DOS PROTECTED MODE INTERFACE (DPMI) SPECIFICATION

Version 1.0
March 12, 1991

Application Program Interface (API) for
Protected Mode DOS Applications

Please send written comments to:
Albert Teng, DPMI Committee Secretary
Mailstop: NW1-18
2801 Northwestern Parkway
Santa Clara, CA 95051
FAX: (408) 765-5165

Intel order no.: 240977-001

Copyright

The founding members of the DPMI Committee are: Borland International, IBM Corporation, Ergo Computer Solutions, Incorporated, Intelligent Graphics Corporation, Intel Corporation, Locus Computing Corporation, Lotus Development Corporation, Microsoft Corporation, Phar Lap Software, Incorporated, Phoenix Technologies Ltd, Quarterdeck Office Systems, and Rational Systems, Incorporated.

Software vendors can receive additional copies of the DPMI Specification at no charge by contacting Intel Literature JP26 at (800) 548-4725, or by writing Intel Literature JP26, 3065 Bowers Avenue, P.O. Box 58065, Santa Clara, CA 95051-8065. DPMI Specification Version 0.9 will be sent out along with Version 1.0 for a period about six months. Once DPMI Specification Version 1.0 has been proven by the implementation of a host, Version 0.9 will be dropped out of the distribution channel.

Disclaimer of Warranty

THE DPMI COMMITTEE EXCLUDES ANY AND ALL IMPLIED WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. NO DPMI COMMITTEE MEMBER MAKES ANY WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS SPECIFICATION, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. NO MEMBER OF THE DPMI COMMITTEE SHALL HAVE ANY LIABILITY FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RESULTING FROM THE USE OR MODIFICATION OF THIS SPECIFICATION.

Warning

This DPMI Version 1.0 Specification has not yet been validated by the implementation of a DPMI 1.0-compliant host. Please report any errors or ambiguities in writing to the DPMI Committee Secretary.

Table of Contents

Chapter 1. Introduction

The DOS Protected Mode Interface (DPMI) allows DOS programs to access the advanced features of 80286-, 80386-, and 80486-based PCs in a well-behaved, hardware-independent fashion that does not compromise system protection. DPMI functions are defined to manage local descriptor tables, perform mode switching, allocate extended memory, allocate DOS (conventional) memory, control the interrupt subsystem, communicate with real mode programs, and read or write certain CPU control registers. Protected mode multitasking environments, memory managers, or operating systems that implement the DPMI functions are called *DPMI hosts*; protected mode applications that request DPMI functions (directly or indirectly) are called *DPMI clients*.

The predecessor to DPMI is the Virtual Control Program Interface (VCPI), developed in 1987 by Phar Lap Software and Quarterdeck Office Systems. VCPI allows 80386 protected-mode DOS Extender applications to coexist with 80386-specific memory managers and expanded memory (EMS) emulators. In the VCPI model, the protected mode DOS application is the client and the EMS emulator is the server, which is invoked via an extension of the EMS Int 67H interface to switch between real mode and protected mode, allocate memory, and inspect or set the 80386 debug registers. If a protected mode application is loaded and a VCPI server is not present, the application simply assumes complete control of the machine and carries out the necessary hardware manipulations directly.

VCPI has been extremely successful, but it can not support the full virtualization needed for *multitasking* of DOS-based protected mode applications. VCPI allows client programs to run at the highest privilege level (Ring 0), making it impossible for a VCPI server to enforce device virtualization for many devices (for example, to run EGA/VGA graphical applications in a window), provide centralized virtual memory management services, or shield one protected mode application from interference by another. Another, somewhat less important drawback of VCPI is that it is fundamentally based on the concept of 80386 hardware paging, and therefore it cannot be implemented on 80286 machines.

A DPMI host is similar in many respects to a VCPI server, in that it provides mode-switching and extended memory management services to client programs. But unlike a VCPI server, a DPMI host can run at a more privileged level than its clients and can use the hardware to enforce a "supervisor/user" protection model. This allows a DPMI host to support centralized virtual memory and maintain full control over client programs' address spaces and access to the hardware. Furthermore, DPMI's functions for memory and interrupt management are more general than those in the VCPI, so DPMI can also be implemented on 80286 machines.

VCPI and DPMI are viewed as complementary standards. Developers can use DPMI to achieve broad compatibility requirements of their DOS-based protected-mode applications, since the majority of PC operating environments can or will support DPMI. Developers can use VCPI to implement system-level device drivers and other programs that require the 80386's highest privilege level.

The initial DPMI prototype was developed by Microsoft for Windows version 3.0, with input from Lotus Corporation and Rational Systems, as part of a general effort to enhance Windows' performance by allowing the Windows kernel to run in extended memory. In parallel, Intel was working with manufacturers of multitasking environments, EMS emulators, and DOS extenders to ensure that an extended VCPI specification could fully utilize the 80386's virtualization and protection features. In February 1990, the parties involved in the above activities agreed to form the DPMI Committee and formulate an industry-wide standard for protected-mode DOS applications. The Committee released the first public DPMI Specification, Version 0.9, in May 1990.

(See Appendix A: Glossary for the definitions of DPMI, DPMI client, DPMI host, Expanded memory, Expanded memory emulator, EMS, Extended memory, and VCPI.)

Chapter 2. DPMI Execution Environment Overview

This section provides a brief overview of such fundamental DPMI concepts as DPMI virtual machines, multiple clients in the same virtual machine, multiple clients in different virtual machines, DPMI multitasking, a unique LDT and IDT per client, extended memory allocation, flat mode and segmented mode memory models, and virtual memory's relationship to page locking and virtual DMA services.

CPU Mode and DPMI Virtual Machines

The DPMI interface can be implemented on any microprocessor that supports 80286 protected mode execution. In other words, on any 80286, 80386, 80486 or other compatible CPUs. However, the full potential of the DPMI interface can only be realized on 80386 or later CPUs.

The Intel 80386 architecture has three operating modes:

- Real Mode. In this mode, the 80386 microprocessor is essentially a fast 8086 and is fully compatible with existing DOS programs.

- Protected Mode. The 80386 microprocessor supports a full 32-bit programming model, with 32-bit registers and addressing modes. In the protected mode, the 80386 microprocessor’s on-chip paged memory management unit is accessible. The 80386 microprocessor can also run 16-bit 286 programs.

- Virtual-86 Mode. In the virtual-86 mode, one or more 8086-compatible programs can be running at the same time, each in its own 'private' address space called a *virtual machine* (VM). Each virtual DOS machine (or virtual DOS environment) appears to be a complete real mode DOS environment to the program running with it. The 80386 processor forces virtual-86 mode programs to run at Ring 3 so that a host operating environment can fully virtualize hardware interrupts, I/O and processor exceptions through the 80386's protection checking mechanisms. (See the definition of Virtual DOS environment in Appendix A: Glossary.)

DPMI services are tied to DOS services and there is always a DOS real mode environment available. In DPMI implementations that are designed to take advantage of the 80386 and later CPUs, a DOS environment will typically run in virtual-86 mode rather than real mode. To avoid describing the virtual-86 and real modes repetitively, the term <u>real mode is used to refer both the real mode and the virtual-86 mode throughout the specification</u>. The term virtual machine is also used to refer to a real mode machine whenever applicable throughout the specification. The term "80386" as used in the specification refers to the Intel 80386 and all later CPUs that are fully compatible with it.

Each time a real mode program calls the DPMI interface to request the initial switch to protected mode, a new DPMI client is created (this process is explained in more detail later). A DOS environment and the one or more protected mode DPMI clients which were launched from it are collectively referred to as a *(DPMI) virtual machine*. In this specification, we often simply use virtual machine to refer both the virtual-86 machine and the DPMI virtual machine with the understanding that the DPMI virtual machine concept is also relevant on 80286 CPUs even though there is no hardware virtual-86 mode support for it.

DPMI Clients and Their Relationships to a Virtual Machine

Many multitasking hosts can run several DPMI virtual machines concurrently. The DPMI clients within the same virtual machine form a program stack with the first (least-recently-created) client sitting at the bottom of the stack and the last (most-recently-created) client sitting at the top of the stack. The topmost client is called the *primary client*. Only one client in the program stack of a VM can be active at one time and the client is called the *current client*.

Each virtual machine has its own private virtual address space. Clients within a virtual machine share the address space of the virtual machine. However, the method of the virtual machine address space is shared is quite different in DPMI 0.9 and DPMI 1.0.

Under a DPMI 0.9 host, all of the DPMI clients in a virtual machine share a single local descriptor table (LDT) and use the same interrupt descriptor table (IDT). When multiple virtual machines are present, each has a different LDT and IDT. Thus, the clients within a virtual machine completely share addressing but the clients in separate machines are isolated from each other. This is fairly staightforward but does not allow 16-bit and 32-bit clients to execute in the same virtual machine.

Under a DPMI 1.0 host, each DPMI client within a virtual machine has its own LDT and IDT; consequently, each client has a distinct context for addressing and the handling of exceptions and interrupts that is not visible to other clients in the same virtual machine or other virtual machines. When a client becomes "current" by virtue of a switch form real to protected mode (by any of several means to be described later), its LDT and IDT become active and any software interrupts and protected mode exceptions it issues are reflected through its IDT. However, the DPMI host is responsible for ensuring that hardware interrupts and real mode exceptions are sent to the primary client of the virtual machine, whether or not the primary client is the current client. It should be noted that in 32-bit DPMI environments, the clients in a particular virtual machine will share the same linear address space defined by a single page table directory, but this does not have any effect on client execution.

DPMI Multitasking Support

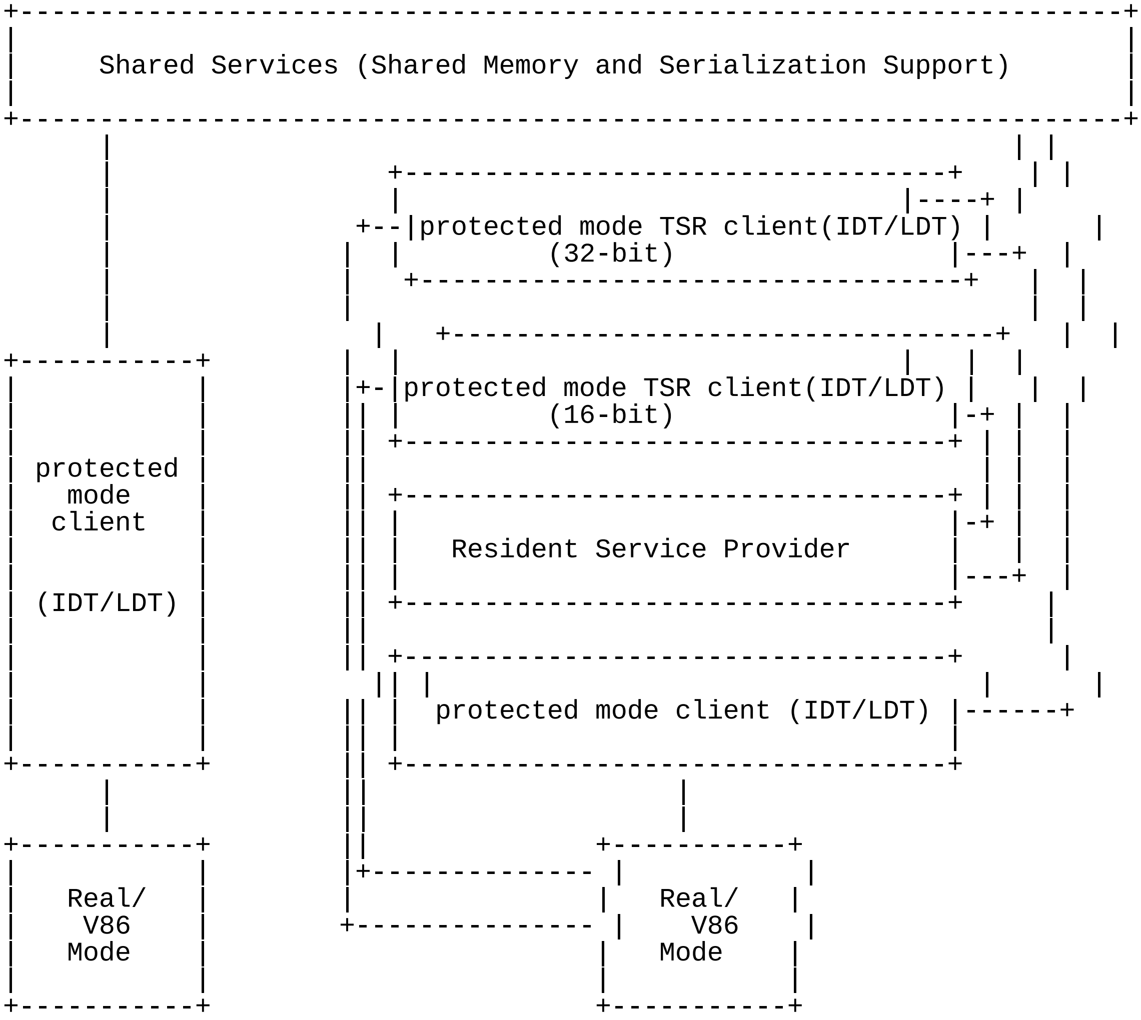
Some DPMI implementations can support several DPMI virtual machines concurrently, and can provide true pre-emptive multitasking by treating each virtual machine as an independently-dispatchable task. There is no multitasking among the clients within the same virtual machine that is defined by DPMI. Therefore, a DPMI client should not assume it owns the resources of the entire machine. In particular, a DPMI client should use the Int 2FH Function 1680H DPMI call to release their CPU time slice when it is idle (for example, when it is polling for keyboard input). This allows the DPMI host to pass the CPU to other clients, or take power-conserving measures on laptop and notebook computers.

In order to protect system integrity, DPMI implementations which are specific to 80386 and later CPUs will typically use the hardware protection mechanisms to virtualize input, output, and interrupts, and to enforce a "supervisor/user" privilege model. Accordingly, unlike DOS and VCPI clients, DPMI clients should make no assumptions about the privilege level at which they will run. For example, when calculating selectors or constructing descriptors to be copied into the LDT, a client must set the privilege bits equal to its CPL. (One way for a client to determine its CPL is to examine the Requestor's Privilege Level field of CS created for the client by the host.)

Under DPMI V 1.0, communication between clients in *different* virtual machines can be accomplished *only* with DPMI shared memory blocks. Clients within the *same* VM can also communicate by DPMI shared memory block as well as by resident service providers, DOS memory blocks, or a switch to real mode (Appendix A:Glossary for information of resident service providers, shared services and memory blocks). DOS memory blocks allocated by a DPMI client will only be visible to other clients running in the same virtual machine. Similarly, resident service providers will only be notified about the loading and termination of DPMI clients in the same virtual machine. However, DPMI shared memory blocks can be accessed by clients in any virtual machine. The shared memory blocks can contain either code or data. DPMI hosts will guarantee that the same linear address is provided to a shared memory block across all virtual machines and clients. The identical linear address requirement ensures that a shared code memory block can execute properly when the memory block is accessed by clients with different LDTs.

Figure 1 on page 9 shows two examples of DPMI virtual machines. The first virtual machine is a virtual machine with a single DPMI client extended from DOS. The client can access a shared memory block. The second virtual machine is a virtual machine with three DPMI clients forming a stack in the address space of the virtual machine. Only one of the clients in the second virtual machine can be active at a time while the two virtual machines could be multitasked under the host. Two of the clients in the second virtual machine obtain protected mode services from a resident service provider that installed itself using the DPMI interface. Three out of the four clients in the two example virtual machines are shown to have access to shared services.

Figure 1. This example uses two DPMI virtual machines to demonstrate the relationship between DPMI clients, shared memory blocks, resident service providers, and DOS address spaces on a 32-bit DPMI host which supports multiple virtual machines.


Shared Services (Shared Memory and Serialization Support)
protected mode TSR client(IDT/LDT)
(32-bit)
protected mode TSR client(IDT/LDT)
(16-bit)
protected mode client
(IDT/LDT)
Resident Service Provider
protected mode client (IDT/LDT)
Real/ V86 Mode
Real/ V86 Mode

Example Virtual machine 1
(single client)

Example Virtual Machine 2
(multiple clients, mixed bitness)

Extended Memory Allocation Environment

A DPMI client can use DPMI calls to allocate *extended memory* above 1 megabyte in protected mode and then to allocate and initialize descriptors that make the memory addressable. DPMI supports the use of 80286 and 80386 extended memory. However, only 32-bit DPMI hosts can support DPMI calls with attributes for 32-bit addressing and paged memory management. Under such hosts, 32-bit clients can be implemented using either a flat or segmented memory model.

- The "flat" memory model represents the application's addresss space as a single array of up to 4 gigabytes. This model essentially "turns off" the segmentation features by initially loading the segment registers with selectors for descriptors that encompass the entire 32-bit address space. Once loaded, the segment registers don't need to be changed and all memory references are "near".

- The segmented memory model represents the application's address space as a collection of segments up to 4 gigabytes each. (Contrast with the 80286, which has a limit of 64 kilobytes per segment.)

DPMI offers a rich set of functions for the allocation and management of both extended memory blocks and descriptors. The table in Appendix D summarizes the various means of descriptor allocation and rules for the use of descriptors in DPMI functions.

Virtual Memory, Page Locking and Virtual DMA Services

Many implementations of DPMI support demand-paged virtual memory allowing DPMI clients to allocate more memory than is physically present in the machine (limited only by the amount of backing store).   In many of such implementations, the extended memory owned by a client is by default swappable or pageable, while the DOS memory in the client's virtual machine is by default locked. DPMI functions are provided that allow the client to lock and unlock both extended memory and DOS memory, and to supply information to the DPMI host that may improve virtual memory performance.

DPMI clients must be careful to lock any memory that might be accessed during a call to DOS, or that is used by interrupt handler, because the DPMI host may not be able to demand-load the client's memory under those circumstances (for example, if the DPMI host uses the DOS file system and DOS is already busy). In DPMI implementations that can handle all possible virtual memory faults, or that do not support virtual memory at all, the lock requests will return a success status but will have no effect.

The virtual memory addresses used by a protected mode program have no direct correspondence to physical memory addresses; virtual addresses which appear to be contiguous may be mapped to discontiguous physical addresses, or may not be currently mapped to physical memory at all (if some of the application's memory has been swapped or paged to backing store). DPMI clients that actually need physical addresses (e.g., to issue commands to a bus-master device or an on-board DMA controller) must use Virtual DMA Services (VDS) services (see Appendix A: Glossary) to communicate with the host in order to achieve accurate physical memory access.

The High Memory Area and Extended Memory Specification

A client program cannot assume that the High Memory Area (HMA) from 1 MB to 1 MB+64 KB-16 B is available in protected mode unless it has explicitly enabled the A20 address line with the appropriate eXtended Memory Specification (XMS) function call before entering protected mode. XMS is the recommended standard way to address the HMA (for additional information about HMA and XMS, see Appendix A: Glossary). Note that this restriction is only important for software that specifically needs to access the HMA.

In reality, the A20 address line is *always* enabled while executing protected mode code under a DPMI host, but some 80386-specific hosts simulate a 1 MB address wrap for compatibility reasons. Under these conditions, the HMA will not be accessible unless the A20 was previously enabled through the XMS interface.

Chapter 3. DPMI Services Overview

DPMI host programs make the services defined in this document available to DPMI clients via software interrupts 2FH and 31H. Int 2FH is the so-called DOS Multiplex Interrupt, used by many different drivers and resident utilities, and most of the DPMI functions supported on Int 2FH can be called in either real mode or protected mode. Furthermore, some of the Int 2FH functions described in this document may also be supported by other types of 80386 control programs or multitasking operating systems.

The Int 2FH functions are:

Function 1680H	Release Current Virtual Machine's Time Slice
Function 1686H	Get CPU Mode
Function 1687H	Return Real-to-Protected Mode Switch Entry Point
Function 168AH	Get Vendor-Specific API Entry Point

The Int 2FH functions will not be discussed further in this overview section. For additional information on the Int 2FH functions, see the Implementation Notes for DPMI Clients and DPMI Function Reference.

The DPMI Int 31H functions are only available to programs which are executing in protected mode. A client invokes one of these functions by placing a function number in AX, passing other parameters in registers or by reference, and executing a software interrupt 31H. For example:

```
mov   ax,function_number   ; select DPMI function
.                          ; load other registers with
.                          ; function specific parameters
int   31h                  ; transfer to DPMI host
jc    error                ; jump if function failed
```

If the function succeeds, it returns the Carry flag clear and (most commonly) other results in registers or data structures in the client's address space; if the function fails, it returns with the Carry flag set and an error code in AX (see Appendix B). DPMI functions preserve all contents of registers and flags that are not otherwise specified in this document; similarly, all DPMI functions can be assumed to be reentrant unless this document explicitly states otherwise. Some DPMI function errors, such as those caused by the client passing an invalid pointer, will cause the host to fault; the client can detect these events by the installation of an exception handler.

Certain implementations of DPMI, called 32-bit hosts, take advantage of the special features of 80386 and later CPUs and can support 32-bit clients as well as 16-bit clients. On such DPMI hosts, Int 31H functions that take pointers as parameters will use the extended 32-bit registers for offsets (for example, ES:EDI instead of ES:DI) when called by 32-bit clients. The high word of the 32-bit registers will be ignored (and preserved) when DPMI services are requested by 16-bit clients.

The Int 31H DPMI services made available to protected mode programs fall into eight general categories:

- Extended Memory Management Services
- LDT Descriptor Management Services
- Page Management Services
- Interrupt Management Services
- Translation Services
- DOS Memory Management Services
- Debug Support Services
- Miscellaneous Services

A general introduction to the Int 31H functions in each of these categories is provided below. For detailed information on the parameters and results of any particular function, see the DPMI Function Reference section of this document.

Extended Memory Management Services

The fundamental services in this group are:

0501H	Allocate Memory Block
0502H	Free Memory Block
0503H	Resize Memory Block
050AH	Get Memory Block Size and Base

These functions work with blocks of linear memory above 1 megabyte and deal with "linear addresses." It's important to note that allocation of memory by these functions does not provide addressability; once an application owns a block of linear memory, it must then use separate DPMI function calls to allocate and initialize descriptors that will provide addressability for the memory, or modify the base and limit fields of existing descriptors.

The following functions are only available on 32-bit DPMI hosts:

0504H	Allocate Linear Memory Block
0505H	Resize Linear Memory Block

These services differ from Functions 0501H and 0503H in that they always page-align allocated blocks, they allow memory blocks to be requested at specific linear addresses, and they draw a distinction between the allocation of linear address space (represented by *uncommitted pages*, see Appendix A: Glossary) and the allocation of linear memory (represented by *committed pages*, see Appendix A: Glossary). Again, the client is responsible for setting up appropriate descriptors for the memory with separate function calls.

The next two functions, also available only on 32-bit hosts, allow the inspection or manipulation of page attributes within memory blocks previously allocated with Functions 0504H and 0505H:

0506H	Get Page Attributes
0507H	Set Page Attributes

The attribute information maintained on a page-by-page basis includes: whether the page is committed or uncommitted, *mapped* (i.e., the page's linear addresses are being used as aliases for a special type of memory object), read-only or writable, and (if the DPMI host supports these capabilities) whether the page has been accessed or modified.

DPMI's support for interprocess communication between protected mode clients via named shared memory segments is implemented in the following functions:

0D00H	Allocate Shared Memory
0D01H	Free Shared Memory
0D02H	Serialize on Shared Memory
0D03H	Free Serialization on Shared Memory

The serialization functions allow applications to temporarily reserve the right of access to shared memory for inspection or update operations, or to simply use a shared memory block handle as a semaphore. Client programs must establish addressability to shared memory by building descriptors in a separate operation.

The last few functions in this group return information about the physical and virtual memory of the system:

0500H	Get Free Memory Information
050BH	Get Memory Information
0604H	Get Page Size

Because many DPMI hosts will support multitasking, some shared resource results returned by the memory information functions are subject to change and should be viewed only as advisory. Other applications may be competing for the same memory resources and/or the DPMI host may choose to limit the amount of memory visible to a particular client.

LDT Descriptor Management Services

The functions for LDT descriptor management are:

0000H	Allocate LDT Descriptor
0001H	Free LDT Descriptor
0002H	Map Real-Mode Segment to Descriptor
0003H	Get Selector Increment Value
0006H	Get Segment Base Address
0007H	Set Segment Base Address
0008H	Set Segment Limit
0009H	Set Descriptor Access Rights
000AH	Create Alias Descriptor
000BH	Get Descriptor
000CH	Set Descriptor
000DH	Allocate Specific LDT Descriptor
000EH	Get Multiple Descriptors
000FH	Set Multiple Descriptors

The LDT descriptor management services allocate, modify, inspect, and free protected mode descriptors in the current client's Local Descriptor Table (LDT). The functions in this group do not allocate memory - either physical memory or virtual address space- in most cases, the memory being mapped by a new or modified descriptor will have been previously allocated using one of the functions in the Extended Memory Management group. Similarly, when a descriptor is released, the corresponding memory usually must be released with a separate function call. LDT management functions that return selectors will set the correct RPL bits for the client's privilege level; functions that take selectors as input ignore the RPL bits of the incoming value.

Descriptors created or modified through LDT descriptor management services may only reference linear addresses between 0 and a host-specific upper bound. Clients may determine this upper bound by using the Get Memory Information call (Int 31H Function 050BH). Clients of 32-bit hosts may construct descriptors to reference anywhere within this range, whether or not memory has been allocated for the linear addresses referenced by the descriptor. 16-bit hosts may be more restrictive with regard to what linear addresses they will allow descriptors to reference: they may reject requests to point a descriptor to addresses that are neither in memory blocks that have been allocated by the client nor in conventional memory. To eliminate any possibility of incompatibility, 16-bit clients should ensure that they only point descriptors to memory that they have previously allocated through the DPMI memory allocation functions.

