Microsoft
SYSTEMS JOURNAL
REPRINT

21

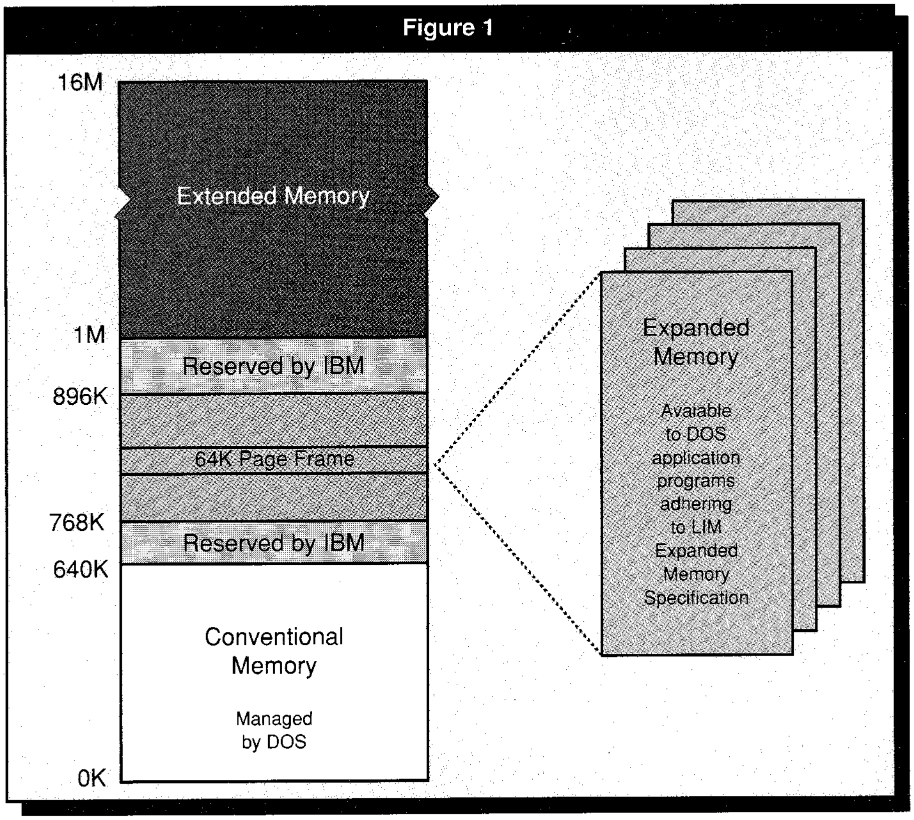
Expanded Memory: Writing Programs That Break the 640K Barrier

Marion Hansen, Bill Krueger, and Nick Stuecklen

When the size of conventional memory was set at 640K, that seemed like all the memory that anyone with a PC could ever use. But as programs written for MS-DOS grew larger, and the amount of data they could handle increased, what had once seemed inexhaustible pinched like a pair of size 8 shoes on size 10 feet. Swapping to disk, or the use of overlays, is a solution, but it often limits performance to unacceptable levels.

That's why Lotus Development Corp., Intel Corp., and Microsoft Corp. got together to do something about DOS's 640K memory limit. Together they came up with the Lotus/Intel/Microsoft Expanded Memory Specification (EMS). The programming examples accompanying this article use the EMS and will run under the AST Research Enhanced Expanded Memory Specification (EEMS), a variation of the EMS, as well.

Expanded memory is memory beyond DOS's 640K limit. Just as DOS manages conventional memory, the Expanded Memory Manager (EMM) manages expanded memory. The EMM can manage up to 8 megabytes (MB) of expanded memory. Programs that adhere to the EMS can use expanded memory without fear of conflict.



▲ **Figure 1** **The Lotus/Intel/Microsoft EMS defines a 64K segment of memory that resides between 640K and 1MB.**

Marion Hansen is a technical writer and editor. She currently writes manuals for Intel's Personal Computer Enhancement Operation. Bill Krueger is a senior software engineer with Intel's Personal Computer Enhancement Operation and helped develop and implement the Lotus/Intel/Microsoft Expanded Memory Specification. Nick Stuecklen is a senior software engineer with a B.S. in computer science.

Figure 2: EMM Functions

Function Number	Function Name	AX Register	Action
1	AH: 40	Get Status	Returns a status code to tell you whether the EMM is present and the hardware/software is working correctly.
2	AH: 41	Get Page Frame Address	Gives the program the location of the page frame.
3	AH: 42	Get Unallocated Page Count	Tells the program the number of unallocated pages and the total number of pages in expanded memory.
4	AH: 43	Allocate Pages	Allocates the number of expanded memory pages requested by the program; assigns a unique EMM handle to the set of pages allocated.
5	AH: 44	Map Handle Page	Maps the specified logical page in expanded memory to the specified physical page within the page frame.
6	AH: 45	Deallocate Pages	Deallocates the pages currently allocated to an EMM handle.
7	AH: 46	Get EMM Version	Returns the version number of the EMM software.
8	AH: 47	Save Page Map	Saves the contents of the page mapping registers of all expanded memory boards.
9	AH: 48	Restore Page Map	Restores the contents of the page mapping registers.
10	AH: 49		Reserved.
11	AH: 4A		Reserved.
12	AH: 4B	Get EMM Handle Count	Returns the number of active EMM handles.
13	AH: 4C	Get EMM Handle Pages	Returns the number of pages allocated to a specific EMM handle.
14	AH: 4D	Get All EMM Handle Pages	Returns the active EMM handles and the number of pages allocated to each one.
15	AH: 4E; AL: 00 AH: 4E; AL: 01 AH: 4E; AL: 02	Get/Set Page Map	Saves and restores the mapping context of the active EMM handle.

