DOS PROTECTED MODE INTERFACE (DPMI)
SPECIFICATION

Protected Mode API For DOS Extended Applications
Version 0.9
Printed April 23, 1990

TABLE OF CONTENTS

1. INTRODUCTION

The DOS Protected Mode Interface (DPMI) was defined to allow DOS programs to access the extended memory of PC architecture computers while maintaining system protection. DPMI defines a specific subset of DOS and BIOS calls that can be made by protected mode DOS programs. It also defines a new interface via software interrupt 31h that protected mode programs use to allocate memory, modify descriptors, call real mode software, etc. Any operating system that currently supports virtual DOS sessions should be capable of supporting DPMI without affecting system security.

Some DPMI implementations can execute multiple protected mode programs in independent virtual machines. Thus, DPMI applications can behave exactly like any other standard DOS program and can, for example, run in the background or in a window (if the environment supports these features). Programs that run in protected mode also gain all the benefits of virtual memory and can run in 32-bit flat model if desired.

Throughout this document, the term "real mode" software is used to refer to code that runs in the low 1 megabyte address space and uses segment:offset addressing. Under many implementations of DPMI, so called real mode software is actually executed in virtual 8086 mode. However, since virtual 8086 mode is a very close approximation of real mode, we will refer to it as real mode in this document.

DPMI services are only available to protected mode programs. Programs running in real mode can not use these services. Protected mode programs must use the service described on page 17 to enter protected mode before calling Int 31h services.

All Int 31h functions will modify flags and the AX register. All other registers will be preserved unless they are specified as return values. Unsupported calls will return with the carry flag set. Since Int 31h is set up as a trap gate, the interrupt flag will not be modified by any Int 31h calls except for memory management and interrupt flag management calls. All memory management calls may enable interrupts. Interrupt flag management calls will modify the interrupt flag as specified by the call. All Int 31h services are reentrant.

Some implementations of DPMI can run 32-bit 80386 specific programs. DPMI functions that take pointers as parameters will use the extended 32-bit registers for offsets (for example, ES:EDI instead of ES:DI) when running 32-bit mode programs. The high word of the 32-bit registers will be ignored when running 16-bit protected mode programs.

DPMI services are provided by what will be referred to as the DPMI host program. The program(s) that use DPMI services are called DPMI clients. Generally, DPMI clients are two categories:

- Extended Applications
- Applications that use DPMI directly

It is believed that most DPMI applications will be extended applications. Extended applications are bound with an extender that is the actual DPMI client and the application calls extender services that then are translated by the client into DPMI calls. The advantage of an extended application over one that calls DPMI services directly is that generally an extender will support more than just DPMI. In fact it is recommended that extenders look for extension services in the following order:

- DPMI
- VCPI/EMS
- XMS
- Top-down (Int 15h)

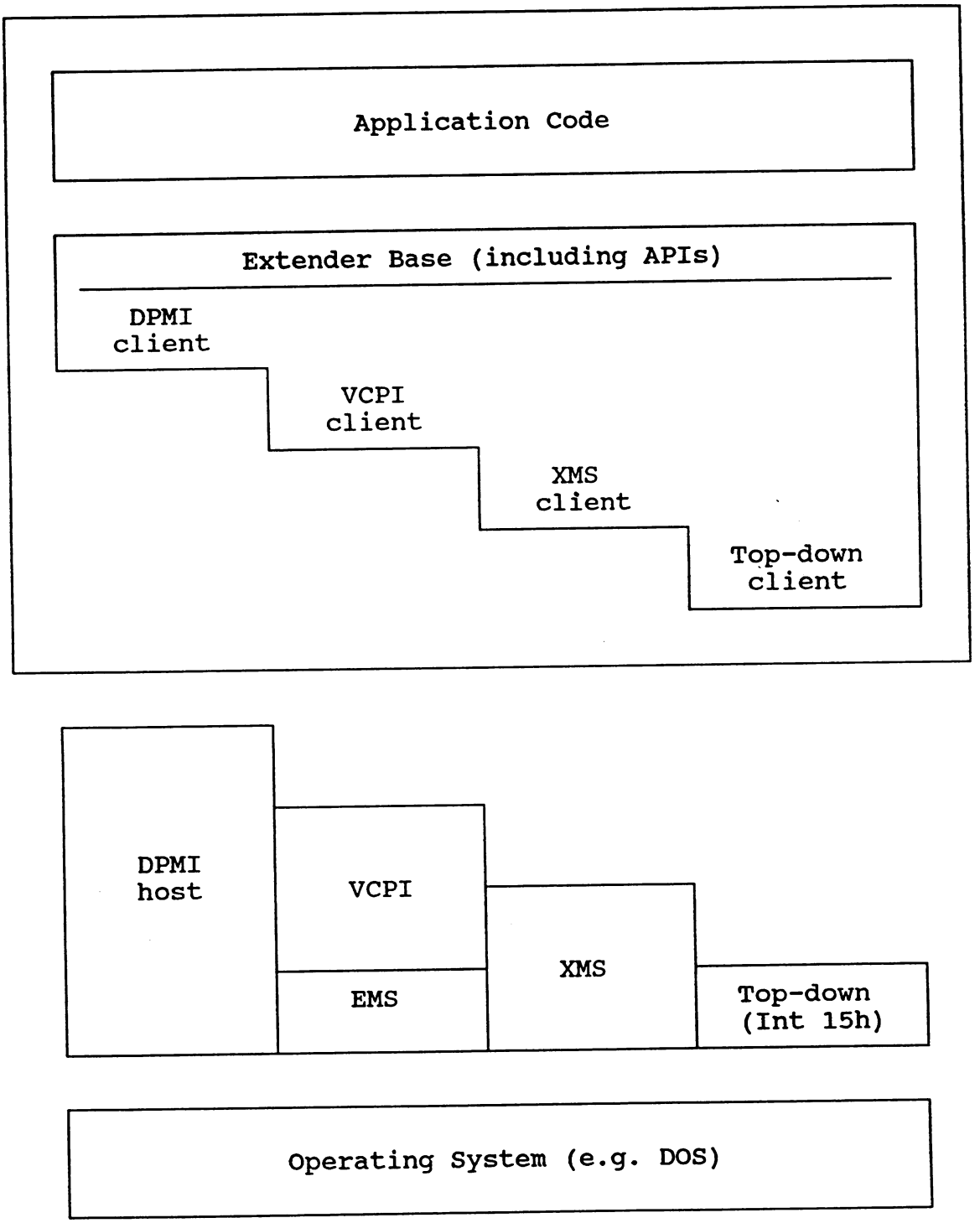
An extender can provide a single set of APIs to the actual application and then translate them to the services that are provided. Where the host extension services are "lacking" in a particular function the extender must provide that function for the application.

Figure 1 on page 3 shows a picture of how this works. The application code sits on top of a set of base extender functions and APIs. The extender then has separate modules for each type of extension service and code to "fill in the slack" where services are lacking. An example of a typical extender service is protected mode program loading. The actual shipped application is the application code bound in with the extender and all of its styles of client support.

The host support is generally an extension of the base OS functions or a device driver used to extend the base OS functions.

This document is intended to provide a definition of the DPMI services that a DPMI host would be required to implement and that a DPMI client would use.

Figure 1. Application/Extender/Client/Host/OS structure



2. GENERAL NOTES FOR PROTECTED MODE PROGRAMS

There are a few basic differences between real mode and protected mode that need to be addressed to convert a real mode program to run in protected mode.

Programs run at a protection level that prevents them from executing privileged instructions such as *lgdt*, *lidt*, etc. The DPMI interface is the only method application programs have for modifying system structures such as descriptors.

While DPMI defines a specific set of functions that will be supported by all implementations, there may be minor differences in individual implementations. Programmers should refer to the notes for their DPMI implementation for documentation on detecting the presence of and calling vendor specific extensions. However, any application that is written to adhere only to standard DPMI calls should work correctly under all implementations of DPMI.

2.1 Virtual DOS Environments

Many DPMI implementations are simulated "virtual DOS" sessions. In other words, the DOS interface and environment presented to the program are not actually the native interface of the operating system. Hardware interrupts, I/O, and processor exceptions will be virtualized by the operating system. This means, for example, that a DPMI program may receive a simulated keyboard interrupt and read simulated I/O from the keyboard controller ports.

In these environments, actual hardware interrupts will be handled by the operating system. The physical interrupts will be invisible to the DPMI application program. If the operating system so chooses, it may reflect a virtual interrupt to the DPMI program. The DPMI program does not need to know, nor should it care, if this is the case. From the program's point of view, the interrupt looks exactly like a "real" interrupt. The operating system will also virtualize I/O to the interrupt controller ports and any other simulated devices.

There are basically three levels of virtualization that DPMI implementations can provide:

2.1.1 No Virtualization

In general, stand-alone single tasking DPMI implementations will not virtualize any hardware devices. These hose extension programs will execute as standard DOS real mode drivers or programs. Extenders which use the services provided by these DPMI host drivers will translate protected mode DOS calls to real mode DOS calls. Normally these extenders will invoke DPMI services to return the processor to real mode (instead of virtual 8086 mode) when calling DOS.

2.1.2 Partial Virtualization

Some environments that execute under DOS will virtualize hardware devices, provide virtual memory, or provide other services that require virtualization of some hardware devices. Under these environments, DPMI applications will always run at a non-privileged ring (usually ring 3). Some or all hardware interrupts will be virtualized, some or all I/O will be virtualized, and virtual memory may be supported. Under these implementations, page locking services usually must be used to lock interrupt and exception handling code.

2.1.3 Complete Virtualization

These environments provide a completely simulated DOS environment. The native operating system is something other than MS-DOS. Under these implementations of DPMI, all devices will be virtualized to some extent. Normally, page locking services will be ignored by these implementations since all physical device interrupt and I/O handling will be performed by the operating system. Programs will always run at a non-privileged ring.

2.2 Descriptor Management

Protected mode code segments can not be modified. This requires programs to allocate an alias data descriptor if they need to store data in a code segment.

Segment arithmetic that works in real mode does not work in protected mode.

Some calls will return a range of descriptors. For example, if a 16-bit mode program allocates a block of memory larger than 64K, the call will allocate several, contiguous descriptors. Each descriptor will have a 64K limit except for the final descriptor which will have a limit that contains the remainder of the block. The call will return the first selector in the array. To get to the next selector, your program must add the value returned by Int 31h call 0003h (see page 28).

