LED MATRIX DISPLAY PROOF-OF-CONCEPT

A viscerally simple scrolling message display circuit with a single ATtiny2313 controlling four 5x7 LED matrix displays.

Hardware Design

Basic Concept

An ATtiny2313 has 20 pins, 18 of which are usable as general I/O *if* we program the `RSTDISBL` fuse to have the `/RESET` pin to function as`PA2`. This means losing SPI-based in-system programming, but that can be fixed with a serial boot loader.

The Para-Light C/A-2570x 5x7 LED dot matrix displays have 12 pins: 7 for the rows, 5 for the columns. Using classic multiplexing (the only way we can do because of the internal hookup of the LEDs in the display module), we can have a single ATtiny2313 control two such displays: 7 pins for the rows, plus 2*5=10 pins, totaling 17 pins. We can use the 18th pin for receiving commands over a serial channel.

In order to double that, all we've got to do is alternate common-anode displays with common-cathode displays. Given that each port has three states (high or sourcing current, low or sinking current and high-impedance or 'disconnected'), we can control the current flow for each row of diodes.

The ATtiny2313 seems to be the only 20-pin part from the AVR family where this hack seems possible -- according to the AVR parametric table, all other 20-pin parts have less than 17 usable I/O pins.

Another bold move was doing away with the traditional current limiting resistors typically used then hooking up LEDs. We'll deal with this in software by pulsing the LEDs very briefly before they blow up (around 30-250 microseconds does the trick just fine). I'm told most LEDs don't like being treated like that, but, in my tests, we've been withstanding it just fine so far.

A Few Complications

The typical AVR I/O port is capable of sustaining about 20-30mA current. While this is enough for a single LED or even two, brightness is reduced when seven of them are on. We fix this by varying the 'on' time: if we are to light only one led at one particular multiplexing instant, we'd have it on for just 30 microseconds; if we are to light seven of them, we leave them on for about 200 microseconds. The human eye's persistence of vision integrates all this, providing the illusion of constant brightness.

Another problem is the `PA2` port -- because it is shared with the `/RESET` signal, it has much weaker drive characteristics. We can't use `PD0` for driving the LEDs because it is also the USART's`RXD` pin, which we need for serial communication. The solution is to make the on/off periods longer when we're using `PA2`. As it turns out, they have to be *a lot* longer, up to the point of becoming more than 70% of the frame refresh time.

The circuit

The circuit is viscerally simple: in essence, the MCU pins are directly connected to the displays. There is just one single pull-up resistor to prevent switching noise from being picked up by the UART when using the device in standalone mode (that is, without being connected to the PC's serial port).

The hookup was designed so that, when the MCU is mounted in the other side of the board, almost all of its ports align right next to the displays' ports, greatly simplifying board routing at the expense of more complicated bit swapping in software.

The `PD1/TXD` pin is used for driving the LEDs when the main application is running but it is reverted to its usual data transmit role in the AVR910-compatible boot loader, allowing for in-system software upgrade.

There is no voltage regulator; you should provided regulated 3-5V through the connector.

The two connectors are there because I plan to have several of those together sharing the same serial and power bus, in a scalable modular system to create larger displays.

Software

Firmware Upload

I'm supposing you have already unpacked the distribution package in a directory of your choosing.

Put the MCU in your favorite ISP-based programming board. If you use avrdude and usbasp like I do, check that the paths and settings in the `Makefile` are OK for your system and type:

```
make upload_boot
```

to send up the bootloader, then:

```
make fuses
```

to set up the fuses. **Don't do that before uploading the bootloader** -- the fuses settings will disable the `/RESET` function, so SPI-based in-system programming will be no longer

available. Besides via the bootloader, the only other way to upload the firmware will be if you own a high voltage programmer such as Atmel's STK-500.

Now connect the circuit to your PC's serial port, open your terminal emulator program, configure it to 19200 8N1 and power the circuit up. The bootload greeting message ("AVR910, ESC quits") should appear. To see if bidirectional communication is ok, type ENTER; the bootloader should respond with '?' ("Unrecognized command").

Close the terminal emulator program and type:

```
make upload_serial
```

Finally, in your terminal emulator program, type ESC to quit the bootloader and start the main application. The scrolling message should appear now.

When you want to break the application and go back to the bootloader, press CTRL-C in your terminal emulator. You should get the bootloader greeting message again.

Commands

The application accepts a few commands from the serial port (at 19,200 8N1):

- CTRL-C: Escapes to the bootloader, as previously mentioned
- CTRL-S: toggle single step mode
- SPACE: single-steps
- ENTER: moves the test dot

Extending the Concept

In principle, this technique can be extended to other processors:

- an ATmega8 could control **six** 5x7 displays: 7 rows plus 3 CA/CC groups times 5 columns equals 22 pins. Adding one for serial communcation, we get 23 pins -- exactly the amount of usable I/O pins an ATmega8 has. We would still need to use the /RESET pin as PC6.
- an ATmega8535 could countrol **ten** 5x7 displays: 7+5*5=32 pins, so we'd still have three left! (The ATMega8535 has 35 I/O pins).

However, using larger chips doesn't seem to make things neither easier nor cheaper. Here's a quick-and-dirty cost analysis:

	I/O	Unit Cost		# of	Pins	Pins	Cost Per Display	
MCU	**Pins**	**1**	**25**	**Displays**	**Used**	**Left**	**1**	**25**
ATmega8515	35	5.27	3.31	10	32	3	0.53	0.33
ATtiny2313	18	2.26	1.42	4	17	1	0.57	0.36

ATmega164P	32	4.82	3.56	10	32	0	0.48	0.36
ATmega853	35	5.70	3.58	10	32	3	0.57	0.36
ATmega8	23	3.66	2.30	6	22	1	0.61	0.38
ATmega16	32	6.56	4.12	10	32	0	0.66	0.41
ATmega162	35	6.77	4.25	10	32	3	0.68	0.43

The columns '1' and '25' are the MCU prices in units or lots of 25, taken from Digikey's site on jun/07.

While the 10-display solution with the ATmega8515 comes first in terms of cost-per-display, my intuition is that multiplexing that many columns might result in too low a brightness to look really good. I didn't perform this calculation, but I'd also guess that the slight cost edge would be lost because of the extra PCB area needed to route all connections. I also have a psychological problem with it: it seems like a waste of a rather capable MCU. But it might be worth the try.

The ATtiny2313 is the second runner up in terms of cost per display. I think it hits the sweet spot because besides the cost, four is a good number to make multiples of and the displays/MCUs ratio make it better suited to get volume discounts even for hobbyists like myself that don't plan to build that many or large display arrays.

Source : http://www.postcogito.org/Kiko/LedMatrixDisplayPOC.html