Dr. Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Undocumented Corner

Taku Okazaki unravels the undocumented Windows interface which the V86 MMGR uses to take over the page tables belonging to a 386 expanded-memory manager.

September 01, 1994
URL:http://www.drdobbs.com/windows/undocumented-corner/184409317

The Windows Global EMM Import Interface

Taku Okazaki ("taQ") is a freelance programmer in the suburbs of Tokyo. He has worked on software for the NEC PC-980x series. You can contact Taku on NIFTY:CXB00750 (CXB00750@niftyserve.or.jp) or on CompuServe at 100213,3351.

Introduction

by Andrew Schulman

I have received a lot of requests for information about an obscure undocumented interface that goes under a variety of names--the Windows/386 Paging Import specification, the Global EMM Import specification, V86MMGR Paging Import, Paging Import/Export specification, and even Paging Import Services specification, to name a few.

As Taku explains this month, this interface is what the V86MMGR (Virtual-8086 Memory Manager) virtual device driver (VxD) in Windows Enhanced mode uses to take over the page tables belonging to a 386 expanded memory manager (EMM) such as EMM386, QEMM, 386MAX, or Helix NetRoom. The page tables include not only expanded memory, but also XMS (extended memory) and UMBs (upper memory blocks).

Perhaps one reason there's so much interest in this interface is that Microsoft's Windows Device Driver Kit (DDK) mentions it in passing. For example, the description of the *_AddFreePhysPage* service provided by the Virtual Machine Manager (VMM) says "the V86MMGR device adds any unused physical pages it finds when using the Global EMM Import function of a 386 LIMulator" (a LIMulator is a paging EMM driver, and the term "Global EMM" refers to the fact that the EMM was present *before* Windows loaded and is therefore visible in all virtual machines). Similarly, the documentation for the *V86MMGR_GetPgStatus* service refers to "paging import from a LIMulator/UMBulator" (an UMBulator--ridiculous name!--is a paging XMS driver). Naturally, avid readers of the DDK wonder what these passing remarks refer to.

Microsoft has a document ("Windows/386 Paging Import Specification") available to a select number of memory-manager vendors, although at least one vendor, Novell, was unable to get the latest (Version 1.11) revision of the specification. Consequently, Novell had to "DIY" (do it yourself), at least according to an engineer at Novell's European Development Centre in Hungerford, England.

Taku's EMMIMP program (available electronically, see page 3) simply dumps out the EMM import structure in a readable form. The V86MMGR device in Windows actually uses the structure to *import* an external-memory manager's page tables. V86MMGR calls the VMM *_MapPhysToLinear* service to access the 218Ch-byte EMM import structure, then processes the structure using VMM services like *_PhysIntoV86*, *_AddFreePhysPage*, *_Assign_Device_V86_Pages*, and *Set_Physical_HMA_Alias*. These services are all documented in the DDK. Thus, the Global EMM Import is basically an undocumented interface that allows those with access to do something everyone else can do by writing a VxD.

If you have comments or suggestions for this column, please contact me on CompuServe in the Undocumented Corner area of the *Dr. Dobb's* CompuServe forum (GO DDJFORUM), where my ID is 76320,302.

Acouple of years ago, I wrote a piece of code for my NEC PC-9800 that jumped from real mode into protected mode, then came back in virtual 8086 (V86) mode. It did nothing but slow down the system, and only a system reset could get rid of it. It was effectively a V86 monitor. I started playing around with the program, adding bits of this and bytes of that. Soon I wasn't playing with it but working on it, and for the following months, quite a bit of my real work was compromised. In the end, I had a full-blown memory manager that supported EMS, XMS, UMB, BIOS ROM remapping, VCPI, and DPMI--almost everything a memory manager needs, with one exception: It couldn't run Windows.

Even though in Japan Windows is not as popular on the PC-9800 as on the PC, I knew that if I were going to sell my V86 memory manager, distribute it as shareware, or do anything else with that half-year's worth of work, my memory manager had to run Windows.

