Chapter 3

Assume Nothing

Chapter 3

Understanding and Using the Zen Timer

When you're pushing the envelope in writing optimized PC code, you're likely to become more than a little compulsive about finding approaches that let you wring more speed from your computer. In the process, you're bound to make mistakes, which is fine—as long as you watch for those mistakes and *learn* from them.

A case in point: A few years back, I came across an article about 8088 assembly language called "Optimizing for Speed." Now, "optimize" is not a word to be used lightly; *Webster's Ninth New Collegiate Dictionary* defines optimize as "to make as perfect, effective, or functional as possible," which certainly leaves little room for error. The author had, however, chosen a small, well-defined 8088 assembly language routine to refine, consisting of about 30 instructions that did nothing more than expand 8 bits to 16 bits by duplicating each bit.

The author of "Optimizing" had clearly fine-tuned the code with care, examining alternative instruction sequences and adding up cycles until he arrived at an implementation he calculated to be nearly 50 percent faster than the original routine. In short, he had used all the information at his disposal to improve his code, and had, as a result, saved cycles by the bushel. There was, in fact, only one slight problem with the optimized version of the routine....

It ran slower than the original version!

The Costs of Ignorance

As diligent as the author had been, he had nonetheless committed a cardinal sin of x86 assembly language programming: He had assumed that the information available to him was both correct and complete. While the execution times provided by Intel for its processors are indeed correct, they are incomplete; the other—and often more important—part of code performance is instruction *fetch* time, a topic to which I will return in later chapters.

Had the author taken the time to measure the true performance of his code, he wouldn't have put his reputation on the line with relatively low-performance code. What's more, had he actually measured the performance of his code and found it to be unexpectedly slow, curiosity might well have led him to experiment further and thereby add to his store of reliable information about the CPU.

There you have an important tenet of assembly language optimization: After crafting the best code possible, check it in action to see if it's really doing what you think it is. If it's not behaving as expected, that's all to the good, since solving mysteries is the path to knowledge. You'll learn more in this way, I assure you, than from any manual or book on assembly language.

Assume nothing. I cannot emphasize this strongly enough—when you care about performance, do your best to improve the code and then *measure* the improvement. If you don't measure performance, you're just guessing, and if you're guessing, you're not very likely to write top-notch code.

Ignorance about true performance can be costly. When I wrote video games for a living, I spent days at a time trying to wring more performance from my graphics drivers. I rewrote whole sections of code just to save a few cycles, juggled registers, and relied heavily on blurry-fast register-to-register shifts and adds. As I was writing my last game, I discovered that the program ran perceptibly faster if I used look-up tables instead of shifts and adds for my calculations. It *shouldn't* have run faster, according to my cycle counting, but it did. In truth, instruction fetching was rearing its head again, as it often does, and the fetching of the shifts and adds was taking as much as four times the nominal execution time of those instructions.

Ignorance can also be responsible for considerable wasted effort. I recall a debate in the letters column of one computer magazine about exactly how quickly text can be drawn on a Color/Graphics Adapter (CGA) screen without causing snow. The letter-writers counted every cycle in their timing loops, just as the author in the story that started this chapter had. Like that author, the letter-writers had failed to take the prefetch queue into account. In fact, they had neglected the effects of video wait states as well, so the code they discussed was actually *much* slower than their estimates. The proper test would, of course, have been to run the code to see if snow resulted, since the only true measure of code performance is observing it in action.

The Zen Timer

Clearly, one key to mastering Zen-class optimization is a tool with which to measure code performance. The most accurate way to measure performance is with expensive hardware, but reasonable measurements at no cost can be made with the PC's 8253 timer chip, which counts at a rate of slightly over 1,000,000 times per second. The 8253 can be started at the beginning of a block of code of interest and stopped at the end of that code, with the resulting count indicating how long the code took to execute with an accuracy of about 1 microsecond. (A microsecond is one millionth of a second, and is abbreviated µs). To be precise, the 8253 counts once every 838.1 nanoseconds. (A nanosecond is one billionth of a second, and is abbreviated ns.)

Listing 3.1 shows 8253-based timer software, consisting of three subroutines: **ZTimerOn**, **ZTimerOff**, and **ZTimerReport**. For the remainder of this book, I'll refer to these routines collectively as the "Zen timer." C-callable versions of the two precision Zen timers are presented in Chapter K on the companion CD-ROM.