2.3 Interrupt Flag Management

The *popf* and *iret* instructions may not modify the state of the interrupt flag since most DPMI implementations will run programs with IOPL < DPL. Programs must execute *cli* or *sti* to modify the interrupt flag state.

This means that the following code sequence will leave interrupts disabled:

```
;
; (Assume interrupts are enabled at this point)
;
        pushf
        cli
        .
        .
        popf                ; Interrupts are still OFF!
```

Note that since some implementations of DPMI will maintain a virtual interrupt state for protected mode DOS programs, the current value of the interrupt flag may not reflect the current virtual interrupt state. Protected mode programs should use the virtual interrupt state services to determine the current interrupt flag state (see page 84).

Since *cli* and *sti* are privileged instructions, they will cause a protection violation and the DPMI provider will simulate the instruction. Because of the overhead involved in processing the exception, *cli* and *sti* should be used as little as possible. In general, you should expect either of these instructions to require at least 300 clocks.

2.4 Interrupts

Protected mode programs can hook both hardware and software interrupts using the DPMI get and set protected mode interrupt vector functions (see page 50). All interrupts from hardware devices such as the timer or keyboard controller will always be reflected to the protected mode interrupt handler first. If the protected mode handler jumps to or calls the previous interrupt handler then the interrupt will be reflected to real mode.

As in real mode, interrupt procedures can either service the interrupt and iret or they can chain to the next handler in the interrupt chain by executing *pushf/call* or by jumping to the next handler. The final handler for all protected mode interrupts will reflect the interrupt to real mode.

When an interrupt is reflected to real mode, the EAX, EBX, ECX, EDX, ESI, EDI, EBP registers, and flags will all be passed from protected to real mode unaltered. The segment registers will contain undefined values unless an API translator (such as a DOS or BIOS translator) explicitly sets a real mode segment register. DPMI will automatically provide a real mode stack for interrupts that are reflected to real mode.

2.4.1 Hardware Interrupts

The interrupt controllers are mapped to the system's default interrupts. On an IBM AT-compatible system, for example, the master interrupt controller is programmed with a base interrupt of 8 and the slave controller has a base of 70h. The virtualized interrupt controllers can be reprogrammed; the base setting may be examined in protected mode with Int 31h function 0400h.

Hardware interrupt procedures and all of their data must reside in locked memory. All memory that is touched by hardware interrupt hooks must be locked. The handler will always be called on a locked stack. See page 10 for more details.

As in real mode, hardware interrupt handlers are called with interrupts disabled. Since *iret* will not restore the interrupt flag, hardware interrupt hooks must execute an *sti* before executing *iret* or else interrupts will remain disabled.

Protected mode hardware interrupt handlers will always be called even for interrupts that occur in real mode. The last hook on the protected mode interrupt chain will reflect the interrupt to real mode.

Protected mode hardware interrupt handlers that need to call software running in real mode must either be sure that the real mode software that they are calling will not modify segment registers or they must use the state save service (see page 63) to save and restore the real mode segment registers. However, any interrupt handler that executes completely in protected mode, or uses translation services 0300h, 0301h, or 0302h does not need to save the real mode register state. Therefore, this is not an issue for most interrupt handlers.

For compatibility with older systems, computers with two interrupt controllers have the BIOS redirect one of the interrupts from the slave controller into the range of the master controller. For example, devices jumpered for IRQ 2 on IBM AT-compatible computers actually interrupt on IRQ 9 (interrupt 71h). In real mode, the BIOS on these systems will convert interrupt 71h to Int 0Ah and EOI the slave controller. A protected mode program that needs access to the redirected interrupt may use variations on either of these techniques:

1. Hook the target interrupt in real mode. This takes advantage of the built in redirection. This is robust on systems where other software has reprogrammed the interrupt controllers, or where the slave interrupt controller may be absent.

2. Hook the actual interrupt in both real and protected mode. In this case, the program must EOI both the slave and master interrupt controllers since the BIOS will not get control. This is more efficient in that there will not be any unnecessary switches to real mode.

2.4.2 Software Interrupts

Most software interrupts executed in real mode will not be reflected to the protected mode interrupt hooks. However, some software interrupts are also reflected to protected mode programs when they are called in real mode. These are:

INT	DESCRIPTION
1Ch	BIOS timer tick interrupt
23h	DOS Ctrl+C interrupt
24h	DOS critical error interrupt

Programs should not terminate during interrupts that were reflected from real mode. Terminating the program at this point may prevent the DPMI host from cleaning up properly.

Of all software interrupts, only Ints 00h-07h will be called with virtual interrupts disabled. For these interrupts, the handler should return with interrupts enabled. All other interrupts will not modify the interrupt flag state.

Since most software interrupts that are executed in real mode are not reflected to protected mode interrupt hooks, programs would be required to install a real mode interrupt hook to monitor these interrupts.

2.5 Virtual Memory and Page Locking

Many implementations of DPMI support virtual memory. In these environments, it will be necessary to lock any memory that can be touched while executing inside of DOS. This is necessary because it may not be possible for the operating system to demand load a page if DOS is busy.

Some DPMI implementations will not call DOS to read or write virtual memory to disk and under these implementations the page locking services may be ignored. Since the entire DPMI session is virtualized, a page fault can be handled at any point while executing the program. However, under all implementations, DPMI applications should lock interrupt code and data. The lock calls will always return success under implementations that ignore these calls.

3. MODE AND STACK SWITCHING

This section contains an overview of how DPMI hosts switch between protected and real mode and handle stack switching. It is important to understand the host maintains the state of the client to prevent overwriting stack data or modifying segment registers.

3.1 Stacks and Stack Switching

Every DPMI task runs on four different stacks: An application ring protected mode stack, a locked protected mode stack, a real mode stack, and a DPMI host ring 0 stack.

The protected mode stack is the one the DPMI client was running on when it switched into protected mode by calling the protected mode entry point (although the client can switch to another protected mode stack if desired). The locked protected mode stack is provided by the DPMI server and is used for simulating hardware interrupts and processing real mode call-backs. The DPMI host provides the real mode stack, which is usually located in the data area provided by the client. The ring 0 stack is only accessible by the DPMI host. However, this stack may contain state information about the currently running program.

3.1.1 Protected Mode Stack

This is the stack that the client uses for normal execution in protected mode. The protected mode stack of a DPMI client can be unlocked if desired. Software interrupts executed in protected mode will be reflected on this stack.

3.1.2 Locked Protected Mode Stack

During hardware interrupts, Int 1Ch, Int 23h, Int 24h, exceptions, and real mode call-back handling in protected mode, the DPMI will host automatically switch to a locked protected mode stack. When the interrupt or call returns, the host will return to the original protected mode stack. Note that there is only one, 4K, locked stack provided by the host. The stack will be switched onto the first time an interrupt or call is reflected to protected mode, and will be switched away from when the client returns. Subsequent nested interrupts or calls will not cause a stack switch. Software interrupts do not automatically switch stacks.

3.1.3 Real Mode Stack

The DPMI host will provide the client with a real mode stack that is at least 200h bytes in size and will always be locked. Interrupts that are reflected into real mode, as well as calls made using the translation services, will be reflected on this stack. DPMI hosts will not automatically switch stacks for hardware interrupt processing in real mode since DOS performs this function automatically.

3.1.4 DPMI Host Ring 0 Stack

DPMI hosts will normally have a stack associated with each DPMI task. The DPMI client will not be able to access this stack in any way -- it is used by the host for execution at ring 0 to handle interrupts and exceptions. This stack will sometimes be used to store state information while switching modes. For example, the original SS:ESP of the protected mode program could be saved on the ring 0 stack while the DPMI host switches onto the locked protected mode stack.

3.2 Default Interrupt Reflection

DPMI hosts provide interrupt vectors for all 100h (256 decimal) interrupts for protected mode clients. When the DPMI client initializes, all interrupt vectors will point to code that will automatically reflect the interrupt to real mode (except for Int 31h and Int 21h, AH=4Ch). When a default interrupt reflection handler is executed it will switch to real mode, preserving the EAX, EBX, ECX, EDX, ESI, EDI, and EBP registers and flags, and reflect the interrupt in real mode. When the real mode interrupt returns, the default interrupt reflection code will switch back to protected mode and return with the modified values of EAX, EBX, ECX, EDX, ESI, EDI, EBP, and flags. Segment registers and the stack pointer will not be passed between modes. Therefore, any API that passes pointers or information in segment registers will need to be translated by a DOS extender.

3.3 Mode Switching

There are three different ways a client can force a mode switch between protected and real mode:

- Execute the default interrupt reflection handler
- Use the translation services to call real mode code
- Use a real mode call-back to switch from real to protected mode
- Use the raw mode switch functions

All mode switches except for the raw mode switches will save some information on the DPMI host's ring 0 stack. This means that programs should not terminate while in nested mode switches unless they are using the raw mode switching services. However, even programs that use raw mode switches should not attempt to terminate from a hardware interrupt or exception handler since the DPMI host performs automatic mode and stack switching to provide these services.

3.4 State Saving

Because DPMI hosts switch stacks automatically across mode switches, it is sometimes necessary to use the state save/restore functions while using the raw mode switch services. The host will maintain information on the "other" mode's current state. This information will include the CS:(E)IP, SS:(E)SP, and segment register values. Since the DPMI client has no way to directly access these values, it will need to call the state saving functions when performing nested mode switches.

For example, during hardware interrupts, the DPMI host will preserve the real mode's segment registers, CS:EIP, and SS:ESP on the ring 0 stack. However, they are not pushed on any stack in the VM -- They are only visible at ring 0. When the raw mode switch functions are called they will overwrite the information saved by the host. At this point, the program would return to the wrong address when the interrupt returned. For more information on state saving, refer to the documentation on page 63.

4. ERROR HANDLING

Most Int 31h calls can fail. The DPMI 0.9 specification does not specify error return codes for most calls. When a call fails it will set the carry flag and return with the value in AX unmodified unless otherwise specified. However, future DPMI implementations will return error codes in the AX register. All specific error codes will have the high bit (bit 15) set. If a function returns with carry set and the high bit of AX clear, it should be treated as a general failure. Specific error codes will allow programs running under future DPMI implementations to take appropriate corrective action in some cases.

5. LOADING DPMI CLIENTS AND EXTENDED APPLICATIONS

All DPMI applications begin execution in real mode. An application must run first as a standard real mode DOS program but it can switch to protected execution by making a few simple calls.

DPMI does not define an executable file format for protected mode programs. Instead, programs must provide their own mechanism for loading and fixing up protected mode code.

