How to Ensure That Your Program Does Not Run Under Windows 95

My primary focus for much of the Windows 95 project was MS-DOS application compatibility, specifically games. To help maintain my sanity, I found myself writing the following account, illustrating the phenomenal creativity of software developers in doing everything within their power to be incompatible with Windows 95.

This chapter is a very technical one, originally written for an audience consisting of developers deeply steeped in the minutiae of protected-mode programming. I have attempted to clarify the jargon, but the issue is unavoidably technical. Here are some terms you will encounter:

- The *Global Descriptor Table (GDT)* is a central data structure that is used by the CPU and that controls how programs access memory. Allowing access to the GDT effectively grants total control of the computer.
- *Real mode* was the only mode supported by the CPU prior to the introduction of the 80286. MS-DOS was originally developed to run in real mode. Real mode has no memory protection. Modern operating systems use real mode only transitionally on their way to

protected mode, which is a mode of the CPU in which memory protection becomes available.

- The *DOS Protected-Mode Interface (DPMI)* and the *Virtual Control Program Interface (VCPI)* are two popular interfaces for writing 32-bit DOS-based programs. Windows 3.0 and higher support DPMI.
- An *MS-DOS extender* (often called a *DOS extender*, or even just an *extender*) is a library that assists in running 32-bit MS-DOS-based programs. DOS extenders typically detect that a DPMI or VCPI server is already running and use the corresponding interface to provide necessary services to the 32-bit MS-DOS-based program. In the absence of such a server, the DOS extender will provide one on its own. At the time of the Windows 95 project, there were several commercial DOS extenders available to choose from.

The interface to MS-DOS and the MS-DOS extender is a register-based one, so all the programs of the era in which these programs were written contained some degree of assembly language. Furthermore, it was considered standard practice for applications to do such things as reprogram hardware devices and replace hardware interrupt handlers (operations that in today's programming environment would be considered beyond the scope of what applications can or even should be allowed to do). Such activities necessitated the use of assembly language, and as a result, many of the topics covered here will be heavy on 80386 assembly language, both 16-bit and 32-bit. I have converted the assembly language back to pseudo-C when possible; when this is not possible, I have attempted to explain, in the accompanying text, what the assembly language does.

Bear in mind that I wrote this chapter back in 1995 as a way of letting off steam. Its purpose was not to ridicule the vendors in question; that was just a side effect of the ironic tone. But just to make sure nobody takes it personally, I've made all the product and company names anonymous.

Prologue

There have been myriad articles describing how to write programs to run properly under Windows 95, but a title sorely lacking is one describing how to write a program that will fail spectacularly when run under Windows 95. This document attempts to fill that void by describing steps you can take when developing your application to ensure that it encounters as many compatibility problems as possible. By carefully following these steps, you can be relatively certain that your program will not run on any Microsoft operating system well into the next century.

MS-DOS-Extended programs

Congratulations on your decision to write an MS-DOS-extended program. You stand at the frontier of protected-mode programming. Unwilling to take the leap to a protected-mode operating system like Windows, you prefer to keep one foot in the comfortable world of MS-DOS while venturing with the other foot into the gnarly depths of protected mode. Here are some tips for making sure your program will resist the pull of Windows for as long as possible.

Just party the GDT yourself

Remember that the Global Descriptor Table (GDT) belongs to the operating system. Any operating system that even pretends to protect itself from applications will not allow applications to access the GDT, much less try to change it.

The easiest way to ensure that you will not run under any operating system whatsoever is to execute an `lgdt` instruction, which loads a new global descriptor table. This instruction bypasses the operating system and enters protected mode directly. It helps if you don't even check if the CPU is in real mode before doing this.

Example 1

All known versions of one game try to enter protected mode by loading the global descriptor table register directly. Under a protected-mode operating system, this crashes with a general protection fault. Another game refuses to run unless it is in real mode and can take complete control of the machine.

Example 2

One game goes into protected mode with the help of a particular DOS extender (which we will call Extender 2), thereby creating the illusion that it wants to play friendly with the operating system. But then it goes ahead and edits GDT selectors directly.

Use VCPI

Windows 95 does not support the Virtual Control Program Interface (VCPI) because that interface gives applications total control of the computer at the lowest level, something generally frowned on in today's computing environment. If you design your program to require VCPI, it will never run on any operating system that wants to remain in charge. It helps if you also display a really confusing error message when you fail to detect a VCPI server.

Example 3

Two games by a popular game publisher (which we will call Publisher 3) merely display the message

```
EMS driver is not VCPI compliant.
```

and unceremoniously exit if a VCPI server cannot be found.

Example 4

Another program by Publisher 3 displays the wonderfully useful error message

```
Protected mode driver is not responding to EMS function calls.
Make sure your EMS driver is not configured with the NOEMS option.
If it is, replace the option 'NOEMS' with 'RAM'.
```

This leads the user on a wild goose chase trying to configure expanded memory in myriad ways, when the real problem has nothing to do with EMS.

Maxim 1: The more confusing the error message, the better.

Write your own (DPMI) DOS extender

An excellent first step in writing an MS-DOS-extended program that will not run under Windows 95 is to try to write your own extender. Writing a DOS extender is not something that should be left in the hands of qualified professionals. If you use a third-party DOS extender, you run the risk that any problems resulting from the extender might be fixed by the vendor, or that the operating system developers might come up with a workaround for the problem in the DOS extender. By writing your own DOS extender, you can ensure that any problem in the extender is completely your own. This way, the operating system developers will give higher priority to problems with commercial DOS extenders rather than your problem, because any such workaround would fix only your program and no other.

Example 5

The DOS extender used by one game is a home-grown one. (You might find it amusing that the author of this extender subsequently came to work for Microsoft for several years.) The application makes an Int 21h function 250Dh call to the DOS extender to set the interrupt vector for IRQ 5, the sound card. The extender saves the application's vector address into a private table and loads AX:ECX with the address of the extender's private wrapper. And then, oh wait, it gets distracted for a little bit, and goes down a random code path that scrambles the AX and ECX registers, loading it instead with random values from uninitialized memory. And then it jumps back to the main code, which tries to set this as the interrupt vector to this garbage address, which fails. The application then hangs waiting for a hardware interrupt that never arrives. The application documentation says

```
Do NOT run Program 5 from a shell in Windows. The virtual memory
'feature' [sic] of Windows causes Program Name to crash.
```

This is an important lesson.

Maxim 2: *If it doesn't run under Windows, then it's Windows' fault.*

The flag that the program in Example 5 checks to decide whether to scramble AX:ECX appears to be a flag that is set if the extender is acting as a DPMI client rather than a server.

Maxim 3: *Detect Windows, and crash.*

Ignore the manuals

If you choose to go with a commercial DOS extender, make sure never to read the manual.

Maxim 4: Manuals are a waste of time.

Better to experiment, and if it doesn't crash the machine, then it's safe to do. Examples of this sort of thing will arise throughout the remainder of this subsection.

Manage the interrupt flag incorrectly

There are a few sections of the DPMI specification that you must be careful never to read. Do not continue reading if you wish to maintain deniability.

First, take a look at the first three paragraphs of section 2.3 (Interrupt Flag Management) from the DPMI 0.9 specification. All emphasis is preserved from the original. (I apologize in advance for the deep technical content, but interrupt management is inherently deep. I'll summarize afterward.)

> *The* `popf` *and* `iret` *instructions may not modify the state of the interrupt flag since most DPMI implementations will run programs with IOPL < DPL. Programs must execute* `cli` *or* `sti` *to modify the interrupt flag state.*
>
> *This means that the following code sequence will leave interrupts disabled.*

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

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

Next, from the same specification, section 2.4.1 (Hardware Interrupts) contains the following instructions:

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

And, again from the same specification, the introduction to Chapter 17 ("Virtual Interrupt State Functions") describes the role and manipulation of interrupt flags:

> *Under many implementations of DPMI, the interrupt flag in protected mode will always be set (interrupts enabled). This is because the program is running under a protected operating system that can not allow programs to disable physical hardware interrupts. However, the operating system will maintain a "virtual" interrupt state for protected mode programs. When the program executes a* `cli` *instruction, the program's virtual interrupt state will be disabled, and the program will not receive any hardware interrupts until it executes an* `sti` *to reenable interrupts (or calls service 0901h).*
>
> *When a protected mode program executes a* `pushf` *instruction, the real processor flags will be pushed onto the stack. Thus, examining the flags pushed on the stack is not sufficient to determine the state of the program's virtual interrupt flag. These services enable programs to get and modify the state of their virtual interrupt flag.*
>
> *The following sample code enters an interrupt critical section and then restores the virtual interrupt state to its previous state.*

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

Okay, what does all that mean? The interrupt flag is a processor flag that controls whether the CPU will respond to hardware interrupts ("interrupts are enabled") or ignore them ("interrupts are disabled"). The `cli` instruction clears the interrupt flag, thereby disabling interrupts. Conversely, the `sti` instruction sets the interrupt flag, enabling interrupts.

The DPMI specification goes to some length describing incorrect ways of manipulating the interrupt flag and spelling out what programs must do in order to do it correctly. Mind you, these rules were not invented by the DPMI specification just to make life difficult for programmers. Rather, these rules are merely a reflection of the behavior of the 80386 processor. In summary, the 80386 processor behaves in the following ways with respect to the interrupt flag:

- The `popf` instruction, which pops a value from the processor stack and stores it in the flags register, will not necessarily alter the status of the interrupt flag.
- The `iret` instruction, which returns from an interrupt, also does not necessarily alter the status of the interrupt flag.
- The `pushf` instruction, which stores the flags register on the processor stack, stores the true state of the interrupt flag, not the state of the interrupt flag as seen by the application.