LISTING 3.1 PZTIMER.ASM

```
; The precision Zen timer (PZTIMER.ASM)
;
; Uses the 8253 timer to time the performance of code that takes
; less than about 54 milliseconds to execute, with a resolution
; of better than 10 microseconds.
;
; By Michael Abrash
;
; Externally callable routines:
;
;  ZTimerOn: Starts the Zen timer, with interrupts disabled.
;
;  ZTimerOff: Stops the Zen timer, saves the timer count,
;     times the overhead code, and restores interrupts to the
;     state they were in when ZTimerOn was called.
;
;  ZTimerReport: Prints the net time that passed between starting
;     and stopping the timer.
;
; Note: If longer than about 54 ms passes between ZTimerOn and
;     ZTimerOff calls, the timer turns over and the count is
;     inaccurate. When this happens, an error message is displayed
;     instead of a count. The long-period Zen timer should be used
;     in such cases.
;
; Note: Interrupts *MUST* be left off between calls to ZTimerOn
;     and ZTimerOff for accurate timing and for detection of
;     timer overflow.
;
; Note: These routines can introduce slight inaccuracies into the
;     system clock count for each code section timed even if
;     timer 0 doesn't overflow. If timer 0 does overflow, the
;     system clock can become slow by virtually any amount of
;     time, since the system clock can't advance while the
;     precison timer is timing. Consequently, it's a good idea
;     to reboot at the end of each timing session. (The
;     battery-backed clock, if any, is not affected by the Zen
;     timer.)
;
; All registers, and all flags except the interrupt flag, are
; preserved by all routines. Interrupts are enabled and then disabled
; by ZTimerOn, and are restored by ZTimerOff to the state they were
; in when ZTimerOn was called.
;

Code  segment word public 'CODE'
      assume     cs:Code, ds:nothing
      public     ZTimerOn, ZTimerOff, ZTimerReport

;
; Base address of the 8253 timer chip.
;
BASE_8253        equu   40h
;
; The address of the timer 0 count registers in the 8253.
;
TIMER_0_8253     equ   BASE_8253 + 0
;
; The address of the mode register in the 8253.
;
MODE_8253        equ   BASE_8253 + 3
;
; The address of Operation Command Word 3 in the 8259 Programmable
; Interrupt Controller (PIC) (write only, and writable only when
; bit 4 of the byte written to this address is 0 and bit 3 is 1).
;
OCW3             equ   20h
;
; The address of the Interrupt Request register in the 8259 PIC
; (read only, and readable only when bit 1 of OCW3 = 1 and bit 0
; of OCW3 = 0).
;
IRR              equ   20h
;
; Macro to emulate a POPF instruction in order to fix the bug in some
; 80286 chips which allows interrupts to occur during a POPF even when
; interrupts remain disabled.
;
MPOPF macro
      local p1, p2
      jmp short p2
p1:   iret             ; jump to pushed address & pop flags
p2:   push  cs         ; construct far return address to
      call  p1         ; the next instruction
      endm

;
; Macro to delay briefly to ensure that enough time has elapsed
; between successive I/O accesses so that the device being accessed
; can respond to both accesses even on a very fast PC.
;
DELAY macro
      jmp   $+2
      jmp   $+2
      jmp   $+2
      endm
```

```
OriginalFlags     db    ?     ; storage for upper byte of
                              ; FLAGS register when
                              ; ZTimerOn called
TimedCount        dw    ?     ; timer 0 count when the timer
                              ; is stopped
ReferenceCount    dw          ; number of counts required to
                              ; execute timer overhead code
OverflowFlag      db    ?     ; used to indicate whether the
                              ; timer overflowed during the
                              ; timing interval
;
; String printed to report results.
;
OutputStr   label byte
            db    0dh, 0ah, 'Timed count: ', 5 dup (?)
ASCIICountEnd     label byte
            db    ' microseconds', 0dh, 0ah
            db    '$'
;
; String printed to report timer overflow.
;
OverflowStr label byte
      db    0dh, 0ah
      db    '****************************************************'
      db    0dh, 0ah
      db    '* The timer overflowed, so the interval timed was  *'
      db    0dh, 0ah
      db    '* too long for the precision timer to measure.     *'
      db    0dh, 0ah
      db    '* Please perform the timing test again with the    *'
      db    0dh, 0ah
      db    '* long-period timer.                               *'
      db    0dh, 0ah
      db    '****************************************************'
      db    0dh, 0ah
      db    '$'

; ********************************************************************
; * Routine called to start timing.                                  *
; ********************************************************************

ZTimerOn    proc  near

;
; Save the context of the program being timed.
;
      push  ax
      pushf
      pop   ax                    ; get flags so we can keep
                                  ; interrupts off when leaving
                                  ; this routine
      mov   cs:[OriginalFlags],ah ; remember the state of the
                                  ; Interrupt flag
      and   ah,0fdh               ; set pushed interrupt flag
                                  ; to 0
      push  ax
;
; Turn on interrupts, so the timer interrupt can occur if it's
; pending.
;
      sti
```

```
;
; Set timer 0 of the 8253 to mode 2 (divide-by-N), to cause
; linear counting rather than count-by-two counting. Also
; leaves the 8253 waiting for the initial timer 0 count to
; be loaded.
;
	mov	al,00110100b		;mode 2
	out	MODE_8253,al
;
; Set the timer count to 0, so we know we won't get another
; timer interrupt right away.
; Note: this introduces an inaccuracy of up to 54 ms in the system
; clock count each time it is executed.
;
	DELAY
	sub	al,al
	out	TIMER_0_8253,al		;lsb
	DELAY
	out	TIMER_0_8253,al		;msb
;
; Wait before clearing interrupts to allow the interrupt generated
; when switching from mode 3 to mode 2 to be recognized. The delay
; must be at least 210 ns long to allow time for that interrupt to
; occur. Here, 10 jumps are used for the delay to ensure that the
; delay time will be more than long enough even on a very fast PC.
;
	rept 10
	jmp	$+2
	endm
;
; Disable interrupts to get an accurate count.
;
	cli
;
; Set the timer count to 0 again to start the timing interval.
;
	mov	al,00110100b		; set up to load initial
	out	MODE_8253,al		; timer count
	DELAY
	sub	al,al
	out	TIMER_0_8253,al		; load count lsb
	DELAY
	out	TIMER_0_8253,al		; load count msb
;
; Restore the context and return.
;
	MPOPF				; keeps interrupts off
	pop	ax
	ret

ZTimerOn	endp

;********************************************************************
;* Routine called to stop timing and get count.                     *
;********************************************************************

ZTimerOff proc	near
```

```
;
; Save the context of the program being timed.
;
     push  ax
     push  cx
     pushf
;
; Latch the count.
;
     mov   al,00000000b          ; latch timer 0
     out   MODE_8253,al
;
; See if the timer has overflowed by checking the 8259 for a pending
; timer interrupt.
;
     mov   al,00001010b          ; OCW3, set up to read
     out   OCW3,al               ; Interrupt Request register
     DELAY
     in    al,IRR                ; read Interrupt Request
                                 ; register
     and   al,1                  ; set AL to 1 if IRQ0 (the
                                 ; timer interrupt) is pending
     mov   cs:[OverflowFlag],al  ; store the timer overflow
                                 ; status
;
; Allow interrupts to happen again.
;
     sti
;
; Read out the count we latched earlier.
;
     in    al,TIMER_0_8253  ; least significant byte
     DELAY
     mov   ah,al
     in    al,TIMER_0_8253  ; most significant byte
     xchg  ah,al
     neg   ax               ; convert from countdown
                            ; remaining to elapsed
                            ; count
     mov   cs:[TimedCount],ax
; Time a zero-length code fragment, to get a reference for how
; much overhead this routine has. Time it 16 times and average it,
; for accuracy, rounding the result.
;
     mov   cs:[ReferenceCount],0
     mov   cx,16
     cli                    ; interrupts off to allow a
                            ; precise reference count
RefLoop:
     call  ReferenceZTimerOn
     call  ReferenceZTimerOff
     loop  RefLoop
     sti
     add   cs:[ReferenceCount],8  ; total + (0.5 * 16)
     mov   cl,4
     shr   cs:[ReferenceCount],cl ; (total) / 16 + 0.5
;
; Restore original interrupt state.
;
     pop   ax                    ; retrieve flags when called
     mov   ch,cs:[OriginalFlags]  ; get back the original upper
                                  ; byte of the FLAGS register
     and   ch,not 0fdh            ; only care about original
                                  ; interrupt flag...
     and   ah,0fdh                ; ...keep all other flags in
                                  ; their current condition
     or    ah,ch                  ; make flags word with original
                                  ; interrupt flag
     push  ax                     ; prepare flags to be popped
;
; Restore the context of the program being timed and return to it.
;
     MPOPF                        ; restore the flags with the
                                  ; original interrupt state
     pop   cx
     pop   ax
     ret

ZTimerOff endp

;
; Called by ZTimerOff to start timer for overhead measurements.
;

ReferenceZTimerOnproc  near
;
; Save the context of the program being timed.
;
     push  ax
     pushf       ; interrupts are already off
;
; Set timer 0 of the 8253 to mode 2 (divide-by-N), to cause
; linear counting rather than count-by-two counting.
;
     mov   al,00110100b     ; set up to load
     out   MODE_8253,al     ; initial timer count
     DELAY
;
; Set the timer count to 0.
;
     sub   al,al
     out   TIMER_0_8253,al  ; load count lsb
     DELAY
     out   TIMER_0_8253,al  ; load count msb
;
; Restore the context of the program being timed and return to it.
;
     MPOPF
     pop   ax
     ret

ReferenceZTimerOnendp

;
; Called by ZTimerOff to stop timer and add result to ReferenceCount
; for overhead measurements.
;
```

```
ReferenceZTimerOff proc     near
;
; Save the context of the program being timed.
;
      push  ax
      push  cx
      pushf
;
; Latch the count and read it.
;
      mov   al,00000000b           ; latch timer 0
      out   MODE_8253,al
      DELAY
      in    al,TIMER_0_8253        ; lsb
      DELAY
      mov   ah,al
      in    al,TIMER_0_8253        ; msb
      xchg  ah,al
      neg   ax                     ; convert from countdown
                                   ; remaining to amount
                                   ; counted down
      add   cs:[ReferenceCount],ax
;
; Restore the context of the program being timed and return to it.
;
      MPOPF
      pop   cx
      pop   ax
      ret

ReferenceZTimerOff endp

; ********************************************************************
; * Routine called to report timing results.                        *
; ********************************************************************

ZTimerReport     proc  near

      pushf
      push  ax
      push  bx
      push  cx
      push  dx
      push  si
      push  ds
;
      push       cs    ; DOS functions require that DS point
      pop        ds    ; to text to be displayed on the screen
      assume     ds    :Code
;
; Check for timer 0 overflow.
;
      cmp   [OverflowFlag],0
      jz    PrintGoodCount
      mov   dx,offset OverflowStr
      mov   ah,9
      int   21h
      jmp   short EndZTimerReport
;
; Convert net count to decimal ASCII in microseconds.
;
```

```
PrintGoodCount:
      mov   ax,[TimedCount]
      sub   ax,[ReferenceCount]
      mov   si,offset ASCIICountEnd - 1
;
; Convert count to microseconds by multiplying by .8381.
;
      mov   dx,8381
      mul   dx
      mov   bx,10000
      div   bx          ;* .8381 - * 8381 / 10000
;
; Convert time in microseconds to 5 decimal ASCII digits.
;
      mov   bx,10
      mov   cx,5
CTSLoop:
      sub   dx,dx
      div   bx
      add   dl,'0'
      mov   [si],dl
      dec   si
      loop  CTSLoop
;
; Print the results.
;
      mov   ah,9
      mov   dx,offset OutputStr
      int   21h
;
EndZTimerReport:
      pop   ds
      pop   si
      pop   dx
      pop   cx
      pop   bx
      pop   ax
      MPOPF
      ret

ZTimerReport     endp

Code  ends
      end
```

The Zen Timer Is a Means, Not an End

We're going to spend the rest of this chapter seeing what the Zen timer can do, examining how it works, and learning how to use it. I'll be using the Zen timer again and again over the course of this book, so it's essential that you learn what the Zen timer can do and how to use it. On the other hand, it is by no means essential that you understand exactly how the Zen timer works. (Interesting, yes; essential, no.)

In other words, the Zen timer isn't really part of the knowledge we seek; rather, it's one tool with which we'll acquire that knowledge. Consequently, you shouldn't worry if you don't fully grasp the inner workings of the Zen timer. Instead, focus on learning how to *use* it, and you'll be on the right road.

Starting the Zen Timer

ZTimerOn is called at the start of a segment of code to be timed. **ZTimerOn** saves the context of the calling code, disables interrupts, sets timer 0 of the 8253 to mode 2 (divide-by-N mode), sets the initial timer count to 0, restores the context of the calling code, and returns. (I'd like to note that while Intel's documentation for the 8253 seems to indicate that a timer won't reset to 0 until it finishes counting down, in actual practice, timers seem to reset to 0 as soon as they're loaded.)

Two aspects of **ZTimerOn** are worth discussing further. One point of interest is that **ZTimerOn** disables interrupts. (**ZTimerOff** later restores interrupts to the state they were in when **ZTimerOn** was called.) Were interrupts not disabled by **ZTimerOn**, keyboard, mouse, timer, and other interrupts could occur during the timing interval, and the time required to service those interrupts would incorrectly and erratically appear to be part of the execution time of the code being measured. As a result, code timed with the Zen timer should not expect any hardware interrupts to occur during the interval between any call to **ZTimerOn** and the corresponding call to **ZTimerOff**, and should not enable interrupts during that time.

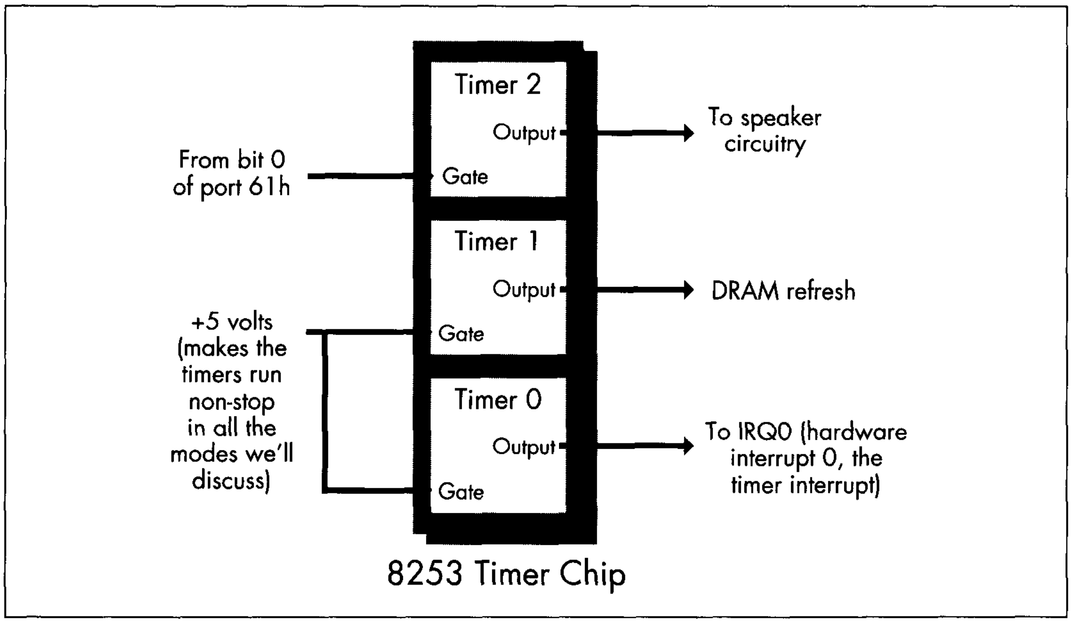
Time and the PC

A second interesting point about **ZTimerOn** is that it may introduce some small inaccuracy into the system clock time whenever it is called. To understand why this is so, we need to examine the way in which both the 8253 and the PC's system clock (which keeps the current time) work.