5.1 Obtaining the Real to Protected Mode Switch Entry Point

This function can be called in real mode to detect the presence of DPMI services and to obtain an address that can be used to begin execution in protected mode.

To Call

AX = 1687h
Execute an Int 2Fh (not an Int 31h)

Returns

If function was successful:
AX = 0
BX = Flags
    Bit 0 = 1 if 32-bit programs are supported
CL = Processor type
    02h = 80286
    03h = 80386
    04h = 80486
DH = DPMI major version number
DL = DPMI minor version number
SI = Number of paragraphs required for DPMI host private data (may be 0)
ES:DI = Address of procedure to call to enter protected mode

If function was not successful:
AX != 0

Programmer's Notes

- This function does not perform the actual transition into protected mode. You need to call the address returned in ES:DI, after allocating the private data area for the DPMI host, to perform the actual real to protected mode switch.

5.2 Calling the Real to Protected Mode Switch Entry Point

After using Int 2Fh function 1687h, to obtain the protected mode entry point, the DPMI client must call the entry point address as described in this section.

To Call

AX = Flags
    Bit 0 = 1 if program is a 32-bit application
ES = Real mode segment of DPMI host data area. This must be the size of the data area returned in SI from the previous function. ES will be ignored if the required data size is zero.
Call the address returned in ES:DI by the previous function

Returns

If function was successful:
Carry flag is clear.
Program is now executing in protected mode.
CS = 16-bit selector with base of real mode CS and a 64K limit
SS = Selector with base of real mode SS and a 64K limit
DS = Selector with base of real mode DS and a 64K limit
ES = Selector to program's PSP with a 100h byte limit
FS and GS = 0 (if running on an 80386 or 80486)
If the program is a 32-bit application the high word of ESP will be 0
All other registers are preserved

If function was not successful:
Carry flag is set.
Program is executing in real mode

Programmer's Notes

- Once in protected mode, all Int 31h calls that are supported by DPMI can be called.
- To terminate the program, execute an Int 21h with AH=4Ch and AL=Error code. This is the standard DOS exit function. Do not use any other DOS termination call -- Only AH=4Ch is supported under DPMI.
- Under different implementations of DPMI the privilege ring of a program will change. Programs should make no assumptions about the ring at which they will run. When creating descriptors, programs should set the DPL of the descriptor to the same ring as their initial code segment. Use the *lar* instruction to determine the protection ring of your program's code segment. All descriptors created by your program should be set to the same protection level.
- Programs that specify that they are 32-bit applications will initially run with a 16-bit code segment. Stack and data selectors for 32-bit programs will be 32-bit (the Big bit will be set). However, all Int 31h calls will require 48-bit pointers even though the program is running in a 16-bit code segment.

- Unless you have explicitly enabled the A20 address line through the XMS interface, do not assume that memory from 1Mb to 1Mb+64K-16 (the High Memory Area) is addressable once your program is running in protected mode. If you want to be able to access the HMA then you must enable the A20 through XMS before entering protected mode. XMS calls are not supported in protected mode. Note that this restriction is only important for software that wishes to access the HMA. Under all implementations of DPMI the physical A20 address line will always be enabled while executing protected mode code. However, some 80386 specific DPMI implementations simulate 1Mb address wrap for compatibility reasons. Under these DPMI implementations, the HMA will not be accessible unless the A20 is enabled through the XMS interface.
- The environment pointer in the current program's PSP will automatically be converted to a descriptor. If you want to free the program's environment memory, you must do so before entering protected mode. In this case, the environment pointer descriptor will point to garbage and should not be used. The DPMI client may change the environment pointer in the PSP after entering protected mode but it must restore it to the selector created by the DPMI host before terminating.
- The caller is allowed to modify or free the DS, SS, and CS descriptors allocated by this call. You may not modify the PSP descriptor or environment pointer descriptor in the PSP. See page 26 for information on freeing descriptors.
- Note that if DS=SS on entry to this call then only one descriptor will be allocated for both DS and SS. In this case, for example, if you changed the base of the DS descriptor you would also change the base of the stack segment.
- For some hosts it may be a good idea for protected mode programs to use some or all of the real mode memory allocated to the real mode program by DOS for protected mode code or data. Protected mode programs that use memory in the first 1Mb should mark the memory as pageable using Int 31h 0602h. See page 76 for details.

Example Code

```
;
; Get the entry point address and save it
;
        mov     ax, 1687h
        int     2Fh
        test    ax, ax
        jnz     Cant_Enter_PMode
        mov     [PMode_Entry_Seg], es
        mov     [PMode_Entry_Off], di

;
; Allocate memory for use by DOS extender if necessary
; NOTE:  This code assumes that the program has already
;        shrunk its memory block so that the DOS
;        memory allocation call will work
;
        test    si, si
        jz      Enter_PMode_Now
        mov     bx, si
        mov     ah, 48h
        int     21h
        jc      Cant_Enter_PMode
        mov     es, ax

;
; Enter protected mode as a 16-bit program
;
Enter_PMode_Now:
        xor     ax, ax
        call    DWORD PTR [PMode_Entry_Off]
        jc      Cant_Enter_PMode

;
; The program is running in protected mode now!
; Protected mode initialization code would go here.
; Mark program's real mode memory as pageable, etc.
;
        .
        .
        .

;
; Quit the program and return to real mode DOS
;
        mov     ax, 4C00h
        int     21h
```

6. TERMINATING A PROTECTED MODE PROGRAM

To terminate a protected mode program execute an Int 21h with AH=4Ch in protected mode. You can return an error code in the AL register. This is the standard DOS terminate API but it must be executed in protected mode to allow the DPMI host to clean up any data structures associated with the protected mode program.

Programs should not be terminated from a hardware interrupt, exception handler, or real mode call-back. Programs should only be terminated from their main thread of execution to allow the DPMI host to clean up properly. However, DOS extenders that use the raw mode switch services for all mode transitions can execute the terminate call after switching from real to protected mode.

7. MODE DETECTION

It is possible to write a program or library that can run in either real or protected mode. This function is supplied so that bimodal code can detect at run time whether it is running under protected mode. Code that only runs in protected mode does not need to perform this test.

To Call

AX = 1686h
Execute an Int 2Fh (not an Int 31h)

Returns

If executing in protected mode under DPMI:
AX = 0

If executing in real mode or not under DPMI then:
AX != 0

Programmer's Notes

- This call will return AX = 0 when the caller is running in protected mode. It will return AX non-zero even when running under environments that support DPMI if the caller is in real (virtual 8086) mode. See page 17 for information on entering protected mode.

8. LDT DESCRIPTOR MANAGEMENT SERVICES

The LDT descriptor management services provide interfaces for allocating, freeing, creating, locking and unlocking protected mode descriptors in the current task's Local Descriptor Table (LDT).

8.1 Allocate LDT Descriptors

This function is used to allocate one or more descriptors from the task's Local Descriptor Table (LDT). The descriptor(s) allocated must be initialized by the application.

To Call

AX = 0000h
CX = Number of descriptors to allocate

Returns

If function was successful:
Carry flag is clear.
AX = Base selector

If function was not successful:
Carry flag is set.

Programmer's Notes

- If more than one descriptor was requested, AX will contain the first of a contiguous array of descriptors. You should add the value returned by function 0003h (see page 28) to get to the next selector in the array.
- The descriptor will be set to present data type, with a base and limit of zero.
- It is up to the caller to fill in the descriptors.
- The privilege level of descriptors will match the application's code segment privilege level. When modifying descriptors, always set the DPL to the same privilege ring as your program's code segment. Use the *lar* instruction to determine the privilege of a descriptor.

8.2 Free LDT Descriptor

This function is used to free descriptors that were allocated through the Allocate LDT Descriptors function.

To Call

AX = 0001h
BX = Selector to free

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- Arrays of descriptors are freed by calling this function for each of the individual descriptors.
- It is valid to free the descriptors allocated for the program's initial CS, DS, and SS. Other descriptors that were not allocated by function 0000h should never be freed by this function unless otherwise specified.

8.3 Segment to Descriptor

This function is used to convert real mode segments into descriptors that are addressable by protected mode programs.

To Call

AX = 0002h
BX = Real mode segment address

Returns

If function was successful:
Carry flag is clear.
AX = Selector mapped to real mode segment

If function was not successful:
Carry flag is set.

Programmer's Notes

- Multiple calls to this function with the same segment will return the same selector.
- Descriptors created by this function should never be modified or freed. For this reason, you should use this function sparingly. If your program needs to examine various real mode addresses using the same selector you should allocate a descriptor and change the base using the Set Segment Base Address function instead of using this function.
- The descriptor's limit will be set to 64K.
- The intent of this function is to allow programs easy access to commonly used real mode segments such as 40h and A000h. Do not use this service to obtain descriptors to private data areas.

8.4 Get Next Selector Increment Value

Some functions such as allocate LDT descriptors and allocate DOS memory can return more than one descriptor. You must call this function to determine the value that must be added to a selector to access the next descriptor in the array.

To Call

AX = 0003h

Returns

Carry flag clear (this function always succeeds)
AX = Value to add to get to next selector

Programmer's Notes

- Do not make any assumptions about the value this function will return.
- The increment value returned will be a power of two.

8.5 Reserved Subfunctions

Functions 0004h and 0005h are reserved and should not be called.

8.6 Get Segment Base Address

This function returns the 32-bit linear base address of the specified segment.

To Call

AX = 0006h
BX = Selector

Returns

If function was successful:
Carry flag is clear.
CX:DX = 32-bit linear base address of segment

If function was not successful:
Carry flag is set.

Programmer's Notes

- This function will fail if the selector specified in BX is invalid

8.7 Set Segment Base Address

This function changes the 32-bit linear base address of the specified selector.

To Call

AX = 0007h
BX = Selector
CX:DX = 32-bit linear base address for segment

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- This function will fail if the selector specified in BX is invalid.
- Your program should only modify descriptors that were allocated through the Allocate LDT Descriptors function.
- The high 8 bits of the base address (contained in CH) will be ignored by 16-bit implementations of DPMI. This is true even when running on 80386 machines.

8.8 Set Segment Limit

This function sets the limit for the specified segment.

