MS-DOS API EXTENSIONS FOR DPMI HOSTS

Version Pre-Release 0.04

MICROSOFT CONFIDENTIAL
TABLE OF CONTENTS

.Begin Table C.

.End Table C.

1. Introduction

While the DOS Protected Mode Interface (DPMI) specification does not support DOS calls from protected mode programs, extenders from many companies, including enhanced mode Windows 3.00, do support Int 21h and other standard DOS and BIOS interrupts commonly used in DOS extended programs.

This document defines an interface that allows the "MS-DOS" extensions to be detected, and provides guidelines on deviations of behavior from DOS calls made in real mode.

2. Detecting the Presence of MS-DOS Extensions

The MS-DOS extensions are supported by all versions of Enhanced mode Windows. Windows version 3.00 does not support the Int 2Fh API detection mechanism or API entry point, but does support all DOS and BIOS calls documented in this text. The correct code sequence for detecting the presence of the MS-DOS extensions is as follows:

```
MS_DOS_Name_String db "MS-DOS", 0

;
;       Note:  This assumes that the program has
;       already called the DPMI real to protected
```

```
;       mode switch entry point and is now running
;       in protected mode
;
Test_For_MS_DOS_Ext_Code:
        mov     ax, 168Ah
        mov     (e)si, OFFSET MS_DOS_Name_String
        int     2Fh
        cmp     al, 8Ah
        jne     Have_MS_DOS_Extensions

;
;       Check for presence of Enhanced Windows 3.00
;
        mov     ax, 1600h
        int     2Fh
        test    al, 7Fh
        jnz
Have_MS_DOS_Extensions_But_No_Call_Back

        (MS-DOS extensions are not present)
```

If the first Int 2Fh succeeds then ES:(E)DI will point to an API entry point that can be called by the program perform functions described in the next section. If the first Int 2Fh fails, but the program is running under Enhanced mode Windows 3.00 then the MS-DOS extensions are supported, but it is not possible to call the API entry point since it was not supported in Windows 3.00.

3. API Entry Point Functions

The MS-DOS extensions provide only two new services for protected mode programs. These are a get version function and a function that returns a selector

that points to the base of the current program's LDT.

To call the API entry point, programs must execute a far call to the address returned in ES:(E)DI from the function described on page .

3.1 Get MS-DOS Extension Version

This function returns the version of MS-DOS extensions supported by the DPMI host. Note that the value returned is not the version of DOS that the host is running on, it is the version of DPMI MS-DOS extensions that are supported by the host.

To Call

AX = 0000h

Returns

Carry flag is clear
AH = Major MS-DOS extension version number
AL = Minor MS-DOS extension version number

3.2 Get Selector to Base of LDT

Note that the DPMI host has the option of either failing this call, or to return a read-only descriptor. If the host returns a writeable LDT base descriptor then system security can be compromised, but performance of some programs (most notably the Windows kernel) can improve dramatically. This allows programs to

avoid ring transitions when examining or modifying LDT selectors. Note that even read-only access to the LDT reduces overhead a great deal in some circumstances. This would reduce the number of ring transitions for a get descriptor/set descriptor calls from two to one.

To Call

AX = 0100h

Returns

If function was successful:
Carry flag is clear
AX = Selector which points to base of current LDT

If function was not successful:
Carry flag is set

Programmer's Notes

- If this function succeeds, the caller must examine the access rights of the descriptor using a *verw* instruction to determine if the descriptor is writeable or is read-only.
- The selector returned by this function may be a GDT selector or an LDT selector. Programs should not assume that this selector exists in a particular descriptor table.
- At some point, the host may choose to move the LDT in linear memory. The host will be responsible for updating the descriptor for this selector. For this reason, all programs, including 32-bit flat model programs, should always access the LDT through this selector only. Never attempt to access the memory at a particular linear address. Never create an alias for this descriptor.

4. Notes for Microsoft Windows Program Writers

While both Standard and Ehnanced mode Windows support DPMI 0.9 with the MS-DOS extensions, Windows programs should not call any DPMI functions other than the following translation services:

AX = 0300h -- Simulate Real Mode Interrupt
AX = 0301h -- Call Real Mode Procedure With Far Return Frame
AX = 0302h -- Call Real Mode Procedure With Iret Frame
AX = 0303h -- Allocate Real Mode Call-Back Address
AX = 0304h -- Free Real Mode Call-Back Address

No other DPMI services, including the state save and raw mode switch translation services, should be called by Windows programs or DLLs. The Windows kernel uses DPMI to allocate memory, manipulate descriptors, and lock pages. All Windows programs should call the appropriate kernel functions to perform these operations. The following are hints for Windows progarmmers on ways to avoid calling DPMI:

Windows programs should call the GetWinFlags function to determine if they are running in protected mode instead of using the DPMI Get Version call.

