Advanced Windows®
by Jeffrey Richter
976 pages, one CD-ROM, $44.95
ISBN: 1-55615-677-4
Pub. Date: March 3, 1995

FOR IMMEDIATE RELEASE
Contact: Cheri Chapman
206-936-5302
206-936-7329 fax
Email: cherico@microsoft.com

Jeffrey Richter's
Advanced Windows®
The Developer's Guide to the Win32® API for Windows NT™ 3.5 and Windows 95

February 7, 1995 — REDMOND, WA — Microsoft Press announced today the publication of ADVANCED WINDOWS®, by seasoned developer, consultant, and critically acclaimed author Jeffrey Richter.

ADVANCED WINDOWS is the first full discussion of the Win32® API for Windows NT™ version 3.5 and Windows 95. It offers expert advice, sample code, and tested techniques for ISVs and corporate developers building new 32-bit applications from scratch or porting existing applications to a Win32 operating system.

Richter examines the core areas of the Windows NT 3.5 and Windows 95 operating systems in detail and explains what the move to Win32 really means in terms of:

- Managing processes and threads, including thread-local storage and thread synchronization
- Exploring virtual memory and using virtual memory in applications
- Writing DLLs—and techniques for using them most effectively
- Sharing code and data among applications using memory-mapped files and sophisticated DLL-injection techniques
- Developing software for international markets using Unicode
- Writing robust, error-free applications using structured exception handling

Along the way, Richter points out how to both unleash the power and avoid the pitfalls of the complex Win32 API. He carefully details Win32 functions and programming methods specific to either Windows 95 or Windows NT.

An appendix explains how to use "message crackers" to help read, write, and maintain source code. A comprehensive table of contents and a thorough, cross-referenced index help readers quickly locate information about any topic in the book.

Source code and executable binaries for all of the more than 25 sample programs presented in ADVANCED WINDOWS are included on one companion CD-ROM. Written for the Intel®, MIPS®, and Digital Alpha AXP™ microprocessors, these programs show true 32-bit programming techniques in action. Visual C++™ development system version 2.0 and either Microsoft® Windows NT (for any CPU platform) or Windows 95 are required.

Jeffrey Richter is the author of *Advanced Windows NT* and *Windows 3.1: A Developer's Guide* and is a contributing editor of *Microsoft Systems Journal*. A full-time software design engineer, Richter is also a consultant, a trainer, and a frequent speaker at developer conferences around the world.

Microsoft Press is the independent book division of Microsoft Corporation and the leading publisher of quality computer books about Microsoft products. More than 10 million users at all skill levels rely on a complete line of Microsoft Press® books to make learning and using software easier. Titles ranging from streamlined tutorials for first-time computer users to technical references for professional programmers are distributed to book and software retailers worldwide. Consumers in the United States can also order directly from the publisher at 1-800-MSPRESS and through the CompuServe® Electronic Mall (GO MSP).

Founded in 1975, Microsoft (NASDAQ "MSFT") is the worldwide leader in software for personal computers. The company offers a wide range of products and services for business and personal use, each designed with the mission of making it easier and more enjoyable for people to take advantage of the full power of personal computing every day.

C H A P T E R F I V E

EXPLORING VIRTUAL MEMORY

In the last chapter, we discussed how the system manages virtual memory, how each process receives its very own private address space, and what a process's address space looks like. In this chapter, we move away from the abstract and examine some of the Win32 functions that give us information about the system's memory management and about the virtual address space in a process.

System Information

To understand how Win32 uses virtual memory, you need to know how the current Win32 implementation works. The *GetSystemInfo* function retrieves information (including virtual memory information) about the current Win32 implementation:

```
VOID GetSystemInfo (LPSYSTEM_INFO lpSystemInfo);
```

You must pass the address of a SYSTEM_INFO structure to this function. The function will initialize the structure's members and return. Here is what the SYSTEM_INFO data structure looks like:

```
typedef struct _SYSTEM_INFO {
  DWORD   dwOemId;
  DWORD   dwPageSize;
  LPVOID  lpMinimumApplicationAddress;
  LPVOID  lpMaximumApplicationAddress;
  DWORD   dwActiveProcessorMask;
  DWORD   dwNumberOfProcessors;
  DWORD   dwProcessorType;
  DWORD   dwAllocationGranularity;
  DWORD   dwReserved;
} SYSTEM_INFO;
```

When the system boots, it determines what the values of these members should be; for a given system the values will always be the same. *GetSystemInfo* exists so that an application can query these values at run time. Of all the members in the structure, only four of them have anything to do with memory. These four members are explained in the table below:

Member Name	Description
dwPageSize	Shows the size of a memory page. On *x*86, MIPS, and PowerPC CPUs, this value is 4 KB. On Alpha CPUs, this value is 8 KB.

lpMinimumApplicationAddress	Gives the minimum memory address of every process's usable address space. On Windows 95, this value is 4,194,304, or 0x00400000, because the bottom 4 MB of every process's address space is inaccessible. On Windows NT, this value is 65,536, or 0x00010000, because the first 64 KB of every process's address space is reserved.
lpMaximumApplicationAddress	Gives the maximum memory address of every process's usable private address space. On Windows 95, this address is 2,147,483,647, or 0x7FFFFFFF, because the shared memory-mapped file region and the shared operating system code are contained in the top 2-GB partition. On Windows NT, this address is 2,147,418,111, or 0x7FFEFFFF, because unusable address space begins just 64 KB below the 2-GB line and extends to the end of the process's address space.
dwAllocationGranularity	Shows the granularity of a reserved region of address space. As of this writing, this value is 65,536 because all implementations of Win32 reserve address space on even 64-KB boundaries.

The System Information Sample Application

The SysInfo application (SYSINFO.EXE), listed in Figure 5-1 beginning on page 130, is a very simple program that calls *GetSystemInfo* and displays the information returned in the SYSTEM_INFO structure. The source code files, resource files, and make file for the application are in the SYSINFO.05 directory on the companion disc. The dialog boxes below show the results of running the SysInfo application on several different platforms.

Windows 95 on Intel *x*86.

Windows NT on Intel *x*86.

Windows NT on MIPS R4000.

Windows NT on DEC Alpha.

SYSINFO.C

```
/*********************************************************
Module name: SysInfo.C
Notices: Copyright (c) 1995 Jeffrey Richter
*********************************************************/


#include "..\AdvWin32.H"   /* See Appendix B for details. */
#include <windows.h>
#include <windowsx.h>

#pragma warning(disable: 4001)     /* Single-line comment */

#include <tchar.h>
#include <stdio.h>
#include "Resource.H"


//////////////////////////////////////////////////////////////


typedef struct {
  const DWORD dwValue;
  LPCTSTR szText;
} LONGDATA;


LONGDATA CPUFlags[] = {
  { PROCESSOR_INTEL_386,     __TEXT("Intel 386")      },
  { PROCESSOR_INTEL_486,     __TEXT("Intel 486")      },
  { PROCESSOR_INTEL_PENTIUM, __TEXT("Intel Pentium")   },
  { PROCESSOR_INTEL_860,     __TEXT("Intel 860")      },
  { PROCESSOR_MIPS_R2000,    __TEXT("MIPS R2000")     },
  { PROCESSOR_MIPS_R3000,    __TEXT("MIPS R3000")     },
  { PROCESSOR_MIPS_R4000,    __TEXT("MIPS R4000")     },
  { PROCESSOR_ALPHA_21064,   __TEXT("DEC Alpha 21064") },
#ifdef PROCESSOR_PPC_601
  { PROCESSOR_PPC_601,       __TEXT("PowerPC 601")    },
  { PROCESSOR_PPC_603,       __TEXT("PowerPC 603")    },
  { PROCESSOR_PPC_604,       __TEXT("PowerPC 604")    },
  { PROCESSOR_PPC_620,       __TEXT("PowerPC 620")    },
#endif
  { 0, NULL }
};
```

Figure 5-1. *(continued)*
The SysInfo application.

