



32x16 and 32x32 RGB LED Matrix

Created by Phillip Burgess



Last updated on 2014-12-15 09:00:16 PM EST

Guide Contents

Guide Contents	2
Overview	3
Powering	6
Wiring the 16x32 Matrix	10
Wiring the 32x32 Matrix	16
Wiring a double-header panel	18
Test Example Code	26
Library	29
Advanced Wiring for the 16x32	32
How the Matrix Works	34
Downloads	36

Overview

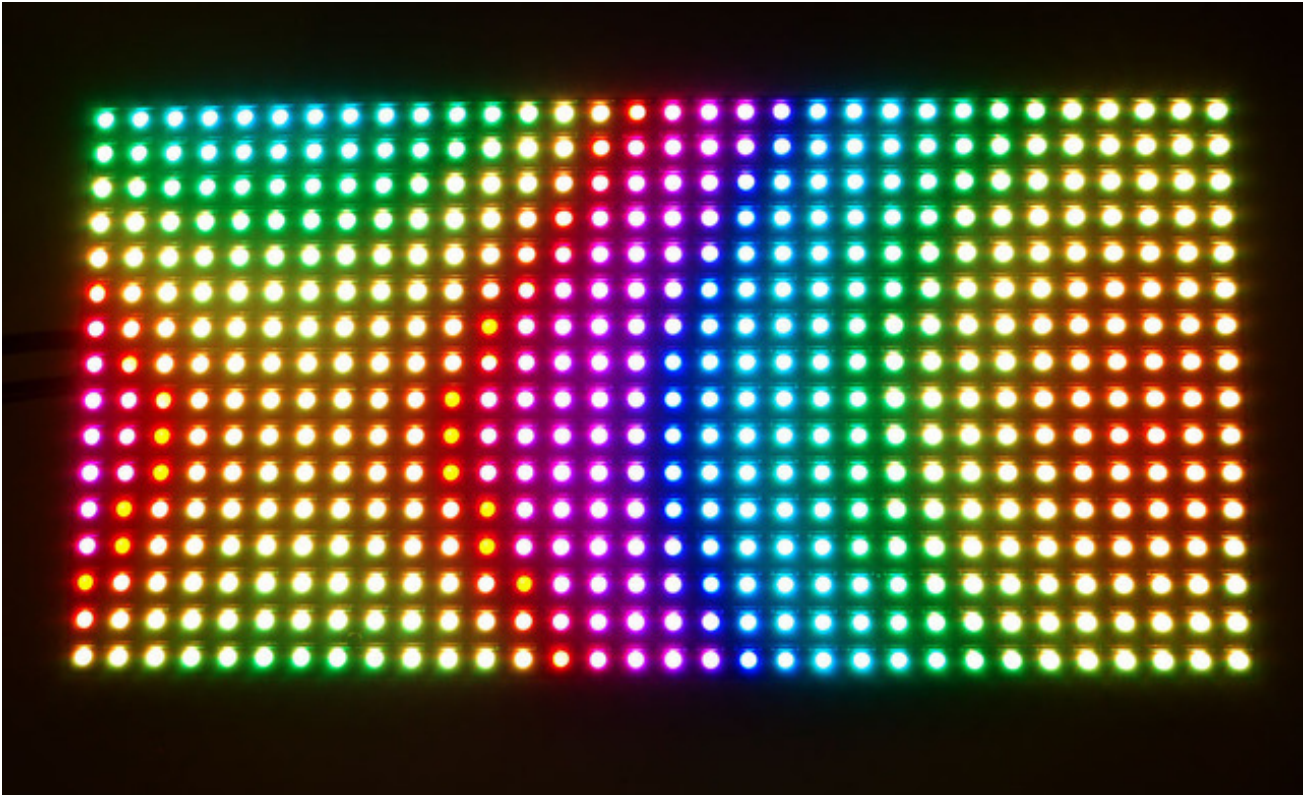
Bring a little bit of Times Square into your home with our 16 x 32 and 32 x 32 RGB LED matrix panels. These panels are normally used to make video walls – here in New York we see them on the sides of buses and on bus stops – to display animations or short video clips. We thought they looked really cool so we picked up a few boxes from the factory. One has 512 bright RGB LEDs arranged in a 16x32 grid on the front, the other has 1024 LEDs in a 32x32 grid. On the back is a PCB with IDC connectors (one set for input, one for output: in theory you can chain these together) and 12 16-bit latches that allow you to drive the display with a 1:8 (16x32) or 1:16 (32x32) scan rate.





These panels require 12 or 13 digital pins (6 bit data, 6 or 7 bit control) and a good 5V power supply, up to 2A per panel (when on full white). We suggest our 2A regulated 5V adapter and either a terminal block DC jack, or solder a jack from our DC extension cord. Please read the rest of our tutorial for more details!

Keep in mind that these displays are designed to be driven by FPGAs or other high speed processors; they do not have built in PWM control of any kind. Instead, you're supposed to redraw the screen over and over to 'manually' PWM the whole thing. On a 16 MHz Arduino Uno, we managed to squeeze 12-bit color (4096 colors) with 20% CPU usage but this display would really shine if driven by an FPGA, CPLD, Propeller, XMOS or other high speed multi-processor controller.



Of course, we wouldn't leave you with a datasheet and a "good luck!" We have a full wiring diagrams and working Arduino library code with examples from drawing pixels, lines, rectangles, circles and text. You'll get your color blasting within the hour! On an Arduino Uno or Mega, you'll need 12 digital pins, and about 800 bytes of RAM to buffer the 12-bit color image (double that for the 32x32 matrix).

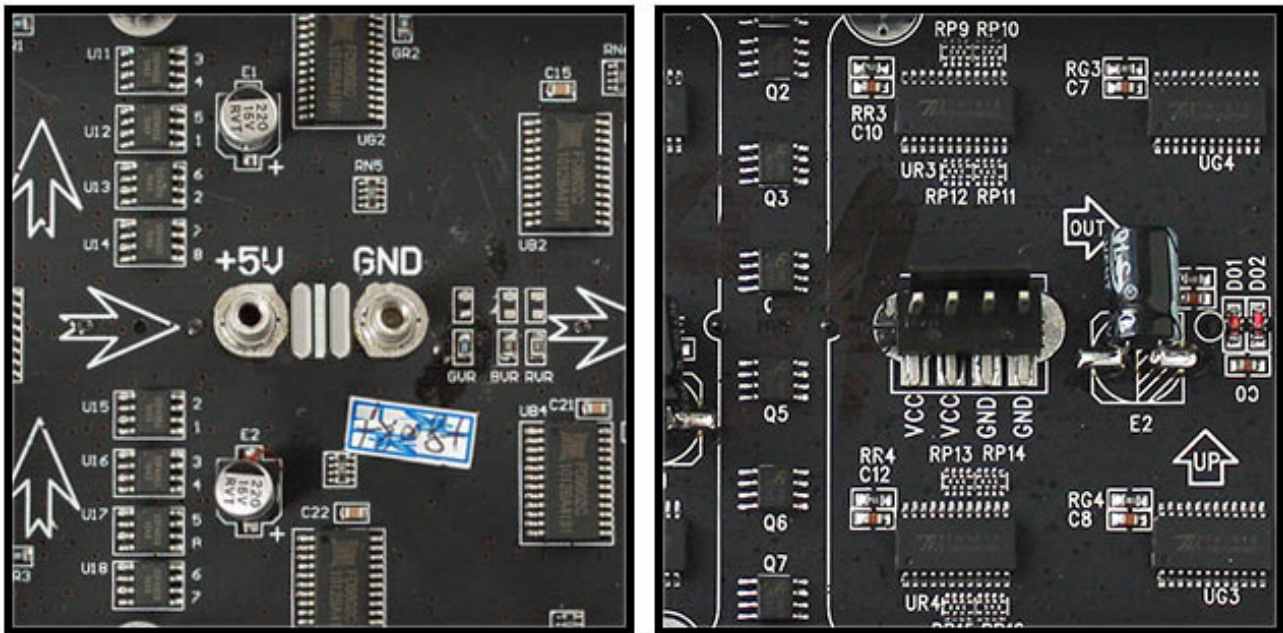
The library works ONLY with the Arduino Uno and Mega. Other boards (such as the Arduino Leonardo) ARE NOT SUPPORTED.

Powering

One handy thing about the design of these matrices is that the power cable is separate from the data connection. This makes it a lot easier to wire and you can easily give the matrix its own 5V supply (it needs up to 2A if you have all the LEDs on bright white!). Let's begin by connecting up a 5V supply...

Our parts suppliers occasionally make revisions to designs. As a result, the connections have changed over time. We'll walk through the different wiring combinations here...pick the explanation that matches the panel(s) you received.

Two different types of power connectors have made an appearance:



On the left is a screw post power connector (with adjacent pads for soldering wires directly). On the right, a Molex-style header. Some 32x32 matrices will have *two* headers...the power cable included with these panels has connectors for both headers.

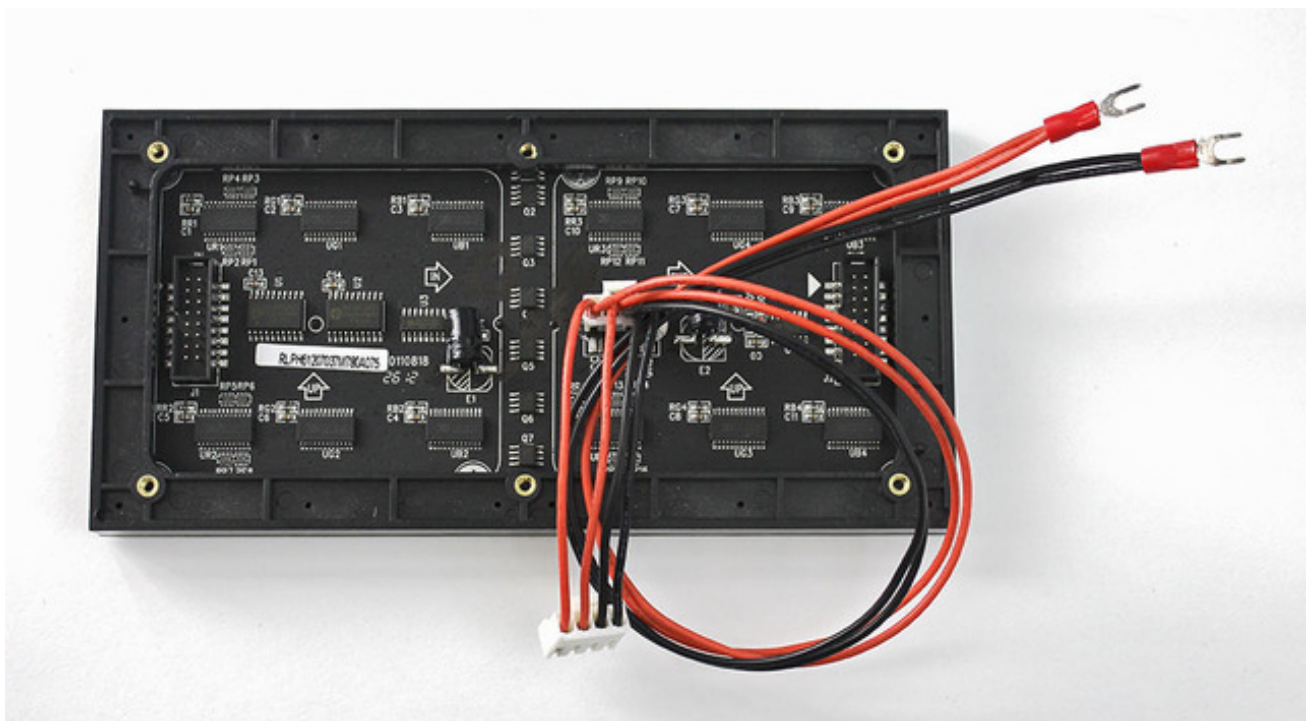
With the posts-and-pads connector, you can either screw down the spades from the power cable, or another approach is to [cut a 2.1mm jack from this extension cord](http://adafruit.it/327) (<http://adafruit.it/327>) and solder it to the pads on the panel back. [This way you can plug the 5V from a wall adapter](http://adafruit.it/276) (<http://adafruit.it/276>) right in (the one we have in the shop is suggested). Simply cut the other half of the cable off, and strip the wiring so you can solder the red wire to +5 and the black wire to ground.



Solder both pins correctly to the power port. Make sure you get this right because there is no protection diode!

