About the Elographics Inc. Touchscreen/Mouse Drivers for Microsoft Windows 3.0

10 Dec 1990

Christopher G. Hill

The text that follows describes (loosely at first) how the mouse drivers for Windows were changed to accommodate the use of a touchscreen as a duplicate pointing device. Some of this description is specific to problems presented by special DOS mode device drivers and emulators that fool MOUSE.COM into believing that input from the touchscreen is really coming from an attached mouse. The following document was written specifically to get a systems level programmer up to speed with the drivers as they existed at the time this was written. It was assumed that he (or she of course) would already be familiar with the concept of touchscreens as pointing devices and with the supporting DOS software as well. Still, for the general reader interested in writing his own device drivers for Windows, some of the more general topics covered here may help give a general overview of how a Windows device driver interacts with Windows and the computer. The topics that come later in this paper are more specific and much of them may only be of interest to those who wish to write a driver for a pointing device, or possibly a communications device. Even so, there is example code that shows how to instance DOS TSRs and virtualize their IRQs. This is invaluable no matter what kind of virtual device driver you happen to be writing. It may even be useful to those who wish to write TSRs that are compatible with Windows.

The code in the second half of this paper will be most meaningful to those who have the Windows DDK since it is derived from that source. That does not mean that it is useless to those who do not.

If you have any questions that you think I might be able to answer, please don't hesitate to contact me on BIX (user name CHill).

Christopher G. Hill

1.0

1.1 Windows 3.0 support for the Elographics Inc. touchscreen is provided by three DOS drivers and two Windows drivers. Together, they control the mouse and the touchscreen and allow the touchscreen to be used in emulation of a mouse. The three DOS drivers are:

3.) MONMOUSE.COM (the "glue" program that makes input from the touchscreen look like input from the mouse).

The two Windows drivers are:

1.2 Of the three DOS drivers, none are reentrant and normally there is no reason for a DOS device driver to be reentrant. When Windows 3.0 is run in its 386 Enhanced mode though, DOS programs that use these devices can defy this strategy since Enhanced mode allows the user to run multiple DOS sessions at once. Each of these DOS sessions will think it has complete control of the computer. In fact, each must be allowed to share the system's resources without knowing about it. This is why Windows 3.0 needs a special mouse and touchscreen device driver just for the 386 Enhanced mode.

> As an aside, Windows has three modes of operation: Real mode, Standard mode, and Enhanced mode. The 80x86 family of chips also have a number of modes of operation: real, 16-bit protected, 32-bit protected, and virtual 8086. In order to reduce confusion, especially between real modes for Windows and the processor, the mode name will be capitalized if it refers to a Windows mode, and lower case if it describes a chip mode.

1.3 Windows behaves differently in each of its three modes of operation. In Real mode, Windows and all its programs operate in the first 640K of memory and can use EMS memory to store code and data. The system processor always runs in real mode and never protected. Any program has complete access to any part of memory without restriction. When Windows runs its "DOS Box" from real mode, it shuts down all background operation and returns the system to its default DOS state. The user is essentially shelled out to DOS in this case. Just before Windows starts the DOS box, all Windows device drivers are given a chance to save the state of their respective device before the DOS session is begun. Once the state of the device is saved, the driver should reset the device to a default DOS state. Once the DOS session is terminated or suspended and Windows is reentered from the DOS session, the drivers restore the saved state of their device.

1.4 In its Standard mode, Windows operates in the processor's protected mode. When Windows' device drivers first initialize, they must do so in protected mode. This places some restrictions and difficulties in the way of doing things like communicating with existing DOS device drivers and even DOS itself. For this reason, Microsoft has added support for Intel's DOS Protected

Mode Interface (DPMI) to Windows when it runs in Standard or Enhanced mode. Programs running in protected mode can do such things as allocate and free LDT entries and change their attributes. Calls can also be made to DOS software interrupts. By manipulating an LDT entry, a driver (or Windows program) can access any part of physical memory. And by using the DPMI's software interrupt emulation, the driver can communicate with DOS and DOS mode programs. There are some special considerations though. First, if you wish to pass a pointer to a DOS program, you must consider where that pointer points. Protected mode selector:offset pairs do not have any correspondence to physical memory. The selector is merely an index into a table of 24 or 32 bit addresses and the offset is measured from the address stored in the table. Memory that is used in protected mode programs and drivers does not necessarily occur in the first megabyte of system memory and real mode programs cannot directly address memory anywhere else. So, your protected mode program will need to ask Windows to allocate it some memory from the first megabyte in the system. Then it will need to make sure to pass the real mode address of this memory to the DOS program as a segment:offset pair (NOT selector:offset). Although there is a DPMI call that allocates memory from a real mode pool, it will always fails under Windows because Windows grabs all available real mode memory before the DPMI sees it. Consequently, one must use the Windows API call to allocate real mode memory.

1.5 Window's use of both real mode and protected mode can complicates things for device drivers if they need to communicate with DOS during their operation. Since the DPMI is only present in Standard and Enhanced modes and not in Real mode, a Windows driver must have separate sections for any procedure that must communicate with DOS. The global variable __WINFLAGS must be used to decide whether or not the DPMI can be used. The constants WF_PMODE and WF_ENHANCED correspond to appropriate bits in __WINFLAGS and can be used to determine in which mode Windows is running.

1.6 As the case in point, MONMOUSE.DRV functions as both the mouse and touchscreen driver for Windows. When it is first initialized, it checks for the existence of a DOS mouse driver (usually MOUSE.COM). If it does not find it in memory, it operates as a mouse driver only and looks for a the existence of a mouse anywhere in the system (specifically - a serial, bus, InPort, PS/2 or HP mouse). If it finds one, it loads code specific to that type of mouse into a Windows locked segment. If it does not find one, then no mouse support will be available to the user. On the other hand, if a DOS mouse driver is found, then the Windows driver hooks into it in the form of an event handler.

