9 IDs and LCDs

Some years ago, the PC press pushed hardware serial numbers that could key software to a single, unique PC. The predicted benefits included killing software piracy, increasing profits, reducing prices, and making things right with the world.

Another prediction gone awry.

I suppose I've been a software pirate, at least by some definition. A while back, I had two PCs: a dying Model 80 from which I transferred programs to a new box. If these PCs had software that expected unique hardware serial numbers, I'd be out of luck, even though I had only one set of hands on one keyboard at any one time… and the Model 80 was in such bad shape it could no longer run the programs.

It got worse. The *first* new system didn't quite work, so I made *two* attempts at transferring the files and programs (well, two sets of attempts, to be precise). It's never been clear what the software transfer fee might be for that situation, but I'm sure paying it twice wouldn't have improved my disposition at all.

Nevertheless, in this chapter I'll describe how to give your Firmware Development Board a unique ID based on the Dallas DS2400 Silicon Serial Number chip. If you combine the (trivial) hardware with the BIOS extensions from Chapter 8, you can produce a system that won't even boot with a missing or incorrect serial number, let alone load and run a program from disk.

Just don't try to charge me for your code, OK?

We'll also add a small character LCD panel that displays status messages, without requiring a terminal or video monitor on your PC. If naught else, it can point out a wrong serial number or show everyone the unchanging value of your embedded PC's clock frequency. That last idea makes as least as much sense as the hardware behind your desktop PC's numeric display.

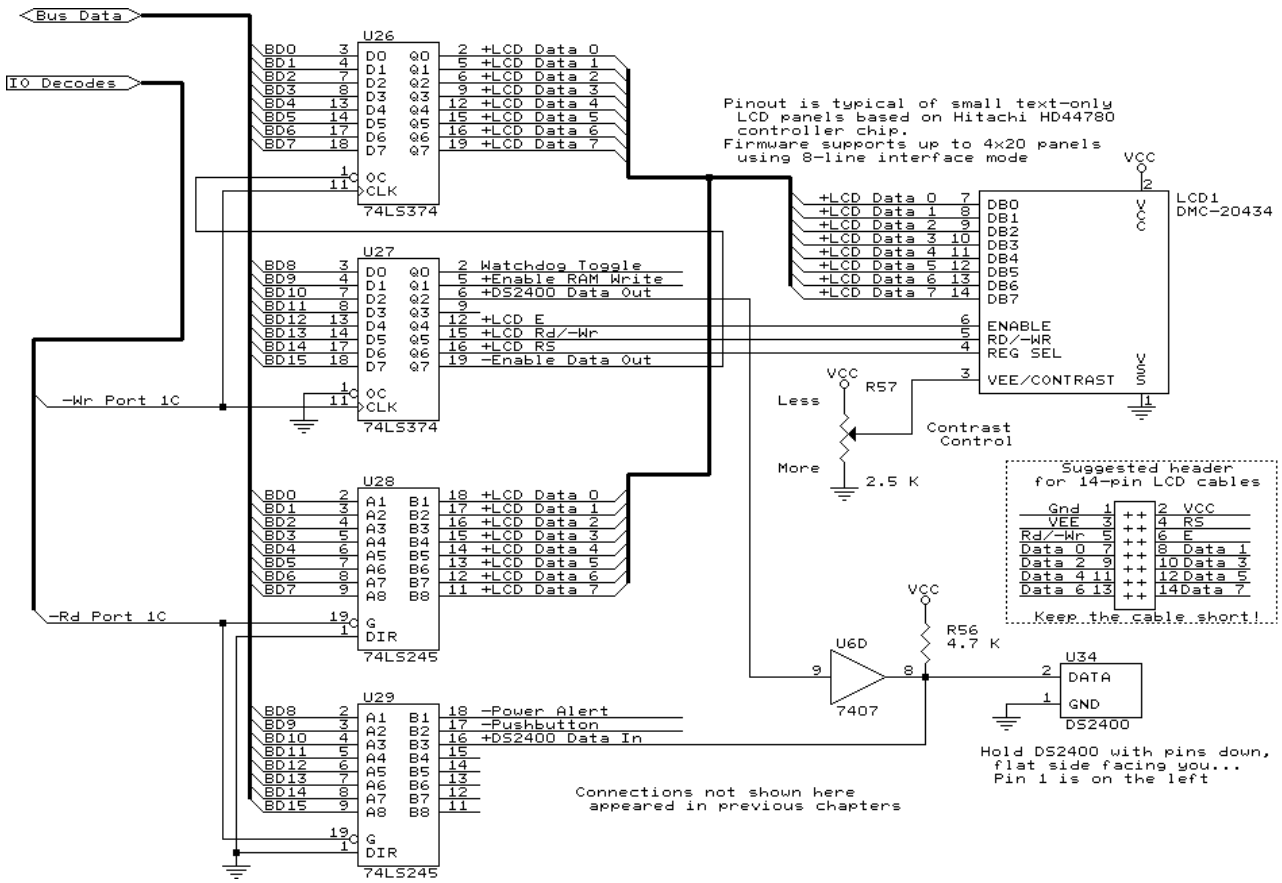
The two devices require completely different interfaces: the DS2400 uses precisely timed bit-serial communications and the LCD runs with a byte-wide parallel port. The Firmware Development Board's 82C54 timer provides timings independent of the CPU speed for the DS2400, while a dollop of firmware timing fixes the LCD.