▲ **Figure 2** **EMM functions provide the tools that application programs need to use expanded memory.**

Contrary to what you may have heard, you can put code as well as data into expanded memory. Programs can store anything in expanded memory except their stacks, which should reside in conventional memory. While placing the stack in expanded memory is theoretically possible, managing a paged stack is generally very difficult.

Expanded memory is implemented in one of two ways. One way is an expanded memory board, where expanded memory physically resides on an add-in board. Intel's Above™ Board and AST's Advantage™ are examples of expanded memory boards. The other way is a LIMulator, such as the Compaq Deskpro 386's CEMM (Compaq Expanded Memory Manager), running on a 386-based system. A LIMulator emulates expanded memory in extended memory (which is memory from 1MB to 16MB) using the 80386 paging hardware.

Application programs can't use expanded memory automatically. This article explains how to write programs that take advantage of expanded memory, including programming techniques and examples, and the EMM functions.

Expanded Memory

In the current DOS environment, code and data can reside in one of three memory locations. Each memory type has advantages and disadvantages.

Conventional Memory: Conventional memory is always available, except whatever is used by application programs and resident software, and it's easily accessible. Moving about in conventional memory, whether through code or data, requires very little overhead. Segment register updates (when the software crosses segment boundaries) are the only substantial software overhead. Segment register updates are common to all three types of memory and as such are not a limitation unique to conventional memory. Conventional memory's drawback is its 640K limit. Large application programs, network software, and resident spelling checkers, to name just three types of software a typical user might have, consume prodigious amounts of conventional memory.

Disk Memory: There's more than enough room on a disk for any software, but the constant paging in and out of data and code in even the simplest applications creates a great deal of overhead. This makes disk memory undesirable for speed-sensitive applications.

DOS is not re-entrant, and you can invoke a terminate-and-stay-resident (TSR) program in the middle of a DOS function. For this reason, TSR programs sometimes have difficulties using DOS for disk I/O.

Expanded Memory: Like conventional memory, expanded memory is nearly always available. And with fully populated expanded memory boards, it is sufficient for most applications. Accessing expanded memory requires slightly more overhead than accessing conventional memory but significantly less overhead than accessing disk memory. When an application stays within a single 64K page, expanded memory overhead is comparable to conventional memory overhead.

Expanded memory is especially suitable for four types of software: TSR programs, graphics packages, databases, and network software.

TSR programs permanently consume the memory they occupy. If a TSR program is large in code or data, it consumes a great deal of conventional memory. A TSR pro-

Figure 3: Main Program

```
#include <dos.h>
#include <stdio.h>

#define EMM_INT 0x67                 /*  EMM interrupt number  */
#define GET_PAGE_FRAME_BASE 0x41     /*  EMM func = get page frame
                                         base address  */
#define GET_FREE_COUNT 0x42          /*  EMM Func = get unallocated
                                         pages count  */
#define ALLOCATE_PAGES 0x43          /*  EMM Func = allocates pages  */
#define MAP_PAGES 0x44               /*  EMM Func = map pages  */
#define DEALLOCATE_PAGES 0x45        /*  EMM Func = deallocate pages  */
#define GET_INT_VECTOR 0x35          /*  DOS func = get interrupt
                                         vector  */
#define DEVICE_NAME_LEN 8            /*  Number of chars in device
                                         driver name field  */
#define VIDEO_RAM_SIZE 4000          /*  Total bytes in video RAM
                                         (char/attr)  */
#define VIDEO_RAM_BASE 0xB0000000    /*  Video RAM start address (MDA) */

union REGS input_regs, output_regs; /*  Regs used for calls to EMM
                                         and DOS  */
struct SREGS segment_regs;
unsigned int emm_status;             /*  Status returned by EMM  */

main ()

{
unsigned int i;
long target_time, current_time;
char *video_ram_ptr = {VIDEO_RAM_BASE}; /*  Pointer to video RAM  */
unsigned int emm_handle;                /*  EMM handle  */
char *expanded_memory_ptr;              /*  Pointer to expanded
                                            memory  */

/*  Ensure that the Expanded Memory Manager software is installed
    on the user's system.  */

   detect_emm();

/*  Get a page of expanded memory.  */

   get_expanded_memory_page (&expanded_memory_ptr, &emm_handle);

/*  Copy the current video RAM contents to expanded memory.  */

   memcpy (expanded_memory_ptr, video_ram_ptr, VIDEO_RAM_SIZE);

/*  Clear the screen to nulls.  */

   memset (video_ram_ptr, '\0', VIDEO_RAM_SIZE);

/*  Delay for 1 second so the user can see the blanked screen.  */

   time (&current_time);
   target_time = current_time + 1;
   while (current_time < target_time)
    {
     time (&current_time);
    }

/*  Restore the video RAM contents from expanded memory.  */

    memcpy (video_ram_ptr, expanded_memory_ptr, VIDEO_RAM_SIZE);

    /*  Deallocate the expanded memory page  */

       release_expanded_memory_page (emm_handle);

       exit(0);
    }
```

▶ **Figure 3 The main program allocates one 16K page of expanded memory, saves the video RAM area to expanded memory, clears the screen and then restores the screen to expanded memory.**

gram that is designed to use expanded memory effectively keeps most of its code and data in expanded memory, while maintaining a small kernel in conventional memory for housekeeping chores, such as trapping interrupts, and activating the rest of the TSR program in expanded memory.