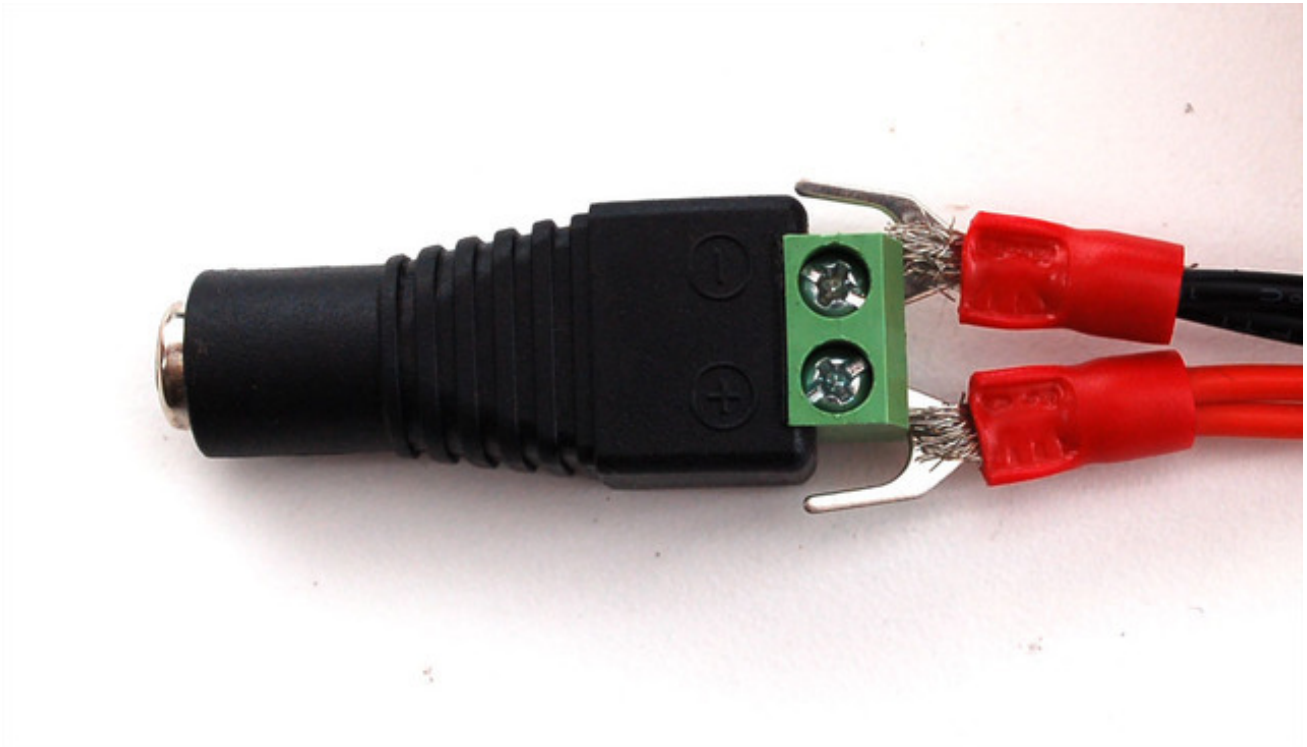


If your panel has the Molex-style header, just plug in the included power cable, observing the correct polarity.



The spades at the opposite end of this power cable can be screwed into a 2.1mm terminal block adapter. Works nicely! Don't allow the exposed connectors to contact metal though...

you may want to cover this with heat-shrink tube or electrical tape.

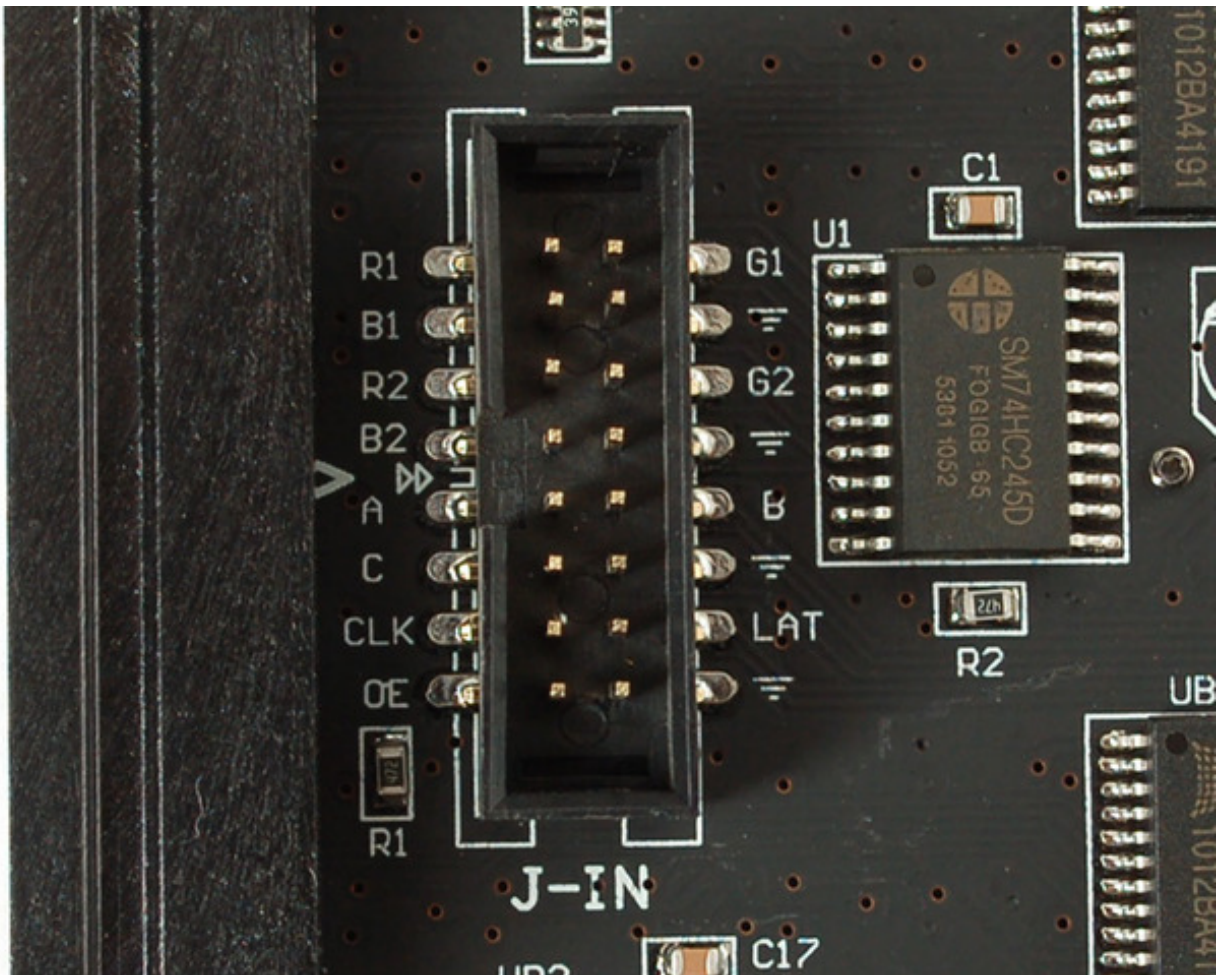


Wiring the 16x32 Matrix

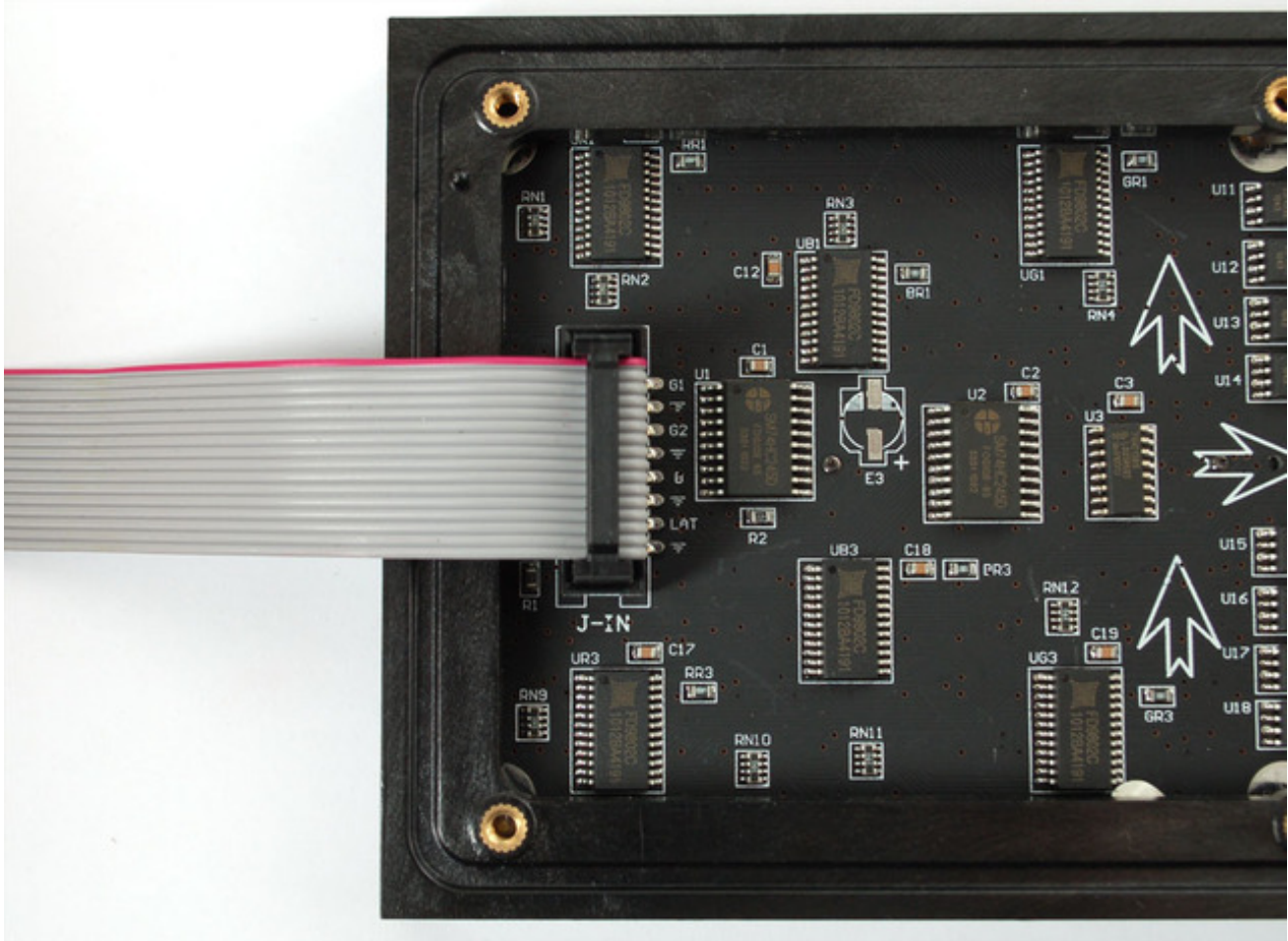
Keep in mind that this matrix was designed to be run by a 250-pin 100 MHz FPGA, not a 16-MHz Arduino. To keep the speed and color resolution somewhat reasonable, we've hardcoded part of the pinout so that the 6-pin data bus (2 red, 2 green and 2 blue) **must** be on digital pins **2** thru **7** (pins 0 and 1 are reserved for uploading/downloading). We've also set the clock pin in the library to be **digital 8**. The other pins can be configured as you'd like, but for our tutorial we'll be using analog pins **0** thru **3** and digital **9**.

There's no way to avoid needing all 6 data pins and the 6 (16x32 display) or 7 (32x32 display) control pins! (While the **OE** pin is technically not required and could be tied permanently to ground, this would cause ghosting on the display so we don't recommend it).

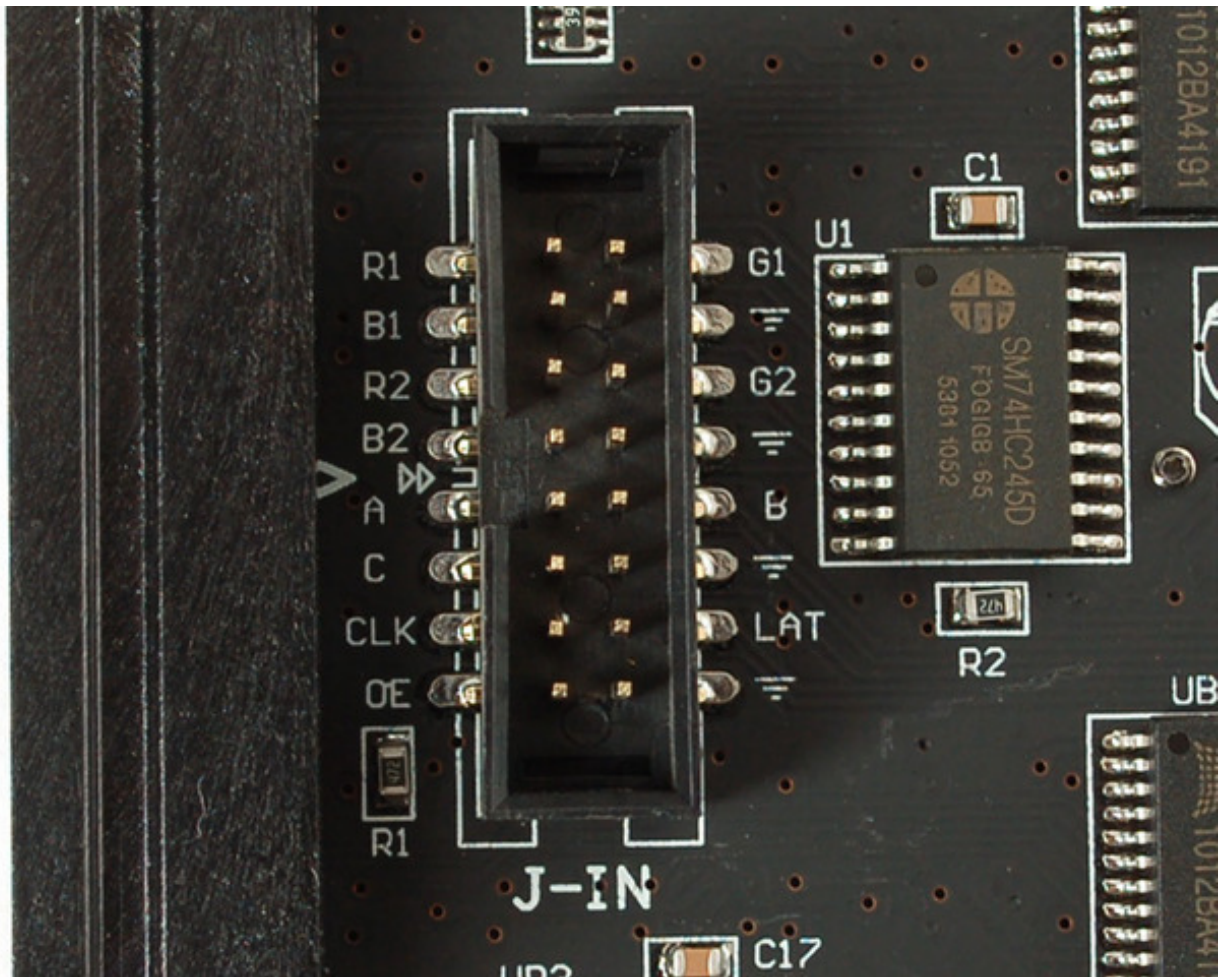
On the back of the matrix panel are two 16-pin (2x8) IDC sockets. Look for the one on the left, with the **IN** label. (The second socket on the right is an output for daisy-chaining multiple panels, but we won't be doing that here.) Not all panels include the individual labels next to the pins, but the wiring is the same regardless, so you can refer to this image in either case:



Plug the IDC cable in as shown:

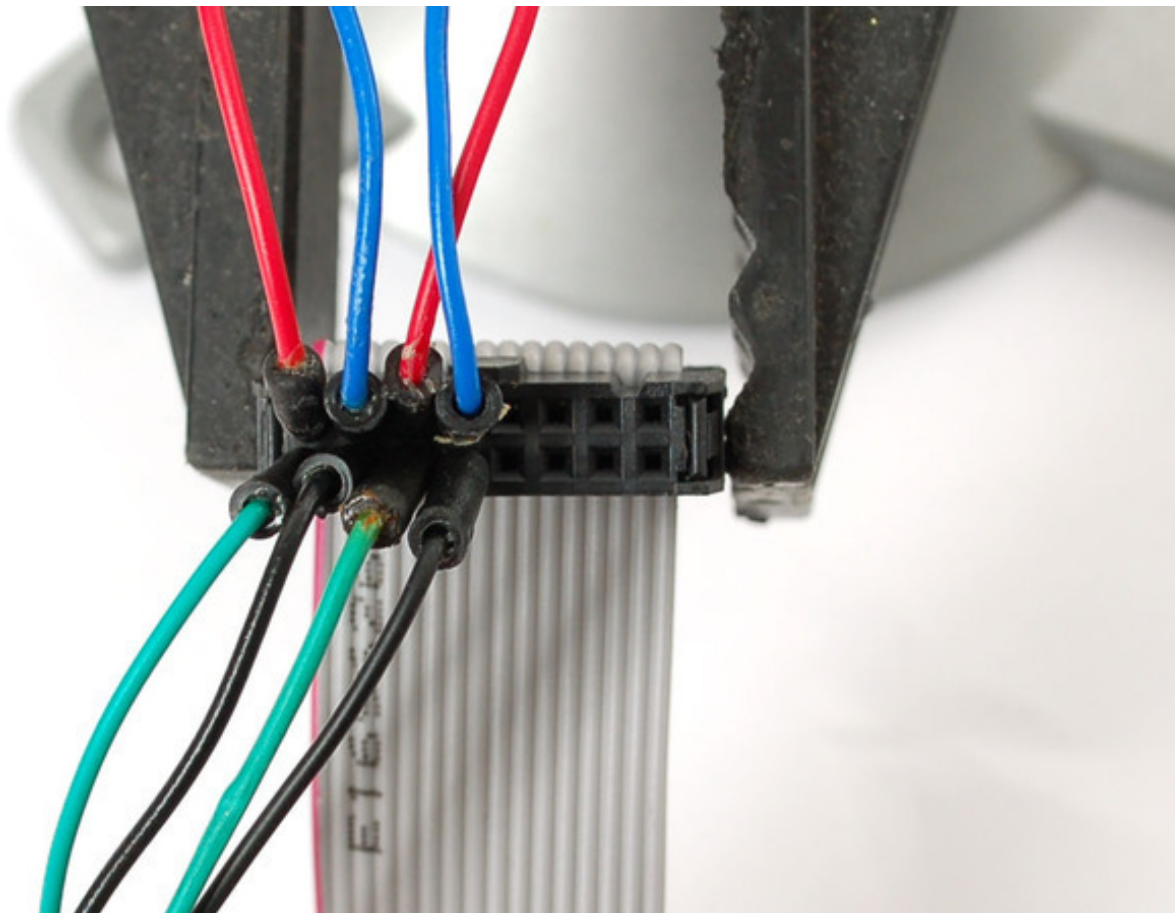


We will test the panel by free-wiring to the other side of the plug. There are **a lot of wires** so keep good track of the colors, and triple-check if it doesn't work! The good news is that even if you mess up the wiring, you won't damage the panel because the power pins are kept separate (but you could short your microcontroller pins to ground which they might not like). Start with the first 8 pins, the color data pins: R1, G1, B1, two grounds, R2, G2 and B2. The panel requires that you write to two LEDs at a time (one on the top half (R1, G1, B1) and one on the bottom half (R2, G2, B2)).

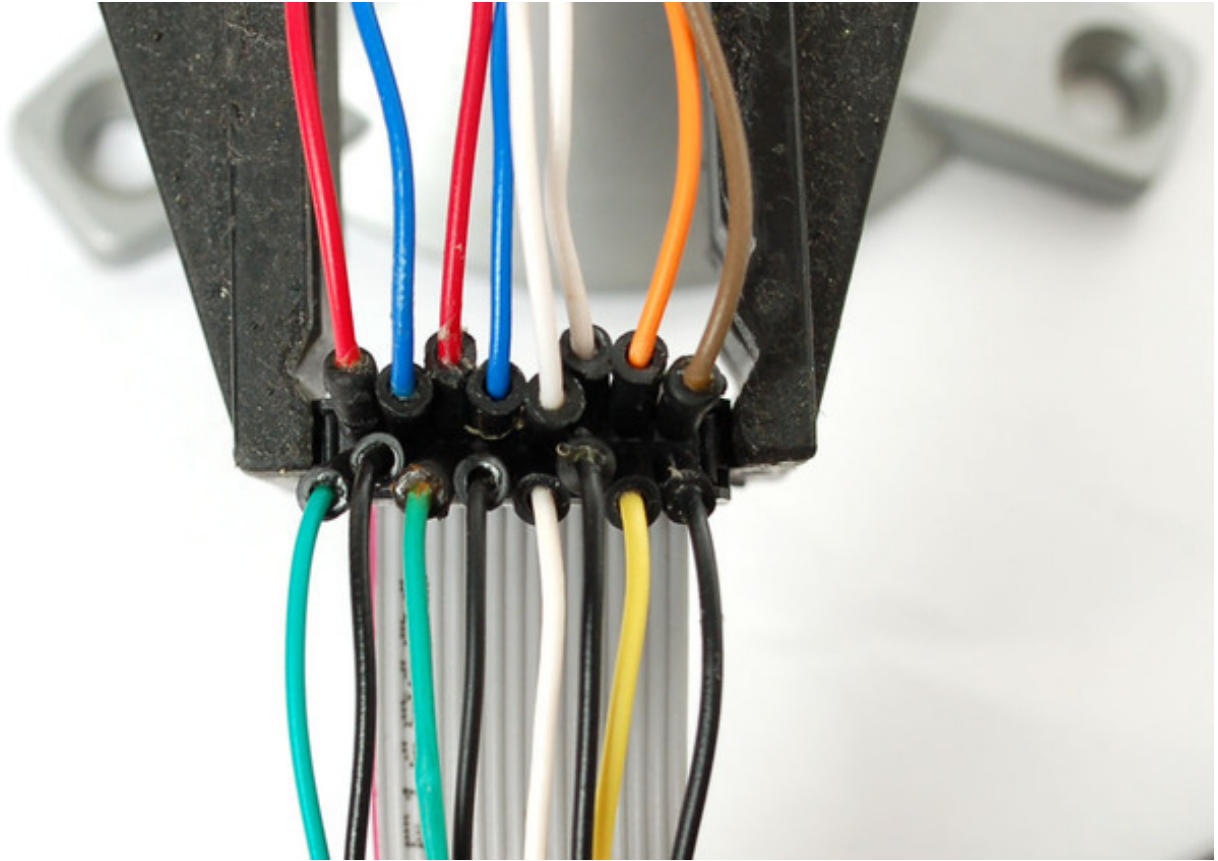


HERE'S THE IMPORTANT THING TO REMEMBER!!! The other side of the IDC cable is *flipped* when you wire it because you're looking -up- into it instead of -down- into the connector. The images below are **correct!**

We'll use red wire for the red LEDs, green for green and blue for blue - makes sense that way! Black is for the ground pins. Look carefully at how the IDC cable is placed in the vise, note where the red stripe is and the 'key' of the connector - its on the top.



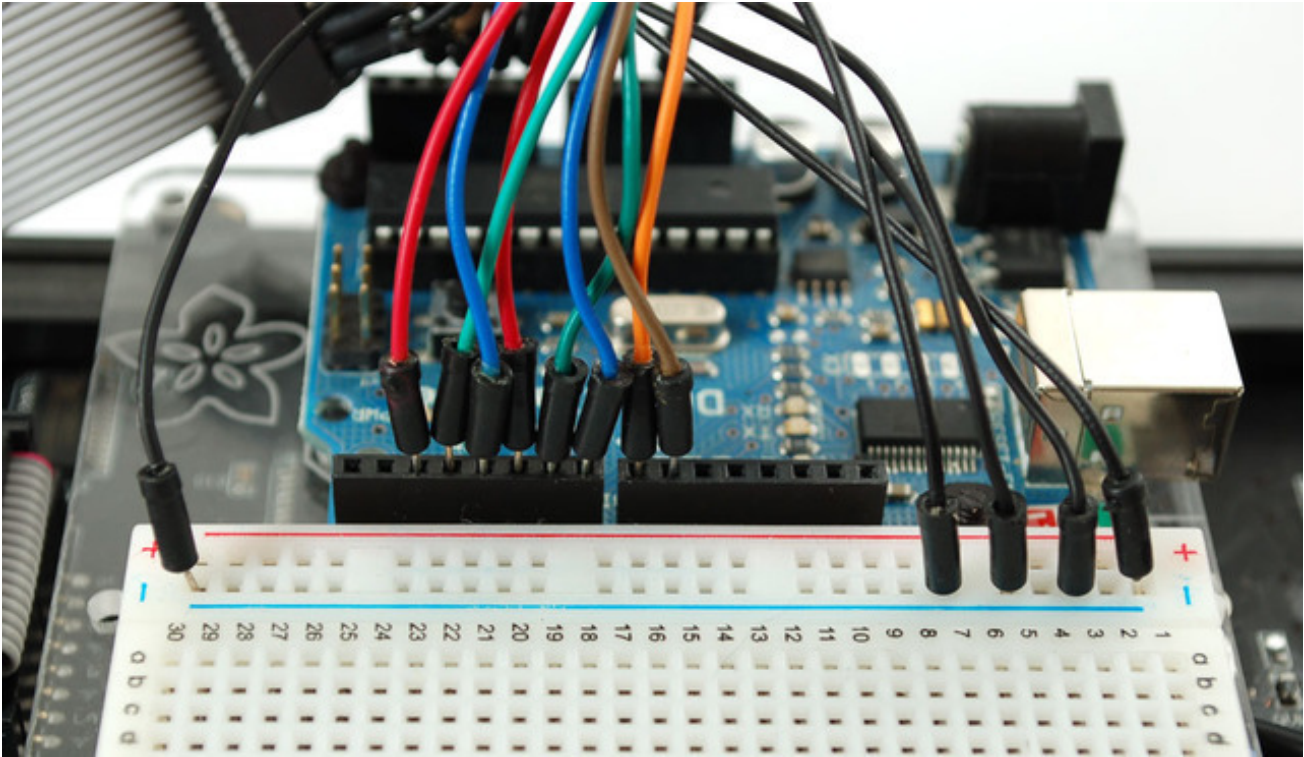
Now we can do the control pins. **A**, **B** and **C** are the row address pins, for multiplexing the display. Those are white wires. **CLK** (clock) is orange, **OE** (output enable) is brown, and **LAT** (data latch) is yellow. The remaining ground pins are black wires again.



Finally, we can connect the wires directly to our Arduino (or other microcontroller). Connect:

- **R1** goes to **digital 2**
- **G1** goes to **digital 3**
- **B1** goes to **digital 4**
- **R2** goes to **digital 5**
- **G2** goes to **digital 6**
- **B2** goes to **digital 7**
- **CLK** (orange) goes to **digital 8**
- **OE** (brown) goes to **digital 9**

If using an **Arduino Mega 2560**, change the above to use **pins 24 through 29** (instead of pins 2 – 7), but following the same order. **Pin 11** should be used in place of pin 8. You'll then need to make a small change in each of the example sketches (explained in the "Test Example Code" section). All other wiring is the same.