Simple Circuitry

The DS2400 and character LCD interfaces require a few additional gates and buffers, as you can see in Schematic 1. Once again, I've omitted the ISA bus

Schematic 1

The DS2400 uses a single open collector output bit and one input bit, while the LCD panel requires a full byte of I/O data and several control lines. Firmware produces all of the required pulse timings through the ports, as these devices do not connect directly to the PC's ISA. The support circuitry for the buffer and driver ICs appeared in earlier chapters.



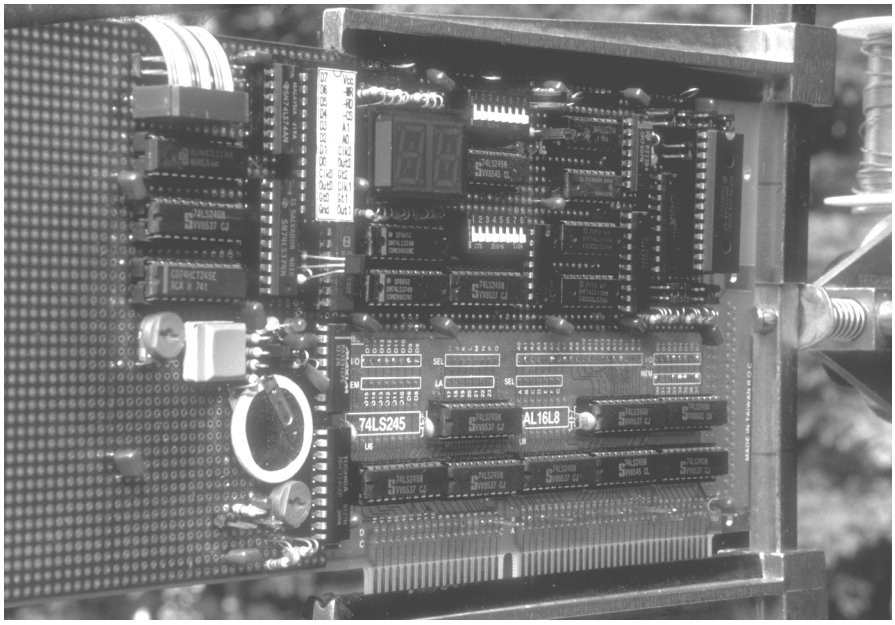
connections and support chips described in previous chapters. Check the Schematics appendix for the complete diagrams.

Photo 1 shows the Firmware Development Board after it gained this chapter's hardware. The DS2400 resembles a plastic transistor in a TO-92 case just below the 82C54. The character LCD panel, connected to the ribbon cable snaking from the top, hangs behind the board in this photo. Chapter 7 presented the circuitry for the lithium cell, MAX691, and the clump of discrete parts providing backup power for the static RAM chip.

The DS2400 claims to use a one wire interface. The chip must have a ground connection, of course, but a single signal wire provides bidirectional data and timing. Both the 7407 open-collector driver and the DS2400 can pull the signal wire to the logic low state, while the 4.7 kΩ pullup resistor supplies the logic high state when the driver transistors switch off.

As you might expect from a CMOS part, the DS2400 does not use TTL voltage levels. Its minimum V_{IH} rating of 3.0 V lies well above TTL's minimum 2.4 V

Photo 1
The DS2400 resembles a plastic transistor just below the 82C54. The LCD panel connects to the ribbon cable going off the top of the board. The coin-shaped lithium battery provides backup power for the static RAM chip in the upper-right corner.



V_{OH}. Keep that in mind if you're trying something truly bizarre… something that doesn't involve an open collector or open drain driver with a pullup resistor.

Although the LCD interface uses more wires and even sports an analog contrast control trimpot, it remains fairly simple. In fact, you can probably connect the panel's V_{EE} terminal to ground to get acceptable contrast without the trimpot. If your LCD panel requires a negative bias voltage, connect the pot between the +5 V and -5 V (or -12 V, in extreme cases) supplies. You should increase the pot to about 10 kΩ to reduce its power dissipation. Better yet, always follow your LCD's data sheet recommendations.

