Column #10, December 1995 by Scott Edwards:

Put your Data Up in Lights Using an LED Display Chip

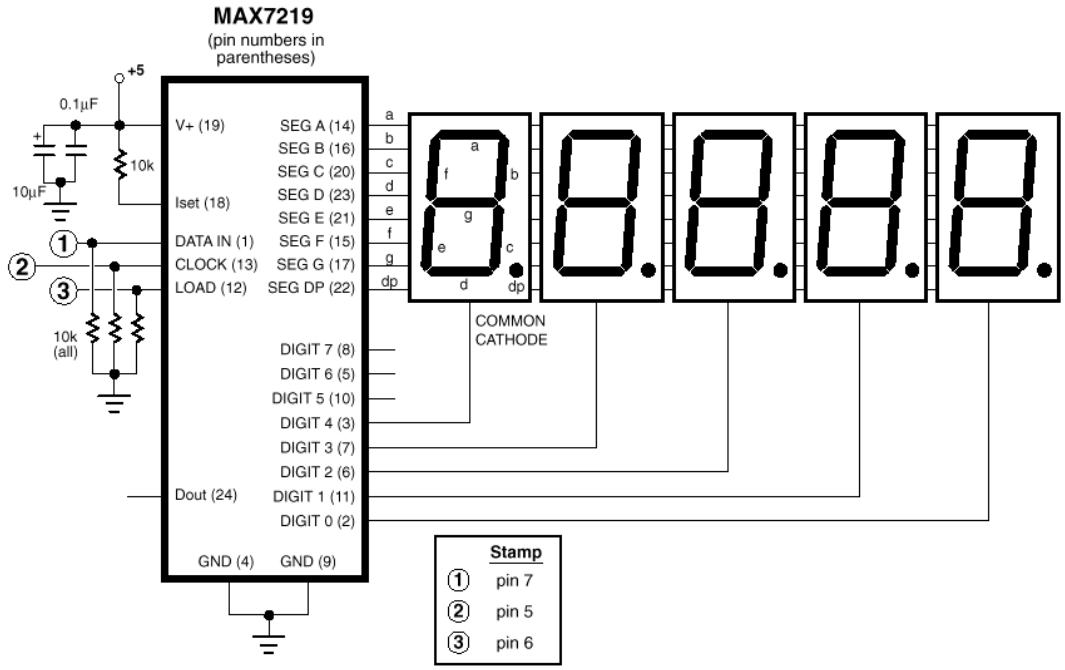
Interfacing the MAX7219 LED Driver

ALTHOUGH most consumer-electronic gear uses liquid-crystal displays (LCDs), military and industrial users are still in love with light-emitting-diode (LED) readouts. Balanced against LEDs' brutally high current draw and poor contrast under bright lighting are their toughness, broad viewing angle, and wide operating temperature range. These are significant advantages that the goop-under-glass construction of LCDs cannot match.

Of course, you might prefer LEDs just for their snazzy appearance. This month's column will show you how to use an off-the-shelf LED driver chip to add an LED display to your Stamp projects with a minimum of hardware and software overhead.

And since so many of you have asked, I'm beginning a new feature this month—a column within a column—introducing the fundamentals of BASIC programming.

Figure 10.1: Schematic diagram for five-digit display based on the MAX7219



Meet the Max

The Maxim MAX7219 LED display driver is the key to creating a Stamp-friendly LED display. The chip's features include:

- Drives up to eight 7-segment (plus decimal point) LED displays, or 64 discrete LEDs.
- Multiplexes display at high speed (more than 1200 Hz) to prevent visible flicker.
- Decodes binary-coded decimal (BCD) digits into patterns of LED segments.
- Controls current to the LEDs and provides software control over brightness.
- Permits software configuration of the display width from one to eight digits.

Figure 10.1 and Listing 10.1 show how the hardware and software go together. Before you run off to build a display based on them, let's discuss some operating principles of multiplexed LED displays.

