



Code 39 is popular because it represents both text and numbers (i.e., A-Z, 0-9, +, -, ., and <space>). Figure 1 shows all the encodation patterns for the Code 39 symbology.

Each Code 39 character is made up of five bars and four spaces, making a total of nine discrete elements. Of these nine elements, three are always about twice as wide as the others. The placement of the wide elements determines which character is represented.

Because Code 39 has only two element widths—wide and narrow—a binary translation comes naturally. Simply think of every narrow element as a 0 and every wide as a 1.

Using Figure 1, you can easily determine that an encoded letter A is represented by the pattern 100001001b or 109h. Similarly, the pattern for B is 001001001b or 49h.

A Code 39 symbol always begins and ends with an encoded asterisk. Referred to as the start/stop code, this character frames the encoded data. You can think of the asterisk as a preamble and closing that lets the decoder know where a Code 39 symbol begins and ends.

## MAGIC WAND

Now that you can decode a symbol with your own eyes, it's time to give some specialized sight to a decoding platform. There are different types of input devices, but for this project, I used a simple wand. Most wands look

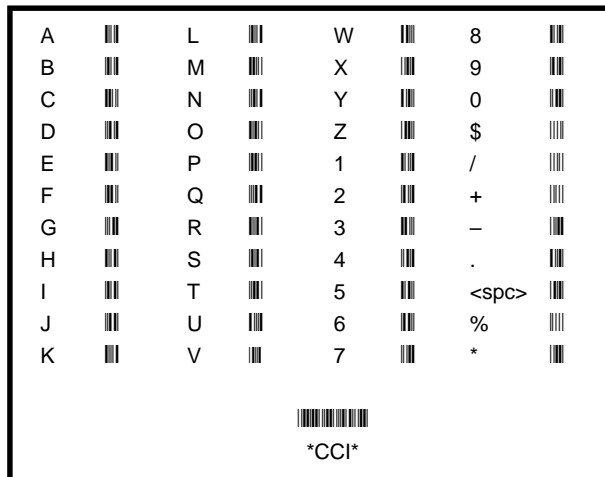


Figure 1—Here's the complete Code 39 character set. You may want to enlarge it on a copier so you can better see the ratio of wide to narrow elements.

like a thick pen and have three signals—+5-V power, ground, and video out.

The internal operation of a wand is rather straightforward. Light from the tip illuminates the symbol, and an internal sensor converts the black bars and white spaces into logic levels.

You can probably build your own wand with an LED, Schmitt trigger, and phototransistor, but I recommend buying one. Hewlett Packard's HBCS-A500 operates at 5 V and draws a mere 5 mA. I bought mine direct for \$110, which I consider to be a small price for a solid product.

## INTO THE PC

The decoding platform I chose to develop for is a '486 running DOS V.6.2. It seemed to be the best middle ground between the embedded world and Windows NT.

My hope is that anyone can quickly adapt a DOS example to suit their needs. Besides, it's a safe bet that you have a PC at your disposal, so here we go.

Listing 1—Sampling the video couldn't get any easier. Embedding the code to set RTS is a convenient way of ensuring that the wand is always enabled.

```
#define COM1      0x3F8    // base address of COM1
#define WHITE     0        // bit 7 is low on white
#define BLACK     0x80    // bit 7 is high on black
BOOL SampleWandVideo(void)
{
    BOOL Data;
    outp(0x3FC,3);        // be sure to raise RTS!
    Data=inp(0x3FE);      // sample the data
    if ((Data&BLACK)==BLACK) // if we're on black, return false
        Return (FALSE);
    else return(TRUE);    // otherwise on white, return true
}
```

Wands don't come ready to connect to a PC, so I constructed an interface. I first created an adapter for the parallel port by connecting the wand's ground to parallel port pin 25, video, to pin 15.

For power, I used a 5-V regulator connected to a 9-V power lump on the wall. However, an AC power converter eliminates any possibility for portable applications, and I wouldn't inflict a battery pack on anyone.

A while back, I read some documentation about the comm port and noticed that each pin

can source 10 mA—more than enough to drive my 5-mA HP wand. Eureka! I cast aside my AC converter and made the interface depicted in Figure 2.

To apply power to the wand, I set DTR high (+12 V). The 100-Ω resistor, in series with DTR, works in conjunction with the wand's internal resistance to yield the required 5 V.

Here's something to keep in mind: When you're constructing your own interface, don't overlook the protection diode or you may damage the wand.

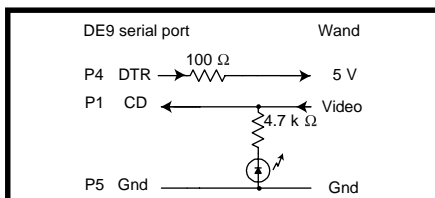
## SOFTWARE SIDE

With the hardware complete, it's time to dig into software. For this application, I used Borland C V.3.1. You can use another compiler, but my examples and source code may not compile without modification.