The 8253 actually contains three timers, as shown in Figure 3.1. All three timers are driven by the system board's 14.31818 MHz crystal, divided by 12 to yield a 1.19318 MHz clock to the timers, so the timers count once every 838.1 ns. Each of the three timers counts down in a programmable way, generating a signal on its output pin when it counts down to 0. Each timer is capable of being halted at any time via a 0 level on its gate input; when a timer's gate input is 1, that timer counts constantly. All in all, the 8253's timers are inherently very flexible timing devices; unfortunately, much of that flexibility depends on how the timers are connected to external circuitry, and in the PC the timers are connected with specific purposes in mind.

Timer 2 drives the speaker, although it can be used for other timing purposes when the speaker is not in use. As shown in Figure 3.1, timer 2 is the only timer with a programmable gate input in the PC; that is, timer 2 is the only timer that can be started and stopped under program control in the manner specified by Intel. On the other hand, the *output* of timer 2 is connected to nothing other than the speaker. In particular, timer 2 cannot generate an interrupt to get the 8088's attention.

Timer 1 is dedicated to providing dynamic RAM refresh, and should not be tampered with lest system crashes result.


Timer 2
Output
Gate
From bit 0 of port 61h
To speaker circuitry
Timer 1
Output
Gate
DRAM refresh
+5 volts (makes the timers run non-stop in all the modes we'll discuss)
Timer 0
Output
Gate
To IRQ0 (hardware interrupt 0, the timer interrupt)
8253 Timer Chip

The configuration of the 8253 timer chip in the PC.
Figure 3.1

Finally, timer 0 is used to drive the system clock. As programmed by the BIOS at power-up, every 65,536 (64K) counts, or 54.925 milliseconds, timer 0 generates a rising edge on its output line. (A millisecond is one-thousandth of a second, and is abbreviated ms.) This line is connected to the hardware interrupt 0 (IRQ0) line on the system board, so every 54.925 ms, timer 0 causes hardware interrupt 0 to occur.

The interrupt vector for IRQ0 is set by the BIOS at power-up time to point to a BIOS routine, **TIMER_INT**, that maintains a time-of-day count. **TIMER_INT** keeps a 16-bit count of IRQ0 interrupts in the BIOS data area at address 0000:046C (all addresses in this book are given in segment:offset hexadecimal pairs); this count turns over once an hour (less a few microseconds), and when it does, **TIMER_INT** updates a 16-bit hour count at address 0000:046E in the BIOS data area. This count is the basis for the current time and date that DOS supports via functions 2AH (2A hexadecimal) through 2DH and by way of the DATE and TIME commands.

Each timer channel of the 8253 can operate in any of six modes. Timer 0 normally operates in mode 3: *square wave mode.* In square wave mode, the initial count is counted down two at a time; when the count reaches zero, the output state is changed. The initial count is again counted down two at a time, and the output state is toggled back when the count reaches zero. The result is a square wave that changes state more slowly than the input clock by a factor of the initial count. In its normal mode of operation, timer 0 generates an output pulse that is low for about 27.5 ms and high for about 27.5 ms; this pulse is sent to the 8259 interrupt controller, and its rising edge generates a timer interrupt once every 54.925 ms.

Square wave mode is not very useful for precision timing because it counts down by two twice per timer interrupt, thereby rendering exact timings impossible. Fortunately, the 8253 offers another timer mode, mode 2 (divide-by-N mode), which is both a good substitute for square wave mode and a perfect mode for precision timing.

Divide-by-N mode counts down by one from the initial count. When the count reaches zero, the timer turns over and starts counting down again without stopping, and a pulse is generated for a single clock period. While the pulse is not held for nearly as long as in square wave mode, it doesn't matter, since the 8259 interrupt controller is configured in the PC to be edge-triggered and hence cares only about the existence of a pulse from timer 0, not the duration of the pulse. As a result, timer 0 continues to generate timer interrupts in divide-by-N mode, and the system clock continues to maintain good time.

Why not use timer 2 instead of timer 0 for precision timing? After all, timer 2 has a programmable gate input and isn't used for anything but sound generation. The problem with timer 2 is that its output can't generate an interrupt; in fact, timer 2 can't do anything but drive the speaker. We need the interrupt generated by the output of timer 0 to tell us when the count has overflowed, and we will see shortly that the timer interrupt also makes it possible to time much longer periods than the Zen timer shown in Listing 3.1 supports.

In fact, the Zen timer shown in Listing 3.1 can only time intervals of up to about 54 ms in length, since that is the period of time that can be measured by timer 0 before its count turns over and repeats. fifty-four ms may not seem like a very long time, but even a CPU as slow as the 8088 can perform more than 1,000 divides in 54 ms, and division is the single instruction that the 8088 performs most slowly. If a measured period turns out to be longer than 54 ms (that is, if timer 0 has counted down and turned over), the Zen timer will display a message to that effect. A long-period Zen timer for use in such cases will be presented later in this chapter.

The Zen timer determines whether timer 0 has turned over by checking to see whether an IRQ0 interrupt is pending. (Remember, interrupts are off while the Zen timer runs, so the timer interrupt cannot be recognized until the Zen timer stops and enables interrupts.) If an IRQ0 interrupt is pending, then timer 0 has turned over and generated a timer interrupt. Recall that **ZTimerOn** initially sets timer 0 to 0, in order to allow for the longest possible period—about 54 ms—before timer 0 reaches 0 and generates the timer interrupt.

Now we're ready to look at the ways in which the Zen timer can introduce inaccuracy into the system clock. Since timer 0 is initially set to 0 by the Zen timer, and since the system clock ticks only when timer 0 counts off 54.925 ms and reaches 0 again, an average inaccuracy of one-half of 54.925 ms, or about 27.5 ms, is incurred each time the Zen timer is started. In addition, a timer interrupt is generated when timer 0 is switched from mode 3 to mode 2, advancing the system clock by up to 54.925 ms, although this only happens the first time the Zen timer is run after a warm or cold boot. Finally, up to 54.925 ms can again be lost when **ZTimerOff** is called, since that routine again sets the timer count to zero. Net result: The system clock will run up to 110 ms (about a ninth of a second) slow each time the Zen timer is used.

Potentially far greater inaccuracy can be incurred by timing code that takes longer than about 110 ms to execute. Recall that all interrupts, including the timer interrupt, are disabled while timing code with the Zen timer. The 8259 interrupt controller is capable of remembering at most one pending timer interrupt, so all timer interrupts after the first one during any given Zen timing interval are ignored. Consequently, if a timing interval exceeds 54.9 ms, the system clock effectively stops 54.9 ms after the timing interval starts and doesn't restart until the timing interval ends, losing time all the while.

The effects on the system time of the Zen timer aren't a matter for great concern, as they are temporary, lasting only until the next warm or cold boot. Systems that have battery-backed clocks, (AT-style machines; that is, virtually all machines in common use) automatically reset the correct time whenever the computer is booted, and systems without battery-backed clocks prompt for the correct date and time when booted. Also, repeated use of the Zen timer usually makes the system clock slow by at most a total of a few seconds, unless code that takes much longer than 54 ms to run is timed (in which case the Zen timer will notify you that the code is too long to time).

Nonetheless, it's a good idea to reboot your computer at the end of each session with the Zen timer in order to make sure that the system clock is correct.

Stopping the Zen Timer

At some point after **ZTimerOn** is called, **ZTimerOff** must always be called to mark the end of the timing interval. **ZTimerOff** saves the context of the calling program, latches and reads the timer 0 count, converts that count from the countdown value that the timer maintains to the number of counts elapsed since **ZTimerOn** was called, and stores the result. Immediately after latching the timer 0 count—and before enabling interrupts—**ZTimerOff** checks the 8259 interrupt controller to see if there is a pending timer interrupt, setting a flag to mark that the timer overflowed if there is indeed a pending timer interrupt.

After that, **ZTimerOff** executes just the overhead code of **ZTimerOn** and **ZTimerOff** 16 times, and averages and saves the results in order to determine how many of the counts in the timing result just obtained were incurred by the overhead of the Zen timer rather than by the code being timed.

Finally, **ZTimerOff** restores the context of the calling program, including the state of the interrupt flag that was in effect when **ZTimerOn** was called to start timing, and returns.

One interesting aspect of **ZTimerOff** is the manner in which timer 0 is stopped in order to read the timer count. We don't actually have to stop timer 0 to read the count; the 8253 provides a special latched read feature for the specific purpose of reading the count while a time is running. (That's a good thing, too; we've no documented way to stop timer 0 if we wanted to, since its gate input isn't connected. Later in this chapter, though, we'll see that timer 0 can be stopped after all.) We simply tell the 8253 to latch the current count, and the 8253 does so without breaking stride.

Reporting Timing Results

ZTimerReport may be called to display timing results at any time after both **ZTimerOn** and **ZTimerOff** have been called. **ZTimerReport** first checks to see whether the timer overflowed (counted down to 0 and turned over) before **ZTimerOff** was called; if overflow did occur, **ZTimerOff** prints a message to that effect and returns. Otherwise, **ZTimerReport** subtracts the reference count (representing the overhead of the Zen timer) from the count measured between the calls to **ZTimerOn** and **ZTimerOff**, converts the result from timer counts to microseconds, and prints the resulting time in microseconds to the standard output.

Note that **ZTimerReport** need not be called immediately after **ZTimerOff**. In fact, after a given call to **ZTimerOff**, **ZTimerReport** can be called at any time right up until the next call to **ZTimerOn**.

You may want to use the Zen timer to measure several portions of a program while it executes normally, in which case it may not be desirable to have the text printed by **ZTimerReport** interfere with the program's normal display. There are many ways to deal with this. One approach is removal of the invocations of the DOS print string function (INT 21H with AH equal to 9) from **ZTimerReport**, instead running the program under a debugger that supports screen flipping (such as Turbo Debugger or CodeView), placing a breakpoint at the start of **ZTimerReport**, and directly observing the count in microseconds as **ZTimerReport** calculates it.

