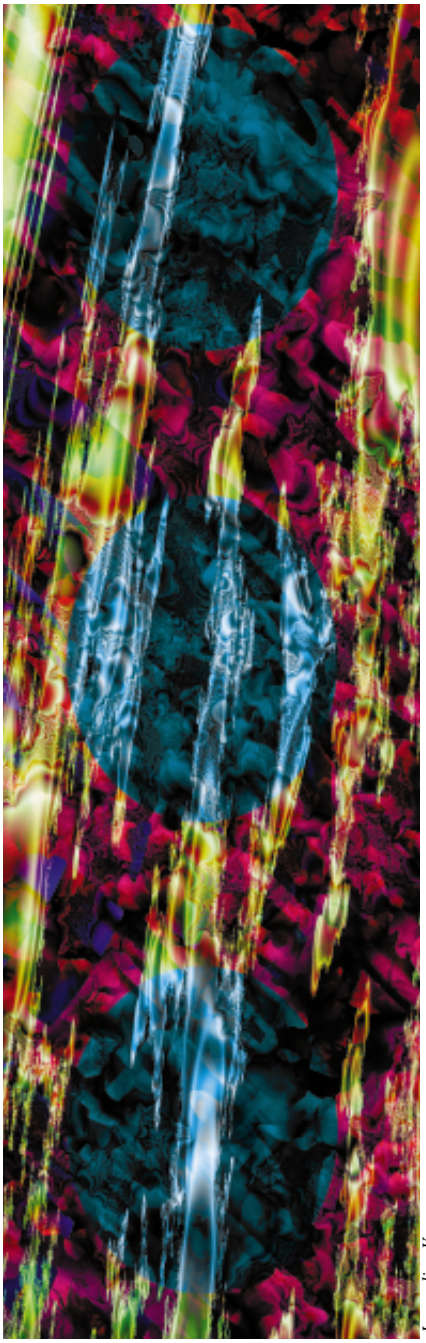
by JEAN GAREAU

<BACK

Embedded x86 Programming: Protected Mode



Jacqueline Vansen

The x86 architecture is ubiquitous on the desktop and is spilling over into embedded systems environments. This article begins a series designed to help you on your way toward implementing an embedded x86-based system.

Intel has shipped millions of 80386, 80486, and Pentiums since 1986, and this figure is increasing rapidly. The x86 is expected to seriously affect the embedded systems market for the following reasons: applications can be developed on a PC (not necessarily on a target), both 16-bit and 32-bit programming are fully supported, a complete diversity of hardware is available, and GUI features—through Windows CE and 95—will become more accessible.[1]

Consequently, many existing RTOSs will likely be ported to this CPU—if not completely rewritten from scratch—to exploit the x86's capabilities. This CPU and its successors are armed with a battery of features that enables the implementation of the most advanced concepts in operating system design. These features also allow the writing of simpler embedded applications by providing 32-bit operations and various memory models that give applications large address space. Development tools (compilers and linkers) also have some benefits, because popular memory models, such as the flat memory version, are simpler to support.

This article initiates a series presenting in-depth technical coverage of the most important features: protected mode (the subject of this article), segmentation, and paging. Functional examples are provided with each article to illustrate the concepts. To understand segmentation and paging concepts and how they can simplify embedded application development, the protected mode must first be explained in detail.

Complete examples that implement various kernel designs—fully documented and tested—can be downloaded from the *ESP* Web site at *www.embedded.com/code*. These examples demonstrate the concepts I'll explain in this series, including a port of μC/OS to protected mode.[2] The source code is provided, as well as ready-to-run executables and additional tools. These various implementations will provide a start to help you improve your applications or even implement your own system.

This CPU is armed with a battery of features that enables implementation of the most advanced concepts in OS design.

REVIEWING THE REAL MODE

Protected mode has its roots in the 8086 processor, the ancestor of the 32-bit 80386. The 8086, although a 16-bit CPU, provides a clever mechanism to access up to 1MB of physical (real) memory: real mode. This addressing mode relies on a combination of segment and offset registers to address bytes in memory (instruction or data). Each instruction uses one of the four segment registers available, either implicitly or explicitly. Address calculation is done by shifting a segment register by four (multiplying by 16) and adding one of the nine general registers, typically the one specified in the instruction (see Figure 1). The result is a 20-bit address, providing 1MB of address space and using 16-bit registers. The carry bit (bit 20) is discarded.

