

An Explanation of VDMDBG.DLL

1.0 Overview

This document describes the mechanism used to support debugging WIN16 applications under Windows NT. The mechanism involves some code in the simulated environment for protected mode interrupt handling. It also involves 3 sections of code in the 32-bit NT environment, one for fielding information from the simulated environment and sending it to the debugger, one for handling it in the debugger, and one for services in a DLL to provide a better interface for the debugger.

16-bit simulated environment
KRNLX86.EXE

32-bit environment
DBG.LIB (in NTVDM.EXE)
WinDbg (or equivalent)
VDMDBG.DLL

2.0 Simulated environment

Once the simulated environment is up and running, if one of the normal debugging type events occurs it gets simulated to happen exactly the way it would on a normal PC. This means that if an INT 3 occurs, it pushes the flags, CS, IP, looks up the address in the IVT and then jumps to it. Same goes for INT 1's and INT D's (GP Faults). If the processor is in protected mode, these interrupts are normally trapped by the DOS extender (most of the time DOSX.EXE) and then reflected down into the real mode interrupts (except for GP Faults).

In order to debug in the simulated environment, it is necessary to catch and process these events. The simplest way to do this is to install our own routines to watch these interrupts and perform the proper communication when they occur. Inside KRNLX86.EXE there is a stub for each of the important interrupts. The stubs are inserted into the interrupt chains at appropriate points. It is important that the stubs are installed as early as possible so that routines that are installed later get control first. In this way, the events are only detected when they are not handled by some handler in the simulated environment. KRNLX86.EXE installs the stubs during its own load process very early on in the Win16 subsystem starting process.

Also, segment loading and unloading notifications are routed to these small stubs so that the debugger can be notified when selectors are associated with programs and their symbols.

The small stubs will perform a BOP (a method of transitioning from the simulated environment back to the real 32-bit NT environment and notifying it that a debug event has occurred). After the BOP, they will either perform an IRET instruction to return control back to the simulated program generating the interrupt, or pass the interrupt down to the previous interrupt handler (continue down the chain). The decision whether to return from or pass the interrupt will be made on the basis of the ContinueDebugEvent continue status parameter. DBG_CONTINUE will IRET, DBG_EXCEPTION_NOT_HANDLE will pass it back down the chain.

3.0 32-bit environment

Once the 32-bit environment has been notified that a debug event has occurred, it begins executing code in NTVDM (DBG.C) which parcels up the register context (based on the type of event which occurred) and communicates all the information to the debugger.

Since the debugger is in another process's context, communication is done through exceptions. NTVDM will raise an exception with the status STATUS_VDM_EVENT. Exception information will be passed via the "lpArguments" parameter to the API RaiseException().

The lpArguments parameter will cause an array of 4 DWORD values to be passed to the debugger. The values of the meanings of the array will be discussed in section 4.0

The debugger should receive this exception and return from the call WaitForDebugEvent (debuggers should always have some thread waiting for debug events). By examining the events dwDebugEventCode member, the debugger can determine the type of the debug event. If this type is EXCEPTION_DEBUG_EVENT, then the

u.Exception.ExceptionRecord.ExceptionCode

member of the debug event structure will contain the exception type. If this value is STATUS_VDM_EVENT then the exception is coming from the 16-bit environment.

When an exception of this type is detected, a debugger should load (if it hasn't done so already) the VDMDBG.DLL and determine the addresses of the key functions needed. The key functions are:

VDMProcessException
VDMGetThreadSelectorEntry
VDMGetPointer
VDMGetThreadContext
VDMSetThreadContext
VDMGetSelectorModule
VDMGetModuleSelector
VDMKillWOW
VDMDetectWOW
VDMBreakThread
VDMModuleFirst
VDMModuleNext
VDMGlobalFirst
VDMGlobalNext

The prototypes and structures used by this DLL are prototyped in the header file VDMDBG.H. Section 5.0 explains each of these functions.

Debuggers should not use these APIs to deal with the 16-bit environment:

GetThreadSelectorEntry
GetThreadContext
SetThreadContext

The APIs ReadProcessMemory and WriteProcessMemory are still useful except that the debugger must convert the 16-bit addresses into 32-bit addresses (using VDMGetPointer).