Multiplexing Makes the Most of the Max

The MAX7219 uses a technique called multiplexing to drive 64 LEDs with just 16 output lines. How? Look Figure 10.1. All of the displays' individual segment anodes (the + connections of the LEDs) are connected in parallel. Within each display, all of the LED cathodes (the – connections) are tied together.

This is called a "common cathode" configuration. Imagine that the *a* segment line is connected to +5 volts, but only one display's common-cathode lines is grounded. The top bar (segment a) of that one display would light up. The other displays would remain dark, lacking a complete path from +5 to ground.

Now, if you grounded a different display's common-cathode line, its segment *a* would light.

And if you switched the ground connection from one display to another 30 or more times a second, it would appear that all of the displays' *a* segments were lit at once.
It's not much of a stretch to see that a fast controller could create the impression of lighting all of the displays with different patterns of lights by rapidly switching the segment lines and scanning the digit lines. That's called *multiplexin*g, and it's what the MAX7219 does.

In addition to multiplexing the displays, the MAX7219 incorporates tables that correlate the numbers 0 to 9 to their corresponding patterns of LEDs. For example, the number 3 is represented by lighting LED segments a, b, c, d, and g. It's normally the program's responsibility to convert digits into LED segments. However, the MAX7219 can perform this conversion for you, depending on a configuration setting. This feature saves at least a dozen bytes of PBASIC program memory in applications that use numeric displays.

Synchronous Serial Communication

The MAX7219 uses a three-wire synchronous-serial interface. We've seen these before with such peripherals as the LTC1298 analog-to-digital converter, DS1620 digital thermometer, and many others. This type of interface sends one bit at a time, just like RS-232 asynchronous serial. It differs in that it requires a separate clock pulse to tell the receiver when to grab the next data bit.

The listing shows how this process works on a BS1-type controller in the code labeled Max_out. The new BS2 controllers have a command called Shiftout that handles the

whole process. In the BS2 version of the program (included with the Parallax AppKit), the code within the For/Next loop is reduced to:

```
shiftout, Data_n,CLK,msbfirst,[temp]
```

...where *temp* is the variable containing data to be sent to the MAX7219.

Final Hardware Notes

As I mentioned at the beginning of the article, one of the reasons for preferring LCDs to LEDs is current draw. The MAX7219 provides a means of setting the segment current of the LEDs through the ISET pin. The smaller this resistor, the greater the current through each LED segment. The value shown in the schematic—10k—sets the maximum segment current of 40 mA. If all eight segments of a particular display are lit, the current draw is 8 x 40 = 320 mA. I mention this because the voltage regulators on the Stamp products are limited to 50 mA; the Counterfeits to 100 mA. You can increase the value of the ISET resistor, but the display won't be as bright. At 60k the segment current will be approximately 10 mA, and combined maximum draw will drop to 80 mA. Depending on the LED displays you choose, this maybe bright enough.

Finally, you may be wondering about the 10k pulldown resistors on the interface between the Stamp and the MAX7219. What purpose do they serve? When a Stamp or Counterfeit resets, either when you first apply power or push the reset button, its pins are in input mode. They are effectively disconnected, so any digital inputs connected to them float. Such inputs frequently float high (logical 1, as though connected to +5V), but noise can cause them to change states at random.

During the time it takes the Stamp to come out of reset, noise on these inputs can put the MAX7219 into test mode, with all segments lit. The resulting current draw may overwhelm the voltage regulator, and prevent the Stamp from ever waking up. Less seriously, it may cause a bright, momentary flash on the display, making the user think that something's wrong with it. The resistors are cheap insurance against such embarrassments.

If it's so *BASIC*, how come I don't understand it?

I've received calls and e-mail recently from folks who read this column regularly and are intrigued by the applications they see here, but don't know anything about programming. They're eager to get started with the Stamp or Counterfeit, but unsure about learning BASIC.

This was news to me, because BASIC has been universally available and very popular since the dawn of the personal-computer era. Most of the early "home computers" had BASIC stored in read-only memory (ROM) right on the machine. Some form of BASIC has been bundled with every version of DOS, and it's starting to show up as a macro language for programs like spreadsheets and word processors.