16-bit DPMI hosts may handle memory blocks at addresses that the client might not anticipate. For example, a 16-bit DPMI host on a 286 could implement virtual memory by allocating memory as "virtual handles" within a 32-bit address space. Consequently, clients should always use the full 32-bit value of the address of a memory block when constructing descriptors to point into the memory block, even when running on a 286 processor.

Page Management Services

The services in this group are only useful under DPMI hosts that support page-oriented virtual memory.

Four functions are provided that allow an application to notify the DPMI host that memory is or is not eligible for paging to disk:

0600H	Lock Linear Region
0601H	Unlock Linear Region
0602H	Mark Real Mode Region as Pageable
0603H	Relock Real Mode Region

These functions are ignored by DPMI implementations that do not support virtual memory; they will always return the Carry flag clear to indicate success, but have no other effect. DPMI hosts which support virtual memory may also choose to ignore these calls, but only if the client can't tell that they have been ignored; for example, such hosts must be able to handle page faults transparently at arbitrary points during a client's execution, including within interrupt and exception handlers.

Although the addresses and lengths of memory regions are specified to these functions in bytes, memory is always locked or unlocked in terms of pages. Page locks are maintained as a count; a client can make multiple requests to lock the same page, and that page will not be eligible for swapping until an equivalent number of unlock requests have been made and the lock count has been decremented to zero.

Two additional functions allow applications to advise the DPMI host about their use of memory:

0702H	Mark Page as Demand Paging Candidate

0703H	Discard Page Contents

When applications discard memory objects or do not access objects for long periods of time, they can call these functions to help the host improve the performance of the demand paging. Since these functions are simply advisory functions, the host may choose to ignore them, and of course, hosts without virtual memory will *always* ignore them. In any case, DPMI clients must function properly whether or not the functions have any effect.

Interrupt Management Services

The following interrupt management services allow protected mode applications to intercept real and protected mode interrupts and hook processor exceptions:

0200H	Get Real Mode Interrupt Vector
0201H	Set Real Mode Interrupt Vector
0202H	Get Processor Exception Handler Vector
0203H	Set Processor Exception Handler Vector
0204H	Get Protected Mode Interrupt Vector
0205H	Set Protected Mode Interrupt Vector
0210H	Get Extended Processor Exception Handler Vector for Protected Mode
0211H	Get Extended Processor Exception Handler Vector for Real Mode
0212H	Set Extended Processor Exception Handler Vector for Protected Mode
0213H	Set Extended Processor Exception Handler Vector for Real Mode

The interrupt management interface seems elaborate, but the large number of apparently overlapping services exists for a good reason. First, it allows client applications to differentiate their handling of exceptions (internal interrupts caused by erroneous operations, addressing problems, and so on) from their handling of interrupts generated by external hardware devices or by execution of the `INT` instruction - even when these occur on the same interrupt number. Secondly, Functions 0202H and 0203H are superseded in DPMI 1.0 by Functions 0210H through 0213H. The latter supply extended information in the stack frame with a common structure for both 16-bit and 32-bit clients, and allow clients to install separate protected mode handlers for exceptions that occur in real mode and protected mode.

The next three functions allow clients to cooperate with the DPMI host in maintaining a "virtual interrupt" flag:

0900H	Get and Disable Virtual Interrupt State
0901H	Get and Enable Virtual Interrupt State
0902H	Get Virtual Interrupt State

For performance reasons, some DPMI hosts will always run in protected mode with the interrupt flag set (interrupts enabled). Such hosts will maintain a "virtual" interrupt state for each protected mode client, so that clients can create critical segments of code and protect vital data structures from their own interrupt service routines. When the program executes a `CLI` instruction or invokes Function 0900H, the program's virtual interrupt state will be disabled, and the program will not receive any hardware interrupts until it executes an `STI` or calls Function 0901H to re-enable interrupts. Note that on such systems `PUSHF` pushes the real (not virtual) interrupt flag onto the stack, and `POPF` and `IRET(D)` do not affect either the real or virtual interrupt status.

Translation Services

The first three services to be discussed in this group are provided so that protected mode programs can call real mode software directly:

0300H	Simulate Real Mode Interrupt
0301H	Call Real Mode Procedure with Far Return Frame
0302H	Call Real Mode Procedure with Interrupt Return Frame

All three services are used in essentially the same way. The protected mode program sets up a data structure that contains values for every real mode register, then issues the function call. The DPMI host saves the protected mode registers, switches the CPU into real mode, loads the registers from the data structure, and then transfers control to the designated address. Parameters may also be passed to the real mode procedure on the stack, if necessary. When the real mode procedure returns, the host stores the (possibly modified) register contents (except for SS, SP, CS, and IP) back into the same data structure, switches the CPU to protected mode, restores the protected mode registers, and resumes execution of the protected mode client. The client can then inspect the data structure to determine the results of the function call.

The three functions listed above differ only in whether the destination real mode routine must exit by a far return or by an interrupt return (`IRET`), and in how the address of the destination real mode routine is supplied to the DPMI host. Function 0300H takes the destination address from a real mode interrupt vector, and ignores the CS and IP fields in the data structure, while Functions 0301H and 0302H obtain the destination address from the data structure. Function 0301H assumes that the real mode routine will exit with a far return, while Functions 0300H and 0302H require the real mode routine to terminate with an interrupt return.

DPMI also provides a mechanism, called a real mode callback, by which real mode software can call a protected mode program with implicit mode switches. A DPMI client creates and destroys such linkages with the following functions:

0303H	Allocate Real Mode Callback Address
0304H	Free Real Mode Callback Address

Real mode callbacks can be used to service interrupts that occur in real mode with a protected mode handler, or to provide protected mode services to real mode programs. For example, many mouse drivers can be configured to call an application-defined address whenever there is a change in the mouse's state. By allocating a real mode callback address and then passing the address to the mouse driver, a protected mode application can arrange to be notified of mouse movements or button clicks even though the mouse driver runs entirely in real mode.

The last two functions in this group give client programs access to the primitive building blocks for the high-level Functions 0300H through 0302H:

0305H	Get State Save/Restore Addresses
0306H	Get Raw CPU Mode Switch Addresses

These functions allow mode switching to be performed much more quickly, but place the entire burden of maintaining the proper system context in each mode on the client program.

DOS Memory Management Services

The DPMI DOS memory management functions are:

0100H	Allocate DOS Memory Block
0101H	Free DOS Memory Block
0102H	Resize DOS Memory Block

These DPMI Int 31H functions are exactly analogous to the DOS real mode functions Allocate Memory Block (Int 21H Function 48H), Free Memory Block (Int 21H Function 49H), and Resize Memory Block (Int 21H Function 4AH). However, the Int 31H functions listed above can be called from protected mode, automatically create and destroy descriptors as necessary so that the DOS memory can be accessed conveniently from protected mode, and do not require explicit use of the DPMI translation services.

The DOS memory management services exist so that protected mode applications can allocate and free memory that is directly addressable by real mode applications, terminate-and-stay-resident utilities, the ROM BIOS, DOS, and DOS device drivers. Such memory is typically used to communicate with real mode software that is ignorant of extended memory and/or is not directly supported by DPMI. For example, an application might invoke the ROM BIOS video driver's palette programming function by allocating a DOS memory block, copying the palette color table from extended memory into the DOS memory block, and finally passing the address of the DOS memory block to the ROM BIOS by issuing the appropriate software interrupt via the DPMI's translation function.

Debug Support Services

DPMI also supports protected mode debuggers by providing access to the CPU's hardware debugging facilities:

0B00H	Set Debug Watchpoint
0B01H	Clear Debug Watchpoint
0B02H	Get State of Debug Watchpoint
0B03H	Reset Debug Watchpoint

These functions are necessary because the DPMI client is not necessarily running at the highest privilege level, and because multitasking hosts may need to maintain the debug registers on a per-client basis.

Miscellaneous Services

DPMI provides applications with information about host services with three functions:

0400H	Get DPMI Version
0401H	Get DPMI Capabilities
0A00H	Get Vendor-Specific API Entry Point

Two DPMI services support the creation of protected mode resident service providers (terminate-and-stay-resident utilities and drivers):

0C00H	Install Resident Service Provider Callback
0C01H	Terminate and Stay Resident

The next four functions support applications and device drivers that must directly access memory-mapped peripheral devices:

0508H	Map Device in Memory Block (Optional)
0509H	Map Conventional Memory in Memory Block (Optional)
0800H	Physical Address Mapping
0801H	Free Physical Address Mapping

Functions 0508H and 0509H are optional, and when they are implemented are only available on 32-bit hosts. Applications which require Functions 0508H or 0509H to execute are not DPMI-compliant. Functions 0800H and 0801H are available on all 16-bit and 32-bit hosts.

DPMI supports two functions related to numeric coprocessors:

0E00H	Get Coprocessor Status
0E01H	Set Coprocessor Emulation

These two functions allow protected mode applications to determine whether a numeric coprocessor is present, and to inspect or set the coprocessor-present and floating point emulation bits which are maintained by the DPMI host on a per-client basis. Using these calls, a DPMI client can install private floating point emulation even in an environment where the host provides emulation and/or there is a real floating point coprocessor present.

Chapter 4. DPMI Client Implementation Notes

Programs that use DPMI services are called DPMI clients. Generally, DPMI clients fall into one of two categories:

- DOS Extenders which use DPMI services as building blocks for a more extensive interface that is exported to application programs running under their control;
- Application programs that call DPMI directly.

In the near term, most client programs will need to be able to run in several different environments, each providing a different interface and range of services. It is recommended that clients test for the existence of such environments in the following order:

- DOS Protected Mode Interface (DPMI)-compatible host;
- Virtual Control Program Interface (VCPI)-compatible server;
- eXtended Memory Specification (XMS)-compatible driver;
- *Top-down memory allocation* (see Appendix A: Glossary) or bottom-up (VDISK-compatible) memory allocation.

Figure 2 on page 21 illustrates a typical DPMI client configuration, consisting of a DOS Extender and a protected-mode application. The application code relies on the DOS Extender functions and APIs. The DOS Extender contains separate modules for each possible environment, and code to implement those services that are lacking in a particular 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.

Figure 2. An example of a DPMI client consisting of a DOS Extender and a protected-mode application. The client should be able to run in the presence of DPMI, VCPI, or XMS environments or in the absence of all three.

```
+----------------------------------------------------------+
|                                                          |
|  +----------------------------------------------------+  |
|  |                                                    |  |
|  |                  Application Code                  |  |
|  |                                                    |  |
|  +----------------------------------------------------+  |
|                                                          |
|  +----------------------------------------------------+  |
|  |           Extender Base (including APIs)           |  |
|  | -------------------------------------------------- |  |
|  |    DPMI                                            |  |
|  |   client                                           |  |
|  +------------+                                       |  |
|               |    VCPI                               |  |
|               |   client                              |  |
|               +------------+                          |  |
|                            |    XMS                   |  |
|                            |   client                 |  |
|                            +------------+             |  |
|                                         | Top-down or |  |
|                                         |  bottom-up  |  |
|                                         |   client    |  |
|                                         +-------------+  |
|                                                          |
+----------------------------------------------------------+
  +------------+
  |            |
  |            +------------+
  |    DPMI    |            |
  |    host    |            +------------+
  |            |    VCPI    |            |
  |            |   server   |    XMS     +--------------+
  |            |            |   driver   |  Top-down    |
  |            |            |            |  (Int 15H)   |
  |            |            |            | or bottom-up |
  |            |            |            |   (VDISK)    |
  +------------+------------+------------+--------------+

  +-----------------------------------------------------+
  |                                                     |
  |             Operating System (e.g. DOS)             |
  |                                                     |
  +-----------------------------------------------------+
```

Client Initialization

DPMI clients are loaded in real mode. In order to enter protected mode, the client must first call Int 2FH Function 1687H (see page 53) to test for the presence of a DPMI host and obtain the address of the real-to-protected mode switch entry point. Function 1687H also returns information about the DPMI host's capabilities and the size of a private data area which will be used by the host to hold client-specific data structures. After allocating the required private data area via the normal real mode memory allocation interface (DOS Int 21H Function 48H), the client makes a `FAR CALL` to the mode switch entry point with the following parameters:

AX = flags

Bit	*Significance*
0	0 = 16-bit application 1 = 32-bit application
1-15	reserved, should be zero

ES = real mode segment of DPMI host private data area (must be at least as large as the size returned in SI by Int 2FH Function 1687H; ES is ignored if the size was zero)

If the Carry flag is set upon return, the mode switch was unsuccessful, the client is still running in real mode, and register AX contains one of the following error codes:

8011H descriptor unavailable (cannot allocate descriptors for CS, DS, ES, SS, PSP, and environment pointer)
8021H invalid value (32-bit program specified but not supported)

If the mode switch was successful, the mode switch routine returns to the caller in protected mode with the Carry flag clear and the following results:

CS = 16-bit selector with base of real mode CS and a 64 KB limit
SS = selector with base of real mode SS and a 64 KB limit
DS = selector with base of real mode DS and a 64 KB limit
ES = selector to program's PSP with a 100H byte limit
FS = 0 (if running on an 80386 or later)
GS = 0 (if running on an 80386 or later)

All other registers are preserved, except that for 32-bit clients the high word of ESP will be forced to zero. 32-bit clients will initially run with a 16-bit code segment, but all Int 31H calls will still require 48-bit pointers, and the stack and data descriptors will be 32-bit (the Big bit will be set in the descriptors). Note that if DS=SS at the time of the mode switch call, only one descriptor may be allocated, and the same selector may be returned in DS and SS. The client is allowed to modify or free the CS, DS, and SS descriptors allocated by the mode switch routine.

The environment pointer in the client program's PSP is automatically converted to a selector during the mode switch. If the client wishes to free the memory occupied by the environment, it should do so *before* entering protected mode and zero the word at PSP:2CH (segment address of the environment). The 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 client should *not* modify or free the PSP or environment descriptors supplied by the DPMI host.

Example: The following code illustrates how a DPMI client would obtain the address of the mode switch entry point, allocate the DPMI host private data area, and enter protected mode.

```
modesw  dd      0                       ; far pointer to DPMI host's
                                        ; mode switch entry point
        .
        .
        .
        mov     ax,1687h                ; get address of DPMI host's
        int     2fh                     ; mode switch entry point
        or      ax,ax                   ; exit if no DPMI host
        jnz     error
        mov     word ptr modesw,di      ; save far pointer to host's
        mov     word ptr modesw+2,es    ; mode switch entry point
        or      si,si                   ; check private data area size
        jz      @@1                     ; jump if no private data area

        mov     bx,si                   ; allocate DPMI private data
        mov     ah,48h                  ; area below 1 MB boundary
        int     21h                     ; transfer to DOS
        jc      error                   ; jump, allocation failed
        mov     es,ax                   ; let ES=segment of data area

@@1:    mov     ax,0                    ; bit 0=0 indicates 16-bit app
        call    modesw                  ; switch to protected mode
        jc      error                   ; jump if mode switch failed
                                        ; else we're in prot. mode now
        .
        .
        .
```

Client Termination

A protected mode client terminates by executing an Int 21H in protected mode, passing the value 4CH in register AH and a return code in register AL. (This mimics the Int 21H Function 4CH termination used by DOS applications in real mode.) The client has the following responsibilities before termination:

- real mode memory that was unlocked for paging with Int 31H Function 0602H must be relocked with Int 31H Function 0603H;

- interrupts hooked by the client for real mode with Int 31H Function 0201H must be released by restoring the address of the original owner of the interrupt.

- protected mode handlers for real mode exceptions installed with Int 31H Function 0213H should be cleaned up if possible.

When the DPMI host detects an Int 21H Function 4CH termination request, it takes the following actions (the detailed comparison of DPMI version 0.9 and version 1.0 host termination handling is in Appendix C, page 158):

- any extended memory blocks that were previously allocated with Function 0501H or 0504H are unlocked and freed (this is the only cleanup action required by a DPMI V 0.9 host);

- the client's local descriptor table (LDT) is freed in its entirety by DPMI 1.0 hosts. (A DPMI version 0.9 client should clean up its own segment descriptors before its termination since some DPMI version 0.9 hosts may not free the terminating client's segment descriptors if the client is not the primary client);
- physical address mappings created with Int 31H Function 0800H are freed;
- mappings created with Int 31H Functions 0508H or 0509H are destroyed;
- the client's interrupt descriptor table (IDT) is freed in its entirety and any client exception handlers installed through Functions 0203H, 0212H and 0213H are deregistered;
- any real mode regions that were unlocked with Function 0602H are relocked;
- any real mode callbacks that were allocated with Function 0303H are deallocated;
- the client's shared memory block allocations and serializations are freed;
- any debug watchpoints that were set with Function 0B00H are cleared;
- the coprocessor state (if any) is restored to the default.

After the DPMI host performs the cleanup activities listed above, it will switch to real mode and re-issue the Int 21H Function 4CH interrupt, passing the return code from the DPMI client down to DOS. DOS will then terminate the client as a real mode process by releasing its DOS memory blocks (whether allocated by real mode Int 21H Function 48H or by DPMI Function 0100H), flushing file buffers, closing file and device handles, and so on.

Clients should only terminate from their main thread of execution, and should not issue the protected mode Int 21H Function 4CH from within a hardware interrupt handler, exception handler, or real mode callback. Client may, however, terminate from within a protected mode routine that has been entered via the DPMI raw mode switch service. Clients which wish to terminate-and-stay-resident to provide services to protected mode clients should use DPMI Function 0C01H rather than Int 21H Function 31H.

Note: Although the DPMI host monitors for Int 21H Function 4CH in protected mode, it ignores all other Int 21H Functions. DOS Extenders typically install an interrupt 21H handler of their own in order to trap and service DOS function requests by a protected mode application; thus, the DOS Extender's Int 21H handler will always see the Function 4CH termination request first. The DOS Extender should perform any cleanup activities of its own and then pass the termination request to the DPMI host by chaining to the original owner of the protected mode Int 21H vector.

Stacks and Mode Switching

At one point in its execution or another, every DPMI client runs on four different stacks: an application stack, a locked protected mode stack, a real mode stack, and a DPMI host stack. It is important to understand how the host maintains these stacks to fully understand the protected mode environment.

The *application stack* is the primary stack that the DPMI client executes on. It is initially the real mode stack that the client was on when it switched into protected mode, although nothing prevents the client from switching protected mode stacks at any point after the initial mode switch. The application stack can be unlocked if desired. Software interrupts executed in protected mode are reflected on this stack.

The *locked protected mode stack* is provided by the DPMI host. The host automatically switches to this stack during servicing of hardware interrupts, software interrupts 1CH, 23H, and 24H, all exceptions, and during the execution of real mode callbacks. Subsequent nested interrupts or calls will not cause a stack switch. If the client switches off this stack, the new stack must also be locked and will become the protected mode stack until it switches back. When the interrupt or call returns, the host switches back to the original protected mode stack. Note that the host must provide a minimum of one 4 KB locked stack, and that software interrupts other than 1CH, 23H, and 24H do *not* use this stack. (Refer Appendix D for descriptor usage rule of locked stack.)

The *real mode stack* is also provided by the DPMI host, and is usually located in the DPMI host data area allocated by the client prior to its initial switch into protected mode. The real mode stack is at least 200H bytes in size and is always located in locked memory. Interrupts that are reflected to real mode, as well as calls to real mode interrupt handlers or procedures via Int 31H Functions 0300H, 0301H, or 0302H, will use this stack.

A *DPMI host stack* is only accessible to the DPMI host; it is used by the host to handle interrupts and exceptions that occur while the host is executing on behalf of the client. The DPMI host stack may also be used to contain state information about the client. For example, when the client requests a mode switch, the original SS:(E)SP of the protected mode program can be saved on the host stack while the DPMI host switches onto the locked protected mode stack.

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

- Execute the default interrupt reflection handler (all interrupts other than Int 31H and Int 21H Function 4CH are initialized by the DPMI host to point to a handler that reflects the interrupt to real mode);

- Use the DPMI translation services (Int 31H Functions 0300H, 0301H, and 0302H) to call a real mode interrupt handler or procedure;

- Allocate a real mode callback address with Int 31H Function 0303H; when a real mode program transfer control to the callback address, the DPMI host will switch the CPU from real mode into protected mode;

- Use the DPMI raw mode switch functions, whose addresses are obtained with Int 31H Function 0306H.

All of these mode switches except for the raw mode switches may save some information on the DPMI host's stack. This means that clients should not terminate within nested mode switches unless they are using the raw mode switching services. However, even clients that use raw mode switches should not attempt to terminate from a hardware interrupt or exception handler or real mode callback because the DPMI host performs automatic mode and stack switching during these events.

Clients that use the raw mode switch services and perform nested mode switches must use the DPMI state save/restore functions (whose addresses may be obtained with Int 31H Function 0305H), causing the host to maintain information on the "other" mode's current state. This information includes the CS:(E)IP, SS:(E)SP, and other segment register contents; values that the client has no way to access directly. For example, during the service of a hardware interrupt that occurs in real mode, the DPMI host may preserve the real mode CS:(E)IP, SS:(E)SP, and segment registers on the host stack. If the client subsequently calls the raw mode switch function without calling the state save function first, it will inadvertently overwrite the real mode information already pushed on the host stack, causing a return to the wrong address when the handler finally executes the IRET.

Example: This example illustrates code that saves the state of the real mode registers using the DPMI save/restore function, switches to real mode using the raw mode switch service, issues a DOS call to open a file, switches back to protected mode using the raw mode switch service, and restores the state of the real mode registers using the save/restore function. The protected mode registers are saved by pushing them on the stack in the usual fashion. The example is intended only to show the logical sequence of execution; in a real program, the real mode and protection mode variables and functions would likely reside in separate segments.

```
savsiz      dw    0                  ; size of state information
realsrs     dd    0                  ; far pointer to real mode
                                     ; save/restore state entry point
protsrs     dd    0                  ; far pointer to protected mode
                                     ; save/restore state entry point
realrms     dd    0                  ; far pointer to real mode
                                     ; raw mode switch entry point
protrms     dd    0                  ; far pointer to protected mode
                                     ; raw mode switch entry point

protds      dw    0                  ; placeholder for protected mode DS
protip      dw    0                  ; placeholder for protected mode IP
protcs      dw    0                  ; placeholder for protected mode CS
protsp      dw    0                  ; placeholder for protected mode SP
protss      dw    0                  ; placeholder for protected mode SS

realsp      dw    0                  ; placeholder for real mode SP
realss      dw    0                  ; placeholder for real mode SS

            .
            .
            .

                                     ; this code is executed during
                                     ; application initialization

      mov   ax,305h                  ; get addresses of DPMI host's
      int   31h                      ; state save/restore entry points
      mov   savsiz,ax                ; save state info buffer size
      mov   word ptr realsrs,cx      ; BX:CX = state save/restore
      mov   word ptr realsrs+2,bx    ; entry point for real mode
      mov   word ptr protsrs,di      ; SI:DI = state save/restore
      mov   word ptr protsrs+2,si    ; entry point for protected mode

      mov   ax,306h                  ; get address of DPMI host's
      int   31h                      ; raw mode switch entry points
      mov   savsiz,ax                ; save state info buffer size
      mov   word ptr realrms,cx      ; BX:CX = raw mode switch
      mov   word ptr realrms+2,bx    ; entry point for real mode
      mov   word ptr protrms,di      ; SI:DI = raw mode switch
      mov   word ptr protrms+2,si    ; entry point for protected mode
```

```
                                    ; must also initialize the
                                    ; sp and realss variables
      .
      .
      .

                                    ; this code is executed during
                                    ; program execution

callopenfile proc
      pusha                         ; save protected mode registers
      push  es

      sub   sp,savsiz               ; allocate space on current stack
      mov   di,sp                   ; to save real mode state
      mov   ax,ss                   ; let ES:DI = address of buffer
      mov   es,ax                   ; to receive state information
      xor   al,al                   ; AL=0 for save state request
      call  protsrs                 ; call state save/restore routine

      mov   protds,ds               ; save current DS for switch back
      mov   protss,ss               ; save current SS
      mov   protsp,sp               ; save current SP
      mov   protip,offset returnfromreal ; save return IP
      mov   protcs,cs               ; save return CS

      mov   ax,seg filename         ; load real mode DS
      mov   dx,realss               ; load real mode SS
      mov   bx,realsp               ; load real mode SP
      mov   si,seg openfile         ; load real mode CS
      mov   di,offset openfile      ; load real mode IP

      jmp   protrms                 ; Go to openfile

returnfromreal:
      mov   ax,ss                   ; let ES:DI = address of buffer
      mov   es,ax                   ; holding state information
      mov   di,sp
      mov   al,1                    ; AL=1 to restore state
      call  protsrs                 ; call state restore routine
      add   sp,savsiz               ; discard state info buffer

      pop   es
      popa                          ; restore protected mode registers
      ret
callopenfile endp

      .
      .
      .

                                    ; this code is executed in real mode

openfile proc
      mov   dx,offset filename
      mov   ah,3dh                  ; issue Open File DOS call
      int   21h
      jc    openerr                 ; check for error (not shown here)
      mov   filehandle,bx           ; save file handle
      mov   ax,protds               ; load protected mode DS
      mov   dx,protss               ; load protected mode SS
      mov   bx,protsp               ; load protected mode SP
      mov   si,protcs               ; load protected mode CS
      mov   di,protip               ; load protected mode IP
      jmp   realrms

openfile endp
```

Handling Interrupts

When a DPMI client switches into protected mode, a unique interrupt descriptor table (IDT) is created for the client by the DPMI host. Initially,all software interrupts (except for Int 31H, Int 2FH and Int 21H Function 4CH) or external hardware interrupts are directed to a handler that simply reflects the interrupt to real mode; i.e. the DPMI host's default handler simply switches the CPU into real mode and re-issues the interrupt, so that it can be serviced by the original real mode owner of the interrupt. The contents of the general registers and flags are passed to the real mode handler and the modified registers and flags are returned to the protected mode handler. Segment registers and the stack pointer are not passed between modes; the contents of the segment registers after the switch to real mode are undefined, and the DPMI host automatically supplies a valid real mode stack.