Each and every exception with the exception code of STATUS_VDM_EVENT should be passed to the function VDMProcessException. This function filters out the extraneous communication between the 16-bit environment and the VDMDBG.DLL. Most of this extraneous communication a debugger can ignore (deals with segment & module loading & unloading, this information will be provided via another interface). If the event is part of this communication process, VDMProcessException will return FALSE, if not, it will return TRUE.

If the function VDMProcessException returns FALSE, the debugger can immediately call the API ContinueDebugEvent with a continue status of DBG_CONTINUE and return to waiting for additional debug events.

If the function VDMProcessException returns TRUE, then the lpArguments parameter of the exception should be processed to determine the type of event (INT 1, INT 3, INT D, etc.). The debugger can then act accordingly. When it has completed operating with the thread, the debugger should call the API ContinueDebugEvent with a continue status of DBG_CONTINUE or DBG_EXCEPTION_NOT_HANDLED.

For the processing of each of the debug events, the debugger should use the API VDMGetThreadSelectorEntry, VDMGetThreadContext, and VDMSetThreadContext instead of the likewise named functions listed above that are exported from KERNEL32.DLL. These functions operate on an x86 CONTEXT structure, even when running on a non-x86 machine. The debugger should likewise present an x86 debugging view (x86 register dump, x86 dis-assembly, breakpoints are INT 3's, etc.)

4.0 Communication Protocol via. Exceptions

The method of communicating between the application and debugger will be exceptions. NTVDM will raise an exception and debugger should receive it. NTVDM can detect whether or not it is being debugged, and will conditionally raise the exception.

If a debugger attaches itself to an existing NTVDM process, the 32-bit environment detects this and communicates an acknowledgement of this attachment. This allows debuggers to perform initialization dealing with the 16-bit environment before beginning any new 16-bit tasks.

For the simulated 16-bit environment, the exception code will always be STATUS_VDM_EVENT.

On NT, each exception can only contain up to 4 DWORD values. Here is how they are used for

this communication. The first 2 DWORDs contain 4 WORD fields. The last two DWORDs are just DWORDs.

```

+-----+
| W1 | W2 | = DW1
+-----+
| W3 | W4 | = DW2
+-----+
|   DW3   |
+-----+
|   DW4   |
+-----+

```

The header file VDMDBG.H contains macros for isolating the individual parts of the exception information.

The W1 field is a WORD which specifies which type of event has occurred. This can be one of the following:

W1 == 0 - Segment Load Notification	(DBG_SEGLOAD)
W1 == 1 - Segment Move Notification	(DBG_SEGMOVE)
W1 == 2 - Segment Free Notification	(DBG_SEGFREE)
W1 == 3 - Module Load Notification	(DBG_MODLOAD)
W1 == 4 - Module Free Notification	(DBG_MODFREE)
W1 == 5 - Int 01h break	(DBG_SINGLSTEP)
W1 == 6 - Int 03h break	(DBG_BREAKPOINT)
W1 == 7 - Int 0Dh break (GP Fault)	(DBG_GPFALT)
W1 == 8 - Divide Overflow	(DBG_DIVOVERFLOW)
W1 == 9 - Invalid Opcode Fault	(DBG_INSTRFAULT)
W1 == 10 - Task starting	(DBG_TASKSTART)
W1 == 11 - Task stop	(DBG_TASKSTOP)
W1 == 12 - DLL starting	(DBG_DLLSTART)
W1 == 13 - DLL stop	(DBG_DLLSTOP)

They are described below.

The debugger will probably need to be smart enough to know how to manage both protected mode selectors and segment numbers from simulated real mode.

Segment/selector to symbol lookup should be mode sensitive and only use segments from the appropriate mode.

4.1 Segment Load Notification

Under Win16, this event is used to indicate that a selector has just been created and that it maps to a module's segment.

When a .EXE or .DLL is loaded, many of these events will be received. No module load notification event will occur (this is the way it is done under Windows 3.1 too).

Under DOS, no segment load notifications will occur.