On the other hand, bookstore shelves are no longer full to bursting with books on BASIC, and the popularity of programming as a leisure activity for computer users has fallen *waaay* behind *Doom* and Internet flame wars. And many schools are no longer teaching BASIC as an introduction to computing. Ivory-tower types have convinced them that anything as understandable as BASIC must cause irreparable damage to the mind. The bizarre C language—the "C" is for "cryptic"—is much more effective at convincing people to leave programming to professionals.

So it's time for a running tutorial on BASIC. From now on, the final section of this column will contain hints and information on programming for newbies. Next issue will kick things off with a discussion of what a program is, and how to set about writing one. Thereafter, we'll look at topics like memory, math, logic, decisions, loops, and subroutines in detail. Naturally, since this column is about the Stamp and workalikes (the Counterfeit), we'll concentrate on PBASIC, but a lot of the concepts will apply to other BASICs, and to programming in general.

A final thought—most programmers agree that there are really only two effective methods for learning to program; looking at someone else's programs and writing your own. So if you have QBASIC on your DOS machine, or own one of those old built-in-BASIC dinosaurs, fire it up and play around with programming. Many of the old manuals contain great tutorials; try 'em out. If you like it (and you will), take the plunge with a Stamp or Counterfeit. We learn by doing.

```
' Program Listing 10.1: MAX7219.BAS (LED Display Driver with BS1)
' This program controls the MAX7219 LED display driver. It demonstrates
' the basics of communicating with the 7219, and shows a convenient
' method for storing setup data in tables. To demonstrate practical
' application of the 7219, the program drives a 5-digit display to
' show the current value of a 16-bit counter (0-65535). The subroutines
' are not specialized for counting; they can display _any_ 16-bit
' value on the LCDs. (A specialized counting routine would be faster,
' since it would only update the digits necessary to maintain the
' count; however, it wouldn't be usable for displaying arbitrary
' 16-bit values, like the results of Pot, Pulsin, or an A-to-D
' conversion).
' Hardware interface with the 7219:
SYMBOL DATA_n = 7     ' Bits are shifted out this pin # to 7219.
SYMBOL DATA_p = pin7  ' " " " " ".
SYMBOL CLK = 5 ' Data valid on rising edge of this clock pin.
SYMBOL Load = 6       ' Tells 7219 to transfer data to LEDs.
' Register addresses for the MAX7219. To control a given attribute
' of the display, for instance its brightness or the number shown
' in a particular digit, you write the register address followed
' by the data. For example, to set the brightness, you'd send
' 'brite' followed by a number from 0 (off) to 15 (100% bright).
SYMBOL dcd = 9 ' Decode register; a 1 turns on BCD decoding.
SYMBOL brite = 10     ' " " " intensity register.
SYMBOL scan = 11      ' " " " scan-limit register.
SYMBOL switch = 12    ' " " " on/off register.
SYMBOL test = 15      ' Activates test mode (all digits on, 100% bright)
' Variables used in the program.
SYMBOL max_dat = b11  ' Byte to be sent to MAX7219.
SYMBOL index = b2     ' Index into setup table.
SYMBOL nonZ = bit1    ' Flag used in blanking leading zeros.
SYMBOL clocks = b3    ' Bit counter used in Max_out.
SYMBOL dispVal = w2   ' Value to be displayed on the LEDs.
SYMBOL decade = w3    ' Power-of-10 divisor used to get decimal digits.
SYMBOL counter = w4   ' The value to be displayed by the demo.
' The program begins by setting up all pins to output low, matching
' the state established by the pulldown resistors.
let port = $FF00      ' Dirs = $FF (all outputs) and Pins = 0 (low).
' Next, it initializes the MAX7219. A lookup table is convenient way
' to organize the setup data; each register address is paired with
' its setting data. The table sets the scan limit to 4 (5 digits,
' numbered 0-4); brightness to 3; BCD decoding to the lower 5 digits