Since a single segment register allows accessing 64K, multiple segments are required to access more memory. Most developers involved in Intel application development have heard of the various memory models that were popular not so long ago: tiny, small, medium, compact, large, and huge. These models proposed various segment combinations in order to overcome the 16-bit limitation when accessing code and data beyond 64K.

To push the 1MB limitation further, some complex schemes were introduced, such as the expanded and extended memory. These schemes helped, but they also increased memory management complexity and consequently, introduced overhead.

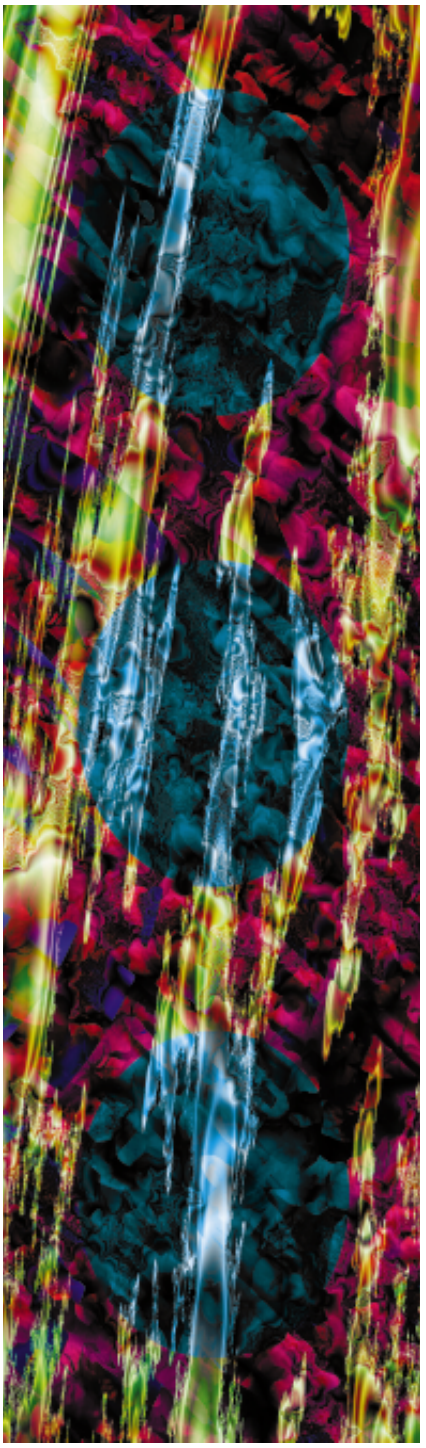
Compilers, linkers, and operating system loaders had the responsibility of assigning proper values to segment registers, to free the application developers from doing so. System programmers, writing programs mainly in assembly, were not so lucky and had to cope with this complex scheme. The source of all this complexity was the infamous 64K limitation due to the 16-bit nature of the CPU.

INTRODUCING THE PROTECTED MODE

In 1986, with the advent of the Intel 80386, things really started to change. For one, this processor is a real 32-bit processor. The main advantages of 32-bit programming over 16-bit and 8-bit programming are speed and simplicity. Instructions themselves are usually faster and a single instruction (string and memory operations, arithmetic, and so forth) can work on 32 bits at the same speed as two 16-bit instructions or four 8-bit instructions. Reducing the number of instructions reduces program sizes and speeds up execution, because fewer instructions have to be fetched and decoded. Smaller size and faster execution are always welcome in embedded systems.

Second, this CPU's memory management unit (MMU) introduces a new addressing mode over real mode called protected mode. This mode offers a high degree of flexibility, making possible very large 4GB flat address space per task or per application (no segment) and up to 64 terabytes (that's 64,000,000,000,000 bytes) of virtual memory! This mode also adds some protection in order to run the software that needs it, such as Unix-like systems have. Advanced memory concepts and protection will be covered in subsequent articles.

In protected mode, the segment registers are indexes into special tables, all



initialized and maintained by the operating system, but interpreted by the CPU. There are three types of tables, all located either in RAM or ROM:

- The Global Descriptor Table (GDT), unique, always accessible
- The Local Descriptor Table (LDT),

FIGURE 1
Real mode address calculation.