W1 = DBG_SEGLOAD (0)
W2 = Unused
W3 = Unused
W4 = Unused
DW3 = Pointer to a SEGMENT_NOTE structure in NTVDM address space
DW4 = Reserved

SEGMENT_NOTE structure field definitions will be:

Selector1 - Selector assigned to new segment
Selector2 - Unused
Segment - Segment within module
Module - Null terminated module name
FileName - Null terminated path to executable image
Type - Code/data information from segment definition
Length - Unused

VDMProcessException will return FALSE for this event.

4.2 Segment Move Notification

A segment has changed from one selector number to another. If the new selector number is 0, this should be considered the same as discarding (freeing) the segment.

This event only happens under Win16. As such, these selectors should be tagged as protected mode only selectors.

W1 = DBG_SEGMOVE (1)
W2 = Unused
W3 = Unused
W4 = Unused
DW3 = Pointer to a SEGMENT_NOTE structure in NTVDM address space
DW4 = Reserved

SEGMENT_NOTE structure field definitions will be:

Selector1 - Old selector number
Selector2 - New selector number (0 to discard segment)
Segment - Unused
Module - Unused
FileName - Unused

Type	- Unused
Length	- Unused

VDMProcessException will return FALSE for this event.

4.3 Segment Free Notification

When a segment is being released, this event will be received.

This event only happens under Win16.

W1	= DBG_SEGFREE (2)
W2	= Unused
W3	= Unused
W4	= Unused
DW3	= Pointer to a SEGMENT_NOTE structure in NTVDM address space
DW4	= Reserved

SEGMENT_NOTE structure field definitions will be:

Selector1	- Selector number
Selector2	- Unused
Segment	- Unused
Module	- Unused
FileName	- Unused
Type	- Unused
Length	- Unused

VDMProcessException will return FALSE for this event.

4.4 Module Load Notification

This event is used to indicate that a module is going to take up a range of memory.

This event is used only under DOS. As such, these segment numbers should be tagged as simulated real mode only selectors.

W1	= DBG_MODLOAD (3)
W2	= Unused
W3	= Unused
W4	= Unused
DW3	= Pointer to a SEGMENT_NOTE structure in NTVDM address space
DW4	= Reserved

SEGMENT_NOTE structure field definitions will be:

Selector1	- Unused
-----------	----------

Selector2	- Unused
Segment	- Starting segment
Module	- Null terminated module name
FileName	- Null terminated path to executable image
Type	- Unused
Length	- Length of module (in bytes)

VDMProcessException will return FALSE for this event.

4.5 Module Free Notification

Module freeing notifications happen under both DOS and Win16. For to determine which selectors to free for a Win16 application, all of the selectors must be scanned to determine if it was associated with this module. Again, this is the way it is done under Windows 3.1.

W1	= DBG_MODFREE (4)
W2	= Unused
W3	= Unused
W4	= Unused
DW3	= Pointer to a SEGMENT_NOTE structure in NTVDM address space
DW4	= Reserved

SEGMENT_NOTE structure field definitions will be:

Selector1	- Unused
Selector2	- Unused
Segment	- Unused
Module	- Null terminated module name
FileName	- Null terminated path to executable image
Type	- Unused
Length	- Unused

VDMProcessException will return FALSE for this event.

4.6 Int 01h break

This event probably requires interaction with the debugger and its internal breakpoint, trace bit setting mechanisms.

W1	= DBG_SINGLE_STEP (5)
W2	= Unused
W3	= Unused
W4	= Unused
DW3	= Unused
DW4	= Reserved

VDMProcessException will return TRUE for this event.

4.7 Int 03h break

This event probably requires interaction with the debugger and its internal breakpoints.

W1	=	DBG_BREAKPOINT (6)
W2	=	Unused
W3	=	Unused
W4	=	Unused
DW3	=	Unused
DW4	=	Reserved

VDMProcessException will return TRUE for this event.

4.8 Int 0Dh break (GP Fault)

This event probably requires interaction with the debugger and its internal breakpoints.

