期間設定好的：

```
void ReturnBlock(PMONINFO mip)
    {                       // ReturnBlock
    _asm pushfd
    _asm cli
    List_Deallocate(list, (VMMLISTNODE) mip);
    _asm popfd
    }                       // ReturnBlock

PMONINFO GetBlock()
    {                       // GetBlock
```

```
PMONINFO mip;

_asm pushfd
_asm cli
mip = List_Allocate(list);
_asm popfd

return mip;
}                       // GetBlock
```

MONITOR 應用程式的一些細節

用來顯示 "request log" 的 MONITOR 應用程式，有一些有趣的細節。它利用 *CreateFile* 獲得此 VSD 的 device handle，然後使用數個 *DeviceIoControl* calls 與此 VSD 通訊（大部份 VSDs 並不會與應用程式有密切的關連，所以它們也就不需要支援 *DeviceIoControl*）。有一個 *DeviceIoControl* calls 用來登錄 APC callback 函式（*userproc*），該函式會被 *OnRequest* 呼叫。另一個 *DeviceIoControl* calls 會呼叫 *ReturnBlock*，釋放一個 *MONINFO* 區塊 -- 當 MONITOR 程式已經做完它該做的功課。MONITOR 程式本身衍孵（spawns）出一個 thread（譯註：該 thread 函式即下面的 *DoMonitor*），該區一目的就是要等待來自 VSD 的 APC calls。這個 thread 也等待一個 event，該意義是「end user 希望中止或結束 events 的顯示」。

```
DWORD __stdcall DoMonitor(CRequestLog* log)
   {                            // DoMonitor
   void (WINAPI *acallback)(PMONINFO) = callback;
   if (!DeviceIoControl(log->m_reqmon,
      REQMON_SETMONITORADDRESS,
      &acallback, sizeof(acallback), NULL, 0, NULL, NULL))
      {                         // can't establish callback
      PostMessage(log->m_hWnd, WM_COMMAND, IDB_PAUSE, 0);
      return 1;
      }                         // can't establish callback
   logptr = log;
   while (WaitForSingleObjectEx(log->m_evkill, INFINITE, TRUE)
      == WAIT_IO_COMPLETION)
      ;                         // until told to quit
```

```
    acallback = NULL;
    DeviceIoControl(log->m_reqmon, REQMON_SETMONITORADDRESS,
       &acallback, sizeof(acallback), NULL, 0, NULL, NULL);
    log->m_thread = NULL;    // we're out of here
    return 0;
    }                              // DoMonitor
```

我們曾在第 10 章談到關於據稱型 API 函式的使用，例如 *WaitForSingleObjectEx* 用來執行一個 "alertable wait"，以支援來自 VxDs 的 APC calls。這個 APC callback 函式只是簡單地 "posts" 一個訊息給 main thread：

```
void WINAPI callback(PMONINFO mip)
    {                          // callback
    PostMessage(logptr->m_hWnd, WM_USER+256, 0, (LPARAM) mip);
    }                          // callback
```

WM_USER+256（一個自定訊息）的處理常式會為程式視窗中的 "list" 控制元件加上一列資訊，然後經由第二次 *DeviceIoControl* call，將這個 *MONINFO* 區塊傳回給 VSD。

OnRequestComplete（譯註：原書誤爲 *OnRequest*）的動作如下所述：它從串列手上取得一個 *MONINFO* 區塊。由於非同步串列（asynchronous lists）的處理規定，*GetBlock* 輔助函式會在呼叫 *List_Allocate* 前 "disables" 中斷。使用一個非同步串列（asynchronous list），可以使這個 VSD 即使在面對 swapping device 時仍能夠安全地監視其 requests。*OnRequestComplete*（譯註：原書誤爲 *OnRequest*）填寫 *MONINFO* 的大部份欄位，作法是將對應資料從 I/O request packet 中拷貝過來。填寫傳輸量（transfer count，程式中命名爲 *nbytes*，易被誤解）時需要對 scatter/gather descriptors 加總計算（如果有的話）。一旦 *OnRequestComplete* 函式（譯註：原書誤爲 *OnRequest*）完成了 *MONINFO* 區塊，它就使用 *_VWIN32_QueueUserApc* 來對一個 asynchronous procedure call（APC）排程，使控制權有機會回到 Windows 應用程式 -- 後者會將此 request 的一份描述內容格式化，提供給 end user 看。

這個 VSD 和你可能寫的其他 VSDs 的重大差異在於，它不構建任何屬於自己的 I/O request。撰寫 VSD 的理由之一是爲了提供或修正既存 drivers 的行爲，而你可能需要做的事情之一就是明確地陳述或執行你自己的 requests。執行 requests 的方法就是使用內部機制產生並提出一個 I/O request packet。稍早我在討論 IOP 資料結構時也對此技術做了描述。在這樣的 VSD 中，你可能會有類似下面這樣的碼：

```
USHORT offset = (USHORT) (dcb->DCB_cmn.DCB_expansion_length
   + FIELDOFFSET(IOP, IOP_ior));
USHORT size = offset + sizeof(IOR)
   + dcb->DCB_max_sg_elements * sizeof(SGD);
PIOP iop = IspCreateIop(size, offset, ISP_M_FL_MUST_SUCCEED);
PIOR ior = &iop->IOP_ior;

iop->IOP_original_dcb = (ULONG) dcb;
iop->IOP_physical_dcb = (ULONG) dcb->DCB_cmn.DCB_physical_dcb;

ior->IOR_next = 0;
ior->IOR_start_addr[1] = 0;
ior->IOR_flags = IORF_VERSION_002;
ior->IOR_private_client = offset;
ior->IOR_req_vol_handle = dcb->DCB_cmn.DCB_vrp_ptr;
ior->IOR_sgd_lin_phys = (ULONG) (ior + 1);
ior->IOR_num_sgds = 0;
ior->IOR_vol_designtr = dcb->DCB_cmn.DCB_unit_number;

[fill in remaining IOR fields to describe request]

IlbIntIoCriteria(iop);
IlbInternalRequest(iop, dcb, OnRequest);
IspDeallocMem((PVOID) ((DWORD) ior - ior->IOR_private_client));
```

IlbInternalRequest 的第三個參數告訴 IOS 說，從 calldown stack 的哪裡開始處理這個 request。指定的 calldown 函式應該是你打算開始的那一個的上一層。（你可以從 calldown stack 的一個條目中取出此一指標）。一如此處所顯示，我們尋求讓「在我們自己的 calldown entry 之後」的那一個先被喚起。如果你提供的是個 NULL 指標，這個 request 會起始於 calldown stack 的頂部。請小心遞迴的情況。

第四章

擴充作業系統

Extending the Operating System

第 16 章

可安裝的檔案系統（Installable File Systems）

前一版 Windows（譯註：Windows 3.1）只能扮演「MS-DOS 圍裙」的角色，因爲只有 MS-DOS 才知道如何管理磁碟中的檔案。對於你唯一的一個檔案系統而言，真實模式是一個可怕的地方：多個 processes 沒辦法同時對檔案做動作，因爲有 Windows critical section 的保護；network redirectors 必須支援一個未公開的 INT 2Fh 協定；CD-ROM drives 必須透過拼拼湊湊的 MSCDEX（它在某些方面模仿著 network redirector）才能處理檔案系統。Windows 95 透過所謂的 Installable File System（IFS）Manager 免除了上述這些問題。

圖 **16-1** 說明 Windows 95 架構下的 IFS Manager 所扮演的角色。IFS Manager 開放（exports）許多 VxD-level services 給系統其他部份使用。它呼叫 file system drivers（FSDs）以實作出各式各樣的檔案系統，例如 FAT 和 CD-ROM 檔案系統。FSDs 內部與 I/O Supervisor（第 15 章所討論）內的 disk drivers 交談。FSDs 也提供網路連結（network connections）下的檔案動作之中的必要的重導向（redirection）動作。Network redirectors 這一主題超出了本書範疇，我並不打算介紹。

雖然 Microsoft 在 DDK 中提供了一個非常好的說明文件，描述一個 driver 如何呼叫 IFS Manager，以及如何被 IFS Manager 呼叫，但是關於如何撰寫 file system drivers，Microsoft 卻沒有給我們太多的指導。爲了引導出我即將給你的這些資訊，我必須寫一個非常簡單的 file system 實例（Low Performance File System，或稱爲 LPFS），並對一些系統元件如 IFS Manager 和 VFAT 做大量的逆向工程。LPFS 雖然簡單，但它還是牽涉了許多程式碼。

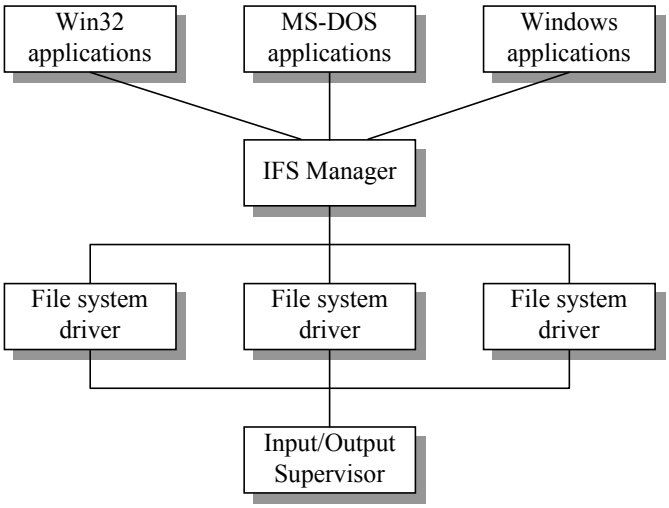


圖 16-1　Installable File System 的大體架構

使用 IFS Services

IFS Manager 提供了許多針對 FSDs 設計的 services，其他一些 services 則略爲一般化些。許多 IFS services 涉及一個無所不在的 control block，稱爲 *ioreq*，其中記錄了某一特定之檔案系統需求（file system request）的相關資訊。在這麼些 services 之間，比較一般化、可被 VxD 使用的是 *IFSMgr_Ring0_FileIO*，允許 VxD 直接對一個本機（local）或遠端（remote）檔案做動作。另一個 service 是 *IFSMgr_InstallFileSystemApiHook*，允許 VxD 攔截（並修改 -- 如果需要的話）一個 request -- 在任何一個 FSD 看到該

request 之前。IFS Manager 也提供一組 pathname 以及 time 的轉換服務，它們主要是對 FSDs 有用，但也可能在其他狀況下有用（非只對 drivers 有用）。

IFS 的 I/O Request 結構

IFS Manager 和其附庸之 VxDs 都使用圖 **16-2** 所示的 I/O request 結構（*ioreq*）做溝通。*ioreq* 結構的一個關鍵特性就是，它的許多欄位可以被數個不同欄位覆蓋（overlaid，譯註），完全視此結構的用途而定。這個結構中的同一部份有時候被同時用於某一 request 的輸入和輸出，並可能採用不同的名稱。在這樣的情況下，你必須小心，千萬不要在使用一個 input 數值之前，就把一個 output 數值儲存到佔用相同空間的 output 欄位去。各式各樣的欄位有著以下的用途：

> 譯註：這裡所謂的覆蓋（overlay），是指同一位置代以不同的欄位名稱及意義。

ir_length　offset 爲 0 的這個 32 位元欄位，主要用途是記錄 read request 或 write request 的資料長度。在那些伴含 pathnames 的呼叫中（例如 *FS_FindFirstFile*），此一欄位以 *ir_attr* 覆蓋，表示這個欄位內含一個「檔案屬性 mask」。*FS_ConnectNetResource* request 則使用此一欄位做爲一個 output 參數（此時欄位名稱改爲 *ir_pathSkip*），表示輸入的路徑名稱中有多少元素用來指定遠端資源（remote resource）。

ir_flags　這個 8 位元欄位通常內含一個子機能（subfunction）代碼。例如，在一個 *FS_FileAttributes* request 中，*ir_flags* 用來區分 "get attributes" 和 "set attributes" 兩種子機能。這個欄位也內含某些 requests 的 flag 值。

ir_user　這個 8 位元欄位內含一個 file system request 的 user ID。Network redirector 可以利用此一欄位來區分使用者。Local FSD 並不使用此一欄位。「覆蓋欄位」*ir_drivenum* 定義於 IFS.H 之中，但並未在任何地方設定。

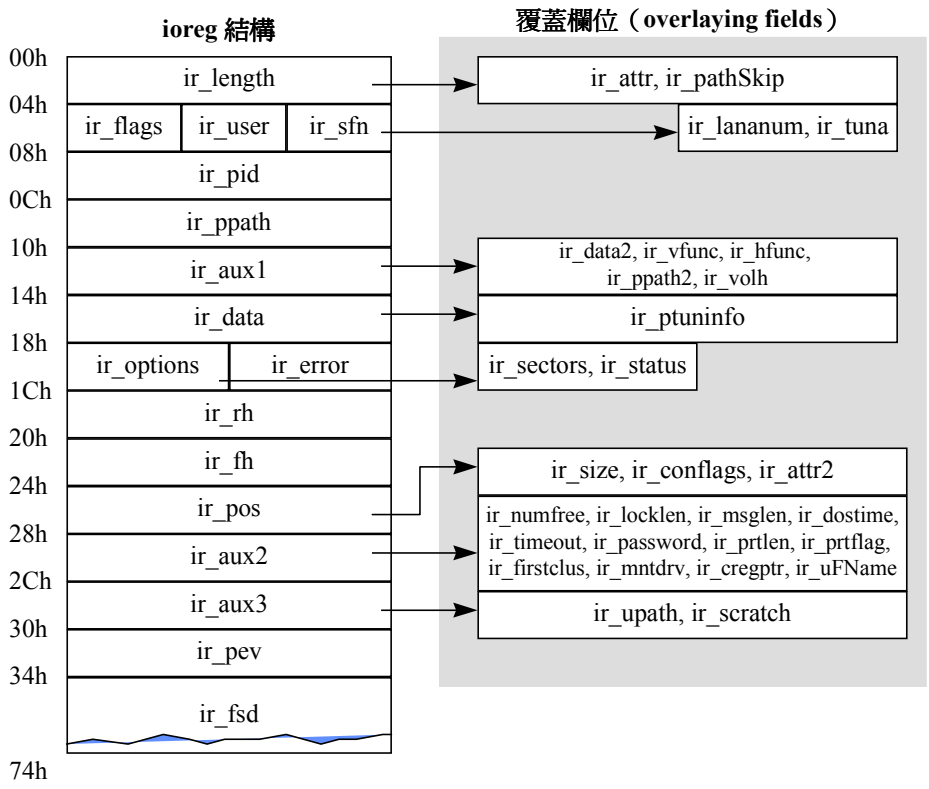


圖 16-2 IFS 的 I/O request 結構

ir_sfn 這個 16-bit 欄位內含一個 system file number，被 IFS Manager 內部用來做爲一個全域性的 file handle。FSDs 似乎並未根據此值做任何事情。某些網路導向的 requests 則以 *ir_lannum* 之名使用此一欄位，用以指出一個 LAN adapter number。一個 volume mounting request 則以 *ir_tuna* 之名使用此一欄位，做爲一個輸出參數，表示 FSD 支援 "tunneling"。當我討論如何更改一個檔案的名稱時，我會解釋 "tunneling" 的意義。

ir_pid 這個欄位指出產生出目前之 request 的那個 process。FSD 應該只把此欄位當做是一個獨一無二的 process 識別碼，不應該對它有任何其他臆想。

ir_ppath 這個欄位指向一個被解析（parsed）過的 pathname 結構。稍後我會討論所

謂的 "parsed pathnames"。

ir_aux1　這個多重目的的欄位有下列用途：

- 如果以 *ir_data2* 之名，它是一個指標，指向輸出緩衝區（針對一個 *FS_TransactNamedPipe* call）
- Volume mounting request 以 *ir_volh* 之名使用此一欄位，做爲一個輸入參數，用以指定 IOS volume request 參數結構，並以 *ir_vfunc* 之名使用此欄位做爲一個輸出參數，以提供 IFS Manager 一個表格位址，此表格乃由 volume-oriented requests 處理函式所構成。
- *FS_OpenFile* 和 *FS_FindFirstFile* requests 均以 *ir_hfunc* 之名使用此欄位，做爲一個表格位址，此表格係由「handle-oriented request 函式指標」構成，由 FSD 填寫。
- *FS_RenameFile* request 以 *ir_ppath2* 之名使用此欄位，指出檔案的新名稱（已被解析過，parsed）。

ir_data　*ir_data* 欄位的主要用途是做爲一個資料緩衝區指標。*FS_OpenFile* 和 *FS_RenameFile* requests 則以 *ir_ptuninfo* 之名使用此欄位，指向 tunneling 資訊。

ir_options　這個 16-bit 欄位通常內含一個 request 的相關 flag bits。*FS_GetDiskInfo* request 以 *ir_sectors* 之名在輸出時使用它，表示在每一個 cluster 之中有多少 sectors。Named pipe calls 則以 *ir_status* 之名使用它，提供狀態回饋資訊。

ir_error　FSDs 在輸出時設定 *ir_error* 欄位，使它成爲一個標準的 MS-DOS 錯誤碼。0 表示正常完成。FSD 函式也傳回相同的數值，做爲一個整數回返值，放在 EAX 暫存器中。IFS Manager 有時候會使用該回返值，有時候會使用 *ir_error* 值，來識別對一個 FSD 的函式呼叫的結果，所以你必須小心地既傳回一個代碼，同時也設定 *ir_error* 欄位。由於 IFS Manager 和 IOS 所使用的錯誤代碼有一些不同，有時候你必須使用 *IOSMapIORSToI21* 或 *IOSMapIORSToI24*，將一個 IOS 錯誤代碼轉換爲 *ir_error* 值。

ir_rh　大部份 requests 使用 *ir_rh* 欄位做爲一個輸入參數，指向 FSD 所使用的某內部

資料結構（用來追蹤一個 logical volume）。不過 Volume mount request 卻是在輸入時使用此一欄位，指向 MS-DOS 的 disk parameter block（DPB）串鏈。「設定 *ir_rh* 爲內部控制區塊位址」是 FSD 成功處理一個 volume mount 的程序中的一部份。

ir_fh Handle-based requests（也就是說那種根據 handle 而非 path 來處理已開啓檔案的 requests）以此欄位做爲一個輸入參數，指向「FSD 用來追蹤 logical volume 所含檔案」之內部結構。FSD 會在一次成功的 *FS_OpenFile* request 或 *FS_FindFirstFile* request 之後供應此值。

ir_pos Read 和 write requests 使用此欄位來指出一個檔案位置。*FS_ConnectNetResource* requests 以 *ir_conflags* 之名使用它，做爲 connection flags。*FS_RenameFile* requests 以 *ir_attr2* 之名使用它，做爲第二組檔案屬性；這些屬性左右了對目標名稱的對待態度。這個欄位也被 *IR_FSD_MAP_DRIVE* request 當做一個未公開的輸入參數，用來對一個 mapped volume 指示一個新的 "original" drive 編號。

在 IFS 的原始設計中，檔案的開啓與截短等等 requests 是以 *ir_size* 之名使用此一欄位，用以指出檔案大小。然而檔案截短之功能終究是取消了，現在我們截短檔案的作法是搜尋適當的偏移位置，寫入 zero bytes。所以，此欄位已不再以 *ir_size* 的角色出現。

ir_aux2 這個多重目的的欄位有下列用途：

- 當做 *ir_numfree*，用以記錄 free clusters 的個數（從 *FS_GetDiskInfo* request 輸出獲得）。
- 當做 *ir_locklen*，用以記錄 region locking request 中的 record 長度。
- 當做 *ir_msglen*，用以記錄送往 named pipe 或 mailslot 的訊息的長度。
- 當做 *ir_dostime*，用以記錄日期和時間值（MS-DOS 格式）。
- 當做 *ir_timeout*，用以記錄 timeout 值。
- 當做 *ir_password*，用以指向 *FS_ConnectNetResource* request 中的密碼。
- 當做 *ir_prtlen*，用以記錄印表機設定字串的長度。

- 當做 *ir_prtflag*，用以記錄各式各樣的印表機 flags。
- 當做 *ir_firstclus*，用以指向檔案的第一個 cluster。此值是 *FS_FindFirstFile* 和 *FS_SearchFile* 的一個可有可無的輸出，只對 VFAT 有用。
- 當做 *ir_mntdrv*，用以記錄新被裝載（mounted）之 volume 的磁碟機代碼。
- 當做 *ir_cregptr*，用以指向 client register structure，內含 IOCTL 參數。
- 當做 *ir_uFName*，用以指向「含大小寫」的 Unicode 名稱（與某些 requests 有關聯）。許多情況下，FSD 會將此名稱用於一個新的或重新命名的檔案。

ir_upath　這個欄位指向未解析（unparsed）的 Unicode 路徑名稱，與那些擁有路徑名稱之 requests 有所關聯。此欄位雖然是 *ir_aux3* 的一個別名，不過在此位置上其實沒有其他用途。

ir_pev　在「處理 IFS event 的過程內」所發生的一個 request 之中，*ir_pev* 欄位指向用以描述此 event 的一個 event 結構。至於對 IFS event 排程機制的任何細部推敲，已逾越了本書範圍。

ir_fsd　這個 64-byte 欄位可被 FSD 用於任何用途。

千萬不要嘗試配置你自己的 *ioreq* 區塊，IFS Manager 會爲這些結構維護一個內部「大池塘」，它們會比你我宣告的結構還要長。IFS Manager 會自行運用此一空間。

只有一種情況，你需要明白地釋放一個 *ioreq* 區塊。你可以使用 *IFSMgr_SchedEvent* 和 *IFSMgr_QueueEvent* 來對 timeout event 排程，IFS Manager 會傳遞一個 *ioreq* 給你的 event callback 函式，使你得以回呼 IFS Manager。當你已經處理完這個 event，你應該呼叫 *IFSMgr_FreeIoreq* 釋放此一 *ioreq*。不要在任何其他場合或爲了其他目的而使用 *IFSMgr_FreeIoreq*。

Ring0 檔案動作

擁有一個 ring0 檔案系統的主要優點是，它允許 VxDs 處理磁碟檔案就像一般應用程式那麼容易。爲了認識這個優點，請想想，在前一版 Windows 中 VxDs 需要什麼，才能夠處理一個磁碟檔案。打開檔案倒還簡單：你可以呼叫 VMM 的 *OpenFile* service，獲得一個 MS-DOS file handle。從這裡開始，你有兩個選擇。你可以在一個 nested V86 execution block 中執行 INT 21h，或是依賴 *Exec_VxD_Int* 和內建的 INT 21h translation services 來爲你處理 nested execution 技術（註）。不論哪種方法，你必須進入 V86 模式中執行部份 MS-DOS。於是，只要你打算進行檔案 I/O，你就得注意是否在 MS-DOS 中觸發了一個不合法的遞迴動作（recursion）。

> 林偉博註：Windows 95 上的 Win16 程式和 DOS 程式，係利用軟體中斷 INT 21h, INT 25h, INT 26h 來使用 IFSMgr services，而 Win32 程式及 VxD 則是利用 *Exec_VxD_Int* 和內建的 INT 21h translation services 達到相同目的。

IFSMgr_Ring0_FileIO service 實作出類似於我們所熟知並喜愛的「MS-DOS 檔案系統呼叫」的東西（表 **16-1**）。這個 service 是 "register-based"。要使用它，你必須在 EAX 暫存器中設定一個子機能代碼，例如 *R0_OPENCREATFILE*，然後在其他的 general 暫存器（譯註）設定參數（視你的子機能而不同）。一如其 MS-DOS 兄弟一樣，*IFSMgr_Ring0_FileIO* service 會清除或設立 carry flag，用以表示成功或失敗。如果呼叫成功，它會清除 carry flag 並利用 general 暫存器傳回結果。如果有錯誤發生，它就設立 carry flag 並利用 AX 暫存器傳回錯誤代碼。這個 service 通常使用與 32 位元版 MS-DOS 所使用的一樣的 general registers 來放置相同的參數和執行結果，不過也有例外。

> 譯註：所謂 general 暫存器是指 EAX, EBX ...，相對則有所謂的 segment 暫存器如 DS, ES ...。請回頭看看圖 **5-7**。

子機能（Subfunction）	說明
R0_CLOSEFILE	關閉一個檔案
R0_DELETEFILE	刪除一個檔案。此機能等同於 INT 21h, function 7141h（這是 INT 21h, function 41h 的長名稱）。和 INT 21h, function 7141h 一樣，此機能接受一個 "file-search attribute mask"，用以制止 "wildcard matching"。
R0_FINDCLOSEFILE [sic]	關閉一個 find-file handle. 此機能等同於 INT 21h, function 71A1h，它並沒有真實模式下的對等品。
R0_FINDFIRSTFILE	找出第一個吻合的檔案。此機能等同於 INT 21h, function 714Eh（這是 INT 21h, function 4Eh 的長名稱）。
R0_FINDNEXTFILE	找出下一個吻合的檔案。此機能等同於 INT 21h, function 714Fh（這是 INT 21h, function 4Fh 的長名稱）。
R0_FILEATTRIBUTES	取得或設定檔案屬性。此機能等同於 INT 21h, function 7143h（這是 INT 21h, function 43h 的長名稱）。
R0_GETFILESIZE	決定檔案的大小。此機能等同於 INT 21h, function 4202h，只不過此機能並不設定檔案位置。
R0_GETDISKFREESPACE	決定一個 logical volume 的自由空間。此機能等同於 INT 21h, function 36h。
R0_FILELOCKS	鎖定(或解除鎖定)某個檔案。此機能等同於 INT 21h, function 5Ch。
R0_OPENCREATFILE 和 R0_OPENCREAT_IN_CONTEXT	打開或產生一個檔案。這些機能等同於 INT 21h, function 716Ch（這是 INT 21h, function 6Ch 的長名稱）。
R0_READFILE 和 R0_READFILE_IN_CONTEXT	讀取檔案。這些機能等同於 INT 21h, function 3Fh，不過你可以提供一個 32 位元的讀取量和 32 位元的檔案位置。
R0_READABSOLUTEDISK	從一個 logical volume 中讀取 sectors。此機能等同於 INT 25h。
R0_RENAMEFILE	重新命名某個檔案。此機能等同於 INT 21h, function 7156h（這是 INT 21h, function 56h 的長名稱）。
R0_WRITEABSOLUTEDISK,	將 sectors 寫到一個 logical volume 中。此機能等同於 INT 26h。
R0_WRITEFILE and R0_WRITEFILE_IN_CONTEXT	寫入一個檔案。此機能等同於 INT 21h, function 40h，不過你可以提供一個 32 位元的寫入量和 32 位元的檔案位置。

表 16-1　IFSMgr_Ring0_FileIO 的子機能（subfunctions）

使用 IFSMgr_Ring0_FileIO

假設你已經開啓一個檔案，打算寫入字串 "Hello, world!"。如果以 assembly 語言來寫程式，可以這麼做：

```
handle  dd     0              ; from R0_OPENCREATFILE call
string  db     'Hello, world!'
lstring equ     $-string
filepos dd     0              ; position in file
...
       mov    eax, R0_WRITEFILE
       mov    ebx, handle
       mov    ecx, lstring
       mov    edx, filepos
       mov    esi, offset string
       VxDCall IFSMgr_Ring0_FileIO
       jc     error           ; skip if error
       cmp    eax, ecx        ; or if less than all bytes
       jne    error           ;   got written
       add    filepos, ecx
```

除了需要自己追蹤檔案位置，並使用 ESI 取代 EDX 做爲資料指標之外，這段碼十分類似你在真實模式中和 MS-DOS 打交道時的動作。

爲了提供一個 C 語言實例，我需要先建立一個 C 表頭檔，符合 DDK 中的 IFSMGR.INC。我決定建立一組 inline 函式，用以簡化對 *IFSMgr_Ring0_FileIO* 的呼叫（你可以在書附光碟的 \CHAP16 的任何子目錄中找到我的 IFSMGR.H 檔）。完整的碼（使用我的 IFSMGR.H，產生一個檔案，內含字串 "Hello, world!"）看起來像這樣：

```
DWORD hfile;
int code;
DWORD action;
static unsigned char *data = "Hello, world!";
DWORD nwritten;

code = R0_OpenCreatFile(FALSE, ACCESS_READWRITE |
    SHARE_DENYREADWRITE, 0, ACTION_CREATEALWAYS, 0,
     "HelloWorld.txt", &hfile, &action);
if (code == 0)
```

```
{                    // file opened okay
R0_WriteFile(FALSE, hfile, strlen(data), 0, data, &nwritten);
R0_CloseFile(hfile);
}                    // file opened okay
```