Windows ~~applications and DLLs~~ programs should use the function calls supplied by the Windows kernel to manipulate selectors instead of using DPMI services. These are:

PrestoChangoSelector (Documented in SDK as ChangeSelector)
AllocSelector
AllocDStoCSAlias
FreeSelector

Windows ~~applications~~ programs should not use the DPMI DOS Memory Managment services. The Windows kernel has two functions named

GlobalDOSAlloc and *GlobalDOSFree* that should be used by Windows applications and DLLs for allocating and freeing DOS addressable memory. Under normal circumstances the Windows kernel will have allocated all free DOS addressable memory and so the DPMI functions will always fail.

All requests to allocate, reallocate, free, or lock memory should be made through kernel functions. For example, *GlobalAlloc, GlobalReAlloc, GlobalFree, etc.*

Known Bugs and Workarounds For Windows 3.00 DPMI

Windows enhanced mode version 3.00 was the first implementation of DPMI 0.9.
~~~~

Int 10h Translation

Windows incorrectly maps Int 10h AH=0Eh (write character TTY) as a write string call (AH=13h). This means that write string does not work from protected mode. However, write character will still work as long as the caller's ES:BP point to any valid data and CX=1. Since Windows only copies the string pointed to by ES:BP and does not change any other registers then AX and BX are passed through to the BIOS correctly for the write character operation. There is no workaround for write string other than using the DPMI translation services.

5. 32-bit programs
~~~~

Many implementations of the MS-DOS extensions, including Enhanced mode Windows, support 32-bit programs on 80386 and 80486 processors. In most cases, the APIs are exactly the same as for 16-bit programs except that pointers are 48-bits. That is, they consist of a segment and a 32-bit offset. DOS read and write calls (AH=3Fh and 40h) the count register (ECX) is also extended to 32 bits. This allows 32-bit programs to perform DOS reads of greater than 64K bytes.

Hosts that support 32-bit programs will ignore and not modify the high word of an extended register unless the DOS or BIOS call returns a pointer (such as the Get Interrupt Vector call). The extended portion of EAX will also be modified on DOS file read and write calls. All other calls will leave the high word of the extended registers unmodified. If code in real mode modifies the extneded portion of a register then that value will be returned to the protected mode DPMI program.

6. DOS and BIOS Calls

Programs running in protected mode under a DPMI host that supports the MS-DOS extensions can make DOS and BIOS calls just as they would when running in real mode (with some minor exceptions). The only difference is that the code and data of the program can reside in memory above one megabyte, and all pointers use protected mode selectors instead of real mode segments to point to data. As in real mode, DOS is called by executing Int 21h.

When an Int 21h is executed in protected mode any data that is required by that call will be copied to a buffer in real mode and then the real mode DOS will be called with a pointer to the copied data. The host is responsible for copying data and translating pointers -- DPMI programs that use the MS-DOS extensions need not worry about how

For example, lets look at a series of DOS calls to open and read a file:

```
;
; Open the file with read-only access
;
mov     ax, 3D00h
mov     dx, OFFSET File_Name_String
int     21h
jc      Error

;
; Read the first 6000h bytes of the file
;
mov     bx, ax
mov     ah, 3Fh
mov     cx, 6000h
mov     dx, OFFSET Read_Buffer
int     21h
jc      Error

;
; Close the file
;
mov     ah, 3Eh
int     21h
```

The open file call takes an ASCIIZ pathname as a parameter. Since the address passed to the protected mode Int 21h handler is a selector:offset (DS contains a selector to program's protected mode data segment), real mode DOS would not be able to access the data. The protected mode DOS translator copies the string into a real mode buffer and then calls DOS in real mode with DS:DX pointing to the real mode buffer. The values in other registers are not modified so the call number in AX will not be changed. When real mode DOS returns, the values returned in the flags and all non-segment registers will be returned to the protected mode program. In this case, the carry flag indicates an error and the

file handle will be returned in AX.

