eXtended Memory Specification (XMS), ver 3.0

January 1991

Microsoft Corporation
Box 97017

One Microsoft Way
Redmond, WA 98073

LOTUS (r)
INTEL (r)
MICROSOFT (r)
AST (r) Research

This specification was jointly developed by Microsoft Corporation, Lotus Development Corporation, Intel Corporation,and AST Research, Inc. Although it has been released into the public domain and is not confidential or proprietary, the specification is still the copyright and property of Microsoft Corporation, Lotus Development Corporation, Intel Corporation, and AST Research, Inc.

Disclaimer of Warranty

MICROSOFT CORPORATION, LOTUS DEVELOPMENT CORPORATION, INTEL CORPORATION, AND AST RESEARCH, INC., EXCLUDE ANY AND ALL IMPLIED WARRANTIES, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. NEITHER MICROSOFT NOR LOTUS NOR INTEL NOR AST RESEARCH MAKE ANY WARRANTY OF REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS SPECIFICATION, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. NEITHER MICROSOFT NOR LOTUS NOR INTEL NOR AST RESEARCH SHALL HAVE ANY LIABILITY FOR SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RESULTING FROM THE USE OR MODIFICATION OF THIS SPECIFICATION.

This specification uses the following trademarks:

Intel is a registered trademark of Intel Corporation, Microsoft is a registered trademark of Microsoft Corporation, Lotus is a registered trademark of Lotus Development Corporation, and AST is a registered trademark of AST Research, Inc.

Extended Memory Specification

The purpose of this document is to define the Extended Memory Specification (XMS) version 3.00 for MS-DOS. XMS allows DOS programs to utilize additional memory found in Intel's 80286 and 80386 based machines in a consistent, machine independent manner. With some restrictions, XMS adds almost 64K to the 640K which DOS programs can access directly. Depending on available hardware, XMS

may provide even more memory to DOS programs. XMS also provides DOS programs with a standard method of storing data in extended memory.

To be considered fully XMS 3.0 compliant, all calls except those associated with UMB support must be implemented. UMB functions 10h, 11h and 12h are optional for XMS 3.0 and may return the Function Not Implemented error code, 80h.

DEFINITIONS:

Extended Memory:
Memory in 80286 and 80386 based machines which is located above the 1MB address boundary.

High Memory Area (HMA):
The first 64K of extended memory. The High Memory Area is unique because code can be executed in it while in real mode. The HMA officially starts at FFFF:10h and ends at FFFF:FFFFh making it 64K-16 bytes in length.

Upper Memory Blocks (UMBs):
Blocks of memory available on some 80x86 based machines which are located between DOS's 640K limit and the 1MB address boundary. The number, size, and location of these blocks vary widely depending upon the types of hardware adapter cards installed in the machine.

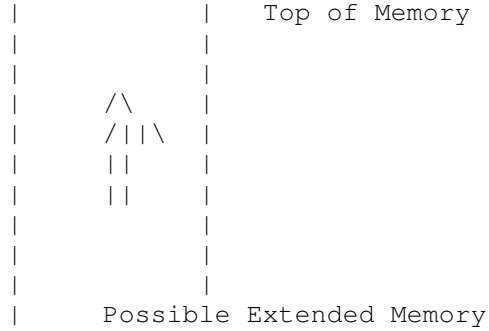
Extended Memory Blocks (EMBs):
Blocks of extended memory located above the HMA which can only be used for data storage.