由於我沒有指定完整路徑，這個實例（置於書附光碟的 \CHAP16\FILEIO 目錄）在 Windows 磁碟目錄中開啓一個名爲 HELLOWORLD.TXT 的檔案。你也可以指定一個完整路徑。*R0_OpenCreatFile* 的各種 flag 參數（如 *ACCESS_READWRITE* 等等）是 IFS.H 中定義的常數，直接轉換爲它們在 MS-DOS 中的對等值。第一個參數，本例爲 FALSE，表示我們是在 global context 中打開此檔案；稍後我將進一步討論 file context 的觀念。

我的 *R0_OpenCreatFile* 函式傳回一個 MS-DOS 錯誤代碼。傳回值若爲 0，表示成功。你可以使用 Win32 表頭檔 WINERROR.H 中的常數來測試其他的傳回值。例如 *ERROR_PATH_NOT_FOUND* 意思是你提供了一個其實並不存在的磁碟目錄。成功開啓檔案之後，*hfile* 內含一個 ring0 file handle，你可以在後繼動作中使用之。變數 *action* 告訴你此函式究竟是產生或開啓一個檔案，其值與一個 MS-DOS open call 的傳回值相同。

如果 open call 成功，上例接著就呼叫 *R0_WriteFile*，將資料寫入檔案的 offset 0 處，並呼叫 *R0_CloseFile* 關閉檔案。兩個動作都使用由 *R0_OpenCreatFile* 傳回的檔案 handle。如果你有一個檔案 handle，是以其他方法從應用程式手中獲得，或是經由直接呼叫 MS-DOS 而得，你不能夠把它應用於 ring0 的檔案 I/O 動作之中，你只能使用 *R0_OpenCreatFile* 所傳回的檔案 handle。

比較精緻的作法是，實作一個 Win32 file API 函式，當做是 ring0 函式，來觸發 *IFSMgr_Ring0_FileIO* calls。然後你就可以直接在你的 VxD 中呼叫這個 Win32 file API 函式。如果你雄心勃勃，甚至可以使用你的 Win32 file APIs 做爲一個一般的 C 函式庫底層，那麼你就可以在你的 VxD 中使用 *fopen* 或 *_open* 等函式了。

只有當 IFS Manager 已經完成其初始化程序中的 *Device_Init* 階段，你才能夠執行 ring0 檔案動作。如果你的初始化次序晚於 IFS Manager （對一個 FSD 而言通常如此），你應該等待，直到收到 *Device_Init* 訊息爲止。如果你的初始化次序較早，你應該等待，直

到收到 *Init_Complete* 訊息爲止。如果你的 driver 是一個 dynamic VxD，請使用 *VMM_GetSystemInitState* 來驗證是否系統已經到達 *Device_Init* 階段（如果你知道你的 driver 是被一個 component 載入，而該 component 在 IFS Manager 之後才初始化的話），或到達 *Init_Complete* 階段（如果你知道你的 driver 是被一個 component 載入，而該 component 在 IFS Manager 之前初始化；或如果你不知道該 component 的初始化次序）。

Global Contexts 和 Thread Contexts

IFSMgr_Ring0_FileIO 的 open, read, write 子機能都有兩個變種：一個是 global context 版本，另一個是 thread context 版本。指定 thread context 的辦法有兩種，一是在 assembly call 中使用 *XXX_IN_CONTEXT* 版的子機能，一是以 TRUE 做爲我的 C 語言外包函式的第一個參數。指定 global context 的辦法也有兩種，一是在 assembly call 中使用原來的子機能名稱，一是以 FALSE 做爲我的 C 語言外包函式的第一個參數。

如果你在 thread context 中打開一個檔案，目前執行中的 process 會擁有其 handle。IFS Manager 不允許其他任何 process 使用此 handle 來存取這個檔案。此外，你應該使用 thread-context 版的 *R0_READ_FILE* 和 *R0_WRITE_FILE* 來處理此檔案。如果你正與某些應用程式一起工作，特別是如果你必須使用 process 的私有記憶體空間來讀寫資料，你應該使用這三個 API 函式的 thread-context 版。

如果你於 global context 中打開一個檔案，任何 process 都可以使用這個 handle。這種情況下，使用 *XXX_IN_CONTEXT* 版的 *R0_READ_FILE* 和 *R0_WRITE_FILE* 就是一種錯誤。一個 VxD 如果是爲了自己的需要（而非爲了應用程式的需要）而存取檔案，通常會使用 global context，以避免被限制於在某個 process 中才能執行動作。

爲什麼需要兩個版本的 R0_READ_FILE 和 R0_WRITE_FILE？ 我很驚訝爲什麼 read 和 write 動作有所謂的 global context 和 thread context 版。因爲 IFS Manager 實施一個規則，規定你在這些動作中必須使用和開檔時一樣的 context。畢竟，如果 IFS Manager 知道如何抱怨，它也應該知道如何選擇正確的 context 才是。結果證明，系統在 ring3 file

handles 之上執行 ring0 讀寫動作，以獲得最佳效率。區別這些讀寫動作，需要不同的動作碼（operation codes）。

字元轉換與路徑名稱

IFS Manager 內部使用 Unicode 路徑名稱，雖然 Windows 95 本身在應用程式的層面上並不如此。如果你發出一個 MS-DOS 命令，像是 TYPE F:\CHAP16\LPFS\FILESYSTEM.CPP 之類，IFS Manager 會把路徑名稱轉換爲 Unicode，然後才把它交給任何可能的 file system drivers。IFS Manager 提供數個轉換用的 services（表 **16-2**），讓 VxD 撰寫者得以輕鬆地將字串在 Unicode 和 "base" 字元集（BCS，Base Character Set，使用於 Windows 應用程式）之間做轉換。

Service	說明
UniToBCS	將一個 Unicode 字串（不是一個 parsed 路徑名稱）轉換爲一個 BCS 字串
BCSToUni	將一個 BCS 字串轉換爲一個 Unicode
BCSToBCS	將一個 BCS 字串轉換爲另一個 BCS 字串（但使用不同的 code page）
UniCharToOEM	將一個 Unicode 字元轉換爲一個 BCS 字元（使用 OEM code page）
UniToUpper	將一個含有大小寫的 Unicode 字串轉換爲全部大寫

表 16-2　做爲路徑名稱轉換用的 IFS services

這些轉換函式並未組成完整的轉換功能網。例如，你就無法將 Unicode 或 BCS 字串轉換爲全部小寫，或是將單一的 Unicode 字元直接轉換爲 ANSI code page。表 **16-2** 所列者爲 IFS Manager 將使用者（及程式）的輸入轉換爲 Unicode 字串（以便給 file system drivers 所用）時的必須品。如果我發出稍早提過的 MS-DOS TYPE 命令，你可以想像 IFS Manager 的兩個步驟：首先呼叫 *BCSToUni* 將檔名轉換爲 Unicode，然後呼叫 *UniToUpper* 再將它轉換爲全部大寫。舉個例子，IFS Manager 可能會這麼做（我並未拆

解 IFS Manager 觀察是否真是如此，但我確信我無法如此近距離地找到所有步驟）：

```
USHORT uniname[MAX_PATH];
PBYTE *oemname = Client_Ptr_Flat(DS, DX);
UINT len = strlen(oemname);
UINT nbytes = BCSToUni(uniname, oemname, len, BCS_OEM);
UniToUpper(uniname, uniname, nbytes);
```

在這程式片段中，我們從一部虛擬機器手中取得一個指標，放在 DS:DX 暫存器中（這正是一個 MS-DOS INT 21h, function 716Ch call 所期望的地方），視之爲一個字串並使用 OEM code page。*CSToUni* 將它轉換爲 Unicode，而 *UniToUpper* 將結果再轉換爲全部大寫。

註注意資料型態 如果你習慣 Win32 程式設計，你可能期望看到 Unicode 字串在宣告時使用如 WCHAR 或 wchar_t 之類的型別。但 DDK 總是使用 USHORT。

如果你打算將結果的 Unicode 字串顯示於除錯終端機上（它可能使用 OEM character set），你就必須先使用 *UniToBCS* 轉換之：

```
BYTE bcsname[MAX_PATH];
int len = UniToBCS(bcsname, uniname, nbytes,
   sizeof(bcsname)-1, BCS_OEM);
bcsname[len] = 0;
```

這個 service 並不會自動加上一個 null 結束符號到回傳的字串身上，通常你得自己來，像上面那樣。

這裡有一個實例，示範一個在 Microsoft Developer Studio 中使用 Windows ANSI 字元集建造起來的 VxD，是如何地將一個字串常數轉換爲 OEM 字元集：

```
BCSToBCS(bcsname, "Grüβ Gott", BCS_OEM, BCS_WANSI,
   sizeof(bcsname));
```

（Grüβ Gott 是德國巴伐利亞省的人對 "Hello, world!" 的發音。隨便打幕尼黑（巴伐利亞首府）的任何一個電話號碼，都可以證明此點）

下面是字元轉換的最後兩個例子：

```
UniCharToOEM(0x00FC);
UniCharToOEM(0x015C);
```

假設 code page 437（MS-DOS Latin US）是目前的 OEM code page，上述第一個例子將 Latin 字元ü（那是 Unicode 中的 00FCh）映射爲其 code page 437 的對等物（81h）。第二個例子企圖將 Unicode 字元（S）映射至 code page 437。由於這個映射是不可能的（在 code page 437 中並無這樣的字元），所以函式會傳回 5Fh，那是 ASCII 中的底線（underscore）字元。

路徑名稱的解析（Pathname Parsing）

IFS Manager 不只提供你 Unicode services，它還爲 file system drivers 提供另一組額外的 services：在呼叫 file system drivers 之前，它把每一個路徑名稱以 Windows 95 的規則加以解析。如果你曾經寫過程式，雜七雜八地呼叫 *_splitpath* 及其他字串處理函式，你立刻就會以感恩的心情來看待這種服務帶來的恩惠。**圖 16-3** 顯示出一個例子，告訴你路徑名稱的解析結構。

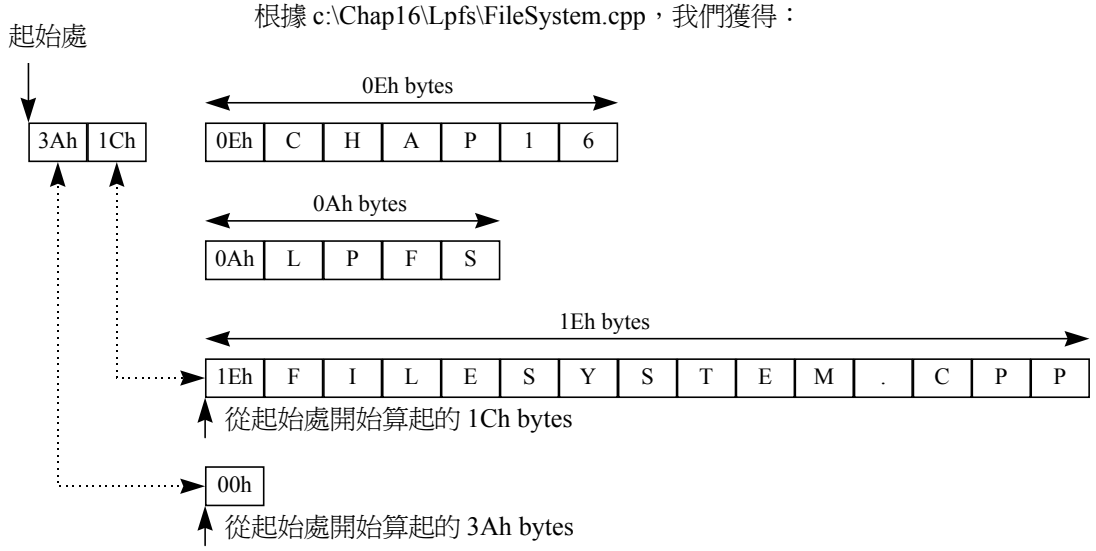


圖 16-3　一個解析後的路徑名稱

ParsedPath 結構以一個 WORD（*pp_totalLength*）開始，表示被解析之名稱及其表頭的總長度（bytes）。第二個 WORD（*pp_prefixLength*）表示結構中的最後一個 *PathElement* 元素項的偏移位置（也是 bytes），此元素項表示路徑名稱的最後一個元素。表頭之後，是一系列的 *PathElement* 結構，每一個代表路徑名稱中的一個目錄階層。每一個元素以一個 WORD 開始（*pe_length*），表示元素長度（包括內含長度值的那個 WORD）並內含 0 個或一個以上的 Unicode 字元。整個結構的最後放置著一個 00h WORD（不包含在 *pp_totalLength* 內）。請注意，所有的計量和偏移值單位都是 bytes 而非 characters。

IFS.H 內含一些巨集，對於路徑名稱的處理十分有用：

- *IFSPathSize* 傳回一個 *ParsedPath* 結構的大小，包括 null 結束符號。
- *IFSPathLength* 傳回一個 *ParsedPath* 結構中的所有 *PathElements* 的總長度。也就是說，它拿 *pp_totalLength* 值減去 *ParsedPath* 表頭長度。
- *IFSLastElement* 傳回一個指標，指向 *ParsedPath* 結構中的最後一個 *PathElement*。這最後一個元素表現的是路徑名稱中的檔名。

- *IFSNextElement* 傳回某個 *PathElement* 之後的下一個 *PathElement* 位址。
- 如果一個 *ParsedPath* 結構表現出根目錄的話，*IFSIsRoot* 傳回 TRUE。如果在路徑名稱中沒有 *PathElement* components，就可以斷定此事，於是 *pp_totalLength* 欄位等於 4，也就是 *ParsedPath* 表頭大小。

你幾乎絕不會需要自己解析一個路徑名稱。在一個 file system driver 的 request 處理常式中，如果經由一些 private API 函式（而非經由 IFS Manager 本身）收到一個路徑名稱（以 base character set 表示），你可以呼叫 *IFSMgr_FSDParsePath*，這個 service 需要一個指標，指向一個 *ioreq*，但你不能夠自己配置 *ioreq* 區塊。舉個例子，假設你使用 *IFSMgr_SetReqHook* 來 "hook" 一個 private INT 21h call，而它包含一個路徑名稱。IFS Manager 完全不知道你的 private 函式，所以也就不知道這個路徑名稱需要被解析。由於你的 hook 函式獲得控制權時連帶有一個 *ioreq* 區塊，所以它可以呼叫 *IFSMgr_FSDParsePath* 來解析這個路徑名稱。除了這個不平常的情況外，你總是會從 IFS Manager 身上獲得一個預先解析好的路徑名稱。

我已經說過，*UniToBCS* service 用來將一個 Unicode 字串轉換爲 base character set。另有一個類似的 service 名爲 *UniToBCSPath*，用來將一個被解析過的路徑名稱轉換爲 base character set。但這次你獲得的是一個指標，指向「放置於 *ioreq* 結構之 *ir_ppath* 欄位」中的被解析路徑名稱。你可以這樣轉換（用於診斷性列印或用於其他目的）：

```
BYTE bcsname[MAX_PATH];
int len = UniToBCSPath(bcsname, pir->ir_ppath->pp_element,
    sizeof(bcsname)-1, BCS_OEM);
bcsname[len] = 0;
```

UniToBCSPath 在被解析的路徑名稱中來回移動，直到到達 null 結束符號。對於每一個元素，它安插一個倒斜線（backslash），後面跟著該元素的 base-character-set 版。雖然這個函式的技術文件中聲稱，它傳回的並不是一個以 null 爲結束符號的字串，但它事實上的確加了一個 null 結束符號，除非這個符號使你的緩衝區滿溢（這一點可以偵測得

到，因爲此時傳回的長度將等於你的緩衝區大小）。此外，我所提供的長度，*sizeof(bcsname) -1*，保留了一個字元，以便加上 null 結束符號（其實 *UniToBCSPath* 函式也會加上）。我所描述的行爲並不是公開的，你我都應該使用公開的函式，並如上述程式碼那樣使用，雖然那比較不方便。

如何從解析過的路徑名稱中的一個元素前進到另一個元素，唔，十分明顯。不明顯的是，對每一個元素該做些什麼處理。IFS Manager 內含有字元轉換之外的一些 services，用來簡化常見於路徑元素上的一些動作（表 **16-3**）。

Service	說明
CreateBasis	將一個 Unicode path element 映射爲一個 basis name（一個 8.3 名稱，沒有句點）
MatchBasisName	將一個 basis name 和一個檔案控制區塊（file control block，FCB）格式名稱比較
AppendBasisTail	在一個 basic name 之後附加一個獨一無二的識別整數
FcbToShort	將一個 FCB 格式的名稱轉換爲 Unicode 8.3 格式
ShortToFcb	將一個 Unicode 8.3 格式的名稱轉換爲 FCB 格式
ShortToLossyFcb	將一個 Unicode 8.3 格式的名稱轉換爲一個 FCB 格式名稱，並只使用 OEM 字元

表 16-3 用來處理路徑元素的各個 IFS services

FCB 名稱和 Basis 名稱

表 **16-3** 中所謂的 FCB 格式有 11 個字元：8 字元的主檔名和 3 字元的副檔名。如果必要，每一個元素後會附加空白，以填滿其空間配額。因此，FUBAR.TXT 的 FCB 格式是 FUBAR　TXT（主檔名和副檔名之間有三個空白）。這些字元是 Unicode 格式，所以此名稱總共長度是 22 bytes。

所謂 basis 名稱，是一個 Unicode FCB 格式名稱，根據一組標準規則，將一個路徑元素的長名稱調整爲 8-byte 主名稱和 3-byte 副名稱。

所謂 8.3 名稱又稱爲是「短」名稱，內含最多 8 個字元的主名稱、一個句點、以及最多 3 個字元的副名稱。「短」名稱中並沒有任何附加空白。例如：

```
USHORT uniname[MAX_PATH];
USHORT name_8_3[12];
USHORT fcbname[11];

BCSToUni(uniname, "FileSystem.cpp", 14, BCS_WANSI);
CreateBasis(fcbname, uniname, 28);
AppendBasisTail(fcbname, 1);
FcbToShort(name_8_3, fcbname, FALSE);
```

在此程式片段中：

- *CreateBasis* 根據 *FileSystem*.cpp 產生出 FILESYSTCPP。
- *AppendBasisTail* 根據 FILESYSTCPP 再加上一個數值尾巴 1，產生出 FILESY~1CPP。
- *FcbToShort* 根據 FILESY~1CPP 產生出 FILESY~1.CPP。

毫無疑問你可以看得出來，這些步驟正是一個 VFAT 相容檔案系統根據一個長檔名製造一個 8.3 別名的過程。

日期和時間的轉換

自從所有的計算工作都以批次模式在那種可與其他機器交談的機器上完成，日期（date）和時間（time）的管理，變成了一件日益重要的事情。區域（local-）和廣域（wide-）網路的降臨，以及國際間數位資訊的交換，已經產生了一種對於精確時間的需求。一個例子就是軟體開發。大部份軟體開發者使用桌上系統，彼此再以一個 LAN（區域網路）連接起來。每一個系統有自己的 clock 以及相關軟體（維護 clock 的運作）。專案開發人員跨越數個時區並不是很罕見的事，那麼，標示爲 02:25 的你的檔案，比起標示爲 02:26 的我的檔案，究竟是比較新或比較舊呢？（我所舉的時間正是許多程式設計者的正常工作時間呵！）如果沒有對 PC clock 的設定以及地球時區做追蹤，任何 MAKE script 或版本控制系統就沒有辦法回答這個問題。這個問題將視我們是否在相同時區、我們的 clocks 有多精準、以及其他因素而定。

IFS 提供三種不同風貌的時間資料。一個是 MS-DOS 格式，已經被使用超過一個年代，以 32 位元記錄時間，如圖 16-4。時間每 2 秒被表現出來，總是以當地時區爲準。這個格式只能夠表現公元 1980~2107 的時間。FAT 檔案系統使用此格式來記錄檔案的最後更新時間。

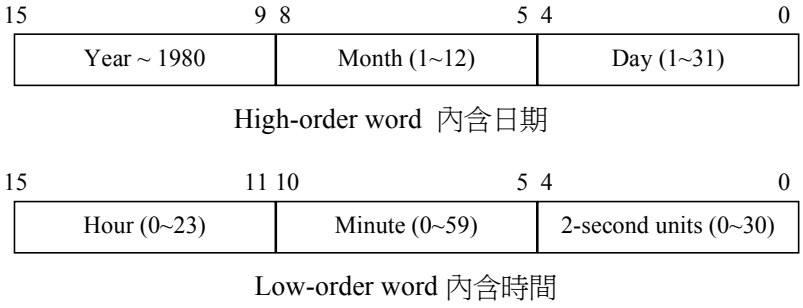


圖 16-4 時間標示法：MS-DOS 格式

Windows NT 率先使用座標化宇宙時間（UTC），以 64 位元表現出自從 1601 年 1 月 1 日（格林威治時間，GMT）後的每 100 奈秒（nanosecond）。如果你對年曆有點認識，上述原點的選擇應該甚爲明白：四百年爲一週期，最後以一個閏年結束。如果你同時對年曆和歷史都知道一點，你可能會想到，這個選擇和一顆蘋果打到牛頓（Isaac Newton）有點關聯。不論如何，你可能會對 100 奈秒（nanosecond）的精準度感到不可思議。如果國際間對於太空探險行動仍然保持在目前的低層次狀態，我們未必會對於誇張的所謂「宇宙」時間感到失望。然而我預期，以奈秒（nanosecond）爲單位，不久就會顯得太過粗糙！

雖然 FAT 檔案使用 MS-DOS 格式而 Win32 APIs 使用 Windows NT 格式，網路使用另一種不同的格式：所謂的網路時間。一個網路時間係使用 32 位元來記錄自從 1970 年 1 月 1 日（格林威治時間）後的每 1 秒。UNIX 及其衍生系統也使用此一格式。人們有時候會把這種時間格式也稱爲是「以座標化宇宙時間（UTC）爲基礎」（意味著 GMT）。這倒使得 Win32 64 位元格式和網路時間所關連的時區混淆不清了。

拿一個時間值在不同格式間做轉換，需要一些繁複的碼。IFS Manager 提供數個 services（表 **16-4**）爲你執行轉換動作。例如，要將一個 FAT 檔案時間（最後修改時間）轉換爲 64 位元 Win32 格式，你可以呼叫 *IFSMgr_DosToWin32Time*。整組函式中唯一遺漏的是「如何以 Win32 格式獲得目前時間」這個函式。通常你不需要使用由 *IFSMgr_GetTimeZoneBias* 傳回的時區，因爲對 MS-DOS 時間格式（唯一使用當地時間者）的轉換已自動考慮了時區。

Service	說明
IFSMgr_Get_NetTime	獲得目前的網路時間
IFSMgr_Get_DOSTime	獲得目前的當地時間（MS-DOS 格式）
IFSMgr_NetToDosTime	將一個網路時間轉換爲 MS-DOS 格式
IFSMgr_DosToNetTime	將一個 MS-DOS 當地時間轉換爲網路格式
IFSMgr_DosToWin32Time	將一個 MS-DOS 當地時間轉換爲 Win32 格式
IFSMgr_Win32ToDosTime	將一個 Win32 時間轉換爲 MS-DOS 格式
IFSMgr_NetToWin32Time	將一個網路時間轉換爲 Win32 時間
IFSMgr_Win32ToNetTime	將一個 Win32 時間轉換爲網路時間
IFSMgr_GetTimeZoneBias	取得當地時區的 bias（偏倚值），以分鐘爲單位。

表 16-4　用來轉換時間格式的各個 IFS services

舉個例子來示範上述 services 的用法。假設你的 FSD 剛產生出一個檔案，你要記錄目前時間以爲檔案產生時間、修改時間和存取時間。你可以呼叫 *IFSMgr_Get_DOSTime*，像這樣：

```
e.created = e.modified = e.accessed = IFSMgr_GetDOSTime();
```

如果你的 FSD 稍後需要填寫一個 *WIN32_FIND_DATA* 結構，做爲 Win32 *FindFirstFile* API 函式的一部份，你可以將你的 MS-DOS 格式時間轉換如下：

```
_WIN32_FIND_DATA* fdp;
fdp->ftCreationTime   = IFSMgr_DosToWin32Time(e.created);
fdp->ftLastAccessTime = IFSMgr_DosToWin32Time(e.accessed);
```

```
fdp->ftLastWriteTime  = IFSMgr_DosToWin32Time(e.modified);
```

在此片段之中，請注意 *_WIN32_FIND_DATA* 的前導底線。官方的 IFS.H 檔案宣告了一個 Win32 結構，也使用這個名字，使你比較容易在一個使用正規 Win32 表頭檔的編譯系統中含入 IFS.H。不幸的是，官方的 IFS.H 檔也定義了 Win32 結構 *_FILETIME* 和 *_BY_HANDLE_INFORMATION*，所以我必須爲此官方檔案加上一些條件編譯，以便本章其他專案可用。

檔案系統的 API Hooking

對於檔案系統呼叫，一個最普遍的動作就是去 "hook" 它們。我不能夠爲「干涉他人之 I/O 動作所帶來的不良衝擊」辯護（註），但是我也不能夠否定它們的功用。如果你有如此傾向，*IFSMgr_InstallFileSystemApiHook* service 可以使你成爲一個忙碌的人。你可以這樣地安裝和移除一個 API hook：

> 林偉博註：我們可以利用 *IFSMgr_InstallFileSystemApiHook* hook service，去攔截 Windows 95 下的檔案複製或讀寫動作，並在其中偷偷加入自己的額外處理。這無疑爲電腦病毒提供了一條散播捷徑。

```
ppIFSFileHookFunc prevhook;
...
BOOL OnSysDynamicDeviceInit()
    {
    prevhook = IFSMgr_InstallFileSystemApiHook(HookProc);
    return TRUE;
    }

BOOL OnSysDynamicDeviceExit()
    {
    IFSMgr_RemoveFileSystemApiHook(HookProc);
    return TRUE;
    }
```

也就是說，你在你的 VxD 初始化時安裝好 hook，在你的 VxD 結束前移除 hook。

IFSMgr_InstallFileSystemApiHook 會取得你的 hook 函式位址做爲參數之一，傳回一個 DWORD 的位址，內含指標，指向另一個 hook 函式。移除 hook 的方式是把你的 hook 函式位址傳給 *IFSMgr_RemoveFileSystemApiHook* 做爲參數。IFS Manager 內部會維護一串 API hooks，所以你和其他人都可以任意地增加或移除它們。

Hook 函式的原型如下：

```
int HookProc(pIFSFunc fsdproc, int fcn, int drive, int flags,
   int cp, pioreq pir);
```

其中的 *fsdproc* 是函式位址（坐落於某些 file system driver 或其他地方），IFS Manager 會呼叫之以實作出這特殊的 I/O request。參數 *fcn* 是一個代碼，表示 30 個可能的 I/O requests 代碼中的一個。這個代碼關係到 file system drivers 所實作的功能。*drive* 參數代表 logical volume 的磁碟機編碼（1 表示 A 碟）。*flags* 參數表示 drive 所服務的是哪一種 device。*cp* 參數（*BCS_ANSI* 或 *BCS_OEM*）表示 IFS Managers 的呼叫者傳給它的是一個 ANSI 或是一個 OEM 字串。最後的 *pir* 參數指向一個 IFS I/O request 結構，其中含有此一 request 的所有參數。

如果沒有 hook 函式存在，IFS Manager 就簡單地執行以下動作：

```
   int status = (*fsdproc)(pir);
```

也就是說，IFS Manager 將呼叫由 *fsdproc* 所指之函式，並以 I/O request 結構位址做爲參數，而把傳回值儲存在一個 status 代碼中。如果你打算透明化地讓你的 request 通過一個 hook 函式，你可能會想寫這樣的 hook 函式：

```
int HookProc(pIFSFunc fsdproc, int fcn, int drive, int flags,
   int cp, pioreq pir)
   {
   return (*fsdproc)(pir);       // don't do this
   }
```

事實上這是錯誤的！如果你可以 hook API 函式，其他的 VxDs 也可以。你必須讓其他

可能的 hooks 也有機會看到每一個 request。所以，你能夠寫的最小型 hook 函式應該是將呼叫串鏈起來，而不只是去處理它：

```
int HookProc(pIFSFunc fsdproc, int fcn, int drive, int flags,
             int cp, pioreq pir)
   {
   return (**prevhook)(fsdproc, fcn, drive, flags, cp, pir);
   }
```

本例之中，你使用 *prevhook*（當你最初呼叫 *IFSMgr_InstallFileSystemApiHook* 時所傳回）來決定串鏈之中的下一個 hook 函式，然後以收到的參數全數不變地呼叫該 hook 函式。如果沒有其他 VxD 曾經攔截（hooked）檔案系統的 API 函式，那麼你會到達一個預設的 hook 函式，在 IFS Manager 中，它會呼叫 *fsdproc* 函式，夾帶 *pir* request，因而真正執行 I/O request。