The second DOS call reads part of the file into a buffer. Once again, since the buffer can not be accessed by real mode DOS, the data must be copied through a buffer. The data will be read into a real mode buffer and copied into the protected mode memory. Since the real mode buffer is usually smaller than 6000h bytes the translator will probably have to break the read into several smaller pieces. However, all the copying of data and multiple reads will be invisible to the caller. The read will behave exactly as if the code were being executed in real mode.

For the final call (close file), the protected mode Int 21h hook just reflects the interrupt to real mode without translating anything. Since the DOS close file command has no pointer parameters, no translation is necessary.

The sample code above is 16-bit code and would work on an 80286 DPMI implementation. However, DPMI supports 32-bit programs on 80386 and 80486 processors. The only difference between 32-bit and 16-bit programs that pointers require a 32-bit offset in the extended register (EDX instead of DX) and that DOS read and write calls take a 32-bit count in ECX. The 32-bit equivalent of the sample code is provided below.

```
;
; Open the file with read-only access
;
mov     ax, 3D00h
mov     edx, OFFSET File_Name_String
int     21h
jc      Error

;
; Read the first 6000h bytes of the file
;
```

```
mov     bx, ax
mov     ah, 3Fh
mov     ecx, 6000h
mov     edx, OFFSET Read_Buffer
int     21h
jc      Error

;
; Close the file
;
mov     ah, 3Eh
int     21h
```

7. DOS State on Entry Into Protected Mode

A host that supports the MS-DOS extensions to DPMI maintains additional state information for each client program. When a program enters protected mode on an extended DPMI host, the state will be as described in the DPMI 0.9 specification with the following additions:

1. The protected mode DTA will be mapped to the real mode DTA when the program enters protected mode. If the DTA address has not been changed from the default at offset 80h in the PSP, then the DTA selector will be the same as the PSP selector. Otherwise, a new descriptor will be allocated. Do not modify or free the DTA descriptor. Use DOS call 2Fh to obtain the address of the DTA.

2. The DOS Ctrl+Break (Int 23h) and critical error (Int 24h) interrupt handlers will be set to default handlers as described on page .

8. Supported DOS Calls

This section describes the differences between real mode DOS and BIOS calls and those made in protected mode. Obviously, pointers use protected mode selectors instead of real mode segments. 32-bit programs must use 48-bit pointers and the size parameters for some calls such as file reads and writes will be 32-bit. For example DOS reads and writes use ECX for the size parameter instead of CX for 32-bit programs. See page for an example of 32-bit DOS code.

All DOS calls that are not mentioned in this section should work exactly as documented in *The MS-DOS Encyclopedia*. The minimum assumed DOS version is 3.xx.

8.1 DOS Calls That Are Not Supported

The following DOS calls are not supported in protected mode. They will fail if called.

8.1.1 Unsupported Interrupts

INT	Description
20h	Terminate Program
25h	Absolute Disk Read
26h	Absolute Disk Write
27h	Terminate And Stay Resident

8.1.2 Unsupported Interrupt 21h DOS Functions

AH	Description
~~00h~~	~~Terminate Process~~
0Fh	Open File with FCB
10h	Close File with FCB
14h	Sequential Read
15h	Sequential Write
16h	Create File with FCB
21h	Random Read
22h	Random Write
23h	Get File Size
24h	Set Relative Record
27h	Random Block Read
28h	Random Block Write

8.2 Calls That Behave Differently In Protected Mode

Function 00h -- Terminate Process

This call should never be executed by standard DPMI appliations. The Windows kernel requires a special version of process termination so that it can close Windows applications. DOS call 0 has been redefined for this purpose. Hosts must implement this function, but application writers should never call it. See page for more information.

8.2.1 Functions 25h and 35h -- Set/Get Interrupt Vector

These functions will set or get the protected mode interrupt vector. They

can be used to hook hardware interrupts (such as the timer or keyboard interrupt) as well as hooking any software interrupts your program wishes to monitor. With a few exceptions, software interrupts issued in real mode will not be reflected to protected mode interrupt handlers. However, all hardware interrupts will be reflected to protected mode interrupt handlers before being reflected to real mode. See page for more information on hooking interrupts in protected mode programs.

32-bit programs must use 48-bit pointers and must use the *iretd* instruction to return from interrupts.