W1	=	DBG_GPFAULT (7)
W2	=	Unused
W3	=	Unused
W4	=	Unused
DW3	=	Unused
DW4	=	Reserved

It is also important to note that all GP Faults will not be sent via this interface. The Win16 subsystem intercepts some of the GP faults in its parameter validation code. Faults in the parameter validation code indicated that an invalid parameter is being passed to one of the 16-bit APIs. There is currently no way to intercept these faults, the APIs will just return errors in the same mechanism as under Windows 3.1.

VDMProcessException will return TRUE for this event.

4.9 Divide Overflow break (Int 0)

This event probably requires interaction with the debugger and its internal breakpoints.

W1	=	DBG_DIVOVERFLOW (8)
W2	=	Unused
W3	=	Unused
W4	=	Unused
DW3	=	Unused
DW4	=	Reserved

VDMProcessException will return TRUE for this event.

4.A Invalid Opcode Fault (Int 6)

W1 = DBG_INSTRFAULT (9)
W2 = Unused
W3 = Unused
W4 = Unused
DW3 = Unused
DW4 = Reserved

VDMProcessException will return TRUE for this event.

4.B Task starting

After all of the image has been loaded for the application, but before executing the first instruction, this event will occur.

W1 = DBG_TASKSTART (0Ah)
W2 = Unused
W3 = Unused
W4 = Unused
DW3 = Pointer to an IMAGE_NOTE structure in NTVDM address space
DW4 = Reserved

IMAGE_NOTE structure field definitions will be:

Module - Null terminated module name
FileName - Null terminated path to executable image

VDMProcessException will return TRUE for this event.

4.C Task stopping

After all of the image has been unloaded for the application this event will occur. None of the segments for the application will be valid. This is provided for the debugger to clean up any internal data it keeps on a per task basis.

W1 = DBG_TASKSTOP (0Bh)
W2 = Unused
W3 = Unused
W4 = Unused
DW3 = Pointer to an IMAGE_NOTE structure in NTVDM address space
DW4 = Reserved

IMAGE_NOTE structure field definitions will be:

Module - Null terminated module name
FileName - Null terminated path to executable image

VDMProcessException will return TRUE for this event.

4.D Dll starting

After the image of the DLL has been loaded, but before the Dll's initialization code is executed, this event will occur.

W1 = DBG_DLLSTART (0Ch)
W2 = Unused
W3 = Unused
W4 = Unused
DW3 = Pointer to an IMAGE_NOTE structure in NTVDM address space
DW4 = Reserved

IMAGE_NOTE structure field definitions will be:

Module - Null terminated module name
FileName - Null terminated path to executable image

VDMProcessException will return TRUE for this event.

4.E Dll stopping

After all of the image of the DLL has been unloaded for the Dll this event will occur. None of the segments for the Dll will be valid. This is provided for the debugger to clean up any internal data it keeps on a per Dll basis.

W1 = DBG_DLLSTOP (0Ch)
W2 = Unused
W3 = Unused
W4 = Unused
DW3 = Pointer to an IMAGE_NOTE structure in NTVDM address space
DW4 = Reserved

IMAGE_NOTE structure field definitions will be:

Module - Null terminated module name
FileName - Null terminated path to executable image

VDMProcessException will return TRUE for this event.

4.F Attach Acknowledgement

Once the 16-bit environment has detected that a debugger is present it sends this event to allow the debugger to perform any initialization processing specific to the 16-bit environment.

W1 = DBG_ATTACH (0Dh)
W2 = Unused
W3 = Unused
W4 = Unused
DW3 = Unused
DW4 = Reserved

VDMProcessException will return TRUE for this event.

5.0 The VDM debugging APIs

These APIs are described below:

VDMProcessException
VDMGetThreadSelectorEntry
VDMGetPointer
VDMGetThreadContext
VDMSetThreadContext
VDMGetSelectorModule
VDMGetModuleSelector
VDMEnumProcessWOW
VDMEnumTaskWOW

The following APIs require WOWDEB.EXE to be loaded and running in the Win16 subsystem. WOWDEB.EXE is started automatically if the debugger is present at process creation time (CreateProcess with debug options set). If the debugger is attached to the Win16 subsystem process after the process has already been created (with the DebugActiveProcess API), then the debugger should spawn WOWDEB.EXE before starting the Win16 task to be debugged.