Drawing and drafting packages frequently have to maintain multiple copies of their graphics bit map. Secondary drawings, double buffers for animations, and additional menus are all stored for later retrieval. Because recall speed is essential, these bit maps must be maintained in memory. Just one monochrome (1 bit per pixel) bit map with 640-by-350 resolution requires nearly 28K of storage. Several such bit map copies can eat up conventional memory, but they are easily accommodated in expanded memory.

Database programs sort huge volumes of data, typically much more than conventional memory are able to handle. Expanded memory can be used to store and sort large databases and is much faster than swapping to disk.

Network software creates large tables and volumes of resident data. Although network software may be used infrequently—usually just for peripheral sharing and file transfers—it can consume up to 50 percent of available conventional memory. Putting network software in expanded

Figure 4: Detect_EMM Subprocedure

```
detect_emm ()

{
static char EMM_device_name [DEVICE_NAME_LEN] = {"EMMXXXX0"};
char *int_67_device_name_ptr;


/*  Determine the address of the routine associated with INT 67 hex.  */

  input_regs.h.ah = GET_INT_VECTOR;  /*  DOS function  */
  input_regs.h.al = EMM_INT;         /*  EMM interrupt number  */
  intdosx (&input_regs, &output_regs, &segment_regs);
  int_67_device_name_ptr =
  (segment_regs.es * 65536) + 10;    /*  Create ptr to device name
                                         field  */

/*  Compare the device name with the known EMM device name.  */

  if(memcmp(EMM_device_name,int_67_device_name_ptr,DEVICE_NAME_LEN) !=0)
   {
     printf ("\x07Abort: EMM device driver not installed\n");
     exit(0);
   }
}
```

▲ **Figure 4** **The detect_emm subprocedure determines whether the EMM driver software is installed.**

Figure 5: Check_Status Subprocedure

```
check_status (emm_status)
unsigned int emm_status;
{
static char *emm_error_strings[] = {
  "no error",
  "EMM software malfunction",
  "EMM hardware malfunction",
  "RESERVED",
  "Invalid EMM handle",
  "Invalid EMM function code",
  "All EMM handles being used",
  "Save/restore page mapping context error",
  "Not enough expanded memory pages",
  "Not enough unallocated pages",
  "Can not allocate zero pages",
  "Logical page out of range",
  "Physical page out of range",
  "Page mapping hardware state save area full",
  "Page mapping hardware state save area already has handle",
  "No handle associated with the page mapping hardware state save area",
  "Invalid subfunction"
 };

/*  IF EMM error, THEN print error message and EXIT  */

   if (emm_status != 0)                    /*  IF EMM error...  */
    {
     emm_status -= 0x7F;                   /*  Make error code
                                               zero-based  */
     printf ("\x07Abort: EMM error = ");  /*  Issue error prefix  */
     printf ("%s\n", emm_error_strings[emm_status]);
                                           /*  Issue actual error
                                               message  */
     exit(0);                              /*  And then exit to
                                               DOS  */
    }
}
```

◄ **Figure 5**

The check_status subprocedure is called after each EMM function to make sure that no EMM errors have occurred.

memory frees conventional memory for software that you use more frequently.

Using application software efficiently is a trade-off between the convenience of generous amounts of expanded memory and the overhead of paging in 64K blocks of it at a time. You should consider two questions when deciding whether to use expanded or conventional memory for your applications.

First, does the code execute a large number of far calls or jumps relative to the time it spends executing other instructions? If it does, put the code in conventional memory. If it doesn't, put the code in expanded memory.

Second, does the application's data require segment register initialization each time it is accessed? If it does, use conventional memory. If it doesn't use expanded memory.

As a rule of thumb, use expanded memory if both the time spent using data or executing code and the preparation overhead are large.

The Page Frame

Expanded memory is managed the same way, whether it resides on an add-in board or is emulated in extended memory. The Lotus/Intel/Microsoft EMS defines a 64K segment of memory that resides between 640K and 1MB. This page frame is a window into expanded memory (see **Figure 1**).

Just after the application program starts executing, it allocates a certain number of 16K pages of expanded memory for its own use. Four pages of expanded memory can be mapped into the expanded memory page frame at one time. By mapping pages in and out of the page frame, the program can access any area of the expanded memory that it allocated.