A `cli` or an `sti` instruction logically disables or enables interrupts, but the DPMI server is permitted to leave physical interrupts enabled at all times. When a hardware interrupt occurs, the DPMI server will handle it and wait for the application to enable interrupts before running the application's hardware interrupt handler. When a program performs a `pushf` instruction, the processor pushes the *physical* interrupt flag and not the virtualized one. (This isn't DPMI's fault; that's just the way the processor works.) As a result, a code sequence like

```
          cli
          pushf
```

may push flags that indicate that interrupts are *enabled*. To obtain the virtualized interrupt state, a program needs to use the DPMI services that manipulate the virtual interrupt state. But that's too much work, so go ahead and assume that the physical and virtual interrupt states are identical.

(Another detail of 80386 assembly language you may encounter in the discussion that follows is that some instructions end in "d." The "d" suffix stands for "double" and represents the 32-bit version of the instruction. The instruction without a "d" is the 16-bit instruction. For example, the 32-bit version of the `popf` instruction is `popfd`. This "d" is often suppressed by 32-bit assemblers and disassemblers, because they don't support 16-bit code and the "32-bit-ness" is implied.)

The remarks regarding the management of the interrupt flag are reiterated in the Phar Lap TNT DOS-Extender Reference Manual in section 8.7 (Interrupt Flag Control under DPMI). Be sure to ignore the sentence

> *Do **not** assume POPFD or IRETD will change the IF, because they won't.*

at the bottom of page 153 of the manual.

***Maxim 5:** Words in boldface don't count.*

In fact, the Phar Lap SDK is filled with annoyingly helpful information. Make sure to ignore everything. (This takes some effort, since there is so much information to ignore, but the rewards are well worth the time spent in not reading it. Be extra sure to take a day to ignore Chapter 11, "Programming for DPMI Compatibility.")

Example 6

The most common way of managing the interrupt flag incorrectly is simply by writing

```
pushfd          ; push flags onto the stack
cli             ; disable interrupts
...
popfd           ; restore original interrupt state
                ; BUG! Does not enable interrupts
```

Notice that this is pretty much word-for-word the code sequence that section 2.3 explicitly cautions against. (The only difference is that the instructions here are the 32-bit versions rather than the 16-bit versions used in the specification.) This is the method of choice, employed by a huge number of chart-topping games, far too many to list here, including one declared "the best action game of all time" (which we will call Program 6). Do this, and you will be in great company.

Example 7

A 16-bit productivity application contains the following noteworthy code sequence:

```
APPNAME!(1):22c4:
        pushf
        cli
        ... five instructions later ...
        popf         ; BUG!  Does not enable interrupts
```

What makes this noteworthy is not that it contains exactly the forbidden code sequence from section 2.3, quoted at the top of this section. No, it is noteworthy because this is not an MS-DOS application but rather a Windows application. It's a Windows application doing something that doesn't work under Windows. That takes nerves. A certain network driver does essentially the same thing.

Maxim 6: *Just because you're a Windows application doesn't mean that you have to be compatible with Windows.*

Example 8

Many 32-bit games (including one we'll call Program 8) take the principle even further and use this code sequence:

```
        pushf
        cli
        ...
        popf            ; BUG!  Does not enable interrupts
```

This code uses `pushf` before the `cli` instead of the `pushfd` instruction, preferring to use a 16-bit instruction when running in 32-bit mode. Not only is this sequence bulkier and slower, but it also misaligns the stack, causing the CPU to run slowly until the final `popf`.

Maxim 7: *Performance degradation is acceptable in the greater interest of not running on Windows 95.*

Curiously, Program 8 was issued by the same publisher that also followed Example 6. We will see this technique many times, so let's capture it in a maxim:

Maxim 8: If you're going to do something wrong, do it wrong in as many different ways as possible.

Example 9

A slightly less prevalent, but just as effective, way to manage the interrupt flag incorrectly is to write something like this:

```
pushfd          ; push flags onto stack
pop     eax     ; pop it into the eax register
cli
...
push    eax     ; push the eax register onto the stack
popfd           ; BUG!  Does not enable interrupts
```

Instead of leaving the processor flag state on the stack, this code sequence saves the flags into a register, then restores them from the register afterward. It really doesn't matter where you save the flags; the effect is the same. This alternate method has been popularized by several top-selling programs.

Example 10

And another way to manage the interrupt flag incorrectly is to borrow the following code from what we will call Program 10:

```
pushf                       ; push flags onto stack
pop     word ptr [oflags]   ; pop it into memory
cli
...
push    word ptr [oflags]   ; push the memory back onto the stack
popf                        ; BUG! Does not enable interrupts
```

This program had the same idea as the program from Example 9, but instead of storing the flags temporarily in a register, it stores them into memory. It really doesn't matter where you sweep the dust; it's still dust.

Example 11

Program 10 comes up with another clever twist on the same theme:

```
cli        ; disable interrupts (?)
iret       ; BUG!  Does not enable interrupts
```

This code thumbs its nose at section 2.4.1, which explicitly states that you "must execute an `sti` before executing `iret`." The `sti` instruction enables interrupts; the `cli` instruction disables them. Program 10 used exactly the opposite instruction from what it was supposed to use.

Maxim 9: Find a rule and break it as blatantly as possible.

Example 12

The authors of another program were even cleverer, spreading this technique over two different subroutines.

```
         pushfd
         call      far xxxx:yyyyyyyy
         ...
xxxx:yyyyyyyy:
         cli
         ...
         iretd            ; BUG!  Does not enable interrupts
```

Splitting up the bad code between a subroutine and its caller might make it look as if you're not violating the rules, but the CPU won't be fooled.

Example 13

One program, which we will call Program 13, takes a related approach in its segment loader:

```
          push    saved_flags
          push    saved_cs
          push    saved_ip
          iret             ; BUG! Does not enable interrupts
```

Again, the `iret` will not restore interrupts. It doesn't matter how you get those values onto the stack before you do the `iret`. The `iret` instruction doesn't care; it will won't enable interrupts.

Example 14

Still another program employs multiple techniques. First, it uses the traditional technique:

```
          pushf            ; 16-bit push in 32-bit code
          cli
          ...
          popf             ; BUG!  Does not enable interrupts
```

But in addition, it sometimes uses

```
pushf            ; 16-bit push in 32-bit code
push     ebp
cli
...
pop      ebp
popf             ; BUG!  Does not enable interrupts
```

interjecting an additional instruction between the `pushf` and the `cli`. Things like this keep operating system developers on their toes, since it prevents them from detecting common error patterns and automatically fixing them.

Maxim 10: *The weirder your code, the harder it will be for the operating system to detect that you are about to do something stupid and rescue you.*

Example 15

A classic example of multiple obfuscatory techniques comes from this program: It employs not one, not two, but three distinct code sequences that mismanage the interrupt flag. First, there's this sequence:

```
pushf                   ; 16-bit push in 32-bit code
pushad
push     ds
push     es
cli
...
pop      es
pop      ds
popad
popf                    ; BUG!  Does not enable interrupts
retd
```

Second, there's this one:

```
pushf                   ; 16-bit push in 32-bit code
cld
cli
...
popf                    ; BUG!  Does not enable interrupts
```

And third:

```
pushf                    ; 16-bit push in 32-bit code
```

```
        call     foo              ; Returns with interrupts disabled
        popf                      ; BUG!  Does not enable interrupts
```

If you use only one technique, the operating system might detect your usage pattern and apply a workaround for it. To avoid that, you should use several techniques. Even if the operating system detects some of your errors, you increase the chances that one of them will slip through.

Example 16

Another application that employs multiple techniques is so-called Program 16, which sometimes employs this code sequence:

```
        pushf                   ; 16-bit push in 32-bit code
        cli
        ...
        popf                    ; BUG! Does not enable interrupts
```

The second method employed by this program is to spread the bug over two subroutines. The first is this:

```
        pushf                   ; 16-bit push in 32-bit code
        pop      ax             ; 16-bit pop in 32-bit code
```

And the second looks like this:

```
        push     ax             ; 16-bit push in 32-bit code
        popf                    ; BUG! Does not enable interrupts
```

Remember to employ as many techniques as possible, so that the operating system has no chance of catching all of them.

Example 17

Two programs by Publisher 17 use general protection (GP) faults on purpose. They do this as a clever way to "call" their fault handler as if it were a subroutine. As part of the handling of the GP fault, the programs restore the flags like this:

```
    push saved_flags
    and word ptr [esp], NOT (IF_MASK + TF_MASK)
                     ; Disable the interrupt and trace flags
    popf
```

which doesn't really have much effect on the interrupt flag.

Example 18

Many games from Publisher 3 try to do the right thing by using the following code sequence:

```
       pushfd
       cli
       ...
                                 ; Now emulate a popfd
       mov      ebp, esp         ; Look at flags on the stack
       test     dword ptr [ebp], 200h ; Were interrupts on?
       jz       skip             ; Jump over next instruction if not
       sti                       ; They were on. Re-enable interrupts
skip:  pop      ebp              ; Finished snooping
       popfd                     ; Pop back the other flags
```

The intent here is to restore the interrupt bit manually, thus emulating the intended behavior of the `popfd` instruction. Unfortunately, this is not correct, because the flags pushed by the `pushfd` instruction reflect the physical and not logical interrupt flag. Since interrupts are physically enabled all the time, the code mistakenly enables logical interrupts.