8.2.2 Function 31h -- Terminate and Stay Resident

The value in DX specifies the number of paragraphs of real mode memory to reserve for the program. The reserved memory must include any memory allocated for the DOS extender when the program switched to protected mode. All protected mode memory allocated through Int 31h or protected mode calls to DOS function 48h will not be deallocated -- It is up to the T&SR program to free any unneeded protected mode memory.

Note that protected mode "pop-up" programs will need to be very careful when saving and restoring the current DOS state to preserve both the real mode and protected mode states of the current task. This type of program must call the state save functions documented on page . It must also save and restore the DTA address in both protected mode and real mode. To get the current real mode DTA the program must use the translation services to call the real mode DOS Get DTA function.

8.2.3 Function 32h -- Get Current Country Data

This call returns a 34-byte buffer that contains a dword call address at offset 12h that is used for case-mapping. This dword will contain a real mode address. If you wish to call the case-mapping procedure you will

need to use the DPMI translation service to simulate a real mode far call (see page).

8.2.4 Functions 3Fh and 40h -- Read/Write File or Device

32-bit programs must specify the size of the read or write in the ECX register instead of the CX register. This allows for read and writes of greater than 64K. The returned count will be in EAX instead of AX. ~~Note that~~ 16-bit programs are still limited to reads of 0FFFFh bytes and the count will be retuned in AX.

8.2.5 Function 44h, Subfunctions 02h, 03h, 04h, and 05h

These IOCTL subfunctions are used to receive data from a device or send data to a device. Since it is not possible to break the transfers into small pieces, the caller should assume that a transfer of greater than 2K bytes will fail unless the address of the buffer is in the DOS addressable first megabyte.

8.2.6 Function 44h, Subfunction 0Ch

Only minor function codes 45h (get iteration count) and 65h (set iteration count) are supported from protected mode. Extensions of this IOCTL for code-page switching (functions function codes 4Ah, 4Ch, 4Dh, 6Ah, and 6Bh) are not supported for protected mode programs. You must use the translation services if you need to use this IOCTL to switch code pages (see page).

8.2.7 Functions 48h, 49h and 4Ah

It is recommended that all memory allocations be made through the DPMI memory allocation services (see page). However, these DOS calls will work in protected mode.

DOS memory allocation calls issued by a protected mode program will allocate extended memory. This memory is not addressable by real mode DOS.

32-bit programs must specify the number of paragraphs to allocate in the EBX register. This allows for memory allocations of greater than 1Mb.

To determine the size of the largest available block set (E)BX to -1 and call function 48h.

8.2.8 Function 4Bh -- Load and Execute Program

This function can not be used to load a program overlay from a protected mode program. However, you can execute another program using subfunction 0. The program will be executed in real mode. However, the child program can enter protected mode using the DPMI real to protected mode switch API.

The environment pointer in the exec parameter block is ignored and should be set to 0 by 16-bit programs. 32-bit programs should place an fword pointer to the command tail at offset 0. (~~~MORE DETAIL~~~)

8.2.9 Function 4Ch -- Terminate Process with Return Code

This is the only supported form of program termination for protected mode DOS programs. It behaves exactly as it would in real mode. It will free any memory that was allocated by the protected mode program, and return to the parent program. The protected mode Int 23h and Int 24h vectors will be restored to the same value as ~~~~~~~.

8.2.10 Function 65h -- Get Extended Country Information

This function is supported for protected mode programs. However, all

doubleword parameters returned will contain real mode addresses. This means the case conversion procedure address and all pointers to tables will contain real mode segment:offset addresses. You must use the translation services to call the case conversion procedure in real mode.

9. Supported BIOS Calls

All BIOS calls that pass parameters in the AX, BX, CX, DX, SI, DI, and BP registers, and that contain no pointers or segment values in these registers will be supported by all implementations of DPMI (provided, of course, that the API is supported by the machine's BIOS).

For the sake of clarity and completeness, this document contains a list of every BIOS API that will be supported, including those that are register based APIs. APIs that are not register based are documented individually.

9.1 Interrupt 10h -- Video

9.1.1 Register Based Functions (supported):

AH	Description
00h	Set Mode
01h	Set Cursor Type
02h	Set Cursor Position

03h	Read Cursor Position
04h	Read Light Pen Position
05h	Select Active Display Page
06h	Scroll Active Page Up
07h	Scroll Active Page Down
08h	Read Attribute/Char at Cursor Position
09h	Write Attribute/Char at Cursor Position
0Ah	Write Character at Cursor Position
0Bh	Set Color Palette
0Ch	Write Dot
0Dh	Read Dot
0Eh	Write Teletype to Active Page
0Fh	Read Current Video State
1Ah	Read/Write Display Combination Code

9.1.2 Function 10h -- Set Palette Registers

All subfunctions of this API are supported.