A perennial issue surrounds interfacing these little LCD panels directly to a microcontroller bus. The Hitachi HD44780 LCD controller bus interface matches the Motorola 6800 microprocessor and, thus, requires a bit of hocus pocus for an Intel-style ISA bus with entirely different control signals and timings. Rather than confront that problem, I elected to use simple port I/O and be done with it. If you want just an LCD panel on the ISA bus without the rest of the Firmware Development Board, the references in the Bibliography appendix should give you a good head start on your interface design. Pay attention to the timings!

Be careful while you wire up the LCD's LS245 and LS374 data lines. It's quite easy to solder one group "backwards" and create some truly baffling bugs. The LCD test firmware includes a counting sequence that should help pin down that problem, but exercising slightly more care than I did while soldering *my* board should avoid it entirely. Note that the LS374 driving the LCD's data lines must have its **`Output Enable`** line controlled by the high-order bit from the other LS374, as we must shut it off while our code reads data from the LCD.

With the hardware out of the way, on to the code…

Bidirectional Bit Banging

A laser trimmer personalizes each Dallas Semiconductor DS2400 Silicon Serial Number chip during production so that, unlike ordinary ICs, every one is unique. The data includes an 8-bit type number, a 48-bit serial number, and an 8-bit cyclic redundancy check (CRC) value to verify the data.

Most ICs transmit and receive synchronous data using at least two wires, one of which supplies a clock signal defining when the other has valid data. The DS2400 takes a minimalist approach to its interface by transmitting all information in pulse width modulated blips on a single wire. Using PWM reduces the hardware, while also requiring moderately complex firmware. Fortunately, firmware becomes

cheaper than hardware, after you've figured out how to make it work, and it costs nothing to reproduce. Sounds like a fair tradeoff.

Incidentally, the DS2400 mates nicely with one of the bidirectional I/O pins on an 8051 microcontroller. I used a 7407 open collector driver on the Firmware Development Board, but an 8051 system can get along with just a wire to the pin. You can't get much cheaper than that! (Well, you can, if you embed the serial number in the CPU die itself… but that's another topic entirely.)

Notice that the DS2400 has no power connection. A minuscule internal capacitor charges up while the signal pin is high and powers the CMOS circuitry while the pin is low. The timing specs ensure that the capacitor does not discharge during a transmission, which means you must *carefully* observe the minimum and maximum pulse width limits. Even if the CRC tells you when you get bad data, you should design the interface correctly, rather than depend on happenstance.

Figure 1 outlines a complete transaction: the PC resets the DS2400, sends a command word, and then clocks the type identifier, serial number, and CRC bits from the chip. The initial reset pulse discharges the internal capacitor and clears the DS2400's circuitry to start each transaction from the same, well-defined state.

I've found that the DS2400 sometimes requires *two* reset pulses the first time you access it after the power goes on. From what I can determine, the DS2400 circuitry jams into an invalid state when the power supply voltage rises slowly and requires a complete reset *before* it will respond properly to a normal reset pulse. In effect, you must completely discharge the internal capacitor and give the chip a solid reset. After that, it works just as the data sheet says it does.

In general, the PC will read the serial number only once during its power-on reset sequence. Leaving the signal line high at the end of the transaction eliminates the

Figure 1
Reading a DS2400 requires a reset pulse followed by 72 time slots for the bidirectional data flow. Each half of the reset pulse must be at least 480 µs long and the data time slots are between 60 and 120 µs each with a mandatory 1 µs delay after each bit. A complete transaction thus occupies less than 10 ms.

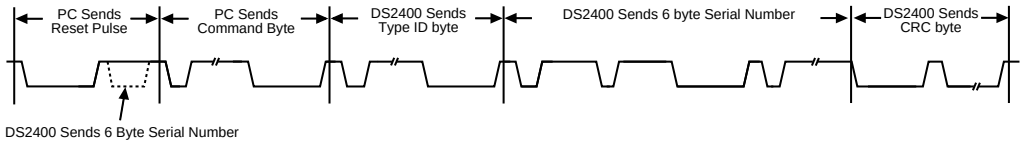


Figure 2
The DS2400 communications protocol requires close cooperation between the PC and the DS2400. The PC determines when each time slot begins and controls the duration of transmitted data, while the DS2400 sets the duration of the pulses it sends in response to the PC's prompting. All times shown in this figure are in microseconds.

