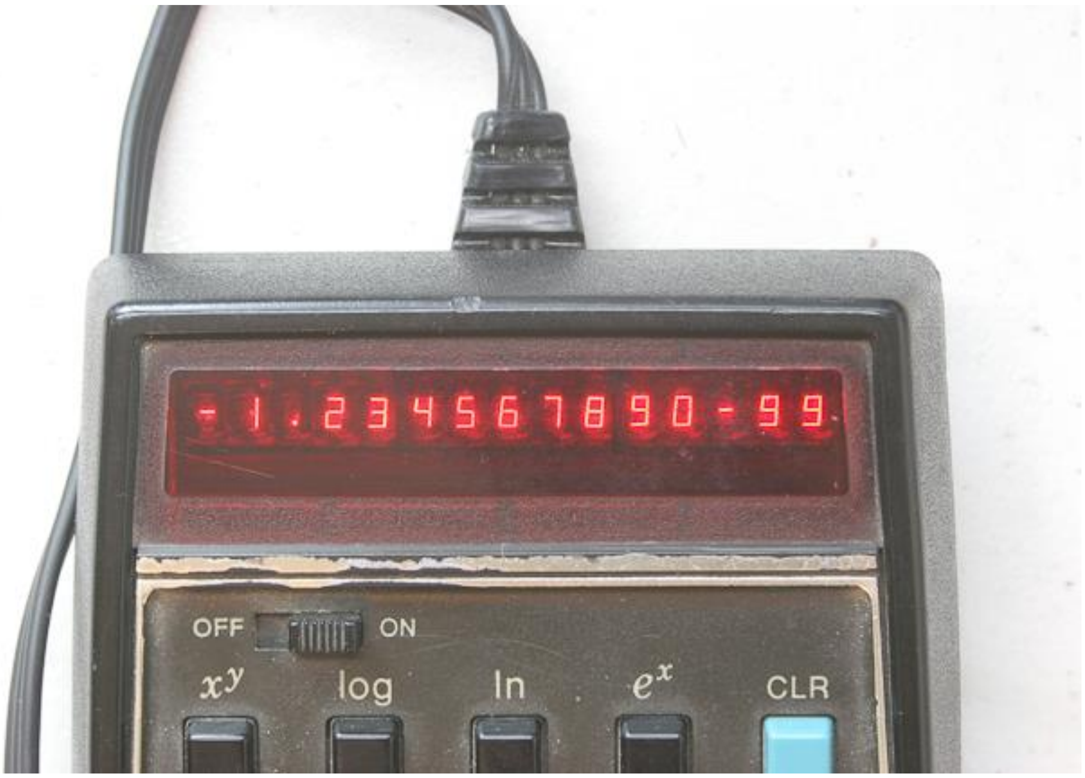
Care and Feeding of the 7-segment [*hp* Bubble Display]

A disclaimer: This is a series of posts about getting the most from these displays. I don't formally blog, but felt I have something to contribute here. I hope SparkFun won't mind my blogging in the forum. Like most blogs this work is not completed yet. I will be adding posts as I make progress. You will also get to see (and comment on) my successes and failures with each posting. Although I've been working on this project for about six weeks, the writing is happening just now. Several posts should come right away. The remainder will come as the project continues in real time.

These displays look halfway decent over a pretty wide range of parameters (significantly wider than the spec says) so you don't have to do everything that I recommend here to get them to work, but I hope to give some tips to get them to look their best.

I love these little guys. They are compact, classy and they remind me of my old hp35 calculator.



If you look closely you can see that the calculator display is actually three 5-digit displays stacked together. (splits between digits 3 & 4 and 8 & 9)

I used these displays in a project in about '76 and loved them. So when I saw SparkFun had them available I immediately grabbed a handful. It appears that's what everyone else did too since they immediately ran out of stock, but they seem to be back in stock now. Great job SparkFun! Very decent price too!

That said driving these from an Arduino is a bit of a challenge. If you take all 8 anodes and 4 cathodes to Arduino pins then you've used up a good portion of the available I/O on the microcontroller. You could use an I/O expander like the 595, but then that adds more parts. On top of that you have to spend a fair amount of CPU power to route data and do timing of the signals. In the end they are still great, but beware the burden on the microcontroller.

There have already been one or two comments on the product page about these LEDs being dim. While they will never match the newer products of today they are not shown in the demo to their best ability. And the product description and Hookup Guide both need some corrections. The next few posts will offer some tips to get the best from these displays.

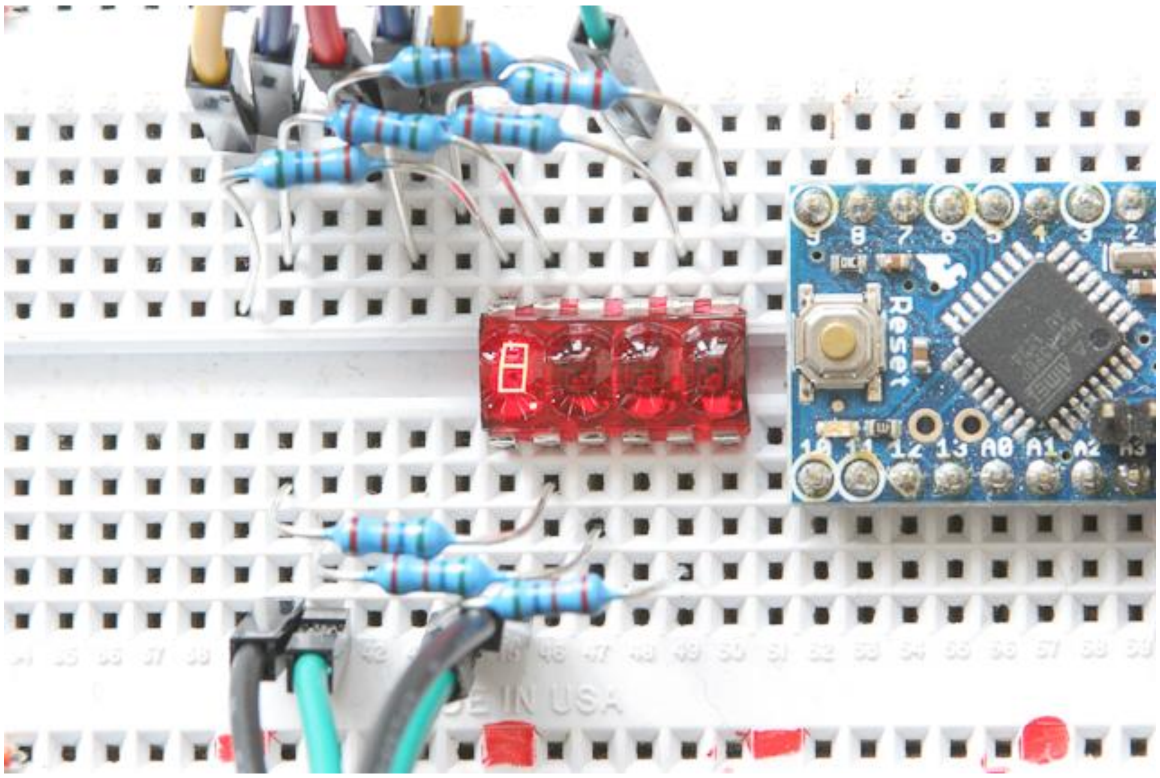
Another post in the product comments was a request for an I2C controller for the display. This would reduce the pins required on the application processor just like the serial backpacks do for the LCD display products. That then will be the ultimate goal of this project; to create a circuit, PCB and software that will take one or two bubble displays and control them over the I2C bus.

NOTE: About the pictures in these posts. I now have a more first-hand respect for the folks who do the SparkFun product shots, the macro ones anyway. I wanted to make these shots useful for comparison of LED brightness. Unfortunately, because they are light emitters, it is difficult to get a proper camera exposure of the LEDs. To keep the photos comparable the breadboard shots are all done at exactly the same exposure; 2 sec @ f/9.

Tips

Tip 1 - Resistors go on the anode side. The Hookup Guide fritzing diagram shows the current limiting resistors on the cathodes. The text talks about the anode option but many folks will just follow the diagram. If you do that the result will be that digit brightness will vary depending on the number of segments lit up. That is, a '1' (2 segments lit) will be brighter than an '8' (7 segments lit). Instead, put the limit resistors on the anode side. Also, the suggested resistor value in the Hookup Guide for each anode is 680 ohms for a 5V system. As we'll see in the next 2 tips, you really want a lower value.

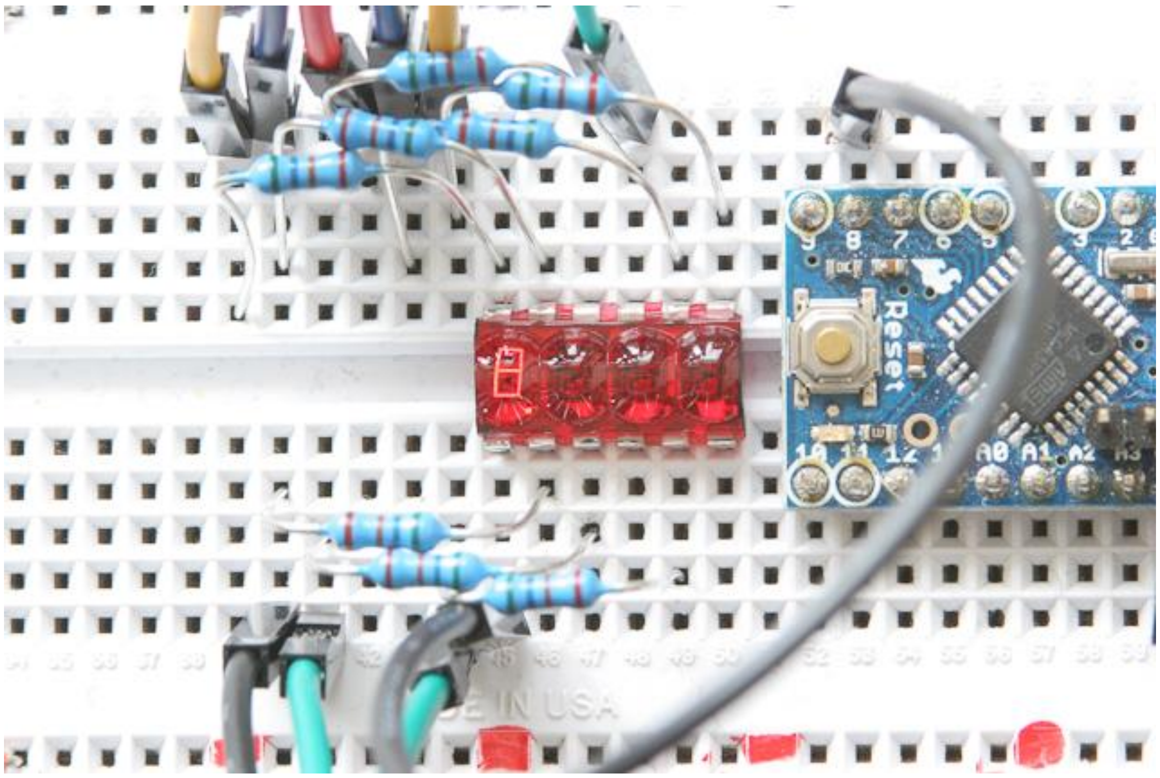
Tip 2 - Drive LED segments to an AVERAGE current of 5mA/segment. The SparkFun product description says that the "displays have a peak forward current per segment of 5mA". That is not right. The hp spec says that the segments have an absolute maximum average current of 5mA. Average is not the same as peak.