The correct way to disable interrupts around a block of code (and therefore a technique to be avoided if you don't want to run on Windows 95) is as follows:

```
   mov      ax, 0900h  ; Disable interrupts and get previous state
   int      31h
   push     ax
   ...
   pop      ax
   int      31h        ; Restore interrupts to previous state
```

Assume that you are running at ring zero

In order to avoid running under any protected operating system, write your program so as to assume that it is running at ring zero, the highest privilege level in the Intel ring architecture. Since protected operating systems will run applications at the lowest rather than highest privilege level, design your program so that attempting to run it with anything short of full privileges will result in erratic behavior.

Example 19

According to the DPMI specification, applications must set the DPL (data privilege level) field in the requested descriptor equal to the CPL (current privilege level) of the current code segment. In English, this means that a program should set its data's privilege level the same as its code privilege level. Program 19 tries to create a data selector alias for its code selector with the DPMI SetAccessRights call, blindly setting the DPL field to zero, requesting the highest privilege level in the Intel 80386 architecture. When the call fails, the application abandons the operation that required writing to the code segment. Later, on a hardware interrupt, that value is accessed, and since it was never initialized, the program faults. The punch line is that none of this was necessary. Since Program 19 is a 32-bit program running in a flat memory model, the program already has a flat alias for the current code selector. It's called the flat data selector, also known as the DS register.

Maxim 11: *Always look for the weirdest, most convoluted way of doing something. That way, you stand a good chance of doing it wrong.*

Example 20

Many older versions of DOS Extender 2 create selectors and try to set their descriptors to DPL zero. The calls fail, but the extender doesn't check the return value. Luckily, the DPMI specification says that newly allocated selectors default to Present Read/Write Data, which is what the extender wanted, anyway.

Maxim 12: *It is better to be lucky than good.*

Example 21

All known versions of DOS Extender 2 fail to manage the coprocessor properly. When it detects that the machine does not have a coprocessor, the extender attempts to turn on coprocessor emulation with the following code sequence:

```
mov     eax, cr0 ; Privileged instruction!
or      al, 4    ; Turn on emulation
mov     cr0, eax ; Privileged instruction!
```

With a truly paranoid operating system, this will crash the application with a GP fault. A less paranoid operating system may silently ignore the accesses to the privileged registers, in which case coprocessor emulation remains disabled. In that case, the application uses coprocessor instructions, thinking that they are being emulated, when in fact there is nobody listening. When the application tries to read the results, it grabs bus noise. This is why two popular games, Program 21a and Program 21b, crash randomly if you try to run them in a DOS box on a machine without a coprocessor.

DPMI provides services for manipulating the coprocessor, which any self-respecting program should avoid if it wants to be sure not to run on Windows 95.

Make unwarranted assumptions about memory

With Windows 95, you are running under a multitasking operating system with virtual memory. Make sure to design your program to exploit this behavior to the worst possible extent.

Allocate all available memory

The easiest way to prevent your program from running under Windows 95 is to try to allocate all the memory in the entire system. Under no circumstances should you fall into the trap of allocating memory on an as-needed basis. Take a cue from the British Treasury, as explained by the fictitious Sir Humphrey Appleby in the television program *Yes, Prime Minister*: "The Treasury does not work out what it needs and then think how to raise the money. The Treasury pitches for as much as it can get away with and then thinks how to spend it." Allocate as much memory as you can, as often as you can. Collect it, save it for a rainy day. If you end up never using it, don't worry. It was just memory.

Since memory is virtual, a query as to how much memory is available will almost always be "Plenty more where that came from." When you issue this query, abuse the result by attempting to allocate everything that was returned. Since disk space doubles as virtual memory, this will simultaneously consume all the memory in the machine as well as all the available disk space, thereby killing two birds with one stone. What's more, you will slow down the machine immensely; after all, allocating 200MB of disk space takes time. (Hey, 200MB was a lot of disk space in 1995.) And still better, you will prevent Windows 95 from being able to allocate memory for annoying things like the Ctrl+Alt+Del dialog box. This dialog box is clearly a waste. After all, the user paid good money for your awesome program in order to run it, not exit it!

Maxim 13: Multitasking is for wimps.

This is such a popular technique, it deserves lots of examples.

Example 22

For starters, you can employ the following simple pseudocode algorithm used by numerous popular (and unpopular) programs, including Program 6 and what we will call Program 22, as well as two programs by Publisher 22:

```
dwMem = DPMI_QueryFree();
AddToHeap(DPMI_Allocate(dwMem), dwMem);
```

Example 23

But that's too simple. You can get a little more aggressive by using the following algorithm popularized by several programs, including one we will call Program 23.

```
 while ((dwMem = DPMI_QueryFree()) != 0) {
     AddToHeap(DPMI_Allocate(dwMem), dwMem);
}
```

Example 24

But those are still rather tame; you can do better. For example, you can use the method employed by a very popular program (which we will call Program 24a) from Publisher 3:

```
for (dwMem = 0x1000000; dwMem; dwMem -= 0x1000) {
     while ((hmem = DPMI_Allocate(dwMem)) != 0) {
          AddToHeap(hmem, dwMem);
     }
}
```

Two more programs, which we will call Program 24b (from Publisher 24) and Program 24c (from Publisher 3) use related but slightly more convoluted algorithms.

By employing a loop like this, not only will you suck up all the memory on the machine, but by attempting to allocate monstrously huge values that you thought were sure to fail, you can thrash the disk quite well while you are at it.

Example 25

Program 25 from Publisher 3 uses a slightly different twist on the same theme. The overall loop looks familiar:

```
for (dwMem = 0x800000; dwMem > 0x4000; dwMem -= 0x1000) {
      fSuccess = AllocateMem(&info, dwMem);
```

```
        if (fSuccess) {
            dwMem += info.dwAddr;
            if (mi.dwAddr > 30 * 1024 * 1024) {
                continue;           /* Throw away high blocks */
            }
        }
        if (fSuccess) {
            AddToHeap(&info);
        }
    }
```

Observe that this loop will iterate ***at least*** 512 times, and typically more than 1,024 times, which explains why Program 25 takes forever to load.

But wait, there's more. I didn't tell you about the `AllocateMem` function. It first queries the amount of available free DPMI memory and allocates via DPMI if the memory is available. Failing that, it queries the amount of available free XMS memory and allocates via XMS if the memory is available. Moreover, in the XMS case, it locks the memory block in an attempt to obtain its physical address and fails if the lock cannot be obtained. This is clever for two reasons: (1) Virtual memory operating systems won't allow the lock to succeed, because the physical memory can get swapped out; and (2) asking for XMS memory after DPMI memory is pointless, because the two come from the same memory pool, anyway.

Maxim 14: *Persistence may not help, but it's fun.*

Example 26

Another program doesn't even waste its time looking at the correct field in the DPMI memory information structure. Instead of looking at the amount of available unlocked memory, locked memory, or even the largest contiguous free block of memory, the program looks at the total available linear address space. It then goes into the following loop:

```
again:
    for (dwMem = dwLinFree; dwMem > 0; dwMem -= 0x8000) {
        if ((hmem = DPMI_Allocate(dwMem)) != 0) {
            AddToHeap(hmem, dwMem);
            goto again;
        }
}
```

Under a virtual memory operating system, total available linear address space can be as high as 4GB, so the loop will take an awfully long time to finish. (This was back in the days when 16MB was a lot of memory.)

Example 27

We're still only scratching the surface. Multiple programs, including one from Publisher 27 and another, which we will call Program 27, combine the two popular techniques in a creative way:

```
while ((dwMem = DPMI_QueryFree()) != 0) {
     AddtoHeap(DPMI_Allocate(dwMem), dwMem);
}
for (dwMem = 0x11000; dwMem > 0; dwMem -= 0x1000) {
     if ((hmem = DPMI_Allocate(dwMem)) != 0) {
          AddToHeap(hmem, dwMem);
          dwMem = 0x11000;
     }
}
```

Observe that every time an allocation succeeds, it completely resets its memory allocation loop and starts all over.

Example 28

Yet another program puts its own personal stamp on an old chestnut.

```
for (dwMem = 32 * 1024 * 1024; !TryToGet(dwMem);
     dwMem -= 0x10000) { }
```

The twist is that if there happens to be exactly no DPMI memory available after the program is loaded, the program goes into an infinite loop.

Example 29

It helps to couple the allocation of all memory (and therefore disk space) with a free disk space check. This technique is carried out to great effect by Program 22, the install programs for Program 24b, anything by Publisher 27, a popular game from Publisher 3, and Program 23. Remarkably, after drying out your disk for swap space, Program 23 complains that there is not enough available disk space to run or install.

Touch all memory after allocating it

For good measure, after allocating all the memory in the machine, touch every byte. That way, if the memory is virtual, the machine will thrash furiously, trying to page in several megabytes of data.

Maxim 15: *Thrashing is good. It reminds the user that the computer is on.*

Once more, we illustrate this point with a series of examples.

Example 30

One program from Publisher 27 allocates all the memory it can, and then fills that memory with the value `0xCC`.

Example 31

Another program from Publisher 31 allocates all the memory it can, and each time you start a new mission, it zeroes out everything.