書附光碟片中的 IFSMonitor 實例示範如何使用一個 file system API hook 來監視你的系統中的 file system calls。這個實例中的 VxD 部份以上述方法 "hooks" API calls，並使用一個非同步（asychronous）串列來記錄每一個 call 的相關資訊。它使用非同步（asychronous）函式呼叫機制，將串列元素送給一個 Windows 程式（其中主要是一個 Windows 95 標準的 list 控制元件，包著一個外框），以便顯示資料。事實上，這個監視程式的大體架構和 IOS request 監視器（我在第 15 章描述過）相同。

圖 16-5 展示 request 的活動，從開啓這個監視程式到數秒後結束爲止。你所看到的是一系列的 "page read" 動作，來自檔案的不同位置，讀進一個特別的 page buffer 中。當我做這些運轉記錄時，我的系統並沒有什麼其他事情發生。

能夠對 file system API calls 做 hook 動作，是非常有威力的一項能力。已經有一些商業化病毒掃描軟體，依賴 API hook 來偵測是否感染。甚至可以跨網路唷。你也可以使用 API hook 來完成檔案壓縮或加密（encryption）。如今，既然有可能 "hook" API calls（並且，最重要的是你可以看到它們），更多應用或許會發生在我們週遭的企業界身上。

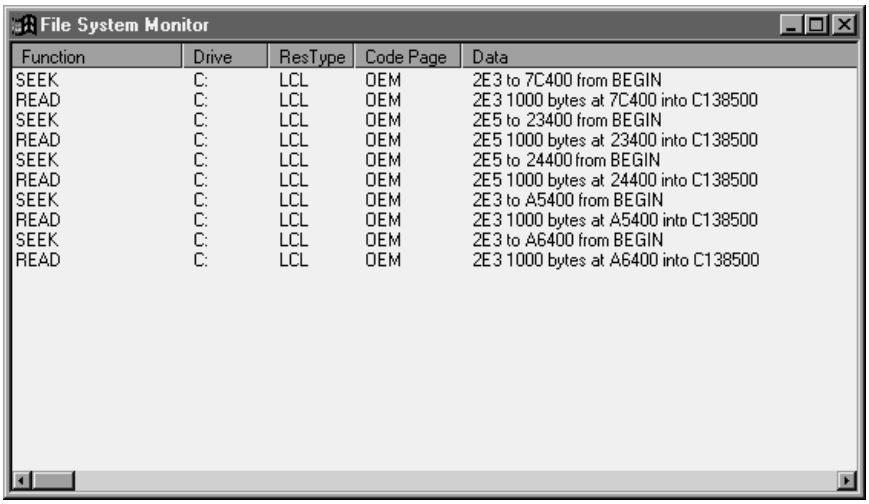


圖 16-5　IFSMonitor 程式的輸出實例

可以看到所有的 API Calls 嗎？　就在本書印刷時刻，我學到一點：一個 API hook 可能無法實際看到所有的 file system API calls。這是我從 WINSDK 論壇上看到的，上面說，一個 CD-ROM drive 如果使用一個真實模式的 MSCDEX，就能夠迴避 hook。

Local File System Driver

所謂 local file system driver（local FSD），在一個 local device 上實作出 Windows 95 的檔案系統模型。也就是說，一個 device 附著到一部「擁有 driver 安裝於其中」的電腦上頭，而不是附著到某些其他電腦，再透過網路來使用 driver。你的電腦中當然擁有 VFAT 檔案系統，VFAT 是歷史悠久的 MS-DOS FAT（file allocation table) 檔案系統的 32 位元版本，支援長檔名，及其他良好的性質。VFAT 由 VFAT.VXD 完成，那是 VMM32 中的一個標準的 VxD。你當然也擁有 VDEF（預設）檔案系統，靠著它，你才能夠處理一個空白磁碟，以便將它格式化。你或許還擁有 CD 檔案系統（CDFS），它支援 CD-ROM 設備。CD 檔案系統包括 VMM32 中的 VCDFSD 元件，以及一個 FSD 元件，像是 CDFS.VXD（被放在 IOSUBSYS 磁碟目錄中）。

簡介 Low Performance File System（LPFS）

舉一個真正的檔案系統做爲範例，實在是遠超出「合理」以及「易學」兩個範圍。我只打算表現出足夠的範例，精確描述一個檔案系統必須供應的 service 函式。我決定實作出一個十分低效率的檔案系統，給第 15 章所開發的 RAM-disk drive 使用（我可不想把我的硬碟拿來做爲實驗品）。我把最後的結果稱爲 Low Performance File System（LPFS）。VTHIN（VFAT 的對應，呵呵）應該也是個好名稱。

LPFS 有一個根目錄，內含唯一一個檔案。這個檔案可以是任何名稱。你不能夠刪除這個檔案，但你可以打開它並且讀寫之，也可以重新命名。這項功能限制使我得以不必爲了複雜的目錄（directory）結構、sector 配置...而傷腦筋並造成程式過於混亂。

我以範例 **16-6** 的 C++ class 架構來完成 LPFS。我定義了一個基礎類別名爲 *CFileSystem*，將一個 IFS 檔案系統的行爲封裝起來。從此開始，我衍生出 *CLocalFileSystem*（一個位於 local device 的檔案系統）和 *CNetworkFileSystem*（一個位於 remote device 的檔案系統，也就是說一個 network redirector）。兩者之間的差異在於 local 檔案系統使用 IOS services，在一個 local resource 上執行 I/O 動作。而 network redirector 將 requests 透過網路傳送出去。兩種檔案系統對大部份來自 IFS Manager 的 calls 都有反應。此外，回答 remote requests（和檔案動作有關）的伺服器本身亦內含一個 local 檔案系統，以便能夠將 requests 完成。LPFS 是一個 local 檔案系統，所以我在 *CLocalFileSystem* 之下又衍生 *CLpfs*，如圖所示。

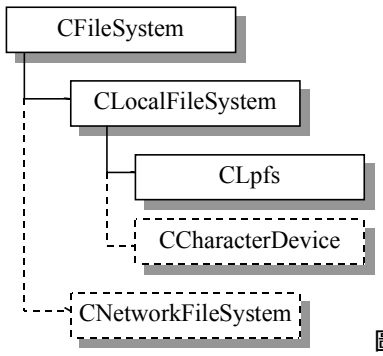


圖 16-6 LPFS 檔案系統實例中的 classes 體系

IFS 相關技術

就像其他的 local FSDs 一樣，LPFS 是一個 static VxD，初始化次序是 *FSD_Init_Order*。它使用 IFS 和 IOS 的 services。我們在本書中所討論的大部份 VxDs 都是 dynamic，它們在處理 *Sys_Dynamic_Device_Init* 和 *Sys_Dynamic_Device_Exit* 兩訊息時分別執行初始化程序和關閉程序。此外，它們並不在乎初始化次序。然而，做爲一個 static VxD，LPFS 需要處理 *Device_Init* 訊息並需要擁有一個特定的初始化次序，以使它自己正確地符合 Windows 95 的組織體系。我使用 *FSD_Init_Order* + 100h 做爲初始化次序，如此一來我的 VxD 就會在 VFAT 以及其他 file system drivers 之後才初始化。稍後我將在「註冊次序的重要性」這一段提示文字中解釋，爲什麼初始化次序對此例而言是如此重要。

LPFS.CPP 是這個 VxD 的主程式，內含 *CLpfs* class 的實作碼。關於 VxD 的初始化部份，顯示於下：

```
#define NULL 0

extern "C" {
#define WANTVXDWRAPS
#include <basedef.h>
#include <vmm.h>
#include <debug.h>
#include <vxdwraps.h>
#include <winerror.h>
#include "iosdcls.h"
#include "ifsmgr.h"
    } // extern "C"

#pragma hdrstop

#include "lpfs.h"

void OnAsyncEvent(PAEP aep);
CFileSystem* CreateInstance();

#pragma VxD_INIT_CODE_SEG
#pragma VxD_INIT_DATA_SEG
```

```
extern "C" extern DRP theDRP;

SYSCTL BOOL OnDeviceInit(HVM hVM, DWORD refdata)
   {                          // OnDeviceInit
   if (!IOS_Get_Version())
      return FALSE;           // IOS not loaded (?)
   theDRP.DRP_aer = (PVOID) OnAsyncEvent;
   IOS_Register(&theDRP);

   return CLpfs::Register(CLpfs::CreateNew);
   }                          // OnDeviceInit
```

你一定不會希望自己定義 NULL，但是 BASEDEF.H 把它定義爲（不正確地）'\0'。C++ 編譯器並無意願去接受這樣一個字元，做爲 NULL 指標的對等物。幸運的是，BASEDEF.H 的定義前後圍繞有一個 *#ifndef*，使它有可能提供一個可運作的定義。以 *extern "C" {...}* 涵蓋 DDK 表頭，可避免因爲 VxD service 的 C++ 外包函式（wrappers）而帶來的名稱裝飾的困窘（譯註，指的是 C++ 的 name mangling）。IFSMGR.H 是我寫的表頭檔，代替一個我原本希望在 DDK 中找到但結果卻沒有的檔案。其中含入一個 DDK 表頭檔（IFS.H）的輕微修改版本。我的修改使得「同時含入 IFS.H 和正規的 Win32 表頭檔」成爲可能。IOSDCLS.H 是我們在第 15 章討論過的同一檔案，用於存取 Input/Output Supervisor。

Device_Init 這個 system control message 的處理常式註冊 IOS 以及 IFS Manager（間接地）。像其他的 file system drivers 一樣，LPFS 需要呼叫那些「只可以經由 IOS linkage block（ILB）才聯絡上」的函式。爲了獲得一個 ILB，你必須呼叫 *IOS_Register*。如果你想註冊 *IOS_Register*，你必須供應一個 driver registration packet 和一個非同步（asynchronous）的 event 函式。我在 DEVDCL.ASM 檔案中定義 DRP 如下：

```
VxD_IDATA_SEG
        public _theDRP
_theDRP  DRP   <EyeCatcher, DRP_TSD, 0, offset32 _theILB, \
             'LowPerfFileSys  ', 0, 0, 0>
VxD_IDATA_ENDS
```

其中的非同步（asynchronous）event 函式可以是空的，像這樣：

```
#pragma VxD_LOCKED_CODE_SEG
#pragma VxD_LOCKED_DATA_SEG

void OnAsyncEvent(PAEP aep)
   {                        // OnAsyncEvent
   aep->AEP_result = AEP_SUCCESS;
   }                        // OnAsyncEvent
```

除非你將 IOS 註冊爲一個所謂的 "noncompliant" layer driver，否則你需要一個「有作爲」的 event 函式。由於我們的 driver 是一個 FSD，你可能已經猜到，我們已經藉由定義了一個 noncompliant driver（它不需要一個非同步 event 函式）而註冊爲一個 FSD。奇怪的是，所有 Microsoft's FSDs 都告訴 IOS 說它們是 type-specific drivers（TSDs），後者的確需要 event 函式。

IFS registration call 被埋藏在 *Register* 成員函式中，稍後我會在「註冊 LPFS」那一節描述之。呼叫 *Register* 函式時，必須有一個 *CLpfs* static 成員函式位址做爲參數，用來產生一個新的 *CLpfs* 實體：

```
CFileSystem* CLpfs::CreateNew()
   {                        // CreateInstance
   return new CLpfs();
   }                        // CreateInstance
```

稍後我將在（數頁之後）一個名爲「產生一個 LPFS 實體」的提示文字中告訴各位，這個函式如何符合大體架構。

CFileSystem，所有關於檔案系統的 classes 的基礎，有著以下介面：

```
class CFileSystem
{                           // CFileSystem
public:
   CFileSystem();
   ~CFileSystem();

   static CFileSystem* (*CreateNewFcn)();
   static int ProviderId;
   static BOOL Register(CFileSystem* (*createnew)());
};                          // CFileSystem
```

Register 函式簡單地將其函式指標參數儲存在 *CreateNewFcn* 變數之中：

```
BOOL CFileSystem::Register(CFileSystem* (*createnew)())
   {                        // CFileSystem::Register
   CreateNewFcn = createnew;
   return TRUE;
   }                        // CFileSystem::Register
```

關於 LPFS 的更多資訊 完整的 LPFS 原始碼放在書附碟片中。我並不打算把它們統統列出於此，因爲我要顯示給你看的，其實只是如何組合 IFS 和 IOS services 來完成一個檔案系統的基礎工作。然而，如果你看過了原始碼，你會發現我有時候是以與此處不同或比此處更一般化的方式來實作函式。我並不打算設計一個用於檔案系統專案開發的 C++ class library，我只是想對實作過程有足夠的解釋，讓你瞭解出現在本章文字中的一些程式片段。

Volume Mounting（Volumn 的裝載）

一個 local FSD 藉由呼叫 *IFSMgr_RegisterMount* 來註冊 IFS Manager：

```
int code = IFSMgr_RegisterMount(mountfunc, version, type);
```

其中 *mountfunc* 表示你的 volume mount 函式，*version* 指出你能夠與之一起工作的 IFS Manager 的最低版本。*type* 參數表示你的 FSD 是 default FSD 或否。你總是應該指定最後兩個參數爲 *IFSMGRVERSION* 和 *NORMAL_FSD*。Microsoft 的 VDEF.VXD（唯一在 DDK 中呈現有原始碼的一個 FSD）則使用 *DEFAULT_FSD* 做爲最後一個參數，它是唯一這麼做的一個 FSD（稍後我將解釋爲何要將 VDEF.VXD 設爲 default FSD 的理由）。傳回值是一個 provider ID，用來識別檔案系統（在 Windows 95 之中，一個 local file system provider 的 provider ID 其實是 IFS Manager 內部陣列的一個索引值。這樣的關係將在下一版 Windows 中改變，所以你不應該依賴這個事實，除非這可以幫助你對一個 FSD 除錯）。

當有人識別出一個新 disk drive 已經出現，或是當一個原已存在的 drive 中的儲存媒體（如碟片）被抽換時，Input / Output supervisor 就會發出 volume mounting requests。然後 IFS Manager 會 "polls" 所有「已經註冊一個 mount function」之 FSDs，以便將一個 FSD 連接於此 volume 身上。"polling" 動作的發生次序與 FSDs 註冊次序相反，只有一個例外，那就是：唯一的那個 default FSD 最後才被 "polled" -- 不管它何時被註冊（林偉博註：此即確保 volume mounting requests 至少會被 Windows 內建之 FSD（VDEF.VXD）處理到）。爲了 "poll" 一個檔案系統，IFS Manager 呼叫被註冊的 mount function，其原型如下：

```
int FS_MountVolume(pioreq pir);
```

參數 *pir* 指向一個 IFS I/O request，其中的 *ir_flags* 指出 *IR_FSD_MOUNT* 子機能，*ir_volh* 指向 logical volume 的 IOS VRP 結構，*ir_rh* 指向 MS-DOS DPB 串鏈的頭，*ir_mntdrv* 指出 volume 的磁碟機代碼。你的 mount function 應該決定是否要爲此 volume 承認這個檔案系統。如果不，它應該簡單地爲 *ir_error* 設定一個錯誤代碼，並以此做爲其傳回值。如果它要承認這個檔案系統，就應該爲你的 FSD 建設所需機構，以管理這個 volume，並設定數個輸出值，然後傳回 0。你應該將輸出值設定如下：

- 在 *ioreq* 之中，設定 *ir_rh* 欄位，使它指向你的內部的 volume-oriented 資料結構。IFS Manager 不會檢查這個結構，但它會以 *ir_rh* 欄位將此指標還遞給你 – 在所有與此 volume 相關的後繼函式呼叫中。
- 在 *ioreq* 之中，設定 *ir_vfunc* 爲一個 *volfunc* 結構位址，該結構內含指標，指向 15 個函式，IFS Manager 會呼叫這 15 個函式來處理 volume-oriented requests（如 *FS_OpenFile*）。
- 在 VRP 之中，設定 *VRP_fsd_entry* 指向你的 *FS_MountVolume* 函式。各式各樣的 IOS components 偶而會直接呼叫你的函式。例如，當實際裝置的 port driver 確定裝置上的媒體已被更換，VOLTRACK 便會呼叫你的函式以執行一個 *IR_FSD_VERIFY* 函式。
- 在 VRP 之中，設定 *VRP_fsd_hvol* 指向你的內部的一個 volume-oriented 資料結構。當 IOS components 爲你的 *FS_MountVolume* 函式建構 I/O requests，它們會使用此一欄位來初始化 *ir_rh*。

註冊次序的重要性 我很辛苦地才學到了下面的心得：如果你修改 *ioreq* 結構並決定拒絕新的 volume，你的修改將被保存於 mount calls 之內，那是 IFS Manager 所發出，給予「低優先權檔案系統」(也就是在你的這個系統之前以 IFS Manager 註冊的檔案系統)。如果你修改了 *ir_rh*，較低優先權的檔案系統或許會崩潰掉 -- 如果它們將此數值解釋爲一個 DPB 指標的話。這個教訓就是：除非必要，不要改變 *ioreq* 或 VRP。但 VFAT 無視於這項警告而修改了 *ir_rh* 欄位 -- 即使當它決定拒絕新的 volume 時。CDFS 和 VDEF（通常也就是唯一出現的兩個檔案系統），並不在乎 *ir_rh* 是否被修改過，但如果你的檔案系統的優先權低於 VFAT，而你使用 *ir_rh* 做爲輸入用欄位，你就有麻煩了。我的忠告是，使用一個大於 *FSD_Init_Order* 的初始化次序（initialization order），於是你的檔案系統的載入和註冊就會比 VFAT 晚，使你能夠在 VFAT 修改 *ir_rh* 之前有機會先看到 mount requests。

將 LPFS 註冊

一如稍早所示，LPFS 的 *Device_Init* 處理常式呼叫 *Register* 函式時，會依賴 IFS Manager 來註冊檔案系統。該函式內含程式碼如下：

```
BOOL CLocalFileSystem::Register(CFileSystem* (*createnew)())
   {                          // CLocalFileSystem::Register
   if (!CFileSystem::Register(createnew))
      return FALSE;
   if (IFSMgr_Get_Version() < IFSMGRVERSION)
      return FALSE;           // back-level system
   if ((ProviderId = IFSMgr_RegisterMount(MountVolumeThunk,
      IFSMGRVERSION, NORMAL_FSD)) < 0)
      return FALSE;
   return TRUE;
   }                          // CLocalFileSystem::Register
```

呼叫 *CFileSystem::Register* 可以記錄 *CLpfs::CreateNew* 函式位址，以便稍後使用。呼叫 *IFSMgr_Get_Version* 有兩個目的。如果 IFS Manager 尚未被載入（我並不清楚這怎麼會發生），我們就不載入我們的 driver。此外，如果發現我們自己執行於一個 back-level 系

統（一個比 driver 所針對的系統更老舊的系統）我們也不打算被載入。*IFSMGRVERSION* 是指 IFS Manager 的版本，此 IFS Manager 之 interfaces 爲我們的 driver 所用，其功能也爲我們所瞭解。將 *IFSMgr_Get_Version* 的傳回值與此常數比較，我們就可以知道是否有我們可以與之合作的 IFS Manager 存在。如果比較結果不相同，我們就傳回 FALSE，而 *Device_Init* 處理常式於是將傳回一個失敗代碼給 VMM，令 VMM 將 LPFS 卸載。IFS Manager 檢查由我們傳給 registration API 的版本號碼常數，決定它可否與我們一起運作。

裝載（Mounting）LPFS Volumes

CLocalFileSystem 有一個 static 成員函式，名爲 *MountVolumeThunk*，用以處理來自 IFS Manager 和 IOS layer drivers（如 VOLTRACK）的裝載請求（mount requests）。mount 協定是以一個 *IR_FSD_MOUNT* request 開始，如下所示：

```
int CLocalFileSystem::MountVolumeThunk(pioreq pir)
   {                  // CLocalFileSystem::MountVolumeThunk
   pir->ir_error = ERROR_ACCESS_DENIED; // assume error

   switch (pir->ir_flags)
      {                    // select volume mount function

   case IR_FSD_MOUNT:
      {                    // IR_FSD_MOUNT
      CLocalFileSystem* lfs = (CLocalFileSystem*)
         (*CreateNewFcn)();
      lfs->m_vrp = (PVRP) pir->ir_volh;
      lfs->m_drive = lfs->m_origdrive =
         (BYTE) pir->ir_mntdrv;
      lfs->m_dpb = pir->ir_rh;
      if ((pir->ir_error = lfs->MountVolume(pir)) == 0)
         {                 // successfully mounted
         pir->ir_rh = (rh_t) lfs;
         lfs->m_vrp->VRP_fsd_hvol = (ULONG) lfs;
         lfs->m_vrp->VRP_fsd_entry = (ULONG)
            MountVolumeThunk;
         }                 // successfully mounted
      else
         delete lfs;        // error trying to mount volume
      break;
```

```
        }                   // IR_FSD_MOUNT
        ...
        }               // select volume mount function

    return pir->ir_error;
    }                   // CLocalFileSystem::MountVolumeThunk
```

我們一開始就先假設此函式失敗。*ERROR_ACCESS_DENIED* 是一個合宜的錯誤代碼，表示「我不承認（認識）這個 volume」。當 mount 函式回返時，某些錯誤代碼會被 IFS Manager 以特殊個案處理，不過上述代碼並非特殊情況之一。不管什麼值留在 *ir_error* 中，最後就成爲此函式的回返值。

產生一個 LPFS 實體 mount 函式必須是一個 static 函式，因爲它的目的是要決定「產生一個 file system object 是否合理」。爲了下此決定，它首先產生一個 file system object（稍後可能會被丟棄），主要是爲了使調查 volume 的工作輕鬆些。這就是 *CreateNew* 函式上場表演的地方了：我們必須爲 *CLocalFileSystem* 之衍生類別產生出一個實體（instance），但是當我們編譯 LOCALFILESYSTEM.CPP 時我們並不知道什麼 class 才是我們的目標。C++ 並不真正能夠爲「執行時期才獲知的 classes」產生物件（譯註：此爲所謂的 dynamic creation），（這正是 MFC 的 *CObject* 和 *CRuntimeClass* 之所由來）。*CreateNewFcn* 是一個指標，指向一個 static 函式，此函式隸屬於討論中的衍生類別所有，用以產生該類別的一個實體（instance）。

一旦有了新的 file system object 在手，mount 函式就可以從 *ioreq* 中將重要參數拷貝到 object 之中。這些參數包括一個 VRP 位址（*ir_volh*），一個 drive 編碼索引（*ir_mntdrv*），以及一個 MS-DOS DPB 位址（*ir_rh*）。然後我們呼叫 *MountVolume*，那是一個虛擬成員函式，必要時候可被改寫（overridden）。此例之中我並不需要改寫 *MountVolume*，所以它只是 *CLocalFileSystem* 的一部份：

```
int CLocalFileSystem::MountVolume(pioreq pir)
    {                       // CLocalFileSystem::MountVolume
    if (!OurVolume(pir))
        return ERROR_ACCESS_DENIED;
```

```
    for (CLocalFileSystem* fs = First; fs; fs = fs->m_next)
        if (fs != this && SameVolume(fs))
            return ERROR_IFSVOL_EXISTS;
    pir->ir_vfunc = &volfuncs;
    return 0;
    }                       // CLocalFileSystem::MountVolume
```

此例之中，*volfuncs* 是一個 static 變數，型別爲 *volfunc*，用來指定一個由虛擬函式指標組成的表格，各虛擬函式用以實作 volume-oriented 函式，如 *FS_OpenFile*。

OurVolume 是一個你應該改寫的虛擬函式，它用來決定目前的 volume 是否內含一個「我們嘗試實作的 file system」的實體（instance）。當你建造一個檔案系統，你應該改寫 *OurVolume* 虛擬函式以檢查 disk volume 的內容，用以決定是否能辨識該 volume。在此極簡化的例子中，我只檢查 master boot record 中的 vendor ID，看看我是不是正在處理一個我們曾於第 15 章見過的 RAM disks：

```
BOOL CLpfs::OurVolume(pioreq pir)
    {                       // CLpfs::OurVolume
    PBYTE bootsec = ReadBootSector();
    if (!bootsec)
        {                   // can't read boot sector
        pir->ir_error = m_error;
        return FALSE;
        }       (           // can't read boot sector
    BOOL ours = (memcmp(bootsec + 3, "WALTONEY", 8) == 0);
    _HeapFree(bootsec, 0);
    return ours;
    }                       // CLpfs::OurVolume
```

你還需要一個類似 *SameVolume* 的函式，它被用來檢查是否有重複的 mount request。稍後我再敘述 *SameVolume*，以便配合 *IR_FSD_VERIFY request*，因爲後者也需要這個 request。

讀取 Boot Sector

ReadBootSector 是 *CLocalFileSystem* 的一部份。它使用一些輔助函式來產生並執行一個

read 動作，從 logical volume 中取出 sector 0。此動作比之我們在第 5 章所討論的，只多一個新特質，所以我願意秀出一些程式。*ReadBootSector* 看起來如下：

```
PBYTE CLocalFileSystem::ReadBootSector()
   {                    // CLocalFileSystem::ReadBootSector
   PBYTE buffer = (PBYTE) _HeapAllocate(m_vrp->VRP_block_size,0);
   if (!buffer)
      return NULL;
   if (m_error = ReadSectorNow(0, buffer))
      {                    // request had error
      _HeapFree(buffer, 0);
      return NULL;
      }                    // request had error
   return buffer;
   }                    // CLocalFileSystem::ReadBootSector
```

這個函式配置一塊記憶體，足夠大到持有一個 disk sector。它從 IOS VRP 中取出 sector 的長度。這個函式然後呼叫一個名爲 *ReadSectorNow* 的輔助函式，將 sector 0 讀進被配置的記憶體中。*ReadSectoNow* 看起來像這樣：

```
int CLocalFileSystem::ReadSectorNow(ULONG sector, PBYTE buffer)
   {                     // CLocalFileSystem::ReadSectorNow
   PIOR ior = CreateIOR(IOR_READ,
      IORF_BYPASS_VOLTRK | IORF_HIGH_PRIORITY);
   ior->IOR_start_addr[0] = sector;
   ior->IOR_buffer_ptr = (ULONG) buffer;
   ior->IOR_xfer_count = 1;
   SatisfyCriteria(ior);
   SendCommandAndWait(ior);
   int status = (int) ior->IOR_status;
   DestroyIOR(ior);

   if (status >= IORS_ERROR_DESIGNTR)
      return IOSMapIORSToI21(status);
   else
      return 0;
   }                     // CLocalFileSystem::ReadSectorNow
```

ReadSectorNow 使用 *CreateIOR* 來產生一個新的 I/O request descriptor，*SatisfyCriteria* 呼叫 IOS criteria 函式以準備一個 request，此 request 是爲了真正傳送到 IOS layer drivers

中的 top level。*SendCommandAndWait*（稍後我將詳細討論）執行 I/O request 並等待它完成。*DestroyIOR* 將 IOR 傳回給 IOS。如果你遺漏了 *DestroyIOR* 步驟，一旦使用者最終退出系統，Windows 95 會「掛」掉，因爲 IOS 會以爲仍有一個未解決的 request，而它應該等待下去。最後，上述函式使用 *IOSMapIORSToI21* 將任何來自 IOS set 的傳回值（錯誤代碼）映射至 IFS set 中。

錯誤檢驗的注意事項　*ReadBootSector* 和 *ReadSectorNow* 函式說明了一種偵測並記錄磁碟錯誤的基本方法。在緊隨而來的下列各例子中，我其實並不處理這些特殊的傳回值。在一個產品化的檔案系統中，你應該測試各種可能的錯誤。然而，「錯誤檢查」手續繁複，使程式難以閱讀，所以我把程式碼清乾淨一些才列出來。（我並不打算教你如何寫程式，而是要教你如何和 IFS 一起運作）。然而測試不尋常狀況卻又不可與此相提並論，我不會遺漏這類測試。

SendCommandAndWait 示範 file system drivers 一般所執行的同步化（synchronous）I/O 動作：

```
#define IOR_event _ureq._IOR_requestor_usage[0]

void CLocalFileSystem::SendCommandAndWait(PIOR ior)
    {                // CLocalFileSystem::SendCommandAndWait
    ior->IOR_event = 0;
    ior->IOR_callback = (CMDCPLT) OnCommandComplete;
    ior->IOR_req_req_handle = (ULONG) ior;
    SendCommand(ior);

    while (TRUE)
        {                // wait for command to finish
        _asm pushfd
        _asm cli
        if (ior->IOR_event)
            break;         // it's done
        IFSMgr_Block((ULONG) &ior->IOR_event);
        _asm popfd
        }                // wait for command to finish
    _asm popfd
    }                // CLocalFileSystem::SendCommandAndWait
```

```
void CLocalFileSystem::OnCommandComplete(PIOR ior)
    {                   // CLocalFileSystem::OnCommandComplete
    ior->IOR_event = 1;
    IFSMgr_Wakeup((ULONG) &ior->IOR_event);
    }                   // CLocalFileSystem::OnCommandComplete
```