In the photo above I wired each anode to 5V thru a 560 ohm resistor (the local R*#%@$+!*K was out of 680's) and connected cathode 1 to ground. This gives a continuous 6.1mA through each segment. Note the brightness. In the next photo I have moved the cathode from ground to D8 on the Arduino Pro Mini and driven that pin with a repeating signal of 3mS low and 9mS high.

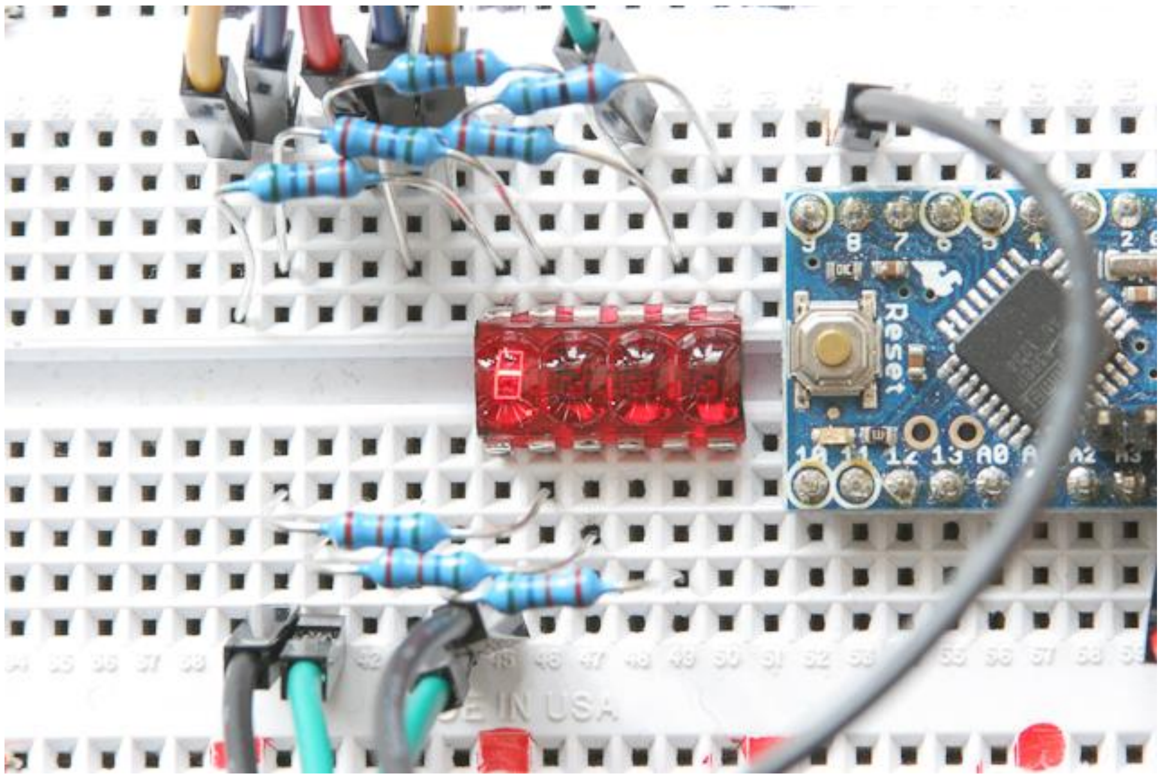
```
#define CATHODE 8

void setup(){
  pinMode(CATHODE, OUTPUT);
}

void loop(){
  digitalWrite(CATHODE, LOW);
  delay(3);
  digitalWrite(CATHODE, HIGH);
  delay(9);
}
```

Compare the display brightness with the previous photo. Much of the brightness has been lost.



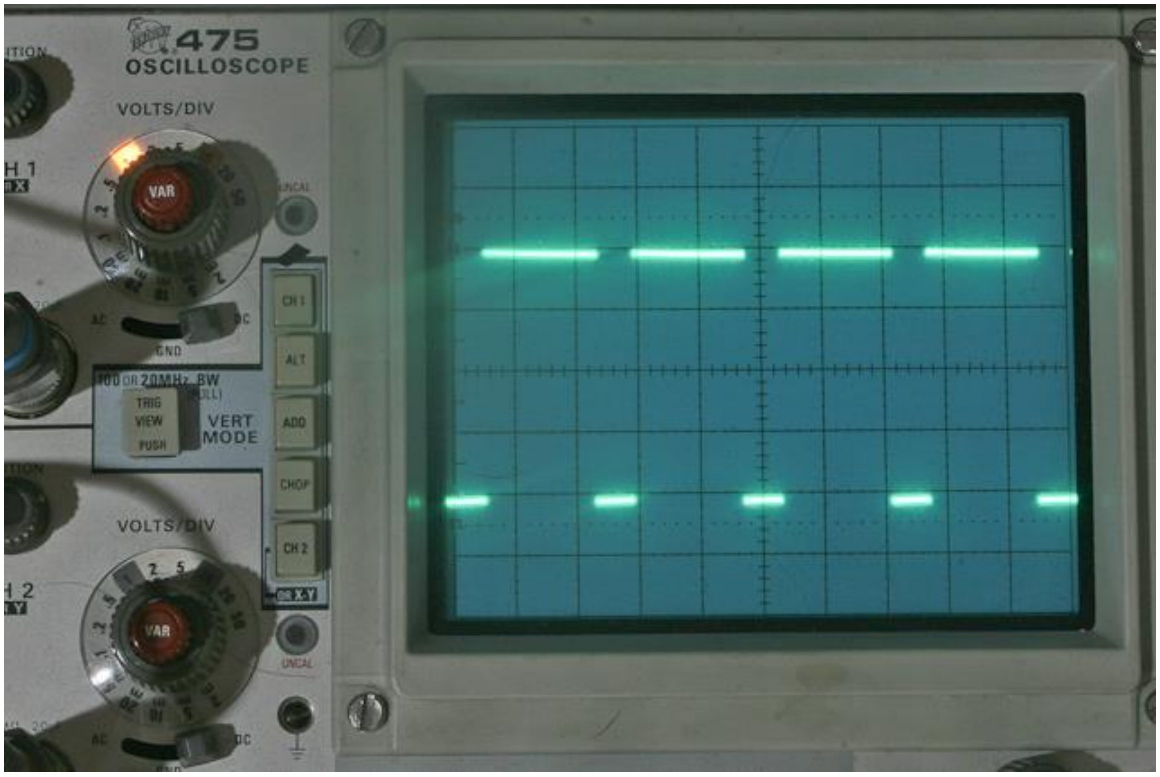
The pulsed cathode signal simulates the real world use of the display. With multiplexing each digit is only lit one quarter of the time. So the "average current" is really only a quarter of what we calculated from the limit resistor value. The result is that the apparent brightness is less, just as if we were using PWM to drive the LED at 25%.

To get the full brightness from these displays you really have to hit them with a much higher pulsed current so that after multiplexing the average current is where we want it to be. In the next photo I have swapped out one 560 ohm resistor with a 56 ohm resistor (you can see the green-blue-black color banded resistor at the top center of the photo).



You can see that the 56 ohm resistor is on segment-g. The brightness difference is even more visible by eye. If you do the math that segment is now being driven at 61mA. Even at 25% duty cycle that’s 15mA average. How is it that the LED is not blowing up? The answer is in the next tip.

Tip 3 - Arduino output low is NOT 0V. The picture below shows the D8 signal on the Arduino driving the cathode as described in the previous post. You can see the 25% duty cycle. You can also see that the vertical height of the signal is four divisions. Vertical scale is set for 1V/div. The ground for the signal is actually another div below the bottom of the signal. So we're missing an entire volt. Our resistor calculations have therefore been incorrect.



Losing that volt on D8 is actually within spec according to the Atmel data sheet (VOL = 0.9V @ IOL=20mA & VCC = 5V). The measurement above was with 6.1mA per segment for a total of about 49mA. If I change the one resistor to 56 ohms, the bottom of the D8 signal comes up another half volt. Imagine what would happen if all segments were on and limit resistors were 56 ohms. Bottom line, the ATmega just can't sink all that current by itself. No problem, it is easy enough to provide transistor buffers for the cathodes.

Tip 4 - Scan fast to eliminate flicker. I started the simulated multiplex (pulsing D8) using 8mS on and 24mS off or about 31Hz. I could see a fair amount of flicker in the digit. Changing the timing to 4mS on and 12mS off (62Hz) I could still see some flicker if I moved my head and looked at the display out of the corner of my eye. Things got reasonably solid at 3mS on and 9mS off (83Hz). Old style television was 30Hz (interlaced) but the phosphors had persistence. Modern LCD TV displays are at 60, 120 or even higher Hz, so 83Hz doesn't seem unreasonable. Multiplex your displays at greater than 80Hz frame rate to minimize flicker. That is, cycle through all digits (a frame) at least 80 times per second.

