

Programmer's Journal

July/August 1990 Volume 8.4

Running unprotected

A four megabyte TSR for the 80386

By Dean C. Wills

It's easy to be excited yet terrified about beginning software development in a new environment. You look forward to discovering fascinating new features; at the same time you face the uncertainty and challenge of exploring unfamiliar territory. Intel's 80386 protected mode lives up to your expectation on both accounts. It is enticing but terribly complex. It solves all the classic problems which have plagued DOS for years. 80386 protected mode supports segments greater than 64K, bypasses the 640K barrier, and supports virtual memory and multitasking.

But it isn't free! The prices for OS/2 and DOS-extender compilers and debuggers are high. The cost problem multiplies when you consider how difficult it is to lay down a chunk of money for something that you don't even know how to use. Well, never fear. Here I'll solve this problem by introducing you to protected mode at a discount, using standard DOS tools and MS-DOS. All the code for this article was developed using MASM v5.1.

Let me pause a moment here and acknowledge that it is possible to write application for protected mode without knowing anything about protected mode programming. You simply stick to the application level and let the operating system manage all the new system features and capabilities. This level of programming can be compared to being a C expert without knowing anything about assembly language or the operating system. Where's the fun in that?

VIRTUAL 8086 MODE

This article focuses on the virtual-8086 (V86) mode of the 80386/486. This subset of protected mode is the best place to begin explaining protected mode programming to DOS programmers because it is the mode capable of running DOS programs unmodified. This mode allows the 80386 to multitask several DOS programs along with other protected mode programs. Each DOS program sits inside a virtual-8086 task which emulates a standard 8086 IBM PC running in non-protected mode.

The DOS program feels that it is running by itself in a completely isolated environment with direct access to all hardware, so that it may perform wanton destruction, as it desires. Since this is the way most fast DOS programs run, DOS and V86 mode make a perfect match. V86 mode was designed so well that it is capable of running even the most ill-behaved killer DOS programs imaginable. This mode, I believe, will guarantee the ultimate success of the 386 and its successors; it preserves the valuable DOS investment yet provides all the features for developing powerful 386 protected mode programs.

A V86 MONITOR

The code provided with this article is for an elementary V86 monitor which will put DOS inside a virtual-8086 machine and perform some basic emulation. The program will place the computer in protected mode, create a virtual-8086 task, return to DOS inside of the virtual-8086 task, and execute a DOS terminate and stay resident call. In virtual 8086 mode, all of your familiar DOS programs will still run, and almost as quickly (though note that some instructions that load segment registers or call software interrupts actually take longer in protected mode). But the computer will be in protected mode, and this will open up the capabilities of the machine. This TSR can now be expanded using extended memory to create programs beyond the capability of the 640K DOS machine. Multitasking software, four-megabyte printer buffers, debuggers and LIM expanded memory emulators are just a few of the possibilities.

But be careful! Systems programming on the 386 will probably be the most challenging programming you've ever attempted. Systems assembly language is very different from application assembly language. The instructions you use control the operation of the chip itself. Debugging becomes a nightmare. Standard debuggers are useless because systems programs are written at a lower level than the debuggers themselves. The only way to tell if the program runs is to try it and see what happens.

If it doesn't work, you probably won't be able to examine memory to get an idea of what went wrong. When a systems program goes awry the computer often hangs or resets itself. The best technique that I know of for surviving systems programming is to proceed with caution and make small, well-thought-out modifications to known good software. You have an advantage now because the sample code provided with this article can be your first level of "known good" software.

REAL VERSUS PROTECTED MODE

The primary difference between protected mode and real mode programming is in the use of the segment registers. In both modes, linear addresses are calculated by adding an offset to a base which is generated by the segment registers. In real and virtual-8086 modes, the CPU generates the base address by multiplying the segment register value by 16. In protected mode, on the other hand, segment registers become selectors that refer to a descriptor table entry which contains the base address corresponding to each selector. In protected mode, all memory access is done via these descriptor table entries. The descriptors also contain other information about the segment including its type (executable, read only, or read/write) and length.