DPMI clients can install their own distinct real mode or protected mode handlers for software and external hardware interrupts with Functions 0201H and 0205H respectively. If a protected mode handler is installed, it is called instead of any real mode handler or the DPMI host's default handler. Just as in real mode, the protected mode handler can either service the interrupt and terminate with an `IRET`, or transfer to the next handler in the chain by executing a `PUSHF/CALL` or a `FAR JMP`. The final handler in the protected mode handler chain (the DPMI host's default handler) will reflect the interrupt to real mode.

Virtual Interrupts

Under many DPMI hosts, interrupts will always remain enabled in protected mode (i.e. the interrupt flag will be set at all times) to allow preemptive multitasking. Such hosts will maintain a virtual interrupt state for each virtual machine, trapping the execution of instructions that ordinarily affect the hardware interrupt flag and adjusting the client's virtual interrupt flag accordingly. When the virtual interrupt flag is cleared by the client's execution of `CLI` or call to DPMI function Int 31H 0900H, the program will not receive any hardware interrupts until it sets the flag again with `STI` or calls Function 0901H. DPMI clients should not use the `PUSHF` instruction to examine their interrupt status. This is because `PUSHF` pushes the real processor flags onto the stack, which do not reflect the state of the client's virtual interrupt flag. Similarly, clients cannot use `IRET(D)` or `POPF` to alter the interrupt flag, because these instructions access the physical interrupt flag and are ignored by the CPU due to the client's privilege level.

Example: The following source code demonstrates how a client would disable virtual interrupts prior to entry to an interrupt-critical section of code, then restore the virtual interrupt flag to its previous state at the end of the critical section:

```
      mov   ax,0900h          ; get previous virtual interrupt
      int   31h               ; flag and disable interrupts
      push  ax                ; save value 0900H or 0901H

      .
      .                       ; interrupt-critical code goes here
      .

      pop   ax
      int   31h               ; restore previous interrupt flag
```

If the client already knows (or does not care about) the previous state of the virtual interrupt flag, it can use `CLI` and `STI` instead of DPMI functions 0900H and 0901H. The programmer should assume that the execution of either of these instructions will be slow.

Hardware Interrupts

The programmable interrupt controllers are mapped by the DPMI host to the system's default interrupt assignments. On an IBM AT-compatible system, for example, the master interrupt controller (IRQ0 through IRQ7) is programmed to use a base interrupt level of 8 and the slave controller (IRQ8 through IRQ15) uses a base interrupt level of 70H.

All of the code and data that may be touched by hardware interrupt handlers must reside in locked memory to avoid page faults at interrupt time. The handler will always be called on a locked stack. As in real mode, hardware interrupt handlers are called with virtual interrupts disabled and the trace flag reset. In systems where the CPU's interrupt flag is virtualized, `IRET` may not restore the interrupt flag. Therefore, clients should execute a `STI` before executing `IRET` or else interrupts will remain disabled.

Protected mode hardware interrupt handlers that call a real mode routine must either ensure that the real mode code will not modify segment registers or use the DPMI state save/restore services (see page 94). However, any interrupt handler that executes completely in protected mode, or uses the translation services (Int 31H Functions 0300H, 0301H, or 0302H), does not need to save the real mode register state.

Personal computers with two programmable interrupt controllers usually have a BIOS that redirects one of the interrupts from the slave controller into the range of the master controller for compatibility with older, 8086/88-based systems. For example, devices jumpered for IRQ2 on PC/AT-compatible computers actually interrupt on IRQ 9 (Int 71H), but the BIOS on these systems converts Int 71H to Int 0AH yet sends the EOI command (appropriately) to the slave controller. A protected mode client that needs access to the redirected interrupt might use a variation on one of the following techniques:

- Install only a real mode handler for the target interrupt, taking advantage of the built-in redirection. This method is robust on systems where other software has reprogrammed the interrupt controllers, or where the slave interrupt controller may be absent.

- Install both real mode and protected mode handlers for the target interrupt. In such cases, the program must send the EOI command to both the slave and master interrupt controllers since the BIOS is never called. This method is more efficient in that there are not any unnecessary switches to real mode.

Software Interrupts

Ordinarily, a handler installed with DPMI Function 0205H will only service software interrupts that are executed in protected mode; real mode software interrupts are passed to handlers installed with DOS Int 21H Function 25H issued from real mode, DPMI Int 31H Function 0201H, or by direct manipulation of the interrupt vector table at real mode address 0000:0000. However, there are three real mode software interrupts that a DPMI host will always reflect to a protected mode handler, if one is installed:

Int 1CH	ROM BIOS timer tick interrupt
Int 23H	DOS Ctrl+C interrupt
Int 24H	DOS critical error interrupt

Clients should never terminate during the processing of interrupts that were reflected from real mode. Such a termination might prevent the DPMI host from cleaning up the client's resources properly.

Protected mode handlers for software interrupts 0-7 are called with virtual interrupts disabled and trace flag reset, and these handlers should return with interrupts enabled. All other software interrupts do not modify the interrupt flag state.

Handling CPU Exceptions

Exceptions are interrupts that are generated internally by the CPU when certain conditions are detected during the execution of a program. Examples of such conditions are: use of an invalid selector, use of a selector for which the program has insufficient privileges, use of an offset outside the limits of a segment, execution of an illegal opcode, or division by zero. The DPMI host distinguishes between exceptions and external hardware interrupts or software interrupts.

Handlers for exceptions can only be installed with Int 31H Functions 0203H, 0212H, or 0213H. If the client does not install a handler for a particular exception, or installs a handler but chains to the host's default handler, the host reflects the exception as a real mode interrupt for exceptions 0, 1, 2, 3, 4, 5, and 7. The default behavior of exceptions 6 and 8-1FH is to terminate the client (some hosts may decide that they have to terminate the VM because the fault came from real mode code or it is in a non-terminatable state).

Function 0203H was defined in DPMI version 0.9 and continues to be supported in DPMI version 1.0 for compatibility reasons. Exception handlers installed with Function 0203H are only called for exceptions that occur in protected mode. All exceptions are examined by the DPMI host. The host processes any exception that it is responsible for, such as page fault for virtual memory management. These *transparent exceptions* are never passed to the client exception handlers. All other exceptions become *visible exceptions* to a client and are passed to the client exception handler (if any) from the DPMI host. The client exception handlers must return with a FAR RETURN, with interrupts disabled on a locked stack, and with the SS, (E)SP, CS, and (E)IP registers at the point of exception pushed on the stack. All other registers are unchanged from their contents at the point of exception. The stack frame for 16-bit handlers installed with Function 0203H has the following format:

```
15              0
+---------------+
|       SS      |
|---------------| 0EH
|       SP      |
|---------------| 0CH
|     Flags     |
|---------------| 0AH
|       CS      |
|---------------| 08H
|       IP      |
|---------------| 06H
|   Error Code  |
|---------------| 04H
|   Return CS   |
|---------------| 02H
|   Return IP   |
+---------------+ 00H <-- SS:SP
```

The stack frame for 32-bit handlers installed with Function 0203H has the following format:

```
31             15                0
+--------------+-----------------+
|    Reserved   |     SS         |
|--------------------------------| 1CH
|               ESP              |
|--------------------------------| 18H
|               EFLAGS           |
|--------------------------------| 14H
|    Reserved   |     CS         |
|--------------------------------| 10H
|               EIP              |
|--------------------------------| 0CH
|           Error Code           |
|--------------------------------| 08H
|              |   Return CS     |
|--------------------------------| 04H
|           Return EIP           |
+--------------------------------+ 00H <-- SS:ESP
```

The error code in the stack frame is only valid for the following exceptions:

08H	Double fault
0AH	Invalid TSS
0BH	Segment not present
0CH	Stack fault
0DH	General protection fault

0EH Page fault

In the case of other exceptions and faults, the value of the error code is undefined. The fields marked `Return CS`, `Return (E)IP`, and `Reserved` in the stack frame should not be modified, but anything else in the stack frame can be altered by the client before returning from the exception handler.

The exception handler must preserve and restore all registers, and must either jump to the next handler in the chain or terminate with a RETF (far return) instruction. In the latter case, the original SS:(E)SP, CS:(E)IP and flags on the stack, including the interrupt flag, will be restored. The exception handler can arrange to transfer control to a more general error-handling routine within the application by modifying the CS:(E)IP that is stored in the stack frame above the Return CS:(E)IP.

DPMI version 1.0 supports an expanded stack frame for exception handlers, and the ability to install separate handlers for exceptions which occur in real mode and in protected mode with Functions 0212H and 0213H. The expanded frame is defined on the stack *above* the frame previously described for handlers installed with Function 0203H. This allows DPMI 0.9-compatible handlers and DPMI 1.0-compatible handlers to coexist in the same handler chain; old handlers will be oblivious to the additional information available beyond the old stack frame, while new handlers can ignore the old frame and use only the expanded frame higher up on the stack.

The format of the expanded stack frame for both 16-bit and 32-bit handlers installed with Functions 0212H and 0213H is as follows:

```
31                        15                          0
+--------------------------+--------------------------+ 58H
|                         PTE                         |
|-----------------------------------------------------| 54H
|                         CR2                         |
|-----------------------------------------------------| 50H
|        Reserved          |            GS            |
|-----------------------------------------------------| 4CH
|        Reserved          |            FS            |
|-----------------------------------------------------| 48H
|        Reserved          |            DS            |
|-----------------------------------------------------| 44H
|        Reserved          |            ES            |
|-----------------------------------------------------| 40H
|        Reserved          |            SS            |
|-----------------------------------------------------| 3CH
|                         ESP                         |
|-----------------------------------------------------| 38H
|                        EFLAGS                       |
|-----------------------------------------------------| 34H
|Exception information bits|            CS            |
|-----------------------------------------------------| 30H
|                         EIP                         |
|-----------------------------------------------------| 2CH
|                     Error code                      |
|-----------------------------------------------------| 28H
|        Reserved          | Return CS (32-bit) or 0  |
|-----------------------------------------------------| 24H
|        Return EIP (32-bit) or CS:IP (16-bit)        |
+--------------------------+--------------------------+ 20H+SS:(E)SP
```

A 32-bit stack frame image is always presented, even for 16-bit handlers, and the offset from SS:(E)SP to the expanded stack frame is always 20H (32) regardless of the handler type. The

DS, ES, FS, and GS registers are saved for both real and protected mode. The client can inspect the VM bit in EFLAGS to determine the mode at the point of exception. The CS field at SS:(E)SP+24H is zero if the handler is running in 16-bit protected mode.

The exception information bits at SS:(E)SP+32H have the following meanings:

Bit	*Significance*	
0	0 =	exception occurred in the client
	1 =	exception occurred in the host (most likely due to page fault or invalid selector passed to host in an Int 31H call)
1	0 =	exception can be retried
	1 =	exception cannot be retried, handler should perform whatever cleanup is possible
2	0 =	host exception should be retried (invalid selector or page causing fault corrected by exception handler, this is the default)
	1 =	host exception is being redirected somewhere other than a retry address
3-15	reserved	

Bits 0 through 2 of the exception information bits are relevant on hosts which support the Exceptions Restartability capability (see Int 31H Function 0401H). Bits 0 and 1 of the exception information bits are supplied to the client by the host. The default state of bit 2 as set by the host is zero, and the client may set the bit to 1 before returning from the exception handler.

Bits 0-14 of the error code at SS:(E)SP+28H are the "virtual" DR6 on debug (Int 1) exceptions, and correspond to debug breakpoints 0-14. In other words, if bits 0 and 2 are set in the error code field on an Int 1 exception, then debug watchpoints 0 and 2 have fired. The handle returned by the Set Debug Watchpoint (Function 0B00H) corresponds to the bit number in the virtual DR6. Bit 15 of the virtual DR6 is set (1) if the Int 1 is due to the trap flag. Breakpoints may be virtualized, and there is no guarantee of correspondence with the actual hardware. The provision for up to 15 breakpoints is made for future CPUs or external debugging hardware (80386 and 80486 CPUs support only four hardware breakpoints).

The PTE and CR2 fields of the expanded stack frame at SS:(E)SP+50H and 54H respectively are only valid for page faults (Int 0EH). Bits 0-7 of the PTE (page table entry) field are from the actual PTE and may be virtualized by the host; the remaining bits of the PTE field are undefined. The CR2 field contains the linear address that caused the fault.

Exception handlers installed with Functions 0212H and 0213H may terminate in any of the following three ways:

- RETF from the old-style stack frame (only modifications to the old-style stack frame will be recognized and a client may not use this type of return for real-mode exceptions);

- Discard the old-style stack frame by adding 20H (32) to (E)SP, then RETF from the new-style (expanded) stack frame;

- FAR JMP to previous owner of the exception (the previous owner should never be CALLed).

The fields at offsets 2CH through 4FH in the expanded stack frame may be modified by an exception handler. Note that the handler should only modify the values in the particular frame (SS:(E)SP+0 or SS:(E)SP+20H) that it will use for the RETF. Altered values in the other frame are ignored by the DPMI host. Real mode exceptions do not have valid data in the old-style frame. A real mode exception handler must discard the old-style stack frame if it returns.

Example: The following code illustrates how a client would install its own exception handler for general protection (GP) faults. The actual handler does nothing more than reach into the stack frame and alter the return address, so that control within the application restarts at a different point after the exception handler exits.

```
prevgp  dd      0                     ; address of previous
                                      ;   GP fault handler

          .
          .                           ; this code is executed during
          .                           ; application initialization...

      mov     ax,0210h                ; get address of previous
      mov     bl,13                   ; owner of GP fault vector
      int     31h
      mov     word ptr prevgp,dx      ; save as far pointer
      mov     word ptr prevgp,cx

      mov     ax,0212h                ; install our GP fault handler
      mov     bl,13
      mov     cx,cs                   ; CX:DX = handler address
      mov     dx,offset _TEXT:gpfisr
      int     31h
      jc      init9                   ; jump, couldn't install
        .                             ; continue with initialization
        .
        .

gpfisr  proc    far                   ; this is the actual exception
                                      ; handler for GP faults

      add     sp,20h                  ; discard "old" stack frame
      push    bp                      ; point CS:IP in stack frame to
      mov     bp,sp                   ; GP fault error message routine
      mov     word ptr [bp+0eh],offset _TEXT:gpferr
      mov     word ptr [bp+12h],cs
      pop     bp
      ret                             ; now return from exception

gpfisr  endp


gpferr  proc    near                  ; this routine executes after
                                      ; GPFISR returns to DPMI host

      mov     ax,4c01h                ; terminate DPMI client with
      int     21h                     ; nonzero return code

gpferr  endp
```

Using Real-Mode Callbacks

The DPMI real mode callback mechanism allows a DPMI protected mode client to be called as a subroutine by real mode programs in a transparent manner. That is, a real mode program can use a real mode callback to pass information to the DPMI client, or obtain services provided by the DPMI client, without necessarily being aware of protected mode or extended memory in any way. The callback mechanism can be thought of as the converse of DPMI Int 31H Functions 0300H, 0301H, and 0302H, which allow a DPMI client to pass information to a real mode program, or obtain services from a real mode program, in a manner that is similarly transparent to the real mode program.

In order to make a real mode callback available, the DPMI client must first call Int 31H Function 0303H with the selector and offset of the protected mode routine which will receive control when the callback is entered, and the selector and offset of a real mode register data structure (in the same format as used for Int 31H Functions 0300H, 0301H, and 0302H). Function 0303H will return a real mode address (segment and offset) for the callback entry point that can be passed to a real mode program via a software interrupt or far call (Int 31H Functions 0300H, 0301H, or 0302H), a DOS memory block, or any other convenient mechanism.

When the real mode program executes a `FAR CALL` to the real mode callback address supplied to it by the DPMI client, the DPMI host saves the contents of all real mode registers into the DPMI client's real mode register data structure, switches the CPU into protected mode, and enters the DPMI client's callback routine with the following conditions:

Interrupts disabled
CS:(E)IP = selector:offset specified in the original call to Int 31H Function 0303H
DS:(E)SI = selector:offset corresponding to real mode SS:SP
ES:(E)DI = selector:offset of real mode register data structure
SS:(E)SP = locked protected mode stack provided by DPMI host
All other registers undefined

The format of the real mode register data structure is as follows: (Note that the content of the 32H bytes data structure are undefined at the time of the original Int 31H Function 0303H call.)

Offset	*Length*	*Contents*
00H	4	DI or EDI
04H	4	SI or ESI
08H	4	BP or EBP
0CH	4	reserved, should be zero
10H	4	BX or EBX
14H	4	DX or EDX
18H	4	CX or ECX
1CH	4	AX or EAX
20H	2	CPU status flags
22H	2	ES
24H	2	DS
26H	2	FS
28H	2	GS
2AH	2	IP
2CH	2	CS
2EH	2	SP
30H	2	SS

The callback procedure can then extract its parameters from the real mode register data structure and/or copy parameters from the real mode stack to the protected mode stack. Recall that the segment register fields of the real mode register data structure contain segment or paragraph addresses that are not valid in protected mode. Far pointers passed in the real mode register data structure must be translated to virtual addresses before they can be used. The recommended procedure is for the DPMI client to allocate a selector for this purpose during its initialization, then use Int 31H Function 0007H within the callback procedure to set the segment base to 16 times the value found in the real mode segment register. The DPMI client should *not* use Int 31H Function 0002H for this purpose, because selectors allocated by Function 0002H can never be freed.

The callback procedure exits by executing an `IRET` with the address of the real mode register data structure in ES:(E)DI, passing information back to the real mode caller by modifying the contents of the real mode register data structure and/or manipulating the contents of the real mode stack. The callback procedure is responsible for setting the proper address for resumption of real mode execution into the real mode register data structure; typically, this is accomplished by extracting the return address from the real mode stack and placing it into the CS:IP fields of the real mode register data structure. After the `IRET`, the DPMI host switches the CPU back into real mode, loads all registers (including CS:IP) with the contents of the real mode register data structure, and finally returns control to the real mode program.

Since the real mode call structure and the selector used for the real mode SS are static, care must be taken when writing DPMI client callback procedures that may be reentered (for example, by a real mode program that services hardware interrupts). The simplest method of avoiding reentrancy is to leave interrupts disabled throughout the entire callback procedure. However, if the amount of code executed by the callback is large, the client may find it more desirable to copy the real mode register data structure into a dynamically allocated buffer and then re-enable interrupts and not use the incoming DS anymore. The real mode register data structure pointed to by ES:(E)DI upon return from the callback procedure is *not* required to be at the same address as the original real mode register data structure.

DPMI hosts must provide a minimum of 16 callback addresses per virtual machine. Real mode callbacks are a limited system resource. A DPMI client should always use Int 31H Function 0304H to free any callbacks that it is no longer using.

Example: The following code is a example 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 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

      assume  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
```

Using Shared Memory

Shared memory blocks can be used for inter-client communication or simply to hold tables or subroutine libraries that are needed by more than one client. Explicit use of shared blocks is necessary because each VM has its own linear address space, and thus cannot inspect or modify the memory owned by a client in another virtual machine. The basic strategy for use of a shared memory block is as follows:

- Allocate a shared memory block;
- Establish addressability for the shared block;
- Make a successful serialization request for the shared block (i.e. obtain ownership or right of access to the block);
- Access the code or data in the shared block;
- Free the serialization of the shared block;
- Deallocate the shared memory block.

Shared memory blocks are allocated with Int 31H Function 0D00H. The client passes the address of a data structure that specifies the ASCIIZ name and requested size for the shared block to the host; the host returns the block's handle, linear base address, and actual size in the same structure. The block's true size is determined by the first client to allocate the block, and the block will have the same linear address for all clients which allocate it.

After the shared block is allocated, the client must allocate one or more descriptors that will be used to address the block with Int 31H Function 0000H. Once descriptor(s) have been allocated and initialized to point to a shared memory block through separate LDT management calls, the client has the physical capability to read, write, or execute addresses within the block as allowed by the access rights/type byte. The client should synchronize with any other clients which might have addressability to the same block, to avoid race conditions or corruption of data. This synchronization is accomplished with Int 31H Function 0D02H (Serialize on Shared Memory) and Int 31H Function 0D03H (Free Serialization on Shared Memory). Serialization can be thought of as representing ownership or right of access to a shared memory block.

In essence, Int 31H Functions 0D02H and 0D03H treat the handle of a shared memory block as a semaphore. The client can request exclusive (read/write) or shared (read-only) serialization with Int 31H Function 0D02H, and the host will grant the serialization if no other client has already obtained a conflicting serialization on the same memory block. The client can then go ahead and manipulate the shared memory block, releasing the serialization with Int 31H Function 0D03H when it is finished using the block. If the Int 31H Function 0D02H serialization request fails, the client will either be suspended until the serialization is available, or the function will return with an error code, depending on the parameters supplied by the client.

The first paragraph (16 bytes) of the shared memory block (or the entire shared block, if smaller than 16 bytes) will always be initialized to zero on the first allocation and can be used by clients as an "area initialized" indicator. For example, a shared memory block might be used by a suite of cooperating client programs to hold a table of static data or a subroutine library. The first client to allocate the shared memory block can obtain exclusive ownership of the block with Int 31H Function 0D02H, load the necessary data or code into the block from disk, set the first 16 bytes of the block to a nonzero value, and finally release its ownership of the block with Int 31H Function 0D03H. Other clients that allocate the shared memory block can check the "area initialized" indicator and know that the desired code or data is already present in memory.

When the client has finished using the shared memory block, it should deallocate the shared block with Int 31H Function 0D01H. After the block is deallocated, the linear addresses within the block are no longer valid for the current client, and may cause an exception if accessed. However, the block is not actually destroyed until *all* clients which have allocated the block have also deallocated it.

Note that a client can make multiple (nested) allocation requests for the same shared memory block, and should assume that each allocation request will return a distinct handle. The shared block will remain physically accessible to the client until each of its handles to the block have been deallocated. Similarly, a client can make multiple serialization requests for the same block, and will retain "ownership" of the block until a corresponding number of deserialization requests have been issued. Lastly, allocation of zero-length shared memory blocks is explicitly allowed, so that clients can use the handles resulting from such allocations as pure semaphores.

Example: The following code illustrates how shared memory can be used to load code and data that can be used by more than one DPMI client. Note that no serialization calls are required if the memory is already initialized.

```
memreqstruc struc
length  dd      ?                       ; number of bytes requested
actual  dd      ?                       ; number of bytes allocated
handle  dd      ?                       ; handle for shared memory block
base    dd      ?                       ; linear address of shared block
nameptr dp      ?                       ; pointer to shared memory name
        dw      0                       ; reserved, must be zero
        dd      0                       ; reserved, must be zero
memreqstruc ends

memname db      'MYBLOCK',0
memreq  memreqstruc <>                  ; allocate request block

        .
        .
        .
        mov     word ptr memreq.length,2000h  ; set requested length
        mov     word ptr memreq.length+2,0    ; of shared block to 8 KB
                                        ; initialize nameptr
        mov     dword ptr memreq.nameptr,offset memname
        mov     word ptr memreq.nameptr+4, ds

        mov     di,ds                   ; ES:DI = address of shared
        mov     es,di                   ; memory request structure
        mov     di,offset memreq
        mov     ax,0d00h                ; DPMI fxn 0D00H = allocate
        int     31h                     ; shared memory block
        jc      error                   ; jump if allocation failed

        mov     cx,1                    ; allocate one LDT descriptor
        mov     ax,0                    ; using DPMI Function 0000H
        int     31h
        jc      error                   ; jump, no descriptor available

        mov     bx,ax                   ; let BX = new selector
        mov     dx,word ptr memreq.base       ; let CX:DX = linear base
        mov     cx,word ptr memreq.base+2     ; address of shared block
        mov     ax,0007h                ; set descriptor base address
        int     31h                     ; using DPMI Function 0007H
        jc      error                   ; jump, function failed

        mov     dx,word ptr memreq.actual     ; let CX:DX = length-1
        mov     cx,word ptr memreq.actual+2   ; of shared memory block
        sub     dx,1
        sbb     cx,0                    ; (BX still = selector)
        mov     ax,8                    ; set descriptor limit using
        int     31h                     ; DPMI Function 0008H
        jc      error                   ; jump, function failed

        mov     es,bx                   ; ES = selector for shared block
        mov     ax,es:[0]               ; is block already initialized?
        or      ax,ax
        jnz     @@1                     ; jump if it's initialized

                                        ; not initialized, get ownership
                                        ; of the shared memory block
        mov     di,word ptr memreq.handle     ; SI:DI = handle for
        mov     si,word ptr memreq.handle+2   ; shared memory block
        mov     dx,0                    ; exclusive + wait for ownership
        mov     ax,0d02h                ; DPMI Fxn 0D02H = serialize
        int     31h
        jc      error                   ; jump if serialization failed

        mov     ax,es:[0]               ; check again if someone else
        or      ax,ax                   ; already initialized block
        jnz     @@2                     ; jump if it's initialized

        .
        .                               ; load code into the shared
        .                               ; memory block here...
        .