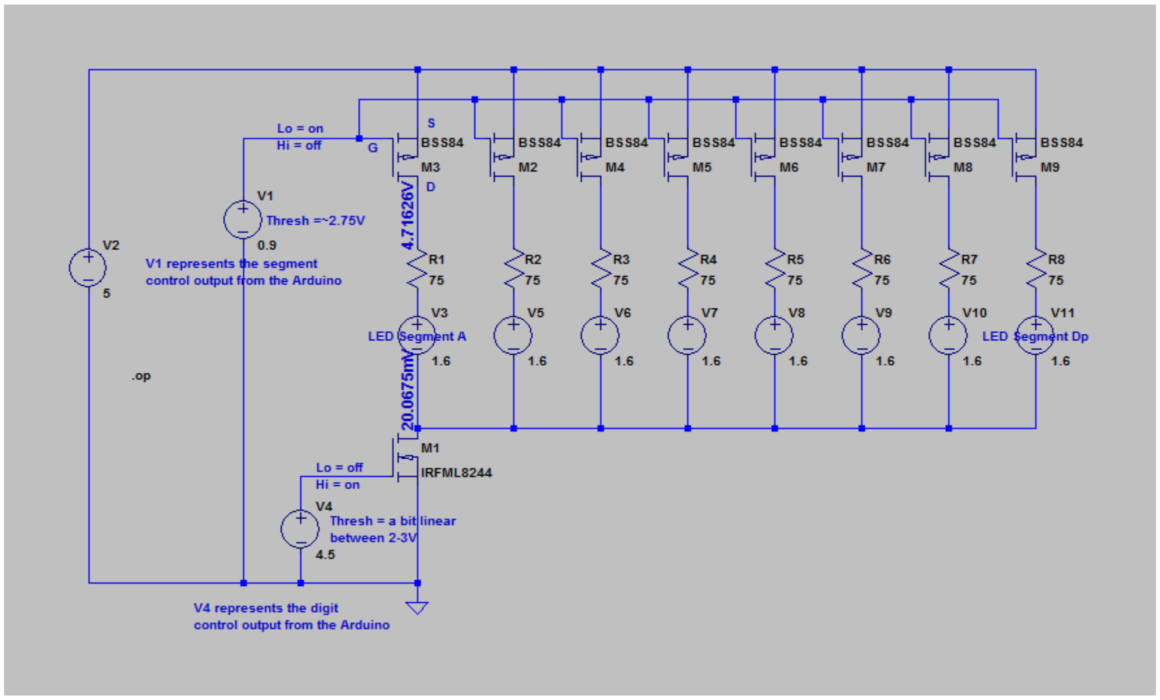
Limit Resistor Value

So I have been beating up on the poor current limit resistor. I quoted 680 ohms, then 560 ohms and even 56 ohms. And I pretty much said or implied that all these values are wrong. So what is the right value for this resistor? To figure that out we really need to decide on the drive circuit. To decide on the drive circuit we need to go back to our specs. For my spec, I want to build an 8-digit display. That is, I want to stack two hp displays side-by-side. I want to run at 5mA average current per segment to get good brightness. Each digit will be on for one-eigth of the time so I want the on-current for each segment (anode) to be 8x5mA = 40mA. That's a max total current of 40mA times 8 possible segments which is 320mA total digit (cathode) current. That is way more than the ATmega can sink so I know I need driver transistors on the digit lines.

I may also need driver transistors on the segment lines. The ATmega can drive 40mA into one pin, but can it drive 40mA into 8 pins at the same time and not overheat? We will have to experiment to find out.

If I was displaying 4 digits then the numbers would be 4x5mA = 20mA per segment (anode) and 8x20mA = 160mA per digit (cathode). While the Arduino might be able to do that, I still think drive transistors are the better design decision at least on the cathodes.

To investigate the drive circuit I did an LTspice simulation.



For transistors, I picked BSS84 MOSFETs (p-ch) for segment (anode) drivers and IRFML8244 MOSFETs (n-ch) for digit (cathode) drivers. I chose the BSS84 because it is common (i.e. I can buy them from Digi-Key), cheap, has a logic level gate and is in the LTspice library. On the anode side the "on" resistance of the transistor does not matter as much. That resistance just adds to the limit resistor. As long as the power dissipation is not above spec an Rds On of a few ohms is acceptable.

The cathode driver needs a bit more attention. The cathode current is the collection all eight segments so it can vary from zero to 320mA. Therefore the greater the Rds On, the more the brightness will vary with the number of segments lit. I chose the IRFML8244 for its Rds On of 41milli-ohms at 4.5V. I also chose it because it is available from Digi-Key, reasonably cheap and is in the LTspice library.

Since MOSFETs are voltage controlled devices I simulated the Arduino outputs as voltage sources (V1 & V4) setting them higher or lower according to the logic output. I also tied all the segments to the same control. That is not how the final circuit will be wired but allows me to simply test the boundary cases.

The forward biased LED is modeled as a 1.6V voltage source (V3, V5, V6, V7, V8, V9, V10 & V11). The whole thing is powered by another 5V source (V2). I only modeled a single digit of the display, but the others would behave the same since only one cathode is ever on at the same time. Note that I am only modeling static LED states "on" and "off". I am not modeling the multiplexing. We have already compensated for the multiplexing in our current calculations above. I have attached the simulation file in case you'd like to try it.

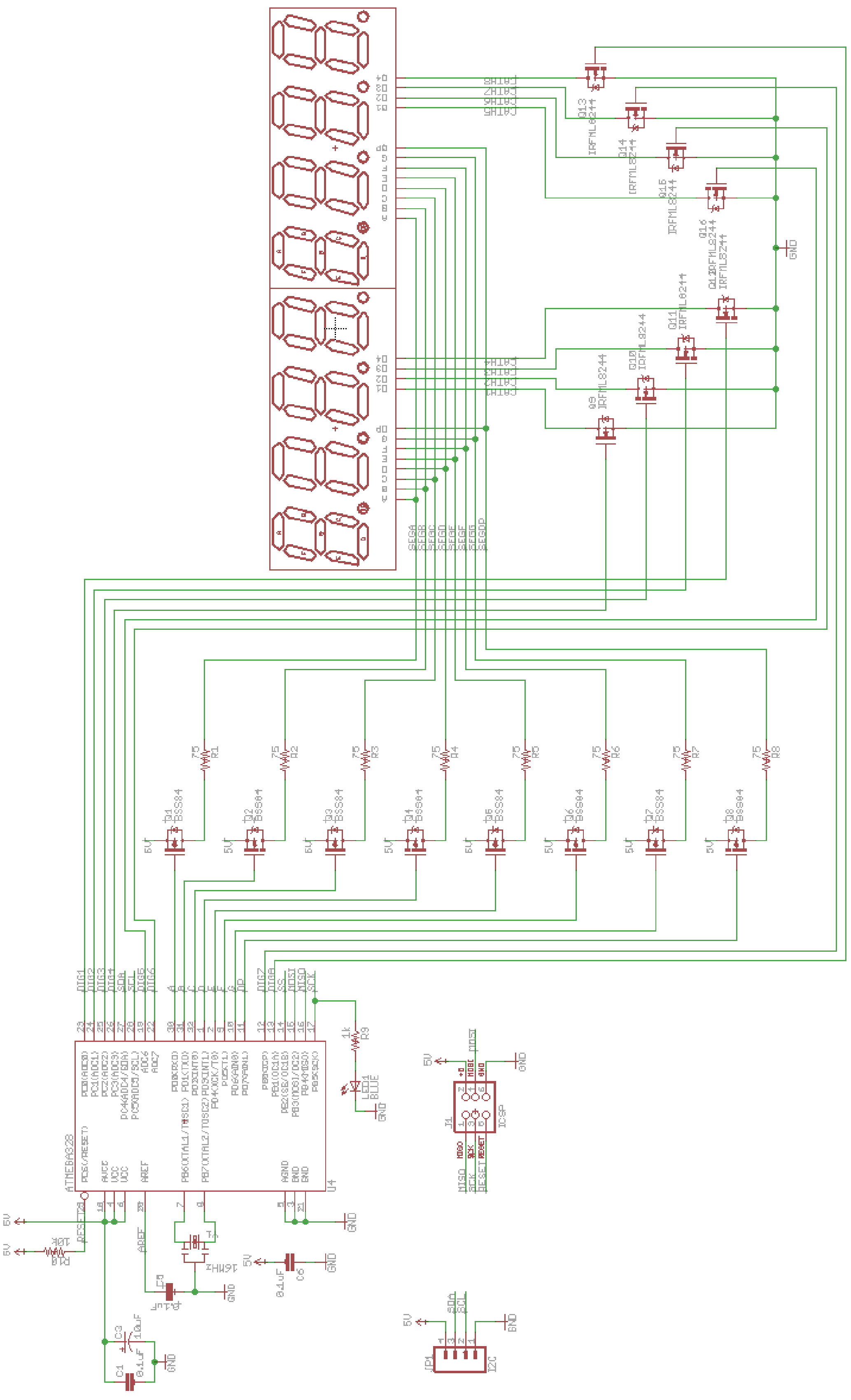
I ran the simulation to determine the operating point with various values for the current limit resistor (R1 thru R8) until I got the current through the resistors (and therefore the LEDs) that I wanted. That turned out to be 75 ohms for an 8-digit display or between 150 and 168 ohms for a 4-digit display. I also made sure that segment current flows would be what I wanted at gate voltages of Vol = 0.9v and Voh = 4.2V, though the loading is so light that I expect these to go closer to the rails in the actual circuit. It will be something to check once it is built up.

Note that the digit cathode voltage is only about 20mV even with all 8 segments lit up. That's just what we want and way better than what we saw with just the ATmega sinking the current.