Whatever compiler or language you choose, the source code is still your best resource for understanding the decoding process. Nevertheless, I want to cover some important areas that will make the source code more readable.

Any given comm port is controlled by 10 programmable one-byte registers, all of which are accessed through port addresses 3F8h-3FEh for COM1, or 2F8-2FE for COM2. Of these registers, the decoder is only interested in two.

Apply power to the wand by writing a 1 to 3FCh (COM1) or 2FCh (COM2). If you're probing DTR, you should see it jump to +12 V. Turn off the wand by writing a 0, and DTR drops to -12 V. With DTR set to +12 V, you can get



**Figure 2**—Because the PC serial port is rated to source 10 mA per pin, I can tap more than enough power off DTR. As a bonus, this unorthodox interface leaves the Rx/Tx lines free for other applications.

the status of the video line by reading bit 7 of port address 3FEh (COM1) or 2FEh (COM2) (see Listing 1).

Do this several times in a loop as you move the wand over a printed page. A little experimentation shows that bit 7 is set when the wand is on a black region and clear when it's on white. Note that lifting the wand away from the white paper produces the same effect as scanning a black region.

When scanning barcodes, most users place the tip of the wand on the quiet zone to the left or right side of the code and then move it across the symbol. When the scan is complete, the user lifts the wand and gets ready for the next scan. The initial placement of the wand on the quiet zone provides the decoder with an easily recognizable moment to begin digitizing input.

As the wand moves across the symbol, the decoder records the time spent on each black or white element. The timer that tracks the element widths is usually nothing more than a loop counter. As the decoder hits a black-to-white or white-to-black edge, it saves the count value, resets the counter, and waits to hit another edge.

The wider the element, the more counts it takes to cross it. If the counter overflows or exceeds a processor-dependent value, the wand has probably been lifted from the paper and you can begin decoding the buffer.

## PATTERN MATCHING

Next, you convert the buffer of counts into discrete element widths. As I mentioned, all Code 39 characters are made up of nine elements—six narrow and three wide. To determine if an element is wide or narrow, compare its width (in counts) against the average of its eight closest neighbors.

If an element is greater than 1.4× the average, it is definitely wide. By

substituting 1s and 0s for wide and narrow, you can build the pattern for a look-up table.

You may wonder, “Why compare against the average of the eight closest neighbors? Why not take the average of all sampled elements and compare against that?” I fell into this trap at first, and wound up with a decoder that didn't work at all.

Fact is, you tend to accelerate as you move the wand across the symbol. So, a narrow element at the start of the symbol may be 10,000 counts, but one at the end will be a miniscule 1000 or even 100 counts. By averaging the element widths on an as-you-go basis, you can factor out most of the error.

When the counts are decomposed into wide and narrow elements, you chunk through the buffer, comparing the binary patterns against a look-up table and appending the decoded characters to a string. That's it!

In the past, you might have resorted to clever look-up methods like hash tables. But, even the slowest processors today provide more than enough horsepower to justify a brute-force sequential-search approach.

## CHECK-OUT TIME

For clarity, I focused on Code 39, but the same principles apply to any symbology type: collect buffer, convert to a binary pattern, and perform a brute-force match. The only thing that varies is the patterns to look for.

I built this decoder on a PC, but you can just as easily apply it to an embedded platform. In fact, an embedded implementation is easier because you have more direct control of pin I/O and timers. Good luck! ☒

*Craig Pataky is a systems engineer with over nine years of experience ranging from simple embedded programming to OS design. You may reach him at [craig@logical-co.com](mailto:craig@logical-co.com) or visit his website at [www.logicfire.com](http://www.logicfire.com).*

## SOFTWARE

Source code for the barcode decoder is available via the Circuit Cellar web site.

## REFERENCE

R. Palmer, *The Bar Code Book*, Helmers Publishing, Peterborough, NH, 1995.

## SOURCES

### HBCS-A500

Hewlett-Packard  
(800) 235-0312  
(408) 654-8675  
Fax: (408) 654-8575  
[www.hp.com](http://www.hp.com)

### Borland C V.3.1

Inprise Corp.  
(800) 457-9527  
(831) 431-1000  
Fax: (831) 431-9527  
[www.inprise.com](http://www.inprise.com)

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, [subscribe@circuitcellar.com](mailto:subscribe@circuitcellar.com) or [www.circuitcellar.com/subscribe.htm](http://www.circuitcellar.com/subscribe.htm).

**SAELIG  
PU 102 P 29  
B/W**

**PG21**