


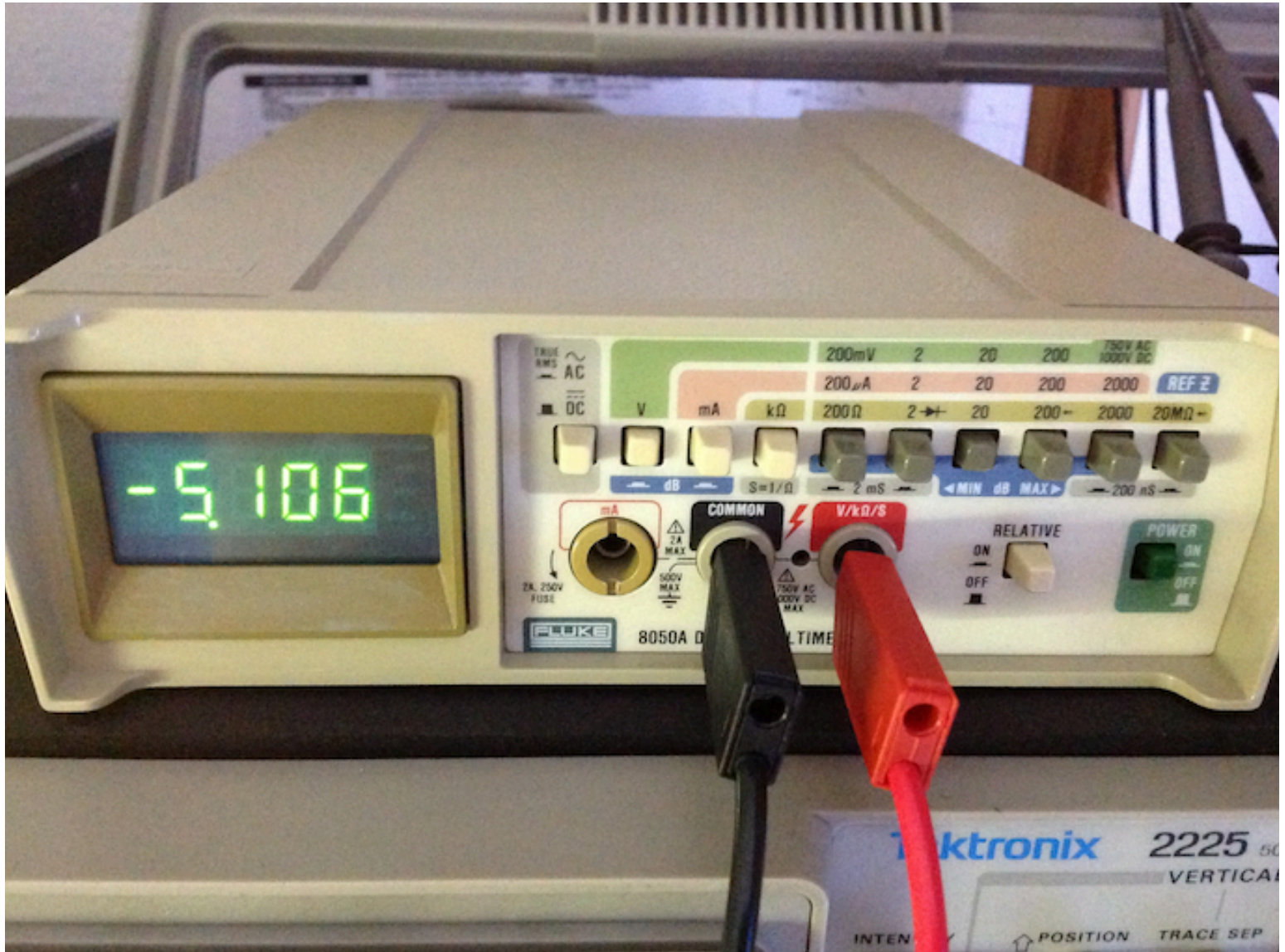


Michael Damkier  Jun 14, 2022 8 min read

A new LED display for a Fluke 8050A multimeter

Keeping old test equipment alive is not just fun, it is how I can support my electronics hobby with good gear on the cheap. Most of the equipment on my bench is over 25 years old. One piece of equipment that sees a lot of use is a Fluke 8050A multimeter.

