

Windows NT and 16 Bit Programs

Matt Pietrek

Note: Several of the figures mentioned are located at the end of this article.

I love running 32-bit programs on my fire-breathing Windows NT® barn burner. Conversely, I hate running 16-bit programs. Nonetheless, because of the astute foresight of my bank, I'm stuck running a cranky 16-bit program if I want to bank online. Personally, I think that anyone still running Windows® 3.1 is probably not going to try online banking.

In any event, judging from my email, I'm not alone in my predicament. I receive a number of requests from people who want to know about the inner workings of 16-bit applications running under Windows NT. With this in mind, I'll devote this month's column to describing the Windows NT support for monitoring 16-bit programs. I'll start with a brief overview of how Windows NT runs programs for 16-bit Windows and MS-DOS®.

Under Windows NT 3.5 and earlier, every MS-DOS and 16-bit Windows-based program ran in its own virtual machine (VM). To these programs, each VM appeared as a relatively complete Windows 3.1-based system, down to having its own Local Descriptor Table (LDT). The activities of each 16-bit program were completely separate from those of other 16-bit programs. As a result, a 16-bit program could blow up with no ill effect on any other program.

Starting with version 3.51, Windows NT acquired the ability to run multiple 16-bit programs in the same VM. These programs shared the same address space and LDT, much like Windows 3.1. The good thing about this capability is that once a 16-bit program loaded, subsequent 16-bit programs would load faster and use less memory by using the already created VM. The downside to sharing VMs is that if one 16-bit program choked, the entire VM would go away (again, much like Windows 3.1). Luckily, the capability of running a badly behaving application in its own VM is still present. In the Run dialog, the Run in Separate Memory Space checkbox lets you specify that a separate VM should be used for the selected program, rather than running it in the communal system VM.

The focal point of VM capability in Windows NT is NTVDM.EXE from the SYSTEM32 directory. Each running instance of NTVDM.EXE constitutes a separate VM. NTVDM.EXE is a Win32® program. However, it uses its separate address space to create a VM that runs alongside regular Win32 processes. In other words, programs for MS-DOS and 16-bit Windows are like children belonging to the adult NTVDM process. NTVDM is just another adult Win32 process, with essentially the same rights and privileges as any other Win32-based program. This architecture even has its own name, Windows On Windows (WOW).

To briefly digress from Windows NT, Windows 95 has quite a different method for mixing MS-DOS, 16-bit Windows, and Win32 apps. Each MS-DOS-based program has its own VM. However, 16-bit Windows-based applications share the same VM and LDT, and have no address space protection from one another.

Under Windows 95, when a Win32 process has the CPU, that process can see the memory of the VM containing all the 16-bit programs. Likewise, 16-bit code running in the context of a 32-bit process (via a thunk) can see the 32-bit code and data of that process. At least the memory of other 32-bit programs isn't visible. This looseness of the Windows 95 address space is required for the flexibility of Windows 95 thunking, as well as to reduce excess thread switching and memory copying.

This concludes the five-cent tour of MS-DOS and 16-bit Windows-based programs running under Windows NT and Windows 95. Now, it's on to the facilities that Windows NT has for snooping around in this environment. The word you want to remember here is VDMDBG (that is, VDM Debug). Nearly

Windows NT and 16 Bit Programs

Matt Pietrek

all of the VDM debugging support is in a DLL named VDMDBG.DLL, which resides in the SYSTEM32 directory.

If you've never heard of VDMDBG.DLL, don't feel bad. Microsoft doesn't mention it in any of the online documentation that I searched. Nonetheless, VDMDBG.H and VDMDBG.LIB come with Visual C++® and the Platform SDK (at least as of the January 1998 edition). In the Platform SDK, there's a .HLP file from September 1994 in the \MSSDK\DOC\MISC directory. Alas, I couldn't find VDMDBG.HLP on my Visual Studio™ 97 CDs.

Using VDMDBG.DLL, you could write a 32-bit debugger program that runs on Windows NT and debugs 16-bit programs. However, VDMDBG.DLL isn't solely for debugger writers. Programmers often wonder how the Windows NT Task Manager (TASKMGR.EXE) is able to display the running 16-bit Windows tasks in the Processes tab. The answer is VDMDBG.DLL, which TASKMGR.EXE links to implicitly. If you look closely, the 16-bit Windows tasks you see in the Task Manager are always subordinate to an NTVDM process instance.

The VDMDBG API

As an experienced debugger writer, I felt like a party to a mugging when I first read VDMDBG.HLP. The introductory text is not for the faint of heart. It took many rereadings before the document made sense. Having undergone the steep learning curve, I can save you some time.

The VDMDBG API can be broken up into two sections. The first part is the simple stuff. The much larger remaining set of APIs is for hardcore programmers. If you're just interested in enumerating through 16-bit Windows tasks, skip to sections 5.8 and 5.9 of VDMDBG.HLP. There, you'll find an understandable description of VDMEnumProcessWOW and VDMEnumTaskWOW.

VDMEnumProcessWOW enumerates all of the VDMs in the system (that is, each instance of NTVDM.EXE). For each VDM, the API invokes a callback function that you define. The primary parameter to the callback function is the process ID for the particular instance of NTVDM.EXE. In addition, the callback receives an attribute DWORD and whatever user-supplied value was given to VDMEnumProcessWOW. The attribute parameter has only one flag value defined, WOW_SYSTEM. This flag is set for the instance of NTVDM.EXE that future 16-bit Windows-based programs will run in if a separate address space wasn't requested. Unlike most Windows enumerations, you return FALSE from the callback function to continue the enumeration.