Building Up the Circuit.

Because I have done some breadboard experiments, calculated spec values and then simulated the drive circuit, I decided to build the circuit straight to PCB. I know that adds risk to the project, but PCBs are relatively cheap these days and I didn't want to buy two sets of parts (one thru hole set and one SMT set). The only real down-side is the time delay if I totally mess things up on the PCB design. I figure on doing at least two rounds of PCB design anyway.

I designed the PCB in Eagle CAD. The following screen capture shows the schematic. You can see the transistor drive circuits are the same as the simulation schematic posted previously.



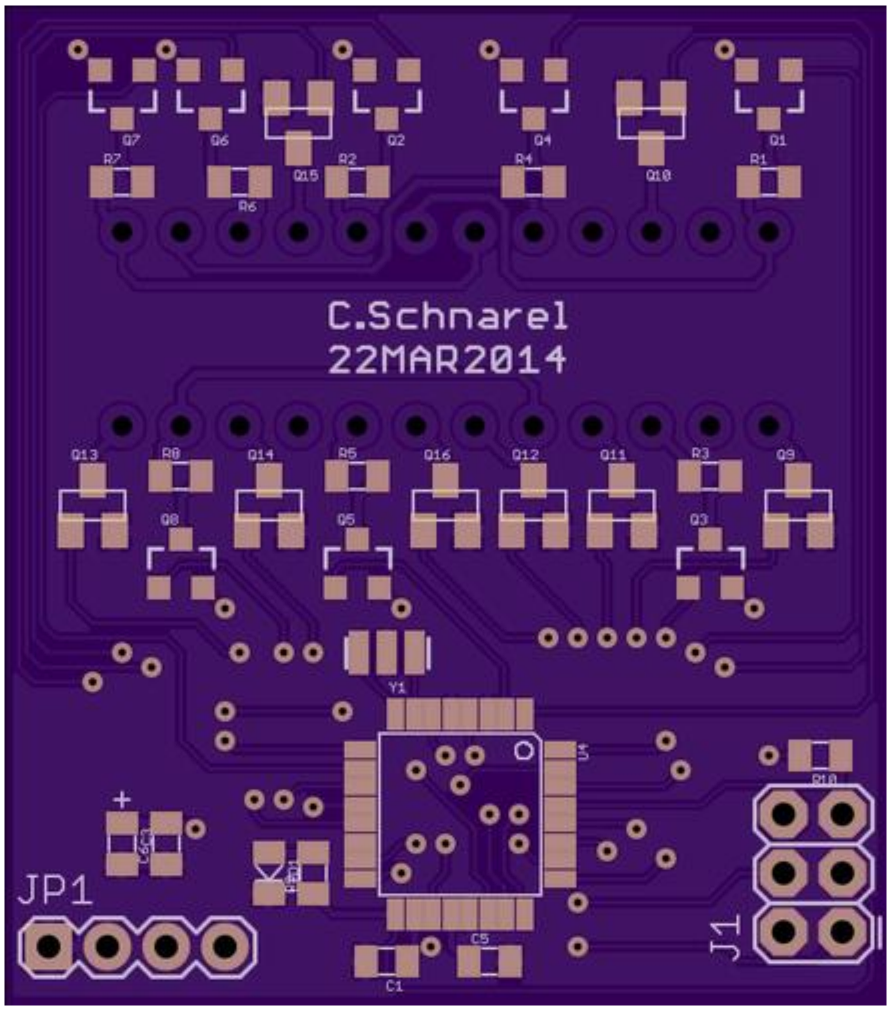
ATMEGA328
U4
DIG1
DIG2
DIG3
DIG4
SDA
SCL
DIG5
DIG6
DIG7
DIG8
SS
MOSI
MISO
SCK
RESET
AREF
16MHz
C1
0.1uF
C3
10uF
C5
C6
R9
1k
LED1
BLUE
R10
10k
5V
GND
SEGA
SEGB
SEGC
SEGD
SEGE
SEGF
SEGG
SEGDP
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
BSS84
R1
R2
R3
R4
R5
R6
R7
R8
75
Q9
Q10
Q11
Q12
Q13
Q14
Q15
Q16
IRFML8244
J1
ICSP
JP1
I2C

You can also see that this design has no serial port for connecting to the Arduino IDE. I knew I would have to program the bootloader onto the board via the ISP port, so I decided that I would just program the display controller application the same way. Turns out you can program either the bootloader or your application via the ISP port directly from the Arduino IDE anyway. Way to go Arduino! I use the SparkFun [Pocket AVR Programmer](https://www.sparkfun.com/products/9825) but you can also [wire up an Arduino](https://learn.sparkfun.com/tutorials/installing-an-arduino-bootloader) to do the programming.

Laying out the board, most of the components were already available in the [SparkFun Eagle Libraries](https://github.com/sparkfun/SparkFun-Eagle-Libraries). The only one that wasn't was the bubble display itself. The SparkFun library does have a similar display with 20mm tall digits. I just copied it and added a 12-pin DIL package variant to that part. The DIL package I also copied from another SparkFun library. It was 14 pins so I just chopped off two.

After routing the board and using the DRC (design rule check) tool I sent the design off to [OSH Park](www.oshpark.com) to be fabricated. My design is combined with others on a panel and then sent by OSH Park to the fab house. Once fab'ed OSH Park receives the panel, breaks apart the individual boards and ships mine back to me. The whole process cost me about $13 for 3 PCBs and takes about two weeks.

The first thing I noticed when the boards arrived was that there was soldermask covering the holes for the bubble displays. Why don't I ever notice these things on the check plot? I went back and looked. The problem is obvious on the OSH Park rendering! DOH!



Turns out I made a mistake when I copied the 14-pin DIP part. Hidden layers don't get copied. I didn't have the solder mask layer visible. Oh well scraping with a hobby knife exposed the pads so I can solder in the displays. I've since fixed my mistake in the component definition and submitted it back to SparkFun to be included in their library. It's in there now.

There were no other obvious errors on the board so I built it up using only one display. I'm pretty used to hand-soldering SMT components. I design with 0805 resistors and caps because that's about as small as I'm comfortable with. The only real challenge with this board was the ceramic resonator. Because the pads are underneath the resonator part I pre-tinned one pad on the PCB and then slid the resonator in pushing it to one edge of the PCB pads. Then I "wicked" the solder to the other pads. I kind of over-did it and had to remove excess solder with some copper braid but it seems to have worked. The key to having the soldering come out right is using liquid flux. It allows the solder to flow more freely.

The soldermask over the display pads was really the only board error. I was able to program a bootloader via the ISP. I'm not sure what good that does since there is no port for the Arduino IDE to connect to. Then I loaded the Blink example sketch via the ISP which worked as expected.

To test the anode drive alternatives, I have soldered only one anode transistor in place on the A segment. On six more I have jumpered the segment signal from the ATmega directly to the LED anode.

The last LED anode on the G segment I have jumpered directly to 5V. The idea is to see how each of the alternatives affect segment brightness.

On to Writing Software!

Before I create some I2C based "interface" to the display controller I just want to see if the multiplexing works. I therefore started with a simple sketch that multiplexes an array of 8 characters.

```
// Include libraries this sketch will use
#include "CharGenROM.h"

// Define pins used in this sketch
// debug LED
#define LED 13
// ADC0-ADC3
#define DIGIT_1  A3
#define DIGIT_2  A2
#define DIGIT_3  A1
#define DIGIT_4  A0
// ADC6, ADC7
#define DIGIT_5  A6
#define DIGIT_6  A7
// PB0, PB1
#define DIGIT_7  8
#define DIGIT_8  9

// Segments written as an 8-bit register

// Declare global variables
char banner[] = {'8', '8', '8', '8', '8', '8', '8', '8'};


void setup() {
  DDRD = 0xFF;  // Segment lines - set up Port D as outputs
  PORTD = 0x05;  //  Initialize the segments to all be off

  // Digit lines - set up the pins as outputs
  pinMode(DIGIT_1, OUTPUT);
  pinMode(DIGIT_2, OUTPUT);
  pinMode(DIGIT_3, OUTPUT);
  pinMode(DIGIT_4, OUTPUT);
  pinMode(DIGIT_5, OUTPUT);
  pinMode(DIGIT_6, OUTPUT);
  pinMode(DIGIT_7, OUTPUT);
  pinMode(DIGIT_8, OUTPUT);
  // Initialize the digits to all be off
  digitalWrite(DIGIT_1, LOW);
  digitalWrite(DIGIT_2, LOW);
  digitalWrite(DIGIT_3, LOW);
  digitalWrite(DIGIT_4, LOW);
  digitalWrite(DIGIT_5, LOW);
  digitalWrite(DIGIT_6, LOW);
  digitalWrite(DIGIT_7, LOW);
  digitalWrite(DIGIT_8, LOW);
}
```

```
void loop() {
    digitalWrite(LED, HIGH);  // just a simple debug indicator
    displayFrame();
    digitalWrite(LED, LOW);  // The LED should be pulsing once for each
frame
}


void displayFrame(){
  // display the 8 chars of the banner array
    PORTD = ~chargen[banner[0]];
    digitalWrite(DIGIT_1, HIGH);
    delay(3);
    digitalWrite(DIGIT_1, LOW);

    PORTD = ~chargen[banner[1]];
    digitalWrite(DIGIT_2, HIGH);
    delay(3);
    digitalWrite(DIGIT_2, LOW);

    PORTD = ~chargen[banner[2]];
    digitalWrite(DIGIT_3, HIGH);
    delay(3);
    digitalWrite(DIGIT_3, LOW);

    PORTD = ~chargen[banner[3]];
    digitalWrite(DIGIT_4, HIGH);
    delay(3);
    digitalWrite(DIGIT_4, LOW);

    PORTD = ~chargen[banner[4]];
    digitalWrite(DIGIT_5, HIGH);
    delay(3);
    digitalWrite(DIGIT_5, LOW);

    PORTD = ~chargen[banner[5]];
    digitalWrite(DIGIT_6, HIGH);
    delay(3);
    digitalWrite(DIGIT_6, LOW);

    PORTD = ~chargen[banner[6]];
    digitalWrite(DIGIT_7, HIGH);
    delay(3);
    digitalWrite(DIGIT_7, LOW);

    PORTD = ~chargen[banner[7]];
    digitalWrite(DIGIT_8, HIGH);
    delay(3);
    digitalWrite(DIGIT_8, LOW);
}
```