The Fluke 8050A multimeter was made in the early 1980's and was a popular instrument so a lot of them are still around. The meter uses an LCD display and it is this display that has proved to be the weak point. Many otherwise functional meters are unuseable because the LCD has gone black or is unresponsive. A few people have rescued their meters by replacing the LCD with an LED display. The results are good, looking even better than the original LCD. My 8050A is still working but the LCD is showing its age and has very low contrast. I thought an LED replacement was in order so here is my take on an LED upgrade for the 8050A.



The approaches I've seen involve wiring the individual LED segments directly to the LCD driver chips. (For example, <http://mrmodemhead.com/blog/gallery/fluke-8050a-led-conversion/> and <https://lous.home.xs4all.nl/fluke/Fluke8050Asite.html>) This has the advantage of few additional active components. The main disadvantage is that there is a lot of wiring to the 8050A, soldering directly to the IC pins with a current-limiting resistor for every segment. Also, since the LCD drivers can only sink a couple of milliamps this limits the display brightness (but since the display is not multiplexed, in the upgrades I've seen this doesn't seem to be a problem).

My approach differs from the others by using a microcontroller (ATmega328P, I used an Arduino UNO for development) and LED driver (MAX7219) to decode the display drive information from the 8050A (just 11 lines) and multiplex the LEDs. This description focuses mainly on the software, the Arduino sketch, and not so much on the hardware. In my prototype, I used what I had on-hand. It's all through-hole parts on perfboard. I am sure

that a PC board with surface mount components based on this circuit or a similar approach could result in an almost drop-in replacement for the LCD.

The Project Code

You can download the Arduino project here:



LED_8050A.zip

Download ZIP • 7KB

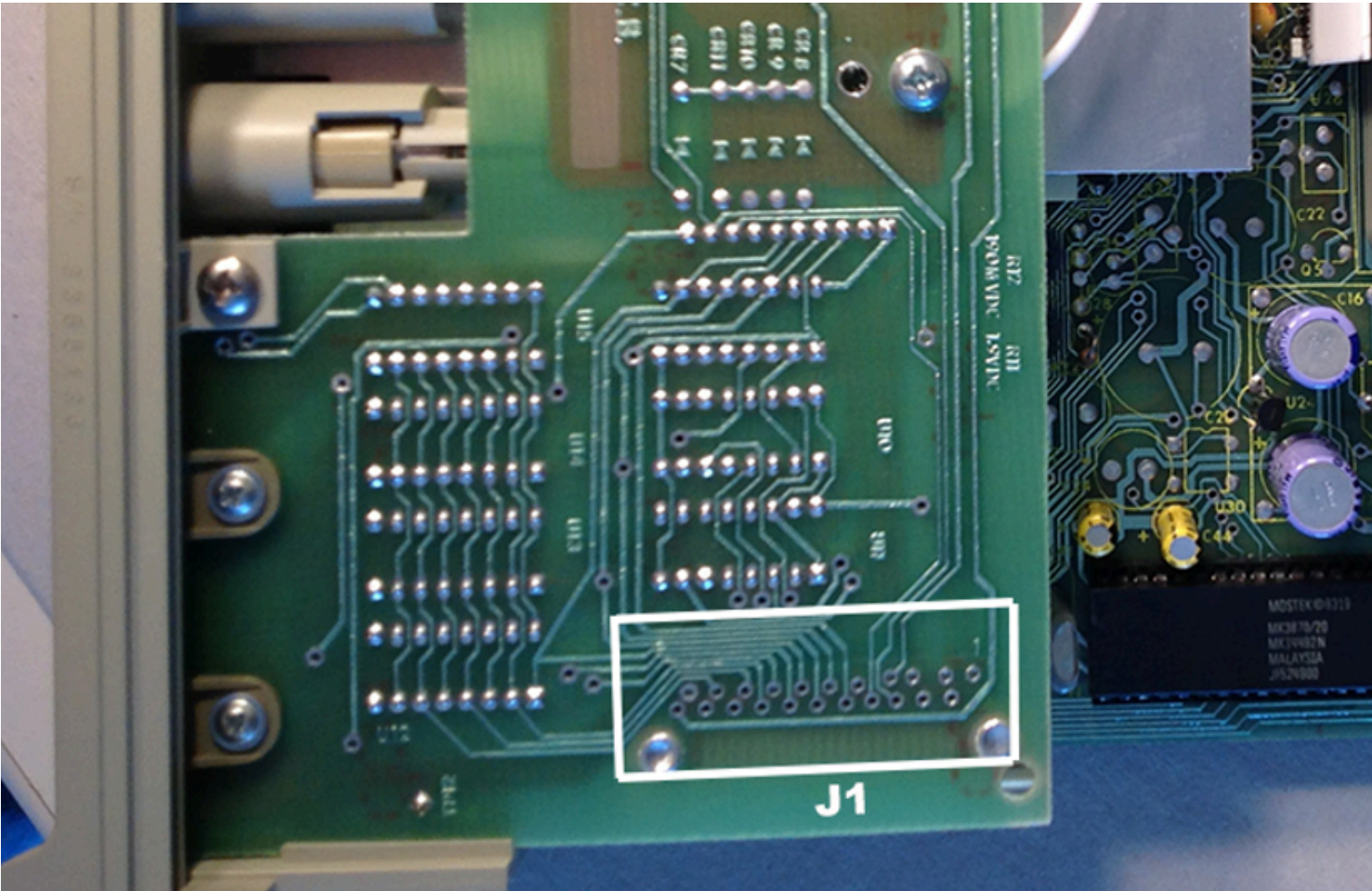


There are two sketches, LED_8050A, which is described below, and an 8050A emulator, Strobe_8050A, that generates strobe and data signals for testing.

[Update: The strobe emulator is only valid for the Fluke's normal operating mode. Strobes are different when setting dB impedance. See the LED_8050A.ino file.]

Inside the Fluke 8050A

The Fluke 8050A drives the LCD display using multiplexed (strobed) data from a 3870 microcontroller. There are five strobe lines and six data lines. These lines are all available at connector J1 on the display board. Also, power for the LCD is supplied (system ground and -5V) and there's a voltage reference for the low battery indicator (not used in my meter).



