4 I/O Time

Admiral Grace Hopper, of COBOL fame, often handed out one foot lengths of wire during her lectures, as she observed that light travels about one foot in one nanosecond. Those wires (now collector's items, of course) helped her put electronic speeds on a human scale: "So *that's* what 70 ns means!"

It turns out that electronic signals in open wire travel at about one third the speed of light in vacuum, covering four inches every nanosecond. One of the wait states we discussed in the previous chapter reaches to the far wall of your domicile, a 16-bit I/O access stretches across the street, and an 8-bit bus access ends well down the block.

In this chapter, we'll add an 8-bit 82C54 timer chip to the Firmware Development, explore several more ISA bus timing issues, and introduce a handy BIOS timing facility you may not have met before.

Keep Admiral Hopper's wires in mind…

Where Does Time Come From?

Timing on PC class machines has always been a problem, because the facilities are only slightly above rudimentary. The system board includes three timers (either a real 82C54 or, more recently, a smidge of circuitry on an LSI support chip), but only one channel can generate an interrupt. Worse, that timer is tied into such diverse features as the BIOS time-of-day routine and the diskette motor turnoff delay, so tinkering with it can be hazardous to your program's operation.

The IBM PC AT added an MC146818A Real-Time Clock to keep track of the date and time while the system power is off. The chip (or its moral equivalent in LSI) can produce periodic interrupts on **`IRQ 8`** at rates ranging from 8.192 kHz down to 2 Hz. The BIOS implements an alarm clock function based on that interrupt, which will come in handy for this code.

While you can certainly use the PC's system-board timers in your code, they are best left for the normal BIOS services (unless, of course, your code doesn't use the normal BIOS services, in which case anything goes). I decided to add a separate 82C54 timer chip to support the Graphic LCD Interface you'll meet in Chapter 12, but in the meantime, we'll put the standard PC facilities to good use.

Schematic 1 shows the three chips needed for the timer circuit. These attach to the ISA bus data buffers and address decoding logic you built in Chapter 2, making the additional wiring for this circuitry fairly easy.

Photo 1 shows the Firmware Development Board with all the circuitry thus far.

The 82C54 timer, essentially identical to the non-CMOS 8254 on the original AT system board, has an 8 MHz clock rating. You must, however, pay close attention to the entire part number. A -2 suffix indicates a 10 MHz clock speed and a slightly faster bus interface. A -5 suffix brands it as a 5 MHz slowpoke that won't work in this circuit. No, the suffixes *don't* make sense; these chips were born back in the bad old days before rational "dash" numbers appeared.

Although It Would Be Nice to have a sensible clock frequency, I decided to use the 14.318 MHz signal from the ISA bus. Pardon the digression, but I must explain why that frequency is what it is.

The Original IBM PC's designers used an Intel 8284A clock generator on the system board to produce the 8088's CPU clock. The 8284 includes a crystal oscillator and produces a CPU clock at 1/3 of the crystal's frequency. Out on the Color Graphics Adapter, they needed a 14.318 MHz signal for the composite video

Schematic 1

The 82C54 timer chip shown here uses the address decoding and bus buffering circuitry presented in Chapter 3. The LS245 drives the ISA bus interrupt request lines, so make sure another device in your target system doesn't use the same signals!