@@2:                                    ; now release ownership of
                                        ; the shared memory block
        mov     di,word ptr memreq.handle     ; SI:DI = handle for
        mov     si,word ptr memreq.handle+2   ; shared memory block
        mov     dx,0                    ; serialization type = exclusive
        mov     ax,0d03h                ; DPMI Fxn 0D03H = release
        int     31h
        jc      error                   ; jump if serialization failed

@@1:                                    ; finished initializing the
                                        ; shared memory block
```

Writing Resident Service Providers

A DPMI client that provides resident protected mode services (also called a "protected mode TSR") must install itself using Int 31H Functions 0C00H and 0C01H (see the detailed definition of these functions on pages 139 through 141). The protected mode TSR first declares its intent to remain resident by calling Function 0C00H, providing the DPMI host with code and data descriptors and callback entry points for 16-bit and/or 32-bit protected mode. If the TSR does not wish to provide services in a particular mode, it provides a code descriptor for that mode containing all zero bytes. The protected mode TSR then terminates and stays resident by calling Int 31H Function 0C01H. Note that after this function call, the TSR's original addressing context is destroyed; its LDT and IDT no longer exist, although any extended memory blocks it owned at time of termination remain allocated.

Whenever another DPMI client *in the same virtual machine* loads or terminates, the DPMI host will inspect its list of protected mode TSRs. If a particular TSR has indicated that it can execute in the active client's mode, the DPMI host will automatically allocate two LDT descriptors in the active client's context, initialize them to the values specified in the protected mode TSR's original

Function 0C00H call, and enter the TSR via a FAR CALL at the offset appropriate to the current mode, passing the following values:

CS = executable selector which maps the same memory as the code descriptor specified in the Int 31H Function 0C00H data structure for the current mode (16-bit or 32-bit)
(E)IP = offset specified in the Int 31H Function 0C00H data structure for the current mode (16-bit or 32-bit)
DS = read/write data selector which maps the same memory as the data descriptor specified in the Int 31H Function 0C00H data structure for the current mode (16-bit or 32-bit)
ES = 0
FS = 0
GS = 0
AX = reason for callback: 0=DPMI client loading, 1=DPMI client terminating
BX = unique handle for the client within the virtual machine)

When a new DPMI client is loaded and executes the initial switch to protected mode, the appropriate callback procedure in the protected mode TSR will be entered by a FAR CALL with AX=0 *before* the DPMI host returns to the new program. The TSR may then hook interrupts, create descriptors, or allocate memory blocks in the new client's context prior to the client's access to such protected mode resources. For example, the initialization callback gives the TSR an opportunity to insert itself in the chain of handlers for any arbitrary interrupt or exception. The TSRs are invoked in the order of their installation but are removed in the reverse order. The TSR may also need to construct per-client data structures in its own memory, and can use the value supplied to it in BX as a "handle" for the client. The TSR must exit from the initialization callback with a RETF.

Similarly, when a DPMI client terminates using Int 21H Function 4CH or Int 31H Function 0C01H, the TSR's callback procedure will entered by a FAR CALL with AX=1 *before* the active client's LDT or IDT has been destroyed. The protected mode TSR can then perform any client termination responsibilities of which the client is unaware (e.g. unmapping of physical memory), release any protected mode resources which it has acquired on behalf of the client (e.g. ownership of shared memory), and deallocate any per-client data structures of its own. The termination callback must exit with a RETF and indicate an action to the DPMI host as follows:

Carry = clear to keep resident services in memory
or
Carry = set to remove resident services from memory

A resident service provider should only be removed from memory of the last client of the virtual machine they are servicing.

Int 31H Functions 0C00H and 0C01H should only be used by DPMI clients which intend to provide resident services to other protected mode clients. If the objective is only to provide resident services to real mode programs, the client should use the DPMI translation service Int 31H Function 0300H to invoke DOS's Int 21H Function 31H directly.

Chapter 5. DPMI Function Reference

The Int 2FH and Int 31H functions supported by DPMI version 1.0 hosts are described in this section. Each function entry has five parts:

- The function name, interrupt and function number, and the DPMI version where the function is first defined. All subsequent DPMI versions can be assumed to support the function as documented unless explicitly noted otherwise.
- A brief description of the function's purpose and usage.
- The parameters supplied by the DPMI client when it makes the function call.
- The results returned for the function by the DPMI host.
- Programmer's notes giving more detailed information about the function and/or describing special uses of the function.

Three tables at the beginning of this section list the Int 31H services sorted by functional group, function number, and name. The Int 2FH functions are not included in these tables. Programming examples and discussions of the use of the functions "in context" are found in the "Implementation Notes for DPMI Clients" section of this document.

DPMI Int 31H Functions Listed by Functional Group

Function Number	Function Name	DPMI 0.9	DPMI 1.0
LDT Management Services			
0000H	Allocate LDT Descriptor	•	•
0001H	Free LDT Descriptor	•	•
0002H	Map Real-Mode Segment to Descriptor	•	•
0003H	Get Selector Increment Value	•	•
0006H	Get Segment Base Address	•	•
0007H	Set Segment Base Address	•	•
0008H	Set Segment Limit	•	•
0009H	Set Descriptor Access Rights	•	•
000AH	Create Alias Descriptor	•	•
000BH	Get Descriptor	•	•
000CH	Set Descriptor	•	•
000DH	Allocate Specific LDT Descriptor	•	•
000EH	Get Multiple Descriptors		•
000FH	Set Multiple Descriptors		•
Extended Memory Management Services			
0500H	Get Free Memory Information	•	•
0501H	Allocate Memory Block	•	•
0502H	Free Memory Block	•	•
0503H	Resize Memory Block	•	•
0504H	Allocate Linear Memory Block		•
0505H	Resize Linear Memory Block		•
0506H	Get Page Attributes		•
0507H	Set Page Attributes		•
0508H	Map Device in Memory Block		•
0509H	Map Conventional Memory in Memory Block		•
050AH	Get Memory Block Size and Base		•
050BH	Get Memory Information		•
0800H	Physical Address Mapping	•	•
0801H	Free Physical Address Mapping		•
0D00H	Allocate Shared Memory		•
0D01H	Free Shared Memory		•
0D02H	Serialize on Shared Memory		•
0D03H	Free Serialization on Shared Memory		•
DOS Memory Management Services			
0100H	Allocate DOS Memory Block	•	•
0101H	Free DOS Memory Block	•	•
0102H	Resize DOS Memory Block	•	•

Interrupt Management Services

0200H	Get Real Mode Interrupt Vector	•	•
0201H	Set Real Mode Interrupt Vector	•	•
0202H	Get Processor Exception Handler Vector	•	•
0203H	Set Processor Exception Handler Vector	•	•
0204H	Get Protected Mode Interrupt Vector	•	•
0205H	Set Protected Mode Interrupt Vector	•	•
0210H	Get Extended Processor Exception Handler Vector in Protected Mode		•
0211H	Get Extended Processor Exception Handler Vector in Real Mode		•
0212H	Set Extended Processor Exception Handler Vector in Protected Mode		•
0213H	Set Extended Processor Exception Handler Vector in Real Mode		•
0900H	Get and Disable Virtual Interrupt State	•	•
0901H	Get and Enable Virtual Interrupt State	•	•
0902H	Get Virtual Interrupt State	•	•

Translation Services

0300H	Simulate Real Mode Interrupt	•	•
0301H	Call Real Mode Procedure with Far Return Frame	•	•
0302H	Call Real Mode Procedure with Interrupt Return Frame	•	•
0303H	Allocate Real Mode Callback Address	•	•
0304H	Free Real Mode Callback Address	•	•
0305H	Get State Save/Restore Addresses	•	•
0306H	Get Raw CPU Mode Switch Addresses	•	•

Page Management Services

0600H	Lock Linear Region	•	•
0601H	Unlock Linear Region	•	•
0602H	Mark Real Mode Region as Pageable	•	•
0603H	Relock Real Mode Region	•	•
0604H	Get Page Size	•	•
0702H	Mark Page as Demand Paging Candidate	•	•
0703H	Discard Page Contents	•	•

Debug Support Services

0B00H	Set Debug Watchpoint	•	•
0B01H	Clear Debug Watchpoint	•	•
0B02H	Get State of Debug Watchpoint	•	•
0B03H	Reset Debug Watchpoint	•	•

Miscellaneous Services

0400H	Get DPMI Version	•	•
0401H	Get DPMI Capabilities		•
0A00H	Get Vendor-Specific API Entry Point	•	•
0C00H	Install Resident Service Provider Callback		•
0C01H	Terminate and Stay Resident		•
0E00H	Get Coprocessor Status		•
0E01H	Set Coprocessor Emulation		•

Reserved Function Numbers

0004H	Reserved
0005H	Reserved
0700H	Reserved
0701H	Reserved

DPMI Int 31H Functions Listed Alphabetically

Function Name	**Function Number**	**DPMI 0.9**	**DPMI 1.0**
Allocate DOS Memory Block	0100H	•	•
Allocate LDT Descriptor	0000H	•	•
Allocate Linear Memory Block	0504H		•
Allocate Memory Block	0501H	•	•
Allocate Real Mode Callback Address	0303H	•	•
Allocate Shared Memory	0D00H		•
Allocate Specific LDT Descriptor	000DH	•	•
Call Real Mode Procedure with Far Return Frame	0301H	•	•
Call Real Mode Procedure with Interrupt Return Frame	0302H	•	•
Clear Debug Watchpoint	0B01H	•	•
Create Alias Descriptor	000AH	•	•
Discard Page Contents	0703H	•	•
Free DOS Memory Block	0101H	•	•
Free LDT Descriptor	0001H	•	•
Free Memory Block	0502H	•	•
Free Physical Address Mapping	0801H		•
Free Real Mode Callback Address	0304H	•	•
Free Serialization on Shared Memory	0D03H		•
Free Shared Memory	0D01H		•
Get and Disable Virtual Interrupt State	0900H	•	•
Get and Enable Virtual Interrupt State	0901H	•	•
Get Coprocessor Status	0E00H		•
Get Descriptor	000BH	•	•
Get DPMI Capabilities	0401H		•
Get DPMI Version	0400H	•	•
Get Extended Processor Exception Handler Vector in Protected Mode	0210H		•
Get Extended Processor Exception Handler Vector in Real Mode	0211H		•
Get Free Memory Information	0500H	•	•
Get Memory Block Size and Base	050AH		•
Get Memory Information	050BH		•
Get Multiple Descriptors	000EH		•
Get Page Attributes	0506H		•
Get Page Size	0604H	•	•
Get Processor Exception Handler Vector	0202H	•	•
Get Protected Mode Interrupt Vector	0204H	•	•
Get Raw CPU Mode Switch Addresses	0306H	•	•
Get Real Mode Interrupt Vector	0200H	•	•
Get Segment Base Address	0006H	•	•
Get Selector Increment Value	0003H	•	•
Get State of Debug Watchpoint	0B02H	•	•
Get State Save/Restore Addresses	0305H	•	•
Get Vendor-Specific API Entry Point	0A00H	•	•

Get Virtual Interrupt State	0902H	•	•
Install Resident Service Provider Callback	0C00H		•
Lock Linear Region	0600H	•	•
Map Conventional Memory in Memory Block	0509H		•
Map Device in Memory Block	0508H		•
Map Real-Mode Segment to Descriptor	0002H	•	•
Mark Page as Demand Paging Candidate	0702H	•	•
Mark Real Mode Region as Pageable	0602H	•	•
Physical Address Mapping	0800H	•	•
Relock Real Mode Region	0603H	•	•
Reserved	0004H		
Reserved	0005H		
Reserved	0700H		
Reserved	0701H		
Reset Debug Watchpoint	0B03H	•	•
Resize DOS Memory Block	0102H	•	•
Resize Linear Memory Block	0505H		•
Resize Memory Block	0503H	•	•
Serialize on Shared Memory	0D02H		•
Set Coprocessor Emulation	0E01H		•
Set Debug Watchpoint	0B00H	•	•
Set Descriptor	000CH	•	•
Set Descriptor Access Rights	0009H	•	•
Set Extended Processor Exception Handler Vector in Protected Mode	0212H		•
Set Extended Processor Exception Handler Vector in Real Mode	0213H		•
Set Multiple Descriptors	000FH		•
Set Page Attributes	0507H		•
Set Processor Exception Handler Vector	0203H	•	•
Set Protected Mode Interrupt Vector	0205H	•	•
Set Real Mode Interrupt Vector	0201H	•	•
Set Segment Base Address	0007H	•	•
Set Segment Limit	0008H	•	•
Simulate Real Mode Interrupt	0300H	•	•
Terminate and Stay Resident	0C01H		•
Unlock Linear Region	0601H	•	•

DPMI Int 31H Functions Listed by Number

Function	**Function**	**DPMI**	**DPMI**
Number	**Name**	**0.9**	**1.0**
0000H	Allocate LDT Descriptor	•	•
0001H	Free LDT Descriptor	•	•
0002H	Map Real-Mode Segment to Descriptor	•	•
0003H	Get Selector Increment Value	•	•
0004H	Reserved		
0005H	Reserved		
0006H	Get Segment Base Address	•	•
0007H	Set Segment Base Address	•	•
0008H	Set Segment Limit	•	•
0009H	Set Descriptor Access Rights	•	•
000AH	Create Alias Descriptor	•	•
000BH	Get Descriptor	•	•
000CH	Set Descriptor	•	•
000DH	Allocate Specific LDT Descriptor	•	•
000EH	Get Multiple Descriptors		•
000FH	Set Multiple Descriptors		•
0100H	Allocate DOS Memory Block	•	•
0101H	Free DOS Memory Block	•	•
0102H	Resize DOS Memory Block	•	•
0200H	Get Real Mode Interrupt Vector	•	•
0201H	Set Real Mode Interrupt Vector	•	•
0202H	Get Processor Exception Handler Vector	•	•
0203H	Set Processor Exception Handler Vector	•	•
0204H	Get Protected Mode Interrupt Vector	•	•
0205H	Set Protected Mode Interrupt Vector	•	•
0210H	Get Extended Processor Exception Handler Vector in Protected Mode		•
0211H	Get Extended Processor Exception Handler Vector in Real Mode		•
0212H	Set Extended Processor Exception Handler Vector in Protected Mode		•
0213H	Set Extended Processor Exception Handler Vector in Real Mode		•
0300H	Simulate Real Mode Interrupt	•	•
0301H	Call Real Mode Procedure with Far Return Frame	•	•
0302H	Call Real Mode Procedure with Interrupt Return Frame	•	•
0303H	Allocate Real Mode Callback Address	•	•
0304H	Free Real Mode Callback Address	•	•
0305H	Get State Save/Restore Addresses	•	•
0306H	Get Raw CPU Mode Switch Addresses	•	•
0400H	Get DPMI Version	•	•
0401H	Get DPMI Capabilities		•
0500H	Get Free Memory Information	•	•
0501H	Allocate Memory Block	•	•

0502H	Free Memory Block	•	•
0503H	Resize Memory Block	•	•
0504H	Allocate Linear Memory Block		•
0505H	Resize Linear Memory Block		•
0506H	Get Page Attributes		•
0507H	Set Page Attributes		•
0508H	Map Device in Memory Block		•
0509H	Map Conventional Memory in Memory Block		•
050AH	Get Memory Block Size and Base		•
050BH	Get Memory Information		•
0600H	Lock Linear Region	•	•
0601H	Unlock Linear Region	•	•
0602H	Mark Real Mode Region as Pageable	•	•
0603H	Relock Real Mode Region	•	•
0604H	Get Page Size	•	•
0700H	Reserved		
0701H	Reserved		
0702H	Mark Page as Demand Paging Candidate	•	•
0703H	Discard Page Contents	•	•
0800H	Physical Address Mapping	•	•
0801H	Free Physical Address Mapping		•
0900H	Get and Disable Virtual Interrupt State	•	•
0901H	Get and Enable Virtual Interrupt State	•	•
0902H	Get Virtual Interrupt State	•	•
0A00H	Get Vendor-Specific API Entry Point	•	•
0B00H	Set Debug Watchpoint	•	•
0B01H	Clear Debug Watchpoint	•	•
0B02H	Get State of Debug Watchpoint	•	•
0B03H	Reset Debug Watchpoint	•	•
0C00H	Install Resident Service Provider Callback		•
0C01H	Terminate and Stay Resident		•
0D00H	Allocate Shared Memory		•
0D01H	Free Shared Memory		•
0D02H	Serialize on Shared Memory		•
0D03H	Free Serialization on Shared Memory		•
0E00H	Get Coprocessor Status		•
0E01H	Set Coprocessor Emulation		•

Int 2FH Function 1680H [1.0]
Release Current Virtual Machine's Time Slice

Called by a client program to indicate that the program is idle (for example, waiting for keyboard input). This allows the DPMI host to pass the CPU to other clients, or take power-conserving measures on laptop and notebook computers.

Call With:

AX	= 1680H

Returns:

if function supported by host

AL	= 0

if function not supported by host

AL	= unchanged (80H)

Notes:

o This function is not specific to DPMI hosts. Some operating systems will recognize this call for programs running in real mode. Programmers are encouraged to use this call in *all* DOS and DPMI-client programs. All DPMI hosts will hook Int 2FH and so a DPMI client can use this API without any other precautions. Non-DPMI programs that can run on DOS 2.xx or earlier should make sure that the Int 2FH vector is non-zero before executing the Int 2FH.

o When an application calls this function it will regain control at intervals, so it should continue to re-issue this function call so long as it has nothing to do.

o DPMI client and application vendors are encouraged to use this function. It can significantly improve the performance of a DOS-based multitasking host.

Int 2FH Function 1686H [0.9]
Get CPU Mode

Returns information about the current CPU mode. Programs which only execute in protected mode do not need to call this function.

Call With:

AX	= 1686H

Returns:

if executing in protected mode

AX	= 0

if executing in real mode or Virtual 86 mode

AX	= nonzero

Notes:

o Some environments support programs or libraries that can execute in either real or protected mode (bimodal code). This function is supplied so that such programs can detect at run time whether they are running in protected mode and make use of system facilities accordingly.

o This function should not be used to determine if a DPMI host is present. A client should make sure that DPMI services are available *before* calling this function; otherwise, the results returned by the function may not be valid.

Int 2FH Function 1687H [0.9]
Obtain Real-to-Protected Mode Switch Entry Point

This function can be called in real mode only to test for the presence of a DPMI host, and to obtain an address of a mode switch routine that can be called to begin execution in protected mode.

Call With:

AX	= 1687h

Returns:

if function successful

AX	= 0	
BX	= flags	
	Bit	*Significance*
	0	0 = 32-bit programs are not supported 1 = 32-bit programs are supported
	1-15	not used
CL	= processor type	
	02H = 80286	
	03H = 80386	
	04H = 80486	
	05H-FFH	Reserved for future Intel processors
DH	= DPMI major version as a decimal number (represented in binary)	
DL	= DPMI minor version as a decimal number (represented in binary)	
SI	= number of paragraphs required for DPMI host private data (may be 0)	
ES:DI	= segment:offset of procedure to call to enter protected mode	

if function unsuccessful (no DPMI host present)

AX	= nonzero

Notes:

- The entry point returned by Int 2FH Function 1687H is *only* called for the *first* switch to protected mode by a DPMI client. For further details on the protocol for switching to protected mode and the environment after switching to protected mode, see page 22.

- Under DPMI hosts, the major version number is returned in DH and the minor version number is returned in DL. There are two decimal digits for the minor version number with the least-significant digit representing the revision number of the minor version number. Under DPMI version 0.9 hosts, DH is returned as 0, and DL is returned as decimal 90 (5AH). In hypothetical DPMI version 2.3, DH would be returned as 2 and DL would be returned as 30 (1EH).

Int 2FH Function 168AH [1.0]
Get Vendor-Specific API Entry Point

Returns an address which can be called to use host-specific extensions to the standard set of DPMI functions. This function is available only in protected mode.

Call With:

AX	= 168AH
DS:(E)SI	= selector:offset of ASCIIZ (null-terminated) string identifying the DPMI host vendor

Returns:

if function successful

AL	= 0
ES:(E)DI	= extended API entry point

and DS, FS, GS, EAX, EBX, ECX, EDX, ESI, and EBP are preserved.

if function unsuccessful

AL	= unchanged (8AH)

Notes:

- The ASCIIZ string specifies a host vendor name or some other unique identifier to obtain a specific extension entry point. The string comparison used to look up the API entry point is case-sensitive.
- Clients must use a `FAR CALL` to reach the extended API entry point.
- All extended API parameters are specified by the vendor.
- DPMI 1.0 clients should use this function in preference to Int 31H Function 0A00H. This method of API extension is preferable to the Int 31H extension as it avoids the creation of a long (and consequently slow) chain of Int 31H handlers which would slow down time-critical DPMI functions. Note that although this function was not documented for DPMI 0.9, it will work under any DPMI 0.9 host.

Int 31H Function 0000H [0.9]
Allocate LDT Descriptors

Allocates one or more descriptors in the task's Local Descriptor Table (LDT). The descriptor(s) allocated must be initialized by the application with other function calls.

Call With:

AX	= 0000H
CX	= number of descriptors to allocate

Returns:

if function successful

Carry flag	= clear
AX	= base selector

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8011H	descriptor unavailable

Notes:

- If more than one descriptor was requested, the function returns a base selector referencing the first of a contiguous array of descriptors. The selector values for subsequent descriptors in the array can be calculated by adding the value returned by Int 31H Function 0003H.

- The allocated descriptor(s) will be set to "data" with the present bit set and a base and limit of zero. The privilege level of the descriptor(s) will match the application's code segment privilege level.

- Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0001H [0.9]
Free LDT Descriptor

Frees an LDT descriptor.

Call With:

AX	= 0001H
BX	= selector for the descriptor to free

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8022H	invalid selector

Notes:

- o Each descriptor allocated with Int 31H Function 0000H must be freed individually with this function, even if it was previously allocated as part of a contiguous array of descriptors.
- o Under DPMI 1.0 hosts, any segment registers which contain the selector being freed are zeroed by this function.
- o Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0002H [0.9]
Segment to Descriptor

Maps a real mode segment (paragraph) address onto an LDT descriptor that can be used by a protected mode program to access the same memory.

Call With:

AX	= 0002H
BX	= real mode segment address

Returns:

if function successful

Carry flag	= clear
AX	= selector for real mode segment

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8011H	descriptor unavailable

Notes:

o The descriptor's limit will be set to 64 KB.

o Multiple calls to this function with the same segment address will return the same selector.

o The intent of this function is to provide clients with easy access to commonly used real mode segments such as the BIOS data area at segment 0040H and the video refresh buffers at segments A000H, B000H, and B800H. Clients should not call this function to obtain descriptors to private data areas.

o Descriptors created by this function can never be modified or freed. For this reason, the function should be used sparingly. Clients which need to examine various real mode addresses using the same selector should allocate a descriptor with Int 31H Function 0000H and change the base address in the descriptor as necessary, using the Set Segment Base Address function (Int 31H Function 0007H).

o Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0003H [0.9]
Get Selector Increment Value

The DPMI functions Allocate LDT Descriptors (Int 31H Function 0000H) and Allocate DOS Memory Block (Int 31H Function 0100H) can allocate an array of contiguous descriptors, but only return a selector for the first descriptor. The value returned by this function can be used to calculate the selectors for subsequent descriptors in the array.

Call With:

AX	= 0003H

Returns:

Carry flag	= clear (this function always succeeds)
AX	= selector increment value

Notes:

o The increment value is always a power of two.

Int 31H Function 0004H [0.9]
Reserved

DPMI Function 0004H is reserved for historical reasons and should not be called.

Int 31H Function 0005H [0.9]
Reserved

DPMI Function 0005H is reserved for historical reasons and should not be called.

Int 31H Function 0006H [0.9]
Get Segment Base Address

Returns the 32-bit linear base address from the LDT descriptor for the specified segment.

Call With:

AX	= 0006H
BX	= selector

Returns:

if function successful

Carry flag	= clear
CX:DX	= 32-bit linear base address of segment

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8022H	invalid selector

Notes:

- o Client programs must use the LSL instruction to query the limit for a descriptor. Note that on 80386 machines, the client must use the 32-bit form of LSL if the segment size is greater than 64 KB.

- o Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0007H [0.9]
Set Segment Base Address

Sets the 32-bit linear base address field in the LDT descriptor for the specified segment.

Call With:

AX	= 0007H
BX	= selector
CX:DX	= 32-bit linear base address of segment

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set		
AX	= error code		
		8022H	invalid selector
		8025H	invalid linear address (changing the base would cause the descriptor to reference a linear address range outside that allowed for DPMI clients)

Notes:

o A DPMI 1.0 host will automatically reload any segment register which contains the selector specified in register BX. It is suggested that DPMI 0.9 hosts also implement this.

o Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0008H [0.9]
Set Segment Limit

Sets the limit field in the LDT descriptor for the specified segment.

Call With:

```
AX          = 0008H
BX          = selector
CX:DX       = 32-bit segment limit
```

Returns:

if function successful
Carry flag = clear

if function unsuccessful

```
Carry flag  = set
AX          = error code
                8021H     invalid value (CX <> 0 on a 16-bit DPMI host; or the limit is greater
                          than 1 MB, but the low twelve bits are not set)
                8022H     invalid selector
                8025H     invalid linear address (changing the limit would cause the
                          descriptor to reference a linear address range outside that allowed
                          for DPMI clients.)
```

Notes:

- The value supplied to the function in CX:DX is the byte length of the segment-1 (i.e., the value returned by the LSL instruction).

- Segment limits greater than or equal to 1 MB must be page-aligned. That is, limits greater than 1 MB must have the low 12 bits set.

- This function has an implicit effect on the "G" (granularity) bit in an 80386 descriptor's extended access rights/type byte; i.e., it is the host's responsibility to set the "G" bit correctly.

- Client programs must use the LSL instruction to *query* the limit for a descriptor. Note that on 80386 machines, the client must use the 32-bit form of LSL if the segment size is greater than 64 KB.

- A DPMI 1.0 host will reload any segment registers which contain the selector specified in register BX. It is suggested that DPMI 0.9 hosts also implement this.

- Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0009H [0.9]
Set Descriptor Access Rights

Modifies the access rights and type fields in the LDT descriptor for the specified segment.

Call With:

AX	= 0009H
BX	= selector
CL	= access rights/type byte
CH	= 80386 extended access rights/type byte

Returns:

if function successful
Carry flag = clear

if function unsuccessful
Carry flag = set
AX = error code

8021H	invalid value (access rights/type bytes invalid)
8022H	invalid selector
8025H	invalid linear address (changing the access rights/type bytes would cause the descriptor to reference a linear address range outside that allowed for DPMI clients.)

Notes:

o The access rights/type byte passed to the function in CL has the following format:

```
+---+---+---+---+---+---+---+---+
| P |  DPL  | 1 |C/D|E/C|W/R| A |
+-+-+---+---+-+-+-+-+-+-+-+-+-+-+
|     |     |   |   |   |   |
|     |     |   |   |   |   +- 0=not accessed, 1=accessed
|     |     |   |   |   +----- data: 0=read, 1=>read/write
|     |     |   |   |          code: must be 1 (readable)
|     |     |   |   +--------- data: 0=expand-up, 1=expand-down
|     |     |   |              code: must be 0 (non-conforming)
|     |     |   +------------- 0=data, 1=code
|     |     +----------------- must be 1
|     +----------------------- must equal caller's CPL
+----------------------------- 0=absent, 1=present
```

If the Present bit is not set in the descriptor, the DPMI host allows any values except in the DPL and "must be 1" bit fields.

- On 80386 (and later) machines, the DPMI host interprets the value passed to the function in CH as follows:

```
+---+---+---+---+---+---+---+---+
| G |B/D| 0 |Avl|    Reserved   |
+-+-+-+-+-+-+-+-+---+---+---+---+
|   |   |   |         |
|   |   |   |         +-- ignored
|   |   |   +------------ can be 0 or 1
|   |   +---------------- must be 0
|   +-------------------- 0=default 16-bit, 1=default 32-bit
+------------------------ 0=byte granular, 1=page granular
```

- A DPMI 1.0 host will reload any segment registers which contain the selector specified in register BX. It is suggested that DPMI 0.9 hosts also implement this.

- Client programs should use the LAR instruction to examine the access rights of a descriptor.

- Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 000AH [0.9]
Create Alias Descriptor

Creates a new LDT data descriptor that has the same base and limit as the specified descriptor.

Call With:

AX	= 000AH
BX	= selector

Returns:

if function successful

Carry flag	= clear
AX	= data selector (alias)

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8011H	descriptor unavailable
	8022H	invalid selector

Notes:

- o The selector supplied to the function may be either a data selector or an executable selector. Note that the published 0.9 specification was in error to say that the function generates an error on a data descriptor.

- o The descriptor alias returned by this function will not track changes to the original descriptor. In other words, if an alias is created with this function, and the base or limit of the original segment is then changed, the two descriptors will no longer map the same memory.

- o Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 000BH [0.9]
Get Descriptor

Copies the local descriptor table (LDT) entry for the specified selector into an 8-byte buffer.

Call With:

AX	= 000BH
BX	= selector
ES:(E)DI	= selector:offset of 8-byte buffer

Returns:

if function successful

Carry flag	= clear

and buffer pointed to by ES:(E)DI contains descriptor

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8022H	invalid selector

Notes:

- o 32-bit programs must use ES:EDI to point to the buffer. 16-bit programs should use ES:DI.
- o Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 000CH [0.9]
Set Descriptor

Copies the contents of an 8-byte buffer into the LDT descriptor for the specified selector.

Call With:

AX	= 000CH
BX	= selector
ES:(E)DI	= selector:offset of 8-byte buffer containing descriptor

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8021H	invalid value (access rights/type byte invalid)
	8022H	invalid selector
	8025H	invalid linear address (descriptor references a linear address range outside that allowed for DPMI clients)

Notes:

- o 32-bit programs must use ES:EDI to point to the buffer. 16-bit programs should use ES:DI.

- o The descriptor's access rights/type byte (byte 5) follows the same format and restrictions as the access rights/type parameter (in CL) for the Set Descriptor Access Rights function (Int 31H Function 0009H). On 80386 (or later) machines, the descriptor's extended access rights/type byte (byte 6) follows the same format and restrictions as the extended access rights/type parameter (in CH) for the same function, except the low-order 4 bits (marked "reserved") are used to set the upper 4 bits of the descriptor's limit.

- o If the descriptor's present bit is not set, then the only error checking is that the client's CPL must be equal to the descriptor's DPL field and the "must be 1" bit in the descriptor's byte 5 must be set.

- o A DPMI 1.0 host will reload any segment register which contains a selector specified in register BX. It is suggested that DPMI 0.9 hosts also implement this.

- o Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 000DH [0.9]
Allocate Specific LDT Descriptor

Allocates a specific LDT descriptor.

Call With:

AX	= 000DH
BX	= selector

Returns:

if function successful
Carry flag = clear

and descriptor has been allocated

if function unsuccessful
Carry flag = set
AX = error code

8011H	descriptor unavailable (descriptor is in use)
8022H	invalid selector (references GDT or beyond the LDT limit)

Notes:

- The first 10H (16) descriptors (selector values 04H-7CH) are reserved for this function and must not be used by the DPMI host.

- Under DPMI 0.9 hosts, if another application has already been loaded, some of descriptors reserved for allocation by this function may be already in use and unavailable. Under DPMI 1.0 hosts, each client has its own LDT and thus will have the full 16 descriptors available for use with this function.

- Resident service providers (protected-mode TSRs) should *not* use this function.

- Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 000EH [1.0]
Get Multiple Descriptors

Copies one or more local descriptor table (LDT) entries into a client buffer.

Call With:

AX	= 000EH
CX	= number of descriptors to copy
ES:(E)DI	= selector:offset of a buffer in the following format:

Offset	*Length*	*Contents*
00H	2	Selector #1 (set by client)
02H	8	Descriptor #1 (returned by host)
0AH	2	Selector #2 (set by client)
0CH	8	Descriptor #2 (returned by host)
.	.	.
.	.	.
.	.	.

Returns:

if function successful

Carry flag	= clear

and buffer contains copies of the descriptors for the specified selectors

if function unsuccessful

Carry flag	= set
AX	= error code 8022H invalid selector
CX	= number of descriptors successfully copied

Notes:

o If an error occurs because of an invalid selector or descriptor, the function returns the number of descriptors which were successfully copied in CX. All of the descriptors which were copied prior to the one that failed are valid.

o 32-bit programs must use ES:EDI to point to the buffer. 16-bit programs should use ES:DI.

o Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 000FH [1.0]
Set Multiple Descriptors

Copies one or more descriptors from a client buffer into the local descriptor table (LDT).

Call With:

AX = 000FH
CX = number of descriptors to copy
ES:(E)DI = selector:offset of a buffer in the following format:

Offset	*Length*	*Contents*
00H	2	Selector #1
02H	8	Descriptor #1
0AH	2	Selector #2
0CH	8	Descriptor #2
.	.	.
.	.	.
.	.	.

Returns:

if function successful
Carry flag = clear

if function unsuccessful
Carry flag = set
AX = error code
8021H invalid value (access rights/type bytes invalid)
8022H invalid selector 8025H invalid linear address (descriptor references a linear address range outside that allowed for DPMI clients)
CX = number of descriptors successfully copied

Notes:

- If an error occurs because of an invalid selector or descriptor, the function returns the number of descriptors which were successfully copied in CX. All of the descriptors which were copied prior to the one that failed are valid. All descriptors from the invalid entry to the end of the table are not updated.

- 32-bit programs must use ES:EDI to point to the buffer. 16-bit programs should use ES:DI.

- A descriptor's access rights/type byte (byte 5) follows the same format and restrictions as the access rights/type parameter (in CL) for the Set Descriptor Access Rights function (Int 31H Function 0009H). On 80386 (or later) machines, the descriptor's extended access rights/type byte (byte 6) follows the same format and restrictions as the extended access rights/type parameter (in CH) for the same function, except the low-order 4 bits (marked "reserved") are used to set the upper 4 bits of the descriptor's limit.

- If the descriptor's present bit is not set, then the only error checking is that the client's CPL must be equal to the descriptor's DPL field and the "must be 1" bit in the descriptor's byte 5 must be set.

- A DPMI 1.0 host will reload any segment register which contains a selector specified in the data structure supplied to this function. It is suggested that DPMI 0.9 hosts also implement this.

- Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0100H [0.9]
Allocate DOS Memory Block

Allocates a block of memory from the DOS memory pool, i.e. memory below the 1 MB boundary that is controlled by DOS. Such memory blocks are typically used to exchange data with real mode programs, TSRs, or device drivers. The function returns both the real mode segment base address of the block and one or more descriptors that can be used by protected mode applications to access the block.

Call With:

AX = 0100H
BX = number of (16-byte) paragraphs desired

Returns:

if function successful
Carry flag = clear
AX = real mode segment base address of allocated block
DX = selector for allocated block

if function unsuccessful
Carry flag = set
AX = error code
0007H memory control blocks damaged (also returned by DPMI 0.9 hosts)
0008H insufficient memory (also returned by DPMI 0.9 hosts).
8011H descriptor unavailable
BX = size of largest available block in paragraphs

Notes:

- If the size of the block requested is greater than 64 KB (BX > 1000H) and the client is a 16-bit program, contiguous descriptors are allocated and the base selector is returned. The consecutive selectors for the memory block can be calculated using the value returned by the Get Selector Increment Value function (Int 31H Function 0003H). Each descriptor has a limit of 64 KB, except for the last which has a limit of blocksize MOD 64 KB.

- If the DPMI host is 32-bit, the client is 16-bit, and more than one descriptor is allocated, the limit of the first descriptor will be set to the size of the entire block. Subsequent descriptors have limits as described in the previous Note. 16-bit DPMI hosts will always set the limit of the first descriptor to 64 KB even when running on an 80386 (or later) machine.

- When the client is 32-bit, this function always allocates only one descriptor.

- Client programs should never modify or free any descriptors allocated by this function. The Free DOS Memory Block function (Int 31H Function 0101H) will deallocate the descriptors automatically.

- The DOS allocation function (Int 21H Function 48H) is used.

- Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0101H [0.9]
Free DOS Memory Block

Frees a memory block that was previously allocated with the Allocate DOS Memory Block function (Int 31H Function 0100H).

Call With:

AX	= 0101H
DX	= selector of block to be freed

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	0007H	memory control blocks damaged (also returned by DPMI 0.9 hosts).
	0009H	incorrect memory segment specified (also returned by DPMI 0.9 hosts).
	8022H	invalid selector

Notes:

- All descriptors allocated for the memory block are automatically freed by this function, and are no longer valid after this function returns.

- Under DPMI 1.0 hosts, any segment registers which contain a selector being freed are zeroed by this function.

- Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0102H [0.9]
Resize DOS Memory Block

Changes the size of a memory block that was previously allocated with the Allocate DOS Memory Block function (Int 31H Function 0100H).

Call With:

AX	= 0102H
BX	= new block size in (16-byte) paragraphs
DX	= selector of block to modify

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	0007H	memory control blocks damaged (also returned by DPMI 0.9 hosts).
	0008H	insufficient memory (also returned by DPMI 0.9 hosts).
	0009H	incorrect memory segment specified (also returned by DPMI 0.9 hosts).
	8011H	descriptor unavailable
	8022H	invalid selector
BX	= maximum possible block size (paragraphs)	

Notes:

o Requests to increase the size of an existing DOS memory block may fail due to subsequent DOS memory block allocations causing fragmentation of DOS memory, or insufficient remaining DOS memory. In addition, the function will fail if the block is growing past a 64 KB boundary and the next descriptor in the LDT is not available.

o A request to decrease the size of a DOS memory block may cause some descriptors that were previously allocated to the block to be freed and the limit of the new last descriptor for the block to be changed.

o Under a DPMI 1.0 host, any segment registers which contain a selector being modified are reloaded by this function and any segment registers which contain a selector being freed are zeroed by this function.

o Client programs should never modify or free any descriptors allocated by this function. The Free DOS Memory Block function (Int 31H Function 0101H) will deallocate the descriptors automatically.

o Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0200H [0.9]
Get Real Mode Interrupt Vector

Returns the contents of the current virtual machine's real mode interrupt vector for the specified interrupt.

Call With:

AX	= 0200H
BL	= interrupt number

Returns:

Carry flag	= clear (this function always succeeds)
CX:DX	= segment:offset of real mode interrupt handler

Notes:

- o The value returned in CX is a real mode segment address, not a selector. Attempts to place this value into a segment register in protected mode may cause a general protection (GP) fault.

- o All 100H (256) real mode interrupt vectors must be made available through this function by the DPMI host.

Int 31H Function 0201H [0.9]
Set Real Mode Interrupt Vector

Sets the current virtual machine's real mode interrupt vector for the specified interrupt.

Call With:

AX	= 0201H
BL	= interrupt number
CX:DX	= segment:offset of real mode interrupt handler

Returns:

Carry flag	= clear (this function always succeeds)

Notes:

- The address passed in CX must be a real mode segment address, not a selector. Consequently, the interrupt handler must either reside in DOS memory (i.e. below the 1 MB boundary) or the client must allocate a real mode callback address. See Int 31H Functions 0100H and 0303H.

- If the interrupt being hooked is a hardware interrupt, the memory that the interrupt handler uses must be locked.

Int 31H Function 0202H [0.9]
Get Processor Exception Handler Vector

Returns the address of the current client's protected mode exception handler for the specified exception number. This function should be avoided by DPMI 1.0 clients (see Notes).

Call With:

AX	= 0202H
BL	= exception number (00H-1FH)

Returns:

if function successful

Carry flag	= clear
CX:(E)DX	= selector:offset of exception handler

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8021H	invalid value (BL not in range 0-1FH)

Notes:

- o The value returned in CX is a valid protected mode selector, not a real mode segment address.
- o 32-bit clients will be returned a 32-bit offset in the EDX register.
- o Clients which run under DPMI 1.0 should use Int 31H Functions 0210H and 0211H to obtain the addresses of exception handlers. This function is supported by DPMI 1.0 hosts solely for compatibility with DPMI 0.9.

Int 31H Function 0203H [0.9]
Set Processor Exception Handler Vector

Sets the address of a handler for a CPU exception or fault, allowing a protected mode application to intercept processor exceptions (such as segment not present faults) that are not handled by the DPMI host and would otherwise generate a fatal error. This function should be avoided by DPMI 1.0 clients (see Notes).

Call With:

AX	= 0203H
BL	= exception/fault number (00H-1FH)
CX:(E)DX	= selector:offset of exception handler

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8021H	invalid value (BL not in range 0-1FH)
	8022H	invalid selector

Notes:

- The value passed in CX should be a valid protected mode code (executable) selector, not a real mode segment address.

- 32-bit clients must supply a 32-bit offset in the EDX register. If the client's handler chains to the next exception handler, it must do so using a 32-bit interrupt stack frame.

- Every exception is first examined by the DPMI host. If the host does not handle the exception, it reflects the exception to the first handler in the protected mode exception handler chain. See page 30 for a complete discussion of the environment and responsibilities of protected mode exception handlers installed with this function.

- Clients which run under DPMI 1.0 should use Int 31H Functions 0212H and 0213H to set the addresses of exception handlers. This function is supported by DPMI 1.0 hosts solely for compatibility with DPMI 0.9.

- Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0204H [0.9]
Get Protected Mode Interrupt Vector

Returns the address of the current protected mode interrupt handler for the specified interrupt.

Call With:

AX	= 0204H
BL	= interrupt number

Returns:

Carry flag	= clear (this function always succeeds)
CX:(E)DX	= selector:offset of exception handler

Notes:

o The value returned in CX is a valid protected mode selector, not a real mode segment address.

o 32-bit clients will be returned a 32-bit offset in the EDX register.

o DPMI hosts must make all 100H (256) interrupt vectors available through this function.

Int 31H Function 0205H [0.9]
Set Protected Mode Interrupt Vector

Sets the address of protected mode handler for the specified interrupt into the interrupt vector.

Call With:

AX	= 0205H
BL	= interrupt number
CX:(E)DX	= selector:offset of exception handler

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8022H	invalid selector

Notes:

- o The value passed in CX should be a valid protected mode code selector, not a real mode segment address.
- o 32-bit clients must supply a 32-bit offset in the EDX register. If the client's handler chains to the next exception handler it must do so using a 32-bit interrupt stack frame.
- o DPMI hosts must support all 100H (256 decimal) interrupt vectors with this function.
- o Hardware interrupts are sent to the primary client of the virtual machine while software interrupts are sent to the current client. (See Appendix A: Glossary for definitions of primary and current client.)
- o Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0210H [1.0]
Get Extended Processor Exception Handler Vector (Protected Mode)

Returns the address of the client's protected mode handler for the specified protected mode exception.

Call With:

AX = 0210H
BL = exception number (00H-1FH)

Returns:

if function successful
Carry flag = clear
CX:(E)DX = selector:offset of exception handler

if function unsuccessful
Carry flag = set
AX = error code
8021H invalid value (BL not in range 00H-1FH)

Notes:

- DPMI 1.0 clients should use this function in preference to Int 31H Function 0202H.

- The protected mode exceptions are sent to the protected mode handler of the current client. (See Appendix A: Glossary for definition of primary client.)

Int 31H Function 0211H [1.0]
Get Extended Processor Exception Handler Vector (Real Mode)

Returns the address of the client's protected mode handler for the specified real mode exception.

Call With:

AX	= 0211H
BL	= exception number (00H-1FH)

Returns:

if function successful

Carry flag	= clear
CX:(E)DX	= selector:offset of exception handler

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8021H	invalid value (BL not in range 00H-1FH)

Notes:

- o CX:(E)DX does not specify a real-mode segment:offset. The reason is that this function allows a client to get the address of the exception handler which will receive control in protected mode when the specified exception occurs in real mode (i.e. the host will provide an implied mode switch for the purposes of servicing the exception, then return to real mode after the handler exits)..

- o Real mode exceptions are sent to the primary client of the virtual machine.

Int 31H Function 0212H [1.0]
Set Extended Processor Exception Handler Vector (Protected Mode)

Sets the address of the client's protected mode handler for the specified protected mode exception.

Call With:

AX	= 0212H
BL	= exception/fault number (00H-1FH)
CX:(E)DX	= selector:offset of exception handler

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8021H	invalid value (BL not in range 00H-1FH)
	8022H	invalid selector

Notes:

- DPMI 1.0 clients should use this function in preference to Int 31H Function 0203H.
- The protected mode exceptions are sent to the protected mode handler of the current client.
- Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0213H [1.0]
Set Extended Processor Exception Handler Vector (Real Mode)

Sets the address of the client's protected mode handler for the specified real mode exception.

Call With:

AX	= 0213H
BL	= exception/fault number (00H-1FH)
CX:(E)DX	= selector:offset of exception handler

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8021H	invalid value (BL not in range 00H-1FH)
	8022H	invalid selector

Notes:

- CX:(E)DX does not specify a real-mode segment:offset. The reason is that this function allows a client to set the address of an exception handler which will receive control in protected mode when the specified exception occurs in real mode (i.e. the host will provide an implied mode switch for the purposes of servicing the exception, then return to real mode after the handler exits).

- Real mode exceptions are sent to the primary client of the virtual machine. (See Appendix A: Glossary for definition of primary client.)

- Refer to the rules for descriptor usage in Appendix D.

Int 31H Function 0300H [0.9]
Simulate Real Mode Interrupt

Simulates an interrupt in real mode. The function transfers control to the address specified by the real mode interrupt vector. The real mode handler must return by executing an `IRET`.

Call With:

AX	= 0300H
BL	= interrupt number
BH	= flags

Bit	*Significance*
0	reserved for historical reason, must be zero
1-7	reserved, must be zero

CX	= number of words to copy from protected mode to real mode stack
ES:(E)DI	= selector:offset of real mode register data structure in the following format:

Offset	*Length*	*Contents*
00H	4	DI or EDI
04H	4	SI or ESI
08H	4	BP or EBP
0CH	4	reserved, should be zero
10H	4	BX or EBX
14H	4	DX or EDX
18H	4	CX or ECX
1CH	4	AX or EAX
20H	2	CPU status flags
22H	2	ES
24H	2	DS
26H	2	FS
28H	2	GS
2AH	2	IP (reserved, ignored)
2CH	2	CS (reserved, ignored)
2EH	2	SP
30H	2	SS

Returns:

if function successful

Carry flag	= clear
ES:(E)DI	= selector:offset of modified real mode register data structure

if function unsuccessful

Carry flag	= set
AX	= error code

8012H	linear memory unavailable (stack)
8013H	physical memory unavailable (stack)
8014H	backing store unavailable (stack)
8021H	invalid value (CX too large)

Notes:

- 32-bit programs must use ES:EDI to point to the real mode register data structure. 16-bit programs should use ES:DI.

- The CS:IP in the real mode register data structure is ignored by this function. The appropriate interrupt handler will be called based on the value passed in BL.

- If the SS:SP fields in the real mode register data structure are zero, 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 register data structure will be pushed on the real mode stack's IRET frame. The interrupt handler will be called with the interrupt and trace flags clear.

- Values placed in the segment register positions of the data structure must be valid for real mode; i.e. the values must be paragraph addresses and not selectors.

- All general register fields in the data structure are DWORDs so that 32-bit registers can be passed to real mode. Note, however, that 16-bit hosts are not required to pass the high word of 32-bit general registers or the FS and GS segment registers to real mode even when running on an 80386 or later CPU.

- The target real mode handler 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 must return on the same stack that it was called on and must return with an IRET.

- When this function returns, the real mode register data structure will contain the values that were returned by the real mode interrupt handler.

- It is the caller's responsibility to remove any parameters that were pushed on the protected mode stack.

Int 31H Function 0301H [0.9]
Call Real Mode Procedure With Far Return Frame

Simulates a FAR CALL to a real mode procedure. The called procedure must return by executing a RETF (far return) instruction.

Call With:

AX	= 0301H
BH	= flags

Bit	*Significance*
0	reserved for historical reason, must be zero
1-7	reserved, must be zero

CX	= number of words to copy from protected mode to real mode stack
ES:(E)DI	= selector:offset of real mode register data structure in the following format:

Offset	*Length*	*Contents*
00H	4	DI or EDI
04H	4	SI or ESI
08H	4	BP or EBP
0CH	4	reserved, ignored
10H	4	BX or EBX
14H	4	DX or EDX
18H	4	CX or ECX
1CH	4	AX or EAX
20H	2	CPU status flags
22H	2	ES
24H	2	DS
26H	2	FS
28H	2	GS
2AH	2	IP
2CH	2	CS
2EH	2	SP
30H	2	SS

Returns:

if function successful

Carry flag	= clear
ES:(E)DI	= selector:offset of modified real mode register data structure

if function unsuccessful

Carry flag	= set
AX	= error code

8012H	linear memory unavailable (stack)
8013H	physical memory unavailable (stack)
8014H	backing store unavailable (stack)
8021H	invalid value (CX too large)

Notes:

- o 32-bit programs must use ES:EDI to point to the real mode register data structure. 16-bit programs should use ES:DI.

- o The CS:IP in the real mode register data structure specifies the address of the real mode procedure to call.

- o If the SS:SP fields in the real mode register data structure are zero, 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.

- o Values placed in the segment register positions of the data structure must be valid for real mode; i.e. the values must be paragraph addresses and not selectors.

- o All general register fields in the data structure are `DWORD`s so that 32-bit registers can be passed to real mode. Note, however, that 16-bit hosts are not required to pass the high word of 32-bit general registers or the FS and GS segment registers to real mode even when running on an 80386 or later CPU.

- o The target real mode procedure 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 must return on the same stack that it was called on and must exit with a `RETF` (far return) and should not clear the stack of any other parameters that were passed to it on the stack.

- o When this function returns, the real mode register data structure will contain the values that were returned by the real mode procedure.

- o It is the caller's responsibility to remove any parameters that were pushed on the protected mode stack.

Int 31H Function 0302H [0.9]
Call Real Mode Procedure With IRET Frame

Simulates a `FAR CALL` with flags pushed on the stack to a real mode procedure. The real mode routine must return by executing an `IRET` instruction.

Call With:

AX	= 0302H
BH	= flags

Bit	*Significance*
0	reserved for historical reason, must be zero
1-7	reserved, must be zero

CX	= number of words to copy from protected mode to real mode stack
ES:(E)DI	= selector:offset of real mode register data structure in the following format:

Offset	*Length*	*Contents*
00H	4	DI or EDI
04H	4	SI or ESI
08H	4	BP or EBP
0CH	4	reserved, ignored
10H	4	BX or EBX
14H	4	DX or EDX
18H	4	CX or ECX
1CH	4	AX or EAX
20H	2	CPU status flags
22H	2	ES
24H	2	DS
26H	2	FS
28H	2	GS
2AH	2	IP
2CH	2	CS
2EH	2	SP
30H	2	SS

Returns:

if function successful

Carry flag	= clear
ES:(E)DI	= selector:offset of modified real mode register data structure

if function unsuccessful

Carry flag	= set
AX	= error code

8012H	linear memory unavailable (stack)
8013H	physical memory unavailable (stack)
8014H	backing store unavailable (stack)
8021H	invalid value (CX too large)

Notes:

- 32-bit programs must use ES:EDI to point to the real mode register data structure. 16-bit programs should use ES:DI.

- The CS:IP in the real mode register data structure specifies the address of the real mode procedure to call.

- If the SS:SP fields in the real mode register data structure are zero, 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 register data structure will be pushed on the real mode stack's IRET frame. The procedure will be called with the interrupt and trace flags clear.

- Values placed in the segment register positions of the data structure must be valid for real mode; i.e. the values must be paragraph addresses and not selectors.

- All general register fields in the data structure are DWORDs so that 32-bit registers can be passed to real mode. Note, however, that 16-bit hosts are not required to pass the high word of 32-bit general registers or the FS and GS segment registers to real mode even when running on an 80386 or later CPU.

- The target real mode handler or procedure 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 must return on the same stack that it was called on and must return with an IRET or discard the flags from the stack with a RETF(2) .

- When this function returns, the real mode register data structure will contain the values that were returned by the real mode procedure.

- It is the caller's responsibility to remove any parameters that were pushed on the protected mode stack.

Int 31H Function 0303H [0.9]
Allocate Real Mode Callback Address

Returns a unique real mode segment:offset, known as a "real mode callback," that will transfer control from real mode to a protected mode procedure. Callback addresses obtained with this function can be passed by a protected mode program to a real mode application, interrupt handler, device driver, or TSR, so that the real mode program can call procedures within the protected mode program or notify the protected mode program of an event.

Call With:

AX	= 0303H
DS:(E)SI	= selector:offset of protected mode procedure to call
ES:(E)DI	= selector:offset of 32H-byte buffer for real mode register data structure to be used when calling callback routine.

Returns:

if function successful

Carry flag	= clear
CX:DX	= segment:offset of real mode callback

if function unsuccessful

Carry flag	= set
AX	= error code 8015H callback unavailable