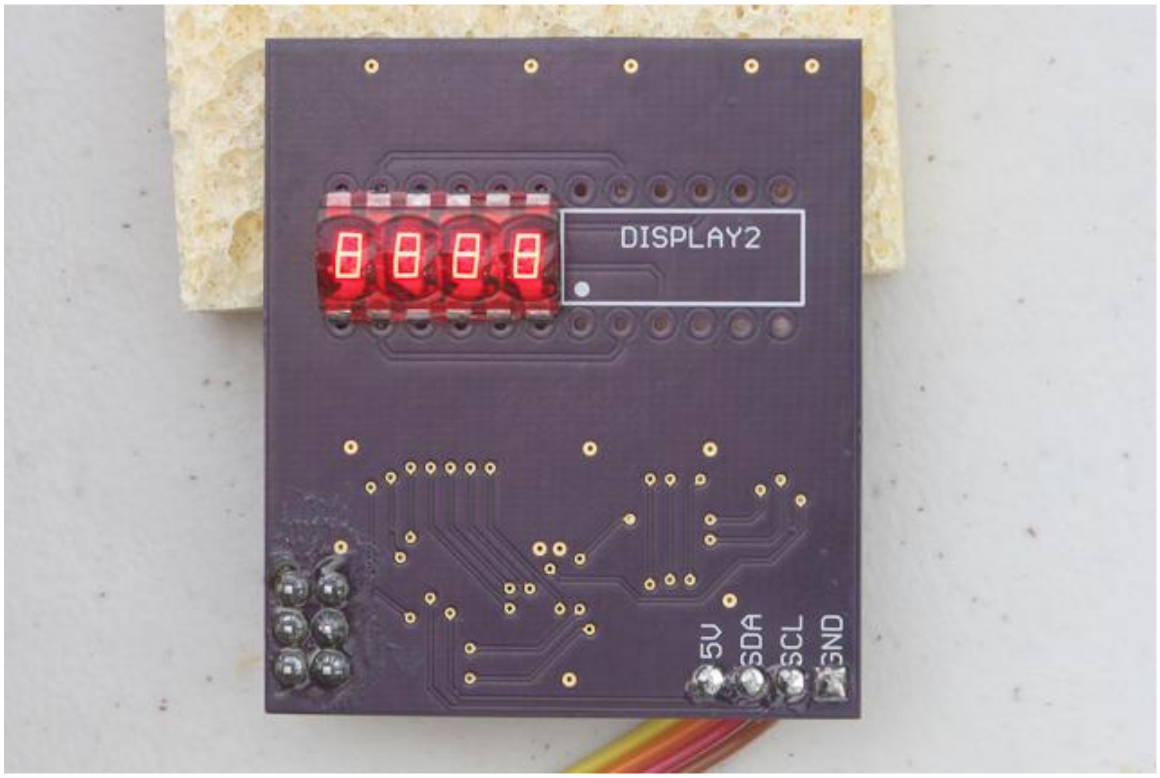
The sketch starts by including CharGenROM.h which is a definition of the segments lit for each ASCII character. Next the pins and the display message are defined.

In setup() the pins are initialized. The segments are all programed at the same time so it is simplest and fastest to control them as a single port (port D) rather than as individual pins. That's also why all the segments were wired to the same port in the hardware. The digit lines are a bit more spread out, plus they are controlled individually for timing so setting them up as individual pins makes sense. It also covers up the labeling error I made on the schematic. (exercise for the reader to find it)

In loop() the routine displayFrame() is called repeatedly. The call is surrounded by calls to digitWrite() which set the LED on D13 to be high while the frame is being displayed and low again afterwards. This is just simple debug mechanism to indicate the loop is repeating.

In displayFrame() each ASCII character is read from the banner array, translated into the appropriate segments and sent to the segment lines on port D. Then the control line for that digit is pulsed for 3 milliseconds. This is repeated for all eight digits. I did not put the digit code into a loop because I don't want the loop timing overhead and because I know I have plenty of code space since this is a dedicated display controller.

Here then is the display running the code.



So the outcome of the experiment is quite interesting. Although the G segment limit resistor is tied directly to the supply it does not appear to be any brighter (okay maybe it is slightly brighter) than the others. The segments driven directly from the ATmega do not appear to be any dimmer than either the G segment or the A segment driven by the transistor.

The following measurements were made while powering the display from the Pocket AVR Programmer. The measured supply voltage is 4.8V. The drop across the anode drive transistor is nearly 0V.

At top of limit resistor

- G: 5V segment measures 4.8V (same voltage measured at the supply)
- F-B: Direct drive segments measure 4.0V
- A: Transistor drive segment measures 4.8V

At anode of segment

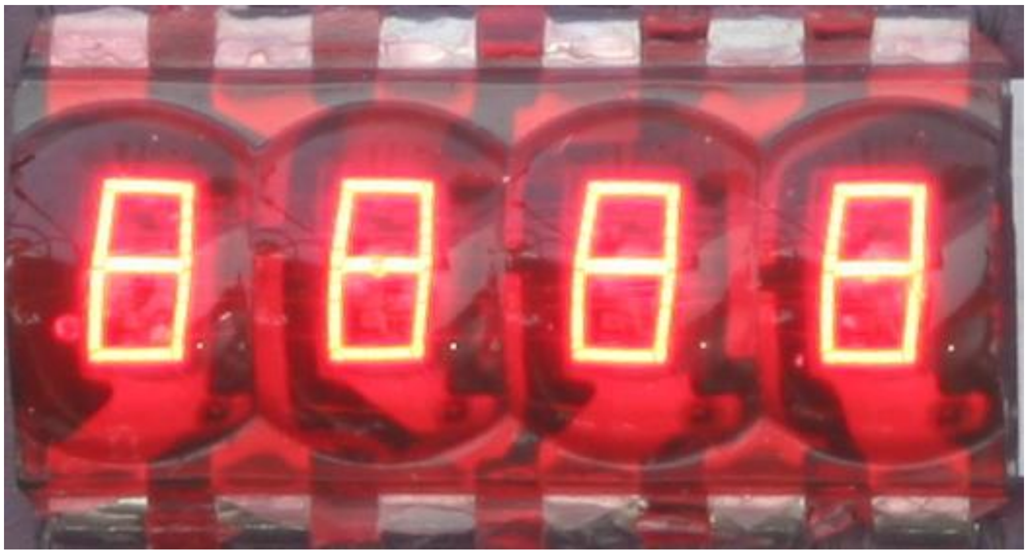
- G: 5V segment measures 1.6V
- F-B: Direct drive segments measure 1.6V
- A: Transistor drive segment measures 1.6V

At cathode of segment

- G: 5V segment measures 0V
- F-B: Direct drive segments measure 0V
- A: Transistor drive segment measures 0V

The calculated current through segments A and G would be the same at (4.8-1.6)/75 = 0.0427 or about 43mA. For the other segments, the current would be (4.0-1.6)/75 = 0.032 or 32mA. There is therefore a real 11mA difference when driving the segments from the ATmega.

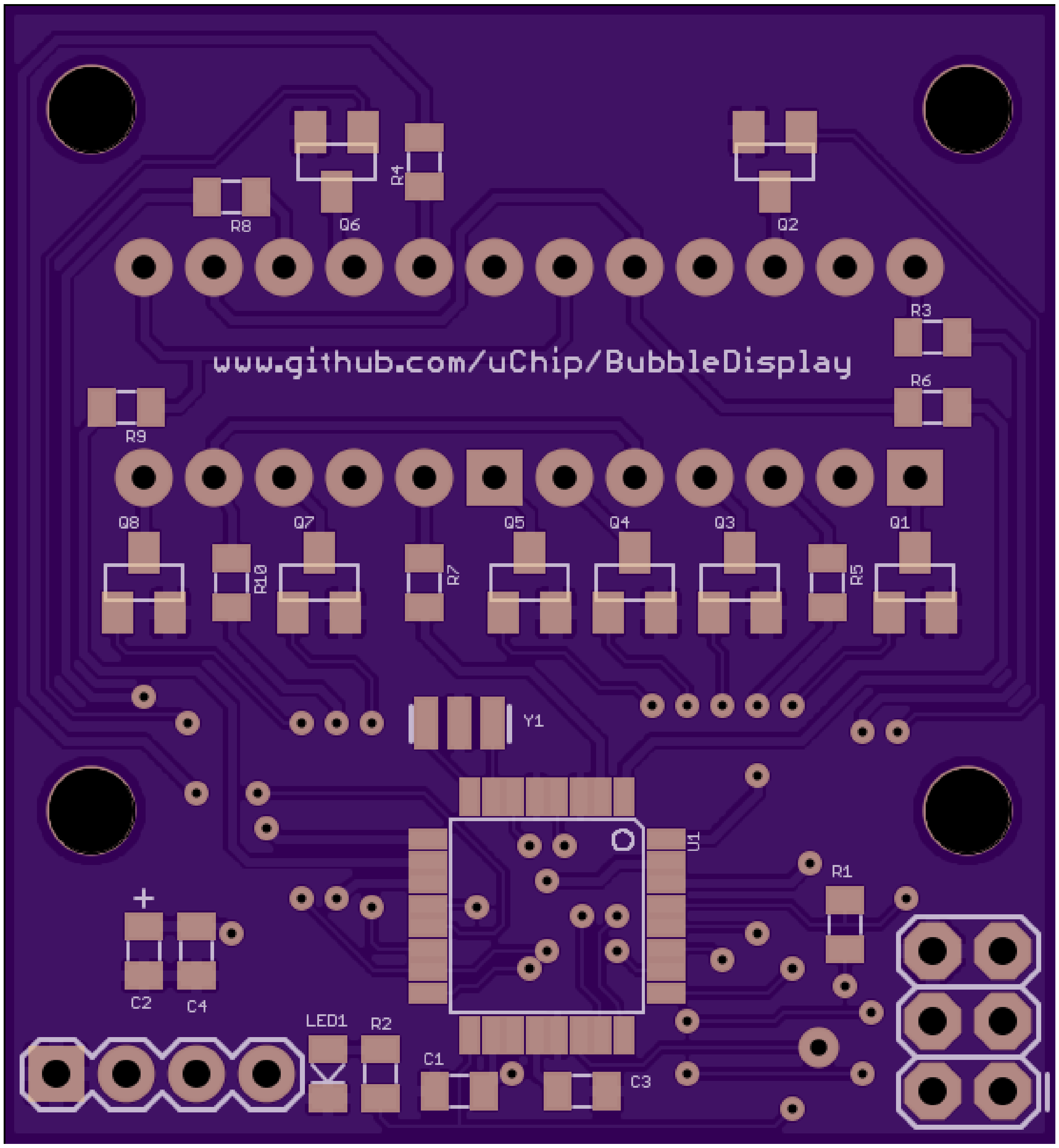
However the picture does not show an appreciable difference in brightness. Below is a blow-up of the display controller picture.