The EEMS allows the page frame to reside at any unused memory address between 0K and 1,024K. Theoretically, this allows a page frame length of

Figure 6: Get_Expanded_Memory_Page Subprocedure

```
get_expanded_memory_page (expanded_memory_ptr_ptr, emm_handle_ptr)

unsigned int *emm_handle_ptr;      /*  16 bit handle returned by EMM  */
char *(*expanded_memory_ptr_ptr); /*  Pointer to expanded memory
                                       page  */

{
unsigned int page_frame_base;      /*  Expanded memory page frame
                                       base  */
unsigned int physical_page = {0}; /*  Physical page number  */


/*  Get unallocated pages count.  */

input_regs.h.ah = GET_FREE_COUNT;     /*  EMM function  */
int86x (EMM_INT, &input_regs, &output_regs, &segment_regs);
emm_status = output_regs.h.ah;
check_status(emm_status);             /*  Check for errors  */
if (output_regs.x.bx < 1)             /*  Check unallocated page
                                          count  */
 {
  printf ("\x07Abort: insufficient unallocated expanded memory pages\n");
  exit(0);
 }


/*  Allocate the specified number of pages.  */

input_regs.h.ah = ALLOCATE_PAGES;      /*  EMM function  */
input_regs.x.bx = 1;                   /*  Number of pages to
                                           allocate  */
int86x (EMM_INT, &input_regs, &output_regs, &segment_regs);
emm_status = output_regs.h.ah;
check_status(emm_status);              /*  Check for errors  */
*emm_handle_ptr = output_regs.x.dx;    /*  Get EMM handle  */


/*  Map the logical page into physical page 0.  */

   input_regs.h.ah = MAP_PAGES;           /*  EMM function  */
   input_regs.h.al = 0;                   /*  Logical page number  */
   input_regs.x.bx = physical_page;       /*  Physical page number  */
   input_regs.x.dx = *emm_handle_ptr;     /*  EMM handle  */
   int86x (EMM_INT, &input_regs, &output_regs, &segment_regs);
   emm_status = output_regs.h.ah;
   check_status(emm_status);              /*  Check for errors  */


/*  Determine the page frame address.  */

   input_regs.h.ah = GET_PAGE_FRAME_BASE; /*  EMM function  */
   int86x (EMM_INT, &input_regs, &output_regs, &segment_regs);
   emm_status = output_regs.h.ah;
   check_status(emm_status);               /*  Check for errors  */
   *expanded_memory_ptr_ptr =
     (output_regs.x.bx * 65536)
     + (physical_page * 16 * 1024);        /*  Set the expanded memory
                                               ptr  */
}
```

▲ **Figure 6** **The get_expanded_memory_page subprocedure returns a pointer to the expanded memory page and a 16-bit tag or handle associated with that page.**

Figure 7: Release_Expanded_Memory_Page Subprocedure

```
release_expanded_memory_page (emm_handle)

unsigned int emm_handle;        /*  Handle identifying which page
                                    set to deallocate  */
{

/*  Release the expanded memory pages by deallocating the handle
    associated with those pages.  */

   input_regs.h.ah = DEALLOCATE_PAGES;  /*  EMM function  */
   input_regs.x.dx = emm_handle;        /*  EMM handle passed in
                                            DX  */
   int86x (EMM_INT, &input_regs, &output_regs, &segment_regs);
   emm_status = output_regs.h.ah;
   check_status(emm_status);            /*  Check for errors  */
}
```

▲ **Figure 7**

The release_expanded_memory_page subprocedure releases the expanded memory pages by de-allocating the handle associated with those pages.

PRACTICAL CONSIDERATIONS, SUCH AS DOS AND APPLICATION PROGRAMS, WHICH USE CONVENTIONAL MEMORY, AND THE BIOS AND ROM ON ADD-IN BOARDS, WHICH USE MEMORY ABOVE 640K, RESTRICT THE PAGE FRAME TO FEWER THAN THE POSSIBLE 64 PAGES.

1MB. Practical considerations, such as DOS and application programs, which use conventional memory, and the BIOS and ROM on add-in boards, which use memory above 640K, restrict the page frame to fewer than the possible 64 pages. Generally, in a typical AT system with an EGA, the maximum number of mappable pages that DOS doesn't rely on is six 16K pages.

When the EMM software is installed, the user selects where in memory (above 640K) the page frame resides. The page frame address is user-selectable, so that if another device uses memory at a particular address, the user can then relocate the page frame.

Checking for Memory

Before an application program can use expanded memory, it must determine if expanded memory and the EMM are present. There are two methods of determining if the EMM is present: the open-handle technique and the get-interrupt-vector technique.

Because the EMM is implemented as a device driver, in the open-handle technique the program issues an open handle command (DOS function 3FH) to determine whether the EMM device driver is present.

In the get-interrupt-vector technique, the program issues a get-interrupt-vector command (DOS function 35H) to get the contents of interrupt vector array entry number 67H. The pointer thus obtained accesses information that tells the program whether the EMM is installed. The get-interrupt-vector technique is easier to implement. Most programs can use either technique, but if a program is a device driver or if it interrupts DOS during file system operations, it must use the get-interrupt-vector technique.

Residents, Transients

Application programs that use expanded memory can be classified as either resident or transient. A transient application program is resident only as long as it executes. When it is finished running, the memory it used is available for other programs. Examples of resident application programs include spreadsheets, word processors, and compilers.

A resident application program remains in memory after it executes. Resident application programs are usually invoked by a hardware interrupt, such as a keystroke, or a software interrupt, such as a RAMdisk. Pop-up desktop programs, RAMdisk drives, and print spoolers are examples of resident application programs.

Resident programs and transient programs handle expanded memory differently. Resident programs may interrupt transient programs that might be using expanded memory, so resident programs must save and restore the state of the page-mapping registers when they use expanded memory.