Notes:

- o DPMI hosts must provide a minimum of 16 callback addresses per client.

- o A descriptor may be allocated for each callback to hold the real mode SS descriptor. Real mode callbacks are a limited system resource. A client should use the Free Real Mode Callback Address function (Int 31H Function 0304H) to release a callback that it is no longer using.

- o For further information on writing real mode callback procedures, see page 34.

- o The contents of the real mode register data structure is not valid after the function call, but only at the time of the actual callback.

Int 31H Function 0304H [0.9]
Free Real Mode Callback Address

Releases a real mode callback address that was previously allocated with the Allocate Real Mode Callback Address function (Int 31H Function 0303H).

Call With:

AX	= 0304H
CX:DX	= real mode callback address to be freed

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8024H	invalid callback address

Notes:

o Real mode callbacks are a limited system resource. A client should release any callback that it is no longer using.

Int 31H Function 0305H [0.9]
Get State Save/Restore Addresses

Returns the addresses of two procedures used to save and restore the state of the current task's registers in the mode which is *not* currently executing.

Call With:

AX	= 0305H

Returns:

Carry flag	= clear (this function always succeeds)
AX	= size of buffer in bytes required to save state
BX:CX	= real mode address of routine used to save/restore state
SI:(E)DI	= protected mode address of routine used to save/restore state

Notes:

- o The real mode address returned by this function in BX:CX is called *only in real mode* to save/restore the state of the protected mode registers. The protected mode address returned by this function in SI:(E)DI is called *only in protected mode* to save/restore the state of the real mode registers; 16-bit programs should call the address in SI:DI, 32-bit programs should call the address in SI:EDI. Registers for the current mode can be saved by simply pushing them on the stack.

- o Both of the state-save procedures are entered by a `FAR CALL` with the following parameters:

AL	= 0 to save state
	= 1 to restore state
ES:(E)DI	= (selector or segment):offset of state-save buffer

  The state-save buffer must be at least as large as the value returned in AX by Int 31H Function 0305H. The state save/restore procedures do not modify any registers. For a further discussion of use of the state save/restore procedures, see page 25.

- o Some DPMI hosts will not require the state to be saved, indicating this by returning a buffer size of zero in AX. In such cases, the addresses returned by this function can still be called, although they will simply return without performing any useful function.

- o Clients do not need to call the state save/restore procedures before using Int 31H Functions 0300H, 0301H, or 0302H. The state save/restore procedures are provided specifically for clients that use the raw mode switch services.

- o A client can use the function to save its state in the destination mode before switching modes using the raw mode switch or issuing real-mode calls from a protected mode hardware interrupt handler. Refer to page 24 for the detailed information on stacks and mode switching.

Int 31H Function 0306H [0.9]
Get Raw Mode Switch Addresses

Returns addresses that can be called for low-level mode switching.

Call With:

AX	= 0306H

Returns:

Carry flag	= clear (this function always succeeds)
BX:CX	= real-to-protected mode switch address
SI:(E)DI	= protected-to-real mode switch address

Notes:

- o 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 by this function in SI:DI, while 32-bit programs should call the address returned in SI:EDI.

- o The mode switch procedures are entered by a `FAR JMP` to the appropriate address with the following parameters:

AX	= new DS
CX	= new ES
DX	= new SS
(E)BX	= new (E)SP
SI	= new CS
(E)DI	= new (E)IP

The processor is placed into the desired mode, and the DS, ES, SS, (E)SP, CS, and (E)IP registers are updated with the specified values; in other words, execution of the client continues in the requested mode at the address provided in registers SI:(E)DI. The values specified to be placed into the segment registers must be appropriate for the destination mode; if invalid selectors are supplied when switching into protected mode, an exception will occur.

The values in (E)AX, (E)BX, (E)CX, (E)DX, (E)SI, and (E)DI after the mode switch are undefined. (E)BP will be preserved across the mode switch call so it can be used as a pointer. On an 80386 or later CPU, the FS and GS segment registers will contain zero after the mode switch.

If interrupts are disabled when the mode switch procedure is invoked, they will not be re-enabled by the DPMI host (even temporarily).

- o It is up to the client to save and restore the state of the task when using this function to switch modes. This usually requires using the state save/restore procedures whose addresses are returned by Int 31H Function 0305H (see page 94).

- Clients may find it more convenient to use Int 31H Functions 0300H, 0301H, and 0302H for mode switching than this function.

Int 31H Function 0400H [0.9]
Get Version

Returns the version number of the DPMI Specification implemented by the DPMI host. Clients can use this information to determine which function calls are supported in the current environment.

Call With:

AX	= 0400H

Returns:

Carry flag	= clear (this function always succeeds)		
AH	= DPMI major version as a binary number		
AL	= DPMI minor version as a binary number		
BX	= flags		
		Bits	*Significance*
		0	0 = host is 16-bit DPMI implementation 1 = host is 32-bit (80386) DPMI implementation
		1	0 = CPU returned to Virtual 86 mode for reflected interrupts 1 = CPU returned to real mode for reflected interrupts
		2	0 = virtual memory not supported 1 = virtual memory supported
		3	reserved, for historical reasons
		4-15	reserved for later use
CL	= processor type		
		02H = 80286	
		03H = 80386	
		04H = 80486	
		05H-FFH	reserved for future Intel processors
DH	= current value of virtual master PIC base interrupt		
DL	= current value of virtual slave PIC base interrupt		

Notes:

o Under DPMI hosts, the major version number is returned in DH and the minor version number is returned in DL. There are two decimal digits for the minor version number with the least-significant digit representing the revision number of the minor version number. Under DPMI version 0.9 hosts, DH is returned as 0, and DL is returned as decimal 90 (5AH). In hypothetical DPMI version 2.3, DH would be returned as 2 and DL would be returned as 30 (1EH).

Int 31H Function 0401H [1.0]
Get DPMI Capabilities

Returns information about the capabilities of the DPMI host, including its support or lack of support for optional features in the DPMI Specification. Clients can use this information to optimize their use of system resources in the current environment.

Call With:

AX = 0401H
ES:(E)DI = selector:offset of 128-byte buffer

Returns:

if function successful
Carry flag = clear (this function always succeeds in DPMI 1.0)
AX = capabilities flags

Bits	*Significance*
0	0 = PAGED ACCESSED/DIRTY capability *not* supported
	1 = PAGED ACCESSED/DIRTY capability supported
1	0 = EXCEPTIONS RESTARTABILITY capability *not* supported
	1 = EXCEPTIONS RESTARTABILITY capability supported
2	0 = DEVICE MAPPING capability *not* supported
	1 = DEVICE MAPPING capability supported
3	0 = CONVENTIONAL MEMORY MAPPING capability *not* supported
	1 = CONVENTIONAL MEMORY MAPPING capability supported
4	0 = DEMAND ZERO-FILL capability *not* supported
	1 = DEMAND ZERO-FILL capability supported
5	0 = WRITE-PROTECT CLIENT capability not supported
	1 = WRITE-PROTECT CLIENT capability supported
6	0 = WRITE-PROTECT HOST capability not supported
	1 = WRITE-PROTECT HOST capability supported
7-15	reserved

CX = reserved, must be 0
DX = reserved, must be 0
ES:(E)DI = selector:offset of 128-byte buffer filled in by host with information as follows:

Offset	*Length*	*Contents*
0	1	Host major version number as a decimal number
1	1	Host minor version number as a decimal number
2	1-126	ASCIIZ (null-terminated) string identifying the DPMI host vendor

if function unsuccessful
Carry flag = set (this function always fails in DPMI 0.9)

Notes:

o PAGE ACCESSED/DIRTY capability means the DPMI host maintains page dirty and accessed bits that can be read and written with the Get and Set Page Attributes functions (Int

31H Functions 0506H and 0507H). This capability must be supported, and must read and write the hardware-level dirty and accessed bits, if the DPMI host does not provide demand-paged virtual memory. If the DPMI host does support virtual memory, this capability is optional, and if present gives the client the ability to read and write virtual page dirty and accessed bits maintained by the host.

o EXCEPTION RESTARTABILITY capability means that a faulting instruction inside the host kernel can always be restarted if the client's exception handler corrects the reason for the exception and returns. Exception restartability allows a client to provide virtual memory under a DPMI host without virtual memory, or to support memory-mapped files under any DPMI host.

o DEVICE MAPPING capability means that the DPMI host supports the optional Map Device function (Int 31H Function 0508H).

o CONVENTIONAL MEMORY MAPPING capability means that the DPMI host supports the optional Map Conventional Memory function (Int 31H Function 0509H).

o DEMAND ZERO-FILL capability means the DPMI host guarantees that all committed pages are initialized to zero when they are created. If this capability is not supported, the contents of newly committed pages are undefined.

o WRITE-PROTECT CLIENT capability means the DPMI host guarantees that the client is running at a privilege level such that write protection of pages is effective for the client's accesses and will geneate page faults.

o WRITE-PROTECT HOST capability means the DPMI host guarantees that the host has configured the system such that write protection of pages is effective for the host's accesses and will generate page faults.

o The host major and minor version numbers are OEM-specific and are not the DPMI version numbers.

Int 31H Function 0500H [0.9]
Get Free Memory Information

Returns information about the amount of available physical memory, linear address space, and disk space for page swapping. Since DPMI clients will often run in multitasking environments, the information returned by this function should only be considered as advisory. DPMI 1.0 clients should avoid use of this function (see the last note of the call).

Call With:

AX	= 0500H
ES:(E)DI	= selector:offset of 48-byte buffer

Returns:

Carry flag = clear (this function always succeeds)

and the buffer is filled in with the following information:

Offset	*Length*	*Contents*
00H	4	Largest available free block in bytes
04H	4	Maximum unlocked page allocation in pages
08H	4	Maximum locked page allocation in pages
0CH	4	Linear address space size in pages
10H	4	Total number of unlocked pages
14H	4	Total number of free pages
18H	4	Total number of physical pages
1CH	4	Free linear address space in pages
20H	4	Size of paging file/partition in pages
24H	0CH	Reserved, all bytes set to 0FFH

Notes:

o 32-bit programs must use ES:EDI to point to the buffer. 16-bit programs should use ES:DI.

o Only the first field of the returned structure is guaranteed to contain a valid value. Any fields that are not supported by the DPMI host will be set by the host to -1 (0FFFFFFFFH) to indicate that the information is not available.

o The field at buffer 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.

o The field at buffer offset 04H specifies the largest number of pages that could be allocated (the value at offset 00H divided by the page size).

o The field at buffer offset 08H specifies the largest block of memory in pages that could be allocated and then locked.

o The field at buffer offset 0CH specifies the size of the total linear address space in pages. This value includes all linear address space that has already been allocated.

o The field at buffer 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 buffer 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.

- The size of the pages used by the DPMI host can be obtained with the Get Page Size function (Int 31H Function 0604H).

- DPMI 1.0 clients should use Int 31H Function 050BH in preference to this function. This function is supported in DPMI 1.0 solely for backward compatibility with DPMI 0.9.

Int 31H Function 0501H [0.9]
Allocate Memory Block

Allocates and commits a block of linear memory.

Call With:

AX	= 0501H
BX:CX	= size of block (bytes, must be nonzero)

Returns:

if function successful

Carry flag	= clear
BX:CX	= linear address of allocated memory block
SI:DI	= memory block handle (used to resize and free block)

if function unsuccessful

Carry flag = set
AX = error code

8012H	linear memory unavailable
8013H	physical memory unavailable
8014H	backing store unavailable
8016H	handle unavailable
8021H	invalid value (BX:CX = 0)

Notes:

- The allocated block is guaranteed to have at least paragraph alignment.

- This function always creates committed pages.

- This function does not allocate any descriptors for the memory block. It is the responsibility of the client to allocate and initialize any descriptors needed to access the memory with additional DPMI function calls.

- Under DPMI hosts that support virtual memory, the memory block will be allocated unlocked. The client can lock some or all of the memory after it is allocated with the Lock Linear Region function (Int 31H Function 0600H).

- Under many DPMI hosts, allocations by this function are page granular. This means, for example, that if the DPMI host uses a page size of 4 KB (1000H), an allocation of 1001H bytes will actually result in an allocation of 2000H bytes. Therefore, it is best to always allocate memory in multiples of the unit of granularity (under DPMI 0.9, use 4K bytes), which can be obtained with Int 31H Function 0604H.

Int 31H Function 0502H [0.9]
Free Memory Block

Frees a memory block that was previously allocated with either the Allocate Memory Block function (Int 31H Function 0501H) or the Allocate Linear Memory Block function (Int 31H Function 0504H).

Call With:

AX	= 0502H
SI:DI	= memory block handle

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8023H	invalid handle

Notes:

- o This call will correctly free all of the possible page types that can occur in a memory block: committed pages, uncommitted pages, and mapped pages (see Appendix A: Glossary).

- o No descriptors are freed by this call. It is the client's responsibility to free any descriptors that it previously allocated to map the memory block. Descriptors should be freed before linear memory blocks.

Int 31H Function 0503H [0.9]
Resize Memory Block

Changes the size of a memory block that was previously allocated with either the Allocate Memory Block function (Int 31H Function 0501H) or the Allocate Linear Memory Block function (Int 31H Function 0504H).

Call With:

AX	= 0503H
BX:CX	= new size of block (bytes, must be nonzero)
SI:DI	= memory block handle

Returns:

if function successful

Carry flag	= clear
BX:CX	= new linear address of memory block
SI:DI	= new handle for memory block

if function unsuccessful

Carry flag	= set
AX	= error code

8012H	linear memory unavailable
8013H	physical memory unavailable
8014H	backing store unavailable
8016H	handle unavailable
8021H	invalid value (BX:CX = 0)
8023H	invalid handle (in SI:DI)

Notes:

- After this function returns, the previous handle for the memory block is invalid and should not be used.

- When increasing the size of a block, this function always creates committed pages. When decreasing the block size, this call will correctly free all possible page types (committed pages, uncommitted pages, and mapped pages). The linear address and handle of the memory block may change as a result of this call.

- It is the client's responsibility to update any descriptors that map the memory block with the new linear address after resizing the block.

- This function returns an error if the client attempts to resize a memory block to zero bytes.

Int 31H Function 0504H [1.0]
Allocate Linear Memory Block

Allocates a block of page-aligned linear address space. The base address of the block may be specified by the client, and pages within the block may be committed or uncommitted.

Call With:

AX = 0504H
EBX = desired page-aligned linear address of memory block, or zero if linear address unspecified
ECX = size of block (bytes, must be nonzero)
EDX = flags

Bit	*Significance*
0	0 = create uncommitted pages 1 = create committed pages
1-31	reserved, should be zero

Returns:

if function successful

Carry flag = clear
EBX = linear address of memory block
ESI = handle for memory block

if function unsuccessful

Carry flag = set
AX = error code

8001H	unsupported function (16-bit host)
8012H	linear memory unavailable
8013H	physical memory unavailable
8014H	backing store unavailable
8016H	handle unavailable
8021H	invalid value (ECX = 0)
8025H	invalid linear address (EBX not page aligned)

Notes:

- A DPMI 1.0 host that is 16-bit only will not support this function.

- A 16-bit client of a 32-bit DPMI 1.0 host can use this function.

- The allocated block is always page-aligned. If a specific linear address is not requested (EBX = 0), the DPMI host allocates the memory block at any available page-aligned linear address. If a specific linear address *is* requested (EBX nonzero), the host either allocates the block at the specified address or returns error code 8012H (linear memory unavailable).

- Int 31H Function 0501H, which can also be used to allocate linear memory blocks, does not necessarily page-align its blocks and does not have the ability to create uncommitted pages or allocate a block at a specific linear address.

Int 31H Function 0505H [1.0]
Resize Linear Memory Block

Changes the size of a memory block that was previously allocated with the Allocate Linear Memory Block function (Int 31H Function 0504H).

Call With:

AX	= 0505H	
ESI	= memory block handle	
ECX	= new block size (bytes, must be nonzero)	
EDX	= flags	
	Bit	*Significance*
	0	0 = create uncommitted pages 1 = create committed pages
	1	0 = do not update segment descriptors 1 = segment descriptor update required
	2-31	reserved, must be zero

and, if bit 1 of EDX is set (1):

ES:EBX	= selector:offset of a buffer containing an array of selectors, 1 word (16 bits) per selector
EDI	= count of selectors in array

Returns:

if function successful

Carry flag	= clear
EBX	= new linear base address of memory block
ESI	= new handle for memory block

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8001H	unsupported function (16-bit host)
	8012H	linear memory unavailable
	8013H	physical memory unavailable
	8014H	backing store unavailable
	8016H	handle unavailable
	8021H	invalid value (ECX=0)
	8023H	invalid handle (in ESI)

Notes:

o A DPMI 1.0 host that is 16-bit only will not support this function.

o A 16-bit client of a 32-bit DPMI 1.0 host can use this function.

o After this function returns, the previous handle for the memory block is invalid and should not be used.

- If this function fails, the block's size and base address are always unmodified.

- If the size of the block is increased, the new pages are committed or uncommitted according to the value of bit 0 of EDX, and the block's linear base address may change. If the size of the block is decreased, pages at the end of the block are freed, and the block's base address is always unchanged.

- If the block's linear base address is changed by this function, and the function was called with bit 1 of EDX set (1), the DPMI host updates the descriptors for each of the segments in the update list which fall within the memory block. Descriptors for segments which do not fall within the memory block are not modified. Expand-up segments fall within the memory block if the segment base is within the block. Expand-down segments fall within the memory block if the (segment base + the limit - 1) is within the block. In either case, the segment base is modified by the distance the block moves, and the segment limit is not changed. The moving of the memory block and the updating of descriptors is performed atomically; i.e. the host will not deliver any hardware interrupts to the client during the update.

- Int 31H Function 0503H, which also resizes linear memory blocks, does not necessarily page-align blocks and cannot create uncommitted pages or update descriptors.

Int 31H Function 0506H [1.0]
Get Page Attributes

Returns the attributes of one or more pages within a linear memory block previously allocated with Int 31H Function 0504H.

Call With:

AX = 0506H
ESI = memory block handle
EBX = base offset in memory block of page (or of first page, if requesting attributes for multiple pages)
ECX = number of pages
ES:EDX = selector:offset of a buffer to receive page attributes, 1 word (16-bits) per page (see Note)

Returns:

if function successful
Carry flag = clear

and buffer at ES:EDX filled in with page attributes (see Note)

if function unsuccessful
Carry flag = set
AX = error code

8001H	unsupported function (16-bit host)
8023H	invalid handle (in ESI)
8025H	invalid linear address (Specified range is not within specified block)

Notes:

o A DPMI 1.0 host that is 16-bit only will not support this function.

o A 16-bit client of a 32-bit DPMI 1.0 host can use this function.

o If EBX is not aligned, it will be rounded down to the next lower page boundary.

o The specified buffer is filled in by the DPMI host with the attributes of the requested pages, 1 word (16-bits) per page, in the following format:

Bits	*Significance*	
0-2	page type (0-7)	
	Value	*Meaning*
	0	uncommitted page
	1	committed page
	2	mapped page
	3-7	currently unused
3	0 = page is read-only	
	1 = page is read/write	
4	0 = accessed/dirty bits not available for this page	
	1 = accessed/dirty bits are supplied for this page in bits 5-6	

5	0 = page has not been accessed (if bit 4=1) 1 = page has been accessed (if bit 4=1)
6	0 = page has not been modified (if bit 4=1) 1 = page has been modified (if bit 4=1)
7-15	reserved, currently zero

- Mapped pages can only occur in memory blocks under DPMI hosts that support the Device Mapping capability or the Conventional Memory Mapping capability. See Int 31H Functions 0401H, 0508H, and 0509H.

- The dirty and accessed bits are only supplied if the DPMI host supports the Page Accessed/Dirty capability. DPMI hosts that support this capability are required to return dirty and accessed bits for all committed pages and for mapped pages created with the Map Conventional Memory call (Int 31H Function 0509H). However, dirty and accessed bits may not be returned for individual mapped pages created with the Map Device call (Int 31H Function 0508H) if the host is using page table entries (PTEs) to virtualize the device.

Int 31H Function 0507H [1.0]
Set Page Attributes

Sets the attributes of one or more pages within a linear memory block previously allocated with Int 31H Function 0504H. This function can be used to change a committed page or a mapped page to an uncommitted page, change an uncommitted page or a mapped page to a committed page, or modify the read/write bit and optionally the accessed and dirty bits on a committed or mapped page.

Call With:

AX	= 0507H
ESI	= memory block handle
EBX	= offset within memory block of page(s) whose attributes are to be modified
ECX	= number of pages
ES:EDX	= selector:offset of a buffer containing page attributes, 1 word (16-bits) per page (see Note)

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set
AX	= error code

8001H	unsupported function (16-bit host)
8002H	invalid state (page in wrong state for request)
8013H	physical memory unavailable
8014H	backing store unavailable
8021H	invalid value (illegal request in bits 0-2 of one or more page attribute words)
8023H	invalid handle (in ESI)
8025H	invalid linear address (specified range is not within specified block)

ECX	= number of pages that have been set

Notes:

o A DPMI 1.0 host that is 16-bit only will not support this function.

o A 16-bit client of a 32-bit DPMI 1.0 host can use this function.

o If EBX is not aligned, it will be rounded down to the next lower page boundary.

o An uncommitted page can be created from:
- a committed page, by releasing the physical memory or backing store allocated to the page;
- a mapped page, by marking it uncommitted; or
  an uncommitted page, by doing nothing.

- A committed page can be created from:
  - an uncommitted page or mapped page, by allocating physical memory or backing store (with undefined, or zero-filled contents) for the page; or
  - a committed page, by doing nothing (page contents unmodified).

- The attribute word (16-bits) specified for a page has the following format (bits 3-6 are only relevant if page is being created committed or its attributes are being modified, i.e. the value in bits 0-2 of the page attribute is 1 or 3):

Bits	*Significance*	
0-2	page type (0-7)	
	Value	*Meaning*
	0	create page uncommitted
	1	create page committed
	2	not allowed
	3	modify attributes without changing page type
	4-7	not allowed
3	0 = page is read-only 1 = page is read/write	
4	0 = don't modify accessed/dirty bits for page 1 = set accessed/dirty bits as specified in bits 5-6	
5	0 = mark page as not accessed (if bit 4=1) 1 = mark page as accessed (if bit 4=1)	
6	0 = mark page as not dirty (if bit 4=1) 1 = mark page as dirty (if bit 4=1)	
7-15	reserved, should be zero	

- This function, and the optional Map Device and Map Conventional Memory functions (Int 31H Functions 0508H and 0509H), are the only means of changing the type of a page within an existing memory block.

- The page read/write bit, and optionally the accessed and dirty bits, can be modified on an existing committed or mapped page, or on a committed page when it is initially created from an uncommitted page or a mapped page. However, the accessed and dirty bits are ignored if the host does not support the Page Accessed/Dirty capability. See Int 31H Function 0401H.

- Visible page faults (page faults that can be serviced by a client-installed exception handler) can only occur for uncommitted pages or read-only pages (for definitions of transparent page fault and visible page fault, see Appendix A: Glossary).

Int 31H Function 0508H [Optional] [1.0]
Map Device in Memory Block

Maps the physical addresses assigned to a device onto the linear addresses of a memory block previously allocated with Int 31H Function 0504H.

Call With:

AX	= 0508H
ESI	= memory block handle
EBX	= offset within memory block of page(s) to be mapped (must be page-aligned)
ECX	= number of pages to map
EDX	= physical address of device (must be page-aligned)

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8001H	unsupported function (Device Mapping Capability not supported)
	8003H	system integrity (invalid device address)
	8023H	invalid handle (in ESI)
	8025H	invalid linear address (specified range is not within specified block or EBX/EDX is not page-aligned)

Notes:

- o 16-bit DPMI hosts will not support this function. A 16-bit client of a 32-bit DPMI 1.0 host can use this function.

- o Support of this call by 32-bit DPMI hosts is optional. Application programs or DOS Extenders which require this call in order to run are *not* DPMI Compliant.

- o Any committed or mapped pages resided in the linear address range that is being mapped into will be uncommitted or unmapped automatically by the host.

- o All pages created by this call have the mapped bit (bit 2) set in the attributes returned by the Get Page Attributes function (Int 31H Function 0506H).

- o This function differs from the Create Physical Address Mapping function (Int 31H Function 0800H) in that this function supports mapping of physical devices within an existing memory block, rather than at an arbitrary linear address. Use of an existing memory block gives 32-bit programs the ability to access physical devices with NEAR pointers, which is often highly desirable for performance reasons.

- o Unlike Int 31H Function 0800H, this function allows mapping of addresses below 1 MB that do not lie within RAM available for use by programs; e.g. this function can be used to map the refresh buffers of IBM-compatible display adapters.

- o If the DPMI host is not virtualizing the device, it must disable any memory caching on the mapped pages; in particular, on the 486 or later, the PCD (page cache disable) bit must be set in the page table entries.

- o DPMI hosts that do not virtualize physical devices can support this function by creating page table entries that map the physical device. The page table entries must be marked as mapped so that the host knows not to attempt freeing of physical memory for the pages when the memory block is freed.

- o DPMI hosts are allowed to support this function for some physical devices and not for others, because mapping of virtualized devices requires page aliasing in the host - a complex task. DPMI hosts with partial support for this function may fail the function call on virtualized devices (such as displays), and allow the call on non-virtualized devices (such as the Weitek coprocessors). Allowing the client to map a physical device so that it can be accessed with NEAR references, for example, may help the client achieve considerably better performance.

Int 31H Function 0509H [Optional] [1.0]
Map Conventional Memory in Memory Block

Aliases linear addresses below the 1 MB boundary onto the linear addresses of a memory block previously allocated with Int 31H Function 0504H.

Call With:

AX	= 0509H
ESI	= memory block handle
EBX	= offset within memory block of page(s) to be mapped (must be page-aligned)
ECX	= number of pages to map
EDX	= linear address of conventional memory (must be page-aligned)

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8001H	unsupported function (Conventional Memory Mapping Capability not supported)
	8003H	system integrity (invalid conventional memory address)
	8023H	invalid handle (in ESI)
	8025H	invalid linear address (specified range is not within specified block, or EBX/EDX is not page aligned)

Notes:

- 16-bit DPMI hosts will not support this function. A 16-bit client of a 32-bit DPMI 1.0 host can use this function.

- Support of this call by 32-bit DPMI hosts is optional. Application programs or DOS Extenders which require this call in order to run are *not* DPMI Compliant.

- Any committed or mapped pages resided in the linear address range that is being mapped into will be uncommitted or unmapped automatically by the host.

- A client may only map conventional memory that it already owns; i.e. memory which the client previously allocated with Int 31H Function 0100H or by calling DOS's Int 21H Function 48H directly via the translation services.

- All pages created by this call have the mapped bit (bit 2) set in the attributes returned by the Get Page Attributes function (Int 31H Function 0506H).

- DPMI hosts that do not implement virtual memory can support this function by simply copying page table entries. The entries must be marked as mapped so that the host knows not to free up those physical pages when the memory block is freed.

- DPMI hosts that provide virtual memory must implement some form of page aliasing in order to support this function.

o The function can provide a large contiguous memory space without virtual memory support.

- Implementors of DPMI hosts which do not provide virtual memory are encouraged to support this function. Without this function, conventional memory may be inaccessible to a 32-bit nonsegmented client, because the client may need contiguous linear memory for its code and data. 32-bit clients can always guarantee that conventional memory is not wasted with the following strategy:

- Call DOS to allocate any free conventional memory

- If the DPMI host supports virtual memory, call the Mark Real Mode Region Pageable function (Int 31H Function 0602H) to ensure that the host has not locked down conventional memory.

- If the host does not support virtual memory but supports the Map Conventional Memory function (Int 31H Function 0509H), allocate a memory block with uncommitted pages, then use Function 0509H to make the physical memory allocated below 640 KB addressable in the memory block, and therefore useable by the 32-bit application program.

Int 31H Function 050AH [1.0]
Get Memory Block Size and Base

Returns the size of a memory block that was previously allocated with Int 31H Function 0501H or 0504H.

Call With:

AX	= 050AH
SI:DI	= memory block handle

Returns:

if function successful

Carry flag	= clear
SI:DI	= size of memory block (bytes)
BX:CX	= base address of memory block *if function unsuccessful*
Carry flag	= set
AX	= error code 8023H invalid handle

Int 31H Function 050BH [1.0]
Get Memory Information

Returns information about available physical and virtual memory. Since DPMI clients will often run in multitasking environments, some of information related to shared resources returned by this function should only be considered as advisory.

Call With:

AX	= 050BH
ES:(E)DI	= selector:offset of 128-byte buffer

Returns:

if function successful
Carry flag = clear (this function always succeeds in DPMI 1.0)

and the buffer pointed to by ES:(E)DI is filled in with the following information:

Offset	*Length*	*Contents*
00H	4	Total allocated bytes of physical memory controlled by DPMI host
04H	4	Total allocated bytes of virtual memory controlled by DPMI host
08H	4	Total available bytes of virtual memory controlled by DPMI host
0CH	4	Total allocated bytes of virtual memory for this virtual machine
10H	4	Total available bytes of virtual memory for this virtual machine
14H	4	Total allocated bytes of virtual memory for this client
18H	4	Total available bytes of virtual memory for this client
1CH	4	Total locked bytes of memory for this client
20H	4	Maximum locked bytes of memory for this client
24H	4	Highest linear address available to this client
28H	4	Size in bytes of largest available free memory block
2CH	4	Size of minimum allocation unit in bytes
30H	4	Size of the allocation alignment unit in bytes
34H	4CH	Reserved, currently zero

if function unsuccessful
Carry flag = set (this function always fails in DPMI 0.9)

Notes:

o DPMI 1.0 clients should use this function in preference to Int 31H Function 0500H.

o The "total available bytes" field of the data structure pointed by ES:(E)DI means the total bytes minus all of the allocated bytes.

Int 31H Function 0600H [0.9]
Lock Linear Region

Locks the specified linear address range.

Call With:

AX	= 0600H
BX:CX	= starting linear address of memory to lock
SI:DI	= size of region to lock (bytes)

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8013H	physical memory unavailable
	8017H	lock count exceeded
	8025H	invalid linear address (unallocated pages)

Notes:

- If the function returns an error, none of the memory has been locked.

- If the specified region overlaps part of a page at the beginning or end of the region, the page(s) will be locked.

- This function may be called more than once for a given page; the DPMI host maintains a lock count for each page.

- This function is ignored by DPMI implementations that do not support virtual memory; the function will return the Carry flag clear to indicate success, but has no other effect. DPMI hosts which support virtual memory may also choose to ignore this function, but such hosts must be able to handle page faults transparently at arbitrary points during a client's execution, including within interrupt and exception handlers.

Int 31H Function 0601H [0.9]
Unlock Linear Region

Unlocks a linear address range that was previously locked using the Lock Linear Region function (Int 31H Function 0600H).

Call With:

AX	= 0601H
BX:CX	= starting linear address of memory to unlock
SI:DI	= size of region to unlock (bytes)

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8002H	invalid state (page not locked)
	8025H	invalid linear address (unallocated pages)

Notes:

- o If the function returns an error, none of the memory has been unlocked.

- o If the specified region overlaps part of a page at the beginning or end of the region, the page(s) will be unlocked.

- o A lock count is maintained for each locked page; the page is not unlocked until the lock count is decremented to zero (i.e. the number of Lock Region Int 31H Function 0600H calls has been balanced by the same number of Unlock Region Int 31H Function 0601H calls).

- o This function is ignored by DPMI implementations that do not support virtual memory; the function will return the Carry flag clear to indicate success, but has no other effect. DPMI hosts which support virtual memory may also choose to ignore this function, but such hosts must be able to handle page faults transparently at arbitrary points during a client's execution, including within interrupt and exception handlers.

Int 31H Function 0602H [0.9]
Mark Real Mode Region as Pageable

Advises the DPMI host that the specified memory below the 1 MB boundary may be paged to disk.

Call With:

AX	= 0602H
BX:CX	= starting linear address of memory to mark as pageable
SI:DI	= size of region to be marked (bytes)

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set		
AX	= error code		
		8002H	invalid state (region already marked as pageable)
		8025H	invalid linear address (region is above 1 MB boundary)

Notes:

- If the function returns an error, none of the memory has been marked as pageable.

- If the specified region overlaps part of a page at the beginning or end of the region, the page(s) will not be marked as pageable.

- Pageability information for a real mode region is maintained as a binary state, not a count. Therefore, multiple calls to this function for the same region have no effect.

- For compatibility with DPMI version 0.9 hosts, a client must call the Relock Real Mode Region function (Int 31H Function 0603H) to relock the memory region before terminating. Memory that remains unlocked after the client has terminated could result in fatal page faults when another program is executed in the same address space. DPMI 1.0 hosts automatically relock real mode memory at client termination.

- Under some DPMI hosts, all conventional memory may be locked by default. If a protected mode program is using memory in the first megabyte of address space, it is recommended that this function be used to turn off automatic page locking for regions of memory that will not be touched at interrupt time.

- The client must not mark memory as pageable in regions that it does not own; i.e. it may only mark as pageable memory that it has previously allocated with Int 31H Function 0100H or by a direct call to DOS via the translation functions. For example, marking all free DOS memory as pageable under some DPMI hosts could cause a page fault to occur while inside of DOS, resulting in a crash. Also, a client should not mark the DPMI host data area as pageable.

- Note that address space marked as pageable by this function can still be locked using the Lock Linear Region function (Int 31H Function 0600H). This function is just an advisory service to allow memory that does not need to be locked to be paged out; it disables any automatic locking of real mode memory performed by the DPMI host.

o This function is ignored by DPMI implementations that do not support virtual memory; the function will return the Carry flag clear to indicate success, but has no other effect. DPMI hosts which support virtual memory may also choose to ignore this function, but such hosts must be able to handle page faults transparently at arbitrary points during a client's execution, including within interrupt and exception handlers.

Int 31H Function 0603H [0.9]
Relock Real Mode Region

Relocks a memory region that was previously declared as pageable with the Mark Real Mode Region as Pageable function (Int 31H Function 0602H).

Call With:

AX	= 0603H
BX:CX	= starting linear address of memory to relock
SI:DI	= size of region to relock (bytes)

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8002H	invalid state (region not marked as pageable)
	8013H	physical memory unavailable
	8025H	invalid linear address (region is above 1 MB boundary)

Notes:

- If the function returns an error, none of the memory has been relocked.

- If the specified region overlaps part of a page at the beginning or end of the region, the page(s) will not be relocked.

- This function is ignored by DPMI implementations that do not support virtual memory; the function will return the Carry flag clear to indicate success, but has no other effect. DPMI hosts which support virtual memory may also choose to ignore this function, but such hosts must be able to handle page faults transparently at arbitrary points during a client's execution, including within interrupt and exception handlers.

- If Function 0602H is implemented as a "no-operation" on a particular DPMI host, this function will likewise do nothing. In other words, this function should not be used to lock memory, but only to restore the default state of the host's conventional memory locking.

Int 31H Function 0604H [0.9]
Get Page Size

Returns the size of a single memory page in bytes.

Call With:

AX	= 0604H

Returns:

if function successful

Carry flag	= clear
BX:CX	= page size in bytes

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8001H	unsupported function (16-bit host)

Int 31H Function 0700H [0.9]
Reserved

Function 0700H is reserved for historical reasons and should not be called.

Int 31H Function 0701H [0.9]
Reserved

Function 0701H is reserved for historical reasons and should not be called.

Int 31H Function 0702H [0.9]
Mark Page as Demand Paging Candidate

Notifies the DPMI host that a range of pages may be placed at the head of the page-out candidate list, forcing these pages to be replaced ahead of other pages even if the memory has been accessed recently. The contents of the pages will be preserved.

Call With:

AX	= 0702H
BX:CX	= starting linear address of pages to mark as paging candidates
SI:DI	= size of region to mark (bytes)

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8025H	invalid linear address (range unallocated)

Notes:

o This function does not force the pages to be swapped to disk immediately and should be treated as advisory only.

o This function will always succeed on hosts that do not implement demand-paged virtual memory.

o Partial pages will not be marked.

o This function is useful, for example, if a client 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 so that the physical memory it occupies can be used for other purposes.

Int 31H Function 0703H [0.9]
Discard Page Contents

Discards the entire contents of a given linear memory range. This function is used when a memory object (such as a data structure) that occupies a given area of memory is no longer needed, so that the area will not be paged to disk unnecessarily. The contents of the discarded region will be undefined.

Call With:

AX	= 0703h
BX:CX	= starting linear address of pages to discard
SI:DI	= size of region to discard (bytes)

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8025H	invalid linear address (range unallocated)

Notes:

o Partial pages and locked pages will not be discarded.

Int 31H Function 0800H [0.9]
Physical Address Mapping

Converts a physical address into a linear address. This function allows device drivers running under DPMI hosts which use paging to reach physical memory that is associated with their devices above the 1 MB boundary Examples of such devices are the Weitek numeric coprocessor (usually mapped at 3 GB), buffers that hold scanner bit maps, and high-end displays that can be configured to make display memory appear in extended memory.

Call With:

AX	= 0800H
BX:CX	= physical address of memory
SI:DI	= size of region to map (bytes)

Returns:

if function successful

Carry flag	= clear
BX:CX	= linear address that can be used to access the physical memory

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8003H	system integrity (DPMI host memory region)
	8021H	invalid value (address is below 1 MB boundary)

Notes:

- It is the caller's responsibility to allocate and initialize a descriptor for access to the memory.

- This function should only be used by clients that absolutely require direct access to a memory mapped device at physical addresses above 1 MB. Clients should not use this function to access memory below the 1 MB boundary (the real mode addressable region). See also Int 31H Functions 0002H, 0508H, and 0509H.

- When this function is called, the DPMI host either creates page table entries that directly map the physical addresses requested and returns the linear address of the created page table entries, or else just returns the linear address of the memory region that is already used to map the requested device. For example, if the client attempts to map a Weitek coprocessor and the host already has a linear region set up to map the Weitek chip and virtualize it, it would simply return the linear address of the existing region. If the host does not virtualize the Weitek chip, it would create 16 page table entries that map the 64 KB Weitek address space and return a linear address corresponding to the new page table entries.

- If the host is not virtualizing the device, it must disable any memory caching on the mapped pages; in particular, on the 80486 the host must set the PCD (page cache disable) bit in the page table entries.

- The host is permitted to fail any memory mapping call. However, the host should support this function whenever possible, to achieve compatibility with application programs that use memory-mapped devices of which the host is not aware. Useful guidelines are that the host should fail any attempt to map addresses below 1 MB, or addresses which the host considers to be general-purpose RAM memory. Attempts to map *any other* physical address should succeed, since the host should either (a) already know about the device and be able to return a linear address used to access the device, or (b) assume the program is attempting to map a legitimate device of which the host has no knowledge.

o Programs and device drivers which need to perform DMA I/O to physical addresses in a virtualized hardware environment should use the Virtual DMA Services (see the Glossary entry for the Virtual DMA Services Specification). Also see page 10 of the DPMI execution environment section.

Int 31H Function 0801H [1.0]
Free Physical Address Mapping

Releases a mapping of physical to linear addresses that was previously obtained with the Physical Address Mapping function (Int 31H Function 0800H).

Call With:

AX	= 0801H
BX:CX	= linear address returned by physical address mapping call

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8025H	invalid linear address

Notes:

- The client should call this function when it is finished using a device previously mapped to linear addresses with the Physical Address Mapping function (Int 31H Function 0800H).

Int 31H Function 0900H [0.9]
Get and Disable Virtual Interrupt State

Disables the virtual interrupt flag and returns the previous state of the virtual interrupt flag.

Call With:

AX	= 0900H

Returns:

Virtual interrupts disabled

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

Notes:

o AH is not changed by this function. Therefore, the previous state can be restored by simply executing another Int 31H. See Int 31H Function 0901H.

o A client that does not need to know the prior interrupt state can execute the `CLI` instruction rather than calling this function. The instruction may be trapped by the host and should be assumed to be very slow.

Int 31H Function 0901H [0.9]
Get and Enable Virtual Interrupt State

Enables the virtual interrupt flag and returns the previous state of the virtual interrupt flag.

Call With:

AX	= 0901H

Returns:

Virtual interrupts enabled

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

Notes:

- o AH is not changed by this function. Therefore, the previous state can be restored by simply executing another Int 31H. See Int 31H Function 0900H.

- o A client that does not need to know the prior interrupt state can execute the `STI` instruction rather than calling this function. The instruction may be trapped by the host and should be assumed to be very slow.

Int 31H Function 0902H [0.9]
Get Virtual Interrupt State

Returns the current state of the virtual interrupt flag.

Call With:

AX	= 0902H

Returns:

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

Notes:

o This function should be used in preference to the PUSHF instruction to examine the interrupt flag, because the PUSHF instruction returns the physical interrupt flag rather than the virtualized (per-client) interrupt flag. On some DPMI hosts, the physical interrupt flag will always be enabled, even when hardware interrupts are not being passed through to the client.

Int 31H Function 0A00H [0.9]
Get Vendor-Specific API Entry Point

Returns an address which can be called to use host-specific extensions to the standard set of DPMI functions. DPMI 1.0 clients should avoid use of this function (see Note).

Call With:

AX	= 0A00H
DS:(E)SI	= selector:offset of ASCIIZ (null-terminated) string which identifies the DPMI host vendor

Returns:

if function successful

Carry flag	= clear
ES:(E)DI	= selector:offset of extended API entry point

and DS, FS, GS, EAX, EBX, ECX, EDX, ESI, and EBP may be modified.

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8001H	unsupported function (extension not found)

Notes:

- The null-terminated string specifies the host-specific vendor name or some other unique identifier to obtain a specific extension entry point. The string comparison used to look up the API entry point is case-sensitive.
- Clients must use a `FAR CALL` to reach the extended API entry point.
- All extended API parameters are specified by the vendor.
- DPMI 1.0 clients should use Int 2FH Function 168AH in preference to this function. DPMI 1.0 hosts support this function solely for backward compatibility with DPMI 0.9 clients.

Int 31H Function 0B00H [0.9]
Set Debug Watchpoint

Sets a debug watchpoint at the specified linear address.

Call With:

AX	= 0B00H
BX:CX	= linear address of watchpoint
DL	= size of watchpoint (1, 2, or 4 bytes)
DH	= type of watchpoint 0 = execute 1 = write 2 = read/write

Returns:

if function successful

Carry flag	= clear
BX	= watchpoint handle

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8016H	too many breakpoints
	8021H	invalid value (in DL or DH)
	8025H	invalid linear address (linear address not mapped or alignment error)

Notes:

- o Under DPMI 1.0, the handle will be in the range 0-14. Under DPMI 0.9, the handle range is not limited.

- o The watchpoint handle corresponds to the bit number in the Virtual DR6 returned in the exception frame (see Int 31H Function 0212H and page 18).

Int 31H Function 0B01H [0.9]
Clear Debug Watchpoint

Clears a debug watchpoint that was previously set using the Set Debug Watchpoint function (Int 31H Function 0B00H), and releases the watchpoint handle.

Call With:

AX	= 0B01H
BX	= watchpoint handle

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8023H	invalid handle

Int 31H Function 0B02H [0.9]
Get State of Debug Watchpoint

Returns the state of a debug watchpoint that was previously set using the Set Debug Watchpoint function (Int 31H Function 0B00H).

Call With:

AX	= 0B02H
BX	= watchpoint handle

Returns:

if function successful

Carry flag	= clear
AX	= watchpoint status

Bit	*Significance*
0	0 = watchpoint has not been encountered 1 = watchpoint has been encountered
1-15	reserved

if function unsuccessful

Carry flag	= set
AX	= error code

8023H	invalid handle

Notes:

o The client can use Int 31H Function 0B03H to clear the watchpoint state without releasing the watchpoint handle.

Int 31H Function 0B03H [0.9]
Reset Debug Watchpoint

Resets the state of a previously defined debug watchpoint; i.e. a subsequent call to Int 31H Function 0B02H will indicate that the debug watchpoint has not been encountered.

Call With:

AX	= 0B03H
BX	= watchpoint handle

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8023H	invalid handle

Int 31H Function 0C00H [1.0]
Install Resident Service Provider Callback

Protected mode resident service providers (protected mode TSRs) can provide services to 16-bit DPMI programs, 32-bit DPMI programs, or both. A resident service provider uses this function to request notification from the host whenever another DPMI program in the same virtual machine is loaded or terminated.

Call With:

AX = 0C00H
ES:(E)DI = selector:offset of 40-byte buffer with the following structure:

Offset	*Length*	*Contents*
00H	8	Descriptor for 16-bit data segment
08H	8	Descriptor for 16-bit code segment
10H	2	Offset of 16-bit callback procedure
12H	2	Reserved
14H	8	Descriptor for 32-bit data segment
1CH	8	Descriptor for 32-bit code segment
24H	4	Offset of 32-bit callback procedure

Returns:

if function successful
Carry flag = clear

if function unsuccessful
Carry flag = set
AX = error code

8021H	invalid value (access rights/type bytes invalid, or offset outside segment limits)
8025H	invalid linear address (descriptor references a linear address range outside that allowed for DPMI clients)
8015H	callback unavailable (host unable to allocate resources for resident handler initialization callback)

Notes:

- A DPMI client that uses this function declares its intent to provide resident protected mode services. The client must subsequently terminate and stay resident using Int 31H Function 0C01H. DPMI clients which intend to stay resident only to provide services to real mode programs should not use this function.

- The data structure provides room for a data descriptor, a code descriptor, and an offset for both 16-bit and 32-bit protected modes. The client can conveniently initialize the descriptor fields to valid values by fetching copies of its current code and data descriptors with Int 31H Function 000BH.

- If only one mode is supported by the resident service provider, then the code descriptor for the unsupported mode should be initialized to zero.

- This function is called on the locked protected mode stack.

- For further details on programming of resident service providers, see page 41.

Int 31H Function 0C01H [1.0]
Terminate and Stay Resident

A resident service provider uses this function after its initialization to terminate execution while leaving its protected mode memory (and optionally some real mode memory) allocated.

Call With:

AX	= 0C01H
BL	= return code
DX	= number of paragraphs (16-byte blocks) of DOS memory to reserve

Returns:

Nothing (this call never returns)

Notes:

o This function should only be used by DPMI clients which only provide resident services to other DPMI protected mode clients. If the objective is only to provide resident services to real mode programs, the client should use the DPMI translation service Int 31H Function 0300H to invoke DOS's Int 21H Function 31H directly.

o The value in DX only specifies the size of DOS allocated memory to reserve. Any protected mode memory owned by the program remains allocated unless it is explicitly released before executing this function. Note that the value in DX must either be 0 or a minimum of 6. If DX is 0, the DPMI host executes a DOS real mode terminate function (Int 21H Function 4CH), and no real mode memory is reserved. If DX is nonzero, the DPMI host requests the DOS real mode terminate-and-stay-resident function (Int 21H Function 31H).

o If the client has not made a prior call to Int 31H Function 0C00H, the client will simply be terminated.

o For further details on programming of resident service providers, see page 41.

Int 31H Function 0D00H [1.0]
Allocate Shared Memory

Allocates a memory block that may be shared by DPMI clients.

Call With:

AX = 0D00H
ES:(E)DI = selector:offset of shared memory allocation request structure in the following format:

Offset	*Length*	*Contents*
00H	4	Requested length of shared memory block (set by client, may be zero)
04H	4	Length actually allocated (set by host)
08H	4	Shared memory handle (set by host)
0CH	4	Linear address of shared memory block (set by host)
10H	6	offset32:selector of ASCIIZ (null-terminated ASCII) name for shared memory block (set by client)
16H	2	Reserved
18H	4	Reserved, must be zero

Returns:

if function successful
Carry flag = clear

and the request structure fields at offsets 04H, 08H, and 0CH updated by host

if function unsuccessful
Carry flag = set
AX = error code

8012H	linear memory unavailable
8013H	physical memory unavailable
8014H	backing store unavailable
8016H	handle unavailable
8021H	invalid value (name for the memory block is too long)

and the request structure fields at offsets 04H, 08H and 0CH unmodified by host

Notes:

- For 16-bit programs, the high word of the offset32 for the ASCIIZ name must be zero.

- The maximum length of the shared memory block name is 128 characters, including the terminal null character.

- The linear address provided by the host is guaranteed to be the same for all clients in all virtual machines using a shared memory block. The client must establish addressability for the block by allocating and initializing a descriptor with separate function calls.

- No assumptions should be made about handle values. Successive allocations of the same shared memory block by the same client may return distinct handles; the client is responsible for tracking and individually deallocating each handle.

- The first client that allocates a shared memory block determines its size; the length requested and the length actually allocated will always be equal, if the allocation succeeds at all. Subsequent allocations by the same or different clients that specify the same or a different size will succeed, but the size of the block will remain unchanged. The actual size of the block is always returned to the client at offset 4 in the shared memory allocation request structure.

- Allocation of zero-length shared memory blocks is explicitly allowed. The handle of a zero-length block can be used with the serialization functions (Int 31H Functions 0D02H and 0D03H) as a semaphore for inter-client communication. The linear address that is returned at offset 0CH in the data structure for zero-length blocks is undefined, and any reference to it may produce a page fault.

- The first paragraph (16 bytes) of the shared memory block (or the entire shared block, if smaller than 16 bytes) will always be initialized to zero on the first allocation and can be used by clients as an "area initialized" indicator. For example, a shared memory block might be used by a suite of cooperating client programs to hold a table of static data or a subroutine library. The first client to allocate the shared memory block can obtain exclusive ownership of the block with Int 31H Function 0D02H, load the necessary data or code into the block from disk, set the first 16 bytes of the block to a nonzero value, and finally release its ownership of the block with Int 31H Function 0D03H. Other clients that allocate the shared memory block can check the "area initialized" indicator and know that the desired code or data is already present in memory.

- Shared memory block allocations and serializations are tracked by the host on a per client basis. All shared memory allocations for a client are freed by the host when the client terminates.

Int 31H Function 0D01H [1.0]
Free Shared Memory

Deallocates a shared memory block.

Call With:

AX	= 0D01H
SI:DI	= handle of shared memory block to free

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set	
AX	= error code	
	8023H	invalid handle

Notes:

- o The shared memory handle becomes invalid after the shared memory block is deallocated, and should not be used in any other function call (such as serialization).

- o The host maintains virtual machine use counts and a global use count for each shared memory block. A virtual machine use count is the number of allocation calls (Int 31H Function 0D00H) that have been issued by a particular virtual machine for the shared block, while the global use count corresponds to the number of virtual machines which have access to the block. When a virtual machine use count reaches zero, the clients in that virtual machine no longer have addressability to the shared memory block; when the global use count reaches zero, the shared memory block is destroyed by the host.