Example 32

Program 27 periodically performs monstrous memory move operations. I've seen as much as 600KB of memory being shuffled by a single instruction. And remember that this means that twice as much memory, or more than 1MB, is being touched, source and destination. Doing more than 1MB of disk I/O causes the machine to slow down substantially, and the sound gets choppy as a result. Moreover, the pattern of memory access used by the program runs counter to the principle of locality of reference, so there is a lot more thrashing going on than really necessary.

Assume that the amount of memory never changes

In a multitasking operating system with virtual memory, the amount of available memory changes dynamically, depending on the memory requirements being placed on the system at any particular moment. Make sure to design your program to behave erratically if the amount of available memory changes.

Example 33

Program 24b first allocates all the memory in the system into a "spare pool." When it needs to allocate memory from the pool, it first attempts to grow the spare pool; if this succeeds, the application crashes. (How could I possibly make this up?) Otherwise, it shrinks the spare pool by the amount it wants to allocate, then allocates the desired value. If this second allocation fails, the application crashes.

Example 34

A very popular productivity application, which we will call Program 34, allocates all the expanded memory it can, then frees it, but assumes that whatever memory the program was ever able to get in the past should be obtainable again in the future. In particular, it means that you cannot run two copies of this program (or a copy of it at the same time that another application that uses expanded memory is running). Once one of its memory allocations that it expected to succeed fails, the program displays a consistency failure error message and spontaneously exits.

Access memory after freeing it

In an unprotected operating system, memory you freed is still there, so you may as well go ahead and touch it. Indeed, if you don't allocate any new memory, the values that used to be in the block are still there, so feel free to access them, use them, store to them, treat them like family. Exploiting this feature ensures that any attempt to impose newfangled concepts like memory protection are doomed to failure.

Maxim 16: *Random addresses are harmless.*

Example 35

Program 13 frees its data segment during its termination phases, but continues to use it afterward.

Example 36

One program deserves a special award for the code that gets executed when you save a large scenario. It frees a data page, then allocates and frees bunches of more memory, and then a while later, goes back and touches the data page that it freed a long time ago. (Ironically, after writing this program, the author came to work for Microsoft on the Windows 95 team! When he learned that the application compatibility team had discovered and worked around his bug, he apologized to me in person.)

Example 37

The sound card configuration program for Program 19 does roughly the same thing. It allocates a page of memory, then frees it, then allocates and frees some more memory, and then touches the original page that it freed a long time ago.

Example 38

Program 24b will access freed memory if you go back and forth between the main menu and the title sequence too many times.

Example 39

Program 21a frequently frees buffers twice (usually on the exit path). The second attempt to free the memory will either fail or (if you are particularly unlucky) free memory that belongs to some other program.

Example 40

Yet another program uses freed memory in a very clever way. It wants to reallocate a 32-bit memory block, so it calls the DPMI "resize memory block" function. The twist here is that the memory block it is resizing happens to contain the timer interrupt handler. To satisfy the resize, the operating system may move the memory to a different linear address. If it does, then the next timer tick will jump into memory that has been freed.

Crash if you have too much memory

Remember, machines with a lot of memory probably aren't wasting their time running MS-DOS, so you may as well assume that you are dealing with a machine with less than 16MB of memory.

Example 41

One game tries to allocate all available memory, but overflows a counter if the total amount of memory exceeds 16MB. (Apparently, the authors decided that maybe machines with 16MB are worth running on, because they subsequently released a patch to work around the problem.)

Example 42

Program 42 by Publisher 27 will go haywire if you have more than 64MB of memory. (See more details in Example 48.)

Example 43

If you run a certain program with more than 32MB of expanded memory, you get the following paradoxical error message:

```
Program 43 requires 560K of free low memory and 1024k of free EMS
(expanded) memory to play. You currently have 599K free low memory
and 65536K free EMS memory. Try moving your TSRs and device drivers
into high memory, or remove them completely. Be sure you are not
running a DOS shell from Windows.
```

A similar message is produced by another title from the same publisher.

Example 44

If you have more than 32MB of expanded memory when running a particular program, you get the even less helpful message:

```
Insufficient expanded memory (EMS) is available. A minimum of 512K
is required to run Program 44. Refer to trouble shooting guide on
the inside back cover of the user's manual.
```

Example 45

Program 34 tries to allocate all the available extended memory by first querying the driver as to how much memory there is, and then attempting to allocate it all under one gigantic handle. If this fails, the program gets totally confused and prints "Unexpected condition: __INITEMSPAGE – Leaving *Program* 34" and then exits. The reason it fails is that handles are limited in size to 16MB per allocation.

Example 46

A different game keeps a table of all the EMS handles that it has allocated, so that it can free them later. This table is hard-coded to 64 entries. Once the program allocates its 65th EMS memory block, it overflows its buffer and corrupts the variable that happens to be stored just past the end of the array. Unfortunately, this variable holds the location of the EMS page frame. As a result, the program attempts to access its EMS memory at the wrong location and ends up corrupting memory pretty badly.

Example 47

Both a system diagnostic tool and a game from Publisher 3 try to determine the amount of memory installed in the machine by querying the CMOS memory. CMOS memory is a very small amount of battery-backed memory that remembers hardware configuration information. The information we're interested in right now is a 16-bit value that specifies the amount of memory in the computer, in kilobytes. The programs interpret the value as a 16-bit signed value, which means that if you have more than 32MB of memory, the programs report that your machine contains a negative amount of memory. The game from Publisher 3 gets particularly confused by this. (Moreover, if the machine contains more than 64MB of memory, the CMOS value will be incorrect, because the maximum amount of memory the CMOS can describe is 65,535KB, or approximately 64MB.)

Assume that linear addresses are physical

Any system in which linear addresses are not physical is obviously imposing too much overhead on your program to make it worth running. It is best to assume that linear addresses are also physical, and preferably in subtle ways, so that the operating system can't possibly work around your little foibles. Fortunately, this is very easy to do.

Example 48

The memory manager used by Program 42 assumes that the values returned by the DPMI `PageAllocate` function will always be less than 64MB, because, in an attempt to conserve memory, it remembers only the bottom 26 bits of the address. Consequently, when the program tries to allocate memory, it receives an incorrect pointer, and the application crashes shortly thereafter.

Example 49

Program 25 throws away any memory that lives above the 30MB linear address. (See the code snippet from Example 25.) The intent is to use only the first 30MB of memory in the system. Since DPMI can put memory anywhere, this tends to throw away memory randomly. If the DPMI server happens to map the virtual memory to high linear addresses, all the memory will be thrown away. This explains why Program 25 reports less than 2MB of memory and why it hits internal assertion failures once it finishes its memory check.

Example 50

The routine that plays the publisher's logo in the demonstration version of Program 25 assumes that the values returned by the DPMI `PageAllocate` function will always be less than 2GB. The problem is that it uses the high bit of the address to indicate that the value is not really a pointer, but rather that the lower 31 bits should be interpreted as an array index. When memory is granted above the 2GB boundary, the engine interprets the address as a really huge array index and faults when it tries to access that nonexisting array element.

Example 51

Another program similarly assumes that all addresses are less than 2GB because it uses the high bit of the address as a flag. As a result, when the program masks that bit off, it ends up with an invalid address and crashes.

Example 52

The same two programs from Publisher 22 that were highlighted in Example 22 first allocate all the memory in the traditional way and store the results as follows:

```
int DPMIMem = DPMI_QueryFreeMemory();
int DPMIStart = DPMI_Alloc(DPMIMem);
int DPMIEnd = DPMIStart + DPMIMem;
```

Then their `malloc` function performs signed comparisons to determine if they've run out of memory:

```
malloc(int cb) {
    cb = (cb + 15) & ~15;        /* Round up to paragraph */
    if (cb > DPMIEnd) {       /* BUG HERE! */
        return 0;                /* Out of memory */
    }
    ...
}
```

If the DPMI memory is allocated above the 2GB boundary, this signed comparison will make the programs think that they have a negative amount of memory, so `malloc` returns zero. Other parts of the application don't check the return value from `malloc` and crash when they try to use that null pointer.

Example 53

This program provides an excellent example of a subtle reliance on physical addresses. The program has a little routine that does a binary search. The central step in a binary search is locating the middle element of the region being searched. The program decides to do this by adding the lower boundary to the upper boundary, then dividing by two. In pseudo-C, the sequence is as follows:

```
middle = (low + high) / 2;
```

If memory is allocated above the 2GB boundary, then the sum will overflow, and the result of the division is consequently not what the programmer intended. The program crashes soon afterward. In assembly language, this exhibits itself as a one-instruction error:

```
mov      ebx, esi
add      ebx, edi
shr      ebx, 1       ; BUG!  Should be rcr ebx, 1
```

***Maxim 17:** If you're going to fail, do so as subtly as possible.*

Even though the DPMI server can put the memory anywhere it wants, feel free to assume that the address will always satisfy some numeric criterion that is most convenient to you. In particular, the most significant bit or bits of the DPMI address can be used as tag bits, so that you crash if the DPMI server returns addresses that are "too large."

Example 54

Two titles from Publisher 31 have the converse problem: They think physical addresses are linear. These programs try to be good citizens and use the VDS (Virtual DMA Services) functions to manipulate the Direct Memory Access (DMA) channel, in this case, to communicate with the sound card. They even call function `8103h` (Lock DMA Buffer) to inform the operating system of where their sound buffer is. In exchange, the operating system returns the physical address of the DMA buffer, so that the program can set the value into the DMA controller.