VDMModuleFirst
VDMModuleNext
VDMGlobalFirst
VDMGlobalNext

The following APIs are obsolete. They return failure conditions.

VDMKillWOW
VDMDetectWOW
VDMBreakThread

5.1 VDMProcessException

BOOL VDMProcessException(

```

        LPDEBUG_EVENT lpDebugEvent
    );

```

The VDMProcessException function performs the pre-processing needed to prepare the debug event for handling by the debugger.

This function filters all VDM debuggee/debugger communication for information which is only used by the VDM debugging DLL (VDMDBG.DLL). This function is only valid in the context of processing for debug events that are of type STATUS_VDM_EVENT.

lpDebugEvent Points to a DEBUG_EVENT structure that was returned from WaitForDebugEvent.

The return value is TRUE if the debug event should be processed by the debugger. The return value is FALSE if the debug event should be ignored. This indicates that no processing should occur for this debug event. The event should be continued using ContinueDebugEvent with a continue status of DBG_CONTINUE.

5.2 VDMGetThreadSelectorEntry

```

BOOL VDMGetThreadSelectorEntry(
    HANDLE hProcess,
    HANDLE hThread,
    WORD wSelector
    LPLDT_ENTRY lpSelectorEntry
);

```

This function is used to return a descriptor table entry for the specified VDM thread corresponding to the specified selector. This function is similar to the API GetThreadSelectorEntry except that it works on the simulated DOS/WIN16 environment, and it works on all systems, not just x86 systems.

This API returns a simulated descriptor entry on non-x86 systems. Simulated descriptor entries may (on some systems) have the base value adjusted to account for the fact that the simulated address space may not begin at linear (32-bit) address 0.

It is also important to note that 16-bit applications may modify the contents of the LDT entries. For example, the Win16 subsystem may change the selector's base value to coalesce discontinuous memory segments. Also, please see the description in VDMGetPointer.

hProcess Supplies a handle to the DOS/WIN16 sub-system process. The handle must have been created with PROCESS_VM_READ access.

hThread Supplies a handle to the thread that contains the specified selector. The handle must have been created with THREAD_QUERY_INFORMATION access.

wSelector Supplies the selector value to look up. The selector value may be a global selector (GDT) or a local selector (LDT).

lpSelectorEntry If the specified selector is contained within the thread's descriptor tables, its descriptor table entry is copied into the data structure pointed to by this parameter. This data can be used to compute the linear base address that segment relative addresses refer to.

The return value is TRUE if the operation was successful. In that case, the data structure pointed to by lpSelectorEntry receives a copy of the specified descriptor table entry.

Refer to the WinHelp entry for the structure of an LDT_ENTRY.

5.3 VDMGetPointer

```
ULONG VDMGetPointer(  
    HANDLE hProcess,  
    HANDLE hThread,  
    WORD wSelector,  
    DWORD dwOffset,  
    BOOL fProtMode  
);
```

This function is used to convert a 16-bit address into a flat 32-bit address.

It is also very important to note that under the WIN16 environment, pointers derived from protected mode selectors may change. WIN16 does this by changing selector's base value to coalesce memory during compaction. For this reason, it is necessary that any addresses evaluated in the 16-bit environment should be reevaluated each time an access into 16-bit memory is needed. An example would be placing and removing 16-bit breakpoint instructions. If the debugger is told to place a breakpoint at a given address of SEL:OFFSET, then when the breakpoint is needed to be removed, the debugger must reevaluate the SEL:OFFSET address since it might have moved in terms of the linear 32-bit address.

hProcess Supplies a handle to the DOS/WIN16 sub-system process. The handle must have been created with PROCESS_VM_READ access.

hThread Supplies a handle to the thread that contains the specified selector. The handle must have been created with THREAD_QUERY_INFORMATION access.

wSelector Supplies the selector value to determine the pointer for.

dwOffset Supplies the offset value to determine the pointer for.

fProtMode Indicates whether the 16-bit address specified is a real mode (v86 mode) address or a protected mode address. Protected mode addresses are translated through the descriptor tables.