Next we'll do the remaining control pins. Connect:

- **A** to **analog 0**
- **B** to **analog 1**
- **C** to **analog 2**
- **LAT** (yellow) to **analog 3**

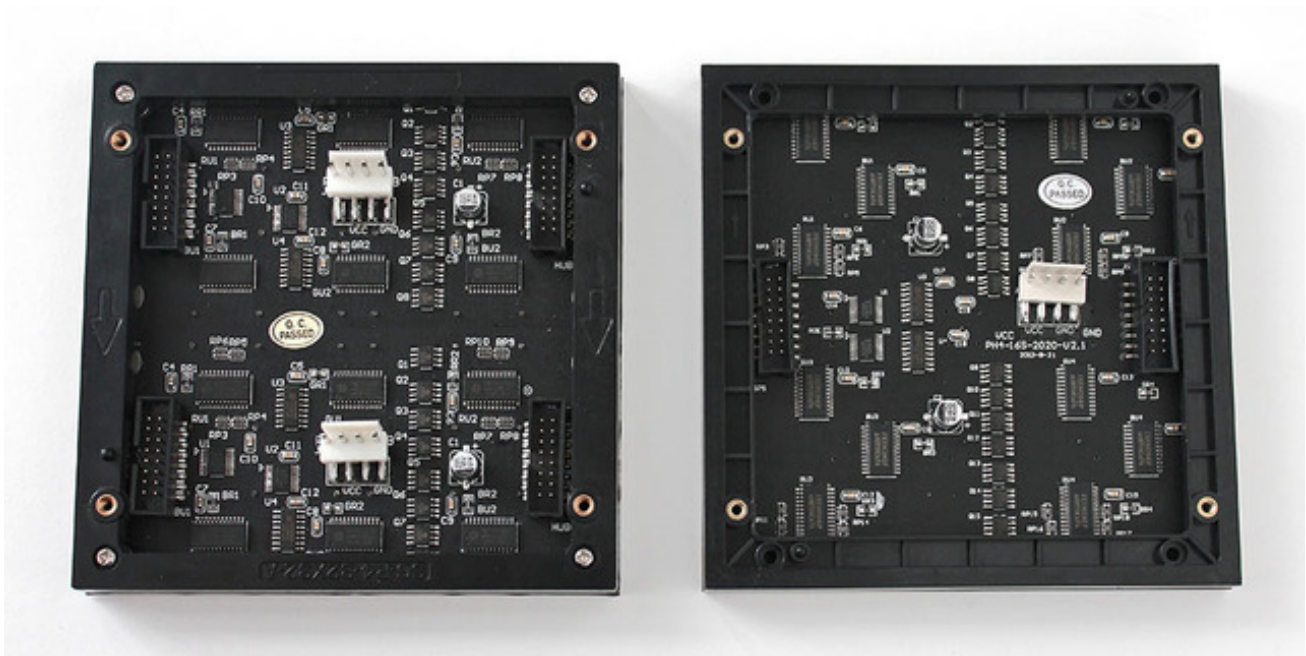
This leaves you with digital **10, 11, 12, 13** and analog **4, 5** to use for other sensors, switches, outputs, etc. (On the Arduino Mega, pin 11 is used, pin 8 is free.)

The multiple ground wires can be connected to the Arduino through one of the power buses on a breadboard.

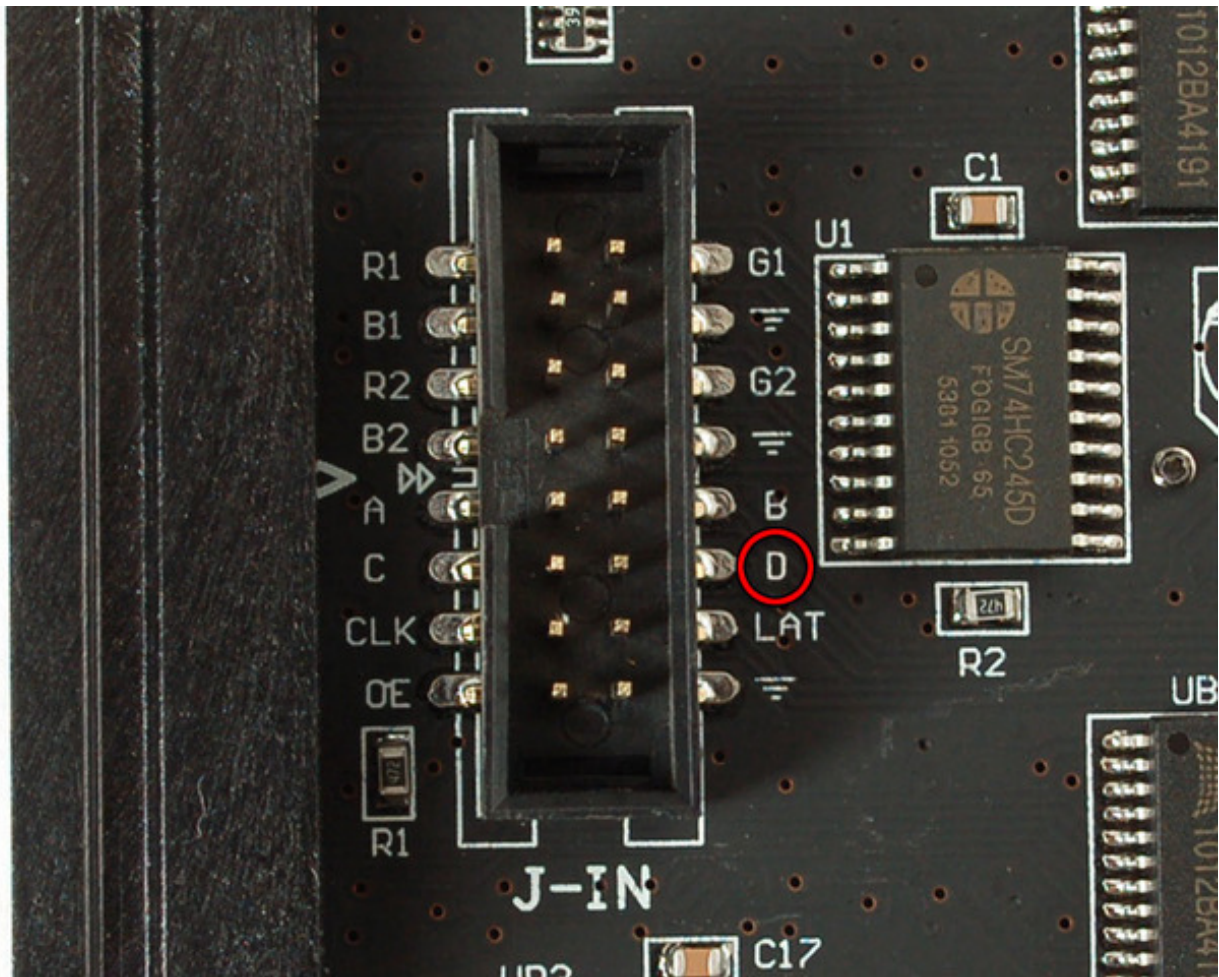
Now you're ready to test the matrix!

Wiring the 32x32 Matrix

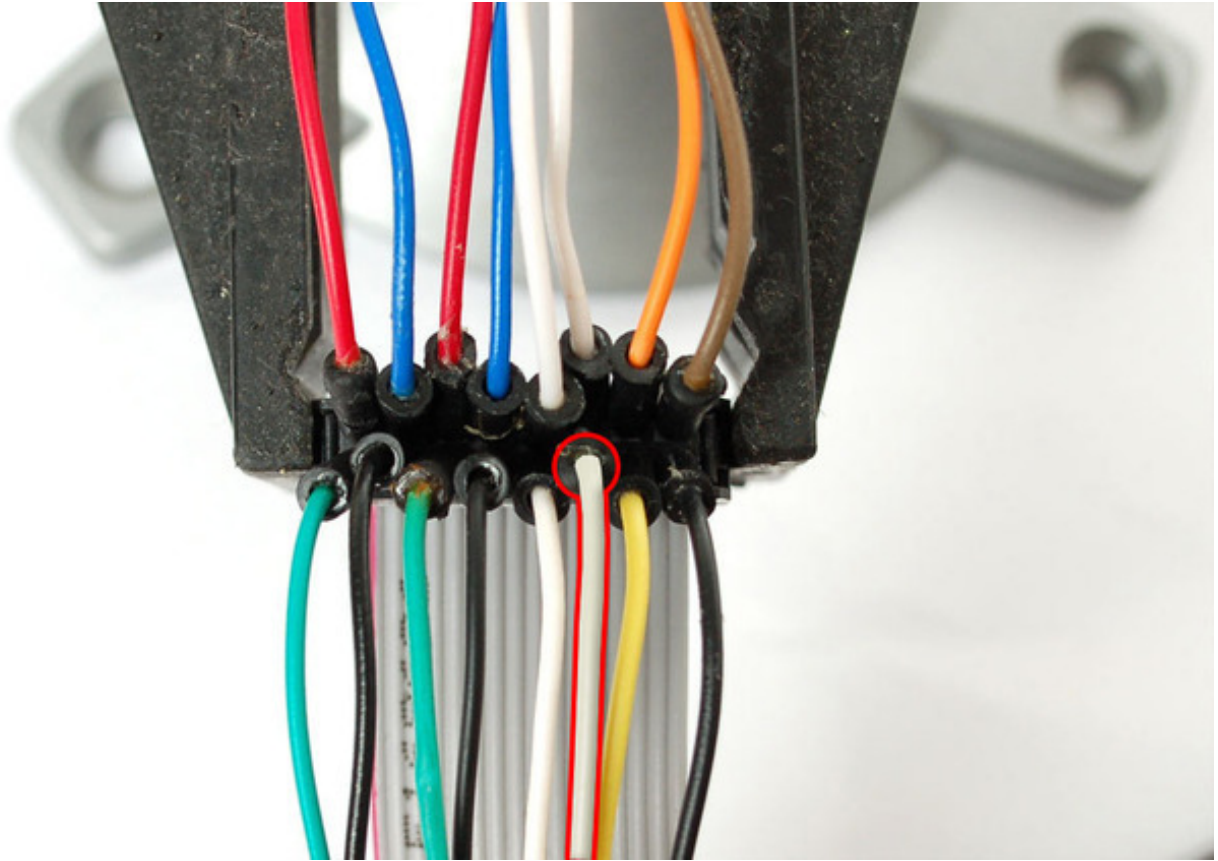
We've seen two variants of the 32x32 panel. One has dual 16-pin IDC headers for input, while the other has a single header (similar to the 16x32 panel). Both types have matching output headers as well...again, not used here.



We'll start off with the single-connector instructions, as they're a bit easier. In fact, it's **nearly identical to the 16x32 case**, with the exception of a single pin: between the 'B' and 'LAT' pins is an additional row address select, 'D':



So you can begin by wiring up the board using the prior 16x32 directions, but don't power it up quite yet. Two wires need to be switched. First, **move the yellow LAT wire** from **analog pin 3** to **digital pin 10**. Next, what was previously a ground wire is now address select D. We'll switch it out for a white wire for consistency, and connect the other end to Arduino pin **A3** (where LAT was previously located). This keeps analog pins 4 & 5 open, which is important if adding I2C peripherals.



Some variants of this panel have the green and blue pins swapped; you'll need to reverse the wiring on those four pins in that case.

That's it for the single-header panel...you can skip ahead to the next section.

Wiring a double-header panel

The double-header panel is a little tougher to wire up, because technically its two 16x32 panels that are stamped on a single board. Both inputs have this pinout:

GND	A
GND	B
GND	C
OE	D
RED	GREEN
BLUE	?