To Call

AX = 0008h
BX = Selector
CX:DX = 32-bit segment limit

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- This function will fail if the selector specified in BX is invalid or the specified limit could not be set. 16-bit DPMI implementations can not set segment limits greater than 0FFFFh (64K) so CX must be zero when calling this function under these implementations of DPMI.
- Segment limits greater than 1 meg must be page aligned. That is, limits greater than one megabyte must have the low 12 bits set.
- Your program should only modify descriptors that were allocated through the Allocate LDT Descriptors function.
- To get the limit of a segment you should use the instruction *lsl* (load segment limit) which is supported on 80286 and 80386 machines. Note that on 80386 machines you will need to use the 32-bit form of *lsl* if the segment has a limit greater than 64K.

8.9 Set Descriptor Access Rights

This function allows a protected mode program to modify the access rights and type fields of a descriptor.

To Call

AX = 0009h
BX = Selector
CL = Access rights/type byte
CH = 80386 extended access rights/type byte (32-bit DPMI implementations only)

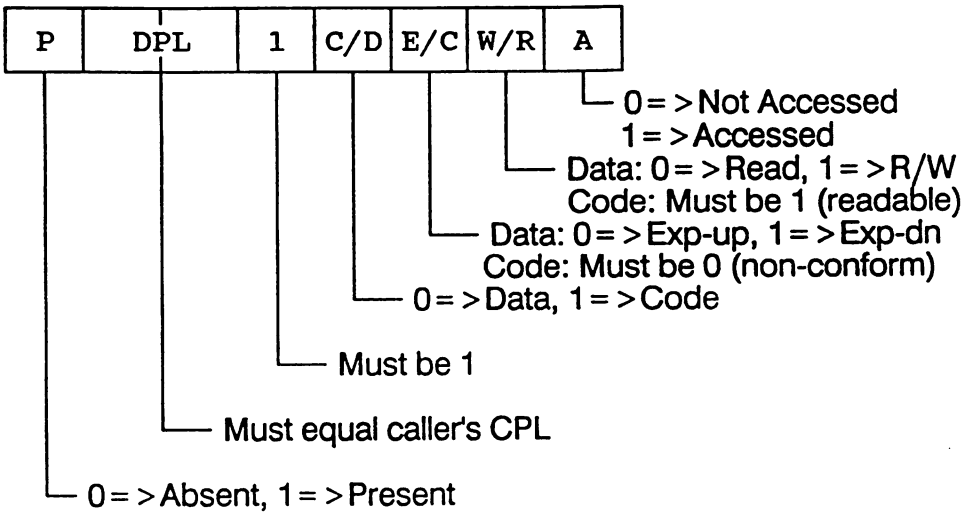
Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

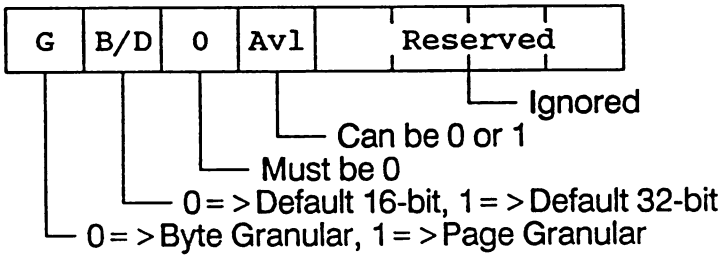
Programmer's Notes

- This function will fail if the selector specified in BX is invalid.
- Your program should only modify descriptors that were allocated through the Allocate LDT Descriptors function.
- To examine the access rights of a descriptor you should use the instruction *lar* (load access rights) which is supported on 80286 and 80386 machines.
- The access rights/type byte passed in CL has the following format:



A parameter which does not meet the above requirements is invalid, and causes the function to return with the carry flag set.

- 16-bit DPMI implementations will ignore the extended access rights/type byte passed in CH even if it is running on an 80386 system. 32-bit DPMI implementations interpret the CH parameter as follows:



A parameter which does not meet the above requirements is invalid, and causes the function to return with the carry flag set.

8.10 Create Code Segment Alias Descriptor

This function will create a data descriptor that has the same base and limit as the specified code segment descriptor.

To Call

AX = 000Ah
BX = Code segment selector

Returns

If function was successful:
Carry flag is clear.
AX = New data selector

If function was not successful:
Carry flag is set.

Programmer's Notes

- This function will fail if the selector specified in BX is not a code segment or is invalid.
- Use the Free LDT Descriptor function to deallocate the alias descriptor.
- The code segment alias descriptor will not track changes to the code descriptor. In other words, if an alias descriptor is created, and then the base or limit of the code segment is changed, the alias descriptor's base or limit would not change.

8.11 Get Descriptor

This function copies the descriptor table entry for a specified descriptor into an eight byte buffer.

To Call

AX = 000Bh
BX = Selector
ES:(E)DI = Pointer to an 8 byte buffer to receive copy of descriptor

Returns

If function was successful:
Carry flag is clear.
ES:(E)DI = Pointer to buffer that contains descriptor

If function was not successful:
Carry flag is set.

Programmer's Notes

- This function will fail if the selector specified in BX is invalid or unallocated.
- 32-bit programs must use ES:EDI to point to the buffer. 16-bit programs should use ES:DI.

8.12 Set Descriptor

This function copies an eight byte buffer into the LDT entry for a specified descriptor.

To Call

AX = 000Ch
BX = Selector
ES:(E)DI = Pointer to an 8 byte buffer that contains descriptor

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- This function will fail if the selector specified in BX is invalid.
- Your program should only modify descriptors that were allocated through the Allocate LDT Descriptors function.
- 32-bit programs must use ES:EDI to point to the buffer. 16-bit programs should use ES:DI.
- The type byte (byte 5) follows the same format and restrictions as the access rights/type parameter (in CL) to Set Descriptor Access Rights. The extended type byte (byte 6) follows the same format and restrictions as the extended access rights/type parameter (in CH) to Set Descriptor Access Rights, except the limit field may have any value, except the low order 4 bits (marked "reserved") are used to set the upper 4 bits of the descriptor's limit.

8.13 Allocate Specific LDT Descriptor

This function attempts to allocate a specific LDT descriptor.**To Call**

AX = 000Dh
BX = Selector

Returns

If function was successful:
Carry flag is clear.
Descriptor has been allocated

If function was not successful:
Carry flag is set.

Programmer's Notes

- This function will fail if the selector specified in BX is in use or is not an LDT selector.
- Use function 0001h to free the descriptor.
- The first 10h (16 decimal) descriptors must be reserved for this function and may not be used by the host.
- If another application has already loaded then some of these descriptors may be in use.

9. DOS MEMORY MANAGEMENT SERVICES

Some applications require the ability to allocate memory in the real mode addressable 1 megabyte region. These services allow protected mode applications to allocate and free memory that is directly addressable by real mode software such as networks and DOS device drivers. Often, this memory is used in conjunction with the API translation services to call real mode software that is not directly supported by DPMI.

9.1 Allocate DOS Memory Block

This function will allocate a block of memory from the DOS free memory pool. It returns both the real mode segment and one or more descriptors that can be used by protected mode applications to access the block.

To Call

AX = 0100h
BX = Number of paragraphs (16 byte blocks) desired

Returns

If function was successful:
Carry flag is clear.
AX = Initial real mode segment of allocated block
DX = Selector for allocated block

If function was not successful:
Carry flag is set.
AX = DOS error code:
- 07h memory control blocks damaged
- 08h insufficient memory available to allocate as requested

BX = Size of largest available block in paragraphs

Programmer's Notes

- o If the size of the block requested is greater than 64K bytes (BX > 1000h) then contiguous descriptors will be allocated. To access the next descriptor for the memory block add the value return by function 0003h (see page 28) to the base selector. If more than one descriptor is allocated under 32-bit DPMI implementations, the limit of the first descriptor will be set to the size of the entire block. All subsequent descriptors will have a limit of 64K except for the final descriptor which will have a limit of Block size MOD 64K. 16-bit DPMI implementations will always set the limit of the first descriptor to 64K even when running on an 80386.
- o Your program should never modify or deallocate any descriptors allocated by this function. The Free DOS Memory Block function will deallocate the descriptors automatically

9.2 Free DOS Memory Block

This function frees memory that was allocated through the Allocate DOS Memory Block function.

To Call

AX = 0101h
DX = Selector of block to free

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.
AX = DOS error code:

07h	memory control blocks damaged
09h	incorrect memory segment specified

Programmer's Notes

- All descriptors allocated for the memory block are automatically freed and therefore should not be accessed once the block is freed by this function.

9.3 Resize DOS Memory Block

This function is used to grow or shrink a memory block that was allocated through the Allocate DOS Memory Block function.

To Call

AX = 0102h
BX = New block size in paragraphs
DX = Selector of block to modify

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.
AX = DOS error code:
- 07h memory control blocks damaged
- 08h insufficient memory available to allocate as requested
- 09h incorrect memory segment specified

BX = Maximum block size possible in paragraphs

Programmer's Notes

- Growing a memory block is often likely to fail since other DOS block allocations will prevent increasing the size of the block. Also, if the size of a block grows past a 64K boundary then the allocation will fail if the next descriptor in the LDT is not free. Therefore, this function is usually only used to shrink a block.
- Shrinking a block may cause some descriptors that were previously allocated to the block to be freed. For example shrinking a block from 140K to 120K would cause the third allocated descriptor to be freed since it is no longer valid. The initial selector will remain unchanged, however, the limits of the remaining two descriptors will change: the first to 120K and the second to 56k.

10. INTERRUPT SERVICES

These services allow protected mode applications to intercept real and protected mode interrupts and hook processor exceptions.

10.1 Get Real Mode Interrupt Vector

This function returns the value of the current task's real mode interrupt vector for the specified interrupt.

To Call

AX = 0200h
BL = Interrupt number

Returns

Carry flag is clear.
CX:DX = Segment:Offset of real mode interrupt handler

Programmer's Notes

- The address returned in CX is a segment, not a selector. Therefore you should not attempt to place the value returned in CX into a segment register in protected mode or a general protection fault may occur.
- Note all 100h (256 decimal) interrupt vectors must be supported by the DPMI host.

10.2 Set Real Mode Interrupt Vector

This function sets the value of the current task's real mode interrupt vector for the specified interrupt.

To Call

AX = 0201h
BL = Interrupt number
CX:DX = Segment:Offset of real mode interrupt handler

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- The address passed in CX must be a real mode segment, not a selector.
- If the interrupt being hooked is a hardware interrupt then you must lock the segment that the interrupt handler runs in as well as any memory the handler may touch at interrupt time.
- The address contained in CX:DX must be a real mode segment:offset, not a selector:offset. This means that the code for the interrupt handler must either reside in DOS addressable memory or you must use a real mode call-back address. Refer to the section on DOS memory management services on page 39 for information on allocating memory below 1 megabyte. Information on real mode call back addresses can be found on page 58.