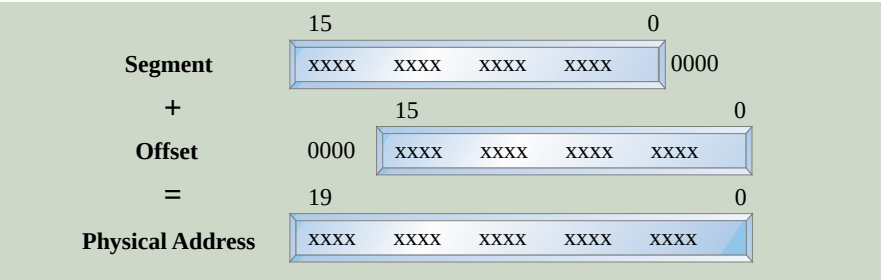


FIGURE 2
Format of an 8-byte segment descriptor.

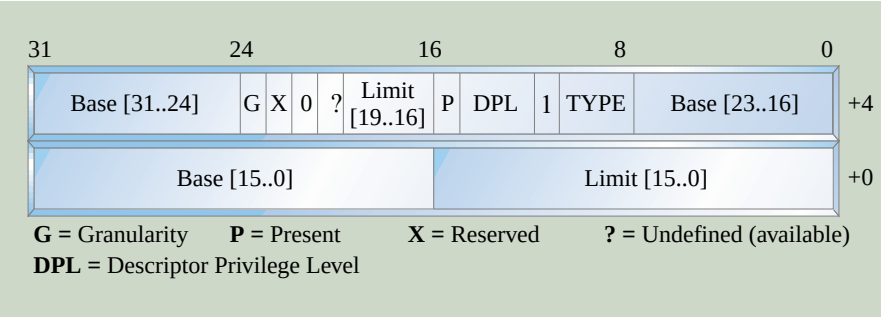


FIGURE 3
Format of a selector.

usually one per task. Zero, one, or many may be present in the system, but only one, if any, is active at all times

- The Interrupt Descriptor Table (IDT), used when interrupts are raised

Each table contains a variable number of descriptors and an 8-byte data structure that describes a memory region with the following attributes (see Figure 2):

- The base address in memory (32-bit)
- The limit (20-bit), expressed either in 4K or 1-byte units
- Control bits: the granularity bit (limit's unit), present bit (useful with swapping), and two protection bits
- The descriptor type, one of the 16 supported, among them: executable, read-only code segment; data segment; stack segment; call, trap, or interrupt gate; task state segment, and others

Segment registers are selectors (indexes) into either the GDT or the LDT (the IDT entries are only used when an interrupt is raised). A selector (such as a segment register) contains a 13-bit index, a 1-bit table identification (GDT/LDT), and a 2-bit protection level. See Figure 3.

A logical address, as used by a program, is still the combination of a segment and a general register, or more precisely, a 16-bit selector and a 16-bit or 32-bit offset. The selector identifies a descriptor, which in turn provides a 32-bit base address, to which the offset is added, forming a final linear 32-bit address as seen in Figure 4. This 32-bit addressing supports up to 4GB (2^{32}) of memory.

All memory accesses within the segment can be done with the offset only, simplifying program coding. This address calculation provides many advantages:

- Because segment registers cover up to 4GB individually, they don't have to be constantly reloaded, even with huge data structures, reducing complexity and increasing speed
- Offsets always start at zero, independently of the segment's location in physical memory, making it easier to debug—the addresses (offsets, for example) never change. Segments can be moved in physical memory without affecting the applications that use them
- An offset must be within the segment's limit; if it isn't, an exception is raised and the operating system, which typically catches it, may stop the faulty application. This feature prevents incorrect memory access, such as jumping outside the code segment or accessing out-of-segment data
- Segments are protected against undesired access, thanks to their descriptors. For instance, an application cannot write into a code segment, which is read-only. Another similar example is to prevent executing from a data segment
- This addressing mode provides a phenomenal virtual address space. Because a selector's index is a 13-bit value, the GDT and LDT tables are limited to 8,192 descriptors (2^{13}). One single descriptor can cover up to 4GB (with a base address of zero, a limit of 1MB (FFFFFh), and the granularity bit set, making the limit a 4K-unit value (4K x 1MB = 4GB)). Considering that the GDT and the active LDT together have a maximum of 16,384 descriptors, the total virtual address space is 64 terabytes

FIGURE 4
Protected mode address calculation: all memory accesses within the segment can be done with the offset only, simplifying program coding.