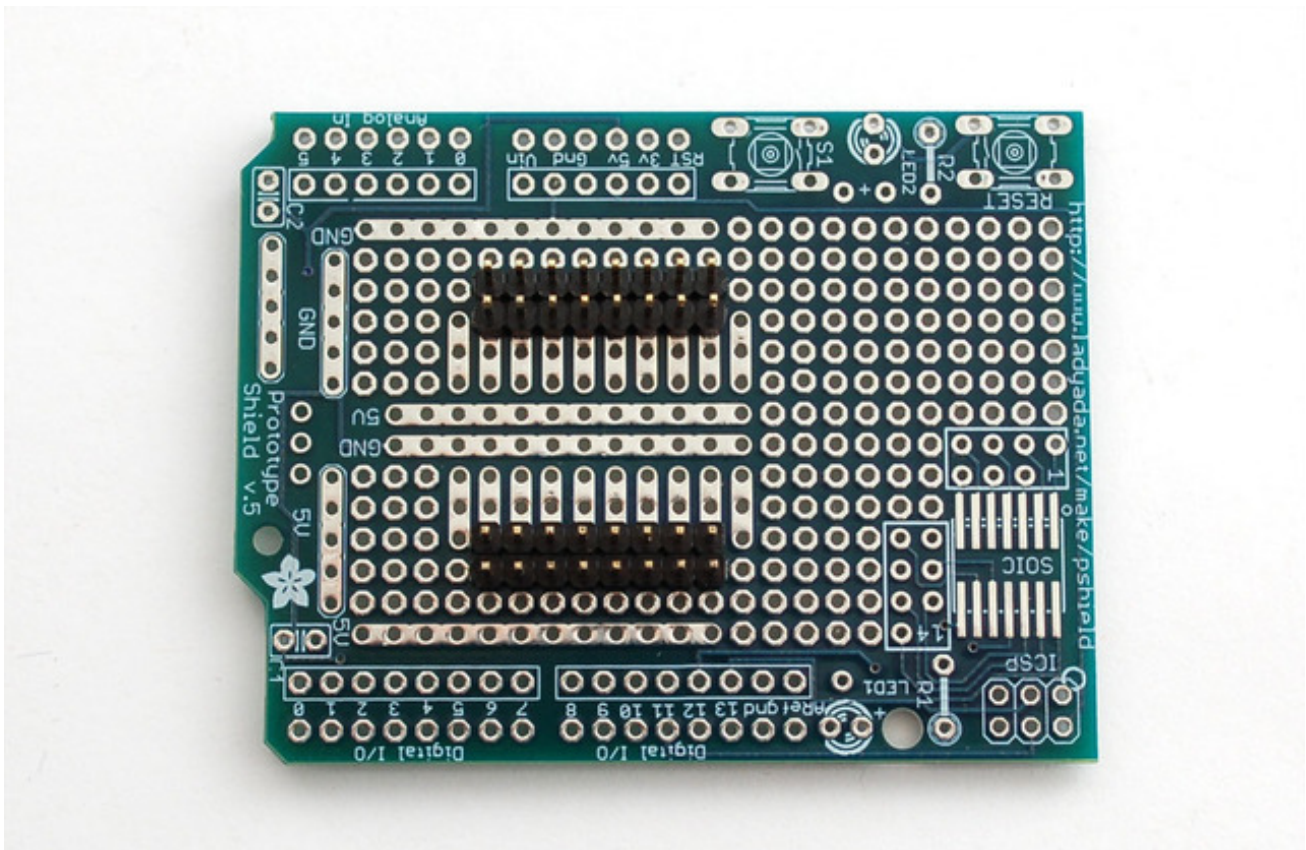
GND	LAT
GND	CLK

Contrast this to the prior pinout which has two red/green/blue sets. This panel also has the extra address pin D (which means you have $1:2^4 = 1:16$ refresh rate)

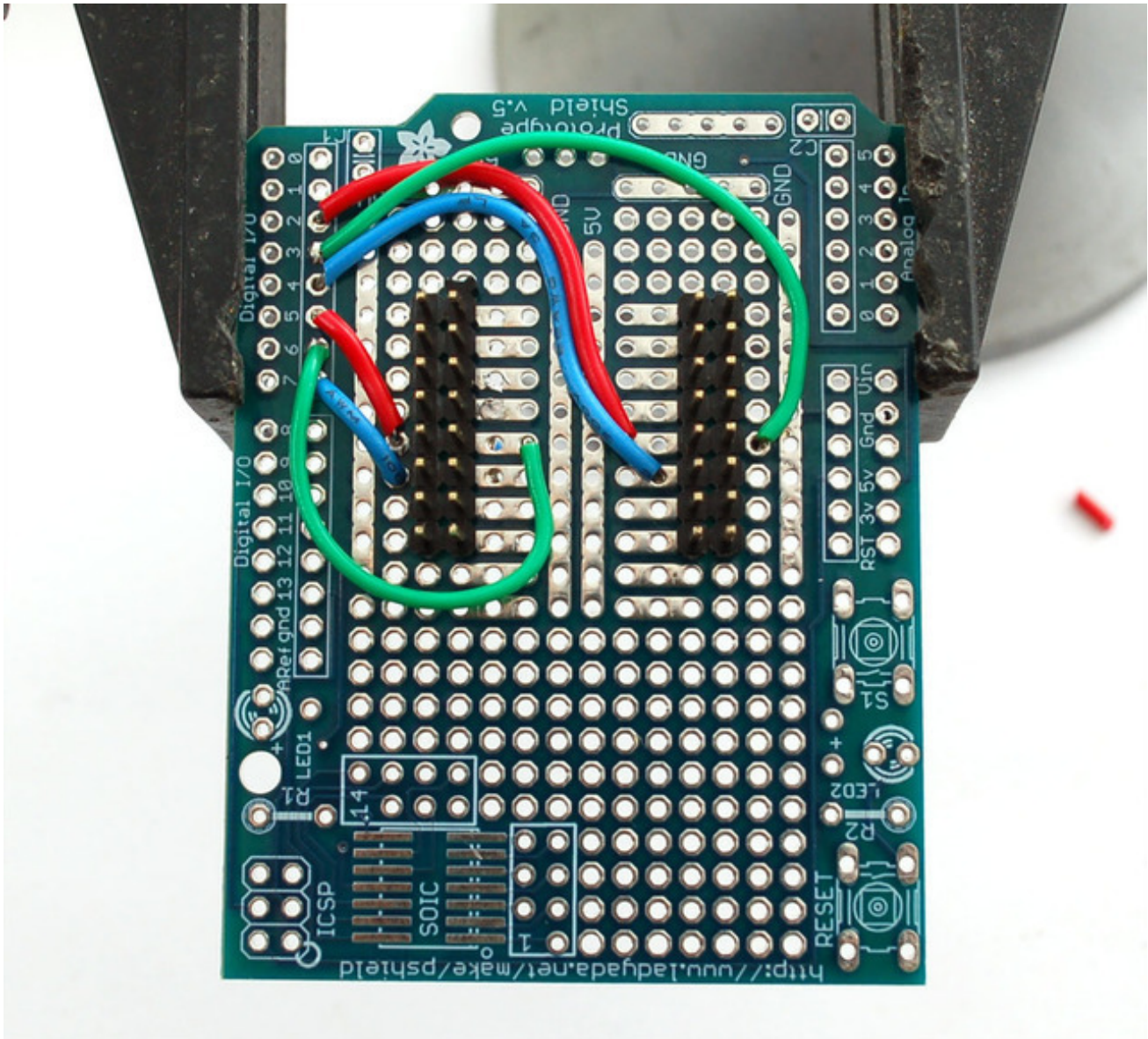
In order to make the panel as compatible as possible with our existing 16x32 panel, we'll do a trick where we connect the control pins **A, B, C, D, OE, LAT** and **CLK** together on both 'halves' of the panel. Then we'll write to both panels at the same time, by having two sets of **RED, GREEN** and **BLUE** pins. That way, it 'looks' like the 16x32 panel with 6 color lines and 6 data lines + 1 extra address pin.

Unfortunately, this makes wiring a little hairy because you have to tie the signal lines together. We suggest using a prototype board to have both IDCs connected together on the PCB. On a breadboard this would just be a huge rat's nest!

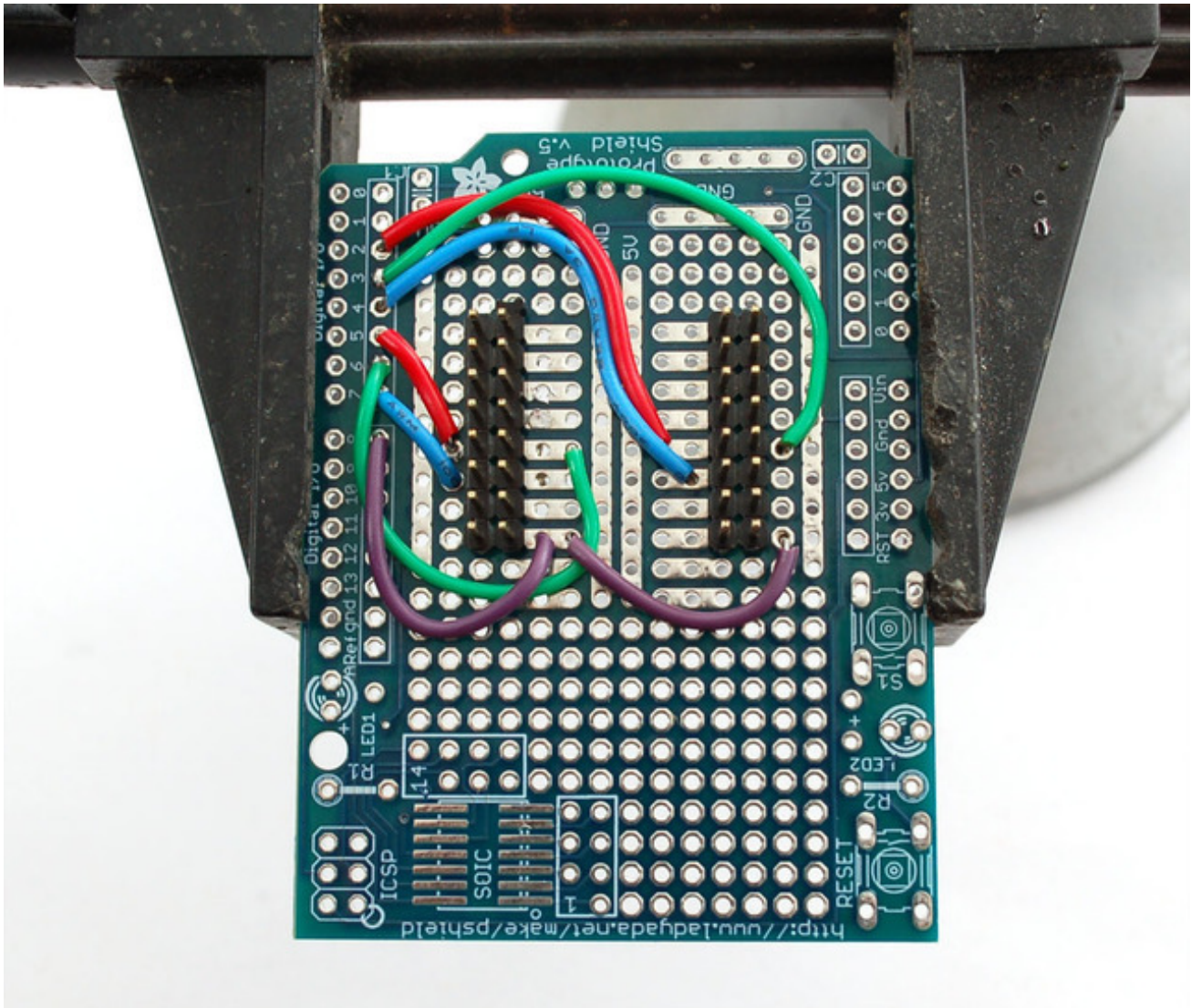
We'll use a protoshield, first place two sets of 2x8 header in the positions shown:



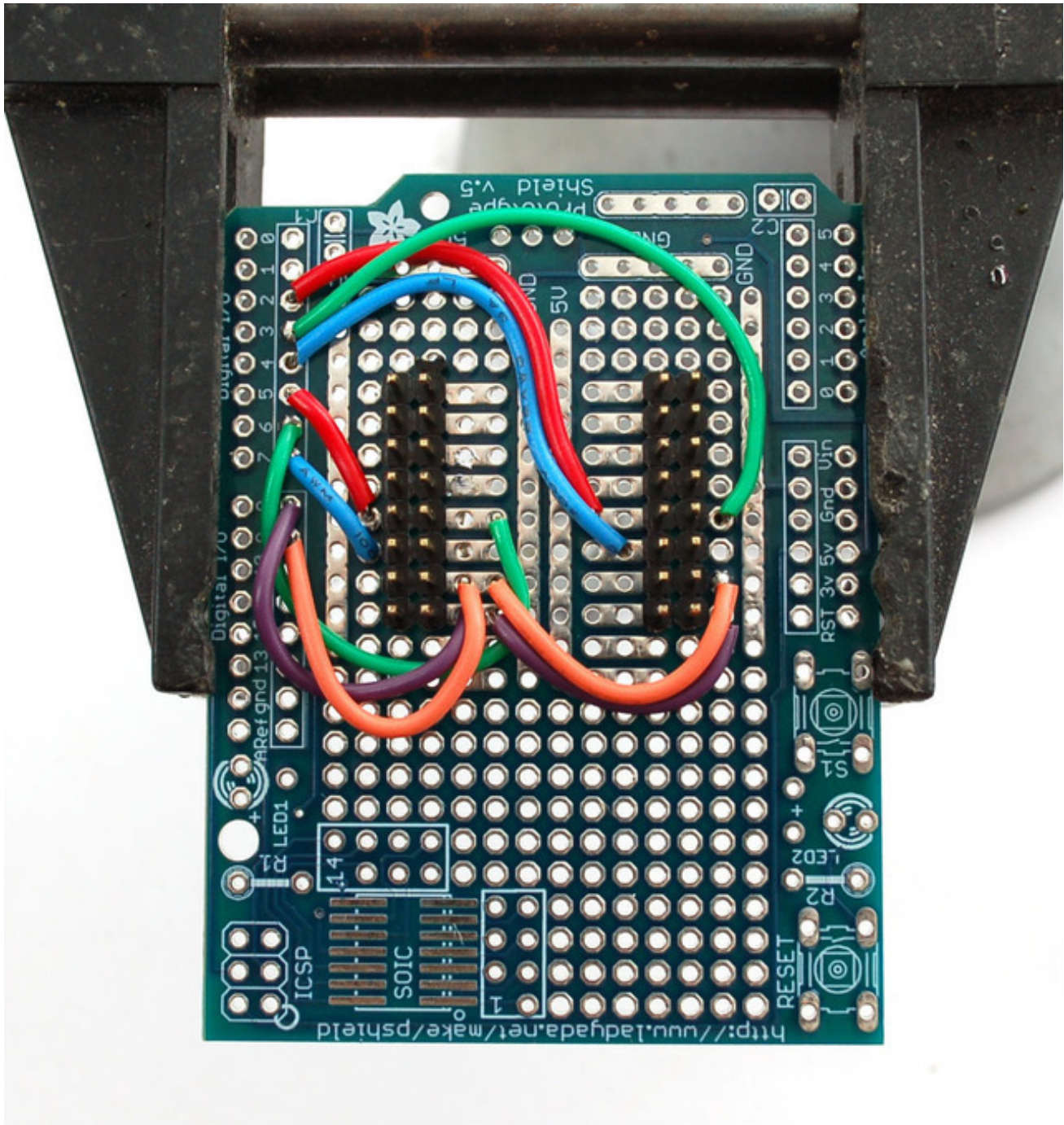
Start by connecting the two RGB pin sets to digital **2-7**