Figure 5-1. *continued*

```
//////////////////////////////////////////////////////////////


LPCTSTR GetFlagStr (DWORD dwFlag, LONGDATA FlagList[],
  LPTSTR pszBuf) {

  int x;
```

```
  for (x = 0; FlagList[x].dwValue != 0; x++) {
    if (FlagList[x].dwValue == dwFlag)
      return(FlagList[x].szText);
  }

  _stprintf(pszBuf, __TEXT("Unknown (%d)"), dwFlag);
  return(pszBuf);
}


//////////////////////////////////////////////////////////


// This function accepts a number and converts it to a
// string, inserting commas where appropriate.
LPTSTR BigNumToString (LONG lNum, LPTSTR szBuf) {
  WORD wNumDigits = 0, wNumChars = 0;

  do {
    // Put the last digit of the string
    // in the character buffer.
    szBuf[wNumChars++] = (TCHAR) (lNum % 10 + __TEXT('0'));

    // Increment the number of digits
    // that we put in the string.
    wNumDigits++;

    // For every three digits put in
    // the string, add a comma (,).
    if (wNumDigits % 3 == 0)
      szBuf[wNumChars++] = __TEXT(',');

    // Divide the number by 10, and repeat the process.
    lNum /= 10;
```

(continued)

Figure 5-1. *continued*

```
    // Continue adding digits to the
    // string until the number is zero.
  } while (lNum != 0);

  // If the last character added to
  // the string was a comma, truncate it.
  if (szBuf[wNumChars - 1] == __TEXT(','))
    szBuf[wNumChars - 1] = 0;

  // Ensure that the string is zero-terminated.
  szBuf[wNumChars] = 0;

  // We added all the characters to the string in
  // reverse order. We must reverse the contents
  // of the string.
  _tcsrev(szBuf);

  // Returns the address of the string. This is the same
  // value that was passed to us initially. Returning it
  // here makes it easier for the calling function to
```

```
  // use the string.
  return(szBuf);
}


//////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog (HWND hwnd, HWND hwndFocus,
  LPARAM lParam) {

  TCHAR szBuf[50];
  SYSTEM_INFO si;

  // Associate an icon with the dialog box.
  SetClassLong(hwnd, GCL_HICON, (LONG)
    LoadIcon((HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE),
      __TEXT("SysInfo")));

  GetSystemInfo(&si);

  // Fill the static controls in the
  // list box with the appropriate number.
```

(continued)

Figure 5-1. *continued*

```
  SetDlgItemText(hwnd, IDC_OEMID,
    BigNumToString(si.dwOemId, szBuf));

  SetDlgItemText(hwnd, IDC_PAGESIZE,
    BigNumToString(si.dwPageSize, szBuf));

  SetDlgItemText(hwnd, IDC_MINAPPADDR,
    BigNumToString((LONG) si.lpMinimumApplicationAddress,
    szBuf));

  SetDlgItemText(hwnd, IDC_MAXAPPADDR,
    BigNumToString((LONG) si.lpMaximumApplicationAddress,
    szBuf));

  _stprintf(szBuf, __TEXT("0x%08X"),
    si.dwActiveProcessorMask);
  SetDlgItemText(hwnd, IDC_ACTIVEPROCMASK, szBuf);

  SetDlgItemText(hwnd, IDC_NUMOFPROCS,
    BigNumToString(si.dwNumberOfProcessors, szBuf));

  SetDlgItemText(hwnd, IDC_PROCTYPE,
    GetFlagStr(si.dwProcessorType, CPUFlags, szBuf));

  SetDlgItemText(hwnd, IDC_ALLOCGRAN,
    BigNumToString(si.dwAllocationGranularity, szBuf));

  return(TRUE);
}
```

```
//////////////////////////////////////////////////////////


void Dlg_OnCommand (HWND hwnd, int id, HWND hwndCtl,
  UINT codeNotify) {

  switch (id) {
    case IDCANCEL:
      EndDialog(hwnd, id);
      break;
  }
}
```

(continued)

Figure 5-1. *continued*

```
//////////////////////////////////////////////////////////


BOOL CALLBACK Dlg_Proc (HWND hDlg, UINT uMsg,
  WPARAM wParam, LPARAM lParam) {

  BOOL fProcessed = TRUE;

  switch (uMsg) {
    HANDLE_MSG(hDlg, WM_INITDIALOG, Dlg_OnInitDialog);
    HANDLE_MSG(hDlg, WM_COMMAND, Dlg_OnCommand);

    default:
      fProcessed = FALSE;
      break;
  }
  return(fProcessed);
}


//////////////////////////////////////////////////////////


int WINAPI WinMain (HINSTANCE hinstExe,
  HINSTANCE hinstPrev, LPSTR lpszCmdLine, int nCmdShow) {

  DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SYSINFO),
    NULL, Dlg_Proc);

  return(0);
}


////////////////////// End Of File //////////////////////
```

SYSINFO.RC

```
//Microsoft Visual C++ generated resource script.
//
#include "Resource.h"
```

```
#define APSTUDIO_READONLY_SYMBOLS
```

(continued)

Figure 5-1. *continued*

```
//////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS


#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
   "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
   "#include ""afxres.h""\r\n"
   "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
   "\r\n"
   "\0"
END

//////////////////////////////////////////////////////////////
#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_SYSINFO DIALOG DISCARDABLE  18, 18, 170, 103
```

(continued)

Figure 5-1. *continued*

```
STYLE WS_MINIMIZEBOX œ WS_POPUP œ WS_VISIBLE œ WS_CAPTION
  œ WS_SYSMENU
CAPTION "System Information"
FONT 8, "System"
```

```
BEGIN
    RTEXT           "OEM ID:",IDC_STATIC,4,4,88,8,SS_NOPREFIX
    RTEXT           "ID_OEMID",IDC_OEMID,96,4,68,8,
                    SS_NOPREFIX
    RTEXT           "Page size:",IDC_STATIC,4,16,88,8,
                    SS_NOPREFIX
    RTEXT           "ID_PAGESIZE",IDC_PAGESIZE,96,16,68,8,
                    SS_NOPREFIX
    RTEXT           "Minimum app. address:",IDC_STATIC,4,28,
                    88,8,SS_NOPREFIX
    RTEXT           "ID_MINAPPADDR",IDC_MINAPPADDR,96,28,68,
                    8,SS_NOPREFIX
    RTEXT           "Maximum app. address:",IDC_STATIC,4,40,88,
                    8,SS_NOPREFIX
    RTEXT           "ID_MAXAPPADDR",IDC_MAXAPPADDR,96,40,68,8,
                    SS_NOPREFIX
    RTEXT           "Active processor mask:",IDC_STATIC,4,52,88,
                    8,SS_NOPREFIX
    RTEXT           "ID_ACTIVEPROCMASK",IDC_ACTIVEPROCMASK,
                    96,52,68,8,SS_NOPREFIX
    RTEXT           "Number of processors:",IDC_STATIC,4,64,88,
                    8,SS_NOPREFIX
    RTEXT           "ID_NUMOFPROCS",IDC_NUMOFPROCS,96,64,68,8,
                    SS_NOPREFIX
    RTEXT           "Processor type:",IDC_STATIC,4,76,88,8,
                    SS_NOPREFIX
    RTEXT           "ID_PROCTYPE",IDC_PROCTYPE,96,76,68,8,
                    SS_NOPREFIX
    RTEXT           "Allocation granularity:",IDC_STATIC,4,88,
                    88,8,SS_NOPREFIX
    RTEXT           "ID_ALLOCGRAN",IDC_ALLOCGRAN,96,88,68,8,
                    SS_NOPREFIX
END


////////////////////////////////////////////////////////////////
//
// Icon
//
```

(continued)

Figure 5-1. *continued*

```
SYSINFO             ICON    DISCARDABLE     "SysInfo.ico"

#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Virtual Memory Status

There is a Win32 function called *GlobalMemoryStatus* that retrieves dynamic information about the current state of memory:

```
VOID GlobalMemoryStatus (LPMEMORYSTATUS lpmstMemStat);
```

I think that this function is very poorly named—*GlobalMemoryStatus* implies that the function is somehow related to the global heaps in 16-bit Windows. Win32 does not have a global heap but does offer the old global heap functions such as *GlobalAlloc* purely to ease the burden of porting a 16-bit Windows application to Win32. I think that *GlobalMemoryStatus* should have been called something like *VirtualMemoryStatus* instead.

When you call *GlobalMemoryStatus*, you must pass the address of a MEMORYSTATUS structure. Here is what the MEMORYSTATUS data structure looks like:

```
typedef struct _MEMORYSTATUS {
  DWORD dwLength;
  DWORD dwMemoryLoad;
  DWORD dwTotalPhys;
  DWORD dwAvailPhys;
  DWORD dwTotalPageFile;
  DWORD dwAvailPageFile;
  DWORD dwTotalVirtual;
  DWORD dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Before calling *GlobalMemoryStatus*, you must initialize the *dwLength* member to the size of the structure in bytes—that is, sizeof(MEMORYSTATUS). This allows Microsoft to add members to this structure in future versions of the Win32 API without breaking existing applications. When you call *GlobalMemoryStatus*, it will initialize the remainder of the structure's members and return. The VMStat sample application in the next section describes the various members and their meanings.

The Virtual Memory Status Sample Application

The VMStat application (VMSTAT.EXE), listed in Figure 5-2, displays a simple dialog box that lists the results of a call to *GlobalMemoryStatus*. The source code files, resource files, and make file for the application are in the VMSTAT.05 directory on the companion disc. Below is the result of running this program on Windows 95 using an 8-MB Intel 486 machine:

The *dwMemoryLoad* member (shown as Memory Load) gives a rough estimate of how busy the memory management system

is. This number can be anywhere from 0 to 100. The exact algorithm used to calculate this value varies between Windows 95 and Windows NT. In addition, the algorithm is subject to change in future versions of the operating system. In practice, the value reported by this member variable is all but useless.

The *dwTotalPhys* member (shown as TotalPhys) indicates the total number of bytes of physical memory (RAM) that exist. On this 8-MB 486 machine, this value is 6,983,680, which is just over 6.6 MB. This value is the exact amount of memory, including any holes in the address space between the low 640 KB and 1 MB of physical memory. The *dwAvailPhys* member (shown as AvailPhys) indicates the total number of bytes of physical memory available for allocation.

The *dwTotalPageFile* member (shown as TotalPageFile) indicates the maximum number of bytes that can be contained in the paging file(s) on your hard disk(s). Although VMStat reported that the paging file is currently 58,777,600 bytes, the system can expand and shrink the paging file as it sees fit. The *dwAvailPageFile* member (shown as AvailPageFile) indicates that 57,204,736 bytes in the paging file(s) are not committed to any process and are currently available should a process decide to commit any private storage.

The *dwTotalVirtual* member (shown as TotalVirtual) indicates the total number of bytes that are private in each process's address space. The value 2,143,289,344 is 4 MB short of being exactly 2 GB. The bottom 4 MB of inaccessible address space accounts for the 4-MB difference. If you run VMStat under Windows NT, you'll see that *dwTotalVirtual* comes back with a value of 2,147,352,576, which is just 128 KB short of being exactly 2 GB. The 128-KB difference exists because the system never lets an application gain access to the 64 KB at the beginning or the 64 KB at the end of a 2-GB mark of address space.

The last member, *dwAvailVirtual* (shown as AvailVirtual), is the only member of the structure specific to the process calling *GlobalMemoryStatus*—all the other members apply to the system and would be the same regardless of which process was calling *GlobalMemoryStatus*. To calculate this value, *GlobalMemoryStatus* adds up all of the free regions in the calling process's address space. The *dwAvailVirtual* value 2,139,422,720 indicates the amount of free address space that is available for VMStat to do with what it wants. If you subtract the *dwAvailVirtual* member from the *dwTotalVirtual* member, you'll see that VMStat has 3,866,624 bytes reserved in its virtual address space.

There is no member that indicates the amount of physical storage currently in use by the process.

VMSTAT.C

```
/************************************************************
Module name: VMStat.C
Notices: Copyright (c) 1995 Jeffrey Richter
************************************************************/


#include "..\AdvWin32.H"    /* See Appendix B for details. */
#include <windows.h>
#include <windowsx.h>

#pragma warning(disable: 4001)      /* Single-line comment */

#include <tchar.h>
#include "Resource.H"
```

Figure 5-2. *(continued)*
The VMStat application.

Figure 5-2. *continued*

```
//////////////////////////////////////////////////////////////

// This function accepts a number and converts it to a string,
// inserting commas where appropriate.
LPTSTR WINAPI BigNumToString (LONG lNum, LPTSTR szBuf) {
  WORD wNumDigits = 0, wNumChars = 0;

  do {
    // Put the last digit of the string
    // in the character buffer.
    szBuf[wNumChars++] = (TCHAR) (lNum % 10 + __TEXT('0'));

    // Increment the number of digits
    // that we put in the string.
    wNumDigits++;

    // For every three digits put in
    // the string, add a comma (,).
    if (wNumDigits % 3 == 0)
      szBuf[wNumChars++] = __TEXT(',');

    // Divide the number by 10, and repeat the process.
    lNum /= 10;
    // Continue adding digits to
    // the string until the number is zero.
  } while (lNum != 0);

  // If the last character added to
  // the string was a comma, truncate it.
  if (szBuf[wNumChars - 1] == __TEXT(','))
    szBuf[wNumChars - 1] = 0;

  // Ensure that the string is zero-terminated.
  szBuf[wNumChars] = 0;

  // We added all the characters to the string in reverse
  // order. We must reverse the contents of the string.
```

```
   _tcsrev(szBuf);

   // Returns the address of the string. This is the same
   // value that was passed to us initially. Returning it
   // here makes it easier for the calling function
   // to use the string.
   return(szBuf);
}
```

(continued)

Figure 5-2. *continued*

```
//////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog (HWND hwnd, HWND hwndFocus,
   LPARAM lParam) {

   TCHAR szBuf[50];
   MEMORYSTATUS ms;

   // Associate an icon with the dialog box.
   SetClassLong(hwnd, GCL_HICON, (LONG)
      LoadIcon((HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE),
      (LPCTSTR) __TEXT("VMStat")));

   // Initialize the structure length before
   // calling GlobalMemoryStatus.
   ms.dwLength = sizeof(ms);
   GlobalMemoryStatus(&ms);

   // Fill the static controls in the
   // list box with the appropriate number.
   SetDlgItemText(hwnd, IDC_MEMLOAD,
      BigNumToString(ms.dwMemoryLoad, szBuf));

   SetDlgItemText(hwnd, IDC_TOTALPHYS,
      BigNumToString(ms.dwTotalPhys, szBuf));

   SetDlgItemText(hwnd, IDC_AVAILPHYS,
      BigNumToString(ms.dwAvailPhys, szBuf));

   SetDlgItemText(hwnd, IDC_TOTALPAGEFILE,
      BigNumToString(ms.dwTotalPageFile, szBuf));

   SetDlgItemText(hwnd, IDC_AVAILPAGEFILE,
      BigNumToString(ms.dwAvailPageFile, szBuf));

   SetDlgItemText(hwnd, IDC_TOTALVIRTUAL,
      BigNumToString(ms.dwTotalVirtual, szBuf));

   SetDlgItemText(hwnd, IDC_AVAILVIRTUAL,
      BigNumToString(ms.dwAvailVirtual, szBuf));

   return(TRUE);
}
```

(continued)