A V86 memory manager like mine runs at the most-privileged level (Ring 0) of protected mode. Windows Enhanced mode also wants to run in Ring 0. Only one program at a time can run in Ring 0, so when Windows starts up, the memory manager must give up control of protected mode.

The Windows DDK documents how Windows can take control of protected mode away from a memory manager. The memory manager can hook INT 2Fh; when Windows starts, it calls INT 2Fh function 1605h, and the memory manager can return the address of an "Enable/Disable Virtual 8086 Mode callback function" in DS:SI. Function 1605h is a very overloaded interface: You can also use it to tell Windows whether it can start up, to identify instance data (see "Undocumented Corner," *DDJ*, April 1994), and to identify virtual device drivers (VxDs) that need loading.

On startup, Windows calls the memory manager's V86 enable/disable function with AX=0. The memory manager switches the CPU from V86 mode into real mode. Windows can then enter protected mode. When Windows exits, it sets the CPU into real mode, and calls the function with AX=1. The memory manager then regains control of protected mode.

So all I needed to do was give my memory manager a mode-switch entry point, and return its address in DS:SI when Windows called INT 2Fh function 1605h. Easy! Now, did Windows run? Partially.

Everything looked okay, except there were no upper memory blocks (UMBs) and no expanded memory (EMS) in the DOS box, even though they were there before Windows started! Why did they seem to disappear under Windows?

I should have expected this. A memory manager uses the 386 paging mechanism to create UMBs and virtual EMS. My memory manager knew which physical pages were used to do that, but I hadn't told any of this to Windows.

One way to carry over UMBs and EMS into Windows would be to use a VxD. But a simple experiment with Microsoft's memory manager, EMM386, showed that I didn't need a VxD. The VxD associated with EMM386 is embedded right inside EMM386.EXE, so if I booted with EMM386 installed and then temporarily renamed EMM386.EXE, Windows wouldn't be able to load the VxD. If the EMM386.EXE file is missing, EMM386 will tell Windows (via INT 2Fh function 1605h, of course) not to load. However, when I created a dummy file named EMM386.EXE that didn't contain the requisite VxD, Windows still started. Furthermore, EMS and the UMBs were in the DOS box.

A VxD by itself was not essential to carrying over EMS and UMBs to Windows, but what was? Searching for documentation was futile, but reverse engineering revealed an undocumented interface between memory managers and Windows that is responsible for carrying over EMS/UMB to Windows. The interface, as I later found out, is called the "Global EMM Import Specification."

The EMM Import Specification

The interface itself is relatively simple. When the V86MMGR (V86 Memory Manager) VxD in Windows 386 Enhanced mode starts up, it generates an IOCTL read to "EMMXXXX0", the EMS device, which returns the physical address of a block of memory; see Figure 1. This block of memory is called the "EMM import structure" and contains information Windows can use to take over the memory manager's page tables and support its own virtual EMS/XMS/UMB drivers.

Figure 2 shows the layout of the EMM import structure; Listing One is the EMM import as a set of C structures. The EMM import is described in more detail in EMMIMP.H, available electronically; see "Availability," page 3.

The structure starts with a header, which includes one of three version numbers. Version 1.00 does not support information on UMBs or XMS free memory; 1.10 does; and 1.11 supplies UMB and XMS information plus the memory manager's vendor and product names. Windows 3.1 can handle all versions, but Windows 3.0 could handle only 1.00. Memory managers handed UMBs over to Windows 3.0 through VxDs.

Frame descriptors provide Windows with a snapshot of memory status below 1 megabyte. There are 64 frame descriptors, one for each 16K of memory (a frame). A 386 page is 4K, so each frame represents four pages. In Figure 2, "Type" indicates either a normal frame, an EMS page frame, or a frame containing a UMB. For an EMS page frame, "Physical Page #" is the EMS physical page number, and "Handle #" and "Logical Page #" are the EMS handle and logical page numbers mapped to the frame, respectively. For a UMB frame, the "Handle #" instead acts as an index into UMB frame descriptors. A UMB descriptor contains four physical page numbers of a UMB frame.