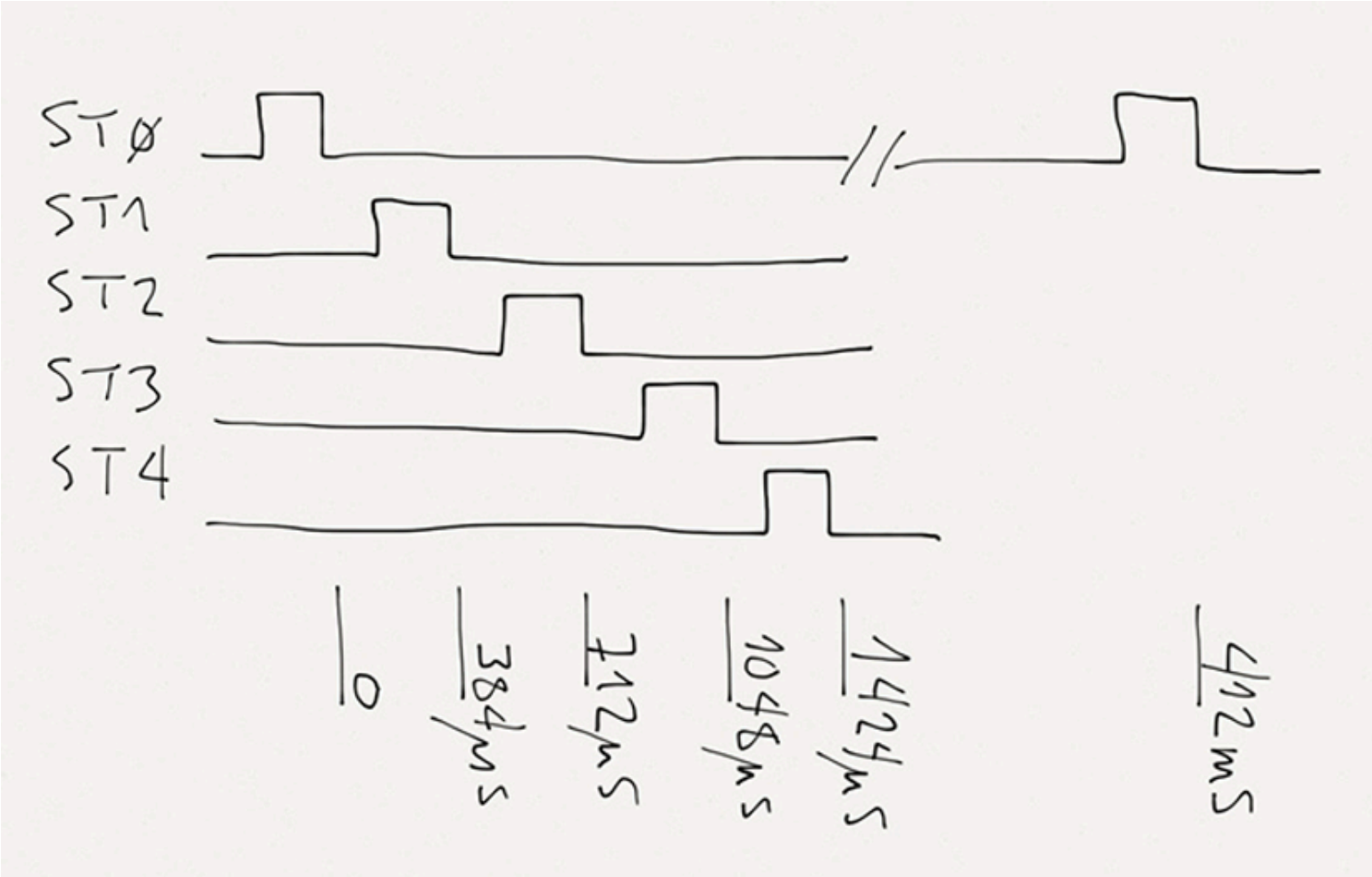
The following table lists the applicable connector pin numbers. (In the image, pin 1 is in the upper-right corner in the box.)

J1 Pin	Function
2	GND
3	-5V
6	batt Vref
10	ST4
11	ST3

12	ST2
13	ST1
14	Z
15	Y
16	X
17	W
18	ST0
20	HV
21	DP

The lines ST0-4 are the five strobe lines. The W, X, Y and Z lines carry BCD number data, W is the most-significant-bit, and the DP and HV lines are for the decimal points and high voltage indicator.

The LCD digit segments and indicators are driven by seven LCD driver ICs, type 4054 and 4056, that decode and latch the strobed data. The 4054/4055/4056 datasheet indicates that the data is latched on the strobe falling edge. The strobe timing, as measured in my 8050A, is as follows.



Pin Change Interrupts

The five strobes are handled by pin change interrupts on the Arduino. A little background. (Refer to the ATmega328 datasheet for more detailed information.) To specify the interrupt pin (or pins) to listen to, you have to set which of three banks and the individual pin(s) in that bank. A separate ISR (interrupt service routine, i.e., the code you want to run when an interrupt occurs) can be set for each bank. It breaks down like this.