Digging down a level, you come to VDMEnumTaskWOW and VDMEnumTaskWOWEx. VDMEnumTaskWOWEx isn't explained in VDMDBG.HLP, but VDMDBG.H provides a comment that describes the additional capabilities. Both APIs take a process ID as input and call another user-defined callback function. For VDMEnumTaskWOW, the callback function receives the thread ID of a 16-bit task running in the VDM, plus the 16-bit HMODULE, the 16-bit HTASK, and the user-supplied DWORD. The VDMEnumTaskWOWEx callback receives all of these parameters and adds a pointer to the file name and module name of the EXE for the 16-bit task. If the relationship between a Win32 thread ID and a 16-bit Windows task isn't clear yet, don't fret. I'll come back to this topic later.

With these simple enumerations, you can easily get at the same information that TASKMGR.EXE uses. From here on out, though, the remainder of VDMDBG's API is a tougher climb. Sure, there are functions that look like they might be used easily at first glance. For instance, VDMGlobalFirst and VDMGlobalNext are nearly dead ringers for the 16-bit GlobalFirst and GlobalNext APIs in TOOLHELP.DLL. Ditto for VDMModuleFirst and VDMModuleNext, which are like TOOLHELP's ModuleFirst and ModuleNext APIs.

The distinctions between these 32-bit APIs and their 16-bit cousins are threefold. For starters, the 32-bit APIs require that the WOWDEB.EXE task runs in the target debuggee's VM. (You will find

Windows NT and 16 Bit Programs

Matt Pietrek

WOWDEB.EXE in your SYSTEM32 directory.) How do you load a task in a particular NTVDM session? Refer to the VDMStartTaskInWOW API, which does not seem to be documented except in a comment from VDMDBG.H.

The second issue is that these APIs require a thread handle. Where can you get a thread handle? There's an OpenProcess API that returns process handles, but there's no corresponding OpenThread API. To my knowledge, there are only two ways to get an original thread handle: to be the creator of the thread, or to be passed a thread handle as part of a debugger process. Since NTVDM.EXE creates the threads for each 16-bit Windows task, the first option is out.

The second option for getting a thread handle is the debugger approach. When acting as a debugger process, the debugger receives notification of each new thread, along with a new thread handle. You could call DebugActiveProcess and attach yourself as a debugger to a running instance of NTVDM.EXE. This isn't necessarily a good idea, though. Once you begin acting as a debugger for another process, you can't detach yourself. If your process terminates, so will the debuggee. In the case of NTVDM.EXE, that could mean any number of 16-bit Windows tasks simply vanishing because your process has terminated.

Speaking of debuggers, this segues nicely into the third hassle with APIs like VDMModuleFirst and VDMGlobalNext. Because they will cause 16-bit system code to execute in KRNL386.EXE, events that a debugger would want to see may be triggered. For this reason, these APIs take a callback address that will be invoked if a debug-related event occurs. Just to enumerate modules or selectors, you're suddenly expected to handle Win32 debug events. Perhaps it's called VDMDBG.DLL for a reason!

I'm not going to describe all the remaining APIs that VDMDBG.DLL provides. It's enough to say that the APIs provide the basic functionality required for a 32-bit program to perform surgery in a 16-bit Windows environment. However, something that's worth spending more time on is how 16-bit system events percolate up to a 32-bit process.

If you've used NotifyRegister and InterruptRegister from the 16-bit TOOLHELP.DLL, you may recall events such as GP faults, module loads, segment frees, tasks starting, and so on. Using VDMDBG.DLL, a Win32 process can see those same events, albeit in a different manner. Figure 1 shows the VDMDBG.DLL events and their 16-bit TOOLHELP.DLL equivalents (where applicable). Alas, I haven't been able to generate all of the listed events in my informal testing.

Let's say you wanted to monitor 16-bit Windows events from a Win32 program. How would you go about doing it? The answer is (not surprisingly) to act as a debugger for the NTVDM process. Before reading ahead, you should be at least somewhat familiar with the Win32 debug architecture and debug loops. See the Win32 documentation and John Robbins' article, "Multiple Threads in Visual Basic 5.0, Part II: Writing a Win32 Debugger" (MSJ, September 1997) if you need assistance with this topic.

VDMDBG.DLL dedicates a special exception code (STATUS_VDM_EVENT) to communicate 16-bit events to a 32-bit debugger. When the 32-bit debugger sees an exception code of STATUS_VDM_EVENT, it can determine the 16-bit event type and other information by reading the exception arguments. Section 4 of VDMDBG.HLP describes most of the 16-bit event types. VDMDBG.H includes a set of macros (W1, W2, and so on) that help crack apart the exception arguments into the desired fields.

Let's look at an example to clarify this point. The situation is as follows: your process is acting as a Win32 debugger on an instance of NTVDM.EXE. The program is dutifully processing debug events

Windows NT and 16 Bit Programs

Matt Pietrek

with a WaitForDebugEvent/ContinueDebugEvent loop. The user starts a new 16-bit program, which the system runs in the VDM you're monitoring. As a result, a DBG_TASKSTART is generated. To the 32-bit debugger, this appears as an exception with a code of STATUS_VDM_EVENT. The W1 field of the ExceptionInformation is 10 (DBG_TASKSTART).

Digging deeper into the example, the VDMDBG documentation states that the W3 field contains a pointer to an IMAGE_NOTE structure. The IMAGE_NOTE structure contains a module name (for instance, NUKEDLL), the associated file name (for instance, C:\COLUMNS\C0L8\ NUKEDLL.EXE), the HMODULE, and the HTASK.

When I first wrote code to try this, I blindly cast the DW3 field to an IMAGE_NOTE pointer, and was baffled when my code faulted. It turns out that the IMAGE_NOTE structure is located in the debuggee process. No problem; just use ReadProcessMemory to suck the information over to the debugger process. The same goes for the SEGMENT_NOTE structure, which is used by several other 16-bit events.