- o It is the client's responsibility to free any descriptors that it has allocated to map the shared memory block.

- o Applications should not depend on this function to release a previous successful serialization for the same shared memory block. Serialization is only released by this function when the virtual machine use count goes to 0 (i.e., the client no longer has access to the shared memory block).

Int 31H Function 0D02H [1.0]
Serialize on Shared Memory

Requests serialization of a shared memory block. Successful serialization symbolizes ownership or right of access to a block, and can be used by DPMI clients to synchronize the inspection or modification of a shared memory block.

Call With:

AX	= 0D02H		
SI:DI	= shared memory block handle		
DX	= option flags		
		Bit	*Significance*
		0	0 = suspend client until serialization available 1 = return immediately with error if serialization not available
		1	0 = exclusive serialization requested 1 = shared serialization requested
		2-15	reserved, must be zero

Returns:

if function successful

Carry flag	= clear

if function unsuccessful

Carry flag	= set		
AX	= error code		
		8004H	deadlock (host detected a deadlock situation)
		8005H	request cancelled with Int 31H Function 0D03H
		8017H	lock count exceeded
		8018H	exclusive serialization already owned by another client
		8019H	shared serialization already owned by another client
		8023H	invalid handle

Notes:

o For each client, the DPMI host maintains four different local (virtual machine) serialization counts (exclusive, shared, pending shared, and pending exclusive) for each shared memory block, as well as a global serialization count. The global serialization count is only updated when the sum of a virtual machine's exclusive and shared serialization counts goes from 0 to 1 (serialize) or 1 to 0 (free).

o A successful exclusive serialization blocks any serialization request (exclusive or shared) for the same block by another virtual machine. Exclusive serialization should be regarded as "ownership for writing," and should only be requested if the client intends to modify the block. A successful shared serialization will only block requests for exclusive serialization by another client. Shared serialization can be thought of as "read-only access," and should used when the client only intends to inspect the block and will not change its contents.

o Setting bit 0 of DX to 1 when the serialization request is made allows a client to determine whether a shared memory area is serialized without being suspended. Clients which "poll" for the availability of a resource in this manner are encouraged to yield the CPU with Int 2FH

Function 1680H at appropriate intervals.

- A serialization call that causes a client to be suspended can be canceled by a client interrupt service routine (such as a keyboard or timer interrupt handler) requesting the Free Serialization function (Int 31H Function 0D03H). In such cases, the original serialization request will return with the Carry flag set and AX = 8005H.

- A client that has been suspended while waiting for serialization of a shared memory block can still service interrupts. Some hosts may need to reissue the serialization request on behalf of the client after the interrupt service routine returns, but this event will be invisible to the client.

- Hosts are not required to detect deadlock. Clients that terminate and stay resident in order to function as resident service providers, executing in the context of other clients, must be careful to avoid deadlocks and incorrect sequencing in acquiring and releasing resources.

Int 31H Function 0D03H [1.0]
Free Serialization on Shared Memory

Releases a shared memory block serialization that was previously obtained with Int 31H Function 0D02H.

Call With:

AX	= 0D03H		
SI:DI	= shared memory block handle		
DX	= option flags		
		Bit	*Significance*
		0	0 = release exclusive serialization 1 = release shared serialization
		1	0 = don't free pending serialization 1 = free pending serialization (see Note)
		2-15	reserved, must be zero

Returns:

if function successful
Carry flag = clear

if function unsuccessful

Carry flag	= set		
AX	= error code		
		8002H	invalid state (client does not own a successful serialization of the specified type)
		8023H	invalid handle

Notes:

- For each client, the DPMI host maintains four different local (virtual machine) serialization counts (exclusive, shared, pending shared, and pending exclusive) for each shared memory block, as well as a global serialization count. The global serialization count is only updated when the sum of a virtual machine's exclusive and shared serialization counts goes from 0 to 1 (serialize) or 1 to 0 (free).

- A client's interrupt handler can call this function with bit 1 of DX set to cancel a serialization request that has suspended the main thread of execution of the same client. In such cases, the original serialization request will return with the Carry flag set and AX = 8005H.

Int 31H Function 0E00H [1.0]
Get Coprocessor Status

Returns information about whether or not a numeric coprocessor exists, the type of coprocessor available (if any), and whether or not the host or client is providing coprocessor emulation.

Call With:

AX = 0E00H

Returns:

if function successful
Carry flag = clear (this function always succeeds in DPMI 1.0)
AX = coprocessor status

Bit	*Significance*
0	MPv (MP bit in the virtual MSW/CR0) 0 = numeric coprocessor is disabled for this client 1 = numeric coprocessor is enabled for this client
1	EMv (EM bit in the virtual MSW/CR0) 0 = client is not emulating coprocessor instructions 1 = client is emulating coprocessor instructions
2	MPr (MP bit from the actual MSW/CR0) 0 = numeric coprocessor is not present 1 = numeric coprocessor is present
3	EMr (EM bit from the actual MSW/CR0) 0 = host is not emulating coprocessor instructions 1 = host is emulating coprocessor instructions
4-7	coprocessor type 00H = no coprocessor 02H = 80287 03H = 80387 04H = 80486 with numeric coprocessor 05H-0FH reserved for future numeric processors
8-15	not applicable

if function unsuccessful
Carry flag = set (this function always fails in DPMI 0.9)

Notes:

- If the real EM (EMr) bit is set, the host is supplying or is capable of supplying floating point emulation.

- If the MPv bit is not set, the host may not need to save the coprocessor state for this virtual machine to improve system performance.

- MPr bit setting should be consistent with the setting of coprocessor type information. Ignore MPr bit information if it is in conflict with the coprocessor type information.

- If the virtual EM (EMv) bit is set, the host delivers all coprocessor exceptions to the client, and the client is performing its own floating point emulation (whether or not a coprocessor is present or the host also has a floating point emulator). In other words, if the EMv bit is set, the host sets the EM bit in the real CR0 while the virtual machine is active, and reflects coprocessor not present faults (Int 7) to the virtual machine.

- A client can determine the CPU type with Int 31H Function 0400H, but a client should *not* draw any conclusions about the presence or absence of a coprocessor based on the CPU type alone.

Int 31H Function 0E01H [1.0]
Set Coprocessor Emulation

Enables or disables the numeric coprocessor for this virtual machine and the reflection of coprocessor exceptions to the client.

Call With:

AX	= 0E01H
BX	= coprocessor bits

Bit	*Significance*
0	new value of MPv bit for client's virtual CR0 0 = disable numeric coprocessor for this client 1 = enable numeric coprocessor for this client
1	new value of EMv bit for client's virtual CR0 0 = client will not supply coprocessor emulation 1 = client will supply coprocessor emulation
2-15	not applicable

Returns:

if function successful

Carry flag = clear

if function unsuccessful

Carry flag = set

AX = error code

8026H	invalid request (client requested disabling coprocessor on a processor which does not support this)

Notes:

- If the MPv bit is not set, the host may not need to save the coprocessor state for this virtual machine to improve system performance.

- If the virtual EM (EMv) bit is set, the host delivers all coprocessor exceptions to the client, so that the client can provide its own floating point emulation (whether or not a coprocessor is present or the host also has a floating point emulator). In other words, if the EMv bit is set, the host sets the EM bit in the real CR0 while the client is active, and reflects coprocessor not present faults (Int 7) to the client.

- Floating point emulation can be tested on a system with a numeric coprocessor by using this function to enable client handling of coprocessor exceptions and disable the coprocessor.

- The client should use Int 31H Function 0212H to register an exception handler for coprocessor not present faults (Int 7) prior to setting the EMv bit with this function.

- A client can determine the CPU type with Int 31H Function 0400H, and the presence or absence of a coprocessor with Int 31H Function 0E00H. The client should not draw any conclusions about the presence or absence of a coprocessor based on the CPU type alone.

Appendix A: Glossary

Client, DPMI

A program which uses the Int 2FH and Int 31H calls defined in the DPMI specification to obtain services from a DPMI host. Each time a real mode program calls the DPMI interface to request the initial switch to protected mode, a new DPMI client is created.

Committed page

A page, within a memory block, that can be treated by the DPMI client as RAM memory. The DPMI host backs up all committed pages with physical memory or backing store. Any page faults on committed pages must be transparent page faults, with the exception of page faults caused by a write to a write-protected page.

Context (DPMI)

A context is a protected mode address space as defined by an LDT and an IDT. Thus, each context has its own linear memory, selectors, and interrupt tables allocated to it.

Conventional memory

Memory which lies below the 1 MB boundary and which can be addressed in real mode.

Current client

The DPMI clients within the same virtual machine form a program stack. Only one client in the program stack of a virtual machine can be active at one time and the client that is currently active is called the current client. When a client becomes "current" by virtue of a real mode callback or a raw mode switch, its LDT and IDT become active and software interrupts and protected mode exceptions it issues are reflected through its IDT.

DOS Protected Mode Interface (DPMI)

DPMI defines an interface for protected mode DOS applications to manage memory, interrupts, exceptions, debugging registers, and coprocessor emulation in the presence of an 80386 control program or true protected mode operating system.

Expanded memory and Expanded memory emulator

Bank-switched memory that exists outside the CPU's normal address space and that can be made addressable in small units called "pages" by calls to a device driver that supports the Expanded Memory Specification. The pages appear in a "page frame" that typically lies above the 640 KB boundary and below the 1 MB boundary. Expanded memory emulators are programs that run on 80386 or later CPUs and use the paging unit to map extended memory into conventional memory, implementing the EMS interface without the need for special memory boards or other hardware.

Expanded Memory Specification (EMS)

The expanded memory management interface defined by Lotus, Intel, and Microsoft. The current version of this specification is 4.0. VCPI is defined as an extension to EMS. The EMS 4.0 Specification can be obtained by writing to Intel Corporation, 5200 N.E. Elam Young Parkway, Hillsboro, OR, 97124, USA.

Extended memory

Memory which lies above the 1 MB boundary and can only be addressed in protected mode.

eXtended Memory Specification (XMS)

A handle-based extended memory management interface defined by Microsoft. The XMS Specification can be obtained by writing to Microsoft Corporation, Box 97017, Redmond, WA, 98073, USA.

Host, DPMI

A program or operating system which implements the Int 2FH and Int 31H functions defined in this specification, and makes these services available to client programs.

Mapped page

A page, within a memory block, which can be treated as present by the DPMI client, but which maps, directly or indirectly, a physical memory-mapped device or committed memory allocated in the DOS conventional memory space. Any page faults on mapped pages (e.g. because the host is virtualizing the device) must be transparent page faults, with the exception of page faults caused by a write to a write-protected page. Mapped pages within memory blocks may not be supported in all DPMI hosts.

Memory block

A contiguous block of linear memory. A memory block is allocated, resized, and freed via calls to the DPMI host, and is identified by an unique handle. Within an existing memory block, the DPMI client can manipulate individual pages.

Primary Client

DPMI clients within the same virtual machine are viewed as a stack of programs with the first DPMI client sitting at the bottom of the stack. A primary client of a VM is the topmost (most recent) client of the stack within the VM. It is the last client of the virtual machine that called the protected mode entry routine obtained through Int 2FH Function 1687H. The DPMI host is responsible for ensuring that hardware interrupts and real mode exceptions are sent to the primary client..

Resident service provider

Resident service provider (somewhat like protected mode TSRs) provides services to programs running in the same virtual machine. Resident service providers are initialized only in the virtual machine in which they are started.

Shared services

Shared services (shared memory and serialization support) allow sharing between programs running in different virtual machines, or between programs running in the same virtual machine.

Top down allocation

Also called Int 15H allocation. A method of extended memory management which relies on the existence of the ROM BIOS function "Get Extended Memory Size" (Int 15H Function 88H). The program which wishes to allocate extended memory first calls Int 15H Function 88H to determine the amount of memory available, then replaces the previous handler for the function with its own, and returns a lesser value to subsequent callers of the function. Compared with bottom-up (VDISK compatible) allocation, this scheme is recommended.

Transparent Exception

An exception which is handled by the DPMI host, without the knowledge of or possibility of intervention by the DPMI client.

Transparent page fault

A page fault which is handled by the DPMI host, without the knowledge of or possibility of intervention by the DPMI client. A transparent page fault can result from host implementation of virtual memory or physical device virtualization.

Uncommitted page

A page whose linear addresses have not been mapped to any physical memory or do not correspond to any area of backing store. A reference to an uncommitted page will always generate a visible page fault (Int 0EH).

Virtual Control Program Interface (VCPI)

The VCPI is the predecessor to DPMI. VCPI extends the EMS interface to allow DOS Extenders to run in the presence of EMS emulators or other 80386 control programs. The Virtual Control Program Interface was a collaborative effort of Phar Lap Software and Quarterdeck Office Systems, and it became an industry standard in 1989. The VCPI Specification can be obtained by writing to Phar Lap Software, Inc., 60 Aberdeen Avenue, Cambridge, MA, 02138, USA.

Virtual DMA Services (VDS)

The VDS Specification defines an interface by which protected mode programs and device drivers can obtain the necessary information to program a DMA transfer using the on board DMA controller or a busmaster DMA controller. The VDS Specification version 1.0 (Microsoft Part # 098-10869) can be obtained by writing to: Microsoft Corporation, Box 97017, Redmond, WA, 98073, USA.

Virtual DOS environment

The environment in which a DOS application runs, under the control of an protected mode supervisor or operating system which supports multiple virtual machines. The DOS application executes as though it has sole use of the operating system services and direct access to the hardware, but the supervisor may selectively virtualize service requests, exceptions, and I/O operations. For example, the supervisor will typically handle all hardware interrupts and CPU exceptions, but may then "reflect" these interrupts or exceptions to the "real mode" handlers.

Virtual machine, DPMI virtual machine

A virtual machine, as the term is used in the DPMI specification, is a DOS address space and all of the DPMI clients which share access to that DOS address space. Multitasking hosts that can support multiple virtual-86 address spaces.

VIsible exception

Every exception is first examined by the DPMI host. If the host does not handle the exception, it passes the exception to the first handle in the protected mode exception handler chain. An exception that is passed to the client handler chain become a visible exception to a client.

Visible page fault

A page fault which is passed to the DPMI client page fault handler, if one is installed. A visible page fault results from access to an uncommitted page, or a write access to a write-protected page.

Appendix B: Error Codes and Messages

Nearly all Int 31H function calls can fail, either because of client errors, unavailable resources, or internal host problems. Most failures due to client errors and all failures due to unavailable resources are reported to the client via error codes. Some client errors, such as passing an invalid pointer in a function call, may cause the host to fault; the client can detect these events by installing an exception handler.

Internal host errors are handled in a host-specific manner and generally not reported to clients with an error code. The only exception to this is the case when a host cannot allocate internal resources. Any Int 31H function is capable of returning error code 8010H to indicate this condition.

A DPMI 1.0 host signals an error by returning from a function with the Carry flag set and an error code in AX. If the error code has bit 15 clear (0), the DPMI host is passing a DOS error code through to the client; for a list of these error codes, consult a DOS technical reference. If the error code has bit 15 set (1), it is generated within the DPMI host, and is interpreted according to the list below. All DPMI 1.0 hosts are required to check for the error conditions listed in this specification, and must return the error codes that are documented for each function.

If Int 31H is invoked with an function number that is not defined in this specification, the DPMI host will return the "Unsupported Function" error code 8001H. The table lists all defined error codes and their messages. Unused error codes are reserved for the later versions of the DPMI spcifications.

Error Code	Name	Explanation
8001H	Unsupported function	Returned in response to any function call which is not implemented by this host, because the requested function is either undefined or optional.
8002H	Invalid state	Some object is in the wrong state for the requested operation.
8003H	System integrity	The requested operation would endanger system integrity, e.g., a request to map linear addresses onto system code or data.
8004H	Deadlock	Host detected a deadlock situation.
8005H	Request cancelled	A pending serialization request was cancelled.
8010H	Resource Unavailable	The DPMI host cannot allocate internal resources to complete an operation.
8011H	Descriptor unavailable	Host is unable to allocate a descriptor.
8012H	Linear memory unavailable	Host is unable to allocate the required linear memory.
8013H	Physical memory unavailable	Host is unable to allocate the required physical memory.
8014H	Backing store unavailable	Host is unable to allocate the required backing store.

8015H	Callback unavailable	Host is unable to allocate the required callback address.
8016H	Handle unavailable	Host is unable to allocate the required handle.
8017H	Lock count exceeded	A locking operation exceeds the maximum count maintained by the host.
8018H	Resource owned exclusively	A request for serialization of a shared memory block could not be satisfied because it is already serialized exclusively by another client.
8019H	Resource owned shared	A request for exclusive serialization of a shared memory block could not be satisfied because it is already serialized shared by another client.
8021H	Invalid value	A numeric or flag parameter has an invalid value.
8022H	Invalid selector	A selector does not correspond to a valid descriptor.
8023H	Invalid handle	A handle parameter is invalid.
8024H	Invalid callback	A callback parameter is invalid.
8025H	Invalid linear address	A linear address range (either supplied as a parameter or implied by the call) is invalid.
8026H	Invalid request	The request is not supported by the underlying hardware.

Appendix C: Differences between DPMI 0.9 and 1.0

This section summarizes the differences between DPMI version 0.9 hosts and DPMI version 1.0 hosts. For more detailed information, see the individual function descriptions and the DPMI 0.9 Specification dated May 15, 1990.

New Functions in DPMI Version 1.0

The following Int 2FH and Int 31H functions are new in the DPMI 1.0 Specification and are not supported by DPMI 0.9 hosts:

Int 2FH Function	**Name**
168AH	Get Vendor-Specific API Entry Point

Int 31H Function	**Name**
000EH	Get Multiple Descriptors
000FH	Set Multiple Descriptors
0210H	Get Extended Processor Exception Handler Vector for Protected Mode
0211H	Get Extended Processor Exception Handler Vector for Real Mode
0212H	Set Extended Processor Exception Handler Vector for Protected Mode
0213H	Set Extended Processor Exception Handler Vector for Real Mode
0401H	Get DPMI Capabilities
0504H	Allocate Linear Memory Block
0505H	Resize Linear Memory Block
0506H	Get Page Attributes
0507H	Set Page Attributes
0508H	Map Device in Memory Block
0509H	Map Conventional Memory in Memory Block
050AH	Get Memory Block Size and Base
050BH	Get Memory Information
0801H	Free Physical Address Mapping
0C00H	Install Resident Service Provider Callback
0C01H	Terminate and Stay Resident
0D00H	Allocate Shared Memory
0D01H	Free Shared Memory
0D02H	Serialize on Shared Memory
0D03H	Free Serialization on Shared Memory
0E00H	Get Coprocessor Status
0E01H	Set Coprocessor Emulation

DPMI Version 0.9 Functions Superseded

The following Int 31H functions are supported for backward compatibility with DPMI 0.9, but are not recommended for use by DPMI 1.0 clients.

Int 31H Function	Name	Comments
0202H	Get Processor Exception Handler Vector	Superseded by Int 31H Functions 0210H and 0211H
0203H	Set Processor Exception Handler Vector	Superseded by Int 31H Functions 0212H and 0213H
0500H	Get Free Memory Information	Superseded by Int 31H Function 050BH
0A00H	Get Vendor-Specific API Entry Point	Superseded by Int 2FH Function 168AH

Error Codes

DPMI 0.9 hosts indicated an error by returning from an Int 31H function call with the Carry bit set. DPMI 1.0 hosts indicate an error by returning from the function with the Carry bit set but, in addition, return an error code in AX that provides more information about the cause of the error. If the Carry flag is set and bit 15 of AX is clear (0), the error code is being passed through from DOS. If the Carry flag is set and bit 15 of AX is set (1), the error code is generated by the DPMI host and the meaning of these codes is found in Appendix B of this document.

Separate LDT and IDT Per Client

Under DPMI 0.9 hosts, all protected mode programs in a virtual machine share a single local descriptor table (LDT) and use the same interrupt descriptor table (IDT). Thus, if the first DPMI client to start is a 16-bit client, no 32-bit clients will be allowed to execute. Likewise, if the first client is a 32-bit program, no 16-bit clients can execute.

DPMI 1.0 hosts provide a unique LDT and IDT for each protected mode client, and 32-bit hosts can run both 16-bit and 32-bit clients at the same time. Hooking an interrupt or exception vector for one client will not cause it to be hooked for other clients. The DPMI host is only responsible for reflecting hardware interrupts to the last task which executed the DPMI switch into protected mode. Communication between clients in separate virtual machines is accomplished by DPMI shared memory blocks. Clients within the same VM can also communicate by DPMI shared memory block as well as by resident service providers, DOS memory blocks, or a switch to real mode.A DPMI 1.0 host determines which LDT and IDT to switch to when entering protected mode by providing unique addresses for each DPMI client's raw mode switch entry points and real mode callbacks.

Termination Handling

DPMI 1.0 host handling of client termination differs from DPMI version 0.9 in the following respects:

- In DPMI version 0.9, all clients of the same virtual machine share a single LDT and use a single IDT, and only those segment descriptors allocated by the client are deallocated when it terminates. A DPMI version 0.9 client should cleanup its own segment descriptors before its termination since some DPMI version 0.9 hosts may not free the terminating client's segment descriptors if the client is not the topmost client. In DPMI version 1.0, each client has its own LDT which is freed in its entirety at termination.

- In DPMI version 0.9, all clients within a VM share a single IDT, and interrupt descriptors and exception handlers are not automatically freed (i.e. the IDT is unchanged when the client terminates). In DPMI version 1.0, exception handlers are deregistered automatically at termination, and each client has its own IDT which is freed in its entirety.

- In DPMI version 0.9, real mode memory that has been unlocked by the client must be explicitly relocked by the client prior to termination. DPMI 1.0 hosts will relock real mode memory automatically.

DPMI Version 0.9 Compatibility Notes

- The selector supplied to Int 31H Function 000AH may be either a data selector or an executable selector. The DPMI 0.9 specification was in error to say that the function generates an error on a data descriptor.

- The DPMI minor version number defined in Int 2FH Function 1687H and Int 31H Function 0400H have two decimal digits (with the least significant digit representing the revision number within the minor revision number). Under DPMI version 0.9 hosts, DL is returned as decimal 90 (5AH). The convention of having two decimal digits returned in DL for the minor version number was not documented in DPMI 0.9 specification.

- Some DPMI version 0.9 clients are incorrectly making use of GDT selector 0040H as referring to 40:0H in real mode. New DPMI host implementations should determine if they want to work around the compatibility problem introduced by these clients.

- If the present bit of the access right/type bype in the LDT descriptor is not set, a DPMI host allows any values in the byte except in the DPL and "must be 1" bit fields. The host requirements for the "must be 1" bit field was not documented in DPMI 0.9 specification.

- To be consistent with some existing DPMI version 0.9 implementation, all DPMI hosts including DPMI version 1.0 hosts must reflect the interrupt in protected mode if it is not hooked as an protected mode exception (this is an update to page 30 exception handling description). Furthermore, some DPMI version 0.9 hosts do not reflect some interrupts down the protected mode exception chain and the clients may need to hook both the protected mode exception chain and protected mode interrupt chain to handle those m interrupts correctly.

Appendix D: Descriptor Usage Rules

The following table shows the DPMI client's restrictions on usage of previously allocated descriptors as input parameters to DPMI functions. The columns represent the ways the DPMI host allocates descriptors for its clients. (The first two columns represents LDT descriptor management functions which allocate descriptors, the third column represents DOS memory functions, and the last column represents "other" descriptors, i.e., unallocated descriptors or descriptors used by the DPMI host for internal purposes.) Each row represents a set of functions where a client passes those previously allocated descriptor(s) to the host as input parameters. A 'N' indicates that an "invalid Selector" error will be generated if the given descriptor is used in the specified function.

Note that a "Y" for a given entry does not indicate that the function will succeed, only that it will not generate an "Invalid Selector" error. Similarly, an "N" does not necessarily indicate a descriptor is invalid for referencing memory, only that it cannot be used with that particular function. This chart does not address the usage of descriptors in pointers.

For example, descriptors allocated by the Allocate LDT Descriptor function may be used in any of the interrogation and modification functions of LDT Descriptor Management, as well as the functions which set exception handlers and interrupt vectors. They may not be passed to Allocate Specific LDT Descriptor or the DOS Memory Block functions.

Descriptor Allocators *Functions referring the allocated descriptors*	Allocate LDT Descriptor, Allocate Specific LDT Descriptor, Create Alias descriptor, Initial CS, DS, SS	Segment To Descriptor, PSP, Environment Pointer, Callback DS, Locked Stack SS	Allocate/ Resize DOS Memory Block	GDT-based Descriptor, System Descriptor, Unallocated Descriptor
Interrogation Functions: Get Segment Base Address, Get Descriptor, Get Multiple Descriptor, Create Segment Alias	Y	Y	Y	N
Modification Functions: Set Segment Base Address, Set Segment Limit, Set Descriptor Access Rights, Set Descriptor, SetMultiple Descriptor, Free LDT Descriptor	Y	N	N	N
Allocate Specific LDT Descriptor	N	N	N	Y[1]
Free/Resize DOS Memory Block	N	N	Y	N
Set Exception Handler/Interrupt Vector	Y	Y[3]	Y	Y[2]

Notes:

1. Unallocated descriptors within the range of the LDT only.
2. GDT-based segment descriptors only.
3. Although this call will succeed, a fault will result if the exception or interrupt occurs, since the segment can never be made executable.