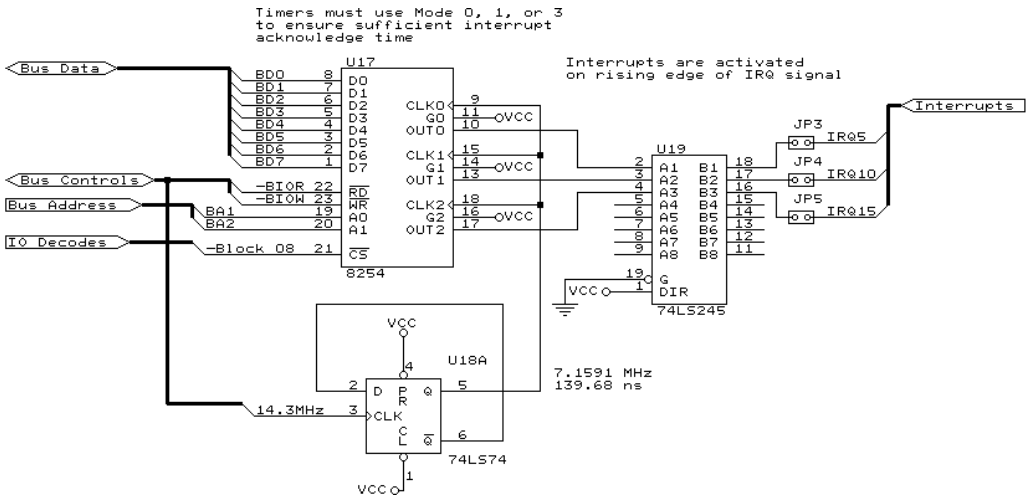
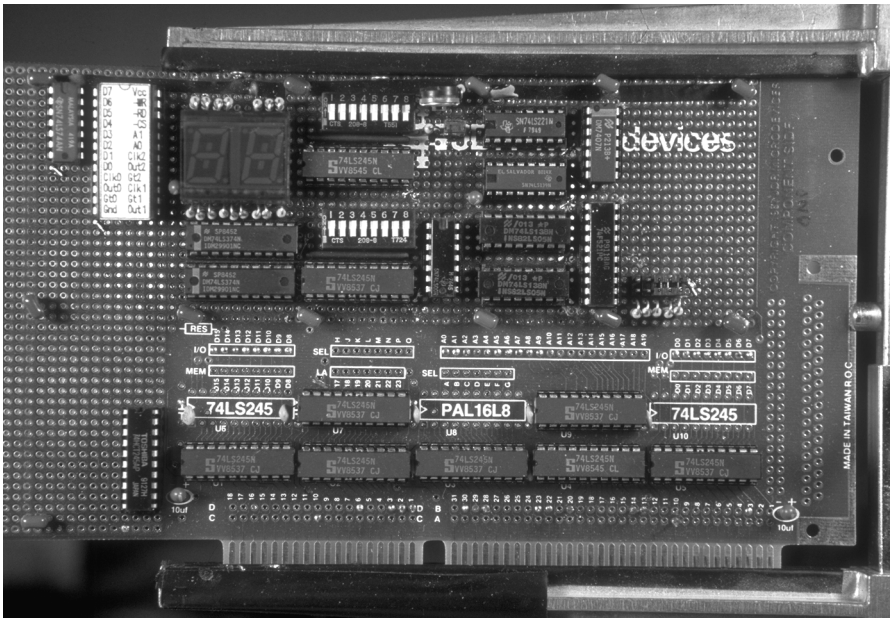


Photo 1
Adding an 82C54 timer, a bus buffer, and a few flipflops requires only three new DIPs. Comparing this with Photo 3 in Chapter 3 shows the additional circuitry to the left of the LEDs and bus buffers. Note the additional bypass capacitors on the ground plane.



signal that could drive a standard television. So… why not run the 8284 at 14.318 MHz and, thus, the 8088 CPU at 4.77 MHz?

The 8284 also produced a **PCLK** output at half the CPU clock rate, or 2.39 MHz. Although the Original PC's 8253-5 timer chip could handle that rate (it went all the way up to a blazing 2.6 MHz, despite the -5 suffix), the longest possible period between BIOS timer interrupts would then be 27.5 ms. That seemed a bit peppy, given that the PC's 8088 CPU could execute only 8000 instructions in that time.

So… they used a flipflop to divide **PCLK** by two, producing a 1.19 MHz clock, which the 8253 then divided by 64 K counts to generate an interrupt every 54.9 ms. That's why every PC in the universe has such a bizarre BIOS timer tick period.

The rest, of course, is history.

When the IBM PC AT came along, its designers could have boosted the BIOS tick rate to take advantage of the new-and-improved 80286 CPU's horsepower. By that time, however, far too many programs depended on that 54.9 ms rate. Only a whole new operating system can change the clock rate; anything less will be chained by historical necessity. And, as you might imagine, those new operating systems had better exhibit absolute backwards compatibility with existing programs.

But, even though the PC's Compatibility Barnacles limit the system board 82C54 to 1.19 MHz, we have no such restriction. The 82C54 has an 8 MHz clock rating, so half of an LS74 flipflop chops the 14.3 MHz bus signal down to 7.16 MHz.

Thus, each of our three timer outputs can produce interrupt periods from about 140 ns to 9.15 ms under firmware control. Admittedly, the faster rates aren't particularly useful, as even a firebreathing Pentium Pro can't accomplish much in 140 ns, but the improved timing resolution will turn out to be a Good Thing.

All three outputs drive interrupt request lines on the ISA bus connector through a LS245, which, in this application, serves as a simple buffer. I didn't include a way to disable these interrupts using software, because I assume that you're building this board for a particular reason and you'll hardwire the interrupts based on your requirements. You can, of course, install hardware jumpers that select different **IRQ** lines on the ISA bus, then disable them entirely by yanking the jumpers.

Because **IRQ 5** may be used by printer ports and sound boards, make sure your target system doesn't have any conflicts with the 82C54 outputs.

Figure 1
The Firmware Development Board now has three I/O devices: sixteen LEDs, a matching set of DIP switches, and an 82C54 timer. Because the address decode logic in Chapter 3 assumes 16-bit I/O accesses, each I/O port must appear at an even address. The 82C54 timer's four internal registers appear as the low-order byte in four consecutive 16-bit I/O ports and may be accessed by either 8-bit or 16-bit bus cycles.