Experimenting with VDMDBG.DLL

To show some of the capabilities of VDMDBG.DLL, I wrote the VDMDBGDemo program, shown in Figure 2. When you first start VDMDBGDemo, the action is in the bottom tree view control. If you have any NTVDM sessions running, they'll show up in the tree view. Otherwise, you can start a 16-bit Windows-based app and hit the refresh button, which updates the tree view contents with current information. If you do nothing at all, the tree view refreshes every 10 seconds via a WM_TIMER message.

The top-level contents of the tree view are filled in by the results of using VDMEnumProcessWOW. In VDMDBGDemo.CPP, this occurs in the PopulateTree and VDMProcessEnumProc functions (see Figure 3). As the callback encounters each NTVDM session, it in turn calls VDMEnumTaskWOWEx. This callback function (VDMTaskEnumProc) adds the second-level tree view nodes that describe each 16-bit Windows process running in the VDM.

When you first run VDMDBGDemo, the upper listbox will remain conspicuously empty, unlike the one shown in Figure 2. This listbox shows the 16-bit events that have occurred in one of the NTVDM sessions. How do you see these events? As I mentioned, a Win32 debug loop is necessary to receive the exceptions that convey 16-bit event information. How are event strings added to the listbox? In the dialog procedure (VDMDBGDemoDlgProc), I use a custom-defined window message (WM_LB_ADDITEM). Why didn't I just use LB_ADDSTRING directly? The reasons will become clear shortly.

Figure 4 contains all of the code for a Win32 debug loop that monitors an NTVDM process. To start a thread that monitors one of the NTVDM sessions, highlight the session (as shown in Figure 2), then click the Attach button. This causes VDMDBGDemo to start a new thread to monitor the selected session. The entry point to this thread's code is the VDMDebugThreadFunc function. Near the very beginning of the function, it calls DebugActiveProcess on the selected NTVDM process. If the call succeeds, the code enters into a WaitForDebugEvent/ContinueDebugEvent loop. This loop will continue indefinitely or until the NTVDM session terminates.

Windows NT and 16 Bit Programs

Matt Pietrek

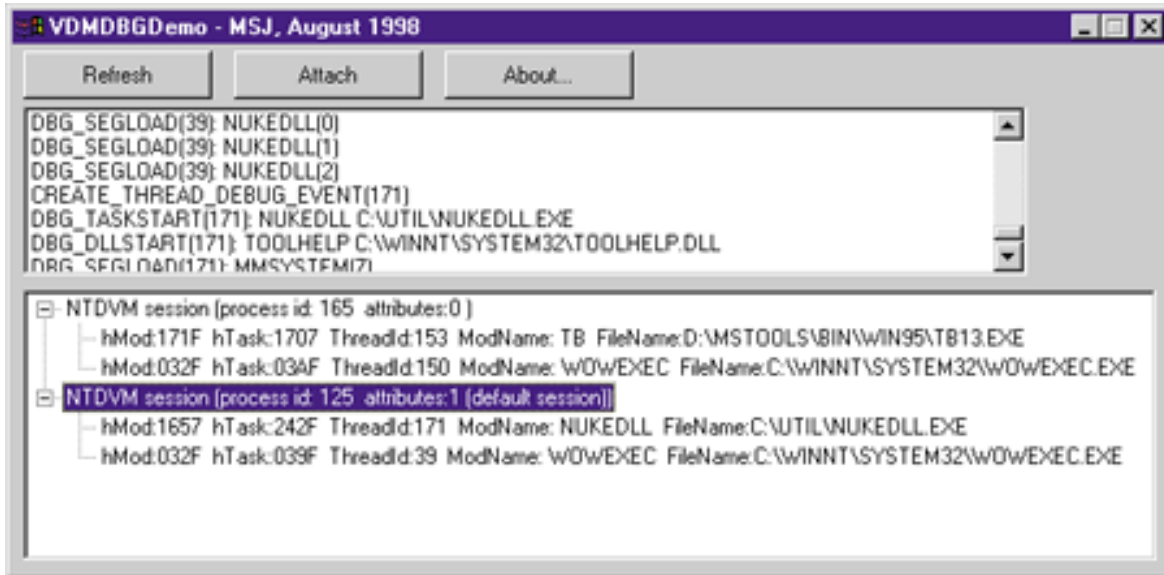


Figure 2 VDMDBGDemo

As each Win32 debug event comes in, the event is passed off to the `HandleDebugEvent` function. This function either emits basic information about the event, or passes the event off to a more specialized function such as `HandleExceptionDebugEvent`. If the event is passed off, the receiving function may close the process, thread, and file handles that the debugger process receives automatically as an added bonus.

Let's look at the `HandleExceptionDebugEvent` function since that's where all of the information about 16-bit Windows-based events eventually winds up. The first thing the code does is check if the exception number is `STATUS_VDM_EVENT`. If it is, the event is passed to `VDMProcessException`, which the online help says is necessary to call for `STATUS_VDM_EVENT` exceptions. Next, the code uses the `W1` macro from `VDMDBG.H` to break out what type of 16-bit event occurred.

A big switch statement further cracks open the remaining event fields to show more information about the particular event. When examining the events, it's important to realize that you won't see 16-bit Windows events that occurred prior to attaching to the NTVDM session. Looking at the switch statement that cracks apart 16-bit Windows events, you'll see that full processing of some 16-bit Windows events requires reading structures from the debuggee's memory (that is, from the NTVDM address space). If the exception isn't a `STATUS_VDM_EVENT`, it's a regular exception that any Win32 process could see. The `HandleExceptionDebugEvent` code just displays some rudimentary information for it. An example of such an exception is the `DebugBreak` (`INT 3`) that all debuggee processes generate when debugging begins.

If you look throughout `VDMDBGDemoDbgLoop.CPP`, you'll see that information is added to the listbox via the `_lbPrintf` function at the bottom of the source file. `_lbPrintf` first acts like `sprintf`, but then calls `strdup` for the resultant string. Last, `_lbPrintf` posts the user-defined message (`WM_LB_ADDITEM`) to the main dialog procedure, with the `LPARAM`, pointing to the just-allocated string buffer.