1.7 When the DOS mouse driver (MOUSE.COM) receives input from the mouse, it will call the event handler in Windows and pass it information via the processor's registers. In real mode, this is a very straight forward process but in protected mode, the story changes a bit. When the mouse is moved or one of its buttons is clicked, Windows detects that a hardware interrupt has occurred. In order to handle this interrupt, it will switch out of protected mode and run the real mode code in MOUSE.COM that handles the mouse's IRQ line. This DOS driver determines what happened to the mouse and checks to see if the event handler needs to be called. If it does, then a call is made to the address of the event handler. This address is expected to be a real mode

segment:offset pair, not a protected mode selector:offset. To make this work, the Windows driver must have allocated what is called a real-mode-call-back address. A real-mode-call-back address is one that can be used by a program running in real mode, but that points to a function that will execute in protected mode and that can be anywhere in the processor's address space (even its virtual address space). What the real-mode-call-back address will point to is a DPMI function that resides in real mode memory and that will switch to protected mode, copy the real mode program's registers, stack, and return address into protected mode memory, and then call the protected mode function associated with that call back address. The protected mode function does what it needs to do and then adjusts a copy of the DOS program's registers and stack (it doesn't have to be the same copy as the one passed to it by the DPMI), and then returns to the DPMI function via an IRET. The DPMI function then copies the changed register values back into the real mode program's registers and returns control to it.

1.8 This is precisely what MONMOUSE.DRV does with its event handler. It registers it with the DPMI as a real-mode-call-back procedure, and then passes the address given it by the DPMI to MOUSE.COM. When the event handler is called by MOUSE.COM it informs Windows as to the mouse's behavior and then makes a return call to MOUSE.COM to clear the motion count registers. This introduces a complication. Since the event handler is operating in protected mode, it needs to use the DPMI to make the call to the DOS mouse driver and the DPMI cannot handle this kind of reentrancy (specifically, it cannot handle an INT 31h DPMI call from within a real-mode-call-back routine). Fortunately, there is an alternative. The Windows DPMI has support for not only DOS INT 21h calls, but also for *some* INT 33h mouse calls. This particular call is one of those that are supported so it can be called directly. Once the event handler does this, it returns to MOUSE.COM via the DPMI.

1.9 As was mentioned earlier, when Real mode or Standard mode Windows starts a DOS session, they allow any Windows driver in the system to save the state of its device and shut down first. When the DOS session is either suspended or closed, Windows will again call the driver; this time to restart operation and restore the discontinued state. For the mouse driver, this means saving its current position, shape, and button state and then calling the hardware reset function. If MONMOUSE.COM was loaded when Windows started, then MONMOUSE.DRV makes a call to MONMOUSE.COM's function 00ffh. When the DOS session terminates, a call is made to its function 01ffh (calls to functions 00feh and 01feh are made when the INT33h initialization and termination code is called as well, but this only happens when Windows proper is loaded and terminated, and only in Real and Standard modes - enhanced mode is a whole different matter as we'll see in a moment).

1.10 The one thing to keep in mind throughout the operation of Windows in Real and Standard modes, is that there is a distinct separation between the operation of a DOS session and the operation of Windows. When a DOS session executes, Windows shuts down almost completely (it only monitors the keyboard for system hotkeys). When the user returns to Windows from a DOS session via hotkey, the DOS session is completely shut down - no background processing goes on and no operation in a window is allowed. For this reason, there are no contentions for

resources between DOS sessions, or between DOS and Windows, in either Real or Standard modes.

2.0

2.1 In Enhanced mode, all this changes. Not only can DOS sessions run concurrently with each other and Windows, but they can be run in a window too. Since DOS programs are selfish, unruly beasts by nature, the potential for disaster is considerable. Windows must be very careful in how it lets each DOS session access system resources.

2.2 DOS sessions are run in the special virtual 8086 mode of the 80386 chip. Each DOS session is given its own virtual address space and virtual system hardware. It is up to the Enhanced mode device drivers (VxDs) to make sure that virtualization of the hardware and instancing of data structures and any DOS mode TSRs is done correctly. Each hardware device in the system is handled by a separate VxD (in this notation, the x gets replaced by a letter representing a particular device; e.g., VMD=Virtual Mouse Device, VPICD=Virtual PIC Device where PIC=Priority Interrupt Controller). In the case of most VxDs, this meant they had to virtualize whatever hardware interrupt their device was attached to and to serialize any requests to that device on a per-task (Virtual Machine or VM) basis. One of the things unique to the VMD is that it has to be aware of the MOUSE.COM program. MOUSE.COM is a special problem because it makes changes in the state of the system mouse and isn't aware of Windows. Since there is only one mouse per computer, and since each DOS mode program, as well as Windows itself, can use the mouse in a different mode, changing its shape, on/off status, and position, a single copy of MOUSE.COM that is shared among all concurrently running DOS programs would be disastrous. MONMOUSE.DRV only adds to the complexity of this situation because it makes Windows rely on MOUSE.COM too. The original Windows VMD solved this problem by making a separate copy of MOUSE.COM for each virtual machine (VM) in the system (ordinarily, each new VM's low memory is mapped into its address space from the original DOS pool - a VxD must specifically ask Windows to copy regions of memory rather than map them). This solution can be extended to the case of MONMOUSE.DRV because Windows itself is run in a VM and so it will get its own copy of MOUSE.COM in its VM.

2.3 The way that the VMD finds MOUSE.COM in order to instance it, is to look at the address stored in the interrupt vector table that corresponds to the mouse driver (INT 33h). If that vector is blank (zeroed out) or points to a single IRET instruction, then it is assumed that MOUSE.COM is not present. If, however, it is found that the INT 33h vector does point to a program, then the segment portion of that vector is used to locate the memory block in which the mouse driver resides. The memory control block (Microsoft uses the term "arena" for this) for that chunk of memory is then inspected for validity and for the size of the block it controls. The segment address of the block and its size are then used to copy it into each new VM.