In this blow-up it does appear to me that the A and G segments are slightly brighter than the others. I think perhaps that the light output is not linear with current flow. It looks like the LEDs have "saturated" from a light standpoint. In any event, the display is distinctly brighter than when driving both anodes and cathodes from the ATmega or when the limit resistor was a much higher value.

None of the ATmega, the cathode drive transistors or the anode drive transistor gets hot. From this I would conclude that the anodes can be driven directly from the ATmega in the next PCB version which eliminates 8 transistor parts. The conclusion is therefore; the next board revision will drive the anodes through the current limit resistor but without a drive transistor removing 8 transistors from the design.

Removing components is easier than adding them. A new board layout has been done and a board ordered from OSH Park. Here is the board rendering.



More Comprehensive Software

While I'm waiting for the new board to be fab'ed, I can still make use of the current board to continue software development.

I know that the controller can drive the display, so instead of iterating on the software design from the controller perspective I'm going to look at it from the perspective of the application processor. What I'd like to do is have the Bubble Display be programmed with an interface similar to other existing displays.

I am choosing to model the Bubble Display interface on the Arduino Liquid Crystal library. The idea is to have the main library calls be the same. Here are the calls I want to model.

> begin() - initializes the interface and ready's the display
> clear() - deletes existing display data and homes display scroll
> home() - display from the beginning of the display buffer
> setCursor(posn) - sets the data entry cursor to posn (only one param since this is only a one line display)
> print(**) - very powerful and useful feature that is inherited. ** lots of parameter options.
> scrollDisplayLeft() - characters move to the left
> scrollDisplayRight() - characters move to the right
> display() - enable the display
> noDisplay() - disable the display
> createChar(which, data) - replace any ASCII char with a custom definition
> write(data) - write one char to the current cursor position then increment the cursor position.

The display shall be an eight-character window onto a 256 character buffer. By default the display shows the first eight characters of the buffer. Scroll commands move the window within the buffer with limits that constrain the display to only show buffer contents. The home call returns the window to the first eight characters.

The data is written into the display buffer at the cursor location. The cursor starts at location zero and is incremented with each character written. The write call sends a single character of data to be written. The print call can send multiple characters as formatted numbers or strings. The setCursor call moves the cursor location without writing any data. The clear call fills the buffer with spaces and resets both the cursor and scrolling. The cursor is always constrained to be between 0 and 255.

The display and noDisplay calls enable and disable the display refresh respectively.

The controller interprets display buffer contents as ASCII coded characters. There is a character generator table that maps the ASCII to the appropriate segments to light up. For example the character '3' lights up segments A,B,C,D&G. Not all ASCII characters can be represented in eight segments. The character generator is often an approximation for the character. For some characters no approximation seems appropriate. Those characters show as blanks. The createChar call allows the application to reprogram any character in the character generator table.

I've implemented one more call in my library that is not in the Liquid Crystal lib. It is testConnection. All of the library calls above have no return value. The testConnection returns a boolean. It returns true if the display exists on the bus and false otherwise. I have found this call to be useful in prototyping and debugging other I2C devices I have done. Thanks to Nick Gammon for creating it.

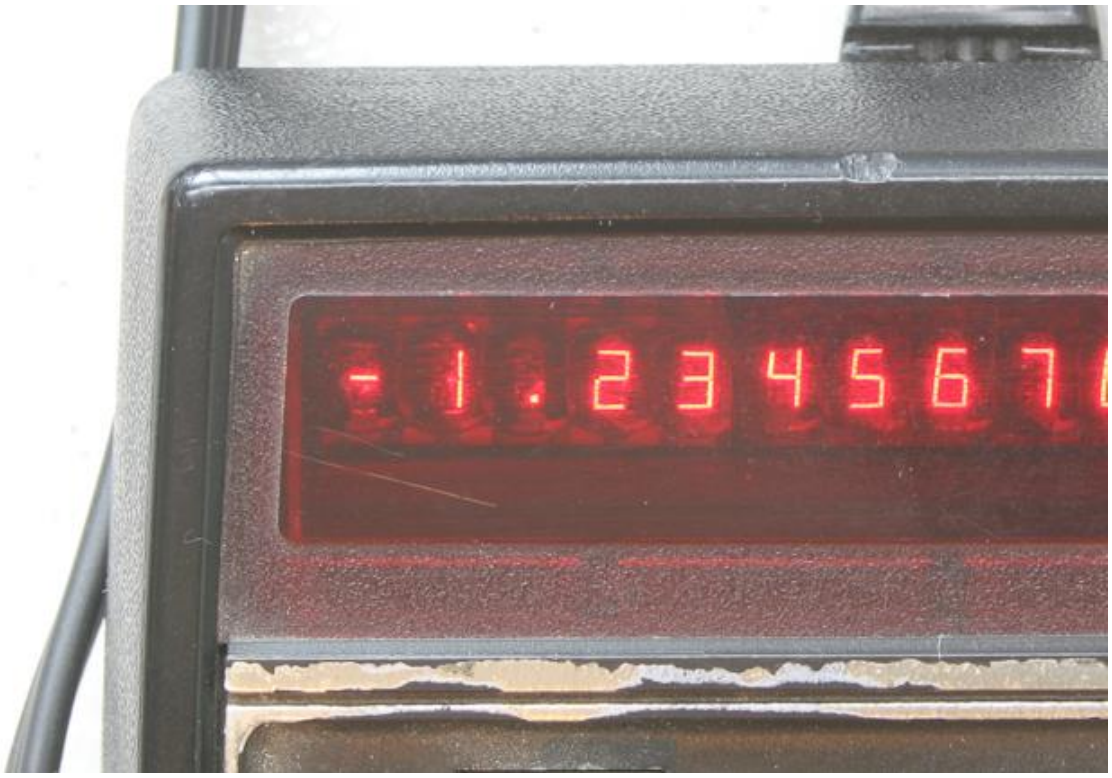
Now that the library is defined it is just a matter of coding it up. Just like the library definition, I modeled the coding on the Liquid Crystal library as well. Thank you Open Source! It makes the job so much easier and probably higher quality than I would have defined on my own. Basically each of the commands is defined as a byte value that is passed over I2C to the display controller. Where parameters are needed, they too are passed as byte values over I2C.

The contents of the library folder should be copied into your Arduino/libraries folder. See the tutorials for adding Arduino libraries for more info.

The most recent versions of these and all the files should always be available in the github repository for this project (www.github.com/uchip/BubbleDisplay).

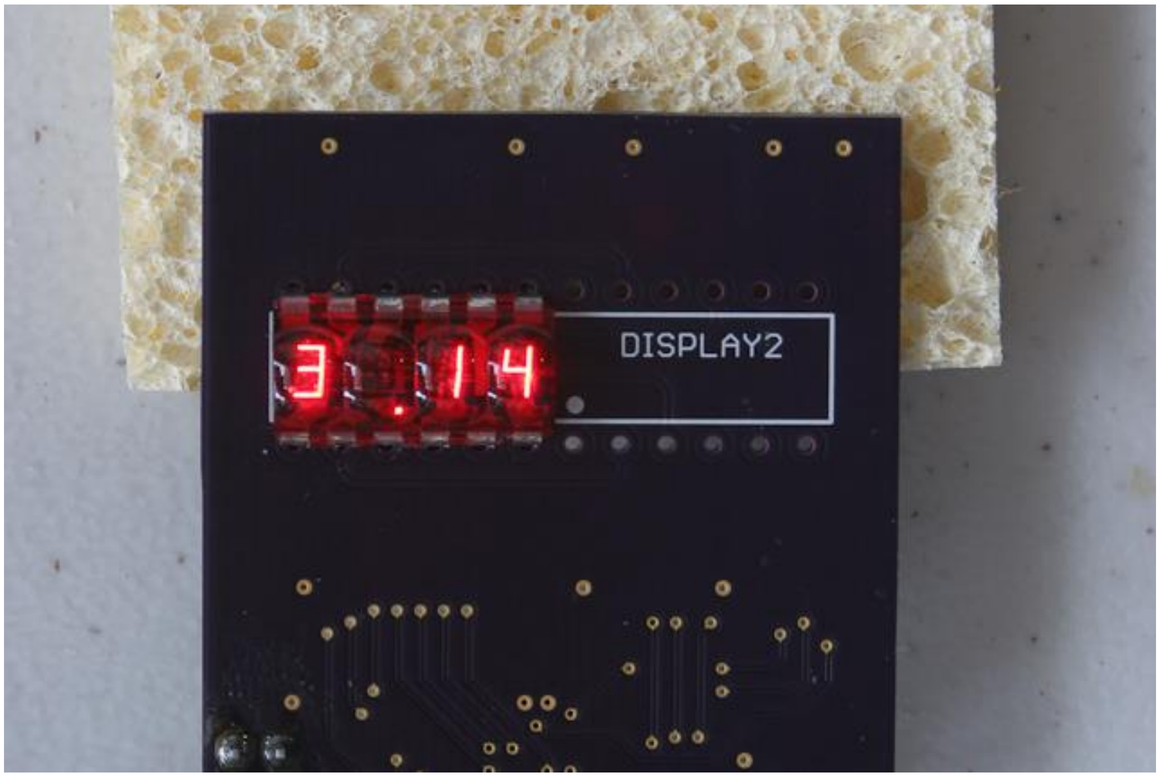
SIDEBAR: Consider the Lowly Dot

One of the most challenging design problems for this project is in regards to the dot, the decimal point. Take a look again at the old *hp* calculator display in the photo below.