A second approach is modification of **ZTimerReport** to place the result at some safe location in memory, such as an unused portion of the BIOS data area.

A third approach is alteration of **ZTimerReport** to print the result over a serial port to a terminal or to another PC acting as a terminal. Similarly, many debuggers can be run from a remote terminal via a serial link.

Yet another approach is modification of **ZTimerReport** to send the result to the printer via either DOS function 5 or BIOS interrupt 17H.

A final approach is to modify **ZTimerReport** to print the result to the auxiliary output via DOS function 4, and to then write and load a special device driver named **AUX**, to which DOS function 4 output would automatically be directed. This device driver could send the result anywhere you might desire. The result might go to the secondary display adapter, over a serial port, or to the printer, or could simply be stored in a buffer within the driver, to be dumped at a later time. (Credit for this final approach goes to Michael Geary, and thanks go to David Miller for passing the idea on to me.)

You may well want to devise still other approaches better suited to your needs than those I've presented. Go to it! I've just thrown out a few possibilities to get you started.

Notes on the Zen Timer

The Zen timer subroutines are designed to be near-called from assembly language code running in the public segment **Code**. The Zen timer subroutines can, however, be called from any assembly or high-level language code that generates OBJ files that are compatible with the Microsoft linker, simply by modifying the segment that the timer code runs in to match the segment used by the code being timed, or by changing the Zen timer routines to far procedures and making far calls to the Zen timer code from the code being timed, as discussed at the end of this chapter. All three subroutines preserve all registers and all flags except the interrupt flag, so calls to these routines are transparent to the calling code.

If you do change the Zen timer routines to far procedures in order to call them from code running in another segment, be sure to make *all* the Zen timer routines far, including **ReferenceZTimerOn** and **ReferenceZTimerOff**. (You'll have to put **FAR PTR** overrides on the calls from **ZTimerOff** to the latter two routines if you do make them far.) If the reference routines aren't the same type—near or far—as the other routines, they won't reflect the true overhead incurred by starting and stopping the Zen timer.

Please be aware that the inaccuracy that the Zen timer can introduce into the system clock time does not affect the accuracy of the performance measurements reported by the Zen timer itself. The 8253 counts once every 838 ns, giving us a count resolution of about 1µs, although factors such as the prefetch queue (as discussed below), dynamic RAM refresh, and internal timing variations in the 8253 make it perhaps more accurate to describe the Zen timer as measuring code performance with an accuracy of better than 10µs. In fact, the Zen timer is actually most accurate in assessing code performance when timing intervals longer than about 100 µs. At any rate, we're most interested in using the Zen timer to assess the relative performance of various code sequences—that is, using it to compare and tweak code—and the timer is more than accurate enough for that purpose.

The Zen timer works on all PC-compatible computers I've tested it on, including XTs, ATs, PS/2 computers, and 386, 486, and Pentium-based machines. Of course, I haven't been able to test it on *all* PC-compatibles, but I don't expect any problems; computers on which the Zen timer doesn't run can't truly be called "PC-compatible."

On the other hand, there is certainly no guarantee that code performance as measured by the Zen timer will be the same on compatible computers as on genuine

IBM machines, or that either absolute or relative code performance will be similar even on different IBM models; in fact, quite the opposite is true. For example, every PS/2 computer, even the relatively slow Model 30, executes code much faster than does a PC or XT. As another example, I set out to do the timings for my earlier book *Zen of Assembly Language* on an XT-compatible computer, only to find that the computer wasn't quite IBM-compatible regarding code performance. The differences were minor, mind you, but my experience illustrates the risk of assuming that a specific make of computer will perform in a certain way without actually checking.

Not that this variation between models makes the Zen timer one whit less useful—quite the contrary. The Zen timer is an excellent tool for evaluating code performance over the entire spectrum of PC-compatible computers.

A Sample Use of the Zen Timer

Listing 3.2 shows a test-bed program for measuring code performance with the Zen timer. This program sets DS equal to CS (for reasons we'll discuss shortly), includes the code to be measured from the file TESTCODE, and calls **ZTimerReport** to display the timing results. Consequently, the code being measured should be in the file TESTCODE, and should contain calls to **ZTimerOn** and **ZTimerOff**.

LISTING 3.2 PZTEST.ASM

```
; Program to measure performance of code that takes less than
; 54 ms to execute. (PZTEST.ASM)
;
; Link with PZTIMER.ASM (Listing 3.1). PZTEST.BAT (Listing 3.4)
; can be used to assemble and link both files. Code to be
; measured must be in the file TESTCODE; Listing 3.3 shows
; a sample TESTCODE file.
;
; By Michael Abrash
;
mystack    segment    para stack 'STACK'
      db    512 dup(?)
mystack    ends
;
Code  segment     para public 'CODE'
      assume      cs:Code, ds:Code
      extrn ZTimerOn:near, ZTimerOff:near, ZTimerReport:near
Start proc  near
      push  cs
      pop   ds    ; set DS to point to the code segment,
                  ; so data as well as code can easily
                  ; be included in TESTCODE
;
      include     TESTCODE ;code to be measured, including
                  ; calls to ZTimerOn and ZTimerOff
;
; Display the results.
;
      call  ZTimerReport
;
; Terminate the program.
;
```

```
      mov   ah,4ch
      int   21h
Start endp
Code  ends
      end   Start
```

Listing 3.3 shows some sample code to be timed. This listing measures the time required to execute 1,000 loads of AL from the memory variable **MemVar**. Note that Listing 3.3 calls **ZTimerOn** to start timing, performs 1,000 **MOV** instructions in a row, and calls **ZTimerOff** to end timing. When Listing 3.2 is named TESTCODE and included by Listing 3.3, Listing 3.2 calls **ZTimerReport** to display the execution time after the code in Listing 3.3 has been run.

LISTING 3.3 LST3-3.ASM

```
; Test file;
; Measures the performance of 1,000 loads of AL from
; memory. (Use by renaming to TESTCODE, which is
; included by PZTEST.ASM (Listing 3.2). PZTIME.BAT
; (Listing 3.4) does this, along with all assembly
; and linking.)
;
      jmp   Skip  ;jump around defined data
;
MemVar      db    ?
;
Skip:
;
; Start timing.
;
      call  ZTimerOn
;
      rept  1000
      mov   al,[MemVar]
      endm
;
; Stop timing.
;
      call  ZTimerOff
```

It's worth noting that Listing 3.3 begins by jumping around the memory variable **MemVar**. This approach lets us avoid reproducing Listing 3.2 in its entirety for each code fragment we want to measure; by defining any needed data right in the code segment and jumping around that data, each listing becomes self-contained and can be plugged directly into Listing 3.2 as TESTCODE. Listing 3.2 sets DS equal to CS before doing anything else precisely so that data can be embedded in code fragments being timed. Note that only after the initial jump is performed in Listing 3.3 is the Zen timer started, since we don't want to include the execution time of start-up code in the timing interval. That's why the calls to **ZTimerOn** and **ZTimerOff** are in TESTCODE, not in PZTEST.ASM; this way, we have full control over which portion of TESTCODE is timed, and we can keep set-up code and the like out of the timing interval.

Listing 3.3 is used by naming it TESTCODE, assembling both Listing 3.2 (which includes TESTCODE) and Listing 3.1 with TASM or MASM, and linking the two resulting OBJ files together by way of the Borland or Microsoft linker. Listing 3.4 shows a batch file, PZTIME.BAT, which does all that; when run, this batch file generates and runs the executable file PZTEST.EXE. PZTIME.BAT (Listing 3.4) assumes that the file PZTIMER.ASM contains Listing 3.1, and the file PZTEST.ASM contains Listing 3.2. The command-line parameter to PZTIME.BAT is the name of the file to be copied to TESTCODE and included into PZTEST.ASM. (Note that Turbo Assembler can be substituted for MASM by replacing "masm" with "tasm" and "link" with "tlink" in Listing 3.4. The same is true of Listing 3.7.)

LISTING 3.4 PZTIME.BAT

```
echo off
rem
rem *** Listing 3.4 ***
rem
rem ***************************************************************
rem * Batch file PZTIME.BAT, which builds and runs the precision  *
rem * Zen timer program PZTEST.EXE to time the code named as the  *
rem * command-line parameter. Listing 3.1 must be named           *
rem * PZTIMER.ASM, and Listing 3.2 must be named PZTEST.ASM. To   *
rem * time the code in LST3-3, you'd type the DOS command:        *
rem *                                                             *
rem * pztime lst3-3                                               *
rem *                                                             *
rem * Note that MASM and LINK must be in the current directory or *
rem * on the current path in order for this batch file to work.   *
rem *                                                             *
rem * This batch file can be speeded up by assembling PZTIMER.ASM *
rem * once, then removing the lines:                              *
rem *                                                             *
rem * masm pztimer;                                               *
rem * if errorlevel 1 goto errorend                               *
rem *                                                             *
rem * from this file.                                             *
rem *                                                             *
rem * By Michael Abrash                                           *
rem ***************************************************************
rem
rem Make sure a file to test was specified.
rem
if not x%1==x goto ckexist
echo ***************************************************************
echo * Please specify a file to test.                              *
echo ***************************************************************
goto end
rem
rem Make sure the file exists.
rem
:ckexist
if exist %1 goto docopy
echo ***************************************************************
echo * The specified file, "%1," doesn't exist.                    *
echo ***************************************************************
goto end
```

```
rem
rem copy the file to measure to TESTCODE.
rem
:docopy
copy %1 testcode
masm pztest;
if errorlevel 1 goto errorend
masm pztimer;
if errorlevel 1 goto errorend
link pztest+pztimer;
if errorlevel 1 goto errorend
pztest
goto end
:errorend
echo *************************************************************
echo * An error occurred while building the precision Zen timer.   *
echo *************************************************************
:end
```

Assuming that Listing 3.3 is named LST3-3.ASM and Listing 3.4 is named PZTIME.BAT, the code in Listing 3.3 would be timed with the command:

```
pztime LST3-3.ASM
```

which performs all assembly and linking, and reports the execution time of the code in Listing 3.3.