But these applications try to do more with the value. They treat this physical address as a linear address and shovel sound data into that location, thinking that the card will pick it up from there. As a result, the application ends up writing sound data into randomly generated locations in memory. Depending on how lucky you are, you might just get static for sound, or you might explode the machine.

Abuse memory locking services

Memory lock calls are ignored in the absence of virtual memory, so make sure to abuse lock functions as much as possible. That way, when you are run in a virtual memory environment, strange and wondrous things start happening that never happened in real mode.

Since none of the standard DOS extenders use virtual memory by default, this is an excellent opportunity to begin behaving randomly. This is so important it's worth restating.

***Maxim 18:** Do something that is supported only under Windows, and do it wrong.*

Lock the wrong thing

The easiest way to abuse memory locking services is simply to pass the wrong thing to the lock and unlock services. Remember, since most DOS extenders don't support virtual memory, these code paths are exercised only when run under Windows. Do your best to ensure that they are buggy. The DPMI specification for the page lock service reads

AX	= 0600h
BX:CX	= Starting 32-bit linear address of memory to lock
SI:DI	= Size of region to lock in bytes

Interpret this incorrectly in as many ways as you can muster.

Example 55

A major productivity application reinterprets the BX:CX parameter as a segment:offset value. Moreover (so the logic must have gone), since the program is a 32-bit program, the value must really be BX:ECX. So it kindly passes BX equal to its data selector and ECX equal to the offset within the data segment. Nice try. And, of course, it meticulously checks the return value and crashes if the lock fails.

Maxim 19: Second-guess the specification whenever possible.

Example 56

The twist taken by a different utility program is to pass garbage in CX and to pass the size of the region in DI:SI instead of SI:DI. This is so messed up, it's sad.

Maxim 20: Knowledge of the English language is optional.

Example 57

Yet another example comes from a different program, which decides to pass the address to lock in CX:DX instead of BX:CX.

Example 58

But the final insult comes from Program 24c (from the infamous Publisher 3), which passes an address of `0x661E5350` and a length of `0xECE1ACB0`. These two values are *hard-coded* into the application. (They are computed from global variables whose values are never modified.) What's more, the program checks for errors like so:

```
int      31h            ; Call DPMI
jc       @F             ; On error, leave error code in EAX
xor      eax, eax       ; On success, set EAX to zero
mov      eax, something ; overwrite EAX with another value
```

It carefully checks for an error and then ignores the result of the check.

Lock too much

A popular way to abuse memory locking services is simply to lock everything in sight. By locking your memory, you force other applications, the disk and CD-ROM cache and prefetch buffers, and even the operating system itself to get paged out. Go for broke and lock everything, whether you need to or not. That way, your program will be sure to own practically every single byte of memory in the entire machine, save for a few kilobytes of memory, through which the entire remainder of the system will be swapping like mad.

A well-behaved program, on the other hand, locks only memory that will be accessed at hardware interrupt time or memory for which rapid access is crucial to functioning.

Maxim 21: *Well-behaved programs are for wimps.*

Example 59

Program 24a from our friends at Publisher 3 allocates all the memory it can and locks it to speed up its CD-ROM transfer rate. In Windows 95, the CD-ROM file system driver is itself pageable, so locking everything forces the CD-ROM file system to be paged out, thus slowing down the CD-ROM transfer rate.

Maxim 22: *Bite the hand that feeds you.*

Lock too little

Another popular way to abuse memory locking services is to forget to lock something important. Section 2.4.1 (Hardware Interrupts) of the DPMI 0.9 specification describes the importance of locking certain memory:

> Hardware interrupt procedures and all of their data must reside in locked memory. *All memory that is touched by hardware interrupt hooks must be locked.*

***Maxim 23:** Words in italics don't count.*

You can combine this advice with the "Lock too much" recommendation to get a double whammy. By locking too much, you cause the important thing you forgot to lock to get paged out more frequently, thus exacerbating the problem.

Example 60

One game (that I personally enjoy playing) locks all the memory it allocates, as well as many parts of conventional memory, but it neglects to lock its conventional-memory music driver, which is called at hardware interrupt time. To ensure erratic behavior, leave things unlocked that should have been locked.

Abuse virtual memory

Never use the DPMI "Get Version" function (0400h), which tells you whether the current DPMI server is using virtual memory. Always assume that your memory is physical. So go ahead and allocate all the memory you can (it must be physical, after all), and use the extra memory as a giant disk cache. That way, if the user is running in a virtual memory system, you get performance degradation instead of performance improvements, because your putative disk cache is actually swap file space.

Actually, if you want to make sure that you don't run under Windows 95, you can make the Get Version call and refuse to run if virtual memory is present or if you cannot put the CPU into real mode.

Example 61

Program 6 is just one of many programs that try to allocate as much memory as possible and use it as a disk cache.

Example 62

Some versions of one particular game will display the error message

```
DOS/16M error: [31] : protected mode already in use in this virtual
machine
```

if you try to run them.

Rely on the high word of the ESP register

It is an unfortunate feature of the 80386 architecture that the high word of the ESP register is not restored properly on a privilege transition from a 32-bit stack selector to a 16-bit stack selector. This means that when a program is executing on a 16-bit stack, the high word of the ESP register changes in ways completely beyond your control. This is not a problem unless you try to mix 32-bit and 16-bit code. For example, if you write

```
        mov ebp, esp
        mov eax, [ebp]
```

in code that is executed on a 16-bit stack, you will probably fault because a garbage high word gets copied from ESP to EBP, which then gets dereferenced as a 32-bit address instead of a 16-bit address.

This doesn't impact most programs, because stack-based instructions (like `push` and `call`) will use only the low word of the ESP register if the stack selector is only 16-bit. However, your program is probably not like most other programs. Take care to make these sorts of invalid assumptions when executing on a 16-bit stack.

Example 63

Many programs by Publisher 63 assume that the high word of the ESP register is always zero while the program is on a 16-bit stack. If the high word ever becomes nonzero, the application crashes.

Example 64

A program from Publisher 24 uses the following code sequence:

```
        cli
        and     esp, 0000FFF8h
        call    fword ptr ...
```

The authors of the program think that the `cli` saves them. It doesn't. Just because interrupts are virtually disabled doesn't mean that the operating system won't interrupt you and run somebody else for a while.

Ignore the DPMI specification

Although this entire section deals with ways of ignoring the DPMI specification, there are some ways to go blatantly out of your way to ensure that you break as many rules as possible.

The first sentence of Chapter 6 ("Terminating a Protected Mode Program") of the DPMI 0.9 specification describes how to terminate a program, and even highlights a detail with italics:

> To terminate a protected mode program, execute an Int 21h with AH = 4 Ch *in protected mode*.

Example 65

Program 24a (from Publisher 3) decided to issue the Int 21h with AH = 4Ch in real mode instead of protected mode. This is yet another illustration of the maxim "Words in italics don't count."

Contain lots of bugs

Another easy way to make sure that your program doesn't run under Windows 95 is to contain flat-out bugs. If you can't come up with bugs of your own, here are some suggestions.

Defy the operating system

It's particularly gratifying to code up bugs that are uncovered only if there is an operating system watching over your shoulder. Serves them right for trying to snoop on what you're doing.

Example 66

One of the programs we encountered in Example 64 tries to set a protected-mode interrupt vector by passing a data selector in `cx` instead of a code selector. I have no clue how this program ever worked at all.

Maxim 24: If you can't be subtle, then be blatant.

Example 67

One game will sometimes access memory beyond the end of its stack. In the absence of memory protection, this memory access will either touch the next page of memory (random memory corruption) or grab bus noise (harmless). In a protected operating system, attempting to access memory that doesn't exist results in a page fault and death.

Example 68

Yet another game produces a creative reinterpretation of the VDS specification. When the program passes its sound buffer to VDS, it does not pass the flag that says, "I am a bad boy and will not use VDS services to access the DMA buffer." VDS therefore thinks that the application is a good boy and relocates the DMA buffer to a better location. The game, however, doesn't care about that possibility and proceeds on the assumption that the buffer it passed in was a valid DMA buffer that did not need to be relocated. Then it fills its DMA buffer with sound and tells the sound card to start playing. The card plays the sound out of the relocated buffer, not the original buffer, and you get static instead of sound effects.

Look where you shouldn't

Another way to make sure any system upgrade will break your program is to crawl into undocumented data structures.

Example 69

A top-selling productivity application walks the System File Table (SFT) chain at startup. This is an internal, undocumented MS-DOS data structure that everybody knows about, anyway, so it's de facto documented. But that's not where the problem is. The authors forgot how the Intel architecture resolves segment:offset pointers to linear addresses. The correct formula for an address of the form `AAAA:BBBB` is to convert it to `000AAAA0 + 0000BBBB`. But here's what the program did instead:

```
mov     eax, lpNextSFT  ; EAX = AAAABBBB
                        ; Load the 32-bit value
mov     ebx, eax        ; EBX = AAAABBBB
                        ; copy it to EBX
shr     eax, 12         ; EAX = 000AAAAB
                        ; shift right 12 bits
and     ebx, 0FFFFh     ; EBX = 0000BBBB
                        ; mask off top 16 bits
cmp     ebx, 0FFFFh     ; Test for end of list
jz      done            ; End found, finished
add     eax, ebx        ; EAX = 000AAAAB + 0000BBBB (?)
```

(The actual code is not exactly like this, but it is essentially the same.) This code computes the address incorrectly if the offset part of the pointer is greater than 4095. Since the SFT is stored as a linked list, one incorrect address computation results in the program wandering randomly through memory until it finally crashes.