Transient programs don't interrupt other programs, so they

▲ **Figure 8**

The pseudo-overlay is loaded into expanded memory by the kernel. The kernel then calls the initialization procedure within the pseudo-overlay. It is the initilization procedure within the pseudo-overlay that returns a data structure to the kernel. The data structure describes the first object that will be located in expanded memory starting at the page frame segment address. It contains the data and extra segments of the pseudo-overlay, the number of subprocedure entry points in the pseudo-overlay, and a list of far pointers to each of the subprocedures contained in the pseudo-overlay. The developer must establish a convention for the sequence of the far pointers and what the procedures they point to do. Other information could be passed in this structure as well, for example, number and types of parameters that are required by the subprocedures in the pseudo-overlay. This example uses a literal to determine the maximum number of far pointers that may be passed. To allocate additional space for a larger number of entries, simply increase the value of max_proc_entries. The example assumes a maximum of 64 entries can be returned.

Figure 8: Kernel Module

```
CODE    SEGMENT PARA PUBLIC 'CODE'
ORG     100h
ASSUME  CS:CODE, DS:DATA, ES:NOTHING, SS:STACK

max_proc_entries              EQU        64

pseudo_over_struct            STRUC
      proc_data_segment       DW         ?
      proc_extra_segment      DW         ?
      proc_entry_count        DW         ?
      proc_entry_ptr          DD max_proc_entries DUP (?)
pseudo_over_struct            ENDS

main  PROC  NEAR

      MOV   AX, DATA                          ; Segment initialization
      MOV   DS, AX

check_for_emm_loaded:
      CALL  test_for_EMM                      ; Use the "interrupt vector"
      JE    get_emm_page_frame                ; technique to determine
      JMP   emm_err_exit                      ; whether EMM is loaded

get_emm_page_frame:
      MOV   AH, 41h                           ; Get the page frame base
      INT   67h                               ; address from EMM
      OR    AH, AH
      JZ    allocate_64K
      JMP   emm_err_exit

allocate_64K:
      MOV   exp_mem_segment, BX               ; Allocate 4 pages of expand-
      MOV   AH, 43h                           ; ed memory for this example.
      MOV   BX, 4                             ; More can be allocated de-
      INT   67h                               ; pending on the number of
      OR    AH, AH                            ; overlays to be loaded.
      JZ    map_64K                           ; Actually, in this case,
      JMP   emm_err_exit                      ; only a single page is re-
                                              ; quired because the example
                                              ; pseudo-overlay is extreme-
                                              ; ly small.
map_64K:
      MOV   handle, DX                        ; Map in the first 4 logical
      MOV   CX, 4                             ; pages at physical pages 0
map_pages_loop:                               ; through 3
      MOV   AH, 44h                           ;    logical page 0 at
      MOV   BX, CX                            ;       physical page 0
      DEC   BX                                ;    logical page 1 at
      MOV   AL, BL                            ;       physical page 1
      MOV   DX, handle                        ;    logical page 2 at
      INT   67h                               ;       physical page 2
      OR    AH, AH                            ;    logical page 3 at
      LOOPE map_pages_loop                    ;       physical page 3
      JE    init_load_struct                  ; If additional overlays were
      JMP   emm_err_exit                      ; required, each overlay
                                              ; would be loaded after map-
                                              ; ping and a new set of
                                              ; logical pages would be
                                              ; mapped at the same
                                              ; physical pages.

init_load_struct:
      MOV   ES, exp_mem_segment               ; Initialize pseudo-overlay
      MOV   DI, 0                             ; environment and procedure
      MOV   CX, (SIZE pseudo_over_struct)     ; pointer area. This struc-
      MOV   AL, 0                             ; ture begins at the page
      REP   STOSB                             ; frame segment address.

      MOV   AX, (SIZE pseudo_over_struct)     ; Compute the load address
      ADD   AX, 000Fh                         ; within expanded memory for
```

CONTINUED

Figure 8: Kernel Module

CONTINUED

```
         AND    AX, 0FFF0h                    ; the overlay. The address is
         MOV    CX, 4                         ; rounded up to the next
         SHR    AX, CL                        ; higher paragraph boundary
         ADD    AX, exp_mem_segment           ; immediately following the
         MOV    parm_block.load_segment, AX   ; pseudo-overlay environment
         MOV    parm_block.reloc_factor, AX   ; & procedure pointer
                                              ; structure. This computa-
                                              ; tion takes into account
                                              ; the maximum number of
                                              ; procedure entry points
                                              ; which the pseudo-overlay
                                              ; is going to return to
                                              ; this program.

         MOV    WORD PTR entry_point[0], 100h ; Build .COM file entry
         MOV    WORD PTR entry_point[2], AX   ; point

         MOV    AH, 4Bh                       ; Load the pseudo-overlay
         MOV    AL, 03h                       ; using the DOS "load
         LEA    DX, pseudo_over_name          ; overlay" function
         PUSH   DS
         POP    ES
         LEA    BX, parm_block
         INT    21h
         JC     emm_err_exit

         PUSH   DS                            ; Transfer control to the
         PUSH   ES                            ; loaded pseudo-overlays
         CALL   DWORD PTR entry_point         ; initialization code
         POP    ES
         POP    DS
         OR     AH, AH
         JZ     call_over_procedures
         JMP    emm_err_exit

call_over_procedures:
         MOV    ES, exp_mem_segment           ; As an example of passing
         MOV    BX, 0                         ; control to a procedure
         MOV    DI, 0                         ; existing in expanded
         MOV    CX, ES:[BX].proc_entry_count  ; memory, each procedure con-
         JCXZ   deallocate_exp_memory         ; tained in the overlay will
                                              ; be called in sequence.
                                              ; Obviously, a single pro-
                                              ; cedure could be called
                                              ; just as easily.

pseudo_over_call_loop:
         PUSH   BX
         PUSH   CX
         PUSH   DI
         PUSH   ES
         PUSH   DS

         LDS    AX, ES:[BX+DI].proc_entry_ptr
         MOV    WORD PTR CS:tp_ent_ptr[0], AX
         MOV    WORD PTR CS:tp_ent_ptr[2], DS

         MOV    AX, 123                       ; Pass 2 numbers to
         MOV    DX, 23                        ; the procedures

         MOV    DS, ES:[BX].proc_data_segment  ; Set up pseudo-overlays
         MOV    ES, ES:[BX].proc_extra_segment ; segment environment
         CALL   DWORD PTR CS:tp_ent_ptr        ; Call each procedure

         POP    DS
         POP    ES
         POP    DI
         POP    CX
         POP    BX
```