ATmega328 Bank Name	ATMega328 Pin Names	Arduino UNO Pins
PCIE0	PCINT0-7	8-13 (14,15 not available)
PCIE1	PCINT8-14 (15 not used)	A0-5 (6,7 not available)
PCIE2	PCINT16-23	0-7

You specify the banks you want to listen on for interrupts using the ATmega328 register PCICR. The bank name in the table is actually the name of a bit in that register. So, in your code you would enable pin change interrupts for Arduino pins A0-A5 by setting the PCIE1 bit in the PCICR register. For example.

```
PCICR = _BV( PCIE1 );
```

(The macro `_BV(bit)` converts a bit number to a byte value with the corresponding bit set. It is implemented as `(1<<(bit))`. The Arduino language also has bit manipulation functions such as `bitSet()`, `bit()`, etc.)

Once you have enabled a bank of pins for pin change interrupts, you need to tell the ATmega which pins in the bank to actually listen to. (You can listen to one or to multiple pins.) There are three registers for this, one for each bank, named PCMSK0, PCMSK1 and PCMSK2. In my code, I'm using bank 1 (PCIE1, Arduino pins A0-A5) so I need to set the appropriate bits in PCMSK1. I'm using the pin change interrupts to trigger a read of the data lines for each strobe. So, I need to listen to each strobe line sequentially. I initialize the PCMSK1 register with the bit corresponding to the pin assigned to ST0 (strobe 0). In the code, ST0 is on Arduino pin A0 which is the bit position PCINT8.

```
PCMSK1 = _BV( PCINT8 );
```

The Interrupt Service Routine (ISR)

The operation is simple. The ISR is triggered by a change on the specified pin for the current strobe number (held in the variable `st`). An interrupt is generated for any change on the pin. We're interested in the falling edge, so a quick test. The scancode lines are read into a two-dimensional array, strobes by scan codes. In order to sample the data for each strobe line, the ISR switches to the next strobe (or wraps around back to ST0) after reading the data. Once all five strobes have been read, the `updateDisplay` flag is set. This causes the `displayUpdate()` function, which is called in the `loop()`, to format the display digits and write them to the LED driver. The `displayUpdate()` function then clears the `updateDisplay` flag. There's plenty of time to update the display between the five strobes so, the formatting is pretty straight ahead.

Here is the ISR code.

```
// pin change interrupt service routine
// - reads the scancode values from the 8050A
// - sets the displayUpdate flag when all strobes are read
ISR( STROBE_INTERRUPT_VECTOR )
{
    if ( digitalRead( StrobePins[ st ] ) == LOW )
    {
        // that was a falling transition
        // (In the 4054/55/56 LCD drivers datasheet, it shows
        // the data being latched on the strobe falling edge.)
        // now, read the scan code values
        for ( uint8_t i = SC0; i < NUM_SCANCODES; i++ )
        {
            scanCodes[ st ][ i ] = digitalRead( ScanCodePins[ i ] );
        }
        // next time, next strobe
        st++;
        if ( st <= NUM_STROBES )
        {
            // scanCodes for all strobes are read
            st = ST0; // reset strobe counter
            // now is the time to update the display
            displayUpdate = true;
        }
        // enable the appropriate strobe pin
        PCMSK0 = StrobePin2PCINT[ st ];
    }
}
```


The pins for the scancodes are in the array `ScanCodePins[]` and the pins (the `PCINTn` values) corresponding to the 8050A strobe lines are held in the array `Strobe2PCINT[]`. This allows for flexible pin assignments, but the strobe pins must all be in the same `PCIEn/PCMSKn` bank.

Before anyone says "hey, you can use port reads for the scancodes and store the results in a single byte." Well, I wrote a version that did that and it really didn't save any time or space. And since there is plenty of both (the ISR runs in less than 90 microseconds with an 8MHz clock and the whole sketch is a little more than 1800 bytes), I figured that making the code a little easier to understand and allowing for more flexible pin assignments was more important than a few microseconds or bytes.

Driving the MAX7219 LED Driver

A quick look at the MAX7219 LED driver and the code to talk to it. (Refer to the datasheet for more detailed information.) The device is capable of scanning up to eight 7-segment LEDs plus decimal points. We'll use six. The segment decode can be set for Code B, regular 7-segment numbers from BCD (binary coded decimal), or no decode for individual segment control. I chose to call digit 0 the most significant. The sketch uses digits 0 and 5 in no-decode mode and digits 1-4 in Code B (BCD) mode. The LED brightness is determined by setting the internal current source with a resistor and can also be changed with PWM. All of these things are programmed by writing to various registers.