一如我在 15 章所說，當 IOS clients 有一個同步（synchronous）命令等待被執行時，大部份的它們避免使用 *IORF_SYNC_COMMAND* flag。取而代之的是，它們有一些其他方法來等待 requests 完成。顯示於以上程式片段中的方法，與 VFAT 所採用的方法很類似。

IOS I/O request descriptor（IOR）內含一個名爲 *_ureq* 的欄位，被用做 IOCTL 參數，或是 IOS client 所想像的任何目的。在這裡，我們使用 *_ureq* union 中的一個 DWORDs，做爲一個 "completion flag"。在送出 I/O request 給 IOS 之前，我們先設此 flag 爲 0。我們的 I/O completion 函式（*OnCommandComplete*）將在此動作最終完成之時，把這個 flag 設爲 1。爲了等待這個動作的完成，我們呼叫 *IFSMgr_Block*，並將 flag word 的位址交遞過去。*IFSMr_Block* 是一個薄薄包裝，其實是呼叫 *_BlockOnID* service 並夾帶 *BLOCK_SVC_INTS* 和 *BLOCK_SVC_ENABLE_INTS* flags。如你所知，*_BlockOnID* 會阻塞（blocks）目前的執行緒（current thread），直到有人呼叫 *_SignalID* 並夾帶相同的 32 位元 blocking ID 做爲其參數爲止。*OnCommandComplete* 函式中所呼叫的 *IFSMgr_Wakeup*，正是用來執行上述那個函式。因此，對 *IFSMgr_Block* 的呼叫動作會在我們等待的那個函式完成之後，才回返。

想必你也知道，*_BlockOnID* 和 *_SignalID* 將其參數視爲任意的 32 位元值。有可能（雖然未必成真）兩個不同的 VxDs 意外地選擇相同的 blocking ID。因此，只要你使用這兩個函式來對某一個動作做同步化控制，你就必須有一個附帶的方法來驗證，被喚醒的呼叫動作是你所期望的那一個。*SendCommandAndWait* 函式中的 while 迴圈就是用來完成這個驗證工作，它檢查 flag word 是否已不再是 0。如果仍是 0，表示某些人可能在呼叫 *_SignalID* 時意外地用到了這個 flag word 位址（而其本意並非如此），所以你必須重複此迴圈（你並不一定要使用這個附帶的 flag 的位址做爲 blocking ID，不過你通常會這麼做）。

此段碼也展示了 *_BlockOnID* 和 *_SignalID* 之間的細微差別。如果訊號已經發生，很重要的一件事是千萬不要去 "block"，因爲這麼做的話你將絕不會醒過來（*_SignalID* 很像 Win32 *PulseEvent* API 函式，後者只釋放一個「等待中的 thread」）。因此，你必須在 "blocking" 之前先檢查 event flag。此外，爲了避免在你的檢查動作和「asynchronous completion 函式」之間有一個 race condition 發生，你必須確定你是「不可被中斷」的。「令中斷失效」只是保險的一部份，你還必須小心，不要引起一個 page fault，因此，「等待迴圈」這一段碼必須置於 locked code segment 之中。

Volumes 上的各種動作

在 IFS Manager 對你的 *MountVolume* 函式完成了一個成功的 *IR_FSD_MOUNT* call 之後，你的 volume 對系統中的其他 components 而言，便成爲「可利用的」。現在你可以收到其他的 *MountVolume* 函式呼叫並夾帶 *IR_FSD_MOUNT* 以外的子功能碼了，你也可以收到「被你指定於 *ir_vfunc* 表格中的函式（那是你傳回給 IFS Manager 的）」的函式呼叫了。

其他的 *MountVolume* 子功能

IFS 和 IOS layer drivers 可以呼叫你的 *MountVolume* 函式來處理下面這些額外的子機能：

IR_FSD_MAP_DRIVE　IFS Manager 會發出一個 *IR_FSD_MAP_DRIVE* request 以支援某些磁碟壓縮方案（例如 Microsoft 的 DRVSPACE），因爲那需要轉換磁碟機的代碼。也就是說，compression driver 可能最初先依據 drive C 上的一個檔案產生出一個 compressed drive Q，但它又必須讓這個 compressed drive 的代碼爲 C，而其 base drive 代碼爲 Q。爲了支援這項性質，FSD 必須追蹤兩個磁碟機代碼：一個是目前的磁碟機代碼，一個是原始的磁碟機代碼。當呼叫 I/O requests 下至 IOS 時，FSD 使用目前的磁碟機代碼；而使用保留下來的原始磁碟機代碼，以處理一個後繼而來的 *IR_FSD_UNMAP_DRIVE* request。例如：

```
class CLocalFileSystem : public CFileSystem
{
...
BYTE m_drive;     // current drive number (A = 0)
BYTE m_origdrive; // original drive number (A = 0)
...
};

int CLocalFileSystem::MapDrive(pioreq pir)
    {                        // CLocalFileSystem::MapDrive
    m_drive = (BYTE) pir->ir_options;
    m_origdrive = (BYTE) pir->ir_pos; // undocumented!
    return 0;
    }                        // CLocalFileSystem::MapDrive
```

順帶一提，*ir_pos* 內含原始磁碟機代碼的一個新值，此事其實並未公開。兩個磁碟機代碼的初值正是原本 *IR_FSD_MOUNT* request 的 *ir_mntdrv* 參數。

IR_FSD_MOUNT_CHILD IFS Manager 使用 *IR_FSD_MOUNT_CHILD* request 來 "mount" 一個「實際上是另一個 volume 的 child volume」的 volume。例如，一個 compressed drive 就是「檔案所居留的那個 volume」的 child。你或許能夠對 parent volume 和 child volume 使用相同的碼。VFAT 對這兩個 request（*IR_FSD_MOUNT* 和 *IR_FSD_MOUNT_CHILD*）有輕微不同的函式，因爲它對 parent volumes 做了兩件事情（分別是針對可抽換之儲存媒體的 boot sector smashing 動作，以及 DPB 查閱動作），與 child volume 無關。

由於某些我不瞭解的原因，parent 和 child volumes 必須使用相同的檔案系統。因此，在這個限制之下，CDFS（譯註：CD ROM File System）不可能提供一個內含「由 VFAT 管理之資料」的 compressed logical volume。

IR_FSD_UNLOAD IOS layer driver 可以送給 FSD 一個 *IR_FSD_UNLOAD* request，強迫 FSD "dismount" 一個 volume。你應該掃清（flush）任何 cache，並清除（clean up）任何你所握有且代表 volume 的其他資源。然而你還是可以收到此一 volume 的其他 requests，所以你不應該釋放「內含你的內部資料結構」的那些記憶體。請一直等待到接收一個 *FS_DisconnectResource* call，才釋放記憶體。

IR_FSD_UNMAP_DRIVE　一個 IOS layer driver 如果負責壓縮處理，就應該送給 FSD 一個 *IR_FSD_UNMAP_DRIVE* request，以消除稍早一個 *IR_FSD_MAP_DRIVE* request 的影響。一項未公開但完全被預期的「進入狀態」是：*ir_rh* 指向 FSD 之中一個針對此 volume 的內部控制區塊。FSD 應該恢復使用原始的磁碟機代碼，例如：

```
int CLocalFileSystem::UnmapDrive(pioreq pir)
   {                        // CLocalFileSystem::UnmapDrive
   m_drive = m_origdrive;
   return 0;
   }                        // CLocalFileSystem::UnmapDrive
```

IR_FSD_VERIFY　VOLTRACK（一個 volume tracking layer driver，處理「儲存媒體可被搬移之 device」）會直接呼叫你的 *MountVolume* 函式並夾帶子機能碼 -- 只要 port driver 表示儲存媒體（碟片）被抽換了。你的 FSD 的任務就是看看「現在被裝載於實際裝置上的 volume」是否與「儲存媒體（碟片）尚未被抽換前所裝載的 volume」相同。畢竟來自 port driver 的訊號有可能被僞造，要不然使用者也可能會退出（ejected）並立刻重新安插一個 volume。

> 林偉博註：port driver 訊息被僞造的最常見例子是由於使用者快速抽換磁片而造成。例如插入 750KB 之 3-1/2" 碟片後，又馬上抽換爲 1.44MB 之 3-1/2" 碟片。

這個子機能的呼叫序列（calling sequence）並未公開，但是對我而言，將 VFAT.VXD 和 VOLTRACK.VXD 反組譯之後，已經可以輕鬆理解。輸入時，*ir_drvh* 指向你的 volume handle，此值被用來 "mounted" 至 drive 身上；*ir_data* 指向一個 256-byte 區域，你可以在其中放置新 volume 的標籤（label）。如果 drive 仍然內含舊的 volume，你應該傳回 0，並設定 *ir_error* 爲 0。如果 drive 內含一個新的 volume，你應該傳回 -1，並設定 *ir_error* 爲 -1，並填寫 *ir_data* 區域，並設定 *ir_aux2* 爲 32-bit volume serial number。一旦此事發生，VOLTRACK 不會對輸出資料做任何動作，不過仍可能在未來某一天被改變。下面是 LPFS 處理 *IR_FSD_VERIFY* 子機能的方式：

```
int CLocalFileSystem::MountVolumeThunk(pioreq pir)
   {                  // CLocalFileSystem::MountVolumeThunk
   pir->ir_error = ERROR_ACCESS_DENIED; // assume error

   switch (pir->ir_flags)
      {                   // select volume mount function
      ...
   case IR_FSD_VERIFY:
      {                   // IR_FSD_VERIFY
      CLocalFileSystem* lfs = (CLocalFileSystem*) CreateNew();
      CLocalFileSystem* old = (CLocalFileSystem*) pir->ir_drvh;
      lfs->m_drive = lfs->m_origdrive = old->m_drive;
      lfs->m_vrp = old->m_vrp;
      pir->ir_error = lfs->VerifyVolume(pir);
      delete lfs;           // used only for verification
      break;
      }                   // IR_FSD_VERIFY
      ...
      }                   // select volume mount function
   return pir->ir_error;
   }                  // CLocalFileSystem::MountVolumeThunk

int CLocalFileSystem::VerifyVolume(pioreq pir)
   {                  // CLocalFileSystem::VerifyVolume
   if (SameVolume((CLocalFileSystem*) pir->ir_drvh))
      return 0;
   GetVolumeLabel((PDWORD) &pir->ir_aux2.aux_ul,
      (char*) pir->ir_data);
   return -1;
   }                  // CLocalFileSystem::VerifyVolume

BOOL CLpfs::SameVolume(CLocalFileSystem* fs)
   {                  // CLpfs::SameVolume
   return TRUE;
   }                  // CLpfs::SameVolume

void CLpfs::GetVolumeLabel(PDWORD pVolSer, char* pVolLabel)
   {                        // CLpfs::GetVolumeLabel
   PBYTE bootsec = ReadBootSector();
   if (!bootsec)
      return;
   *pVolSer = *(PDWORD) (bootsec + 0x39);
   memcpy(pVolLabel, bootsec + 0x43, 11);
   pVolLabel[11] = 0;
   _HeapFree(bootsec, 0);
   }                        // CLpfs::GetVolumeLabel
```

在 *MountVolumeThunk* 函式中，我產生一個額外的 file system object，以便能夠比較容易地面對 volume 執行 I/O（我總是在回返之前拋棄這個額外的 object）。然後我呼叫一個名爲 *VerifyVolume* 的成員函式，執行真正的 request。*VerifyVolume* 內部呼叫一個名爲 *SameVolume* 的虛擬函式，比較新舊 volume 之間任何重要的資訊。VFAT 做了一些檢查，包括 vendor ID, serial number, label, 以及磁碟格式等等，都必須吻合才行（VFAT 會把一個「可抽換之 volume」的 vendor ID 重新寫入，如此一來它就可以分辨大量生產的 volumes。我曾在第 15 章的一段注意方塊中討論這個多少有點問題的動作）。最後兩個函式，*CLpfs::SameVolume* 和 *CLpfs::GetVolumeLabel* 基本上只是模型，我寫它們只是爲了萬一它們真的被呼叫的話，LPFS 還是可以正常運作。順帶一提，它們並不會被呼叫，因爲我用來測試 LPFS 的 RAM disk 並沒有可抽換的媒體（碟片）。

Volume-Oriented I/O Requests

爲了實作一個 I/O request 並能夠配合你的 volume，IFS Manager 呼叫你所提供的函式（其位址放在 *ir_vfunc* 表格中 -- 當你的 driver 從 *IR_FSD_MOUNT* 中回返時）。DDK 技術文件係以名稱來描述這些函式，例如 *FS_DeleteFile* 等等，所以我也遵循相同的習慣。下面就是這些函式：

FS_DeleteFile　*FS_DeleteFile* 刪除那些出現在 *ir_ppath* 中的檔案。由於 LPFS 不允許你殺掉其唯一一個檔案，所以我沒有例子可以爲你示範這個函式。這個函式實作起來並不簡單，因爲你可能必須因爲 wildcard 的緣故，搜尋最低階的 directory，以便找到要被刪除的檔案。稍後討論到 *FS_FindFirstFile* 函式時，我會顯示一個例子，告訴你如何在一個 FSD 中完成所謂的「wildcard 搜尋」。

FS_Dir　*FS_Dir* 在一個 directory 上執行，例如產生或刪除一個 directory，檢查某個 directory 的存在，或將一個 directory 名稱在長名稱和 8.3 型式之間轉換。

FS_DirectDiskIO　IFS Manager 呼叫 *FS_DirectDiskIO* 以處理 MS-DOS INT 25h（絕對磁碟讀取）和 INT 26h（絕對磁碟寫入）這兩個 requests。你的 FSD 應該以三個步驟來實作這個函式：

1. 將此 request 所關係的 memory pages 鎖住。如果無法鎖住此 pages，就設立 *IORF_DOUBLE_BUFFER* flag，如此一來 IOS 只在它所擁有的被鎖定緩衝區（locked buffer）中執行 I/O。
2. 建立並執行一個 IOS request，以讀寫 sectors。
3. 解除你稍早鎖定的 pages。

舉個例子：

```
int CLocalFileSystem::DirectDiskIO(pioreq pir)
   {                    // CLocalFileSystem::DirectDiskIO
   DWORD opcode;
   DWORD pageflags = PAGEMAPGLOBAL;
   DWORD iorflags = IORF_BYPASS_VOLTRK | IORF_HIGH_PRIORITY
      | IORF_SYNC_COMMAND;

   switch (pir->ir_flags)
      {                 // switch on function code
   case DIO_ABS_READ_SECTORS:
      opcode = IOR_READ;
      pageflags |= PAGEMARKDIRTY;
      break;
   case DIO_ABS_WRITE_SECTORS:
      opcode = IOR_WRITE;
      break;
   case DIO_SET_LOCK_CACHE_STATE:
      return pir->ir_error = 0;
      }                 // switch on function code

   DWORD nbytes = pir->ir_length * m_vrp->VRP_block_size;
   if (!nbytes)
      return pir->ir_error = 0;
   DWORD buffer = (DWORD) pir->ir_data;
   DWORD firstpage = buffer >> 12;
   DWORD npages = ((buffer + nbytes - 1) >> 12) - firstpage
      + 1;
   DWORD locked = _LinPageLock(firstpage, npages, pageflags);
   if (locked)
      pir->ir_data = (ubuffer_t) ((locked & ~4095)
         | (buffer & 4095));
   else
      iorflags |= IORF_DOUBLE_BUFFER;
```

```
PIOR ior = CreateIOR(opcode, iorflags);
ior->IOR_xfer_count = pir->ir_length;
ior->IOR_start_addr[0] = pir->ir_pos;
ior->IOR_buffer_ptr = (DWORD) pir->ir_data;
SatisfyCriteria(ior);
SendCommandAndWait(ior);
pir->ir_error = (short)
    ((ior->IOR_status >= IORS_ERROR_DESIGNTR)
    ? IOSMapIORSToI24(ior->IOR_status, opcode) : 0);
DestroyIOR(ior);

if (locked)
    _LinPageUnlock(locked >> 12, npages, PAGEMAPGLOBAL);

return pir->ir_error;
}                       // CLocalFileSystem::DirectDiskIO
```

其中 *ir_flags* 參數表示我們要執行哪一個 direct disk 函式。在這裡以及後數頁之中，我省略了 switch 指令中的 default case；一個產品化的系統其實不應該如此。由於 LPFS 不維護任何 caches，所以它省略了 *DIO_SET_LOCK_CACHE_STATE*。爲了鎖住 I/O 緩衝區，這個函式首先決定緩衝區的長度（*nbytes*），也就是將「*ir_length* 所表示的傳輸量」乘以「記錄在 VRP 中的區塊大小」。然後再決定緩衝區中的起始 linear page 號碼（*firstpage*）和 pages 個數（*npages*）。*_LinPageLock* 會把那些 pages 鎖定於實際記憶體（physical memory）。使用 *PAGEMAPGLOBAL* flag 是爲了使 locked buffer 在所有的位址空間（address context，address space）中都可用，這是呼叫 IOS 時一個必要而不可或缺的條件。如果你正在執行一個 read 動作，你還應該設定 *PAGEMARKDIRTY* flag，使 paging supervisor 瞭解，memory pages 中的資料已經改變。

如果 *_LinPageLock* 成功，你就爲那個「使用 locked 位址」的真正 I/O 動作計算一個新位址。如果 *_LinPageLock* 失敗，你就設定 *IORF_DOUBLE_BUFFER* flag，強迫 IOS 在自己的 locked page buffers 上，進行自己的資料拷貝動作。

執行 I/O 動作的那些程式碼很類似我們討論過的其他許多實例。如果有錯誤發生，你就需要利用 *IOSMapIORSToI24* 函式，將 IOS 錯誤代碼映射爲一個 MS-DOS 錯誤代碼。

直接磁碟 I/O 動作的最後一個步驟是是將稍早你鎖定的 pages 解鎖。將 locked page number 交給 *_LinPageUnlock* 很重要，提供 *PAGEMAPGLOBAL* flag 也很重要。由於你在呼叫 *_LinPageLock* 時使用這個 flag，如果你在對那些 pages 解鎖時，不供應這個 flag，VMM 會造成記憶體流失（memory leak）的情況，無法回收這些 pages。

FS_DisconnectResource 當你的眾多 volumes 中的某一個被卸載（unloaded）或被刪除（deleted），你可以利用 *FS_DisconnectResource* 函式來完成所需工作。你應該拋棄所有的 caches 並釋放與此 volume 有關的所有資源，包括你自己內部使用的資料結構。例如：

```
int CLocalFileSystem::DisconnectResource(pioreq pir)
   {                // CLocalFileSystem::DisconnectResource
   delete this;
   return pir->ir_error = 0;
   }                // CLocalFileSystem::DisconnectResource
```

FS_FileAttributes *FS_FileAttributes* 用來取得或設定檔案屬性。此函式的第一個實作步驟，毫無疑問是找出用以描述這特定檔案的 directory entry。LPFS 使用一種很蠢的 directory 格式：一整個 disk sector 中只有一個 entry。雖然 LPFS 磁碟中不會有太多 directory 結構（事實上只有一個），我還是寫了一個公用函式，用以執行 directory 查閱功能：

```
BOOL CLpfs::FindDirectoryEntry(ParsedPath* path,
   DirectoryEntry* ep, ULONG& sector,
   BOOL wantfile /* = TRUE */)
   {                        // CLpfs::FindDirectoryEntry
   CPosition pos(path, wantfile);
   GetRootDirectoryEntry(ep, sector);
   while (!pos.AtEnd()
      && FindNextDirectoryEntry(pos, ep, sector))
      pos.Step();           // descend through pathname
   return pos.AtEnd();
   }                        // CLpfs::FindDirectoryEntry
```

這個函式企圖填寫呼叫者的 *ep* 結構，填寫內容是個 directory entry，後者若不是用以表現「被 path 所指定的檔案」（當 *wantfile* 被設爲 TRUE），就是表現「內含該檔案的

一個 directory」（當 *wantfile* 被設爲 FALSE）。其中使用的一些機構或許並沒有任何一般化效用，但如果我沒有深入挖掘該那些機構，我就沒辦法解釋這個例子如何運作。*CPosition* 是我寫的一個 iterator class（譯註），用來幫助在一個解析過的路徑名稱中移動。其宣告如下：

> 譯註：所謂 iterator class，一如 MFC 中的 *POSITION*。你也可以從 C++ Standard Library（或稱 Standard Template Library，STL）中學習所謂的 iterator。基本上 iterator 就是一種「泛型指標」。

```
class CPosition
{
public:

CPosition(ParsedPath* path, BOOL wantfile) {
    m_path = path;
    m_pos = 4;
    m_last = wantfile
        ? path->pp_totalLength
        : path->pp_prefixLength;}

ParsedPath*     m_path;
int             m_pos;
int             m_last;

BOOL            AtEnd() const {return m_pos >= m_last;}
PathElement*    Current() const {
                  return (PathElement*)
                      ((PBYTE) m_path + m_pos);}
void            Step(){m_pos += Current()->pe_length;}
};
```

其中建構式（constructor）記錄一個指標（指向被解析過的路徑名稱），和一個目前位置（由參數傳來）。它也記錄了一個停止位置，當 *wantfile* 被設爲 TRUE，表示 iterator 的使用者將下降至路徑名稱中的最後一個元素；當 *wantfile* 被設爲 FALSE，則表示將停止於最低的 directory level。如果你已經停止於你說要停止的地點，*AtEnd()* 便傳回 TRUE。*Step()* 可以帶你前進至下一個路徑元素。*Current()* 會傳回一個指標，指向目前的路徑元素。

回到對 *FindDirectoryEntry* 的討論，我們產生一個 iterator 以協助在一個解析過的路徑名稱中前進，並呼叫 *GetRootDirectoryEntry* 以求獲得對 directory tree 來回移動的起始點。後繼再呼叫 *FindNextDrectoryEntry*，就會在 pathname 和 directory tree 上來回移動，爲求找出預期的 entry。

可能你會想對 *FindNextDirectoryEntry* 內含的一些程式碼有更多瞭解。基本上，你應該拿你正在搜尋的名稱來和目前磁碟目錄中的每一個 entry 的長名稱及 8.3 名稱做比較（假設你同時保有長名稱和短名稱的話）。爲了準備這些測試，你首先得檢查看看，是否目前的路徑元素是一個短名稱。如果是，就使用 *ShortToLossyFcb* 來建構一個 FCB 格式的名稱，以備稍後使用：

```
PathElement* element = pos.Current();
int elsize = element->pe_length;

BOOL islong = elsize > 12;
USHORT basename[11];
if (!islong)
   ShortToLossyFcb(basename, element-pe_unichars, elsize);
```

（*ShortToLossyFcb* 可以優雅地處理那些「沒有對應之 OEM code page」的 Unicode 字元 -- 以一個底線（underscore）替換之。由於你通常會在磁碟中以 OEM code page 來儲存名稱，所以你應該以此函式取代 *ShortToFcb*）。然後你以迴圈走訪目前 directory 中的所有 entries。如果某個 entry 的長名稱和你所搜尋的名稱長度相同，你就應該把它轉爲全部大寫，然後對每一個 word 一一比較：

```
USHORT ucname[LFNMAXNAMELEN];
UniToUpper(ucname, e.longname, e.namelen*2);
if (memcmp(ucname, element->pe_unichars, elsize-2) == 0)
   break; // i.e., leave the loop over directory entries
```

如果長名稱不吻合，你就將短名稱（可能是 FCB 格式）與稍早的 *ShortToLossyFcb* 結果一一比較（針對每一個 word）：

```
if (!islong
    && memcmp(ep->basename, basename, sizeof(basename)) == 0)
    break;            // short names match
```

一旦 *FindDirectoryEntry* 找到討論中的那個檔案的 directory entry，你的 *FS_FileAttributes* 函式就應該驗證 *ir_flags* 中的子功能碼，例如：

```
int CLpfs::FileAttributes(pioreq pir)
    {                       // CLpfs::FileAttributes
    DirectoryEntry e;
    ULONG sector;

    if (!FindDirectoryEntry(pir->ir_ppath, &e, sector))
        return pir->ir_error = m_error;

    switch (pir->ir_flags)
        {                   // select subfunction
    case GET_ATTRIBUTES:
        pir->ir_attr = e.attr;
        break;
        ...
        }                   // select subfunction

    return pir->ir_error = 0;
    }                       // CLpfs::FileAttributes
```

FS_FindFirstFile, FS_FindNextFile, FS_FindClose　這三個函式協力實作出一般的 Win32 API calls 的檔案搜尋動作。*FS_FindFirstFile* 發動檔案搜尋，搜尋對象可以包括 wildcards，並產生一個 context handle。*FS_FindNextFile* 根據這個 context handle 繼續搜尋動作，直到不再有任何可能爲止。*FS_FindClose* 關閉上述的 context handle。

對一個 directory 做 wildcard 搜尋，其基礎演算法是：對每一個 directory entry 分別比較其長名稱和短名稱（與搜尋目標 -- 一個 pattern（型類）-- 做比較）。Windows 95 使用的搜尋語意比 MS-DOS 所使用的複雜，Windows 95 加上一組所謂「務必吻合」的屬性觀念：不只是檔案名稱必須與 pattern 吻合，也不只是屬性（*FS_FindFirstFile* 所獲得者）必須吻合，「務必吻合」遮罩（mask）中指定的所有屬性也都必須吻合。

讓我舉個例子，說明所謂「務必吻合」遮罩（mask）是如何影響搜尋策略。考慮一個 **(1)** *.* **(2)** 屬性遮罩爲 12h（亦即 *FILE_ATTRIBUTE_DIRECTORY* 和 *FILE_ATTRIBUTE_HIDDEN*）**(3)** 無「務必吻合」遮罩 的搜尋動作。這項搜尋會比對每一個 directory 中的每一個一般檔案、隱藏的 directory、以及隱藏檔案，就像 MS-DOS 一樣。如果此一搜尋含有一個「務必吻合」遮罩（mask）爲 10h（*FILE_ATTRIBUTE_DIRECTORY*)，那麼只有 directory entries（不論隱藏或否）會被拿來比對。

Windows 95 中的檔案搜尋也會爲了如何處理長短名稱而變得複雜。秀一個例子給你看，我就可以好好解釋這項規則。首先是根據某些 flag bits 而做的一些設定，那些 flags 是 IFS Manager 藉由 *ir_attr* 的較高位元組（high-order byte）傳給你的：

```
int matchsemantics = 0;
if (pir->ir_attr & (FILE_FLAG_KEEP_CASE | FILE_FLAG_IS_LFN))
   matchsemantics |= UFLG_NT;
else
   matchsemantics |= UFLG_DOS;
if (pir->ir_attr & FILE_FLAG_WILDCARDS)
   matchsemantics |= UFLG_META;
BOOL hasdot = (pir->ir_attr & FILE_FLAG_HAS_DOT) != 0;

BYTE excludemask = (~pir->ir_attr) & (FILE_ATTRIBUTE_HIDDEN
   | FILE_ATTRIBUTE_SYSTEM | FILE_ATTRIBUTE_DIRECTORY);
```

這段碼產生出一個 *matchsemantics*，將被用來做爲對 IFS pattern 比對函式的一個參數；還有一個 *hasdot*，表示 pattern 中是否內含一個句點（dot）；還有一個 *excludemask*，內含屬性位元（這些位元無法被表現於一個 matching directory entry 之中）。

在一個針對所有 directory entries 的迴圈中，你首先得驗證目前的 entry 是否有正確的屬性：

```
   if ((e.attr & excludemask) || !TestMustMatch(pir, e.attr))
      continue;
```

TestMustMatch 是一個由 Microsoft 於 IFS.H 中提供的巨集，用以實作出「務必吻合」屬性的相關規則。你我都不需要真正瞭解那些規則。接下來你應該拿長短檔名來和

pattern 做比對，像這樣：

```
USHORT thisname[LFNMAXNAMELEN+1];
thisname[UniToUpper(thisname, e.longname, e.namelen*2)/2] = 0;

if (IFSMgr_MetaMatch(pattern, thisname, matchsemantics))
   break;         // found match

thisname[FcbToShort(thisname, e.basename, hasdot)/2] = 0;
if (IFSMgr_MetaMatch(pattern, thisname, matchsemantics))
   break;         // found match
```

