Taming the x86 beast

Jim Turley - March 18, 2004

Athough the x86 architecture is a quarter century old, the introduction of protected mode changes how you program newer members of the family.

If you've never had the pleasure of programming an x86-family processor in assembly language, you don't know pain. The 32-bit x86 chips (including the '386, '486, and Pentium-class parts from Intel and AMD) do an amazing job of dragging one of the world's oldest processor designs into the modern age. They do it while still maintaining binary compatibility with chips that are three generations and 15 years behind. It's impressive, really, but it makes these chips tricky to program with low-level code.

Although they can be binary compatible with the '286 and earlier chips, the '386 and later processors achieve this feat by shutting off their best features. It's better to run these chips in their souped-up "protected" mode. Protected mode enhances and extends the familiar x86 programming model to make these into real 32-bit processors without the nasty limitations of their predecessors. A protected-mode '386 or '486 crosses the Neolithic architecture of the 8086 with modern programming and features. It's an uneasy admixture and one that takes some practice to master.

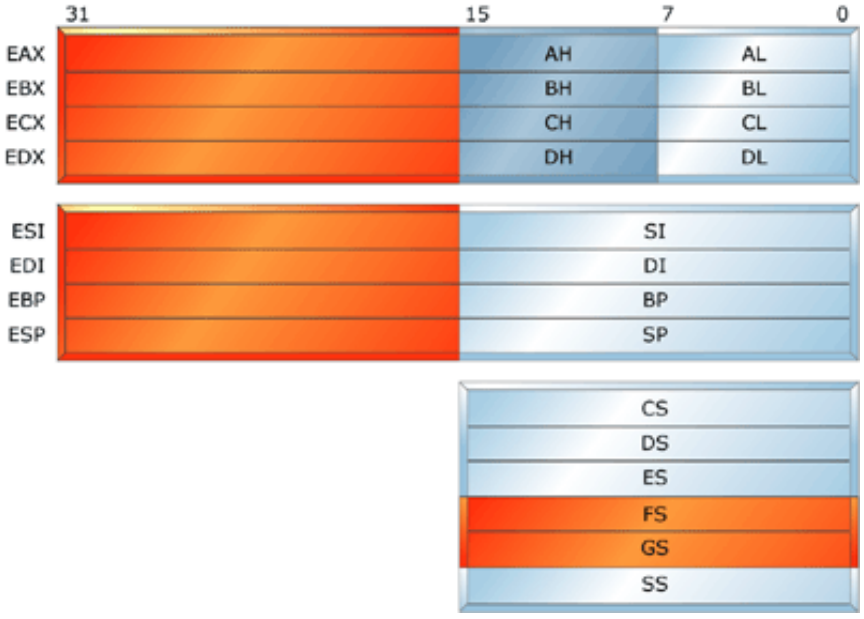

Figure 1: Protected mode extends most 16-bit registers to 32 bits

In the beginning

As Figure 1 shows, the '386 and later processors keep the standard x86 register set but extend most of the registers out to 32 bits. You can still address the low byte of the accumulator as AL and the upper byte as AH (and the pair as AX), but now you can also use a 32-bit version called EAX. The same rules apply to BX, CX, and DX. Likewise, the stack pointer and the other pointer registers have been elongated but still keep their shorter nicknames.

Memory segment registers, the bane of all x86 programs and programmers, are still with us, alas. Not only haven't these registers disappeared, there are two more of 'em. The new FS and GS segment registers augment the usual set of four. Although the segment registers look the same as before, they work completely differently, as we'll see.

Segment registers are one of the oddest remnants of the x86 family's long history. In the 1980s, these registers allowed chips with 16-bit registers to form 20-bit addresses—enough to reach a whopping (for the time) 1MB of memory. The only peculiarity was that programmers had to assemble addresses in two parts, a "segment" part and an "offset" part. The chip automatically

combined these two parts every time it accessed memory. Since x86 chips had only four segment registers (two of which were dedicated to code and stack), you had to be careful how you juggled your address pointers. Usually, it was just easier to keep everything within a 64KB (16-bit) boundary and ignore the segment registers as much as possible.

How to extend this already-overextended model to 32 bits? Intel could have simply lengthened the segment registers to 24 bits or so, or added a gaggle of segment registers. But the company took a characteristically bizarre approach. Rather than extend the segment registers, it simply redefined how they work. Grab your seats.