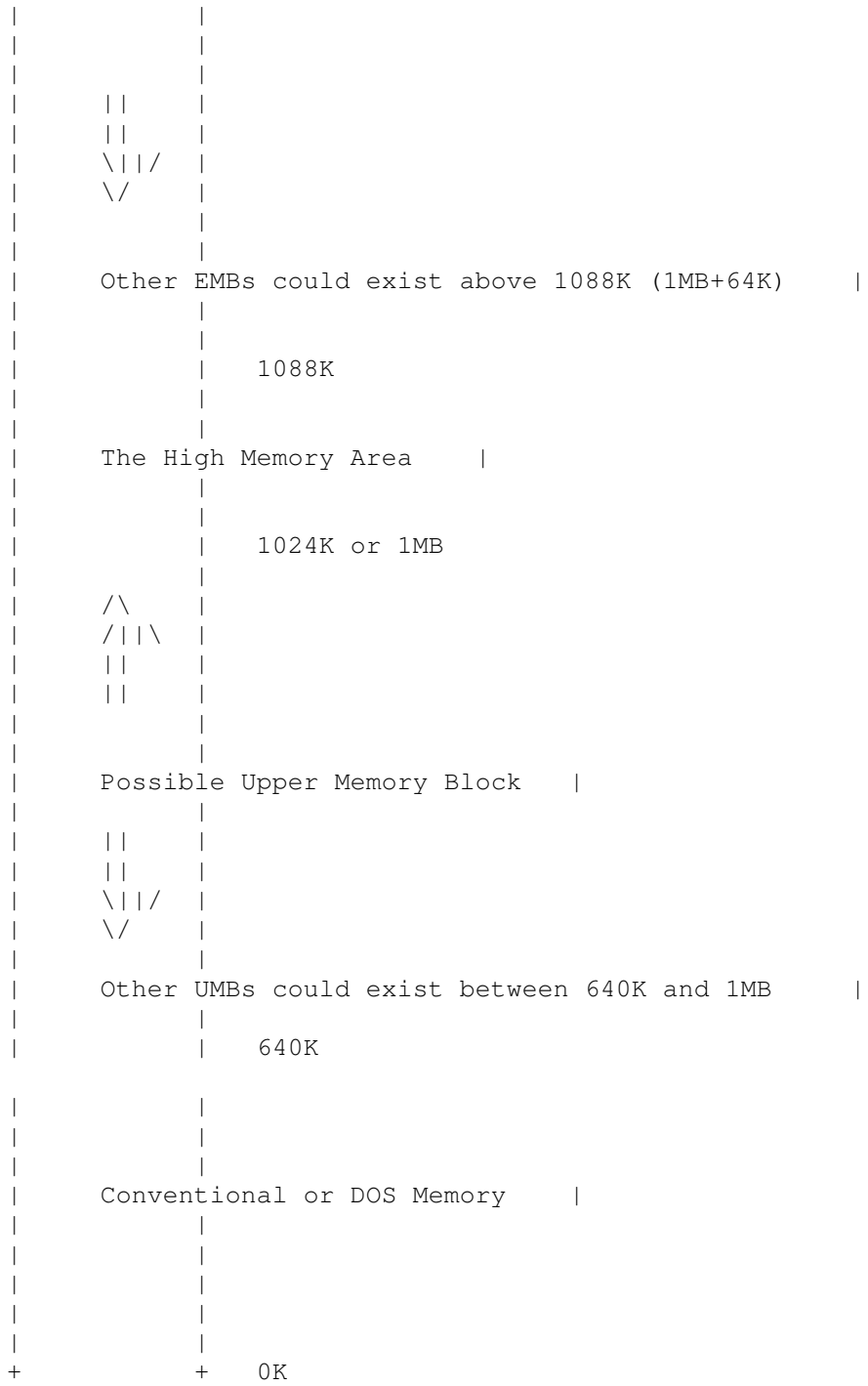
A20 Line:
The 21st address line of 80x86 CPUs. Enabling the A20 line allows access to the HMA.

XMM:
An Extended Memory Manager. A DOS device driver which implements XMS. XMMs are machine specific but allow programs to use extended memory in a machine-independent manner.

HIMEM.SYS:
The Extended Memory Manager currently being distributed by Microsoft.

Helpful Diagram:





DRIVER INSTALLATION:

An XMS driver is installed by including a DEVICE= statement in the machine's CONFIG.SYS file. It must be installed prior to any other devices or TSRs which use it. An optional parameter after the driver's name (suggested name "/HMAMIN=") indicates the minimum amount of space in the HMA a program can use. Programs which use less than the minimum will not be placed in the HMA. See "Prioritizing HMA Usage" below for more

information. A second optional parameter (suggested name "/NUMHANDLES=") allows users to specify the maximum number of extended memory blocks which may be allocated at any time.

NOTE: XMS requires DOS 3.00 or above.

THE PROGRAMMING API:

The XMS API Functions are accessed via the XMS driver's Control Function. The address of the Control Function is determined via INT 2Fh. First, a program should determine if an XMS driver is installed. Next, it should retrieve the address of the driver's Control Function. It can then use any of the available XMS functions. The functions are divided into several groups:

1. Driver Information Functions (0h)
2. HMA Management Functions (1h-2h)
3. A20 Management Functions (3h-7h)
4. Extended Memory Management Functions (8h-Fh)
5. Upper Memory Management Functions (10h-11h)

DETERMINING IF AN XMS DRIVER IS INSTALLED:

The recommended way of determining if an XMS driver is installed is to set AH=43h and AL=00h and then execute INT 2Fh. If an XMS driver is available, 80h will be returned in AL.

Example:

```
        ; Is an XMS driver installed?
        mov     ax,4300h
        int     2Fh
        cmp     al,80h
        jne     NoXMSDriver
```

CALLING THE API FUNCTIONS:

Programs can execute INT 2Fh with AH=43h and AL=10h to obtain the address of the driver's control function. The address is returned in ES:BX. This function is called to access all of the XMS functions. It should be called with AH set to the number of the API function requested. The API function will put a success code of 0001h or 0000h in AX. If the function succeeded (AX=0001h), additional information may be passed back in BX and DX. If the function failed (AX=0000h), an error code may be returned in BL. Valid error codes have their high bit set. Developers should keep in mind that some of the XMS API functions may not be implemented by all drivers and will return failure in all cases.

Example:

```
        ; Get the address of the driver's control function
        mov     ax,4310h
        int     2Fh
```

```
            mov     word ptr [XMSControl],bx        ; XMSControl is a DWORD
            mov     word ptr [XMSControl+2],es

            ; Get the XMS driver's version number
            mov     ah,00h
            call    [XMSControl]    ; Get XMS Version Number
```

    NOTE: Programs should make sure that at least 256 bytes of stack space
        is available before calling XMS API functions.