Figure 2a
The second half of the reset pulse allows the DS2400 to respond with a "presence detect" pulse: the DS2400 pulls the data line low shortly after the PC raises it. Although it's not completely unambiguous, this behavior indicates that *something* is on the wire.

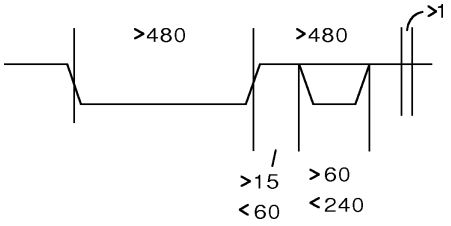


Figure 2b and 2c
The PC pulls the data line low at the start of each transmit time slot. If it is transmitting a 1 bit, it must release the line within 15 µs. The line must remain low for the entire duration of the 60-120 µs time slot to transmit a 0 bit.

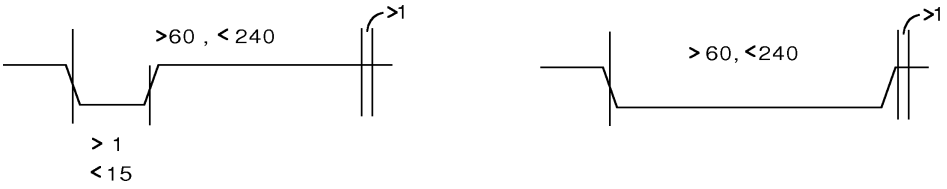
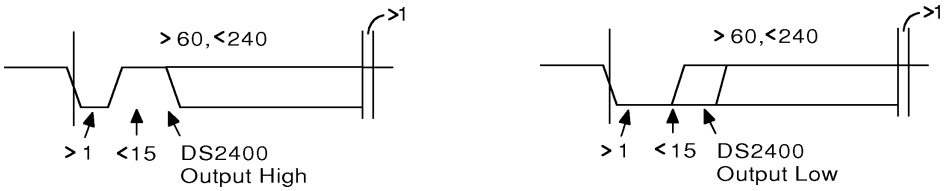


Figure 2d and 2e
The PC also pulls the data line low at the start of each receive time slot, but must release it within a few microseconds to allow the DS2400 to control the pulse width. The DS2400 will release the line immediately to transmit a 1 bit or hold it down for at least 15 µs to transmit a 0 bit. In either case, the PC must sample the DS2400 within 15 µs after the start of the time slot.



few milliwatts dissipated in the pullup resistor, which is surely orders of magnitude more than the DS2400 idle power that's left unspecified in its data sheet.

Figure 2 details the five waveforms used by the DS2400 interface. The reset pulse sports the easiest requirement with only two minimum times: no less than 480 µs for both the high and low parts. The transmit and receive pulses, however, last only a few microseconds and the PC must accurately measure the received pulses to decide what data it's getting.

When a project's timing specs are denominated in microseconds and you're charged with writing the pulse measurement firmware, get ready for some serious thinking. The first issue must be whether the task is feasible at all. After settling that issue, you can decide the best way to implement it.

Sometimes you really *will* be asked to do the impossible. I recall discussing a project proposal with an engineer who'd been charged with sampling an analog voltage at something like 20 megasamples/sec, then storing 16-bit samples direct to disk through a custom ISA bus board. If you've been paying attention so far, you know that can't *possibly* work (why?). My advice to him: present the ISA bus limits to his managers, then hide behind a bush if the project continued in defiance of reality.

Fortunately, the specs changed when he explained how the ISA bus worked and what those limitations meant to the project. Sometimes you don't get off that lightly, but it's certainly worth the effort.

Hardware Timing

When you use the DS2400 in an 8051 microcontroller system, you can determine pulse widths by simply counting instruction cycles in a timing loop. Not so with embedded PCs! In fact, *x*86 CPUs encourage programming as an experimental science: figure out how long a loop should last, write it, measure the results, then tweak the loop count to make the answer come out right. With any luck, a few iterations will converge on the desired value.

Your luck will go the other way, though, when you change PC board vendors, move the program from RAM to EPROM, or just twiddle a few BIOS setup values, as the different timings can clobber your precisely tuned loop. Most of the routine tricks in 8051 code just don't apply here. In fact, delay loops fall in the "better to avoid if at all possible" category, although I'll show you a reasonable use for one in the LCD interface code.