9.1.3 Function 13h -- Write String

This call is supported provided the string is not longer than 2K bytes.

9.1.4 Functions that are not Fully Supported

Many of these functions have APIs that are register based. All register based calls are supported. However, any APIs that contain pointer parameters are not supported under DPMI.

AH	DESCRIPTION
11h	Character Generator
12h	Alternate Select

14h Load LCD Character Font
15h Return Physical Display Parameters
1Bh Return Functionality/State Info
1Ch Save/Restore Video State

9.2 Interrupt 11h -- Equipment Determination

Since interrupt 11h is register based, it will be supported by all implementations of DPMI.

COMPATIBILITY WARNING: EISA machines will destroy the high word of the EAX register on machines with 80386 CPUs.

9.3 Interrupt 12h -- Memory Size Determination

Since interrupt 12h is register based, it will be supported by all implementations of DPMI.

9.4 Interrupt 13h -- Diskette / Fixed Disk Interface

Application programs have no reason to use this interrupt. In any case, since direct disk access will not be allowed most implementations of DPMI, programs can not rely on these functions.

9.5 Interrupt 14h -- Asynchronous Communications

Since all asynchronous communication APIs are register based, they are all supported.

AH	Description
00h	Initialize Communications Port
01h	Send Character
02h	Receive Character
03h	Read Status
04h	Extended Initialize
05h	Extended Communications Port Control

9.6 Interrupt 15h -- System Services

9.6.1 Register Based Functions (supported):

ah	description
00h	Turn Cassette Motor On
01h	Turn Cassette Motor Off
40h	Read/Modify Profiles
42h	Request System Power-Off
43h	Read System Status
44h	Activate/Deactivate Internal Modem
80h	Device Open
81h	Device Close

82h	Program Termination
84h	Joystick Support
86h	Wait
87h	Extended Memory Size
C3h	Enable/Disable Watchdog Time-Out
C4h	Programmable Option Select (POS)

9.6.2 Function C0h -- Return System Configuration Parameters

This call is supported. The pointer to the system descriptor vector will be in ES:EBX for 32-bit programs.

9.6.3 Function C1h -- Pointing Device Interface

This interface will not be supported under most implementations of DPMI. Programs that use a mouse are encouraged to use the Int 33h interface documented on page .

9.6.4 Functions that are Not Supported:

ah	Description
02h	Read Blocks from Cassette
03h	Write Blocks to Cassette
0Fh	Format Unit Periodic Interrupt
21h	Power-On Self-Test Error Log
41h	Wait for External Event
4Fh	Keyboard Intercept
83h	Wait Event
85h	System Request Key Pressed
87h	Move Block
89h	Switch Processor to Protected Mode
90h	Device Busy
91h	Interrupt Complete

C1h	Return Extended BIOS Data Area Seg

9.7 Interrupt 16h -- Keyboard

Since all keyboard APIs are register based, they are all supported.

ah	description
00h	Keyboard Read
01h	Keyboard Status
02h	Shift Status
03h	Set Typematic Rate
04h	Keyboard Click Adjustment
05h	Keyboard Write
10h	Extended Keyboard Read
11h	Extended Keyboard Status
12h	Extended Shift Status

9.8 Interrupt 17h -- Printer

Since all printer APIs are register based, they are all supported.

ah	description
00h	Print Character
01h	Initialize the Printer
02h	Read Status

9.9 Interrupt 1Ah -- System-Timer and Real-Time Clock

9.9.1 Register Based Functions (supported):

AH	Description
00h	Read System-Timer Time Counter
01h	Set System-Timer Time Counter
02h	Read Real-Time Clock Time
03h	Set Real-Time Clock Time
04h	Read Real-Time Clock Date
05h	Set Real-Time Clock Date
07h	Set Real-Time Clock Alarm
08h	Set Real-Time Clock Activated Power On
09h	Read Real-Time Clock Alarm Status
0Ah	Read System-Timer Day Counter
0Bh	Set System-Timer Day Counter

9.9.2 Function 06h -- Set Real-Time Clock Alarm

Although this call is register based and therefore requires no translation before being passed to real mode, the caller is required to hook the real mode interrupt 4Ah vector to intercept the alarm interrupt.