Port	Read	Write
0308	Timer 0 Count/Status	Timer 0 Count
030A	Timer 1 Count/Status	Timer 1 Count
030C	Timer 2 Count/Status	Timer 2 Count
030E	nothing	Timer Control Register
031E	Switches (16-bit)	LED digits (16-bit)

I/O Addressing

The Firmware Development Board's address decoding logic requires 16-bit I/O operations directed to even addresses, but you don't need an I/O device for each and every data bit within a given port. For example, I wired the 82C54 to the low-order byte of the data bus and left the high-order bus byte disconnected. The hardware ignores the high byte during writes and the firmware must ignore it during reads.

The 82C54 has four internal addresses that are usually accessed as successive one-byte I/O ports. On this board, however, they must be located at successive *even* addresses, because the FDB's decoding logic cannot handle *byte* reads or writes to an *odd* address. As you can see in Schematic 1, the 82C54's **A0** and **A1** inputs come from the buffered address lines **BA1** and **BA2**, respectively.

Figure 1 shows the I/O port mapping for the devices we've added so far. The timer uses 8-bit I/O operations, while the LEDs and switches accept 16-bit operations.

You will recall that the only difference between 8-bit and 16-bit I/O operations is that the board activates **-IOCS16** to indicate that it can handle both bytes. If **-IOCS16** remains inactive, the system board breaks 16-bit operations into two 8-bit accesses by writing the low-order byte to the first address and the high-order byte to the next address.

The system board swaps the bytes around to put the proper bytes at the proper I/O locations, but the Firmware Development Board does not include the circuitry to handle its end of this dance. If you must support all possible I/O accesses, check the references for the details and prepare to spend a while designing that circuitry.

If the CPU is executing an 8-bit I/O operation, it ignores the **-IOCS16** signal and simply writes the byte to the specified address. Because the 82C54 connects to only the low-order byte of the ISA bus, it will respond correctly to either 16-bit or 8-bit I/O operations directed to even addresses. The high-order byte will be mush during 16-bit accesses, but in this case we don't care.

The only other difference is the bus timing, with 8-bit I/O operations requiring twice as many cycles. As it turns out, those extra cycles are critical to properly using the 82C54 on this board.

Timing from Both Sides

Because the 82C54 is the first nontrivial I/O device on the Firmware Development Board, we should explore some of the details that ensure it actually works. Even if you're in the firmware business, you must (at least) be able to interpret the hardware specs… perhaps to point out bugs that the hardware folks must fix?

The timing values and diagrams in this chapter come from Solari's *ISA & EISA Theory and Operation*, a weighty tome that replaces his earlier *AT Bus Design* book. Despite the fact that it costs about $90, it's essential if you're doing this sort of stuff.

Figure 2a shows the (considerably simplified) timings for a 16-bit ISA bus I/O read cycle and Figure 2b shows the corresponding timings for the 82C54's read operation. As is traditional in these hardware timing diagrams, the horizontal (time) axis is not drawn to scale and you can't compare intervals by eyeball.

Figure 2a
ISA bus timings for a 16-bit I/O read operation. Obviously, the time axis is not to scale!

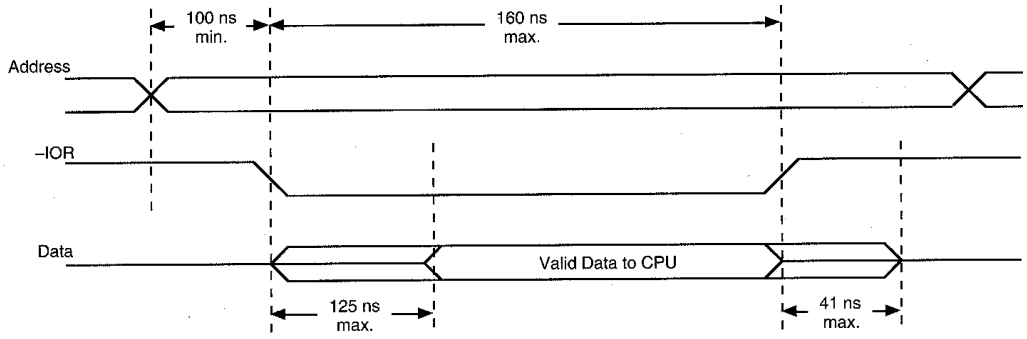


Figure 2b
82C54 read cycle timings, showing the 220 ns delay from the address lines and 120 ns delay from -RD to the first valid data output. This is barely within the limits shown above, but does not include the ISA bus buffer and address decoding times.