In `_lbPrintf`, why didn't I just call `SendDlgItemMessage` with `LB_ADDSTRING`? Consider this: I'm not dealing with just a single thread here. Two threads then? Nope, I'm not so fortunate. At a minimum, there are three threads involved: the main `VDMDBGDemo` thread that drives the display dialog, the debug loop thread, and all the threads in the NTVDM process. Why is NTVDM involved in this? When

Windows NT and 16 Bit Programs

Matt Pietrek

WaitForDebugEvent returns, the debuggee process (in this case NTVDM) is completely frozen until you call ContinueDebugEvent.

Since SendMessage and SendDlgItemMessage are quite sensitive to thread synchronization issues, I decided to avoid the possibility of a deadlock altogether. By posting rather than sending the WM_LB_ADDITEM messages, the messages are queued until it's safe to add items to the listbox. Another reason why I went this route is that the user may select multiple NTVDM sessions to monitor, compounding the potential for deadlock scenarios even further.

Sprinkled throughout NTVDMDbgDemoDbgLoop.CPP are references to PSAPI.DLL and the GetModuleFileNameExW function. What's that for? One of the notifications that a debugger gets is for module loads. Unfortunately, it receives an HMODULE but no module name. Within your own process, you could call GetModuleHandle to translate the HMODULE to a meaningful file name. However, since I'm a debugger, the HMODULE is relative to another process. With a thorough understanding of Portable Executable files, you could craft some code to read the module name from the debuggee process. Being lazy, and not wanting to reinvent the wheel, I used the GetModuleFileNameExW function in PSAPI.DLL instead. I described this API in my August 1996 column. Since not everybody may have PSAPI.DLL on their system, I used LoadLibrary and GetProcAddress to avoid linking implicitly to PSAPI.DLL.

Some Observations

Before writing VDMDBGDemo, I had a vague idea of what NTVDM.EXE was and how non-Win32-based programs ran in Windows NT. After running VDMDBGDemo many dozens of times, I've learned quite a few more details.

For starters, MS-DOS-based programs seem to always run in separate NTVDM sessions. I was never able to get an MS-DOS-based program to run in the same session as a 16-bit Windows-based program. Nor was I able to get two independently started MS-DOS-based programs to run in the same NTVDM session. In fact, NTVDM sessions running MS-DOS programs don't show up in VDMEnumProcessWOW enumerations.

In the events listbox, you'll see that for each event, I included the thread ID that the event occurred in. (Look for the number in parenthesis following the event name.) Through experimentation, I learned that (as expected) each NTVDM session has an initial main thread that hangs around for the whole life of the session. Next, each 16-bit Windows task started in the session gets its own thread. In fact, the main NTVDM thread is associated with a 16-bit Windows task called WOWEXEC.EXE. A third type of thread executes the initial DebugBreak (INT 3) that occurs when a process is run under a debugger.

I was especially intrigued by thread usage when a new task starts in an NTVDM session. All of the initial DLL and segment loading is done by the primary NTVDM thread; namely, the thread that runs the WOWEXEC.EXE task. Only after that code successfully completes does NTVDM create a new thread to act as the host for the new task. Once the new thread is created for the task, all further events related to a task occur in the thread assigned to it. Incidentally, running each 16-bit Windows task as its own thread isn't unique to Windows NT; Windows 95 does this as well.

Wrap-up

For a long time, VDMDBG.DLL has lived its life in obscurity. The fact that its APIs, header files, and online help hadn't been updated for years didn't help the cause. However, when I popped in the January 1998 Platform SDK, I saw that several new APIs such as VDMGetDbgFlags and VDMIsModuleLoaded had been added to VDMDBG.H. It seems that there's finally new life being breathed into this API.

Windows NT and 16 Bit Programs

Matt Pietrek

Mind you, when I looked at my Windows NT 4.0 Service Pack 3 installation, I didn't find those APIs in its VDMDBG.DLL. I then checked my Windows NT 5.0 beta installation and found that its VDMDBG.DLL contained those APIs. Let's hope this means that VDMDBG.DLL will get the full Win32 API treatment someday soon. It's happened before with other obscure but useful system DLLs.

Figure 1 VDMDBG and TOOLHELP Events

VDMDBG Event	TOOLHELP Equivalent
DBG_SEGLOAD	NFY_LOADSEG
DBG_SEGMOVE	
DBG_SEGFREE	NFY_FREESEG
DBG_MODLOAD	
DBG_MODFREE	NFY_DELMODULE
DBG_SINGLESTEP	INT_1
DBG_BREAK	INT_3
DBG_GPFALT	INT_GPFALT
DBG_DIVOVERFLOW	INT_DIVO
DBG_INSTRFAULT	INT_UDINSTR
DBG_TASKSTART	NFY_STARTTASK
DBG_TASKSTOP	NFY_EXITTASK
DBG_DLLSTART	NFY_STARTDLL
DBG_DLLSTOP	
DBG_ATTACH	
DBG_TOOLHELP	
DBG_STACKFAULT	INT_STKFAULT
DBG_WOWINIT	
DBG_TEMPBP	
DBG_MODMOVE	
DBG_INIT	
DBG_GPFALT2	