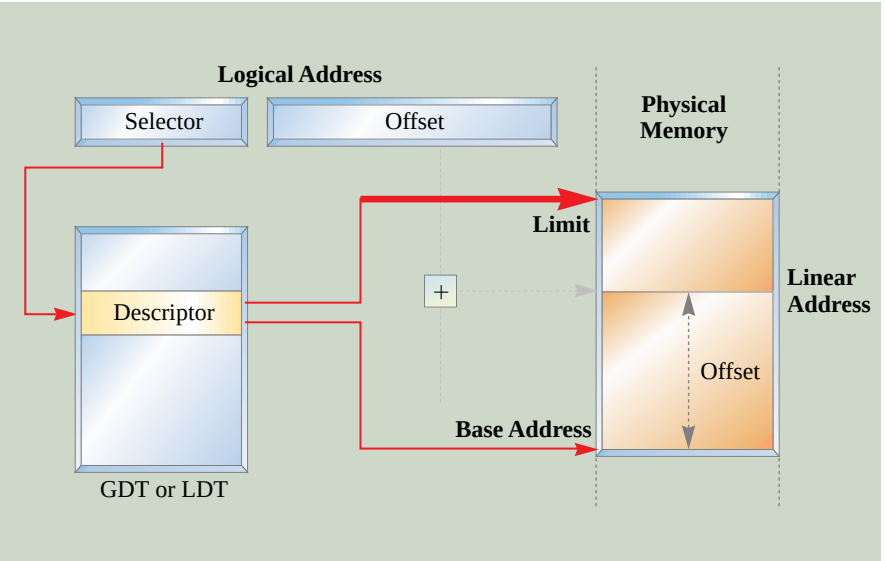
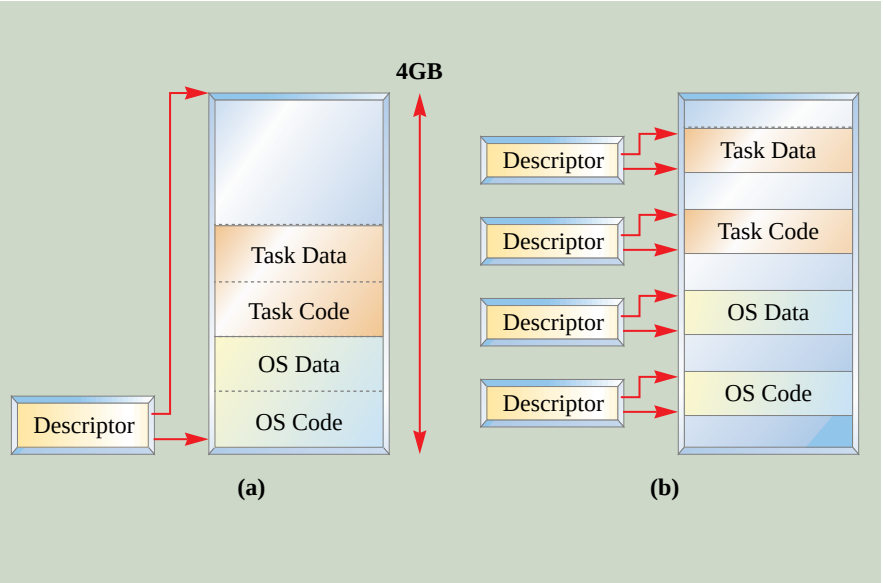


FIGURE 5
Simple (a) vs. complex segmentation (b).



LISTING 1
Sixteen-bit and 32-bit code segments. Opcodes are shown on the left.

```
1. _TEXT        SEGMENT   PARA USE16 PUBLIC 'CODE'
2. 33 C0        xor       ax,ax
3. 66 33 C0     xor       eax,eax
4. _TEXT        ENDS
5.
6. _TEXT        SEGMENT PARA USE32 PUBLIC 'CODE'
7. 33 C0        xor       eax,eax
8. 66 33 C0     xor       ax,ax
9. _TEXT        ENDS
```

(16K * 4GB). Although this is astronomical, one must realize that segmentation always produces a 32-bit linear address, limiting the physical address space to 4GB, still quite sufficient

- One embedded operating system may use a few segments, whereas another may use hundreds of them, as illustrated in Figure 5. In (a), an embedded application can reside in the operating system with a single segment covering 4GB. In (b), the same application might own its proper segments, all distinct from the operating system segments. These various approaches can be justified depending on the system constraints.