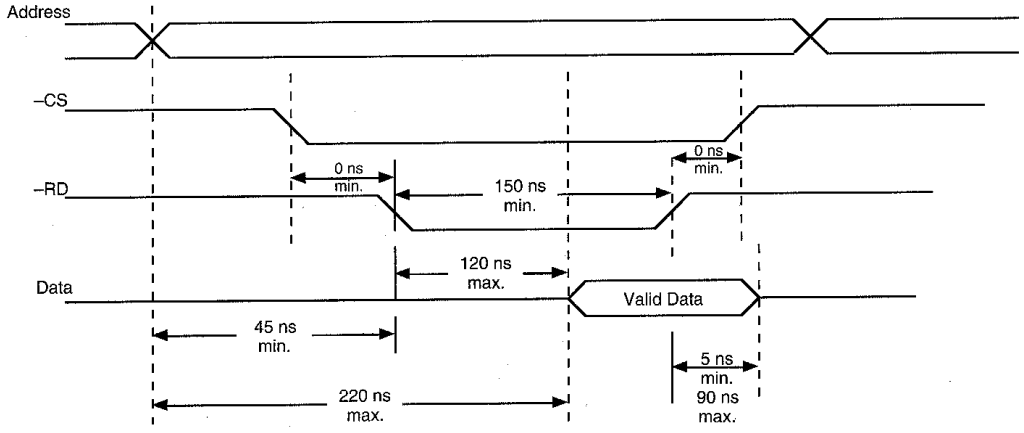
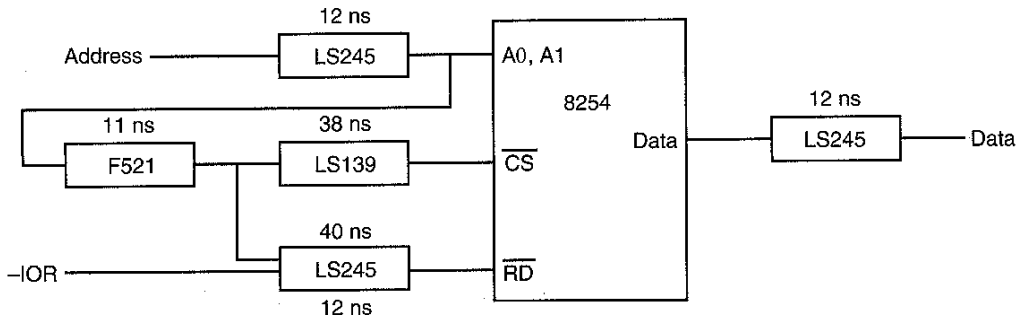


Figure 2c
This simplified Firmware Development Board schematic shows the delay through each IC involved in a ISA bus read operation. The two values shown for the LS245 in the Data path correspond to the DIR and Data inputs. You must add the delays shown in Figure 2a to the total delays along each path to find the total times.



According to Figure 2a, the ISA bus address settles at least 100 ns before **-IOR** goes active. At most 125 ns later, the data from the board must become valid and remain valid until well after **-IOR**'s trailing edge. However, the data drivers must turn off no more than 41 ns after **-IOR** goes inactive, thus preventing contention with the next cycle's signals.

As you can see in Figure 2b, however, the 82C54's output doesn't become valid until 220 ns *after* the address stabilizes or 120 ns *after* the beginning of the **-RD** pulse, whichever comes *later*. This may look close enough, but remember that the 82C54 isn't directly connected to the ISA bus.

Figure 2c shows the delays produced as signals pass through some of the key ICs supporting the 82C54. The LS245 buffers add a 12 ns delay to the address lines on the input side and another 12 ns on the data output, making valid data lag its address by about 244 ns. Comparing that with Figure 2A tells you that this setup just won't work.

That's precisely why wait states were invented. Recall, from Chapter 3, that a single wait state adds 120 ns to the duration of **-IOR**, ample time for the 82C54's data to arrive and settle down before the bus cycle ends.

There are several other paths though the logic that merit checking, such as the bus address to **-CS** path versus the **-IOR** to **-RD** path. In order to verify that the chip will work, you must analyze *all* the paths though the chips and add up their

timings. Hint: even if you work the *New York Times* crossword puzzle in ink, use a pencil for this job!

Obviously, if you must do a lot of this timing, figuring, and checking, you will depend on an automated timing analysis program that verifies every path in your schematic using built-in delay tables for each IC and interconnection. It's expensive, but… better you should find problems *before* the board goes into production, than suffer the slings and arrows of outrageous glitches.

Figure 3a
ISA bus timings for a 16-bit I/O write operation. Again, the time axis is unscaled.