2.4 There are two big assumptions being made by the VMD here. The first is that the DOS mouse driver is a .COM program. The second is that there is only one program chained into the INT 33h vector. If either of these two assumptions are wrong, the VMD will not work correctly

and will probably crash the system. The first assumption means that the segment pointed to by the INT 33h vector is the first and only segment owned by that program, and that its first 100h bytes contain the program's PSP. If this is the case, then the VMD merely has to decrement the segment value by one to find that memory block's control header. If, on the other hand, the file is an .EXE type file, then it is possible that the routine pointed to by the vector is in one of many segments - all within the same memory block, and no guarantee as to which is the first segment in memory. Even if the entry point was in the first segment of that program, it won't be the first one in the memory block. Unlike a .COM program, an .EXE program does not have its PSP stored in a code segment. It is in its own segment, and *that* segment is the first one in the memory block. This means that *if* the code pointed to by the INT 33h vector is the first or only segment in a that *program*, then one must decrement that segment's value by 11h instead of 1h in order to find the control header.

2.5 If MONMOUSE.COM is loaded in memory, the first assumption is invalid. Since MONMOUSE.COM takes over the INT 33h vector, the VMD instances it but not MOUSE.COM. This means that if multiple VM's are using the mouse, MOUSE.COM's capacity to handle input from a single source will be exceeded. In the case that the MONMOUSE.DRV driver for Windows is being used and Windows is running in Enhanced mode, one mouse-using VM is always running. That VM is the one that Windows runs in. If a DOS session is then started, another VM is created. If a program in that new VM tries to access MOUSE.COM, trouble starts.

2.6 What this means is that the VMD must be rewritten to check for the existence of MONMOUSE.COM. If it finds it in memory, then MONMOUSE.COM must be queried for MOUSE.COM's address in memory. Once the this is known, MOUSE.COM can also be instanced along with MONMOUSE.COM. There is a third part to this though. MONMOUSE.COM gets its input from ELODEV.EXE and for the same reasons that MOUSE.COM and MONMOUSE.COM needed to be instanced, so too does ELODEV.EXE. This brings up assumption number two: the DOS touchscreen driver is an .EXE file. This means that we need to take extra care in finding its memory block's control header. Fortunately, ELODEV's entry point is in the physically first segment of the program. ELODEV's PSP occupies the immediately preceding 256 bytes, and the program's memory control block is the paragraph right before that. The new VMD (let's call it the VMMD from now on, MM=Monitor Mouse) must also instance ELODEV.EXE for each new VM created.

2.7 Instancing data is only half of the story for VxDs. The other half is virtualizing any hardware that it controls. Hardware is virtualized so that each VM will think it has that hardware to itself exclusively. It also is a way of telling Windows which VM is to handle and IRQ event (otherwise it will broadcast the event to *all* VMs). This has the effect of queueing events for the separate VM's as they occur. When the VxD is notified of an event, it can then decide to which VM's queue to add it. Typically, the VM that currently has the focus will be given the event. However, if the focus changes from one VM to another while a critical event is in process, it may be necessary to continue passing events to the old VM until the critical event is over. This is

especially important where the PIC and asynchronous hardware interrupts are concerned. All mouse and touchscreen events are asynchronous hardware interrupts so this concerns us. The VMD takes care of this problem for mouse events. Fortunately, touchscreen controllers work very much like mouse controllers (take this last statement from the programmer's point of view, not the engineer's please) so we can just duplicate much of the code that already exists for mouse events, being careful to check for the existence ELODEV.EXE and MONMOUSE.COM in memory.

2.8 A momentary digression is necessary in order to explain how VxDs work before trying to explain how the VMMD does what it needs to do. If Windows is not being run in enhanced mode, it never even bothers with things like VxDs. As a matter of fact, the program WIN.COM that one invokes to start Windows is just a shell that checks to see what kind of processor is present and how much memory a computer has. It checks the command line to see if the user specified a particular mode of Windows to operate in, and then it loads and executes one of three different kernels. So Real mode Windows, Standard mode Windows and Enhanced mode Windows are three distinct programs and only the 386 version of Windows loads any VxDs. The file in the Windows system directory called WIN386.EXE contains a number of default VxDs. When Windows starts up in Enhanced mode, the SYSTEM.INI file gets read in and any drivers specified in the [386Enh] section of this file are loaded. Those drivers that start with an asterisk are internal to the WIN386.EXE file. For example, when you first run setup, it determines what type of mouse is attached to the computer. If it finds a Microsoft or IBM mouse of any type, it adds the following entry to the SYSTEM.INI file in the [386Enh] section:

This tells Enhanced mode Windows to associate the internal VMD with the mouse device. If the asterisk had not preceded the driver name, then Windows would have looked for a driver named VMD (no extension) in its system directory.

2.9 Enhanced mode Windows loads all VxDs into memory before it executes code or uses data in any of them. Each VxD can have code in any one of three different segments. The code in each one of those segments has its data in a separate, associated, data segment. When a VxD is loaded, its code is combined with other VxD code in the appropriate segments and its data are combined in a like manner. This is done so that all initialization code and data can be thrown away once the initialization phase is over. Initialization occurs in four separate phases:

2.10 The first is the real mode initialization. Each driver can have an optional initialization routine that runs in real mode. This code has easy access to all DOS services (except program termination - that one is a no-no) and DOS memory. It can check for the existence of TSR's that need to be instanced. This is its primary function. Information that it can pass on to any protected mode code is minimal. It is allowed to pass a pointer to a structure that contains addresses and sizes of memory blocks to be instanced, and it can pass a 32-bit value to protected mode initialization code via the EDX register. Once all VxDs have executed their real mode initialization code, that whole segment and the real mode initialization data segment are tossed out of memory and Windows begins execution in 32-bit protected mode.

Note that all VxDs operate in 32-bit protected mode with a privilege level of 0 at all times while the rest of Windows runs in 16-bit protected mode, and all DOS VM's operate in virtual 8086 mode. This means that all drivers have access to 4 gigabytes of data, code and stack space (virtual memory not included). Since Windows 3.0 operates with only one GDT, which doubles for the only LDT as well, and since all drivers are loaded in a single segment with a base address of 0000h and a size limit of 0ffffffffh (4 gigabytes), any part of physical memory can be addressed (ES, and DS hold the same selector, and CS is just an alias for that same segment - for Windows 3.0, the selector for DS/ES is 0030h, and for CS is 0028h)

2.11 Next comes the protected mode initialization. It occurs in three phases. The first is called the critical initialization phase. The VxD's critical initialization procedure is called with all hardware interrupts turned off. For this reason, it should only do what is necessary for the other initialization code to operate reliably. After each VxD has a chance to execute its critical initialization code, the next phase of initialization starts.