Figure 5-2. *continued*

```
////////////////////////////////////////////////////////////

void Dlg_OnCommand (HWND hwnd, int id, HWND hwndCtl,
  UINT codeNotify) {

  switch (id) {
    case IDCANCEL:
      EndDialog(hwnd, id);
      break;
  }
}


////////////////////////////////////////////////////////////

BOOL CALLBACK Dlg_Proc (HWND hDlg, UINT uMsg,
  WPARAM wParam, LPARAM lParam) {

  BOOL fProcessed = TRUE;

  switch (uMsg) {
    HANDLE_MSG(hDlg, WM_INITDIALOG, Dlg_OnInitDialog);
    HANDLE_MSG(hDlg, WM_COMMAND, Dlg_OnCommand);

    default:
      fProcessed = FALSE;
      break;
  }
  return(fProcessed);
}


////////////////////////////////////////////////////////////

int WINAPI WinMain (HINSTANCE hinstExe,
  HINSTANCE hinstPrev, LPSTR lpszCmdLine, int nCmdShow) {

  DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMSTAT),
    NULL, Dlg_Proc);

  return(0);
}


////////////////////// End Of File //////////////////////
```

(continued)

Figure 5-2. *continued*

VMSTAT.RC

```
//Microsoft Visual C++ generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
```

```
//////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS


#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
   "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
   "#include ""afxres.h""\r\n"
   "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
   "\r\n"
   "\0"
END

//////////////////////////////////////////////////////////////
#endif    // APSTUDIO_INVOKED
```

(continued)

Figure 5-2. *continued*

```
//////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_VMSTAT DIALOG DISCARDABLE  60, 27, 129, 101
STYLE WS_MINIMIZEBOX œ WS_POPUP œ WS_VISIBLE œ WS_CAPTION
     œ WS_SYSMENU
CAPTION "Virtual Memory Status"
FONT 8, "System"
BEGIN
   RTEXT         "Memory load:",IDC_STATIC,4,4,52,8
   RTEXT         "Text",IDC_MEMLOAD,60,4,60,8
   RTEXT         "TotalPhys:",IDC_STATIC,4,20,52,8
   RTEXT         "Text",IDC_TOTALPHYS,60,20,60,8
   RTEXT         "AvailPhys:",IDC_STATIC,4,32,52,8
   RTEXT         "Text",IDC_AVAILPHYS,60,32,60,8
   RTEXT         "TotalPageFile:",IDC_STATIC,4,48,52,8
   RTEXT         "Text",IDC_TOTALPAGEFILE,60,48,60,8
```

```
    RTEXT           "AvailPageFile:",IDC_STATIC,4,60,52,8
    RTEXT           "Text",IDC_AVAILPAGEFILE,60,60,60,8
    RTEXT           "TotalVirtual:",IDC_STATIC,4,76,52,8
    RTEXT           "Text",IDC_TOTALVIRTUAL,60,76,60,8
    RTEXT           "AvailVirtual:",IDC_STATIC,4,88,52,8
    RTEXT           "Text",IDC_AVAILVIRTUAL,60,88,60,8
END


////////////////////////////////////////////////////////////
//
// Icon
//

VMStat              ICON    DISCARDABLE     "VMStat.Ico"

#ifndef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Determining the State of an Address Space

Win32 offers a function that lets you query certain information (for example, size, storage type, and protection attributes) about a memory address in your address space. In fact, the VMMap sample application shown later in this chapter uses this function to produce the virtual memory map dumps that appeared in Chapter 4. This Win32 function is called *VirtualQuery*:

```
DWORD VirtualQuery(LPVOID lpAddress,
  PMEMORY_BASIC_INFORMATION lpBuffer,
  DWORD dwLength);
```

When you call *VirtualQuery*, the first parameter, *lpAddress*, must contain the virtual memory address that you want information about. The *lpBuffer* parameter is the address to a MEMORY_BASIC_INFORMATION structure that you must allocate. This structure is defined in WINNT.H as follows:

```
typedef struct _MEMORY_BASIC_INFORMATION {
  PVOID BaseAddress;
  PVOID AllocationBase;
  DWORD AllocationProtect;
  DWORD RegionSize;
  DWORD State;
  DWORD Protect;
  DWORD Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

The last parameter, *dwLength*, specifies the size of a MEMORY_BASIC_INFORMATION structure. *VirtualQuery* returns the number of bytes copied into the buffer.

Based on the address that you pass in the *lpAddress* parameter, *VirtualQuery* fills the MEMORY_BASIC_INFORMATION structure with information about the range of adjoining pages that share the same state, protection attributes, and type. See the table on the following page for a description of the structure's members.

Member Name	Description
BaseAddress	This is the same value as the *lpAddress* parameter rounded down to an even page boundary.
AllocationBase	Identifies the base address of the region containing the address specified in the *lpAddress* parameter.
AllocationProtect	Identifies the protection attribute assigned to the region when it was initially reserved.
RegionSize	Identifies the size, in bytes, for all pages starting at *BaseAddress* that have the same protection attributes, state, and type as the page containing the address specified in the *lpAddress* parameter.
State	Identifies the state (MEM_FREE, MEM_RESERVE, or MEM_COMMIT) for all adjoining pages that have the same protection attributes, state, and type as the page containing the address specified in the *lpAddress* parameter. If the state is free, the *AllocationBase*, *AllocationProtect*, *Protect*, and *Type* members are undefined. If the state is reserve, the *Protect* member is undefined.
Protect	Identifies the protection attribute (PAGE_*) for all adjoining pages that have the same protection attributes, state, and type as the page containing the address specified in the *lpAddress* parameter.
Type	Identifies the type of physical storage (MEM_IMAGE, MEM_MAPPED, or MEM_PRIVATE) that is backing all adjoining pages that have the same protection attributes, state, and type as the page containing the address specified in the *lpAddress* parameter. For Windows 95, this member will always indicate MEM_PRIVATE.

The *VMQuery* Function

When I was first learning how the Win32 memory architecture is designed, I used *VirtualQuery* as my guide. In fact, if you examine the first edition of this book, you'll see that the VMMAP.EXE program was much simpler than the new version I

present in the next section. In the old version, I had a very simple loop that called *VirtualQuery* repeatedly, and for each call, I simply constructed a single line containing the members of the MEMORY_BASIC_INFORMATION structure. I studied this dump and tried to piece the Win32 memory management architecture together while referring to the Windows NT 3.1 SDK documentation (which was rather poor at the time). Well, I've come a long way, baby—I now know that the *VirtualQuery* function and the MEMORY_BASIC_INFORMATION structure are not good for creating a process's virtual address space memory map.

The problem is that the MEMORY_BASIC_INFORMATION structure does not return all of the information that the system has stored internally. If you have a memory address and want to obtain some sim-ple information about it, *VirtualQuery* is great. If you just want to know whether there is committed physical storage to an address or whether a memory address can be read from or written to, *VirtualQuery* works fine. But if you want to know the total size of a reserved region or the number of blocks in a region, or whether a region contains a thread's stack, a single call to *VirtualQuery* is just not going to give you the information you're looking for.

In order to obtain much more complete memory information, I have created my own function, named *VMQuery*:

```
BOOL VMQuery (PVOID pvAddress, PVMQUERY pVMQ);
```

This function is similar to *VirtualQuery* in that it takes a memory address specified by the *pvAddress* parameter and a pointer to a structure that is to be filled, specified by the *pVMQ* parameter. This structure is a VMQUERY structure that I have also defined:

```
typedef struct {
  // Region information
  PVOID pvRgnBaseAddress;
  DWORD dwRgnProtection;  // PAGE_*
  DWORD dwRgnSize;
  DWORD dwRgnStorage;     // MEM_*: Free, Image,
                          //        Mapped, Private
  DWORD dwRgnBlocks;
  DWORD dwRgnGuardBlks;   // If > 0, region contains thread stack
  BOOL  fRgnIsAStack;     // TRUE if region contains thread stack

  // Block information
  PVOID pvBlkBaseAddress;
  DWORD dwBlkProtection;  // PAGE_*
  DWORD dwBlkSize;
  DWORD dwBlkStorage;     // MEM_*: Free, Reserve, Image,
                          //        Mapped, Private
} VMQUERY, *PVMQUERY;
```

As you can see from just a quick glance, my VMQUERY structure contains much more information than *VirtualQuery*'s MEMORY_BASIC_INFORMATION structure. My structure is divided into two distinct parts: region information and block information.