Figure 3 VDMDBGDemo.CPP

```
//=====
// VDMDBGDemo - Matt Pietrek 1998
// Microsoft Systems Journal, August 1998
// FILE: VDMDBGDemo.CPP
//=====
#include <windows.h>
#include <COMMCTRL.H>
#include <stdio.h>
#include <process.h>
#include <tchar.h>
#include <vdmdbg.h>
#pragma hdrstop
#include "VDMDBGDemo.h"
#include "VDMDBGDemoDbgLoop.h"

// Helper function prototypes
void Handle_WM_INITDIALOG(HWND hDlg);
void Handle_WM_COMMAND(HWND hWndDlg, WPARAM wParam, LPARAM lParam );
void Handle_WM_CLOSE( HWND hDlg );
void Handle_WM_SIZE(HWND hWndDlg, WPARAM wParam, LPARAM lParam );
void Handle_WM_NOTIFY( HWND hDlg, WPARAM wParam, LPARAM lParam );
BOOL CALLBACK VDMDBGDemoDlgProc(HWND,UINT,WPARAM,LPARAM);
void GetSetPositionInfoFromRegistry( BOOL fSave, POINT *lppt );
```

Windows NT and 16 Bit Programs

Matt Pietrek

```
void    PopulateTree( HWND hWndTree );

HTREEITEM AddTreeViewSubItem(   HWND hWndTree, HTREEITEM hTreeItem,
                               LPTSTR pszItemText, BOOL fItemData = FALSE,
                               LPARAM itemData = 0 );

TCHAR gszRegistryKey[] = _T("Software\\WheatyProductions\\VDMDBGDemo");

TCHAR gszMBTitle[] = _T( "VDMDBGDemo, by Matt Pietrek - MSJ August 1998" );

TCHAR gszAboutText[] =
    _T("VDMDBGDemo displays information and events in NTVDM processes\r\n\r\n")
    _T("To monitor events in a NTVDM session, highlight the desired session, ")
    _T("then click the Attach button");

TCHAR gszCloseWarning[] =
    _T("There is at least one NTVDM session being monitored.\r\n\r\nIf you ")
    _T("exit, these sessions will be terminated\r\n\r\nExit anyway?");

// ===== Start of code =====
HWND g_hWndTree = 0;
HWND g_hDlg = 0;
DWORD g_cAttachedProcesses = 0;

BOOL
WINAPI
VDMTaskEnumProc(   DWORD dwThreadId, WORD hMod16, WORD hTask16,
                  PSZ pszModName, PSZ pszFileName, LPARAM lpUserDefined )
{
    TCHAR szTaskInfo[512];

    wsprintf(szTaskInfo,
        _T("hMod:%04X hTask:%04X ThreadId:%u ModName: %hs FileName:%hs"),
        hMod16, hTask16, dwThreadId, pszModName, pszFileName );

    HTREEITEM hTreeItem = AddTreeViewSubItem(g_hWndTree, (HTREEITEM)lpUserDefined,
        szTaskInfo );

    return FALSE;
}

BOOL WINAPI
VDMProcessEnumProc(   DWORD dwProcessId, DWORD dwAttributes, LPARAM lpUserDefined )
{
    TCHAR szVDMInfo[256];

    wsprintf( szVDMInfo, _T("NTVDM session (process id: %u attributes:%X %s)"),
        dwProcessId, dwAttributes,
        dwAttributes & WOW_SYSTEM ? _T("(default session)") : _T("") );

    HTREEITEM hTreeItem = AddTreeViewSubItem( g_hWndTree, NULL, szVDMInfo,
        TRUE, dwProcessId );

    VDMEnumTaskWOWEx( dwProcessId, VDMTaskEnumProc, (LPARAM)hTreeItem );

    TreeView_Expand( g_hWndTree, hTreeItem, TVE_EXPAND );

    // _beginthread( VDMDebugThreadFunc, 0, (void *)dwProcessId );

    return FALSE;
}

int WINAPI WinMain(   HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow )
{
    InitCommonControls();

    // Bring up the user interface
    DialogBox( hInstance, MAKEINTRESOURCE(IDD_VDMDBGDEMO),
        0, (DLGPROC)VDMDBGDemoDlgProc );
    GetLastError();
    return 0;
}

BOOL CALLBACK VDMDBGDemoDlgProc(   HWND hDlg,UINT msg,WPARAM wParam, LPARAM lParam )
{
    // The dialog procedure for the main window
    switch ( msg )
```

Windows NT and 16 Bit Programs

Matt Pietrek

```
{
    case WM_INITDIALOG:
        Handle_WM_INITDIALOG( hDlg ); return TRUE;
    case WM_CLOSE:
        Handle_WM_CLOSE( hDlg ); break;
    case WM_SIZE:
        Handle_WM_SIZE( hDlg, wParam, lParam ); break;
    case WM_COMMAND:
        Handle_WM_COMMAND( hDlg, wParam, lParam ); break;
    case WM_NOTIFY:
        Handle_WM_NOTIFY( hDlg, wParam, lParam ); break;
    case WM_TIMER:
        PopulateTree( g_hWndTree ); break;
    case WM_LB_ADDITEM:
        {
            TCHAR * pwszItem = (TCHAR *)lParam;
            SendDlgItemMessage(hDlg, IDC_LIST1, LB_ADDSTRING, 0,
                (LPARAM)pwszItem);
            delete []pwszItem;
        }
        break;
    // let everything else fall through
}
return FALSE;
}

//=====
// Walk through the list of objects, adding each object name to the root of
// the treeview

void PopulateTree( HWND hWndTree )
{
    TreeView_DeleteAllItems( hWndTree );

    VDMenuProcessWOW( VDMenuProcessEnumProc, (LPARAM)hWndTree );
}

void Handle_WM_INITDIALOG(HWND hDlg)
{
    // Get the window coordinates where the program was last running,
    // and move the window to that spot.
    POINT pt;
    GetSetPositionInfoFromRegistry( FALSE, &pt );

    SetWindowPos(hDlg, 0, pt.x, pt.y, 0, 0,
        SWP_NOSIZE | SWP_NOREDRAW | SWP_NOZORDER | SWP_NOACTIVATE);

    g_hDlg = hDlg;
    g_hWndTree = GetDlgItem(hDlg, IDC_TREE1);

    PopulateTree( g_hWndTree );

    SetTimer( hDlg, 0, 10000, 0 ); // Make a 2 second timer
}

void Handle_WM_COMMAND(HWND hWndDlg, WPARAM wParam, LPARAM lParam )
{
    WORD wNotifyCode = HIWORD(wParam); // notification code
    WORD wID = LOWORD(wParam); // item, control, or accelerator id
    HWND hwndCtl = (HWND) lParam; // handle of control

    if ( IDC_BUTTON_REFRESH == wID )
    {
        if ( BN_CLICKED == wNotifyCode )
            PopulateTree( g_hWndTree );
    }
    else if ( IDC_BUTTON_ATTACH == wID )
    {
        HTREEITEM hTreeItem = TreeView_GetSelection( g_hWndTree );

        TVITEM tvi;
        tvi.hItem = hTreeItem;
        tvi.mask = TVIF_PARAM;
        tvi.lParam = 0;

        // Get the process ID for the session that we stashed away earlier
        if ( TreeView_GetItem( g_hWndTree, &tvi ) && tvi.lParam )
        {