2.12 The main protected mode initialization phase is where most of the work is done. The VMMD determines what kind of mouse and touchscreen controllers are attached to the system, whether ELODEV and MOUSE.COM are loaded in memory, and what resources must be virtualized. All virtualization takes place here and any VM specific data structures are initialized here. When this is done, the final phase of initialization begins. This is called the "initialization complete" phase. It is mostly a cleanup process but here is where the VMMD instances MOUSE.COM, MONMOUSE.COM and ELODEV.EXE if they are in memory.

2.13 After all of the VxDs have been initialized and all the initialization code and data have been removed from memory, the hardware device drivers (like MONMOUSE.DRV) are loaded and initialized. These drivers must be able to run in real mode as well as protected mode as they will be used by Windows no matter what mode *it* runs in. Once this phase is finished, Windows creates a VM for itself, and starts up the system shell (default is the program manager). Any non-Windows application is given its own VM in which to run.

3.0

3.1 To give you a more concrete idea of what the above means, I'll make a chronological list of what parts of the VMMD (same for the VMD) are loaded, the functions within that segment that execute, and just what they do.

3.1.0 VxD_REAL_INIT_SEG

3.1.1 Int33_Real_Init:

This is the real mode initialization procedure. It runs in a 16-bit real mode segment but can use the 32-bit GP registers since it is guaranteed to be executed

on a 386 or better processor. Here is where the VMMD figures out what IRQ ELODEV is using and whether it is a serial or bus controller. It does the same for the system mouse. If MOUSE.COM is not in memory, the this code returns immediately with a double word of zero in EDX. If it is found, then the mouse type is placed in DX and the upper word of EDX is set to 0ffffh. If MOUSE.COM is not version 6.0 or better, then zero is placed in DX. Next, this procedure attempts to find ELODEV. If it is not found, the high word of EDX is left as is. If it is found, then the high word of EDX is set as follows: The high byte will contain 0 if a bus touchscreen controller is attached, and 0ffh if a serial controller is present. The low byte will contain the IRQ in use by the touchscreen.

Note: Ideally, this is where the VMMD should figure out which programs needing instancing are present in memory. Since the original VMD does this in the initialization complete phase, we do it there too. It is harder to do there when MONMOUSE.COM is present because no INT 33h calls can be made or emulated by the DPMI in that section of code.

Returns:
if MOUSE.COM not present:

0
if present:

=

if ELODEV not present:

0ffffh

if present:

HIBYTE =

0 if bus controller

0ffh if serial controller

LOBYTE =

IRQ number in use by controller

if MOUSE.COM earlier than 6.00:

0

else

DH = mouse type

0 = unknown

1 = bus mouse

2 = serial mouse

3 = InPort mouse

4 = PS/2 mouse

5 = HP mouse

DL = IRQ used

0 = PS/2 mouse

2-5,7 other types of mouse

3.1.2 Int33_Real_Init Code:

```
BeginProc Int33_Real_Init

;  If another mouse driver is loaded then don't load -- Just abort our load
;

bx, Duplicate_From_INT2F OR Duplicate_Device_ID

Int33_RI_Abort_Load

es


ax, 3533h
; Get interrupt vector for

21h
; int 33h through DOS
```

```
edx, edx

dx, es

dx, bx
; Q: Point to 0:0?

SHORT Int33_RI_Done
;    Y: No Int 33h driver

BYTE PTR es:[bx], 0CFh
; Q: Point to an IRET?

SHORT Int33_RI_Get_Type

edx, edx

SHORT Int33_RI_Done
;
;   Now get the mouse type
;
Int33_RI_Get_Type:

edx, -1
; edx never 0 if mouse present

ax, 24h
; Mouse get type call

cx, cx
; Zero in case call fails

33h
; Do it

dx, cx
; Pass this info to prot mode

;*****************************************************************
*********
; if Elodev present, use high word of edx to pass data to protected mode
; init so we can virtualize Elodev's resources. (IRQ or comm port)
;
```

```
ifdef MONMOUSE


ax,65h

bx,bx
;monmouse here?

33h

bx,65h

short RMI_gotmonmouse

RMI_FindElodev

short RMI_mm_exit

short RMI_1
;otherwise Elodev int# returned in dl
RMI_gotmonmouse:

ax,65h

bx,3

33h
RMI_1:

byte ptr [RMI_intcall+1],dl ;elodev int# returned in dl

ax,ax

bx,offset RMI_InfoDataStruc

bx
;Elodev trashes bx on each call

short RMI_intcall
RMI_intcall:

65h
;This only works because we're in real
```

```
bx
;  mode. Don't try this at home kids.

ax,[bx.intrpt]

bx,[bx.comport]

bx,bx

short RMI_2
;if negative, then no comm port

short RMI_2
;if zero, then no comm port

ah,-1
;-1 means serial card
RMI_2:

[RMI_ED_IRQ],ax
RMI_mm_exit:


[RMI_ED_IRQ],0

short RMI_noIRQ

edx,10h
;otherwise put info in high word

dx,[RMI_ED_IRQ]
; for use by Int33_Init procedure

edx,10h
RMI_noIRQ:
endif
;*****************************************************************
*********
Int33_RI_Done:

bx, bx
```

```
si, si

ax, Device_Load_Ok

es

;
;   Another mouse driver exists.  Don't load
;
Int33_RI_Abort_Load:

bx, bx

si, si

ax, Abort_Device_Load + No_Fail_Message

es

EndProc Int33_Real_Init
```

3.2.0 VxD_ICODE_SEG

3.2.1 Int33_Critical_Init:

Here we allocate a real mode call back for the mouse driver. This is necessary for some of the mouse calls that need to be supported in each DOS VM.