The region portion describes information about the region, and the block portion contains information about the block containing the address specified by the *pvAddress* parameter. The table below describes all the members:

Member Name	Description
pvRgnBaseAddress	Identifies the base address of the virtual address space region containing the address specified in the *pvAddress* parameter.
dwRgnProtection	Identifies the protection attribute that was assigned to the region of address space when it was initially - reserved.
dwRgnSize	Identifies the size, in bytes, of the region that was - reserved.
dwRgnStorage	Identifies the type of physical storage that is used for the bulk of the blocks in the region. The value is one of the following: MEM_FREE, MEM_IMAGE, MEM_MAPPED, or MEM_PRIVATE. Windows 95 doesn't distinguish between different storage types, so this member will always be MEM_FREE or MEM_PRIVATE under Windows 95.
dwRgnBlocks	Identifies the number of blocks contained within the region.
dwRgnGuardBlks	Identifies the number of blocks that have the PAGE_GUARD protection attribute flag turned on. This value will usually be either 0 or 1. If it's 1, that's a good indicator that the region was reserved to contain a thread's stack. Under Windows 95, this member will always be 0.
fRgnIsAStack	Identifies whether the region contains a thread's stack. This value is determined by taking a "best guess" because it is impossible to be 100 percent sure whether a region contains a stack.
pvBlkBaseAddress	Identifies the base address of the block that contains the address specified in the *pvAddress* parameter.
dwBlkProtection	Identifies the protection attribute for the block that contains the address specified in the *pvAddress* parameter.
dwBlkSize	Identifies the size, in bytes, of the block that contains the address specified in the *pvAddress* parameter.

(continued)

continued

Member Name	Description
dwBlkStorage	Identifies the content of the block that contains the address specified in the *pvAddress* parameter. The

	value is one of the following: MEM_FREE, MEM_RESERVE, MEM_IMAGE, MEM_MAPPED, or MEM_PRIVATE. Under Windows 95, this member will never be MEM_IMAGE or MEM_MAPPED.

There is no doubt that *VMQuery* must do a significant amount of processing, including many calls to *VirtualQuery*, in order to obtain all this information—which means it executes much more slowly than *VirtualQuery*. For this reason, you should think carefully when deciding which of these two functions to call. If you do not need the extra information obtained by *VMQuery*, call *VirtualQuery*.

The VMQUERY.C file, listed in Figure 5-3, shows how I obtain and massage all the information needed to set the members of the VMQUERY structure. The VMQUERY.C and VMQUERY.H files are in the VMMAP.05 directory on the companion disc. Rather than go into detail in the text about how I process this data, I'll let my comments (sprinkled liberally throughout the code) speak for themselves.

VMQUERY.C

```
/*************************************************************
Module name: VMQuery.C
Notices: Copyright (c) 1995 Jeffrey Richter
*************************************************************/


#include "..\AdvWin32.H"     /* See Appendix B for details.*/
#include <windows.h>
#include <windowsx.h>

#pragma warning(disable: 4001)      /* Single-line comment */

#include "VMQuery.H"


//////////////////////////////////////////////////////////////////////
```

Figure 5-3. *(continued)*
The VMQuery listings.

Figure 5-3. *continued*

```
typedef struct {
  DWORD dwRgnSize;
  DWORD dwRgnStorage;     // MEM_*: Free, Image,
                  //        Mapped, Private
  DWORD dwRgnBlocks;
  DWORD dwRgnGuardBlks;   // If > 0, region contains
                  // thread stack
  BOOL  fRgnIsAStack;     // TRUE if region contains
                  // thread stack
} VMQUERY_HELP;
```

```
// Global-static variable that holds the
// allocation granularity value for this CPU platform. This
// variable is initialized the first time VMQuery is called.
static DWORD gs_dwAllocGran = 0;


//////////////////////////////////////////////////////////////


// When NTBUG_VIRTUALQUERY is defined, the code below
// compensates for a bug in Windows NT's implementation of
// the VirtualQuery function.
#define NTBUG_VIRTUALQUERY

#ifdef NTBUG_VIRTUALQUERY
DWORD NTBug_VirtualQuery (LPVOID lpvAddress,
  PMEMORY_BASIC_INFORMATION pmbiBuffer, DWORD cbLength) {

  DWORD dwRetVal = VirtualQuery(lpvAddress,
    pmbiBuffer, cbLength);

  if (dwRetVal == cbLength) {
    // If successful, correct the MBI structure's values.

    if (((DWORD) pmbiBuffer->AllocationBase % 0x1000)
      == 0xFFF) {
      // If the AllocationBase member ends with 0xFFF,
      // the address is 1 byte off.
      pmbiBuffer->AllocationBase = (PVOID)
        ((PBYTE) pmbiBuffer->AllocationBase + 1);
    }
```

(continued)

Figure 5-3. *continued*

```
    if ((pmbiBuffer->RegionSize % 0x1000) == 0xFFF) {
      // If the RegionSize member ends with 0xFFF,
      // the size is 1 byte off.
      pmbiBuffer->RegionSize++;
    }

    if ((pmbiBuffer->State != MEM_FREE) &&
       (pmbiBuffer->AllocationProtect == 0)) {
      // If the the region is not free and the
      // AllocationProtect member is 0, AllocationProtect
      // should be PAGE_READONLY.
      pmbiBuffer->AllocationProtect = PAGE_READONLY;
    }
  }
  return(dwRetVal);
}


#define VirtualQuery NTBug_VirtualQuery
#endif


//////////////////////////////////////////////////////////////
```

```
// This function iterates through all the blocks in a
// region and initializes a structure with its findings.
static BOOL VMQueryHelp (PVOID pvAddress,
  VMQUERY_HELP *pVMQHelp) {

  MEMORY_BASIC_INFORMATION MBI;
  PVOID pvRgnBaseAddress, pvAddressBlk;
  BOOL fOk;
  DWORD dwProtectBlock[4] = { 0 };
    // 0 = reserved, PAGE_NOACCESS, PAGE_READWRITE

  // Zero the contents of the structure.
  memset(pVMQHelp, 0, sizeof(*pVMQHelp));

  // From the passed memory address, obtain the
  // base address of the region that contains it.
  fOk = (VirtualQuery(pvAddress,
    &MBI, sizeof(MBI)) == sizeof(MBI));
```

(continued)

Figure 5-3. *continued*

```
  if (!fOk) {
    // If we can't get any information about the passed
    // address, return FALSE, indicating an error.
    // GetLastError() will report the actual problem.
    return(fOk);
  }

  // pvRgnBaseAddress identifies the region's
  // base address and will never change.
  pvRgnBaseAddress = MBI.AllocationBase;

  // pvAddress identifies the address of the first block
  // and will change as we iterate through the blocks.
  pvAddressBlk = pvRgnBaseAddress;

  // Save the memory type of the physical storage block.
  pVMQHelp->dwRgnStorage = MBI.Type;

  for (;;) {
    // Get info about the current block.
    fOk = VirtualQuery(pvAddressBlk, &MBI, sizeof(MBI));
    if (!fOk) {
      // Couldn't get the information, end loop.
      break;
    }

    // Check to see whether the block we got info for is
    // contained in the requested region.
    if (MBI.AllocationBase != pvRgnBaseAddress) {
      // Found a block in the next region; end loop.
      break;
    }


    // We have found a block contained
    // in the requested region.
```

```
      // The following if statement is for detecting stacks in
      // Windows 95. Windows 95 stacks are in a region wherein
      // the last 4 blocks have the following attributes:
      // reserved block, PAGE_NOACCESS, PAGE_READWRITE,
      // and another reserved block.
      if (pVMQHelp->dwRgnBlocks < 4) {
        // If this is the 0th through 3rd block, make
        // a note of the block's protection in our array.
```