IFSMgr_MetaMatch 是「pattern 比對函式」。它需要兩個以 null 爲結束符號並全部大寫的 Unicode 字串，以及一個指示器（上例的 *matchsemantics*），指示使用什麼樣的比對語意。如果是 *UFLG_NT*，表示程式應該使用 Win32 比對法，其中 '*' 是一個正規字元，而 '.' 亦只不過是名稱中的另一個字元。如果是 *UFLG_DOS*（我們雖不在此函式中使用之，卻在 *FS_SearchFile* 函式使用）表示使用 MS-DOS 比對法，其中 '?' 可吻合任何一個字元而 '.' 表示主檔名和副檔名之間的區隔符號。如果是 *UFLG_META*，表示 pattern 內含 wildcard 字元，那就需要比「單純的字串比對」更複雜的工作。

這裡列出本頁以及前頁程式碼所實作出來的搜尋規則：

- 如果 pattern 內含 wildcard 字元（當 *FILE_FLAG_WILDCARDS* 設立），我們將把 *UFLG_META* flag 交給 *IFSMgr_MetaMatch*，使它可以解析它們。
- 如果 pattern 的最後元素是一個長名稱（當 *FILE_FLAG_KEEP_CASE* 或 *FILE_FLAG_IS_LFN* 任一設立），我們將使用 Win32 語意來進行 pattern 的比對。否則我們面對的就是 8.3 相容格式，於是就以 MS-DOS 語意來比對。
- 如果 pattern 之中有一個句點（當 *FILE_FLAG_HAS_DOT* 設立），只要我們所比對的一個 directory entry 不含副檔名，我們將爲其短檔名加上一個尾端句點（需設定 *FcbToShort* 的最後參數爲 TRUE）。這麼做的理由是，MS-DOS 把 pattern "x." 解釋爲「主檔名爲 "x"，無副檔名」，卻把 pattern "x" 解釋爲「主檔名爲 "x"，有無副檔名均可」。這些文字解釋恐怕遠不及來一段程式碼更清楚。

如果一開始的 *FS_FindFirstFile* 呼叫成功，你就得產生一個某種型態的控制區塊，在其中記錄搜尋行動的進行狀況。你應該設定 *ir_fh* 使它等於一個 handle（或許是你的結構的位址），使你稍後能找到這個區塊。你的 *FS_FindNextFile* 函式將使用上述的「狀況資訊」，以便從最近點來重新進行搜尋。IFS Manager 最終會呼叫你的 *FS_FindClose* 函式，釋放該控制區塊。

你還需要完成一個 *hndlfunc* 結構，其位址被 IFS Manager 設於 *ir_hfunc* 中。有一個名爲 *SetHandleFunc*（定義於 IFS.H）的巨集可以幫助你填寫結構內容：

```
#undef vt
#define vt(f) f##Thunk,   // 譯註：## 即所謂 merging operator，是 C/C++ 標準運算子。
                          //       應可在各家編譯器之 online help 中查到。
static hndlmisc findmisc = {
   IFS_VERSION,
   IFS_REVISION,
   NUM_HNDLMISC, {
   vt(EmptyFunc)          // HM_SEEK
   vt(FindClose)          // HM_CLOSE
   vt(EmptyFunc)          // HM_COMMIT
   vt(EmptyFunc)          // HM_FILELOCKS
   vt(EmptyFunc)          // HM_FILETIMES
   vt(EmptyFunc)          // HM_PIPEREQUEST
   vt(EmptyFunc)          // HM_HANDLEINFO
   vt(EnumerateHandle)    // HM_ENUMHANDLE
   }};

SetHandleFunc(pir, FindNextFileThunk,
   EmptyFuncThunk, &findmisc);
```

findmisc 結構提供了某些函式的位址，這些都是 IFS Manager 在搜尋迴圈中可能呼叫的函式。你應該在 *SetHandleFunc* 巨集中提供你的 *FS_FindNextFile* 函式位址。我在實作 LPFS 時，給予 IFS Manager 的是一個 thunk 函式的位址。這個 thunk 函式會從 *ir_rh* 中抽取出一個 *this* 指標，然後呼叫對應的成員函式（與某個 *CLpfs* object 有關聯）。

FS_FlushVolume *FS_FlushVolume* 會把任何懸而未決的輸出資料「清掃（flushes）」到 device 去。*ir_options* 可以內含兩個 flag 位元，用以控制額外行爲。如果

VOL_REMOUNT flag 被設立，你應該拋棄你爲此 device 所收集的所有資訊，並使情況就像 volume 剛被 "mounted" 上去一樣。例如，你可能在某人重新格式化你的 volume 後，收到一個 *VOL_REMOUNT* request，意思是這個磁碟的 "geometry" 可能已被改變。

如果 *VOL_DISCARD_CACHE* flag 被設立，你應該拋棄任何你已經維護的 "cached" 資料緩衝區，不讓它們被寫到 device 去。

最後，你應該送出一個 *IOR_FLUSH_DRIVE* 或是一個 *IOR_FLUSH_DRIVE_AND_DISCARD* request 給 device。例如：

```
int CLpfs::FlushVolume(pioreq pir)
   {                          // CLpfs::FlushVolume
   PIOR ior = CreateIOR((pir->ir_options & VOL_DISCARD_CACHE)
      ? IOR_FLUSH_DRIVE_AND_DISCARD : IOR_FLUSH_DRIVE, 0);
   if (!ior)
      return pir->ir_error = ERROR_NOT_ENOUGH_MEMORY;
   SendCommandAndWait(ior);
   DestroyIOR(ior);           // ignore any error

   return pir->ir_error = 0;
   }                          // CLpfs::FlushVolume
```

FS_GetDiskInfo *FS_GetDiskInfo* 可以獲得磁碟自由空間的相關資訊。真正生活中，你一定有什麼方法可以取得那些資訊。LPFS 只不過是將這些資訊組合起來：

```
int CLpfs::GetDiskInfo(pioreq pir)
   {                          // CLpfs::GetDiskInfo
   pir->ir_length = 512;      // bytes per sector
   pir->ir_size = 4096;       // total allocation units
   pir->ir_sectors = 1;       // number of sectors per alloc unit
   pir->ir_numfree = 42;      // number of free alloc units (a lie)
   return pir->ir_error = 0;
   }                          // CLpfs::GetDiskInfo
```

這個函式的技術文件中說（其實並不正確），一個與 FAT 相容的檔案系統，應該傳回一個指標，指向所謂的 "FAT allocation byte"（放在 *ir_data* 中）。事實上 FSD 應該直接以 *ir_flags* 傳回這個 byte。

FS_GetDiskParms 這個函式傳回 MS-DOS DPB 的真實模式位址。如果你的 FSD 並不維護 DPB，你可以令此呼叫失敗。如果你的確維護有一個 DPB，你可以在處理 *IR_FSD_MOUNT* 時找到它，作法是搜尋整個 DPBs 串列（list），起始點則是 *ir_rh* 所記錄的位址。

FS_Ioctl16Drive *FS_Ioctl16Drive* 函式在 volume 之上執行一個 I/O control（IOCTL）動作。大部份的 FSDs 會把這個 request 直接遞交給 disk device。例如：

```
int CLocalFileSystem::Ioctl16Drive(pioreq pir)
   {                       // CLocalFileSystem::Ioctl16Drive
   PIOR ior = CreateIOR(IOR_GEN_IOCTL, IORF_SYNC_COMMAND);
   if (!ior)
      return pir->ir_error = ERROR_NOT_ENOUGH_MEMORY;

   // macro to simplify grossly long field name access:
   #define ioctl(f) ior->_ureq.sdeffsd_req_usage._IOR_ioctl_##f

   CLIENT_STRUCT* pRegs = (CLIENT_STRUCT*) pir->ir_cregptr;
   ioctl(client_params) = (ULONG) pRegs;
   ioctl(function) = _ClientAX;
   ASSERT(_ClientBL-1 == m_drive);
   ioctl(drive) = m_drive;
   ioctl(control_param) = _ClientCX;

   if (pir->ir_options & IOCTL_PKT_LINEAR_ADDRESS)
      ioctl(buffer_ptr) = (ULONG) pir->ir_data;
   else
      {                    // 16-bit IOCTL
      ior->IOR_flags |= IORF_16BIT_IOCTL;
      ioctl(buffer_ptr) = (ULONG) Client_Ptr_Flat(DS, DX);
      }                    // 16-bit IOCTL

   SendCommandAndWait(ior);
   USHORT status = ior->IOR_status;
   _ClientAX = (USHORT) ioctl(return);
   DestroyIOR(ior);

   if (status >= IORS_ERROR_DESIGNTR)
      status = (USHORT) IOSMapIORSToI21(status);
   else
      status = 0;     // success after error is still success
```

```
return pir->ir_error = (int) status;
}                       // CLocalFileSystem::Ioctl16Drive
```

這個例子緊密地反映出 DDK 中的 VDEF 範例在處理 IOCTL request 時的作法。它首先產生並初始化一個 IOS I/O request descriptor，用以執行一個 *IOR_GEN_IOCTL* request。IOCTL request 的格式並沒有在任何地方公開過，但你可以從這個例子中推演出來：

- *_IOR_ioctl_client_params* 指向發出此一 IOCTL request 的 VM 的 client register structure（CRS）。你可以在 *ioreq* 結構的 *ir_cregptr* 欄位中發現一個指標，指向該結構（CRS）。
- *_IOR_ioctl_function* 內含「來自 client 端 AX 暫存器」中的 IOCTL function request。
- *_IOR_ioctl_drive* 內含這個 request 所指定的磁碟機（索引碼，從 0 起算）
- *_IOR_ioctl_control_param* 內含「來自 client 端 CX 暫存器」中的控制參數。
- *_IOR_ioctl_buffer_ptr* 指向 client 的資料緩衝區。如果一個 16 位元 client 程式已經申請這個函式，你可以根據 client 的 DS:DX 暫存器，呼叫 *Map_Flat* 來產生此一位址（這個動作在我的 *Client_Ptr_Flat* 巨集中完成）。如果一個 32 位元 client 程式已經申請了這個函式，你會收到緩衝區位址，如同一個參數。
- *_IOR_ioctl_return* 在函式回返時，內含 IOCTL completion code。

由於你正在與 IOS 打交道，你必須敲入數百個額外的字元來表現這些想法。只不過三行，我就放棄了，並寫下小小的 *ioctl* 巨集（你在上述程式碼中看到了）。沒有這個巨集，我知道每一行我少不了會出現兩個編譯錯誤，而且我也沒辦法有緊密的邏輯思考。

你可能已經注意到，*Ioctl16Drive* 函式使用 *IORF_SYNC_COMMAND* flag，雖然我曾說過，大部份 FSDs 都使用它們自己的邏輯來等待 request。由於我所使用的「command-waiting 邏輯」依賴 *_ureq* 的一個欄位，而 IOCTL request 也需要它，所以後者基本上不使用這裡所呈現的邏輯。

FS_OpenFile　*FS_OpenFile* 是開檔的基本函式，我將在下一節詳細討論它。

FS_QueryResourceInfo *FS_QueryResourceInfo* 將你的檔案系統的基本資訊提供給 IFS Manager。*ir_options* 值是一個 "level" indicator。Local 檔案系統只需要回應一個 level 2 質詢（inquiry），例如：

```
int CLpfs::QueryResourceInfo(pioreq pir)
   {                        // CLpfs::QueryResourceInfo
   if (pir->ir_options == 2)
      {                     // answer level 2 query
      pir->ir_length =
          sizeof(((DirectoryEntry*) NULL)->longname);
      pir->ir_options = FS_CASE_IS_PRESERVED
          | FS_UNICODE_STORED_ON_DISK
          | FS_VOL_SUPPORTS_LONG_NAMES;
      return pir->ir_error = 0;
      }                     // answer level 2 query
   return pir->ir_error = ERROR_INVALID_FUNCTION;
   }                        // CLpfs::QueryResourceInfo
```

你應該設定 *ir_length* 等於你的檔案系統所支援的檔名最長長度。像 VFAT 就支援 255 字元的檔名，我的 LPFS 只支援 234 字元的檔名。你應該使用可以象徵你的檔案系統能力的 flag bits 來設定 *ir_options*。LPFS 記錄的是：它以 Unicode 儲存長檔名並保存其中的大小寫。如果你的檔案系統在此 volume 上有壓縮功能，你也可以登記 *FS_VOL_IS_COMPRESSED*。

FS_RenameFile *FS_RenameFile* 可以對一個或多個檔案重新命名。如果使用者在一個 MS-DOS 的 RENAME 命令中，於 source 端指定 wildcards，並使用長檔名，MS-DOS 會列舉所有的 source 檔案，使你的 driver 得以一再被呼叫以對每個檔案重新命名。IFS Manager 有時候會期望你來做列舉動作，以滿足一個以 8.3 檔名爲基礎的重新命名動作。（例如當一個古老的 MS-DOS 程式使用 INT 21h, function 17h，也就是 FCB 的重新命名功能）。結果，你實作這一函式的方法是首先寫一個迴圈，找到所有的 source 檔。在迴圈中，你需要做以下四件基本事情：

1. 建構檔案的新名稱。如果你已經完成一個 8.3 名稱的 wildcard 比對動作，你需要使用搜尋條件來建立新名稱。如果你是以一個長名稱來對檔案重新命名，

IFS Manager 會給你新名稱。

2. 檢驗並確定目前沒有相同名稱的檔案存在。
3. 根據新的長檔名和短檔名，建構一個新的 directory entry。
4. 要不將 directory entry 覆寫回檔案起始的 directory 中，要不就把它搬移到一個新的 directory。

這裡我只打算討論其中兩件工作：如何爲一個擁有長名稱之檔案建構一個短檔名，以及如何從 *ioreq* 結構的欄位中選擇「標的長檔名」。我想現在的你已經熟悉了這個程序的結果：當你在一個 FAT volume 上對檔案重新命名，VFAT 會建構一個新的短檔名，命名方式是以長檔名的前數個字元爲基礎，再加上一個 ~ 符號以及一個整數。就像長檔名一樣，短檔名也必須是獨一無二的。IFS Manager 提供一個 *MatchBasisName* service，其唯一目的就是讓你輕鬆地選擇一個獨一無二的短檔名。

爲了使用 *MatchBasisName*，你需要一些初始設定：

```
USHORT basename[11];
int maxtail = 0;
CreateBasis(basename, newname->pe_unichars, elsize-2);
```

在這裡，*newname* 指向一個 *PathElement* item，內含長名稱（標的物）。*CreateBasis* 將一如你所知道地，取得檔名最前 8 個字元（如果必要則添加空白），並附加副檔名的前 3 個字元（如果必要也添加空白），產生出一個 11 字元的 "basis" 名稱，這是最終結果（短名稱）的起點。舉個例子，如果標的名稱是 *HelloWorld*.txt，其 basis 名稱將是 HELLOWORTXT。變數 *maxtail* 內含尾端整數的最大值，如果標的目錄（directory）中已經內含 HELLOW~1.TXT 和 HELLOW~3.TXT 兩個檔案，*maxtail* 就會等於 3。

爲了驗證標的名稱尚未存在，你必須使用迴圈，將標的目錄（directory）整個走過一遍，將目錄中的檔案的大寫名稱和 *ir_ppath2* 名稱比較。（*ir_ppath2* 是大寫的標的名稱。你所做的是一個不分大小寫的比較，因爲 Windows 95 把兩個「只有大小寫不同，其餘都相同」的檔名視爲相同。迴圈之中應該有類似這樣的碼：

```
int tail = MatchBasisName(basename, dup.basename);
if (tail > maxtail)
   maxtail = tail;
if (tail == -1 && !newlong)
   return pir->ir_error = ERROR_ACCESS_DENIED;
```

MatchBasisName 把一個 basis 名稱（*basename*）和一個 FCB 格式名稱（其中可能有一個數值尾巴，前導以一個 ~ 符號）拿來比較。比較方式很特殊：

- 如果 directory 名稱有一個數值尾巴（例如 HELLOW~1TXT），*MatchBasisName* 會拿尾巴前的字元來和 basis 名稱比對。如果只有那些字元相同，就傳回尾巴值（一個整數）。因此，HELLOWORTXT 將被視爲與 HELLOW~1TXT 相同，而 *MatchBasisName* 會因此傳回 1。
- 如果 directory 名稱之中並沒有數值尾巴，但是吻合 basis 名稱，*MatchBasisName* 就傳回 -1。如果你正在處理一個 8.3 重新命名動作，這個傳回值表示「名稱重複」。
- 否則，*MatchBasisName* 傳回 0，表示並不吻合。如果你離開此迴圈前並沒有找到一個同義長檔名，你就可以重新命名這個檔案了。至於建構短檔名，可以呼叫 *AppendBasisTail*：

```
AppendBasisTail(basename, maxtail+1);
```

所以，如果你已經有了 HELLOW~1.TXT 和 HELLOW~3.TXT，你最終會建構出 HELLOW~4.TXT 做爲這個重新命名之檔案的短檔名。你可以在兩個地方發現新的長檔名。通常，你會被要求保存新名稱的大小寫，所以你可以使用 *ir_uFName* 中的 Unicode 字串；否則，你可以使用 *ir_ppath* 的最後一個元素（全部大寫）：

```
string_t newname;
if (pir->ir_attr2 & FILE_FLAG_KEEP_CASE)
   newname = pir->ir_uFName;
else
   newname = IFSLastElement(pir->ir_ppath2)->pe_unichars;
```

FS_SearchFile　*FS_SearchFile* 是 *FS_FindFirstFile* 族群系列函式。*ir_flags* 欄位告訴你是否你正在做 find-first 動作（如果 *ir_flags* 等於 *SEARCH_FIRST*）或是一個 find-next 動作（如果 *ir_flags* 等於 *SEARCH_NEXT*）。除了使用一個 MS-DOS 風格的結構來記錄結果之外，你可以使用輕微不同的演算法，以求符合搜尋條件。你可以依靠收到一個 8.3 相容的搜尋條件，於是就可以建立一個 FCB 風格的搜尋條件，用於「比較迴圈」之中：

```
PathElement* name = (PathElement*)((PBYTE) pir->ir_ppath
   + pir->ir_ppath->pp_prefixLength);
USHORT basename[11];
ShortToLossyFcb(basename, name->pe_unichars, name->pe_length);
int matchsemantics = UFLG_DOS;
if (pir->ir_attr & FILE_FLAG_WILDCARDS)
   matchsemantics |= UFLG_META;
```

假設你的 directory entry 記錄了一個 FCB 格式的短檔名，你可以這種方法來配對：

```
if (IFSMgr_MetaMatch(basename, e.basename, matchsemantics))
   break;         // found match
```

也就是說，你呼叫 *IFSMgr_MetaMatch*，夾帶兩個 FCB 格式名稱，和一個 semantics indicator，其中包括 *UFLG_DOS*（由於 *UFLG_DOS* 等於 0，比較精確但比較沒有啓示的說法是，你並沒有在 semantics indicator 之中包含 *UFLG_NT*）。

Tunneling

在量子力學中，tunneling 是一個程序（process），藉由它，一個粒子可以從一個地點移動到另一個地點，不管有無干涉（intervening）或明顯無法克服的能量障礙。IFS Manager 提供了一個類似的性質，幫助32位元的傳統檔案也能使用長檔名。

假設一個對長檔名毫無意識的 16 位元程式，要改寫 *HelloWorld.txt*，並且不准遺失任何資料。一般的安全作法是先產生一個暫時檔，再殺掉原檔案，再將暫時檔改名為與原先相同的檔名。由於 16 位元程式只知道短檔名，所以它遵循以下步驟：

1. 產生一個暫時檔案（例如 FOO.BAR）

2. 殺掉原檔案，它只知道原檔名為 HELLOW~1.TXT。一般而言，殺掉檔案同時也就殺掉了其長檔名。

3. 將暫時檔案重新命名為 HELLOW~1.TXT。

請注意，在這個過程中，長檔名消失了。或許你可以說，長檔名無法從能量障礙中跳脫出來。

為了避免長檔名遺失，IFS Managers 會將長檔名和短檔名記憶 15 秒鐘。如果檔案的產生或重新命名發生於這 15 秒內，就會使用原來的短檔名，使 IFS Manager 迴避掉一個原本要傳給 *FS_OpenFile* 或 *FS_RenameFile* 的所謂 tunneling 結構，一如上例所述。FSD 會禮遇 tunneling 結構中的名稱，使重新產生或重新命名的檔案最終仍有相同的長檔名和短檔名。當然啦，在這過渡時期，可能會有人偷偷地產生一個完全不同的檔案，卻有剛與目前所討論之長檔名相同的長檔名。這種情況下 tunneling 會失敗，因為它不能夠產生一個重複的長檔名。

不消說，我認為 tunneling 太過複雜，不適合實作於我的 LPFS 之中。因此你應該諒解我的 LPFS「不允許刪除任何檔案」的決定。

施行於檔案的各種動作

IFS Manager 在你的 volume 上打開一個檔案，方法是呼叫你的 *FS_OpenFile* 函式。*ir_ppath* 欄位指向一個被解析過的檔案路徑全名，那是開啓標的物。*ir_flags* 內含 mode 和 sharing 選項，*ir_optionsontains* flags 表示如果被指定的檔案已存在或是不存在，要採取什麼行爲。*ir_attr* 內含一些 flags，描述此一解析過的路徑名稱以及一個新生檔案應該保持的屬性。一般而言，開檔是一個複雜的程序。由於 LPFS 只允許你打開一個檔案，而且不能夠被刪除或改名，所以我笨笨地完成了下面這樣的函式：

```
int CLpfs::OpenFile(pioreq pir)
    {                       // CLpfs::OpenFile
    int mode = pir->ir_flags & ACCESS_MODE_MASK;
    int options = pir->ir_options;
    DirectoryEntry e;
    ULONG sector;
    WORD action = 0;
    if (!FindDirectoryEntry(pir->ir_ppath, &e, sector))
        {                   // file not found
        if (options & ACTION_NEXISTS_CREATE)
            m_error = ERROR_ACCESS_DENIED;
        return pir->ir_error = m_error;
        }                   // file not found

    if (!(options & (ACTION_EXISTS_OPEN | ACTION_TRUNCATE)))
        return pir->ir_error = ERROR_FILE_EXISTS;

    action = ACTION_OPENED;
    e.accessed = IFSMgr_Get_DOSTime();
    if (mode == ACCESS_WRITEONLY || mode == ACCESS_READWRITE)
        e.modified = e.accessed;
    WriteSectorNow(sector, (PBYTE) &e);

    #undef vt
    #define vt(f) f##Thunk,

    static hndlmisc openmisc = {
        IFS_VERSION,
        IFS_REVISION,
        NUM_HNDLMISC, {
        vt(FileSeek)
        vt(CloseFile)
```

```
    vt(CommitFile)
    vt(LockFile)
    vt(FileDateTime)
    vt(EmptyFunc)          // NamedPipeUNCRequest
    vt(EmptyFunc)          // NamedPipeHandleInfo
    vt(EnumerateHandle)
    }};

CFile* fp = new CFile(this, &e, sector);
if (mode == ACCESS_WRITEONLY || mode == ACCESS_READWRITE)
    {                      // open for writing
    fp->m_flags |= CFile::FF_OUTPUT;
    if (options & ACTION_TRUNCATE)
       TruncateFile(fp);
    action = ACTION_REPLACED;
    fp->m_pos = fp->m_size; // append from here
    }                      // open for writing

pir->ir_fh = (fh_t) fp;
pir->ir_dostime = e.modified;
pir->ir_size = e.size;
pir->ir_attr = e.attr;
pir->ir_options = action;
SetHandleFunc(pir, ReadFileThunk, WriteFileThunk,
    &openmisc);

return pir->ir_error = 0;
}                          // CLpfs::OpenFile
```

FindDirectoryEntry 嘗試尋找那個描述「我們正在開啓之檔案」的 directory entry。如果搜尋失敗，而 *ACTION_NEXISTS_CREATE* action flag 有被設立的話，真正的檔案系統會產生一個新檔案，但 LPFS 沒有這個能力。尋找檔案可能是個錯誤，除非呼叫者想要打開一個既存檔案。有兩種不同的 action flags 用以表示說，打開一個原已存在的檔案是安全的：*ACTION_EXISTS_OPEN* 意味呼叫者想要讀取或附加資料到一個既存檔案上，*ACTION_TRUNCATE* 意味呼叫者想要從頭覆寫一個既存檔案。

如果你的檔案系統爲每一個檔案動作記錄了一個時間戳記（timestamp），你可以呼叫 *IFSMgr_GetDOSTime* 以捕捉目前時間，並重寫 directory entry。你也可以簡單地捕捉並記錄一個「最後修改」時間戳記。

你或許需要一個結構，用以追蹤 volume 上的每一個被開啓檔案。我定義了下面的 class 來滿足目標：

```
class CFile
   {                          // CFile
   public:
      CFile(CLpfs* fs, DirectoryEntry* ep, ULONG dirsector);
      ~CFile();

      CFile*  m_next;         // next open file
      CFile*  m_prev;         // previous open file
      CLpfs*  m_fs;           // owning file system object
      ULONG   m_direntry;     // sector where directory entry is
      ULONG   m_size;         // current size of file
      ULONG   m_pos;          // current position in file
      ULONG   m_sector;       // sector where file is located
      BYTE    m_flags;        // flags
      enum FILEFLAGS {
         FF_OUTPUT = 0x01,    // opened for output
         };
   };                         // CFile
```

開啓一個檔案，包括產生一個 *CFile* object。此外，對於一個輸出檔，你還必須將其長度截割爲 0。

最後，你應該以檔案相關資訊填寫 *ioreq* 結構的數個欄位：*ir_fh* 持有你的 file handle（或許是你的某個內部資料結構的位址），*ir_dostime* 持有此檔的最後修改時間，*ir_size* 持有檔案的目前大小，*ir_options* 持有一個代碼，描述你如何開啓檔案，*ir_attr* 持有檔案的實際屬性。你還需要填寫一個隸屬於 IFS Manager 的 *hndlfunc* 結構，填入的是「準備在檔案上執行動作的函式」的位址。

順帶一提，一個真正的檔案系統需要完成許多額外檢驗，才能正確實作出 *FS_OpenFile*。舉個例子，如果檔案系統正接管的一個檔案是被開啓於 Windows 95 啓動之前的真實模式中，*ir_ppath* 可能是 NULL。Windows 95 DDK 文件中對許多其他必要的檢驗都有說明，你可以在 DDK 的 "Design and Implementation Guide" 中的 "MS-DOS/Win32" 那一節中的 "File System Driver Reference" 之下的 "FS_OpenFile" 小節找到。

對於已開啓檔案的各種動作

你的 local FSD 提供 8 個函式，係用於已開啓的檔案身上。我將討論其中最重要的數個。請參考 DDK 之中對於 *FS_CommitFile*, *FS_EnumerateHandle*, 和 *FS_LockFile* 的說明，因爲這裡沒有討論它們。

FS_CloseFile *FS_CloseFile* 會將任何輸出緩衝區的內容清理到磁碟之中，刪除檔案的內部結構，並將某一被開啓檔案的一系列動作善後掉。由於 LPFS 並不對資料做 buffering 或 cache 工作，此函式實作起來也就十分簡單：

```
int CLpfs::CloseFile(pioreq pir)
   {                          // CLpfs::CloseFile
   CFile* fp = (CFile*) pir->ir_fh;
   m_error = 0;
   if (fp->m_flags & CFile::FF_OUTPUT)
      {                       // file was open for output
      DirectoryEntry e;
      ReadSectorNow(fp->m_direntry, (PBYTE) &e);
      e.size = fp->m_size;
      m_error = WriteSectorNow(fp->m_direntry, (PBYTE) &e);
      }                       // file was open for output
   delete fp;
   pir->ir_pos = 0;           // no file locks to remember
   return pir->ir_error = m_error;
   }                          // CLpfs::CloseFile
```

這個函式重新寫入 directory entry 以捕捉一個輸出檔案的大小的任何變化，並刪除 *CFile* object，那是 LPFS 用來追蹤檔案用的。當整個 volume 被鎖住，IFS Manager 在檔案層面提供一個方法，使鎖定狀態能夠保持。這個機制包括令 *FS_CloseFile* 利用 *ir_pos* 傳回一個結構，詳述任何目前尚未解決的鎖定狀態（locks）。此機制實在已經超越了本書範圍。

FS_FileSeek *FS_FileSeek* 是一個建議的 service，允許一個 FSD 將其對檔案的 pre-fetches（預先取得）最佳化。 這個函式之所以是建議性的，因爲 read 和 write 函式都提供有一個檔案位置，會改寫被 FSD 記錄的任何東西。無論如何，我還是決定追蹤 LPFS

的目前位置，甚至雖然它從不被使用於任何地方：

```
int CLpfs::FileSeek(pioreq pir)
   {                          // CLpfs::FileSeek
   CFile* fp = (CFile*) pir->ir_fh;
   ULONG pos = pir->ir_pos;
   switch (pir->ir_flags)
      {                       // select on seek origin option
   case FILE_BEGIN:
      break;                  // relative to beginning
   case FILE_END:
      pos += fp->m_size;      // relative to file size
      break;
   default:
      ASSERT(FALSE);
      break;
      }                       // select on seek origin option
   fp->m_pos = pos;
   pir->ir_pos = pos;
   return pir->ir_error = 0;
   }                          // CLpfs::FileSeek
```