I included an 82C54 timer chip on the Firmware Development Board for just such applications: it can measure time intervals with 139 ns resolution, up to a total

Listing 1
The PC and DS2400 communicate by means of pulse width modulated signals. This routine produces low and high pulses without depending on the CPU clock or timing loops. It also enforces a minimum 1 µs idle time between each pair of pulses. Because read and write signals are identical, this code samples the data line shortly after the rising edge; the caller decides whether the bit is useful or not.

```
int PulseDS2400(LowTime,HighTime)
WORD LowTime;
WORD HighTime;
{

WORD TestValue;
int RetValue;

   TestValue;                          /* keep the compiler happy...         */
   HighTime;
   LowTime;

asm {
   CLI                                 ; need precise timing for first pulse

   MOV   DX,#SYNC_ADDR_A               ; show sync on printer port
   IN    AL,DX
   OR    AL,#$02
   OUT   DX,AL

   MOV   DX,#I8254_BASE_A+6            ; set mode
   MOV   AL,#DS_MODE
   OUT   DX,AL

   MOV   DX,#I8254_BASE_A+(DS_CTR*2)   ; set low time
   MOV   AX,6[BP]
   OUT   DX,AL
   XCHG  AH,AL
   OUT   DX,AL

   MOV   DX,#CTLS_ADDR_A               ; set DS2400 data bit low
   MOV   AX,#0
   OUT   DX,AX

   CALL  WaitTimer

   MOV   DX,#I8254_BASE_A+(DS_CTR*2)   ; set read delay interval
   MOV   AX,#TIME_RD
   OUT   DX,AL
   XCHG  AH,AL
   OUT   DX,AL

   MOV   DX,#CTLS_ADDR_A               ; set DS2400 data bit high = floating
   MOV   AX,#DS2400OUT_A
   OUT   DX,AX

   CALL  WaitTimer

   MOV   DX,#STAT_ADDR_A               ; read output from DS2400
   IN    AX,DX
   AND   AX,#DS2400IN_A                ; isolate the data bit
   MOV   -2[BP],AX
```

Listing continues on next page

Listing continued from previous page

```
MOV   DX,#I8254_BASE_A+(DS_CTR*2)      ; set rest of high time
   MOV   AX,4[BP]
   SUB   AX,#TIME_RD
   ADD   AX,#7                         ;  ... plus mandatory recovery time
   OUT   DX,AL
   XCHG  AH,AL
   OUT   DX,AL

   STI                                 ; re-enable interrupts

   CALL  WaitTimer

   JMP   PulseDone

;--- wait for timer interval to expire

WaitTimer EQU *
   MOV   DX,#I8254_BASE_A+6            ; latch & read back counter 0 status
   MOV   AL,#DS_LATCH
   OUT   DX,AL

   MOV   DX,#I8254_BASE_A+(DS_CTR*2)   ; read counter status
   IN    AL,DX

   TEST  AL,#DS_PIN                    ; wait for pin to go high again
   JZ    WaitTimer
   RET

PulseDone EQU *

   MOV   DX,#SYNC_ADDR_A               ; clear sync pulse
   IN    AL,DX
   XOR   AL,#$02
   OUT   DX,AL
}

   return !!RetValue;                              /* return Boolean value   */
}
```

duration of 9.1 ms (64 K counts × 139 ns). Because it uses the 14.3 MHz ISA bus oscillator signal, its time intervals do not depend on the CPU clock frequency or system board hardware and you'll get the same results every time.

Because the read and write pulses resemble each other so closely, I combined their code into the routine shown in Listing 1. `PulseDS2400()`, written using 8086 assembler within a Micro-C wrapper, accesses its input arguments and local variables on the stack.

Although the 82C54 timers can produce hardware interrupts, I use simple software polling to detect the end of each pulse. Remember that hardware interrupt handlers must start by saving the CPU registers and setting up the segment registers. When you are dealing with microsecond pulses, that can take far too long. In fact, you must disable external interrupts before starting the precision timing code.

(Yes, there are sneaky ways to streamline hardware interrupt handlers under very special, carefully controlled, exotic conditions. The key is knowing exactly when the handler will be active, then preloading the registers for it. For an example of this, check my *Firmware Furnace* column in *Circuit Cellar INK* Issue 3. The DS2400 didn't call for such gymnastics, so I could get away with polling. *Whew!*)

The first timing loop meters out the low part of the pulse. If the mainline code calls for a 0 bit, this loop will last for 90 µs. When the mainline code reads data from the DS2400, the pulse ends after 5 µs. `PulseDS2400()` also sets a sync bit in the parallel printer port to mark its entry into this code; if that bit never goes low again, you know you have hardware problems on the Firmware Development Board. Plug the LED-and-switch box (from Chapter 1) into your parallel port and watch the firmware in action for some reassurance or diagnostic tracing.