```

Windows NT and 16 Bit Programs

Matt Pietrek

```
        // start a new thread to act as the debug loop
        _beginthread( VDMDebugThreadFunc, 0, (void *)tvi.lParam );
    }
    else // User didn't pick a valid line
    {
        MessageBox( hWndDlg, _T("Please select an NTVDM Session line"),
            0, MB_OK );
    }
}
else if ( IDC_BUTTON_ABOUT == wID )
{
    if ( BN_CLICKED == wNotifyCode )
        MessageBox( hWndDlg, gszAboutText, gszMBTitle, MB_OK );
}
}

void Handle_WM_NOTIFY( HWND hDlg, WPARAM wParam, LPARAM lParam )
{
    if ( wParam != IDC_TREE1 )
        return;

    LPNMHDR pnmh = (LPNMHDR)lParam;

    if ( NM_DBLCLK != pnmh->code )
        return;

    HTREEITEM hTreeItem = TreeView_GetSelection( g_hWndTree );

    TVITEM tvi;
    tvi.hItem = hTreeItem;
    tvi.mask = TVIF_PARAM;
    if ( TreeView_GetItem( g_hWndTree, &tvi ) )
    {
        if ( tvi.lParam )
            _beginthread( VDMDebugThreadFunc, 0, (void *)tvi.lParam );
    }
}

void Handle_WM_CLOSE( HWND hDlg )
{
    // If g_cAttachedProcesses is non-zero at program exit time, we need to
    // warn the user that exiting will terminate any debug loops, and hence
    // make the 16 bit programs and the associated NTVDM sessions go away.
    if ( g_cAttachedProcesses )
    {
        if ( IDNO == MessageBox( hDlg, gszCloseWarning, gszMBTitle, MB_YESNO ) )
            return;
    }

    // Save off the window's X,Y coordinates for next time
    RECT rect;
    if ( GetWindowRect( hDlg, &rect ) )
        GetSetPositionInfoFromRegistry( TRUE, (LPPOINT)&rect );

    KillTimer( hDlg, 0 );

    EndDialog( hDlg, 0 );
}

void Handle_WM_SIZE( HWND hWndDlg, WPARAM wParam, LPARAM lParam )
{
    RECT tvRect, dlgRect;
    POINT pt;

    WORD nClientWidth = LOWORD( lParam );
    WORD nClientHeight = HIWORD( lParam );

    GetClientRect( hWndDlg, &dlgRect ); // Get size of dialog
    GetWindowRect( g_hWndTree, &tvRect ); // Get screen position of child

    pt.x = tvRect.left; // Get the X,Y coordinates for the top left
    pt.y = tvRect.top; // and reuse them in the resized client
    ScreenToClient( hWndDlg, &pt ); // Calculate screen X,Y of child window

    WORD tvWidth = nClientWidth - ( pt.x * 2 ); // Equal spacing on all borders
    WORD tvHeight = nClientHeight - (WORD)( pt.y + pt.x );
    MoveWindow( g_hWndTree, pt.x, pt.y, tvWidth, tvHeight, TRUE );
}
```

Windows NT and 16 Bit Programs

Matt Pietrek

```
}

void GetSetPositionInfoFromRegistry( BOOL fSave, POINT *lppt )
{
    // Function that saves or restores the coordinates of a dialog box
    // in the system registry.  Handles the case where there's nothing there.
    //
    HKEY hKey;
    DWORD dataSize, err, disposition;
    TCHAR szKeyName[] = _T("DlgCoordinates");

    if ( !fSave )                // In case the key's not there yet, we'll
        lppt->x = lppt->y = 0;    // return 0,0 for the coordinates

    // Open the registry key (or create it if the first time being used)
    err = RegCreateKeyEx( HKEY_CURRENT_USER, gszRegistryKey, 0, 0,
        REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS,
        0, &hKey, &disposition );
    if ( ERROR_SUCCESS != err )
        return;

    if ( fSave )                // Save out coordinates
    {
        RegSetValueEx(hKey,szKeyName, 0, REG_BINARY,(PBYTE)lppt,sizeof(*lppt));
    }
    else                        // read in coordinates
    {
        dataSize = sizeof(*lppt);
        RegQueryValueEx( hKey, szKeyName, 0, 0, (PBYTE)lppt, &dataSize );
    }

    RegCloseKey( hKey );
}

HTREEITEM AddTreeviewSubItem(   HWND hWndTree, HTREEITEM hTreeItem,
                                LPTSTR pszItemText, BOOL fItemData,
                                LPARAM itemData )
{
    TVINSERTSTRUCT tvi;

    tvi.hParent = hTreeItem;
    tvi.hInsertAfter = TVI_LAST;
    tvi.item.mask = TVIF_TEXT;
    tvi.item.pszText = pszItemText;
    tvi.item.cchTextMax = strlen( pszItemText );
    if ( fItemData )
    {
        tvi.item.mask |= TVIF_PARAM;
        tvi.item.lParam = itemData;
    }

    return TreeView_InsertItem( hWndTree, &tvi );
}
}
```