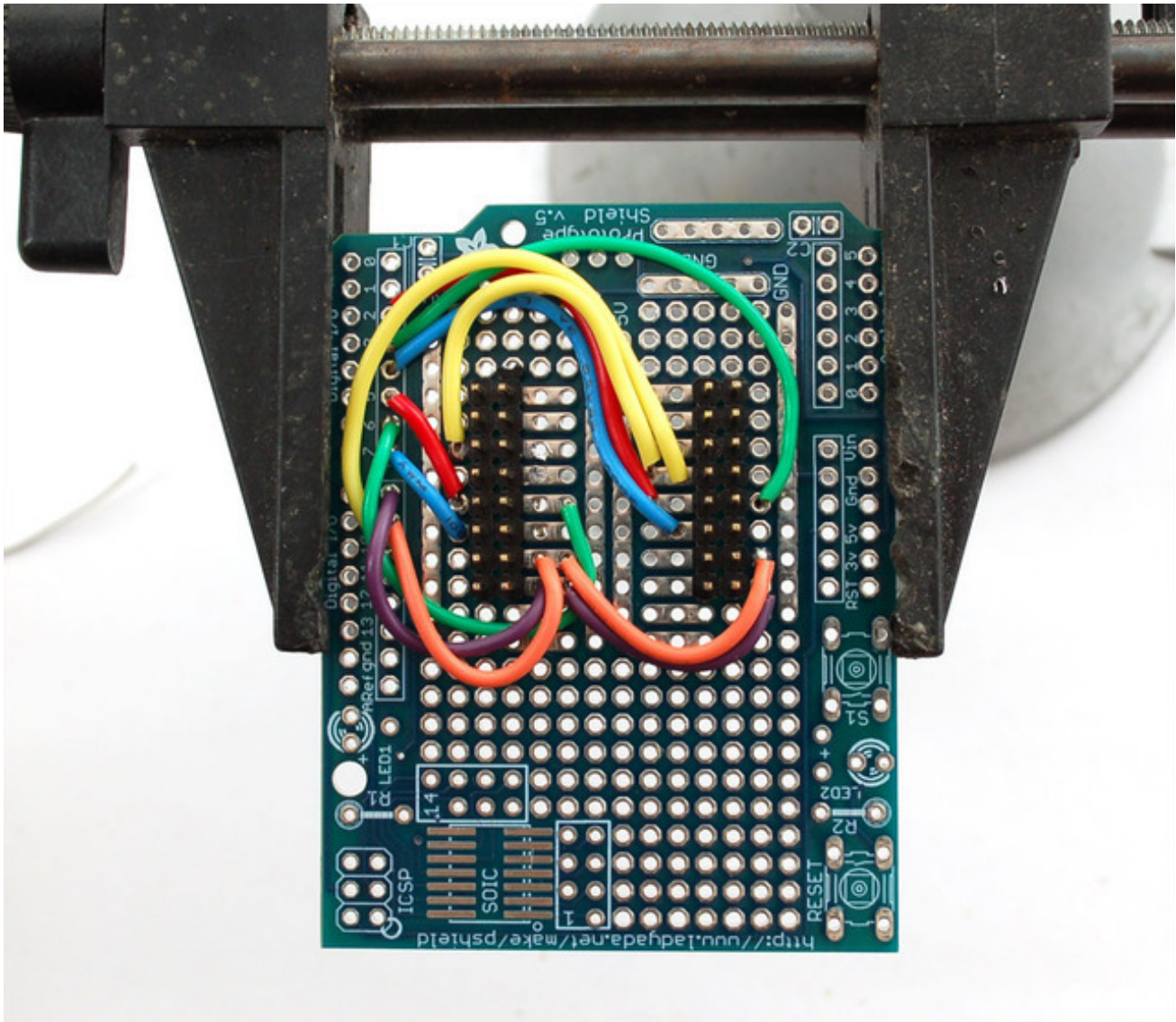
Then tie both **CLK** pins together, and to digital **8**



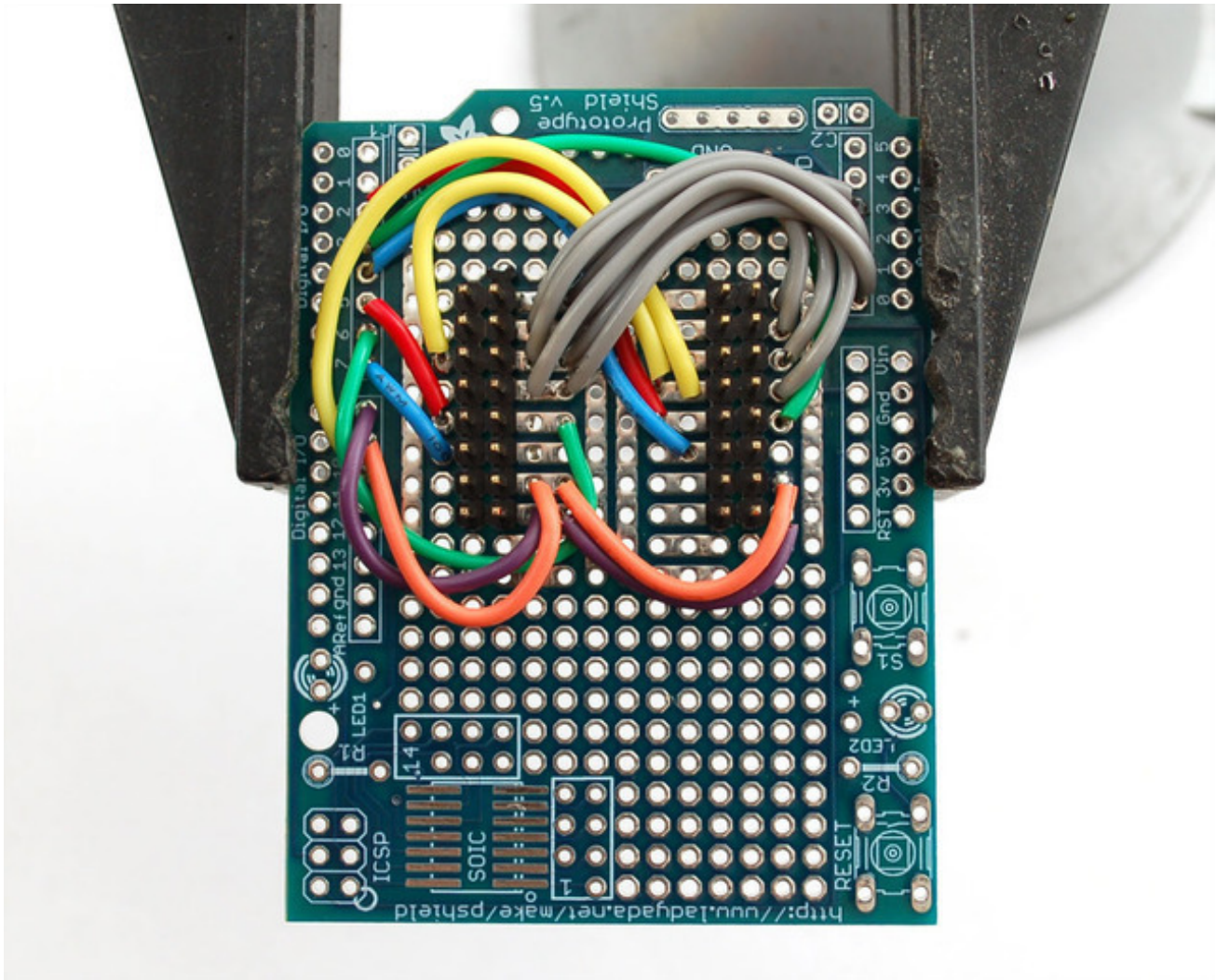
Tie both **LAT** pins together, and to digital **9**



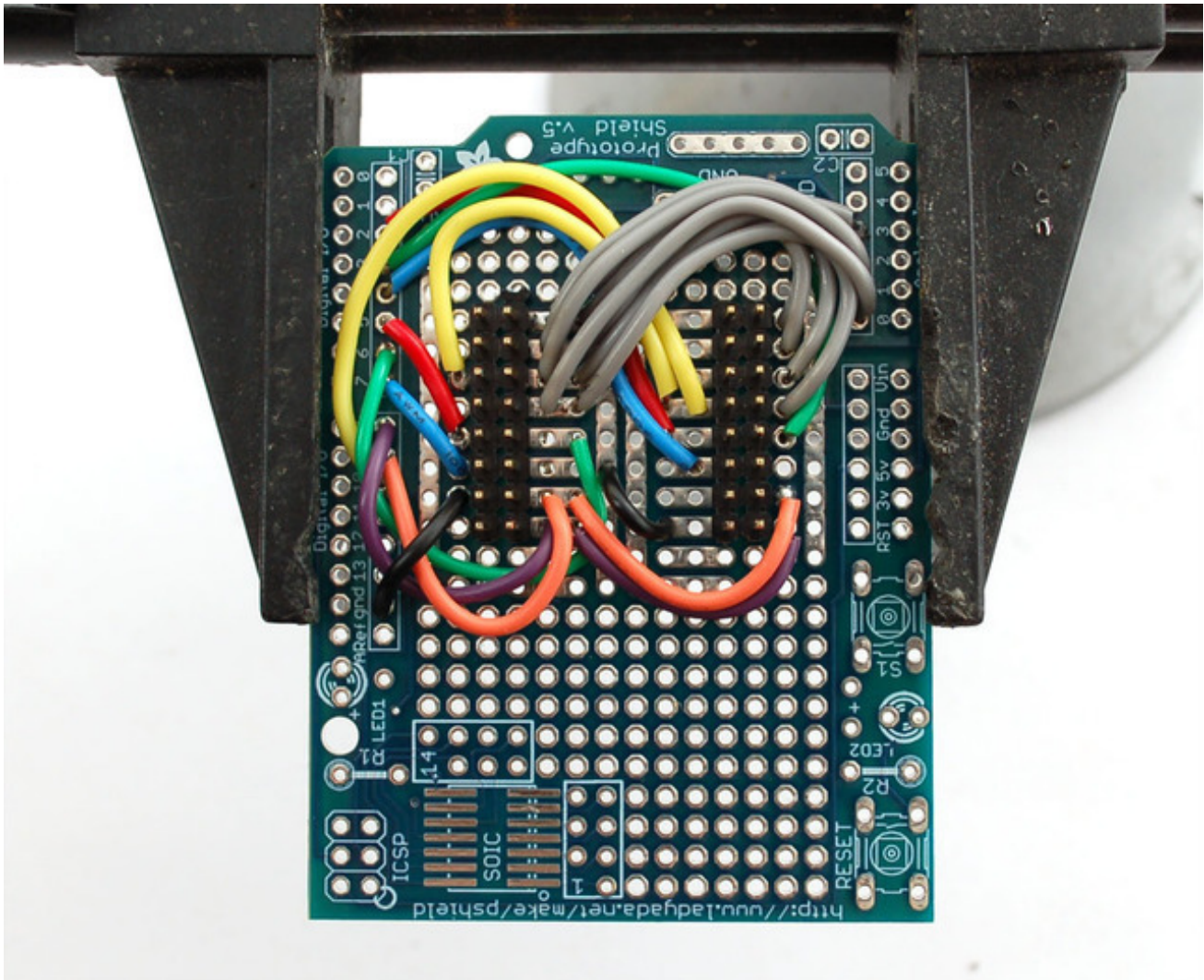
Tie both **OE** pins together, and to digital **10** (We accidentally soldered the left hand **OE** to the wrong hole, it should go right above **RED**)!



Tie all four **ADDR (A B C D)** pins together and to **analog 0-3**



Finally, connect one of the ground pins from each side to the common Arduino ground. The remaining GND pins are tied together on the RGB panel!



This leaves you with digital pins 11, 12, 13 and analog 4, 5 to use for other sensors, switches, outputs, etc.