The return value is a 32-bit linear address pointing to the memory in the simulated DOS/WIN16 environment that represents the 16-bit address specified. The return value is NULL, if the address specified is invalid. On some systems, NULL may be returned for the address 0:0.

To determine the address of the simulated 16-bit memory, **VDMGetPointer** may be called with the address 0:0 and an **fProtMode** of FALSE.

5.4 **VDMGetThreadContext**

```
BOOL VDMGetThreadContext(  
    LPDEBUG_EVENT lpDebugEvent,  
    LPVDMCONTEXT lpVDMContext  
);
```

The context of a specified simulated DOS or WIN16 thread can be retrieved using **VDMGetThreadContext**. The context returned will always be that of an x86 system.

This API returns a simulated context for x86 and non-x86 systems. Under some systems, values within the context are meaningless. For example, the **CONTEXT_DEBUG_REGISTERS** portions on RISC systems have no effect.

Release 1 of Windows NT has a 286 emulator on RISC systems. For this reason, only the 16-bit registers can be supported on RISC systems.

lpDebugEvent Points to a **DEBUG_EVENT** structure that was returned from **WaitForDebugEvent**.

lpVDMContext If the specified thread is a simulated DOS or WIN16 thread, its context is copied into the data structure pointed to by this parameter.

The return value is TRUE if the operation was successful. In that case, the data structure pointed to by **<lpVDMContext>** receives a copy of the simulated context.

Refer to the WinHelp for the structure of a **VDMCONTEXT** (same as x86 **CONTEXT** structure in **NTI386.H**).

5.5 **VDMSetThreadContext**

```
BOOL VDMSetThreadContext(  
    LPDEBUG_EVENT lpDebugEvent,  
    LPVDMCONTEXT lpVDMContext  
);
```

The VDMSetThreadContext function sets the simulated context in the specified DOS or WIN16 thread. The function allows selective context to be set based on the value of the ContextFlags member of the context structure. This API operates only when debugging a simulated DOS or WIN16 thread. The caller must have a thread handle which was created with THREAD_SET_CONTEXT access.

The context set will always be that of an x86 system.

lpDebugEvent	Points to a DEBUG_EVENT structure that was returned from WaitForDebugEvent.
lpVDMContext	Supplies the address of a context structure that contains the context that is to be set in the specified thread. The value of the ContextFlags member of this structure specifies which portions of a thread's context are to be set. Some values in the context structure are not settable and are silently set to the correct values. This includes CPU status register bits that specify processor mode, debug register global enabling bits, and other state that must be completely controlled by the system.

The return value is TRUE if the context was set; otherwise it is FALSE if an error occurred.

Refer to the WinHelp for the structure of a VDMCONTEXT (same as x86 CONTEXT structure in NTI386.H).

5.6 VDMGetSelectorModule

```
BOOL VDMGetSelectorModule(  
    HANDLE hProcess,  
    HANDLE hThread,  
    WORD wSelector,  
    PUINT lpSegmentNumber,  
    LPSTR lpModuleName,  
    UINT nSize,  
    LPSTR lpModulePath,  
    UINT nPathSize  
);
```

The VDMGetSelectorModule function is intended to provide an interface such that debuggers can determine which module belongs to a code or data address. Given the selector for that address, the function will return the module name, and the segment number (0 based) of the segment that corresponds to that selector. If the selector is not a selector which directly corresponds to a module, the function will return FALSE.

hProcess	Supplies a handle to the process of the 16-bit environment
----------	--

hThread	Supplies a handle to a thread in the 16-bit environment
wSelector	Supplies the selector value to look up.
lpSegmentNumber	Returns the segment number within the module.
lpModuleName	Buffer to receive the module name.
nModuleSize	Size of the buffer.
lpModulePath	Buffer to receive the module path name.
nPathSize	Size of the buffer.

The return value is TRUE if the selector is mapped directly to a module. This means it must be either a code or data segment. Selectors allocated using the global memory management functions are not mapped directly to a module. The return value is FALSE if the function is not successful.

The function returns the segment number in the address specified by the lpSegmentNumber parameter, and the module name in the address specified by the lpModuleName parameter.

It is up to the debugger to determine from the module and segment number information the correct symbol, if a symbol lookup is needed.