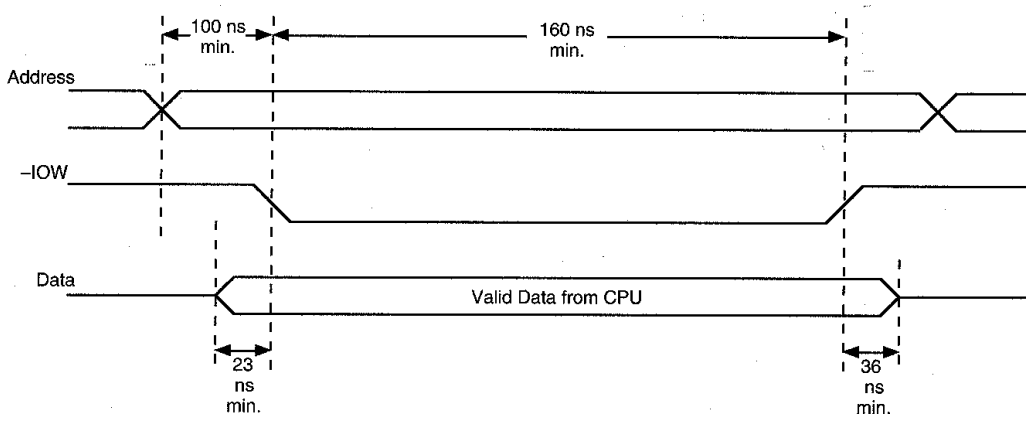


Figure 3b
82C54 write operation timings. This looks like it might work, if you ignore the buffer and decoding delays between the chip and the ISA bus signals.

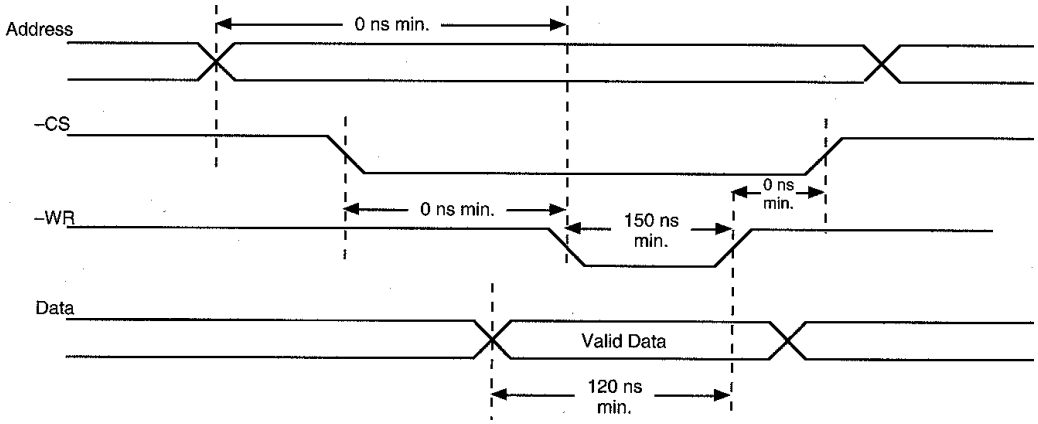
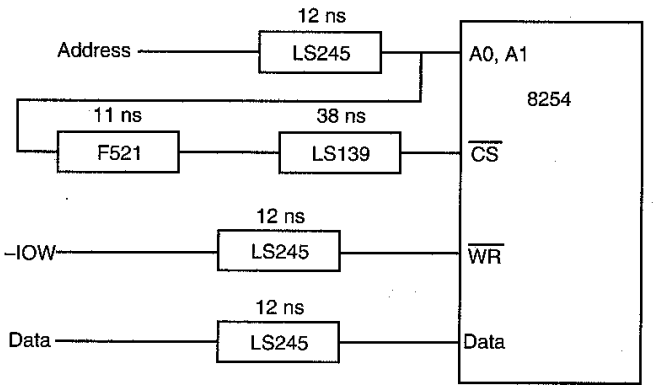


Figure 3c
These IC timings apply to ISA bus write operations. You must add the times shown in Figure 3a to the delays along each path to find the total times.



As an exercise, work though the write cycle diagrams shown in Figure 3 and see if an additional 40 feet of wait state will suffice here, too.

The simplified schematic in Figure 3c summarizes the circuit delays during a write cycle. Notice that **-IOW** gets there a bit faster than **-IOR** did during read cycles.

Now, here's the punch line. If you use 8-bit I/O operations, the ISA bus hardware inserts *three* extra wait states, with no extra effort on your part and no additional circuitry on your board. Basically, by picking the right I/O operation, we can tune the system to work correctly, even with this rather poky IC.

The down side: should you (or someone else, say, the poor soul maintaining your code after you've gone on to bigger & better things) substitute a 16-bit I/O operation, the access will fail. Worse, the timings seem pretty close, making it likely that some boards will work some of the time and others won't work most of the time. It's all a matter of the tolerances found on the system board and your chips.

Try to find *that* bug!