' (the only ones we're displaying), and switches the display on. The
' MAX7219 expects data in 16-bit packets, but our lookup table holds
' a series of 8-bit values. That's why the loop below is designed to
' pulse the Load line _every_other_ byte transmitted.
for index = 0 to 7    ' Retrieve 8 items from table.
  lookup index,(scan,4,brite,3,dcd,$1F,switch,1),max_dat
  gosub Max_out
  let bit0 = index & 1        ' Look at lowest bit of index.
  if bit0 = 0 then noLoad
  pulsout Load,1              ' If it's 1, pulse Load line.
NoLoad:                       ' Else, don't pulse.
next                          ' next item from table.
```

```
' ====================== MAIN PROGRAM LOOP ==========================
' Now that the MAX7219 is properly initialized, we're ready to send it
' data. The loop below increments a 16-bit counter and displays it on
' the LEDs connected to the MAX. Subroutines below handle the details
' of converting binary values to binary-coded decimal (BCD) digits and
' sending them to the MAX.
Loop:
  let dispVal = counter
  gosub MaxDisplay
  let counter = counter+1
goto loop
' ======================== SUBROUTINES ============================
' The MAX7219 won't accept a number like "2742" and display it on
' the LEDs. Instead, it expects the program to send it individual
' digits preceded by their position on the display. For example,
' "2742" on a five-digit display would be expressed as:
' "digit 5: blank; digit 4: 2; digit 3: 7; digit 2: 4; digit 1: 2"
' The routine MaxDisplay below does just that, separating a value
' into individual digits and sending them to the MAX7219. If the
' lefthand digits are zero (as in a number like "102") the
' routine sends blanks, not zeros until it encounters the first
' non-zero digit. This is called "leading-zero blanking."
MaxDisplay:
  let decade = 10000          ' Start with highest digit first.
  let nonZ = 0        ' Reset non-zero digit flag.
  for index = 5 to 1 step -1  ' Work from digit 5 to digit 1.
   let max_dat = index        ' Send the digit address.
  gosub Max_out
  let max_dat = dispVal/decade ' Get the digit value (0-9).
  if max_dat = 0 then skip    ' If digit <> 0 then nonZ = 1.
  let nonZ = 1        ' If a non-zero digit has already
skip:                         ' ..come, or the current digit is not
  if nonZ = 1 OR max_dat <> 0 OR index = 1 then skip2 '..0, or the
  let max_dat = 15            '..only digit is 0, send the digit,
skip2:                        '..else send a blank.
  gosub Max_out               ' Send the data in max_dat and
  pulsout Load,1              ' ..pulse the Load line.
  let dispVal = dispVal//decade ' Get the remainder of value/decade.
  let decade = decade/10      ' And go to the next smaller digit.
next                          ' Continue for all 5 digits.
return                        ' Done? Return.
' Here's the code responsible for sending data to the MAX7219. It
' sends one byte at a time of the 16 bits that the MAX expects. The
' program that uses this routine is responsible for pulsing the
' Load line when all 16 bits have been sent. To talk to the MAX7219,
' Max_out places the high bit (msb) of max_dat on DATA_p, the data pin,
' then pulses the clock line. It shifts the next bit into position by
' multiplying max_dat by 2. It repeats this process eight times.
' In order to avoid hogging the bit-addressable space of w0, the
' routine uses a roundabout way to read the high bit of max_dat: if
' max_dat < $80 (%10000000) then the high bit must be 0, so a 0
' appears on DATA_p. If max_dat >= to $80, then a 1 appears on DATA_p.
Max_out:
  for clocks = 1 to 8         ' Send eight bits.
  let DATA_p = 0              ' If msb of max_dat = 1, then let
```

```
  IF max_dat < $80 then skip3 '..DATA_p = 1, else DATA_p = 0.
  let DATA_p = 1
skip3:
  pulsout CLK,1               ' Pulse the clock line.
  let max_dat = max_dat * 2   ' Shift max_dat one bit to the left.
next                          ' Continue for eight bits.
return                        ' Done? Return.
```