API FUNCTION DESCRIPTIONS:

    The following XMS API functions are available:

       0h) Get XMS Version Number
       1h) Request High Memory Area
       2h) Release High Memory Area
       3h) Global Enable A20
       4h) Global Disable A20
       5h) Local Enable A20
       6h) Local Disable A20
       7h) Query A20
       8h) Query Free Extended Memory
       9h) Allocate Extended Memory Block
       Ah) Free Extended Memory Block
       Bh) Move Extended Memory Block
       Ch) Lock Extended Memory Block
       Dh) Unlock Extended Memory Block
       Eh) Get Handle Information
       Fh) Reallocate Extended Memory Block
      10h) Request Upper Memory Block
      11h) Release Upper Memory Block
      12h) Realloc Upper Memory Block
      88h) Query any Free Extended Memory
      89h) Allocate any Extended Memory Block
      8Eh) Get Extended EMB Handle
      8Fh) Realloc any Extended Memory

Each is described below.

Get XMS Version Number (Function 00h):

    ARGS: AH = 00h
    RETS: AX = XMS version number
          BX = Driver internal revision number
          DX = 0001h if the HMA exists, 0000h otherwise
    ERRS: None

    This function returns with AX equal to a 16-bit BCD number representing the revision of the DOS Extended Memory Specification which the driver implements (e.g. AX=0235h would mean that the driver implemented XMS version 2.35).   BX is set equal to the driver's internal revision number mainly for debugging purposes. DX indicates the existence of the HMA (not its

availability) and is intended mainly for installation programs.

NOTE: This document defines version 3.00 of the specification.

Request High Memory Area (Function 01h):

```
ARGS:   AH = 01h
        If the caller is a TSR or device driver,
            DX = Space needed in the HMA by the caller in bytes
        If the caller is an application program,
            DX = FFFFh
RETS:   AX = 0001h if the HMA is assigned to the caller, 0000h otherwise
ERRS:   BL = 80h if the function is not implemented
      BL = 81h if a VDISK device is detected
      BL = 90h if the HMA does not exist
      BL = 91h if the HMA is already in use
      BL = 92h if DX is less than the /HMAMIN= parameter
```

This function attempts to reserve the 64K-16 byte high memory area for the caller. If the HMA is currently unused, the caller's size parameter is compared to the /HMAMIN= parameter on the driver's command line. If the value passed by the caller is greater than or equal to the amount specified by the driver's parameter, the request succeeds. This provides the ability to ensure that programs which use the HMA efficiently have priority over those which do not.

NOTE: See the sections "Prioritizing HMA Usage" and "High Memory Area Restrictions" below for more information.

Release High Memory Area (Function 02h):

```
ARGS:   AH = 02h
RETS:   AX = 0001h if the HMA is successfully released, 0000h otherwise
ERRS:   BL = 80h if the function is not implemented
      BL = 81h if a VDISK device is detected
      BL = 90h if the HMA does not exist
      BL = 93h if the HMA was not allocated
```

This function releases the high memory area and allows other programs to use it. Programs which allocate the HMA must release it before exiting. When the HMA has been released, any code or data stored in it becomes invalid and should not be accessed.

Global Enable A20 (Function 03h):

```
ARGS:   AH = 03h
RETS:   AX = 0001h if the A20 line is enabled, 0000h otherwise
ERRS:   BL = 80h if the function is not implemented
      BL = 81h if a VDISK device is detected
      BL = 82h if an A20 error occurs
```

This function attempts to enable the A20 line. It should only be used by programs which have control of the HMA. The A20 line should be turned off via Function 04h (Global Disable A20) before a program releases control of the system.

NOTE: On many machines, toggling the A20 line is a relatively slow operation.

Global Disable A20 (Function 04h):

ARGS: AH = 04h
RETS: AX = 0001h if the A20 line is disabled, 0000h otherwise
ERRS: BL = 80h if the function is not implemented
BL = 81h if a VDISK device is detected
BL = 82h if an A20 error occurs
BL = 94h if the A20 line is still enabled

This function attempts to disable the A20 line. It should only be used by programs which have control of the HMA. The A20 line should be disabled before a program releases control of the system.

NOTE: On many machines, toggling the A20 line is a relatively slow operation.

Local Enable A20 (Function 05h):

ARGS: AH = 05h
RETS: AX = 0001h if the A20 line is enabled, 0000h otherwise
ERRS: BL = 80h if the function is not implemented
BL = 81h if a VDISK device is detected
BL = 82h if an A20 error occurs

This function attempts to enable the A20 line. It should only be used by programs which need direct access to extended memory. Programs which use this function should call Function 06h (Local Disable A20) before releasing control of the system.

NOTE: On many machines, toggling the A20 line is a relatively slow operation.