However, you'll find plenty of similarly sharp objects in this book, so I'll show how to use the hardware either way and trust you to do the right thing. This chapter's checkout program, `TimeTest.C`, can use either 16-bit or 8-bit I/O operations. You also have control of the FDB's wait state generator and can see just how well your hardware responds to various conditions. Oh, yes, comment your code, too.

Count the Ways...

Although the 82C54 registers and operations should be familiar to most of you, I'll provide a capsule summary to get us all to the same starting point. A complete "how to use the 82C54" exposition lies well beyond this book's charter… fortunately for us, we don't need quite *that* much detail.

The 82C54 contains three independent counter/timer channels, each with three pins: a **`Clock`** input, a **`Gate`** control input, and an **`Output`**. Unlike the counter/timers found in 8051 microcontrollers, these gizmos count *down*. Therefore, you just load them with the number of counts you want, rather than the two's complement of that value. If you prefer BCD to binary, you can have them count down from 9999 by decades instead of from FFFF in hexits. Honest.

Each 82C54 timer/counter channel can run in any one of six different modes, known, naturally enough, as Modes 0 through 5. Each mode uses the channel's three I/O pins in slightly different ways. You must match the channel's mode, register setup, and external hardware configuration to your application.

Mode 0 implements a simple counter: the firmware loads a delay value and the 82C54 decrements it by one count on the falling edge of each **`Clock`** pulse, as long as the **`Gate`** pin remains high. The channel's **`Output`** pin goes low when you load the counter and returns high when the counter hits zero. After that, even though the counter continues counting, the output pin won't change. This mode can generate precise delays for hardware gadgets, because the output can produce a single, clean, delayed edge with no additional hardware.

Mode 1 works the same way, except that a rising edge on the **`Gate`** input starts the count. In effect, this mode produces a precision delay starting from a hardware, rather than a firmware, trigger. Because I tied all the **`Gate`** inputs high on the Firmware Development Board, we can't try this mode without some rewiring.

Mode 2 divides the **`Clock`** input frequency by the programmed count value. It produces a single low **`Output`** pulse each time the count hits zero. The counter automatically reloads its initial value from an internal register and continues counting, giving you one pulse for every *n* **`Clock`** input pulses with **`Gate`** high. This comes in handy for dividing a clock frequency by a 16-bit value.

Mode 3, similar to Mode 2, produces a regular square-wave output with a period of *n* **`Clock`** cycles. Channel 0 of the system board's 82C54 uses this mode for the familiar 54.9 ms BIOS timer interrupt on **`IRQ 0`**. Note that the counter decrements by two, not one, on each **`Clock`** cycle. The **`Gate`** input can, thus, synchronize the **`Output`** phase to an external signal. I'll leave the **`Gate`** pins tied

Figure 4
Each Counter channel in the 82C54 uses configuration bits from the 8-bit Control Word written to the Control Register at address 030E. The Firmware Development Board, just like the PC itself, does not support all possible 82C54 modes and configurations!

The names for each Control Word bit are:

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

The SC bits select the Counter channel:

SC	Function
00	Counter 0
01	Counter 1
10	Counter 2
11	used for Read-Back commands

The RW bits specify the data I/O format for each counter. The counters are always 16 bits wide, but you can write or read a single byte in special situations:

RW	Function
00	used for Counter Latch commands
01	Read/write Counter MSB only
10	…LSB only
11	Read/write LSB followed by MSB

The M bits specify the Counter's operating mode, in the usual binary fashion, from 000 = Mode 0 to 101 = Mode 5. Specifying Mode 6 or 7 sets Mode 2 or 3, respectively.

The BCD bit selects the counting radix:

BCD	Radix
0	Binary, maximum count = FFFF hex
1	BCD, maximum count = 9999 decimal

high until we have something that demands synchronization, such as the Graphic LCD Interface circuitry.

Modes 4 and 5 are basically Modes 0 and 1 with a twist: the **Output** remains high until the counter hits zero, at which point it produces a single low pulse.

If you plan to use an 82C54 with anything other than a regular, periodic, essentially square **Clock** input, *read the data sheet carefully and pay attention to the clock specs*. It turns out that the 82C54 has several gotchas that become obvious only *after* you've designed the thing into a circuit and it behaves, well, strangely. See Chapter 13 for an example of how **Clock** input irregularities can affect an 82C54 counter.

Setting up an 82C54 counter channel is straightforward: just write an 8-bit *Control Word* to the Control Register, which resides at I/O address 030E on the Firmware Development Board. Figure 4 shows the Control Word bit layout. You must write one Control Word for each channel, then write one or two bytes to the channel's address to set the initial counter value. On the FDB, the three counters reside at addresses 0308, 030A, and 030C.

Hardware Checkout

Until we begin wiring up the Graphic LCD Interface hardware in Chapter 13, the 82C54's Mode 3 will suffice to exercise the hardware and illustrate the firmware techniques. Listing 1 shows the code required to set up all three counters for square-wave output.