Figure 4 VDMDBGDemoDbgLoop

VDMDBGDemoDbgLoop.H

```
void VDMDebugThreadFunc( void * args );
```

VDMDBGDemoDbgLoop.CPP

```
//=====
// VDMDBGDemo - Matt Pietrek 1998
// Microsoft Systems Journal, August 1998
// FILE: VDMDBGDemoDbgLoop.CPP
//=====
#include <windows.h>
#include <stdio.h>
#include <vdmdbg.h>
#include <tchar.h>
#include <stdarg.h>
#include "VDMDBGDemo.h"
```

Windows NT and 16 Bit Programs

Matt Pietrek

```
#include "VDMDBGDemoDbgLoop.h"

//===== Prototypes =====
int _lbPrintf( LPTSTR format, ... );
void HookupToPSAPI( void );

DWORD WINAPI GetModuleFileNameExW( // Stolen from PSAPI.H, which not everybody
    HANDLE hProcess, // may have...
    HMODULE hModule,
    LPWSTR lpFilename,
    DWORD nSize );
typedef DWORD (WINAPI *PFNGETMODULEFILENAMEEXW)(HANDLE,HMODULE,LPWSTR,DWORD);

//===== Variables =====
PFNGETMODULEFILENAMEEXW pfnGetModuleFileNameExW = 0;

__declspec(thread) HANDLE tls_hProcess; // A per-thread variable

// Defined in VDMDBGDemo.CPP
extern HWND g_hDlg;
extern DWORD g_cAttachedProcesses;

//===== Code =====
void HandleExceptionDebugEvent( DEBUG_EVENT &de )
{
    EXCEPTION_RECORD & exrec = de.u.Exception.ExceptionRecord;

    DWORD dwExceptionCode = exrec.ExceptionCode;

    if ( STATUS_VDM_EVENT == dwExceptionCode )
    {
        VDMProcessException( &de );

        LPTSTR pszEventName = 0;

        switch ( Wl( exrec ) )
        {
            case DBG_SEGLOAD:
                {
                    pszEventName = _T("DBG_SEGLOAD");
                    SEGMENT_NOTE segNote;
                    DWORD cbRead = 0;
                    ReadProcessMemory( tls_hProcess, (PVOID)DW3(exrec),
                                        &segNote, sizeof(segNote), &cbRead );
                    if ( sizeof(SEGMENT_NOTE) != cbRead )
                        break;

                    _lbPrintf( _T("%s(%u): %hs(%u)"), pszEventName,
                                de.dwThreadId, segNote.Module, segNote.Segment);
                    return;
                }

            case DBG_TASKSTART: // These events can all be handled
            case DBG_DLLSTART: // in the same manner
            case DBG_DLLSTOP:
            case DBG_TASKSTOP:
                {
                    switch( Wl(exrec) )
                    {
                        case DBG_TASKSTART:
                            pszEventName = _T("DBG_TASKSTART"); break;
                        case DBG_DLLSTART:
                            pszEventName = _T("DBG_DLLSTART"); break;
                        case DBG_DLLSTOP:
                            pszEventName = _T("DBG_DLLSTOP"); break;
                        case DBG_TASKSTOP:
                            pszEventName = _T("DBG_TASKSTOP"); break;
                    }
                }

            IMAGE_NOTE imgNote;
            DWORD cbRead;
            ReadProcessMemory( tls_hProcess, (PVOID)DW3(exrec),
                                &imgNote, sizeof(imgNote), &cbRead );
            if ( sizeof(IMAGE_NOTE) != cbRead )
                break;

            _lbPrintf( _T("%s(%u): %hs %hs"), pszEventName,
                        de.dwThreadId, imgNote.Module, imgNote.FileName);
        }
    }
}
```

Windows NT and 16 Bit Programs

Matt Pietrek

```
        return;
    }

    case DBG_MODFREE:
    {
        pszEventName = _T("DBG_MODFREE");
        SEGMENT_NOTE segNote;
        DWORD cbRead;
        ReadProcessMemory( tls_hProcess, (PVOID)DW3(exrec),
            &segNote, sizeof(segNote), &cbRead );
        if ( sizeof(SEGMENT_NOTE) == cbRead )
        {
            _lbPrintf(_T("%s(%u): %hs"),
                pszEventName, de.dwThreadId, segNote.Module);
            return;
        }
        // else, fall through...
        break;
    }

    case DBG_SEGMOVE: pszEventName = _T("DBG_SEGMOVE"); break;
    case DBG_SEGFREE: pszEventName = _T("DBG_SEGFREE"); break;
    case DBG_MODLOAD: pszEventName = _T("DBG_MODLOAD"); break;
    case DBG_SINGLESTEP: pszEventName = _T("DBG_SINGLESTEP"); break;
    case DBG_BREAK: pszEventName = _T("DBG_BREAK"); break;
    case DBG_GPFALT: pszEventName = _T("DBG_GPFALT"); break;
    case DBG_DIVOVERFLOW: pszEventName = _T("DBG_DIVOVERFLOW"); break;
    case DBG_INSTRFAULT: pszEventName = _T("DBG_INSTRFAULT"); break;
    case DBG_ATTACH: pszEventName = _T("DBG_ATTACH"); break;
}

_lbPrintf( _T("%s(%u)"), pszEventName, de.dwThreadId );

return;
}

_lbPrintf(
    _T("EXCEPTION_DEBUG_EVENT(%u): Code:%08X Address:%08X %s chance"),
    de.dwThreadId, dwExceptionCode, exrec.ExceptionAddress,
    de.u.Exception.dwFirstChance ? _T("first") : _T("second") );
}

void HandleLoadDllDebugEvent( DEBUG_EVENT &de )
{
    LOAD_DLL_DEBUG_INFO & loadInfo = de.u.LoadDll;

    if ( pfnGetModuleFileNameExW )
    {
        TCHAR szModName[MAX_PATH];

        szModName[0] = 0;

        pfnGetModuleFileNameExW(tls_hProcess, (HINSTANCE)loadInfo.lpBaseOfDll,
            szModName, sizeof(szModName) );

        _lbPrintf( _T("LOAD_DLL_DEBUG_EVENT(%u): %08X %s"),
            de.dwThreadId, loadInfo.lpBaseOfDll, szModName );
    }
    else
        _lbPrintf( _T("LOAD_DLL_DEBUG_EVENT(%u): %08X"),
            de.dwThreadId, loadInfo.lpBaseOfDll );

    CloseHandle( loadInfo.hFile ); // Don't need this, so close it
}

void HandleCreateThreadDebugEvent( DEBUG_EVENT &de )
{
    CloseHandle( de.u.CreateThread.hThread ); // Don't need this, so close it

    _lbPrintf( _T("CREATE_THREAD_DEBUG_EVENT(%u)"), de.dwThreadId );
}

void HandleCreateProcessDebugEvent( DEBUG_EVENT &de )
{
    CREATE_PROCESS_DEBUG_INFO & cpdi = de.u.CreateProcessInfo;

    tls_hProcess = cpdi.hProcess;
}
```