Local Disable A20 (Function 06h):

ARGS: AH = 06h
RETS: AX = 0001h if the function succeeds, 0000h otherwise
ERRS: BL = 80h if the function is not implemented
BL = 81h if a VDISK device is detected
BL = 82h if an A20 error occurs
BL = 94h if the A20 line is still enabled

This function cancels a previous call to Function 05h (Local Enable A20). It should only be used by programs which need direct access to

extended memory. Previous calls to Function 05h must be canceled before releasing control of the system.

NOTE: On many machines, toggling the A20 line is a relatively slow operation.

Query A20 (Function 07h):

ARGS: AH = 07h
RETS: AX = 0001h if the A20 line is physically enabled, 0000h otherwise
ERRS: BL = 00h if the function succeeds
BL = 80h if the function is not implemented
BL = 81h if a VDISK device is detected

This function checks to see if the A20 line is physically enabled. It does this in a hardware independent manner by seeing if "memory wrap" occurs.

Query Free Extended Memory (Function 08h):

ARGS: AH = 08h
RETS: AX = Size of the largest free extended memory block in K-bytes
DX = Total amount of free extended memory in K-bytes
ERRS: BL = 80h if the function is not implemented
BL = 81h if a VDISK device is detected
BL = A0h if all extended memory is allocated

This function returns the size of the largest available extended memory block in the system.

NOTE: The 64K HMA is not included in the returned value even if it is not in use.

Allocate Extended Memory Block (Function 09h):

ARGS: AH = 09h
DX = Amount of extended memory being requested in K-bytes
RETS: AX = 0001h if the block is allocated, 0000h otherwise
DX = 16-bit handle to the allocated block
ERRS: BL = 80h if the function is not implemented
BL = 81h if a VDISK device is detected
BL = A0h if all available extended memory is allocated
BL = A1h if all available extended memory handles are in use

This function attempts to allocate a block of the given size out of the pool of free extended memory. If a block is available, it is reserved for the caller and a 16-bit handle to that block is returned. The handle should be used in all subsequent extended memory calls. If no memory was allocated, the returned handle is null.

NOTE: Extended memory handles are scarce resources. Programs should try to allocate as few as possible at any one time. When all

of a driver's handles are in use, any free extended memory is unavailable.

Free Extended Memory Block (Function 0Ah):

```
ARGS:   AH = 0Ah
        DX = Handle to the allocated block which should be freed
RETS:   AX = 0001h if the block is successfully freed, 0000h otherwise
ERRS:   BL = 80h if the function is not implemented
      BL = 81h if a VDISK device is detected
      BL = A2h if the handle is invalid
      BL = ABh if the handle is locked
```

This function frees a block of extended memory which was previously allocated using Function 09h (Allocate Extended Memory Block). Programs which allocate extended memory should free their memory blocks before exiting. When an extended memory buffer is freed, its handle and all data stored in it become invalid and should not be accessed.

Move Extended Memory Block (Function 0Bh):

```
ARGS:   AH = 0Bh
        DS:SI = Pointer to an Extended Memory Move Structure (see below)
RETS:   AX = 0001h if the move is successful, 0000h otherwise
ERRS:   BL = 80h if the function is not implemented
      BL = 81h if a VDISK device is detected
      BL = 82h if an A20 error occurs
      BL = A3h if the SourceHandle is invalid
      BL = A4h if the SourceOffset is invalid
      BL = A5h if the DestHandle is invalid
      BL = A6h if the DestOffset is invalid
      BL = A7h if the Length is invalid
      BL = A8h if the move has an invalid overlap
      BL = A9h if a parity error occurs
```

Extended Memory Move Structure Definition:

```
ExtMemMoveStruct    struc
    Length              dd  ?   ; 32-bit number of bytes to transfer
    SourceHandle        dw  ?   ; Handle of source block
    SourceOffset        dd  ?   ; 32-bit offset into source
    DestHandle          dw  ?   ; Handle of destination block
    DestOffset          dd  ?   ; 32-bit offset into destination block
ExtMemMoveStruct    ends
```

This function attempts to transfer a block of data from one location to another. It is primarily intended for moving blocks of data between conventional memory and extended memory, however it can be used for moving blocks within conventional memory and within extended memory.

NOTE: If SourceHandle is set to 0000h, the SourceOffset is interpreted as a standard segment:offset pair which refers to memory that is

directly accessible by the processor. The segment:offset pair is stored in Intel DWORD notation. The same is true for DestHandle and DestOffset.

SourceHandle and DestHandle do not have to refer to locked memory blocks.

Length must be even. Although not required, WORD-aligned moves can be significantly faster on most machines. DWORD aligned move can be even faster on 80386 machines.

If the source and destination blocks overlap, only forward moves (i.e. where the source base is less than the destination base) are guaranteed to work properly.

Programs should not enable the A20 line before calling this function. The state of the A20 line is preserved.

This function is guaranteed to provide a reasonable number of interrupt windows during long transfers.