Test Example Code

We have example code ready to go for these displays. It's compatible with the Arduino Uno or Mega...but **not** other boards like the Leonardo, nor “Arduino-like” boards such as Netduino...programming gurus *might* be able to port it to other microcontrollers by adapting the C++ source, but as written it does some pretty low-level, non-portable things.

The library works ONLY with the Arduino Uno and Mega. Other boards (such as the Arduino Leonardo) ARE NOT SUPPORTED.

Two libraries need to be downloaded and installed: first is the [RGB Matrix Panel library \(http://adafru.it/aHj\)](http://adafru.it/aHj) (this contains the low-level code specific to this device), and second is the [Adafruit GFX Library \(http://adafru.it/aJa\)](http://adafru.it/aJa) (which handles graphics operations common to many displays we carry). Download both ZIP files, uncompress and rename the folders to 'RGBmatrixPanel' and 'Adafruit_GFX' respectively, place them inside your Arduino libraries folder and restart the Arduino IDE. If this is all unfamiliar, we have a [tutorial introducing Arduino library concepts and installation \(http://adafru.it/aYG\)](http://adafru.it/aYG).

Now you are ready to test! Open up the IDE and load

File→Examples→RGBmatrixPanel→testcolors_16x32 (for the 16x32 panel) or

File→Examples→RGBmatrixPanel→colorwheel_32x32 (for the 32x32 panel).

If you are using the 32x32 panel, before you upload this code to the Arduino, edit the pin definitions to match the specific wiring used by your panel (single- or double-header interface). Comments in the file will direct you to what needs changed (if anything).

If using an Arduino Mega 2560, in addition to wiring changes previously mentioned, you'll need to make a small change to each of the example sketches. This line:

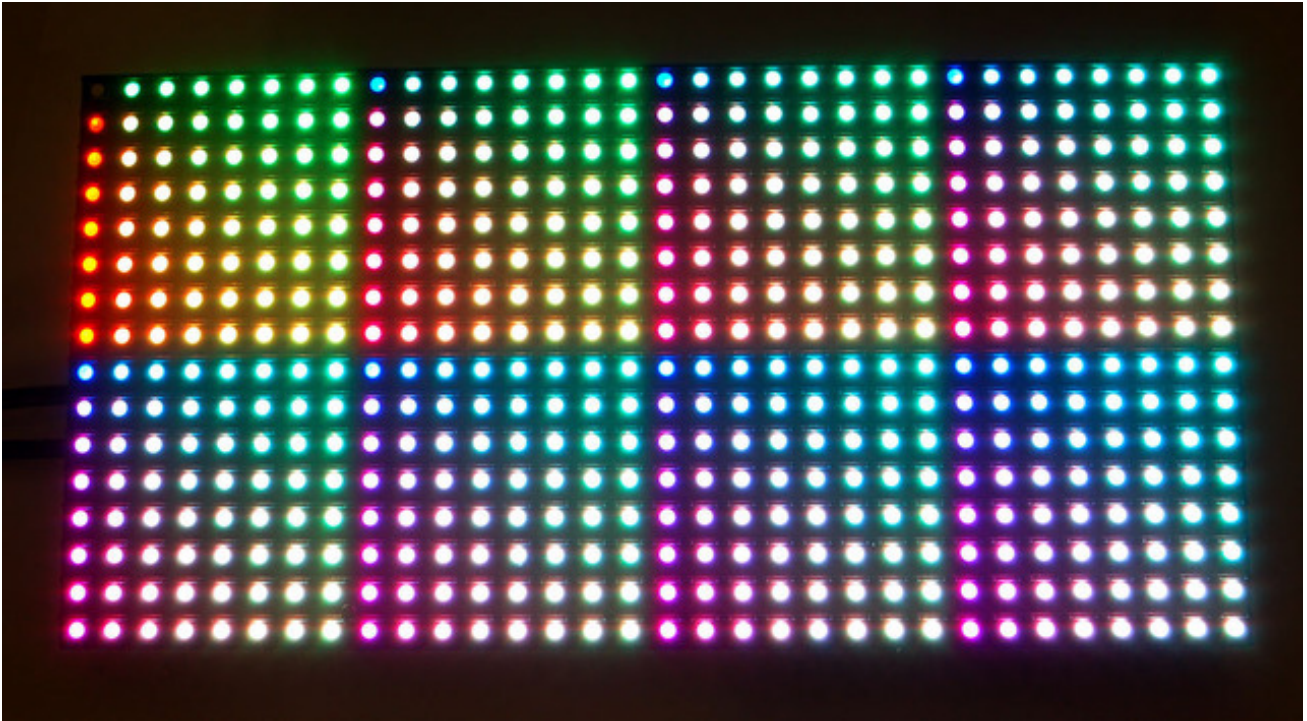
```
#define CLK 8 // MUST be on PORTB! (Use pin 11 on Mega)
```

Should be changed to:

```
#define CLK 11
```

(Any of digital pins 10-13 and 50-53 can be used for this function on the Mega, with the corresponding wiring change. The examples all reference pin 11, as pin 10 may be in use for the 32x32 panel.)

After uploading, with the 16x32 panel you should see the following:



This is a test pattern that shows 512 colors (out of 4096) on the 512 pixels. Since there's no really elegant way to show a 3-dimensional color space (R/G/B) in two dimensions, there's just repeating grids of red/green with increasing blue. Anyways, this shows you the range of colors you can achieve!

or, with the 32x32 panel:



Now that you've got it working here are a few things to look for:

The most useful line to look at is:

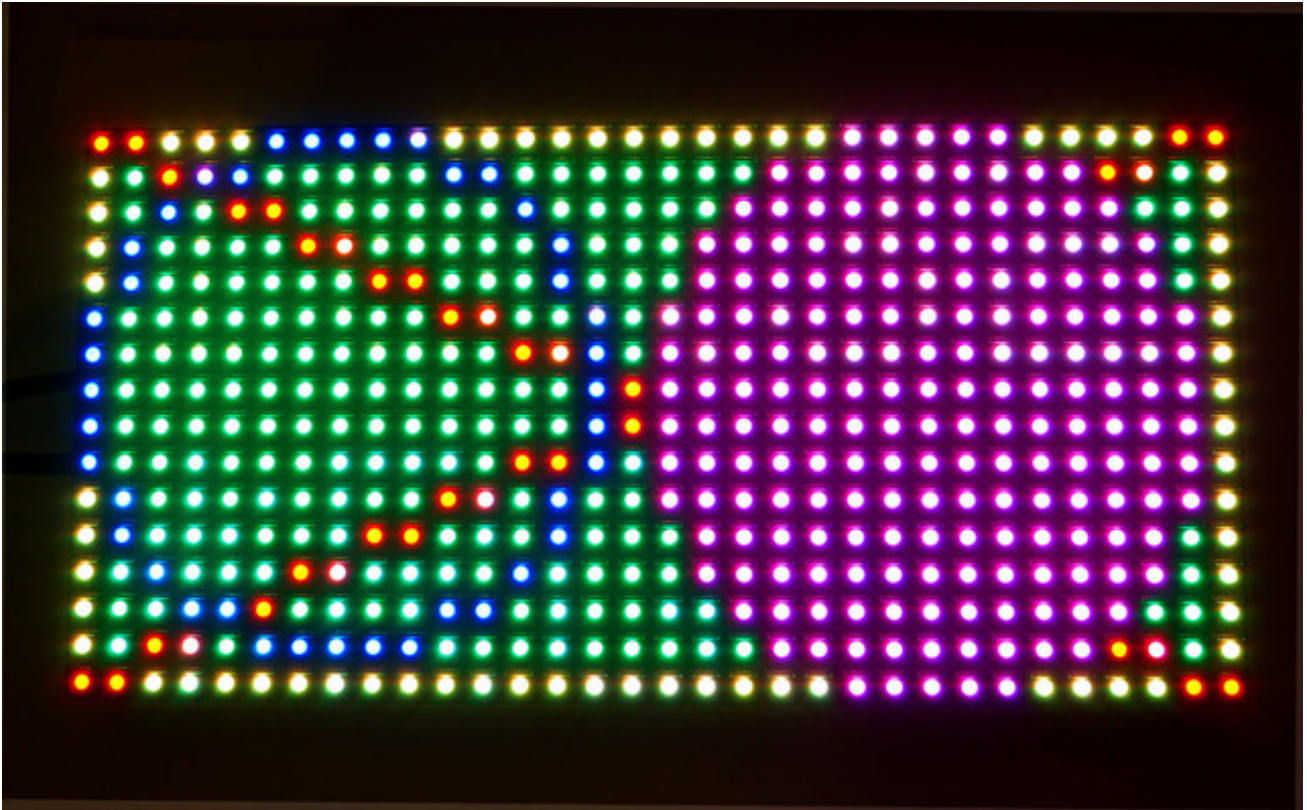
```
matrix.drawPixel(x, y, matrix.Color333(r, g, b));
```

which is where we actually draw to the display. This code only draws one pixel at a time. The **x** and **y** coordinates are the individual pixels of the display. **(0,0)** is in the top left corner, **(31, 15)** is in the bottom right (remember that we start counting at 0 here!). To create a color, you will want to use the helper function **Color333** which will take three 3-bit numbers and combine them into a single packed integer. So for example, the first argument, **r** can range from 0 to 7. Likewise for **g** and **b**. To make a pixel that is pure red, **r** would be 7 and **g**, **b** would be 0. To make a white pixel, set all to 7. To make a black (off) pixel, set the colors to 0. A similar function, **Color444**, accepts three 4-bit numbers for up to 4096 colors.

Now we can open up the next example, which shows the rest of the library capabilities.

Library

Next up, load the **testshapes_16x32** or **testshapes_32x32** example sketch, which will test every drawing element available (again, you may need to edit the pin numbers for the 32x32 panel).



The most simple thing you may want to do is draw a single pixel, we saw this introduced above.

```
// draw a pixel in solid white
matrix.drawPixel(0, 0, matrix.Color333(7, 7, 7));
```

Next we will fill the screen with green by drawing a really large rectangle. The first two arguments are the top left point, then the width in pixels, and the height in pixels, finally the color

```
// fix the screen with green
matrix.fillRect(0, 0, 32, 16, matrix.Color333(0, 7, 0));
```

Next we will draw just the outline of a rectangle, in yellow


```
// draw a box in yellow
matrix.drawRect(0, 0, 32, 16, matrix.Color333(7, 7, 0));
```

Next you may want to draw lines. The **drawLine** procedure will draw a line in any color you want, we used this to draw a big X

```
// draw an 'X' in red
matrix.drawLine(0, 0, 31, 15, matrix.Color333(7, 0, 0));
matrix.drawLine(31, 0, 0, 15, matrix.Color333(7, 0, 0));
```

The next shapes we draw are circles. You can draw the outline of a circle with **drawCircle** or fill a circle with **fillCircle**. The first two arguments are the center point, the third argument is the radius in pixels, finally the color to use.

```
// draw a blue circle
matrix.drawCircle(7, 7, 7, matrix.Color333(0, 0, 7));

// fill a violet circle
matrix.fillCircle(23, 7, 7, matrix.Color333(7, 0, 7));
```

fill allows you to fill the entire screen with a single color

```
// fill the screen with 'black'
matrix.fill(matrix.Color333(0, 0, 0));
```

Finally, we draw the text that is shown up top as the demonstration image. We can use the **print** function, which you'll be familiar with from **Serial**. You can use **print** to print strings, numbers, variables, etc. However, we need to set up the printing before just going off and doing it! First, we must set the cursor location with **setCursor** which is where the top left pixel of the first character will go, this can be anywhere but note that text characters are 8 pixels high by default. Next **setTextSize** lets you set the size to 1 (8 pixel high) or 2 (16 pixel high for really big text!), you probably want just to stick with 1 for now. Lastly we can set the color of the text with **setTextColor**. Once this is all done, we can just use **print('1')** to print the character "1".

```
// draw some text!
matrix.setCursor(1, 0); // start at top left, with one pixel of spacing
matrix.setTextSize(1); // size 1 == 8 pixels high

// print each letter with a rainbow color
matrix.setTextColor(matrix.Color333(7,0,0));
matrix.print('1');
```

```

matrix.setTextColor(matrix.Color333(7,4,0));
matrix.print('6');
matrix.setTextColor(matrix.Color333(7,7,0));
matrix.print('x');
matrix.setTextColor(matrix.Color333(4,7,0));
matrix.print('3');
matrix.setTextColor(matrix.Color333(0,7,0));
matrix.print('2');

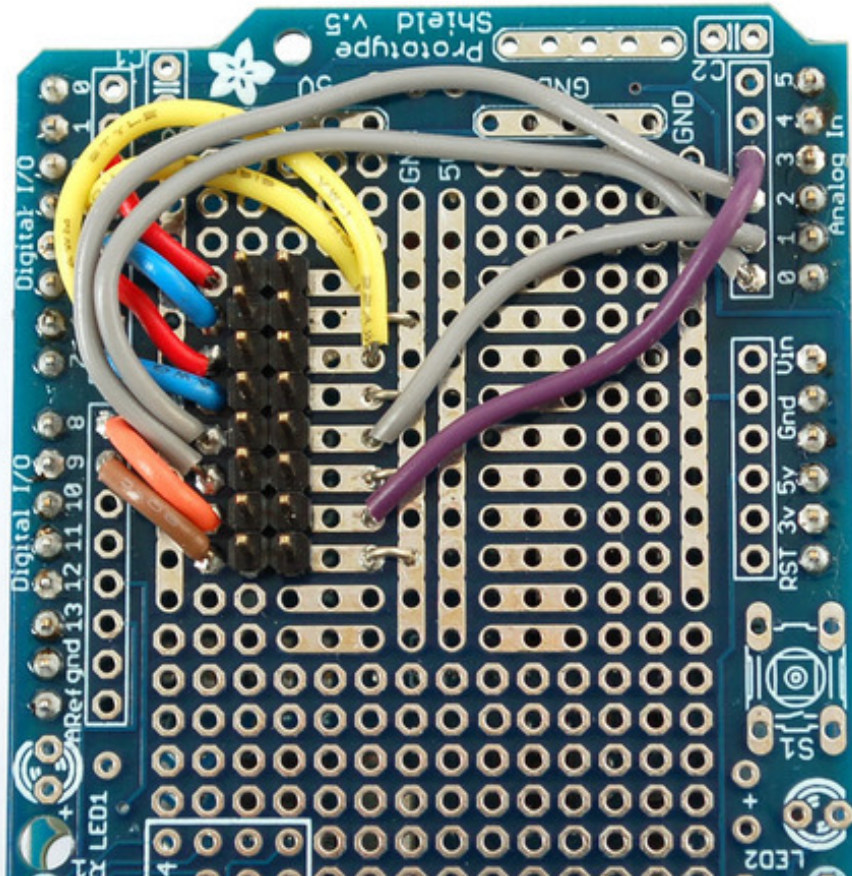
matrix.setCursor(1, 9); // next line
matrix.setTextColor(matrix.Color333(0,7,7));
matrix.print('*');
matrix.setTextColor(matrix.Color333(0,4,7));
matrix.print('R');
matrix.setTextColor(matrix.Color333(0,0,7));
matrix.print('G');
matrix.setTextColor(matrix.Color333(4,0,7));
matrix.print("B");
matrix.setTextColor(matrix.Color333(7,0,4));
matrix.print("*");