10.3 Get Processor Exception Handler Vector

This function returns the CS:(E)IP of the current protected mode exception handler for the specified exception number.

To Call

AX = 0202h
BL = Exception/fault number (00h-1Fh)

Returns

If function was successful:
Carry flag is clear.
CX:(E)DX = Selector:Offset of exception handler

If function was not successful:
Carry flag is set.
The value passed in BL was invalid.

Programmer's Notes

- The value returned in CX is a valid protected mode selector, not a real mode segment.
- 32-bit mode programs will be returned a 32-bit offset in the EDX register.

10.4 Set Processor Exception Handler Vector

This function allows protected mode applications to intercept processor exceptions that are not handled by the DPMI environment. Programs may wish to handle exceptions such as not present segment faults which would otherwise generate a fatal error.

Every exception is first examined by the protected mode operating system. If it can not handle the exception it then reflects it through the protected mode exception handler chain. The final handler in the chain may either reflect the exception as an interrupt (as would happen in real mode) or it may terminate the current program.

To Call

AX = 0203h
BL = Exception/fault number (00h-1Fh)
CX:(E)DX = Selector:Offset of exception handler

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.
The value passed in BL was invalid.

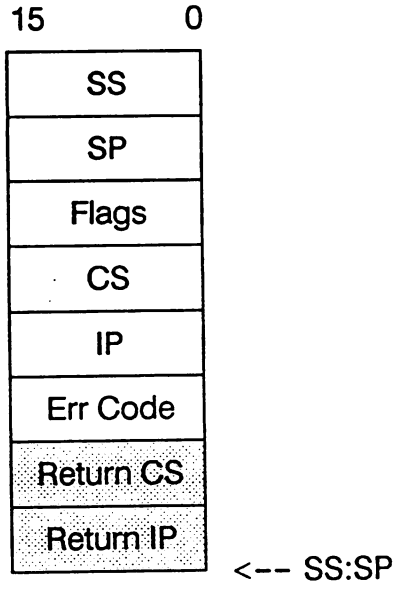
Programmer's Notes

- The value passed in CX must be a valid protected mode code selector, not a real mode segment.
- 32-bit mode programs must supply a 32-bit offset in the EDX register. If your handler chains to the next exception handler it must do so using a 32-bit interrupt stack frame.
- The handler should return using a far return instruction. The original SS:(E)SP, CS:(E)IP and flags on the stack, including the interrupt flag, will be restored.
- All fault stack frames have an error code. However, the error code is only valid for exceptions 08h, 0Ah, 0Bh, 0Ch, 0Dh, and 0Eh.
- The handler must preserve and restore all registers.
- The exception handler will be called on a locked stack with interrupts disabled. The original SS, (E)SP, CS, and (E)IP will be pushed on the exception handler stack frame.
- The handler must either return from the call by executing a far return or jump to the next handler in the chain (which will execute a far return or chain to the next handler).

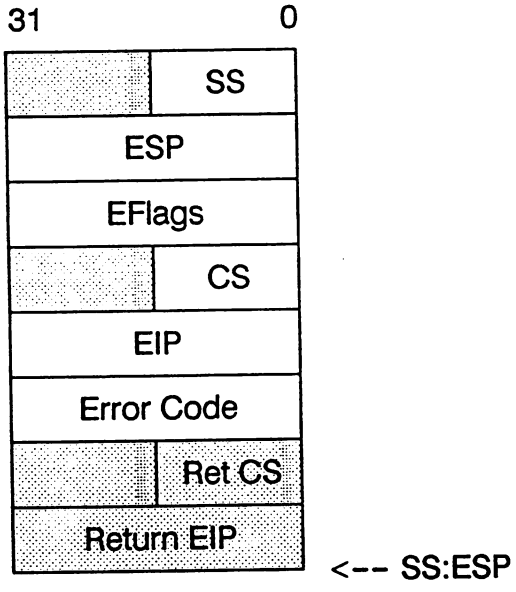
- The procedure can modify any of the values on the stack pertaining to the exception before returning. This can be used, for example, to jump to a procedure by modifying the CS:IP on the stack. Note that the procedure must not modify the far return address on the stack -- it must return to the original caller. The caller will then restore the flags, CS:(E)IP and SS:(E)SP from the stack frame.
- If the DPMI client does not handle an exception, or jumps to the default exception handler, the host will reflect the exception as an interrupt for exceptions 0, 1, 2, 3, 4, 5, and 7. Exceptions 6, and 8-1Fh will be treated as fatal errors and the client will be terminated.
- Exception handlers will only be called for exceptions that occur in protected mode.

Call-Back Stack Frames

Stack frame for 16-bit programs:



Stack frame for 32-bit programs:



Shaded fields should not be modified. Other fields can be modified before returning from the exception handler.

10.5 Get Protected Mode Interrupt Vector

This function returns the CS:(E)IP of the current protected mode interrupt handler for the specified interrupt number.

To Call

AX = 0204h
BL = Interrupt number

Returns

Carry flag is clear.
CX:(E)DX = Selector:Offset of exception handler

Programmer's Notes

- The value returned in CX is a valid protected mode selector, not a real mode segment.
- 32-bit mode programs will be returned a 32-bit offset in the EDX register.
- All 100h (256 decimal) interrupt vectors must be supported by the DPMI host.

10.6 Set Protected Mode Interrupt Vector

This function sets the address of the specified protected mode interrupt vector.

To Call

AX = 0205h
BL = Interrupt number
CX:(E)DX = Selector:Offset of exception handler

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- The value passed in CX must be a valid protected mode code selector, not a real mode segment.
- 32-bit mode programs must supply a 32-bit offset in the EDX register. If your handler chains to the next exception handler it must do so using a 32-bit interrupt stack frame.
- Note all 100h (256 decimal) interrupt vectors must be supported by the DPMI host.

11. TRANSLATION SERVICES

These services are provided so that protected mode programs can call real mode software that DPMI does not support directly. The protected mode program sets up a data structure that contains the values for every register. The data structure is defined as:

Offset	Register
00h	EDI
04h	ESI
08h	EBP
0Ch	Reserved by system
10h	EBX
14h	EDX
18h	ECX
1Ch	EAX
20h	Flags
22h	ES
24h	DS
26h	FS
28h	GS
2Ah	IP
2Ch	CS
2Eh	SP
30h	SS

You will notice that all of the fields are dwords so that 32 bit registers can be passed to real mode. Most real mode software will ignore the high word of the extended registers. However, you can write a real mode procedure that uses 32-bit registers if you desire. Note that 16-bit DPMI implementations may not pass the high word of 32-bit registers or the FS and GS segment registers to real mode even when running on an 80386 machine.

Any interrupt handler or procedure called must return with the stack in the same state as when it was called. This means that the real mode code may switch stacks while it is running but it must return on the same stack that it was called on and it must pop off the entire far return/iret structure.

After the call or interrupt is complete, all real mode registers and flags except SS, SP, CS, and IP will be copied back to the real mode call structure so that the caller can examine the real mode return values.

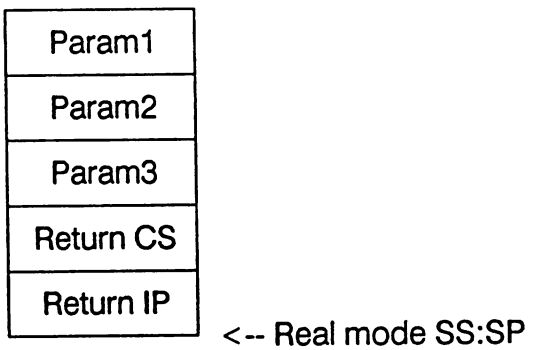
Remember that the values in the segment registers should be real mode segments, not protected mode selectors.

The translation services will provide a real mode stack if the SS:SP fields are zero. However, the stack provided is relatively small. If the real mode procedure/interrupt routine uses more than 30 words of stack space then you should provide your own real mode stack.

It is possible to pass parameters to real mode software on the stack. The following code will call a real mode procedure with 3 word parameters:

```
Protected_Mode_Code:
        push    Param1
        push    Param2
        push    Param3
        (Set ES:DI to point to call structure)
        mov     cx, 3           ; Copy 3 words
        mov     ax, 0301h       ; Call real mode proc
        int     31h             ; Call the procedure
        add     sp, 6           ; Clean up stack
```

The real mode procedure would be called with the following data on the real mode stack:



If your program needs to perform a series of calls to a real mode API it is sometimes more convenient to use the translation services to call a real mode procedure in your own program. That procedure can then issue the API calls in real mode and then return to protected mode. This also avoids the overhead of a mode switch for each API call.

There is also a mechanism for protected mode software to gain control from real mode via a real mode call-back address. Real mode call-backs can be used to hook real mode interrupts or to be called in protected mode by a real mode driver. For example, many mouse drivers will call a specified address whenever the mouse is moved. This service allows the call-back to be handled by software running in protected mode.

11.1 Simulate Real Mode Interrupt

This function simulates an interrupt in real mode. It will invoke the CS:IP specified by the real mode interrupt vector and the handler must return by executing an *iret*.

To Call

AX = 0300h
BL = Interrupt number
BH = Flags
    Bit 0 = 1 resets the interrupt controller and A20 line
    Other flags reserved and must be 0
CX = Number of words to copy from protected mode to real mode stack
ES:(E)DI = Selector:Offset of real mode call structure

Returns

If function was successful:
Carry flag is clear.
ES:(E)DI = Selector:Offset of modified real mode call structure

If function was not successful:
Carry flag is set.

Programmer's Notes

- The CS:IP in the real mode call structure is ignored by this service. The appropriate interrupt handler will be called based on the value passed in BL.
- If the SS:SP fields are zero then a real mode stack will be provided by the DPMI host. Otherwise, the real mode SS:SP will be set to the specified values before the interrupt handler is called.
- The flags specified in the real mode call structure will be pushed on the real mode stack iret frame. The interrupt handler will be called with the interrupt and trace flags clear.
- When the Int 31h returns, the real mode call register structure will contain the values that were returned by the real mode interrupt handler.
- It is up to the caller to remove any parameters that were pushed on the protected mode stack.
- 32-bit programs must use ES:EDI to point to the real mode call structure. 16-bit programs should use ES:DI.
- The flag to reset the interrupt controller and A20 line is ignored by DPMI implementations that run in Virtual 8086 mode. It causes DPMI implementations that return to real mode to set the interrupt controller and A20 address line hardware to its normal real mode state.