Configuration-specific bugs

Or you can simply add bugs that are uncovered only by particular system configurations. These bugs would have also manifested themselves under DOS, but that's okay. Your customer support can just say that it's a bug in Windows 95. Customers will believe you. (The examples in this section were actually reported as bugs in Windows 95.)

Example 70

One particularly resourceful program, as part of what must be its copy protection, frequently runs a subroutine that does CPU detection, which in turn generates code on the fly and calls it. What type of code the program generates depends on the type of CPU detected. The code that is generated for 80386s reads from privileged system registers, which is disallowed in a protected operating system. As a result, the program crashes when it tries to touch these registers.

This program claimed to run under Windows 3.1, so the question now is how the program managed to survive in spite of this obvious goof-up. It turns out that one of the conditions that contributes to the on-the-fly generated code is whether the word "EMM386" appears in the file C:\CONFIG.SYS (note that the filename is hard-coded, which is a completely separate issue). If the six letters appear anywhere in the file, be it in a comment or a syntax error or wherever else, then the program doesn't touch the privileged registers.

Thus, even under Windows 3.1, the program fails to run unless you mention "EMM386" somewhere in C:\CONFIG.SYS.The Intel manual says

> *Software which uses [test registers] may be incompatible with future processors.*

And even under plain old MS-DOS with an empty *config.sys*, the program crashes if you have a Pentium, because one of the registers the program tries to access no longer exists on that chip.

Maxim 25: *Intel will never release a new CPU.*

Example 71

Program 71 is a game that crashes after you get past the title screen. On certain machines, the program spends a lot of its time trying to negotiate audio control on the CD-ROM by executing the following routine:

```
/*
 * SendCDROMRequest: Call real-mode interrupt 2Fh
 *                                    function 1510h,
 * "Send CD-ROM device driver request".
 * This function requires ES:BX -> request packet,
 * CX = CD-ROM drive letter (0=A, 1=B, ...)
 */
void SendCDROMRequest(unsigned long PacketLowLinear)
{
  struct _SREGS sregs;      /* segment registers go here */
  struct _REGS regs;        /* arithmetic registers go here */
  regs.w.ax = 0x1510;       /* AX = function code */
  regs.w.cx = CDROMDrive;  /* CX = drive letter */
  sregs.es = PacketLowLinear >> 4; /* Convert to seg:off */
  regs.w.bx = PacketLowLinear & 15;
  rmint86x(0x2F, &regs, &regs, &sregs);
}
```

This function looks rather harmless by itself, but let's look at the `rmint86x` function:

```
/*
 *  rmint86x - Reflect an interrupt into real mode.
 */
void rmint86x(int n, struct _REGS *in,
                      struct _REGS *out, struct SREGS *sr)
{
  struct DPMI_REALMODEREGS rmr = { 0 };
  struct _SREGS sregs = { 0 };
  struct _REGS regs = { 0 };
  /* Copy the "in" registers to the rmr structure */
  rmr.eax = in->w.ax; rmr.ebx = in->w.bx;
  rmr.ecx = in->w.cx; rmr.edx = in->w.dx;
  rmr.esi = in->w.si; rmr.edi = in->w.di;
  rmr.flags = in->w.cflag & 1;
  /* Copy the "sr" registers to the rmr structure */
  rmr.es = sr->es; rmr.cs = sr->cs;
  rmr.ss = sr->ss; rmr.ds = sr->ds;

  /*
   * To issue the interrupt, we must set ES:EDI to point to the
   * rmr structure we just initialized, AX to the function
   * code 0x0300, and BX to the interrupt we wish to call.
   */
  sregs.es = GetSS();
  regs.x.edi = (unsigned long)&rmr;
  regs.w.ax = 0x0300;   /* Function code to call an interrupt */
  regs.w.bx = n;
```

```
    int86x(0x31, &regs, &regs, &sregs);
    ... /* Copy results from regs into out */
}
```

Again, it looks harmless by itself. But now look at how the two functions interact. The `rmint86x` function loads `sr->ss` into `rmr.ss`, but `SendCDROMRequest` never initializes the field. The interrupt, therefore, is executed on a randomly selected stack somewhere in no-man's-land, corrupting memory randomly and generally causing a mess. Surprisingly, the program doesn't seem to mind random memory corruption! It only keels over when the randomly selected stack points into the ROM BIOS. Since ROMs are read-only, pushing a value and then popping it won't give you the same value back.

Example 72

One particular game will crash if your `BLASTER` environment variable does not contain the letter `T` in its value.

Example 73

The install program for a tremendously important productivity application will hang if you are installing to a network drive larger than 4GB, because the program goes into an infinite loop trying to figure out how much disk space the install is going to require. The bug is particularly amusing, because this version is the special-edition *network version* of the program, and network servers are the ones most likely to have a lot of disk capacity.

Example 74

Program 34 has a similar problem. It complains about insufficient disk space if the drive that holds the database has more than 2GB of disk space free.

MS-DOS programs

The following remarks apply to MS-DOS programs, whether DOS-extended or conventional.

Checking the amount of memory available

Remember that just because the operating system provides a "Query available memory" function doesn't mean you should use it.

Example 75

The installer for all programs from Publisher 27 uses the following code fragment:

```
for (kbAlloc = 1; (hmem = XMS_Allocate_KB(kbAlloc)) != 0;
      kbAlloc++) {
     XMS_Free(hmem);
}
```

Using a pathetically horrid algorithm, this fragment takes a torturous path to determine how much extended memory is installed. A special XMS service, Query Free Extended Memory, returns the answer directly, but that would make things too easy. Better to go into an $O(n)$ loop to find out how much memory there is.

Maxim 26: *Slower is better.*

Moreover, the technique from Example 75 truly thrashes the disk, for each allocation causes the swap file to grow, and each free causes the swap file to shrink. Since each allocation and free requires $O(n)$ of work to obtain and release the appropriate amount of disk space, the algorithm's running time increases to $O(n^2)$.

We watched this algorithm in action on a machine with 96MB of extended memory and 1GB of disk space. (Hey, that was an unimaginably big machine at the time.) It was quite amusing. Note that a 16-bit integer value is used to hold the amount of memory, so the loop runs forever on a machine with more than 64MB of memory.

Example 76

The install program for Program 21b is particularly gruesome. To determine how much memory is installed in the machine, it uses the following code fragment:

```
for (cbAlloc = 1024; (p = malloc(cbAlloc)) != NULL;
      cbAlloc += 1024) {
     free(p);
}
```

This is already problematic for reasons just explained, but there's more. If the `malloc` function is unable to satisfy the allocation from its own memory pool, it calls the DPMI memory allocation function to get more. In the memory obtained from DPMI, the `malloc` function stores bookkeeping information in the first few bytes and the last few bytes before returning it to the caller. Correspondingly, the `free` function checks if the memory being freed is the last element of a chunk of memory allocated from DPMI; if so, then it releases the memory back to the DPMI server.

The net effect of all this on the server is as if the code were written like this:

```
for (cbAlloc = 4096; p = DPMI_Allocate(cbAlloc);
     cbAlloc += 4096) {
     p[0] = 0;              /* Touch the first byte */
     p[cbAlloc-1] = 0;      /* Touch the last byte */
     DPMI_Free(p);
}
```

This thrashes the machine pretty badly on multiple levels. The installation also takes more than five minutes to run to completion on some machines, which would make the user think that the program is hung if it weren't for all that disk activity. This brings us back to Maxim 15: Thrashing is good. It reminds the user that the computer is on.

Example 77

One program claims in its manual to require 2.5MB of extended memory, but when the program does its memory check, it looks at the wrong field. Instead of looking at the total free XMS memory, it looks at the largest free block to determine whether you have enough memory. Never mind that it does cope with fragmented XMS memory.

Hook interrupts incorrectly

Hardware interrupt handlers must preserve all registers. You can do interesting things if you decide to break the rule.

Example 78

Program 22 installs a timer tick handler that, once a second, goes off and does some nontrivial work in code written in the C language. The assembly-language wrapper saves all the 16-bit registers before calling the C routine. Unfortunately, the C part was written with 32-bit instructions enabled, so the high words of extended registers get trashed and are never restored. The result is that every second, the program destroys the high words of its own registers.

***Maxim 27:** If you can't convince the operating system to screw you up, take matters into your own hands.*

Example 79

An MS-DOS network driver installs a hook for interrupt 21h, which finishes up with code similar to this:

```
inc      reenter_count
pushf
call     prevInt21Vector ; Send the call down the chain
push     bp
mov      bp, sp
pushf
pop      [bp+6]            ; Pop flags into iret frame
pop      bp
dec      reenter_count
iret
```

(I say ”similar to” because the true code is actually buggier than even this.) If you don’t understand the assembly language, don’t worry. The intent is to return the original flags from the interrupt back to the caller. But software interrupts are dispatched with interrupts disabled, so the attempt to pop the flags into the `iret` frame will pop a disabled interrupt flag, which will be picked up by the `iret`. (This code runs in 8086 emulation mode, not protected mode, so the remarks we saw regarding interrupt flag management in DPMI do not apply here.) The result is that the call returns with interrupts disabled.

The proper way of hooking the interrupt vector is to write

```
inc      reenter_count
pushf
call     prevInt21Vector ; Send the call down the chain
pushf
dec      reenter_count
popf
retf     2
```

It is not only correct, but smaller as well.