There are three system descriptor tables:

- GDT-the global descriptor table
- LDT-the local descriptor table
- IDT-the interrupt descriptor table

The GDT contains a list of descriptors which are shared by all tasks. Whether these selectors can be used by a task is another matter entirely, determined by the privilege level of the task. There are four privilege levels ranging from 0 (highest) to 3 (lowest). Attempted accesses to higher privilege system resources generate GP or General Protection faults. Virtual-8086 mode is fixed at the bottom of the totem pole with privilege level 3.

The LDT is optional and will normally exist for each task only if it is desirable to completely isolate one task from another. Since our example V86 monitor will currently be executing only one task at a time, we won't use a separate LDT.

THE INTERRUPT TABLE

Faults generate interrupts which are defined in the IDT. This table is also used for hardware interrupts and for the INT *n* software instructions. There are 32 possible faults or exceptions, only some of which are defined on the 386. These correspond to interrupts 0 through 31. Faults and exceptions are not new with protected mode. The original 8086 contained a few faults (pun intended) and a warning that "interrupts 0 through 31 were reserved for future processors." Interrupt 0, for example, is the divide fault.

Since the warning was not heeded by the designers of the PC, several of the interrupt faults new with the 286 and 386 conflict with the usage on the PC. For example, both the GP fault and the LPT2's IRQ5 generate INT 0Dh. The numeric coprocessor fault #16 conflicts with the INT 10h video ROM BIOS call. In addition, DOS requires that the interrupts be serviced by the routines listed in the interrupt vector table located at the start of memory.

In our V86 monitor example, the solution to these problems has been handled as follows. First of all, the 8259s are reprogrammed so that the hardware interrupts occur at interrupts 20h through 27h for the first chip and 28h through 2Fh for the second chip. Set this way, the hardware interrupts won't conflict with the exceptions and faults. Each of the descriptors in the IDT is then set up with privilege level 0.

This means that if a program running in virtual-8086 mode attempts to call the video BIOS via INT 10h, a GP exception will occur because the privilege level of V86 mode is fixed at 3. At that point, the GP exception handler will emulate the software interrupt and pass control to the routine listed in the interrupt vector table. Hardware interrupts will emulate the old DOS vectors in the same manner.

OTHER DESCRIPTOR TYPES

A number of different types of descriptors can be contained in the descriptor tables. The two basic types are segment descriptors which indicate the base, length, and type of a memory area, and a gate descriptor which is an indirect jump to another location. The GDT table will be made up of the necessary segment descriptors, whereas the IDT will contain gate descriptors pointing to the addresses of the fault, exception, and hardware interrupt routines.

The formats of these two descriptors are different. The gate descriptors are easily created by the MASM **DW** directive. The segment descriptors are convoluted and are most easily created using a software routine that adjusts the data into its proper format. This is done by the **adjust** subroutine. This subroutine adjusts all segment descriptors except for gate descriptors, which are already in the proper format.

TO THE PROTECTED MODE

Once the GDT and IDT are adjusted, the move to protected mode is made using the INT 15h service AH=89h routine (switch processor to protected mode). This routine also reprograms the 8259-interrupt controllers to use 20h and 28h as the base addresses for hardware interrupts.

Once you are in protected mode, a far jump should be made to clean out the prefetch queue which was read while the processor was still in real mode. In addition, all the segment registers must be reset to protected mode descriptors

or to 0. In the example code, these steps are not explicitly necessary because they are taken care of inside the INT 15h ROM BIOS routine.

Each task has a task state segment (TSS) which defines the parameters for the task. This is used when multitasking so that tasks can call one another, and all the information pertaining to that task can be saved somewhere. The definition of the TSS is a descriptor itself, and it is defined in the GDT and loaded into the task register once we are in protected mode. The TSS of a task contains, among other things, the descriptor for the LDT of the task. This is loaded to become the current LDT when the task is made active. The TSS must therefore be defined in the GDT, for if it were defined in the LDT it would end up redefining itself when the LDT is loaded during task activation.