```

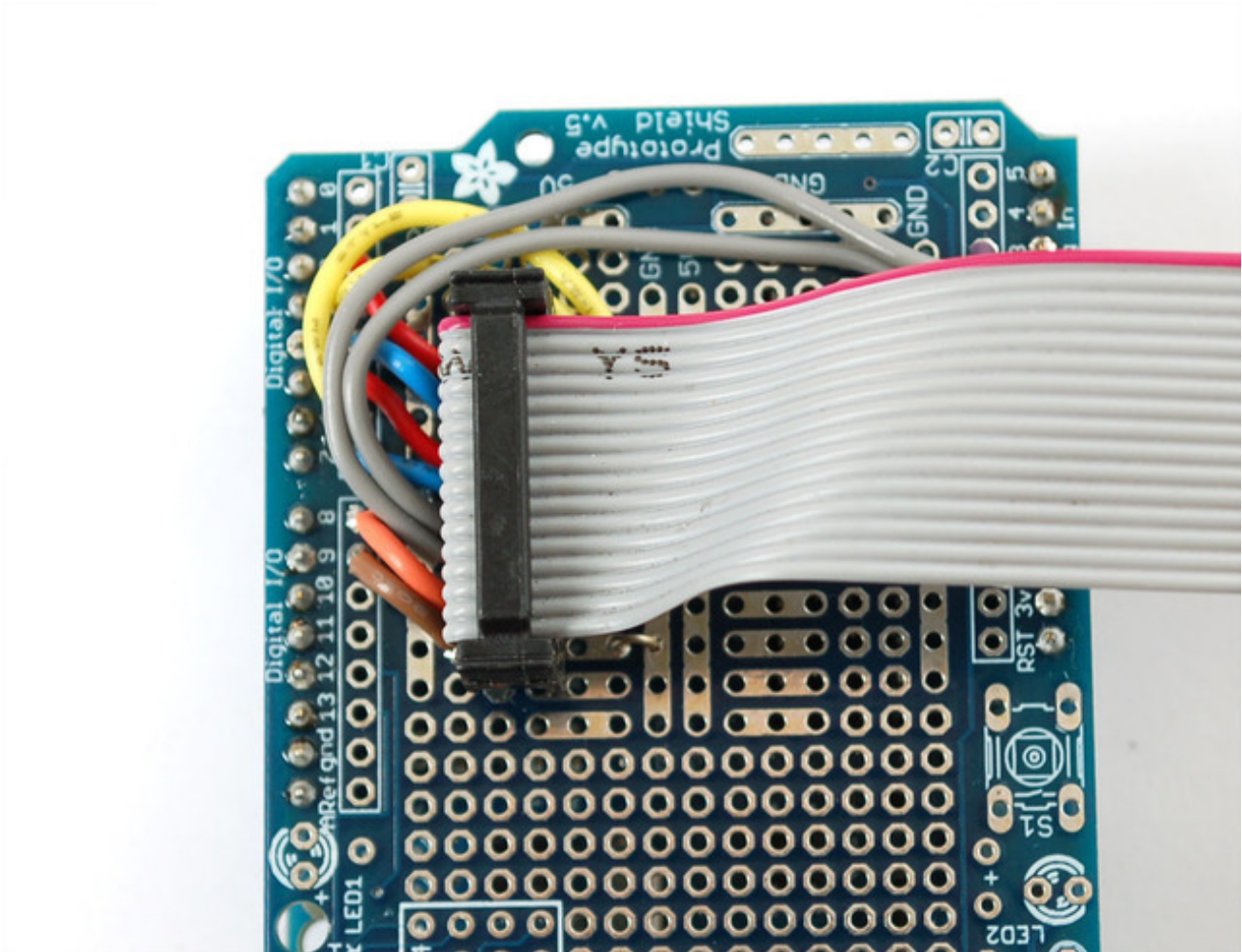


Advanced Wiring for the 16x32

Once you've tested the 16x32 panel, we suggest making a more permanent solution. We used a proto shield and some header to make a custom-wired shield that is easy to use. You can see that now because we're looking **down** at the plug, the wiring matches the diagram from the back of the panel, where **R1** is in the top left corner.



Match the red stripe of the cable to **R1**



We can build a similar proto shield for the single-header 32x32 panel. There will be one less ground connection (immediately above the left end of the purple wire) – instead this connects to analog pin 3. The right end of the purple wire should instead come around to digital pin 10.

How the Matrix Works

There's zero documentation out there on how these matrices work, and no public datasheets or spec sheets so we are going to try to document how they work.

First thing to notice is that there are 512 RGB LEDs in a 16x32 matrix. Like pretty much every matrix out there, **you can't drive all 512 at once**. One reason is that would require a lot of current, another reason is that it would be really expensive to have so many pins. Instead, the matrix is divided into 8 interleaved sections/strips. The first section is the 1st 'line' and the 9th 'line' (32 x 2 RGB LEDs = 64 RGB LEDs), the second is the 2nd and 10th line, etc until the last section which is the 7th and 16th line. You might be asking, why are the lines paired this way? wouldn't it be nicer to have the first section be the 1st and 2nd line, then 3rd and 4th, until the 15th and 16th? The reason they do it this way is so that the lines are interleaved and look better when refreshed, otherwise we'd see the stripes more clearly.

So, on the PCB is 12 LED driver chips. These are like 74HC595s but they have 16 outputs and they are constant current. 16 outputs * 12 chips = 192 LEDs that can be controlled at once, and 64 * 3 (R G and B) = 192. So now the design comes together: You have 192 outputs that can control one line at a time, with each of 192 R, G and B LEDs either on or off. The controller (say an FPGA or microcontroller) selects which section to currently draw (using A, B, and C address pins - 3 bits can have 8 values). Once the address is set, the controller clocks out 192 bits of data (24 bytes) and latches it. Then it increments the address and clocks out another 192 bits, etc until it gets to address #7, then it sets the address back to #0

The only downside of this technique is that despite being very simple and fast, it has **no PWM control** built in! The controller can **only** set the LEDs **on or off**. So what do you do when you want full color? You actually need to draw the entire matrix over and over again at very high speeds to PWM the matrix manually. For that reason, you need to have a very fast controller (50 MHz is a minimum) if you want to do a lot of colors and motion video and have it look good.

How quickly can we feed data to the matrix? Forum users Andrew Silverman and Ryan Brown have been [posting their progress \(http://adafru.it/aO2\)](http://adafru.it/aO2) driving the 16x32 matrix with an FPGA, and the limit appears to be somewhere between 40 and 50 MHz. Ryan writes: "I haven't validated 100% pixel correctness, but 50 MHz seems to work for me [...] 67MHz definitely did not work." He also provided this graph showing current draw relative to clock frequency:

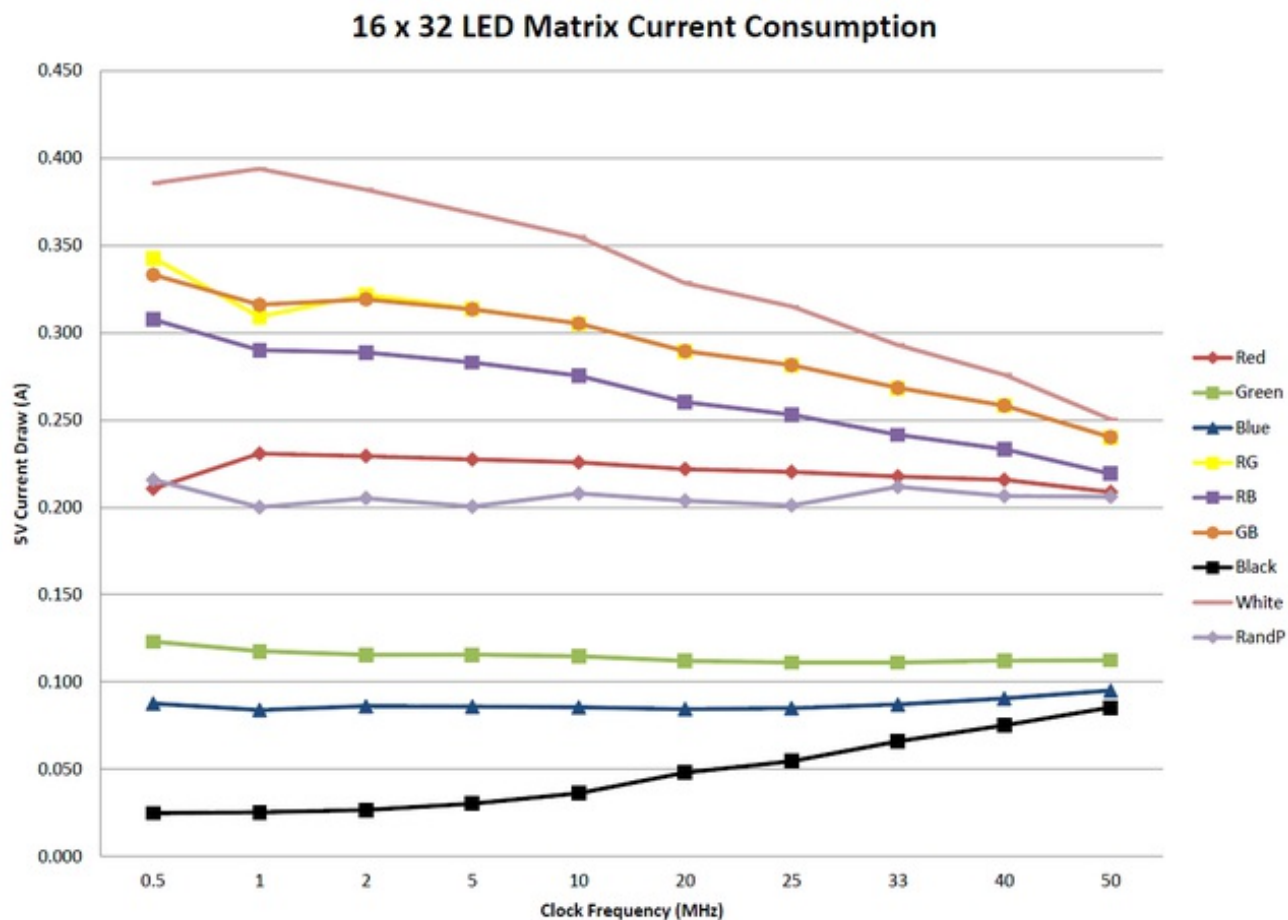


Image above by rhb.me (CC By-NC-SA)

“Notice that the LED panel current consumption decreases as clock frequency increases. This suggests that the LED ‘on time’ is decreasing. I’m guessing this is caused by frequency-invariant delays in the LED driver shift registers.”

Downloads

Download our **RGBmatrixPanel** library (<http://adafru.it/aHj>) by clicking the **ZIP** button near the top left corner, rename the uncompressed folder **RGBmatrixPanel**. Check that the **RGBmatrixPanel** folder contains **RGBmatrixPanel.cpp** and **RGBmatrixPanel.h**. Similarly, [download the Adafruit_GFX library here \(http://adafru.it/aja\)](http://adafru.it/aja) . Rename the uncompressed folder **Adafruit_GFX** and confirm it contains **Adafruit_GFX.cpp** and **Adafruit_GFX.h**. Place both library folders inside your **<arduinofolder>/libraries/** folder. You may need to create the libraries subfolder if its your first library. Restart the IDE.