Everything you know is wrong

On the old 8086, any memory address was there for the asking. It didn't matter if "real" memory was out there; the programmer was expected to avoid producing addresses that didn't correspond to physical memory or I/O. Now a program, no matter how privileged, cannot access an area of memory unless that area has been previously "described" to it.

In protected mode, segment registers are now pointers into a table. Their value has no bearing on memory addresses at all; they're simply an index. The real memory address is formed in a far more convoluted way. Previously, segment registers could hold any value at all, from 0000 through FFFF. That value formed the upper 16 bits of the eventual 20-bit address. Now, segment registers must be multiples of 8, so values of 0008, 0010, and 0018 are legal but 0001, 0002, and DCF3 (for example) are not. Why multiples of eight? Because each entry in the Global Descriptor Table is eight bytes long.

What's that you say? Never heard of the Global Descriptor Table (GDT)? You will if you'll be doing any protected-mode programming. In fact, you'll become intimate with the GDT and its arcane details, described herein with relish and enthusiasm.

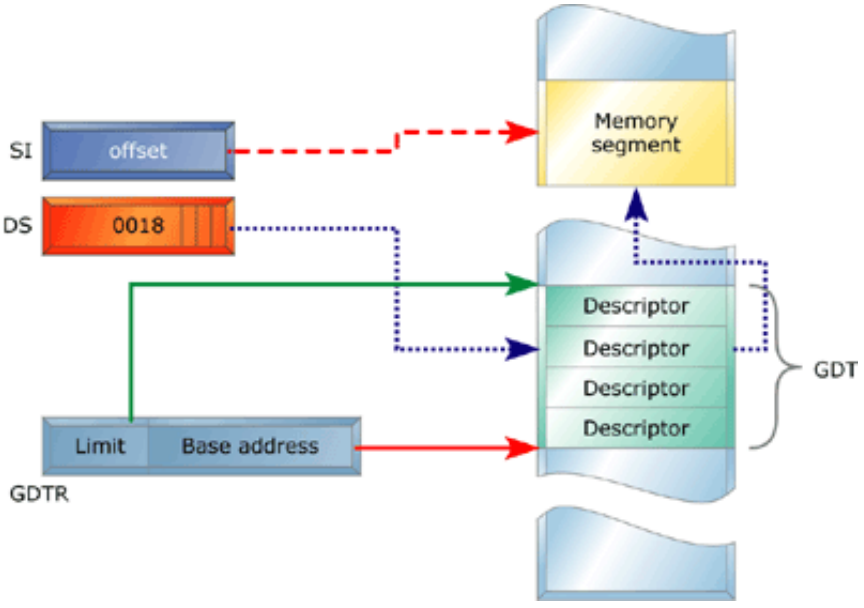

Figure 2: Segment registers and offsets index into the descriptor table

As Figure 2 shows, segment registers now act as pointers into the GDT. The GDT is a table that you create. Each eight-byte entry in the GDT describes the boundaries and characteristics of one segment of memory. That's right—you now define your own memory segments. You must define at least three GDT entries, but you can create thousands if you like. The GDT can be located anywhere in memory (we'll return to this paradox in a moment) and its base address is defined by a new x86 register, the GDTR.

Because segment registers must now be multiples of eight, their low three bits are more or less ignored. (These bits are used later for task-level privilege control, a subject for another day.) This makes the remaining 13 bits a byte offset into the GDT that identifies one of your user-defined GDT entries, called a segment descriptor. A segment descriptor tells the world (and the processor itself) everything it needs to know about one segment of memory: where it starts; how long it is; whether it holds code, data, or stack; and various other obscure details.

Gone are the days of fixed 64KB memory segments. Now a segment can be one byte long, 4GB in size, or anything in between. You can fine-tune a segment of memory to start at any arbitrary address in the entire 32-bit address space, including odd addresses or addresses that aren't a multiple of 16. It's all up to you. Want a 2,816-byte stack segment that can't overflow? You've got it. Want a code segment that's just a wee bit over the usual 64KB limit? No problem. Always wanted a data segment that can't be accidentally overwritten by code? It's yours.

With privilege comes responsibility. You must define each and every detail of each memory segment, a bit of a chore and fraught with tribulations. Ready to define some memory segments? Splendid. Let's begin.