Example 80

An electronic mail program uses a driver that hooks Int 21h, which actually scrambles registers on DOS calls that it does not understand. As a result, with the driver installed, disk utilities that use the new, long filename services and IOCTLs begin acting very oddly because they get random results from file system calls.

Maxim 28: *Microsoft will never release a new version of the operating system.*

Example 81

Program 71 does what many games do: speed up the timer, and trickle timer ticks to the previous handler at the original rate. For example, if you speed up the timer fourfold, then one out of every four timer ticks should be delivered to the original timer handler. This program's timer tick handler is quite complicated and is not worth reprinting here, but here's an equivalent formulation:

```
[if this tick should be trickled]
    pushf
    call     dword ptr PrevInt8Hook
[endif]
    mov      al, 20h
    out      20h, al          ; Nonspecific EOI
[restore registers and return]
```

Before I can explain what a "Nonspecific EOI" is, I have to explain how the programmable interrupt controller (PIC) works. The job of the PIC is to manage all the hardware interrupts in the system. Each hardware device is assigned a priority. The timer has the highest priority, the keyboard the next, and so on. When a hardware device raises an interrupt, the PIC compares that device's priority against the priority of the interrupt currently being handled (if any). If the new interrupt is higher priority, then it interrupts the old interrupt. If it is less than or equal to the current interrupt, then the new interrupt must wait until the old interrupt is complete. Notice that if a hardware device raises an interrupt while its previous interrupt is still outstanding, the new interrupt must wait since the old and new interrupts have equal priority. (This description is simplified but it's good enough for the purpose of this story.)

To indicate that the processing of an interrupt is complete, a hardware interrupt handler issues an "End of interrupt" command (EOI). Once an interrupt is acknowledged, the PIC will permit a pending interrupt of equal or lower priority to be generated.

There are two ways to acknowledge an interrupt. One is to issue a "specific EOI," wherein you tell the PIC which IRQ you are acknowledging. (In this case, we would say that we are acknowledging a timer interrupt.) The other is to issue a "nonspecific EOI," which acknowledges whatever interrupt is currently active.

Given this background on how to conclude an interrupt handler, perhaps you can spot the problem here. When the timer tick is trickled through, the original timer tick handler will perform the EOI on the interrupt. Then this routine will perform another EOI, and even worse, a nonspecific EOI. If the timer tick interrupted another interrupt, this ends up acknowledging the other interrupt. This typically confuses the handler of the other interrupt because you have told the PIC, "Go ahead and let another interrupt occur from that device."

If that device happens to have an interrupt waiting, then the interrupt handler will be reentered when it thought it had blocked reentry by not acknowledging the interrupt. Reentering non-reentrant code leads to what can be euphemistically described as suboptimal behavior.

Use uninitialized variables

Uninitialized variables are a very popular way of making sure that the user uses a boot disk. (Let's face it. Nobody likes boot disks.) The boot disk trick works merely accidentally rather than by design, because it forces the user to start the system afresh, at which point all unused memory gets initialized to zeros by the BIOS power-on self-test.

Example 82

Both a business productivity application and a very popular game assume that uninitialized variables have the value zero. There isn't much that can be said aside from the fact that it happens.

Example 83

A game from Publisher 31 appears to hang when you quit the game. The relevant code looks like this:

```
        mov     cx, 21h
        mov     dx, portnum
@@:     in      al, dx
        loopd   @B              ; BUG! loopd uses ecx, not cx
```

The `loop` instruction decrements the CX register and jumps if the result is not zero. That's clearly what the author of this code intended to use, since it loads the CX register with a constant value. Unfortunately, the actual instruction is `loopd`, and the "d" suffix converts this from a 16-bit instruction to a 32-bit one: The `loopd` instruction decrements the 32-bit ECX register rather than the 16-bit CX register. (The CX register is an alias for the bottom 16 bits of the ECX register.) Since this program didn't initialize the full ECX register, the program pauses for an arbitrary period at shutdown.

Maxim 29: *The high words of 32-bit registers are always zero.*

But actually, the duration is not arbitrary. The high word of the ECX register happens to contain an address. When run outside Windows, the program pauses for a variable amount of time that depends on how much memory you have. In protected mode, the program pauses for an amount of time that depends on where DPMI decided to place the memory. Since Windows 95's DPMI server places the memory at very high addresses, the machine pauses for several hours before finally exiting.

Example 84

Two major productivity applications assume that the high words of all the 32-bit registers are zero at program startup, thereby demonstrating yet another manifestation of Maxim 29. Those programs implicitly assume that they are the only 32-bit programs that run on the machine; they're too lazy to execute eight instructions at the beginning of the program to zero out those 32-bit registers.

Maxim 30: *Even business applications can benefit from a boot disk.*

Error handling

Let's face it, nobody like errors. You can exhibit your displeasure at error conditions by using them to trigger wildly unexpected behavior.

Don't check for errors

By not checking for errors, you can do all sorts of interesting things.

Maxim 31: *Error checking is for wimps.*

If you insist on checking for errors, at least have the presence of mind to do something incredibly stupid when the error occurs.

Example 85

During planet landing, a space-themed game requires thirty XMS handles. The program does not check the return code from the allocation call, and if you run out of XMS handles, the allocation function returns zero. When the application tries to perform an XMS memory move into handle 0, fascinating things happen because handle 0 is an alias for conventional memory. And the first byte of handle 0 is the interrupt vector table. Consider what happens when the program tries to initialize that block of memory it just allocated.

The default configuration of *himem*.sys provides 32 handles, one of which is used by the SmartDrive disk cache, and another of which is used by the MSCDEX CD-ROM driver, leaving exactly thirty handles for applications. If the user were stupid enough to load any other drivers that happened to allocate XMS memory, the game would crash.

Example 86

An educational program for some reason attempts to open the null string as a filename. If the MS-DOS "open" command succeeds, it returns with the carry flag clear and the file handle in the AX register. If it fails, it returns with the carry flag set and an error code in the AX register. In this case, the function returns with carry set and sets the AX register to the error code 2 (file not found). The program ignores the set carry and continues operating on handle 2, which normally corresponds to the standard error output, and the output goes to the screen. So the program works completely by accident. This is another illustration of Maxim 12: It is better to be lucky than good.

Example 87

One game, when it cannot obtain expanded memory, first displays an error message, and then it calls through a null pointer.

Maxim 32: *Don't bother testing the error paths. The game is already over.*

Example 88

If, when checking the DOS version, a disk utility program finds that it is running on DOS 1.0, the program tries to exit in a DOS 1.0-compatible way but gets it fundamentally wrong and crashes instead. Another example of Maxim 32: Don't test the error paths.

Check for errors when there are none

The converse problem is to check for an error return code from a function that doesn't return an error code.

Example 89

A software development system judiciously checks the *command.com* errorlevel variable after every command, including the DOS internal commands. But the DOS internal commands don't set the errorlevel variable. As a result, the program ends up checking whatever random errorlevel is lying around. If the errorlevel happens to be nonzero, the program gives up.

Abuse expanded memory

Many of the techniques that DOS-extended programs use to abuse memory also apply to expanded and extended memory. Included here are additional examples of exploiting the expanded memory architecture in ways not previously possible.

Example 90

This game is actually not a DOS-extended application, but it is listed here because it also allocates all the memory in the system. In this case, the memory in question is expanded memory. Since Windows 95 memory is virtual, this causes the poor machine to thrash its brains out.

Paradoxically, this program runs better the less memory you give it.

Example 91

Two games that were popular in their day initiate disk activity into an EMS page frame. Meanwhile, the programs also have a timer hook that remaps expanded memory at hardware interrupt time. If the timing is right, an expanded memory page could be the target of a disk operation, only to be unmapped while the transfer is in progress. When run outside Windows, in real mode, this works by accident because the disk driver will put the result at the specified address, regardless of whether what is there now is the same as what was there when the I/O was initiated. But in protected mode with 32-bit file access, this essentially pulls the rug out from underneath the file system while the file system thought it owned the disk buffer.

Try to lock XMS memory

The return value from a successful XMS 3.0 "lock" call is the physical address of the contiguous memory block. This is not supported in a virtual memory environment for several reasons:

- The block may not exist in contiguous physical pages.
- The block's physical address may change as the memory is paged in and out.
- Even if the memory were locked and forced physically contiguous, there isn't much that a program can do with the physical address, anyway (although a program can sure try; see "Assume that linear addresses are physical").

Note also that the XMS 2.0 and 3.0 specifications *disagree* on whether the address of a locked XMS block is physical or linear. The 2.0 specification returns a linear address, but the 3.0 specification returns a physical address. Moreover, third-party extended memory managers themselves disagree on whether to return a linear or physical address, so even if you checked the version of the XMS provider, you *still* don't know whether you got a linear or physical address. (I believe that the decision to return physical addresses in 3.0 was a mistake of the highest order.) The moral of the story is not even to bother trying, because you can't even be sure that the answer you get is the answer you expected.

Example 92

A very popular game attempts to lock XMS memory if you configure it to use XMS memory for sound buffers. When the lock attempt fails, an error message is displayed and the program immediately terminates. This is particularly creative, because the way to turn off the use of XMS memory for sound buffers is to run the program and go into the Options menu, which you can't do, because the program quits before it gets to that point.

Example 93

Another game uses XMS memory for its overlay management. At startup, it allocates an XMS block, locks it, saves the locked address, unlocks the block, and frees the block. This is already scary, holding on to the address of a block that no longer exists. The program files away this value; later, when the "real" XMS allocation is made to hold the overlay code, the new allocation is also locked and its address compared with the address of the first block. If they are different, then the application concludes that XMS is not installed. (How could I possibly make this up?)