5.7 VDMGetModuleSelector

```

BOOL VDMGetModuleSelector(
    HANDLE hProcess,
    HANDLE hThread,
    UINT uSegmentNumber,
    LPSTR lpModuleName,
    LPWORD lpSelector
);

```

The VDMGetModuleSelector function is the reverse operation of the VDMGetSelectorModule function. A module name and segment number are converted into a selector number.

hProcess	Supplies a handle to the process of the 16-bit environment
hThread	Supplies a handle to a thread in the 16-bit environment
uSegmentNumber	Supplies the segment number to look up.
lpModuleName	Specifies the module name of the segment.

lpSelector Returns the selector value.

The return value is TRUE if the module and segment are found. Also, the lpSelector value is filled-in with the selector of that segment. Otherwise, the return value is FALSE.

It is up to the debugger to convert symbol names and expressions into modules, segment numbers and offsets. Using the modules and segment numbers, a selector value can be determined. In combination with the offset, the selector can be used to index into the simulated Win16 environment for reading, writing, etc.

5.8 VDMEnumProcessWOW

```
INT VDMEnumProcessWOW(  
    PROCESSENUMPROC fp,  
    LPARAM lparam  
);
```

The VDMEnumProcessWOW function enumerates all of the processes in the system which are Win16 subsystem processes. In NT's first release, there is only one Win16 subsystem. For later releases, there may be more than one process to support address space separation for Win16 tasks.

fp Supplies the address of a callback routine.

lparam Supplies a parameter to the callback routine.

The return value is the number of Win16 subsystem processes, or the number enumerated before enumeration was terminated.

```
BOOL ProcessEnumProc(  
    DWORD dwProcessId,  
    DWORD dwAttributes,  
    LPARAM lparam  
);
```

The callback function will be called once for each Win16 subsystem process in the system.

dwProcessId Provides the process id of a Win16 subsystem.

dwAttributes Provides flags indicating information about the process.

lparam Provides the parameter passed to VDMEnumProcessWOW

The callback function should return a non-zero value to terminate enumeration.

The dwAttributes field should be compared with the bit mask WOW_SYSTEM to determine if

the process is the main Win16 subsystem process. The main Win16 subsystem process will be the process that the next Win16 task that specifies no address space separation requirements.

Win16 subsystem processes which are created due to address space separation will not have the WOW_SYSTEM bit enabled.

5.9 VDMEnumTaskWOW

```
INT VDMEnumTaskWOW(  
    DWORD dwProcessId,  
    TASKENUMPROC fp,  
    LPARAM lparam  
);
```

The VDMEnumTaskWOW function enumerates all of the Win16 tasks currently running in the Win16 subsystem process id specified.

dwProcessId Supplies the process id of the Win16 subsystem.

fp Supplies the address of a callback routine.

lparam Supplies a parameter to the callback routine.

The return value is the number of Win16 tasks or the number of tasks enumerated before terminating.

```
BOOL TaskEnumProc(  
    DWORD dwThreadId,  
    WORD hMod16,  
    WORD hTask16,  
    LPARAM lparam  
);
```

The callback function will be called once for each Win16 task.

dwThreadId Provides the thread id of the Win16 task

hMod16 Provides the Win16 module handle for the task

hTask16 Provides the Win16 task handle for the task

lparam Provides the parameter passed to VDMEnumTaskWOW

The callback function should return a non-zero value to terminate enumeration.

5.A VDMModuleFirst

```

BOOL VDMModuleFirst(
    HANDLE hProcess,
    HANDLE hThread,
    LPMODULEENTRY lpModuleEntry,
    DEBUGEVENTPROC lpEventProc,
    LPVOID lpData
);

```

This function is used to start enumerating all of the modules currently loaded in the 16-bit Windows environment. It behaves in the same manner as the Windows 3.1 ToolHelp API ModuleFirst().

hProcess Supplies a handle to the process of the 16-bit environment

hThread Supplies a handle to a thread in the 16-bit environment

lpModuleEntry Specifies the address of a MODULEENTRY structure which will be filled in with the first module in the module list.

lpEventProc Specifies the address of a procedure which might be called in the event of a debug event occurring while the communication session with the 16-bit environment is occurring.

lpData Specifies a parameter to pass to the debug event procedure

The return value is TRUE if the operation was successful. In that case, the MODULEENTRY structure will be filled in with information for the first module in the module list.