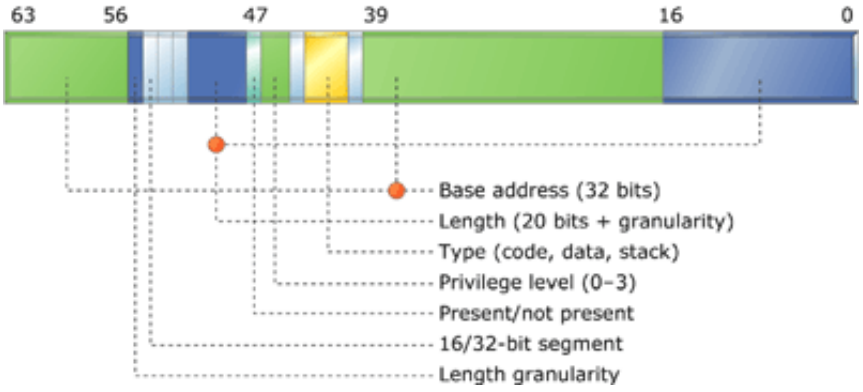

Figure 3: Every segment of memory is described by an eight-byte descriptor

Know your limits

As Figure 3 shows, each eight-byte memory descriptor is divided up into several fields. Foremost among these is the 32-bit base address field. Naturally enough, this is where you define exactly where this segment of memory begins. Notice that with one-byte granularity, you can start the segment absolutely anywhere you like. No more chunkiness in your memory map!

Next in importance comes the segment length field. Ideally, this would also be 32 bits long but with half the descriptor already used up, Intel made some necessary compromises. You have 20 bits in which to define the length of the memory segment, which gives you one-byte granularity as long as your segment is smaller than 1MB. To define a segment larger than that, you'll have to make do with 4KB granularity. Setting the granularity bit (bit 55) in a segment descriptor tells the processor to interpret the other 20 bits in units of 4KB. Thus, you can define a single segment that spans the processor's entire 4GB address range.

The segment type field (bits 41 to 43) tells the processor whether this segment will hold code, data, or a stack. If it's a code segment, you get to decide whether the code is execute-only or executable with read permission (in other words, if you've got constants sprinkled in your code). For data and stack segments, you can declare them read/write accessible or read-only. A read-only stack doesn't make much sense but a read-only data segment can prevent accidental overwriting of variables, effectively creating "soft ROM" space in your memory. Several other control bits are defined in each descriptor, but we'll ignore them for now.

Notice that a segment can be code, data, or stack, but only one of the three. They're mutually exclusive. The '386 and later processors won't let you execute code from a data segment or store variables in a code segment. If you want to store data with your code, or place your stack with your data, you have to create two different segment descriptors for the same space. In other words, you alias the same address range through two different segment descriptors.

More levels of indirection

Now that we've laid the groundwork let's see how the system works in practice. We'll load register CS with 0008, pointing to the second descriptor (offset 0008) in the GDT. That descriptor, in turn, says our code segment starts at 32-bit address 00420C78. For long-time x86 programmers, it's strange to load CS with one address and yet fetch code from an entirely different address. Remember, segment registers don't hold addresses anymore, just indirect pointers to addresses.

Let's say the length of this code segment is defined as 5,150 bytes, so all branches, jumps, and offsets must be smaller than this limit. It's okay to jump to a different code segment, but not to offsets beyond the end of this code segment.

Now we'll load register DS with 0010, pointing to the third (offset 0010) descriptor in the GDT. (The GDT itself might be located at any arbitrary 32-bit address but we'll assume it's at 0000 for simplicity.) That descriptor says there's a data segment beginning at address 0C500 that is a mere 24 bytes long. Our stack segment register, SS, points to descriptor 0018, and so on. It's perfectly okay to leave some segment registers (for example, FS and GS) empty or undefined; they don't have to be loaded with valid GDT pointers. You won't be able to use those segments until they do hold valid pointers, though. Unlike earlier x86 processors, loading just any old value won't do. Unless you've gone to the trouble of defining 8,192 different segments in your GDT, most 16-bit segment values are illegal.

Any number of weird and unexpected things happen with this new segmentation system. For one, you have to be much more careful about what you put into your segment registers. Most values are illegal and even many legal values won't work. For example, CS must point to a segment descriptor that defines a code segment; pointing CS to a data or stack segment causes the processor to take a fault. Conversely, the stack segment SS cannot point to a code segment—that also causes a fault.

Incrementing a segment register doesn't work anymore. It used to be common practice to "walk" through memory by bumping the value in the segment register every 64KB. That trick doesn't work, and besides, it shouldn't be necessary. Since segments can be any size you want them to be, the idea is to make them the right size in the first place, not butt them against each other. On the other hand, you can overlap segments all you want. The end of one data segment can overlap the beginning of another code segment—it's possible, though not always a wise choice.