CONTINUED

don't need to save and restore state. A resident program typically keeps the EMM handles assigned to it and the expanded memory pages allocated to it by the EMM until the system is rebooted. A transient program, in contrast, should return its handle and pages just before it exits to DOS.

EMM Functions

The EMM functions, summarized in **Figure 2**, provide the tools that application programs need to use expanded memory. Functions 1 through 7 are general-purpose functions. Functions 8 and 9 are for interrupt service routines, device drivers, and other memory-resident software. Functions 10 and 11 are reserved. Functions 12 through 14 are for utility programs. Finally, Function 15 is for multitasking operating systems, although it can be used for interrupt service routines as easily as Functions 8 and 9.

To use expanded memory, programs must perform these steps in the following order:

1. Check for the presence of the EMM by using the get-interrupt-vector or open-handle techniques.
2. Check whether the EMM's version number is valid (only if the application is EMM version–specific)—Function 7 (Get EMM Version).
3. Determine if enough unallocated expanded memory pages exist for the program—Function 3 (Get Unallocated Page Count).
4. Save the state of expanded memory hardware (only if it is a resident program)—Function 8 (Save Page Map) or Function 15 (Get/Set Page Map).
5. Allocate the number of 16K expanded memory pages

Figure 8: Kernel Module

CONTINUED

```
   ADD   DI, 4                          ; Adjust index to the next
      LOOP  pseudo_over_call_loop          ; procedure (4 bytes long)
                                           ; pointer & loop till all
                                           ; have been called


deallocate_exp_memory:
      MOV   AH, 45h                        ; Return the allocated
      MOV   DX, handle                     ; pages to the expanded
      INT   67h                            ; memory manager
      OR    AH, AH
      JNZ   emm_err_exit

exit:
      MOV   AH, 4Ch                        ; Return a normal exit code
      MOV   AL, 0
      INT   21h

emm_err_exit:
      MOV   AL, AH                         ; Display the fact that
      MOV   AH, 09h                        ; an EMM error occurred
      LEA   DX, emm_err_msg                ; Go to the normal exit
      INT   21h
      JMP   exit

      tp_ent_ptr            DD         ?   ; CS relative far pointer
                                           ; used for transfer to the
main  ENDP                                 ; procedures in the
                                           ; pseudo_overlay
```

Figure 9: Procedure to Test for the Presence of EMM

```
test_for_EMM    PROC NEAR

       MOV     AX, 3567h              ; Issue "get interrupt vector"
       INT     21h

       MOV     DI, 000Ah              ; Use the SEGMENT in ES
                                      ; returned by DOS, place
                                      ; the "device name field"
                                      ; OFFSET in DI.

       LEA     SI, EMM_device_name    ; Place the OFFSET of the EMM
                                      ; device name string in SI,
                                      ; the SEGMENT is already in DS.

       MOV     CX, 8                  ; Compare the name strings
       CLD                            ; Return the status of the
       REPE    CMPSB                  ; compare in the ZERO flag
       RET

test_for_EMM ENDP

CODE        ENDS
```

▲ **Figure 9** **This procedure tests for the presence of the EMM in the system. The carry flag is set if the EMM is present. The carry flag is clear if the EMM is not present.**

needed by the program—Function 4 (Allocate Pages).

6. Map the set of expanded memory pages (up to four) into the page frame—Function 5 (Map Handle Page).

7. Determine the expanded memory page frame base address—Function 2 (Get Page Frame Address).

8. Read/write to the expanded memory segment within the page frame, just as you read or write to conventional memory.

9. Deallocate the expanded memory pages when the program is finished using them—Function 6 (Deallocate Pages).

10. Restore the state of expanded memory hardware (only if it is a memory-resident program)—Function 9 (Restore Page Map) or Function 15 (Get/Set Page Map).

Each EMM function's number is passed in register AX. The EMM will return the function's status in the same register.

Programs use Int 67 to invoke the EMM. This works like DOS Int 21: preload certain registers and issue an Int 67. All required registers are rigidly specified, and certain conventions exist; for example, the AX register always returns status.

