

Chapter 2: Assume Nothing

When you're pushing the envelope in assembler, 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--so 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. (We'll discuss this code and various optimizations to it at length in Chapter 7.)

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% 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!

As diligent as the author had been, he had nonetheless committed a cardinal sin of 8088 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 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 8088, and there you have an important part of the Zen of assembler: 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 when programming the 8088, 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 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

One key to mastering the Zen of assembler is clearly 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 us). To be precise, the 8253 counts once every 838.1 nanoseconds. (A nanosecond is one-billionth of a second, and is abbreviated ns).

Listing 2-1 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."

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. The Zen timer will be our primary tool for the remainder of [The Zen of Assembly Language](#), 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 the timer, since we will use it heavily throughout [The Zen of Assembly Language](#).

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 seems 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 2-1. All three timers are driven by the system board's 14.31818 megahertz 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 2-1, timer 2 is the only timer with a programmable gate input in the PC; that is, timer 2 is the only timer which 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.

Finally, timer 0 is used to drive the system clock. As programmed by the BIOS at power-up, every 65,536 (64 K) 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 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 routine 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 6 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 2 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 1 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 2-1 supports.

In fact, the Zen timer shown in Listing 2-1 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. 54 ms may not seem like a very long time, but an 8088 can perform more than 1000 divides in 54 ms, and division is the single instruction 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, such as ATs, 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, even 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 Symdeb 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, Symdeb (and undoubtedly other debuggers as well) can be run from a remote terminal by running Mode to set up the serial port, then starting Symdeb and executing the command **=com1** or **=com2**.

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 assembler 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. 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 1 us, 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 10 us. In fact, we'll see in Chapter 5 why the Zen timer is actually most accurate in assessing code performance when timing intervals longer than about 100 us. At any rate, we're most interested using in 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 80386-based AT-compatible 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 this book on an XT-compatible computer, only to find that the computer wasn't quite IBM-compatible as regarded 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 2-2 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 2-3 shows some sample code to be timed. This listing measures the time required to execute 1000 loads of AL from the memory variable **MemVar**. Note that Listing 2-3 calls **ZTimerOn** to start timing, performs 1000 **mov** instructions in a row, and calls **ZTimerOff** to end timing. When Listing 2-2 is named TESTCODE and included by Listing 2-3, Listing 2-2 calls **ZTimerReport** to display the execution time after the code in Listing 2-

3 has been run.

It's worth noting that Listing 2-3 begins by jumping around the memory variables **MemVar**. This approach lets us avoid reproducing Listing 2-2 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 2-2 as TESTCODE. Listing 2-2 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 2-3 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 2-3 is used by naming it TESTCODE, assembling both Listing 2-2 (which includes TESTCODE) and Listing 2-1 with MASM, and linking the two resulting OBJ files together by way of the Microsoft Linker. Listing 2-4 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 2-4) assumes that the file PZTIMER.ASM contains Listing 2-1 and the file PZTEST.ASM contains Listing 2-2. 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 2-4. The same is true of Listing 2-7.)

Assuming that Listing 2-3 is named LST2-3 and Listing 2-4 is named PZTIME.BAT, the code in Listing 2-3 would be timed with the command:

```
pztime LST2-3
```

which performs all assembly and linking and reports the execution time of the code in Listing 2-3.

When the above command is executed on a PC, the time reported by the Zen timer is 3619 us, or about 3.62 us per load of AL from memory. (While the exact number is 3.619 us 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 8088 at the start of timing, for that last digit to have any significance.) Given the 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 Appendix A for official execution times.) Fear not, the Zen timer is right--**mov al,[MemVar]** really does take 17 cycles as used in Listing 2-3. Exactly why that is so is just what this book (and particularly the next three chapters) is all about.

In order to perform any of the timing tests in this book, enter Listing 2-1 and name it PZTIMER.ASM, enter Listing 2-2 and name it

PZTEST.ASM, and enter Listing 2-4 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 2-2, 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 2-5 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 allowing 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's 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 2-1 doesn't stop the timer; it latches a count, but the timer keeps running.) What to 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, 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 2-5 so that it can assemble to either use or not use the undocumented timer-stopping feature, as you please. The **PS2**

equates selects between the two modes of operation. If **PS2** is 1 (as it is in Listing 2-5), 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 problem, but doesn't work on all computers.

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 floppy 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 The Zen of Assembly Language 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 2-5, 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.

A SAMPLE 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 2-6 shows a test-bed program for the long-period Zen timer. While this program is similar to Listing 2-2, it's worth noting that Listing 2-6 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.

As with the precision Zen timer, the program in Listing 2-6 is used by naming the file containing the code to be timed TESTCODE, then assembling both Listing 2-6 and Listing 2-5 with MASM and linking the two files together by way of the Microsoft Linker. Listing 2-7 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 2-5 and the file LZTEST.ASM contains Listing 2- 6.

Listing 2-8 shows sample code that can be timed with the test-bed program of Listing 2-6. Listing 2-8 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.

When LZTIME.BAT is run on a PC with the following command line (assuming the code in Listing 2-8 is the file LST2-8):

lztime lst2-8

the result is 72,544 us, or about 3.63 us 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 multiply reflects the time required to execute the BIOS code that handles the 18.2 timer interrupts that occur each second.

Note that the above command takes about 10 minutes to finish on a PC, with most of that time spent assembling Listing 2-8. Why? Because MASM is notoriously slow at assembling **rept** blocks, and the block in Listing 2-8 is repeated 20000 times.

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, Models 50 and 60, and Model 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 1 us, a period of time in which a 25-MHz 80386 computer can execute as many as 12 instructions (although a 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 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 us of inaccuracy. (The nature of this problem will become more apparent when we discuss the prefetch queue.) 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 8088 assembly language, and it's a tool we'll use well for the remainder of this book. With the timer in hand, let's begin our trek toward the Zen of assembler, dispelling old assumptions and acquiring new knowledge along the way.