MIXING 16-BIT AND 32-BIT CODE

Protected mode is not synonymous with 32-bit. The Intel CPUs, in protected mode, support 16-bit and 32-bit segments. This makes them ideal CPUs to run legacy 16-bit applications as well as new 32-bit systems.

Most x86 assemblers support some directives to indicate whether a specific segment of code will be executed in 16-bit mode (the USE16 directive) or in 32-bit (USE32). The assembler will generate the appropriate code, but it's up to the operating system to load the task adequately—that is, to ensure the USE16 segments are run in 16-bit mode and USE32 segments are run in 32-bit mode.

By comparing disassembled 16-bit with 32-bit code, one notices many similarities, as shown in Listing 1. For instance, the 16-bit instruction `xor ax,ax` (line 2) and the 32-bit instruction `xor eax,eax` (line 7) produce identical opcodes (`33h C0h`). How does the CPU make the difference between the 16-bit AX and 32-bit EAX registers? This has to do with the current segment mode. Whenever the code segment—the CS segment register—is loaded with a selector, the CPU loads the descriptor into an internal, inaccessible register (reserved for the CPU only) and ana-

LISTING 2
Although many instructions are identical in 16-bit and 32-bit, addressing modes and addresses always generate different opcodes, making the code incompatible.

```
1. _TEXT                SEGMENT   PARA USE16 PUBLIC ´CODE´
2. 8D 06 0024           lea       ax,MyVar
3. _TEXT                ENDS
4.
5. _TEXT                SEGMENT   PARA USE32 PUBLIC ´CODE´
6. 66 8D 05 00000024    lea       ax,MyVar
7. 8D 05 00000024       lea       eax,MyVar
8. _TEXT                ENDS
```

lyzes its type. One bit in the descriptor determines whether this is a 16-bit or 32-bit segment. If this is a 16-bit segment, the opcodes `33h C0h` mean `xor ax,ax`; if this is a 32-bit segment, they mean `xor eax,eax`. In a 16-bit code segment, opcodes work with 16-bit operands and addresses; in a 32-bit code segment, opcodes work on 32-bit operands and addresses. Consequently, the CPU has only one instruction set and it works for both 16-bit and 32-bit modes.

To provide maximum flexibility, the opcodes `66h` and `67h` respectively override the operand and address size. For example, in a 16-bit code segment, `66h 33h C0h` (line 3) would work on 32-bit EAX whereas the same instruction, in a 32-bit code segment (line 8), would affect only the lower 16-bit AX.

This condition is made possible because the CPU really has one and only one bank of 32-bit general registers. It's the current CPU mode (either real mode, 16-bit protected mode, or 32-bit protected mode) that indicates whether 16-bit or 32-bit operands and addresses are used and how they are used. Typically, the size override opcodes are used in 16-bit code to access 32-bit values. They are rarely used in 32-bit application code.

Serious incompatibilities between 16-bit and 32-bit code still exist, especially regarding the address mode encoding (the 32-bit mode supports more addressing modes) and the addresses themselves (encoded over 16 or 32 bits), as shown in Listing 2. The 16-bit instruction `lea ax,MyVar` (line 2) has no direct opcode equivalent in a 32-bit segment, even with size override opcodes (lines 6 and 7).

Mixing 16-bit and 32-bit code is more than often an issue and should be avoided whenever possible. There is, however, one case in which this mix can't be avoided: upon starting up, the CPU starts running in real mode. A 32-bit system will have to switch into 32-bit protected mode and will have to mix at some point or another some 16-bit code (system initialization) and 32-bit code (rest of the system).

If both your assembler and linker accept 16-bit and 32-bit segments, you can simply put the real mode code into a USE16 segment and the 32-bit protected mode code into a USE32 segment. Once the 16-bit code is ready to go 32-bit, it simply jumps into the USE32 segment. The two segments don't have to be part of the same program, although having two programs doesn't always make things simpler. I prefer to keep the initialization in one single source file.

Again, this solution only works if your tools fully support a 16-bit and 32-bit code mix. Unfortunately, some recent tools only support flat memory model and do not fully support 16-bit code. For instance, some linkers will not permit a jump from a 16-bit segment into a 32-bit one. However, switching into 32-bit protected mode requires exactly that. Thus, the issue becomes how to write 16-bit instructions with tools that only support 32-bit programming.