11.2 Call Real Mode Procedure With Far Return Frame

This function calls a real mode procedure. The called procedure must execute a far return when it completes.

To Call

AX = 0301h
BH = Flags
    Bit 0 = 1 resets the interrupt controller and A20 line
    Other flags reserved and must be 0
CX = Number of words to copy from protected mode to real mode stack
ES:(E)DI = Selector:Offset of real mode call structure

Returns

If function was successful:
Carry flag is clear.
ES:(E)DI = Selector:Offset of modified real mode call structure

If function was not successful:
Carry flag is set.

Programmer's Notes

- The CS:IP in the real mode call structure specifies the address of the real mode procedure to call.
- The real mode procedure must execute a far return when it has completed.
- If the SS:SP fields are zero then a real mode stack will be provided by the DPMI host. Otherwise, the real mode SS:SP will be set to the specified values before the procedure is called.
- When the Int 31h returns, the real mode call structure will contain the values that were returned by the real mode procedure.
- It is up to the caller to remove any parameters that were pushed on the protected mode stack.
- 32-bit programs must use ES:EDI to point to the real mode call structure. 16-bit programs should use ES:DI.
- The flag to reset the interrupt controller and A20 line is ignored by DPMI implementations that run in Virtual 8086 mode. It causes DPMI implementations that return to real mode to set the interrupt controller and A20 address line hardware to its normal real mode state.

11.3 Call Real Mode Procedure With Iret Frame

This function calls a real mode procedure. The called procedure must execute an *iret* when it completes.

To Call

AX = 0302h
BH = Flags
    Bit 0 = 1 resets the interrupt controller and A20 line
    Other flags reserved and must be 0
CX = Number of words to copy from protected mode to real mode stack
ES:(E)DI = Selector:Offset of real mode call structure

Returns

If function was successful:
Carry flag is clear.
ES:(E)DI = Selector:Offset of modified real mode call structure

If function was not successful:
Carry flag is set.

Programmer's Notes

- The CS:IP in the real mode call structure specifies the address of the real mode procedure to call.
- The real mode procedure must execute an *iret* when it has completed.
- If the SS:SP fields are zero then a real mode stack will be provided by the DPMI host. Otherwise, the real mode SS:SP will be set to the specified values before the procedure is called.
- When the Int 31h returns, the real mode call structure will contain the values that were returned by the real mode procedure.
- The flags specified in the real mode call structure will be pushed the real mode stack iret frame. The procedure will be called with the interrupt and trace flags clear.
- It is up to the caller to remove any parameters that were pushed on the protected mode stack.
- 32-bit programs must use ES:EDI to point to the real mode call structure. 16-bit programs should use ES:DI.
- The flag to reset the interrupt controller and A20 line is ignored by DPMI implementations that run in Virtual 8086 mode. It causes DPMI implementations that return to real mode to set the interrupt controller and A20 address line hardware to its normal real mode state.

11.4 Allocate Real Mode Call-Back Address

This service is used to obtain a unique real mode SEG:OFFSET that will transfer control from real mode to a protected mode procedure.

At times it is necessary to hook a real mode interrupt or device call-back in a protected mode driver. For example, many mouse drivers call an address whenever the mouse is moved. Software running in protected mode can use a real mode call-back to intercept the mouse driver calls.

To Call

AX = 0303h
DS:(E)SI = Selector:Offset of procedure to call
ES:(E)DI = Selector:Offset of real mode call structure

Returns

If function was successful:
Carry flag is clear.
CX:DX = Segment:Offset of real mode call address

If function was not successful:
Carry flag is set.

Call-Back Procedure Parameters

Interrupts disabled
DS:(E)SI = Selector:Offset of real mode SS:SP
ES:(E)DI = Selector:Offset of real mode call structure
SS:(E)SP = Locked protected mode API stack
All other registers undefined

Return from Call-Back Procedure

Execute an IRET to return
ES:(E)DI = Selector:Offset of real mode call structure to restore (see note)

Programmer's Notes

- Since the real mode call structure is static, you must be careful when writing code that may be reentered. The simplest method of avoiding reentrancy is to leave interrupts disabled throughout the entire call. However, if the amount of code executed by the call-back is large then you will need to copy the real mode call structure into another buffer. You can then return with ES:(E)DI pointing to the buffer you copied the data to -- it does not have to point to the original real mode call structure.
- The called procedure is responsible for modifying the real mode CS:IP before returning. If the real mode CS:IP is left unchanged then the real mode call-back will be executed immediately and your procedure will be called again. Normally you will want to pop a return address off of the real mode stack and place it in the real mode CS:IP. The example code in the next section demonstrates chaining to another interrupt handler and simulating a real mode iret.
- To return values to the real mode caller you must modify the real mode call structure.
- Remember that all segment values in the real mode call structure will contain real mode segments, not selectors. If you need to examine data pointed to by a real mode seg:offset pointer you should not use the segment to selector service to create a new selector. Instead, allocate a descriptor during initialization and change the descriptor's base to 16 times the real mode segment's value. This is important since selectors allocated though the segment to selector service can never be freed.
- DPMI hosts should provide a minimum of 16 call-back addresses per task.

Example Code

The following code is a sample of a real mode interrupt hook. It hooks the DOS Int 21h and returns an error for the delete file function (AH=41h). Other calls are passed through to DOS. This example is somewhat silly but it demonstrates the techniques used to hook a real mode interrupt. Note that since DOS calls are reflected from protected mode to real mode, the following code will intercept all DOS calls from both real mode and protected mode.

```
;*****************************************************
; This procedure gets the current Int 21h real mode
; Seg:Offset, allocates a real mode call-back address,
; and sets the real mode Int 21h vector to the call-
; back address.
;*****************************************************
Initialization_Code:
;
; Create a code segment alias to save data in
;
        mov     ax, 000Ah
        mov     bx, cs
        int     31h
        jc      ERROR
        mov     ds, ax
        ASSUMES DS,_TEXT
;
; Get current Int 21h real mode SEG:OFFSET
;
        mov     ax, 0200h
        mov     bl, 21h
        int     31h
        jc      ERROR
        mov     [Orig_Real_Seg], cx
        mov     [Orig_Real_Offset], dx
;
; Allocate a real mode call-back
;
        mov     ax, 0303h
        push    ds
        mov     bx, cs
        mov     ds, bx
        mov     si, OFFSET My_Int_21_Hook
        pop     es
        mov     di, OFFSET My_Real_Mode_Call_Struc
        int     31h
        jc      ERROR
;
; Hook real mode int 21h with the call-back address
;
        mov     ax, 0201h
        mov     bl, 21h
        int     31h
        jc      ERROR
```

```
;*******************************************************
;
; This is the actual Int 21h hook code.  It will return
; an "access denied" error for all calls made in real
; mode to delete a file.  Other calls will be passed
; through to DOS.
;
; ENTRY:
;    DS:SI -> Real mode SS:SP
;    ES:DI -> Real mode call structure
;    Interrupts disabled
;
; EXIT:
;    ES:DI -> Real mode call structure
;
;*******************************************************

My_Int_21_Hook:
        cmp     es:[di.RealMode_AH], 41h
        jne     Chain_To_DOS
;
; This is a delete file call (AH=41h).  Simulate an
; iret on the real mode stack, set the real mode
; carry flag, and set the real mode AX to 5 to indicate
; an access denied error.
;
        cld
        lodsw                       ; Get real mode ret IP
        mov     es:[di.RealMode_IP], ax
        lodsw                       ; Get real mode ret CS
        mov     es:[di.RealMode_CS], ax
        lodsw                       ; Get real mode flags
        or      ax, 1               ; Set carry flag
        mov     es:[di.RealMode_Flags], ax
        add     es:[di.RealMode_SP], 6
        mov     es:[di.RealMode_AX], 5
        jmp     My_Hook_Exit
;
; Chain to original Int 21h vector by replacing the
; real mode CS:IP with the original Seg:Offset.
;
Chain_To_DOS:
        mov     ax, cs:[Orig_Real_Seg]
        mov     es:[di.RealMode_CS], ax
        mov     ax, cs:[Orig_Real_Offset]
        mov     es:[di.RealMode_IP], ax

My_Hook_Exit:
        iret
```

11.5 Free Real Mode Call-Back Address

This function frees a real mode call-back address that was allocated through the allocate real mode call-back address service.

To Call

AX = 0304h
CX:DX = Real mode call-back address to free

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- Real mode call-backs are a limited resource. Your code should free any break point that it is no longer using.

11.6 Get State Save/Restore Addresses

When a program uses the raw mode switch services (see page 65) or issues DOS calls from a hardware interrupt handler, it will need to save the state of the current task before changing modes. This service returns the addresses of two procedures used to save the state of the current task's registers. For example, the real mode address is used to save the state of the protected mode registers. The protected mode address is used to save the state of the real mode registers. This can be used to save the state of the alternate mode's registers before they are modified by the mode switch call. The current mode's registers can be saved by simply pushing them on the stack.

Note: It is not necessary to call this service if using the translation services 0300h, 0301h or 0302h. It is provided for programs that use the raw mode switch service.

To Call

AX = 0305h

Returns

If function was successful:
Carry flag is clear
AX = Size of buffer in bytes required to save state
BX:CX = Real mode address used to save/restore state
SI:(E)DI = Protected mode address used to save/restore state

If function was not successful:
Carry flag is set

Parameters To State-Save Procedures

Execute a far call to the appropriate address (real or pmode) with:
ES:(E)DI = Pointer to state-save buffer
AL = 0 to save state
AL = 1 to restore state

Programmer's Notes

- Some implementations of DPMI will not require the state to be saved. In this case, the buffer size returned will be zero. However, it is still valid to call the addresses returned, although they will just return without performing any useful function.
- The save/restore functions will not modify any registers.
- The address returned in BX:CX must only be called in real mode. The address returned in SI:(E)DI must only be called in protected mode.
- 16-bit programs should call the address returned in SI:DI to save the real mode state. 32-bit programs should call the address returned in SI:EDI.