When the above command is executed on an original 4.77 MHz IBM PC, the time reported by the Zen timer is 3619 µs, or about 3.62 µs per load of AL from memory. (While the exact number is 3.619 µs per load of AL, I'm going to round off that last digit from now on. No matter how many repetitions of a given instruction are timed, there's just too much noise in the timing process—between dynamic RAM refresh, the prefetch queue, and the internal state of the processor at the start of timing—for that last digit to have any significance.) Given the test PC's 4.77 MHz clock, this works out to about 17 cycles per **MOV**, which is actually a good bit longer than Intel's specified 10-cycle execution time for this instruction. (See the MASM or TASM documentation, or Intel's processor reference manuals, for official execution times.) Fear not, the Zen timer is right—**MOV AL,[MEMVAR]** really does take 17 cycles as used in Listing 3.3. Exactly why that is so is just what this book is all about.

In order to perform any of the timing tests in this book, enter Listing 3.1 and name it PZTIMER.ASM, enter Listing 3.2 and name it PZTEST.ASM, and enter Listing 3.4 and name it PZTIME.BAT. Then simply enter the listing you wish to run into the file *filename* and enter the command:

```
pztime <filename>
```

In fact, that's exactly how I timed each of the listings in this book. Code fragments you write yourself can be timed in just the same way. If you wish to time code directly in place in your programs, rather than in the test-bed program of Listing 3.2, simply insert calls to **ZTimerOn**, **ZTimerOff**, and **ZTimerReport** in the appropriate places and link PZTIMER to your program.

The Long-Period Zen Timer

With a few exceptions, the Zen timer presented above will serve us well for the remainder of this book since we'll be focusing on relatively short code sequences that generally take much less than 54 ms to execute. Occasionally, however, we will need to time longer intervals. What's more, it is very likely that you will want to time code sequences longer than 54 ms at some point in your programming career. Accordingly, I've also developed a Zen timer for periods longer than 54 ms. The long-period Zen timer (so named by contrast with the precision Zen timer just presented) shown in Listing 3.5 can measure periods up to one hour in length.

The key difference between the long-period Zen timer and the precision Zen timer is that the long-period timer leaves interrupts enabled during the timing period. As a result, timer interrupts are recognized by the PC, allowing the BIOS to maintain an accurate system clock time over the timing period. Theoretically, this enables measurement of arbitrarily long periods. Practically speaking, however, there is no need for a timer that can measure more than a few minutes, since the DOS time of day and date functions (or, indeed, the DATE and TIME commands in a batch file) serve perfectly well for longer intervals. Since very long timing intervals aren't needed, the long-period Zen timer uses a simplified means of calculating elapsed time that is limited to measuring intervals of an hour or less. If a period longer than an hour is timed, the long-period Zen timer prints a message to the effect that it is unable to time an interval of that length.

For implementation reasons, the long-period Zen timer is also incapable of timing code that starts before midnight and ends after midnight; if that eventuality occurs, the long-period Zen timer reports that it was unable to time the code because midnight was crossed. If this happens to you, just time the code again, secure in the knowledge that at least you won't run into the problem again for 23-odd hours.

You should not use the long-period Zen timer to time code that requires interrupts to be disabled for more than 54 ms at a stretch during the timing interval, since when interrupts are disabled the long-period Zen timer is subject to the same 54 ms maximum measurement time as the precision Zen timer.

While permitting the timer interrupt to occur allows long intervals to be timed, that same interrupt makes the long-period Zen timer less accurate than the precision Zen timer, since the time the BIOS spends handling timer interrupts during the timing interval is included in the time measured by the long-period timer. Likewise, any other interrupts that occur during the timing interval, most notably keyboard and mouse interrupts, will increase the measured time.

The long-period Zen timer has some of the same effects on the system time as does the precision Zen timer, so it's a good idea to reboot the system after a session with the long-period Zen timer. The long-period Zen timer does not, however, have the same potential for introducing major inaccuracy into the system clock time during a single timing run since it leaves interrupts enabled and therefore allows the system clock to update normally.

Stopping the Clock

There's a potential problem with the long-period Zen timer. The problem is this: In order to measure times longer than 54 ms, we must maintain not one but two timing components, the timer 0 count and the BIOS time-of-day count. The time-of-day count measures the passage of 54.9 ms intervals, while the timer 0 count measures time within those 54.9 ms intervals. We need to read the two time components simultaneously in order to get a clean reading. Otherwise, we may read the timer count just before it turns over and generates an interrupt, then read the BIOS time-of-day count just after the interrupt has occurred and caused the time-of-day count to turn over, with a resulting 54 ms measurement inaccuracy. (The opposite sequence—reading the time-of-day count and then the timer count—can result in a 54 ms inaccuracy in the other direction.)

The only way to avoid this problem is to stop timer 0, read both the timer and time-of-day counts while the timer is stopped, and then restart the timer. Alas, the gate input to timer 0 isn't program-controllable in the PC, so there's no documented way to stop the timer. (The latched read feature we used in Listing 3.1 doesn't stop the timer; it latches a count, but the timer keeps running.) What should we do?

As it turns out, an undocumented feature of the 8253 makes it possible to stop the timer dead in its tracks. Setting the timer to a new mode and waiting for an initial count to be loaded causes the timer to stop until the count is loaded. Surprisingly, the timer count remains readable and correct while the timer is waiting for the initial load.

In my experience, this approach works beautifully with fully 8253-compatible chips. However, there's no guarantee that it will always work, since it programs the 8253 in an undocumented way. What's more, IBM chose not to implement compatibility with this particular 8253 feature in the custom chips used in PS/2 computers. On PS/2 computers, we have no choice but to latch the timer 0 count and then stop the BIOS count (by disabling interrupts) as quickly as possible. We'll just have to accept the fact that on PS/2 computers we may occasionally get a reading that's off by 54 ms, and leave it at that.

I've set up Listing 3.5 so that it can assemble to either use or not use the undocumented timer-stopping feature, as you please. The **PS2** equate selects between the two modes of operation. If **PS2** is 1 (as it is in Listing 3.5), then the latch-and-read method is used; if **PS2** is 0, then the undocumented timer-stop approach is used. The latch-and-read method will work on all PC-compatible computers, but may occasionally produce results that are incorrect by 54 ms. The timer-stop approach avoids synchronization problems, but doesn't work on all computers.