The 82C54 **`Output`** pin goes low during the pulse and returns high at the end. The delay loops load the timer count registers to start the interval, then detect the ending by polling the output status bit. Polling involves writing a latch command to the 82C54 and reading the status register, making it slightly slower than reading a port input bit. Any current PC runs fast enough to adds only a few microseconds of delay at the usual ISA bus speeds. If execution time became critical, you could route the timer output bit back through an input port and poll it directly.

After the low pulse ends, the code writes a 1 to the DS2400 output bit, thus allowing the signal line to float high, and sets a 2 µs delay. When that interval expires, the code samples the DS2400 and records the input value. During a read operation, the DS2400 controls the signal line and the sampled value represents the incoming bit. For a bit written from the PC, the sampled value contains no information and will be ignored by the caller.

Following the sample, the code loads the timer with the rest of the required high time, plus the mandatory 1 µs idle time, and waits for it to expire. Because the DS2400 output is high, any additional delays have no effect and the code can enable interrupts. The DS2400 spec sets the minimum idle time, but does *not* specify a maximum. Even a protracted delay at this point will not cause a data error, because the capacitor inside the DS2400 remains charged from the input line.

The sample code includes routines that use `PulseDS2400()` to write the 8-bit command word (always 0F for the DS2400) and read back all 64 bits. It's quite straightforward, except for one little, teeny, tiny, essential detail: you must transmit and receive bytes starting with low-order bit. Believe it or not, that critical fact appears nowhere in the DS2400 data sheets on my shelf!

Checking Your ID

After you read the contents of a DS2400, you should verify the data by computing the data's CRC value. You may, if you like, assume that nothing can possibly go wrnog (*oops!*) with such a simple interface, but I recommend checking the CRC just to be sure. The CRC byte's protection includes the Type ID byte, which is always 01 for DS2400, and all six bytes of the serial number.

The DS2400 data sheet presents the CRC algorithm in 8051 assembly language. While just transliterating the code into *x*86 assembler certainly looks tempting, remember that the two CPUs set their ALU status flags differently for what seem to be the same instructions. A simple line-for-line conversion will produce the wrong answer. Listing 2 shows how the code I wrote for the CRC calculation takes advantage of the *x*86 CPU's 16-bit registers.

Listing 2

The DS2400 includes an 8-bit CRC computed over its Type ID and serial number data. If the byte you compute using this algorithm matches the one read from the DS2400, the odds are pretty good that no errors occurred during the transfer. This code is based on the 8-bit algorithm shown in the DS2400 data sheet, while using 16-bit registers to shuffle the bits. Remember that the 8051 and 80x86 CPUs set their ALU flags differently.

```
int CRCDS2400(pData)
BYTE *pData;
{
int CRC;
              pData = pData;               /* keep compiler happy */
              CRC = 0;                     /* clear both bytes */

asm {
              MOV       AH,#0              ; set up initial CRC
              MOV       SI,4[BP]           ; set up data pointer
              MOV       CX,#7              ; set up byte counter
OuterLoop EQU *
              PUSH      CX                 ; save byte counter
              MOV       CX,#8              ; set up bit counter
              MOV       BL,[SI]            ; fetch data byte
CRCLoop EQU   *
              MOV       AL,BL              ; set up data byte
              XOR       AL,AH              ; combine low bit with old CRC
              ROR       AX,1               ;  ... put result into high bit of CRC
              JNC       Zero               ;  ... and into carry flag, too
              XOR       AH,#$0C            ; stir in constant (pre-shifted!)
Zero          EQU       *
              SHR       BL,1               ; set up next data bit
              LOOP      CRCLoop            ; repeat for all bits
              INC       SI                 ; aim at next byte
              POP       CX                 ; fetch byte counter
              LOOP      OuterLoop          ; and repeat for each byte
              MOV       -2[BP],AH          ; set up return value
}
              return CRC;
}
```

Just when you think you've got everything right, though, there's always one more gotcha. I recall a discussion with a group who simply could not get the right CRC value. They used the Official Dallas Algorithm on an 8051 CPU (by Dallas, no less), eliminating any possible translation error. Nevertheless, the CRC bytes they read from their DS2400 chips simply did not match their computed values, no matter how often they checked their code and hardware.

After considerable headscratching and many calls to Dallas' tech support, they found the problem. It seems there was a little glitch in the manufacturing code driving the laser trimmer on the DS2400 production line. It burned the unique serial number into each chip, then computed and burned *the wrong CRC value!*

Talk about hard-to-find bugs!

Needless to say, that was fixed PDQ and all current DS2400 chips work fine. However, should you have a stash of old DS2400s lying around, check them *very* carefully just to be sure… they'll make excellent Show and Tell items.

I've said it before and I'll say it again: hell hath no fury like that of an unjustified assumption.