3.2.2 Int33_Critical_Init Code:

```
BeginProc Int33_Critical_Init, PUBLIC

esi, OFFSET32 Int33_Pmode_Mapper

edx, edx
; no reference data



ecx, eax

ecx, 10h
; CX = CS to set into vector


edx, eax
```

```
edx, dx
; EDX=EIP to set into vector


eax, 33h
; interrupt number


Set_PM_Int_Vector
EndProc Int33_Critical_Init
```

3.2.3 Int33_Init:

A check is made in the [386Enh] section of the SYSTEM.INI file for the presence of the MouseSoftInit string. If it is set to true and the mouse is not a serial type, then all hardware mouse resets are reflected to mouse function 21h (software reset). Next, the EDX register is checked (it contains the data passed back to Windows in EDX in the real initialization phase) and the IRQ's for both the mouse and touchscreen are virtualized if their respective TSR's are present by calling the VMD_Set_Mouse_Type function.

3.2.4 Int33_Int Code:

```
BeginProc Int33_Init, PUBLIC
;
;high word of edx will contain Elodev info if any (-1 if not)



[Mouse_TS_Info],edx


edx, edx
; Q: Any int 33 at all?

SHORT I33_I_Alloc_Data_Area

I33_Exists:

[I33_Installed], True
; Remember we're here

;**********
```

```
eax,TRUE

edi,OFFSET32 I33_Wait_For_Untouch

esi,esi

Get_Profile_Boolean

al,al

short I33_nowait

[WaitForUntouch],TRUE
I33_nowait:


dh, VMD_Type_Serial
; Q: Is the mouse type serial?

SHORT I33_No_Fake_Init
;    N: Don't do this hack
;**********
;    Y: Only allow soft inits

eax, TRUE

edi, OFFSET32 I33_Soft_Init_INI

esi, esi
; [386enh] section


al, al
; Q: user want conversion?

short dont_convert_inits

al, 33
;   Y: convert call 0's to 33's
dont_convert_inits:

[I33_init_func], al
```

```
I33_No_Fake_Init:

dh, VMD_Type_Undefined
; Q: Did we get mouse type?

SHORT I33_I_Alloc_Data_Area


dl, dl
; Q: Is it on a valid IRQ

SHORT I33_I_Alloc_Data_Area


eax, dl
; EAX = IRQ number

ecx, dh
; ECX = Mouse type
;*****************************************************************
*********
ifdef MONMOUSE
;The extra info in edx is due to modifications for touchscreen support
;

edx,10h
;test for serial controller

edx,-1
;hi(edx)=-1 if no Elodev

short I33I_GotElodev

edx,edx
;pass 0 in edx if no Elodev
I33I_GotElodev:
;EDX = Elodev info
endif
;*****************************************************************
*********
```

```
; Tell the mouse driver
I33_I_Alloc_Data_Area:


eax, eax

SHORT I33_Init_Fatal_Error

[Int33_CB_Offset], eax

;*****************************************************************
*********
ifdef WINDOW_STATE

eax,VDD_Set_VMType

esi,OFFSET32 VTSD_Set_VMType

Hook_Device_Service

short
;Huh? It didn't work!

[VDD_VMType_Proc],esi
I33_I_not_hooked:
endif
;*****************************************************************
*********


eax, 33h

esi, OFFSET32 Int33_Soft_Int



esi, OFFSET32 I33_V86_Call_Back

edx, edx

Allocate_V86_Call_Back

[I33_V86_BP_Offset], ax
```

```
        ; save offset and...

        eax, 16

        [I33_V86_BP_Seg], ax
        ; ...real-mode segment



        I33_Init_Fatal_Error:



        EndProc Int33_Init
```

3.2.5 VMD_Set_Mouse_Type:

This procedure does all the virtualizing of the mouse and touchscreen hardware. Devices attached to a serial port do this by calling the VCD (Virtual Comm Device) and asking it to globalize that comm port. Devices not attached to a comm port, (or serial devices if the VCD has not been initialized yet) request to virtualize their IRQ via the VPICD. In the latter case, the VPICD returns a handle for that particular IRQ. This handle will be used when the focus changes from one VM to another.