LISTING 3.5 LZTIMER.ASM

```
;
; The long-period Zen timer. (LZTIMER.ASM)
; Uses the 8253 timer and the BIOS time-of-day count to time the
; performance of code that takes less than an hour to execute.
; Because interrupts are left on (in order to allow the timer
; interrupt to be recognized), this is less accurate than the
; precision Zen timer, so it is best used only to time code that takes
; more than about 54 milliseconds to execute (code that the precision
; Zen timer reports overflow on). Resolution is limited by the
; occurrence of timer interrupts.
;
; By Michael Abrash
;
; Externally callable routines:
;
;  ZTimerOn: Saves the BIOS time of day count and starts the
;     long-period Zen timer.
;
;  ZTimerOff: Stops the long-period Zen timer and saves the timer
;     count and the BIOS time-of-day count.
;
;  ZTimerReport: Prints the time that passed between starting and
;     stopping the timer.
;
; Note: If either more than an hour passes or midnight falls between
;     calls to ZTimerOn and ZTimerOff, an error is reported. For
;     timing code that takes more than a few minutes to execute,
;     either the DOS TIME command in a batch file before and after
;     execution of the code to time or the use of the DOS
;     time-of-day function in place of the long-period Zen timer is
;     more than adequate.
;
; Note: The PS/2 version is assembled by setting the symbol PS2 to 1.
;     PS2 must be set to 1 on PS/2 computers because the PS/2's
;     timers are not compatible with an undocumented timer-stopping
;     feature of the 8253; the alternative timing approach that
;     must be used on PS/2 computers leaves a short window
;     during which the timer 0 count and the BIOS timer count may
;     not be synchronized. You should also set the PS2 symbol to
;     1 if you're getting erratic or obviously incorrect results.
;
; Note: When PS2 is 0, the code relies on an undocumented 8253
;     feature to get more reliable readings. It is possible that
;     the 8253 (or whatever chip is emulating the 8253) may be put
;     into an undefined or incorrect state when this feature is
;     used.
;
;     *******************************************************************
;     * If your computer displays any hint of erratic behavior          *
;     *    after the long-period Zen timer is used, such as the floppy *
;     *    drive failing to operate properly, reboot the system, set   *
;     *    PS2 to 1 and leave it that way!                              *
;     *******************************************************************
;
```

```
; Note: Each block of code being timed should ideally be run several
;     times, with at least two similar readings required to
;     establish a true measurement, in order to eliminate any
;     variability caused by interrupts.
;
; Note: Interrupts must not be disabled for more than 54 ms at a
;     stretch during the timing interval. Because interrupts
;     are enabled, keys, mice, and other devices that generate
;     interrupts should not be used during the timing interval.
;
; Note: Any extra code running off the timer interrupt (such as
;     some memory-resident utilities) will increase the time
;     measured by the Zen timer.
;
; Note: These routines can introduce inaccuracies of up to a few
;     tenths of a second into the system clock count for each
;     code section timed. Consequently, it's a good idea to
;     reboot at the conclusion of timing sessions. (The
;     battery-backed clock, if any, is not affected by the Zen
;     timer.)
;
; All registers and all flags are preserved by all routines.
;

Code  segment word public 'CODE'
      assume     cs:Code, ds:nothing
      public     ZTimerOn, ZTimerOff, ZTimerReport


;
; Set PS2 to 0 to assemble for use on a fully 8253-compatible
; system; when PS2 is 0, the readings are more reliable if the
; computer supports the undocumented timer-stopping feature,
; but may be badly off if that feature is not supported. In
; fact, timer-stopping may interfere with your computer's
; overall operation by putting the 8253 into an undefined or
; incorrect state. Use with caution!!!
;
; Set PS2 to 1 to assemble for use on non-8253-compatible
; systems, including PS/2 computers; when PS2 is 1, readings
; may occasionally be off by 54 ms, but the code will work
; properly on all systems.
;
; A setting of 1 is safer and will work on more systems,
; while a setting of 0 produces more reliable results in systems
; which support the undocumented timer-stopping feature of the
; 8253. The choice is yours.
;
PS2                   equ   1
;
; Base address of the 8253 timer chip.
;
BASE_8253             equ   40h
;
; The address of the timer 0 count registers in the 8253.
;
TIMER_0_8253          equ   BASE_8253 + 0
;
; The address of the mode register in the 8253.
;
MODE_8253             equ   BASE_8253 + 3
;
```

```
; The address of the BIOS timer count variable in the BIOS
; data segment.
;
TIMER_COUNT      equu   46ch
;
; Macro to emulate a POPF instruction in order to fix the bug in some
; 80286 chips which allows interrupts to occur during a POPF even when
; interrupts remain disabled.
;
MPOPF macro
      local p1, p2
      jmp short p2
p1:   iret             ;jump to pushed address & pop flags
p2:   push  cs         ;construct far return address to
      call  p1         ; the next instruction
      endm

;
; Macro to delay briefly to ensure that enough time has elapsed
; between successive I/O accesses so that the device being accessed
; can respond to both accesses even on a very fast PC.
;
DELAY macro
      jmp   $+2
      jmp   $+2
      jmp   $+2
      endm

StartBIOSCountLowdw    ?          ;BIOS count low word at the
                                  ; start of the timing period
StartBIOSCountHigh     dw    ?    ;BIOS count high word at the
                                  ; start of the timing period
EndBIOSCountLow        dw    ?    ;BIOS count low word at the
                                  ; end of the timing period
EndBIOSCountHigh dw    ?          ;BIOS count high word at the
                                  ; end of the timing period
EndTimedCount          dw    ?    ;timer 0 count at the end of
                                  ; the timing period
ReferenceCount         dw    ?    ;number of counts required to
                                  ; execute timer overhead code
;
; String printed to report results.
;
OutputStr   label byte
            db    0dh, 0ah, 'Timed count: '
TimedCountStr    db    10 dup (?)
            db    ' microseconds', 0dh, 0ah
            db    '$'
;
; Temporary storage for timed count as it's divided down by powers
; of ten when converting from doubleword binary to ASCII.
;
CurrentCountLow        dw    ?
CurrentCountHigh dw    ?
;
; Powers of ten table used to perform division by 10 when doing
; doubleword conversion from binary to ASCII.
;
PowersOfTen label word
      dd    1
      dd    10
      dd    100
      dd    1000
      dd    10000
      dd    100000
      dd    1000000
      dd    10000000
      dd    100000000
      dd    1000000000
PowersOfTenEnd   label word
;
; String printed to report that the high word of the BIOS count
; changed while timing (an hour elapsed or midnight was crossed),
; and so the count is invalid and the test needs to be rerun.
;
TurnOverStr label byte
      db    0dh, 0ah
      db    '****************************************************'
      db    0dh, 0ah
      db    '* Either midnight passed or an hour or more passed *'
      db    0dh, 0ah
      db    '* while timing was in progress. If the former was  *'
      db    0dh, 0ah
      db    '* the case, please rerun the test; if the latter   *'
      db    0dh, 0ah
      db    '* was the case, the test code takes too long to    *'
      db    0dh, 0ah
      db    '* run to be timed by the long-period Zen timer.    *'
      db    0dh, 0ah
      db    '* Suggestions: use the DOS TIME command, the DOS   *'
      db    0dh, 0ah
      db    '* time function, or a watch.                       *'
      db    0dh, 0ah
      db    '****************************************************'
      db    0dh, 0ah
      db    '$'

;********************************************************************
;* Routine called to start timing.                                  *
;********************************************************************

ZTimerOn    proc  near

;
; Save the context of the program being timed.
;
      push  ax
      pushf
;
; Set timer 0 of the 8253 to mode 2 (divide-by-N), to cause
; linear counting rather than count-by-two counting. Also stops
; timer 0 until the timer count is loaded, except on PS/2
; computers.
;
      mov   al,00110100b           ;mode 2
      out   MODE_8253,al
;
; Set the timer count to 0, so we know we won't get another
; timer interrupt right away.
; Note: this introduces an inaccuracy of up to 54 ms in the system
; clock count each time it is executed.
;
```

```
        DELAY
        sub   al,al
        out   TIMER_0_8253,al        ;lsb
        DELAY
        out   TIMER_0_8253,al        ;msb
;
; In case interrupts are disabled, enable interrupts briefly to allow
; the interrupt generated when switching from mode 3 to mode 2 to be
; recognized. Interrupts must be enabled for at least 210 ns to allow
; time for that interrupt to occur. Here, 10 jumps are used for the
; delay to ensure that the delay time will be more than long enough
; even on a very fast PC.
;
        pushf
        sti
        rept 10
        jmp   $+2
        endm
        MPOPF
;
; Store the timing start BIOS count.
; (Since the timer count was just set to 0, the BIOS count will
; stay the same for the next 54 ms, so we don't need to disable
; interrupts in order to avoid getting a half-changed count.)
;
        push  ds
        sub   ax,ax
        mov   ds,ax
        mov   ax,ds:[TIMER_COUNT+2]
        mov   cs:[StartBIOSCountHigh],ax
        mov   ax,ds:[TIMER_COUNT]
        mov   cs:[StartBIOSCountLow],ax
        pop   ds
;
; Set the timer count to 0 again to start the timing interval.
;
        mov   al,00110100b           ;set up to load initial
        out   MODE_8253,al           ; timer count
        DELAY
        sub   al,al
        out   TIMER_0_8253,al        ;load count lsb
        DELAY
        out   TIMER_0_8253,al        ;load count msb
;
; Restore the context of the program being timed and return to it.
;
        MPOPF
        pop   ax
        ret

ZTimerOn    endp

;********************************************************************
;* Routine called to stop timing and get count.                     *
;********************************************************************

ZTimerOff proc    near

;
; Save the context of the program being timed.
;
```

```
        pushf
        push  ax
        push  cx
;
; In case interrupts are disabled, enable interrupts briefly to allow
; any pending timer interrupt to be handled. Interrupts must be
; enabled for at least 210 ns to allow time for that interrupt to
; occur. Here, 10 jumps are used for the delay to ensure that the
; delay time will be more than long enough even on a very fast PC.
;
        sti
        rept  10
        jmp   $+2
        endm

;
; Latch the timer count.
;

if PS2

        mov   al,00000000b
        out   MODE_8253,al            ;latch timer 0 count
;
; This is where a one-instruction-long window exists on the PS/2.
; The timer count and the BIOS count can lose synchronization;
; since the timer keeps counting after it's latched, it can turn
; over right after it's latched and cause the BIOS count to turn
; over before interrupts are disabled, leaving us with the timer
; count from before the timer turned over coupled with the BIOS
; count from after the timer turned over. The result is a count
; that's 54 ms too long.
;

else

;
; Set timer 0 to mode 2 (divide-by-N), waiting for a 2-byte count
; load, which stops timer 0 until the count is loaded. (Only works
; on fully 8253-compatible chips.)
;
        mov   al,00110100b            ;mode 2
        out   MODE_8253,al
        DELAY
        mov   al,00000000b            ;latch timer 0 count
        out   MODE_8253,al

endif

        cli                           ;stop the BIOS count
;
; Read the BIOS count. (Since interrupts are disabled, the BIOS
; count won't change.)
;
        push  ds
        sub   ax,ax
        mov   ds,ax
        mov   ax,ds:[TIMER_COUNT+2]
        mov   cs:[EndBIOSCountHigh],ax
        mov   ax,ds:[TIMER_COUNT]
```

```
      mov   cs:[EndBIOSCountLow],ax
      pop   ds
;
; Read the timer count and save it.
;
      in    al,TIMER_0_8253        ;lsb
      DELAY
      mov   ah,al
      in    al,TIMER_0_8253        ;msb
      xchg  ah,al
      neg   ax                     ;convert from countdown
                                   ; remaining to elapsed
                                   ; count
      mov   cs:[EndTimedCount],ax
;
; Restart timer 0, which is still waiting for an initial count
; to be loaded.
;

ife PS2

      DELAY
      mov   al,00110100b           ;mode 2, waiting to load a
                                   ; 2-byte count
      out   MODE_8253,al
      DELAY
      sub   al,al
      out   TIMER_0_8253,al        ;lsb
      DELAY
      mov   al,ah
      out   TIMER_0_8253,al        ;msb
      DELAY

endif

      sti                          ;let the BIOS count continue
;
; Time a zero-length code fragment, to get a reference for how
; much overhead this routine has. Time it 16 times and average it,
; for accuracy, rounding the result.
;
      mov   cs:[ReferenceCount],0
      mov   cx,16
      cli                          ;interrupts off to allow a
                                   ; precise reference count
RefLoop:
      call  ReferenceZTimerOn
      call  ReferenceZTimerOff
      loop  RefLoop
      sti
      add   cs:[ReferenceCount],8        ;total + (0.5 * 16)
      mov   cl,4
      shr   cs:[ReferenceCount],cl ;(total) / 16 + 0.5
;
; Restore the context of the program being timed and return to it.
;
      pop   cx
      pop   ax
      MPOPF
      ret
```

```
ZTimerOff endp

;
; Called by ZTimerOff to start the timer for overhead measurements.
;

ReferenceZTimerOnproc  near
;
; Save the context of the program being timed.
;
      push  ax
      pushf
;
; Set timer 0 of the 8253 to mode 2 (divide-by-N), to cause
; linear counting rather than count-by-two counting.
;
      mov   al,00110100b     ;mode 2
      out   MODE_8253,al
;
; Set the timer count to 0.
;
      DELAY
      sub   al,al
      out   TIMER_0_8253,al        ;lsb
      DELAY
      out   TIMER_0_8253,al        ;msb
;
; Restore the context of the program being timed and return to it.
;
      MPOPF
      pop   ax
      ret

ReferenceZTimerOnendp

;
; Called by ZTimerOff to stop the timer and add the result to
; ReferenceCount for overhead measurements. Doesn't need to look
; at the BIOS count because timing a zero-length code fragment
; isn't going to take anywhere near 54 ms.
;

ReferenceZTimerOff proc     near
;
; Save the context of the program being timed.
;
      pushf
      push  ax
      push  cx

;
; Match the interrupt-window delay in ZTimerOff.
;
      sti
      rept  10
      jmp   $+2
      endm

      mov   al,00000000b
      out   MODE_8253,al           ;latch timer
```

```
;
; Read the count and save it.
;
      DELAY
      in    al,TIMER_0_8253        ;lsb
      DELAY
      mov   ah,al
      in    al,TIMER_0_8253        ;msb
      xchg  ah,al
      neg   ax                     ;convert from countdown
                                   ; remaining to elapsed
                                   ; count
      add   cs:[ReferenceCount],ax
;
; Restore the context and return.
;
      pop   cx
      pop   ax
      MPOPF
      ret

ReferenceZTimerOff endp

;********************************************************************
;* Routine called to report timing results.                         *
;********************************************************************

ZTimerReport     proc  near

      pushf
      push  ax
      push  bx
      push  cx
      push  dx
      push  si
      push  di
      push  ds
;
      push       cs    ;DOS functions require that DS point
      pop        ds    ; to text to be displayed on the screen
      assume     ds    :Code
;
; See if midnight or more than an hour passed during timing. If so,
; notify the user.
;
      mov   ax,[StartBIOSCountHigh]
      cmp   ax,[EndBIOSCountHigh]
      jz    CalcBIOSTime     ;hour count didn't change,
                             ; so everything's fine
      inc   ax
      cmp   ax,[EndBIOSCountHigh]
      jnz   TestTooLong ;midnight or two hour
                             ; boundaries passed, so the
                             ; results are no good
      mov   ax,[EndBIOSCountLow]
      cmp   ax,[StartBIOSCountLow]
      jb    CalcBIOSTime     ;a single hour boundary
                             ; passed--that's OK, so long as
                             ; the total time wasn't more
                             ; than an hour
```

```
;
; Over an hour elapsed or midnight passed during timing, which
; renders the results invalid. Notify the user. This misses the
; case where a multiple of 24 hours has passed, but we'll rely
; on the perspicacity of the user to detect that case.
;
TestTooLong:
      mov   ah,9
      mov   dx,offset TurnOverStr
      int   21h
      jmp   short ZTimerReportDone
;
; Convert the BIOS time to microseconds.
;
CalcBIOSTime:
      mov   ax,[EndBIOSCountLow]
      sub   ax,[StartBIOSCountLow]
      mov   dx,54925          ;number of microseconds each
                              ; BIOS count represents
      mul   dx
      mov   bx,ax             ;set aside BIOS count in
      mov   cx,dx             ; microseconds
;
; Convert timer count to microseconds.
;
      mov   ax,[EndTimedCount]
      mov   si,8381
      mul   si
      mov   si,10000
      div   si                ;* .8381 = * 8381 / 10000
;
; Add timer and BIOS counts together to get an overall time in
; microseconds.
;
      add   bx,ax
      adc   cx,0
;
; Subtract the timer overhead and save the result.
;
      mov   ax,[ReferenceCount]
      mov   si,8381           ;convert the reference count
      mul   si                ; to microseconds
      mov   si,10000
      div   si                ;* .8381 = * 8381 / 10000
      sub   bx,ax
      sbb   cx,0
      mov   [CurrentCountLow],bx
      mov   [CurrentCountHigh],cx
;
; Convert the result to an ASCII string by trial subtractions of
; powers of 10.
;
      mov   di,offset PowersOfTenEnd - offset PowersOfTen - 4
      mov   si,offset TimedCountStr
CTSNextDigit:
      mov   bl,'0'
CTSLoop:
      mov   ax,[CurrentCountLow]
      mov   dx,[CurrentCountHigh]
      sub   ax,PowersOfTen[di]
```

```
        sbb     dx,PowersOfTen[di+2]
        jc      CTSNextPowerDown
        inc     bl
        mov     [CurrentCountLow],ax
        mov     [CurrentCountHigh],dx
        jmp     CTSLoop
CTSNextPowerDown:
        mov     [si],bl
        inc     si
        sub     di,4
        jns     CTSNextDigit
;
;
; Print the results.
;
        mov     ah,9
        mov     dx,offset OutputStr
        int     21h
;
ZTimerReportDone:
        pop     ds
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        MPOPF
        ret

ZTimerReport    endp

Code    ends
        end
```