Notice that the dot is placed inside the box formed by segments C,D,E&G. This puts the decimal above the character baseline and more-or-less centered horizontally.

Now look at the dot in the Bubble Display.



That dot is placed below and to the right of the segments. It is below the character baseline and far enough outside the segments that it is almost but not quite between characters. It is this placement that poses the challenge.

In our software oriented design the display takes an array of ASCII characters (even non-numeric ones) and tries to display them in seven segments plus this dot. But about the only time the dot is used is when the ASCII character is the decimal point which in our software definition is a separate character.

If we were building a simple calculator we might custom craft our binary-number-to-display-segment-conversion to overlay the decimal on top of the appropriate digit. For example, to display 3.1415926 the decimal should be combined with the 3 to display correctly. On our 8 character display we would be able to show 8 digits plus place the decimal. But when we send the same information as an ASCII representation we get 9 separate characters.

We could try to be smart and combine the decimal into the character beside it. It would be something like;

- use the character at location N to look up segments
- if the character at location N+1 is a decimal then OR in the DP segment
- show the segments
- increment N, if the character at N is a decimal then increment N again

The problem with this code is that it really complicates (read as slows down) the multiplexing code and complicates the calculations for scrolling and cursor placement. It is telling that even *hp* decided to use a whole character for the decimal in the hp35.

But with the dot placed so far to the side, showing the decimal as its own character doesn't look right either.

At this point in time I think the priority is to get the basic display functions working. We can always come back and experiment with code optimizations later. So for now the code will treat the decimal as a separate character. If anyone has ideas for how this might be improved feel free to chime in.

SIDEBAR: File Organization

There are a growing number of source files, hardware design files, documentation files and project management files associated with this project. They are all available in the project repository on github. I believe that a description of the file organization might be useful. Plus there is one file that I think folks might find confusing so I want to explain that.

There are currently four folders of files.

- examples <- stuff in here gets put in your sketches folder
- firmware <- stuff in here gets put in your sketches folder
- hardware <- stuff in here gets put in your eagle project folder
- library <- stuff in here gets put in your Arduino library folder

What documentation there is currently is at the top level (peer level to the folders).

The hardware folder contains the EagleCAD design files. There are files for the schematic, the board and the Bill of Materials (BOM).

The firmware folder contains the files needed to build the code that goes into the display controller. Once the code is loaded into the display you can ignore the firmware folder and its contents.

The library folder contains the files that create a set of calls that interface to the display. This code will run on the application processor. The library code does the "behind the scenes" work of allowing the application processor to communicate with the display processor thru a set of defined commands and parameters sent over the I2C bus.

The examples folder contains sketches (applications) that demonstrate or test the calls in the library and indirectly the display itself. The examples folder also contains the source code from the earlier experiments. It's a bit messy right now. At some point in the future I will clean it up. Once that happens the examples folder will get moved into the library folder where it belongs.

There is one file you need to build the firmware that is not in the firmware folder. That file is the BubbleCommands.h file in with the library files. If it is needed, why is the file not in with the firmware source? We want to make sure that the application processor and the display processor talk the same language. We do that by defining the commands in one place. That way we are ensured that both processors are looking at the identical "dictionary" because there is one and only one copy. For the same reason, that is why the definitions cannot be combined into the respective .h files. Since the firmware is built infrequently and the application sketches built more frequently, BubbleCommands.h needs to live with the library files. As long as the library is installed correctly the firmware sketch will find BubbleCommands.h and should build without problems. If firmware does not build, you can copy BubbleCommands.h into the sketch folder for the firmware, just don't change it!

When the display is built and its firmware code programmed in, the only folder of interest in [i]using[/i] the display is the library folder. The contents of the library folder get copied into your (for Windows, make the appropriate translation Mac and Linux folks) MyDocuments/Arduino/libraries folder. You won't need to make any changes to these library files, but they need to be there to use the library in your application sketch.

I'll post again if folder definitions change radically from this description.

Display Controller Firmware

Since the library is coded, the commands that can be sent to the display controller are now defined. The next task is to program the display controller to accept and respond to the commands. Each library call sends a command across the I2C bus to the display controller. The commands are single bytes defined in the shared file BubbleCommands.h stored with the library source. (See the previous posting **SIDEBAR: File Organization**.) A few commands need one or two parameters. Each parameter is also a single byte so the entire command message can be 1, 2 or 3 bytes in length.

According to the I2C bus protocol the bus Master (in this case the application processor) sends a packet across the bus starting with the address of the slave, followed by each byte of the message. The Slave (that would be the display controller processor in this case) acknowledges being addressed and having received each subsequent byte of the message.

On the Arduino the I2C bus protocol is easily handled for you by a library called Wire. The Wire library was used in coding up the BubbleDisplay library and will be used again here in the display controller firmware.

The main function of the display controller, multiplexing the display, was already coded and demonstrated in the earlier posting. The refresh code has been modified from the earlier version. Controls have been added to allow scrolling through a larger data buffer and to enable/disable the digit drive. Here is one digit of the new version of the multiplexing code inside Arduino's loop() routine.

```
void loop() { // Do this over and over

  // Display 8 chars of the displayBuff starting at the scroll posn

  // Take the character value stored in the displayBuff at index scroll
and
  // use that value to look up the segment values in the character
generator
  // table, chargen.  Send the segment values to the segment lines,
PORTD.
  PORTD = chargen[displayBuff[scroll + 0]];

  // Only pulse the digit line (cathode) high if the
  // display is enabled (the enable variable is true)
  // The digit control line is pulsed high to turn on the transistor
driver which
  // pulls the digit cathode low lighting the segments.
  if(enable) digitalWrite(DIGIT_1, HIGH);
  // wait out the time to show this digit
  delay(DIGIT_TIME);

  // always turn the digit off again even if the display is not enabled
  digitalWrite(DIGIT_1, LOW);

  // Now repeat the same steps for each of the other digits in the 
display
  .
  .
  .
}  // End of loop().  Finished with multiplexing the display one time
```

What needs to be added now is the code that listens for commands on the I2C bus and takes appropriate action based on which command was received.

The Wire library takes care of listening on the I2C bus. When Wire is initialized it is given a function to call when bus data arrives. In the display controller source there is a function called receiveEvent() that processes the bus data. In setup() the line

```
Wire.onReceive(receiveEvent);
```

tells Wire to call receiveEvent() when the data comes in. The receiveEvent() call stores the message into a short array and then enters a switch statement. The switch executes different code depending on the value of the command. For most commands only a single line of code is needed to modify a variable which changes the behavior of the display refresh or saves data. A few commands require multiple lines of code because they need parameter bounds checking but these are still short enough that it doesn't make sense to call out to a separate routine. Here is the receiveEvent() code.

```
// This function executes whenever data is received from I2C bus
// addressed for this device.
// Function is registered as an event, see setup()
void receiveEvent(int howMany)
{
  // We never expect to receive commands plus parameters longer than 3 
bytes.
  // Since this is a dedicated controller about the only way to recover
  // from the error is to start all over again.
  if(howMany > 3) reboot();

  // read the data into the array for processing
  for(int i=0;i<howMany;i++){
    rcvArray[i] = Wire.read(); // receive byte as a character
  }

  // based on the value of the first byte, the command, decide
  // which code to execute.
  switch(rcvArray[0]){
  case LED_CLEARDISPLAY:
    // reinitialize all display controls to their default values
    clearDisplay();
    break;
  case LED_HOMECURSOR:
    // the next received data will be stored in displayBuff at index 0
```

```
      cursor = 0;
      break;
    case LED_HOMESCROLL:
      // start displaying characters from displayBuff starting at index 0
      scroll = 0;
      break;
    case LED_DISPLAYON:
      // allow the digit lines to be pulsed
      enable = true;
      break;
    case LED_DISPLAYOFF:
      // do not pulse the digit lines
      enable = false;
      break;
    case LED_SCROLLLEFT:
      // Although the index increases (goes right) the characters
      // appear to have shifted left
      scroll++;
      if(scroll>256-8) scroll = 256-8; //don't go past the end
      break;
    case LED_SCROLLRIGHT:
      // Although the index decreases (goes left) the characters
      // appear to have shifted right
      scroll--;
      if(scroll>256-8) scroll = 0; //this would only be true if
      // scroll wrapped when decremented past 0
      break;
    case LED_SETCURSOR:
      // Allow the application to set where the next character goes.
      // Since the parameter is a byte, there are no illegal values.
      cursor = rcvArray[1];
      break;
    case LED_SETSCROLL:
      // Allow the application to set the scroll position, but
      // not higher than 8 from the end of displayBuff.
      scroll = rcvArray[1];
      if(scroll>256-8) scroll = 256-8; //don't go past the end
      break;
    case LED_CREATECHAR:
      // The application can override any character generator definition.
      // Valid characters are 0 to 127 so constrain the parameter to
  that.
      // The new definition will persist until the display controller is
      // restarted.
      chargen[rcvArray[1] & 0x7F] = rcvArray[2];
      break;
    case LED_DATA:
      // New data to be displayed.  Valid values are 0 to 7F (0-127).
      // If more data is provided than the buffer will hold, the cursor
      // automatically wraps back to the beginning.
      displayBuff[cursor++] = rcvArray[1] & 0x7F;
      break;
    case LED_RESTART:
      // Resets the display controller.
      reboot();
      break;
    default:
```

```
    // Invalid command.  Reset the display controller to try to
recover.
    reboot();
  }
}
```

The firmware source is in Display_Firmware.ino. You can find that in the firmware folder of the github repository.