(continued)

Figure 5-3. *continued*

```
        dwProtectBlock[pVMQHelp->dwRgnBlocks] =
          (MBI.State == MEM_RESERVE) ? 0 : MBI.Protect;
      } else {
        // We have already seen 4 blocks in this region.
        // Shift the protection values down in the array.
        MoveMemory(&dwProtectBlock[0], &dwProtectBlock[1],
          sizeof(dwProtectBlock) - sizeof(DWORD));

        // Add the new protection value to the end
        // of the array.
        dwProtectBlock[3] =
          (MBI.State == MEM_RESERVE) ? 0 : MBI.Protect;
      }

      // Add 1 to the number of blocks in the region.
      pVMQHelp->dwRgnBlocks++;

      // Add the block's size to the reserved region size.
      pVMQHelp->dwRgnSize += MBI.RegionSize;

      // If the block has the PAGE_GUARD protection attribute
      // flag, add 1 to the number of blocks with this flag.
      if (MBI.Protect & PAGE_GUARD) {
        pVMQHelp->dwRgnGuardBlks++;
      }


      // Take a best guess as to the type of physical storage
      // committed to the block. This is a guess because some
      // blocks can convert from MEM_IMAGE to MEM_PRIVATE or
      // from MEM_MAPPED to MEM_PRIVATE; MEM_PRIVATE can
      // always be overridden by MEM_IMAGE or MEM_MAPPED.
      if (pVMQHelp->dwRgnStorage == MEM_PRIVATE) {
        pVMQHelp->dwRgnStorage = MBI.Type;
      }


      // Get the address of the next block.
      pvAddressBlk = (PVOID)
        ((PBYTE) pvAddressBlk + MBI.RegionSize);
    }

    // After examining the region, check to see whether it is
    // a thread stack.
```

```
   // Windows NT: Assume a thread stack if the region contains
   //             at least 1 block with the PAGE_GUARD flag.
```

(continued)

Figure 5-3. *continued*

```
   // Windows 95: Assume a thread stack if the region contains
   //             at least 4 blocks wherein the last 4 blocks
   //             have the following attributes:
   //             3rd block from end: reserved
   //             2nd block from end: PAGE_NOACCESS
   //             1st block from end: PAGE_READWRITE
   //             block at end: another reserved block.
   pVMQHelp->fRgnIsAStack =
      (pVMQHelp->dwRgnGuardBlks > 0) œœ
      ((pVMQHelp->dwRgnBlocks >= 4) &&
       (dwProtectBlock[0] == 0) &&
       (dwProtectBlock[1] == PAGE_NOACCESS) &&
       (dwProtectBlock[2] == PAGE_READWRITE) &&
       (dwProtectBlock[3] == 0));

   // Return that the function completed successfully.
   return(TRUE);
}


//////////////////////////////////////////////////////////////////


BOOL VMQuery (PVOID pvAddress, PVMQUERY pVMQ) {

   MEMORY_BASIC_INFORMATION MBI;
   VMQUERY_HELP VMQHelp;
   BOOL fOk;

   if (gs_dwAllocGran == 0) {
      // If this is the very first time a thread in this
      // application is calling us, we must obtain the size
      // of a page used on this system and save this value
      // in a global-static variable.
      SYSTEM_INFO SI;
      GetSystemInfo(&SI);
      gs_dwAllocGran = SI.dwAllocationGranularity;
   }


   // Zero the contents of the structure.
   memset(pVMQ, 0, sizeof(*pVMQ));

   // Get the MEMORY_BASIC_INFORMATION for the passed address.
   fOk = VirtualQuery(pvAddress,
      &MBI, sizeof(MBI)) == sizeof(MBI);
```

(continued)

Figure 5-3. *continued*

```
   if (!fOk) {
      // If we can't get any information about the passed
      // address, return FALSE, indicating an error.
```

```
  // GetLastError() will report the actual problem.
  return(fOk);
}

// The MEMORY_BASIC_INFORMATION structure contains valid
// information. Time to start setting the members
// of our own VMQUERY structure.

// First, fill in the block members. We'll get the
// data for the region containing the block later.
switch (MBI.State) {
  case MEM_FREE:
    // We have a block of free address space that
    // has not been reserved.
    pVMQ->pvBlkBaseAddress = NULL;
    pVMQ->dwBlkSize = 0;
    pVMQ->dwBlkProtection = 0;
    pVMQ->dwBlkStorage = MEM_FREE;
    break;

  case MEM_RESERVE:
    // We have a block of reserved address space that
    // does NOT have physical storage committed to it.
    pVMQ->pvBlkBaseAddress = MBI.BaseAddress;
    pVMQ->dwBlkSize = MBI.RegionSize;

    // For an uncommitted block, MBI.Protect is invalid.
    // So we will show that the reserved block inherits
    // the protection attribute of the region in which it
    // is contained.
    pVMQ->dwBlkProtection = MBI.AllocationProtect;
    pVMQ->dwBlkStorage = MEM_RESERVE;
    break;

  case MEM_COMMIT:
    // We have a block of reserved address space that
    // DOES have physical storage committed to it.
    pVMQ->pvBlkBaseAddress = MBI.BaseAddress;
    pVMQ->dwBlkSize = MBI.RegionSize;
    pVMQ->dwBlkProtection = MBI.Protect;
    pVMQ->dwBlkStorage = MBI.Type;
    break;
}
```

(continued)

Figure 5-3. *continued*

```
// Second, fill in the region members now that we have
// used the MBI data obtained from the first call to
// VirtualQuery. We might have to call VirtualQuery again
// to obtain complete region information.
switch (MBI.State) {
  case MEM_FREE:
    // We have a block of address space
    // that has not been reserved.
    pVMQ->pvRgnBaseAddress = MBI.BaseAddress;
    pVMQ->dwRgnProtection = MBI.AllocationProtect;
    pVMQ->dwRgnSize = MBI.RegionSize;
    pVMQ->dwRgnStorage = MEM_FREE;
```

```
      pVMQ->dwRgnBlocks = 0;
      pVMQ->dwRgnGuardBlks = 0;
      pVMQ->fRgnIsAStack = FALSE;
      break;

    case MEM_RESERVE:
      // We have a reserved region that does NOT have
      // physical storage committed to it.
      pVMQ->pvRgnBaseAddress = MBI.AllocationBase;
      pVMQ->dwRgnProtection = MBI.AllocationProtect;

      // To get complete information about the region, we
      // must iterate through all the region's blocks.
      VMQueryHelp(pvAddress, &VMQHelp);

      pVMQ->dwRgnSize = VMQHelp.dwRgnSize;
      pVMQ->dwRgnStorage = VMQHelp.dwRgnStorage;
      pVMQ->dwRgnBlocks = VMQHelp.dwRgnBlocks;
      pVMQ->dwRgnGuardBlks = VMQHelp.dwRgnGuardBlks;
      pVMQ->fRgnIsAStack = VMQHelp.fRgnIsAStack;
      break;

    case MEM_COMMIT:
      // We have a reserved region that DOES have
      // physical storage committed to it.
      pVMQ->pvRgnBaseAddress = MBI.AllocationBase;
      pVMQ->dwRgnProtection = MBI.AllocationProtect;

      // To get complete information about the region, we
      // must iterate through all the region's blocks.
      VMQueryHelp(pvAddress, &VMQHelp);
```

(continued)

Figure 5-3. *continued*

```
      pVMQ->dwRgnSize = VMQHelp.dwRgnSize;
      pVMQ->dwRgnStorage = VMQHelp.dwRgnStorage;
      pVMQ->dwRgnBlocks = VMQHelp.dwRgnBlocks;
      pVMQ->dwRgnGuardBlks = VMQHelp.dwRgnGuardBlks;
      pVMQ->fRgnIsAStack = VMQHelp.fRgnIsAStack;
      break;
  }

  // Return that the function completed successfully.
  return(fOk);
}

////////////////////////// End Of File //////////////////////////
```