Lock Extended Memory Block (Function 0Ch):

```
ARGS:   AH = 0Ch
        DX = Extended memory block handle to lock
RETS:   AX = 0001h if the block is locked, 0000h otherwise
      DX:BX = 32-bit physical address of the locked block
ERRS:   BL = 80h if the function is not implemented
        BL = 81h if a VDISK device is detected
        BL = A2h if the handle is invalid
        BL = ACh if the block's lock count overflows
        BL = ADh if the lock fails
```

This function locks an extended memory block and returns its base address as a 32-bit physical address. Locked memory blocks are guaranteed not to move. The 32-bit pointer is only valid while the block is locked. Locked blocks should be unlocked as soon as possible.

NOTE: A block does not have to be locked before using Function 0Bh (Move Extended Memory Block).

"Lock counts" are maintained for EMBs.

Unlock Extended Memory Block (Function 0Dh):

```
ARGS:   AH = 0Dh
        DX = Extended memory block handle to unlock
RETS:   AX = 0001h if the block is unlocked, 0000h otherwise
ERRS:   BL = 80h if the function is not implemented
        BL = 81h if a VDISK device is detected
        BL = A2h if the handle is invalid
        BL = AAh if the block is not locked
```

This function unlocks a locked extended memory block. Any 32-bit pointers into the block become invalid and should no longer be used.

Get EMB Handle Information (Function 0Eh):

```
ARGS:   AH = 0Eh
        DX = Extended memory block handle
RETS:   AX = 0001h if the block's information is found, 0000h otherwise
        BH = The block's lock count
        BL = Number of free EMB handles in the system
        DX = The block's length in K-bytes
ERRS:   BL = 80h if the function is not implemented
        BL = 81h if a VDISK device is detected
        BL = A2h if the handle is invalid
```

This function returns additional information about an extended memory block to the caller.

NOTE: To get the block's base address, use Function 0Ch (Lock Extended Memory Block).

Reallocate Extended Memory Block (Function 0Fh):

```
ARGS:   AH = 0Fh
        BX = New size for the extended memory block in K-bytes
        DX = Unlocked extended memory block handle to reallocate
RETS:   AX = 0001h if the block is reallocated, 0000h otherwise
ERRS:   BL = 80h if the function is not implemented
        BL = 81h if a VDISK device is detected
        BL = A0h if all available extended memory is allocated
        BL = A1h if all available extended memory handles are in use
        BL = A2h if the handle is invalid
        BL = ABh if the block is locked
```

This function attempts to reallocate an unlocked extended memory block so that it becomes the newly specified size. If the new size is smaller than the old block's size, all data at the upper end of the old block is lost.

Request Upper Memory Block (Function 10h):

```
ARGS:   AH = 10h
        DX = Size of requested memory block in paragraphs
RETS:   AX = 0001h if the request is granted, 0000h otherwise
        BX = Segment number of the upper memory block
          If the request is granted,
          DX = Actual size of the allocated block in paragraphs
        otherwise,
          DX = Size of the largest available UMB in paragraphs
ERRS:   BL = 80h if the function is not implemented
```

BL = B0h if a smaller UMB is available
BL = B1h if no UMBs are available

This function attempts to allocate an upper memory block to the caller. If the function fails, the size of the largest free UMB is returned in DX.

NOTE: By definition UMBs are located below the 1MB address boundary. The A20 Line does not need to be enabled before accessing an allocated UMB.

UMBs are paragraph aligned.

To determine the size of the largest available UMB, attempt to allocate one with a size of FFFFh.

UMBs are unaffected by EMS calls.

Release Upper Memory Block (Function 11h):

ARGS: AH = 11h
DX = Segment number of the upper memory block
RETS: AX = 0001h if the block was released, 0000h otherwise
ERRS: BL = 80h if the function is not implemented
BL = B2h if the UMB segment number is invalid

This function frees a previously allocated upper memory block. When an UMB has been released, any code or data stored in it becomes invalid and should not be accessed.

Reallocate Upper Memory Block (Function 12h)

ARGS:
AH = 12h
BX = New size for UMB in paragraphs
DX = Segment number of the UMB to reallocate
RETS:
AX = 1 if the block was reallocated, 0 otherwise
ERRS:
BL = 80h if the function is not implemented
BL = B0h if no UMB large enough to satisfy the request is available. In this event, DX is returned with the size of the largest UMB that is available.
BL = B2h if the UMB segment number is invalid

This function attempts to reallocate an Upper Memory Block to a newly specified size. If the new size is smaller than the old block's size, all data at the upper end of the block is lost.

Super Extended Memory Support

These changes are intended to provide support for extended memory pools up to 4 Gb in size. The current XMS API, since it uses 16-bit values to specify block sizes in Kb, is limited to 64 Mb maximum block size. Future machines are expected to support memory above 64 MB.

This support is implemented in the form of extensions to existing functions, rather than entirely new entry points, to allow for more efficient implementations.

Programs should generally use the existing functions, instead of these extended ones, unless they have an explicit need to deal with memory above 64 Mb.