3.2.6 VMD_Set_Mouse_Type Code:

```
BeginDoc
;*****************************************************************
*********
;   VMD_Set_Mouse_Type
;   DESCRIPTION:
;   ENTRY:
;
;
;
;
;
;
;
DL = IRQ number
;
If DH = -1 then
;
serial controller
```

```
;
else (if zero)
;
bus controller
;   EXIT:
;
;
;
;
;
;
  00 = Mouse already virtualized
;
  01 = Could not virtualize interrupt
;   USES:
;
;
;   Called from Int33_Init and also from VMD_API_Proc (called from mouse
;     driver via API service function 100h.) This should only execute on
;     the first of these calls. Int33_Init will not be called if there is
;     no int33h mouse driver in the system.
;=============================================================
===============
EndDoc

BeginProc VMD_Set_Mouse_Type, Service



[VMD_IRQ_Number], -1

VMD_SMT_2nd_Init

[VMD_Mouse_Type], cl
;=============================================================
===============
ifdef MONMOUSE



edx,[Mouse_TS_Info]

eax,dl
```

```
ecx,dh

edx,10h

edx,-1

short VMD_SMT_noElodev

edx,edx
VMD_SMT_noElodev:


edx,edx

short VMD_ts_done
;no DOS mouse driver

eax,dl
;eax = IRQ line

dh,dh
;bus or serial touchscreen controller?

short bus_controller
;if dh=0 then bus controller


[VTSD_IRQ_Number],al

eax

VCD_Get_Version

eax,eax

eax

short bus_controller
serial_controller:

al,5
;convert IRQ# to com port number
```

```
al
;IRQ4->COM1, IRQ3->COM2


[VTSD_COM_Port],al


eax,al

edx,edx

VCD_Set_Port_Global

short VMD_ts_done

bus_controller:

al,2
;map IRQ2 to IRQ9

short VMD_SMT_1

al,9
VMD_SMT_1:
;
[VMD_Mouse_Type],VMD_Type_PS2
;
short VMD_ts_done

[VTSD_IRQ_Number],al


edi,[VTSD_IRQ_Number]

[VTSD_IRQ_Desc.VID_IRQ_Number],di

edi,offset32 VTSD_IRQ_Desc

VPICD_Virtualize_IRQ

VMD_SMT_Cant_Virt_Int
```

```
        [VTSD_IRQ_Handle],eax

        [VTSD_Virt_IRQ],True
VMD_ts_done:

        endif

        ;=============================================================
        ================


        eax, eax

        SHORT VMD_SMT_Have_IRQ

        eax


        short VMD_SMT_Cant_Virt_Int
VMD_SMT_Have_IRQ:

        al, 2

        SHORT VMD_SMT_Valid_IRQ

        al, 9
VMD_SMT_Valid_IRQ:

        [VMD_IRQ_Number], al


        [VMD_Mouse_Type], VMD_Type_PS2

        SHORT VMD_SMT_Success


        [VMD_Mouse_Type], VMD_Type_Serial

        SHORT VMD_SMT_Virt_IRQ


        eax
```

```
eax, eax

eax

SHORT VMD_SMT_Virt_IRQ
;
;   Note:  This code assumes that COM1 is on IRQ4 and COM2 is on IRQ3.
;   It is not possible to have a serial mouse on COM3 or COM4 with this
;   code.
;

al, 5
; AL = -1 or -2

al


[VMD_COM_Port], al

eax, al
; EAX = # of COM mouse is on

edx, edx
; Port is globally owned


SHORT VMD_SMT_Success

VMD_SMT_Virt_IRQ:

edi, [VMD_IRQ_Number]

[VMD_IRQ_Desc.VID_IRQ_Number], di

edi, OFFSET32 VMD_IRQ_Desc


SHORT VMD_SMT_Cant_Virt_Int

[VMD_IRQ_Handle], eax

[VMD_Virt_IRQ], True
```

```
VMD_SMT_Success:



;
;   Could not virtualize interrupt.  Return error.
;
VMD_SMT_Cant_Virt_Int:


[VMD_IRQ_Number], -1


eax, eax

eax



;
;   Mouse already virtualized.
;
VMD_SMT_2nd_Init:

eax, eax



EndProc VMD_Set_Mouse_Type
```

3.2.7 Get_Mouse_Instance:

This procedure executes in the final phase of initialization. Its primary function is to instance any related DOS device drivers that it can find. In this particular case we are concerned with MOUSE.COM, MONMOUSE.COM and ELODEV.EXE. We search first for MOUSE.COM. If we find a program pointed to by the INT 33h interrupt vector, then we get it's size and location and instance it. In the case that the program that was just instanced happens to be MONMOUSE.COM and not MOUSE.COM, we check for a signature at an offset of 18h bytes past the INT 33h entry point. If it matches the string "MONMOUSE" (this is assembly language folks, no null terminator is implied by the double quotes), then the immediately following words contain MOUSE.COM's segment in memory and the address of ELODEV's PSP. If present, these values are used to instance both MOUSE.COM and ELODEV.EXE. If they are not present, that means that we

now have MOUSE.COM present and that we do not have MONMOUSE.COM. We know nothing about ELODEV yet. In the latter case, we must grovel around in the interrupt table and DOS memory to find ELODEV. If we find it, then it is instanced, if not, we quit.

3.2.8 Get_Mouse_Instance Code:

```
BeginDoc
;*****************************************************************
*********
;   Get_Mouse_Instance
;
;
;
;
;
;
;
;
;
(Device call-out API)
;
(VMD_Device_ID)
;
;
;   ASSUMES:
;
;
;
;
;   ENTRY:
;   EXIT:
;
;
;   USED:   All but EBP
;-----------------------------------------------------------------------
BeginProc Get_Mouse_Instance, PUBLIC



[I33_Installed], True

GMI_Done
```

```
[ebp.Client_AX], (W386_Int_Multiplex SHL 8) + W386_Device_Broadcast

[ebp.Client_BX], VMD_Device_ID

[ebp.Client_CX], VMD_CO_API_Test_Inst

eax, W386_API_Int
; Int 2Fh

; Ask Mr. VM for info

eax, [ebp.Client_CX]
; Non-zero means instanced



eax, eax

GMI_Done




edi, word ptr ds:[33h*4+2]

eax,edi
; Save in eax

edi
; arena seg

edi,4
; edi points to `driver'
```

```
[edi].arena_signature, 4dh

GMI_no_TSR_mouse
;   N: not TSR mouse

[edi].arena_owner, ax
; Q: correct owner?

GMI_no_TSR_mouse
;   N: not TSR mouse




ecx, [edi].arena_size
; get program size

ecx, 4
; size in bytes

edi,10h
; Point back to driver


esi,offset32 TSR_Mouse1_Inst


[esi.InstLinAddr],edi
; Address of item

[esi.InstSize],ecx
; instance whole program

[esi.InstType],ALWAYS_Field
```

```
eax,eax

GMI_MemErr
;=============================================================
================
ifdef MONMOUSE
;If monmouse.com was loaded along with mouse.com, then monmouse.com has been
;instanced instead of mouse.com. We must now instance monmouse.com as well as
;elodev.exe if it is present.
;
;get mouse.com address from monmouse if it's loaded.
;

ebx

ebx,(4*33h)
;get monmouse address (if it's there)

esi,word ptr[ebx+2]

esi,4

eax,word ptr[ebx]

esi,eax

esi,18h
;MonMouse signature from entry point

edi,offset32 MM_ID_Str

ecx,LENGTHMMIDSTR

cmpsb

short NoMonMouse


ebx,eax
;bx=mouse seg, hi(ebx)=elodev psp

;al=elodev int #
```

```
[ElodevInt],al

eax,bx

edi,eax

edi
;point to arena now

edi,4
;make 32-bit offset


[edi].arena_signature, 4dh

GMI_no_TSR_mouse

;   (should never happen here)

[edi].arena_owner, ax

GMI_no_TSR_mouse


ecx, [edi].arena_size

ecx, 4
; convert to size in bytes

edi,10h
; Point back to driver


esi,offset32 TSR_Mouse2_Inst

[esi.InstLinAddr],edi

[esi.InstSize],ecx

[esi.InstType],ALWAYS_Field

ebx,10h
```

```
;psp is in high word

edi,ebx
;save elodev psp

ebx

_AddInstanceItem,<esi,0>

eax,eax

GMI_MemErr

short mm_has_elodev
NoMonMouse:


ebx
;1st jumped around if no monmouse
; Get elodev address from monmouse.com or memory if mm not loaded
;

[ElodevInt],0

short SkipSearch
; Go grovel around in real mode memory for Elodev vector

FindElodev
;returns with carry set if not found

short NoElodev
;jump around this whole mess if

;  we don't find Elodev

[ElodevInt],al
SkipSearch:
;
; We should problably use Elodev's getinfo call to determine it's PSP but...
; What we will hope is that Elodev's PSP is immediately before its code seg
;   in memory, and that its memory block arena header is immediately before
;   that. Maybe later we will play around with allocating real mode memory
;   to pass to Elodev for a getinfo call.
```

```
;

edi,[ElodevInt]

edi,2

edi,2

edi,word ptr [edi]

edi,10h
;subtract size of the PSP
mm_has_elodev:

eax,edi

edi
;point to arena header

edi,4




[edi].arena_signature,4dh

short NoElodev
;   N: not TSR

[edi].arena_owner, ax
; Q: correct owner?

short NoElodev
;   N: not TSR mouse


ecx, [edi].arena_size
; get program size

ecx, 4
; size in bytes
```

```
edi,10h
; Point back to driver


esi,offset32 TSR_Mouse3_Inst

[esi.InstLinAddr],edi

[esi.InstSize],ecx

[esi.InstType],ALWAYS_Field

_AddInstanceItem,<esi,0>

eax,eax

short GMI_MemErr
NoElodev:
endif
;=============================================================
================

GMI_Done:



GMI_no_TSR_mouse:

esi,offset32 MS_Driver_Name

IFDEF DEBUG

short GMI10

GMI10:
ENDIF

short GMI_Done

GMI_MemErr:
        debug_out "Get_Mouse_Inst Error insuf mem"
        VMMCall Fatal_Memory_Error
EndProc Get_Mouse_Instance
```