VMQUERY.H

```
/******************************************************************
Module name: VMQuery.H
Notices: Copyright (c) 1995 Jeffrey Richter
******************************************************************/
```

```
typedef struct {
  // Region information
  PVOID pvRgnBaseAddress;
  DWORD dwRgnProtection;  // PAGE_*
  DWORD dwRgnSize;
  DWORD dwRgnStorage;     // MEM_*: Free, Image,
                          //        Mapped, Private
  DWORD dwRgnBlocks;
  DWORD dwRgnGuardBlks;   // If > 0, region contains
                          // thread stack
  BOOL  fRgnIsAStack;     // TRUE if region contains
                          // thread stack

  // Block information
  PVOID pvBlkBaseAddress;
  DWORD dwBlkProtection;  // PAGE_*
  DWORD dwBlkSize;
  DWORD dwBlkStorage;     // MEM_*: Free, Reserve, Image,
                          //        Mapped, Private
} VMQUERY, *PVMQUERY;
```

(continued)

Figure 5-3. *continued*

```
//////////////////////////////////////////////////////////////

BOOL VMQuery (PVOID pvAddress, PVMQUERY pVMQ);

////////////////////////// End Of File //////////////////////////
```

The Virtual Memory Map Sample Application

The VMMap application (VMMAP.EXE), listed in Figure 5-4 beginning on page 160, walks its own address space and shows the regions and the blocks within regions. The source code files, resource files, and make file for the application are in the VMMAP.05 directory on the companion disc. When you start the program, the following window appears:

The contents of this application's list box were used to produce the virtual memory map dumps presented in Figure 4-5 on page 112, Figure 4-6 on page 115, and Figure 4-7 on page 120 in Chapter 4.

Each entry in the list box shows the result of information obtained by calling my *VMQuery* function. The main loop looks like this:

```
PVOID pvAddress = 0x00000000;
BOOL fOk = TRUE;
VMQUERY VMQ;

.
.
.
  while (fOk) {
    fOk = VMQuery(pvAddress, &VMQ);

    if (fOk) {
      // Construct the line to be displayed, and
      // add it to the list box.
      ConstructRgnInfoLine(&VMQ, szLine, sizeof(szLine));
      ListBox_AddString(hWndLB, szLine);

  #if 1
      // Change the 1 above to a 0 if you do not want
      // to see the blocks contained within the region.

      for (dwBlock = 0; fOk && (dwBlock < VMQ.dwRgnBlocks);
        dwBlock++) {

        ConstructBlkInfoLine(&VMQ, szLine, sizeof(szLine));
        ListBox_AddString(hWndLB, szLine);

        // Get the address of the next region to test.
        pvAddress = ((BYTE *) pvAddress + VMQ.dwBlkSize);
        if (dwBlock < VMQ.dwRgnBlocks - 1) {
          // Don't query the memory info after
          // the last block.
          fOk = VMQuery(pvAddress, &VMQ);
        }
      }
  #endif

      // Get the address of the next region to test.
      pvAddress = ((BYTE *) VMQ.pvRgnBaseAddress +
        VMQ.dwRgnSize);
    }
  }
```

This loop starts walking from virtual address 0x00000000 and ends when *VMQuery* returns FALSE, indicating that it can no longer walk the process's address space. With each iteration of the loop, there is a call to *ConstructRgnInfoLine*; this function fills a character buffer with information about the region. Then this information is appended to the list.

Within this main loop, there is a nested loop that iterates through each of the blocks in the region. Each iteration of this loop calls *ConstructBlkInfoLine* to fill a character buffer with information about the region's blocks. Then the information is appended to the list box. It's very easy to walk the process's address space using the *VMQuery* function.

VMMAP.C

```
/******************************************************************
Module name: VMMap.C
Notices: Copyright (c) 1995 Jeffrey Richter
******************************************************************/


#include "..\AdvWin32.H"    /* See Appendix B for details. */
#include <windows.h>
#include <windowsx.h>

#pragma warning(disable: 4001)      /* Single-line comment */
#include <tchar.h>
#include <stdio.h>            // For sprintf
#include <string.h>           // For strchr
#include "Resource.H"
#include "VMQuery.H"


//////////////////////////////////////////////////////////////


// Set COPYTOCLIPBOARD to TRUE if you want the
// memory map to be copied to the clipboard.
#define COPYTOCLIPBOARD    FALSE


#if COPYTOCLIPBOARD
// Function to copy the contents of a list box to the clipboard.
// I used this function to obtain the memory map dumps
// for the figures in this book.


void CopyControlToClipboard (HWND hwnd) {
  int nCount, nNum;
  TCHAR szClipData[20000] = { 0 };
  HGLOBAL hClipData;
  LPTSTR lpClipData;
  BOOL fOk;

  nCount = ListBox_GetCount(hwnd);
  for (nNum = 0; nNum < nCount; nNum++) {
    TCHAR szLine[1000];
    ListBox_GetText(hwnd, nNum, szLine);
```

Figure 5-4. *(continued)*
The VMMap application.

Figure 5-4. *continued*

```
    _tcscat(szClipData, szLine);
    _tcscat(szClipData, __TEXT("\r\n"));
  }

  OpenClipboard(NULL);
  EmptyClipboard();

  // Clipboard accepts only data that is in a block allocated
```

```
  // with GlobalAlloc using the GMEM_MOVEABLE and
  // GMEM_DDESHARE flags.
  hClipData = GlobalAlloc(GMEM_MOVEABLE œ GMEM_DDESHARE,
    sizeof(TCHAR) * (_tcslen(szClipData) + 1));
  lpClipData = (LPTSTR) GlobalLock(hClipData);

  _tcscpy(lpClipData, szClipData);

#ifdef UNICODE
  fOk = (SetClipboardData(CF_UNICODETEXT, hClipData)
    == hClipData);
#else
  fOk = (SetClipboardData(CF_TEXT, hClipData) == hClipData);
#endif
  CloseClipboard();

  if (!fOk) {
    GlobalFree(hClipData);
    MessageBox(GetFocus(),
      __TEXT("Error putting text on the clipboard"),
      NULL, MB_OK œ MB_ICONINFORMATION);
  }
}

#endif


////////////////////////////////////////////////////////////


LPCTSTR GetMemStorageText (DWORD dwStorage) {
  LPCTSTR p = __TEXT("Unknown");
  switch (dwStorage) {
    case MEM_FREE:    p = __TEXT("Free   "); break;
    case MEM_RESERVE: p = __TEXT("Reserve"); break;
    case MEM_IMAGE:   p = __TEXT("Image  "); break;
```

(continued)

Figure 5-4. *continued*

```
    case MEM_MAPPED:  p = __TEXT("Mapped "); break;
    case MEM_PRIVATE: p = __TEXT("Private"); break;
  }
  return(p);
}


////////////////////////////////////////////////////////////


LPTSTR GetProtectText (DWORD dwProtect, LPTSTR szBuf,
  BOOL fShowFlags) {
  LPCTSTR p = __TEXT("Unknown");
  switch (dwProtect & ~(PAGE_GUARD œ PAGE_NOCACHE)) {
    case PAGE_READONLY:          p = __TEXT("-R--"); break;
    case PAGE_READWRITE:         p = __TEXT("-RW-"); break;
    case PAGE_WRITECOPY:         p = __TEXT("-RWC"); break;
    case PAGE_EXECUTE:           p = __TEXT("E---"); break;
    case PAGE_EXECUTE_READ:      p = __TEXT("ER--"); break;
```

```
      case PAGE_EXECUTE_READWRITE:  p = __TEXT("ERW-"); break;
      case PAGE_EXECUTE_WRITECOPY:  p = __TEXT("ERWC"); break;
      case PAGE_NOACCESS:           p = __TEXT("----"); break;
    }
    _tcscpy(szBuf, p);
    if (fShowFlags) {
      _tcscat(szBuf, __TEXT(" "));
      _tcscat(szBuf, (dwProtect & PAGE_GUARD)   ?
        __TEXT("G") : __TEXT("-"));
      _tcscat(szBuf, (dwProtect & PAGE_NOCACHE) ?
        __TEXT("N") : __TEXT("-"));
    }
    return(szBuf);
  }


  ////////////////////////////////////////////////////////////

  void ConstructRgnInfoLine (PVMQUERY pVMQ,
    LPTSTR szLine, int nMaxLen) {

    int nLen;

    _stprintf(szLine, __TEXT("%08X     %s  %10u  "),
      pVMQ->pvRgnBaseAddress,
      GetMemStorageText(pVMQ->dwRgnStorage),
      pVMQ->dwRgnSize);
```