You also can't "wrap around" the end of segment like before. Earlier x86 chips with their 64KB segments allowed pointers to roll over from FFFF to 0000 automatically. Programs that accidentally (or purposely) ran off the end of a segment started over at the beginning. Programs that run off the end of a segment now are greeted with a segmentation fault.

Strangely, there is no way to access a known physical address. Previously, you could divine the location of an instruction or data value by examining its segment and offset registers. Now, those values tell you nothing about where something is actually located in memory. Even the compiler doesn't know the relationship between a segment value and an actual address. Segment addresses are resolved at run time by the chip itself, not by software. To figure out an actual physical address, you'd have to deconstruct the GDT at run time, teasing the values out of the appropriate segment descriptors.

This leads us to a subtler problem—where do you look? The GDT is just a garden-variety table in memory, but you'd have to know where in memory to find it. The GDTR holds its base address, so that's a start (assuming you can access the GDTR, which is a privileged operation). But what data segment do you use to access the table itself? Even if you know the GDT's base address, you don't know what data segment—if any—covers that area of memory. It's entirely possible that no data segment covers the area where the GDT resides. Unless you (or another programmer) created such a data segment, there's no way to read the GDT as data; it's off limits. And even if there were an appropriate data segment, how would you know what its segment index was? In other words, what value should you load into the DS register to access the right data segment?

This highlights still another peculiarity of the x86 in protected mode. Segment registers are now effectively keys that unlock various area of memory. With the right key, you can execute code from a code segment or read (and possibly write) data to/from a data segment. But without the key, you have no idea what code or data segments might be available. You can't just experiment with different segment values, either, since almost all undefined values will cause a processor fault.

The situation is not entirely hopeless. Some new protected-mode instructions tell you almost everything you want to know about segment registers and descriptors. The new LSL (load segment limits) instruction tells you the length of the segment (if there is one) for any segment register value you care to test. The VERR (verify for read) instruction tells you if it's okay to read from that segment (in other words, that it's not an execute-only code segment), while VERW (verify for write) reports whether the segment is writable. The LAR (load access rights) instruction reports on the minutiae of the segment's privileges and other details. There's no instruction to determine the base address of a segment, which makes physical addressing impossible unless you have inside knowledge of the GDT.

Oh brave new world

No combination of segment-descriptor attributes will give you a "traditional" 8086-style memory segment that's simultaneously readable, writable, and executable. However, you can create much the same effect through other means, such as defining three separate segments that happen to share the same base address and length. You still can't load the same segment value into CS, DS, and SS—they'd have to be three separate indexes—but once loaded, all three segment registers would point to the same physical address range.

If you're sick of x86-style memory segmentation (or never developed a taste for it in the first place), you can create a "flat" memory model. Simply define three segments (code, data, and stack) with a base address of 0000 and a length of 4GB. Now everything is accessible and offsets will correspond to actual addresses.

Typically, segment descriptors are all created at system-initialization time to define all areas of memory you expect to use and their expected uses (code, data, stack). You might define a single executable portion, one or two data segments, and a stack segment big enough to hold a worst-case stack frame. By and large, these segments would occupy distinct areas of physical memory. You could, if you wished, assign them to contiguous addresses, end to end, so that the end of one segment coincided with the beginning of another.

Although you can't write into a segment that's been defined as executable, there's nothing wrong with writing into a segment that's defined as writable and just happens to cover the same address range as a code segment. If the code-segment descriptor and the data-segment descriptor have the same base-address and limit fields, you can write into your code space and execute your data space.

Protected-mode programs can still make inter-segment (far) jumps and subroutine calls. Just like before, a FAR CALL will change both the CS segment register and the IP instruction pointer. Only now, CS gets a new index into the GDT, which should point to a new segment of code space. This new segmentation mechanism makes a nice neat way to wrap functions, tasks, or applications inside their own private code segments. It goes a long way toward preventing programs from contaminating one another. Each FAR CALL also invokes a hardware privilege check to ensure that the calling function has permission to access the called function—an entirely new and completely automatic feature of protected-mode x86 chips that we'll cover in a future issue.

Jim Turley is the editor in chief of *Embedded Systems Programming*, a semiconductor industry veteran, and the author of seven books, including the *Essential Guide to Semiconductors*. You can reach him at jturley@cmp.com.