Programming

The following two examples contain programs that have both code and data in expanded memory. The first example (written in *Microsoft C,* Version 3.00) illustrates how expanded memory can be used to save and restore data. The main program (see **Figure 3**) calls a series of subprocedures that allocate one 16K page of expanded memory, save the video RAM area (the user's screen) to expanded memory, clear the screen, and then restore the screen from expanded memory. The program assumes the user has a monochrome display adapter operating in text mode (nongraphics) and video page zero is displayed.

The program contains four subprocedures. The detect_emm

Figure 10: Pseudo-overlay Module

```
CODE     SEGMENT PARA PUBLIC 'CODE'
ASSUME   CS:CODE, DS:DATA
ORG      100h

actual_proc_entries         EQU        2

overlay_entry_struct        STRUC
        proc_data_segment   DW         ?
        proc_extra_segment  DW         ?
        proc_entry_count    DW         ?
        proc_entry_ptr      DD         actual_proc_entries DUP (?)
overlay_entry_struct        ENDS
```

▲ **Figure 10** The kernel loads the pseudo-overlay into expanded memory.The kernel calls the initialization procedure within the pseudo-overlay. The initialization procedure returns a data structure to the kernel. The data structure describes the first object that will be located in expanded memory starting at the page frame segment address. It contains the data and extra segments of the pseudo-overlay, the number of subprocedure entry points in the pseudo-overlay, and a list of far pointers to each of the subprocedures contained in the pseudo-overlay.

Figure 11: Procedure to Identify Overlay

```
command_line_entry_point        PROC        NEAR

        MOV     AX, DATA                    ; Set up local data
        MOV     DS, AX                      ; segment

        LEA     DX, overlay_err_msg         ; Display overlay error
        MOV     AH, 09h                     ; message
        INT     21h

        MOV     AX, 4C00h                   ; Exit back to DOS
        INT     21h

command_line_entry_point        ENDP
```

▲ **Figure 11** This procedure merely informs a user that this is the overlay and cannot be executed from the command line.

Figure 12: Data Segment for the Pseudo-overlay Module

```
DATA  SEGMENT PARA PUBLIC 'DATA'

sum_msg         DB   0Dh, 0Ah, 'Sum of numbers = ', '$'
diff_msg        DB   0Dh, 0Ah, 'Difference of numbers = ', '$'
overlay_err_msg DB   'Overlay cannot be executed via the command line$'
powers_of_ten   DW   10000, 1000, 100, 10, 1

DATA  ENDS

END   command_line_entry_point
```

▲ **Figure 12** This is the data segment for the pseudo-overlay program.

subprocedure (see **Figure 4**) determines whether the EMM software is installed. If it is installed, the subprocedure returns to the caller. If the EMM software isn't installed, the subprocedure generates an error message and exits the program.

The get_expanded_memory_page subprocedure (see **Figure 6**) returns a pointer to the expanded memory page and a 16-bit tag or handle associated with that page. The subprocedure uses the EMM to allocate a page of expanded memory. If an unallocated page exists, the procedure allocates it and maps it in and returns the EMM handle that is associated with that page.

The check_status subprocedure (see **Figure 5**) is called after each EMM function to verify that no EMM errors have occurred. The release_expanded_memory_pagesubprocedure (see **Figure 7**) releases expanded memory pages by deallocating the handle associated with those pages.

The second example illustrates one program loading another program into expanded memory, which is especially applicable for developers of terminate-and-stay-resident (TSR) applications. Both programs are written in *Microsoft Macro Assembler*, Version 4.0.

The first program, expanded_memory_dispatcher_kernel (see **Figure 8**), loads a set of subprocedures into expanded memory, from where they can be invoked at any time. The set of loaded subprocedures is called a pseudo-overlay. This program loads only one pseudo-overlay and immediately invokes all the subprocedures contained in it. You can easily load as many pseudo-overlays as you want by allocating additional pages in expanded memory, mapping up to four of the newly allocated pages into the page frame, and then loading additional pseudo-overlays.

The program has one subprocedure, test_for_EMM (see **Figure 9**), which determines whether the EMM software is installed and returns the appropriate status.

The kernel program loads the program OVERLAY.EXE (see **Figure 10**) into expanded memory. A pseudo-overlay can't be larger than 64K because of the four-page EMM page frame, so the developer must decompose the program into separate modules that contain code or data no larger than 64K. You can have up to 8MB of expanded memory and, therefore, up to 128 overlays.

Although the DOS "load overlay" function (DOS function 4B03H) is used to load the pseudo-overlays, the code and any data loaded remain resident after the load takes place. The subprocedures contained in the pseudo-overlay can be accessed by using the list of pointers returned to the kernel by the initialization code in the pseudo-overlay.

The pseudo-overlay program has five subprocedures. If the pseudo-overlay program is invoked from the command line, then the command_line_entry_point subprocedure (see **Figure 11**) tells the user that this is a pseudo-overlay and thus can't be executed.