The `SerNum.C` program reads back the DS2400 data, calculates the expected checksum, and displays everything. To judge from the DS2400s I have on hand, Dallas produced about 400,000 parts before I bought my stash, at least if they're assigning the serial numbers in roughly sequential order.

If your circuit doesn't work the first time, `SerNum` can also write the command byte in a tight loop and scoping the DS2400 data pin should quickly reveal the problem. The interface circuit is simple enough that a solder splash ought to be the extent of your troubles. But, you never can tell…

LCDs Redux

The Firmware Development Board sports a pair of seven-segment LED digits that we've used to report error conditions and display status information. The messages tend to be cryptic, but suffice for our simple purposes. If you must dress your system up for company, though, you probably want a few lines of legible text output. The small LCD panels you see on microcontroller projects can be equally handy on embedded PCs. Making them work is a simple matter of firmware.

I reviewed the operating principles of small LCD character panels in *Circuit Cellar INK* Issue 8. Refer back to that column for details of how they work and what all

the interface lines mean. For now, I'll concentrate on the specifics of the Firmware Development Board's hardware, rather than cover that ground once again.

The common denominator of all these LCD panels lies in the Hitachi HD44780 LCD controller that provides their (admittedly limited) intelligence. Most of the small character LCD panels you'll find use this controller or one similar to it, but you should get the data sheets for your panel just in case you catch an oddball.

The LCDTest.C program can handle any display based on the HD44780 controller, up to the chip's 80-character maximum size. I preset the code for 4 rows of 20 characters, using a pair of constants that define the number of visible rows and columns. You can tweak those two lines, recompile, and have your new display up and running, pretty much regardless of its size or shape.

As shown in Schematic 1, the LCD panel interface uses port I/O, rather than a direct PC ISA bus connection. Although a bit-banged port interface runs much slower than a direct bus hookup, the firmware can still update the entire display faster than those liquid crystals can respond. That's fast enough.

Listing 3 shows the few lines of code that write a byte to the display. The LCD's **Enable** line latches data and commands into the controller, so the code must write

Listing 3
This routine writes a single byte to the character LCD panel. The Mode parameter determines whether the byte is a character for the display buffer or a command to the LCD controller. After writing data to the port, the code calls BlipEnable to pulse the LCD's Enable input high and low to strobe the byte into the LCD controller.

```
LCDSendByte(Data,Mode)
WORD Data;
WORD Mode;
{
WORD PortValue;

   if (!(Mode & SENDFORCE)) {         /* want to force without wait?       */
    LCDWaitBusy();
   }

   PortValue = (Mode & SENDDATA) ? LCD_RS : 0;    /* cmd/data, -WR, enabled*/
   PortValue |= Data;                             /* combine data          */
   outpw(CTLS_ADDR,PortValue);

   BlipEnable(PortValue);

   PortValue |= LCD_RW;                           /* disable -Wr line      */
   outpw(CTLS_ADDR,PortValue);

}
```

the 16-bit port three times. First it sets up the data and control lines with **Enable** low, then sets **Enable** high, and, finally, lowers **Enable** to finish the operation.

Because we cannot read the outgoing data back into the CPU, the code stores it in the `PortValue` variable between writes. This is a classic hardware *vs.* software tradeoff: implementing the hardware with TTL gates means that additional functions require more chips and occupy more board space. Had I built the Firmware Development Board using PALs or gate arrays, the additional hardware would be essentially free. Think of this as practice for real life.

Maintaining a copy of the most recent output data is a simple trick that suffices for now. I'll describe the real world problems that arise from this sort of situation later on in this chapter.

Listing 4 presents `LCDReadByte()`, which reads status and data from the LCD controller. Two other routines call this function to check the controller's **Busy** status flag and to read the display buffer during vertical scrolling. Although both functions can be simulated, with some difficulty, in firmware, I used Steve Ciarcia's favorite programming language: Solder. Sometimes, if there's just room for just one more chip, you must choose which one provides the most benefit. Ah, tradeoffs.

Despite my tirade about software delay loops, you'll find one buried in the heart of `LCDReadByte()`, just after the code raises the **Enable** line. Some LCD panels I've used require more time than expected to drive data back to the Firmware Development Board. The value I settled on creates a 150 µs pause on an 80 MHz '486DX2, about four times the delay my worst panel expects. If you get erratic vertical scrolling on a faster CPU, you know what the problem is and how to fix it.