This is the *only* thing the program does with the locked address. It doesn't care what address it gets, so long as (1) the lock succeeds, and (2) all allocations return the same address. Aside from this brief mental lapse, the program uses standard XMS functions to access the memory; locking the block is completely superfluous.

Rely on race conditions

In a multitasking operating system, you are not guaranteed to get a constant and regular stream of CPU cycles. Design your program to exploit this latency so that any variation in the rate at which instructions are executed will be exhibited as erratic program behavior.

Outrace the timer

The classic way of exploiting race conditions is to misuse the timer. Due to the nature of multitasking, some timer ticks may be delayed. If you program the timer to a high rate, a timer tick may even become delayed by an amount greater than or equal to the timer tick interval. To catch up, the operating system will send down two timer ticks in direct succession, known as a *burst*. (Mind you, the same thing happened in DOS, too, as you may have noticed when we briefly discussed the operation of the PIC in Example 81.) Make sure your program goes berserk when faced with timer tick bursts.

Example 94

A children's title increases the timer rate to four times normal and increments a global variable (which we call `wTicks`) at each tick. Then when it needs to delay for a few ticks, it uses the following procedure:

```
void delay(unsigned nTicks)
{
    unsigned wTickStop = wTicks + nTicks;
    while (wTickStop != wTicks);
}
```

This routine fails in the presence of timer bursts because of the test for equality: If two timer ticks arrive simultaneously just as the delay is about to complete, `wTicks` will jump from `wTickStop-1` to `wTickStop+1`, and the program will remain in the loop for another six hours until the tick count rolls over and reaches `wTickStop` a second time.

The function should have been written like this:

```
void delay(unsigned nTicks)
{
    unsigned wTickStart = wTick;
    while (wTicks - wTickStart < nTicks);
}
```

which operates correctly in the face of both timer tick bursts and timer wraparound. Using an equality test against a timer variable is, sadly, a common occurrence.

Example 95

Due to interesting architectural decisions, the timer tick handlers for the same two programs by Publisher 17 have extremely high latency, so high that timer ticks end up bursting. The programs attempt to calibrate the CPU against the timer, of course using an equality test to break out of the loop. Furthermore, care was taken that the loop will *not* be broken after waiting for the timer to wrap around, because the variable that tracks how much time has elapsed explicitly pegs at 255 instead of wrapping around to zero. So the program really hangs; it doesn't just pause for six hours.

This should be a maxim, but I'm at a loss
for a good way of expressing it.

Example 96

Program 13 uses a related trick. At startup, it tries to calibrate the speed of your machine as follows. The function `csCur` computes the current time in hundredths of a second since midnight.

```
Calibrate(void)
{
     DWORD csEnd, csStart;
     int i, j;

     do {
          /* wait for time to change */
          csStart = csCur();
          while (csCur() != csStart);

          /* remember the starting time */
          csStart = csCur();

          /* do some pointless computation */
          for (i = 0; i < 200; i++)
            for (j = 0; j < 7500; j++);

          /* remember the ending time */
          csEnd = csCur();
     } while (csStart >= csEnd);    /* Avoid midnight bug */
                                  /* and introduce a new one */
     ... finish calibration ...

}
```

On a fast machine, say, a Pentium P5-90 (hey, that was a fast machine back then), the count to $200 \times 7500 = 1{,}500{,}000$ all happens within a single clock tick, so this code spins forever, waiting for the machine to slow down enough that the code can detect a clock tick. The comparison at the bottom of the loop was added to guard against a bad calibration if the program is run too close to midnight. Unfortunately, while fixing that bug, a new one was introduced where fast machines simply hang forever during calibration.

Outrace the keyboard

Race conditions involving the keyboard are never found in real mode, because normal human beings cannot type at thousands of keystrokes per second. But a normal human being *can* out-type a program that is not running at full speed due to multitasking and swapping. As with the timer, burst keyboard interrupts are entirely possible.

Example 97

A program from Publisher 24 as well as Program 24a from the very popular Publisher 3 maintain a table that records which keys are down and which keys are up. On receipt of a keyboard interrupt, the appropriate bit in the table is set or cleared. Meanwhile, the main loop scans the table to decide what needs to be done. As a result, if keypresses are burst into the application, the bit is set and cleared before the main loop even sees the keystroke. For Program 24a, this means that the cloak command sometimes fails to work. The title from Publisher 24 also tracks in a separate variable the scan of the most recently pressed key. Somehow, the fact that the most recent action was a key-up for which there was never the down leads to the curious side effect that rapid presses of the Enter key should be interpreted as a request to exit the game without saving. (I don't know what logic led to that result, but that's what happens. Go figure.)

Example 98

Two programs from Publisher 31 also use the key table technique, but they push the envelope even harder. When a key-down is received, they set the bit. When a key-up is received, they *toggle* the bit. The result is that if a stray key-up is received, the key goes *down* instead of staying up, resulting in what is often called a stuck-key problem.

Outrace the sound card

Another way to rely on races between software and hardware is to program the sound card to raise an interrupt before you are ready to handle the interrupt. That way, when the interrupt arrives during the race window, it gets dropped on the floor and your program gets stuck in a loop waiting for something that already happened.

Example 99

One common method is to write code like this (translated to C):

```
/* initiate a DMA transfer. The hardware interrupt
 * handler will set interrupted = TRUE when the DMA completes */
StartDMA();
interrupted = FALSE;
while (!interrupt) { } /* wait for it to complete */
```

Observe that if the DMA completes before *StartDMA* returns, the program ends up clearing the *interrupted* flag after the value was set by the interrupt handler. Then the program ends up waiting for an interrupt that never arrives.

Outsmart the file system

A common technique is to perform filename validation within the program instead of using the file system. After all, the file system shouldn't be considered the final arbiter on which filenames are legal. In particular, refusal to accept the tilde as a filename character will ensure that long filenames die the horrible death they deserve. (Any filename more than eight characters takes too much work to remember, anyway.)

Example 100

For some reason, the tilde seems to be the character that causes the most trouble. One personal enrichment program displays all files in its directory listing, but if you select a file whose name contains a tilde, it claims the filename is invalid and refuses to load it. One particular personal finance program will load files containing tildes, but will refuse to save them, claiming that the name is illegal. And a data analysis tool, when faced with a filename containing a tilde, simply ignores the tilde and reports "File not found."

Example 101

The winner in this category is a travel planning tool. When you give it a file whose name contains a tilde, it displays the following error message:

```
Program 101: The following critical error has occurred. The files
have been closed. Contact Company 101 if you cannot resolve the
problem based upon the message shown below. Please write down the
debugging information that will be displayed so we can help you
with the problem. Please press the ENTER key now, debugging
information will then appear.

Error DBCMD/1010 Illegal characters in alias: SAMPLE~1
Quit
```

(The remaining debugging spew has been omitted.) Moreover, once this error occurs, the entire application corrupts itself irreparably. You have to delete the program from your hard drive and reinstall it from scratch.

***Maxim 33:** When faced with the unusual, self-destruct.*

Summary of maxims

You can print out these two pages, fold than up, and keep then in your pocket. Well, maybe not, since nobody is really interested in writing MS-DOS-based games, anymore. But if you did, this would have been a handy guide to ensuring that your game doesn't run on Windows 95.

1. The more confusing the error message, the better.
2. If it doesn't run under Windows, then it's Windows' fault.
3. Detect Windows, and crash.
4. Manuals are a waste of time.
5. Words in boldface don't count.
6. Just because you're a Windows application doesn't mean that you have to be compatible with Windows.
7. Performance degradation is acceptable in the greater interest of not running on Windows 95.
8. If you're going to do something wrong, do it wrong in as many different ways as possible.
9. Find a rule and break it as blatantly as possible.
10. The weirder your code, the harder it will be for the operating system to detect that you are about to do something stupid and rescue you.
11. Always look for the weirdest, most convoluted way of doing something. That way, you stand a good chance of doing it wrong.
12. It is better to be lucky than good.
13. Multitasking is for wimps.
14. Persistence may not help, but it's fun.
15. Thrashing is good. It reminds the user that the computer is on.
16. Random addresses are harmless.

17. If you're going to fail, do so as subtly as possible.
18. Do something that is supported only under Windows, and do it wrong.
19. Second-guess the specification whenever possible.
20. Knowledge of the English language is optional.
21. Well-behaved programs are for wimps.
22. Bite the hand that feeds you.
23. Words in italics don't count.
24. If you can't be subtle, then be blatant.
25. Intel will never release a new CPU.
26. Slower is better.
27. If you can't convince the operating system to screw you up, take matters into your own hands.
28. Microsoft will never release a new version of the operating system.
29. The high words of 32-bit registers are always zero.
30. Even business applications can benefit from a boot disk.
31. Error checking is for wimps.
32. Don't bother testing the error paths. The game is already over.
33. When faced with the unusual, self-destruct.

Epilogue

Sometime after Windows 95 shipped and became a smash hit, a group of prominent game developers visited Microsoft as guests of the DirectX team. At some point, they learned of "this guy" on the Windows 95 team who worked so hard at getting all their games to work that he "officially went insane." Thus intrigued, they asked to be introduced to this insane guy, and they presented me with a letter, signed by all of them, with the message, "Thanks for going insane."

I still have that letter.