The Pagemap Physical Address field of an EMS descriptor holds the physical address of an array of 386 page-table values, describing which physical pages are owned by the EMS handle.

The Global EMM Import structure also supplies the real-mode INT 67h vector so that Windows can install a V86 breakpoint. Windows replaces the first byte of the real-mode INT 67h handler with an ARPL instruction (63h). When a program calls the INT 67h handler, an Illegal Opcode Exception (INT 6) is generated, so Windows can trap these calls.

The HMA page table physical address in Figure 2 points to a table of 386 page-table values that make up the Higher Memory Area (HMA). This table is necessary when HMA consists of pages higher than 1MB+64KB. Windows adds these pages to its memory pool (via the *VMM _AddFreePhysPage* service) or calls *Set_Physical_HMA_Alias* for each page, depending on the value of *Flag0* in the header.

Windows uses free page lists to add memory to its pool of free memory (again, via *_AddFreePhysPage*). For memory managers that don't share EMS and XMS memory, this list can be used to provide free EMS memory to Windows. (All XMS memory is allocated by Windows when it starts up.)

Bit 0 of the *Flag* field in an XMS descriptor indicates whether or not the handle is allocated prior to Windows startup. If bit 0 is 1, then the handle is free, and Windows can use the handle for its XMS driver. If the memory manager itself handles XMS calls while Windows is running, this array could be empty.

Reaching for the EMM Import Structure

Earlier, I stated that the EMM import structure address is returned by an IOCTL read to "EMMXXXX0". But what if this device does not exist? A memory manager with EMS disabled (such as EMM386 with "NOEMS" or QEMM with "EMS=NONE") does not install "EMMXXXX0" because EMS is not active. Instead, a device with a slightly different name ("EMMQXXX0" for EMM386 and QEMM, "QMMXXXX0" for 386MAX) is installed.

Windows does not try to IOCTL-read all these devices if it can't find "EMMXXXX0"; it issues the IOCTL-read to "EMMXXXX0" only. When memory managers receive a call to INT 2Fh function 1605h, they rename the device to "EMMXXXX0". The IOCTL-read to "EMMXXXX0" is issued afterwards, so Windows need not know the actual device name. Memory managers restore the no-EMS device name when INT 2Fh function 1606h (the Windows exit broadcast) is detected; see Table 1.

Issuing INT 2Fh AX=1605h and IOCTL-reading the "EMMXXXX0" device gives you the physical address to the EMM import structure. Simply hex-dumping the structure would most likely yield garbage, especially if you have not run Windows since your last boot.

The reason is simple: Memory managers don't set up the EMM import structure at the time of the IOCTL read or the INT 2Fh function 1605h call. They don't set up the structure until the disable V86-mode switch function is called. Why? Remember that the EMM import structure is supposed to be a snapshot of memory usage. It must reflect the status of memory just before Windows gets control of protected mode. Many programs intercept INT 2Fh, and memory managers are towards the end of the INT 2Fh chain (because they are one of the first installed in CONFIG.SYS). If the EMM import structure were setup at INT 2Fh, other programs could map EMS memory or allocate/free/resize EMS/XMS handles, all of which would make the structure stale. The same reasoning applies to IOCTL-read. Thus, to get the EMM import structure, we must call the memory manager's V86 enable/disable routine.

Calling the routine to disable V86 mode puts the machine into real mode and leaves the system in a very unstable state. When the CPU is in real mode, its paging mechanism is disabled so there are no UMBs or EMS during that time. If a TSR or driver that hooks an interrupt uses EMS, it probably won't work properly; if it is in UMB, it will simply disappear. The interrupt chain is intact, so an interrupt--hardware or software--would be disastrous. For this reason, when you call the entry point, you must:

- Disable interrupts.