The solution is to write the 16-bit code in 32-bit segments. Now this is not trivial, because mnemonics will be assembled as 32-bit instructions, and they will cause problems when they're executed in real mode. Using size override opcodes may not entirely resolve the issue because some instructions are simply incompatible, as shown in Listing 2. Simply put, 16-bit instructions cannot be written as such in a 32-bit segment. But there's still a solution.

By disassembling the real-mode instructions from a 16-bit, you can directly put the resulting opcodes in a 32-bit segment, using assembly directives such as DB, DW (which allow you to enter values in numerical form). This is really directly encoding 16-bit instructions. I agree that this isn't a high-tech solution, although the use of macros can maintain the code's readability. Also, the 16-bit portion typically consists of a few instructions and it affects little code overall. Furthermore, these few instructions are unlikely to change. This method is demonstrated a little bit further.

There are other alternatives. If you are designing a 16-bit system, you are free to stay in 16-bit as long as you want, by writing everything in a USE16 segment and either far jumping into a 16-bit segment or not jumping at all (since the CPU maintains a valid code segment after the protected mode is activated, as we will see later). This method obviously requires tools that support 16-bit programming.

ACTIVATING THE PROTECTED MODE

Now let's take a look at an example that shows how to activate protected mode. This example starts in real mode and switches into protected mode in order to execute 32-bit code in a flat memory model. See Listing 3. The purpose of this example is to demonstrate basic protected mode concepts; if you're looking for ready-to-run examples, be sure to take a look at those at *www.embedded.com/code*.

This example assumes that after the

LISTING 3
Switching from 16-bit real mode to 32-bit protected mode.

```
1. ; ProtMode.asm
2. ; Copyright (C) 1998, Jean L. Gareau
3. ;
4. ; This program demonstrates how to switch from 16-bit real mode into
5. ; 32-bit protected mode. Some real mode instructions are implemented with macros
6. ; in order for them to use 32-bit operands.
7. ;
8. ; This program has been assembled with MASM 6.11:
9. ;    C:\>ML ProtMode32.asm
10.
11.              .386P                        ; Use 386+ privileged instructions
12.
13. ;-----------------------------------------------------------------------------;
14. ; Macros (to use 32-bit instructions while in real mode)                     ;
15. ;-----------------------------------------------------------------------------;
16.
17. LGDT32       MACRO    Addr                ; 32-bit LGDT Macro in 16-bit
18.              DB       66h                 ; 32-bit operand override
19.              DB       8Dh                 ; lea (e)bx,Addr
20.              DB       1Eh
21.              DD       Addr
22.              DB       0Fh                 ; lgdt fword ptr [bx]
23.              DB       01h
24.              DB       17h
25. ENDM
26.
27. FJMP32       MACRO    Selector,Offset     ; 32-bit Far Jump Macro in 16-bit
28.              DB       66h                 ; 32-bit operand override
29.              DB       0EAh                ; far jump
30.              DD       Offset              ; 32-bit offset
31.              DW       Selector            ; 16-bit selector
32. ENDM
33.
34.              PUBLIC   _EntryPoint         ; The linker needs it.
35.
36. _TEXT        SEGMENT PARA USE32 PUBLIC ´CODE´
37.              ASSUME   CS:_TEXT
38.
39.              ORG      5000h               ; => Depends on code location. <=
40.
41. ;-----------------------------------------------------------------------------;
42. ; Entry Point. The CPU is executing in 16-bit real mode.                     ;
43. ;-----------------------------------------------------------------------------;
44.
45. _EntryPoint:
46.
47.              LGDT32   fword ptr GdtDesc    ; Load GDT descriptor
48.
49.              mov      eax,cr0              ; Get control register 0
50..             or       ax,1                 ; Set PE bit (bit #0) in (e)ax
51.              mov      cr0,eax              ; Activate protected mode!
52.              jmp      $+2                  ; To flush the instruction queue.
```

CPU has been reset, some system tests have successfully run, the interrupts have been disabled, and the CPU is still running in real mode. The BIOS, if any, is ignored because it works only in real mode and is irrelevant in protected mode. The code can be run anywhere in the first 1MB of memory.