Listing 1

This Micro-C code fragment shows how to initialize all three 82C54 channels to Mode 3. The #define macro eliminates a good deal of repetitive typing and ensures that you write the proper bytes in the correct sequence to the appropriate ports. Note that this code uses 8-bit I/O operations to ensure adequate bus timing.

```
#define LOADTIMERB(i,c,t) outp(I8254_BASE+6,c);   \
   outp(I8254_BASE+2*i,t & 0x00ff);               \
   outp(I8254_BASE+2*i,t >> 8);

putstr("Timer 0 = 1 ms square wave @ pin 10\n");
putstr("Timer 1 = 2 ms square wave @ pin 13\n");
putstr("Timer 2 = 3 ms square wave @ pin 17\n");

outp(SYNC_ADDR,0x01);                /* sync at start                    */

LOADTIMERB(0,0x0036,7159);           /*  1 ms @ 7.1591 MHz               */
LOADTIMERB(1,0x0076,14318);          /*  2 ms @ 7.1591 MHz               */
LOADTIMERB(2,0x00B6,21477);          /*  3 ms @ 7.1591 MHz               */

outp(SYNC_ADDR,0x00);
```

Figure 5
The 82C54 includes latches that capture both bytes of the counters, allowing two successive 8-bit read operations to return valid 16-bit data, regardless of the delay between the reads. A Read-Back Command Word written to the Control Register at address 030E activates the counter latches. Note that you can latch all three counters with one command. If bit 4 is zero, the first byte you read from the selected Counter channels will be the Status byte.

Bit	Function
7	Must be 1
6	Must be 1
5	0 = latch current Count of selected Counter channels
4	0 = latch current Status of selected Counter channels
3	1 = select Counter 2
2	1 = select Counter 1
1	1 = select Counter 0
0	Must be 0

Figure 6
The Status Byte read back from the counter channels gives you enough information to figure out what the counter is doing. Bits 5-0 echo the bits previously written to the counter's Control Word. In the code for this chapter, we use bit 7 to discover the state of the Output pin without connecting it to an input port.

Bit	Meaning
7	State of Output pin (0 = low, 1 = high)
6	Null Count (1 = not counting, 0 = counter loaded)
5	RW1
4	RW0
3	M2
2	M1
1	M0
0	BCD

The `#define` macro bottles up the three `outp()` functions that load the 82C54 Control Word and write both counter bytes for a single channel. If your code space gets unreasonably tight, for whatever strange reason, you can convert this macro to a function call.

After that code starts the timers, fire up your oscilloscope to verify the outputs. Should your test bench lack a scope, Listing 2 may be of more interest, because it measures and displays the results on your host system using the target PC's timers. If what you see on the host looks good, the hardware probably works OK… but, if it fails, prepare to cadge a scope from someone. You can use a frequency counter to

Listing 2

This routine computes and displays output frequencies by reading the status of each counter's Output pin. The code uses the BIOS alarm clock routine shown in Listing 3 to measure one second of real time while it checks for ounter Output pin transitions.

```
putstr("Timers should produce 1000, 500, and 333 counts/sec\n");
putstr("  +/- one or two counts for sampling error...\n");
putstr("LEDs display iteration counter in hex\n");
putstr("Uses 8-bit I/O operations\n");

outp(SYNC_ADDR,0x02);                  /* sync at start                    */
LOADTIMERB(0,0x0036,7159);             /*  1 ms @ 7.1591 MHz               */
LOADTIMERB(1,0x0076,14318);            /*  2 ms @ 7.1591 MHz               */
LOADTIMERB(2,0x00B6,21477);            /*  3 ms @ 7.1591 MHz               */

Counter = 0;
while (!chkch()) {

   Status[0] = Status[1] = Status[2] = 0;
   Edges[0] = Edges[1] = Edges[2] = 0;

   if (SetAlarm(0x000F,0x4240,&Alarm)) {          /* alarm in 1 second     */
      printf("Cannot use BIOS alarm function, code %02x\n",Alarm);
      break;
   }

   outp(SYNC_ADDR,0x01);
   while (!Alarm) {
      outp(I8254_BASE+6,0x00EE);                  /* latch all status bytes*/
      for (Timer=0; Timer<3; ++Timer) {
         PortValue = inp(I8254_BASE+2*Timer) & 0x0080;
         Edges[Timer] += (0 != (Status[Timer] ^ PortValue));
         Status[Timer] = PortValue;               /* save for next pass    */
      }
   }

   outp(SYNC_ADDR,0x00);
   printf("\r%5u %5u %5u counts/sec",
          Edges[0]/2,Edges[1]/2,Edges[2]/2);

   outpw(BOARD_BASE+LED_OFFSET,~ByteToSegs(Counter));
   ++Counter;
}
putstr("Counters continue to run...\n");
```

measure the pulses, although a scope will give you a better idea of the actual operation and reveal any marginal signal levels.