- Call the V86 enable/disable routine with AX=0, to put the machine into real mode.
- Immediately call the routine again with AX=1, to restore V86 mode.

This will reveal the EMM import structure. (Well, at least with EMM386 and QEMM. 386MAX requires another twist, described later.)

The EMMIMP Program

EMMIMP is a DOS program that dumps the EMM import structure in a readable form. EMMIMP.C (Listing Two) basically emulates Windows' interaction with 386 memory managers.

The *emulate_win_init* and *emulate_win_term* routines in EMMIMP.C emulate INT 2Fh functions 1605h and 1606h, respectively . *Emulate_win_init* receives the parameter *ver*, which specifies the version of Windows it should pretend to be (300h for 3.0, 30Ah for 3.1). *Emulate_win_init* also returns the mode-switch entry-point address. *Get_emm_imp_addr* issues an IOCTL-read to "EMMXXXX0" to get the physical address of the EMM import structure. *Switch_to_real* and *switch_to_prot* handle calls to the mode-switch entry point. Note that some crucial registers are saved before calling the mode-switch entry point, since register preservation is not guaranteed across the call.

The EMM import structure is somewhere in extended memory, so for the EMMIMP (a real-mode program) to read it, the program must copy the structure to conventional memory. MOVEEXT.C (Listing Three) has three functions for the job: one for the PC (INT 15h function 87h), another for the NEC PC-9800 (INT 1Fh function 90h), and a third for 386MAX.

Figure 3 shows an excerpt of EMMIMP output when running under EMM386 Version 4.45. The output shows EMS and UMB frame descriptors. Frames 0x10 through 0x27 (4000:0--9C00:0) are large-frame EMS physical pages. Frames 0x32 through 0x37 (C800:0--DC00:0) are UMB frames with indexes into UMB descriptors. Frames 0x38 through 0x3b (E000:0--EC00:0) are EMS frames, with frame 0x38 mapped to EMS handle 1, page 1. The UMB descriptors show where the UMB memory comes from. For example, 16K of UMB memory at C800:0 (frame 0x37) has UMB descriptor index 0, which in turn shows that the frame maps into physical pages 120h through 123h.

There are two EMS handles: 0, the system handle; and 1, named "test." After each EMS handle is a dump of the page map, showing the memory owned by the handle. For example, logical page 0 of EMS handle 1 corresponds to pages 150h through 153h.

The free-page list shows pages not used by the memory manager while Windows is running. This is information you can't get using the EMS, XMS, or VCPI interface. In Figure 3, there are 52 free pages starting from 15Ch.

386MAX

Just as with any memory manager, most of 386MAX's system code and data are in extended memory, but trying to copy them into conventional memory buffer with INT 15h function 87h won't work. 386MAX will either refuse to copy anything or (in the case of the EMM import structure) will fill the buffer with 0s.

One way to access the system area of 386MAX is to enter protected mode via VCPI. This involves setting up the GDT, IDT, and page tables--just to copy some bytes of extended memory. After switching the CPU into real mode to refresh the EMM import structure, you must directly enter protected mode. This, after all, is what Windows is doing, albeit on a much grander scale.

Using INT 15h function 87h won't work in real mode because 386MAX's INT 15h handler is in a UMB, which disappears in real mode. The *move_ext_mem_real* routine in Listing Three enters protected mode and copies memory. Since you don't do any far calls and interrupts are disabled, there is no need for code segments or an IDT. Only a GDT with a minimum of segments is created.