Query Any Free Extended Memory (Function 88h)

```
Entry:
      AH = 88h
Exit:
      EAX = Size of largest free extended memory block in Kb.
      BL = 0 if no error occurs, otherwise it takes an error code.
      ECX = Highest ending address of any memory block.
      EDX = Total amount of free memory in Kb.
Errors:
      BL = 80h if the function is not implemented.
      BL = 81h if a VDISK device is detected.
      BL = A0h if all extended memory is allocated.
```

This function uses 32-bit values to return the size of available memory, thus allowing returns up to 4GByte. Additionally, it returns the highest known physical memory address, that is, the physical address of the last byte of memory. There may be discontinuities in the memory map below this address.

The memory pool reported on is the same as that reported on by the existing Query Free Extended Memory function. If the highest memory address is not more than 64 Mb, then these two functions will return the same results.

Because of its reliance on 32-bit registers, this function is only available on 80386 and higher processors. XMS drivers on 80286 machines should return error code 80h if this function is called.

If error code 81h is returned, the value in ECX will still be valid. If error code A0h is returned, EAX and EDX will be 0, and ECX will still be valid.

Allocate Any Extended Memory (Function 89h)

```
Entry:
      AH = 89h
      EDX = Amount of extended memory requested, in Kb.
Exit:
      AX = 1 if the block is allocated, 0 if not
      DX = Handle to allocated block.
Errors:
      BL = 80h if the function is not implemented.
      BL = 81h if a VDISK device is detected.
      BL = A0h if all available extended memory is allocated.
      BL = A1h if all available extended memory handles are in use.
```

This function is similar to the existing Allocate Extended Memory, except that it uses a 32-bit instead of a 16-bit value to specify the amount of memory requested. It allocates from the same memory and handle pool as the current function. Since it requires a 32-bit register, this function can be supported only on 80386 and higher processors, and XMS drivers on 80286 machines should return error code 80h.

Get Extended EMB Handle Information (Function 8Eh)

```
      Entry:
            AH = 8Eh
            DX = Extended memory block handle.
      Exit:
            AX = 1 if the block's information is found, 0 if not
            BH = Block lock count
            CX = Number of free EMB handles in the system
            EDX = Block's length in Kb.
      Errors:
            BL = 80h if the function is not implemented.
            BL = 81h if a VDISK device is detected.
            BL = A2h if the handle is invalid.
```

This function is similar to the Get EMB Handle Information function. Since it uses a 32-bit register to report the block size, it can be used to get information on blocks larger than 64 Mb. It also uses a 16-bit instead of 8-bit register to report the number of free handles, allowing the handle pool to be extended beyond 256 entries.

Because of its reliance on a 32-bit register, this function is available on 80386 and higher processors. XMS drivers on 80286 machines should return error code 80h if this function is called.

Reallocate Any Extended Memory (Function 8Fh)

```
      Entry:
            AH = 8Fh
            EBX = New size for extended memory block, in Kb.
            DX = Unlocked handle for memory block to be resized.
      Exit:
            AX = 1 if the block is reallocated, 0 if not
      Errors:
            BL = 80h if the function is not implemented.
            BL = 81h if a VDISK device is detected.
            BL = A0h if all available extended memory is allocated.
            BL = A1h if all available extended memory handles are in use.
            BL = A2h if the handle is invalid.
            BL = ABh if the block is locked.
```

This function is similar to the existing Reallocate Extended Memory, except that it uses a 32-bit instead of a 16-bit value to specify the amount of memory requested. It allocates from the same memory and handle pool as the current function. Since it requires a 32-bit register, this function can be supported only on 80386 and higher processors, and XMS drivers on 80286 machines should return error code 80h.

PRIORITIZING HMA USAGE:

For DOS users to receive the maximum benefit from the High Memory Area, programs which use the HMA must store as much of their resident code in it as is possible. It is very important that developers realize that the HMA is allocated as a single unit.

For example, a TSR program which grabs the HMA and puts 10K of code into it may prevent a later TSR from putting 62K into the HMA. Obviously, regular DOS programs would have more memory available to them below the 640K line if the 62K TSR was moved into the HMA instead of the 10K one.

The first method for dealing with conflicts such as this is to require programs which use the HMA to provide a command line option for disabling this feature. It is crucial that TSRs which do not make full use of the HMA provide such a switch on their own command line (suggested name "/NOHMA").

The second method for optimizing HMA usage is through the /HMAMIN= parameter on the XMS device driver line. The number after the parameter is defined to be the minimum amount of HMA space (in K-bytes) used by any driver or TSR. For example, if "DEVICE=HIMEM.SYS /HMAMIN=48" is in a user's CONFIG.SYS file, only programs which request at least 48K would be allowed to allocate the HMA. This number can be adjusted either by installation programs or by the user himself. If this parameter is not specified, the default value of 0 is used causing the HMA to be allocated on a first come, first served basis.

Note that this problem does not impact application programs. If the HMA is available when an application program starts, the application is free to use as much or as little of the HMA as it wants. For this reason, applications should pass FFFFh in DX when calling Function 01h.

HIGH MEMORY AREA RESTRICTIONS:

- Far pointers to data located in the HMA cannot be passed to DOS. DOS normalizes any pointer which is passed into it. This will cause data addresses in the HMA to be invalidated.