This may be related to the ribbon cable layout that comes naturally to these panels, although the glitch doesn't appear on my scope (not that glitches ever do, of course). You might want to experiment with different cable layouts to see if bracketing the **Enable** wire or the data lines with pairs of quiet ground lines helps. Trust me on this: if you haven't used firmware to fix hardware before, this is a good time to get started…

Why not use one of the 82C54 timers on the Firmware Development Board? Well, if we had the services of a realtime operating system at our disposal, I'd have no qualms about invoking a programmed delay. Unfortunately, we don't, and I have plans for this timer later on that preclude dedicating it to a character LCD. In real life, you might also find the overhead involved in setting up a 150 µs delay through an operating system might take longer than the delay itself. Measure first, then decide what to do.

Listing 4
This routine reads a byte from the LCD controller; the Mode parameter determines whether it comes from the display buffer or the controller logic. The loop in the asm{} section compensates for delays caused by cable capacitance and must be tuned for each system… there is no need for a precise measurement, just a minimum delay.

```
int LCDReadByte(Mode)
WORD Mode;
{
WORD Data;
WORD PortValue;

   outp(SYNC_ADDR,inp(SYNC_ADDR) | 0x02);         /* scope sync            */

   PortValue = (SENDDATA == Mode) ? LCD_RS : 0;   /* read data or addr?    */
   PortValue |= LCD_RW + LCD_NODRV;   /* LCD read and disable drivers      */
   outpw(CTLS_ADDR,PortValue);

   PortValue |= LCD_E;                            /* strobe data from LCD  */
   outpw(CTLS_ADDR,PortValue);

asm {
      MOV   CX,#RD_DELAY              ; delay to allow data settling time
?lp1  LOOP  ?lp1
}

   Data = inpw(STAT_ADDR) & 0x00FF;   /* fetch and isolate LCD data        */

   PortValue &= ~LCD_E;                           /* remove LCD strobe     */
   outpw(CTLS_ADDR,PortValue);

   outp(SYNC_ADDR,inp(SYNC_ADDR) & ~0x02);        /* sync off              */

   return Data;
}
```

The remaining LCD code adds cursor positioning, string output, simple scrolling and some control character processing. It's straightforward code that you should review to see how it's done. The LCDTest.C program provides several test loops with scope trigger outputs on the parallel port, as well as a routine that copies incoming serial characters to the LCD, so you can check out new displays by hand.

Extra credit project: you can add ANSI cursor positioning support fairly easily. Peek ahead to Chapter 16 for details and some sample code. You can then send the same text to a small character LCD or a big bitmapped panel with similar results.

Shared Bits

The `PortValue` variable lets `LCDSendByte()` toggle the LCD **`Enable`** line while holding all the other bits steady. Unfortunately, `LCDSendByte()` has no way to preserve the other bits previously written to that port, because it can't read back

the current port contents. That has no effect on the demonstration program, but in real life you may have a killer problem.

For example, the watchdog timer routines I discussed in Chapter 7 send a new bit to the watchdog about six times a second. That interrupt handler cannot read back the existing bits, either, so if your code writes a byte to the LCD controller at the same time as the watchdog interrupt routine, your bits get clobbered. When the LCD write completes, it returns the favor by wiping the watchdog bit.

This problem can be easy to solve in a single program, at least after you recognize it. Simply put, all your routines must refer to a single global `PortValue` variable that holds the current output port state at all times. If an interrupt handler can change the port, as is usually the case, you must protect your mainline code by enabling and disabling interrupts around the sections where you update `PortValue`.

Unless you disable interrupts, you can experience some devilishly subtle bugs. For example, when your mainline code loads `PortValue` into a register to change the bits, as most C compilers will, your interrupt handler can fetch the *old* `PortValue` from the variable, change a few bits, and write it back. The mainline code will then destroy the handler's bits when it writes its *new* version of `PortValue` from the register into the variable, atop whatever the handler just stored.

It's worth working through a few examples on paper to convince yourself that you're in a lot of trouble. Look for those instants in time when `PortValue` differs from the actual port bits: *that's* when the interrupt handler will strike. Make sure only one routine can alter `PortValue` at any time, always keep the variable in sync with the hardware port, and disable interrupts around the critical regions.

You'll see these problems again in subsequent chapters, but I should at least point out the land mine right now. The solutions to this problem are both well known and well documented in the multitasking literature. You've surely read about semaphores, critical regions, and threads of execution by now. If not, well, those same issues will crop up when you write multithreaded Windows programs.

Release Notes

The code includes two Micro-C programs: `SerNum.C` for the DS2400 circuits and `LCDTest.C` for character LCD panels. You can combine the essential routines into a single program, but pay attention to the shared bits problem. When you stir in the watchdog code from Chapter 7, it gets still more complex.

In the next chapter, we'll see how to put C code into a BIOS extension, so you can write a program that can tell you you're not authorized to use your own PC…