Table 1: Interaction of Windows and memory managers, (a) at Windows startup; (b) at Windows exit.

```
     Windows                    Memory Manager

 (a) INT 2Fh AX=1605h           Set device name to "EMMXXXX0".
                                Return mode-switch entry-point address.
     IOCTL-read "EMMXXXX0"      Return physical address of EMM import structure.
     Call mode-switch
            entry-point (AX=0)
                                Set CPU to real mode.
Windows runs_

 (b) Set CPU to real mode
     Call mode-switch
            entry-point (AX=1)  Setup the EMM import structure.
                                Regain protected mode.
     INT 2Fh AX=1606h           Restore device name.
```

Figure 1: Getting the physical address of the EMM import structure.

```
#pragma pack(1)
typedef struct {
    unsigned long addr;
    unsigned char maj, min;
    } EMM_IMPORT_ADDR;  // must be 6 bytes
unsigned long get_emm_import_addr(void)
{
    unsigned long addr = 0;
    EMM_IMPORT_ADDR impaddr;
    int emm = open("EMMXXXX0", O_RDONLY);
    if (emm == -1)
        return 0L;      // fail: no EMM
    impaddr.addr = 1;
    #define IOCTLREAD 2
    if (ioctl(emm, IOCTLREAD, (void far *) &impaddr, sizeof(impaddr)) != 0)
        addr = impaddr.addr;
    close(emm);
    return addr;        // physical address
}
```

Figure 2: Layout of the EMM import structure.

```
Header: (all versions)
    byte    Flag0
    byte    ???
    word    Size of struct (bytes)
    word    Version #
    dword   ???
Snapshot of memory status below 1MB: (all versions)
    array of 64 frame descriptors
        a frame descriptor is:
        byte    Type
        byte    Handle #
        word    Logical Page #
        byte    Physical Page #
        byte    Flag
    followed by:
    byte    ???
UMB information: (all versions)
    byte    # of UMB frame descriptors
    array of UMB frame descriptors
        a UMB frame descriptor is an array of 4 dwords
        each dword is 386 physical page number
EMS handle information: (all versions)
    byte    # of EMS handle descriptors
    array of EMS handle descriptors
        where an EMS handle descriptor is:
        byte    Handle #
        byte    Flag
        byte    Handlename[8]
        word    EMS pages owned by handle
        dword   Pagemap Physical Address
free memory information: (version 1.10, 1.11)
    dword   Real mode INT 67h vector
    dword   HMA page table physical address
    byte    # of free page list
    array of free page list
        where a free page list is:
        dword   Physical page #
        dword   # of free pages
XMS information: (version 1.10, 1.11)
    byte    # of XMS handle descriptors
    array of XMS handle descriptors
        where an XMS handle descriptor is:
        word    Handle #
        word    Flag
        word    Size (KB)
        dword   Base Address
free UMB information: (version 1.10, 1.11)
    byte    # of free UMB descriptors
    array of free UMB descriptors
        where a free UMB descriptor is:
        word    Segment
        word    Paragraphs
Product information: (version 1.11)
    vendor name of the memory manager
    product name of the memory manager
```