Moreover, because it uses an undocumented feature, the timer-stop approach could conceivably cause erratic 8253 operation, which could in turn seriously affect your computer's operation until the next reboot. In non-8253-compatible systems, I've observed not only wildly incorrect timing results, but also failure of a diskette drive to operate properly after the long-period Zen timer with **PS2** set to 0 has run, so be alert for signs of trouble if you do set **PS2** to 0.

Rebooting should clear up any timer-related problems of the sort described above. (This gives us another reason to reboot at the end of each code-timing session.) You should *immediately* reboot and set the **PS2** equate to 1 if you get erratic or obviously incorrect results with the long-period Zen timer when **PS2** is set to 0. If you want to set **PS2** to 0, it would be a good idea to time a few of the listings in this book with **PS2** set first to 1 and then to 0, to make sure that the results match. If they're consistently different, you should set **PS2** to 1.

While the the non-PS/2 version is more dangerous than the PS/2 version, it also produces more accurate results when it does work. If you have a non-PS/2 PC-compatible computer, the choice between the two timing approaches is yours.

If you do leave the **PS2** equate at 1 in Listing 3.5, you should repeat each code-timing run several times before relying on the results to be accurate to more than 54 ms, since variations may result from the possible lack of synchronization between the timer 0 count and the BIOS time-of-day count. In fact, it's a good idea to time code more than once no matter which version of the long-period Zen timer you're using, since interrupts, which must be enabled in order for the long-period timer to work properly, may occur at any time and can alter execution time substantially.

Finally, please note that the *precision* Zen timer works perfectly well on both PS/2 and non-PS/2 computers. The PS/2 and 8253 considerations we've just discussed apply *only* to the long-period Zen timer.

Example Use of the Long-Period Zen Timer

The long-period Zen timer has exactly the same calling interface as the precision Zen timer, and can be used in place of the precision Zen timer simply by linking it to the code to be timed in place of linking the precision timer code. Whenever the precision Zen timer informs you that the code being timed takes too long for the precision timer to handle, all you have to do is link in the long-period timer instead.

Listing 3.6 shows a test-bed program for the long-period Zen timer. While this program is similar to Listing 3.2, it's worth noting that Listing 3.6 waits for a few seconds before calling **ZTimerOn**, thereby allowing any pending keyboard interrupts to be processed. Since interrupts must be left on in order to time periods longer than 54 ms, the interrupts generated by keystrokes (including the upstroke of the Enter key press that starts the program)—or any other interrupts, for that matter—could incorrectly inflate the time recorded by the long-period Zen timer. In light of this, resist the temptation to type ahead, move the mouse, or the like while the long-period Zen timer is timing.

LISTING 3.6 LZTEST.ASM

```
; Program to measure performance of code that takes longer than
; 54 ms to execute. (LZTEST.ASM)
;
; Link with LZTIMER.ASM (Listing 3.5). LZTIME.BAT (Listing 3.7)
; can be used to assemble and link both files. Code to be
; measured must be in the file TESTCODE; Listing 3.8 shows
; a sample file (LST3-8.ASM) which should be named TESTCODE.
;
; By Michael Abrash
;
mystack    segment    para stack 'STACK'
     db    512 dup(?)
mystack    ends
;
Code  segment    para public 'CODE'
      assume     cs:Code, ds:Code
      extrn ZTimerOn:near, ZTimerOff:near, ZTimerReport:near
Start proc  near
      push  cs
      pop   ds     ;point DS to the code segment,
                   ; so data as well as code can easily
                   ; be included in TESTCODE
;
; Delay for 6-7 seconds, to let the Enter keystroke that started the
; program come back up.
;
      mov   ah,2ch
      int   21h               ;get the current time
      mov   bh,dh             ;set the current time aside
DelayLoop:
      mov   ah,2ch
      push  bx                ;preserve start time
      int   21h               ;get time
      pop   bx                ;retrieve start time
      cmp   dh,bh             ;is the new seconds count less than
                              ; the start seconds count?
      jnb   CheckDelayTime    ;no
      add   dh,60             ;yes, a minute must have turned over,
                              ; so add one minute
CheckDelayTime:
      sub   dh,bh             ;get time that's passed
      cmp   dh,7              ;has it been more than 6 seconds yet?
      jb    DelayLoop         ;not yet
;
      include    TESTCODE     ;code to be measured, including calls
                              ; to ZTimerOn and ZTimerOff
;
; Display the results.
;
      call  ZTimerReport
;
; Terminate the program.
;
      mov   ah,4ch
      int   21h
Start endp
Code  ends
      end   Start
```

As with the precision Zen timer, the program in Listing 3.6 is used by naming the file containing the code to be timed TESTCODE, then assembling both Listing 3.6 and Listing 3.5 with MASM or TASM and linking the two files together by way of the Microsoft or Borland linker. Listing 3.7 shows a batch file, named LZTIME.BAT, which does all of the above, generating and running the executable file LZTEST.EXE. LZTIME.BAT assumes that the file LZTIMER.ASM contains Listing 3.5 and the file LZTEST.ASM contains Listing 3.6.