我很驚訝（就像或許你現在也有的驚訝一樣）這裡爲什麼沒有第三個 case，像我們通常在一個檔案搜尋動作中會看到的那樣：「相對於目前位置而移動」。由於 IFS Manager 在任何時刻都知道檔案的目前位置，所以這裡不需要有第三個 case。IFS Manager 並不需要知道檔案的大小（它可能因爲新資料被寫入檔案而時時有所改變），因此只要處理 *FILE_BEGIN* 和 *FILE_END* 這兩個 case。

FS_FileDateTime　*FS_FileDateTime* 可以設定或取出「你可能指定給一個被打開檔案的三個時間」中的一個。你必須做出一個「最後被修改」的時間，你也可以做出一個「產生時間」和一個「最後被處理」的時間。本機（local）的檔案動作並不會觸發這個函式，但我發現，如果透過網路寫檔，就會呼叫它。LPFS 這樣地實作這個函式：

```
int CLpfs::FileDateTime(pioreq pir)
   {                          // CLpfs::FileDateTime
   CFile* fp = (CFile*) pir->ir_fh;
   DirectoryEntry e;
   ReadSectorNow(fp->m_direntry, (PBYTE) &e);
```

```
    BOOL changed = FALSE;
    switch (pir->ir_flags)
        {                       // perform requested operation
    case GET_MODIFY_DATETIME:
        pir->ir_dostime = e.modified;
        pir->ir_options = 0;
        break;

    case SET_MODIFY_DATETIME:
        if (!(fp->m_flags & CFile::FF_OUTPUT))
            return pir->ir_error = ERROR_ACCESS_DENIED;
        e.modified = pir->ir_dostime;
        changed = TRUE;
        break;
    ...
    }                           // perform requested operation

    if (changed)
        WriteSectorNow(fp->m_direntry, (PBYTE) &e);
    return pir->ir_error = 0;
    }                           // CLpfs::FileDateTime
```

我想此間並沒有什麼特別值得注意的事情。

FS_ReadFile *FS_ReadFile* 將資料從檔案傳輸到一塊記憶體緩衝區中。要正確實作出這個函式，需得花費許多工夫，大部份是因爲你應該非同步地（asynchronously）執行任何 I/O 動作（以儘量使 CPU 和週邊裝置能同時各自運作，提昇系統效率），同時也因爲你應該和 VCACHE driver 一起運作來管理一個 data cache。LPFS 以十分輕便的手法來實作此一函式：

```
int CLpfs::ReadFile(pioreq pir)
    {                       // CLpfs::ReadFile
    CFile* fp = (CFile*) pir->ir_fh;
    PBYTE dp = (PBYTE) pir->ir_data;
    ULONG nbytes = (ULONG) pir->ir_length;
    ULONG pos = (ULONG) pir->ir_pos;

    if (pos > fp->m_size)
        pos = fp->m_size;
    if (pos + nbytes > fp->m_size)
        nbytes = fp->m_size - pos;
```

```
    BYTE data[512];
    ReadSectorNow(fp->m_sector, data);
    memcpy(dp, data + pos, nbytes);

    pos += nbytes;
    fp->m_pos = pos;

    pir->ir_pos = pos;
    pir->ir_length = nbytes;
    return pir->ir_error = 0;
    }                       // CLpfs::ReadFile
```

如果你要正確實作此一函式，必須使用一組 sector-sized buffers，並使用一個或多個 IOR_READ requests 非同步地（asynchronously）填充其內容。你還需要使用 VCACHE 措施來維護一個 disk records 的 cache，以儘量減少實際的 I/O 動作。你的 *FS_ReadFile* 函式應該傳回 *ERROR_IO_PENDING* 以警告 IFS Manager 說程序正在行進。在你已經讀取 sectors 以滿足特定的 request 之後，你就應該把資料從 sector buffers 拷貝過來並呼叫 *IFSMgr_CompleteAsync*，表示完成。

FS_WriteFile　*FS_WriteFile* 是最後一個重要的 handle-based requests。它把資料從一塊記憶體緩衝區傳送到檔案去。要正確實作此一函式，需花費許多工夫，因爲就像 *FS_ReadFile* 一樣，你需要維護一個內含資料的 sector-sized buffers cache，並以非同步方式進行實際的寫入動作。LPFS 以一個十分愚蠢的方式來實作這個函式：

```
int CLpfs::WriteFile(pioreq pir)
    {                       // CLpfs::WriteFile
    if (ReadOnly())
        return pir->ir_error = ERROR_WRITE_PROTECT;

    CFile* fp = (CFile*) pir->ir_fh;
    ASSERT(fp->m_flags & CFile::FF_OUTPUT);
    PBYTE dp = (PBYTE) pir->ir_data;
    ULONG nbytes = (ULONG) pir->ir_length;
    ULONG pos = (ULONG) pir->ir_pos;

    if (pos > 512)          // maximum allowed file size for LPFS
        pos = 512;
    if (pos + nbytes > 512)
```

```
        nbytes = 512-pos;

    if (nbytes == 0)
        {                   // truncate file
        fp->m_pos = pos;
        TruncateFile(fp);
        }                   // truncate file
    else
        {                   // write some data
        BYTE data[512];
        ReadSectorNow(fp->m_sector, data);
        memcpy(data + pos, dp, nbytes);
        WriteSectorNow(fp->m_sector, data);
        }                   // write some data

    pos += nbytes;
    fp->m_pos = pos;
    if (pos > fp->m_size)
        fp->m_size = pos;

    pir->ir_length = nbytes;
    pir->ir_pos = pos;
    return pir->ir_error = 0;
    }                       // CLpfs::WriteFile
```

ReadOnly 是一個輔助函式，用來檢查此一 volume 是否爲唯讀（read-only）：

```
BOOL CLocalFileSystem::ReadOnly()
 {                          // CLocalFileSystem::ReadOnly
 return (m_vrp->VRP_event_flags & VRP_ef_write_protected) != 0;
 }                          // CLocalFileSystem::ReadOnly
```

這份實作碼甚至不允許檔成長超過 512 bytes，我知道你一定不會以此做爲一個真正的檔案系統。

林偉博註：如果想對 Windows 95 的可安裝檔案系統，做更進一步的探討，可參考 *Inside The Windows 95 File System*（Stan Mitchell/O'Reilly/1997.05）。

侯俊傑註：如果想深入瞭解 Windows NT 的檔案系統，則可參考 *Windows NT File System Internals - A Developer's Guide*（Rajeev Nagar/O'Reilly/1997.09）。

第 17 章

DOS 保護模式介面（Protected Mode Interface）

我決定將大多數人在 Windows 系統程式設計領域中以之做爲起點的主題，拿來做爲本書的句點。在 Windows 3.0 和 3.1 中，ring3 程式常常需要大量仰賴 Dos Protected Mode Interface（DPMI）的幫助。DPMI 提供了一個對 Virtual Machine Manager（VMM）的介面，可用來處理系統層面的事物，例如配置記憶體和 selectors、規劃 debug 暫存器等等。DPMI 仍然存在於 Windows 95 之中，16 位元程式仍然需要靠它完成某些任務。

DPMI 是一組以暫存器爲導向（register-oriented）的 API，軟體中斷 INT 31h 可以讓 ring3 保護模式程式從一個 DPMI host（如 Windows 95）身上獲得系統服務。DPMI 在 1989 和 1990 期間被發展出來，並成爲 Windows 3.0 的一部份。Microsoft 爲此發展出一套 INT 31h 介面，介於 Windows KRNL386 模組和 Windows 3.0 VMM 之間。在此同時，其他重要軟體（如 Lotus 1-2-3 v3）的開發廠商正需要 DOS extender 技術來獲得對保護模式和延伸記憶體（extended memory）的控制。DOS extenders 不但需要監控整部機器（並因此必須能夠發出高權級（privileged）指令如 LGDT 和 MOV into CR0），還必須在 Virtual Control Program Interface（VCPI）相容的記憶體管理器保護下執行。後來，Enhanced-mode

Windows 3.0 接管了整部機器，也否定了上述兩者（DOS extender 和 VCPI 記憶體管理器）的必要性。

爲了避免與大部份普及性高的桌上型軟體不相容，Microsoft 決定公開 INT 31h 介面。一如我在第 4 章所說，Microsoft 和其他數家廠商組成了 DPMI 標準委員會，負責起草此一介面並出版文件[1]。由於委員會認爲應該有更多性質加到此介面中，因此把這份標準命名爲 0.9 版。DPMI 後來又進步到 1.0 版，但是 Microsoft 只實作出其中一部份。Windows 3.x 和 Windows 95 以及 Windows NT 各版本都只固守著 DPMI 0.9，其他廠商如 Qualitas 和 Quarterdeck Office System 倒是在其記憶體管理器商業產品中加入了額外性質。但當你面對 Windows，你面對的是 DPMI 0.9！

DPMI 內含一個方法，可以讓應用程式從真實模式切換到保護模式，也內含許多函式，可以提供給保護模式應用程式使用。這些 API 函式讓程式得以做記憶體和 selectors 的管理、真實模式和保護模式的切換、以及存取共享資源如 debug 暫存器和算術協同處理器等等。16 位元 Windows 程式有時候需要使用其中一些函式來執行系統層面的工作。

切換到保護模式

DPMI 環境中的全部程式都是起源於真實模式（事實上可能是 V86 模式，但其間的差異對於程式而言不很重要）。應用程式可以發出 INT 2Fh / function 1687h 偵測 DPMI host 存在與否，同時獲得「模式切換常式」的位址：

```
toprot   dd 0
hostsize dw 0

mov  ax, 1687h                ; get DPMI switch routine address
```

[1] 這份文件就是由 DPMI 委員會發行的 *DOS Protected Mode Interface（DPMI）Specification, Version 0.9*（1990）。這個版本已經絕版，但你可以向 Intel 公司索取 1.0 版，文件號碼是 240977-001。

```
int  2Fh                        ;   ..
test ax, ax                     ; AX unchanged if no DPMI host,
jnz  nodpmi                     ;   so skip if not 0

mov  word ptr toprot, di        ; ES:DI -> mode switch routine
mov  word ptr toprot+2, es      ;    ..
mov  hostsize, si               ; SI = size of host's memory block
```

INT 2Fh, function 1687h 會偵測 DPMI host 存在與否。如果不存在，此程式 "hooked" INT 2Fh 時就不會改變 AX 暫存器的內容，因此中斷回返後 AX 仍然內含 1687h。DPMI host 會把 AX 設爲 0，這也表示了 host 的存在。Host 還會把模式切換常式的位址放到 ES:DI 暫存器中，並將它的記憶體需求設定於暫存器 SI（以 paragraphs 爲單位），應用程式應該在模式切換前配置出這一塊 MS-DOS 記憶體。表 **17-1** 說明在 DPMI host 回返之後，各暫存器的意義。

暫存器	內容
AX	0 表示成功。非 0 表示 DPMI host 不存在。
BX	旗標值，只有 bit 0 有意義。如果此旗標設立，表示 DPMI host 支援 32 位元程式。當然啦，Windows 95 總是設立此旗標。
CL	CPU 型別（2 表示 80286，依此類推）
DH	DPMI 主版本號碼（00h）
DL	DPMI 次版本號碼（5Ah）
SI	DPMI 私用的記憶體空間，以 paragraphs 爲單位。
ES:DI	模式切換常式的位址

表 17-1　一次成功的 INT 2Fh / function 1687h call 之後，暫存器的內容。

準備好之後，應用程式可以把 host 所需的記憶體位址設定在 ES 暫存器中，然後呼叫模式切換常式：

```
mov  bx, hostsize           ; size of host data area
test bx, bx                 ; any data area needed?
jz   @F                     ; if not, okay
```

```
   mov  ah, 48h                ; if so, allocate DOS memory
   int  21h                    ;   ..
   jc   error                  ; skip if error
   mov  es, ax                 ; ES -> host data area
@@:
   xor  ax, ax                 ; bit 0 == 0 => 16-bit app
   call [toprot]               ; switch to protected mode
   jc   error                  ; skip if error
```

上述程式碼使用 MS-DOS INT 21h, function 48h 配置一塊記憶體給 DPMI host 使用，並將 ES 暫存器設定爲配置而來之記憶體的 paragraph 位址。然後呼叫模式切換常式。如果 DPMI host 切換失敗，回返時 carry flag 會設立，而 CPU 仍處於真實模式。否則 CPU 當然就進入了保護模式。由於保護模式程式用的是 segment selectors 而非 paragraph numbers，所以 DPMI host 必須在回返至應用程式前，改變 segment 暫存器。其新內容如下：

- CS 暫存器內含一個 code selector，起始處與原來真實模式 CS segment 的線性位址相同，但其 limit 欄位爲 64KB。舉個例子，如果你是在真實模式的 2DF0h:908h 處呼叫模式切換常式，那麼 CS selector 的 base 位址就是 0002DF00h。
- DS 暫存器內含一個 data selector，起始處與原來真實模式 DS segment 的線性位址相同，但其 limit 爲 64KB。
- SS 暫存器內含一個 data selector，起始處與原來真實模式 SS segment 的線性位址相同，但其 limit 爲 64KB。如果在模式切換之前，SS 和 DS 有相同的 paragraph number，回返時它們將擁有相同的 selector。
- ES 暫存器內含一個 data selector，其 base 位址與應用程式的 Program Segment Prefix（PSP）相同，而其 limit 爲 256 bytes（標準 PSP 的大小）。此外，DPMI host 會以一個保護模式 selector 取代 PSP offset 2Ch 中的環境指標（environment pointer）。
- FS 和 GS 暫存器（如果有的話）將爲 0。
- 所有 general 暫存器都不會改變，但如果你將 AX 設爲 1，表示這是一個 32 位元程式的話，ESP 暫存器的較高半部（high half）會變爲 0。

一旦程式進入保護模式，就可以被稍後數節即將描述的 DPMI 服務驅動。如果你的程式是一個 DOS extender，就可以使用那些服務來載入並執行一個應用程式。如果你要的只是執行一個需要許多記憶體的程式，這個程式並不需要 DOS extender 的監控，因爲 Windows(及其他 DPMI hosts)對於標準的 MS-DOS 和 BIOS 中斷提供了很好的支援。請參考 *Testing the Windows DOS Extender* 一文 （作者 Walter Oney, Windows/DOS Developer's Journal, 1994/02），其中有詳細的說明。事實上我在本章所建立的 DPMI 實例，是一個短小的 "bootstrap" 程式，可以透明化地將一般的真實模式 C 程式轉換爲一個 DPMI client 程式，不需要任何商用的 DOS extender 協助。

你可以執行 INT 21h function 4C，退出 DPMI 保護模式，這也是退出 MS-DOS 的一般作法。請不要嘗試其他 MS-DOS 中斷，因爲它們並不能夠在 DPMI 中正常運作，除非你的 DOS extender 攔截了它們並且轉換爲 INT 21h, function 4C call。

DPMI 函式

保護模式程式發出 DPMI requests 的方式是，將參數放進 general 暫存器和 segment 暫存器中，再將一個函式代碼放進 AX，然後發出一個軟體中斷 31h。位址 000C4h 上的記憶體（譯註：每一個中斷向量是 4 bytes，所以 31h*4 = 49*4=196=C4h），將不再內含 INT 31h 的真實模式中斷向量，而是 JMP 指令的一部份，目的是爲了相容於老舊的 CP/M 作業系統。由於記憶體內並非中斷服務常式位址，當 DPMI 創始之初，世上應該沒有任何一個真實模式軟體和 INT 31h 指令有關。保護模式 IDT 可以內含一個 gate 指向 DPMI host，這並不會衝擊到「爲了傳統目的而使用真實模式中斷向量」的那些真實模式程式。某些 DPMI 函式需要指標做爲參數，如果你在模式切換時宣佈你的程式是一個 16 位元程式，那麼你就必須提供 16:16 遠程指標（也就是指標之中有一個 16 位元 selector 和一個 16 位元 offset）做爲參數。如果你的程式是 32 位元 client，你必須提供 16:32 遠程指標（也就是指標之中有一個 16 位元 selector 和一個 32 位元 offset）。我將遵循 DPMI 的規定，並以 ES:[E]DI 的方式表示指標參數是 16:16 或 16:32。括弧中的 [E] 表示用的是 32 位元程式中的擴充（extended）暫存器（如 EDI），否則就是 16 位元程式中的暫存器（如 DI）。

32 位元 DPMI Clients 讓我重複一次，雖然 Windows 95 同時支援 Win16 和 Win32 程式，其核心其實是個 16 位元 DPMI client。只有那些在 32 位元 DOS extender（如 Phar Lap Software 的 TNT）中執行的程式，才是 32 位元 DPMI clients。並沒有太多理由使你需要在 Windows 95 中使用一個 32 位元 DOS extender，因爲你可以寫 Win32 console 程式取而代之。但如果你需要回溯相容於 Windows 3.1，或者如果你的客戶要求的是 MS-DOS 應用程式，你只好使用它囉。如果你必須這麼做，請確定你的 DOS extender 至少支援 Win32 API 中非圖形介面的部份，否則你會面臨移植問題。

DPMI host 回返時，如果 carry flag 設立，表示有錯誤發生；否則表示成功。除了少數例外，host 會保留所有「未被明白當做輸出用」的暫存器內容。唯一例外發生於那些「令 selectors 失效」的函式身上，此時 host 通常會將它所發現的任何一個「持有新的無效值」的 segment 暫存器設爲 0。

DPMI 以群組方式來組織其函式。函式代碼的最高 8 位元（將被放在 AH 暫存器中）表示函式群組（請看表 **4-4**）。較低 8 位元表示群組中的一個函式。例如，群組 03h 中的函式用來呼叫真實模式軟體，而函式 0301h 是所謂的 "Call Real Mode Procedure With Far Return Frame"。雖然，用於協同處理器管理的 0Exxh services，嚴格來說屬於 DPMI 1.0，但所有 0.9 hosts 都有實作它們，因爲如果想在保護模式應用程式中適當地支援浮點數運算，這些 services 是必要的。

我並不想在這裡囉囉唆唆地解釋所有的 DPMI 函式。你可以看 DPMI 0.9 規格書，或是 Ralf Brown 和 Jim Kyle 合著的 ***PC Interrupts*** 一書（Addison-Wesley, 1991）。不過我認爲完整地看看 DPMI 的語彙是有幫助的，所以我要給你一些實例，示範如何使用 DPMI 主要函式。我所展示的是 Windows 應用程式仍然可能使用的部份。

使用 Selector 管理函式

DPMI 函式中最重要的一群是 00xxh，用來做 selector 管理。你可以使用 0000h 函式來配置一個或多個 descriptors。例如：

```
mov   ax, 0000       ; function 0000: allocate descriptor
mov   cx, 2          ; we want two descriptors
int   31h            ; get descriptor (return in AX)
jc    initfail       ; die if can't
```

這段碼將 DPMI 的 "Allocate LDT Descriptors" 函式代碼載入 AX 之中，並設定 CX 爲 2，表示我們要配置兩個 descriptors，然後發出 31h 中斷，通知 DPMI host。回返時，如果 host 沒有完成任務，carry flag 將設立；否則 carry flag 會被清除而 AX 內含第一個被配置的 selectors。這個 selectors 是 ring3 LDT data selectors，這是可以確認的，因爲它們必定吻合 *xxx7h* 或 *xxxFh* 的型式。它們的 base 和 limit 都是 0。如果沒有更多動作加諸於它們身上，那麼你能夠做的就是經由這兩個 selectors 在目前 VM 的線性位址 0 處讀（或寫）單一一個 bytes。

設定 Base 和 Limit

配置一個 selector 之後，下一個動作是設定其 base 和 limit。函式 0008h 用來設定 limit。要設定一個大小爲 64-KB 的 segment ，可以這麼做：

```
mov   ax, 0008h          ; function 0008h: set segment limit
mov   bx, es             ; BX = selector
xor   cx, cx             ; CX:DX = new limit
xor   dx, dx             ;   ..
dec   dx                 ;   (namely, 64 KB)
int   31h                ; cal DPMI to set limit
jc    initfail           ; die if can't
```

呼叫函式 0008h 時，你必須先將 BX 設定爲目標物（一個 selector），然後將 CX:DX 設爲新的 limit，其值應該比 segment 的長度少 1，這就是爲什麼我在本例中設定爲 0000h:FFFFh 的緣故。如果你的程式是一個 32 位元 DPMI client，你必須把 limit 切割放進 CX:DX 之中。

找出 segment 長度　雖然 DPMI 的 0008h 函式讓你改變一個 segment 的 limit，但卻沒有任何 DPMI 函式可以讓你查詢 limit。你可以使用 LSL（Load Segment Limit）指令輕易知道 segment 到底多大。

函式 0007h 用來設定一個 segment 的 base：

```
mov   ax, 0007h          ; function 0007h: set segment base
mov   bx, es             ; BX = selector
mov   ecx, edx           ; CX:DX = base address to set
shr   ecx, 16            ;   ..
int   31h                ; set base address for target segment
jc    initfail           ; die if error setting base
```

在這一段碼中，我們首先讓 ES 持有目標物（一個 selector），並讓 EDX 持有新的 base。然後把 selector 放進 BX 中，把 base 放進 CX:DX 中，再發出 INT 31h, function 0007h。再強調一次，即使是一個 32 位元 client，也必須使用 CX:DX 來存放 base。

配置 多個 Selectors

函式 0000h 傳回一個 selector，指向爲你配置的一系列連續 selectors 中的第一個。爲了找出後繼的 selectors，你必須爲第一個 selector 加上一個數值，此值可能隨著不同的 DPMI hosts 而有變化。呼叫 DPMI 函式 0003h 可以獲得這個所謂的 selector 累加值：

```
mov   ax, 0003h      ; functon 0003h: get selector increment value
int   31h            ;   (return in AX)
mov   selincr, ax    ; save for later use
```

在 Windows 95 中，上述動作的傳回值總是 8。如果你十分在意移植性，那麼就應該藉由函式 0003h 的傳回值取得下一個 selector 而不是直接使用 8。一般情況下，配置多個 selectors 的主要理由是爲了管理 huge 陣列。huge 屬性告訴 16 位元 C 編譯器說，你的陣列可能超過 64-KB segment。Microsoft 編譯器所產生的陣列定址碼會假設「鄰近的 segments 相差值爲常數 *_AHINCR*」，此值是 *_AHSHIFT* 的 2 冪次方數。*_AHINCR* 和 *_AHSHIFT* 都是外部常數。在真實模式模組之中，它們分別爲 4096 和 12。在 Windows 程式中，它們是從 Windows KERNEL 模組匯入（imported），分爲爲 8 和 3。

Windows 記憶體管理器有一個鮮爲人知的事實，那就是爲了滿足大於 64 KB 的記憶體需求，它會配置多個 segments。例如 GDI bitmaps 就常常超過 64 KB，但卻被視爲一個

單位。當你呼叫 *GlobalAlloc*，Windows 使用「對等於 DPMI 0000h 函式」的 API 函式來配置足夠的、鄰近的 descriptors，以便讓 16 位元程式能夠存取整塊記憶體。它把第一個 descriptor 的 limit 設爲區塊的整個長度（事實上是長度減 1，因爲 limit 欄位總是 segment 大小減 1），最後一個 selector 的 limit 則設定爲區塊完整長度除以 64 KB 的商數。中間各個 segments 的 limits 都設爲 64-KB。於是，應用程式如果知道如何使用 32 位元定址，就可以只使用第一個 descriptor 來存取整塊記憶體，至於 16 位元程式則可以使用 huge 陣列的算術運算取代之。

釋放 Selectors

當你不再需要某個 selector，你應該使用 DPMI 函式 0001h 刪除之。例如：

```
mov  ax, 0001h            ; functon 0001h: cancel selector
mov  bx, selector          ; BX ==> selector to cancel
int  31h                   ; call DPMI
```

如果一直不刪除 selectors，很有可能會耗盡這項資源。因爲一個 LDT 中總共只有 8192 個 descriptors。Windows 應用程式必須時時刻刻把這一點記在心裡。這對於 DOS-extended 應用程式倒不是什麼大問題。

雖然你可以利用函式 0000h 一次獲得多個 selectors，卻必須以函式 0001h 一次刪除一個 selector。

Alias Selectors

如你所知，使用保護模式的部份理由是爲了所謂的「保護」，使你不能夠把資料儲存在 code segment 中，也不能夠在 data segment 中執行程式碼。但有時候你需要規避這項規則。C runtime library 的標準函式 *int86* 就是個例子：這個函式必須修改一個 INT 指令，把它所收到的中斷號碼安插進去。這在真實模式中沒有問題：

```
   mov  al, intno
   mov  byte ptr intinstr+1, al     ; okay in real mode
   jmp  short $+2
intinstr:
   int  00h
```

這段碼把中斷代號儲存於 *intinstr* 處的 INT 指令的第二個 byte，然後執行一個 JMP 指令以清空 CPU 的 **instruction pipeline**（除非是在一部 Pentium 機器上，否則這需得一些工作），然後執行這個剛被修改過的 INT 指令。就像程式註解所示，在真實模式中修改一個指令是被容許的，但是在保護模式中 MOV 指令的目的地如果是 code segment，會引起 general protection fault。爲了繞過「修改程式碼」這個禁令，你可以爲 code selector 產生一個 data alias，像這樣：

```
   mov  ax, 000Ah                  ; function 000Ah: create alias
   mov  bx, cs                     ; BX ==> selector to be aliased
   int  31h                        ; get alias, return in AX
   jc   error                      ; skip if error
   mov  es, ax                     ; access code as data via ES
   mov  al, intno                  ; interrupt # from parameter
   mov  byte ptr es:intinstr+1, al ; modify program using alias
   jmp  short $+2                  ; drain pre-fetch queue
intnstr:
   int  00h                        ; execute modified instruction
   mov  ax, 0001h                  ; function 0001h: cancel selector
   mov  bx, es                     ; BX ==> selector to cancel
   int  31h                        ; go cancel the selector
```

商業化的 16 位元 DOS extenders 內含修正版的 runtime library *int86* 函式，基本內容正如上段程式碼所示。真實生活中，當你第一次執行這個函式，會產生 alias descriptor，你應該把它儲存起來以準稍後還可以再利用，因爲這兩個 DPMI calls 相當耗時。

一個經過省略掉的問題 DPMI host 應該預防一個並未顯示於上的問題。你可以堅定地相信 INT 31h 的處理常式在處理 0001h 函式時，儲存了所有的暫存器值並於最後恢復之。所以會有一個 PUSH ES 指令出現在起頭處，以及一個 POP ES 指令出現在尾端。但是上一段程式碼中的 ES，內含的是你正準備刪除的 selector，當 POP 指令發生，它的內容就成爲一個非法值了，於是 DPMI host 可能會回覆你 general protection fault。然而設計良好的 DPMI 0.9 host 如 Windows，會檢查被存起來的 segment register images（在恢復暫存器原值之前），並將它們之中任何一個內含非法 selectors 者清爲 0。至於 DPMI 1.0 規格則已明確要求 host 得這麼做。所以如果在刪除了一個 selector 之後，你發現你的 segment 暫存器中有一個或多個內容爲 0，請不要驚訝。

BIOS Data Area

位於線性位址 400h 的 BIOS data area，在 Windows 之中有其特殊之處：它在保護模式和真實模式的位址都相同：都是 0040h:0000h。Selector 40h 很顯然是一個 ring0 GDT selector。的確，它是位於 GDT，但其 DPL 卻是 3。意思是 ring3 程式可以用它來做資料參考之用。於是如果你想在 Windows 環境下從一個真實模式或保護模式程式中存取 BIOS data area，只要把 40h 載入一個 segment 暫存器，即可開始動作。

Selector 40h 的這一特性，是有歷史淵源的。我記得好像是 Lotus 1-2-3 v3（以及其他許多普及的 extended-DOS 應用程式）把 40h 內建於數個地方，成爲一個無法重新定位（nonrelocatable）的常數。爲了不要破壞這些重要的應用軟體，Microsoft 決定破例。要知道，原本 ring3 程式只能夠使用 ring3 LDT selectors（可利用 DPMI 函式 0000h 配置而得）。

其他的特殊位址

除了 400h，PCs 上面還有其他特殊的記憶體位址。例如標準 VGA 系統中的文字緩衝區位於線性位址 B8000。雖然 DPMI hosts 不需要（而且通常也不）提供對那些位址的透明化處理動作，一如你對 BIOS data area 的所做所爲，但它們的確支援一個可以簡化處理動作的函式（0002h）。函式 0002h 需要一個真實模式的 paragraph 位址做爲參數，並傳回一個 64-KB data selector。它的作用除了比函式 0000h, 0007h, 和 0008h 的組合更容易使用之外，還有一個優點就是，面對同一真實模式位址總是傳回相同的 selector。