The initialization subprocedure (see **Figure 13**) is critical. The kernel calls this subprocedure after the program is loaded. The initialization subprocedure passes back to the kernel the data segment environment, a count of the number of callable subprocedures in the

Figure 13: Pseudo-overlay Data Structure Initialization Procedure

```
initialization  PROC    FAR

MOV  AX, DATA                                            ; Set up a local
MOV  DS, AX                                              ; data segment

MOV  AH, 41h                                             ; Get the page
INT  67h                                                 ; frame segment
OR   AH, AH                                              ; address from EMM
JNZ  error

MOV  ES, BX                                              ; Create pointer
MOV  DI, 0                                               ; to the page frame
                                                         ; segment address

MOV  ES:[DI].proc_data_segment, DS                       ; Return local data
MOV  ES:[DI].proc_extra_segment, DS                      ; & extra segment
                                                         ; back to the kernel

MOV  WORD PTR ES:[DI].proc_entry_count, 2                ; Return the number
                                                         ; of call-
                                                         ; able procedures
MOV  WORD PTR ES:[DI].proc_entry_ptr[0], OFFSET sum      ; Return
MOV  WORD PTR ES:[DI].proc_entry_ptr[2], SEG    sum      ; pointer to each
MOV  WORD PTR ES:[DI].proc_entry_ptr[4], OFFSET diff     ; local callable
MOV  WORD PTR ES:[DI].proc_entry_ptr[6], SEG    diff     ; procedure in the
                                                         ; pseudo-overlay
                                                         ; back to kernel

exit:  MOV   AH, 0                                       ; Set status in AH
                                                         ; = passed
error: RET                                               ; Return status
                                                         ; in AH
```

▲ **Figure 13** **The initialization subprocedure is called by the kernel after the program is loaded. It passes to the kernel the data segment environment, a count of the number of callable subprocedures in the overlay, and a far pointer to each subprocedure.**

Figure 14: Procedure to Add AX and DX

```
sum     PROC    FAR
        ADD     AX, DX                  ; Add numbers
        PUSH    AX                      ; Display sum message
        LEA     DX, sum_msg
        MOV     AH, 09h
        INT     21h
        POP     AX
        CALL    display_result          ; Display sum
        RET
sum     ENDP
```

▲ **Figure 14** **This procedure adds AX and DX and displays the result.**

overlay, and a far pointer to each subprocedure.

The sum and diff subprocedures are examples of typical applications. The sum subprocedure (see **Figure 14**) adds the numbers in the AX and DX registers and displays the result, while the diff subprocedure (see **Figure 15**) subtracts the numbers in the AX and DX registers and displays the result. The display_result procedure (see **Figure 16**) converts the result into printable ASCII form and then displays it.

The pseudo_overlay program places data into expanded memory. The data segment for the pseudo_overlay program is shown in **Figure 12**. The common data area for both programs is shown in **Figure 17**.

To Get EMS

If you're interested in developing application programs that use expanded memory, call Intel for a free copy of the Lotus/Intel/Microsoft Expanded Memory Specification. In the continental United States, but outside Oregon, call (800) 538-3373. In Oregon, Alaska, Hawaii, or outside the United States (except Canada), call (503) 629-7354. In Canada, call (800) 235-0444. For more information on the AST EEMS, contact the AST Product Information Center at (714) 863-1480. ❏

Figure 15: Procedure to Subtract AX and DX

```
diff     PROC     FAR

         SUB      AX, DX                  ; Subtract numbers
         PUSH     AX                      ; Display difference message
         LEA      DX, diff_msg
         MOV      AH, 09h
         INT      21h
         POP      AX
         CALL     display_result          ; Display difference
         RET

diff     ENDP
```

▲ **Figure 15** **This procedure subtracts AX and DX and displays the result.**

Figure 16: Procedure to Display Number in AX in Decimal

```
display_result  PROC     NEAR

        LEA     DI, powers_of_ten
        MOV     CX, 5
display_loop:
        XOR     DX, DX              ; Divide the number passed
        DIV     WORD PTR [DI]       ; in AX by descending powers of ten
        ADD     AL, '0'             ; Convert digit to ASCII

        PUSH    DX                  ; Output the digit
        MOV     DL, AL
        MOV     AH, 02h
        INT     21h
        POP     AX

        ADD     DI, 2
        LOOP    display_loop
        RET

display_result        ENDP
```

▲ **Figure 16** **This procedure displays the number in AX in decimal.**

Figure 17:
Data and Stack Segment for the Kernel and the Pseudo-overlay

```
DATA               SEGMENT PARA PUBLIC 'DATA'

emm_err_msg         DB     'EMM error occurred$' ; EMM diagnostic message
pseudo_over_name    DB     'OVERLAY.EXE', 0      ; Name of pseudo-overlay
EMM_device_name     DB     'EMMXXXX0'            ; Standard EMM device name
exp_mem_segment     DW     ?                     ; Temp for expanded
                                                 ; memory page frame
                                                 ; segment address
handle              DW     ?                     ; Temp for handle allo-
                                                 ; cated to the kernel
entry_point  DD            ?                     ; Far pointer to the
                                                 ; entry point for a .COM
                                                 ; file
parm_block_struct   STRUC                        ; Structure definition
    load_segment    DW     ?                     ; for a "load overlay"
    reloc_factor    DW     ?                     ; parameter block
parm_block_struct   ENDS
parm_block          parm_block_struct <>         ; The actual parameter
                                                 ; block

DATA         ENDS


STACK    SEGMENT PARA STACK 'STACK'
         local_stack     DW 256 DUP ('^^')
STACK    ENDS

END        main
```

◀ **Figure 17** **This is the common data area for the kernel and pseudo-overlay programs.**