3.3.0 MOUSE_TEXT

3.3.1 Initialize:

We are now executing code from MONMOUSE.DRV. This is its initialization phase. In the case of Enhanced mode Windows, this procedure is executed in a 16-bit protected mode segment (this is also true for Standard mode too but for now we're talking about VMMD's initialization). In a typically redundant Windows fashion, the mouse driver is checked for again. If a DOS mouse driver is present, a real mode call back is allocated via the DPMI. The returned address is passed to the DOS mouse driver as an event handler. If there is no DOS mouse driver in memory, a systematic check for a mouse is made. If a mouse is found, then the appropriate driver is copied into a Windows locked segment (one that is never swapped out of memory nor moved) and a pointer is returned to its enable and disable procedures. (The same is true in the case of the INT 33h DOS mode driver). Finally, after the appropriate driver is set up for use, a check is made to see if the VMD (VMMD in our case) has registered any API procedures. If so, it makes a call to the mouse API function 100h, passing it the mouse type and IRQ.

3.3.2 Initialize Code:

```
cProc
cBegin
;----------------------------------------------------------------------;
;
;
;
;----------------------------------------------------------------------;

bx,bx
;Current DOS mouse drivers implement

ax,INT33H_GETINFO

MOUSE_SYS_VEC

bh,6
;Major version in bh, started @ 6.??

try_mouse_reset

short sys_mouse_present

try_mouse_reset:
```

```
        ax,ax
        ;Check for DOS mouse driver

        INT33H_RESET

        MOUSE_SYS_VEC

        ax,ax
        ;Zero if no mouse installed

        check_inport

        sys_mouse_present:

        mouse_flags,MF_INT33H

        [mouse_type],ch

        check_I33:

        I33_init
        ;Use the installed mouse driver

        short got_mouse

        check_inport:

        inport_search
        ;Try to find InPort mouse

        check_bus
        ;InPort mouse wasn't found

        check_ps2
        ;Was found, didn't respond

        mouse_type, MT_INPORT
        ;InPort mouse

        short got_mouse
        ;Was found, responded

        check_bus:
```

```
mouse_type, MT_BUS

bus_search
;Next up, the old bus mouse

got_mouse

check_ps2:

mouse_type, MT_PS2

ps2_search
;PS/2 mouse port?

got_mouse

check_serial:

mouse_type, MT_SERIAL
;Assume serial mouse

serial_search

got_mouse

mouse_type, MT_NO_MOUSE

si,DataOFFSET device_int

short resize_ds

got_mouse:

;
;
;
;

ax

bx
```

```
di

es


di, di

es, di

ax, 1684h
;Get device API entry point

bx, VMD_DEVICE_ID

2Fh

ax, es

ax, di
;Q: Does VMD have API entry point?

copy_mouse_routines

cs
;Return to here after call to Win386

bx, offset vmd_call_done;virtual mouse driver

bx

es
;Call this SEG:OFF by doing a far

di
;return


ax, 100h
;Set mouse type & int VECTOR API call

bl, mouse_type
```

```
bh, vector
;
; Hard code this for now. In the future, dx needs to hold Elodev info but
; this must be gotten outside of Int33h routines and VMD needs to Instance
; Elodev outside of Int33 (In case there is no MOUSE.COM driver)
;
;
dx,[ElodevInfo]
;

dx,dx

BogusFarRetProc PROC FAR

;"Return" to VMD's API entry point
BogusFarRetProc ENDP

vmd_call_done:
copy_mouse_routines:

es

di

bx

ax
no_vmd:

mouse_flags,MF_MOUSE_EXISTS

di,DataOFFSET device_int

;
;
;
;
;
;
;


ds
```

```
;Destination is in Data

es


cs
;Source is in Code

ds


movsb

es

ds



si

di

ds


AllocDStoCSAlias,<ds>

IntCS,ax


ds

di

si

resize_ds:

si

;
;
```

```
;


ax,1
;Successful initialization
cEnd
```

3.4.0 VxD_CODE_SEG

3.4.1 VMD_API_Proc:

This procedure is the API entry point for the mouse device. It has only two functions. The first, function 0, merely returns a non-zero version number in the client's AX register. The second is function 100h. It serves to register the system's mouse type and IRQ with the VMD. It does this by calling VMD_Set_Mouse_Type. This will only have an effect on the system if MOUSE.COM is not resident in memory since VMD_Set_Mouse_Type will have already been called during the VMMD initialization phase.