I used Microsoft's Macro Assembler (MASM) v. 6.11, the latest revision. Also, instead of using the provided linker, I used Microsoft Visual C++ v. 5.0's linker, which is the latest incremental linker from Microsoft. This linker generates COFF file executable, which is the standard under Windows NT, (MASM's linker outputs less popular OMF files). The incremental linker has one drawback: it doesn't fully support 16-bit segments. Consequently, 16-bit instructions must be directly encoded, as I explained earlier.

The shortest way to execute 32-bit code is to load the GDT, activate the protected mode, and jump into a 32-bit segment. This order is not strict, though. For instance, you may first activate the protected mode, load the GDT, and make the jump. Either way is the same.

The GDT in Listing 3 is already constructed (lines 97 to 121). Even if you intend to dynamically add descriptors in it later, you can initially have it with a few static descriptors right from the start. In the example, the GDT occupies 24 bytes and contains three entries:

- Entry 0 is null. This entry cannot be referred to; segment registers can be initialized to zero (thus pointing to it), but using them raises an exception. This feature is aimed at identifying NULL far pointer references. Nevertheless, it could contain some data because the descriptor is never used by the CPU. In the example, it simply contains zero
- Entry 1 (Selector 08h) is used for kernel code, with a base of zero, a limit of FFFFFh, with the granularity bit set (making the limit 4GB),

LISTING 3 (cont.)
Switching from 16-bit real mode to 32-bit protected mode.

```
53.
54. ; The CPU is now executing in 16-bit protected mode. Make a far jump in order to
55. ; load CS with a selector to a 32-bit executable code descriptor.
56.
57.             FJMP32    08h,Start32         ; Jump to Start32 (below)
58.
59. ; This point is never reached. Data follow.
60.
61. GdtDesc:                                  ; GDT descriptor
62.             dw        GDT_SIZE - 1        ; GDT limit
63.             dd        Gdt                 ; GDT base address (below)
64.
65. Start32:
66.
67. ;-----------------------------------------------------------------------------;
68. ; The CPU is now executing in 32-bit protected mode.                         ;
69. ;-----------------------------------------------------------------------------;
70.
71. ; Initialize all segment registers to 10h (entry #2 in the GDT)
72.
73.             mov       ax,10h              ; entry #2 in GDT
74.             mov       ds,ax               ; ds = 10h
75.             mov       es,ax               ; es = 10h
76.             mov       fs,ax               ; fs = 10h
77.             mov       gs,ax               ; gs = 10h
78.             mov       ss,ax               ; ss = 10h
79.
80. ; Set the top of stack to allow stack operations.
81.
82.             mov       esp,8000h           ; arbitrary top of stack
83
84. ; Other initialization instructions come here.
85. ;           ...
86.
87. ; This point is never reached. Data follow.
88.
89. ;-----------------------------------------------------------------------------;
90. ; GDT                                                                         ;
91. ;-----------------------------------------------------------------------------;
92.
93. ; Global Descriptor Table (GDT) (faster accessed if aligned on 4).
94.
95.             ALIGN     4
96.
97. Gdt:
98.
99. ; GDT[0]: Null entry, never used.
100.
101.            dd        0
102.            dd        0
103.
104. ; GDT[1]:    Executable, read-only code, base address of 0, limit of FFFFFh,
```

and the type set to executable, read-only code

- Entry 2 (Selector 10h) is used for kernel data, also with a base of zero, a limit of FFFFFh, granularity bit set but with a type of writable segment. This data segment overlaps the code segment. Together, they provide a flat memory address space

The GDT is loaded by initializing the GDTR register with the GDT base address and size, both stored in a 6-byte data structure (line 61). The GDTR register is normally loaded by executing the 16-bit instruction `lgdt fword ptr address`. But this instruction cannot be written as such because the resulting 32-bit opcodes will not work in real mode (the address mode will be incorrect). To make things worse, addresses must be expressed as 32-bit values (a linker constraint), whereas 16-bit values are normally expected in real mode. To overcome this problem I used the 16-bit instructions `mov ebx,address` and `lgdt fword ptr [bx]`, encoded in the LGDT32 macro (lines 17 to 25), and called from line 47. The address is specified as a 32-bit value, although only the lowest 16-bit portion is used. But above all, it properly loads the GDT register while the CPU is running in real mode.