LISTING 3.7 LZTIME.BAT

```
echo off
rem
rem *** Listing 3.7 ***
rem
rem ***************************************************************
rem * Batch file LZTIME.BAT, which builds and runs the            *
rem * long-period Zen timer program LZTEST.EXE to time the code   *
rem * named as the command-line parameter. Listing 3.5 must be    *
rem * named LZTIMER.ASM, and Listing 3.6 must be named            *
rem * LZTEST.ASM. To time the code in LST3-8, you'd type the      *
rem * DOS command:                                                *
rem *                                                             *
rem * lztime lst3-8                                               *
rem *                                                             *
rem * Note that MASM and LINK must be in the current directory or *
rem * on the current path in order for this batch file to work.   *
rem *                                                             *
rem * This batch file can be speeded up by assembling LZTIMER.ASM *
rem * once, then removing the lines:                              *
rem *                                                             *
rem * masm lztimer;                                               *
rem * if errorlevel 1 goto errorend                               *
rem *                                                             *
rem * from this file.                                             *
rem *                                                             *
rem * By Michael Abrash                                           *
rem ***************************************************************
rem
rem Make sure a file to test was specified.
rem
if not x%1==x goto ckexist
echo ***************************************************************
echo * Please specify a file to test.                              *
echo ***************************************************************
goto end
rem
rem Make sure the file exists.
rem
:ckexist
if exist %1 goto docopy
echo ***************************************************************
echo * The specified file, "%1," doesn't exist.                    *
echo ***************************************************************
goto end
rem
rem copy the file to measure to TESTCODE.
:docopy
copy %1 testcode
masm lztest;
if errorlevel 1 goto errorend
masm lztimer;
if errorlevel 1 goto errorend
link lztest+lztimer;
if errorlevel 1 goto errorend
lztest
goto end
:errorend
echo ***************************************************************
echo * An error occurred while building the long-period Zen timer. *
echo ***************************************************************
:end
```

Listing 3.8 shows sample code that can be timed with the test-bed program of Listing 3.6. Listing 3.8 measures the time required to execute 20,000 loads of AL from memory, a length of time too long for the precision Zen timer to handle on the 8088.

LISTING 3.8 LST3-8.ASM

```
;
; Measures the performance of 20,000 loads of AL from
; memory. (Use by renaming to TESTCODE, which is
; included by LZTEST.ASM (Listing 3.6). LZTIME.BAT
; (Listing 3.7) does this, along with all assembly
; and linking.)
;
; Note: takes about ten minutes to assemble on a slow PC if
;     you are using MASM
;
      jmp    Skip  ;jump around defined data
;
MemVar      db     ?
;
Skip:
;
; Start timing.
;
      call   ZTimerOn
;
      rept   20000
      mov    al,[MemVar]
      endm
;
; Stop timing.
;
      call   ZTimerOff
```

When LZTIME.BAT is run on a PC with the following command line (assuming the code in Listing 3.8 is the file LST3-8.ASM)

```
lztime lst3-8.asm
```

the result is 72,544 µs, or about 3.63 µs per load of AL from memory. This is just slightly longer than the time per load of AL measured by the precision Zen timer, as we would expect given that interrupts are left enabled by the long-period Zen timer. The extra fraction of a microsecond measured per **MOV** reflects the time required to execute the BIOS code that handles the 18.2 timer interrupts that occur each second.

Note that the command can take as much as 10 minutes to finish on a slow PC if you are using MASM, with most of that time spent assembling Listing 3.8. Why? Because MASM is notoriously slow at assembling **REPT** blocks, and the block in Listing 3.8 is repeated 20,000 times.

Using the Zen Timer from C

The Zen timer can be used to measure code performance when programming in C—but not right out of the box. As presented earlier, the timer is designed to be called from assembly language; some relatively minor modifications are required before the **ZTimerOn** (start timer), **ZTimerOff** (stop timer), and **ZTimerReport** (display timing results) routines can be called from C. There are two separate cases to be dealt with here: small code model and large; I'll tackle the simpler one, the small code model, first.

Altering the Zen timer for linking to a small code model C program involves the following steps: Change **ZTimerOn** to **_ZTimerOn**, change **ZTimerOff** to **_ZTimerOff**, change **ZTimerReport** to **_ZTimerReport**, and change **Code** to **_TEXT**. Figure 3.2 shows the line numbers and new states of all lines from Listing 3.1 that must be changed. These changes convert the code to use C-style external label names and the small model C code segment. (In C++, use the "C" specifier, as in

```
extern "C" ZTimerOn(void);
```

when declaring the timer routines **extern**, so that name-mangling doesn't occur, and the linker can find the routines' C-style names.)

That's all it takes; after doing this, you'll be able to use the Zen timer from C, as, for example, in:

```
ZTimerOn();
for (i=0, x=0; i<100; i++)
     x += i;
ZTimerOff();
ZTimerReport();
```

(I'm talking about the precision timer here. The long-period timer—Listing 3.5—requires the same modifications, but to different lines.)

```
Line #   Nw State
----     ---------------------
  47     _TEXT   segment word public 'CODE'
  48             assume     cs:_TEXT, ds:nothing
  49       public     _ZTimerOn, _ZTimerOff, _ZTimerReport
 140     _ZTimerOn     proc  near
 210     _ZTimerOn     endp
 216     _ZTimerOff proc     near
 296     _ZTimerOff endp
 372     _ZTimerReport proc  near
 384       assume     ds:_TEXT
 437     _ZTimerReport endp
 439     _TEXT   ends
```

These are the lines in Listing 3.1 that must be changed for use with small code model C, and the states of the lines after the changes are made.

Changes for use with small code model C.
Figure 3.2

Altering the Zen timer for use in C's large code model is a tad more complex, because in addition to the above changes, all functions, including the internal reference timing routines that are used to calculate overhead so it can be subtracted out, must be converted to far. Figure 3.3 shows the line numbers and new states of all lines from Listing 3.1 that must be changed in order to call the Zen timer from large code model C. Again, the line numbers are specific to the precision timer, but the long-period timer is very similar.

The full listings for the C-callable Zen timers are presented in Chapter K on the companion CD-ROM.

Watch Out for Optimizing Assemblers!

One important safety tip when modifying the Zen timer for use with large code model C code: Watch out for optimizing assemblers! TASM actually replaces

```
call        far ptr ReferenceZTimerOn
```

with

```
push        cs
call        near ptr ReferenceZTimerOn
```

(and likewise for **ReferenceZTimerOff**), which works because **ReferenceZTimerOn** is in the same segment as the calling code. This is normally a great optimization, being both smaller and faster than a far call. However, it's not so great for the Zen

```
Line #   New State
----     --------------------------
  47     PZTIMER_TEXT  segment word public 'CODE'
  48           assume    cs:PZTIMER_TEXT, ds:nothing
  49      public _ZTimerOn, _ZTimerOff, _ZTimerReport
 140     _ZTimerOn  proc far
 210     _ZTimerOn  endp
 216     _ZTimerOff proc far
 267        call  far ptr ReferenceZTimerOn
 268        call  far ptr ReferenceZTimerOff
 296     _ZTimerOff endp
 302     ReferenceZTimerOn   proc  far
 336     ReferenceZTimerOff proc   far
 372     _ZTimerReport proc  far
 384      assume ds:PZTIMER_TEXT
 437     _ZTimerReport endp
 439     PZTIMER_TEXT  ends
```

These are the lines in Listing 3.1 that must be changed for use with large code model C, and the states of the lines after the changes are made.

Changes for use with large code model C.
Figure 3.3

timer, because our purpose in calling the reference timing code is to determine exactly how much time is taken by overhead code—including the far calls to **ZTimerOn** and **ZTimerOff**! By converting the far calls to push/near call pairs within the Zen timer module, TASM makes it impossible to emulate exactly the overhead of the Zen timer, and makes timings slightly (about 16 cycles on a 386) less accurate.

What's the solution? Put the **NOSMART** directive at the start of the Zen timer code. This directive instructs TASM to turn off all optimizations, including converting far calls to push/near call pairs. By the way, there is, to the best of my knowledge, no such problem with MASM up through version 5.10A.

In my mind, the whole business of optimizing assemblers is a mixed blessing. In general, it's nice to have the assembler shortening jumps and selecting sign-extended forms of instructions for you. On the other hand, the benefits of tricks like substituting push/near call pairs for far calls are relatively small, and those tricks can get in the way when complete control is needed. Sure, complete control is needed very rarely, but when it is, optimizing assemblers can cause subtle problems; I discovered TASM's alteration of far calls only because I happened to view the code in the debugger, and you might want to do the same if you're using a recent version of MASM.

I've tested the changes shown in Figures 3.2 and 3.3 with TASM and Borland C++ 4.0, and also with the latest MASM and Microsoft C/C++ compiler.

Further Reading

For those of you who wish to pursue the mechanics of code measurement further, one good article about measuring code performance with the 8253 timer is "Programming Insight: High-Performance Software Analysis on the IBM PC," by Byron Sheppard, which appeared in the January, 1987 issue of *Byte.* For complete if somewhat cryptic information on the 8253 timer itself, I refer you to Intel's *Microsystem Components Handbook,* which is also a useful reference for a number of other PC components, including the 8259 Programmable Interrupt Controller and the 8237 DMA Controller. For details about the way the 8253 is used in the PC, as well as a great deal of additional information about the PC's hardware and BIOS resources, I suggest you consult IBM's series of technical reference manuals for the PC, XT, AT, Model 30, and microchannel computers, such as the Models 50, 60, and 80.

For our purposes, however, it's not critical that you understand exactly how the Zen timer works. All you really need to know is what the Zen timer can do and how to use it, and we've accomplished that in this chapter.

Armed with the Zen Timer, Onward and Upward

The Zen timer is not perfect. For one thing, the finest resolution to which it can measure an interval is at best about 1µs, a period of time in which a 66 MHz Pentium computer can execute as many as 132 instructions (although an 8088-based PC would be hard-pressed to manage two instructions in a microsecond). Another problem is that the timing code itself interferes with the state of the prefetch queue and processor cache at the start of the code being timed, because the timing code is not necessarily fetched and does not necessarily access memory in exactly the same time sequence as the code immediately preceding the code under measurement normally does. This prefetch effect can introduce as much as 3 to 4 µs of inaccuracy. Similarly, the state of the prefetch queue at the end of the code being timed affects how long the code that stops the timer takes to execute. Consequently, the Zen timer tends to be more accurate for longer code sequences, since the relative magnitude of the inaccuracy introduced by the Zen timer becomes less over longer periods.

Imperfections notwithstanding, the Zen timer is a good tool for exploring C code and x86 family assembly language, and it's a tool we'll use frequently for the remainder of this book.