10. Mouse Driver Interface

DPMI supports a subset of the standard Int 33h mouse driver interface for protected mode programs. It may be necessary for programs to call the mouse

driver in real mode either before switching to protected mode or by using the translation services to completely save and restore the mouse driver state.

10.1 Mouse Calls that Are Supported

10.1.1 Register Based Calls

ah	description
00h	Reset Mouse and Get Status
01h	Show Mouse Pointer
02h	Hide Mouse Pointer
03h	Get Mouse Position and Button Status
04h	Set Mouse Pointer Position
05h	Get Button Press Information
06h	Get Button Release Information
07h	Set Horizontal Limits for Pointer
08h	Set Vertical Limits for Pointer
0Ah	Set Text Pointer Type
0Bh	Read Mouse Motion Counters
0Dh	Turn on Light Pen Emulation
0Eh	Turn off Light Pen Emulation
0Fh	Set Mickeys to Pixels Ratio
10h	Set Mouse Pointer Exclusion Area
13h	Set Double Speed Threshold
15h	Get Mouse Save State Buffer Size
1Ah	Set Mouse Sensitivity
1Bh	Get Mouse Sensitivity
1Ch	Set Mouse Interrupt Rate
1Dh	Select Pointer Page
1Eh	Get Pointer Page

20h	Enable Mouse
21h	Reset Mouse Driver
22h	Set Language for Mouse Driver
23h	Get Language Number
24h	Get Mouse Information

10.1.2 Function 09h -- Set Pointer Shape

This call works exactly as it would in real mode. However, 32-bit programs must use ES:EDX to point to the pointer image buffer.

10.1.3 Function 0Ch -- Set User-Defined Mouse Event Handler

32-bit programs must call this function with ES:EDX = Selector:Offset of handler and will need to execute a 32-bit far return to return from the event call-back. For both 16-bit and 32-bit programs the protected mode DS will not point to the mouse driver data segment when the event handler is called. Do not rely on any specific value in the DS register when the event handler is called.

10.1.4 Functions 16h and 17h -- Save/Restore Mouse Driver State

These calls work exactly as they would in real mode. However, 32-bit programs must use ES:EDX to point to the buffer.

10.2 Mouse Calls that Are Not Supported

ah	description
14h	Swap User-Defined Event Handlers
18h	Set Alternate Event Handler
19h	Get Address of Alternate Event Handler

1Fh Disable Mouse Driver

11. NETBIOS

Some implementations of DPMI support NetBIOS calls in protected mode, although this is not required. Programs can determine wether or not the current DPMI implementation supports NetBIOS calls from protected mode by examining the flags returned from the Get Version call (see page). A program that uses NetBIOS and needs to run on any DPMI implementation will need to use the translation services documented on page .

32-bit programs can call NetBIOS if it is supported. In this case, ES:EBX must be used to point to the Network Control Block (NCB). However, pointers within the NCB are restricted to a 16-bit offset. Therefore, all buffers must reside within the first 64K of the buffer's segment.

12. Interrupts 23h and 24h

DOS provides two interrupts that programs can hook to handle Ctrl+Break and critical device errors. These interrupts are reflected to protected mode programs if the program hooks the interrupt in protected mode. Although both of these interrupts can be used to terminate a program in real mode, they can not be used to terminate protected mode programs.

Protected mode Int 23h and Int 24h handlers must reside in locked memory and all data that they touch must also be locked. This is required to prevent a page fault from occurring at a time when DOS can not be called to read the data in

from disk. These interrupt handlers will always be called on a locked stack.

12.1.1 Interrupt 23h

Interrupt 23h is the DOS Ctrl+Break interrupt. This interrupt will be reflected to protected mode if a protected mode interrupt handler is installed. Unlike real mode DOS, the interrupt handler must return. This interrupt can not be used to terminate a protected mode program and the value of the carry flag will be ignored when the interrupt returns.It is suggested that you set a flag in your program that will be examined later and then execute an *iret* to return from the interrupt.

Int 23h is ignored for protected mode programs unless it is hooked in protected mode.

12.1.2 Interrupt 24h

When Int 24h is called in protected mode, SS:(E)BP will point to a standard Int 24h stack frame. 32-bit programs will be called with a 32-bit iret frame but the rest of the stack frame will be exactly as that of a 16-bit Int 24h. The values on the stack will contain the values passed to DOS in real mode. Therefore, the segment register values on the stack will be real mode segments, not selectors.