Figure 3: EMMIMP output on EMM386 (excerpt).

```
C:\> emmimp
mode switch entry:03af0992
emm import structure address:00119000
flag0:0x4
size:0x23c bytes
version:0x10b
frame[0x10] (4000:0):large EMS (phys page 04)
  ...
frame[0x27] (9c00:0):large EMS (phys page 27)
frame[0x32] (c800:0):UMB/c800/c900/ca00/cb00/umb desc index:0
frame[0x33] (cc00:0):UMB/cc00/cd00/ce00/cf00/umb desc index:1
frame[0x34] (d000:0):UMB/d000/d100/d200/d300/umb desc index:2
frame[0x35] (d400:0):UMB/d400/d500/d600/d700/umb desc index:3
frame[0x36] (d800:0):UMB/d800/d900/da00/db00/umb desc index:4
frame[0x37] (dc00:0):UMB/dc00/dd00/de00/df00/umb desc index:5
frame[0x38] (e000:0):EMS (phys page 00)/mapped to handle 1 page 1
frame[0x39] (e400:0):EMS (phys page 01)
frame[0x3a] (e800:0):EMS (phys page 02)
frame[0x3b] (ec00:0):EMS (phys page 03)
resv2:0x74
# of umb desc:6
umbdesc[ 0]:00000120 00000121 00000122 00000123
umbdesc[ 1]:00000124 00000125 00000126 00000127
umbdesc[ 2]:00000128 00000129 0000012a 0000012b
umbdesc[ 3]:0000012c 0000012d 0000012e 0000012f
umbdesc[ 4]:00000130 00000131 00000132 00000133
umbdesc[ 5]:00000134 00000135 00000136 00000137
EMS handle  0: name="", 24 EMS pages, pagemap at 0011b000
    log page[0]:00040267 00041067 00042067 00043067
    ...
    log page[17]:0009c267 0009d067 0009e067 0009f067
EMS handle  1: name="test", 3 EMS pages, pagemap at 0011b180
    log page[0]:00150267 00151067 00152067 00153067
    log page[1]:00154267 00155067 00156067 00157067
    log page[2]:00158267 00159067 0015a067 0015b067
realmode int 67 vector:03af:02b0
```

```
hma_page_table_addr:00000000
# of free page lists:1
free page list[00]:page 0000015c, 52 pages
# of XMS info:0
# of umb free seg:1
umb free seg:c93a, 0x16c6 paragraphs
maker:"MICROSOFT           "
product:"EMM386 4.45         "
```

Listing One

```
/* emmimp.h, EMM import structure -- copyright (c) taQ 1994  */
typedef unsigned long uint32;   /* double word */
typedef unsigned short uint16;  /* word */
typedef unsigned char uint8;    /* byte */

typedef struct {
    uint8 type;
    uint8 handle;
    uint16 logpage;
    uint8 physpg_num;
    uint8 flag;
} emmimp_frame_desc_t;  /* frame descriptors */
typedef struct {
    uint8  flag0;
    uint8  resv0;
    uint16 size;
    uint16 version;
    uint32 resv1;
    emmimp_frame_desc_t frame[64];
    uint8 resv2;
 } emmimp_hdr_t;    /* header */
typedef struct {
    uint32 physpgnum[4];
} emmimp_umb_desc_t;    /* UMB descriptor */
typedef struct {
    uint8 handle;
    uint8 flag;
    uint8 hndlname[8];
    uint16 pages;
    uint32 pgmap_phys_addr;
} emmimp_ems_desc_t;    /* EMS descriptor */
typedef struct {
    uint32 physpgnum;
    uint32 numpg;
} free_page_list_t; /* free page list */
typedef struct {
    uint16 handle;
    uint16 flag;
    uint32 kb;
    uint32 baseaddr;
 } emmimp_xms_desc_t;   /* XMS descriptor */
typedef struct {
    uint16 startseg;
    uint16 parasize;
}  free_umb_desc_t; /* free UMB descriptor */
```