- Disk I/O directly into the HMA (via DOS, INT 13h, or otherwise) is not recommended.

- Programs, especially drivers and TSRs, which use the HMA *MUST* use as much of it as possible. If a driver or TSR is unable to use at least 90% of the available HMA (typically ~58K), they must provide a command line switch for overriding HMA usage. This will allow the user to configure his machine for optimum use of the HMA.

- Device drivers and TSRs cannot leave the A20 line permanently turned on. Several applications rely on 1MB memory wrap and will overwrite the

HMA if the A20 line is left enabled potentially causing a system crash.

- Interrupt vectors must not point into the HMA. This is a result of the previous restriction. Note that interrupt vectors can point into any allocated upper memory blocks however.

ERROR CODE INDEX:

If AX=0000h when a function returns and the high bit of BL is set,

BL=80h if the function is not implemented
81h if a VDISK device is detected
82h if an A20 error occurs
8Eh if a general driver error occurs
8Fh if an unrecoverable driver error occurs
90h if the HMA does not exist
91h if the HMA is already in use
92h if DX is less than the /HMAMIN= parameter
93h if the HMA is not allocated
94h if the A20 line is still enabled
A0h if all extended memory is allocated
A1h if all available extended memory handles are in use
A2h if the handle is invalid
A3h if the SourceHandle is invalid
A4h if the SourceOffset is invalid
A5h if the DestHandle is invalid
A6h if the DestOffset is invalid
A7h if the Length is invalid
A8h if the move has an invalid overlap
A9h if a parity error occurs
AAh if the block is not locked
ABh if the block is locked
ACh if the block's lock count overflows
ADh if the lock fails
B0h if a smaller UMB is available
B1h if no UMBs are available
B2h if the UMB segment number is invalid

IMPLEMENTATION NOTES FOR DOS XMS DRIVERS:

- A DOS XMS driver's control function must begin with code similar to the following:

```
XMMControl  proc    far

          jmp     short XCControlEntry    ; For "hookability"
          nop                     ; NOTE: The jump must be a short
          nop                     ;  jump to indicate the end of
          nop                     ;  any hook chainThe nop's
                                  ;  allow a far jump to be
                                  ;  patched in.
XCControlEntry:
```

- XMS drivers must preserve all registers except those containing returned values across any function call.

- XMS drivers are required to hook INT 15h and watch for calls to functions 87h (Block Move) and 88h (Extended Memory Available). The INT 15h Block Move function must be hooked so that the state of the A20 line is preserved across the call. The INT 15h Extended Memory Available function must be hooked to return 0h to protect the HMA.

- In order to maintain compatibility with existing device drivers, DOS XMS drivers must not hook INT 15h until the first non-Version Number call to the control function is made.

- XMS drivers are required to check for the presence of drivers which use the IBM VDISK allocation scheme. Note that it is not sufficient to check for VDISK users at installation time but at the time when the HMA is first allocated. If a VDISK user is detected, the HMA must not be allocated. Microsoft will publish a standard method for detecting drivers which use the VDISK allocation scheme.

- XMS drivers which have a fixed number of extended memory handles (most do) should implement a command line parameter for adjusting that number (suggested name "/NUMHANDLES=")

- XMS drivers should make sure that the major DOS version number is greater than or equal to 3 before installing themselves.

- UMBs cannot occupy memory addresses that can be banked by EMS 4.0. EMS 4.0 takes precedence over UMBs for physically addressable memory.

- All driver functions must be re-entrant. Care should be taken to not leave interrupts disabled for long periods of time.

- Allocation of a zero length extended memory buffer is allowed. Programs which hook XMS drivers may need to reserve a handle for private use via this method. Programs which hook an XMS driver should pass all requests for zero length EMBs to the next driver in the chain.

- Drivers should control the A20 line via an "enable count." Local Enable only enables the A20 line if the count is zero. It then increments the count. Local Disable only disables A20 if the count is one. It then decrements the count. Global Enable/Disable keeps a flag which indicates the state of A20. They use Local Enable/Disable to actually change the state.

- Drivers should always check the physical A20 state in the local Enable-Disable calls, to see
that the physical state matches the internal count. If the physical state does not match, it should
be modified so that it matches the internal count. This avoids problems with applications that
modify A20 directly.

IMPLEMENTATION OF CODE FOR HOOKING THE XMS DRIVER:

In order to support the hooking of the XMS driver by multiple

pieces of code, the following code sample should be followed.
Use of other methods for hooking the XMS driver will not work
in many cases. This method is the official supported one.

The basic strategy is:

  Find the XMS driver header which has the "near jump" dispatch.

  Patch the near jump to a FAR jump which jumps to my HOOK XMS
      driver header.

NOTES:

  o This architecture allows the most recent HOOKer to undo his
      XMS driver hook at any time without having to worry about
      damaging a "hook chain".

  o This architecture allows the complete XMS hook chain to be
      enumerated at any time. There are no "hidden hooks".

  o This architecture allows the HOOKer to not have to worry
      about installing an "INT 2F hook" to hook the AH=43h
      INT 2Fs handled by the XMS driver. The base XMS driver
      continues to be the only one installed on INT 2Fh AH=43h.