Protected mode Int 24h handlers must iret. Since programs can not be terminated by a critical error handler, an attempt to abort the program (returning with AL=02h) will be ignored and the DOS call will be failed. That is, a return of AL=02 will be converted to AL=03 by the host.

The default Int 24h handler will fail all critical errors. Therefore, unless the protected mode Int 24h vector is hooked, all DOS calls that generate a critical error will fail.

13. Additional Host Support Required to Run Microsoft Windows in Protected Mode

13.1 "Undocumented" DOS Calls That Must Be Supported

The Windows kernel calls several DOS functions that are not documented in the *MS-DOS Encyclopedia*. Since these calls are required to run Windows, DPMI servers that support the MS-DOS extensions must support them in protected mode.

AH=1Fh and AH=32h

These calls return a pointer in DS:BX that points to a table inside of DOS. ~~~(WHAT ARE THEY???)~~~

AH=50h -- Get PSP

This call is identical to DOS call 62h. The host must return a selector that points to the current PSP in BX.

AH=51h -- Set PSP

This call is used by the Windows kernel to switch PSPs when running multiple Windows applications. The call will be made with BX=Selector of PSP to set as current PSP. The PSP memory is guaranteed to be in the DOS addressable 1 Mb and paragraph aligned. The host must convert the base address of the selector into a real mode segment and pass the call on to real mode DOS.

AH=52h -- Get Pointer to List of Lists

This call returns a pointer in ES:BX that points to a table inside of DOS.

AH=53h -- Translate BIOS Parameter Block

This call takes ~~~~ FIND OUT ABOUT WHICH ARE INPUT AND WHAT IS RETURNED~~~

AH=55h -- Create PSP

This call is identical to DOS call 26h. The caller passes a selector to a 100h byte paragraph aligned block of memory in the first Mb of linear address space. The host must convert the selector base address into a real mode segment and pass the call on the real mode DOS. The Windows kernel will be responsible for creating the environment and modifying the envionrment pointer in the newly created PSP.
~~~(CHECK OUT ALL IOCTLS)~~~~

AH=5Dh -- Server DOS Calls

~~~ DOCUMENT ALL THAT NEED TO BE SUPPORTED ~~~~

13.2 Special handling of DOS call 0

Int 21h, AH=00h is the original DOS 1.00 terminate call. DPMI programs should always terminate using DOS call 4Ch. However, since Windows applications have seperate PSPs, the Windows kernel must have a mechanism for terminaing the Windows applicationsso that networks can clean up, and DOS can close open files. The standard AH=4Ch terminate call is inappropriate for this purpose

since it would cause the kernel to be terminated.

Under the MS-DOS extensions of DPMI, DOS call 0 terminates the current PSP and then returns to the caller (which will be the Windows kernel). When the host intercepts an Int 21h, AH=0 it should patch the current PSP so t~~

GDT Selector 40h Must Be Reserved

Many Windows hardware drivers use the constant selector value 40h to access the BIOS RAM area at 0040:0000-0040:02FF. For many 3rd party drivers, GDT selector 40h must be mapped with a linear address of 400h and a limit of 2FEh. This compatibilty constraint applies to other DPMI applications besides Windows. Hosts should define selector 40h as specified here or should be capable of handling a GP fault when a program attempts to load a segment register with 40h and substitute an appropriate selector.

NOTE TO SOFTWARE DEVELOPERS: DON'T USE THIS! Windows drivers and applications should link to the kernel defined absolute export variable __0040h. DPMI applications should use Int 31h function 0002h. However, for compatibility with Windows and other DPMI applications, hosts should map selector 40h as specified above.

Windows Must Run At Ring 3

Because of the way the Windows kernel manipulates memory handles, it must always run at Ring 3.

13.1 Direct Disk I/O

Windows Enhanced and Standard mode support interrupts 25h, 26h, and a subset of Int 13h diskette functions so that the File Manager can format diskettes. Application programs have no reason to use either of these interrupts. DPMI hosts must support these functions if they wish allow diskettes to be formatted using the Windows file manager. In any case, if these functions are not supported then appropriate error codes should be returned when these APIs are called.

Windows supports a subset of the diskette functions that are used by the File Manager to format diskettes. (~~WHAT ARE THEY~~)