The 82C54 can return a *Status Byte* for each counter that indicates, among other things, whether the counter's **Output** pin is currently high or low. Figures 5 and 6 show the Control Word and Status Byte values you'll use. A short routine can easily count the number of **Output** flips per second to calculate and display the output frequency of each channel.

(Now, why do you write an 8-bit Control *Word* and read an 8-bit Status *Byte*? Because, just because… it didn't start with me!)

Listing 3
The BIOS INT 15h Function 83h provides an alarm clock for PC programs. This routine shows how to set the delay and specify the address of a byte that will change when the time expires. The code is mostly assembler within a Micro-C wrapper.

```
/*------------------------------------------------------------------------- */
/* Set up BIOS real-time alarm clock to delay interval in microseconds     */
/* Returns 0 if OK, error code if not (also sets *pbAlarm to return code)  */
/* High-order bit of *pbAlarm is set when delay time expires                */
/* Remember to split the microsecond count manually... 1 sec = 1000000 us  */
/* The asm{} section starts with BP pushed and loaded with SP               */

WORD SetAlarm(DelayHi,DelayLow,pbAlarm)
WORD DelayHi;
WORD DelayLow;
BYTE *pbAlarm;
{

DelayHi;                                /* keep compiler happy          */
DelayLow;
pbAlarm;

asm {
              MOV       BX,SP             aim at the stack
              MOV       CX,8[BX]          get delay high byte
              MOV       DX,6[BX]            ... low byte
              MOV       BX,4[BX]          get pointer to alarm byte
              MOV       AX,DS               ... set up segment
              MOV       ES,AX
              MOV       AX,#$8300         Set Event Wait Interval function
              INT       $15

              JC        ?seterr           C set on error
              XOR       AL,AL               ... clear return flag

?seterr       MOV       BX,SP             set up for return code
              MOV       BX,4[BX]
              MOV       ES:[BX],AL
              XOR       AH,AH             clear high byte of return value
}
}
```

Remember the MC146818A Real-Time Clock I mentioned earlier? Here's one place it comes in handy. BIOS Function `Int 0x15, AH=0x83` uses the periodic interrupt from that chip to set a delay; when the delay expires, the BIOS interrupt handler flips bit 7 in the byte of your choice. This relieves your code of the need to worry about interrupts: after you set the alarm, you simply test the target byte whenever it's convenient during your program's action loop.

The code in Listing 2 sets up the three 82C54 timers with periods of 1, 2, and 3 ms, then enters a loop that ends when it gets a byte from the serial port. Until then, it sets up a 1 s alarm and enters an inner loop that latches and reads back the current status for each channel. Although the 82C54 latches all three channels at once, you must fetch the three Status Bytes with three separate read operations.

Determining when each **`Output`** pin changes requires nothing more than XORing the current state with its value during the previous iteration. The `Status` array holds that information for each timer, while the count in the `Edges` array increments on each change.

Listing 3 shows the code that invokes the BIOS alarm clock. That function expects a four-byte delay value, measured in microseconds, and the `seg:off` address of a byte to change when the delay expires. Because the Micro-C compiler doesn't support `long` (four-byte) variables, I put the high and low halves of the 32-bit delay value into two 16-bit parameters.

When the BIOS function sets bit 7 of the `Alarm` byte after one second, the inner loop ends. Each **`Output`** cycle produces two counts in the `Edges` array, making the displayed frequency half that amount. The code sends the results through the serial port to your host system, sets the BIOS alarm again, and begins accumulating another set of counts.

You will see slight variations in the displayed totals as the program runs. The software counts, not being synchronized to the hardware's **`Output`** pulses, may miss or gain one count at the beginning and end of each second. However, if the totals differ from the correct values by more than few counts, look for something gone badly wrong.

Just for fun, I tossed in the `ByteToSegs()` function that converts a byte into a pair of hex digits on the seven-segment LED display on the Firmware Development Board. A line of code sends the iteration counter through that routine and gives a running count on the LEDs. As an exercise, adjust the code to display the digits backwards and upside down so they read correctly from the *top* of the board…

Release Notes

TimeTest.C includes a variety of tests I used to get the 82C54 running and verify the bus timings. You must have a scope to check some of the routines, but using just the self-testing code *should* get you on the air if you do careful wiring.

The circuitry shown in this chapter differs slightly from that in Chapter 13, where we'll adapt the 82C54 timer to drive a bitmapped, graphic LCD panel. You must check out the timer using the schematics and code from this chapter, then modify the wiring as shown in the back of the book.

Take stock of what you have working on your Firmware Development Board: I/O address decoding, a pair of LED digits, some DIP switches, and a trio of high-resolution hardware timers. So far, so good!