Listing Two

```
/* emmimp.c, dumping the EMM import structure --  copyright (c) taQ 1994  */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dos.h>
#include <io.h>
#include <fcntl.h>
#include "emmimp.h"
#include "moveext.h"

#ifndef __TURBOC__
#define asm _asm
#endif

static int for_386max = 0;
static int raw_dump = 0;
static int machine;

static void emulate_win_init(uint16 ver, void (far **entryp)(void))
{
    void (far *entry)(void);

    asm push ds
    asm push es
    asm mov dx, 0
    asm mov ax, 0
    asm mov es, ax
    asm mov ds, ax
    asm mov di, ver
    asm mov si, 0
    asm mov cx, 0
    asm mov bx, 0
```

```
    asm mov ax, 1605h
    asm int 2fh
    asm mov word ptr entry, si
    asm mov word ptr entry + 2, ds
    asm pop es
    asm pop ds
    *entryp = entry;
}
static void emulate_win_term(void)
{
    asm mov ax, 1606h
    asm int 2fh
}
static uint32 get_emm_imp_addr(void)
{
    uint8 buf[6];
    int fd;

    fd = open("EMMXXXX0", O_RDONLY | O_BINARY);
    if (fd < 0) {
        printf("can't open EMMXXXX0\n");
        return 0;
    }
    memset(buf, 0, sizeof buf);
    buf[0] = 1;
    asm lea dx, word ptr buf
    asm mov cx, 6
    asm mov bx, fd
    asm mov ax, 4402h
    asm int 21h
    asm jc fail
    printf("emm import structure address:%08lx\n", *(uint32 *) buf);
    printf("emm import structure version:%d.%02d\n", buf[4], buf[5]);
    close(fd);
    return *(uint32 *) buf;
fail:
    printf("emm import struct address get failed\n");
    close(fd);
    return 0;
}
static void switch_to_real(void (far *entry)(void))
{
    asm push ds
    asm push es
    asm push di
    asm push si
    asm push bp
    asm mov ax, 0
    asm call dword ptr entry;
    asm pop bp
    asm pop si
    asm pop di
    asm pop es
    asm pop ds
}
static void switch_to_prot(void (far *entry)(void))
{
    asm push ds
    asm push es
    asm push di
    asm push si
    asm push bp
    asm mov ax, 1
    asm call dword ptr entry;
    asm pop bp
    asm pop si
    asm pop di
    asm pop es
    asm pop ds
}
static uint16 winver = 0x30a;   /* Windows version */
main(int ac, uint8 **av)
{
    void (far *entry)(void);
    int i;
    uint32 emm_imp_addr;
    static uint8 buf[8192];

    get_options(ac, av);    // not shown
    if (!for_386max)
        machine = machine_type();
    emulate_win_init(winver, &entry);
    if (!entry) {
        printf("illegal mode switch entry\n");
        emulate_win_term();
        return 1;
    }
    printf("mode switch entry:%08lx\n", entry);

    emm_imp_addr = get_emm_imp_addr();

    if (!emm_imp_addr) {
        emulate_win_term();
        return 1;
    }
    if (!for_386max) {
        disable();  /* disable interrupts while in real mode */
```

```
        switch_to_real(entry);
        switch_to_prot(entry);
        enable();   /* now Ok to enable interrupts */
        emulate_win_term();
        if (machine == MC_98)
            move_ext_mem_98(emm_imp_addr, buf, sizeof buf);
        else
            move_ext_mem_pc(emm_imp_addr, buf, sizeof buf);
        dump_emm_imp(emm_imp_addr, buf);
    } else {
        disable();
        switch_to_real(entry);
        move_ext_mem_real(emm_imp_addr, buf, sizeof buf);
        switch_to_prot(entry);
        enable();
        emulate_win_term();
        dump_emm_imp(emm_imp_addr, buf);
    }
    return 0;
}
```