Example Code

The following code is a sample protected mode timer interrupt handler that saves the state of the real mode registers, issues DOS calls, and restores the state. This code assumes that the Int 31h function 0305h has been executed and that the call address and buffer size have been saved in local variables.

```
Sample_Timer_Code:
        pushf
        call    FAR PTR cs:[Next_Timer_Handler]
        sti
;
; Save protected mode registers
;
        push    ds
        push    es
        pusha
;
; Save real mode registers
;
        mov     ds, cs:[My_Local_DS]
        mov     ax, ss
        mov     es, ax
        sub     sp, [State_Save_Size]
        mov     di, sp
        xor     al, al
        call    [PM_Save_Restore_State]

;
; Raw mode switch here
;
        .
        .
        .
;
; Restore real mode registers
;
        mov     ax, ss
        mov     es, ax
        mov     di, sp
        mov     al, 1
        call    [PM_Save_Restore_State]
        add     sp, [State_Save_Size]
;
; Restore protected mode registers and return
;
        popa
        pop     es
        pop     ds

        iret
```

11.7 Get Raw Mode Switch Addresses

This function returns addresses that can be jumped to for low-level mode switching.**To Call**

AX = 0306h

Returns

If function was successful:
Carry flag is clear
BX:CX = Real -> Protected mode switch address
SI:(E)DI = Protected -> Real mode switch address

If function was not successful:
Carry flag is set

Parameters To State-Save Procedures

Execute a far jump to the appropriate address (real or pmode) with:
AX = New DS
CX = New ES
DX = New SS
(E)BX = New (E)SP
SI = New CS
(E)DI = New (E)IP

The processor will be placed in the desired mode. The DS, ES, SS, (E)SP, CS, and (E)IP will contain the values specified. The (E)BP register will be preserved across the call and so can be used as a pointer. The values in (E)AX, (E)BX, (E)CX, (E)DX, (E)SI, and (E)DI will be undefined. On an 80386 or 80486 the FS and GS segment registers will contain zero after the mode switch.

Programmer's Notes

- The address returned in BX:CX must only be called in real mode to switch into protected mode. The address returned in SI:(E)DI must only be called in protected mode to switch into real mode.
- 16-bit programs should call the address returned in SI:DI to switch from protected to real mode. 32-bit programs should call the address returned in SI:EDI.
- It is up to the caller to save and restore the state of the task when using this function to switch modes. This usually requires using the state save function (see page 63).
- The parameters must contain segment values appropriate for the mode that is being switched to. If invalid selectors are specified when switching into protected mode, an exception will occur.
- Applications may find functions 0300h, 0301h, 0302h, and 0304h more convenient to use than using this type of mode switching.

12. GET VERSION

Function 0400h returns the version of DPMI services supported. Note that this is not necessarily the version of any operating system that supports DPMI. It should be used by programs to determine what calls are legal in the current environment.

To Call

AX = 0400h

Returns

AH = Major version
AL = Minor version
BX = Flags
Bit 0 = 1 if running under an 80386 DPMI implementation
Bit 1 = 1 if processor is returned to real mode for reflected interrupts (as opposed to Virtual 8086 mode).
Bit 2 = 1 if virtual memory is supported
Bit 3 is reserved and undefined
All other bits are zero and reserved for later use
CL = Processor type
02 = 80286
03 = 80386
04 = 80486
DH = Current value of virtual master PIC base interrupt
DL = Current value of virtual slave PIC base interrupt
Carry flag clear (call can not fail)

Programmer's Notes

None

13. MEMORY MANAGEMENT SERVICES

These functions are provided to allocate linear address space.

13.1 Get Free Memory Information

This function is provided so that protected mode applications can determine how much memory is available. Under DPMI implementations that support virtual memory, it is important to consider issues such as the amount of available physical memory.

Note that since DPMI applications will often run in multi-tasking environments, this function must be considered only advisory.

To Call

AX = 0500h
ES:(E)DI = Selector:Offset of 30h byte buffer

Returns

If function was successful:
Carry flag is clear.
ES:(E)DI = Selector:Offset of buffer with the following structure:

Offset	Description
00h	Largest available free block in bytes
04h	Maximum unlocked page allocation
08h	Maximum locked page allocation
0Ch	Linear addr space size in pages
10h	Total number of unlocked pages
14h	Number of free pages
18h	Total number of physical pages
1Ch	Free linear address space in pages
20h	Size of paging file/partition in pages
24h-2Fh	Reserved

If function was not successful:
Carry flag is set.

Programmer's Notes

- 32-bit programs must use ES:EDI to point to the buffer. 16-bit programs should use ES:DI.
- DPMI implementations that do not support virtual memory (returned in flags from Get Version call) will only fill in the first field. This value specifies that largest allocation that could be made using function 0501h. Other fields will be set to -1.
- Only the first field of this structure is guaranteed to contain a valid value. All fields that are not returned by the DPMI implementation will be set to -1 (0FFFFFFFFh) to indicate that the information is not available.
- The field at offset 00h specifies the largest block of contiguous linear memory in bytes that could be allocated if the memory were to be allocated and left unlocked.
- The field at offset 04h specifies the number of pages that could be allocated. This is the value returned by field 00h / page size.
- The field at offset 08h specifies the largest block of memory in pages that could be allocated and then locked.
- The field at offset 0Ch specifies the size of the total linear address space in pages. This includes all linear address space that has already been allocated.
- The field at offset 10h specifies the total number of pages that are currently unlocked and could be paged out. This value also contains any free pages.
- The field at offset 14h specifies the number of physical pages that currently are not in use.
- The field at offset 18h specifies the total number of physical pages that the DPMI host manages. This value includes all free, locked, and unlocked physical pages.
- The field at offset 20h specifies the size of the DPMI host's paging partition or file in pages.
- To determine the size of pages for the DPMI host call the Get Page Size service (see page 78).

13.2 Allocate Memory Block

This function allocates and commits linear memory.

To Call

AX = 0501h
BX:CX = Size of memory block to allocate in bytes

Returns

If function was successful:
Carry flag is clear
BX:CX = Linear address of allocated memory block
SI:DI = Memory block handle (used to resize and free)

If function was unsuccessful:
Carry flag is set

Programmer's Notes

- This function does not allocate any selectors for the memory block. It is the responsibility of the caller to allocate and initialize any selectors needed to access the memory.
- Under DPMI implementations that support virtual memory the memory block will be allocated unlocked. If some or all of the memory should be locked you will need to use either the lock selector function or the lock linear region function.
- Under many implementations of DPMI, allocations will be page granular. This means that an allocation of 1001h bytes will result in an allocation of 2000h bytes. Therefore it is best to always allocate memory in multiples of 4K.

13.3 Free Memory Block

This function frees a memory block that was allocate through the allocate memory block function.

To Call

AX = 0502h
SI:DI = Handle of memory block to free

Returns

If function was successful:
Carry flag is clear

If function was unsuccessful:
Carry flag is set

Programmer's Notes

- Your program must also free any selectors that it allocated to point to the memory block.

13.4 Resize Memory Block

This function changes the size of a memory block that was allocated through the allocate memory block function.

To Call

AX = 0503h
BX:CX = New size of memory block to allocate in bytes
SI:DI = Handle of memory block to resize

Returns

If function was successful:
Carry flag is clear
BX:CX = New linear address of memory block
SI:DI = New handle of memory block

If function was unsuccessful:
Carry flag is set

Programmer's Notes

- This function may change the linear address of the memory block and the memory handle. Therefore, you will need to update any selectors that point to the block after resizing it. You must use the new handle instead of the old one.
- This function will generate an error if a memory block is resized to 0 bytes.

14. PAGE LOCKING SERVICES

These services are only useful under DPMI implementations that support virtual memory. They will be ignored by 16-bit DPMI implementations (although they will always return with carry clear to indicate success).

Some implementations of DPMI may ignore these calls. However, if the calls are ignored then the DPMI host will be able to handle page faults at arbitrary points during the application's execution including interrupt and exception handler code.

Although memory ranges are specified in bytes, the actual unit of memory that will be locked will be one or more pages. Page locks are maintained as a count. When the count is decremented to zero, the page is unlocked and can be swapped to disk. This means that if a region of memory is locked three times then it must be unlocked three times before the pages will be unlocked.

14.1 Lock Linear Region

This function locks a specified linear address range.

To Call

AX = 0600h
BX:CX = Starting linear address of memory to lock
SI:DI = Size of region to lock in bytes

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- If this function fails then none of the memory will be locked.
- If the specified region overlaps part of a page at the beginning or end of the region, the page(s) will be locked.

14.2 Unlock Linear Region

This function unlocks a specified linear address range that was previously locked using the Lock Linear Region function.

To Call

AX = 0601h
BX:CX = Starting linear address of memory to unlock
SI:DI = Size of region to unlock in bytes

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- If this function fails then none of the memory will be unlocked.
- An error will be returned if the memory was not previously locked or if the specified region is invalid.
- If the specified region overlaps part of a page at the beginning or end of the region, the page(s) will be unlocked.
- Even if the function succeeds, the memory will remain locked if the lock count is not decremented to zero.

14.3 Mark Real Mode Region as Pageable

Under some implementations of DPMI, all memory in virtual 8086 mode is locked by default. If a protected mode program is using memory in the first megabyte of address space, it is a good idea to use this function to turn off automatic page locking for regions of memory that will not be touched at interrupt time.

Do not mark memory as pageable in regions that are not owned by your application. For example, you should not mark all free DOS memory as pageable since it may cause a page fault to occur while inside of DOS (causing a crash). Also, do not mark the DPMI host data area as pageable.

It is very important to relock any real mode memory using function 0603h before terminating a program. Memory that remains unlocked after a program has terminated could result in fatal page faults when other software is executed in that address space.

Note that address space marked as pageable by this function can be locked using function 0600h. This function is just an advisory service to allow memory that does not need to be locked to be paged out. This function just disables any automatic locking of real mode memory performed by the DPMI host.

To Call

AX = 0602h
BX:CX = Starting linear address of memory to mark as pageable
SI:DI = Size of region to page in bytes

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- If this function fails then none of the memory will be unlocked.
- If the specified region overlaps part of a page at the beginning or end of the region, the page(s) will be not be marked as pageable.
- When your program terminates it should call function 0603h to relock the memory region.
- Unlike the lock and unlock calls, the pageability of the real mode region is maintained as a binary state, not a count. Therefore, do not call this function multiple times for a given linear region.

14.4 Relock Real Mode Region

This function is used to relock memory regions that were marked as pageable by the previous function.

To Call

AX = 0603h
BX:CX = Starting linear address of memory to relock
SI:DI = Size of region to page in bytes

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- If this function fails then none of the memory will be relocked.
- If the specified region overlaps part of a page at the beginning or end of the region, the page(s) will be not be relocked.

14.5 Get Page Size

This function returns the size of a single memory page in bytes.