      This avoids all of the problems of undoing a software
      interrupt hook.

```
;
; When I wish to CHAIN to the previous XMS driver, I execute a FAR JMP
;     to the address stored in this DWORD.
;
PrevXMSControlAddr    dd      ?

;
; The next two data items are needed ONLY if I desire to be able to undo
;     my XMS hook.
; PrevXMSControlJmpVal stores the previos XMS dispatch near jump offset
;     value that is used to unhook my XMS hook
; PrevXMSControlBase stores the address of the XMS header that I hooked
;
PrevXMSControlBase    dd      ?
PrevXMSControlJmpVal  db      ?

;
; This is MY XMS control header.
;
MyXMSControlFunc proc FAR
      jmp     short XMSControlEntry
      nop
      nop
      nop
XMSControlEntry:

......

Chain:
```

```
      jmp     cs:[PrevXMSControlAddr]

MyXMSControlFunc endp


.......
;
; This is the code which installs my hook into the XMS driver.
;
  ;
  ; See if there is an XMS driver to hook
  ;
      mov     ax,4300h
      int     2Fh
      cmp     al,80h
      jne     NoXMSDrvrToHookError
  ;
  ; Get the current XMS driver Control address
  ;
      mov     ax,4310h
      int     2Fh
NextXMSHeader:
      mov     word ptr [PrevXMSControlAddr+2],es
      mov     word ptr [PrevXMSControlBase+2],es
      mov     word ptr [PrevXMSControlBase],bx
      mov     cx,word ptr es:[bx]
      cmp     cl,0EBh                         ; Near JUMP
      je      ComputeNearJmp
      cmp     cl,0EAh                         ; Far JUMP
      jne     XMSDrvrChainMessedUpError
ComputeFarJmp:
      mov     si,word ptr es:[bx+1]           ; Offset of jump
      mov     es,word ptr es:[bx+1+2]         ; Seg of jump
      mov     bx,si
      jmp     short NextXMSHeader

ComputeNearJmp:
      cmp     word ptr es:[bx+2],9090h        ; Two NOPs?
      jne     XMSDrvrChainMessedUpError       ; No
      cmp     byte ptr es:[bx+4],90h          ; Total of 3 NOPs?
      jne     XMSDrvrChainMessedUpError       ; No
      mov     di,bx                           ; Save pointer to header
      xor     ax,ax
      mov     al,ch                           ; jmp addr of near jump
      mov     [PrevXMSControlJmpVal],al
      add     ax,2                            ; NEAR JMP is 2 byte instruction
      add     bx,ax                           ; Target of jump
      mov     word ptr [PrevXMSControlAddr],bx
  ;
  ; Now INSTALL my XMS HOOK
  ;
      cli                             ; Disable INTs in case someone calls
                                      ;       XMS at interrupt time
      mov     byte ptr es:[di],0EAh   ; Far Immed. JUMP instruction
      mov     word ptr es:[di+1],offset MyXMSControlFunc
      mov     word ptr es:[di+3],cs
      sti
```

```
    .....

    ;
    ; Deinstall my XMS hook. This can be done IF AND ONLY IF my XMS header
    ;   still contains the near jump dispatch
    ;
        cmp     byte ptr [MyXMSControlFunc],0EBh
        jne     CantDeinstallError
        mov     al,0EBh
        mov     ah,[PrevXMSControlJmpVal]
        les     bx,[PrevXMSControlBase]
        cli                             ; Disable INTs in case someone calls
                                        ;       XMS at interrupt time
        mov     word ptr es:[bx],ax
        mov     word ptr es:[bx+2],9090h
        mov     byte ptr es:[bx+4],90h
        sti
    ....
```

IMPLEMENTATION NOTES FOR HIMEM.SYS:

- HIMEM.SYS currently supports true AT-compatibles, 386 AT machines, IBM PS/2s, AT&T 6300 Plus systems and Hewlett Packard Vectras.

- If HIMEM finds that it cannot properly control the A20 line or if there is no extended memory available when HIMEM.SYS is invoked, the driver does not install itself. HIMEM.SYS displays the message "High Memory Area Unavailable" when this situation occurs.

- If HIMEM finds that the A20 line is already enabled when it is invoked, it will NOT change the A20 line's state. The assumption is that whoever enabled it knew what they were doing. HIMEM.SYS displays the message "A20 Line Permanently Enabled" when this situation occurs.

- HIMEM.SYS is incompatible with IBM's VDISK.SYS driver and other drivers which use the VDISK scheme for allocating extended memory. However, HIMEM does attempt to detect these drivers and will not allocate the HMA if one is found.

- HIMEM.SYS supports the optional "/HMAMIN=" parameter. The valid values are decimal numbers between 0 and 63.

- By default, HIMEM.SYS has 32 extended memory handles available for use. This number may be adjusted with the "/NUMHANDLES=" parameter. The maximum value for this parameter is 128 and the minimum is 0. Each handle currently requires 6 bytes of resident space.