舉個例子，下面程式碼（書附光碟的 \CHAP17\DPMIDEMO 目錄中）會在 VGA 螢幕的第 16 行顯示藍色字串 "Hello, World!"，背景爲青藍色：

```
unsigned int b800;
unsgned short _far *video;
static unsigned char msg[] =
   {'H', 0xB5
   ,'e', 0xB5
   ,'l', 0xB5
   ,'l', 0xB5
   ,'o', 0xB5
```

```
    ,',', 0xB5
    ,' ', 0xB5
    ,'W', 0xB5
    ,'o', 0xB5
    ,'r', 0xB5
    ,'l', 0xB5
    ,'d', 0xB5
    ,'!', 0xB5};

pause();
_asm
    {                           // get selector for video memory
    mov   ax, 0002h             ; function 0002h: segment to descriptor
    mov   bx, 0B800h            ; BX = real-mode segment
    int   31h                   ; get selector (return in AX)
    mov   b800, ax              ; save selector
    }                           // get selector for video memory
vdeo = MAKELP(b800, 15*2*80);
_fmemcpy(video, msg, sizeof(msg));
```

呼叫 DPMI 函式 0002h 時，你應該將 BX 設定爲你想存取的真實模式的 paragraph。如果成功，會在 AX 中傳回 selector。

你並不需要刪除一個以函式 0002 獲得的 selector，你也不需要改變其 base 或 limit。由於以此法配置的一個 selector 絕不會消失，所以使用函式 0002h 要特別謹慎。

在 Windows 程式中，你可以使用一些預先定義好的 selectors，它們由 Windows kernel 產生並匯出。例如 *_B800H* 就被定義於線性位址 000B8000h。其他常被用來參考某些低位址區的符號尚包括 *_0000H*, *_0040H*, *_A000H*, *_B000*, *_C000H*, *_D000H*, *_E000H*, 以及 *_ROMBIOS*（不是 *_F000H*）。

在 Windows 程式中配置 Selector

16 位元 Windows 程式通常不應該使用 DPMI 函式來配置和維護 selector。有一些 API 函式（*AllocSelector*, *SetSelectorBase*, *SetSelectorLimit*）做同樣的事情，而且更快。它們之所以比較快是因爲使用 Windows KERNEL 碼，繞過了 DPMI 並直接處理 System VM 的 LTD。但是它們並沒有全方位的機能，而且有時候你必須使用老舊的 *GlobalAlloc* 函

式來模擬一大塊記憶體的保留並產生一個 selector 指向它。例如 Windows 的 hook 函式所在的 segment 就必須有 Windows kernel 所維護的一個 arena header（換句話說是經由 *GlobalAlloc* 配置），否則 Windows 會假設它們屬於一個已結束而未 unhooking 的程式所有，因而將它們刪除。

arena header　Windows 描述一個全域性記憶體區塊時所使用的控制區塊。

使用記憶體管理服務

DPMI 提供了一些用以管理 extended 記憶體和 V86 記憶體的函式。人們之所以需要 DOS extenders，第一個主要原因是爲了獲得 extended 記憶體（譯註：1MB 以上的記憶體）。爲配置一塊 extended 記憶體，可以使用 DPMI 函式 0501h：

```
mov   ax, 0501h               ; functon 0501h: allocate memory
xor   cx, cx                  ; BX:CX = length (128 KB)
mov   bx, 2                   ;   ..
int   31h                     ; (retuns linaddr in BX:CX)
jc    err501                  ; skip if error
mov   word ptr hmem, di       ; SI:DI = memory block handle
mov   word ptr hmem+2, si     ;   ..
```

函式 0501h 要求你在 BX:CX 中（甚至即使你的程式是個 32 位元 client）放置你希望配置的記憶體大小，並傳回兩個值：在 SI:DI 中傳回 handle，在 BX:CX 中傳回 base 線性位址。你可以使用 base 線性位址來設定你將存取的這塊記憶體的 selector base。記住，有了這個 handle，你才能夠利用 0502h 函式釋放記憶體：

```
mov   ax, 0502h               ; function 0502: free memory block
mov   di, word ptr hmem       ; SI:DI = memory block handle
mov   si, word ptr mem+2
int   31h                     ; release memory
```

一如你所猜測，函式 0501h 是 *_PageAllocate*（一個 VxD call）的一層薄包裝，而 0502h 是 *_PageFree* 的一層薄包裝。在 VxD 層面，某些 VxD 會注意 *End_PM_App* 系統控

制訊息，此訊息用來表示保護模式程式結束。如果此時尚有任何 DPMI 記憶體區塊存在，VMM 會自動釋放它們。因爲這層保障，DPMI clients 常常肆無忌憚地忽略釋放以函式 0501h 配置而來的記憶體。

在 Windows 中，另一個配置 extended 記憶體（譯註：1MB 以上）的方法是在 DPMI client 程式中發出 INT 21h, function 48h call。Windows 內建的 DOS extender 可以使用 extended 記憶體來滿足這項請求，而不把它下傳到 MS-DOS 手中。如你所預期，AX 傳回的是該記憶體區塊的一個 selector。你也可以使用 MS-DOS 函式 4Ah 和 49h 來重設記憶體大小以及釋放之。

由於 Windows 攔截 INT 21h, function 48h，把它解釋爲「對 extended 記憶體的索求」，所以如果有個輕鬆的辦法能夠配置 V86 記憶體，應該會有所幫助。函式 0100h（allocate）, 0101h（free）, 以及 0102h（resize）都能滿足這項需求。但是不要在 Windows 程式中使用這些函式，因爲它們總不會成功。失敗的原因是 Windows kernel 永遠會將 1st MB 的所有記憶體配置光光，以便接管其管理工作。Windows 程式應該使用 *GlobalDosAlloc* 在 System VM 中配置 V86 記憶體。

V86 記憶體的可見度 在一部 VM 中配置的 V86 記憶體，對另一部 VM 並不可見。舉個例子，如果一個 Windows 程式呼叫 *GlobalDosAlloc* 或是配置一塊 low memory（1MB 內的記憶體），唯一能夠輕鬆存取該區塊的程式是在 System VM 中跑的程式。相反的，如果一個 extended DOS 程式使用 DPMI 函式 0100h 配置一塊 low memory，通常只有相同 VM 中的程式才看得到它。對 VxDs 而言，永遠可能在任何 VM 的 V86 區域中看到任何記憶體 -- 只要使用你所獲得的 "high linear" 位址。此位址是將 VM control block 中的 *CB_high_Linear* 欄位加上你希望的 V86 位址。

使用中斷攔截（Interrupt Hooking）服務函式

DPMI 提供了一些類似 MS-DOS 的 *Get Vector* 和 *Set Vector*（INT 21h, function 35h 和 25）的服務，允許保護模式程式 "hook" 中斷。在保護模式中 hooking 一個中斷，意味著要改變 interrupt descriptor table（IDT）的對應項目（entry）。由於 host 無心地隱藏了 IDT，如果你要檢閱或改變它的話，需要作業系統的幫助。

下面的討論可以澄清「如何在一個 DPMI 環境中對中斷進行 hooking」的一些迷惑。

保護模式與 V86 模式的中斷

我們要考慮的第一個細節是，軟體中斷可能發生在保護模式或 V86 模式。保護模式的 INT 指令會流經 IDT 內一筆一筆的記錄（entries），所以你可以使用 DPMI 函式 0204h 和 0205h 加以檢閱並修改。V86 中斷向量存在於 0000h:0000h 的中斷向量表（事實上中斷向量係經由一個特殊的 V86 IDT，但一般而言 DPMI host 會迴轉並重新分派（redispatches）真實模式向量表格中的位址所指出的 V86 程式）。你可以使用 DPMI 函式 0200h 和 0201h 來檢閱和修改真實模式的中斷向量。

中斷反射（Interrupt Reflection）

一部機器擁有兩種模式（保護模式和 V86 模式）的中斷，令人感到困惑，因爲 DPMI host 會把一個保護模式軟體中斷反射（reflects）到 V86 模式中 -- 如果沒有任何人 hook 它的話。讓我們看看 INT 5Ch，這是通往 NetBIOS 的中斷介面。假設有一個保護模式程式使用這個中斷，向系統中的 NetBIOS 申請一個功能，但是系統未曾載入 VNETBIOS VxD。那麼，除非有些程式已經 "hooked" 保護模式的 5Ch 中斷，否則 DPMI host 會簡單地把 CPU 切換到 V86 模式然後 "dispatch" 真實模式的 NetBIOS handle。這個中斷使用 ES:BX 指向一個廣爲人知的控制區塊（一個網路控制區塊，network control block，簡稱 NCB），內含申請參數、一個或多個緩衝區位址、以及一個或許會有的 callback 函式位址。由於保護模式 client 程式內部是以 selectors 來思考，而真實模式的 NetBIOS 供應軟體內部是以 paragraphs 來思考，你當然可以想像，這項申請會失敗，因爲這個

NCB 沒有足夠的智慧到達真實模式處理常式手中。

攔截（Hooking）保護模式中斷

在 Windows 環境中，通常你會有 VNETBIOS 或某些 32 位元 NetBIOS 產品， 它們使用 *Set_PM_Int_Vector VxD* service 來攔截（hook）INT 5Ch。VNETBIOS 會將 NCB 中的所有指標爲轉化真實模式位址，將此中斷反射（reflect）至真實模式的「NetBIOS 供應軟體」中。你也可以靠自己的力量完成相同的事情：

```
mov  ax, 0205h           ; function 0205h: set PM int vector
mov  bl, 5Ch             ; BL = interrupt number
mov  cx, cs              ; CX:[E]DX -> new handler
mov  dx, offset int5c    ;   ..
int  31h                 ; go hook interrupt
```

（也可以先使用函式 0204h 取得前一個處理常式的位址，這麼一來你就可以在稍後恢復之。不過此例中請忘了這些細節吧）

當你的程式稍後發出 INT 5Ch，你就可以在自己的 service 常式（上例的 *int5c*）獲得控制權。然而控制權並非立刻從 INT 指令轉移到你的程式碼：

- Ring3 之中如果發生軟體中斷 50h~5Fh，會引起 general protection faults。因爲 IDT gate entries 的 Descriptor Privilege Level（DPL）是 0。Windows 會 "traps"（誘捕）這些中斷，因爲它們也被用於硬體中斷。VMM 設計者不再企圖令 VPICD 的中斷服務常式設法理解發生的是一個硬體中斷而不是一個 ring3 INT 指令，他們現在的選擇是去 "trap" 這個 INT 指令。注意，由於 DPL 所產生的 GP fault 被儲存在 gate entry 中，和 INT 指令的任何其他種 general trap 不一樣。一個控制程式有可能 "trap" 所有的 V86 模式 INT 指令，作法是設定 IOPL 爲 2 或更小，但是沒有什麼方法能夠 "trap" 所有的保護模式 INT 指令，除非設定所有的 gate DPLs。
- 軟體中斷 60h~FFh 也會引起 general protection faults，但是理由不一樣。爲了節省空間，Windows 只配置 180h bytes 給 IDTs，剛好足夠涵蓋中斷 00h~5Fh。更高的中斷比較罕見，相較之下，節省空間比即時處理更爲重要。

- 其他許多軟體中斷向量被導至 IDT 所指之處，也就是你以 DPMI 函式 0205h 所設定者。爲了確定你的程式即將直接獲得控制權而無來自 VMM 的任何干涉，你必須在 "hooking" 此中斷之後立刻在一個除錯器中檢閱 IDT。如果 gate 的 DPL 爲 3，並且指向你的服務常式，你就知道這個中斷將以最快的路徑直奔你所設定的碼。否則你就知道，VMM 將控制權交到你手上之前，會先玩弄此一中斷。

意外的是，當你 "hook" 某個中斷而它對應於正規的真實模式 IRQ（例如 08h~0Fh 或 70h~77h），你也就爲此硬體中斷建立了 VM 處理常式。如果系統缺乏一個 ring0 中斷服務常式，那麼 VPICD 就會呼叫你的程式以處理真正的硬體中斷。這樣的機制正是 COMM.DRV 如何在前一版 Windows 中運作的原因。

在 Windows 之中 "hooking" 保護模式中斷的另一方法是發出 MS-DOS 函式 35。內建的 DOS extender 會將此函式解釋爲一個申請(request)-- 申請對保護模式向量進行 hook 動作，並且不要反射至 V86 模式。

Processor Exceptions（處理器異常）

DPMI 的中斷管理之中，最令人迷惑的就是「正規中斷（regular interrupts）」和「處理器所產生的異常(processor generated exceptions)」之間的差異。它們其實很容易說明：general protection fault 是一種異常情況，當你的程式做了某些不良行爲或是做了某些非常好的事情（例如清除中斷或呼叫 NetBIOS）時，由 CPU 辨識出來。至於正規中斷則是在一個程式發出 INT 指令或是當一個硬體裝置中斷了 CPU 時發生。但是你如何解釋以下的差異：一個硬體中斷 IRQ5（它在 VM 之中是 INT 0Dh）、一個 general protection fault（也是 INT 0Dh）、以及軟體所產生的 INT 0D 指令？你當然可以檢視中斷控制器的 in-service 暫存器，看看儲存在 stack 中的是哪一個指令的位址，但這些探索式的動作並不總是能夠有效運作。

DPMI 嘗試幫助你分辨中斷，它的作法是讓你個別 "hook" 異常情況（exceptions），不和中斷混在一起。你可以使用 DPMI 函式 0202h 和 0203h 來取得或設定一個保護模式

異常向量(exception vector)。於是在前一節的假設問題中,GP faults 可以流往你的 0203h 異常情況處理常式,而 IRQ 5 和軟體 INT 0Ds 流往你的 0205h 中斷處理常式。(如果有人故意發出一個 INT 0Dh 指令,他們或許活該處於混亂之中,因爲你會認爲網路卡或 IRQ 5 上的其他硬體裝置被中斷了)

「DPMI 處理器異常」之處理常式和一般的中斷處理常式有很大的不同。當你 "hook" 一個一般中斷,你會獲得控制權,堆疊也會設定妥當,以便你做一個 IRET 或一個 IRETD 動作,回返中斷發生處。此外,暫存器值(除了 CS:[E]IP 和 [E]SP)會和中斷發生時相同。然而當你 "hook" 一個異常(exception),DPMI 呼叫你,用的是一個特別的 4-KB locked stack,而非中斷發生時正在使用的那個堆疊。此堆疊內含一些有關於目標物(一個 exception)的資訊,幫助你修復損毀,並回到中斷發生的那一點上(圖 **17-1**)。

Exception frame 之中,offset 00h 是回返位址,是你的異常處理常式應該回返的地點。其他欄位都是異常發生時的暫存器內容。(16 位元和 32 位元 DPMI clients 收到的資訊稍有不同,這反應出一個事實:指標在 32 位元程式中的寬度比較大。這裡我只說明及顯示 16 位元的情況)

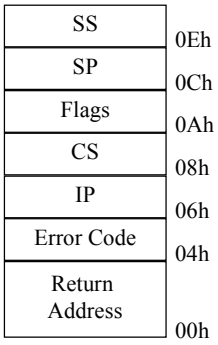


圖 17-1 進入一個 16 位元 DPMI 異常處理常式時,堆疊的佈局

你的處理常式必須完成兩件事情之一,以便回應異常情況:修補損毀(使程式繼續執行),或結束程式。幾乎我所遭遇的每一個案例之中,我都希望我的處理常式能夠使用原來的

堆疊（也就是程式被中斷時的環境）。例如某些時候我想列印一個錯誤訊息，然後執行一個 *longjmp* 以迴避某些問題，或是我想要喚起如 floating-point emulator 之類的東西。這都必須取得原來的暫存器值。結果，最終我幾乎總是把控制權移轉給一個看起來類似正規中斷服務常式的程式。

舉個例子，下面程式碼在 DPMI 環境下 "hooks" 「**Coprocessor Not Present** 異常情況」（中斷 07h）：

```
mov   ax, 0202h               ; function 0202h: get exception vector
mov   bl, 7                   ; BL = interrupt number
int   31h                     ; (return in CX:DX)
mov   word ptr org07, dx      ; save current int 07h handler address
mov   word ptr org07+2, cx    ;   ..

mov   ax, 0203h               ; function 0203h: set exception vector
mov   bl, 7                   ; BL = interrupt number
mov   cx, cs                  ; CX:DX -> exception handler
mov   dx, offset int07        ;   ..
int   31h                     ;   ..
```

第一組指令使用函式 0202h 獲得目前的 07h 異常向量。第二組指令使用函式 0203h 將 *int07* 建立爲一個異常處理常式。該函式只做一件事情：將控制權移轉到一個正規的中斷服務常式去：

```
static void _far _loadds int07
(unsigned int code,           // exception error code
unsigned int oldip,           // interrupt-old IP
unsigned int oldcs,           // interrupt-old CS
unsigned int oldflags,        // interrupt-old flags
unsigned int oldsp,           // nterrupt-old SP
unsigned int oldss)           // interrupt-old SS
{                             // int07
unsigned short _far *stack = MAKELP(oldss, oldsp -= 6);
stack[0] = oldip;
stack[1] = oldcs;
stack[2] = oldflags;

_asm
   {                          // redirect restart address
   mov   oldcs, cs
```

```
    mov   ax, offset emulate
    mov   oldip, ax
    }                           // redirect restart address
}                               // int07
```

這個小小程式的動作非常明白。參數的宣告反映出進入異常處理常式時的堆疊內容，如**圖 17-1** 所示。你需要知道的第一件事情就是，DPMI 會在我們回返之後，以參數位置中所發現的位址、旗標、堆疊，重新處理（redispatch）發出中斷的程式。換句話說，DPMI 會以傳址（by location）而非傳值（by value）的方式將參數傳遞給上述 *int07* 函式，這很像是在 C++ 中使用 reference 變數。*int07* 函式中首先計算一個遠程指標（變數 *stack*），使它指向中斷發生時程式堆疊位置的第 6 個 bytes 處，然後以 flags、code selector、instruction pointer 填充標準的 16 位元 interrupt frame。這些步驟模擬了中斷發生時的 CPU 正常動作。然後我們改變原本記錄在參數 *oldcs*:*oldip* 中的指令位址，使它指向一個名爲 *emulate* 的函式。

Coprocessor Emulation 備忘錄

DPMI 0.9 hosts 實作出 1.0 規格中的 0Exxh 函式，爲的是提供對 math coprocessor 的處理。例如，假設我打算在一部擁有真正算術協同處理器（math coprocessor）的機器上測試一個浮點運算模擬器（floating-point emulator），並假設這個模擬器靠著誘捕（trapping）所謂的 "not-present exceptions" 而生存。（16 位元 Windows 程式的模擬器則是以存在軟體中斷 30h~3Ah 取代之。編譯器、聯結器、以及程式載入器很心協力地在你的程式內放置中斷指令或正規 80x87 指令 -- 視你是否擁有一個算術協同處理器而定）。我必須設立斷存器 CR0 中的 EM-bit以使 "not-present exception" 被 enabled 起來，但是 CR0 處於保護模式，非 ring3 碰觸得到。

孔肥 CR0 的保護，是 DPMI 函式 0E01h 的一個主要目的。呼叫這個函式並搭配 BX 爲 2，可以使 emulation 旗爲 EM-bit 的設立而 enables 起來 -- 在我的 VM 處於控制情況之下（VCPD，也就是管理協同處理器的那個 VxD，發出一個 *Call_When_Task_Switched* call，於是便可以虛擬化此 EM-bit）。呼叫此函式並搭配 BX

為 0，則會因為 EM-bit 的清除而將 emulation 給 disables 掉。

真實模式程式當然不能夠呼叫 DPMI 函式，它們根本不在乎 CR0。Windows 以及其他大部份 V86 監視器注意到，CR0 的唯一改變就是 EM-bit，於是虛擬化此一改變。我曾經寫過一個 TSR 工具程式，用以修補 Pentium 的除法運算問題，這個工具程式利用「對指令的誘捕（trapping）」來模擬除法運算和其他一些有問題的指令。雖然它在 VCPI 記憶體管理器如 EMM386 和 QEMM 之下能夠順利運作，卻很要命地慢，這是因為「將 EM-bit 的變化虛擬化」的高成本所致。

上面所顯示的異常處理常式的效益就是，我們的模擬常式可以在一個 not-present exception 發生之後，像一個正常的中斷處理常式一樣取得控制權。在 DPMIDEMO.C 程式（位於書附光碟的 \CHAP17\DPMIDEMO 目錄）之中，一旦確定發生了 not-present exception，我便執行 FNOP 指令，下面這個簡單的處理常式於是獲得控制權：

```
typedef struct tagIFAME
   {                          // inerrupt frame
   unsigned short es, ds;
   unsigned short di, si, bp, sp, bx, dx, cx, ax;
   unsigned short ip, cs, flags;
   } IFRAME;                  // interrupt frame

static void (interrupt _far emulate)(IFRAME f)
   {                          // emulate
   _asm sti                   // enable virtual interrupts
   printf("Coprocessor not-present exception occurred"
          " at %4.4X:%4.4X\n", f.cs, f.ip);
   f.ip += 2;                 // skip "emulated" instruction
   }                          // emulate
```

DPMI 對於中斷和異常的不同處理態度，反應出一個事實，那就是 VxDs 使用兩個不同的 services 來 "hook" 它們：*Set_PM_Int_Vector* 用於中斷而 *Hook_PM_Fault* 用於異常。這項差異困擾了系統程式員，他們嘗試使用不同的 DPMI services（你可以從先前的例子中看到其所帶來的複雜度）。然而還有其他令人困惑的地方，因為某些異常（一般而言是 0~5 和 7）被暗示為中斷。你不可以像對待異常一樣地去 "hook" 它們。因此，

上例之中並沒有強烈必要去 hook "not-present exception"：我把 INT 7 視爲正規中斷般地 "hook" 之（但也因此我沒有一個用來對付異常的例子可以提供給你）。其他的處理器異常（6 以及 8~1F）會使 DPMI 結束你的程式 -- 如果你不處理它們的話。

呼叫真實模式

DPMI 函式 0300h（Simulate Real Mode Interrupt）對 Windows 程式有潛在的重要性，即使在 Windows 95 中亦然。這個函式允許你在 current VM 中喚起一個真實模式中斷處理常式，並供應基本機制，允許應用程式執行「Windows 內建之 DOS extender」所沒有處理的某些功能。爲了執行這些功能，你首先得準備一個真實模式的暫存器結構（圖 17-2），內含暫存器（特別是 segment 暫存器）值，那是你希望真實模式中斷處理常式使用的值。然後你應該把 ES:[E]DI 設爲該結構的位址，並將 BX 設爲想要發出的中斷號碼，將 CX 設爲 0，將 AX 設爲 0300h，然後發出 INT 31h，執行你所希望的真實模式中斷。你可以在真實模式暫存器結構中檢視執行結果。

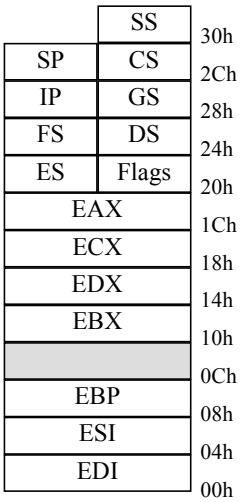


圖 17-2 DPMI 眞實模式暫存器結構

圖 17-2 所描述的真實模式暫存器結構，有一部份符合了 PUSHAD 指令將暫存器放進堆疊中的方式。遺漏的欄位（offset 0Ch）對應於 ESP 暫存器，那是 POPAD 指令在恢復暫存器值時會跳過的一個值。其他欄位的排列次序並沒有什麼特別理由。你應該注意一點，你可以遞交 32 位元 general 暫存器值到（或來自）使用此一結構的真實模式碼，即使雖然大部份 16 位元程式根本不知道 general 暫存器有較高的（擴充的）那一部份。

舉個例子。我需要在用以建造本節範例程式的那個 bootstrap loader 中釋放一塊 MS-DOS 記憶體；由於 DPMI 函式 0101h 只針對以函式 0100h 配置而來的記憶體才有作用，所以我必須發出一個 INT 21, function 49h。但由於 Windows 攔截了此一中斷，使它成爲一個 extended memory management call，所以我必須使用 DPMI 函式 0300h，在真實模式中執行這個函式。下面程式碼做的正是這些動作：

```
mov   bp, sp         ; save stack pointer
xor   eax, eax       ; ge DWORD of zeros
push  eax            ; build real-mode register structure: ss:sp
push  eax            ; cs:ip
push  eax            ; fs, gs
push  ax             ; ds
push  fixseg         ; es (segment to release)
pushf                ; flags
mov   ecx, 4900h     ; eax = DOS function 49h
push  ecx            ;   ..
push  eax            ; ecx
push  eax            ; edx
push  eax            ; ebx
push  eax            ; reserved (ESP slot in PUSHAD)
push  eax            ; ebp
push  eax            ; esi
push  eax            ; edi

mov   ax, ss         ; ES:DI -> real-mode register structure
mov   es, ax         ;   ..
mov   di, sp         ;   ..
mov   ax, 0300h      ; function 0300: simulate real-mode interrupt
mov   bx, 21h        ; BX = interrupt number (21)
xor   cx, cx         ; CX = number of bytes of stack to copy
int   31h            ; release memory

mov   sp, bp         ; restore stack past register structure
```

此段碼大部份用來建造位於堆疊中的一個真實模式暫存器結構（內含許多 0）。有一點我沒有提到，那就是如果你把真實模式暫存器結構中的 SS:SP 設爲 0，DPMI 便會找出一個堆疊並在其中執行真實模式程式。這樣的服務非常方便，可以節省你「配置並追蹤一塊真實模式記憶體，並且每次要反射（reflect）一個中斷時就填充 SS:SP」的麻煩。

其他的 DPMI 03xxh 服務函式　DPMI 的 03xxh 系列函式的威力甚至比我所描述的還要強大。除了使用 0300h 來模擬真實模式中斷，你還可以使用 0301h 來執行對於真實模式函式的遠程呼叫（far call），以及使用 0302h 來呼叫一個「結束時用的是 IRET 指令」的真實模式函式。這三項服務都允許你在 BX 中指定一個 flag 位元，使 A20 位址線爲 0，就像你在一般應用程式中希望把超過 1MB 的位址摺返（wrap around）爲 0 一樣。此外，這三項服務也允許你在 CX 中指定想要「從保護模式堆疊拷貝到真實模式堆疊」的 16 位元參數個數，這項性質使得「在保護模式中呼叫真實模式程式」變得十分簡單。

處理來自真實模式的呼叫

另一個威力強大的 DPMI 函式是 0303h（Allocate Real Mode Callback Address）。所謂「真實模式 callback」是一個 16:16 指標，真實模式程式可以呼叫它以到達你所寫的一個保護模式程式內。舉個例子，如果你想要爲某個真實模式中斷提供一個保護模式處理常式，你可以使用函式 0303h 產生出一個位址，然後使用 0201h 函式將此位址安裝爲真實模式中斷向量。要使用函式 0303h，首先必須配置一個永久的真實模式暫存器結構：

```
RMREGS rmregs;
```

呼叫函式 0303h 時，你必須提供這個結構的位址，但是一直到 DPMI 喚起「保護模式 callback 函式」，此結構才會被使用。這個結構的生命期必須與「真實模式 callback」的生命期相同。因此，通常我們會將這塊空間保留在 static 記憶體中。但是只要你有把握正確維護其生命，當然也可以把它配置在堆疊之中。接下來你必須以 DS:SI 指向你所寫的保護模式函式，以 ES:DI 指向真實模式暫存器結構，然後呼叫 0303h 函式。在我的範例

程式中，我想要 "trap" Ctrl-Break 訊號。如果是在真實模式，我必須 hook INT 1Bh。第一個步驟就是建立一個「真實模式 callback 函式」，我命名爲 *int1b*：

```
mov   ax, 0303h               ; function 0303h: allocate real-mode callback
mov   si, cs                  ; DS:SI -> protected-mode procedure to call
mov   ds, si                  ;   ..
mov   si, offset int1b
mov   di, ss                  ; ES:DI -> real-mode register structure
mov   es, di                  ;   ..
lea   di, rmregs              ;   ..
int   31h                     ; call DPMI, return in CX:DX
mov   word ptr callbak, dx    ; save callback address for 0304h
mov   word ptr calback+2, cx  ;  ..
```

函式 0303h 的傳回值放在 CS:DX 之中。由於這個遠程指標的較高半部是一個真實模式的 paragraph 號碼，把它放到一個 segment 暫存器是沒有用的，這就是爲什麼 DPMI 把它放在一個 general 暫存器中傳回給你的原因。