Windows NT and 16 Bit Programs

Matt Pietrek

```
CloseHandle( cpdi.hFile );      // Don't need these, so
CloseHandle( cpdi.hThread );   // close it!

if ( pfnGetModuleFileNameExW )
{
    TCHAR szModName[MAX_PATH];

    szModName[0] = 0;

    pfnGetModuleFileNameExW( tls_hProcess, (HINSTANCE)cpdi.lpBaseOfImage,
        szModName, sizeof(szModName) );

    _lbPrintf( _T("CREATE_PROCESS_DEBUG_EVENT(%u): %08X %s"),
        de.dwThreadId, cpdi.lpBaseOfImage, szModName );
}
else
    _lbPrintf( _T("CREATE_PROCESS_DEBUG_EVENT: %08X"), cpdi.lpBaseOfImage );
}

void HandleExitThreadDebugEvent( DEBUG_EVENT &de )
{
    _lbPrintf( _T("EXIT_THREAD_DEBUG_EVENT(%u): exit code:%u"),
        de.dwThreadId, de.u.ExitThread.dwExitCode );
}

void HandleDebugEvent( DEBUG_EVENT &de )
{
    LPTSTR s;

    switch ( de.dwDebugEventCode )
    {
        case EXCEPTION_DEBUG_EVENT:
            HandleExceptionDebugEvent( de ); return;
        case LOAD_DLL_DEBUG_EVENT:
            HandleLoadDllDebugEvent( de ); return;
        case CREATE_THREAD_DEBUG_EVENT:
            HandleCreateThreadDebugEvent( de ); return;
        case CREATE_PROCESS_DEBUG_EVENT:
            HandleCreateProcessDebugEvent( de ); return;
        case EXIT_THREAD_DEBUG_EVENT:
            HandleExitThreadDebugEvent( de ); return;
        case EXIT_PROCESS_DEBUG_EVENT: s = _T("EXIT_PROCESS_DEBUG_EVENT"); break;
        case UNLOAD_DLL_DEBUG_EVENT: s = _T("UNLOAD_DLL_DEBUG_EVENT"); break;
        case OUTPUT_DEBUG_STRING_EVENT: s = _T("OUTPUT_DEBUG_STRING_EVENT");
            break;
        case RIP_EVENT: s = _T("RIP_EVENT"); break;
    }

    _lbPrintf( _T("%s(%u)"), s, de.dwThreadId );
}

void VDMDebugThreadFunc( void * args )
{
    DWORD dwProcessId = (DWORD)args;    // "args" is just the NTVDM process Id

    if ( !DebugActiveProcess( dwProcessId ) )
        return;

    // Increment the count describing how many debug loops we have.
    InterlockedIncrement( (LPLONG)&g_cAttachedProcesses );

    HookupToPSAPI();    // In case the user has PSAPI.DLL, we can use it to
                        // get a better description of debug events

    DEBUG_EVENT de;

    while ( WaitForDebugEvent( &de, INFINITE ) )
    {
        HandleDebugEvent( de );

        if ( EXIT_PROCESS_DEBUG_EVENT == de.dwDebugEventCode )
            break;

        ContinueDebugEvent( de.dwProcessId, de.dwThreadId, DBG_CONTINUE );
    }

    // Decrement the count describing how many debug loops we have.
}
```

Windows NT and 16 Bit Programs

Matt Pietrek

```
    InterlockedDecrement( (LPLONG)&g_cAttachedProcesses );
}

void HookupToPSAPI( void )
{
    HMODULE hModPSAPI;

    if ( !pfnGetModuleFileNameExW )
    {
        hModPSAPI = LoadLibrary( _T("PSAPI.DLL") );
        if ( hModPSAPI )
            pfnGetModuleFileNameExW = (PFNGETMODULEFILENAMEEXW)
                GetProcAddress( hModPSAPI, "GetModuleFileNameExW" );
    }
}

int _lbPrintf( LPTSTR format, ... )
{
    TCHAR szBuffer[1024];
    va_list argptr;

    va_start(argptr, format);
    int retValue = wvsprintf(szBuffer, format, argptr);
    va_end(argptr);

    LPTSTR pwszDbgEvent = _wcsdup( szBuffer );

    PostMessage( g_hDlg, WM_LB_ADDITEM, 0, (LPARAM)pwszDbgEvent );

    return retValue;
}
```