The character generator data table has been around for a while. I got it from SparkFun who got it from the Arduino playground. It was originally written by Dean Reading. The table translates ASCII codes 0 thru 127 into 8 1-bit segment values (7-segments plus decimal point). The table is a list of segment values. The ASCII code acts as an index into the table to pick the right one. I rewrote the table to turn the segment bits around to match my hardware. I added the decimal point (dp) segment and tweaked a couple of character definitions. Here is a snippet of the table showing a few characters. The whole table file is in the firmware folder.

```
// Character generator for 7-segment displays
uint8_t     chargen[] = {
//  dpGFEDCBA  Segments
  .
  .
  .
  0b01110111, // 65 'A'
  0b01111100, // 66 'B'
  0b00111001, // 67 'C'
  0b01011110, // 68 'D'
  0b01111001, // 69 'E'
  0b01110001, // 70 'F'
  .
  .
  .
  0b00000000, // 87 'W'  NO DISPLAY
  .
  .
  .
```

A "1" for a segment value means that segment will be lit. Some characters, "W" for example, cannot be drawn on a 7 segment display. Those characters are just left as blank (no segments lit).

A quirk of the ATmega processor (separate program and data address spaces) causes data tables like this to be copied into RAM when the program starts. For this application, copying into RAM has the advantage of allowing the table to be updated dynamically. The createChar() command overwrites the data in the table with new data from the application processor. That data will stay in RAM until the display controller loses power or is rebooted. Upon restart the unmodified table is again copied from the program space into RAM.

That's about it for the display controller side. The library is also done. In coding it up I added one additional command, restart(). I was getting tired of pulling power to restore the character generator after modifying it so I added the command to do it from software. I have an idea for one more command, but I'll leave that for a future posting.

There is one more bit of code to write. That is the application sketch code that will verify that all the commands work. That "all in one" might be too much to serve as a clear example so there might be a couple more simple example sketches as well. I'll work on those and post separately.

I have also received notification from OSH Park that the next rev of the PCB is back from fab. It should be here in a couple of days.

Test & Example Sketches

The examples folder has been moved inside the BubbleDisplay folder inside the library folder. This move will have the examples show up in the Arduino IDE Examples menu. The sketches are pretty simple and reasonably well commented so I won't go into great detail here. There are ten examples. Three of these are new code written by me.

- BubbleTest
- ScrollingBanner
- BubbleMeter

The BubbleTest sketch uses all of the library calls. For most of the calls there is a visual change to the display. If you upload the sketch and open the serial monitor the text in the monitor tells you what to expect on the display. If the serial monitor text and the display match then the system is working.

The ScrollingBanner sketch uses the scrollDisplayLeft() and scrollDisplayRight() calls to move a string of text back-and-forth across the display.

The BubbleMeter sketch measures a voltage on Arduino pin A0 and coverts the measurement into a floating point number. It then uses print() to present that number on the display, effectively building a simple voltmeter.

One of the reasons for modeling the BubbleDisplay library on the existing LiquidCrystal library is to make it easy to reuse code between the two. To test this idea I took seven of the examples from the LiquidCrystal library and ported them to BubbleDisplay.

- Blink
- CustomCharacter
- Display
- HelloWorld
- Scroll
- SerialDisplay
- setCursor

Here is an example (somewhat tongue-in-cheek) of the changes made to the HelloWorld.ino example. Stuff to be removed has been lined out. Stuff to be added is colored blue. I left variable names unchanged so even though this is now written for the LED display it is still referred to as "lcd". Notice that the majority of the changes are to the comments.

The code works within the physical limits (8 characters) of the display. Instead of showing “`hello, world!`” the display shows “`hello   w`”. (the comma character in chargen is blank) Then the seconds counter overwrites the last few characters.

```
/*
  ~~LiquidCrystal~~ BubbleDisplay Library - Hello World

 Demonstrates the use ~~a 16x2 LCD display~~ hp BubbleDisplay.  ~~The LiquidCrystal~~
~~ library works with all LCD displays that are compatible with the~~
~~ Hitachi HD44780 driver. There are many of them out there, and you~~
~~ can usually tell them by the 16-pin interface.~~  The BubbleDisplay library
is modeled after the LiquidCrystal library and implements many of the same
library calls.  To demonstrate this, this example from LiquidCrystal has
been ported to BubbleDisplay and the changes noted.

 This sketch prints "Hello World!" to the ~~LCD~~ display
 and shows the time.

  The circuit:
~~ * LCD RS pin to digital pin 12~~
~~ * LCD Enable pin to digital pin 11~~
~~ * LCD D4 pin to digital pin 5~~
~~ * LCD D5 pin to digital pin 4~~
~~ * LCD D6 pin to digital pin 3~~
~~ * LCD D7 pin to digital pin 2~~
~~ * LCD R/W pin to ground~~
~~ * 10K resistor:~~
~~ * ends to +5V and ground~~
~~ * wiper to LCD VO pin (pin 3)~~
 * BubbleDisplay 5V pin to 5V supply
 * BubbleDisplay GND pin to GND
 * BubbleDisplay SDA pin to SDA or A4
 * BubbleDisplay SCL pin to SCL or A5

 Library originally added 18 Apr 2008
 by David A. Mellis
 library modified 5 Jul 2009
 by Limor Fried (http://www.ladyada.net)
 example added 9 Jul 2009
 by Tom Igoe
 modified 22 Nov 2010
 by Tom Igoe
 modified 19 Apr 2014
 by Chip Schnarel

 This example code is in the public domain.

~~ http://www.arduino.cc/en/Tutorial/LiquidCrystal~~
 http://www.github.com/uchip/BubbleDisplay
 */

// include the library code:
#include <Wire.h>
#include <~~LiquidCrystal~~BubbleDisplay.h>

// initialize the library ~~with the numbers of the interface pins~~
```

```
~~LiquidCrystal~~BubbleDisplay lcd(~~12, 11, 5, 4, 3, 2~~);

void setup() {
  // set up the I2C bus
  Wire.begin();
  // set up the ~~LCD's number of columns and rows:~~ display
  lcd.begin(~~16, 2~~);
  // Print a message to the ~~LCD~~display.
  lcd.print("hello, world!");
}

void loop() {
  // set the cursor to column ~~0, line 1~~5
~~  // (note: line 1 is the second row, since counting begins with 0):~~
  lcd.setCursor(~~0, 1~~5);
  // print the number of seconds since reset:
  lcd.print(millis()/1000);
}
```

Here is another example, SerialDisplay.ino. See that though this code is more complex, the number of changes is the same or less. There are no changes at all to the loop() routine code.

```
/*
  ~~LiquidCrystal~~BubbleDisplay Library - Serial Input

Demonstrates the use ~~a 16x2 LCD display~~ hp BubbleDisplay.  ~~The LiquidCrystal~~
~~ library works with all LCD displays that are compatible with the~~
~~ Hitachi HD44780 driver. There are many of them out there, and you~~
~~ can usually tell them by the 16-pin interface.~~  The BubbleDisplay library
is modeled after the LiquidCrystal library and implements many of the same
library calls.  To demonstrate this, this example from LiquidCrystal has
been ported to BubbleDisplay and the changes noted.

 This sketch displays text sent over the serial port
 (e.g. from the Serial Monitor) on an attached L~~C~~ED.

 The circuit:
~~* LCD RS pin to digital pin 12~~
~~ * LCD Enable pin to digital pin 11~~
~~ * LCD D4 pin to digital pin 5~~
~~ * LCD D5 pin to digital pin 4~~
~~ * LCD D6 pin to digital pin 3~~
~~ * LCD D7 pin to digital pin 2~~
~~ * LCD R/W pin to ground~~
~~ * 10K resistor:~~
~~ * ends to +5V and ground~~
~~ * wiper to LCD VO pin (pin 3)~~
 * BubbleDisplay 5V pin to 5V supply
 * BubbleDisplay GND pin to GND
 * BubbleDisplay SDA pin to SDA or A4
 * BubbleDisplay SCL pin to SCL or A5

 Library originally added 18 Apr 2008
 by David A. Mellis
 library modified 5 Jul 2009
 by Limor Fried (http://www.ladyada.net)
 example added 9 Jul 2009
 by Tom Igoe
 modified 22 Nov 2010
 by Tom Igoe
 modified 19 Apr 2014
 by Chip Schnarel

 This example code is in the public domain.

 ~~http://www.arduino.cc/en/Tutorial/LiquidCrystal~~
 http://www.github.com/uchip/BubbleDisplay
 */

// include the library code:
#include <Wire.h>
#include <~~LiquidCrystal~~BubbleDisplay.h>

// initialize the library with the numbers of the interface pins
~~LiquidCrystal~~BubbleDisplay lcd(~~12, 11, 5, 4, 3, 2~~);

void setup(){
  // set up the I2C bus
  Wire.begin();
  // set up the ~~LCD's number of columns and rows:~~ display
  lcd.begin(~~16, 2~~);
  // initialize the serial communications:
  Serial.begin(9600);
}

void loop()
{
  // when characters arrive over the serial port...
  if (Serial.available()) {
    // wait a bit for the entire message to arrive
    delay(100);
    // clear the screen
    lcd.clear();
    // read all the available characters
    while (Serial.available() > 0) {
      // display each character to the L~~C~~ED
      lcd.write(Serial.read());
    }
  }
}
```

Not every program has this level of reuse. CustomCharacter.ino for instance has many more changes because behaviors and character definitions are different. LiquidCrystal ADDs a custom character and BubbleDisplay REPLACEs any character definition. LiquidCrystal uses a 5x8 bit character definition compared to BubbleDisplay's 8 bit definition. Still, even if not all the code can be reused, the similar concepts make porting code easier to think about.

That's it for example descriptions. You can find all the examples in the libraries/BubbleDisplay/examples folder in the repository on github.

The new PCBs have arrived so the next step is to build one up and see how it performs.

PCB Version 2

Blah, blah, blah