To Call

AX = 0604h

Returns

If function was successful:
Carry flag is clear
BX:CX = Page size in bytes

If function was not successful:
Carry flag is set

Programmers Notes

None

15. DEMAND PAGING PERFORMANCE TUNING SERVICES

Some applications will discard memory objects or will not access objects for long periods of time. These services can be used to improve the performance of demand paging.

Although these functions are only relevant for DPMI implementations that support virtual memory, other implementations will ignore these functions (it will always return carry clear). Therefore your code can always call these functions regardless of the environment it is running under.

Since both of these functions are simply advisory functions, the operating system may choose to ignore them. In any case, your code should function properly even if the functions fail.

15.1 Reserved Subfunctions

Functions 0700h and 0701h are reserved and should not be called.

15.2 Mark Page as Demand Paging Candidate

This function is used to inform the operating system that a range of pages should be placed at the head of the page out candidate list. This will force these pages to be swapped to disk ahead of other pages even if the memory has been accessed recently. However, all memory contents will be preserved.

This is useful, for example, if a program knows that a given piece of data will not be accessed for a long period of time. That data is ideal for swapping to disk since the physical memory it now occupies can be used for other purposes.

To Call

AX = 0702h
BX:CX = Starting linear address of pages to mark
SI:DI = Number of bytes to mark as paging candidates

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- This function does not force the pages to be swapped to disk immediately.
- Partial pages will not be discarded.

15.3 Discard Page Contents

This function discards the entire contents of a given linear memory range. It is used after a memory object that occupied a given piece of memory has been discarded.

The contents of the region will be undefined the next time the memory is accessed. All values previously stored in this memory will be lost.

To Call

AX = 0703h
BX:CX = Starting linear address of pages to discard
SI:DI = Number of bytes to discard

Returns

If function was successful:
Carry flag is clear.

If function was not successful:
Carry flag is set.

Programmer's Notes

- Partial pages will not be discarded.

16. PHYSICAL ADDRESS MAPPING

Memory mapped devices such as network adapters and displays sometimes have memory mapped at physical addresses that lie outside of the normal 1Mb of memory that is addressable in real mode. Under many implementations of DPMI, all addresses are linear addresses since they use the paging mechanism of the 80386. This service can be used by device drivers to convert a physical address into a linear address. The linear address can then be used to access the device memory.

Some implementations of DPMI may not support this call because it could be used to circumvent system protection. This call should only be used by programs that absolutely require direct access to a memory mapped device.

To Call

AX = 0800h
BX:CX = Physical address of memory
SI:DI = Size of region to map in bytes

Returns

If function was successful:
Carry flag is clear.
BX:CX = Linear address that can be used to access the physical memory

If function was not successful:
Carry flag is set.

Programmer's Notes

- Under DPMI implementations that do not use the 80386 paging mechanism, the function will always succeed and the address returned will be equal to the physical address parameter passed into this function.
- It is up to the caller to build an appropriate selector to access the memory.
- Do not use this service to access memory that is mapped in the first megabyte of address space (the real mode addressable region).

17. VIRTUAL INTERRUPT STATE FUNCTIONS

Under many implementations of DPMI, the interrupt flag in protected mode will always be set (interrupts enabled). This is because the program is running under a protected operating system that can not allow programs to disable physical hardware interrupts. However, the operating system will maintain a "virtual" interrupt state for protected mode programs. When the program executes a *cli* instruction, the program's virtual interrupt state will be disabled, and the program will not receive any hardware interrupts until it executes an *sti* to reenable interrupts (or calls service 0901h).

When a protected mode program executes a *pushf* instruction, the real processor flags will be pushed onto the stack. Thus, examining the flags pushed on the stack is not sufficient to determine the state of the program's virtual interrupt flag. These services enable programs to get and modify the state of their virtual interrupt flag.

The following sample code enters an interrupt critical section and then restores the virtual interrupt state to it's previous state.

```
;
; Disable interrupts and get previous interrupt state
;
        mov     ax, 0900h
        int     31h
;
; At this point AX = 0900h or 0901h
;
        .
        .
        .
;
; Restore previous state (assumes AX unchanged)
;
        int     31h
```

17.1 Get and Disable Virtual Interrupt State

This function will disable the virtual interrupt flag and return the previous state of the virtual interrupt flag.

To Call

AX = 0900h

Returns

Carry flag clear (this function always succeeds)
Virtual interrupts are disabled
AL = 0 if virtual interrupts were previously disabled
AL = 1 if virtual interrupts were previously enabled

Programmer's Notes

- AH will not be changed by this procedure. Therefore, to restore the previous state, simply execute an Int 31h.

17.2 Get and Enable Virtual Interrupt State

This function will enable the virtual interrupt flag and return the previous state of the virtual interrupt flag.

To Call

AX = 0901h

Returns

Carry flag clear (this function always succeeds)
Virtual interrupts are enabled
AL = 0 if virtual interrupts were previously disabled
AL = 1 if virtual interrupts were previously enabled

Programmer's Notes

- AH will not be changed by this procedure. Therefore, to restore the previous state, simply execute an Int 31h.

17.3 Get Virtual Interrupt State

This function will return the current state of the virtual interrupt flag.

To Call

AX = 0902h

Returns

Carry flag clear (this function always succeeds)
AL = 0 if virtual interrupts are disabled
AL = 1 if virtual interrupts are enabled

Programmer's Notes

None

18. GET VENDOR SPECIFIC API ENTRY POINT

Some DOS extenders provide extensions to the standard set of DPMI calls. This call is used to obtain an address which must be called to use the extensions. The caller points DS:(E)SI to a null terminated string that specifies the vendor name or some other unique identifier to obtain the specific extension entry point.

To Call

AX = 0A00h
DS:(E)SI = Pointer to null terminated string

Returns

If function was successful:
Carry flag is clear
ES:(E)DI = Extended API entry point
DS, FS, GS, EAX, EBX, ECX, EDX, ESI, and EBP may be modified

If function was not successful:
Carry flag is set

Programmer's Notes

- Execute a far call to call the API entry point.
- All extended API parameters are specified by the vendor.
- The string comparison used to return the API entry point is case sensitive.

19. DEBUG REGISTER SUPPORT

The 80386 processor supports special registers that are used for debugging. Since the instructions to modify these registers can only be executed by code running at privileged level zero, protected mode debuggers running in DPMI environments can not modify the registers directly. These services provide mechanisms for setting and clearing debug watchpoints and detecting when a watchpoint has caused a fault.

19.1 Set Debug Watchpoint

This function will set a debug watchpoint at a specified linear address.

To Call

AX = 0B00h
BX:CX = Linear address of watchpoint
DL = Size of watchpoint (1, 2, or 4)
DH = Type of watchpoint
    0 = Execute
    1 = Write
    2 = Read/Write

Returns

If function was successful:
Carry flag is clear
BX = Debug watchpoint handle

If function was not successful:
Carry flag is set

Programmer's Notes

None

19.2 Clear Debug Watchpoint

This function will clear a debug watchpoint that was set using the Set Debug Watchpoint function.

To Call

AX = 0B01h
BX = Debug watchpoint handle

Returns

If function was successful:
Carry flag is clear

If function was not successful:
Carry flag is set

Programmer's Notes

- This call frees the debug watchpoint handle

19.3 Get State of Debug Watchpoint

This function returns the state of a debug watchpoint that was set using the Set Debug Watchpoint function.

To Call

AX = 0B02h
BX = Debug Watchpoint Handle

Returns

If function was successful:
Carry flag is clear
AX = Status flags
Bit 0 = 1 if watch point has been executed

If function was not successful:
Carry flag is set

Programmer's Notes

- To clear the watchpoint state the caller must use function 0B03h.

19.4 Reset Debug Watchpoint

This function resets the state of a previously defined debug watchpoint.

To Call

AX = 0B03h
BX = Debug Watchpoint Handle

Returns

If function was successful:
Carry flag is clear

If function was not successful:
Carry flag is set

Programmer's Notes

None

20. OTHER APIS

In general, any software interrupt interface that passes parameters in the EAX, EBX, ECX, EDX, ESI, EDI, and EBP registers will work as long as none of the registers contains a segment value. In other words, if a software interrupt interface is completely register based without any pointers, segment register, or stack parameters, that API could work under any DPMI implementation.

More complex APIs require the caller to use the translation services described on page 52.

21. NOTES FOR DOS EXTENDERS

Many programs that use DPMI will be bound to DOS extenders so that they will be able to run under any DOS environment. Existing DOS extenders support APIs that differ from the Int 31h interface. Usually, DOS extenders use an Int 21h multiplex for their extended APIs.

Extenders that support DPMI will need to initialize differently when they are run under DPMI environments. They will need to enter protected mode using the DPMI real to protected mode entry point, install their own API handlers, and then load the DOS extended application program.

21.1 Initialization of Extenders

DOS extenders should check for the presence of DPMI before attempting to allocate memory or enter protected mode using any other API. DOS extenders should check for APIs in the following order:

DOS Protected Mode Interface
Virtual Control Program Interface
eXtended Memory Specification
Int 15h memory allocation

When DPMI services are detected, extenders that provide interfaces that extend or are different from the basic DPMI interface will switch into protected mode and initialize any internal data structures. DPMI compatible extenders that provide no API extensions should simply execute the protected mode application in real mode.

21.2 Installing API Extensions

DOS extenders typically use Int 21h to implement API extensions. Under DPMI, a DOS extender will need to install an API translation library by hooking Int 21h via then get and set protected mode interrupt vector functions (see page 50). The DOS extender library then gets to see every DOS call executed by the application program. If the API does not have any pointers then the interrupt can be reflected to the original interrupt handler. The default handler will pass the interrupt to real mode. Other APIs can be explicitly mapped by the DOS extender.

WARNING: The translation library code should be in locked memory to prevent page faults while DOS is in a critical section. This could happen, for instance, if a program called DOS reentrantly from an Int 24h (critical error).

21.3 Loading the Application Program

Once the API translation library has been initialized, the DOS extender can load the application program using standard DOS calls. Memory should be allocated using the DPMI memory allocation services.

21.4 Providing API Extensions

DPMI call 0A00h provides a standard mechanism for providing vendor specific extensions to the standard APIs. To support extensions under a DPMI environment, the translation library should hook the Int 31h chain (using the DOS get/set vector calls) and watch for call 0A00h. When this call is issued with the proper string parameter, the Int 31h hook code should modify ES:(E)DI, clear the carry flag on the stack, and iret without passing the call down the Int 31h chain. If the string passed in ES:(E)DI does not match the extensions supported by the library then the call should be passed down the Int 31h chain.