With the GDT register set, protected mode is activated by setting bit #0 in the CR0 register (lines 49 to 51). CR0 is a control register that controls segmentation and paging, among other things. CR0 can only be read from or written to by using register operands (no memory nor immediate operands). The example uses the AX register, although the opcodes will use EAX when executed in real mode. As soon as the bit is set in CR0, protected mode kicks in and the CPU starts executing 16-bit instructions, but in protected mode (segment registers become indexes into a table).

The content of all segment registers is unknown at this point. However, it is

LISTING 3 (cont.)
Switching from 16-bit real mode to 32-bit protected mode.

```
105. ;  granularity bit (G) set (making the limit 4GB)
106.
107.            dw      0FFFFh          ; Limit[15..0]
108.            dw      0000h           ; Base[15..0]
109.            db      00h             ; Base[23..16]
110.            db      10011010b       ; P(1) DPL(00) S(1) 1 C(0) R(1) A(0)
111.            db      11001111b       ; G(1) D(1) 0 0 Limit[19..16]
112.            db      00h             ; Base[31..24]
113.
114. ; GDT[2]: Writable data segment, covering the save address space than GDT[1].
115.
116.            dw      0FFFFh          ; Limit[15..0]
117.            dw      0000h           ; Base[15..0]
118.            db      00h             ; Base[23..16]
119.            db      10010010b       ; P(1) DPL(00) S(1) 0 E(0) W(1) A(0)
120.            db      11001111b       ; G(1) B(1) 0 0 Limit[19..16]
121.            db      00h             ; Base[31..24]
122.
123. GDT_SIZE   EQU     $ - offset Gdt  ; Size, in bytes
124.
125. _TEXT      ENDS
126.            END
```

guaranteed that they can still be used to access subsequent instructions or data. And immediately after the protected mode is activated, the CPU's instruction queue must be flushed because it contains pre-fetched real mode instructions, no longer valid in protected mode. The queue can be flushed by executing a jump to the next instruction (line 52).

The last thing to do in 16-bit is to switch to 32-bit. This step is achieved by loading the code segment register (CS) with a selector referring to a 32-bit executable code descriptor. The second entry in the GDT is such a descriptor. At line 57, the FJMP32 jump macro (lines 27 to 32) is executed with the selector 08h and the Start32 offset. Because the descriptor contains a base address of zero, the offset simply has to be the physical location of

the first instruction to execute in 32-bit mode. In this case, this instruction is at Start32 (line 65). The entry into the 32-bit mode marks the end of the 16-bit code (as well as the tricky encoded instructions).

Once in 32-bit mode, the best thing to do is initialize the data registers (DS, ES, FS, GS) and the stack register (SS) (lines 73 to 78). Note that there are 16-bit and 32-bit stack segments: the size determines how many bytes (two or four) a normal push saves on the stack (the push and pop opcodes are identical in 16-bit and 32-bit). The example uses the third GDT entry as a combined data and 32-bit stack segment (selector 10h). Finally, ESP (the stack pointer) is set to an arbitrary top of stack.

And that's it—the code is running in 32-bit protected mode, in a flat address space of 4GB! A complete system would have to continue its initialization, such as loading and initializing the Interrupt Descriptor Table (IDT), set-up some hardware, and so on. These issues are beyond the scope of the article, although they are fully addressed in the examples that can be downloaded from the Web site. The next articles will explore segmentation (including protection) and paging in detail. ESP

Jean Gareau graduated from the Polytechnic School of the University of Montreal in 1992 with an M.S. in electrical engineering. He has been involved since 1989 in the development of operating systems, system tools, and large commercial applications. He can be reached at jeangareau@yahoo.com.

REFERENCES

1. Ganssle, Jack G., "Future Challenges," *Embedded Systems Programming 1997 Buyer's Guide*, p. 7.

2. Labrosse, Jean J. *μC/OS The Real-Time Kernel*. Lawrence, KS: R&D Publications, Fourth Printing, 1992.

OTHER SOURCES

80386 Programmer's Reference Manual, Order Number 230985-001, Chandler, AZ: Intel Corp., 1987.

Reference: Microsoft MASM 6.11, Document Number DB35749-1292, Redmond, WA: Microsoft Corp., 1992.

Programmer's Guide: Microsoft MASM 6.11, Document Number DB35747-1292, Redmond, WA: Microsoft Corp., 1992.