(continued)

Figure 5-4. *continued*

```
    if (pVMQ->dwRgnStorage != MEM_FREE) {
      _stprintf(_tcschr(szLine, 0), __TEXT("%5u  "),
        pVMQ->dwRgnBlocks);
      GetProtectText(pVMQ->dwRgnProtection,
        _tcschr(szLine, 0), FALSE);
    }

    _tcscat(szLine, __TEXT("     "));

    // Try to obtain the module pathname for this region.
    nLen = _tcslen(szLine);
    if (pVMQ->pvRgnBaseAddress != NULL)
      GetModuleFileName((HINSTANCE) pVMQ->pvRgnBaseAddress,
        szLine + nLen, nMaxLen - nLen);

    if (pVMQ->pvRgnBaseAddress == GetProcessHeap()) {
        _tcscat(szLine, __TEXT("Default Process Heap"));
    }

    if (pVMQ->fRgnIsAStack) {
      _tcscat(szLine, __TEXT("Thread Stack"));
    }
  }


  ////////////////////////////////////////////////////////////

  void ConstructBlkInfoLine (PVMQUERY pVMQ,
```

```
  LPTSTR szLine, int nMaxLen) {

  _stprintf(szLine, __TEXT("   %08X  %s  %10u        "),
    pVMQ->pvBlkBaseAddress,
    GetMemStorageText(pVMQ->dwBlkStorage),
    pVMQ->dwBlkSize);

  if (pVMQ->dwBlkStorage != MEM_FREE) {
    GetProtectText(pVMQ->dwBlkProtection,
      _tcschr(szLine, 0), TRUE);
  }
}
```

(continued)

Figure 5-4. *continued*

```
////////////////////////////////////////////////////////////////


void Dlg_OnSize (HWND hwnd, UINT state, int cx, int cy) {
  SetWindowPos(GetDlgItem(hwnd, IDC_LISTBOX), NULL, 0, 0,
    cx, cy, SWP_NOZORDER);
}

////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog (HWND hwnd, HWND hwndFocus,
  LPARAM lParam) {

  HWND hWndLB = GetDlgItem(hwnd, IDC_LISTBOX);
  PVOID pvAddress = 0x00000000;
  TCHAR szLine[200];
  RECT rc;
  DWORD dwBlock;
  VMQUERY VMQ;
  BOOL fOk = TRUE;

   // Associate an icon with the dialog box.
  SetClassLong(hwnd, GCL_HICON, (LONG)
    LoadIcon((HINSTANCE) GetWindowLong(hwnd, GWL_HINSTANCE),
    __TEXT("VMMap")));

  // Make a horizontal scroll bar appear in the list box.
  ListBox_SetHorizontalExtent(hWndLB,
    150 * LOWORD(GetDialogBaseUnits()));

  // The list box must be sized first because the system
  // doesn't send a WM_SIZE message to the dialog box when
  // it's first created.
  GetClientRect(hwnd, &rc);
  SetWindowPos(hWndLB, NULL, 0, 0, rc.right, rc.bottom,
    SWP_NOZORDER);


  // Walk the virtual address space, adding
  // entries to the list box.
```

```
  while (fOk) {
    fOk = VMQuery(pvAddress, &VMQ);
```

(continued)

Figure 5-4. *continued*

```
      if (fOk) {
        // Construct the line to be displayed, and
        // add it to the list box.
        ConstructRgnInfoLine(&VMQ, szLine, sizeof(szLine));
        ListBox_AddString(hWndLB, szLine);

#if 1
        // Change the 1 above to a 0 if you do not want
        // to see the blocks contained within the region.

        for (dwBlock = 0; fOk && (dwBlock < VMQ.dwRgnBlocks);
          dwBlock++) {

          ConstructBlkInfoLine(&VMQ, szLine, sizeof(szLine));
          ListBox_AddString(hWndLB, szLine);

          // Get the address of the next region to test.
          pvAddress = ((BYTE *) pvAddress + VMQ.dwBlkSize);
          if (dwBlock < VMQ.dwRgnBlocks - 1) {
            // Don't query the memory info after
            // the last block.
            fOk = VMQuery(pvAddress, &VMQ);
          }
        }
#endif

        // Get the address of the next region to test.
        pvAddress = ((BYTE *) VMQ.pvRgnBaseAddress +
          VMQ.dwRgnSize);
      }
    }

#if COPYTOCLIPBOARD
    CopyControlToClipboard(hWndLB);
#endif
    return(TRUE);
}


////////////////////////////////////////////////////////////


void Dlg_OnCommand (HWND hwnd, int id, HWND hwndCtl,
  UINT codeNotify) {
```

(continued)

Figure 5-4. *continued*

```
  switch (id) {
    case IDCANCEL:
      EndDialog(hwnd, id);
```

```
      break;
  }
}

//////////////////////////////////////////////////////////


BOOL CALLBACK Dlg_Proc (HWND hDlg, UINT uMsg,
  WPARAM wParam, LPARAM lParam) {

  BOOL fProcessed = TRUE;
  switch (uMsg) {
    HANDLE_MSG(hDlg, WM_INITDIALOG, Dlg_OnInitDialog);
    HANDLE_MSG(hDlg, WM_COMMAND, Dlg_OnCommand);
    HANDLE_MSG(hDlg, WM_SIZE, Dlg_OnSize);

    default:
      fProcessed = FALSE;
      break;
  }
  return(fProcessed);
}


//////////////////////////////////////////////////////////


int WINAPI WinMain (HINSTANCE hinstExe,
  HINSTANCE hinstPrev, LPSTR lpszCmdLine, int nCmdShow) {

  DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMMAP),
    NULL, Dlg_Proc);
  return(0);
}


////////////////////////// End Of File //////////////////////////
```

(continued)

Figure 5-4. *continued*

VMMAP.RC

```
//Microsoft Visual C++ generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
```

```
#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
   "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
   "#include ""afxres.h""\r\n"
   "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
   "\r\n"
   "\0"
END

//////////////////////////////////////////////////////////////
#endif    // APSTUDIO_INVOKED
```

(continued)

Figure 5-4. *continued*

```
//////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_VMMAP DIALOG DISCARDABLE  10, 18, 250, 250
STYLE WS_MINIMIZEBOX œ WS_MAXIMIZEBOX œ WS_POPUP œ WS_VISIBLE
   œ WS_CAPTION œ WS_SYSMENU œ WS_THICKFRAME
CAPTION "Virtual Memory Map"
FONT 8, "Courier"
BEGIN
   LISTBOX        IDC_LISTBOX,0,0,0,0,NOT LBS_NOTIFY
                  œ LBS_NOINTEGRALHEIGHT œ NOT WS_BORDER
                  œ WS_VSCROLL œ WS_HSCROLL œ WS_GROUP
                  œ WS_TABSTOP
END


//////////////////////////////////////////////////////////////
//
// Icon
//

VMMap                ICON    DISCARDABLE     "VMMap.Ico"

#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
//
```

// Generated from the TEXTINCLUDE 3 resource.
//

///
#endif   // not APSTUDIO_INVOKED