After the move to the protected mode is complete, we go about initializing the segment registers and the protected mode data structures, such as our TSS. We also have to reinitialize all the segment registers whenever we go from V86/real mode to protected mode or vice-versa. This is because segment values (selectors) defined in one mode turn out to be meaningless in the other mode. For example, the segment of color video RAM in V86/real mode is B800h; in the protected mode the selector is 40h.

Setting up the TSS is straightforward. We don't need to fill every element of the TSS because some of these are automatically set by the CPU during task transitions. Others, such as the LDT page register and the stack to use while in the protected mode, are static and must be defined in advance.

THROUGH THE GATE

Now comes the tricky part. We pretend as if we entered protected mode via an `int_gate` called from a V86 program instead of by INT 15h. We do this by pushing the necessary V86 register values on the stack in the correct order. We push old style ($\text{base} * 16$) segment values on the stack. The CS and IP are defined to point at the exit to DOS routine. The flag register on the stack has Virtual Mode (bit 17) set so that after the return the processor will execute in V86 mode. The flag register on the stack must also be set to say that IOPL is level 3 for this task or GP exception will be generated on the **popf**, **iret**, **cli**, **sti** and **pushf** instructions.

This feature allows the processor the ability to prevent one task from clearing interrupts and dying, taking all of the other active tasks with it. If the Nested task (bit 14) is set in the current flags, the processor will attempt to return to a back-linked task instead of going to the stack for the **iretd**. We clear the NT bit and the **iretd** will pop all the values off the stack and change to V86 mode.

In V86 mode, the program calls DOS to terminate and stay resident. The software will run in V86 mode until a hardware or software interrupt or exception occur. Either of these will cause an `int_gate` transition through the IDT to the interrupt service routine. An `int_gate` transition also clears interrupts so it won't be necessary to worry about hardware interrupts occurring during protected mode processing. There are three important interrupt processing routines. The first is **intvm**, which handles the hardware interrupts; next is **fault13**, which handles the general protection exception caused by the software interrupts; and the third is **fail**, which handles bugs or shortcomings in the V86 monitor itself.

HANDLING INTERRUPTS

The hardware and software interrupt handlers are almost identical. They both determine which interrupt should be issued to the V86 program, and pass this interrupt number to the **emulateint** routine. This routine saves the V86 code segment and instruction pointer along with the flags on the V86 stack. It also sets the V86 code segment and instruction pointer equal to the address from the V86 interrupt vector table at the start of system memory. This process parallels the actual interrupting mechanism of real mode.

The GP exception handler has a bit of translation to do before it emulates the software interrupt. It first must access the user memory using protected mode descriptors to determine which INT n instruction was attempted. This is done by having a descriptor defined which is capable of accessing the entire four-gigabyte address space with 32-bit offsets. The V86 code segment and instruction pointer, pushed on the stack by the `int_gate` transition, are combined using the $\text{base} * 16 + \text{off-set}$ method to generate a 32-bit offset. This offset points at the faulting instruction. This instruction is verified to make sure that it is an INT n instruction. If it is, the vector number is retrieved from memory and the interrupt is emulated.

The hardware interrupt handlers are simpler. They undo the translation via **intvm** which is now being done by the 8259 and refer the hardware interrupt back to the standard DOS vector number. Other faults and exceptions also use **intvm** to send the interrupt to the V86 interrupt handler. Interrupts which cannot be handled in this way are passed to the **fail** routine. The **fail** routine places an error message and stack dump on the screen and terminates the program.

IN CONCLUSION

That completes the explanation of the software. I've left a few details out which cloud the general picture, yet which are easily understood through examination of the source code. The code is defined to work with a color monitor; a small modification of the GDT can be made to convert it to monochrome. The code doesn't detect whether you have an 80386 chip. If you want to add this, see "386 Now" by Steve Baker in *PJ 7.2* (March/April 1989).

I hope this will get you going in 386 protected mode. By now you probably realize that an 80386 programmer's reference manual would be a good investment. When you modify the software, leave a trail of successive backups and be meticulous in the modifications, or you just might get lost in the protected mode!

Program Source available by request