Listing Three

```
/* moveext.c, extended memory move -- copyright (c) taQ 1994 */
#pragma inline
#include <stdio.h>
#include <string.h>
#include <dos.h>
#include "moveext.h"

#ifndef __TURBOC__
#define asm _asm
#endif

typedef unsigned short uint16;  /* word */
typedef unsigned char  uint8;   /* byte */
/* segment descriptor */
typedef struct {
    uint16 lim0_15;     /* Bits 0-15 of the segment limit */
    uint16 base0_15;        /* Bits 0-15 of the segment base */
    uint8 base16_23;        /* Bits 16-23 of the segment base */
    uint8 acc_byte;         /* Access byte */
    uint8 lim16_19;     /* Bits 16-19 of the segment limit
                       plus some flags */
    uint8 base24_31;        /* Bits 24-31 of the segment base */
} desc_t;
/* convert far pointer to address */
static uint32 fp_to_addr(void far *p)
{
    uint32 addr = FP_SEG(p);
    addr <<= 4;
    addr += FP_OFF(p);
    return addr;
}
static void set_data_gdt(desc_t *gdtp, uint32 addr, uint16 limit)
{
    gdtp->lim0_15 = limit;
    gdtp->base0_15 = addr;
    gdtp->base16_23 = (addr >> 16) & 0xff;
    gdtp->base24_31 = (addr >> 24) & 0xff;
    gdtp->acc_byte = 0x93;  /* data, expand up, r/w, dpl = 0 */
    gdtp->lim16_19 = 0;
}
/* move extended memory, PC-AT */
int move_ext_mem_pc(uint32 srcaddr, void far *dest, uint16 bytes)
{
    void far *p;
    static desc_t gdt[6];   /* 10h: src descriptor, 18h: dest */

    memset(gdt, 0, sizeof gdt);
    /* set src & dest descriptors */
    set_data_gdt(gdt + 2, srcaddr, bytes);
    set_data_gdt(gdt + 3, fp_to_addr(dest), bytes);

    p = &gdt;
    asm les si, p
    asm mov ah, 87h
    asm mov cx, bytes
    asm inc cx
    asm shr cx, 1
    asm int 15h
    /* error check omitted */
    return 1;
}
/* move extended memory, PC-9800 */
int move_ext_mem_98(uint32 srcaddr, void far *dest, uint16 bytes)
{
    desc_t far *p;
    static desc_t gdt[6];   /* 10h: src descriptor, 18h: dest */

    memset(gdt, 0, sizeof gdt);
    /* set src & dest descriptors */
    set_data_gdt(gdt + 2, srcaddr, 0xffff);
```

```
    set_data_gdt(gdt + 3, fp_to_addr(dest), 0xffff);

    p = gdt;
    asm mov cx, bytes
    asm les bx, p
    asm mov si, 0
    asm mov di, 0
    asm mov ah, 90h
    asm clc
    asm int 1fh
    asm jnc _OK
    return 0;
_OK:
    return 1;
}
typedef struct {
    uint16 len;
    uint32 physaddr;
    uint16 filler;
} gdt_val_t;
/* move extended memory, generic, from real mode, assumes interrupts disabled*/
int move_ext_mem_real(uint32 srcaddr, void far *dest,
    uint16 bytes)
{
    gdt_val_t gdt_val;
    static desc_t gdt[3];   /* 8h: src descriptor, 10h: dest */

    memset(gdt, 0, sizeof gdt);
    /* set src & dest descriptors, limit must be 0xffff */
    set_data_gdt(gdt + 1, srcaddr, 0xffff);
    set_data_gdt(gdt + 2, fp_to_addr(dest), 0xffff);

    gdt_val.len = sizeof gdt;
    gdt_val.physaddr = fp_to_addr(&gdt);
    gdt_val.filler = 0;

    asm push ds
    asm push es
    asm push si
    asm push di
    asm .386p
    /* load gdt */
    asm lgdt fword ptr gdt_val
    /* enter protected mode */
    asm mov eax, cr0
    asm or eax, 1
    asm mov cr0, eax
    /* ds <- src descriptor */
    asm mov ax, 8h
    asm mov ds, ax
    asm xor si, si
    /* es <- dest descriptor */
    asm mov ax, 10h
    asm mov es, ax
    asm xor di, di
    asm mov cx, bytes
    asm cld
    asm rep movsb
    /* back to real mode */
    asm mov eax, cr0
    asm and eax, not 1
    asm mov cr0, eax
    asm .8086
    asm pop di
    asm pop si
    asm pop es
    asm pop ds
    return 1;
}
int machine_type(void)
{
    uint16 rc;
    union REGS regs;
    rc = MC_UNKNOWN;    /* unknown */
    /* do PC-AT get realtime clock call, NOP interrupt in PC-9800 */
    regs.x.cx = 0;
    regs.h.ah = 4;
    int86(0x1a, &regs, &regs);
    /* if ch contains current century, then PC-AT */
    if (regs.h.ch == 0x19 || regs.h.ch == 0x20)
        rc = MC_PC; /* PC-AT */
    else
        rc = MC_98; /* PC-98 */
    return rc;
}
```