繼續這個例子，你可以 "hook" 真實模式 INT 1Bh 向量如下：

```
mov   ax, 0200h               ; function 0200h: get real-mode vector
mov   bl, 1Bh                 ; BL = interrupt number
int   31h                     ; call DPMI, return in CX:DX
mov   word ptr org1b, dx      ; save old 1B vector
mov   word ptr org1b+2, cx    ;   ..
mov   ax, 0201h               ; function 0201h: set real-mode vector
mov   bl, 1Bh                 ; BL = interrupt number
mov   dx, word ptr callback   ; CX:DX = new handler address
mov   cx, word ptr callback+2 ;  ..
int   31h                     ; (always succeeds)
```

函式 0200h 未取出目前的向量，我們應該在稍後將它恢復回來。函式 0201h 會安裝一個新的向量，其位址放在 CX:DX 之中。

現在你可以執行一個費時的程序，然後中斷之。例如：

```
static volatile int breakflag;
...
   breakflag = 0;
   puts("INT 1Bh hooked. Press Ctrl-Break to test it . . .");
   for (i = 0; i < 10000000 && !breakflag; ++i)
      ;
   if (breakflag)
      printf("Loop terminated due to Ctrl-Break"
             " with counter = %ld\n", i);
   else
      printf("Loop terminated by itself after %ld iterations\n", i);
...
```

當你執行這個大型迴路並按下 Ctrl-Break，Windows 會把「鍵盤中斷」傳遞給 BIOS。後者辨識出此動作爲 "break" 之意，於是在真實模式中發出 INT 1Bh。由於你已經 "hooked" 了真實模式中斷向量，控制權會被遞交到「真實模式 callback」位址上，也就到達你的保護模式 *int1b* 常式中。那個常式會在一個 4-KB locked DPMI 堆疊上收到控制權；離開前應該適當改變真實模式的暫存器，並以 IRET 指令離開。例如：

```
static void (interrupt _far int1b)(IFRAME f)
{                              // int1b
unsigned short _far *stack;    // pointer to real-mode stack
RMEGS _far *rp;                // pointer to real-mode registers

breakflag = 1;

stack = MAKELP(f.ds, f.si);
rp = MAKELP(f.es, f.di);
rp->ip = stack[0];
rp->cs = stack[1];
rp->flags = stack[2];
rp->sp += 6;
}    // int1b
```

此函式所執行的唯一實在動作是設定一個全域變數（*breakflag*）用以結束主程式中的迴圈。其餘程式碼都是爲了行政管理而加上去，用以適當地解開（unwind）真實模式 INT 1Bh call。我們首先產生一個遠程指標（變數 *stack*）指向堆疊，那是真實模式程式執行的地方。DPMI 以 DS:SI 指出這個堆疊的保護模式版本。Microsoft C 的 interrupt function

prolog 會儲存 DS:SI，本例告訴你如何取出其原始內容。DPMI 也以 ES:DI 指向一個真實模式暫存器結構的位址，這個結構一開始內含的是 INT 1Bh 發生時的暫存器內容，而在我們回返到 DPMI 之後，真實模式程式將再次執行起來，此時結構內容是我們離開時的暫存器值。本例之中，我把 CS, IP 和 Flags 暫存器從真實模式的堆疊中 pop 出來，模擬一個真實模式 IRET 指令。

當我第一次需要在保護模式程式中處理 Ctrl-Break 時，我遇到的瓶頸是不知道如何跳出所有這些 DPMI 函式所構成的鐵環。唔，這就是進步的代價。

其他的 DPMI 函式

結束 DPMI 的討論之前，我還要談談其他一些雜項函式。第一個要談的是 0400h（Get Version）的突兀行爲。在 Windows 所有 Enhanced-mode 版本之中，此函式會傳回 005Ah，這是 0.90 的二進位整數。至於在 Windows Standard-mode 版本中傳回的是 0090h，是 0.90 的「二進位碼十進位表示法」。此值是錯誤的，但甚至 Windows NT 3.1 都還保留著這個錯誤。NT 3.51 就做了改善，傳回的是 005Ah。所以如果你曾經看過有些軟體聲稱使用 DPMI 0.144 版，你可以猜想它大概是在 Windows 3.0 或 3.1 的 standard-mode 上執行。

DPMI 0.9 的各函式之中，最沒有用的大概是 0604h（Get Page Size）了。Windows 使用 4096-byte pages，並且大概會一直這麼用下去，所以這個 service 總是傳回 4096。事實上，DPMI 標準委員會打算把它改名爲 "Function Which Returns 4096"。

相對於 0604h 的無用，0800h（Map Physical Address）卻是價值非凡。這個函式從 BX:CX 取得一個實際位址，並從 SI:DI 中取得 bytes 個數，傳回的是映射的線性位址（記錄於 BX:CX 之中）。對於那些「需要處理硬體裝置之實際記憶體」的 ring3 驅動程式而言，函式 0800h 價值非凡。一如你所猜測，這個函式其實是 *_MapPhysToLinear* service（我們曾經在討論 "connection with I/O programming" 時談過）的一層薄薄包裝而已。

最後我要討論的是奇怪的 *09xxh* 函式，它們可以幫助你管理你的 VMs 的虛擬中斷狀態（virtual interrupt state）。真實模式碼，特別是真實模式中斷處理常式，通常內含像這樣的動作序列：

```
pushf               ; save flags
cli                 ; disable interrups
...
popf                ; restore flags, including interrupt state
```

PUSHF 指令會儲存 Flags 暫存器（其中包括 **Interrupt Enable** flag）於堆疊內。CLI 指令會將處理器 disables，使某些敏感工作可以不被打擾地完成。最後的 POPF 指令用來恢復原先的 Flags 暫存器值，於是又恢復了原先的中斷旗標。一般而言，你應該使用像這樣的程式碼，而不要假設在離開一段敏感程式碼時只執行一個 STI 指令就好了。偶而我會在 VxDs 之中使用類似的程式碼（但改用 PUSHFD 和 POPFD 以便儲存和恢復 EFlags 的全部 32 個位元），理由相同。

但如果你在保護模式 ring3 程式中執行上述三個指令，會發生一件意想不到的事情。當你執行 PUSHF，Intel 晶片會完美地把 Flags 暫存器儲存起來，這些被儲存的 flags 用以表示真實機器的 enable state。接下來的 CLI 指令會引起一個 general protection fault，因爲當 Windows 正在執行保護模式程式時，會把 IOPL（I/O Privilege Level，Flags 暫存器中的一個 flag）設爲 0（回憶一下，V86 應用程式的 IOPL 通常是 3，以避免效率上的不良報應）。因此，Windows 會 "trap" CLI 指令並清除 current VM 的虛擬中斷旗標，留下真實中斷旗標（它當時有可能處於 enabled 狀態）。雖然實際機器可以處理這段敏感程式碼執行時期所發生的中斷，VMM 卻不會將任何中斷反射（reflect）給適當的 VM，因爲虛擬中斷旗標是 off。因此 VMM 必須特別對待 CLI 指令中暗含的真意。

但是 POPF 指令並不會實現我們所期望的事情。它並不會引起一個 fault，也不會引起 中斷旗標的任何改變。所以 CPU 一如以往地繼續保持 enabled，而 VM 繼續保持 disabled。這樣的行爲不能說是錯誤，但 Intel 面對 Pentium 機器時重新做了思考，提供對於 virtual enable state 一個比較穩健強固（robust）的維護方法。

DPMI *09xxh* 系列函式的目的正是爲了避免虛擬中斷旗標帶來的問題。現在，請以下面的 DPMI calls 取代稍早的那三個指令：

```
mov  ax, 0900h       ; get, then disable virtual interrupt flag
int  31h             ; returns 0900h or 0901h in AX
push ax              ; save result
...
pop  ax              ; restore 0900 or 0901h
int  31h             ; restore virtual interrupt flag
```

第一個動作 0900h call 會把虛擬中斷給 disable 掉，並傳回 0900h（如果虛擬中斷原本 disabled 的話）或 0901h （如果虛擬中斷原本 enabled 的話）。我們把這個傳回值儲存在堆疊之中，直到處理完這一段敏感程式碼爲止。然後我們設定 AX 暫存器爲我們所儲存的值，並再次呼叫 DPMI。如果虛擬中斷最初是 disabled，我們執行的就是 0900h 函式，繼續保持 disabled。如果虛擬中斷最初是 enabled，我們執行的就是 0901h 函式，繼續保持 enabled。

附 錄

Plug and Play Device Identifiers

這份附錄呈現的是 DEVIDS.TXT 檔案中的資訊（包括其中的錯誤和漏失），這個檔案是 Windows generic device IDs 和 Plug and Play BIOS device 型別代碼的終極來源。本附錄第一部份列出 generic device IDs，第二部份列出 device 型別代碼。你可以從 CompuServe PLUGPLAY 論壇下載 DEVIDS.ZIP 檔，獲得這份文件的最新版本。此份文件也可從 Microsoft 的網站獲得，請使用以下的 URL：

http://www.microsoft.com/window/download/devids.txt.

第一部份：Windows Generic Device IDs

許多 devices，像是 interrupt controller 和 keyboard controller，沒有標準的 EISA ID。還有一些 generic device 類別（像是 VGA display adapters）也沒有 EISA ID，因爲它們是一種 devices 類別，而非某家廠商所製作的單一 devices。目前還沒有另一組 IDs 被用來識別各種 buses。爲了讓程式能夠辨識那些沒有 EISA IDs 的各種 devices，並定義相容的 devices, Microsoft 保留了一個 EISA 前置詞 PNP。這些 IDs 定義如下。

System Devices - PNP0XXX

Device ID	說明	System Device - PNP0xxx
Interrupt Controllers		
PNP0000	AT interrupt controller	
PNP0001	EISA interrupt controller	
PNP0002	MCA interrupt controller	
PNP0003	Advanced Program Interrupt Controller (APIC)	
PNP0004	Cyrix SLiC MP interrupt controller	
Timers		
PNP0100	AT timer	
PNP0101	EISA timer	
PNP0102	MCA timer	
DMA Controllers		
PNP0200	AT DMA controller	
PNP0201	EISA DMA controller	
PNP0202	MCA DMA controller	
Keyboards		
PNP0300	IBM PC/XT keyboard controller (83-key)	
PNP0301	IBM PC/AT keyboard controller (86-key)	
PNP0302	IBM PC/XT keyboard controller (84-key)	
PNP0303	IBM enhanced keyboard (101/102-key, PS/2 mouse support)	
PNP0304	Olivetti keyboard (83-key)	
PNP0305	Olivetti keyboard (102-key)	
PNP0306	Olivetti keyboard (86-key)	
PNP0307	Microsoft Windows keyboard	
PNP0308	General Input Device Emulation Interface (GIDEI) legacy	
PNP0309	Olivetti keyboard (A101/102-key)	
PNP030A	AT&T 302 keyboard	
Parallel Devices		
PNP0400	Standard LPT printer port	
PNP0401	ECP printer port	
Serial Devices		

Device ID	說明 System Device - PNP0xxx
PNP0500	Standard PC COM port
PNP0501	16550A-compatible COM port
Disk Controllers	
PNP0600	Generic ESDI/IDE/ATA-compatible hard disk controller
PNP0601	Plus Hardcard II
PNP0602	Plus Hardcard IIXL/EZ
PNP0603	HP Omnibook IDE controller
PNP0700	PC standard floppy disk controller
PNP0701	HP Omnibook floppy disk controller
ID Included for Compatibility with Early Device ID List	
PNP0802	Microsoft Sound System-compatible device (obsolete; use PNPB0*xx* instead)
Display Adapters	
PNP0900	VGA-compatible
PNP0901	Video Seven VRAM/VRAM II/1024i
PNP0902	8514/A-compatible
PNP0903	Trident VGA
PNP0904	Cirrus Logic laptop VGA
PNP0905	Cirrus Logic VGA
PNP0906	Tseng Labs ET4000
PNP0907	Western Digital VGA
PNP0908	Western Digital laptop VGA
PNP0909	S3 Inc. 911/924
PNP090A	ATI Ultra Pro/Plus (Mach 32)
PNP090B	ATI Ultra (Mach 8)
PNP090C	XGA-compatible
PNP090D	ATI VGA Wonder
PNP090E	Weitek P9000 graphics adapter
PNP090F	Oak Technology VGA
PNP0910	Compaq Qvision
PNP0911	XGA/2
PNP0912	Tseng Labs W32/W32i/W32p
PNP0913	S3 Inc. 801/928/964

Device ID	說明 System Device - PNP0xxx
PNP0914	Cirrus Logic 5429/5434 (memory mapped)
PNP0915	Compaq Advanced VGA (AVGA)
PNP0916	ATI Ultra Pro Turbo (Mach 64)
PNP0917	Reserved by Microsoft
PNP0930	Chips & Technologies Super VGA
PNP0931	Chips & Technologies Accelerator
PNP0940	NCR 77c22e Super VGA
PNP0941	NCR 77c32blt
PNP09FF	Plug and Play monitors (VESA DDC)
Peripheral Buses	
PNP0A00	ISA bus
PNP0A01	EISA bus
PNP0A02	MCA bus
PNP0A03	PCI bus
PNP0A04	VESA/VL bus
Real Time Clock	**BIOS, and System Board Devices**
PNP0800	AT-style speaker sound
PNP0B00	AT real time clock
PNP0C00	Plug and Play BIOS (only created by the root enumerator)
PNP0C01	System board
PNP0C02	ID for reserving resources required by Plug and Play motherboard registers (not specific to a particular device)
PNP0C03	Plug and Play BIOS event notification interrupt
PNP0C04	Math coprocessor
PNP0C05	APM BIOS (version-independent)
PNP0C06	Reserved for identification of early PnPBIOS implementation
PNP0C07	Reserved for identification of early PnPBIOS implementation
PCMCIA Controller Chipsets	
PNP0E00	Intel 82365-compatible PCMCIA controller
PNP0E01	Cirrus Logic CL-PD6720 PCMCIA controller
PNP0E02	VLSI VL82C146 PCMCIA controller
Mice	

Device ID	說明	System Device - PNP0xxx
PNP0F00	Microsoft bus mouse	
PNP0F01	Microsoft serial mouse	
PNP0F02	Microsoft InPort mouse	
PNP0F03	Microsoft PS/2 mouse	
PNP0F04	MouseSystems mouse	
PNP0F05	MouseSystems 3-Button mouse (COM2)	
PNP0F06	Genius mouse (COM1)	
PNP0F07	Genius mouse (COM2)	
PNP0F08	Logitech serial mouse	
PNP0F09	Microsoft BallPoint serial mouse	
PNP0F0A	Microsoft Plug and Play mouse	
PNP0F0B	Microsoft Plug and Play BallPoint mouse	
PNP0F0C	Microsoft-compatible serial mouse	
PNP0F0D	Microsoft-compatible InPort mouse	
PNP0F0E	Microsoft-compatible PS/2 mouse	
PNP0F0F	Microsoft-compatible BallPoint serial mouse	
PNP0F10	Texas Instruments Quick Port mouse	
PNP0F11	Microsoft-compatible bus mouse	
PNP0F12	Logitech PS/2 mouse	
PNP0F13	PS/2 port for PS/2 mice	
PNP0F14	Microsoft Kids mouse	
PNP0F15	Logitech bus mouse	
PNP0F16	Logitech SWIFT device	
PNP0F17	Logitech-compatible serial mouse	
PNP0F18	Logitech-compatible bus mouse	
PNP0F19	Logitech-compatible PS/2 mouse	
PNP0F1A	Logitech-compatible SWIFT device	
PNP0F1B	HP Omnibook mouse	
PNP0F1C	Compaq LTE trackball PS/2 mouse	
PNP0F1D	Compaq LTE trackball serial mouse	
PNP0F1E	Microsoft Kids trackball mouse	
PNP0F1F	Reserved by Microsoft Input Device Group	

Device ID	說明	System Device - PNP0xxx
PNP0F20	Reserved by Microsoft Input Device Group	
PNP0F21	Reserved by Microsoft Input Device Group	
PNP0F22	Reserved by Microsoft Input Device Group	
PNP0FFF	Reserved by Microsoft Systems	

Network Adapters - PNP8xxx

Device ID	說明	Network Adapters - PNP8xxx
PNP8001	Novell/Anthem NE3200	
PNP8004	Compaq NE3200	
PNP8006	Intel EtherExpress/32	
PNP8008	HP Ethertwist EISA LAN Adapter/32 (HP27248A)	
PNP8065	Ungermann-Bass NIUps or NIUps/EOTP	
PNP8072	DEC (DE211) Etherworks MC/TP	
PNP8073	DEC (DE212) Etherworks MC/TP_BNC	
PNP8078	DCA 10-MB MCA	
PNP8074	HP MC LAN Adapter/16 TP (PC27246)	
PNP80c9	IBM Token Ring	
PNP80ca	IBM Token Ring II	
PNP80cb	IBM Token Ring II/Short	
PNP80cc	IBM Token Ring 4/16 MB	
PNP80d3	Novell/Anthem NE1000	
PNP80d4	Novell/Anthem NE2000	
PNP80d5	NE1000-compatible	
PNP80d6	NE2000-compatible	
PNP80d7	Novell/Anthem NE1500T	
PNP80d8	Novell/Anthem NE2100	
PNP80dd	SMC ArcNetPC	
PNP80de	SMC ArcNet PC100, PC200	
PNP80df	SMC ArcNet PC110, PC210, PC250	
PNP80e0	SMC ArcNet PC130/E	
PNP80e1	SMC ArcNet PC120, PC220, PC260	

Device ID	說明	Network Adapters - PNP8xxx
PNP80e2	SMC ArcNet PC270/E	
PNP80e5	SMC ArcNet PC600W, PC650W	
PNP80e7	DEC DEPCA	
PNP80e8	DEC (DE100) Etherworks LC	
PNP80e9	DEC (DE200) Etherworks Turbo	
PNP80ea	DEC (DE101) Etherworks LC/TP	
PNP80eb	DEC (DE201) Etherworks Turbo/TP	
PNP80ec	DEC (DE202) Etherworks Turbo/TP_BNC	
PNP80ed	DEC (DE102) Etherworks LC/TP_BNC	
PNP80ee	DEC EE101 (built-in)	
PNP80ef	DECpc 433 WS (built-in)	
PNP80f1	3Com EtherLink Plus	
PNP80f3	3Com EtherLink II or IITP (8-bit or 16-bit)	
PNP80f4	3Com TokenLink	
PNP80f6	3Com EtherLink 16	
PNP80f7	3Com EtherLink III	
PNP80fb	Thomas Conrad TC6045	
PNP80fc	Thomas Conrad TC6042	
PNP80fd	Thomas Conrad TC6142	
PNP80fe	Thomas Conrad TC6145	
PNP80ff	Thomas Conrad TC6242	
PNP8100	Thomas Conrad TC6245	
PNP8105	DCA 10 MB	
PNP8106	DCA 10 MB fiber optic	
PNP8107	DCA 10 MB twisted pair	
PNP8113	Racal NI6510	
PNP811C	Ungermann-Bass NIUpc	
PNP8120	Ungermann-Bass NIUpc/EOTP	
PNP8123	SMC StarCard Plus (WD/8003S)	
PNP8124	SMC StarCard Plus with on-board hub (WD/8003SH)	
PNP8125	SMC EtherCard Plus (WD/8003E)	
PNP8126	SMC EtherCard Plus with boot ROM socket (WD/8003EBT)	

Device ID	說明 Network Adapters - PNP8xxx
PNP8127	SMC EtherCard Plus with boot ROM socket (WD/8003EB)
PNP8128	SMC EtherCard Plus TP (WD/8003WT)
PNP812a	SMC EtherCard Plus 16 with boot ROM socket (WD/8013EBT)
PNP812d	Intel EtherExpress 16 or 16TP
PNP812f	Intel TokenExpress 16/4
PNP8130	Intel TokenExpress MCA 16/4
PNP8132	Intel EtherExpress 16 (MCA)
PNP8137	Artisoft AE-1
PNP8138	Artisoft AE-2 or AE-3
PNP8141	Amplicard AC 210/XT
PNP8142	Amplicard AC 210/AT
PNP814b	Everex SpeedLink /PC16 (EV2027)
PNP8155	HP PC LAN Adapter/8 TP (HP27245)
PNP8156	HP PC LAN Adapter/16 TP (HP27247A)
PNP8157	HP PC LAN Adapter/8 TL (HP27250)
PNP8158	HP PC LAN Adapter/16 TP Plus (HP27247B)
PNP8159	HP PC LAN Adapter/16 TL Plus (HP27252)
PNP815f	National Semiconductor Ethernode *16AT
PNP8160	National Semiconductor AT/LANTIC Ethernode 16-AT3
PNP816a	NCR Token-Ring 4 MB ISA
PNP816d	NCR Token-Ring 16/4 MB ISA
PNP8191	Olicom 16/4 Token-Ring adapter
PNP81c3	SMC EtherCard PLUS Elite (WD/8003EP)
PNP81c4	SMC EtherCard PLUS 10T (WD/8003W)
PNP81c5	SMC EtherCard PLUS Elite 16 (WD/8013EP)
PNP81c6	SMC EtherCard PLUS Elite 16T (WD/8013W)
PNP81c7	SMC EtherCard PLUS Elite 16 Combo (WD/8013EW or 8013EWC)
PNP81c8	SMC EtherElite Ultra 16
PNP81e4	Pure Data PDI9025-32 (Token Ring)
PNP81e6	Pure Data PDI508+ (ArcNet)
PNP81e7	Pure Data PDI516+ (ArcNet)
PNP81eb	Proteon Token Ring (P1390)

Device ID	說明	Network Adapters - PNP8xxx
PNP81ec	Proteon Token Ring (P1392)	
PNP81ed	Proteon ISA Token Ring (1340)	
PNP81ee	Proteon ISA Token Ring (1342)	
PNP81ef	Proteon ISA Token Ring (1346)	
PNP81f0	Proteon ISA Token Ring (1347)	
PNP81ff	Cabletron E2000 Series DNI	
PNP8200	Cabletron E2100 Series DNI	
PNP8209	Zenith Data Systems Z-Note	
PNP820a	Zenith Data Systems NE2000-compatible	
PNP8213	Xircom Pocket Ethernet II	
PNP8214	Xircom Pocket Ethernet I	
PNP821d	RadiSys EXM-10	
PNP8227	SMC 3000 Series	
PNP8231	Advanced Micro Devices AM2100/AM1500T	
PNP8263	Tulip NCC-16	
PNP8277	EXOS 105	
PNP828A	Intel '595-based Ethernet	
PNP828B	TI2000-style Token Ring	
PNP828C	AMD PCNet Family cards	
PNP828D	AMD PCNet32 (VL version)	
PNP82bd	IBM PCMCIA-NIC	
PNP8321	DEC Ethernet (all types)	
PNP8323	SMC EtherCard (all types except 8013/A)	
PNP8324	ArcNet-compatible	
PNP8326	Thomas Conrad (all ArcNet types)	
PNP8327	IBM Token Ring (all types)	
PNP8385	Remote network access (RNA) driver	
PNP8387	RNA point-to-point protocol driver	

SCSI and Proprietary CD Adapters - PNPAxxx

Device ID	說明 SCSI and Proprietary CD Adapters - PNPAxxx
PNPA000	Adaptec 154x-compatible SCSI controller
PNPA001	Adaptec 174x-compatible SCSI controller
PNPA002	Future Domain 16-700-compatible controller
PNPA003	Panasonic proprietary CD-ROM adapter (SBPro/SB16)
PNPA01B	Trantor 128 SCSI controller
PNPA01D	Trantor T160 SCSI controller
PNPA01E	Trantor T338 parallel SCSI controller
PNPA01F	Trantor T348 parallel SCSI controller
PNPA020	Trantor Media Vision SCSI controller
PNPA022	Always IN-2000 SCSI controller
PNPA02B	Sony proprietary CD-ROM controller
PNPA02D	Trantor T13b 8-bit SCSI controller
PNPA02F	Trantor T358 parallel SCSI controller
PNPA030	Mitsumi LU-005 single-speed CD-ROM controller + drive
PNPA031	Mitsumi FX-001 single-speed CD-ROM controller + drive
PNPA032	Mitsumi FX-001 double-speed CD-ROM controller + drive

Sound/Video Capture and Multimedia Adapters - PNPBxxx

Device ID	說明　Sound/Video Capture and Multimedia Adapters - PNPBxxx
PNPB000	Sound Blaster 1.5-compatible sound device
PNPB001	Sound Blaster 2.0-compatible sound device
PNPB002	Sound Blaster Pro-compatible sound device
PNPB003	Sound Blaster 16-compatible sound device
PNPB004	Thunderboard-compatible sound device
PNPB005	Adlib-compatible FM synthesizer device
PNPB006	MPU401-compatible
PNPB007	Microsoft Windows Sound System-compatible sound device
PNPB008	Compaq Business audio device
PNPB009	Plug and Play Microsoft Windows Sound System device
PNPB00A	Media Vision Pro Audio Spectrum device (Trantor SCSI enabled, Thunder Chip disabled)
PNPB00B	Media Vision Pro Audio 3D device
PNPB00C	MusicQuest MQX-32M
PNPB00D	Media Vision Pro Audio Spectrum Basic device (no Trantor SCSI, Thunder Chip enabled)
PNPB00E	Media Vision Pro Audio Spectrum device (Trantor SCSI enabled, Thunder Chip enabled)
PNPB00F	Media Vision Jazz-16 chipset (OEM versions)
PNPB010	Auravision VxP500 chipset - Orchid Videola
PNPB018	Media Vision Pro Audio Spectrum 8-bit device
PNPB019	Media Vision Pro Audio Spectrum Basic device (no Trantor SCSI, Thunder Chip disabled)
PNPB020	Yamaha OPL3-compatible FM synthesizer device
PNPB02F	Joystick/game port

Modems - PNPCxxx ~ PNPDxxx

Device ID	說明
PNPC000	Compaq 14400 modem (TBD)
PNPC001	Compaq 2400/9600 modem (TBD)

第二部份：Device Type Codes

Base Type	Subtype	Interface Type
0: Reserved		
1: Mass storage device	0: SCSI controller	
	1: IDE controller (standard ATA-compatible)	0: Generic IDE
	2: Floppy controller (standard 765-compatible)	0: Generic floppy
	3: IPI controller	0: General IPI
	80h: Other mass storage controller	
2: Network interface controller	0: Ethernet	0: General Ethernet
	1: Token Ring controller	0: General Token Ring
	2: FDDI controller	0: General FDDI
	80h: Other network interface controller	
3: Display controller	0: VGA controller (standard VGA-compatible)	0: Generic VGA-compatible
		1: VESA SVGA-compatible controller
	1: XGA-compatible controller	0: General XGA-compatible controller
	80h: other display controller	
4: Multimedia controller	0: Video controller	0: General video controller
	1: Audio controller	0: General audio controller
	80h: Other multimedia controller	
5: Memory	0: RAM	0: General RAM
	1: FLASH memory	0: General FLASH memory
	80h: Other memory device	
6: Bridge controller	0: Host processor bridge	0: General host processor bridge
	1: ISA bridge	0: General ISA bridge
	2: EISA bridge	0: General EISA bridge
	3: Micro Channel bridge	0: General Micro Channel bridge

Base Type	Subtype	Interface Type
	4: PCI bridge	0: General PCI bridge
	5: PCMCIA bridge	0: General PCMCIA bridge
	80h: Other bridge device	
7: Communications device	0: RS-232 device (XT-compatible COM)	0: Generic XT-compatible
		1: 16450-compatible
		2: 16450-compatible
	1: AT-compatible parallel port	0: Generic AT parallel port
		1: Model-30 bidirectional port
		2: ECP 1.x-compliant port
	80h: Other communications device	
8: System Peripherals	0: Programmable Interrupt Controller (8259-compatible)	0: Generic 8259 PIC
		1: ISA PIC (8259-compatible)
		2: EISA PIC (8259-compatible)
	1: DMA controller (8237-compatible)	0: Generic DMA controller
		1: ISA DMA controller
		2: EISA DMA controller
	2: System timer (8254-compatible)	0: Generic system timer
		1: ISA system timer
		2: EISA system timers (two timers)
	3: Real time clock	0: Generic RTC controller
		1: ISA RTC controller
	80h: Other system peripheral	
9: Input device	0: Keyboard controller	0: Not applicable
	1: Digitizer (Pen)	0: Not applicable
	2: Mouse controller	0: Not applicable
	80h: Other input controller	
0Ah: Docking station	0: Generic docking station	0: Not applicable
	80h: Other type of docking station	
0Bh: CPU type	0: 386-based processor	0: Not applicable
	1: 486-based processor	0: Not applicable

Base Type	Subtype	Interface Type
	2: Pentium-based processor	0: Not applicable