5.B VDMModuleNext

```

BOOL VDMModuleNext(
    HANDLE hProcess,
    HANDLE hThread,
    LPMODULEENTRY lpModuleEntry,
    DEBUGEVENTPROC lpEventProc,
    LPVOID lpData
);

```

This function is used to continue enumerating all of the modules currently loaded in the 16-bit Windows environment. It behaves in the same manner as the Windows 3.1 ToolHelp API ModuleNext().

hProcess Supplies a handle to the process of the 16-bit environment

hThread Supplies a handle to a thread in the 16-bit environment

lpModuleEntry Specifies the address of a **MODULEENTRY** structure which will be used to determine the next module and which will be filled in with the next module in the module list.

lpEventProc Specifies the address of a procedure which might be called in the event of a debug event occurring while the communication session with the 16-bit environment is occurring.

lpData Specifies a parameter to pass to the debug event procedure

The return value is **TRUE** if the operation was successful. In that case, the **MODULEENTRY** structure will be filled in with information for the next module in the module list.

5.C VDMGlobalFirst

```
BOOL VDMGlobalFirst(  
    HANDLE hProcess,  
    HANDLE hThread,  
    LPGLOBAENTRY lpGlobalEntry,  
    WORD wFlags,  
    DEBUGEVENTPROC lpEventProc,  
    LPVOID lpData  
);
```

This function is used to continue enumerating all of the global memory blocks currently allocated in the 16-bit Windows environment. It behaves in the same manner as the Windows 3.1 ToolHelp API **GlobalFirst()**.

hProcess Supplies a handle to the process of the 16-bit environment

hThread Supplies a handle to a thread in the 16-bit environment

lpModuleEntry Specifies the address of a **GLOBAENTRY** structure which will be filled in with information for the first global memory block matching the specified flags

wFlags Specifies which types of global memory blocks to enumerate

lpEventProc Specifies the address of a procedure which might be called in the event of a debug event occurring while the communication session with the 16-bit environment is occurring.

lpData Specifies a parameter to pass to the debug event procedure

The return value is **TRUE** if the operation was successful. In that case,

the GLOBAENTRY structure will be filled in with information for the first global memory block that matches the specified flags.

5.D VDMGlobalNext

```
BOOL VDMGlobalNext(  
    HANDLE hProcess,  
    HANDLE hThread,  
    LPGLOBAENTRY lpGlobalEntry,  
    WORD wFlags,  
    DEBUGEVENTPROC lpEventProc,  
    LPVOID lpData  
);
```

This function is used to continue enumerating all of the global memory blocks currently allocated in the 16-bit Windows environment. It behaves in the same manner as the Windows 3.1 ToolHelp API GlobalNext().

hProcess	Supplies a handle to the process of the 16-bit environment
hThread	Supplies a handle to a thread in the 16-bit environment
lpModuleEntry	Specifies the address of a GLOBAENTRY structure which will be used to determine the next global memory block and which will be filled in with information for the next global memory block matching the specified flags
wFlags	Specifies which types of global memory blocks to enumerate
lpEventProc	Specifies the address of a procedure which might be called in the event of a debug event occurring while the communication session with the 16-bit environment is occurring.
lpData	Specifies a parameter to pass to the debug event procedure

The return value is TRUE if the operation was successful. In that case, the GLOBAENTRY structure will be filled in with information for the next global memory block that matches the specified flags.

5.E VDMKillWOW

```
BOOL VDMKillWOW(void);
```

This function is obsolete and performs no operation. It returns FALSE.

5.F VDMDetectWOW

```
BOOL VDMDetectWOW(void);
```

This function is obsolete and performs no operation. It returns FALSE.

5.G VDMBreakThread

```
BOOL VDMBreakThread(  
    HANDLE hProcess  
    HANDLE hThread  
);
```

This function is obsolete and performs no operation. It returns FALSE.