When powered up, the MAX7219 initializes to a minimalist state and enters shutdown mode. It only scans one digit, it doesn't do any decode and the PWM brightness is at minimum. So, we need to set these things for our use and bring it out of shutdown mode in the Arduino `setup()` function.

```
void setup()  
{  
    . . .  
    // initialize MAX7219 registers  
    writeRegister( REGISTER_SCAN_LIMIT, REGISTER_DATA_SCAN_LIMIT );  
    writeRegister( REGISTER_DECODE_MODE, REGISTER_DATA_DECODE_MODE  
);  
    writeRegister( REGISTER_INTENSITY, REGISTER_DATA_INTENSITY );  
    // load initial digit data  
    writeDigits();
```

```
// turn on the display
writeRegister( REGISTER_SHUTDOWN, 0x01 );
. . .
}
```

The 7219 registers are sixteen bits with the high-order byte the register address and the low-order the register data. Writing the registers is straightforward using a three-wire protocol. Here is the writeRegister() function.

```
void writeRegister( uint8_t regAddr, uint8_t data ) {
    // combine the address and data to a 16-bit value
    uint16_t reg = regAddr;
    reg <= 8;
    reg += data;
    uint16_t mask = 0x8000;
    // (from the datasheet)
    // For the MAX7219, serial data at DIN, sent in 16-bit packets,
    // is shifted into the internal 16-bit shift register with each
    // rising edge of CLK regardless of the state of LOAD. The data
    // is then latched...on the rising edge of LOAD.
    digitalWrite( PIN_LOAD, LOW );
    while ( mask )
    {
        digitalWrite( PIN_CLK, LOW );
        digitalWrite( PIN_DIN, reg & mask ? HIGH : LOW );
        // The test to HIGH/LOW is needed because reg is 16-bit
        // and function expects (truncates to) 8-bit.
        digitalWrite( PIN_CLK, HIGH );
        mask <= 1;
    }
    digitalWrite( PIN_LOAD, HIGH );
}
```

Testing

To test the circuit, I also wrote an emulator to send 8050A signals to the display breadboard. It is a very simple Arduino sketch that loops the strobes and takes user input to send the appropriate scancodes.

Hardware notes

The display in the 8050A is 4-1/2 digits with indicators for dB (decibel mode), HV (high voltage warning), REL (relative mode) and BT (low battery indicator for battery equipped meters, mine is not). I used six green 7-segment LEDs. Digit 0, the MSD, is the minus sign and numeral 1, digits 1-4 display 0-9 and digit 5 uses the a, g and d segments for dB, HV and REL, respectively. (You could use the a segment of digit 0 for the BT indicator.) This mimics the original 8050A display. The LEDs are wired on a small bit of perfboard and the two ICs, with a 14-pin header and bypass capacitors on another bit of board that is affixed behind the LED board. I set the appropriate fuses in the ATmega328 to use the internal 8MHz oscillator so there was no need for a separate resonator/crystal. This two-board package fills the space that was occupied by the original LCD and its plastic housing. I used a bit of green plastic fitted to the original display frame piece. Except for the 14-conductor ribbon cable soldered to the J1 vias, the 8050A is not modified in any way.

CAUTION! Be sure to remember that the Fluke GND pin is the *positive* rail (the +5V) for the LED circuitry. The J1 pin labeled -5V goes to the LED circuitry 'GND'.

[Update: The power supply in the Fluke doesn't provide many extra milliAmperes limiting the brightness you can get from the LEDs. So I added a small 5V supply (salvaged from a USB power adapter) to provide power for the LED circuitry. Reference the USB supply to the Fluke GND. That is, connect the USB supply positive lead to the the point on your LED circuit that is connected to the Fluke's J1 GND pin. Remove the connection from your LED circuit to the J1 -5V pin. Connect the USB supply negative lead to that point on your LED circuit.]