3.4.2 VMD_API_Proc Code:

```
BeginProc VMD_API_Proc



[ebp.Client_AX], 0

SHORT VMD_API_Test_Set

[ebp.Client_AX], 300h
VMD_API_Worked:

[ebp.Client_Flags], NOT CF_Mask


VMD_API_Test_Set:

[ebp.Client_AX], 100h

SHORT VMD_API_Invalid

eax, [ebp.Client_BH]

eax

ecx, [ebp.Client_BL]
```

```
;*****************************************************************
*********
ifdef MONMOUSE
;
;

edx,[ebp.Client_DX]
endif
;*****************************************************************
*********


SHORT VMD_API_Worked

VMD_API_Invalid:

[ebp.Client_Flags], CF_Mask

EndProc VMD_API_Proc
```

3.4.3 VMD_Set_Focus:

VMD_Set_Focus gets called whenever focus is switched from one VM to another. In the case of a global or mouse specific set_focus call, the appropriate action is taken to insure system resources are in order before the focus is allowed to change. This mostly consists of completing any ongoing IRQ handling, and changing the VM handle to reflect the new VM. If "WaitForUntouch" was set to TRUE in the [386Enh] section of the SYSTEM.INI file, then a VMM call is made to mimic a real mode interrupt to MONMOUSE.COM. This will cause the system to wait until untouch occurs before completing the focus switch. What this does is to prevent touch events from bleeding through to the new VM. This is useful if the user is swapping to and from fullscreen DOS VMs from Windows. "WaitForUntouch" should be set to FALSE or omitted from the [386Enh] section if the user will be running Windowed DOS VMs or Winodws programs.

3.4.4 VMD_Set_Focus Code:

```
BeginProc VMD_Set_Focus

edx, edx
; Q: Critial set-focus?

SHORT VMD_SF_Focus_Change

edx, VMD_Device_ID
```

```
;    N: Q: Mouse?

VMD_SF_Exit
;
VMD_SF_Focus_Change:

;************************************
; attempt to wait for untouch here
;
ifdef

[VTSD_IRQ_Number],-1

short VTSD_SF_NoUntouch

ifdef




eax

ebx

ecx


ecx,ebx

eax,ebx
;hold onto current VM handle

Get_Next_VM_Handle

eax,ebx
;only one handle?

short l1
; Y: it must be the System VM

ecx,ebx
; N: hang on to it for later
```

```
l0:

Get_Next_VM_Handle

ebx,eax
;does it match the system VM?

short l1
; Y: then we're done, ecx has "last"

ecx,ebx
; N: save handle for next loop

short l0
;continue search with next handle
l1:

esi,ecx
;this is the handle we want (I hope)


ecx

ebx

eax


esi,[Int33_CB_Offset]

[esi.I33_CB_Window_State],-1

short VTSD_SF_NoUntouch
endif

[WaitForUntouch],False

short VTSD_SF_NoUntouch



[VMD_SF_Gate],0
```

```
short VMD_SF_2

[VMD_SF_GATE],1


eax

ebx

eax,65h
;call monmouse

ebx,2
;wait for untouch

33h

ebx

eax

[VMD_SF_GATE],0
VMD_SF_2:

VTSD_SF_NoUntouch:
endif
;
;************************************

ebx, [VMD_Owner]
; Get old/set new owner

[VMD_IRQ_Number], -1
; Q: Mouse virtualized yet?

VMD_SF_Exit
;    N: Nothing to do


Int33_Update_State
; Update old owner

ebx
```

```
ebx, [VMD_Owner]

Int33_Update_State
; Update new owner

ebx
;*****************************************************************
*********
ifdef
;
; Just like the mouse code, but for the touchscreen instead
;

[VTSD_IRQ_Number],-1

short VTSD_SF_Exit_a


esi,[VTSD_COM_Port]

esi,esi

short VTSD_SF_COM_Ctlr
VTSD_SF_1:

[VTSD_Virt_IRQ],True

short VTSD_SF_Exit

eax,[VTSD_IRQ_Handle]

VPICD_Get_Complete_Status

ecx,VPICD_Stat_Virt_Req

short VTSD_SF_Exit

VPICD_Clear_Int_Request

ebx,[VMD_Owner]

VPICD_Set_Int_Request
```

```
short VTSD_SF_Exit
VTSD_SF_COM_Ctlr:


eax,Set_Device_Focus

edx,VCD_Device_ID

ebx,[VMD_Owner]

System_Control
VTSD_SF_Exit:

VTSD_SF_Exit_a:
endif
;*****************************************************************
*********

esi, [VMD_COM_Port]

esi, esi

SHORT VMD_SF_COM_Mouse


[VMD_Virt_IRQ], True

SHORT VMD_SF_Exit


eax, [VMD_IRQ_Handle]


ecx, VPICD_Stat_Virt_Req

SHORT VMD_SF_Exit
;    N: Done!

;    Y: Clear old owner's

ebx, [VMD_Owner]
;
```

```
SHORT VMD_SF_Exit

VMD_SF_COM_Mouse:

eax, Set_Device_Focus

edx, VCD_Device_ID

ebx, [VMD_Owner]

VMD_SF_Exit:

EndProc VMD_Set_Focus
```